Deductive Synthesis

Type-Driven Synthesis

Lecture #12
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
Dimensions in program synthesis

[Gulwani 2010]

Rich Functional Specifications

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

Structural constraints
what is the space of programs to explore?

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

\[
P := [S] \\
\textbf{while} (P \neq []) \\
p = P.\text{dequeue}(); \\
\textbf{if} (\text{ground}(p) \land p([i]) = [o]) \\
\text{return } p; \\
P.\text{enqueue}(\text{expand}(p))
\]

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
Top-Down Propagation

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

\[ \text{If } \text{spec1} \models \perp, \text{ discard } E1@E2 \text{ altogether!} \]

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

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

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

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

fold $f \ acc \ x$  
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

[$\text{Feser, Chaudhuri, Dillig '15}$]

\[
\begin{align*}
\text{map } F \ x & \\
\rightarrow & \\
\downarrow & \\
\lambda y \cdot y + 1
\end{align*}
\]

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

select : Table \(\rightarrow\) [Col] \(\rightarrow\) Table

filter : Table \(\rightarrow\) (Row \(\rightarrow\) Bool) \(\rightarrow\) Table

Synthesizer

out.rows = in.rows \(\wedge\) out.cols < in.cols

out.rows < in.rows \(\wedge\) out.cols = in.cols

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

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

\[ \text{select : Table } \rightarrow \text{ [Col] } \rightarrow \text{ Table} \]

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

\[ \text{filter : Table } \rightarrow \text{ (Row } \rightarrow \text{ Bool) } \rightarrow \text{ Table} \]

select : Table → [Col] → Table

filter : Table → (Row → Bool) → Table

unsat

sat
“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 \rightarrow xs:List e \rightarrow List e
insert x xs =
  match xs with
  Nil \rightarrow Cons x xs ...
  ...

Expected
e
and got
List e
Hmm.

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

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

Conventional types are not enough.
Refinement Types

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

Nat

I am a type!

base types
Refinement Types

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

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

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

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

\[
\text{measure len :: List } \alpha \to \text{Int} \\
\text{len Nil} = 0 \\
\text{len (Cons } _ \alpha \text{ xs) = len xs + 1}
\]
Refinement Types

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

\[ \tau ::= \{ \nu: B \mid P \} \]

\[ \mid x: \tau_1 \rightarrow \tau_2 \]

\[ \mid \alpha \]

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

Expressions

Types

Type schemas

T-num

\[
\frac{(n = 0, 1, \ldots)}{\Gamma \vdash n :: \{ \nu: \text{Int} \mid \nu = n \}}
\]

T-var

\[
\frac{(x: T \in \Gamma)}{\Gamma \vdash x :: \{ \nu: T \mid \nu = x \}}
\]

T-abs

\[
\frac{\Gamma; x: T \vdash e :: T'}{\Gamma \vdash \lambda x: T. \; e :: T \rightarrow T'}
\]

T-app

\[
\frac{\Gamma \vdash e_1 :: x: T \rightarrow T' \quad \Gamma \vdash e_2 :: T}{\Gamma \vdash e_1 \; e_2 :: T'[x \mapsto e_2]}
\]
// Insert x into a sorted list xs
insert :: x:e → xs:List e → List e
                   {List e | elems ν = elems xs ∪ {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)

measure elems :: List α → Set α
  elems Nil = {}
  elems (Cons x xs) = elems xs ∪ {x}
data SList e where
  Nil :: SList e
  Cons :: h:e →
        t:SList \{v:e | v > h\} →
        SList e

// Insert x into a sorted list xs
insert :: x:e → xs:SList e → SList e
           \{SList e | elems ν = elems xs ∪ \{x\}\}
Rejecting Incomplete Programs?

// Insert x into a sorted list xs
insert :: x:e \rightarrow xs:SList e \rightarrow SList e
    \{SList e \mid \text{elems } \nu = \text{elems } xs \cup \{x\}\}

insert x xs =
    match xs with
        Nil \rightarrow Nil
        Cons h t \rightarrow ...
Rejecting Incomplete Programs?

```haskell
// Insert x into a sorted list xs
insert :: x:e -> xs:SLList e -> SLList 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 ∪ {x}

{SLList e | elems ν = elems xs ∪ {x}}
{SLList e | elems ν = elems Nil ∪ {x}}
{SLList e | elems ν = {x}}
Expected {SLList e | elems ν = {x}}
and got {SLList e | elems ν = {}}
```
http://tiny.cc/synquid
Synthesis From Refinement Types

$\Gamma \vdash ? :: T$

$I. \text{top-down enumerative search}$
Synthesis From Refinement Types

\[ \chi_1 :: T_1; \ldots ; \phi_1; \ldots \vdash ?? :: T \]

I. top-down enumerative search

II. round-trip type checking
Synthesis From Refinement Types

\[ \Gamma \vdash ??? :: T \]

I. top-down enumerative search

II. round-trip type checking

III. condition abduction
Example

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

\[\vdash ?? :: \{\text{List} \ \text{Neg} \mid \text{len} \ \nu \geq 5\}\]
Lab #10

• Synthesis of Sorting

\[
\text{data } \text{SList } e \text{ where }
\]
\[
\text{Nil } :: \text{ SList } e
\]
\[
\text{Cons } :: h : e \rightarrow
\]
\[
t : \text{SList} \{v : e \mid v > h\} \rightarrow
\]
\[
\text{SList } e
\]

\[
\text{sort } :: \text{ List } \rightarrow \text{ SList}
\]

\[
\text{insert} \quad \rightarrow \quad \text{insertion-sort}
\]

\[
\text{extractMin} \quad \rightarrow \quad \text{selection-sort}!
\]

we synthesize these as well of course!!