

# **Fissile Type Analysis: Modular Checking of Almost Everywhere Invariants**



Devin Coughlin    Bor-Yuh Evan Chang

**University of Colorado Boulder**

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

# In this talk

Example property of interest:  
**safety of reflective method calls**

Type system:  
**dependent refinement types**

# Reflective method call dispatches based on runtime string value

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

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

# Reflective method call dispatches based on runtime string value

```
class Callback
    var sel : Str
    var obj : Obj
    def call()
        this.obj.[this.sel]()
```

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

# Reflective method call dispatches based on runtime string value

```
class Callback
    var sel : Str
    var obj : Obj
    def call()
        this.obj.[this.sel]()
```

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

If **sel** held string “**notifyDidClick**” would call **notifyDidClick ()** on **obj**.

# Reflective method call dispatches based on runtime string value

```
class Callback
    var sel : Str
    var obj : Obj
    def call()
        this.obj.[this.sel]()
```

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

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

Ensure reflection safety with **dependent refinement type** expressing required **relationship**

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

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

# Ensure reflection safety with **dependent refinement type** expressing required **relationship**

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

obj must "respond to" sel

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

# Ensure reflection safety with **dependent refinement type** expressing required **relationship**

```
class Callback
  var sel : Str
  var obj : Obj | r2 sel
    obj must "respond to" sel
def call()
  this.obj.[this.sel]()
```

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

# Ensure reflection safety with **dependent refinement type** expressing required **relationship**

```
class Callback
  var sel : Str
  var obj : Obj | r2 sel
    obj must "respond to" sel
def call()
  this.obj.[this.sel]()

  Shorthand for obj :: {ν : Obj | ν r2 sel}
```

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

# Similar relationship for array bounds safety

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

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

# Similar relationship for array bounds safety

```
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

# Similar relationship for array bounds safety

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

idx must be a valid  
index into buf

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

Guarantees no  
“ArrayOutOfBoundsException” error

# Similar relationship for array bounds safety

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

idx must be a valid  
index into buf

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

These kinds of **relationships** are  
important to **many safety properties**

# Updating relationship causes type error

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

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

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

# Updating relationship causes type error

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

Field type says: `obj` must  
always respond to `sel`

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

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

# Updating relationship causes type error

```
class Callback
```

```
var sel : Str
```

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

Field type says: `obj` must  
always respond to `sel`

o guaranteed to  
respond to `s`

```
def update(s : Str, o : Obj | r2 s)
```

```
this.sel = s
```

```
this.obj = o
```

```
def call()
```

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

# Updating relationship causes type error

```
class Callback
```

```
var sel : Str
```

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

Field type says: **obj** must  
always respond to **sel**

o guaranteed to  
respond to **s**

```
def update(s : Str, o : Obj | r2 s)
```

```
this.sel = s
```

```
this.obj = o
```

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

```
def call()
```

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

# Updating relationship causes type error

```
class Callback
```

```
var sel : Str
```

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

Field type says: **obj** must  
always respond to **sel**

o guaranteed to  
respond to **s**

```
def update(s : Str, o : Obj | r2 s)
```

```
this.sel = s
```

```
this.obj = o
```

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

```
def call()
```

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

False alarm: no runtime error

## Two styles of **reasoning** to determine **false alarm**

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

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

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

## Two styles of **reasoning** to determine **false alarm**

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

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

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

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

## Two styles of reasoning to determine **false alarm**

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

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

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

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

# Two styles of reasoning to determine false alarm

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

```
def update(s : Str, o : Obj) r2 s )
```

```
this.sel = s
this.obj = o
```

Reasoning about **effects** of  
**imperative updates**

```
def call()
```

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

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

# Two styles of reasoning to determine false alarm

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

```
def update(s : Str, o : Obj) r2 s )
```

```
this.sel = s
this.obj = o
```

Reasoning about **effects** of  
**imperative updates**

```
Str, o : Obj
```

Relationship violated

```
def call()
```

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

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

# Two styles of reasoning to determine false alarm

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

```
def update(s : Str, o : Obj) r2 s )
```

```
this.sel = s
this.obj = o
```

Reasoning about **effects** of  
**imperative updates**

