From Datalog to FLIX: A Declarative Language for Fixed Points on Lattices

Magnus Madsen
University of Waterloo, Canada
mmadsen@uwaterloo.ca

Ming-Ho Yee
University of Waterloo, Canada
ming-ho.yee@uwaterloo.ca

Ondřej Lhoták
University of Waterloo, Canada
olhotak@uwaterloo.ca

http://flix.github.io/

PLDI 2016, Santa Barbara, USA

Motivation

Implementation of static analyses is difficult:

- analyses are often interrelated:
  - (e.g. conditional constant propagation)
  - (e.g. points-to analysis and call graph construction)
- requires a complicated arrangement of work lists.

⇒ **Hard to ensure correctness.**

⇒ **Hard to ensure performance/scalability.**

⇒ **Renewed interest in declarative programming.**

What is Declarative Programming?
(aka logic programming)

The ***what***, not the ***how***.

Find $\boldsymbol{x}$ such that:

$$3 + x = 15$$

⇒ Easy to understand whether we are solving the right problem!

What is Declarative Programming?

Separates the choice of:

- **evaluation strategy** (e.g. work list order)
- **data structures** (e.g. bitsets/hashsets/BDDs)

from the specification of the problem.

⇒ **We can leave these choices up to the solver or override them when needed.**

What is **Datalog?**

A declarative language for constraints on relations:

- Prolog-style rules (horn clauses).
- Successfully used in large-scale points-to analyses [Bravenboer et al.], [Smaragdakis et al.]

Useful theoretical and practical properties:

- Every Datalog program eventually terminates.
- Every Datalog program has a unique solution.

$\Rightarrow$ **Debugging is easier!**

Examples

- Computing your aunts and uncles:

```
AuntOrUncle(x, z) :- Parent(x, y), Brother(y, z).
AuntOrUncle(x, z) :- Parent(x, y), Sister(y, z).
```

- Computing the transitive closure of a graph:

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

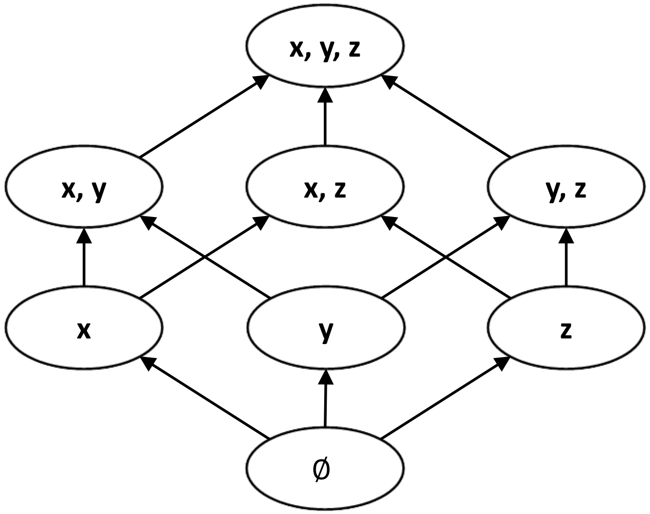
What we **can** and **can't** do in Datalog:

Analyses based on *relations*:
- Points-To
- Definite Assignment
- Reaching Definitions
- ...

Analyses based on *lattices*:
- Sign Analysis
- Constant Propagation
- Interval Analysis
- ...

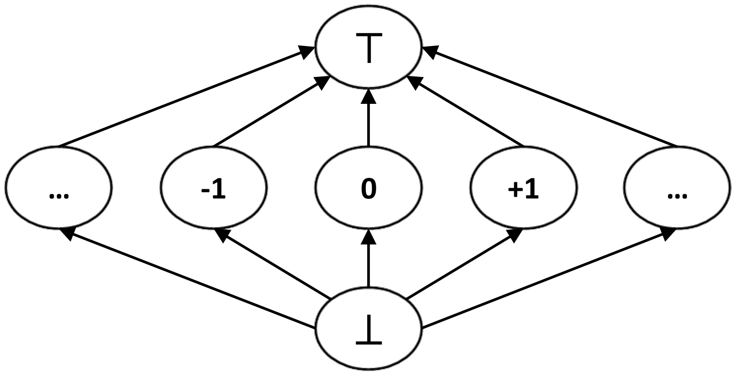
Example: Constant Propagation

What Datalog has:



fixed finite set

What we wanted:



infinite set

⇒ **We need lattices.**

⇒ **We need functions.**

Introducing **Flix**

A blend of **Logic** and **Functional** programming.

