Art of Invariant Generation applied to Symbolic Bound Computation

Part 1

Sumit Gulwani
(Microsoft Research, Redmond, USA)

Oregon Summer School
July 2009

Outline

- Symbolic Bound Computation Problem
  - Motivation, Definition, Reduction to Invariant Generation

- Art of Invariant Generation
  - Colorful Logic
  - Fixpoint Brush
  - Program Transformations

- Application to Symbolic Bound Computation Problem

Motivation: Bound Computation

Program execution consumes physical resources.

- Time
- Memory
- Network Bandwidth
- Power

Bounding such resources is important.

- Economic reasons
- Environment might have hard resource constraints.

Bounding such resources requires computing bound on # of visits to control-locations that consume such resources.

Motivation: Bound Computation

Program execution affects quantitative properties of data.

- Secrecy: information leakage.
- Robustness: error/uncertainty propagation.

Bounding such properties is important for correctness.

Bounding such properties requires computing bound on # of visits to control-locations that affect properties of the data.

Motivation: Static Computation of Worst-case Bound

- Provide immediate feedback during code development
  - Use of unfamiliar APIs
  - Code Editing

- Identify corner cases (unlike profiling)

Symbolic Bound Computation: A Quantitative Problem

Let π be a control-location inside a procedure P with inputs X. Let Visits(X) denote the number of visits to π when P is invoked with X.

Symbolic Bound: An integer valued expression B(X) is a symbolic bound if it upper bounds Visits(X).

Precision of a Symbolic Bound

Relative Precision: A symbolic bound B1(X) is more precise than B2(X) if $\forall X$: $B1(X) \leq B2(X)$

Absolute Precision: A symbolic bound is precise if there exists a worst-case family of inputs W(X) that realizes the bound (upto multiplicative/additive constants c_1/c_2)

- $\forall X$ satisfying W(X): $(B(X)/c_1) - c_2 \leq \text{Visits}(X) \leq B(X)$
  - Relaxing the condition $c_1 = 1$ and $c_2 = 0$ is required since it would be practically impossible to find closed-form representations of Visits(X). But it still ensures that the bound B(X) is asymptotically tight.
- $\forall k>0$: $\exists X$ such that $W(X) \wedge B(X) \geq k$
  - The family W(X) describes inputs that lead to increasingly larger evaluations for the bound expression.

Example

```
Inputs: int n, bool[] A
i := 0;
while (i < n)
      j := i+1;
      while (j < n)
            π₁: if (A[j]) { π₂: ConsumeResource(); j--; n--; }
                j++;
      i++;
```

- n^2 is a precise bound for # of visits to π_1.
  - Precision Witness: $W = \forall j(1 \leq j \leq n \Rightarrow \neg A[j])$, $c_1 = 4$, $c_2 = 0$
- n is a precise bound for # of visits to π_2.
  - $W = \forall j(1 \leq j \leq n \Rightarrow A[j])$, $c_1 = 1$, $c_2 = 0$

Bound Computation vs. Safety/Liveness Checking

- Safety: Is π never visited?
  - Violation is a finite trace
- Liveness: Is π visited finite number of times?
  - Violation is an infinite trace
- Bound Computation: Bound on maximum visits to π.
  - Quantitative question as opposed to Boolean!
  - How about checking validity/precision of a given bound?
- Checking Validity of Bound
  - Safety property
- Checking Precision of Bound (given constants c_1, c_2)
  - Not even a trace property!
  - Given precision witness, realization check is safety property.

Our Approach to (Precise) Bound Computation

- Different solutions possible that form a lattice with $\leq$ as partial order and Max/Min as LUB/GLB operators.

- We show how to reduce bound computation to invariant generation. The more powerful the invariant generator, the more precise the bound.
  - We will study design of relevant invariant generator tools.
  - These are general principles useful for other applications too.

Reducing Bound Computation to Invariant Generation

Inputs: int n

	S_1
π:	S_2
	S_3

$\longrightarrow$

	c := 0;
	S_1
π:	c++; S_2
	S_3

Claim: If $c < F(n)$ is an invariant at π, then
Max(0,F(n)) is a bound on Visits(π).

Importance of Max Rule

Corollary: If c<F(n) is a loop invariant, then Max(0,F(n)) is an upper bound on number of loop iterations.

