

# Abstractions for Language Independent Program Transformations

Karl Trygve Kalleberg  
University of Bergen



UNIVERSITETET I BERGEN

# Introduction

- Motivation and context
- Obstructions
- Suggested solutions

# Introduction

- Motivation and context
  - dependable software development is an unsolved problem
  - most projects implemented with dozens of languages
  - program transformation techniques and tools hold a good promise
- Obstructions
  - current program transformation implementations are
    - mostly language-specific
    - often bound to concrete language infrastructure
  - **wanted: techniques that improve language independence (LI)**
- Suggested solutions – support for:
  - **abstracting over concrete program object models**
  - **cross-cutting concerns, and flexible adaptation of generic algorithms**
  - **strategic rewriting for graph-like program models**

# Outline of Dissertation

- Introduction
- Programmable Software Transformation Systems
- Strategic Term Rewriting
- Program Object Model Adapters
- Modularising Cross-cutting Transformation Concerns
- An Extensible Transformation Language
- Strategic Graph Rewriting
- Language Extensions
- Interactive Transformation as Transformation Libraries
- Extending Compilers with Transformation and Editing Environments
- Code Generation for Axiom-based Unit Testing
- Discussion
- Further Work
- Conclusion
- Summary



# Flow of Dissertation

- Analyse shortcomings of the state-of-the-art
- Identify some techniques which support LI well
  - strategic programming
  - component-based transformation system architectures
- Suggest complementary LI techniques
  - program object model adapters
    - liberate transformation system from specific program object model
  - aspects for transformation concerns
    - modularize cross-cutting concerns; retroactive algorithm adaptation
  - richer program models
    - strategic programming on both graph- and tree-like models
- Validate through case-studies
  - IDEs, framework-specific analysis and transformation, software testing tools

# Presentation Perspectives

- Language-centric
  - three extensions to the Stratego programming environment
    - two language extensions
    - one runtime extension
  - reimplemented Stratego runtime
- Technique-centric
  - abstraction technique over program models
  - facility for adapting transformation algorithms
  - strategic rewriting on graphs

– *State of the Art*

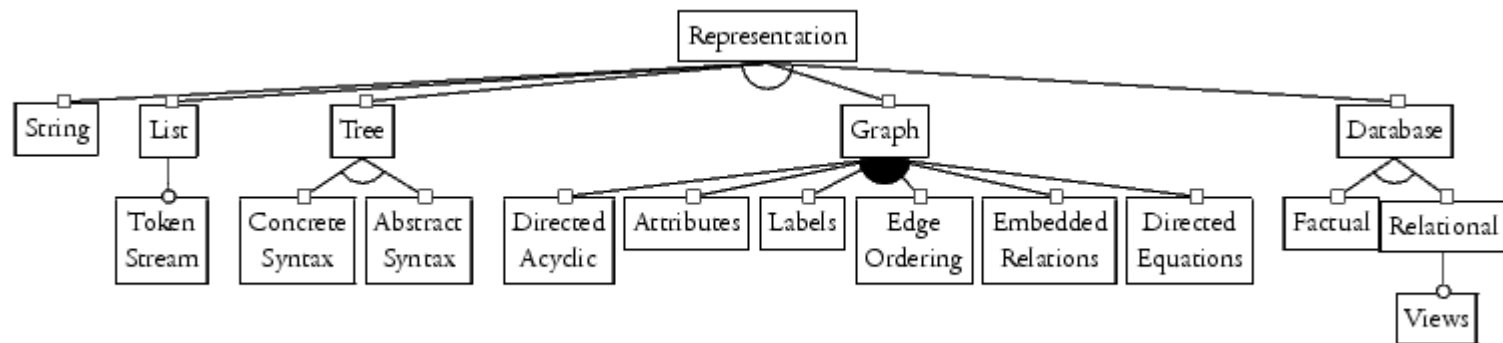
# Survey of Software Transformation Systems

- Motivation

- overview of the state-of-the-art in architectures and designs
- architecture/design manuals absent for program transformation systems
- understand models used to describe software

