Lecture 13
Deductive Synthesis (II)

Nadia Polikarpova

Before we begin

Behavioral constraints

User input that specifies what the program should *do*

- examples, reference implementation, assertions, pre-/post-conditions, ...

Without b.c., output is garbage

Structural constraints

User input that specifies what the program should *look like*

- grammars, sketches, sets of components, max size / max # of local variables, ...

Without s.c. synthesis is slow

Some s.c. are hard-coded or semi-hard-coded

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 transformations / decompositions

Deductive synthesis

The synthesis problem:

- Find x such that $Q(a, x)$ whenever $P(a)$

Using semantic-preserving transformations, gradually rewrite the problem above into:

- Find T such that T whenever $P(a)$
- where T is a term that does not mention x

Two approaches

Transformation rules

today

A set of inference rules for decomposing a synthesis problem into simpler problems

- Axioms (terminal rules) for solving elementary problems
- Rules have side conditions to prove

Depth- or best-first search in the space of derivations

[Kneuss et al.'13]

Theorem proving

Tuesday

Extract the program from a constructive proof of $\exists x. \forall a. P(a) \Rightarrow Q(a, x)$

- Instead of inventing custom rules, reuse an existing theorem prover
- ... but augment its rules with term extraction
- Reuse the prover's search strategy!

[Manna, Waldinger'80]

Leon is...

→ A deductive synthesis framework

- with powerful terminal rules that use inductive synthesis
  - Symbolic Term Exploration
  - Condition Abduction
- cost-based search for derivations

A synthesis-aided language

- functional language + choose
- interaction model

The Leon synthesis framework

Deductive synthesis is good at figuring out high-level structure

Inductive synthesis is good at generating straight-line fragments

Idea: combine them!

- first decompose the synthesis problem using deductive rules
- then use inductive synthesizers as terminal rules

Synthesis problem

inputs

outputs

$$[\![\bar{a}\ \langle \Pi \rhd \phi \rangle\ \bar{x}]\!]$$

path condition

synthesis predicate

c.f. deductive synthesis problem

- Find x such that $Q(a, x)$ whenever $P(a)$

Synthesis judgment

Instead of transforming synthesis problems directly, Leon transforms synthesis judgments:

$$\underbrace{[\![\bar{a}\ \langle \Pi \rhd \phi \rangle\ \bar{x}]\!]}_{\text{synthesis problem}} \vdash \ \langle P, \bar{T} \rangle$$

precondition

program terms

Meaning

- relation refinement

$$\Pi \wedge P \vDash \phi[\bar{x} \mapsto \bar{T}]$$

- domain preservation

$$\Pi \wedge (\exists \bar{x}. \phi) \vDash P$$

Inference rules

one-point

$$\frac{x_0 \notin \mathrm{vars}(t) \qquad [\![\bar{a}\ \langle \Pi \rhd \phi[x_0 \mapsto t] \rangle\ \bar{x}]\!] \vdash \langle P, \bar{T} \rangle}{[\![\bar{a}\ \langle \Pi \rhd x_0 = t \land \phi \rangle\ x_0, \bar{x}]\!] \vdash \langle P, (t, \bar{T}) \rangle}$$

case-split

$$\frac{[\![\bar{a}\ \langle \Pi \rhd \phi_1 \rangle\ \bar{x}]\!] \vdash \langle P_1, \bar{T}_1 \rangle \quad [\![\bar{a}\ \langle \Pi \land \neg P_1 \rhd \phi_2 \rangle\ \bar{x}]\!] \vdash \langle P_2, \bar{T}_2 \rangle}{[\![\bar{a}\ \langle \Pi \rhd \phi_1 \lor \phi_2 \rangle\ \bar{x}]\!] \vdash \langle P_1 \lor P_2, \text{if } P_1 \text{ then } \overline{T_1} \text{ else } \overline{T_2} \rangle}$$

list-rec

$$\frac{\begin{array}{c}\Pi[\mathrm{l} \mapsto h :: t] \Rightarrow \Pi[\mathrm{l} \mapsto t] \qquad [\![\bar{a}\ \langle \Pi[l \mapsto \emptyset] \rhd \phi[l \mapsto \emptyset] \rangle\ x]\!] \vdash \langle \top, T_1 \rangle \\ [\![h, t, r, \bar{a}\ \langle \Pi[\mathrm{l} \mapsto h :: t] \land \phi[l \mapsto t, x \mapsto r] \rhd \phi[l \mapsto h :: t] \rangle\ x]\!] \vdash \langle \top, T_2 \rangle\end{array}}{[\![l, \bar{a}\ \langle \Pi \rhd \phi \rangle\ x]\!] \vdash \langle P, \mathrm{rec}(l, \bar{a}) \rangle}$$

