# Implementing, and Keeping in Check, a DSL Used in E-Learning

Oliver Westphal(✉) and Janis Voigtländer

University of Duisburg-Essen, Duisburg, Germany
{oliver.westphal,janis.voigtlaender}@uni-due.de

**Abstract.** We discuss a DSL intended for use in an education setting when teaching the writing of interactive Haskell programs to students. The DSL was previously presented as a small formal language of specifications capturing the behavior of simple console I/O programs, along with a trace-based semantics. A prototypical implementation also exists. When going for productive application in an actual course setting, some robustness and usability questions arise. For example, if programs written by students are mechanically checked and graded by the implementation, what guarantees are there for the educator that the assessment is correct? Does the implementation really agree with the on-paper semantics? What else can inform the educator's writing of a DSL expression when developing a new exercise task? Which activities beyond testing of student submissions can be mechanized based on the specification language? Can we, for example, generate additional material to hand to students in support of task understanding, before, and feedback or trusted sample solutions, after their own solution attempts? Also, how to keep the framework maintainable, preserving its guarantees when the expressiveness of the underlying DSL is to be extended? Our aim here is to address these and related questions, by reporting on connections we have made and concrete steps we have taken, as well as the bigger picture.

## 1 Introduction

We previously presented a small formal specification language for describing interactive behavior of console I/O programs [12]. Given such specifications, we can check whether some candidate program has the specified behavior by repeatedly running the program and matching its traces against the specification, thus either finding a counterexample or gaining sufficient confidence that the program behaves as desired. We plan to use this approach of specification and testing to automatically grade student submissions on the subject of writing interactive programs [8] in our Haskell programming course, and therefore we are developing an implementation. The goal of the implementation is to not only get a working version of the on-paper definitions but a DSL-based framework that makes it easy to design and adapt exercise tasks for use in an e-learning system [9,11].


K. Nakano and K. Sagonas (Eds.): FLOPS 2020, LNCS 12073, pp. 179–197, 2020.
https://doi.org/10.1007/978-3-030-59025-3_11

Central to this goal is a guarantee that if something different is happening than an educator is expecting, then that is not the fault of the DSL implementation itself and instead is therefore fixable by said educator. Moreover, the educator should not be left alone with their mismatch in expectations. The framework's implementation should provide some means for them to investigate what is actually going on and where they are possibly missing a connection. This article is about how we can provide such means.

## 2 Specification Language Overview

Our specification and testing framework consists of four major components: a way to express specifications that describe read/write behavior of programs, a notion of traces for capturing program runs, a function to determine whether a trace represents program behavior that is valid regarding a specification, and a testing procedure to summarily check programs against specifications.

As an example, take the following program that reads in a natural number and then that many additional numbers and finally prints their sum:

$$
\begin{aligned}
&\mathit{main} :: \mathsf{IO}\ () \\
&\mathit{main} = \mathbf{do}\ n \leftarrow \mathit{readLn} \\
&\qquad \mathbf{let}\ \mathit{loop}\ \mathit{xs} = \mathbf{if}\ \mathit{length}\ \mathit{xs} == n\ \mathbf{then}\ \mathit{print}\ (\mathit{sum}\ \mathit{xs}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else\ do}\ x \leftarrow \mathit{readLn} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{loop}\ (x : \mathit{xs}) \\
&\qquad \mathit{loop}\ [\,]
\end{aligned}
$$

A specification for the behavior of this program looks as follows:

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

Testing against the specification is done by randomly generating suitable input sequences (note that for this specification the first input to a program should always be non-negative) and then comparing each trace resulting from running the program on such input with the expectation encoded in the specification. The trace of a program is the sequence of read and written values. For example, given the inputs $2, 7, 13$, the trace of the above program would be $?2\,?7\,?13\,!20\,\mathsf{stop}$. We now give a brief overview of specifications, traces, and what/how we test.