```
Str, o : Obj
```

Relationship violated

Relationship restored

```
def call()
```

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

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

**Idea: Selectively  
alternate between  
reasoning styles in  
verification**

Fissile Type Analysis **combines** two styles of reasoning

Fissile Type Analysis **combines** two styles of reasoning

Automated **reasoning**  
about **global invariants**

# Fissile Type Analysis **combines** two styles of reasoning

Automated **reasoning**  
about **global invariants**

$\Gamma \vdash \dots$   
**Flow-Insensitive  
Type Systems**

# Fissile Type Analysis **combines** two styles of reasoning

Automated **reasoning**  
about **global invariants**

$\Gamma \vdash \dots$   
**Flow-Insensitive  
Type Systems**

Automated **reasoning**  
about **execution**

# Fissile Type Analysis **combines** two styles of reasoning

Automated **reasoning**  
about **global invariants**

$\Gamma \vdash \dots$   
**Flow-Insensitive  
Type Systems**

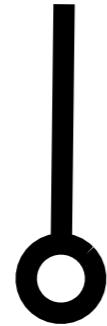
Automated **reasoning**  
about **execution**

$\gamma(\cdot) = \dots$   
**Abstract Interpretation/  
Flow Analysis**

# Verification of **almost-everywhere** invariants with **intertwined** type and flow analysis

# Verification of **almost-everywhere** invariants with intertwined type and flow analysis

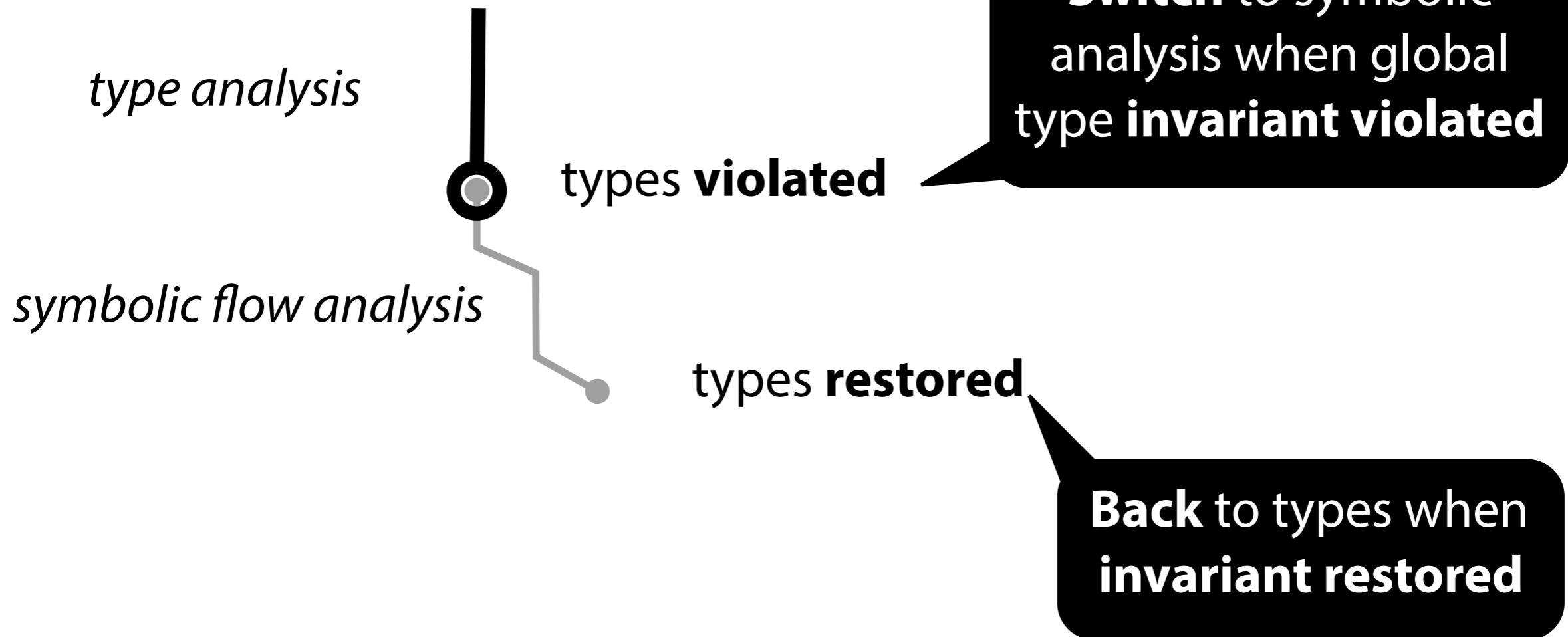
*type analysis*



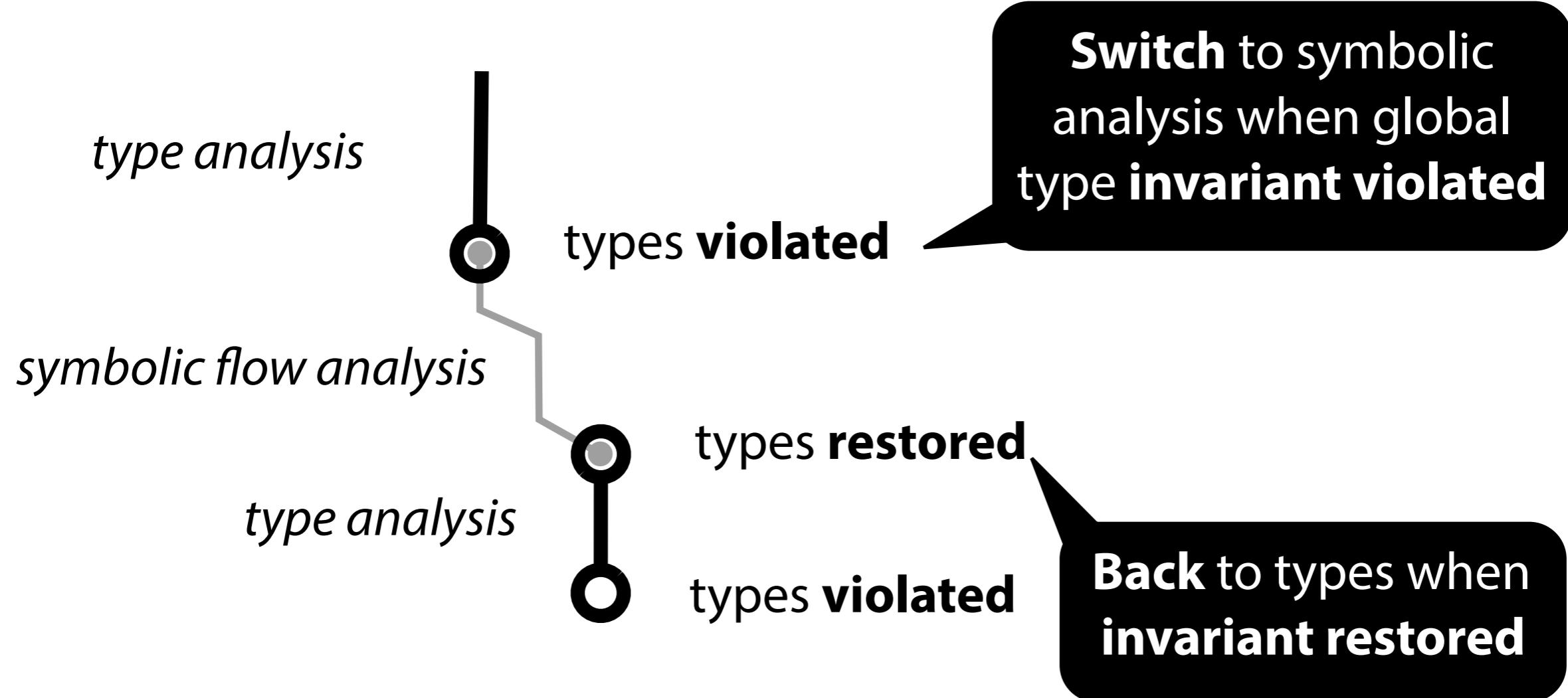
types **violated**

**Switch** to symbolic  
analysis when global  
type **invariant violated**

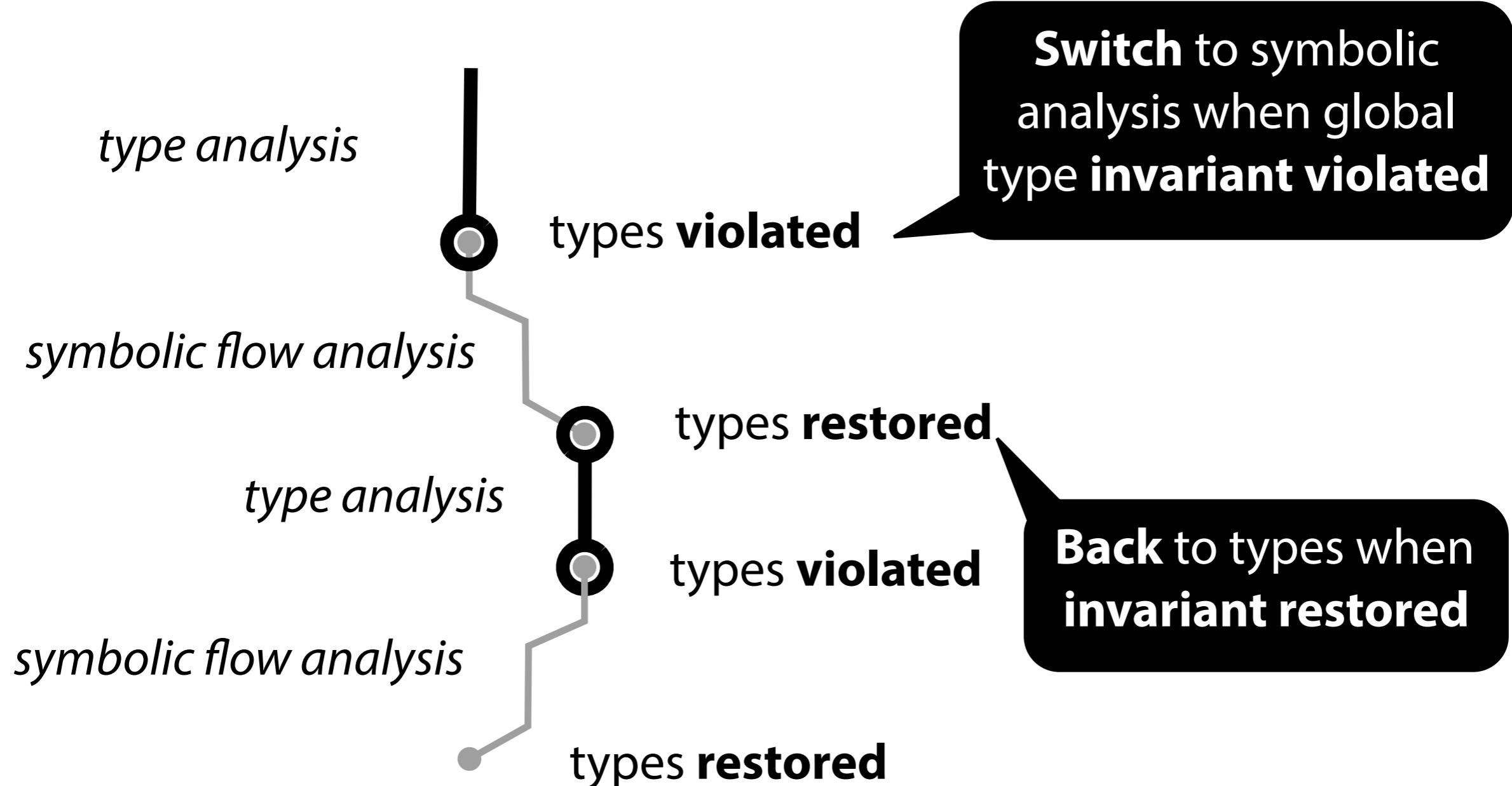
# Verification of **almost-everywhere** invariants with intertwined type and flow analysis



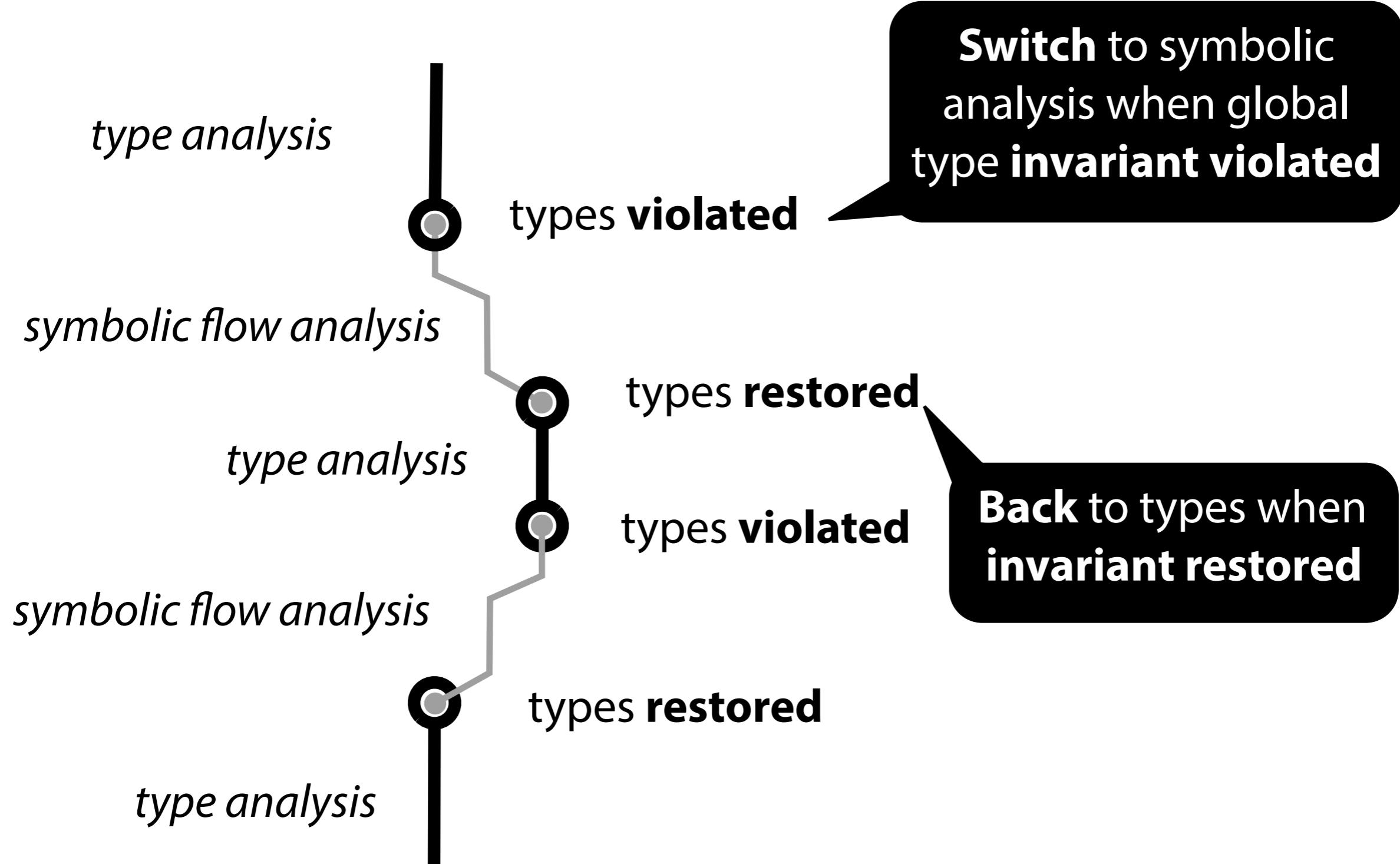
# Verification of **almost-everywhere** invariants with intertwined type and flow analysis



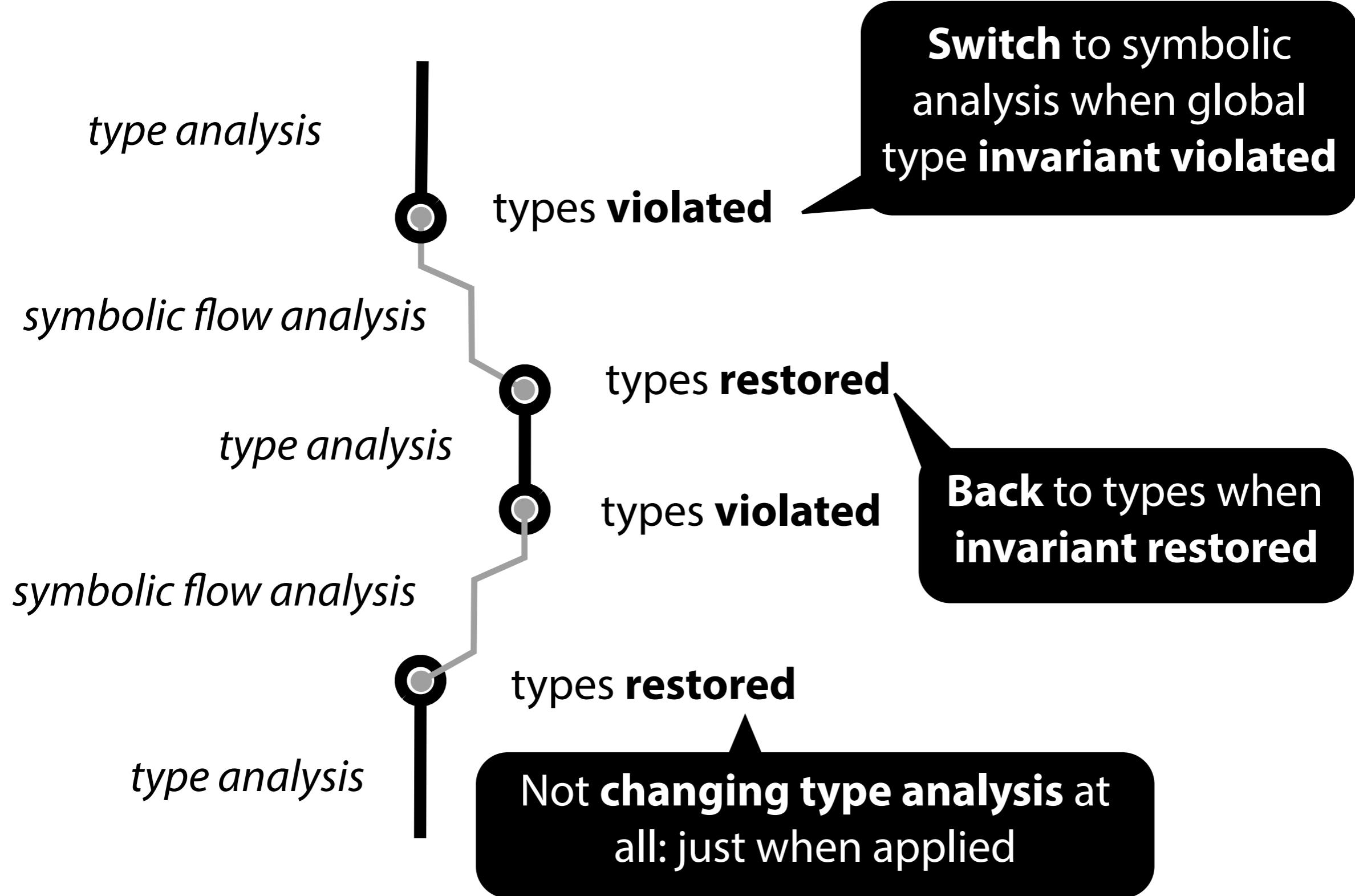
# Verification of **almost-everywhere** invariants with intertwined type and flow analysis



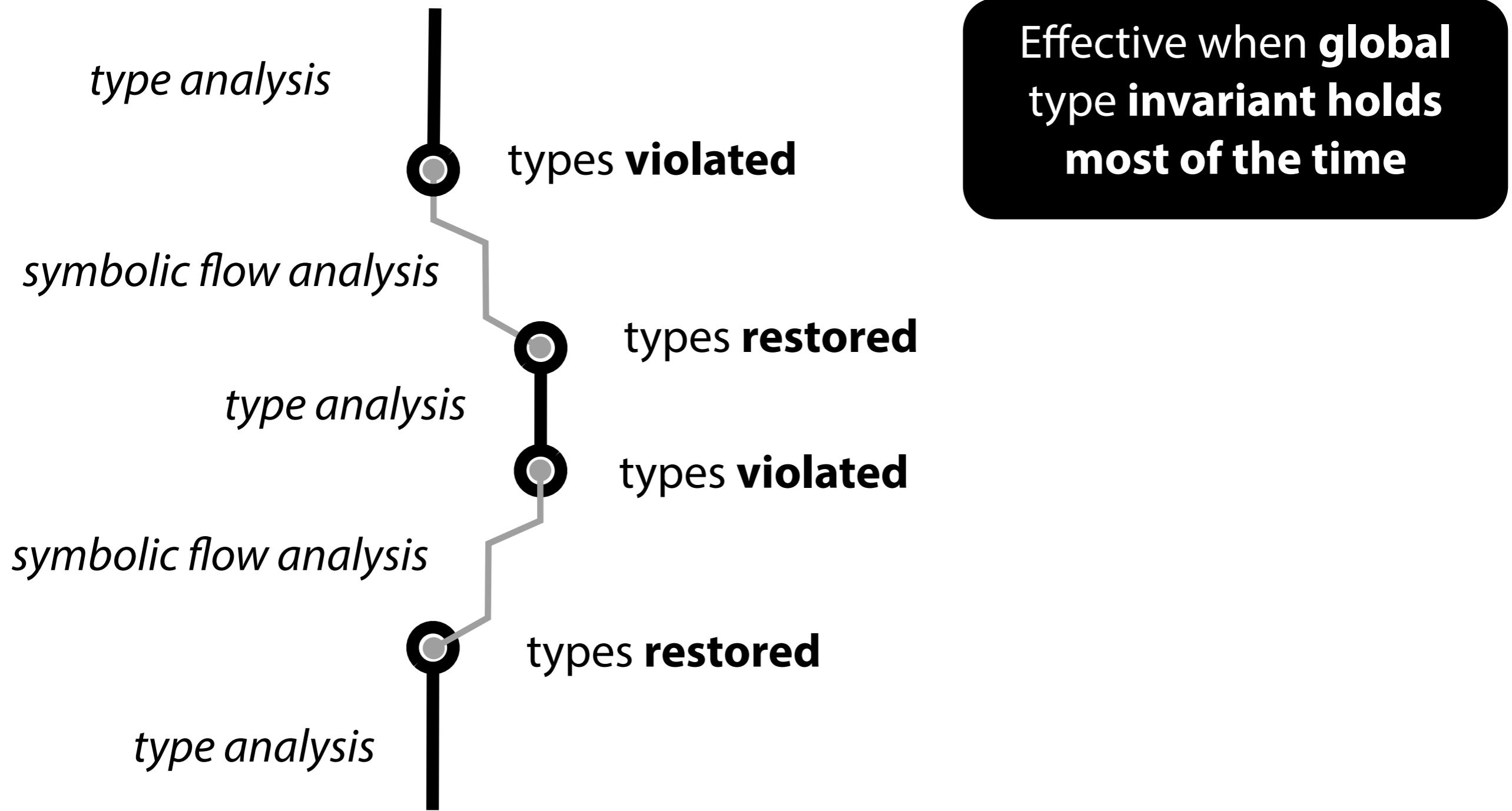
# Verification of **almost-everywhere** invariants with intertwined type and flow analysis



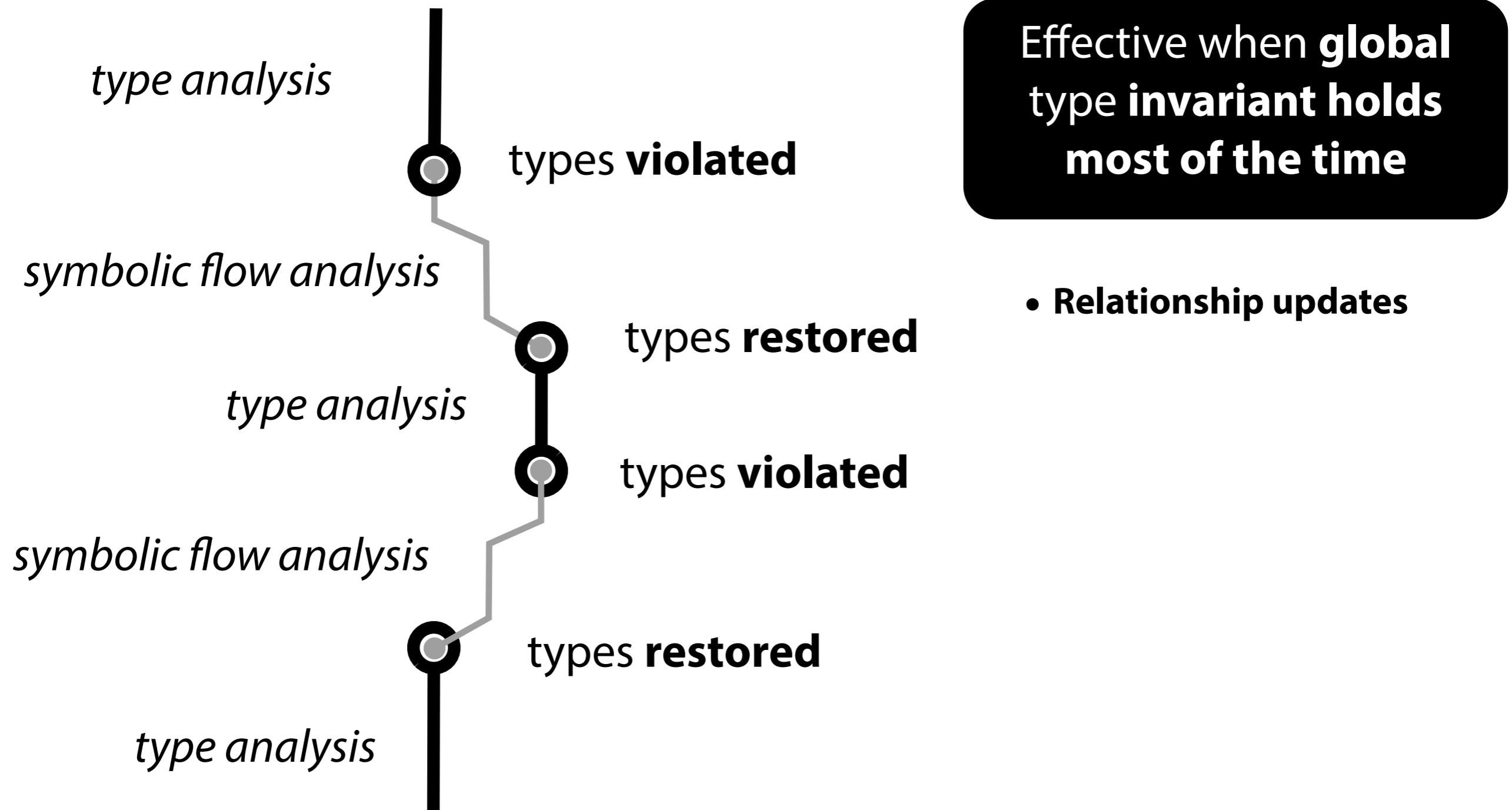
# Verification of **almost-everywhere** invariants with intertwined type and flow analysis



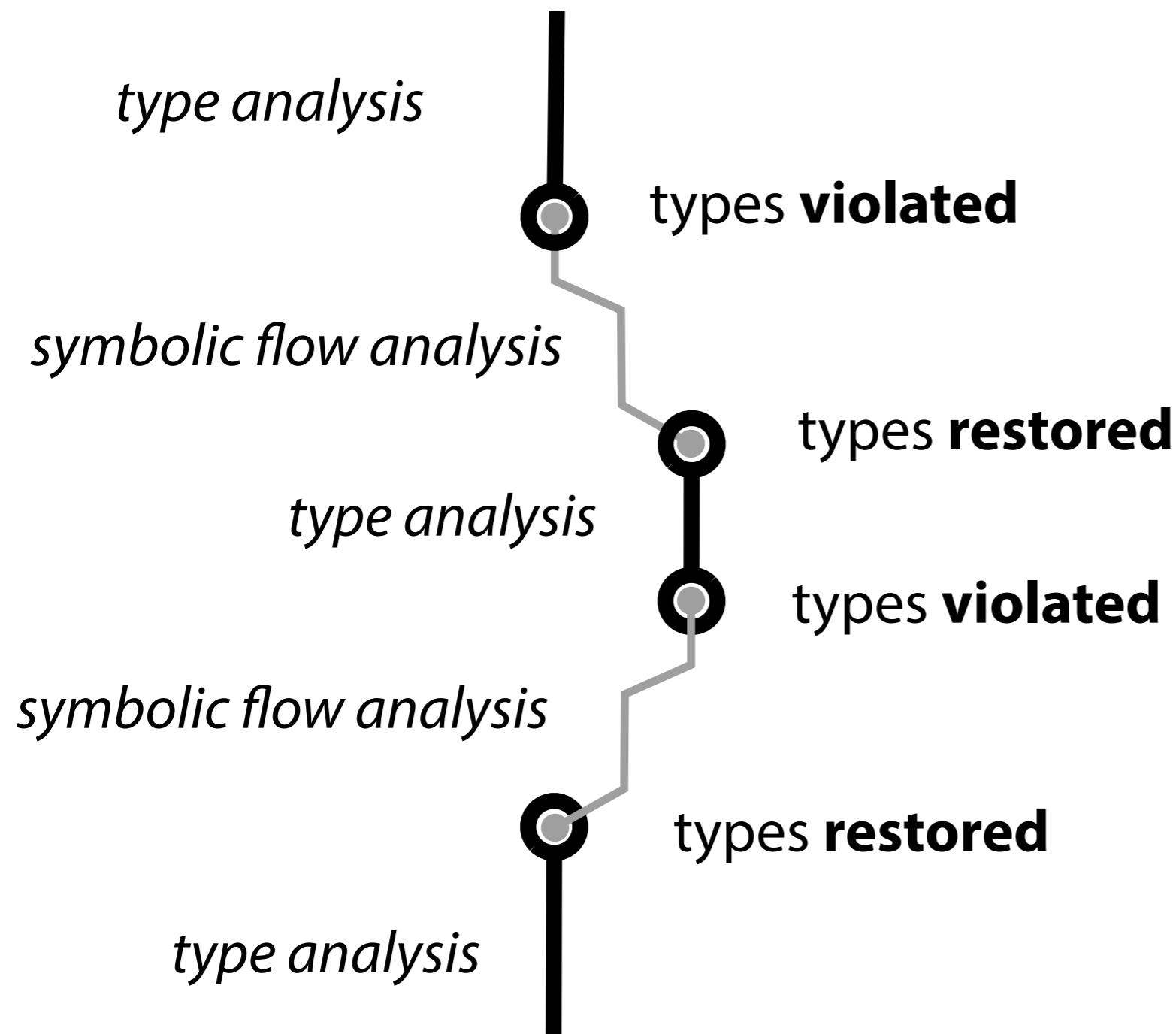
# Verification of almost-everywhere invariants with intertwined type and flow analysis



# Verification of **almost-everywhere** invariants with intertwined type and flow analysis



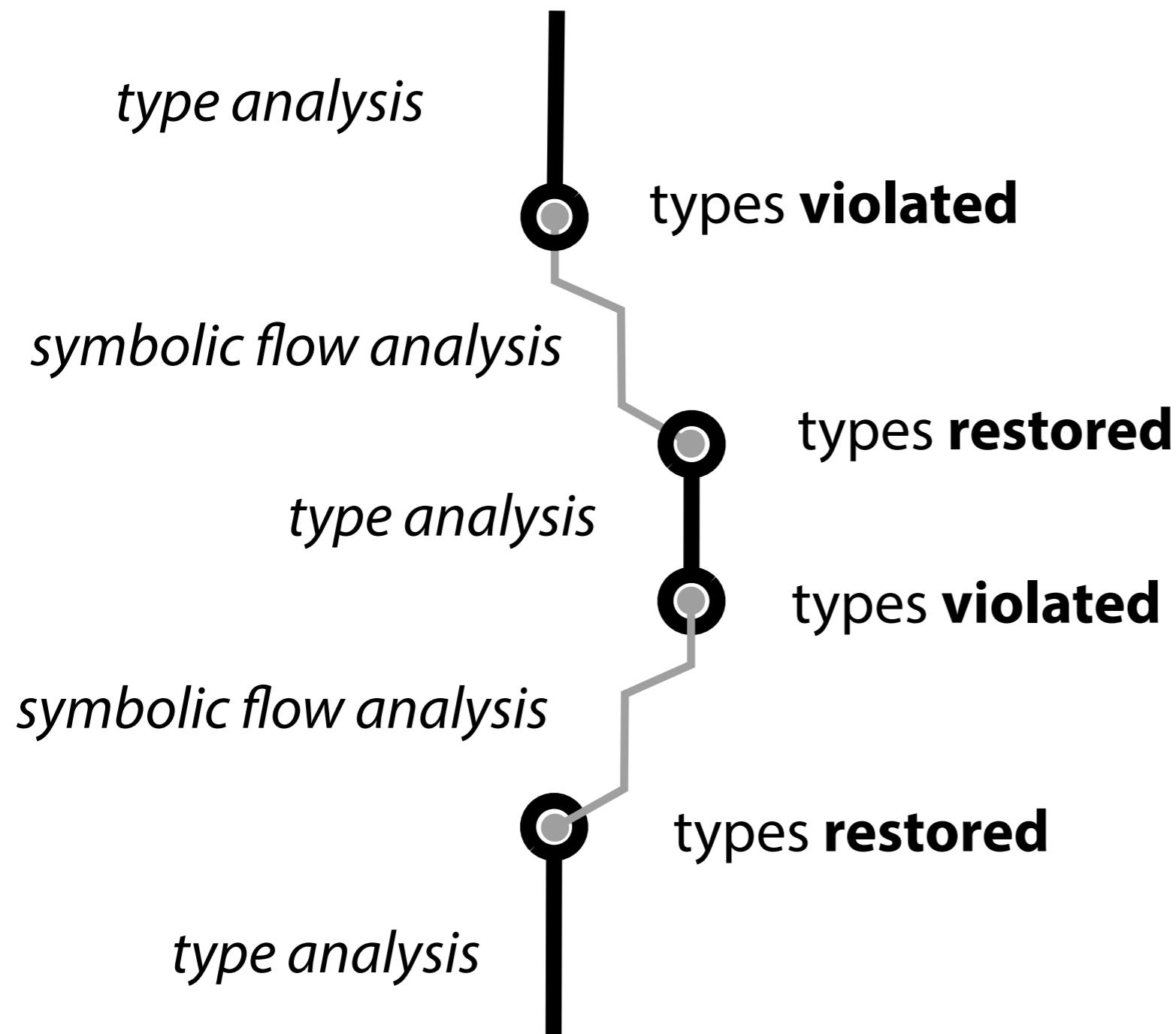
# Verification of **almost-everywhere** invariants with intertwined type and flow analysis



Effective when **global type invariant holds most of the time**

- Relationship updates
- Occurrence typing

# Verification of **almost-everywhere** invariants with intertwined type and flow analysis



Effective when **global type invariant holds most of the time**

- Relationship updates
- Occurrence typing
- Tagged unions

Play to the **strengths** of each intertwined **analysis**

Play to the **strengths** of each intertwined **analysis**

## Flow-Insensitive Types

- Easy to **specify global** invariants
- **Fast**
- Natural for **modular** reasoning
- Good **error reporting**

Play to the **strengths** of each intertwined **analysis**

## Flow-Insensitive Types

- Easy to **specify global** invariants
- **Fast**
- Natural for **modular** reasoning
- Good **error reporting**

## Symbolic Flow Analysis

- Natural for **local** reasoning about  
**heap mutation**
- **Precise**
- Can be disjunctive/path-sensitive

# Play to the **strengths** of each intertwined **analysis**

## Flow-Insensitive Types

- Easy to **specify global** invariants
- **Fast**
- Natural for **modular** reasoning
- Good **error reporting**

flow-sensitive typing?  
ownership types?  
alias types?  
permissions?  
effects?

## Symbolic Flow Analysis

- Natural for **local** reasoning about  
**heap mutation**
- **Precise**
- Can be disjunctive/path-sensitive

# Play to the **strengths** of each intertwined **analysis**

## Flow-Insensitive Types

- Easy to **specify global** invariants
- **Fast**
- Natural for **modular** reasoning
- Good **error reporting**

flow-sensitive typing?  
ownership types?  
alias types?  
permissions?  
effects?

Goal: keep **types**  
as **simple as**  
**possible**

## Symbolic Flow Analysis

- Natural for **local** reasoning about  
**heap mutation**
- **Precise**
- Can be disjunctive/path-sensitive

# Play to the **strengths** of each intertwined **analysis**

## Flow-Insensitive Types

- Easy to **specify global** invariants
- **Fast**
- Natural for **modular** reasoning
- Good **error reporting**

flow-sensitive typing?  
ownership types?  
alias types?  
permissions?  
effects?

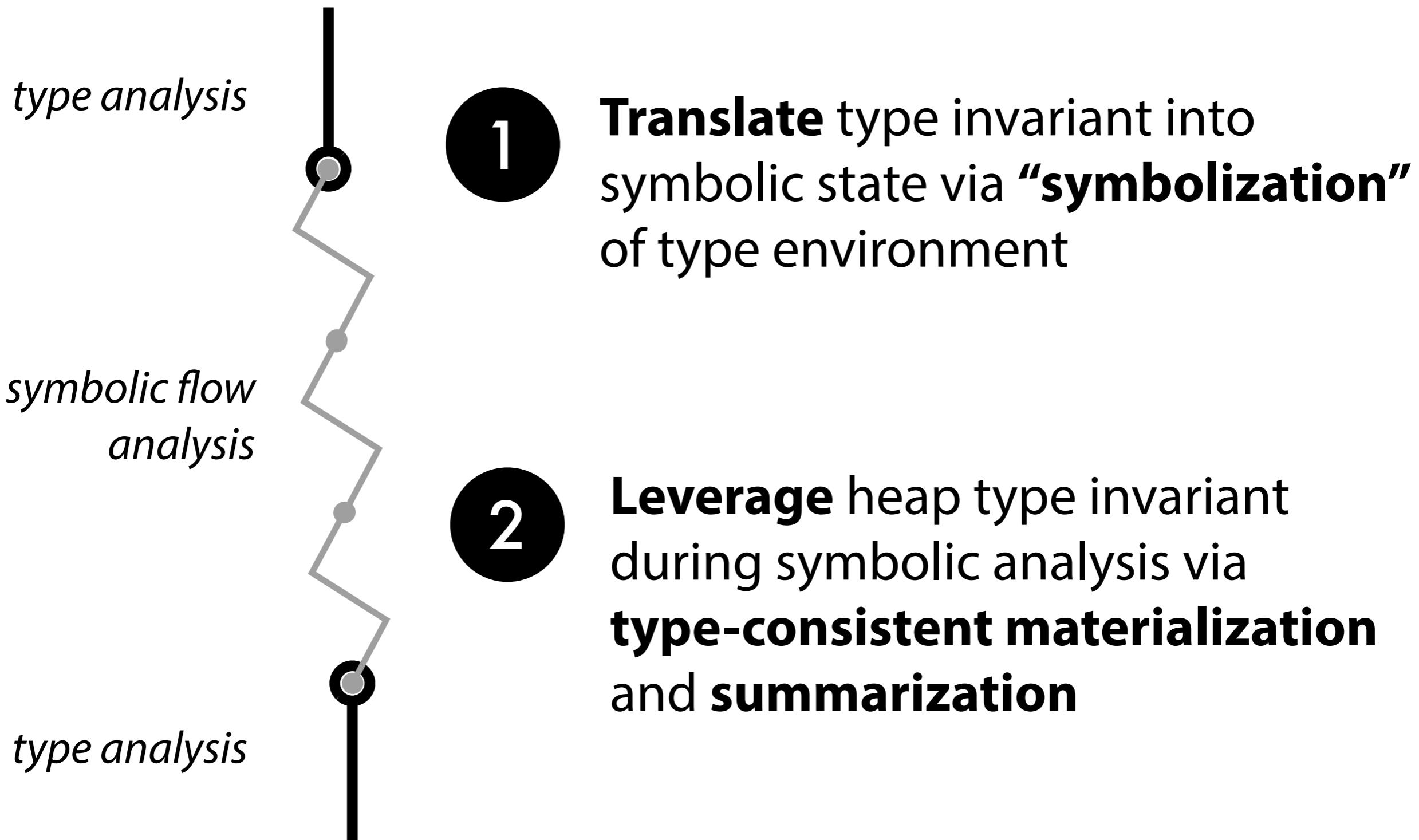
Goal: keep **types**  
