Deductive Synthesis

Programs vs. Proofs

Closing Arguments
A bullet we’ve missed

Constraint-based synthesis
- How to solve constraints about infinitely many inputs? CEGIS
- How to encode semantics of looping / recursive programs?
  - Bounded reasoning
  - Unbounded / deductive reasoning

Enumerative (and deductive) synthesis
- How to use deductive reasoning to guide the search?
method SumMax (a: array<int>)
    returns (sum: int, max: int)
{
    sum, max := 0, 0;
    var i := 0;
    while (i < a.Length)
    {  
        if (max < a[i]) { max := a[i]; }  
        sum := sum + a[i];  
        i := i + 1;
    }
}
Invariant inference

method SumMax (a: array<int>)
  returns (sum: int, max: int)
  requires a != null;
  ensures sum <= a.Length * max;
  {
    sum, max := 0, 0;
    var i := 0;
    while (i < a.Length)
      invariant ??;
      decreases ??;
      {
        if (max < a[i]) { max := a[i]; }
        sum := sum + a[i];
        i := i + 1;
      }
  }

BLAST
Astrée
FB Infer
LiquidHaskell
SeaHorn
...

correct!

invariant 0 <= i <= a.Length && sum <= i * max;
decreases a.Length - i;

can’t find an invariant that lets me prove correctness
method SumMax (a: array<int>)
  returns (sum: int, max: int)

  requires a != null;
  ensures sum <= a.Length * max;
  
  {  
    var i;  
    ??;  
    while (??)  
      invariant ??;  
      decreases ??;
    
    }  
  }

found a correct program!

  sum, max := 0, 0;
  var i := 0;
  while (i < a.Length)
    invariant 0 <= i <= a.Length & sum <= i * max;
    decreases a.Length - i;
    {  
      if (max < a[i]) { max := a[i]; }  
      sum := sum + a[i];  
      i := i + 1;
    }

VS3
Natural Synthesis
(Leon)
(Synquid)

can’t find a (program, invariant) pair that I can prove correct
From Verification to Synthesis

**Verification**
- **Program logic**: $\forall x. Q(x)$
- **SMT**: $\text{UNSAT}$
- **Result**: $\checkmark$ SAT

**Inference**
- **Program logic**: $\exists I. \forall x. Q(I, x)$
- **Result**: $\text{failure}$

**Synthesis**
- **Program logic**: $\exists P,I. \forall x. Q(P,I,x)$
- **Result**: $\text{failure}$

---

**On the bright side:** not much harder than inference!
How verification works

Verification

∀x. Q(x)
Step 1: Break Down Loops

{\{T\}\{
    sum := 0;
    max := 0;
    i := 0;
    \{i \leq \text{len}(a) \land s \leq i \cdot m\}\}

while (i < a.length)
    \text{invariant}\ i \leq \text{a.length} \land \sum \leq i \cdot \text{max};
    \{i \leq \text{len}(a) \land s \leq i \cdot m \land i < \text{len}(a)\}\{
        if (max < a[i]) max := a[i];
        sum := sum + a[i];
        i := i + 1;
    \}
\{s \leq \text{len}(a) \cdot m\}\{
    \{s \leq \text{len}(a) \cdot m\}\}
Step 2: Construct Transition Relation

\[
\begin{align*}
\{ \top \} & \quad s' = 0 \\
\text{sum} := 0; \quad m' = 0 \\
\text{max} := 0; \quad i' = 0 \\
\text{i := 0;} \quad & \quad \{ i \leq \text{len}(a) \land s \leq i \cdot m \} \\
\end{align*}
\]

\[
\begin{align*}
\{ i \leq \text{len}(a) \land s \leq i \cdot m \land i < \text{len}(a) \} & \quad s' = s + a[i] \\
\text{if (max < a[i]) max := a[i];} & \quad m < a[i] \rightarrow m' = a[i] \\
\text{sum := sum + a[i];} & \quad \neg (m < a[i]) \rightarrow m' = m \\
\text{i := i + 1;} & \quad i' = i + 1 \\
\{ i \leq \text{len}(a) \land s \leq i \cdot m \} & \quad \neg (i < \text{len}(a)) \\
\end{align*}
\]