- Inspired by Datalog.
- User-defined **lattices**.
- User-defined **monotone filter** and **transfer functions**.

- Interoperates with languages on the JVM.

The Anatomy of a **Datalog** Rule

Terms: Variables or Constants

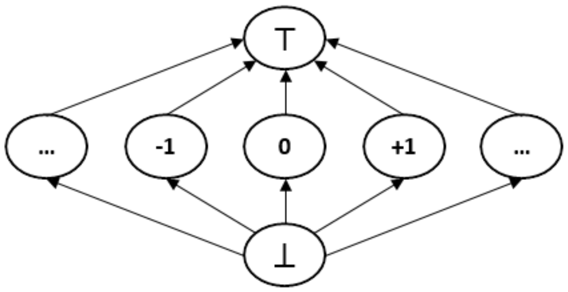
$$\overbrace{H(\bar{t})}^{\text{Head}} \Leftarrow \underbrace{B(\bar{t}), \ldots, B(\bar{t}).}_{\text{Body}}$$

Predicates

The Anatomy of a **Flix** Rule

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

Datalog vs. Flix



The **Datalog** program:

```
A("foo").
A("bar").
```

has the minimal model:

```
{A("foo"), A("bar")}
```

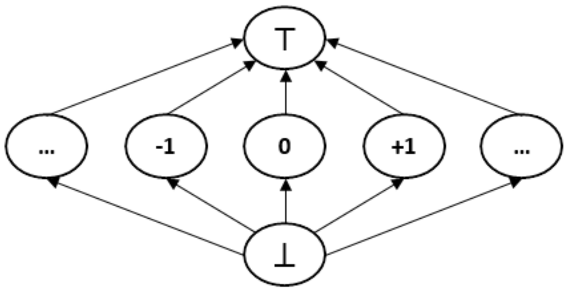
The **Flix** program:

```
A(Cst(1)).
A(Cst(2)).
B(Cst(3)).
```

has the minimal model:

```
{A(Top), B(Cst(3))}
```

Flix Semantics



The **Flix** program:

```
A(Cst(1)). B(Cst(2)).
R(x) :- A(x).
R(x) :- B(x).
```

has the minimal model:

```
{A(Cst(1)),
 B(Cst(2)),
 R(Top)}
```

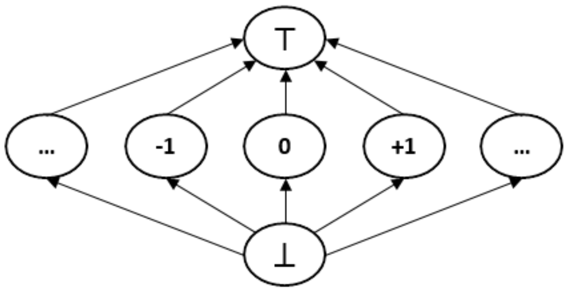
The **Flix** program:

```
A(Cst(1)). B(Cst(2)).
R(x) :- A(x), B(x).
```

has the minimal model:

```
{A(Cst(1)),
 B(Cst(2))}
```

Constant Propagation



Input Relations:

```
rel AddExp(r: Var, x: Var, y: Var)
rel DivExp(r: Var, x: Var, y: Var)
```

