Program Transformations for Automated Verification

Abhik Roychoudhury
(National University of Singapore)
I. V. Ramakrishnan
(State University of New York at Stony Brook)

An Example

\[\text{even}(0). \quad \text{even}(\text{s}(\text{s}(X))) :- \text{even}(X).\]
\[\text{odd}(\text{s}(0)). \quad \text{odd}(\text{s}(\text{s}(X))) :- \text{odd}(X).\]
\[\text{strange}(X) :- \text{even}(X), \text{odd}(X).\]
\[\text{strange}(\text{s}(\text{s}(X))) :- \text{strange}(X).\]
\[\text{even}(0). \quad \text{even}(\text{s}(\text{s}(X))) :- \text{even}(X).\]
\[\text{odd}(\text{s}(0)). \quad \text{odd}(\text{s}(\text{s}(X))) :- \text{odd}(X).\]

Proof by transformations

- We can run a routine syntactic check on the transformed definition of \text{strange}/1 to prove \(\forall X \neg \text{strange}(X)\)

- Tabled evaluation of the query \text{strange}(X) is one such check.

Transformation: Unfolding

\[B :- G, A, G'\]
\[A_1 :- B_d_1\]
\[\ldots\]
\[A_n :- B_d_n\]

\(\sigma_i = \text{mgu}(A_i, A_j)\)
Transformation: Folding

\[
\begin{align*}
A_1 :&= B_1 \\
... \\
A_n :&= B_n
\end{align*}
\]

Clauses defining A

\[
\begin{align*}
A_1 \theta_1 = ... = A_n \theta_n = A
\end{align*}
\]

\[
\begin{align*}
B :&= G, B_1 \theta_1, G \\
... \\
B :&= G, B_n \theta_n, G
\end{align*}
\]

Fold

\[
\begin{align*}
B :&= G, A, G
\end{align*}
\]

In slow motion ...

\[
\begin{align*}
\text{Clauses defining } A
\end{align*}
\]

\[
\begin{align*}
\text{Unf}(\text{resolution step})
\end{align*}
\]

\[
\begin{align*}
\text{Odd}(\text{resolution step})
\end{align*}
\]

\[
\begin{align*}
\text{Uniform}(\text{resolution step})
\end{align*}
\]

Salient points

- We consider only definite logic programs.
- Semantics considered is the Least Herbrand Model.
- Unfolding corresponds to one step of resolution.
- Folding corresponds to
  - Remembering definitions from previous programs.
  - Recognizing instances of such definitions
Organization
- Traditional issues in transformations
- Transformations for deduction – Basics
- Features of Transformation proofs
- Verifying infinite state reactive systems
- Experimental work
- Ongoing research directions

Transform for efficiency
- Application of unfold/fold transformations leads to:
  - Deforestation (removal of intermediate data-structures)
  - Tupling (Avoiding multiple traversals of same data structure)
- Unfold/fold rules form heart of program specialization techniques: Conjunctive partial deduction.
- Extensive literature on this topic, outside our scope.

Folding is reversible?
- Only if all clauses participating in folding appear in the same program.
- Least Herbrand Model preserved by any interleaving of unfolding and (reversible) folding.

\[ \text{def}(X, Y, Z, R) : - \text{append}(X, Y, I), \text{append}(I, Z, R). \]
\[ \text{def}([], Y, Z, R) : - \text{append}(Y, Z, R). \]
\[ \text{def}([H|X1], Y, Z, [H|R1]) : - \text{def}(X1, Y, Z, R1). \]
Need irreversible folding

d(X,Y,Z,R) :- append(X,Y,I), append(I,Z,R).

d([],Y,Z,R) :- append(Y,Z,R).

d([H|X1],Y,Z,[H|R1]) :- append(X1,Y,I1), append(I1,Z,R1).

d([],Y,Z,R) :- append(Y,Z,R).

d([H|X1],Y,Z,[H|R1]) :- da(X1,Y,Z,R1).

Unf*

Fold

d([],Y,Z,R) :- append(Y,Z,R).

d([H|X1],Y,Z,[H|R1]) :- da(X1,Y,Z,R1).

Correctness Issues

p :- r
q :- r.
r.
p :- q.
q :- p.
r.

