Lecture 11
Type-Driven Synthesis

Module II

Behavioral constraints

assertions
types
pre/post-conditions

+ bounded guarantees

+ unbounded guarantees

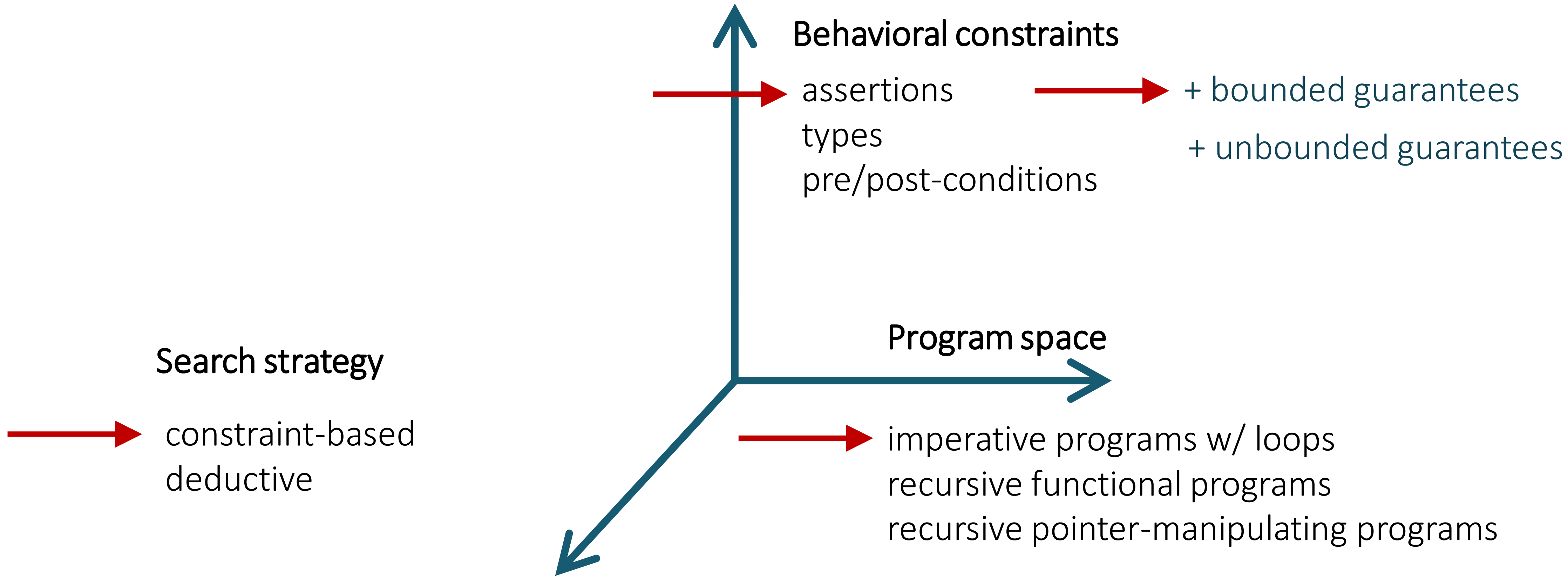
Program space

imperative programs w/ loops
recursive functional programs
recursive pointer-manipulating programs

