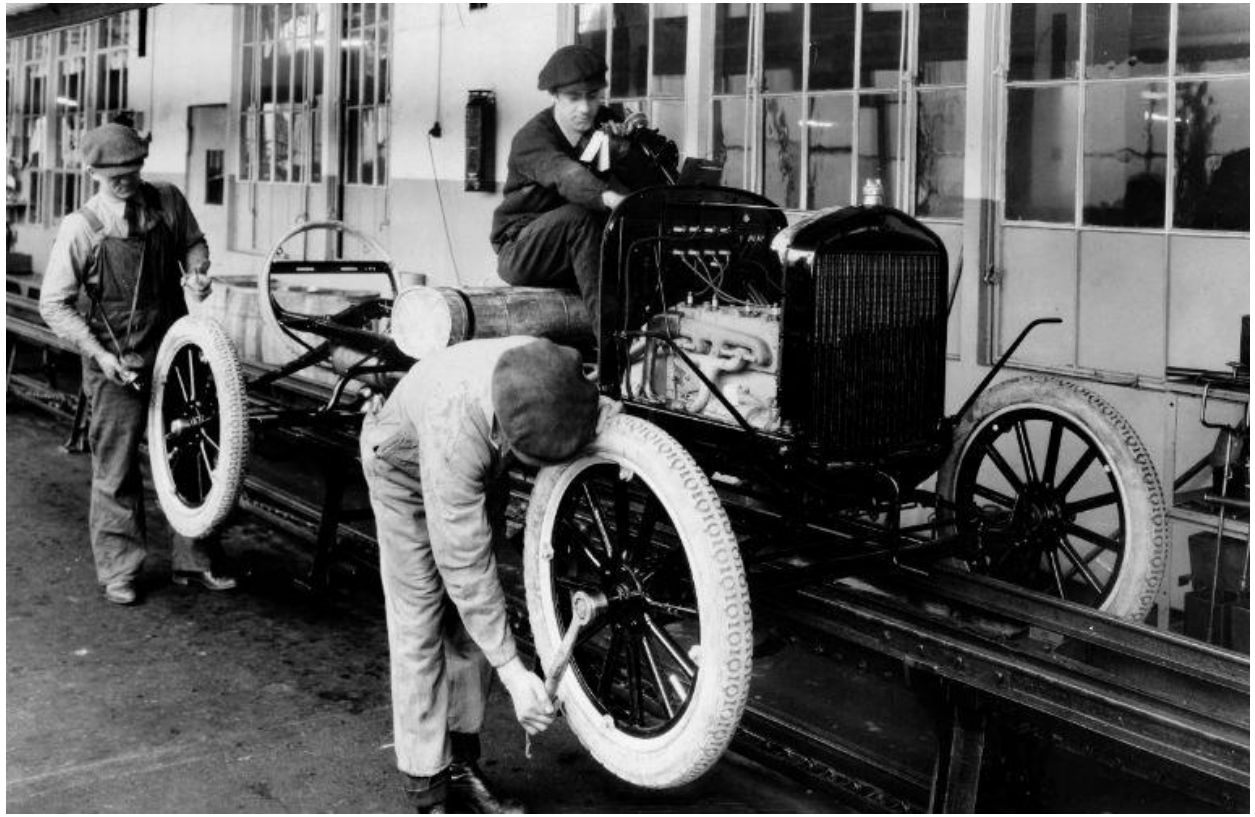
generating programs from types

Nadia Polikarpova

lambda
D A λ S

UC San Diego
COMPUTER SCIENCE & ENGINEERING

goal: automate programming



```
append:
    push ebp
    mov ebp, esp
    push eax
    push ebx
    push len
    call malloc
    mov ebx, [ebp + 12]
    mov [eax + info], ebx
    mov dword [eax + next], 0
    mov ebx, [ebp + 8]
    cmp dword [ebx], 0
    je null_pointer
    mov ebx, [ebx]

next_element:
    cmp dword [ebx + next], 0
    je found_last
    mov ebx, [ebx + next]
    jmp next_element

found_last:
    push eax
    push addMes
    call puts
    add esp, 4
    pop eax
    mov [ebx + next], eax

go_out:
    pop ebx
    pop eax
    mov esp, ebp
    pop ebp
    ret 8

null_pointer:
    push eax
    push nullMes
    call puts
    add esp, 4
    pop eax
    mov [ebx], eax
    jmp go_out
```

Assembly

```
append:
    push ebp
    mov ebp, esp
    push eax
    push ebx
    push len
    call malloc
    mov ebx, [ebp + 12]
    mov [eax + info], ebx
    mov dword [eax + next], 0
    mov ebx, [ebp + 8]
    cmp dword [ebx], 0
    je null_pointer
    mov ebx, [ebx]

next_element:
    cmp dword [ebx + next], 0
    je found_last
    mov ebx, [ebx + next]
    jmp next_element

found_last:
    push eax
    push addMes
    call puts
    add esp, 4
    pop eax
    mov [ebx + next], eax

go_out:
    pop ebx
    pop eax
    mov esp, ebp
    pop ebp
    ret 8

null_pointer:
    push eax
    push nullMes
    call puts
    add esp, 4
    pop eax
    mov [ebx], eax
    jmp go_out
```

Assembly

```c
void insert(node *xs, int x) {
  node *new;
  node *temp;
  node *prev;

  new = (node *)malloc(sizeof(node));
  if(new == NULL) {
    printf("Insufficient memory.");
    return;
  }
  new->val = x;
  new->next = NULL;
  if (xs == NULL) {
    xs = new;
  } else if(x < xs->val) {
    new->next = xs;
    xs = new;
  } else {
    prev = xs;
    temp = xs->next;
    while(temp != NULL && x > temp->val) {
      prev = temp;
      temp = temp->next;
    }
    if(temp == NULL) {
      prev->next = new;
    } else {
      new->next = temp;
      prev->next = new;
    }
  }
}
```

C

```
append:
    push ebp
    mov ebp, esp
    push eax
    push ebx
    push len
    call malloc
    mov ebx, [ebp + 12]
    mov [eax + info], ebx
    mov dword [eax + next], 0
    mov ebx, [ebp + 8]
    cmp dword [ebx], 0
    je null_pointer
    mov ebx, [ebx]

next_element:
    cmp dword [ebx + next], 0
    je found_last
    mov ebx, [ebx + next]
    jmp next_element

found_last:
    push eax
    push addMes
    call puts
    add esp, 4
    pop eax
    mov [ebx + next], eax

go_out:
    pop ebx
    pop eax
    mov esp, ebp
    pop ebp
    ret 8

null_pointer:
    push eax
    push nullMes
    call puts
    add esp, 4
    pop eax
    mov [ebx], eax
    jmp go_out
```

Assembly

```c
void insert(node *xs, int x) {
  node *new;
  node *temp;
  node *prev;

  new = (node *)malloc(sizeof(node));
  if(new == NULL) {
    printf("Insufficient memory.");
    return;
  }
  new->val = x;
  new->next = NULL;
  if (xs == NULL) {
    xs = new;
  } else if(x < xs->val) {
    new->next = xs;
    xs = new;
  } else {
    prev = xs;
    temp = xs->next;
    while(temp != NULL && x > temp->val) {
      prev = temp;
      temp = temp->next;
    }
    if(temp == NULL) {
      prev->next = new;
    } else {
      new->next = temp;
      prev->next = new;
    }
  }
}
```

C

```haskell
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)
```

Haskell

what's next?

```
append:
    push ebp
    mov ebp, esp
    push eax
    push ebx
    push len
    call malloc
    mov ebx, [ebp + 12]
    mov [eax + info], ebx
    mov dword [eax + next], 0
    mov ebx, [ebp + 8]
    cmp dword [ebx], 0
    je null_pointer
    mov ebx, [ebx]

next_element:
    cmp dword [ebx + next], 0
    je found_last
    mov ebx, [ebx + next]
    jmp next_element

found_last:
    push eax
    push addMes
    call puts
    add esp, 4
    pop eax
    mov [ebx + next], eax

go_out:
    pop ebx
    pop eax
    mov esp, ebp
    pop ebp
    ret 8

null_pointer:
    push eax
    push nullMes
    call puts
    add esp, 4
    pop eax
    mov [ebx], eax
    jmp go_out
```

```
void insert(node *xs, int x) {
  node *new;
  node *temp;
  node *prev;

  new = (node *)malloc(sizeof(node));
  if(new == NULL) {
    printf("Insufficient memory.");
    return;
  }
  new->val = x;
  new->next = NULL;
  if (xs == NULL) {
    xs = new;
  } else if(x < xs->val) {
    new->next = xs;
    xs = new;
  } else {
    prev = xs;
    temp = xs->next;
    while(temp != NULL && x > temp->val) {
      prev = temp;
      temp = temp->next;
    }
    if(temp == NULL) {
      prev->next = new;
    } else {
      new->next = temp;
      prev->next = new;
    }
  }
}
```

```
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)
```

?

Assembly C Haskell future

149

How to split a string in Haskell?

How do I split a string on a custom separator? I want the following behavior:

```
split ‘,’ “my,comma,separated,list”  → [“my”, “comma”, “separated”, “list”]
```

stackoverflow

149

How to split a string in Haskell?

How do I split a string on a custom separator? I want the following behavior:

```
split ‘,’ “my,comma,separated,list”  → [“my”, “comma”, “separated”, “list”]
```

157

You can implement it like this:

```
split :: Char -> String -> [String]
split c s = case dropWhile (== c) s of
                 "" -> []
                 s' -> w : split c s’’
  where (w, s'') = break (== c) s'
```

stackoverflow

149

How to split a string in Haskell?

How do I split a string on a custom separator? I want the follo… behavior:

```
split ‘,’ “my,comma,separated,list”  → […ed”, “list”]
```

wanted: automatic programmer's assistant

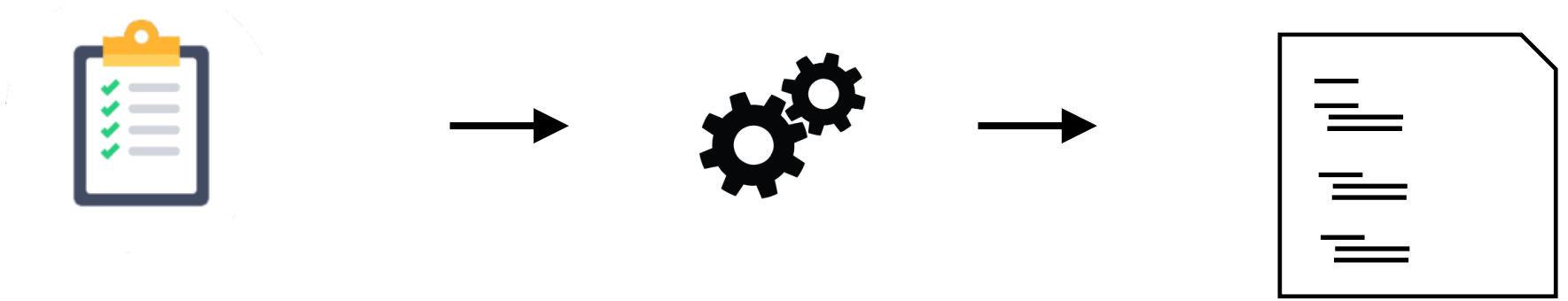
157

You can implem…

```
split :: Char -> String -> [String]
split c s = case dropWhile (== c) s of
                 "" -> []
                 s' -> w : split c s’’
  where (w, s'') = break (== c) s'
```

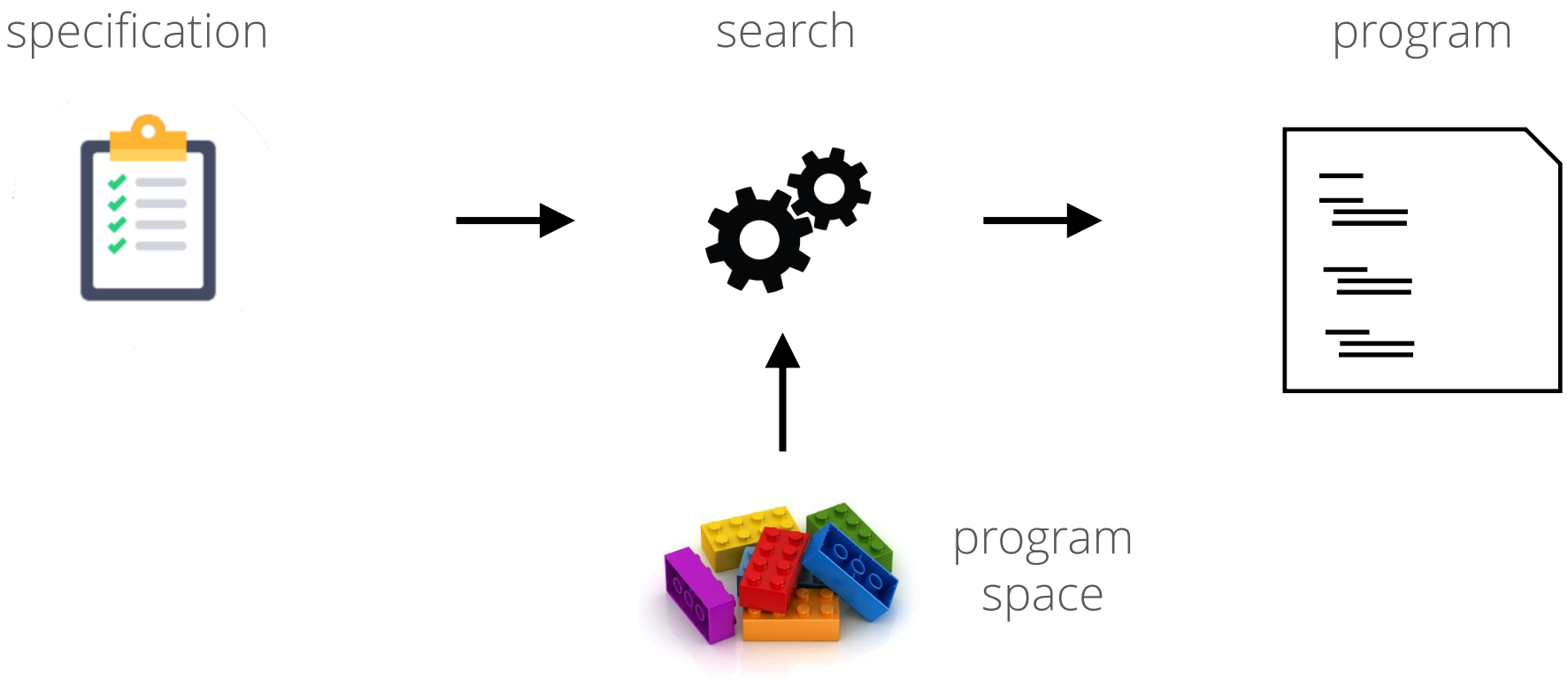
program synthesis

specification

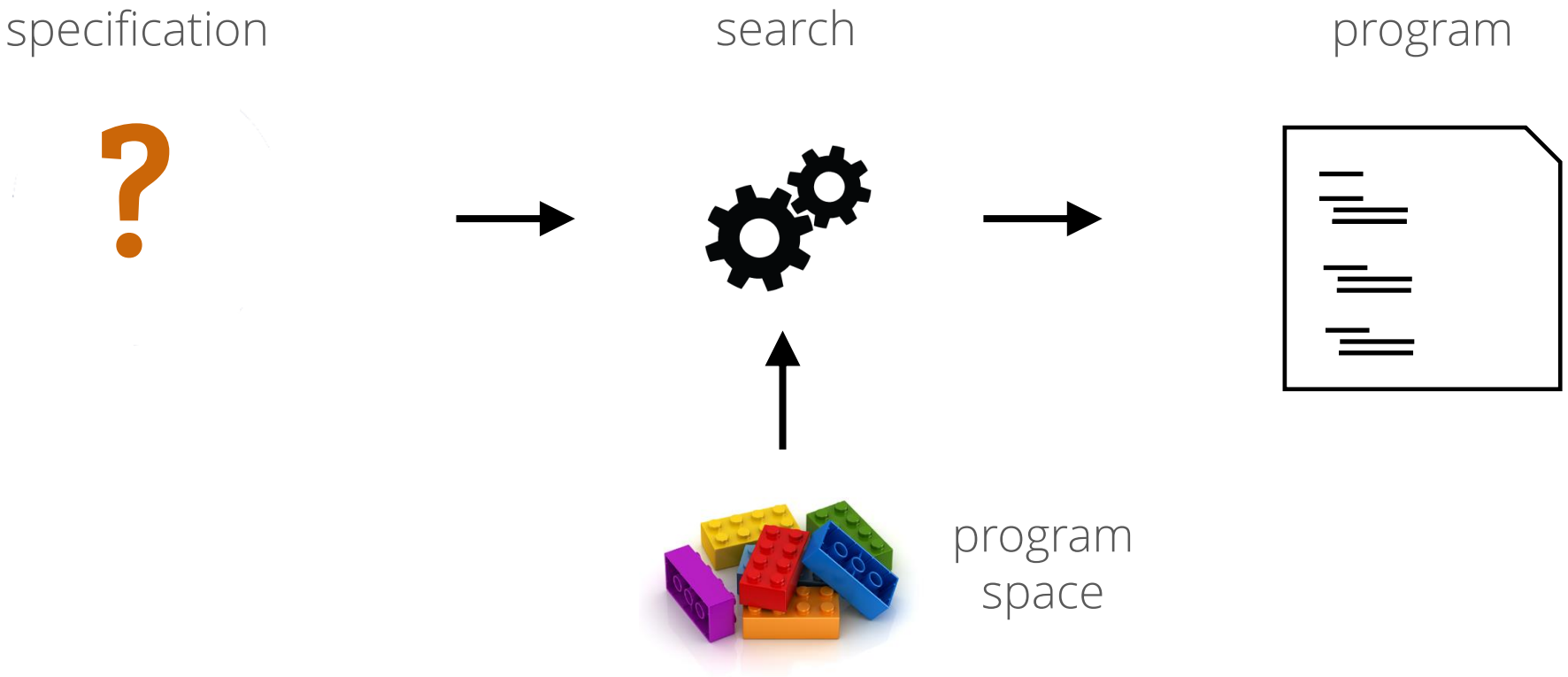


program

program synthesis



program synthesis



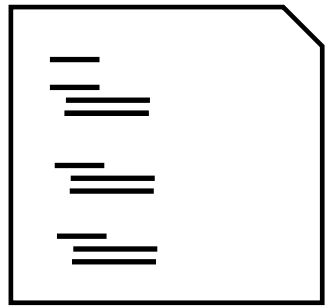
program synthesis

specification

examples
assertions
natural language
...

search

program



program space

program synthesis

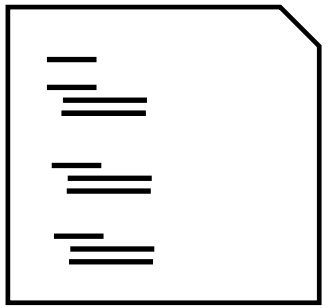
specification

examples
assertions
natural language
types!

search

program
space

program



the future is (almost) here

the future is (almost) here

Hoogle

Char -> String -> [String] Search

```
split :: Char -> String -> [String]
```

ghc Util

the future is (almost) here

Hoogle

Char -> String -> [String]

Search

```
split :: Char -> String -> [String]
```

ghc Util

what about generating new programs?

