Infeasible Paths Elimination by Symb. Execution Techniques:

Proof of Correctness and Preservation of Paths

Romain Aissat, Frederic Voisin and Burkhart Wolff

Univ - Paris-Sud / LRI

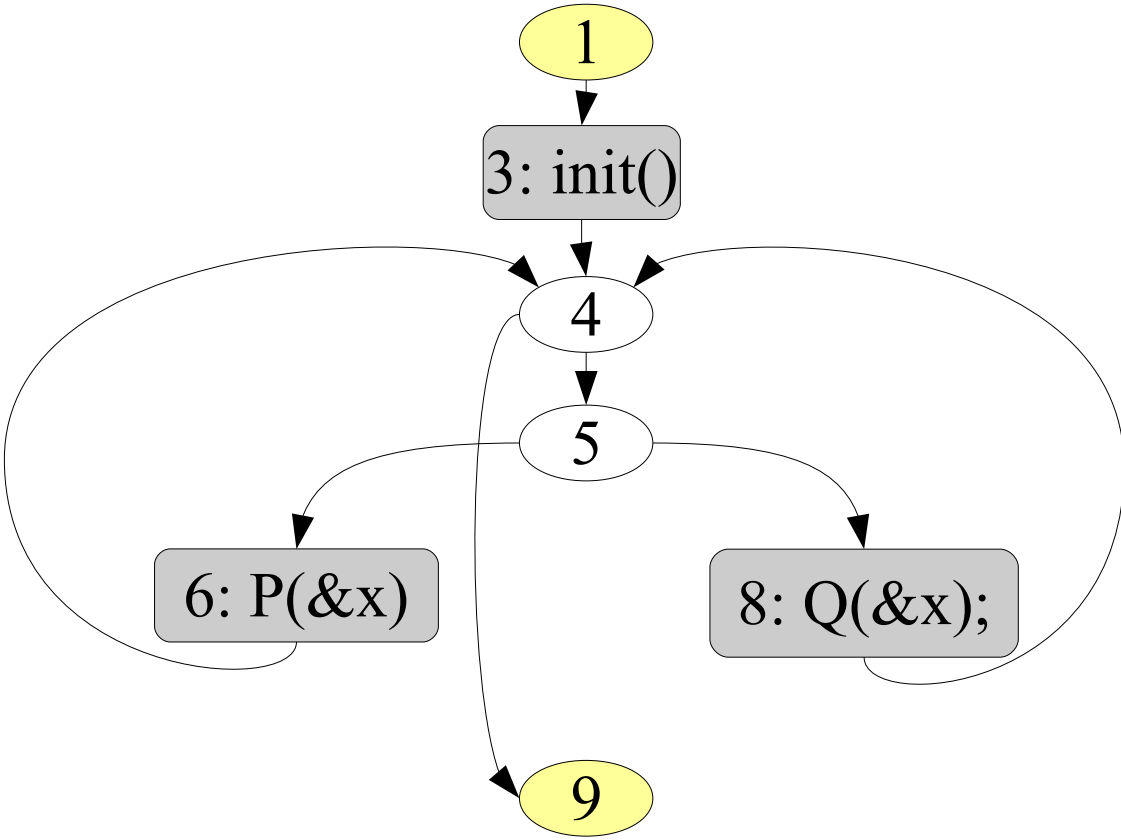
Abstract

TRACER [8] is a tool for verifying safety properties of sequential C programs. TRACER attempts at building a finite symbolic execution graph which over-approximates the set of all concrete reach- able states and the set of feasible paths.
We present an abstract framework for TRACER and similar CEGAR-like systems [2, 3, 5, 6, 9]. The framework provides 1) a graph-transformation based method for reducing the feasible paths in control-flow graphs, 2) a model for symbolic execution, subsumption, predicate abstraction and invariant generation. In this framework we formally prove two key properties: correct construction of the symbolic states and preservation of feasible paths. The framework focuses on core operations, leaving to concrete prototypes to “fit in” heuristics for combining them.

