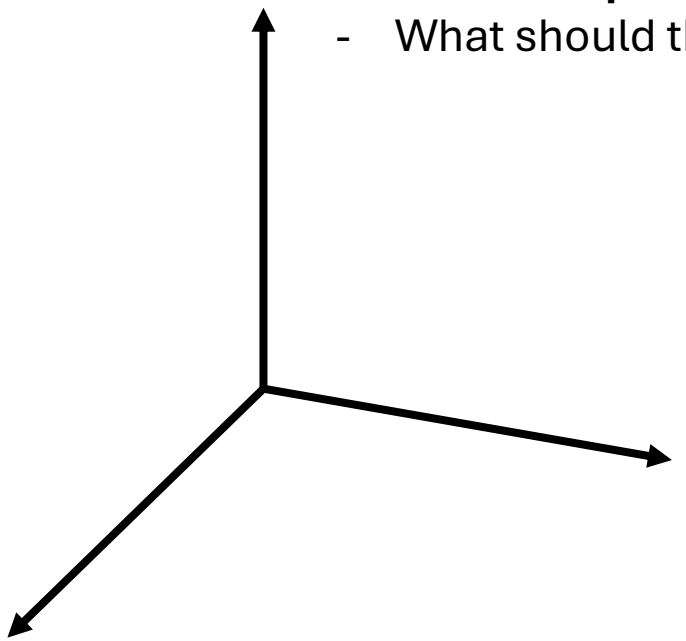
Machine Programming

Lecture 3 – Type Systems and Top-Down Enumerative Synthesis

Ziyang Li

Dimensions in Program Synthesis

Behavioral Specification
- What should the program do?



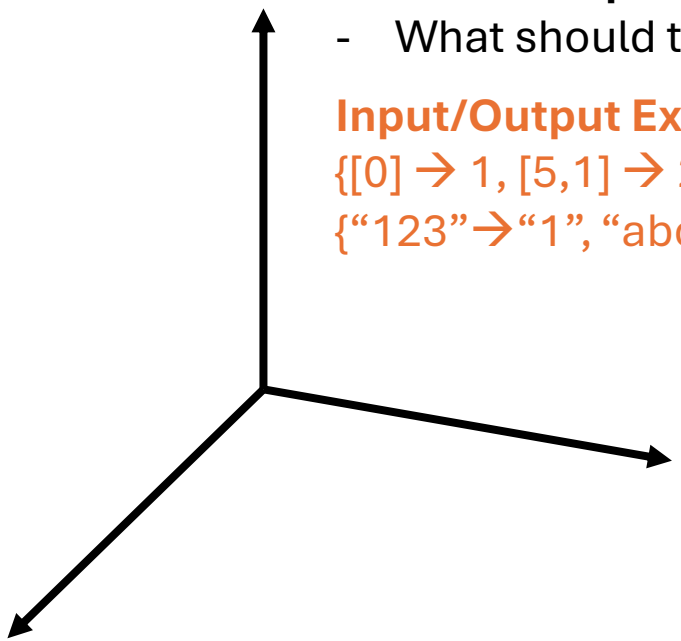
Synthesis Strategy
- How do we find such a program?

Structural Specification
- What is the space of the programs?

Last Week

Behavioral Specification
- What should the program do?

Input/Output Examples
{[0] → 1, [5,1] → 2}
{“123”→“1”, “abc”→“a”}



Synthesis Strategy
- How do we find such a program?

Enumeration
- Enumerating all programs with a grammar
- Bottom-up vs top-down

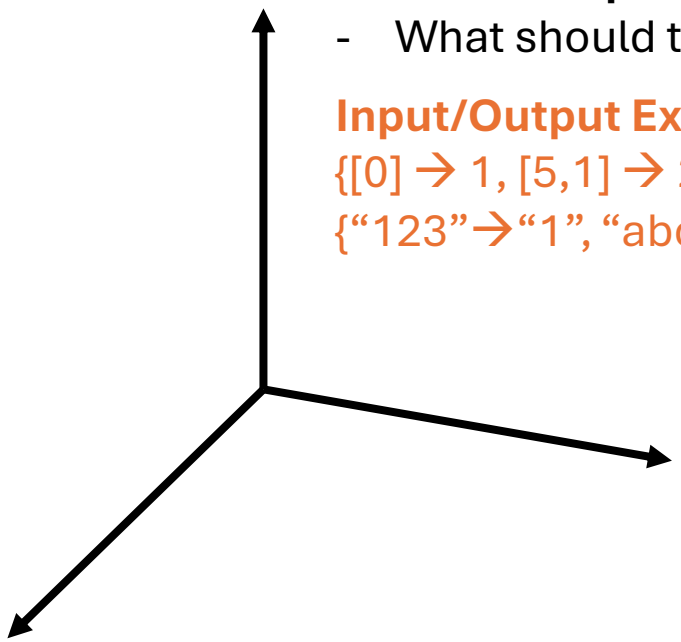
Structural Specification
- What is the space of the programs?

Context-Free / Regular Tree Grammar
Expr e ::= c | e + e | e * e

Last Week

Behavioral Specification
- What should the program do?

Input/Output Examples
{[0] → 1, [5,1] → 2}
{“123”→“1”, “abc”→“a”}



Synthesis Strategy
- How do we find such a program?

Enumeration
- Enumerating all programs with a grammar
- Bottom-up vs top-down

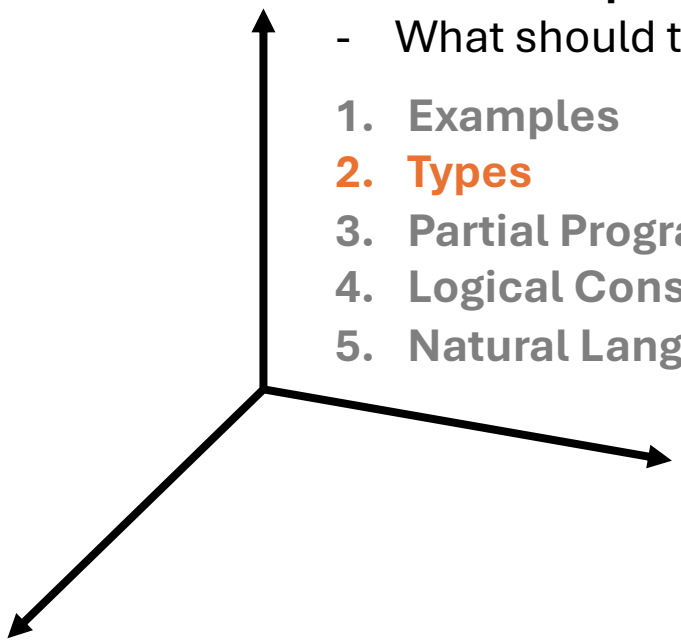
Structural Specification
- What is the space of the programs?

Context-Free / Regular Tree Grammar
Expr e ::= c | e + e | e * e

Today

Behavioral Specification
- What should the program do?

1. **Examples**
2. **Types**
3. **Partial Programs**
4. **Logical Constraints**
5. **Natural Language**



Synthesis Strategy
- How do we find such a program?

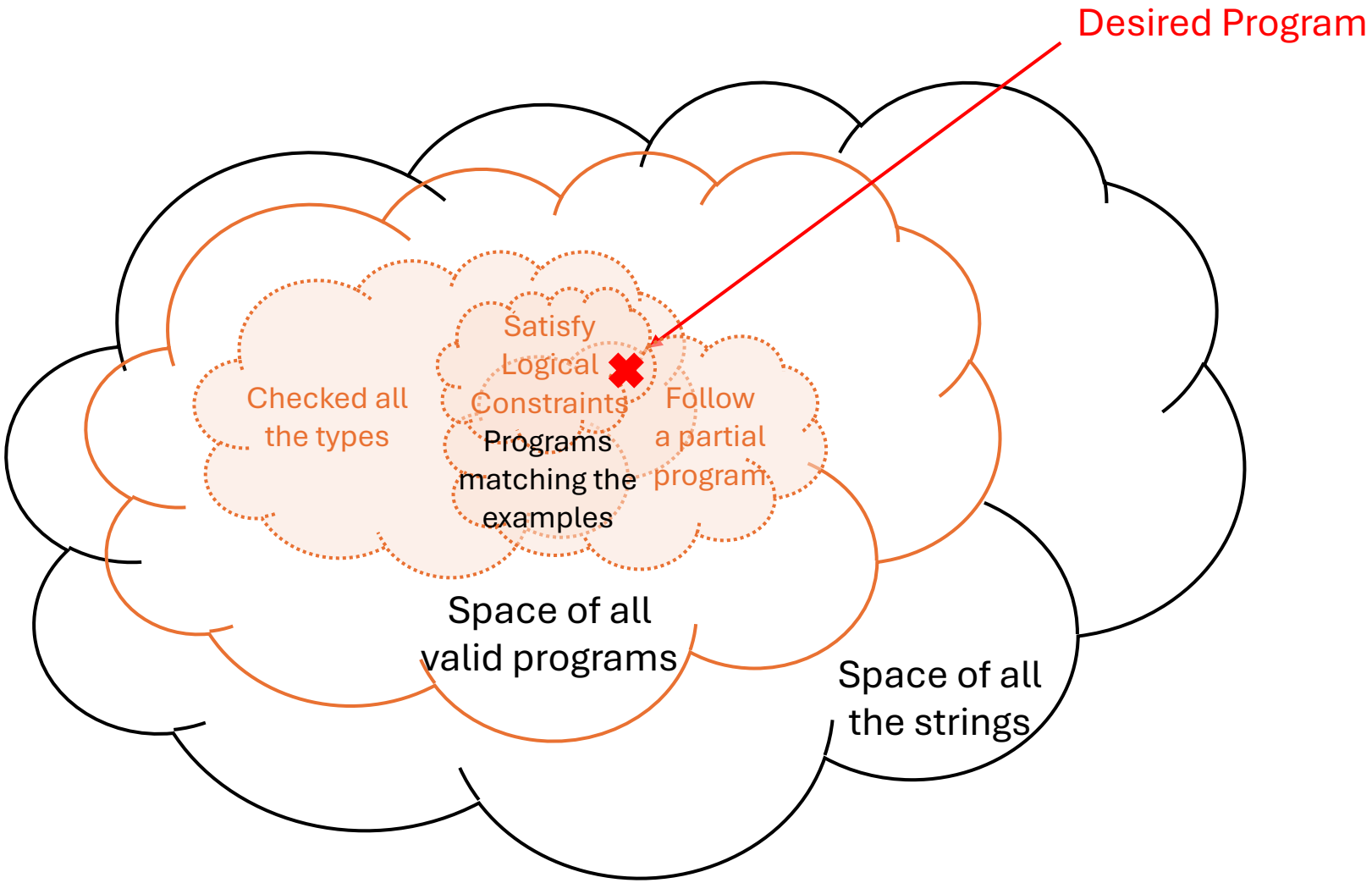
Enumeration
- Enumerating all programs with a grammar
- Bottom-up vs top-down

Structural Specification
- What is the space of the programs?

