ALLOY*: General-Purpose Higher-Order Relational Constraint Solver

Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, Daniel Jackson
{aleks,jnear,eskang,dnj}@csail.mit.edu

ICSE 2015
Florence, Italy
**What is ALLOY\(^*\)**

**ALLOY\(^*\):** a more powerful version of the alloy analyzer
**ALLOY**: a more powerful version of the alloy analyzer

**alloy**: general-purpose relational specification language

**alloy analyzer**: automated bounded solver for alloy
**ALLOY**: a more powerful version of the alloy analyzer

**alloy**: general-purpose relational specification language

**alloy analyzer**: automated bounded solver for alloy

**typical uses of the alloy analyzer**

- bounded software verification → but no software synthesis
- analyze safety properties of event traces → but no liveness properties
- find a safe full configuration → but not a safe partial conf
- find an instance satisfying a property → but no min/max instance
**What is Alloy**

**Alloy**: a more powerful version of the alloy analyzer

**Alloy**: general-purpose relational specification language

**Alloy analyzer**: automated bounded solver for alloy

**Typical uses of the alloy analyzer**

- bounded software verification
- analyze safety properties of event traces
- find a safe full configuration
- find an instance satisfying a property

→ but no software synthesis
→ but no liveness properties
→ but not a safe partial conf
→ but no min/max instance

higher-order
What is Alloy*?

Alloy*: a more powerful version of the alloy analyzer

alloy: general-purpose relational specification language
alloy analyzer: automated bounded solver for alloy

typical uses of the alloy analyzer
  • bounded software verification → but no software synthesis
  • analyze safety properties of event traces → but no liveness properties
  • find a safe full configuration → but not a safe partial conf
  • find an instance satisfying a property → but no min/max instance

higher-order

Alloy*

  • capable of automatically solving arbitrary higher-order formulas
**First-order Vs. Higher-Order:** *clique*

**first-order:** finding a graph and a *clique* in it
- every two nodes in a clique must be connected

![Graph with nodes n1, n2, n3, n4 and edges](image)

Alloy Analyzer: automatic, bounded, relational constraint solver

A solution (automatically found by Alloy):

\[ \text{clqNodes} = \{n1, n3\} \]
First-Order Vs. Higher-Order: \textit{clique}

\textbf{first-order}: finding a graph and a \textit{clique} in it

\begin{itemize}
  \item every two nodes in a clique must be connected
\end{itemize}

\begin{center}
\begin{tikzpicture}
  \node (n1) [shape=circle,draw=green] at (0,0) {n1};
  \node (n2) [shape=circle,draw=green] at (-1,-1) {n2};
  \node (n3) [shape=circle,draw=green] at (2,-1) {n3};
  \node (n4) [shape=circle,draw=green] at (1,1) {n4};

  \draw [->, thick, red] (n1) edge (n2);
  \draw [->, thick, red] (n1) edge (n3);
  \draw [->, thick, red] (n1) edge (n4);
  \draw [->, thick, red] (n4) edge (n3);
  \draw [->, thick, red] (n2) edge (n3);
  \draw [->, thick, red] (n2) edge (n4);

  \node at (-1.5, -1) {edges};

  \node at (n1) [above] {n1 \hspace{0.5cm} key: 5};
  \node at (n2) [above] {n2 \hspace{0.5cm} key: 0};
  \node at (n3) [above] {n3 \hspace{0.5cm} key: 6};
  \node at (n4) [above] {n4 \hspace{0.5cm} key: 1};
\end{tikzpicture}
\end{center}

\texttt{sig Node \{ key: one Int \}}
**First-Order Vs. Higher-Order: clique**

**first-order**: finding a graph and a **clique** in it
- every two nodes in a clique must be connected

```
sig Node { key: one Int }

run {
  some edges: Node -> Node |
  some clqNodes: set Node |
  clique[edges, clqNodes]
}
```

Alloy Analyzer: automatic, bounded, relational constraint solver

A solution (automatically found by Alloy):
```
clqNodes = {n1, n3}
```
**First-Order Vs. Higher-Order: **\textit{clique}

**first-order:** finding a graph and a \textit{clique} in it

- every two nodes in a clique must be connected

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{clique_diagram}
\end{figure}

\textbf{sig} Node \{ key: one Int \}

\textbf{run} {
  some edges: Node -> Node |
  some clqNodes: set Node |
  clique[edges, clqNodes]
}

\textbf{pred} clique[edges: Node->Node, clqNodes: set Node] {
  all disj n1, n2: clqNodes | n1->n2 in edges
}
**First-Order Vs. Higher-Order: clique**

**first-order**: finding a graph and a clique in it
- every two nodes in a clique must be connected

```
sig Node { key: one Int }

run {
  some edges: Node -> Node |
  some clqNodes: set Node |
  clique[edges, clqNodes]
}

pred clique[edges: Node->Node, clqNodes: set Node] {
  all disj n1, n2: clqNodes | n1->n2 in edges
}
```

**Alloy Analyzer**: automatic, bounded, relational constraint solver
First-Order Vs. Higher-Order: clique

**first-order**: finding a graph and a clique in it
- every two nodes in a clique must be connected

**Alloy Analyzer**: automatic, bounded, relational constraint solver

A solution (automatically found by Alloy): \(\text{clqNodes} = \{n_1, n_3\}\)
**First-Order Vs. Higher-Order: clique**

**first-order**: finding a graph and a clique in it
- every two nodes in a clique must be connected

```
sig Node { key: one Int }
run {
  some edges: Node -> Node |
  some clqNodes: set Node |
  clique[edges, clqNodes]
}

pred clique[edges: Node->Node, clqNodes: set Node] {
  all disj n1, n2: clqNodes | n1->n2 in edges
}
```

- **Alloy Analyzer**: automatic, bounded, relational constraint solver
- a **solution** (automatically found by Alloy): \(clqNodes = \{n_1, n_3\}\)
**higher-order**: finding a graph and a maximal clique in it
- there is no other clique with more nodes

Diagram:
- n1 key: 5
- n2 key: 0
- n3 key: 6
- n4 key: 1

**maxClique**
**First-Order Vs. Higher-Order:** maxClique

**higher-order:** finding a graph and a maximal clique in it

- there is no other clique with more nodes

```alloy
pred maxClique[edges: Node->Node, clqNodes: set Node] {
  clique[edges, clqNodes]
  all ns: set Node |
      not (clique[edges, ns] and #ns > #clqNodes)
}
```

expressible but not solvable in Alloy!

**definition of higher-order (as in Alloy):**

- quantification over all sets of atoms
**Higher-order**: finding a graph and a maximal clique in it

- there is no other clique with more nodes

```
pred maxClique[edges: Node->Node, clqNodes: set Node] {
    clique[edges, clqNodes]
    all ns: set Node |
        not (clique[edges, ns] and #ns > #clqNodes)
}
run {
    some edges: Node -> Node |
    some clqNodes: set Node |
        maxClique[edges, clqNodes]
}
```
**First-Order Vs. Higher-Order:** maxClique

**higher-order:** finding a graph and a maximal clique in it
- there is no other clique with more nodes

**expressible but not solvable in Alloy!**

```
sig Node { key: Int }
pred clique[edges: Node->Node, clq: set Node] {
  all disj n1, n2: clq | n1->n2 in edges
}
pred maxClique[edges: Node->Node, clq: set Node] {
  clique[edges, clq]
  all ns: set Node |
  not (clique[edges, ns] and #ns > #clq)
}
run { // find a maximal clique in a given graph
  let edges = Node -> Node |
  some clq: set Node | maxClique[edges, clq]
}
```

```
Alloy Analyzer 4.2_2015-02-22 (build date: 2015-02-22)

Executing "Run run1"

Sig this/Node scope <= 3
Sig this/Node in [[Node$0], [Node$1], [Node$2]]
Generating facts...
Simplifying the bounds...
Solver=minisatprover(jni) Bitwidth=4 MaxSeq=4 Skolemization ...
Generating CNF...
Generating the solution...

A type error has occurred: (see the stacktrace)
Analysis cannot be performed since it requires higher quantification that could not be skolemized.

Line 10, Column 7
```
First-Order Vs. **Higher-Order**: \textit{maxClique}

**higher-order**: finding a graph and a \textit{maximal clique} in it
- there is no other clique with more nodes

**expressible but not solvable** in Alloy!

- **definition** of higher-order (as in Alloy):
  - quantification over all \textit{sets} of atoms
## Solving **maxClique** Vs. Program **Synthesis**

<table>
<thead>
<tr>
<th>Program Synthesis</th>
<th>maxClique</th>
</tr>
</thead>
<tbody>
<tr>
<td>find <strong>some</strong> program AST s.t., for all possible values of its inputs its specification holds</td>
<td>find <strong>some</strong> set of nodes s.t., it is a clique and for all possible other sets of nodes not one is a larger clique</td>
</tr>
<tr>
<td><strong>some</strong> program: ASTNode</td>
<td><strong>some</strong> clq: set Node</td>
</tr>
<tr>
<td>all env: Var -&gt; Val</td>
<td>clique[clq] and</td>
</tr>
<tr>
<td>spec[program, env]</td>
<td>all ns: set Node</td>
</tr>
<tr>
<td></td>
<td>not (clique[ns] and #ns &gt; #clq)</td>
</tr>
</tbody>
</table>
Solving **maxClique** Vs. Program **Synthesis**

<table>
<thead>
<tr>
<th><strong>program synthesis</strong></th>
<th><strong>maxClique</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>find some program AST s.t., for all possible values of its inputs its specification holds</td>
<td>find some set of nodes s.t., it is a clique and for all possible other sets of nodes not one is a larger clique</td>
</tr>
<tr>
<td><strong>some</strong> program: ASTNode</td>
<td><strong>some</strong> clq: set Node</td>
</tr>
<tr>
<td><strong>all</strong> env: Var -&gt; Val</td>
<td><strong>clique</strong>[clq] <strong>and</strong></td>
</tr>
<tr>
<td>spec[program, env]</td>
<td><strong>all</strong> ns: set Node</td>
</tr>
<tr>
<td></td>
<td><strong>not</strong> (<strong>clique</strong>[ns] <strong>and</strong> #ns &gt; #clq)</td>
</tr>
</tbody>
</table>

**similarities:**
- the same **some/all** \((\exists \forall)\) pattern
- the **all** quantifier is higher-order
### Solving **maxClique** Vs. Program **Synthesis**

<table>
<thead>
<tr>
<th>Program Synthesis</th>
<th><strong>maxClique</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>find some program AST s.t., for all possible values of its inputs its specification holds</td>
<td>find some set of nodes s.t., it is a clique and for all possible other sets of nodes not one is a larger clique</td>
</tr>
<tr>
<td>some program: ASTNode</td>
<td>some clq: set Node</td>
</tr>
<tr>
<td>all env: Var -&gt; Val</td>
<td>clique[clq] and</td>
</tr>
<tr>
<td>spec[program, env]</td>
<td>all ns: set Node</td>
</tr>
<tr>
<td></td>
<td>not (clique[ns] and #ns &gt; #clq)</td>
</tr>
</tbody>
</table>

**Similarities:**
- The same *some/all* ($\exists\forall$) pattern
- The *all* quantifier is higher-order

**How do existing program synthesizers work?**
CEGIS: A Common Approach for Program Synthesis

original synthesis formulation

\[
\text{run } \{ \text{some} \ \text{prog}: \text{ASTNode} \mid \text{all} \ \text{env}: \text{Var} \rightarrow \text{Val} \mid \text{spec}[\text{prog}, \text{env}] \} \\

\]

Counter-Example Guided Inductive Synthesis [Solar-Lezama, ASPLOS'06]
original synthesis formulation

run \{ \text{some} \ prog: \text{ASTNode} \ | \ \text{all} \ env: \text{Var} -> \text{Val} \ | \ \text{spec}[\text{prog}, \text{env}] \ \}\n
Counter-Example Guided Inductive Synthesis [Solar-Lezama, ASPLOS'06]

1. search: find \text{some} program and \text{some} environment s.t. the spec holds, i.e.,
   
   run \{ \text{some} \ prog: \text{ASTNode} \ | \ \text{some} \ env: \text{Var} -> \text{Val} \ | \ \text{spec}[\text{prog}, \text{env}] \ \}\n
   to get a concrete \textit{candidate} program $\text{prog}$
original synthesis formulation

\[
\text{run } \{ \text{some } \text{prog}: \text{ASTNode} | \text{all } \text{env}: \text{Var} \rightarrow \text{Val} | \text{spec}[\text{prog}, \text{env}] \} \]

Counter-Example Guided Inductive Synthesis [Solar-Lezama, ASPLOS'06]

1. **search**: find *some* program and *some* environment s.t. the spec holds, i.e.,
   \[
   \text{run } \{ \text{some } \text{prog}: \text{ASTNode} | \text{some } \text{env}: \text{Var} \rightarrow \text{Val} | \text{spec}[\text{prog}, \text{env}] \} \]
   to get a concrete *candidate* program $\text{prog}$

2. **verification**: check if $\text{prog}$ holds for *all* possible environments:
   \[
   \text{check } \{ \text{all } \text{env}: \text{Var} \rightarrow \text{Val} | \text{spec}[\text{prog}, \text{env}] \} \]
   Done if verified; else, a concrete *counterexample* $\text{env}$ is returned as witness.
CEGIS: A Common Approach for Program Synthesis

original synthesis formulation

\[
\text{run} \{ \text{some} \ \text{prog}: \ASTNode \ | \ \text{all} \ \text{env}: \Var -> \Val \ | \ \text{spec}[\text{prog}, \text{env}] \}
\]

Counter-Example Guided Inductive Synthesis [Solar-Lezama, ASPLOS'06]

1. search: find some program and some environment s.t. the spec holds, i.e.,
   \[
   \text{run} \{ \text{some} \ \text{prog}: \ASTNode \ | \ \text{some} \ \text{env}: \Var -> \Val \ | \ \text{spec}[\text{prog}, \text{env}] \}
   \]
   to get a concrete candidate program $prog$

2. verification: check if $prog$ holds for all possible environments:
   \[
   \text{check} \{ \text{all} \ \text{env}: \Var -> \Val \ | \ \text{spec}[$prog$, \text{env}] \}
   \]
   Done if verified; else, a concrete counterexample $env$ is returned as witness.

