μZ

Fix-point engine in 

Krystof Hoder
Nikolaj Bjorner
Leonardo de Moura

Fixed Points

- Fixed point of function f is an a such that $f(a)=a$
- Minimal fixed point a wrt. (partial) ordering $<$ for each fixed point a' it holds that $a \leq a'$
- For us the f is a monotonous relation transformer and a is a relation
- We can iterate f on an empty relation and when we reach $f^{n+1}(\varnothing)=f^{n}(\varnothing)$, $f^{n}(\varnothing)$ is a minimal fixed point

Fixed Points

Alternative view:

- Datalog program
- relation transformer is one iteration of bottom-up evaluation
- relation is represented by the set of derived facts

r(0,0,1).

r(x,y,z):-r(y,z,x).

$f(r)=\{(x,y,z) \mid (x,y,z)=(0,0,1) \vee r(y,z,x)\}$

{}, {r(0,0,1)}, {r(0,0,1), r(0,1,0)},
{r(0,0,1), r(0,1,0), r(1,0,0)}

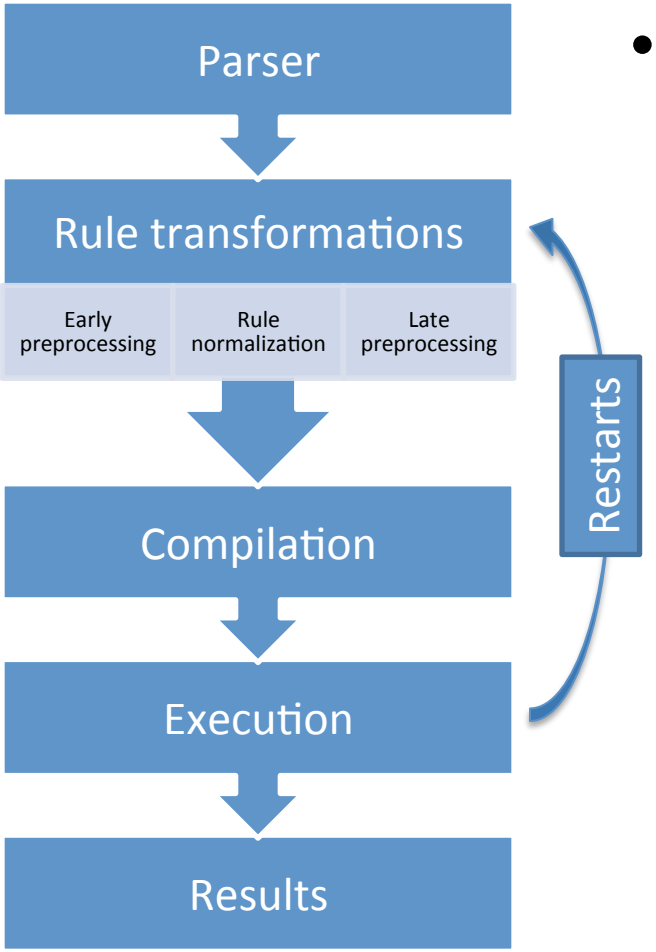
Motivation

- Horn EPR applications (Datalog)
  - Points-to analysis
  - Security analysis
  - Deductive data-bases and knowledge bases (Yago)
- Many areas of software analysis use fixed points
  - Model-checking
    - Set of reachable states is minimal fixed point
  - Abstract interpreters
    - Fixed points using approximations on infinite latices
  - Using first-order engines here requires an extra layer

