

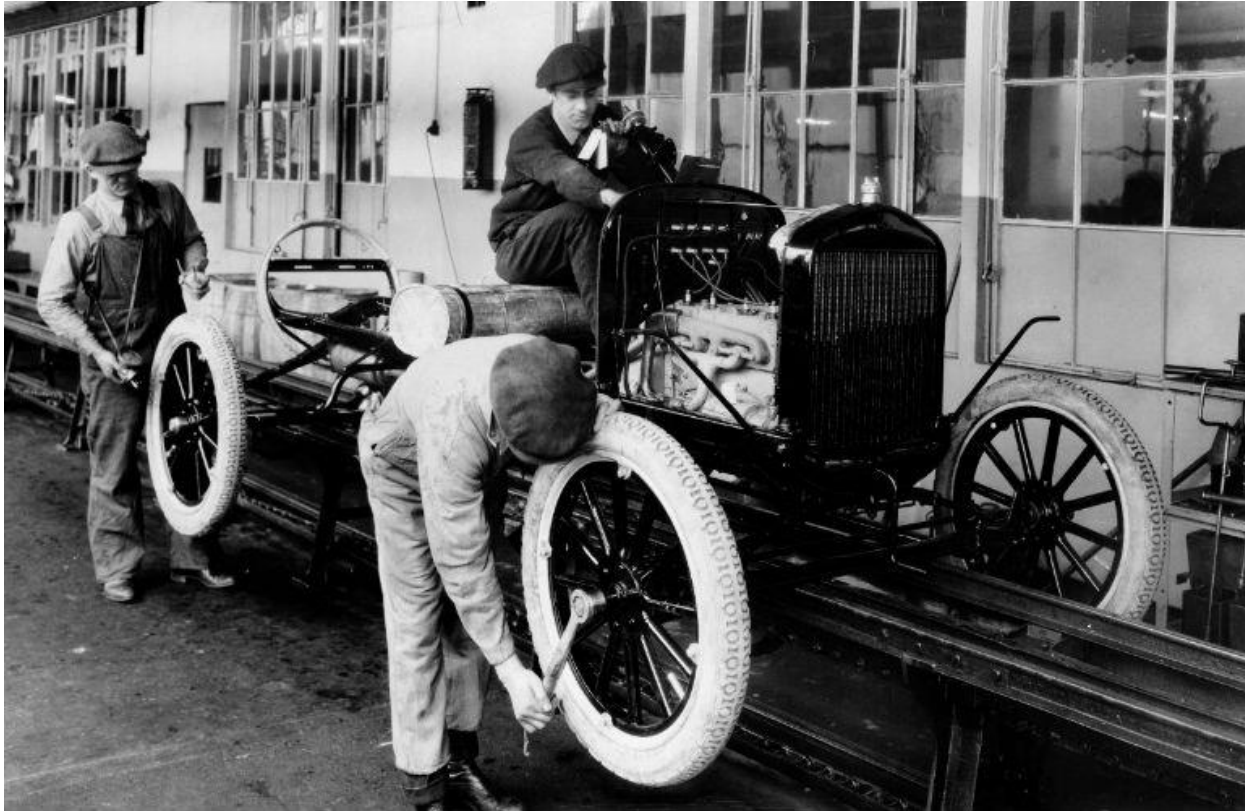
type-driven program synthesis

Nadia Polikarpova



UCSDCSE
Computer Science and Engineering

goal: automate programming



program synthesis

specification



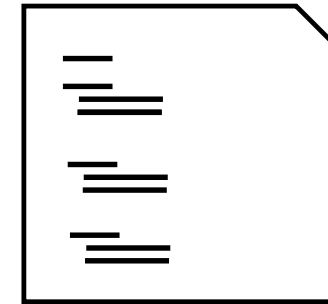
search



program
space



program



program synthesis

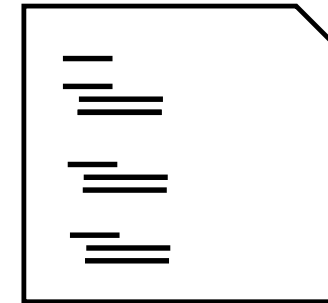
specification



search

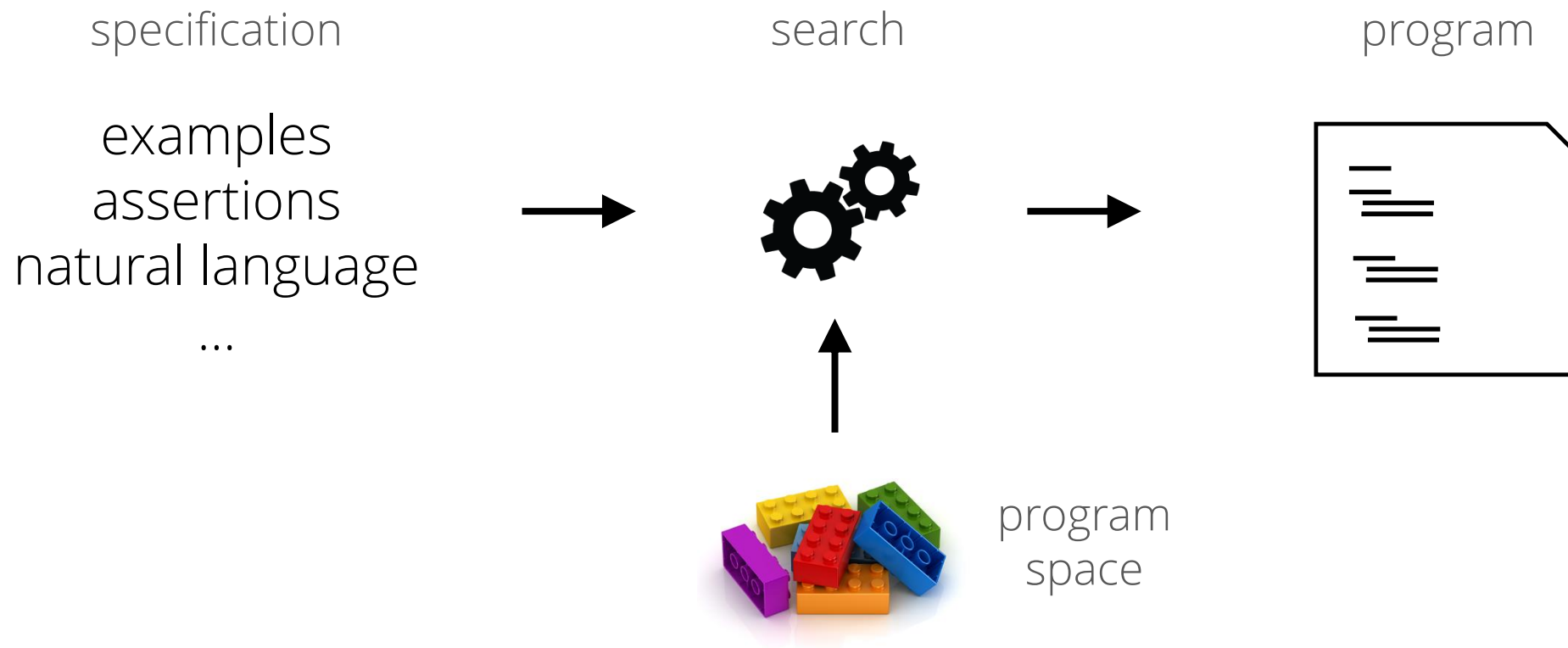


program



program
space

program synthesis



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

$a \rightarrow [a] \rightarrow [a]$

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?



this talk

1. synquid

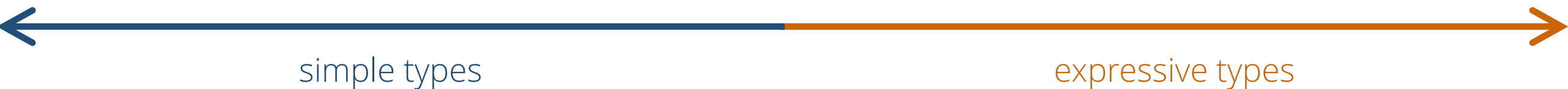
recursive program synthesis
from refinement types

2. hoogle+

component-based synthesis
from Haskell types

3. future work

best types for synthesis?



