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**

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

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

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

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

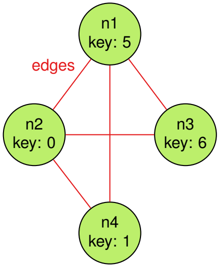
ALLOY*

- capable of automatically solving arbitrary higher-order formulas

First-Order Vs. Higher-Order: `clique`

first-order: finding a graph and a clique in it

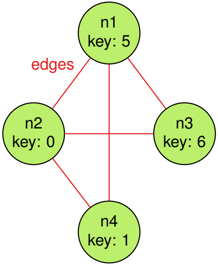
- every two nodes in a clique must be connected



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

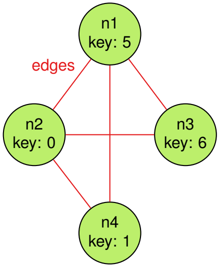



```
sig Node { key: one Int }
```

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

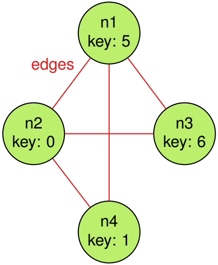



```
sig Node { key: one Int }

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

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

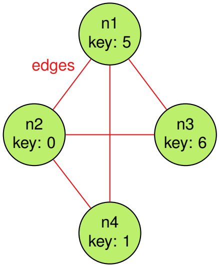



```
sig Node { key: one Int }

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
}
```

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



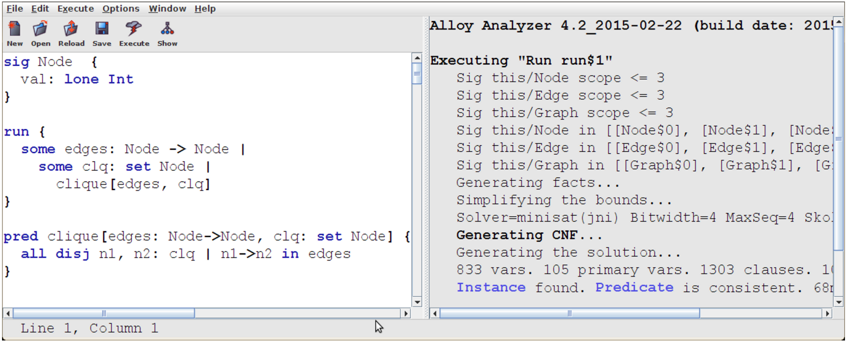

```
sig Node { key: one Int }

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



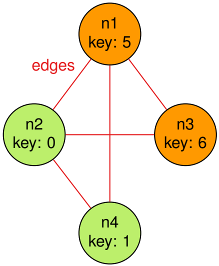
- **Alloy Analyzer**: automatic, bounded, relational constraint solver

- a **solution** (automatically found by Alloy): **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
}
```

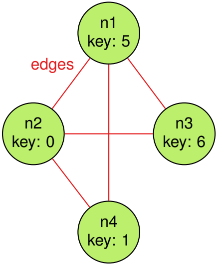
- **Alloy Analyzer**: automatic, bounded, relational constraint solver

- a **solution** (automatically found by Alloy): **clqNodes** = $\{n_1, n_3\}$

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

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

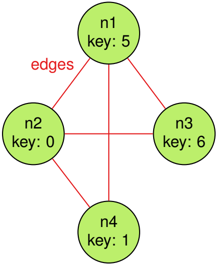
- there is no other clique with more nodes



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

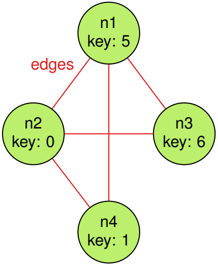



```
pred maxClique[edges: Node->Node, clqNodes: set Node] {
  clique[edges, clqNodes]
  all ns: set Node |
    not (clique[edges, ns] and #ns > #clqNodes)
}
```

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



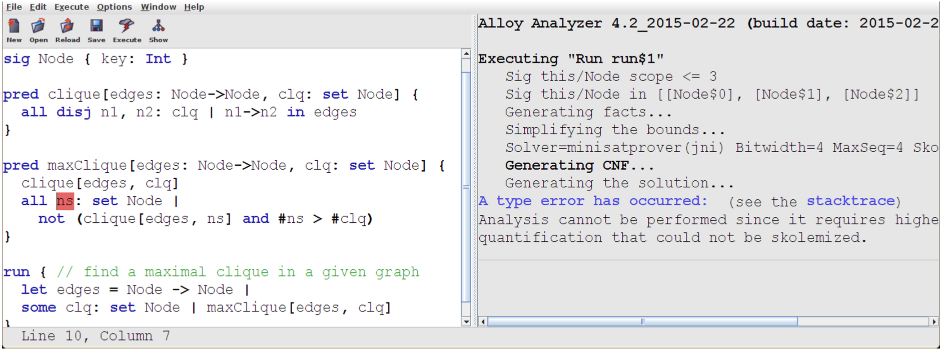

```
pred maxClique[edges: Node->Node, clqNodes: set Node] {
  clique[edges, clqNodes]
  all ns: set Node |
    not (clique[edges, ns] and #ns > #clqNodes)
}

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 <u>not solvable</u> in Alloy!

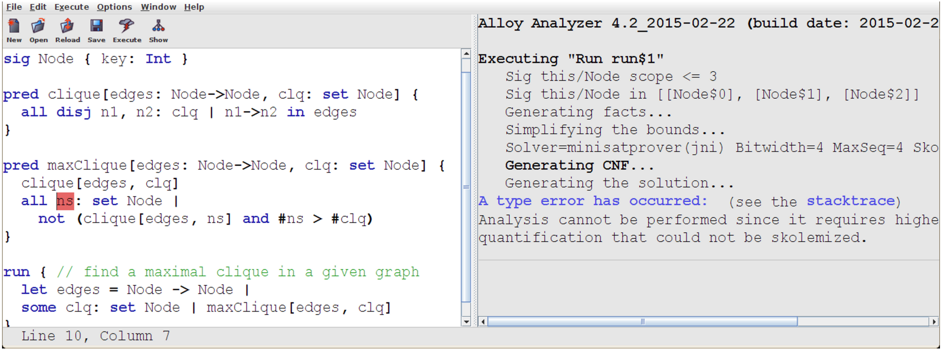


First-Order Vs. **Higher**-Order: `maxClique`

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

- there is no other clique with more nodes

expressible but <u>not solvable</u> in Alloy!



