Type-Intertwined Heap Analysis


Devin Coughlin


Ross Holland


Bor-Yuh Evan Chang

University of Colorado Boulder

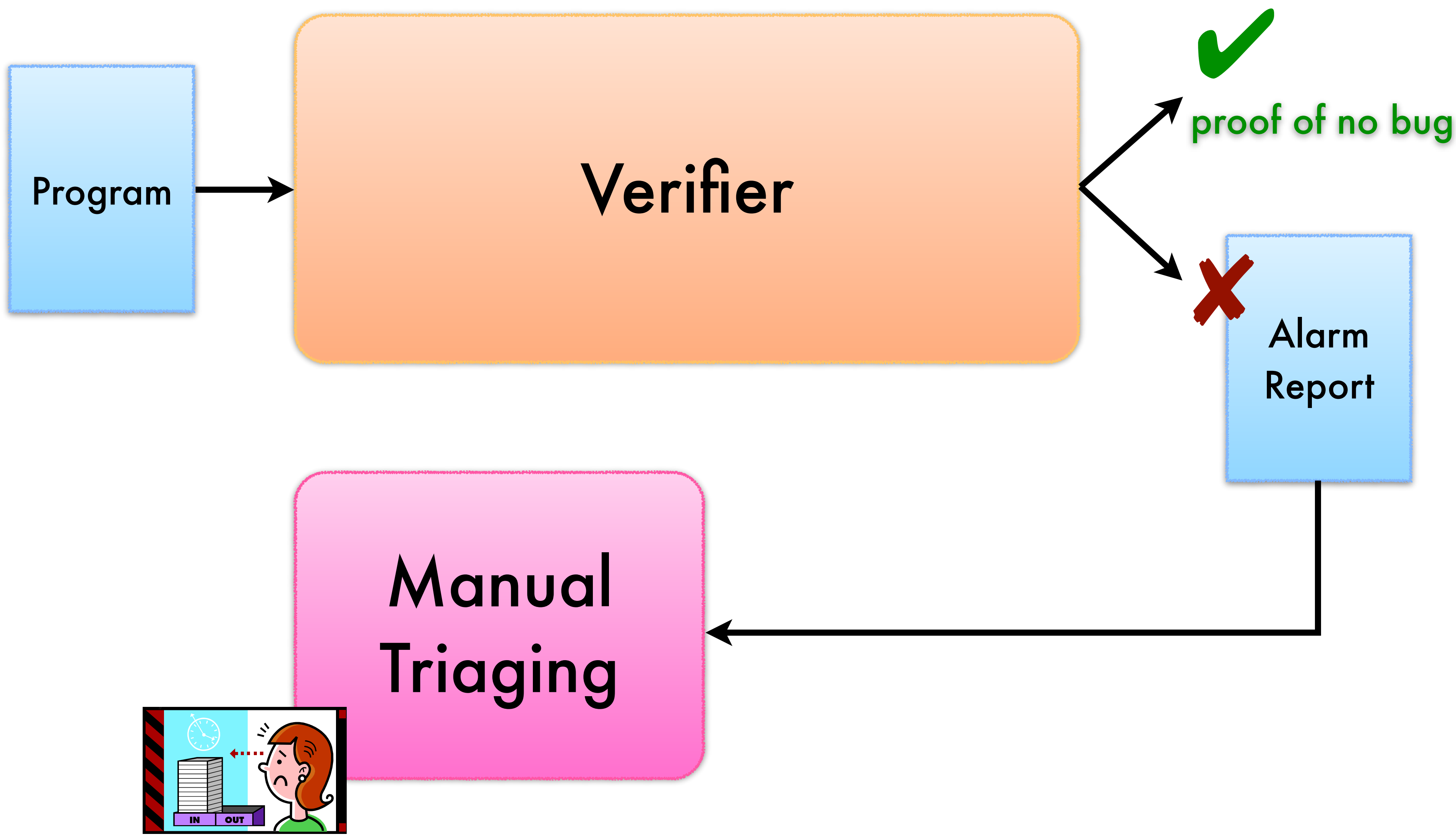
Aarhus University
May 18, 2015

PLV

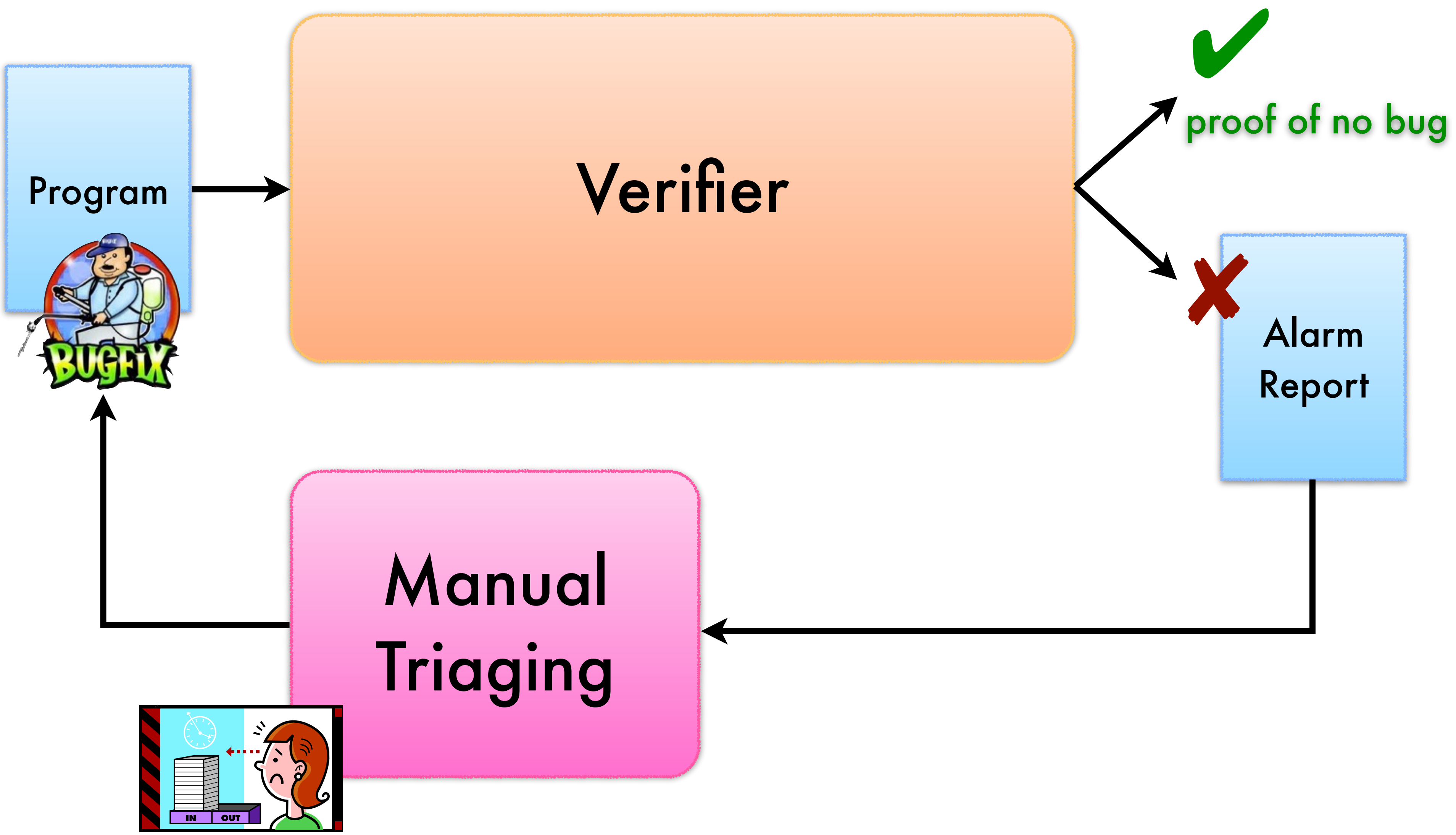
Lab: **Program analysis** in the **whole** bug mitigation **process**

Lab: **Program analysis** in the **whole** bug mitigation **process**

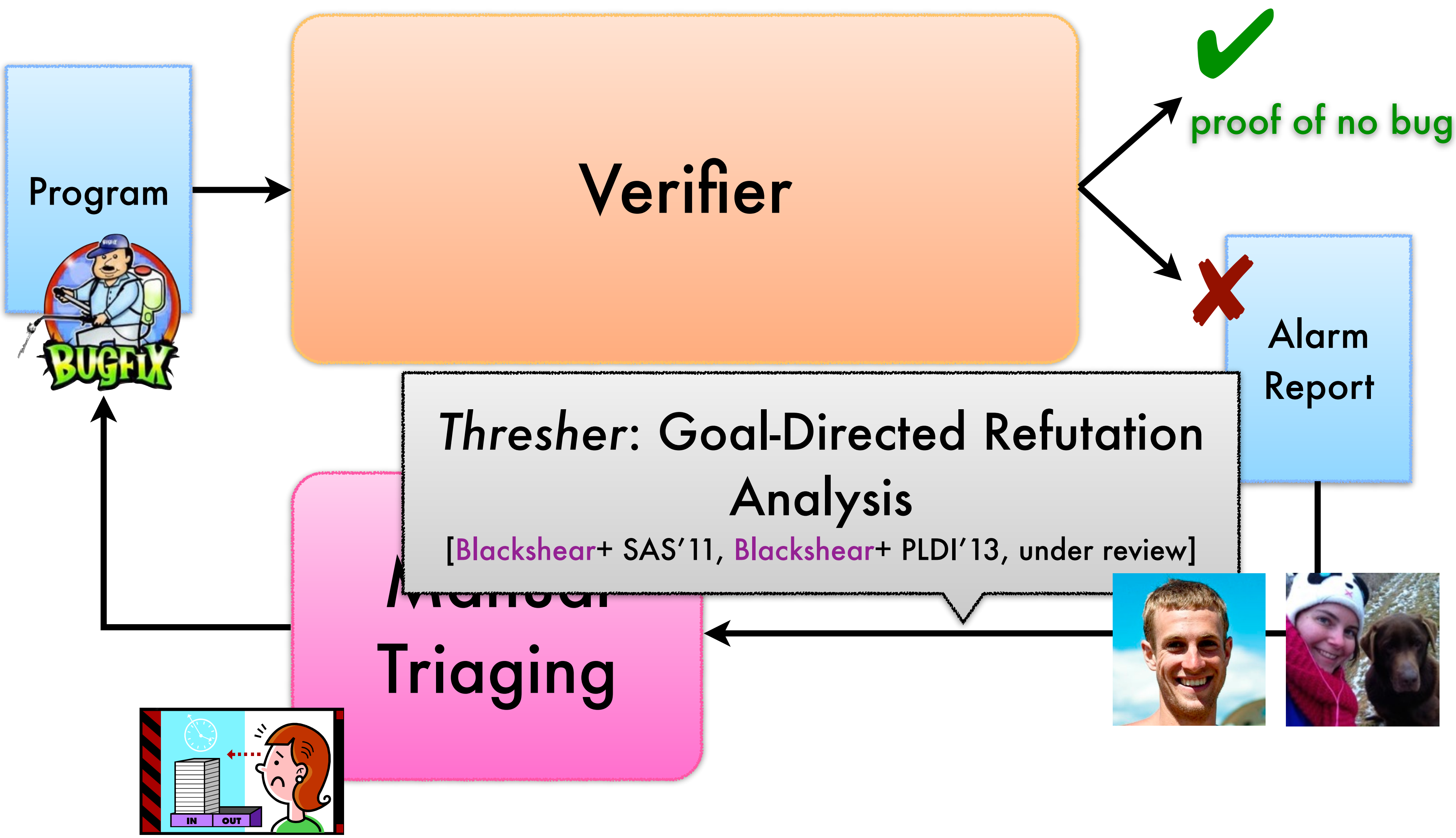
Lab: **Program analysis** in the **whole** bug mitigation **process**



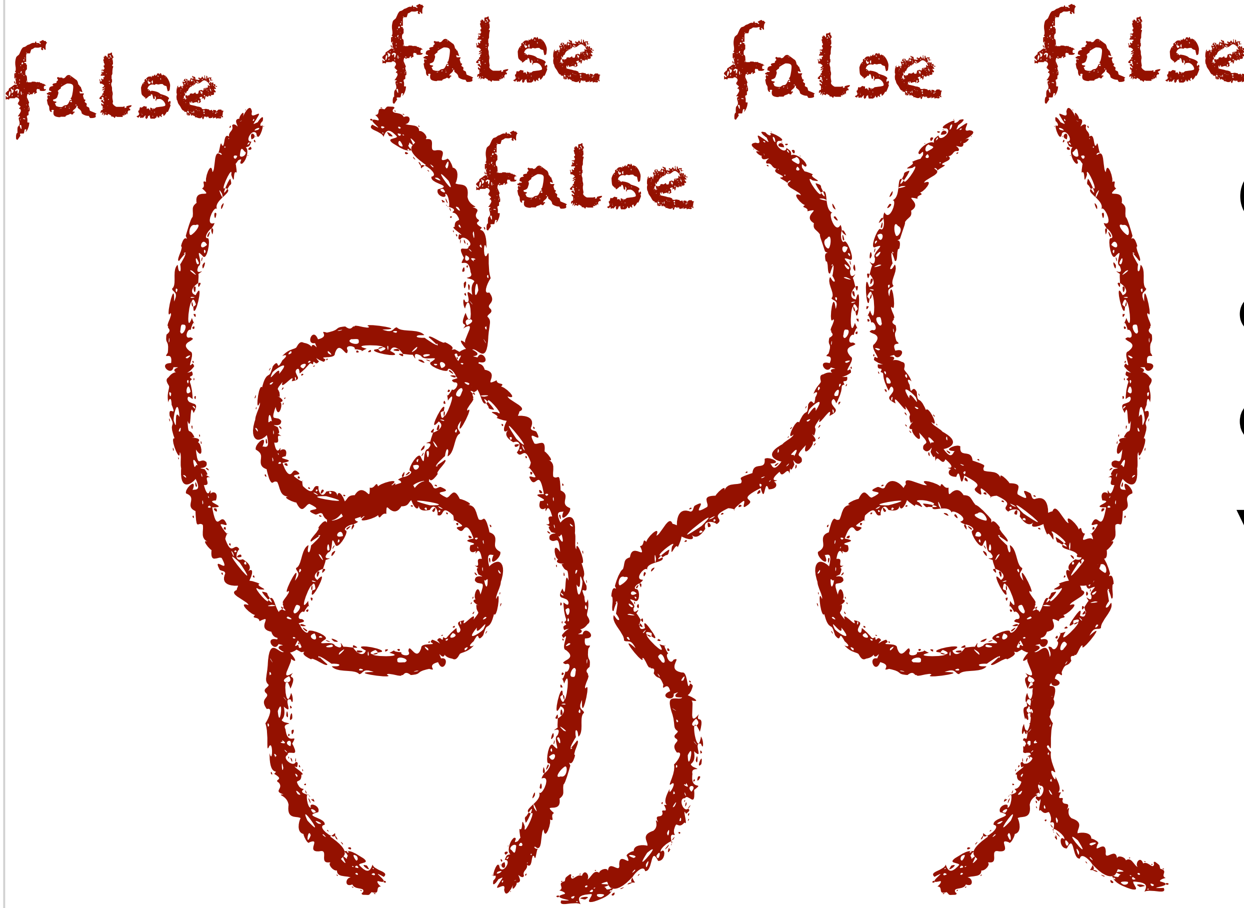
Lab: **Program analysis** in the **whole** bug mitigation **process**



Lab: **Program analysis** in the **whole** bug mitigation **process**



Lab:



Given a program configuration **goal**, derive a **contradiction** w.r.t. its reachability

$$(x \mapsto \hat{x} * \hat{x} \cdot f \mapsto \hat{a} * \text{true}) \wedge \hat{a} \neq \text{null}$$

backwards abstract interpretation of separation logic

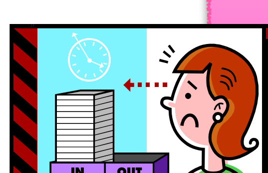
of no bug

Alarm Report

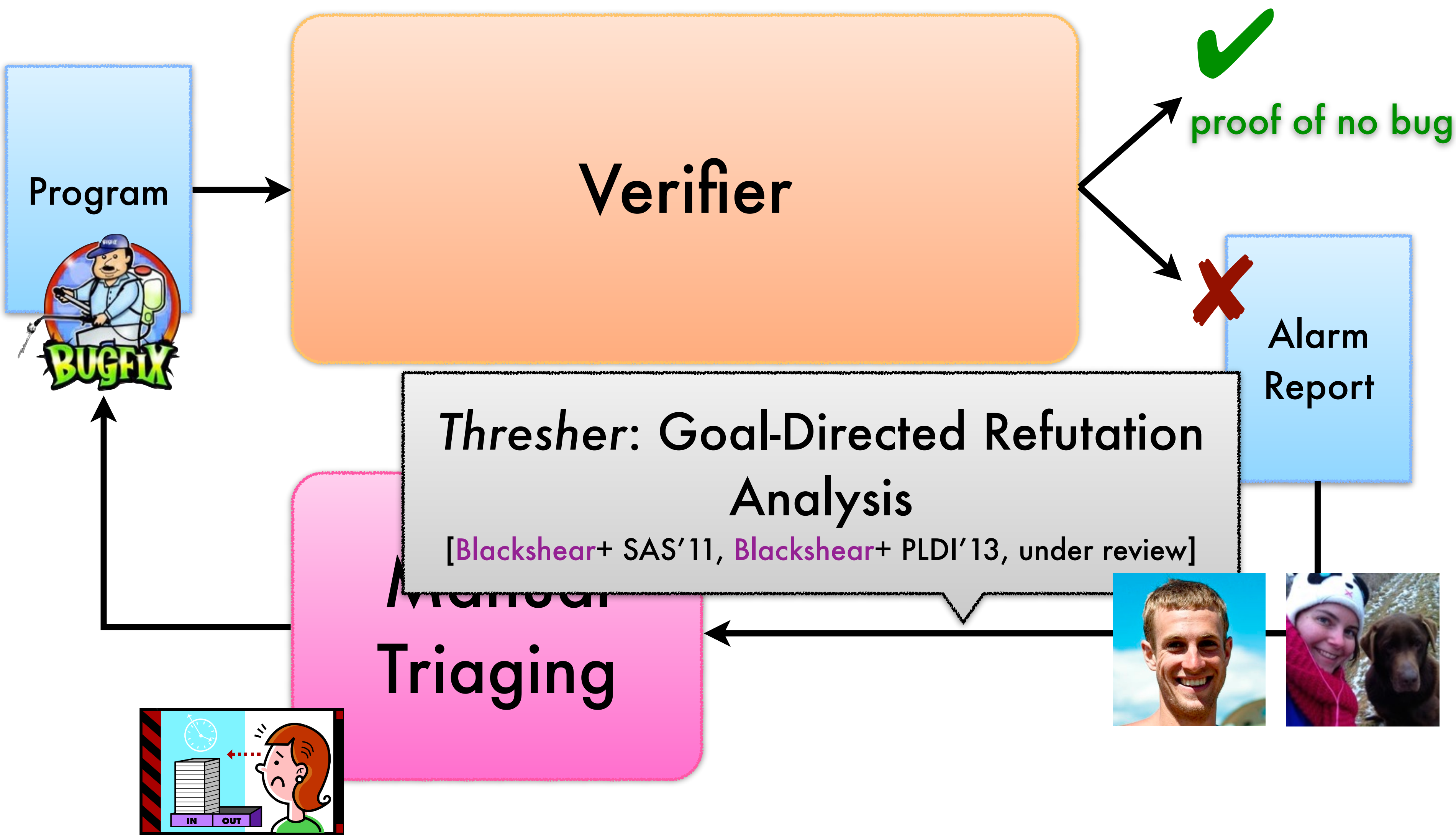
Thresher: Goal-Directed Refutation Analysis

[Blackshear+ SAS'11, Blackshear+ PLDI'13, under review]

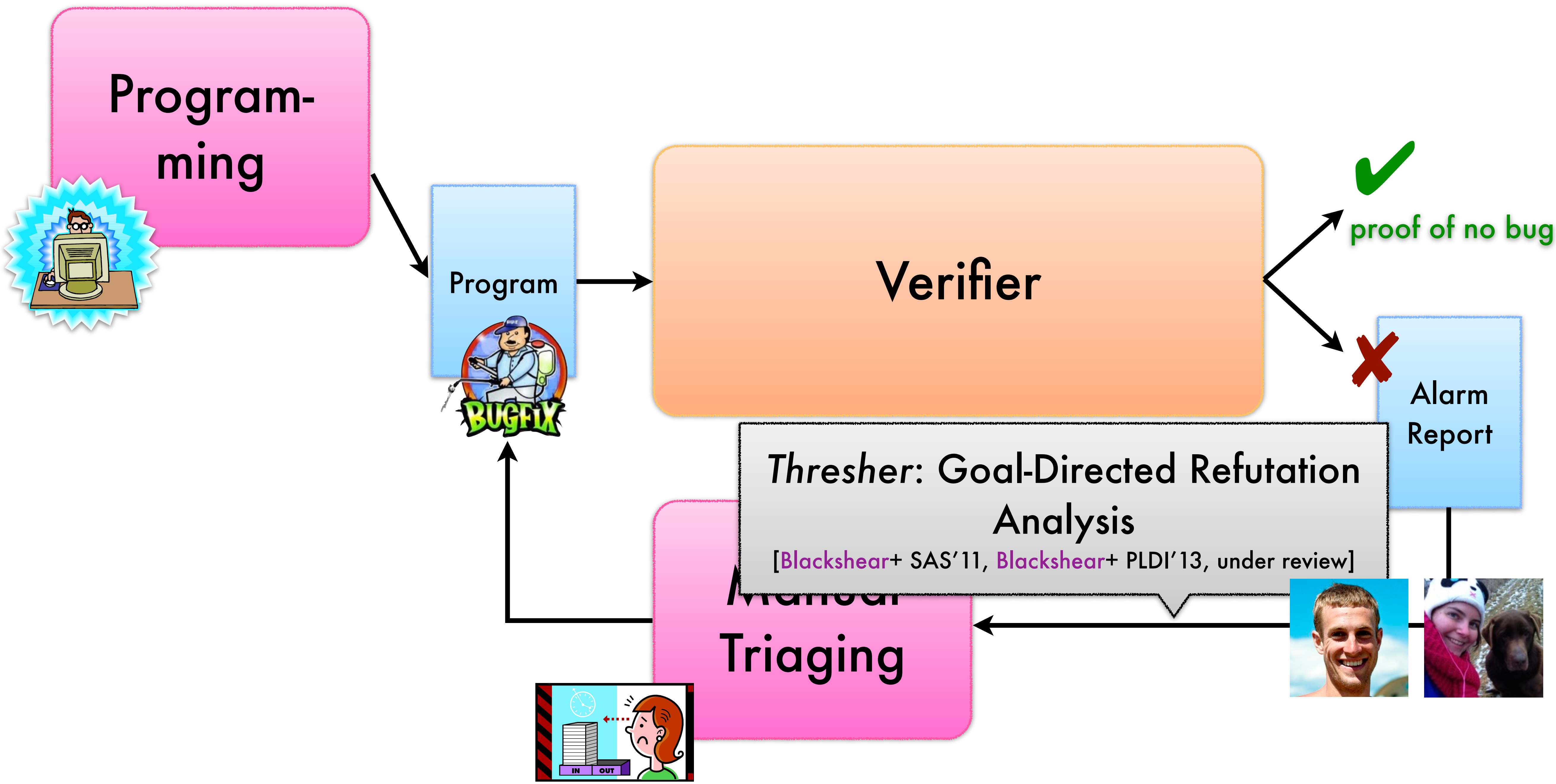
Triaging



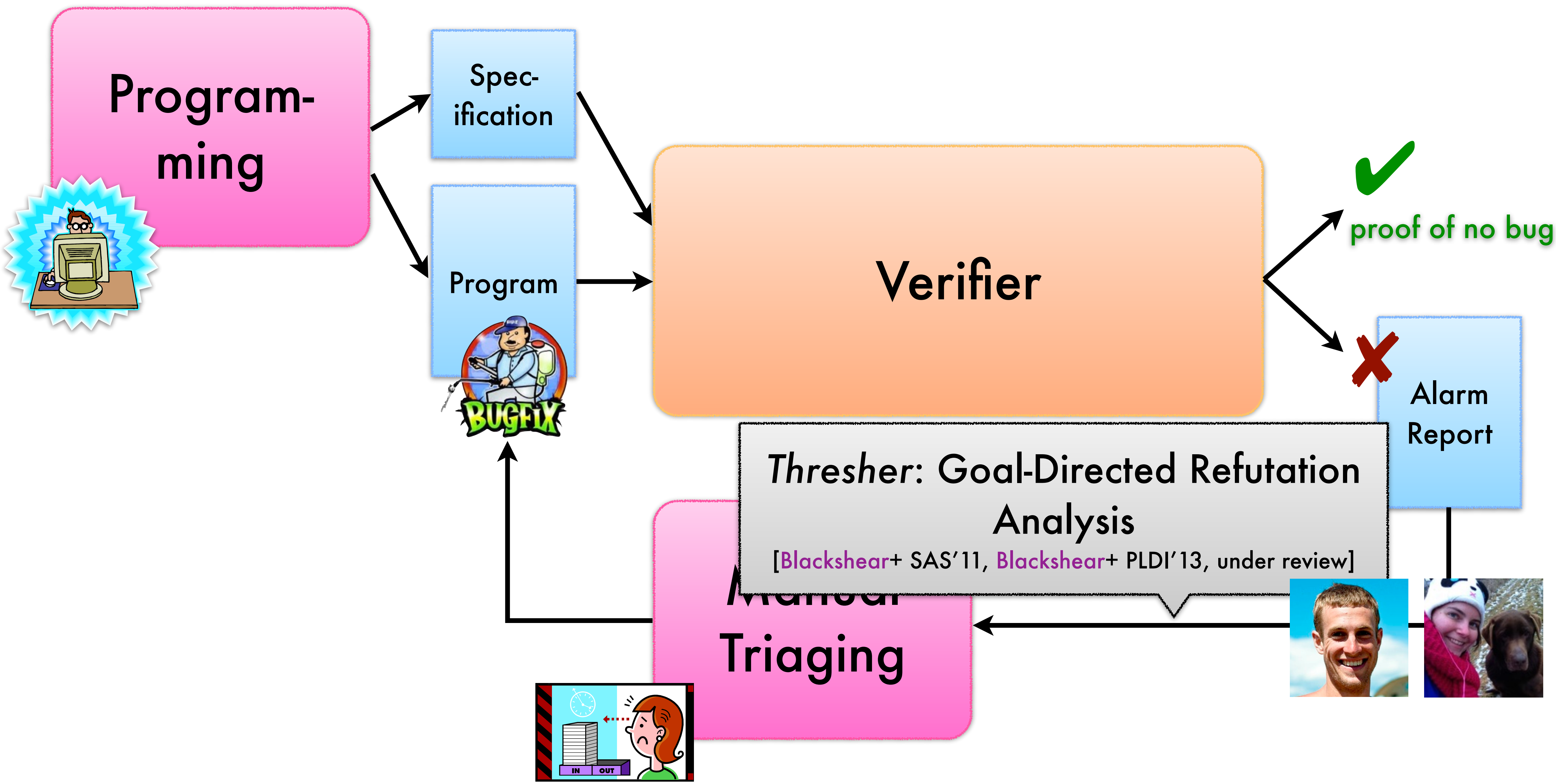
Lab: **Program analysis** in the **whole** bug mitigation **process**



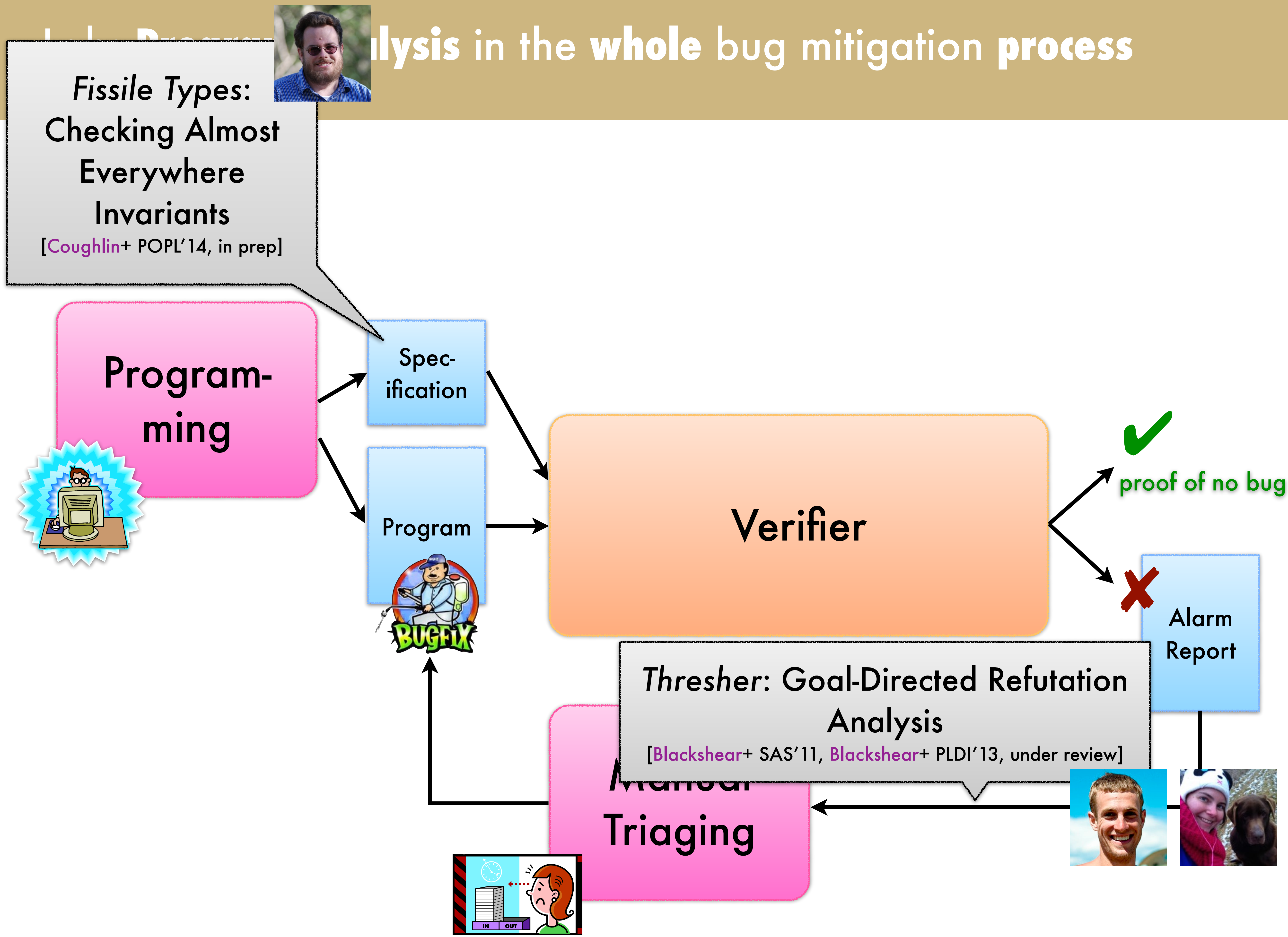
Lab: **Program analysis** in the **whole** bug mitigation **process**



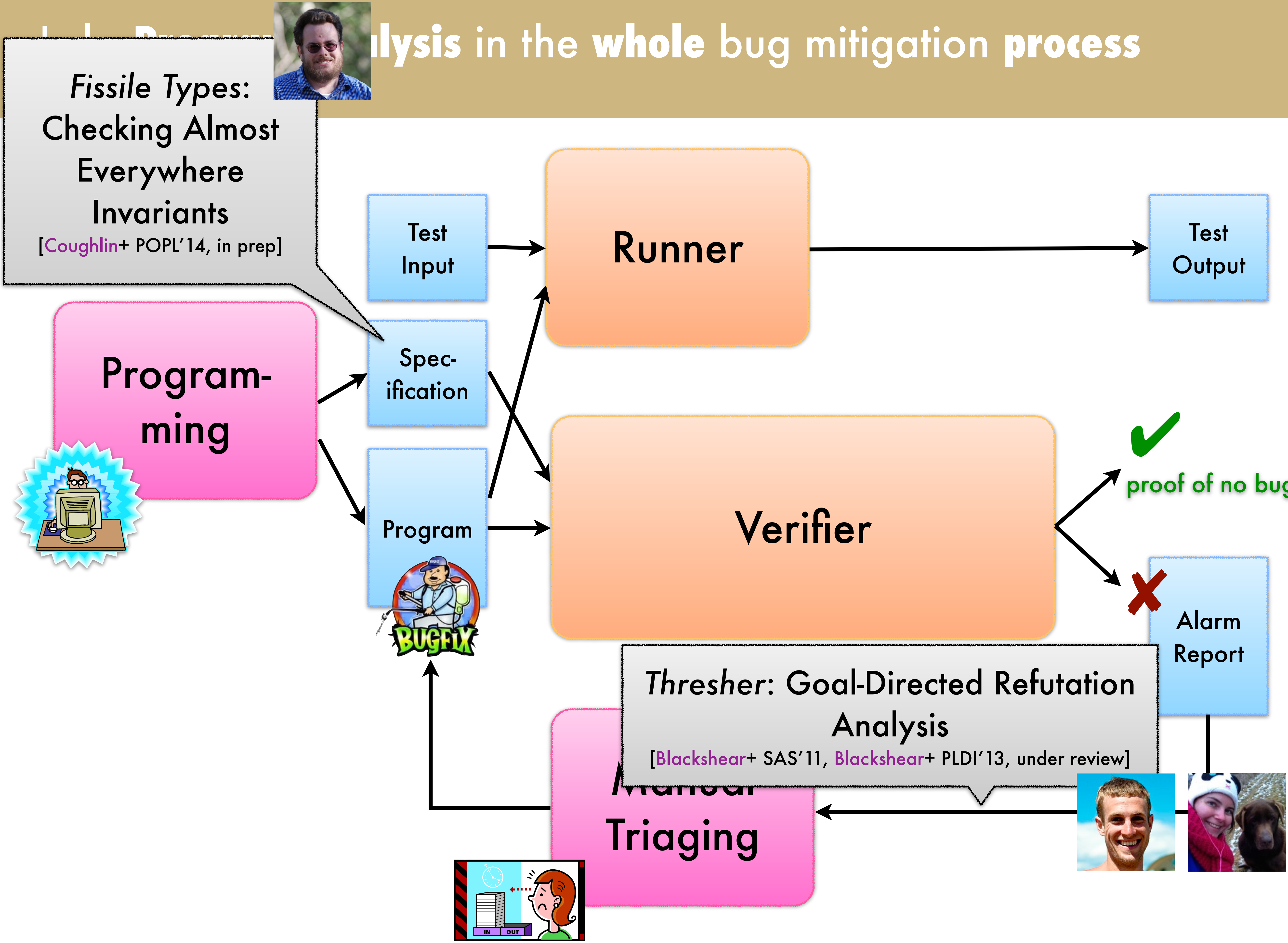
Lab: **Program analysis** in the **whole** bug mitigation **process**

