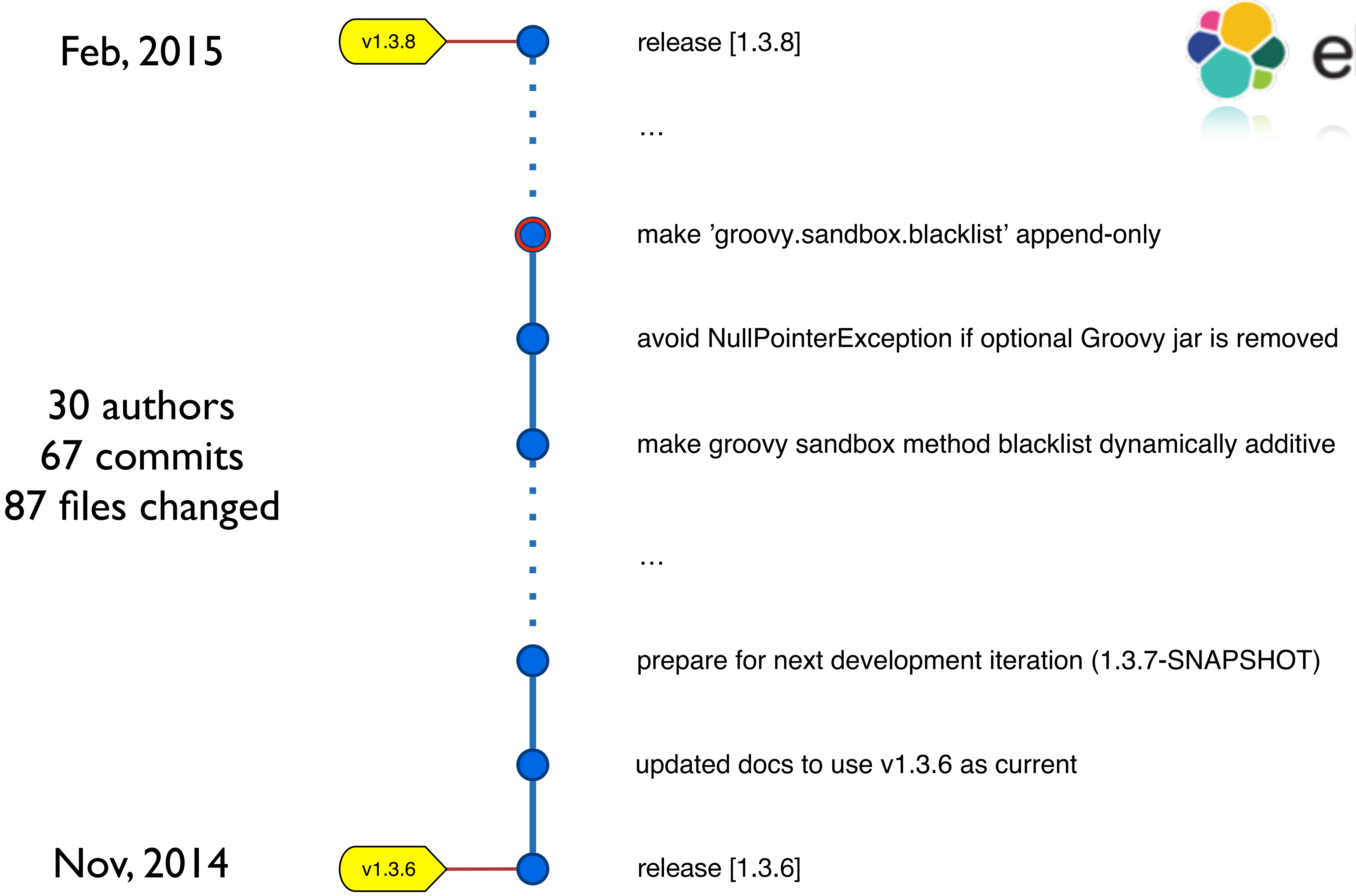
Semantic Slicing of Software Version Histories

Yi Li / U Toronto
Julia Rubin / MIT
Marsha Chechik / U Toronto

ASE 2015 / Lincoln, NE

Motivation

elastic



Feb, 2015

v1.3.8

release [1.3.8]

…

make ’groovy.sandbox.blacklist’ append-only

avoid NullPointerException if optional Groovy jar is removed

30 authors
67 commits
87 files changed

make groovy sandbox method blacklist dynamically additive

…

prepare for next development iteration (1.3.7-SNAPSHOT)

updated docs to use v1.3.6 as current

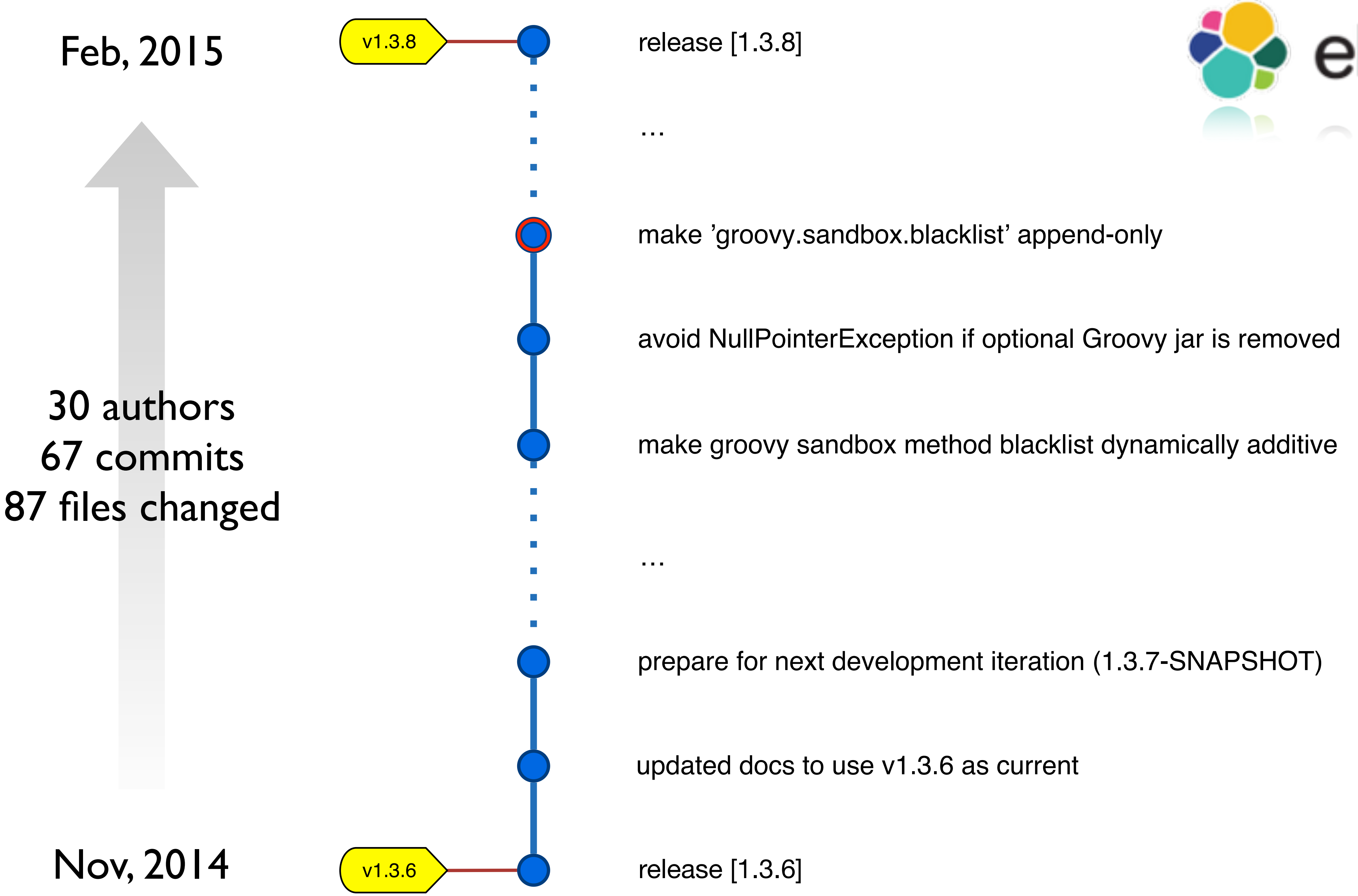
Nov, 2014

v1.3.6

release [1.3.6]

Motivation

elastic



Feb, 2015

30 authors
67 commits
87 files changed

Nov, 2014

release [1.3.8]

…

make ’groovy.sandbox.blacklist’ append-only

avoid NullPointerException if optional Groovy jar is removed

make groovy sandbox method blacklist dynamically additive

…

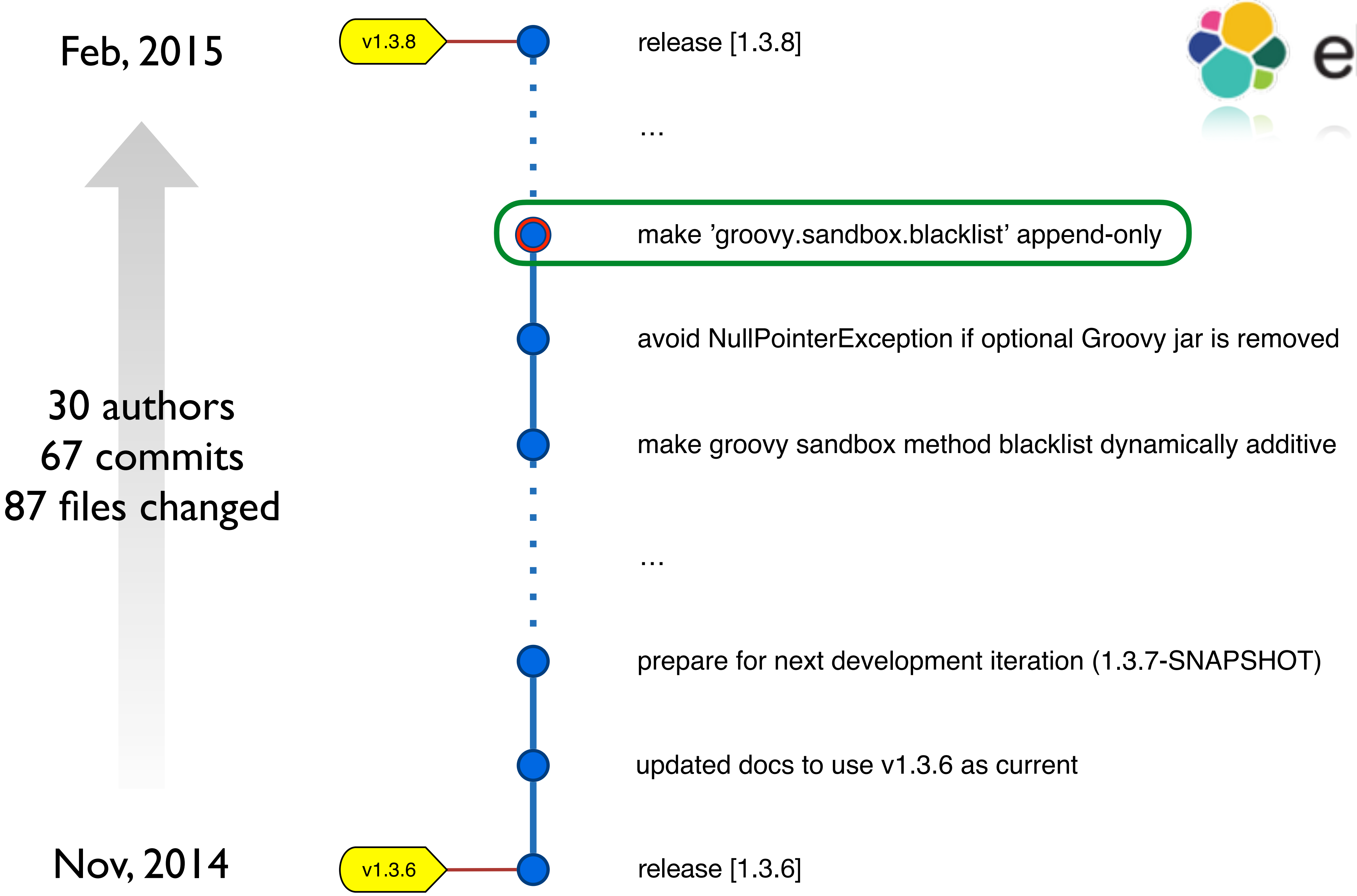
prepare for next development iteration (1.3.7-SNAPSHOT)

updated docs to use v1.3.6 as current

release [1.3.6]

Motivation

elastic



Feb, 2015

v1.3.8

release [1.3.8]

...

make 'groovy.sandbox.blacklist' append-only

avoid NullPointerException if optional Groovy jar is removed

make groovy sandbox method blacklist dynamically additive

30 authors
67 commits
87 files changed

...

prepare for next development iteration (1.3.7-SNAPSHOT)

updated docs to use v1.3.6 as current

Nov, 2014

v1.3.6

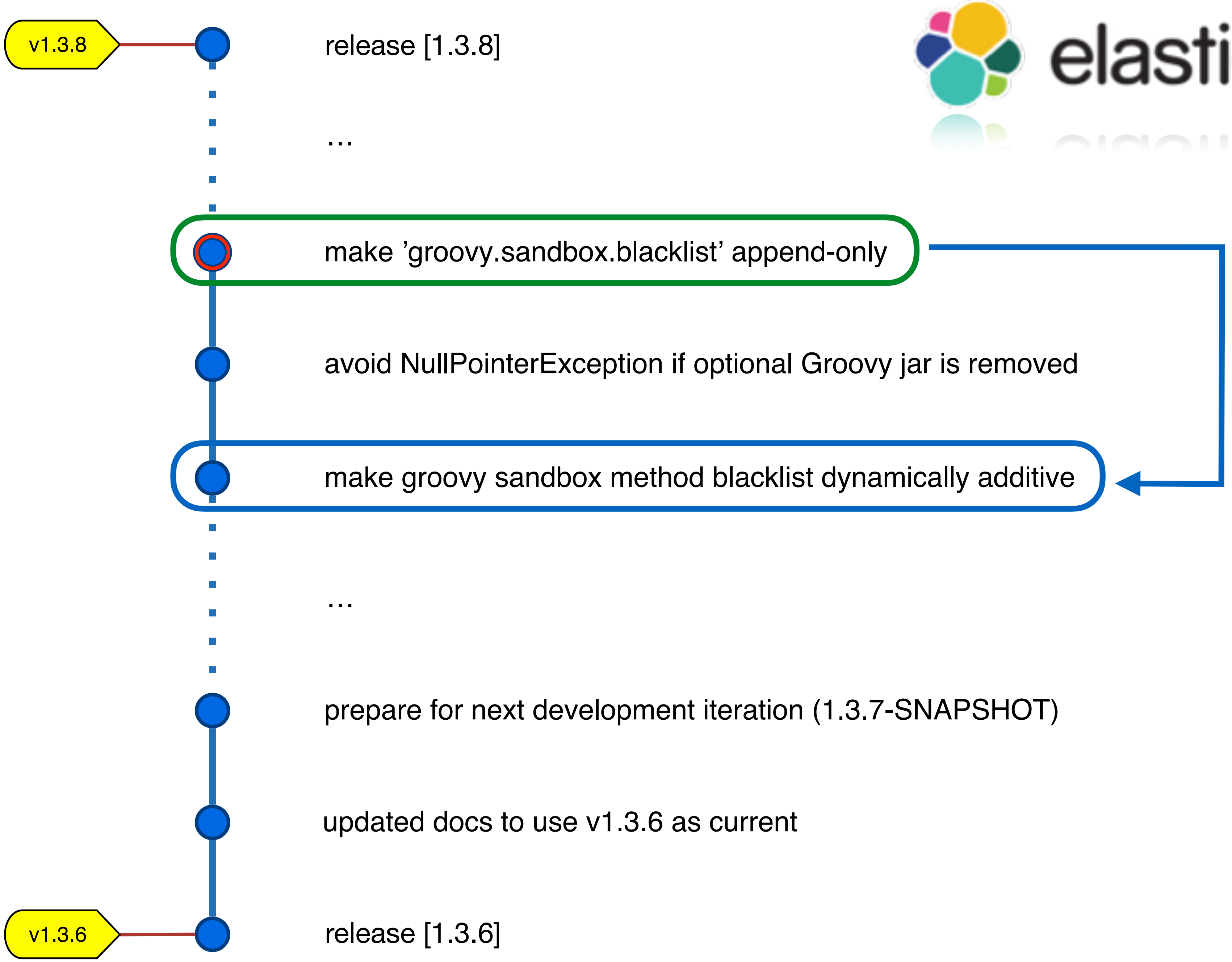
release [1.3.6]

Motivation

Feb, 2015

30 authors
67 commits
87 files changed

Nov, 2014



v1.3.8 release [1.3.8]

...

make 'groovy.sandbox.blacklist' append-only

avoid NullPointerException if optional Groovy jar is removed

make groovy sandbox method blacklist dynamically additive

...

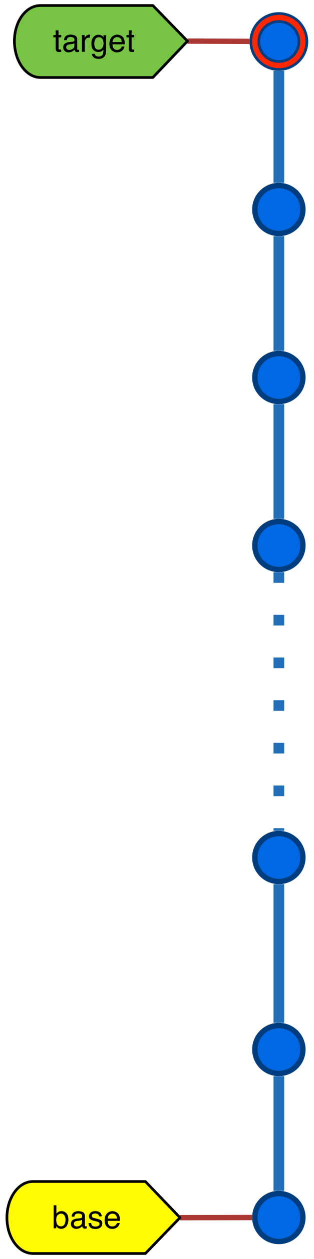
prepare for next development iteration (1.3.7-SNAPSHOT)

updated docs to use v1.3.6 as current

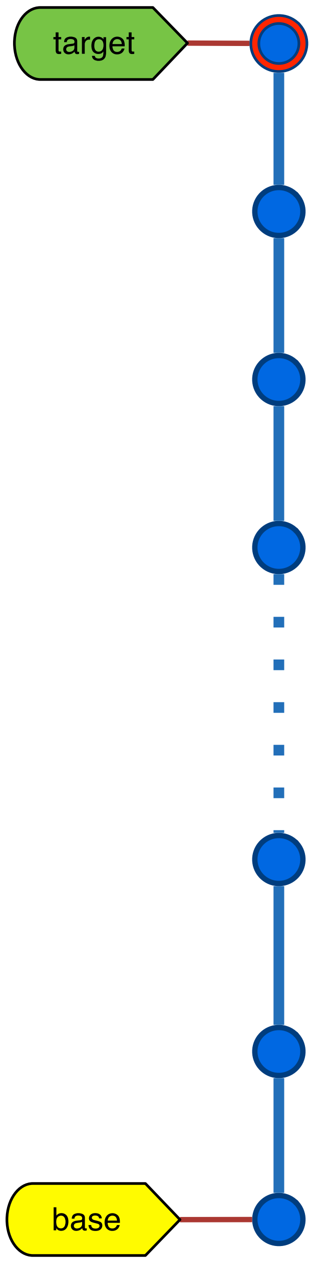
v1.3.6 release [1.3.6]

elastic

Why is it so hard?



Why is it so hard?



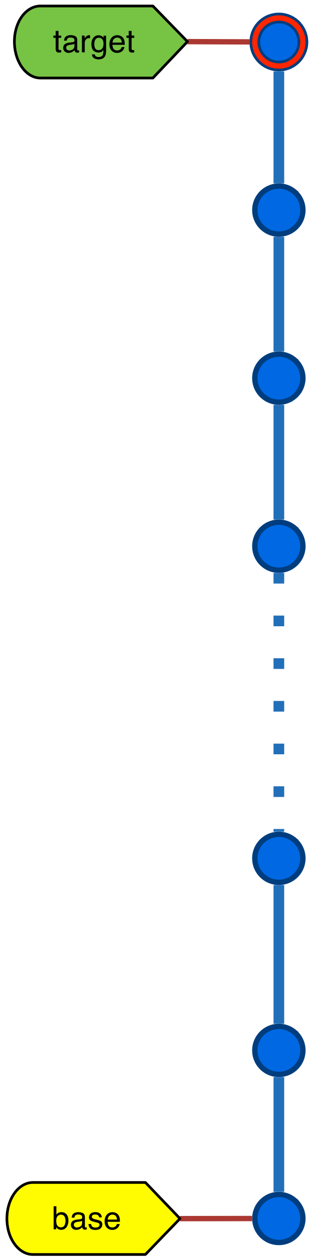
Options?

✗ 1. Pick target commits

2. Pick the whole history

3. Manually identify necessary commits

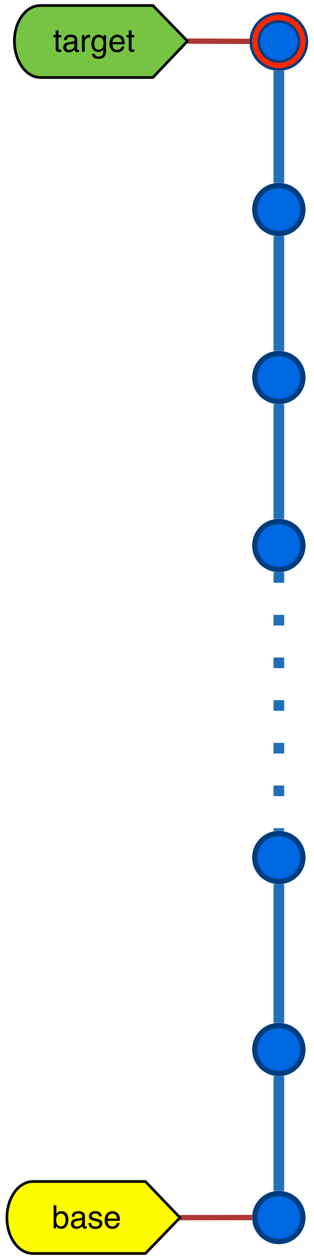
Why is it so hard?



Options?

1. ✗ Pick target commits
2. ✗ Pick the whole history
3. Manually identify necessary commits

Why is it so hard?



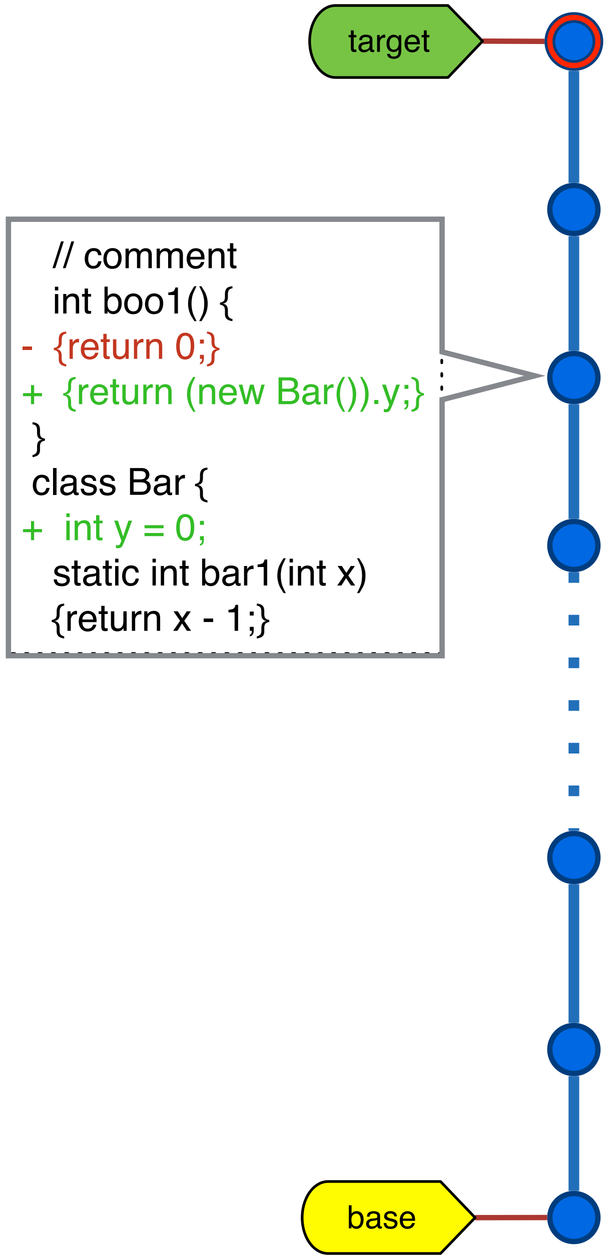
Options?

✗ 1. Pick target commits

✗ 2. Pick the whole history

✗ 3. Manually identify necessary commits

Why is it so hard?



```
   // comment
   int boo1() {
-  {return 0;}
+  {return (new Bar()).y;}
 }
 class Bar {
+  int y = 0;
   static int bar1(int x)
   {return x - 1;}
```

Options?