μZ

- Efficient Datalog engine
- Encapsulates SMT solving using Z3
- Extensible

Architecture

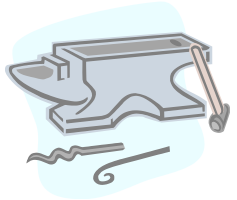


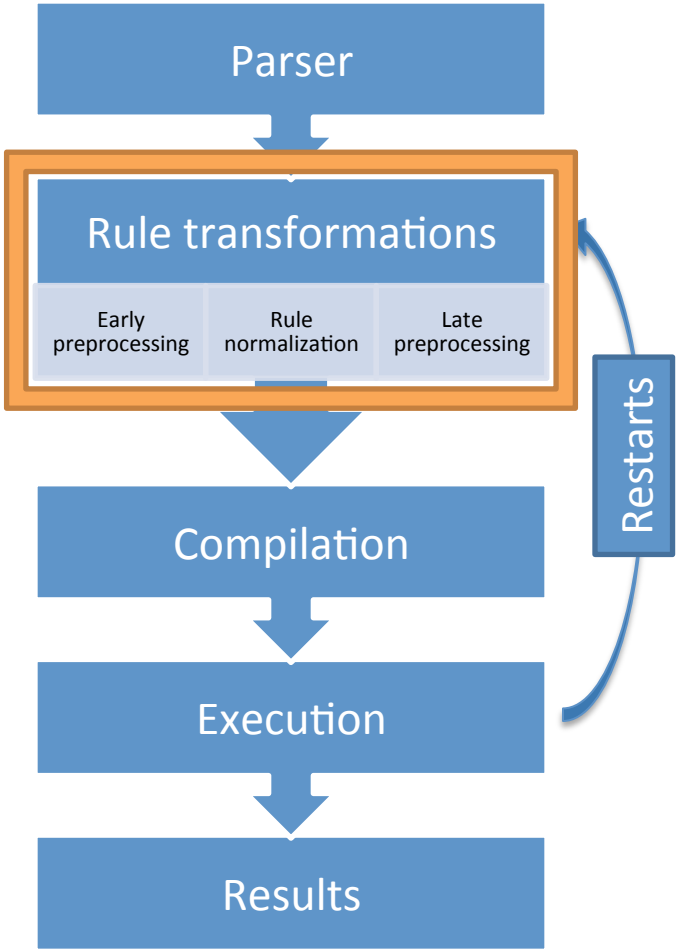
- Datalog

```
PointsTo(v2, h2) :-
        Load(v2, v1, f),
        PointsTo(v1, h1),
        HeapPointsTo(h1, f, h2).
Load("b", "global", "Function").
Prototype(”f2::N.js:33", h1) :- GlobalFunctionPrototype(h1).
Prototype(”f6::N.js:37", h1) :- GlobalFunctionPrototype(h1).
```

- Prolog without functions
- Finite domains
- Evaluation using relation algebra
  - join, project, select, union