\[
\begin{align*}
\{ i \leq \text{len}(a) \land s \leq i \cdot m \land \neg (i < \text{len}(a)) \} & \quad s' = s \\
\text{skip} & \quad m' = m \\
\{ s \leq \text{len}(a) \cdot m \} & \quad i' = i \\
\end{align*}
\]
Step 3: Generate VC

\[
\begin{align*}
\forall s, m, i, s', m', i'.
\end{align*}
\]

\[
\begin{align*}
{s'} &= 0 \\
{m'} &= 0 \\
{i'} &= 0
\end{align*}
\]

\[
\begin{align*}
\{T\} \\
\text{sum := 0;} \\
\text{max := 0;} \\
\text{i := 0;} \\
\{i \leq \text{len}(a) \land s \leq i \cdot m\}
\end{align*}
\]

\[
\begin{align*}
\{i \leq \text{len}(a) \land s \leq i \cdot m \land i < \text{len}(a)\} \\
\text{if } (\text{max} < a[i]) \text{ max := } a[i]; \\
\text{sum := sum + a[i];} \\
\text{i := i + 1;} \\
\{i \leq \text{len}(a) \land s \leq i \cdot m\}
\end{align*}
\]

\[
\begin{align*}
\{i \leq \text{len}(a) \land s \leq i \cdot m \land \neg(i < \text{len}(a))\} \\
\text{skip} \\
\{s \leq \text{len}(a) \cdot m\}
\end{align*}
\]
From verification to inference

Verification

\( \forall x. \ Q(x) \)

SMT

UNSAT / SAT

Inference

\( \exists I. \ \forall x. \ Q(I, x) \)
Invariant Inference

$$
\begin{align*}
\{pre\} & \quad \text{init} & \{I\} & \quad \{pre\} \\
\{I \land c\} & \quad \text{body} & \{I\} & \quad \{I \land c\} \\
\{I \land \neg c\} & \quad \text{final} & \{post\} & \quad \{I \land \neg c\} \\
\end{align*}
$$

$$
\begin{align*}
\exists I. \forall x . \\
& \quad \{pre \land init \Rightarrow I'\} \\
& \quad \{I'\} \\
& \quad \{I \land body \Rightarrow I'\} \\
& \quad \{I'\} \\
& \quad \{I \land \neg c \land final \Rightarrow post\} \\
\end{align*}
$$

verification condition
Horn Constraints

Constraints of this form are called Horn constraints (clauses)

\[ \phi \Rightarrow I' \]
\[ I \land \psi \Rightarrow I' \]
\[ I \Rightarrow \omega \]

How do we find \( I \)?

- Fix a *domain* (search space)
- Search for an element of the *domain* that makes all clauses true
Horn constraints for SumMax

\[ s' = 0 \land m' = 0 \land i' = 0 \Rightarrow I' \]

\[ I \land i < \text{len}(a) \land s' = s + a[i] \land \\
m < a[i] \land m' = a[i] \land i' = i + 1 \Rightarrow I' \]

\[ I \land i < \text{len}(a) \land s' = s + a[i] \land \\
m \geq a[i] \land m' = m \land i' = i + 1 \Rightarrow I' \]

\[ I \Rightarrow s \leq \text{len}(a) \cdot m \lor i < \text{len}(a) \]

(Solution: \( i \leq \text{len}(a) \land s \leq i \cdot m \))

Can we solve this with...

- Enumerative search?
  - \( I ::= T \Leftarrow T \mid I \&\& I \)
  - \( T ::= x \mid T . \text{length} \mid T + T \mid T \ast T \)

- Sketch?

