Efficiently Reasoning about Programs

Neil Immerman

College of Computer and Information Sciences
University of Massachusetts, Amherst
Amherst, MA, USA

people.cs.umass.edu/~immerman
Thm. Turing 1936: Halt undecidable.

Arithmetic Hierarchy

\( \text{FO}(N) \)

r.e. complete

Halt

co-r.e. complete

Halt

\( \text{FO} \exists(N) \)

Recursive

\( \text{FO} \forall(N) \)

Primitive Recursive

\( \text{SO}[2^{\Omega(1)}] \)

EXPTIME

\( \text{QSAT} \)

PSPACE complete

\( \text{SO}[n^{O(1)}] \)

PSPACE

PTIME Hierarchy

\( \text{SO} \)

NP complete

\( \text{NP} \cap \text{co-NP} \)

P complete

P

\( \text{FO}[n^{O(1)}] \)

PSPACE complete

\( \text{FO}(\text{LFP}) \)

‟truly feasible‟

\( \text{AC}^1 \)

\( \text{FO}(\text{CFL}) \)

\( \text{FO}(\text{TC}) \)

\( \text{FO}(\text{DTC}) \)

\( \text{FO}(\text{REGULAR}) \)

\( \text{FO}(\text{COUNT}) \)

\( \text{FO} \)

LOGTIME Hierarchy

\( \text{ThC}^0 \)

\( \text{AC}^0 \)
**Thm.** [Turing 1936] Halt undecidable.
Halt is r.e. complete
Halt is r.e. complete

\[ \exists w \in \Sigma^* (\alpha(w)) \iff M_\alpha \in \text{Halt} \]
Halt is r.e. complete

\[ \exists w \in \Sigma^* (\alpha(w)) \iff M_{\alpha} \in \text{Halt} \]

Any arbitrary search problem can be translated to Halt.
- Halt is r.e. complete
- \( \exists w \in \sum^* (\alpha(w)) \iff M_\alpha \in \text{Halt} \)
- Any arbitrary search problem can be translated to Halt.
- **Cannot check correctness** of arbitrary input program.
Halt is r.e. complete

∃w ∈ Σ* (α(w)) ⇔ M_α ∈ Halt

Any arbitrary search problem can be translated to Halt.

Cannot check correctness of arbitrary input program.

Long-Term Societal Goal:
Halt is r.e. complete

\[ \exists w \in \Sigma^* (\alpha(w)) \iff M_\alpha \in \text{Halt} \]

Any arbitrary search problem can be translated to Halt.

Cannot check correctness of arbitrary input program.

Long-Term Societal Goal:

Automatic help to produce programs that are
Halt is r.e. complete

\[ \exists w \in \Sigma^* (\alpha(w)) \iff M_\alpha \in \text{Halt} \]

Any arbitrary search problem can be translated to Halt.

**Cannot check correctness** of arbitrary input program.

**Long-Term Societal Goal:**

**Automatic help** to produce programs that are

**certified** to **safely** and **faithfully**
- Halt is r.e. complete
- \( \exists w \in \Sigma^* (\alpha(w)) \iff M_\alpha \in \text{Halt} \)
- Any arbitrary search problem can be translated to Halt.
- **Cannot check correctness** of arbitrary input program.

- **Long-Term Societal Goal:**
  - **Automatic help** to produce programs that are
  - **certified** to **safely** and **faithfully**
  - **do** what they **should do**
Halt is r.e. complete

\[ \exists w \in \sum^* (\alpha(w)) \iff M_\alpha \in Halt \]

Any arbitrary search problem can be translated to Halt.

Cannot check correctness of arbitrary input program.

Long-Term Societal Goal:

Automatic help to produce programs that are certified to safely and faithfully do what they should do and not do what they should not do.
**Thm.** [Turing 1936] Halt undecidable.

**Thm.** [Cook 1971] SAT is NP complete.
SAT is NP complete.
SAT is NP complete.

\[ \exists w \in \Sigma^{n^{O(1)}} \ (\alpha(w)) \iff \varphi_{\alpha} \in \text{SAT} \]
SAT is NP complete.

\[ \exists w \in \Sigma^{O(1)} (\alpha(w)) \iff \varphi_{\alpha} \in \text{SAT} \]

Arbitrary exponential search problem is translated to SAT.
➤ **SAT** is NP complete.

➤ \( \exists w \in \Sigma^{n^{O(1)}} (\alpha(w)) \iff \varphi_\alpha \in \text{SAT} \)

➤ Arbitrary exponential search problem is translated to **SAT**.