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

Solving `maxClique` Vs. Program **Synthesis**

program synthesis	**maxClique**
find <u>some</u> program AST s.t., for <u>all</u> possible values of its inputs its specification holds	find <u>some</u> set of nodes s.t., it is a clique and for <u>all</u> possible other sets of nodes not one is a larger clique

`some program: ASTNode |`
`all env: Var -> Val |`
`spec[program, env]`
	`some clq: set Node |`
	`clique[clq] and`
	`all ns: set Node |`
	`not (clique[ns] and #ns > #clq)`

Solving `maxClique` Vs. Program **Synthesis**

program synthesis	**maxClique**
find some program AST s.t., for all possible values of its inputs its specification holds	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

`some program: ASTNode |`
`  all env: Var -> Val |`
`    spec[program, env]`
	`some clq: set Node |`
	`  clique[clq] and`
	`  all ns: set Node |`
	`    not (clique[ns] and #ns > #clq)`

similarities:

- the same **`some`**/**`all`** (∃∀) pattern
- the **`all`** quantifier is higher-order

Solving `maxClique` Vs. Program **Synthesis**

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

```
some program: ASTNode |
  all env: Var -> Val |
    spec[program, env]
```

```
some clq: set Node |
  clique[clq] and
  all ns: set Node |
    not (clique[ns] and #ns > #clq)
```

similarities:

- the same **`some`**/**`all`** (∃∀) pattern
- the **`all`** quantifier is higher-order

how do existing program synthesizers work?

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]

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`

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.

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.

ALLOY*

ALLOY* key insight

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

ALLOY*

generality

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

ALLOY*

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

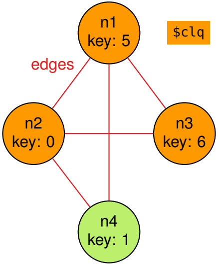
- key solver features for efficient implementation:
  - *partial instances*
  - *incremental solving*

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

```
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
```




```
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]   searching for next candidate  (increment)
   [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]   searching for next candidate  (increment)
   [Some4All]  candidate found (candidate)
   [Some4All]   verifying candidate (condition, pi) success (#cand = 3)
                 |- [OR] solving splits (formula)
                 |- [OR]   trying choice (formula, bounds) unsat
                 |- [OR]   trying choice (formula, bounds) unsat
                 |-   [Some4All] started (formula, bounds)
   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 (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
```

```
                                    ⋮
[Some4All]   searching for next candidate  (increment)
[Some4All]  candidate found (candidate)
[Some4All]   verifying candidate (condition, pi) counterexample
             |- [Some4All] started (formula, bounds)
             |- [Some4All]  candidate found (candidate)
             |- [Some4All]   verifying candidate (condition, pi) success (#cand = 1)
[Some4All]   searching for next candidate  (increment)
[Some4All]  candidate found (candidate)
[Some4All]   verifying candidate (condition, pi) counterexample
             |- [Some4All] started (formula, bounds)
             |- [Some4All]  candidate found (candidate)
             |- [Some4All]   verifying candidate (condition, pi) success (#cand = 1)
[Some4All]   searching for next candidate  (increment)
[Some4All]  candidate found (candidate)
[Some4All]   verifying candidate (condition, pi) counterexample
             |- [Some4All] started (formula, bounds)
             |- [Some4All]  candidate found (candidate)
             |- [Some4All]   verifying candidate (condition, pi) success (#cand = 1)
[Some4All]   searching for next candidate  (increment)
[Some4All]  candidate found (candidate)
[Some4All]   verifying candidate (condition, pi) counterexample
             |- [Some4All] started (formula, bounds)
             |- [Some4All]  candidate found (candidate)
             |- [Some4All]   verifying candidate (condition, pi) success (#cand = 1)
[Some4All]   searching for next candidate  (increment)
[Some4All]  candidate found (candidate)
[Some4All]   verifying candidate (condition, pi) counterexample
             |- [Some4All] started (formula, bounds)
             |- [Some4All]  candidate found (candidate)
             |- [Some4All]   verifying candidate (condition, pi) success (#cand = 1)
[Some4All]   searching for next candidate  (increment)
[Some4All]  candidate found (candidate)
[Some4All]   verifying candidate (condition, pi) counterexample
             |- [Some4All] started (formula, bounds)
             |- [Some4All]  candidate found (candidate)
             |- [Some4All]   verifying candidate (condition, pi) success (#cand = 1)
[Some4All]   searching for next candidate  (increment)
[Some4All]  candidate found (candidate)
[Some4All]   verifying candidate (condition, pi) counterexample
             |- [Some4All] started (formula, bounds)
             |- [Some4All]  candidate found (candidate)
             |- [Some4All]   verifying candidate (condition, pi) success (#cand = 1)
[Some4All]   searching for next candidate  (increment)
No counterexample found. Assertion may be valid. 91365ms.
```

Semantics: General Idea

- 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
     - → `FOL` : first-order formula
     - → `OR` : disjunction
     - → `∃∀` : higher-order top-level $\forall$ quantifier (not skolemizable)

Semantics: General **Idea**

- CEGIS: defined only for a single idiom (the ∃∀ formula pattern)
- ALLOY*: generalized to arbitrary formulas
  1. perform standard transformation: NNF and skolemization
  2. decompose arbitrary formula into known idioms
     - → `FOL` : first-order formula
     - → `OR` : disjunction
     - → `∃∀` : higher-order top-level ∀ quantifier (not skolemizable)
  3. solve using the following decision procedure
     - → `FOL` : solve directly with Kodkod (first-order relational solver)
     - → `OR` : solve each disjunct separately
     - → `∃∀` : apply CEGIS