If we instead claim F(n) to be an upper bound, we get an unsound conclusion. Consider, for example:

```
Test(int n1, int n2)
    int c1:=0; while (c1<n1) c1++;
    int c2:=0; while (c2<n2) c2++;
```

- c1 < n1 is a loop invariant. Suppose we regard n1 to be an upper bound for first loop. (Similarly, for c2 and n2).
- Thus, n1+n2 is an upper bound for Test procedure.
  - But this is clearly wrong when say n1=100 and n2=-100.

Example: Bound Computation from Invariants

```
Inputs: int n
c := 0;
x := 0; y := n;
while (x < y)
    c++;
    π:
    if (*) x := x+2;
    else y := y-2;
```

- Consider the inductive loop invariant: $2c = x+(n-y) \wedge x<y$
- Projecting out x and y yields $c < n/2$.
- Thus, $\mathrm{Max}(0,n/2)$ is an upper bound on Visits(π).

Language of Bound Expressions

- Max Operator: Control-flow/Choice between paths
- Addition Operator: Sequencing/Multiple paths
- Non-linear Operators
  - Multiplication: Nested loops
  - Logarithm: Binary search
  - Exponentiation: Recursive procedures
- Quantitative Attributes: Iteration over data-structures
  - Number of nodes in a list, or a list of lists
  - Number of nodes in a tree/Height of a tree
  - Number of bits in a bit-vector

Language of Invariants Required

Invariants required may be:

- Non-linear
- Disjunctive
- Refer to numerical properties of data-structures

A universal precise invariant generator does not exist!

We will study principles of invariant generation, and then apply a variety of these techniques to our problem.

Art of Invariant Generation

1. Program Transformations

   – Reduce need for sophisticated invariant generation.
   – E.g., control-flow refinement, loop-flattening/peeling, non-standard cut-points, quantitative attributes instrumentation.

2. Colorful Logic

   – Language of Invariants
   – E.g., arithmetic, uninterpreted fns, lists/arrays

3. Fixpoint Brush

   – Automatic generation of invariants in some shade of logic, e.g., conjunctive/k-disjunctive/predicate abstraction.
   – E.g., Iterative, Constraint-based, Proof Rules

Colorful Logic

We will briefly study decision procedures for following logics.

- Linear Arithmetic
- Uninterpreted Functions
- Linear Arithmetic + Uninterpreted Functions
- Theory of Arrays
- Theory of Lists
- Non-linear Arithmetic

Decision Procedures

$\text{Decide}_T(\phi)$ = Yes, if ϕ is satisfiable
= No, if ϕ is unsatisfiable

Without loss of generality, we can assume that ϕ is a conjunction of atomic facts.

- Why?
  - $\text{Decide}(\phi_1 \vee \phi_2)$ is sat iff $\text{Decide}(\phi_1)$ is sat or $\text{Decide}(\phi_2)$ is sat.
- What is the trade-off?
  - Converting ϕ into DNF may incur exponential blow-up.

Colorful Logic

- Linear Arithmetic
- Uninterpreted Functions
- Linear Arithmetic + Uninterpreted Functions
- Theory of Arrays
- Theory of Lists
- Non-linear Arithmetic

Linear Arithmetic

Expressions $e := y \mid c \mid e_1 \pm e_2 \mid c \times e$
Atomic facts $g := e \geq 0 \mid e \neq 0$

Note that $e=0$ can be represented as $e \geq 0 \wedge e \leq 0$
$e>0$ can be represented as $e-1 \geq 0$
(over integer LA)

- The decision problem for integer LA is NP-hard.
- The decision problem for rational LA is PTime.
  - PTime algorithms are complicated to implement. Popular choice is a worst-case exponential algorithm called "Simplex"
  - We will study a PTime algorithm for a special case.

