Deductive Synthesis

Programs vs. Proofs

Closing Arguments
Reasoning

\( \lambda \)-calculus

Dependent Types

Type Theory (basics)

Axiomatic Semantics

Syntax Guided Synthesis

Counterexample Guided Inductive Synthesis

Satisfiability Modulo Theory

Type Directed Synthesis

Programming by Example

Refinement Types

Synthesis
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

```dsharp
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; }
}
```
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;
        }
    }
method SumMax (a: array<int>) returns (sum: int, max: int)
  requires a != null;
  ensures sum <= a.Length * max;
{
  var i;
  ??;
  while (??)
    invariant ??;
    decreases ??;
  {
    ??;
  }
}

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;
  }

found a correct program!

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

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}

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

{\top}

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

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

{\top}

\begin{align*}
\{& \text{i} \leq \text{len(a)} \land \text{s} \leq \text{i} \cdot \text{m} \land \neg (\text{i} < \text{len(a)})\} \\
& \text{skip} \\
\{& \text{s} \leq \text{len(a)} \cdot \text{m}\}
\end{align*}
Viewed as a Transition System

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

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

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

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

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

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

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

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

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

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

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

\[
\forall x, x'. \quad \text{pre} \land \text{init} \Rightarrow \text{inv}'
\]

\[
\text{inv} \land c \land \text{body} \Rightarrow \text{inv}'
\]

\[
\text{inv} \land \neg c \land \text{final} \Rightarrow \text{post}
\]

verification condition
From verification to inference

Verification

$\forall x. Q(x)$

SMT

UNSAT / SAT

Inference

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

\[
\exists I. \forall x .
pre \land \text{init} \Rightarrow I'
\]

\[
I \land c \land \text{body} \Rightarrow I'
\]

\[
I \land \neg c \land \text{final} \Rightarrow \text{post}
\]

\[
\{ \text{pre} \}
\]

\[
\text{init}
\]

\[
\{ I \}
\]

\[
\text{while (c) invariant ??;}
\]

\[
\{ \text{body} \}
\]

\[
\{ I \land c \}
\]

\[
\{ pre \}
\]

\[
\text{init(x, x')}
\]

\[
\{ I[x \mapsto x'] \}
\]

\[
\{ I \}
\]

\[
\text{body(x, x')}
\]

\[
\{ I \land c \}
\]

\[
\text{body(x, x')}
\]

\[
\{ I[x \mapsto x'] \}
\]

\[
\{ \text{final} \}
\]

\[
\{ I \land \neg c \}
\]

\[
\{ post \}
\]

\[
\text{final(x, x')}
\]

\[
\{ I \land \neg c \}
\]

\[
\{ post \}
\]

\[
\text{final(x, x')}
\]

\[
\{ post \}
\]

\[
\text{verification condition}
\]
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?

- 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?
  - \( I ::= T \leq T \mid I \&\& I \)
  - \( T ::= x \mid T.length \mid T + T \mid T * T \)
- 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\} \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)\}

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

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

\[
\begin{align*}
    \text{Domain} &= \text{all conjunctions of predicates from } \{p, q, r\} \\
    &\equiv \{i \leq \text{len}(a), i \geq \text{len}(a), i \neq \text{len}(a)\}
\end{align*}
\]

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

- Finds the strongest solution
- Did not have to look at all candidates
- Relies on efficient weakening operation
Greatest Fixpoint (backward search)

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

= SMT

UNSAT / SAT

Inference

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

Fix the domain

lattice \rightarrow \text{lattice search}

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

- **Verification**
  - SMT
  - UNSAT / SAT

- **Inference**
  - Fix the domain
    - lattice $\rightarrow$ lattice search
    - otherwise $\rightarrow$ combinatorial search

- **Synthesis**
Synthesizing a Loop

\[
\exists S, G, I. \ \forall x.
\\{ \text{pre} \}
\ \Rightarrow
\ {\text{post}}
\]

\[
\text{while } (??) \text{ invariant } ??; \{ ?? \} ??
\]

\[
\{ \text{pre} \}
\ {S_f(x, x')} \{ I[x \mapsto x'] \}
\]

\[
\{ \text{I} \wedge G_0 \}
\ G_1 \rightarrow S_1(x, x') \{ I[x \mapsto x'] \}
\]

\[
\{ \text{I} \wedge \neg c \}
\ S_f(x, x') \{ \text{post} \}
\]

\[
\{ \text{I} \wedge G_0 \wedge G_1 \wedge S_1 \Rightarrow I' \}
\ \wedge
\]

\[
\{ \text{I} \wedge G_0 \wedge G_2 \wedge S_2 \Rightarrow I' \}
\ \wedge
\]

\[
\{ \text{I} \wedge \neg G_0 \wedge S_f \Rightarrow \text{post} \}
\]

synthesis condition

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

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

Can we solve this with...

