Unfold/Fold Transformations of Logic Programs

Abhik Roychoudhury

Department of Computer Science

State University of New York at Stony Brook

Joint work with : K. Narayan Kumar

C.R. Ramakrishnan

I.V. Ramakrishnan

Logic Programming

- Directly executable specifications. Natural to specify problems with:
  - *Backtracking Search*
  - *Fixed point computations*
- Applications : *NLP, Program Analysis, Model Checking*
- A (definite) logic program is a finite set of Horn clauses.

```
app([], L, L).
app([H|X], Y, [H|Z]) :- app(X, Y, Z).
```

- Computation proceeds by query evaluation via resolution.

```
app([1,3], [2], X)    app(X, [1], [0,1])
X = [1,3,2]           X = [0]
```

Semantics of Logic Programs

$M(P)$ = Least Herbrand Model of P
= Gnd. atoms which are logical consequences of P
= Gnd. atoms which are proved by resolution with clauses of P

$P =$
```
p(a).
p(f(X)) :- p(X).
```

$P' =$
```
p(f(X)) :- p(X).
```

$M(P) = \{\texttt{p(a)}, \texttt{p(f(a))}, \texttt{p(f(f(a)))}, \ldots\}$

$M(P') = \{\}$

Proofs in P:
```
p(a)    p(f(a))    p(f(f(a))) ...
 |         |           |
 □       p(a)       p(f(a))
           |           |
           □         p(a)
                       |
                       □
```

Proofs in P': None

Theorem $A \in M(P) \Leftrightarrow A$ appears in the root of a proof.

Program Transformation : Unfolding

$$\begin{array}{l} \mathtt{q} :- \mathtt{g}, \boxed{\mathtt{p}}, \mathtt{g}'. \\ \\ \mathtt{p} :- Bd_1. \\ \mathtt{p} :- Bd_2. \\ \vdots \\ \mathtt{p} :- Bd_n. \end{array} \quad \xrightarrow{Unfold} \quad \begin{array}{l} \mathtt{q} :- \mathtt{g}, \; Bd_1, \; \mathtt{g}'. \\ \mathtt{q} :- \mathtt{g}, \; Bd_2, \; \mathtt{g}'. \\ \vdots \\ \mathtt{q} :- \mathtt{g}, \; Bd_n, \; \mathtt{g}'. \\ \mathtt{p} :- Bd_1. \\ \vdots \\ \mathtt{p} :- Bd_n. \end{array}$$

- An application of unfolding is a resolution step.

Program Transformation : Folding

$$\frac{P_j\ (j \le i):\quad \begin{array}{l} \mathtt{p} :- Bd_1. \\ \vdots \\ \mathtt{p} :- Bd_n. \end{array}}{P_i:\quad \begin{array}{l} \mathtt{q} :- \mathtt{g},\ \boxed{\begin{array}{l} Bd_1, \\ Bd_2, \\ \\ Bd_n, \end{array}}\ \mathtt{g}'. \end{array} \quad \xrightarrow{\textit{Fold}} \quad \mathtt{q} :- \mathtt{g}, \boxed{\mathtt{p}}, \mathtt{g}'.}$$

- Replaces occurrence of clause body (from a previous program in the transformation sequence) by its head.

Applying Unfold/Fold Transformations: Example

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

$\downarrow$ *Unfold*

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

da([H|X1], Y, Z, R) :- app(X1, Y, I1), app([H|I1], Z, R).
```

$\downarrow$ *Unfold*

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

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

$\downarrow$ *Fold*

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

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

Transformations for Optimization

- Application of unfold/fold transformations can achieve :
  - Deforestation (removal of intermediate data structures)
  - Tupling (avoiding multiple traversals of the same data structure).
- Unfold/fold LP transformations are used for "compiling control". Transform a program P to P' s.t. eval. of P' using Prolog's SLD resolution strategy $\equiv$ eval. of P using a more efficient control strategy.
- Certain program specialization techniques for LP are based on application of unfold/fold rules.

LP Transformations for Deduction

- Can be used to construct inductive proofs of program properties.
- Unfolding : Base Case & Finite part of Induction step.
  Folding: Recognize Induction Hypothesis.
- *Illustration:* Transform **thm** s.t. its equivalence with **gen** can be inferred from the syntax of its transformed defn.