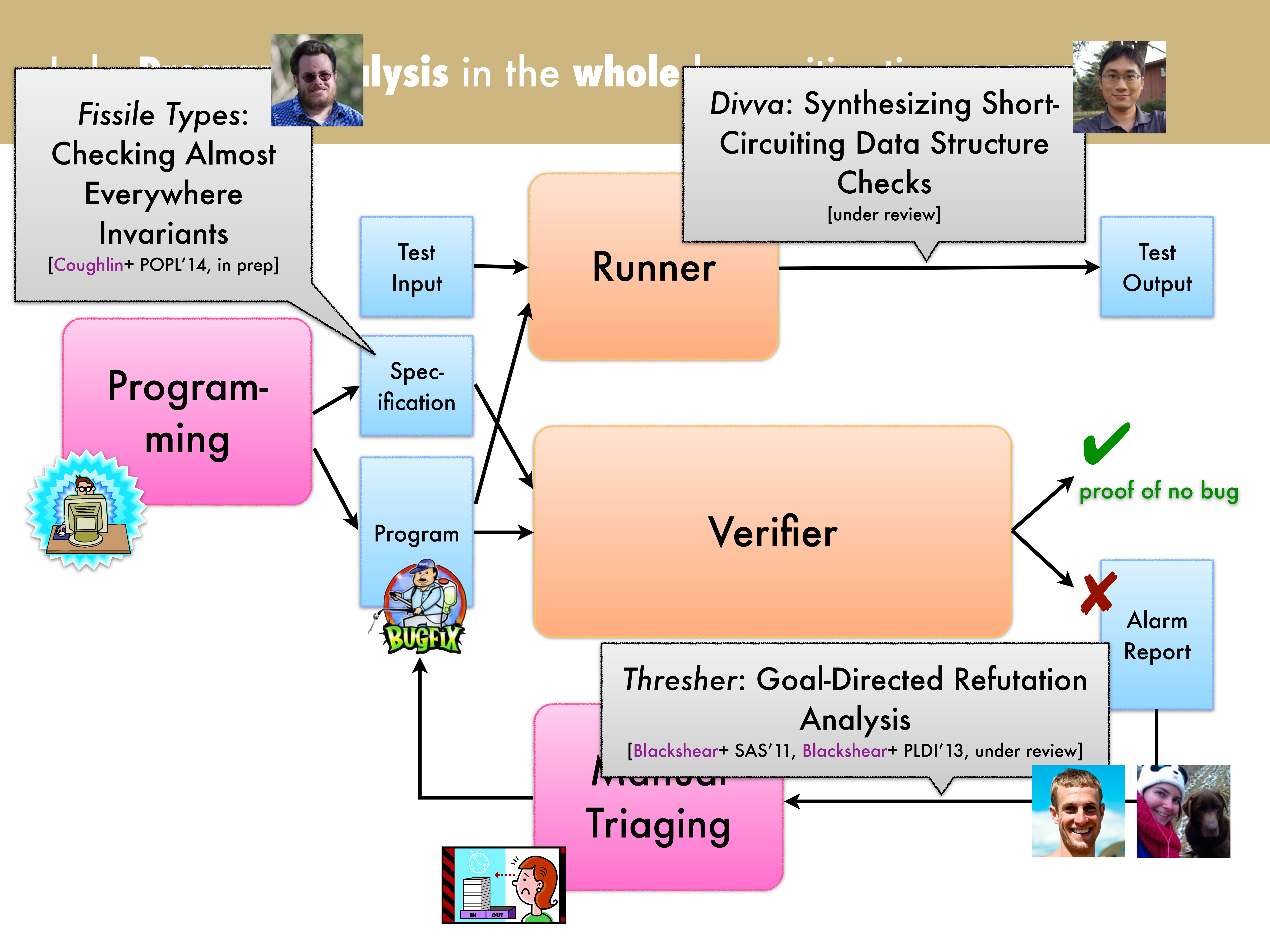


…lysis in the **whole** bug mitigation **process**



…lysis in the **whole** bug mitigation **process**




lysis in the whole
Fissile Types: Checking Almost Everywhere Invariants
[Coughlin+ POPL'14, in prep]
Divva: Synthesizing Short-Circuiting Data Structure Checks
[under review]
Test Input
Runner
Test Output
Program-ming
Spec-ification
Program
BUGFIX
Verifier
proof of no bug
Alarm Report
Thresher: Goal-Directed Refutation Analysis
[Blackshear+ SAS'11, Blackshear+ PLDI'13, under review]
Triaging
IN
OUT

...lysis in the whole...

Fissile Types: Checking Almost Everywhere Invariants

[Coughlin+ POPL'14, in prep]

Divva: Synthesizing Short-Circuiting Data Structure Checks

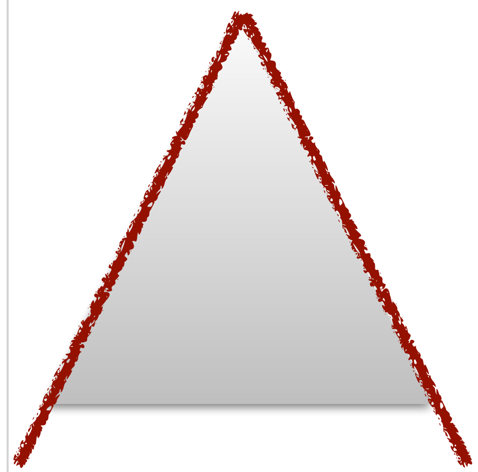
[under review]

Test Input

Spec-ification

Program

Program-ming



Use **static shape analysis** to **synthesize short-circuiting dynamic validation** of data structure invariants

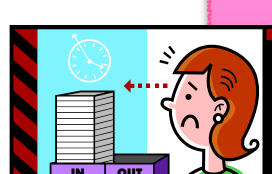
uninterpreted inductive separation logic predicates

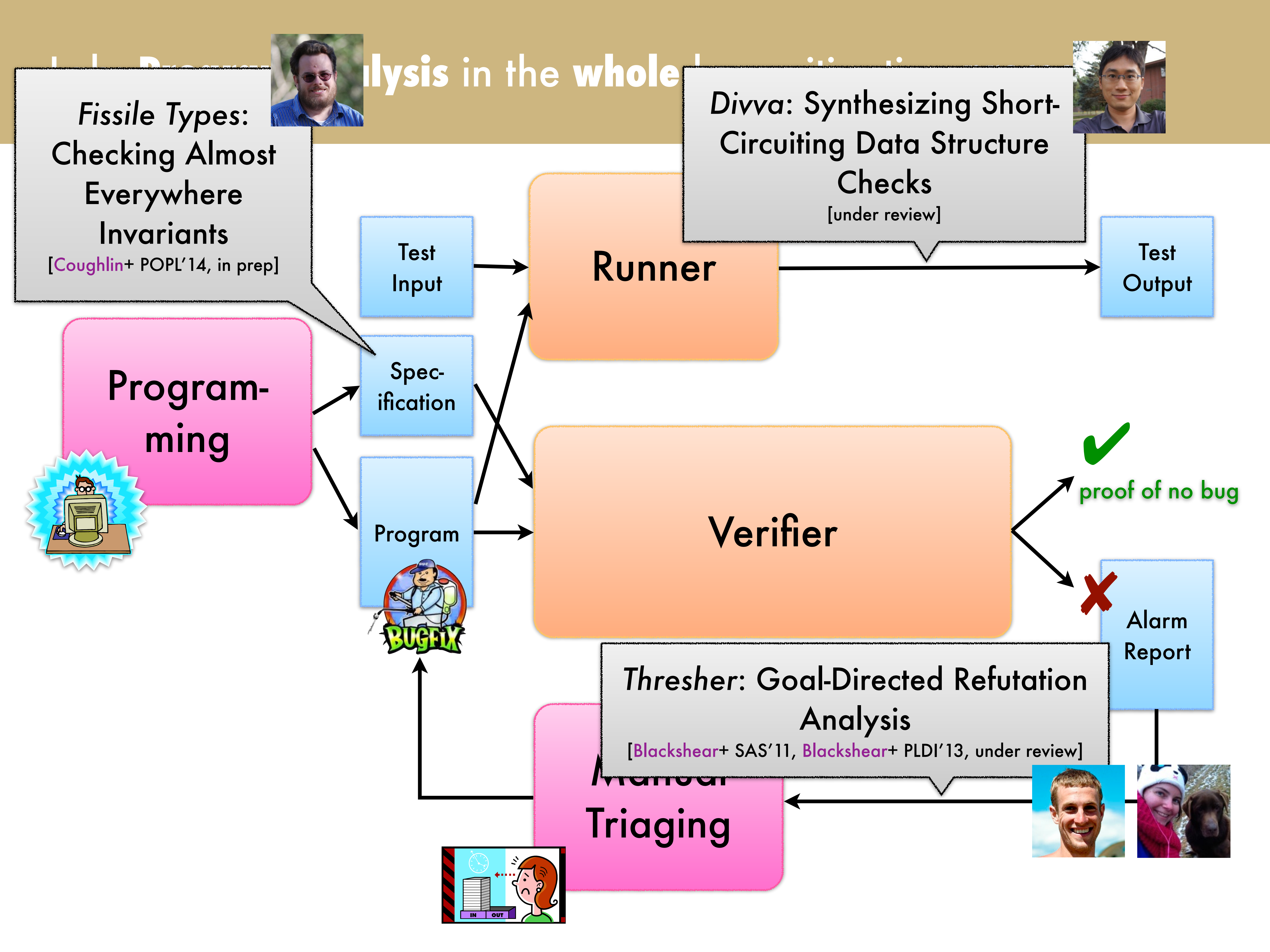
Alarm Report

Thresher: Goal-Directed Refutation Analysis

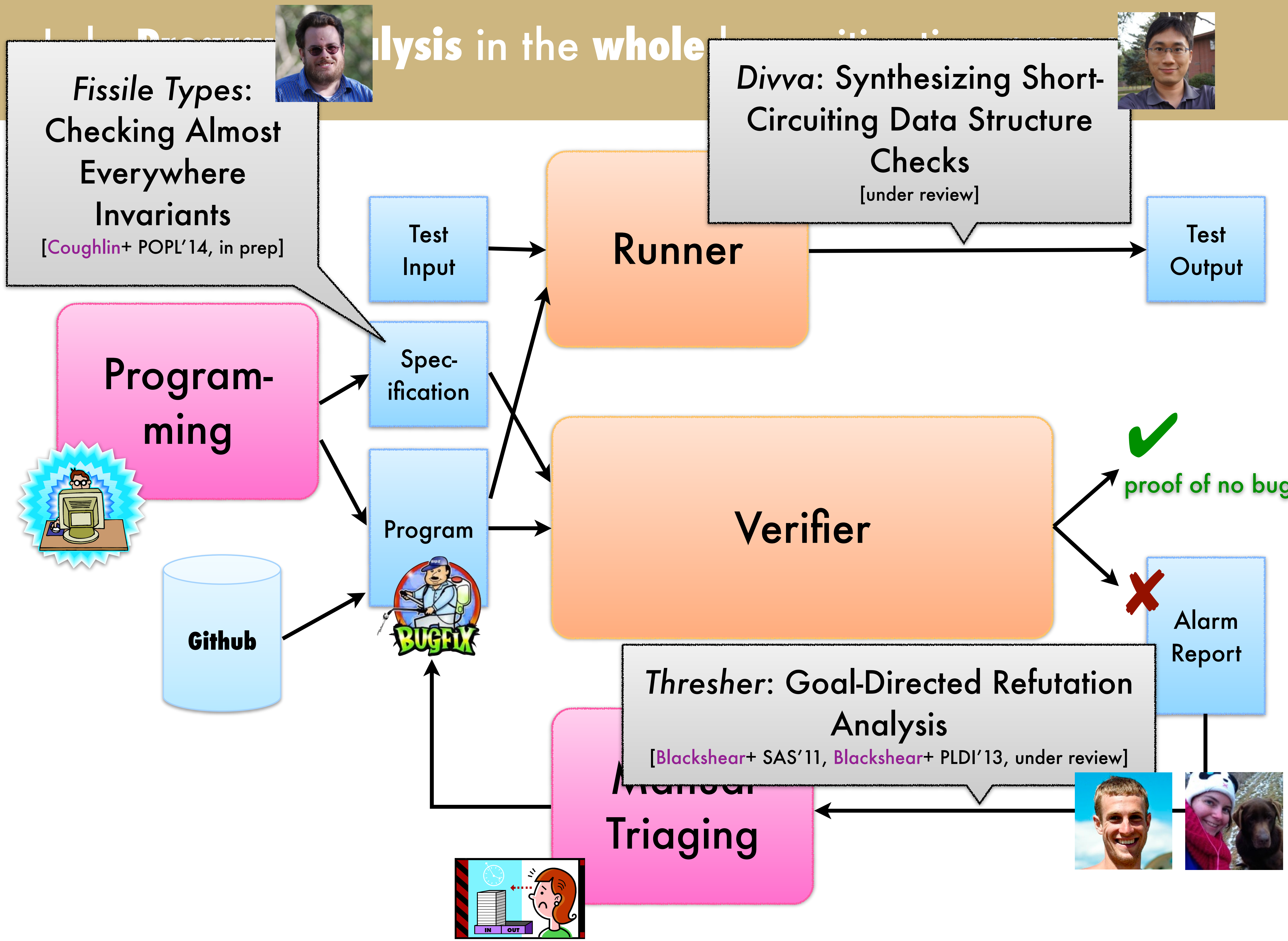
[Blackshear+ SAS'11, Blackshear+ PLDI'13, under review]

Triaging

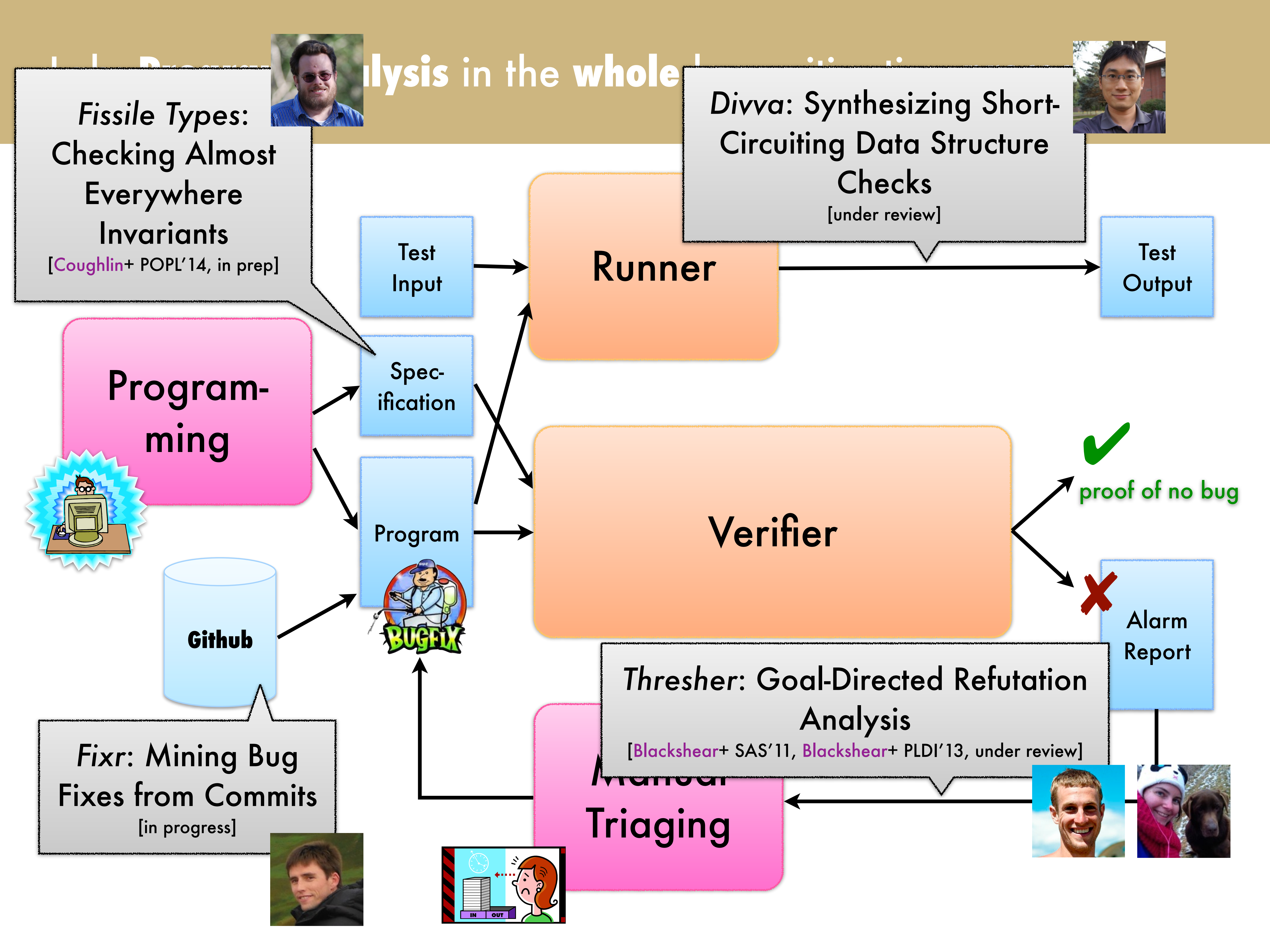




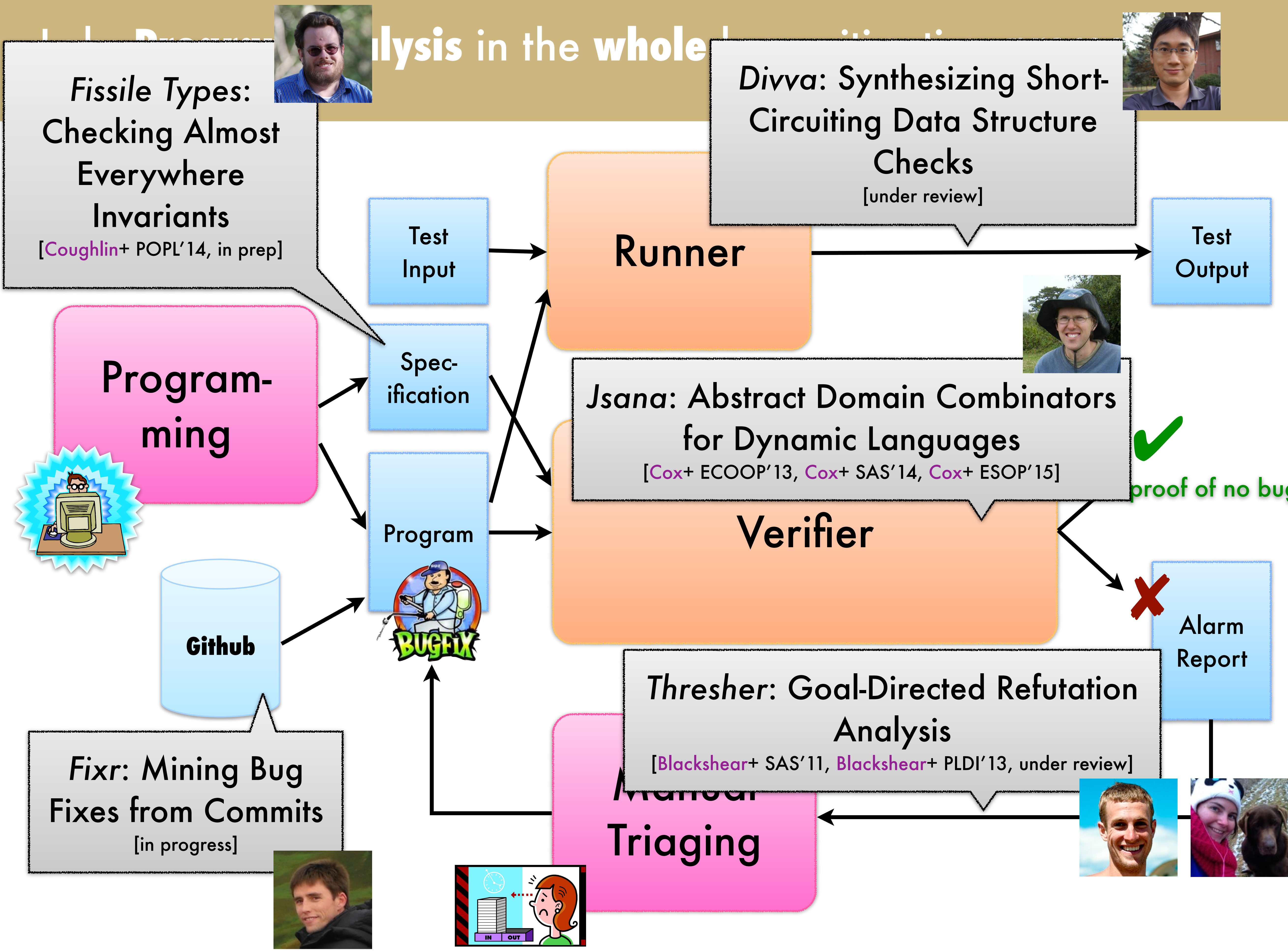
lysis in the whole
Fissile Types: Checking Almost Everywhere Invariants
[Coughlin+ POPL'14, in prep]
Divva: Synthesizing Short-Circuiting Data Structure Checks
[under review]
Test Input
Runner
Test Output
Program-ming
Spec-ification
Program
BUGFIX
Verifier
proof of no bug
Alarm Report
Thresher: Goal-Directed Refutation Analysis
[Blackshear+ SAS'11, Blackshear+ PLDI'13, under review]
Triaging
IN
OUT



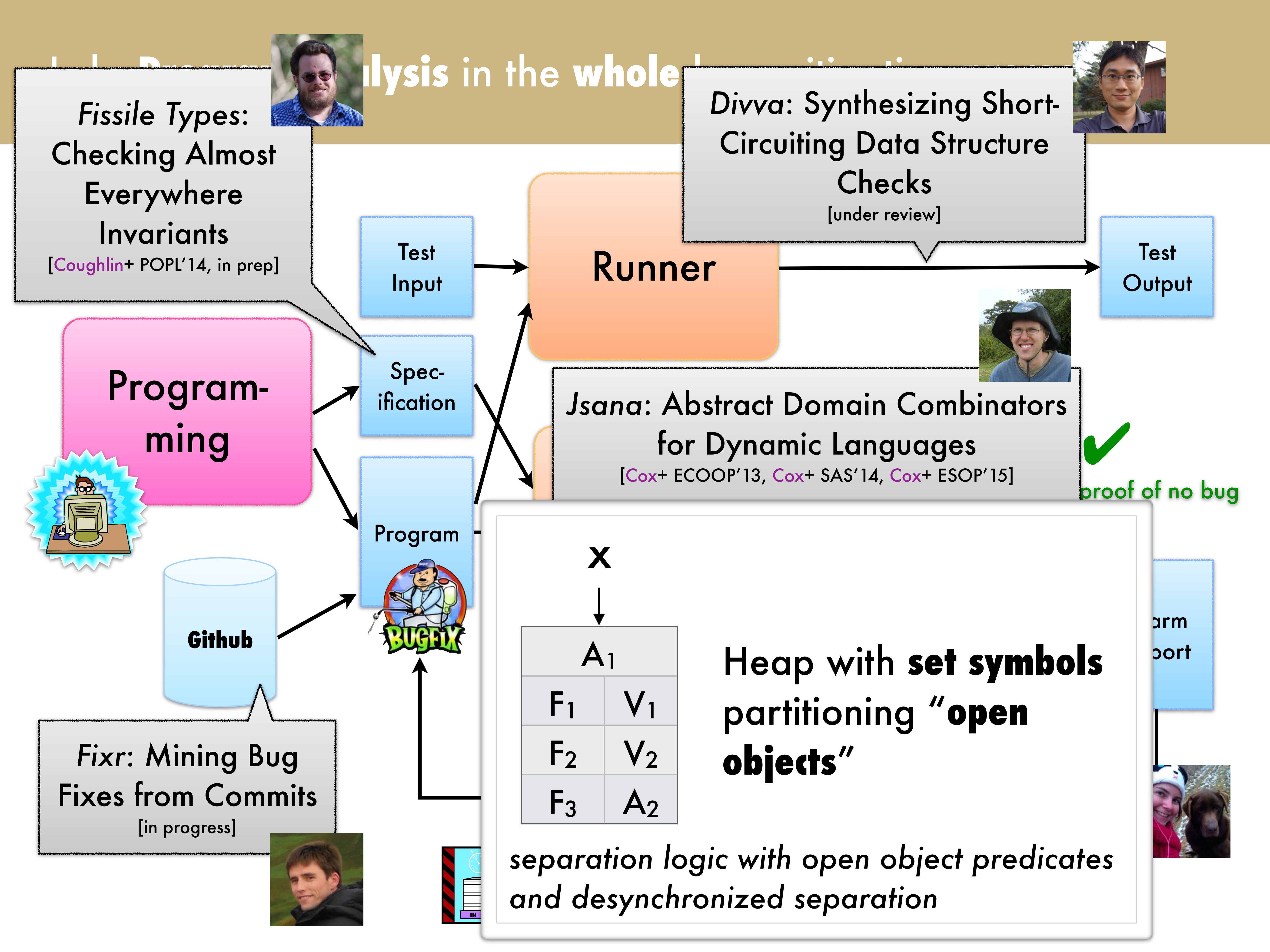
in the whole
Fissile Types: Checking Almost Everywhere Invariants
[Coughlin+ POPL'14, in prep]
Divva: Synthesizing Short-Circuiting Data Structure Checks
[under review]
Test Input
Runner
Test Output
Program-ming
Spec-ification
Program
BUGFIX
Github
Verifier
proof of no bug
Alarm Report
Thresher: Goal-Directed Refutation Analysis
[Blackshear+ SAS'11, Blackshear+ PLDI'13, under review]
Triaging
IN
OUT



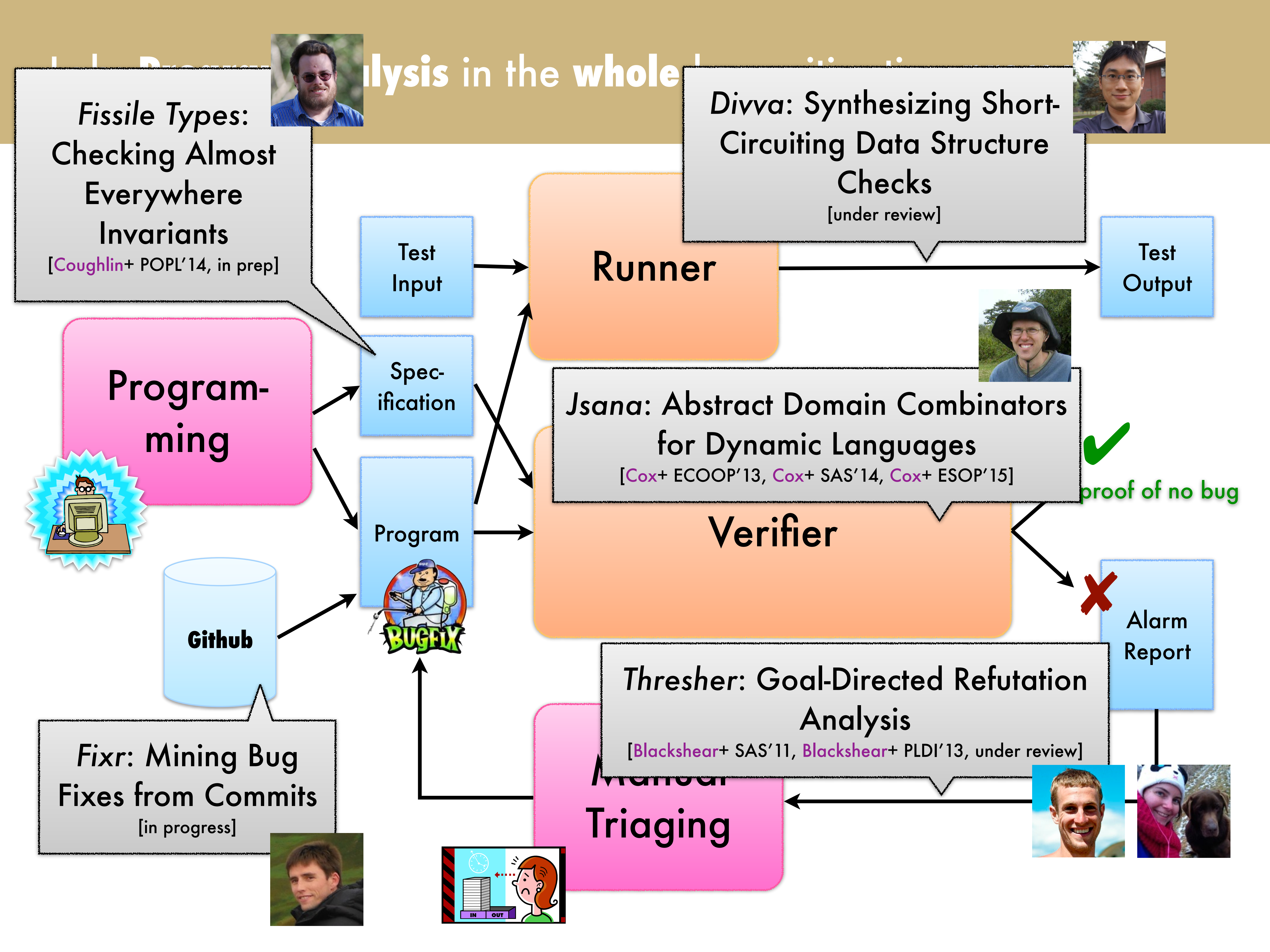
lysis in the whole
Fissile Types: Checking Almost Everywhere Invariants
[Coughlin+ POPL'14, in prep]
Divva: Synthesizing Short-Circuiting Data Structure Checks
[under review]
Test Input
Runner
Test Output
Program-ming
Spec-ification
Program
Verifier
proof of no bug
Alarm Report
Github
BUGFIX
Thresher: Goal-Directed Refutation Analysis
[Blackshear+ SAS'11, Blackshear+ PLDI'13, under review]
Triaging
Fixr: Mining Bug Fixes from Commits
[in progress]
IN
OUT



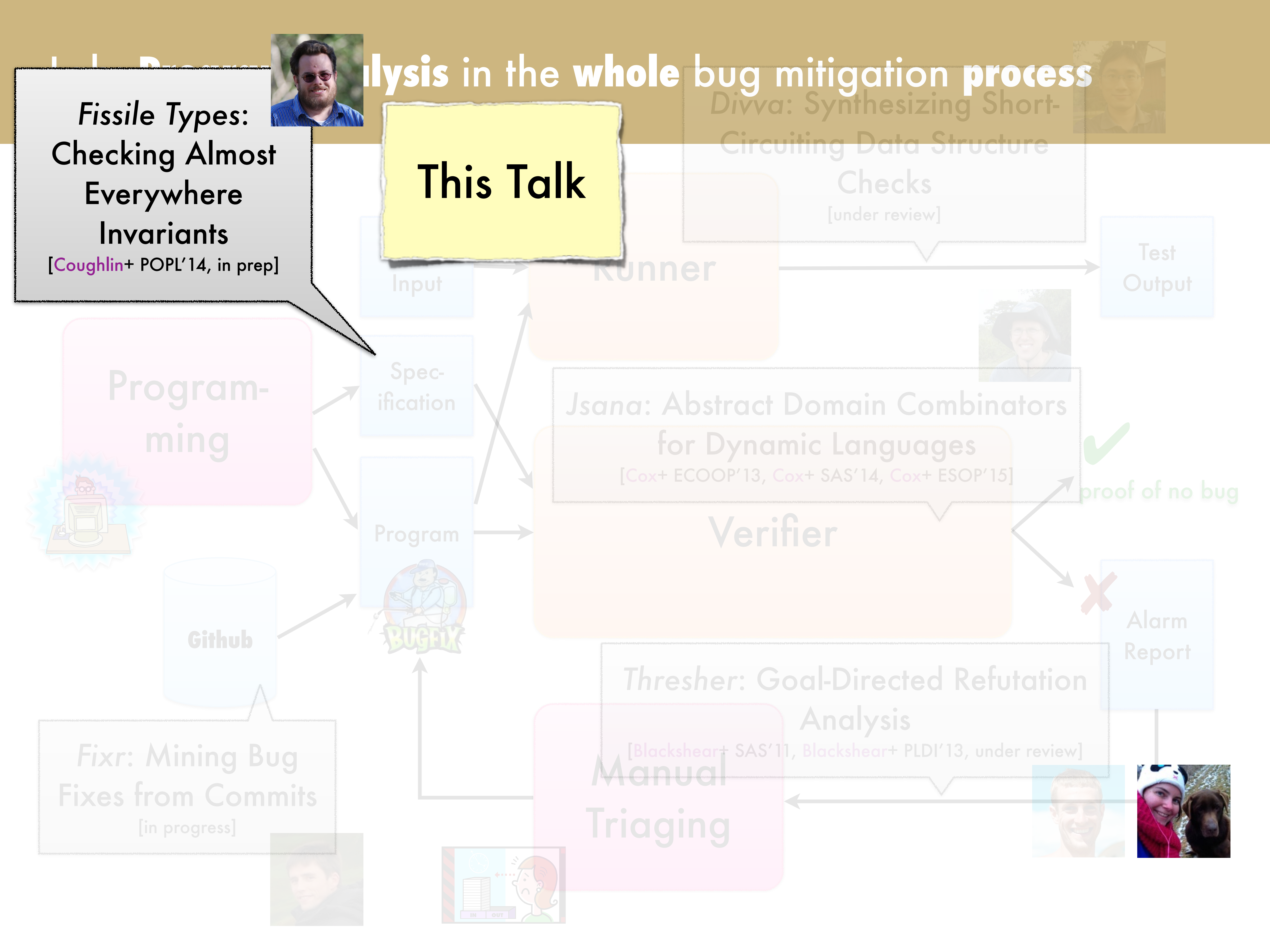
Fissile Types: Checking Almost Everywhere Invariants
[Coughlin+ POPL'14, in prep]
Divva: Synthesizing Short-Circuiting Data Structure Checks
[under review]
Test Input
Runner
Test Output
Program-ming
Spec-ification
Program
BUGFIX
Jsana: Abstract Domain Combinators for Dynamic Languages
[Cox+ ECOOP'13, Cox+ SAS'14, Cox+ ESOP'15]
proof of no bug
Verifier
Alarm Report
Github
Thresher: Goal-Directed Refutation Analysis
[Blackshear+ SAS'11, Blackshear+ PLDI'13, under review]
Triaging
Fixr: Mining Bug Fixes from Commits
[in progress]
IN
OUT



lysis in the whole
Fissile Types: Checking Almost Everywhere Invariants
[Coughlin+ POPL'14, in prep]
Divva: Synthesizing Short-Circuiting Data Structure Checks
[under review]
Test Input
Runner
Test Output
Program-ming
Spec-ification
Program
Jsana: Abstract Domain Combinators for Dynamic Languages
[Cox+ ECOOP'13, Cox+ SAS'14, Cox+ ESOP'15]
proof of no bug
arm
port
Github
BUGFIX
Fixr: Mining Bug Fixes from Commits
[in progress]
x
A1
F1 V1
F2 V2
F3 A2
Heap with set symbols partitioning "open objects"
separation logic with open object predicates and desynchronized separation



lysis in the whole
Fissile Types: Checking Almost Everywhere Invariants
[Coughlin+ POPL'14, in prep]
Divva: Synthesizing Short-Circuiting Data Structure Checks
[under review]
Test Input
Runner
Test Output
Program-ming
Spec-ification
Program
BUGFIX
Jsana: Abstract Domain Combinators for Dynamic Languages
[Cox+ ECOOP'13, Cox+ SAS'14, Cox+ ESOP'15]
proof of no bug
Verifier
Alarm Report
Github
Thresher: Goal-Directed Refutation Analysis
[Blackshear+ SAS'11, Blackshear+ PLDI'13, under review]
Fixr: Mining Bug Fixes from Commits
[in progress]
Triaging
IN
OUT


lysis in the whole bug mitigation process
Fissile Types: Checking Almost Everywhere Invariants
[Coughlin+ POPL'14, in prep]
This Talk
Divva: Synthesizing Short-Circuiting Data Structure Checks
[under review]
Input
Runner
Test Output
Program-ming
Spec-ification
Program
Jsana: Abstract Domain Combinators for Dynamic Languages
[Cox+ ECOOP'13, Cox+ SAS'14, Cox+ ESOP'15]
Verifier
proof of no bug
Alarm Report
Github
BUGFIX
Thresher: Goal-Directed Refutation Analysis
[Blackshear+ SAS'11, Blackshear+ PLDI'13, under review]
Manual Triaging
Fixr: Mining Bug Fixes from Commits
[in progress]

Type-Intertwined Heap Analysis


Devin Coughlin


Ross Holland


Bor-Yuh Evan Chang

University of Colorado Boulder

Aarhus University
May 18, 2015

PLV

How to **type check** a program that is **almost** well-typed?