- Survey content

- feature models for describing design space



# Survey of Software Transformation Systems

- Motivation
  - overview of the state-of-the-art in architectures and designs
  - architecture/design manuals absent for program transformation systems
  - understand models used to describe software
- Survey content
  - feature models for describing design space
    - based on a dozen modern program transformation systems
  - discussion comparing and contrasting certain design tradeoffs
  - focus: *interplay between program object model (POM) design and transformation languages*

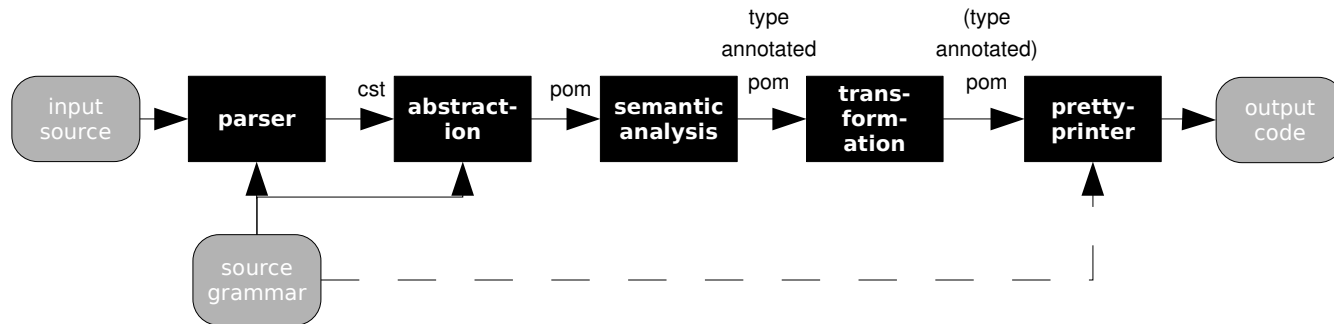


# Observation: No Unified Program Object Model

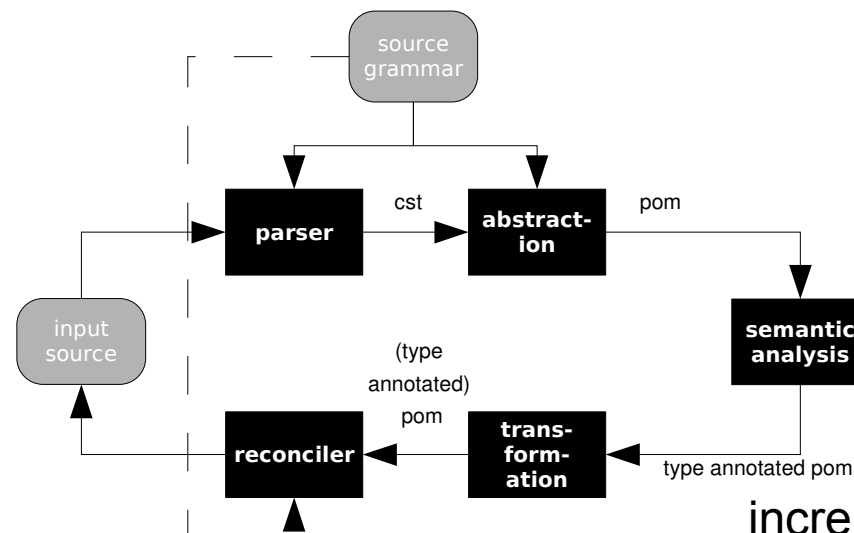
- POM design depends heavily on intent of system
  - transformation problem guides POM design
  - POM design restricts transformation expressiveness
- Derived (high-level) POMs are more abstract
  - mostly used for analysis and to guide source transformations
- Sliding scales of tradeoffs:

<b>POM</b>	<b>high-level</b>	<b>low-level</b>
<i>(subject) language</i>	general	specific
<i>problem</i>	specific	general
<i>source code transformation</i>	difficult	easy
<i>program model</i>	graphs, relations	trees, lists