simple types

expressive types

more programmer-friendly
more ambiguous

simple types

expressive types

more programmer-friendly
more ambiguous

less programmer-friendly
less ambiguous

simple types

expressive types

simple types

expressive types

this talk

part I: synquid

refinement types → recursive programs

simple types

expressive types

this talk

part II: hoogle+

H+

Haskell types → function compositions

part I: synquid

refinement types → recursive programs

simple types

expressive types

part I

synquid

refinement types → recursive programs

part I

synquid

refinement types → recursive programs

Polikarpova, Kuraj, Solar-Lezama: *Program Synthesis from Polymorphic Refinement Types.* [PLDI'16]

part I

synquid

refinement types → recursive programs

1. types as specifications
2. type-directed search

Polikarpova, Kuraj, Solar-Lezama: *Program Synthesis from Polymorphic Refinement Types.* [PLDI'16]

part I

synquid

refinement types → recursive programs

1. types as specifications

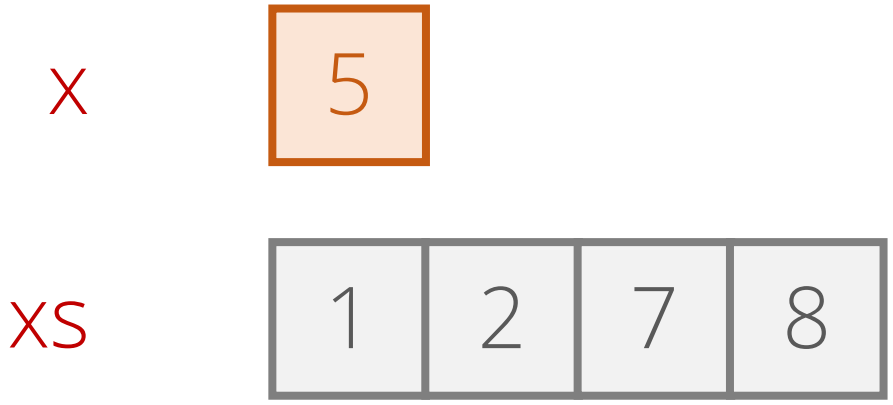
example: insert into a sorted list

example: insert into a sorted list

Input:

example: insert into a sorted list

Input:



example: insert into a sorted list

Input:

x

xs

Output:

ys

insert in a functional language

```
insert x xs =
```

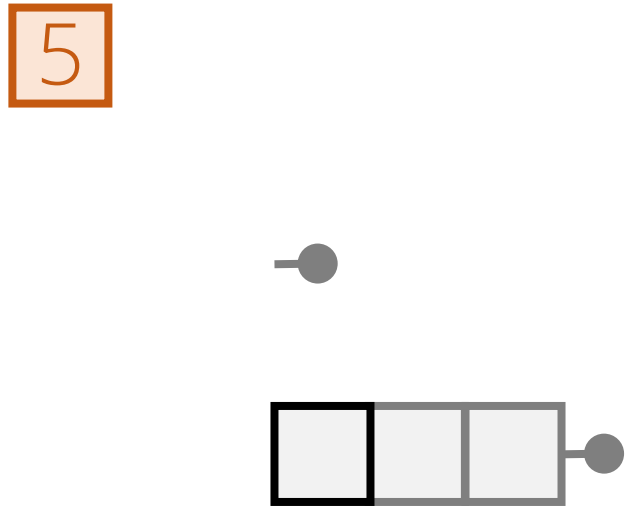
insert in a functional language

```
insert x xs =
  match xs with
```

insert in a functional language

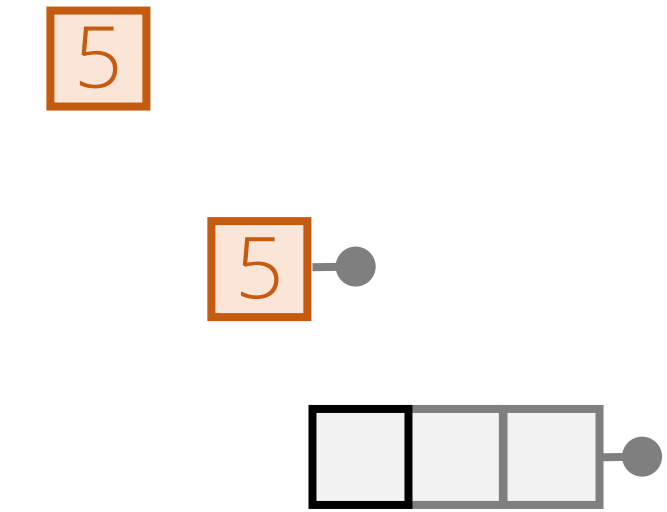
```
insert x xs =
  match xs with
    Nil →

    Cons h t →
```



insert in a functional language

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



insert in a functional language

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

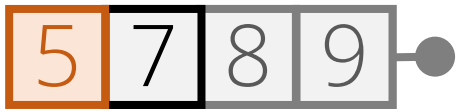
5

insert in a functional language

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

insert in a functional language

```
insert x xs =
  match xs with
    Nil →
      Cons x Nil
    Cons h t →
      if x ≤ h
        then Cons x xs
```



insert in a functional language

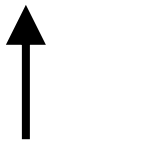
```
insert x xs =
  match xs with
    Nil →
      Cons x Nil
    Cons h t →
      if x ≤ h
        then Cons x xs
        else Cons h (insert x t)
```

insert in a functional language

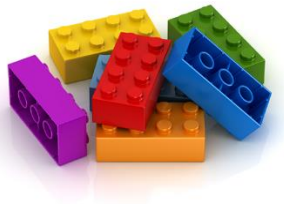
```
insert x xs =
  match xs with
    Nil →
      Cons x Nil
    Cons h t →
      if x ≤ h
        then Cons x xs
        else Cons h (insert x t)
```

our goal

specification



program space



Nil, Cons, ≤, …

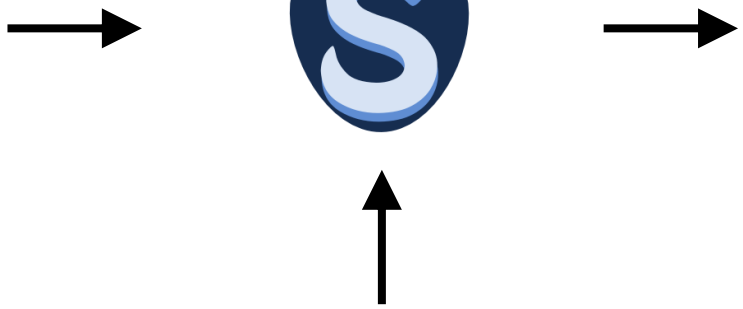
code

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

our goal

specification

?



code

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

program space

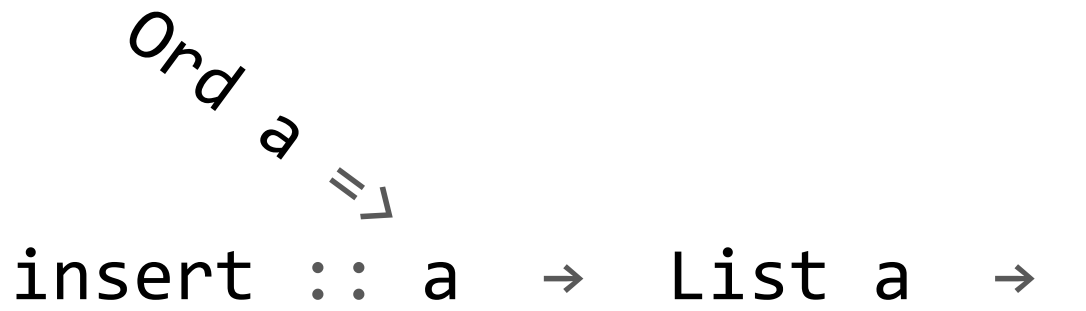
Nil, Cons, ≤, …

types are specifications

types are specifications

```
insert :: a  →  List a  →  List a
```

types are specifications



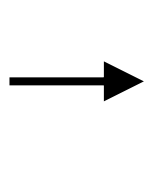
types are specifications

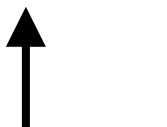
```
insert :: a  →  List a  →  List a
```

insert in synquid

specification

```
a → List a
    → List a
```





Nil, Cons, ≤, …

insert in synquid

specification

code

```
a → List a
    → List a
```

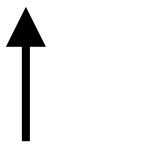
Nil, Cons, ≤, …

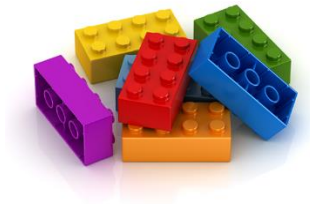
insert in synquid

specification

code

```
a → List a
    → List a
```





`Nil, Cons, ≤, …`

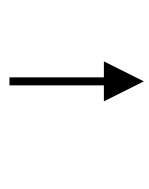
demo: insert

http://comcom.csail.mit.edu/demos/#1-insert

insert in synquid

specification

```
a → List a
    → List a
```



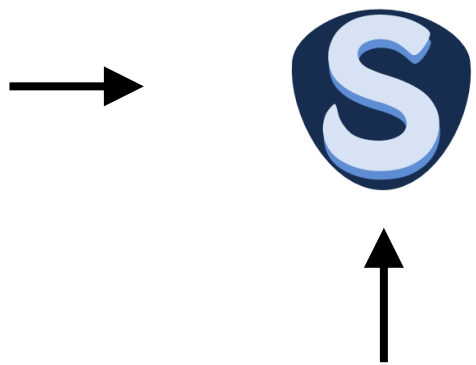
Nil, Cons, ≤, …

insert in synquid

specification

code

```
a → List a
    → List a
```



```
Nil
```

Nil, Cons, ≤, …

insert in synquid

specification

code

```
a → list a
```

ambiguous!

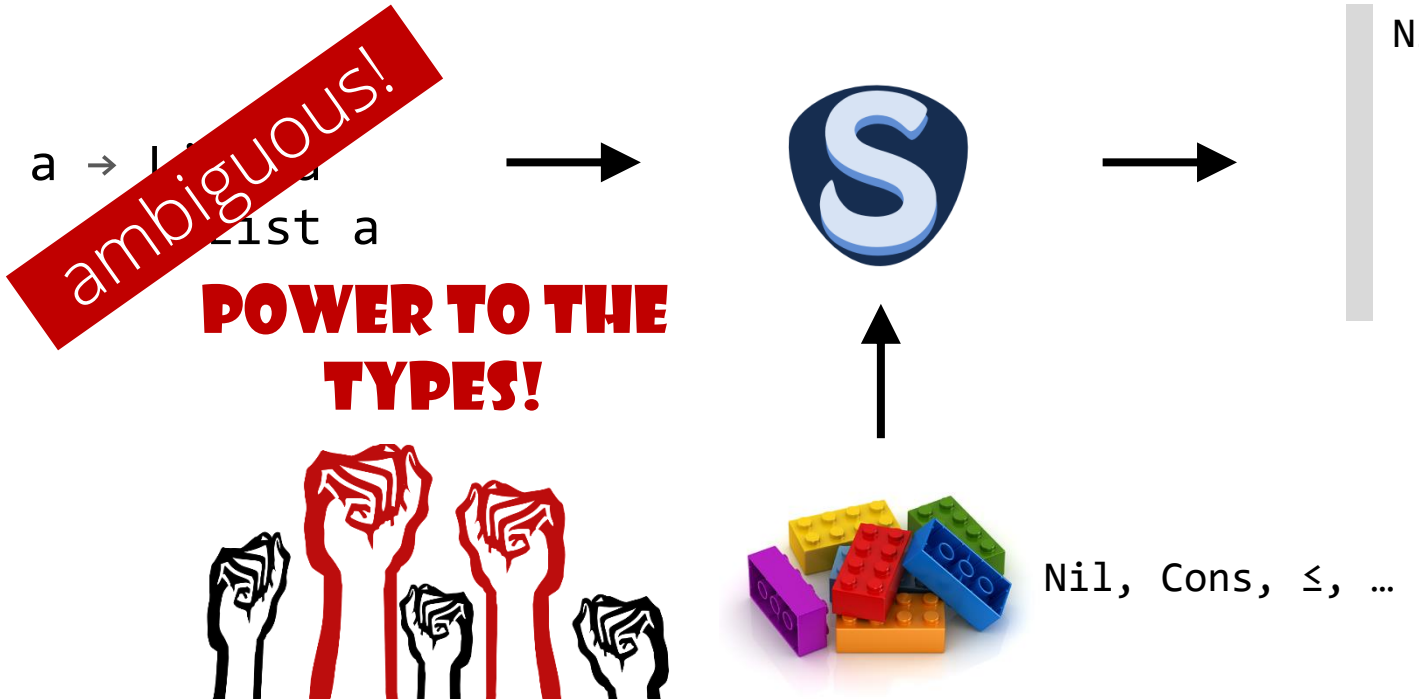
```
Nil
```

Nil, Cons, ≤, …

insert in synquid

specification

code



specification for insert

specification for insert

Input:

X

specification for insert

Input:

x

xs: sorted list

specification for insert

Input:

x

xs: sorted list

Output:

ys: sorted list

specification for insert

Input:

x

xs: sorted list

Output:

ys: sorted list

elems ys = elems xs ∪ {x}

specification for insert

Input:

x

xs: sorted list

Output:

ys: sorted list

elems ys = elems xs ∪ {x}

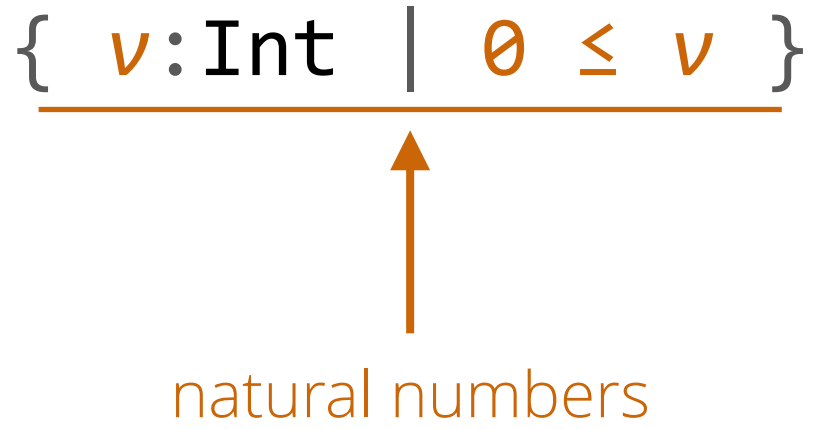
can I write this as a type?

refinement types

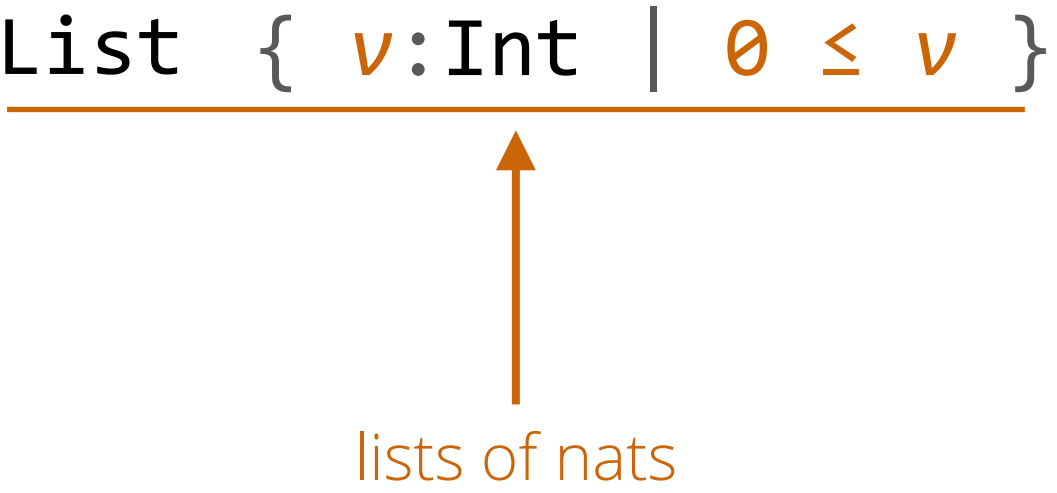
`Int`

refinement types

refinement types



refinement types



refinement types: sorted lists

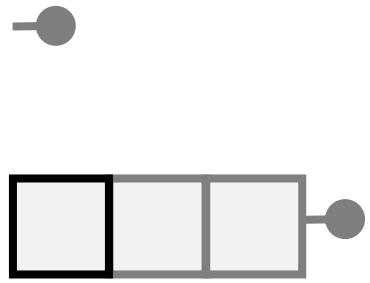
```
data List a where
```

refinement types: sorted lists

```
data List a where
  Nil :: List a
```

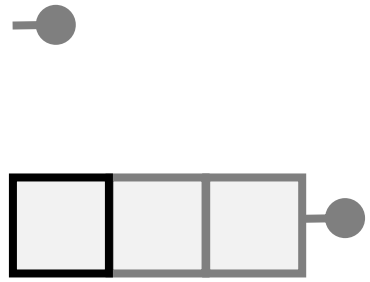
refinement types: sorted lists

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



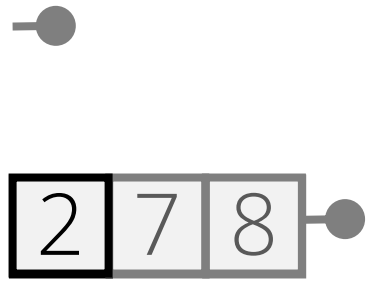
refinement types: sorted lists

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



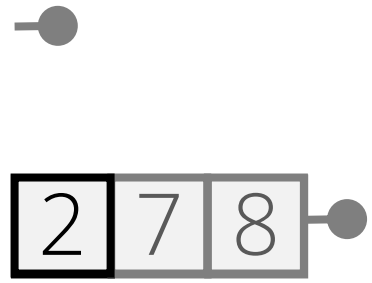
refinement types: sorted lists

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



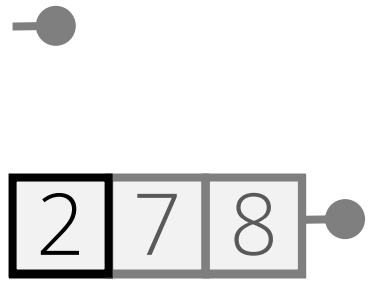
refinement types: sorted lists

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



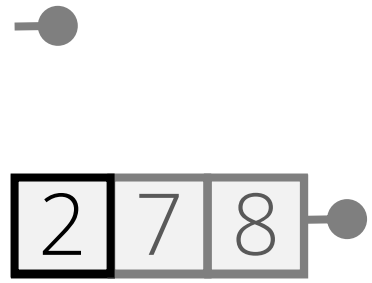
refinement types: sorted lists

```
data SList a where
  Nil :: SList a
  Cons :: h:a →
          t:SList {v:a | h ≤ v} →
          SList a
```



refinement types: sorted lists

```
data SList a where
  Nil :: SList a
  Cons :: h:a →
          t:SList {v:a | h ≤ v} →
          SList a
```



all you need
is one simple predicate!

refinement type for insert

```
insert :: x:a  →  xs:List a   →
                  List a
```

refinement type for insert

```
insert :: x:a  →  xs:SList a  →
                  List a
```

refinement type for insert

```
insert :: x:a  →  xs:SList a  →
                  SList a
```

refinement type for insert

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

insert in synquid

specification

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

demo: insert

http://comcom.csail.mit.edu/demos/#1-insert

part I

synquid

refinement types → recursive programs

1. types as specifications
2. type-directed search

how did this happen?

specification

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

synthesis as search

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

specification

synthesis as search

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

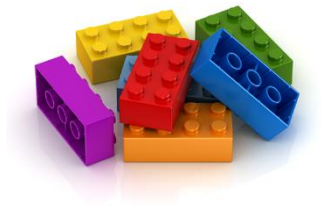
specification

synthesis as search

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

specification

components

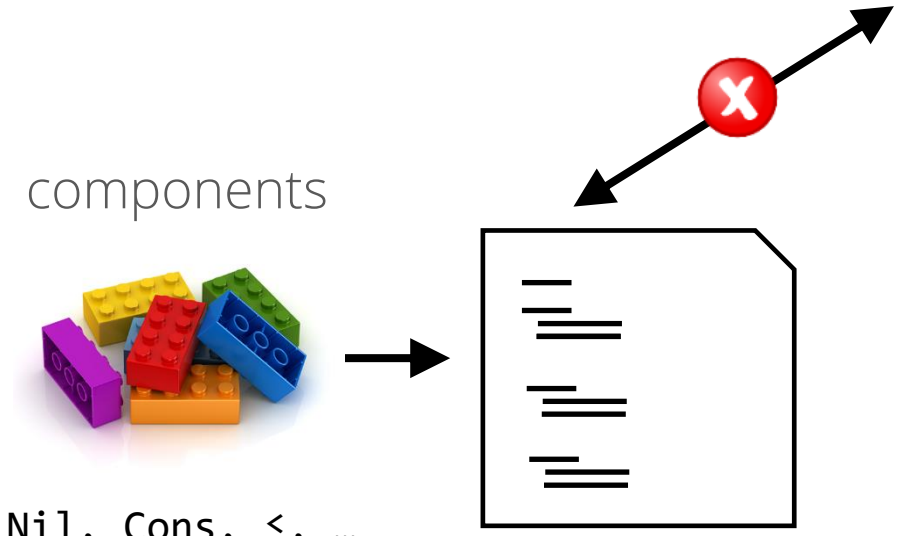


`Nil, Cons, ≤, …`

synthesis as search

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

specification



components

Nil, Cons, ≤, …

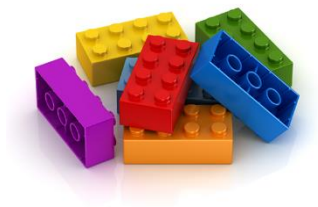
code

synthesis as search

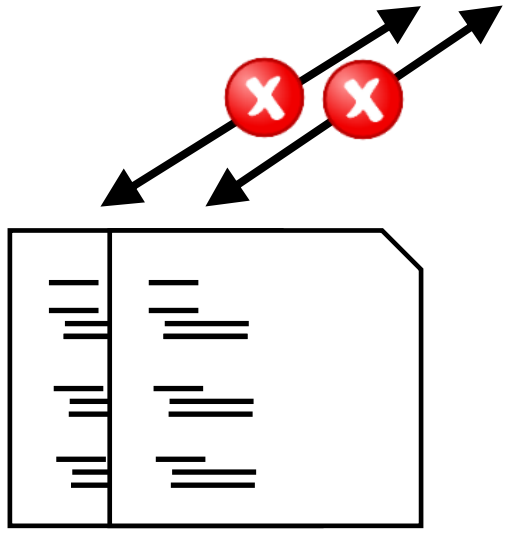
```
insert :: x:a →
  xs:SList a →
  {v:SList a | elems v =
    elems xs ∪ {x}}
```