Irreversible folding does not preserve semantics.
Circularities introduced; reduces Least Herbrand Model.

Correctness issues

q :- Bd1
p :- ... *

1. Show $\alpha(q) < \alpha(p)$ for a measure $\alpha$ and $wfo$ <
2. Can be achieved if $\alpha(q) < \alpha(Bd1) < \alpha(p)$
3. $\alpha(q) < \alpha(Bd1)$ is typically ensured by
   - Restricting syntax of $q$ and book-keeping during bfs.
4. Restrictions on syntax are in fact unnecessary.

Organization

- Some traditional issues in transformations
- Transformations for deduction – Basics
- Distinctive Features of Transformation proofs
- Verifying infinite state reactive systems
- Experimental work
- Ongoing research directions
Recap of Example

\begin{align*}
even(0). & \quad \text{even}(\text{s}(\text{s}(\text{x}))) \iff \text{even}(\text{x}). \\
\text{odd}(0). & \quad \text{odd}(\text{s}(\text{s}(\text{x}))) \iff \text{odd}(\text{x}). \\
\text{strange}(\text{x}) & \iff \text{even}(\text{x}), \text{odd}(\text{x}).
\end{align*}

\textbf{Transform}

\begin{align*}
\text{strange}(\text{s}(\text{s}(\text{x}))) & \iff \text{strange}(\text{x}).
\end{align*}

\textbf{Proves} \quad \forall \text{x} \quad \neg \text{strange}(\text{x})

Uncovering schema

\begin{align*}
\text{strange}(\text{x}) & \iff \text{even}(\text{x}), \text{odd}(\text{x}). \\
\text{strange}(0) & \iff \text{odd}(0). \\
\text{strange}(\text{s}(\text{s}(\text{y}))) & \iff \text{even}(\text{y}), \text{odd}(\text{s}(\text{s}(\text{y}))).
\end{align*}

Unfold

\begin{align*}
\text{odd}(0). & \\
\text{odd}(\text{s}(\text{s}(\text{y}))) & \iff \text{odd}(\text{y}).
\end{align*}

Prove by inducting on the scheme of even/2

Base case of proof

\begin{align*}
\text{strange}(0) & \iff \text{odd}(0). \\
\text{strange}(\text{s}(\text{s}(\text{x}))) & \iff \text{even}(\text{x}), \text{odd}(\text{s}(\text{s}(\text{x}))).
\end{align*}

Unfold

\begin{align*}
\text{odd}(0). & \\
\text{odd}(\text{s}(\text{s}(\text{y}))) & \iff \text{odd}(\text{y}).
\end{align*}

\textbf{Prove} \quad \neg \text{strange}(0)

Initiating induction step

\begin{align*}
\text{strange}(0) & \iff \text{fail}. \\
\text{strange}(\text{s}(\text{s}(\text{y}))) & \iff \text{even}(\text{y}), \text{odd}(\text{s}(\text{s}(\text{y}))).
\end{align*}

Unfold

\begin{align*}
\text{odd}(0). & \\
\text{odd}(\text{s}(\text{s}(\text{y}))) & \iff \text{odd}(\text{y}).
\end{align*}

\textbf{The inductive step:} \quad X = \text{s}(\text{y})
Recognize Induction Hyp.

\[ \text{strange}(s(s(Y))) : - \text{even}(Y), \text{odd}(Y) \]

\[ \text{strange}(s(s(Y))) : - \text{strange}(Y) \]

Recall \( \text{strange}(X) : - \text{even}(X), \text{odd}(X) \) in \( P \)

Finally, run a syntactic check on \( \text{strange}/1 \)

Unfold/fold for deduction

- Transform \( p \) and \( q \) s.t. \( p = q \) can be inferred by a computational induction of their definitions.
- Unfolding: Base case and finite part of induction step
- Folding: Recognize induction hypothesis.
- Infer \( p = q \) based on syntax, if after transformation:
  - \( p(0) \)
  - \( q(0) \)
  - \( p(s(X)) : - p(X) \)
  - \( q(s(X)) : - q(X) \)
- Can define a testable notion of syntactic equivalence based on this idea.

