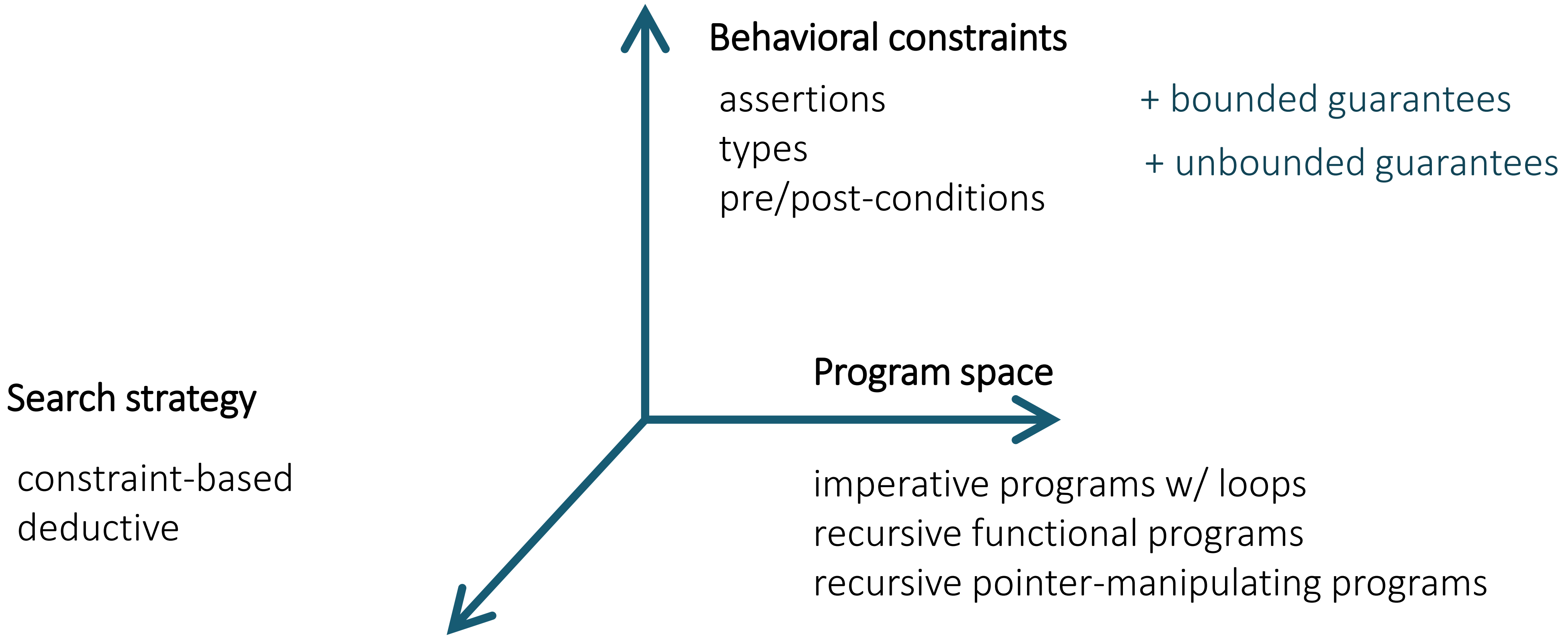
