Lecture #12

Deductive Synthesis

Type-Directed Synthesis
Reasoning

- λ-calculus
- Dependent Types

Type Theory (basics)

Axiomatic Semantics

Satisfiability Modulo Theory

Programming by Example

Syntax Guided Synthesis

Counterexample Guided Inductive Synthesis

Type Directed Synthesis

Refinement Types

Synthesis

AxiomaFc

SemanFsfiability

Modulo Theory

Programming by Example

Syntax Guided Synthesis

Counterexample Guided Inductive Synthesis

Type Directed Synthesis

Refinement Types
Dimensions in program synthesis

[Gulwani 2010]

Behavioral constraints
how do you tell the system
what the program should do?

Structural constraints
what is the space of programs
to explore?

Rich Functional Specifications

Enumerative + Deduction

Search strategy
How does the system find
the program you want?

Context Free Grammar
Sketches (programs with holes)
Top-down Enumeration

- Start from initial symbol
- Repeatedly expand nonterminals using productions

\[ \text{top-down (}\langle T, V, R, S\rangle, \{x \rightarrow y\}\text{)} \]

\[ P := [S] \]

\[ \text{while (}P \neq []\text{)} \]

\[ p = P.\text{dequeue}(); \]

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

\[ \text{return } p; \]

\[ P.\text{enqueue}(\text{expand}(p)) \]

**expand**(p, R)

\[ L \rightarrow [\text{sort}(L), L[N..N], L + L, x] \]

\[ L[N..N] \rightarrow [\text{sort}(L)[N..N], L[N..N][N..N], ..., L[0..N], L[N..0]] \]
Pruning

• When can we discard a subprogram?

  ‣ It’s equivalent to something we have already explored
  ‣ No matter what we combine it with, it cannot satisfy the spec

Equivalence reduction
(also: symmetry breaking)

Top-down propagation

impossible, regardless of how we expand N and L

```
@ sort(x)
sort(sort(x))
L + L
[N] + L
```

```
[ ] \rightarrow [ ]
```

```
[ ] \rightarrow [ ]
```

```
L + L
```

```
[N] + L
```
Top-Down Propagation

• **Idea**: once we pick the production, infer specs for subprograms

If \( \text{spec1} \models \bot \), discard \( E_1 \oplus E_2 \) altogether!

For starters: \( \text{spec} = \{ x \rightarrow y \} \)
Example: PBE + TDP

• $\lambda^2$: synthesis for list combinators

map $f \times x$  
map ($\lambda y. \ y + 1$) $[1, -3, 1, 7] \rightarrow [2, -2, 2, 8]$

filter $f \times x$  
filter ($\lambda y. \ y > 0$) $[1, -3, 1, 7] \rightarrow [1, 1, 7]$

fold $f \ \text{acc} \ x$  
fold ($\lambda y z. \ y + z$) 0 $[1, -3, 1, 7] \rightarrow 6$

fold ($\lambda y z. \ y + z$) 0 $[] \rightarrow 0$

[Feuer, Chaudhuri, Dillig ‘15]
Example: PBE + TDP

• $\lambda^2$: synthesis for list combinators

[Feser, Chaudhuri, Dillig '15]

$[1 \rightarrow 2, -3 \rightarrow -2, 7 \rightarrow 8]$

Implemented as a hard-coded set of rules that derive examples for sub-program(s) given the examples for the whole program and the combinator

This can be generalized with deductive reasoning!
Example: Deduction + TDP

- **Morpheus**: synthesis of table transformations

**Input-output examples**

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>Tom</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Bob</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>Tom</td>
<td>12</td>
</tr>
</tbody>
</table>

**Components**

- project: Table $\rightarrow \text{[Col]} \rightarrow$ Table
- filter: Table $\rightarrow (\text{Row} \rightarrow \text{Bool}) \rightarrow$ Table