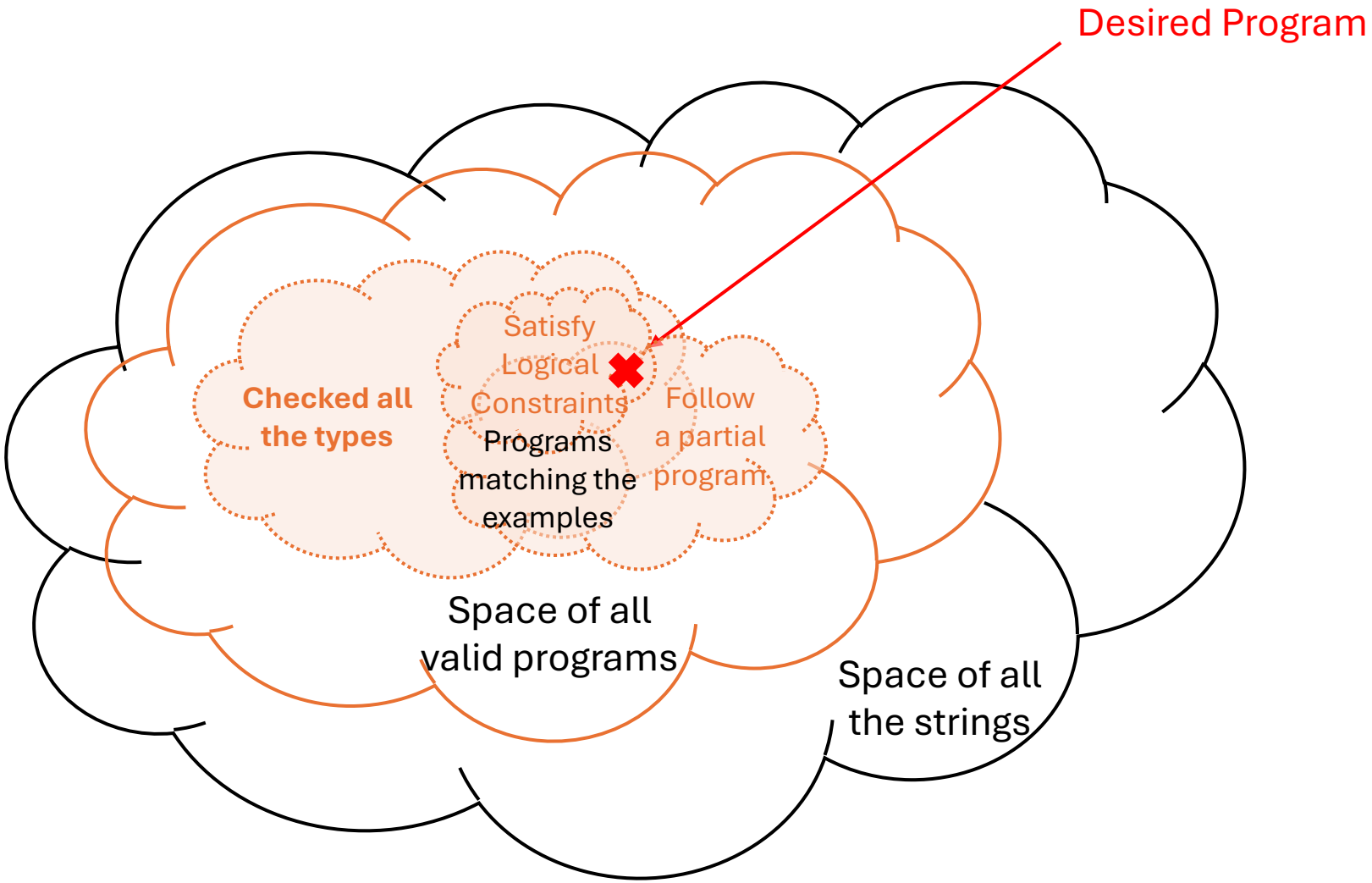
Context-Free / Regular Tree Grammar

Expr e ::= c | e + e | e * e

High Level Picture



High Level Picture



Specification: Types

- Fundamentals
  - Example: A Domain Specific Language (DSL) on List Manipulation
  - Type System
  - Type Checking
  - Pruning Top-Down Search with Types
- Advanced
  - Polymorphic Types
  - Refinement Types

List Manipulation DSL

Synthesis Goal:

Input

```
[ [71, 75, 83],
  [90, 87, 95],
  [68, 77, 80] ]
```

Output

```
[ [75, 83],
  [90, 95],
  [77, 80] ]
```

List Manipulation DSL

Synthesis Goal:

Input

```
[ [71, 75, 83],
  [90, 87, 95],
  [68, 77, 80] ]
```

Output

```
[ [75, 83],
  [90, 95],
  [77, 80] ]
```

`dropmins`: for each inner list corresponding to a list of grades, drop the lowest grade

List Manipulation DSL

`dropmins`: for each inner list corresponding to a list of grades, drop the lowest grade

Input	Output

`[ [71, 75, 83],`
`  [90, 87, 95],`
`  [68, 77, 80] ]`
	`[ [75, 83],`
	`  [90, 95],`
	`  [77, 80] ]`

```
def dropmins(input: List[List[int]]) -> List[List[int]]:
  # implementation
```

List Manipulation DSL

`dropmins`: for each inner list corresponding to a list of grades, drop the lowest grade

Input

```
[ [71, 75, 83],
  [90, 87, 95],
  [68, 77, 80] ]
```

Output

```
[ [75, 83],
  [90, 95],
  [77, 80] ]
```

```
def dropmins(input: …) -> …:
  outputs = []
  for grades in input:
    dropmin = []
    min_num = min(grades)
    for grade in grades:
      if grade > min_num:
        dropmin.append(grade)
    outputs.append(dropmin)
  return outputs
```

List Manipulation DSL

`dropmins`: for each inner list corresponding to a list of grades, drop the lowest grade

Input

```
[ [71, 75, 83],
  [90, 87, 95],
  [68, 77, 80] ]
```

Output

```
[ [75, 83],
  [90, 95],
  [77, 80] ]
```

```
def dropmins(input: …) -> …:
  outputs = []
  for grades in input:
    dropmin = []
    min_num = min(grades)
    for grade in grades:
      if grade > min_num:
        dropmin.append(grade)
    outputs.append(dropmin)
  return outputs
```

```
def dropmins(input: …) -> …:
  outputs = []
  for grades in input:
    dropmin = [g for g in grades if g > min(grades)]
    outputs.append(dropmin)
  return outputs
```

List Manipulation DSL

`dropmins`: for each inner list corresponding to a list of grades, drop the lowest grade

Input

```
[ [71, 75, 83],
  [90, 87, 95],
  [68, 77, 80] ]
```

Output

```
[ [75, 83],
  [90, 95],
  [77, 80] ]
```

```
def dropmins(input: …) -> …:
  outputs = []
  for grades in input:
    dropmin = []
    min_num = min(grades)
    for grade in grades:
      if grade > min_num:
        dropmin.append(grade)
    outputs.append(dropmin)
  return outputs
```

```
def dropmins(input: …) -> …:
  outputs = []
  for grades in input:
    dropmin = [
      g for g in grades
      if g > min(grades)
    ]

    outputs.append(dropmin)
  return outputs
```

List Manipulation DSL

`dropmins`: for each inner list corresponding to a list of grades, drop the lowest grade

Input

```
[ [71, 75, 83],
  [90, 87, 95],
  [68, 77, 80] ]
```

Output

```
[ [75, 83],
  [90, 95],
  [77, 80] ]
```

```
def dropmins(input: …) -> …:
  outputs = []
  for grades in input:
    dropmin = []
    min_num = min(grades)
    for grade in grades:
      if grade > min_num:
        dropmin.append(grade)
    outputs.append(dropmin)
  return outputs
```

```
def dropmins(input: …) -> …:
  outputs = []
  for grades in input:
    dropmin = [
      g for g in grades
      if g > min(grades)
    ]
    outputs.append(dropmin)
  return outputs
```

```
def dropmins(input: …) -> …:
  outputs = [
    [
      g for g in grades
      if g > min(grades)
    ]
    for grades in input
  ]
  return outputs
```

List Manipulation DSL

`dropmins`: for each inner list corresponding to a list of grades, drop the lowest grade

Input

```
[ [71, 75, 83],
  [90, 87, 95],
  [68, 77, 80] ]
```

Output

```
[ [75, 83],
  [90, 95],
  [77, 80] ]
```

```
def dropmins(input: …) -> …:
  outputs = []
  for grades in input:
    dropmin = []
    min_num = min(grades)
    for grade in grades:
      if grade > min_num:
        dropmin.append(grade)
    outputs.append(dropmin)
  return outputs
```

```
def dropmins(input: …) -> …:
  outputs = []
  for grades in input:
    dropmin = [
      g for g in grades
      if g > min(grades)
    ]
    outputs.append(dropmin)
  return outputs
```

```
def dropmins(input: …) -> …:
  return [
    [
      g for g in grades
      if g > min(grades)
    ]
    for grades in input
  ]
```

