Specifying Teletype Behavior for the Automated Handling of Exercises on Interactive Haskell Programs

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 teletype I/O programs. The design is driven by the concrete application case of testing basic interactive Haskell programs written by students in our course on programming paradigms, but is in general language independent. Specifications are structurally similar to regular expressions, but are augmented with global variables that track state and history of program runs, enabling expression of a wide range of dynamic behavior. We give a semantics for our language based on program traces. From this semantics we derive rules for describing the set of all traces valid for a given specification in a way that enables us to mechanically check program behavior against specifications in a probabilistic fashion. Furthermore, we claim that the language is useful not only for testing but for a range of related activities like generation of exercises and sample solutions or for providing helpful feedback.

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. Since checking submissions by hand is tedious and error prone, we employ an e-learning system [3] that automatically tests submissions against sets of QuickCheck [1] specifications wherever possible. An added benefit for students is that they get immediate feedback and can revise their submissions accordingly and incrementally.

This approach works well for tasks asking for implementations of pure functions. But doing the same for tasks involving I/O is significantly more complicated. While Swierstra and Altenkirch [4] showed how one can change the monad underlying an I/O program in order to get an inspectable representation of what the program is doing, thus in principle allowing to check properties by using QuickCheck or similar tools on command execution traces, practical application is cumbersome. Specifically, for every I/O exercise task we want to handle using this approach, we currently need to implement the following components:

- A generator for valid input sequences.
- A predicate validating if a given program trace satisfies the desired behavior for a given input sequence.
- A way to provide feedback in case the behavior did not match the task, usually by showing a run/trace of a correct sample solution program on the relevant input sequence.

Since we usually allow students some degree of freedom when it comes to how their program should prompt for input, in what form exactly it should print its computation results, whether there are optional output messages, etc., these components have to cover a lot of different cases. Moreover, they are

To appear in EPTCS.

usually 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 kept in check by designing a domain-specific language for specifying interactive program behavior in a way that allows the components listed above to be generated automatically. We therefore make the following contributions:

- We describe the syntax (Section 4) and semantics (Section 5) of a language suitable for specifying the interactive behavior of simple teletype programs. Key features of our approach are history aware variables that allow access to all values previously read into them and an encoding of optional or ambiguous output behavior.
- We use an interpretation of I/O programs as black-boxes that produce (finite) traces when supplied with a sequence of inputs. This makes our formalism in principle independent of any particular programming language.
- In order to give a general framework for testing program behavior, we generalize the standard notion of a trace, covering a single program run, to representing the complete set of accepted behavior for a given sequence of inputs. We provide a set of rules to automatically derive random elements of such traces for a given specification (Section 6).
- We implement the presented approach as an embedded domain-specific language (EDSL) in Haskell (Section 6.4).

For the time being we restrict ourselves to programs with the two I/O primitives `readLn` and `print` and consider only programs that operate on integers. It is however relatively straightforward to generalize the approach to, for example, include string values.

2 Overview

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

"Add up all the numbers in a given list."

A simple Haskell solution could look like this:

$$\begin{aligned} &sum :: [Int] \to Int \\ &sum\ [\,] \quad\ = 0 \\ &sum\ (x : xs) = x + sum\ xs \end{aligned}$$

To test this, we could define QuickCheck properties like the following:

$$\begin{aligned} &propEmpty :: [Int] \to Bool \\ &propEmpty\ xs = null\ xs \Rightarrow sum\ xs == 0 \end{aligned} \qquad \begin{aligned} &propAdd :: Int \to [Int] \to Bool \\ &propAdd\ x\ xs = sum\ (x : xs) == sum\ xs + x \end{aligned}$$

By looking at these properties it is easy to see what is the intended behavior of the sum function.

Now consider a variation of the above description that might appear as an exercise task in a course section introducing I/O programs[1]:

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

[1] In fact this task or a variation of it occurs in almost every introductory Haskell textbook.

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 structure considerably.

```
main = do
  n ← readLn
  go n 0
  where
    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 can we even describe more formally than in the verbal description above what the desired behavior should be?

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 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 inputs to outputs. Instead, programs will result in a sequence of potentially interleaved input and output actions when provided with a sequence of inputs. We call this the trace of a program. If we want to check whether or not some program satisfies a certain behavior, we have to check if all of the possible traces a program can produce do match this behavior.

For the above program description, the set of intended traces is basically

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

where each ? stands for an input action and each ! for an output action. If we now assume that we never supply negative numbers as inputs (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 in [4], corresponding tests can be automated. First, an alternative monad is defined that represents teletype programs:

```
data IO_tt a
  = GetLine (String → IO_tt a)
  | PutLine String (IO_tt a)
  | Return a

instance Monad IO_tt 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_tt a
readLn = fmap read (GetLine Return)

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

Now, the 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_{tt} (). 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:

$$
\begin{array}{ll}
run_{tt} :: IO_{tt}\ () \to [String] \to Trace & \textbf{data}\ Trace \\
run_{tt}\ (GetLine\ f)\quad (x : xs) = Read\ x\ (run_{tt}\ (f\ x)\ xs) & \quad = Read\ String\ Trace \\
run_{tt}\ (PutLine\ s\ ma)\ xs \quad\ = Write\ s\ (run_{tt}\ ma\ xs) & \quad \mid Write\ String\ Trace \\
run_{tt}\ (Return\ ())\quad [\,] \quad\ = Stop & \quad \mid Stop
\end{array}
$$

If we now assume existence of a predicate $checkCorrectness :: Trace \to Bool$ and of a generator $validInputs :: Gen\ [String]$, we can use QuickCheck again to automatically test whether a program fulfills the intended behavior. But writing such a predicate and generator is generally not as straightforward as one would hope.