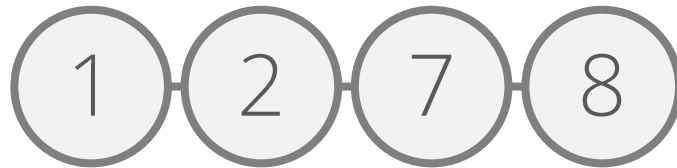
example: insert into a sorted list

input:

x



xS



output:

ys



program synthesis

specification

in: x , sorted xs
out: sorted ys
 $\text{elems } ys = \text{elems } xs \cup \{x\}$

formalize this



code

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

$[], :, \leq$

program synthesis

specification

= type

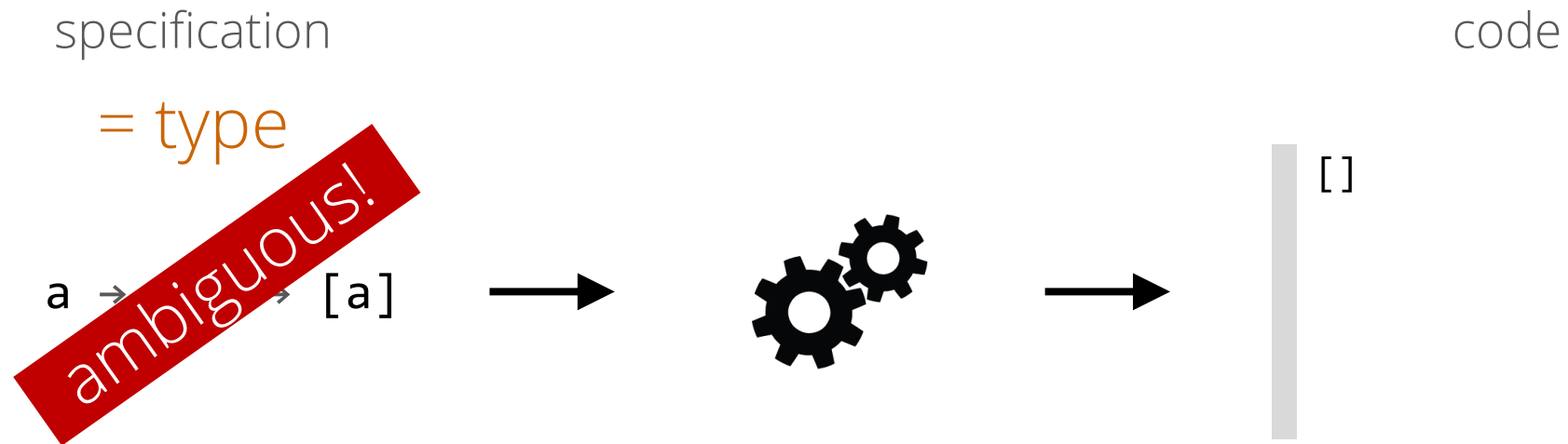
$a \rightarrow [a] \rightarrow [a]$



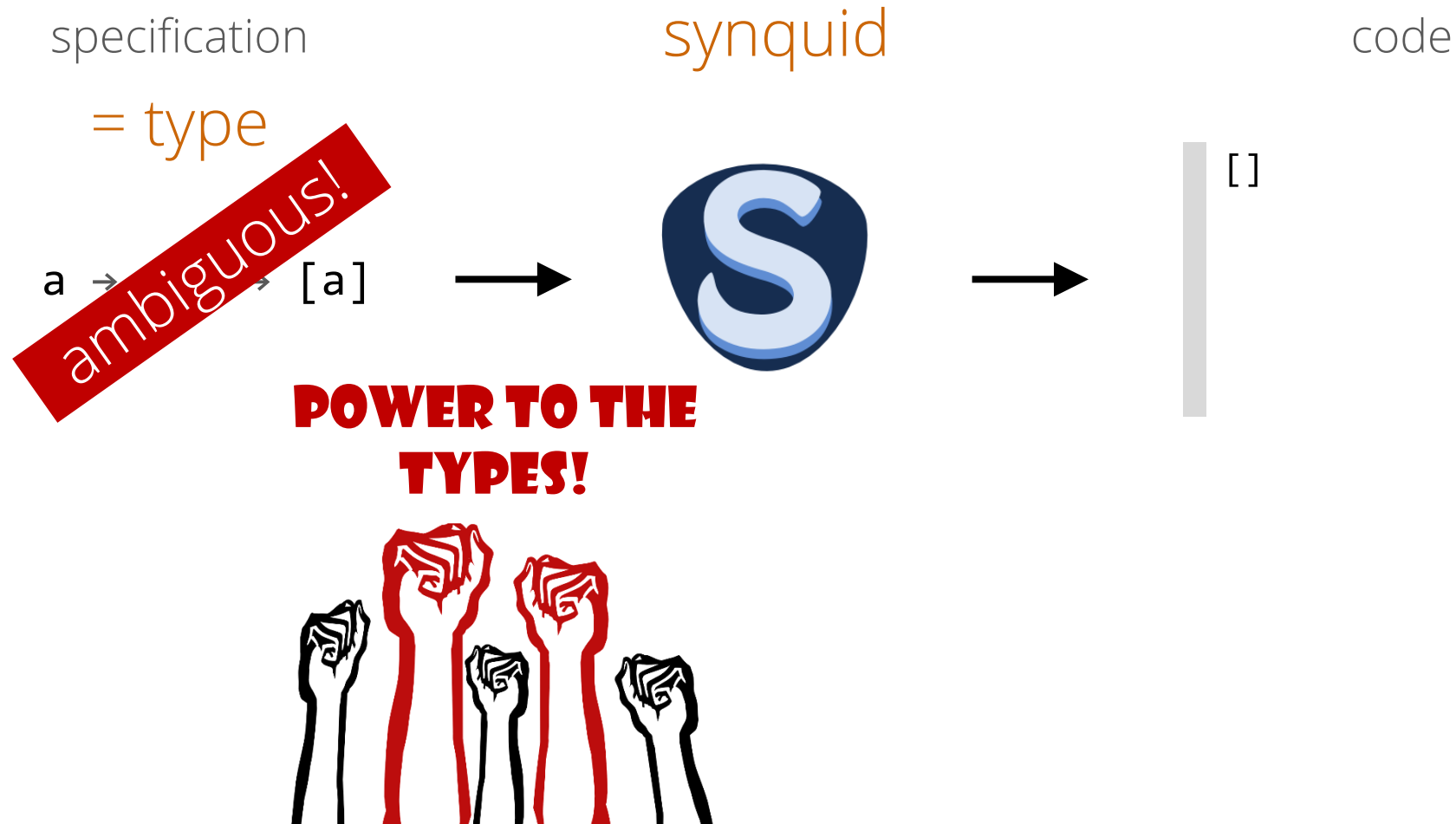
code

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

program synthesis



program synthesis



refinement types

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

insert in synquid

sorted list



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



set of elements

synquid

specification

```
insert :: x:a →  
xs:SList a →  
{v:SList a | elems v =  
  elems xs U {x}}
```



code

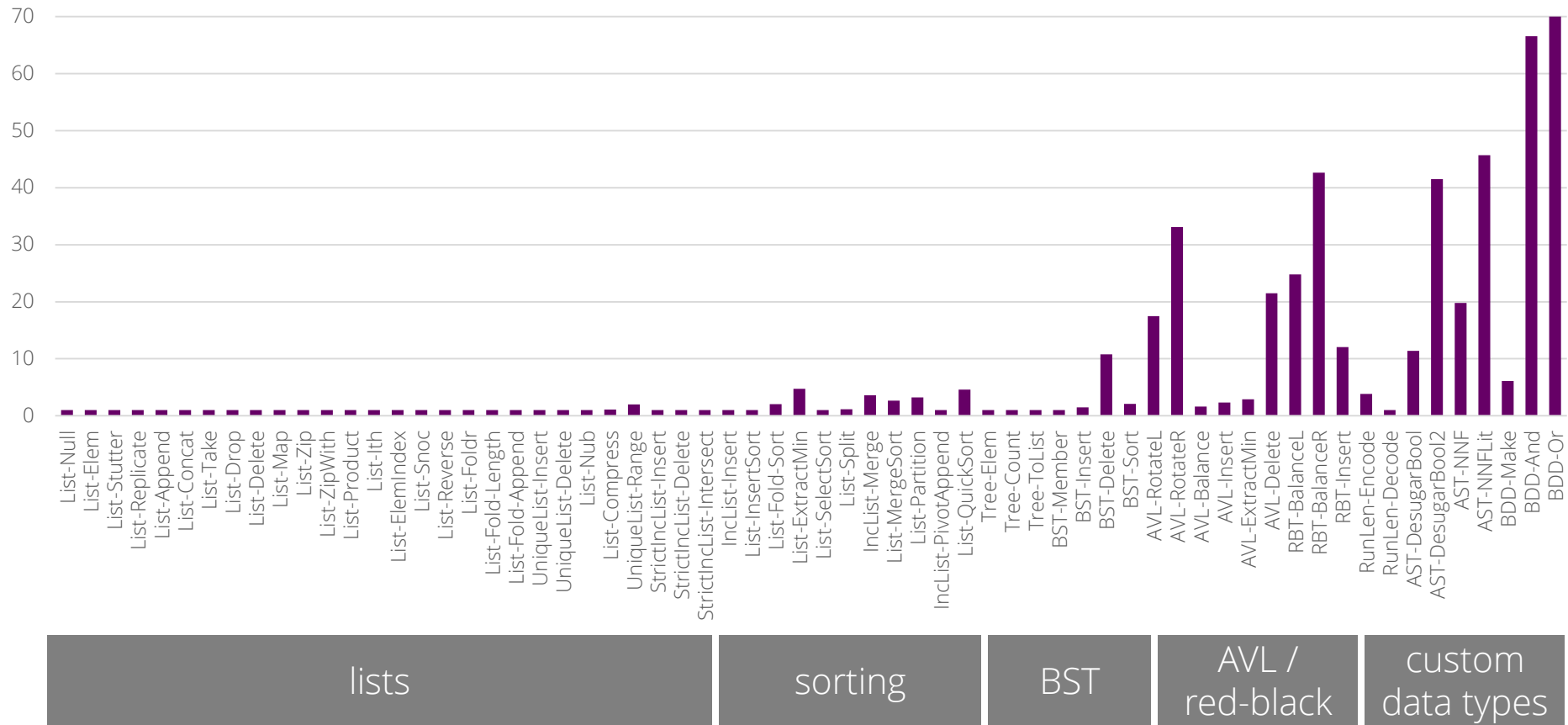
```
match xs with  
[] → x : []  
h : t →  
  if x ≤ h  
  then x : xs  
  else h : (insert x t)
```