How to **type check** a program that is **almost** well-typed?

almost?

A motivating example

Property of interest:
reflective method call safety

Specification system:
dependent-refinement types

A motivating example

Property of interest:

reflective method call safety

type safety

Specification system:

dependent-refinement types

simple types

A motivating example

Property of interest:
reflective method call safety

Specification system:
dependent-refinement types

Reflective method call is **dispatch** based **on a run-time value**

```
class Callback
 var sel: Str
 var obj: Obj

 def call()
  this.obj.[this.sel]()
```

Reflective method call is **dispatch** based **on a run-time value**

```
class Callback
  var sel: Str
  var obj: Obj

  def call()
    this.obj.[this.sel]()
```

Calls method with name (selector) stored in on `sel` object stored in `obj`

Reflective method call is **dispatch** based **on a run-time value**

```
class Callback
 var sel: Str
 var obj: Obj

 def call()
  this.obj.[this.sel]()
```

Calls method with name (selector) stored in on `sel` object stored in `obj`

If `sel` held string "`notifyClick`" would call `notifyClick()` on `obj`.

Reflective method call is **dispatch** based **on a run-time value**

```
class Callback
  var sel: Str
  var obj: Obj

  def call()
    this.obj.[this.sel]()
```

Calls method with name (selector) stored in on `sel` object stored in `obj`

Run-time error if `obj` does not **respond to** `sel` – i.e., method does not exist

A **dependent-refinement type** expresses the required **relationship**

```
class Callback
 var sel: Str
 var obj: Obj

 def call()
  this.obj.[this.sel]()
```

A **dependent-refinement type** expresses the required **relationship**

```
class Callback
 var sel: Str
 var obj: Obj | r2 sel

 def call()
  this.obj.[this.sel]()
```

obj must “respond to” sel

A **dependent-refinement type** expresses the required **relationship**

```
class Callback
 var sel: Str
 var obj: Obj | r2 sel

 def call()
  this.obj.[this.sel]()
```

`obj` must "respond to" `sel`

Shorthand for $\mathrm{obj} :: \{\nu : \mathsf{Obj} \mid \nu\ \mathsf{r2}\ \mathrm{sel}\}$

A **dependent-refinement type** expresses the required **relationship**

```
class Callback
  var sel: Str
  var obj: Obj | r2 sel

  def call()
    this.obj.[this.sel]()
```

obj must "respond to" sel

A **dependent-refinement type** expresses the required **relationship**

```
class Callback
  var sel: Str
  var obj: Obj | r2 sel

  def call()
    this.obj.[this.sel]()
```

obj must "respond to" sel

Guarantees no `MethodNotFound` error in `call()`

A **dependent-refinement type** expresses the required **relationship**

```
class Callback
  var sel: Str
  var obj: Obj | r2 sel

  def call()
    this.obj.[this.sel]()
```

Array indexing safety is a similar **relationship**

```
class Iterator
 var idx: Int
 var buf: Obj[] | indexedBy idx

 def get(): Obj
  return this.buf[this.idx]
```

Array indexing safety is a similar **relationship**

```
class Iterator
 var idx: Int
 var buf: Obj[] | indexedBy idx

 def get(): Obj
  return this.buf[this.idx]
```

`idx` must be a valid index into `buf`

Array indexing safety is a similar **relationship**

idx must be a valid index into buf

```
class Iterator
 var idx: Int
 var buf: Obj[] | indexedBy idx

 def get(): Obj
  return this.buf[this.idx]
```

Guarantees no `ArrayOutOfBounds` error here

Array indexing safety is a similar **relationship**

idx must be a valid index into buf

```
class Iterator
 var idx: Int
 var buf: Obj[] | indexedBy idx

 def get(): Obj
  return this.buf[this.idx]
```

Recurring theme: **Relationships** are important to **many safety properties**

Relationships can be broken

```
class Callback
 var sel: Str
 var obj: Obj | r2 sel

 def call()
  this.obj.[this.sel]()
```

Relationships can be broken

```
class Callback
 var sel: Str
 var obj: Obj | r2 sel

 def call()
  this.obj.[this.sel]()

 def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```

Relationships can be broken

obj must **always** respond to sel

```
class Callback
  var sel: Str
  var obj: Obj | r2 sel

  def call()
    this.obj.[this.sel]()

  def update(s: Str, o: Obj | r2 s)
    this.sel = s
    this.obj = o
```

Relationships can be broken

```
class Callback
 var sel: Str
 var obj: Obj | r2 sel

 def call()
  this.obj.[this.sel]()

 def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```

`obj` must **always** respond to `sel`

o guaranteed to respond to s

Relationships can be broken

```
class Callback
 var sel: Str
 var obj: Obj | r2 sel

 def call()
  this.obj.[this.sel]()

 def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```

`obj` must **always** respond to `sel`

`o` guaranteed to respond to `s`

Type error: **old** `obj` may not respond to **new** `sel`

Relationships can be broken

```
class Callback
 var sel: Str
 var obj: Obj | r2 sel

 def call()
  this.obj.[this.sel]()

 def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```

`obj` must **always** respond to `sel`

False alarm: no run-time `MethodNotFound`

`o` guaranteed to respond to `s`

Type error: **old** `obj` may not respond to **new** `sel`

Why is this a false alarm?

```
class Callback
 var sel: Str
 var obj: Obj | r2 sel

 def call()
  this.obj.[this.sel]()

 def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```

Why is this a false alarm?

```
class Callback
 var sel: Str
 var obj: Obj | r2 sel

 def call()
  this.obj.[this.sel]()

 def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```

Reasoning by **global invariant**: call safe because relationship holds

Why is this a false alarm?

```
class Callback
  var sel: Str
  var obj: Obj | r2 sel

  def call()
    this.obj.[this.sel]()

  def update(s: Str, o: Obj | r2 s)
    this.sel = s
    this.obj = o
```

Reasoning by **global invariant**: call safe because relationship holds

Reasoning about **local effects** of imperative updates

Why is this a false alarm?

```
class Callback
  var sel: Str
  var obj: Obj | r2 sel

  def call()
    this.obj.[this.sel]()

  def update(s: Str, o: Obj | r2 s)
    this.sel = s
    this.obj = o
```

Reasoning by **global invariant**: call safe because relationship holds

Reasoning about **local effects** of imperative updates

Why is this a false alarm?

```
class Callback
  var sel: Str
  var obj: Obj | r2 sel

  def call()
    this.obj.[this.sel]()

  def update(s: Str, o: Obj | r2 s)
    this.sel = s
    this.obj = o
```

Reasoning by **global invariant**: call safe because relationship holds

Relationship violated

Reasoning about **local effects** of imperative updates

Why is this a false alarm?

```
class Callback
  var sel: Str
  var obj: Obj | r2 sel

  def call()
    this.obj.[this.sel]()

  def update(s: Str, o: Obj | r2 s)
    this.sel = s
    this.obj = o
```

Reasoning by **global invariant**: call safe because relationship holds

Relationship violated

Relationship restored

Reasoning about **local effects** of imperative updates

Why is this a false alarm?

two reasoning styles

```
class
  var s
  var obj: Obj | r2 sel

  def call()
   this.obj.[this.sel]()

  def update(s: Str, o: Obj | r2 s)
   this.sel = s
   this.obj = o
```

Reasoning by **global invariant**: call safe because relationship holds

Relationship violated

Relationship restored

Reasoning about **local effects** of imperative updates

two **reasoning styles**

Idea: Selectively **alternate** between and **intertwine** these two **reasoning styles** in verification

Idea: Selectively **alternate** between and **intertwine** these **two reasoning styles** in verification

$x : \tau, \ldots$

flow-insensitive typing

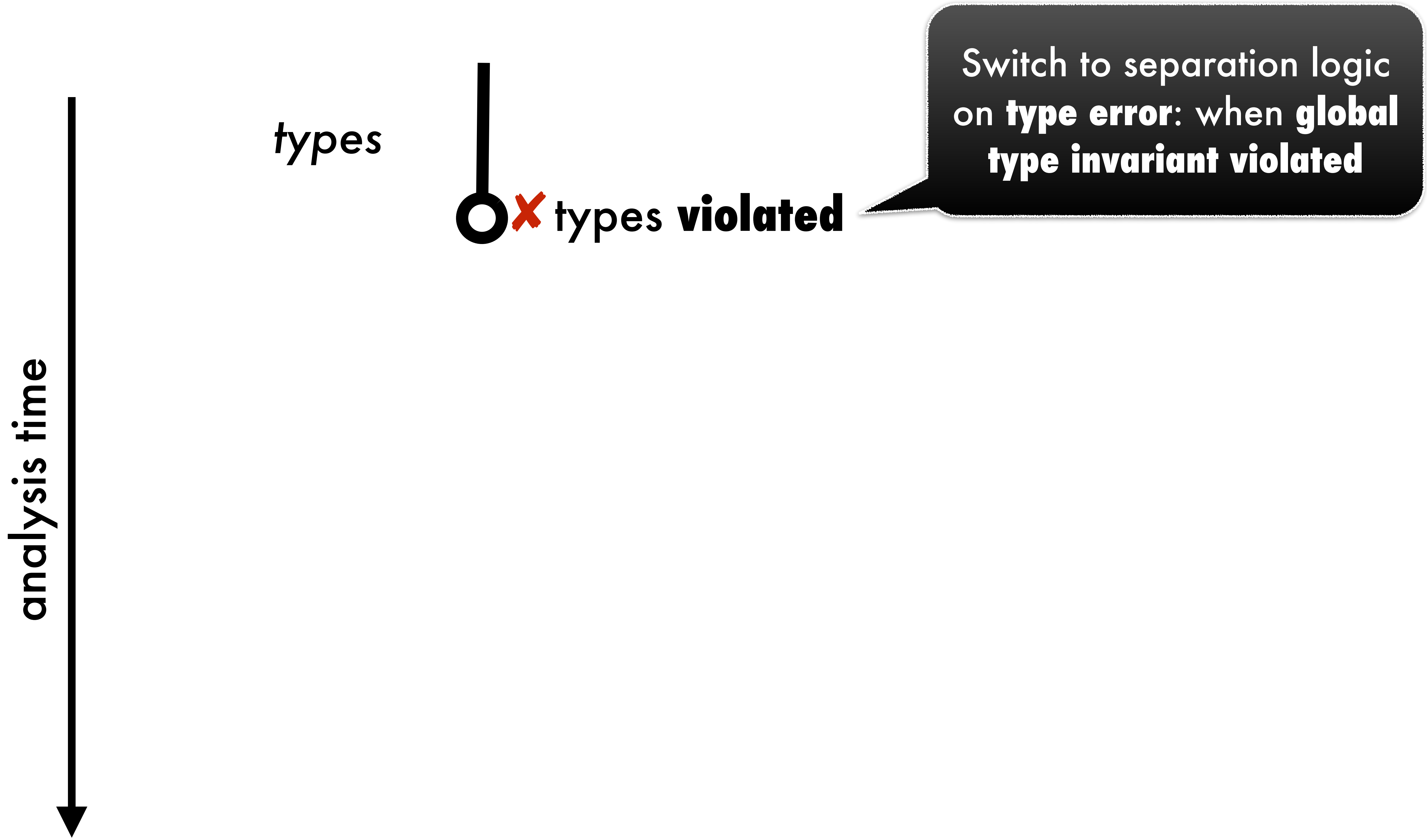
$x \mapsto \hat{a} * \hat{a} \cdot f \mapsto \hat{v} * \cdots$

flow-sensitive abstract interpretation

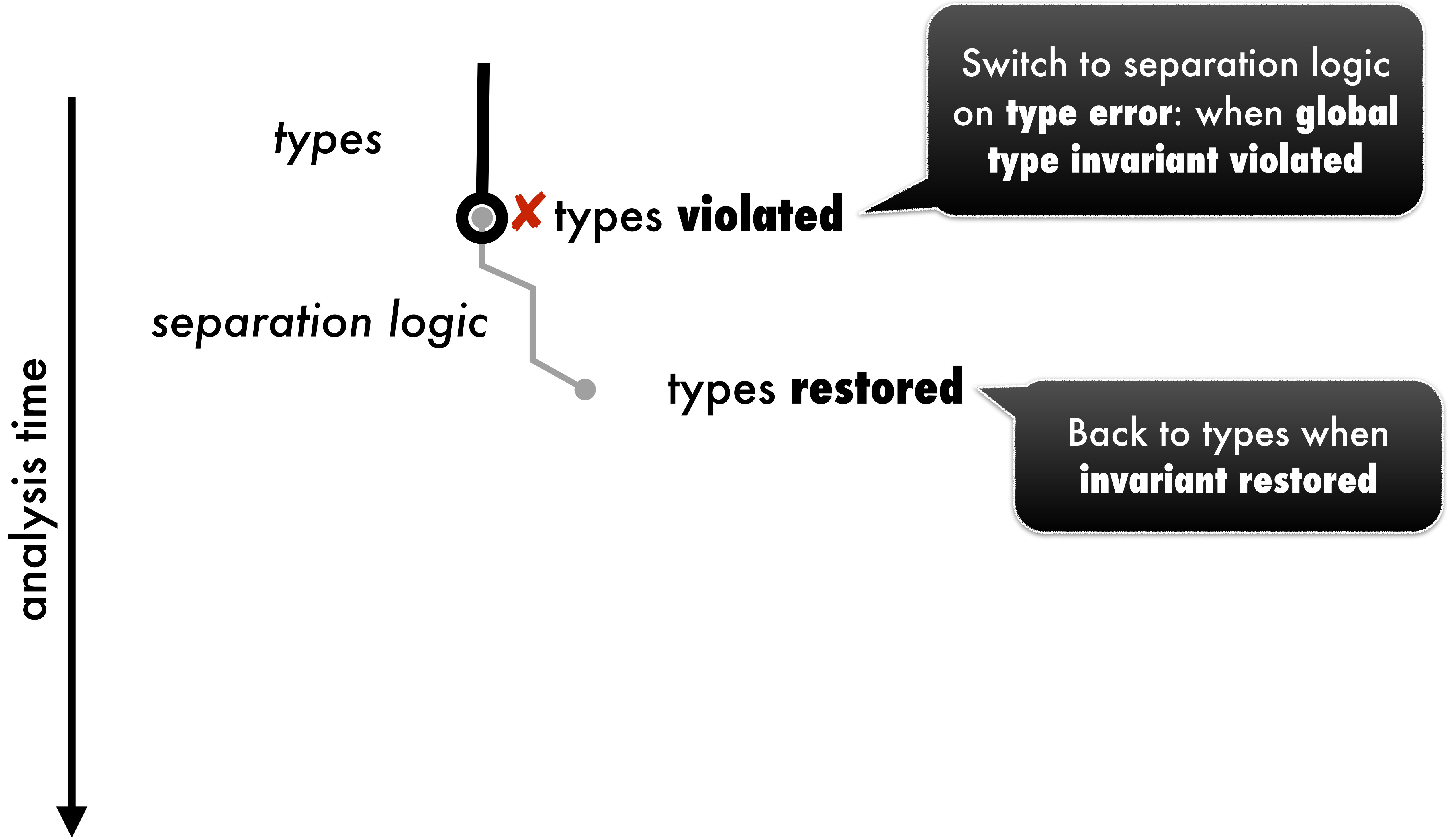
Verification of **almost-everywhere invariants** with **intertwined** type- and separation logic-based analysis

analysis time

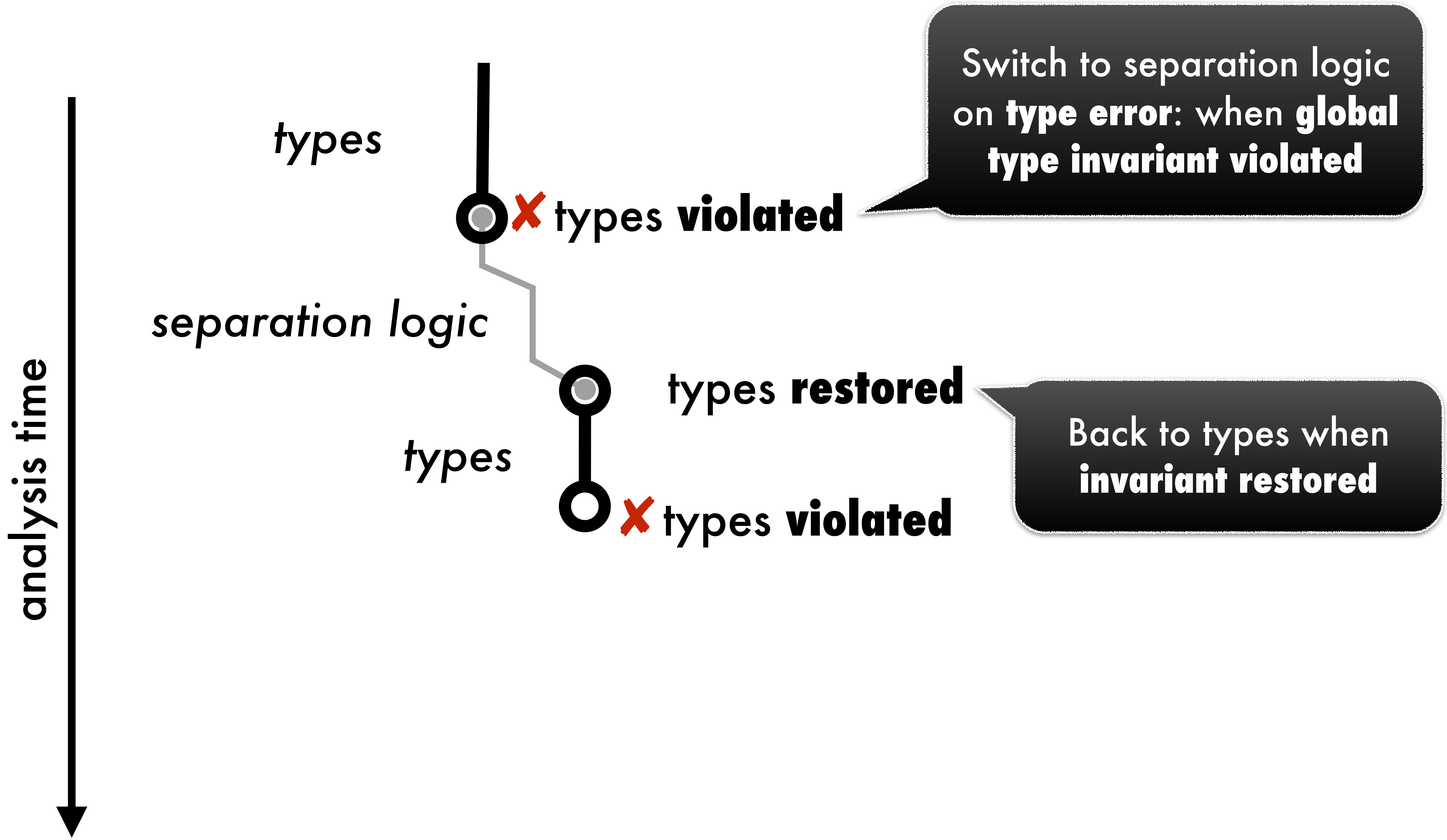
Verification of **almost-everywhere invariants** with **intertwined** type- and separation logic-based analysis



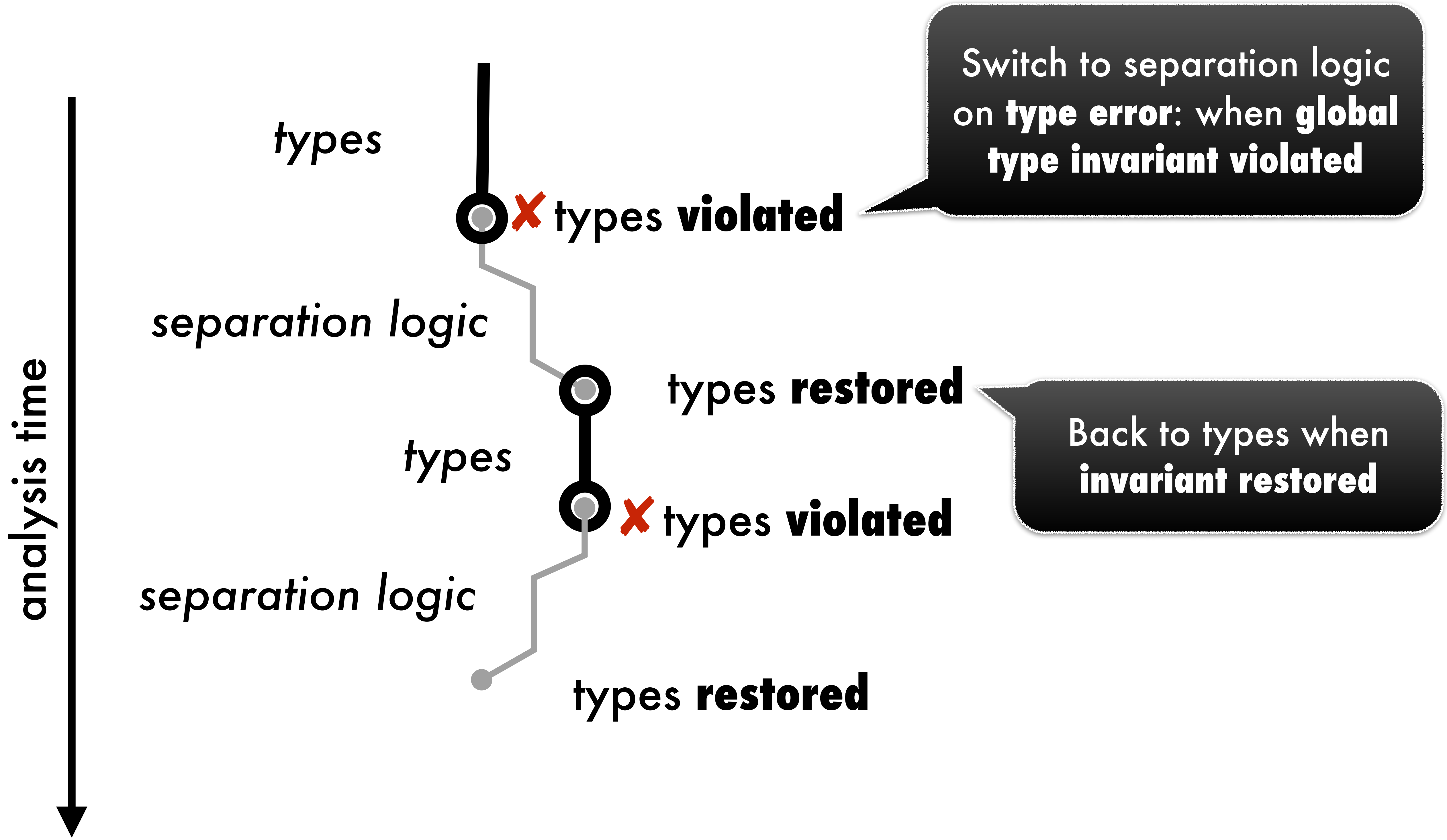
Verification of **almost-everywhere invariants** with **intertwined** type- and separation logic-based analysis



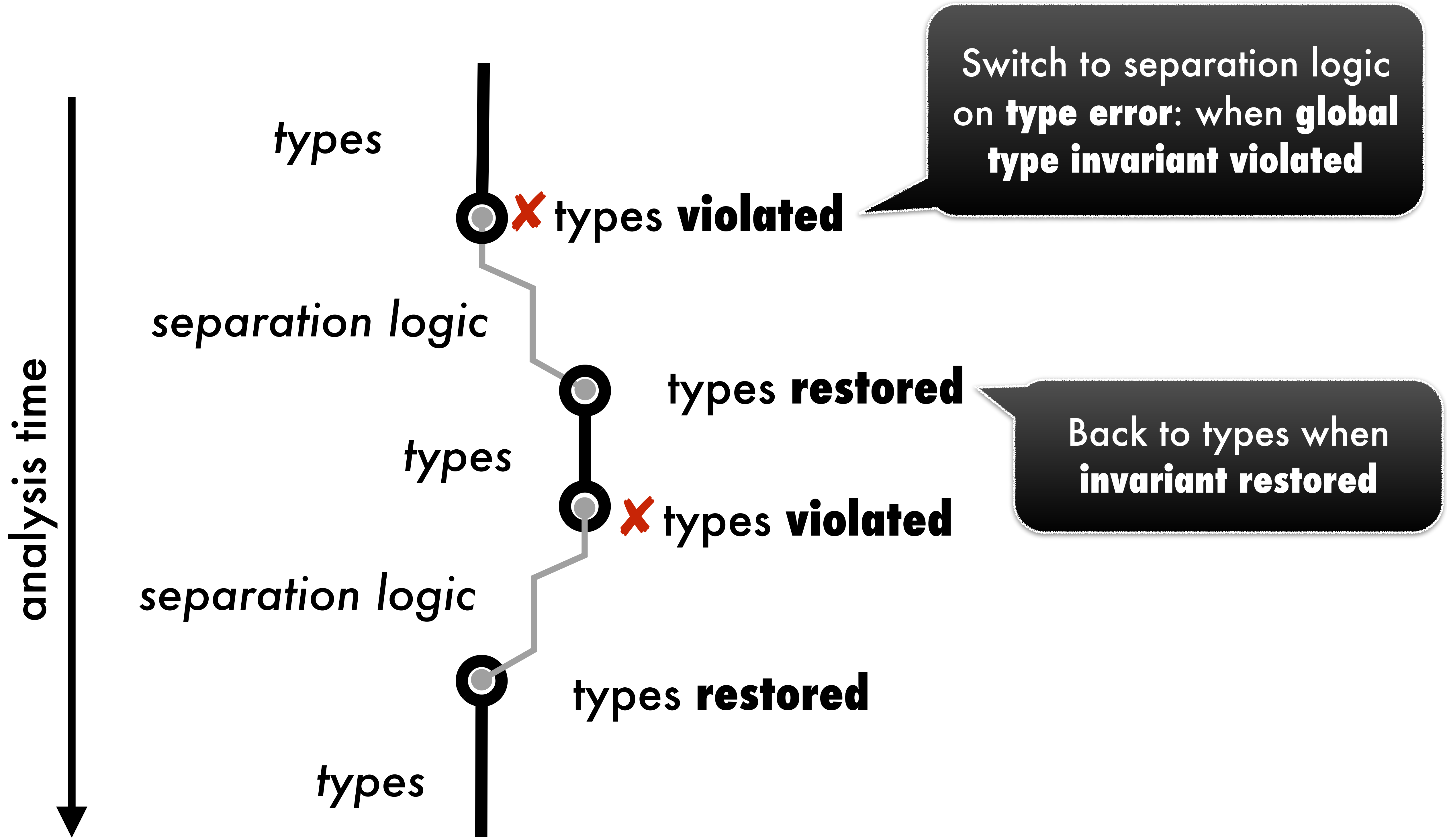
Verification of **almost-everywhere invariants** with **intertwined** type- and separation logic-based analysis



Verification of **almost-everywhere invariants** with **intertwined** type- and separation logic-based analysis



Verification of **almost-everywhere invariants** with **intertwined** type- and separation logic-based analysis



Key Ideas

ok Tolerating temporary violations with **almost type-consistent heaps**

Coughlin and Chang. *POPL* 2014.

Key Ideas

ok Tolerating temporary violations with **almost type-consistent heaps**

Coughlin and Chang. *POPL* 2014.

Type-intertwined framing with **gated separation**

Under preparation.

Key Ideas

ok Tolerating temporary violations with **almost type-consistent heaps**

Coughlin and Chang. *POPL* 2014.

$\lhd\!\!\!\!\ast$ Type-intertwined framing with **gated separation**

Under preparation.

Key Ideas

ok Tolerating temporary violations with **almost type-consistent heaps**

Coughlin and Chang. *POPL* 2014.

Type-intertwined framing with **gated separation**

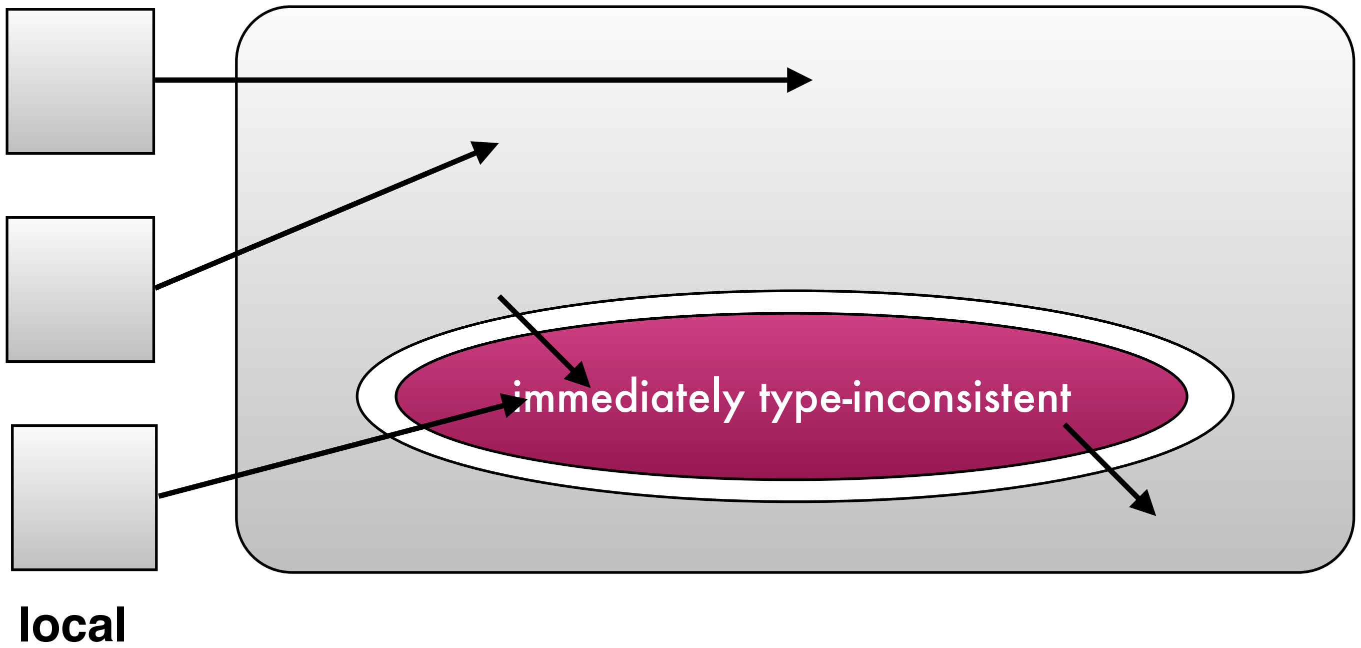
Under preparation.