Our specification language and surrounding tooling solves exactly this problem. It allows us to state precisely what behavior we intend a program to have, and provides a way to automatically generate the behavior dependent components in the presented approach.

3 Specifications

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`."

To motivate the design of our small DSL, let us go through this description step by step and see what constructs are necessary to describe the behavior formally.

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

$$[\triangleright \mathbf{?}]\ \mathbf{???}\ [\mathbf{?} \triangleright]$$

We use (silent) concatenation to glue several smaller 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 describe computations over the values referenced by those variables:

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

Right now it is not clear what the argument to *sum* should be, but we will fill this 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 the program runs) how many values we need to read overall. Therefore we need some mechanism for variable iteration (rather than just some fixed times concatenation). We mark the part of a specification we would like to iterate with $\to^{\mathbf{E}}$ and introduce a marker $\mathbf{E}$ to indicate where/when the iteration process should end:

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

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

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

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

We can use this to repeatedly read in a value:

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

But now we have a problem, or actually two. Old values we "assigned" to x are lost, and we have no way of knowing when to stop. The key feature of our DSL that solves this is the fact that variables do not just store a current value like in most conventional 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 predicate, but now we can also fill in the missing argument to the summation:

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

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 is often useful when we do not care about ill formed or otherwise undesirable inputs, especially in an educative setting where we usually introduce new concepts one step at a time. In the beginning of a course we usually do not want students to, for example, have to worry about correctly parsing inputs. 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 can 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's):

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

One thing to note here is that the above specification is very rigid. There is no flexibility with regard to the interaction allowed. But one might want a specification to capture the fact that a program can have some extra behavior that does not really influence the core functionality. For example, we could modify the previous behavior description as follows.

"Read a natural number n from `stdin`*, then read n additional numbers and print the sum of those n numbers to* `stdout`*.* ***The program might print in between how many more summands it is expecting.***"

We encode such optional behavior directly inside the primitive used for output. That is, instead of giving a single term 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:

$$[\triangleright n]^{\mathbb{N}}\big(([\{\varepsilon, n_C - len(x_A)\}\triangleright][\triangleright x]^{\mathbb{Z}})\bigtriangleup_{len(x_A)=n_C}\mathbf{E}\big)^{\rightarrow^{\mathbf{E}}}[\{sum(x_A)\}\triangleright]$$

At first glance it might seem overly complicated or restrictive that we introduce this kind of non-determinism only in outputs and not, for example, as 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, we can choose a different task to fulfill in each iteration. Abstractly this is not a problem, but since we do not consider programs students write to be capable of true non-deterministic choice, we cannot write a program that actually does this. Therefore such a choice operator introduces not only the form of optionality we want to encode, but also enables a form of meta statement that we think should not be part of the language.

Note that this does not mean that specifications cannot require completely different behavior depending on some input. For example, we can clearly write specifications of the form $[\triangleright x]^{\mathbb{Z}}(s_1 \bigtriangleup_{p(x)} s_2)$. But since $p(x)$ is deterministically defined once x is known, there is no non-determinism involved here. We model optionality in this context by introducing an empty specification **0** that we can use to under a certain condition skip over parts of a specification.

3.2 Testing behavior

Consider now the following trace: ?2 ?5 ?3 !8 *stop*. Does this trace match the specification developed above? We can check this by going from left to right (and possibly in loops) through the specification and seeing if the trace actions match each required action, while keeping track of the contents of variables.

Starting with ?2, we compare this 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 of the body without encountering an **E**, we just check the remaining trace against the iteration body again. When we do encounter an end marker, we continue by checking the remaining trace against the specification following the whole iteration.

In this case this means we start by checking ?5 ?3 !8 *stop* against $([\{\varepsilon, n_C - len(x_A)\}\triangleright][\triangleright x]^{\mathbb{Z}}) \bigtriangleup_{len(x_A)=n_C} \mathbf{E}$. We first evaluate the branching predicate 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 $[\{\varepsilon, n_C - len(x_A)\}\triangleright]$. 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 is $[\triangleright x]^{\mathbb{Z}}$ which matches ?5 and results in 5 being added to x (actually, to be assigned to x_C and added to x_A). Since we have no specification left to check, but are inside an iteration, we again check against the 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 iteration the branching predicate evaluates to true, thus ending the loop due to the occurrence of **E** found to the right of the branching construct.

All that is left now is to check ?8 *stop* against $[\{sum(x_A)\}\triangleright]$. Since we have $sum(x_A) = sum([5,3]) = 8$, 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 *stop* is a valid program run for the specified behavior.

