Deductive Synthesis

Programs vs. Proofs

Closing Arguments
Reasoning

λ-calculus

Dependent Types

Synthesis

Programming by Example

Syntax Guided Synthesis

Counterexample Guided Inductive Synthesis

Type Directed Synthesis

Refinement Types

Type Theory (basics)

Axiomatic Semantics

Satisfiability Modulo Theory
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?
Verification

```daffny
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 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; }
}
```

Dafny
AutoProof
VCC
Verifast
...
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;
  }
}

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

can’t find an invariant that lets me prove correctness

correct!

BLAST
Astrée
FB Infer
LiquidHaskell
...

Invariant inference
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;
}

can’t find a (program, invariant) pair that I can prove correct

VS3
Natural Synthesis
(Leon)
(Synquid)
From Verification to Synthesis

Verification

Program logic

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

SMT

UNSAT ⚡ SAT

Inference

Program logic

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

unknown formulas for invariants

Synthesis

Program logic

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

unknown formulas for invariants and commands

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

\[ \forall x. Q(x) \]
Step 1: eliminate loops

{\top}
sum := 0;
max := 0;
i := 0;
while (i < a.length)
    invariant i <= a.length &&
    sum <= i * max;
{assert i < a.length; if (max < a[i]) max := a[i];
    sum := sum + a[i];
i := i + 1;
}
{s <= len(a) \cdot m}

{\top}
sum := 0;
max := 0;
i := 0;
{i <= len(a) \land s <= i \cdot m}\n
{\top}
assert i < a.length;
if (max < a[i]) max := a[i];
sum := sum + a[i];
i := i + 1;
{i <= len(a) \land s <= i \cdot m}

{\top}
skip
{i <= len(a) \land s <= i \cdot m \land \neg (i < len(a))}\n
{s <= len(a) \cdot m}
Viewed as a Transition System

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

\[
\begin{align*}
\{T\} & \quad \text{sum'} := 0; \\
\ & \quad \text{max'} := 0; \\
\ & \quad \text{i'} := 0; \\
\ & \quad \{i' \leq \text{len}(a) \land s' \leq i' \cdot m'\}
\end{align*}
\]

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

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

\[
\begin{align*}
\{i \leq \text{len}(a) \land s \leq i \cdot m \land \neg(i < \text{len}(a))\} & \quad \text{skip} \\
\ & \quad \{s \leq \text{len}(a) \cdot m\}
\end{align*}
\]

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

\[
\begin{align*}
\{\top\} & \quad s' = 0 \land m' = 0 \land i' = 0 \\
\{i' \leq \text{len}(a) \land s' \leq i' \cdot m'\} & \\
\{i \leq \text{len}(a) \land s \leq i \cdot m \land i < \text{len}(a)\} & \\
& \quad s' = s + a[i] \land i' = i + 1 \land m < a[i] \rightarrow m' = a[i] \land \neg(m < a[i]) \rightarrow m' = m \\
& \quad \{i' \leq \text{len}(a) \land s' \leq i' \cdot m'\} \\
\{i \leq \text{len}(a) \land s \leq i \cdot m \land \neg(i < \text{len}(a))\} & \\
& \quad s' = s \land m' = m \land i' = i \\
& \quad \{s' \leq \text{len}(a) \cdot m'\}
\end{align*}
\]

∀s, m, i, s', m', i'.

\[
\begin{align*}
T & \land s' = 0 \land m' = 0 \land i' = 0 \\
\Rightarrow & \quad i' \leq \text{len}(a) \land s' \leq i' \cdot m' \\
i & \leq \text{len}(a) \land s \leq i \cdot m \land i < \text{len}(a) \land \\
& \quad s' = s + a[i] \land i' = i + 1 \land \\
m & < a[i] \rightarrow m' = a[i] \land \\
& \quad \neg(m < a[i]) \rightarrow m' = m \\
\Rightarrow & \quad i' \leq \text{len}(a) \land s' \leq i' \cdot m' \\
i & \leq \text{len}(a) \land s \leq i \cdot m \land \neg(i < \text{len}(a)) \land \\
& \quad s' = s \land m' = m \land i' = i \\
\Rightarrow & \quad s' \leq \text{len}(a) \cdot m'
\end{align*}
\]
VC for Transition Systems

\[\forall x, x'. \quad \text{pre} \land \text{init} \Rightarrow \text{inv}'\]
\[\text{inv} \land c \land \text{body} \Rightarrow \text{inv}'\]
\[\text{inv} \land \neg c \land \text{final} \Rightarrow \text{post}\]