\[ \exists I. \forall x . Q(I, x) \]

\[ \exists c. \forall x . Q(I[c], x) \]
Idea: Lattice Search

Domain = all conjunctions of predicates from $S$

$S = \{ i \leq \text{len}(a), i \geq \text{len}(a), i \neq \text{len}(a) \}$

$i' = 0 \land \text{len}(a) > 0 \Rightarrow I'$

$I \land i < \text{len}(a) \land i' = i + 1 \Rightarrow I'$

$I \Rightarrow i = \text{len}(a) \lor i < \text{len}(a)$
Example

Domain = all conjunctions of predicates from $S$

$S = \{ i \leq \text{len}(a), \ i \geq \text{len}(a), \ i \neq \text{len}(a) \}$

\[ i' = 0 \land \text{len}(a) > 0 \Rightarrow I' \]
\[ I \land i < \text{len}(a) \land i' = i + 1 \Rightarrow I' \]
\[ I \Rightarrow i = \text{len}(a) \lor i < \text{len}(a) \]

\[ I = i \leq \text{len}(a) \land i \geq \text{len}(a) \land i \neq \text{len}(a) \]
Example

Domain = all conjunctions of predicates from $S$

$S = \{ i \leq \text{len}(a), \ i \geq \text{len}(a), \ i \neq \text{len}(a) \}$

$i' = 0 \land \text{len}(a) > 0 \Rightarrow I'$

$I \land i < \text{len}(a) \land i' = i + 1 \Rightarrow I'$

$I \Rightarrow i = \text{len}(a) \lor i < \text{len}(a)$

$I = \ ? \ i \leq \text{len}(a) \land i \neq \text{len}(a)$
Example

Domain = all conjunctions of predicates from $S$

$S = \{i \leq \text{len}(a), i \geq \text{len}(a), i \neq \text{len}(a)\}$

- $i' = 0 \land \text{len}(a) > 0 \Rightarrow I'$
- $I \land i < \text{len}(a) \land i' = i + 1 \Rightarrow I'$
- $I \Rightarrow i = \text{len}(a) \lor i < \text{len}(a)$
Least Fixpoint (forward search)

Domain = all conjunctions of predicates from $S$

$S = \{ i \leq \text{len}(a), \ i \geq \text{len}(a), \ i \neq \text{len}(a) \}$

- $i' = 0 \land \text{len}(a) > 0 \Rightarrow I'$
- $I \land i < \text{len}(a) \land i' = i + 1 \Rightarrow I'$
- $I \Rightarrow i = \text{len}(a) \lor i < \text{len}(a)$

- Finds the strongest solution
- Did not have to look at all candidates
- Relies on efficient weakening operation
From verification to inference

Verification

\[ \forall x. Q(x) \]

SMT

UNSAT / SAT

Inference

\[ \exists I. \forall x. Q(I, x) \]

Fix the domain lattice
lattice search
otherwise
combinatorial search
From inference to synthesis

Verification

\[ \forall x. Q(x) \]

SMT

UNSAT / SAT

Inference

\[ \exists I. \forall x. Q(I, x) \]

Fix the domain
lattice \(\rightarrow\) lattice search
otherwise \(\rightarrow\) combinatorial search

Synthesis
Synthesizing a Loop

\[
\begin{align*}
\{ \text{pre} \} & \quad \exists S,G,I. \ \forall x. \quad \text{pre} \land S_0 \Rightarrow I' \\
\text{while (??)} & \quad \land \\
\text{invariant ??;} & \quad I \land G_0 \land G_1 \land S_1 \Rightarrow I' \\
\{ \text{??} \} & \quad \land \\
\{ \text{post} \} & \quad I \land G_0 \land G_2 \land S_2 \Rightarrow I' \\
\{ \text{post} \} & \quad \land \\
\{ \text{pre} \} & \quad I \land \neg G_0 \land S_f \Rightarrow \text{post} \\
\{ \text{pre} \} & \quad \exists S,G,I. \ \forall x. \quad Q(P,I,x)
\end{align*}
\]
Solving synthesis constraints

\[ I \land G_i \land S_i \land \psi \Rightarrow I' \]
\[ I \land G_i \land S_i \Rightarrow \omega \]
\[ T \Rightarrow G_i \lor G_j \]

Can we solve this with...

- Enumerative search?
  - Sure (slow)

- Sketch?
  - Yep! (need to provide sketch for invariants too)
  - Look ma, we made an unbounded synthesizer out of Sketch!

- Lattice search?  
  - Great for G, not so great for S (why?)