r = x + y

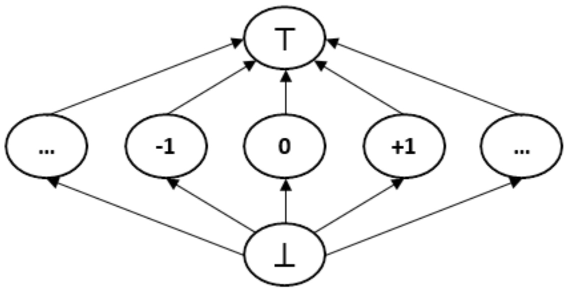
r = x / y

Computed Lattices:

```
lat LocalVar(x: Var, v: Constant)
```

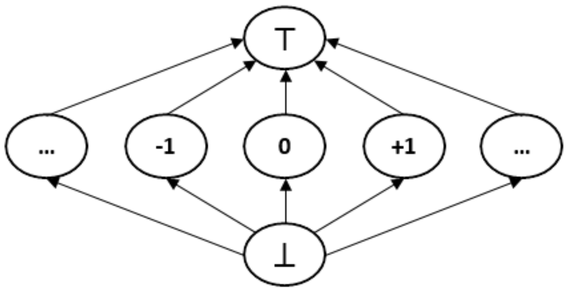
key **lattice**

Lattice Definition




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

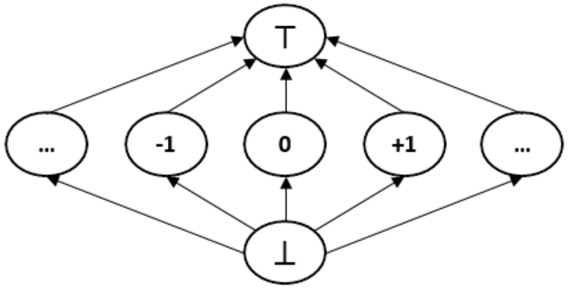
Lattice Definition




```
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
}
```

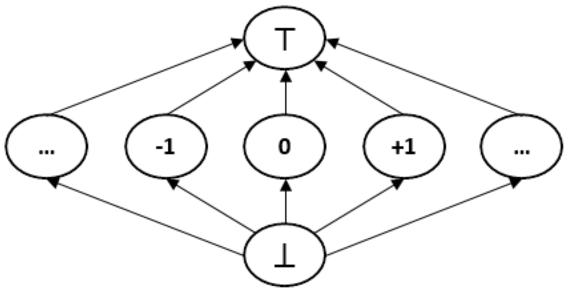
And define `lub` and `glb` in similar way…

Analysis Rules



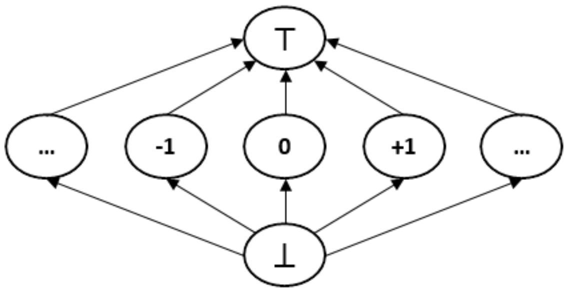

```
LocalVar(r, sum(x, y)) :- AddExp(r, v1, v2),
                          LocalVar(v1, x),
                          LocalVar(v2, y).

LocalVar(r, div(x, y)) :- DivExp(r, v1, v2),
                          LocalVar(v1, x),
                          LocalVar(v2, y).
```

Transfer Function



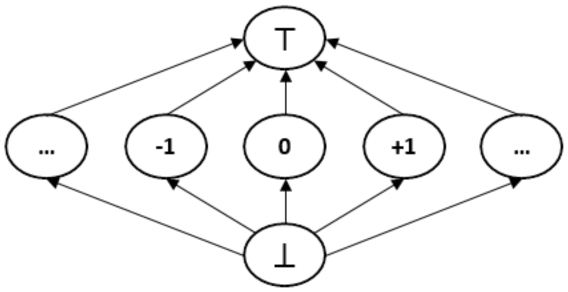

```
def sum(e1: Constant, e2: Constant): Constant
  = match (e1, e2) with {
      case (_, Bot)             => Bot
      case (Bot, _)             => Bot
      case (Cst(n1), Cst(n2))  => Cst(n1 + n2)
      case _                    => Top
    }
```

Example: Finding Bugs




```
ArithmeticError(r) :- isMaybeZero(y),
                      DivExp(r, n, d),
                      LocalVar(d, y).
```

Filter Function




```
def isMaybeZero(e: Constant): Bool
  = match e with {
      case Bot    => false
      case Cst(n) => n == 0
      case Top    => true
  }
```

Experiments

We have expressed three analyses in Flix:

- The Strong Update Analysis [Lhoták and Chung]
- The IFDS algorithm [Reps, Horwitz, and Sagiv]
  - Instantiated with the Alias-Set analysis [Naeem and Lhoták]
- The IDE algorithm [Sagiv, Reps, and Horwitz]

Strong Update Analysis

- Hybrid points-to analysis for C programs:
  - *flow-sensitive* for *singleton* points-to sets.
  - *flow-insensitive* for everything else.