\[
\begin{align*}
\{\text{pre}\} & \quad \text{init} & \quad \{\text{pre}\} & \quad \text{init}(x, x') \\
\{\text{inv}\} & \quad \{\text{inv}\} & \quad \{\text{inv}\} & \quad \{\text{inv}\} \\
\text{while } (c) & \quad \text{body} & \quad \text{body}(x, x') & \quad \text{body}(x, x') \\
\{\text{inv} \land c\} & \quad \{\text{inv} \land c\} & \quad \{\text{inv} \land c\} & \quad \{\text{inv} \land c\} \\
\{\text{inv}\} & \quad \{\text{inv}\} & \quad \{\text{inv}\} & \quad \{\text{inv}\} \\
\{\text{post}\} & \quad \{\text{post}\} & \quad \{\text{post}\} & \quad \{\text{post}\} \\
\text{final} & \quad \text{final}(x, x') & \quad \text{final}(x, x') & \quad \text{final}(x, x') \\
\{\text{inv} \land \neg c\} & \quad \{\text{inv} \land \neg c\} & \quad \{\text{inv} \land \neg c\} & \quad \{\text{inv} \land \neg c\} \\
\{\text{post}\} & \quad \{\text{post}\} & \quad \{\text{post}\} & \quad \{\text{post}\} \\
\end{align*}
\]
From verification to inference

Verification

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

SMT

UNSAT / SAT

Inference

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

\[
\begin{align*}
\{\text{pre}\} & \quad \text{init} & \quad \{\text{pre}\} & \quad \text{init}(x, x') & \quad \{\text{pre}\} \\
\{I\} & \quad \text{body} & \quad \{I\} & \quad \text{body}(x, x') & \quad \{I\} \\
\{I \land c\} & \quad \text{final} & \quad \{I \land c\} & \quad \text{final}(x, x') & \quad \{I \land c\} \\
\{I \land \neg c\} & \quad \text{final} & \quad \{I \land \neg c\} & \quad \{post\} & \quad \{post\} \\
\{post\} & \quad \text{final} & \quad \{post\} & \quad \text{final}(x, x') & \quad \{post\} \\
\end{align*}
\]

\[\exists I. \forall x. (\text{pre} \land \text{init} \Rightarrow I') \land (I \land c \land \text{body} \Rightarrow I') \land (I \land \neg c \land \text{final} \Rightarrow \text{post})\]

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 \textit{domain} (search space)
- Search for an element of the \textit{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?
  - \( I ::= T \leq T \lor I \land I \)
  - \( T ::= x \lor T.length \lor T + T \lor T \ast T \)

- 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 \{p, q, r\} ≡ \{i ≤ \text{len}(a), i ≥ \text{len}(a), i ≠ \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 \{p, q, r\} \equiv \{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
\{p, q, r\} \equiv \{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
\{p, q, r\} \equiv \{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
\{p, q, r\} \equiv \{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
Greatest Fixpoint (backward search)

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

Error: \[ I \Rightarrow \omega \]
Greatest Fixpoint (backward search)

- Finds the *weakest* solution
- Relies on efficient *strengthening* operation
  - hard to implement

\[
\phi \Rightarrow I'
\]
\[
I \land \psi \Rightarrow I'
\]
\[
I \Rightarrow \omega
\]
From verification to inference

Verification

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

\[\text{SMT} \]

\[\text{UNSAT} / \text{SAT} \]

Inference

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

Fix the domain

\[ \text{lattice} \rightarrow \text{lattice search} \]

\[ \text{otherwise} \rightarrow \text{combinatorial search} \]
From inference to synthesis

Verification

∀x. Q(x)

SMT

UNSAT / SAT

Inference

∃I. ∀x. Q(I, x)

Fix the domain
lattice → lattice search
otherwise → combinatorial search

Synthesis

∀x. Q(x)
Synthesizing a Loop

```
{pre}
??
while (??) invariant ??;
{ ?? }
??
{post}

exists S, G, I. ∀ x.
pre ∧ S_i ⇒ I'
∧
I ∧ G_0 ∧ G_1 ∧ S_1 ⇒ I'
∧
I ∧ G_0 ∧ G_2 ∧ S_2 ⇒ I'
∧
I ∧ ¬G_0 ∧ S_f ⇒ post
```

synthesis condition

```
∃ P, I. ∀ x. Q(P, I, x)
```
Synthesis constraints

Similar to Horn constraints but not quite

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

Domain for \( G \): like in inference

Domain for \( S \):
- conjunction of equalities, one per variable
Solving synthesis constraints

Can we solve this with...

- Enumerative search?
  - Sure (slow)
- Sketch?
  - Yep!
  - Look we made an unbounded synthesizer out of Sketch!
- Lattice search?
  - Great for G, not so great for S (why?)

