

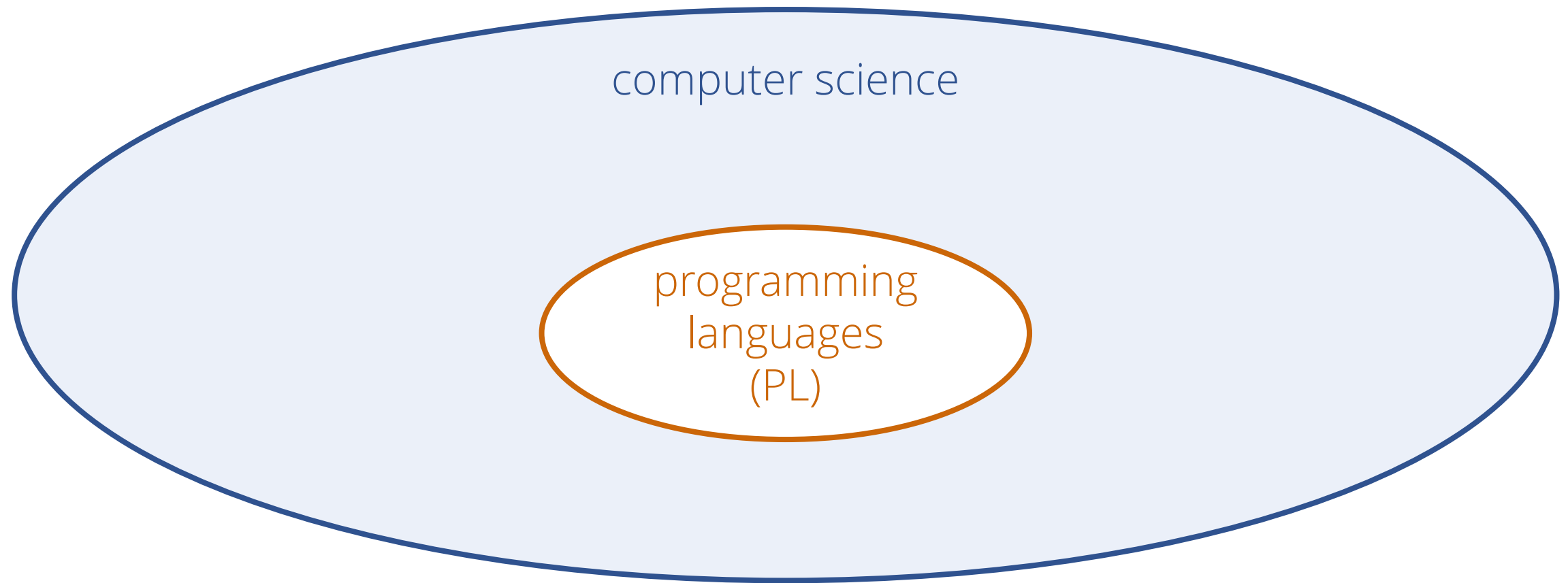
generating programs from types

Nadia Polikarpova



UCSDCSE
Computer Science and Engineering

my research



our goal: automate programming



example: insert into a sorted list

Input:

x

5

xs

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

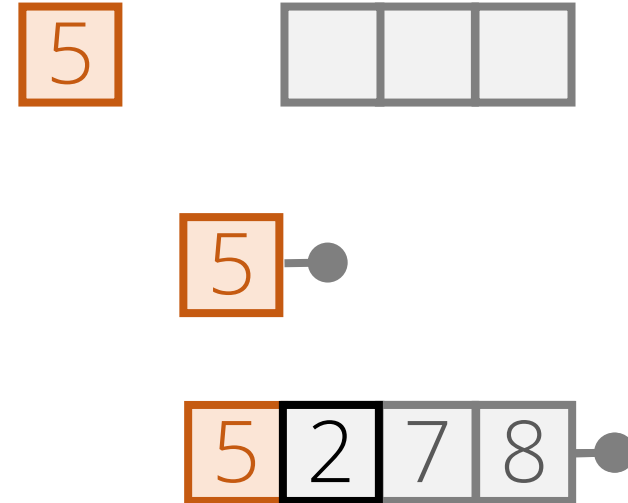
Output:

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)
```



```

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

automates
pointer manipulation &
memory management



```

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
```

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

← automates
argument passing
& memory access

```
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 →



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'
```

▲ 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

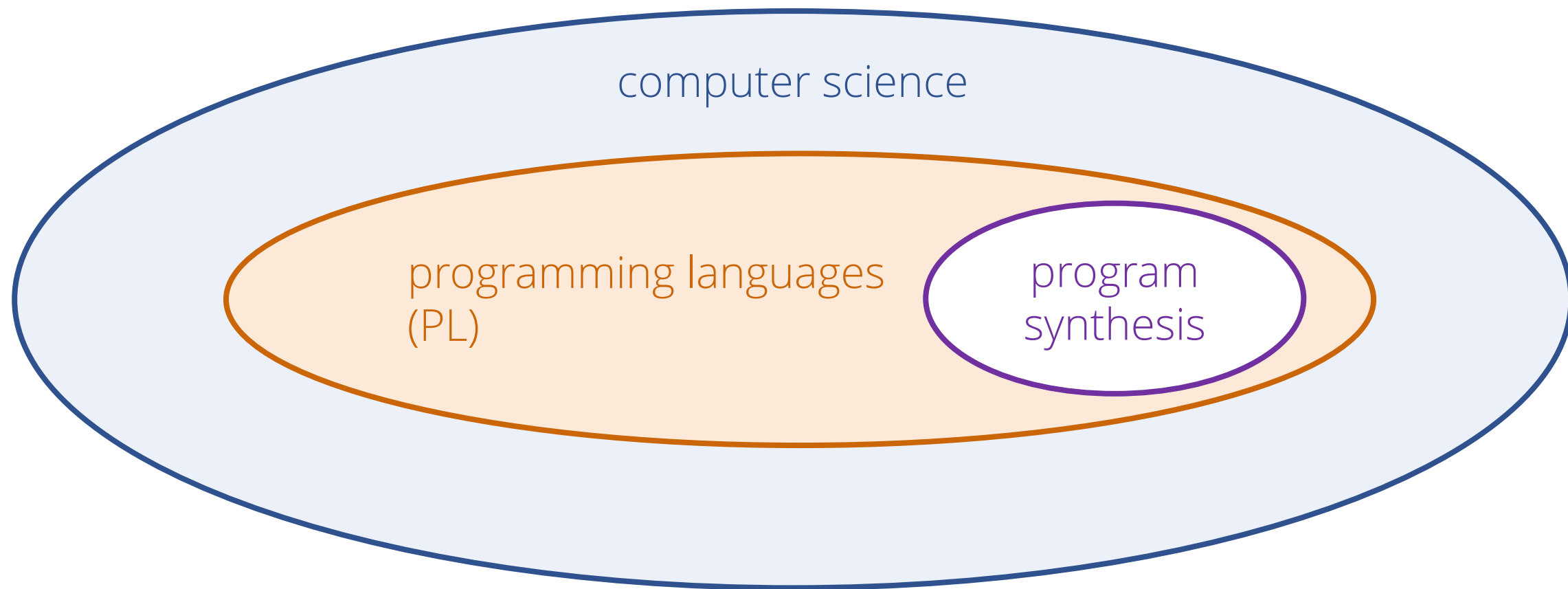
▲ You can implement

157

▼

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

my research



program synthesis

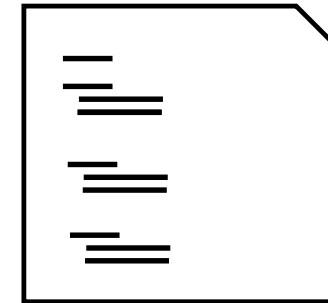
specification



search



program



program
space

type-driven program synthesis

specification

type

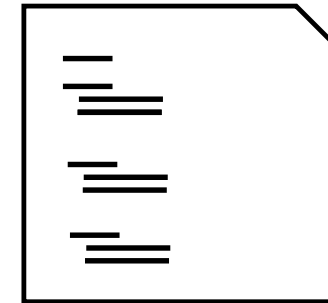


search

type-
directed



program



program
space

this talk

synquid



synthesis from
refinement types

hoogle+



towards a
programmer's assistant
for Haskell

this talk

synquid



synthesis from
refinement types

1. types as specifications
2. type-directed search
3. case study

this talk

synquid



synthesis from
refinement types