as **simple as**  
**possible**

## Symbolic Flow Analysis

- Natural for **local** reasoning about  
**heap mutation**
- **Precise**
- Can be disjunctive/path-sensitive

**Complexity** lies in **handoff** between  
analyses and in **symbolic analysis**

# Key Contributions

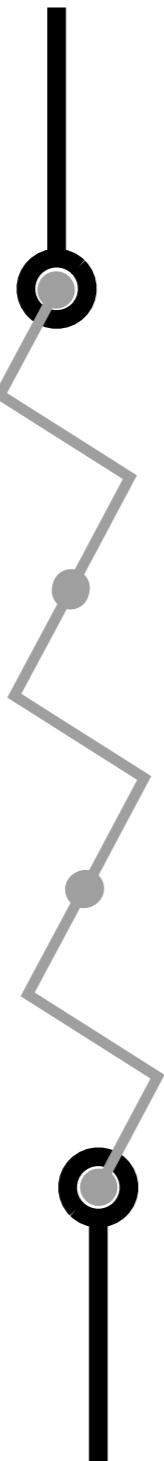


# Key Contributions

*type analysis*

*symbolic flow analysis*

*type analysis*



1

**Translate** type invariant into symbolic state via “**symbolization**” of type environment

Reason precisely only **when** type invariant violated

2

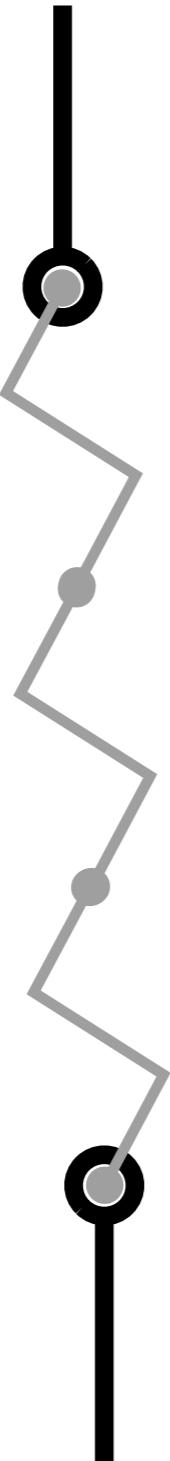
**Leverage** heap type invariant during symbolic analysis via **type-consistent materialization** and **summarization**

# Key Contributions

*type analysis*

*symbolic flow analysis*

*type analysis*



1

**Translate** type invariant into symbolic state via “**symbolization**” of type environment

Reason precisely only **when** type invariant violated

2

**Leverage** heap type invariant during symbolic analysis via **type-consistent materialization** and **summarization**

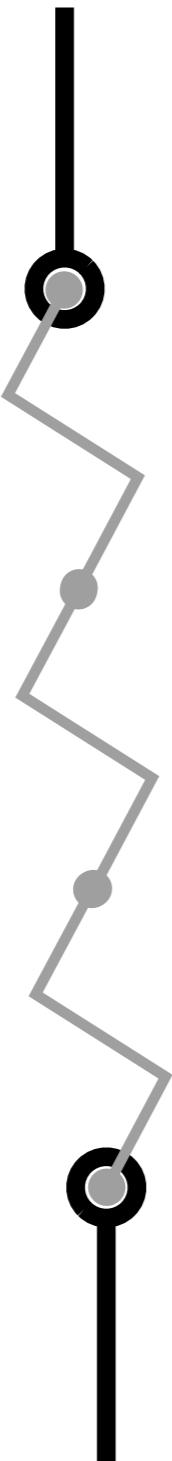
Reason precisely only for locations **where** type invariant violated

# Key Contributions

*type analysis*

*symbolic flow  
analysis*

*type analysis*



1

**Translate** type invariant into symbolic state via “**symbolization**” of type environment

Reason precisely only **when** type invariant violated

2

**Leverage** heap type invariant during symbolic analysis via **type-consistent materialization** and **summarization**

Reason precisely only for locations **where** type invariant violated

**Symbolization splits a type environment into facts about values and storage for those values**

# Symbolization splits a type environment into facts about values and storage for those values

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

# Symbolization splits a type environment into facts about values and storage for those values

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

# Symbolization splits a type environment into facts about values and storage for those values

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

# Symbolization splits a type environment into facts about **values** and **storage** for those values

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

Type environment

Maps local **variables**  
to dependent **types**

$$\Gamma \quad \begin{array}{l} s : \text{Str} \\ o : \text{Obj} \mid r2 s \\ \text{this} : \text{Callback} \end{array}$$

Refinements  
refer to  
**variables**

# Symbolization splits a type environment into facts about values and storage for those values

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

Type environment

Maps local **variables** to dependent **types**

$$\Gamma \quad \begin{array}{l} s : \text{Str} \\ o : \text{Obj} \mid \text{r2 } s \\ \text{this} : \text{Callback} \end{array}$$

Symbolic state

**symbolize**

$$\begin{array}{l} \tilde{\Gamma} \\ \tilde{E} \\ \tilde{s} : \text{Str} \\ \tilde{o} : \text{Obj} \mid \text{r2 } \tilde{s} \\ \text{this} : \tilde{t} \\ \tilde{\Gamma} \end{array}$$

Refinements refer to **variables**

# Symbolization splits a type environment into facts about **values** and **storage** for those values

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

## Type environment

Maps local **variables** to dependent **types**

$$\Gamma \quad \begin{array}{l} s : \text{Str} \\ o : \text{Obj} \mid \text{r2 } s \\ \text{this} : \text{Callback} \end{array}$$

Refinements refer to **variables**

Maps local variables to symbolic **values**

**symbolize**

## Symbolic state

$$\begin{array}{l} \tilde{\Gamma} \\ \tilde{E} \\ \tilde{s} : \text{Str} \\ \tilde{o} : \text{Obj} \mid \text{r2 } \tilde{s} \\ \text{this} : \tilde{t} \\ \tilde{t} : \text{Callback} \end{array}$$

# Symbolization splits a type environment into facts about values and storage for those values

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

## Type environment

Maps local **variables** to dependent **types**

$$\Gamma \quad \begin{array}{l} s : \text{Str} \\ o : \text{Obj} \mid \text{r2 } s \\ \text{this} : \text{Callback} \end{array}$$

Refinements refer to **variables**

Maps local variables to symbolic **values**

**symbolize**

## Symbolic state

$$\begin{array}{l} \tilde{\Gamma} \\ \tilde{E} \\ \tilde{s} : \text{Str} \\ \tilde{o} : \text{Obj} \mid \text{r2 } \tilde{s} \\ \text{this} : \tilde{t} \\ \tilde{t} : \text{Callback} \end{array}$$

Maps symbolic values to dependent types **lifted** to symbolic values (**symbolic facts**)

# Symbolization splits a type environment into facts about values and storage for those values

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

## Type environment

Maps local **variables** to dependent **types**

