Program Transformations for Automated Verification

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

An Example

```
even(0).        even(s(s(X))) :- even(X).
odd(s(0)).      odd(s(s(X))) :- odd(X).
strange(X) :- even(X), odd(X).
```

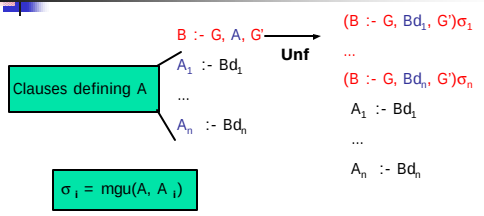
Transform

```
strange(s(s(X))) :-  strange(X).
even(0).        even(s(s(X))) :- even(X).
odd(s(0)).      odd(s(s(X))) :- odd(X).
```

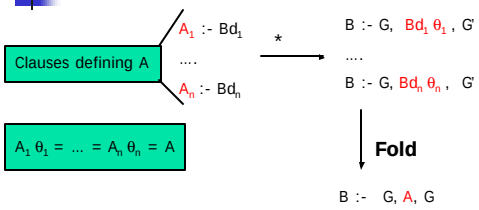

Proof by transformations

- We can run a routine syntactic check on the transformed definition of strange/1 to prove $\forall X \neg$ strange(X)
- Tabled evaluation of the query strange(X) is one such check.

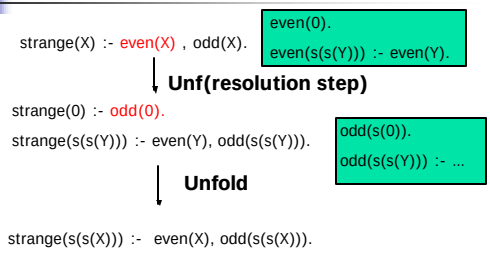
Transformation: Unfolding



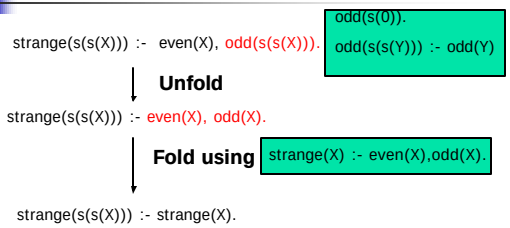
Transformation: Folding



In slow motion ...



In slow motion...



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

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

(Unfold | fold) *

```
da([], Y, Z, R) :- append(Y, Z, R).
da([H|X1], Y, Z, [H|R1]) :- da(X1, Y, Z, R1).
```

Avoids construction/traversing of list I.

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 ?

```
p :- r, s                p :- r, s              p :- r, s
q :- u,v, r,s    ⟶      q :- u,v, p    ⟶      q :- u,v,r,s
```

Only if all clauses participating in folding appear in the same program.

⇓

Least Herbrand Model preserved by any interleaving of unfolding and (reversible) folding.

Need irreversible folding

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

Unf*

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

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

Fold

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

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

Correctness Issues

p:- r
q :- r.
r.

fold →

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

fold →

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

Irreversible folding does not preserve semantics.

Circularities introduced; reduces Least Herbrand Model.

Correctness issues

q:- Bd1 → p :- Bd1, Bd2 → p :- q, Bd2
p :- ... * **fold**

1. Show $\alpha(q) < \alpha(p)$ for a measure α 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 txf.
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

even(0). even(s(s(X))) :- even(X).

odd(s(0)). odd(s(s(X))) :- odd(X).

strange(X) :- even(X), odd(X).

Transform

strange(s(s(X))) :- strange(X).

Proves $\forall X \neg$ strange(X)

Uncovering schema

strange(X) :- even(X) , odd(X).

even(0).
even(s(s(Y))) :- even(Y).

Unfold

strange(0) :- odd(0).

strange(s(s(Y))) :- even(Y), odd(s(s(Y))).