---

out.rows = in.rows $\land$ out.cols < in.cols

out.rows < in.rows $\land$ out.cols = in.cols

[Feng et al’17]
Example: Deduction + TDP

\[ \exists x, y. \ (y.\text{rows} = x.\text{rows} \land y.\text{cols} < x.\text{cols}) \]
\[ \land x.\text{rows} = 3 \land x.\text{cols} = 3 \]
\[ \land y.\text{rows} = 2 \land y.\text{cols} = 3 \]
\[ \rightarrow \text{UNSAT} \]

\[ \exists x, y. \ (y.\text{rows} < x.\text{rows} \land y.\text{cols} = x.\text{cols}) \]
\[ \land x.\text{rows} = 3 \land x.\text{cols} = 3 \]
\[ \land y.\text{rows} = 2 \land y.\text{cols} = 3 \]
\[ \rightarrow \text{SAT} \]

project: Table → [Col] → Table

filter: Table → (Row → Bool) → Table

\[ \text{out.rows} = \text{in.rows} \land \text{out.cols} < \text{in.cols} \]

\[ \text{out.rows} < \text{in.rows} \land \text{out.cols} = \text{in.cols} \]
“Synthesis-Friendly Verification”

• Good deductive system for synthesis?
  1. good at rejecting incomplete programs
  2. general
  3. expressive

• Type checkers can do 1 and 2!
  ▸ and by extending our type system, can do 3 as well
// 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)

data List e where
  Nil :: List e
  Cons :: h:e → t: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 xs ...
  ...

Expected
e
and got
List e

[Rejection Incomplete Programs]
[Pierce, Turner. TPLS'00]
// Insert x into a sorted list \( xs \)

\[
\text{insert} :: \ x : \text{e} \rightarrow \ xs : \text{List e} \rightarrow \ \text{List e}
\]

\[
\text{insert x xs} =
\]
\[
\text{match } \text{xs with}
\]
\[
\text{Nil} \rightarrow \text{Nil}
\]
\[
\text{Cons } h \ t \rightarrow
\]
\[
\text{if } x \leq h
\]
\[
\text{then } \text{Cons x xs}
\]
\[
\text{else } \text{Cons h (insert x t)}
\]

Conventional types are not enough.
Refinement Types

\[ n :: \{ v : \text{Int} \mid v \geq 0 \} \]

Basic types

Nat

I am a type!
Refinement Types

\[ n :: \{ \nu :: \text{Int} \mid \nu \geq 0 \} \]

\[ \text{max} :: x :: \text{Int} \to y :: \text{Int} \to \{ \nu :: \text{Int} \mid x \leq \nu \land y \leq \nu \} \]

\[ \text{xs} :: \{ \nu :: \text{List Nat} \mid \text{len} \nu = 2 \} \]

\textbf{data} List \(\alpha\) \textbf{where}

- \text{Nil} :: \{ List \(\alpha\) \mid \text{len} \nu = 0 \}
- \text{Cons} :: x :: \alpha \to \text{xs} :: \text{List} \(\alpha\)
  \[ \to \{ \nu :: \text{List} \alpha \mid \text{len} \nu = \text{len} \text{xs} + 1 \} \]

\textbf{measure} \text{len} :: \text{List} \(\alpha\) \to \text{Int}

\[ \text{len} \text{Nil} = 0 \]
\[ \text{len} \ (\text{Cons} \ _ \ \text{xs}) = \text{len} \ \text{xs} + 1 \]
Refinement Types

\[ e ::= x \mid e\ e \mid \lambda x:T.\ e \]

\[ \tau ::= \{\nu: B \mid P\} \]
\[ \mid x: \tau_1 \to \tau_2 \]
\[ \mid \alpha \]

\[ T ::= \tau \mid \forall \alpha.\ T \]