✗ 1. Pick target commits

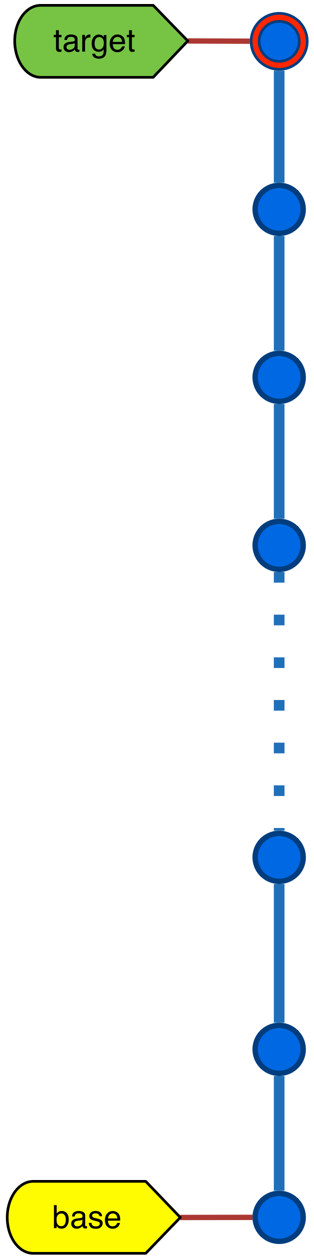
✗ 2. Pick the whole history

✗ 3. Manually identify necessary commits

Existing version control tools:

- Code treated as plain texts
- Do not understand the semantics
- User provided semantic/logical grouping is inaccurate!

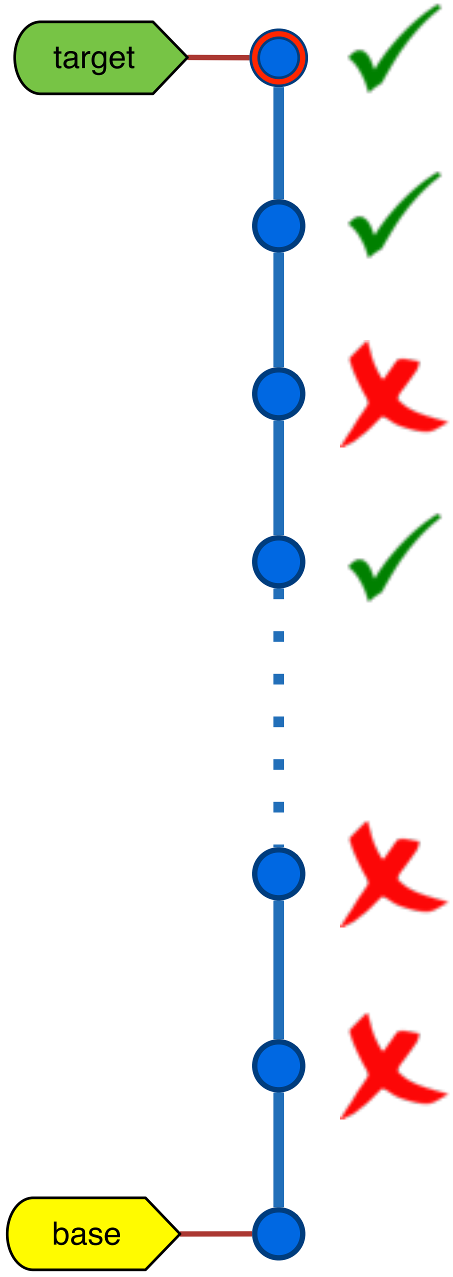
What can we do?



Exploit existing artifacts:

- Strictly *structured data*
- Well-defined language *syntax* and *semantics*
- Carefully designed *test suites*

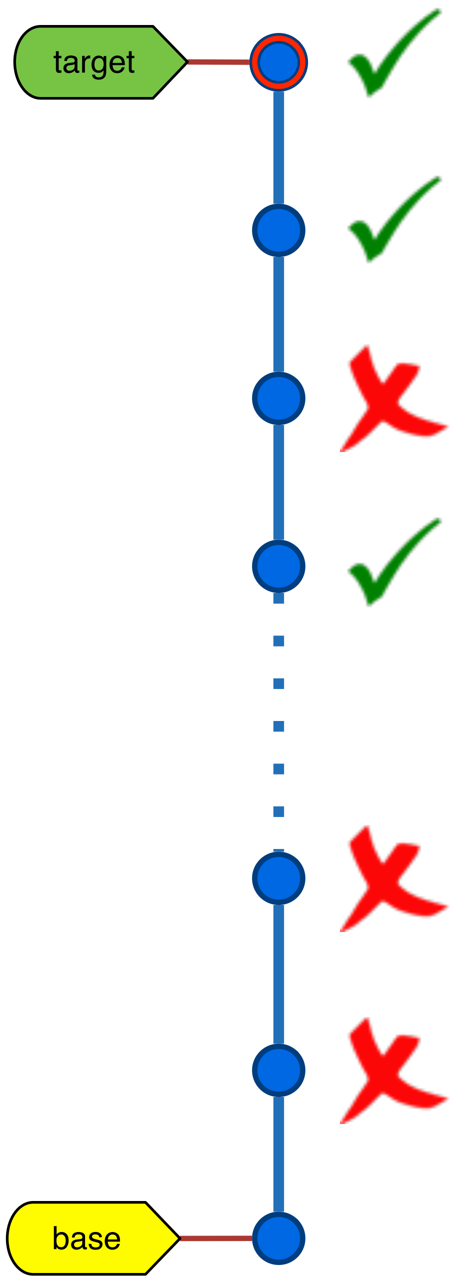
What can we do?



Exploit existing artifacts:

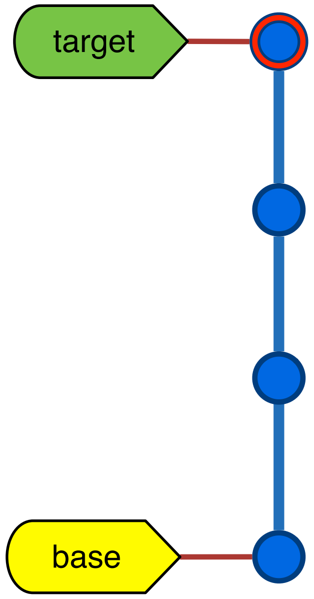
- Strictly *structured data*
- Well-defined language *syntax* and *semantics*
- Carefully designed *test suites*

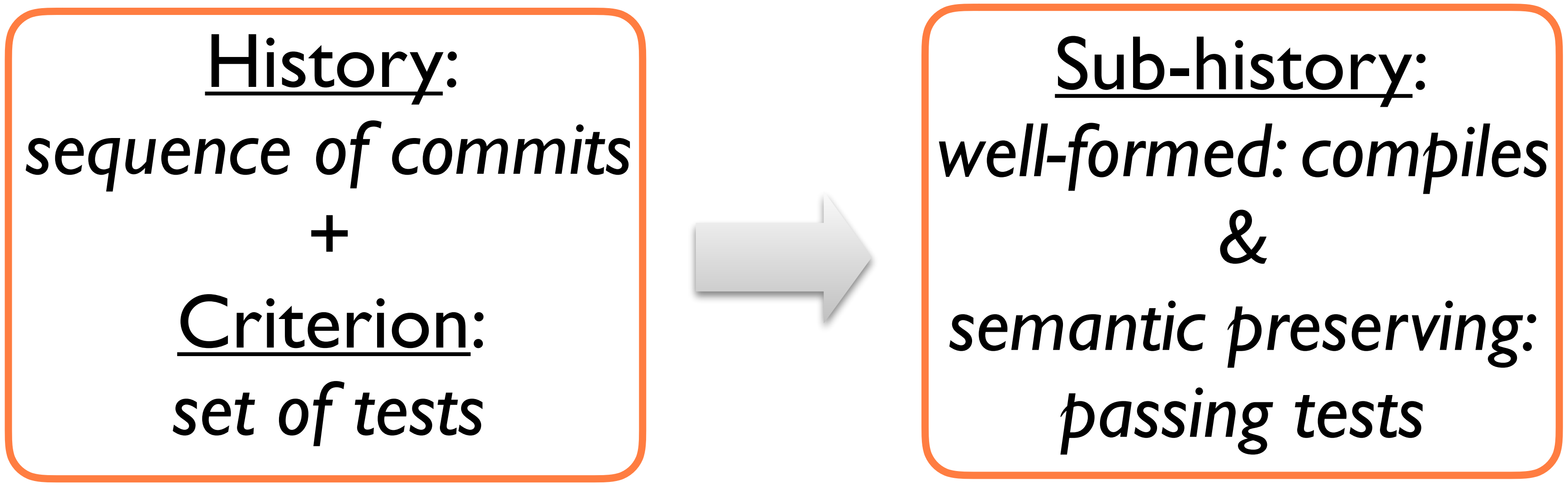
Solution: Semantic Slicing



Exploit existing artifacts:

- Strictly *structured data*
- Well-defined language *syntax* and *semantics*
- Carefully designed *test suites*





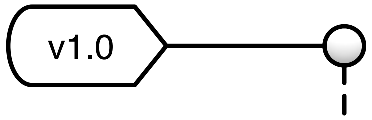
History:
sequence of commits
+
Criterion:
set of tests

Sub-history:
well-formed: compiles
&
semantic preserving: passing tests

Outline

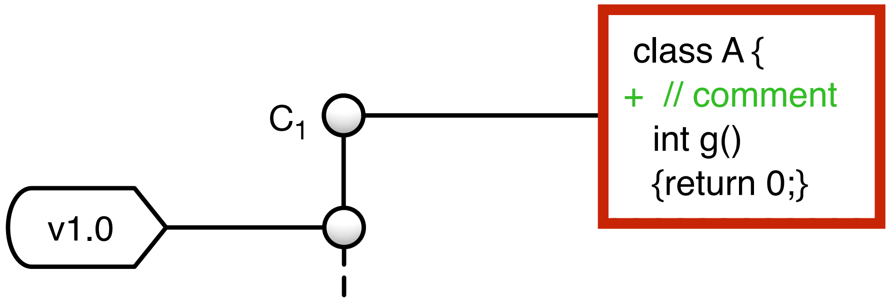
1. Introduction
2. Dependency Hierarchy
3. CSlicer Algorithm
4. Evaluation
5. Related Work & Conclusion

Running Example



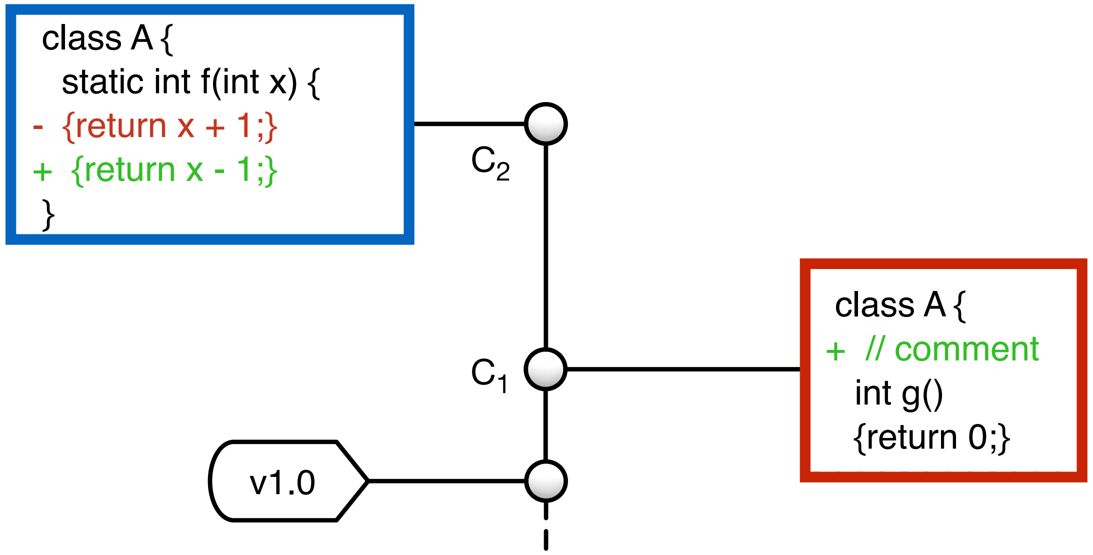
```
class A {
  int g()
  {return 0;}
}
class B {
  static int f(int x)
  {return x + 1;}
}
```

Running Example



```
class A {
  // comment
  int g()
  {return 0;}
}
class B {
  static int f(int x)
  {return x + 1;}
}
```

Running Example

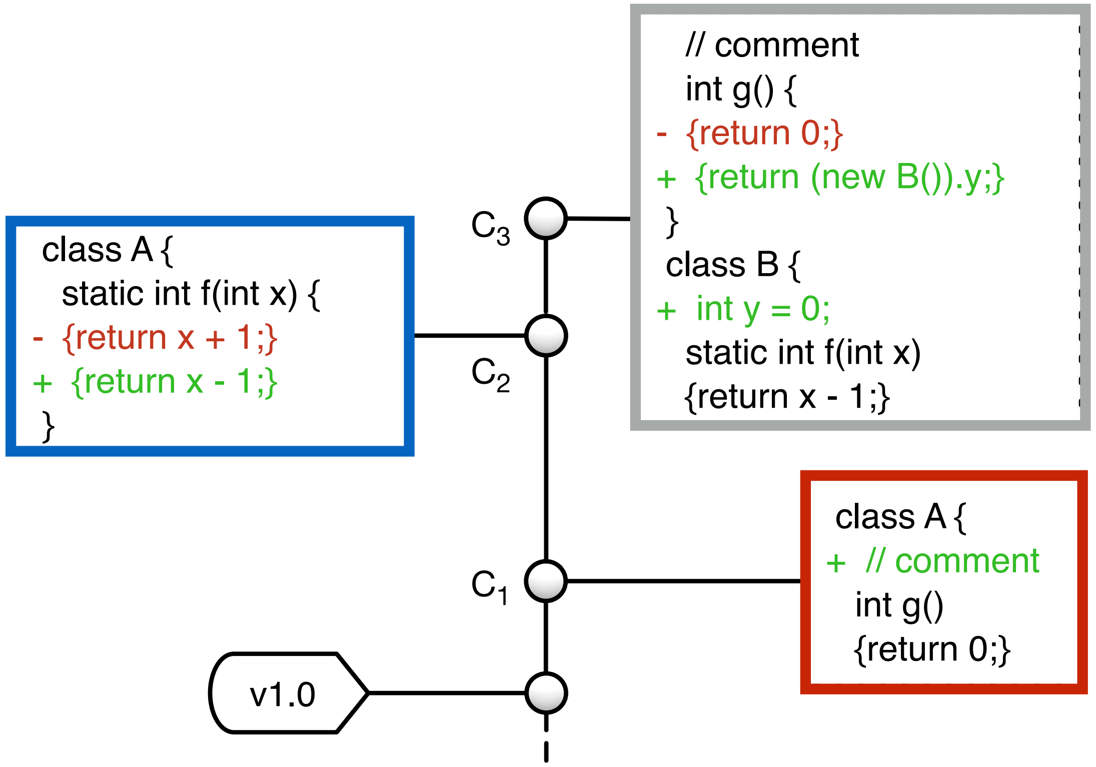


```
class A {
  static int f(int x) {
-  {return x + 1;}
+  {return x - 1;}
}
```

```
class A {
+  // comment
   int g()
   {return 0;}
```

```
class A {
  // comment
  int g()
  {return 0;}
}
class B {
  static int f(int x)
  {return x - 1;}
}
```

Running Example



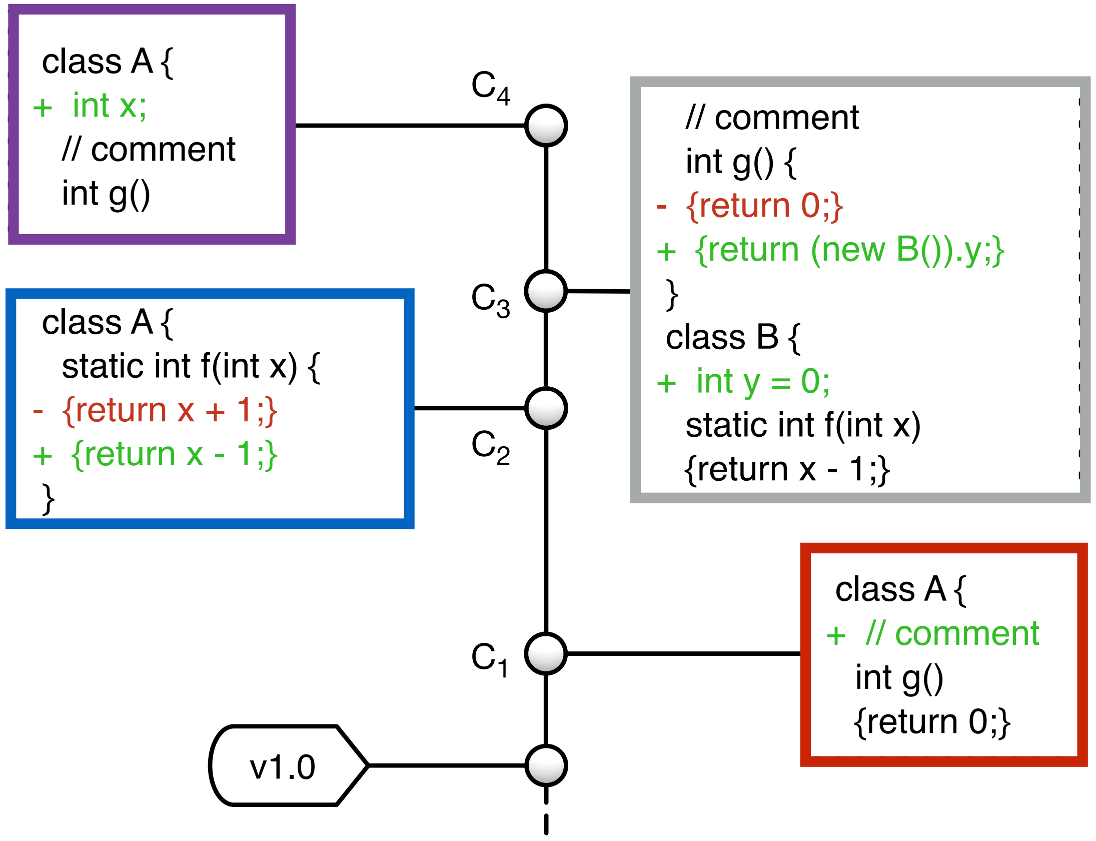
```
class A {
  static int f(int x) {
-  {return x + 1;}
+  {return x - 1;}
 }
```

```
   // comment
   int g() {
-  {return 0;}
+  {return (new B()).y;}
  }
 class B {
+  int y = 0;
   static int f(int x)
   {return x - 1;}
```

```
 class A {
+  // comment
   int g()
   {return 0;}
```

```
class A {
 // comment
 int g()
 {return (new B()).y;}
}
class B {
 int y = 0;
 static int f(int x)
 {return x - 1;}
}
```

Running Example



```
 class A {
+  int x;
   // comment
   int g()
```

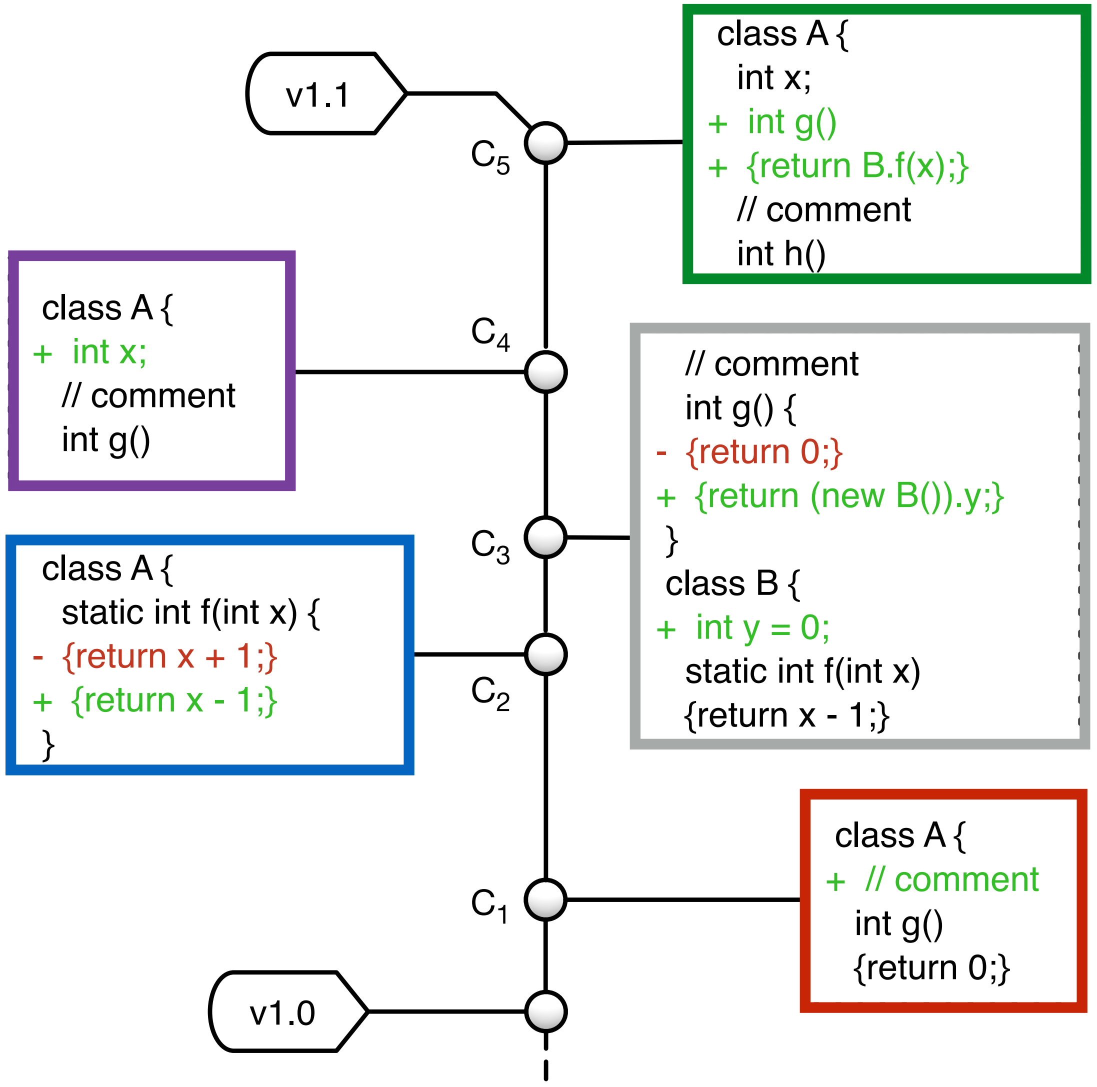
```
   // comment
   int g() {
-  {return 0;}
+  {return (new B()).y;}
 }
 class B {
+  int y = 0;
   static int f(int x)
   {return x - 1;}
```

```
 class A {
   static int f(int x) {
-  {return x + 1;}
+  {return x - 1;}
 }
```

```
 class A {
+  // comment
   int g()
   {return 0;}
```

```
class A {
 int x;
 // comment
 int g()
 {return (new B()).y;}
}
class B {
 int y = 0;
 static int f(int x)
 {return x - 1;}
}
```

Running Example



```
class A {
  int x;
+ int g()
+ {return B.f(x);}
  // comment
  int h()
```

```
class A {
+ int x;
  // comment
  int g()
```

```
  // comment
  int g() {
- {return 0;}
+ {return (new B()).y;}
 }
class B {
+ int y = 0;
  static int f(int x)
  {return x - 1;}
```

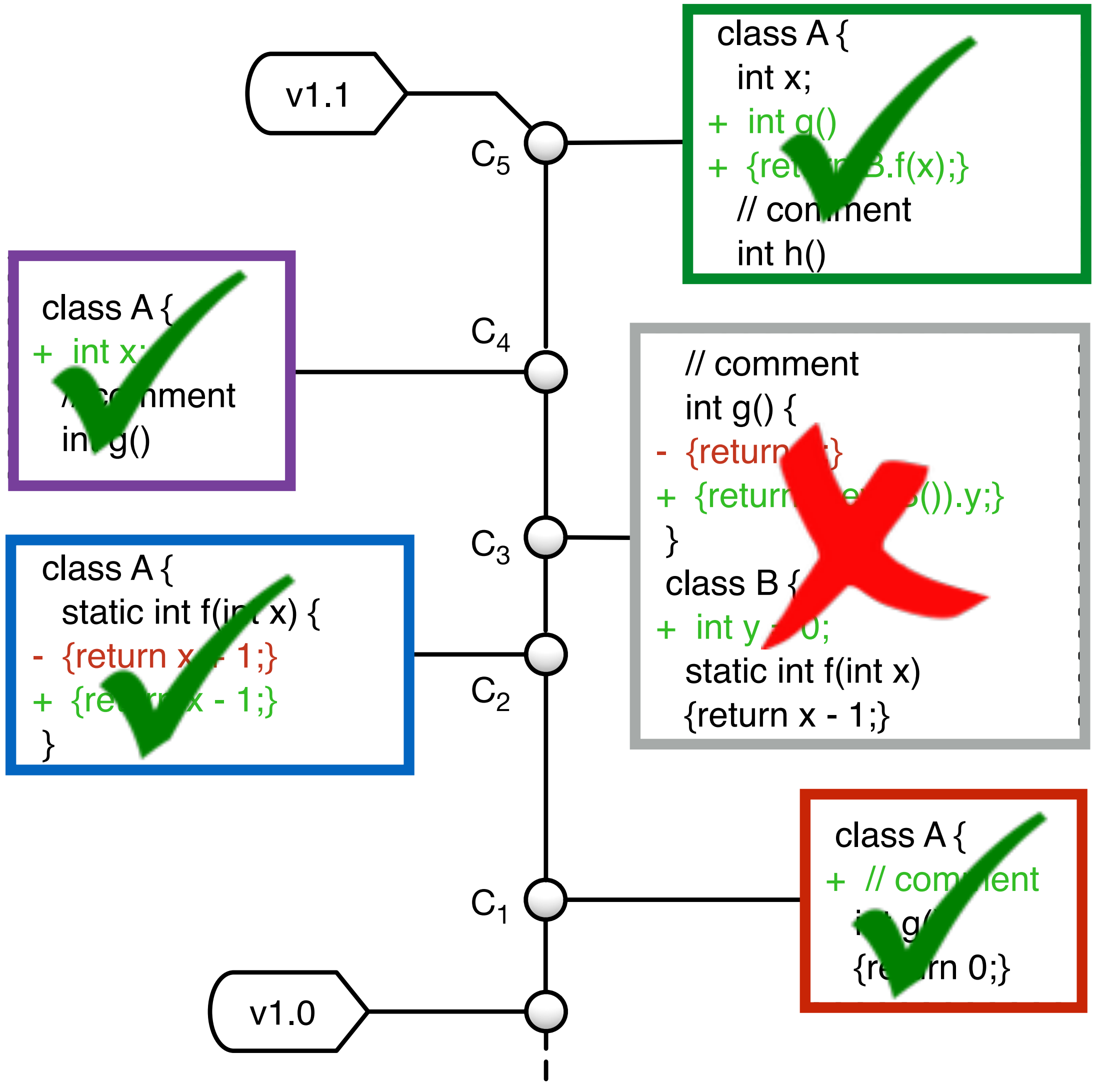
```
class A {
  static int f(int x) {
- {return x + 1;}
+ {return x - 1;}
}
```

```
class A {
+ // comment
  int g()
  {return 0;}
```

