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

Oliver Westphal
University of Duisburg-Essen
oliver.westphal@uni-due.de

Abstract. We present the design of a framework to describe parametrized exercise tasks on Haskell-I/O programming. Parametrized tasks can be instantiated randomly to quickly generate different instances of a task. Such automatic task generation is useful in many different ways. Manual task creation can be a time-consuming process, so formulating a task design once and then automatically generating different variations can save valuable time for the educator. The descriptions of tasks also serve as easy to understand documentation and can be reused in new task designs. On the student’s side automatic task generation, together with an automated assessment system, enables practicing on as many fresh exercise tasks as needed. Students can also each be given a slightly different version of tasks, reducing issues regarding plagiarism arising naturally in an e-learning environment. Our task generation is centered around a specification language for I/O behavior we developed in earlier work. The task generation framework, an embedded domain specific language in Haskell, provides powerful primitives for the creation of various artifacts from specifications, including program code. We do not go into detail on the technical realization of these primitives. Our focus is on showcasing how such artifacts can be used as an alternative to the verbal description of requirements for different types of programming exercise tasks. By doing so, we are able to automatically generate a diverse range of task designs.

1 Introduction

We have recently designed and implemented a language for specifying console I/O programs [15,16] allowing us to formulate desired I/O behavior. The I/O behavior of programs, written in Haskell, can be tested probabilistically against specified behavior. We built this language to bring our testing capabilities of tasks on Haskell I/O more in line with how one can test tasks on pure programs, for example, using QuickCheck [1]. These automatic testing capabilities are used in the e-learning system [8,13] we use in our course on programming paradigms [9].

We now also aim to automatically generate the tasks themselves. This has a variety of advantages. Automatic task generation can help educators to create different variations of a common exercise task idea much quicker. When combined
with automated assessment students have the opportunity to practice with as
many fresh exercise tasks as they need. Automatically generated tasks can also
be used to reduce plagiarism issues by giving students slight variations of the
same task. We will (only) present automatic generation for tasks on Haskell I/O
in this work, since our specification language is designed to describe the I/O
behavior of programs. However, we believe our approach can be adapted for
other types of exercise tasks as well.

Hand-written (programming) exercise tasks usually rely heavily on verbal
descriptions. For example we might pose a task like this:

“Read a positive integer $n$ from the console. Then, read $n$ integers one after the
other. Finally, output their sum.”

Such verbal descriptions are a big problem when trying to generate tasks auto-
matically, as natural language generation is not exactly easy. Because of this,
many task generation systems rely on templates defining a fixed (verbal) frame
for a task. Such templates contain gaps to be filled to form a concrete task. Dif-
ferent (randomized) choices to fill these gaps result in different task variations.
Depending on the domain for which tasks are generated, writing such a fixed
framing, can be difficult. For example, it is easy for many different math tasks,
where the verbal frame can be something like “Solve for $x$.” together with a
randomized equation. For programming tasks finding a fixed and general verbal
frame is more difficult. Take, for example, the verbal description from above. We
could use a fixed verbal skeleton like “First ... Then ... Finally” and fill it with
random predefined descriptions. However, this is not a very flexible approach.
Instead, our approach uses artifacts like program code or example runs of a
program to achieve a fixed descriptions. Take, for example, the following task:

```
Give the programs interaction trace for input sequence 2, 4, 9.

```

```
prog :: IO ()
prog = do
  n <- readLn
  let loop s l =
    if l == n then print s
    else do
      v <- readLn
      loop (s + v) (l + 1)
  loop 0 0
```

The verbal description does not need to change, apart from maybe the given
input sequence, no matter what we give as the program text. However, a solution
to this task is fundamentally different than in the previous example. Instead of
requiring a correctly behaving program, we simply ask for one specific run of a
given program. Executing programs “by hand” is an important skill to have, but
this task is not a substitute for a more open-ended programming task.

Our main contribution is to show that there is in fact a rich spectrum of
different I/O tasks between these two extremes, as we will explore in section 4.
Moving along this spectrum yields a diverse collection of tasks requiring different skills to solve. We design tasks ranging from program reading and comprehension over completion of partial programs all the way to writing original programs.

These designs are expressed in a newly developed Haskell embedded domain specific language (EDSL). This EDSL has two purposes. On the one hand it provides components to describe task designs and generate concrete task instances thereof. On the other hand, and equally important, it encourages descriptions that separate orthogonal aspects of task designs. This separation makes maintaining, expanding and reusing task designs much easier for the educator. Especially, when working with task designs created by another author, as it can also act as a form of documentation.

We build on top of the existing implementation of the specification language from our previous work. Starting from a specification, hand-written or generated randomly, we derive example runs and programs satisfying the specification. These artifacts are then used to build tasks. For example, we can give examples runs and ask for a corresponding program. Or we give a program and ask for runs of that program. The testing capabilities of the specification language allow us to automatically check solution candidates for such generated tasks for correctness.