Introduction: Control Flow Paths(CFG)

- A simple C program and its control-flow graph

```
1     void f(int x, bool b)
2     {
3        init();
4        while (x > 0){
5           if(b)
6              {P(&x); }
7           else
8              {Q(&x);}
9        return (x==0);
10    }
```



Paths of max 2 loop traversals: {[1,3,4,9], [1,3,4,5,6,4,9], [1,3,4,5,8,4,9], [1,3,4,5,6,4,5,6,4,9], [1,3,4,5,8,4,5,8,4,9],[1,3,4,5,6,4,8,4,9], [1,3,4,5,8,4,5..]}

Introduction : Control Flow Paths

- Not all paths in the CFG are actually *feasible*, i.e. possible wrt. to the operational semantics for concrete input values for *x* and *b*.

- Actually, [1,3,4,5,6,4,8,4,9], [1,3,4,5,8,4,5,6,4,9] are *infeasible*, ie no input exists that makes this execution possible (assuming C semantics)

Introduction : Control Flow Paths

- Worse:

  number of paths with k loop traversals : 2^k
  number of feasible paths ¨ ¨ ¨ : 2 * k

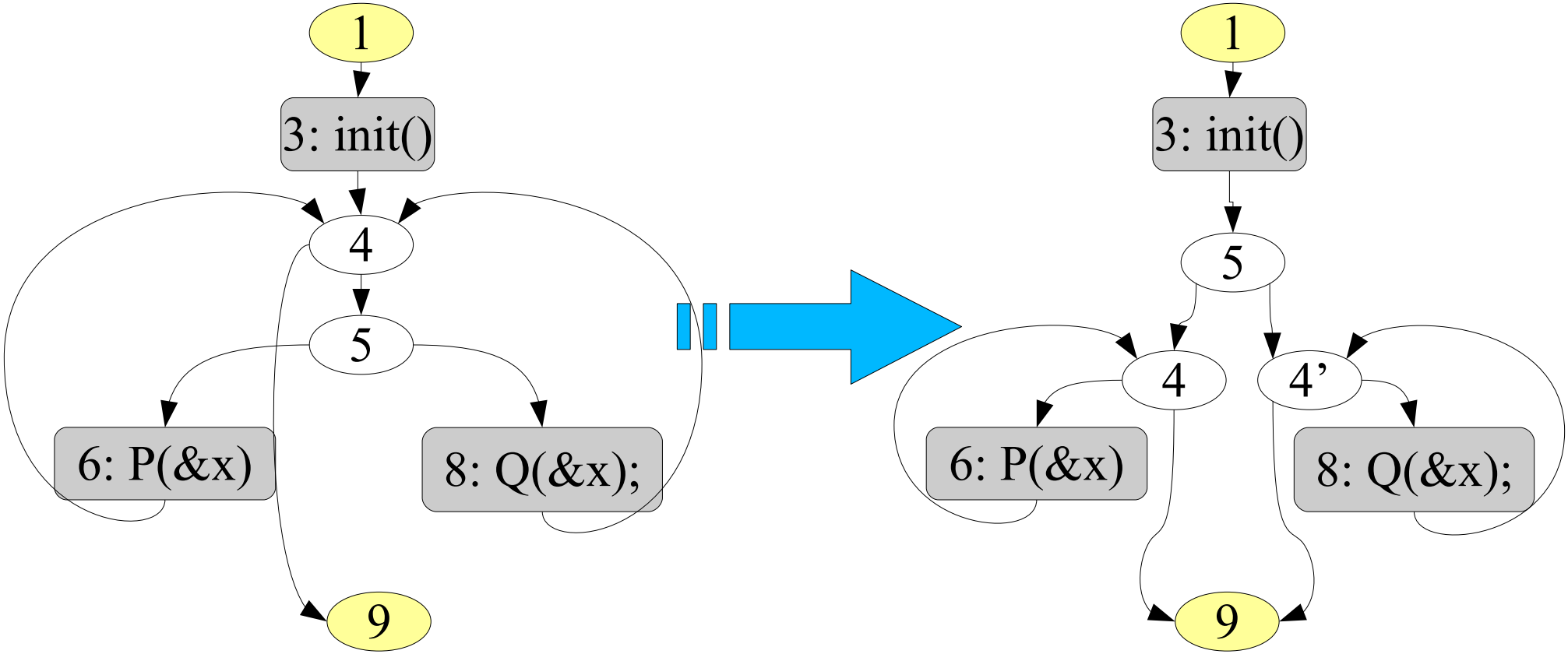
  So, the probability for picking randomly
  a feasible path decreases asymptotically to 0

- Even worse: experiments show, that this gap is typical in practical programs.