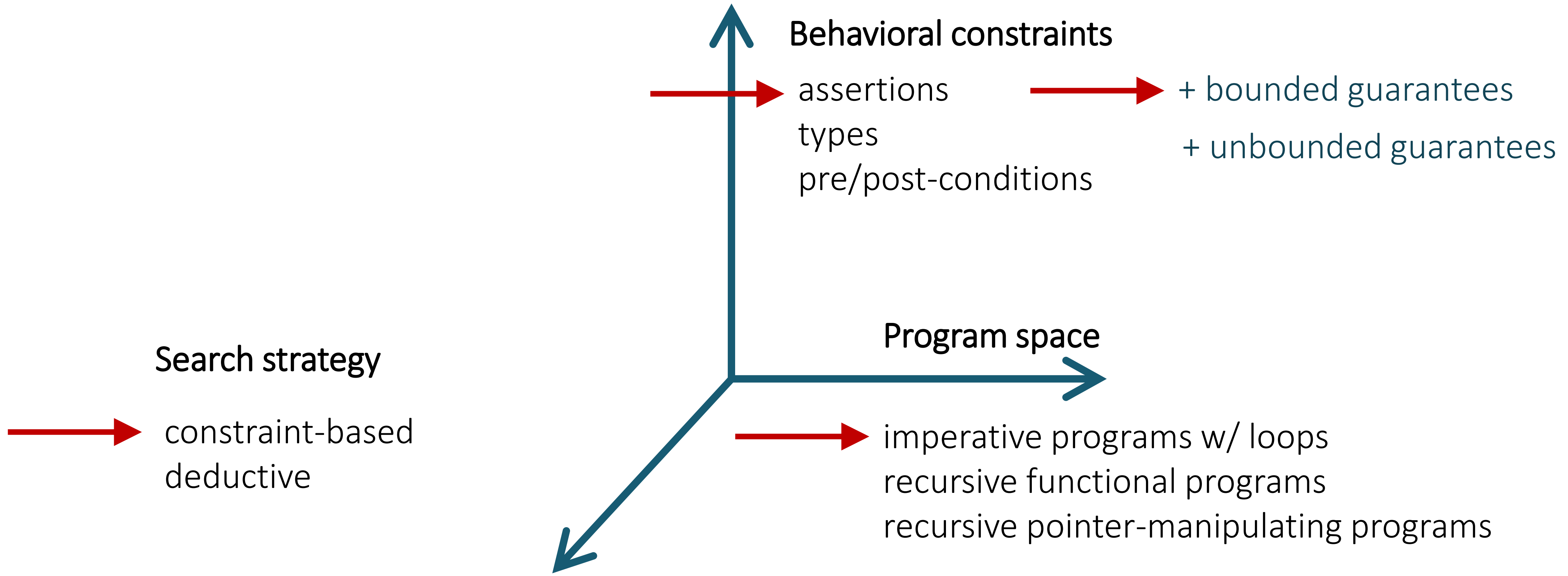
Lecture 11
Type Systems

