Devin Coughlin
Bor-Yuh Evan Chang

Fissile Type Analysis: Modular Checking of Almost-Everywhere Invariants

University of Colorado Boulder

Goal: Verify Reflective Method Call with Dependent Types

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

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

Reflective call dispatches to method with **name** stored in `sel` on object stored in `obj`

Dependent type specifies **required relationship**: `obj` must r2 ("**respond to**") method named in `sel`

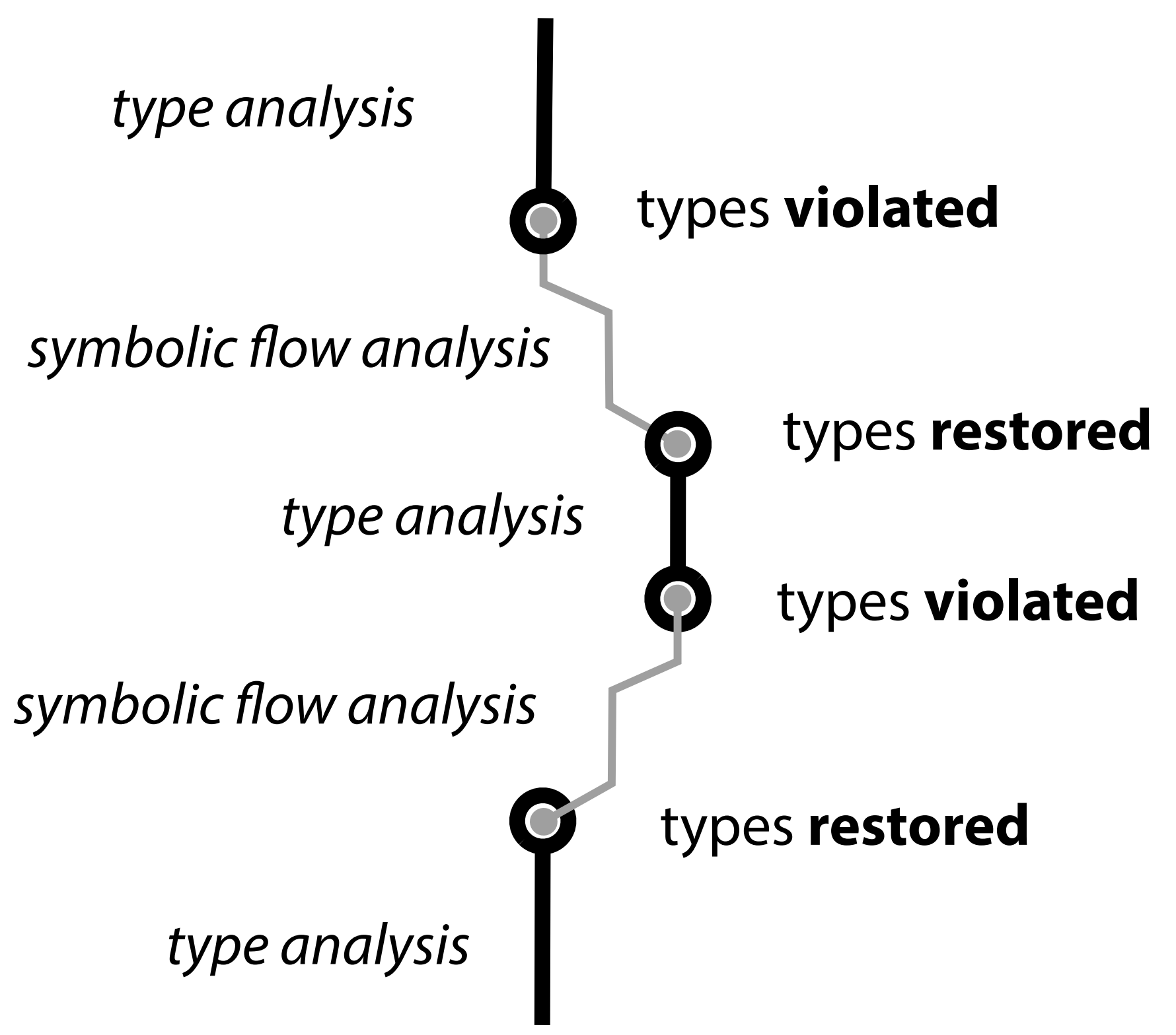
Problem: Imperative Updates Violate Flow-Insensitive Types