➤ **SAT** is not **feasible** in the **worst case**.
- **SAT** is NP complete.
- \( \exists w \in \Sigma^{n^{O(1)}} (\alpha(w)) \iff \varphi_\alpha \in \text{SAT} \)
- Arbitrary exponential search problem is translated to SAT.
- **SAT** is not feasible in the worst case.
- Every reasonable search problem can be encoded as an instance of SAT.
SAT is NP complete.

\[ \exists w \in \Sigma^{n^{O(1)}} (\alpha(w)) \iff \varphi_{\alpha} \in \text{SAT} \]

Arbitrary exponential search problem is translated to SAT.

SAT is not feasible in the worst case.

Every reasonable search problem can be encoded as an instance of SAT.

Great progress in design of SAT Solvers.
SAT is NP complete.

\[ \exists w \in \Sigma^{n^{O(1)}} \binom{\alpha(w)}{\varphi_{\alpha} \in \text{SAT}} \]

Arbitrary exponential search problem is translated to SAT.

SAT is not feasible in the worst case.

Every reasonable search problem can be encoded as an instance of SAT.

Great progress in design of SAT Solvers.

Fast, general-purpose problem solvers.
Verification by Reduction to SAT

When and why does this work?

▶ How general and powerful can we make it?
When and why does this work?
Verification by Reduction to SAT

When and why does this work?
How general and powerful can we make it?
Static

1. Read entire input
2. Compute boolean query $Q(input)$
3. Classic Complexity Classes are static: FO, NC, P, NP, …
**Static**

1. Read entire input
2. Compute boolean query $Q(\text{input})$
3. Classic Complexity Classes are static: FO, NC, P, NP, ... 
4. What is the fastest way **upon reading the entire input**, to compute the query?
Background: Dynamic Complexity

**Static**
1. Read entire input
2. Compute boolean query $Q(input)$
3. Classic Complexity Classes are static: FO, NC, P, NP, …
4. What is the fastest way upon reading the entire input, to compute the query?

**Dynamic**
1. Long series of Inserts, Deletes, Changes, and, Queries
2. On query, very quickly compute $Q(current\, database)$
3. Dynamic Complexity Classes: Dyn-FO, Dyn-NC
Background: Dynamic Complexity

**Static**
1. Read entire input
2. Compute boolean query $Q\text{(input)}$
3. Classic Complexity Classes are static: FO, NC, P, NP, . . .
4. What is the fastest way **upon reading the entire input**, to compute the query?

**Dynamic**
1. Long series of Inserts, Deletes, Changes, and, Queries
2. On **query**, **very quickly** compute $Q\text{(current database)}$
3. Dynamic Complexity Classes: Dyn-FO, Dyn-NC
4. What **additional information** should we maintain? — **auxiliary data structure**
Dynamic (Incremental) Applications

- Databases
- LaTexing a file
- Performing a calculation
- Processing a visual scene
- Understanding a natural language
- Verifying a circuit
- Verifying and compiling a program
Dynamic (Incremental) Applications

- Databases
- LaTeXing a file
- Performing a calculation
- Processing a visual scene
- Understanding a natural language
- Verifying a circuit
- Verifying and compiling a program
- Surviving in the wild
Parity

<table>
<thead>
<tr>
<th>Current Database: S</th>
<th>Request</th>
<th>Auxiliary Data: b</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Current Database: $S$</td>
<td>Request</td>
<td>Auxiliary Data: $b$</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------</td>
<td>------------------</td>
</tr>
<tr>
<td>00000000</td>
<td><strong>ins(3,S)</strong></td>
<td>0</td>
</tr>
<tr>
<td>Current Database: $S$</td>
<td>Request</td>
<td>Auxiliary Data: $b$</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------</td>
<td>---------------------</td>
</tr>
<tr>
<td>00000000</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>00100000</td>
<td><strong>ins(3,S)</strong></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Parity

<table>
<thead>
<tr>
<th>Current Database: $S$</th>
<th>Request</th>
<th>Auxiliary Data: $b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>00100000</td>
<td><em>ins</em>(3,$S$)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td><em>ins</em>(7,$S$)</td>
<td></td>
</tr>
</tbody>
</table>

$$S'(x) \equiv S(x) \lor x = a \quad S'(x) \equiv S(x) \land x \neq a$$

$$b' \equiv (b \land S(a)) \lor (b' \equiv (b \land \neg S(a)) \lor (\neg b \land \neg S(a)))$$
### Parity

