generating programs from types

Nadia Polikarpova

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, [ebp]
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) {
        prev = temp;
        temp = temp->next;
        while(temp != NULL && x > temp->val) {
            temp = temp->next;
        }
        if (temp == NULL) {
            prev->next = new;
        } else {
            new->next = temp;
            prev->next = new;
        }
    } else {
        prev = temp;
        temp = xs->next;
        while(temp != NULL && x > temp->val) {
            temp = temp->next;
        }
        if (temp == NULL) {
            prev->next = new;
        } else {
            new->next = temp;
            prev->next = new;
        }
    }
}
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)
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)
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"]
```
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"]
```

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?

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:

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

wanted: automatic programmer’s assistant
program synthesis
program synthesis

specification → search → program

program space
program synthesis

specification → search → program

program space
program synthesis

specification
examples
assertions
natural language
...

search

program space

program
program synthesis

specification
examples
assertions
natural language
types!

search

program

program space
the future is (almost) here
the future is (almost) here

Hoogle

Char -> String -> [String]

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

ghc Util
the future is (almost) here

Char -> String -> [String]

\texttt{split :: Char -> String -> [String]}

\texttt{ghc Util}

what about generating new programs?
simple types  expressive types
simple types

more programmer-friendly
more ambiguous

expressive types
more programmer-friendly
more ambiguous

less programmer-friendly
less ambiguous

simple types

expressive types
this talk

part I: synquid

refinement types → recursive programs

simple types

expressive types
this talk

part I: synquid

part II: hoogle+

H+

Haskell types $\rightarrow$ function compositions

simple types

refinement types $\rightarrow$ recursive programs

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

1. types as specifications
2. type-directed search

refinement types → recursive programs

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

synquid

1. types as specifications

refinement types → recursive programs
example: insert into a sorted list
example: insert into a sorted list

Input:

X
5
example: insert into a sorted list

Input:

```
X
```

```
XS
```

```
1 2 7 8
```

```
5
```
example: insert into a sorted list

Input:

\[
\begin{align*}
X & \quad 5 \\
xs & \quad 1 \; 2 \; 7 \; 8
\end{align*}
\]

Output:

\[
\begin{align*}
ys & \quad 1 \; 2 \; 5 \; 7 \; 8
\end{align*}
\]
insert in a functional language

\[
\text{insert } x \; \text{xs } = \quad \boxed{5}
\]
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 =
  \textbf{match} \; xs \; \textbf{with}
  \begin{align*}
  & \text{Nil} \rightarrow \\
  & \text{Cons} \; x \; \text{Nil} \\
  & \text{Cons} \; h \; t \rightarrow
  \end{align*}
insert in a functional language

insert x xs =
  match xs with
  Nil →
  Cons x Nil
  Cons h t →
insert in a functional language

\[
\text{insert } x \; \text{xs} = \\
\text{match xs with} \\
\text{Nil } \rightarrow \\
\text{Cons } x \; \text{Nil} \\
\text{Cons } h \; t \rightarrow \\
\text{if } x \leq h
\]
insert in a functional language

```haskell
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

```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)
```

insert in a functional language

```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)
```
our goal

specification

program space

code

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

Nil, Cons, ≤, …
our goal

specification

? → S → 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
types are specifications
types are specifications

\[
\text{insert} :: a \rightarrow \text{List } a \rightarrow \text{List } a
\]
types are specifications

\[
\text{Ord } a \implies \text{ insert } :: a \to \text{ List } a \to \text{ List } a
\]
types are specifications

insert :: a → List a → List a
insert in synquid

specification

\[ a \rightarrow \text{List } a \rightarrow \text{List } a \]

Nil, Cons, \leq, …
insert in synquid

specification

\[ a \rightarrow \text{List } a \rightarrow \text{List } a \]

code

Nil, Cons, ≤, …
insert in synquid

specification

\[ a \rightarrow \text{List} \ a \rightarrow \text{List} \ a \]

code

Nil, Cons, ≤, …
demo: insert

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

\[ a \rightarrow \text{List } a \rightarrow \text{List } a \]
**insert in synquid**

specification

\[ a \rightarrow \text{List } a \rightarrow \text{List } a \]

code

Nil, Cons, ≤, …
insert in synquid

specification

\[ a \rightarrow \text{list } a \rightarrow \text{Nil, Cons, } \leq, \ldots \]

code

\[ \text{Nil} \]
insert in synquid

specification

code

### list a

POWER TO THE TYPES!

a → list a

Nil, Cons, ≤, …
specification for insert
specification for insert

Input: X
specification for insert

Input:

\[ x \]

\[ x_s: \text{sorted list} \]
specification for insert

Input:
  x
  xs: sorted list

Output:
  ys: sorted list
specification for insert

Input:
- \( x \)
- \( xs \): sorted list

Output:
- \( ys \): sorted list
  - \( \text{elems } ys = \text{elems } xs \cup \{x\} \)
specification for insert

Input:

\[ x \]
\[ \text{xs: sorted list} \]

Output:

\[ \text{ys: sorted list} \]
\[ \text{elems } ys = \text{elems } xs \cup \{x\} \]

can I write this as a type?
refinement types

Int
refinement types

\{ \text{v: Int} \mid 0 \leq \text{v} \}
refinement types

\{ v:\text{Int} \mid 0 \leq v \}

natural numbers
refinement types

\[ \text{List} \left\{ v : \text{Int} \mid 0 \leq v \right\} \]

lists of nats
refinement types: sorted lists

data List a where
refinement types: sorted lists

