# SLEEC: Semantics-rich Libraries for Effective Exascale Computation

X-Stack Kickoff Meeting
September 18, 2012

## Team

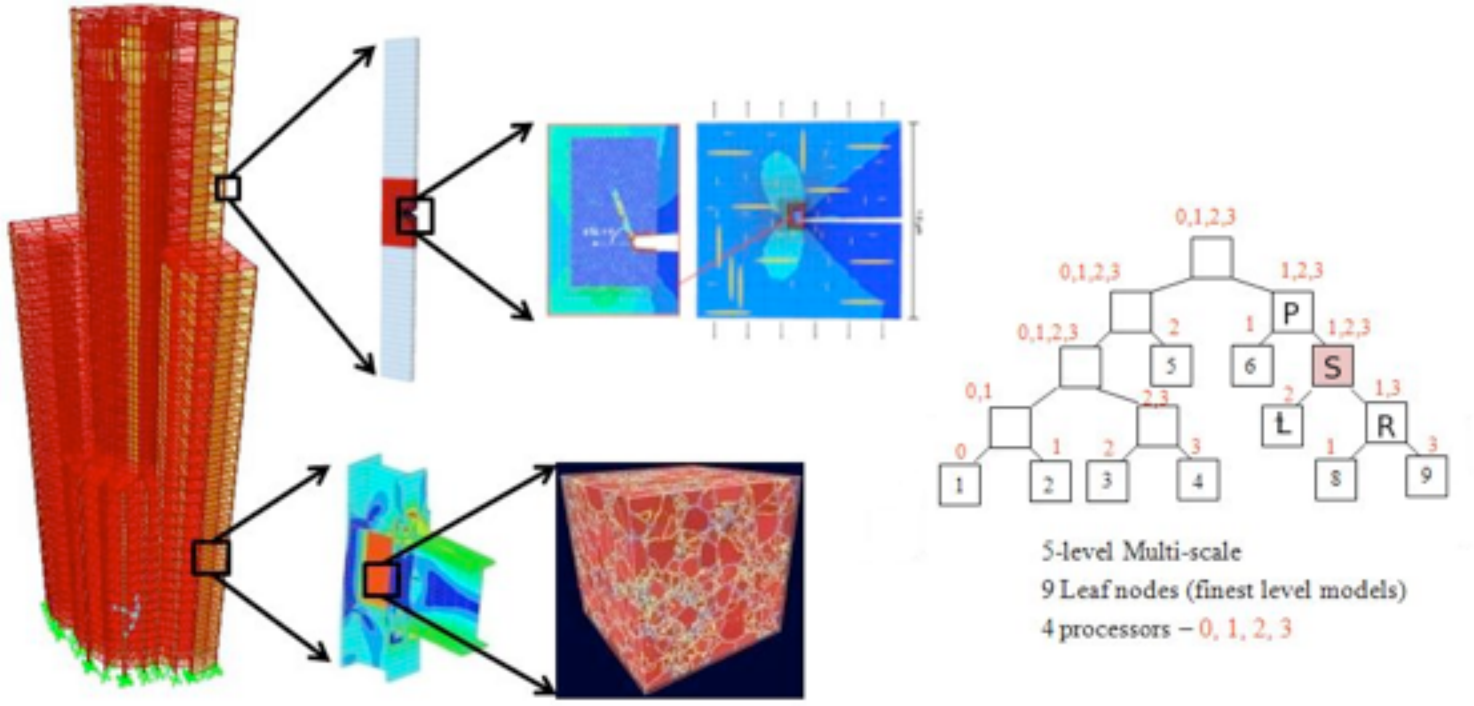
- Purdue University
  - Milind Kulkarni
  - Arun Prakash
  - Vijay Pai
  - Sam Midkiff
- Sandia National Labs
  - Michael Parks

