A Framework for Generating Diverse Haskell-I/O Exercise Tasks

Oliver Westphal

University of Duisburg-Essen

Handwritten Task

```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
```

Fixed Verbal Task Descriptions

```haskell
{- Give the output of the following
 - program for input 9!
 -}
main :: IO ()
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
```

Program trace: ?9 !9 !10 !5 !6 !3 !4 !2 !1 stop

Fixed Verbal Task Descriptions

```
{- Give the output of the following
 - program for input 9!
 -}
main :: IO ()
main = do
  v <- readLn
  let loop x
        | x < 4 = print x
        | odd x = do
          print x
          loop (x+3)
        | otherwise = do
          print x
          loop (x `div` 4)
  loop v
```

Program trace: ?9 !12 !3 stop

Why Haskell-I/O Tasks?

Specification Language [TFPIE 2019]

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

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

$$\updownarrow$$

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

Specifications in Haskell

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

$$\updownarrow$$

```
example :: Specification
example =
  readInput "n" nats <>
  tillExit (
    branch (length (getAll "x") == getCurrent "n")
    (readInput "x" ints)
    exit
  ) <>
  writeOutput [var o] [sum (getAll "x")]
```

Implementation probabilistically checks student programs against specifications

Task Generation

Automatic Task Generation

Two components for a task:

- Goal or solution requirement
- Description communicating that goal

How to generate both automatically?

Automatic Task Generation

Two components for a task:

- Goal or solution requirement
- 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 or example behavior

Code as Description

Advantages:

- More precise than verbal description
- Easily understandable if kept simple
- Automatic generation & transformation possible

Disadvantages:

- Might already be a valid solution
- Knowledge of respective programming language required
- No tasks with only verbal description

Task from Example Behavior

```haskell
{- Write a program capable of these interactions:
 - ?0 !0 stop
 - ?1 ?-3 !-3 stop
 - ?2 ?1 ?5 !6 stop
 - ?2 ?10 ?10 !20 stop
 - ?2 ?-3 ?-2 !-5 stop
 -}
main :: IO ()
main = undefined
```

Behavior as Description

Advantages:

- No leaking of program structure

Disadvantages:

- No longer requires exact behavior of specification
- Hard-coding cases possible
- Might include (or not include) corner cases

Basic Task Recipe

1. Take a base specification
2. Derive artifacts:
   - Progam representation(s)
   - Execution traces
3. Build a question/task description from these artifacts

Solution candidates are also automatically checkable!

Task Types

- Three types of tasks possible:
  - make a decision (given both code and behavior)
  - give behavior (given a program)
  - write a program (given behavior or another program)
- Roughly corresponding to program-reading, -understanding and -writing abilities

An EDSL for Task Generation

Basic Setup

```haskell
data TaskInstance s = TaskInstance
  { question :: Description
  , requires :: Require s }

newtype Require s = Require
  { check :: s -> Property }

data TaskDesign p s = TaskDesign
  { parameter :: Gen p
  , inst :: p -> Gen (TaskInstance s) }

genTaskInstance :: TaskDesign p s -> Gen (TaskInstance s)
genTaskInstance task = do
  p <- parameter task
  inst task p
```

Combinators

```haskell
forFixed :: p -> (p -> Gen (TaskInstance s))
         -> TaskDesign p s
forFixed p = TaskDesign (pure p)

forUnknown :: Gen p -> (p -> Gen (TaskInstance s))
           -> TaskDesign p s
forUnknown g i = TaskDesign g i

solveWith :: Description -> Require s -> TaskInstance s
solveWith d r = TaskInstance d r

exactAnswer :: (Eq a, Show a) => a -> Require a
exactAnswer x = Require $ \s -> s === x
```

Primitives for I/O Tasks

Given from specification language's implementation:

```
fulfills :: Program -> Specification -> Property
accept :: Specification -> Trace -> Bool
```

Task goals:

```
randomSpecification :: Gen Specification
similarSpecifications :: Gen (Specification,Specification)
```

Requirements:

```
behavior :: Specification -> Require Program
sampleTrace :: Specification -> Require Trace
triggeringDifference :: Specification -> Specification
                     -> Require [Input]
```

Primitives for I/O Tasks

For descriptions:

```
type Code = Description

haskellProgram :: Specification -> Gen Code
pythonProgram :: Specification -> Gen Code
exampleTraces :: Specification -> Int -> Gen [Trace]
multipleChoice :: Show a => Int -> [a] -> [a]
               -> Gen (Description, [Int])
```

Primitives for I/O Tasks

For descriptions:

```
type Code = Description

haskellProgram :: Specification -> Gen Code
pythonProgram :: Specification -> Gen Code
exampleTraces :: Specification -> Int -> Gen [Trace]
multipleChoice :: Show a => Int -> [a] -> [a]
               -> Gen (Description, [Int])
```

DISCLAIMER!