We will not go into the technical details of how we create these artifacts in the implementation. Instead this presentation focuses on the framework’s versatility in expressing interesting task ideas and generating variations.

We will first give a short overview of the previously introduced specification language. Next we will define the EDSL for describing tasks. Using both the specification language and the language of tasks together, we show how to design a diverse range of exercise tasks on Haskell-I/O.

2 Specifying and testing I/O behavior

Our previous work [15] introduced a specification language for I/O behavior. The goal of the language is to enable easy QuickCheck testing for I/O behavior. Specifications expressed in the language describe program behavior in terms of traces, i.e., sequences of read and written values, a program should produce. Programs are tested against specifications by repeatedly checking for different (carefully randomized) inputs whether the trace of that program matches the specification. We will not present a formal introduction of specifications as we did in our previous work [15]. A high-level overview of the language’s features is enough for this work.

The language has typical elements of a standard imperative language, but varies in some aspects, most notably the use of variables. It defines primitives for reading and writing values from and to the console, a branching construct to choose sub-specifications based on Boolean conditions, and an iteration construct. Iteration is done through loops, but with explicit exit markers instead of a termination condition. Here is the summation behavior from the previous section expressed as a specification:

\[
[\triangleright n \in N \mid \triangleright x \in Z \triangleright \text{len}(x, A) = n \land E \rightarrow R [\text{sum}(x, A) \triangleright]\]

Reading and writing, i.e., the primitive I/O action we want to observe, are written as \([\triangleright x]\theta\) and \([t\triangleright]\), with \(x\) specifying the variable into which to read the value and \(t\) being a term describing the value to be written. Reading actions also have an annotation \(\tau\) specifying the set of legal inputs we expect at that point. For example, we expect the first value read in the example specification to be a natural number. Branching on a Boolean condition \(c\) is written as \(s_F \triangleright \triangleright \triangleright c \triangleright \triangleright \triangleright s_T\), choosing the right branch \(s_T\) if \(c\) holds. Sub-specifications to be repeated are marked by \(\rightarrow^k\), and such a loop terminates on reaching an exit marker \(E\).

Variables in the specification language behave differently compared to classical imperative languages. They accumulate all values read into them. Variables are then either used as single values, i.e., the last value read into them, or as a list of all past values of that variable. The subscript of a variable indicates how it is used. \(C\) stands for the current value and \(A\) for all values. By design, specifications only define how inputs and outputs are interleaved and what the output values should be. They cannot describe internal states of a program.

The implementation, accompanying our previous work, exposes, besides constructors for \texttt{Specifications}, a simple API for testing programs against specifications. Testing relies on programs being expressed in a variant of the standard Haskell IO monad in which we can observe I/O effects [15,11]. This allows us to take a program, i.e., a value of the inspectable \(IO_{rep}\) type, and run it on an input sequence.

\[
runProgram :: String \rightarrow IO_{rep} () \rightarrow Trace
\]

We can then check whether the program run, encoded by its \texttt{Trace} of I/O actions, satisfies the behavior described by some \texttt{Specification}.

\[
accept :: Specification \rightarrow Trace \rightarrow Bool
\]

Repeating this process for randomly generated input sequences, we formulate a QuickCheck \texttt{Property} stating that a program satisfies a specification.

\[
fulfills :: IO_{rep} () \rightarrow Specification \rightarrow Property
\]

We also provide an interpreter to turn specifications into executable programs.

\[
buildComputation :: Specification \rightarrow IO_{rep} ()
\]

See [10] for details on this.

## 3 Describing parameterized tasks

This section introduces a small embedded domain specific language in Haskell to describe exercise tasks, including automatically checkable requirements. The language enables clear and concise descriptions of parameterized tasks. Descriptions can be built from orthogonal components allowing for quick and easy reuse.
and modification. We will use this language in section 4 to discuss different categories of exercise tasks on Haskell I/O. The EDSL itself, however, can be used to describe parameterized tasks on any topic.

The language consists of three separate components. Descriptions of concrete exercise tasks, called task instances, a (sub-)language for describing requirements of correct solutions, and ways to express general task designs, i.e., generators for concrete tasks. Generally speaking, task designs bundle up generators for parameters together with a recipe for turning parameters into task instances.

The design goals of the EDSL are as follows:

– Clearly and concisely communicate the task’s idea through its description, without exposing computational details or requiring knowledge thereof.
– Separate the basic building blocks of tasks into orthogonal and reusable components.
– The main purpose of a task’s description is to be read by educators. Automatically checking whether a solution candidate fulfills a task’s requirements is only a secondary feature.

First off we need a data type for concrete task instances.

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

The type parameter `s` represents the type of solution the `TaskInstance` expects. For simplicity we treat `Description` as an abstract string-like type for which we assume standard layout combinators exist [2]. Each `TaskInstance` can have a default `given` value of type `s`. By convention we treat this value as a somehow incomplete version of a correct solution to be used as a starting point for solving the task.