specification

components



```
Nil, Cons, ≤, …
```



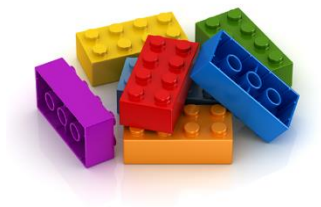
code

synthesis as search

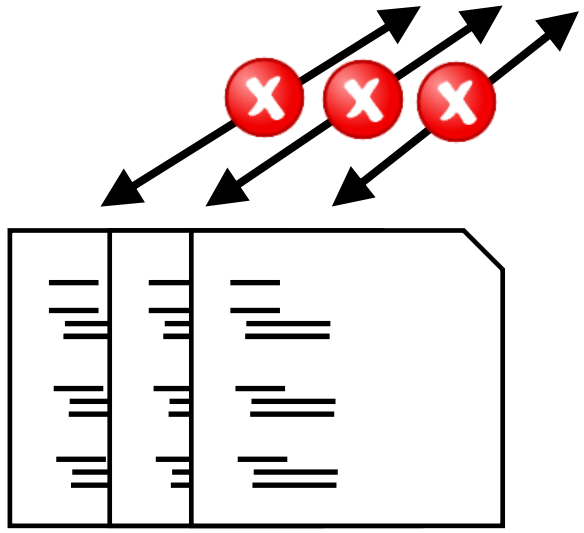
```
insert :: x:a →
  xs:SList a →
  {v:SList a | elems v =
    elems xs ∪ {x}}
```

specification

components



`Nil, Cons, ≤, …`

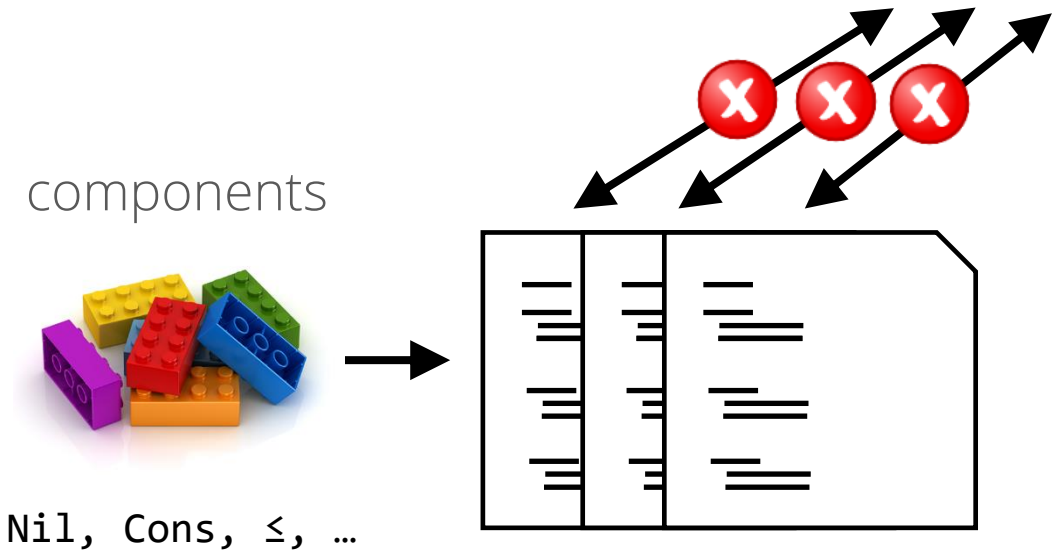


code

synthesis as search

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

specification



components

Nil, Cons, ≤, …

code

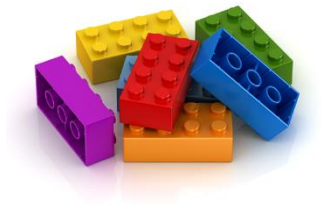
…

synthesis as search

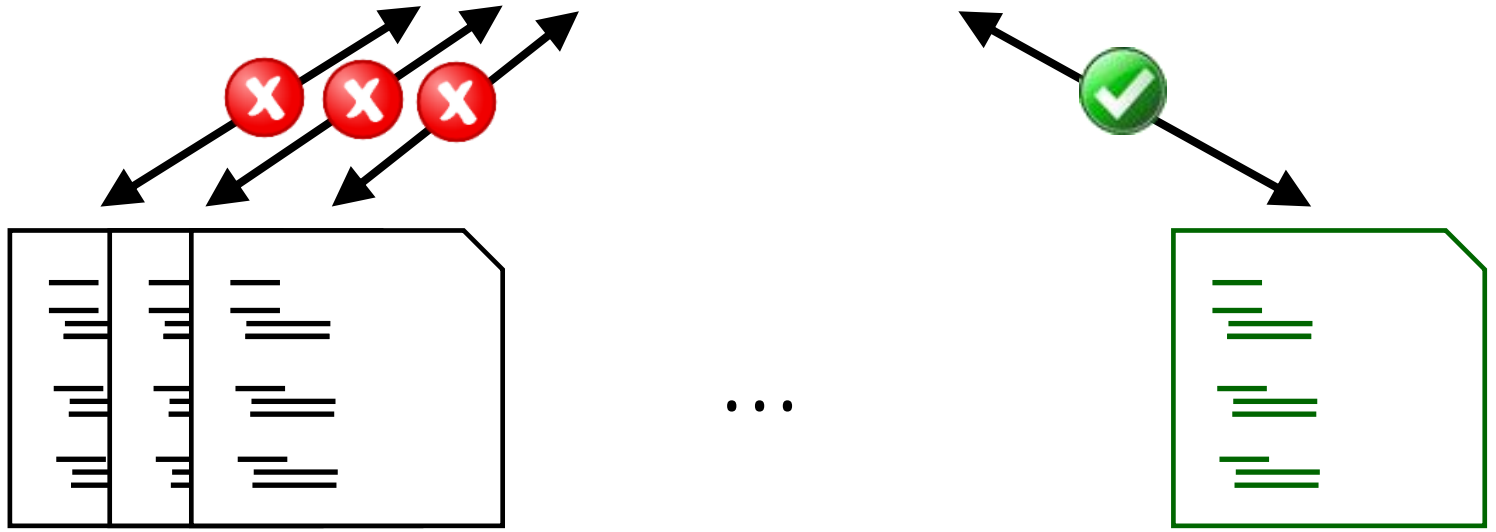
```
insert :: x:a →
  xs:SList a →
  {v:SList a | elems v =
    elems xs ∪ {x}}
```

