

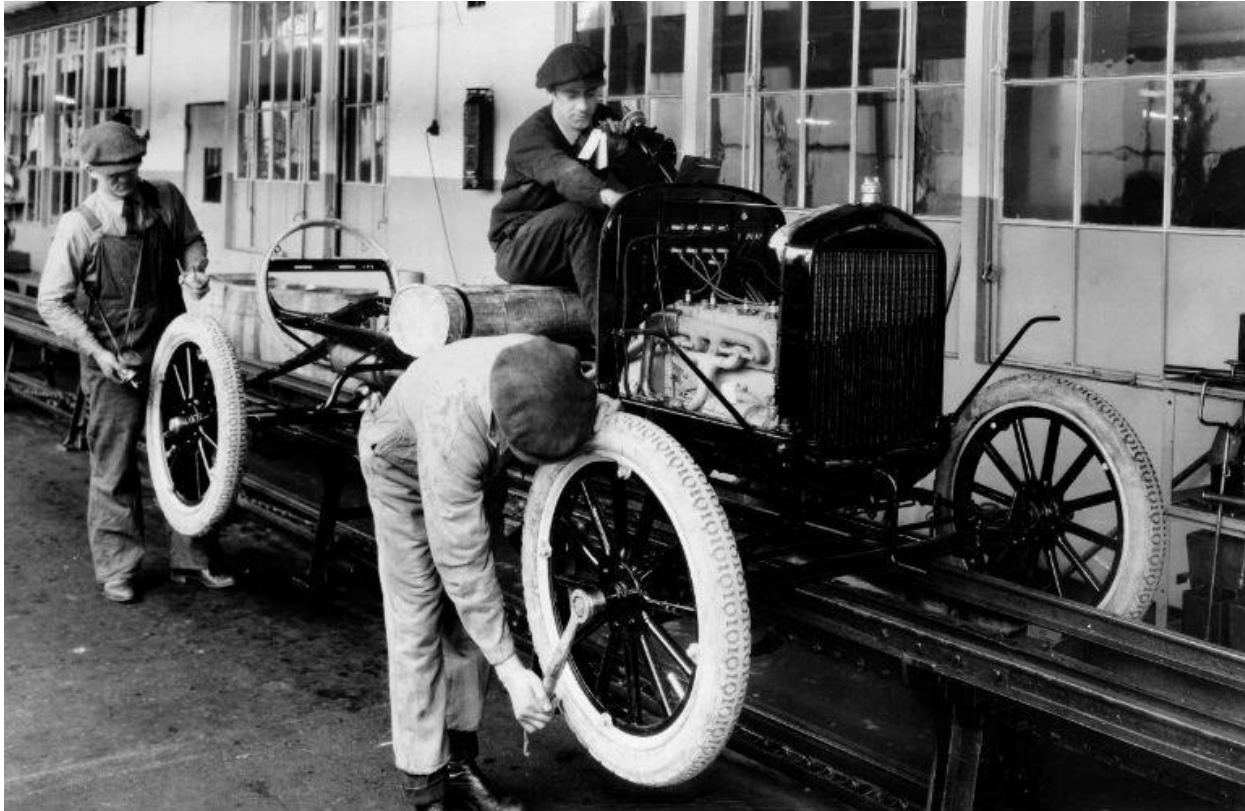
synthesizing programs from types

Nadia Polikarpova
Tutorial Fest @ POPL'20



UCSDCSE
Computer Science and Engineering

goal: automate programming



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
```

Assembly

```
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

```
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

?

future

 How to split a string in Haskell?

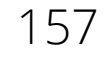
149

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

```
split ',' "my,comma,separated,list" → ["my", "comma", "separated", "list"]
```

 You can implement it like this:

157



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

▲ How to split a string in Haskell?

149

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

```
split ',' "my,comma,separated,list" → ["my", "comma", "separated", "list"]
```

wanted: automatic
programmer's assistant

▲ You can implement

157

▼