Test case:
a.g()==−1

```
class TestA {
  public void t1()
  A a = new A();
  {assertEquals(-1, a.g();}
}

class A {
  int x;
  int g()
  {return B.f(x);}
  // comment
  int h()
  {return (new B()).y;}
}
class B {
  int y = 0;
  static int f(int x)
  {return x - 1;}
}
```

Running Example



```
class A {
  int x;
+ int g()
+ {return B.f(x);}
  // comment
  int h()
```

```
class A {
+ int x;
  // comment
  int g()
```

```
  // comment
  int g() {
- {return 0;}
+ {return (new B()).y;}
 }
class B {
+ int y = 0;
  static int f(int x)
  {return x - 1;}
```

```
class A {
  static int f(int x) {
- {return x + 1;}
+ {return x - 1;}
}
```

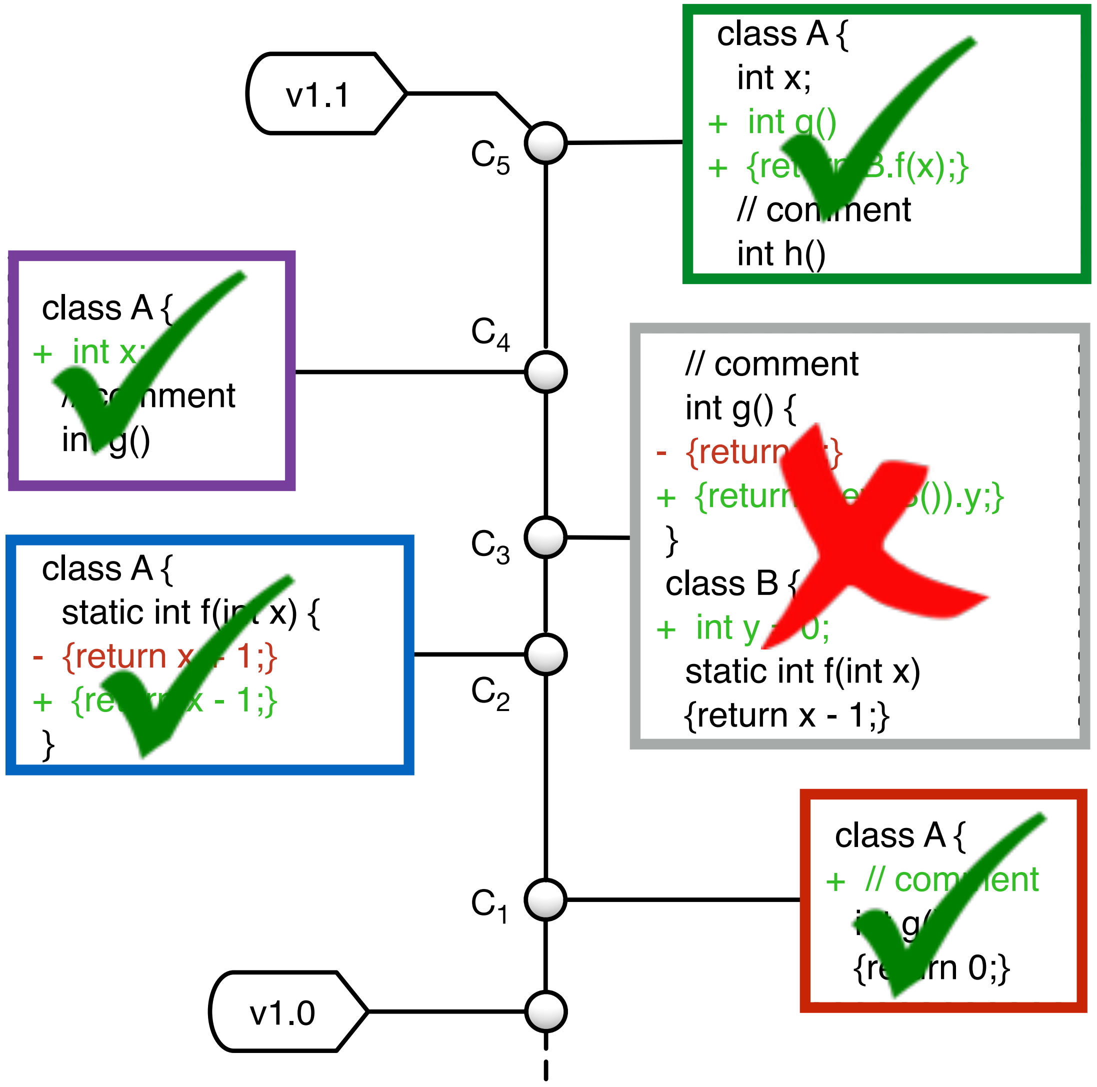
```
class A {
+ // comment
  int g()
  {return 0;}
```

Test case:
`a.g()==−1`

```
class TestA {
  public void t1()
  A a = new A();
  {assertEquals(-1, a.g();}
}
```

```
class A {
  int x;
  int g()
  {return B.f(x);}
  // comment
  int h()
  {return 0;}
}
class B{
  static int f(int x)
  {return x - 1;}
}
```

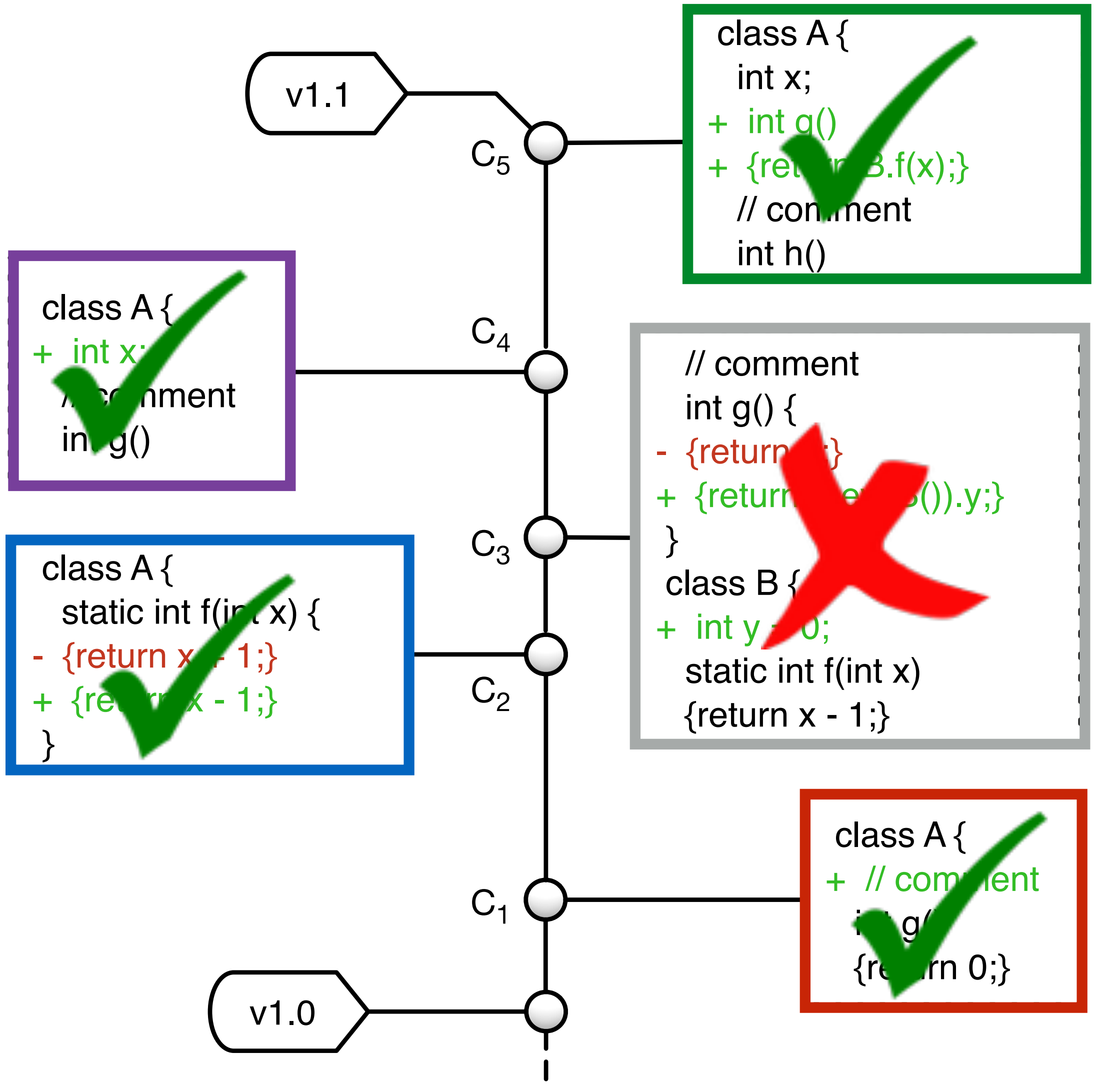
Running Example



Test case:
a.g()==-1

```
class TestA {
 public void t1()
 A a = new A();
 {assertEquals(-1, a.g();}
}


class A {
 int x;
 int g()
 {return B.f(x);}
 // comment
 int h()
 {return 0;}
}
class B{
 static int f(int x)
 {return x - 1;}
}
```

Running Example



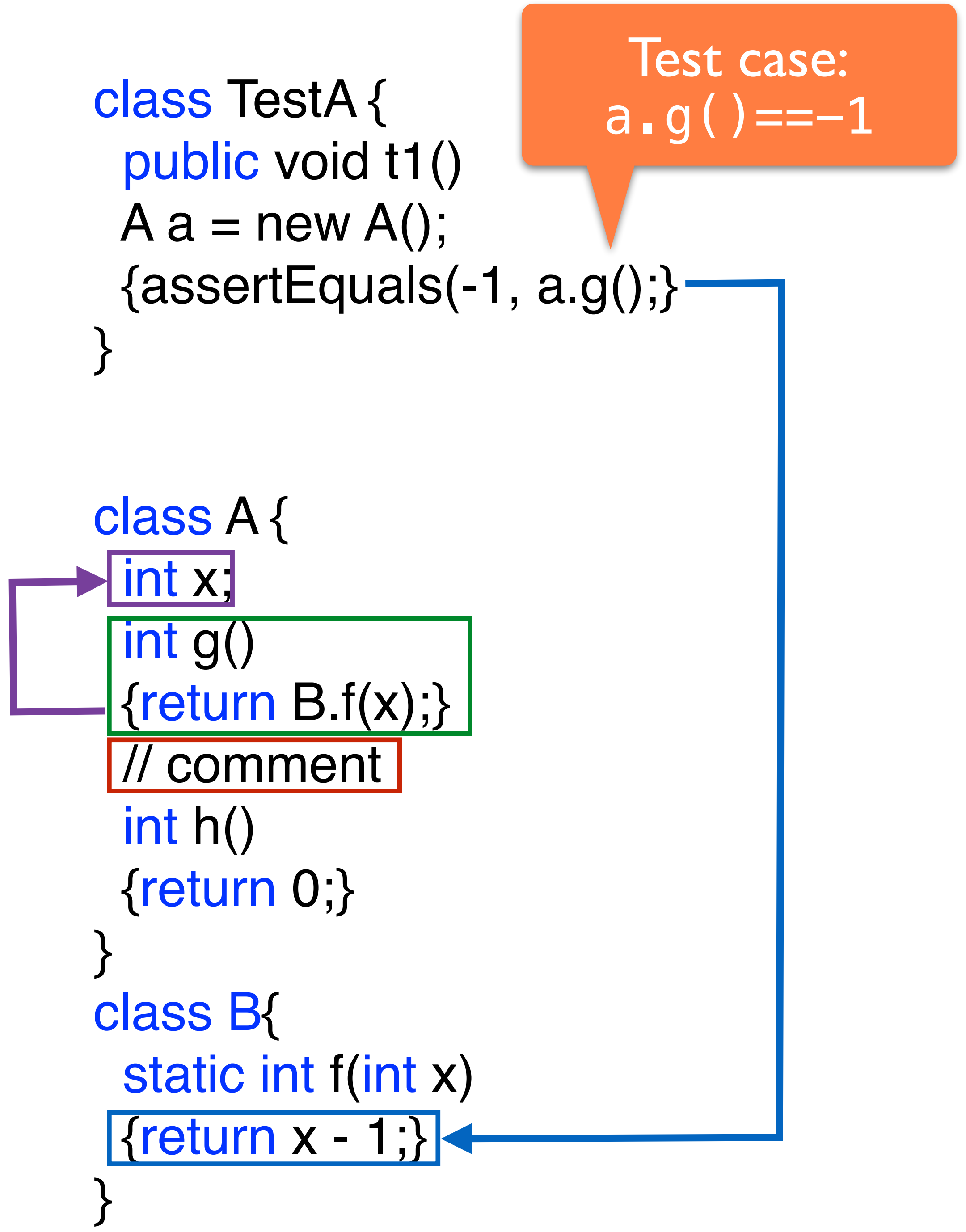
```
class A {
  int x;
+  int g()
+  {return B.f(x);}
  // comment
  int h()
```

```
class A {
+  int x;
  // comment
  int g()
```

```
  // comment
  int g() {
-  {return 0;}
+  {return (new B()).y;}
 }
 class B {
+  int y = 0;
  static int f(int x)
  {return x - 1;}
```

```
class A {
  static int f(int x) {
-  {return x + 1;}
+  {return x - 1;}
}
```

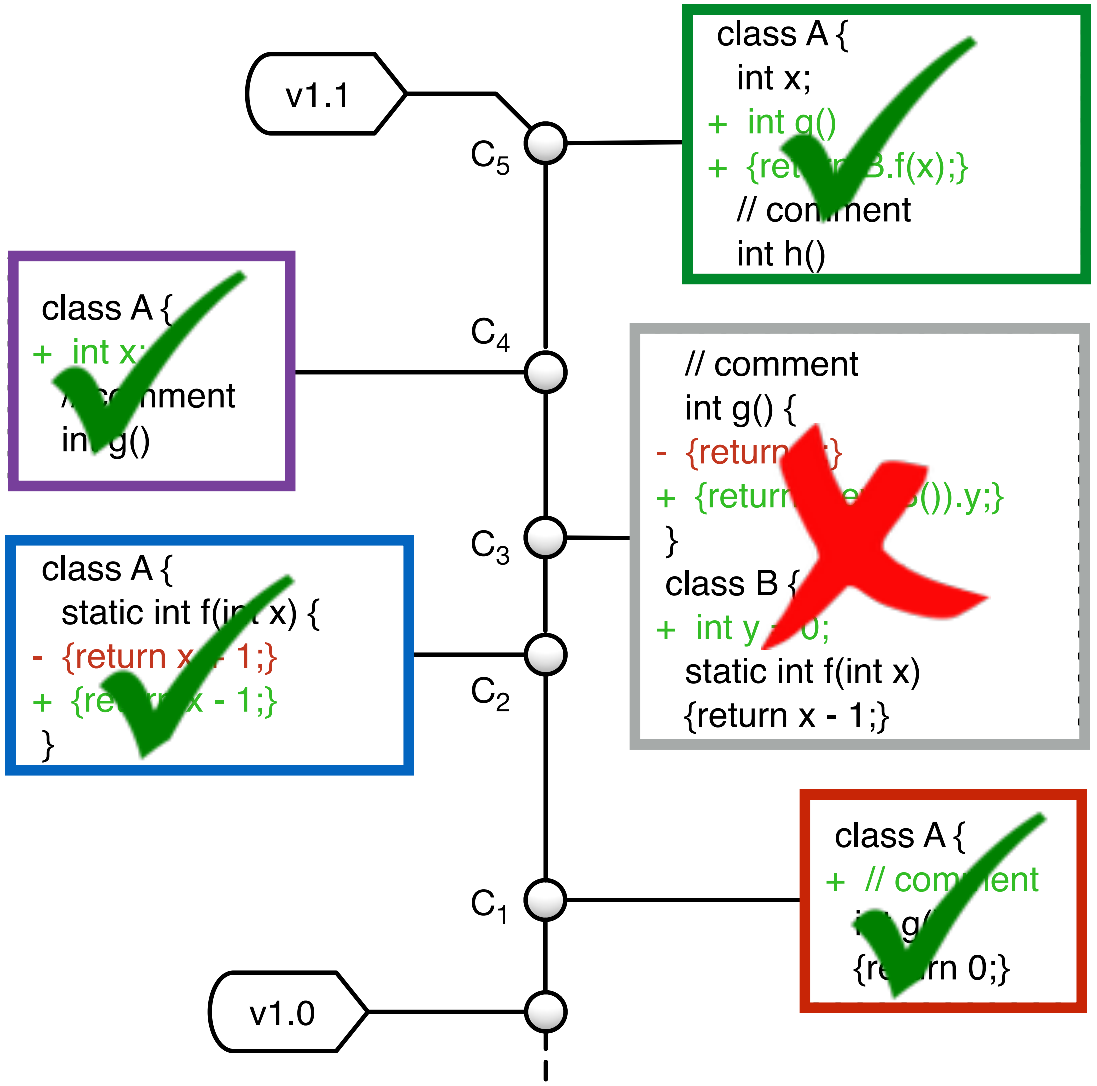
```
class A {
+  // comment
  int g()
  {return 0;}
```



Test case:
`a.g()==-1`

```
class TestA {
  public void t1()
  A a = new A();
  {assertEquals(-1, a.g();}
}


class A {
  int x;
  int g()
  {return B.f(x);}
  // comment
  int h()
  {return 0;}
}
class B{
  static int f(int x)
  {return x - 1;}
}
```

Running Example



```
class A {
  int x;
+ int g()
+ {return B.f(x);}
  // comment
  int h()
```

```
class A {
+ int x;
  // comment
  int g()
```

```
class A {
  static int f(int x) {
- {return x + 1;}
+ {return x - 1;}
}
```

```
  // comment
  int g() {
- {return 0;}
+ {return (new B()).y;}
  }
class B {
+ int y = 0;
  static int f(int x)
  {return x - 1;}
```

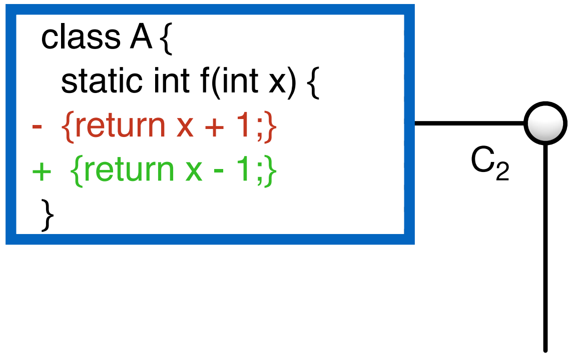
```
class A {
+ // comment
  int g()
  {return 0;}
```

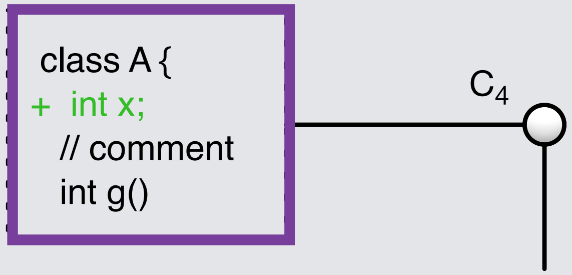
Test case:
a.g()==−1

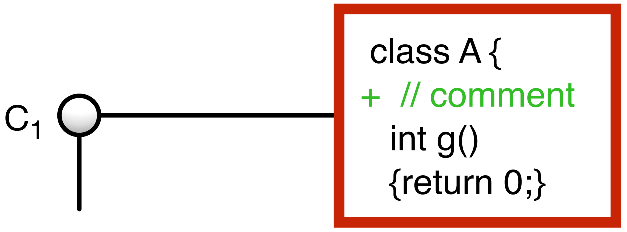
```
class TestA {
 public void t1()
 A a = new A();
 {assertEquals(-1, a.g();}
}

class A {
 int x;
 int g()
 {return B.f(x);}
 // comment
 int h()
 {return 0;}
}
class B{
 static int f(int x)
 {return x - 1;}
}
```

Dependency Hierarchy

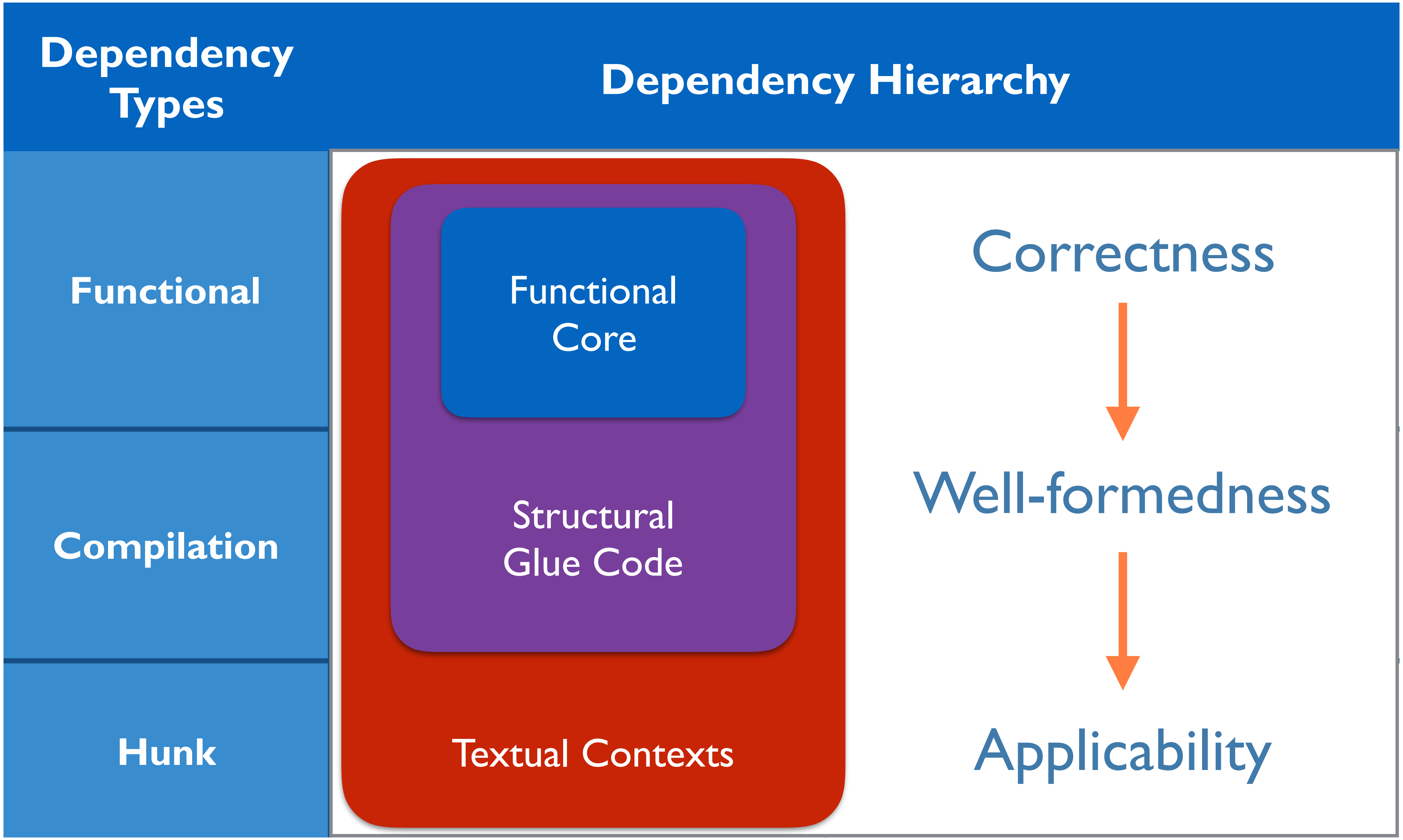
Dependency Types	Examples	Definitions

Functional	
	class A {
	static int f(int x) {
	- {return x + 1;}
	+ {return x - 1;}
	}
	C_2
		required for maintaining the semantic behaviours (e.g., pass the same tests)

Compilation	
	class A {
	+ int x;
	// comment
	int g()
	C_4
		required for maintaining the wellformedness of the program (e.g., free from compilation errors)

Hunk	 C_1 class A { + // comment int g() {return 0;}	specific to text-based version control systems (e.g., Git)