Organization

- Some traditional issues in transformations
- Transformations for deduction – Basics
- Features of Transformation proofs
- Verifying infinite state reactive systems
- Experimental work
- Ongoing research directions

A generate-test program

\[ \text{gen}([1|\_]) \]
\[ \text{gen}([H|X]) : - \text{gen}(X) \]
\[ \text{test}([\_]) \]
\[ \text{test}([0|X]) : - \text{test}(X) \]
\[ \text{gentest}(X) : - \text{gen}(X), \text{test}(X) \]

\text{gen}: Generates strings with \( 1 \)
\text{test}: Tests for \( 0^* \)
\text{Prove} \forall X \leftrightarrow \text{gentest}(X)
A look at induction schemes

ACL2 (first-order theorem prover) produces the schema obtained from gen/1

\[ X = [] \]

\[ X = [H|T] \land H = 1 \]

\[ X = [H|T] \land H \neq 1 \land \neg \text{gen}(T) \]

\[ X = [H|T] \land H = 1 \land \neg \text{test}(T) \]

Proof by transformations

\[ \text{gentest}([1|X]) : \neg \text{test}([1|X]). \]

\[ \text{gentest}([H|X]) : \neg \text{gen}(X), \text{test}([H|X]). \]

\[ \text{gentest}([1|X]) : \neg \text{false}. \]

\[ \text{gentest}([H|X]) : \neg \text{gen}(X), \text{test}([H|X]). \]

Unfold/fold induction schemes

- In any unfold/fold based proof of \( p = q \)
  - Schema to induct is created gradually by unfolding.
  - Inbuilt Unification, spurious cases ignored (\( X = [] \))
  - Since the schema is gradually constructed, unnecessary case splits may be avoided

- Scheme is still obtained from program predicates. Idea traces back to recursion analysis used in Boyer-Moore prover to generate schemes.
Not so Inductionless

- Induction scheme in an unfold/fold proof is not given a-priori. It is constructed:
  - From recursive definition of program predicates
  - Gradually via unfolding
- Contrast with inductive techniques which do not require any scheme (Inductionless Induction)
  - Given axioms $P$ (our pgm) and first-order axiomatization $A$ of the minimal Herbrand model of $P$, property $\Psi$ holds iff $\Psi \cup A \cup P$ is consistent.

Other techniques - Tabling

Folding is achieved by remembering formulae.

\[
p : q
\]
\[
p : r, s
\]
\[
w : Bd, q
\]
\[
w : Bd, r, s
\]
\[
w : Bd, p
\]

1. Tabled evaluation of logic programs combines unfolding with tabulation of atoms.
2. Detection of tabled atom succeeded not by folding, but feeding existing answers. (Answer Clause Resolution)

Other techniques - Rippling

- Takes in induction schema and rewrites induction conclusion.
- Rewrite induction conclusion to obtain complete copies of the induction hypothesis. Restricts rewrite rules for this purpose.
- Unfold/fold proofs do not input explicit schema.
- The strategies for guiding unfolding are not necessarily similar to rippling. The purpose is however similar: create opportunities for folding.

Organization

- Some traditional issues in transformations
- Transformations for deduction – Basics
- Distinctive Features of Transformation proofs
- Verifying infinite state reactive systems
- Experimental work
- Ongoing research directions
Overall perspective

- Transition relation of finite/infinite state system captured as (constraint) logic program.
- Temporal properties captured as logic program preds.
- Checking of temporal properties of finite state systems then proceeds by query evaluation.
- Due to loops in the state transition graph of the systems being verified
  - Memoized query evaluation is needed.
  - Ensures termination.

Classes of infinite state systems can be verified by query evaluation of constraint logic programs.
- Real-time systems
- Systems with integer data and finitely many control locations

What about infinite families of finite state systems (parameterized systems)?