The `Require` type encodes the conditions under which a solution candidate is deemed correct. Requirements are not constructed directly, instead the EDSL provides constructor functions for different requirements. The simplest requirements are predicates on the solution type `s`.

```haskell
requirePure :: (s -> Bool) -> Require s
```

For more complex requirements we use QuickCheck’s `Property` type to enable randomized testing. QuickCheck also provides feedback in case the `Property` fails.

```haskell
requireProp :: (s -> Property) -> Require s
```

We can also add an arbitrary IO pre-processing step to a requirement.

```haskell
after :: Require s' -> (s -> IO (Maybe s')) -> Require s
```

`Maybe` here indicates that pre-processing might fail, in which case the requirement is not fulfilled. One usage of this combinator, we will see later, is to compile programs given as textual input to actual Haskell values usable in a `Property`. 
We define a primitive for building the conjunction of two requirements.

\[(/\) :: Require \(s \rightarrow\) Require \(s \rightarrow\) Require \(s\)]

Lastly we might require a correct solution to “match” the given value of the TaskInstance. For example, filling in gaps in a given skeleton. We define a class to specify what matching a skeleton means for a specific type.

```haskell
class Matches \(s\) where
  matches :: \(s \rightarrow s \rightarrow\) Bool
```

Conceptually an instance of Matches defines a partial order on \(s\) where \(matches t s\) evaluates to True iff \(s\) is an extended version of the partial solution \(t\).

```haskell
mustMatch :: Matches \(s\) \(\Rightarrow\) \(s \rightarrow\) Require \(s\)
mustMatch = requirePure \(\circ\) matches
```

Checking whether a requirement holds for some value will in general require IO. Either to run QuickCheck or because we used after.

```haskell
check :: Require \(s \rightarrow\) s \rightarrow IO Bool
```

Being able to represent concrete tasks, we can now define parameterized tasks as regular Haskell functions from parameters to TaskInstance values. For example, we can define a simple (non-I/O) task requiring adding up two numbers:

```haskell
taskAdd :: Int \(\rightarrow\) Int \(\rightarrow\) TaskInstance Int
taskAdd \(x\) \(y\) = TaskInstance
  \{ question = text ("Give the sum of " \(\#\) show \(x\) \(\#\) " and " \(\#\) show \(y\))
  , given = Nothing
  , requires = exactAnswer \((x + y)\)
  \}
exactAnswer :: (Eq \(a\), Show \(a\)) \(\Rightarrow\) \(a \rightarrow\) Require \(a\)
exactAnswer \(x\) = requireProp \$ \(\lambda\) \(s\) \(\rightarrow\) \(s =\equiv= x\)
```

Defining exactAnswer in terms of QuickCheck’s (\(\equiv\)) operator, we get informative feedback from QuickCheck’s output in case of a test failure. Giving the wrong solution to an instance of the above task might, for example, result in the following error:

```haskell
  > check (requires (taskAdd 2 3)) 4
  *** Failed! Falsified (after 1 test):
  4 /= 5
```

The last step to automatic task generation is to couple a parameterized TaskInstance with a generator of its expected parameter.

```haskell
data TaskDesign \(s\) = \(\forall\)\(p\),TaskDesign
  \{ parameter :: Gen \(p\)
  , instantiate :: \(p \rightarrow\) TaskInstance \(s\) \}
```
To instantiate a design we generate a parameter value and pass it to `instantiate`:

```
generateTaskInstance :: TaskDesign s → IO (TaskInstance s)
generateTaskInstance (TaskDesign param inst) =
  generate (inst <$> param)
```

We define combinators to aid in our goal of clearly communicating both a task’s idea and requirements. Instead of using the `TaskDesign` constructor itself we use

```
for :: Gen p → (p → TaskInstance s) → TaskDesign s
for = TaskDesign
```

resulting in the general pattern

```
someTask :: TaskDesign s
someTask = for someRandomParameter doSomething
  where someRandomParameter :: Gen p
doSomething :: p → TaskInstance s
```

The arguments to `for` are deliberately named to make the expression read like a high-level description of the task.

Generators can be combined and modified by specialized instantiations of well known combinators on monads and arrows [3]. The new names of these combinators reflect their domain specific usage and thereby aid in hiding computational details from task descriptions. The first of these combinators is

```
fixed :: p → Gen p
fixed = pure
```

allowing us to write `for (fixed parameter) doSomething` when we do not want to generate randomized tasks but still use the EDSL to communicate our design. Next we define

```
from :: (a → Gen b) → Gen a → Gen b
from = (<<)
```

so that we can describe parameter generators in terms of existing generators:

```
for (randomParameterB `from` randomParameterA) doSomething
```

Splitting up parameter generators separates different layers of randomness. Each layer can then easily be changed independently. For example, we can change `randomParameterA` to a fixed example

```
for (randomParameterB `from` fixed a) doSomething
```

while `randomParameterB` is untouched. The parameter used to create a task instance is still randomized, but with one layer of randomness less.
The `instantiate` function can only ever take a single argument. For tasks with multiple randomized parameters we define combinators for tuple generators.