# Observation: A Few Typical Architectures



source-to-source



incrementally updating



# Survey Conclusions

- Mature and accomplished field of research
  - numerous theoretical foundations for program transformation
  - 200+ program transformation systems exist
- Design space is very rich
  - many features tackle efficiency vs expressiveness tradeoffs
  - derived information expensive to keep fresh during transformation
- *Abstraction facilities for both POM and transformation language are necessary for language independence*

# Strategic Term Rewriting

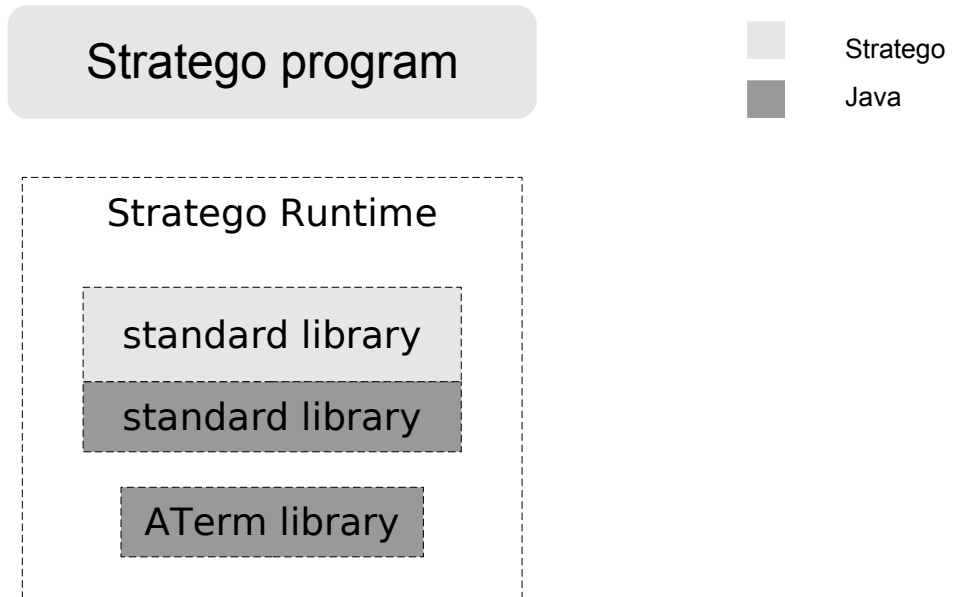
- Separation of concerns
  - data processing rules – often language specific
  - rule application strategies – often language independent
- Formal basis
  - System S: core calculus for strategic term rewriting
    - term building and matching operators – tree matching
    - generic traversal primitives – tree traversal
    - based around the term (tree) representation of programs
- Implementations
  - Stratego, Strafunski, Tom, ...
    - domain-specific languages for program transformation



# Stratego/XT

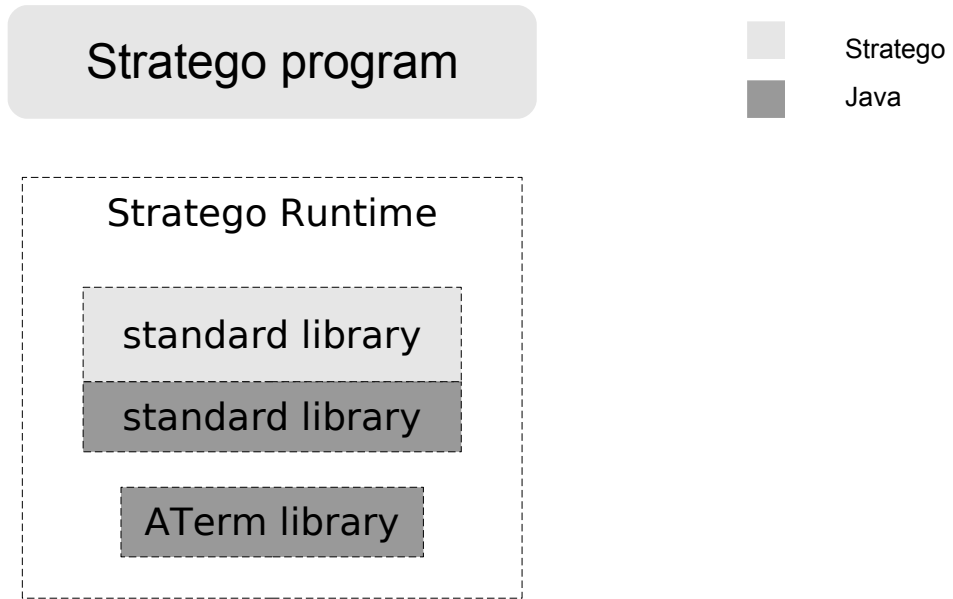
- Stratego
  - modular language
  - domain-specific
    - program transformation
  - compiled into efficient native programs
  - implementation of System S
- XT
  - Component architecture
  - Collection of transformation components
  - Composed into larger transformation pipelines
- System S in Stratego
  - Basic operators
    - `id`, `fail`
  - Term matching, building
    - `?`, `!`
  - Generic tree traversals
    - `one`, `all`
  - Strategy combinators
    - `sequence`, `choice`
  - Rewrite rules
    - `conditional`
    - `dynamic`