Prove by inducting on the scheme of even/2

Base case of proof

strange(0) :- odd(0).

strange(s(s(X))) :- even(X), odd(s(s(X))).

odd(s(0)).
odd(s(s(Y))) :- ...

Unfold

strange(0) :- fail.

strange(s(s(X))) :- even(X), odd(s(s(X))).

Prove $\neg$ strange(0)

Initiating induction step

strange(0) :- fail.

strange(s(s(Y))) :- even(Y), odd(s(s(Y))).

odd(s(0)). odd(s(s(Y))):- odd(Y).

Unfold

strange(s(s(Y))) :- even(Y), odd(Y).

The inductive step: X = s(Y))

Recognize Induction Hyp.

strange(s(s(Y))) :- even(Y), odd(Y).

↓ **Fold using** strange(X) :- even(X),odd(X).

strange(s(s(Y))) :- strange(Y).

Recall strange(X) :- even(X), odd(X) in P_{init}

Finally, run a syntactic check on 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

```
gen([1|_]).        gen([H|X]) :- gen(X).
test([ ]).         test([0|X]) :- test(X).
gentest(X) :- gen(X), test(X).
```

gen: Generates strings with 1

test : Tests for 0*

Prove ∀X ¬ gentest(X)

A look at induction schemes

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

- X = []
- X = [H|T] ∧ H =1
- X = [H|T] ∧ H ≠ 1 ∧ ¬ gen(T)
- X = [H|T] ∧ H ≠ 1 ∧ ¬ test(T)

Redundant

```
gen([1|_]).
gen([H|T]) :- gen(T).
```

Merge ?

Proof by transformations

```
gentest(X) :- gen(X), test(X).
```

```
gentest([1|X]) :- test([1|X]).
gentest([H|X]) :- gen(X), test([H|X]).
```

```
gentest([1|X]) :- false.
gentest([H|X]) :- gen(X), test([H|X]).
```


Proof by Transformations

```
gentest([1|X]) :- false.
gentest([H|X]) :- gen(X), test([H|X]).
```

```
gentest([1|X]) :- false.
gentest([0|X]) :- gen(X), test(X).
```

...

Unfold/fold induction schemes

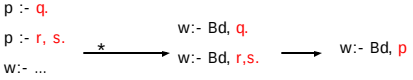
- In any unfold/fold based proof of $p \equiv q$
  - Schema to induct is created gradually by unfolding.
  - Inbuilt Unification, spurious cases ignored (X = [])
  - Since the schema is gradually constructed, unnecessary case splits may be avoided
- Scheme is still obtained from program predicates. Idea 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 Ψ holds iff $\Psi \cup A \cup P$ is consistent.

Other techniques - Tabling

Folding is achieved by remembering formulae.



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

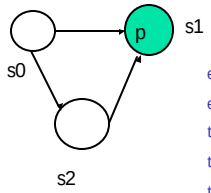
Overall perspective

- 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

Overall perspective

- 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 ...

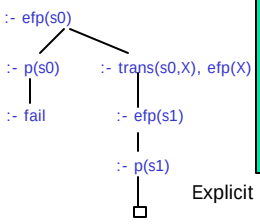


s0 |= EF p

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


... by Query Evaluation

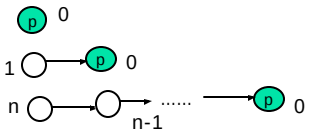
Check s0 |= EF p by evaluating query efp(s0)




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

Explicit state Model Checking

Parameterized systems



Verify 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 EFp in start state of each member of the family

```
efp(X) :- p(X).
efp(X) :- trans(X, Y), efp(Y).
```

Deduce ∀X nat(X)⇒efp(X)

Model Checking by Query Evaluation

- Proves property for individual members
  :- efp(1).
- Enumerates members for which property holds
  :- efp(X).