Extend separation logic with a **summary** for an **almost type-consistent heap**

ok

Extend separation logic with a **summary** for an **almost type-consistent heap**

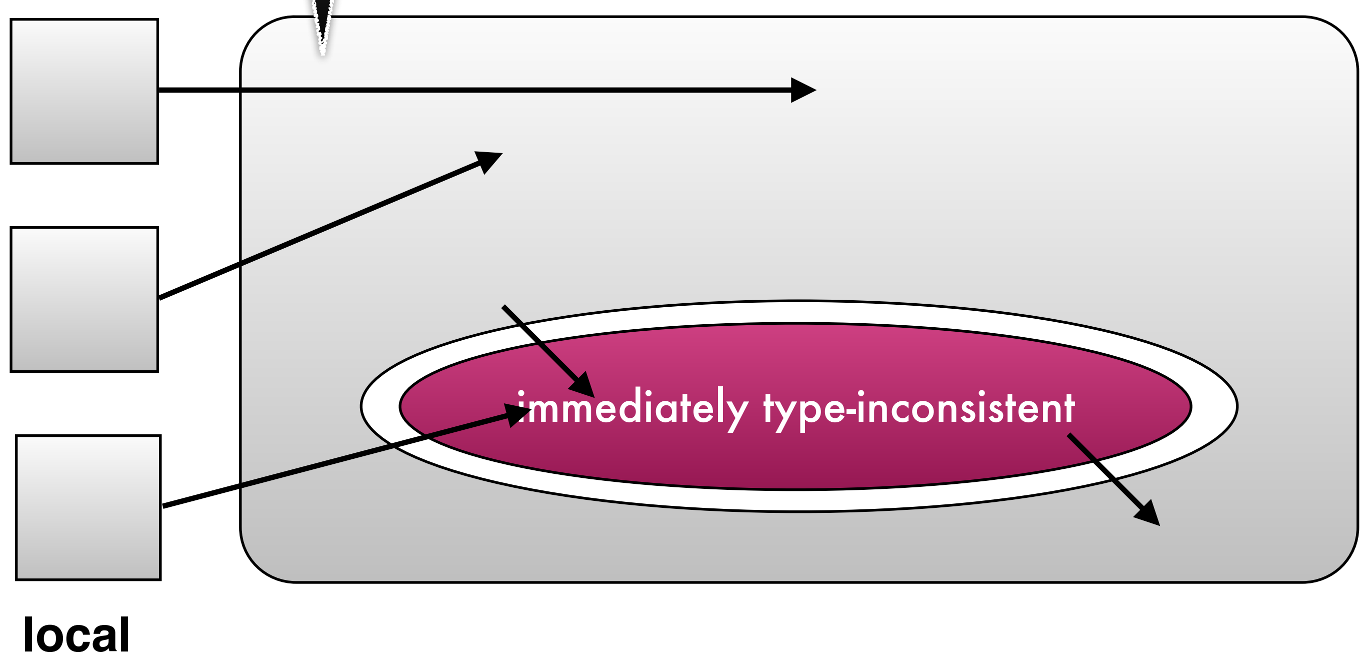
ok



Extend separation logic with a **summary** for an **almost type-consistent heap**

ok

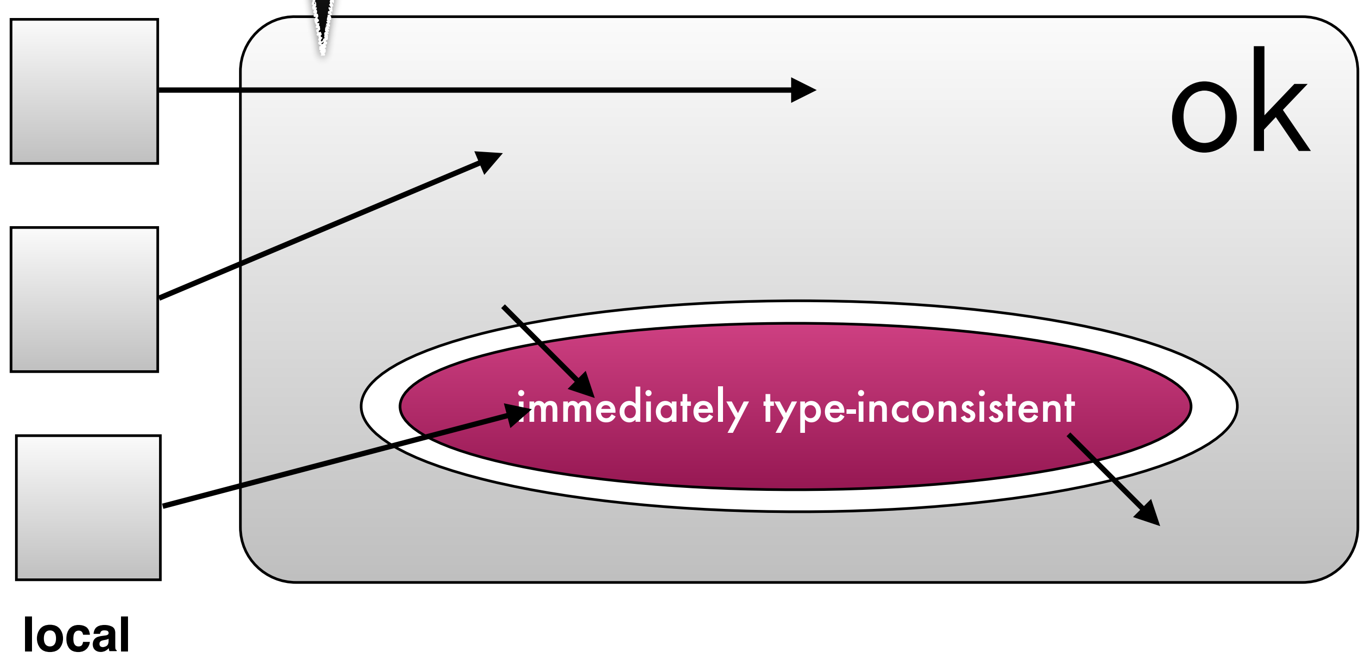
Summarize cells that are **not immediately type-inconsistent** into single formula literal



Extend separation logic with a **summary** for an **almost type-consistent heap**

ok

Summarize cells that are **not immediately type-inconsistent** into single formula literal

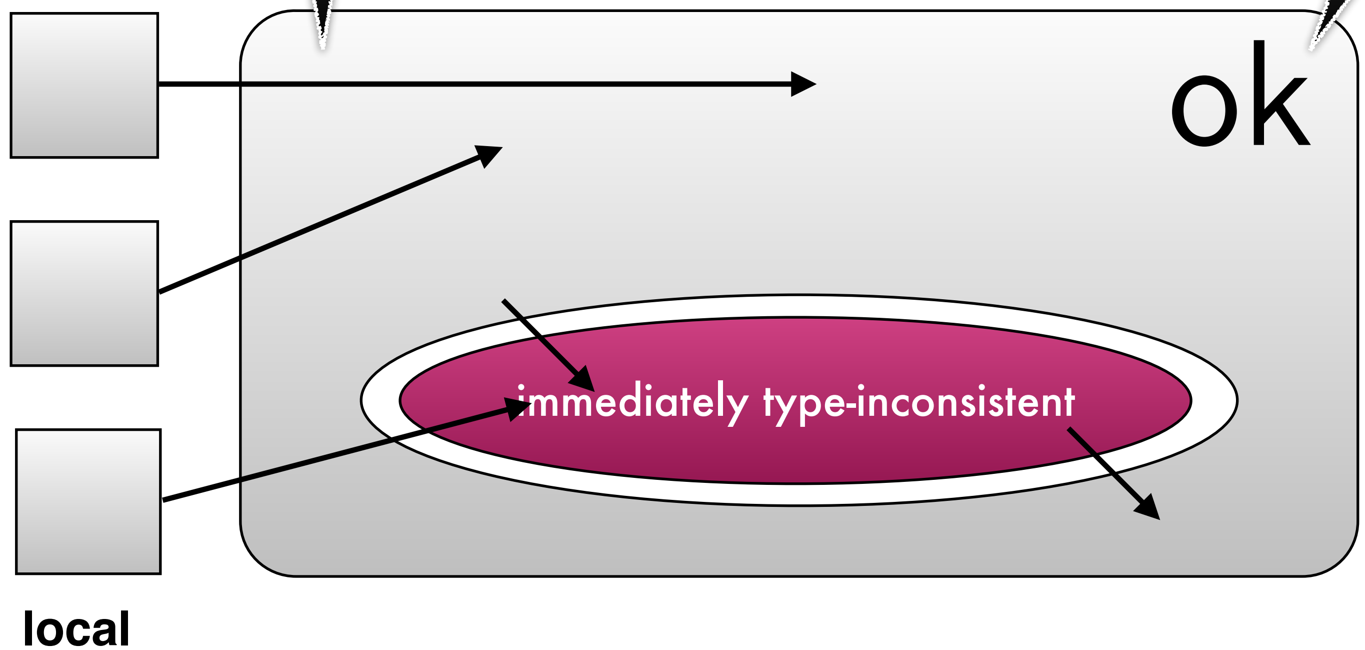


Extend separation logic with a **summary** for an **almost type-consistent heap**

ok

Summarize cells that are **not immediately type-inconsistent** into single formula literal

Describes storage **without** explicitly **enumerating** it

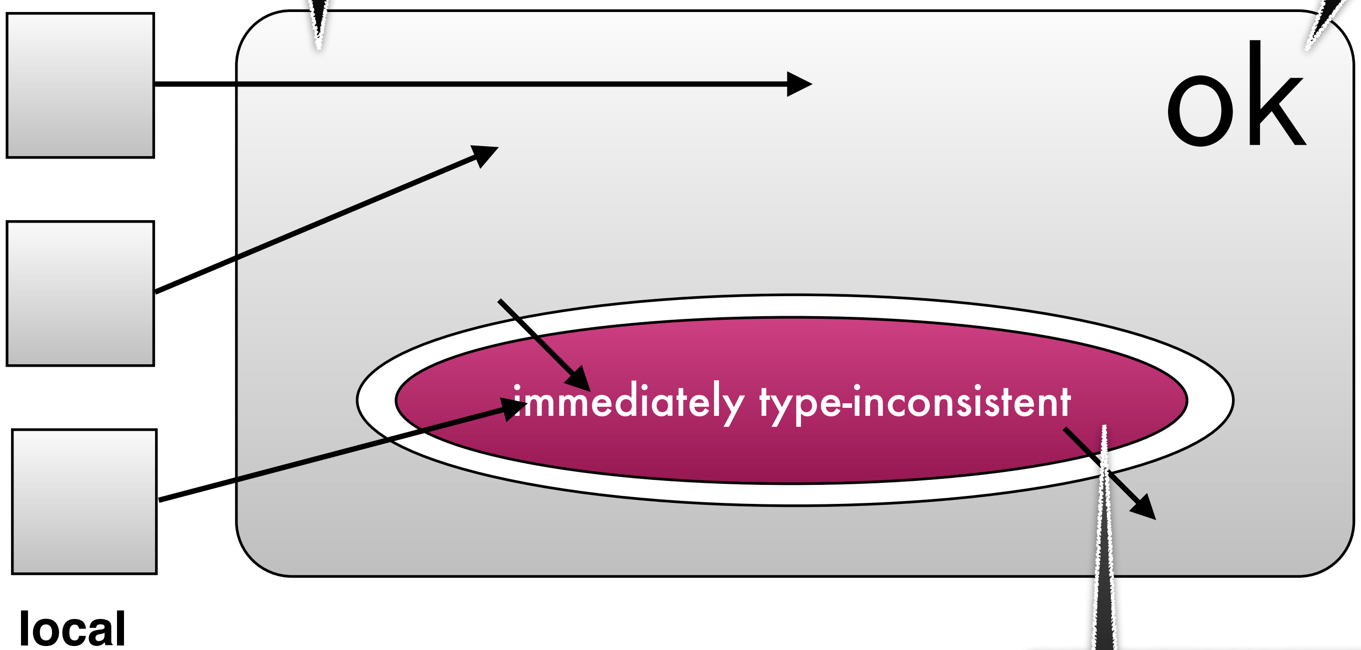


Extend separation logic with a **summary** for an **almost type-consistent heap**

ok

Summarize cells that are **not immediately type-inconsistent** into single formula literal

Describes storage **without** explicitly **enumerating** it



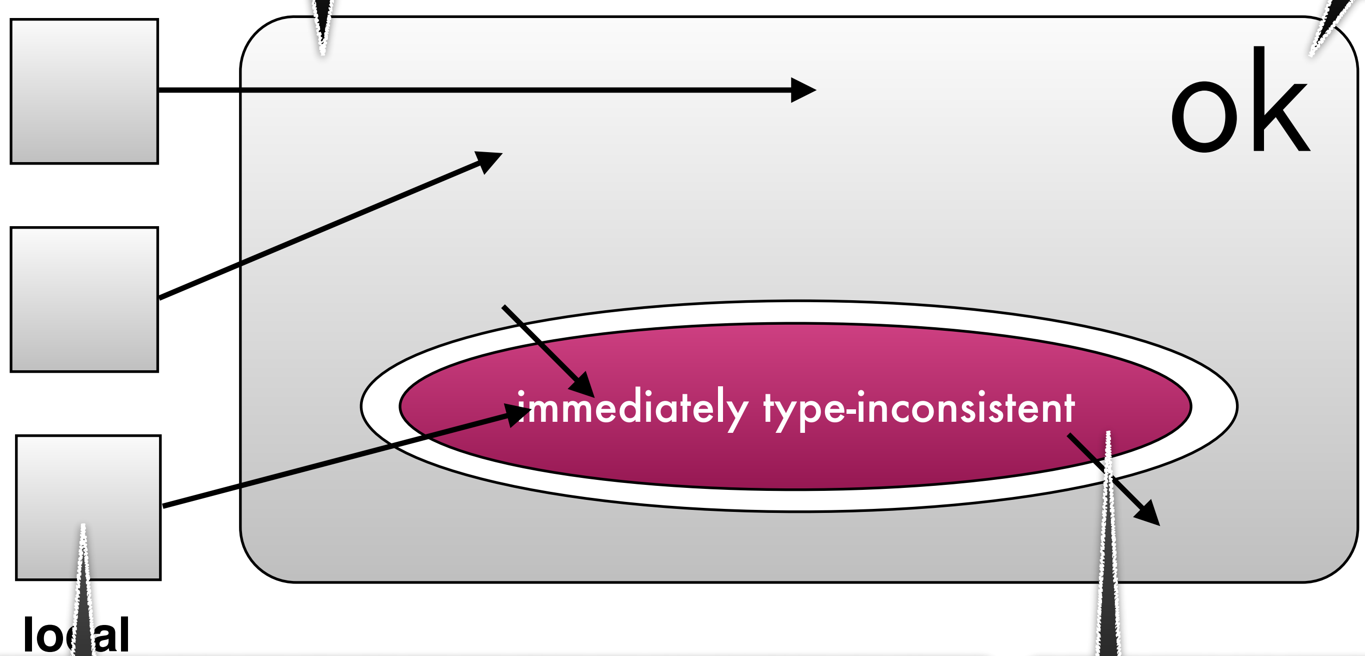
Value stored in location **differs from location's declared type**

Extend separation logic with a **summary** for an **almost type-consistent heap**

ok

Summarize cells that are **not immediately type-inconsistent** into single formula literal

Describes storage **without** explicitly **enumerating** it



Not immediately type-inconsistent but still transitively type-inconsistent

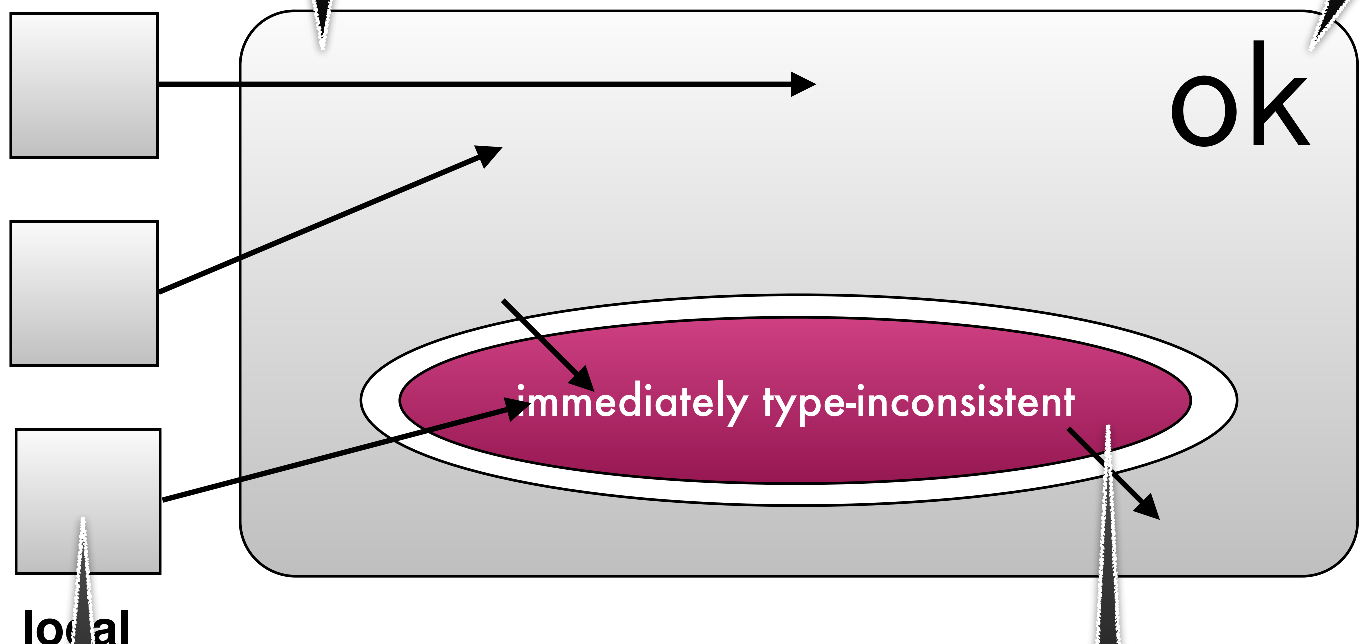
Value stored in location **differs from location's declared type**

Extend separation logic with a **summary** for an **almost type-consistent heap**

ok

Summarize cells that are **not immediately type-inconsistent** into single formula literal

Describes storage **without** explicitly **enumerating** it



$$\mathsf{ok} \circledast \hat{a} \cdot f \mapsto \hat{v}$$

Not immediately type-inconsistent but still transitively type-inconsistent

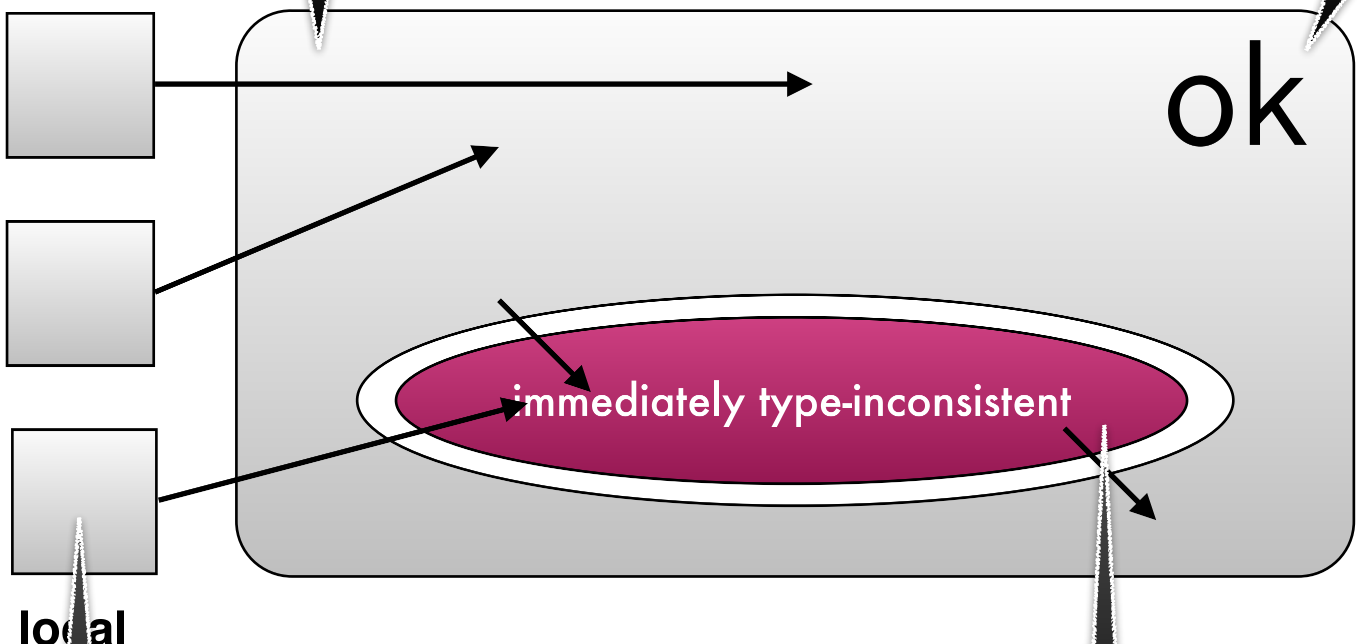
Value stored in location **differs from location's declared type**

Extend separation logic with a **summary** for an **almost type-consistent heap**

ok

Summarize cells that are **not immediately type-inconsistent** into single formula literal

Describes storage **without** explicitly **enumerating** it



$$\mathsf{ok} * \hat{a}\cdot f \mapsto \hat{v}$$

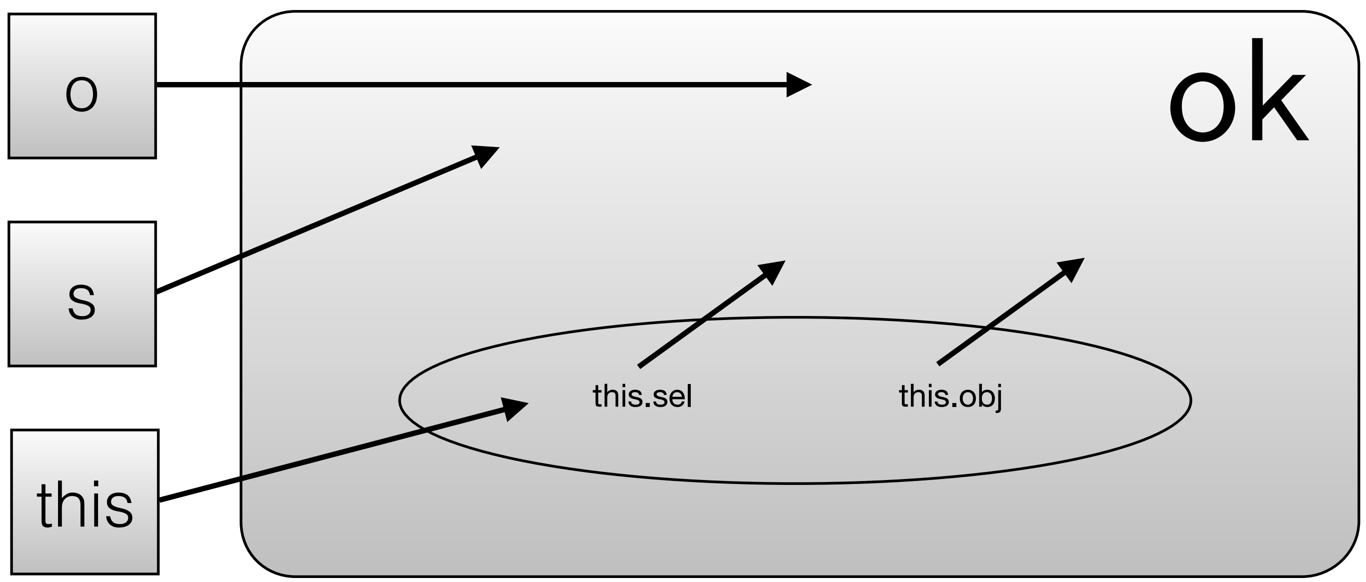
Not immediately type-inconsistent but still transitively type-inconsistent

Value stored in location **differs from location's declared type**

Split heap into two regions: **almost type-consistent** and (potentially) **immediately type-inconsistent**

Separation logic can **materialize** from the almost type-consistent summary

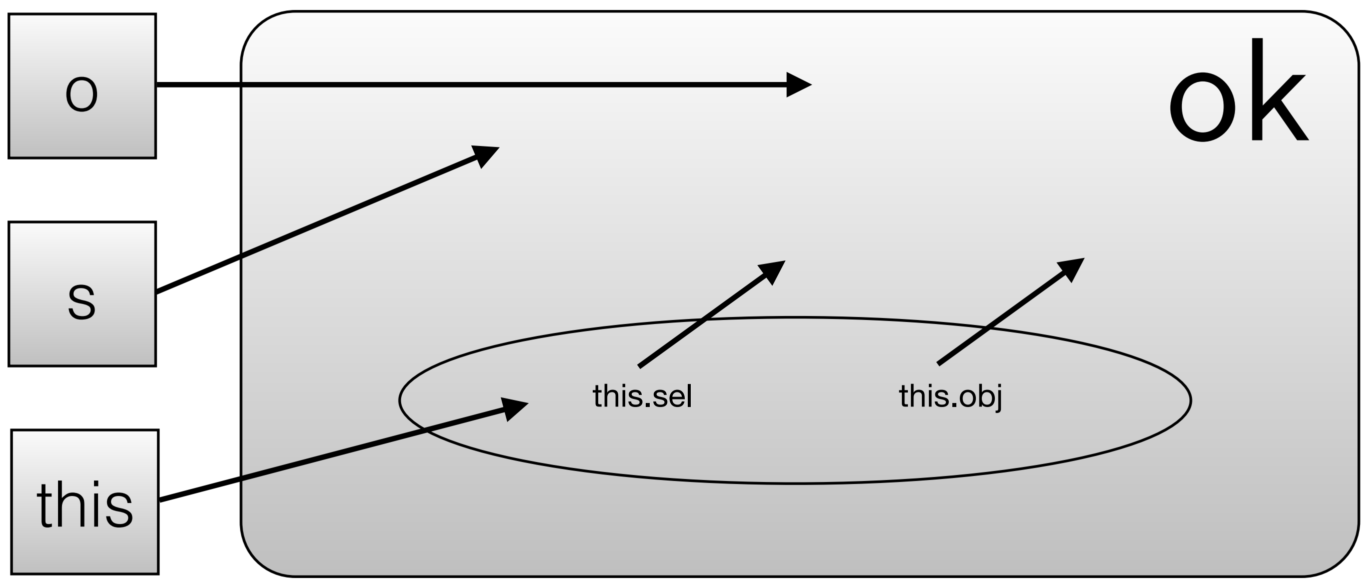
```
def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```



Separation logic can **materialize** from the almost type-consistent summary

Handoff to separation logic on type error, **heap** is **consistent** with declared types

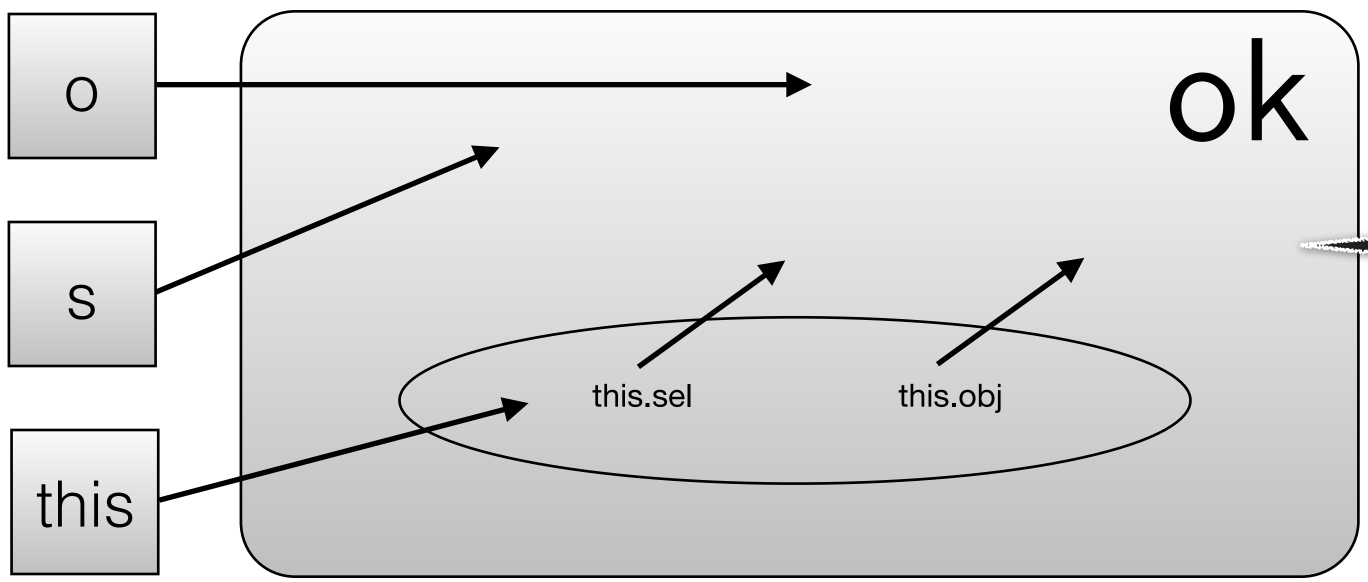
```
def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```



Separation logic can **materialize** from the almost type-consistent summary

Handoff to separation logic on type error, **heap** is **consistent** with declared types

```
def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```

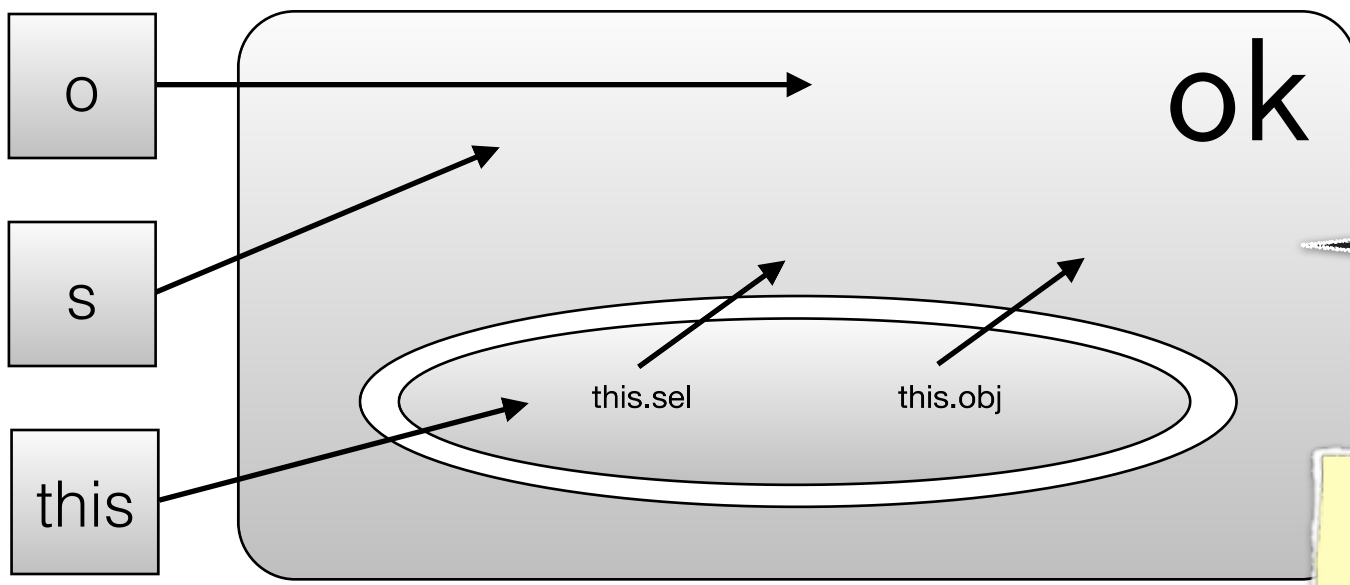


Entire heap consists of the almost type-consistent summary

Separation logic can **materialize** from the almost type-consistent summary

Handoff to separation logic on type error, **heap** is **consistent** with declared types

