Flix: A Language for Static Analysis

Magnus Madsen, **Ming-Ho Yee**, Ondřej Lhoták

October 20, 2016

Datalog

- A declarative programming language
  - Syntactic subset of Prolog, but different semantics
  - Every Datalog program terminates with a unique solution
  - [Ceri, Gottlob, and Tanca, TKDE 1989]

- Datalog has been used for points-to analyses
  - Separates specification from implementation
  - [Bravenboer and Smaragdakis, OOPSLA ‘09]

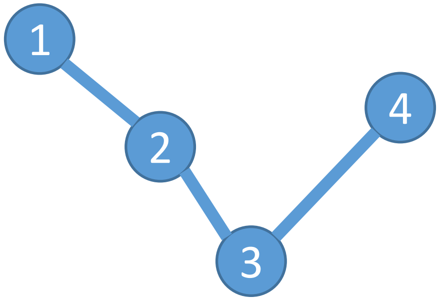
Example: Transitive Closure

```
// Rules
Path(x, y) :- Edge(x, y).
Path(x, z) :- Path(x, y), Edge(y, z).
```

Head

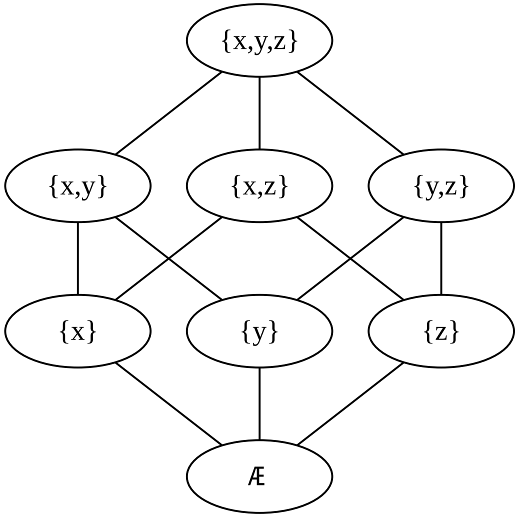
Body

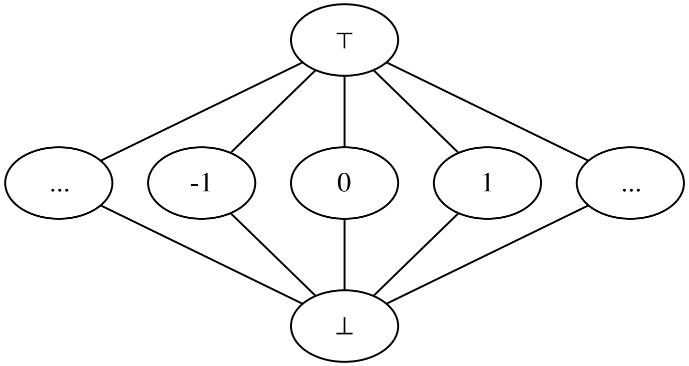
```
// Facts
Edge(1, 2).
Edge(2, 3).
Edge(3, 4).
```



Limitations of Datalog

- No user-defined lattices
- No functions
- Poor interoperability





A Language for Static Analysis

- Flix extends Datalog with lattices and functions
    - Logic language
    - Functional language
    - [Madsen, Yee, and Lhoták, PLDI ’16]

- Flix is implemented on the JVM

The Anatomy of a **Datalog** Rule

$$H(\bar{t}) \Leftarrow B(\bar{t}), \ldots, B(\bar{t}).$$

The Anatomy of a **Datalog** Rule

$$H(\bar{t}) \Leftarrow \underbrace{B(\bar{t}), \ldots, B(\bar{t}).}_{\text{Body}}$$

The Anatomy of a **Datalog** Rule

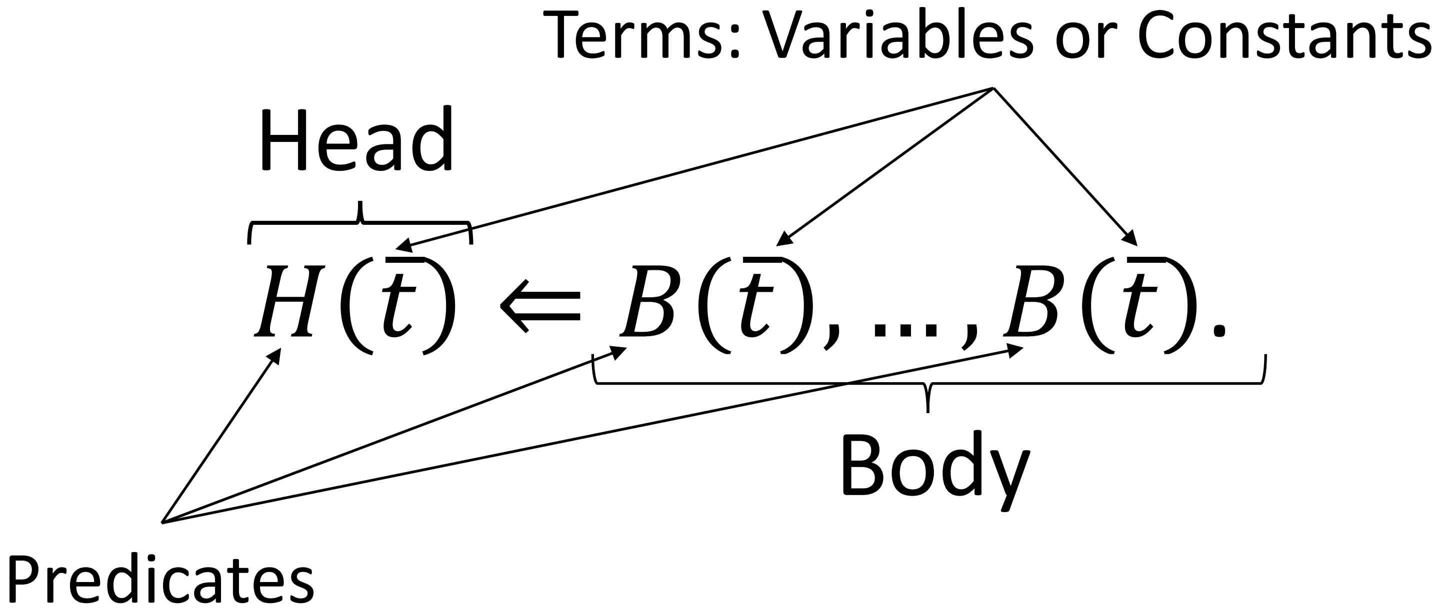
$$\overbrace{H(\bar{t})}^{\text{Head}} \Leftarrow \underbrace{B(\bar{t}), \ldots, B(\bar{t})}_{\text{Body}}.$$

The Anatomy of a **Datalog** Rule

$$\overbrace{H(\bar{t})}^{\text{Head}} \Leftarrow \underbrace{B(\bar{t}), \ldots, B(\bar{t}).}_{\text{Body}}$$

Predicates

The Anatomy of a **Datalog** Rule



Terms: Variables or Constants

$$\overbrace{H(\bar{t})}^{\text{Head}} \Leftarrow \underbrace{B(\bar{t}), \dots, B(\bar{t}).}_{\text{Body}}$$

Predicates

The Anatomy of a **Datalog** Rule

$$H(\bar{t}) \Leftarrow B(\bar{t}), \ldots, B(\bar{t}).$$

The Anatomy of a **Flix** Rule

$$H_\ell\left(\bar{t}, f(\bar{t})\right) \Leftarrow \varphi(\bar{t}), B_\ell(\bar{t}), \ldots, B_\ell(\bar{t}).$$

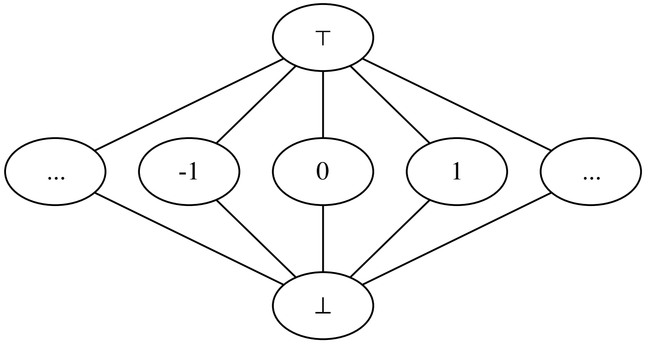
The Anatomy of a **Flix** Rule

$$H_{\ell}\left(\bar{t}, f(\bar{t})\right) \Leftarrow \overbrace{\varphi(\bar{t})}^{\text{Filter Function}}, B_{\ell}(\bar{t}), \ldots, B_{\ell}(\bar{t}).$$