3. induction: incrementally find a new program that additionally satisfies $env$:
   \[
   \text{run} \{ \text{some} \ \text{prog}: \ASTNode \ | \ \text{some} \ \text{env}: \Var -> \Val \ | \ \text{spec}[\text{prog}, \text{env}] \ \text{and} \ \text{spec}[\text{prog}, \text{env}] \}
   \]
   If UNSAT, return no solution; else, go to 2.
**ALLOY**

**ALLOY** key insight

CEGIS can be applied to solve **arbitrary higher-order** formulas

Wide applicability (in contrast to specialized synthesizers)

Program synthesis: SyGuS benchmarks

Security policy synthesis: Margrave

Solving graph problems: max-cut, max-clique, min-vertex-cover

Bounded verification: Turán’s theorem
Alloy*

generality

- solve arbitrary higher-order formulas
- no domain-specific knowledge needed
generality

- solve arbitrary higher-order formulas
- no domain-specific knowledge needed

implementability

- key solver features for efficient implementation:
  - partial instances
  - incremental solving
**Alloy**

**Generality**
- solve arbitrary higher-order formulas
- no domain-specific knowledge needed

**Implementability**
- key solver features for efficient implementation:
  - partial instances
  - incremental solving

**Wide applicability** (in contrast to specialized synthesizers)
- program synthesis: SyGuS benchmarks
- security policy synthesis: Margrave
- solving graph problems: max-cut, max-clique, min-vertex-cover
- bounded verification: Turán’s theorem
Generality: Nested Higher-Order Quantifiers

```haskell
fun keysum[nodes: set Node]: Int {
  sum n: nodes | n.key
}

pred maxMaxClique[edges: Node->Node, clq: set Node] {
  maxClique[edges, clq]
  all ns: set Node |
  not (maxClique[edges,clq2] and
    keysum[ns] > keysum[clq])
}

run maxMaxClique for 5
```

```
$clq
```

```
edges
```

```
n1
key: 5

n2
key: 0

n3
key: 6

n4
key: 1
```

```
Executing "Run maxMaxClique for 5"
Solver=minisat(jni) Bitwidth=5 MaxSeq=5 SkolemDepth=3 Symmetry=20
13302 vars. 831 primary vars. 47221 clauses. 66ms.
Solving...

[Some4All] started (formula, bounds)
[Some4All] candidate found (candidate)
[Some4All] verifying candidate (condition, pi) counterexample
  - [OR] solving splits (formula)
  - [OR] trying choice (formula, bounds) unsat
  - [OR] trying choice (formula, bounds) instance
    - [Some4All] started (formula, bounds)
    - [Some4All] candidate found (candidate)
    - [Some4All] verifying candidate (condition, pi) success (#cand = 1)
[Some4All] candidate found (candidate)
[Some4All] verifying candidate (condition, pi) counterexample
  - [OR] solving splits (formula)
  - [OR] trying choice (formula, bounds) unsat
  - [OR] trying choice (formula, bounds) instance
    - [Some4All] started (formula, bounds)
    - [Some4All] candidate found (candidate)
    - [Some4All] verifying candidate (condition, pi) success (#cand = 1)
[Some4All] candidate found (candidate)
[Some4All] verifying candidate (condition, pi) counterexample
  - [OR] solving splits (formula)
  - [OR] trying choice (formula, bounds) unsat
  - [OR] trying choice (formula, bounds) instance
    - [Some4All] started (formula, bounds)
    - [Some4All] candidate found (candidate)
    - [Some4All] verifying candidate (condition, pi) success (#cand = 3)
[Some4All] candidate found (candidate)
[Some4All] verifying candidate (condition, pi) counterexample
  - [OR] solving splits (formula)
  - [OR] trying choice (formula, bounds) unsat
  - [OR] trying choice (formula, bounds) instance
    - [Some4All] started (formula, bounds)
    - [Some4All] candidate found (candidate)
    - [Some4All] verifying candidate (condition, pi) success (#cand = 3)
Instance found. Predicate is consistent. 490ms.
```
Generality: Checking Higher-Order Properties

// 'edges' must be symmetric and irreflexive
pred edgeProps[edges: Node -> Node] {
    (~edges in edges) and (no edges & iden)
}

// Turan's theorem: max number of edges in a
// (k+1)-free graph with n nodes is \( \frac{(k-1)n^2}{2k} \)
check Turan {
    all edges: Node -> Node | edgeProps[edges] implies
    some mClq: set Node {
        maxClique[edges, mClq]
        let n = #Node, k = #mClq, e = (#edges).div[2] |
        e <= k.minus[1].mul[n].mul[n].div[2].div[k]
    } for 7 but 0..294 Int
CEGIS: defined only for a single idiom (the $\exists \forall$ formula pattern)
Semantics: General Idea

- CEGIS: defined only for a single idiom (the $\exists \forall$ formula pattern)
- ALLOY*: generalized to arbitrary formulas
Semantics: General Idea

- CEGIS: defined only for a single idiom (the $\exists \forall$ formula pattern)
- ALLOY*: generalized to arbitrary formulas
  1. perform standard transformation: NNF and skolemization
Semantics: General Idea

- **CEGIS**: defined only for a **single** idiom (the $\exists \forall$ formula pattern)
- **ALLOY***: generalized to **arbitrary** formulas

1. perform standard transformation: NNF and skolemization

2. decompose arbitrary formula into known idioms
   - $\text{FOL}$: first-order formula
   - $\text{OR}$: disjunction
   - $\exists \forall$: higher-order top-level $\forall$ quantifier (not skolemizable)
Semantics: General Idea

- CEGIS: defined only for a single idiom (the $\exists \forall$ formula pattern)
- Alloy*: generalized to arbitrary formulas

1. perform standard transformation: NNF and skolemization

2. decompose arbitrary formula into known idioms
   - $\text{FOL}$: first-order formula
   - $\text{OR}$: disjunction
   - $\exists \forall$: higher-order top-level $\forall$ quantifier (not skolemizable)

3. solve using the following decision procedure
   - $\text{FOL}$: solve directly with Kodkod (first-order relational solver)
   - $\text{OR}$: solve each disjunct separately
   - $\exists \forall$: apply CEGIS
some prog: Node | acyclic[prog]
all eval: Node -> (Int+Bool) | semantics[eval] implies spec[prog, eval]

\[\begin{align*}
\forall (\text{conj}: \quad & \text{prog in Node and acyclic[prog],} \\
\exists (\text{eQuant}: \quad & \text{some eval ...},) \\
\exists (\text{aQuant}: \quad & \text{all eval ...})
\end{align*}\]
ALLOY* Implementation Caveats

some prog: Node | acyclic[prog] all eval: Node -> (Int+Bool) | semantics[eval] implies spec[prog, eval]

→ \( \exists (conj: \ prog \ in \ Node \ and \ acyclic[\ prog], eQuant: \ some \ eval \ ...\), aQuant: all eval ...) \)

1. candidate search

solve \( conj \land eQuant \)

→ candidate instance \( cand: \ values \ of \ all \ relations \ except \ eQuant.var \)
**ALLOY** Implementation Caveats

\[\text{some prog: Node |}
\text{acyclic[prog]}
\]
\[\text{all eval: Node -> (Int+Bool) |}
\text{semantics[eval] implies spec[prog, eval]}
\]

\[\forall(\text{conj: $prog$ in Node and acyclic[$prog$],}
\text{eQuant: some eval ...},
\text{aQuant: all eval ...})\]

1. candidate search

- \(\text{solve } \text{conj} \land \text{eQuant}\)
- \(\text{candidate instance } \text{$cand$}: \text{values of all relations except } \text{eQuant}.\text{var}\)

2. verification

- \(\text{solve } \neg \text{aQuant} \text{ against the } \text{$cand$ partial instance}\)
- \(\text{counterexample } \text{$cex$}: \text{value of the } \text{eQuant}.\text{var} \text{ relation}\)
ALLOY* Implementation Caveats

some prog: Node | acyclic[prog]
   all eval: Node -> (Int+Bool) | semantics[eval] implies spec[prog, eval] → ∃(conj: prog in Node and acyclic[prog],
   eQuant: some eval ..., aQuant: all eval ...)

1. candidate search
   ● solve conj ∧ eQuant
   → candidate instance $cand$: values of all relations except $eQuant.var$

2. verification
   ● solve ¬aQuant against the $cand$ partial instance
   → counterexample $cex$: value of the $eQuant.var$ relation

partial instance
- partial solution known upfront
- enforced using bounds
**ALLOY* Implementation Caveats**

```plaintext
some prog: Node | acyclic[prog]  
all eval: Node -> (Int+Bool) | semantics[eval] implies spec[ prog, eval]  
∀(conj: $prog in Node and acyclic[$prog], 
   eQuant: some eval ..., 
   aQuant: all eval ...)  
```

1. candidate search

- **solve** `conj ∧ eQuant`
- `→ candidate instance $cand$: values of all relations except `eQuant.var`

2. verification

- **solve** `¬aQuant` against the `$cand` partial instance
- `→ counterexample $cex$: value of the `eQuant.var` relation`

3. induction

- **use incremental solving** to add
  - replace `eQuant.var` with `$cex` in `eQuant.body`
  to previous search condition

**partial instance**
- partial solution known upfront
- enforced using `bounds`
some prog: Node | acyclic[prog]
all eval: Node -> (Int+Bool) | semantics[eval] implies spec[prog, eval]

\[\exists (conj: \text{ prog in Node and acyclic[prog]},
\text{ eQuant: some eval ...},
\text{ aQuant: all eval ...})\]

1. candidate search
   - solve \( conj \land eQuant \)
   \[\text{ candidate instance } \text{ cand: values of all relations except } eQuant.var\]

2. verification
   - solve \( \neg aQuant \) against the \( \text{ cand } \) partial instance
   \[\text{ counterexample } \text{ cex: value of the } eQuant.var \text{ relation}\]

3. induction
   - use incremental solving to add
     \[\text{ replace } eQuant.var \text{ with } \text{ cex in } eQuant.body\]
     to previous search condition

partial instance
- partial solution known upfront
- enforced using bounds

incremental solving
- continue from prev solver instance
- the solver reuses learned clauses
**ALLOY* Implementation** Caveats

some prog: Node |
ayclic[prog]

all eval: Node -> (Int+Bool) |
semantics[eval] implies spec[prog, eval]

→ \( \exists (conj: \ prog \ in \ Node \ and \ acyclic[\ prog], \ eQuant: \ some \ eval \ ...), \ aQuant: \ all \ eval \ ...) \)

---

1. candidate search

- solve \( conj \land eQuant \)
- \( \rightarrow \) **candidate instance** $\text{cand}$: values of all relations except \( eQuant \).var

---

2. verification

- solve \( \neg aQuant \) against the $\text{cand}$ **partial instance**
- \( \rightarrow \) **counterexample** $\text{cex}$: value of the \( eQuant \).var relation

---

3. induction

- use **incremental solving** to add
  - replace \( eQuant \).var with $\text{cex}$ in \( eQuant \).body
  
  to previous search condition

---

? **what if the increment formula is not first-order**
  - optimization 1: use its weaker “first-order version”
2. domain constraints

“for all possible eval, if the semantics hold then the spec must hold”

vs.

“for all eval that satisfy the semantics, the spec must hold”
2. domain constraints

“for all possible eval, if the semantics hold then the spec must hold”

vs.