\[
(\&\&\&\&): \text{Monad } m \Rightarrow (a \to m b) \to (a \to m b') \to a \to m (b, b') \\
(****): \text{Monad } m \Rightarrow (a \to m b) \to (a' \to m b') \to (a, a') \to m (b, b')
\]

Looking ahead to section 4, here is an example of how these combinators can be used to describe a `TaskDesign`.

\[
\text{for}
\]

\[
((\text{exampleTrace } \&\&\&\& \text{haskellProgram}) \text{`from` randomSpecification})
\]

giveInteractionTrace

The names of the individual components and the usage of the combinators clearly communicate the basic idea of this `TaskDesign`. The expression reads almost like actual instructions for a task. This makes it easy for someone familiar with the EDSL, like a teaching assistant, to quickly modify and reuse parts of the design.

4 Building Tasks on Haskell I/O

With a general mechanism for describing tasks in place, we will now build some actual tasks on Haskell I/O programming. The source code of the implementation and all examples from this paper, can be found at [https://github.com/fmidue/IOTasks](https://github.com/fmidue/IOTasks).

First we introduce an alias for the type of inspectable I/O computations (see section 2) to clearly separate it from syntactic program text.

\[
\text{type ExecutableHaskell} = \text{IO}_{rep} ()
\]

Our example tasks do not expect `ExecutableHaskell` as solution candidates but instead use syntactic `HaskellCode`. For brevity we keep `HaskellCode` abstract here. It is enough to know that we can inspect and print out values of this type.

Using `fulfills` from section 2 we can construct our first I/O specific requirement for a correct task solution.

\[
\text{mustSatisfy} :: \text{Specification} \to \text{Require ExecutableHaskell}
\]

\[
\text{mustSatisfy } s = \text{requireProp } (\text{`fulfills' } s)
\]

In order to be able to also check requirements like `mustSatisfy` we provide a function

\[
\text{compile} :: \text{HaskellCode} \to \text{IO} (\text{Maybe ExecutableHaskell})
\]

to obtain semantic programs from syntactic representations. Together with `after` we can now build semantic requirements for syntactic programs. For example,

\footnote{The repository also contains instructions explaining how to generate and inspect random task instances, for the given examples.}
passesCompiler :: Require HaskellCode
passesCompiler = requirePure (const True) `after` compile

requires program text to be valid Haskell. We will define additional requirements as we discuss the various example tasks

As hinted at earlier, we cannot rely only on verbal descriptions to convey a task’s requirements. Instead we will use fixed verbal instructions in combination with program code and/or interaction traces. Our framework provides generators to build programs satisfying a given specification.

haskellProgram :: Specification \rightarrow Gen HaskellCode
pythonProgram :: Specification \rightarrow Gen PythonCode

We use randomized generators to create program code from specifications as there usually are different ways to implement the given behavior. Having access to different programs for the same specification is also useful for certain types of task designs, as we will see later on. For program code we mainly use Haskell code in our tasks, but it is also useful to have access to code in other languages and paradigms. For example, we use Python code to highlight how I/O looks different in Haskell compared to an imperative language. Our students should already know Python as it is taught in their introductory programming course.

We will not discuss the full details of this code generation. Our focus is on showcasing the different types of tasks expressible by the framework.

Essentially the code generation translates a given specification into an abstract program representation, agnostic about implementation techniques. These programs can be translated to a concrete language’s syntax, like Haskell or Python, by choosing appropriate embeddings of iteration, branching, state passing, etc. Rewriting the intermediate representation, using predefined rules, we obtain slightly different programs for the same specification. This approach also allows us to generate programs with (randomized) gaps or programs containing certain syntactic errors and anti-patterns.

The framework also provides generators for example traces matching a specification.

elementTrace :: Specification \rightarrow Gen Trace
elementTraces :: Int \rightarrow Specification \rightarrow Gen [Trace]

Since all of the artifact generators require specifications as parameters, suitable generators are assumed to exist. These generators are meant to be implemented by an educator/user as it is difficult to provide good generic generators. A sketch of how to write generators for specifications can be found in \[16\]. For our purposes we assume to have two generators:

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

---

\[16\] All code shown in this section is part of the framework’s API, except for expressions of type \( p \rightarrow \text{TaskInstance} \ s \) and \( \text{TaskDesign} \ s \). Values of these two types should be read as defined by the framework’s user.
The first generates a sensible random specification, for the educator's (task-specific) definition of sensible. The second generates a pair of similar looking specifications with differing behavior. Differing specification, for example, might have slightly different loop-termination conditions or varying outputs.