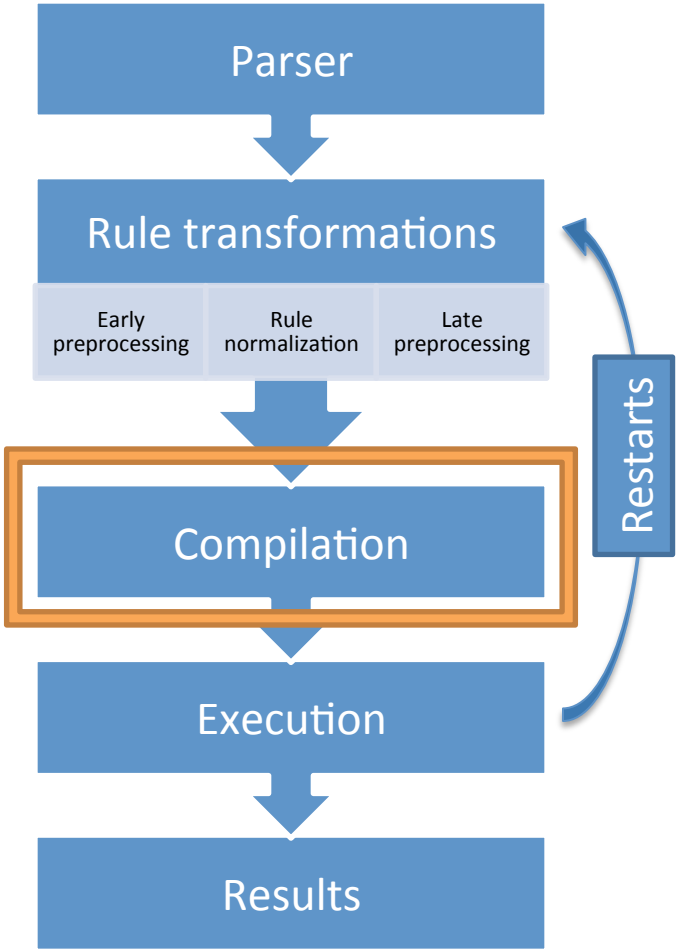
Architecture





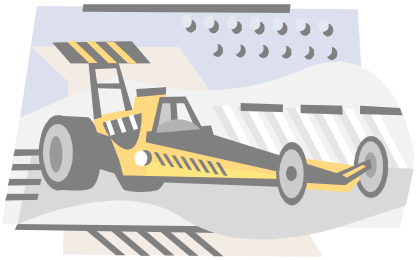
- Rule transformations
  - Normalization
    - Tail contains at most two predicates
    - Corresponds to join planning in databases
    - Identifies common subexpressions
  - Preprocessing
    - Add tracing columns if we want proofs
    - Magic Sets for goal orientation
    - Equivalent transformations of rules to improve performance
      - Inlining (non-growing)
      - Elimination of redundant arguments
  - Restarts
    - There is often little information about the relations at the beginning
    - We may restart and redo the transformations when we know more
      - e.g. sizes of relations

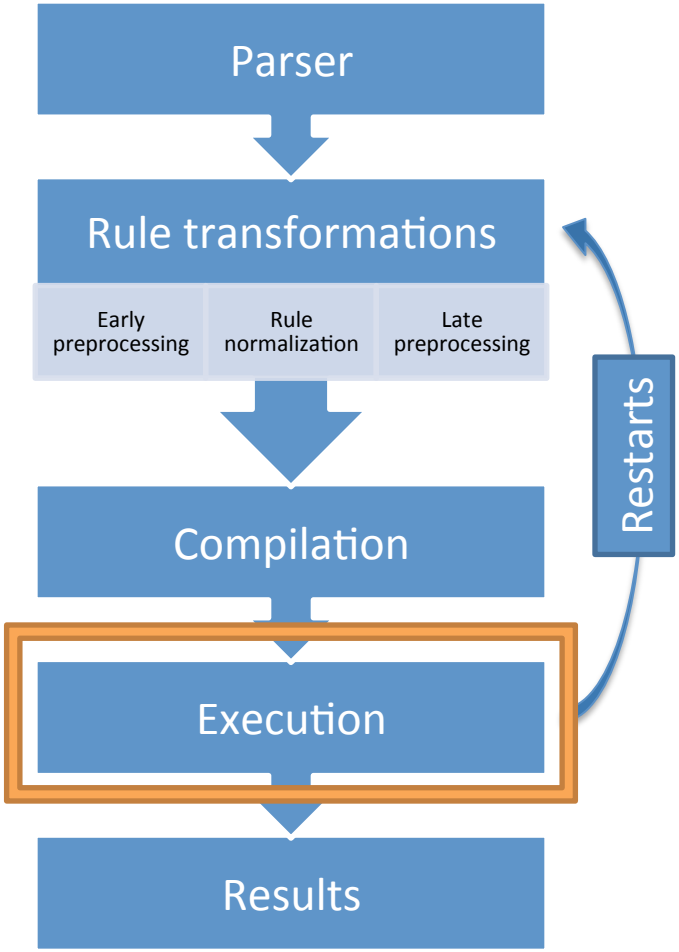
Architecture



- Compilation
  - Into register machine
  - Straightforward for non-recursive rules
  - Recursive rules stratified and compiled using delta relations
  - Compile each SCC separately
    - Split SCC into core and acyclic part
    - Compile the acyclic part like non-recursive