```
def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```



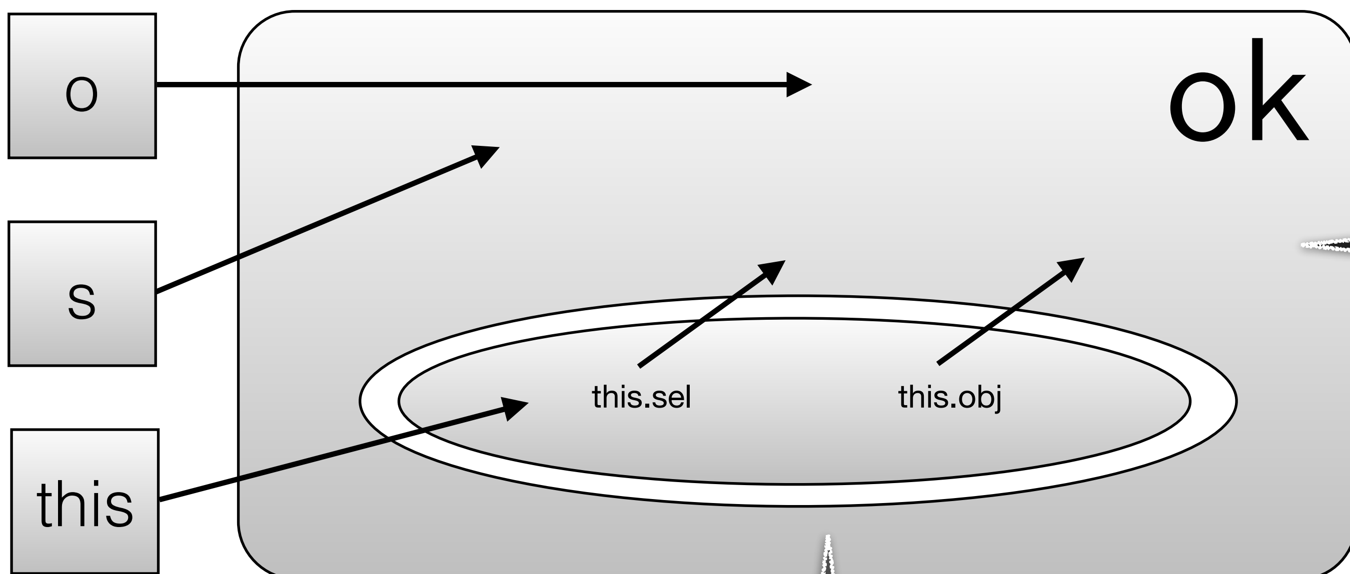
Entire heap consists of the almost type-consistent summary

materialize
`this.sel`
`this.obj`

Separation logic can **materialize** from the almost type-consistent summary

Handoff to separation logic on type error, **heap** is **consistent** with declared types

```
def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```

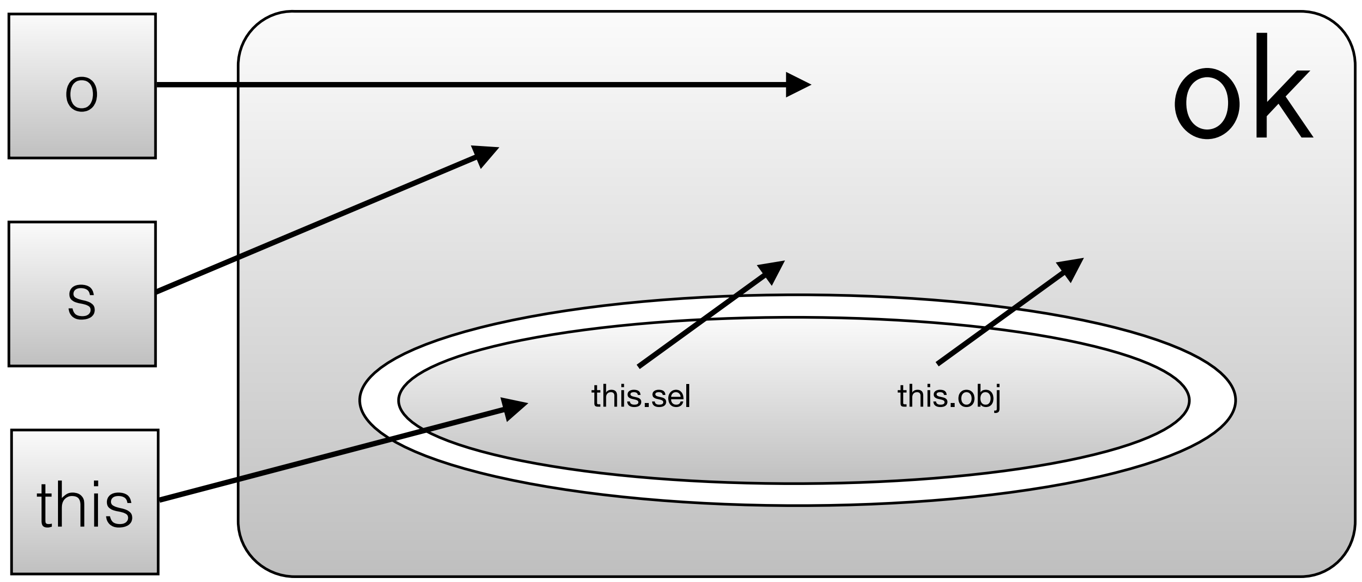


Entire heap consists of the almost type-consistent summary

Analysis reasons explicitly about contents of materialized cells

Values in **materialized** storage allowed to **differ from declared types**

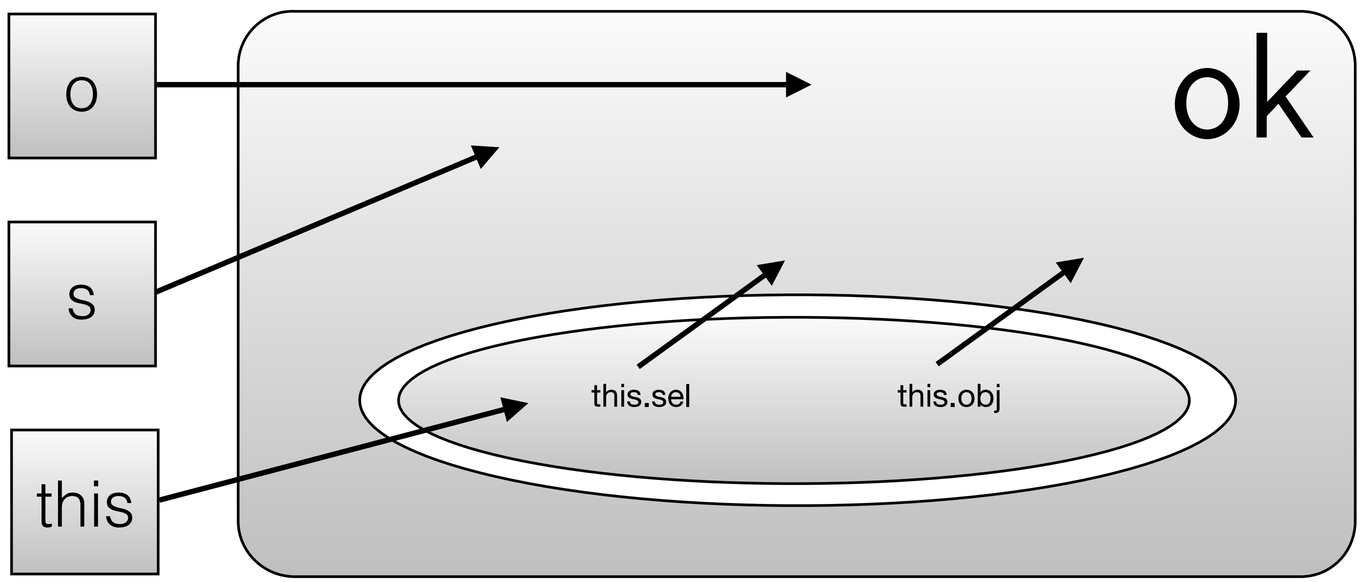
```
def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```



Values in **materialized** storage allowed to **differ from declared types**

Analysis performs **strong updates** on materialized cells

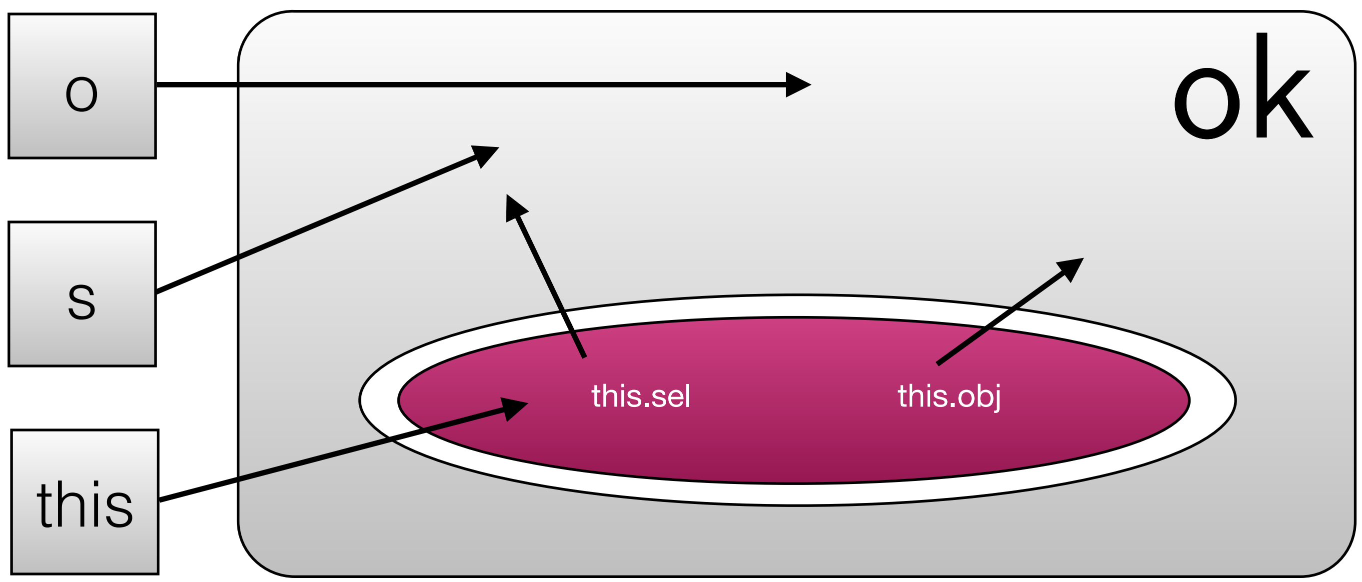
```
def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```



Values in **materialized** storage allowed to **differ from declared types**

Analysis performs **strong updates** on materialized cells

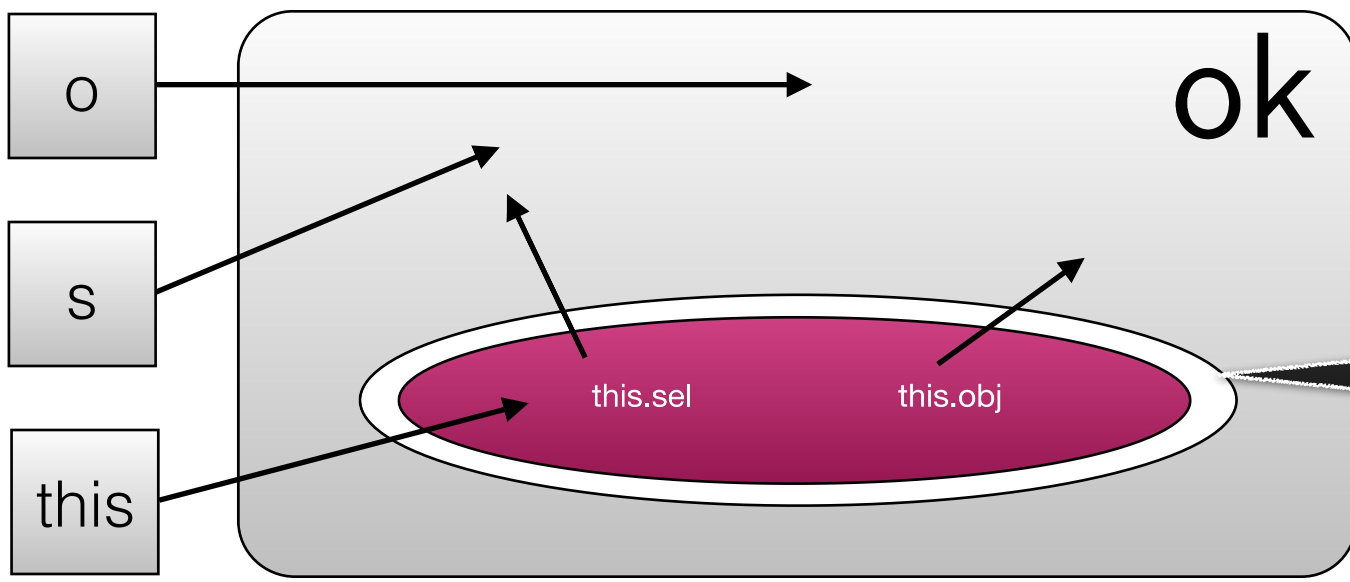
```
def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```



Values in **materialized** storage allowed to **differ from declared types**

Analysis performs **strong updates** on materialized cells

```
def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```

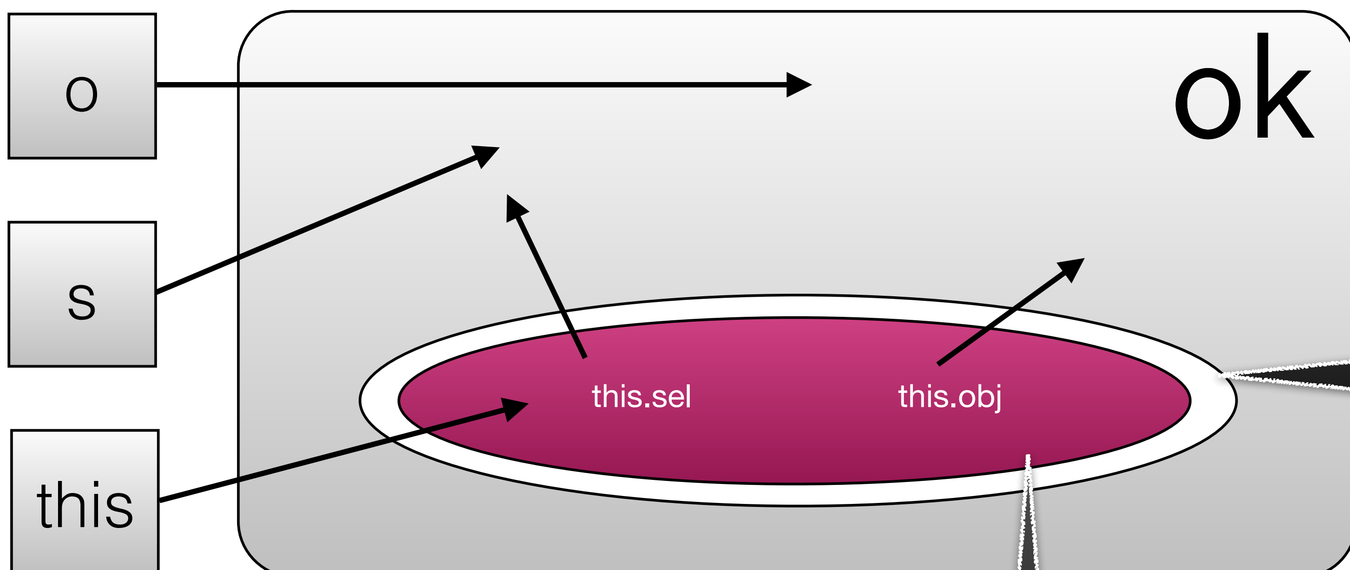


After first update, storage is **immediately type-inconsistent**

Values in **materialized** storage allowed to **differ from declared types**

Analysis performs **strong updates** on materialized cells

```
def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```



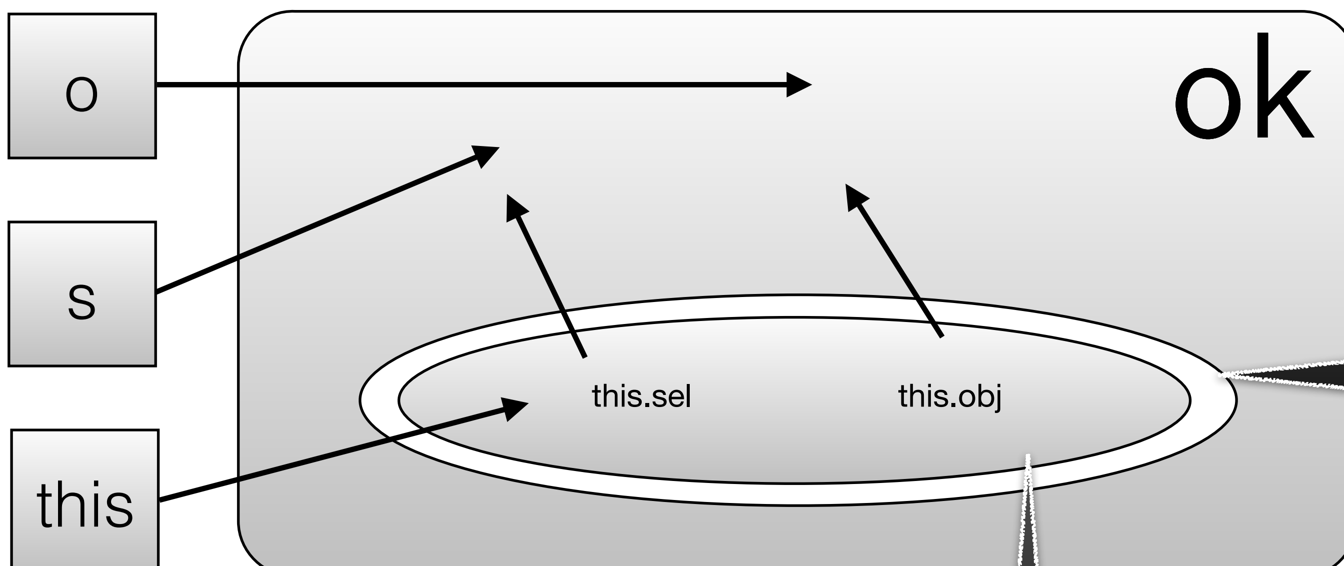
After first update, storage is **immediately type-inconsistent**

`this.obj` does not respond to `this.sel`

Values in **materialized** storage allowed to **differ from declared types**

Analysis performs **strong updates** on materialized cells

```
def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```



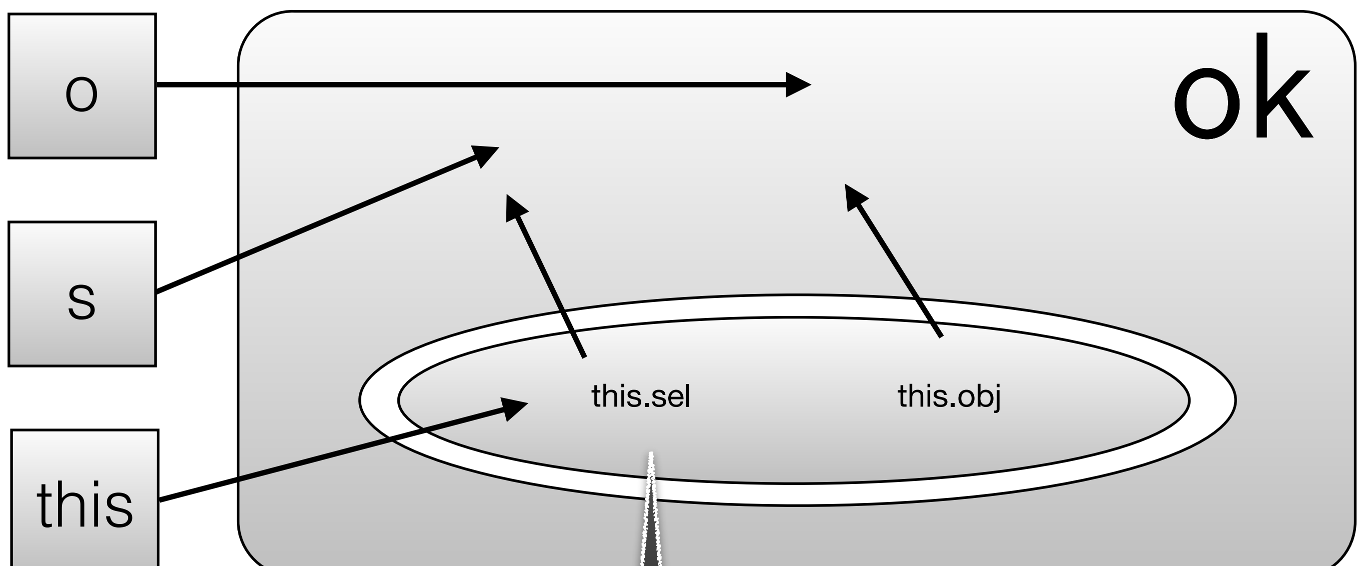
After first update, storage is **immediately type-inconsistent**

`this.obj` does not respond to `this.sel`

Values in **materialized** storage allowed to **differ from declared types**

Analysis performs **strong updates** on materialized cells

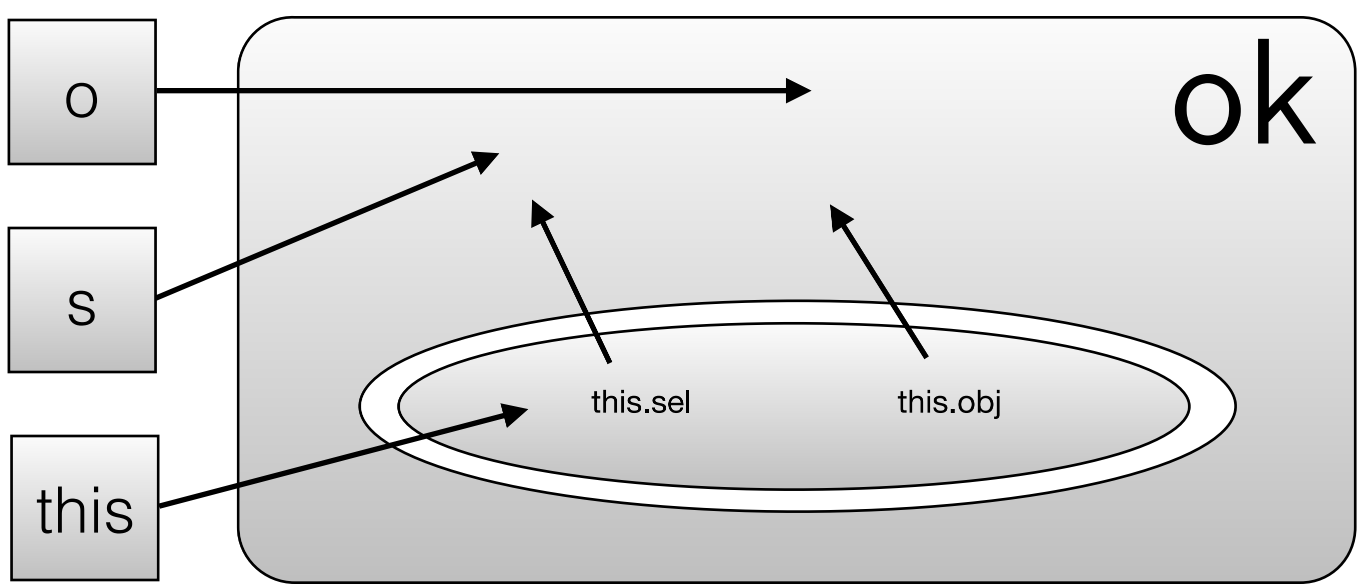
```
def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```



After second update, storage is **no longer immediately type-inconsistent** again

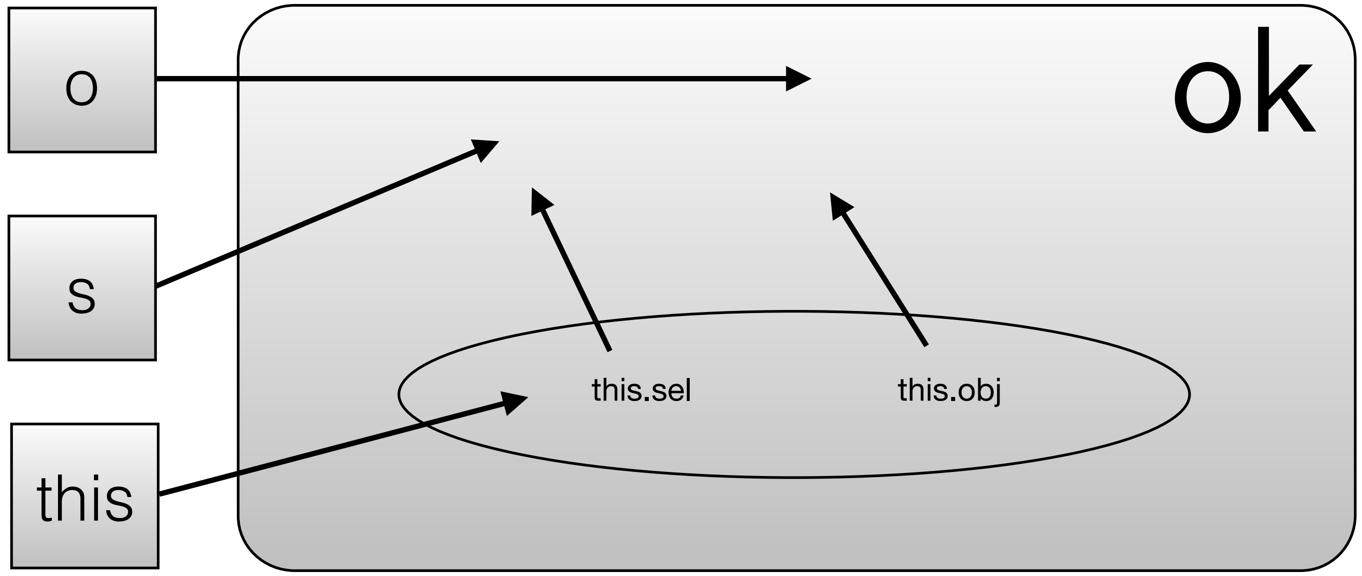
Can **summarize** not immediately type-inconsistent locations back into the almost type-consistent heap

```
def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```



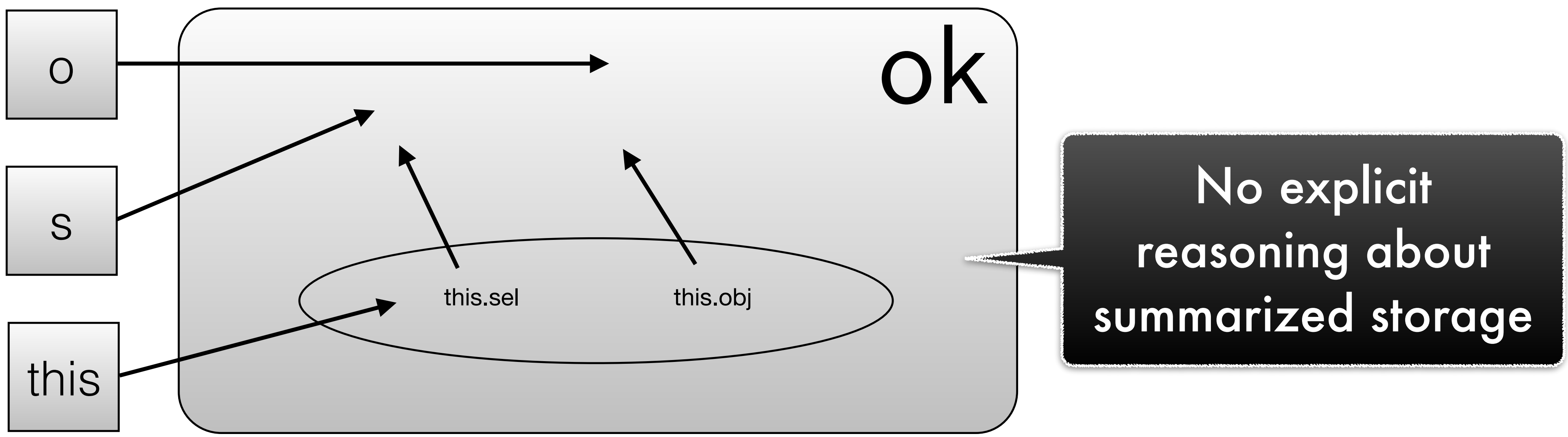
Can **summarize** not immediately type-inconsistent locations back into the almost type-consistent heap

```
def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```



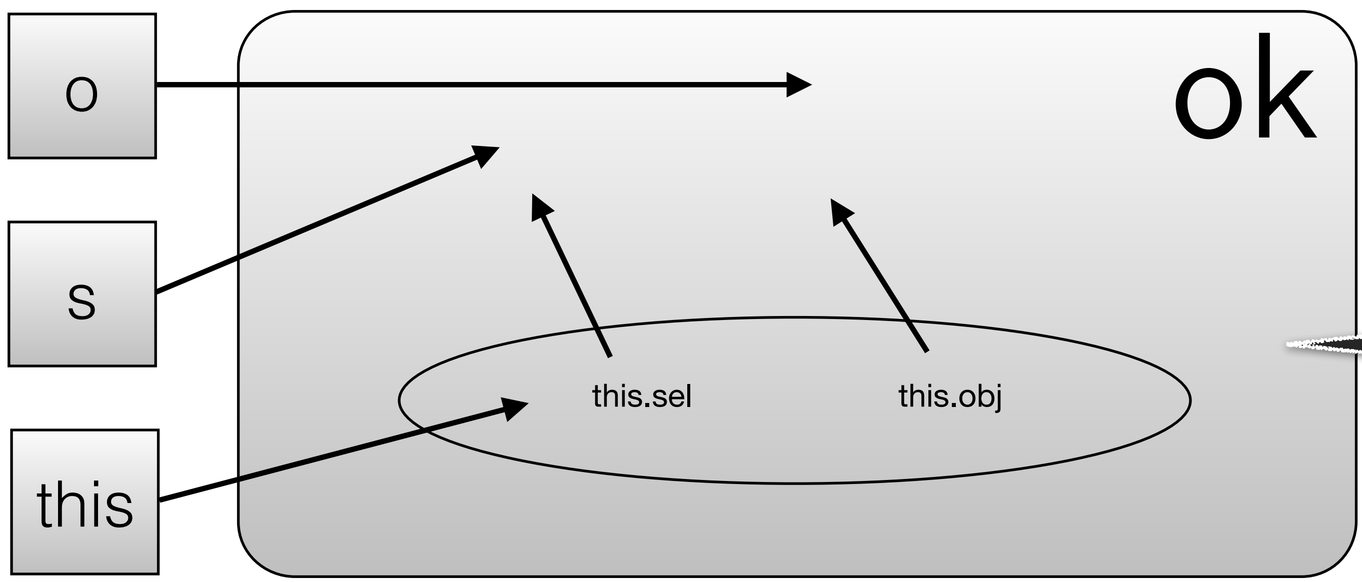
Can **summarize** not immediately type-inconsistent locations back into the almost type-consistent heap

```
def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```



Can **summarize** not immediately type-inconsistent locations back into the almost type-consistent heap

```
def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```



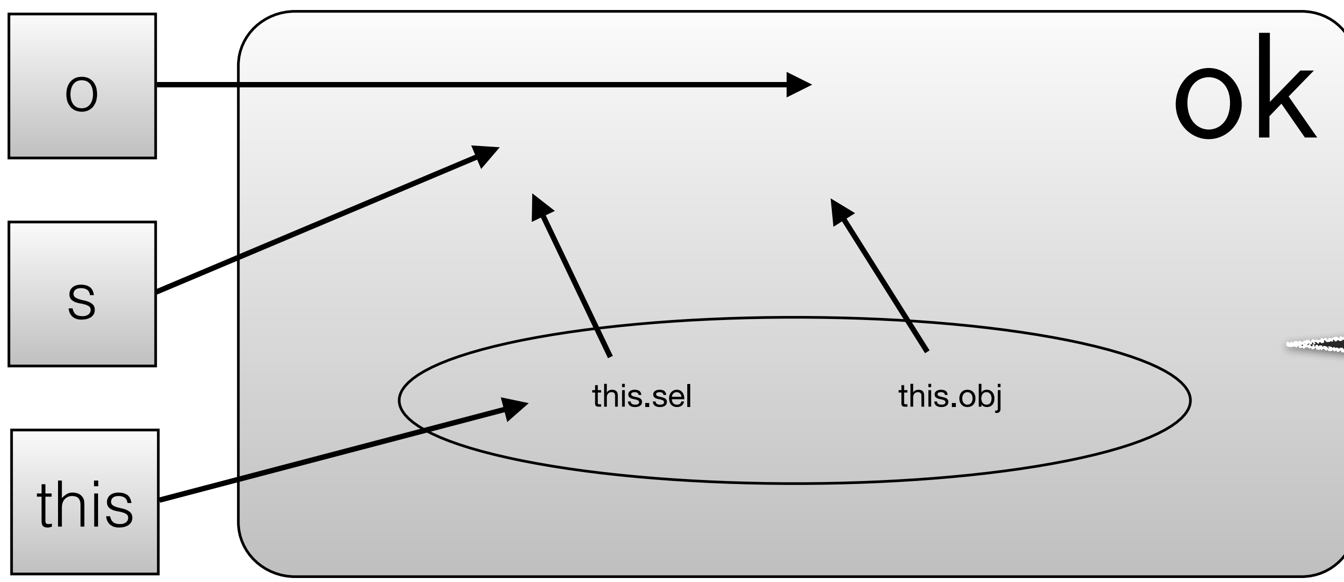
No explicit reasoning about summarized storage

If **entire heap** does **not** have immediately type-inconsistent locations then the heap is type-consistent

Can **summarize** not immediately type-inconsistent locations back into the almost type-consistent heap

Safe to **return to type checking**

```
def update(s: Str, o: Obj | r2 s)
  this.sel = s
  this.obj = o
```



No explicit reasoning about summarized storage

If **entire heap** does **not** have immediately type-inconsistent locations then the heap is type-consistent

Soundness

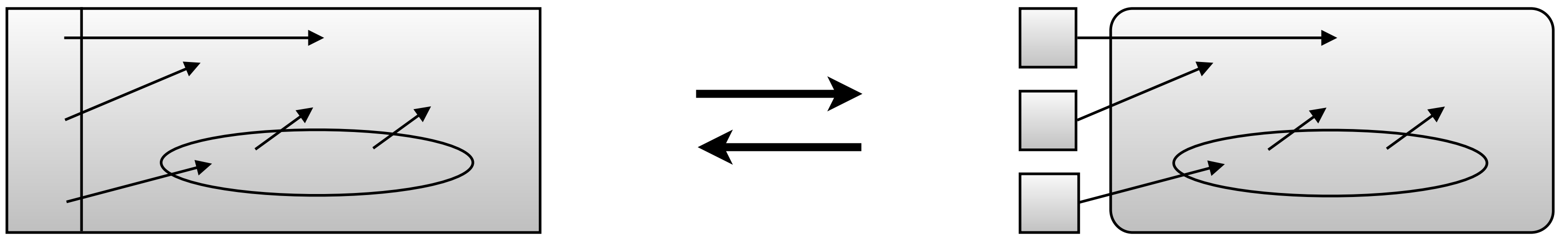
[Fissile Types, *POPL* 2014]

Soundness

[Fissile Types, *POPL* 2014]

Theorem (*Soundness of Handoff*).

