synthesizing programs from types

Nadia Polikarpova
Tutorial Fest @ POPL’20
goal: automate programming
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;
        }
    }
}

int main() {
    return 0;
}
How to split a string in Haskell?

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

```haskell
split ' , ' "my, comma, separated, list" \rightarrow ["my", "comma", "separated", "list"]
```

You can implement it like this:

```haskell
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:

```haskell
split '(),' "my,comma,separated,list" \rightarrow ["my", "comma", "separated", "list"]
```

You can implement it like this:

```haskell
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

specification
- examples
- assertions
- natural language
  types!

search

program

program space

…
the future is (almost) here

Hoogle

Char -> String -> [String]

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

ghc Util
why types?

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

specification

Char → String → [String]
simple types

more programmer-friendly
more ambiguous

expressive types

less programmer-friendly
less ambiguous
this tutorial

part I: synquid

refinement types ➔ recursive programs

part II: resyn

resource types ➔ efficient programs

part III: hoogle+

Haskell types ➔ function compositions

simple types ➔ expressive types
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 2016]
part I

synquid

refinement types → recursive programs

1. types as specifications
example: insert into a sorted list

Input:

```
X
xs
```

Output:

```
ys
```
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 synquid**

**specification**

```
? -> S
```

**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)
```
types are specifications

\texttt{Ord}\ a \Downarrow

\texttt{insert :: a } \rightarrow \texttt{ List a } \rightarrow \texttt{ List a}
insert in synquid

specification

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

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

code

\[ ? \]

\[ \text{Nil, Cons, } \leq, \ldots \]
demo: insert

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

specification

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

code

Nil, Cons, ≤, …

Nil
insert in synquid

specification

\[ a \to \text{List} \]  
\[ \text{List} \to \text{Nil, Cons, } \leq, \ldots \]  

code

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

specification

code

a → list a

POWER TO THE TYPES!

ambigious!

Nil

Nil, Cons, ≤, ...
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

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

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

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

lists of nats
demo: insert

http://comcom.csail.mit.edu/demos/#1-insert
refinement types: sorted lists

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

all you need is one simple predicate!
refinement type for insert

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

**specification**

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

**code**

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

http://comcom.csail.mit.edu/demos/#2-sets
part I

synquid

1. types as specifications
2. type-directed search

refinement types → recursive programs
insert in synquid

specification

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

code

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

specification

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

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

Nil, Cons, ≤, ...

components

code

2^{70}

specification
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}}

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

our work: top-down type checking

specification

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

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

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

hopeless: output must always contain x!
rejecting hopeless programs

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

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

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

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

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

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

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

SMT solver: INVALID!
top-down type checking: the hard part

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

\[
\text{insert } x \ xs =
\begin{array}{l}
\text{match } xs \text{ with} \\
\text{Nil } \to \text{Cons } x \ \text{Nil} \\
\text{Cons } h \ t \to \\
\text{Cons } h \ (\text{insert } x \ ??) \\