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

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

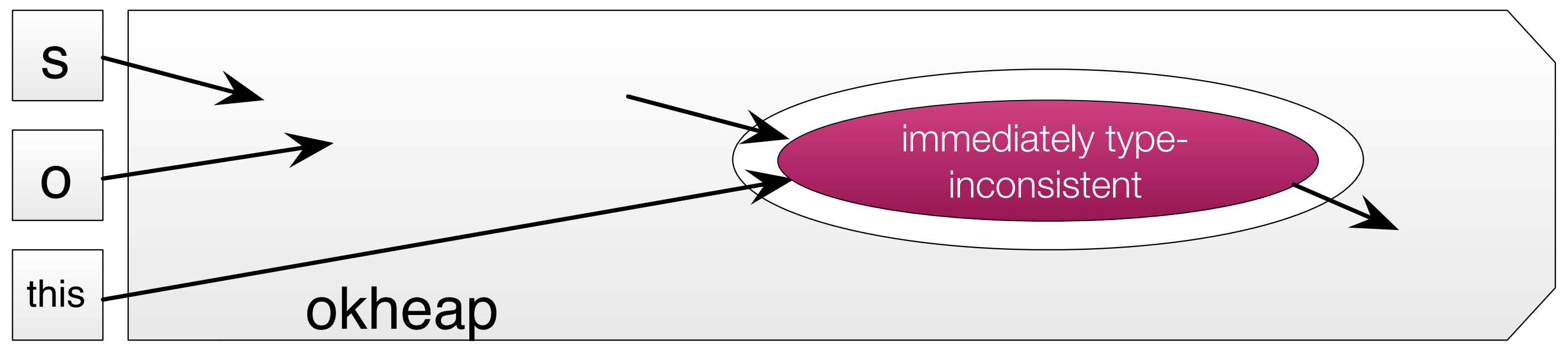
Type error is **false alarm:** next instruction **restores** type

Approach: Intertwined Type and Flow Analysis



Analysis will be effective if types hold **almost everywhere** — that is, if programmers **violate** the flow-insensitive **typing** only **briefly.**

Split Heap Into Two Regions In Symbolic Analysis



Almost type-consistent okheap: field values at worst **only transitively** inconsistent with declared types

Immediately type-inconsistent heap: field values may be inconsistent with declared types

Crucially allow **pointers between** two regions

Fissile Type Analysis

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

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

Split type environment into facts about **values** and **initially type-consistent** symbolic **memory**.

$$\widetilde{E}\ \begin{array}{l} \mathrm{s} : \tilde{\mathrm{s}} \\ \mathrm{o} : \tilde{\mathrm{o}} \\ \mathrm{this} : \tilde{\mathrm{t}} \end{array} \quad \widetilde{\Gamma}\ \begin{array}{l} \tilde{\mathrm{s}} : \mathsf{Str} \\ \tilde{\mathrm{o}} : \mathsf{Obj} \mid \mathsf{r2}\ \tilde{\mathrm{s}} \\ \tilde{\mathrm{t}} : \mathsf{Callback} \end{array} \quad \widetilde{H}\ \mathsf{okheap}$$

Leverage heap type invariant via **type-consistent materialization** from okheap.

$$\widetilde{E}\ \begin{array}{l} \mathrm{s} : \tilde{\mathrm{s}} \\ \mathrm{o} : \tilde{\mathrm{o}} \\ \mathrm{this} : \tilde{\mathrm{t}} \end{array} \quad \widetilde{\Gamma}\ \begin{array}{l} \tilde{\mathrm{s}} : \mathsf{Str} \\ \tilde{\mathrm{o}} : \mathsf{Obj} \mid \mathsf{r2}\ \tilde{\mathrm{s}} \\ \tilde{\mathrm{t}} : \mathsf{Callback} \end{array} \quad \widetilde{H}\ \begin{array}{c} \mathsf{okheap} \\ * \\ \tilde{\mathbf{t}} \mapsto \{\mathbf{sel} \mapsto \widetilde{\mathbf{sel}} * \mathbf{obj} \mapsto \widetilde{\mathbf{obj}}\} \end{array}$$

```
this.sel = s
```