specification

components

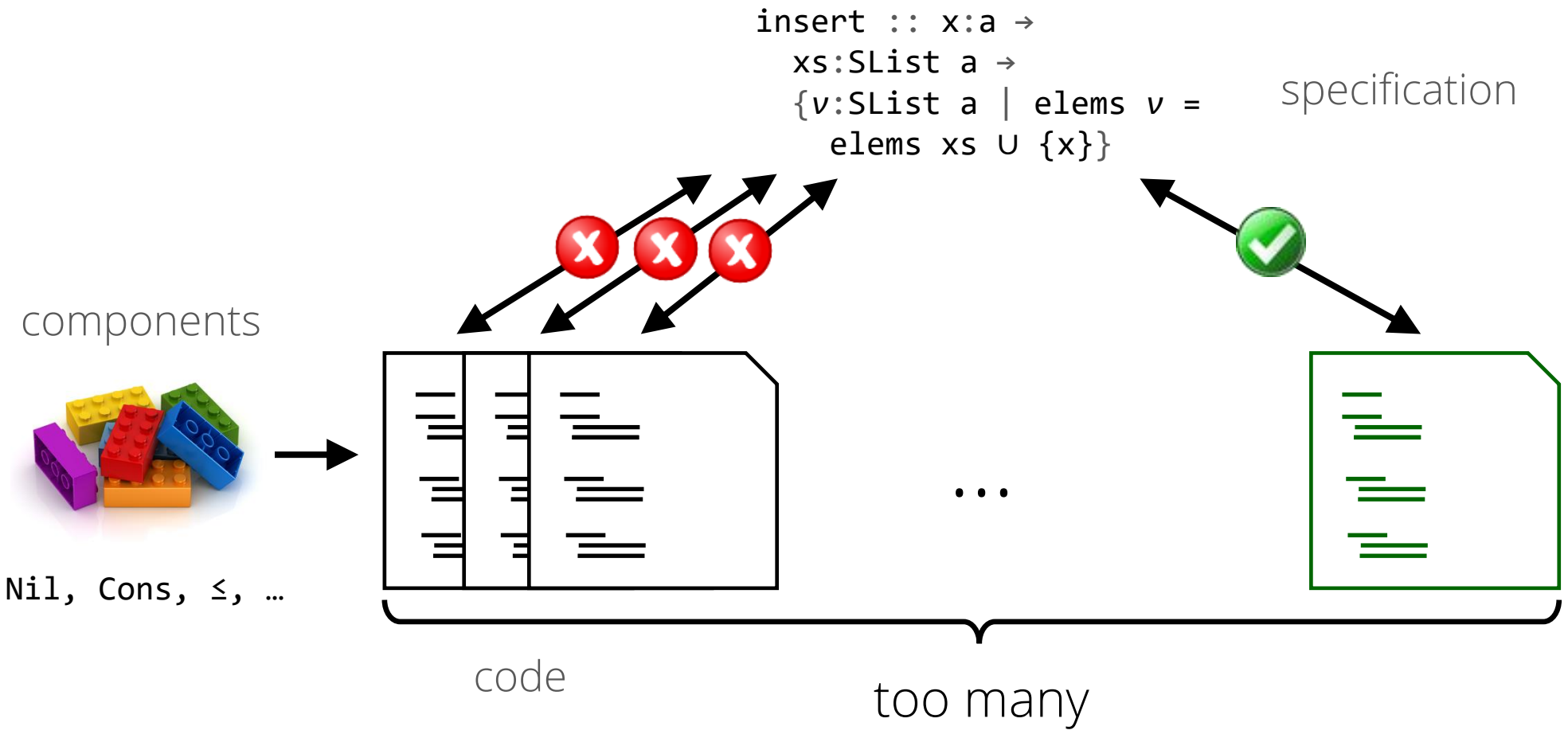


Nil, Cons, ≤, …

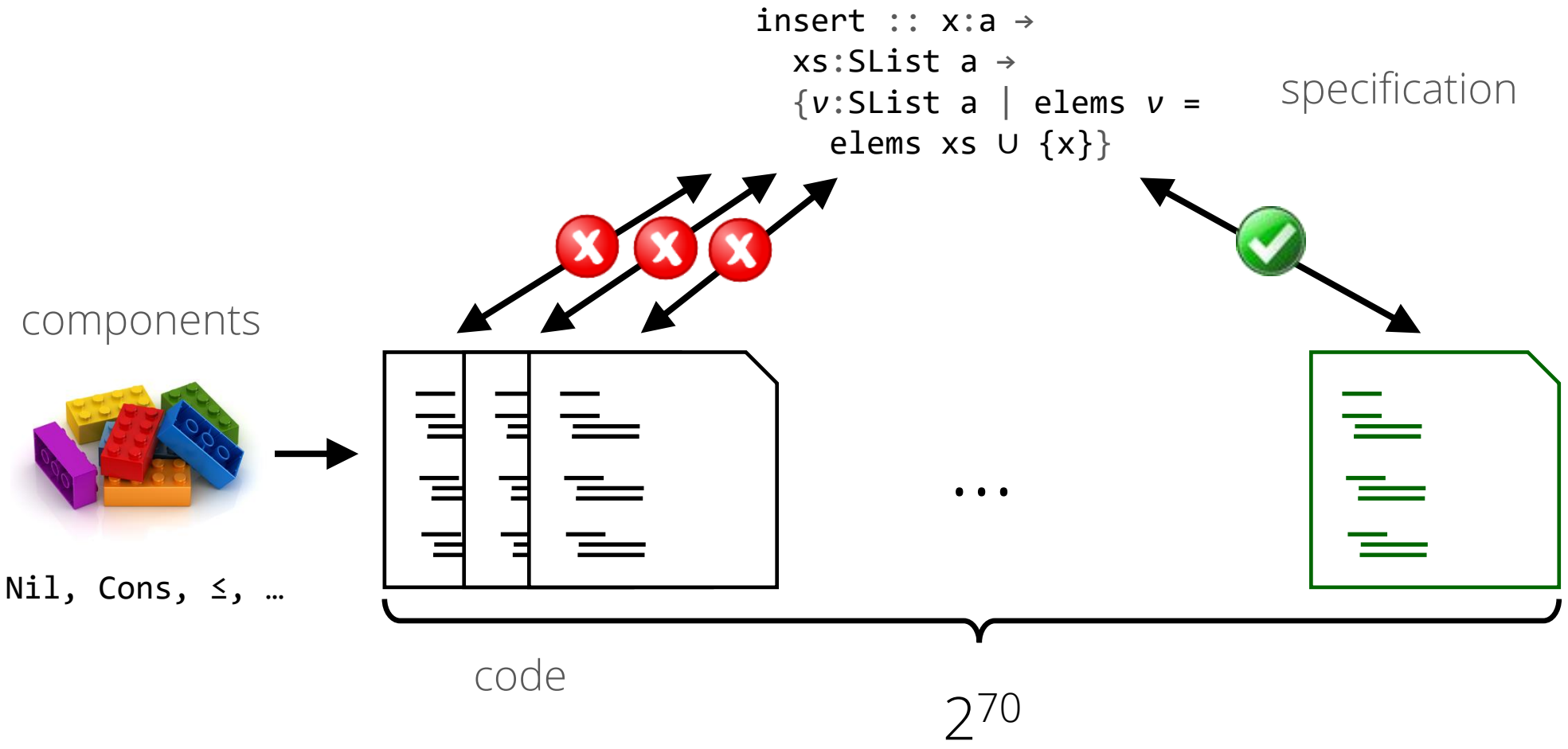


code

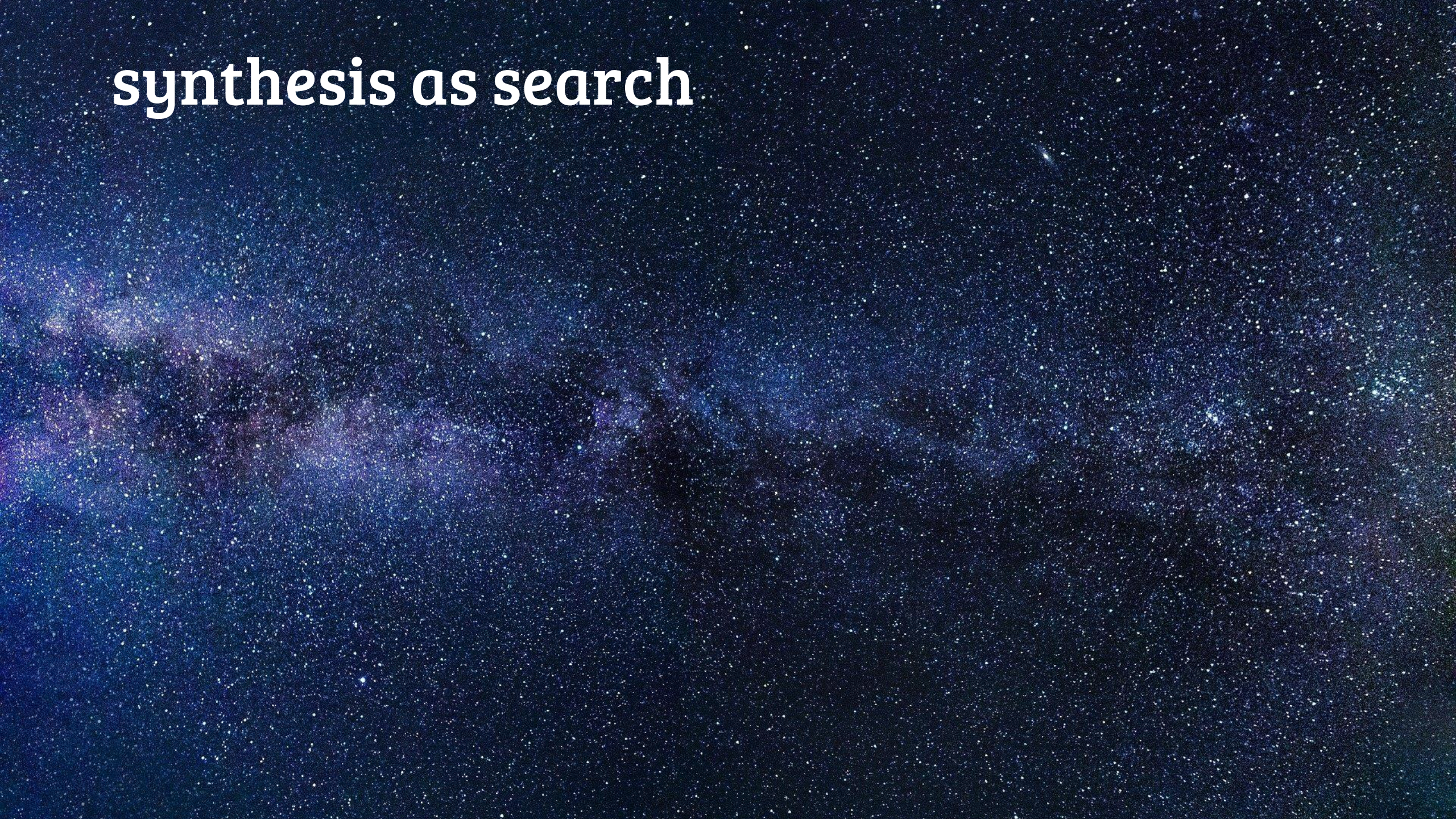
synthesis as search



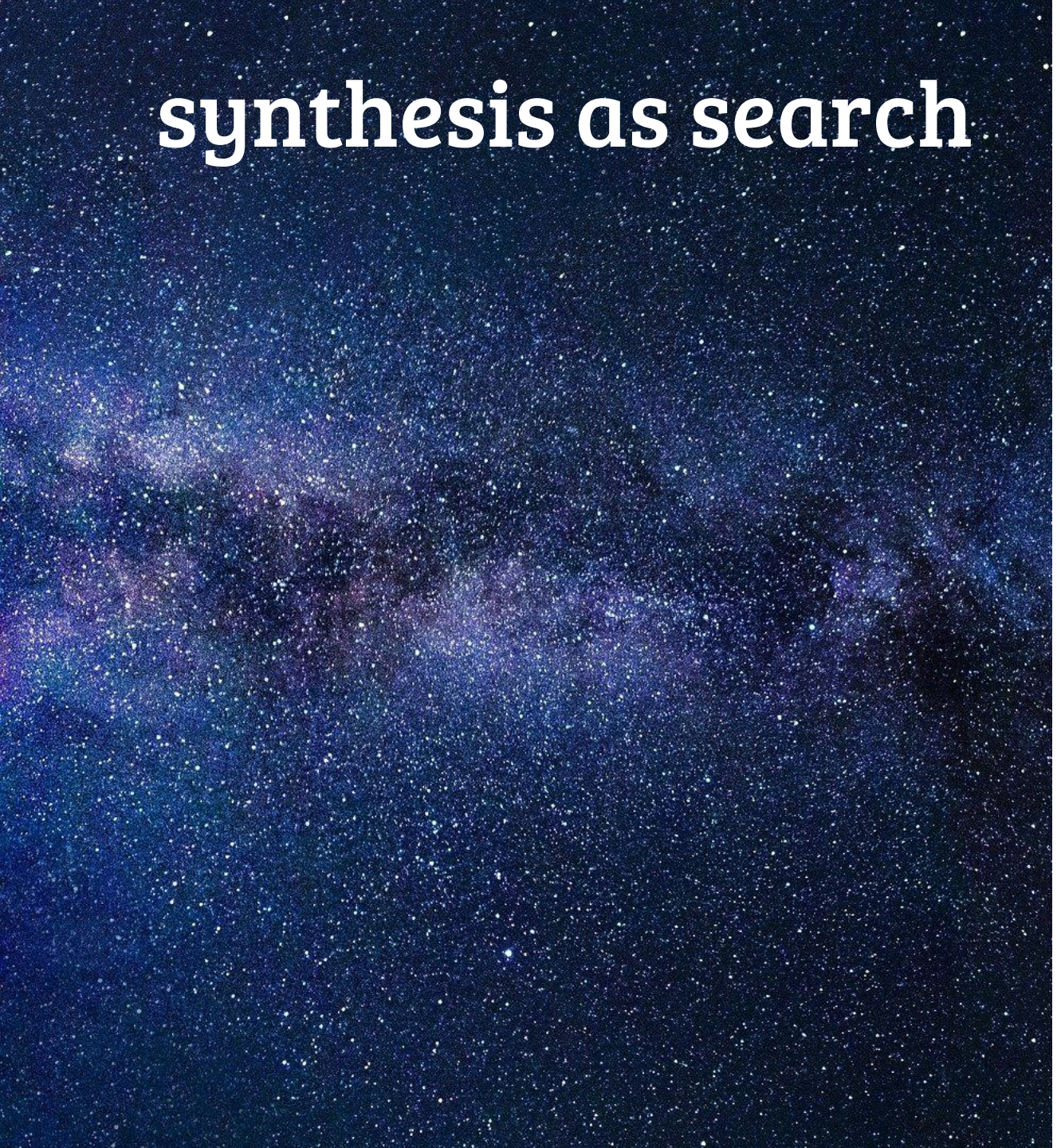
synthesis as search

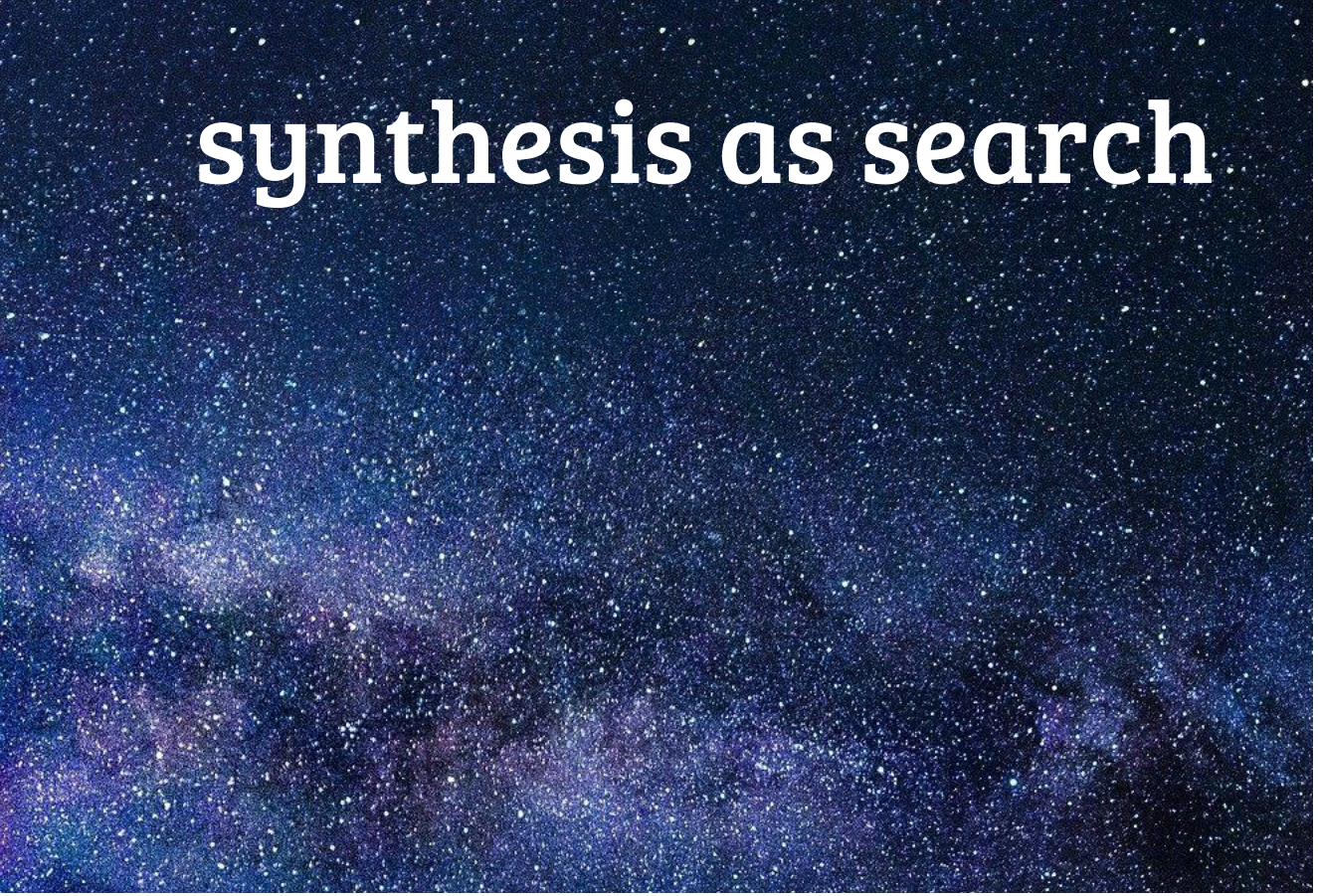


synthesis as search



synthesis as search




synthesis as search

key idea: reject hopeless programs early

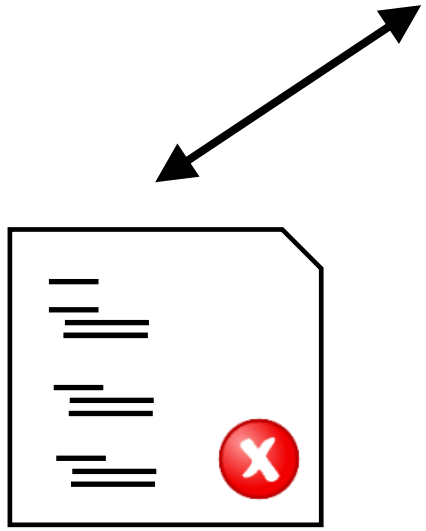
key idea: reject hopeless programs early

(during construction)

rejecting hopeless programs

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

specification

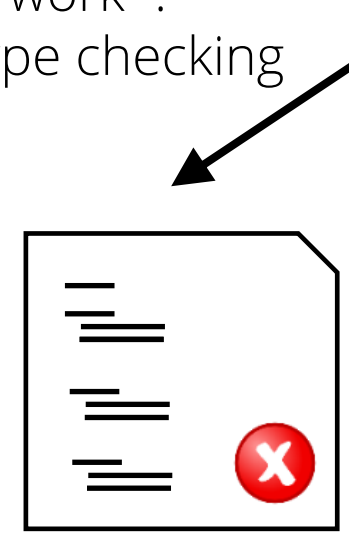


rejecting hopeless programs

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

specification

previous work*:
bottom-up type checking



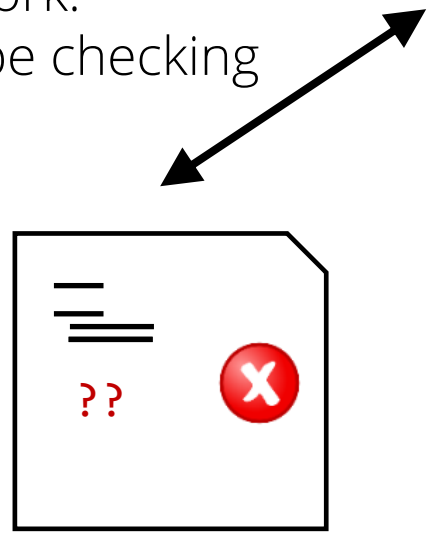
*Rondon, Kawaguchi, Jhala: *Liquid types.* [PLDI 2008]

rejecting hopeless programs

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

specification

our work:
top-down type checking

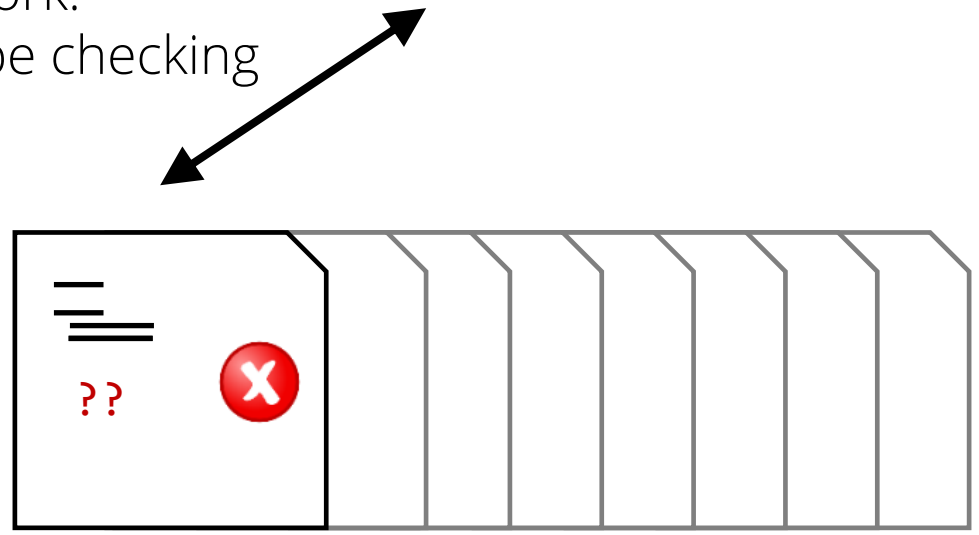


rejecting hopeless programs

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

specification

our work:
top-down type checking



rejecting hopeless programs

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

rejecting hopeless programs

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

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

rejecting hopeless programs

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

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

hopeless?

rejecting hopeless programs

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

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

rejecting hopeless programs

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

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

rejecting hopeless programs

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

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

hopeless?

rejecting hopeless programs

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

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

rejecting hopeless programs

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

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

rejecting hopeless programs

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

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

hopeless?

rejecting hopeless programs

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

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

hopeless:
output must
always contain x!

rejecting hopeless programs

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

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

2^{50}

hopeless:
output must
always contain x!

top-down type checking

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

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

top-down type checking

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

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

top-down type checking

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

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

top-down type checking

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

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

Constraints:
∀x: {} = {} ∪ {x}

top-down type checking

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

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

Constraints:
∀x: {} = {} ∪ {x}

SMT solver: INVALID!

top-down type checking

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

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

Constraints:
∀x: {} = {} ∪ {x}

SMT solver: INVALID!

insert in synquid

specification

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

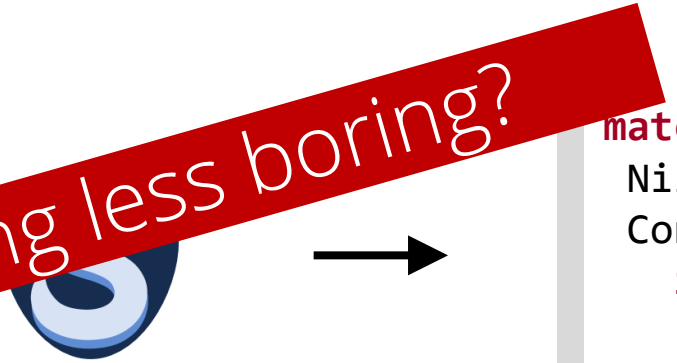
code

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

insert in synquid

specification

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



something less boring?

code

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

case study: negation normal form

$$\neg(a \Rightarrow b \vee c)$$

case study: negation normal form

$\neg(a \Rightarrow b \vee c)$

1. only $\neg$, $\wedge$, $\vee$
2. $\neg$ only at variables

case study: negation normal form

$\neg(a \Rightarrow b \vee c)$

1. only ¬, ∧, ∨
2. ¬ only at variables

case study: negation normal form

$\neg(a \Rightarrow b \vee c)$

⇒ def.

$\neg(\neg a \vee (b \vee c))$

1. only ¬, ∧, ∨
2. ¬ only at variables

case study: negation normal form

$\neg(a \Rightarrow b \vee c)$

⇒ def.

$\neg(\neg a \vee (b \vee c))$

De Morgan

$\neg\neg a \wedge \neg(b \vee c)$

1. only ¬, ∧, ∨
2. ¬ only at variables

case study: negation normal form

$\neg(a \Rightarrow b \vee c)$

⇒ def.

$\neg(\neg a \vee (b \vee c))$

De Morgan

$\neg\neg a \wedge \neg(b \vee c)$

double-neg

$a \wedge \neg(b \vee c)$

1. only ¬, ∧, ∨
2. ¬ only at variables

case study: negation normal form

$\neg(a \Rightarrow b \vee c)$

⇒ def.

$\neg(\neg a \vee (b \vee c))$

De Morgan

$\neg\neg a \wedge \neg(b \vee c)$

double-neg

$a \wedge \neg(b \vee c)$

De Morgan

$a \wedge (\neg b \vee \neg c)$

1. only ¬, ∧, ∨
2. ¬ only at variables

nnf: data types

```
data Fml where
```

nnf: data types

```
data Fml where
  Var :: String     → Fml
```

nnf: data types

```
data Fml where
  Var :: String     → Fml
  Not :: Fml        → Fml
```

nnf: data types

```
data Fml where
  Var :: String     → Fml
  Not :: Fml        → Fml
  And :: Fml → Fml → Fml
```

nnf: data types

```
data Fml where
  Var :: String     → Fml
  Not :: Fml        → Fml
  And :: Fml → Fml → Fml
  Or  :: Fml → Fml → Fml
```

nnf: data types

```
data Fml where
  Var :: String     → Fml
  Not :: Fml        → Fml
  And :: Fml → Fml → Fml
  Or  :: Fml → Fml → Fml
  Imp :: Fml → Fml → Fml
```

nnf: data types

```
data Fml where
  Var :: String     → Fml
  Not :: Fml        → Fml
  And :: Fml → Fml → Fml
  Or  :: Fml → Fml → Fml
  Imp :: Fml → Fml → Fml
```

```
data NNF where
```

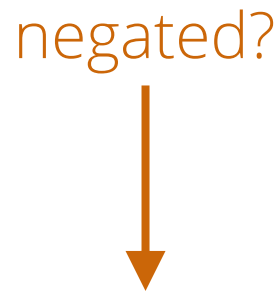
nnf: data types

```
data Fml where
  Var :: String     → Fml
  Not :: Fml        → Fml
  And :: Fml → Fml → Fml
  Or  :: Fml → Fml → Fml
  Imp :: Fml → Fml → Fml
```

```
data NNF where
  NAtom :: String → Bool
                  → NNF
```

nnf: data types

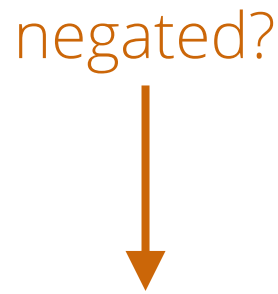
```
data Fml where
  Var :: String     → Fml
  Not :: Fml        → Fml
  And :: Fml → Fml → Fml
  Or  :: Fml → Fml → Fml
  Imp :: Fml → Fml → Fml
```




```
data NNF where
  NAtom :: String → Bool
                  → NNF
```

nnf: data types

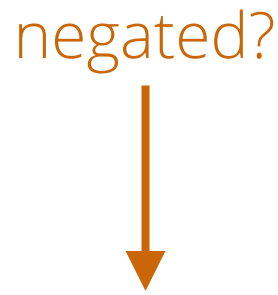
```
data Fml where
  Var :: String     → Fml
  Not :: Fml        → Fml
  And :: Fml → Fml → Fml
  Or  :: Fml → Fml → Fml
  Imp :: Fml → Fml → Fml
```




```
data NNF where
  NAtom :: String → Bool
                  → NNF
  NAnd  :: NNF → NNF → NNF
```

nnf: data types

```
data Fml where
  Var :: String     → Fml
  Not :: Fml        → Fml
  And :: Fml → Fml → Fml
  Or  :: Fml → Fml → Fml
  Imp :: Fml → Fml → Fml
```




```
data NNF where
  NAtom :: String → Bool
                  → NNF
  NAnd  :: NNF → NNF → NNF
  NOr   :: NNF → NNF → NNF
```

nnf: specification

```
data Fml where
  Var :: String     → Fml
  Not :: Fml        → Fml
  And :: Fml → Fml → Fml
  Or  :: Fml → Fml → Fml
  Imp :: Fml → Fml → Fml
```

```
data NNF where
  NAtom :: String → Bool
                  → NNF
  NAnd :: NNF → NNF → NNF
  NOr  :: NNF → NNF → NNF
```

nnf: specification

```
data Fml where
  Var :: String     → Fml
  Not :: Fml        → Fml
  And :: Fml → Fml → Fml
  Or  :: Fml → Fml → Fml
  Imp :: Fml → Fml → Fml
```

```
data NNF where
  NAtom :: String → Bool
                  → NNF
  NAnd :: NNF → NNF → NNF
  NOr  :: NNF → NNF → NNF
```

```
measure eval :: Fml → Bool where
```

nnf: specification

```
data Fml where
  Var :: String     → Fml
  Not :: Fml        → Fml
  And :: Fml → Fml → Fml
  Or  :: Fml → Fml → Fml
  Imp :: Fml → Fml → Fml
```

```
data NNF where
  NAtom :: String → Bool
                  → NNF
  NAnd :: NNF → NNF → NNF
  NOr  :: NNF → NNF → NNF
```

```
measure eval :: Fml → Bool where
  Var v    → env v
```

nnf: specification

```
data Fml where
  Var :: String    → Fml
  Not :: Fml       → Fml
  And :: Fml → Fml → Fml
  Or  :: Fml → Fml → Fml
  Imp :: Fml → Fml → Fml
```

```
data NNF where
  NAtom :: String → Bool
                  → NNF
  NAnd :: NNF → NNF → NNF
  NOr  :: NNF → NNF → NNF
```

```
measure eval :: Fml → Bool where
  Var v   → env v
  Not f   → !(eval f)
```

nnf: specification

```
data Fml where
  Var :: String     → Fml
  Not :: Fml        → Fml
  And :: Fml → Fml → Fml
  Or  :: Fml → Fml → Fml
  Imp :: Fml → Fml → Fml
```

```
data NNF where
  NAtom :: String → Bool
                  → NNF
  NAnd :: NNF → NNF → NNF
  NOr  :: NNF → NNF → NNF
```

```
measure eval :: Fml → Bool where
  Var v   → env v
  Not f   → !(eval f)
  And l r → eval l &&  eval r
```

nnf: specification

```
data Fml where
  Var :: String     → Fml
  Not :: Fml        → Fml
  And :: Fml → Fml → Fml
  Or  :: Fml → Fml → Fml
  Imp :: Fml → Fml → Fml
```

```
data NNF where
  NAtom :: String → Bool
                  → NNF
  NAnd :: NNF → NNF → NNF
  NOr  :: NNF → NNF → NNF
```

```
measure eval :: Fml → Bool where
  Var v   → env v
  Not f   → !(eval f)
  And l r → eval l &&  eval r
  Or l r  → eval l ||  eval r
```

nnf: specification

```
data Fml where
  Var :: String     → Fml
  Not :: Fml        → Fml
  And :: Fml → Fml → Fml
  Or  :: Fml → Fml → Fml
  Imp :: Fml → Fml → Fml
```

```
data NNF where
  NAtom :: String → Bool
                  → NNF
  NAnd :: NNF → NNF → NNF
  NOr  :: NNF → NNF → NNF
```

```
measure eval :: Fml → Bool where
  Var v   → env v
  Not f   → !(eval f)
  And l r → eval l &&  eval r
  Or l r  → eval l ||  eval r
  Imp l r → eval l ==> eval r
```

nnf: specification

```
data Fml where
  Var :: String    → Fml
  Not :: Fml       → Fml
  And :: Fml → Fml → Fml
  Or  :: Fml → Fml → Fml
  Imp :: Fml → Fml → Fml
```

```
data NNF where
  NAtom :: String → Bool
                  → NNF
  NAnd :: NNF → NNF → NNF
  NOr  :: NNF → NNF → NNF
```

```
measure eval :: Fml → Bool where
  Var v   → env v
  Not f   → !(eval f)
  And l r → eval l &&  eval r
  Or l r  → eval l ||  eval r
  Imp l r → eval l ==> eval r
```

```
measure nEval :: NNF → Bool where
```

nnf: specification

```
data Fml where
  Var :: String     → Fml
  Not :: Fml        → Fml
  And :: Fml → Fml → Fml
  Or  :: Fml → Fml → Fml
  Imp :: Fml → Fml → Fml
```

```
data NNF where
  NAtom :: String → Bool
                  → NNF
  NAnd :: NNF → NNF → NNF
  NOr  :: NNF → NNF → NNF
```

```
measure eval :: Fml → Bool where
  Var v   → env v
  Not f   → !(eval f)
  And l r → eval l &&  eval r
  Or l r  → eval l ||  eval r
  Imp l r → eval l ==> eval r
```

```
measure nEval :: NNF → Bool where
  NAtom neg v → if neg then env v
                       else !(env v)
```

nnf: specification

```
data Fml where
  Var :: String     → Fml
  Not :: Fml        → Fml
  And :: Fml → Fml → Fml
  Or  :: Fml → Fml → Fml
  Imp :: Fml → Fml → Fml
```

```
data NNF where
  NAtom :: String → Bool
                  → NNF
  NAnd :: NNF → NNF → NNF
  NOr  :: NNF → NNF → NNF
```

```
measure eval :: Fml → Bool where
  Var v   → env v
  Not f   → !(eval f)
  And l r → eval l &&  eval r
  Or l r  → eval l ||  eval r
  Imp l r → eval l ==> eval r
```

```
measure nEval :: NNF → Bool where
  NAtom neg v → if neg then env v
                       else !(env v)
  NAnd l r  → nEval l && nEval r
```

nnf: specification

```
data Fml where
  Var :: String     → Fml
  Not :: Fml        → Fml
  And :: Fml → Fml → Fml
  Or  :: Fml → Fml → Fml
  Imp :: Fml → Fml → Fml
```

```
data NNF where
  NAtom :: String → Bool
                  → NNF
  NAnd :: NNF → NNF → NNF
  NOr  :: NNF → NNF → NNF
```

```
measure eval :: Fml → Bool where
  Var v   → env v
  Not f   → !(eval f)
  And l r → eval l &&  eval r
  Or l r  → eval l ||  eval r
  Imp l r → eval l ==> eval r
```

```
measure nEval :: NNF → Bool where
  NAtom neg v → if neg then env v
                       else !(env v)
  NAnd l r  → nEval l && nEval r
  NOr l r   → nEval l || nEval r
```

nnf: specification

```
nnf :: f:Fml  → {ν:NNF | nEval ν = eval f}
```

nnf: synthesized code

```
nnf :: f:Fml  →  {ν:NNF | nEval ν = eval f}
```

nnf: synthesized code

```
nnf :: f:Fml  →  {v:NNF | nEval v = eval f}
nnf p = match p with
    BoolLiteral x2 → if x2
        then NOr (NAtom dummy x2) (NAtom dummy False)
        else NAnd (NAtom dummy x2) (NAtom dummy True)
    Var x16 → NAtom x16 False
    Not x20 → match x20 with
      BoolLiteral x22 → if x22
        then nnf (BoolLiteral False)
        else nnf (BoolLiteral True)
      Var x28 → NAtom x28 True
      Not x32 → nnf x32
      And x36 x37 → NOr (nnf (Not x36)) (nnf (Not x37))
      Or x46 x47 → NAnd (nnf (Not x46)) (nnf (Not x47))
      Implies x56 x57 → NAnd (nnf x56) (nnf (Not x57))
    And x65 x66 → NAnd (nnf x65) (nnf x66)
    Or x73 x74 → NOr (nnf x73) (nnf x74)
    Imp x81 x82 → NOr (nnf x82) (nnf (Not x81))
```

nnf: synthesized code

```
nnf :: f:Fml  →  {v:NNF | nEval v = eval f}
nnf p = match p with
    BoolLiteral x2 → if x2
        then NOr (NAtom dummy x2) (NAtom dummy False)
        else NAnd (NAtom dummy x2) (NAtom dummy True)
    Var x16 → NAtom x16 False
    Not x20 → match x20 with
      BoolLiteral x22 → if x22
        then nnf (BoolLiteral False)
        else nnf (BoolLiteral True)
      Var x28 → NAtom x28 True
      Not x32 → nnf x32
      And x36 x37 → NOr (nnf (Not x36)) (nnf (Not x37))
      Or x46 x47 → NAnd (nnf (Not x46)) (nnf (Not x47))
      Implies x56 x57 → NAnd (nnf x56) (nnf (Not x57))
    And x65 x66 → NAnd (nnf x65) (nnf x66)
    Or x73 x74 → NOr (nnf x73) (nnf x74)
    Imp x81 x82 → NOr (nnf x82) (nnf (Not x81))
```