  This is the source of inefficiency of many static analysis techniques: symb exec testing, abstract interpretation, random testers ...

Introduction: Blue Calculus

- Remedy: Transformation of the CFG



After transformation: No infeasible paths any more . . .

Contributions

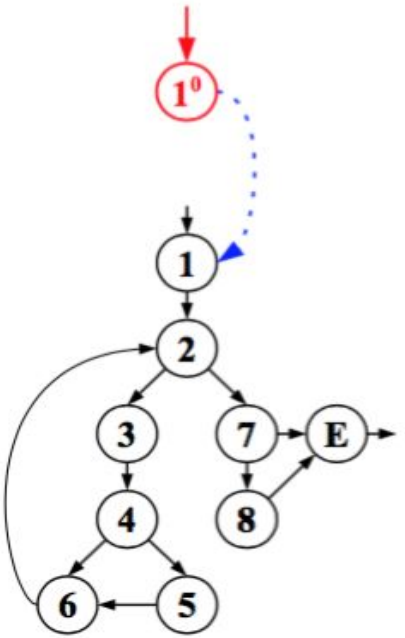
- Rational Reconstruction of TRACER [Jaffar et al, CAV12] into a formal model ATRACER
- ATRACER presents TRACER by 6 (non-deterministic) Graph-Transformation Rules on Red-Black-Graphs*
- Proof in Isabelle/HOL: Derivations in ATRACER preserve semantics
- Proof in Isabelle/HOL: feasible paths should be preserved
- There should be less infeasible paths in general**

* no heuristics modeled

**which is experimentally confirmed, but we can't give strong guarantees

Red-Black Graphs

```
1  lock=0; new=old+1
2  while(new!=old){
3       k=1; old=new
4       if(*){lock=0;
                new=new+1;}
6  };
7  if (lock==0)
8       error()
```