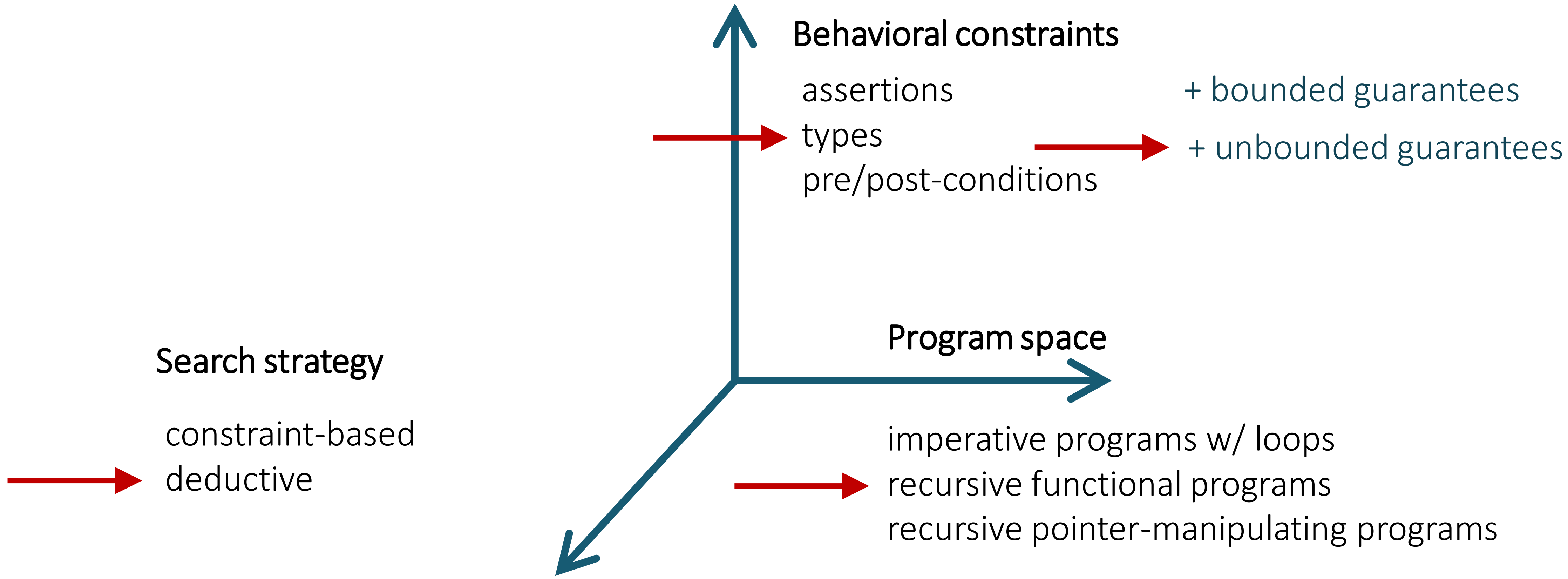
Search strategy

constraint-based
deductive

Last 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

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

Type-driven program synthesis

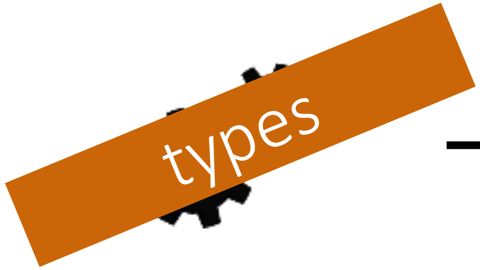
specification search program

programmer-friendly
informative

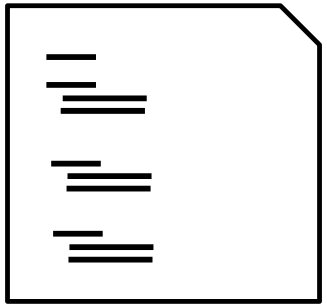
Type-driven program synthesis

specification

search



program



programmer-friendly
informative

help prune
the search space

Which program do I have in mind?

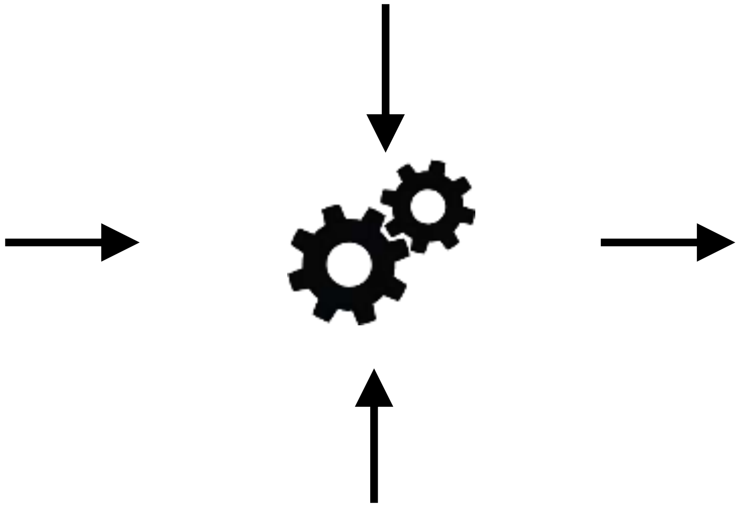
`Char -> String -> [String]` split string at custom separator

`a -> Int -> [a]` list with n copies of input value

Type-driven program synthesis

type system

type



program

context

This week

intro to type systems

enumerating well-typed terms

synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

This week

intro to type systems

enumerating well-typed terms

synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

What is a type system?