Strong Update Analysis

$$
\begin{array}{llr}
\ell : p = \&a & \{a\} \subseteq pt(p) & [\textsc{AddrOf}] \\
\ell : p = q & pt(q) \subseteq pt(p) & [\textsc{Copy}] \\
\ell : *p = q & \forall a \in pt(p) \; . \; pt(q) \sqsubseteq su[\underline{\ell}](a) & [\textsc{Store}] \\
 & \forall a \in pt(p) \; . \; pt(q) \subseteq pt(a) & \\
\ell : p = *q & \forall a \in pt(q) \; . \; ptsu[\overline{\ell}](a) \subseteq pt(p) & [\textsc{Load}] \\
\ell_1 \in pred(\ell_2) & \forall a \in \mathcal{A} \; . \; su[\underline{\ell_1}](a) \sqsubseteq su[\overline{\ell_2}](a) & [\textsc{CFlow}] \\
\ell \in \mathcal{L} & \forall a \in \mathcal{A} \setminus kill(\ell) \; . \; su[\overline{\ell}](a) \sqsubseteq su[\underline{\ell}](a) & [\textsc{Preserve}]
\end{array}
$$

$$
\text{Where } ptsu[\overline{\ell}](a) \triangleq \begin{cases} su[\overline{\ell}](a) & \text{if } su[\overline{\ell}](a) \neq \top \\ pt(a) & \text{if } su[\overline{\ell}](a) = \top \end{cases}
$$

Strong Update Analysis

```
Pt(p, a) :- AddrOf(p, a).                                       [AddrOf]
Pt(p, a) :- Copy(p, q), Pt(q, a).                               [Copy]
SUAfter(l, a, Single(b)) :-                                     [Store]
  Store(l, p, q), Pt(p, a), Pt(q, b).
PtH(a, b) :- Store(l, p, q), Pt(p, a), Pt(q, b).
Pt(p, b) :- Load(l, p, q), Pt(q, a), PtSU(l, a, b).             [Load]
SUBefore(l2, a, t) :- CFG(l1, l2), SUAfter(l1, a, t).           [CFlow]
SUAfter(l, a, t) :- SUBefore(l, a, t), Preserve(l, a).          [Preserve]

PtSU(l, a, b) :- PtH(a, b), SUBefore(l, a, t), filter(t, b).
```

IFDS & IDE

Inter-procedural context-sensitive dataflow analyses:

- expressed as graph reachability problems.
- Interprocedural Finite Distributive Subset (IFDS)
  - pure graph reachability.
- Inteprocedural Distributive Environments (IDE)
  - IFDS with composition of micro-functions along the path.

- Anecdotally, these algorithms are hard to understand.