$$\Gamma \quad \begin{array}{l} s : \text{Str} \\ o : \text{Obj} \mid \text{r2 } s \\ \text{this} : \text{Callback} \end{array}$$

Refinements refer to **variables**

Maps local variables to symbolic **values**

symbolize →

## Symbolic state

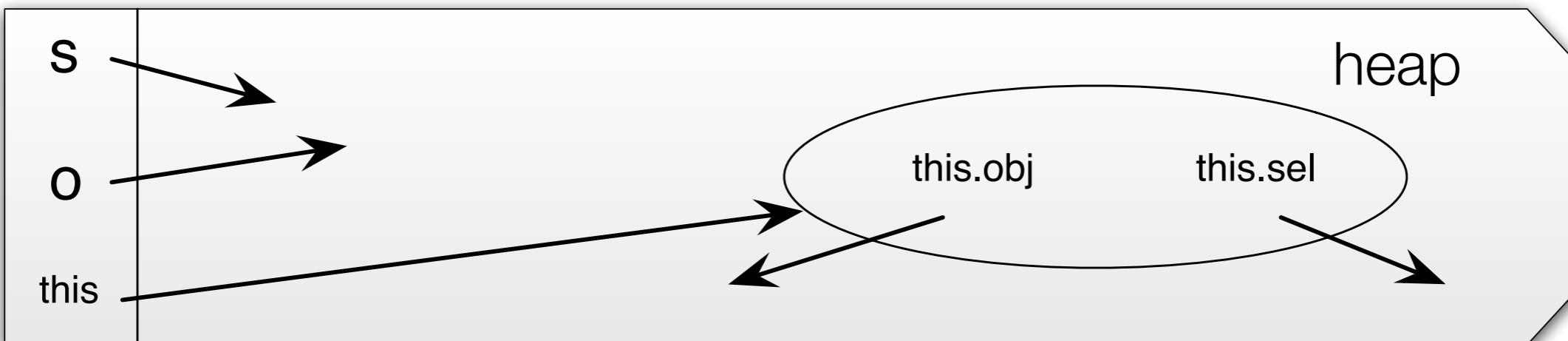
$$\begin{array}{l} \tilde{\Gamma} \\ \tilde{s} : \tilde{\text{Str}} \\ \tilde{o} : \tilde{\text{Obj}} \mid \text{r2 } \tilde{s} \\ \text{this} : \tilde{\text{t}} \\ \tilde{\text{t}} : \text{Callback} \end{array}$$

Refinements refer to **values**

Maps symbolic values to dependent types **lifted** to symbolic values (**symbolic facts**)

# Symbolization allows local variables to hold values inconsistent with declared types

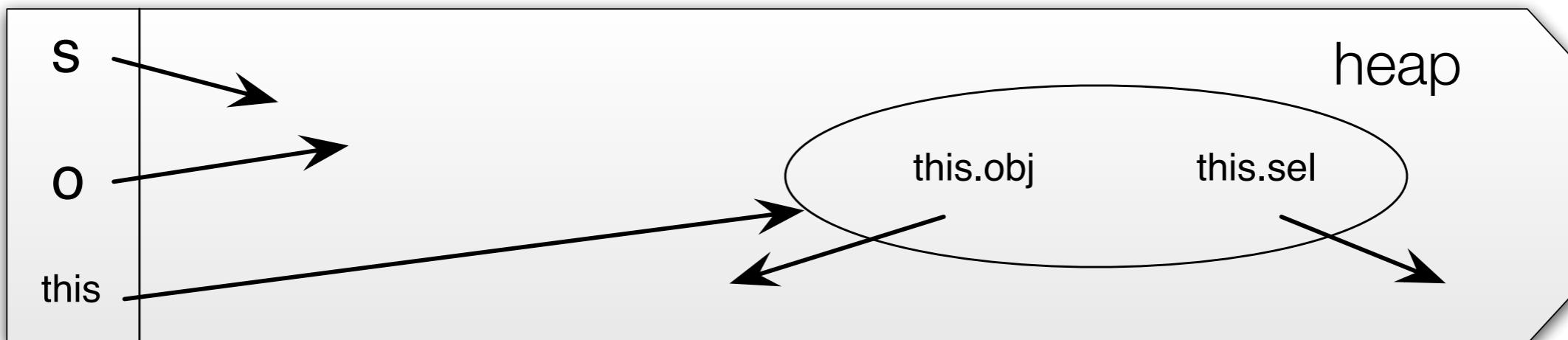
```
def update(s:Str, o:Obj | r2 s)
  this.sel = s
  this.obj = o
```

$$\Gamma \quad \begin{array}{l} s : \text{Str} \\ o : \text{Obj} \mid \text{r2 } s \\ \text{this} : \text{Callback} \end{array}$$


# Symbolization allows local variables to hold values inconsistent with declared types

A type environment constrains local variables

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

$$\Gamma \quad \begin{array}{l} s : \text{Str} \\ o : \text{Obj} \mid \text{r2 } s \\ \text{this} : \text{Callback} \end{array}$$


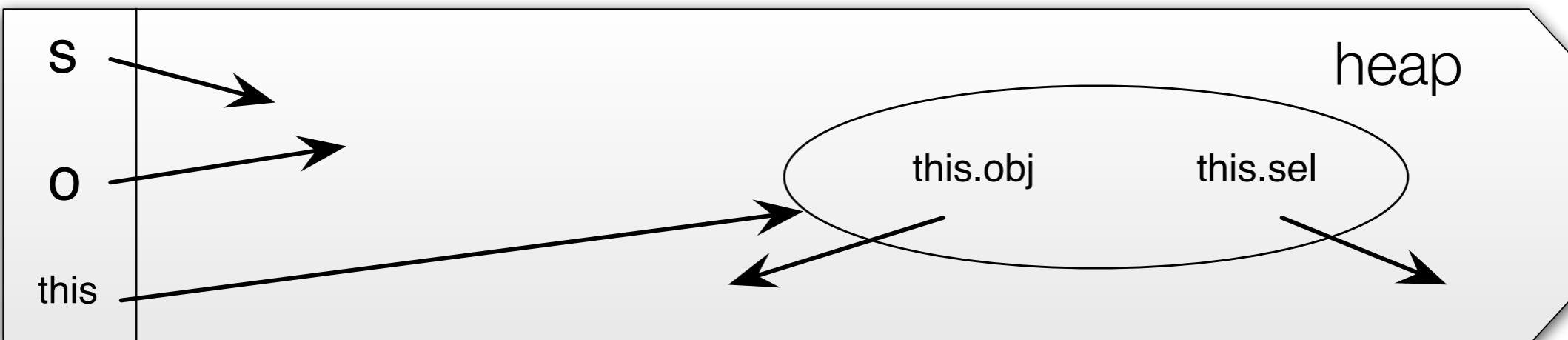
# Symbolization allows local variables to hold values inconsistent with declared types

A type environment constrains local variables

$$\Gamma \quad \begin{array}{l} s : \text{Str} \\ o : \text{Obj} \mid r2 s \\ \text{this} : \text{Callback} \end{array}$$

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

But also constrains the reachable heap to be **type-consistent**: fields must **conform to declared types**



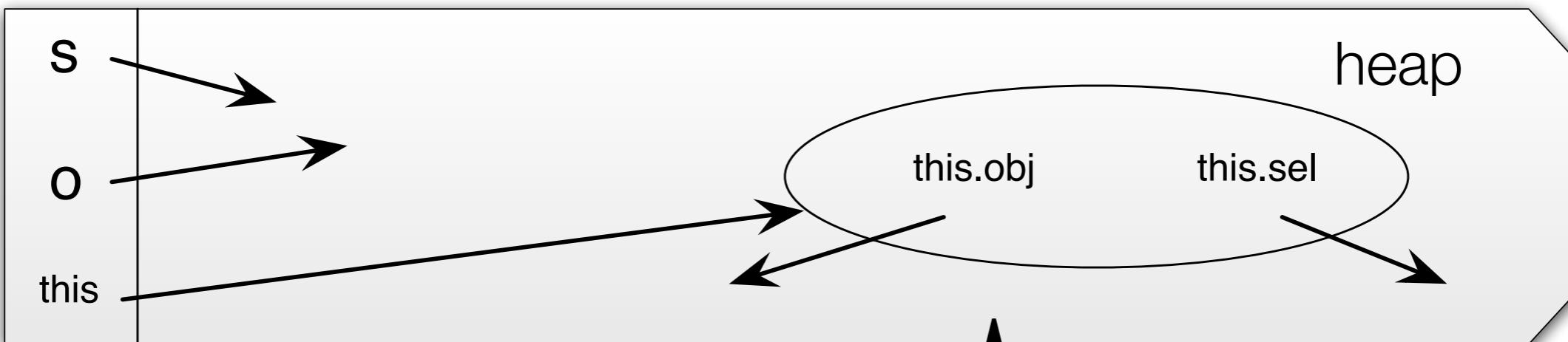
# Symbolization allows local variables to hold values inconsistent with declared types

A type environment constrains local variables

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

$$\Gamma \quad \begin{array}{l} s : \text{Str} \\ o : \text{Obj} \mid \text{r2 } s \\ \text{this} : \text{Callback} \end{array}$$

But also constrains the reachable heap to be **type-consistent**: fields must **conform to declared types**



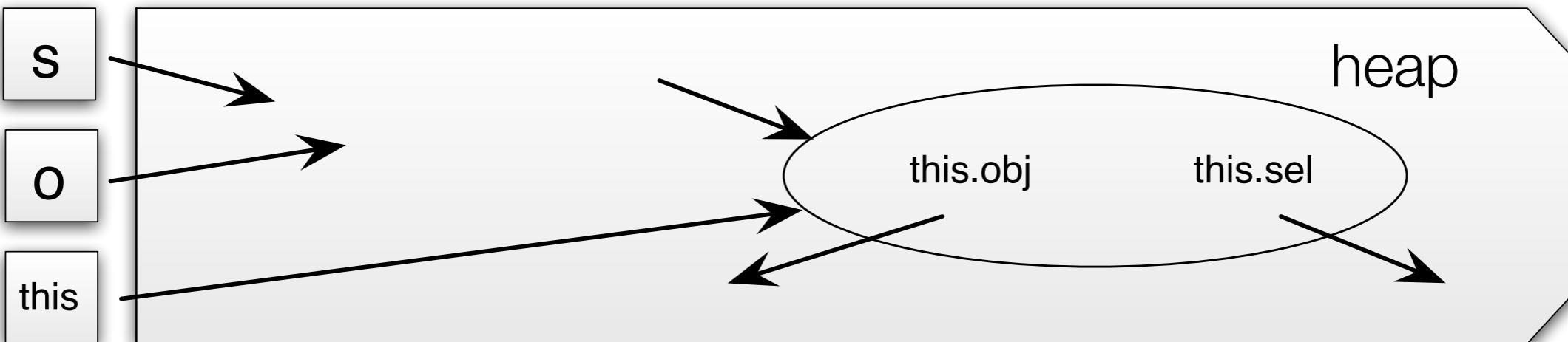
This picture captures the **fully type-consistent concrete state**

# Symbolization allows local variables to hold values inconsistent with declared types

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

$$\Gamma \quad \begin{array}{l} s : \text{Str} \\ o : \text{Obj} \mid \text{r2 } s \\ \text{this} : \text{Callback} \end{array}$$

symbolize

$$\tilde{\mathbf{E}} \quad \tilde{\Gamma}$$


# Symbolization allows local variables to hold values inconsistent with declared types

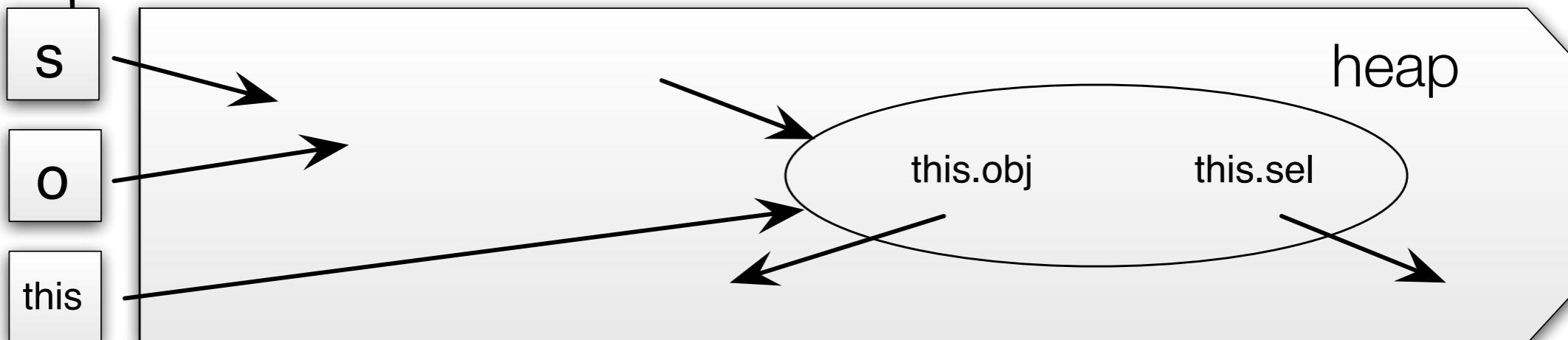
Symbolic environment  
allows, e.g., int in

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

$\Gamma$   
s : Str  
o : Obj | r2 s  
this : Callback

symbolize  $\rightarrow$

$\tilde{E}$      $\tilde{\Gamma}$



# Symbolization allows local variables to hold values inconsistent with declared types

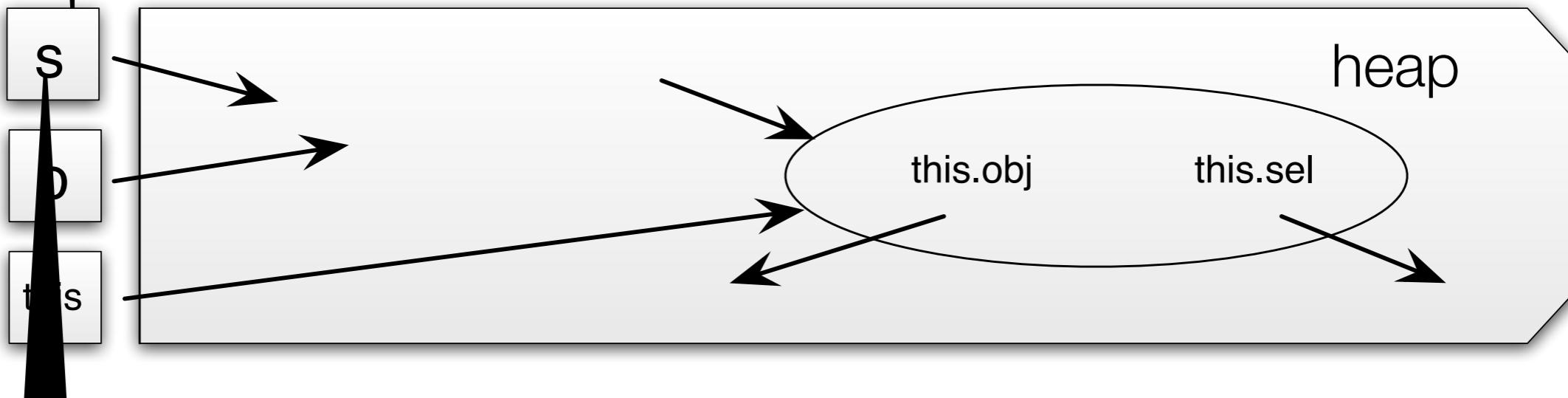
Symbolic environment  
allows, e.g., int in

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

$\Gamma$   
 $s : \text{Str}$   
 $o : \text{Obj} \mid r2 s$   
 $this : \text{Callback}$

symbolize

$\tilde{E}$      $\tilde{\Gamma}$



**Immediately type-inconsistent:** value stored  
without dereferences violates a type constraint

# Symbolization allows local variables to hold values inconsistent with declared types

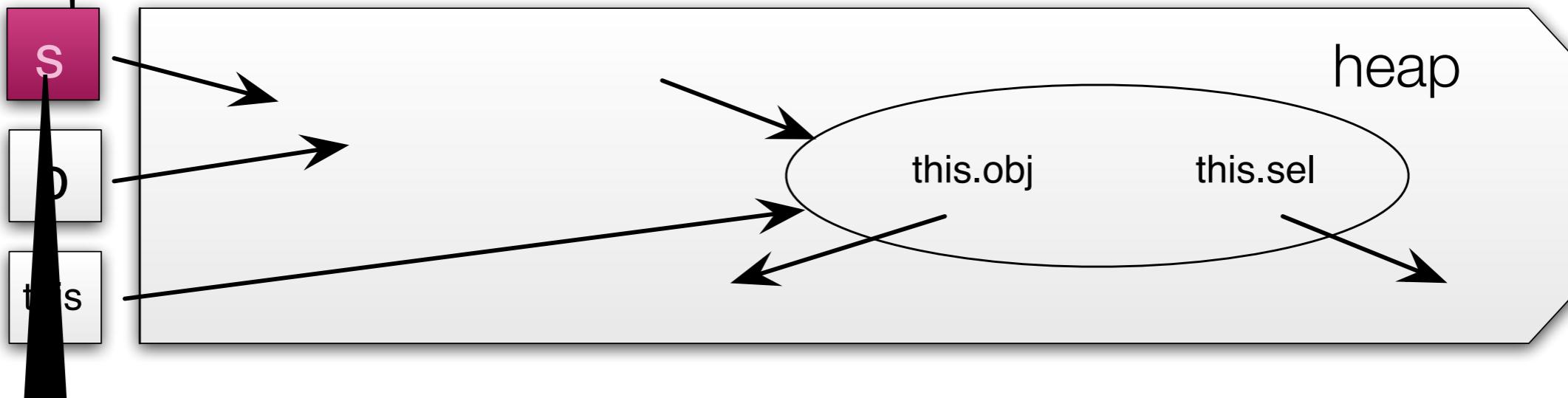
Symbolic environment  
allows, e.g., int in

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

$\Gamma$   
 $s : \text{Str}$   
 $o : \text{Obj} \mid r2 s$   
 $this : \text{Callback}$

symbolize

$\tilde{E}$      $\tilde{\Gamma}$



**Immediately type-inconsistent:** value stored  
without dereferences violates a type constraint

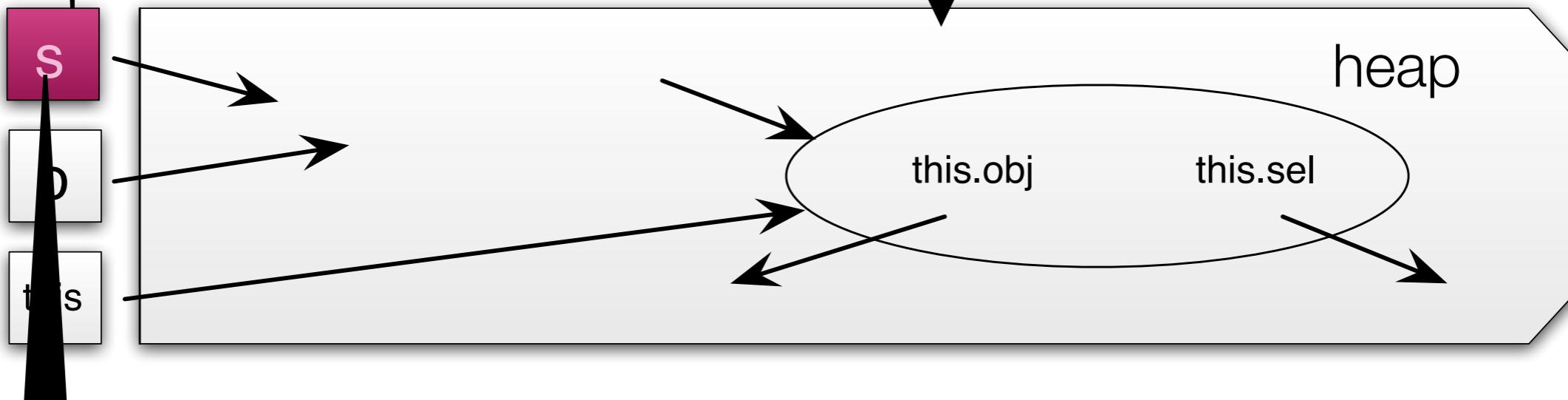
# Symbolization allows local variables to hold values inconsistent with declared types

Symbolic environment  
allows, e.g., int in

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

s : Str  
o : Obj | r2 s  
this : Callback

Grey indicates storage that is *not*  
immediately type-inconsistent



**Immediately type-inconsistent:** value stored  
without dereferences violates a type constraint

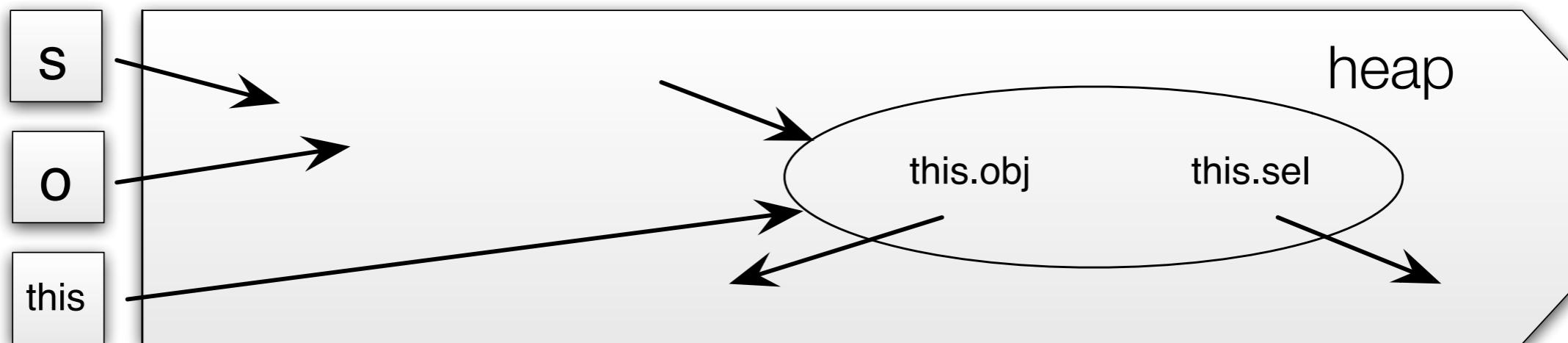
# Symbolization **unpacks local cells**, but **symbolic facts about values still constrain the heap**

Type  
environment

$$\Gamma \quad \begin{array}{l} s : \text{Str} \\ o : \text{Obj} \mid r2\ s \\ \text{this} : \text{Callback} \end{array}$$

symbolize

Symbolic  
fact map