<table>
<thead>
<tr>
<th>Current Database: $S$</th>
<th>Request</th>
<th>Auxiliary Data: $b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>00100000</td>
<td><strong>ins(3, S)</strong></td>
<td>1</td>
</tr>
<tr>
<td>00100001</td>
<td><strong>ins(7, S)</strong></td>
<td>0</td>
</tr>
<tr>
<td>Current Database: $S$</td>
<td>Request</td>
<td>Auxiliary Data: $b$</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
<td>-------------------</td>
</tr>
<tr>
<td>00000000</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>00100000</td>
<td><strong>ins</strong> (3, $S$)</td>
<td>1</td>
</tr>
<tr>
<td>00100001</td>
<td><strong>ins</strong> (7, $S$)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td><strong>del</strong> (3, $S$)</td>
<td></td>
</tr>
</tbody>
</table>

For $x \equiv S(x) \lor x = a$ $S'(x) \equiv S(x) \land x \neq a$

$b' \equiv (b \land S(a)) \lor b' \equiv (b \land \neg S(a)) \lor (\neg b \land \neg S(a))$
Parity

<table>
<thead>
<tr>
<th>Current Database: $S$</th>
<th>Request</th>
<th>Auxiliary Data: $b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>00100000</td>
<td><strong>ins</strong>(3,$S$)</td>
<td>1</td>
</tr>
<tr>
<td>00100001</td>
<td><strong>ins</strong>(7,$S$)</td>
<td>0</td>
</tr>
<tr>
<td>00000001</td>
<td><strong>del</strong>(3,$S$)</td>
<td>1</td>
</tr>
</tbody>
</table>
Parity $\in$ Dyn-FO

<table>
<thead>
<tr>
<th>Current Database: $S$</th>
<th>Request</th>
<th>Auxiliary Data: $b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>00100000</td>
<td>ins(3,S)</td>
<td>1</td>
</tr>
<tr>
<td>00100001</td>
<td>ins(7,S)</td>
<td>0</td>
</tr>
<tr>
<td>00000001</td>
<td>del(3,S)</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
\text{ins}(a,S) \quad \equiv \quad S'(x) \equiv S(x) \lor x = a
\]
\[
b' \equiv (b \land S(a)) \lor (\neg b \land \neg S(a))
\]
\[
\text{del}(a,S) \quad \equiv \quad S'(x) \equiv S(x) \land x \neq a
\]
\[
b' \equiv (b \land \neg S(a)) \lor (\neg b \land S(a))
\]
Parity

- Does binary string $w$ have an odd number of 1’s?
- **Static**: $\text{TIME}[n], \text{FO}[\Omega(\log n / \log \log n)]$
- **Dynamic**: $\text{Dyn-TIME}[1], \text{Dyn-FO}$
Parity
- Does binary string $w$ have an odd number of 1’s?
- **Static:** TIME[$n$], FO[$\Omega(\log n / \log \log n)$]
- **Dynamic:** Dyn-TIME[1], Dyn-FO

REACH$_u$
- Is $t$ reachable from $s$ in undirected graph $G$?
- **Static:** not in FO, requires FO[$\Omega(\log n / \log \log n)$]
- **Dynamic:** in Dyn-FO  [Patnaik, I]
Dynamic Examples

Parity
- Does binary string \( w \) have an odd number of 1’s?
- **Static:** \( \text{TIME}[n], \text{FO}[\Omega(\log n / \log \log n)] \)
- **Dynamic:** \( \text{Dyn-TIME}[1], \text{Dyn-FO} \)

\( \text{REACH}_u \)
- Is \( t \) reachable from \( s \) in undirected graph \( G \)?
- **Static:** not in FO, requires \( \text{FO}[\Omega(\log n / \log \log n)] \)
- **Dynamic:** in \( \text{Dyn-FO} \) [Patnaik, I]

connectivity, minimum spanning trees, \( k \)-edge connectivity, . . .
Fact: [Dong & Su] \( \text{REACH}(\text{acyclic}) \in \text{DynFO} \)
Fact: [Dong & Su] REACH(acyclic) \(\in\) DynFO

\[
\text{ins}(a, b, E) : P'(x, y) \equiv P(x, y) \lor (P(x, a) \land P(b, y))
\]
Fact: [Dong & Su] \( \text{REACH(acyclic)} \in \text{DynFO} \)

\[
\text{ins}(a, b, E) : P'(x, y) \equiv P(x, y) \lor (P(x, a) \land P(b, y))
\]

![Graph]

\[
\text{del}(a, b, E): \quad P'(x, y) \equiv P(x, y) \land \left[ \neg(P(x, a) \land P(b, y)) \right.