With all of these tools we now describe a diverse range of task designs. Following Le and Pinkwart [6] we classify these tasks into three classes:

1) Tasks with a single correct answer
2) Tasks with multiple correct answers but only a single solution strategy
3) Tasks with multiple different solution strategies

In our case these classes correspond to the complexity of requirement descriptions and roughly to task difficulty. Our introductory examples for programming tasks from section 1 sit at the two extreme points of this class spectrum. The task with only a verbal description and no restrictions on the programming techniques to use is a perfect example of a class 3 task. Adding additional requirements to such a task moves it further towards or into class 2. On the other side of the spectrum, asking for a given program’s behavior on a specific input is a class 1 task. The rest of this section will explore different points on this spectrum. Starting with class 1 tasks, we try to get as close to verbal-only free form tasks as possible. Pedagogically, this spectrum can also be seen as a progression of consecutive exercise tasks, developing students abilities to read, reason about, expand and finally write programs [12].

4.1 Tasks with one correct answer

Tasks with only a single correct answer cannot require a student to do any “real” programming. Even for small programs there is almost never only one right answer. Class 1 tasks are usually quiz-like tasks that focus on program reading and comprehension or simple program completion.

The simplest option to build such a task from our I/O related primitives is giving students two (or more) artifacts and asking them whether these artifacts originated from the same specification. For example, given two programs, determine whether they have the same behavior.

```haskell
data BinDesc = Yes | No deriving (Eq, Ord, Enum, Show)

decision :: TaskDesign BinDesc
decision = for (equalityProblem 'from' similarSpecifications) checkAgreement

checkAgreement :: (BinDesc, HaskellCode, HaskellCode) -> TaskInstance BinDesc
checkAgreement (haveSameBehavior, p1, p2) = TaskInstance
  { question = text "Do these two programs have the same behavior?"
    $$ text (show p_1) $$ text "---" $$ text (show p_2)
    , given = Nothing
    , requires = exactAnswer haveSameBehavior }
```
For simplicity we assume that there are at least two different programs for each specification.

\[
\text{equalityProblem} :: (\text{Specification, Specification}) \rightarrow \text{Gen (BinDesc, HaskellCode, HaskellCode)}
\]

\[
\text{equalityProblem} (s_1, s_2) = \text{do}
\text{sameBehavior} ← \text{elements [No, Yes]}
(p_1, p_2) ← \text{if sameBehavior} == \text{Yes}
\text{then differentPrograms} s_1 s_1
\text{else differentPrograms} s_1 s_2
\text{pure (sameBehavior, p_1, p_2)}
\]

\[
\text{differentPrograms} :: \text{Specification → Specification} \rightarrow \text{Gen (HaskellCode, HaskellCode)}
\]

\[
\text{differentPrograms} s_1 s_2 = \text{do}
p_1 ← \text{haskellProgram} s_1
p_2 ← \text{haskellProgram} s_2 'suchThat' (\neq p_1)
\text{return} (p_1, p_2)
\]

To illustrate what instances of the \text{decision-task} from above can look like here is an example of two similar looking programs with slightly different behavior. The second program is obtained by modifying the specification underlying the first program, in this case the loop-termination condition was randomly changed and the no longer needed initial input deleted.

\[
p_1 = \text{do}
n ← \text{readLn}
\text{let loop } xs =
\text{if length } xs == n
\text{then do return } xs
\text{else do}
v ← \text{readLn}
\text{loop } (xs ++ [v])
ys ← \text{loop } []
\text{print } (\text{sum } ys)
\]

\[
p_2 = \text{do}
\text{let loop } xs \text{ acc } =
\text{if acc } == 5
\text{then do print } xs
\text{else do}
v ← \text{readLn}
\text{loop } (xs + v) (\text{acc + 1})
\text{loop } 0 0
\]

Tasks on program completion use the possibility to generate partial programs we hinted at in section \[\text{3}\] We use a generator

\[
\text{haskellWithGaps} :: \text{Specification → Gen HaskellCode}
\]

that produces an I/O program with gaps. These gaps need to be filled with either \text{readLn} or \text{print}. The different types of these two functions ensure there is only one correct solution. For such a program we require choosing an appropriate expression for each gap.

\[
\text{completion1} :: \text{TaskDesign HaskellCode}
\text{completion1} = \text{for (haskellWithGaps 'from' randomSpecification) fillGaps}
\]
fillGaps :: HaskellCode → TaskInstance HaskellCode
fillGaps skeleton = TaskInstance
  { question = text "Complete the following program."
    $\$ text "(Replace ? with readLn or print)"
    , given = Just skeleton
    , requires = passesCompiler /\ mustMatch skeleton }

For the last type of answer, traces, we can give a Haskell program and some
input sequence and ask students to execute the program on that input. Fixing
the input sequence, there is only one correct solution to such a task.

comprehension₁ :: TaskDesign Trace
comprehension₁ = for
  ((exampleTrace & & & haskellProgram) 'from' randomSpecification)
giveInteractionTrace
giveInteractionTrace :: (Trace, HaskellCode) → TaskInstance Trace
giveInteractionTrace (t, prog) = TaskInstance
  { question = text ("Give the program’s trace for input sequence")
    <> text (show (inputs t))
    $\$ text (show prog)
    , given = Nothing
    , requires = exactAnswer t }

inputs :: Trace → [String]

This design can, for example, generate the task from page 2.