```
thm(X) :- gen(X), test(X).
gen([]).
gen([0|X]) :- gen(X).
...
```

$$\xrightarrow{Unf/Fold}$$

```
thm([]).
thm([0|X]) :- thm(X).
gen([]).
gen([0|X]) :- gen(X).
...
```

Infer $\mathtt{thm} \equiv \mathtt{gen}$ *i.e.* $\forall \mathtt{X}\ \mathtt{gen}(\mathtt{X}) \Rightarrow \mathtt{test}(\mathtt{X})$.

A Problem with Folding

- All unfold/fold transformation seq. are **not** semantics preserving.

$$\begin{array}{lcl} \texttt{p :- q.} & \xrightarrow{Fold} & \texttt{p :- p.} \\ \texttt{q.} & & \texttt{q.} \end{array}$$

- Folding may *introduce* circularity in predicate definitions, thereby removing finite proof paths.

- *State of the Art*: [TS84,TS86,KF87,GK94,PP99]
  To preserve semantics, existing transformation systems conservatively impose syntactic restrictions on the clauses participating in folding, *e.g.* only a single clause and/or only non-recursive clauses

Proving Correctness of Transformations

- *Need to prove:* $M(P_i) = M(P_{i+1})$
- $\forall A \in M(P_i)$ $\exists$ proof of A in P_{i+1} (& vice-versa).
- By induction on some measure α of ground atoms.

$P_j (j \leq i)$: $\quad \mathtt{p} :- Bd_1.$

P_i: $\quad \mathtt{q} :- \mathtt{g}, \boxed{Bd_1,}\ \mathtt{g}'. \quad \xrightarrow{Fold} \quad P_{i+1}$: $\quad \mathtt{q} :- \mathtt{g}, \boxed{\mathtt{p}}, \mathtt{g}'.$

- This can only be shown if $\alpha(\mathtt{p}) \prec \alpha(\mathtt{q})$ (by induction hypothesis).
- Syntactic restrictions on clauses of $\mathtt{p}$ ensure this inequality.

Our transformation framework

- Attach abstract measures to program clauses and atoms:
  - Update *clause measures* in each transformation step.
  - Impose (automatically checkable) conditions on *clause measures* to disallow incorrect folding steps.
  - Induct on an ordering of *atom measures* to prove preservation of program semantics.
- The only restrictions in a folding step are on the clause measures and not on the syntax of clauses.
- Previously existing unfold/fold transformation systems can be derived by simply instantiating the measures.

[PPDP '99]

Clause Measures

For any transformation sequence $P_0, P_1, \ldots, P_n$ annotate each clause $C \in P_i$ with a pair $(\gamma^i_{lo}(C), \gamma^i_{hi}(C))$ where:

- $\gamma^i_{lo}(C), \gamma^i_{hi}(C)$ denote the minimum and maximum savings (in proof size) incurred by applying C in a ground proof in P_i.
- a folding step is allowed when the previously accumulated savings are not offset by this folding step.
- In general, the clause measures can come from a commutative group $\langle \mathcal{M}, \oplus \rangle$ where $\oplus$ is monotonic w.r.t. $\succ$.
- Appropriate instantiation of this algebraic structure yields previously existing works, *e.g.* [TS84, TS86, KF87, GK94, PP99].

Unfolding (Revised)

$$\mathtt{q} :- \mathtt{g}, \boxed{\mathtt{p}}, \mathtt{g}'.\ (\gamma, \gamma')$$

$$\mathtt{p} :- Bd_1.\ (\gamma_1, \gamma_1')$$

$$\mathtt{p} :- Bd_2.\ (\gamma_2, \gamma_2')$$

$$\vdots$$

$$\mathtt{p} :- Bd_n.\ (\gamma_n, \gamma_n')$$

$$\xrightarrow{Unf}$$

$$\begin{array}{lcl}
\mathtt{q} :- \mathtt{g}, & Bd_1, & \mathtt{g}'.\ (\gamma \oplus \gamma_1, \gamma' \oplus \gamma_1') \\
\mathtt{q} :- \mathtt{g}, & Bd_2, & \mathtt{g}'.\ (\gamma \oplus \gamma_2, \gamma' \oplus \gamma_2') \\
\vdots & & \\
\mathtt{q} :- \mathtt{g}, & Bd_n, & \mathtt{g}'.\ (\gamma \oplus \gamma_n, \gamma' \oplus \gamma_n')
\end{array}$$

$$\mathtt{p} :- Bd_1.\ (\gamma_1, \gamma_1')$$

$$\vdots$$

$$\mathtt{p} :- Bd_n.\ (\gamma_n, \gamma_n')$$

Folding (Revised)

Program P_j $(j \le i)$		Program P_i			
$\mathsf{p} :- Bd_1.$	(δ_1, δ_1')	$\mathsf{q} :- \mathsf{g},$	$Bd_1,$	$\mathsf{g}'.$	(γ_1, γ_1')