\lor (\exists uv)(P(x, u) \land E(u, v) \land P(v, y)

\land P(u, a) \land \neg P(v, a) \land (a \neq u \lor b \neq v))\right]
\]
Reachability Problems

\[
\begin{align*}
\text{REACH} &= \left\{ G \mid G \text{ directed}, s \xrightarrow[*]{G} t \right\} \\
\text{REACH}_d &= \left\{ G \mid G \text{ directed}, \text{ outdegree} \leq 1 s \xrightarrow[*]{G} t \right\} \\
\text{REACH}_u &= \left\{ G \mid G \text{ undirected}, s \xrightarrow[*]{G} t \right\} \\
\text{REACH}_a &= \left\{ G \mid G \text{ alternating}, s \xrightarrow[*]{G} t \right\}
\end{align*}
\]
Facts about dynamic REACHABILITY Problems:

\[ \text{REACH(acyclic)} \in \text{Dyn-FO} \quad [\text{DS}] \]

\[ \text{REACH}_d \in \text{Dyn-QF} \quad [\text{H}] \]

\[ \text{REACH}_u \in \text{Dyn-FO} \quad [\text{PI}] \]

\[ \text{REACH} \in \text{Dyn-FO(COUNT)} \quad [\text{H}] \]

\[ \text{PAD(} \text{REACH}_a \text{)} \in \text{Dyn-FO} \quad [\text{PI}] \]
**Thm.** \( \text{REACH} \in \text{Dyn-FO} \)

[Samir Datta, Raghav Kulkarni, Anish Mukherjee, Thomas Schwentick, Thomas Zeume]


\( \text{REACH} \leq \text{Matrix Rank} \in \text{Dyn-FO} \)
**Thm. 1** [Hesse]  \( \text{REACH}_d(acyclic) \in \text{Dyn-FO} \)

**proof:** Maintain \( E, E^*, D \) (outdegree = 1).

\( \text{ins}(a, b, E) \): (ignore if outdegree or acyclicity violated)

\[
E'(x, y) \equiv E(x, y) \lor (x = a \land y = b)
\]

\[
D'(x) \equiv D(x) \lor x = a
\]

\[
E^*(x, y) \equiv E^*(x, y) \lor (E^*(x, a) \land E^*(b, y))
\]
**Thm. 1** [Hesse] \(\text{REACH}_d(\text{acyclic}) \in \text{Dyn-FO}\)

**proof:** Maintain \(E, E^*, D\) (outdegree = 1).

**ins**\((a, b, E)\): (ignore if outdegree or acyclicity violated)

\[
E'(x, y) \equiv E(x, y) \lor (x = a \land y = b)
\]
\[
D'(x) \equiv D(x) \lor x = a
\]
\[
E^{*'}(x, y) \equiv E^*(x, y) \lor (E^*(x, a) \land E^*(b, y))
\]

**del**\((a, b, E)\):

\[
E'(x, y) \equiv E(x, y) \land (x \neq a \lor y \neq b)
\]
\[
D'(x) \equiv D(x) \land x \neq a
\]
\[
E^{*'}(x, y) \equiv E^*(x, y) \land \neg (E^*(x, a) \land E(a, b) \land E^*(b, y))
\]

\(\square\)
Reasoning About reachability – can we get to y from x by following a sequence of pointers –
**Reasoning About reachability** – can we get to \( y \) from \( x \) by following a sequence of pointers – is **crucial** for understanding programs and **proving** that they meet their specifications.
In general, reasoning about reachability is **undecidable**.

- Can express tilings and thus runs of Turing Machines.
In general, reasoning about reachability is **undecidable**.

- Can express tilings and thus runs of Turing Machines.
- Even worse, can express **finite path** and thus **finite** and thus **standard natural numbers**. Thus satisfiability of FO(TC) is as hard as the Arithmetic Hierarchy [Avron].

- For now, restrict to acyclic fields.
For now, **restrict** to **acyclic** fields.

- $n(x, y)$ means that $x$ points to $y$. 

- For now, restrict to acyclic fields.

- $n(x, y)$ means that $x$ points to $y$.

- Use predicate symbol, $n^*$, but not $n$. 
For now, restrict to acyclic fields.

$n(x, y)$ means that $x$ points to $y$.

Use predicate symbol, $n^*$, but not $n$.