List Manipulation DSL

`dropmins`: for each inner list corresponding to a list of grades, drop the lowest grade

Input

```
[ [71, 75, 83],
  [90, 87, 95],
  [68, 77, 80] ]
```

Output

```
[ [75, 83],
  [90, 95],
  [77, 80] ]
```

```
def dropmins(input: …) -> …:
  return [
    [
      g for g in grades
      if g > min(grades)
    ]
    for grades in input
  ]
```

```
def dropmins(input: …) -> …:
  return [

    filter(grades, lambda g: g > min(grades))


    for grades in input
  ]
```

List Manipulation DSL

`dropmins`: for each inner list corresponding to a list of grades, drop the lowest grade

Input

```
[ [71, 75, 83],
  [90, 87, 95],
  [68, 77, 80] ]
```

Output

```
[ [75, 83],
  [90, 95],
  [77, 80] ]
```

```
def dropmins(input: …) -> …:
  return [
    filter(
      grades,
      lambda g: g > min(grades))
    for grades in input
  ]
```

```
def dropmins(input: …) -> …:
  return map(input, lambda grades:
    filter(grades, lambda g:
      g > min(grades)
    )
  )
```

List Manipulation DSL

`dropmins`: for each inner list corresponding to a list of grades, drop the lowest grade

Input

```
[ [71, 75, 83],
  [90, 87, 95],
  [68, 77, 80] ]
```

Output

```
[ [75, 83],
  [90, 95],
  [77, 80] ]
```

```
def dropmins(input: …) -> …:
  return map(input, lambda grades:
    filter(grades, lambda g:
      g > min(grades)
    )
  )
```

```
def dropmins(input: …) -> …:
  return map(input, lambda grades:
    filter(grades, lambda g:
      g > reduce(grades, 0, lambda acc, cur:
        acc if acc < cur else cur
      )
    )
  )
```

```
def dropmins(input: …) -> …:
  return map(input, lambda grades:
    filter(grades, lambda g:
      g > reduce(grades, 0, lambda acc, cur:
        acc if acc < cur else cur
      )
    )
  )
```

map

filter

reduce

```
def dropmins(input: …) -> …:
  return map(input, lambda grades:
    filter(grades, lambda g:
      g > reduce(grades, 0, lambda acc, cur:
        acc if acc < cur else cur
      )
    )
  )
```

```
map       : [U] -> (U -> V) -> [V]

filter    : [U] -> (U -> bool) -> [U]

reduce    : [U] -> V -> (V -> U -> V) -> V
```

```
map       : [U] -> (U -> V) -> [V]
```

Take a list of U, apply the mapping function U → V, obtaining a list of V.

```
filter    : [U] -> (U -> bool) -> [U]
```

Take a list of U, use the filtering function U → bool to keep only the Us that produces true. The result is again a list of U.

```
reduce    : [U] -> V -> (V -> U -> V) -> V
```

Take a list of U and an initial state V. From the left, continuously apply the transition function V → U → V to accumulate the state V. After traversing the entire list, return the final V.

Typing Practices

```
sum

min

len

sort
```

Typing Practices

```
sum          : [Int] -> Int

min          : [Int] -> Int

len          : [Int] -> Int

sort         : [Int] -> [Int]
```

Implementation Practices

```
sum          : [Int] -> Int

min          : [Int] -> Int

len          : [Int] -> Int

sort         : [Int] -> [Int]
```

Your Tools:

```
map        : [U] -> (U -> V) -> [V]
filter     : [U] -> (U -> bool) -> [U]
reduce     : [U] -> V -> (V -> U -> V) -> V
```

Implementation Practices

```
sum          : [Int] -> Int
  sum(list) = reduce(list, 0, lambda acc, n: acc + n)
min          : [Int] -> Int

len          : [Int] -> Int

sort         : [Int] -> [Int]
```

Your Tools:

```
map       : [U] -> (U -> V) -> [V]
filter    : [U] -> (U -> bool) -> [U]
reduce    : [U] -> V -> (V -> U -> V) -> V
```

Implementation Practices

```
sum          : [Int] -> Int
  sum(list) = reduce(list, 0, lambda acc, n: acc + n)
min          : [Int] -> Int
  min(list) = reduce(list, None, lambda acc, n: n if acc is None or n < acc else n)
len          : [Int] -> Int

sort         : [Int] -> [Int]
```

Your Tools:

```
map       : [U] -> (U -> V) -> [V]
filter    : [U] -> (U -> bool) -> [U]
reduce    : [U] -> V -> (V -> U -> V) -> V
```

Implementation Practices

```
sum          : [Int] -> Int
  sum(list) = reduce(list, 0, lambda acc, n: acc + n)
min          : [Int] -> Int
  min(list) = reduce(list, None, lambda acc, n: n if acc is None or n < acc else n)
len          : [Int] -> Int
  len(list) = reduce(list, 0, lambda acc, n: acc + 1)
sort         : [Int] -> [Int]
```

Your Tools:

```
map       : [U] -> (U -> V) -> [V]
filter    : [U] -> (U -> bool) -> [U]
reduce    : [U] -> V -> (V -> U -> V) -> V
```

Implementation Practices

```
sum          : [Int] -> Int
  sum(list) = reduce(list, 0, lambda acc, n: acc + n)
min          : [Int] -> Int
  min(list) = reduce(list, None, lambda acc, n: n if acc is None or n < acc else n)
len          : [Int] -> Int
  len(list) = reduce(list, 0, lambda acc, n: acc + 1)
sort         : [Int] -> [Int]
  ???
```

Your Tools:

```
map       : [U] -> (U -> V) -> [V]
filter    : [U] -> (U -> bool) -> [U]
reduce    : [U] -> V -> (V -> U -> V) -> V
```

```
dropmins(input) =
  map(input, lambda grades:
    filter(grades, lambda g:
      g > reduce(grades, 0, lambda acc, cur:
        acc if acc < cur else cur
      )
    )
  )
```

```
map       : [U] -> (U -> V) -> [V]

filter    : [U] -> (U -> bool) -> [U]

reduce    : [U] -> V -> (V -> U -> V) -> V
```

List Manipulation DSL Syntax

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …

Var ::= String

BinaryOp ::= > | < | == | + | - | …
```

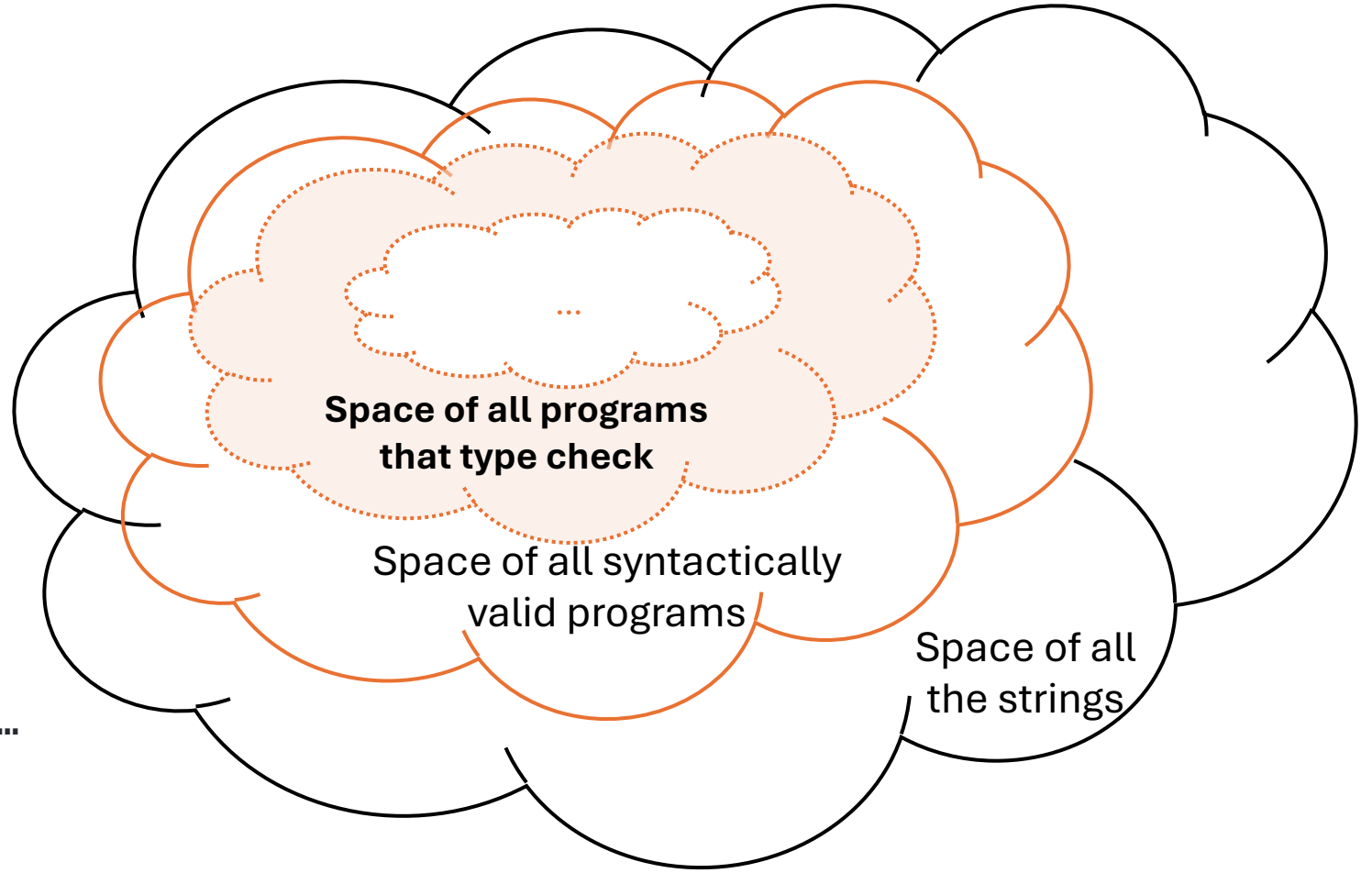
Typing Checking

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …

Var ::= String

BinaryOp ::= > | < | == | + | - | …
```

```
true + map(3, list, 5)

1 + (lambda x: x + 1)

map(input, lambda x: x + 1)
```