4.2 Tasks with multiple correct answers

Before moving on to tasks on actual programming, we first look at a class 2
variant of the last task from the previous section. Instead of giving a fixed input
sequence we ask for an interaction trace with a certain property. One possibility
is to give two similar looking programs with different semantics and ask for an
input sequence for which the given programs exhibit different I/O behavior.

comprehension₂ :: TaskDesign [String]
comprehension₂ = for
  ((specificationAnd haskellProgram **** specificationAnd haskellProgram)
    'from' similarSpecifications)
findDiffSequence

findDiffSequence :: (Specification, HaskellCode), (Specification, HaskellCode)
→ TaskInstance [String]

findDiffSequence ((s₁, p₁), (s₂, p₂)) = TaskInstance
  { question = text "Find inputs resulting in different behavior."
    $\$ text (show p₁) $\$ text "---" $\$ text (show p₂)
    , given = Nothing
    , requires = triggerDifference s₁ s₂ }
triggerDifference :: Specification \rightarrow Specification \rightarrow Require [String]

\( \text{triggerDifference } s_1 \ s_2 = \text{requireProp } \lambda is \rightarrow \)
\((=/=) \ 'on' \ (\text{runProgram } is \circ \text{buildComputation}) \ s_1 \ s_2 \)

specificationAnd :: (Specification \rightarrow Gen a) \rightarrow Specification
\rightarrow Gen (Specification, a)

specificationAnd \ g = \text{pure } \& \& \& \ g

The requirement uses the \text{buildComputation} function shown in section 2 to derive executable programs from the specifications. Executing the specifications this way is easier than executing the displayed programs since these only exist in a textual form.

So far, our tasks are straightforward with regard to the questions asked. For programming tasks beyond gap filling we now need to describe the required behavior of programs as well as restrict which solution strategies are valid. To start off, we give interaction traces, i.e., example runs, to specify behavior and fix a solution strategy by providing a skeleton to complete.

\[
\text{completion_2} :: \text{TaskDesign} \ \text{HaskellCode} \\
\text{completion_2} = \text{for} \\
(\text{exampleTraces} \ 5 \ 'from' \ \text{fixed specification}) \\
\text{matchExamples} \\
\text{where specification} = \ldots \\
\text{matchExamples} :: [\text{Trace}] \rightarrow \text{TaskInstance} \ \text{HaskellCode} \\
\text{matchExamples} ts = \text{TaskInstance} \\
\{ \text{question} = \text{text} "Complete the program to match the examples:'" \\
\&\& \ \text{vcat (map (text} \circ \text{show) ts)} \\
, \text{given} = \text{Just skeleton} \\
, \text{requires} = \text{produceTraces ts 'after' compile /\ mustMatch skeleton} \} \\
\text{where skeleton} = \text{fromSourceString } \\text{unlines} \\
["main = do" \\
, " ?" \\
, "$ while ? ? ?" \\
, "$while :: (a \rightarrow \text{Bool}) \rightarrow (a \rightarrow \text{IO \ a}) \rightarrow a \rightarrow \text{IO \ a}" \\
, "$while = \ldots"] \\
\text{produceTraces} :: [\text{Trace}] \rightarrow \text{Require} \ \text{ExecutableHaskell} \\
\text{produceTraces} ts = \text{requirePure } \lambda p \rightarrow \\
\text{all } (\lambda t \rightarrow \text{runProgram } (\text{inputs} t) \ p == t) \ ts \\
\text{fromSourceString} :: \text{String} \rightarrow \text{HaskellCode}
\]

This task asks to produce the given examples by mimicking an imperative loop using the higher-order \text{while} function. For simplicity we use a fixed specification instead of a random one. Using a random specification, without any further restrictions, potentially results in an unsolvable task since the skeleton can have the wrong structure for the underlying behavior. With a suitable generator, however, the underlying specification can be randomized as well.
Using traces to describe behavior requirements has a disadvantage one needs to be aware of. Such descriptions may not fully characterize the underlying specification’s behavior. The description essentially gives a list of unit tests for a solution to fulfill. The underlying specification guarantees there is at least one program solving the task without hard-coding the given examples. However, a program working only for the given examples and crashing on all others inputs still meets the requirement. In the above task such solutions are largely ruled out by the skeleton. For free-form programming tasks that do not use a skeleton, this is a more serious problem. Our solution is to use program code itself to describe the required behavior. Doing so naturally gives rise to two exercise types: refactoring and cross-language re-implementation.

Refactoring tasks ask to rewrite a given program into a program that satisfies certain properties the original program does not have. Both programs should behave identically, with regards to I/O, for the same inputs. The original program therefore fully describes the behavior a correct solution should have.