### 2.1 Specifications

There are three atomic forms of specifications: $\mathbf{0}$ is the empty specification, $[\triangleright\, x\,]^{\tau}$ is for reading a value typed by $\tau \subseteq \mathbb{Z}$ into a variable $x$, and $[\,\Theta \triangleright\,]$ is for outputting the result of evaluating any term $t \in \Theta$, where $\Theta$ represents a set of possible outputs. Variables are always associated with the lists of all values previously read into them. Accessing variables, in terms, can then be done in two different ways: either as $x_A$, giving precisely the list of all read values for $x$, or as $x_C$, giving only the last, most current value.

Two specifications $s_1$ and $s_2$ can be composed sequentially, denoted by $s_1 \cdot s_2$. Sequential composition is defined to be associative and to have $\mathbf{0}$ as the neutral element. It will often be performed silently by writing just $s_1 s_2$ instead of $s_1 \cdot s_2$. By $s_1 \angle c \rangle s_2$ we denote the specification that either requires $s_1$ or requires $s_2$ to be adhered to, depending on which Boolean value the term $c$ evaluates to under the current variable assignment, where True chooses the branch on the right. Lastly, $s^{\rightarrow \mathbf{E}}$ stands for the repetition of specification $s$ until, inside $s$, the iteration exit-marker $\mathbf{E}$ is encountered, which behaves similarly to the `break` command found in many imperative languages (under various names).

### 2.2 Traces

Traces are sequences $m_0 v_0\, m_1 v_1\, \ldots\, m_n v_n\, \mathsf{stop}$, where $n \in \mathbb{N}$, $m_i \in \{?, !\}$, and $v_i \in \mathbb{Z}$. Here $?v$ denotes the reading of the value $v$ and $!v$ denotes the writing of the value $v$. Besides these ordinary traces there is also a notion of generalized traces that capture the complete behavior mandated by a specification for a single concrete input sequence. Basically, generalized traces are traces in which each output place is a *set* of all possible outputs a program can make at that point, potentially including the empty output $\varepsilon$ to indicate optionality.[1] A covering relation $\prec$ relates ordinary traces and generalized ones. If an ordinary trace $t$ is covered by a generalized trace $t_g$, denoted by $t \prec t_g$, it means that one can replace each set of outputs in $t_g$ by an element from that set and end up with $t$. For example, it holds that $?2\,!3\,!8\,\mathsf{stop} \prec ?2\,!\{3,6\}\,!\{\varepsilon, 7\}\,!\{8\}\,\mathsf{stop}$.

### 2.3 Acceptance Criterion

The conditions under which a program run, encoded by a trace, is considered to represent valid behavior regarding a specification are defined by the *accept*-function in Fig. 1 (coming straight from [12]): exactly if $accept(s, k_I)(t, \Delta_I)$ evaluates to True does the ordinary trace $t$ represent behavior valid for specification $s$. Here $\Delta_I$ is the variable assignment mapping each variable occurring in $s$ to the empty list and a continuation argument $k$ is used to keep track of the current iteration context. When entering an iteration, we build a new continuation that either repeats the loop body or restores the previous iteration context, depending on whether it is called with End or Exit. Calls with Exit or End, respectively, happen if we encounter an exit-marker $\mathbf{E}$ or hit the end of a specification, i.e., in the case $accept(\mathbf{0}, k)(t, \Delta)$. The initial continuation $k_I$ takes care of the handling at the top-level of the specification. Hence, $k_I$ is only intended to be called with End at the very end of traversing the specification overall. It then tests whether the remaining trace equals stop, which indicates acceptance of the initially given trace; or else the result of *accept* is False.

[1] In the full formulation from [12], consecutive outputs in generalized traces are additionally normalized into a single output action that chooses from a set of value *sequences*. We ignore this detail here in favor of a more straightforward presentation.

### 2.4 Testing

For the testing of programs against specifications, the *accept*-function is not used directly but is instead modified into a function *traceSet* that takes a specification and describes the set of all generalized traces valid for that specification. Intuitively, *traceSet* is obtained from "solving" $accept(s, k_I)(t, \Delta_I)$ for $t$.