Typing Checking

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …

Var ::= String

BinaryOp ::= > | < | == | + | - | …
```



Types

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …

Var ::= String

BinaryOp ::= > | < | == | + | - | …
```

```
Type ::= Int
       | Bool
       | List[Type]
       | Type -> Type
```

Typing Rule

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```

```
Type ::= Int
       | Bool
       | List[Type]
       | Type -> Type
```

$$\frac{\text{<Premises>}}{\text{<Context> |– <Expr> : <Type>}}$$

Typing Rule

$$\frac{\text{<Premises>}}{\text{<Context>} \vdash \text{<Expr>} : \text{<Type>}}$$

```
Expr ::= Var
       | lambda Var. Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```

```
Type ::= Int
       | Bool
       | List[Type]
       | Type -> Type
```

[Int] $$\frac{}{C \vdash 1 : \text{Int}}$$

[True] $$\frac{}{C \vdash \text{true} : \text{Bool}}$$

[False] $$\frac{}{C \vdash \text{false} : \text{Bool}}$$

Typing Rule

$$\frac{\text{<Premises>}}{\text{<Context>} \vdash \text{<Expr>} : \text{<Type>}}$$

```
Expr ::= Var
       | lambda Var. Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```

```
Type ::= Int
       | Bool
       | List[Type]
       | Type -> Type
```

[Arith] $$\frac{C \vdash \mathrm{e}_1 : \mathrm{Int} \quad C \vdash \mathrm{e}_2 : \mathrm{Int}}{C \vdash \mathrm{e}_1 + e_2 : \mathrm{Bool}}$$

[Equality] $$\frac{C \vdash \mathrm{e}_1 : \mathrm{Int} \quad C \vdash \mathrm{e}_2 : \mathrm{Int}}{C \vdash \mathrm{e}_1 == e_2 : \mathrm{Bool}}$$

Typing Rule

$$\frac{\text{<Premises>}}{\text{<Context> |– <Expr> : <Type>}}$$

```
Expr ::= Var
       | lambda Var. Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```

```
Type ::= Int
       | Bool
       | List[Type]
       | Type -> Type
```

$$\text{[Var]}\ \frac{(v \mapsto \tau) \in C}{C\ \vdash \text{Var}(v) : \tau}$$

$$\text{[Lambda]}\ \frac{C \cup \{x \mapsto \tau_1\} \vdash e : \tau_2}{C\ \vdash \lambda x.e : \tau_1 \to \tau_2}$$

Typing Rule

$$\frac{\text{<Premises>}}{\text{<Context> |– <Expr> : <Type>}}$$

```
Expr ::= Var
       | lambda Var. Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```

```
Type ::= Int
       | Bool
       | List[Type]
       | Type -> Type
```

[Map] $$\frac{C \vdash e_1 : \text{List}[\tau_1] \quad C \vdash e_2 : \tau_1 \to t_2}{C \vdash \text{map}(e_1, e_2) : \text{List}[\tau_2]}$$

[Filter] $$\frac{C \vdash e_1 : \text{List}[\tau_1] \quad C \vdash e_2 : \tau_1 \to \text{Bool}}{C \vdash \text{filter}(e_1, e_2) : \text{List}[\tau_1]}$$

Type Checking that Succeeds

```
len : [Int] -> Int
  len(list) = reduce(list, 0, lambda acc, n: acc + 1)
```

Type Checking that Succeeds

```
len : [Int] -> Int
  len(list) = reduce(list, 0, lambda acc, n: acc + 1)
```

$$\dfrac{\dots \quad \dfrac{(\text{list} \mapsto \text{List[Int]}) \in C}{C \vdash \text{list} : \text{List[Int]}} \quad \dfrac{}{C \vdash 0 : \text{Int}} \quad \dfrac{\dfrac{(\text{acc} \mapsto \text{Int}) \in C}{C \vdash \text{acc} : \text{Int}} \quad \dfrac{(\text{n} \mapsto \text{Int}) \in C}{C \vdash \text{n} : \text{Int}} \quad \dfrac{\dfrac{(\text{acc} \mapsto \text{Int}) \in C}{C \vdash \text{acc} : \text{Int}} \quad \dfrac{}{C \vdash 1 : \text{Int}}}{C \vdash \text{acc} + 1 : \text{Int}}}{C \vdash \text{lambda acc, n. acc} + 1 : \text{Int} \to \text{Int} \to \text{Int}}}{C \vdash \text{reduce(list, 0, lambda acc, n. acc} + 1) : \text{Int}}$$

$$C \vdash \text{len} : [\text{Int}] \to \text{Int}$$ ✅

Type Checking that Failed

```
gt0 : [Int] -> [Int]
  gt0(list) = filter(list, 0)
```

$$\text{[Filter]}\ \frac{C \vdash e_1 : \text{List}[\tau_1] \quad C \vdash e_2 : \tau_1 \to \text{Bool}}{C \vdash \text{filter}(e_1, e_2) : \text{List}[\tau_1]}$$

Type Checking that Failed

```
gt0 : [Int] -> [Int]
  gt0(list) = filter(list, 0)
```

[Filter] $\dfrac{C \vdash e_1 : \text{List}[\tau_1] \quad C \vdash e_2 : \tau_1 \to \text{Bool}}{C \vdash \text{filter}(e_1, e_2) : \text{List}[\tau_1]}$

$$\dfrac{\dfrac{(\text{list} \mapsto \text{List}[\text{Int}]) \in C}{C \vdash \text{list} : \text{List}[\text{Int}]} \qquad \dfrac{}{C \vdash 0 : \text{Int}}}{C \vdash \text{filter}(\text{list}, 0) : \text{❌}}$$

Type Checking that Succeeded (Corrected)

```
gt0 : [Int] -> [Int]
  gt0(list) = filter(list, lambda x: x > 0)
```

$$\text{[Filter]}\ \frac{C \vdash e_1 : \text{List}[\tau_1] \quad C \vdash e_2 : \tau_1 \to \text{Bool}}{C \vdash \text{filter}(e_1, e_2) : \text{List}[\tau_1]}$$

$$\dfrac{\dfrac{(\text{list} \mapsto \text{List[Int]}) \in C}{C \vdash \text{list} : \text{List[Int]}} \quad \dfrac{\dfrac{\dfrac{(\text{x} \mapsto \text{Int}) \in C}{C \vdash \text{x} : \text{Int}} \quad \dfrac{}{C \vdash 0 : \text{Int}}}{C \vdash x > 0 : \text{Bool}} \quad \dfrac{(\text{x} \mapsto \text{Int}) \in C}{C \vdash \text{x} : \text{Int}}}{C \vdash \text{lambda } x.\, x > 0 : \text{Int} \to \text{Bool}}}{\dfrac{\dots \qquad C \vdash \text{filter(list, lambda x. x} > 0) : \text{List[Int]}}{C \vdash \text{gt0} : \text{List[Int]} \to \text{List[Int]}}}$$

✅

Typing Rules

```
Expr ::= Var
       | lambda Var. Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
Type ::= Int
       | Bool
       | List[Type]
       | Type -> Type
```

$$\text{[Int]}\ \frac{}{C \vdash \text{IntLiteral} : \text{Int}}$$

$$\text{[True]}\ \frac{}{C \vdash \text{true} : \text{Bool}}$$

$$\text{[False]}\ \frac{}{C \vdash \text{false} : \text{Bool}}$$

$$\text{[Arith]}\ \frac{C \vdash e_1 : \text{Int} \quad C \vdash e_2 : \text{Int}}{C \vdash e_1 + e_2 : \text{Bool}}$$

$$\text{[Equality]}\ \frac{C \vdash e_1 : \text{Int} \quad C \vdash e_2 : \text{Int}}{C \vdash e_1 == e_2 : \text{Bool}}$$

$$\text{[Var]}\ \frac{(v \mapsto \tau) \in C}{C \vdash \text{Var}(v) : \tau}$$

$$\text{[Lambda]}\ \frac{C \cup \{x \mapsto \tau_1\} \vdash e : \tau_2}{C \vdash \lambda x.\, e : \tau_1 \to \tau_2}$$

$$\text{[Map]}\ \frac{C \vdash e_1 : \text{List}[\tau_1] \quad C \vdash e_2 : \tau_1 \to t_2}{C \vdash \text{map}(e_1, e_2) : \text{List}[\tau_2]}$$

$$\text{[Filter]}\ \frac{C \vdash e_1 : \text{List}[\tau_1] \quad C \vdash e_2 : \tau_1 \to \text{Bool}}{C \vdash \text{filter}(e_1, e_2) : \text{List}[\tau_1]}$$

