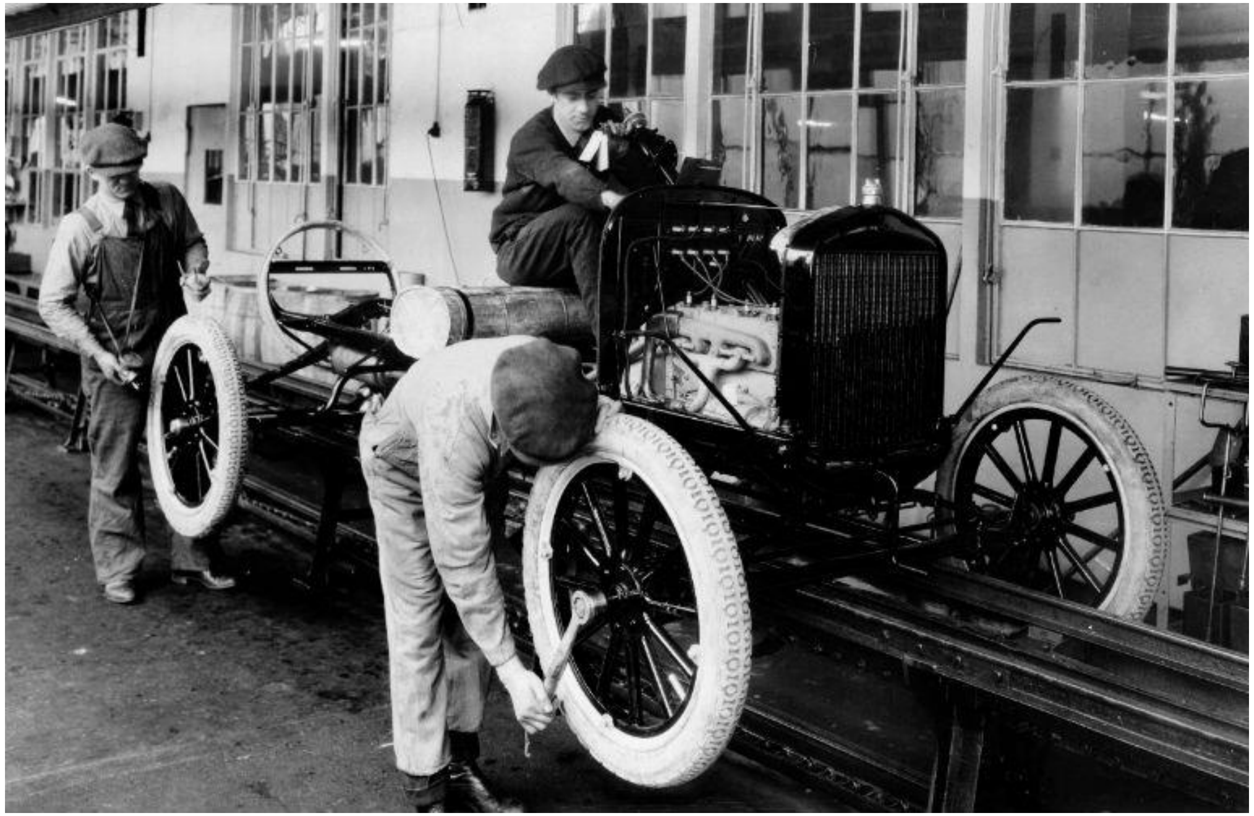
type-driven program synthesis

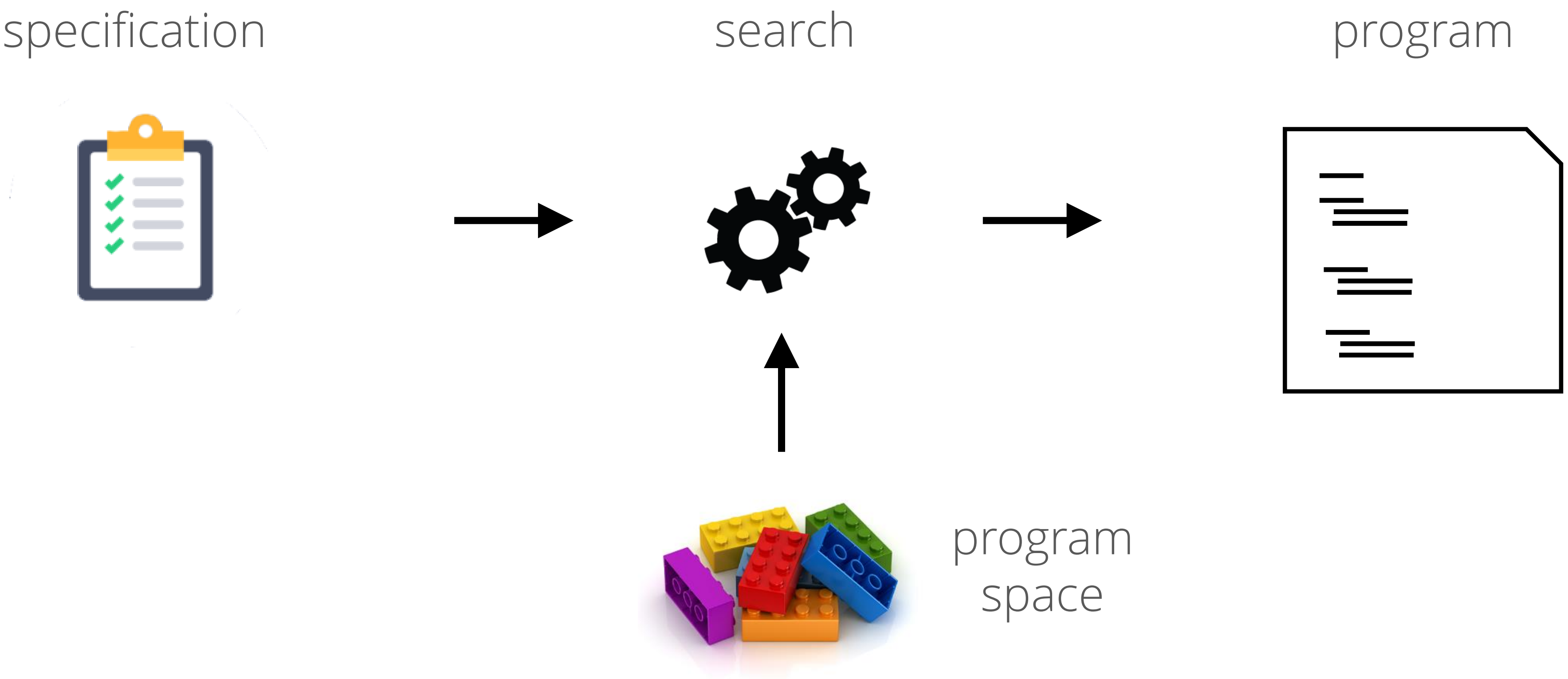
Nadia Polikarpova

UCSDCSE
Computer Science and Engineering

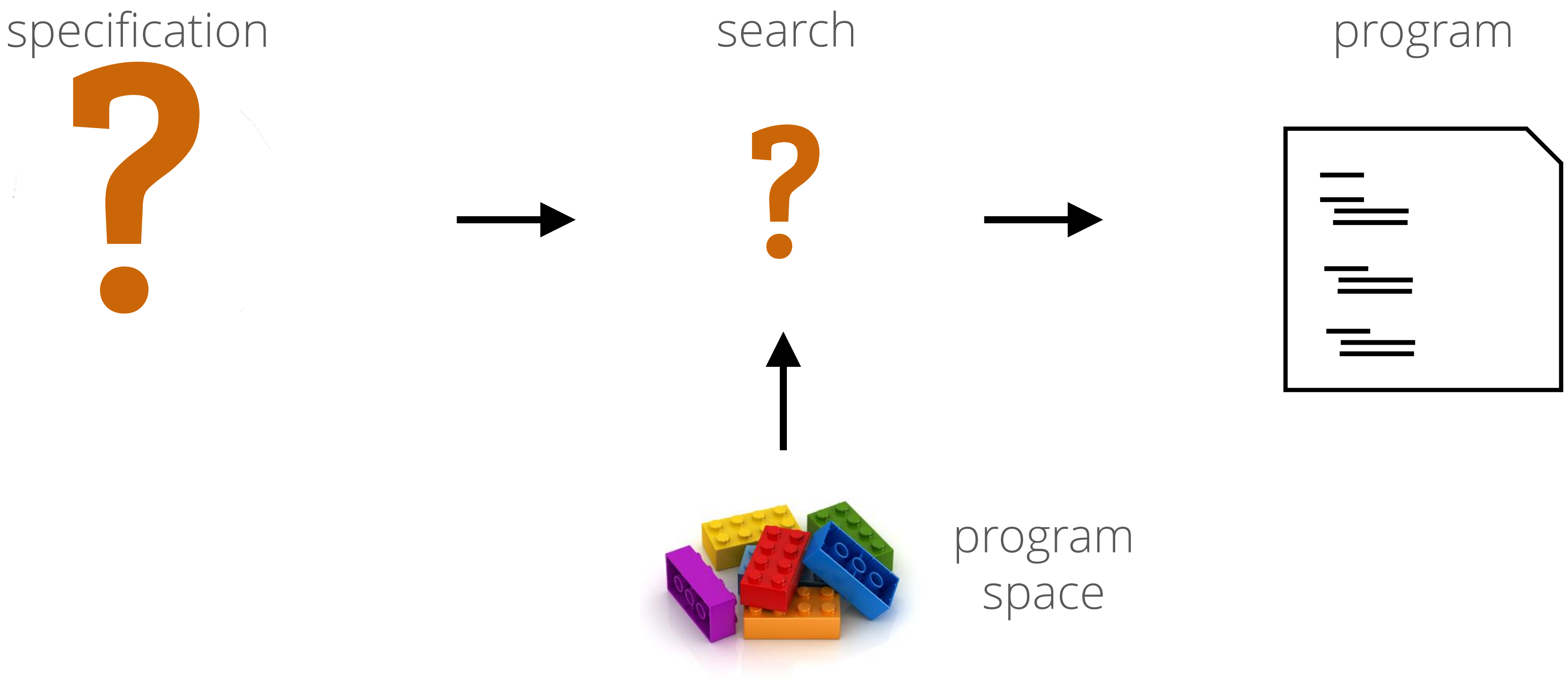
goal: automate programming



program synthesis



program synthesis

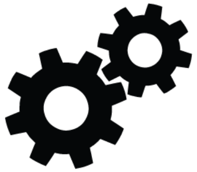


program synthesis

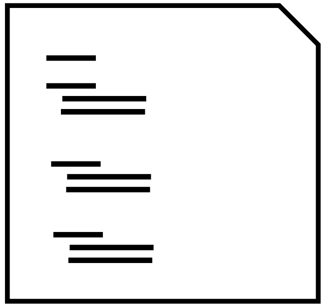
specification

examples
assertions
natural language
…

search



program



program
space

what makes a good spec?

specification

1. programmer-friendly
   easier to write than the program

2. informative
   minimal ambiguity

3. synthesizer-friendly
   easy to check, guides the search

how about **types**?

specification

`a → [a] → [a]`

1. programmer-friendly
   widely used for bug finding, API search, etc
   concise
2. informative
   precise
   advanced types can express many properties
3. synthesizer-friendly
   type checking is automatic and compositional

more programmer-friendly
more ambiguous

less programmer-friendly
less ambiguous

simple types

expressive types

this talk

1. synquid
recursive program synthesis
from refinement types

2. hoogle+
component-based synthesis
from Haskell types

3. future work
best types for synthesis?

simple types

expressive types

this talk

1. synquid
recursive program synthesis
from refinement types

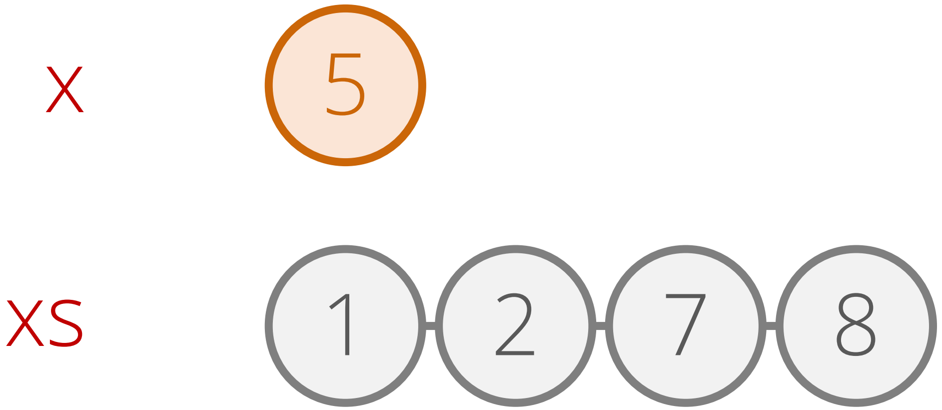
2. hoogle+
component-based synthesis
