type-driven program synthesis

Nadia Polikarpova
goal: automate programming
program synthesis

specification

search

program

program space
program synthesis

specification

search

program space

program
program synthesis

specification
examples
assertions
natural language
...

search

program

program space
what makes a good spec?

1. programmer-friendly
easier to write than the program

2. informative
minimal ambiguity

3. synthesizer-friendly
easy to check, guides the search
how about types?

1. programmer-friendly
   widely used for bug finding, API search, etc
   concise

2. informative
   precise
   advanced types can express many properties

3. synthesizer-friendly
   type checking is automatic and compositional
this talk

1. synquid
   recursive program synthesis from refinement types

2. hoogle+
   component-based synthesis from Haskell types

3. future work
   best types for synthesis?

simple types  expressive types
this talk

1. synquid recursive program synthesis from refinement types

2. hoogle+ component-based synthesis from Haskell types

3. future work best types for synthesis?

simple types expressive types
example: insert into a sorted list

input:
x
xs

output:
ys
program synthesis

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

formalize this

match xs with
  [] → x : []
h : t →
  if x ≤ h
  then x : xs
  else h : (insert x t)

[], :, ≤
program synthesis

specification

= type

a → [a] → [a]

code

match xs with
  [] → x : []
  h : t →
    if x ≤ h
      then x : xs
    else h : (insert x t)
program synthesis

specification

= type

ambiguous!

a → [a]  →  code  →  []
program synthesis

specification

= type

ambiguous!

POWER TO THE TYPES!

synquid

[a]

[]

code
refinement types

\{ \text{shape} \mid v : \text{Int}, 0 \leq v \}
insert in synquid

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

sorted list

set of elements
**synquid**

**specification**

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

**code**

\[
\text{match } x : S \text{ with} \\
[] \to x : [] \\
h : t \to \\
\text{if } x \leq h \\
\text{then } x : x : s \\
\text{else } h : (\text{insert } x : t)
\]

\[
[], :, \leq
\]
what else can it do?
limitation

sometimes writing a (sufficiently) complete refinement type is harder than writing the program

POWER TO THE TYPES???
this talk

1. synquid
recursion program synthesis
from refinement types

2. hoole+
component-based synthesis
from Haskell types

3. future work
best types for synthesis?

simple types
expressive types
example: first non-empty

input:

\[
\begin{align*}
\text{xs} & \quad \begin{array}{c}
\text{\hspace{1cm}} \\
5 \\
2 \\
3
\end{array} \\
\text{d} & \quad 0
\end{align*}
\]

output:

\[
\begin{align*}
5
\end{align*}
\]

is there a library function for that?

ask Hoogle!
example: first non-empty

type query: \( \text{xs} : \text{[Maybe a]} \to \text{d : a} \to \text{a} \)

solution: \( \langle \text{xs} \, \text{d} \rangle \to \text{fromMaybe d (listToMaybe (catMaybes \text{xs}))} \)
example: first non-empty

type query: \[ \text{xs: [Maybe } a \text{] } \rightarrow \text{ d: } a \rightarrow a \]
solution: \[ \text{\( \text{\( xs \quad d \rightarrow \) fromMaybe } d \quad (\text{listToMaybe } (\text{catMaybes } \text{xs})) \) } \]

wanted: component-based synthesis

Hoogle+ to the rescue!
Hoogle+

Haskell libraries

```
[Maybe a] → a → a
```

search

programs
Hoogle+
prior work: graph-based search

[Sypet: Feng et al, POPL’17]

Petri net:

- **fromMaybe**
- **listToMaybe**
- **catMaybes**
- **[Maybe a]**
- **[a]**

Type query:

\[ [Maybe \ a] \to a \to a \]
prior work: graph-based search

Petri net:

```

Maybe (Maybe a) -> fromMaybe

listToMaybe

[Maybe a] -> catMaybe
```

```

Maybe a -> fromMaybe

listToMaybe

[a] -> a

fromMaybe
```

Type query:

```
[Maybe a] -> a -> a
```

[Sypet: Feng et al, POPL’17]
prior work: graph-based search

Petri net:

- fromMaybe: \( \text{Maybe (Maybe a)} \rightarrow \text{Maybe a} \)
- listToMaybe: \( \text{Maybe a} \rightarrow [\text{Maybe a}] \)
- catMaybes: \( [\text{Maybe a}] \rightarrow [\text{a}] \)
- fromMaybe: \( [\text{a}] \rightarrow \text{a} \)

type query:

\( [\text{Maybe a}] \rightarrow \text{a} \rightarrow \text{a} \)
prior work: graph-based search

Petri net:

- `Maybe (Maybe a)`
- `fromMaybe` (from `Maybe a` to `a`)
- `listToMaybe`
- `catMaybees`
- `Maybe [a]` to `a`
- `a`
- `fromMaybe` (from `[a]` to `Maybe a`)

Type query:

```
[Maybe a] → a → a
```
challenge: polymorphism

1. infinitely many types

... [[[a]]] [[a]] [a] a
challenge: polymorphism

1. infinitely many types
2. transition explosion
solution: abstraction

```
listToMaybe :: [τ] -> Maybe τ
```

- any type
- list of any type
solution: abstraction

abstract transition net:
results

spent the whole time building the net
Hoople+

- type query
  - \([\text{Maybe ambiguous!}] \rightarrow a\)
  - \(\text{H}^+\)
- search
  - \(\text{H}^+\)
- programs

? can we filter out irrelevant results without extra input from the user?
three kinds of irrelevant results

type query: $\text{xs} : [(\text{Maybe } a)] \rightarrow \text{d} : a \rightarrow a$

1. discard arguments
   $\text{xs} \rightarrow \text{d}$
   polymorphism makes this worse

2. always crash
   $\text{xs} \rightarrow \text{foldr} (\text{head } []) \text{ d} \text{ xs}$

3. duplicates
   $\text{xs} \rightarrow \text{fromMaybe} \text{ d} (\text{head} \text{ xs})$
   $\text{xs} \rightarrow \text{maybe} \text{ d} \text{ id} (\text{head} \text{ xs})$
test-based filtering

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?
substructural types

variable use

structural
unrestricted

relevant
at least once

linear
exactly once

\( \forall x \ d \rightarrow \ d \)

\( \forall x \ d \rightarrow \text{bool} \ d \ d \)
  
  \((\text{null} \ x)\)

\( \forall x \ d \rightarrow \text{fromMaybe} \ d \)
  
  \((\text{listToMaybe} \)
    
  \((\text{catMaybes} \ x)\))

- definitely useless
- likely useless
search with substructural types

Petri net = linear types!

Petri net + copy transitions = relevant types + linear bias
this talk

1. synquid
   recursive program synthesis
   from refinement types

2. hoogle+
   component-based synthesis
   from Haskell types

3. future work
   best types for synthesis?
future work

specification

type

refinements?
examples?
natural language?

more components

H++

programs

more complex programs
more relevant results
(advanced) linear types

can linear types help synthesize more complex programs?

example: concatenate a list of lists

\[ \text{concat} :: \text{xs} : \text{[[a]]} \to [a] \]

sufficiently complete if interpreted as a linear type!

non-linear use requires annotations

\[ \text{replicate} :: n : \text{Nat} \to x : n^*a \to [a] \]
partial refinement types

refinement types are great... until they aren’t

eexample: are all elements of this list negative?

\[
\text{allNeg} :: \text{xs} : [\text{Int}] \rightarrow \text{Bool}
\]

partial refinements:

\[
\text{allNeg} :: \text{xs} : [\text{Neg}] \rightarrow \{\nu : \text{Bool} | \nu\}
\]
\[
\text{xs} : \{\nu : [\text{Nat}] | \text{len } \nu > 0\} \rightarrow \{\nu : \text{Bool} | \neg \nu\}\]
type-driven program synthesis

1. synquid
   recursive program synthesis from refinement types

2. hoogle+
   component-based synthesis from Haskell types

3. future work
   linear types
   partial refinement types

simple types               expressive types