Difference Constraints

- A special case of Linear Arithmetic
- Constraints of the form $x \leq c$ and $x-y \leq c$
  - We can represent $x \leq c$ by $x-u \leq c$, where u is a special zero variable. Wlog, we will assume henceforth that we only have constraints $x-y \leq c$
- Reasoning required: $x-y \leq c_1 \wedge y-z \leq c_2 \Rightarrow x-z \leq c_1+c_2$
- $O(n^3)$ (saturation-based) decision procedure
  - Represent contraints by a matrix $M_{n \times n}$
    - where $M[i][j] = c$ represents $x_i - x_j \leq c$
  - Repeatedly apply following rule as in shortest path computation.
    - $M[i][j] = \min_k \{ M[i][j], M[i][k]+M[k][j] \}$
  - ϕ is unsat iff $\exists i$: $M[i][i] < 0$

Colorful Logic

- Linear Arithmetic
- ➢ Uninterpreted Functions
- Linear Arithmetic + Uninterpreted Functions
- Theory of Arrays
- Theory of Lists
- Non-linear Arithmetic

Uninterpreted Functions

Expressions $e := x \mid F(e_1,e_2)$

Atomic fact $g := e_1=e_2 \mid e_1 \neq e_2$

Axiom $\forall e_1,e_2,e_1',e_2' : e_1=e_1' \wedge e_2=e_2' \Rightarrow F(e_1,e_2)=F(e_1',e_2')$

(called congruence axiom)

(saturation-based) Decision Procedure

- Represent equalities $e_1=e_2 \in G$ in Equivalence DAG (EDAG)
  - Nodes of an EDAG represent congruence classes of expressions that are known to be equal.
- Saturate equalities in the EDAG by following rule:
  - If $C(e_1)=C(e_1') \wedge C(e_2)=C(e_2')$, Merge $C(F(e_1,e_2))$, $C(F(e_1',e_2'))$

  where $C(e)$ denotes congruence class of expression e
- Declare unsatisfiability iff $\exists$ $e_1 \neq e_2$ in G s.t. $C(e_1) = C(e_2)$

Uninterpreted Functions: Example

$y = F^5(y) \quad \wedge \quad y = F^3(y) \quad \wedge \quad y \neq F(y)$

$F(y) = F^4(y)$

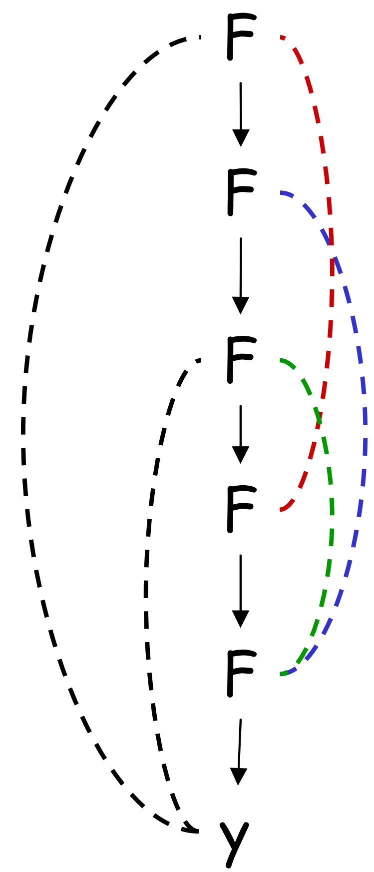
$F^2(y) = F^5(y)$

$y = F^2(y)$

$F(y) = F^3(y)$

$y = F(y)$

$\perp$: unsat



Uninterpreted Functions: Complexity

- Complexity of congruence closure : O(n log n), where n is the size of the input formula
  - In each step, we merge 2 congruence classes. The total number of steps required is thus n, where n is a bound on the original number of congruence classes.
  - The complexity of each step can be O(log n) by using union-find data structure.

Colorful Logic

- Linear Arithmetic
- Uninterpreted Functions
- ➢ Linear Arithmetic + Uninterpreted Functions
- Theory of Arrays
- Theory of Lists
- Non-linear Arithmetic

Combination of Linear Arithmetic and Uninterpreted Functions

Expressions $e := y \mid c \mid e_1 \pm e_2 \mid c \times e \mid F(e_1, e_2)$

Atomic Facts $g := e \geq 0 \mid e \neq 0$

Axioms: Combined axioms of linear arithmetic + uninterpreted fns.

Decision Procedure: Nelson-Oppen methodology for combining decision procedures

Combining Decision Procedures

- Nelson-Oppen gave an algorithm in 1979 to combine decision procedures for theories T_1 and T_2, where:
  - T_1 and T_2 have disjoint signatures
    - except equality
  - T_1, T_2 are stably infinite
- Complexity is $O(2^{n^2} \times (W_1(n)+W_2(n))$.
- If T_1, T_2 are convex, complexity is $O(n^3 \times (W_1(n)+W_2(n)))$.