## Motivation

- Modern computational science applications composed of many different libraries
  - Computational libraries, communication libraries, data structure libraries, etc.
  - Peridigm, developed by co-PI Mike Parks, builds on 10 different Trilinos libraries
- Each library has its own idioms and expected usage
- Determining right way to compose and use libraries to solve a problem is difficult

# Motivation: Compositional complexity

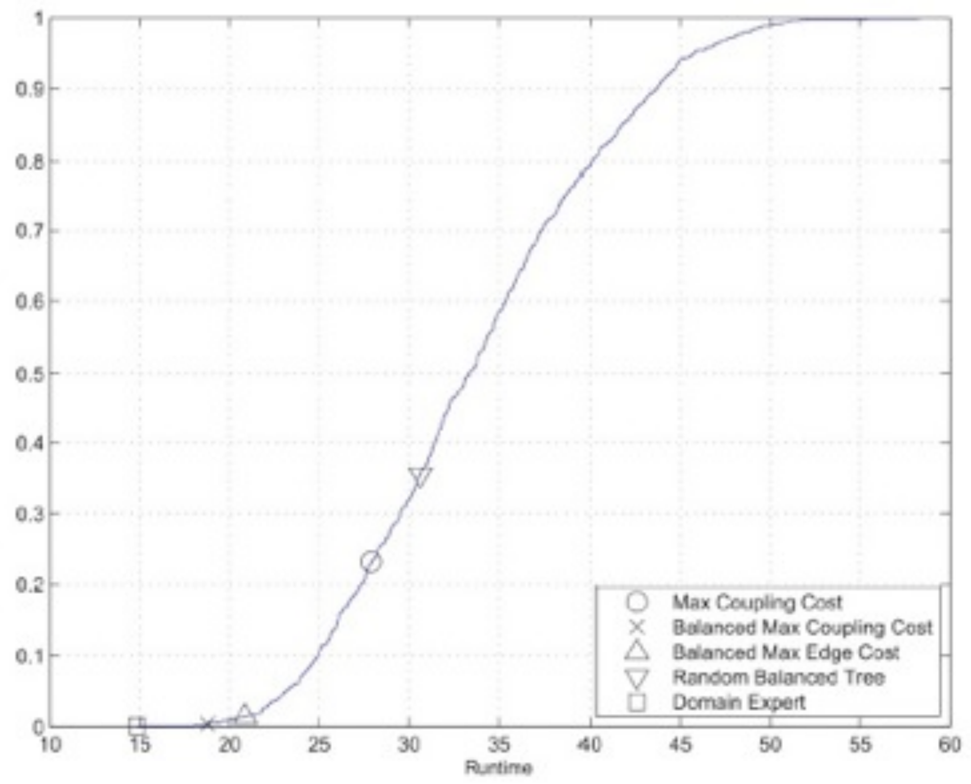
- Consider loosely-coupled multi-scale computational mechanics problem (developed by co-PI Arun Prakash)
- Must determine right way to decompose problem, couple separate solutions, etc.



5-level Multi-scale
9 Leaf nodes (finest level models)
4 processors – 0, 1, 2, 3

# Motivation: Compositional complexity

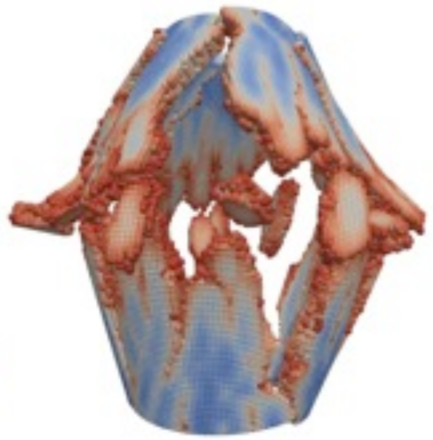
- Simple case: fixed number of subdomains, only consider how to couple them together
- Vast space of configurations: 8 subdomains → 135K possible schedules
- Large variation in performance of different orders
- Exploration of different variants requires knowledge of domain semantics, cost estimates