“for all eval that satisfy the semantics, the spec must hold”

● logically equivalent, but, when “for” implemented as CEGIS:
2. domain constraints

"for all possible eval, if the semantics hold then the spec must hold" vs. "for all eval that satisfy the semantics, the spec must hold"

- logically equivalent, but, when "for" implemented as CEGIS:

```alloy
define synth(prog: Node) {
define all eval: Node -> (Int+Bool) | semantics[eval] implies spec[prog, eval]
}
define some prog: Node |
define some eval: Node -> (Int+Bool) | semantics[eval] implies spec[prog, eval]
da valid candidate doesn't have to satisfy the semantics predicate!
```
2. domain constraints

"for all possible eval, if the semantics hold then the spec must hold" vs. "for all eval that satisfy the semantics, the spec must hold"

- logically equivalent, but, when "for" implemented as CEGIS:

```
pred synth(prog: Node) {  
  all eval: Node -> (Int+Bool) |  
  semantics[eval] implies spec[prog, eval]  
}  
→ candidate search  
some prog: Node |  
some eval: Node -> (Int+Bool) |  
semantics[eval] implies spec[prog, eval]  
→ a valid candidate doesn't have to satisfy the semantics predicate!
```

```
pred synth(prog: Node) {  
  all eval: Node -> (Int+Bool) when semantics[eval]  
  spec[prog, eval]  
}  
→ candidate search  
some prog: Node |  
some eval: Node -> (Int+Bool) when semantics[eval] |  
spec[prog, eval]  
→ a valid candidate must satisfy the semantics predicate!
```
evaluation goals
evaluation goals

1. scalability on classical higher-order graph problems
   \[\text{? does ALLOY* scale beyond “toy-sized” graphs}\]

2. applicability to program synthesis
   \[\text{expressiveness: how many SyGuS benchmarks can be written in ALLOY*}\]
   \[\text{power: how many SyGuS benchmarks can be solved with ALLOY*}\]

3. benefits of the two optimizations
   \[\text{do ALLOY* optimizations improve overall solving times}\]
evaluation goals

1. scalability on classical higher-order graph problems
   - does ALLOY* scale beyond “toy-sized” graphs

2. applicability to program synthesis
   - expressiveness: how many SyGuS benchmarks can be written in ALLOY*
   - power: how many SyGuS benchmarks can be solved with ALLOY*
   - scalability: how does ALLOY* compare to other synthesizers
**ALLOY* Evaluation**

**evaluation goals**

1. scalability on classical higher-order graph problems
   - does ALLOY* scale beyond “toy-sized” graphs

2. applicability to program synthesis
   - expressiveness: how many SyGuS benchmarks can be written in ALLOY*
   - power: how many SyGuS benchmarks can be solved with ALLOY*
   - scalability: how does ALLOY* compare to other synthesizers

3. benefits of the two optimizations
   - do ALLOY* optimizations improve overall solving times
Evaluation: **Graph Algorithms**

![Graph Algorithms Evaluation](image)

- **max clique**
- **max cut**
- **max indep. set**
- **min vertex cover**

Solving Time (s) vs. # Nodes graph showing the performance of different graph algorithms as the number of nodes increases.
Evaluation: Program Synthesis

effectiveness
- we extended Alloy to support bit vectors
- we encoded 123/173 benchmarks, i.e., all except “ICFP problems”
  - reason for skipping ICFP: 64-bit bit vectors (not supported by Kodkod)
  - (aside) not one of them was solved by any of the competition solvers
expressiveness

- we extended Alloy to support bit vectors
- we encoded 123/173 benchmarks, i.e., all except “ICFP problems”
  - reason for skipping ICFP: 64-bit bit vectors (not supported by Kodkod)
  - (aside) not one of them was solved by any of the competition solvers

power

- Alloy* was able to solve all different categories of benchmarks
  - integer benchmarks, bit vector benchmarks, let constructs, synthesizing multiple functions at once, multiple applications of the synthesized function
Evaluation: Program Synthesis

expressiveness
- we extended Alloy to support bit vectors
- we encoded 123/173 benchmarks, i.e., all except “ICFP problems”
  - reason for skipping ICFP: 64-bit bit vectors (not supported by Kodkod)
  - (aside) not one of them was solved by any of the competition solvers

power
- Alloy* was able to solve all different categories of benchmarks
  - integer benchmarks, bit vector benchmarks, let constructs, synthesizing multiple functions at once, multiple applications of the synthesized function

scalability
- many of the 123 benchmarks are either too easy or too difficult
  → not suitable for scalability comparison
- we primarily used the integer benchmarks
- we also picked a few bit vector benchmarks that were too hard for all solvers
Evaluation: Program Synthesis

scalability comparison (integer benchmarks)

Solving Time (s)

- Alloy*
- Enumerative
- Stochastic
- Symbolic
- Sketch

benchmarks:
- parity-AIG-d1: full parity circuit using AND and NOT gates
- parity-NAND-d1: full parity circuit using AND always followed by NOT

custom tweaks in Alloy* synthesis models:
- create and use a single type of gate
- impose partial ordering between gates

parity-AIG-d1

```plaintext
sig AIG extends BoolNode {
  left, right: one BoolNode
  invLhs, invRhs, invOut: one Bool
}
pred aig_semantics[eval: Node->(Int+Bool)] {
  all n: AIG | eval[n] = ((eval[n.left] ^ n.invLhs) && (eval[n.right] ^ n.invRhs)) ^ n.invOut
}
run synth for 0 but -1..0 Int, exactly 15 AIG
```

parity-NAND-d1

```plaintext
sig NAND extends BoolNode {
  left, right: one BoolNode
}
pred nand_semantics[eval: Node->(Int+Bool)] {
  all n: NAND | eval[n] = !(eval[n.left] && eval[n.right])
}
run synth for 0 but -1..0 Int, exactly 23 NAND
```

solving time w/ partial ordering: 20s
solving time w/o partial ordering: 80s
Evaluation: Program Synthesis

**scalability comparison** (select bit vector benchmarks)

- benchmarks
  - parity-AIG-d1: full parity circuit using AND and NOT gates
  - parity-NAND-d1: full parity circuit using AND always followed by NOT
Evaluation: Program Synthesis

**scalability comparison** (select bit vector benchmarks)

- benchmarks
  - parity-AIG-d1: full parity circuit using AND and NOT gates
  - parity-NAND-d1: full parity circuit using AND always followed by NOT

- all solvers (including ALLOY*) time out on both (limit: 1000s)
scalability comparison (select bit vector benchmarks)

- benchmarks
  - parity-AIG-d1: full parity circuit using AND and NOT gates
  - parity-NAND-d1: full parity circuit using AND always followed by NOT

- all solvers (including ALLOY*) time out on both (limit: 1000s)

- custom tweaks in ALLOY* synthesis models:
  - create and use a single type of gate
  - impose partial ordering between gates

solving time w/ partial ordering: 20s
solving time w/o partial ordering: ∞
Evaluation: Program Synthesis

**scalability comparison** (select bit vector benchmarks)

- benchmarks
  - **parity-AIG-d1**: full parity circuit using AND and NOT gates
  - **parity-NAND-d1**: full parity circuit using AND always followed by NOT

- all solvers (including ALLOY*) time out on both (limit: 1000s)

- custom tweaks in ALLOY* synthesis models:
  - create and use a single type of gate
  - impose partial ordering between gates

<table>
<thead>
<tr>
<th>parity-AIG-d1</th>
<th>parity-NAND-d1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sig AIG extends BoolNode {</code></td>
<td><code>sig NAND extends BoolNode {</code></td>
</tr>
<tr>
<td>left, right: <code>one</code> BoolNode</td>
<td>left, right: <code>one</code> BoolNode</td>
</tr>
<tr>
<td>invLhs, invRhs, invOut: <code>one</code>Bool</td>
<td>}</td>
</tr>
<tr>
<td><code>pred aig_semantics[eval: Node-&gt;(Int+Bool)] {</code></td>
<td><code>pred nand_semantics[eval: Node-&gt;(Int+Bool)] {</code></td>
</tr>
<tr>
<td>all n: AIG</td>
<td>eval[n] = ((eval[n.left] ^ n.invLhs) &amp;&amp; (eval[n.right] ^ n.invRhs)) ^ n.invOut}</td>
</tr>
<tr>
<td><code>run synth for 0 but -1..0 Int, exactly 15 AIG</code></td>
<td><code>run synth for 0 but -1..0 Int, exactly 23 NAND</code></td>
</tr>
</tbody>
</table>
Evaluation: Program **Synthesis**

**scalability comparison** (select bit vector benchmarks)

- benchmarks
  - parity-AIG-d1: full parity circuit using AND and NOT gates
  - parity-NAND-d1: full parity circuit using AND always followed by NOT

- all solvers (including ALLOY*) time out on both (limit: 1000s)

- custom tweaks in ALLOY* synthesis models:
  - create and use a single type of gate
  - impose partial ordering between gates

### parity-AIG-d1

```plaintext
sig AIG extends BoolNode {
    left, right: one BoolNode
    invLhs, invRhs, invOut: one Bool
}
pred aig_semantics[eval: Node->(Int+Bool)] {
    all n: AIG |
    eval[n] = ((eval[n.left] ^ n.invLhs) &&
              (eval[n.right] ^ n.invRhs)) ^ n.invOut}
run synth for 0 but -1..0 Int, exactly 15 AIG
```

- solving time w/ partial ordering: 20s
- solving time w/o partial ordering: 80s

### parity-NAND-d1

```plaintext
sig NAND extends BoolNode {
    left, right: one BoolNode
}
pred nand_semantics[eval: Node->(Int+Bool)] {
    all n: NAND |
    eval[n] = !(eval[n.left] &&
               eval[n.right])
}
run synth for 0 but -1..0 Int, exactly 23 NAND
```

- solving time w/ partial ordering: 30s
- solving time w/o partial ordering: ∞
## Evaluation: Benefits of \textsc{Alloy}*: Optimizations

<table>
<thead>
<tr>
<th>base</th>
<th>w/ optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>max2</strong></td>
<td>0.4s</td>
</tr>
<tr>
<td><strong>max3</strong></td>
<td>7.6s</td>
</tr>
<tr>
<td><strong>max4</strong></td>
<td>t/o</td>
</tr>
<tr>
<td><strong>max5</strong></td>
<td>t/o</td>
</tr>
<tr>
<td><strong>max6</strong></td>
<td>t/o</td>
</tr>
<tr>
<td><strong>max7</strong></td>
<td>t/o</td>
</tr>
<tr>
<td><strong>max8</strong></td>
<td>t/o</td>
</tr>
<tr>
<td><strong>array-search2</strong></td>
<td>140.0s</td>
</tr>
<tr>
<td><strong>array-search3</strong></td>
<td>t/o</td>
</tr>
<tr>
<td><strong>array-search4</strong></td>
<td>t/o</td>
</tr>
<tr>
<td><strong>array-search5</strong></td>
<td>t/o</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>base</th>
<th>w/ optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>turan5</strong></td>
<td>3.5s</td>
</tr>
<tr>
<td><strong>turan6</strong></td>
<td>12.8s</td>
</tr>
<tr>
<td><strong>turan7</strong></td>
<td>235.0s</td>
</tr>
<tr>
<td><strong>turan8</strong></td>
<td>t/o</td>
</tr>
<tr>
<td><strong>turan9</strong></td>
<td>t/o</td>
</tr>
<tr>
<td><strong>turan10</strong></td>
<td>t/o</td>
</tr>
</tbody>
</table>
**ALLOY**

**Conclusion**

**ALLOY** is

- general purpose constraint solver
- capable of efficiently solving arbitrary higher-order formulas
- sound & complete within given bounds

why is this important?

accessible to wider audience, encourages new applications

potential impact

– abundance of tools that build on Alloy/Kodkod, for testing, program analysis, security, bounded verification, executable specifications, ...

Thank You!

http://alloy.mit.edu/alloy/hola
**Conclusion**

**ALLOY** is
- general purpose constraint solver
- capable of efficiently solving arbitrary higher-order formulas
- sound & complete within given bounds

higher-order and alloy historically
- bit-blasting higher-order quantifiers: attempted, deemed intractable
- previously many ad hoc mods to alloy
  - aluminum, razor, staged execution, ...

Why is this important?
- accessible to wider audience, encourages new applications
- potential impact
  - abundance of tools that build on Alloy/Kodkod, for testing, program analysis, security, bounded verification, executable specifications, ...
**Conclusion**

**Alloy**

- general purpose constraint solver
- capable of efficiently solving arbitrary higher-order formulas
- sound & complete within given bounds

**higher-order and alloy historically**

- bit-blasting higher-order quantifiers: attempted, deemed intractable
- previously many ad hoc mods to alloy
  - aluminum, razor, staged execution, ...

**why is this important?**

- accessible to wider audience, encourages new applications
- potential impact
  - abundance of tools that build on Alloy/Kodkod, for testing, program analysis, security, bounded verification, executable specifications, ...
**Conclusion**

**ALLOY** is
- general purpose constraint solver
- capable of efficiently solving arbitrary higher-order formulas
- sound & complete within given bounds

higher-order and alloy historically
- bit-blasting higher-order quantifiers: attempted, deemed intractable
- previously many ad hoc mods to alloy
  - aluminum, razor, staged execution, ...

why is this important?
- accessible to wider audience, encourages new applications
- potential impact
  - abundance of tools that build on Alloy/Kodkod, for testing, program analysis, security, bounded verification, executable specifications, ...

Thank You!

http://alloy.mit.edu/alloy/hola
first-order: finding a clique in a graph
**First-Order Vs. Higher-Order: clique**

**first-order:** finding a **clique** in a graph

```plaintext
    all disj n1, n2: clq | n1->n2 in edges // every two nodes in ‘clq’ are connected
}
```

---

**Alloy encoding**

- **Nodes**: \{n1, n2, n3, n4\}
  - **Key**: \{(n1 -> 5), (n2 -> 0), (n3 -> 6), (n4 -> 1)\}

- **Edges**: \{(n1 -> n2), (n1 -> n3), (n1 -> n4), (n2 -> n3), (n2 -> n4), (n3 -> n1), (n4 -> n1), (n3 -> n2), (n4 -> n2)\}

- **Fixed relations**
  - clq = \{\}
  - \{n1, n2, n3, n4\}

---

**Lower bound**

- **Upper bound**

---

A solution (automatically found by Alloy):

- **clq** = \{n1, n3\}
first-order: finding a clique in a graph

```
pred clique[edges: Node->Node, clq: set Node] {
    all disj n1, n2: clq | n1->n2 in edges // every two nodes in ‘clq’ are connected
}
run {  // find a clique in a given graph
    let edges = n1->n2 + n1->n3 + ... |
    some clq: set Node | clique[edges, clq]
}
```
**First-Order Vs. Higher-Order: clique**

**first-order**: finding a *clique* in a graph

```alloy
definition clique[edges: Node->Node, clq: set Node] {
    all disj n1, n2: clq | n1->n2 in edges // every two nodes in 'clq' are connected
}
run { // find a clique in a given graph
    let edges = n1->n2 + n1->n3 + ... |
    some clq: set Node | clique[edges, clq]
}
```

**Alloy encoding**:

```
N1: {n_1} | N2: {n_2} | N3: {n_3} | N4: {n_4}
```

**nodes**

- n1 key: 5
- n2 key: 0
- n3 key: 6
- n4 key: 1

**edges**
First-Order Vs. Higher-Order: clique

**First-order**: finding a **clique** in a graph

```
pred clique[edges: Node->Node, clq: set Node] {
    all disj n1, n2: clq | n1->n2 in edges // every two nodes in 'clq' are connected
}
run { // find a clique in a given graph
    let edges = n1->n2 + n1->n3 + ... |
    some clq: set Node | clique[edges, clq]
}
```