The Anatomy of a **Flix** Rule

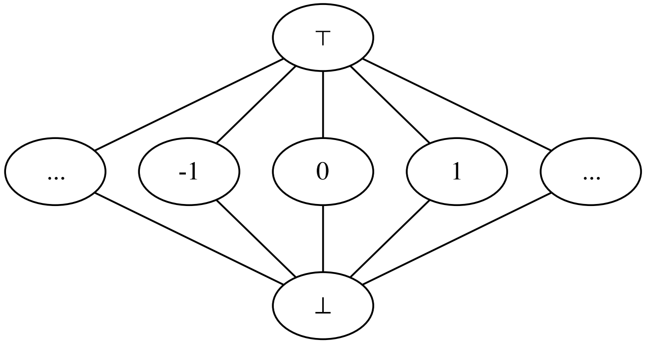
$$H_{\ell}(\bar{t}, \underbrace{f(\bar{t})}_{\text{Transfer Function}}) \Leftarrow \overbrace{\varphi(\bar{t})}^{\text{Filter Function}}, B_{\ell}(\bar{t}), \ldots, B_{\ell}(\bar{t}).$$

Constant Propagation



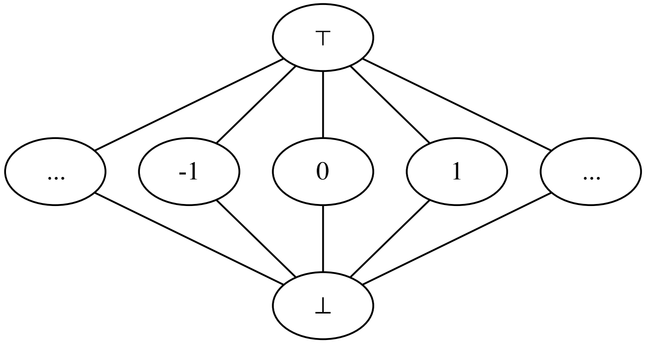
```
enum Constant {
  case Top, case Cst(Int), case Bot
}

def leq(e1: Constant, e2: Constant): Bool =
  match (e1, e2) with {
    case (Bot, _)              => true
    case (Cst(n1), Cst(n2)) => n1 == n2
    case (_, Top)              => true
    case _                     => false
  }

def lub(e1: Constant, e2: Constant): Constant = …
def glb(e1: Constant, e2: Constant): Constant = …
```

Constant Propagation



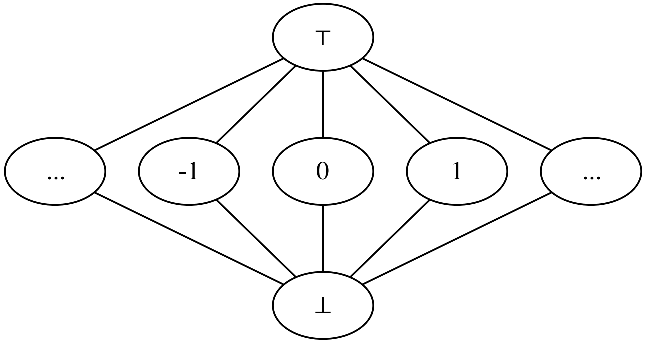
```
def sum(e1: Constant, e2: Constant): Constant =
  match (e1, e2) with {
    case (_, Bot)               => Bot
    case (Bot, _)               => Bot
    case (Cst(n1), Cst(n2)) => Cst(n1 + n2)
    case _                      => Top
  }
```

Constant Propagation



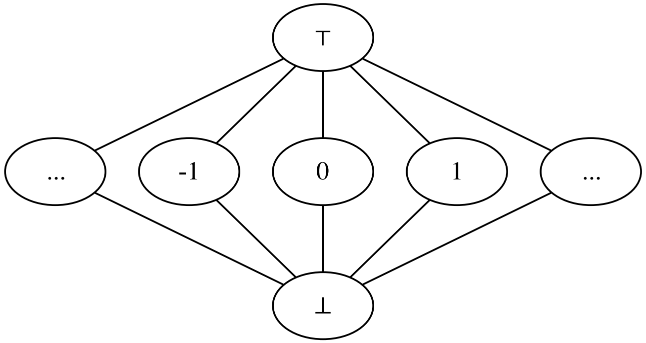
```
// analysis inputs
rel AsnStm(r: Str, c: Int)              // r = c
rel AddStm(r: Str, x: Str, y: Str)  // r = x + y

// analysis outputs
lat LocalVar(k: Str, v: Constant)

// rules
LocalVar(r, Cst(c)) :- AsnStm(r, c).

LocalVar(r, sum(v1, v2)) :- AddStm(r, x, y),
                            LocalVar(x, v1),
                            LocalVar(y, v2).
```

Constant Propagation



```
LocalVar(r, Cst(c)) :- AsnStm(r, c).


// input facts
AsnStm("x", 0).
AsnStm("x", 1).


// output facts
```

Constant Propagation

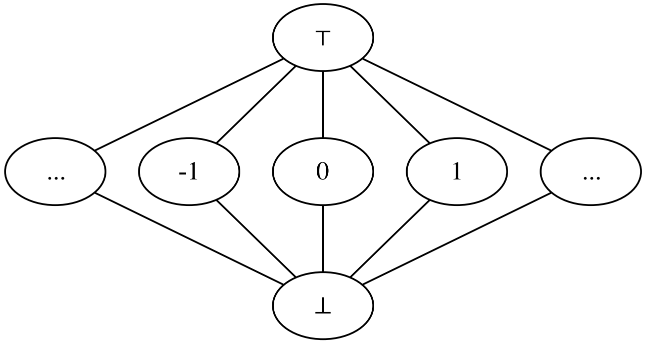


```
LocalVar(r, Cst(c)) :- AsnStm(r, c).
```

```
// input facts
AsnStm("x", 0).
AsnStm("x", 1).
```

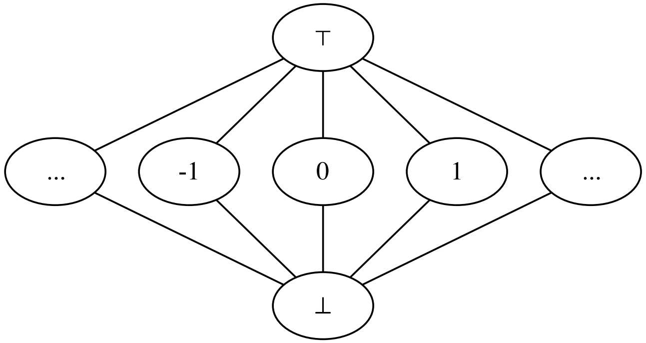
```
// output facts
LocalVar("x", Cst(0)).
```

Constant Propagation



```
LocalVar(r, Cst(c)) :- AsnStm(r, c).


// input facts
AsnStm("x", 0).
AsnStm("x", 1).


// output facts
LocalVar("x", Cst(0)).
LocalVar("x", Cst(1)).
```

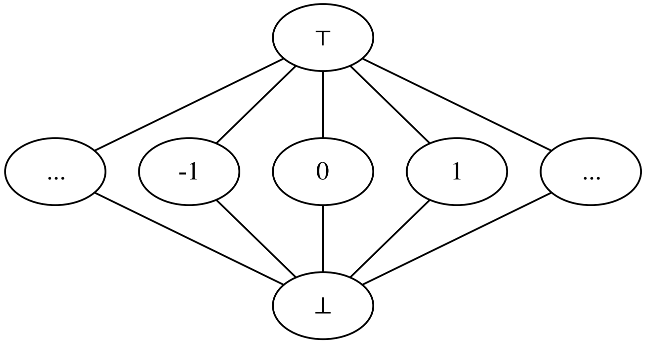
Constant Propagation



```
LocalVar(r, Cst(c)) :- AsnStm(r, c).


// input facts
AsnStm("x", 0).
AsnStm("x", 1).


// output facts
```