\end{array}
\]

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

\[
\text{SList } \{v:a \mid h \leq v\}
\]

\[
\text{insert } x \text{ xs } = \\
\begin{align*}
\text{match } \text{xs} \text{ with} \\
\text{Nil} & \to \text{Cons } x \text{ Nil} \\
\text{Cons } h \text{ t} & \to \text{Cons } h \text{ (insert } x \text{ ??)}
\end{align*}
\]

insert :: x:τ → xs:SList τ → SList τ
top-down type checking: the hard part

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

\text{insert } x \text{ } xs =
\text{match } xs \text{ with}
\text{Nil } \rightarrow \text{ Cons } x \text{ Nil}
\text{Cons } h \text{ } t \rightarrow
\text{Cons } h \text{ (insert } x \text{ ??)}
top-down type checking: the hard part

\[ \{ v : a \mid h \leq v \} \]

Constraints:
\[ \forall x, h : h \leq x \]

SMT solver: INVALID!

\[
\text{insert } x \text{ } x s = \\
\text{match } x s \text{ with} \\
\text{Nil } \to \text{ Cons } x \text{ Nil} \\
\text{Cons } h \text{ } t \to \\
\text{Cons } h \text{ } (\text{insert } x \text{ ??})
\]
condition abduction

{v:a | h ≤ v}

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

**specification**

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

**code**

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

http://comcom.csail.mit.edu/demos/#3-nnf
this tutorial

part I: synquid

part II: resyn

part III: hoogle+

H+

Haskell types → function compositions

refinement types → recursive programs

resource types → efficient programs

simple types

expressive types
part II

resyn

resource types → efficient programs
synquid: limitation

specification

refinement
type

components

code

but is it efficient?
demo: inefficient sets

http://comcom.csail.mit.edu/demos/#4-sets-slow
\[ O(|xs| \cdot |ys|) \]

\[ \text{intersect } xs \text{ ys} = \]
\[
\text{match } xs \text{ with}
\quad \text{Nil } \rightarrow \text{ Nil}
\quad \text{Cons } x \text{ xt } \rightarrow
\qquad \text{if } !(\text{member } x \text{ ys})
\qquad \quad \text{then intersect } xt \text{ ys}
\qquad \text{else Cons } x (\text{intersect } xt \text{ ys})
\]

\[ O(|xs| + |ys|) \]

\[ \text{intersect } = \lambda \text{ xs. } \lambda \text{ ys.}
\]
\[
\text{match } xs \text{ with}
\quad \text{Nil } \rightarrow \text{ Nil}
\quad \text{Cons } x \text{ xt } \rightarrow
\quad \text{match } ys \text{ with}
\qquad \text{Nil } \rightarrow \text{ Nil}
\qquad \text{Cons } y \text{ yt } \rightarrow
\qquad \quad \text{if } x < y
\qquad \qquad \text{then intersect } xt \text{ ys}
\qquad \text{else if } y < x
\qquad \qquad \text{then intersect } xs \text{ yt}
\qquad \text{else Cons } x (\text{intersect } xs \text{ ys})\]
part II

resyn

1. resource types

2. resource-guided synthesis

part II

resyn

1. resource types

resource types → efficient programs
synthesizing efficient programs

we have:
find the intersection of two sorted lists

we want:
find the intersection of two sorted lists in linear time

O(|xs| \cdot |ys|)

O(|xs| + |ys|)
find the intersection of two sorted lists in linear time

O(|xs| + |ys|)
synthesizing efficient programs

specification

find the intersection of two sorted lists in linear time

code

MORE POWER TO THE TYPES!
\( re^2 = \text{refinements} + \text{resources} \)

\[
\{ \, v: \text{Int} \mid 0 \leq v \leq 1 \, \}
\]

refinement  potential
$re^2 = \text{refinements + resources}$

$\text{List } \{ v: \text{Int} \mid 0 \leq v \leq 1 \}$

lists of nats with length units of potential
re² type for intersect

\[
\text{intersect} :: \text{xs:SList } \{a \mid | 1\} \rightarrow \\
\text{ys:SList } \{a \mid | 1\} \rightarrow \\
\{v: \text{List } a \mid \text{elems } v = \text{elems } \text{xs} \cap \text{elems } \text{ys}\}
\]
demo: efficient sets

http://comcom.csail.mit.edu/demos/#4-sets-fast
part II

resyn

1. resource types

2. resource-guided synthesis

resource types → efficient programs
rejecting hopeless programs

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

top-down type checking: now with resources!

specification
top-down type checking

\[
\text{intersect} :: \text{xs:SLList } \{a\|1\} \rightarrow \text{ys:SLList } \{a\|1\} \rightarrow \{\text{List a}|\ldots\}
\]

\[
\text{intersect } \text{xs } \text{ys } =
\text{match } \text{xs } \text{with}
\hspace{1em} \text{Nil } \rightarrow \text{Nil}
\hspace{1em} \text{Cons } x \hspace{1em} x\hspace{1em} t \rightarrow
\hspace{2em} \text{if } (!\text{member } x \hspace{1em} \text{ys})
\hspace{3em} \text{then } \text{intersect } x\hspace{1em} t \hspace{1em} \text{ys}
\hspace{3em} \text{else } ??
\]
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

\[
\text{intersect} :: \text{x}s:\text{SList} \{a|1\} \to \text{ys:\text{SList}} \{a|1\} \to \{\text{List a|…}\}
\]

\[
\text{intersect xs ys =}
\]

\[