Nadia Polikarpova

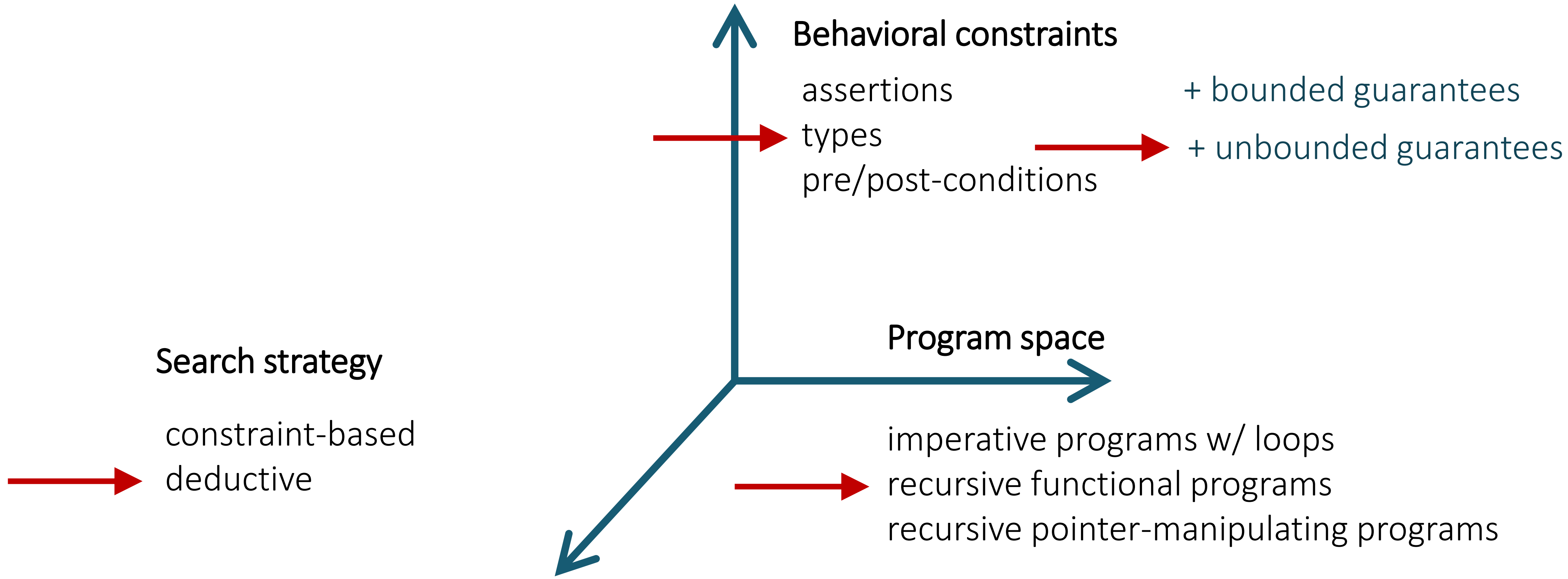
Module II


Behavioral constraints
assertions
types
pre/post-conditions
+ bounded guarantees
+ unbounded guarantees
Search strategy
constraint-based
deductive
Program space
imperative programs w/ loops
recursive functional programs
recursive pointer-manipulating programs

Last week


Behavioral constraints
assertions
types
pre/post-conditions
+ bounded guarantees
+ unbounded guarantees
Search strategy
constraint-based
deductive
Program space
imperative programs w/ loops
recursive functional programs
recursive pointer-manipulating programs

This week


Behavioral constraints
assertions
types
pre/post-conditions
+ bounded guarantees
+ unbounded guarantees
Program space
imperative programs w/ loops
recursive functional programs
recursive pointer-manipulating programs
Search strategy
constraint-based
deductive

Example: insert into a sorted list

Input:

x

5

xs

1	2	7	8

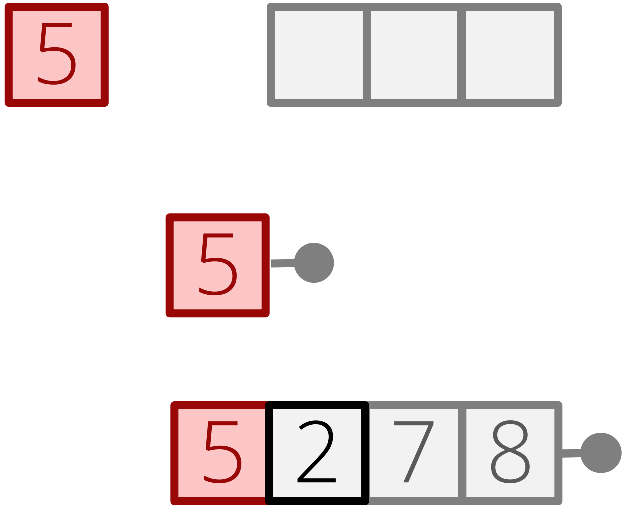
Output:

ys

1	2	5	7	8

In a functional language

```
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)
```



Specification for insert

Input:

x

xs: sorted list

Output:

ys: sorted list

elems ys = elems xs ∪ {x}

How can I express this formally?

Types!

How can I verify this for all inputs?

Type checking!

Agenda

Today:

- Simple types and how to check them
- Refinement types and how to check them
- Specification for insert as a refinement type

Thursday:

- How to use refinement type checking for synthesis?