~~`LocalVar("x", Cst(0)).`~~
~~`LocalVar("x", Cst(1)).`~~

Constant Propagation



```
LocalVar(r, Cst(c)) :- AsnStm(r, c).
```

```
// input facts
AsnStm("x", 0).
AsnStm("x", 1).
```

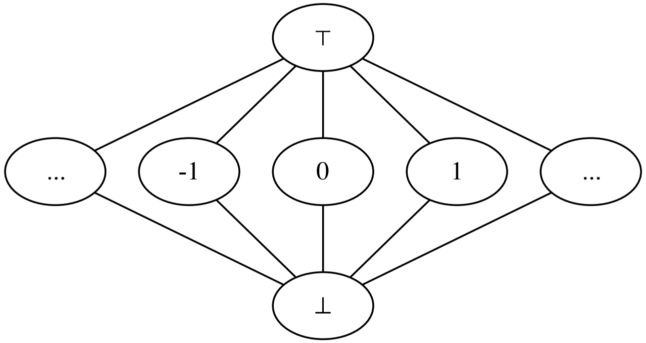
```
// output facts
```

~~`LocalVar("x", Cst(0)).`~~

~~`LocalVar("x", Cst(1)).`~~

```
LocalVar("x", lub(Cst(0), Cst(1))).
```

Constant Propagation



```
LocalVar(r, Cst(c)) :- AsnStm(r, c).


// input facts
AsnStm("x", 0).
AsnStm("x", 1).


// output facts
```

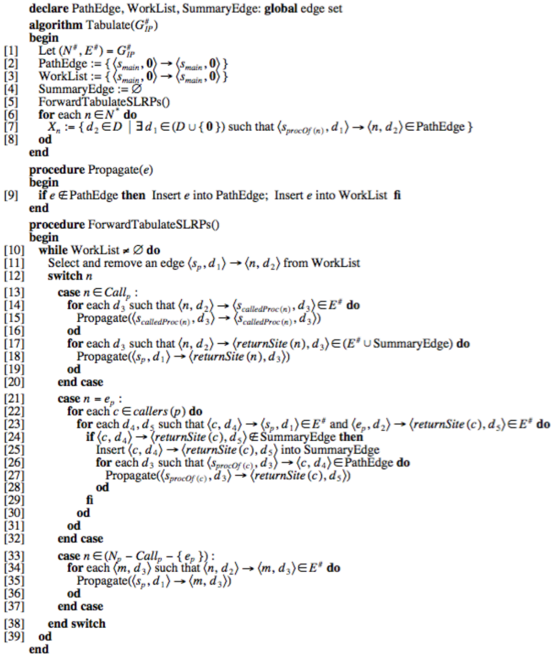
~~LocalVar("x", Cst(0)).~~
~~LocalVar("x", Cst(1)).~~

```
LocalVar("x", Top).
```

More Analyses in Flix

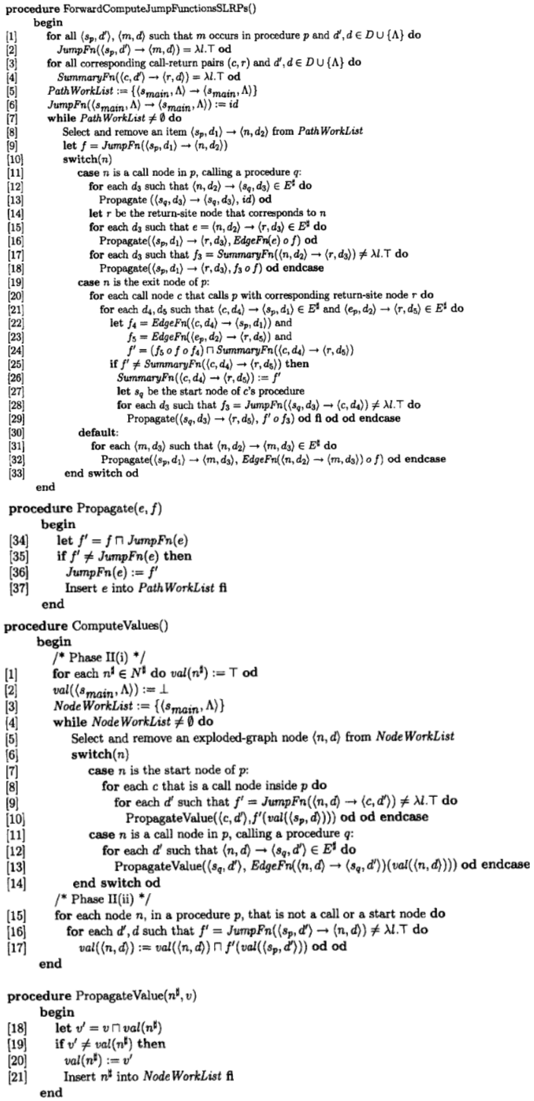
- Strong Update analysis
  - [Lhoták and Chung, POPL ‘11]
- IFDS algorithm
  - [Reps, Horwitz, and Sagiv, POPL ‘95]
- IDE algorithm
  - [Sagiv, Reps, and Horwitz, TCS ‘96]

IFDS



```
declare PathEdge, WorkList, SummaryEdge: global edge set
algorithm Tabulate($G^{\#}_{IP}$)
begin
  Let $(N^{\#}, E^{\#}) = G^{\#}_{IP}$
  PathEdge := $\{ \langle s_{main}, \mathbf{0} \rangle \to \langle s_{main}, \mathbf{0} \rangle \}$
  WorkList := $\{ \langle s_{main}, \mathbf{0} \rangle \to \langle s_{main}, \mathbf{0} \rangle \}$
  SummaryEdge := $\varnothing$
  ForwardTabulateSLRPs()
  for each $n \in N^{*}$ do
    $X_n := \{ d_2 \in D \mid \exists d_1 \in (D \cup \{ \mathbf{0} \})$ such that $\langle s_{procOf(n)}, d_1 \rangle \to \langle n, d_2 \rangle \in$ PathEdge $\}$
  od
end

procedure Propagate($e$)
begin
  if $e \notin$ PathEdge then  Insert $e$ into PathEdge;  Insert $e$ into WorkList  fi
end

procedure ForwardTabulateSLRPs()
begin
  while WorkList $\neq \varnothing$ do
    Select and remove an edge $\langle s_p, d_1 \rangle \to \langle n, d_2 \rangle$ from WorkList
    switch $n$

      case $n \in Call_p$ :
        for each $d_3$ such that $\langle n, d_2 \rangle \to \langle s_{calledProc(n)}, d_3 \rangle \in E^{\#}$ do
          Propagate($\langle s_{calledProc(n)}, d_3 \rangle \to \langle s_{calledProc(n)}, d_3 \rangle$)
        od
        for each $d_3$ such that $\langle n, d_2 \rangle \to \langle returnSite(n), d_3 \rangle \in (E^{\#} \cup$ SummaryEdge) do
          Propagate($\langle s_p, d_1 \rangle \to \langle returnSite(n), d_3 \rangle$)
        od
      end case

      case $n = e_p$ :
        for each $c \in callers(p)$ do
          for each $d_4, d_5$ such that $\langle c, d_4 \rangle \to \langle s_p, d_1 \rangle \in E^{\#}$ and $\langle e_p, d_2 \rangle \to \langle returnSite(c), d_5 \rangle \in E^{\#}$ do
            if $\langle c, d_4 \rangle \to \langle returnSite(c), d_5 \rangle \notin$ SummaryEdge then
              Insert $\langle c, d_4 \rangle \to \langle returnSite(c), d_5 \rangle$ into SummaryEdge
              for each $d_3$ such that $\langle s_{procOf(c)}, d_3 \rangle \to \langle c, d_4 \rangle \in$ PathEdge do
                Propagate($\langle s_{procOf(c)}, d_3 \rangle \to \langle returnSite(c), d_5 \rangle$)
              od
            fi
          od
        od
      end case

      case $n \in (N_p - Call_p - \{ e_p \})$ :
        for each $\langle m, d_3 \rangle$ such that $\langle n, d_2 \rangle \to \langle m, d_3 \rangle \in E^{\#}$ do
          Propagate($\langle s_p, d_1 \rangle \to \langle m, d_3 \rangle$)
        od
      end case

    end switch
  od
end
```