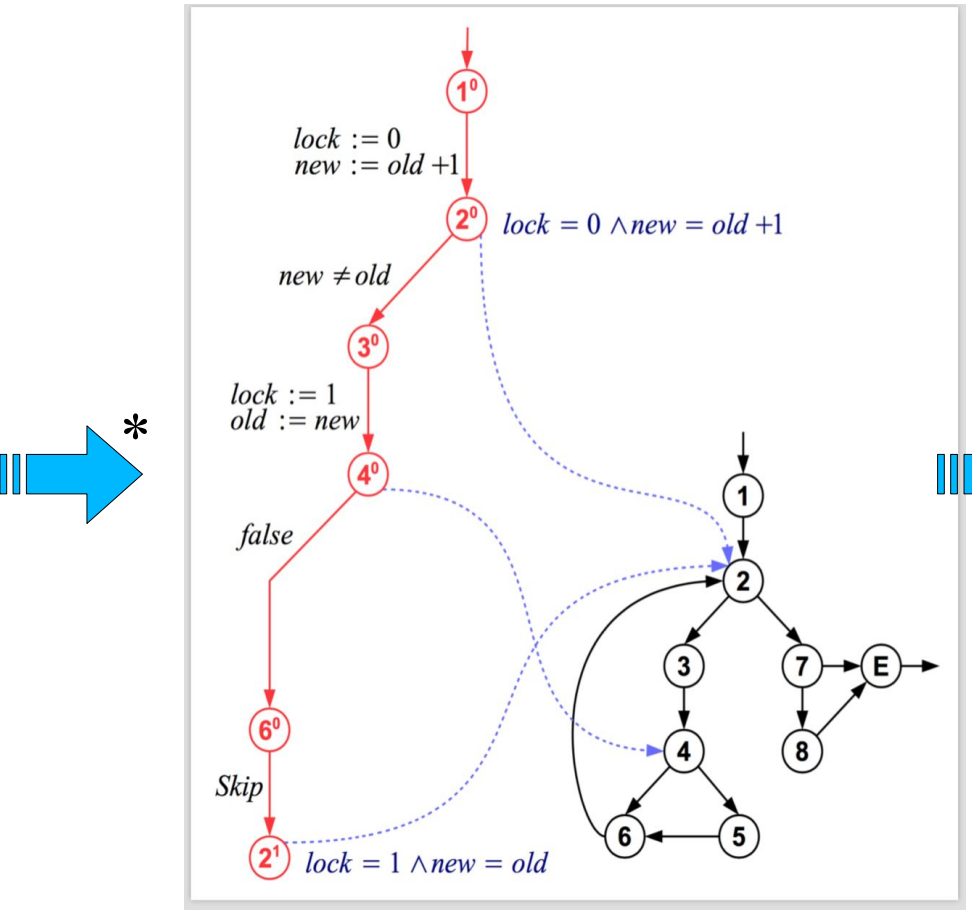


Graph Transformation Rules:

1) init

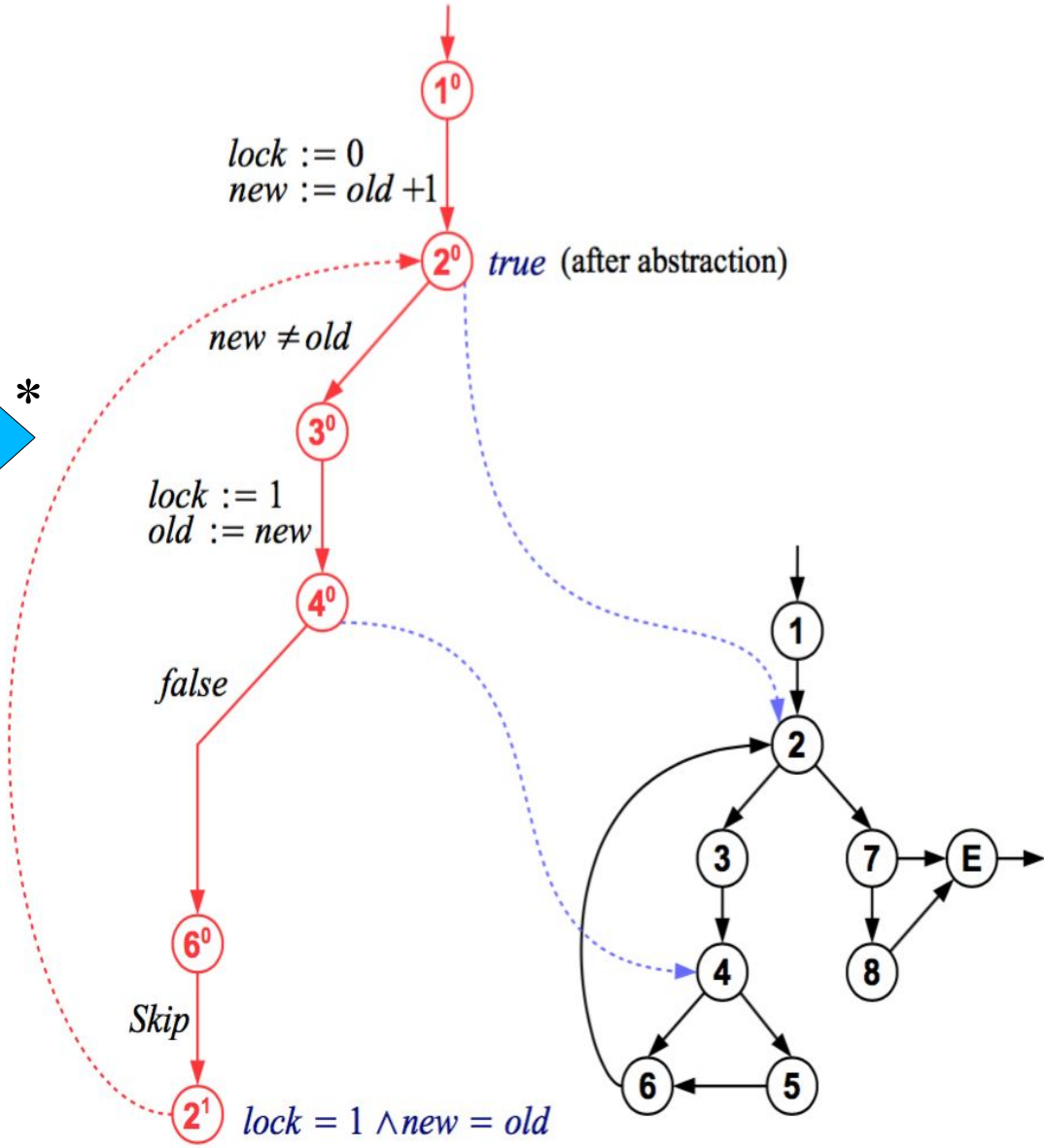
...