Typing Rules in the Wild

Datafun [Arntzenius et. al. 2016]

$$\frac{\Delta;\Gamma \vdash \mathcal{L} \Rightarrow \Delta' \quad \Delta,\Delta';\Gamma \vdash e : \mathrm{L}}{\Delta;\Gamma \vdash \bigvee(\mathcal{L})\ e : \mathrm{L}}$$

$$\frac{\Delta;\Gamma \vdash \mathcal{L} \Rightarrow \Delta' \quad \Delta,\Delta';\Gamma \vdash e : \mathrm{A}}{\Delta;\Gamma \vdash \{e \mid \mathcal{L}\} : \{\mathrm{A}\}}$$

$$\frac{\Delta;\cdot \vdash e : \mathrm{A} \quad \Delta;\Gamma \vdash p : \mathrm{A} \Rightarrow \Delta' \quad \Delta,\Delta';\Gamma \vdash e_1 : \mathrm{C} \quad \Delta;\Gamma}{\Delta;\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ p \to e_1;\ \_ \to e_2 : \mathrm{C}}$$

$$\frac{\Delta;\Gamma \vdash \mathcal{L}_1 \Rightarrow \Delta_1 \quad \Delta;\Gamma \vdash \mathcal{L}_2 \Rightarrow}{\Delta;\Gamma \vdash \mathcal{L}_1,\mathcal{L}_2 \Rightarrow \Delta_1,\Delta_2}$$

$$\frac{\Delta;\Gamma \vdash e : \{\mathrm{A}\} \quad \Delta;\Gamma \vdash p : \mathrm{A} \Rightarrow \Delta}{\Delta;\Gamma \vdash p \in e \Rightarrow \Delta'}$$

$$\frac{}{\Delta;\Gamma \vdash \_ : \mathrm{A}}$$

$$\frac{\Delta;\Gamma \vdash p_1 : \mathrm{A}_1 \Rightarrow \Delta_1 \quad \Delta;}{\Delta;\Gamma \vdash (p_1,p_2) : \mathrm{A}_1 \times}$$

$$\frac{\Delta;\Gamma \vdash e : \underset{eq}{\mathrm{A}}}{\Delta;\Gamma \vdash !e : \underset{eq}{\mathrm{A}} \Rightarrow \cdot} \qquad \frac{}{\Delta;\Gamma \vdash}$$

WebAssembly [Haas et. al. 2017]

(contexts) $C ::= \{\mathsf{func}\ tf^*,\ \mathsf{global}\ tg^*,\ \mathsf{table}\ n^?,\ \mathsf{memory}\ n^?,\ \mathsf{local}\ t^*,\ \mathsf{label}\ (t^*)^*,\ \mathsf{return}\ (t^*)^?\}$

Typing Instructions $\boxed{C \vdash e^* : tf}$

$$\frac{}{C \vdash t.\mathbf{const}\ c : \epsilon \to t} \qquad \frac{}{C \vdash t.unop : t \to t} \qquad \frac{}{C \vdash t.binop : t\ t \to t} \qquad \frac{}{C \vdash t.testop : t \to \mathsf{i32}} \qquad \frac{}{C \vdash t.relop : t\ t \to \mathsf{i32}}$$

$$\frac{t_1 \neq t_2 \qquad sx^? = \epsilon \Leftrightarrow (t_1 = \mathbf{i}n \wedge t_2 = \mathbf{i}n' \wedge |t_1| < |t_2|) \vee (t_1 = \mathbf{f}n \wedge t_2 = \mathbf{f}n')}{C \vdash t_1.\mathbf{convert}\ t_2\_sx^? : t_2 \to t_1} \qquad \frac{t_1 \neq t_2 \qquad |t_1| = |t_2|}{C \vdash t_1.\mathbf{reinterpret}\ t_2 : t_2 \to t_1}$$

$$\frac{}{C \vdash \mathbf{drop} : t \to \epsilon} \qquad \frac{}{C \vdash \mathbf{select} : t\ t\ \mathsf{i32} \to t}$$

$$\frac{tf = t_1^n \to t_2^m \qquad C, \mathsf{label}\ (t_1^n) \vdash e^* : tf}{C \vdash \mathbf{loop}\ tf\ e^*\ \mathbf{end} : tf}$$

$$\frac{\vdash e_1^* : tf \qquad C, \mathsf{label}\ (t_2^m) \vdash e_2^* : tf}{e_2^*\ \mathbf{end} : t_1^n\ \mathsf{i32} \to t_2^m}$$

$$\frac{t^*}{32 \to t^*} \qquad \frac{(C_{\mathsf{label}}(i) = t^*)^+}{C \vdash \mathbf{br\_table}\ i^+ : t_1^*\ t^*\ \mathsf{i32} \to t_2^*}$$

$$\frac{= tf}{: tf} \qquad \frac{tf = t_1^* \to t_2^* \qquad C_{\mathsf{table}} = n}{C \vdash \mathbf{call\_indirect}\ tf : t_1^*\ \mathsf{i32} \to t_2^*}$$

$\Sigma;\Delta;\Gamma;\Theta \vdash e : \tau \Rightarrow \Gamma'$

T-U32

$$\frac{}{\Sigma;\Delta;\Gamma;\Theta \vdash \boxed{n} : \mathsf{u32} \Rightarrow \Gamma}$$

T-Move

$$\frac{\Delta;\Gamma;\Theta \vdash_{\mathsf{uniq}} \pi \Rightarrow \{\ ^{\mathsf{uniq}}\pi\ \} \quad \Gamma(\pi) = \tau^{\mathrm{SI}} \quad \mathbf{noncopyable}_\Sigma\ \tau^{\mathrm{SI}}}{\Sigma;\Delta;\Gamma;\Theta \vdash \boxed{\pi} : \tau^{\mathrm{SI}} \Rightarrow \Gamma[\pi \mapsto \tau^{\mathrm{SI}^\dagger}]}$$

T-Copy

$$\frac{\Delta;\Gamma;\Theta \vdash_{\mathsf{shrd}} p \Rightarrow \{\bar{\ell}\} \quad \Delta;\Gamma \vdash_{\mathsf{shrd}} p : \tau^{\mathrm{SI}} \quad \mathbf{copyable}_\Sigma\ \tau^{\mathrm{SI}}}{\Sigma;\Delta;\Gamma;\Theta \vdash \boxed{p} : \tau^{\mathrm{SI}} \Rightarrow \Gamma}$$

T-Borrow

$$\frac{\Gamma(r) = \emptyset \qquad \Gamma;\Theta \vdash r\ \mathsf{rnic} \quad \Delta;\Gamma;\Theta \vdash_\omega p \Rightarrow \{\bar{\ell}\} \qquad \Delta;\Gamma \vdash_\omega p : \tau^{\mathrm{XI}}}{\Sigma;\Delta;\Gamma;\Theta \vdash \boxed{\&r\ \omega\ p} : \&r\ \omega\ \tau^{\mathrm{XI}} \Rightarrow \Gamma[r \mapsto \{\bar{\ell}\}]}$$

T-LetRegion

$$\frac{\Sigma;\Delta;\Gamma,\ r \mapsto \{\};\Theta \vdash \boxed{e} : \tau^{\mathrm{SI}} \Rightarrow \Gamma',\ r \mapsto \{\bar{\ell}\}}{\Sigma;\Delta;\Gamma;\Theta \vdash \boxed{\mathtt{letrgn}\ \langle r \rangle\ \{\ e\ \}} : \tau^{\mathrm{SI}} \Rightarrow \Gamma'}$$

T-Branch

$$\frac{\Sigma;\Delta;\Gamma;\Theta \vdash \boxed{e_1} : \mathtt{bool} \Rightarrow \Gamma_1 \qquad \Sigma;\Delta;\Gamma_1;\Theta \vdash \boxed{e_2} : \tau_2^{\mathrm{SI}} \Rightarrow \Gamma_2 \quad \Sigma;\Delta;\Gamma_1;\Theta \vdash \boxed{e_3} : \tau_3^{\mathrm{SI}} \Rightarrow \Gamma_3 \qquad \tau^{\mathrm{SI}} = \tau_2^{\mathrm{SI}} \vee \tau^{\mathrm{SI}} = \tau_3^{\mathrm{SI}} \quad \Delta;\Gamma_2;\Theta \vdash^+ \tau_2^{\mathrm{SI}} \rightsquigarrow \tau^{\mathrm{SI}} \dashv \Gamma_2' \qquad \Delta;\Gamma_3;\Theta \vdash^+ \tau_3^{\mathrm{SI}} \rightsquigarrow \tau^{\mathrm{SI}} \dashv \Gamma_3' \qquad \Gamma_2' \uplus \Gamma_3' = \Gamma'}{\Sigma;\Delta;\Gamma;\Theta \vdash \boxed{\mathtt{if}\ e_1\ \{\ e_2\ \}\ \mathtt{else}\ \{\ e_3\ \}} : \tau^{\mathrm{SI}} \Rightarrow \Gamma'}$$

T-Seq

$$\frac{\Sigma;\Delta;\Gamma;\Theta \vdash \boxed{e_1} : \tau_1^{\mathrm{SI}} \Rightarrow \Gamma_1 \quad \Sigma;\Delta;\mathrm{gc\text{-}loans}_\Theta(\Gamma_1);\Theta \vdash \boxed{e_2} : \tau_2^{\mathrm{SI}} \Rightarrow \Gamma_2}{\Sigma;\Delta;\Gamma;\Theta \vdash \boxed{e_1;\ e_2} : \tau_2^{\mathrm{SI}} \Rightarrow \Gamma_2}$$