IFDS

```
declare PathEdge, WorkList, SummaryEdge: global edge set
algorithm Tabulate(G^#_IP)
begin
  Let (N^#, E^#) = G^#_IP
  PathEdge := { ⟨s_main, 0⟩ → ⟨s_main, 0⟩ }
  WorkList := { ⟨s_main, 0⟩ → ⟨s_main, 0⟩ }
  SummaryEdge := ∅
  ForwardTabulateSLRPs()
  for each n ∈ N^* do
    X_n := { d_2 ∈ D | ∃ d_1 ∈ (D ∪ { 0 }) such that ⟨s_procOf(n), d_1⟩ → ⟨n, d_2⟩ ∈ PathEdge }
  od
end

procedure Propagate(e)
begin
  if e ∉ PathEdge then  Insert e into PathEdge;  Insert e into WorkList  fi
end

procedure ForwardTabulateSLRPs()
begin
  while WorkList ≠ ∅ do
    Select and remove an edge ⟨s_p, d_1⟩ → ⟨n, d_2⟩ from WorkList
    switch n

      case n ∈ Call_p :
        for each d_3 such that ⟨n, d_2⟩ → ⟨s_calledProc(n), d_3⟩ ∈ E^# do
          Propagate(⟨s_calledProc(n), d_3⟩ → ⟨s_calledProc(n), d_3⟩)
        od
        for each d_3 such that ⟨n, d_2⟩ → ⟨returnSite(n), d_3⟩ ∈ (E^# ∪ SummaryEdge) do
          Propagate(⟨s_p, d_1⟩ → ⟨returnSite(n), d_3⟩)
        od
      end case

      case n = e_p :
        for each c ∈ callers(p) do
          for each d_4, d_5 such that ⟨c, d_4⟩ → ⟨s_p, d_1⟩ ∈ E^# and ⟨e_p, d_2⟩ → ⟨returnSite(c), d_5⟩ ∈ E^# do
            if ⟨c, d_4⟩ → ⟨returnSite(c), d_5⟩ ∉ SummaryEdge then
              Insert ⟨c, d_4⟩ → ⟨returnSite(c), d_5⟩ into SummaryEdge
              for each d_3 such that ⟨s_procOf(c), d_3⟩ → ⟨c, d_4⟩ ∈ PathEdge do
                Propagate(⟨s_procOf(c), d_3⟩ → ⟨returnSite(c), d_5⟩)
              od
            fi
          od
        od
      end case

      case n ∈ (N_p − Call_p − { e_p }) :
        for each ⟨m, d_3⟩ such that ⟨n, d_2⟩ → ⟨m, d_3⟩ ∈ E^# do
          Propagate(⟨s_p, d_1⟩ → ⟨m, d_3⟩)
        od
      end case

    end switch
  od
end
```