Dependency Hierarchy



Dependency Types

Dependency Hierarchy

Functional

Compilation

Hunk

Functional Core

Structural Glue Code

Textual Contexts

Correctness

Well-formedness

Applicability

Outline

1. Introduction
2. Dependency Hierarchy
3. CSlicer Algorithm
4. Evaluation
5. Related Work & Conclusion

CSlicer Overview

Input:

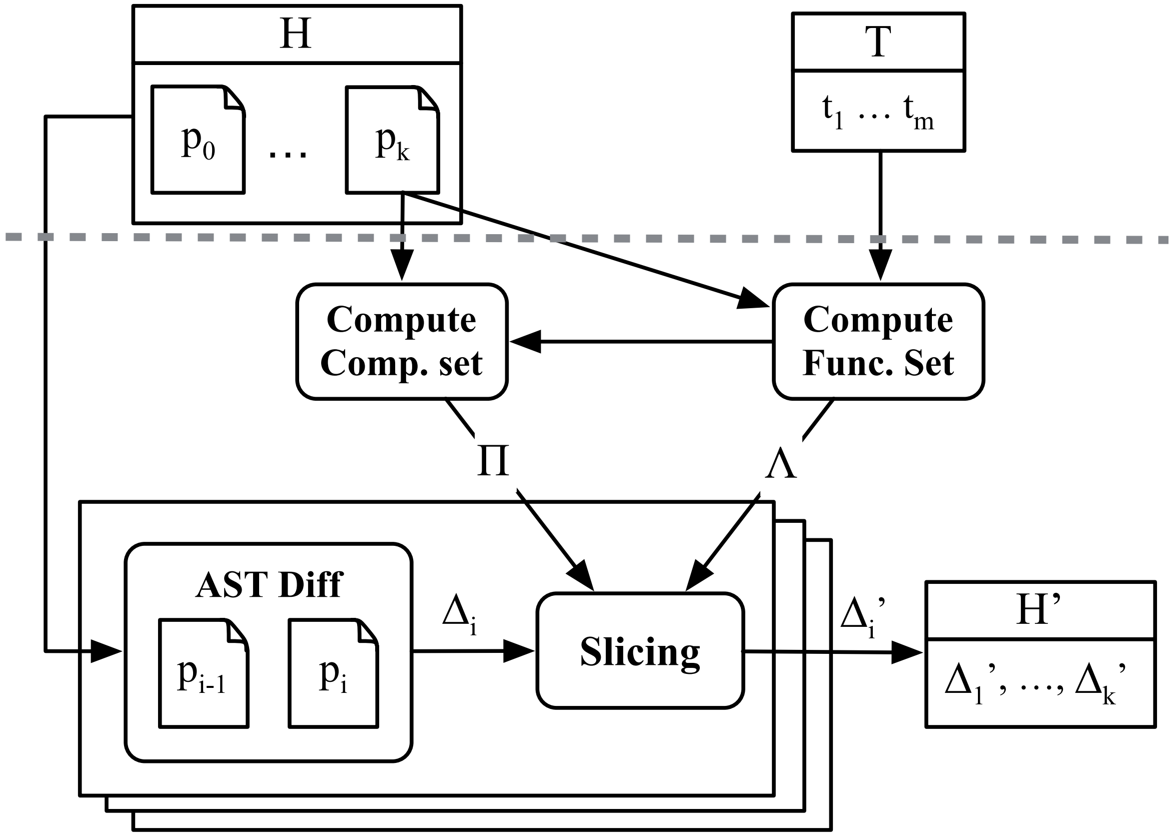
- $H = p_0 \ldots p_k$ *well-formed*
- $T = \{t_1, \ldots, t_m\}$ *tests for* p_k

Slicing core:

- FUNC set: Λ
- COMP set: Π
- Slicer(Λ, Π, Δ_i) = Δ_i'

Output:

- $H' = <\Delta_1', \ldots, \Delta_k'>$ *slice*



CSlicer Overview

Input:

- $H = p_0 \ldots p_k$ *well-formed*
- $T = \{t_1, \ldots, t_m\}$ *tests for* p_k

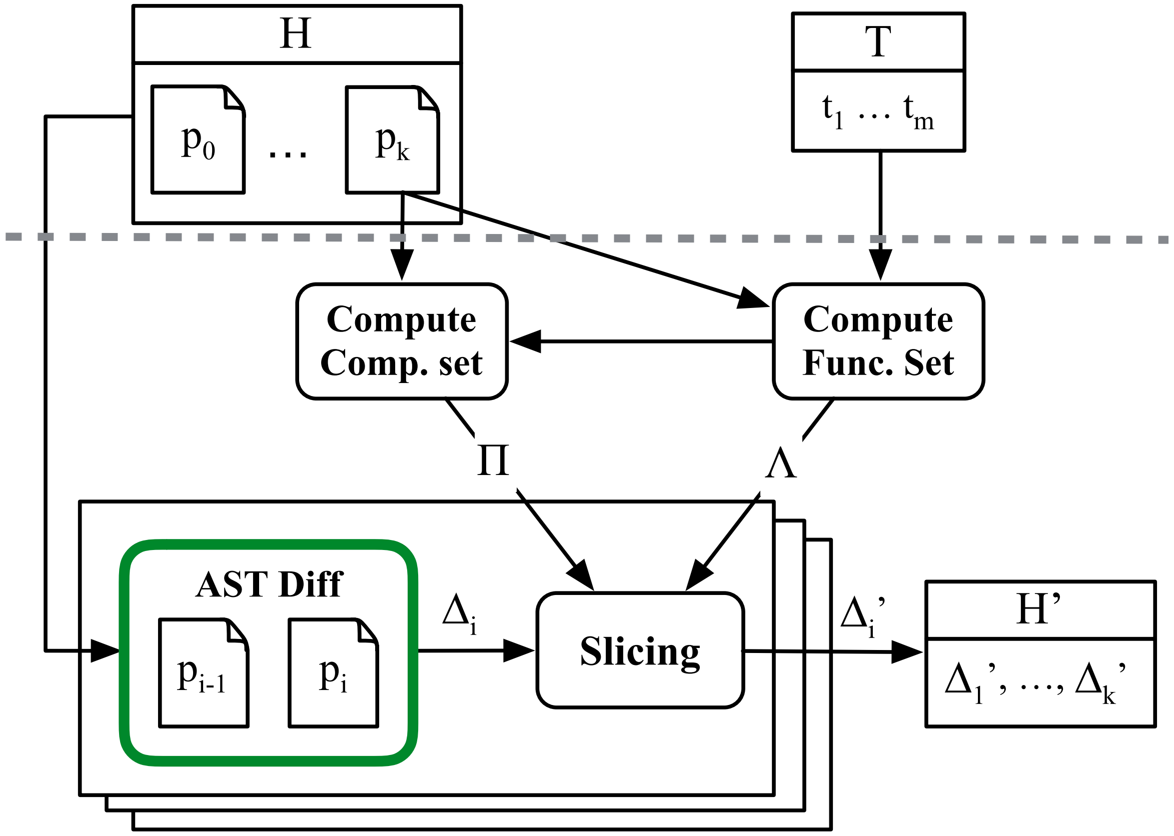
Slicing core:

- FUNC set: Λ
- COMP set: Π
- Slicer(Λ, Π, Δ_i) = Δ_i'

Output:

- $H' = <\Delta_1', \ldots, \Delta_k'>$ *slice*

1. AST differencing



CSlicer Overview

Input:

- $H = p_0 \ldots p_k$ *well-formed*
- $T = \{t_1, \ldots, t_m\}$ *tests for* p_k

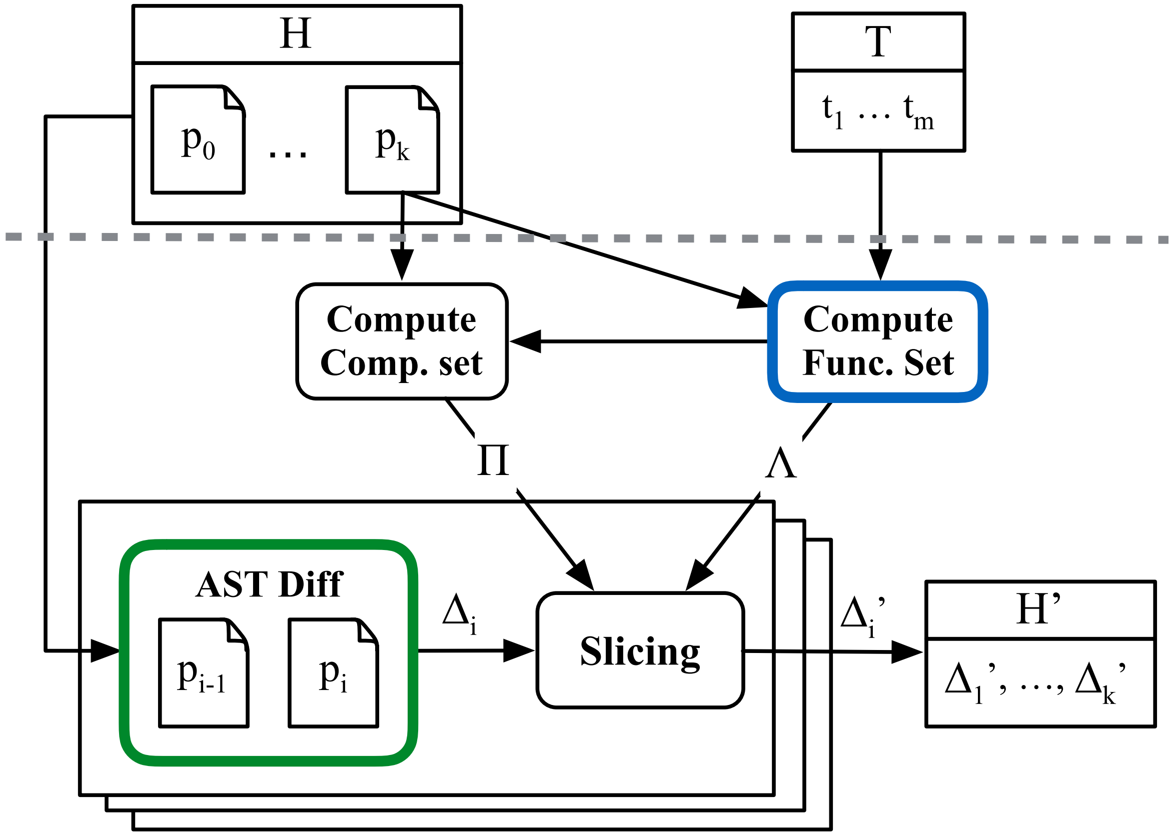
Slicing core:

- FUNC set: Λ
- COMP set: Π
- Slicer$(\Lambda, \Pi, \Delta_i) = \Delta_i'$

Output:

- $H' = <\Delta_1', \ldots, \Delta_k'>$ *slice*

1. AST differencing
2. Compute Functional set



CSlicer Overview

Input:

- $H = p_0 \ldots p_k$ *well-formed*
- $T = \{t_1, \ldots, t_m\}$ *tests for p_k*

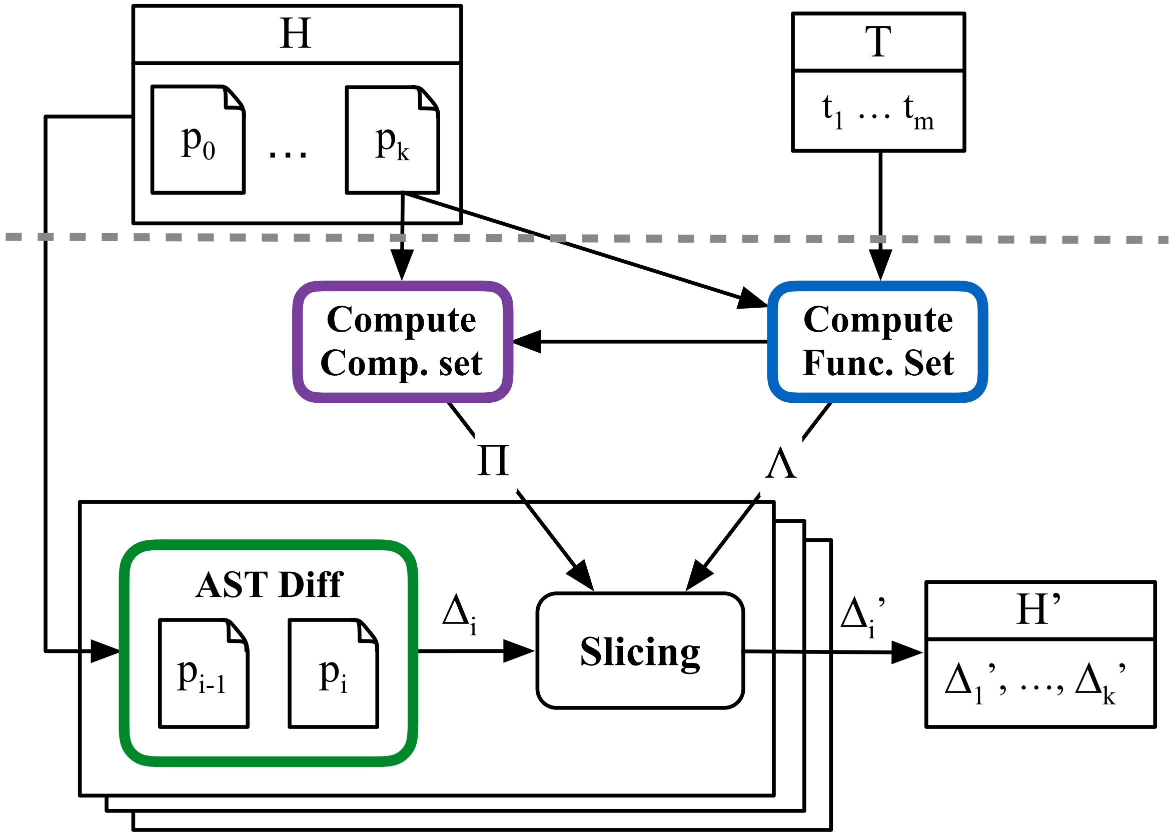
Slicing core:

- FUNC set: Λ
- COMP set: Π
- Slicer(Λ, Π, Δ_i) = Δ_i'

Output:

- $H' = <\Delta_1', \ldots, \Delta_k'>$ *slice*

1. AST differencing
2. Compute Functional set
3. Compute Compilation set



CSlicer Overview

Input:

- $H = p_0 \ldots p_k$ *well-formed*
- $T = \{t_1, \ldots, t_m\}$ *tests for* p_k

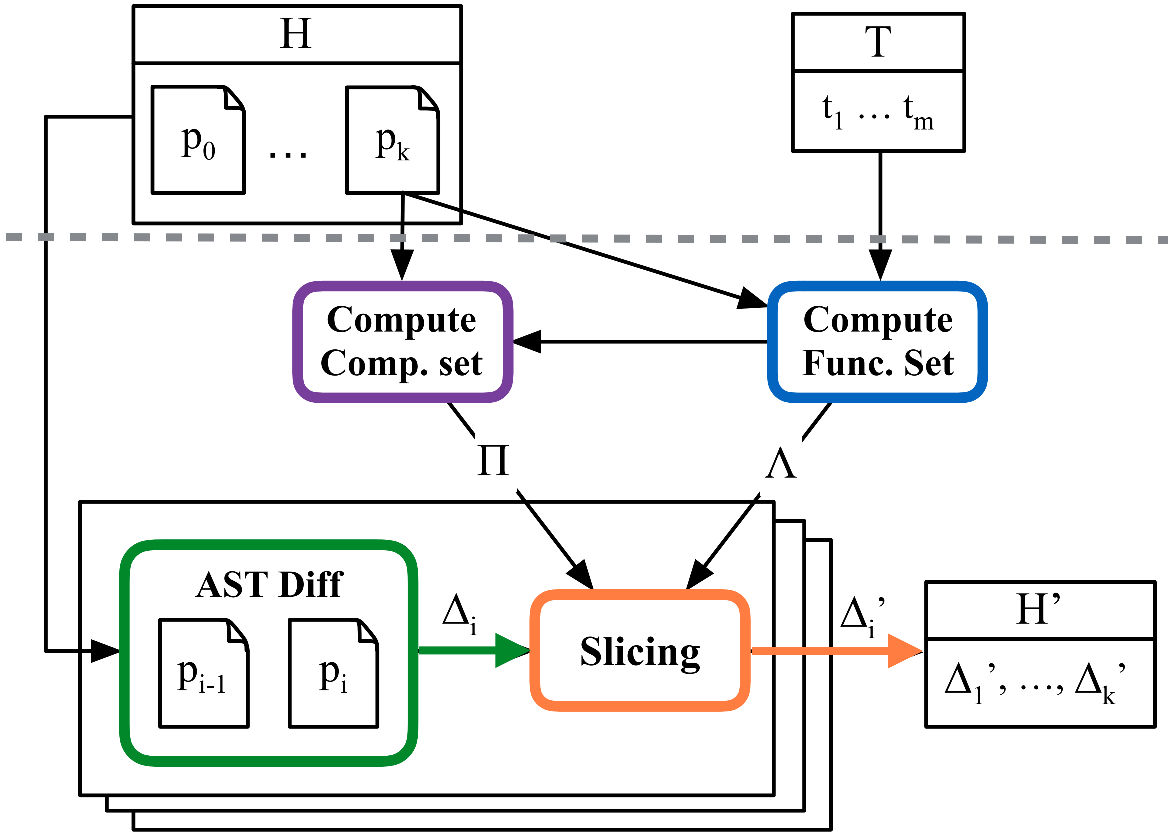
Slicing core:

- FUNC set: Λ
- COMP set: Π
- Slicer$(\Lambda, \Pi, \Delta_i) = \Delta_i'$

Output:

- $H' = <\Delta_1', \ldots, \Delta_k'>$ *slice*

1. AST differencing
2. Compute Functional set
3. Compute Compilation set
4. Changeset Slicing

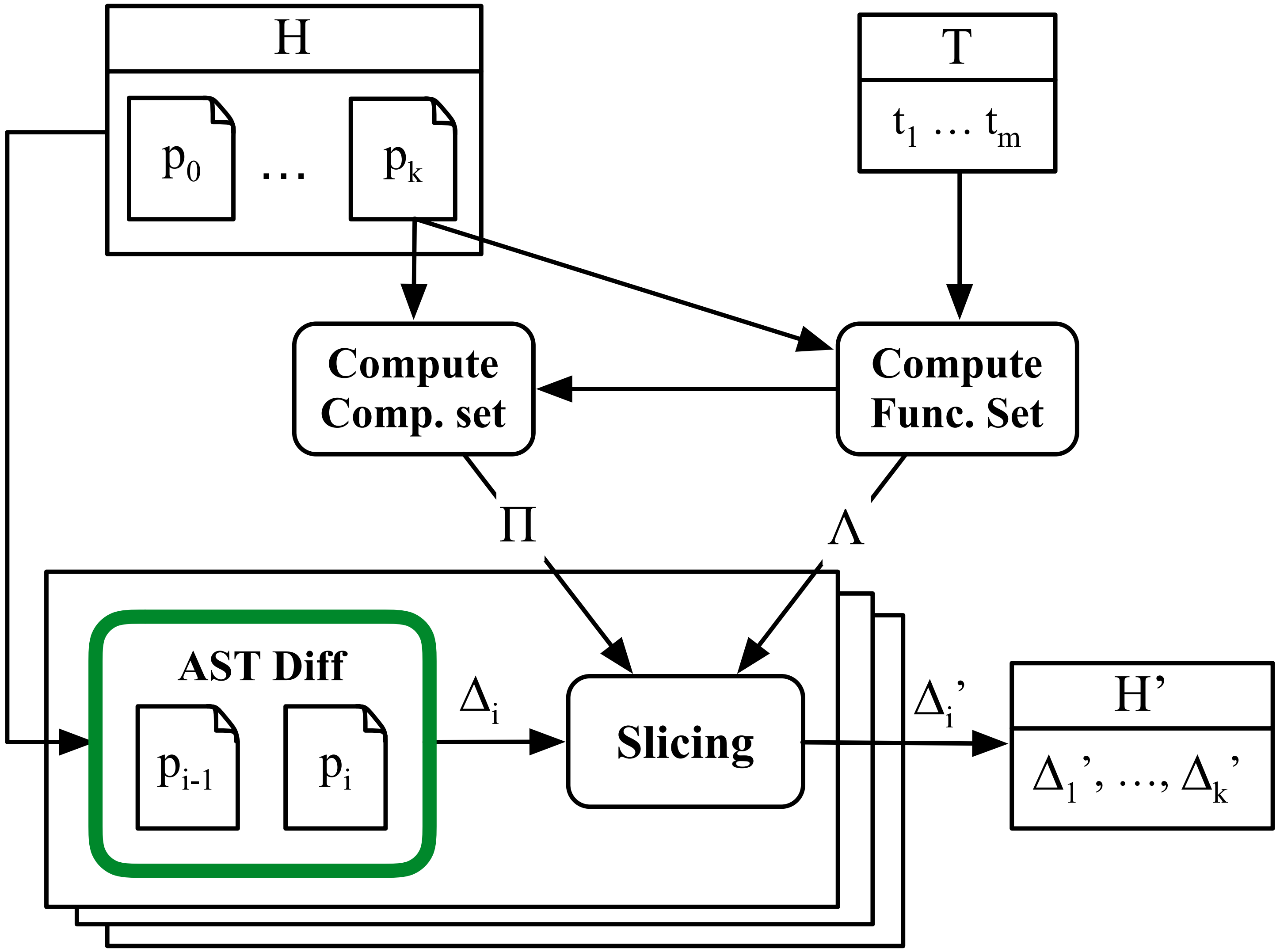


Language Model

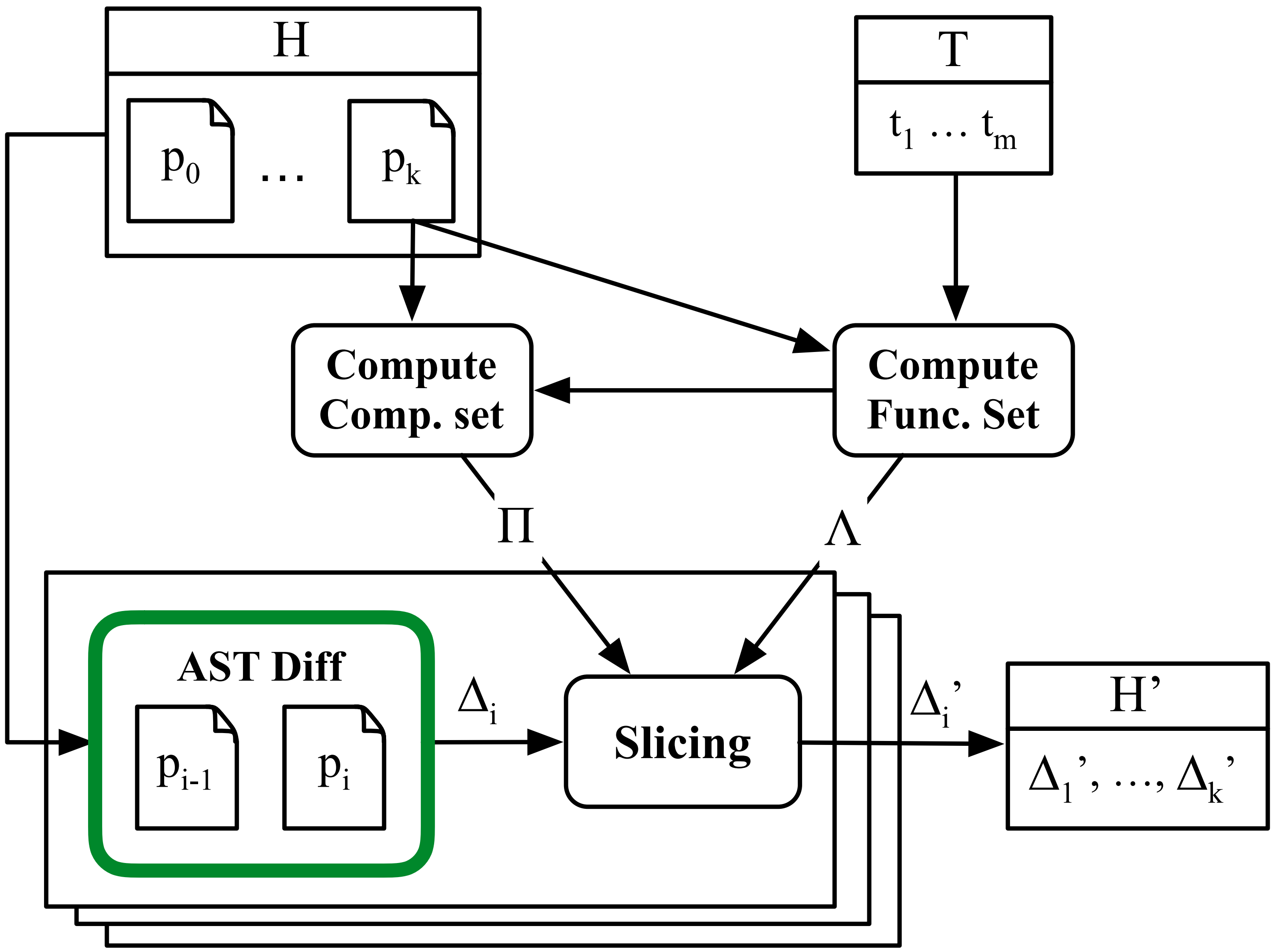
Simplified language model:

- Featherweight Java [Igarashi et al., ACM TOPLAS'01]
- Core object-oriented features and type system
- No reflection, abstract class, etc.
- Advanced Java features can be handled as algorithmic extensions