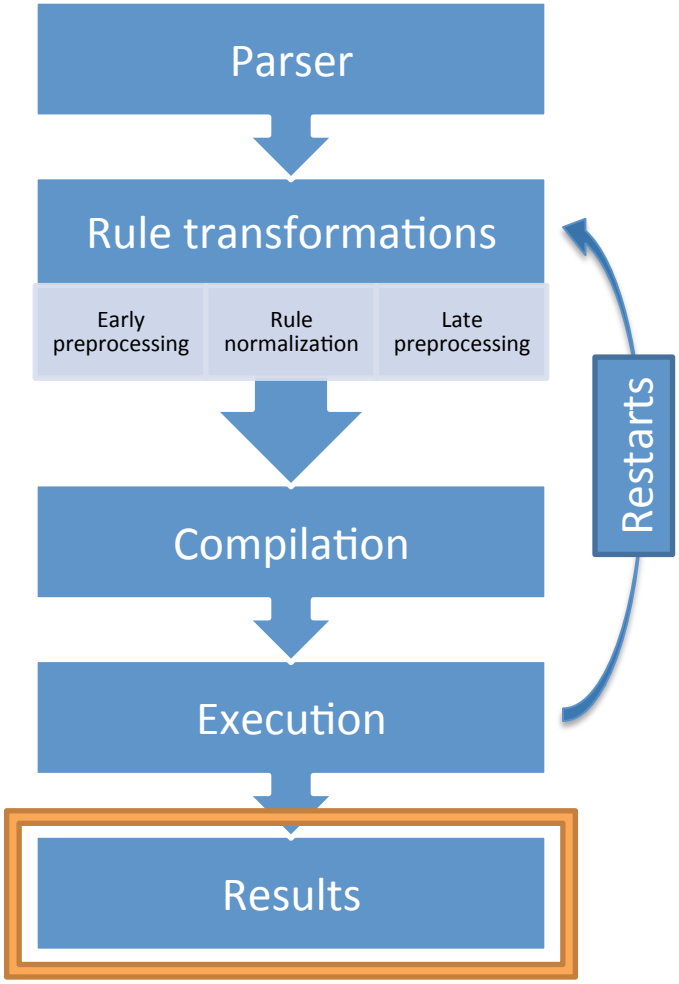
Architecture





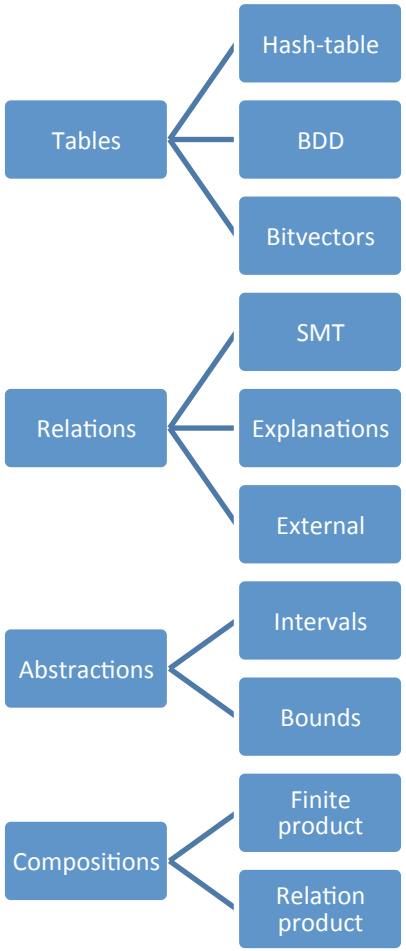
- Execution
  - Profiling data for each instruction and rule are collected
    - Profile guided rule transformations
    - Feedback to the user

Architecture



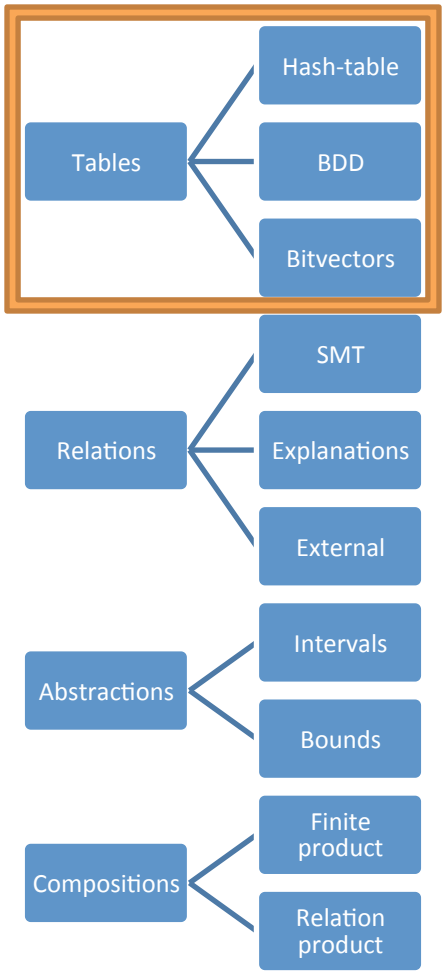
- Results of execution
  - Fixed point
  - Answers to a query
  - Possibly with an derivation tree
    - For each tuple in Finite Datalog
    - For each relation in Abstract Datalog