$\vdots$		$\vdots$	$\vdots$		
$\mathsf{p} :- Bd_n.$	(δ_n, δ_n')	$\mathsf{q} :- \mathsf{g},$	$Bd_n,$	$\mathsf{g}'.$	(γ_n, γ_n')

If: $\gamma_1 \succ \delta_1', \gamma_2 \succ \delta_2', \ldots, \gamma_n \succ \delta_n'$

Program P_{i+1}

$$\mathsf{q} :- \mathsf{g}, \boxed{\mathtt{p}}, \mathsf{g}'. \quad (min_{1 \le l \le n}(\gamma_l \ominus \delta_l'),\ max_{1 \le l \le n}(\gamma_l' \ominus \delta_l))$$

Relating Atom and Clause Measures

- Roughly, for a clause $C = (A{:}\!-\ Bd)$ in program P_i appearing in a (canonical) proof, the clause measures of C bound the difference in atom measures of head and body.

$$\gamma^i_{lo}(C) \ \preceq \alpha(A) \ominus \sum_{B \in Bd} \alpha(B) \ \preceq \gamma^i_{hi}(C)$$

- By imposing $\gamma^i_{hi}(D) \prec \gamma^i_{lo}(C)$ in a folding step, we ensure $\alpha(\mathtt{p}) \prec \alpha(\mathtt{q})$.

$P_j (j \leq i)$: $\qquad D = (\mathtt{p} : -Bd_1.)$

P_i: $\qquad C = (\mathtt{q} : -\mathtt{g}, \boxed{Bd_1,} \ \mathtt{g}'.) \quad \xrightarrow{Fold} \quad P_{i+1}: \quad \cdots$

Why approximate clause measures?

Clause measures denote the savings in proof size by applying that clause in a ground proof.

Cannot compute exact savings for clauses introduced by folding.

$$
\begin{array}{l}
C_1\text{: } \texttt{p :- r, t.} \quad (x_1) \\
C_2\text{: } \texttt{p :- s, t.} \quad (x_2) \\
C_3\text{: } \texttt{q :- r.} \quad (x_3) \\
C_4\text{: } \texttt{q :- s.} \quad (x_4)
\end{array}
\quad \xrightarrow{\mathit{Fold}} \quad
\begin{array}{l}
C'\text{: } \texttt{p :- q, t.} \quad (?) \\
C_3\text{: } \texttt{q :- r.} \quad (x_3) \\
C_4\text{: } \texttt{q :- s.} \quad (x_4)
\end{array}
$$

The savings by using C' in a proof depends on whether C_3 or C_4 is used to resolve `q` in that proof.

Implications of our framework

- Application of transformations are restricted only by mechanically checkable conditions on clause measures, and not by pgm. syntax.
- Allows us to identify an abstract program property (Measure Consistency) which ensures correctness of unfold/fold transformations. Can introduce new transformations without redoing correctness proofs.
- Existing Txf. Systems are specific instances of our framework $\Rightarrow$ allows us to understand strengths and limitations of existing systems.
- Construction of new unfold/fold txf. systems which is more powerful (in terms of transformation sequences allowed).

Application in Verification

- Used to build an automated prover for temporal properties of parameterized concurrent systems *e.g.* proving mutual exclusion in n process token ring for any n

- *1. Encode transition rel. and temporal property as LP preds.*
  2. Reduce verification proof obligation to pred. equiv. $\mathtt{p} \equiv \mathtt{q}$.
  3. Transform defn. of $\mathtt{p}$ *and* $\mathtt{q}$ *s.t.* $\mathtt{p} \equiv \mathtt{q}$ *inferred from syntax.*

- Case studies: Java Meta-Locking algo., Cache coherence protocols

- Experimental numbers for mutual exclusion in Java Meta-locking algo:

Running time:	5 mins, 23 secs
# Unfolding:	1981
# Folding:	311

[TACAS'00]

Extensions and Future Work

- This work has been extended to obtain similar generalized unfold/fold transformations for normal logic programs [ASIAN99].

  Uniform proof of correctness w.r.t. different semantics of negation : well-founded model, stable model, partial stable model.

- Can we develop similar abstract transformation frameworks for studying correctness of unf./fold in other programming paradigms *e.g.*
  - *functional languages* [Sands96]
  - *process algebraic languages* [FS98]