IDE



```
procedure ForwardComputeJumpFunctionsSLRPs()
    begin
      for all $\langle s_p, d' \rangle, \langle m, d \rangle$ such that $m$ occurs in procedure $p$ and $d', d \in D \cup \{\Lambda\}$ do
        $JumpFn(\langle s_p, d' \rangle \to \langle m, d \rangle) = \lambda l.\top$ od
      for all corresponding call-return pairs $(c, r)$ and $d', d \in D \cup \{\Lambda\}$ do
        $SummaryFn(\langle c, d' \rangle \to \langle r, d \rangle) = \lambda l.\top$ od
      $PathWorkList := \{\langle s_{main}, \Lambda \rangle \to \langle s_{main}, \Lambda \rangle\}$
      $JumpFn(\langle s_{main}, \Lambda \rangle \to \langle s_{main}, \Lambda \rangle) := id$
      while $PathWorkList \neq \emptyset$ do
        Select and remove an item $\langle s_p, d_1 \rangle \to \langle n, d_2 \rangle$ from $PathWorkList$
        let $f = JumpFn(\langle s_p, d_1 \rangle \to \langle n, d_2 \rangle)$
        switch($n$)
          case $n$ is a call node in $p$, calling a procedure $q$:
            for each $d_3$ such that $\langle n, d_2 \rangle \to \langle s_q, d_3 \rangle \in E^{\sharp}$ do
              Propagate $(\langle s_q, d_3 \rangle \to \langle s_q, d_3 \rangle, id)$ od
            let $r$ be the return-site node that corresponds to $n$
            for each $d_3$ such that $e = \langle n, d_2 \rangle \to \langle r, d_3 \rangle \in E^{\sharp}$ do
              Propagate($\langle s_p, d_1 \rangle \to \langle r, d_3 \rangle, EdgeFn(e) \circ f$) od
            for each $d_3$ such that $f_3 = SummaryFn(\langle n, d_2 \rangle \to \langle r, d_3 \rangle) \neq \lambda l.\top$ do
              Propagate($\langle s_p, d_1 \rangle \to \langle r, d_3 \rangle, f_3 \circ f$) od endcase
          case $n$ is the exit node of $p$:
            for each call node $c$ that calls $p$ with corresponding return-site node $r$ do
              for each $d_4, d_5$ such that $\langle c, d_4 \rangle \to \langle s_p, d_1 \rangle \in E^{\sharp}$ and $\langle e_p, d_2 \rangle \to \langle r, d_5 \rangle \in E^{\sharp}$ do
                let $f_4 = EdgeFn(\langle c, d_4 \rangle \to \langle s_p, d_1 \rangle)$ and
                    $f_5 = EdgeFn(\langle e_p, d_2 \rangle \to \langle r, d_5 \rangle)$ and
                    $f' = (f_5 \circ f \circ f_4) \sqcap SummaryFn(\langle c, d_4 \rangle \to \langle r, d_5 \rangle)$
                if $f' \neq SummaryFn(\langle c, d_4 \rangle \to \langle r, d_5 \rangle)$ then
                  $SummaryFn(\langle c, d_4 \rangle \to \langle r, d_5 \rangle) := f'$
                  let $s_q$ be the start node of $c$'s procedure
                  for each $d_3$ such that $f_3 = JumpFn(\langle s_q, d_3 \rangle \to \langle c, d_4 \rangle) \neq \lambda l.\top$ do
                    Propagate($\langle s_q, d_3 \rangle \to \langle r, d_5 \rangle, f' \circ f_3$) od fi od od endcase
          default:
            for each $\langle m, d_3 \rangle$ such that $\langle n, d_2 \rangle \to \langle m, d_3 \rangle \in E^{\sharp}$ do
              Propagate($\langle s_p, d_1 \rangle \to \langle m, d_3 \rangle, EdgeFn(\langle n, d_2 \rangle \to \langle m, d_3 \rangle) \circ f$) od endcase
        end switch od
    end

procedure Propagate($e, f$)
    begin
      let $f' = f \sqcap JumpFn(e)$
      if $f' \neq JumpFn(e)$ then
        $JumpFn(e) := f'$
        Insert $e$ into $PathWorkList$ fi
    end

procedure ComputeValues()
    begin
      /* Phase II(i) */
      for each $n^{\sharp} \in N^{\sharp}$ do $val(n^{\sharp}) := \top$ od
      $val(\langle s_{main}, \Lambda \rangle) := \bot$
      $NodeWorkList := \{\langle s_{main}, \Lambda \rangle\}$
      while $NodeWorkList \neq \emptyset$ do
        Select and remove an exploded-graph node $\langle n, d \rangle$ from $NodeWorkList$
        switch($n$)
          case $n$ is the start node of $p$:
            for each $c$ that is a call node inside $p$ do
              for each $d'$ such that $f' = JumpFn(\langle n, d \rangle \to \langle c, d' \rangle) \neq \lambda l.\top$ do
                PropagateValue($\langle c, d' \rangle, f'(val(\langle s_p, d \rangle))$) od od endcase
          case $n$ is a call node in $p$, calling a procedure $q$:
            for each $d'$ such that $\langle n, d \rangle \to \langle s_q, d' \rangle \in E^{\sharp}$ do
              PropagateValue($\langle s_q, d' \rangle, EdgeFn(\langle n, d \rangle \to \langle s_q, d' \rangle)(val(\langle n, d \rangle))$) od endcase
        end switch od
      /* Phase II(ii) */
      for each node $n$, in a procedure $p$, that is not a call or a start node do
        for each $d', d$ such that $f' = JumpFn(\langle s_p, d' \rangle \to \langle n, d \rangle) \neq \lambda l.\top$ do
          $val(\langle n, d \rangle) := val(\langle n, d \rangle) \sqcap f'(val(\langle s_p, d' \rangle))$ od od
    end

procedure PropagateValue($n^{\sharp}, v$)
    begin
      let $v' = v \sqcap val(n^{\sharp})$
      if $v' \neq val(n^{\sharp})$ then
        $val(n^{\sharp}) := v'$
        Insert $n^{\sharp}$ into $NodeWorkList$ fi
    end
```