Relation representation



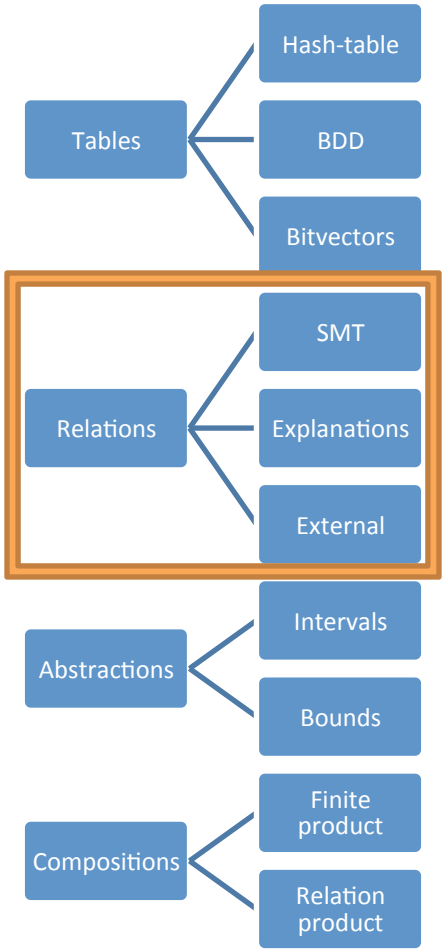
- Plugin architecture
- Plugins need to provide basic relation operations
  - Optional specialized operations for better performance
    - join-project, select-project, intersection,…

Relation representation



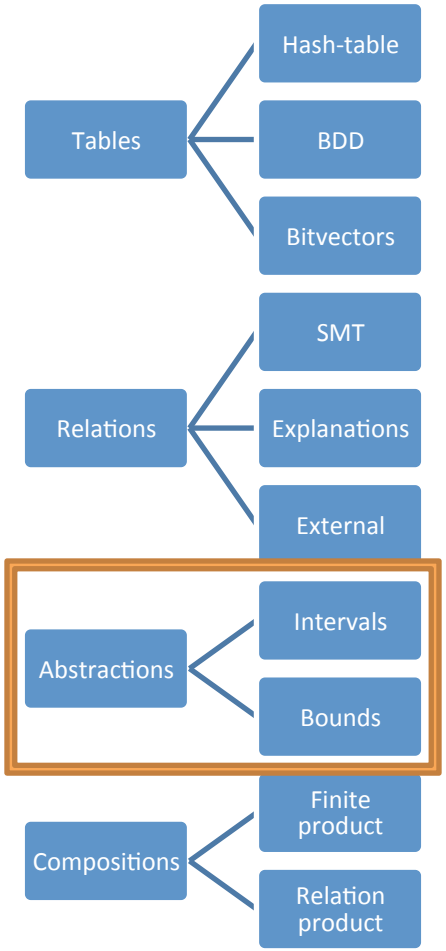
- Tables
  - Represent finite domains
  - Hash-tables
    - With indexes on subsets of columns
      - for joins, selects
  - Bitvectors
    - Small domain relations
  - BDDs
    - Good for low entropy relations

Relation representation

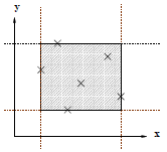


- Relations
  - Represent arbitrary domains
  - SMT relation
    - Relation operations implemented using SMT solver
    - r(1,2) <-> r_0=1 & r_1=2
    - union <-> disjunction
    - is_empty <-> is unsatisfiable
    - …
  - Explanations
    - Lightweight relation for building proof trees
  - External relations
    - User can provide their own relations using extended Z3 API