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:

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

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

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

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

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

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

← 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

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

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

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

Type checking vs inference

The problem of discovering the derivation of $\Gamma \vdash e :: T$ is called *type 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?
- But: can also leverage top-level type to make checking more efficient

Function types

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

Syntax of terms (programs)

$$\mid x \mid e\ e \mid \lambda x.\, e$$

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

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

Exercise 1

Infer the type of $\lambda x.\, inc\ x$ in $\Gamma = [inc\colon \mathrm{Int} \to \mathrm{Int}]$ 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 $$\frac{\Gamma; x\colon T \vdash e :: T'}{\Gamma \vdash \lambda x\colon T.\, e :: T \to T'}$$

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

Inference algorithm

Goal: compute the type of term from the types of subterms

$$\text{T-app}\ \dfrac{\text{T-var}\ \dfrac{}{\Gamma \vdash \mathit{inc}\ ::} \qquad \dfrac{}{\Gamma \vdash 1\ ::}\ \text{T-num}}{\Gamma \vdash \mathit{inc}\ 1\ ::}$$

$\Gamma = [\mathit{inc} : \mathrm{Int} \to \mathrm{Int}]$

Inference algorithm

Problem: to compute the types of this term,
we had to *guess* the type of x:

$$
\dfrac{\dfrac{\dfrac{}{\Gamma, x : ? \vdash inc ::}\ \text{T-var} \qquad \dfrac{}{\Gamma, x : ? \vdash x ::}\ \text{T-var}}{\Gamma, x : ? \vdash inc\ x ::}\ \text{T-app}}{\Gamma \vdash \lambda x. inc\ x ::}\ \text{T-abs}
$$

Solution: constraint-based type inference

- aka Hindley-Milner type inference

Constraint-based type inference

[Hindley'69][Milner'78]

Idea: separate inference into constraint generation and constraint solving

1. Whenever you need to guess a type, generate a *type variable*
2. Whenever two types must match, generate a *unification constraint*
3. Solve unification constraints to assign types to type variables

$$\Gamma \vdash \lambda x.\, inc\ x :: ?$$

Example

Type derivation

$$\dfrac{\dfrac{\text{T-var}\;\dfrac{}{\Gamma, x{:}\,\alpha \vdash inc :: \mathrm{Int} \to \mathrm{Int}} \qquad \dfrac{}{\Gamma, x{:}\,\alpha \vdash x :: \alpha}\;\text{T-var}}{\Gamma, x{:}\,\alpha \vdash inc\ x :: \mathrm{Int}}\;\text{T-app}}{\Gamma \vdash \lambda x.\, inc\ x :: \alpha \to \mathrm{Int}}\;\text{T-abs}$$

Type assignment

$$\alpha \;\to\; \mathrm{Int}$$

Unification constraints

$$\alpha \sim \mathrm{Int}$$

$$\Gamma = [inc{:}\,\mathrm{Int} \to \mathrm{Int}]$$

What about type checking?

Type derivation

$$\dfrac{\dfrac{}{\Gamma, x{:}\, \alpha \vdash inc :: \mathrm{Int} \to \mathrm{Int}}\text{T-var} \quad \dfrac{}{\Gamma, x{:}\, \alpha \vdash x :: \alpha}\text{T-var}}{\dfrac{\Gamma, x{:}\, \alpha \vdash inc\ x :: \mathrm{Int}}{\Gamma \vdash \lambda x.\, inc\ x :: \alpha \to \mathrm{Int}}\text{T-abs}}\text{T-app}$$

$$\mathrm{Int} \to \mathrm{Int}$$

Type assignment

$$\alpha \ \to \ \mathrm{Int}$$

Unification constraints

$$\alpha \sim \mathrm{Int}$$

$$\alpha \to \mathrm{Int} \sim \mathrm{Int} \to \mathrm{Int}$$

$$\Gamma = [inc{:}\, \mathrm{Int} \to \mathrm{Int}]$$

Can we do better?

