Lecture 14
Enumeration with Deduction.
Type Systems.

Nadia Polikarpova

Deductive reasoning for synthesis

Main idea: Look for the proof to find the program

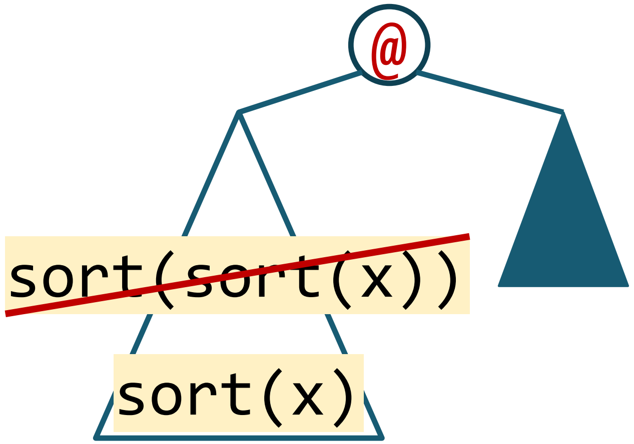
- The space of valid program derivations is smaller than the space of all programs
- The result is provably correct!

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

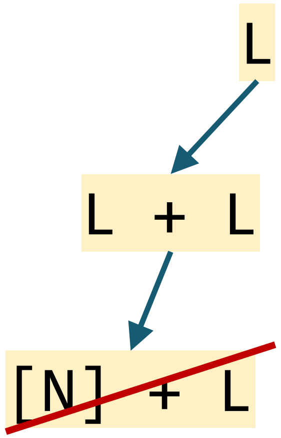
When can we discard a subprogram?

It’s equivalent to something we have already explored



Equivalence reduction

No matter what we combine it with, it cannot fit the spec

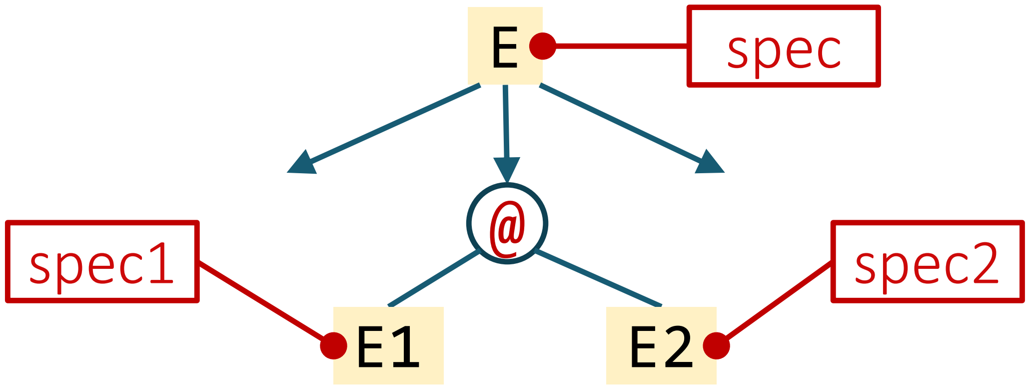


[] → []
...

Top-down propagation

Top-down propagation

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

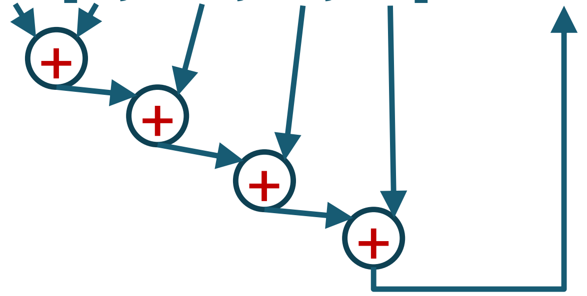


If `spec1 = ⊥`, discard `E1 @ E2` altogether!