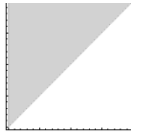
Relation representation



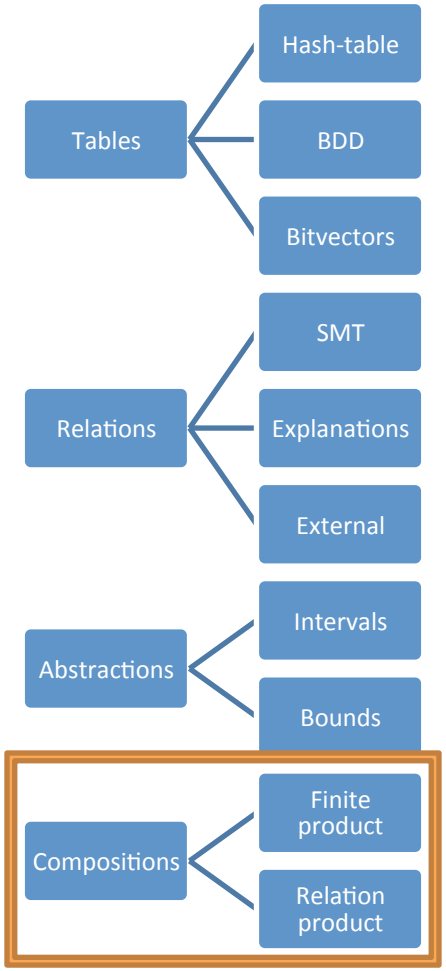
- Abstract domains
  - Relations do not need to be precise
  - Widening operations
    - Guarantee convergence of infinite domains
    - Specialized compilation mode to improve precision
  - Interval relation
    - upper and lower bound for each column



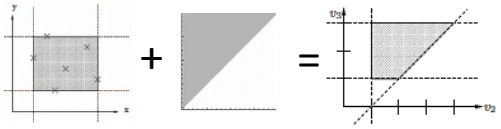
  - Bounds relation
    - inequalities between columns



Relation representation



- Compositions
  - Finite product: Table x Relation
    - Precise operations
    - Use
      - explanations for Finite Datalog
      - (possibly) context sensitivity in points-to analysis
  - Relation product: Relation x Relation
    - May be imprecise
      - relation implementations can be aware of each other to increase precision
    - Use
      - explanations for Abstract Datalog
      - combining abstract domains
      - intervals + bounds = pentagons



Rule preprocessing

Goal orientation

Removing unbound head variables

Coalescing similar rules

Inlining

- Goal orientation
  - Magic Sets
    - 1980's Datalog optimization technique

```
1<2
2<3
…
99<100
x<z :- x<y, y<z
```

- Query:
  q(x) :- x<4
- We only need part of the '<' relation
- Introduce auxiliary 'r' (reachable) relation:

```
r(4).
r(x) :- r(y), x<y
x<z :- r(y), x<y, y<z
```

- Now evaluation of '<' is restricted only to tuples that may influence the result

Rule preprocessing

- Goal orientation
- Removing unbound head variables
- Coalescing similar rules
- Inlining

• Removing unbound head variables

```
Load("vtmp1176", "vtmp1173", x).
Check(x) :- Load("vtmp1176", x, y).
=>
Load3("vtmp1176", "vtmp1173").
Check(x) :- Load("vtmp1176", x, y).
Check(x) :- Load3("vtmp1176", x).
```

- Unbound variables in head
  - Expensive for some table representations
  - Hash-table must store a tuple for each element in the domain
- Possible exponential increase of number of rules
  - Exponential with arity of relations

Benchmark	Size [statements/kb]	Untransformed	Transformed
alert_01.js	1827/390	90ms	90ms
settings.js	2636/515	130ms	100ms
prototype.js	25862/5460	2175ms	650ms

Rule preprocessing

Goal orientation

Removing unbound head variables

Coalescing similar rules

Inlining

- Coalescing similar rules

```
Prototype("f163::N.js:335", h1) :- GlobalFunctionPrototype(h1).
Prototype("f164::N.js:373", h1) :- GlobalFunctionPrototype(h1).
=>
Prototype(x, h1) :- GlobalFunctionPrototype(h1), Aux(x).
Aux("f163::N.js:335").
Aux("f164::N.js:373").
```

  - Replace several simpler rules with one more complex

Rule preprocessing

- Goal orientation
- Removing unbound head variables
- Coalescing similar rules
- Inlining