```
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

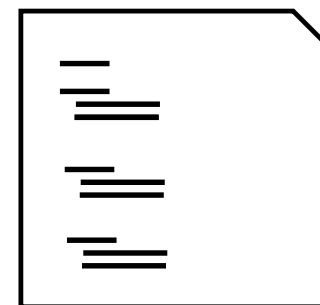
specification



search



program



program
space

program synthesis

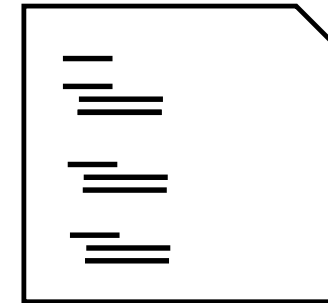
specification



search

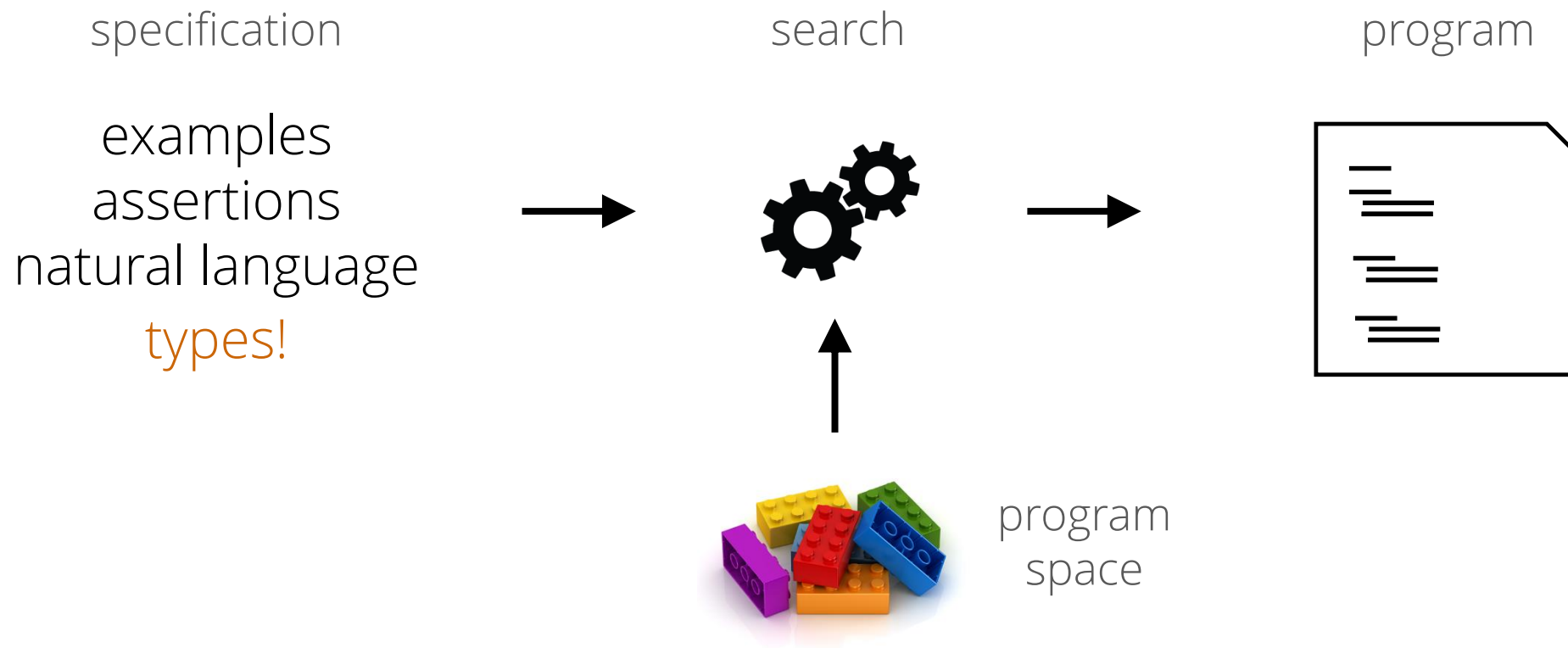


program



program
space

program synthesis



the future is (almost) here

Hoogλe

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

```
ghc Util
```

why types?

specification

`Char → String → [String]`

1. programmer-friendly

widely used for bug finding, API search, etc
more concise than examples

2. informative

more precise than natural language
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 tutorial

part I: synquid



refinement types →
recursive programs

part II: resyn



resource types →
efficient programs

part III: hoogle+

H+

Haskell types →
function compositions

simple types

expressive types

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 2016]

part I

synquid



refinement types →
recursive programs

1. types as specifications

example: insert into a sorted list

Input:

x

5

xs

1	2	7	8
---	---	---	---

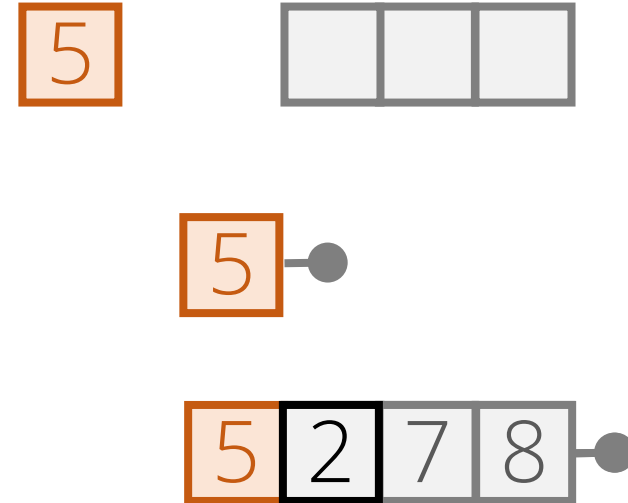
Output:

ys

1	2	5	7	8
---	---	---	---	---

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 synquid

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

$\text{Ord } a \Rightarrow$
`insert :: a → List a → List a`

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

$a \rightarrow \text{List } a$
 $\rightarrow \text{List } a$



code

`Nil`



`Nil, Cons,  $\leq$ , ...`

insert in synquid

specification

$a \rightarrow \text{list } a$

ambiguous!



code

Nil



Nil, Cons, $\leq$, ...

insert in synquid

specification

code

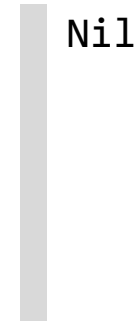
$a \rightarrow \text{list } a$

ambiguous!

**POWER TO THE
TYPES!**



$\text{Nil}, \text{Cons}, \leq, \dots$



specification for insert

Input:

x

xs : sorted list

Output:

ys : sorted list

$\text{elems } ys = \text{elems } xs \cup \{x\}$

can I write this as a type?

refinement types

$$\{ \textcolor{brown}{v}:\text{Int} \mid \frac{\textcolor{brown}{0} \leq \textcolor{brown}{v}}{\text{refinement}} \}$$

refinement types

$$\{ \textcolor{brown}{v}:\text{Int} \mid \textcolor{brown}{0} \leq \textcolor{brown}{v} \}$$


natural numbers

refinement types

List { $v:\text{Int}$ | $0 \leq v$ }



lists of nats

demo: insert

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

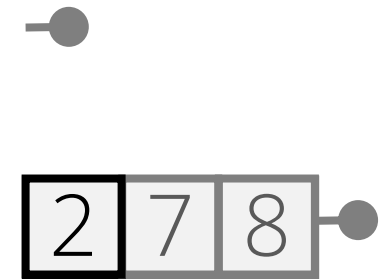
refinement types: sorted lists

```
data SList a where
```

```
Nil :: List 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: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 U {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)
```