**Expressions**

**Types**

**Base types**

**Function types**

**Type variables**

**Type schemas**

---

\[ \Gamma \vdash n :: \{\nu: \text{Int} \mid \nu = n\} \]

\[ \Gamma \vdash x: T \vdash e :: T' \]

\[ \Gamma \vdash \lambda x:T.\ e :: T \to T' \]

\[ \Gamma \vdash e_1 :: x: T \to T' \quad \Gamma \vdash e_2 :: T \]

\[ \Gamma \vdash e_1\ e_2 :: T'[x \mapsto e_2] \]
// Insert x into a sorted list xs

insert :: x:e → xs:List e → List e

{List e | elems ν = elems xs U {x}}

insert x xs =

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

Expected
{List e | elems ν = elems xs U {x}}

but got
{List e | elems xs ⊆ elems ν}

measure elems :: List α → Set α

elems Nil = {}
elems (Cons x xs) = elems xs U {x}
Rejecting Incomplete Programs

// Insert x into a sorted list xs
insert :: x:e → xs:List e →
{List e | elems ν = elems xs U {x}}

insert x xs =
match xs with
  Nil → Nil
  Cons h t → ...

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

insert x xs =
match xs with
  Nil → Nil
  Cons h t → ...

measure elems :: List α → Set α
  elems Nil = {}
  elems (Cons x xs) = elems xs U {x}
// Insert x into a sorted list xs
insert :: x:e → xs:SList e →
{SList e | elems ν = elems xs U {x}}

insert x xs =
  match xs with
  Nil → ...
  Cons h t → ...

data SList e where
  Nil :: SList e
  Cons :: h:e →
  t:SList {ν:e | ν > h} →
  SList e
// Insert x into a sorted list xs
insert :: x:e → xs:SList e →
   {SList e | elems ∨ = elems xs ∪ {x}}

insert x xs =
   match xs with
   Nil → ...
   Cons h t → Cons h t

Expected {SList e | elems ∨ = elems xs ∪ {x}}
but got  {SList e | elems ∨ = elems t ∪ {h}}

measure elems :: List α → Set α
  elems Nil = {}
elems (Cons x xs) = elems xs ∪ {x}
// Insert x into a sorted list xs

insert :: x:e → xs:SList e →
        {SList e | elems ν = elems xs ∪ {x}}

        Cons h t

insert x xs =
    match xs with
        Nil → ...
        Cons h t → Cons h ??

data SList e where
    Nil :: SList e
    Cons :: h:e →
            t:SList {v:e | v > h} →
            SList e

measure elems :: List α → Set α
    elems Nil = {}
    elems (Cons x xs) = elems xs ∪ {x}
Synthesis From Refinement Types

I. top-down enumerative search
Synthesis From Refinement Types

\[ x_1 :: T_1; \ldots \vdash \text{??} :: T \]

I. top-down enumerative search

II. round-trip type checking
Synthesis From Refinement Types

\[ x_1 :: T_1; \ldots \Gamma \vdash ?? :: T \]

\[ \phi_1; \ldots \Gamma \vdash ?? :: T \]

\[ \text{if} \quad ?? :: \text{Bool} \mid \nu = P \]

\[ \text{then} \quad P \vdash \checkmark :: T \quad \text{else} \quad \neg P \vdash ?? :: T \]

I. top-down enumerative search

II. round-trip type checking

III. condition abduction
Example

\[
\begin{align*}
\text{Nil} &:: \{\text{List } a \mid \text{len } ν = 0\}; 0 ; 5 ; -5 \\
\text{zeros} &:: n:\text{Nat} \rightarrow \{\text{List Zero } \mid \text{len } ν = n\} \\
\text{replicate} &:: n:\text{Nat} \rightarrow x:a \rightarrow \{\text{List } a \mid \text{len } ν = n\} \\
\text{Cons} &:: x:a \rightarrow xs:\text{List } a \rightarrow \{\text{List } a \mid \text{len } ν = \text{len } xs + 1\}
\end{align*}
\]

\[\vdash ?? :: \{\text{List Neg } \mid \text{len } ν ≥ 5\}\]

\[\not\vdash \text{Nil} :: \{\text{List } a \mid \text{len } ν = 0\}\]

\[\not\vdash \text{Nil} :: \{\text{List } a \mid \text{len } ν = 0\} \rightarrow \text{len } ν ≥ 5\]
Example