$$\widetilde{\Gamma} \quad \begin{array}{l} \widetilde{s} : \text{Str} \\ \widetilde{o} : \text{Obj} \mid r2\ \widetilde{s} \\ \widetilde{\text{t}} : \text{Callback} \end{array}$$


# Symbolization unpacks local cells, but symbolic facts about values still constrain the heap

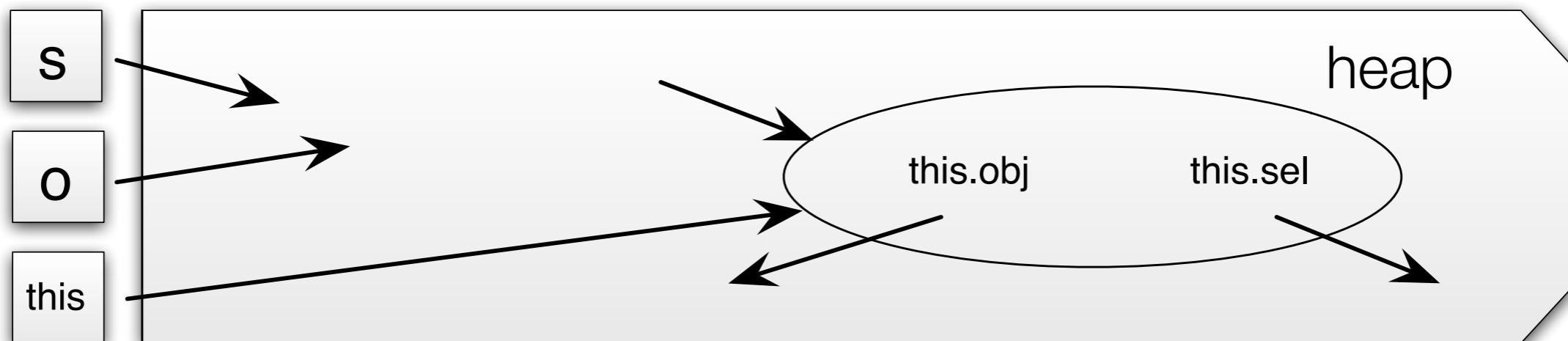
Type environment

$$\Gamma \quad \begin{array}{l} s : \text{Str} \\ o : \text{Obj} \mid r2\ s \\ \text{this} \circled{Callback} \end{array}$$

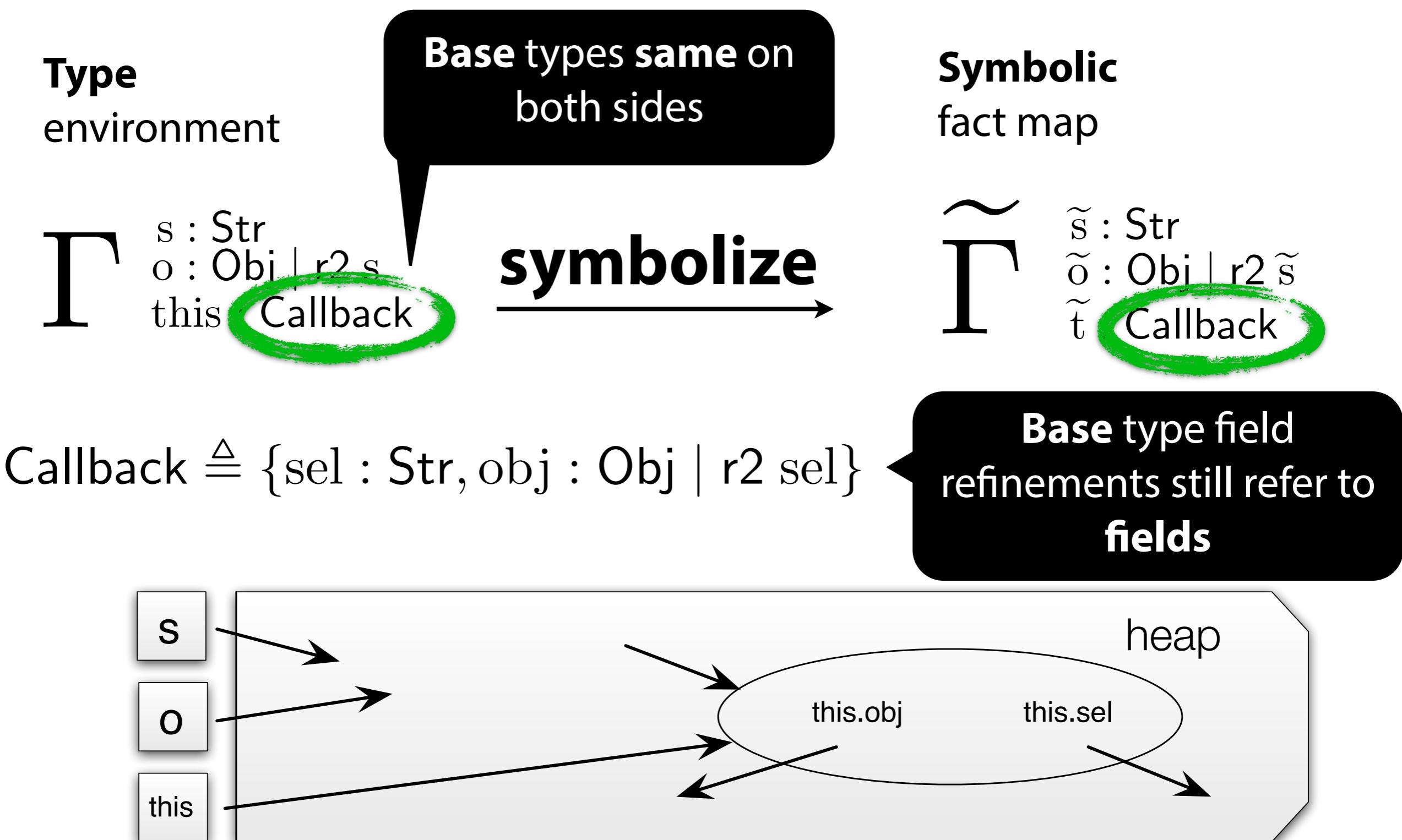
Base types **same** on both sides

**symbolize**

Symbolic fact map

$$\widetilde{\Gamma} \quad \begin{array}{l} \widetilde{s} : \text{Str} \\ \widetilde{o} : \text{Obj} \mid r2\ \widetilde{s} \\ \widetilde{\text{t}} \circled{Callback} \end{array}$$


# Symbolization unpacks local cells, but symbolic facts about values still constrain the heap



# Summarize heap locations that are *not* immediately type-inconsistent with okheap

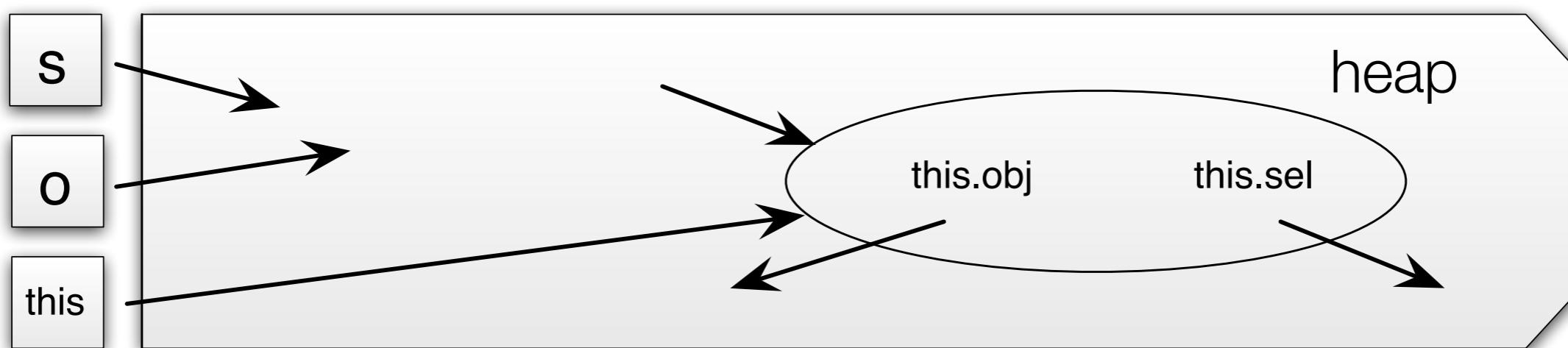
Symbolic Heap

$\tilde{H}$

okheap

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

Concrete State



# Summarize heap locations that are *not* immediately type-inconsistent with okheap

Symbolic Heap

$\tilde{H}$

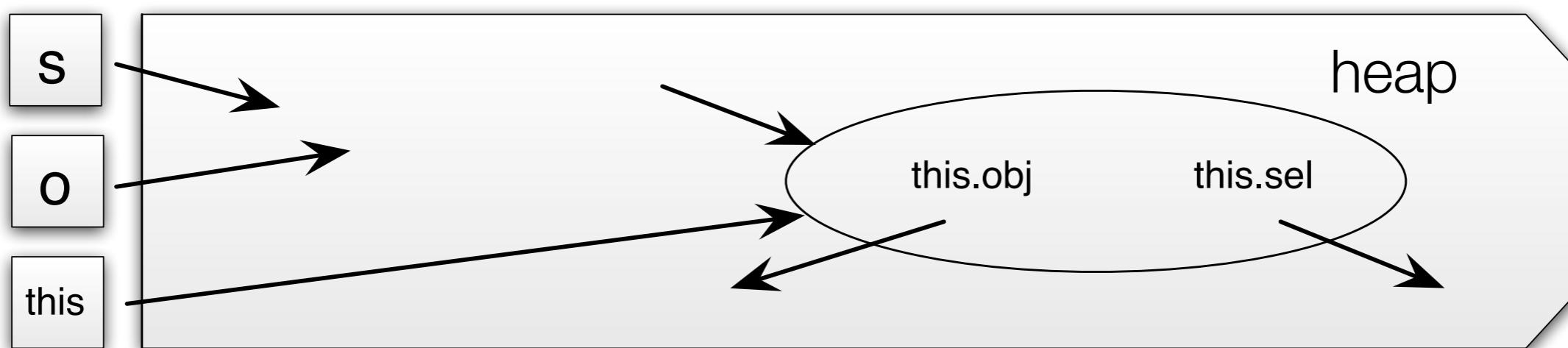
okheap



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

Formula literal: **concretization**  
includes every subheap that is *not*  
**immediately type inconsistent**

Concrete State



# Summarize heap locations that are *not* immediately type-inconsistent with okheap

Symbolic Heap

$\tilde{H}$

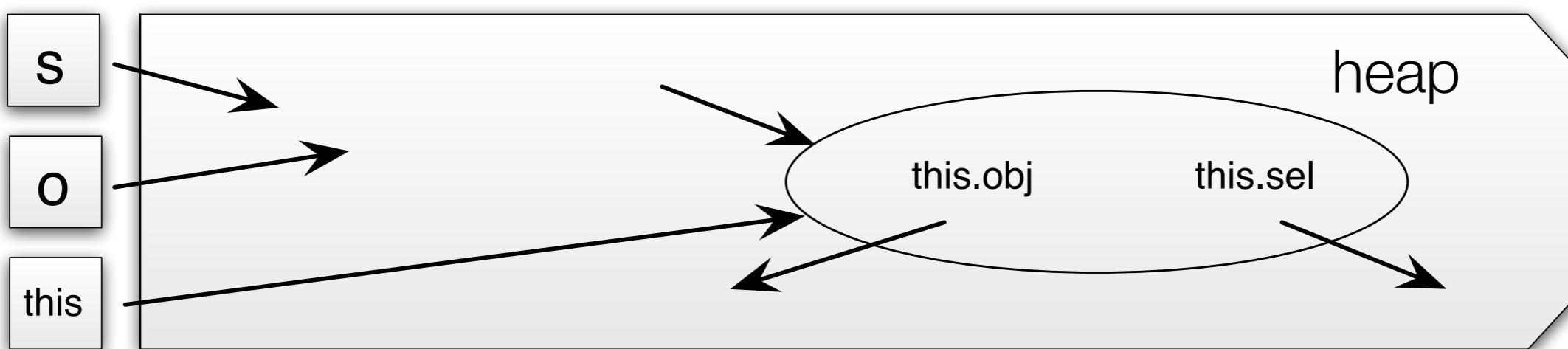
okheap

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

Describes storage **without explicitly enumerating it**

Formula literal: **concretization** includes every subheap that is *not* immediately type inconsistent

Concrete State



# Summarize heap locations that are *not* immediately type-inconsistent with okheap

Symbolic Heap

$\tilde{H}$

okheap

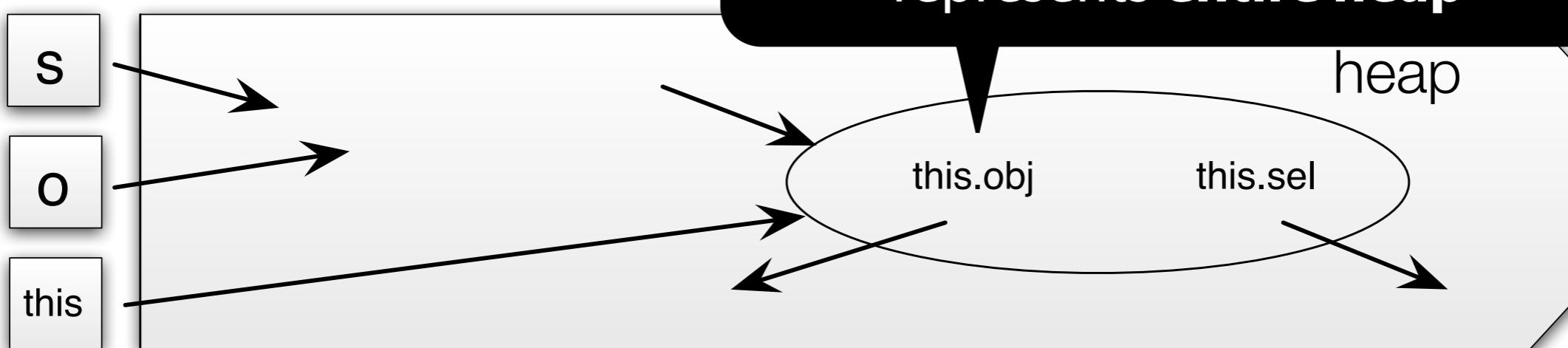
```
def update(s:Str, o:Obj | r2 s)
  this.sel = s
  this.obj = o
```

Describes storage **without explicitly enumerating it**

Concrete State

Formula literal: **concretization** includes every subheap that is *not* immediately type inconsistent

Immediately after switch, type **invariants still hold** so **okheap** represents **entire heap**

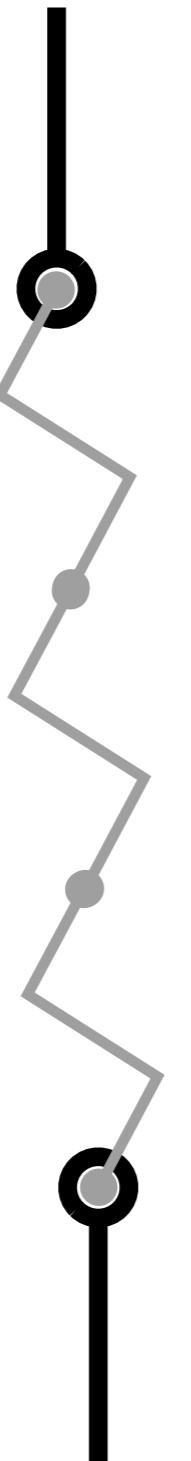


# Key Contributions

*type analysis*

*symbolic flow  
analysis*

*type analysis*



1

**Translate** type invariant into symbolic state via “**symbolization**” of type environment

2

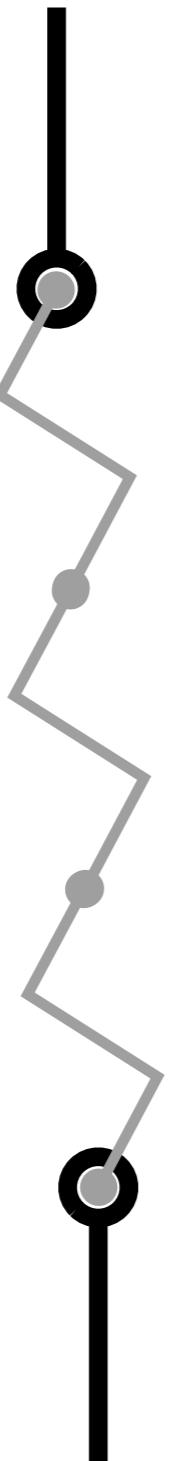
**Leverage** heap type invariant during symbolic analysis via **type-consistent materialization** and **summarization**

# Key Contributions

*type analysis*

*symbolic flow  
analysis*

*type analysis*



1

**Translate** type invariant into symbolic state via “**symbolization**” of type environment

2

**Leverage** heap type invariant during symbolic analysis via **type-consistent materialization** and **summarization**

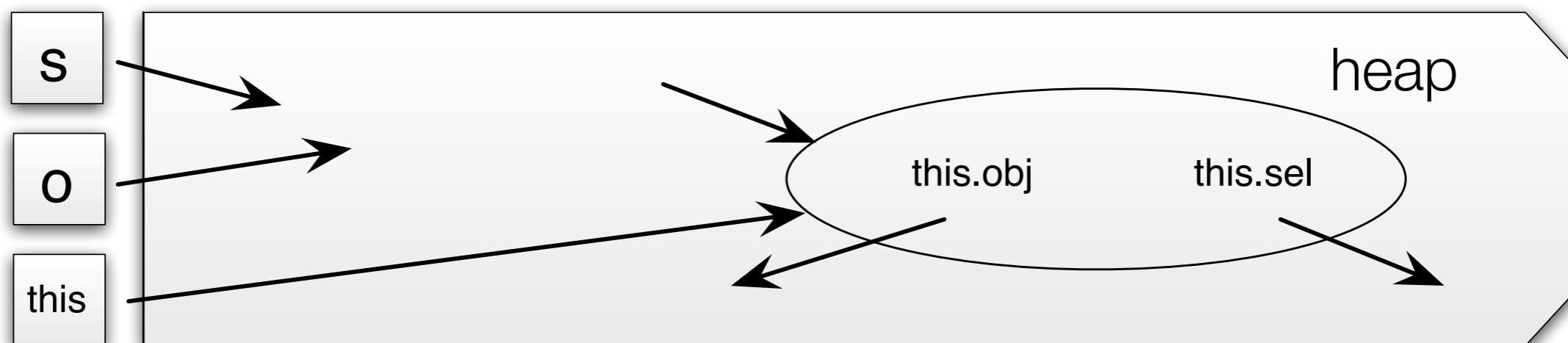
# Leverage heap type invariant via type-consistent materialization

Symbolic State

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

$\tilde{H}$  okheap

Concrete State



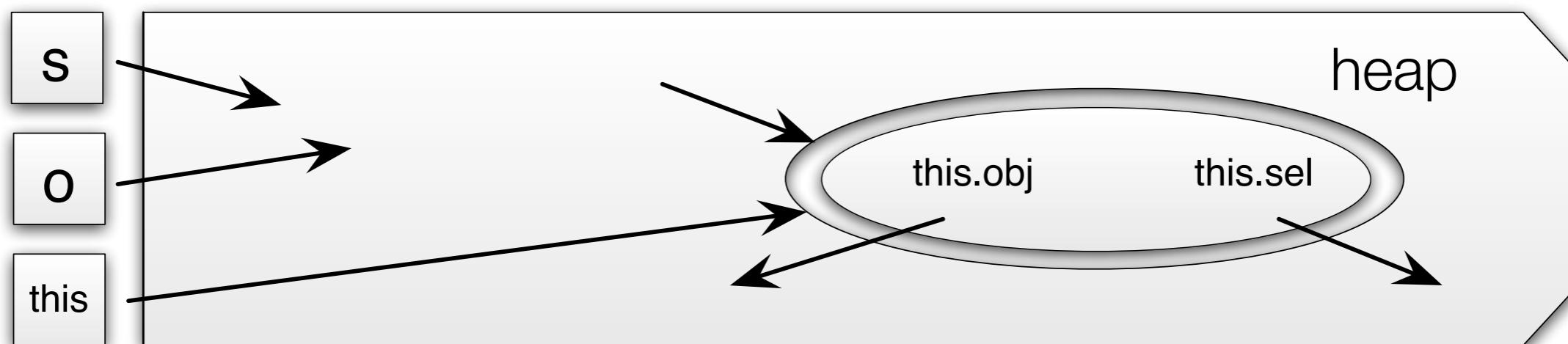
# Leverage heap type invariant via type-consistent materialization

Materialize onto standard  
**separation-logic** explicit heap

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

$\widetilde{H}$     okheap \*  $\widetilde{\text{this}} \mapsto \{\text{sel} \mapsto \widetilde{\text{sel}} * \text{obj} \mapsto \widetilde{\text{obj}}\}$

## Concrete State



# Leverage heap type invariant via type-consistent materialization

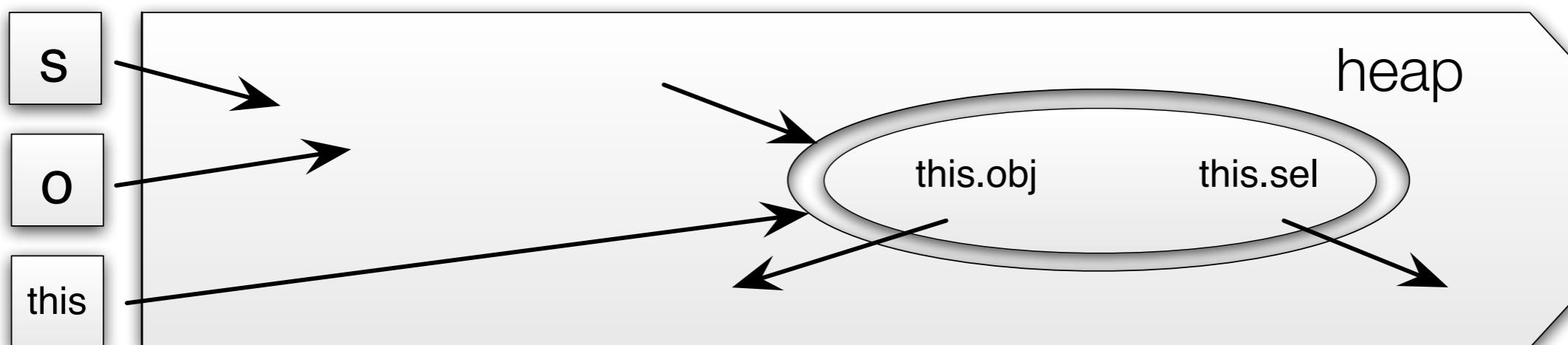
Materialize onto standard  
**separation-logic** explicit heap

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

$$\widetilde{H} \quad \text{okheap} * \widetilde{\text{this}} \mapsto \{\text{sel} \mapsto \widetilde{\text{sel}} * \text{obj} \mapsto \widetilde{\text{obj}}\}$$

**Must-alias** and **dis-alias** guarantee  
requires **case split** on materialization

## Concrete State



# Leverage heap type invariant via type-consistent materialization

Materialize onto standard  
**separation-logic** explicit heap

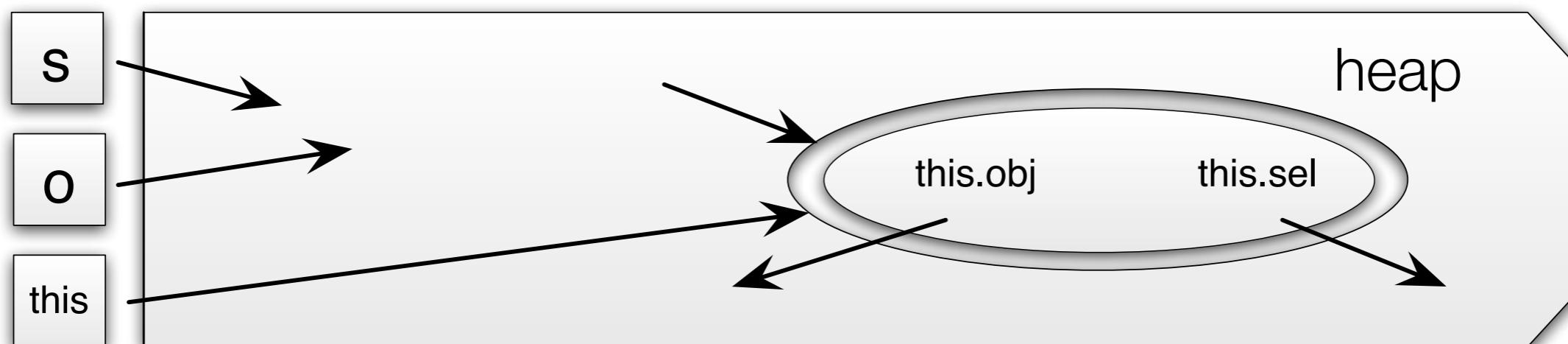
```
def update
  this.sel
  this.obj
```

Materialized storage  
guaranteed to be **not**  
**immediately type-**  
**inconsistent**

$$\widetilde{H} \quad \text{okheap} * \widetilde{\text{this}} \mapsto \{\text{sel} \mapsto \widetilde{\text{sel}} * \text{obj} \mapsto \widetilde{\text{obj}}\}$$

**Must-alias** and **dis-alias** guarantee  
requires **case split** on materialization

## Concrete State



# Leverage heap type invariant via type-consistent materialization

Materialize onto standard  
**separation-logic** explicit heap