The following axioms assure that $n^*$ is the reflexive transitive closure of some acyclic, functional $n$. 
For now, restrict to acyclic fields.

$n(x, y)$ means that $x$ points to $y$.

Use predicate symbol, $n^*$, but not $n$.

The following axioms assure that $n^*$ is the reflexive transitive closure of some acyclic, functional $n$.

\[
\text{acyclic} \equiv \forall xy \ (n^*(x, y) \land n^*(y, x) \iff x = y)
\]
For now, restrict to acyclic fields.

$n(x, y)$ means that $x$ points to $y$.

Use predicate symbol, $n^*$, but not $n$.

The following axioms assure that $n^*$ is the reflexive transitive closure of some acyclic, functional $n$.

\[
\text{acyclic} \equiv \forall xy \ (n^*(x, y) \land n^*(y, x) \iff x = y)
\]

\[
\text{transitive} \equiv \forall xyz \ (n^*(x, y) \land n^*(y, z) \to n^*(x, z))
\]

- For now, **restrict** to **acyclic** fields.
- \( n(x, y) \) means that \( x \) points to \( y \).
- Use predicate symbol, \( n^* \), **but not** \( n \).
- The following axioms assure that \( n^* \) is the reflexive transitive closure of some acyclic, functional \( n \).

\[
\text{acyclic} \equiv \forall xy (n^*(x, y) \land n^*(y, x) \iff x = y)
\]

\[
\text{transitive} \equiv \forall xyz (n^*(x, y) \land n^*(y, z) \to n^*(x, z))
\]

\[
\text{linear} \equiv \forall xyz (n^*(x, y) \land n^*(x, z) \to n^*(y, z) \lor n^*(z, y))
\]
Assume acyclic, transitive and linear axioms, as integrity constraints.
Assume acyclic, transitive and linear axioms, as integrity constraints.

Automatically transform a program manipulating linked lists to an $\forall \exists$ correctness condition.
Effectively-Propositional Reasoning about Reachability in Linked Data Structures

- Assume acyclic, transitive and linear axioms, as integrity constraints.
- Automatically transform a program manipulating linked lists to an $\forall \exists$ correctness condition.
- Using Hesse’s dynQF algorithm for REACH$_d$, these $\forall \exists$ formulas are closed under weakest precondition.
Assume acyclic, transitive and linear axioms, as integrity constraints.

Automatically transform a program manipulating linked lists to an $\forall \exists$ correctness condition.

Using Hesse’s dynQF algorithm for $\text{REACH}_d$, these $\forall \exists$ formulas are closed under weakest precondition.

The negation of the correctness condition is $\exists \forall$, thus equi-satisfiable with a propositional formula (EPR).
Assume acyclic, transitive and linear axioms, as integrity constraints.

Automatically transform a program manipulating linked lists to an $\forall\exists$ correctness condition.

Using Hesse’s dynQF algorithm for $\text{REACH}_d$, these $\forall\exists$ formulas are closed under weakest precondition.

The negation of the correctness condition is $\exists\forall$, thus equi-satisfiable with a propositional formula (EPR).

Use a SAT solver to automatically prove correctness or find counter-example runs, typically in only a few seconds.
Effectively-Propositional Reasoning (EPR)

- FO-SAT is **undecidable** (co-r.e. complete).
► **FO-SAT** is **undecidable** (co-r.e. complete).
► **EPR**: $\exists \forall$ formulas; no function symbols.
Effectively-Propositional Reasoning (EPR)

- FO-SAT is **undecidable** (co-r.e. complete).
- EPR: $\exists \forall$ formulas; no function symbols.
- Constant symbols: $c_1, \ldots, c_k$
Effectively-Propositional Reasoning (EPR)

- **FO-SAT** is **undecidable** (co-r.e. complete).
- **EPR**: $\exists \forall$ formulas; no function symbols.
- constant symbols: $c_1, \ldots, c_k$
- $\varphi = \exists x_1 \ldots x_s \forall y_1 \ldots y_t (\alpha(\overline{x}, \overline{t}, \overline{c}))$

EPR-SAT $\in \Sigma_p^2$ (2nd level polynomial-time hierarchy)

If $t$ is fixed, then reducible to SAT.

Z3 seems to do very well for us on EPR-SAT.
Effectively-Propositional Reasoning (EPR)

- **FO-SAT** is **undecidable** (co-r.e. complete).
- **EPR**: $\exists \forall$ formulas; no function symbols.
- Constant symbols: $c_1, \ldots, c_k$
- $\phi = \exists x_1 \ldots x_s \forall y_1 \ldots y_t (\alpha(\bar{x}, \bar{t}, \bar{c}))$
- Small model: $\phi \in \text{FO-SAT}$ iff has model size $\leq k + s$. 