**Alloy encoding**:

- **Atoms**:
  - N1: \{n_1\}  |  N2: \{n_2\}  |  N3: \{n_3\}  |  N4: \{n_4\}

- **Fixed relations**:
  - Node: \{n_1, n_2, n_3, n_4\}
  - key: \{(n_1 \rightarrow 5), (n_2 \rightarrow 0), (n_3 \rightarrow 6), (n_4 \rightarrow 1)\}
  - edges: \{(n_1 \rightarrow n_2), (n_1 \rightarrow n_3), (n_1 \rightarrow n_4), (n_2 \rightarrow n_3), (n_2 \rightarrow n_4),
            (n_2 \rightarrow n_1), (n_3 \rightarrow n_1), (n_4 \rightarrow n_1), (n_3 \rightarrow n_2), (n_4 \rightarrow n_2)\}

Diagram:

- n1: key: 5
- n2: key: 0
- n3: key: 6
- n4: key: 1

**Edges**:
- n1 -> n2
- n1 -> n3
- n1 -> n4
- n2 -> n3
- n2 -> n4
- n2 -> n1
- n3 -> n1
- n4 -> n1
- n3 -> n2
- n4 -> n2
# First-Order Vs. Higher-Order: clique

**first-order**: finding a *clique* in a graph

```plaintext
pred clique[edges: Node->Node, clq: set Node] {
    all disj n1, n2: clq | n1->n2 in edges // every two nodes in 'clq' are connected
}
run { // find a clique in a given graph
    let edges = n1->n2 + n1->n3 + ... |
    some clq: set Node | clique[edges, clq]
}
```

**Alloy encoding:**

- **Atoms**
  - N1: \{n1\} | N2: \{n2\} | N3: \{n3\} | N4: \{n4\}

- **Fixed relations**
  - **Node:** \{n1,n2,n3,n4\}
  - **key:** \{(n1 → 5), (n2 → 0), (n3 → 6), (n4 → 1)\}
  - **edges:** \{(n1 → n2), (n1 → n3), (n1 → n4), (n2 → n3), (n2 → n4), (n3 → n1), (n4 → n1), (n3 → n2), (n4 → n2)\}

- **Relations to be solved**
  - **clq:** {} | \{n1,n2,n3,n4\}

**Lower bound**  **Upper bound**  → set of nodes: efficiently translated to SAT
(one bit for each node)

---

**Diagram:**

```
edges
N1: \{n1\}  N2: \{n2\}  N3: \{n3\}  N4: \{n4\}

Node: \{n1,n2,n3,n4\}
key: \{(n1 \text{→} 5), (n2 \text{→} 0), (n3 \text{→} 6), (n4 \text{→} 1)\}
edges: \{(n1 \text{→} n2), (n1 \text{→} n3), (n1 \text{→} n4), (n2 \text{→} n3), (n2 \text{→} n4), (n3 \text{→} n1), (n4 \text{→} n1), (n3 \text{→} n2), (n4 \text{→} n2)\}
clq: {} | \{n1,n2,n3,n4\}
```

**Set of nodes:** efficiently translated to SAT
(one bit for each node)
First-Order Vs. Higher-Order: clique

**first-order**: finding a *clique* in a graph

```alloy
def pred clique[edges: Node->Node, clq: set Node] {
    all disj n1, n2: clq | n1->n2 in edges // every two nodes in 'clq' are connected
}
run { // find a clique in a given graph
    let edges = n1->n2 + n1->n3 + ... |
    some clq: set Node | clique[edges, clq]
}
```

**Alloy encoding**:

- **Atoms**:
  - N1: \{n1\}
  - N2: \{n2\}
  - N3: \{n3\}
  - N4: \{n4\}

- **Fixed relations**:
  - Node: \{n1, n2, n3, n4\}
  - Key: \{(n1 -> 5), (n2 -> 0), (n3 -> 6), (n4 -> 1)\}
  - Edges: \{(n1 -> n2), (n1 -> n3), (n1 -> n4), (n2 -> n3), (n2 -> n4),
             (n3 -> n1), (n4 -> n1), (n3 -> n2), (n4 -> n2)\}

- **Relations to be solved**:
  - Clq: \{\}, \{n1, n2, n3, n4\}

  **Set of nodes**: efficiently translated to SAT (one bit for each node)

- **Solution** (automatically found by Alloy): \(\text{clq} = \{n_1, n_3\}\)
**First-Order Vs. **Higher-**Order: maxClique**

**higher-order**: finding a maximal clique in a graph
**higher-order**: finding a maximal clique in a graph

```alloy
def maxClique[edges: Node->Node, clq: set Node] {
    clique[edges, clq]
    all ns: set Node |
        not (clique[edges, ns] and #ns > #clq)
}
```
**higher-order**: finding a **maximal clique** in a graph

```alloy
def pred maxClique[edges: Node->Node, clq: set Node] {
    clique[edges, clq]
    all ns: set Node |
        not (clique[edges, ns] and #ns > #clq)
}
run { // find a maximal clique in a given graph
    let edges = n1->n2 + n1->n3 + ... |
    some clq: set Node | maxClique[edges, clq]
}
```

![Graph diagram](image)
First-Order Vs. **Higher-Order**: maxClique

**higher-order**: finding a maximal clique in a graph

```alloy
def pred maxClique[edges: Node->Node, clq: set Node] {
    clique[edges, clq] =
    all ns: set Node |
        not (clique[edges, ns] and #ns > #clq)
}
run { // find a maximal clique in a given graph
    let edges = n1->n2 + n1->n3 + ... |
    some clq: set Node | maxClique[edges, clq]
}
```

expressible but not solvable in Alloy!

[Image of Alloy Analyzer output showing type error]

**definition of higher-order (as in Alloy):**
– quantification over all sets of atoms

**maxClique**: check all possible sets of nodes and ensure not one is a clique larger than clq

**number of bits required for direct encoding to SAT**: $2^{#Node}$
**higher-order**: finding a maximal clique in a graph

```alloy
def pred maxClique[edges: Node->Node, clq: set Node] {
  clique[edges, clq]
  all ns: set Node |
    not (clique[edges, ns] and #ns > #clq)
}
run { // find a maximal clique in a given graph
  let edges = n1->n2 + n1->n3 + ... |
  some clq: set Node | maxClique[edges, clq]
}
```

- **definition** of higher-order (as in Alloy):
  - quantification over all sets of atoms

![Graph example](image-url)
**higher-order**: finding a maximal clique in a graph

```alloy
def pred maxClique[edges: Node->Node, clq: set Node] {
  clique[edges, clq]
  all ns: set Node |
    not (clique[edges, ns] and #ns > #clq)
}
run {
// find a maximal clique in a given graph
  let edges = n1->n2 + n1->n3 + ... | 
  some clq: set Node | maxClique[edges, clq]
}
```

- **definition** of higher-order (as in Alloy):
  - quantification over all sets of atoms
- **maxClique**: check all possible sets of nodes and ensure not one is a clique larger than `clq`
higher-order: finding a maximal clique in a graph

```alloy
def pred maxClique[edges: Node->Node, clq: set Node] {
    clique[edges, clq]
    all ns: set Node |
        not (clique[edges, ns] and #ns > #clq)
}
run { // find a maximal clique in a given graph
    let edges = n1->n2 + n1->n3 + ... |
    some clq: set Node | maxClique[edges, clq]
}
```

- **definition** of higher-order (as in Alloy):
  - quantification over all sets of atoms
- **maxClique**: check all possible sets of nodes and ensure not one is a clique larger than `clq`
- Number of bits required for direct encoding to SAT: $2^\#\text{Node}$
Solving \textbf{maxClique}: \textit{Idea}

\begin{verbatim}
run {
    some clq: set Node |
        clique[edges, clq] and
    all ns: set Node |
        not (clique[edges, ns] and #ns > #clq)
}
\end{verbatim}

\textit{intuitive iterative algorithm}
Solving maxClique: Idea

\[
\text{run } \{ \\
\quad \text{some } clq: \text{ set } \text{ Node } | \\
\qquad \text{clique[edges, clq] and} \\
\qquad \text{all } ns: \text{ set } \text{ Node } | \\
\qquad\quad \text{not (clique[edges, ns] and } \#ns > \#clq) \\
\}\n\]

intuitive iterative algorithm

1. find some clique $clq$
Solving maxClique: Idea

run {
    some clq: set Node |
    clique[edges, clq] and
    all ns: set Node |
    not (clique[edges, ns] and #ns > #clq)
}

intuitive iterative algorithm

1. find some clique $clq$

2. check if $clq$ is maximal
   ⇔ find some clique $ns > clq$ from step 1
   – if not found: return $clq$
Solving \texttt{maxClique}: \textbf{Idea}

\begin{verbatim}
run {
  some clq: set Node |
  clique[edges, clq] and
  all ns: set Node |
    not (clique[edges, ns] and #ns > #clq)
}
\end{verbatim}

\textbf{intuitive iterative algorithm}

1. \textbf{find} some clique $clq$

2. \textbf{check} if $clq$ is maximal
   \hspace{1em} $\iff$ find some clique $ns > clq$ from step 1
   \hspace{1em} – if not found: return $clq$
Solving maxClique: Idea

run {
    some clq: set Node |
    clique[edges, clq] and
    all ns: set Node |
    not (clique[edges, ns] and #ns > #clq)
}

intuitive iterative algorithm

1. find some clique $clq$

2. check if $clq$ is maximal
   $\iff$ find some clique $ns > clq$ from step 1
   – if not found: return $clq$

3. assert that every new $clq$ must be $\geq$ than $ns$ from step 2;
   goto step 1
Solving \texttt{maxClique}: \textbf{Idea}

```plaintext
code
run {
    some clq: set Node |
    clique[edges, clq] and
    all ns: set Node |
        not (clique[edges, ns] and #ns > #clq)
}
```

\textbf{intuitive iterative algorithm}

1. \textbf{find} some clique $\textit{clq}$

2. \textbf{check} if $\textit{clq}$ is maximal
   \iff find some clique $\textit{ns} > \textit{clq}$ from step 1
   \quad – if not found: return $\textit{clq}$

3. \textbf{assert} that every new $\textit{clq}$ must be $\geq$ than $\textit{ns}$ from step 2;
   goto step 1
Solving maxClique: Idea

```plaintext
run {
    some clq: set Node |
    clique[edges, clq] and
    all ns: set Node |
    not (clique[edges, ns] and #ns > #clq)
}
```

intuitive iterative algorithm

1. **find** some clique $clq$

2. **check** if $clq$ is maximal
   ⇒ find some clique $ns > clq$ from step 1
      – if not found: return $clq$

3. **assert** that every new $clq$ must be $\geq$ than $ns$ from step 2;
   goto step 1
Solving maxClique: Idea

intuitive iterative algorithm

1. find some clique $clq$

2. check if $clq$ is maximal
   ⇔ find some clique $ns > clq$ from step 1
      – if not found: return $clq$

3. assert that every new $clq$ must be ≥ than $ns$ from step 2;
   goto step 1
Solving maxClique: Idea

run {
  some clq: set Node |
  clique[edges, clq] and
  all ns: set Node |
    not (clique[edges, ns] and #ns > #clq)
}

intuitive iterative algorithm

1. find some clique $clq$

2. check if $clq$ is maximal
   ⇔ find some clique $ns > $clq from step 1
      – if not found: return $clq$

3. assert that every new $clq$ must be ≥ than $ns$ from step 2;
   goto step 1

UNSAT → return $clq$
Solving maxClique: Idea

run {
  some clq: set Node |
  clique[edges, clq] and
  all ns: set Node |
  not (clique[edges, ns] and #ns > #clq)
}

intuitive iterative algorithm
Solving maxClique: Idea

\[
\text{run } \{ \\
\quad \text{some } \text{clq: set Node } | \\
\quad \text{clique[edges, clq] and} \\
\quad \text{all ns: set Node } | \\
\quad \quad \text{not (clique[edges, ns] and } #\text{ns > #clq}) \\
\} 
\]

intuitive iterative algorithm

1. find some clique $\text{clq}$
Solving **maxClique**:** Idea**

run {
    some clq: set Node |
    clique[edges, clq] and
    all ns: set Node |
    not (clique[edges, ns] and #ns > #clq)
}

**intuitive iterative algorithm**

1. **find** some clique $\text{clq}$

2. **check** if $\text{clq}$ is maximal
   $\iff$ **find** some clique $\text{ns} > \text{clq}$ from step 1
   – if not found: return $\text{clq}$
Solving maxClique: **Idea**

run {
    some clq: set Node |
        clique[edges, clq] and
    all ns: set Node |
        not (clique[edges, ns] and #ns > #clq)
}

**intuitive iterative algorithm**

1. **find** some clique $clq$

2. **check** if $clq$ is maximal
   $\iff$ find some clique $ns > clq$ from step 1
   – if not found: return $clq$

3. **assert** that every new $clq$ must be $\geq$ than $ns$ from step 2;
   goto step 1
Solving maxClique: Idea

run {
    some clq: set Node |
        clique[edges, clq] and
    all ns: set Node |
        not (clique[edges, ns] and #ns > #clq)
}

find candidate clique

intuitive iterative algorithm

1. find some clique $clq$

2. check if $clq$ is maximal
   \[ \Leftrightarrow \text{find some clique } ns > clq \text{ from step 1} \]
   – if not found: return $clq$

3. assert that every new $clq$ must be \( \geq \) than $ns$ from step 2;
   goto step 1
**Solving maxClique: Idea**

```
run {
    some clq: set Node |
    clique[edges, clq] and
    all ns: set Node |
        not (clique[edges, ns] and #ns > #clq)
}
```