---

Model Checking ...

```
efp(S) :- p(S).
epf(S) :- trans(S,T), epf(T).
trans(s0,s1).
trans(s0,s2).
trans(s2,s1).
p(s1).
```

---

... by Query Evaluation

Check `s0 ⊨ EF p` by evaluating query `epf(s0)`

```
epf(s0) :- p(s0).
epf(s0) :- trans(s0,X), epf(X).
trans(s0,s1).
trans(s0,s2).
trans(s2,s1).
p(s1).
```

Explicit state Model Checking
Parameterized systems

\[ \begin{array}{c}
0 \\
1 \quad 0 \\
n \quad \ldots \\
n-1 \quad 0
\end{array} \]

Verify \( \text{EFp} \) in start state of each member of the family

Infinite family of finite state systems e.g. occur widely in distributed algorithms

Query Eval is not enough

Verify \( \text{EFp} \) in start state of each member of the family

\[
\text{efp}(X) :- p(X).
\]

\[
\text{efp}(X) :- \text{trans}(X, Y), \text{efp}(Y).
\]

Deduce \( \forall X \, \text{nat}(X) \Rightarrow \text{efp}(X) \)

Model Checking by Query Evaluation

- Proves property for individual members
  \( \vdash \text{efp}(1) \)
- Enumerates members for which property holds
  \( \vdash \text{efp}(X) \)

Verify via Transformations

- Prove \( \forall X \, \text{nat}(X) \Rightarrow \text{efp}(X) \) by induction on the structure of \( \text{nat}/1 \). (computational induction)
  - \( \text{nat}/1 \) defined as
    - \( \text{nat}(0) \)
    - \( \text{nat}(s(X)) :- \text{nat}(X) \)
  - \( \text{efp}/1 \) should include the clauses
    - \( \text{efp}(0) \)
    - \( \text{efp}(s(X)) :- \text{efp}(X) \)

- Achieved by transforming \( \text{efp}/1 \).

Recap on induction

- Unfolding
  - Base case (Model Checking)
  - Initiating induction step
- Folding
  - Recognizing instance of induction hypothesis
- Use structural induction on transformed def.
Verify parameterized systems

- Encode temporal property/parameterized system as a logic program $P_0$
- Convert verification proof obligation to a predicate equivalence proof obligation $p = q$ in $P_0$
- Construct a transformation sequence $P_0, ..., P_k$
  - Semantics of $P_0 =$ Semantics of $P_k$
  - Can infer $p = q$ in $P_k$ via a syntactic check.

Is it any different?

- Prove temporal property $EFp$ about inf. family $st$
  - Encode property $EFp$ as predicate $efp$
  - Induction step is $efp(st(N)) \Rightarrow efp(st(N+1))$
- Temporal property $EFp$ (denoting reachability) is encoded by $efp/1$ predicate
  - $efp(X) \Rightarrow p(X)$
  - $efp(X) \Rightarrow \text{trans}(X,Y)$, $efp(Y)$
- Recognizing induction hypothesis $efp(st(N))$ may involve folding using the predicate $efp$
- Such a folding is semantics preserving? NOT known

Correctness of folding

- Irreversible folding reduces Least Herbrand model
  - $p :- q \rightarrow p :- p$
  - $q :- q$
- Preservation of Least Herbrand Model proved by induction on some measure of ground atoms.
- To let this induction go through
  - Unexpected restrictions imposed on syntax of clauses participating in folding.
  - Book-keeping maintained at each unfold/fold step.
  - Restrictions on book-keeping of transformed clauses.

Restrictions on syntax

$p :- Body$

$q :- G, Body, G' \rightarrow q :- G, p, G'$

1. Can introduce circularity if $p = q$
2. Prevented by demanding $p < q$ in a measure $\alpha$
3. Show $\alpha(p) < \alpha(Body) < \alpha(q)$
4. Places syntactic restrictions on $Body$
Remove such restrictions

- Attach measures to both clauses and atoms
- Atom measure denotes proof size.
- Relate atom and clause measures. For a clause A :- Body with integer annotations (γ, γ′)
  - γ, γ′ bound α(A) – α(Body)
  - Clause measures are updated in every step.
  - Conditions on α now ensured by imposing restrictions on clause measures γ
- No restriction on syntax. More general rules.