The entire state is **type-consistent** *iff* all locations are **not immediately type-inconsistent**.

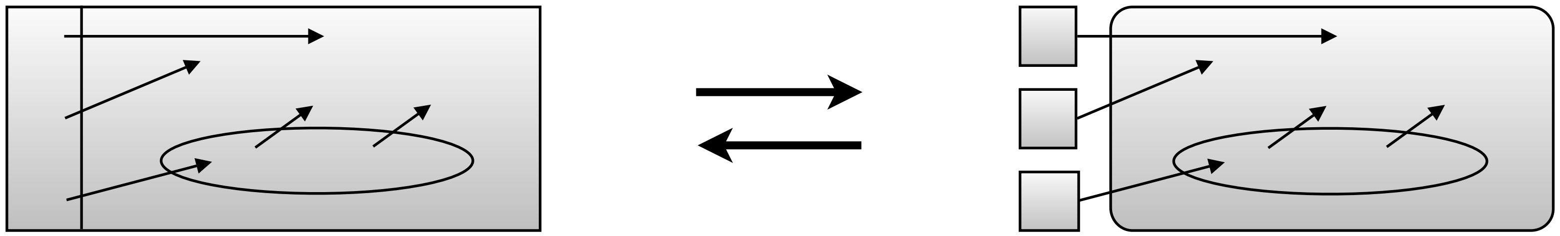


Soundness

[Fissile Types, *POPL* 2014]

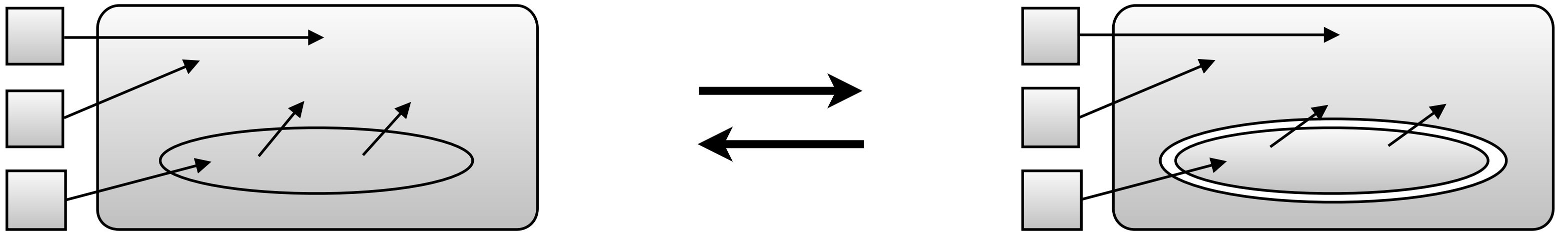
Theorem (*Soundness of Handoff*).

The entire state is **type-consistent** *iff* all locations are **not immediately type-inconsistent**.



Theorem (*Soundness of Materialization/Summarization*).

Locations that are **not immediately type-inconsistent** can be safely *materialized* and *summarized* into the almost type-consistent heap **ok**.



Case Study: Reflection in Objective-C

Prototype analysis implementation

- Plugin for clang static analyzer in C++

9 Objective-C benchmarks

- 6 libraries and 3 applications
- 1,000 to 176,000 lines of code

Manual type annotations

- 76 r2 annotations on system libraries
- 136 annotations on benchmark code

Case Study: Reflection in Objective-C

Prototype analysis implementation

- Plugin for clang static analyzer in C++

9 Objective-C benchmarks

- 6 libraries and 3 applications
- 1,000 to 176,000 lines of code

Including Skim, Adium, and OmniGraffle

Manual type annotations

- 76 r2 annotations on system libraries
- 136 annotations on benchmark code

Case Study: Reflection in Objective-C

Why Objective-C?
Statically typed plus reflective method call

Prototype analysis implementation

Plugin for clang static analyzer in C++

9 Objective-C benchmarks

6 libraries and 3 applications

1,000 to 176,000 lines of code

Including Skim, Adium, and OmniGraffle

Manual type annotations

76 r2 annotations on system libraries

136 annotations on benchmark code

Empirical Evaluation Questions

Precision: What is improvement over **flow-insensitive checking** alone?

Cost: What is the cost of analysis in **running time**?

Precision

	size		false alarms	
benchmark	**(loc)**	**reflective call sites**	**flow-insensitive**	**almost-everywhere**
OAuth	1248	7	7	2 (-71%)
SCRecorder	2716	12	2	0 (-100%)
ZipKit	3301	28	0	0 (-)
Sparkle	5289	40	4	1 (-75%)
ASIHTTPRequest	14620	68	50	10 (-80%)
OmniFrameworks	160769	192	82	74 (-10%)
Vienna	37327	186	59	38 (-36%)
Skim	60211	207	43	43 (-0%)
Adium	176629	587	87	70 (-20%)
combined	461080	1327	334	238 (-29%)

Precision

	size		false alarms	
benchmark	(loc)	reflective call sites	flow-insensitive	almost-everywhere
OAuth	1248	7	7	2 (-71%)
SCRecorder	2716	12	2	0 (-100%)
ZipKit	3301	28	0	0 (-)
Sparkle	5289	40	4	1 (-75%)
ASIHTTPRequest	14620	68	50	10 (-80%)
OmniFrameworks	160769	192	82	74 (-10%)
Vienna	37327	186	59	38 (-36%)
Skim	60211	207	43	43 (-0%)
Adium	176629	587	87	70 (-20%)
combined	461080	1327	334	238 (-29%)

Baseline: standard, **flow-insensitive** type analysis – no switching

Precision

benchmark	size (loc)	size reflective call sites	false alarms flow-insensitive	false alarms almost-everywhere
OAuth	1248	7	7	2 (-71%)
SCRecorder	2716	12	2	0 (-100%)
ZipKit	3301	28	0	0 (-)
Sparkle	5289	40	4	1 (-75%)
ASIHTTPRequest	14620	68	50	10 (-80%)
OmniFrameworks	160769	192	82	74 (-10%)
Vienna	37327	186	59	38 (-36%)
Skim	60211	207	43	43 (-0%)
Adium	176629	587	87	70 (-20%)
combined	461080	1327	334	238 (-29%)

Baseline: standard, **flow-insensitive** type analysis – no switching

Precision

	size		false alarms	
benchmark	**(loc)**	**reflective call sites**	**flow-insensitive**	**almost-everywhere**
OAUTH	1248	7	7	2 (-71%)
SCRECORDER	2716	12	2	0 (-100%)
ZIPKIT	3301	28	0	0 (-)
SPARKLE	5289	40	4	1 (-75%)
ASIHTTPREQUEST	14620	68	50	10 (-80%)
OMNIFRAMEWORKS	160769	192	82	74 (-10%)
VIENNA	37327	186	59	38 (-36%)
SKIM	60211	207	43	43 (-0%)
ADIUM	176629	587	87	70 (-20%)
combined	461080	1327	334	238 (-29%)

Baseline: standard, **flow-insensitive** type analysis – no sw...

Almost everywhere techniques show 29% improvement in false alarms

Precision

benchmark	size		false alarms	
	(loc)	reflective call sites	flow-insensitive	almost-everywhere
OAUTH	1248	7	7	2 (-71%)
	2716	12	2	0 (-100%)
	3301	28	0	0 (-)
	5289	40	4	1 (-75%)
	14620	68	50	10 (-80%)
	160769	192	82	74 (-10%)
VIENNA	37327	186	59	38 (-36%)
SKIM	60211	207	43	43 (-0%)
ADIUM	176629	587	87	70 (-20%)
combined	461080	1327	334	238 (-29%)

Also found a **real reflection bug** in Vienna, which we reported and which was fixed

Baseline: standard, **flow-insensitive** type analysis – no sw

Almost everywhere techniques show 29% improvement in false alarms

Cost: Analysis **Time**

	size	analysis time	
benchmark	**(loc)**	**Time**	**Rate (kloc/s)**
OAuth	1248	0.24s	5.3
SCRecorder	2716	0.28s	10.8
ZipKit	3301	0.10s	33
Sparkle	5289	0.67s	7.9
ASIHTTPRequest	14620	0.50s	27.2
OmniFrameworks	160769	4.25s	37.8
Vienna	37327	2.79s	13.4
Skim	60211	2.49s	24.1
Adium	176629	8.79s	20.1
combined	461080	20.09s	23

Cost: Analysis **Time**

	size	analysis time	
benchmark	**(loc)**	**Time**	**Rate (kloc/s)**
OAUTH	1248	0.24s	5.3
SCRECORDER	[illegible]	0.28s	10.8
ZIPKIT	3301	0.10s	33
[illegible]	5289	0.67s	7.9
[illegible]	14620	0.50s	27.2
[illegible]	160769	4.25s	37.8
[illegible]	37327	2.79s	13.4
SKIM	60211	2.49s	24.1
ADIUM	176629	8.79s	20.1
combined	461080	20.09s	23

Includes analysis time but **not parsing, base type checking**

Cost: Analysis **Time**

	size	analysis time	
	(loc)	Time	Rate (kloc/s)
[illegible]	1248	0.24s	5.3
[illegible]	[illegible]	0.28s	10.8
[illegible]	3301	0.10s	33
[illegible]	5289	0.67s	7.9
[illegible]	14620	0.50s	27.2
[illegible]	160769	4.25s	37.8
[illegible]	37327	2.79s	13.4
[illegible]	60211	2.49s	24.1
ADIUM	176629	8.79s	20.1
combined	461080	20.09s	23

Does **not** include **system headers**

Includes analysis time but **not parsing, base type checking**

Cost: Analysis **Time**

	size	analysis time	
benchmark	**(loc)**	**Time**	**Rate (kloc/s)**
OAuth	1248	0.24s	5.3
SCRecorder	2716	0.28s	10.8
ZipKit	3301	0.10s	33
Sparkle	5289	0.67s	7.9
ASIHTTPRequest	14620	0.50s	27.2
OmniFrameworks	160769	4.25s	37.8
Vienna	37327	2.79s	13.4
Skim	60211	2.49s	24.1
Adium	176629	8.79s	20.1
combined	461080	20.09s	23

Fast: 5 to 38 kloc/s with most time spent **analyzing system headers**

Cost: Analysis **Time**

benchmark	size (loc)	analysis time: Time	analysis time: Rate (kloc/s)
OAuth	1248	0.24s	5.3
SCRecorder	2716	0.28s	10.8
ZipKit	3301	0.10s	33
Sparkle	5289	0.67s	7.9
ASIHTTPRequest	14620	0.50s	27.2
OmniFrameworks	160769	4.25s	37.8
Vienna	37327	2.79s	13.4
Skim	60211	2.49s	24.1
Adium	176629	8.79s	20.1
combined	461080	20.09s	23

Fast: 5 to 38 kloc/s with most time spent **analyzing system headers**

Interactive speeds

Cost: Analysis **Time**

	size	analysis time	
benchmark	**(loc)**	**Time**	**Rate (kloc/s)**
OAuth	1248	0.24s	5.3
SCRecorder	2716	0.28s	10.8
ZipKit	3301	0.10s	33
Sparkle	5289	0.67s	7.9
ASIHTTPRequest	14620	0.50s	27.2
OmniFrameworks	160769	4.25s	37.8
Vienna	37327	2.79s	13.4
Skim	60211	2.49s	24.1
Adium	176629	8.79s	20.1
combined	461080	20.09s	23

Fast: 5 to 38 kloc/s with most time spent **analyzing system headers**

Higher rate for projects with **larger translation units**

Cost: Analysis **Time**

benchmark	size (loc)	analysis time: Time	analysis time: Rate (kloc/s)
OAuth	1248	0.24s	5.3
SCRecorder	2716	0.28s	10.8
ZipKit	3301	0.10s	33
Sparkle	5289	0.67s	7.9
ASIHTTPRequest	14620	0.50s	27.2
OmniFrameworks	160769	4.25s	37.8
Vienna	37327	2.79s	13.4
Skim	60211	2.49s	24.1
Adium	176629	8.79s	20.1
combined	461080	20.09s	23

Fast: 5 to 38 kloc/s with most time spent **analyzing** sy

H **units**

Maintains key benefit of flow-insensitive analyses: **speed**

Key Ideas

ok Tolerating temporary violations with **almost type-consistent heaps**

Coughlin and Chang. *POPL* 2014.

Type-intertwined framing with **gated separation**

Under preparation.

Key Ideas

ok Tolerating temporary violations with **almost type-consistent heaps**

Coughlin and Chang. *POPL* 2014.

Type-intertwined framing with **gated separation**

Under preparation.

Key Ideas

ok Tolerating temporary violations with **almost type-consistent heaps**

Coughlin and Chang. *POPL* 2014.

Type-intertwined framing with **gated separation**

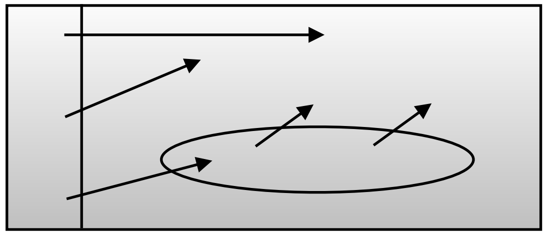
Under preparation.

Recall: Soundness

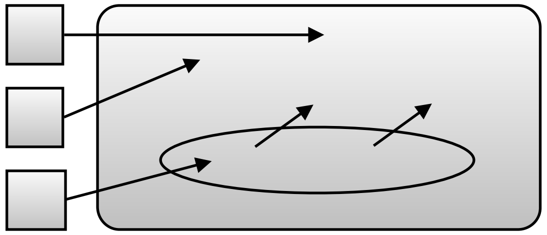
[Fissile Types, *POPL* 2014]

Theorem (*Soundness of Handoff*).

The entire state is **type-consistent** *iff* all locations are **not immediately type-inconsistent**.

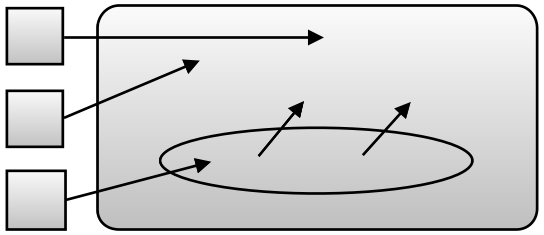


⇄

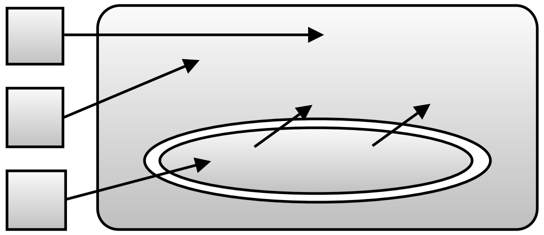


Theorem (*Soundness of Materialization/Summarization*).

Locations that are **not immediately type-inconsistent** can be safely *materialized* and *summarized* into the almost type-consistent heap **ok**.



⇄

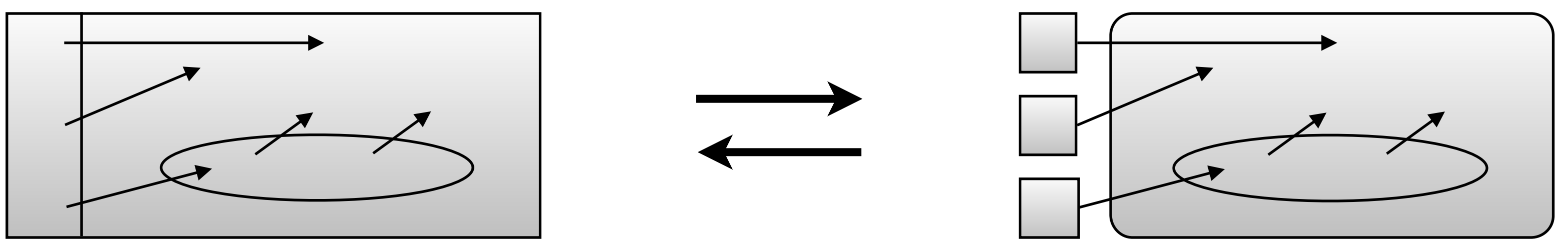


Recall: Soundness

[Fissile Types, *POPL* 2014]

Theorem (*Soundness of Handoff*).

The entire state is **type-consistent** *iff* all locations are **not immediately type-inconsistent**.



Theorem (*Soundness of Materialization/Summarization*).

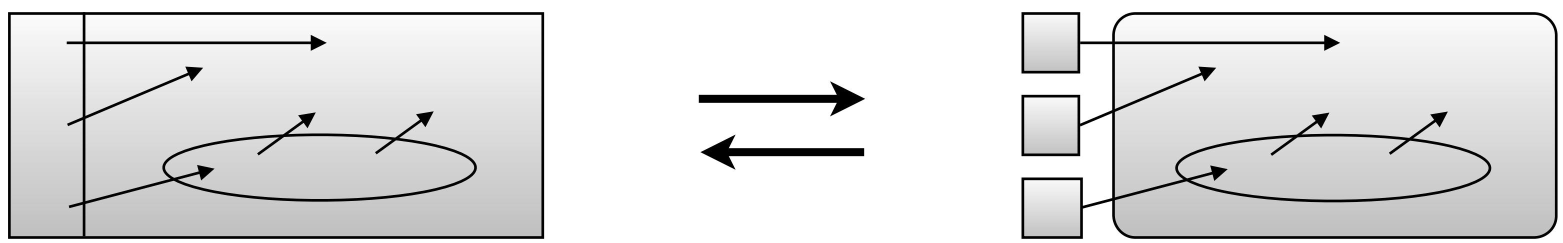
Locations that are **not immediately type-inconsistent** can be safely *materialized* and *summarized* into the almost type-consistent heap **ok**.

Recall: Soundness

[Fissile Types, *POPL* 2014]

Theorem (*Soundness of Handoff*).

The entire state is **type-consistent** *iff* all locations are **not immediately type-inconsistent**.



Theorem (*Soundness of Materialization/Summarization*).

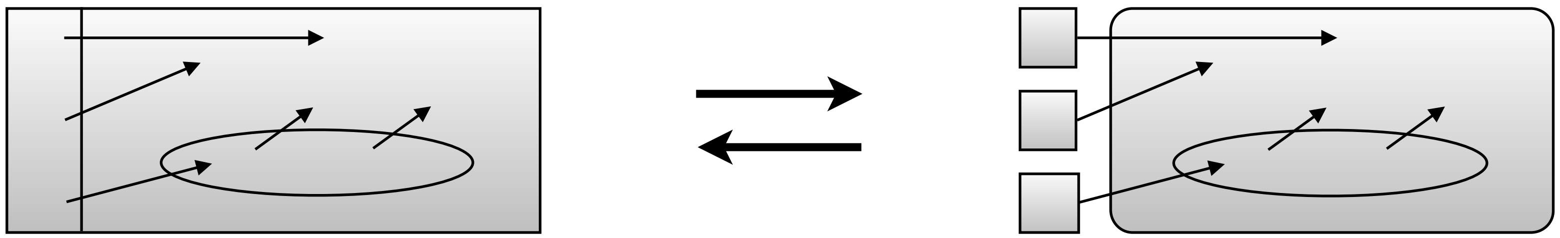
Locations that are **not immediately type-inconsistent** can be safely *materialized* and *summarized* into the almost type-consistent heap **ok**.

Recall: Soundness

[Fissile Types, *POPL* 2014]

Theorem (*Soundness of Handoff*).

The entire state is **type-consistent** *iff* all locations are **not immediately type-inconsistent**.



Theor

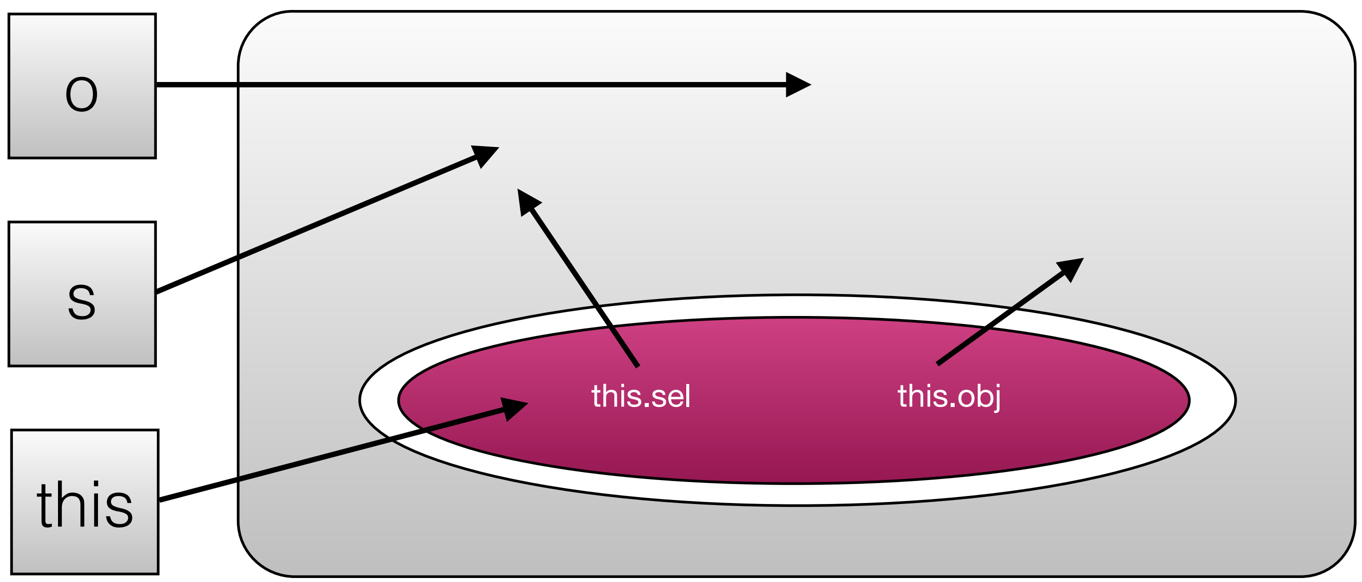
Locatio

materi

No way to **frame** and then handoff to types

Type-intertwined frame rule with standard separating conjunction is **unsound**

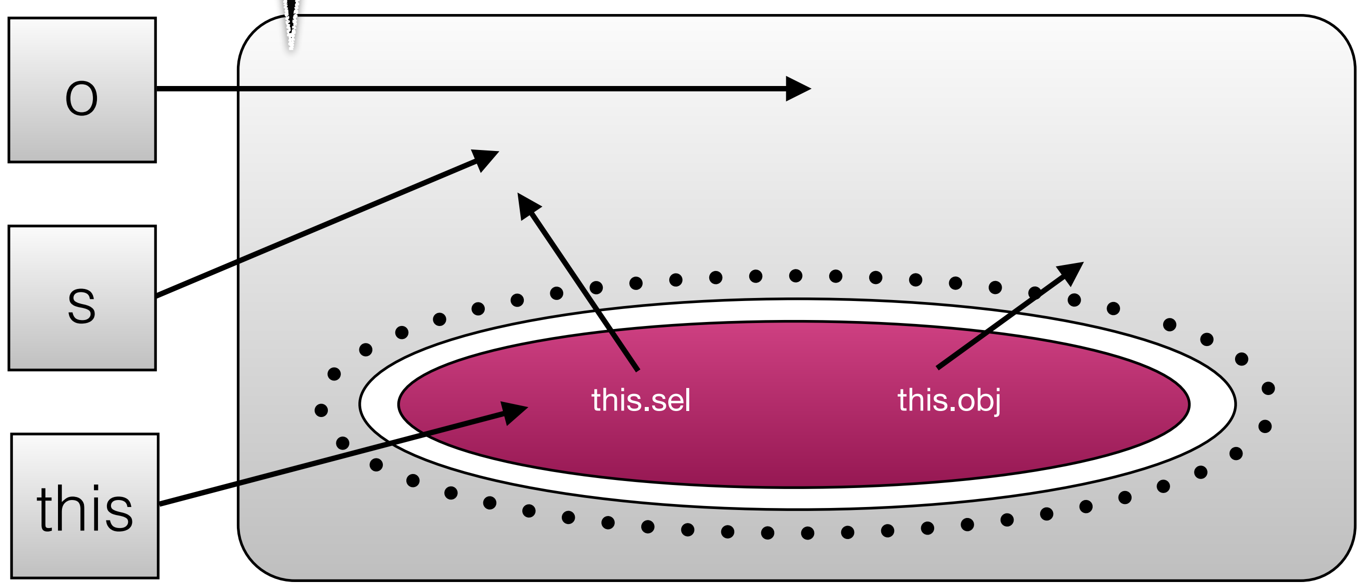
```
def badUpdate(s: Str, o: Obj | r2 s)
  this.sel = s
  this.call()
  this.obj = o
```



Type-intertwined frame rule with standard separating conjunction is **unsound**

Immediately type-inconsistent portion of heap is **disjoint** from almost type-consistent summary

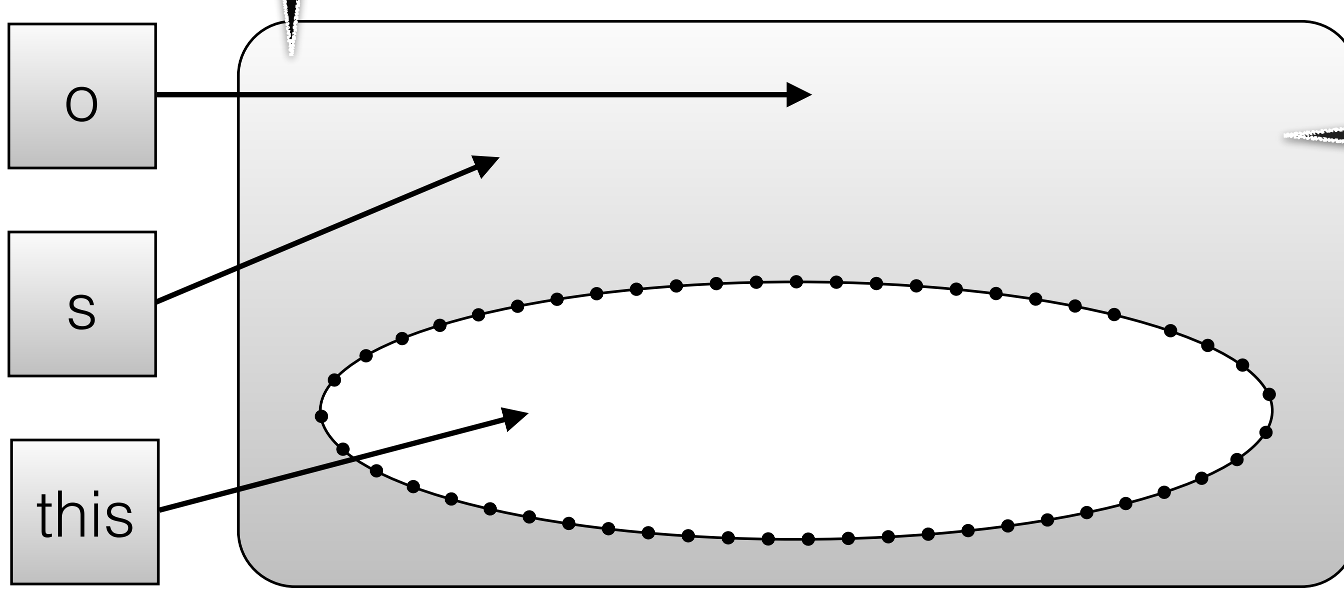
```
def badUpdate(s: Str, o: Obj | r2 s)
  this.sel = s
  this.call()
  this.obj = o
```



Type-intertwined frame rule with standard separating conjunction is **unsound**

Immediately type-inconsistent portion of heap is **disjoint** from almost type-consistent summary

```
def badUpdate(s: Str, o: Obj | r2 s)
  this.sel = s
  this.call()
  this.obj = o
```

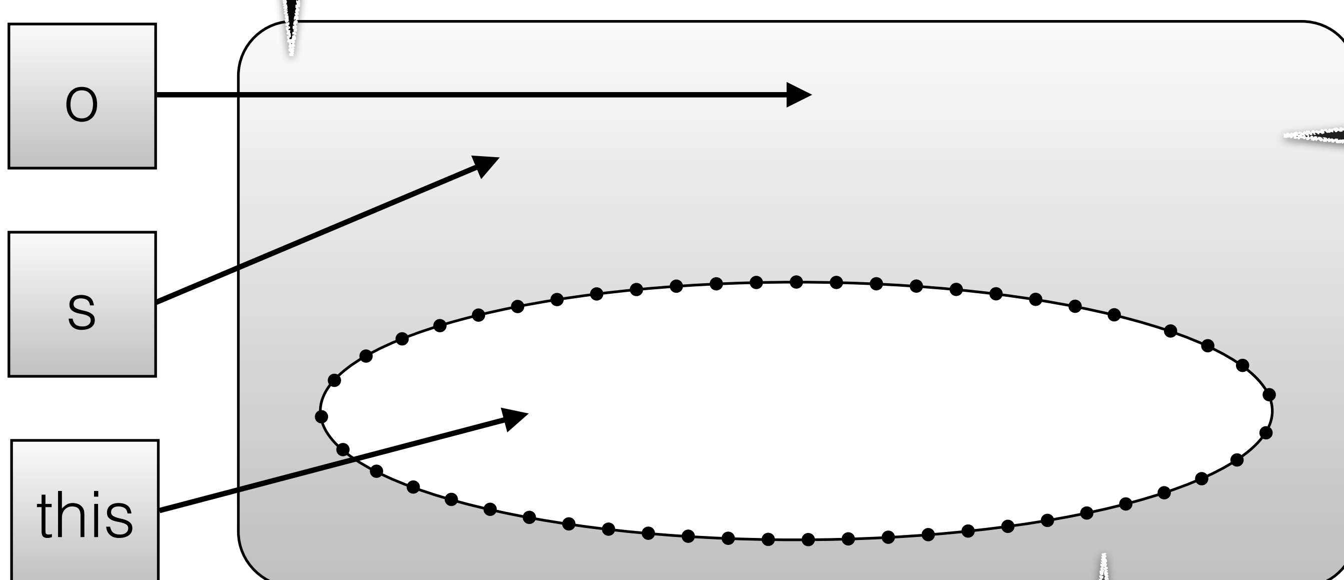


After framing out, entire heap is almost type-consistent

Type-intertwined frame rule with standard separating conjunction is **unsound**

```
def badUpdate(s: Str, o: Obj | r2 s)
  this.sel = s
  this.call()
  this.obj = o
```

Immediately type-inconsistent portion of heap is **disjoint** from almost type-consistent summary



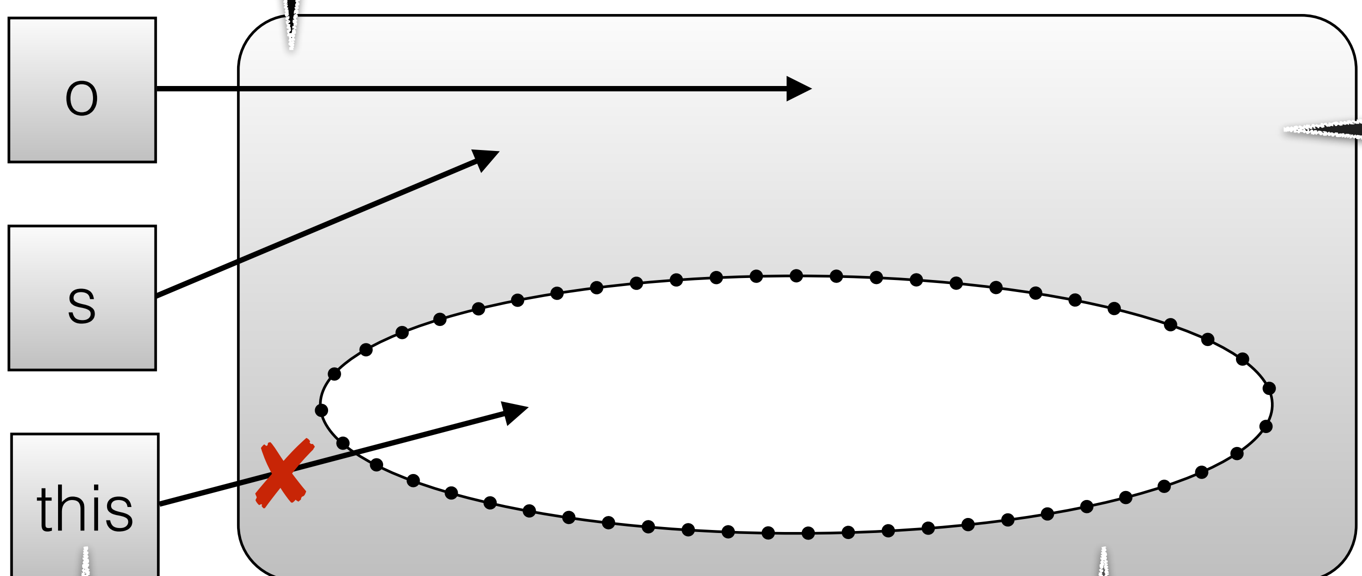
After framing out, entire heap is almost type-consistent

Analysis might **unsoundly** switch to back to type checking

Type-intertwined frame rule with standard separating conjunction is **unsound**

Immediately type-inconsistent portion of heap is **disjoint** from almost type-consistent summary

```
def badUpdate(s: Str, o: Obj | r2 s)
  this.sel = s
  this.call()
  this.obj = o
```



After framing out, entire heap is almost type-consistent

Points to type-inconsistent memory

Analysis might **unsoundly** switch to back to type checking

So what if there's no type-intertwined framing?