```
data List a  where
  Nil :: List a
```
refinement types: sorted lists

\[
data \ List \ a \ where
\begin{align*}
\text{Nil} & :: \ List \ a \\
\text{Cons} & :: \ h : a \to \\
& \quad t : \ List \ a \to \\
& \quad \text{List} \ a
\end{align*}
\]
refinement types: sorted lists

```haskell
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

```haskell
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

\[
\text{insert} :: \text{x:a} \rightarrow \text{xs:SList a} \rightarrow \text{SList a}
\]
refinement type for insert

\[
\text{insert} :: x : a \to xs : \text{SList} a \to \\
\{ v : \text{SList} a \mid \text{elems } v = \text{elems } xs \cup \{ x \}\}
\]
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 \leq h
      then Cons x xs
      else Cons h (insert x t)
```
demo: insert

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

synquid

1. types as specifications
2. type-directed search

refinement types → recursive programs
how did this happen?

**specification**

\[
\text{insert} :: x :: a \rightarrow \\
\text{xs} :: \text{SList a} \rightarrow \\
\{v :: \text{SList a} \mid \text{elems} \ v = \\
\text{elems} \ \text{xs} \cup \{x\}\}
\]

**code**

\[
\text{match} \ \text{xs} \ \text{with} \\
\text{Nil} \rightarrow \text{Cons} \ x \ \text{Nil} \\
\text{Cons} \ h \ t \rightarrow \\
\text{if} \ x \leq h \\
\text{then} \ \text{Cons} \ x \ \text{xs} \\
\text{else} \ \text{Cons} \ h \ (\text{insert} \ x \ t)
\]
synthesis as search

```
insert :: x:a \rightarrow
xs:SList a \rightarrow
\{v:SList a \mid \text{elems } v =
\text{elems } xs \cup \{x\}\}
```
synthesis as search

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

\[
\text{insert} :: \text{x:a} \rightarrow \\
\text{xs:SList a} \rightarrow \\
\{v:\text{SList a} \mid \text{elems v} = \\
\text{elems xs} \cup \{x\}\}
\]

components

Nil, Cons, ≤, …
synthesis as search

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

Nil, Cons, ≤, ...

code
synthesis as search

\[
\text{insert :: } x : a \rightarrow \\
x : \text{SList } a \rightarrow \\
\{ v : \text{SList } a \mid \text{elems } v = \\
\text{elems } x \text{S } \cup \{ x \}\}
\]

Nil, Cons, ≤, …
synthesis as search

\[
\text{insert} :: x : a \rightarrow \\
x : SList a \rightarrow \\
\{ v : SList a \mid \text{elems } v = \\
\text{elems } x : SList a \cup \{ x \}\}
\]
insert :: x:a →
xs:SList a →
{v:SList a | elems v =
elems xs U {x}}
synthesis as search

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

Nil, Cons, ≤, …

components

code

specification
**synthesis as search**

\[
\text{insert :: } x : a \rightarrow \\
xs : \text{SList a} \rightarrow \\
\{ v : \text{SList a} \mid \text{elems v} = \\
\text{elems xs} \cup \{x\}\}
\]

components

Nil, Cons, ≤, …

code

too many

specification
synthesis as search

\[
\text{insert} :: x:a \rightarrow \\\nxs:SList a \rightarrow \\\n\{v:SList a \mid \text{elems } v = \\\n\text{elems } xs \cup \{x\}\}
\]

components

Nil, Cons, ≤, …

code

\[2^{70}\]

specification
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

\[
\text{insert} :: x:a \rightarrow \\
\text{xs:}\text{SList a} \rightarrow \\
\{v:\text{SList a} \mid \text{elems } v = \\
\text{elems } \text{xs} \cup \{x\}\}
\]
rejecting hopeless programs

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

previous work*:
bottom-up type checking

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

```
insert :: x:a \rightarrow
xs:SList a \rightarrow
\{v:SList a | \text{elems } v = \text{elems } xs \cup \{x\}\}
```
rejecting hopeless programs

our work:
  top-down type checking

specification

insert :: x:a →
xs:SList a →
{v:SList a | elems v =
elems xs ∪ {x}}
rejecting hopeless programs

\[ x : a \rightarrow xs : \text{SList} \ a \rightarrow \{ v : \text{SList} \ a \mid \text{elems} \ v = \text{elems} \ xs \cup \{x\} \} \]
rejecting hopeless programs

\[ x : a \rightarrow xs : SList a \rightarrow \{ v : SList a \mid \text{elems } v = \text{elems } xs \cup \{ x \} \} \]

\[ \text{insert } x \text{ } xs = ?? \]
rejecting hopeless programs

\[ x : a \rightarrow xs : SList a \rightarrow \{ v : SList a \mid \text{elems } v = \text{elems } xs \cup \{x\} \} \]

\[ \text{insert } x \text{ } xs = ?? \]

hopeless?
rejection of hopeless programs

\[
x : a \rightarrow xs : \text{SList}\ a \rightarrow \{v : \text{SList}\ a \mid \text{elems}\ v = \text{elems}\ xs \cup \{x\}\}\\
i\text{nterst}\ x\ xs = ??
\]
rejecting hopeless programs