1. types as specifications

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` $\rightarrow$ `List a` $\rightarrow$ `List a`

insert in synquid

specification

`a → List a`
`→ List a`



code

`Nil`



`Nil, Cons, ≤, ...`

insert in synquid

specification

$a \rightarrow \text{list } a$

ambiguous!



code

Nil



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

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

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

refinement types: sorted lists

```
data List a where
```

```
Nil :: List a
```

```
Cons :: h:a →
```

```
  t:SList {v:a | h ≤ v} →  
  List 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 U {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)
```

this talk

synquid



synthesis from
refinement types

1. types as specifications
2. type-directed search
3. case study

this talk

synquid



synthesis from
refinement types

2. type-directed search

insert in synquid

specification

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



code

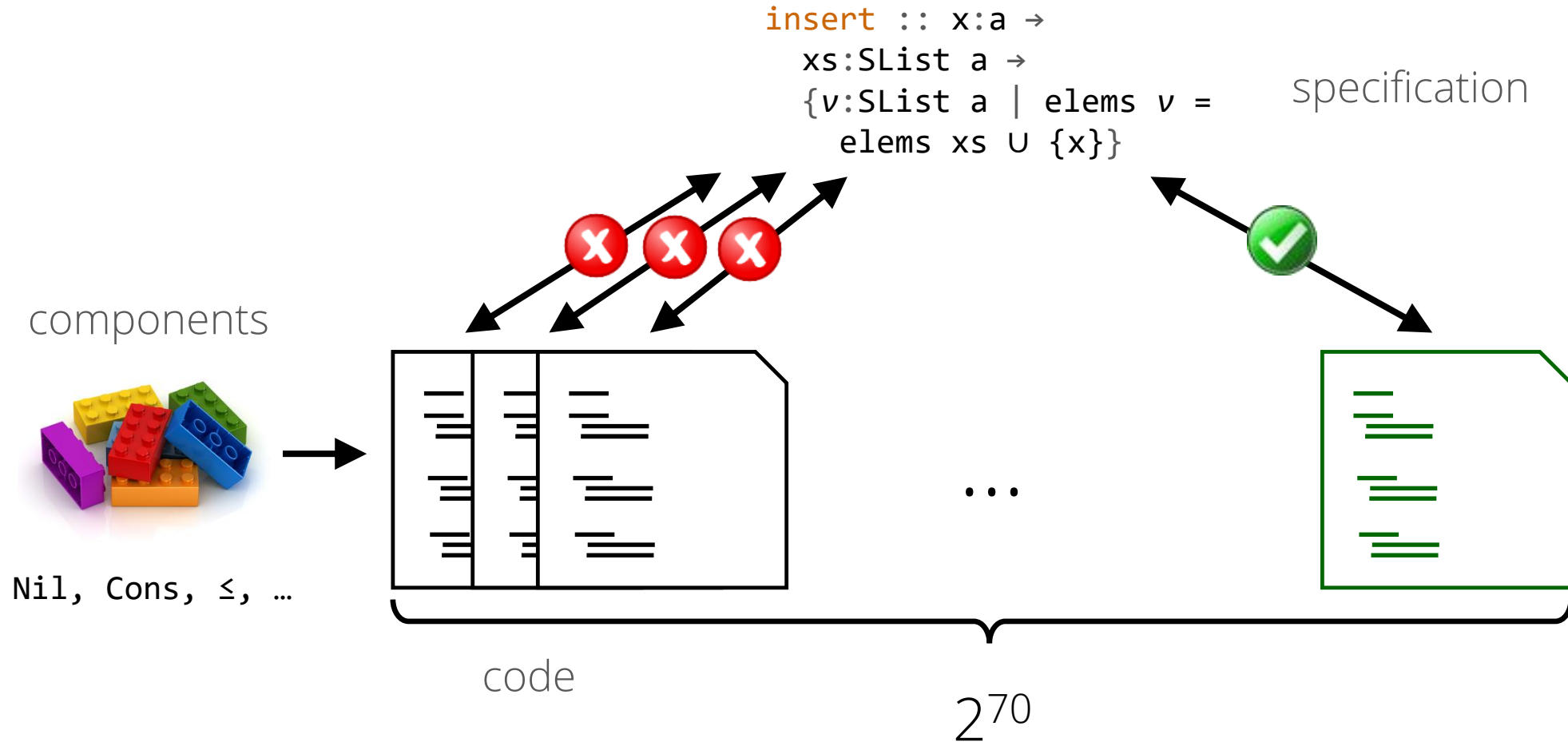
```
match xs with  
Nil → Cons x Nil  
Cons h t →  
  if x ≤ h  
  then Cons x xs  
  else Cons h (insert x t)
```