**intuitive iterative algorithm**

1. **find** some clique $clq$

2. **check** if $clq$ is maximal
   \[\iff\]
   - **find** some clique $ns > clq$ from step 1
     \[-\text{if not found: return } clq\]

3. **assert** that every new $clq$ must be $\geq$ than $ns$ from step 2;
   goto step 1
Solving maxClique: Idea

```plaintext
run {
    some clq: set Node |
    clique[edges, clq] and
    all ns: set Node |
    not (clique[edges, ns] and #ns > #clq)
}
```

Intuitive iterative algorithm

1. **find** some clique $clq$

2. **check** if $clq$ is maximal
   $\iff$ find some clique $ns > clq$ from step 1
   – if not found: return $clq$

3. **assert** that every new $clq$ must be $\geq$ than $ns$ from step 2;
   goto step 1
Solving maxClique: Idea

run {
    some clq: set Node |
    clique[edges, clq] and
    all ns: set Node |
    not (clique[edges, ns] and #ns > #clq)
}

intuitive iterative algorithm

1. find some clique $clq$

2. check if $clq$ is maximal
   ⇔ find some clique $ns > clq$ from step 1
      – if not found: return $clq$

3. assert that every new $clq$ must be ≥ than $ns$ from step 2;
   goto step 1
original synthesis formulation

\[
\text{run } \{ \text{some prog: ASTNode } | \text{all env: Var } \rightarrow \text{Val } | \text{spec[prog, env]} \} \]

Counter-Example Guided Inductive Synthesis [Solar-Lezama, ASPLOS’06]
original synthesis formulation

\[
\text{run } \{ \text{some } \text{prog}: \text{ASTNode} \mid \text{all } \text{env}: \text{Var} \rightarrow \text{Val} \mid \text{spec}[\text{prog}, \text{env}] \} \\
\]

**Counter-Example Guided Inductive Synthesis** [Solar-Lezama, ASPLOS'06]

1. search: find some program and some environment s.t. the spec holds, i.e.,
   \[
   \text{run } \{ \text{some } \text{prog}: \text{ASTNode} \mid \text{some } \text{env}: \text{Var} \rightarrow \text{Val} \mid \text{spec}[\text{prog}, \text{env}] \} \\
   \]
   to get a concrete candidate program $\text{prog}$
CEGIS: A Common Approach for Program Synthesis

original synthesis formulation

\[
\text{run } \{ \text{some } \text{prog: ASTNode} \mid \text{all } \text{env: Var} \rightarrow \text{Val} \mid \text{spec[prog, env]} \}\]

Counter-Example Guided Inductive Synthesis \cite{Solar-Lezama2006}

1. search: find some program and some environment s.t. the spec holds, i.e.,
   \[
   \text{run } \{ \text{some } \text{prog: ASTNode} \mid \text{some } \text{env: Var} \rightarrow \text{Val} \mid \text{spec[prog, env]} \}\]
   to get a concrete candidate program \$\text{prog}\$

2. verification: check if \$\text{prog}\$ holds for all possible environments:
   \[
   \text{check } \{ \text{all } \text{env: Var} \rightarrow \text{Val} \mid \text{spec[\$\text{prog}, \text{env}\]} \}\]
   Done if verified; else, a concrete counterexample \$\text{env}\$ is returned as witness.
CEGIS: A Common Approach for Program Synthesis

original synthesis formulation

run { some prog: ASTNode | all env: Var -> Val | spec[prog, env] }

Counter-Example Guided Inductive Synthesis [Solar-Lezama, ASPLOS'06]

1. search: find some program and some environment s.t. the spec holds, i.e.,
   run { some prog: ASTNode | some env: Var -> Val | spec[prog, env] }
   to get a concrete candidate program $prog$

2. verification: check if $prog$ holds for all possible environments:
   check { all env: Var -> Val | spec[$prog, env] }
   Done if verified; else, a concrete counterexample $env$ is returned as witness.

3. induction: incrementally find a new program that additionally satisfies $env$: 
   run { some prog: ASTNode | 
     some env: Var -> Val | spec[prog, env] and spec[prog, $env] } 
   If UNSAT, return no solution; else, go to 2.
Program Synthesis with ALLOY

abstract sig Node {} abstract sig IntNode, BoolNode extends Node {} abstract sig Var extends IntNode {}

sig ITE extends IntNode { cond: one BoolNode, then: one IntNode, elsen: one IntNode }
sig GTE extends BoolNode { left: one IntNode, right: one IntNode }

program semantics

fact acyclic { all x: Node | x \notin x.^(cond+then+elsen+left+right) }

generic synthesis predicate

// for all 'eval' relations for which the semantics hold, the spec must hold as well
pred synth[root: Node] { all env: Var -> one Int | some eval: Node -> (Int+Bool) | env in eval and semantics[eval] and spec[root, eval] }

spec for max2 (the only benchmark-specific part)

one sig X, Y extends Var {}

// the result is equal to either X or Y and is greater or equal than both
AST nodes

abstract sig Node {}
abstract sig IntNode, BoolNode extends Node {}
abstract sig Var extends IntNode {}

sig ITE extends IntNode {
    cond: one BoolNode,
    then: one IntNode,
    elsen: one IntNode
}

sig GTE extends BoolNode {
    left: one IntNode,
    right: one IntNode
}
Program **Synthesis with ALLOY**

### AST nodes

abstract sig Node {}  
abstract sig IntNode, BoolNode extends Node {}  
abstract sig Var extends IntNode {}

sig ITE extends IntNode {  
  cond: one BoolNode,  
  then: one IntNode,  
  elsen: one IntNode  
}

sig GTE extends BoolNode {  
  left: one IntNode,  
  right: one IntNode  
}

### program semantics

fact acyclic {  
  all x: Node | x !in x.(cond+then+elsen+left+right)  
}

pred semantics[eval: Node -> (Int+Bool)] {  
  all n: IntNode | one eval[n] and eval[n] in Int  
  all n: BoolNode | one eval[n] and eval[n] in Bool  
  all n: ITE |  
    eval[n.cond] = True implies  
    eval[n.then] = eval[n] else eval[n.elsen] = eval[n]  
  all n: GTE |  
    eval[n.left] >= eval[n.right] implies  
    eval[n] = True else eval[n] = False  
}
Program **Synthesis** with ALLOY*

**AST nodes**

```latex
abstract sig Node {}
abstract sig IntNode, BoolNode extends Node {}
abstract sig Var extends IntNode {}

sig ITE extends IntNode {
    cond: one BoolNode,
    then: one IntNode,
    elsen: one IntNode
}

sig GTE extends BoolNode {
    left: one IntNode,
    right: one IntNode
}
```

**program semantics**

```latex
fact acyclic {
    all x: Node | x !in x.(cond+then+elsen+left+right)
}

pred semantics[eval: Node -> (Int+Bool)] {
    all n: IntNode | one eval[n] and eval[n] in Int
    all n: BoolNode | one eval[n] and eval[n] in Bool
    all n: ITE |
        eval[n.cond] = True implies
        eval[n.then] = eval[n] else eval[n.elsen] = eval[n]
    all n: GTE |
        eval[n.left] >= eval[n.right] implies
        eval[n] = True else eval[n] = False
}
```

**generic synthesis predicate**

```latex
// for all 'eval' relations for which the
// semantics hold, the spec must hold as well
pred synth[root: Node] {
    all env: Var -> one Int |
    some eval: Node -> (Int+Bool) |
        env in eval and
        semantics[eval] and
        spec[root, eval]
}
```
Program **Synthesis** with **ALLOY**

**AST nodes**

abstract sig Node {}
abstract sig IntNode, BoolNode extends Node {}
abstract sig Var extends IntNode {}

sig ITE extends IntNode {
    cond: one BoolNode,
    then: one IntNode,
    else: one IntNode
}

sig GTE extends BoolNode {
    left: one IntNode,
    right: one IntNode
}

**program semantics**

fact acyclic {
    all x: Node | x !in x.^{cond+then+elsen+left+right}
}

pred semantics[eval: Node -> (Int+Bool)] {
    all n: IntNode | one eval[n] and eval[n] in Int
    all n: BoolNode | one eval[n] and eval[n] in Bool
    all n: ITE |
        eval[n.cond] = True implies
        eval[n.then] = eval[n] else eval[n.elsen] = eval[n]
    all n: GTE |
        eval[n.left] >= eval[n.right] implies
        eval[n] = True else eval[n] = False
}

**generic synthesis predicate**

// for all 'eval' relations for which the semantics hold, the spec must hold as well
pred synth[root: Node] {
    all env: Var -> one Int |
        some eval: Node -> (Int+Bool) |
            env in eval and
            semantics[eval] and
            spec[root, eval]
}

**spec for max2** (the only benchmark-specific part)

one sig X, Y extends Var {}

// the result is equal to either X or Y and // is greater or equal than both
pred spec[root: Node, eval: Node -> (Int+Bool)] {
    (eval[root] = eval[X] or eval[root] = eval[Y]) and
    (eval[root] >= eval[X] and eval[root] >= eval[Y])
}
1. candidate search

\[
\text{facts[]} \land \\
\text{some prog: Node} \mid \\
\text{all env: Var \to one Int} \mid \\
\text{some eval: Node \to (Int+Bool)} \mid \\
\text{env in eval} \land \\
\text{semantics[eval]} \land \\
\text{spec[prog, eval]}
\]
**ALLOY Execution: Example**

1. **candidate search**

```plaintext
facts[] and
some prog: Node | all env: Var -> one Int | some eval: Node -> (Int+Bool) | env in eval and semantics[eval] and spec[prog, eval]
```

// NNF + skolemized

```plaintext
facts[] and $prog in Node and
all env: Var -> one Int | some eval: Node -> (Int+Bool) | env in eval and semantics[eval] and spec[$prog, eval]
```

2. **verification**

```plaintext
not (all env: Var -> one Int | some eval: Node -> (Int+Bool) | env in eval and semantics[eval] and spec[$prog, eval])
```

implemented as "partial instance"

// NNF + skolemized

```plaintext
!($env in Node -> Int all eval: Node -> (Int+Bool) | !($env in eval) or !semantics[eval] or !spec[$prog, eval])
```

// converted to Proc

```plaintext
E A (conj: facts[] and $prog in Node, // used for search eQuant: some env | some eval, // used for verification aQuant: all env | some eval, // used for verification)
```

3. **induction**

```plaintext
facts[] and some prog: Node | some eval: Node -> (Int+Bool) | env in eval and semantics[eval] and spec[prog, eval]
```

• body of aQuant from step 1 with env replaced • by the concrete value ($env_cex)

• implemented using "incremental solving"

```plaintext
body of aQuant from step 1 with env replaced • by the concrete value ($env_cex)
```

• implemented using "incremental solving"
ALLOY* Execution: Example

1. candidate search

\[
\begin{align*}
\text{facts}[] \quad \text{and} \\
\text{some prog: Node |} \\
\text{all env: Var -&gt; one Int |} \\
\text{some eval: Node -&gt; (Int+Bool) |} \\
\text{env in eval \quad and} \\
\text{semantics[eval] \quad and} \\
\text{spec[prog, eval]} \\
\end{align*}
\]

// NNF + skolemized

\[
\begin{align*}
\text{facts}[] \quad \text{and} \\
\text{some prog: Node |} \\
\text{all env: Var -&gt; one Int |} \\
\text{some eval: Node -&gt; (Int+Bool) |} \\
\text{env in eval \quad and} \\
\text{semantics[eval] \quad and} \\
\text{spec[prog, eval]} \\
\end{align*}
\]

// converted to Proc

\[
\begin{align*}
\exists (\text{conj: facts}[] \quad \text{and} \quad \text{prog in Node,} \\
\text{all env: Var -&gt; one Int |} \\
\text{some eval: Node -&gt; (Int+Bool) |} \\
\text{env in eval \quad and} \\
\text{semantics[eval] \quad and} \\
\text{spec[prog, eval]} \\
\end{align*}
\]

// used for search

\[
\begin{align*}
(e\text{Quant: some env | some eval ...}, \\
\text{all env: Var -&gt; one Int |} \\
\text{some eval: Node -&gt; (Int+Bool) |} \\
\text{env in eval \quad and} \\
\text{semantics[eval] \quad and} \\
\text{spec[prog, eval]} \\
\end{align*}
\]

// used for verification

\[
\begin{align*}
(a\text{Quant: all env | some eval ...})
\end{align*}
\]

• body of \(a\text{Quant}\) from step 1 with \textit{env} replaced
  • by the concrete value \((\text{id}_cex)\) from step 2
  • implemented using “incremental solving”
**ALLOY* Execution: Example**

### 1. candidate search

- `facts[] and some prog: Node | all env: Var -> one Int | some eval: Node -> (Int+Bool) | env in eval and semantics[eval] and spec[prog, eval]`

  // NNF + skolemized

- `facts[] and $prog in Node and all env: Var -> one Int | some eval: Node -> (Int+Bool) | env in eval and semantics[eval] and spec[$prog, eval]`

  // converted to Proc

\[
\forall (conj: facts[] and $prog in Node, \\
\quad \text{// used for search} \\
\quad e\text{Quant}: \text{some env | some eval ...}, \\
\quad \text{// used for verification} \\
\quad a\text{Quant}: \text{all env | some eval ...})
\]

### 2. verification

- `not(all env: Var -> one Int | some eval: Node -> (Int+Bool) | env in eval and semantics[eval] and spec[prog, eval])`

  implemented as “partial instance”
**ALLOY** Execution: **Example**

1. candidate search

```plaintext
facts[] and
some prog: Node |
all env: Var -> one Int |
some eval: Node -> (Int+Bool) |
env in eval and
semantics[eval] and
spec[prog, eval] // NNF + skolemized
```

```plaintext
facts[] and $prog in Node and
all env: Var -> one Int |
some eval: Node -> (Int+Bool) |
env in eval and
semantics[eval] and
spec[$prog, eval] // converted to Proc
```

```plaintext
∃∀(conj: facts[] and $prog in Node, // used for search
eQuant: some env | some eval ..., // used for verification
aQuant: all env | some eval ...)
```