An important feature of our language is that we can also reverse this procedure, then looking for a (random) trace that will match the specification (instead of starting from a given trace). For the specification used above, this could for example yield the trace form ?3 !$\{\varepsilon, 3\}$?-1 !$\{\varepsilon, 2\}$?7 !$\{\varepsilon, 1\}$?4 !$\{10\}$ *stop*, where the values of inputs are random elements of the expected type and the output values are the results

of evaluating all output possibilities. 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 *stop*, we see that this is not a valid trace since the first output does not actually allow for the printing of 4. If we do this for enough random traces derived from the specification, we either find a counterexample, showing that behavior is not satisfied, or gain reasonable confidence in the correctness of a program.

4 Syntax

Figure 1 gives the full syntax of our language by defining the set *Spec* of all specifications. Similarly, Figure 2 defines the term language used for the description of outputs and branching predicates. We distinguish different subsets of 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 $T_{\mathbb{B}}$ is the set of terms evaluating to a Boolean value. We write $[\mathbb{Z}]$ instead of $\mathbb{Z}^*$ for sequences of integers when we want to emphasize that we are dealing with list values as opposed to words over integers. With the exception of (Write) the

$$\frac{\tau \in Ty \quad x \in Var}{[\triangleright x]^{\tau} \in Spec}\ (\text{Read}) \qquad Ty = \{\mathbb{N}, \mathbb{Z}, \mathbb{Z}_+, \mathbb{Z}_-\}$$

$$\frac{\Theta \subseteq (T_{\mathbb{Z}} \cup \{\varepsilon\}), \Theta \cap T_{\mathbb{Z}} \neq \emptyset}{[\Theta \triangleright] \in Spec}\ (\text{Write}) \qquad \frac{s_1 \in Spec \quad s_2 \in Spec}{s_1 \cdot s_2 \in Spec}\ (\text{Seq})$$

$$\frac{s_1 \in Spec \quad s_2 \in Spec \quad p \in T_{\mathbb{B}}}{s_1 \bigtriangleup_{p} s_2 \in Spec}\ (\text{Branch}) \qquad \frac{s \in Spec}{s^{\rightarrow \mathbf{E}} \in Spec}\ (\text{Till-}\mathbf{E})$$

$$\frac{}{\mathbf{0} \in Spec}\ (\text{Nop}) \qquad \frac{}{\mathbf{E} \in Spec}\ (\text{LoopEnd})$$

Figure 1: Language syntax

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

$$\frac{x \in Var}{x_C \in T_{\mathbb{Z}}}\ (\text{Current}) \qquad \frac{x \in Var}{x_A \in T_{[\mathbb{Z}]}}\ (\text{All})$$

$$\frac{f : D_1 \times \ldots \times D_n \to D \quad t_1 \in T_{D_1}, \ldots, t_n \in T_{D_n} \quad f \in Func}{f(t_1, \ldots, t_n) \in T_D}\ (\text{Function})$$

Figure 2: Syntax of terms

For terms we restrict ourselves to expressions involving 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 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 we might be interested in automatically generating random specifications for exercise tasks. In that case we can control to some extent what kind of tasks are generated by choosing different sets for *Func*.

We make the following assumptions regarding specifications, some structurally, some more about semantic well-formedness:

- 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 inputs). 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 end marker might be reachable but we do not analyze the branching conditions.
- No (sub-)specification of the form $\mathbf{E} \cdot s$ exists, since everything occurring after an $\mathbf{E}$ is "dead code" anyway. This primarily simplifies the definition of our semantics in the next section by avoiding additional rules for rewriting such cases.

Additionally, sequential composition of specifications is defined to be associative, i.e., $s_1 \cdot (s_2 \cdot s_3) \equiv (s_1 \cdot s_2) \cdot s_3$, therefore we can just write $s_1 \cdot s_2 \cdot s_3$ instead, or indeed $s_1\, s_2\, s_3$. Furthermore we define sequential composition to have higher precedence than branching, and $\rightarrow^{\mathbf{E}}$ to have higher precedence than sequential composition, i.e., $s_1 \cdot s_2 \bigtriangleup_p s_3 \equiv (s_1 \cdot s_2) \bigtriangleup_p s_3$ and $s_1 \cdot s_2^{\rightarrow^{\mathbf{E}}} \equiv s_1 \cdot (s_2^{\rightarrow^{\mathbf{E}}})$.

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

We capture the behavior of a program as a set of traces. For each specification we therefore define a set containing all program runs that fulfill the specified behavior. A program then fulfills a specification if the set of all its possible traces is a subset of the set defined by the specification.

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 an execution path. We use *Tr* to denote the set of all traces.

5.1 Matching Traces and Specifications

We define a relation $\Delta \vdash t \models s$ where $t \in \mathit{Tr}$ is a trace, $s \in \mathit{Spec}$ is a specification, and $\Delta : \mathit{Var} \rightarrow [\mathbb{Z}]$ is a variable assignment mapping variable names to list values. The statement $\Delta \vdash t \models s$ is interpreted as: The trace t matches the behavior described by s under the variable assignment Δ.