ALLOY* Implementation **Caveats**

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

→

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

ALLOY* Implementation **Caveats**

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

→

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

1. candidate search

- `solve` $conj \wedge eQuant$

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

ALLOY* Implementation **Caveats**

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

→

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

1. candidate search

- `solve` $conj \wedge eQuant$

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

2. verification

- `solve` $\neg aQuant$ against the `$cand` *partial instance*

→ *counterexample* `$cex`: value of the $eQuant.var$ relation

ALLOY* Implementation **Caveats**

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

→

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

1. candidate search

- `solve` $conj \wedge eQuant$

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

2. verification

- `solve` $\neg aQuant$ 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**

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

→

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

1. candidate search

- `solve` $conj \wedge eQuant$

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

2. verification

- `solve` $\neg aQuant$ against the `$cand` *partial instance*

→ *counterexample* `$cex`: value of the $eQuant.var$ relation

partial instance
- partial solution known upfront
- enforced using *bounds*

3. induction

- use *incremental solving* to add

  replace $eQuant.var$ **with** `$cex` **in** $eQuant.body$

  to previous search condition

ALLOY* Implementation **Caveats**

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

→

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

1. candidate search

- `solve` $conj \wedge eQuant$

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

2. verification

- `solve` $\neg aQuant$ against the `$cand` *partial instance*

→ *counterexample* `$cex`: value of the $eQuant.var$ relation

partial instance
- partial solution known upfront
- enforced using *bounds*

3. induction

- use *incremental solving* to add

  `replace` $eQuant.var$ **`with`** `$cex` **`in`** $eQuant.body$

  to previous search condition

incremental solving
- continue from prev solver instance
- the solver reuses learned clauses

ALLOY* Implementation **Caveats**

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

→

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

1. candidate search

- `solve` $conj \wedge eQuant$

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

2. verification

- `solve` $\neg aQuant$ against the `$cand` *partial instance*

→ *counterexample* `$cex`: value of the $eQuant.var$ relation

partial instance
- partial solution known upfront
- enforced using *bounds*

3. induction

- use *incremental solving* to add

  `replace` $eQuant.var$ **`with`** `$cex` **`in`** $eQuant.body$

  to previous search condition

incremental solving
- continue from prev solver instance
- the solver reuses learned clauses

? *what if the increment formula is not first-order*

– optimization 1: use its weaker “first-order version”

ALLOY* **Optimization**

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"

ALLOY* **Optimization**

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:

```
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!

✗

ALLOY* **Optimization**

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!

✔

ALLOY* Evaluation

evaluation goals

ALLOY* Evaluation

evaluation goals

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

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

ALLOY* Evaluation

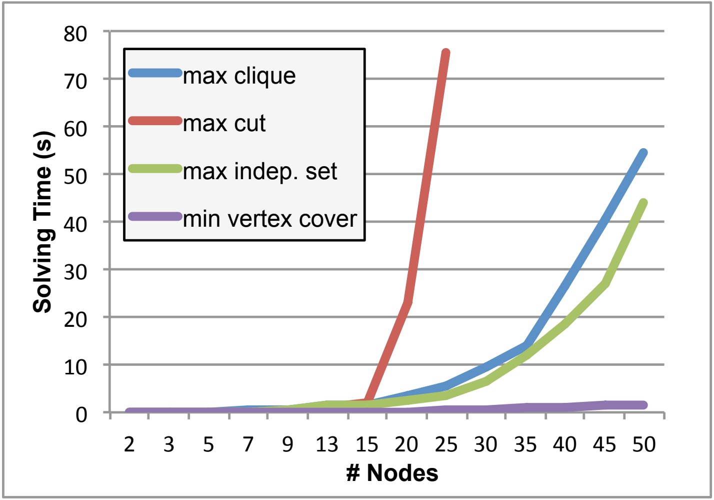
evaluation goals

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



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

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

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

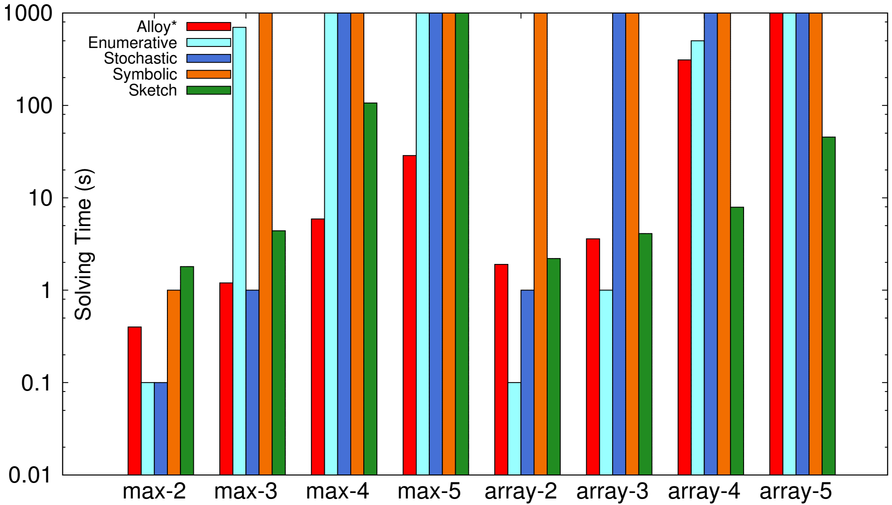
- ALLOY* was able to solve **all** different **categories** of benchmarks
  - integer benchmarks, bit vector benchmarks, let constructs, synthesizing multiple functions at once, multiple applications of the synthesized function

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)



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)

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

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

```
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

```
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
```

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

```
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

```
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 ALLOY* **Optimizations**

	base	w/ optimizations
`max2`	0.4s	0.3s
`max3`	7.6s	0.9s
`max4`	t/o	1.5s
`max5`	t/o	4.2s
`max6`	t/o	16.3s
`max7`	t/o	163.6s
`max8`	t/o	987.3s
`array-search2`	140.0s	1.6s
`array-search3`	t/o	4.0s
`array-search4`	t/o	16.1s
`array-search5`	t/o	485.6s

	base	w/ optimizations
`turan5`	3.5s	0.5s
`turan6`	12.8s	2.1s
`turan7`	235.0s	3.8s
`turan8`	t/o	15.0s
`turan9`	t/o	45.0s
`turan10`	t/o	168.0s

ALLOY* Conclusion

ALLOY* is

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

ALLOY* 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, ...

