

Describing Console I/O Behavior for Testing Student Submissions in Haskell

Oliver Westphal

Janis Voigtländer

Universität Duisburg-Essen
Germany

`oliver.westphal@uni-due.de`

`janis.voigtlaender@uni-due.de`

We present a small, formal language for specifying the behavior of simple console I/O programs. The design is driven by the concrete application case of testing interactive Haskell programs written by students. Specifications are structurally similar to lexical analysis regular expressions, but are augmented with features like global variables that track state and history of program runs, enabling expression of an interesting range of dynamic behavior. We give a semantics for our specification language based on acceptance of execution traces. From this semantics we derive a definition of the set of all traces valid for a given specification. Sampling that set enables us to mechanically check program behavior against specifications in a probabilistic fashion. Beyond testing, other possible uses of the specification language in an education context include related activities like providing more helpful feedback, generating sample solutions, and even generating random exercise tasks.

1 Introduction

In our course on programming paradigms we teach the main concepts of Haskell. For students to gain practical experience with the language, we give weekly exercise tasks and let them submit solutions for review and grading. Since checking submissions by hand is tedious and potentially error-prone, we employ an e-learning system [7, 9], automatically testing submissions against sets of QuickCheck properties [1] wherever possible. An added benefit for students is that they get immediate feedback and can revise their submissions accordingly and incrementally.

This approach works very well for exercise tasks about implementing pure functions. But applying it to tasks about I/O programming is significantly more complicated. While Swierstra and Altenkirch [8] showed how one can change the monad underlying a Haskell I/O program in order to get an inspectable representation of what the program is doing, thus in principle enabling the checking of properties formulated over executions via QuickCheck or similar tools, practical application is cumbersome. Specifically, for every I/O exercise task we want to handle that way, we currently need to implement the following components:

- A generator of input sequences, respecting the task’s invariants.
- A way of checking if a given execution trace exhibits the desired behavior for a given input sequence.
- A method of providing feedback in case the behavior did not match the expectations, e.g., explaining the mismatch or showing what would be correct behavior for the relevant input sequence.

Since we typically allow students some degree of freedom when it comes to how their program should prompt for input values, in what form exactly it should print any computed result values, whether there are additional/optional output messages, etc., these components have to cover a lot of different cases even for a single exercise task. Moreover, the components are not directly related to each other, much

less derived from a common source, thus leaving room for inconsistencies and other errors. We lack an overall framework, and it shows.

Our main idea here is that much of this complexity can be tamed by designing a domain-specific language for specifying interactive program behavior in a way that enables automatic generation of the components listed above. We therefore make the following contributions:

- We design (Section 3), then describe the syntax (Section 4) and semantics (Section 5), of a small language suitable for specifying simple programming tasks about console I/O. Key features are history aware variables that allow access to all values previously read into them and an encoding of optional or ambiguous output behavior.
- For any given specification, we show how to generate generalized traces that cover the complete behavior accepted by that specification under a fixed sequence of inputs. Based on this trace generation process we can build the three necessary components, mentioned above, for automatically testing exercise tasks (Section 6).
- We start to implement the presented approach via an embedded domain-specific language (EDSL) in Haskell (Section 7). More details and further development are reported in a companion paper [10].

For the sake of simplicity of the described formalism, we will only consider programs that read and write integers. It is however straightforward to generalize the approach to, for example, include string values, and the implemented EDSL actually does so. Along with the focus on integers in the formalization, we will assume that I/O programs to test are written using the two primitive operations *readLn* and *print* that incorporate conversion from and to strings¹.

2 Current practice

Consider the following verbal description of a function one might give as an exercise task to a beginning programmer:

“Add up all the numbers in a given list.”

A simple Haskell solution could look like this:

```
sum :: [Int] → Int
sum []      = 0
sum (x : xs) = x + sum xs
```

To test this solution, we could use QuickCheck properties like the following ones:

```
propSing :: Int → Bool           propAdd :: [Int] → [Int] → Bool
propSing = λx → sum [x] == x     propAdd = λxs ys → sum (xs ++ ys) == sum xs + sum ys
```

Now consider another task, which might appear in a course section introducing I/O programs:

“Read a natural number n from `stdin`, then read n additional numbers and print the sum of those n numbers to `stdout`.”

The following Haskell solution has basically the same computational content as the function further above. But the fact that the program has to fetch its inputs on its own, and to report the computed result value back to the user, changes the overall code structure considerably.

¹This choice of primitive operations is also not meant as a real restriction. In the actual implementation we also provide primitives that directly operate on strings.

```

main :: IO ()
main = do n ← readLn
        go n 0

go :: Int → Int → IO ()
go 0 res = print res
go n res = do x ← readLn
            go (n - 1) (x + res)

```

Now how do we test such a program? How, even, can we describe more formally than in the second verbal description above what behavior is desired?

First we need to consider what it is that we want to test. In the case of the simple *sum* function, we wanted to test the result value of the computation. In the I/O case, we are also interested in the interaction of the program with the outside world (in what order are which values read and printed, etc.). We therefore can no longer view such programs as just mappings from input values to output values. Instead, programs will result in a sequence of potentially interleaved input and output actions. We call such a sequence a trace of a program. If we want to check whether some program exhibits a certain desired behavior, we have to check the traces it can produce.

For the above I/O task description, the set of intended traces is basically

$$\{ ?0 !0 stop, ?1 ?v_1 !v_1 stop, ?2 ?v_1 ?v_2 !(v_1 + v_2) stop, \dots \}$$

where each *?* stands for an input action and each *!* for an output action. If we now assume that we never supply negative numbers as input values (at least not for the first input), then the Haskell I/O program given above indeed produces exactly all, and only, traces from this set.

Following the approach presented by Swierstra and Altenkirch [8], corresponding tests can be automated. First, an alternative monad is defined that represents a semantic domain for console I/O programs²:

```

data IO_rep a
  = GetLine (String → IO_rep a)
  | PutLine String (IO_rep a)
  | Return a

instance Monad IO_rep where
  GetLine f >>= g = GetLine (λs → f s >>= g)
  PutLine s ma >>= g = PutLine s (ma >>= g)
  Return a >>= g = g a
  return = Return

```

Next, the *IO* primitives to be used are implemented for this new representation:

```

readLn :: Read a ⇒ IO_rep a
readLn = fmap read (GetLine Return)

print :: Show a ⇒ a → IO_rep ()
print x = PutLine (show x) (Return ())

```

Now, any potential Haskell solution to the I/O task given further above, *main = do ...*, can not only be used at type *IO ()*, but also at type *IO_rep ()*. Since values of that type are more inspectable than those of a normal *IO* type, we can then “run” *main* in a kind of simulation mode that produces an explicit trace as a data structure when given some concrete inputs:

```

run_rep :: IO_rep () → [String] → Trace
run_rep (GetLine f) (x : xs) = Read x (run_rep (f x) xs)
run_rep (PutLine s ma) xs    = Write s (run_rep ma xs)
run_rep (Return ()) []      = Stop

data Trace
  = Read String Trace
  | Write String Trace
  | Stop

```

²Swierstra and Altenkirch use a representation based on input and output of single characters. We are currently not interested in such a fine-grained inspection and therefore always require programs to read or write whole lines.