⇒ def.

nnf: synthesized code

```
nnf :: f:Fml  →  {v:NNF | nEval v = eval f}
nnf p = match p with
    BoolLiteral x2 → if x2
        then NOr (NAtom dummy x2) (NAtom dummy False)
        else NAnd (NAtom dummy x2) (NAtom dummy True)
    Var x16 → NAtom x16 False
    Not x20 → match x20 with
      BoolLiteral x22 → if x22
        then nnf (BoolLiteral False)
        else nnf (BoolLiteral True)
      Var x28 → NAtom x28 True
      Not x32 → nnf x32
      And x36 x37 → NOr (nnf (Not x36)) (nnf (Not x37))
      Or x46 x47 → NAnd (nnf (Not x46)) (nnf (Not x47))
      Implies x56 x57 → NAnd (nnf x56) (nnf (Not x57))
    And x65 x66 → NAnd (nnf x65) (nnf x66)
    Or x73 x74 → NOr (nnf x73) (nnf x74)
    Imp x81 x82 → NOr (nnf x82) (nnf (Not x81))
```

double-neg

⇒ def.

nnf: synthesized code

```
nnf :: f:Fml  →  {ν:NNF | nEval ν = eval f}
nnf p = match p with
    BoolLiteral x2 → if x2
        then NOr (NAtom dummy x2) (NAtom dummy False)
        else NAnd (NAtom dummy x2) (NAtom dummy True)
    Var x16 → NAtom x16 False
    Not x20 → match x20 with
      BoolLiteral x22 → if x22
        then nnf (BoolLiteral False)
        else nnf (BoolLiteral True)
      Var x28 → NAtom x28 True
      Not x32 → nnf x32
      And x36 x37 → NOr (nnf (Not x36)) (nnf (Not x37))
      Or x46 x47 → NAnd (nnf (Not x46)) (nnf (Not x47))
      Implies x56 x57 → NAnd (nnf x56) (nnf (Not x57))
    And x65 x66 → NAnd (nnf x65) (nnf x66)
    Or x73 x74 → NOr (nnf x73) (nnf x74)
    Imp x81 x82 → NOr (nnf x82) (nnf (Not x81))
```

double-neg

De Morgan

⇒ def.

this talk

part II: hoogle+

H+

Haskell types → function compositions

part I: synquid

refinement types → recursive programs

simple types

expressive types

part II

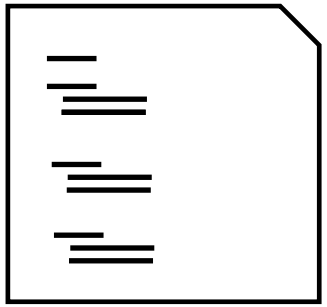
hoogle+

Haskell types → function compositions

synquid: limitation

specification

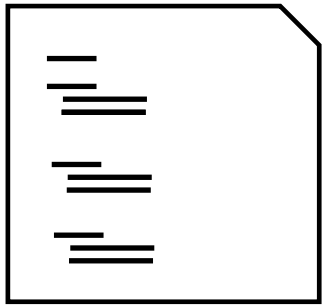
program



synquid: limitation

specification

program



part II

hoogle+

1. overcoming ambiguity
2. helping beginners with types

Haskell types → function compositions

Guo et al.: *Program Synthesis by Type-Guided Abstraction Refinement* [POPL'20]

James et al.: *Digging for Fold: Synthesis-Aided API Discovery for Haskell* [OOPSLA'20]

inspiration: hoogle

Hoogle

Char -> String -> [String]

Search

```
split :: Char -> String -> [String]
```

ghc Util

inspiration: hoogle

Hoogle

Char -> String -> [String]

Search

```
split :: Char -> String -> [String]
```

ghc Util

cannot compose functions!

example: compress a list

Input:

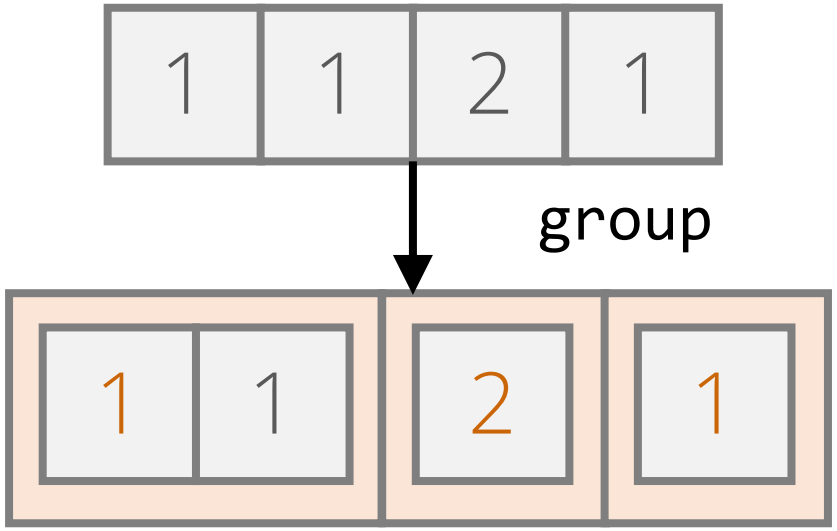
example: compress a list

Input:

Output:

example: compress a list

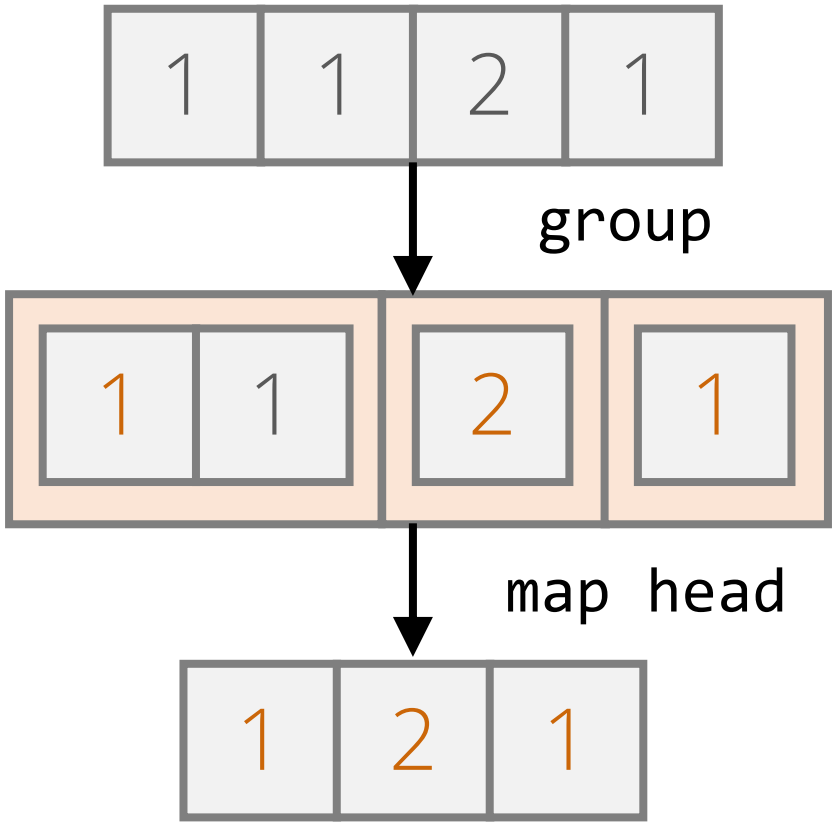
Input:



Output:

example: compress a list

Input:



Output:

compress: specification

```
compress ::          [a] → [a]
```

compress: specification

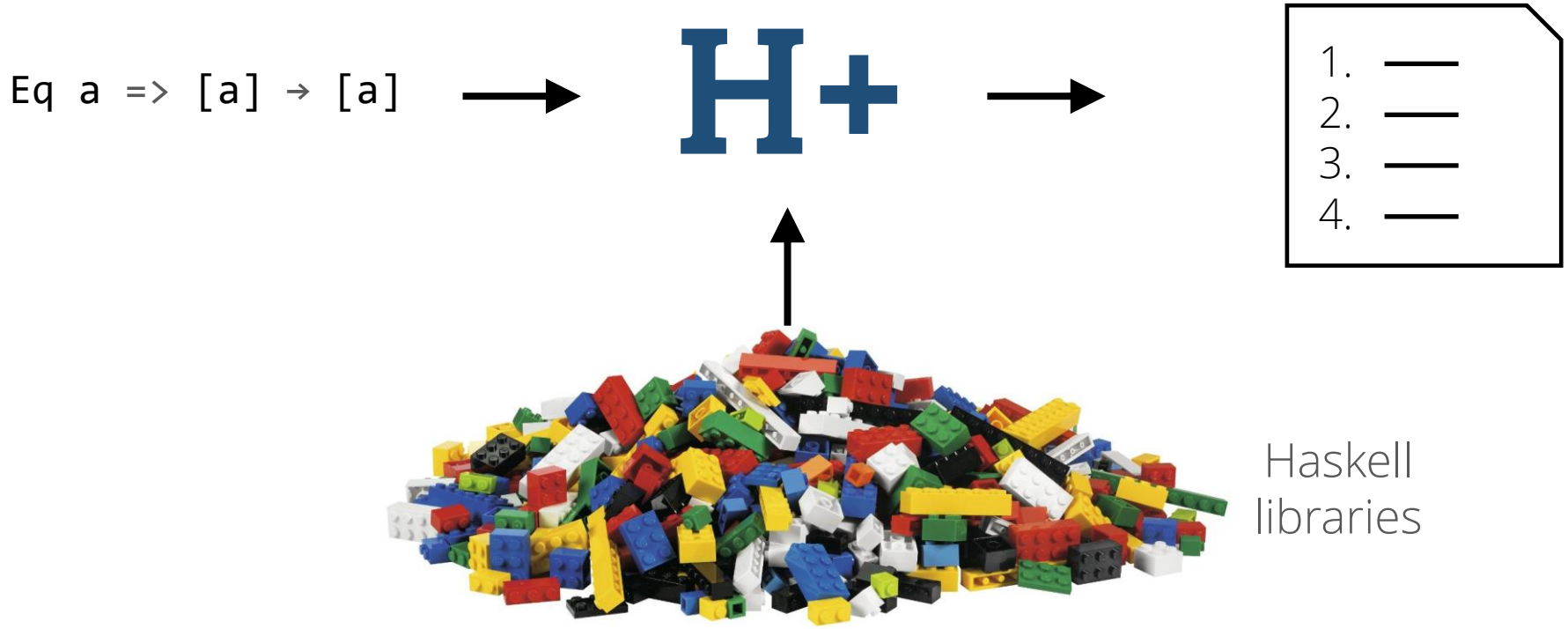
```
compress :: Eq a => [a] → [a]
```

hoogle+

specification

programs

```
Eq a => [a] → [a]
```



hoogle+

specification

programs

```
Eq a => [a] → [a]
```

1. —
2. —
3. —
4. —

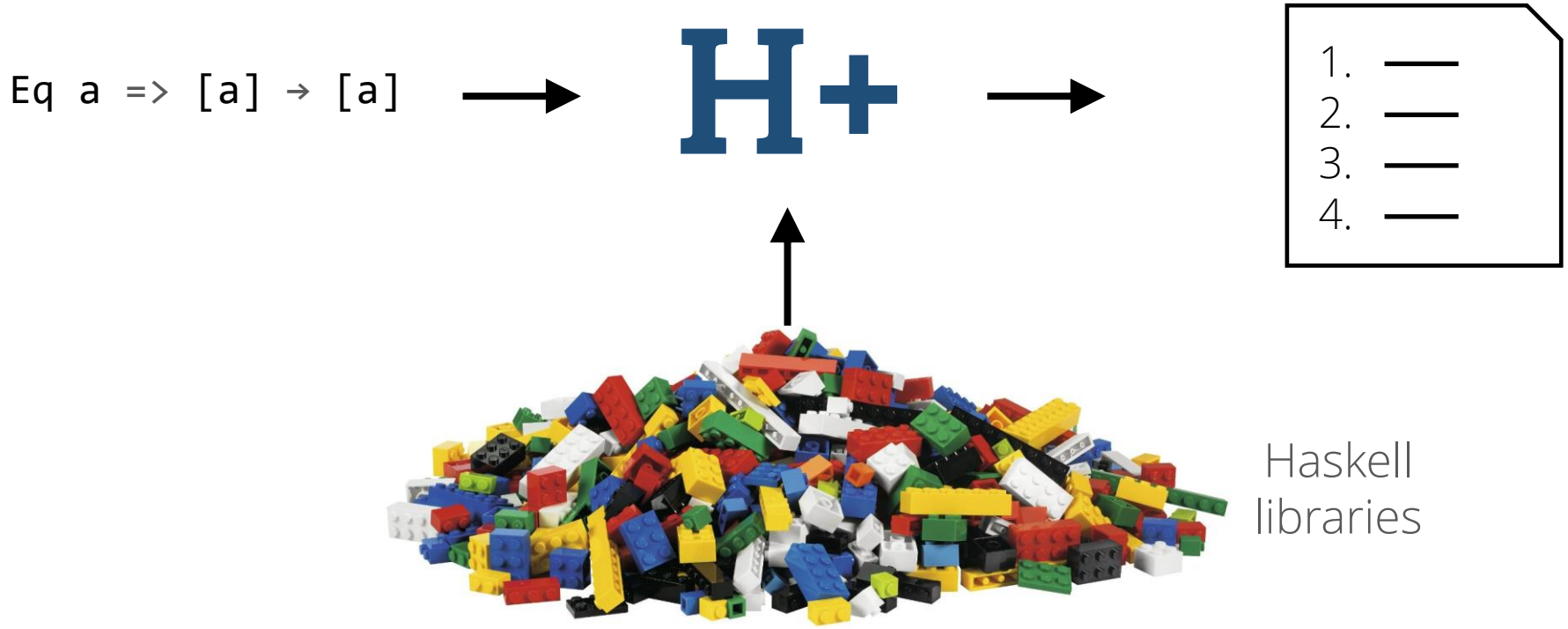
Haskell libraries

hoogle+

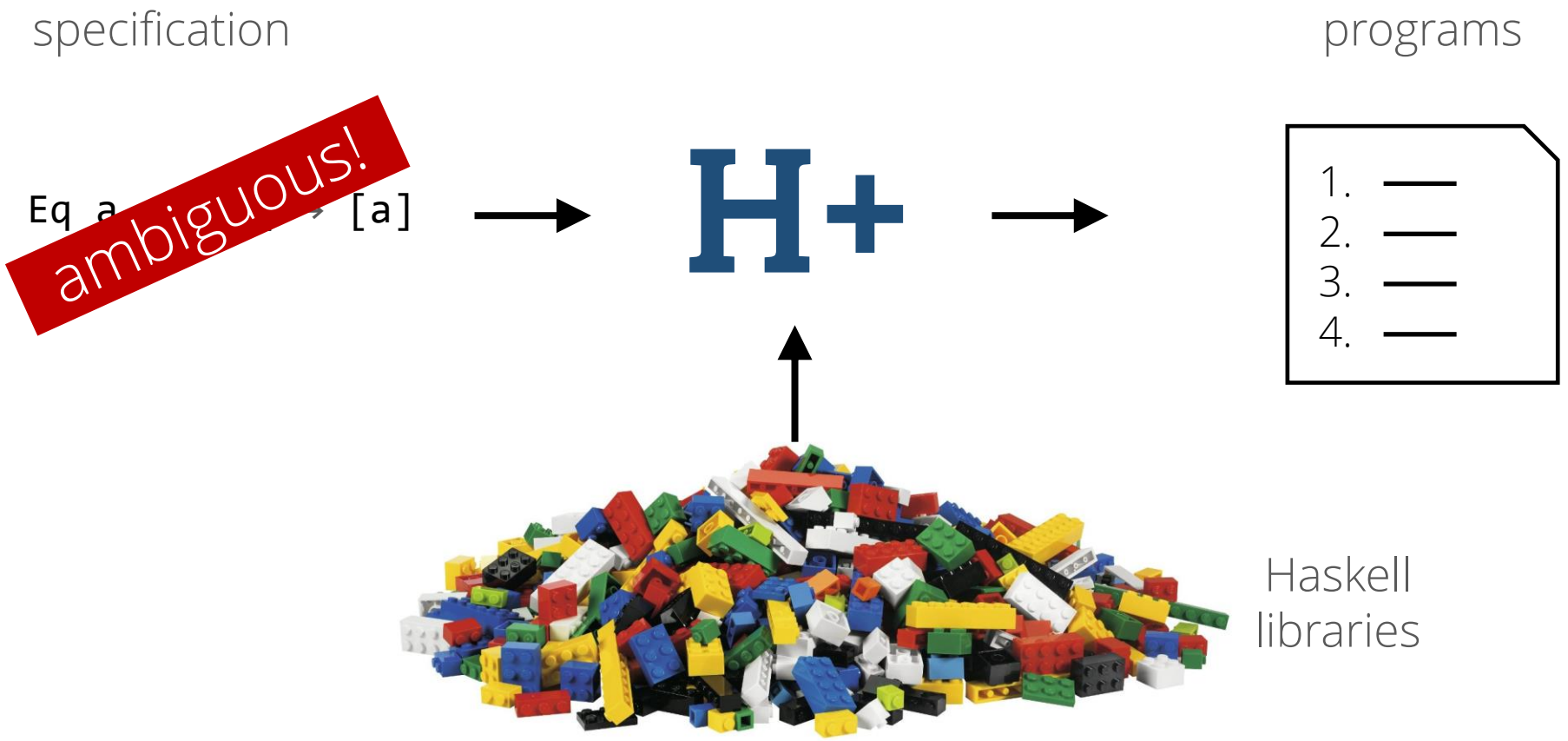
specification

programs

```
Eq a => [a] → [a]
```



hoogle+


specification
programs
Eq a
[a]
ambiguous!
H+
1.
2.
3.
4.
Haskell
libraries

hoogle+

Hoogλe+

Eq a => [a] → [a]

Search

```
\xs -> concat (group xs)
```

```
[0,1]    -> [0,1]
[0,0]    -> [0,0]
```

```
\xs -> head (group xs)
```

```
[0,1]    -> [0]
[0,0]    -> [0,0]
```

```
\xs -> last (group xs)
```

```
[0,1]    -> [1]
[0,0]    -> [0,0]
```

```
\xs -> map head (group xs)
```

```
[0,1]    -> [0,1]
[0,0]    -> [0]
```

hoogle+

Hoogλe+

Eq a => [a] → [a]

Search

```
\xs -> concat (group xs)
```

```
[0,1]    -> [0,1]
[0,0]    -> [0,0]
```

```
\xs -> head (group xs)
```

```
[0,1]    -> [0]
[0,0]    -> [0,0]
```

```
\xs -> last (group xs)
```

```
[0,1]    -> [1]
[0,0]    -> [0,0]
```

```
\xs -> map head (group xs)
```

```
[0,1]    -> [0,1]
[0,0]    -> [0]
```

part III

hoogle+

Haskell types → function compositions

1. overcoming ambiguity

too many irrelevant results

Hoogλe+ `Eq a => [a] → [a]` Search

```
\xs -> []
```

```
\xs -> xs
```

```
\xs -> head []
```

```
\xs -> tail xs
```

too many irrelevant results

Hoogλe+ `Eq a => [a] → [a]` Search

```
\xs -> []
```

```
\xs -> xs
```

```
\xs -> head []
```

always crashes!

```
\xs -> tail xs
```

too many irrelevant results

Hoogλe+ `Eq a => [a] → [a]` Search

```
\xs -> []
```

ignores the argument!

```
\xs -> xs
```

```
\xs -> head []
```

always crashes!

```
\xs -> tail xs
```

too many irrelevant results

Hoogλe+ `Eq a => [a] → [a]` Search

```
\xs -> []
```

ignores the argument!

```
\xs -> xs
```

ignores the type class!

```
\xs -> head []
```

always crashes!

```
\xs -> tail xs
```

ignores the type class!

too many irrelevant results

Hoogλe+ | `Eq a => [a] → [a]` | Search

```
\xs -> head (group xs)
```

```
\xs -> init (head (group xs))
```

```
\xs -> tail (head (group xs))
```

too many irrelevant results

Hoogλe+ `Eq a => [a] → [a]` Search

```
\xs -> head (group xs)
```

```
\xs -> init (head (group xs))
```

```
\xs -> tail (head (group xs))
```

duplicate!