Verify via Transformations

- Prove ∀ X nat(X) ⇒ efp(X) by induction on the structure of nat/1. (computational induction)
- nat/1 defined as
  - nat(0). nat(s(X)) :- nat(X).
- efp/1 should include the clauses
  - efp(0). efp(s(X)) :- efp(X).
- Achieved by transforming 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 $p \equiv q$ in P_0
- Construct a transformation sequence $P_0,\ldots,P_k$
  - Semantics of P_0 = Semantics of P_k
  - Can infer $p \equiv q$ in P_k via a syntactic check.

Is it any different ?

- Prove temporal property EF p about inf. family st
  - Encode property EFp as predicate efp
  - Induction step is efp(st(N)) ⇒ efp(st(N+1))
- Temporal property EFp (denoting reachability) is encoded by efp/1 predicate
  - efp(X) :- p(X). efp(X) :- 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 → 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' → q :- G, p, G'

1. Can introduce circularity if $p = q$
2. Prevented by demanding $p < q$ in a measure α
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 $\neg$ bad in a parameterized system (Initial states: init/1, Transitions: trans/2)
  - None of the initial states satisfy bad/1
  - Also, ¬bad ⊗ bad

Both proved by induction on term (process) structure

$\forall X$ (init(X) $\wedge$ bad(X) $\vdash$ false)

$\forall X,Y$ (trans(X,Y) $\wedge$ bad(Y) $\vdash$ trans(X,Y) $\wedge$ bad(X))

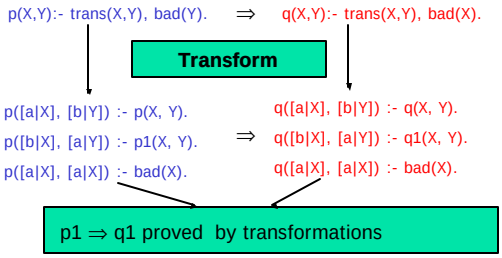
Mutex in token ring

```
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|T], [1,0|T]).          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(X).
```

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,Y (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

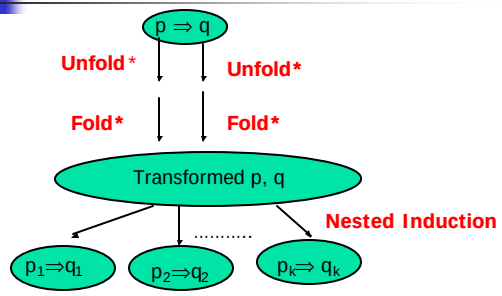
The proof structure



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



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 |= Φ and return
  - Yes if M |= Φ
  - 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 !

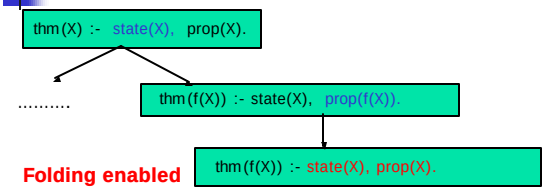
More on integration

- 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 ⇒ 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.

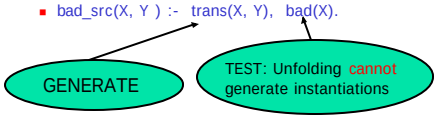
Need more than termination



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 ¬ bad of (init/1, trans/2) by transformations:
  - Exploit generate-test nature of predicates to be transformed
  - ∀X,Y (trans(X,Y) ∧ bad(Y) ⇒ trans(X,Y) ∧ bad(X))
  - bad_dest(X, Y) :- trans(X, Y), bad(Y).
  - bad_src(X, Y) :- trans(X, Y), bad(X).



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:
  - 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

Protocol	Invariant	Time	#unf	#ded
Mesi	Reader-writer excl.	2.9	308	63
Berkeley	<2 dirty cache lines	6.8	503	146
Illinois	<2 dirty cache lines	35.7	2501	137
Client-server	<2 exclusive clients	8.8	404	204
Java Metalock	Mutual exclusion in object access	129.8	1981	311

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

???