## Motivation: Difficult interaction between libraries

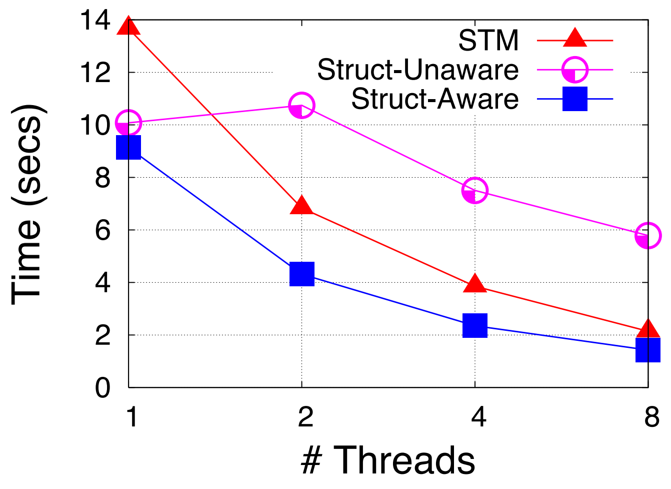
- Peridigm: computational peridynamics code
  - Allows modeling of materials under stress without explicit accounting for discontinuities (fractures, etc.)
- Built on Trilinos components
  - Set of computation and communication libraries
- Requires careful coordination of data movement operations to manage shadow data, etc. needed by solvers
  - But data movement requirements can be directly inferred from which equations are being solved