IDE

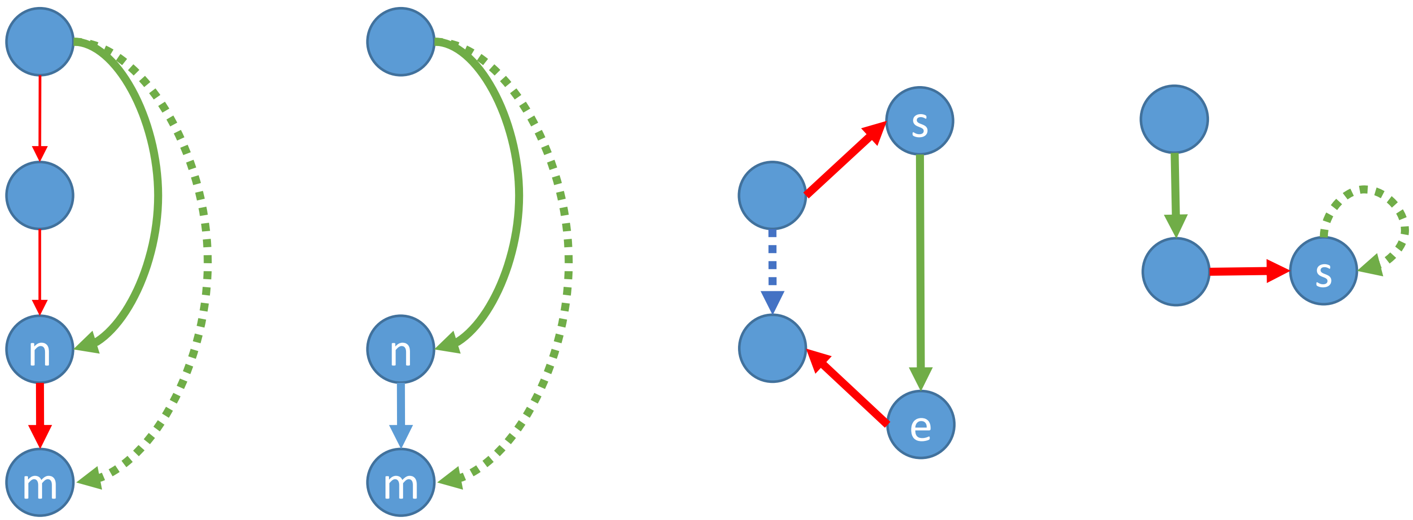
```
procedure ForwardComputeJumpFunctionsSLRPs()
  begin
    for all ⟨s_p, d'⟩, ⟨m, d⟩ such that m occurs in procedure p and d', d ∈ D ∪ {Λ} do
      JumpFn(⟨s_p, d'⟩ → ⟨m, d⟩) = λl.⊤ od
    for all corresponding call-return pairs (c, r) and d', d ∈ D ∪ {Λ} do
      SummaryFn(⟨c, d'⟩ → ⟨r, d⟩) = λl.⊤ od
    PathWorkList := {⟨s_main, Λ⟩ → ⟨s_main, Λ⟩}
    JumpFn(⟨s_main, Λ⟩ → ⟨s_main, Λ⟩) := id
    while PathWorkList ≠ ∅ do
      Select and remove an item ⟨s_p, d_1⟩ → ⟨n, d_2⟩ from PathWorkList
      let f = JumpFn(⟨s_p, d_1⟩ → ⟨n, d_2⟩)
      switch(n)
        case n is a call node in p, calling a procedure q:
          for each d_3 such that ⟨n, d_2⟩ → ⟨s_q, d_3⟩ ∈ E^♯ do
            Propagate (⟨s_q, d_3⟩ → ⟨s_q, d_3⟩, id) od
          let r be the return-site node that corresponds to n
          for each d_3 such that e = ⟨n, d_2⟩ → ⟨r, d_3⟩ ∈ E^♯ do
            Propagate(⟨s_p, d_1⟩ → ⟨r, d_3⟩, EdgeFn(e) o f) od
          for each d_3 such that f_3 = SummaryFn(⟨n, d_2⟩ → ⟨r, d_3⟩) ≠ λl.⊤ do
            Propagate(⟨s_p, d_1⟩ → ⟨r, d_3⟩, f_3 o 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 ⟨c, d_4⟩ → ⟨s_p, d_1⟩ ∈ E^♯ and ⟨e_p, d_2⟩ → ⟨r, d_5⟩ ∈ E^♯ do
              let f_4 = EdgeFn(⟨c, d_4⟩ → ⟨s_p, d_1⟩) and
                  f_5 = EdgeFn(⟨e_p, d_2⟩ → ⟨r, d_5⟩) and
                  f' = (f_5 o f o f_4) ⊓ SummaryFn(⟨c, d_4⟩ → ⟨r, d_5⟩)
              if f' ≠ SummaryFn(⟨c, d_4⟩ → ⟨r, d_5⟩) then
                SummaryFn(⟨c, d_4⟩ → ⟨r, d_5⟩) := f'
                let s_q be the start node of c's procedure
                for each d_3 such that f_3 = JumpFn(⟨s_q, d_3⟩ → ⟨c, d_4⟩) ≠ λl.⊤ do
                  Propagate(⟨s_q, d_3⟩ → ⟨r, d_5⟩, f' o f_3) od fi od od endcase
        default:
          for each ⟨m, d_3⟩ such that ⟨n, d_2⟩ → ⟨m, d_3⟩ ∈ E^♯ do
            Propagate(⟨s_p, d_1⟩ → ⟨m, d_3⟩, EdgeFn(⟨n, d_2⟩ → ⟨m, d_3⟩) o f) od endcase
      end switch od
  end

procedure Propagate(e, f)
  begin
    let f' = f ⊓ JumpFn(e)
    if f' ≠ JumpFn(e) then
      JumpFn(e) := f'
      Insert e into PathWorkList fi
  end

procedure ComputeValues()
  begin
    /* Phase II(i) */
    for each n^♯ ∈ N^♯ do val(n^♯) := ⊤ od
    val(⟨s_main, Λ⟩) := ⊥
    NodeWorkList := {⟨s_main, Λ⟩}
    while NodeWorkList ≠ ∅ do
      Select and remove an exploded-graph node ⟨n, d⟩ 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(⟨n, d⟩ → ⟨c, d'⟩) ≠ λl.⊤ do
              PropagateValue(⟨c, d'⟩, f'(val(⟨s_p, d⟩))) od od endcase
        case n is a call node in p, calling a procedure q:
          for each d' such that ⟨n, d⟩ → ⟨s_q, d'⟩ ∈ E^♯ do
            PropagateValue(⟨s_q, d'⟩, EdgeFn(⟨n, d⟩ → ⟨s_q, d'⟩)(val(⟨n, d⟩))) 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(⟨s_p, d'⟩ → ⟨n, d⟩) ≠ λl.⊤ do
        val(⟨n, d⟩) := val(⟨n, d⟩) ⊓ f'(val(⟨s_p, d'⟩)) od od
  end

procedure PropagateValue(n^♯, v)
  begin
    let v' = v ⊓ val(n^♯)
    if v' ≠ val(n^♯) then
      val(n^♯) := v'
      Insert n^♯ into NodeWorkList fi
  end
```