Deductive system for proving facts about programs and types

Defined using *inference rules* over *judgments*

typing judgement

"under context Gamma, term e has type T"

A simple type system: syntax

example program: increment

$$\lambda x.\, x + 1$$

A simple type system: syntax

Inference rules = typing rules

Typing derivations

A derivation of $\Gamma \vdash e :: T$ is a tree where

1. the root is $\Gamma \vdash e :: T$
2. children are related to parents via inference rules
3. all leaves are axioms

Typing derivations

let's build a derivation of

$$\cdot \vdash \lambda x.\, x + 1 :: \mathrm{Int} \rightarrow \mathrm{Int}$$

we say that $\lambda x.\, x + 1$ is well-typed in the empty context

and has type $\mathrm{Int} \rightarrow \mathrm{Int}$

Typing derivations

$$\cdot \vdash \lambda x. x + 1 :: \mathrm{Int} \to \mathrm{Int}$$

Typing derivations

is $(\lambda x.\, x) + 1$ well-typed (in the empty context)?

no! no way to build a derivation of $\cdot \vdash (\lambda x.\, x) + 1 :: \_$

we say that $(\lambda x.\, x) + 1$ is ill-typed

Let’s add lists!

Example program: head with default

$$\lambda x.\,\mathrm{match}\ x\ \mathrm{with}\ nil \to 0 \mid y{:}ys \to y$$

Typing rules

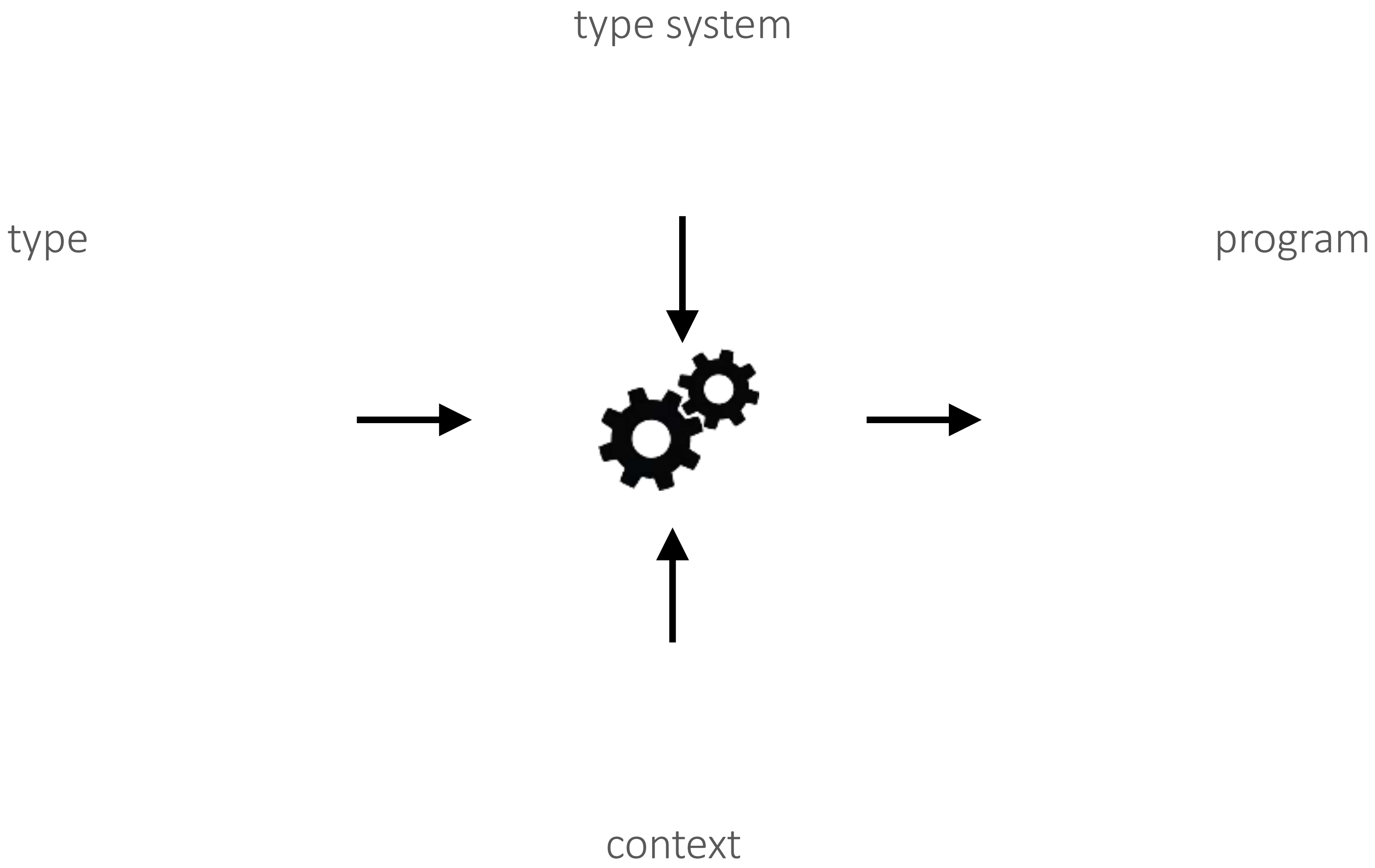
what should the t-match tule be?