Organization

- Some traditional issues in transformations
- Transformations for deduction – Basics
- Distinctive Features of Transformation proofs
- Verifying infinite state reactive systems
- Automation and Experimental work
- Summary and Ongoing Research

Verifying Invariants

- Verify an invariant ¬bad in a parameterized system (Initial states: init/1, Transitions: trans/2)
  - None of the initial states satisfy bad/1
  - Also, ¬bad

Both proved by induction on term (process) structure:

∀X (init(X) ∧ bad(X) ⇒ false)

∀X,Y (trans(X,Y) ∧ bad(Y) ⇒ trans(X,Y) ∧ bad(X))

Mutex in token ring

- init([0,1]). init([0|X]) :- init(X).  // initial states
- trans(X,Y) :- t1(X,Y).  // pass token along chain
- trans([1|X],[0|Y]) :- t2(X,Y).  // pass token along the end
- t1([0],[1]). t1([H|T], [H|T1]) :- t1(T,T1).
- t2([0],[1]). t2([H|X], [H|Y]) :- t2(X,Y).
- bad([1|X]) :- one_more(X). bad([0|X]) :- bad(X).
- one_more([1|_]). one_more([0|X]) :- one_more(X).
- bad_dest(X,Y) :- trans(X,Y), bad(Y).
- bad_src(X,Y) :- trans(X,Y), bad(Y).
Verifying Invariants

- ∀X (init(X) ∧ bad(X) ⇒ false)
- Define bad_init(X) := bad(X), init(X).
- Transform bad_init/1 to show bad_init ⇒ false

- ∀X (trans(X,Y) ∧ bad(Y) ⇒ trans(X,Y) ∧ bad(X))
- Define bad_dest(X, Y) := trans(X, Y), bad(Y).
- Define bad_src(X, Y) := trans(X, Y), bad(X).
- Transform bad_dest/2 and bad_src/2 to show bad_dest ⇒ bad_src

The proof structure

p(X,Y) :- trans(X,Y), bad(Y).
q(X,Y) :- trans(X,Y), bad(X).

Transform

p(a[X], [b|Y]) :- p(X, Y).
p(b[X], [a|Y]) :- p(X, Y).
p(a[X], [a|X]) :- bad(X).
q(a[X], [b|Y]) :- q(X, Y).
q(b[X], [a|Y]) :- q(X, Y).
q(a[X], [a|X]) :- bad(X).

p1 ⇒ q1 proved by transformations

Nested induction

- Nested induction proofs simulated by nesting predicate equivalence proof obligations.
- A proof of p = q needs lemma p1 = q1. This lemma is again proved by unfold/fold.
- Proof of top-level obligation terminates when all such encountered lemma are proved.
- Geared to automate
  - Nested induction proofs, where
  - Each induction needs no hypothesis strengthening.

Proof search skeleton

p = q

Unfold*

Fold*

Transformed p, q

Nested Induction

p1 ⇒ q1

p2 ⇒ q2

p3 ⇒ q3
Unfold-fold-unfold ....

- Nesting of proofs leads to
  - Interleaving of unfolding and folding steps
  - Unfolding - Algorithmic verification steps
  - Folding - Deductive Verification steps
- Compare to theorem prover invoking model checker as decision procedure: Tighter Integration.
- Opposite approach. Extends the evaluation technique of a model checker with limited deduction.

State-of-the art

- Model Checker invoked by Theorem Prover.
  - Current model checkers check $M \models \Phi$ and return
    - Yes if $M \models \Phi$
    - Counterexample trace/tree otherwise
  - Instead, we let our logic programming based model checker return a full-fledged justification [PPDP 00]
  - Recent efforts in verification community to let a model checker return proof/disproof. [CAV, FST&TCS 01] Proof/disproof manipulated by Theorem Prover. But !

More on integration

- Feeding model checking proof/disproof into provers
  - Advances state-of-the art in integration
  - Coarser integration than ours: TP/MC steps not interleaved
- LP forms uniform representation which both model checking and deductive proof steps can manipulate.
  - Deductive steps (folding etc.) can be applied lazily
    - Reduces to model checking for finite state systems
    - No additional overhead for theorem proving

Guiding unfolding

- To prove $p \Rightarrow q$, unfold before fold. Many issues in guiding the unfolding search itself.
- Criteria for guiding unfolding
  - Termination (Memoization based techniques)
  - Convergence (Excessive unfolding should not disable deductive steps)
  - Selection order (Left-to-right)
- Recall, deductive steps applied only on-demand.
  - Need to avoid unfolding steps which disable folding.
Need more than termination

\[ \text{thm}(X) :- \text{state}(X), \text{prop}(X). \]

\[ \text{thm}(f(X)) :- \text{state}(X), \text{prop}(f(X)). \]

\[ \text{thm}(f(X)) :- \text{state}(X), \text{prop}(X). \]

**Folding enabled**

1. Termination guided unfolding does not detect this.
2. Costly to check for applicable folding in each step.

Under the carpet

- Heuristics to identify unfolding steps which are likely to disable opportunities for folding.