Nil :: \(\{\text{List } a \mid \text{len } \nu = 0\}; \ 0 \ ; \ 5 \ ; -5\)

zeros :: \(n: \text{Nat} \rightarrow \{\text{List Zero } \mid \text{len } \nu = n\}\)

replicate :: \(n: \text{Nat} \rightarrow x:a \rightarrow \{\text{List } a \mid \text{len } \nu = n\}\)

Cons :: \(x:a \rightarrow xs: \text{List } a \rightarrow \{\text{List } a \mid \text{len } \nu = \text{len } xs + 1\}\)

\(-\) \(\text{zeros } \(?? \:: \text{Nat}\) :: \{\text{List Zero } \mid \text{len } \nu = 0\}\)

\{-\} \(\{\text{Int } \mid \nu < 0\}\)

\(-\) \(\{\text{Int } \mid \nu = 0\}\)
Example

Nil :: {List a | len ν = 0}; 0; 5; -5
zeros :: n:Nat → {List Zero | len ν = n}
replicate :: n:Nat → x:a → {List a | len ν = n}
Cons :: x:a → xs:List a → {List a | len ν = len xs + 1}

\[ \vdash ?? :: \{ \text{List Neg} | \text{len } ν ≥ 5 \} \]

\[ \text{zeros} \Rightarrow \text{replicate} \Rightarrow \text{Cons} \Rightarrow \text{Nil} \]

\[ \text{n = 0} \neq \text{len } ν = n \rightarrow \text{len } ν ≥ 5 \]
Example

\[
\text{Nil} :: \{\text{List a} \mid \text{len } \nu = 0\}; 0; 5; -5
\]
\[
\text{zeros} :: n:\text{Nat} \rightarrow \{\text{List Zero} \mid \text{len } \nu = n\}
\]
\[
\text{replicate} :: n:\text{Nat} \rightarrow x:a \rightarrow \{\text{List a} \mid \text{len } \nu = n\}
\]
\[
\text{Cons} :: x:a \rightarrow xs:\text{List a} \rightarrow \{\text{List a} \mid \text{len } \nu = \text{len } xs + 1\}
\]

\[
\vdash ?? :: \{\text{List Neg} \mid \text{len } \nu \geq 5\}
\]

\[
\text{replicate} \ (?? :: \text{Nat}) \ (?? :: a) :: \{\text{List a} \mid \text{len } \nu = ??\}
\]

\[
\{\text{Int} \mid \nu \geq 0\}
\]

\[
5 :: \{\text{Int} \mid \nu = 5\}
\]

\[
n = 5 \implies \text{len } \nu = n \rightarrow \text{len } \nu \geq 5
\]
Example

$\text{Nil} :: \{\text{List}\ a \mid \text{len}\ \nu = 0\}; 0; 5; -5$

$\text{zeros} :: n:\text{Nat} \to \{\text{List}\ \text{Zero} \mid \text{len}\ \nu = n\}$

$\text{replicate} :: n:\text{Nat} \to x:a \to \{\text{List}\ a \mid \text{len}\ \nu = n\}$

$\text{Cons} :: x:a \to xs:\text{List}\ a \to \{\text{List}\ a \mid \text{len}\ \nu = \text{len}\ xs+1\}$

$\vdash ?? :: \{\text{List}\ \text{Neg} \mid \text{len}\ \nu \geq 5\}$

$\text{replicate}\ (?? :: \text{Nat})\ (?? :: a) :: \{\text{List}\ a \mid \text{len}\ \nu = 5\}$

$\{\text{Int} \mid \nu < 0\}$

$5 :: \{\text{Int} \mid \nu = 5\}$

$0 :: \{\text{Int} \mid \nu = 0\}$

$5 :: \{\text{Int} \mid \nu = 5\}$

$-5 :: \{\text{Int} \mid \nu = -5\}$
Lab #11

• Synthesis of Sorting

```
data SList e where
  Nil :: SList e
  Cons :: h:e → t:SList {v:e | v > h} → SList e
sort :: List → SList
insert    ←→    insertion-sort
extractMin ←→    selection-sort!
```

we synthesize these as well of course!!