If we now assume existence of a predicate $checkCorrectness :: Trace \rightarrow Bool$ and of a random generator $validInputs :: Gen [String]$, then we can use QuickCheck again to automatically test whether a program has the intended behavior. But writing such a generator and in particular the predicate is generally not as straightforward as one might hope. Additionally, requiring feedback in case the predicate returns *False* adds significant extra complexity.

Our specification language and surrounding tooling solve exactly these problems. We can state what behavior we want to see, and automatically get the relevant components required for testing.

3 Specifications

The main goal of the specification language is to describe the behavior that a correct solution for some task should have. We want such descriptions to be concise, intuitive and easily adaptable toward new tasks. The design currently does not include any abstraction facilities or try to achieve compositionality since at the moment we focus only on specifying small scale exercise tasks. For the same reason, we do not care to capture all possible console I/O programs, or rather their behavior. A lot of behavior is uninteresting or ill-suited for exercise tasks. For example, we want behavior that necessarily requires interaction, so that students actually have to make use of the I/O-primitives to solve the task. Section 3.3 will go into more detail regarding this aspect. To motivate the design of our small DSL, we will first go through the example task from above step by step and see what constructs are necessary to describe the intended behavior formally (Section 3.1). We also explain how a specification is to be interpreted intuitively in terms of execution traces (Section 3.2).

3.1 Describing behavior

Recall the second task description from the previous section:

“Read a natural number n from `stdin`, then read n additional numbers and print the sum of those n numbers to `stdout`.”

Since we want to speak about interactive behavior, we first need notations for input and output primitives. We use square brackets to describe such atomic actions and distinguish inputs from outputs via a triangle arrow into something or out of something. This gives us the following basic skeleton for our specification:

$$[\triangleright ?] ??? [? \triangleright]$$

We use (silent) concatenation to glue several shorter specifications together. The above skeleton already encodes that we first read something and at some later point should print something back.

Next, in order to relate inputs and outputs, we need variables to reference read values at later points and functions to express computations over the values referenced by those variables:

$$[\triangleright n] ??? [sum(?) \triangleright]$$

Right now it is not clear what the argument to *sum* should be, but we will fill it in shortly.

The middle part of our example specification should correspond to the reading-in of the n numbers we want to sum. Since n is determined by the first read value, we do not know up front (before a program runs) how many values we need to read overall. Therefore we need some mechanism for flexible iteration (rather than just some fixed times concatenation of sub-specifications). We mark the

part of a specification we would like to iterate with $\rightarrow^{\mathbf{E}}$ and introduce a marker \mathbf{E} to indicate where/when the iteration process should finish:

$$[\triangleright n](???\mathbf{E})\rightarrow^{\mathbf{E}}[sum(?)\triangleright]$$

Now the middle part is repeated until the exit marker is hit. However, up to now we have no way to skip over certain parts of a specification or to choose between alternatives based on some condition. In order for our iteration process to not always terminate after the first round, we need to introduce a branching construct:

$$[\triangleright n](?\angle?\searrow\mathbf{E})\rightarrow^{\mathbf{E}}[sum(?)\triangleright]$$

Now we can fill in a condition that only when satisfied gives control to the right branch, leading in our case to the termination of the iteration process. Otherwise the left branch will be used.

We can use this new construct to repeatedly read in a value:

$$[\triangleright n](([\triangleright x]\angle?\searrow\mathbf{E})\rightarrow^{\mathbf{E}}[sum(?)\triangleright])$$

But now we have a problem, or actually two. In each round the old value we “assigned” to x previously is lost, and we have no way of knowing when to stop. The key feature of our DSL that helps solve both issues is the fact that variables do not just store a current value like in most programming languages. Variables instead hold lists of all values assigned to them in chronological order. There are then two different ways to access a variable, either as the traditional current value, denoted via the subscript C (current), or as the list of all values read into that variable so far, denoted with the subscript A (all). This gives us the expressive power to not only construct the missing branching condition but now also fill in the missing argument to the summation:

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

One thing the verbal description states that is not yet present in the DSL expression is the fact that the first number should not be negative. This kind of restriction (in a task) is often useful when we do not care about ill-formed or otherwise undesirable inputs, especially in an educational setting where we usually introduce new concepts one step at a time. That is, in the beginning of a course we might not want students to, for example, have to worry about checking inputs for correctness. But later on we might explicitly require them to do so. Our specification language therefore provides the necessary flexibility to go both ways. Each occurrence of the primitive for reading has to be annotated with the set of values we expect there (and the way in which the specification will then be used determines whose job it is to take care of those expectations, the students’ and/or the tester’s³):

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

The specification we have arrived at now (and which is essentially, up to a minuscule syntactic difference, already a valid expression in our DSL) is quite rigid, as there is no flexibility with regard to the interaction allowed. Continuing our example, one might want to allow the programs to have some extra behavior that does not really influence the core functionality. For example, we could modify the previous task description as follows:

³In our current setting we only ever present well-formed inputs to the students’ programs.

“Read a natural number n from `stdin`, then read n additional numbers and print the sum of those n numbers to `stdout`. Additionally, when the program is still expecting at least one additional summand, it might print how many more summands it is expecting, before reading in the next input.”

We encode such optional behavior directly inside the output primitive. That is, instead of giving a single term to describe what we expect as output, we use a set of possible terms. This set might contain the “empty” term ϵ representing no output and thereby optionality:

$$[\triangleright n]^{\mathbb{N}} \left(([\{\epsilon, n_C - \text{len}(x_A)\} \triangleright] [\triangleright x]^{\mathbb{Z}}) \angle \text{len}(x_A) = n_C \triangleright \mathbf{E} \right)^{\rightarrow \mathbf{E}} [\{\text{sum}(x_A)\} \triangleright]$$

While this way of expressing optionality can look a bit cumbersome compared to, for example, simply flagging an output as optional via a dedicated construct, it is far more expressive since the set we can give there is rather arbitrary. For example, we could allow the programs, for whatever reason, to output exactly any multiple of the result of some value computation⁴.

At first glance it might seem overly complicated or restrictive that we only introduce this specific kind of non-determinism, in outputs, and not, for example, a general non-deterministic choice operator. But such a more general operator would allow us to write specifications that represent statements like “Fulfill either task A or task B”. If we now make such a specification part of an iteration expression, a different task can be chosen to be fulfilled in each round. Abstractly this is fine, but since programs (in the language in which students write their submissions) are not actually capable of true non-deterministic choice, we cannot have a program that really behaves that way. Therefore such a choice operator introduces not only the form of optionality we do want to encode; instead it enables also a form of meta statement that we think should not be part of the specification language itself.

Note that this does not mean that specifications cannot require completely different behavior depending on some input. For example, we can write specifications of the form $[\triangleright x]^{\mathbb{Z}} (s_1 \angle p(x_C) \triangleright s_2)$. But since $p(x_C)$ is deterministically defined once x_C is known, there is no non-determinism involved here. Combining this kind of deterministic branching with the possibility to have an empty specification, which we denote by $\mathbf{0}$, we can write specifications like $\mathbf{0} \angle p(x_C) \triangleright s$, which only requires s to be fulfilled if $p(x_C)$ evaluates to *True*.