\[
x : a \to xs : SList a \to \\
\{ v : SList a \mid \text{elems } v = \text{elems } xs \cup \{x\} \}
\]

\[
\text{insert } x \hspace{1em} xs = \\
\text{match } xs \text{ with} \\
\text{Nil } \to \text{ ?? } \\
\text{Cons } h \hspace{1em} t \to \text{ ?? }
\]
rejecting hopeless programs

\[
x : a \rightarrow xs : SList\ a \rightarrow \\
\{ v : SList\ a \mid \text{elems}\ v = \text{elems}\ xs \cup \{ x \} \}
\]

insert x xs =
  \text{match}\ xs\ with
  \text{Nil} \rightarrow ???
  \text{Cons}\ h\ t \rightarrow ???

hopeless?
rejecting hopeless programs

\[
x : \text{a} \to \ \text{xs} : \text{SList a} \to \\
\{ v : \text{SList a} \mid \text{elems } v = \text{elems xs} \cup \{x\} \}
\]

\[
\text{insert } x \text{ xs } = \\
\text{match } \text{xs} \text{ with} \\
\text{Nil } \to \text{ ??} \\
\text{Cons h t } \to \text{ ??}
\]
rejecting hopeless programs

\[ x : a \rightarrow xs : \text{SList} \ a \rightarrow \{ v : \text{SList} \ a \mid \text{elems} \ v = \text{elems} \ xs \cup \{x\} \} \]

\[ \text{insert} \ x \ xs = \]

\[ \text{match} \ xs \ \text{with} \]

\[ \text{Nil} \rightarrow \text{Nil} \]

\[ \text{Cons} \ h \ t \rightarrow ?? \]
rejecting hopeless programs

\[\text{x:a } \rightarrow \text{xs:SList a } \rightarrow \{v:SList a \mid \text{elems } v = \text{elems } xs \cup \{x\}\}\]

\[
\text{insert } x \text{ xs } = \\
\text{match xs with} \\
\text{Nil } \rightarrow \text{Nil} \\
\text{Cons } h \text{ t } \rightarrow \text{??}
\]
rejecting hopeless programs

\[
x : a \rightarrow xs : SList a \rightarrow \{ v : SList a \mid \text{elems } v = \text{elems } xs \cup \{ x \}\}
\]

\[
\text{insert } x \text{ } xs = \\
\quad \text{match } xs \text{ with} \\
\quad \quad \text{Nil } \rightarrow \text{Nil} \\
\quad \quad \text{Cons } h \text{ } t \rightarrow ??
\]

hopeless: output must always contain x!
rejecting hopeless programs

\[ x : a \rightarrow xs : SList a \rightarrow \{ v : SList a \mid \text{elems } v = \text{elems } xs \cup \{ x \} \} \]

\[
\text{insert } x \text{ } xs = \\
\text{match } xs \text{ with} \\
\text{Nil } \rightarrow \text{Nil} \\
\text{Cons } h \text{ } t \rightarrow \\
\]

2\(^{50}\)

hopeless: output must always contain x!
top-down type checking

\[
x : a \rightarrow xs : SList\ a \rightarrow \\
\{ v : SList\ a \mid \text{elems}\ v = \text{elems}\ xs \cup \{ x \}\}
\]

\[
\text{insert}\ x\ xs = \\
\text{match}\ xs\ \text{with} \\
\text{Nil} \rightarrow \text{Nil} \\
\text{Cons}\ h\ t \rightarrow \ ??
\]
top-down type checking

\{ v : \text{SList } a \mid \text{elems } v = \text{elems } xs \cup \{ x \}\}
top-down type checking

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

\[
\text{insert} \ x \ xs = \\
\quad \text{match} \ xs \ \text{with} \\
\quad \quad \text{Nil} \rightarrow \text{Nil} \\
\quad \quad \text{Cons} \ h \ t \rightarrow ??
\]
top-down type checking

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

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

insert x xs =
match xs with
  Nil \rightarrow \text{Nil}
  \text{Cons } h \ t \rightarrow ??
top-down type checking

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

\text{insert}\ x\ xs = \\
\text{match}\ xs\ \text{with} \\
\text{Nil} \rightarrow \text{Nil} \\
\text{Cons}\ h\ t \rightarrow ??