FO-SAT is undecidable (co-r.e. complete).

EPR: $\exists \forall$ formulas; no function symbols.

constant symbols: $c_1, \ldots, c_k$

$\varphi = \exists x_1 \ldots x_s \forall y_1 \ldots y_t (\alpha(x, t, c))$

small model: $\varphi \in \text{FO-SAT}$ iff has model size $\leq k + s$.

EPR-SAT $\in \Sigma^P_2$ (2nd level polynomial-time hierarchy)
Effectively-Propositional Reasoning (EPR)

- **FO-SAT** is **undecidable** (co-r.e. complete).
- **EPR**: \( \exists \forall \) formulas; no function symbols.
- constant symbols: \( c_1, \ldots, c_k \)
- \( \varphi = \exists x_1 \ldots x_s \forall y_1 \ldots y_t (\alpha(\bar{x}, \bar{t}, \bar{c})) \)
- **small model**: \( \varphi \in \text{FO-SAT} \) iff has model size \( \leq k + s \).
- **EPR-SAT** \( \in \Sigma_2^p \) (2nd level polynomial-time hierarchy)
- If \( t \) is fixed, then reducible to **SAT**.
Effectively-Propositional Reasoning (EPR)

- **FO-SAT** is undecidable (co-r.e. complete).
- **EPR**: $\exists \forall$ formulas; no function symbols.
- Constant symbols: $c_1, \ldots, c_k$
- $\varphi = \exists x_1 \ldots x_s \forall y_1 \ldots y_t (\alpha(\overline{x}, \overline{t}, \overline{c}))$
- Small model: $\varphi \in \text{FO-SAT}$ iff has model size $\leq k + s$.
- **EPR-SAT** $\in \Sigma^p_2$ (2nd level polynomial-time hierarchy)
- If $t$ is fixed, then reducible to **SAT**.
- Z3 seems to do very well for us on **EPR-SAT**.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>P, Q</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Solving time (Z3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLL: reverse</td>
<td>2</td>
<td>2</td>
<td>11</td>
<td>2</td>
<td>133</td>
<td>3</td>
</tr>
<tr>
<td>SLL: filter</td>
<td>5</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>280</td>
<td>4</td>
</tr>
<tr>
<td>SLL: create</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>36</td>
<td>3</td>
</tr>
<tr>
<td>SLL: delete</td>
<td>5</td>
<td>0</td>
<td>12</td>
<td>1</td>
<td>152</td>
<td>3</td>
</tr>
<tr>
<td>SLL: deleteAll</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>106</td>
<td>3</td>
</tr>
<tr>
<td>SLL: insert</td>
<td>8</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>178</td>
<td>3</td>
</tr>
<tr>
<td>SLL: find</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>64</td>
<td>3</td>
</tr>
<tr>
<td>SLL: last</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>74</td>
<td>3</td>
</tr>
<tr>
<td>SLL: merge</td>
<td>14</td>
<td>2</td>
<td>31</td>
<td>2</td>
<td>2255</td>
<td>3</td>
</tr>
<tr>
<td>SLL: rotate</td>
<td>6</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>73</td>
<td>3</td>
</tr>
<tr>
<td>SLL: swap</td>
<td>14</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>965</td>
<td>5</td>
</tr>
<tr>
<td>DLL: fix</td>
<td>5</td>
<td>2</td>
<td>11</td>
<td>2</td>
<td>121</td>
<td>3</td>
</tr>
<tr>
<td>DLL: splice</td>
<td>10</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>167</td>
<td>4</td>
</tr>
</tbody>
</table>
**Thm. 2** [Hesse] Reachability of functional graphs is in DynQF.

**proof idea:** If adding an edge, $e$, would create a cycle, then we maintain relation $p^*$ – the path relation without the edge completing the cycle – as well as $E^*$, $E$ and $D$.

Surprisingly this can all be maintained via quantifier-free formulas, **without remembering which edges we are leaving out** in computing $p^*$.
Thm. 2 [Hesse] Reachability of functional graphs is in DynQF.

proof idea: If adding an edge, $e$, would create a cycle, then we maintain relation $p^*$ – the path relation without the edge completing the cycle – as well as $E^*$, $E$ and $D$.