The next example gives a program accumulating list values and outputting the result of a computation, expressible as a fold, on this list. Students are asked to rewrite this program into a version not using any list, directly carrying out the computation.

```haskell
refactoring :: TaskDesign HaskellCode
refactoring = for
(specificationAnd haskellFoldProgram `from` fixed specification)
rewriteToNoLists
where specification = undefined
rewriteToNoLists :: (Specification, Description) → TaskInstance HaskellCode
rewriteToNoLists (spec, prog) = TaskInstance
  { question =
    text "Re-write the program such that it does not use lists."
    $$ prog,
    , given = Nothing
    , requires = (mustSatisfy spec `after` compile) \ noLists
  }
noLists :: Require HaskellCode
noLists = requirePure $ λp →
  not (containsFunction "++" p ∨ containsFunction ":" p)
containsFunction :: String → HaskellCode → Bool
haskellFoldProgram :: Specification → Gen Description
We check the requirement of not using lists by simply verifying that solution code does not contain the functions to build lists. Once again, the exact details on how haskellFoldProgram is implemented internally are outside the scope of this presentation. We basically take a program and look for a function known to be a fold, e.g. `sum` or `length`, that is used on the result of a list accumulating loop. This function’s base case and recursive step are then “inlined” into the loop.
4.3 Tasks with different solution strategies

For our last example we finally arrive at a task very similar to the verbal-only
free-form tasks from the beginning. The task requires re-implementing a Python
program in Haskell. As already stated above, we choose Python since our stu-
dents learn it as their first programming language and are therefore already
familiar with it.

\[
\text{pythonToHaskell} :: \text{TaskDesign HaskellCode} = \text{for (specificationAnd pythonProgram `from` \text{randomSpecification}) rewriteAsHaskell}
\]

\[
\text{rewriteAsHaskell} :: (\text{Specification, PythonCode}) \rightarrow \text{TaskInstance HaskellCode} = \text{TaskInstance}
\]

\[
\begin{align*}
\text{question} &= \text{text "Write the following program in Haskell:"} \\
\text{given} &= \text{Nothing} \\
\text{requires} &= \text{mustSatisfy s `after` compile}
\end{align*}
\]

This task does not fix a solution strategy. By starting from a Python program it
precisely states the required behavior. The Python program also does not contain
any information on how a Haskell program with the identical behavior could
look like. I/O programs in Haskell usually have a different structure compared
to imperative languages (e.g. recursive functions vs. explicit loops). Take, for
example, the following task instance generated from the above des-

Re-implement the following Python program in Haskell:

\[
\begin{align*}
n &= \text{int(input())} \\
x &= [] \\
\text{while len}(x) \neq n : \\
v &= \text{int}(\text{input()}) \\
x &= x + [v] \\
\text{print} \left( \text{sum}(x) \right)
\end{align*}
\]

The required behavior is the same as the verbal description from the intro-
duction. It is clear that we need to use some form of repetition in our solution,
but that information is also contained in the verbal description ("read n integers
one after the other"). Only the usage of a list to store read integers is not in the
verbal description. For everyone with basic programming skills this should not
be anything new. We therefore argue that the above Python program can replace
the verbal description without loosing precision or providing additional hints.

5 Related Work

Tools for automatic task generation exist in a variety of different application
areas, for example, general science questions [14], math related tasks [5] and
programming tasks [7,10]. Some systems for natural language questions can gen-
erate tasks from databases of domain specific text [14], but most approaches use
templates together with parameter generators, similar to our TaskDesigns. In contrast to our flexible EDSL approach most of these systems use rigid template formats provided as inputs to the task generation. Consequently, these systems are usually embedded inside a specific e-learning environment. Our framework can, in principle, be used with any e-learning system. We already use a modular e-learning system [8,13] and plan to integrate the framework in that context.

Our generated task instances do not provide any detailed feedback apart from maybe some QuickCheck outputs. There are other automatic assessment systems providing more detailed feedback, including suggestions on how to fix mistakes. A survey of different automatic assessment systems for programming tasks and the feedback they can generate is presented by Keuning et al. [4].

6 Conclusion & Future work

The presented framework can be used to describe a diverse range of exercise task designs and generate concrete randomized instances from these designs. Separating task descriptions into orthogonal components makes modifying and reusing tasks easy. Task designs also serve as high-level documentation for the task’s idea if we choose descriptive names for the individual components.