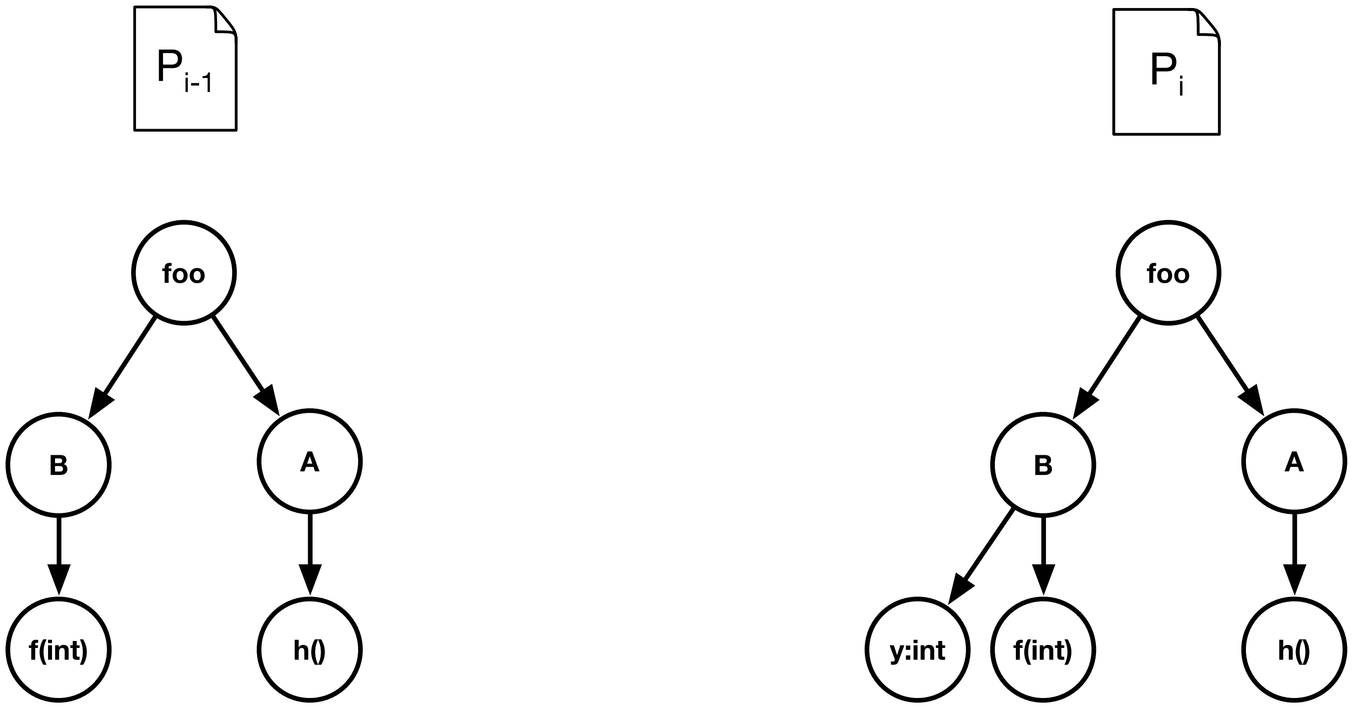
AST Differencing



AST Differencing



AST Differencing



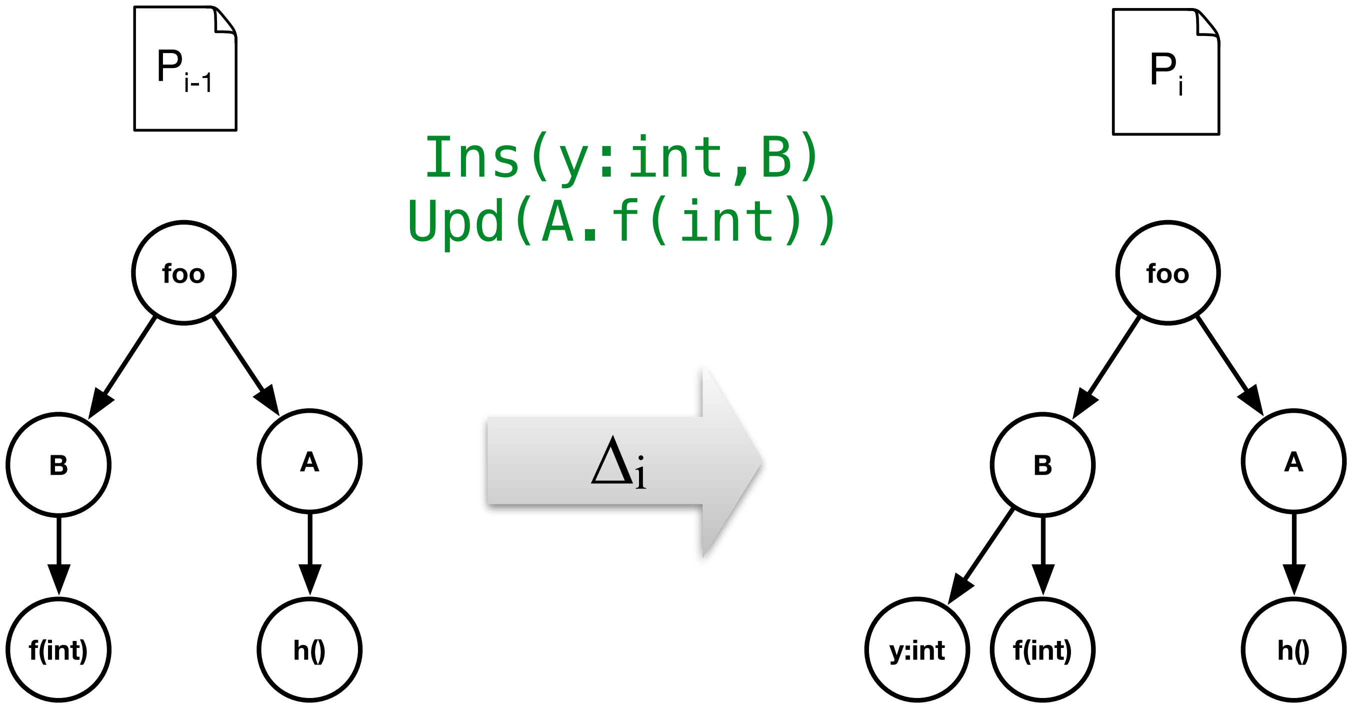
Edit Operations:

+ Ins((x,n,v),y)
– Del(x)
∗ Upd(x,v)

Compare two abstract syntax trees:

- Ignore cosmetic changes; match on unique names
- Focus on structural nodes (class, method, and field)
- Structural differencing [Fluri et al., IEEE TSE'07]

AST Differencing



```
Ins(y:int,B)
Upd(A.f(int))
```

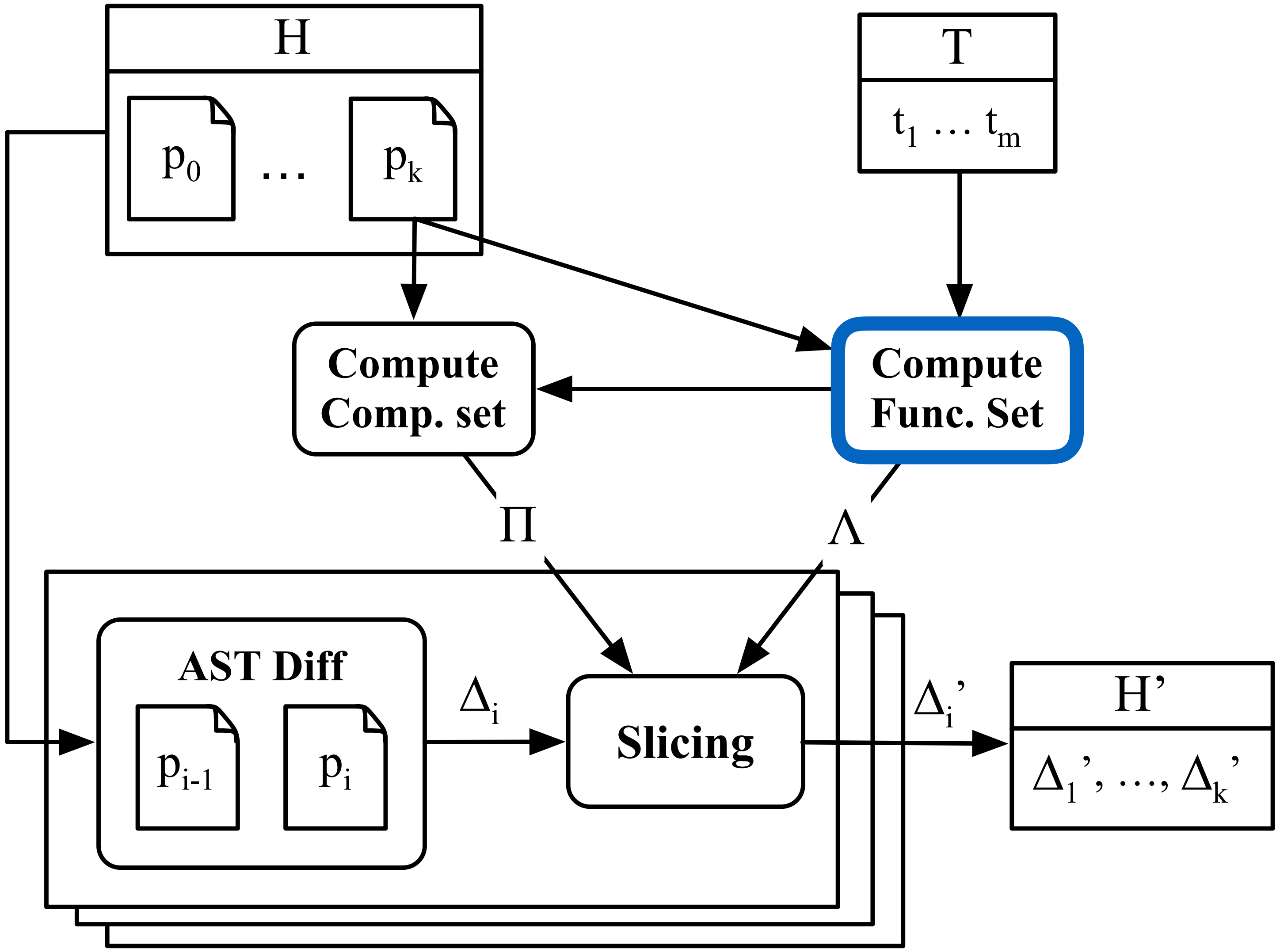
Edit Operations:

+ Ins((x,n,v),y)
− Del(x)
∗ Upd(x,v)

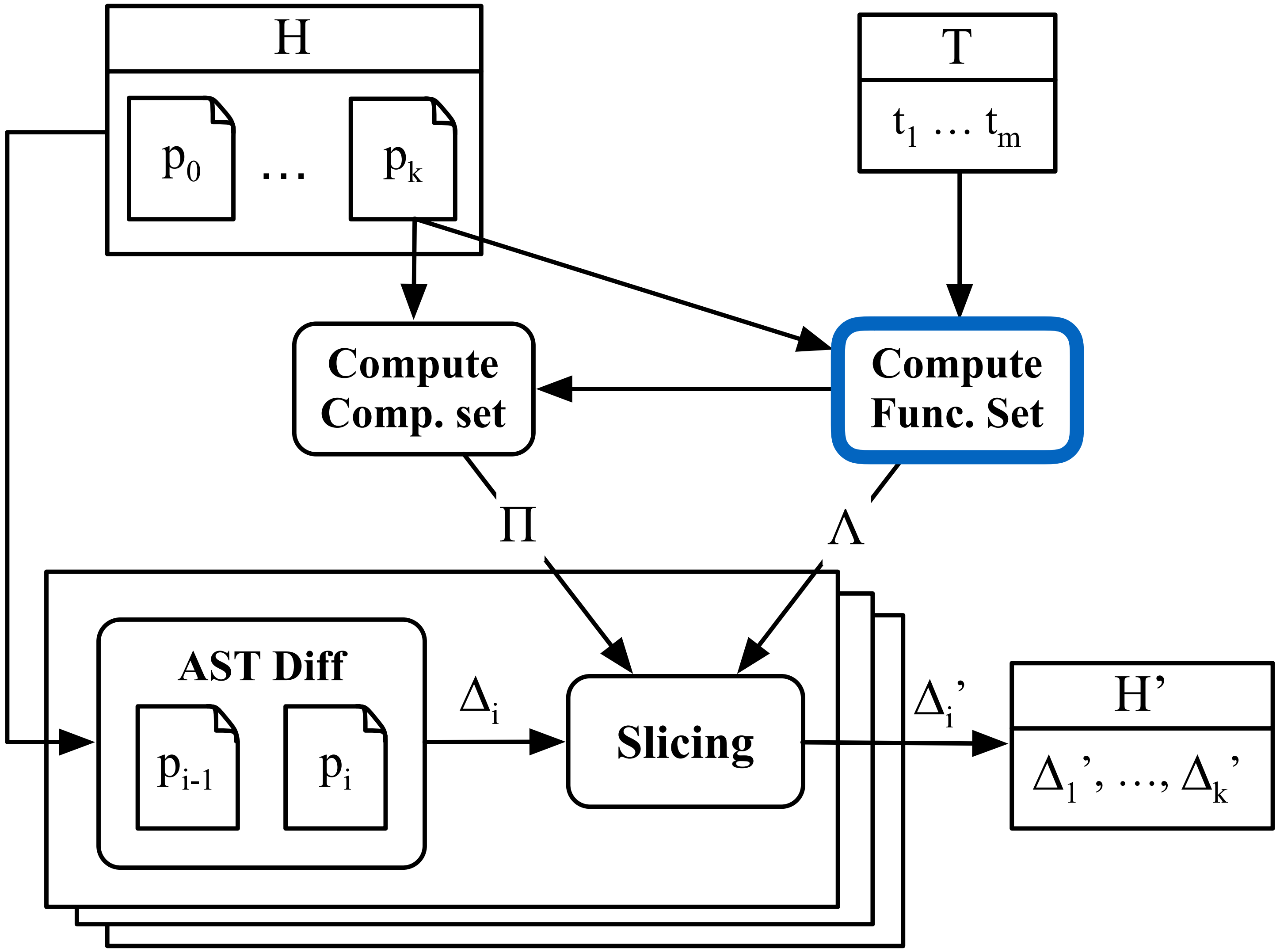
Compare two abstract syntax trees:

- Ignore cosmetic changes; match on unique names
- Focus on structural nodes (class, method, and field)
- Structural differencing [Fluri et al., IEEE TSE'07]

Compute Functional Set



Compute Functional Set

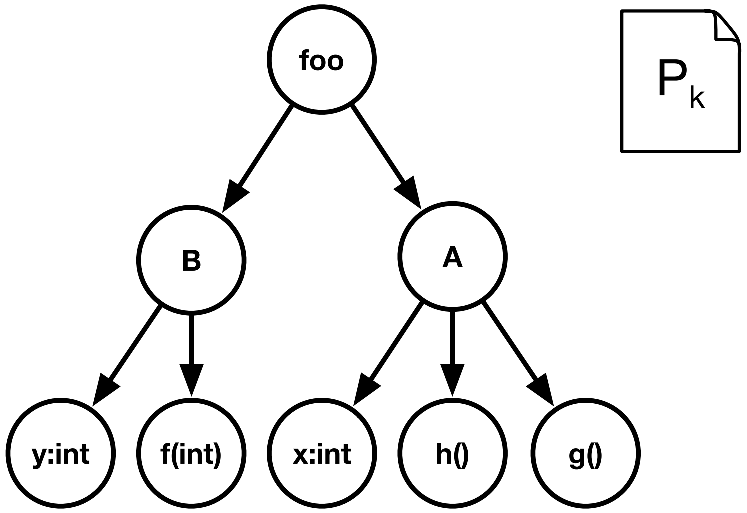


Compute Functional Set

Functional Set:

- Nodes directly traversed during test execution
- Dynamic analysis
- Ensure functional correctness

```
class A {
  int x;
  int g()
  {return B.f(x);}
  // comment
  int h()
  {return (new B()).y;}
}
class B {
  int y = 0;
  static int f(int x)
  {return x - 1;}
}
```



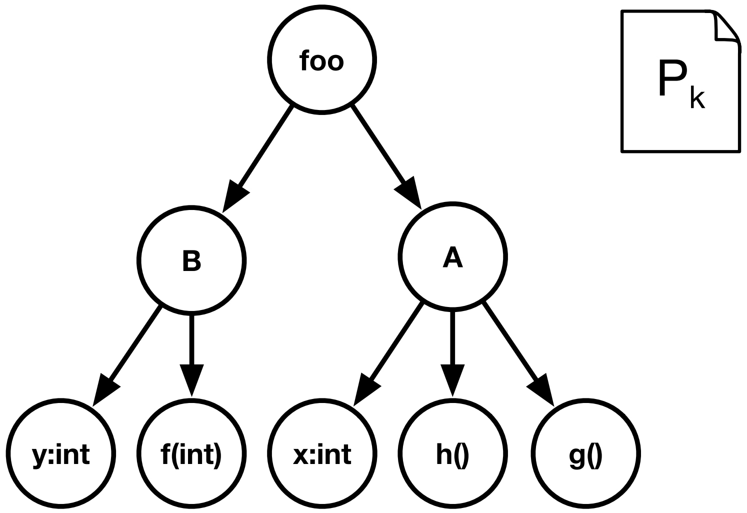
Compute Functional Set

Functional Set:

- Nodes directly traversed during test execution
- Dynamic analysis
- Ensure functional correctness

```
class A {
  int x;
  int g()
  {return B.f(x);}
  // comment
  int h()
  {return (new B()).y;}
}
class B {
  int y = 0;
  static int f(int x)
  {return x - 1;}
}
```

Test case:
`a.g()==-1`



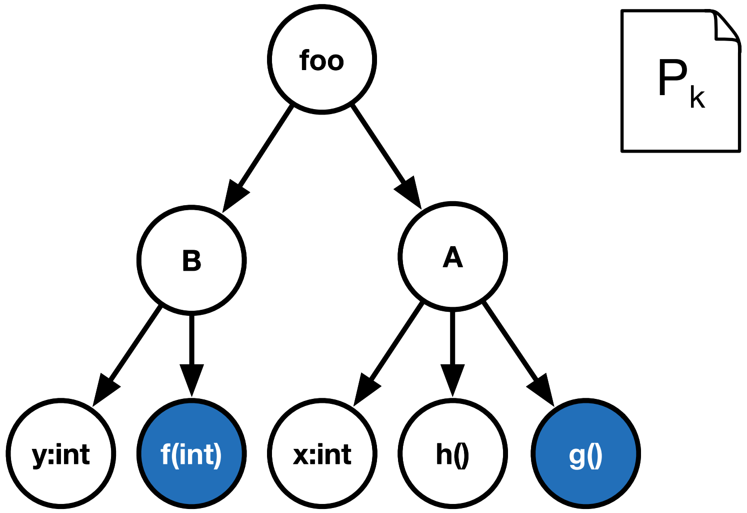
Compute Functional Set

Functional Set:

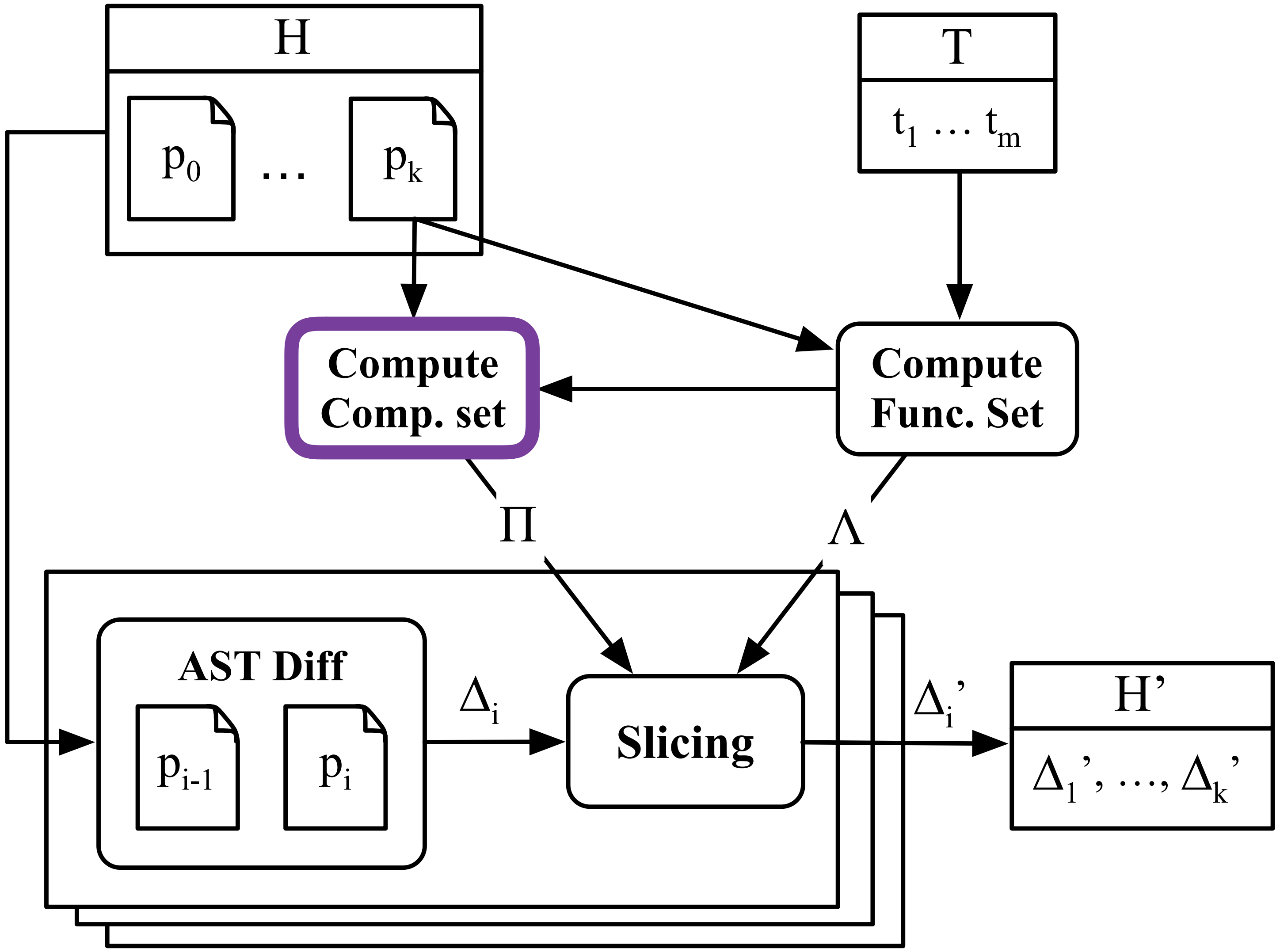
- Nodes directly traversed during test execution
- Dynamic analysis
- Ensure functional correctness

```
class A {
  int x;
  int g()
  {return B.f(x);}
  // comment
  int h()
  {return (new B()).y;}
}
class B {
  int y = 0;
  static int f(int x)
  {return x - 1;}
}
```

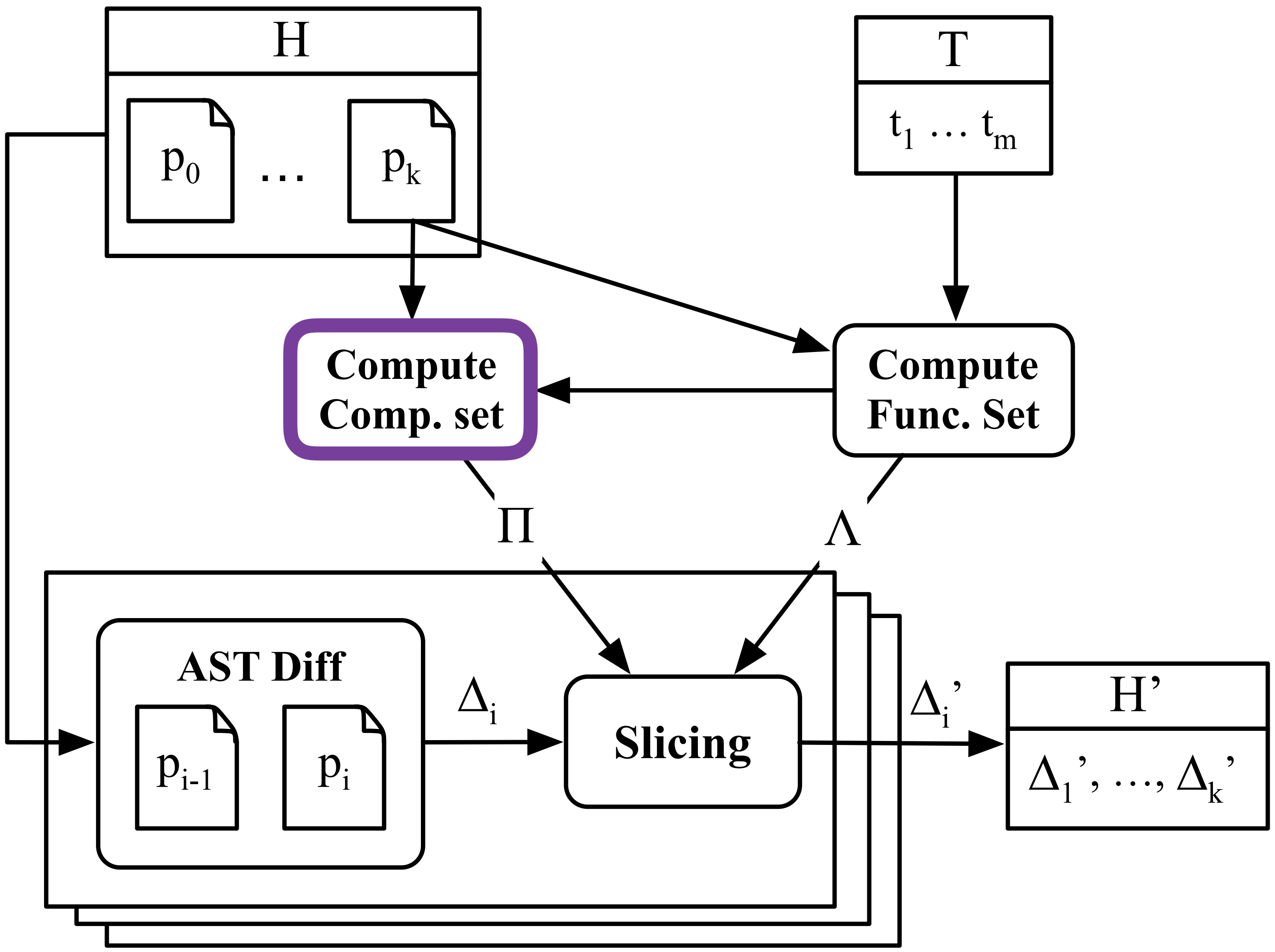
Test case:
a.g()==-1



Compute Compilation Set



Compute Compilation Set



Compute Compilation Set

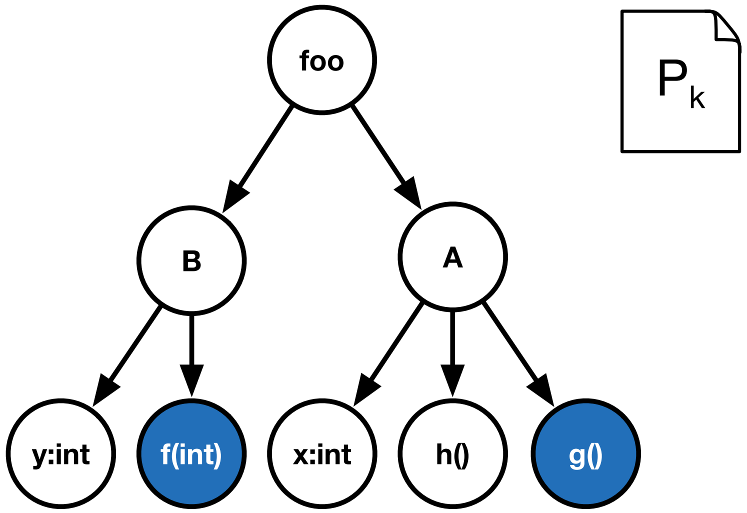
Compilation Set:

- Nodes referenced by the functional set
- Static analysis
- Ensure type safety

Inference Rules:

- Enclosing classes should exist
- Accessed fields should exist
- etc.

```
class A {
 int x;
 int g()
 {return B.f(x);}
 // comment
 int h()
 {return (new B()).y;}
}
class B {
 int y = 0;
 static int f(int x)
 {return x - 1;}
}
```



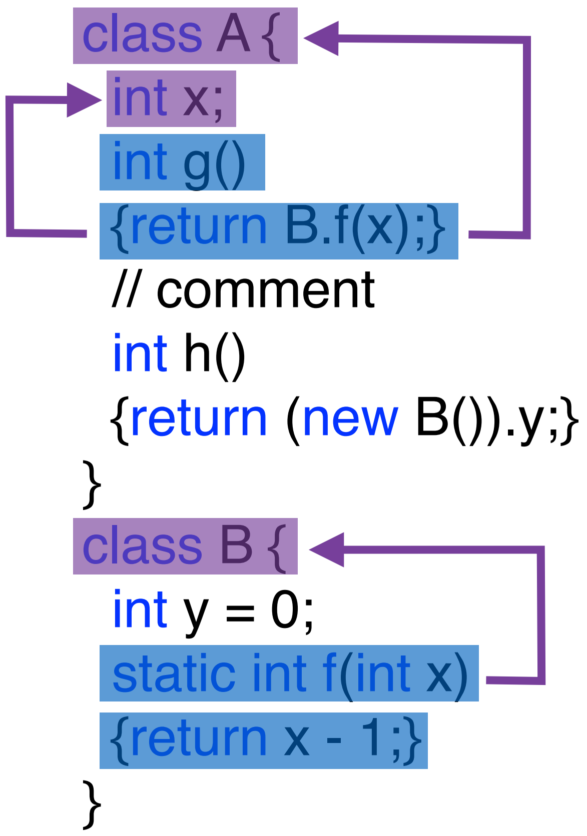
Compute Compilation Set

Compilation Set:

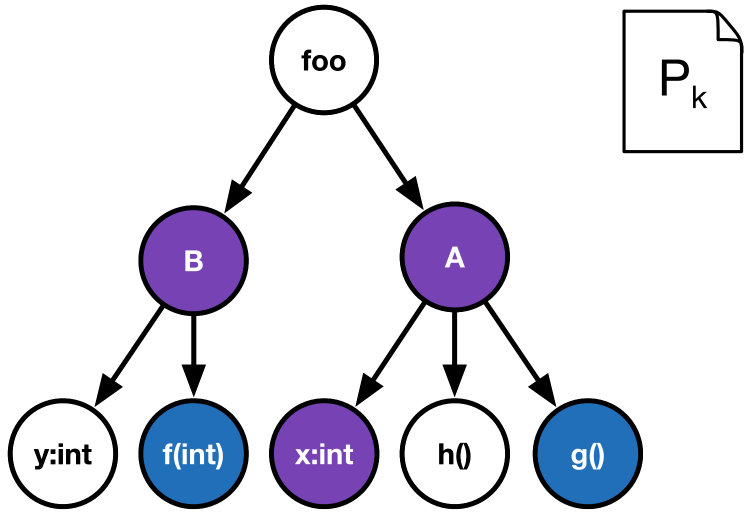
- Nodes referenced by the functional set
- Static analysis
- Ensure <u>type safety</u>

Inference Rules:

- Enclosing classes should exist
- Accessed fields should exist
- etc.



```
class A {
  int x;
  int g()
  {return B.f(x);}
  // comment
  int h()
  {return (new B()).y;}
}
class B {
  int y = 0;
  static int f(int x)
  {return x - 1;}
}
```



Compute Compilation Set

$$\frac{C <: D \quad C \in \Pi}{D \in \Pi}\,[\text{L.1}] \qquad \frac{f : C \in \Pi}{C \in \Pi}\,[\text{L.2}] \qquad \frac{C(\overline{D\ f})\{\texttt{super}(\overline{f})\texttt{;}\overline{\texttt{this}.f = f\texttt{;}}\} \in \Pi}{C \in \Pi \quad \overline{D} \in \Pi \quad \overline{f} \in \Pi}\,[\text{K1}]$$

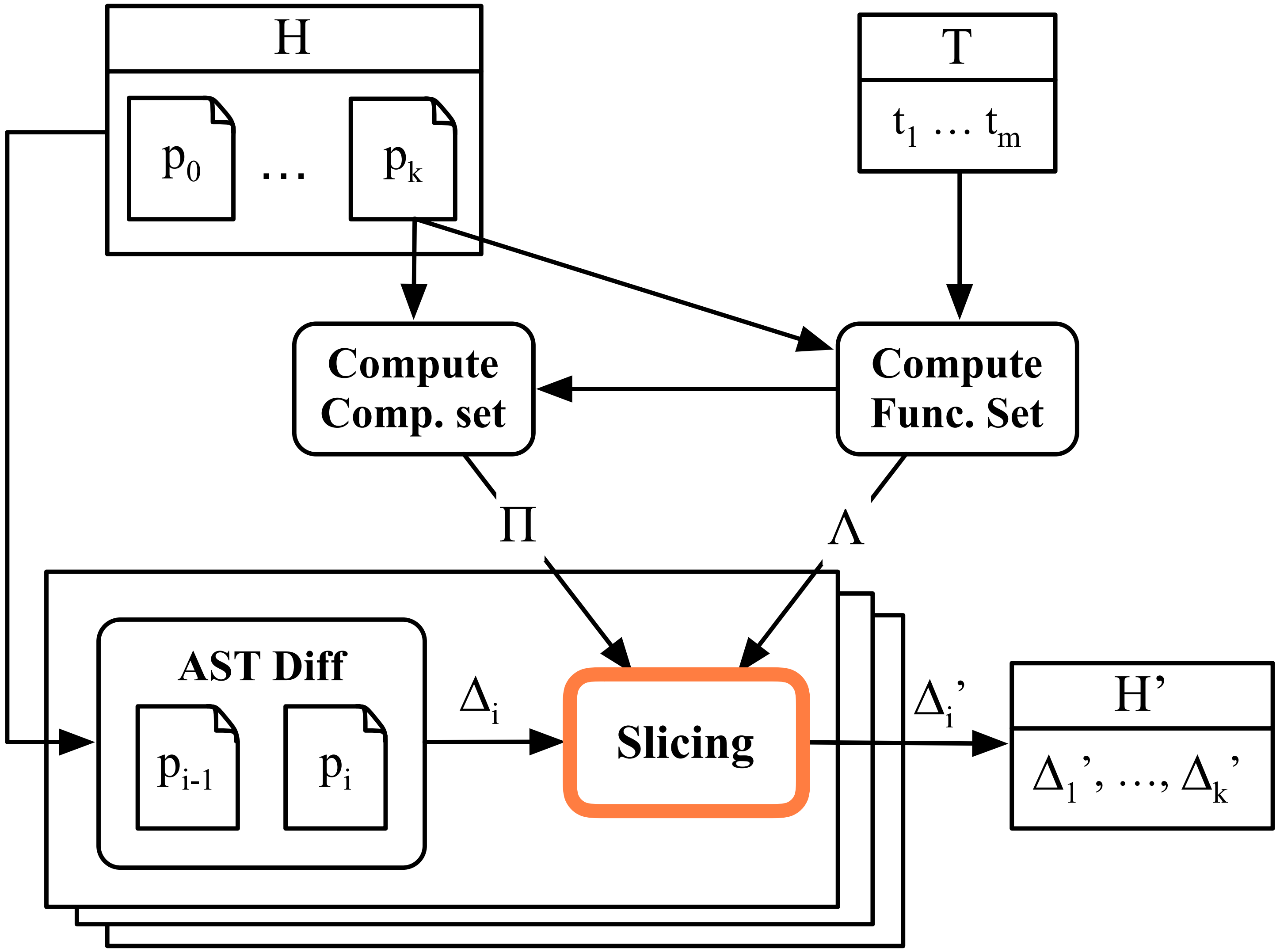
$$\frac{C\ \ m(\overline{D\ \ x})\{\texttt{return}\ \ e\texttt{;}\} \in \Pi}{C \in \Pi \quad \overline{D} \in \Pi}\,[\text{M1}] \qquad \frac{\ldots\{\texttt{return}\ \ e.f\texttt{;}\} \in \Pi}{f \in \Pi}\,[\text{E1}] \qquad \frac{\ldots\{\texttt{return}\ \ e.m(\overline{e})\texttt{;}\} \in \Pi}{m \in \Pi}\,[\text{E2}]$$

$$\frac{\ldots\{\texttt{return}\ \ \texttt{new}\ \ C(\overline{e})\texttt{;}\} \in \Pi}{C \in \Pi}\,[\text{E3}] \qquad \frac{\ldots\{\texttt{return}\ \ (C)e\texttt{;}\} \in \Pi}{C \in \Pi}\,[\text{E4}] \qquad \frac{x \in \Pi}{\textsc{Parent}(x) \in \Pi}\,[\text{P1}] \qquad \frac{x \in \Lambda}{x \in \Pi}\,[\text{T1}]$$

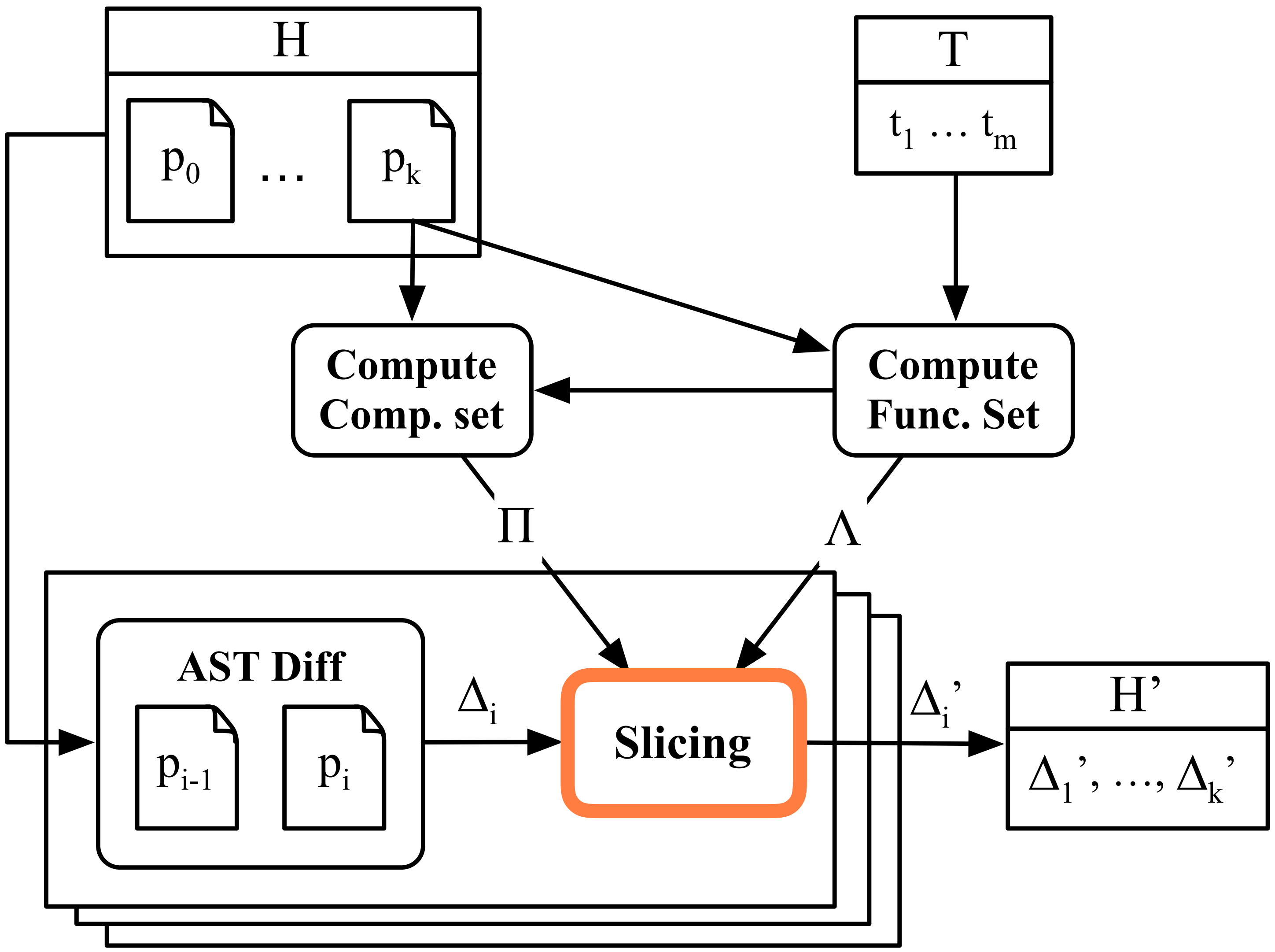
Inference Rules:

- Based on [Kastner & Apel, ASE'08]
- Tailored for method-field level granularity
- Complete for our language model

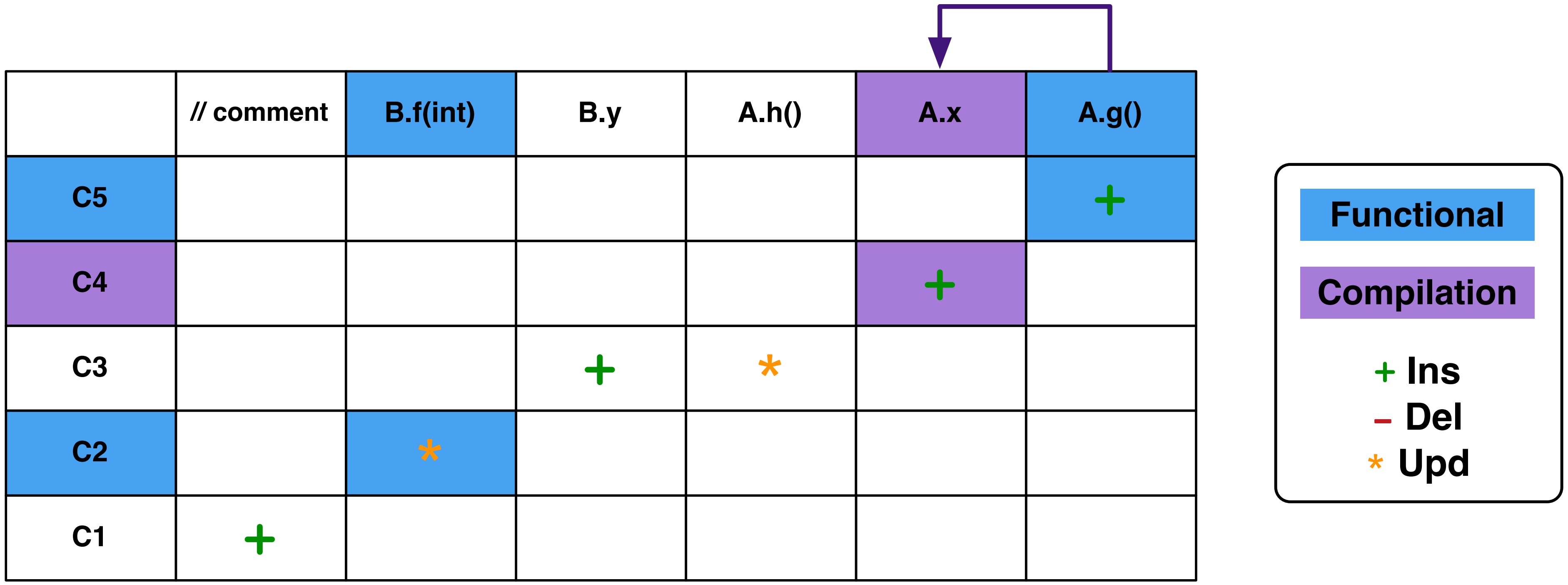
Changeset Slicing



Changeset Slicing



Changeset Slicing



	// comment	B.f(int)	B.y	A.h()	A.x	A.g()
C5						+
C4					+	
C3			+	*		
C2		*				
C1	+					

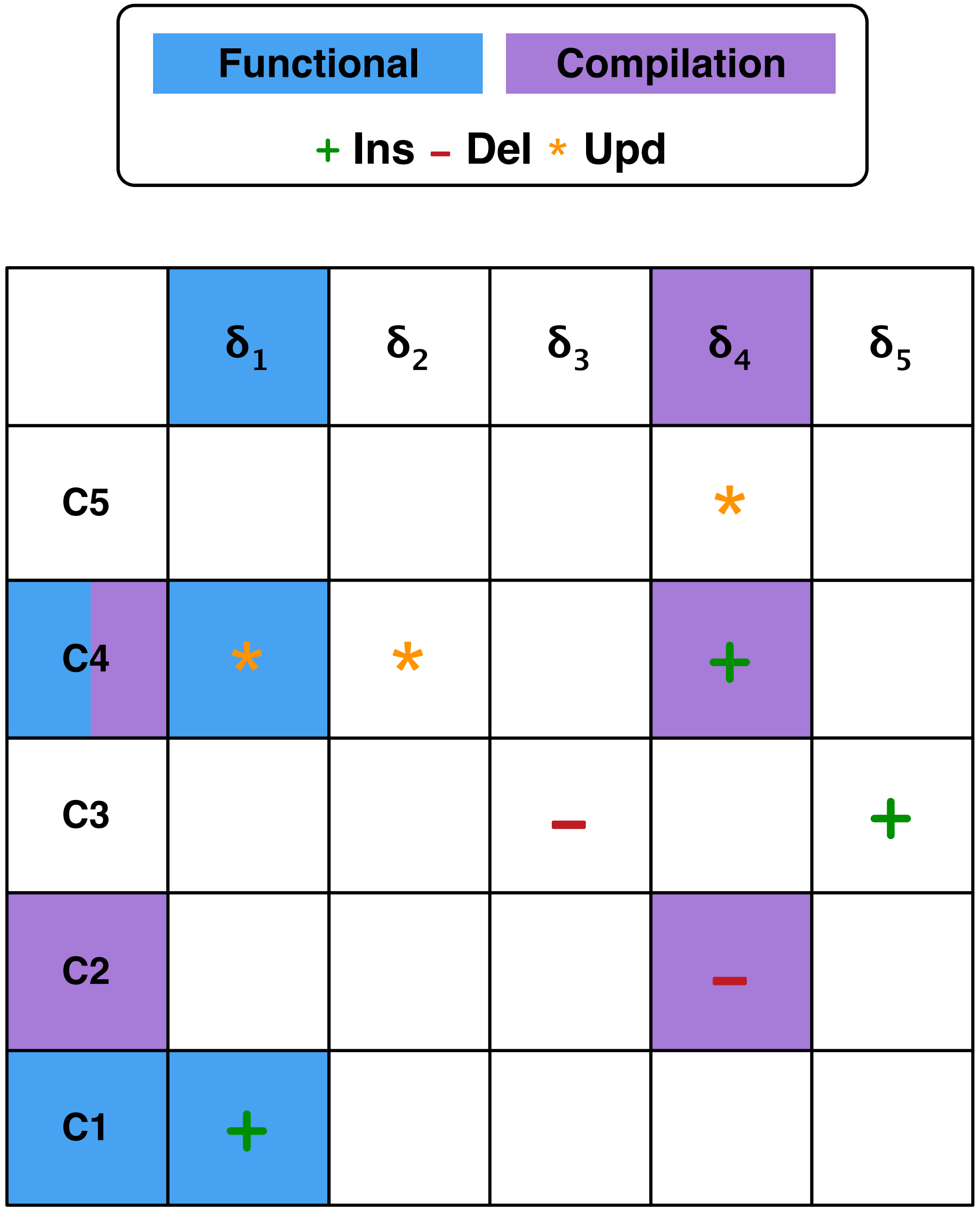
<u>Change Matrix</u>: maps *atomic changes* to commits

- Cells are marked by change types
- Atomic changes are color coded

Changeset Slicing

General Slicing Rules:

- Keep blue cells
- Keep purple +, –
- Drop white – unless affecting method lookup



Functional Compilation

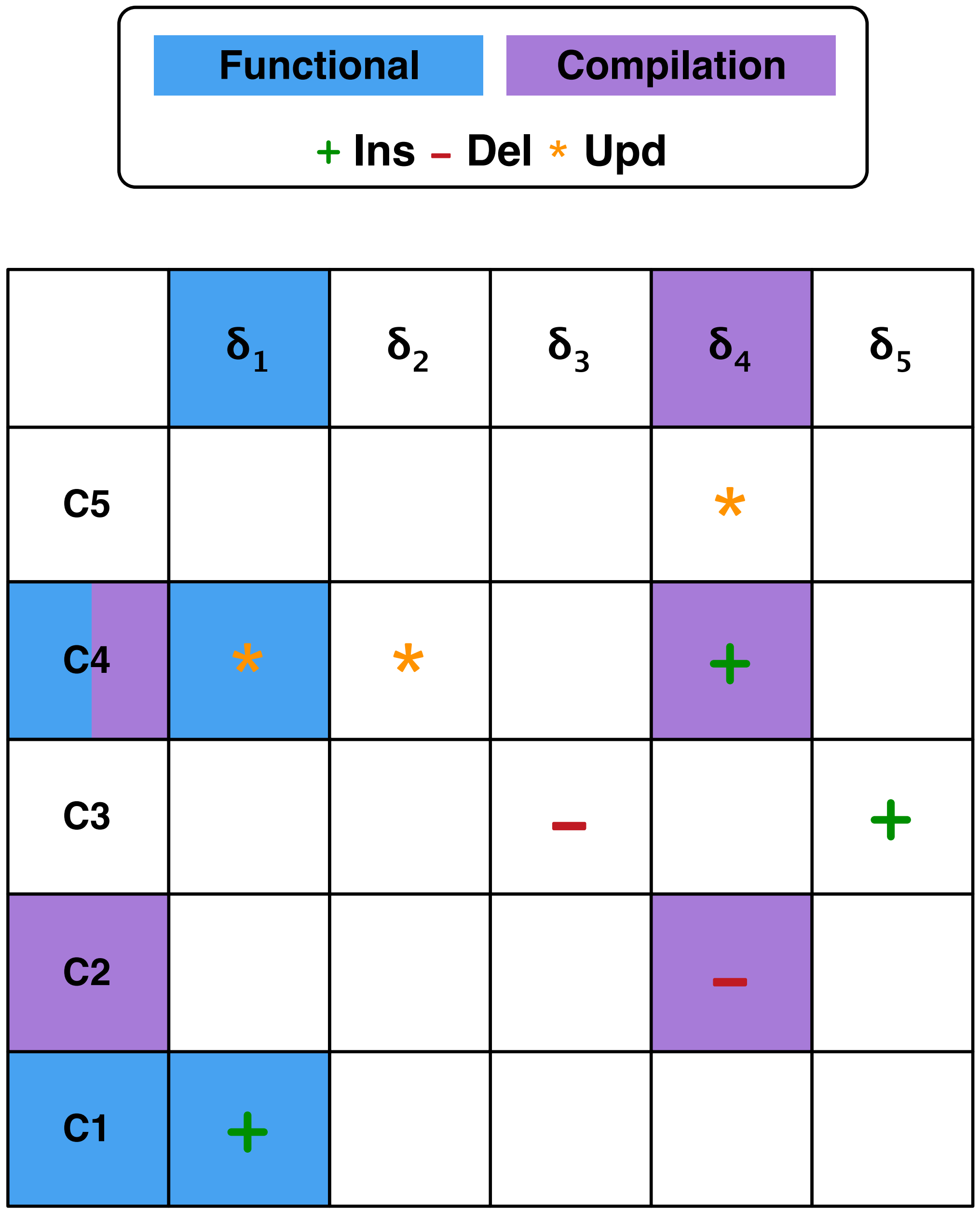
+ Ins – Del * Upd

	δ_1	δ_2	δ_3	δ_4	δ_5
C5				*	
C4	*	*		+	
C3			–		+
C2				–	
C1	+				

Changeset Slicing

Side-effects (Git):