Red-Black Graphs



*

*



Graph Transformation Rules:

...

2) symbolic execution assign
3) symbolic execution assume
4) abstraction
5) subsumption
6) cut - rule (for infeasible paths)

Formalisation

- Tasks:
  - Red-Black Graphs, Paths, Fringes
  - Labelled Transition Systems
  - States, Configurations, Symbolic Execution
  - Formalizing Graph Transformation Step Relation as Inductive Definition
  - Proof of Correctness and Preservation

Formalisation (1)

- Basic Machinery
  Raw graphs, labelled transition systems:

```
record 'a arc = src :: "'a"
                tgt :: "'a"

record 'a rgraph = root :: "'a"
                   arcs :: "'a arc set"

record ('a,'b,'c) lts = "'a rgraph" + labf :: "'a arc ⇒ ('b,'c) label"
```

  - coherent arc sequences

```
fun cas :: "'a ⇒ 'a arc list ⇒ 'a ⇒ bool"
where
  "cas v1 [] v2 = (v1 = v2)"
| "cas v1 (a#seq) v2 = (src a = v1 ∧ cas (tgt a) seq v2)"
```

  - paths

```
"subpath g v1 seq v2 ≡ cas v1 seq v2 ∧ v1 ∈ verts g ∧ set seq ⊆ arcs g"
```

Formalisation (2)

- Execution Data:
  states, stores, (shallow) expressions :

```
type_synonym ('b,'c) state = "'b ⇒ 'c"
type_synonym ('b,'c) aexp  = "('b,'c) state ⇒ 'c"
type_synonym ('b,'c) bexp  = "('b,'c) state ⇒ bool"
```

  - framing :

```
definition vars :: "('b,'c) aexp ⇒ 'b set" where
  "vars e = {v. ∃σ val. e (σ(v := val)) ≠ e σ}"
```

  - program expressions as core syntax :

```
datatype ('b,'c) label =
  Skip | Assume "('b,'c) bexp" | Assign 'b "('b,'c) aexp"
```

Formalisation (3)

- Red-Black-Graphs:
  - ```
    record ('a,'b,'c) pre_RedBlack =
      red           :: "('a × nat) rgraph"
      black         :: "('a,'b,'c) lts"
      subs          :: "(('a × nat) × ('a ×  nat)) set"
      init_conf     :: "'b conf"
      confs         :: "('a × nat) ⇒ ('b,'c) conf"
      marked        :: "('a × nat) ⇒ bool"
      strengthenings :: "('a × nat) ⇒ ('b,'c) bexp"
    ```
  - red part
  - black part
  - the set of subsumption links
  - initial configuration (contains precondition if any)
  - mapping for symbolic variables to additional constraints
    ```

Formalisation (4)

- Example (out of 6): Symbolic Execution.