Verify \( \neg \text{bad of (init/1, trans/2)} \) by transformations:

- Exploit \text{generate-test} nature of predicates to be transformed
- \( \forall X,Y \ ( \text{trans}(X,Y) \land \text{bad}(Y) \Rightarrow \text{trans}(X,Y) \land \text{bad}(X) ) \)
- \( \text{bad_dest}(X, Y) :- \text{trans}(X, Y), \text{bad}(Y). \)
- \( \text{bad_src}(X, Y) :- \text{trans}(X, Y), \text{bad}(X). \)

**TEST:** Unfolding cannot generate instantiations

Implementation of prover

- A program transformation based prover implemented on top of XSB tabled LP system.
- Good for inductive proofs without strengthening.
- Used for inductive verification of invariants for parameterized networks.
- In these proofs, domain knowledge exploited as:
  - \text{generate-test} nature of the problem helps identify redundant unfolding steps
  - Topology of the parameterized network (Linear, Tree etc) used to choose nested proof obligations after transformation

Case Studies

- Parameterized versions of multiprocessor cache coherence protocols
  - Berkeley-RISC, MESI, Illinois (bus based protocols)
  - Also studied before by other verification techniques
  - A multi-bus hierarchical tree network of cache agents
- Client-server protocol by Steve German
  - Well-studied benchmark in last 2 years: 2001-2002
- Java Meta-Locking Algorithm (Sun Microsystems)
  - Ensures mutual exclusion of object access by Java threads
### Some numbers

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Invariant</th>
<th>Time</th>
<th>#unf</th>
<th>#ded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mesi</td>
<td>Reader-write excl.</td>
<td>2.9</td>
<td>308</td>
<td>63</td>
</tr>
<tr>
<td>Berkeley</td>
<td>&lt;2 dirty cache lines</td>
<td>6.8</td>
<td>503</td>
<td>146</td>
</tr>
<tr>
<td>Illinois</td>
<td>&lt;2 dirty cache lines</td>
<td>35.7</td>
<td>2501</td>
<td>137</td>
</tr>
<tr>
<td>Client-server</td>
<td>&lt;2 exclusive clients</td>
<td>8.8</td>
<td>404</td>
<td>204</td>
</tr>
<tr>
<td>Java</td>
<td>Mutual exclusion in object access</td>
<td>129.8</td>
<td>1981</td>
<td>311</td>
</tr>
</tbody>
</table>

### Organization

- Some traditional issues in transformations
- Transformations for deduction – Basics
- Distinctive Features of Transformation proofs
- Verifying infinite state reactive systems
- Automation and Experimental work
- Summary and Ongoing Research

### Summary

- Predicate equivalences can be proved by induction on recursive structure (**computational induction**).
- Unfold/fold transformations make the applicability of such inductive arguments explicit.
- Verification problem for infinite state reactive systems can be turned into showing predicate equivalences... ...hence solved by unfold/fold transformations.
- **Different** from the traditional usage of transformations in improving program efficiency.

### Related work

- Hsiang-Srivas [1985]
  - The need to suspend Prolog evaluation
  - Extend Prolog evaluation with "forward chaining using certain clauses" (limited folding)
- Kanamori-Fujita [1986]
  - Use of computational induction schemes to prove universal formulae about Least Herbrand Model.
  - Uses transformations s.t. suggested computational induction schemes are easily proved.
Related work

- Pettorossi-Proietti [1993]
  - Unfold/fold transformations to prove equivalence of atoms/predicates
  - Applications in program synthesis
- Our work [1999]
  - Use transformations for inductively proving properties of infinite state reactive systems.
  - Cleaner, more general folding rule
  - Tightly integrates model checking and induction

Ongoing Work

- Extend the transformation rules and strategies to constraint logic programs
  - Enables verification of larger class of reactive systems - parameterized real-time systems
- Combine limited hypothesis strengthening with the inductive proofs of invariants based on unfold/fold
  - Generate strengthened hypothesis from the failure of a unfold/fold based inductive proof
  - Corresponds to top-down invariant strengthening

???