- Keeping original commit
- Dependencies between white cells
- Detection and resolution



	δ_1	δ_2	δ_3	δ_4	δ_5
C5				*	
C4	*	*		+	
C3			–		+
C2				–	
C1	+				

Changeset Slicing

🟦 Functional 🟪 Compilation

+ Ins – Del * Upd

Side-effects (Git):

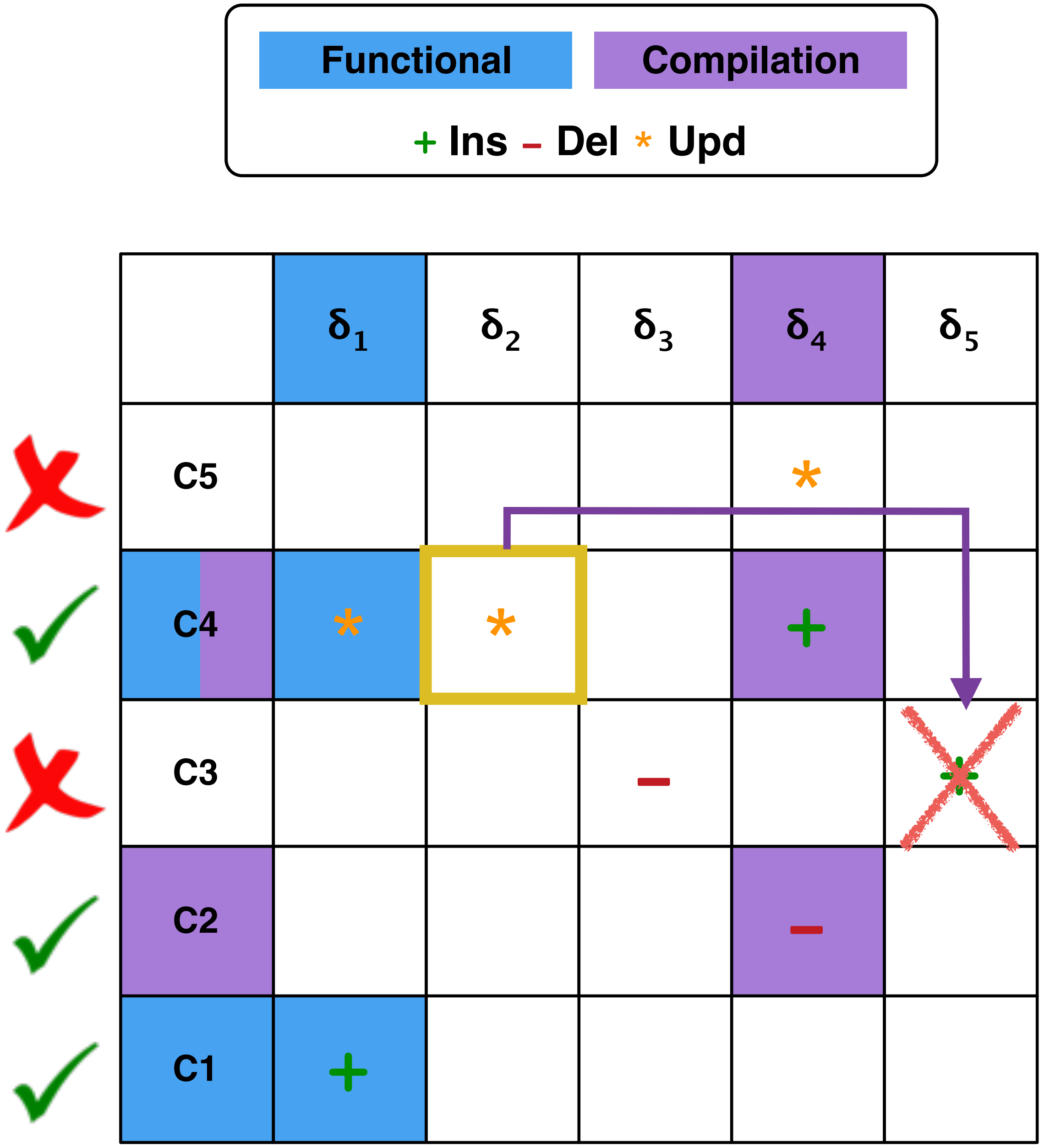
- Keeping original commit
- Dependencies between white cells
- Detection and resolution

		🟦 δ_1	δ_2	δ_3	🟪 δ_4	δ_5
✗	C5				*	
✓	🟦🟪 C4	🟦 *	*		🟪 +	
✗	C3			–		+
✓	🟪 C2				🟪 –	
✓	🟦 C1	🟦 +				

Changeset Slicing

Side-effects (Git):

- Keeping original commit
- Dependencies between white cells
- Detection and resolution



		δ_1	δ_2	δ_3	δ_4	δ_5
✗	C5				*	
✓	C4	*	*		+	
✗	C3			–		~~+~~
✓	C2				–	
✓	C1	+				

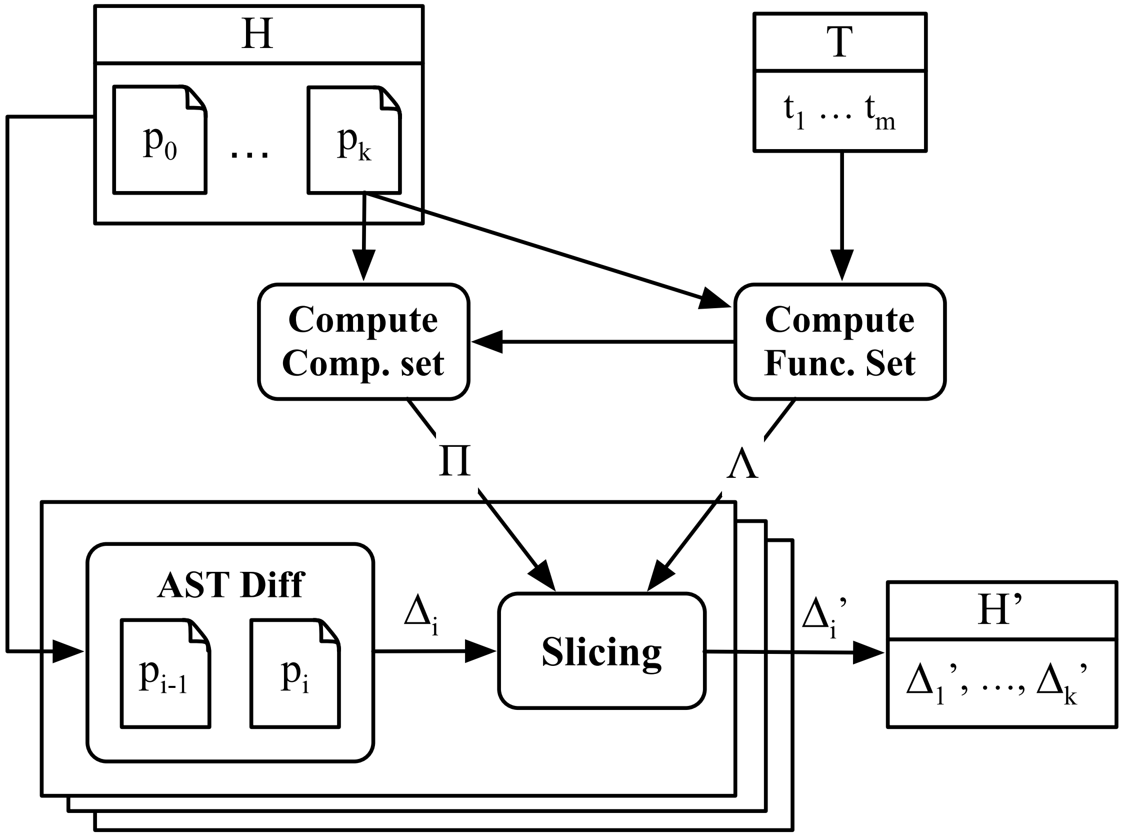
Changeset Slicing

Side-effects (Git):

- Keeping original commit
- Dependencies between white cells
- Detection and resolution

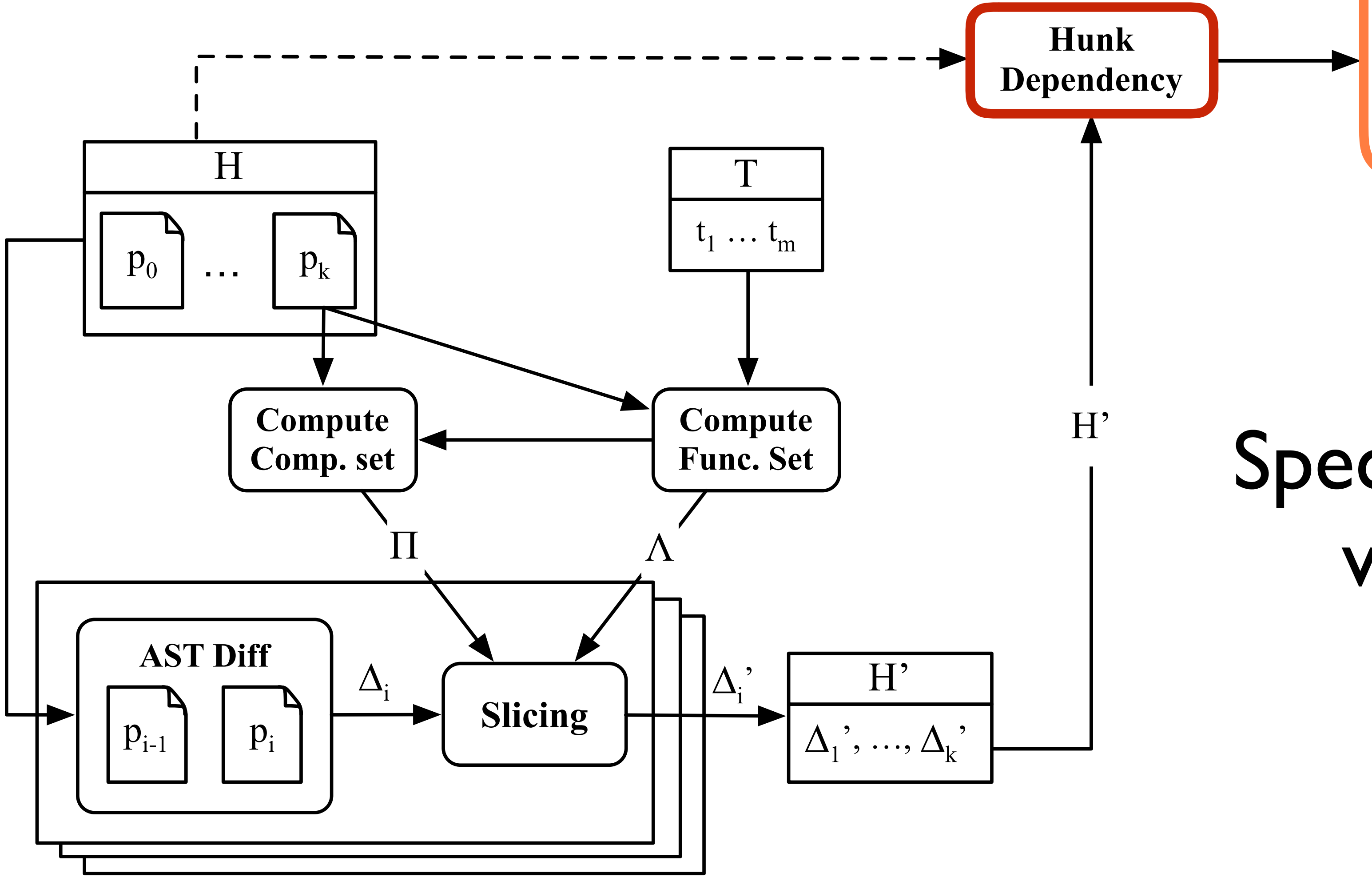
		δ_1	δ_2	δ_3	δ_4	δ_5
	CN		~~*~~			
✗	C5				*	
✓	C4	*	~~*~~		+	
✗	C3			–		~~+~~
✓	C2				–	
✓	C1	+				

Hunk Dependency



Hunk Dependency

HunkDeps(H’)



Applicable history slice

Specific to text-based version control

Outline

1. Introduction
2. Dependency Hierarchy
3. CSlicer Algorithm
4. Evaluation
5. Related Work & Conclusion

Evaluation

Research questions

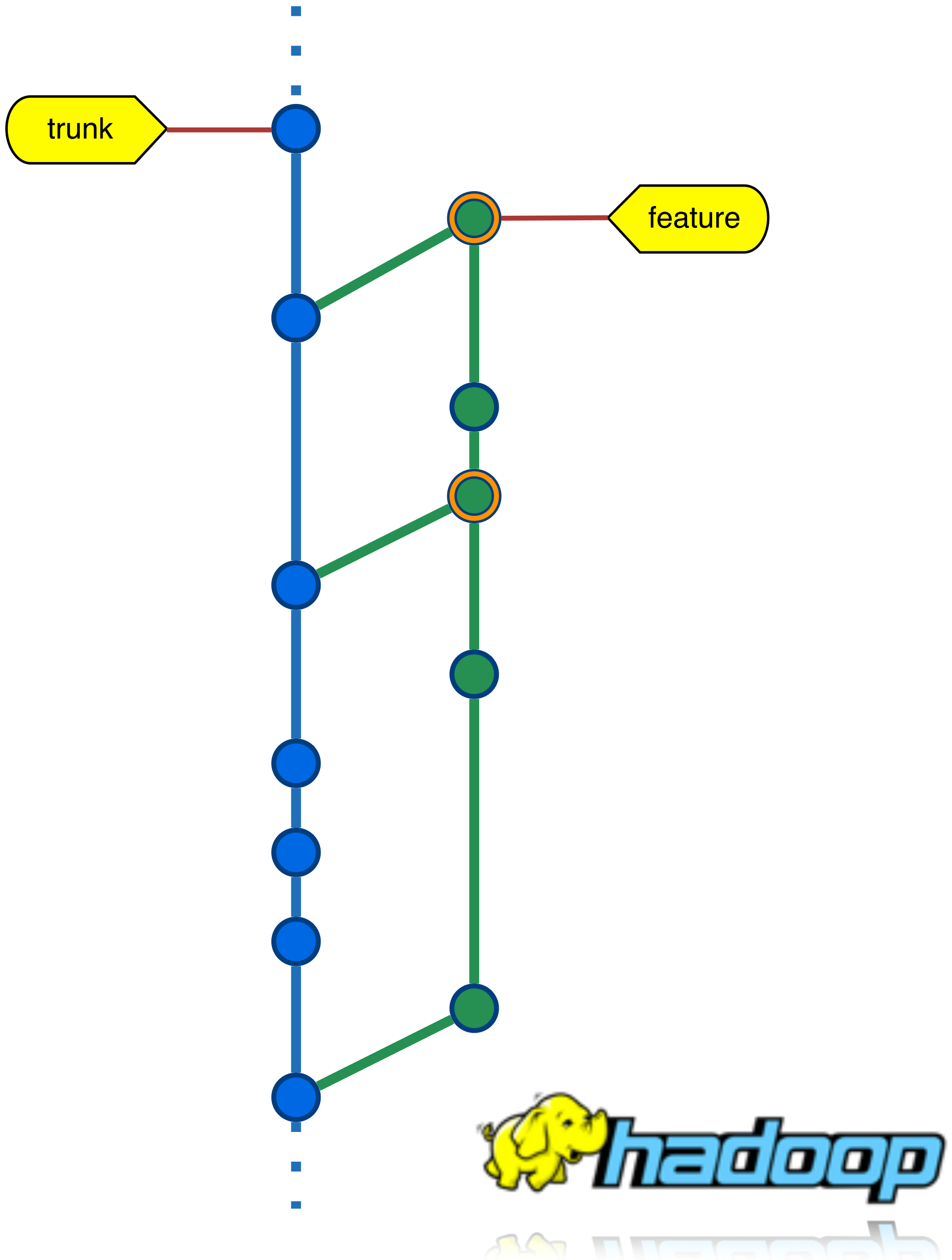
- Accuracy: do we find what we want?
- Effectiveness: reduction rate?
- Efficiency: performance w.r.t. project scale & history length?

Subjects

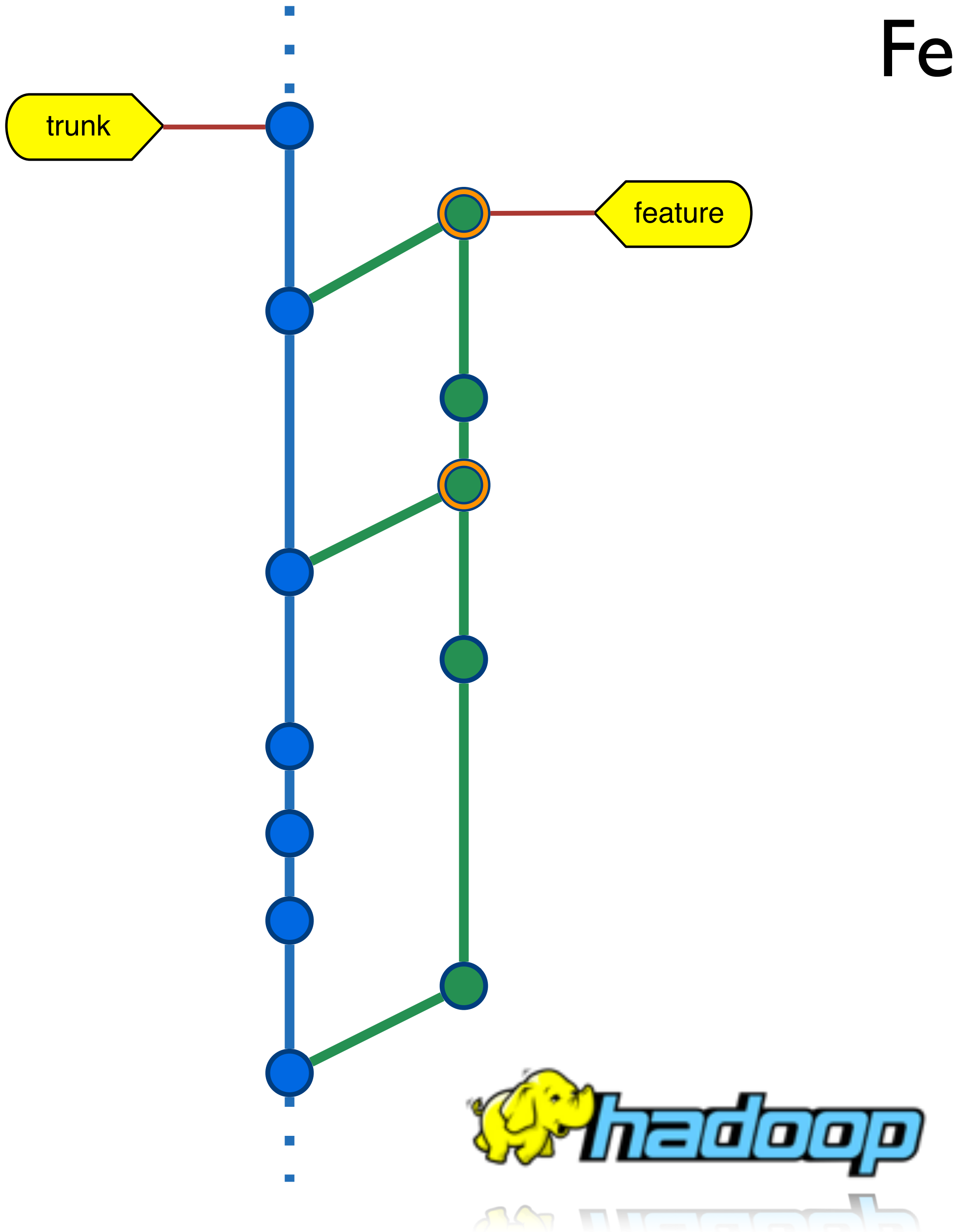
- Advanced Java features not tested: abstract class, reflection, etc.
- Non-Java changes are included by default

Project	# Java Files	LOC	# Authors
Hadoop	5,861	1,291K	169
Elasticsearc	3,865	616K	649
Maven	1,048	142K	78
CSlicer	141	18K	2

Accuracy



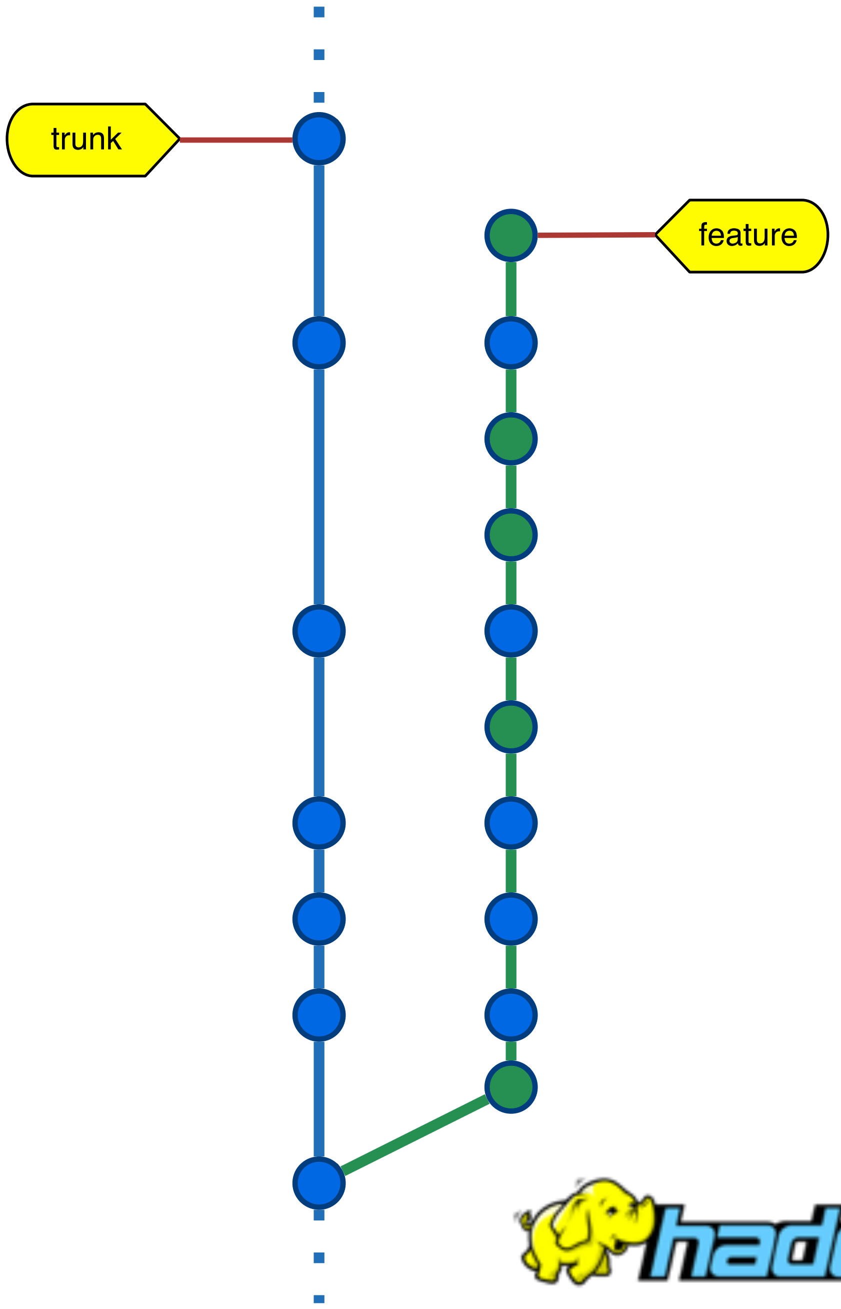
Accuracy



Feature branch

- Merges with the main branch periodically
- 42 feature commits + 47 merges
- 58 accompanied test cases