In almost all cases the set of valid generalized traces for a given specification is infinite. There are two different ways the set of traces can become infinite, one harmless but the other one not so much. The first way is that we can arbitrarily choose a value from a potentially infinite set at every input step. This form of infinity is not really problematic, though, since we can work around it by sampling traces at random instead of computing all possibilities. The second way in which a *traceSet*-result can grow infinitely large is when we consider a specification that exhibits potentially non-terminating behavior. In this case sampling does not help us, since we can get stuck in an endless loop. But as long as we do not choose input values such that the behavior described by a specification becomes non-terminating, we can compute results of *traceSet*.[2] We will treat the *traceSet*-function as a black box here, since its technical details are not important here. Its correctness in the implementation, of course, is! (See Sect. 4.)

The actual testing, for some program $p$ and specification $s$, is done by repeatedly applying the following steps, resulting in either a counterexample or increased confidence in the appropriateness of $p$.

1. Use the *traceSet*-function to (randomly) sample a generalized trace for $s$.
2. From this trace, extract the sequence of inputs.
3. Determine whether the ordinary trace resulting from running $p$ on these inputs is covered by said generalized trace.

## 3 Comparing Theory and Implementation/Use

Our overall framework is hosted at https://github.com/fmidue/IOTasks. Besides the source code of the implementation, that repository also contains various usage examples. Here, let us consider an example task and compare its formulations in the on-paper version and in the implemented DSL. We take the same example as earlier: reading in a natural number and then as many further integers as that first number says, and finally printing those integers' sum. Recall that the specification $[\triangleright n]^{\mathbb{N}}([\triangleright x]^{\mathbb{Z}} \measuredangle len(x_A) = n_C \searrow \mathbf{E})^{\rightarrow \mathbf{E}}[\{sum(x_A)\} \triangleright]$ encodes this behavior. Transliterating it into our DSL (which is an EDSL using deep embedding [3]), we get the following Haskell expression:

[2] That is easier said than done. We do at the moment not have a general solution to reliably generate "suitable" inputs only, beyond simple typing as expressed by the $\tau$ in $[\triangleright x]^{\tau}$, and therefore currently rely on using only specifications that do not involve non-terminating behavior for any well-typed inputs at all.

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

**Fig. 1.** Trace acceptance.

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

Here $sum :: \mathsf{Num}\ a \Rightarrow \mathsf{Term}\ [a] \rightarrow \mathsf{Term}\ a$ and $length :: \mathsf{Term}\ [a] \rightarrow \mathsf{Term\ Int}$ are redefinitions of the respective standard functions in the context of a Term type constructor. Similarly, *getAll* and *getCurrent* have types $\mathsf{String} \rightarrow \mathsf{Term}\ [a]$ and $\mathsf{String} \rightarrow \mathsf{Term}\ a$, respectively.[3]

Values of a Term type can be evaluated under an appropriate variable environment via the function $evalTerm :: \mathsf{Term}\ a \rightarrow \mathsf{Environment} \rightarrow a$. Our encoding of terms here differs from the original presentation [12], where we used an applicative-style [7] representation for terms that enabled the usage of normal Haskell functions in specifications. The new encoding is useful in case we need access to the syntactic structure of terms (see Sect. 6), as we can preserve this information in the redefinitions. However, if we do not need such inspection of terms, redefining standard functions in the new context most likely does not add

[3] There are no guarantees that we can actually use a term constructed with *getAll* or *getCurrent* at any particular instantiation for type $a$. Checks happen at runtime.

any benefit. For this reason, our implementation is suitably polymorphic over the type constructor for terms used in specifications.

To facilitate checking programs (such as student submissions) against a specification such as the one seen above, we use an approach presented by Swierstra and Altenkirch [10] to acquire an inspectable representation of I/O behavior, plus random testing via QuickCheck [1] to generate test cases and test the candidate program according to the procedure described in Sect. 2.4.