We presented examples of using the framework to describe tasks on Haskell I/O. The variety in these tasks stems from domain specific primitives providing different artifacts around which the parameterized tasks are built. These artifacts are used as stand-ins for verbal descriptions to precisely state task requirements even if tasks are built from randomized specifications. Tasks created this way have a slightly different feel compared to traditional hand-written ones. Even though there are some restrictions to our approach, we can still create a wide range of different task types. To the best of our knowledge automatically deriving artifacts for communicating requirements is a novel approach to automatic task generation in the context of programming tasks.

We have not yet had the opportunity to use tasks like the ones shown in section 4 in practice. However, our tasks on Haskell I/O-programming already use the specification language and its testing facilities. We plan to test the presented approach to task design in the next iteration of our programming paradigms course. We are especially interested to see whether tasks on program completion and comprehension benefit students when learning Haskell-I/O. Writing high-quality generators for specifications should also be investigated further. Good generators have a big influence on the quality of concrete task instances.

References

1. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: International Conference on Functional Programming, Proceedings, pp. 268–279. ACM (2000). [https://doi.org/10.1145/351240.351266]
2. Hughes, J.: The design of a pretty-printing library. In: Advanced Functional Programming. AFP 1995. LNCS, vol. 925, pp. 53–96. Springer (1995). [https://doi.org/10.1007/3-540-59451-5_3]
3. Hughes, J.: Generalising monads to arrows. Science of Computer Programming 37(1-3), 67–111 (2000). https://doi.org/10.1016/S0167-6423(99)00023-4
4. Keuning, H., Jeuring, J., Heeren, B.: A systematic literature review of automated feedback generation for programming exercises. ACM Transactions on Computing Education 19(1), 3:1–3:43 (2019). https://doi.org/10.1145/3231711
5. Kurt-Karaoglu, F., Schwinning, N., Striewe, M., Zurmaar, B., Goedicke, M.: A framework for generic exercises with mathematical content. In: International Conference on Learning and Teaching in Computing and Engineering. LaTICE 2015. pp. 70–75. IEEE (2015). https://doi.org/10.1109/LaTICE.2015.11
6. Le, N.T., Pinkwart, N.: Towards a classification for programming exercises. In: Proceedings of the 2nd Workshop on AI-supported Education for Computer Science. AIEDCS 2014. (2014)
7. Mosbeck, M., Hauer, D., Jantsch, A.: VELS: VHDL e-learning system for automatic generation and evaluation of per-student randomized assignments. In: IEEE Nordic Circuits and Systems Conference. NORCAS 2018: NORCHIP and International Symposium of System-on-Chip (SoC). pp. 1–7. IEEE (2018). https://doi.org/10.1109/NORCHIP.2018.8573455
8. Rahn, M., Richter, A., Waldmann, J.: The Leipzig Auto-tool e-learning/e-testing system. In: Symposium on Math Tutoring, Tools and Feedback. Open Universiteit Nederland. (2008). http://www.imm.htwk-leipzig.de/~waldmann/talk/08/ou08/tool.pdf accessed July 2020
9. Siegburg, M., Voigtländer, J., Westphal, O.: Automatische Bewertung von Haskell-Programmieraufgaben. In: Proceedings of the Fourth Workshop “Automatische Bewertung von Programmieraufgaben”. pp. 19–26. GI (2019). https://doi.org/10.18420/abp2019-3
10. Striewe, M., Balz, M., Goedicke, M.: A flexible and modular software architecture for computer aided assessments and automated marking. In: CSEDU 2009 - Proceedings of the First International Conference on Computer Supported Education, Lisbon, Portugal, March 23-26, 2009 - Volume 2. pp. 54–61. INSTICC Press (2009)
11. Swierstra, W., Altenkirch, T.: Beauty in the beast. A Functional Semantics for the Awkward Squad. In: Haskell Workshop, Proceedings. pp. 25–36. ACM (2007). https://doi.org/10.1145/1291201.1291206
12. Van Merriënoor, J.J.G., De Croock, M.B.M.: Strategies for computer-based programming instruction: Program completion vs. program generation. Journal of Educational Computing Research 8(3), 365–394 (1992). https://doi.org/10.2190/MJDX-9PP4-KFMT-09PM
13. Waldmann, J.: Automatische Erzeugung und Bewertung von Aufgaben zu Algorithmen und Datenstrukturen. In: Proc. ABP, CEUR WS vol. 2015 (2017)
14. Welbl, J., Liu, N.F., Gardner, M.: Crowdsourcing multiple choice science questions. In: Proceedings of the 3rd Workshop on Noisy User-generated Text, NUT@EMNLP 2017. pp. 94–106. Association for Computational Linguistics (2017). https://doi.org/10.18653/v1/w17-4413
15. Westphal, O., Voigtländer, J.: Describing Console I/O Behavior for Testing Student Submissions in Haskell. In: Eighth and Ninth International Workshop on Trends in Functional Programming in Education. Proceedings. EPTCS, vol. 321, pp. 19–36. EPTCS (2020). https://doi.org/10.4204/EPTCS.321.2
16. Westphal, O., Voigtländer, J.: Implementing, and Keeping in Check, a DSL Used in E-Learning. In: Proceedings of the 15th International Symposium on Functional and Logic Programming. LNCS, vol. 12073. Springer (2020). https://doi.org/10.1007/978-3-030-59025-3_11