exercise: sets

<http://comcom.csail.mit.edu/demos/#2-sets>

part I

synquid



refinement types →
recursive programs

1. types as specifications
2. type-directed search

insert in synquid

specification

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



code

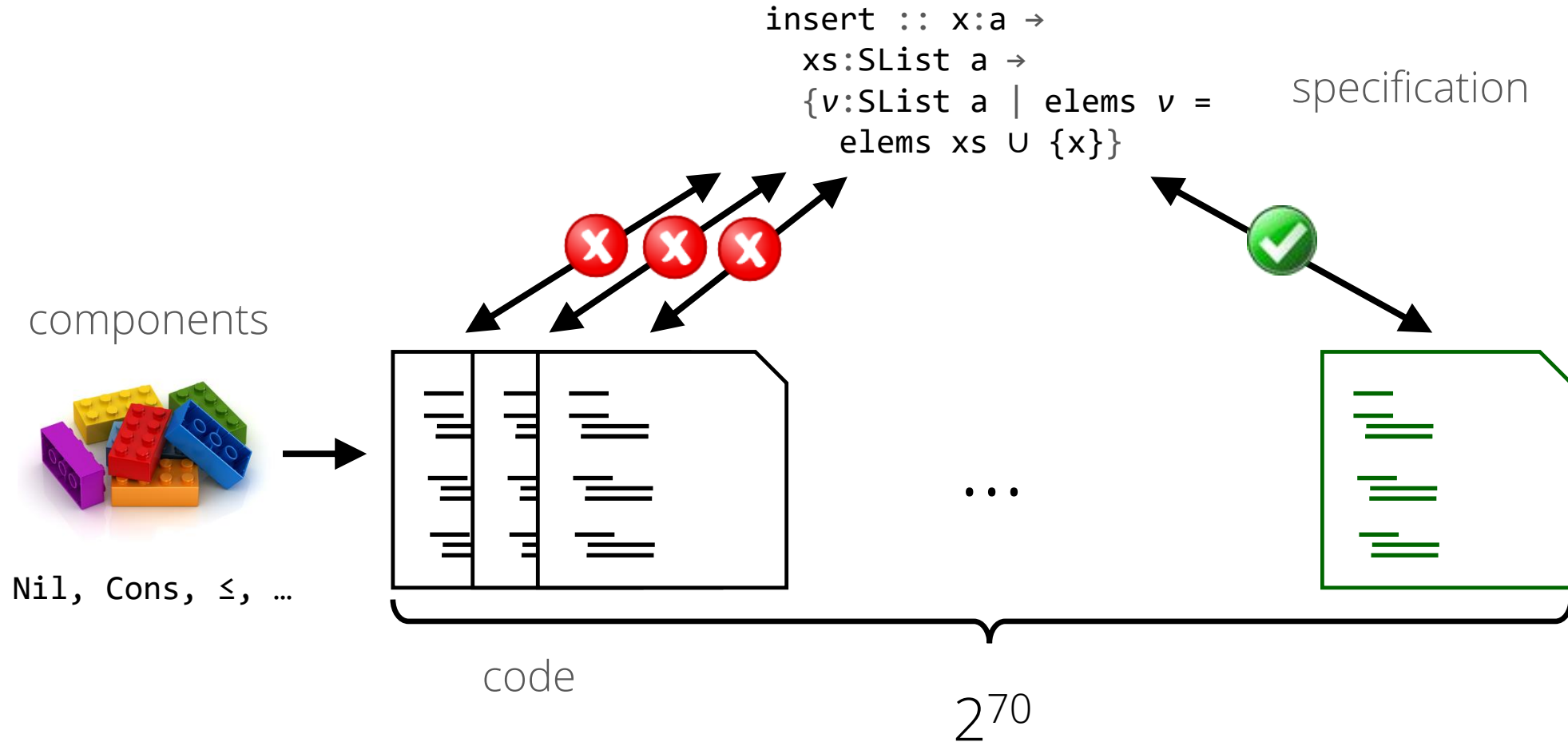
```
match xs with  
Nil → Cons x Nil  
Cons h t →  
  if x ≤ h  
  then Cons x xs  
  else Cons h (insert x t)
```