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

VS3 [Srivastava, 2010]
The takeaway

We can reason about unbounded loops using loop invariants
  • Hoare logic soundly translates a program with a loop (and invariant) into three straight-line programs

We can synthesize a program with a loop by synthesizing those three straight-line programs (and the invariant)!
  • Can use existing synthesis techniques

Powerful idea: to synthesize a provably correct program, look for the program and its proof together
Deductive Synthesis

for real
Deductive reasoning for synthesis

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 2$y$ such that $4y = 4a$
Find 2$y$ such that $y = a$
Find 2$a$ such that $\top$
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 \( \) holds

• and the corresponding \( t_i \) is an acceptable solution
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></td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td></td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>Apply inference rules to add new assertions and goals</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>$t$</td>
</tr>
</tbody>
</table>

(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:

\[ \text{div}(i, j), \text{rem}(i, j) \iff \text{find } (q, r) \text{ s.t. } \]

\[ i = q \times j + r \land 0 \leq r < j \]

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. $i = q \times j + r \land 0 \leq r &lt; j$</td>
<td>$q$ $r$</td>
</tr>
<tr>
<td>and-split 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. $0 \leq i$</td>
<td>4. $0 &lt; j$</td>
<td>$q$ $r$</td>
</tr>
<tr>
<td>trans 2</td>
<td>5. $i = 0 + r \land 0 \leq r &lt; j$</td>
<td>$0$ $r$</td>
</tr>
<tr>
<td>$0 \times v \rightarrow 0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[q \rightarrow 0, v \rightarrow j]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>trans 5</td>
<td>6. $i = r \land 0 \leq r &lt; j$</td>
<td>$0$ $r$</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>$0$ $i$</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>$0$ $i$</td>
</tr>
<tr>
<td>[]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Example: step case

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>3. $0 \leq i$ 4. $0 &lt; j$</td>
<td>$i = q \ast j + r \land 0 \leq r &lt; j$</td>
<td>$q$, $r$</td>
</tr>
<tr>
<td>$u + 1) \ast v \rightarrow u \ast v + v$</td>
<td>$i = q' \ast j + j + r \land 0 \leq r &lt; j$</td>
<td>$q' + 1$, $r$</td>
</tr>
<tr>
<td>trans 2</td>
<td>trans 9</td>
<td>$i - j = q' \ast j + r \land 0 \leq r &lt; j$</td>
</tr>
<tr>
<td>$u = div(u, v) \ast v + rem(u, v)$ \land $0 \leq rem(u, v) &lt; v$</td>
<td>$u = i - j, v = j$</td>
<td>$u = i - j, v = j$</td>
</tr>
<tr>
<td>$q' \rightarrow div(i - j, j), r \rightarrow rem(i - j, j)$</td>
<td></td>
<td>$q' \rightarrow div(i - j, j), r \rightarrow rem(i - j, j)$</td>
</tr>
<tr>
<td>$u = i - j, v = j$</td>
<td></td>
<td>$u = i - j, v = j$</td>
</tr>
<tr>
<td>$q' \rightarrow div(i - j, j), r \rightarrow rem(i - j, j)$</td>
<td></td>
<td>$q' \rightarrow div(i - j, j), r \rightarrow rem(i - j, j)$</td>
</tr>
</tbody>
</table>

**Induction**

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

12. $(i - j, j) < (i, j) \land$

13. $\neg(i < j)$
Example: put them together

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>8. $i &lt; j$</td>
<td></td>
<td>$0$</td>
</tr>
<tr>
<td>13. $\neg(i &lt; j)$</td>
<td></td>
<td>$\text{div}(i - j, j) + 1$</td>
</tr>
</tbody>
</table>

resolve 8 & 13

if $i < j$ then 0 else $\text{div}(i - j, j) + 1$

if $i < j$ then $i$ else $\text{rem}(i - j, j)$
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

View synonyms

**Origin**

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

(Oxford Dictionary)
concise

adjective • US /kənˈsaɪs/

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

She wrote up a concise summary of the day’s events.

(Cambridge Dictionary)
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;
}
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)

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

It’s called code.
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>i</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>[16,77,31]</td>
<td>46</td>
</tr>
<tr>
<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

\[ \text{sorted(input)}[-1] - \text{sorted(input)}[-2] \]
Project Requirements

We will need **Observational Equivalence**.

Following this pseudo-code:

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

\[
\text{while (true)}
\]

\[
P := \text{grow}(P);
\]

\[
\text{forall (p in P)}
\]

\[
\text{if (S \sim p \land p([i]) = [o])}
\]

\[
\text{return p;}
\]

\[
p_1 \equiv_o p_2 \iff \bigwedge_{\{i \rightarrow o\} \in \text{spec}} p_1([i]) = p_2([i])
\]
That something extra

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