The theories of linear arithmetic and uninterpreted functions satisfy all of the above conditions.

Convex Theory

A theory is convex if the following holds.

Let $G = g_1 \wedge \ldots \wedge g_n$

If $G \Rightarrow e_1{=}e_2 \vee e_3{=}e_4$, then $G \Rightarrow e_1{=}e_2$ or $G \Rightarrow e_3{=}e_4$

Examples of convex theory:

- Rational Linear Arithmetic
- Uninterpreted Functions

Examples of Non-convex Theory

- Theory of Integer Linear Arithmetic

$2 \leq y \leq 3 \Rightarrow y=2 \vee y=3$

But $2 \leq y \leq 3 \not\Rightarrow y=2$ and $2 \leq y \leq 3 \not\Rightarrow y=3$

- Theory of Arrays

$y=sel(upd(M,a,0),b) \Rightarrow y=0 \vee y=sel(M,b)$

But $y=sel(upd(M,a,0),b) \not\Rightarrow y=0$ and

$y=sel(upd(M,a,0),b) \not\Rightarrow y=sel(M,b)$

Stably Infinite Theory

- A theory T is stably infinite if for all quantifier-free formulas ϕ over T, the following holds:

  If ϕ is satisfiable, then ϕ is satisfiable over an infinite model.

- Examples of stably infinite theories
  - Linear arithmetic, Uninterpreted Functions

- Examples of non-stably infinite theories
  - A theory that enforces finite # of distinct elements. Eg., a theory with the axiom: $\forall x,y,z\ (x=y \vee x=z \vee y=z)$. Consider the quantifier free formula ϕ: $y_1=y_2$.

    ϕ is satisfiable but doesn't have an infinite model.

Nelson-Oppen Methodology

- Purification: Decompose ϕ into $\phi_1 \wedge \phi_2$ such that ϕ_i contains symbols from theory T_i.
  - This can be done by introducing dummy variables.

- Exchange variable equalities between ϕ_1 and ϕ_2 until no more equalities can be deduced.
  - Sharing of disequalities is not required because of stably-infiniteness.
  - Sharing of disjunctions of equalities is not required because of convexity.

- ϕ is unsat iff ϕ_1 is unsat or ϕ_2 is unsat.

Combining Decision Procedures: Example

$y_1 \leq 4y_3 \leq F(2y_2-y_1) \;\wedge\; y_1=F(y_1) \;\wedge\; y_2=F(F(y_1)) \wedge y_1 \neq 4y_3$

Purification

$a_1=2y_2-y_1$

$y_1 \leq 4y_3 \leq a_2 \wedge y_1 \neq 4y_3$

$y_1 = y_2$

$y_1 = a_2$

$a_2=F(a_1)$

$y_1=F(y_1) \wedge y_2=F(F(y_1))$

$y_1 = a_1$

$\xleftarrow{y_1=y_2}$

$\xrightarrow{y_1=a_1}$

$\xleftarrow{y_1=a_2}$

Saturation

$\perp$: unsat

Colorful Logic

- Linear Arithmetic
- Uninterpreted Functions
- Linear Arithmetic + Uninterpreted Functions
- ➢ Theory of Arrays
- Theory of Lists
- Non-linear Arithmetic

Theory of Arrays

Expressions $e := y \mid Select(M,e)$

$M := A \mid Update(M,e_1,e_2)$

Atomic Facts $g := e_1 = e_2 \mid e_1 \neq e_2$

Axioms $Select(Update(F,e_1,e_2), e_3) = e_2$ if $e_1 = e_3$

$= Select(F,e_3)$ o.w.

- The decision problem is NP-complete.
- Use the above rule to rewrite any select applied to an Update. Then use the decision procedure for Uninterpreted Fns.
- <u>Key Idea:</u> Normalization

Colorful Logic

- Linear Arithmetic
- Uninterpreted Functions
- Linear Arithmetic + Uninterpreted Functions
- Theory of Arrays
- Theory of Lists
- Non-linear Arithmetic

Theory of Lists

Expressions $e := y \mid e.f$

Atomic Facts $g := B(e_1,e_2,e_3) \mid \neg g$

$$R(e_1,e_2) =^{def} B(e_1,e_1,e_2)$$

Axioms: Not first order logic axiomatizable!