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

SMT solver: INVALID!
top-down type checking

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

\text{Constraints: } \forall x : \emptyset = \emptyset \cup \{x\}

SMT solver: INVALID!
**insert in synquid**

**specification**

```haskell
insert :: x:a -> xs:SList a ->
{v:SList a | elems v =
elems xs U {x}}
```

**code**

```haskell
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**

\[
\text{insert} :: x : a \rightarrow \\
x : \text{SList } a \rightarrow \\
\{ v : \text{SList } a \mid \text{elems } v = \\
\text{elems } x : \text{SList } a \cup \{ x \}\}
\]

**code**

\[
\text{match } xs \text{ with} \\
\text{Nil } \rightarrow \text{Cons } x \text{ Nil} \\
\text{Cons } h \ t \rightarrow \\
\text{if } x \leq h \\
\text{then } \text{Cons } x \ t \\
\text{else } \text{Cons } h \ \text{(insert } x \ t) \\
\]

---

*something less boring?*
case study: negation normal form

\neg(a \Rightarrow b \lor c)
case study: negation normal form

\[ \neg(a \Rightarrow b \lor c) \]

1. only \( \neg, \land, \lor \)
2. \( \neg \) only at variables
case study: negation normal form

\neg (a \Rightarrow b \lor c)

1. only $\neg$, $\land$, $\lor$
2. $\neg$ only at variables
case study: negation normal form

\[ \neg (a \Rightarrow b \lor c) \]

\[ \neg (\neg a \lor (b \lor c)) \]

\[ \Rightarrow \text{def.} \]

1. only \( \neg, \land, \lor \)
2. \( \neg \) only at variables
case study: negation normal form

\[ \neg (a \Rightarrow b \lor c) \]

\[ \neg (\neg a \lor (b \lor c)) \]

\[ \neg \neg a \land \neg (b \lor c) \]

1. only \( \neg, \land, \lor \)
2. \( \neg \) only at variables
case study: negation normal form

\[ \neg(a \Rightarrow b \lor c) \]

\[ \Rightarrow \text{ def.} \]

\[ \neg(\neg a \lor (b \lor c)) \]

\[ \neg\neg a \land \neg(b \lor c) \]

De Morgan

double-neg

1. only \( \neg, \land, \lor \)

2. \( \neg \) only at variables

\[ a \land \neg(b \lor c) \]
case study: negation normal form

\[ \neg(a \Rightarrow b \lor c) \]

\[ \Rightarrow \text{def.} \]

\[ \neg(\neg a \lor (b \lor c)) \]

\[ \neg \neg a \land \neg (b \lor c) \]

\[ \text{De Morgan} \]

\[ a \land \neg (b \lor c) \]

\[ \text{De Morgan} \]

\[ a \land (\neg b \lor \neg c) \]

1. only \(\neg, \land, \lor\)

2. \(\neg\) only at variables
nnf: data types

data Fml where
nnf: data types

data Fml where
  Var :: String → Fml
nnf: data types

```haskell
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**

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

```haskell
data NNF where
```

```haskell
```
nnf: data types

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

```haskell
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
```

negated?
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

negated?
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**

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

```haskell
data NNF where
    NAtom :: String → Bool → NNF
    NAnd :: NNF → NNF → NNF
    NOr :: NNF → NNF → NNF
```
nnf: specification

\textbf{data Fml where}
Var :: String → Fml
Not :: Fml → Fml
And :: Fml → Fml → Fml
Or :: Fml → Fml → Fml
Imp :: Fml → Fml → Fml

\textbf{measure eval :: Fml → Bool where}

\textbf{data NNF where}
NAtom :: String → Bool → NNF
NAnd :: NNF → NNF → NNF
NOr :: NNF → NNF → NNF
**nnf: specification**

```haskell
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
```

```haskell
data NNF where
  NAtom :: String → Bool → NNF
  NAnd :: NNF → NNF → NNF → NNF
  NOr :: NNF → NNF → NNF
```
nnf: specification

\[
\text{data Fml where}
\begin{align*}
\text{Var} & \:: \text{String} \rightarrow \text{Fml} \\
\text{Not} & \:: \text{Fml} \rightarrow \text{Fml} \\
\text{And} & \:: \text{Fml} \rightarrow \text{Fml} \rightarrow \text{Fml} \\
\text{Or} & \:: \text{Fml} \rightarrow \text{Fml} \rightarrow \text{Fml} \\
\text{Imp} & \:: \text{Fml} \rightarrow \text{Fml} \rightarrow \text{Fml}
\end{align*}
\]

\[
\text{measure eval} :: \text{Fml} \rightarrow \text{Bool where}
\begin{align*}
\text{Var} \; v & \rightarrow \text{env} \; v \\
\text{Not} \; f & \rightarrow !\text{eval} \; f
\end{align*}
\]

\[
\text{data NNF where}
\begin{align*}
\text{NAtom} & \:: \text{String} \rightarrow \text{Bool} \rightarrow \text{NNF} \\
\text{NAnd} & \:: \text{NNF} \rightarrow \text{NNF} \rightarrow \text{NNF} \\
\text{NOr} & \:: \text{NNF} \rightarrow \text{NNF} \rightarrow \text{NNF}
\end{align*}
\]
nnf: specification

data Fml where
  Var :: String \rightarrow Fml
  Not :: Fml \rightarrow Fml
  And :: Fml \rightarrow Fml \rightarrow Fml
  Or :: Fml \rightarrow Fml \rightarrow Fml
  Imp :: Fml \rightarrow Fml \rightarrow Fml

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

data NNF where
  NAtom :: String \rightarrow Bool \rightarrow NNF
  NAnd :: NNF \rightarrow NNF \rightarrow NNF
  NOR :: NNF \rightarrow NNF \rightarrow NNF
nnf: specification

```haskell
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
```

```haskell
data NNF where
  NAtom :: String -> Bool
  NAnd :: NNF -> NNF -> NNF
  NOr :: NNF -> NNF -> NNF
```
nnf: specification

```haskell
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
```

```haskell
data NNF where
  NAtom :: String → Bool → NNF
  NAnd :: NNF → NNF → NNF
  NOr :: NNF → NNF → NNF
```
nnf: specification

```haskell
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
```