\text{match xs with}
\]

\[
\text{Nil \to Nil}
\]

\[
\text{Cons x xt \to}
\]

\[
\text{if !(member x ys)}
\]

\[
\text{then intersect xt ys}
\]

\[
\text{else ??}
\]
top-down type checking

\[
\text{intersect} :: \text{xs:SList } \{a||1\} \rightarrow \text{ys:SList } \{a||1\} \rightarrow \{\text{List } a|\ldots\}
\]

\[
\text{intersect } \text{xs } \text{ys} = \\
\begin{align*}
\text{match } \text{xs with} \\
\text{Nil} \rightarrow \text{Nil} \\
\text{Cons } \text{x } \text{xt} \rightarrow \\
\quad \text{if } !\text{(member } \text{x } (\text{ys :: SList } \{a||p\})) \\
\quad \text{then } \text{intersect } \text{xt } (\text{ys :: SList } \{a||q\}) \\
\quad \text{else } ??
\end{align*}
\]
top-down type checking

\[
\text{intersect} :: xS : \text{SLList} \{a\mid |1\} \rightarrow yS : \text{SLList} \{a\mid |1\} \rightarrow \{\text{List} \ a\mid \ldots\}
\]

1. total potential must be partitioned into two uses:

\[
1 = p + q
\]
**top-down type checking**

\[
\text{intersect} :: \text{xs: SList \{a|1\}} \rightarrow \text{ys: SList \{a|1\}} \rightarrow \{\text{List a|...}\}
\]

\[
\text{match xs with}
\]
\[
\text{Nil} \rightarrow \text{Nil}
\]
\[
\text{Cons x xt} \rightarrow
\]
\[
\text{if } \neg\text{(member x (ys :: SList \{a|p\}))}
\text{then intersect xt (ys :: SList \{a|q\})}
\text{else ??}
\]

Constraints: \(\exists p, q: 1 = p + q\)
\[p \geq 1\]
top-down type checking

\[
\text{intersect} :: \text{x}:\text{SList}\{a||1\} \rightarrow \text{ys} : \text{SList}\{a||1\} \rightarrow \{\text{List a}||...\}
\]

\[
\text{intersect } x\text{s } y\text{s} =  \\
\text{match } x\text{s with} \\
\text{Nil } \rightarrow \text{Nil} \\
\text{Cons } x\text{ xt } \rightarrow \\
\text{if } !(\text{member } x \ (\text{ys} :: \text{SList}\{a||p\})) \\
\text{then intersect xt (ys} :: \text{SList}\{a||q\}) \\
\text{else } ??
\]

Constraints: \(\exists p, q: \)

1. \(1 = p + q\)
2. \(p \geq 1\)
3. \(q \geq 1\)
top-down type checking

\[
\text{intersect} :: \text{xs:SList } \{a||1\} \rightarrow \text{ys:SList } \{a||1\} \rightarrow \{\text{List a}|…\}
\]

\[
\text{intersect } \text{x} \text{s } \text{ys }= \\
\text{match } \text{x} \text{s with} \\
\text{Nil }\rightarrow \text{Nil} \\
\text{Cons } \text{x } \text{xt }\rightarrow \\
\text{if } !\text{(member } \text{x } (\text{ys :: SList } \{a||p\})) \\
\text{then } \text{intersect } \text{xt } (\text{ys :: SList } \{a||q\}) \\
\text{else } ??
\]

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+
Haskell types → function compositions

simple types
expressive types
part III

hoogle+

H +

Haskell types → function compositions
synquid/resyn: limitations

specification

hard to write

program
synquid/resyn: limitations

- specification: hard to write
- does not scale
- program
part III

hoogle+

H +

Haskell types → function compositions

1. Scaling to large libraries
2. Overcoming ambiguity

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

\[ (a \rightarrow b) \rightarrow [a] \rightarrow [b] \]

\textbf{Data.List}

\texttt{map \ f \ xs} is the list obtained by applying \texttt{f} to each element of \texttt{xs}, i.e.,

\texttt{map \ f \ [x1, x2, ..., xn] \ = \ [f \ x1, f \ x2, ..., f \ xn]}
hoogle needs synthesis!

No results found!
hoogle+

specification

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

programs

1. 
2. 
3. 
4. 

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

Hoog\lambda+  
\[ n:\text{Int} \to xs:a \to [a] \]

[]  
xs  
take \ n \ xs  
drop \ n \ xs  

ignore arguments of the query!
future work: quality of result

Hoogλe+  

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

Search

take n xs

drop n xs

take n (init xs)  

drop n (tail xs)  

too similar!
future work: quality of result

Hoog\textsc{λ}e+

\begin{align*}
n: \text{Int} & \rightarrow xs: [a] \rightarrow [a] \\
\text{take} & \ n \ xs \\
\text{drop} & \ n \ xs \\
\text{concat} & \ (\text{replicate} \ n \ xs) \\
\text{repeat} & \ (xs \ !! \ n)
\end{align*}

how do I know what these programs do?
**future work: comprehension**

$$n: \text{Int} \rightarrow xs: [a] \rightarrow [a]$$

- **take** $n$ $xs$
  - $2$ $[1,2,3]$ $\rightarrow$ $[1,2]$
  - $3$ $[1]$ $\rightarrow$ $[1]$

- **drop** $n$ $xs$
  - $2$ $[1,2,3]$ $\rightarrow$ $[3]$
  - $3$ $[1]$ $\rightarrow$ $[]$

- **concat** ($\text{replicate} \ n \ xs$)
  - $2$ $[1,2,3]$ $\rightarrow$ $[1,2,3,1,2,3]$
  - $3$ $[1]$ $\rightarrow$ $[1,1,1]$

- **repeat** ($xs$ $!!$ $n$)
  - $2$ $[1,2,3]$ $\rightarrow$ $[3,3,3,3,...]$
  - $3$ $[1]$ $\rightarrow$ error
this tutorial

part I: synquid

part II: resyn

part III: hoole+ H+ (Haskell types → function compositions)

refinement types → recursive programs

resource types → efficient programs

simple types

expressive types