3.2 Valid program runs

Consider now the following trace we might get from a program: `?2?5?3!8stop`. The program first reads in the numbers 2, 5 and 3, then prints 8 and stops. Does this trace match the specification developed above, i.e., could a program fulfilling the specification have such a run? If not, we have just found evidence that the program under consideration does not fulfill the specification. We can check the validity of the trace by going from left to right (and possibly in loops) through the specification and seeing if the trace actions match the required actions, while keeping track of the contents of variables.

Starting with `?2`, we compare it to $[\triangleright n]^{\mathbb{N}}$. Since both are input actions and moreover 2 is a natural number, as required, we continue by checking the remaining trace against the rest of the specification.

Next we have to check the iteration. To do this, we first check the trace against the iteration body while remembering the context in which the iteration occurred, i.e., the specification following it and the iteration body we might have to repeat. When we hit the end (but not exit marker) of the body, that is, we did not encounter an \mathbf{E} , we just check the remaining trace against the iteration body again. When

⁴This expressiveness really pays off when we move to outputting arbitrary strings, since we can then specify that we allow any output string as long as it contains the required result somewhere.

we do encounter an exit marker, we continue by checking the remaining trace against the specification following the whole iteration.

For our current case, the iteration's body is $([\{\varepsilon, n_C - \text{len}(x_A)\} \triangleright] [\triangleright x]^{\mathbb{Z}}) \angle \text{len}(x_A) = n_C \searrow \mathbf{E}$. So we have to check $?5 ?3 !8 \text{stop}$ against that. We first evaluate the branching condition to determine which sub-specification we have to match against. Since x_A has length 0 at the moment, we choose the left branch, which means we have to check $?5$ against $[\{\varepsilon, n_C - \text{len}(x_A)\} \triangleright]$. The trace action here is an input, but the specification calls for an output action. However, since ε is contained in the set of possible outputs, this is not problematic. After all, we can simply skip this output step, hoping we then find a match for the trace action. And indeed the next action required by the specification is $[\triangleright x]^{\mathbb{Z}}$, which matches $?5$ and results in 5 being assigned to x (actually, to be assigned to x_C and appended to x_A). Since we have no specification left to check locally, but are inside an iteration, we again check against the whole iteration body. This results in 3 also being read into x , which now conceptually holds the list $[5, 3]$ (as x_A , with x_C being the 3 from the end of that list). Therefore, in the next round the branching condition evaluates to *True*, thus ending the loop due to the occurrence of \mathbf{E} found to the right of the branching construct.

All that is left now is to check $!8 \text{stop}$ against $[\{\text{sum}(x_A)\} \triangleright]$. Since we have $\text{sum}(x_A) = \text{sum}([5, 3]) = 8$, this check is positive, also taking into account that *stop* matches the empty specification. Overall, we can conclude that the trace $?2 ?5 ?3 !8 \text{stop}$ is a valid program run for the specification.

An important feature of our specification language is that we also can essentially reverse this procedure, then looking for a (random) trace that will match the specification (instead of starting from a given trace). For the example specification above, this could yield $?3 !\{\varepsilon, 3\} ?-1 !\{\varepsilon, 2\} ?7 !\{\varepsilon, 1\} ?4 !\{10\} \text{stop}$ as one possible trace form, where the values of inputs are random elements of the expected types and the output values are the results of evaluating all output possibilities at the respective points. Such generalized traces can then be used to test programs by checking if a program's trace for the same input sequence is covered by the "specification trace". For example, if a program produces the trace $?3 !4 ?-1 !2 ?7 !1 ?4 !10 \text{stop}$, we see that this is not a valid trace since the first output action does not actually allow the printing of 4. If we do this for enough input sequences of random traces derived from the specification, we either find a counterexample or gain reasonable confidence in the correctness of a program (submission).

3.3 Restrictions on expressiveness

As we already hinted at earlier, the expressiveness of the specification language is restricted at several points. That rules out specifications of certain kinds of behavior, for good or bad. Most notably, we deliberately ruled out general non-determinism, as already explained in Section 3.1. On the one hand, such restrictions keep the syntax and semantics of the specification language simple and have the potential to enable additional reasoning about specifications. On the other hand, they are motivated by our interpretation of a single specification as describing exactly one pattern of behavior, and one we actually consider useful in our educational setting at that. Concerning the latter point, we for example would like the specified pattern to enforce actual interactivity. That is, at its core the behavior should rely on a (somewhat alternating) sequence of reads and writes and should not be expressible in a different way. Consider, for example, the following Haskell program:

$\text{main} :: IO ()$	$\text{loop} :: Int \rightarrow IO ()$
$\text{main} = \text{do } n \leftarrow \text{readLn}$	$\text{loop } n \mid n \leq 0 = \text{return } ()$
$\text{loop } n$	$\text{loop } n = \text{print } n \gg \text{loop } (n - 1)$

A specification corresponding to this program is not expressible in our DSL (reading and writing integer values), and that was a design goal. The non-expressibility is due to the facts that in our specifications an iteration process can only end based on some predicate over the global variable state (contents of variables, their history) and that only inputs can alter this state, leaving the above kind of “output-driven loops” impossible to encode. According to our motivation, this restriction is a good thing. We only want inherently interactive behavior to be expressible, whereas the above program can be rewritten as

$$\begin{array}{ll} \text{main} :: IO () & \text{loop} :: Int \rightarrow String \\ \text{main} = \text{do } n \leftarrow \text{readLn} & \text{loop } n \mid n \leq 0 = "" \\ \quad \text{print } (\text{loop } n) & \text{loop } n = \text{show } n ++ "\n" ++ \text{loop } (n - 1) \end{array}$$

with exactly one input action at the beginning, then all computation happening in a non-I/O loop, and exactly one output action at the end, which overall is not an attractive teaching example when we actually want to cover interactive I/O in Haskell and how programs must be structured to organize sequences of input and output actions in interesting ways.

If we for a moment would lift our restriction to just use integers as inputs and outputs, we could write a specification like $[\triangleright n]^{\mathbb{Z}}[\{\text{loop}(n)\} \triangleright]$ for such behavior, with the second version of *loop* above. From this it is immediately clear that the interactive core of the program/task here is almost trivial, so we do not want it. Put differently, we wanted to make sure that there are as few as possible ways in our DSL to encode essentially non-interactive computations in only seemingly interactive guise. Note that even if we do indeed allow strings for input and output, as we do for practical usage in our course, we can still prevent creation of such “boring tasks” via the DSL by controlling which functions are allowed in terms for conditions and outputs, for example preventing something like *loop* from appearing there as it does in the hypothetical specification $[\triangleright n]^{\mathbb{Z}}[\{\text{loop}(n)\} \triangleright]$. We will see in the next section that this is encoded in the definition of valid terms by parameterizing it over some set of available functions.