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

measure nEval :: NNF → Bool where
```

---

39
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
NAnd :: NNF → NNF → NNF
NOr :: NNF → NNF → NNF

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

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
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
- 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

\[
\text{nnf} :: f:Fml \rightarrow \{v: \text{NNF} \mid \text{nEval } v = \text{eval } f\}
\]
nnf: synthesized code

\[ \text{nnf} :: f : \text{Fml} \rightarrow \{ v : \text{NNF} \mid \text{nEval } v = \text{eval } f \} \]
nnf: synthesized code

\[
\text{nnf} :: f : \text{Fml} \rightarrow \{\nu : \text{NNF} | \text{nEval} \, \nu = \text{eval} \, f\}
\]

\[
\text{nnf} \, p = \text{match} \, p \, \text{with}
\]

\[
\begin{align*}
\text{BoollLiteral} \, x_2 & \rightarrow \text{if} \, x_2 \\
\text{then} \, \text{NOr} \, (\text{NAtom} \, \text{dummy} \, x_2) & (\text{NAtom} \, \text{dummy} \, \text{False}) \\
\text{else} & \, \text{NAnd} \, (\text{NAtom} \, \text{dummy} \, x_2) (\text{NAtom} \, \text{dummy} \, \text{True})
\end{align*}
\]

\[
\begin{align*}
\text{Var} \, x_{16} & \rightarrow \text{NAtom} \, x_{16} \, \text{False} \\
\text{Not} \, x_{20} & \rightarrow \text{match} \, x_{20} \, \text{with} \\
\text{BoollLiteral} \, x_{22} & \rightarrow \text{if} \, x_{22} \\
\text{then} \, \text{nnf} \, (\text{BoollLiteral} \, \text{False}) & \\
\text{else} \, \text{nnf} \, (\text{BoollLiteral} \, \text{True})
\end{align*}
\]

\[
\begin{align*}
\text{Var} \, x_{28} & \rightarrow \text{NAtom} \, x_{28} \, \text{True} \\
\text{Not} \, x_{32} & \rightarrow \text{nnf} \, x_{32} \\
\text{And} \, x_{36} \, x_{37} & \rightarrow \text{NOr} \, (\text{nnf} \, (\text{Not} \, x_{36})) (\text{nnf} \, (\text{Not} \, x_{37})) \\
\text{Or} \, x_{46} \, x_{47} & \rightarrow \text{NAnd} \, (\text{nnf} \, (\text{Not} \, x_{46})) (\text{nnf} \, (\text{Not} \, x_{47})) \\
\text{Implies} \, x_{56} \, x_{57} & \rightarrow \text{NAnd} \, (\text{nnf} \, x_{56}) (\text{nnf} \, (\text{Not} \, x_{57}))
\end{align*}
\]

\[
\begin{align*}
\text{And} \, x_{65} \, x_{66} & \rightarrow \text{NAnd} \, (\text{nnf} \, x_{65}) (\text{nnf} \, x_{66}) \\
\text{Or} \, x_{73} \, x_{74} & \rightarrow \text{NOr} \, (\text{nnf} \, x_{73}) (\text{nnf} \, x_{74}) \\
\text{Imp} \, x_{81} \, x_{82} & \rightarrow \text{NOr} \, (\text{nnf} \, x_{82}) (\text{nnf} \, (\text{Not} \, x_{81}))
\end{align*}
\]
nnf: synthesized code

\[
\text{nnf} :: f : \text{Fml} \to \{ v : \text{NNF} \mid \text{nEval } v = \text{eval } f \}
\]

\[
nnf \ p = \text{match } p \text{ with }
\]

- \text{BoolLiteral } x2 \rightarrow \text{if } x2
  \begin{align*}
  & \text{then } \text{NOr} \ (\text{NAtom dummy } x2) \ (\text{NAtom dummy False}) \\
  & \text{else } \text{NAnd} \ (\text{NAtom dummy } x2) \ (\text{NAtom dummy True})
  \end{align*}

- \text{Var } x16 \rightarrow \text{NAtom } x16 \ \text{False}

- \text{Not } x20 \rightarrow \text{match } x20 \text{ with }
  \begin{align*}
  & \text{BoolLiteral } x22 \rightarrow \text{if } x22 \\
  & \quad \text{then } \text{nnf} \ (\text{BoolLiteral False}) \\
  & \quad \text{else } \text{nnf} \ (\text{BoolLiteral True})
  \end{align*}

- \text{Var } x28 \rightarrow \text{NAtom } x28 \ \text{True}

- \text{Not } x32 \rightarrow \text{nnf } x32

- \text{And } x36 \ x37 \rightarrow \text{NOr} \ (\text{nnf } (\text{Not } x36)) \ (\text{nnf } (\text{Not } x37))

- \text{Or } x46 \ x47 \rightarrow \text{NAnd} \ (\text{nnf } (\text{Not } x46)) \ (\text{nnf } (\text{Not } x47))

- \text{Implies } x56 \ x57 \rightarrow \text{NAnd} \ (\text{nnf } x56) \ (\text{nnf } (\text{Not } x57))

- \text{And } x65 \ x66 \rightarrow \text{NAnd} \ (\text{nnf } x65) \ (\text{nnf } x66)

- \text{Or } x73 \ x74 \rightarrow \text{NOr} \ (\text{nnf } x73) \ (\text{nnf } x74)

- \text{Imp } x81 \ x82 \rightarrow \text{NOr} \ (\text{nnf } x82) \ (\text{nnf } (\text{Not } x81))

\Rightarrow \text{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))
   Or x65 x66 → NAnd (nnf x65) (nnf x66)
   Imp x73 x74 → NOr (nnf x73) (nnf x74)
```