Figure 3 gives the rules defining the relation. The functions *eval* and *update* evaluate terms to values and update the environment (both C and A versions of variables), respectively. Their definitions are straightforward and are therefore omitted. We write $\mathit{eval}(\Delta, \Theta)$ for evaluating every term in Θ.

The core matching mechanism is given by the first four rules. We match traces against specifications by going through the specifications from left to right, consuming matching trace elements and updating variables along the way. If we arrive at the empty specification and consumed the whole trace, then we know that this particular trace matches the specified behavior. In the other case we will get stuck at some

$$\frac{\mathit{update}(\Delta, x, v) \vdash t' \models s' \quad v \in \tau}{\Delta \vdash ?vt' \models [\triangleright x]^{\tau} \cdot s'} \text{ (Match-Read)}$$

$$\frac{\Delta \vdash t' \models s' \quad v \in \mathit{eval}(\Delta, \Theta \setminus \{\varepsilon\})}{\Delta \vdash !vt' \models [\Theta\triangleright] \cdot s'} \text{ (Match-Write)} \qquad \frac{\Delta \vdash t \models s \quad \varepsilon \in \Theta}{\Delta \vdash t \models [\Theta\triangleright] \cdot s} \text{ (Skip-Write)}$$

$$\frac{}{\Delta \vdash \mathit{stop} \models \mathbf{0}} \text{ (Stop)}$$

$$\frac{\Delta \vdash t \models s_{11} \cdot s_2 \quad \mathit{eval}(\Delta, p) = \texttt{False}}{\Delta \vdash t \models (s_{11} \bigtriangleup_p s_{12}) \cdot s_2} \text{ (Left-Ch)} \qquad \frac{\Delta \vdash t \models s_{12} \cdot s_2 \quad \mathit{eval}(\Delta, p) = \texttt{True}}{\Delta \vdash t \models (s_{11} \bigtriangleup_p s_{12}) \cdot s_2} \text{ (Right-Ch)}$$

$$\frac{\Delta \vdash t \models s \cdot \{s, s'\} \quad \mathit{stop} \not\models s}{\Delta \vdash t \models s^{\rightarrow_{\mathbf{E}}} \cdot s'} \text{ (Enter)} \qquad \frac{\Delta \vdash t \models s \cdot \{s, s'\}}{\Delta \vdash t \models \{s, s'\}} \text{ (Again)} \qquad \frac{\Delta \vdash t \models s'}{\Delta \vdash t \models \mathbf{E} \cdot \{s, s'\}} \text{ (Exit)}$$

$$\frac{\Delta \vdash t \models s}{\Delta \vdash t \models \mathbf{0} \cdot s} \text{ (Elim-}\mathbf{0}\text{)} \qquad \frac{\Delta \vdash t \models s \cdot \mathbf{0} \quad s \neq s_1 \cdot s_2 \quad s \neq \{s_1, s_2\} \quad s \neq \mathbf{0}}{\Delta \vdash t \models s} \text{ (Extend-Atom)}$$

Figure 3: Trace matching

point in the derivation where the trace and the specification do not agree on an action to take and the action required by the specification can also not be skipped via (Skip-Write).

The rest of the rules apply the same left to right traversal to the other syntactic constructs. To handle iteration without the need for extra context information, (Enter) introduces a new syntactic form $\{s, s'\}$ called a jump-point. It encodes the two options for continuing after matching against the body of an iteration. If we reduce a specification to just a jump-point, it means we have not encountered the end of that iteration and therefore match again against the first component of that jump-point. But if we hit an $\mathbf{E}$, we exit the loop and continue with the second component of the jump-point. Note that our assumptions about well-formed specifications from the previous section guarantee that an $\mathbf{E}$ always occurs directly in front of a jump-point once we reach it. Additionally, in the case of nested iterations there is never more than one top-level jump-point, since an outer jump-point moves into the second component of the inner jump-point once we enter the inner iteration.

In order to not get caught in an infinite derivation, (Enter) requires that the loop body does not match an empty trace that does not take any actions. It is however not immediately clear how to check this. An alternative formulation of this productivity requirement can be expressed by the following two modified rules augmenting jump-points with an indication of the length of the trace on the last unrolling of the loop body.

$$\frac{\Delta \vdash t \models s \cdot \{s, s'\}^{|t|}}{\Delta \vdash t \models s^{\rightarrow_{\mathbf{E}}} \cdot s'} \text{ (Enter)} \qquad \frac{\Delta \vdash t \models s \cdot \{s, s'\}^{|t|} \qquad |t| < n}{\Delta \vdash t \models \{s, s'\}^{n}} \text{ (Again)}$$

This uses the observation that the empty trace can only match specifications that do not specify any interactions. After matching against such a specification, the remaining trace has to be the same as before. Therefore, recording the current length of the trace and making sure it has decreased before each further iteration guarantees we do not fall into an infinite derivation sequence.