ALLOY* 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, ...

ALLOY* **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, ...

Thank You!

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

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, ...

First-Order Vs. Higher-Order: `clique`

first-order: finding a clique in a graph

First-Order Vs. Higher-Order: `clique`

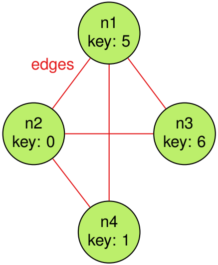
first-order: finding a clique in a graph

```
pred clique[edges: Node->Node, clq: set Node] {
  all disj n1, n2: clq | n1->n2 in edges // every two nodes in ’clq’ are connected
}
```

First-Order Vs. Higher-Order: clique

first-order: finding a clique in a graph

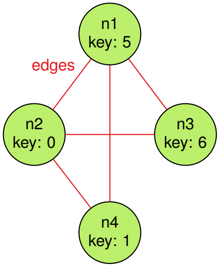
```
pred clique[edges: Node->Node, clq: set Node] {
  all disj n1, n2: clq | n1->n2 in edges // every two nodes in 'clq' are connected
}
run {  // find a clique in a given graph
  let edges = n1->n2 + n1->n3 + ... |
  some clq: set Node | clique[edges, clq]
}
```



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]
}
```



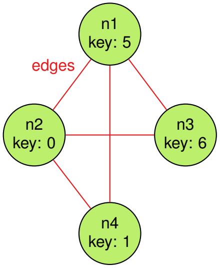
Alloy encoding:

N1: $\{n_1\}$ | **N2**: $\{n_2\}$ | **N3**: $\{n_3\}$ | **N4**: $\{n_4\}$ — **atoms**

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:

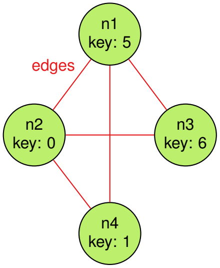
N1: $\{n_1\}$ | **N2**: $\{n_2\}$ | **N3**: $\{n_3\}$ | **N4**: $\{n_4\}$ — **atoms**

Node: $\{n_1, n_2, n_3, n_4\}$
key: $\{(n_1 \to 5), (n_2 \to 0), (n_3 \to 6), (n_4 \to 1)\}$
edges: $\{(n_1 \to n_2), (n_1 \to n_3), (n_1 \to n_4), (n_2 \to n_3), (n_2 \to n_4), (n_2 \to n_1), (n_3 \to n_1), (n_4 \to n_1), (n_3 \to n_2), (n_4 \to n_2)\}$
— **fixed relations**

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:

N1: $\{n_1\}$ | **N2**: $\{n_2\}$ | **N3**: $\{n_3\}$ | **N4**: $\{n_4\}$ — **atoms**

Node: $\{n_1, n_2, n_3, n_4\}$
key: $\{(n_1 \to 5), (n_2 \to 0), (n_3 \to 6), (n_4 \to 1)\}$
edges: $\{(n_1 \to n_2), (n_1 \to n_3), (n_1 \to n_4), (n_2 \to n_3), (n_2 \to n_4),$
$(n_2 \to n_1), (n_3 \to n_1), (n_4 \to n_1), (n_3 \to n_2), (n_4 \to n_2)\}$
— **fixed relations**

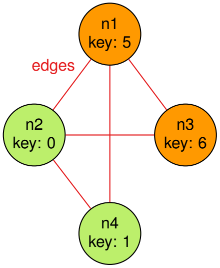
clq: $\{\}$, $\{n_1, n_2, n_3, n_4\}$ — **relations to be solved**

lower bound, upper bound → **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

```
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:

N1: $\{n_1\}$ | **N2**: $\{n_2\}$ | **N3**: $\{n_3\}$ | **N4**: $\{n_4\}$ — **atoms**

Node: $\{n_1, n_2, n_3, n_4\}$
key: $\{(n_1 \to 5), (n_2 \to 0), (n_3 \to 6), (n_4 \to 1)\}$
edges: $\{(n_1 \to n_2), (n_1 \to n_3), (n_1 \to n_4), (n_2 \to n_3), (n_2 \to n_4), (n_2 \to n_1), (n_3 \to n_1), (n_4 \to n_1), (n_3 \to n_2), (n_4 \to n_2)\}$ — **fixed relations**

clq: $\{\}$, $\{n_1, n_2, n_3, n_4\}$ — **relations to be solved**

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

- a **solution** (automatically found by Alloy): **clq** = $\{n_1, n_3\}$

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

higher-order: finding a maximal clique in a graph

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

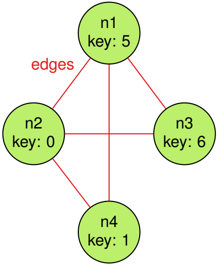
higher-order: finding a maximal clique in a graph