Before

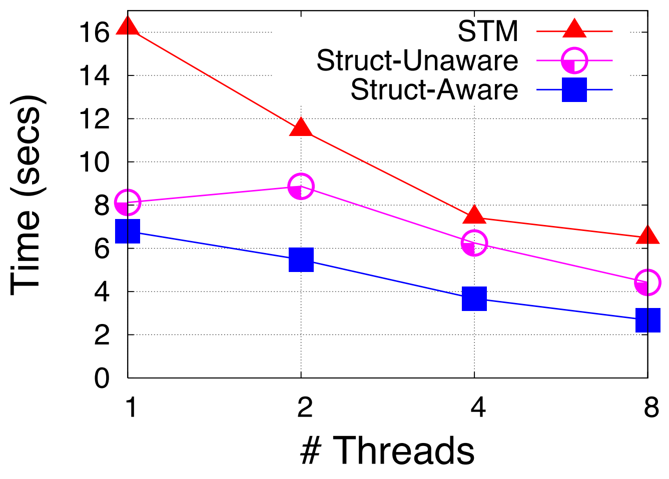

After

# Prior results

- Exploiting library semantics to improve lock placement



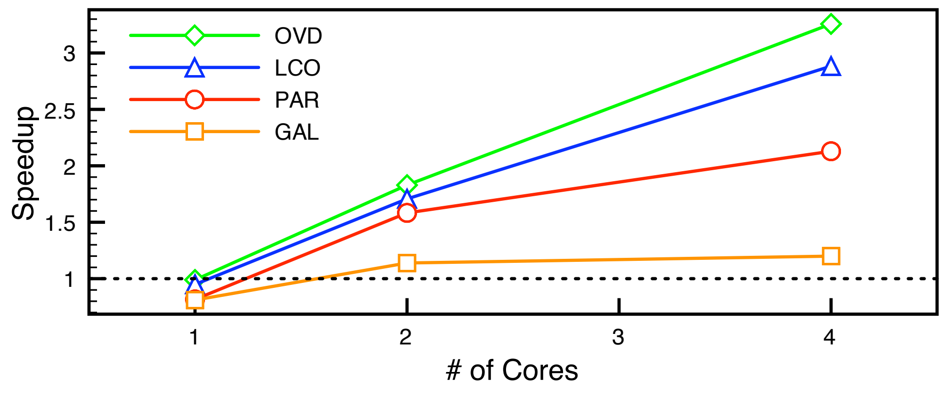
(a) Genome



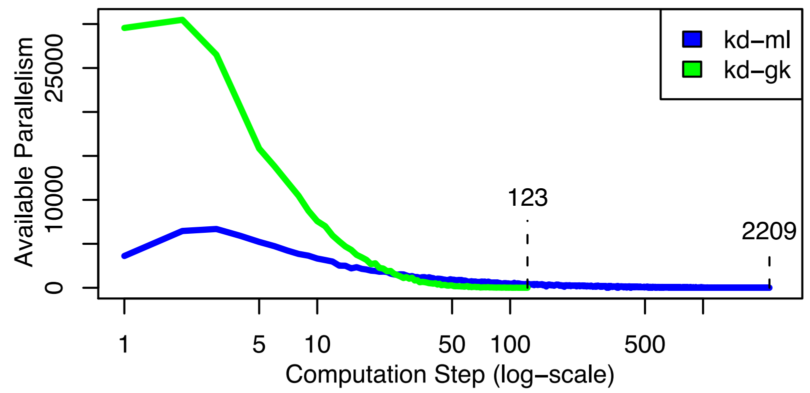
(b) Vacation

# Prior results

- Exploiting library semantics to improve parallelism and locality



(a) Performance of mesh generation with and without exploiting locality semantics



(b) Available parallelism in agglomerative clustering with and without exploiting data structure semantics

## Motivation: Why not compilers?

- Compilers do not understand library calls as abstractions
  - Option one: see them as black boxes which give no information → no opportunity for optimization
  - Option two: break abstraction boundaries and try to optimize → many transformation opportunities are only possible by understanding *semantics* of abstractions
- Needed: a way for compilers to *understand* abstractions
  - Broadway project attempted this, but focused on analyzing across abstractions, not semantics-driven transformations

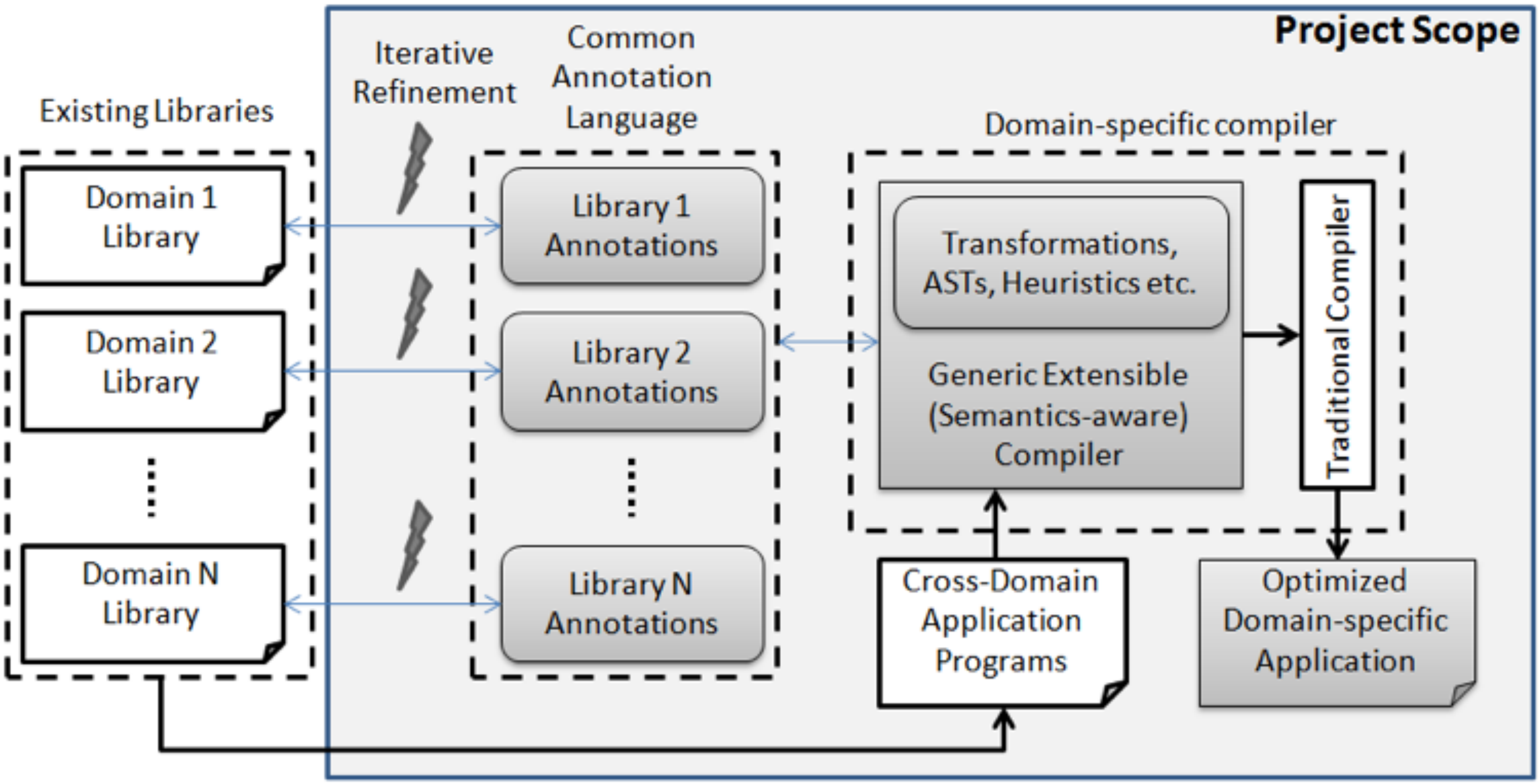
## Motivation: Why not domain-specific languages?