Note that the left to right traversal of the specification is essential in order to update the environment in a consistent way. Otherwise the result of evaluating terms in the (*-Write) rules could depend on the overall order of rule applications. The rules are therefore formulated in a way so that they cannot be applied on sub-specifications but only the current specification as a whole. This especially manifests in the premises of the (Extend-Atom) rule that ensures we actually end up with the **0**-specification in the end by adding **0** to a specification that would otherwise not match any rule.

5.2 Program Correctness

Using the matching relation, we can formulate a notion of program correctness. A program is considered an implementation of a specification s if and only if for every trace t that can be produced by the program we have $\Delta_I \vdash t \models s$, where Δ_I is the initial environment mapping every variable occurring in s to the empty list.

6 Testing framework

Now that we defined syntax and semantics of our DSL, how can we actually test programs against specifications? First we need a way to obtain a test case from a specification. The obvious idea is to generate some trace from a specification and to check whether the program results in the same trace when provided with the same inputs. However, things are not that simple. Consider the specification $[\triangleright x][\{\varepsilon, x_C\}\triangleright][\{\varepsilon, x_C\}\triangleright]$. For arbitrary input $v \in \mathbb{Z}$, the valid traces from this specification are $?v\,stop$, $?v\,!v\,stop$, and $?v\,!v\,!v\,stop$. Therefore, the test case for v should check for any of those traces. As a consequence we cannot use a single simple trace as the description of such a test case. We instead use a generalized notion of traces to encode all combinations of possible behavior, for given inputs, into a single trace. Another issue is how to actually come up with valid input sequences, fit for a given specification, in the first place. That, again, will be dealt with based on the generation of generalized traces.

6.1 Generalized traces

In the same way we allowed different output possibilities in the specification language, we now allow different possibilities in each output part of a trace. Moreover, we combine sequences of adjacent output actions into a single action containing a sequence of values. This normalization simplifies the comparison of traces (against each other or against a specification), since we do not have to use any look-ahead or backtracking. The single generalized trace that combines the traces from above is $?v\,!\{\varepsilon, v, vv\}\,stop$. This trace now completely covers the possible behavior of a correct program for the given input. The set of all such *normalized* traces Tr_N is defined by the following rules:

$$\frac{v \in \mathbb{Z} \quad t' \in Tr_N}{?v t' \in Tr_N} \qquad \frac{v \in \mathbb{Z} \quad t' \in Tr_N \quad V \subseteq \mathbb{Z}^* \quad V \setminus \{\varepsilon\} \neq \emptyset}{!V\,?v t' \in Tr_N}$$

$$\frac{}{stop \in Tr_N} \qquad \frac{V \subseteq \mathbb{Z}^* \quad V \setminus \{\varepsilon\} \neq \emptyset}{!V\,stop \in Tr_N}$$

Every ordinary trace can easily be converted into such a normalized form by way of the following embedding function $t_N : Tr \to Tr_N$.

$$\begin{aligned}
t_N(t) &= t_N(t, \varepsilon) \\
t_N(!v\, t', w) &= t_N(t', wv) \\
t_N(?v\, t', w) &= \begin{cases} ?v\, t_N(t', \varepsilon) & \text{, if } w = \varepsilon \\ !\{w\}?v\, t_N(t', \varepsilon) & \text{, otherwise} \end{cases} \\
t_N(stop, w) &= \begin{cases} stop & \text{, if } w = \varepsilon \\ !\{w\}\ stop & \text{, otherwise} \end{cases}
\end{aligned}$$

For normalized traces we can determine whether one is *covering* the other, that is if the set of different program runs represented by one trace is a subset of the runs represented by the other. This is encoded in the relation $\sqsubseteq \subseteq Tr_N \times Tr_N$. The relation is defined by the following rules:

$$\frac{t_1' \sqsubseteq t_2'}{?v\, t_1' \sqsubseteq ?v\, t_2'} \qquad \frac{t_1' \sqsubseteq t_2' \quad V_1 \subseteq V_2}{!V_1\, t_1' \sqsubseteq !V_2\, t_2'} \qquad \frac{t_1' \sqsubseteq t_2' \quad \varepsilon \in V}{?v\, t_1' \sqsubseteq !V\ ?v\, t_2'} \qquad \frac{\varepsilon \in V}{stop \sqsubseteq !V\ stop} \qquad \frac{}{stop \sqsubseteq stop}$$

Moreover, two normalized traces belong to the same test case if they have the same sequence of inputs, i.e., the traces become identical if we remove all of their output actions. We call such traces *similar*, denoted $t_1 \simeq t_2$. Since the sequence of inputs is not altered during the normalization of an ordinary trace, this notion of similarity also extends to $Tr \times Tr$.

Covering and similarity are closely related. On the one hand we have $t_1 \sqsubseteq t_2 \Rightarrow t_1 \simeq t_2$, on the other hand it follows from the definitions that a least upper bound with regard to $\sqsubseteq$ exists for every set of similar traces. We can use this to define the *generalized* trace for a set of similar non-normalized traces $T \subseteq Tr$ as

$$t_g(T) = \bigsqcup \{t_N(t) \mid t \in T\}$$

That is we first normalize every trace in T and then find a trace that covers every program run represented by the different traces. Intuitively a generalized trace can be thought of as an (incomplete) ad-hoc specification for a single test case derived from some examples.