synthesis as search

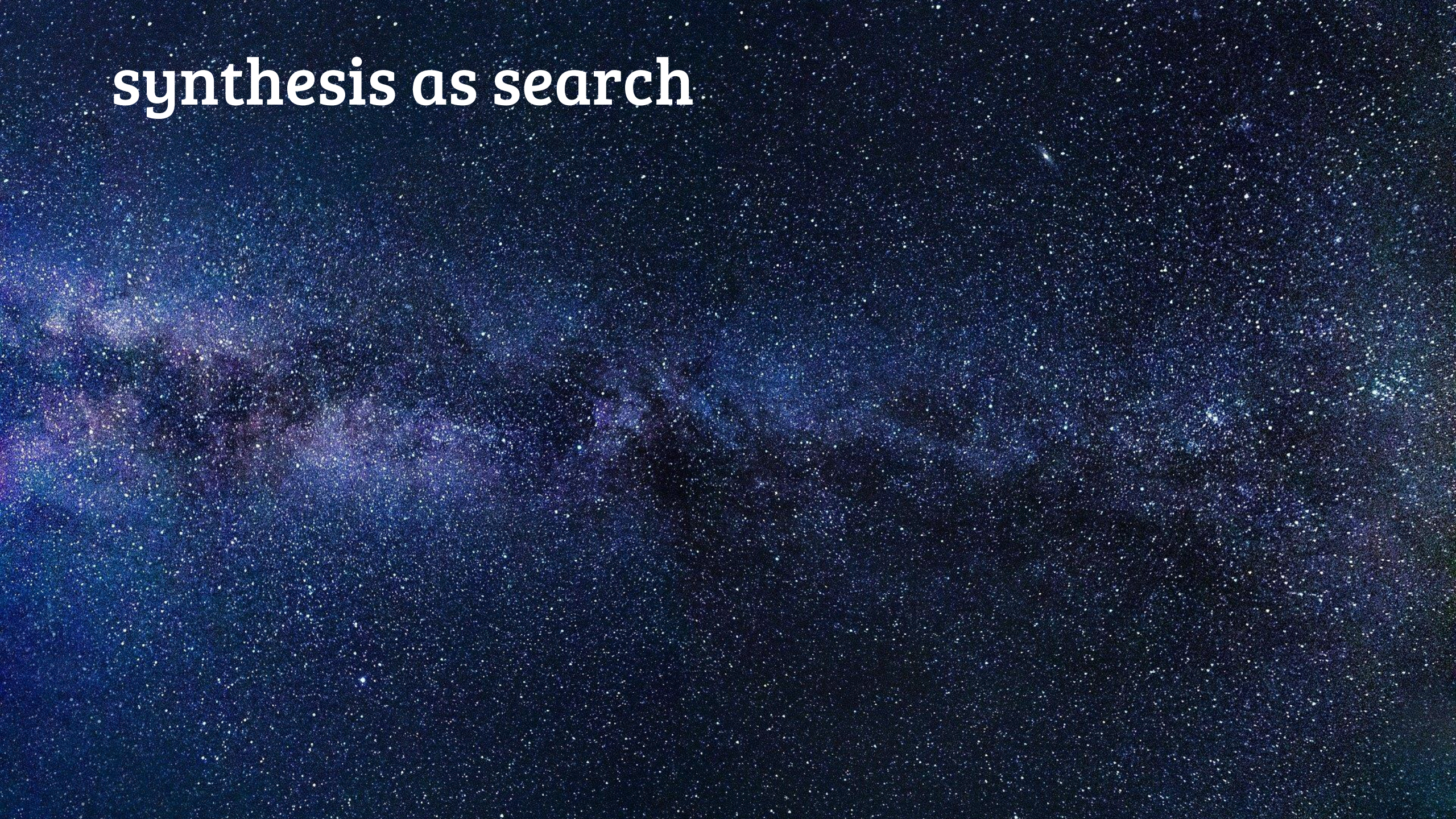
specification

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

synthesis as search



synthesis as search

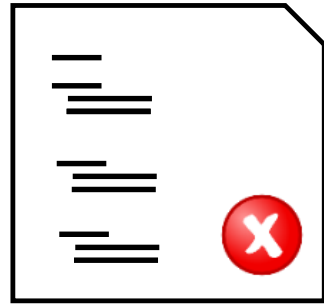




key idea: reject hopeless programs early
(during construction)

rejecting hopeless programs

previous work*:
bottom-up type checking



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

specification

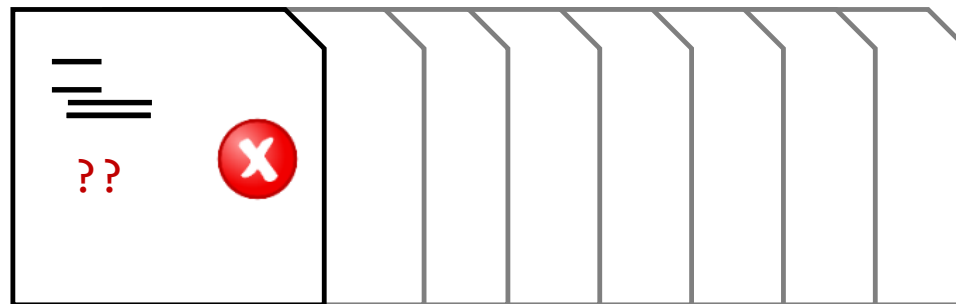
*Rondon, Kawaguchi, Jhala: *Liquid types*. [PLDI 2008]

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

```
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

$x:a \rightarrow xs:SList\ a \rightarrow$
 $\{v:SList\ a \mid elems\ v = elems\ xs \cup \{x\}\}$



`insert x xs = ??`

`match xs with`

`Nil → Nil`

`Cons h t →`



2^{50}



hopeless:
output must
always contain x!

top-down type checking

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



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



Constraints:

$\forall x: \{\} = \{\} \cup \{x\}$

SMT solver: INVALID!

top-down type checking: the hard part

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



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



hopeless:
cannot guarantee
output is sorted!

top-down type checking: the hard part

SList a



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