IFDS

- Input: the exploded super-graph.
  - The super-graph is the inter-procedural CFG.
  - The exploded super-graph is a copy of the CFG for each analysis element (in the distributive subset).

- Output: Path Edges + Summary Edges.

IFDS – Graphical Formulation



Exploded Super-Graph

Path Edge

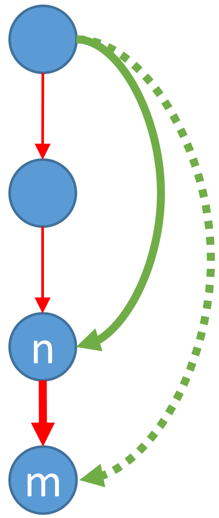
Summary Edge

Inferred Edge

IFDS

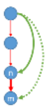
(node, element)-pair

```
PathEdge(d1, m, d3) :-
   PathEdge(d1, n, d2),
   CFG(n, m),
   d3 <- eshIntra(n, d2).
```



- The CFG is represented as a *tabulated relation*.
- The exploded super-graph (`eshIntra`) must be represented as a *function computed on-demand*.

IFDS



```
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

```
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).
```

flix/flix: The Flix Programm ×

GitHub, Inc. [US] https://github.com/flix/flix

This repository Search Pull requests Issues Gist

flix / flix

Unwatch 7 Unstar 4 Fork 0

Code Issues 32 Pull requests 0 Pulse Graphs Settings

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

3,192 commits 2 branches 2 releases 4 contributors

Branch: master New pull request Create new file Upload files Find file Clone or download

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.

Flix | The Programming La

flix.github.io

Documentation Download Getting Started GitHub

Fork me on GitHub

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

Download Documentation

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

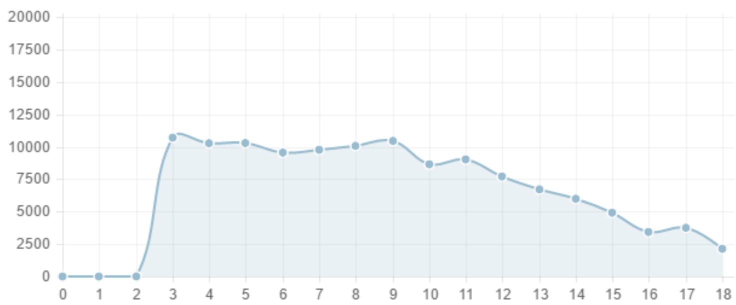
Flix (localhost:8000)

localhost:8000/#

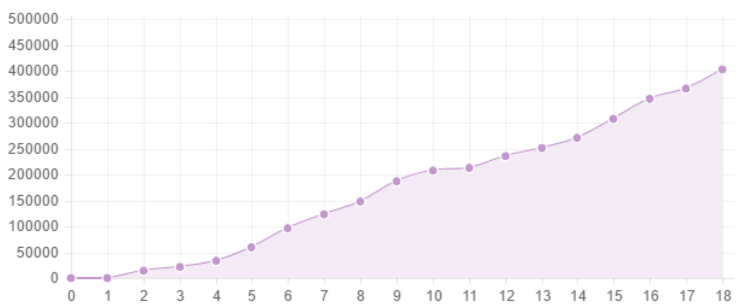
Flix Debugger | Minimal Model | Performance | Compiler | Refresh | Running

Welcome to the Flix Debugger

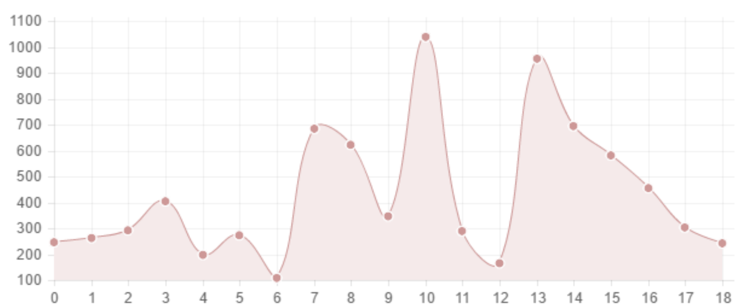
Worklist (2,130 items)



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

Flix (localhost:8000)

localhost:8000/#

Flix Debugger Minimal Model Performance Compiler Refresh Running

Performance / Rules

The table shows the number of miliseconds spent in evaluation of each rule.