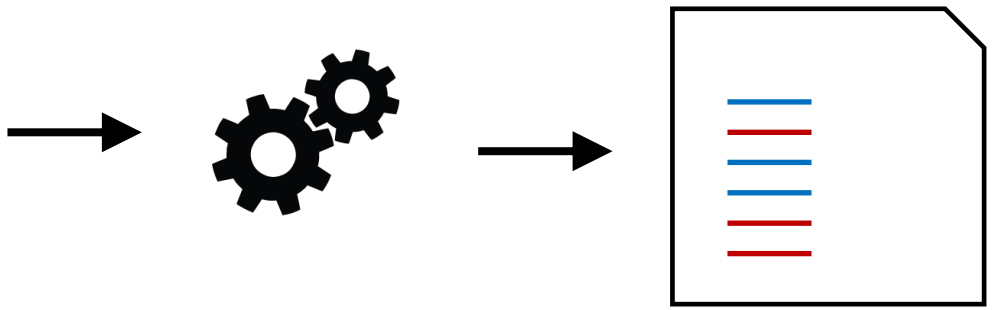
test-based filtering

type query

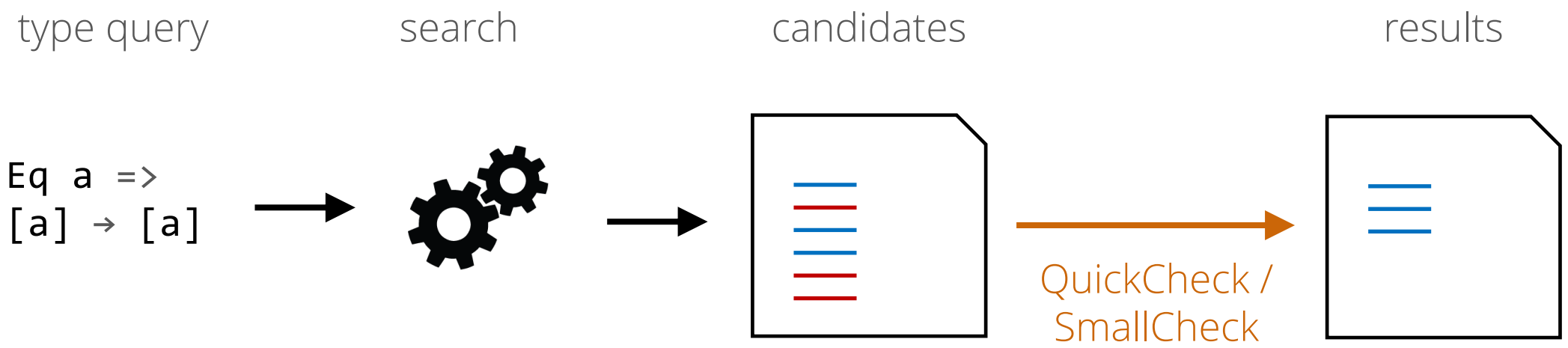
search

candidates

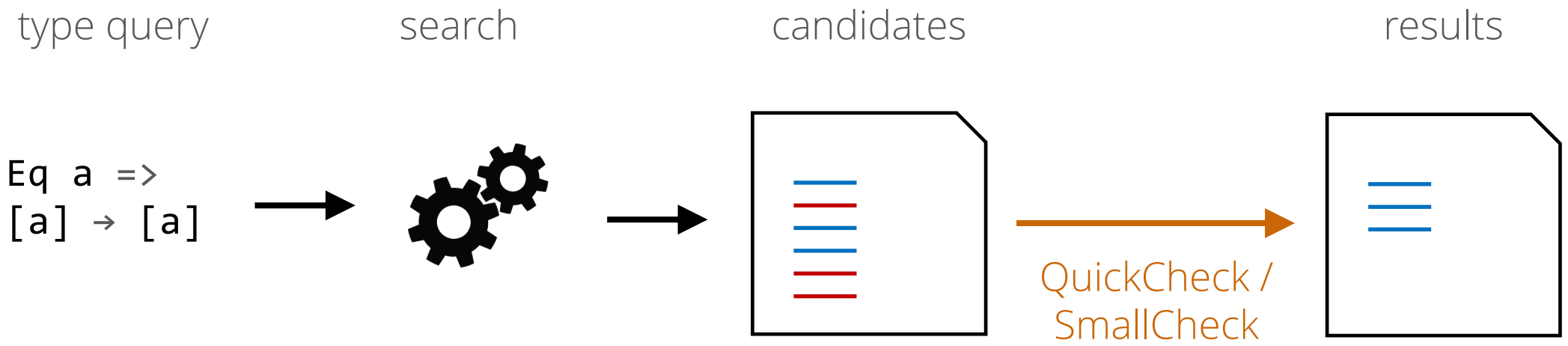
```
Eq a =>
[a] → [a]
```



test-based filtering

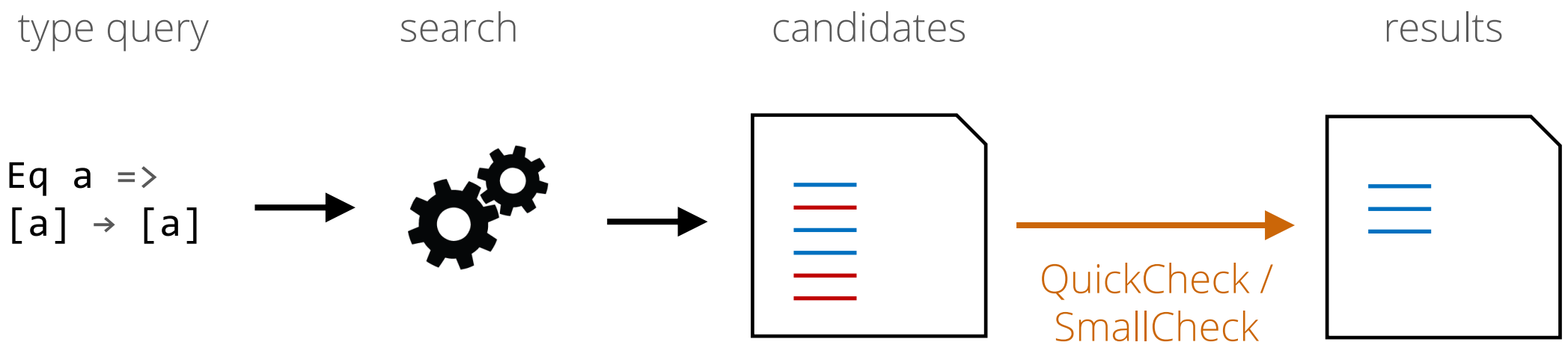


test-based filtering



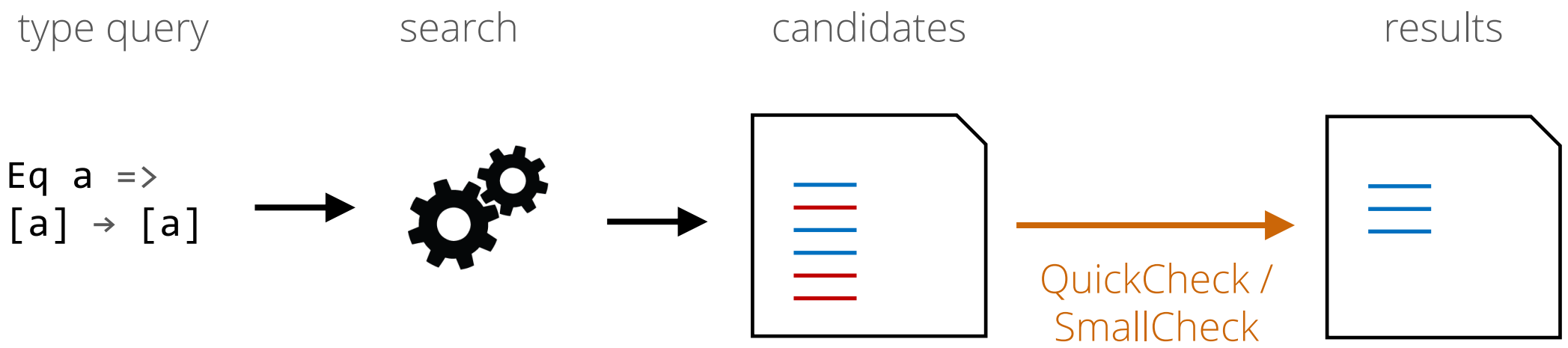
1. does it crash on all inputs?

test-based filtering



1. does it crash on all inputs?
2. is the output always the same as another candidate?

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?

comprehension

Hoogλe+ `Eq a => [a] → [a]` Search

```
\xs -> concat (group xs)
```

```
\xs -> head (group xs)
```

```
\xs -> last (group xs)
```

```
\xs -> map head (group xs)
```

comprehension

Hoogλe+ `Eq a => [a] → [a]` Search

```
\xs -> concat (group xs)
```

```
\xs -> head (group xs)
```

```
\xs -> last (group xs)
```

```
\xs -> map head (group xs)
```

how do I know what these programs do?

test-based comprehension

Hoogλe+

Eq a => [a] → [a]

Search

```
\xs -> concat (group xs)
```

```
[0,1]    -> [0,1]
[0,0]    -> [0,0]
```

```
\xs -> head (group xs)
```

```
[0,1]    -> [0]
[0,0]    -> [0,0]
```

```
\xs -> last (group xs)
```

```
[0,1]    -> [1]
[0,0]    -> [0,0]
```

```
\xs -> map head (group xs)
```

```
[0,1]    -> [0,1]
[0,0]    -> [0]
```

test-based comprehension

Hoogλe+

Eq a => [a] → [a]

Search

```
\xs -> concat (group xs)

[0,1]    -> [0,1]
[0,0]    -> [0,0]
```

```
\xs -> head (group xs)

[0,1]    -> [0]
[0,0]    -> [0,0]
```

```
\xs -> last (group xs)

[0,1]    -> [1]
[0,0]    -> [0,0]
```

```
\xs -> map head (group xs)

[0,1]    -> [0,1]
[0,0]    -> [0]
```

part III

hoogle+

Haskell types → function compositions

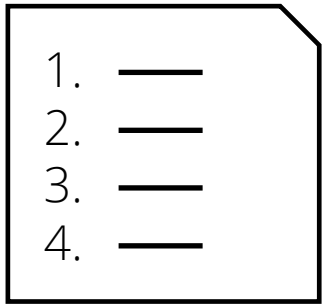
1. overcoming ambiguity
2. helping beginners with types

hoogle+

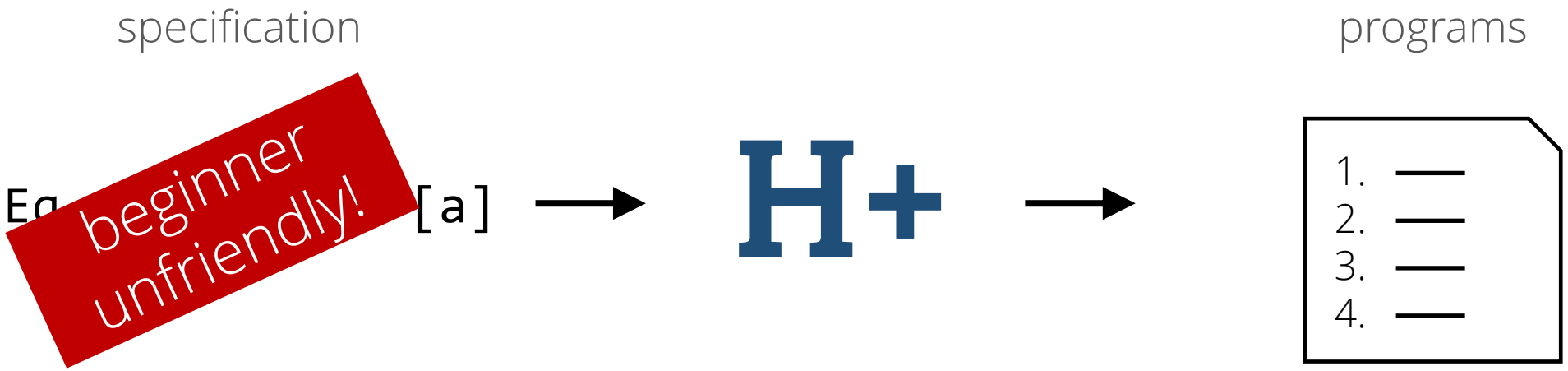
specification

programs

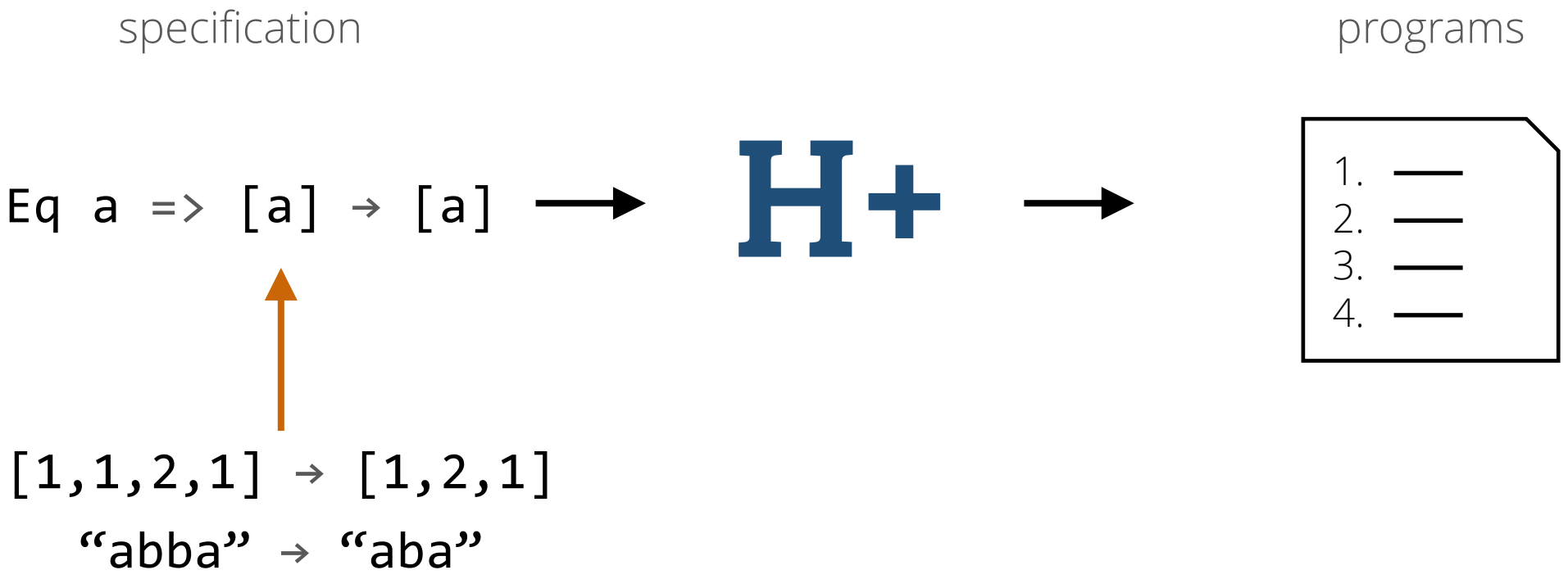
```
Eq a => [a] -> [a]
```



hoogle+



can we infer type from tests?

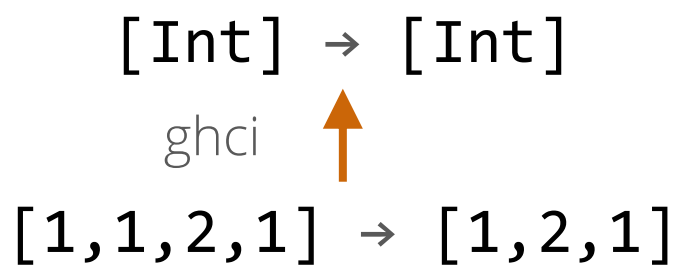


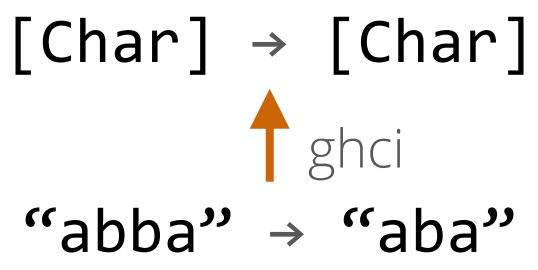
types from tests

`[1,1,2,1]` → `[1,2,1]`

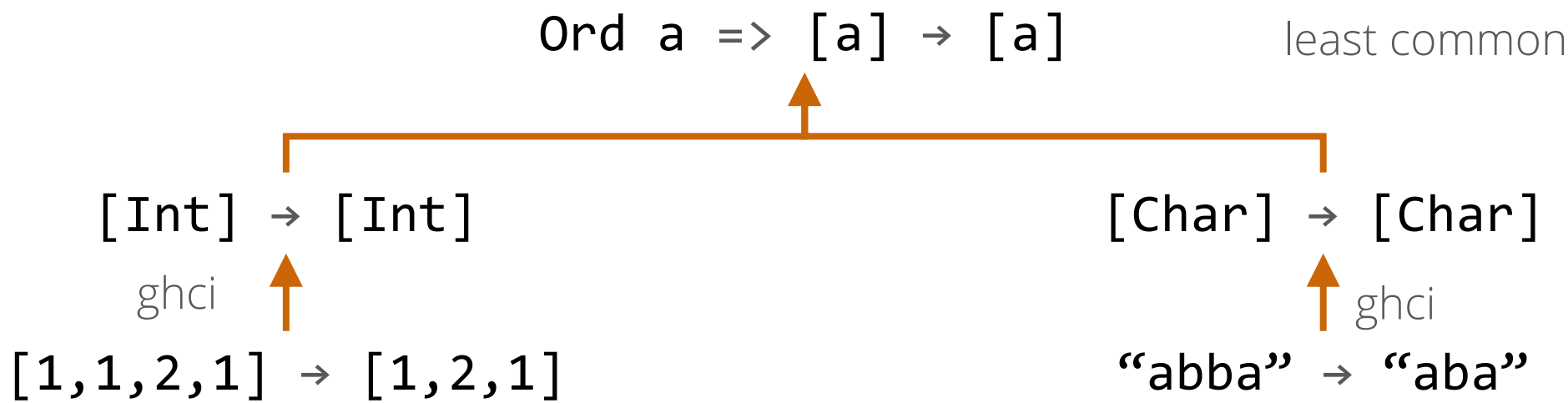
“abba” → “aba”

