type-driven program synthesis

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

Input:

\[ x \]
\[ xs \]

\[ 1 \ 2 \ 7 \ 8 \]

Output:

\[ y_s \]

\[ 1 \ 2 \ 5 \ 7 \ 8 \]
insert in a functional language

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

**what's next?**

automates argument passing & memory access
this talk

I. specifications
   are the new cool language feature

II. program synthesis
    is the way to make it happen

III. synthesis-aided programming
    is the future
this talk

I. specifications

II. program synthesis

III. synthesis-aided programming
this talk

I. specifications

how do I tell the machine what I want?
specification for insert

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

Output:
\[ y_s: \text{sorted list} \]
\[
\text{elems } y_s = \text{elems } x_s \cup \{x\}
\]
easy to write ☺
hard to execute ☹
program synthesis

**specification**

**in:** $x$, sorted $xs$
**out:** sorted $ys$
$\text{elems } ys = \text{elems } xs \cup \{x\}$

**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)
```
program synthesis

specification

in: x, sorted xs
out: sorted ys
elems ys = elems xs ∪ \{x\}

Synquid

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 synthesis

specification

in: x, sorted xs
out: sorted ys
elems ys = elems
xs U \{x\}

formalize this

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

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

\[ \text{Ord } a \quad \downarrow \]

\textbf{insert} :: \( a \rightarrow \text{List } a \rightarrow \text{List } a \)

ordinary types are insufficient!
refinement types

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

shape \hspace{1cm} \text{refinement}
from lists to sorted lists

data List a  where
  Nil :: List a
  Cons :: h:a →
    t:List a →
    List a
from lists to 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!
insert in Synquid

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

Synquid generates correct code in under a second!
this talk

I. specifications

II. program synthesis

III. synthesis-aided programming
this talk

II. program synthesis

how do we get from specification to code?
type-driven program synthesis

**Specification**

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

**Code**

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

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

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

Nil, Cons, ≤, ...

components

code

2^{70}

specification
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 U {x}}
reject hopeless programs early

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

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

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

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

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

\[ \text{insert } x \text{ } xs = \]
\[
\text{match } xs \text{ with}
\]
\[
\text{Nil } \rightarrow \text{Nil}
\]
\[
\text{Cons } h \text{ } t \rightarrow \]
\[
2^{50}
\]
reject hopeless programs early

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

\text{insert } x \ xs =
\begin{align*}
\text{match } xs \text{ with} \\
\text{Nil } & \rightarrow \text{Nil} \\
\text{Cons } h \ t & \rightarrow ??
\end{align*}
reject hopeless programs early

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

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

{v : SList a | \text{elems } v = \{x\}}

insert \ x \ xs =
\begin{align*}
\text{match } \ xs \ \text{with} \\
\Nil & \rightarrow \Nil \\
\text{Cons } h \ t & \rightarrow ???
\end{align*}

Expected
{v : SList a | \text{elems } v = \{x\}}
and got
{v : SList a | \text{elems } v = \{\}}
specification

```haskell
insert :: x:a \to
xs:SList a \to
\{v:SList a \mid \text{elems } v =\}
\text{elems } xs \cup \{x\}
```

code

```haskell
match xs with
  Nil \to \text{Cons } x \text{ Nil}
  Cons h t \to
    \text{if } x \leq h
    \text{then } \text{Cons } x \text{ xs}
    \text{else } \text{Cons } h (\text{insert } x t)
```
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

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

\[ \Rightarrow \text{ def.} \]
\[ \neg(\neg a \lor (b \lor c)) \]

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

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

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

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
```

data NNF where

```
    NAtom :: String -> Bool -> NNF
    NAnd :: NNF -> NNF -> NNF
    NOr :: NNF -> NNF -> NNF
```
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

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

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

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

\[
\text{nnf} :: f : \text{Fml} \rightarrow \{v: \text{NNF} \mid \text{nEval } v = \text{eval } f\}
\]
\[
nnf p = \text{match } p \text{ with
}
\text{BoolLiteral } x2 \rightarrow \text{ if } x2
\hspace{1cm} \text{then } \text{NOr} (\text{NAtom dummy } x2) (\text{NAtom dummy False})
\hspace{1cm} \text{else } \text{NAnd} (\text{NAtom dummy } x2) (\text{NAtom dummy True})
\]
\[
\text{Var } x16 \rightarrow \text{NAtom } x16 \text{ False}
\]
\[
\text{Not } x20 \rightarrow \text{ match } x20 \text{ with
}
\text{BoolLiteral } x22 \rightarrow \text{ if } x22
\hspace{1cm} \text{then } \text{nnf} (\text{BoolLiteral False})
\hspace{1cm} \text{else } \text{nnf} (\text{BoolLiteral True})
\]
\[
\text{Var } x28 \rightarrow \text{NAtom } x28 \text{ True}
\]
\[
\text{Not } x32 \rightarrow \text{nnf } x32
\]
\[
\text{And } x36 \ x37 \rightarrow \text{NOr} (\text{nnf } (\text{Not x36})) (\text{nnf } (\text{Not x37}))
\]
\[
\text{Or } x46 \ x47 \rightarrow \text{NAnd} (\text{nnf } (\text{Not x46})) (\text{nnf } (\text{Not x47}))
\]
\[
\text{Implies } x56 \ x57 \rightarrow \text{NAnd} (\text{nnf } x56) (\text{nnf } (\text{Not x57}))
\]
\[
\text{And } x65 \ x66 \rightarrow \text{NAnd} (\text{nnf } x65) (\text{nnf } x66)
\]
\[
\text{Or } x73 \ x74 \rightarrow \text{NOr} (\text{nnf } x73) (\text{nnf } x74)
\]
\[
\text{Imp } x81 \ x82 \rightarrow \text{NOr} (\text{nnf } x82) (\text{nnf } (\text{Not x81}))
\]

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

double-neg

De Morgan
this talk

I. specifications

II. program synthesis

III. synthesis-aided programming
this talk

how can synthesis help us build software?

III. synthesis-aided programming
the future of programming

Let there be Instagram!
the future of programming (not)
orthogonal concerns

I worry about all concerns together 😞
synthesis-aided programming

- functional requirements
- data invariants
- security policies
- resource constraints

specification

I worry about one concern at a time 😊
synthesis weaves them together
synthesis-aided programming

- functional requirements
- security policies
- specification
information leaks

electronic health records

social networks

e-commerce
information leaks

policy checks

social networks
information leaks

This Is Almost Certainly James Comey’s Twitter Account

Ashley Feinberg
3/30/17 3:29pm • Filed to: JAMES COMEY

Reinhold Niebuhr
@projected87
Joined February 2014

Likes

Who to follow: @manual: View all

United States Presid: @WhiteHouse

In South Africa, immigration leaves corrupt officials and new state
Many migrants to South Africa say they are ripe to corruption and violence.
One Nigerian is now suing the authorities for damages

James Comey
Brian Comey
Ashley Feinberg

social networks
information security with Lifty

program

security policies

Lifty

program with checks
synthesis-aided programming

functional requirements

resource constraints

specification
timing attacks

Compare the lengths of two strings:

password

attempt

execution time reveals length!
resource-aware programming with ReSyn

\[
\text{password} \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1
\]

\[
\text{attempt} \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1
\]

\[
\text{compare} :: \text{pass:List a} \rightarrow \text{att:List } \{a \mid \mid 1\} \rightarrow \{v: \text{Bool} \mid v = (\text{len key} = \text{len guess})\}
\]
synthesis-aided programming