- The decision problem is NP-complete.
- Decision Procedure: Saturate using the following derivation rules without creating any new terms. The tricky detail is to prove completeness.

Derivation Rules: $R(x,y) \wedge R(x,z) \Rightarrow B(x,y,z) \vee B(x,z,y)$

$R(x,y) \Rightarrow x=y \vee R(x.f,y)$

$R(x,y) \wedge R(x,z) \Rightarrow B(x,y,z) \vee B(x,z,y)$

etc.

Colorful Logic

- Linear Arithmetic
- Uninterpreted Functions
- Linear Arithmetic + Uninterpreted Functions
- Theory of Arrays
- Theory of Lists

➢ Non-linear Arithmetic

Non-linear Operators

Expressions $e := y \mid c \mid e_1 \pm e_2 \mid c \times e \mid nl(e1,e_2)$

Atomic facts $g := e \geq 0 \mid e \neq 0$

Axioms: User-provided first order axioms for nl operator.

- View a non-linear relationship $3\log x + 2^x \leq 5y$ over {x,y} as a linear relationship over {log x, 2^x, y}
- User provides semantics of non-linear operators using directed inference rules of form $L \Rightarrow R$.
  - Exponentiation: $e_1 \leq e_2 + c \Rightarrow 2^{e1} \leq 2^{e2} \times 2^c$
  - Logarithm: $e_1 \leq ce_2 \wedge 0 \leq e_1 \wedge 0 \leq e_2 \Rightarrow \log(e_1) \leq \log c + \log(e_2)$
  - Multiplication: $e_1 \leq e_2 + c \wedge e \geq 0 \Rightarrow ee_1 \leq ee_2 + ec$

Non-linear Operators

Expressions $e := y \mid c \mid e_1 \pm e_2 \mid c \times e \mid nl(e1,e_2)$

Atomic facts $g := e \geq 0 \mid e \neq 0$

Axioms: User-provided first order axioms for nl operator.

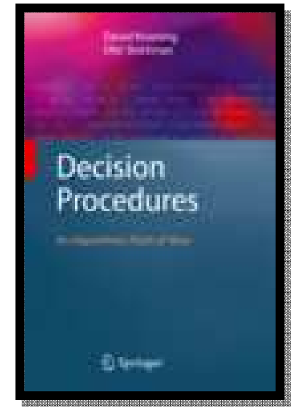
- (semi-) Decision Procedure: Saturate using the axioms provided by the user.
- Termination Heuristic (called Expression Abstraction): Restrict new fact deduction to a small set of expressions, either given by user or constructed heuristically from program syntax.

Logic: Recap

Key Ideas: Normalization, Saturation w/o creating new terms or over heuristically constructed terms.

- Linear Arithmetic: Non-saturation decision procedures.
- Uninterpreted Functions: Saturation using the axiom over efficient EDAG data-structure.
- Linear Arithmetic + Uninterpreted Functions: Modular construction, Sharing of variable equalities.
- Theory of Arrays: Normalization using the axiom
- Theory of Lists: Saturation using a special set of derivation rules.
- Non-linear Arithmetic: Saturation using user-provided axiomatization over user-provided set of expressions.

Logic: References



- Decision Procedures: An Algorithmic Point of View; Daniel Kroening, Ofer Strichman
  - Linear Arithmetic, Uninterpreted Fns, Combination, Arrays, Bit-vectors
- Back to the Future: Revisiting Precise Program Verification using SMT Solvers; Lahiri, Qadeer; POPL '08
  - Reachability
- A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis; Gulavani, Gulwani; CAV '08
  - Non-linear operators

How to write a good PL paper on logic?

- Identify a class of programs that make use of domain-specific constructs.
  - E.g., Programs manipulating bit-vectors, strings.
- Develop a language of facts with following properties:
  - Can describe useful properties of those programs.
  - Closed under weakest precondition.
  - Amenable to efficient reasoning.
- Develop a decision procedure for the logic.
  - Proving completeness is usually the hard part.