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

First-Order Vs. Higher-Order: maxClique

higher-order: finding a maximal clique in a graph

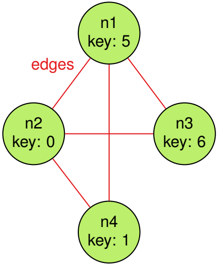
```
pred maxClique[edges: Node->Node,  clq: set Node] {
  clique[edges, clq]
  all ns: set Node |
    not (clique[edges, ns] and #ns > #clq)
}
run {  // find a maximal clique in a given graph
  let edges = n1->n2 + n1->n3 + ... |
  some clq: set Node | maxClique[edges, clq]
}
```



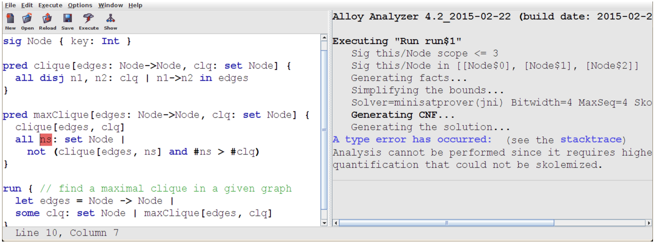
First-Order Vs. **Higher**-Order: `maxClique`

higher-order: finding a maximal clique in a graph

```
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]
}
```



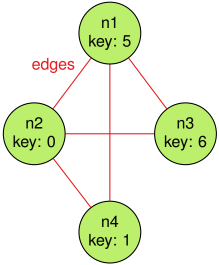
expressible but not solvable in Alloy!



First-Order Vs. Higher-Order: maxClique

higher-order: finding a maximal clique in a graph

```
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]
}
```

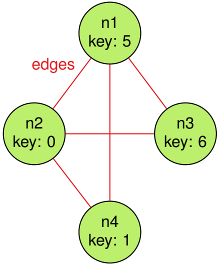


- **definition** of higher-order (as in Alloy):
  - quantification over all sets of atoms

First-Order Vs. Higher-Order: maxClique

higher-order: finding a maximal clique in a graph

```
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]
}
```

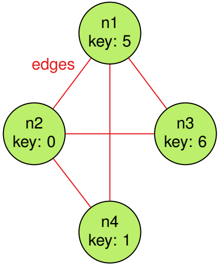


- **definition** of higher-order (as in Alloy):
  - quantification over all sets of atoms
- `maxClique`: check all possible sets of nodes and ensure not one is a clique larger than `clq`

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

higher-order: finding a maximal clique in a graph

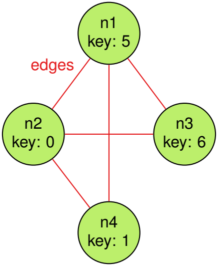
```
pred maxClique[edges: Node->Node,  clq: set Node] {
  clique[edges, clq]
  all ns: set Node |
    not (clique[edges, ns] and #ns > #clq)
}
run {  // find a maximal clique in a given graph
  let edges = n1->n2 + n1->n3 + ... |
  some clq: set Node | maxClique[edges, clq]
}
```



- **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: $2^{\#Node}$

Solving `maxClique`: **Idea**

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

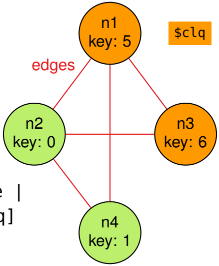


intuitive iterative algorithm

Solving `maxClique`: **Idea**

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

```
some clq: Set Node |
 clique[edges, clq]
```



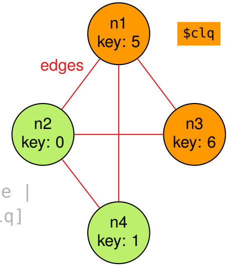
intuitive iterative algorithm

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)
}
```

```
some clq: Set Node |
 clique[edges, clq]
```




```
some ns: Set Node |
 clique[edges, ns] and #ns > 2
```

intuitive iterative algorithm

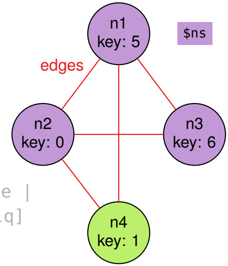
1. find <u>some</u> clique `$clq`

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

Solving `maxClique`: **Idea**

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

```
some clq: Set Node |
 clique[edges, clq]
```



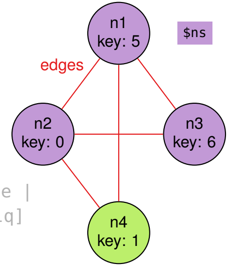

```
some ns: Set Node |
 clique[edges, ns] and #ns > 2
```

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 `maxClique`: **Idea**

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




```
some clq: Set Node |
 clique[edges, clq]
```

```
some ns: Set Node |
 clique[edges, ns] and #ns > 2
```

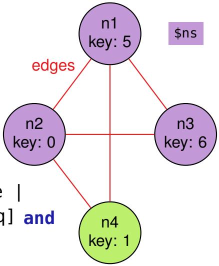
intuitive iterative algorithm

1. find <u>some</u> clique `$clq`
2. check if `$clq` is maximal
   ⇔ find <u>some</u> 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)
}
```

```
some clq: Set Node |
 clique[edges, clq] and
 #clq >= 3
```



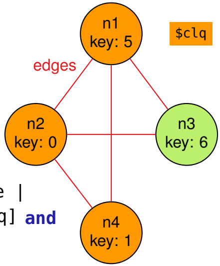
intuitive iterative algorithm

1. find <u>some</u> clique `$clq`
2. check if `$clq` is maximal
   ⇔ find <u>some</u> 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)
}
```