$$\widetilde{E}\ \begin{array}{l} \mathrm{s} : \tilde{\mathrm{s}} \\ \mathrm{o} : \tilde{\mathrm{o}} \\ \mathrm{this} : \tilde{\mathrm{t}} \end{array} \quad \widetilde{\Gamma}\ \begin{array}{l} \tilde{\mathrm{s}} : \mathsf{Str} \\ \tilde{\mathrm{o}} : \mathsf{Obj} \mid \mathsf{r2}\ \tilde{\mathrm{s}} \\ \tilde{\mathrm{t}} : \mathsf{Callback} \end{array} \quad \widetilde{H}\ \begin{array}{c} \mathsf{okheap} \\ * \\ \tilde{\mathrm{t}} \mapsto \{\mathrm{sel} \mapsto \tilde{\mathbf{s}} * \mathrm{obj} \mapsto \widetilde{\mathrm{obj}}\} \end{array}$$

```
this.obj = o
```

$$\widetilde{E}\ \begin{array}{l} \mathrm{s} : \tilde{\mathrm{s}} \\ \mathrm{o} : \tilde{\mathrm{o}} \\ \mathrm{this} : \tilde{\mathrm{t}} \end{array} \quad \widetilde{\Gamma}\ \begin{array}{l} \tilde{\mathrm{s}} : \mathsf{Str} \\ \tilde{\mathrm{o}} : \mathsf{Obj} \mid \mathsf{r2}\ \tilde{\mathrm{s}} \\ \tilde{\mathrm{t}} : \mathsf{Callback} \end{array} \quad \widetilde{H}\ \begin{array}{c} \mathsf{okheap} \\ * \\ \tilde{\mathrm{t}} \mapsto \{\mathrm{sel} \mapsto \tilde{\mathrm{s}} * \mathrm{obj} \mapsto \tilde{\mathbf{o}}\} \end{array}$$

Summarize type-consistent storage back into okheap. Requires **reasoning** explicitly **only** about memory **locations** where type constraint is **violated**.

$$\widetilde{E}\ \begin{array}{l} \mathrm{s} : \tilde{\mathrm{s}} \\ \mathrm{o} : \tilde{\mathrm{o}} \\ \mathrm{this} : \tilde{\mathrm{t}} \end{array} \quad \widetilde{\Gamma}\ \begin{array}{l} \tilde{\mathrm{s}} : \mathsf{Str} \\ \tilde{\mathrm{o}} : \mathsf{Obj} \mid \mathsf{r2}\ \tilde{\mathrm{s}} \\ \tilde{\mathrm{t}} : \mathsf{Callback} \end{array} \quad \widetilde{H}\ \mathsf{okheap}$$

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

Return to **type analysis** now that heap consists solely of **okheap.**

Soundness: Concretization of Base Types

When the **entire heap** is **okheap**, types have **same** meaning in the types domain and the symbolic domain.

$$\gamma(B) \triangleq \left\{ (H,a) \;\middle|\; \begin{array}{l} \text{exists } o \text{ where } H(a) = \langle o,B \rangle \text{ and} \\ \quad ① \text{ for all methods } m \\ \quad\quad o(m) \in \gamma(B, (\overline{p : T_p}) \to B_{\mathrm{ret}}) \text{ and} \\ \quad ② \text{ for all fields } f \\ \quad\quad (H,o,o(f)) \in \gamma(T^{\mathrm{F}}_f). \end{array} \right\}$$

$$\widetilde{\gamma}(B) \triangleq \left\{ (H^{\mathrm{ok}}, H^{\mathrm{mat}}, a) \;\middle|\; \begin{array}{l} \text{exists } o \text{ where } H^{\mathrm{ok}} \ast H^{\mathrm{mat}}(a) = \langle o,B \rangle \text{ and} \\ \quad ① \text{ for all methods } m \\ \quad\quad o(m) \in \gamma(B, (\overline{p : T_p}) \to B_{\mathrm{ret}}) \text{ and} \\ \quad ② \text{ for all fields } f \\ \quad\quad f \in \mathrm{dom}(o) \text{ and if } a \in \mathrm{dom}(H^{\mathrm{ok}}) \text{ then} \\ \quad\quad\quad (H,o,o(f)) \in \widetilde{\gamma}(T^{\mathrm{F}}_f). \end{array} \right\}$$