from Haskell types

3. future work
best types for synthesis?

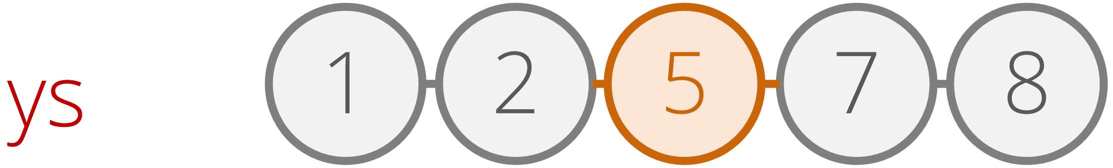
simple types ⟷ expressive types

example: insert into a sorted list

input:



output:

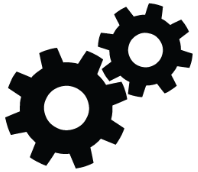


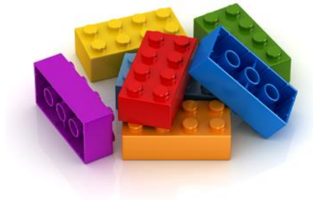
program synthesis

specification

in: x, sorted xs
out: sorted ys
elems ys = elems xs ∪ {x}

formalize this





[], :, ≤

code

```
match xs with
 [] → x : []
 h : t →
   if x ≤ h
     then x : xs
     else h : (insert x t)
```

program synthesis

specification

= type

`a → [a] → [a]`

code

```
match xs with
 [] → x : []
 h : t →
   if x ≤ h
     then x : xs
     else h : (insert x t)
```

program synthesis

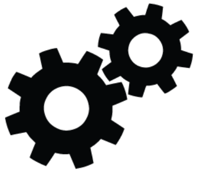
specification

= type

a → → [a]

ambiguous!

→



→

code

```
[]
```

program synthesis

specification

= type

a → [a]

ambiguous!

synquid

code

```
[]
```

POWER TO THE TYPES!

refinement types

$$\{\ \nu : \underbrace{\texttt{Int}}_{\text{shape}}\ \mid\ \underbrace{0 \leq \nu}_{\text{refinement}}\ \}$$

insert in synquid

sorted list

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

set of elements

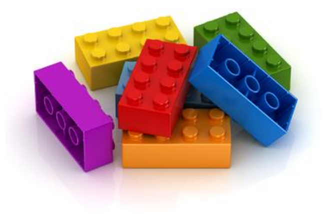
synquid

specification

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

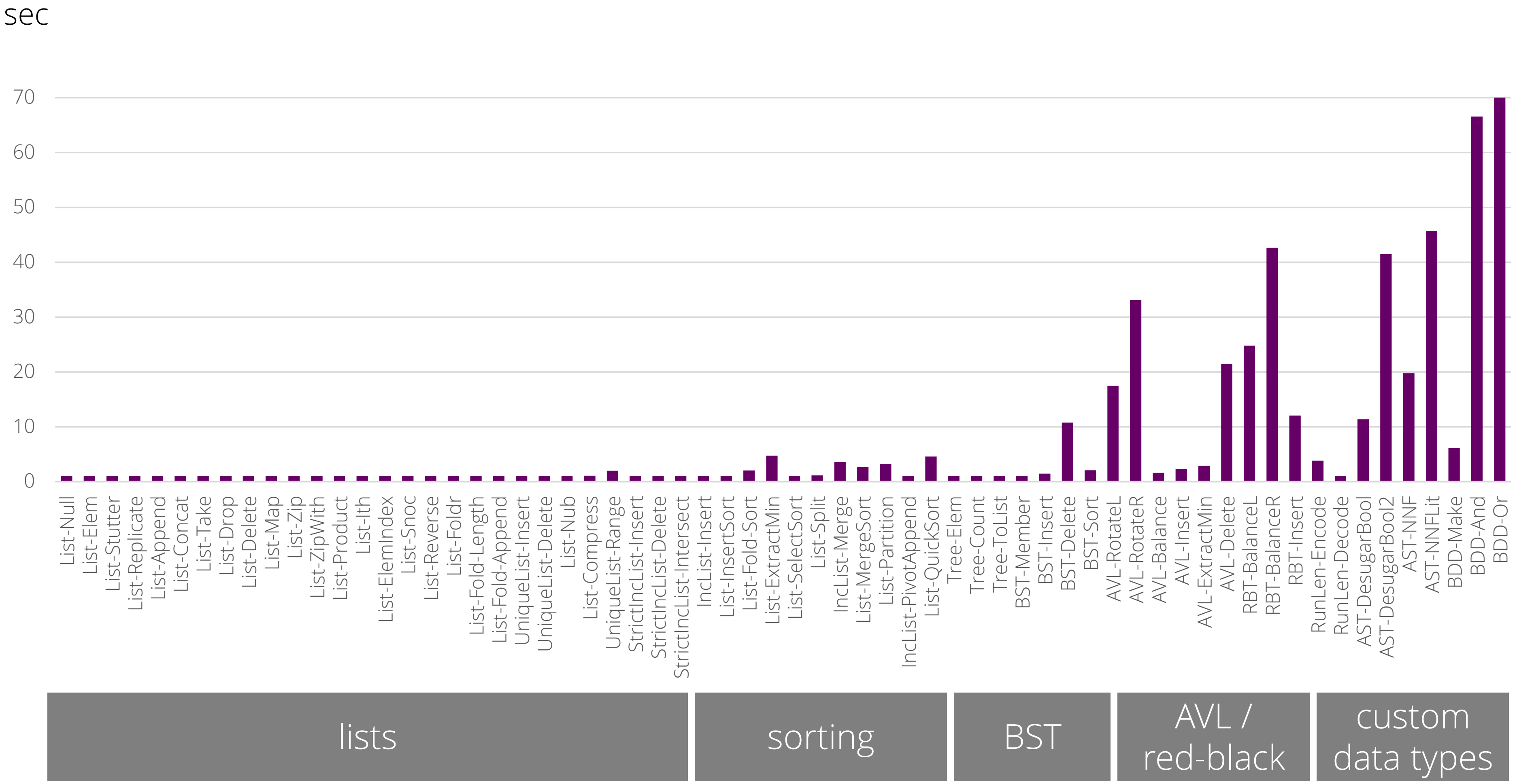
code

```
match xs with
 [] → x : []
 h : t →
   if x ≤ h
     then x : xs
     else h : (insert x t)
```