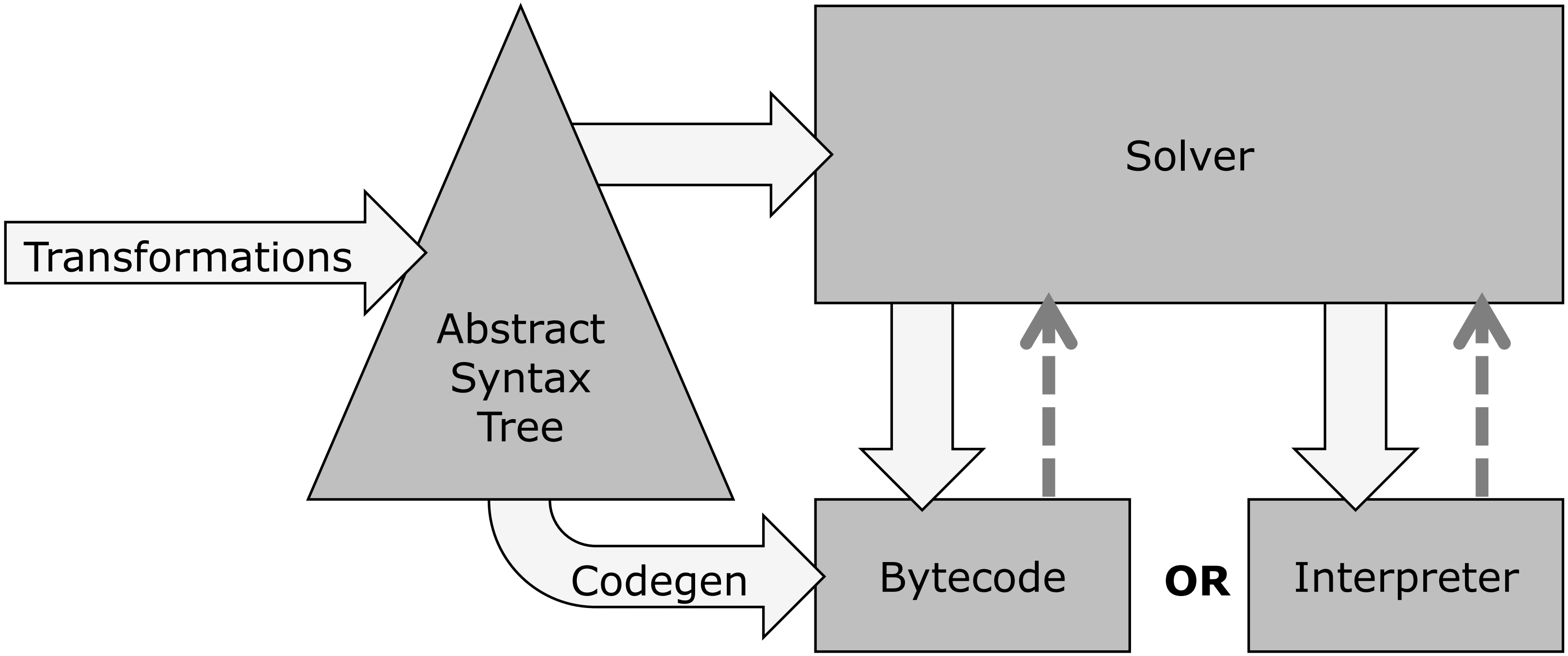
IFDS in Flix

```
PathEdge(d1, m, d3) :-
    CFG(n, m),
    PathEdge(d1, n, d2),
    d3 <- eshIntra(n, d2).
PathEdge(d1, m, d3) :-
    CFG(n, m),
    PathEdge(d1, n, d2),
    SummaryEdge(n, d2, d3).
PathEdge(d3, start, d3) :-
    PathEdge(d1, call, d2),
    CallGraph(call, target),
    EshCallStart(call, d2, target, d3),
    StartNode(target, start).
SummaryEdge(call, d4, d5) :-
    CallGraph(call, target),
    StartNode(target, start),
    EndNode(target, end),
    EshCallStart(call, d4, target, d1),
    PathEdge(d1, end, d2),
    d5 <- eshEndReturn(target, d2, call).

EshCallStart(call, d, target, d2) :-
    PathEdge(_, call, d),
    CallGraph(call, target),
    d2 <- eshCallStart(call, d, target).

Result(n, d2) :-
    PathEdge(_, n, d2).
```

IDE in Flix

```
JumpFn(d1, m, d3, comp(long, short)) :-
    CFG(n, m),
    JumpFn(d1, n, d2, long),
    (d3, short) <- eshIntra(n, d2).
JumpFn(d1, m, d3, comp(caller, summary)) :-
    CFG(n, m),
    JumpFn(d1, n, d2, caller),
    SummaryFn(n, d2, d3, summary).
JumpFn(d3, start, d3, identity()) :-
    JumpFn(d1, call, d2, _),
    CallGraph(call, target),
    EshCallStart(call, d2, target, d3, _),
    StartNode(target, start).
SummaryFn(call, d4, d5, comp(comp(cs, se), er)) :-
    CallGraph(call, target),
    StartNode(target, start),
    EndNode(target, end),
    EshCallStart(call, d4, target, d1, cs),
    JumpFn(d1, end, d2, se),
    (d5, er) <- eshEndReturn(target, d2, call).

EshCallStart(call, d, target, d2, cs) :-
    JumpFn(_, call, d, _),
    CallGraph(call, target),
    (d2, cs) <- eshCallStart(call, d, target).

InProc(p, start) :- StartNode(p, start).
InProc(p, m) :- InProc(p, n), CFG(n, m).

Result(n, d, apply(fn, vp)) :-
    ResultProc(proc, dp, vp),
    InProc(proc, n),
    JumpFn(dp, n, d, fn).

ResultProc(proc, dp, apply(cs, v)) :-
    Result(call, d, v),
    EshCallStart(call, d, proc, dp, cs).
```

Back-end Architecture



Lambda Functions

- Functions are first-class
  - Can be nested, stored in variables, passed as arguments, returned from functions…

- No nested methods in bytecode
- Target of a call must be a method reference

- Need a closure conversion pass

Implementing Closures...?

```
// Scala
val a = 10
val f = (x: Int, y: Int) => a + x + y
f(1, 2) // 13
```

```
// Compiled Scala
class anon$fun(a$0: Int) extends Function2 {
  def apply(x: Int, y: Int) = a$0 + x + y
}
val a = 10
val f = new anon$fun(a)
f.apply(1, 2) // 13
```

Using `invokedynamic`

- Flix uses the same strategy as Java 8 and Scala 2.12
  - Create closure object with `invokedynamic`

- `invokedynamic` represents a dynamic call site
  - Initially, target method is unknown
  - `invokedynamic` calls bootstrap method to link target
  - Subsequent calls skip bootstrap and directly call target

Implementing Closures

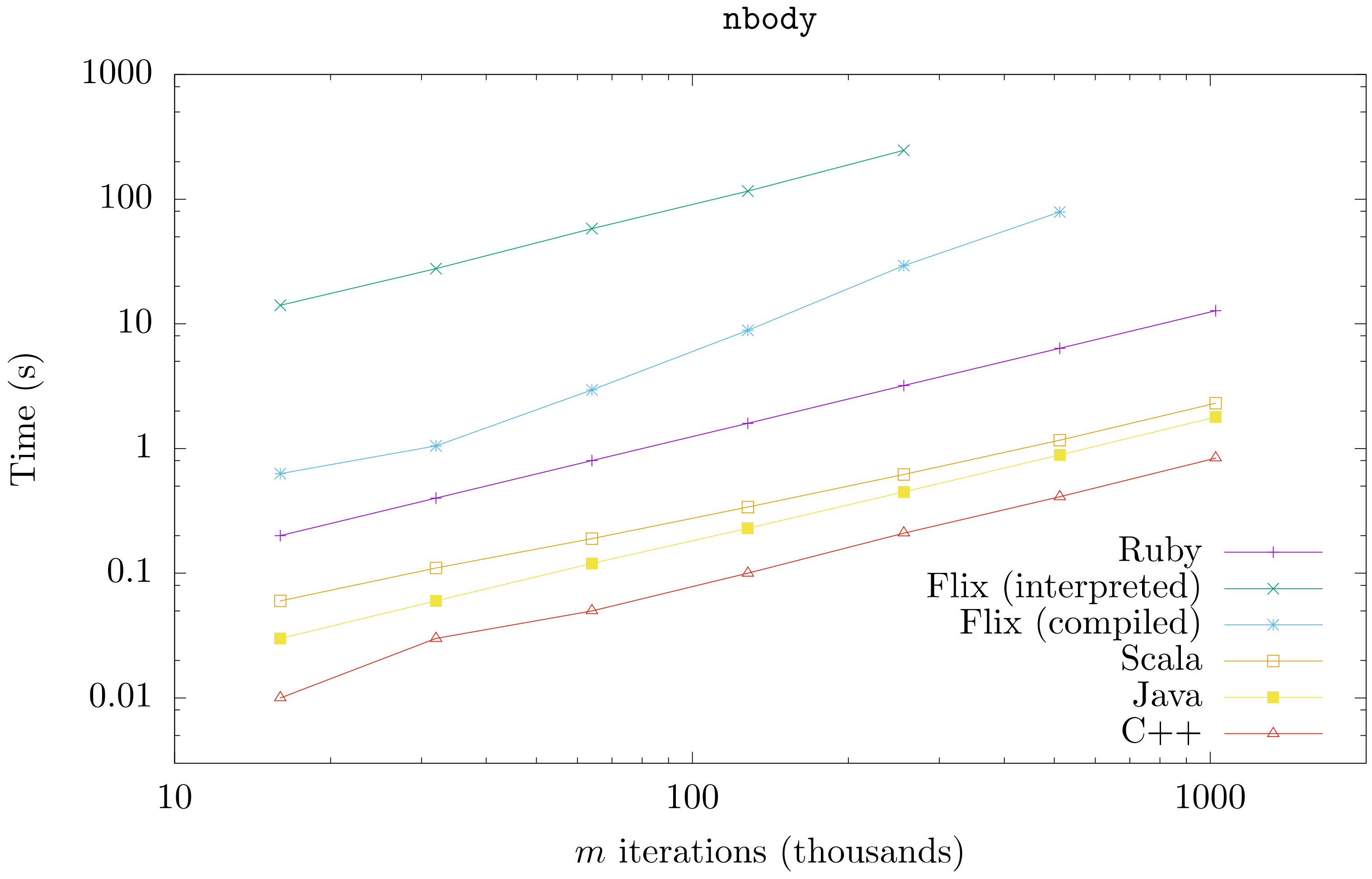
- Closure creation
  - `invokedynamic` call to Java’s `LambdaMetafactory`
  - Static arguments: functional interface, method handle
  - Dynamic arguments: captured values

