generating programs from types

Nadia Polikarpova
my research
our goal: automate programming
example: insert into a sorted list

Input:

\[
\begin{array}{c}
X \\
xS \\
\end{array}
\]

Output:

\[
\begin{array}{c}
yS \\
\end{array}
\]

1  2  5  7  8
insert in a functional language

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

automates pointer manipulation & memory management

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)
what's next?

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;
        }
    }
}

assembly
C
Haskell
future

automates argument passing & memory access

?
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" → ["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:

```
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'
```
my research

computer science

programming languages (PL)

program synthesis
program synthesis

specification \rightarrow search \rightarrow program

program space
type-driven program synthesis

specification → search → program

type → type-directed

program space
this talk

synquid

synthesis from refinement types

hoogle+

towards a programmer’s assistant for Haskell
this talk

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

?  →  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

Nil, Cons, ≤, …
types are specifications

\[
\text{Ord } a \quad \Downarrow \\
\text{insert} :: a \to \text{List } a \to \text{List } a
\]
insert in synquid

specification

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

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

code

Nil, Cons, ≤, …
insert in synquid

specification

```latex
a \rightarrow \text{List a}
```

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

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

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

\[
\text{match } xs \text{ with } \\
\quad \text{Nil } \rightarrow \text{Cons } x \text{ Nil} \\
\quad \text{Cons } h \, t \rightarrow \\
\quad \quad \text{if } x \leq h \\
\quad \quad \quad \text{then } \text{Cons } x \, xs \\
\quad \quad \quad \text{else } \text{Cons } h \left(\text{insert } x \, t\right)
\]
this talk

1. types as specifications
2. type-directed search
3. case study
**insert in synquid**

**specification**

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

**code**

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

specification

\[
\text{insert} :: x:a \to \\
xs:\text{SList} a \to \\
\{v:\text{SList} a \mid \text{elems } v = \\
\text{elems } xs \cup \{x\}\}
\]
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, \leq, ...
synthesis as search
synthesis as search
synthesis as search
key idea: reject hopeless programs early
reject hopeless programs early

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

specification
reject hopeless programs early

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

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

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

Expected
\[
\{ v : SList a \mid \text{elems } v = \{x\} \}
\]
and got
\[
\{ v : SList a \mid \text{elems } v = \{\} \}
\]
**insert in synquid**

**specification**

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

**code**

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

**components**
this talk

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 \lor c)

\Rightarrow \text{ def.}

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

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

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

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

1. only \neg, \land, \lor

2. \neg only at variables
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
  NAnd :: NNF → NNF → NNF
  NOr :: NNF → NNF → NNF
```

negated?
nff: 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

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

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

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

\[
\text{BoolLiteral } x2 \rightarrow \text{if } x2
\]

\[
\text{then } \text{NOr } (\text{NAtom dummy } x2) (\text{NAtom dummy False})
\]

\[
\text{else } \text{NAnd } (\text{NAtom dummy } x2) (\text{NAtom dummy True})
\]

\[
\text{Var } x16 \rightarrow \text{NAtom } x16 \text{ False}
\]

\[
\text{Not } x20 \rightarrow \text{match } x20 \text{ with}
\]

\[
\text{BoolLiteral } x22 \rightarrow \text{if } x22
\]

\[
\text{then } \text{nnf } (\text{BoolLiteral False})
\]

\[
\text{else } \text{nnf } (\text{BoolLiteral True})
\]

\[
\text{Var } x28 \rightarrow \text{NAtom } x28 \text{ True}
\]

\[
\text{Not } x32 \rightarrow \text{nnf } x32
\]

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

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

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

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

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

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

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

double-neg

De Morgan

46
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
synquad: limitations

specification: hard to write

program: does not scale
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!

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

No results found!
\[(a \rightarrow \text{Maybe } b) \rightarrow [a] \rightarrow \text{Int}\]
future work: quality of result

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

Hoog\lambda e+

\[
[]
\]

\[
\text{xs}
\]

\[
\text{take } n \text{ xs}
\]

\[
\text{drop } n \text{ xs}
\]

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

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

- take \( n \) \( xs \)
- drop \( n \) \( xs \)
- take \( n \) \((\text{init} \; xs)\)
- drop \( n \) \((\text{tail} \; xs)\)

too similar!
future work: quality of result

\[
\text{Hoog\lambda+} \quad \text{let } n \text{ as Int, } xs \text{ as list of } a \text{, then:}
\]

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

- \text{take } n \ xs
- \text{drop } n \ xs
- \text{concat (replicate } n \ xs)\]
- \text{repeat (xs !! n)}

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

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

\[
\begin{align*}
\text{n:} & \text{Int } \rightarrow \text{xs:} [\text{a}] \rightarrow [\text{a}] \\
\text{take n xs} & \\
2 \ [\text{a,b,c}] & \rightarrow [\text{a,b}] \\
3 \ [\text{a}] & \rightarrow [\text{a}] \\
\text{drop n xs} & \\
1 \ [\text{a,b,c}] & \rightarrow [\text{b,c}] \\
3 \ [\text{a}] & \rightarrow [] \\
\text{concat (replicate n xs)} & \\
2 \ [\text{a,b,c}] & \rightarrow [\text{a,b,c,a,b,c}] \\
3 \ [\text{a}] & \rightarrow [\text{a,a,a}] \\
\text{repeat (xs !! n)} & \\
1 \ [\text{a,b,c}] & \rightarrow [\text{b,b,b,b,…}] \\
3 \ [\text{a}] & \rightarrow \text{error}
\end{align*}
\]
thank you!

synquid

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

hoogle+

http://goto.ucsd.edu/hoogle_plus