synthesis as search

specification

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

synthesis as search

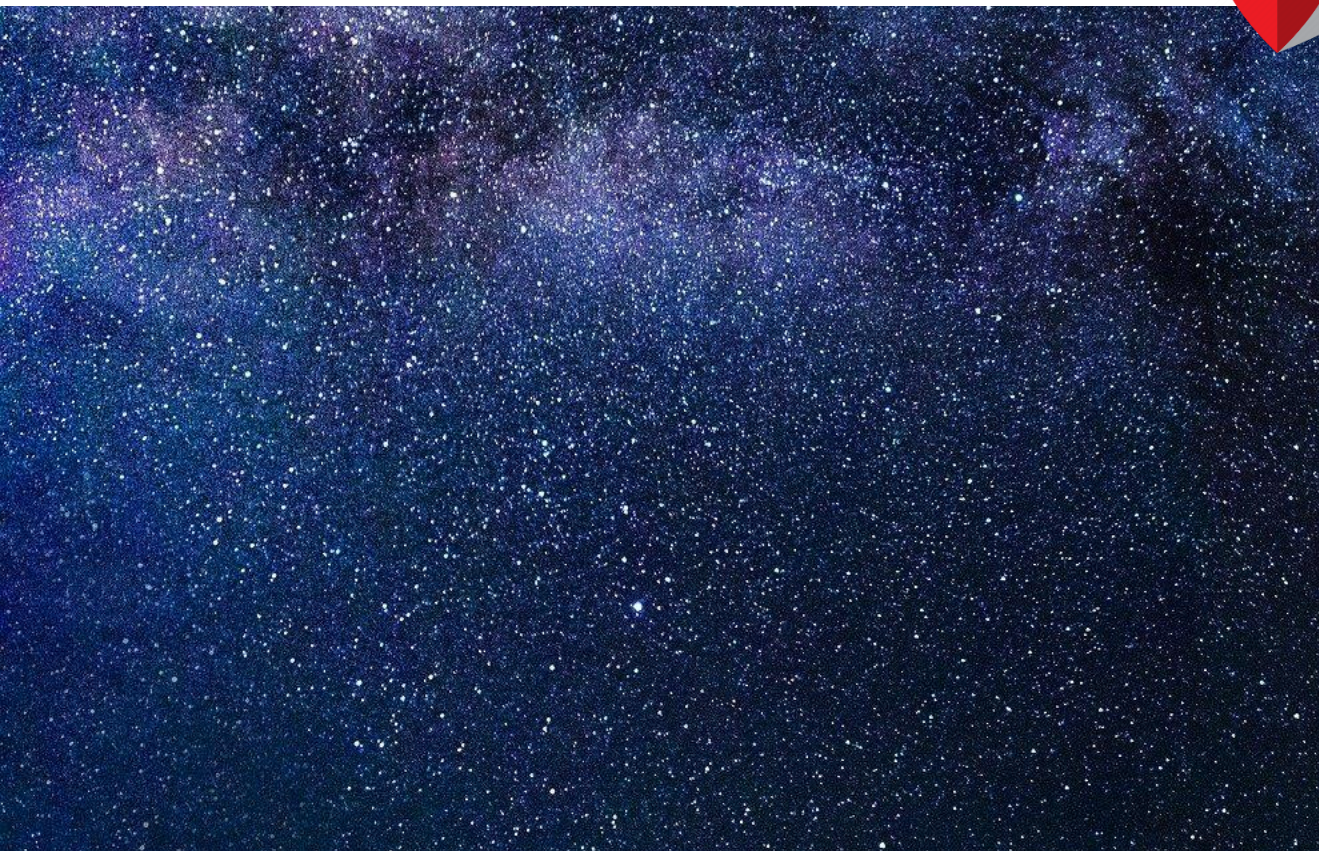


synthesis as search



synthesis as search









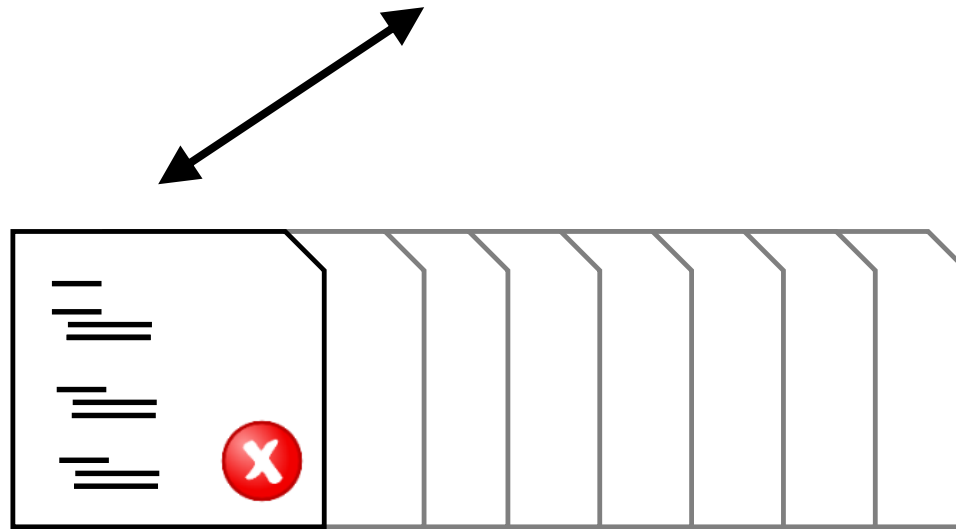


key idea: reject hopeless programs early

reject hopeless programs early

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

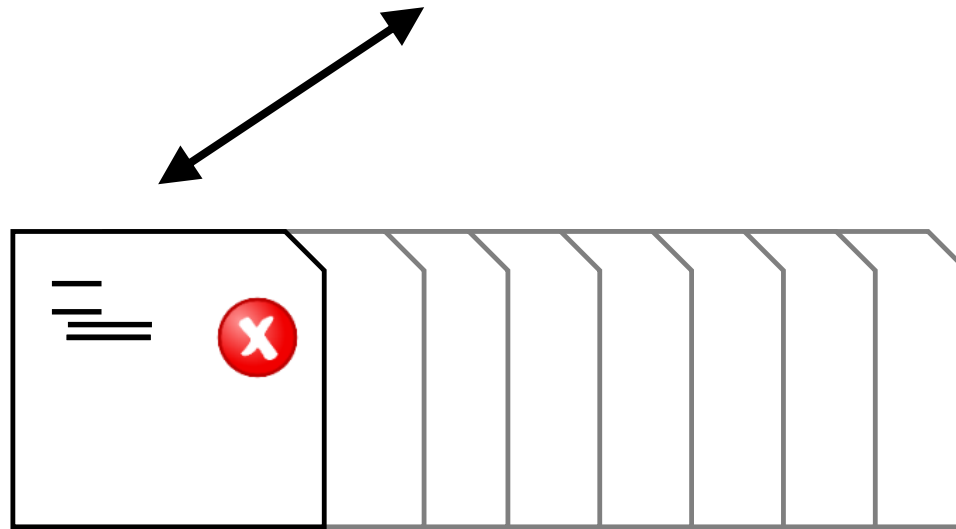
specification



reject hopeless programs early

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

specification



reject hopeless programs early

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



insert x xs =
match xs **with**

$\text{Nil} \rightarrow \text{Nil}$

$\text{Cons } h \ t \rightarrow$



2^{50}

Expected

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

and got

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



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)
```

components



this talk

synquid



synthesis from
refinement types

1. types as specifications
2. type-directed search
3. case study

this talk

synquid



synthesis from
refinement types

3. case study

case study: negation normal form

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

$\Rightarrow$ 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 $\neg$, $\wedge$, $\vee$

2. $\neg$ only at variables

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
```

```
  NOrr  :: NNF → NNF → NNF
```

negated?



nnf: specification

data Fml **where**

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

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
```

data NNF **where**

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

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 → {v:NNF | nEval v = 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))
```

double-neg

De Morgan

⇒ def.