Notice however that we have

$$t_g(\{?1?1stop, ?1!1?1!1stop\}) = ?1!\{\varepsilon, 1\}?1!\{\varepsilon, 1\}stop$$

So a possible distinction between "never make an output after an input" and "always make an output after an input" is blurred, into "optionally make outputs after inputs, deciding anew each time". This is similar to our design choice against full blown non-determinism. In the same way we do not want to encode meta statements like "match either specification A or specification B" we also do not want to encode dependencies between output choices. In a specification (part) like $[\triangleright x]^\tau[\{\varepsilon, x_C\}\triangleright]$ the choice made for output is completely non-deterministic, even over several invocations if that part would occur inside a loop. If one wanted to specify that each iteration in such a loop has the same behavior (make the output or do not make it), as would be the case for a typical program, one would have to write two separate specifications, one with $[\triangleright x]^\tau$ in the loop body and one with $[\triangleright x]^\tau[\{x_C\}\triangleright]$.

6.2 Testing

With these tools we can now define the set of generalized traces that represent all valid program runs with regard to some specification s as

$$Tr^G_\Delta(s) = \{t_g(T) \mid T \in (\{t \in Tr \mid \Delta \vdash t \models s\}/\simeq)\}$$

with $X/\simeq$ denoting the quotient set with regard to $\simeq$.
For $s = [\triangleright n]^{\mathbb{N}}([\{\varepsilon, n_C - len(x_A)\}\triangleright][\triangleright x]^{\mathbb{Z}} \bigwedge_{len(x_A)=n_C} \mathbf{E})^{\rightarrow_{\mathbf{E}}}[\{sum(x_A)\}\triangleright]$ and the initial variable environment Δ_I that maps every name to the empty list, we get the following set of generalized traces:

$$\begin{aligned}
Tr^G_{\Delta_I}(s) = \{&t_g(\{?0\,!0\,stop\}),\\
&t_g(\{?1\,?v_1\,!v_1\,stop,\ ?1\,!1\,?v_1\,!v_1\,stop\}),\\
&t_g(\{?2\,?v_1\,?v_2\,!(v_1+v_2)\,stop,\ ?2\,!2\,?v_1\,?v_2\,!(v_1+v_2)\,stop,\\
&\quad ?2\,?v_1\,!1\,?v_2\,!(v_1+v_2)\,stop,\ ?2\,!2\,?v_1\,!1\,?v_2\,!(v_1+v_2)\,stop\}),\\
&\cdots \mid v_1, v_2, \cdots \in \mathbb{Z}\}\\
= \{&?0\,!0\,stop,\\
&?1\,!\{\varepsilon, 1\}?v_1\,!v_1\,stop,\\
&?2\,!\{\varepsilon, 2\}\,?v_1\,!\{\varepsilon, 1\}\,?v_2\,!(v_1+v_2)\,stop,\\
&\cdots \mid v_1, v_2, \cdots \in \mathbb{Z}\}
\end{aligned}$$

We can give an alternative formulation of our notion of program correctness in terms of such a set. A trace $t \in Tr$ is valid for the specification s if and only if there exists a $t_g \in Tr^G_{\Delta_I}(s)$ such that $t_N(t) \sqsubseteq t_g$. Or to put it the other way around, if we choose an arbitrary $t_g \in Tr^G_{\Delta_I}(s)$ and run a program on the input sequence of that trace, the program results in a trace t that is covered by t_g if the program satisfies the specification. This is the core of our automatic testing framework, since we can derive rules for generating all, and exactly the, elements of $Tr^G_{\Delta_I}(s)$ from the matching rules in Figure 3. These rules are shown in Figure 4. In (Produce-Combined-Write) we use the language concatenation of V_1 and V_2 to combine consecutive outputs into single words. For notational simplicity we also assume that $eval(\Delta, \varepsilon) = \varepsilon$.

6.3 Feedback generation

Generalized traces provide a convenient way to generate feedback in case we found an input sequence for which a program does not have the desired behavior. A mismatch between the produced trace and the specification trace can either be a mismatch between actions taken or the program outputs something that is not in the set of valid outputs for that step. In both case a simple message like "*Expected ... but program produced ..., for input sequence: ...*" can be generated and presented to the user. 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 educative setting.