2. verification

```plaintext
not(all env: Var -> one Int |
some eval: Node -> (Int+Bool) |
env in eval and
semantics[eval] and
spec[$prog, eval]
```

```plaintext
// NNF + skolemized
$env in Node -> Int
all eval: Node -> (Int+Bool) |
!($env in eval) or
!semantics[eval] or
!spec[$prog, eval] // converted to Proc
```

implemented as “partial instance”
1. candidate search

\[
\text{facts[]} \text{ and some prog: Node} \mid \\
\text{all env: Var -> one Int} \mid \\
\text{some eval: Node -> (Int Bool)} \mid \\
\text{env in eval and} \\
\text{semantics[eval] and} \\
\text{spec[prog, eval]}
\]

// NNF + skolemized
\[
\text{facts[]} \text{ and $prog$ in Node} \text{ and} \\
\text{all env: Var -> one Int} \mid \\
\text{some eval: Node -> (Int Bool)} \mid \\
\text{env in eval and} \\
\text{semantics[eval] and} \\
\text{spec[$prog$, eval]}
\]

// converted to Proc
\[
\exists (\text{conj: facts[]} \text{ and }$prog$ \text{ in Node,} \\
\text{used for search} \\
\text{eQuant: some env} \mid \text{some eval ...}, \\
\text{used for verification} \\
\text{aQuant: all env} \mid \text{some eval ...})
\]

2. verification

\[
\text{not(all env: Var -> one Int} \mid \\
\text{some eval: Node -> (Int Bool)} \mid \\
\text{env in eval and} \\
\text{semantics[eval] and} \\
\text{spec[prog, eval]})
\]

// NNF + skolemized
\[
\text{env in Node -> Int} \\
\text{all eval: Node -> (Int Bool) \mid} \\
!(\text{env in eval}) \text{ or} \\
!\text{semantics[eval] or} \\
!\text{spec[prog, eval]}
\]

// converted to Proc
\[
\exists (\text{conj: env in Node -> Int,} \\
\text{used for search} \\
\text{eQuant: some eval ...}, \\
\text{used for verification} \\
\text{aQuant: all eval ...})
\]
# ALLOY Execution: Example

## 1. candidate search

\[
\text{facts}[] \land \\
\exists \text{ prog: Node} | \\
\forall \text{ env: Var} \rightarrow \exists \text{ Int} | \\
\exists \text{ eval: Node} \rightarrow (\text{Int} + \text{Bool}) | \\
\text{env in eval} \land \\
\text{semantics[eval]} \land \\
\text{spec[prog, eval]}
\]

// converted to Proc
\[
\forall (\text{conj: facts}[] \land \text{prog in Node}, \\
\text{used for search}) \\
\forall (\text{eQuant: some env \lor some eval ...}, \\
\text{used for verification}) \\
\forall (\text{aQuant: all env \lor some eval ...})
\]

## 2. verification

\[
\neg (\forall \text{ env: Var} \rightarrow \exists \text{ Int} | \\
\exists \text{ eval: Node} \rightarrow (\text{Int} + \text{Bool}) | \\
\text{env in eval} \land \\
\text{semantics[eval]} \land \\
\text{spec[prog, eval]})
\]

// converted to Proc
\[
\forall (\text{conj: env in Node} \rightarrow \text{Int}, \\
\text{used for search}) \\
\forall (\text{eQuant: some eval ...}, \\
\text{used for verification}) \\
\forall (\text{aQuant: all eval ...})
\]

## 3. induction

\[
\text{facts}[] \land \\
\exists \text{ prog: Node} | \\
\exists \text{ env: Var} \rightarrow \exists \text{ Int} | \\
\exists \text{ eval: Node} \rightarrow (\text{Int} + \text{Bool}) | \\
\text{env in eval} \land \\
\text{semantics[eval]} \land \\
\text{spec[prog, eval]}
\]

- body of \text{aQuant} from step 1 with \text{env} replaced
  by the concrete value ($\text{env}_c$) from step 2
- implemented using “incremental solving”
Semantics: General Idea

1. convert formula to Negation Normal Form (NNF)
   - boolean connectives left: $\land$, $\lor$, $\neg$
   - negation pushed to leaf nodes
   - no negated quantifiers

2. perform skolemization
   - top-level $\exists$ quantifiers replaced by skolem variables (relations)

3. decompose formula into a tree of FOL, OR, and $\forall$A nodes
   - FOL: first-order formula
   - OR: disjunction
   - $\forall$A: higher-order top-level $\forall$ quantifier (not skolemizable)

4. solve using the following decision procedure
   - FOL: solve directly with Kodkod (first-order relational solver)
   - OR: solve each disjunct separately
   - $\forall$A: apply CEGIS
Semantics: General Idea

1. convert formula to Negation Normal Form (NNF)
   → boolean connectives left: ∧, ∨,¬
   → negation pushed to leaf nodes
   → no negated quantifiers

2. perform skolemization
   → top-level ∃ quantifiers replaced by skolem variables (relations)
Semantics: General Idea

1. convert formula to Negation Normal Form (NNF)
   → boolean connectives left: ∧, ∨, ¬
   → negation pushed to leaf nodes
   → no negated quantifiers

2. perform skolemization
   → top-level ∃ quantifiers replaced by skolem variables (relations)

3. decompose formula into a tree of FOL, OR, and ∀∃ nodes
   → FOL : first-order formula
   → OR : disjunction
   → ∀∃ : higher-order top-level ∀ quantifier (not skolemizable)
Semantics: General Idea

1. convert formula to Negation Normal Form (NNF)
   → boolean connectives left: ∧, ∨, ¬
   → negation pushed to leaf nodes
   → no negated quantifiers

2. perform skolemization
   → top-level ∃ quantifiers replaced by skolem variables (relations)

3. decompose formula into a tree of FOL, OR, and ∀ nodes
   → FOL : first-order formula
   → OR : disjunction
   → ∀ : higher-order top-level ∀ quantifier (not skolemizable)

4. solve using the following decision procedure
   → FOL : solve directly with Kodkod (first-order relational solver)
   → OR : solve each disjunct separately
   → ∀ : apply CEGIS
type Proc = FOL(form: Formula)  // first-order formula
  | OR(disjs: Proc list)  // list of disjuncts (at least some should be higher-order)
  | ∀(conj: FOL,
      allForm: Formula,  // original ∀x·f formula
      existsProc: Proc)  // translation of the dual ∃ formula (T(∃x·f))
Semantics: Formula Decomposition

```haskell
type Proc = FOL(form: Formula)  // first-order formula
  \* OR(disjs: Proc list)          // list of disjuncts (at least some should be higher-order)
  \* \forall(conj: FOL,          // first-order conjuncts (alongside the higher-order \forall quantifier)
    allForm: Formula,          // original \forall x.f formula
    existsProc: Proc)        // translation of the dual \exists formula (T(\exists x.f))

T : Formula \rightarrow Proc  // translates arbitrary formula to a tree of Procs
let T = \lambda(f).
```
Semantics: Formula Decomposition

```plaintext
type Proc = FOL(form: Formula)  // first-order formula
| OR(disjs: Proc list)         // list of disjuncts (at least some should be higher-order)
| ∀(conj: FOL,
    allForm: Formula,  // original ∀x·f formula
    existsProc: Proc)  // translation of the dual ∃ formula (T(∃x·f))

T : Formula → Proc // translates arbitrary formula to a tree of Procs
let T = λ(f).
  let fnnf = skolemize(nnf(f))  // convert to NNF and skolemize
```

Semantics: Formula Decomposition

\[
\text{type \ Proc} = \text{FOL}(\text{form: Formula}) \quad \text{// first-order formula}
\]

\[
| \ 	ext{OR(\text{disjs: Proc list})} \quad \text{// list of disjuncts (at least some should be higher-order)}
\]

\[
| \ 	ext{∃∀(\text{conj: FOL,}}
\]

\[
\text{allForm: Formula,} \quad \text{// original } \forall x \cdot f \text{ formula}
\]

\[
\text{existsProc: Proc) \quad \text{// translation of the dual } \exists \text{ formula } (T(∃x \cdot f))
\]

\[
T : \text{Formula} \rightarrow \text{Proc} \quad \text{// translates arbitrary formula to a tree of Procs}
\]

\[
\text{let } T = \lambda(f). \quad \text{translating negation}
\]

\[
\text{let } f_{\text{nnf}} = \text{skolemize(nnf}(f))
\]

\[
\text{match } f_{\text{nnf}} \text{ with}
\]

\[
\text{| ¬f \rightarrow FOL}(f_{\text{nnf}})
\]

- negation can be only in leaves

\[
\Rightarrow \text{must be first-order}
\]
**Semantics: Formula Decomposition**

```haskell
type Proc = FOL(form: Formula) // first-order formula
 | OR(disjs: Proc list) // list of disjuncts (at least some should be higher-order)
 | ∀(conj: FOL,
   allForm: Formula, // original ∀x·f formula
   existsProc: Proc) // translation of the dual ∃ formula (T(∃x·f))

T : Formula → Proc // translates arbitrary formula to a tree of Pros

let T = λ(f).
let fnnf = skolemize(nnf(f))
match fnnf with
 | f → FOL(fnnf)
 | ¬f → FOL(¬f)
 | ∀x·f → fail "can't happen"
```

**Translating the ∃ quantifier**

- There can’t be top-level ∃ quantifiers after skolemization.
Semantics: Formula Decomposition

**Type**

\[ \text{type} \ Proc = \text{FOL}(\text{form}: \text{Formula}) \quad \text{// first-order formula} \]

\[ \quad | \quad \text{OR}(\text{disjs}: \text{Proc list}) \quad \text{// list of disjuncts (at least some should be higher-order)} \]

\[ \quad | \quad \exists(\text{conj}: \text{FOL,}) \quad \text{// first-order conjuncts (alongside the higher-order } \forall \text{ quantifier)} \]

\[ \quad \quad \text{allForm: Formula,} \quad \text{// original } \forall x \cdot f \text{ formula} \]

\[ \quad \quad \text{existsProc: Proc} \quad \text{// translation of the dual } \exists \text{ formula } (\mathcal{T}(\exists x \cdot f)) \]

\[ \mathcal{T} : \text{Formula} \rightarrow \text{Proc} \quad \text{// translates arbitrary formula to a tree of Procs} \]

\[
\begin{align*}
\text{let } \mathcal{T} & = \lambda(f). \\
\text{let } f_{\text{nff}} & = \text{skolemize}(\text{nnf}(f)) \\
\text{match } f_{\text{nff}} \text{ with} \\
| \neg f_s & \rightarrow \text{FOL}(f_{\text{nff}}) \\
| \exists x \cdot f_s & \rightarrow \text{fail} \ "\text{can't happen}\" \\
| \forall x \cdot f_s & \rightarrow \text{let } p = \mathcal{T}((\exists x \cdot f_s) \\
\text{if } (x.\text{mult} = \text{SET}) \quad \| \quad \neg(p \text{ is FOL}) \\
\text{\quad } \exists \forall(\text{FOL(true), } f_{\text{nff}}, p) \\
\text{else} \\
\text{\quad FOL}(f_{\text{nff}}) \\
\end{align*}
\]

**Translating the } \forall \text{ quantifier**

- translate the dual } \exists \text{ formula first \ (where the } \exists \text{ quantifier will be skolemizable) \\

- if multiplicity of this } \forall \text{ quantifier is } \text{SET or the dual is } \text{not} \text{ first-order \\
  - then: } f_{\text{nff}} \text{ is higher-order} \\
    \quad \rightarrow \text{create } \exists \forall \text{ node} \\
  - else: } f_{\text{nff}} \text{ is first-order} \\
    \quad \rightarrow \text{create FOL node}
Semantics: Formula Decomposition

\[ \textbf{type} \ Proc = \text{FOL}(\text{form: Formula}) \quad // \text{first-order formula} \\
| \quad \text{OR}(\text{disjs: Proc list}) \quad // \text{list of disjuncts (at least some should be higher-order}) \\
| \quad \exists\forall(\text{conj: FOL, allForm: Formula, existsProc: Proc}) \quad // \text{first-order conjuncts (alongside the higher-order } \forall \text{ quantifier}) \\
\]

\[ \mathcal{T}: \text{Formula} \rightarrow \text{Proc} \quad // \text{translates arbitrary formula to a tree of Procs} \]

\[ \text{let } \mathcal{T} = \lambda(f). \]

\[ \text{let } f_{\text{nnf}} = \text{skolemize}(\text{nnf}(f)) \]

\[ \text{match } f_{\text{nnf}} \text{ with} \]

\[ | \quad \neg f_s \quad \rightarrow \quad \text{FOL}(f_{\text{nnf}}) \]

\[ | \quad \exists x \cdot f_s \quad \rightarrow \quad \text{fail} "\text{can't happen}" \]

\[ | \quad \forall x \cdot f_s \quad \rightarrow \quad \text{let } p = \mathcal{T}(\exists x \cdot f_s) \]

\[ \quad \text{if } (x.\text{mult} = \text{SET}) \ || \quad \neg (p \text{ is FOL}) \]

\[ \quad \exists\forall(\text{FOL(true), } f_{\text{nnf}}, p) \]

\[ \quad \text{else} \]

\[ \quad \text{FOL}(f_{\text{nnf}}) \]

\[ | \quad f_1 \lor f_2 \quad \rightarrow \quad \text{OR}([\mathcal{T}(f_1), \mathcal{T}(f_2)]) \]