double-neg

⇒ def.
nnf: synthesized code

```haskell
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 (Not 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 I: synquid

H+  refinement types → recursive programs
Haskell types → function compositions

part II: hoogle+

simple types
expressive types
part II

hoogle+

H +

Haskell types →
function compositions
synquid: limitation

specification → S → program
synquid: limitation

specification → hard to write → S → program
part II

hoogle+

H+

1. overcoming ambiguity
2. helping beginners with types

Haskell types $\rightarrow$ 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

Hoole

| Char -> String -> [String] | Search |

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

ghc Util
inspiration: hoogle

Char -> String -> [String]

\textbf{split} :: Char -> String -> [String]

ghc Util
example: compress a list

Input:  

1 1 2 1
example: compress a list

Input: 1 1 2 1

Output: 1 2 1
example: compress a list

Input:

1 1 2 1

Output:

1 2 1
example: compress a list

Input:

Output:
compress: specification

compress :: [a] → [a]
compress: specification

compress :: Eq a => [a] -> [a]
hoogle+

specification

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

H+

programs

1. 
2. 
3. 
4. 

Haskell libraries
Hoogle+ specification

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

demo!

Haskell libraries

programs

1. ___
2. ___
3. ___
4. ___
**hoogle+**

specification

\[ \text{Eq } a \Rightarrow [a] \rightarrow [a] \]

\[
\begin{array}{c}
\text{H}^+ \\
\text{Haskell libraries}
\end{array}
\]

programs

1. 
2. 
3. 
4. 

49
hoogle+

specification

Eq a a → [a]

ambiguous!

H+ →

programs

1. 
2. 
3. 
4. 

Haskell libraries
hoogle+

Eq a => [a] → [a]

\(\text{concat (group } \text{xs)}\)

\[
\begin{array}{c|c|c}
\text{xs} & [0,1] & [0,0] \\
\hline
[0,1] & [0,1] & [0,0] \\
[0,0] & [0,0] & [0,0] \\
\end{array}
\]

\(\text{head (group } \text{xs)}\)

\[
\begin{array}{c|c|c}
\text{xs} & [0,1] & [0,0] \\
\hline
[0,1] & [0] & [0,0] \\
[0,0] & [0] & [0,0] \\
\end{array}
\]

\(\text{last (group } \text{xs)}\)

\[
\begin{array}{c|c|c}
\text{xs} & [0,1] & [0,0] \\
\hline
[0,1] & [1] & [0,0] \\
[0,0] & [0,0] & [0,0] \\
\end{array}
\]

\(\text{map head (group } \text{xs)}\)

\[
\begin{array}{c|c|c}
\text{xs} & [0,1] & [0,0] \\
\hline
[0,1] & [0,1] & [0,0] \\
[0,0] & [0] & [0] \\
\end{array}
\]
<table>
<thead>
<tr>
<th>Eq a =&gt; [a] → [a]</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle \langle x : n \rangle \mapsto \text{concat} (\text{group} x) \rangle )</td>
</tr>
<tr>
<td>[0, 1] → [0, 1]</td>
</tr>
<tr>
<td>[0, 0] → [0, 0]</td>
</tr>
<tr>
<td>( \langle \langle x : n \rangle \mapsto \text{head} (\text{group} x) \rangle )</td>
</tr>
<tr>
<td>[0, 1] → [0]</td>
</tr>
<tr>
<td>[0, 0] → [0, 0]</td>
</tr>
<tr>
<td>( \langle \langle x : n \rangle \mapsto \text{last} (\text{group} x) \rangle )</td>
</tr>
<tr>
<td>[0, 1] → [1]</td>
</tr>
<tr>
<td>[0, 0] → [0, 0]</td>
</tr>
<tr>
<td>( \langle \langle x : n \rangle \mapsto \text{map head} (\text{group} x) \rangle )</td>
</tr>
<tr>
<td>[0, 1] → [0, 1]</td>
</tr>
<tr>
<td>[0, 0] → [0]</td>
</tr>
</tbody>
</table>
hoogle+

Haskell types → function compositions

1. overcoming ambiguity
too many irrelevant results

\textbf{Hoog\lambda+} \quad \text{Eq } a \Rightarrow [a] \rightarrow [a] \\

\[ \text{xs} \rightarrow [] \]

\[ \text{xs} \rightarrow \text{xs} \]

\[ \text{xs} \rightarrow \text{head} [] \]

\[ \text{xs} \rightarrow \text{tail} \text{ xs} \]
too many irrelevant results

\Hoog\lambda+  

$\text{Eq } a \Rightarrow [a] \rightarrow [a]$  

Search

$\\chi s \rightarrow []$

$\\chi s \rightarrow \chi s$

$\\chi s \rightarrow \text{head } []$

$\\chi s \rightarrow \text{tail } \chi s$

always crashes!
too many irrelevant results

\[ \text{Eq } a \Rightarrow [a] \rightarrow [a] \]

\[ \text{\textbackslash}xs \rightarrow [] \]

ignores the argument!

\[ \text{\textbackslash}xs \rightarrow xs \]

\[ \text{\textbackslash}xs \rightarrow \text{head} [ ] \]

always crashes!

\[ \text{\textbackslash}xs \rightarrow \text{tail} \ xs \]
too many irrelevant results

\texttt{Hoog\lambda e^+}

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{( a \Rightarrow [a] \rightarrow [a] )}</td>