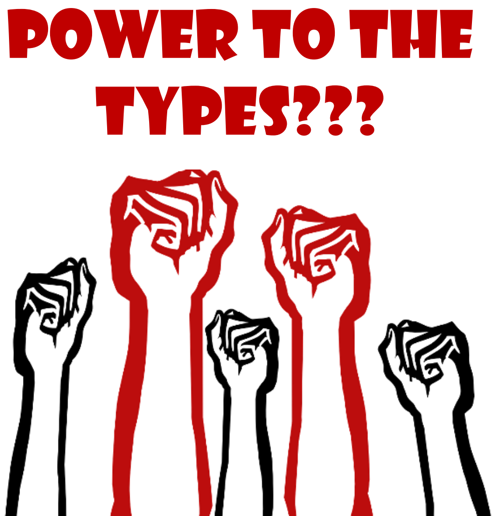
[], :, ≤

what else can it do?



limitation

sometimes writing a (sufficiently) complete
refinement type
is harder than writing the program



this talk

1. synquid
recursive program synthesis
from refinement types

2. hoogle+
component-based synthesis
from Haskell types

3. future work
best types for synthesis?

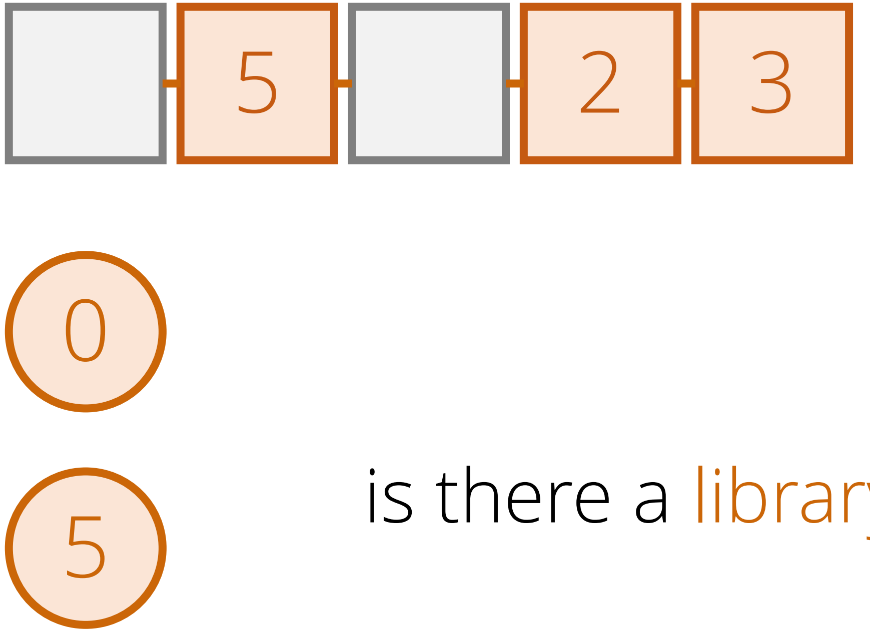
simple types ⟷ expressive types

example: first non-empty

input:

xs

d



output:

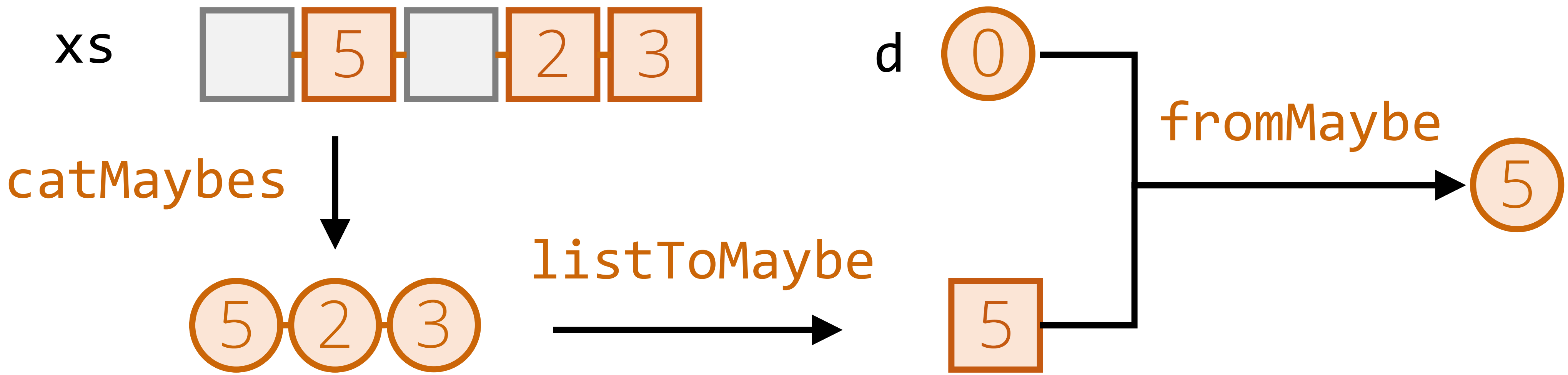
is there a library function for that?

ask [Hoogle]!