```
def update
  this.sel
  this.obj
```

Materialized storage  
guaranteed to be **not**  
**immediately type-**  
**inconsistent**

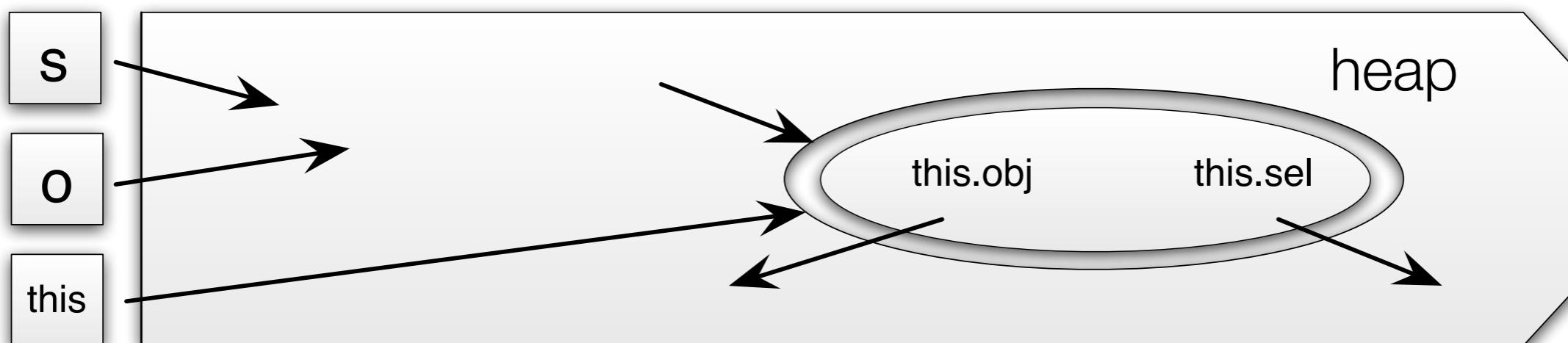
$\tilde{H}$

okheap \*  $\tilde{\text{this}} \mapsto \{\text{sel} \mapsto \tilde{\text{sel}} * \text{obj} \mapsto \tilde{\text{obj}}\}$

**Must-alias** and **dis-alias** guarantee  
requires **case split** on materialization

**Concrete State**

Value stored in **obj**  
**responds to** value  
stored in **sel**



# Leverage heap type invariant via type-consistent materialization

Materialize onto standard  
**separation-logic** explicit heap