```
Cons :: h:τ →  
      t: SList {τ | h ≤ v} →  
      SList τ
```

top-down type checking: the hard part

`SList {v:a | h ≤ v}`



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

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

top-down type checking: the hard part

$\{v:a \mid h \leq v\}$



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

top-down type checking: the hard part

$\{v:a \mid h \leq v\}$



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

Constraints:
 $\forall x, h: h \leq x$

SMT solver: INVALID!

condition abduction

$\{v:a \mid h \leq v\}$



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

Constraints:
 $\forall x, h: h \leq x$

SMT solver: INVALID!

insert in synquid

specification

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



components



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: negation normal form

<http://comcom.csail.mit.edu/demos/#3-nnf>

this tutorial

part I: synquid



refinement types →
recursive programs

part II: resyn



resource types →
efficient programs

part III: hoogle+

H+

Haskell types →
function compositions

simple types

expressive types

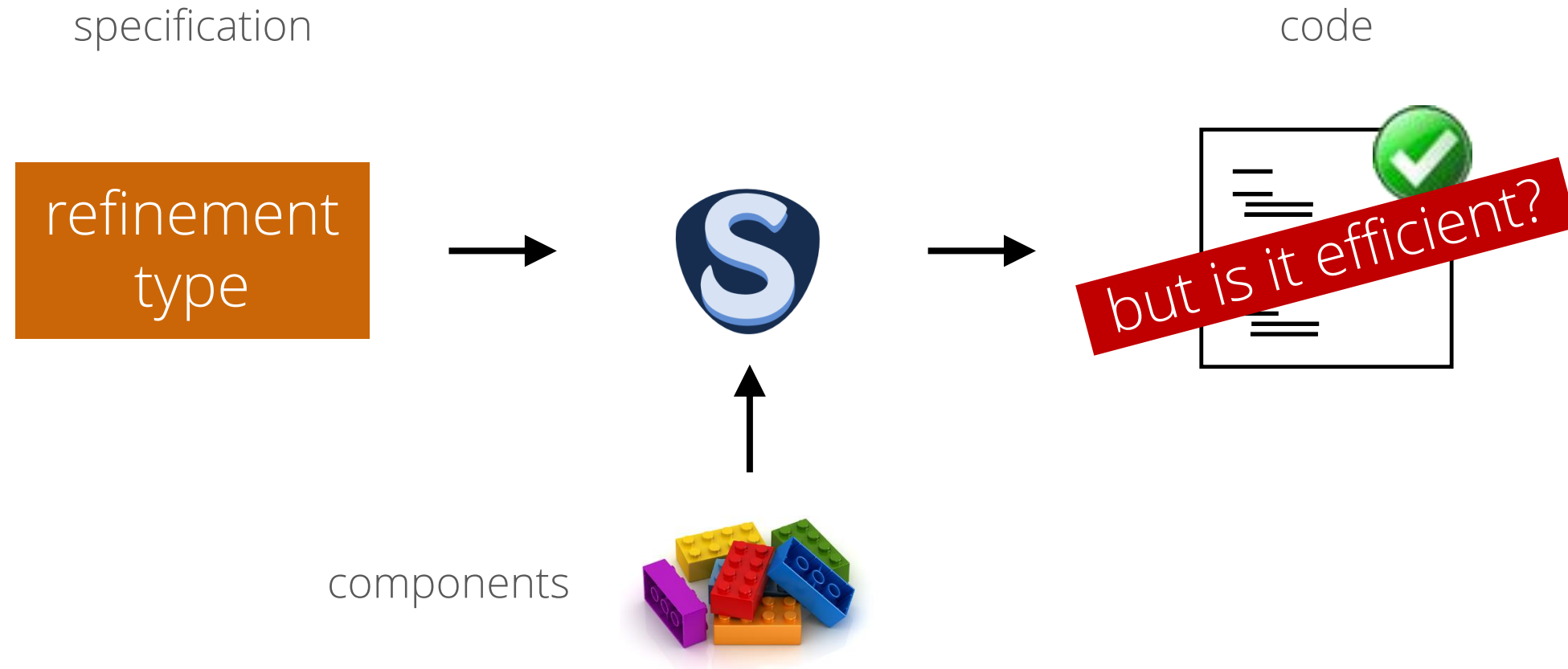
part II

resyn



resource types →
efficient programs

synquid: limitation



demo: inefficient sets

<http://comcom.csail.mit.edu/demos/#4-sets-slow>

$$O(|xs| \cdot |ys|)$$

```
intersect xs ys =  
  match xs with  
    Nil → Nil  
    Cons x xt →  
      if !(member x ys)  
      then intersect xt ys  
      else Cons x (intersect xt ys)
```

$$O(|xs| + |ys|)$$

```
intersect = λ xs. λ ys.  
  match xs with  
    Nil → Nil  
    Cons x xt →  
      match ys with  
        Nil → Nil  
        Cons y yt →  
          if x < y  
          then intersect xt ys  
          else if y < x  
          then intersect xs yt  
          else Cons x (intersect xs ys)
```

part II

resyn



resource types →
efficient programs

1. resource types
2. resource-guided synthesis

Knoth, Wang, Polikarpova, Hoffman: *Resource-Guided Program Synthesis*. [PLDI 2019]

part II

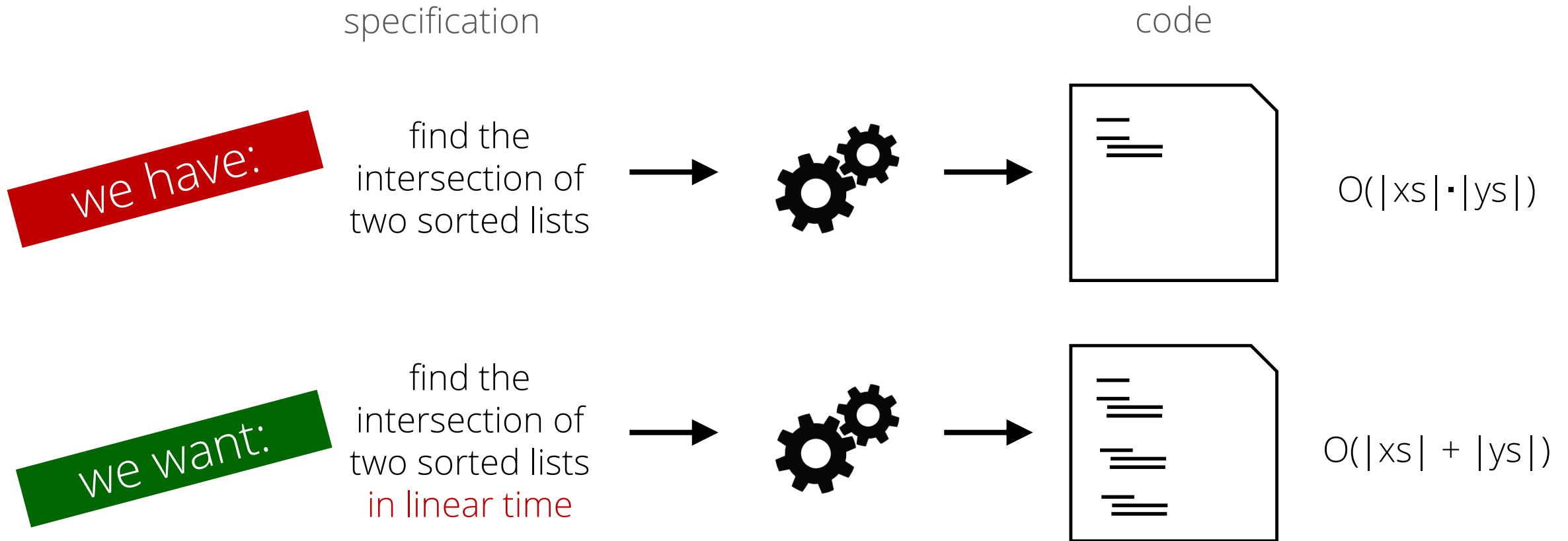
resyn



resource types →
efficient programs

1. resource types

synthesizing efficient programs

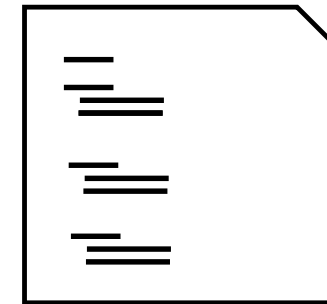


synthesizing efficient programs

specification

code

type?
find the
intersection of
two sorted lists
in linear time



$O(|xs| + |ys|)$

synthesizing efficient programs

specification

code

find a
type?
merging of
two sorted lists
in linear time



**MORE POWER TO
THE TYPES!**



re² = refinements + resources

$$\{ v:\text{Int} \mid \underbrace{0 \leq v}_{\text{refinement}} \mid \underbrace{1}_{\text{potential}} \}$$

re² = refinements + resources

List { v: Int | 0 ≤ v | 1 }



lists of nats with length units of potential

re² type for intersect