- Implementation of these primitives is still work in progress
- A basic prototype exists
- Needs a solid foundation and to be scaled up
- Hence, focus on possibilities in task design

Examples

Write A Program - Design

```
task :: TaskDesign Specification Program
task = forUnknown randomSpecification $ \s -> do
  prog <- pythonProgram s
  return $
  ("Re-implement the given Python program in Haskell:"
  $$ prog
  ) `solveWith` behavior s
```

Write A Program - Instance

```
n = int(input())
x = []
while len(x) != n :
  v = int(input())
  x += [v]
print(sum(x))
-- Re-implement the given Python program in Haskell.
main :: IO ()
main = undefined
```

randomized

Make a Decision - Design

```haskell
task :: TaskDesign (Specification,Specification) [Int]
task = forUnknown similarSpecifications $
  \(spec1,spec2) -> do
    p <- haskellProgram spec1
    ts1 <- exampleTraces spec1 2
    ts2 <- exampleTraces spec2 2
    (choices, solution) <- multipleChoice 3 ts1 ts2
    return $
      ("Which of these traces can this program produce?"
       $$ p
       $$ choices
      ) `solveWith` exactAnswer solution
```

Make a Decision - Instance

```
-- Which of the given traces can the program below produce?
-- 1) ?"-6" ?"-9" ?"10" ?"-3" ?"7" !"-1" stop
-- 2) ?"3" ?"-3" ?"-2" ?"6" !"1" stop
-- 3) ?"1" ?"-6" !"-6" stop
prog :: IO ()
prog = do
 n <- readLn
 let loop1 x1 =
       if (length x1 == n)
         then do return x1
         else do
           v1 <- readLn
           loop1 (x1 ++ [v1])
 x3 <- loop1 []
 print (sum x3)
```

randomized

randomized

Give Execution Traces - Design

```haskell
task :: TaskDesign (Specification,Specification) [Input]
task = forUnknown similarSpecifications $
  \(spec1,spec2) -> do
    p1 <- haskellProgram spec1
    p2 <- haskellProgram spec2
    return $
      ("Give a sequence of input values for which the two"
       ++ "programs below behave differently!"
       $$ p1
       $$ "---"
       $$ p2
      ) `solveWith` triggeringDifference spec1 spec2
```

Give Execution Traces - Instance

```haskell
{- Give a sequence of input values for which the two
 - programs below behave differently.
 -}
```

```haskell
prog1 :: IO ()
prog1 = do
  n <- readLn
  let loop x =
        if (length x == n)
          then do return x
          else do
            v <- readLn
            loop (x ++ [v])
  y <- loop []
  print (sum y)
```

randomized

```haskell
prog2 :: IO ()
prog2 = do
  let loop x l =
        if (l == 5)
          then do return x
          else do
            v <- readLn
            loop (x ++ [v])
                 (l + 1)
  y <- loop [] 0
  print (sum y)
```

randomized

Conclusion & Future Work

- Lots of interesting ideas for exercise tasks based on program code and/or example behavior
- Tasks differ from (traditional) handwritten tasks
- Variety depends on generating interesting specifications/programs/examples
- Different task types need to be evaluated with regard to usefulness to students

Conclusion & Future Work

- Lots of interesting ideas for exercise tasks based on program code and/or example behavior
- Tasks differ from (traditional) handwritten tasks
- Variety depends on generating interesting specifications/programs/examples
- Different task types need to be evaluated with regard to usefulness to students
- Source-code and examples:
  `https://github.com/fmidue/IOTasks`
- Demo (hand-written tasks):
  `https://autotool.fmi.iw.uni-due.de/spec-demo`