<td><img src="Image" alt="Search" /></td>
</tr>
<tr>
<td>\texttt{( xs \rightarrow [] )}</td>
<td>ignores the argument!</td>
</tr>
<tr>
<td>\texttt{( xs \rightarrow xs )}</td>
<td>ignores the type class!</td>
</tr>
<tr>
<td>\texttt{( xs \rightarrow \text{head} [] )}</td>
<td>always crashes!</td>
</tr>
<tr>
<td>\texttt{( xs \rightarrow \text{tail} ; xs )}</td>
<td>ignores the type class!</td>
</tr>
</tbody>
</table>
too many irrelevant results

\[\lambda \, \text{xs} \rightarrow \text{head} \left( \text{group} \, \text{xs} \right) \]

\[\lambda \, \text{xs} \rightarrow \text{init} \left( \text{head} \left( \text{group} \, \text{xs} \right) \right) \]

\[\lambda \, \text{xs} \rightarrow \text{tail} \left( \text{head} \left( \text{group} \, \text{xs} \right) \right) \]
too many irrelevant results

<table>
<thead>
<tr>
<th>Eq a =&gt; [a] → [a]</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{x} \rightarrow \text{head (group x)} )</td>
</tr>
<tr>
<td>( \text{x} \rightarrow \text{init (head (group x))} )</td>
</tr>
</tbody>
</table>
| \( \text{x} \rightarrow \text{tail (head (group x))} \) | duplicate!
test-based filtering

type query search candidates

Eq \ a \Rightarrow
[a] \rightarrow [a]
test-based filtering

Type query: Eq a => [a] → [a]
Search
Candidates
Results
QuickCheck / SmallCheck
test-based filtering

1. does it crash on all inputs?
test-based filtering

Type query $\text{Eq a } \Rightarrow [a] \rightarrow [a]$  
Search  
Candidates  
Results

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

Eq \( a \Rightarrow [a] \rightarrow [a] \)

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

\[ \text{Eq} \ a \Rightarrow [a] \rightarrow [a] \]

\[ \text{\textbackslash}xs \rightarrow \text{concat} \ (\text{group} \ \text{xs}) \]

\[ \text{\textbackslash}xs \rightarrow \text{head} \ (\text{group} \ \text{xs}) \]

\[ \text{\textbackslash}xs \rightarrow \text{last} \ (\text{group} \ \text{xs}) \]

\[ \text{\textbackslash}xs \rightarrow \text{map head} \ (\text{group} \ \text{xs}) \]
comprehension

Eq a => [a] → [a]

\(\text{concat}\) (\text{group}\ \text{xs})

\(\text{head}\) (\text{group}\ \text{xs})

\(\text{last}\) (\text{group}\ \text{xs})

\(\text{map head}\) (\text{group}\ \text{xs})

how do I know what these programs do?
test-based comprehension

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eq a =&gt; [a] → [a]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{concat (group } \text{xs)})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0,1] → [0,1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0,0] → [0,0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{head (group } \text{xs)})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0,1] → [0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0,0] → [0,0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{last (group } \text{xs)})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0,1] → [1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0,0] → [0,0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{map head (group } \text{xs)})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0,1] → [0,1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0,0] → [0]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
test-based comprehension

\[ \text{Hoog\lambda e} \]

\[
\begin{align*}
\text{Eq } a &= [a] \rightarrow [a] \\
\text{\(\forall x\)} &= \text{concat (group } x\text{)} \\
[0,1] \rightarrow [0,1] &\quad [0,0] \rightarrow [0,0] \\
\text{\(\forall x\)} &= \text{head (group } x\text{)} \\
[0,1] \rightarrow [0] &\quad [0,0] \rightarrow [0,0] \\
\text{\(\forall x\)} &= \text{last (group } x\text{)} \\
[0,1] \rightarrow [1] &\quad [0,0] \rightarrow [0,0] \\
\text{\(\forall x\)} &= \text{map head (group } x\text{)} \\
[0,1] \rightarrow [0,1] &\quad [0,0] \rightarrow [0]
\end{align*}
\]
part III

hoogle+

1. overcoming ambiguity

2. helping beginners with types

Haskell types → function compositions
hoogle+

specification

\[ \text{Eq } a \Rightarrow [a] \rightarrow [a] \rightarrow H^+ \rightarrow \]

programs

1. 
2. 
3. 
4. 
Eq. [a] → H⁺ →

**specification**

**beginner unfriendly!**

**programs**

1. 
2. 
3. 
4. 

hoogle+