```
intersect :: xs:SList {a || 1} →  
           ys:SList {a || 1} →  
           {v:List a | elems v = elems xs ∩ elems ys}
```

demo: efficient sets

<http://comcom.csail.mit.edu/demos/#4-sets-fast>

part II

resyn



resource types →
efficient programs

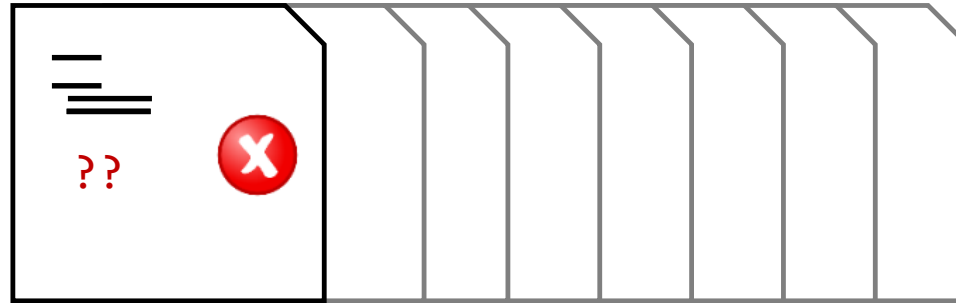
1. resource types
2. resource-guided synthesis

rejecting hopeless programs

```
intersect :: xs:SList {a || 1} →  
           ys:SList {a || 1} →  
           {v:List a | elems v = elems xs ∩ elems ys}
```

specification

top-down type checking:
now with resources!



top-down type checking

```
intersect :: xs:SList {a|1} → ys:SList {a|1} → {List a|...}
```



```
intersect xs ys =  
  match xs with  
    Nil → Nil  
    Cons x xt →  
      if !(member x ys)  
      then intersect xt ys  
      else ??
```

top-down type checking

```
intersect :: xs:SList {a||1} → ys:SList {a||1} → {List a|...}
```



```
intersect xs ys =  
  match xs with  
    Nil → Nil  
    Cons x xt →  
      if !(member x ys)  
        then intersect xt ys  
        else ??
```



hopeless:
not enough
potential in ys for
both calls!

top-down type checking

```
intersect :: xs:SList {a||1} → ys:SList {a||1} → {List a|...}
```



```
intersect xs ys =  
  match xs with  
    Nil → Nil  
    Cons x xt →  
      if !(member x ys)  
      then intersect xt ys  
      else ??
```

top-down type checking

`intersect :: xs:SList {a||1} → ys:SList {a||1} → {List a|...}`



```
intersect xs ys =  
  match xs with  
    Nil → Nil  
    Cons x xt →  
      if !(member x (ys :: SList {a||p}))  
      then intersect xt (ys :: SList {a||q})  
      else ??
```

top-down type checking

`intersect :: xs:SList {a||1} → ys:SList {a||1} → {List a|...}`