Type derivation

$$\dfrac{\dfrac{\overline{\Gamma, x : \mathrm{Int} \vdash inc :: \mathrm{Int} \to \mathrm{Int}}\ \text{T-var} \qquad \overline{\Gamma, x : \mathrm{Int} \vdash x :: \mathrm{Int}}\ \text{T-var}}{\Gamma, x : \mathrm{Int} \vdash inc\ x :: \mathrm{Int}}\ \text{T-app}}{\Gamma \vdash \lambda x.\, inc\ x :: \mathrm{Int} \to \mathrm{Int}}\ \text{T-abs}$$

Type assignment

Unification constraints

$$\mathrm{Int} \sim \mathrm{Int}$$

$$\Gamma = [inc : \mathrm{Int} \to \mathrm{Int}]$$

Bidirectional type-system

[Pierce, Turner'00]

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 $\dfrac{\Gamma \vdash e \uparrow T' \quad \Gamma \vdash T \sim T'}{\Gamma \vdash e \downarrow T}$

Polymorphism (aka "generics")

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

Terms

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

$$\mid T_1 \rightarrow T_2$$ (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 $$\frac{\Gamma \vdash e :: \forall \alpha.\, S \qquad \Gamma \vdash T}{\Gamma \vdash e :: S[\alpha \mapsto T]}$$

Exercise 3

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

using the following rules:

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

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

$$\text{T-app} \quad \frac{\Gamma \vdash e_1 :: T \rightarrow T' \qquad \Gamma \vdash e_2 :: T}{\Gamma \vdash e_1\ e_2 :: T'}$$

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

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

Agenda

Today:

- Simple types and how to check them
- Refinement types and how to check them
- Specification for insert as a refinement type

Thursday:

- How to use refinement type checking for synthesis?

Types as specifications

```
insert :: ∀ a . a  →  List a  →  List a
```

Conventional types are not enough

```
// Insert x into a sorted list xs
insert :: x:a  →  xs:List a  →  List a
insert x xs =
  match xs with
    Nil → Nil
    Cons h t →
      if x ≤ h
        then Cons x xs
        else Cons h (insert x t)
```

Refinement types

[Rondon et al.'08]

```
Nat
```

base types

```
max :: x: Int → y: Int → { v: Int | x ≤ v ∧ y ≤ v }
```

dependent function types

```
xs :: { v: List Nat }
```

polymorphic datatypes

```
data List α where
    Nil  ::  { List α | len v = 0 }
    Cons  ::  x: α  → { List α | len v = len
                        xs + 1 }
```

```
measure len :: List α → Int
    len Nil = 0
    len (Cons _ xs) = len xs + 1
```

Refinement types

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

Terms

$$T \ ::= \{\mathrm{v}{:}\ \mathrm{B} \mid \mathrm{e}\}$$ (basic types)

$$\mid x{:}T_1 \rightarrow T_2$$ (function types)

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

Types

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

Type schemas

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

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

T-app
$$\frac{\Gamma \vdash e_1 :: x{:}T \rightarrow T' \qquad \Gamma \vdash e_2 :: T}{\Gamma \vdash e_1\ e_2 :: T'[x \mapsto e_2]}$$

Example

Let's check that $\Gamma \vdash \mathrm{inc}\ 5 :: \mathrm{Nat}$

- $\mathrm{Nat} = \{\nu : \mathrm{Int} \mid \nu \geq 0\}$
- $\Gamma = [\mathrm{inc}: y: \mathrm{Int} \rightarrow \{\nu : \mathrm{Int} \mid \nu = y + 1\}]$

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

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

$$\text{T-var} \quad \frac{(x: T \in \Gamma)}{\Gamma \vdash x :: \{\nu : T \mid \nu = x\}}$$

$$\text{T-app} \quad \frac{\Gamma \vdash e_1 :: x: T \rightarrow T' \qquad \Gamma \vdash e_2 :: T}{\Gamma \vdash e_1\ e_2 :: T'[x \mapsto e_2]}$$

We need subtyping!