Location	Rule	Hits	Total Time (msec)	Latency (msec/op)	Throughput (ops/msec)
SUopt.flix:102:1	SU(l2,a,t) :- CFG(l1, l2), SU(l1,a,t), killNot(a, k), Kill(l2,k).	918,300	18,582 msec	0.0202 msec/op	49 ops/msec
SUopt.flix:86:1	Pt(p,b) :- Load(l,p,q), Pt(q,a), filter(t, b), PtH(a,b), SU(l,a,t).	930,051	3,745 msec	0.0040 msec/op	248 ops/msec
SUopt.flix:101:1	SU(l2,a,t) :- CFG(l1, l2), SU(l1,a,t), Multi(a).	915,193	3,245 msec	0.0035 msec/op	282 ops/msec
SUopt.flix:112:1	SU(l,a,f(b)) :- Clear(l), PtH(a,b).	810	784 msec	0.9679 msec/op	1 ops/msec
SUopt.flix:118:1	Kill(l, top(42)) :- Phi(l,_,_).	1	133 msec	133.0000 msec/op	0 ops/msec
SUopt.flix:79:1	SU(l,a,f(b)) :- Store(l,p,q), Pt(p,a), Pt(q,b).	28,099	62 msec	0.0022 msec/op	453 ops/msec
SUopt.flix:81:1	PtH(a,b) :- Store(l,p,q), Pt(p,a), Pt(q,b).	28,099	50 msec	0.0018 msec/op	562 ops/msec
SUopt.flix:87:1	Pt(p,b) :- FILoad(p,q,_), Pt(q,a), PtH(a,b).	14,859	27 msec	0.0018 msec/op	550 ops/msec
SUopt.flix:82:1	PtH(a,b) :- FIStore(p,q,_), Pt(p,a), Pt(q,b).	28,099	25 msec	0.0009 msec/op	1,124 ops/msec
SUopt.flix:74:1	Pt(p,a) :- Copy(p,q), Pt(q,a).	14,050	23 msec	0.0016 msec/op	611 ops/msec
SUopt.flix:70:1	Pt(p,a) :- AddrOf(p,a).	1	17 msec	17.0000 msec/op	0 ops/msec
SUopt.flix:117:1	Kill(l,f(b)) :- Store(l,p,q), Pt(p,b).	14,050	14 msec	0.0010 msec/op	1,004 ops/msec

What's in the paper: MATH

(just a bit ...)

- From Datalog to Flix semantics:
  - explains the relationship between Datalog and Flix.
  - develops the model-theoretic semantics of Flix.
- Presents semi-naïve evaluation for Flix:
  - the basis for efficient Datalog and Flix solvers.
- Experimental results and details of analyses:
  - the Strong Update analysis, and
  - the IFDS Alias-Set analysis.

Summary: Flix!

- A new declarative and functional programming language for fixed point computations on lattices.
- Inspired by Datalog and extended with lattices and monotone transfer/filter functions.
- Implementation freely available:

  http://github.com/flix

- Documentation and more information:

  http://flix.github.io

Thank You!

This page intentionally left blank