6.4 Implementation and Shortcomings

We have built an EDSL for our design in Haskell and implemented a naive version of the testing approach explained thus far. Within this framework, checking programs against our running example specification currently can be done like so, using an *Applicative* style for defining terms over variables:

$$\frac{v \in \tau \quad \Delta' = update(\Delta, v, x) \quad t' \in Tr^G_{\Delta'}(s')}{?v\,t' \in Tr^G_\Delta([\triangleright x]^\tau \cdot s')} \text{ (Produce-Read)}$$

$$\frac{V = V_1 \cdot V_2 \quad V_1 = eval(\Delta, \Theta) \quad !V_2\,t' \in Tr^G_\Delta(s')}{!V\,t' \in Tr^G_\Delta([\Theta\triangleright] \cdot s')} \text{ (Produce-Combined-Write)}$$

$$\frac{V = eval(\Delta, \Theta) \quad ?v\,t' \in Tr^G_\Delta(s')}{!V\,?v\,t' \in Tr^G_\Delta([\Theta\triangleright] \cdot s')} \text{ (Produce-Single-Write)}$$

$$\frac{V = eval(\Delta, \Theta)}{!V\,stop \in Tr^G_\Delta([\Theta\triangleright] \cdot \mathbf{0})} \text{ (Produce-Last-Write)} \quad \frac{}{stop \in Tr^G_\Delta(\mathbf{0})} \text{ (Stop)}$$

$$\frac{t \in Tr^G_\Delta(s_{11} \cdot s_2) \quad eval(\Delta, p) = \texttt{False}}{t \in Tr^G_\Delta((s_{11} \bigtriangleup_p s_{12}) \cdot s_2)} \text{ (Left-Ch)} \quad \frac{t \in Tr^G_\Delta(s_{12} \cdot s_2) \quad eval(\Delta, p) = \texttt{True}}{t \in Tr^G_\Delta((s_{11} \bigtriangleup_p s_{12}) \cdot s_2)} \text{ (Right-Ch)}$$

$$\frac{t \in Tr^G_\Delta(s \cdot \{s, s'\}) \quad stop \not\models s}{t \in Tr^G_\Delta(s^{\to \mathbf{E}} \cdot s')} \text{ (Enter)} \quad \frac{t \in Tr^G_\Delta(s \cdot \{s, s'\})}{t \in Tr^G_\Delta(\{s, s'\})} \text{ (Again)} \quad \frac{t \in Tr^G_\Delta(s')}{t \in Tr^G_\Delta(\mathbf{E} \cdot \{s, s'\})} \text{ (Exit)}$$

$$\frac{t \in Tr^G_\Delta(s)}{t \in Tr^G_\Delta(\mathbf{0} \cdot s)} \text{ (Elim-}\mathbf{0}\text{)} \quad \frac{t \in Tr^G_\Delta(s \cdot \mathbf{0}) \quad s \neq s_1 \cdot s_2 \quad s \neq \{s_1, s_2\} \quad s \neq \mathbf{0}}{t \in Tr^G_\Delta(s)} \text{ (Extend-Atom)}$$

Figure 4: trace set derivation rules

```
spec :: Specification
spec =
  readInput "n" NatTy <>
  tillE (
    branch
      ((λxs n → length xs == n) <$> getAll xs <*> getCurrent n)
      (optional (writeOutput [(λxs n → n − length xs) <$> getAll "xs" <*> getCurrent "n"])
        <> readInput "x" IntTy)
      e
  ) <>
  writeOutput [sum <$> getAll "xs"]
test :: IO_tt () → IO ()
test prog = quickCheck $ prog `fulfills` spec
```

While our implementation works well for small examples like the one shown here, there are some shortcomings for more complex specifications. A main problem, as with all forms of probabilistic testing, is the generation of test cases. At the moment we are not able to make any guarantees when it comes to coverage of branches or other measures of quality for test case distributions. Also, for specifications that use nested branching or very specific branching predicates, simply generating random input values is not sufficient to eventually fulfill the predicate at all. Consider for example the specification $([\triangleright x]^{\mathbb{Z}} \bigtriangleup_{sum(x_A)=c} \mathbf{E})^{\to \mathbf{E}}$ for some constant c. In order to reach the desired sum and end the loop, the next input has to be exactly

the difference between c and the current sum. Selecting this one element at random is very unlikely. So our implementation usually fails to finish generating even one valid trace for such specifications. This is certainly something we are planning to address in the future.

7 Restrictions on Expressiveness

As we already hinted at earlier, the expressiveness of the 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 explained in Section 2. On the one hand, such restrictions keep the syntax and semantics of the language simple and have the potential of enabling 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 an educative setting, teaching Haskell I/O. Specifically, we would like the specified pattern to enforce actual interactivity. That is, at its core it should rely on an (alternating) sequence of reads and writes and should not be expressible in a different way. Consider, for example, the following program:

$$
\begin{aligned}
&\mathit{main} :: \mathit{IO}\ () \\
&\mathit{main} = \mathbf{do} \\
&\quad n \leftarrow \mathit{readLn} \\
&\quad \mathit{loop}\ n \\
&\mathit{loop} :: \mathit{Int} \to \mathit{IO}\ () \\
&\mathit{loop}\ n \mid n \leqslant 0 = \mathit{return}\ () \\
&\mathit{loop}\ n \qquad\quad = \mathit{print}\ n \gg \mathit{loop}\ (n-1)
\end{aligned}
$$

A specification for such a program is not expressible in the presented DSL, and that was a design goal. The non-expressibility is due to the facts that an iteration can only end based on some predicate over the global variable state, and that only inputs can alter this state, leaving the above kind of "output driven loops" impossible to encode. According to our motivation, this is a good thing. We only want inherently interactive behavior to be expressible, while the above program can be reformulated like so:

$$
\begin{aligned}
&\mathit{main} :: \mathit{IO}\ () \\
&\mathit{main} = \mathbf{do} \\
&\quad n \leftarrow \mathit{readLn} \\
&\quad \mathit{print}\ \$\ \mathit{loop}\ n \\
&\mathit{loop} :: \mathit{Int} \to \mathit{String} \\
&\mathit{loop}\ n \mid n \leqslant 0 = \texttt{""} \\
&\mathit{loop}\ n = \mathit{show}\ n +\!\!+ \texttt{"\textbackslash n"} +\!\!+ \mathit{loop}\ (n-1)
\end{aligned}
$$

which is not an attractive teaching aim when we actually want to handle interactive I/O in Haskell.