$$\gamma(B \restriction R^{\mathrm{F}}_1, \cdots, R^{\mathrm{F}}_n) \triangleq \left\{ (H,o,v) \;\middle|\; \begin{array}{l} (H,v) \in \gamma(B) \text{ and} \\ \text{for all refinements } R_i \\ \quad (H,o,v) \in \gamma(R^{\mathrm{F}}_i) \end{array} \right\}$$

$$\widetilde{\gamma}(B \restriction R^{\mathrm{F}}_1, \cdots, R^{\mathrm{F}}_n) \triangleq \left\{ (H^{\mathrm{ok}}, H^{\mathrm{mat}}, v) \;\middle|\; \begin{array}{l} (H,v) \in \widetilde{\gamma}(B) \text{ and} \\ \text{for all refinements } R_i \\ \quad (H^{\mathrm{ok}} \ast H^{\mathrm{mat}}, o, v) \in \gamma(R^{\mathrm{F}}_i) \end{array} \right\}$$

Types domain **Symbolic domain**

Evaluation: Reflective Method Call in Objective-C

	size		false alarms				analysis time	
benchmark	**(loc)**	**reflective call sites**	**flow-insensitive**	**almost-everywhere**	**symbolic sections**	**maximum materializations**	**Time**	**Rate (kloc/s)**
OAUTH	1248	7	7	2 (-71%)	7	1	0.24s	5.3
SCRECORDER	2716	12	2	0 (-100%)	2	2	0.28s	10.8
ZIPKIT	3301	28	0	0 (-)	0	0	0.10s	33.0
SPARKLE	5289	40	4	1 (-75%)	3	1	0.67s	7.9
ASIHTTPREQUEST	14620	68	50	10 (-80%)	59	2	0.50s	27.2
OMNIFRAMEWORKS	160769	192	82	74 (-10%)	9	1	4.25s	37.8
VIENNA	37327	186	59	38 (-36%)	28	2	2.79s	13.4
SKIM	60211	207	43	43 (-0%)	0	0	2.49s	24.1
ADIUM	176629	587	87	70 (-20%)	17	1	8.79s	20.1
combined	461080	1327	334	238 (-29%)	125	2	20.09s	23.0

- Significant **improvement** in **precision**
  - **Multiple** simultaneous **materializations** required (cf. linear locations)
- Runs at **interactive speeds**
  - But **not too many** materializations — **case split manageable**
- **Specification** burden is **reasonable**

Abstract

We present a generic analysis approach to the *imperative relationship update problem*, in which destructive updates temporarily violate a global invariant of interest. Such invariants can be conveniently and concisely specified with dependent refinement types, which are efficient to check flow-insensitively. Unfortunately, while traditional flow-insensitive type checking is fast, it is inapplicable when the desired invariants can be temporarily broken. To overcome this limitation, past works have directly ratcheted up the complexity of the type analysis and associated type invariants, leading to inefficient analysis and verbose specifications. In contrast, we propose a *generic lifting* of modular refinement type analyses with a symbolic analysis to efficiently and effectively check concise invariants that hold *almost everywhere*. The result is an efficient, highly modular flow-insensitive type analysis to *optimistically* check the preservation of global relationship invariants that can fall back to a precise, disjunctive symbolic analysis when the optimistic assumption is violated. This technique permits programmers to temporarily break and then re-establish relationship invariants—a flexibility that is crucial for checking relationships in real-world, imperative languages. A significant challenge is selectively violating the global type consistency invariant over heap locations, which we achieve via *almost type-consistent heaps*. To evaluate our approach, we have encoded the problem of verifying the safety of reflective method calls in dynamic languages as a refinement type checking problem. Our analysis is capable of validating reflective call safety at interactive speeds on commonly-used Objective-C libraries and applications.