example: first non-empty

type query: `xs:[Maybe a] → d:a → a`

solution: `\xs d → fromMaybe d (listToMaybe (catMaybes xs))`



example: first non-empty

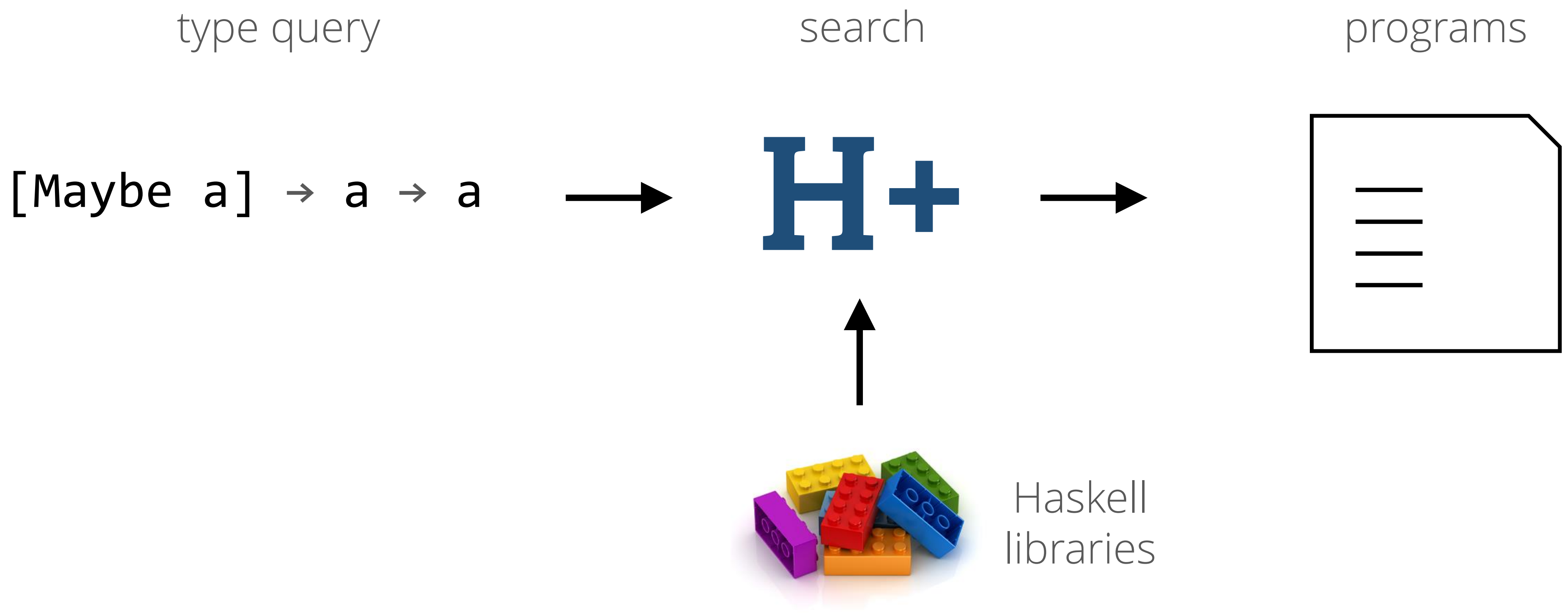
type query: `xs:[Maybe a] → d:a → a`

solution: `\xs d → fromMaybe d (listToMaybe (catMaybes xs))`

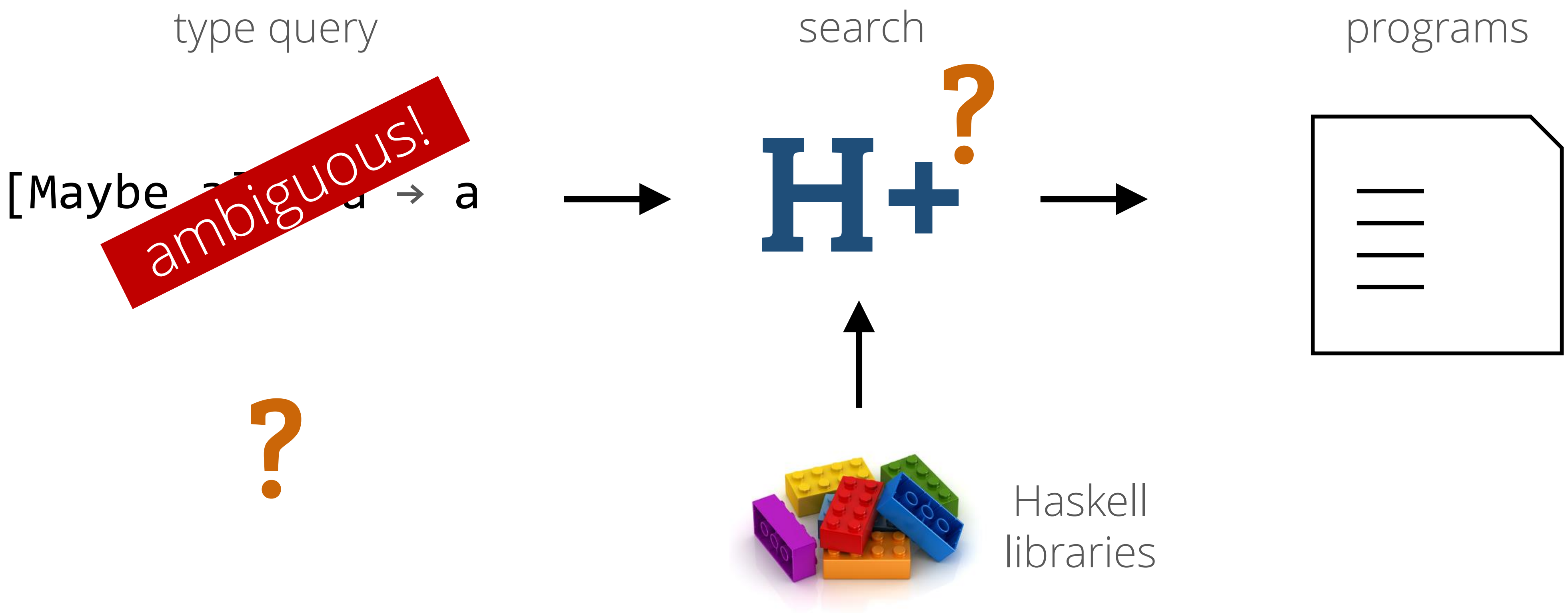
wanted: component-based synthesis

Hoogle+ to the rescue!

Hoogle+



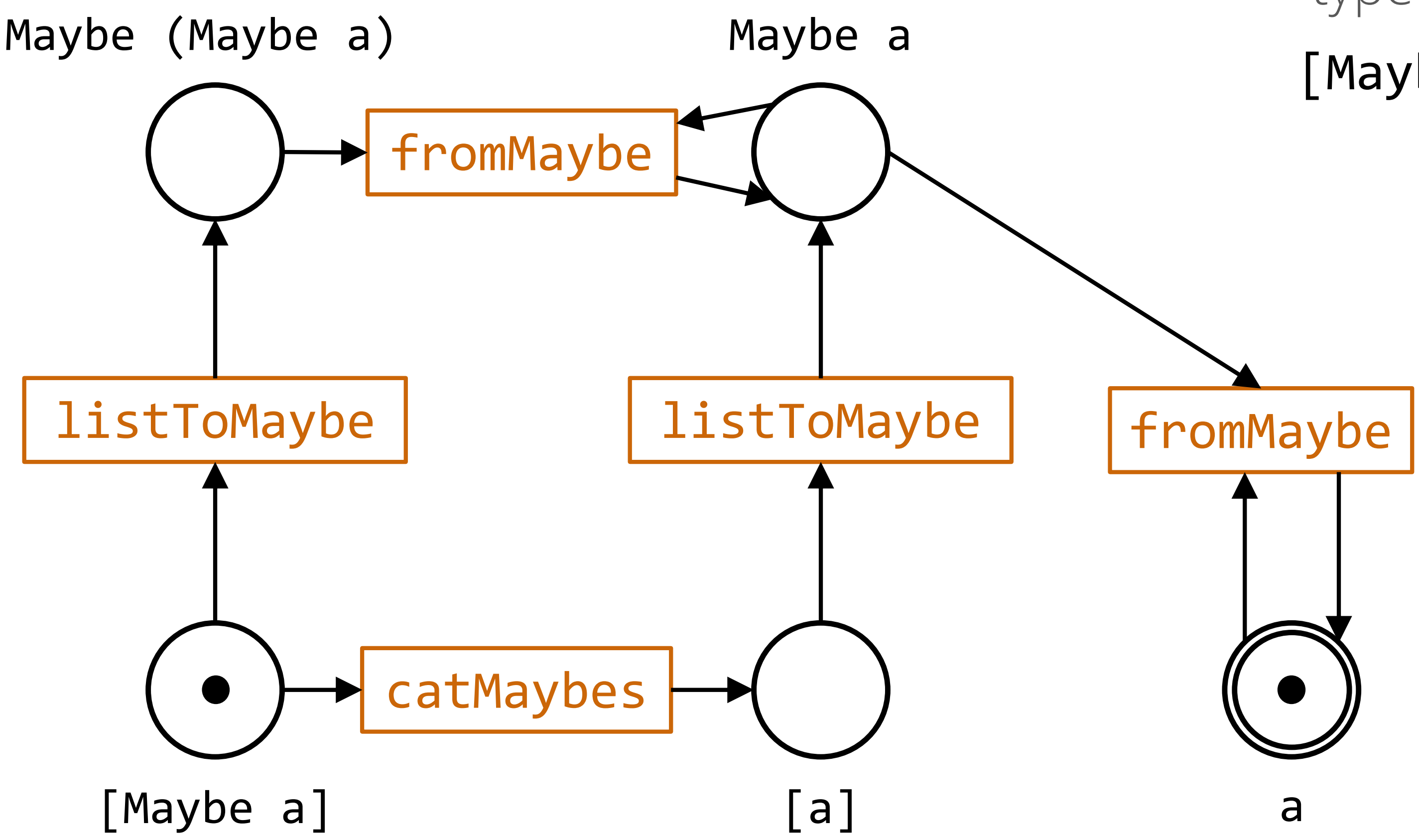
Hoogle+


type query
[Maybe a]
a → a
ambiguous!
search
H+
programs
Haskell
libraries

[Sypet: Feng et al, POPL'17]

prior work: graph-based search

Petri net:



type query:

`[Maybe a] → a → a`

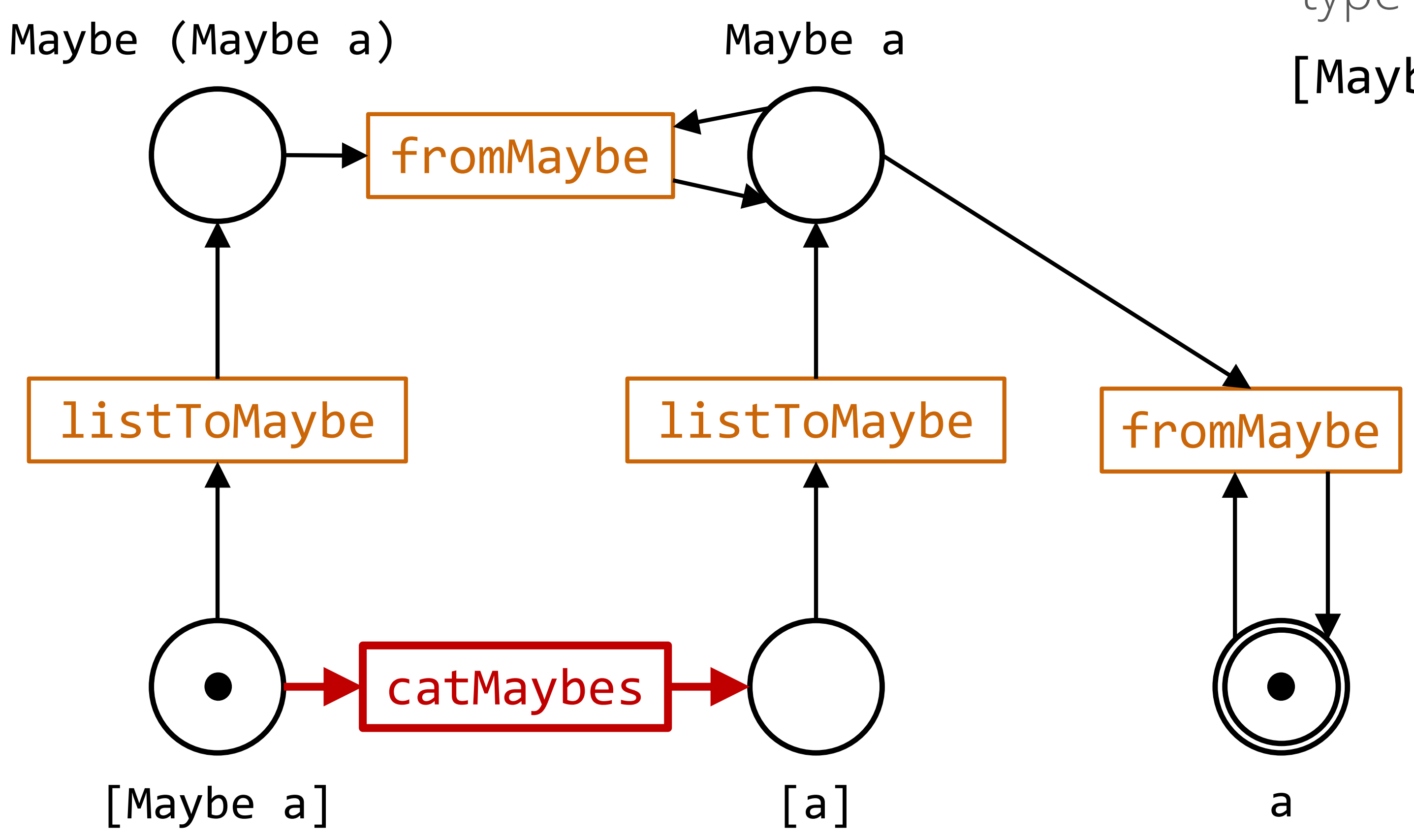
[Sypet: Feng et al, POPL'17]

prior work: graph-based search

type query:

`[Maybe a] → a → a`

Petri net:



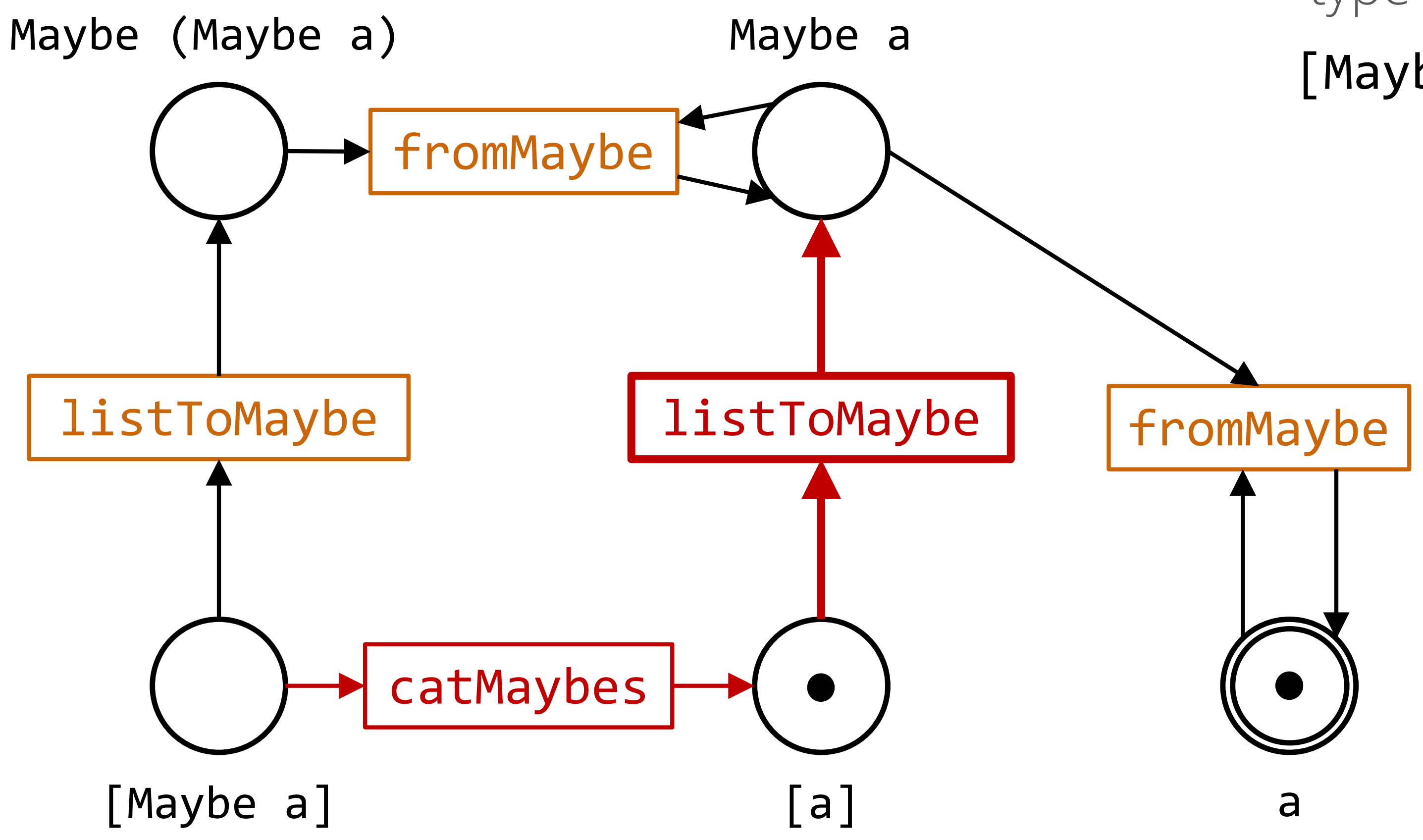
[Sypet: Feng et al, POPL'17]

prior work: graph-based search

type query:

[Maybe a] → a → a

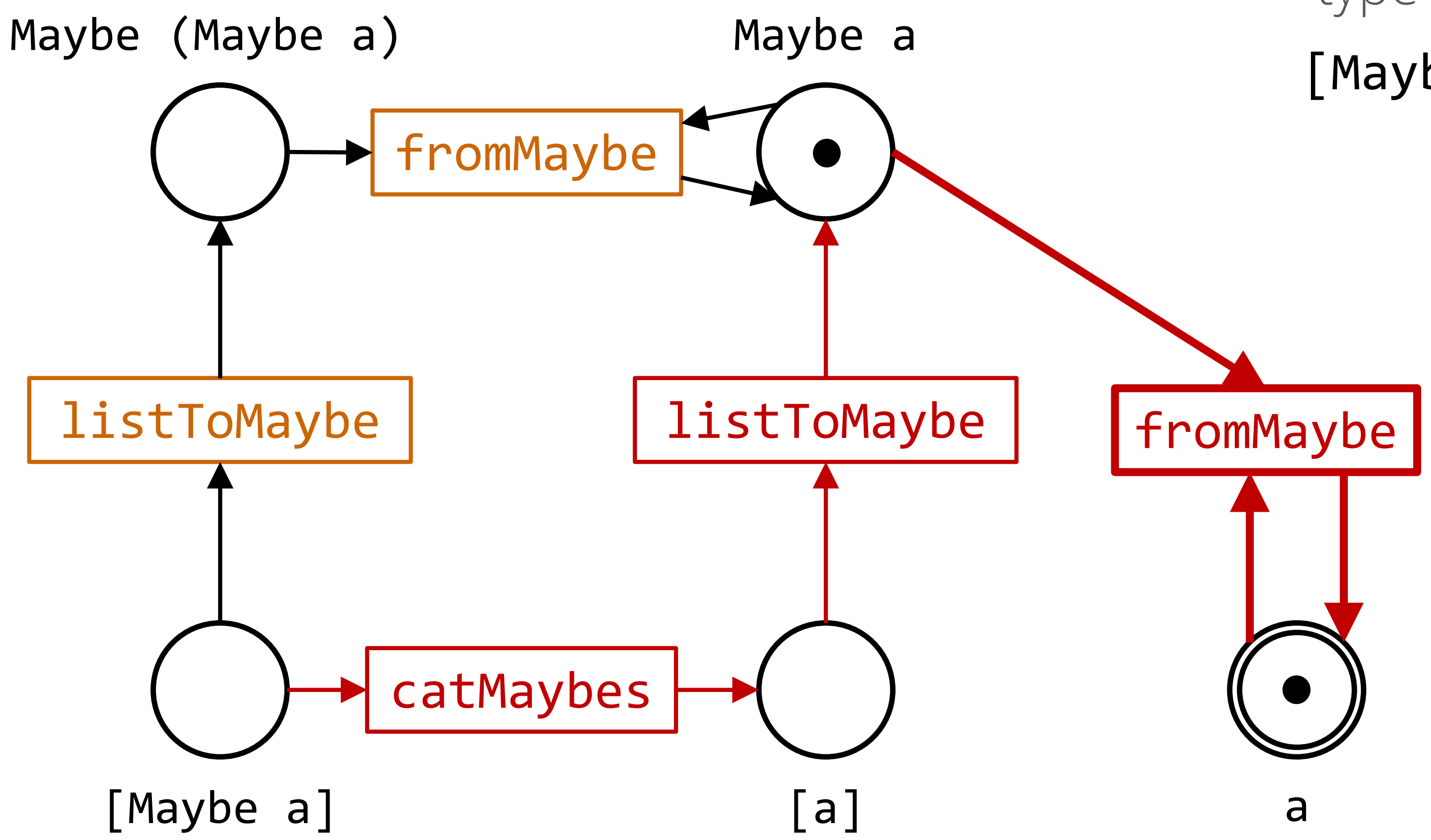
Petri net:



[Sypet: Feng et al, POPL'17]

prior work: graph-based search

Petri net:

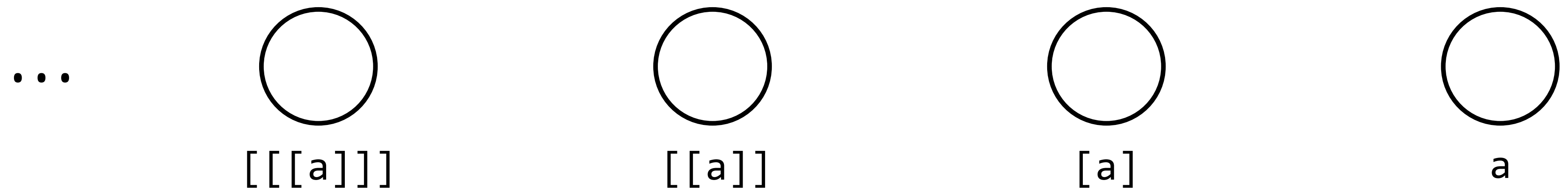


type query:

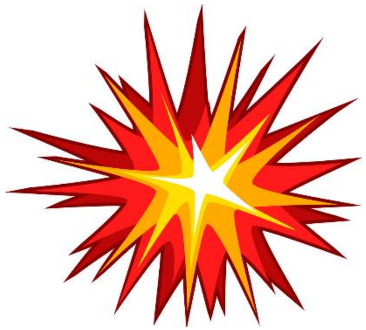
[Maybe a] → a → a

challenge: polymorphism

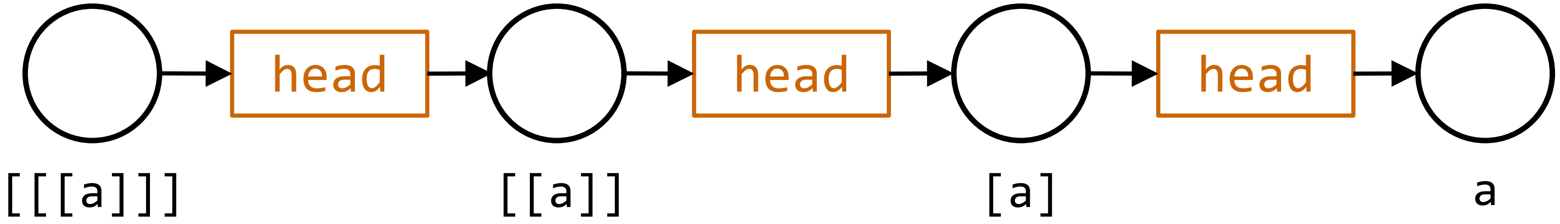
1. infinitely many types



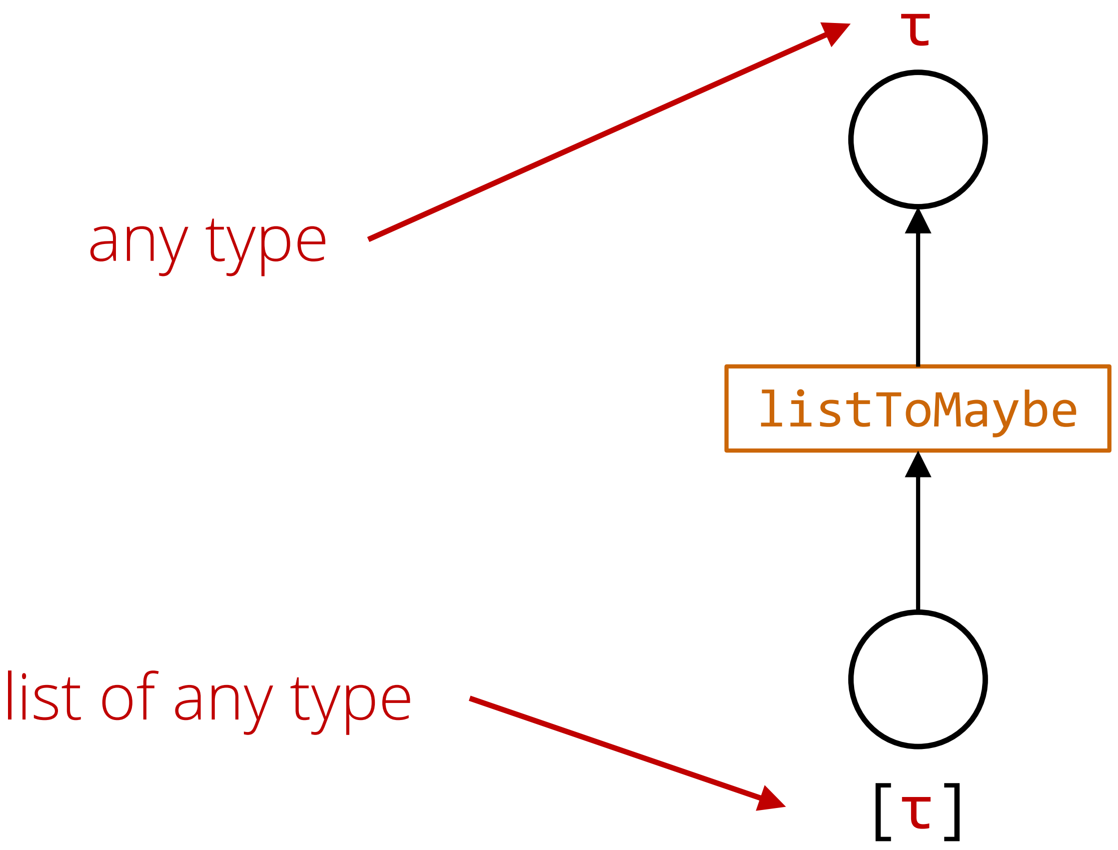
challenge: polymorphism



1. infinitely many types
2. transition explosion

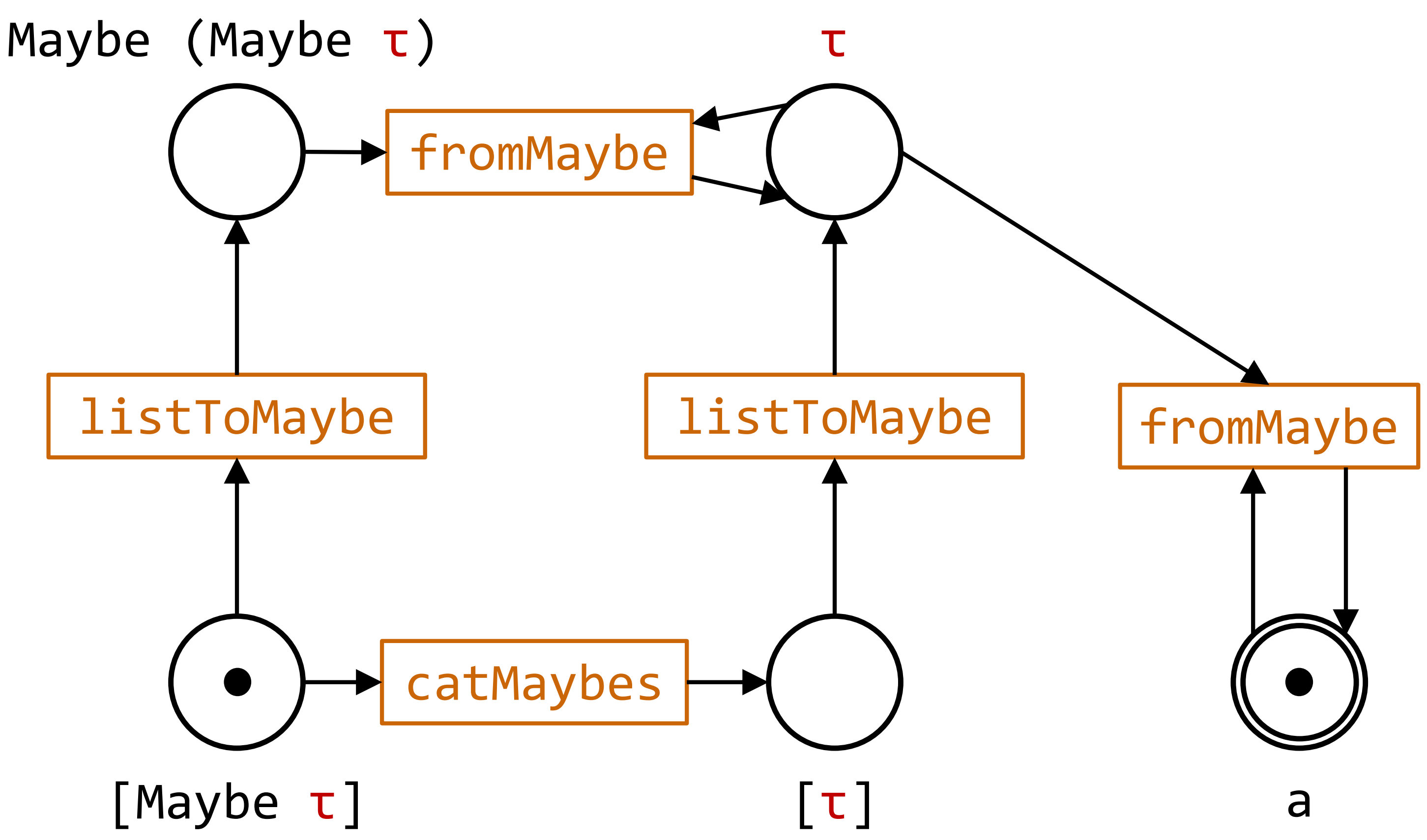


solution: abstraction

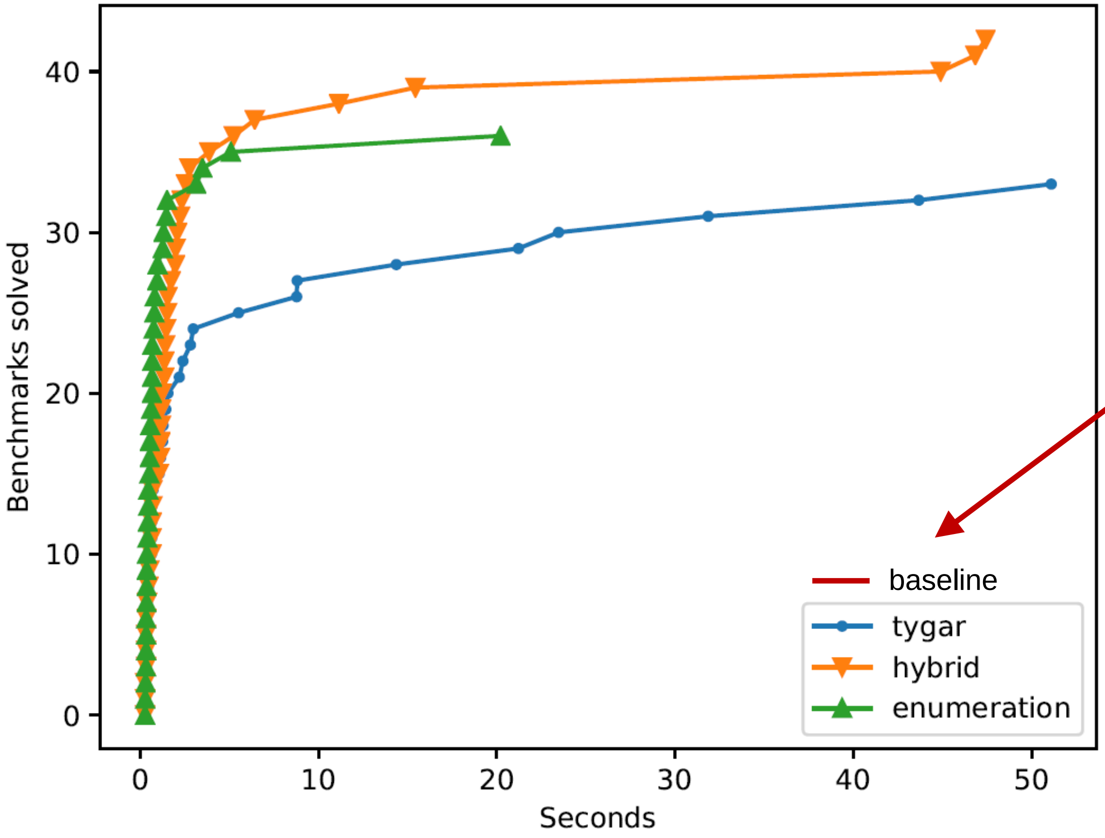


solution: abstraction

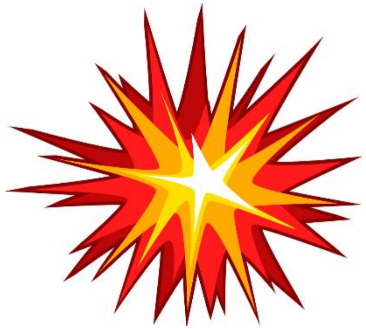
abstract transition net:



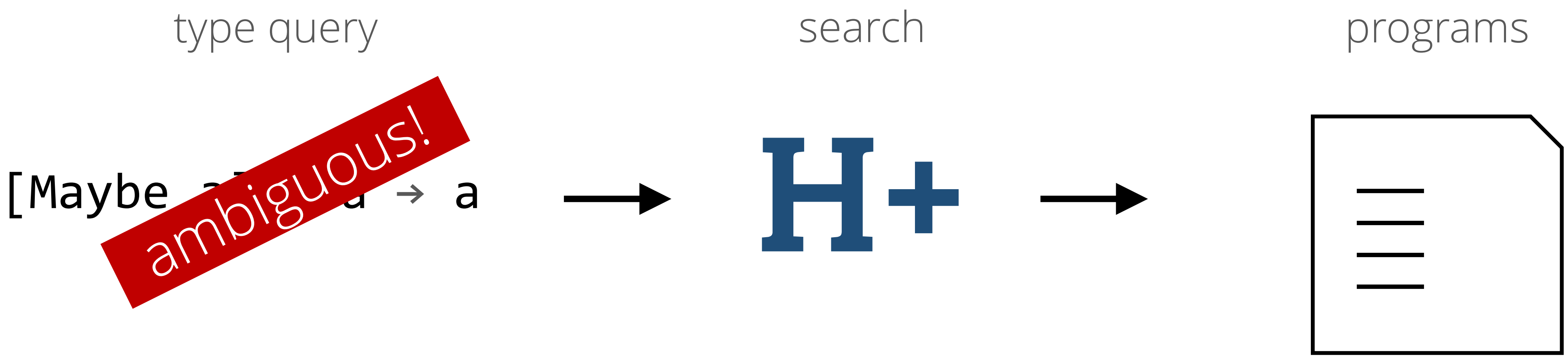
results



spent the whole time
building the net



Hoogle+



? can we filter out irrelevant results without extra input from the user?