`[]`, `:`, `≤`

what else can it do?

sec



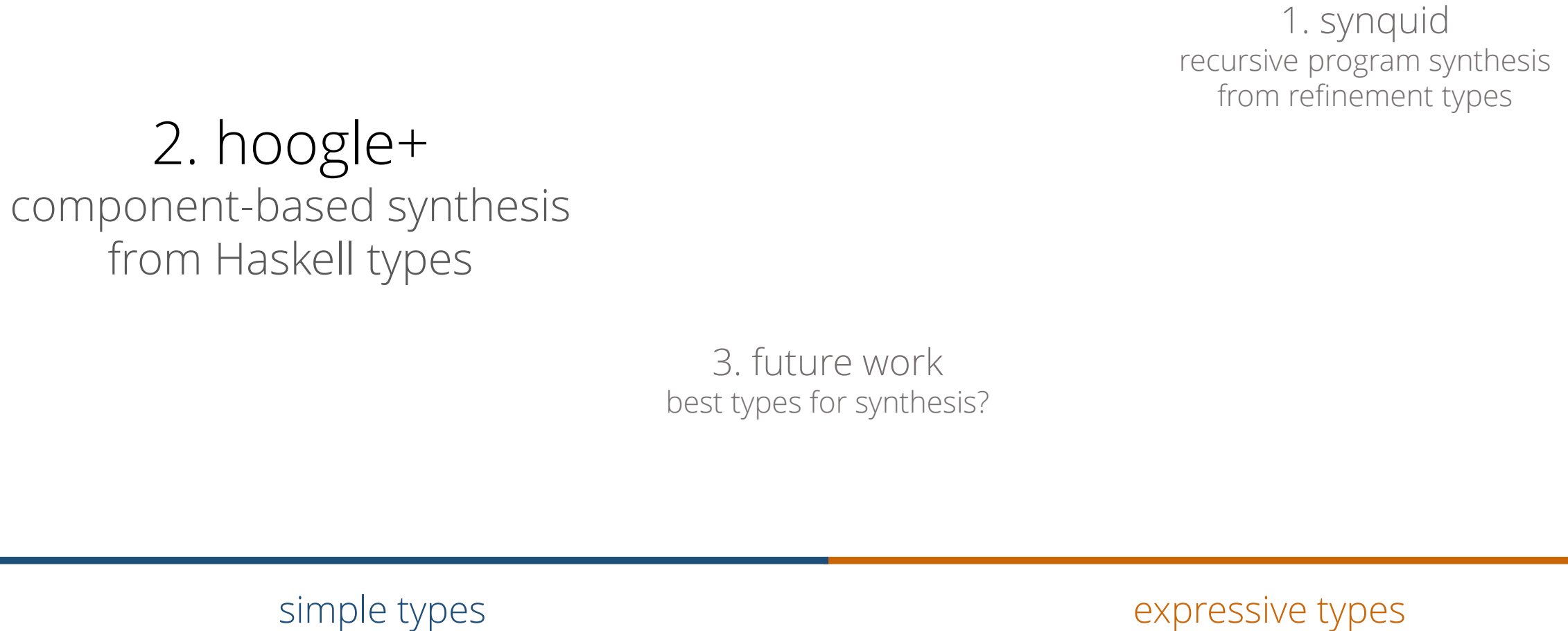
limitation

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

**POWER TO THE
TYPES???**



this talk



example: first non-empty

input:

xs



d



output:



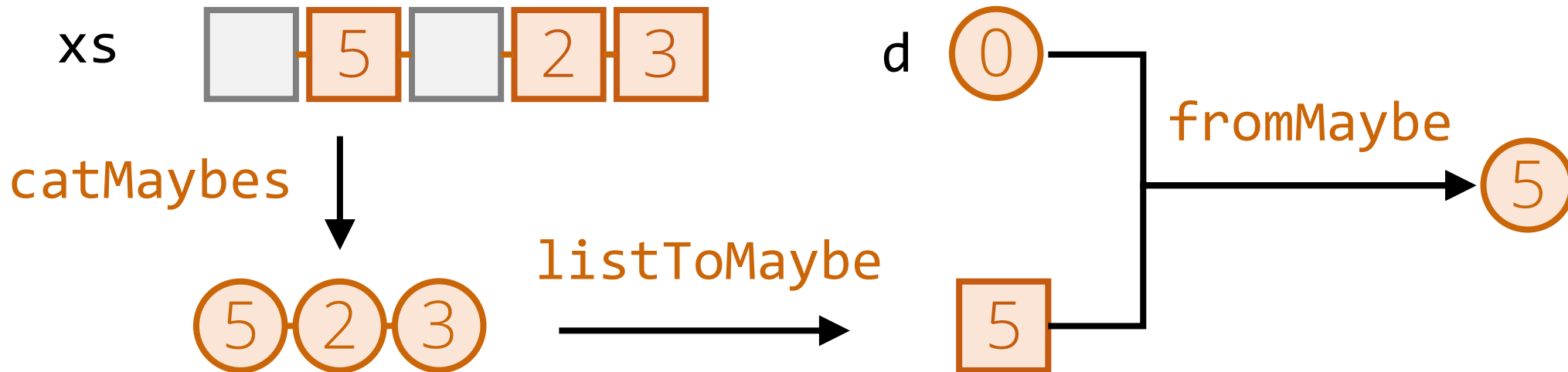
is there a **library function** for that?

ask [Hoogle!](#)

example: first non-empty

type query: $xs : [Maybe\ a] \rightarrow d : a \rightarrow a$

solution: $\backslash xs\ d \rightarrow \text{fromMaybe}\ d\ (\text{listToMaybe}\ (\text{catMaybes}\ xs))$