λ²: TDP for list combinators

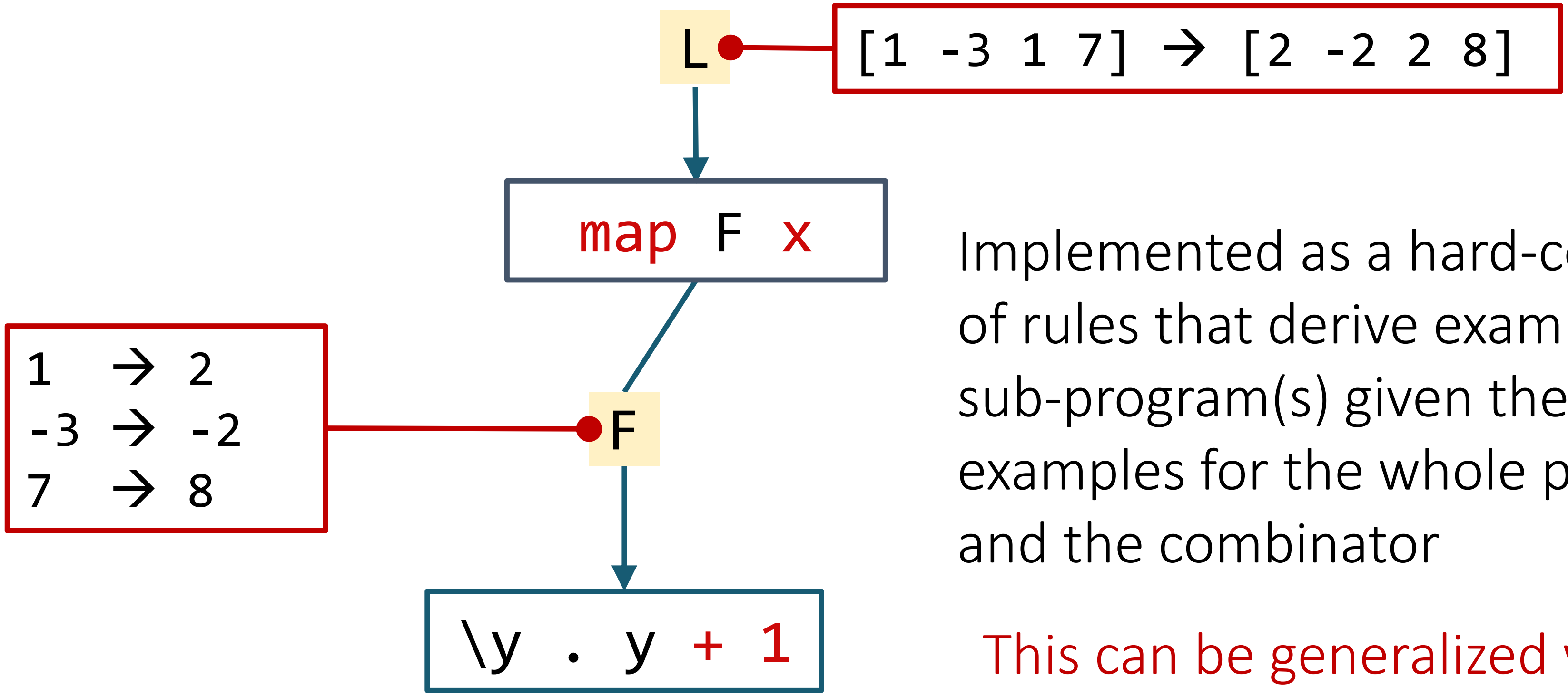
[Feser, Chaudhuri, Dillig '15]

```
map f x             map (\y . y + 1) [1, -3, 1, 7] → [2, -2, 2, 8]

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

fold f acc x        fold (\y z . y + z) 0 [1, -3, 1, 7] → 6
```




```
                    fold (\y z . y + z) 0 [] → 0
```

λ²: TDP for list combinators



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!

Morpheus: TDP with deduction

[Feng et al'17]

Input-output examples

ID	Name	Age
1	Alice	8
2	Bob	18
3	Tom	12

→

ID	Name	Age
2	Bob	18
3	Tom	12

Synthesizer

straight-line program

Components

with partial specifications!

```
select : Table → [Col] → Table                out.rows = in.rows
                                              && out.cols < in.cols

filter : Table → (Row → Bool) → Table         our.rows < in.rows
                                              && out.cols = in.cols
```

Morpheus: TDP with deduction

[Feng et al'17]

x

ID	Name	Age
1	Alice	8
2	Bob	18
3	Tom	12

→

y

ID	Name	Age
2	Bob	18
3	Tom	12

??