- Inlining

```
p(x):-q(x).
q(x):-r(x).
r("a").
=>
p("a").
```

  - Eliminate relations by replacing their occurrences by their definitions
  - Need to be careful to avoid blow-up
  - We inline only if it does not increase problem size
  - Often reveals unreachable rules:

```
p(x):-q("b").
q(x):-r(x).
r("a").
=>
all eliminated
```

How PDR works

```
(init (C 1 0 0 0 1 0))
(pdr-rule (=> (C a1 a2 a3 a4 b1 b2)
              (C a2 a3 a4 a1 b2 b1)))
(query (C 1 0 0 0 0 1))
```

- Builds over-approximations for states reachable up to 1, 2, … steps
  - Over-approximations represented by lemmas
  - Refinement (lemma addition) guided by counter-example search
  - When step k and k+1 have same approximations, we have inductive invariant

```
Final lemmas:
0 steps:
a1 & ~a2 & b1

1 step:
(a1 & ~a2 & b1) |
  (~a1 &
    b2 &
    a4)

2 steps:
(a1 & ~a2 & b1) |
  ( (~a1 | b1) &
    ( b2 | a3) &
    ( a4 | b1) )

3 steps:
(a1 & ~a2 & b1) |
  ( (~a1 | a2 | b1) &
    ( b2 | a3) &
    ( a4 | b1 | a2) )

4 steps:
(a1 & ~a2 & b1) |
  ( (~a1 | a2 | b1) &
    ( b2 | a3 | a1) &
    ( a4 | b1 | a2) )

5 steps:
(a1 & ~a2 & b1) |
  ( ( b2 | a3 | a1) &
    ( a4 | b1 | a2) )

6 steps:
(a1 & ~a2 & b1) |
  ( ( b2 | a3 | a1) &
    ( a4 | b1 | a2) )
```

Learning Lemmas

```
(init (C 1 0 0 0 1 0))
(pdr-rule (=> (C a1 a2 a3 a4 b1 b2)
              (C a2 a3 a4 a1 b2 b1)))
(query (C 1 0 0 0 0 1))
```

1 0 0 0 1 0
0 1 0 0 0 1
0 0 1 0 1 0
0 0 0 1 0 1

1 0 0 0 0 1
0 1 0 0 1 0
0 0 1 0 0 1
0 0 0 1 1 0

- Elementary query in PDR:
  - Is state reachable in *k* steps?
- How to answer it?
  - If violates lemmas for *k* steps, unreachable
  - Check the initial set, if found, then reachable
  - If k>0, try to find predecessor state and ask "Is reachable in *k*-1 steps?"
- When is unreachable in k steps, we may add as new lemma for *k*, *k*-1,…,0 steps
- is not very strong, we try to strengthen it
  - Find such that and is over-approximation of states reachable in *k* steps
  - Dropping literals, unreachability proof analysis

```
Final lemmas:
0 steps:
a1 & ~a2 & b1

1 step:
(a1 & ~a2 & b1) |
  (~a1 &
    b2 &
    a4)

2 steps:
(a1 & ~a2 & b1) |
  ( (~a1 | b1) &
    ( b2 | a3) &
    ( a4 | b1) )

3 steps:
(a1 & ~a2 & b1) |
  ( (~a1 | a2 | b1) &
    ( b2 | a3) &
    ( a4 | b1 | a2) )

4 steps:
(a1 & ~a2 & b1) |
  ( (~a1 | a2 | b1) &
    ( b2 | a3 | a1) &
    ( a4 | b1 | a2) )

5 steps:
(a1 & ~a2 & b1) |
  ( ( b2 | a3 | a1) &
    ( a4 | b1 | a2) )

6 steps:
(a1 & ~a2 & b1) |
  ( ( b2 | a3 | a1) &
    ( a4 | b1 | a2) )
```

