**Problem:** Local loss of precision in whole-program JavaScript static analysis can incur a massive slowdown as a result of dispatch.

```javascript
var o = { foo : function() { ... },
         bar : function() { ... },
         baz : function() {
             ... ; throw 'error'; 
         });
var prop = someFunction();
o[prop]();
```

**Example:** Over-approximation of `prop`'s value leads to spurious data-flow at the `o[prop]` call site.

**Our Solution:** Provide the whole-program analysis with a mechanism to refine its abstract state at particularly problematic locations.

**Goal-Directed Abstract Interpretation**

ThresherJS attempts to refute queries by soundly exploring the state-space backwards from a given location and abstract state. If all backwards paths reach a contradiction, the query is refuted; if some path reaches an entrypoint without contradiction, the query is not refuted – there may exist a concrete witness.

**Example:** Dynamic property read transfer function. Informally, the precondition state asserts that a prototype lookup on `y` with respect to property `z` yields some object `x'` whose `z` field satisfies any precondition constraints on `x`.

**Constraint Language**

<table>
<thead>
<tr>
<th>Constraints</th>
<th><code>c ::= x \mapsto \bar{x}</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Constraint</td>
<td><code>\pi(\bar{x}, \bar{x}, u, pg) = \bar{x}_3</code></td>
</tr>
<tr>
<td>Prototype Constraint</td>
<td><code>c; \varphi</code></td>
</tr>
<tr>
<td>abstract states</td>
<td><code>\varphi</code></td>
</tr>
<tr>
<td><code>\varphi</code> in State</td>
<td><code>c; \varphi</code></td>
</tr>
</tbody>
</table>

**Concretization**

- `\gamma(e; d) = \{(l, h, p, \eta); \sigma(e, \eta) \cap \Delta = \bot\}`
- `\gamma(\pi(\bar{x}_1, \bar{x}_2, u, pg) = \bar{x}_3)`
- `\exists` path `\exists \eta_1` to `\eta_2` in
  - `(l, h, p, \eta)`
  - `\forall u` in `\eta_1`, `\eta_2` in `\eta_2`
  - `\eta_2` in `\eta_2`

**Worked Example**

- `o\mapsto \hat{o}; \mapsto \hat{p} \land x \mapsto \hat{x} \wedge\pi(\hat{o}, \hat{p}, \{o \mapsto \hat{x}\}) = \hat{z} \wedge\hat{z}[\hat{p}] \mapsto \hat{n} \land \hat{n} : 5 \land \hat{p} \neq \hat{p}`
- `Read property matches write property and \(x \neq z\)`
- `\{o \mapsto \hat{o}; \mapsto \hat{p} \land x \mapsto \hat{x} \wedge\pi(\hat{o}, \hat{p}, \{o \mapsto \hat{x}\}) = \hat{z} \wedge\hat{z}[\hat{p}] \mapsto \hat{n} \land \hat{n} : 5\}
- `\{o \mapsto \hat{o}; \mapsto \hat{p} \land \pi(\hat{o}, \hat{p}, \{o \mapsto \hat{x}\}) = \hat{z} \wedge\hat{z}[\hat{p}] \mapsto \hat{n} \land \hat{n} : 5\}

**Abstract**

JavaScript is notoriously difficult to analyze due to its rampant use of standard dynamic features (e.g., duck typing, dynamic dispatch, first-class functions, and run-time string evaluation), as well as its idiosyncratic approach to scoping (scope object chains) and inheritance (prototyping). Therefore, despite its near-universal adoption as a clientside scripting language and its increasing use in server-side and mobile applications, JavaScript is rarely analyzed in practice and can be quite buggy, unreliable, and unsafe.

We present a novel technique to improve precision and efficiency of JavaScript analysis by combining a forwards whole-program type analysis with a goal-directed backwards abstract interpretation refutation mechanism. The backwards abstract interpretation can operate either as a standalone tool to refute false alarms that arise from over-approximation in the forwards analysis, or on-line, refuting spurious data-flow on demand at critical points during the whole-program forwards analysis.

**References**