Complex terminal rules

Symbolic Term Exploration

- Essentially Sketch

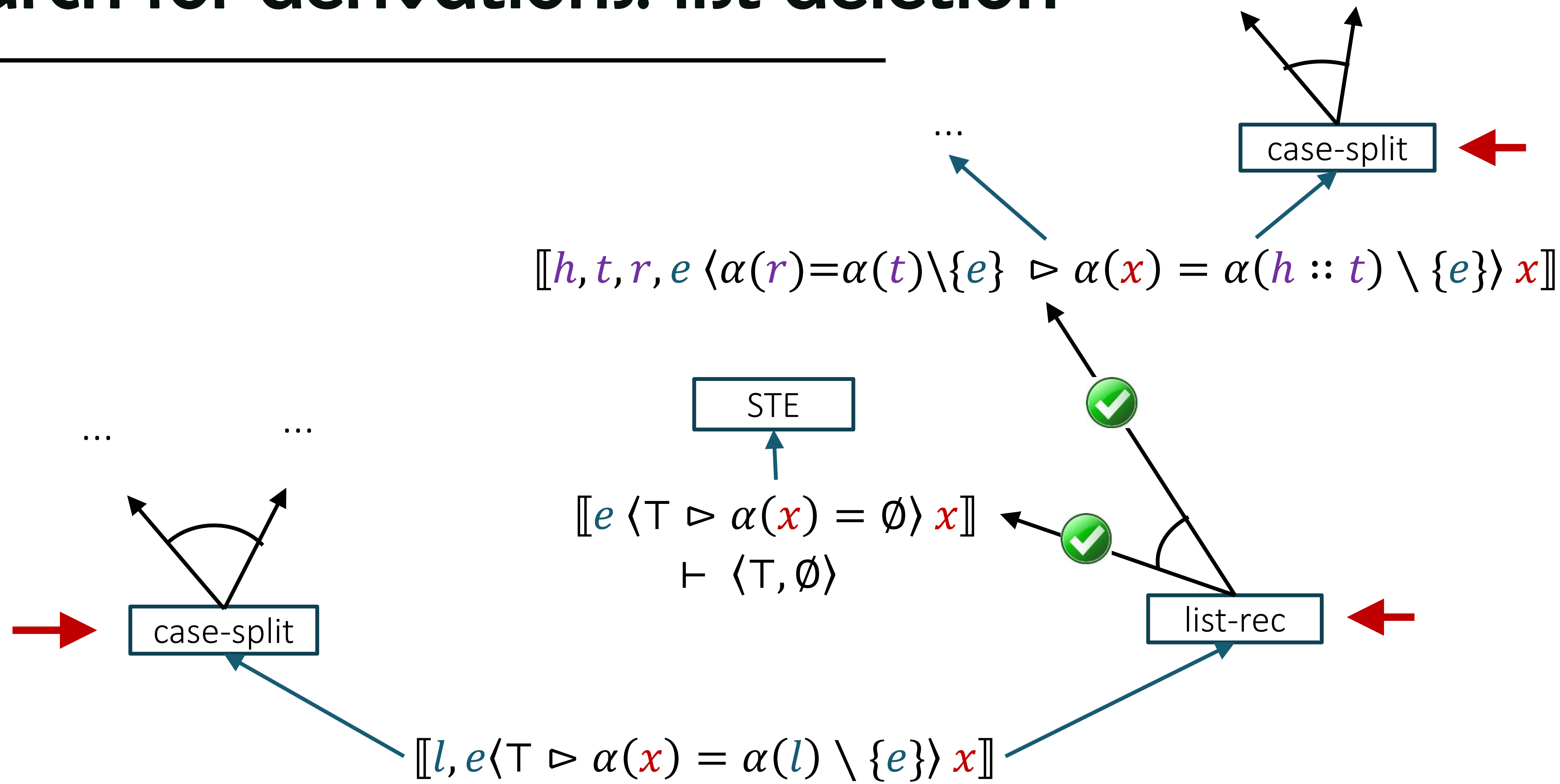
$$\frac{\mathrm{STE}(\Pi, \phi) = \bar{T}}{[\![\bar{a}\ \langle \Pi \rhd \phi \rangle\ \bar{x}]\!] \vdash \langle \top, \bar{T} \rangle}$$

Condition Abduction

- Essentially EUSolver

$$\frac{\mathrm{CA}(\Pi, \phi) = \bar{T}}{[\![\bar{a}\ \langle \Pi \rhd \phi \rangle\ \bar{x}]\!] \vdash \langle \top, \bar{T} \rangle}$$

Search for derivations: list deletion



Modern deductive synthesizers

Combine deductive synthesis with modern techniques

- automated reasoning
- inductive synthesis

Are still mostly interactive!

- search in the space of derivations is generally hard
- even a little user guidance goes a long way
- Examples: Fiat, Bellmania

Fiat

[Delaware et al. ‘15]

Input: a high-level spec, e.g. a database query

```
query NumOrders (author: string) : nat :=
  Count (For (o in Orders) (b in Books)
         Where (author = b!Author)
         Where (b!ISBN = o!ISBN)
         Return ())
```

Iteratively refined into efficient implementations via automated tactics

```
meth NumOrders (p: rep , author: string) : nat :=
  let (books, orders) := p in
  ret (books, orders,
       fold_left
       (\ count tup .
         count + bcount orders (Some tup!ISBN, []))
        (bnd books (Some author, None, [])) 0)
```

Bellmania

[Itzhaky et al. '16]

Deriving parallel divide-and-conquer implementations of dynamic programming algorithms

- start from a naive implementation
- at each step, the user picks a tactic to transform the program
- tool checks side-conditions
  - i.e. that the transformation produces an equivalent program
  - but has no idea where the whole thing is going