```
some clq: Set Node |
 clique[edges, clq] and
 #clq >= 3
```



intuitive iterative algorithm

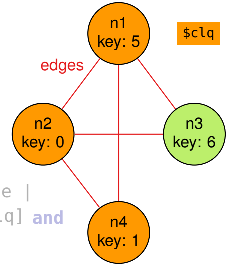
1. find <u>some</u> clique `$clq`

2. check if `$clq` is maximal
   ⇔ find <u>some</u> clique `$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)
}
```




```
some clq: Set Node |
 clique[edges, clq] and
 #clq >= 3
```

```
some ns: Set Node |
 clique[edges, ns] and #ns > 3
```

intuitive iterative algorithm

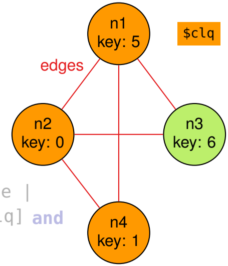
1. find <u>some</u> clique `$clq`

2. check if `$clq` is maximal
   ⇔ find <u>some</u> clique `$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)
}
```




```
some clq: Set Node |
 clique[edges, clq] and
 #clq >= 3
```

```
some ns: Set Node |
 clique[edges, ns] and #ns > 3
```

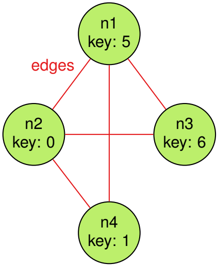
UNSAT ⟶ `return $clq`

intuitive iterative algorithm

1. find <u>some</u> clique `$clq`
2. check if `$clq` is maximal
   ⇔ find <u>some</u> 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**

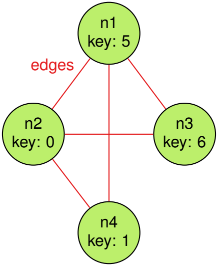
```
run {
  some clq: set Node |
    clique[edges, clq] and
    all ns: set Node |
      not (clique[edges, ns] and #ns > #clq)
}
```



intuitive iterative algorithm

Solving `maxClique`: **Idea**

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

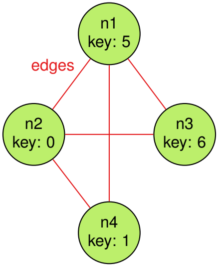


intuitive iterative algorithm

1. find some clique `$clq`

Solving `maxClique`: **Idea**

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

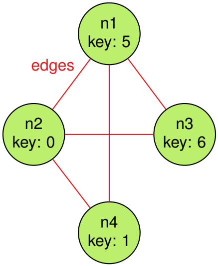


intuitive iterative algorithm

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 `maxClique`: Idea

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

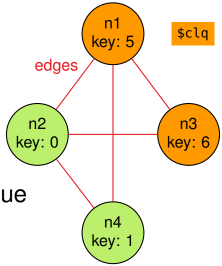


intuitive iterative algorithm

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

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`

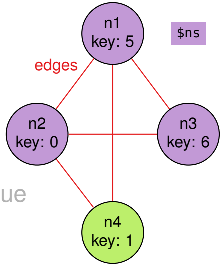
2. check if `$clq` is maximal
   ⇔ find some clique `$ns` > `$clq` from step 1
   – if not found: return `$clq`

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)
}
```

find **candidate** clique



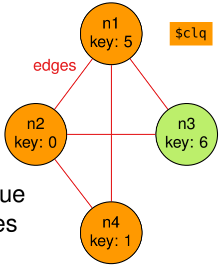
verify `$clq` (is it maximal?)
→ counterexample: `$ns`

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)
}
```



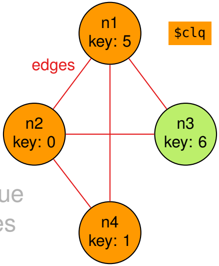
find **candidate** clique
with at least 3 nodes

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)
}
```



find **candidate** clique
with at least 3 nodes

verify `$clq` (is it maximal?)
UNSAT ⟶ `return $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

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]

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`

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.

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*

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 **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

```
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
}
```

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]
}
```

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
}
```

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])
}
```

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]
```

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]
```

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

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

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

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

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

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

```
// converted to Proc
∃∀(conj:   $env in Node -> Int,
    // used for search
   eQuant: some eval  ...,
    // used for verification
   aQuant: all eval  ...)
```

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

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

```
// converted to Proc
∃∀(conj:   $env in Node -> Int,
    // used for search
   eQuant: some eval ...,
    // used for verification
   aQuant: all eval ...)
```

3. induction

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

- body of *aQuant* from step 1 with `env` replaced by the concrete value (`$env_cex`) from step 2
- implemented using “incremental solving”

Semantics: General Idea

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

Semantics: General **Idea**

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

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

Semantics: General **Idea**

1. convert formula to Negation Normal Form (NNF)
   - → boolean connectives left: $\wedge$, $\vee$, $\neg$
   - → 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 `∃∀` nodes
   - → `FOL` : first-order formula
   - → `OR` : disjunction
   - → `∃∀` : higher-order top-level $\forall$ quantifier (not skolemizable)

Semantics: General Idea

1. convert formula to Negation Normal Form (NNF)
   - → boolean connectives left: $\wedge$, $\vee$, $\neg$
   - → 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 `∃∀` nodes
   - → `FOL` : first-order formula
   - → `OR` : disjunction
   - → `∃∀` : 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
   - → `∃∀` : apply CEGIS

Semantics: Formula **Decomposition**

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

Semantics: Formula **Decomposition**

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

$\mathcal{T}$: Formula → Proc // translates arbitrary formula to a tree of Procs

let $\mathcal{T}$ = $\lambda(f)\cdot$

Semantics: Formula **Decomposition**

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

$\mathcal{T}$: Formula → Proc // translates arbitrary formula to a tree of Procs

let $\mathcal{T} = \lambda(f)\cdot$

  let $f_{nnf} = skolemize(nnf(f))$

- convert to NNF and skolemize

Semantics: Formula **Decomposition**

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

$\mathcal{T}$: Formula → Proc // translates arbitrary formula to a tree of Procs

let $\mathcal{T} = \lambda(f) \cdot$
  let $f_{nnf} = skolemize(nnf(f))$
  match f_{nnf} **with**
  | $\neg f_s \rightarrow$ FOL(f_{nnf})

translating negation

- negation can be only in leaves

⇒ must be first-order

Semantics: Formula **Decomposition**

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

$\mathcal{T}$: Formula → Proc // translates arbitrary formula to a tree of Procs

let $\mathcal{T} = \lambda(f) \cdot$

let $f_{nnf} = skolemize(nnf(f))$

match f_{nnf} **with**

| $\neg f_s \rightarrow$ FOL(f_{nnf})

| $\exists x \cdot f_s \rightarrow$ ***fail*** *"can't happen"*

translating the ∃ quantifier

- there can't be top-level ∃ quantifiers after skolemization

Semantics: Formula **Decomposition**

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

$\mathcal{T}$: Formula → Proc // translates arbitrary formula to a tree of Procs