Typing rules

Example: head with default

$$\cdot \vdash \lambda x.\,\mathrm{match}\ x\ \mathrm{with}\ nil \to 0 \mid y{:}\,ys \to y \;::\; \mathrm{List} \to \mathrm{Int}$$

Type system → synthesis



This week

intro to type systems

enumerating well-typed terms

synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

Enumerating well-typed terms

how should I enumerate all terms of type $\mathrm{List} \to \mathrm{List}$?

(up to depth 2, in the empty context)

naïve idea: syntax-guided enumeration

1. enumerate all terms *generated by the grammar*
2. type-check each term and throw away ill-typed ones

Syntax-guided enumeration

31 complete programs enumerated

only 2 have the type $\text{List} \rightarrow \text{List}$!

can we do better?

Enumerating well-typed terms

how should I enumerate all terms of type $\mathrm{List} \rightarrow \mathrm{List}$?

(up to depth 2, in the empty context)

better idea: type-guided enumeration

enumerate all derivations *generated by the type systems*

extract terms from derivations (well-typed by construction)

Synthesis as proof search

input: synthesis goal $\Gamma \vdash ? :: T$

output: derivation of $\Gamma \vdash e :: T$ for some e

search strategy: top-down enumeration of derivation trees

like syntax-guided top-down enumeration but

derivation trees instead of ASTs

typing rules instead of grammar

Type-guided enumeration

only 2 programs fully constructed!

all other programs *rejected early*

What's wrong with this search?

Enumerated 3 programs:

nil

$(\lambda x. x)\ nil$

$(\lambda x. nil)\ nil$

They are all equivalent!

Redundant programs

Generating programs on the left is a waste of time!

Idea: only generate programs *in normal form*

Restrict type system to make redundant programs ill-typed

Normal-form programs

elimination forms

introduction forms

base types

types

Bidirectional typing judgments

“under context Gamma, i checks against type T”

“under context Gamma, e generates type T”

[Pierce, Turner. Local Type Inference. 2000]

Bidirectional typing rules

Type-guided enumeration

This week

intro to type systems

enumerating well-typed terms

synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

Simple types are not enough

specification

code

“duplicate every element in a list”

```
stutter :: List → List
```

$\lambda xs.xs$

Simple types are not enough

specification

code

“insert element
into so l list”

ambiguous!

```
insert ::
  In    List → List
```

$$\lambda x.\, \lambda xs.\, xs$$

Type-driven synthesis in 3 easy steps

1. Annotate types with extra specs

    examples, logical predicates, resources, …

2. Design a type system for annotated types

    propagate as much info as possible from conclusion to premises

3. Perform type-directed enumeration as before

This week

intro to type systems

enumerating well-typed terms

synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

Type + examples

specification

"duplicate every element in a list"

→

Myth

→

code

$$
\begin{aligned}
&\text{fix } f(xs).\ \text{match } xs \text{ with} \\
&\quad nil \to nil \\
&\mid\ y{:}\,ys \to y{:}\,y{:}\,f(ys)
\end{aligned}
$$