- DSLs are a great fit for this
  - Bake abstractions into the language
  - Optimize code at high level of abstraction based on semantic properties
  - Shown to be effective in various domains
    - SPL/Spiral for digital signal processing, Tensor contraction engine, etc.
- But they are not generalizable
  - New domain? New DSL!
  - What about applications that span domains? (e.g., multiphysics codes)
- Needed: a generic infrastructure for incorporating domain knowledge

## SLEEC: Principles

- Abstractions carried by domain libraries
  - Domain experts encode semantics, not compiler writers
  - *Need effective annotation language for capturing semantics*
- Compiler should be domain agnostic
  - Same infrastructure used for optimization and transformation regardless of domain
  - *Need common IR for capturing abstractions*
- Compiler should be able to optimize for various objectives
  - Do not want to focus solely on performance
  - *Need generic optimization ability and cost models*

# SLEEC: Overview



## SLEEC: Components

- Annotation language for capturing semantic properties of domain libraries
- High-level intermediate representation to represent programs that use annotated domain libraries
- Transformation strategies that leverage annotations to perform semantics-driven code transformations
- Optimization heuristics that use domain-specific cost models to find more efficient program variants
- Iterative refinement techniques that let the compiler work with incomplete information and infer missing information when possible

## Simple example

- Consider annotated linear algebra library that supports two methods
  - Matrix multiply
  - Equation solve
- Operations have mathematical properties that establish equivalence
  - Can solve ABx = b in two ways:
    - C = AB followed by Cx = b
    - Az = b followed by Bx = z
  - Latter may be more effective if A & B have special properties (e.g., triangular)

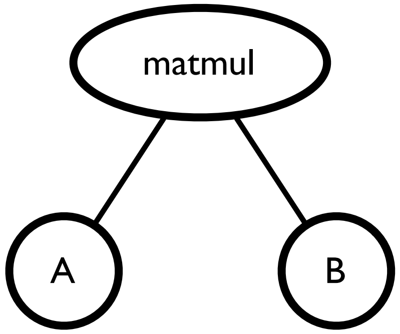
## Simple example

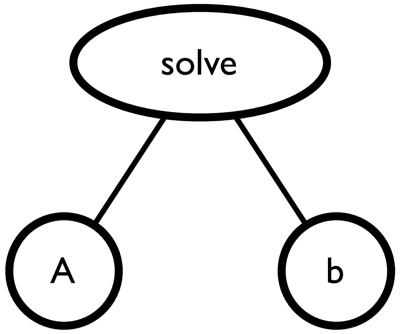
- Program code

```
matmul(A, B, C);
...
solve(C, b, x);
```