example: first non-empty

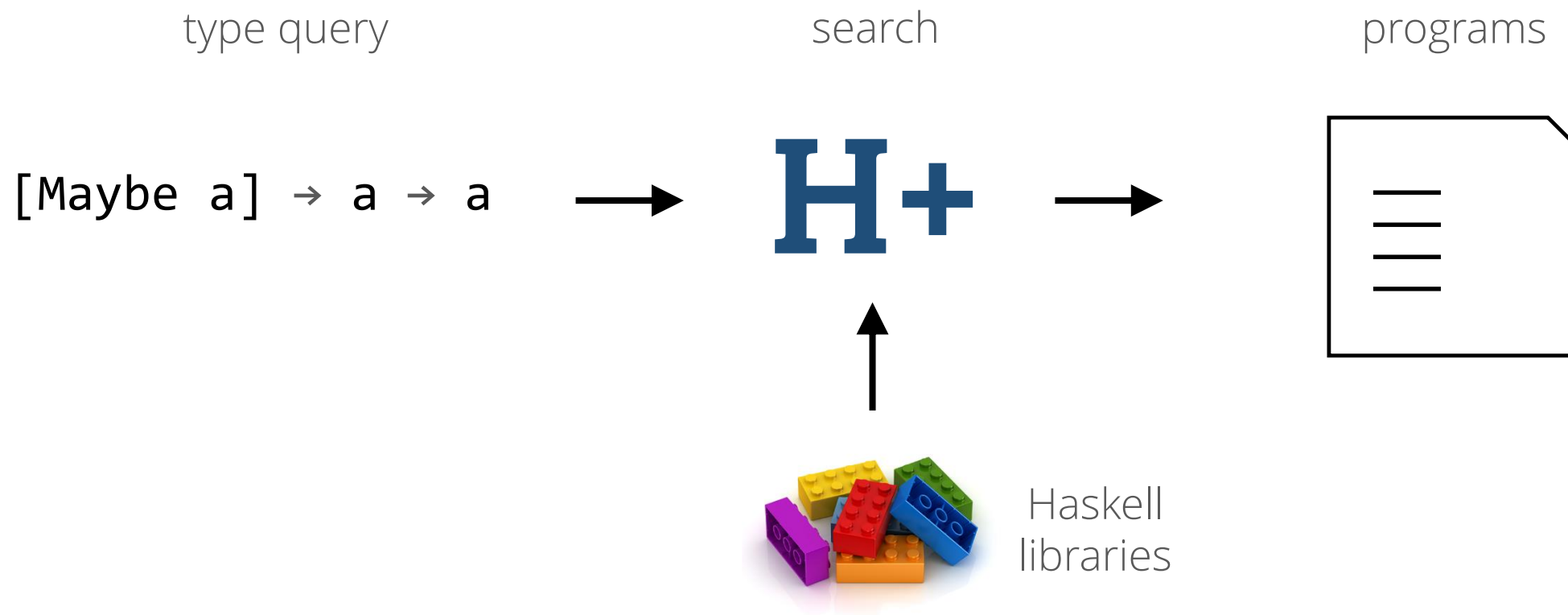
type query: `xs:[Maybe a] → d:a → a`

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

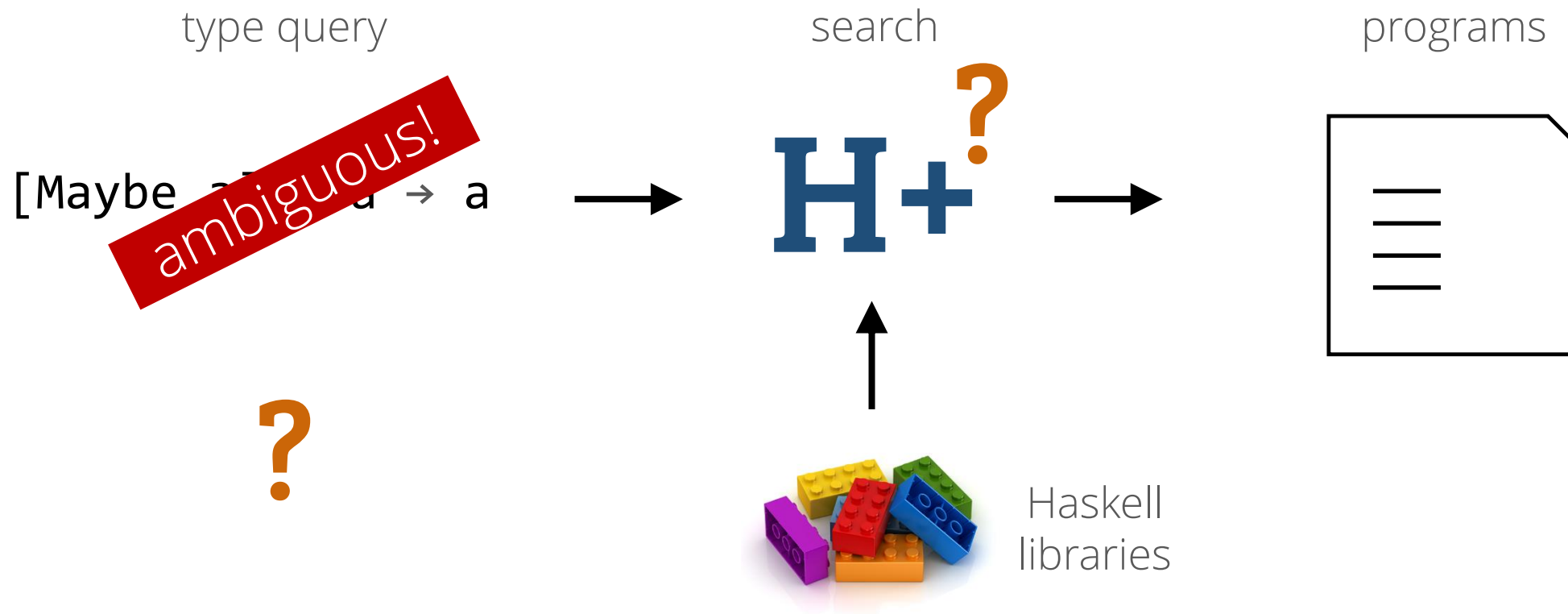
wanted: component-based synthesis

Hoogle+ to the rescue!

Hoogle+



Hoogle+

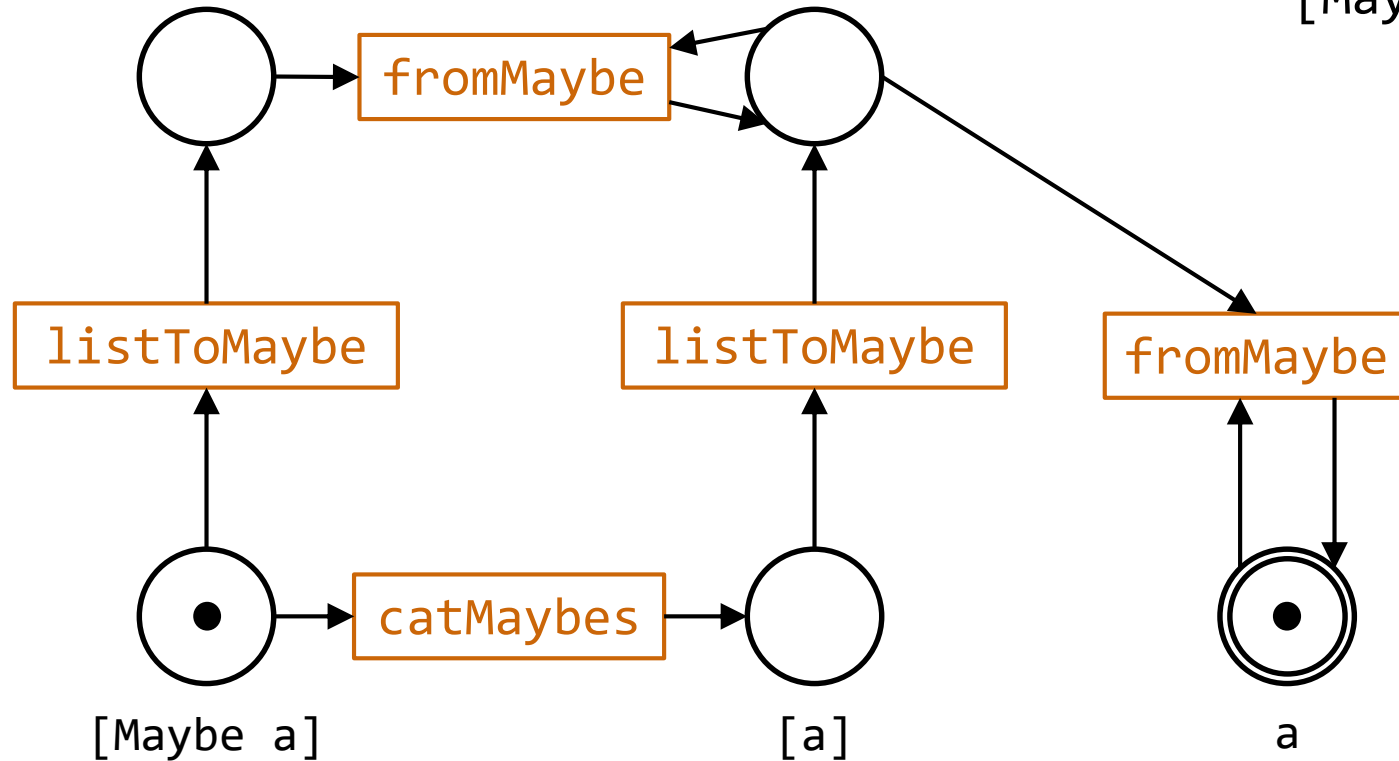


prior work: graph-based search

Petri net:

Maybe (Maybe a)

Maybe a



type query:

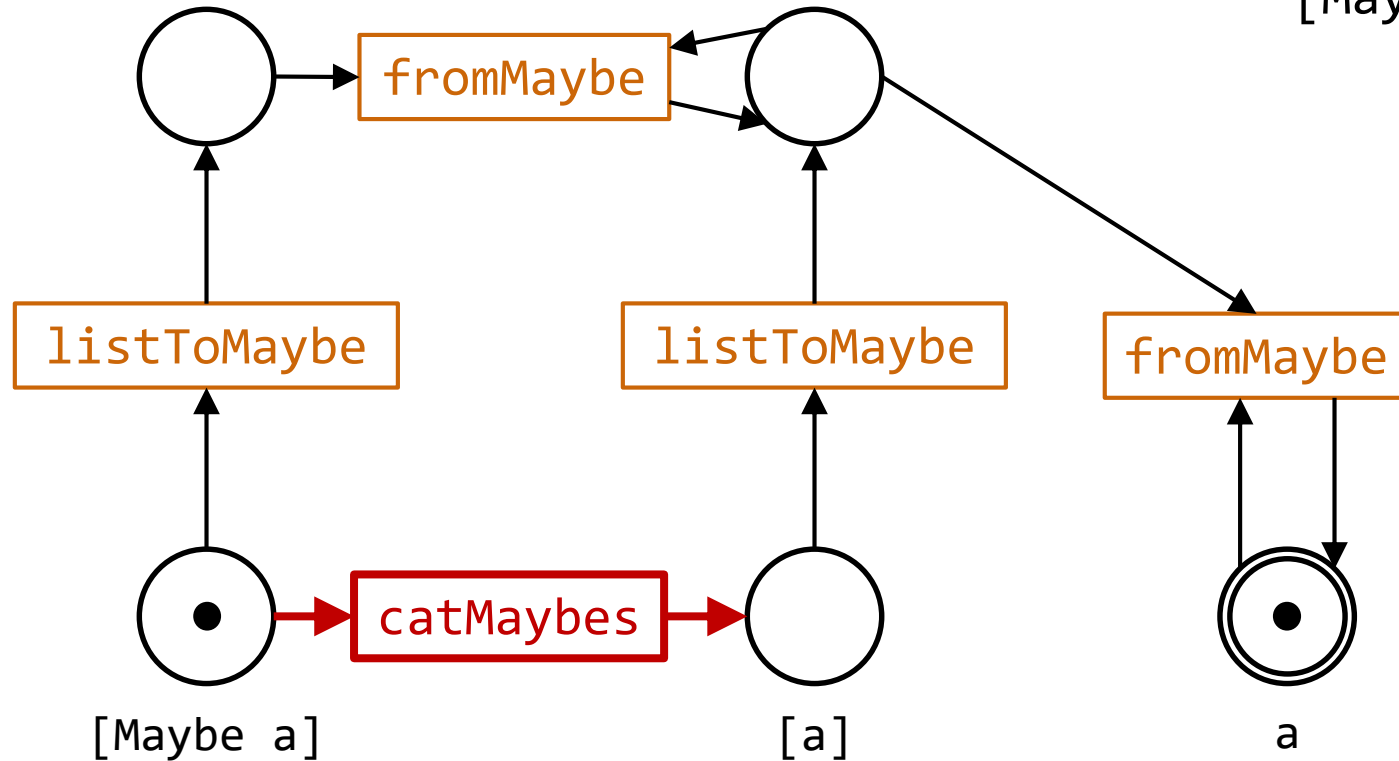
$[Maybe\ a] \rightarrow a \rightarrow a$

prior work: graph-based search

Petri net:

Maybe (Maybe a)

Maybe a



type query:

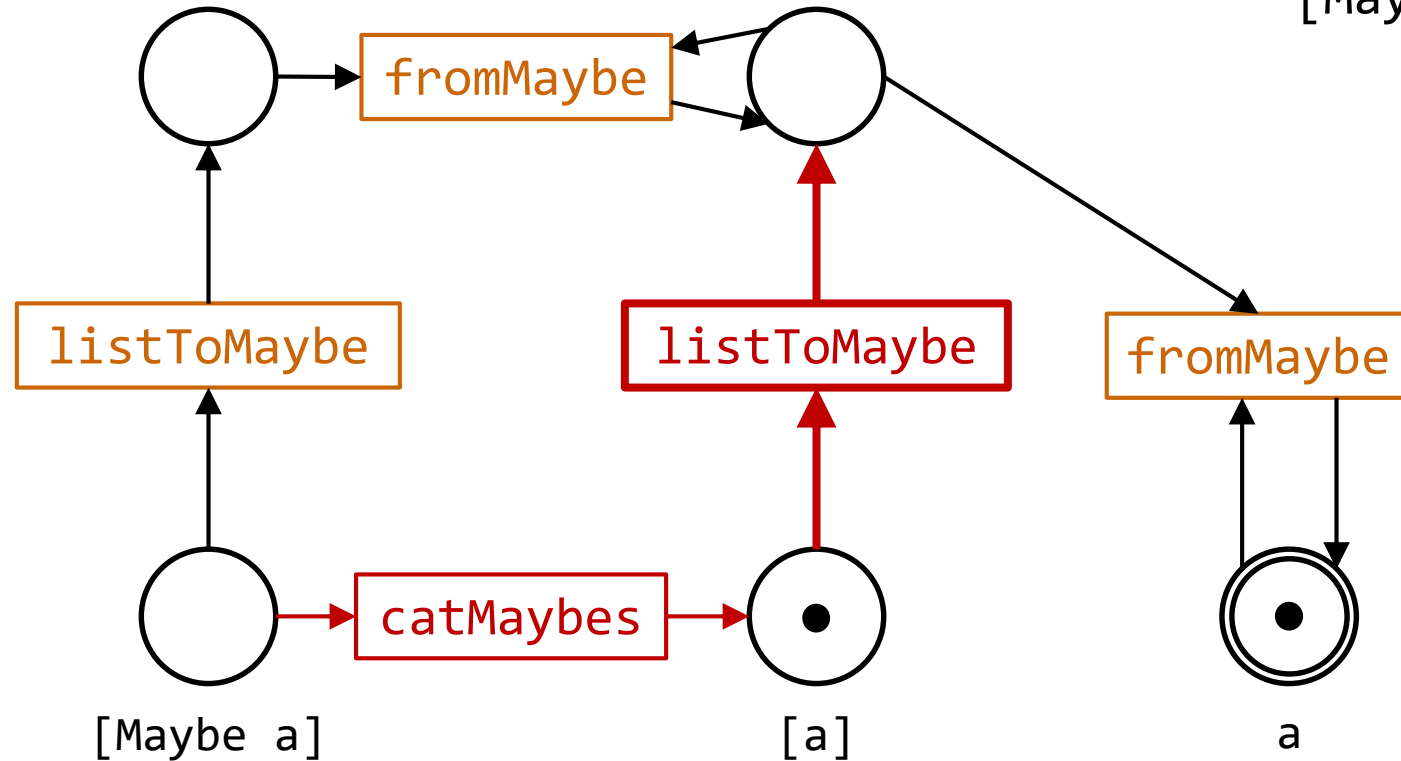
$[Maybe\ a] \rightarrow a \rightarrow a$

prior work: graph-based search

Petri net:

Maybe (Maybe a)

Maybe a



type query:

$[Maybe\ a] \rightarrow a \rightarrow a$

prior work: graph-based search

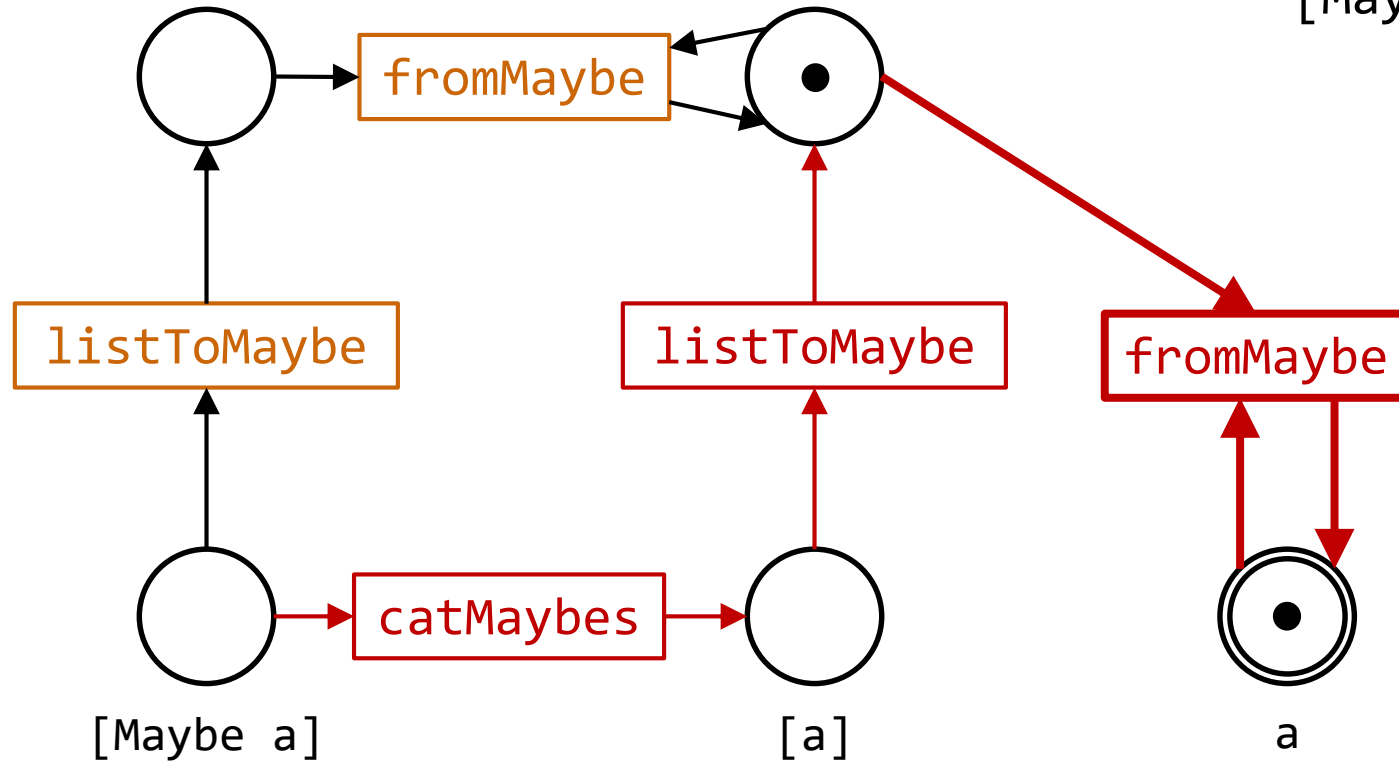
Petri net:

Maybe (Maybe a)

Maybe a

type query:

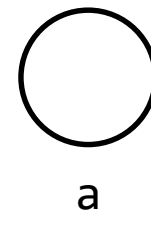
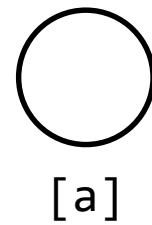
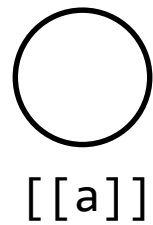
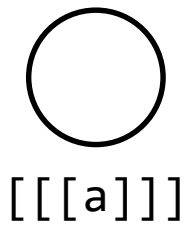
$[Maybe\ a] \rightarrow a \rightarrow a$



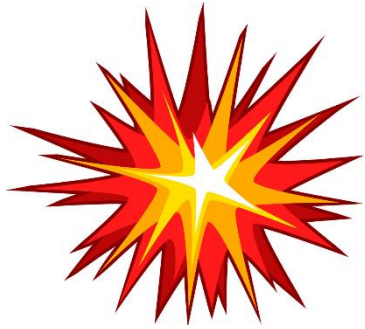
challenge: polymorphism

1. infinitely many types

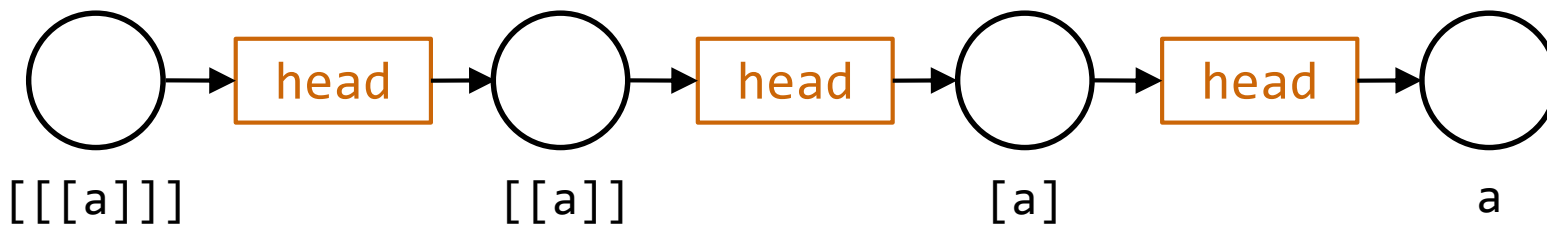
...