4 Syntax

Figure 1 gives the full syntax of our language by defining the set *Spec* of all specifications as well as the term language used for the description of output values and branching conditions. We distinguish different subsets of the set of all terms by a subscript indicating the type of value a term evaluates to. For example, $T_{\mathbb{Z}}$ denotes the set of all terms that evaluate to an integer and $T_{\mathbb{B}}$ the set of terms evaluating to a Boolean value. We write $[\mathbb{Z}]$ instead of \mathbb{Z}^* for sequences of integers here, emphasizing that we are dealing with list values as opposed to words over integers.

With the exception of (Write), the rules are straightforward; there, we require that the set of possible output values contains at least one real term. That is, we deliberately rule out actions of the forms $[\{\} \triangleright]$ and $[\{\varepsilon\} \triangleright]$. Giving an empty set of terms would always result in an unsatisfiable specification; giving a singleton set containing ε would be equivalent to $\mathbf{0}$.

For terms we restrict ourselves to a not further specified set *Func* of functions and the elements of some variable set *Var* used in the specification, or more precisely, the different access variants of those variables. In principle we could choose any set of functions we want, as long as evaluation of terms is well-defined. Making *Func* itself a parameter of the specification language is useful if one wants to enforce some conditions to guarantee certain properties of specifications. We could, for example, choose *Func* to be the set of all total functions if we want some guarantees on termination. Or if we are interested in automatically generating random specifications for exercise tasks, we can control to some extent what kind of tasks are generated by choosing different sets for *Func*.

$\frac{\tau \subseteq \mathbb{Z} \quad x \in \text{Var}}{[\triangleright x]^\tau \in \text{Spec}} \text{ (Read)}$	$\frac{s_1 \in \text{Spec} \quad s_2 \in \text{Spec}}{s_1 \cdot s_2 \in \text{Spec}} \text{ (Seq)}$	
$\frac{\Theta \subseteq (T_{\mathbb{Z}} \cup \{\varepsilon\}), \Theta \setminus \{\varepsilon\} \neq \emptyset}{[\Theta \triangleright] \in \text{Spec}} \text{ (Write)}$	$\frac{s_1 \in \text{Spec} \quad s_2 \in \text{Spec} \quad c \in T_{\mathbb{B}}}{s_1 \angle c \searrow s_2 \in \text{Spec}} \text{ (Branch)}$	
$\frac{s \in \text{Spec}}{s \rightarrow^{\mathbf{E}} \in \text{Spec}} \text{ (Till-}\mathbf{E})$	$\frac{}{\mathbf{E} \in \text{Spec}} \text{ (LoopExit)}$	$\frac{}{\mathbf{0} \in \text{Spec}} \text{ (Nop)}$
.....		
$\frac{x \in \text{Var}}{x_C \in T_{\mathbb{Z}}} \text{ (Current)}$	$\frac{x \in \text{Var}}{x_A \in T_{[\mathbb{Z}]}} \text{ (All)}$	
$\frac{f : D_1 \times \dots \times D_n \rightarrow D \quad t_1 \in T_{D_1}, \dots, t_n \in T_{D_n} \quad f \in \text{Func}}{f(t_1, \dots, t_n) \in T_D} \text{ (Function)}$		

Figure 1: Syntax of specifications (top) and terms (bottom)

We make the following assumptions regarding the structure and semantic well-formedness of specifications:

- A variable x_C does not occur in a term before x occurred in an input action, since this would make the evaluation of that term fail. A corresponding issue does not exist for x_A since we can define it to initially evaluate to the empty list.
- Every loop eventually reaches an occurrence of \mathbf{E} (given the right sequence of input values). If we are not interested in actual termination, we can alternatively loosen this so that every $\rightarrow^{\mathbf{E}}$ just “binds” an occurrence of \mathbf{E} , i.e., an exit marker is present but we do not analyze the branching conditions to reach it.

The purpose of this restriction is to let specifications only ever describe finite behavior. In practice, this condition does not necessarily have to be checked. Since we cannot prevent students from submitting programs with infinite loops, we usually work with timeouts. When we, for example, test a model solution against an accidentally non-terminating specification during our task development activities, these very same timeouts will “discover” this fact. It is, therefore, unlikely that we will pose tasks based on such ill-formed specifications.

Additionally, sequential composition of specifications is defined to be associative, i.e., $s_1 \cdot (s_2 \cdot s_3) = (s_1 \cdot s_2) \cdot s_3$, therefore we can just write $s_1 \cdot s_2 \cdot s_3$ instead, or indeed $s_1 s_2 s_3$. Also, $\mathbf{0}$ is the neutral element of sequential composition, meaning $\mathbf{0} \cdot s = s = s \cdot \mathbf{0}$. Moreover, we define sequential composition to have higher precedence than branching and $\rightarrow^{\mathbf{E}}$ to have higher precedence than sequential composition, i.e., $s_1 \cdot s_2 \angle c \searrow s_3 = (s_1 \cdot s_2) \angle c \searrow s_3$ and $s_1 \cdot s_2 \rightarrow^{\mathbf{E}} = s_1 \cdot (s_2 \rightarrow^{\mathbf{E}})$.

Also note that we have no real notion of variable scope in our language. Every variable is global and changes to it will be visible at every point in time after that change occurred.

5 Semantics

Recall that we want to analyze I/O programs by simulating program runs and then inspecting the resulting traces. In Section 2 we gave a data type for representing such traces. However, since we restrict ourselves

$$\begin{aligned}
\text{accept}([\triangleright x]^\tau \cdot s', k)(t, \Delta) &= \begin{cases} \text{accept}(s', k)(t', \text{store}(x, v, \Delta)) & , \text{ if } t = ?v t' \wedge v \in \tau \\ \text{False} & , \text{ otherwise} \end{cases} \quad (1a) \\
\text{accept}([\Theta \triangleright] \cdot s', k)(t, \Delta) &= \begin{cases} \text{accept}([\Theta \setminus \{\varepsilon\}] \triangleright \cdot s', k)(t, \Delta) & , \text{ if } \varepsilon \in \Theta \\ \quad \vee \text{accept}(s', k)(t, \Delta) & \\ \text{accept}(s', k)(t', \Delta) & , \text{ if } \varepsilon \notin \Theta \wedge t = !v t' \\ & \wedge v \in \text{eval}(\Theta, \Delta) \\ \text{False} & , \text{ otherwise} \end{cases} \quad (1b) \\
\text{accept}((s_1 \angle c \triangleright s_2) \cdot s', k)(t, \Delta) &= \begin{cases} \text{accept}(s_2 \cdot s', k)(t, \Delta) & , \text{ if } \text{eval}(c, \Delta) = \text{True} \\ \text{accept}(s_1 \cdot s', k)(t, \Delta) & , \text{ otherwise} \end{cases} \quad (1c) \\
\text{accept}(s^{\rightarrow E} \cdot s', k)(t, \Delta) &= \text{accept}(s, k')(t, \Delta) & (1d) \\
&\quad \text{with } k'(cont) = \begin{cases} \text{accept}(s, k') & , \text{ if } cont = \text{End} \\ \text{accept}(s', k) & , \text{ if } cont = \text{Exit} \end{cases} \\
\text{accept}(\mathbf{E} \cdot s', k)(t, \Delta) &= k(\mathbf{Exit})(t, \Delta) & (1e) \\
\text{accept}(\mathbf{0}, k)(t, \Delta) &= k(\mathbf{End})(t, \Delta) & (1f) \\
k_I(cont)(t, \Delta) &= \begin{cases} \text{True} & , \text{ if } cont = \text{End} \wedge t = \text{stop} \\ \text{False} & , \text{ if } cont = \text{End} \wedge t \neq \text{stop} \\ \text{error} & , \text{ if } cont = \text{Exit} \end{cases}
\end{aligned}$$

Figure 2: Trace acceptance

to I/O programs that only read and write integer values in the formalization, we will use here the more restrictive and more compact version of traces already informally introduced in Section 3.2.