## Simple example: Abstract

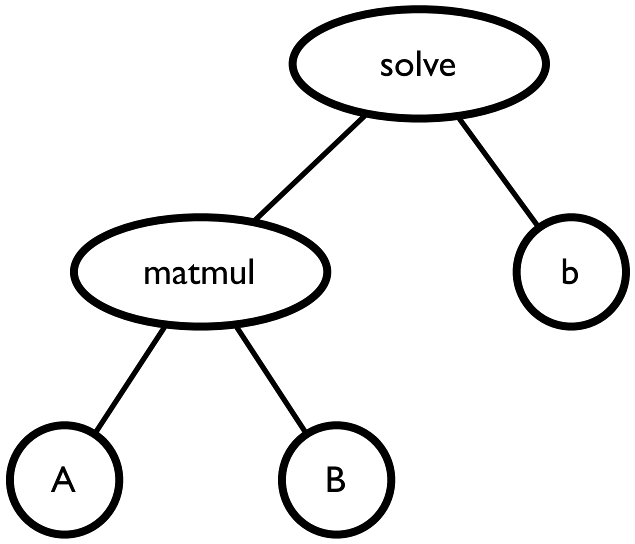
- Abstract into high level representation
- Expression tree to capture flow of data
- Library methods represented as high level operations
- Operands can be subtrees, too, to support composition





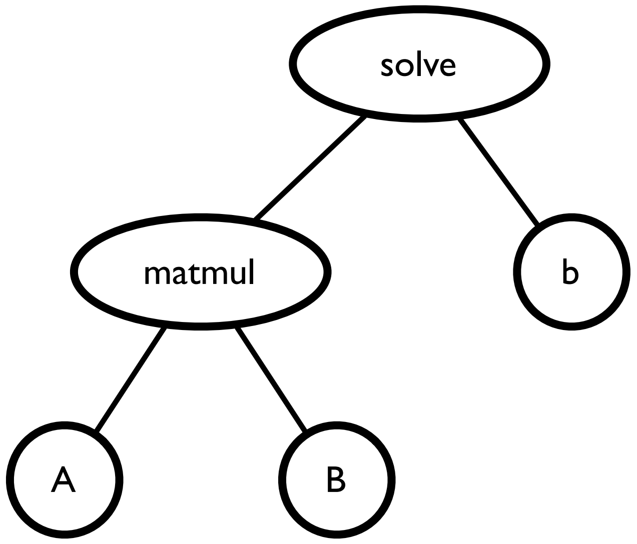
## Simple example: Abstract

- Abstract into high level representation
- Expression tree to capture flow of data
- Library methods represented as high level operations
- Operands can be subtrees, too, to support composition



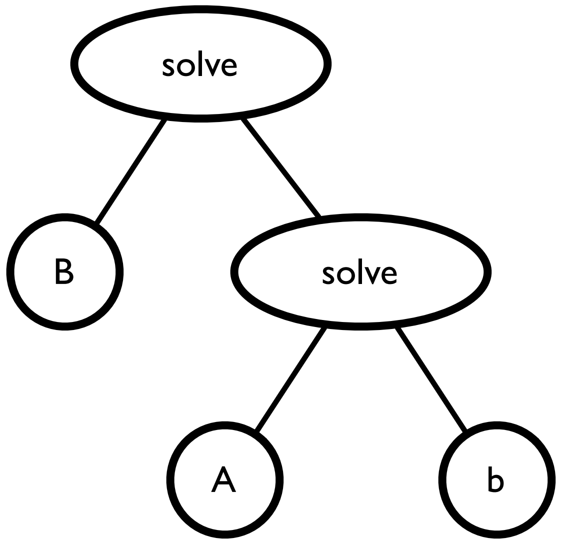
## Simple example: Transform

- Transformations expressed as rewrite rules on expression trees
- Rewrites match operation types (domain specific) but compiler applies them without understanding domain semantics



