**Goal:** Verify Reflective Method Call with Dependent Types

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

var sel : Str
var obj : Obj | r2 sel

Problem: Imperative Updates Violate Flow-Insensitive Types

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

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.

---

**Fissile Type Analysis:**

def update(s : Str, o : Obj | r2 s)
    Split type environment into facts about values and initially type-consistent symbolic memory.

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

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

---

**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: Modular Checking of Almost-Everywhere Invariants**

---

**Soundness:** Concretization of Base Types

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

Types domain

Symbolic domain

---

**Evaluation:** Reflective Method Call in Objective-C

<table>
<thead>
<tr>
<th>benchmark</th>
<th>size</th>
<th>reflection calls</th>
<th>false shares</th>
<th>concretize everywhere</th>
<th>symbolic sections</th>
<th>maximum materialization</th>
<th>performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>504</td>
<td>129</td>
<td>7</td>
<td>217/115</td>
<td>5</td>
<td>1</td>
<td>2.79s</td>
</tr>
<tr>
<td>Swift</td>
<td>479</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0.28s</td>
</tr>
<tr>
<td>Java</td>
<td>795</td>
<td>2</td>
<td>9</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>0.33s</td>
</tr>
<tr>
<td>Groff</td>
<td>528</td>
<td>40</td>
<td>4</td>
<td>11/7</td>
<td>3</td>
<td>0</td>
<td>0.87s</td>
</tr>
<tr>
<td>Phobos</td>
<td>1462</td>
<td>68</td>
<td>10</td>
<td>10/100</td>
<td>10</td>
<td>0</td>
<td>3.39s</td>
</tr>
<tr>
<td>Python</td>
<td>1552</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>2.78s</td>
</tr>
<tr>
<td>Value</td>
<td>272/2</td>
<td>506</td>
<td>50</td>
<td>50/50</td>
<td>50</td>
<td>0</td>
<td>2.78s</td>
</tr>
<tr>
<td>Java</td>
<td>1764/2</td>
<td>862</td>
<td>87</td>
<td>70/100</td>
<td>100</td>
<td>0</td>
<td>2.88s</td>
</tr>
</tbody>
</table>

---

• 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 hitched 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.