VS3 [Srivastava, 2010]
Main idea: Look for the proof to find the program

- The space of valid program derivations is smaller than the space of all programs
- The result is provably correct!

Applications:

✓ Constraint-based search: use loop invariants to encode the space of correct looping programs
✓ Enumerative search: prune unverifiable candidates early
✓ Deductive search: search in the space of provably correct implementations
Deductive Synthesis: Principle

Problem definition:
• Find $x$ such that $Q(x,a)$ whenever $P(a)$

Using semantic-preserving transformations, gradually *refine* the problem above into:
• Find $T$ such that $\top$ whenever $P(a)$
  ‣ where $T$ is a term that does not mention $x$

Find $x$ such that $x + x = 4a$

Find $x$ such that $2x = 4a$

Find $2y$ such that $4y = 4a$

Find $2y$ such that $y = a$

Find $2a$ such that $\top$
Deductive synthesis: challenges

Define a set of transformation rules that is sound
  • A solution to the transformed problem is a solution to the original problem

... and “complete”
  • All programs we care about can be derived

In most situations, multiple rules apply to a problem
  • Need a search strategy!
Idea: extract the program from a constructive proof of

$$\forall a. \exists x. P(a) \rightarrow Q(a, x)$$

• There is no need to invent any new reasoning or inference: reuse an existing theorem prover / proof assistant

• ...but record the proof steps and augment them with program extraction rules.

• Reuse any automation that exists in the prover!
Synthesis as Theorem Proving

Intuition (by example).

Axioms

1. $\text{head}(x :: xs) = x$
2. $\text{tail}(x :: xs) = xs$

Prove

$\exists l. \text{head}(l) = 5 \land \text{tail}(l) = []$
Synthesis as Theorem Proving

Sequent:

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_i(a, x)$</td>
<td>$G_i(a, x)$</td>
<td>$t_i(a, x)$</td>
</tr>
</tbody>
</table>

Meaning: if $\forall x. \bigwedge_i A_i(a, x)$ holds, then $G_i(a, x)$ holds

- and the corresponding $t_i$ is an acceptable solution

[Manna, Waldinger’80]
Synthesis as Theorem Proving

Synthesis problem: “Find $x$ such that $P$ whenever $Q$”

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P(a)$</td>
<td>$Q(a, x)$</td>
<td>$x$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
</tbody>
</table>

Apply inference rules to add new assertions and goals

$\vdots$

$T$

$(where \ t \ does \ not \ contain \ x)$
Inference Rules

Splitting
• E.g. split assertion $A_1 \land A_2$ into two assertions $A_1$ and $A_2$

Transformation
• Apply a rewrite rule $s \rightarrow t$ to a subterm of assertion / goal
• Apply the unifying substitution of the rewrite to the output!

Resolution
• Let $A[P], B[P]$ two assertions (both with subformula $P$); add assertion $A[\top] \lor B[\bot]$

Induction
• Introduce an induction hypothesis
Example: Quotient and Remainder

Specification:

\[ \text{div}(i, j), \text{rem}(i, j) \iff \text{find } (q, r) \text{ s.t.} \]
\[ i = q \cdot j + r \land 0 \leq r < j \]

where \( 0 \leq i \land 0 < j \)
## Example: base case

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $0 \leq i \land 0 &lt; j$</td>
<td>2. $i = q \cdot j + r \land 0 \leq r &lt; j$</td>
<td>$q$</td>
</tr>
<tr>
<td>and-split 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. $0 \leq i$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. $0 &lt; j$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>transform 2</td>
<td>5. $i = 0 + r \land 0 \leq r &lt; j$</td>
<td>$0$</td>
</tr>
<tr>
<td>$0 \cdot v \rightarrow 0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[q \rightarrow 0, v \rightarrow j]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>transform 5</td>
<td>6. $i = r \land 0 \leq r &lt; j$</td>
<td>$0$</td>
</tr>
<tr>
<td>$0 + v \rightarrow v$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[v \rightarrow r]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>resolve 6 &amp;</td>
<td>7. $0 \leq i &lt; j$</td>
<td>$0$</td>
</tr>
<tr>
<td>$v = v$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[v \rightarrow i, r \rightarrow i]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>resolve 7 &amp; 3</td>
<td>8. $i &lt; j$</td>
<td>$0$</td>
</tr>
<tr>
<td>[]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Notes:
- $d_i v(i, j)$
- $r e_m(i, j)$