Therefore a trace is a sequence of values $v_i \in \mathbb{Z}$ marked either as input, denoted $?v_i$, or as output, denoted $!v_i$. Each trace ends with the element *stop* indicating the end of execution. We use Tr to denote the set of all traces (regardless of a certain program or specification).

5.1 Matching traces and specifications

In order to determine whether a given trace is valid for a given specification, we define a function *accept* such that $\text{accept}(s, _)(t, _) = \text{True}$ exactly if a given trace $t \in Tr$ exhibits behavior specified by $s \in Spec$. Figure 2 gives the detailed definition of this function. Other than the trace and the specification that the trace is checked against, the function also takes a variable environment Δ and a continuation function k as additional inputs. The continuation k takes care of managing the current iteration context, as informally described when discussing how to check an iteration in Section 3.2. It encodes how to proceed if we *Exit* from the current context to an outer one or if we just *End* a round inside the current context and continue with another round of the iteration.

The functions *eval* and *store* evaluate terms and store values in the environment, respectively. Their definitions are straightforward and are therefore omitted here. We write $\text{eval}(\Theta, \Delta)$ for evaluating, under Δ , every term in a set Θ .

At its core, *accept* traverses a specification from left to right, consuming matching trace elements and updating variables along the way. If we are left with exactly the empty specification or if we encounter

an exit marker of some iteration, we call k with the appropriate argument to indicate whether we want to continue the iteration process or exit from it, and pass the remaining trace and current variable environment along. Note that this also covers the case where we completely consumed the specification in the outermost context, i.e., the initial one. If the trace is then also fully consumed already, the acceptance match is successful. The initial continuation k_I is defined such that in the case of End it performs exactly this check (see Figure 2) and therefore finishes the computation.

It is important to note that the cases (1a) to (1e) are all defined with the associativity of sequential composition in mind. Therefore we do not have a case for $accept((s \cdot s') \cdot s'', k)(t, \Delta)$ since this can always be rewritten as $accept(s \cdot (s' \cdot s''), k)(t, \Delta)$. Moreover, the fact that $\mathbf{0}$ is the neutral element of sequential composition means that the rules can also be applied to specifications like $[\triangleright x]^\tau$ by expanding them to $[\triangleright x]^\tau \cdot \mathbf{0}$. By the same reasoning, we do not need an explicit rule like $accept(\mathbf{0} \cdot s', k)(t, \Delta) = accept(s', k)(t, \Delta)$. Additionally, (1f) is only applicable if the specification is exactly $\mathbf{0}$. Otherwise this would allow for the initiation of another round of iteration at any point in the specification, since for example $s \cdot s'$ can always be rewritten as $s \cdot \mathbf{0} \cdot s'$. Note that for \mathbf{E} in (1e) the situation is different. Even though it might seem strange at first, since one would probably never write a specification containing “dead code” s' like this, cases exist where a specification of the form $\mathbf{E} \cdot s'$ does arise. Consider, for example, $((s_1 \angle c_1 \triangleright (s_2 \angle c_2 \triangleright \mathbf{E})) \cdot s_3)^{\rightarrow, \mathbf{E}}$, which is a perfectly reasonable specification to write. Now in order to leave the loop via that \mathbf{E} , both conditions must evaluate to *True*, and by (1c) we are now left with matching against $\mathbf{E} \cdot s_3$. So in order to correctly handle such specifications, (1e) needs to discard everything following an occurrence of \mathbf{E} .

5.2 Program correctness

Using the *accept*-function, we can formulate a notion of program (trace) correctness. A program is considered to be an implementation of a specification s if and only if for every execution trace t that can arise from the program, it holds $accept(s, k_I)(t, \Delta_I) = True$, where Δ_I is the initial environment containing no values, i.e., $eval(x_A, \Delta_I)$ evaluates to the empty list for every variable x occurring in s , and k_I is the initial continuation as shown in Figure 2.

6 Testing framework

Now that we defined syntax and semantics of our DSL, how can we actually test programs against specifications? Recall that we want to be able to automatically generate the following three components from a given specification:

- A generator of input sequences that respect the task’s invariants.
- A way of checking whether a trace exhibits the desired behavior.
- A method of providing feedback in case the actual behavior did not match the expectations (e.g., a correct run on the respective input sequence).

With the *accept*-function from above, we could already cross off the second item on the list. For the generation of simple feedback, we could modify *accept* such that it returns additional information in case the matching is unsuccessful. However, for more elaborate feedback, like example runs, we need another, less localized, approach. One way to enable more general feedback is to take the equation $accept(s, k_I)(t, \Delta_I) = True$, for some specification s , and solve for t . A solution for t is a valid program run. Additionally, managing to compute such a t also immediately gives us a valid input sequence for the specification, thereby meeting our remaining requirement. We can further generalize this idea by not

computing a concrete trace (choosing one possibility for each output action) but rather a general structure representing all accepted runs for a given input sequence. Such a structure, called a generalized trace, then represents an all-encompassing test case for a certain fixed input sequence. We will use generalized traces to build all of our three required components.

Given a mechanism to actually compute such test cases, we can test programs by repeatedly running the following steps:

1. generate a generalized trace t_g from the given specification s ,
2. extract the input sequence from t_g ,
3. run the program under testing on that input sequence, resulting in trace t_p ,
4. check whether t_p is one of the traces represented by t_g .

Before we will look at how to compute generalized traces from specifications, we will first define what generalized traces are exactly and how they relate to ordinary traces. Then we will give the definition of a modified version of *accept* that describes the complete set of generalized traces for a certain specification. Even though this set is usually infinite, the function is constructed in a way that lets us derive a method for generating specific generalized traces.

6.1 Generalized traces

Analogously to how we allowed different potential output values in a single output action in the specification language, we now allow different values in each output step of a trace.

Consider the specification $[\triangleright x]^{\mathbb{Z}}[\{\varepsilon, x_C\} \triangleright]$. For arbitrary but fixed input value $v \in \mathbb{Z}$, the valid traces for this specification are $?v \text{ stop}$ and $?v !v \text{ stop}$. The “test case for a fixed v ” should allow any of those two traces. Therefore the generalized trace for this specification under a fixed input v is $?v !\{\varepsilon, v\} \text{ stop}$.