~~select~~ ?? x

filter ?? x

$$\exists x\ y: y.\mathrm{rows} = x.\mathrm{rows} \wedge y.\mathrm{cols} < x.\mathrm{cols}$$
$$\wedge\, x.\mathrm{rows} = 3 \wedge x.\mathrm{cols} = 3$$
$$\wedge\, y.\mathrm{rows} = 2 \wedge y.\mathrm{cols} = 3$$

SMT

UNSAT

$$\exists x\ y: y.\mathrm{rows} < x.\mathrm{rows} \wedge y.\mathrm{cols} = x.\mathrm{cols}$$
$$\wedge\, x.\mathrm{rows} = 3 \wedge x.\mathrm{cols} = 3$$
$$\wedge\, y.\mathrm{rows} = 2 \wedge y.\mathrm{cols} = 3$$

SMT

SAT

```
select : Table → [Col] → Table
filter : Table → (Row → Bool) → Table
```

```
out.rows = in.rows && out.cols < in.cols
our.rows < in.rows && out.cols = 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 type checkers for *expressive type systems* can do 3 as well

Type Systems

What is a type system?

Formalization of a typing discipline of a language

- independently of a particular type checking algorithm (more or less)
- if a type checking algorithm exists, type system is *decidable*

Deductive system for proving facts about programs and types

- defined using *inference rules* over *judgments*

environment / context
(declares free variables of $\mathfrak{I}$)

$$\Gamma \vdash \mathfrak{I}$$

assertion
for example:

$e :: T$	"e has type T"
T	"T is well-formed"
$T' <: T$	"T' is a subtype of T"

typically:

$$x_1 : T_1, \dots, x_n : T_n$$

Simple type system

$e \ ::= \text{true} \mid \text{false} \mid n \mid e + e$

Syntax of terms (programs)

$T \ ::= \text{Bool} \mid \text{Int}$

Syntax of types

Inference Rules

$$\text{T-true} \quad \frac{}{\Gamma \vdash \text{true} :: \text{Bool}}$$

$$\text{T-false} \quad \frac{}{\Gamma \vdash \text{false} :: \text{Bool}}$$

$$\text{T-num} \quad \frac{(n = 0, 1, \ldots)}{\Gamma \vdash n :: \text{Int}}$$

label → T-plus

$$\text{T-plus} \quad \frac{\Gamma \vdash e_1 :: \text{Int} \qquad \Gamma \vdash e_2 :: \text{Int}}{\Gamma \vdash e_1 + e_2 :: \text{Int}}$$

← premises

← conclusion

Type derivations

$\emptyset \vdash 1 + 2 :: \mathrm{Int}$ is a valid judgment, because....

$$\dfrac{\dfrac{}{\emptyset \vdash 1 :: \mathrm{Int}}\,\text{T-num} \qquad \dfrac{}{\emptyset \vdash 2 :: \mathrm{Int}}\,\text{T-num}}{\emptyset \vdash 1 + 2 :: \mathrm{Int}}\,\text{T-plus}$$

We say that $1 + 2$ is *well-typed* (and has type Int)

Type derivations

$\emptyset \vdash 1 + \mathrm{true} :: \mathrm{Int}$ is not a valid judgment, because....

$$\text{T-plus}\ \dfrac{\text{T-num}\ \dfrac{}{\emptyset \vdash 1 :: \mathrm{Int}} \qquad \emptyset \vdash \mathrm{true} :: \mathrm{Int}}{\emptyset \vdash 1 + \mathrm{true} :: \mathrm{Int}}$$

We say that $1 + \mathrm{true}$ is *ill-typed* (or *not typable*)

Type checking vs inference

The problem of discovering the derivation of $\Gamma \vdash e :: T$ is called *type reconstruction* or *type checking*

The problem of discovering the type T such that there exists a derivation of $\Gamma \vdash e :: T$ is called *type inference*

If we have a mechanism for inference, we can also do checking

- How?

The goal of inference is to free the programmer from writing *type annotations*

Function types

$$e \ ::= \text{true} \mid \text{false} \mid n \mid e + e$$

Syntax of terms (programs)

$$\mid x \mid e\ e \mid \lambda x{:}T.\,e$$

(variable, application, lambda abstraction)

$$T \ ::= \text{Bool} \mid \text{Int}$$

(basic types)

Syntax of types

$$\mid T_1 \to T_2$$

(function types)

T-var