**translating disjunction**

- translate both disjuncts
- skolemization through disjunction is not sound → must create OR node (and later solve each side separately)
- optimization: only if \( f_1 \lor f_2 \) is first-order as a whole, then it is safe to return \( \text{FOL}(f_1 \lor f_2) \)
Semantics: Formula Decomposition

def type Proc = FOL(form: Formula)  // first-order formula
      | OR(disjs: Proc list)  // list of disjuncts (at least some should be higher-order)
      | ∀(conj: FOL,
         allForm: Formula,  // first-order conjuncts (alongside the higher-order ∀ quantifier)
         existsProc: Proc)  // original ∀x.f formula
    // translation of the dual ∃ formula (T(∃x.f))

T : Formula → Proc  // translates arbitrary formula to a tree of Procs

let T = λ(f).
    let fnnf = skolemize(nnf(f))
    match fnnf with
      | ¬fs  → FOL(fnnf)
      | ∃x.fs → fail "can’t happen"
      | ∀x.fs → let p = T(∃x.fs)
          if (x.mult = SET) || ¬(p is FOL)
              ∀(FOL(true),fnnf,p)
          else
              FOL(fnnf)
      | f1 ∨ f2 → OR([T(f1), T(f2)])
      | f1 ∧ f2 → T(f1) ∧ T(f2)

translating conjunction

- translate both conjuncts
- compose the two resulting Procs

FOL ∧ FOL → FOL
FOL ∧ OR → OR
FOL ∧ ∀ → ∀
OR ∧ OR → OR
OR ∧ ∀ → OR
∀ ∧ ∀ → ∀
Semantics: Formula Evaluation

\[ S : \text{Proc} \to \text{Instance} \text{ option} \]

\[
\text{let } S = \lambda(p). \]

\[
\text{let } S = \lambda(p). \]

\[
\text{match } p \text{ with}
\]

\[
| \text{FOL} \to \text{solve } p \text{ form} \]

\[
| \text{OR} \to \ldots \text{ // apply } S \text{ to each Proc in } p \text{ disj}; \text{return the first solution found} \]

\[
| \text{E} \to \text{let } p \text{ cand} = p \text{ conj} \sqcup p \text{ existsProc}
\]

\[
\text{match } S(p \text{ cand}) \text{ with}
\]

\[
| \text{None} \to \text{None} \text{ // no candidate solution found} \Rightarrow \text{return UNSAT} \]

\[
| \text{Some}(cand) \to \text{ // candidate solution found} \Rightarrow \text{proceed to verify the candidate}
\]

\[
\text{match } S(T(\neg p \text{ allForm})) \text{ with} \]

\[
| \text{None} \to \text{Some}(cand) \text{ // no counterexample found} \Rightarrow \text{cand} \text{ is the solution} \]

\[
| \text{Some}(cex) \to \text{let } q = p \text{ allForm} \]

\[
\text{// encode the counterexample as a formula: use only the body of the } \forall \text{ quant.}
\]

\[
\text{// in which the quant. variable is replaced with its concrete value in cex}
\]

\[
\text{let } f\text{ cex} = \text{replace}(q \text{. body}, q \text{. var}, \text{eval}(cex, q \text{. var})) \]

\[
\text{// add the counterexample encoding to the candidate search condition}
\]

\[
S(p \text{ cand} \sqcup T(f\text{ cex})) \]

\[
\text{partial instance encode cand as partial instance}
\]

\[
\text{counterexample encoding}
\]

\[
\text{no domain-specific knowledge necessary}
\]

\[
\text{incremental solving}
\]

\[
\text{add } T(f\text{ cex}) \text{ to the existing } S(p \text{ cand}) \text{ solver}
\]

28
Semantics: Formula Evaluation

\[ S : \text{Proc} \rightarrow \text{Instance option} \]

\[ \text{let } S = \lambda(p). \]

\[ \text{match } p \text{ with} \]

\[ | \text{FOL} \rightarrow \text{solve } p.form \]

\[ \text{match } S \text{ to each Proc in } p \text{ disj; return the first solution found} \]

\[ \text{let } p_{\text{cand}} = p_{\text{conj}} \equiv p_{\text{existsProc}} \text{ match } S(p_{\text{cand}}) \text{ with} \]

\[ | \text{None} \rightarrow \text{None} \] \[ \Rightarrow \text{return UNSAT} \]

\[ | \text{Some}(cand) \rightarrow \text{candidate solution found} \] \[ \Rightarrow \text{proceed to verify the candidate} \]

\[ \text{match } S(T(\neg p_{\text{allForm}})) \text{ with} \]

\[ | \text{None} \rightarrow \text{Some}(cand) \] \[ \Rightarrow \text{no counterexample found} \] \[ \Rightarrow \text{cand is the solution} \]

\[ | \text{Some}(cex) \rightarrow \text{let } q = p_{\text{allForm}} \]

\[ \text{encoding the counterexample as a formula: use only the body of the } \forall \text{ quant.} \]

\[ \text{in which the quant. variable is replaced with its concrete value in } cex \]

\[ \text{let } f_{cex} = \text{replace}(q_{\text{body}}, q_{\text{var}}, \text{eval}(cex, q_{\text{var}})) \]

\[ \text{add the counterexample encoding to the candidate search condition} \]

\[ S(p_{\text{cand}}) \equiv T(f_{cex}) \]

\[ \text{partial instance encode cand as partial instance} \]

\[ \text{counterexample encoding} \]

\[ \text{no domain-specific knowledge necessary} \]

\[ \text{incremental solving} \]

\[ \text{add } T(f_{cex}) \text{ to the existing } S(p_{\text{cand}}) \text{ solver} \]
Semantics: Formula Evaluation

\[ S : \text{Proc} \rightarrow \text{Instance option} \]

let \( S = \lambda(p) \cdot \)

match \( p \) with
  | FOL \rightarrow solve \( p.form \)
  | OR \rightarrow \ldots \text{ // apply } S \text{ to each } \text{Proc in } p.disj; \text{return the first solution found}
Semantics: Formula Evaluation

\[ S : \text{Proc} \rightarrow \text{Instance option} \]

\[
\text{let } S = \lambda(p) .
\]

\[
\text{match } p \text{ with}
\]

\[
| \text{FOL} \rightarrow \text{solve } p.\text{form}
\]

\[
| \text{OR} \rightarrow \ldots \quad \text{// apply } S \text{ to each Proc in } p.\text{disj; return the first solution found}
\]

\[
| \exists \rightarrow \text{let } p_{\text{cand}} = p.\text{conj} \land p.\text{existsProc}
\]

\[
\text{match } S(p_{\text{cand}}) \text{ with}
\]

\[
| \text{None} \rightarrow \text{None} \quad \text{// no candidate solution found } \Rightarrow \text{return UNSAT}
\]

\[
| \text{Some}(\text{cand}) \rightarrow \quad \text{// candidate solution found } \Rightarrow \text{proceed to verify the candidate}
\]

\[
\text{match } S(T(\neg p.\text{allForm})) \text{ with} \quad \text{// try to falsify } \text{cand } \Rightarrow \text{must run } S \text{ against the } \text{cand} \text{ instance}
\]

\[
| \text{None} \rightarrow \text{Some}(\text{cand}) \quad \text{// no counterexample found } \Rightarrow \text{cand is the solution}
\]

\[
| \text{Some} (\text{cex}) \rightarrow \text{let } q = p.\text{allForm}
\]

\[
\quad \text{// encode the counterexample as a formula: use only the body of the } \forall \text{ quant.}
\]

\[
\quad \text{// in which the quant. variable is replaced with its concrete value in } \text{cex}
\]

\[
\quad \text{let } f_{\text{cex}} = \text{replace}(q.\text{body}, q.\text{var}, \text{eval}(\text{cex}, q.\text{var}))
\]

\[
\quad \text{// add the counterexample encoding to the candidate search condition}
\]

\[
S(p_{\text{cand}} \land T(f_{\text{cex}}))
\]
**Semantics: Formula Evaluation**

\[ S : \text{Proc} \rightarrow \text{Instance option} \]

\[
\text{let } S = \lambda(p). \\
\quad \text{match } p \text{ with} \\
\quad | \text{FOL} \rightarrow \text{solve } p.\text{form} \\
\quad | \text{OR} \rightarrow ... \quad \text{// apply } S \text{ to each Proc in } p.\text{disj}; \text{return the first solution found} \\
\quad | \exists \rightarrow \text{let } p_{\text{cand}} = p.\text{conj} \land p.\text{existsProc} \\
\quad \quad \text{match } S(p_{\text{cand}}) \text{ with} \\
\quad \quad | \text{None} \rightarrow \text{None} \quad \text{// no candidate solution found } \Rightarrow \text{return UNSAT} \\
\quad \quad | \text{Some}(cand) \rightarrow \quad \text{// candidate solution found } \Rightarrow \text{proceed to verify the candidate} \\
\quad \quad \quad \text{match } S(T(\neg p.\text{allForm})) \text{ with} \quad \text{// try to falsify } cand \Rightarrow \text{must run } S \text{ against the } cand \text{ instance} \\
\quad \quad \quad | \text{None} \rightarrow \text{Some}(cand) \quad \text{// no counterexample found } \Rightarrow \text{cand is the solution} \\
\quad \quad \quad | \text{Some}(cex) \rightarrow \text{let } q = p.\text{allForm} \\
\quad \quad \quad \quad \quad \text{// encode the counterexample as a formula: use only the body of the } \forall \text{ quant.} \\
\quad \quad \quad \quad \quad \text{// in which the quant. variable is replaced with its concrete value in } cex \\
\quad \quad \quad \quad \quad \text{let } f_{\text{cex}} = \text{replace}(q.\text{body}, q.\text{var}, \text{eval}(cex, q.\text{var})) \\
\quad \quad \quad \quad \quad \text{// add the counterexample encoding to the candidate search condition} \\
\quad \quad \quad \quad \quad S(p_{\text{cand}} \land T(f_{\text{cex}}))
\]
Semantics: Formula Evaluation

\[ S : \text{Proc} \to \text{Instance option} \]

\[
\text{let } S = \lambda(p). \\
\text{match } p \text{ with} \]
\[ | \text{FOL} \to \text{solve } p.\text{form} \]
\[ | \text{OR} \to \ldots \text{ // apply } S \text{ to each } \text{Proc in } p.\text{disj; return the first solution found} \]
\[ | \exists \to \text{let } p_{\text{cand}} = p.\text{conj} \land p.\text{existsProc} \\
\quad \text{match } S(p_{\text{cand}}) \text{ with} \]
\[ | \text{None} \to \text{None} \text{ // no candidate solution found } \Rightarrow \text{return UNSAT} \]
\[ | \text{Some(}c\text{and}) \to \text{ // candidate solution found } \Rightarrow \text{proceed to verify the candidate} \]
\[ \quad \text{match } S(T(\neg p.\text{allForm})) \text{ with} \text{ // try to falsify } c\text{and} \Rightarrow \text{must run } S \text{ against the } c\text{and instance} \]
\[ | \text{None} \to \text{Some(}c\text{and}) \text{ // no counterexample found } \Rightarrow \text{cand is the solution} \]
\[ | \text{Some(}c\text{ex}) \to \text{let } q = p.\text{allForm} \]
\[ \quad \text{// encode the counterexample as a formula: use only the body of the } \forall \text{ quant.} \]
\[ \quad \text{// in which the quant. variable is replaced with its concrete value in } c\text{ex} \]
\[ \quad \text{let } f_{\text{cex}} = \text{replace}(q.\text{body}, q.\text{var}, \text{eval}(c\text{ex}, q.\text{var})) \]
\[ \quad \text{// add the counterexample encoding to the candidate search condition} \]
\[ S(p_{\text{cand}} \land T(f_{\text{cex}})) \]

- **Partial instance**: Encode `cand` as partial instance
- **Counterexample encoding**: No domain-specific knowledge necessary
- **Incremental solving**: Add `T(f_{\text{cex}})` to the existing `S(p_{\text{cand}})` solver
Semantics: Formula Evaluation

\[ S : \text{Proc} \to \text{Instance} \text{ option} \]

\[
\text{let } S = \lambda(p). \\
\quad \text{match } p \text{ with } \\
\quad \quad | \text{FOL} \to \text{solve } p.\text{form} \\
\quad \quad | \text{OR} \to \ldots \quad // \text{apply } S \text{ to each Proc in } p.\text{disj}; \text{return the first solution found} \\
\quad \quad | \exists \to \text{let } p_{\text{cand}} = p.\text{conj} \land p.\text{existsProc} \\
\quad \quad \quad \text{match } S(p_{\text{cand}}) \text{ with } \\
\quad \quad \quad \quad | \text{None} \to \text{None} \quad // \text{no candidate solution found} \Rightarrow \text{return UNSAT} \\
\quad \quad \quad \quad | \text{Some}(\text{cand}) \to \quad // \text{candidate solution found} \Rightarrow \text{proceed to verify the candidate} \\
\quad \quad \quad \quad \quad \text{match } S(T(\neg p.\text{allForm})) \text{ with } \quad // \text{try to falsify } \text{cand} \Rightarrow \text{must run } S \text{ against the } \text{cand} \text{ instance} \\
\quad \quad \quad \quad \quad \quad | \text{None} \to \text{Some}(\text{cand}) \quad // \text{no counterexample found} \Rightarrow \text{cand is the solution} \\
\quad \quad \quad \quad \quad \quad | \text{Some}(\text{cex}) \to \text{let } q = p.\text{allForm} \\
\quad \quad \quad \quad \quad \quad \quad // \text{encode the counterexample as a formula: use only the body of the } \forall \text{ quant.} \\
\quad \quad \quad \quad \quad \quad \quad // \text{in which the quant. variable is replaced with its concrete value in } \text{cex} \\
\quad \quad \quad \quad \quad \quad \quad \text{let } f_{\text{cex}} = \text{replace}(q.\text{body}, \text{} q.\text{var}, \text{eval}(\text{cex}, \text{} q.\text{var})) \\
\quad \quad \quad \quad \quad \quad \quad \quad // \text{add the counterexample encoding to the candidate search condition} \\
\quad \quad \quad \quad \quad \quad \quad \quad S(p_{\text{cand}} \land T(f_{\text{cex}})) \\
\]

partial instance encode \( \text{cand} \) as partial instance

counterexample encoding no domain-specific knowledge necessary

incremental solving add \( T(f_{\text{cex}}) \) to the existing \( S(p_{\text{cand}}) \) solver

