Ideas for Automatic Programming-Task Generation Based on Generating Sample Code

Oliver Westphal **Janis Voigtländer**

University of Duisburg-Essen

Handwritten Tasks

Usual Haskell IO task we give our students:

```haskell
{- Write a program which first reads a positive
 - integer n from the console, and then reads n
 - integers one after the other and finally outputs
 - their sum.
 -}
main :: IO ()
main = undefined
```

- Underlying specification of intended behavior
- Automatic checking of solutions against appropriate random inputs
- Currently: Description and behavior are handwritten

Automatic Task Generation

Every task has two components:

- A goal or solution requirement
- A description communicating that goal

How to generate both automatically?

Automatic Task Generation

Every task has two components:

- A goal or solution requirement
- A description communicating that goal

How to generate both automatically?

- Specification language to express requirements
- Ideally: Generate verbal description from specification
- Here: Communicate requirements through generated program code

Task Example

```haskell
{- Give the output of the following
 - program for input 7!
 -}
main = do
  v <- readLn
  let loop 1 = print 1
      loop x = do
        print x
        if even x
          then loop (x div 2)
          else loop (x + 1)
  loop v
```

- Task description independent from expected solution
- Lots of possibilities for (generating) tasks

Specifying Task Goals

Specification Language

- Read values and storing them in list-valued variables
- Output result of computation over variables
- Access variables either as complete list or last read value
- Basic branching and iteration

$$[\triangleright n]^{\mathbb{N}} \cdot ([\triangleright x]^{\mathbb{Z}} \measuredangle len(x_A) = n_C \succ \mathbf{E})^{\rightarrow^{\mathbf{E}}} \cdot [sum(x_A) \triangleright]$$

$$\updownarrow$$

"Read a positive integer n from the console, and then read n integers one after the other and finally output their sum."

Task Descriptions

Code as Description

Advantages:

- More precise than verbal description
- Easily understandable if kept simple
- Can be generated/transformed automatically

Disadvantages:

- Might already be a valid solution
- Requires knowledge of respective programming language
- No verbal description only tasks

Generating Programs

Direct Interpretation

- Idea: Build programs based on an interpreter for specifications

- ```
  [[·]] :: Specification -> Semantics ()
  [[[▷x]^τ]]    = do v <- readLn
                     modify (store v x)
  [[[{t}▷]]]    = print =<< gets (evalTerm t)
  [[s →^E]]     = let loop = do [[s]]
                                loop
                  in catchError loop (\Exit -> return ())
  [[E]]         = throwError Exit
  [[s1 ∠c∠ s2]] = ifM (gets (evalTerm c)) [[s2]] [[s1]]
  [[s1 · ... · sn]] = do {[[s1]]; ... ;[[sn]]}
    ...
  newtype Semantics a = Semantics { runSemantics ::
    Environment -> IO (Either Exit a, Environment) }
  ```
  ```

Direct Interpretation - Example

- For: $[\triangleright n]^{\mathbb{N}} \cdot ([\triangleright x]^{\mathbb{Z}} \measuredangle len(x_A) = n_C \measuredangle \mathbf{E})^{\rightarrow^{\mathbf{E}}} \cdot [sum(x_A) \triangleright]$
- Correct, but confusing for students

```
p :: IO ()
p = void $ runSemantics p' [("n",[]),("x",[])]
p' :: Semantics ()
p' = do
  v <- readLn
  modify (store v "n")
  let loop = do ifM (gets (evalTerm (len(x_A) = n_C)))
                    (throwError Exit)
                    (do v <- readLn
                        modify (store v "x"))
                loop
  catchError loop (\Exit -> return ())
  print =<< gets (evalTerm (sum(x_A)))
```

Direct Interpretation

- Haskell's encapsulation of side-effects leads to verbose programs
- Contrast this with a direct interpretation in Python:

```
n_A = []
x_A = []

n_A += [int(input())]
while True:
  if len(x_A) == n_A[-1]:
    break
  else:
    x_A += [int(input())]
print(sum(x_A))
```

- Same principle but much more readable

Ongoing Work

- Improve generation of idiomatic Haskell programs
- Ways to obtain different “solutions” from the same specification
- Possible directions:
  - Use GHC pipeline to simplify programs (inlining etc.)
  - Custom static analysis
  - Rewrite based on common patterns
  - Template Haskell
  - Intermediate language

Tasks Ideas

Assumptions

- We can translate specifications into several different program representations
- $[\triangleright n]^{\mathbb{N}} \cdot ([\triangleright x]^{\mathbb{Z}} \measuredangle len(x_A) = n_C \trianglerighteq \mathbf{E})^{\rightarrow^{\mathbf{E}}} \cdot [\{sum(x_A)\} \triangleright]$ could yield these programs (or a Python version thereof)

```haskell
p1 = do
  n <- readLn
  let loop xs =
        if length xs == n
          then return xs
          else do
            v <- readLn
            loop (xs ++ [v])
  xs <- loop []
  print (sum xs)
```

```haskell
p2 = do
  n <- readLn
  loop 0 n

loop s m =
  if m == 0
    then print s
    else do
      v <- readLn
      loop (s + v) (m - 1)
```

Assumptions

- Additionally we assume we can generate random specifications
- Control complexity of generated specifications
- Mutate a base specification into several related but different specifications

Basic Task Recipe

1. Take a base specification
2. Derive multiple artifacts like
   - Progam representation(s)
   - Execution traces
3. Give students some of these artifacts
4. Ask them to “recreate” the others

These task are also automatically checkable!

Task Types

- Three types of tasks, where students have to
  - make a decision
  - give execution traces of some program
  - write a program
- Testing the ability to read, understand and produce programs
- We currently use only the third type

Task Type 1:
Make a Decision

Example

```
{- Which execution traces can the given program produce?
 - Here ?u stands for reading u and !v for printing v
 - a) ?2 ?7 ?4 !11
 - b) !3 ?6 ?2 ?9 !17
 - c) ?4 ?15 ?(-3) ?8 ?1 !20
 -}
main = do
  n <- readLn
  loop 0 n

loop s m =
  if m == 0
    then print s
    else do
      v <- readLn
      loop (s + v) (m - 1)
```

Other Options

- **Given:** Two programs
  Task: Do these programs have the same behavior? (Yes/No)
- **Given:** Program with compiler error(s).
  Task: Mark all lines that contain errors.

  E.g.: Introduce errors with hand-designed rules that resemble common student mistakes.

Task Type 2:
Give Execution Traces

Example

```
{- Find an input sequence for which the two programs
 - below behave differently.
 -}
p1 = do
  n <- readLn
  let loop xs =
        if length xs == n
          then return xs
          else do
            v <- readLn
            loop (xs ++ [v])
  xs <- loop []
  print (sum xs)
```

```
p2 = do
  n <- readLn
  loop 0 n

loop s m = do
  v <- readLn
  if m == 0
    then print s
    else
      loop (s + v) (m - 1)
```

Other Options

- **Given:** Some program
  Task: Give an execution trace that the program can produce

  Possible to impose additional conditions (maximizing coverage etc.)

Task Type 3:
Write a program

Example

```python
n_A = []
x_A = []

n_A += [int(input())]
while True:
  if len(x_A) == n_A[-1]:
    break
  else:
    x_A += [int(input())]
print(sum(x_A))
```

```haskell
{- Write a Haskell program that has the same behavior as
 - as the given Python program.
 -}
main :: IO ()
main = undefined
```

Example

Alternatively: Force solution style

```haskell
{- Write a Haskell program that has the same behavior as
 - as the given Python program.
 -}
main :: IO ()
main = do
  n <- readLn
  let loop s m = undefined
  loop 0 0
```

Only allow replacing `undefined`

Other Options

- **Given:** Haskell program using (not using) a certain feature or abstraction
  Task: Rewrite the program such that is does not (does) use that feature or abstraction
- **Given:** Any full or partial program
  Task: Change/complete the program such that it
  1. can produce a specified set of traces.
  2. matches a reference program (Haskell or otherwise).
  3. simply compiles.

Conclusion

- The combination of fixed verbal description + program code offers lots of interesting ideas for exercise tasks
- Being able to generate the necessary ingredients opens up a huge range of automatically generatable exercises.
- It is definitely worthwhile to explore actually creating these
- We are open to suggestions on how this can be done best