### Additional:
- $0 \leq i \land 0 < j$
- $i = q \cdot j + r \land 0 \leq r < j$
- $q$
- $r$
**Example: step case**

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>3. (0 \leq i)</td>
<td>2. (i = q \cdot j + r \land 0 \leq r &lt; j)</td>
<td>(q) (r)</td>
</tr>
<tr>
<td>4. (0 &lt; j)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**transform 2**

\[(u + 1) \cdot v \rightarrow u \cdot v + v\]

\[\lceil q \rightarrow q' + 1, u \rightarrow q', v \rightarrow j \rceil\]

**transform 9**

...  

**induction**

11. \((u, v) < (i, j) \Rightarrow\)

\[0 \leq u \land 0 < v \Rightarrow\]

\[u = \text{div}(u, v) \cdot v + \text{rem}(u, v) \land 0 \leq \text{rem}(u, v) < v\]

**termination**

**IH**

**resolve 10 & 11**

\[
\begin{align*}
\text{u} & = i - j, v = j \\
q' & \rightarrow \text{div}(i - j, j), r \rightarrow \text{rem}(i - j, j)
\end{align*}
\]

12. \((i - j, j) < (i, j) \land 0 \leq i - j \land 0 < j\)

\[\text{div}(i - j, j) + 1 \quad \text{rem}(i - j, j)\]

13. \(\neg (i < j)\)

...
Example: put them together

<table>
<thead>
<tr>
<th></th>
<th>assertions</th>
<th>goals</th>
<th>(div(i, j))</th>
<th>(rem(i, j))</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.</td>
<td>(i &lt; j)</td>
<td></td>
<td>0</td>
<td>(i)</td>
</tr>
<tr>
<td>13.</td>
<td>(\neg(i &lt; j))</td>
<td>(div(i - j, j) + 1)</td>
<td>(rem(i - j, j))</td>
<td></td>
</tr>
<tr>
<td>resolve 8 &amp; 13</td>
<td>[]</td>
<td>T</td>
<td>(\text{if } i &lt; j \text{ then } 0 \text{ else } div(i - j, j) + 1)</td>
<td>(\text{if } i &lt; j \text{ then } i \text{ else } rem(i - j, j))</td>
</tr>
</tbody>
</table>
Almost Done
Synthesis!

- Automatic programming?
  - but I have to tell the computer what I want...

---

???

Python, Haskell, ...

C

assembly

machine code

Synthesis = an unusually concise / intuitive programming language + a compiler that sometimes doesn’t work 😊
**concise**

**ADJECTIVE**

Giving a lot of information clearly and in a few words; brief but comprehensive.  
‘*a concise account of the country’s history*’

- **Synonyms**
  - *succinct*, short, brief, to the point, pithy, incisive, short and sweet, crisp

**Origin**

Late 16th century: from French concis or Latin concisus, past participle of concidere ‘cut up, cut down’, from con- ‘completely’ + caedere ‘to cut’.
concise

*adjective* • US [ˈkənˌsaɪs] /kənˈsaɪs/

expressing what needs to be said without unnecessary words; short and clear:

*She wrote up a concise summary of the day’s events.*
struct Node {
    int val;
    Node next;
}

Node mklist(int n, int[n] elems) {
    Node h = null, t = null;
    for (int i = 0; i < n; i++) {
        Node nt = new Node(val=elems[i]);
        if (t == null) { h = nt; t = nt; } 
        else { t.next = nt; t = nt; }
    }
    return h;
}

bit is_sorted(Node h, fun R) {
    return h == null || h.next == null || R(h.val, h.next.val) && is_sorted(h.next, R);
}

int length(Node h) {
    if (h == null) return 0;
    else return 1 + length(h.next);
}