If we for one moment lift our restriction to just use integers for input and output, we could write a specification like $[\triangleright n]^{\mathbb{Z}}[\{\mathit{loop}(n)\}\triangleright]$ for this behavior, with the second version of *loop* above. From this it is immediately clear that the interactive core of the original program/task is almost trivial. Looking at this from the other side, 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.[2]

[2]Note that, even if we did indeed allow strings for input and output, as we will eventually have to do for practical usage in our course, we could still prevent creation of such "boring tasks" via the DSL, by controlling which functions are allowed inside *Func* from Figure 2.

8 Related Work

The general mechanism for building inspectable representations of side-effecting programs is provided in the Haskell IOSpec library[3]. It supports not only teletype I/O but also forking processes, mutable references and software transactional memory. However, its API is very minimal and no other higher abstractions currently exist.

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 assessment tools for programming tasks is presented by Keuning et al. [2]. Most tools use some form of automatic testing, either on specified test cases or by comparing submissions to the results of model 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 allow for 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

The general testing framework does not formulate any special search strategy for finding test cases. This is far from ideal. As already mentioned above, we can easily 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 is given. Even though the testing framework can check programs against the kind of small and relatively simple specification that we would use in an educative setting, general applicability 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. But why stop there? We already have a list of areas in mind for which we would like to develop some automatization, and our DSL is designed partially with these possibilities in mind. We are therefore confident that we can extend the automatization beyond simple probabilistic testing.

What follows is a short list of some items we would like to implement in the future:

- **Task generation** With the presented formal description of behavior, especially the syntax, we can automatically generate specifications as the basis for new tasks. What is missing is a way to control the complexity of such specifications in a way that lets us produce tasks of a certain kind and difficulty. The language already provides ways to control expected inputs and some guidance is possible by choosing an appropriate term language (restricting the set *Func*). A general framework for defining and describing such meta properties together with an e-learning system could then be used, for example, to provide every student with an individual or even unique task while guaranteeing fairness in terms of difficulty etc.

[3] `https://hackage.haskell.org/package/IOSpec`

- **Sample solutions** It should be fairly obvious that for every specification in our language we can automatically generate a (Haskell) program with the respective behavior. However, such a program is most likely not an ideal candidate for a sample solution to show to students. Some 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 hints to the root of the problem but just a basic counterexample for which the program behaves in a wrong way somehow. One potential way to improve upon this would be to try to infer a specification for the wrongly behaving program and then compare that specification to the original one. This way it might be possible to identify sources of common mistakes and provide hints on how a student might solve those.
- **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. One could, for example, add further I/O capabilities to the set of basic actions, like reading and writing files. But also completely different domains are possible, like transformations of list or other data structures that use a certain set of predefined combinators. Adapting the approach to such pure contexts would allow us to also automatically generate and check tasks of that kind, something that might not be necessarily possible otherwise.

10 Conclusion

We presented a formal language for describing the behavior of an interactive program with regard to the teletype primitives *readLn* and *print*. By doing so we gain the ability to automatically generate tests and test cases to probabilistically check whether a program satisfies some behavior. We believe that this is not only an improvement for our indented use in automatically grading exercises, but that the language could also be useful for testing interactive behavior in more general domains. 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 Martin Odersky & Philip Wadler, editors: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, ACM, pp. 268–279, doi:10.1145/351240.351266. Available at `https://doi.org/10.1145/351240.351266`.

[2] Hieke Keuning, Johan Jeuring & Bastiaan Heeren (2019): *A Systematic Literature Review of Automated Feedback Generation for Programming Exercises*. *TOCE* 19(1), pp. 3:1–3:43, doi:10.1145/3231711. Available at `https://doi.org/10.1145/3231711`.

[3] Mirko Rahn, Alf Richter & Johannes Waldmann (2008): *The Leipzig autotool E-Learning/E-Testing System*. In: *Englisch und Deutsch. In: Symposium on Math Tutoring, Tools and Feedback. Open Universiteit Nederland, S*, pp. 1–41. Available at `http://www.imn.htwk-leipzig.de/~waldmann/talk/08/ou08/tool.pdf`.

[4] Wouter Swierstra & Thorsten Altenkirch (2007): *Beauty in the beast*. In Gabriele Keller, editor: *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, ACM, pp. 25–36, doi:10.1145/1291201.1291206. Available at `https://doi.org/10.1145/1291201.1291206`.