## Simple example: Transform

- Transformations expressed as rewrite rules on expression trees
- Rewrites match operation types (domain specific) but compiler applies them without understanding domain semantics



## Simple example: Concretize

- Re-materialize back to source code, or transform to other, lower-level IR

```
solve(A, b, z);
...
solve(B, z, x);
```

## Annotation language

- Domain libraries annotated by domain experts to interface with compiler infrastructure
- Questions
  - How to abstract libraries into IR
  - What kinds of transformations are legal
    - Represent as rewrite rules
    - How to verify? Can we synthesize?
  - How to concretize
    - Can this be inferred?

## Annotation language: cost models

- Most annotations deal with library interface
  - Semantic properties are associated with library specification, not implementation
- Can also provide cost estimates for library methods
  - Implementation and *architecture* specific
- Can express other properties of implementation
  - Energy estimates
  - Accuracy information

## Compiler infrastructure

- Compiler does not explicitly understand domains
  - But is extensible, allowing IR to be extended as new domains are added
- Transformations are just pattern-matched rewrite rules
  - Can use domain-specific information such as domain-specific equivalences, domain-specific properties
  - Can also substitute equivalent implementations of same method
- Generic compiler + annotated domain library = domain-specific compiler

# Compiler infrastructure: cost-driven optimization

- Applying transformations to program generates semantically equivalent program variants
    - No "best" variant: different implementations will work better in different situations or optimize for different metrics
- Compilation as optimization problem
    - Minimize objective function
        - FLOPs, energy efficiency, etc.
    - Subject to constraints
        - Semantically equivalent to original program, meets accuracy constraints, etc.
- Same infrastructure can be used to optimize for a variety of metrics

## Iterative refinement

- Typical problem with domain-specific languages or annotation approaches: what if program is incompletely annotated?
- Want compiler to still produce useful results
- Key property: compilation process is about optimization, not correctness
    - Lack of information does not raise correctness issues
    - As more annotations are provided, compilation results improve

## Iterative refinement: inference

- Can we infer missing information?
- Transformation annotations
    - Can we use synthesis techniques to infer legal transformations?
- Cost models
    - Can we use machine learning techniques to build cost models automatically?

## Potential impacts

- Programmability: Programmers can focus on developing methods, using high level libraries, without worrying about careful optimization
- Performance portability: Ability to select between library variants automatically eases transition to new architectures
- Scalability: Cost models can incorporate parallelism, locality, communication to enhance scalability
- Energy efficiency: Parameterized compilation can optimize for energy use instead of performance without rewriting infrastructure
- Resilience: Cost models can incorporate resilience information (e.g., algorithmic fault tolerance information), compilation can choose variants based on resilience properties

## Implementation plan

- Work driven by "challenge" applications and domains
  - Computational mechanics and multiscale techniques (lead: Arun Prakash)
  - Peridynamics and Trilinos libraries (lead: Michael Parks)
- Build compiler infrastructure in ROSE
  - Compiler infrastructure and optimization strategies (leads: Milind Kulkarni and Sam Midkiff)
  - Annotation language and IR (leads: Milind Kulkarni and Sam Midkiff)
  - Cost models and performance modeling (lead: Vijay Pai)

## Concrete deliverables

- Annotation language
- Common IR
- Generic compiler infrastructure
- “Showcase” annotated libraries

## Conclusions

- We want to work with you!
  - Finding and and annotating new domains
  - Verification and synthesis for transformations
  - Translating between different representations
  - Runtime targets/constraints for compilation

https://engineering.purdue.edu/~milind/sleec

https://engineering.purdue.edu/SLEEC