Accuracy



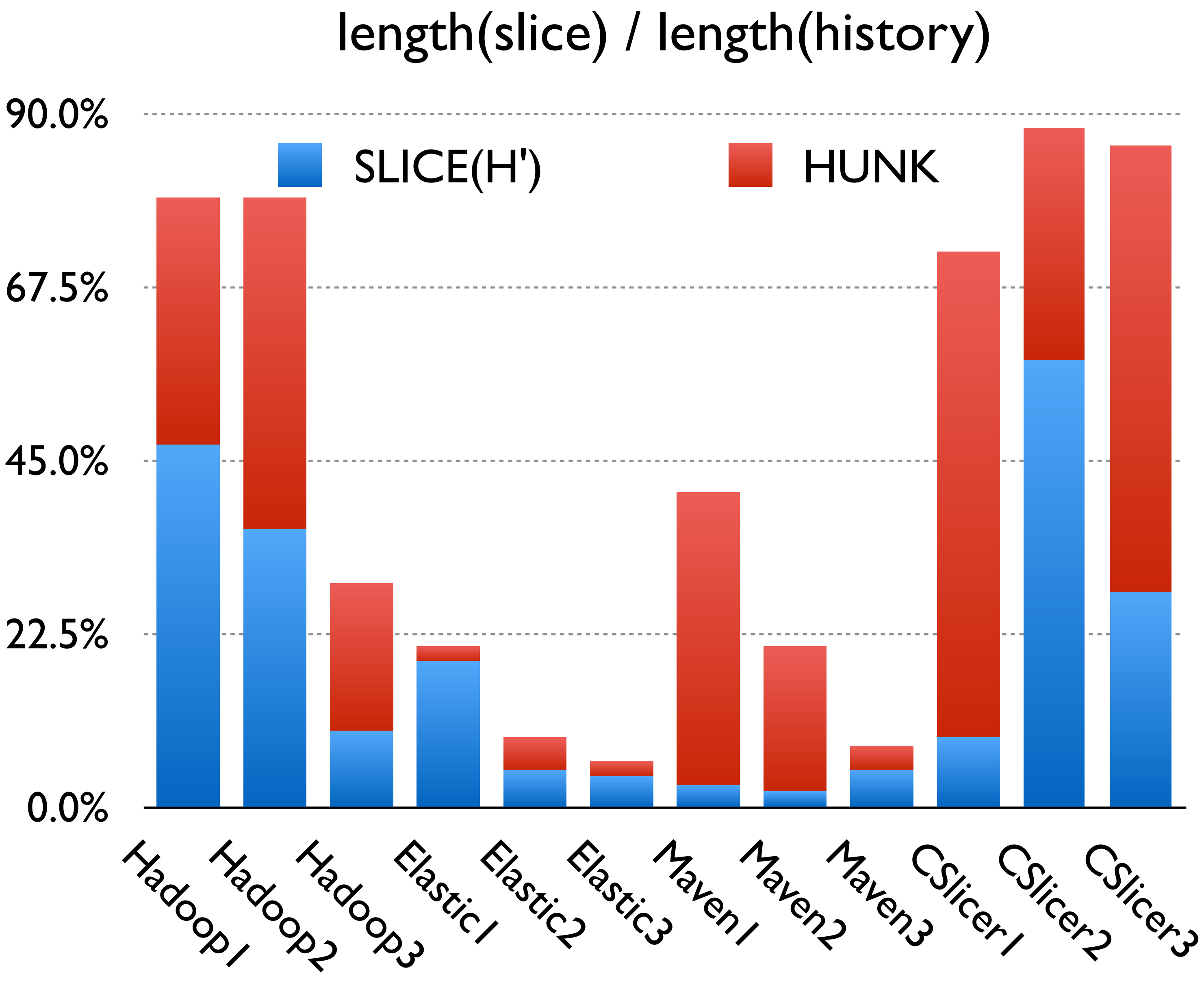
Feature branch

- Merges with the main branch periodically
- 42 feature commits + 47 merges
- 58 accompanied test cases

Case Study:

- Separate feature changes
- Identified 65 out of 267 commits related to the feature
- 41 matches original

Effectiveness



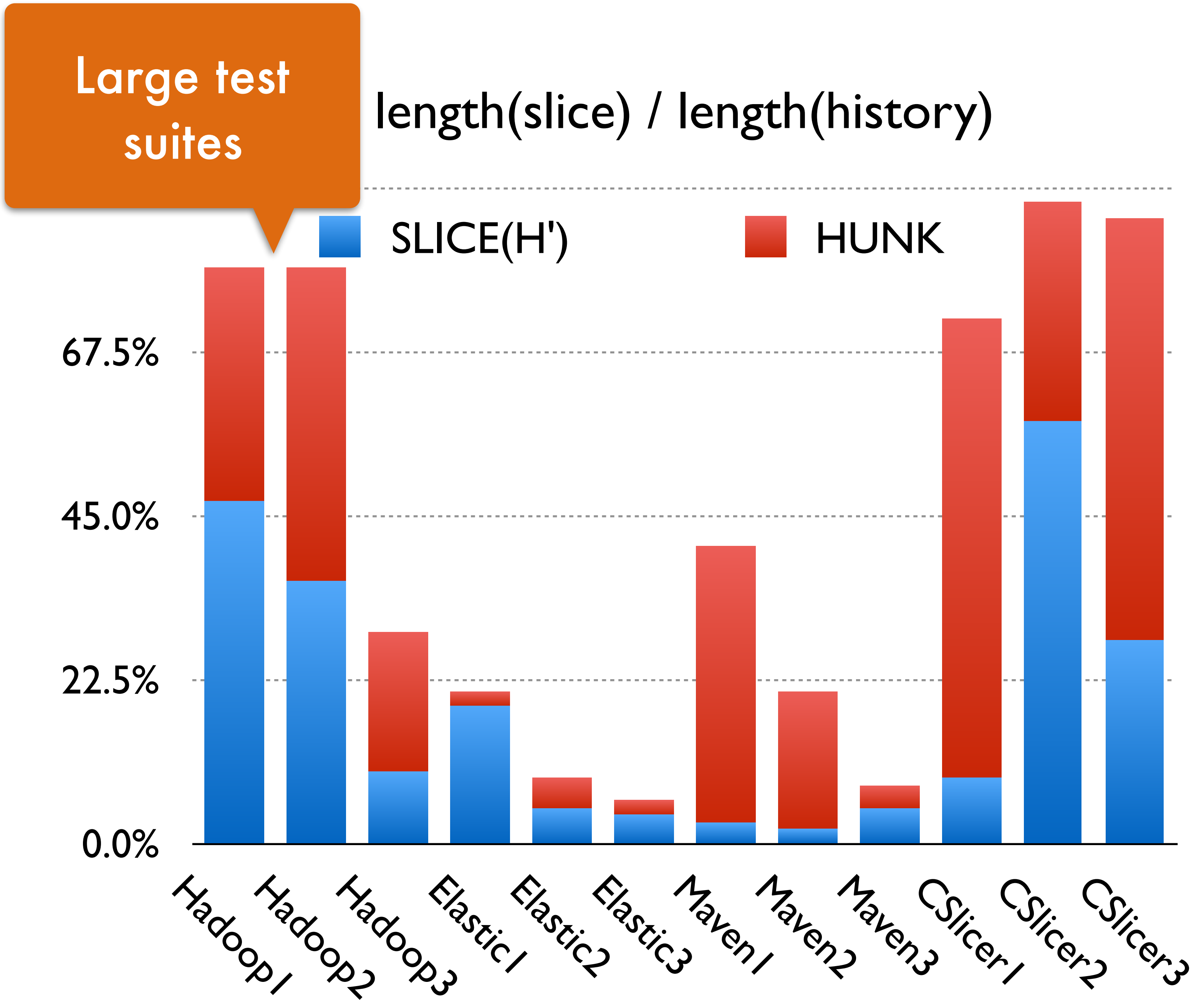
Average Reduction:

~80%!

Reduction depends on:

1. tests complexity
2. committing styles

Effectiveness



Large test suites

length(slice) / length(history)

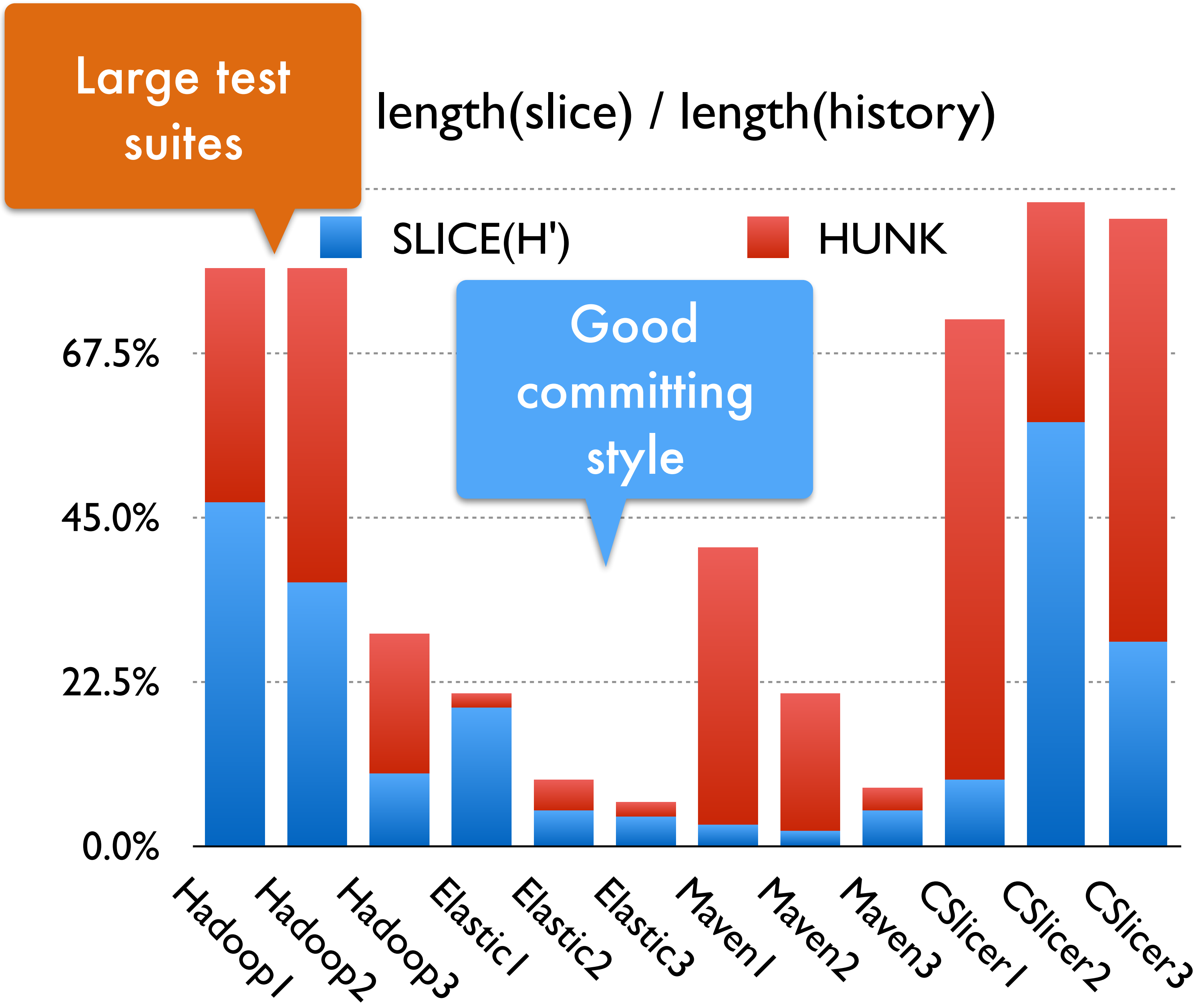
Average Reduction:

~80%!

Reduction depends on:

1. tests complexity
2. committing styles

Effectiveness



Large test suites

Good committing style

Average Reduction:

~80%!

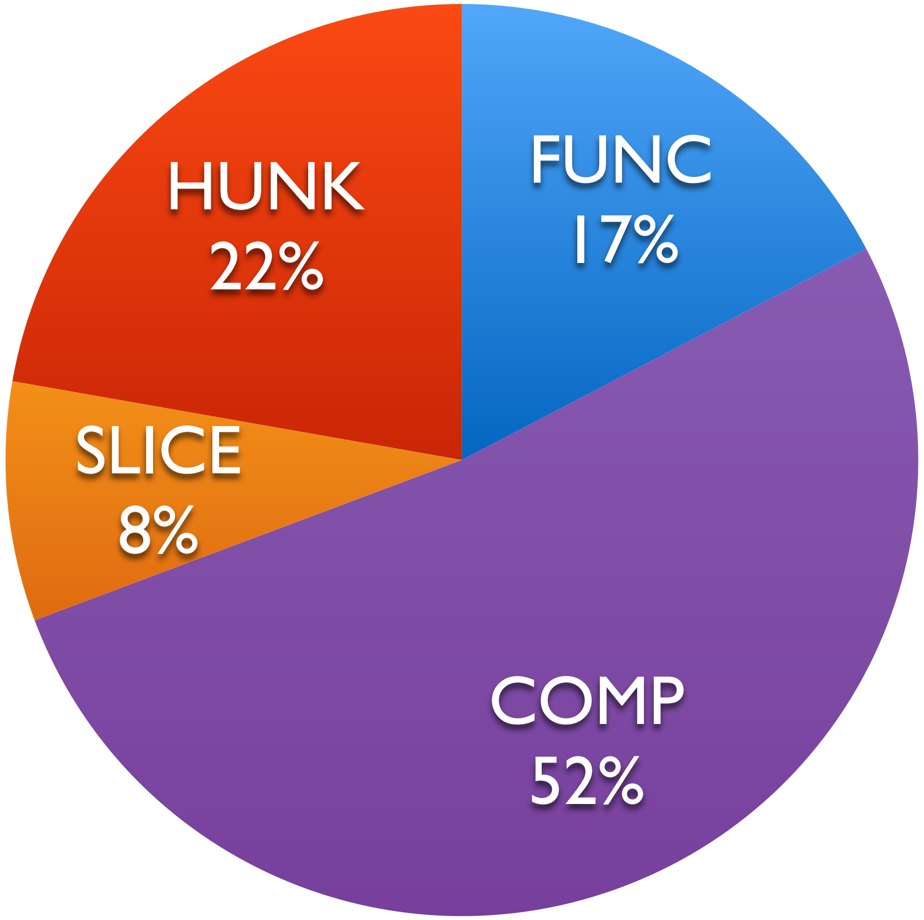
Reduction depends on:

1. tests complexity
2. committing styles

Performance

- Total CSlicer time: 2 ~ 65 s
- Major part spent in functional & compilation set computation
- History length has little effects on performance for large projects

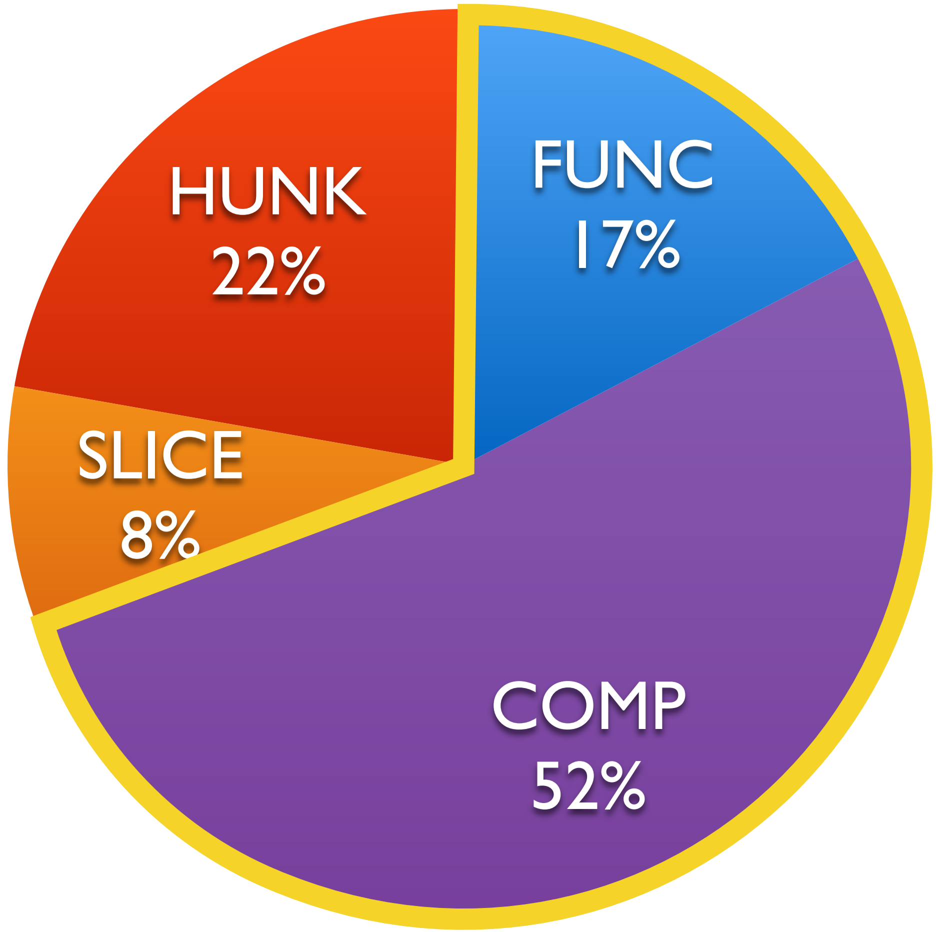
CSlicer time breakdown



Performance

- Total CSlicer time: 2 ~ 65 s
- Major part spent in functional & compilation set computation
- History length has little effects on performance for <u>large</u> projects

CSlicer time breakdown



Outline

1. Introduction
2. Dependency Hierarchy
3. CSlicer Algorithm
4. Evaluation
5. Related Work & Conclusion

Related Work

Change Representation

- Code change classification [Falleri et al., ASE'14; Chawathe, SIGMOD'96]
- History granularity transformation [Muslu et al., ASE'15]

Change Impact Analysis

- Compute affected regression tests [Ren et al., OPPSLA'04]
- Fault localization [Zhang et al., ICSM'01]

Conclusion & Future Work

CSlicer: history semantic slicing

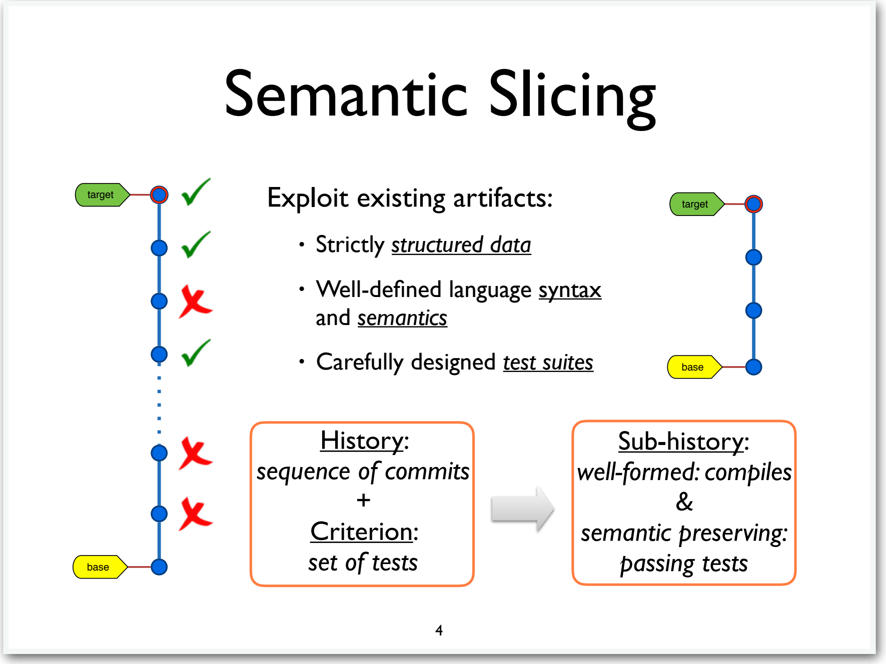
- Filling the gap between texts and semantics
- Adapted to existing version control tools
- Many interesting applications: history comprehension; functionality transferring …

What's next?

bitbucket.org/liyistc/gitslice

- Handle distributed histories
- Slice integration — the "paste" step

Questions?



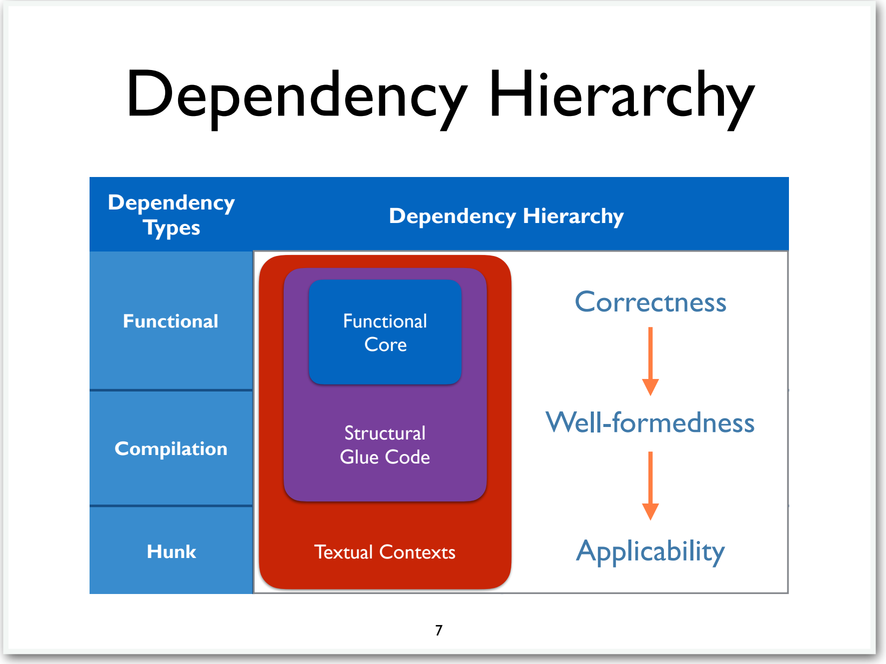
Semantic Slicing

Exploit existing artifacts:

- Strictly *structured data*
- Well-defined language *syntax* and *semantics*
- Carefully designed *test suites*

History:
sequence of commits
+
Criterion:
set of tests

Sub-history:
well-formed: compiles
&
semantic preserving: passing tests



Dependency Hierarchy

Dependency Types | Dependency Hierarchy

Functional

Compilation

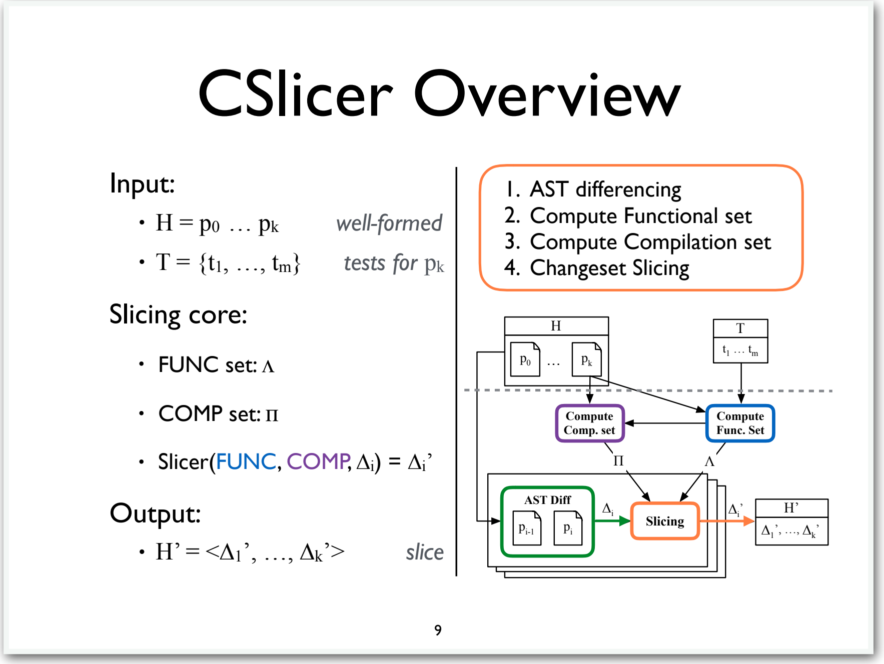
Hunk

Functional Core

Structural Glue Code

Textual Contexts

Correctness → Well-formedness → Applicability



CSlicer Overview

Input:

- $H = p_0 \dots p_k$ *well-formed*
- $T = \{t_1, \dots, t_m\}$ *tests for* p_k

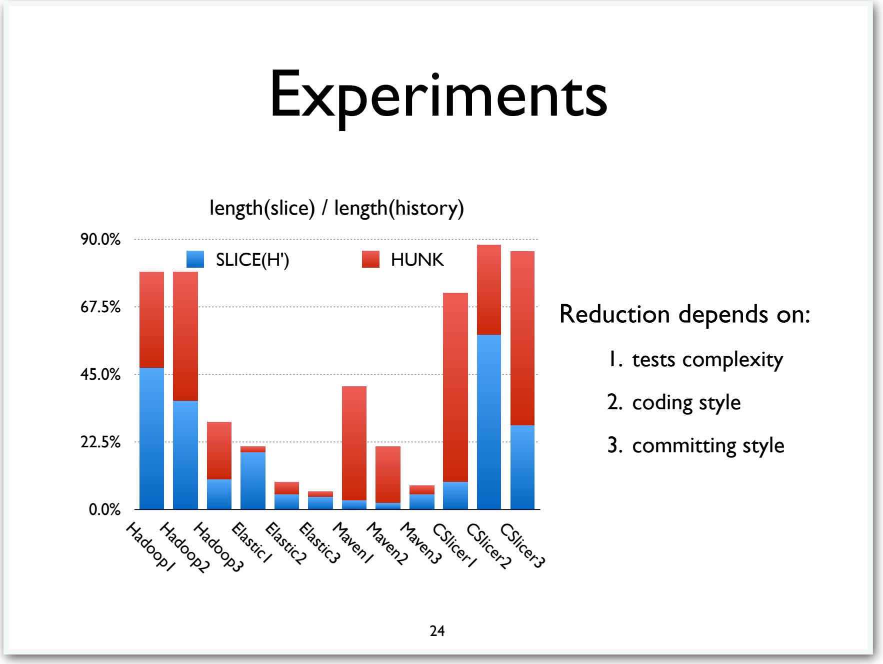
Slicing core:

- FUNC set: Λ
- COMP set: Π
- Slicer(FUNC, COMP, Δ_i) = Δ_i’

Output:

- $H’ = <\Delta_1’, \dots, \Delta_k’>$ *slice*

1. AST differencing
2. Compute Functional set
3. Compute Compilation set
4. Changeset Slicing



Experiments

Reduction depends on:

1. tests complexity
2. coding style
3. committing style