# Stratego Runtime

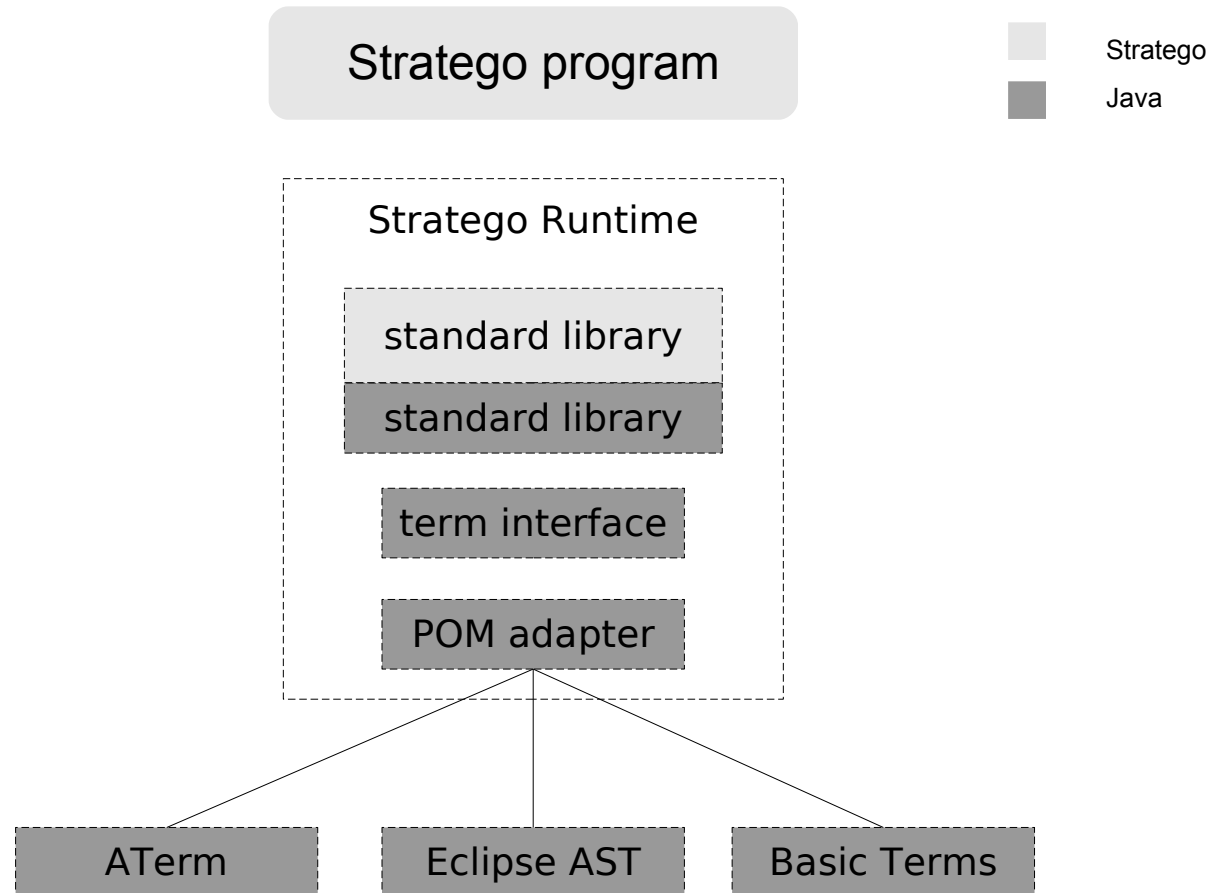


– *Abstractions for  
Language Independent  
Transformations*

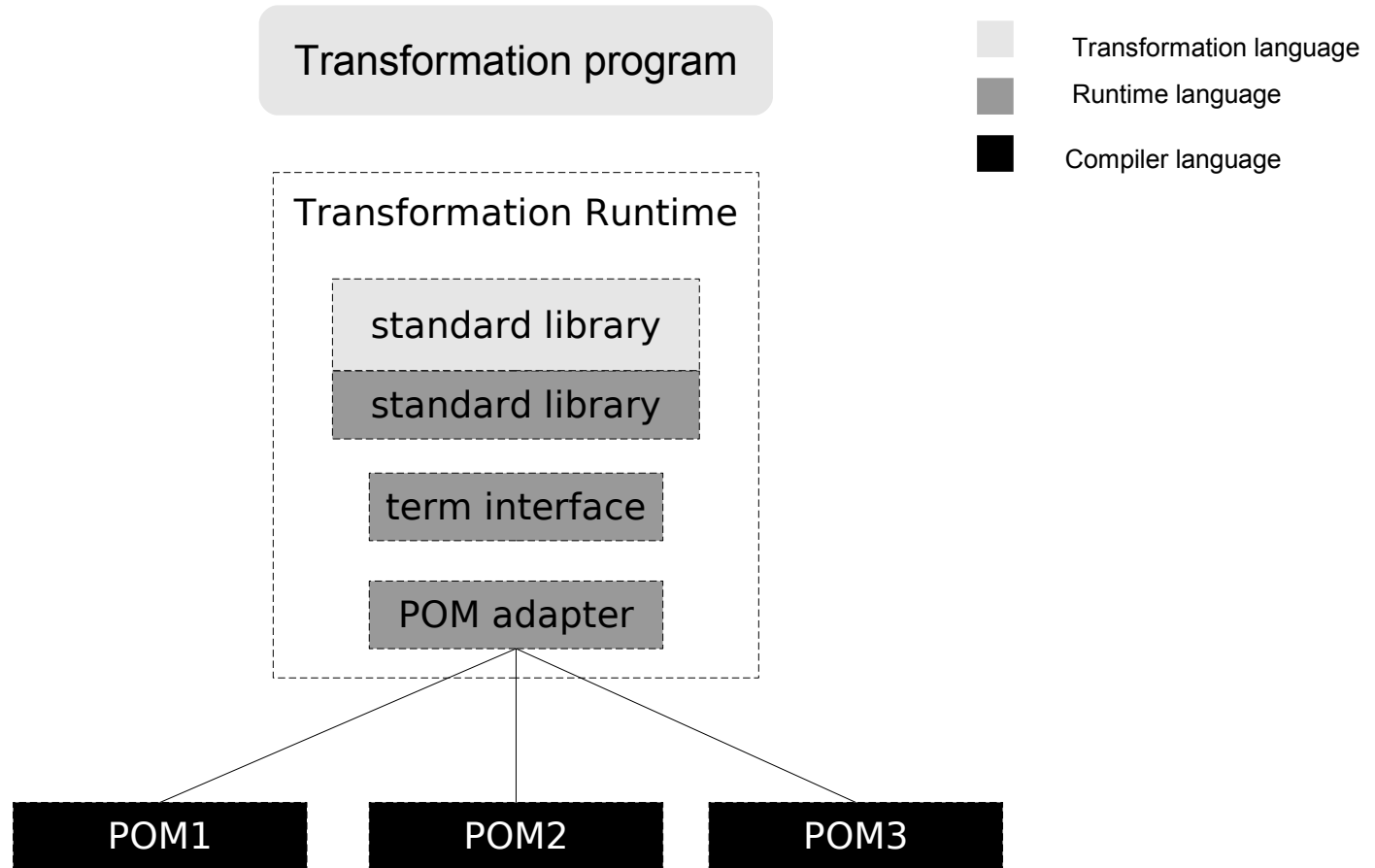