So what if there's no type-intertwined framing?

```
class Callback
 var sel: Str
 var obj: Obj | r2 sel

 def call()
  this.obj.[this.sel]()
```

So what if there's no type-intertwined framing?

```
class Callback
 var sel: Str
 var obj: Obj | r2 sel

 def call()
  this.obj.[this.sel]()

var o = … object with a method m …
var cb = new Callback("m",o)
cb.call()
```

Callback in JavaScript

JavaScript

```
var Callback = Class({
  __init__: function(s,o){…},
  call: function(){
    return this.obj[this.sel].apply(this.obj)
  },
})

var o = … object with a method m …
var cb = New(Callback,“m”,o)
cb.call()
```

Callback in JavaScript

A class "meta-feature" **library**

JavaScript

```
var Callback = Class({
  __init__: function(s,o){…},
  call: function(){
    return this.obj[this.sel].apply(this.obj)
  },
})
```

```
var o = … object with a method m …
var cb = New(Callback,"m",o)
cb.call()
```

Callback in JavaScript

A class "meta-feature" library

```
var Callback = Class({
  __init__: function(s,o){…},
  call: function(){
    return this.obj[this.sel].apply(this.obj)
  },
})

var o = … object with a method m …
var cb = New(Callback,"m",o)
cb.call()
```

Callback in JavaScript

A class "meta-feature" **library**

```
var Callback = Class({
  __init__: function(s,o){…},
  call: function(){
    return this.obj[this.sel].apply(this.obj)
  },
})
```

JavaScript

```
var o = … objec
var cb = New
cb.call()
```

Want to **type check** this call "like before"
– i.e., using the same type system

Callback in JavaScript

A class “meta-feature” **library**

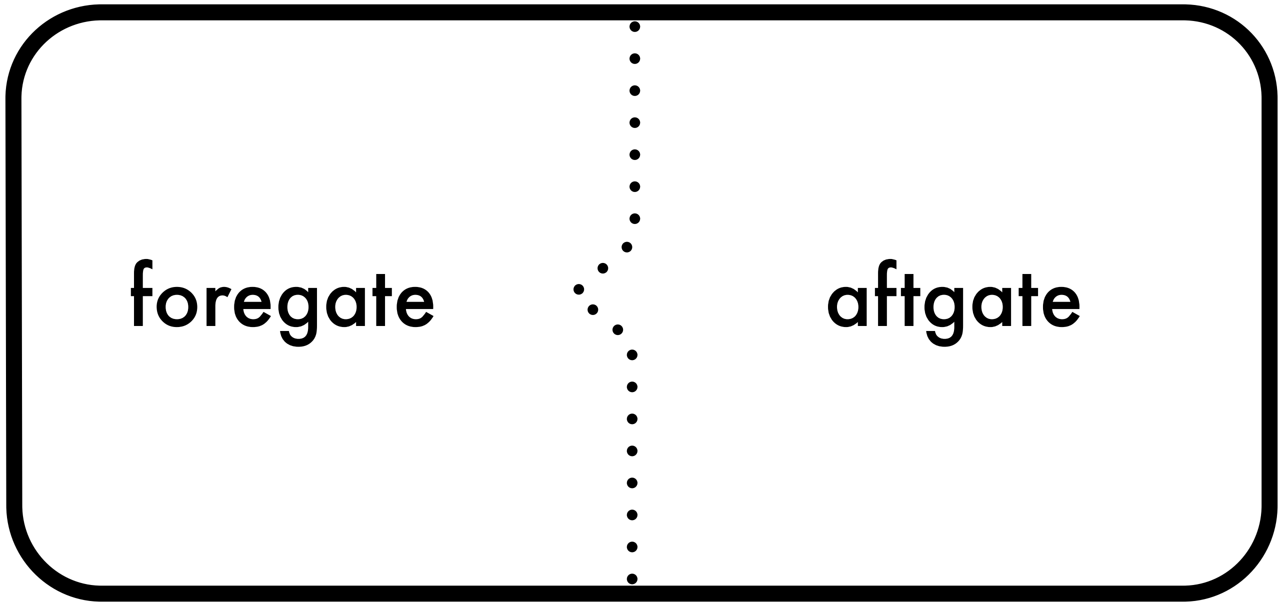
JavaScript

```
var Callback = Class({
  __init__: function(s,o){…},
  call: function(){
    return this.obj[this.sel].apply(this.obj)
  },
})

var o = … objec
var cb = New
cb.call()
```

Want to **type check** this call “like before”
– i.e., using the same type system

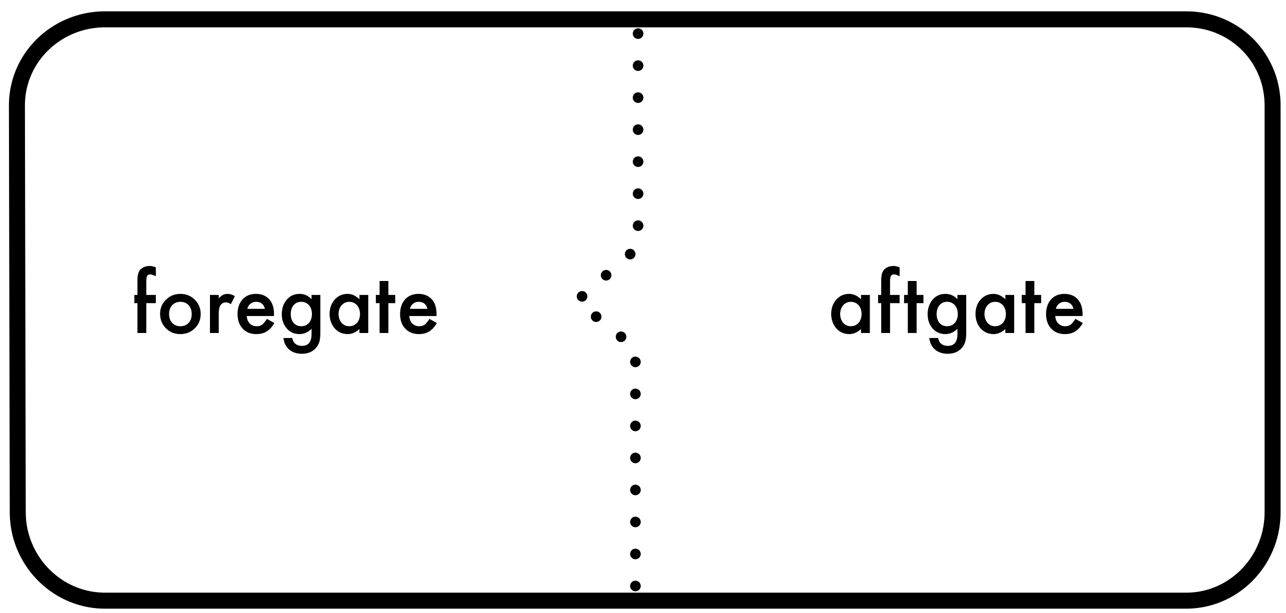
gated separation

Gated separation expresses a **dis-pointing** relation between **foregate** and **aftgate**



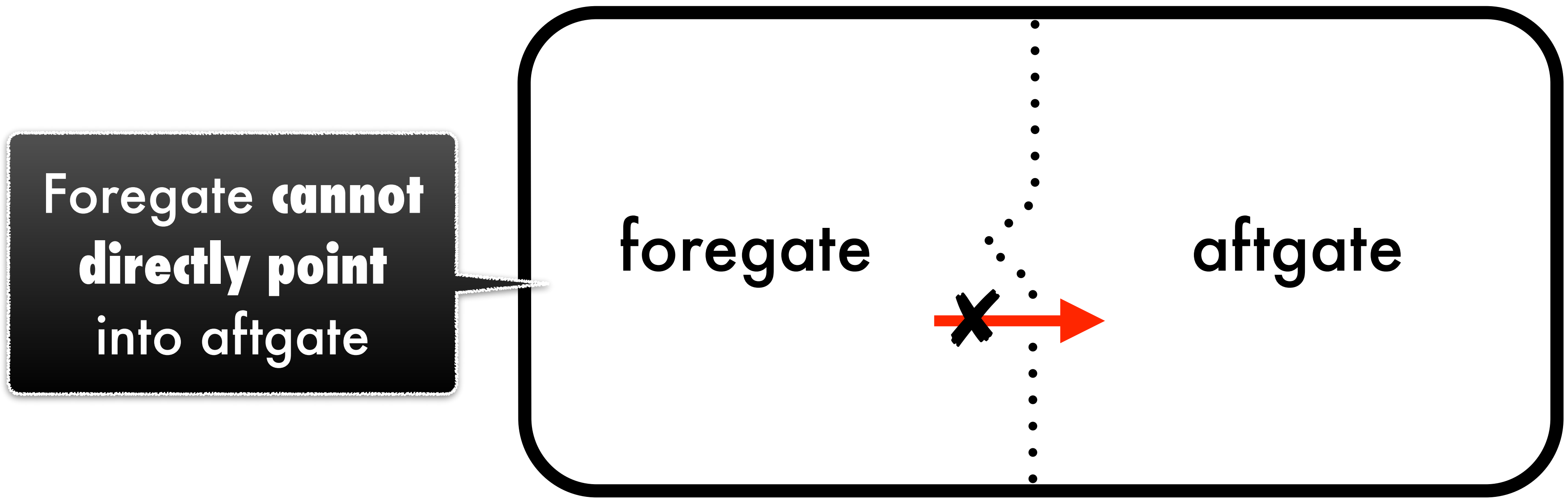
Gated separation expresses a **dis-pointing** relation between **foregate** and **aftgate**

$M_{\mathrm{fore}} \triangleleft\!\!\!\!\ast \; M_{\mathrm{aft}}$



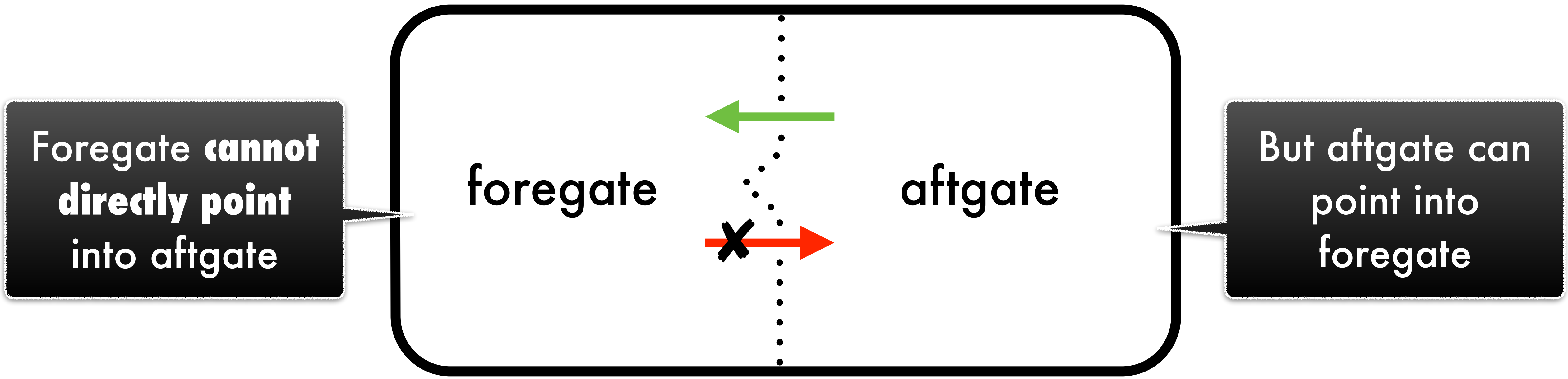
Gated separation expresses a **dis-pointing** relation between **foregate** and **aftgate**

$M_{fore} \lhd\!\!\!\ast\; M_{aft}$



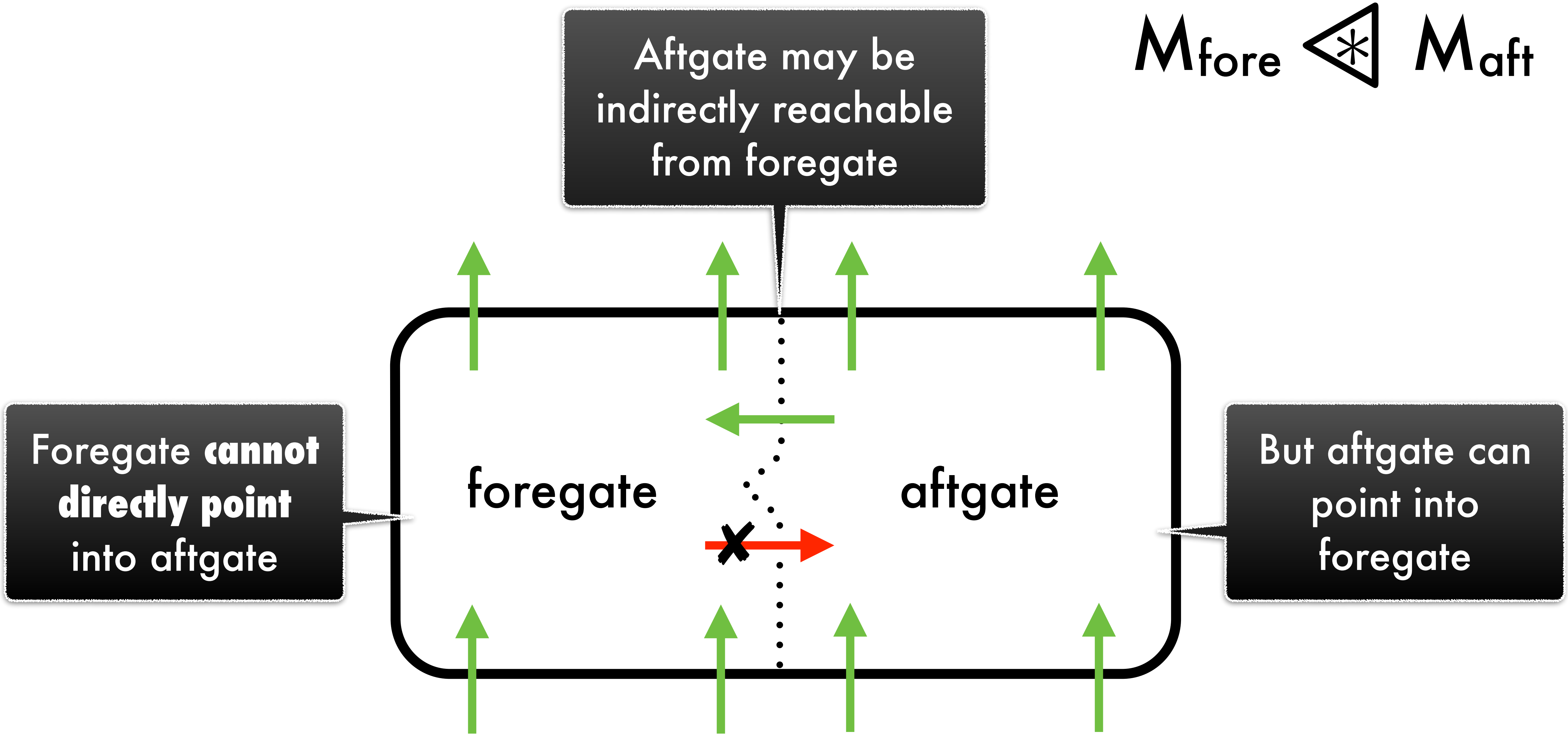
Gated separation expresses a **dis-pointing** relation between **foregate** and **aftgate**

$M_{fore} \lhd\!\!\!\ast\; M_{aft}$



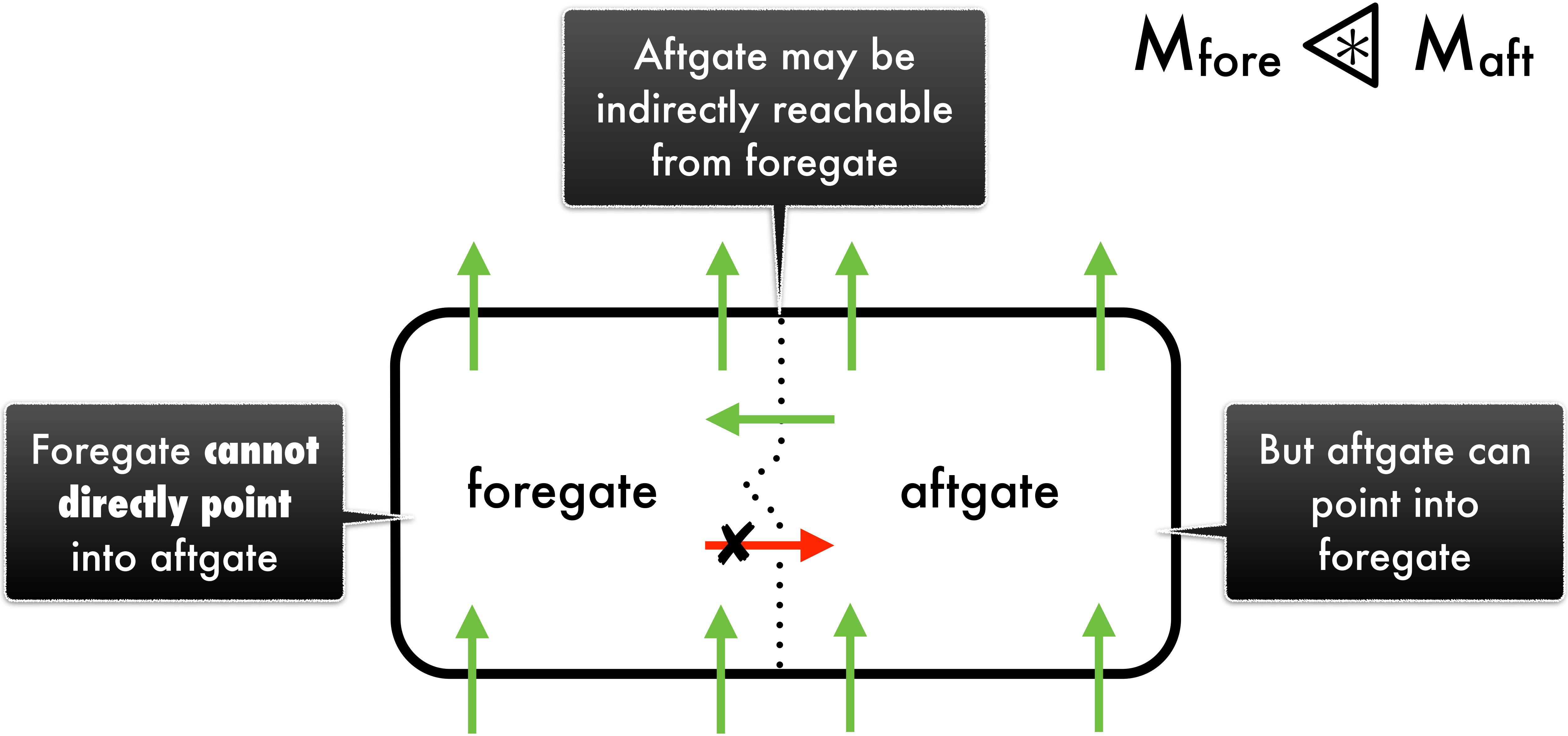
Gated separation expresses a **dis-pointing** relation between **foregate** and **aftgate**

$$M_{fore} \mathrel{\triangleleft\!\!\!*} M_{aft}$$



Gated separation expresses a **dis-pointing** relation between **foregate** and **aftgate**

$M_{fore} \lhd\!\!\!\ast\; M_{aft}$

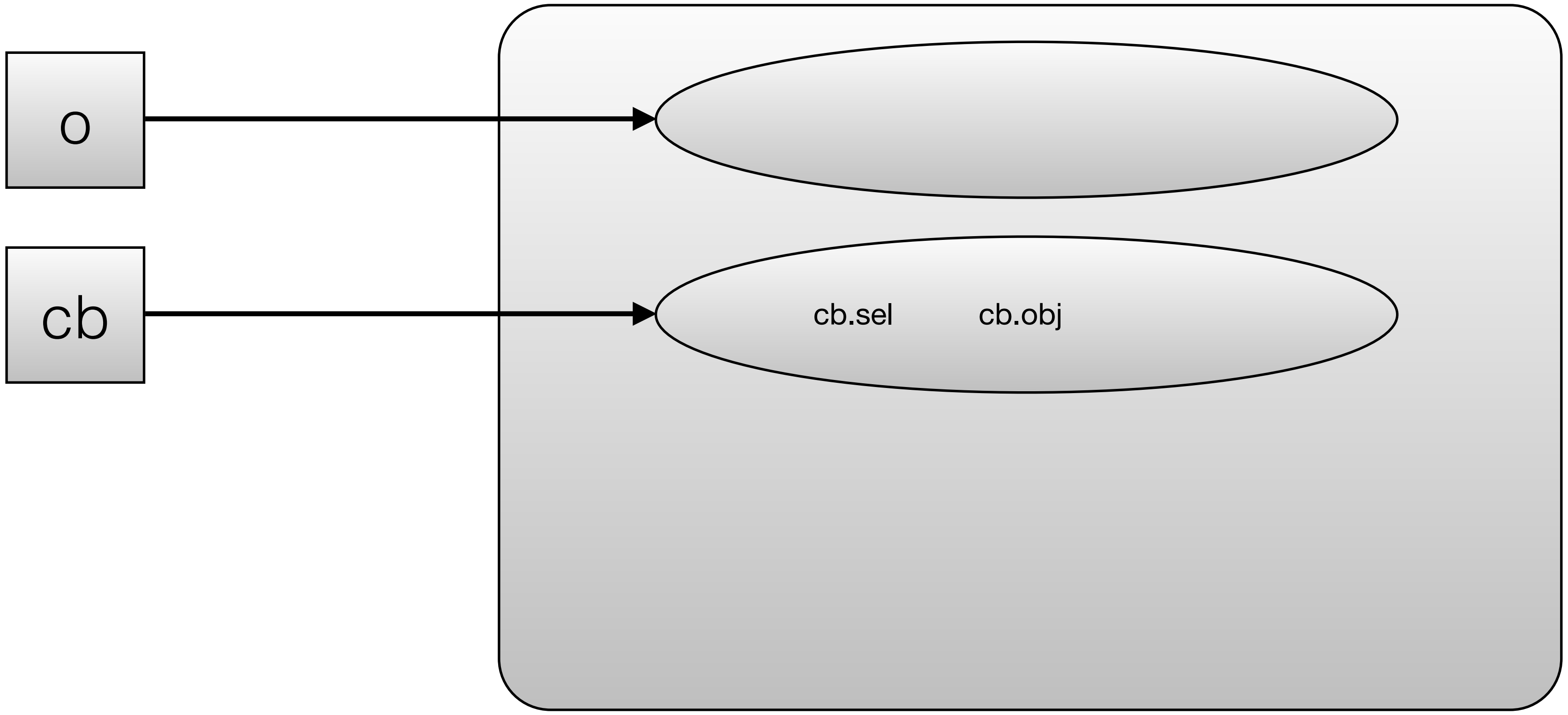


Gated separation is a **non-commutative strengthening** of standard separating conjunction restricting the **contents** of the **foregate**

Callback in JavaScript

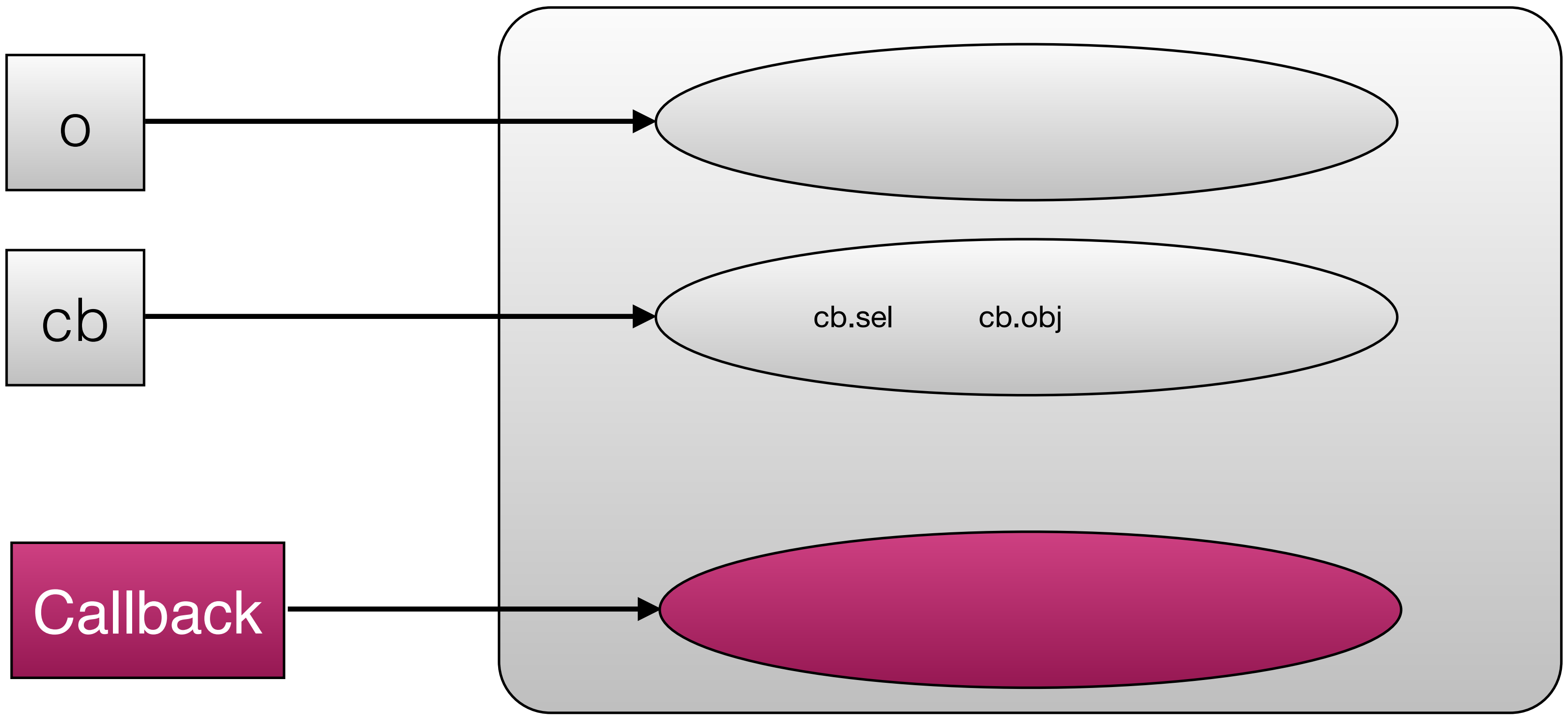
```
var Callback = Class({
  __init__: function(s,o){…},
  call: function(){
    return this.obj[this.sel].apply(this.obj)
  },
})

var o = … object with a method m …
var cb = New(Callback,"m",o)
cb.call()
```

Callback in JavaScript



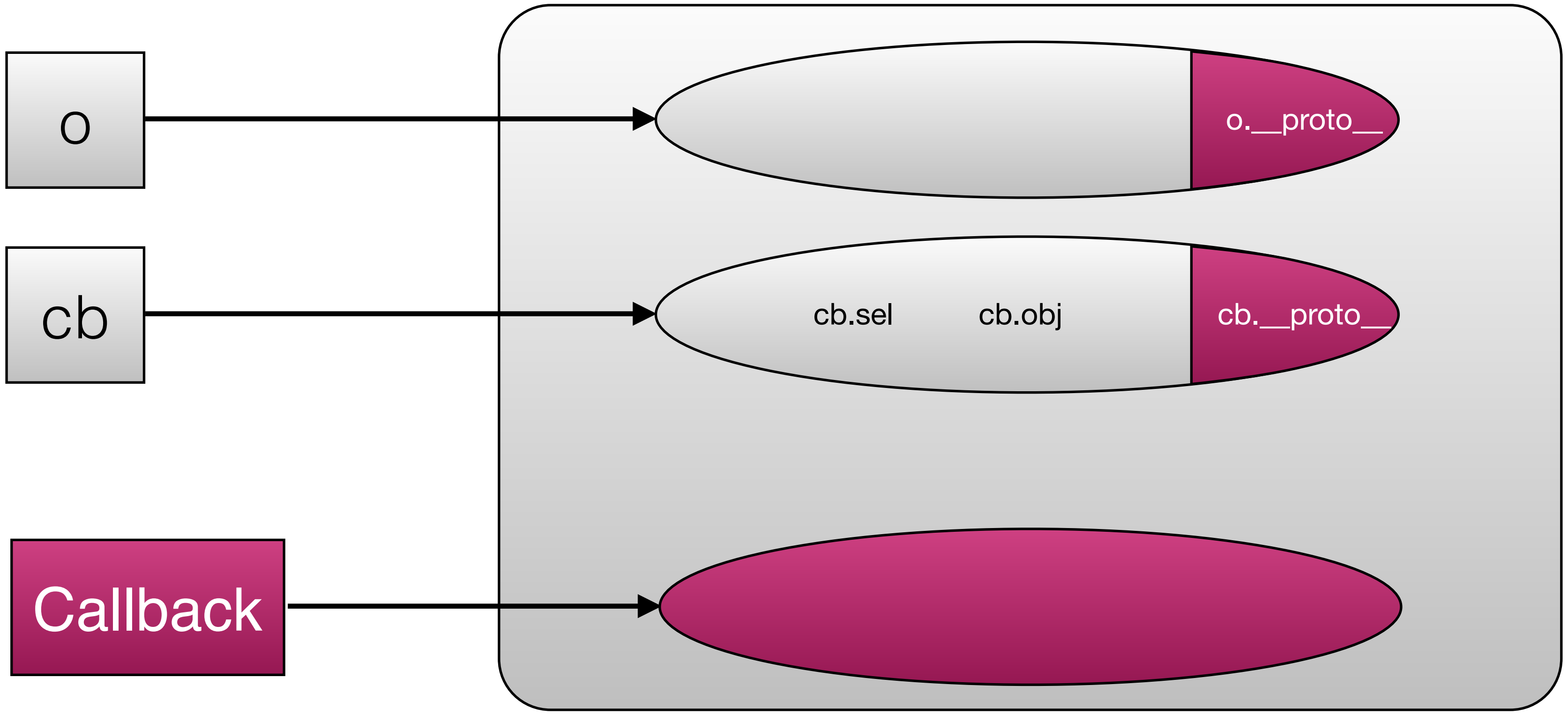

```
var o = … object with a method m …
var cb = New(Callback,"m",o)
cb.call()
```

Callback in JavaScript



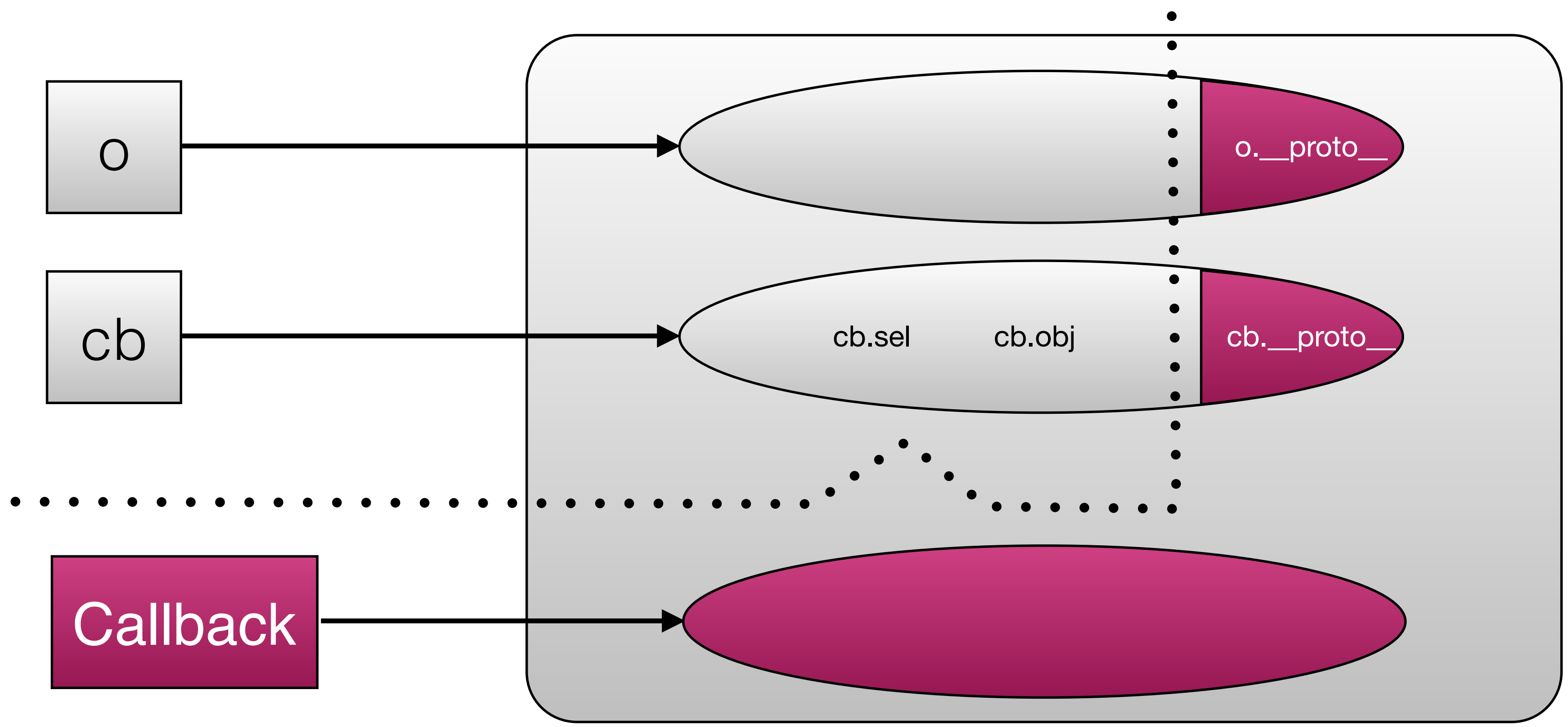

```
var o = ... object with a method m ...
var cb = New(Callback,"m",o)
cb.call()
```