Subtyping

Intuitively, T' is a subtype of T if all values of type T' also belong to T

- written $T' <: T$
- e.g. $\text{Nat} <: \text{Int}$ or $\{\nu : \text{Int} \mid \nu = 5\} <: \text{Nat}$

Defined via inference rules:

$$\text{Sub-base}\ \frac{[\![\Gamma]\!] \wedge e' \Rightarrow e}{\Gamma \vdash \{v\colon B \mid e'\} <: \{v\colon B \mid e\}}$$

$$\text{Sub-fun}\ \frac{\Gamma \vdash T_1 <: T_1' \qquad \Gamma; x\colon T_1 \vdash T_2' <: T_2}{\Gamma \vdash x\colon T'_1 \rightarrow T_2' <: x\colon T_1 \rightarrow T_2}$$

Refinement type inference

[Rondon et al.'08]

Idea: separate inference into (subtyping) constraint generation and (subtyping) constraint solving

1. Whenever you need to guess a type, generate a *type variable*
2. Whenever two types must match, generate a *subtyping constraint*
3. Solve subtyping constraints to assign refined types to type variables

$$\Gamma \vdash \lambda x.\, \text{inc}\ x :: \text{Nat} \to \text{Nat}$$

Example

Type derivation

$$\dfrac{\dfrac{\dfrac{}{\Gamma, x{:}\,\alpha \vdash inc :: y{:}\,\text{Int} \to \{\nu{:}\,\text{Int} \mid \nu = y + 1\}}\;\text{T-var} \qquad \dfrac{}{\Gamma, x{:}\,\alpha \vdash x :: \alpha}\;\text{T-var}}{\Gamma, x{:}\,\alpha \vdash inc\ x :: \{\nu{:}\,\text{Int} \mid \nu = x + 1\}}\;\text{T-app}}{\Gamma \vdash \lambda x.\, inc\ x :: x{:}\alpha \to \{\nu{:}\,\text{Int} \mid \nu = x + 1\}}\;\text{T-abs}$$

$$\text{Nat} \to \text{Nat}$$

Subtyping constraints

$$\alpha <: \text{Int}$$

$$\cancel{x{:}\alpha \to \{\nu{:}\,\text{Int} \mid \nu = x + 1\} <: \text{Nat} \to \text{Nat}}$$

$$\text{Nat} <: \alpha$$

$$x{:}\,\text{Nat} \vdash \{\nu{:}\,\text{Int} \mid \nu = x + 1\} <: \text{Nat}$$

Type assignment

$$\alpha \to \{\nu{:}\,\text{Int} \mid P\}$$

$$P \to true$$

Horn clauses

$$P \Rightarrow true$$

$$\nu \geq 0 \Rightarrow P$$

$$x \geq 0 \land \nu = x + 1 \Rightarrow \nu \geq 0$$

$$\Gamma = [\text{inc}{:}\, y{:}\,\text{Int} \to \{\nu{:}\,\text{Int} \mid \nu = y + 1\}]$$

Agenda

Today:

- Simple types and how to check them
- Refinement types and how to check them
- Specification for insert as a refinement type

Thursday:

- How to use refinement type checking for synthesis?

Specification for insert

Input:

x

xs: sorted list

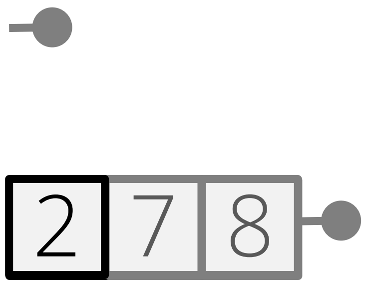
Output:

ys: sorted list

elems ys = elems xs ∪ {x}

Refinement types: sorted lists

```
data List a where
  Nil  :: List a
  Cons :: h:a →
          t:List a →
          List a
```



all you need
is one simple predicate!

Refinement types as specs

[Rondon et al. PLDI'08]

```
// Insert x into a sorted list xs
insert :: x:a  →  xs:SList a  →
            {ν:SList a | 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)
```

Expected
{ν:SList e|elems ν = elems xs ∪ {x}}
and got
{ν:SList e|elems xs ⊆ elems ν}

Incomplete programs?

```
// Insert x into a sorted list xs
insert :: x:a  →  xs:SList a  →
          {ν:SList a | elems ν = elems xs ∪ {x}}
insert x xs =
  match xs with
    Nil → Nil
    Cons h t → ...
```

Bidirectional type checking

```
{ν:SList a | elems ν = {x}}
```

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