# Term-based Runtime



# Program Object Model Adapters



# Program Object Model Adapters

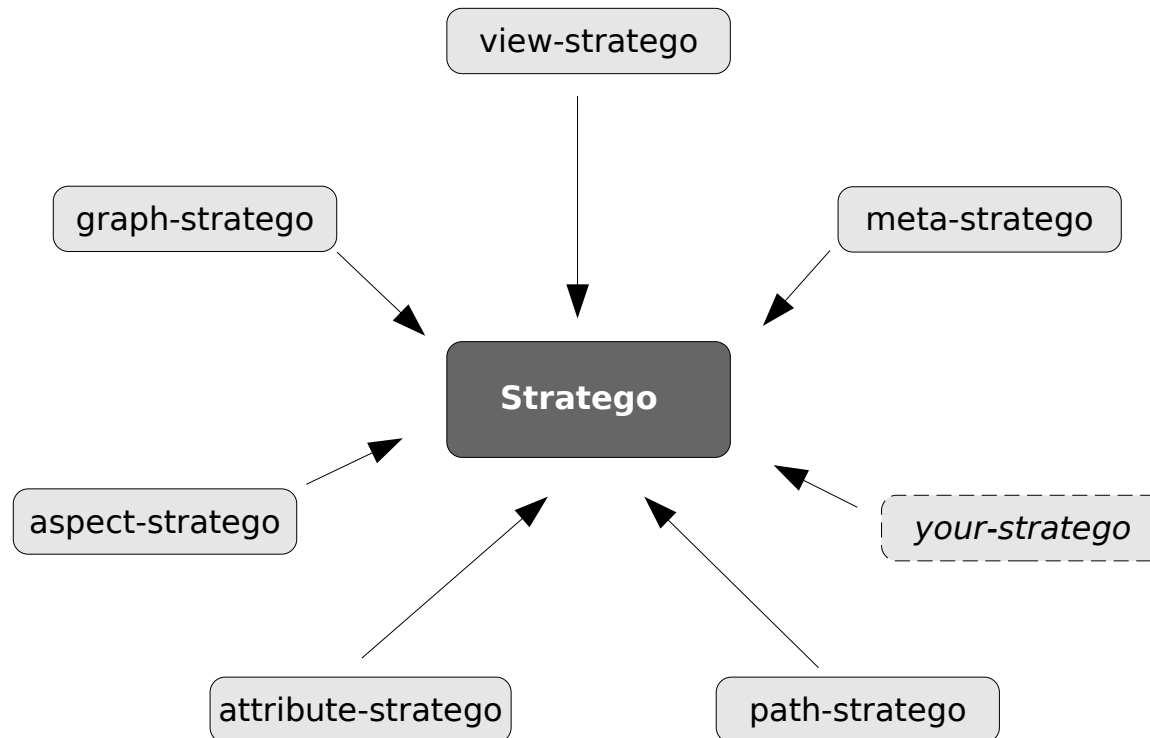


# POM Adapter Summary

- Serialization-free
- Supports heterogeneous POM models
  - multiple different POMs simultaneously
- Applicable to tree/term-based rewriting systems
- *Solution to POM abstraction problem*
  - transformation language abstractions still wanted