$$\frac{(x{:}T \in \Gamma)}{\Gamma \vdash x :: T}$$

T-abs
$$\frac{\Gamma; x{:}T \vdash e :: T'}{\Gamma \vdash \lambda x{:}T.\,e :: T \to T'}$$

T-app
$$\frac{\Gamma \vdash e_1 :: T \to T' \qquad \Gamma \vdash e_2 :: T}{\Gamma \vdash e_1\ e_2 :: T'}$$

Exercise

Infer the type of $(\lambda x\colon \mathrm{Int}.\, x + x)\ 5$ in $\emptyset$ using the rules

T-num $$\frac{(n = 0, 1, \dots)}{\Gamma \vdash n :: \mathrm{Int}}$$

T-plus $$\frac{\Gamma \vdash e_1 :: \mathrm{Int} \qquad \Gamma \vdash e_2 :: \mathrm{Int}}{\Gamma \vdash e_1 + e_2 :: \mathrm{Int}}$$

T-var $$\frac{(x\colon T \in \Gamma)}{\Gamma \vdash x :: T}$$

T-abs $$\frac{\Gamma; x\colon T \vdash e :: T'}{\Gamma \vdash \lambda x\colon T.\, e :: T'}$$

T-app $$\frac{\Gamma \vdash e_1 :: T \to T' \qquad \Gamma \vdash e_2 :: T}{\Gamma \vdash e_1\ e_2 :: T'}$$

Type checking vs inference

In type inference, we interpret rules left-to-top-to-right:

$$\text{T-app}\ \frac{\dfrac{\cdots}{\emptyset \vdash (\lambda x{:}\,\mathrm{Int}.\, x + x) ::} \qquad \dfrac{\cdots}{\emptyset \vdash 5 ::}}{\emptyset \vdash (\lambda x{:}\,\mathrm{Int}.\, x + x)\ 5 ::}$$

Type information flows leaves-to-root (“bottom-up”)

That’s why we need type annotations on lambda arguments!

Type annotations

$$\text{T-abs'}\quad \frac{\Gamma; x : ? \vdash e :: }{\Gamma \vdash \lambda x.\, e :: ? \to ?}$$

Without the annotation, we don't know what type to give x while analyzing e

If we were doing checking (not inference), this is not a problem:

$$\text{T-abs''}\quad \frac{\Gamma; x : T_1 \vdash e :: T_2}{\Gamma \vdash \lambda x.\, e :: T_1 \to T_2}$$

Bidirectional type-system

Rules differentiate between type inference and checking

$\Gamma \vdash e \uparrow T$

"e generates T in Γ"

$\Gamma \vdash e \downarrow T$

"e checks against T in Γ"

I-var $\dfrac{(x : T \in \Gamma)}{\Gamma \vdash x \uparrow T}$

C-abs $\dfrac{\Gamma; x : T_1 \vdash e \downarrow T_2}{\Gamma \vdash \lambda x. e \downarrow T_1 \rightarrow T_2}$

C-I $\dfrac{\Gamma \vdash e \uparrow T' \quad \Gamma \vdash T = T'}{\Gamma \vdash e \downarrow T}$

Can we *infer* the type of $(\lambda x. x + x)\ 5$ using bidirectional rules?

C-app $\dfrac{\Gamma \vdash e_2 \uparrow T \qquad \Gamma \vdash e_1 \downarrow T \rightarrow T'}{\Gamma \vdash e_1\ e_2 \downarrow T'}$

Polymorphism (aka “generics”)

$$e \;::=\; \text{true} \mid \text{false} \mid n \mid e + e$$
$$\mid x \mid e\; e \mid \lambda x{:}T.\, e$$

Terms

$$T \;::=\; \text{Bool} \mid \text{Int}$$ (basic types)

$$\mid T_1 \to T_2$$ (function types)

$$\mid \alpha$$ (type variables)

Types

$$S \;::=\; T \mid \forall \alpha.\, S$$

Type schemas

T-gen $$\frac{\Gamma; \alpha \vdash e :: S}{\Gamma \vdash e :: \forall \alpha.\, S}$$

T-inst $$\frac{\Gamma \vdash e :: \forall \alpha.\, S \qquad \Gamma \vdash T}{\Gamma \vdash e :: S[\alpha \mapsto T]}$$

Example

Let's infer the type of $id\ 5$ in Γ
where $\Gamma = [\mathrm{id} : \forall\alpha.\alpha \ \rightarrow \alpha]$

using the following rules:

T-num $\dfrac{(n = 0, 1, \dots)}{\Gamma \vdash n :: \mathrm{Int}}$

T-var $\dfrac{(x{:}T \in \Gamma)}{\Gamma \vdash x :: T}$

T-app $\dfrac{\Gamma \vdash e_1 :: T \rightarrow T' \quad \Gamma \vdash e_2 :: T}{\Gamma \vdash e_1\ e_2 :: T'}$

T-gen $\dfrac{\Gamma; \alpha \vdash e :: S}{\Gamma \vdash\ e :: \forall\alpha.S}$

T-inst $\dfrac{\Gamma \vdash e :: \forall\alpha.S \qquad \Gamma \vdash T}{\Gamma \vdash\ e :: S[\alpha \mapsto T]}$