Generalizations

- PDR works for ***linear*** Transformers
  - Generalize to ***non-linear***

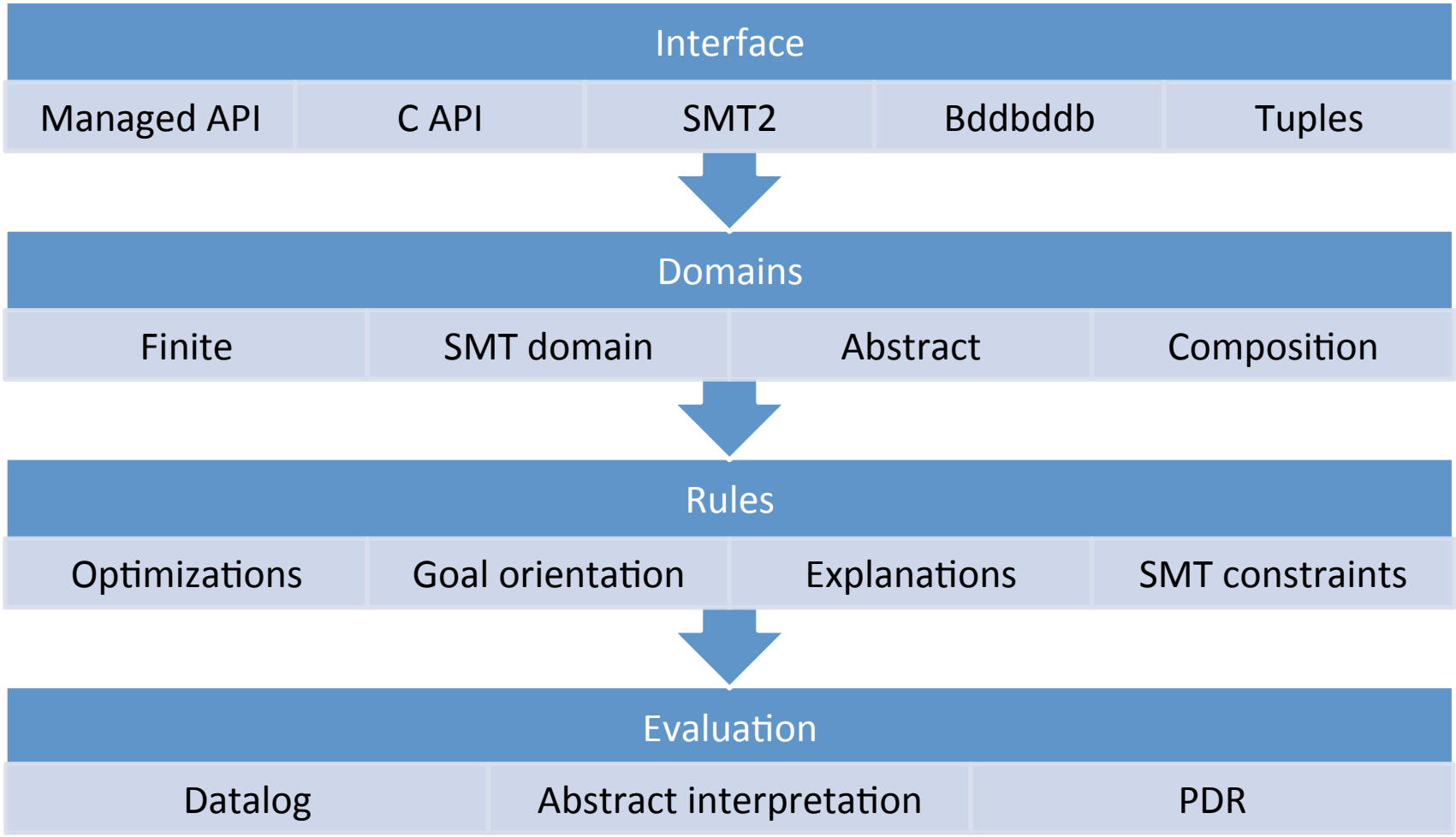
$$\mathscr{F}(R)(\vec{x}) \quad = \quad \exists \vec{y}, \vec{z} \,.\, I(\vec{x}) \vee R(\vec{y}) \wedge R(\vec{z}) \wedge T(\vec{y}, \vec{z}, \vec{x})$$

- PDR works with a ***single*** Transformer
  - Work with ***multiple*** transformers.

  $\Rightarrow$ A Solver for Datalog/Boolean Programs

- PDR is for ***propositional*** logic
  - Search ***Modulo Theories***

Summary


Interface
Managed API
C API
SMT2
Bddbddb
Tuples
Domains
Finite
SMT domain
Abstract
Composition
Rules
Optimizations
Goal orientation
Explanations
SMT constraints
Evaluation
Datalog
Abstract interpretation
PDR