- Closure call
  - Emit an interface call

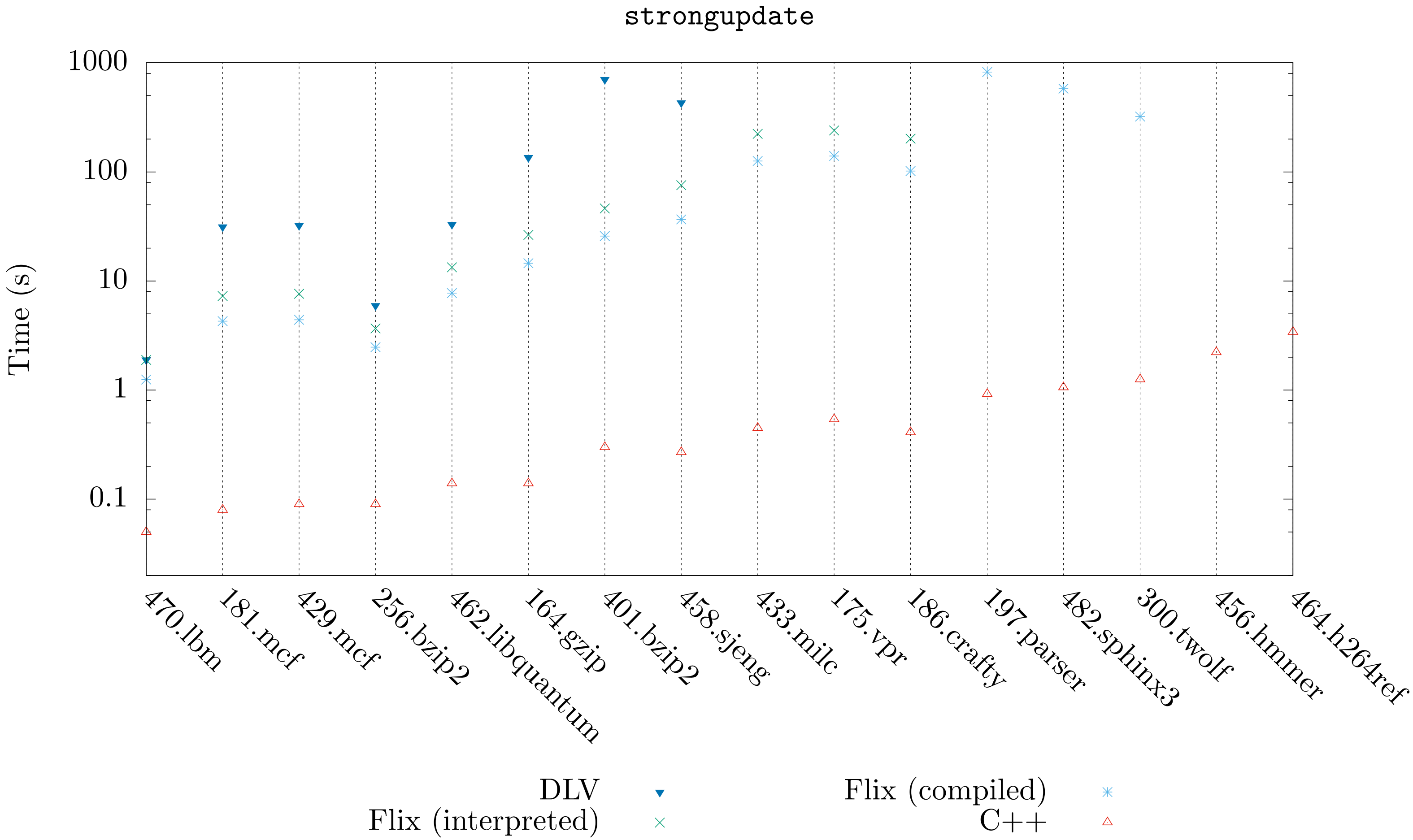
Generating Functional Interfaces

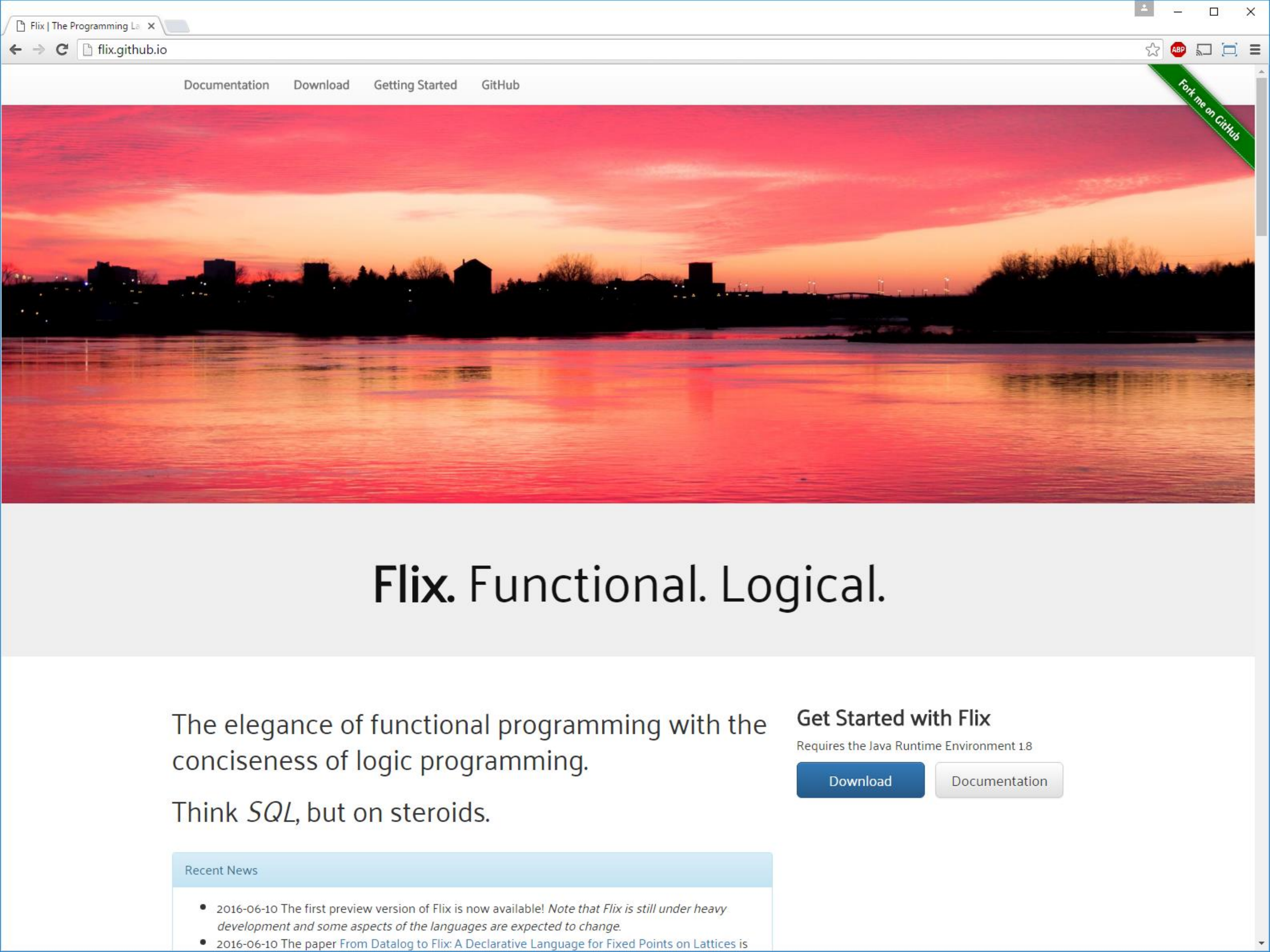
- A closure object implements a functional interface
    - Interface is provided by the implementation
- Flix generates its own functional interfaces
- Before code generation, traverse AST to collect type signatures of closures
    - Generate the interfaces

Evaluation – nbody



Evaluation – strongupdate





Flix. Functional. Logical.