```
let 𝒯 = λ(f) ·
  let f_nnf = skolemize(nnf(f))
  match f_nnf with
  | ¬f_s     → FOL(f_nnf)
  | ∃x · f_s → fail "can't happen"
  | ∀x · f_s → let p = 𝒯(∃x · f_s)
                 if (x.mult = SET) || ¬(p is FOL)
                   ∃∀(FOL(true), f_nnf, p)
                 else
                   FOL(f_nnf)
```

translating the ∀ quantifier

- translate the dual ∃ formula first (where the ∃ quantifier will be skolemizable)
- if multiplicity of this ∀ quantifier is SET or the dual is **not** first-order
  - then: f_{nnf} is higher-order → create ∃∀ node
  - else: f_{nnf} is first-order → create FOL node

Semantics: Formula **Decomposition**

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

$\mathcal{T}$: Formula → Proc // translates arbitrary formula to a tree of Procs

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

translating disjunction

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

Semantics: Formula **Decomposition**

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

$\mathcal{T}$: Formula → Proc // translates arbitrary formula to a tree of Procs

---

let $\mathcal{T} = \lambda(f)\cdot$

  let $f_{nnf} = skolemize(nnf(f))$

  match f_{nnf} **with**

  | $\neg f_s \rightarrow$ FOL(f_{nnf})

  | $\exists x \cdot f_s \rightarrow$ ***fail*** *"can't happen"*

  | $\forall x \cdot f_s \rightarrow$ **let** $p = \mathcal{T}(\exists x \cdot f_s)$

    if $(x.mult$ = SET$) \,||\, \neg(p$ **is** FOL$)$

      ∃∀(FOL(true), f_{nnf}, p)

    else

      FOL(f_{nnf})

  | $f_1 \vee f_2 \rightarrow$ OR([$\mathcal{T}(f_1)$, $\mathcal{T}(f_2)$])

  | $f_1 \wedge f_2 \rightarrow \mathcal{T}(f_1) \curlywedge \mathcal{T}(f_2)$

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**

$\mathcal{S}$: `Proc` → `Instance` **`option`**

`let` $\mathcal{S} = \lambda(p) \cdot$

Semantics: Formula **Evaluation**

$\mathcal{S}$: `Proc` → `Instance` **`option`**

`let` $\mathcal{S}$ = $\lambda(p)\cdot$
  `match` p **`with`**
  | `FOL` → $\textit{solve}\ \ p.\textit{form}$

Semantics: Formula **Evaluation**

$\mathcal{S}$: `Proc` → `Instance` **`option`**

```
let S = λ(p) .
  match p with
  | FOL → solve p.form
  | OR  → ... // apply S to each Proc in p.disj; return the first solution found
```

Semantics: Formula **Evaluation**

$\mathcal{S}$: `Proc` → `Instance` **option**

```
let 𝒮 = λ(p) ·
  match p with
  | FOL → solve p.form
  | OR  → ... // apply 𝒮 to each Proc in p.disj; return the first solution found
  | ∃∀  → let p_cand = p.conj ∧ p.existsProc
          match 𝒮(p_cand) with
          | None        → None // no candidate solution found ⇒ return UNSAT
          | Some(cand)  → // candidate solution found ⇒ proceed to verify the candidate
              match 𝒮(𝒯(¬p.allForm)) with // try to falsify cand ⇒ must run 𝒮 against the cand instance
              | None       → Some(cand) // no counterexample found ⇒ cand is the solution
              | Some(cex)  → let q = p.allForm
                             // encode the counterexample as a formula: use only the body of the ∀ quant.
                             // in which the quant. variable is replaced with its concrete value in cex
                             let f_cex = replace(q.body, q.var, eval(cex, q.var))
                               // add the counterexample encoding to the candidate search condition
                               𝒮(p_cand ∧ 𝒯(f_cex))
```

Semantics: Formula **Evaluation**

$\mathcal{S}$: `Proc` → `Instance` **option**

---

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

partial instance
encode $cand$ as partial instance

Semantics: Formula **Evaluation**

```
𝒮 : Proc → Instance option
let 𝒮 = λ(p) ·
  match p with
  | FOL → solve p.form
  | OR  → ... // apply 𝒮 to each Proc in p.disj; return the first solution found
  | ∃∀  → let p_cand = p.conj ∧ p.existsProc
          match 𝒮(p_cand) with
          | None        → None // no candidate solution found ⇒ return UNSAT
          | Some(cand)  → // candidate solution found ⇒ proceed to verify the candidate
            match 𝒮(𝒯(¬p.allForm)) with // try to falsify cand ⇒ must run 𝒮 against the cand instance
            | None      → Some(cand) // no counterexample found ⇒ cand is the solution
            | Some(cex) → let q = p.allForm
                          // encode the counterexample as a formula: use only the body of the ∀ quant.
                          // in which the quant. variable is replaced with its concrete value in cex
                          let f_cex = replace(q.body, q.var, eval(cex, q.var))
                           // add the counterexample encoding to the candidate search condition
                           𝒮(p_cand ∧ 𝒯(f_cex))
```

partial instance
encode *cand* as partial instance

counterexample encoding
no domain-specific knowledge necessary

Semantics: Formula **Evaluation**

$\mathcal{S}$: `Proc` → `Instance` **option**