```
intersect xs ys =  
  match xs with  
    Nil → Nil  
    Cons x xt →  
      if !(member x (ys :: SList {a||p}))  
      then intersect xt (ys :: SList {a||q})  
      else ??
```

1. total potential must be partitioned into two uses:

Constraints: $\exists p, q:$

$$\mathbf{1} = p + q$$

top-down type checking

`intersect :: xs:SList {a||1} → ys:SList {a||1} → {List a|...}`



`intersect xs ys =`

`match xs with`

`Nil → Nil`

`Cons x xt →`

`if !(member x (ys :: SList {a||p}))`

`then intersect xt (ys :: SList {a||q})`

`else ??`

`member ::`

`y: a → ys: SList {a||1}`

`→ {List a |...}`

2. p must be enough to call
member:

Constraints: $\exists p, q:$

$$\mathbf{1} = p + q$$

$$p \geq \mathbf{1}$$

top-down type checking

`intersect :: xs:SList {a||1} → ys:SList {a||1} → {List a|...}`



`intersect xs ys =`

`match xs with`

`Nil → Nil`

`Cons x xt →`

`if !(member x (ys :: SList {a||p}))`

`then intersect xt (ys :: SList {a||q})`

`else ??`

3. q must be enough to call
intersect:

Constraints: $\exists p, q:$

$$\mathbf{1} = p + q$$

$$p \geq \mathbf{1}$$

$$q \geq \mathbf{1}$$

top-down type checking

`intersect :: xs:SList {a||1} → ys:SList {a||1} → {List a|...}`



`intersect xs ys =
 match xs with
 Nil → Nil
 Cons x xt →
 if !(member x (ys :: SList {a||p}))
 then intersect xt (ys :: SList {a||q})
 else ??`

Constraints: $\exists p, q:$

$$1 = p + q$$

$$p \geq 1$$

$$q \geq 1$$

SMT solver: UNSAT!



exercise: compress

<http://comcom.csail.mit.edu/demos/#5-compress>

this tutorial

part I: synquid



refinement types →
recursive programs

part II: resyn



resource types →
efficient programs

part III: hoogle+

H+

Haskell types →
function compositions

simple types

expressive types

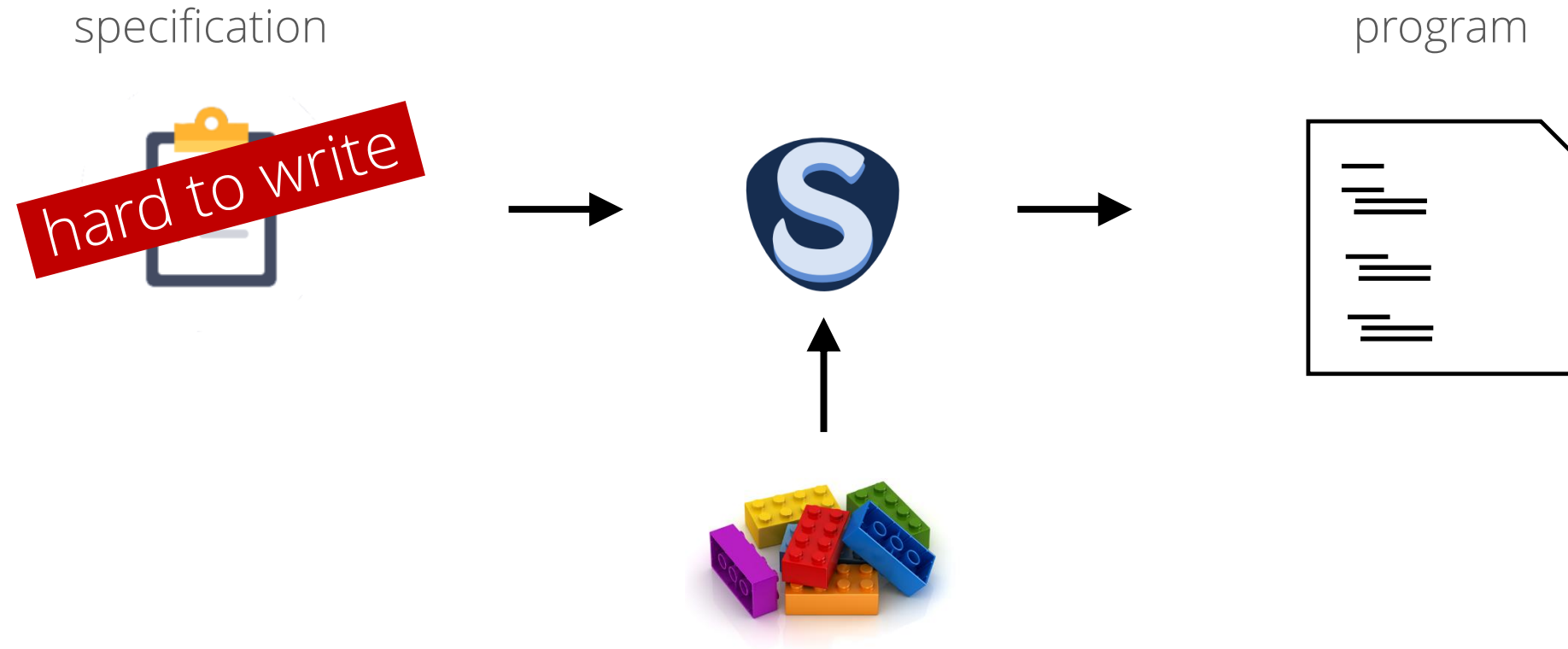
part III

hoogle+

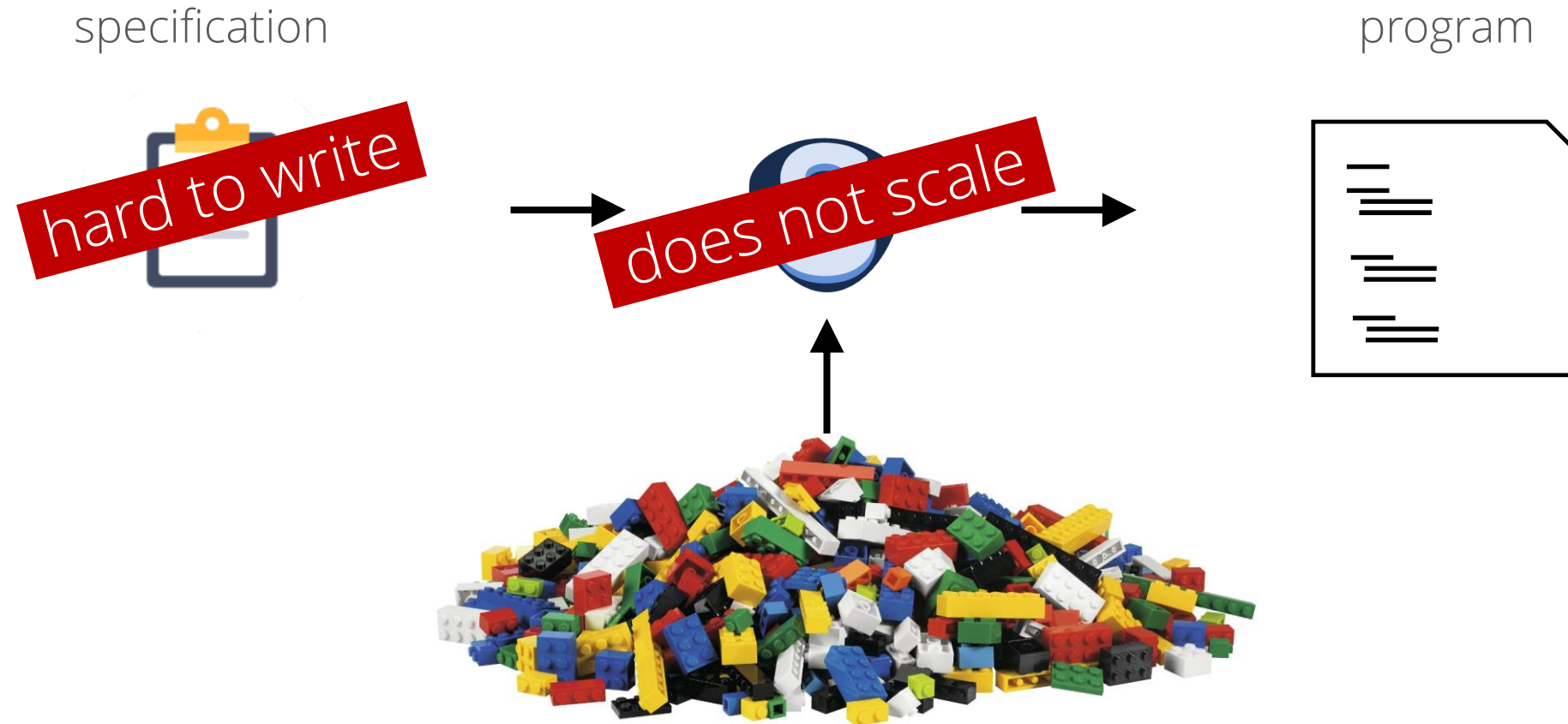


Haskell types →
function compositions

synquid/resyn: limitations



synquid/resyn: limitations



part III

hoogle+

H+

Haskell types →
function compositions

1. ~~handling large libraries~~
POPL talk on Wed at 11:45

2. overcoming ambiguity

Guo, James, Justo, Zhou, Wang, Jhala, Polikarpova:
Program Synthesis by Type-Guided Abstraction Refinement.
[POPL 2020]

inspiration: hoogle

Hoogλe