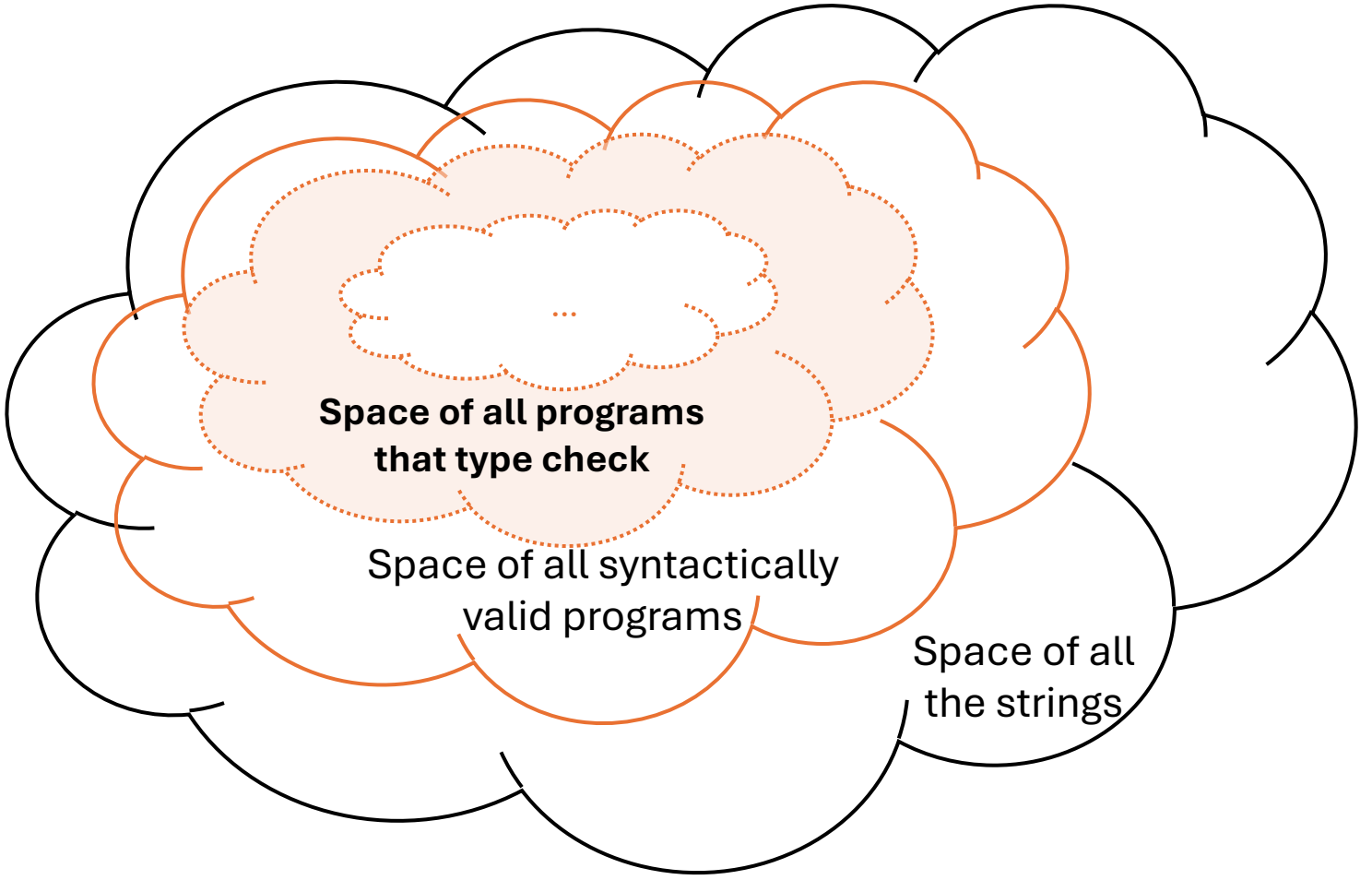
T-Let

$$\frac{\Sigma;\Delta;\Gamma;\Theta \vdash \boxed{e_1} : \tau_1^{\mathrm{SI}} \Rightarrow \Gamma_1 \qquad \Delta;\Gamma_1;\Theta \vdash^+ \tau_1^{\mathrm{SI}} \rightsquigarrow \tau_a^{\mathrm{SI}} \dashv \Gamma_1' \quad \forall r \in \text{free-regions}(\tau_a^{\mathrm{SI}}).\ \Gamma_1' \vdash r\ \mathtt{rnrb} \qquad \Sigma;\Delta;\mathrm{gc\text{-}loans}_\Theta(\Gamma_1',\ x\ :\ \tau_a^{\mathrm{SI}});\Theta \vdash \boxed{e_2} : \tau_2^{\mathrm{SI}} \Rightarrow \Gamma_2,\ x\ :\ \tau^{\mathrm{SD}}}{\Sigma;\Delta;\Gamma;\Theta \vdash \boxed{\mathtt{let}\ x : \tau_a^{\mathrm{SI}} = e_1;\ e_2} : \tau_2^{\mathrm{SI}} \Rightarrow \Gamma_2}$$

Rust (Oxide) [Weiss et. al. 2021]

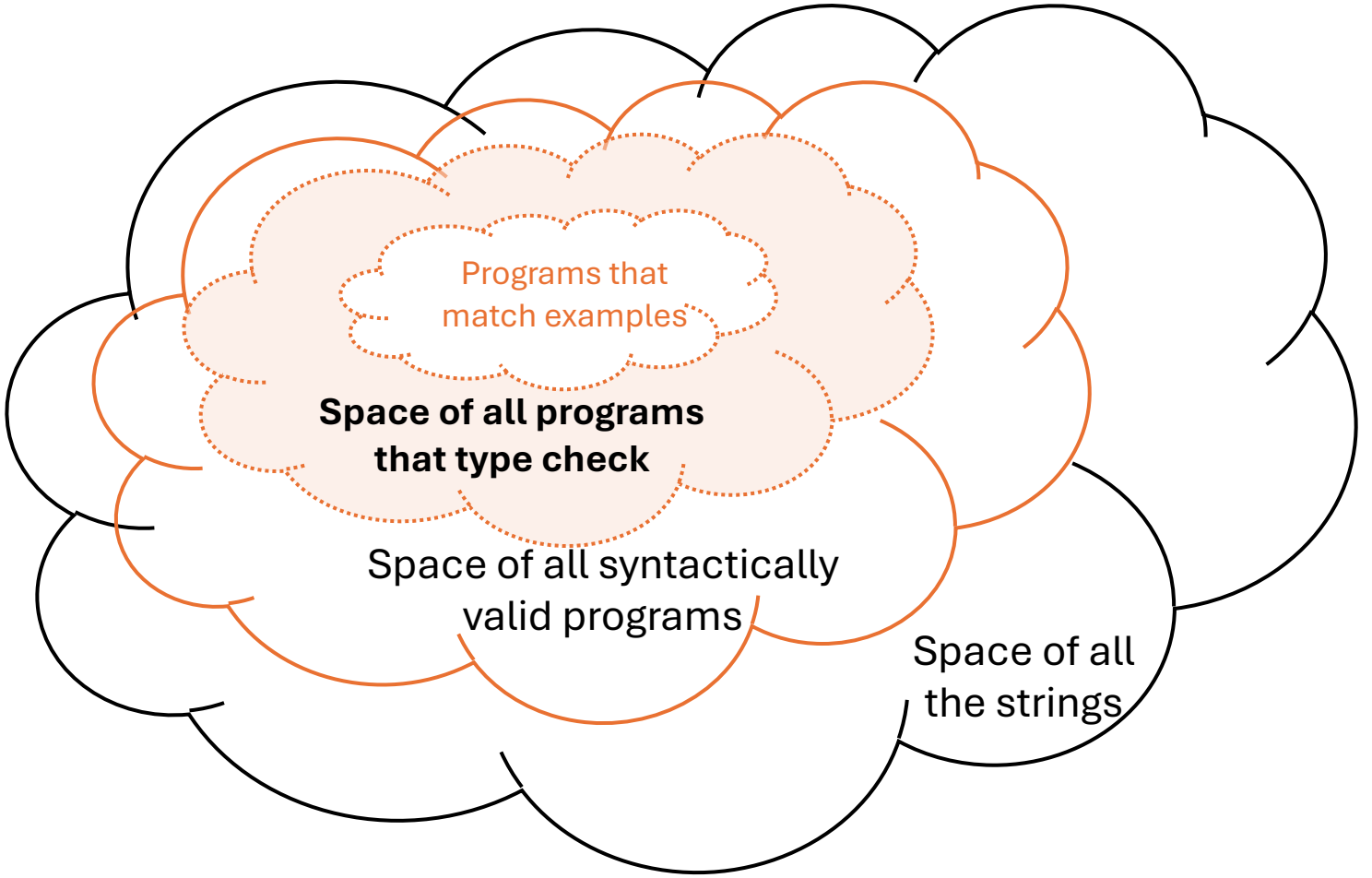
Typing Checking

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```



Typing Checking

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```



Top-Down Enumerative Search

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```

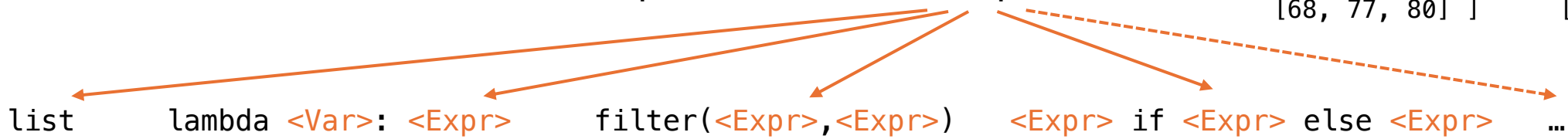
```
dropmin(list) = <Expr>
```

```
[ [71, 75, 83],     [ [75, 83],
  [90, 87, 95], →     [90, 95],
  [68, 77, 80] ]      [77, 80] ]
```

Top-Down Enumerative Search

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```

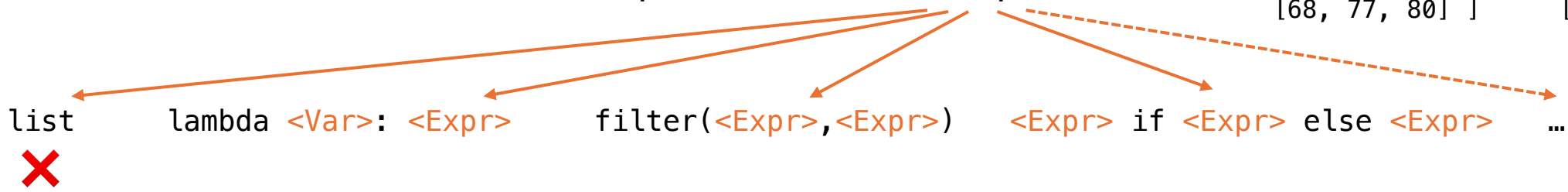
```
[ [71, 75, 83],      [ [75, 83],
  [90, 87, 95], →      [90, 95],
  [68, 77, 80] ]       [77, 80] ]
```