Moreover, and unlike for specifications, we fuse sequences of adjacent output steps into a single output step, then containing (possibly various) sequences of values. For example, if we had a specification like $[\triangleright x]^{\mathbb{Z}}[\{\varepsilon, x_C\} \triangleright][\{\varepsilon, x_C\} \triangleright]$, without this normalization we would end up with $?v !\{\varepsilon, v\} !\{\varepsilon, v\} \text{ stop}$ as our generalized trace. Since we cannot distinguish between the two outputs anyway, due to the black-box nature of our approach, we just combine them into a single action. The normalized trace in this case is therefore $?v !\{\varepsilon, v, vv\} \text{ stop}$. This trace completely covers the possible behavior of a correct program for specification $[\triangleright x]^{\mathbb{Z}}[\{\varepsilon, x_C\} \triangleright][\{\varepsilon, x_C\} \triangleright]$ and fixed input v .

The set of all *generalized* traces in general, Tr_G , is defined by the following rules:

$$\frac{v \in \mathbb{Z} \quad t \in Tr_G}{?v t \in Tr_G} \quad \frac{v \in \mathbb{Z} \quad t \in Tr_G \quad V \subseteq \mathbb{Z}^* \quad V \setminus \{\varepsilon\} \neq \emptyset}{!V ?v t \in Tr_G}$$

$$\frac{}{\text{stop} \in Tr_G} \quad \frac{V \subseteq \mathbb{Z}^* \quad V \setminus \{\varepsilon\} \neq \emptyset}{!V \text{ stop} \in Tr_G}$$

We now need a way to determine whether an ordinary trace is *covered* by a generalized trace. That is, we want to check whether the concrete run represented by the ordinary trace is contained in the set of runs the generalized trace is representing. Since in a generalized trace consecutive outputs are fused, we first normalize ordinary traces in that respect as well if we want to compare them to generalized ones. We normalize ordinary traces by way of the function $[\cdot] : Tr \rightarrow Tr_G$ that embeds Tr into Tr_G , the image being exactly the subset of Tr_G with only singleton sets, of non-empty words, in output steps:

$$\begin{aligned} \lceil t \rceil &= \lceil t \rceil_{\varepsilon} & \lceil !v t' \rceil_w &= \lceil t' \rceil_{wv} \\ \lceil ?v t' \rceil_w &= \begin{cases} ?v \lceil t' \rceil_{\varepsilon} & , \text{ if } w = \varepsilon \\ !\{w\} ?v \lceil t' \rceil_{\varepsilon} & , \text{ otherwise} \end{cases} & \lceil stop \rceil_w &= \begin{cases} stop & , \text{ if } w = \varepsilon \\ !\{w\} stop & , \text{ otherwise} \end{cases} \end{aligned}$$

Now we can define the covering relation $\prec \subseteq [Tr] \times Tr_G$ as follows:

$$\frac{t_1 \prec t_2}{?v t_1 \prec ?v t_2} \quad \frac{w \in V \quad t_1 \prec t_2}{!\{w\} t_1 \prec !V t_2} \quad \frac{\varepsilon \in V \quad t_1 \prec t_2}{t_1 \prec !V t_2} \quad \frac{}{stop \prec stop}$$

Note that, due to the typing of this relation, neither the trace on the left nor that on the right of any occurrence of \prec can contain directly consecutive output steps. The definition of \prec is also what allows us to generate feedback in case we found an invalid program run. If $\lceil t_p \rceil \not\prec t_g$, for some program trace t_p and a generalized trace t_g , it could hypothetically be for one of three reasons. First, it could be because the two traces, at some position, disagree on the value that is read in. This case does not occur in our setting, since we use the input sequence of the generalized trace to construct the ordinary trace. Therefore in our setting there are actually only two possible reasons for why a trace is not covered. Either the structure of the traces does not line up, i.e., their first constructor differs and the respective step in t_g cannot be skipped (i.e., it is not $!\{\varepsilon, \dots\}$), or for some output step there is a complete or partial mismatch between the expected and actual output, i.e., $w \notin V$. In both cases a simple message like “*Expected: ..., but got: ...*” or “*The output value ... is not covered by ...*” can be generated and presented to the user, along with the input that triggered the error. Additionally, the generalized trace can be given (in some pretty-printed form) to showcase all possible runs of the program on that particular input sequence. Depending on the application setting it might however be useful to restrict this to just one particular example run, especially when the target specification is hidden, which might be the case in an educational setting.

6.2 Computing generalized traces

Equipped with the definition of generalized traces, we now need a way to actually compute them for a given specification. The basic idea from the beginning of this section was to solve for t in the equation $accept(s, k_I)(t, \Delta_I) = True$. We can do this by evaluating $accept$ with t unfixed and on demand extending the trace with the appropriate steps such that we never fall into the *False* case. To do this, we need to pick random elements of the annotated set for each input action and choose one of the possible outputs of each write action. This will then give us a particular non-generalized trace that matches the specification.

Let us take $[\triangleright n]^{\mathbb{N}}([\{\varepsilon, n_C - len(x_A)\} \triangleright][\triangleright x]^{\mathbb{Z}} \angle len(x_A) = n_C \triangle \mathbf{E}) \rightarrow^E [\{sum(x_A)\} \triangleright]$ as an example. Running $accept$ “in reverse” could result, for example, in one of the following traces: $?1 ?4 !4 stop$, $?1 !1 ?4 !4 stop$, $?2 ?3 ?7 !10 stop$. If we want to get a generalized trace instead, all we need to do is to not choose a single output action but rather extend the trace with all possible outputs. This would, for example, result in merging the first two traces from above into a single generalized trace $?1 !\{\varepsilon, 1\} ?4 !\{4\} stop$. If we now do also not choose individual inputs from the respective sets, we get a function that describes the set of all possible generalized traces for a given specification. For the specification above, this set would be the following one:

$$\begin{aligned} S = \{ & ?0 !\{0\} stop, \\ & ?1 !\{\varepsilon, 1\} ?v_1 !\{v_1\} stop, \\ & ?2 !\{\varepsilon, 2\} ?v_1 !\{\varepsilon, 1\} ?v_2 !\{v_1 + v_2\} stop, \\ & \dots \mid v_1, v_2, \dots \in \mathbb{Z} \} \end{aligned}$$