```
map :: (a -> b) -> [a] -> [b]
```

Data.List

`map f xs` is the list obtained by applying `f` to each element of `xs`, i.e.,

$$\text{map } f \text{ [x1, x2, ..., xn]} == [f \text{ x1}, f \text{ x2}, \dots, f \text{ xn}]$$

hoogle needs synthesis!

Hoogλe

(a -> Maybe b) -> [a] -> Int

Search

No results found!

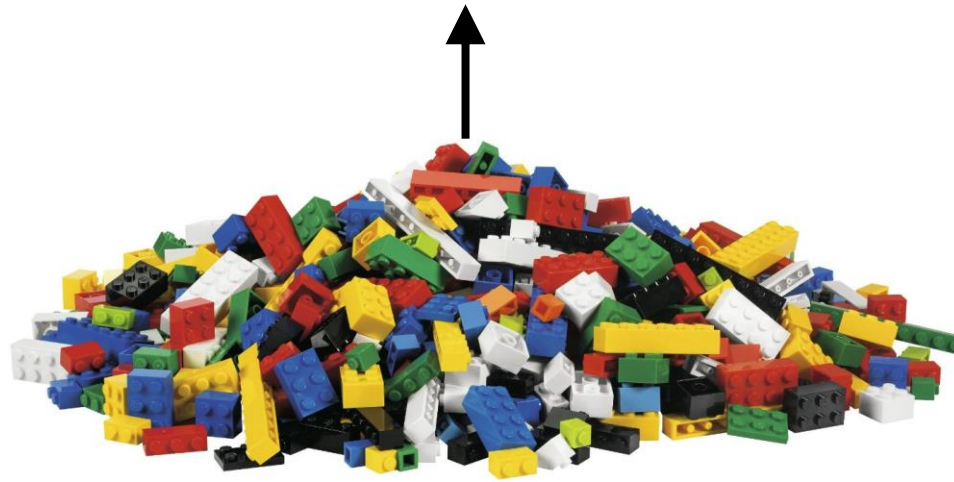
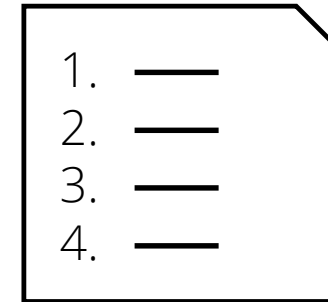
hoogle+

specification

```
(a -> Maybe b)  
  -> [a] -> Int
```

→ **H+** →

programs



Haskell
libraries

demo: count just

http://goto.ucsd.edu/hoogle_plus

exercise: apply n times

http://goto.ucsd.edu/hoogle_plus

future work: quality of result

Hoogλe+

[]



xs



take n xs

drop n xs

ignore arguments
of the query!

future work: quality of result

Hoogλe+

take n xs

drop n xs

take n (init xs)



drop n (tail xs)



too similar!

future work: quality of result

Hoogλe+

```
take n xs
```

```
drop n xs
```

```
concat (replicate n xs)
```

```
repeat (xs !! n)
```

how do I know
what these
programs do?

future work: comprehension

Hoogλe+

`n: Int -> xs: [a] -> [a]`

Search

take n xs

2 [1,2,3] -> [1,2]

3 [1] -> [1]

drop n xs

2 [1,2,3] -> [3]

3 [1] -> []

concat (replicate n xs)



2 [1,2,3] -> [1,2,3,1,2,3]

3 [1] -> [1,1,1]

repeat (xs !! n)

2 [1,2,3] -> [3,3,3,3,...]

3 [1] -> error

this tutorial

part I: synquid



refinement types →
recursive programs

part II: resyn



resource types →
efficient programs

part III: hoogle+

H+

Haskell types →
function compositions

simple types

expressive types