Top-Down Enumerative Search

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```

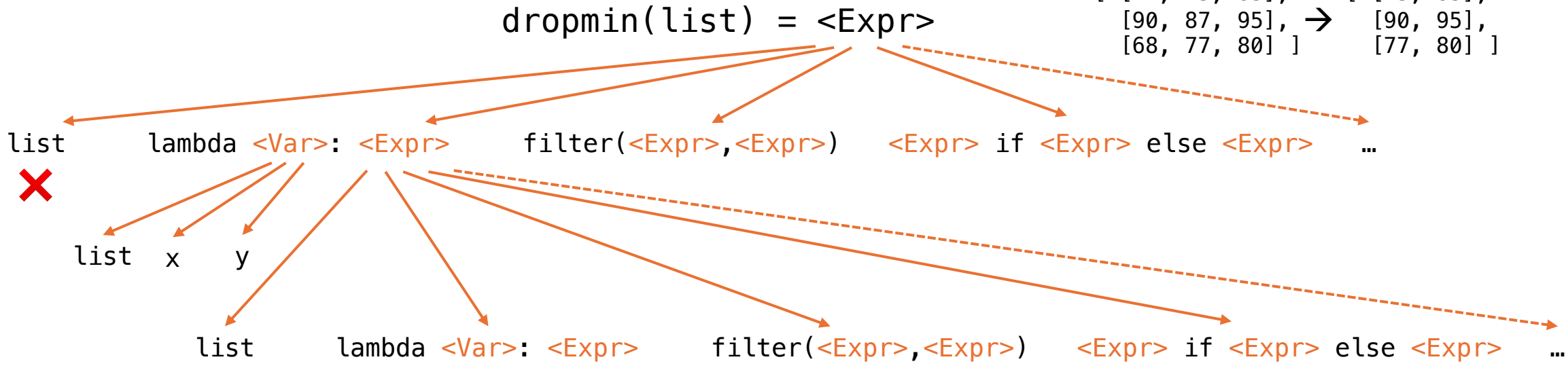
```
[ [71, 75, 83],      [ [75, 83],
  [90, 87, 95], →      [90, 95],
  [68, 77, 80] ]       [77, 80] ]
```



Top-Down Enumerative Search

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```

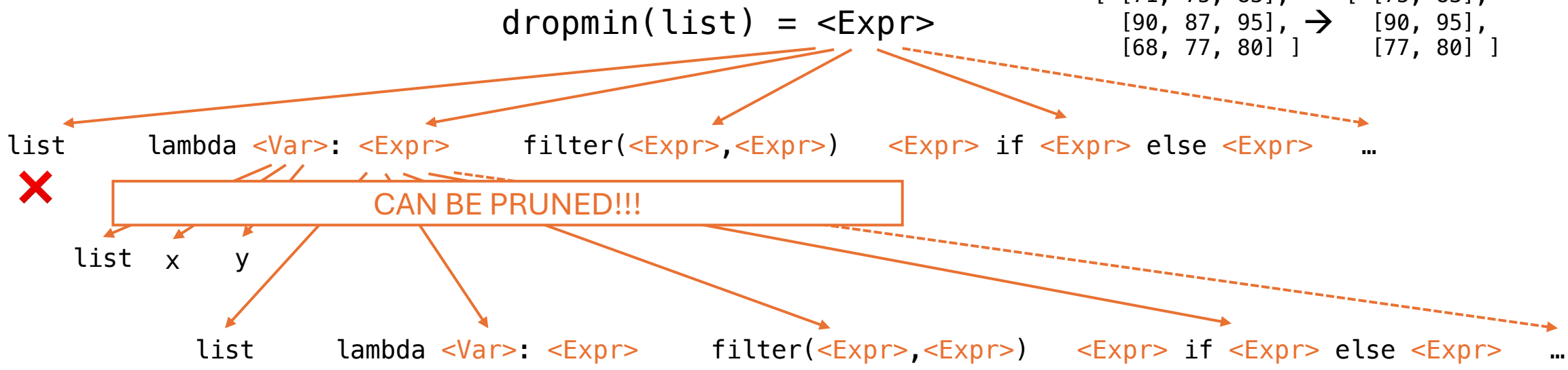
```
[ [71, 75, 83],     [ [75, 83],
  [90, 87, 95], →     [90, 95],
  [68, 77, 80] ]      [77, 80] ]
```



Top-Down Enumerative Search

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```

```
[ [71, 75, 83],      [ [75, 83],
  [90, 87, 95], →      [90, 95],
  [68, 77, 80] ]       [77, 80] ]
```



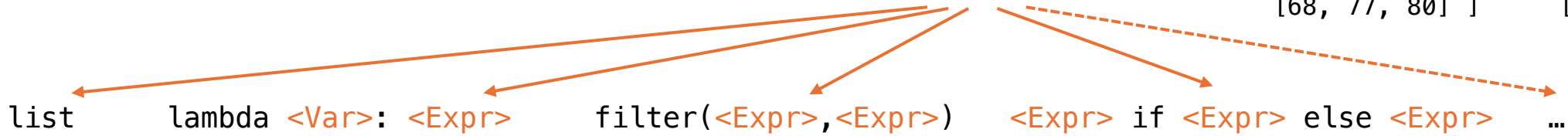
Type-Guided Top-Down Search

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```

```
[ [71, 75, 83],     [ [75, 83],
  [90, 87, 95], →     [90, 95],
  [68, 77, 80] ]      [77, 80] ]
```

dropmin : List[Int] -> List[Int]

dropmin(list) = <Expr>



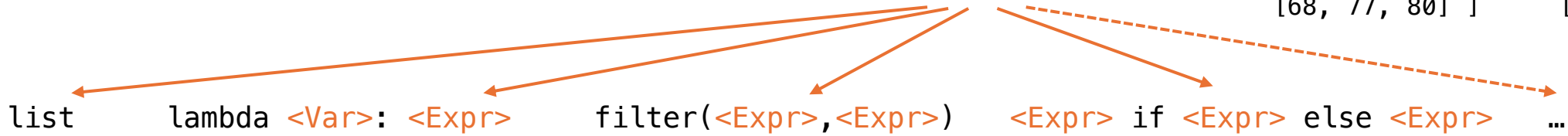
Type-Guided Top-Down Search

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```

```
[ [71, 75, 83],      [ [75, 83],
  [90, 87, 95], →      [90, 95],
  [68, 77, 80] ]       [77, 80] ]
```

dropmin : List[Int] -> List[Int]

dropmin(list) = <Expr>



Type-Guided Top-Down Search

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```

```
[ [71, 75, 83],      [ [75, 83],
  [90, 87, 95], →      [90, 95],
  [68, 77, 80] ]       [77, 80] ]
```

dropmin : List[Int] -> List[Int]

dropmin(list) = <Expr>

list lambda <Var>: <Expr> filter(<Expr>,<Expr>) <Expr> if <Expr> else <Expr> …

List[int]

$$\frac{(\text{list} \mapsto \text{List[Int]}) \in C}{C \vdash \text{list} : \text{List[Int]}}$$

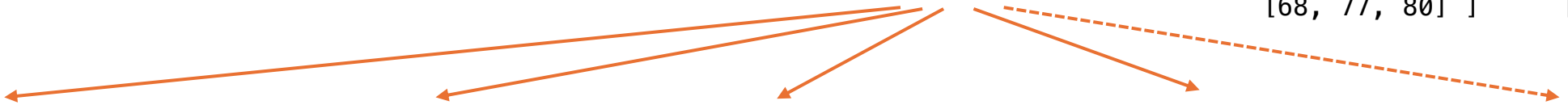
Type-Guided Top-Down Search

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```

```
[ [71, 75, 83],    [ [75, 83],
  [90, 87, 95], →    [90, 95],
  [68, 77, 80] ]     [77, 80] ]
```

dropmin : List[Int] -> List[Int]

dropmin(list) = <Expr>



List[int] ✅

```
[ [71, 75, 83],    [ [75, 83],
  [90, 87, 95], →    [90, 95],
  [68, 77, 80] ]     [77, 80] ]
```

❌

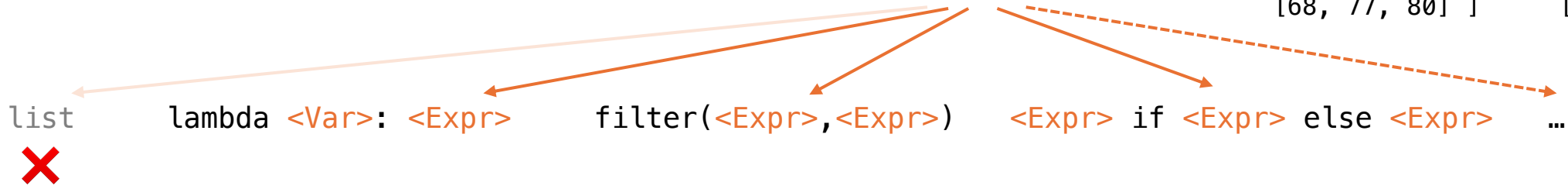
Type-Guided Top-Down Search

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```

```
[ [71, 75, 83],      [ [75, 83],
  [90, 87, 95], →      [90, 95],
  [68, 77, 80] ]       [77, 80] ]
```

dropmin : List[Int] -> List[Int]

dropmin(list) = <Expr>



Type-Guided Top-Down Search

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```

```
[ [71, 75, 83],      [ [75, 83],
  [90, 87, 95], →      [90, 95],
  [68, 77, 80] ]       [77, 80] ]
```

dropmin : List[Int] -> List[Int]

dropmin(list) = <Expr>

list ✗

lambda <Var>: <Expr>

List[Int]

filter(<Expr>,<Expr>)