Surprisingly this can all be maintained via quantifier-free formulas, **without remembering which edges we are leaving out** in computing $p^*$. □

Using Thm. 2, the above methodology has been extended to cyclic deterministic graphs.

Extensions

Extensions to EPR: we can have functions symbols, as long as we can guarantee the the closure of the function symbols on any finite set remains finite.
Extensions

- Extensions to EPR: we can have functions symbols, as long as we can guarantee the closure of the function symbols on any finite set remains finite.
- What data structures can we handle: lists, doubly linked lists, cyclic lists; binary trees, ...
Extensions

- Extensions to EPR: we can have functions symbols, as long as we can guarantee the closure of the function symbols on any finite set remains finite.
- What data structures can we handle: lists, doubly linked lists, cyclic lists; binary trees, . . .
- The [CAV13] and [POPL14] papers assume that correct invariants are given for each loop. On-going work to automatically generate and prove loop invariants:
Extensions

- Extensions to EPR: we can have functions symbols, as long as we can guarantee the closure of the function symbols on any finite set remains finite.
- What data structures can we handle: lists, doubly linked lists, cyclic lists; binary trees, . . .
- The [CAV13] and [POPL14] papers assume that correct invariants are given for each loop. On-going work to automatically generate and prove loop invariants:
Extensions

- Extensions to EPR: we can have functions symbols, as long as we can guarantee the closure of the function symbols on any finite set remains finite.
- What data structures can we handle: lists, doubly linked lists, cyclic lists; binary trees, ... 
- The [CAV13] and [POPL14] papers assume that correct invariants are given for each loop. On-going work to automatically generate and prove loop invariants:
- Padon, I, Karbyshev, Sagiv, Shoham, “Decidability of Inferring Inductive Invariants” [POPL16].
Extensions

- Extensions to EPR: we can have functions symbols, as long as we can guarantee the the closure of the function symbols on any finite set remains finite.
- What data structures can we handle: lists, doubly linked lists, cyclic lists; binary trees, . . .
- The [CAV13] and [POPL14] papers assume that correct invariants are given for each loop. On-going work to automatically generate and prove loop invariants:
  - Padon, I, Karbyshev, Sagiv, Shoham, “Decidability of Inferring Inductive Invariants” [POPL16].
  - Padon, McMillan, Panda, Sagiv, Shoham, “Ivy: Safety Verification by Interactive Generalization” [PLDI16].
Extensions

- Extensions to EPR: we can have functions symbols, as long as we can guarantee the closure of the function symbols on any finite set remains finite.
- What data structures can we handle: lists, doubly linked lists, cyclic lists; binary trees, . . .
- The [CAV13] and [POPL14] papers assume that correct invariants are given for each loop. On-going work to automatically generate and prove loop invariants:
  - Padon, I, Karbyshev, Sagiv, Shoham, “Decidability of Inferring Inductive Invariants” [POPL16].
  - Padon, McMillan, Panda, Sagiv, Shoham, “Ivy: Safety Verification by Interactive Generalization” [PLDI16].
  - Karbyshev, Bjorner, Itzhaky, Rinetzky, Shoham, “Property-Directed Inference of Universal Invariants or Proving Their Absence” [CAV15].
Deductive verification by reductions to EPR

Program $\text{Tr} \exists^* \forall^*$

Universal $\forall^*$

Invariant $\text{Inv}$

$\text{Inv} \implies \varphi$

Universal Desired Property $\forall^* \exists^*$

Front-End

Formula

$\text{Inv}(X) \land \exists \text{Tr} \exists \land \neg \text{Inv}(X')$

EPR Solver

Counterexample to Induction (CTI)

Proof

Y

N
When does this work?
When does this work?
When doesn’t this work?
- $\text{Init} \rightarrow \text{Inv}; \quad \text{Inv} \land \text{Tr} \rightarrow \text{Inv}'; \quad \text{Inv} \rightarrow \text{Safe}$
Simple Example: loop Invariants

1: x := 1;
2: y := 2;
while * do {
    3: assert \text{odd}[x];
    4: x := x + y;
    5: y := y + 2
}
6:
Simple Example: loop Invariants

$\text{Inv} = \text{odd}[x] \land \neg \text{odd}[y]$

1: $x := 1$
2: $y := 2$
while * do {
3: assert odd[$x$];
4: $x := x + y$;
5: $y := y + 2$
}
6:
Herbrand Thm. \( \varphi \) universal \( \Rightarrow \)

\[ \varphi \in \text{FO-SAT} \iff \varphi \text{ has Herbrand model, } \mathcal{H} \models \varphi \]
Herbrand Thm. \( \varphi \) universal \( \Rightarrow \)
\[ \varphi \in \text{FO-SAT} \Leftrightarrow \varphi \text{ has Herbrand model, } \mathcal{H} \models \varphi \]