[Osera, Zdancewic , Type-and-Example-Directed Program Synthesis. 2015]

Types + examples: syntax

values

vectors of examples

type refined with examples

context

Example: singleton

no search! simply propagate the spec top-down

Type-driven synthesis in 3 easy steps

1. Annotate types with examples
2. Design a type system for annotated types
3. Perform type-directed enumeration as before

This week

intro to type systems

enumerating well-typed terms

synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

Polymorphic types

Polymorphic types for synthesis

$\lambda x.\,nil$

$\lambda x.\,[0], \lambda x.\,[1], \ldots$

$\lambda x.\,[x]$

$\lambda x.\,[\text{double } 0], \lambda x.\,[\text{dec } 0]$

$\lambda x.\,[0,0], \lambda x.\,[0,1], \ldots$

$\lambda x.\,[x, x]$

which of these programs match the polymorphic type?

Polymorphic types for synthesis

$\lambda x.\, nil$

$\lambda x.\, [0], \lambda x.\, [1], \ldots$

$\lambda x.\, [x]$

$\lambda x.\, [\text{double } 0], \lambda x.\, [\text{dec } 0]$

$\lambda x.\, [0,0], \lambda x.\, [0,1], \ldots$

$\lambda x.\, [x, x]$

1. $\lambda x.\, nil$

eliminate ambiguity!

2. $\lambda x.\, [x]$

prune the search!

3. $\lambda x.\, [x, x]$

Polymorphic types

base types

types

type schemas (polytypes)

contexts

Judgments

type checking:

"under context Gamma, i checks against a schema S"

type inference:

"under context Gamma, e generates type T"

Typing rules

how do we guess T'?
Hindley-Milner type inference!

This week

intro to type systems

enumerating well-typed terms

synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

Refinement types

```
Nat
```

base types

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

dependent function types

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

polymorphic datatypes

[Rondon, Kawaguchi, Jhala. PLDI'2008]

Refinement types: measures

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

syntactic sugar:

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

example: duplicate every element in a list

```
stutter :: ??
```

Refinement types: sorted lists

```
data SList α where
    Nil   ::  SList α
    Cons   ::  x: α  → xs: SList {α | x ≤ ν }
                     → SList α
```

example: insert an element into a sorted list

```
insert :: ??
```

Refinement types

base types

types

type schemas (polytypes)

contexts

Example: increment

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

we need subtyping!

$$\Gamma \vdash \lambda x.\, \text{inc}\ x \Leftarrow \text{Nat} \rightarrow \text{Nat}$$

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 $\{v\!: \text{Int} \mid v = 5\} <: \text{Nat}$

sub-base
$$\frac{[\![\Gamma]\!] \wedge \phi' \Rightarrow \phi}{\Gamma \vdash \{\nu\!:\ B \mid \phi'\} <: \{\nu\!:\ B \mid \phi\}}$$

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

$\text{Pos} <: \text{Nat}$ ✔

$\text{Int} \rightarrow \text{Int} <: \text{Int} \rightarrow \text{Nat}$ ✘

$\text{Int} \rightarrow \text{Int} <: \text{Nat} \rightarrow \text{Int}$ ✘

$\text{x:Int} \rightarrow \{\text{Int} \mid \nu = x + 1\} <: \text{Nat} \rightarrow \text{Nat}$ ✔

Typing rules

Example: increment

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

subtyping constraints

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

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

implications

$\nu \geq 0 \Rightarrow \mathit{true}$

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

SMT solver: VALID!

Refinement type checking

idea: separate type checking into
subtyping constraint generation and subtyping constraint solving

1. Generate a constraint for every subtyping premise in derivation
2. Reduce subtyping constraints to implications
3. Use SMT solver to check implications

[Rondon, Kawaguchi, Jhala. PLDI'2008]

This week

intro to type systems

enumerating well-typed terms

synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

Synthesis from refinement types

specification

“duplicate every element in a list”

```
stutter ::
  xs:List a →
  {ν:List a | len ν =
    2 * len xs}
```

code

```
match xs with
 Nil → Nil
 Cons h t →
  Cons h (Cons h (stutter t))
```

[Polikarpova, Kuraj, Solar-Lezama, Program Synthesis from Polymorphic Refinement Types. 2016]