<Expr> if <Expr> else <Expr>

…

[Lambda] $\dfrac{C \cup \{x \mapsto \tau_1\} \vdash e : \tau_2}{C \vdash \lambda x.e : \tau_1 \to \tau_2}$

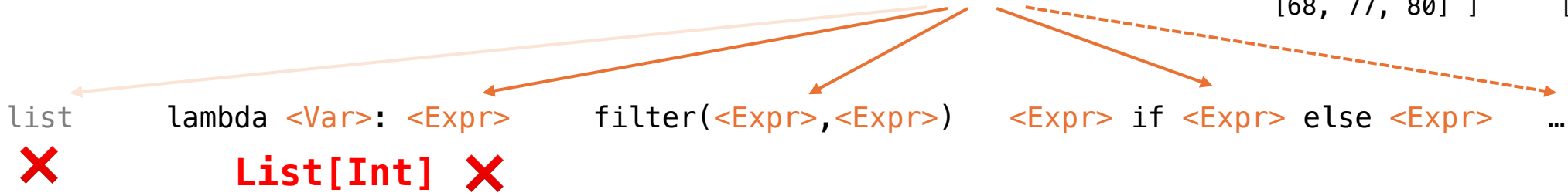
Type-Guided Top-Down Search

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```

```
[ [71, 75, 83],      [ [75, 83],
  [90, 87, 95], →      [90, 95],
  [68, 77, 80] ]       [77, 80] ]
```

dropmin : List[Int] -> List[Int]

dropmin(list) = <Expr>



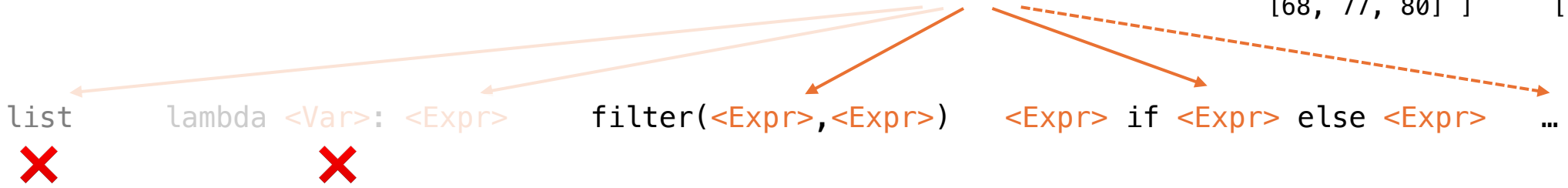
Type-Guided Top-Down Search

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```

```
[ [71, 75, 83],      [ [75, 83],
  [90, 87, 95], →      [90, 95],
  [68, 77, 80] ]       [77, 80] ]
```

dropmin : List[Int] -> List[Int]

dropmin(list) = <Expr>



Type-Guided Top-Down Search

```
Expr ::= Var
       | lambda Var: Expr
       | filter(Expr, Expr)
       | map(Expr, Expr)
       | reduce(Expr, Expr, Expr)
       | Expr if Expr else Expr
       | Expr BinaryOp Expr
       | true | false
       | 0 | 1 | 2 | …
```

```
[ [71, 75, 83],      [ [75, 83],
  [90, 87, 95], →      [90, 95],
  [68, 77, 80] ]       [77, 80] ]
```

dropmin : List[Int] -> List[Int]

dropmin(list) = <Expr>

Polymorphism

Base types

```
Int
```

Parametric types (Polymorphism)

```
∀T, List[T]
```

Parametric type instance (Concrete; Monomorphic)

```
List[Int]
```

Polymorphism

Base types

```
Int
```

Parametric types (Polymorphism)

```
∀T, List[T]
```

Parametric type instance (Concrete; Monomorphic)

```
List[Int]
```

Types as sets

Base types

```
Int      = {…, -2, -1, 0, 1, 2, …}
```

Parametric types (Polymorphism)

```
∀T, List[T]      = {…,[], [“a”, “b”], [3], [false, false, true], …}
```

Parametric type instance (Concrete; Monomorphic)

```
List[Int]      = {[], [1], [2, 3], [1, 2, 3], …}
```

Types as sets that can be refined!

Base types

```
Int      = {…, -2, -1, 0, 1, 2, …}
```

Parametric types (Polymorphism)

```
∀T, List[T]   = {…,[], [“a”, “b”], [3], [false, false, true], …}
```

Parametric type instance (Concrete; Monomorphic)

```
List[Int]     = {[], [1], [2, 3], [1, 2, 3], …}
```

Refinement types

```
{v : Int | v > 0}
```

Types as sets that can be refined!

Base types

```
Int      = {…, -2, -1, 0, 1, 2, …}
```

Parametric types (Polymorphism)

```
∀T, List[T]   = {…,[], [“a”, “b”], [3], [false, false, true], …}
```

Parametric type instance (Concrete; Monomorphic)

```
List[Int]     = {[], [1], [2, 3], [1, 2, 3], …}
```

Refinement types

```
{v : Int | v > 0}   = {1, 2, 3, …}
```

Types as sets that can be refined!

Base types

```
Int      = {…, -2, -1, 0, 1, 2, …}
```

Parametric types (Polymorphism)

```
∀T, List[T]   = {…,[], [“a”, “b”], [3], [false, false, true], …}
```

Parametric type instance (Concrete; Monomorphic)

```
List[Int]     = {[], [1], [2, 3], [1, 2, 3], …}
```

Refinement types

```
{v : Int | v > 0}   = {1, 2, 3, …}
∀T, {ls : List[T] | len(ls) > 0}  = {[1], [2, 3], [1, 2, 3], …}
```

Refinement Types for Synthesis

“absolute value”

Typing Specification (Original)

```
abs :: Int -> Int
```

Typing Specification (with Refinement)

```
abs :: Int -> {v : Int | v >= 0}
```

Refinement Types for Synthesis

“duplicate every element in a list”

Typing Specification (Original)

```
stutter :: List[T] -> List[T]
```

Typing Specification (with Refinement)

```
stutter :: in: List[T] -> {v: List[T] | len(v) = 2 * len(x)}
```

Refinement Types for Synthesis

“sort the elements in an array”

Typing Specification (Original)

```
sort :: List[T] -> List[T]
```

Typing Specification (with Refinement)

```
sort :: List[T] -> {v: List[T] | is_sorted(v)}
```

Refinement Types for Synthesis

“sort the elements in an array”

Typing Specification (Original)

```
sort :: List[T] -> List[T]
```

Typing Specification (with Refinement)

```
sort :: List[T] -> {v: List[T] | is_sorted(v)}
is_sorted(v) = ∀i,j, s.t., 0 <= i < j < len(v), v[i] <= v[j]
```

Refinement Types for Synthesis

“insert an element into an array”

Typing Specification (Original)

```
insert :: T -> List[T] -> List[T]
```

Typing Specification (with Refinement)

```
insert :: (e: T) -> (x: List[T]) -> {y : List[T] | elems(y) = elems(x) U {e}}
```

How to Enforce Constraints in Refinement Types

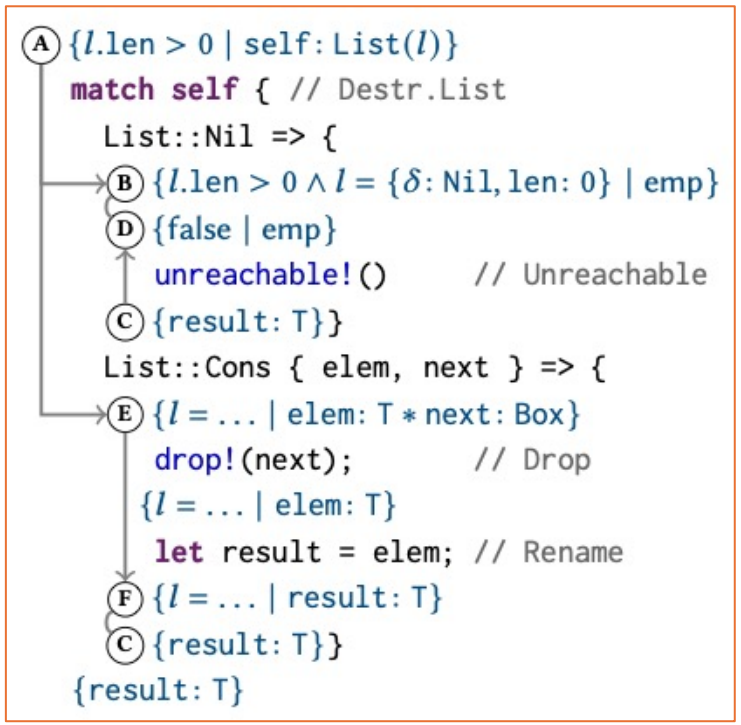
- similar strategies in top-down enumerative synthesis
- with type checkers capable of checking and strengthening refinement types
- during the type checking, we may rely on human defined predicates such as “len” and “elems”

How to Enforce Constraints in Refinement Types

Leveraging Rust Types for Program Synthesis

JONÁŠ FIALA, Department of Computer Science, ETH Zurich, Switzerland
SHACHAR ITZHAKY, Technion, Israel
PETER MÜLLER, Department of Computer Science, ETH Zurich, Switzerland
NADIA POLIKARPOVA, University of California, San Diego, USA
ILYA SERGEY, National University of Singapore, Singapore

```
#[ensures((^self).len() ==
          (*self).len() + (^result).len())]
fn last_mut(&mut self) -> &mut List<T> {
  todo!()
}
```



Summary

- Functional Language with Higher-Order Functions
- Type System, Typing Rules, and Type Checking
- Top-down Enumerative Synthesis Guided by Types
- (A little bit of) Refinement Types for Synthesis

Week 2

- Assignment 1
  - Released: https://github.com/machine-programming/assignment-1
  - Due Thursday of the Third Week (Sep 11)
  - Autograder not up yet; will be later today
  - API keys were sent out
- Waitlisted students
  - Please contact me by sending emails; will add you to Courselore, GradeScope, and give you API keys
- Any questions?