Cor. Complete FO-UNSAT methodology:
Herbrand Thm. \( \varphi \) universal \( \Rightarrow \)

\[ \varphi \in \text{FO-SAT} \iff \varphi \text{ has Herbrand model, } \mathcal{H} \models \varphi \]

Cor. Complete FO-UNSAT methodology:

Skolemize \( \varphi \): \( \varphi_S \) is universal:

\[ \varphi_S = \forall \bar{x} \ (\alpha(\bar{x})) \]

\[ \varphi \in \text{FO-SAT} \iff \varphi_S \in \text{FO-SAT} \]
Herbrand Thm. \( \varphi \) universal \( \Rightarrow \)

\[ \varphi \in \text{FO-SAT} \iff \varphi \text{ has Herbrand model, } \mathcal{H} \models \varphi \]

Cor. Complete FO-UNSAT methodology:

Skolemize \( \varphi \): \( \varphi_S \) is universal: \( \varphi_S = \forall \overline{x} (\alpha(\overline{x})) \);

\[ \varphi \in \text{FO-SAT} \iff \varphi_S \in \text{FO-SAT} \]

\( \text{grnd}(\alpha) \stackrel{\text{def}}{=} \{ \alpha(\overline{t}) \mid \overline{t} \in |\mathcal{H}| \} \)
- **Herbrand Thm.** \( \varphi \) universal \( \Rightarrow \)
  \[ \varphi \in \text{FO-SAT} \iff \varphi \text{ has Herbrand model, } \mathcal{H} \models \varphi \]
- **Cor.** Complete FO-UNSAT methodology:
- Skolemize \( \varphi \): \( \varphi_S \) is universal: \( \varphi_S = \forall \bar{X} (\alpha(\bar{X})) \);
  \[ \varphi \in \text{FO-SAT} \iff \varphi_S \in \text{FO-SAT} \]
- \( \text{grnd}(\alpha) \overset{\text{def}}{=} \{ \alpha(\bar{t}) \mid \bar{t} \in |H| \} \)
- \( \varphi \in \text{FO-UNSAT} \iff \text{grnd}(\alpha) \in \text{UNSAT} \)
Herbrand Thm. \( \varphi \) universal \( \Rightarrow \)
\( \varphi \in \text{FO-SAT} \iff \varphi \) has Herbrand model, \( \mathcal{H} \models \varphi \)

Cor. Complete FO-UNSAT methodology:

Skolemize \( \varphi \): \( \varphi_S \) is universal: 
\( \varphi_S = \forall \overline{x} \left( \alpha(\overline{x}) \right) \)

\( \varphi \in \text{FO-SAT} \iff \varphi_S \in \text{FO-SAT} \)

\( \text{grnd}(\alpha) \overset{\text{def}}{=} \{ \alpha(\overline{t}) \mid \overline{t} \in |\mathcal{H}| \} \)

\( \varphi \in \text{FO-UNSAT} \iff \text{grnd}(\alpha) \in \text{UNSAT} \)

Herbrand Thm. \( \varphi \) universal \( \Rightarrow \)

\( \varphi \in \text{FO-SAT} \iff \varphi \) has Herbrand model, \( \mathcal{H} \models \varphi \)

Cor. Complete FO-UNSAT methodology:

Skolemize \( \varphi \): \( \varphi_S \) is universal: \( \varphi_S = \forall \bar{x} \; (\alpha(\bar{x})) \);

\( \varphi \in \text{FO-SAT} \iff \varphi_S \in \text{FO-SAT} \)

grnd(\( \alpha \)) \( \overset{\text{def}}{=} \{ \alpha(\bar{t}) \mid \bar{t} \in |\mathcal{H}| \} \)

\( \varphi \in \text{FO-UNSAT} \iff \text{grnd}(\alpha) \in \text{UNSAT} \)


Can Understand Decidability of Checking FO Inductive Invariants, via bounded depth of nesting of functions in \( \bar{t} \) needed for unsatisfiability.
Thank You!

Anindya Banerjee, Bill Hesse, Yotam Feldman, Shachar Itzhaky, Aleksandr Karbyshev, Ori Lahav, Aleksandar Nanevski, Oded Padon, Sushant Patnaik, Mooly Sagiv, Sharon Shoham