three kinds of irrelevant results

type query: `xs:[Maybe a] → d:a → a`

1. discard arguments `\xs d → d`

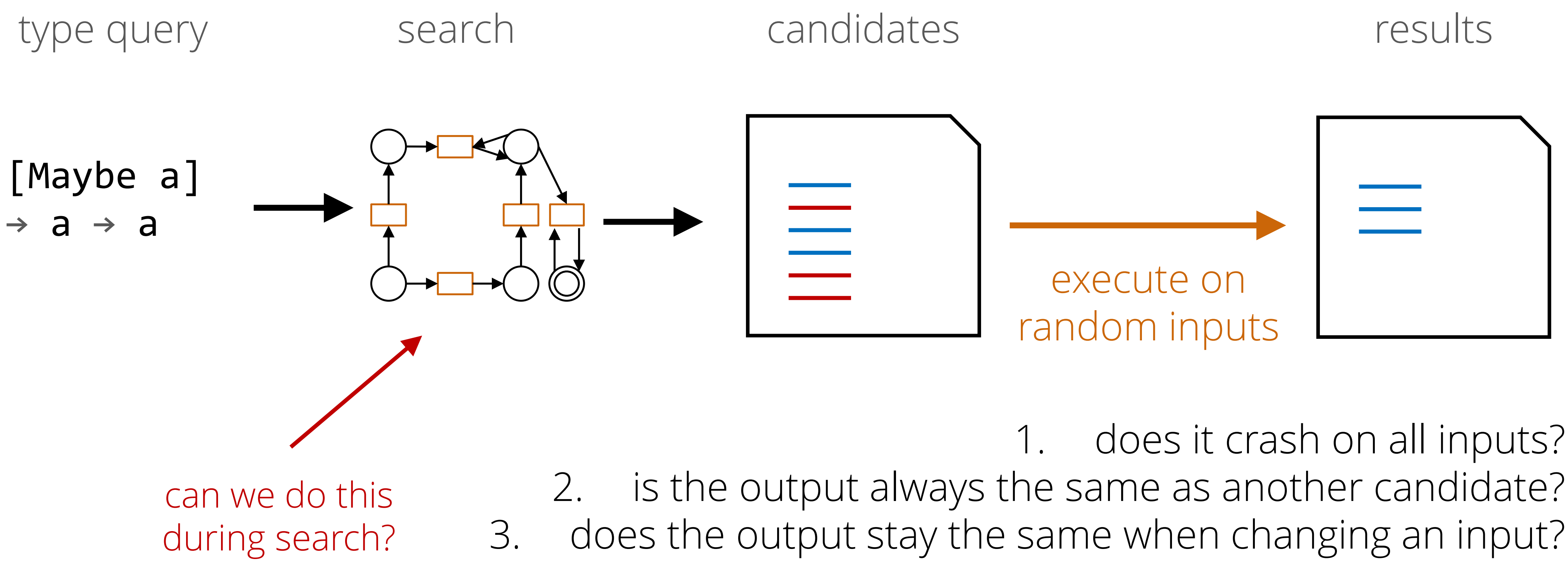
polymorphism makes this worse

2. always crash `\xs d → foldr (head []) d xs`

3. duplicates

```
\xs d → fromMaybe d (head xs)
\xs d → maybe d id (head xs)
```

test-based filtering



1. does it crash on all inputs?
2. is the output always the same as another candidate?
3. does the output stay the same when changing an input?

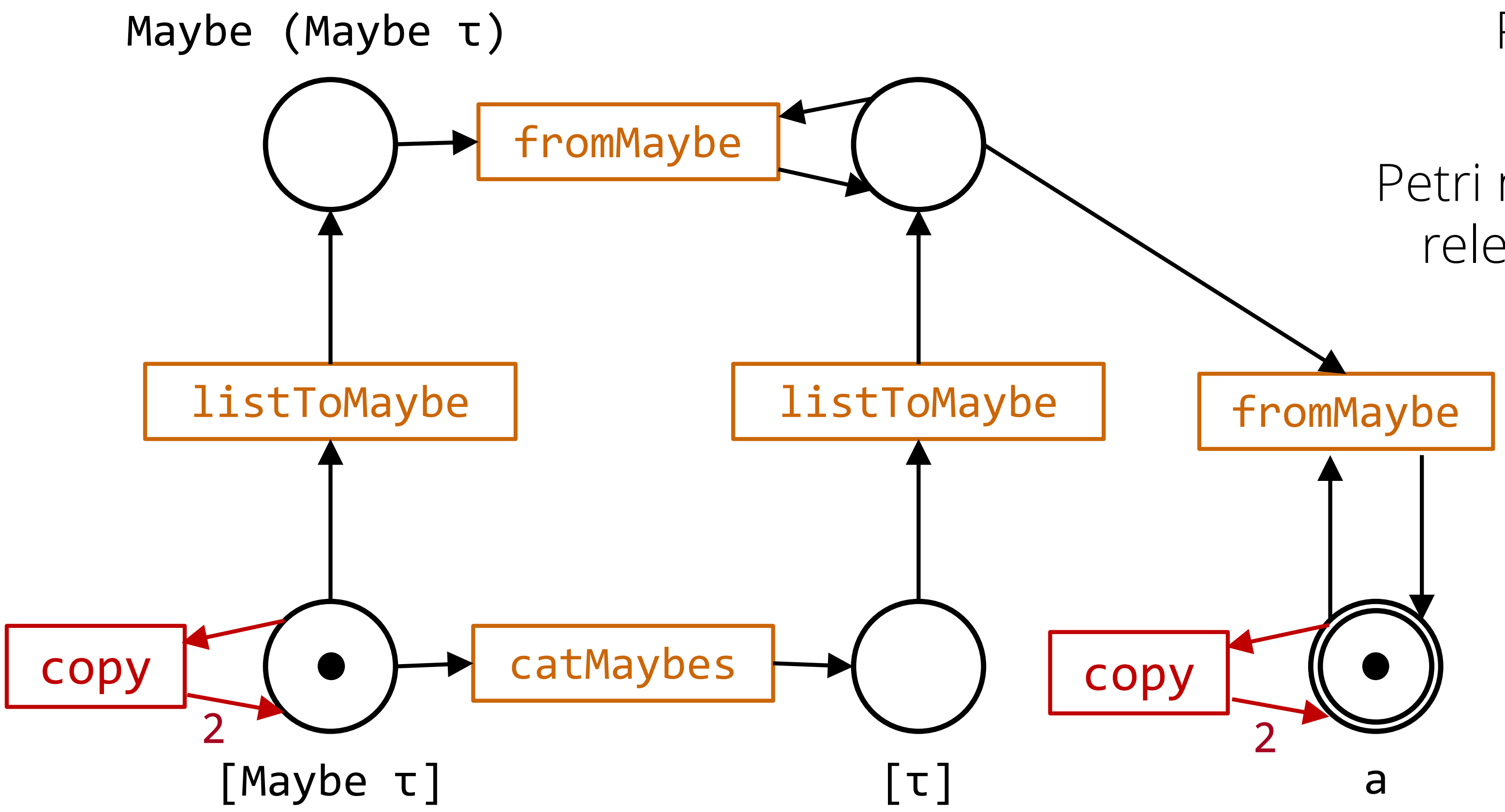
substructural types

program \ variable use	structural unrestricted	relevant *at least* once	linear *exactly* once
`\xs d → d`	✓	✗	✗
`\xs d → bool d d (null xs)`	✓	✓	✗
`\xs d → fromMaybe d (listToMaybe (catMaybes xs))`	✓	✓	✓

definitely useless

likely useless

search with substructural types



Petri net = linear types!

Petri net + copy transitions = relevant types + linear bias

this talk

1. synquid
recursive program synthesis
from refinement types

2. hoogle+
component-based synthesis
from Haskell types

3. future work
best types for synthesis?

simple types

expressive types

future work

specification

type

refinements?

examples?

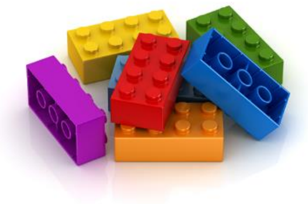
natural language?

→ H++ →

programs

more complex programs

more relevant results



more components

(advanced) linear types

can linear types help synthesize more complex programs?

example: concatenate a list of lists

```
concat :: xs:[[a]] → [a]
```

sufficiently complete if interpreted as a linear type!

non-linear use requires annotations

```
replicate :: n: Nat → x:n*a → [a]
```

partial refinement types

refinement types are great… until they aren't

example: are all elements of this list negative?

```
allNeg :: xs:[Int] → Bool
```

partial refinements:

```
allNeg :: xs:[Neg]                  → {ν:Bool|ν}
          xs:{ν:[Nat]|len ν > 0} → {ν:Bool|!ν}
```

type-driven program synthesis

1. synquid
recursive program synthesis
from refinement types

2. hoogle+
component-based synthesis
from Haskell types

3. future work
linear types
partial refinement types

simple types

expressive types