types from tests

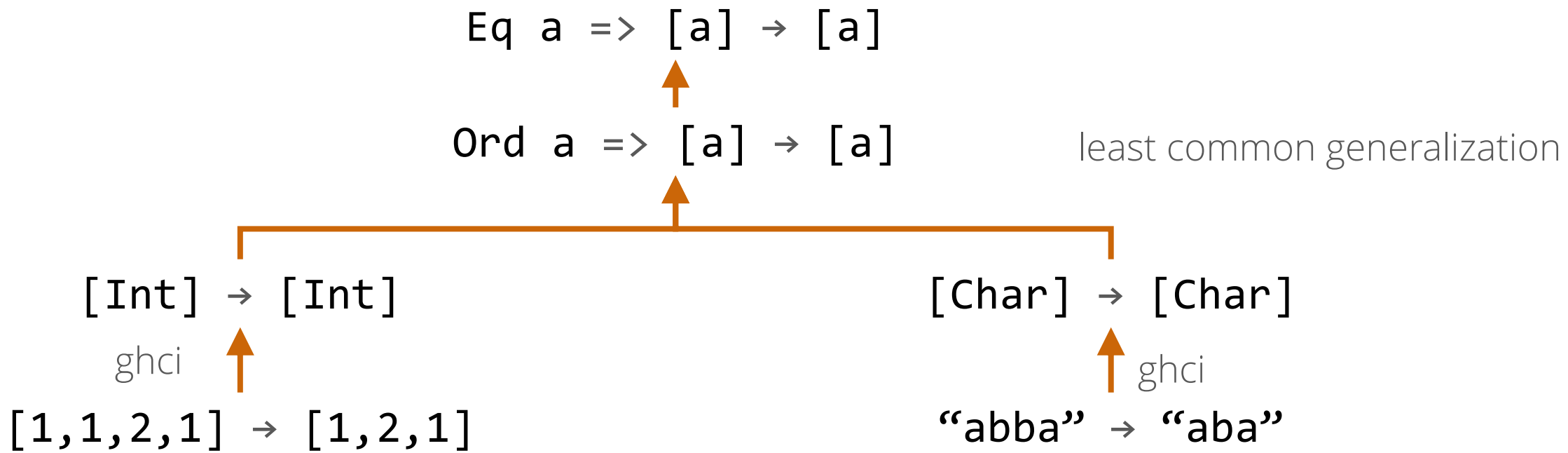




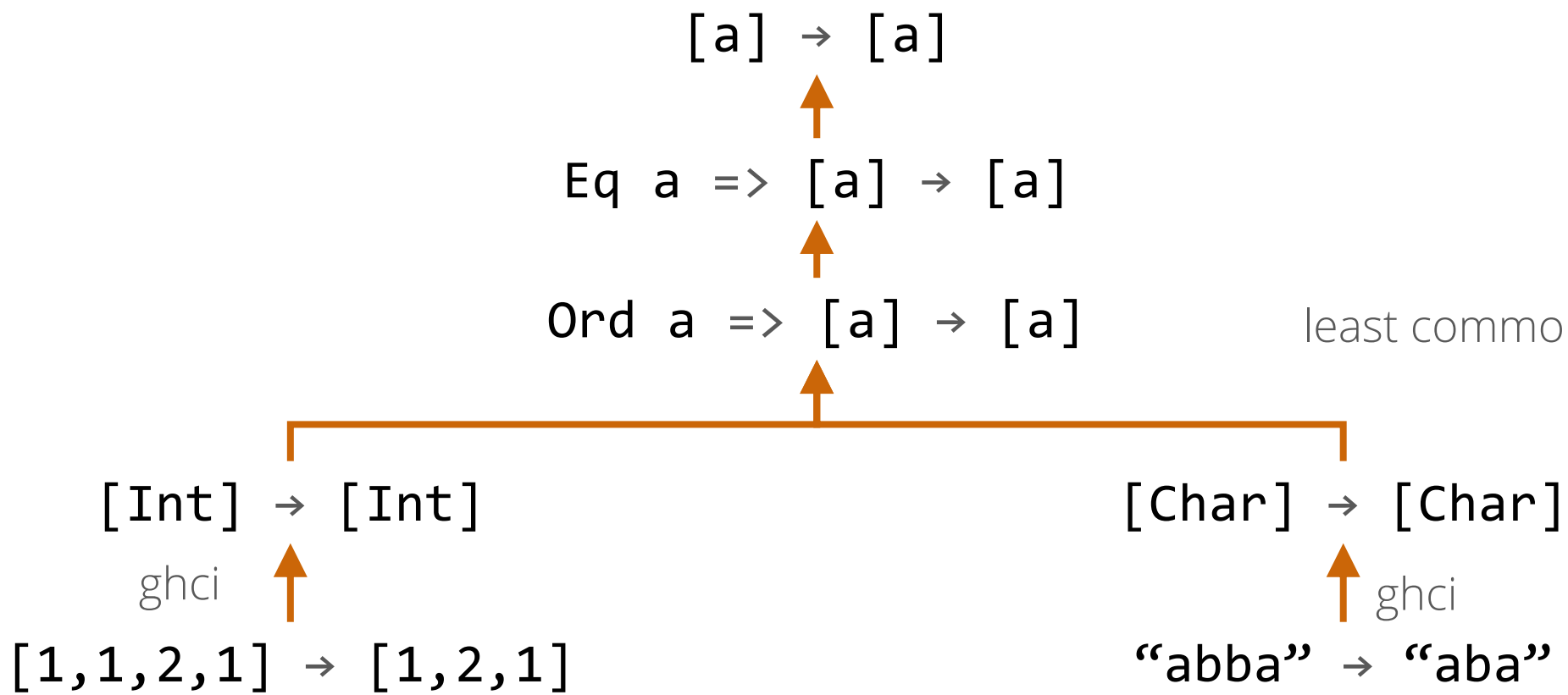
types from tests



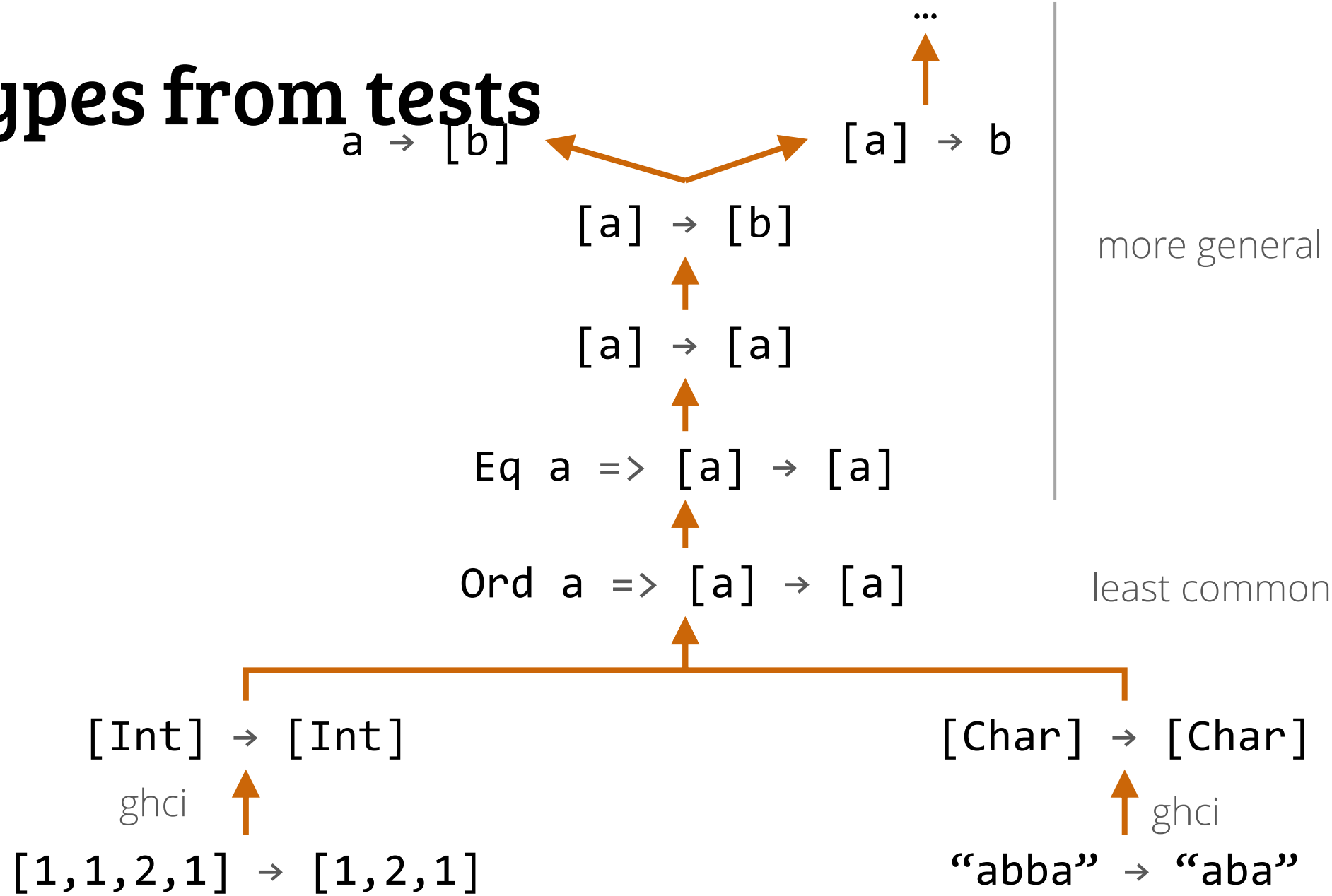
types from tests



types from tests



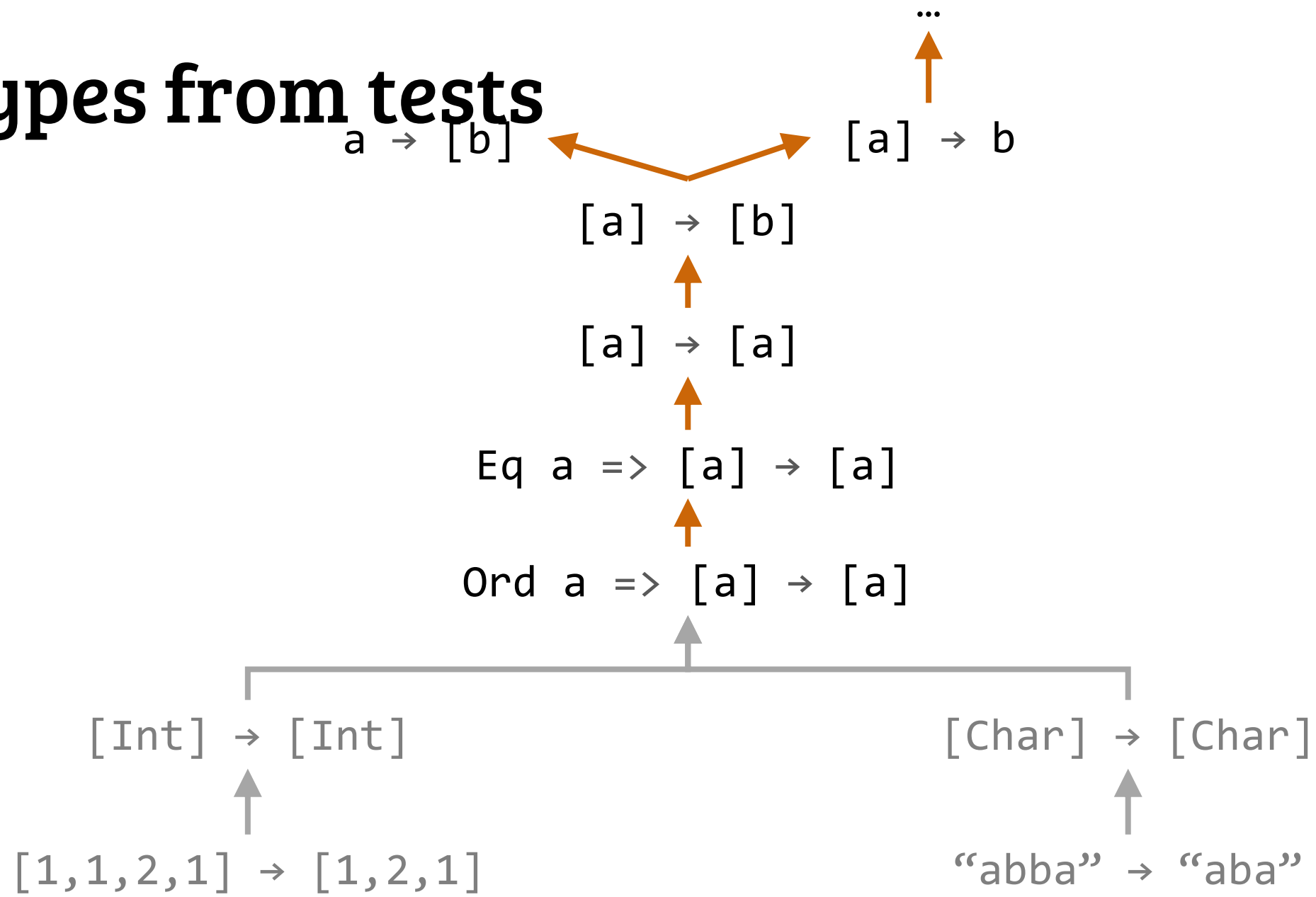
types from tests



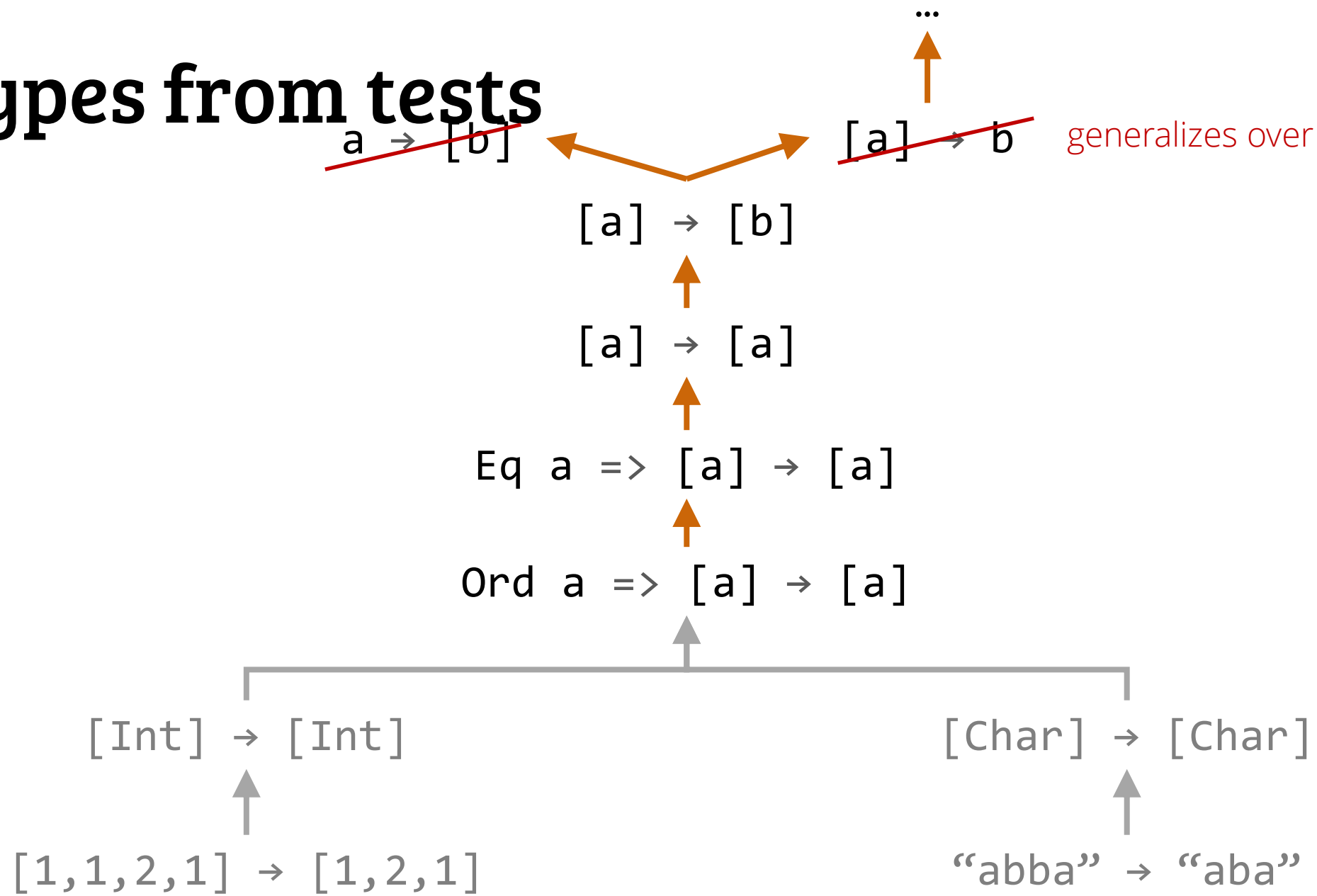
more general types

least common generalization

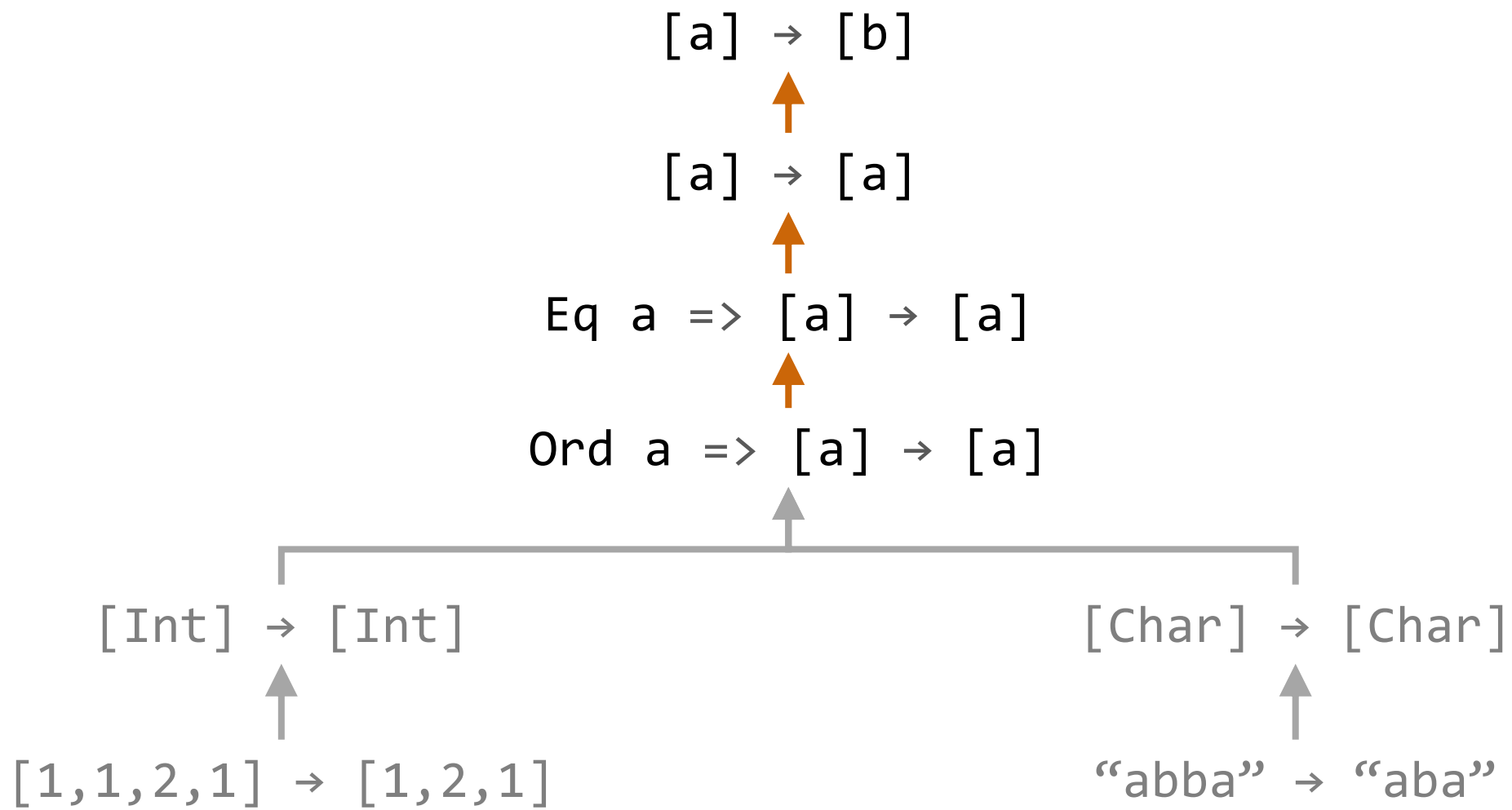
types from tests



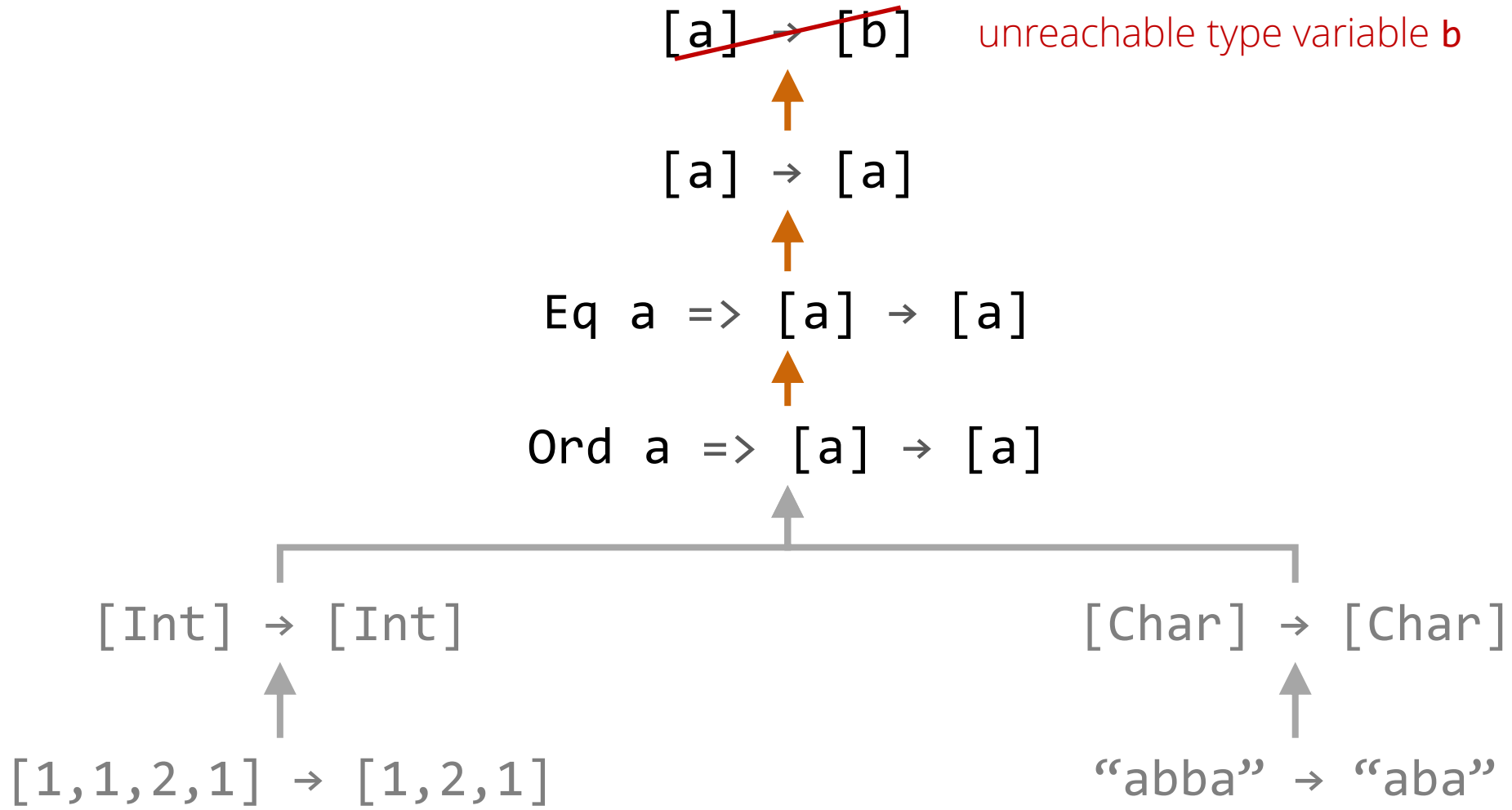
types from tests



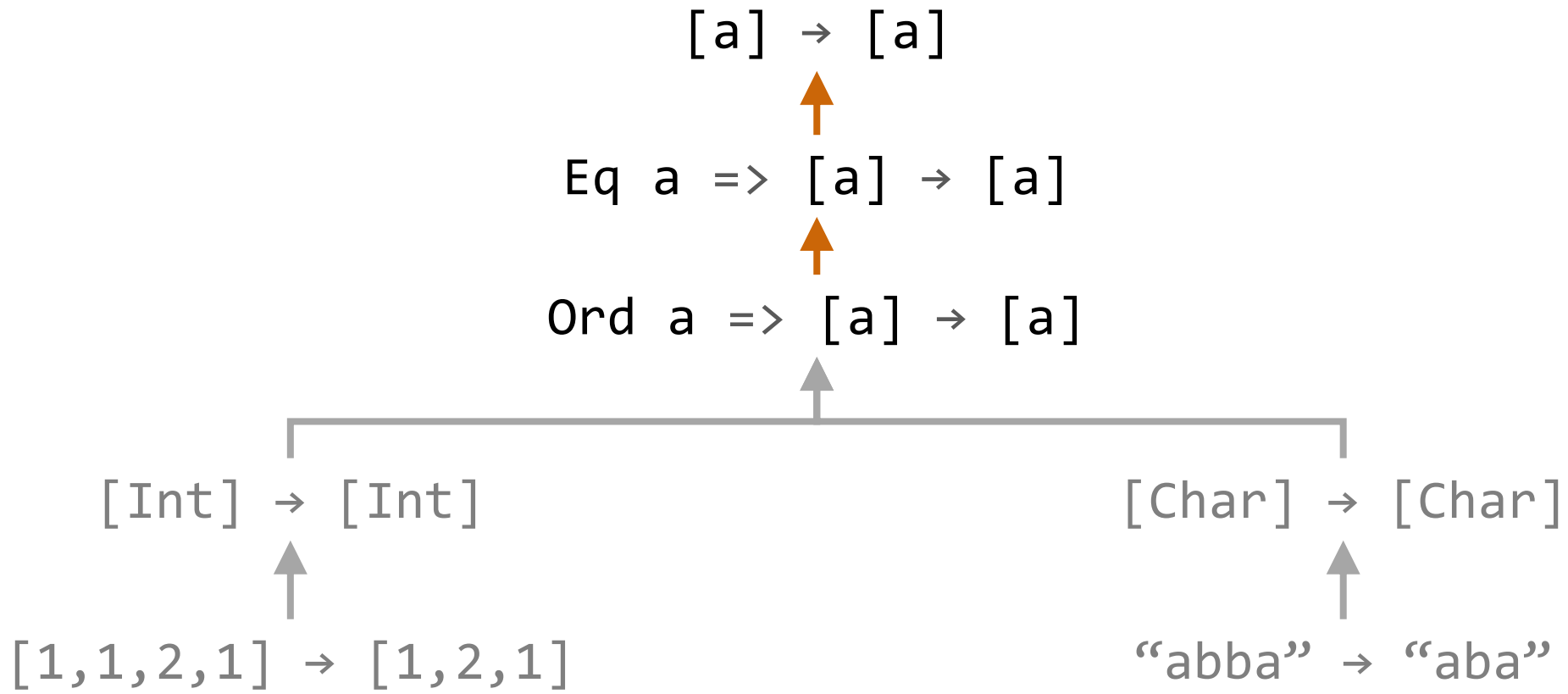
types from tests



types from tests



types from tests



types from tests

```
[a] → [a]
```

```
Eq a => [a] → [a]
```

```
Ord a => [a] → [a]
```

part II