can we infer type from tests?

specification

\[ \text{Eq } a \implies [a] \rightarrow [a] \rightarrow \text{H}^+ \rightarrow \]

[1,1,2,1] \rightarrow [1,2,1]

“abba” \rightarrow “aba”

programs

1. ___
2. ___
3. ___
4. ___
types from tests

\[ [1,1,2,1] \rightarrow [1,2,1] \]

“abba” \rightarrow “aba”
types from tests

\[ \text{[Int]} \rightarrow \text{[Int]} \]
ghci
\[ [1,1,2,1] \rightarrow [1,2,1] \]

\[ \text{[Char]} \rightarrow \text{[Char]} \]
ghci
“abba” \rightarrow “aba”
types from tests

Ord a => [a] -> [a]

[Int] -> [Int]
ghci
[1,1,2,1] -> [1,2,1]

[Char] -> [Char]
ghci
“abba” -> “aba”
types from tests

Eq a ⇒ [a] → [a]

Ord a ⇒ [a] → [a]

[Int] → [Int]
 ghci
 [1,1,2,1] → [1,2,1]

[Char] → [Char]
 ghci
 “abba” → “aba”
types from tests

\[ \text{Int} \rightarrow \text{Int} \]

\[ \text{Ord a} \Rightarrow [\text{a}] \rightarrow [\text{a}] \]

\[ \text{Eq a} \Rightarrow [\text{a}] \rightarrow [\text{a}] \]

\[ [\text{a}] \rightarrow [\text{a}] \]

\[ \text{ghci} \rightarrow \text{ghci} \]

\[ [1,1,2,1] \rightarrow [1,2,1] \]

\[ \text{“abba”} \rightarrow \text{“aba”} \]

least common generalization
types from tests

\[ a \rightarrow [b] \]
\[ [a] \rightarrow [b] \]
\[ [a] \rightarrow [a] \]
\[ Eq \ a \Rightarrow [a] \rightarrow [a] \]
\[ Ord \ a \Rightarrow [a] \rightarrow [a] \]
\[ [Int] \rightarrow [Int] \]
\[ ghci \]
\[ [1,1,2,1] \rightarrow [1,2,1] \]

more general types

least common generalization

\[ [a] \rightarrow b \]
\[ \ldots \]

\[ [Char] \rightarrow [Char] \]
\[ ghci \]
\[ “abba” \rightarrow “aba” \]
types from tests

\[ a \rightarrow [b] \]

\[ [a] \rightarrow [b] \]

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

\[ \text{Eq } a \Rightarrow [a] \rightarrow [a] \]

\[ \text{Ord } a \Rightarrow [a] \rightarrow [a] \]

\[ [\text{Int}] \rightarrow [\text{Int}] \]

\[ [1,1,2,1] \rightarrow [1,2,1] \]

\[ [\text{Char}] \rightarrow [\text{Char}] \]

\[ "abba" \rightarrow "aba" \]
types from tests

- \( a \rightarrow [b] \)
- \([a] \rightarrow [b]\)
- \([a] \rightarrow [a]\)
- \(\text{Eq } a \Rightarrow [a] \rightarrow [a]\)
- \(\text{Ord } a \Rightarrow [a] \rightarrow [a]\)
- \([\text{Int}] \rightarrow [\text{Int}]\)
- \([1,1,2,1] \rightarrow [1,2,1]\)
- \([\text{Char}] \rightarrow [\text{Char}]\)
- \(\text{“abba”} \rightarrow \text{“aba”}\)

generalizes over complex type
types from tests

\[ [a] \rightarrow [b] \]

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

Eq \( a \Rightarrow [a] \rightarrow [a] \)

Ord \( a \Rightarrow [a] \rightarrow [a] \)

[Int] \rightarrow [Int]

[Int] \rightarrow [Int]

[1,1,2,1] \rightarrow [1,2,1]

[Char] \rightarrow [Char]

“abba” \rightarrow “aba”
types from tests

\[ [a] \rightarrow [b] \]

unreachable type variable \( b \)

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

\[ \text{Eq } a \Rightarrow [a] \rightarrow [a] \]

\[ \text{Ord } a \Rightarrow [a] \rightarrow [a] \]

\[ [\text{Int}] \rightarrow [\text{Int}] \]

\[ [1,1,2,1] \rightarrow [1,2,1] \]

\[ [\text{Char}] \rightarrow [\text{Char}] \]

“abba” \rightarrow “aba”
types from tests

\[
\begin{align*}
[a] &\rightarrow [a] \\
\text{Eq } a &\Rightarrow [a] \rightarrow [a] \\
\text{Ord } a &\Rightarrow [a] \rightarrow [a] \\
[\text{Int}] &\rightarrow [\text{Int}] \\
[1, 1, 2, 1] &\rightarrow [1, 2, 1] \\
[\text{Char}] &\rightarrow [\text{Char}] \\
\text{“abba”} &\rightarrow \text{“aba”}
\end{align*}
\]
types from tests

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

\textbf{Eq} \ a \Rightarrow [a] \rightarrow [a]

\textbf{Ord} \ a \Rightarrow [a] \rightarrow [a]
part II

hoogle+

H +

Haskell types → function compositions

user study
user study

30 participants
user study

30 participants

4 tasks
user study

30 participants
4 tasks
2 with Hoole, then 2 with Hoolge+
results: completed tasks

- Hoogle: 29 tasks
- Hoogle+: 44 tasks
results: completed tasks

Hoogle: 29
Hoogle+: 44

51% more!
modes of specification

- type only: 19%
- type + test: 39%
- test only: 42%
modes of specification: beginners

- test only: 54%
- type only: 19%
- type + test: 27%
this talk

part I: synquid

Haskell types → refinement types
→ recursive programs

H+

Haskell types → function compositions

part II: hoogle+

simple types

expressive types