# An Extensible Transformation Language





# Example: A Logging Aspect

```
module simple-logger
strategies
  invoked(|s) = ![ "Rule '", s, "' invoked" ]
  ...
aspects
  pointcut log-rules(n) = rules(** => n)
  aspect simple-logger =
    before      : log-rules(r) = log(|Debug, <invoked(|r)>)
    after fail   : log-rules(r) = log(|Debug, <failed(|r)>)
    after succeed: log-rules(r) = log(|Debug, <succeeded(|r)>)
    after       : log-rules(r) = log(|Debug, <finished(|r)>)
```

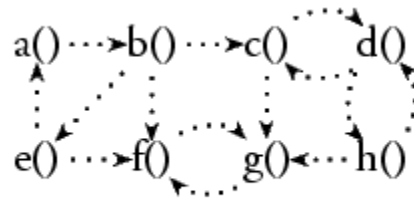
# AspectStratego Summary

- **Captures**
  - typical cross-cutting concerns
  - dynamic type checking of terms
  - algorithm extensions and adaptation
    - allows additional degrees of LI in generic transformation algorithms
    - example: generic data flow algorithms (constant propagation)
- **Limitations**
  - not declarative enough
    - may often require grey box reuse
  - aspect ordering/interaction not solved

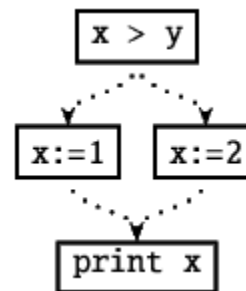
# GraphStratego

- Motivation

- extend System S/Stratego to handle cyclic structures
  - control/data flow graphs, call graphs, ...
- apply strategic programming to common compiler program models