```
def update
  this.sel
  this.obj
```

Materialized storage  
guaranteed to be **not**  
**immediately type-**  
**inconsistent**

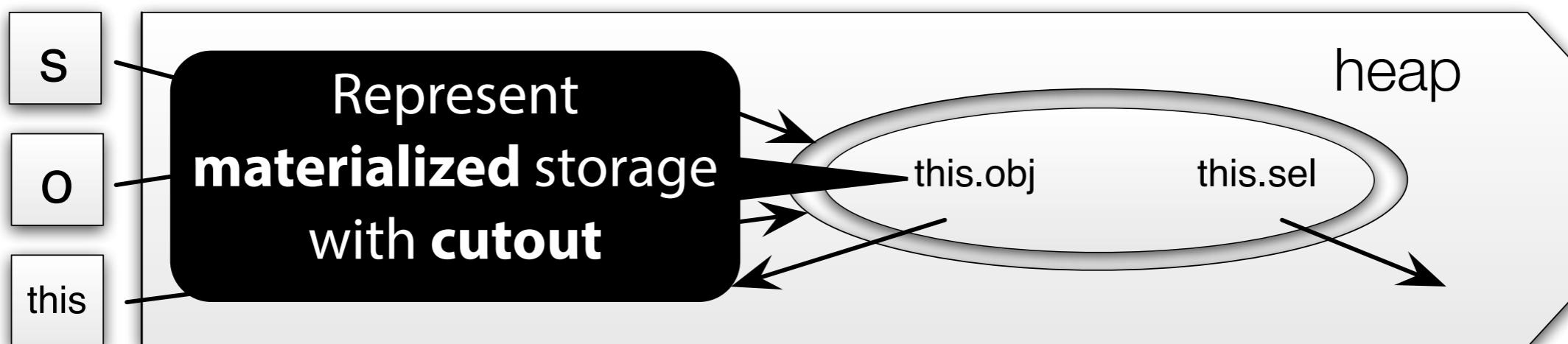
$\widetilde{H}$

okheap \*  $\widetilde{\text{this}} \mapsto \{\text{sel} \mapsto \widetilde{\text{sel}} * \text{obj} \mapsto \widetilde{\text{obj}}\}$

**Must-alias** and **dis-alias** guarantee  
requires **case split** on **materialization**

**Concrete State**

Value stored in **obj**  
**responds to** value  
stored in **sel**



# Leverage heap type invariant via type-consistent materialization

Materialize onto standard  
**separation-logic** explicit heap