$$\begin{aligned}
\text{traceSet}([\triangleright x]^\tau \cdot s', k)(\Delta) &= \bigcup_{v \in \tau} \{?v\} \cdot \text{traceSet}(s', k)(\text{store}(x, v, \Delta)) & (2a) \\
\text{traceSet}([\ominus \triangleright] \cdot s', k)(\Delta) &= \text{eval}(\ominus, \Delta) \odot \text{traceSet}(s', k)(\Delta) & (2b) \\
\text{traceSet}((s_1 \angle c \triangleright s_2) \cdot s', k)(\Delta) &= \begin{cases} \text{traceSet}(s_2 \cdot s', k)(\Delta), & \text{if } \text{eval}(c, \Delta) = \text{True} \\ \text{traceSet}(s_1 \cdot s', k)(\Delta), & \text{otherwise} \end{cases} & (2c) \\
\text{traceSet}(s \xrightarrow{\mathbf{E}} \cdot s', k)(\Delta) &= \text{traceSet}(s, k')(\Delta) & (2d) \\
&\quad \text{with } k'(cont) = \begin{cases} \text{traceSet}(s, k'), & \text{if } cont = \text{End} \\ \text{traceSet}(s', k), & \text{if } cont = \text{Exit} \end{cases} \\
\text{traceSet}(\mathbf{E} \cdot s', k)(\Delta) &= k(\text{Exit})(\Delta) & (2e) \\
\text{traceSet}(\mathbf{0}, k)(\Delta) &= k(\text{End})(\Delta) & (2f) \\
k_I^T(cont)(\Delta) &= \begin{cases} \{stop\} & , \text{if } cont = \text{End} \\ \text{error} & , \text{if } cont = \text{Exit} \end{cases} \\
V \odot T' &= \bigcup_{t' \in T'} \begin{cases} \{!(V \cdot V') t''\} & , \text{if } t' = !V' t'' \\ \{!V t'\} & , \text{otherwise} \end{cases}
\end{aligned}$$

Figure 3: Trace set generation (differences to Figure 2 in gray)

In Figure 3 the definition of this trace set generation function is given. The most notable conceptual deviation from the *accept*-function, apart from turning an acceptor into a set generator, is case (2b). It avoids the case distinction from the corresponding part in Figure 2 since we consider all possible output values of a write action and combine consecutive output values into single words. The notation $V_1 \cdot V_2$ in the definition of the corresponding helper (and elsewhere in the figure, for traces) denotes language concatenation. For notational simplicity, we also assume that $\text{eval}(\varepsilon, \Delta) = \varepsilon$.

The connection between *accept* and *traceSet* can be given as follows: Let $s \in \text{Spec}$ and $t \in \text{Tr}$, then $\text{accept}(s, k_I)(t, \Delta_I) = \text{True}$ if and only if there exists a $t' \in \text{traceSet}(s, k_I^T)(\Delta_I)$ such that $\lceil t \rceil \prec t'$.

Additionally, with *traceSet* we do not only have a generator for generalized traces but also an interpreter. If we generate a generalized trace not by choosing inputs at random, but from a given sequence, we essentially execute a specification for that input sequence.

7 Implementation

We have started to build an EDSL for our design in Haskell and implemented the testing approach explained thus far⁵. Within our framework, we provide a data type for describing a *Specification*, the IO_{rep} type from Section 2, and a function $\text{fulfills} :: IO_{rep} \rightarrow \text{Specification} \rightarrow \text{Property}$ that constructs a value of QuickCheck's *Property* type given a program and a specification. When tested, this property will generate generalized traces for the given specification with the help of a QuickCheck generator based on the *traceSet*-function. From such a trace we then extract the sequence of generated inputs and

⁵A demo showcasing the system in the context of the automatic grading system we use is available at <https://autotool.fmi.iw.uni-due.de/tfpie19>. Note that the specifications used in that demo differ slightly from the ones presented here, as they additionally handle string inputs and outputs. The EDSL itself is available at <https://github.com/fmidue/IOTasks>.

use run_{rep} , also shown in Section 2, to generate a *Trace*. Lastly, using \prec , we check if this result trace is covered by the initial generalized trace and generate an appropriate error/feedback message if that is not the case. Note that we did not need to implement *accept* since, as already stated in the previous section, *traceSet* together with \prec can be used instead.

If we now, for example, take the specification $[\triangleright n]^{\mathbb{N}}([\triangleright x]^{\mathbb{Z}} \angle len(x_A) = n_C \triangle \mathbf{E}) \rightarrow^E [\{sum(x_A)\} \triangleright]$, we can express it in the EDSL like so:

```
spec :: Specification
spec = readInput "n" nats
      <>
      tillExit
      (branch ((λxs n → length xs == n) <$> getAll @Int "x" <*> getCurrent "n")
              (readInput "x" ints)
              exit)
      <>
      writeOutput [sum <$> getAll @Int "x"]
```

We define terms needed for branching and outputs using *Applicative*-style and with the help of the accessors *getCurrent* and *getAll* that correspond to the *C* and *A* subscripts of variables.

Now assume we have a program $prog :: IO_{rep} ()$. We can check this program against the specification by running *quickCheck (prog 'fulfills' spec)*. This can either result in a successful test run where QuickCheck did not generate a counterexample:

```
> quickCheck (prog 'fulfills' spec)
+++ OK, passed 100 tests.
```

or a counterexample is found and we get an error message telling us what went wrong:

```
> quickCheck (prog 'fulfills' spec)
*** Failed! Falsifiable:
Input sequence: ?7 ?2 ?9 ?1 ?-5 ?1 ?7 ?1
Expected run (generalized): ?7 ?2 ?9 ?1 ?-5 ?1 ?7 ?1 !{16} stop
Actual run: ?7 ?2 ?9 ?1 ?-5 ?1 ?7 !15 stop
Error:
AlignmentMismatch:
  Expected: ?1
  Got: !15
```

Here the program we are testing reads one value less than it should, resulting in a missing input step in its trace. When checking whether the generalized trace covers the program trace here, we get stuck when checking $!15\ stop$ against $?1!{16}\ stop$. This type of mismatch, as already mentioned in Section 6.1, is one of two possibilities for the coverage check to go wrong in our setting. The other possible source of a mismatch manifests when the program writes an output that is not part of the set of valid outputs at the respective position. For such a program, which for example does not include the last read number into the sum, the error message looks like this:

```
> quickCheck (prog 'fulfills' spec)
*** Failed! Falsifiable:
Input sequence: ?3 ?-2 ?0 ?6
Expected run (generalized): ?3 ?-2 ?0 ?6 !{4} stop
```

```
Actual run: ?3 ?-2 ?0 ?6 !-2 stop
Error:
  OutputMismatch:
    the value -2 is not covered by {4}
```

Note that, as stated in Section 6.1, there could theoretically also be a mismatch between read values, but since we derive the input sequence from the generalized trace, this cannot occur in our setting.

In both of the error messages shown above, the input sequences all contain only relatively small numbers. This is not by chance. When constructing generalized traces, we currently only draw inputs from the range of -10 to 10 for integer and 0 to 10 for natural numbers. This obviously reduces our capabilities to catch certain errors in a program. On the other hand, since we currently choose elements from these sets completely at random, larger value sets would often result in the generation of very long input sequences or in generation failing to terminate in reasonable time. This is especially the case if specifications include very particular branching conditions. Consider, for example, the specification $([\triangleright x]^{\mathbb{Z}} \angle sum(x_A) = c \searrow \mathbf{E})^{-\mathbf{E}}$ for some constant c . In order to reach the desired sum and end the loop, at some point the next input has to be exactly the difference between c and the current sum. Hitting this one element at random is extremely unlikely when drawing from large sets. We therefore usually fail to generate even a single trace for such specifications. Additionally, we are currently not able to give any guarantees when it comes to coverage of branches or other measures of quality for test case distributions. These are points we are planning to address in the future.