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

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**

[Manne, Waldinger’80]

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

| $T$ | $t$ |

(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:

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

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><strong>and-split 1</strong></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><strong>trans 2</strong></td>
<td>5. $i = 0 + r \land 0 \leq r &lt; j$</td>
<td>$0$ $r$</td>
</tr>
<tr>
<td>$0 \times v \rightarrow v$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[q \rightarrow 0, v \rightarrow j]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>trans 5</strong></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><strong>resolve 6</strong></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><strong>resolve 7 &amp; 3</strong></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. $0 \leq i$</td>
<td>$i = q \ast j + r \land 0 \leq r &lt; j$</td>
<td>$q$</td>
</tr>
<tr>
<td>4. $0 &lt; j$</td>
<td></td>
<td>$r$</td>
</tr>
<tr>
<td><strong>trans 2</strong></td>
<td></td>
<td><strong>div</strong>($i, j$)  <strong>rem</strong>($i, j$)</td>
</tr>
<tr>
<td>$(u + 1) \ast v \rightarrow u \ast v + v$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[q \rightarrow q' + 1, u \rightarrow q', v \rightarrow j]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>trans 9</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>induction</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11. $(u, v) &lt; (i, j) \Rightarrow$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$0 \leq u \land 0 &lt; v \Rightarrow$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$u = \text{div}(u, v) \ast v + \text{rem}(u, v)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\land 0 \leq \text{rem}(u, v) &lt; v$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>resolve 10 and 11</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$u = i - j, v = j$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q' \rightarrow \text{div}(i - j, j), r \rightarrow \text{rem}(i - j, j)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>...</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>12.</strong> $(i - j, j) &lt; (i, j) \land$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$0 \leq i - j \land 0 &lt; j$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>div</strong>($i - j, j$) + 1</td>
<td><strong>rem</strong>($i - j, j$)</td>
</tr>
<tr>
<td><strong>13.</strong> $\neg(i &lt; j)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>...</strong></td>
</tr>
</tbody>
</table>
## 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 |      |         |
| []             | $\top$ |         |

- If $i < j$ then $0$ else $\text{div}(i - j, j) + 1$
- If $i < j$ then $i$ else $\text{rem}(i - j, j)$
Synthesis!

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

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

level of abstraction

- ???
- Python, Haskell, ...
- C
- assembly
- machine code
**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 concisisus, past participle of concidere ‘cut up, cut down’, from con- ‘completely’ + caedere ‘to cut’.
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.
// Insert x into a sorted list xs

}\!

insert :: x: e \!\rightarrow\! xs : List e \!\rightarrow\! List e

insert x xs =
  match xs with
  | Nil \!\rightarrow\! Cons x Nil
  | Cons h t \rightarrow
    if x \leq h
      then Cons x xs
      else Cons h (insert x t)

qualifier \{x \leq y, x \neq y\}

data List a where
  Nil :: List a
  Cons :: x: a \!\rightarrow\! xs: List a \!\rightarrow\! List a

termination measure len :: List a \!\rightarrow\! \{Int | _v \geq 0\} where
  Nil \!\rightarrow\! 0
  Cons x xs \!\rightarrow\! 1 + len xs

measure elems :: List a \!\rightarrow\! Set a where
  Nil \!\rightarrow\! []
  Cons x xs \!\rightarrow\! [x] + elems xs

data IList a where
  INil :: IList a
  ICons :: x: a \!\rightarrow\! xs: IList \{a | x \leq _v\} \!\rightarrow\! IList a

termination measure ilen :: IList a \!\rightarrow\! \{Int | _v \geq 0\} where
  INil \!\rightarrow\! 0
  ICons x xs \!\rightarrow\! 1 + ilen xs

measure ielems :: IList a \!\rightarrow\! Set a where
  INil \!\rightarrow\! []
  ICons x xs \!\rightarrow\! [x] + ielems xs

insert :: x: a \!\rightarrow\! xs: IList a \!\rightarrow\! \{IList a | ielems _v == ielems xs + [x]\}
insert = ??

Implementation of insert
Synquid specification of insert
```c
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;
}

implementations

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

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);
}

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**
What went wrong?

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.
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 July.

• This is an arbitrary deadline. The project itself can be completed within a day (~100 lines of code).

• Still, if you have too many exams / are leaving the country / are joining the military, course staff will be merciful.

• Despite popular demand, projects are individual.
Project Requirements

We will build a compiler.

Don’t worry: it’s going to be a very small compiler.

Input language:
\[ E ::= \# \mid v \mid E \bowtie E \]

Output language:
\[ S ::= C^* \]
\[ C ::= \text{push } n \mid \text{pull } idx \mid \text{calc } '\diamond' \]

...and you don’t even have to write a parser.
Project Requirements

We will build a compiler in SKETCH.

Instead of writing the semantic actions, we will just write a sketch for them.

We will use the same sketch for all actions!

\[
E ::= \# \{ \ldots \}\|
v\{ \ldots \}\|
E \triangleright E \{ \ldots \}\]

struct Ins {
int op;
int arg;
}
Instructions & Guidance

Specification: the compiled code, when executed, has to give the same result as interpreting the original program.

\[ \text{execute} \circ (\text{compile } p) \equiv \text{eval } p \]

• For \( p \): use PBE (a handful of example programs should get you there).
• For the input to \( p = \) values of input variables: use CEGIS.
Use every resource available

- Solutions to labs  (available in Moodle)
- TA’s office hours  (by appointment)