```
def update
  this.sel
  this.obj
```

Materialized storage  
guaranteed to be **not**  
**immediately type-**  
**inconsistent**

$\widetilde{H}$

okheap \*  $\widetilde{\text{this}} \mapsto \{\text{sel} \mapsto \widetilde{\text{sel}} * \text{obj} \mapsto \widetilde{\text{obj}}\}$

**Must-alias** and **dis-alias** guarantee  
requires **case split** on **materialization**

**Concrete State**

Value stored in **obj**  
**responds to** value  
stored in **sel**

Analysis can **assume** that **type invariant**  
**initially holds on all materialized storage**

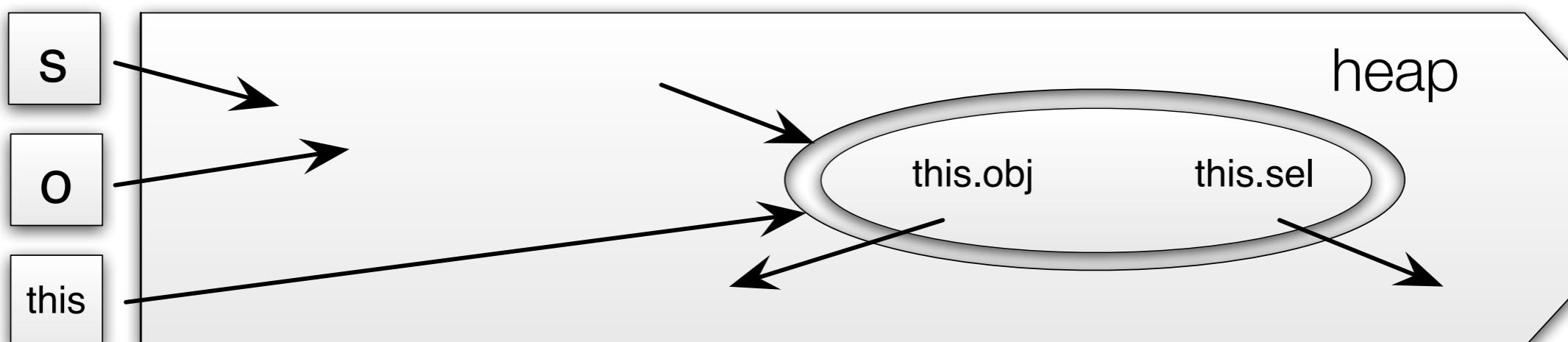
# Strong updates on materialized storage to detect invariant restoration

Symbolic State

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

$\widetilde{H}$  okheap \*  $\widetilde{\text{this}} \mapsto \{\text{sel} \mapsto \widetilde{\text{sel}} * \text{obj} \mapsto \widetilde{\text{obj}}\}$

Concrete State



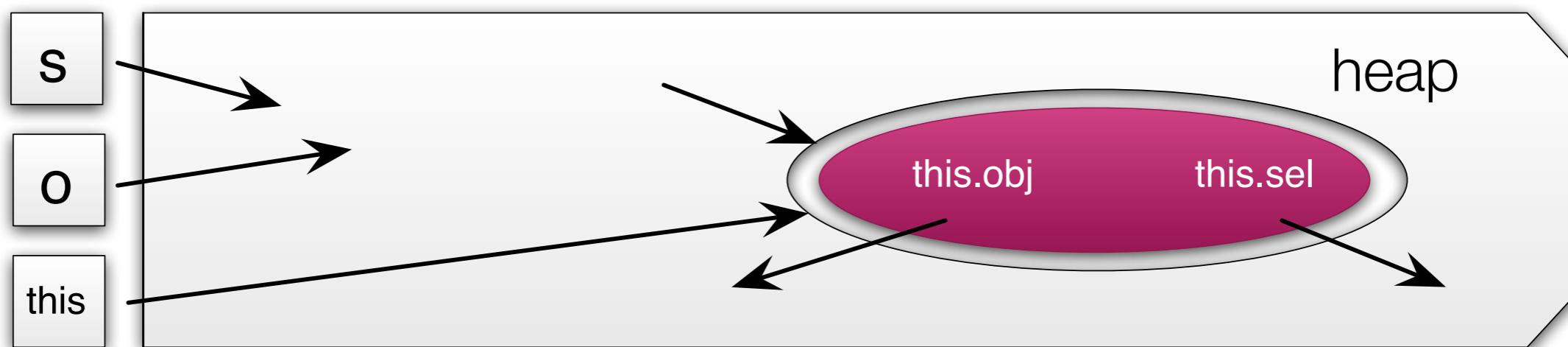
# Strong updates on materialized storage to detect invariant restoration

Symbolic State

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

$\widetilde{H}$  okheap \*  $\widetilde{\text{this}} \mapsto \{\text{sel} \mapsto \widetilde{s} * \text{obj} \mapsto \widetilde{\text{obj}}\}$

Concrete State



# Strong updates on materialized storage to detect invariant restoration

Symbolic State

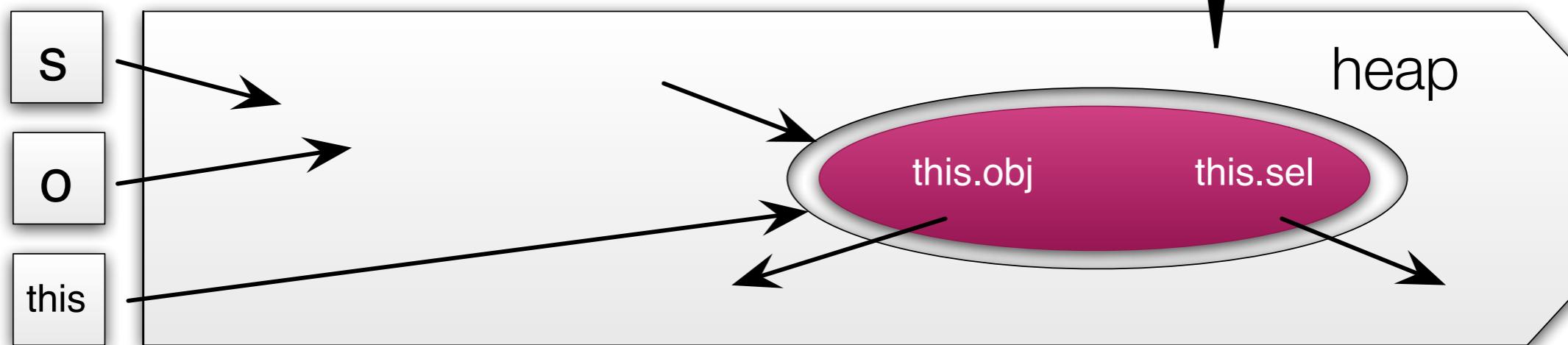
```
def update(s:Str, o:Obj | r2 s)
    this.sel = s
    this.obj = o
```

$\tilde{H}$

okheap \*  $\tilde{\text{this}} \mapsto \{\text{sel} \mapsto \tilde{s} * \text{obj} \mapsto \tilde{\text{obj}}\}$

Concrete State

Type invariant violated



# Strong updates on materialized storage to detect invariant restoration

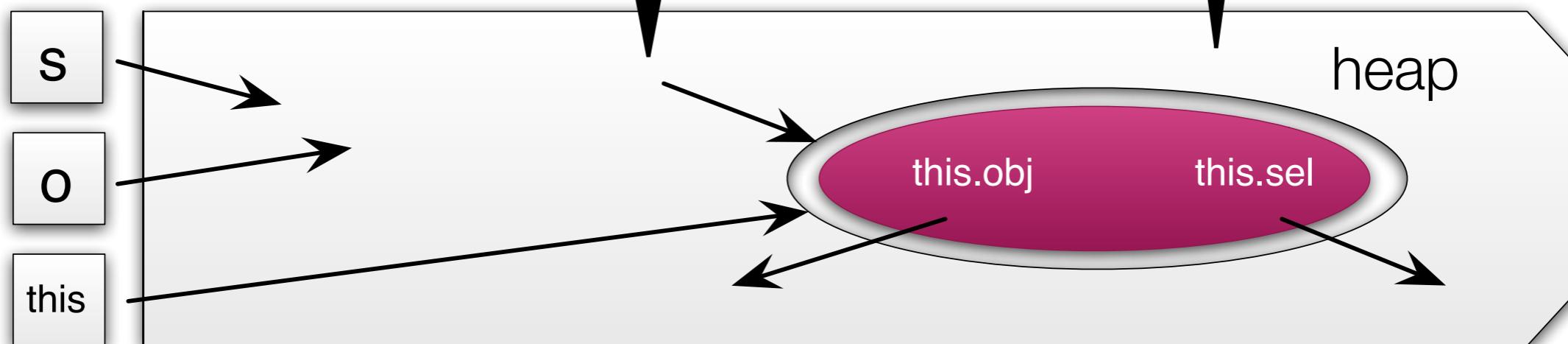
Symbolic State

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

$\widetilde{H}$     okheap \*  $\widetilde{\text{this}} \mapsto \{\text{sel} \mapsto \widetilde{s} * \text{obj} \mapsto \widetilde{\text{obj}}\}$

**Surprising:** can **soundly** permit **pointers** in and **out** of the region that is **not** immediately type-consistent

Type invariant violated



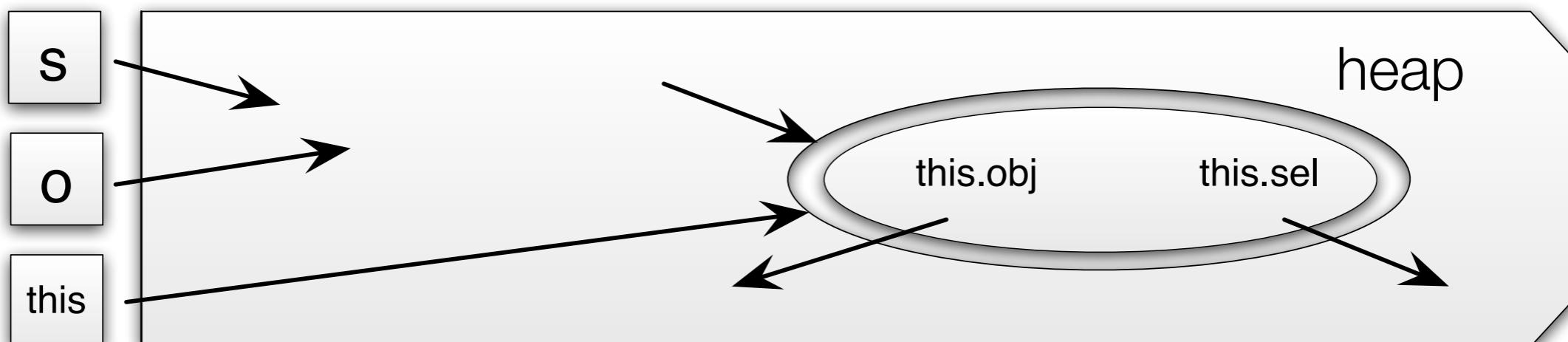
# Strong updates on materialized storage to detect invariant restoration

Symbolic State

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

$$\widetilde{H} \quad \text{okheap} * \widetilde{\text{this}} \mapsto \{\text{sel} \mapsto \widetilde{s} * \text{obj} \mapsto \widetilde{o}\}$$

Concrete State



# Strong updates on materialized storage to detect invariant restoration

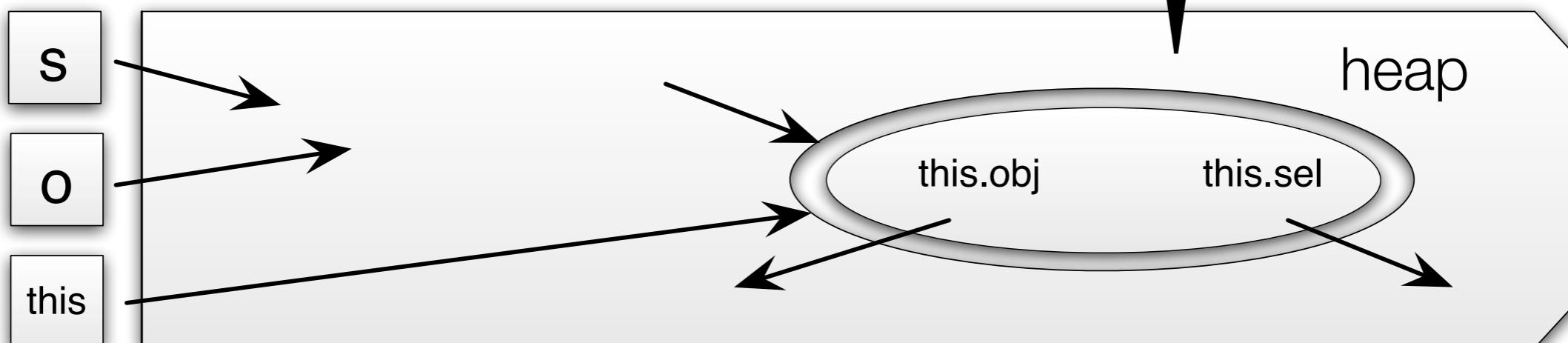
Symbolic State

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

$\tilde{H}$

okheap \*  $\tilde{\text{this}} \mapsto \{\text{sel} \mapsto \tilde{s} * \text{obj} \mapsto \tilde{o}\}$

Concrete State



No longer immediately  
type-inconsistent

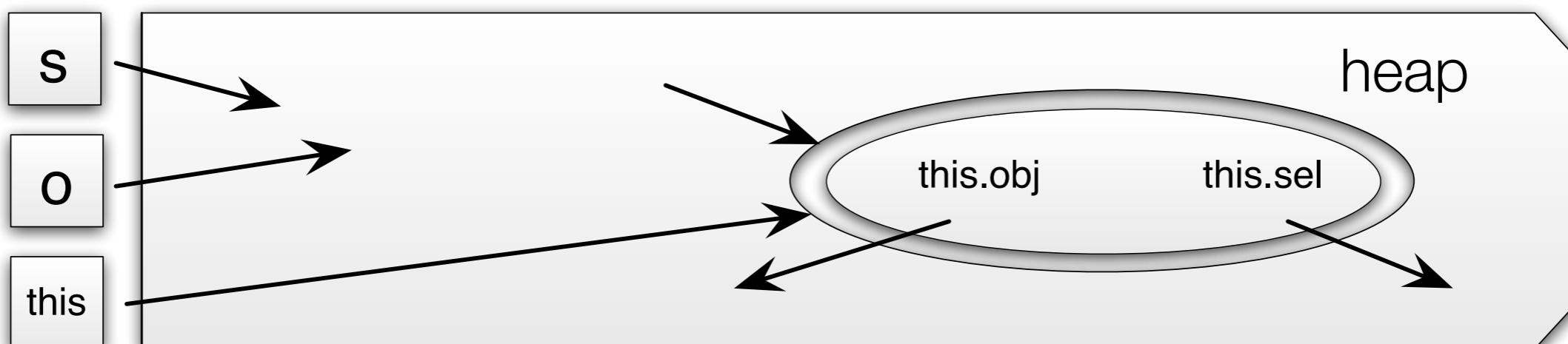
Safely **summarize** storage that is not immediately type inconsistent

**Symbolic State**

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

$\widetilde{H}$  okheap \*  $\widetilde{\text{this}} \mapsto \{\text{sel} \mapsto \widetilde{s} * \text{obj} \mapsto \widetilde{o}\}$

**Concrete State**



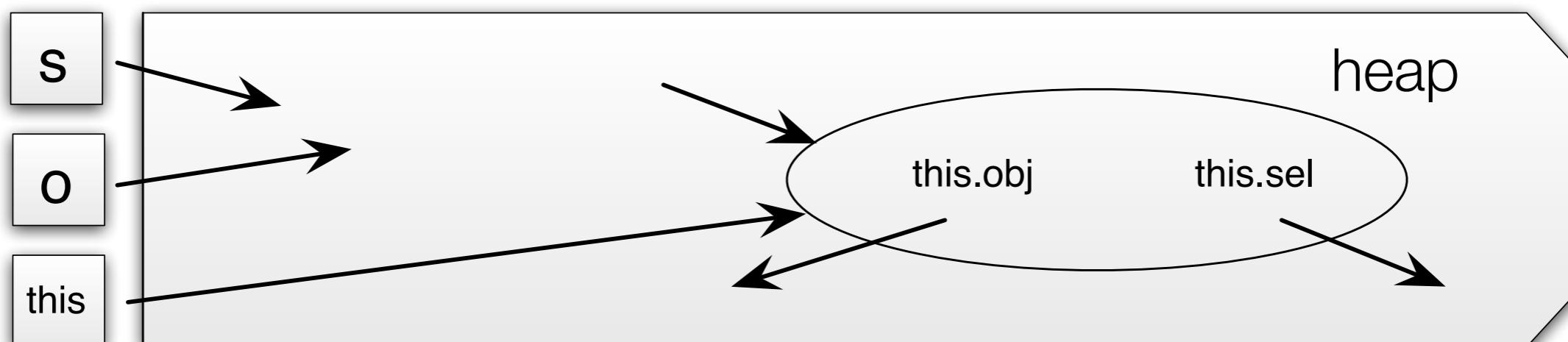
# Safely **summarize** storage that is not immediately type inconsistent

**Symbolic State**

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

$\tilde{H}$  okheap

## Concrete State



# Safely **summarize** storage that is not immediately type inconsistent

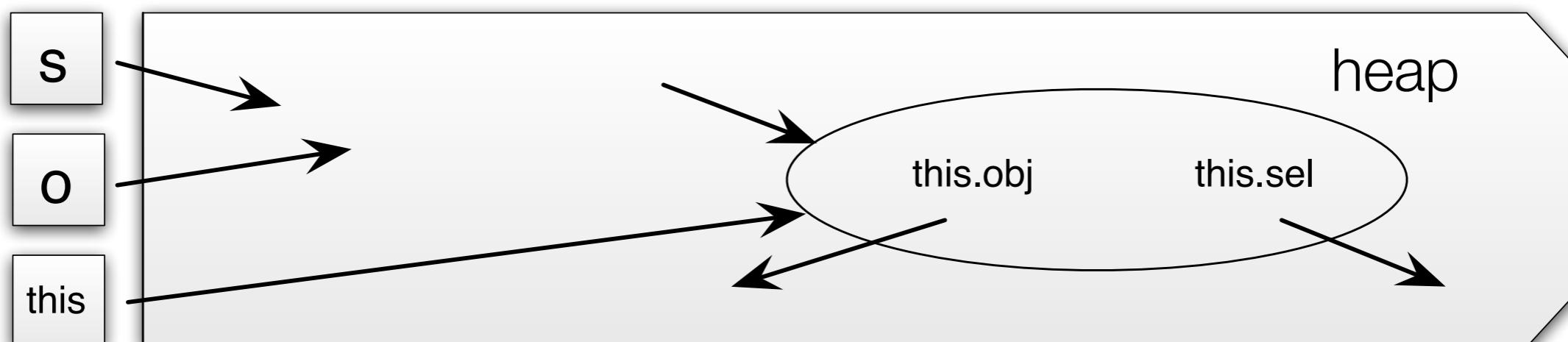
**Symbolic State**

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

$\tilde{H}$  okheap

Only need to reason **precisely** about  
**part of heap where invariant broken**,  
so helps **manage alias explosion**

## Concrete State



# Safely **summarize** storage that is not immediately type inconsistent

**Symbolic State**

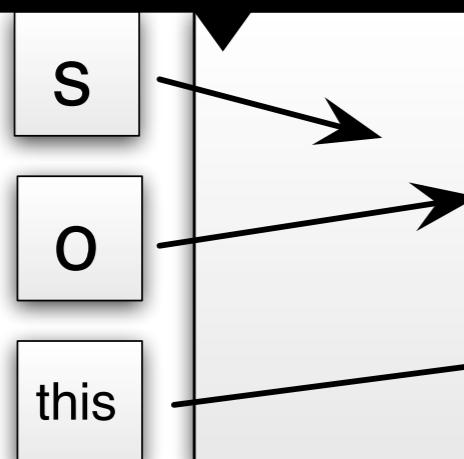
```
def update(s:Str, o:Obj | r2 s)
    this.sel = s
    this.obj = o
```

$\tilde{H}$

okheap

**Entire heap** is type consistent so safe to **return to type checking**

Only need to reason **precisely** about **part of heap where invariant broken**, so helps **manage alias explosion**



# Safely **summarize** storage that is not immediately type inconsistent

**Symbolic State**

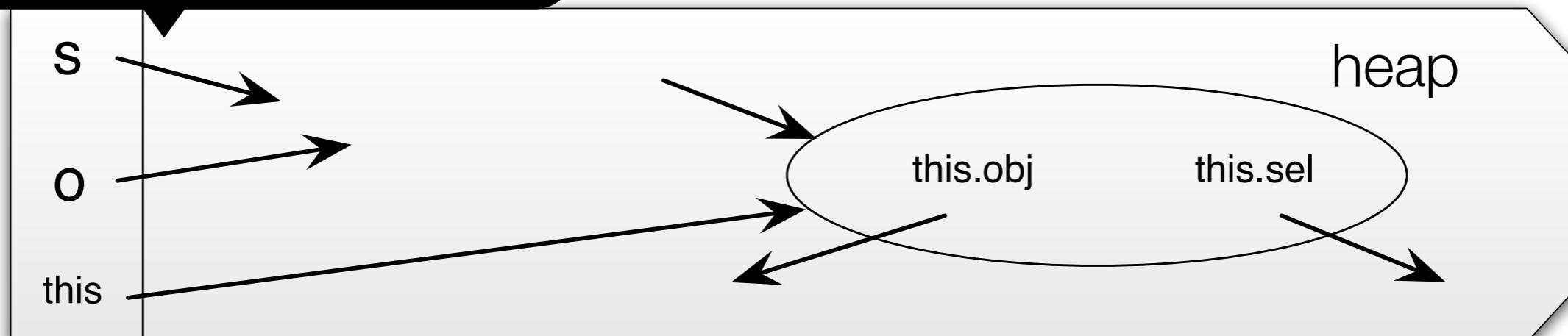
```
def update(s:Str, o:Obj | r2 s)
    this.sel = s
    this.obj = o
```

$\tilde{H}$

okheap

Only need to reason **precisely** about  
**part of heap where invariant broken**,  
so helps **manage alias explosion**

**Entire heap** is type  
consistent so safe to  
**return to type checking**

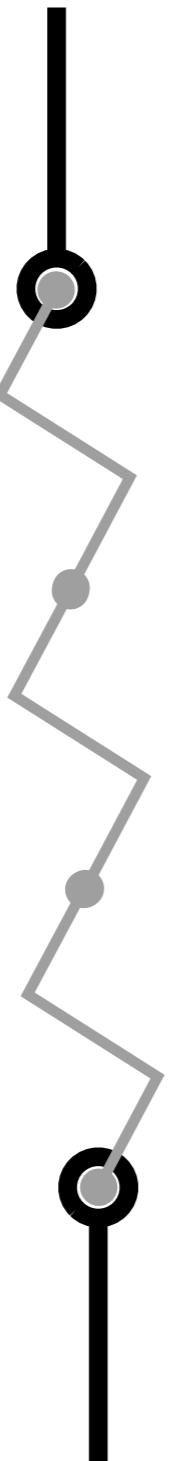


# Key Contributions

*type analysis*

*symbolic flow  
analysis*

*type analysis*



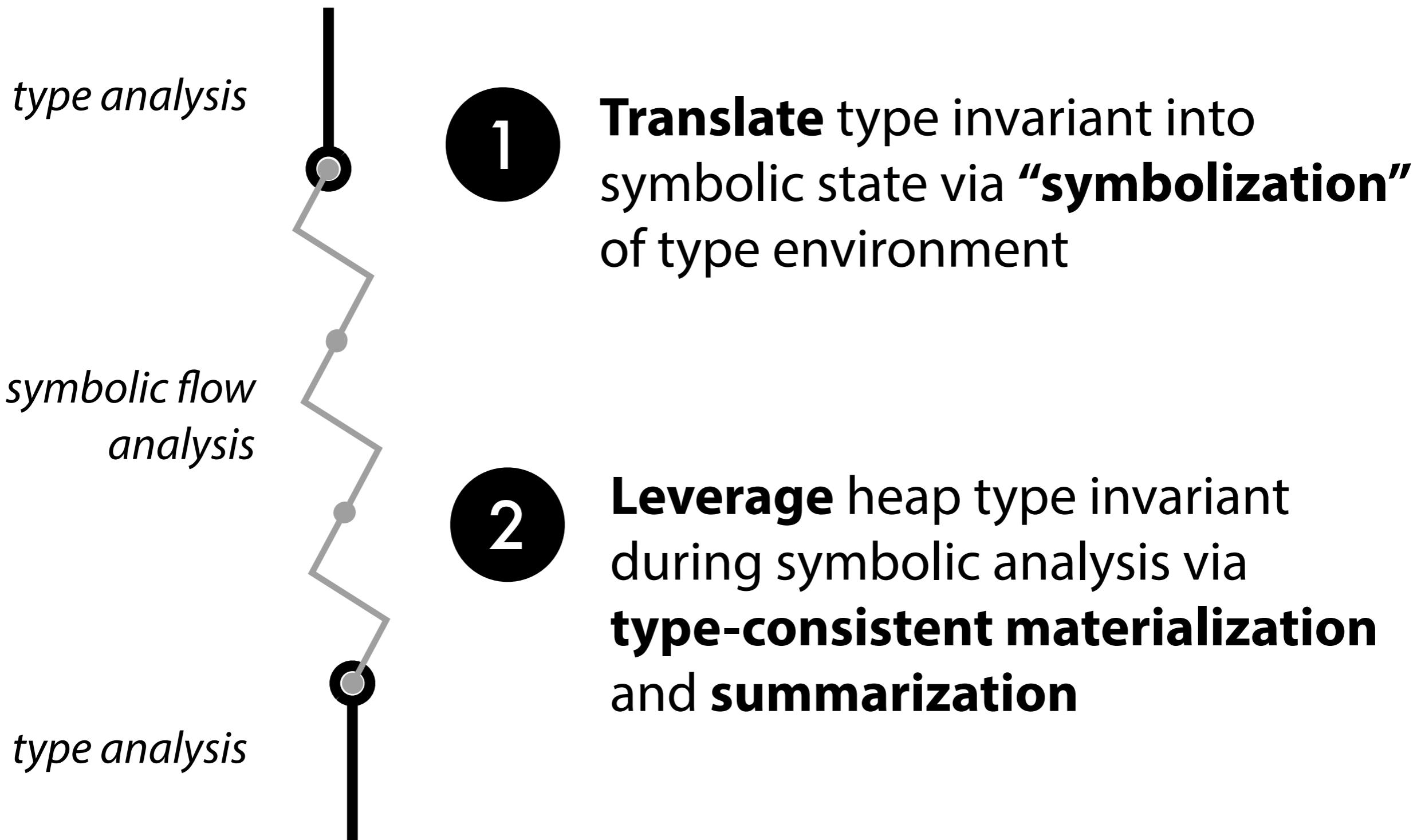
1

**Translate** type invariant into symbolic state via “**symbolization**” of type environment

2

**Leverage** heap type invariant during symbolic analysis via **type-consistent materialization** and **summarization**

# Key Contributions

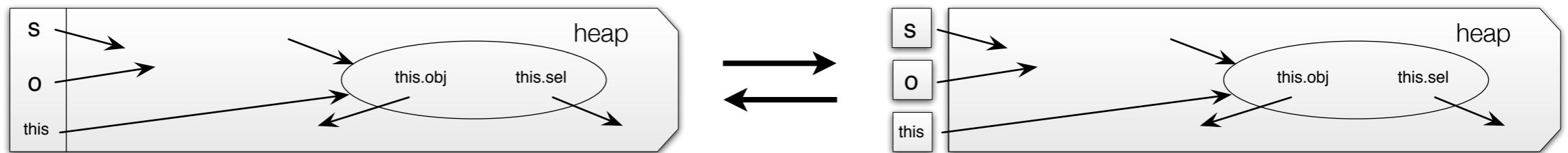


# Fissile Type Analysis is **sound**

# Fissile Type Analysis is **sound**

**Theorem (Soundness of Handoff).**

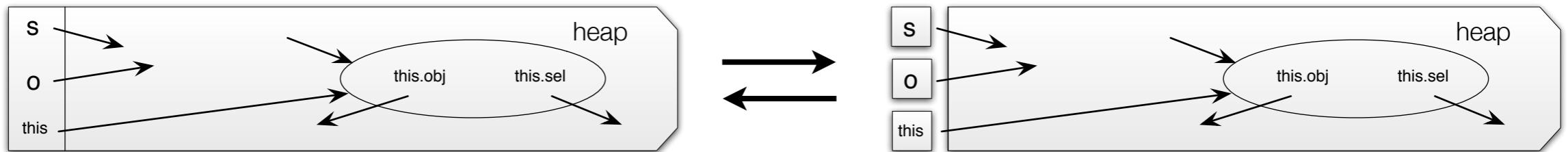
The **entire state** is **type-consistent** iff all locations are **not immediately type-inconsistent**.



# Fissile Type Analysis is **sound**

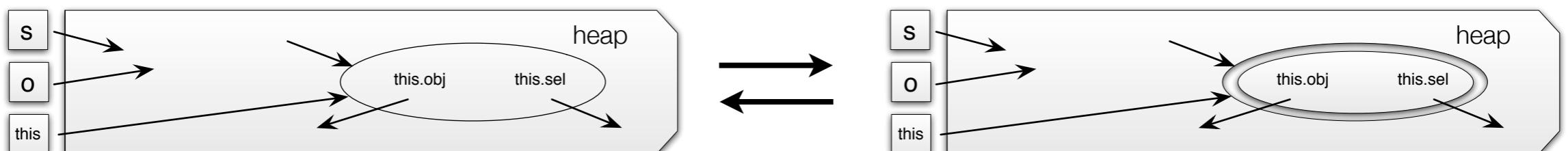
**Theorem (Soundness of Handoff).**

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



**Theorem (Soundness of Materialization/Summarization).**

Storage that is **not immediately type-inconsistent** can be safely *materialized* and *summarized* into **okheap**.



# Evaluation

**Analysis mechanics:** How often is **symbolic reasoning** required?

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

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

# 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**

# Analysis mechanics

<b>benchmark</b>	<b>size</b>		
	<b>(loc)</b>	<b>symbolic sections</b>	<b>maximum materializations</b>
OAUTH	1248	7	1
SCRECODER	2716	2	2
ZIPKIT	3301	0	0
SPARKLE	5289	3	1
ASIHTTPREQUEST	14620	59	2
OMNIFRAMEWORKS	160769	7	1
VIENNA	37327	28	2
SKIM	60211	0	0
ADIUM	176629	16	1
<b>combined</b>	<b>461080</b>	<b>125</b>	<b>2</b>

# Analysis mechanics

benchmark	size (loc)	symbolic sections	maximum materializations
OAUTH	1248	7	1
	2716	2	2
	3301	0	0
	5289	3	1
ASIHTTPRequest	14620	59	2
OMNIFRAMEWORKS	160769	7	1
VIENNA	37327	28	2
SKIM	60211	0	0
ADIUM	176629	16	1
<b>combined</b>	<b>461080</b>	<b>125</b>	<b>2</b>

Number of **successful**  
switches to **symbolic**  
**analysis** and back

# Analysis mechanics

benchmark	size (loc)	symbolic sections	maximum materializations
OAUTH	1248	7	1
	2716	2	2
	3301	0	0
	5289	3	1
ASIHTTPRequest	14620	59	2
OMNIFRAMEWORKS	160769	7	1
VIENNA	37327	28	2
SKIM	60211	0	0
ADIUM	176629	16	1
combined	461080	125	2

Number of **successful**  
switches to **symbolic**  
**analysis** and back

A **significant** number of **switches**:  
Approach successfully handles when **developers break**  
and **restore** global **invariants**

# Analysis mechanics

benchmark	size (loc)	symbolic sections	maximum materializations
Maximum number of <b>simultaneous</b> materialized storage locations		7	1
		2	2
		0	0
		3	1
		59	2
OMNIFRAMEWORKS	160769	7	1
VIENNA	37327	28	2
SKIM	60211	0	0
ADIUM	176629	16	1
<b>combined</b>	<b>461080</b>	<b>125</b>	<b>2</b>

A **significant** number of **switches**:  
Approach successfully handles when **developers break**  
and **restore** global **invariants**

# Analysis mechanics

benchmark	size (loc)	symbolic sections	maximum materializations
Maximum number of <b>simultaneous</b> materialized storage locations		7	1
		2	2
		0	0
		3	1
		59	2
OMNIFRAMEWORKS	160769	7	1
VIENNA	37327	28	2
SKIM	60211	0	0
ADIUM	176629	16	1
<b>combined</b>	<b>461080</b>	<b>125</b>	<b>2</b>

A **significant** number of **switches**:

Approach successfully handles when **developers break** and **restore** global **invariants**

At most **2 simultaneous materializations**:

**Aliasing case splits will not blow up**

# Analysis mechanics

	<b>size</b>		
<b>benchmark</b>	<b>(loc)</b>	<b>symbolic sections</b>	<b>maximum materializations</b>
OAUTH	1248	7	1
SCRECODER	2716	2	2
ZIPKIT	3301	0	0
SPARKLE	5289	3	1
ASIHTTPREQUEST	14620	59	2
OMNIFRAMEWORKS	160769	7	1
VIENNA	37327	28	2
SKIM	60211	0	0
ADIUM	176629	16	1
<b>combined</b>	<b>461080</b>	<b>125</b>	<b>2</b>

A **significant** number of **switches**:

Approach successfully handles when **developers break** and **restore** global **invariants**

At most **2 simultaneous materializations**:

**Aliasing case splits will not blow up**

# Analysis mechanics

benchmark	size (loc)	symbolic sections	maximum materializations
OAUTH	1248	7	1
SCRECODER	2716	2	2
ZIPKIT	3301	0	0
SPARKLE	5289	3	1
ASIHTTPREQUEST	14620	59	2
OMNIFRAMEWORKS	160769	7	1
VIENNA	37327	28	2
SKIM	60211	0	0
ADILIM	176629	16	1
		125	2

Approaches limited to **one-at-a-time**  
**materialization not sufficient**

atches:

Approach successfully handles when **developers break**  
and **restore** global **invariants**

At most **2 simultaneous materializations**:  
**Aliasing case splits will not blow up**

# Precision

<b>benchmark</b>	<b>size</b>		<b>false alarms</b>	
	<b>(loc)</b>	<b>reflective call sites</b>	<b>flow-insensitive</b>	<b>almost-everywhere</b>
OAUTH	1248	7	7	2 (-71%)
SCRECODER	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%)
<b>combined</b>	<b>461080</b>	<b>1327</b>	<b>334</b>	<b>238 (-29%)</b>

# Precision

<b>benchmark</b>	<b>size</b>		<b>false alarms</b>	
	<b>(loc)</b>	<b>reflective call sites</b>	<b>flow-insensitive</b>	<b>almost-everywhere</b>
OAUTH	1248	7	7	2 (-71%)
SCRECODER	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%)
<b>combined</b>	<b>461080</b>	<b>1327</b>	<b>334</b>	<b>238 (-29%)</b>

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

# Precision

<b>benchmark</b>	<b>size</b>		<b>false alarms</b>	
	<b>(loc)</b>	<b>reflective call sites</b>	<b>flow-insensitive</b>	<b>almost-everywhere</b>
OAUTH	1248	7	7	2 (-71%)
SCRECODER	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%)
<b>combined</b>	<b>461080</b>	<b>1327</b>	<b>334</b>	<b>238 (-29%)</b>

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

# Precision

<b>benchmark</b>	<b>size</b>		<b>false alarms</b>	
	<b>(loc)</b>	<b>reflective call sites</b>	<b>flow-insensitive</b>	<b>almost-everywhere</b>
OAUTH	1248	7	7	2 (-71%)
SCRECODER	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%)
<b>combined</b>	<b>461080</b>	<b>1327</b>	<b>334</b>	<b>238 (-29%)</b>

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

**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 switching

**Almost everywhere** techniques show **29% improvement** in false alarms

# Cost: Analysis time

<b>benchmark</b>	<b>size (loc)</b>	<b>analysis time</b>	
		<b>Time</b>	<b>Rate (kloc/s)</b>
OAUTH	1248	0.24s	5.3
SCRECODER	2716	0.28s	10.8
ZIPKIT	3301	0.10s	33.0
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
<b>combined</b>	<b>461080</b>	<b>20.09s</b>	<b>23.0</b>

# Cost: Analysis time

benchmark	size (loc)	analysis time Time	Rate (kloc/s)
OAUTH	1248	0.24s	5.3
SCRECODER	2713	0.28s	10.8
ZIPKIT	3301	0.10s	33.0
SCRECODER	5289	0.67s	7.9
SKIM	14620	0.50s	27.2
SKIM	160769	4.25s	37.8
SKIM	37327	2.79s	13.4
SKIM	60211	2.49s	24.1
ADIUM	176629	8.79s	20.1
<b>combined</b>	<b>461080</b>	<b>20.09s</b>	<b>23.0</b>

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

# Cost: Analysis time

	size (loc)	analysis time Time	Rate (kloc/s)
Does not include system headers	1248	0.24s	5.3
Includes analysis time but <b>not parsing, base type checking</b>	2713	0.28s	10.8
	3301	0.10s	33.0
	5289	0.67s	7.9
	14620	0.50s	27.2
	160769	4.25s	37.8
	37327	2.79s	13.4
	60211	2.49s	24.1
	ADIUM	8.79s	20.1
<b>combined</b>	461080	20.09s	23.0

# Cost: Analysis time

benchmark	size (loc)	analysis time	
		Time	Rate (kloc/s)
OAUTH	1248	0.24s	5.3
SCRECODER	2716	0.28s	10.8
ZIPKIT	3301	0.10s	33.0
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
<b>combined</b>	<b>461080</b>	<b>20.09s</b>	<b>23.0</b>

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

# Cost: Analysis time

benchmark	size (loc)	analysis time	
		Time	Rate (kloc/s)
OAUTH	1248	0.24s	5.3
SCRECODER	2716	0.28s	10.8
ZIPKIT	3301	0.10s	33.0
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
<b>combined</b>	<b>461080</b>	<b>20.09s</b>	<b>23.0</b>

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

Interactive speeds

# Cost: Analysis time

<b>benchmark</b>	<b>size (loc)</b>	<b>analysis time</b>	
		<b>Time</b>	<b>Rate (kloc/s)</b>
OAUTH	1248	0.24s	5.3
SCRECODER	2716	0.28s	10.8
ZIPKIT	3301	0.10s	33.0
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
<b>combined</b>	<b>461080</b>	<b>20.09s</b>	<b>23.0</b>

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

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

# Cost: Analysis time

<b>benchmark</b>	<b>size (loc)</b>	<b>analysis time</b>	
		<b>Time</b>	<b>Rate (kloc/s)</b>
OAUTH	1248	0.24s	5.3
SCRECODER	2716	0.28s	10.8
ZIPKIT	3301	0.10s	33.0
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
<b>combined</b>	<b>461080</b>	<b>20.09s</b>	<b>23.0</b>

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

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

# Summary

- Check **almost everywhere** heap invariants with **intertwined type** and **symbolic flow analysis**
  - **Translate** type environment into symbolic state with **symbolization**
  - **Leverage** heap type invariant during symbolic analysis via **type-consistent materialization** and **summarization**
- Approach is **very fast** and **scales to large programs**

**Fissile Type Analysis** yields significant  
precision improvement at little cost  
in performance

**Fissile Type Analysis** yields significant  
precision improvement at little cost  
in performance

**Why?**

**Fissile Type Analysis** yields significant  
precision improvement at little cost  
in performance

**Why?**

Because **almost-everywhere**  
invariants hold **almost everywhere**