Callback in JavaScript




```
var o = ... object with a method m ...
var cb = New(Callback,"m",o)
cb.call()
```

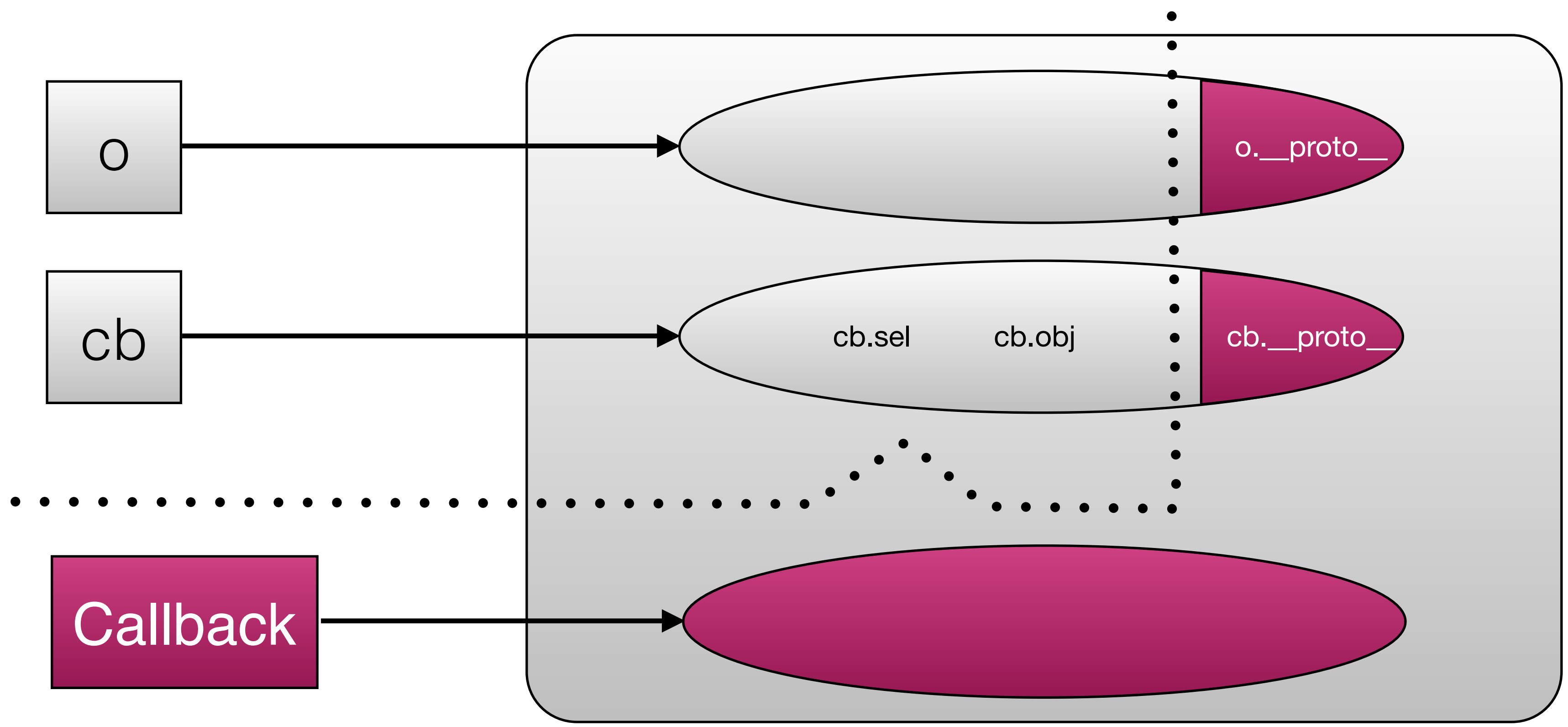
Callback in JavaScript



`var o =` *... object with a method m ...*

```
var cb = New(Callback,“m”,o)
cb.call()
```

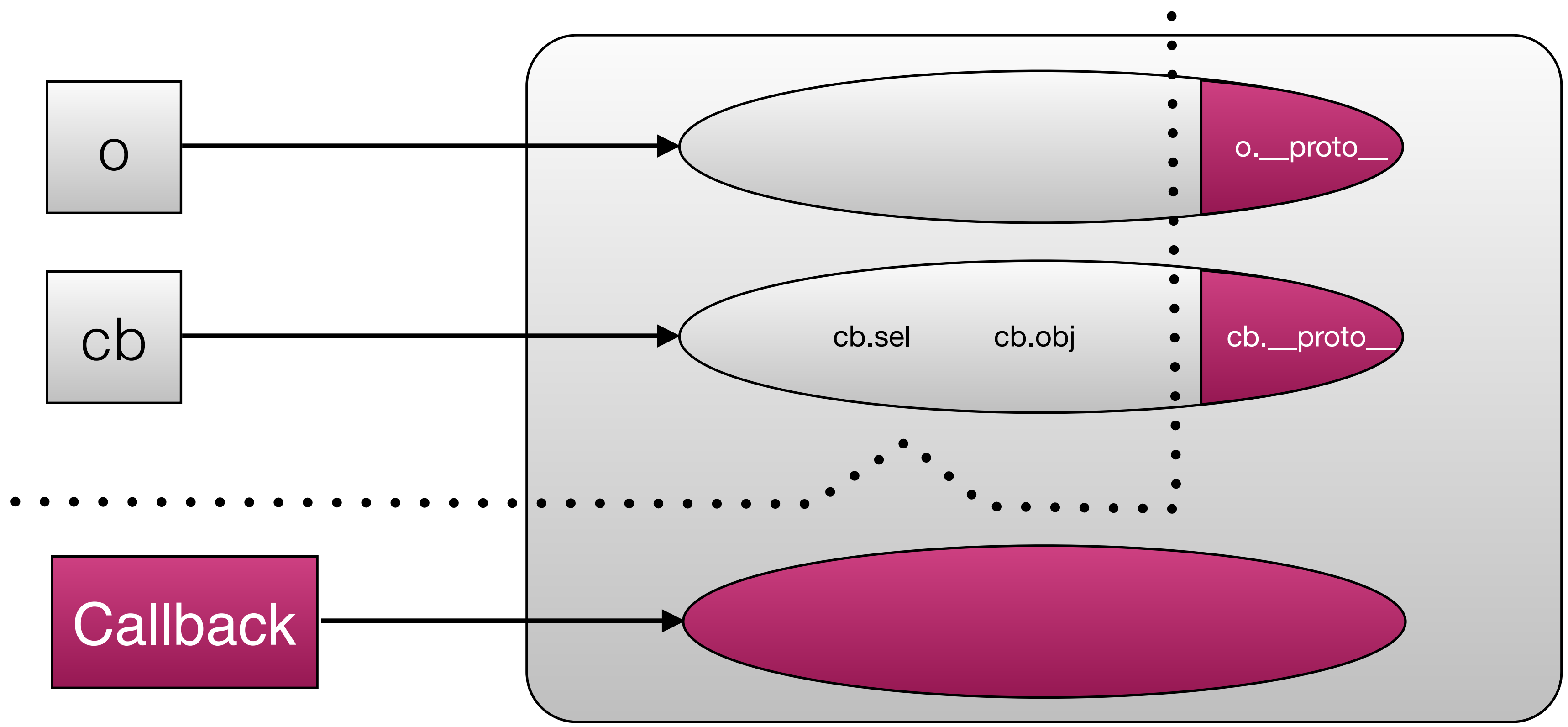
Callback in JavaScript




```
var o = ... ol
var cb = Ne
cb.call()
```

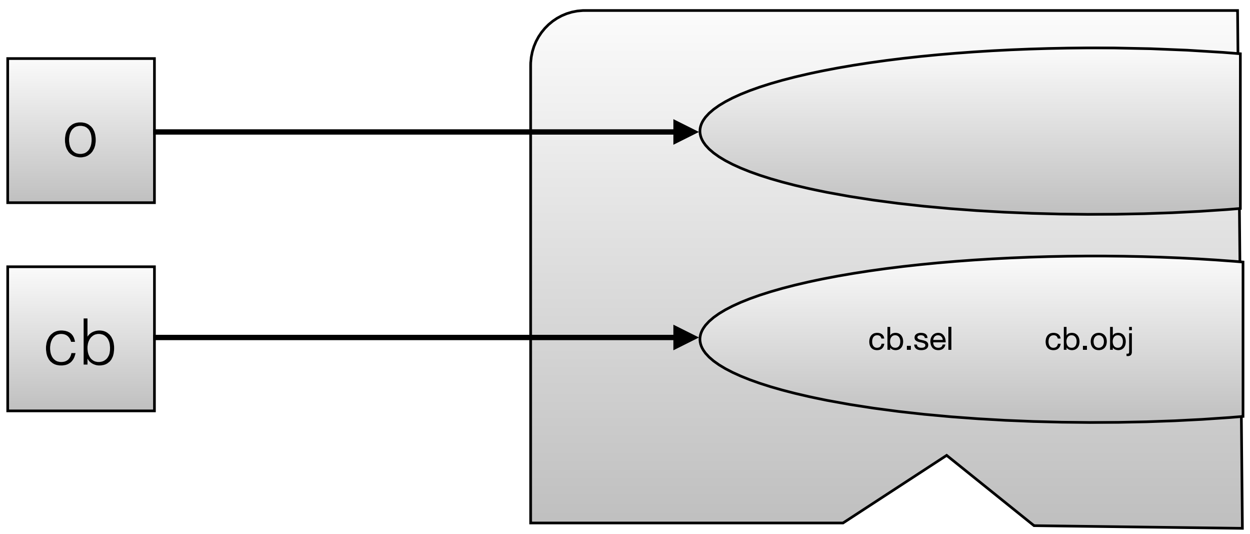
“Non-typeable heap” is gate-separated

Strong enough to ensure **type-intertwined frame rule** is sound



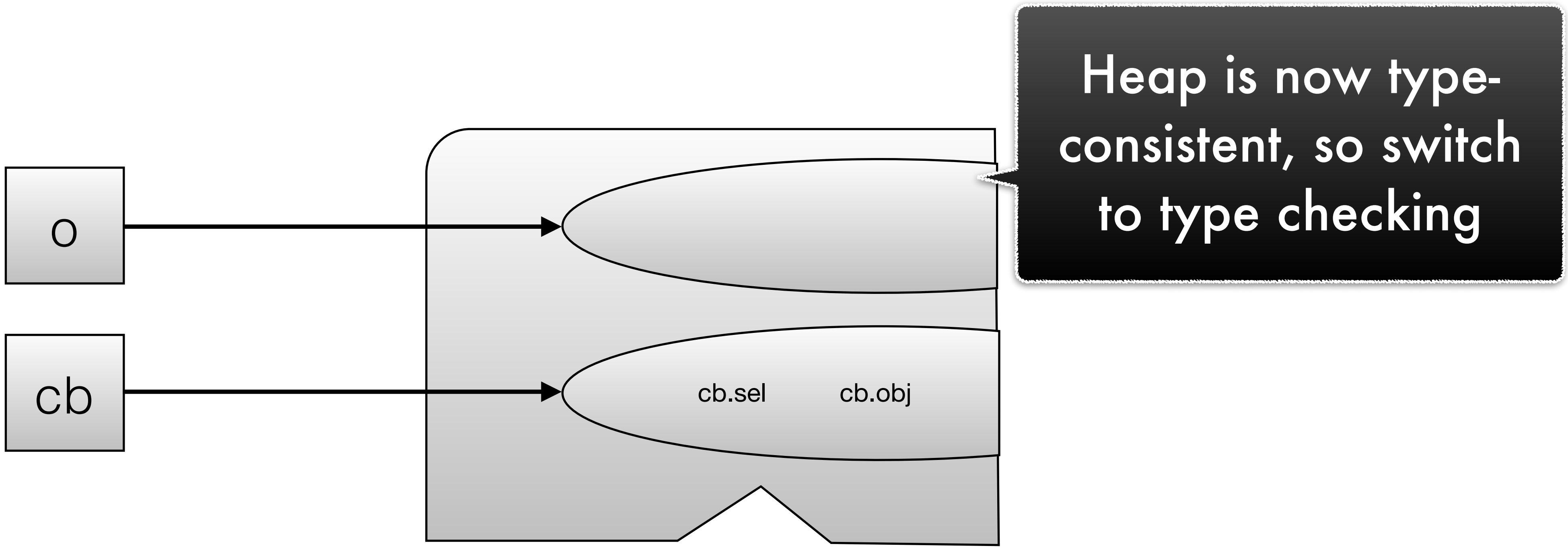

```
var o = ... object with a method m ...
var cb = New(Callback,"m",o)
cb.call()
```

Strong enough to ensure **type-intertwined frame rule** is sound



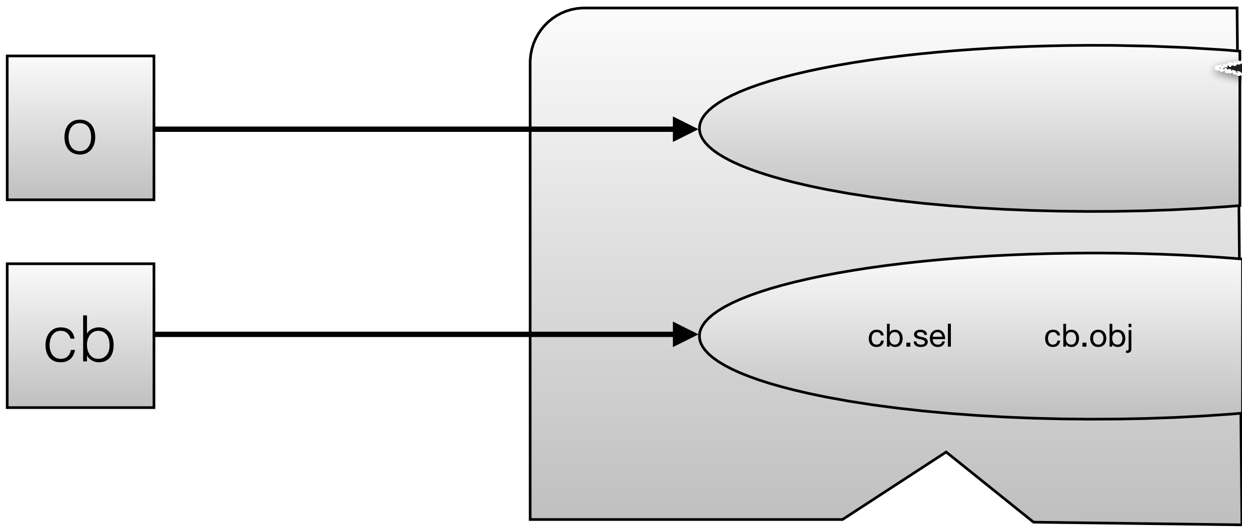

```
var o = … object with a method m …
var cb = New(Callback,"m",o)
cb.call()
```

Strong enough to ensure **type-intertwined frame rule** is sound




```
var o = ... object with a method m ...
var cb = New(Callback,“m”,o)
cb.call()
```

Strong enough to ensure **type-intertwined frame rule** is sound



Heap is now type-consistent, so switch to type checking

```
var o = … ob
var cb = Ne
cb.call()
```

Framed-out memory **cannot be touched** during type checking because of gating

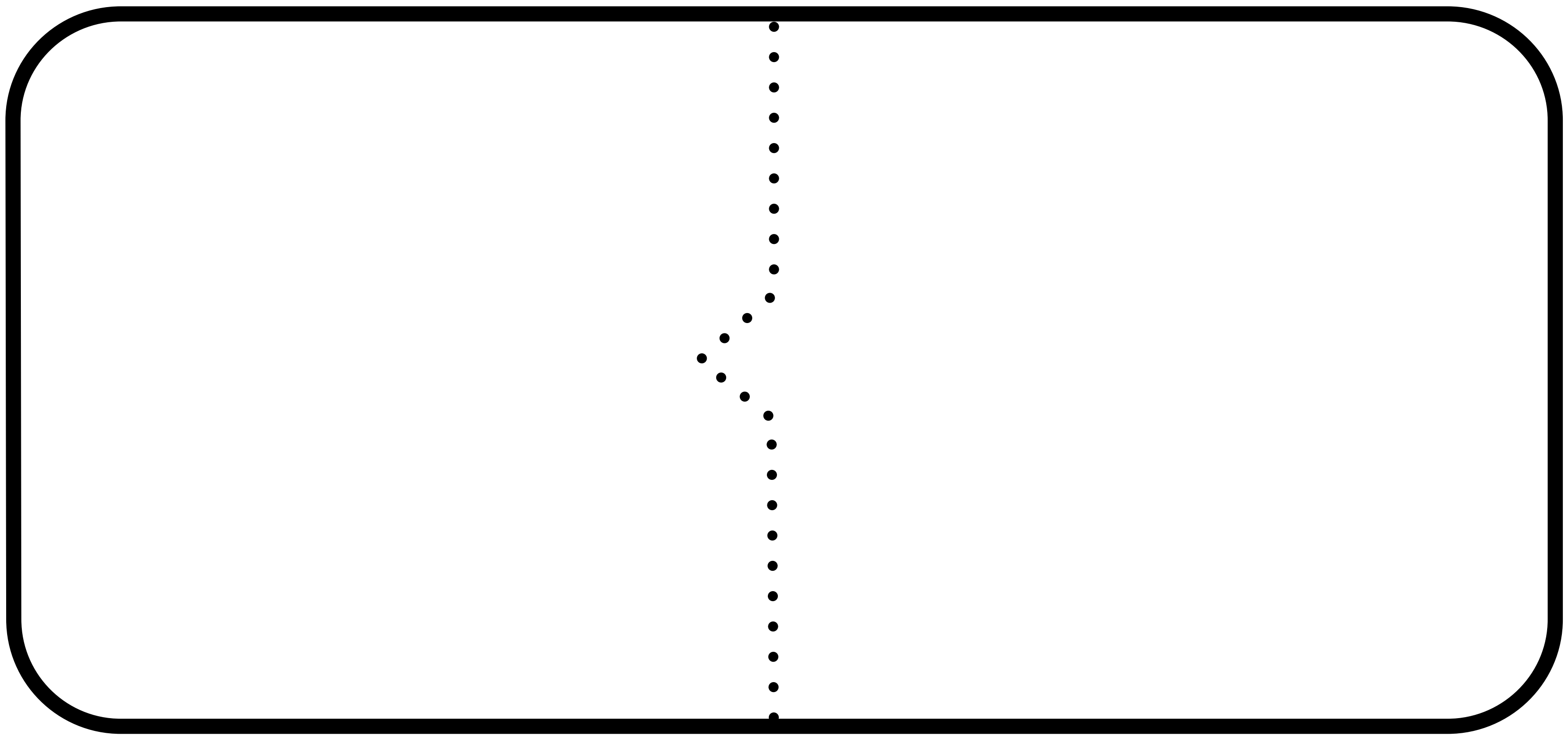
Challenge: **Static analysis** with gated separation

Challenge: **Static analysis** with gated separation

```
x.f = y
```

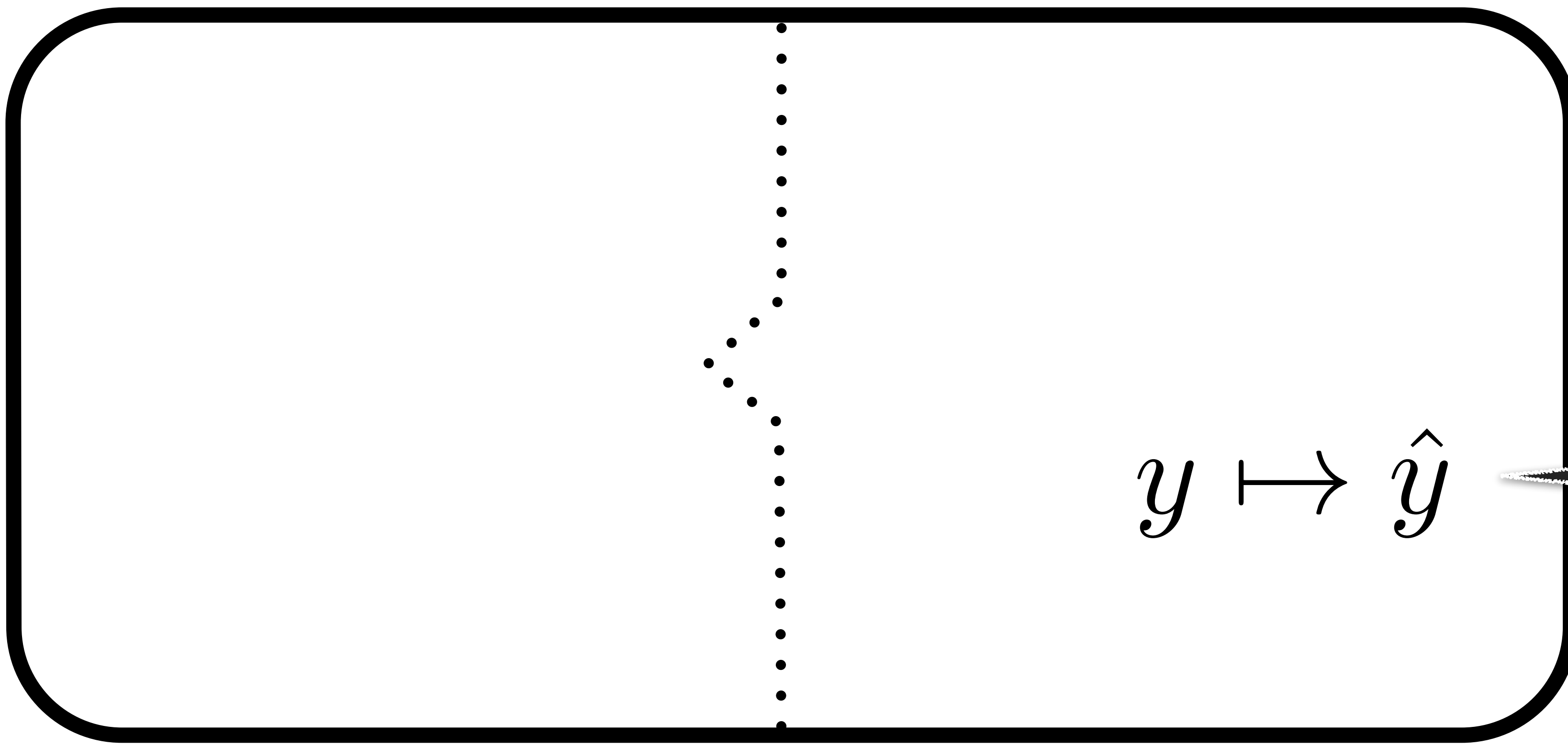
Challenge: **Static analysis** with gated separation

`x.f = y`



Challenge: **Static analysis** with gated separation

```
x.f = y
```



Value to be written

Challenge: **Static analysis** with gated separation

Cell to write in a foregate

`x.f = y`

$\hat{x} \cdot f \mapsto -$

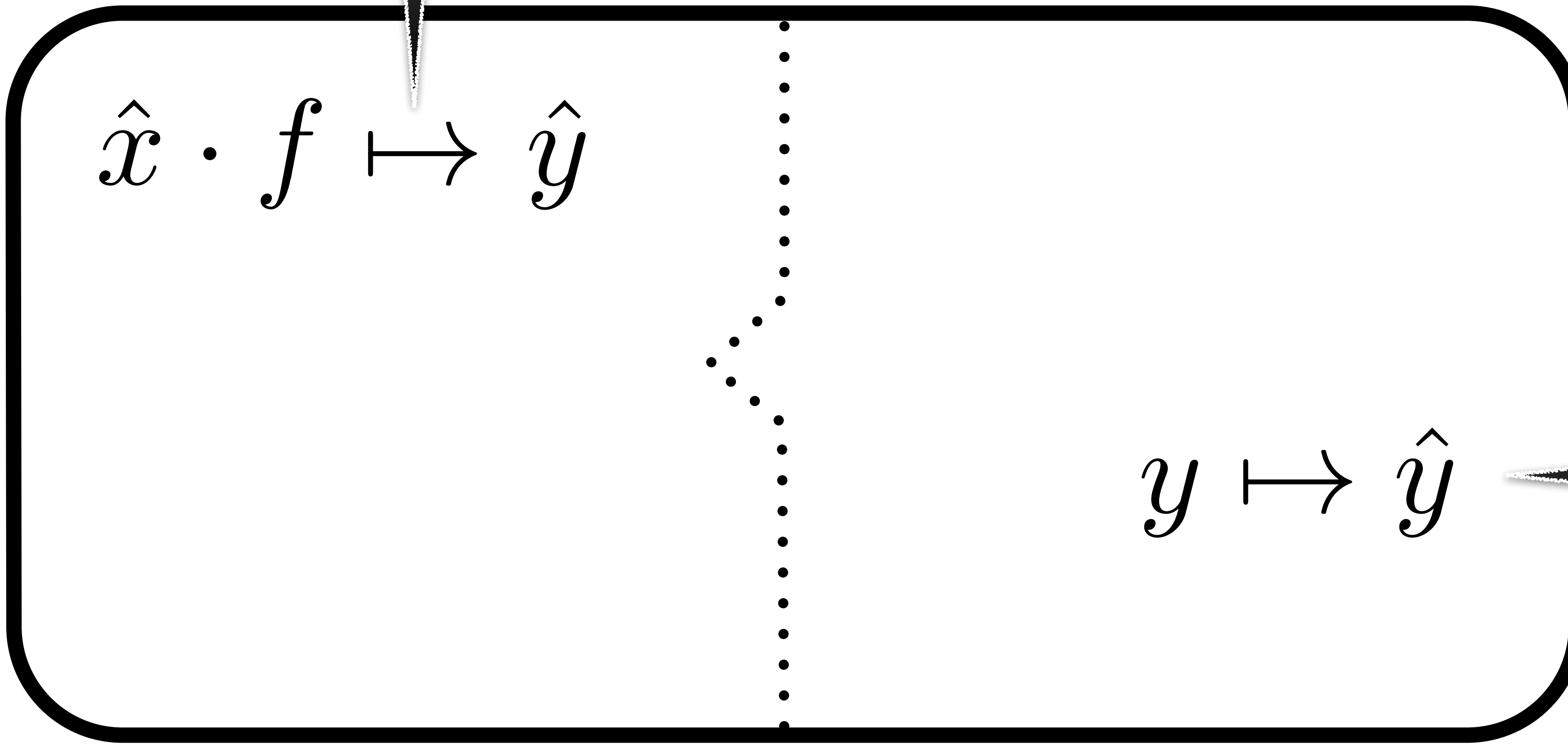
$y \mapsto \hat{y}$

Value to be written

Challenge: **Static analysis** with gated separation

```
x.f = y
```

Cell to write in a foregate



Value to be written

Challenge: **Static analysis** with gated separation

`x.f = y`

Cell to write in a foregate

$\hat{x} \cdot f \mapsto \hat{y}$

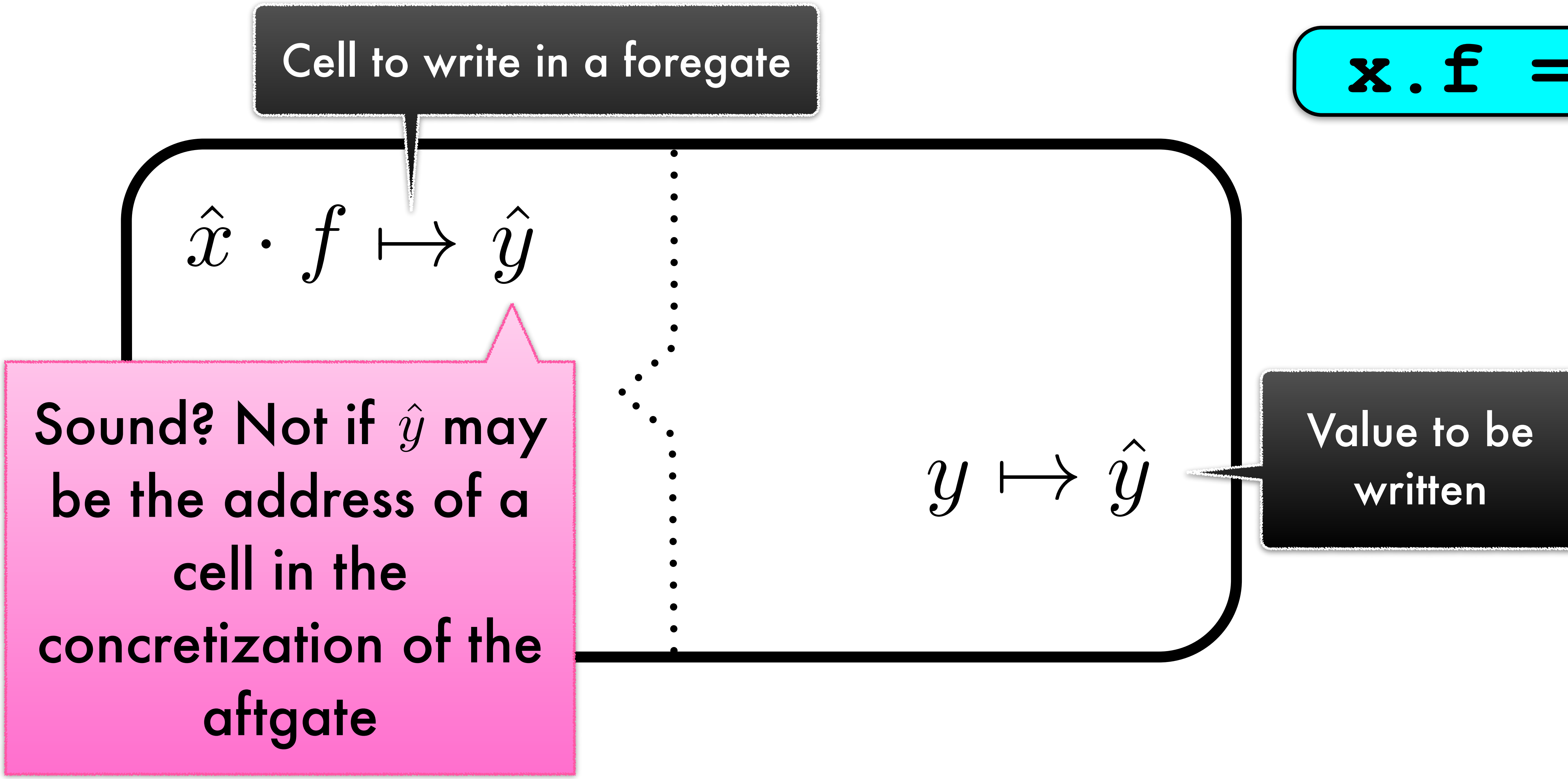
$y \mapsto \hat{y}$

Value to be written

Sound? Not if $\hat{y}$ may be the address of a cell in the concretization of the aftgate

Challenge: **Static analysis** with gated separation

`x.f = y`



Transfer functions for **writes require rearrangement** and weakening **of gated separation**

How to **type check** a program that is **almost** well-typed?

How to **type check** a program that is **almost** well-typed?

almost?

Type-Intertwined Separation Logic

Type-Intertwined Separation Logic

ok Tolerating temporary violations with **almost type-consistent heaps**

Coughlin and Chang. *POPL* 2014.

Type-Intertwined Separation Logic

ok

Tolerating temporary violations with **almost type-consistent heaps**

Coughlin and Chang. *POPL* 2014.

When the type invariant is temporarily broken

Type-Intertwined Separation Logic

ok

Tolerating temporary violations with **almost type-consistent heaps**

Coughlin and Chang. *POPL* 2014.

When the type invariant is temporarily broken

$\triangleleft\!\!\!\ast$

Type-intertwined framing with **gated separation**

Under preparation.

Type-Intertwined Separation Logic

ok Tolerating temporary violations with **almost type-consistent heaps**

Coughlin and Chang. *POPL* 2014.

When the type invariant is temporarily broken

$\triangleleft\!\!\ast$ Type-intertwined framing with **gated separation**

Under preparation.

When the type invariant applies to only part of the heap

www.cs.colorado.edu/~bec
pl.cs.colorado.edu

CU PLV

Cerny

Chang

Hammer

Sankaranaryananan

Somenzi