```
let S = λ(p) ·
  match p with
  | FOL → solve p.form
  | OR  → ... // apply S to each Proc in p.disj; return the first solution found
  | ∃∀  → let p_cand = p.conj ∧ p.existsProc
          match S(p_cand) with
          | None        → None // no candidate solution found ⇒ return UNSAT
          | Some(cand)  → // candidate solution found ⇒ proceed to verify the candidate
            match S(T(¬p.allForm)) with // try to falsify cand ⇒ must run S against the cand instance
            | None       → Some(cand) // no counterexample found ⇒ cand is the solution
            | Some(cex)  → let q = p.allForm
                 // encode the counterexample as a formula: use only the body of the ∀ quant.
                 // in which the quant. variable is replaced with its concrete value in cex
                 let f_cex = replace(q.body, q.var, eval(cex, q.var))
                  // add the counterexample encoding to the candidate search condition
                  S(p_cand ∧ T(f_cex))
```

partial instance
encode *cand* as partial instance

counterexample encoding
no domain-specific knowledge necessary

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

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

```
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!

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

```
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"*

solution: add new syntax for domain constraints

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

Domain Constraints Semantics

first-order logic semantics

```
all x: X when dom[x] | body[x]   ⟺  all x: X | dom[x] implies body[x]
some x: X when dom[x] | body[x]  ⟺  some x: X | dom[x] and body[x]
```

Domain Constraints Semantics

first-order logic semantics

```
all x: X when dom[x] | body[x]    ⟺  all x: X | dom[x] implies body[x]
some x: X when dom[x] | body[x]   ⟺  some x: X | dom[x] and body[x]
```

De Morgan's Laws (consistent with classical logic)

```
not (all x: X when dom[x] | body[x])    ⟺  some x: X when dom[x] | not body[x]
not (some x: X when dom[x] | body[x])   ⟺  all x: X when dom[x] | not body[x]
```

Domain Constraints Semantics

first-order logic semantics

```
all x: X when dom[x] | body[x]    ⟺  all x: X | dom[x] implies body[x]
some x: X when dom[x] | body[x]   ⟺  some x: X | dom[x] and body[x]
```

De Morgan's Laws (consistent with classical logic)

```
not (all x: X when dom[x] | body[x])    ⟺  some x: X when dom[x] | not body[x]
not (some x: X when dom[x] | body[x])   ⟺  all x: X when dom[x] | not body[x]
```

changes to the ALLOY* semantics

- converting higher-order ∀ to ∃: $\forall x \cdot f \;\rightarrow\; \exists x \cdot f$ (domain constraints stay with x)
- encoding a counterexample as a formula: in

$$\texttt{let}\; f_{cex} = replace(q.body,\; q.var,\; eval(cex,\; q.var))$$

$q.body$ is expanded according to the first-order semantics above

Optimization 2: **First-Order Increments**

problem: search space too big, counterexamples not focused

```
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

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

→

- quantifies over evaluations of `Node`s instead of only `Var`s
- counterexamples encode entire `eval` relation, instead of only values of variables

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 `Node`s instead of only `Var`s
- 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]
}
```

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 `Node`s instead of only `Var`s
- 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

```
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])
```

⇔

```
some env: Var -> one Int |
  all eval: Node -> (Int+Bool)
  when env in eval && semantics[eval] |
    not 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 `Node`s instead of only `Var`s
- 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])
```

⇔

```
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

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

→

- quantifies over evaluations of `Node`s instead of only `Var`s
- 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])
```

⇔

```
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

First-Order Increments Semantics

- always translate the counterexample encoding formula to `FOL`

$$\mathcal{S}(p_{cand} \wedge \mathcal{T}(f_{cex}))$$
$$\downarrow$$
$$\mathcal{S}(p_{cand} \wedge \mathcal{T}_{\texttt{fo}}(f_{cex}))$$

First-Order Increments Semantics

- always translate the counterexample encoding formula to FOL

$$\mathcal{S}(p_{cand} \wedge \mathcal{T}(f_{cex}))$$
$$\downarrow$$
$$\mathcal{S}(p_{cand} \wedge \mathcal{T}_{\mathtt{fo}}(f_{cex}))$$

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

```
// T_fo : Formula → FOL
let T_fo(f) = match p = T(f) with
              | FOL → p
              | ∃∀  → p.conj ∧ T_fo(p.existsProc)
              | OR  → FOL(reduce ∨, (map T_fo, p.disjs).form)
```

First-Order Increments Semantics

- always translate the counterexample encoding formula to FOL

$$\mathcal{S}(p_{cand} \wedge \mathcal{T}(f_{cex}))$$
$$\downarrow$$
$$\mathcal{S}(p_{cand} \wedge \mathcal{T}_{\texttt{fo}}(f_{cex}))$$

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

```
// T_fo : Formula → FOL
let T_fo(f) = match p = T(f) with
              | FOL → p
              | ∃∀  → p.conj ∧ T_fo(p.existsProc)
              | OR  → FOL(reduce ∨, (map T_fo, p.disjs).form)
```

- $\mathcal{T}_{\texttt{fo}}$ produces strictly less constrained encoding

First-Order Increments Semantics

- always translate the counterexample encoding formula to FOL

$$\mathcal{S}(p_{cand} \wedge \mathcal{T}(f_{cex}))$$
$$\downarrow$$
$$\mathcal{S}(p_{cand} \wedge \mathcal{T}_{\mathtt{fo}}(f_{cex}))$$

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

```
// T_fo : Formula → FOL
let T_fo(f) = match p = T(f) with
              | FOL → p
              | ∃∀  → p.conj ∧ T_fo(p.existsProc)
              | OR  → FOL(reduce ∨, (map T_fo, p.disjs).form)
```

- $\mathcal{T}_{\mathtt{fo}}$ produces strictly less constrained encoding
- potential trade-off:
  - efficient incremental solving vs.
  - more CEGIS iterations (due to weaker encoding)