this talk

synquid



synthesis from
refinement types

1. types as specifications
2. type-directed search
3. case study

this talk

synquid



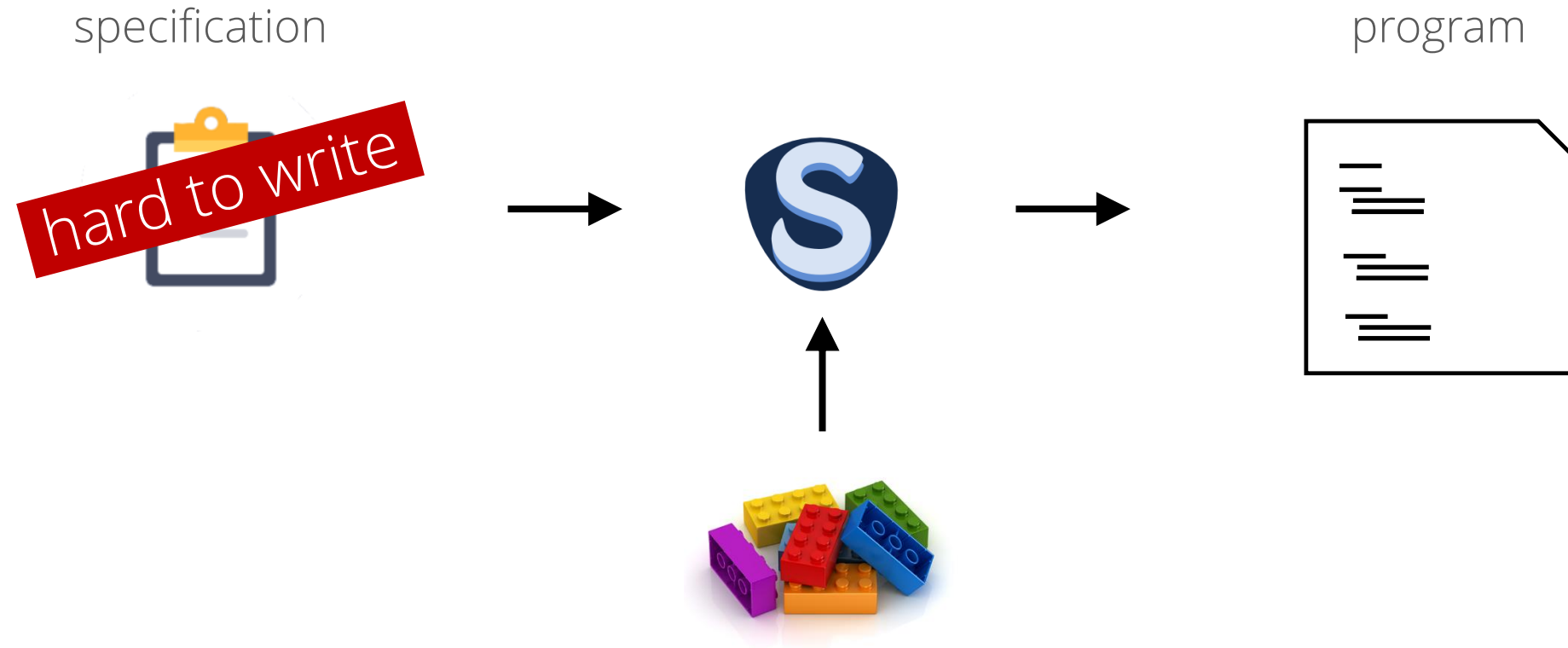
synthesis from
refinement types

hoogle+

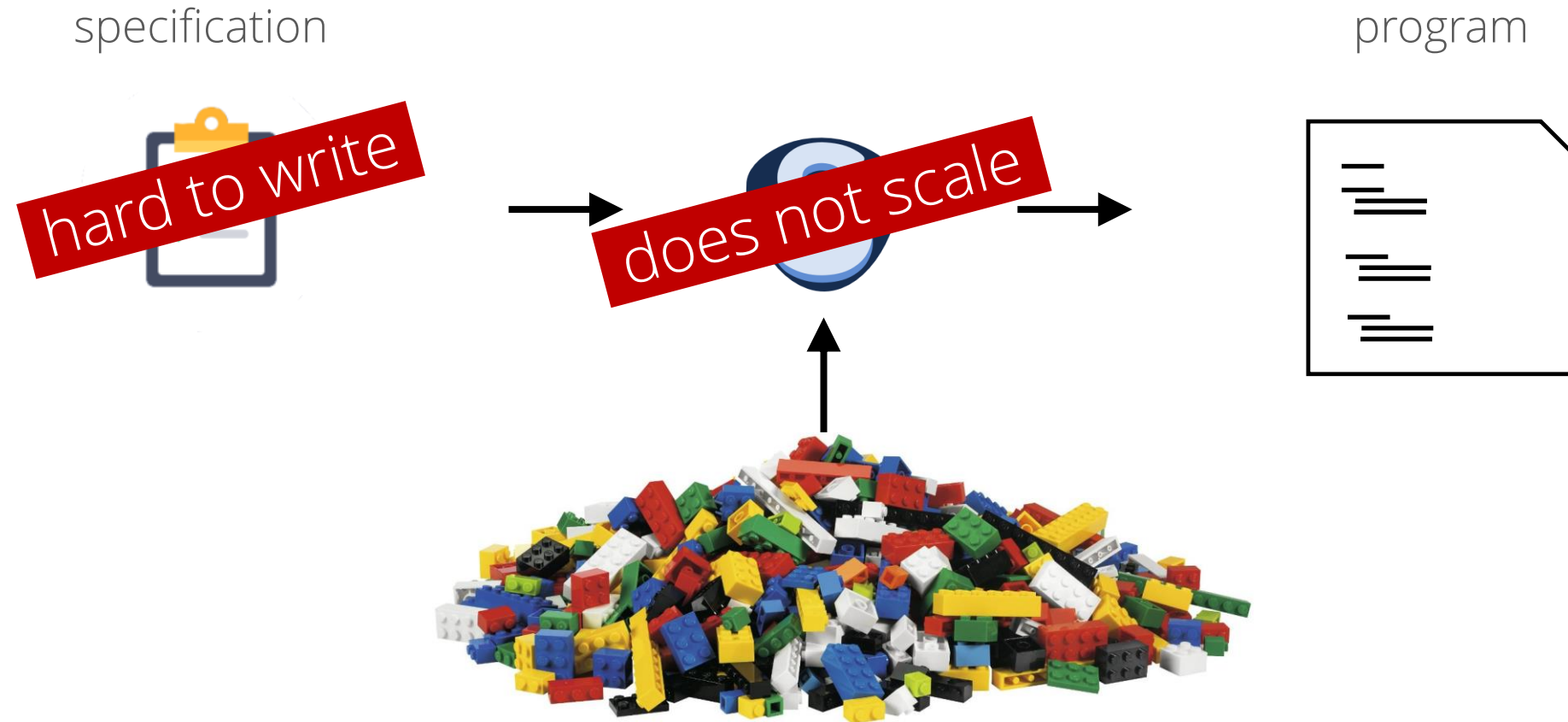


programmer's assistant
for Haskell

synquid: limitations



synquid: limitations



inspiration: hoogle

Hoogle

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

Data.List