```
"se_extends rb ra c' rb' ≡
   ui_arc ra ∈ arcs (black rb)                                    (* 1 *)
 ∧ ArcExt.extends (red rb) ra (red rb')                           (* 2 *)
 ∧ src ra ∉ subsumees (subs rb)                                   (* 3 *)
 ∧ se (confs rb (src ra)) (labf (black rb)(ui_arc ra)) c'         (* 4 *)
 ∧ rb' = (| red       = red rb',
            black     = black rb,
            subs      = subs rb,
            init_conf = init_conf rb,
            confs     = (confs rb) (tgt ra := c'),
            marked    = (marked rb)(tgt ra := marked rb (src ra)),
            strengthenings = strengthenings rb |)                 (* 5 *)"
```

- ui_arc ra, the (unindexed) black counterpart of red arc ra must exist in the black graph,
- ArcExt.extends is an abbreviation that states that the source of ra must be an existing vertex of the red graph, but not its target, and that the new red graph is obtained by adding ra to the arcs of the old one,
- the source of ra is is not already subsumed,
- c' is the new configuration obtained by symbolic execution of ra
- the new red-black graph rb' is constructed from the old one by the resp. updates

Formalisation (5)

- The Blue Calculus:

```
inductive RedBlack :: "('a,'b,'c) pre_RedBlack ⇒ bool" where
  init :
    "fst (root (red rb)) = init (black rb)          ⟹
     arcs (red rb) = {}                             ⟹
     subs rb = {}                                   ⟹
     (confs rb) (root (red rb)) = init_conf rb      ⟹
     marked rb = (λ rv. False)                      ⟹
     strengthenings rb = (λ rv. (λ σ. True))        ⟹ RedBlack rb"
| se_step :
    "RedBlack rb ⟹ se_extends rb ra c' rb'          ⟹ RedBlack rb'"
| mark_step :
    "RedBlack rb ⟹ mark_extends rb rv rb'           ⟹ RedBlack rb'"
| subsum_step :
    "RedBlack rb ⟹ subsum_extends rb sub rb'        ⟹ RedBlack rb'"
| abstract_step :
    "RedBlack rb ⟹ abstract_extends rb rv e rb'     ⟹ RedBlack rb'"
| strengthen_step :
    "RedBlack rb ⟹ strengthen_extends rb rv e rb'  ⟹ RedBlack rb'"
```

Proof (1)

- Lemma: red paths lead always to weaker configurations:

```
theorem gt_calc_se_rel :
assumes "RedBlack rb"
assumes "subpath (red rb) r1 s r2 (subs rb)"
assumes "se_star (confs rb r1) (trace (ui_as s) (labelling (black rb))) c"
shows   "c ⊑ (confs rb r2)"
```

- Lemma: All red-black sub-paths included in the corresponding “pure black” path-sets:

```
theorem gt_calc_correct :
assumes "RedBlack rb"
shows   "RedBlack_subpaths_from rb rv
         ⊆Graph.subpaths_from (black rb) (fst rv)"
```

Proof (2)

- Main Result: all feasible paths were preserved by the “blue calculus”:

```
theorem gt_calc_preserves2 :
assumes "RedBlack rb"
shows   "feasible_paths (black rb) (init_conf rb) ⊆ RedBlack_paths rb"
```

Development Effort

- Effort for entire theory development:
  - 12 theories,
  - 7932 loc
  - main proof:
    2000 loc
    Isar-style
    highly struc-
    tured proof
  - proof techniques:
    standard in-
    duction proofs;
    but many, many
    cases to consider

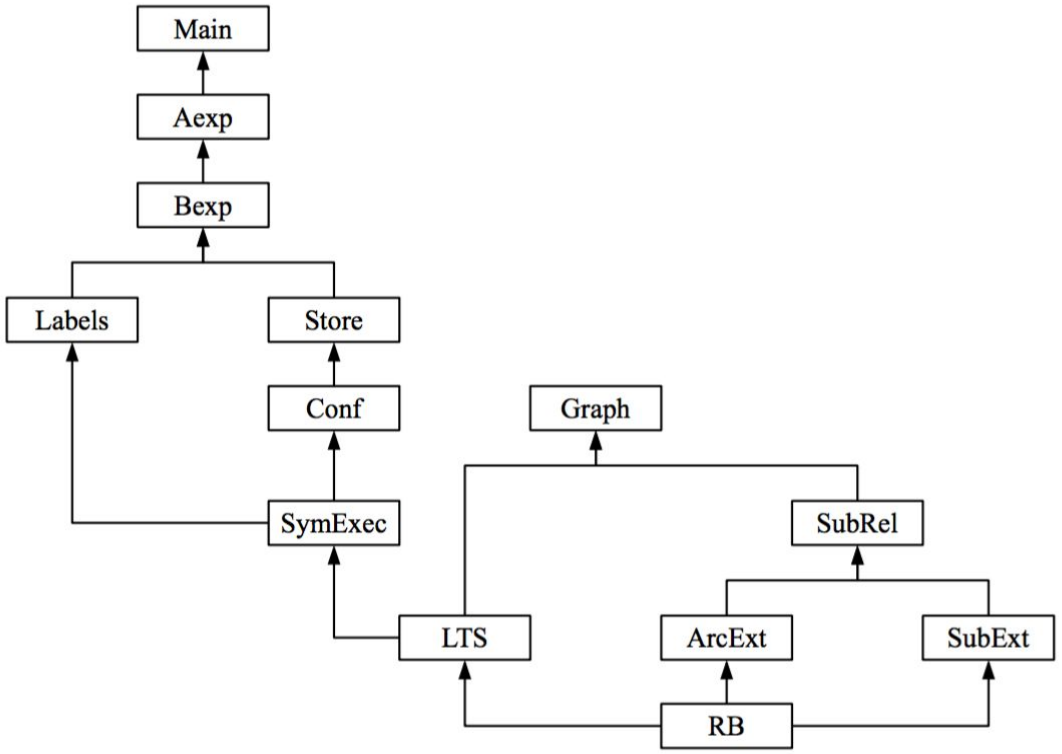


Figure 1: The hierarchy of theories.

Experimental Evaluation(1)

- The framework was implemented in an OCaml prototype, providing also some first heuristics.
- Mergesort:

Parameters:

- *a*: abstraction method (1: constraints removal, 2: symbolic var. updates),
- *r*: restarts (✓: enabled),
- *la*: look-ahead depth.

Columns:

- *l*: length of paths,
- *P*: number of paths,
- *FP*: number of feasible paths.

	a	*r*	*la*	$l = 30$		$l = 50$		$l = 100$	
				P	*FP*	*P*	*FP*	*P*	*FP*
S				1 224	140	24 434	2 300	∼ 25.8M	∼ 2.3M
S′	1 & 2	✓	0	317		5 577		∼ 5.8M	
	1	✓	2	140		2 300		∼ 2.3M	
	2	✓		140		4 652		∼ 23.9M	
	2			210		3 271		∼ 3.2M	

Experimental Evaluation(2)

- Bubblesort:

Parameters:

- a: abstraction method (1: constraints removal, 2: symbolic var. updates),
- r: restarts (✓: enabled),
- la: look-ahead depth.

Columns:

- l: length of paths,
- P: number of paths,
- FP: number of feasible paths.

	a	r	la	$l = 30$		$l = 50$		$l = 100$
				P	FP	P	FP	P
S				1 474	20	643 692	217	$\sim 2.3 \times 10^{12}$
S'	1	✓	2	741		32 962		$\sim 1.2 \times 10^{12}$
	2	✓		203		44 504		$\sim 2.9 \times 10^{11}$
	1	✓	8	285		69 457		$\sim 6.6 \times 10^{10}$
	2	✓		103		13 249		$\sim 6.4 \times 10^{9}$

Conclusion

- first known formal proof for a predicate abstraction framework.
- available in AFP soon.
- widely appliquable for enhancements many static analysis techniques
- Main difficulty: working on graphs and giving it enough inductive structure: red part, blue calc.
- Many open research problems, including heuristics, code-generation, calls in the labelling language.