28
Optimization 1: **Domain Constraints**

**problem:** domain for eval too unconstrained

```
pred synth[root: Node] {  
    all eval: Node -> (Int+Bool) |  
    semantics[eval] implies spec[root, eval]  
}  
```
Optimization 1: **Domain Constraints**

**problem**: domain for `eval` too unconstrained

```plaintext
def synth[root: Node] {
    all eval: Node -> (Int+Bool) |
    semantics[eval] implies spec[root, eval]
}
```

→ candidate search condition:

```plaintext
def synth[root: Node] {
    all eval: Node -> (Int+Bool) |
    semantics[eval] implies spec[root, eval]
}
```

- a valid candidate **doesn’t** have to satisfy the `semantics` predicate!
Optimization 1: **Domain Constraints**

**problem:** domain for `eval` too unconstrained

```
pred synth[root: Node] {
  all eval: Node -> (Int+Bool) |
  semantics[eval] implies spec[root, eval]
}
```

→ candidate search condition:

```
some root: Node |
  some eval: Node -> (Int+Bool) |
  semantics[eval] implies spec[root, eval]
```

- a valid candidate **doesn’t** have to satisfy the `semantics` predicate!
- although logically correct, takes too many steps to converge

> “for all possible `eval`, if the semantics hold then the spec must hold”  
vs.  
> “for all `eval` that satisfy the semantics, the spec must hold”
Optimization 1: **Domain Constraints**

**problem:** domain for `eval` too unconstrained

```prolog
pred synth[root: Node] { 
  all eval: Node -> (Int+Bool) | 
  semantics[eval] \implies spec[root, eval] 
}
```

→ candidate search condition:

```prolog
some root: Node | 
  some eval: Node -> (Int+Bool) | 
  semantics[eval] \implies spec[root, eval]
```

- a valid candidate **doesn’t** have to satisfy the `semantics` predicate!
- although logically correct, takes too many steps to converge

- “for all possible `eval`, if the semantics hold then the `spec` must hold”
  vs. “for all `eval` that satisfy the semantics, the `spec` must hold”

**solution:** add new syntax for domain constraints

```prolog
pred synth[root: Node] { 
  all eval: Node -> (Int+Bool) 
  when semantics[eval] | 
  spec[root, eval] 
}
```
first-order logic semantics

\[
\begin{align*}
\text{all } x : X \text{ when } \text{dom}[x] \lor \text{body}[x] & \iff \text{all } x : X | \text{dom}[x] \text { implies body}[x] \\
\text{some } x : X \text{ when } \text{dom}[x] \lor \text{body}[x] & \iff \text{some } x : X | \text{dom}[x] \text { and body}[x]
\end{align*}
\]
first-order logic semantics

\[ \text{all } x : X \text{ when } \text{dom}[x] \mid \text{body}[x] \iff \text{all } x : X \mid \text{dom}[x] \text{ implies } \text{body}[x] \]

\[ \text{some } x : X \text{ when } \text{dom}[x] \mid \text{body}[x] \iff \text{some } x : X \mid \text{dom}[x] \text{ and } \text{body}[x] \]

De Morgan’s Laws (consistent with classical logic)

\[ \text{not } (\text{all } x : X \text{ when } \text{dom}[x] \mid \text{body}[x]) \iff \text{some } x : X \text{ when } \text{dom}[x] \mid \text{not } \text{body}[x] \]

\[ \text{not } (\text{some } x : X \text{ when } \text{dom}[x] \mid \text{body}[x]) \iff \text{all } x : X \text{ when } \text{dom}[x] \mid \text{not } \text{body}[x] \]
## Domain Constraints Semantics

### First-order logic semantics

\[
\text{all } x: X \text{ when } \text{dom}[x] \lor \text{body}[x] \iff \text{all } x: X \mid \text{dom}[x] \implies \text{body}[x]
\]

\[
\text{some } x: X \text{ when } \text{dom}[x] \lor \text{body}[x] \iff \text{some } x: X \mid \text{dom}[x] \land \text{body}[x]
\]

### De Morgan’s Laws (consistent with classical logic)

\[
\text{not } (\text{all } x: X \text{ when } \text{dom}[x] \lor \text{body}[x]) \iff \text{some } x: X \text{ when } \text{dom}[x] \land \text{not body}[x]
\]

\[
\text{not } (\text{some } x: X \text{ when } \text{dom}[x] \lor \text{body}[x]) \iff \text{all } x: X \text{ when } \text{dom}[x] \land \text{not body}[x]
\]

### Changes to the Alloy\(^*\) semantics

- Converting higher-order \(\forall\) to \(\exists\): \(\forall x \cdot f \rightarrow \exists x \cdot f\) (domain constraints stay with \(x\))
- Encoding a counterexample as a formula: in

\[
\text{let } f_{cex} = \text{replace}(q.body, q.var, \text{eval}(cex, q.var))
\]

\(q.body\) is expanded according to the first-order semantics above
Optimization 2: **First-Order Increments**

**problem:** search space too big, counterexamples not focused

```prolog
pred synth[root: Node] {
    all eval: Node -> (Int+Bool)
    when semantics[eval] |
        spec[root, eval]
}
```
**Optimization 2: First-Order Increments**

**Problem:** search space too big, counterexamples not focused

```plaintext
pred synth[root: Node] {
    all eval: Node -> (Int+Bool)
    when semantics[eval] |
        spec[root, eval]
}
```

- quantifies over evaluations of Nodes instead of only Vars
- counterexamples encode entire eval relation, instead of only values of variables
### Optimization 2: First-Order Increments

**Problem:** search space too big, counterexamples not focused

```plaintext
pred synth[root: Node] {
  all eval: Node -> (Int+Bool)
  when semantics[eval] | spec[root, eval] →
```

- quantifies over evaluations of Nodes instead of only Vars
- counterexamples encode entire eval relation, instead of only values of variables

**Idea:** rewrite the `synth` predicate to separate `env` from `eval`

```plaintext
pred synth[root: Node] {
  all env: Var -> one Int |
  some eval: Node -> (Int+Bool)
  when env in eval && semantics[eval] | spec[root, eval]
}
```
Optimization 2: **First-Order Increments**

**problem:** search space too big, counterexamples not focused

```
pred synth[root: Node] {
  all eval: Node -> (Int+Bool)
  when semantics[eval] | spec[root, eval]
}
```

- quantifies over evaluations of Nodes instead of only Vars
- counterexamples encode entire eval relation, instead of only values of variables

**idea:** rewrite the `synth` predicate to separate `env` from `eval`

```
pred synth[root: Node] {
  all env: Var -> one Int |
  some eval: Node -> (Int+Bool)
  when env in eval && semantics[eval] |
    spec[root, eval]
}
```

**consequence:** higher-order verification

```
not (all env: Var -> one Int |
    some eval: Node -> (Int+Bool)
    when env in eval && semantics[eval] |
    spec[$root, eval])
```
Optimization 2: **First-Order Increments**

**problem:** search space too big, counterexamples not focused

```latex
def pred synth[root: Node] {
    all eval: Node -> (Int+Bool)
    when semantics[eval] | spec[root, eval]  
}
```

- quantifies over evaluations of Nodes instead of only Vars
- counterexamples encode entire eval relation, instead of only values of variables

**idea:** rewrite the `synth` predicate to separate `env` from `eval`

```latex
pred synth[root: Node] {
    all env: Var -> one Int | some eval: Node -> (Int+Bool)
    when env in eval && semantics[eval] | spec[root, eval]
}
```

**consequence:** higher-order verification

```latex
\not \ (all \ env: \ Var \ -> \ one \ Int \ | \ some \ eval: \ Node \ -> \ (Int+Bool) \\
\begin{aligned}
when \ env \ in \ eval \ && \ semantics[eval] \ | \\
& spec[$root, eval])
\end{aligned} \\
\Leftrightarrow \\
\begin{aligned}
some \ env: \ Var \ -> \ one \ Int \ | \\
all \ eval: \ Node \ -> \ (Int+Bool) \\
when \ env \ in \ eval \ && \ semantics[eval] \ |
& \ not \ spec[$root, eval]
\end{aligned}
```
Optimization 2: **First-Order Increments**

**problem:** search space too big, counterexamples not focused

```plaintext
pred synth[root: Node] {
  all eval: Node -> (Int+Bool) when semantics[eval] |
  spec[root, eval]
}
```

- quantifies over evaluations of Nodes instead of only Vars
- counterexamples encode entire eval relation, instead of only values of variables

**idea:** rewrite the synth predicate to separate env from eval

```plaintext
pred synth[root: Node] {
  all env: Var -> one Int | some eval: Node -> (Int+Bool) when env in eval && semantics[eval] |
  spec[root, eval]
}
```

**consequence:** higher-order verification

```plaintext
not (all env: Var -> one Int | some eval: Node -> (Int+Bool) when env in eval && semantics[eval] |
  spec[$root, eval]) ⇔

  some env: Var -> one Int | all eval: Node -> (Int+Bool) when env in eval && semantics[eval] |
  not spec[$root, eval]
```

- nested CEGIS loops ✓
- higher-order counterexample encoding → cannot use incremental solving ✗
Optimization 2: **First-Order Increments**

**Problem:** search space too big, counterexamples not focused

```prolog
pred synth[root: Node] {
  all eval: Node -> (Int+Bool)  
  when semantics[eval] | spec[root, eval]  
}
```

- quantifies over evaluations of Nodes instead of only Vars
- counterexamples encode entire eval relation, instead of only values of variables

**Idea:** rewrite the `synth` predicate to separate `env` from `eval`

```prolog
pred synth[root: Node] {
  all env: Var -> one Int  
  some eval: Node -> (Int+Bool)  
  when env in eval && semantics[eval] | spec[root, eval]  
}
```

**Consequence:** higher-order verification

```prolog
not (all env: Var -> one Int  
    some eval: Node -> (Int+Bool)  
    when env in eval && semantics[eval] | spec[$root, eval])
```

↔

```prolog
some env: Var -> one Int  
all eval: Node -> (Int+Bool)  
when env in eval && semantics[eval] |  
    not spec[$root, eval]
```

- nested CEGIS loops ✔
- higher-order counterexample encoding
  → cannot use incremental solving ✗

**Solution:** force counterexample encodings to be first order
always translate the counterexample encoding formula to F0L

\[ S(p_{\text{cand}} \land T(f_{\text{cex}})) \]

\[ \downarrow \]

\[ S(p_{\text{cand}} \land T_{\text{fo}}(f_{\text{cex}})) \]
always translate the counterexample encoding formula to FOL

\[ S(p_{cand} \land T(f_{cex})) \]

\[ \downarrow \]

\[ S(p_{cand} \land \mathcal{T}_{fo}(f_{cex})) \]

apply the same idea of flipping \( \forall \) to \( \exists \) to implement \( \mathcal{T}_{fo} \)

```haskell
// \( \mathcal{T}_{fo} : \text{Formula} \rightarrow \text{FOL} \)
let \( \mathcal{T}_{fo}(f) = \text{match } p = T(f) \text{ with} \)
| FOL  \rightarrow p
| \exists \forall \rightarrow p.\text{conj} \land \mathcal{T}_{fo}(p.\text{existsProc})
| OR   \rightarrow FOL(\text{reduce } \lor, (\text{map } \mathcal{T}_{fo}, p.\text{disjs}).\text{form})
```
always translate the counterexample encoding formula to FOL

\[
S(p_{\text{cand}} \land \mathcal{T}(f_{\text{cex}})) \\
\downarrow \\
S(p_{\text{cand}} \land \mathcal{T}_{\text{fo}}(f_{\text{cex}}))
\]

apply the same idea of flipping $\forall$ to $\exists$ to implement $\mathcal{T}_{\text{fo}}$

// $\mathcal{T}_{\text{fo}} : \text{Formula} \to \text{FOL}$

let $\mathcal{T}_{\text{fo}}(f) = \text{match } p = \mathcal{T}(f) \text{ with}$

<table>
<thead>
<tr>
<th>FOL \to p</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\exists \forall \to p.\text{conj} \land \mathcal{T}_{\text{fo}}(p.\text{existsProc})$</td>
</tr>
<tr>
<td>OR \to FOL(reduce $\lor$, (map $\mathcal{T}_{\text{fo}}$, p.disjs).form)</td>
</tr>
</tbody>
</table>

$\mathcal{T}_{\text{fo}}$ produces strictly less constrained encoding
First-Order Increments Semantics

- always translate the counterexample encoding formula to FOL
  \[
  S(p_{cand} \land T(f_{cex})) \\
  \downarrow \\
  S(p_{cand} \land T_{fo}(f_{cex}))
  \]

- apply the same idea of flipping $\forall$ to $\exists$ to implement $T_{fo}$
  ```
  // $T_{fo}$ : Formula → FOL
  let $T_{fo}(f) = \text{match } p = T(f) \text{ with}
                     | FOL → p
                     | $\exists$∀ → $p$.conj ∧ $T_{fo}(p\.existsProc)$
                     | $\lor$ → FOL(reduce $\lor$, (map $T_{fo}$, $p\.disjs$.form))
  ```

- $T_{fo}$ produces strictly less constrained encoding

- potential trade-off:
  - efficient incremental solving vs.
  - more CEGIS iterations (due to weaker encoding)