Node reverse(Node h) {
    Node i = {| h | null |}, j = {| h | null |};
    while (i != null) {
        t := i.next;
        i.next := j;
        j := i;
        i := i.next;
    }
    return j;
}

harness void main(int n, int[n] elems) {
    assume n < 8;
    Node h = mklist(n, elems); assume is_sorted(h, (x,y) -> x < y);
    h = reverse(h);
    assert is_sorted(h, (x,y) -> x > y);
    assert length(h) == n;
}

Implementation of reverse

Sketch of reverse
**data List e where**
- Nil :: List e
- Cons :: e → List e → List e

// Insert x into a sorted list xs
**insert :: x: e → xs: List e → List e**

insert x xs =
  match xs with
  Nil → Cons x Nil
  Cons h t →
    if x ≤ h
      then Cons x xs
      else Cons h (insert x t)

**Synquid specification of insert**

**qualifier** \{x ≤ y, x ≠ y\}

**data List a where**
- Nil :: List a
- Cons :: x: a → xs: List a → List a

**termination measure** len :: List a → {Int | _v ≥ 0} where
- Nil → 0
- Cons x xs → 1 + len xs

**measure** elems :: List a → Set a where
- Nil → []
- Cons x xs → [x] + elems xs

**data IList a where**
- INil :: IList a
- ICons :: x: a → xs: IList {a | x ≤ _v} → IList a

**termination measure** ilen :: IList a → {Int | _v ≥ 0} where
- INil → 0
- ICons x xs → 1 + ilen xs

**measure** ielems :: IList a → Set a where
- INil → []
- ICons x xs → [x] + ielems xs

**insert :: x: a → xs: IList a → {IList a | ielems _v == ielems xs + [x]}**
insert = ??
What we

Some day we won’t even need coders any more. We’ll be able to just write the specification and the program will write itself.

Oh wow, you’re right! We’ll be able to write a comprehensive and precise spec and bam, we won’t need programmers any more!

Exactly!

And do you know the industry term for a project specification that is comprehensive and precise enough to generate a program?

Code

Uh... no...

It’s called code.

CommitStrip.com
What went wrong?

Writing specifications is hard!
- have to think about all the corner cases
- have to encode hidden assumptions as formulas

Writing specifications for a synthesis tool is even harder!
- it’s the user’s responsibility to define the search space
- too much freedom — divergence (or just very long running times)
- too restrictive — non-realizability (and that’s hard to debug)
- have to reduce to the logical fragment that the tool supports
What can we do??

Improved specification language
we have seen a lot of repetitive mundane stuff

Reusable specifications
declarative specs are highly composable

Partial specifications
less to write; but heuristics are needed to choose the desired program

Better algorithms...
faster search = less to worry about = easier spec work
Final Project
Logistics

Due: end of August.

- This is an arbitrary deadline. The project itself can be completed within within a few days.
- Still, if you have too many exams / are leaving the country / are joining the military, course staff will be merciful.
- Like homework assignments, projects will be done in pairs.
Project Requirements

We will build a synthesizer.

Don’t worry: it’s going to be a rather small synthesizer.

Mode of specification:

<table>
<thead>
<tr>
<th>PBE</th>
<th>i</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[16,77,31]</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>[60, 9, 61, 63, 1]</td>
<td>2</td>
</tr>
</tbody>
</table>

Output language:

You choose!

- But — must include some kind of
  list operations

sorted(input)[-1] - sorted(input)[-2]
We will need **Observational Equivalence**.

Following this pseudo-code:

\[
P := [V \sim t \mid V ::= t \in R \land \text{ground}(t)]
\]

\[
\text{while } (\text{true}) \quad P += \text{grow}(P);
\]

\[
\text{for all } (p \text{ in } P) \quad \text{if } (S \sim p \land p([i]) = \lbrack o \rbrack) 
\quad \text{return } p;
\]

\[
p_1 \equiv_o p_2 \iff \bigwedge_{\{i \rightarrow o\} \in \text{spec}} p_1([i]) = p_2([i])
\]
One more thing

Come up with some small improvement, either in:

- the interface in which the user interacts with the tool
- the size of the program space that can feasibly be explored
That something extra

Symbolic examples
Synthesis of body of lambdas
Topmost-level constraint solving
User-defined sketches
Condition abduction for corner cases
Bye!