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

```
map f [x1, x2, ..., xn] == [f x1, f x2, ..., f xn]
```

hoogle needs synthesis!

Hoogλe

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

Search

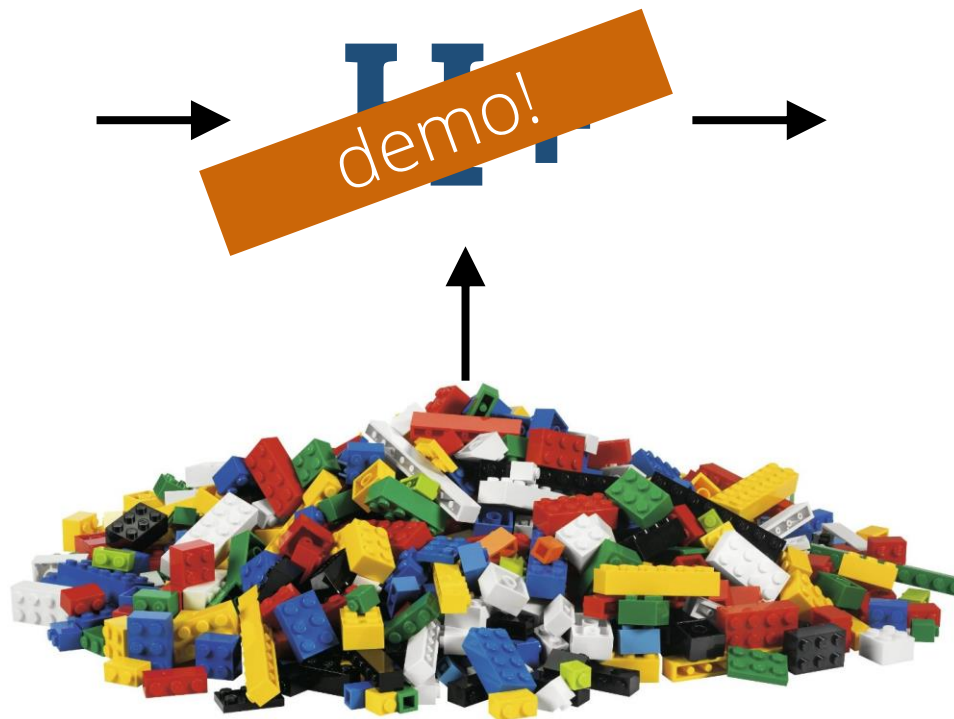
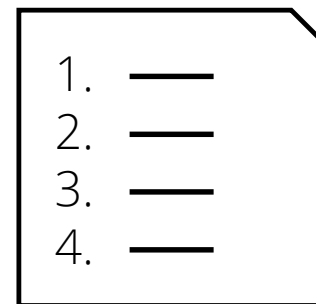
No results found!

hoogle+

specification

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

programs



Haskell
libraries

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+

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

Search

```
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 [a,b,c] -> [a,b]

3 [a] -> [a]

drop n xs

1 [a,b,c] -> [b,c]

3 [a] -> []

concat (replicate n xs)



2 [a,b,c] -> [a,b,c,a,b,c]

3 [a] -> [a,a,a]

repeat (xs !! n)

1 [a,b,c] -> [b,b,b,b,...]

3 [a] -> error

thank you!

synquid



<http://comcom.csail.mit.edu/comcom/#Synquid>

hoogle+



http://goto.ucsd.edu/hoogle_plus