The elegance of functional programming with the conciseness of logic programming.

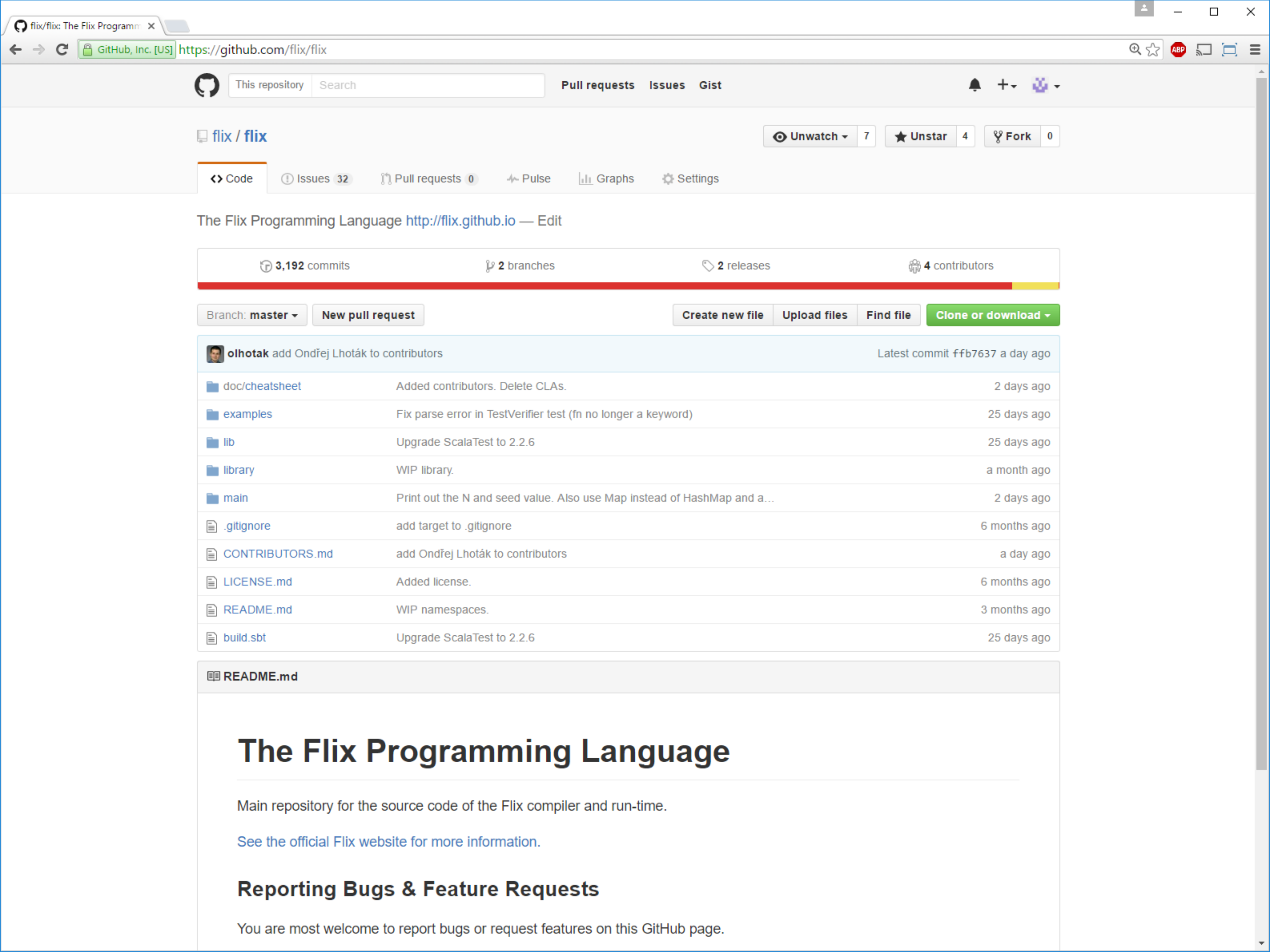
Think *SQL*, but on steroids.

Get Started with Flix

Requires the Java Runtime Environment 1.8

Recent News

- 2016-06-10 The first preview version of Flix is now available! *Note that Flix is still under heavy development and some aspects of the languages are expected to change.*
- 2016-06-10 The paper From Datalog to Flix: A Declarative Language for Fixed Points on Lattices is



The Flix Programming Language http://flix.github.io — Edit

3,192 commits · 2 branches · 2 releases · 4 contributors

olhotak add Ondřej Lhoták to contributors — Latest commit `ffb7637` a day ago

doc/cheatsheet	Added contributors. Delete CLAs.	2 days ago
examples	Fix parse error in TestVerifier test (fn no longer a keyword)	25 days ago
lib	Upgrade ScalaTest to 2.2.6	25 days ago
library	WIP library.	a month ago
main	Print out the N and seed value. Also use Map instead of HashMap and a…	2 days ago
.gitignore	add target to .gitignore	6 months ago
CONTRIBUTORS.md	add Ondřej Lhoták to contributors	a day ago
LICENSE.md	Added license.	6 months ago
README.md	WIP namespaces.	3 months ago
build.sbt	Upgrade ScalaTest to 2.2.6	25 days ago

README.md

The Flix Programming Language

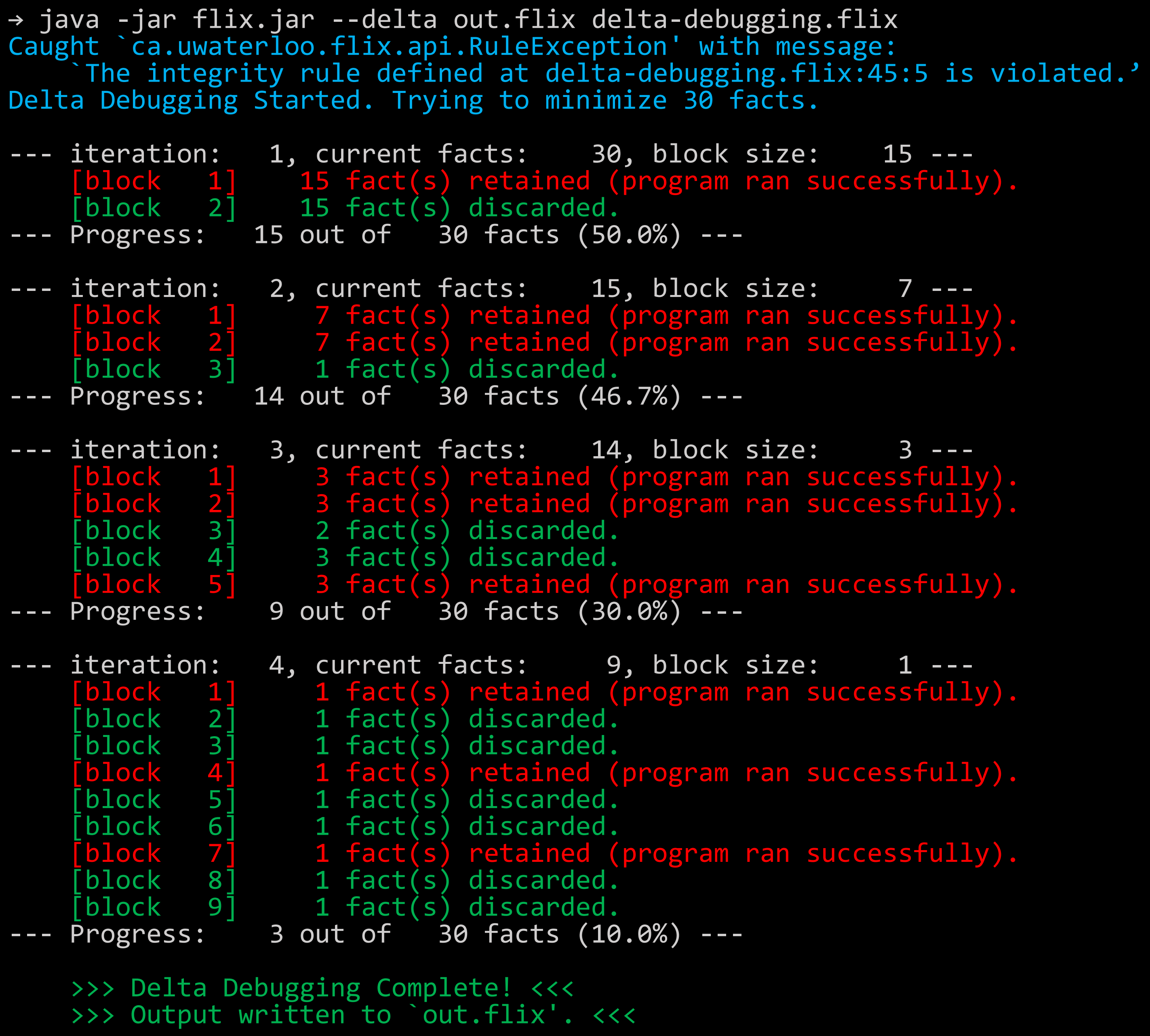
Main repository for the source code of the Flix compiler and run-time.