hoogle+

user study

Haskell types → function compositions

user study

30 participants

user study

30 participants

4 tasks

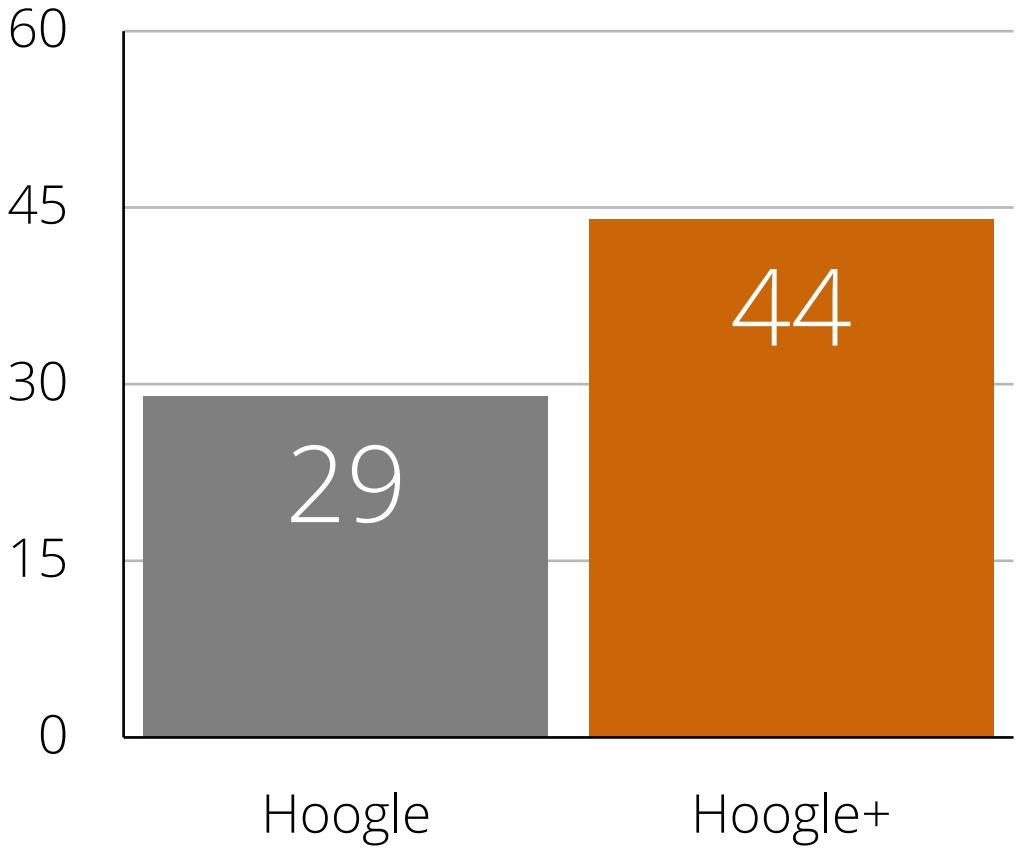
user study

30 participants

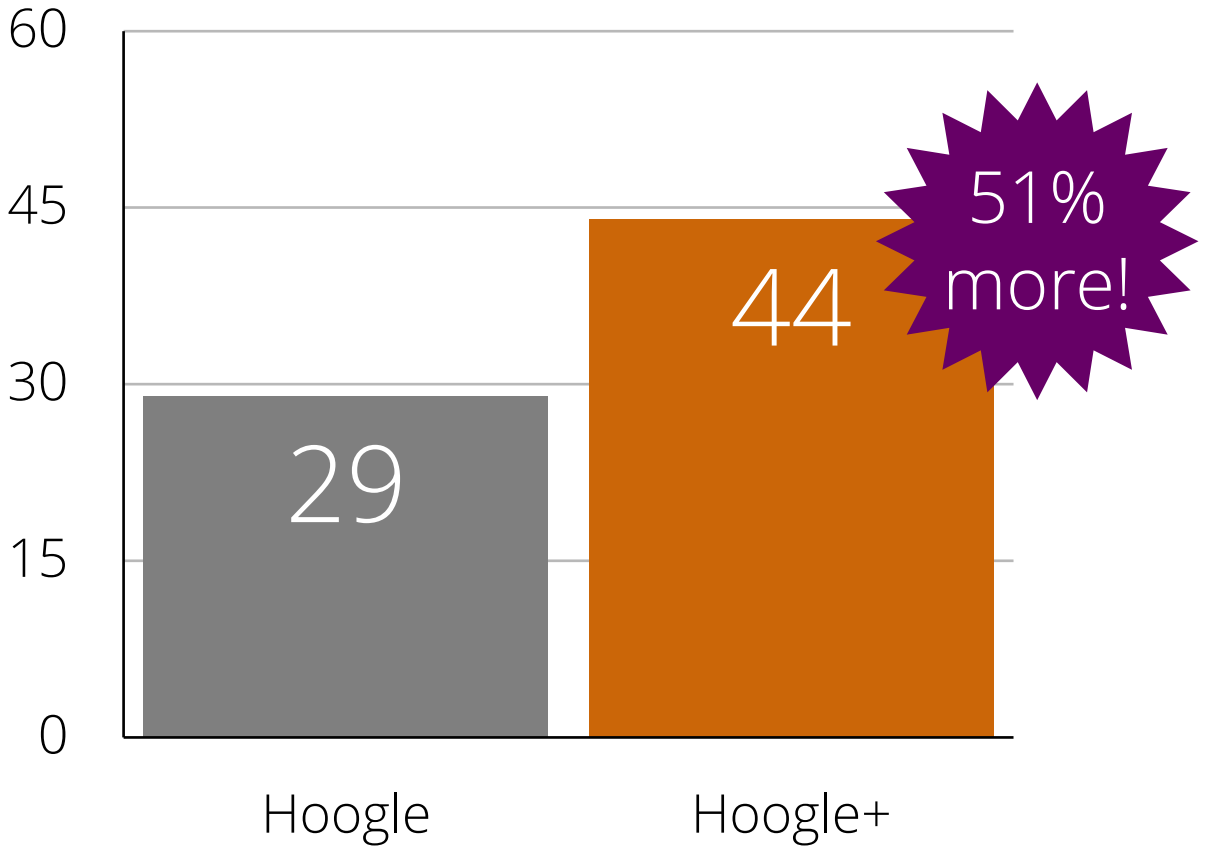
4 tasks

2 with Hoogle, then 2 with Hoolge+

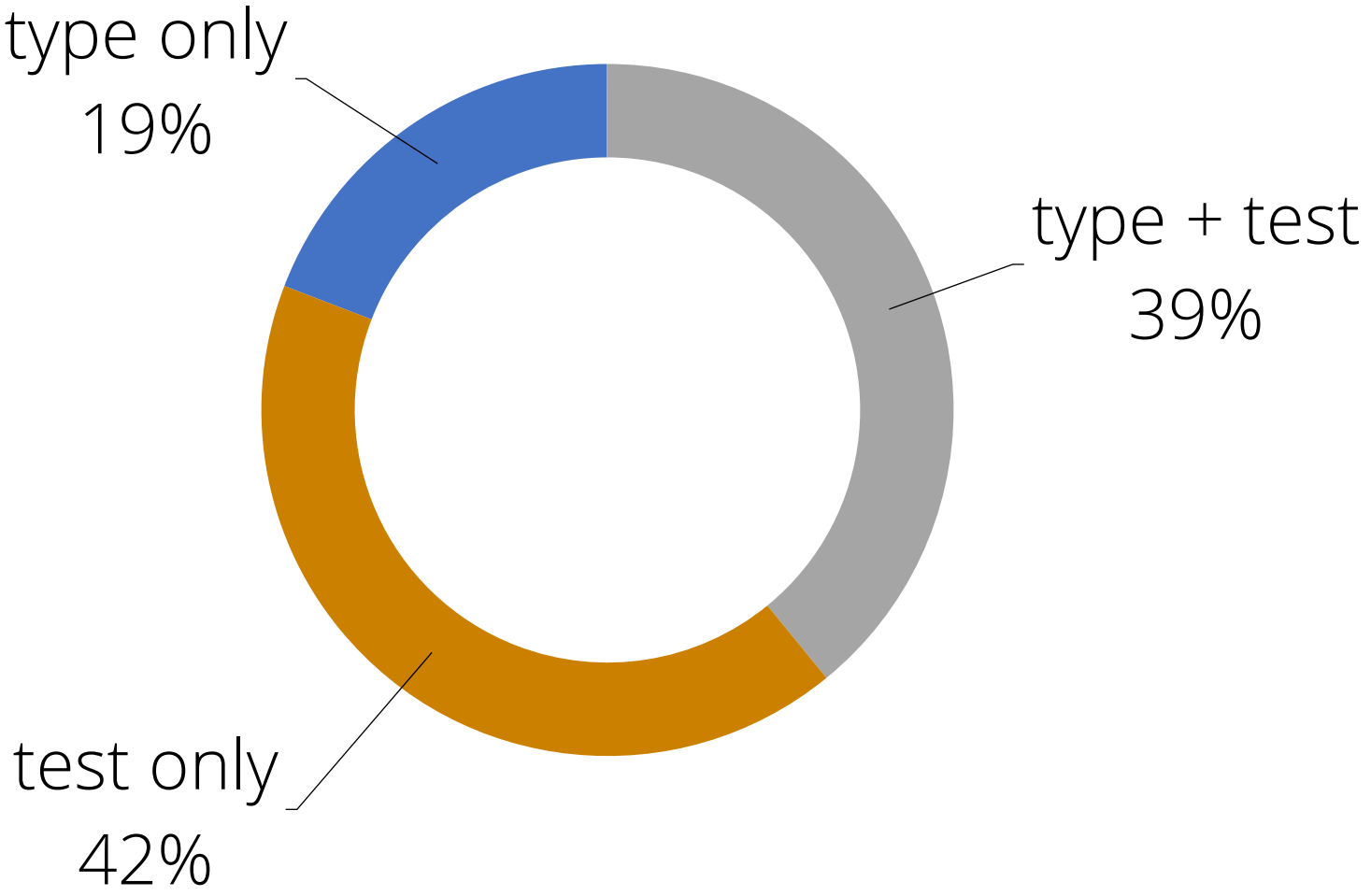
results: completed tasks



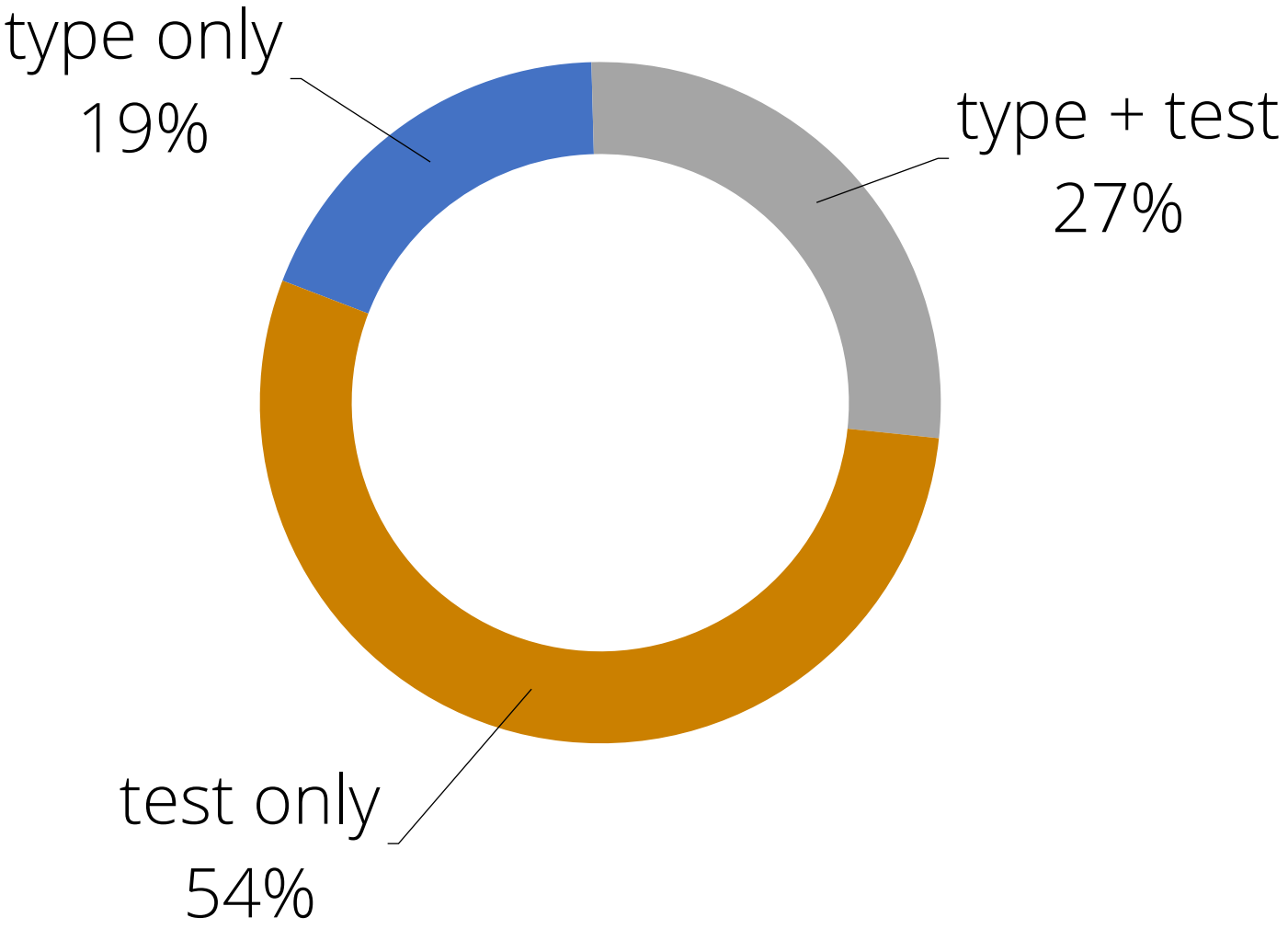
results: completed tasks



modes of specification



modes of specification: beginners



this talk

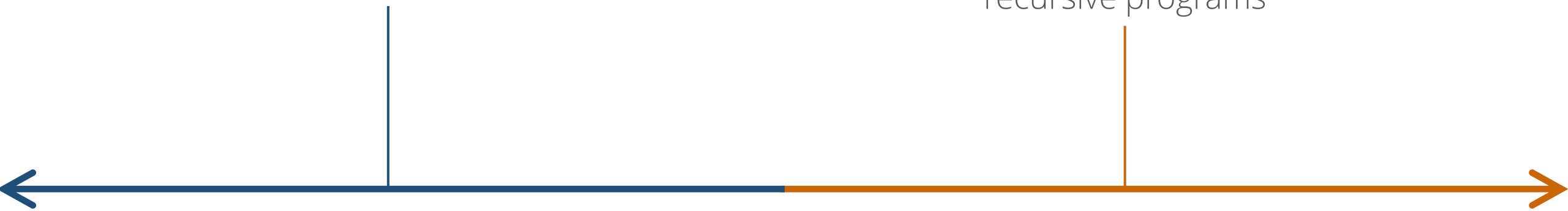
part II: hoogle+

H+

Haskell types → function compositions

part I: synquid

refinement types → recursive programs



simple types

expressive types