Synthesis from refinement types

specification

“insert element
into sorted list”

```
insert :: x:a →
  xs:SList a →
  {ν:SList a | elems ν =
    elems xs ∪ {x}}
```

code

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

Type-driven synthesis in 3 easy steps

1. Annotate types with logical predicates
2. Design a type system for annotated types
3. Perform type-directed enumeration as before

Type-directed enumeration for insert

```
x:a  →  xs:SList a  →
  {ν:SList a | elems ν = elems xs ∪ {x}}
```

```
insert = ??
```

Type-directed enumeration

```
{ν:SList a | elems ν = elems xs ∪ {x}}
```

↕

```
insert x xs =  ??
```

context:
`x: a`
`xs: SList a`

Type-directed enumeration

$\{\nu\text{:SList a} \mid \text{elems } \nu = \text{elems xs} \cup \{x\}\}$

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

context:
x: a
xs: SList a

Type-directed enumeration

context:
x: a
xs: SList a
elems xs = {}

```
{ν:SList a | elems ν = elems xs ∪ {x}}
```

$\updownarrow$

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

Type-directed enumeration

{ν:SList a | elems ν = elems xs ∪ {x}}

context:
x: a
xs: SList a
elems xs = {}

Constraints:
∀x: {} = {} ∪ {x}

SMT solver: INVALID!

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

The hard part: application

```
x:a  →  xs:SList a  →
  {ν:SList a | elems ν = elems xs ∪ {x}}
```

↕

```
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons h t →
      Cons h (insert x ??)
```

should this program be rejected?

yes!
cannot guarantee output is sorted!

Round-trip type-checking (RTTC)

type checking:

“under context Gamma, i checks against schema S”

type strengthening:

“under context Gamma, e checks against type T *and* generates a stronger type T’”

RTTC rules

The hard part: application

elems will depend on the missing part...

but sortedness we can already check!

$$\{\nu\text{:SList a} \mid \text{elems } \nu = \text{elems xs} \cup \{x\}\}$$

↕

```
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons h t →
      Cons h (insert x ??)
```

The hard part: application

$\{\nu\text{:a} \mid h \le \nu\}$

↕

```
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons h t →
      Cons h (insert x ??)
```

context:
x: a
xs: SList a
h: a
t: SList {a|h≤ν}

Constraints:
∀x, h: h ≤ x

SMT solver: INVALID!

```
insert :: x:τ →
  xs:SList τ →
  SList τ
```

Synquid: contributions

Unbounded correctness guarantees

Round-trip type system to reject incomplete programs

- + GFP Horn Solver

Refinement types can express complex properties in a simple way

- handles recursive, HO functions
- automatic verification for a large class of programs due to polymorphism (e.g. sorted list insert)

Synquid: limitations

User interaction

- refinement types can be large and hard to write
- components need to be annotated (how to mitigate?)

Expressiveness limitations

- some specs are tricky or impossible to express
- cannot synthesize recursive auxiliary functions

Condition abduction is limited to liquid predicates

Cannot generate arbitrary constants

No ranking / quality metrics apart from correctness

Synquid: questions

Behavioral constraints? Structural constraints? Search strategy?

- Refinement types
- Set of components + built-in language constraints
- Top-down enumerative search with type-based pruning

Typo in the example in Section 3.2

- $\{B_0 \mid \bot\} \to \{B_1 \mid \bot\} \to \{\text{List Pos} \mid \text{len } \nu = \cancel{2}5\}$

Can RTTC reject these terms?

`inc ?? :: {Int | v = 5}`

- where `inc :: x:Int → {Int | v = x + 1}`
- NO! don't know if we can find `?? :: {Int | v + 1 = 5}`

`nats ?? :: List Pos`

- where `nats :: n:Nat → {List Nat | len v = n}`
  `Nat = {Int | v >= 0}`, `Pos = {Int | v > 0}`
- YES! `n:Nat → {List Nat | len v = n}` not a subtype of
  `_ → List Pos`

`duplicate ?? :: {List Int | len v = 5}`

- where `duplicate :: xs:List a → {List a | len v = 2*(len xs)}`
- YES! using a consistency check (len v = 2*(len xs) ∧ len v = 5 → UNSAT)