See the official Flix website for more information.

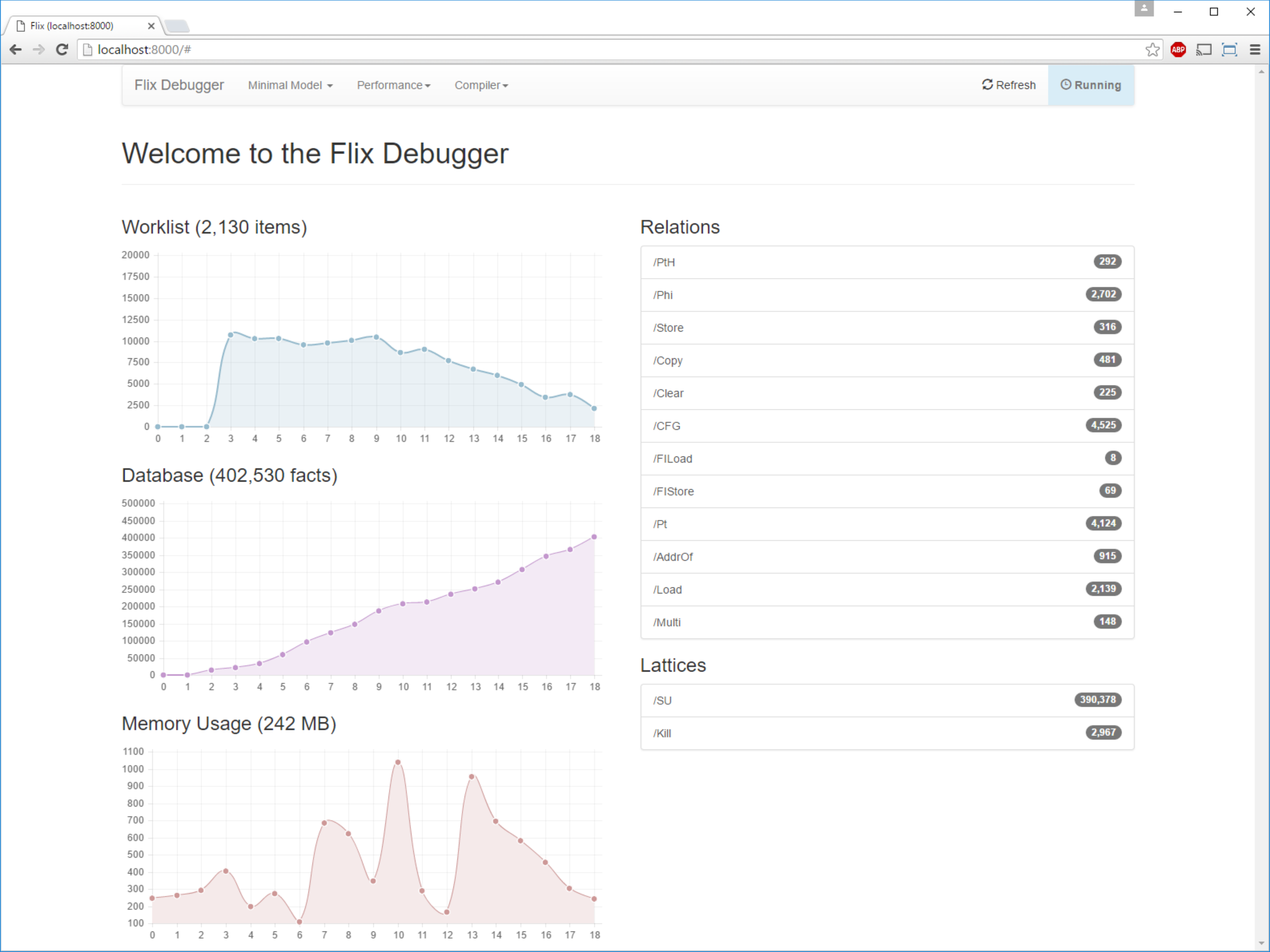
Reporting Bugs & Feature Requests

You are most welcome to report bugs or request features on this GitHub page.

```
→ java -jar flix.jar --verifier Sign.flix
-- VERIFIER ERROR ------------------------------------ Sign.flix

>> The function is not monotone.

Counter-example: x1$1402 -> Zer, x2$1406 -> Neg, y1$1404 -> Pos,
y2$1408 -> Pos

The function was defined here:
238|    def or(e1: Sign, e2: Sign): Sign = match (e1, e2) with {
            ^^
```



```
→ java -jar flix.jar --delta out.flix delta-debugging.flix
Caught `ca.uwaterloo.flix.api.RuleException' with message:
    `The integrity rule defined at delta-debugging.flix:45:5 is violated.’
Delta Debugging Started. Trying to minimize 30 facts.

--- iteration:   1, current facts:    30, block size:    15 ---
    [block   1]    15 fact(s) retained (program ran successfully).
    [block   2]    15 fact(s) discarded.
--- Progress:   15 out of   30 facts (50.0%) ---

--- iteration:   2, current facts:    15, block size:     7 ---
    [block   1]     7 fact(s) retained (program ran successfully).
    [block   2]     7 fact(s) retained (program ran successfully).
    [block   3]     1 fact(s) discarded.
--- Progress:   14 out of   30 facts (46.7%) ---

--- iteration:   3, current facts:    14, block size:     3 ---
    [block   1]     3 fact(s) retained (program ran successfully).
    [block   2]     3 fact(s) retained (program ran successfully).
    [block   3]     2 fact(s) discarded.
    [block   4]     3 fact(s) discarded.
    [block   5]     3 fact(s) retained (program ran successfully).
--- Progress:    9 out of   30 facts (30.0%) ---

--- iteration:   4, current facts:     9, block size:     1 ---
    [block   1]     1 fact(s) retained (program ran successfully).
    [block   2]     1 fact(s) discarded.
    [block   3]     1 fact(s) discarded.
    [block   4]     1 fact(s) retained (program ran successfully).
    [block   5]     1 fact(s) discarded.
    [block   6]     1 fact(s) discarded.
    [block   7]     1 fact(s) retained (program ran successfully).
    [block   8]     1 fact(s) discarded.
    [block   9]     1 fact(s) discarded.
--- Progress:    3 out of   30 facts (10.0%) ---

    >>> Delta Debugging Complete! <<<
    >>> Output written to `out.flix'. <<<
```



Welcome to the Flix Debugger

Worklist (2,130 items)

Database (402,530 facts)

Memory Usage (242 MB)

Relations

/PtH	292
/Phi	2,702
/Store	316
/Copy	481
/Clear	225
/CFG	4,525
/FILoad	8
/FIStore	69
/Pt	4,124
/AddrOf	915
/Load	2,139
/Multi	148

Lattices

/SU	390,378
/Kill	2,967

Summary

- Flix is a declarative language for static analysis
  - Inspired by Datalog, but supports lattices and functions
- Bytecode generator is first step for performance
  - Much work remains to be done

- Implementation available: http://github.com/flix
- Documentation and more: http://flix.github.io