For the same reasons, we do not shrink the found counterexample, like QuickCheck normally does, to obtain a small or even minimal counterexample. In our example the size of the test case, i.e., of the generalized trace, depends on the first input value, since it determines the number of required executions of the loop body. Choosing a smaller number for that input would yield a candidate for a smaller counterexample. But in general, such conditions can be far more complicated. A possible solution to this problem might be to not rely on random test case generation and shrinking in the first place but instead use an enumerative testing approach [2, 6], systematically generating test cases up to a certain size. However, we currently have not done any concrete work in this direction.

8 Related work

The general mechanism for building inspectable representations of side-effecting programs [8] is provided in the Haskell IOSpec library⁶. It supports not only console I/O but also forking processes, mutable references, and software transactional memory. However, its API is very minimal and no higher-level abstractions currently exist.

Another testing approach that deals with stateful computations is Quviq QuickCheck [3, 4] for Erlang⁷. Instead of testing specific programs, like we do, they test stateful APIs. A specification of such an API is a semantic model, given in Erlang, of the API together with pre- and post-conditions for each stateful action. Testing is then done by generating random sequences of actions based on the pre-conditions and checking the result of the actual API calls against the model and post-conditions. Any found sequence of API calls that do not behave in accordance with the semantic model is simplified via shrinking to obtain a small counterexample.

Due to the large number of existing automatic task grading and assessment tools, we cannot give a complete overview here. A survey (with a focus on feedback generation) of different automatic as-

⁶<https://hackage.haskell.org/package/IOSpec>

⁷A Haskell version can be found at <http://hackage.haskell.org/package/quickcheck-state-machine>.

assessment tools for programming tasks is presented by Keuning et al. [5]. Most tools use some form of automatic testing, either on specified test cases or by comparing submissions to the results of sample solutions. Additionally, a number of tools use program transformation or static analysis techniques to determine how a program deviates from a sample solution. Task specification is usually done through unit or property tests or by providing sample solutions in the respective programming language. As far as we can tell, no existing system is using a formal specification language for defining intended behavior. Some systems support automatic generation of exercise tasks, but this is usually restricted to gap-filling tasks derived from sample solutions.

9 Future work

9.1 Improvements to testing

Our testing framework does not formulate any special search strategy for finding test cases. It is easy to get stuck while searching for a test case when the random values we choose for inputs do not lead to termination of iterations. Also, no guarantees on coverage of the complete range of described behavior are given. Reliable application in a general setting requires that we find solutions for these shortcomings.

9.2 Moving beyond testing

Automatic testing of programs is a nice first step when it comes to automating parts of educational activities. We already have a list of areas in mind for which we would also like some automatization, and our DSL is designed partly with these possibilities in mind:

- **Task generation.** We can automatically generate syntactically valid specifications as the basis for new tasks. However, we lack a way of controlling the complexity of such specifications. The specification language provides ways to control expected input values and some guidance is possible by restricting the term language (via the set *Func*). A way of defining and describing meta properties could be used, for example, to provide students with individual tasks while assuring fairness in terms of difficulty etc.
- **Sample solutions.** It should be fairly obvious that for every expression in our specification language we can automatically generate a (Haskell) program with the respective behavior. Unfortunately, such a program is most likely not an ideal sample solution to show to students. Manual attempts at transforming such naively generated programs into idiomatic ones suggest that this might be possible in general using basic rewriting and optimization techniques.
- **Generation of helpful feedback.** Currently the feedback we gain from a failing test case is limited. There are no real pointers to the root of the problem but just a basic counterexample for which the program behaves in a certain wrong way. Inferring, for example, a specification for the wrongly behaving program and comparing it to the target might reveal the source of an error, enabling us to provide more precise feedback.
- **Alternative domains.** The general approach of defining a language for specifications from which we can generate black-box tests for free form solutions can potentially be adapted to other programming task domains as well. This could include additional I/O capabilities, like reading and writing files. But adapting the approach to pure domains like transformations of lists using a certain set of predefined combinators seems possible as well. This would allow us to apply automatic task generation etc. in pure contexts.

10 Conclusion

We presented a formal language for specifying the interactive behavior of console I/O programs. By doing so, we gain the ability to automatically generate tests and test cases to probabilistically check whether a program submission is correct. This is a significant improvement to our ability to state and automatically grade exercises. Additionally, the fact that we can manipulate the formal descriptions of behavior programmatically opens up a wide range of possibilities for further automatization and analyses.

References

- [1] Koen Claessen & John Hughes (2000): *QuickCheck: a lightweight tool for random testing of Haskell programs*. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ACM, pp. 268–279, doi:10.1145/351240.351266.
- [2] Jonas Duregård, Patrik Jansson & Meng Wang (2012): *Feat: Functional Enumeration of Algebraic Types*. In: *Proceedings of the Fifth ACM SIGPLAN Haskell Symposium*, ACM, pp. 61–72, doi:10.1145/2364506.2364515.
- [3] John Hughes (2007): *QuickCheck Testing for Fun and Profit*. In: *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages, LNCS 4354*, Springer, pp. 1–32, doi:10.1007/978-3-540-69611-7_1.
- [4] John Hughes (2016): *Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane*. In: *A List of Successes That Can Change the World – Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday, LNCS 9600*, Springer, pp. 169–186, doi:10.1007/978-3-319-30936-1_9.
- [5] Hieke Keuning, Johan Jeuring & Bastiaan Heeren (2019): *A Systematic Literature Review of Automated Feedback Generation for Programming Exercises*. *ACM Transactions on Computing Education* 19(1), pp. 3:1–3:43, doi:10.1145/3231711.
- [6] Colin Runciman, Matthew Naylor & Fredrik Lindblad (2008): *SmallCheck and Lazy SmallCheck – automatic exhaustive testing for small values*. In: *Proceedings of the First ACM SIGPLAN Haskell Symposium*, ACM, pp. 37–48, doi:10.1145/1411286.1411292.
- [7] Marcellus Sieburg, Janis Voigtländer & Oliver Westphal (2019): *Automatische Bewertung von Haskell-Programmieraufgaben*. In: *Proceedings of the Fourth Workshop “Automatische Bewertung von Programmieraufgaben”*, GI, pp. 19–26, doi:10.18420/abp2019-3.
- [8] Wouter Swierstra & Thorsten Altenkirch (2007): *Beauty in the Beast – A Functional Semantics for the Awkward Squad*. In: *Proceedings of the 11th ACM SIGPLAN Haskell Workshop*, ACM, pp. 25–36, doi:10.1145/1291201.1291206.
- [9] J. Waldmann (2017): *Automatische Erzeugung und Bewertung von Aufgaben zu Algorithmen und Datenstrukturen*. In: *Proceedings of the Third Workshop “Automatische Bewertung von Programmieraufgaben”*, CEUR Workshop Proceedings 2015, CEUR-WS.org.
- [10] Oliver Westphal & Janis Voigtländer (2020): *Implementing, and Keeping in Check, a DSL Used in E-Learning*. In: *Proceedings of the 15th International Symposium on Functional and Logic Programming, LNCS 12073*, Springer, doi:10.1007/978-3-030-59025-3_11.