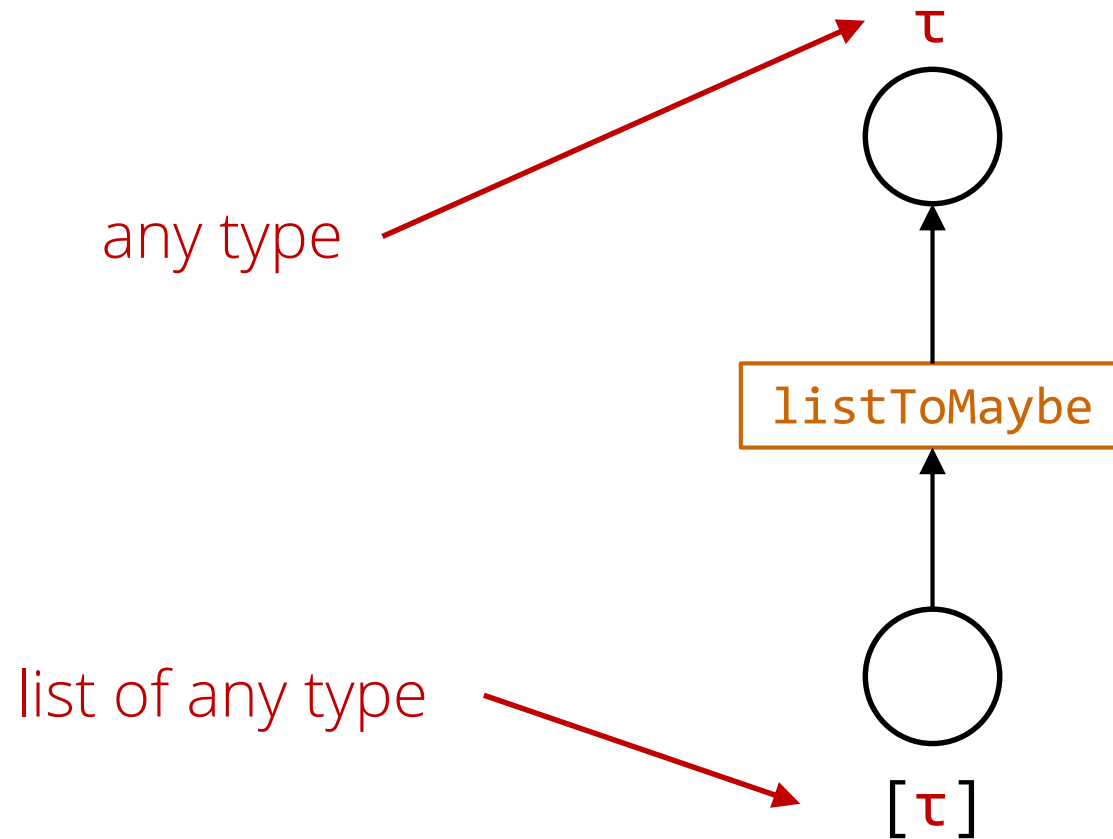
challenge: polymorphism



1. infinitely many types
2. transition explosion

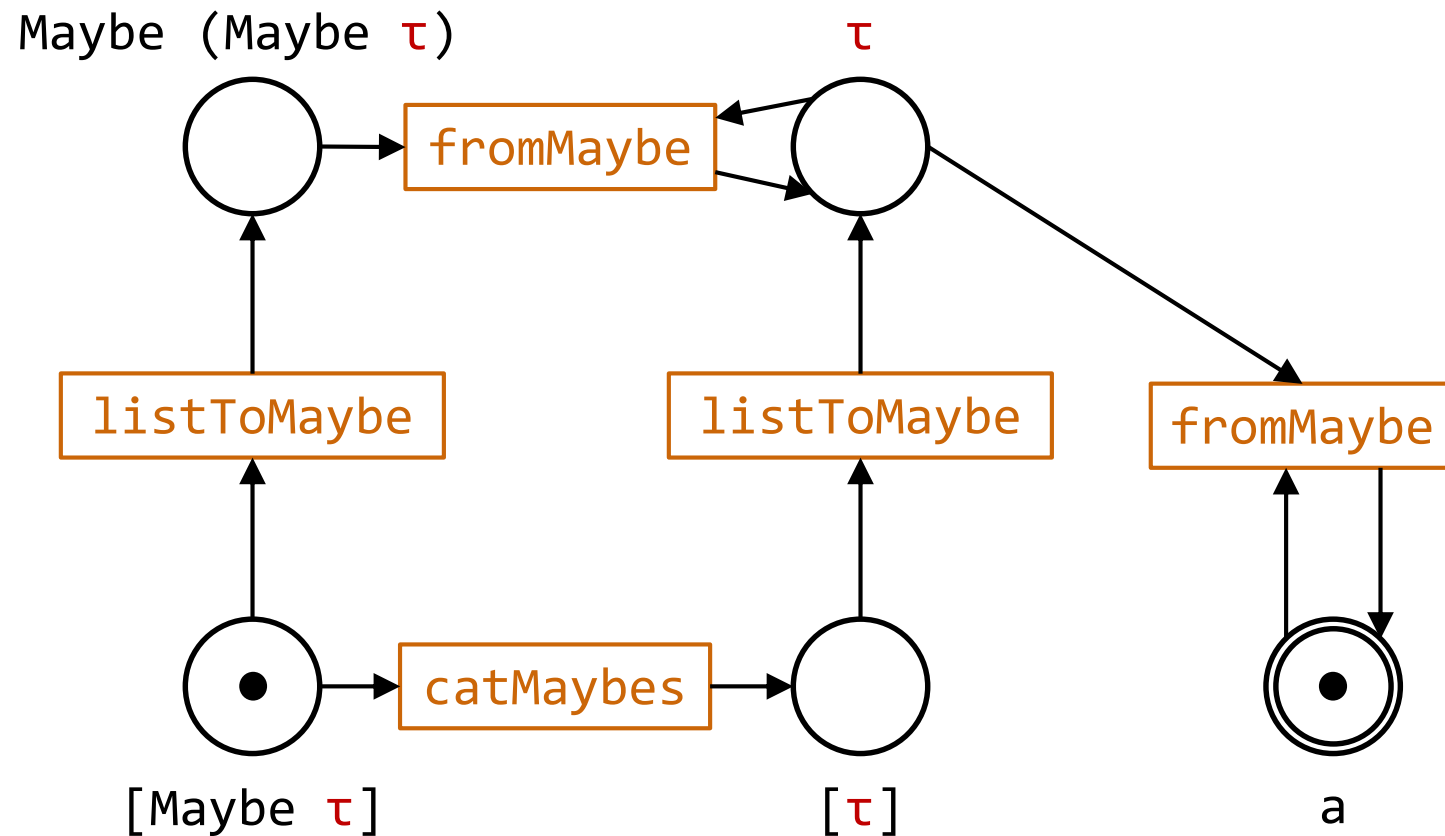


solution: abstraction

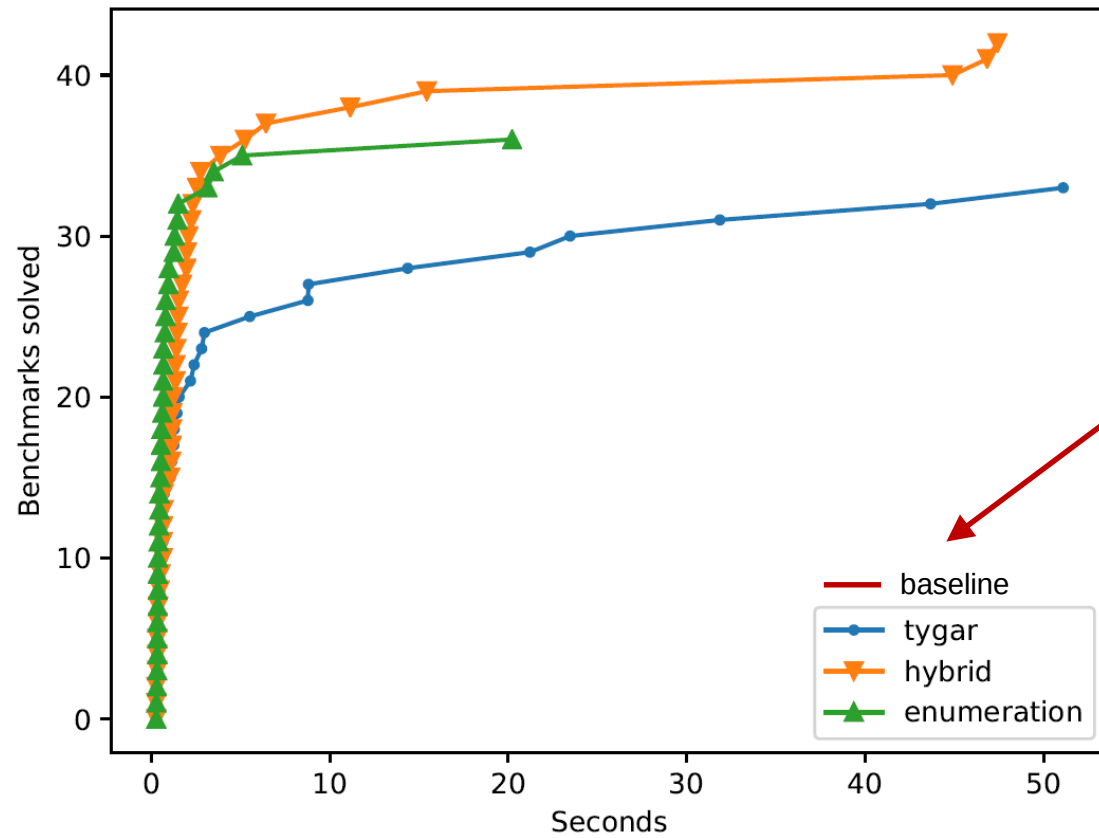


solution: abstraction

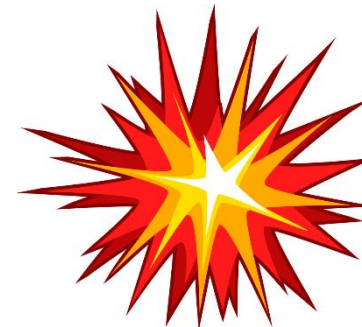
abstract
transition
net:



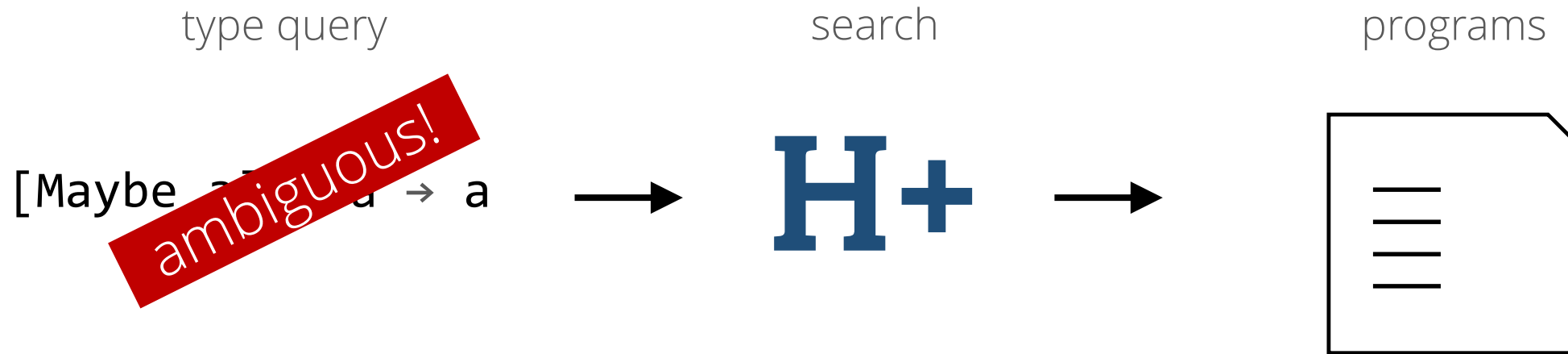
results



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`

2. always crash

polymorphism
makes this worse

`\xs d → foldr (head []) d xs`

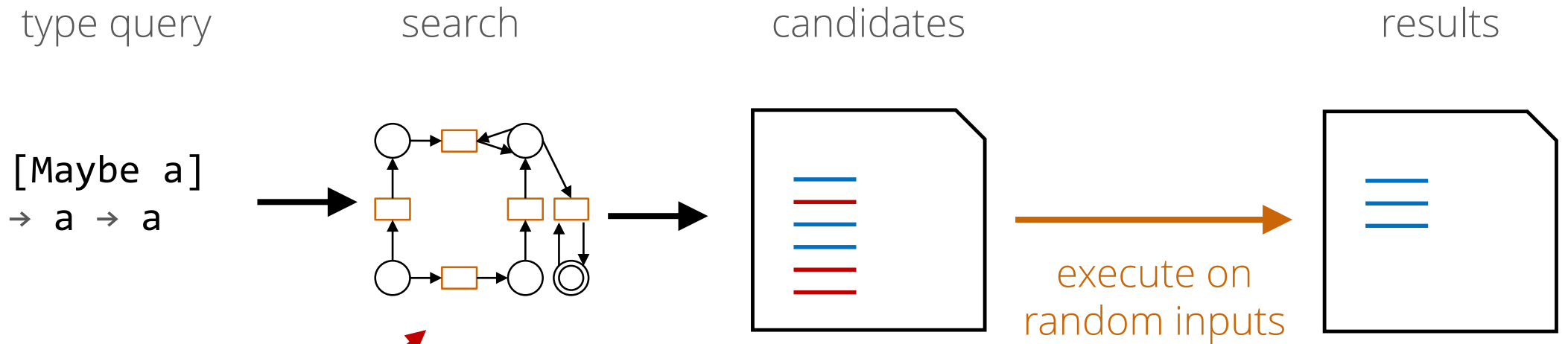


3. duplicates

`\xs d → fromMaybe d (head xs)`

`\xs d → maybe d id (head xs)`

test-based filtering

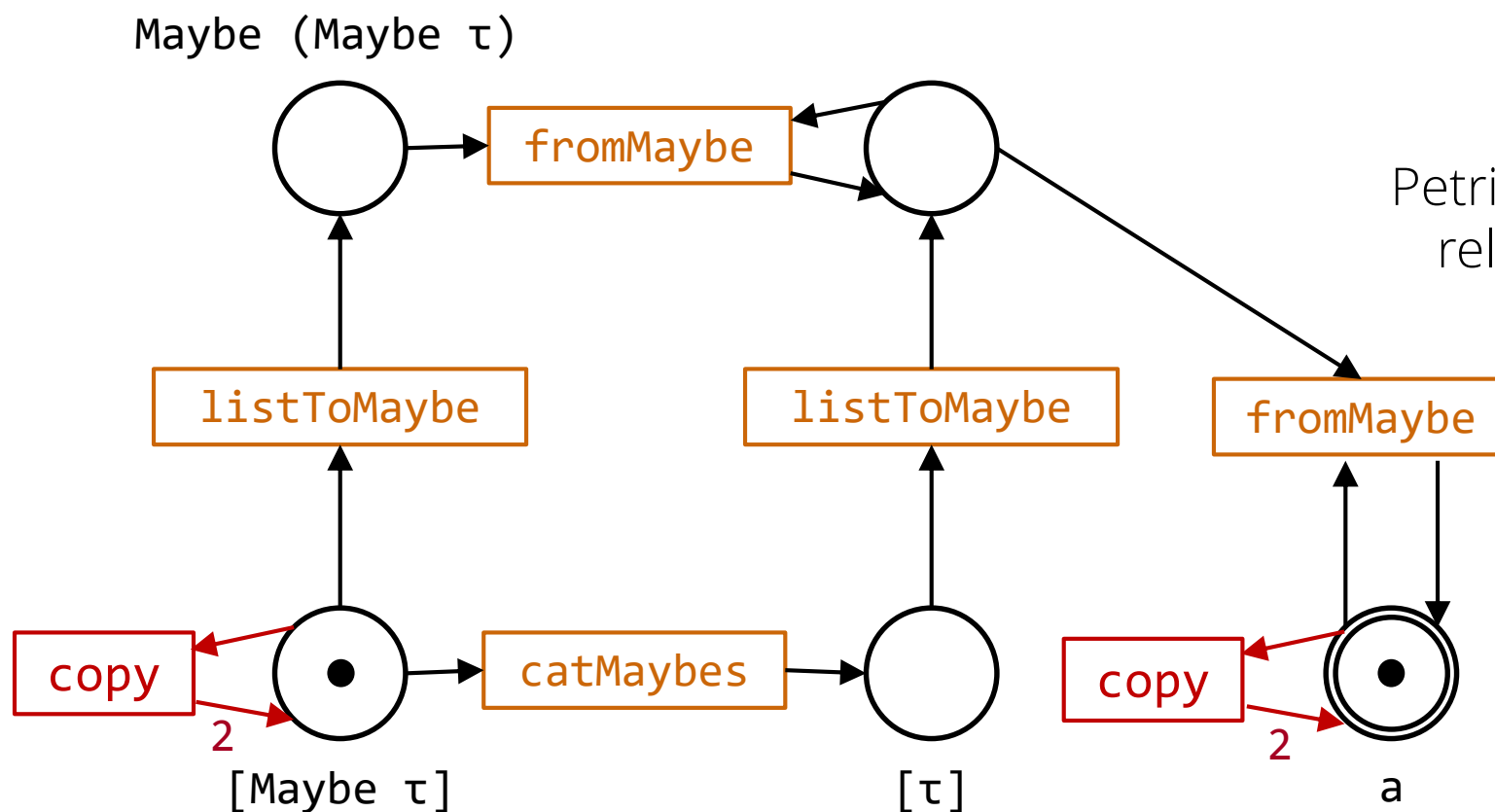


1. does it crash on all inputs?
2. is the output always the same as another candidate?
3. does the output stay the same when changing an input?

substructural types

program	variable use	structural unrestricted	relevant <i>at least once</i>	linear <i>exactly once</i>
<code>\xs d → d</code>				 ← definitely useless
<code>\xs d → bool d d (null xs)</code>				 ← likely useless
<code>\xs d → fromMaybe d (listToMaybe (catMaybes xs))</code>				

search with substructural types



Petri net = linear types!

Petri net + **copy transitions** =
relevant types + linear bias

this talk

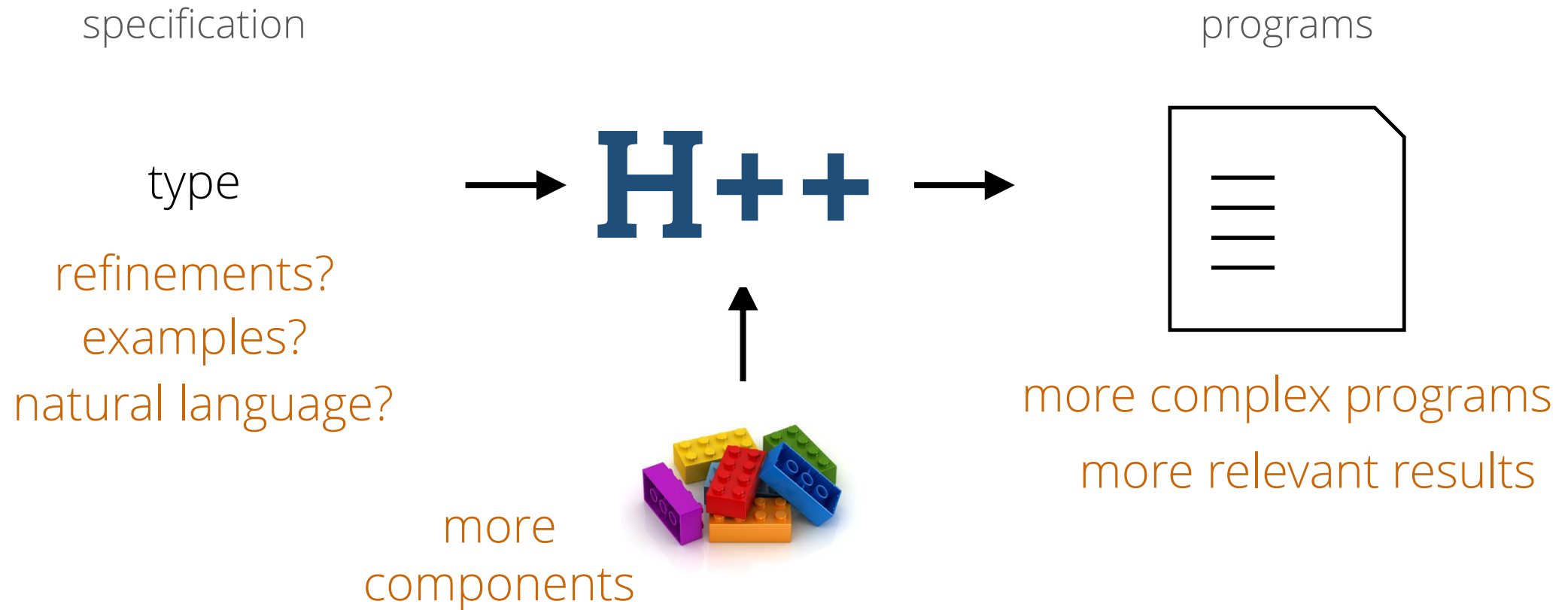
1. synquid
recursive program synthesis
from refinement types

2. hoogle+
component-based synthesis
from Haskell types

3. future work
best types for synthesis?



future work



(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] → {v:Bool | v}
xs:{v:[Nat] | len v > 0} → {v:Bool | !v}
```

type-driven program synthesis