call graph



flow graph

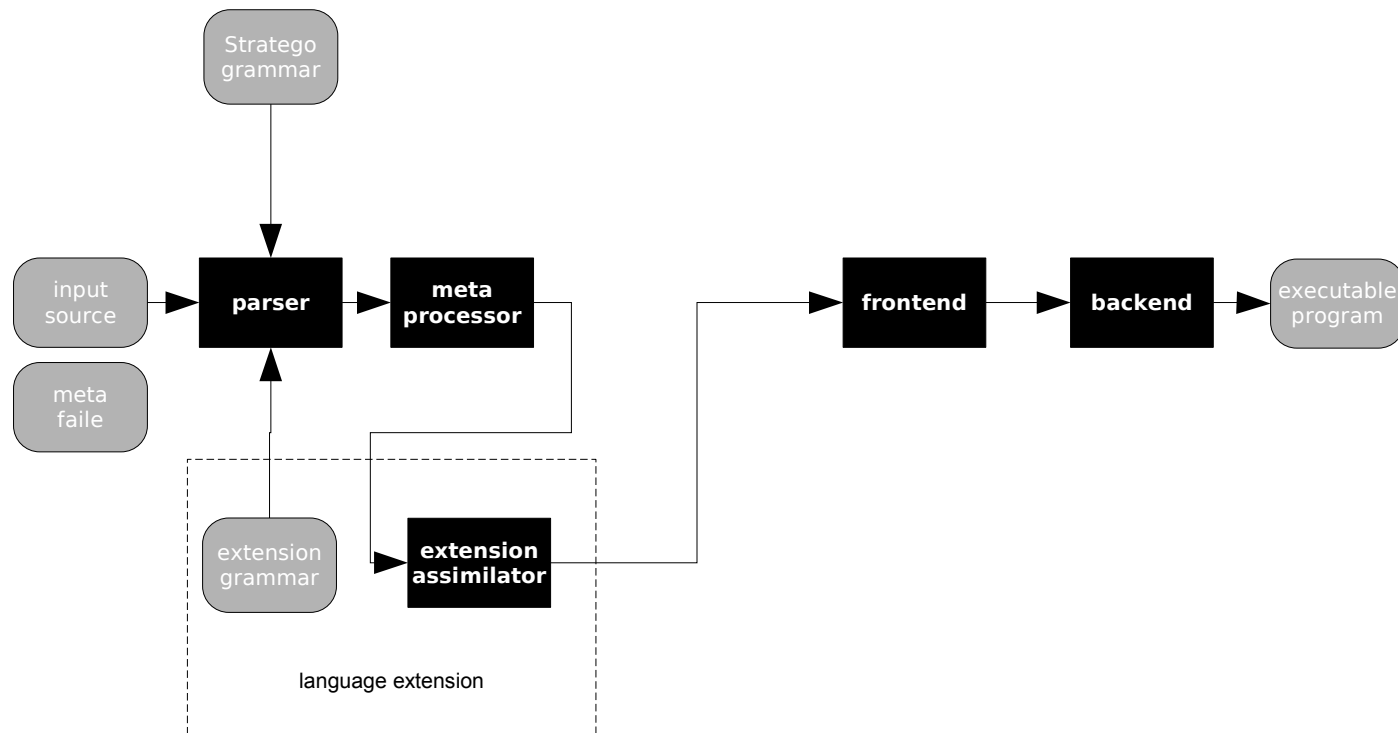


# GraphStratego Summary

- Results
  - program analysis on flow graphs possible
  - basic graph algorithms are expressible
  - termination of graph traversals ensured due to phased traversals



# An Extensible Compiler Pipeline



– *Case Studies*

# Case: Spooifax – an IDE For Stratego

- Motivation
  - illustrate application to interactive program transformation
  - investigate interoperation capabilities of the new runtime
  - provide good development environment for Stratego developers
- Experience
  - editor scripting with Stratego almost for free due to new runtime
  - entry-level for new Stratego programmers lowered due to modern development aids
  - robust parsing of incomplete code very difficult with current parsing formalisms



# Case: Transformation and Analysis Scripts

- Framework-specific transformations and analyses
  - developers write analysis & transformation scripts
  - reuse compiler infrastructure for parsing, type checking

# Case: Transformation and Analysis Scripts

- Framework-specific transformations and analyses
  - developers write analysis & transformation scripts
  - reuse compiler infrastructure for parsing, type checking

Java

```
String x;  
for(int i = 0; i < x.size(); i++) { ... }
```

Stratego

```
check-for =  
  ?ForStatement(_, e, _, _)  
  ; <topdown(try(call-to-immutable))> e  
  
call-to-immutable =  
  ?MethodInvocation(_, _, _, _, _, [])  
  ; binding-of => MethodBinding(class-name, _, _, _)  
  ; <list-contains(?class-name)> immutable-classes  
  ; emit-warn(|"Call to method on immutable object in loop iteration")
```

# Transformation and Analysis Scripts – Summary

- Experience

- plugging into the Eclipse JDT with POM adapter was quick and easy
- prototype is reasonably efficient
  - process simple analysis on 2.7 MLoC in ~4 minutes
- Stratego easily captures many interesting analyses and transformation problems

# Case: Code Generation for Axiom-based Unit Testing

- Motivation

- apply techniques and tools to implementing software development tool
- experiment with axiom-based unit testing techniques

- Experience

- constructed JAxT, a test generator tool
  - plugin for Eclipse
  - based on the ECJ bindings
- several revisions as testing techniques evolved
  - required multiple rewrites of JAxT
- code inspector and generator was trivial to implement
  - revision cycle was quick - (most time spent on GUI)
- code templates could benefit from concrete syntax



# Further Work

- **Program Object Model Adapters**
  - apply to higher-level program models, e.g. UML diagrams
  - improve tools for automatic derivation of adapters
- **AspectStratego**
  - aspect design patterns → new constructs for algorithm adaptation
  - runtime weaving of dynamically loaded components
- **GraphStratego**
  - describe formally as an extension to System S
  - redesign implementation to work optimally with POM adapter
- **Views**
  - a robust and practical mapping/abstraction mechanism over POMs



# Conclusion

- Contributions

- survey of program transformation systems
- program object model adapter technique
- aspect language for cross-cutting concerns in transformations
- language for strategic graph rewriting
- validation through several case-studies

- Result

- techniques are applicable in practice; improve language independence
- additional work is warranted
  - formalise graph language
  - view mechanism for POMs