So far, this is essentially the approach described in our previous work [12] (apart from the different term representation).[4] But how do we guarantee that the implementation behaves according to the formal on-paper definitions? That is, if the system tells a student that their submitted solution is correct, is that really the case? And conversely, does the system only reject wrong submissions and does it provide valid explanations in each case? In the next section we will look at exactly these questions. But there are also other important properties an educator might expect from the framework besides technical correctness. Generally, when posing tasks using the implemented system, there are various artifacts in play (some explicit and technical, some more virtual), such as:

- The idea/intention the educator has about what should be done in the task. In the case of the above example, the idea could be something like "I want them to realize a simple I/O loop, so they should write a program that reads a number and then as many further numbers and finally prints a sum."
- The DSL expression (and possibly additional data) capturing, hopefully, the desired behavior.
- The verbal task description handed to students ("Write a program which ...").
- A sample solution; for sanity checking and possibly for later also being handed to students.
- Any supporting material the students get as part of the task description. For example, a run of the sample solution on some specific input sequence.

All of these and potentially further artifacts must be kept in sync with each other in order to arrive at a consistent and usable exercise task. Therefore, we want to provide support for making sure that they indeed are in sync. One potential way to achieve this consistency is to generate some artifacts from others, along with correctness guarantees/arguments for those generators. Another way is to establish processes the educator follows in creating some artifacts either in isolation or together. For example, we can check different hand-written artifacts against each other inside the system itself. A simple example would be to check if a sample solution is accepted by the task specification in DSL form.

We will come back to such issues later. After establishing confidence in the technical core of the implementation, we will show that there are indeed certain provisions an educator can employ to support and verify their usage of

[4] A live online demonstration of the prototype implementation for that previous article is available at https://autotool.fmi.iw.uni-due.de/tfpie19, showcasing the approach. (Note that this demo still uses applicative-style terms.)

the framework. For now, Fig. 2 shows our current "bigger picture". The dashed arrows represent activities (creation of artifacts etc.) by the educator, while most of the solid arrows represent technical flow, i.e., where the implementation/system is active. As can already be seen, there are various connections between some source and some target that can be realized via different routes, indicating opportunities for automatic support of educator activities. We will come back to specific ingredients later on.

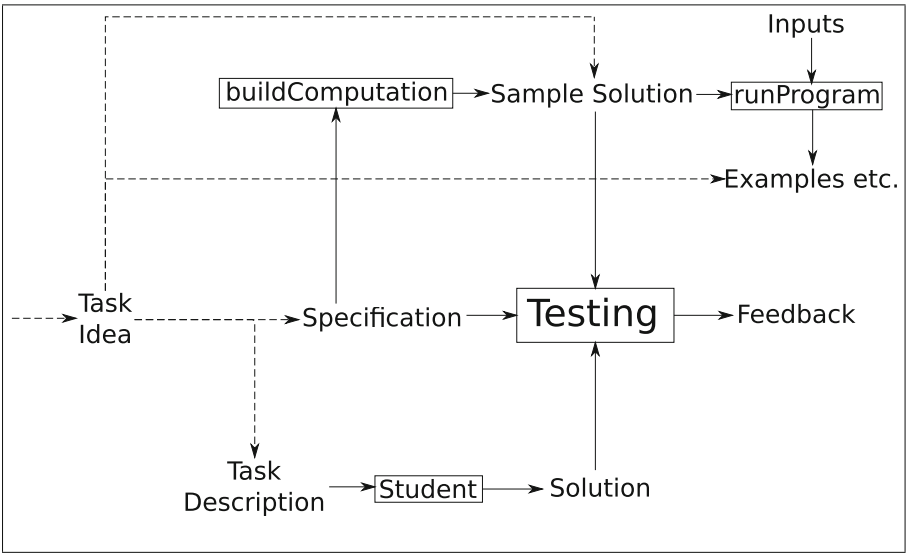


**Fig. 2.** Artifacts and flow.

## 4 Validating the Implementation

The guarantees we are going to provide to users of our implementation rely on the correctness of the technical core of the system, i.e., of the components involved in testing solution candidates against specifications. As described earlier (Sect. 2.4), testing is done by repeatedly sampling from the set of generalized traces for a specification and checking if the program under testing produces a matching trace for the same input sequence. The actual semantics of when a specification and a trace match is given by the Boolean-valued *accept*-function defined on specification-trace-pairs (see Fig. 1). The relationship between that semantics and the testing approach is stated as follows [12]:

Let s $\in$ *Spec* and t $\in$ *Tr*, then $accept(s, k_I)(t, \Delta_I) = \mathsf{True}$ if and only if there exists a $t_g \in traceSet(s, k_I^T)(\Delta_I)$ such that $t \prec t_g$.

The implementation thus does not need to rely on the *accept*-function to do the testing. However, implementing *accept* anyway gives us a way to check, even

programmatically to some extent (i.e., mechanically testing the test framework), whether the implementation behaves according to the on-paper definitions:

- The *accept*-function can be translated almost verbatim to a Haskell program.
- It is clear, from code inspection/review of this program, that both the Haskell and the on-paper version of *accept* compute the same function.
- Therefore, any implementation of the testing approach, no matter how technically involved, can be validated using the *accept*-semantics and turning the statement displayed above into an automatically checkable property or properties.

This reasoning does not just establish the connection between the implemented *traceSet*-function and the on-paper semantics but "in the other direction" also validates that the test framework assesses student submissions correctly.

While the reasoning above is, of course, no substitute for a formal correctness proof, it can still provide us with some confidence that an implementation of *traceSet* behaves correctly. For example, we test properties derived via this approach in our continuous integration setup. Due to that, we can refactor and extend our implementation with confidence and do not need to worry about its correctness after each change. Since at the moment we are mainly interested in exploring different features and implementation details for our framework, validating correctness through automatic tests is sufficient for us so far.

To automatically test that an implementation of *traceSet* behaves as stated above, we need to check two properties corresponding to the two directions of the bi-implication in the statement relating it to *accept*. The first one is:

1. If we have a specification $s$ and a trace $t$ such that $accept(s)(t)$ holds[5], then *the* generalized trace $t_g$ sampled from $traceSet(s)$ for the specific input sequence found in $t$ has to cover $t$.

To check this property, we need access to a source of pairs of specifications and traces for which $accept(s)(t) = \mathsf{True}$. Systematically building such pairs is not exactly easy. Therefore, our testing of this property currently relies on checking it for hand-written examples, i.e., on unit tests. Instead of writing down specific traces that match a given specification, one can also use full, known to be correct, programs and check that the testing procedure, usually employed for student submissions, never finds any counterexample. This way, one can view each of these unit tests as a very specific property test encoding a weakened version of the above property; the testing procedure itself will still use random inputs.

Fortunately, the property corresponding to the converse direction is far easier to check on a wide range of test cases. That property can be stated as follows:

2. For a specification $s$, sample a generalized trace using *traceSet*. From this generalized trace $t_g$, build an ordinary trace $t$ by randomly replacing each output set in $t_g$ by an element of that set (potentially dropping the output there altogether if $\varepsilon$ is chosen). Since, by construction, those are exactly the $t$ that are covered by $t_g$, now $accept(s)(t)$ has to evaluate to $\mathsf{True}$.

[5] For notational simplicity, we leave out the continuation and environment here.

Since the only input to this property is a specification $s$ without any further requirements, we are not limited to carefully hand-crafted test cases here but can instead use randomly generated specifications.

Randomly generated specifications have further uses as well. For example, they can be used for certain regression tests, for sanity checking of structural properties of the DSL that we do expect to hold, but which are not explicit from, say, the definition of the *accept*-semantics, such as that specifications form a monoid via sequential composition and neutral element $\mathbf{0}$, and even for checking more involved properties of the operations of the specification language, such as the not immediately obvious (but semantically important) equivalence between $s^{\rightarrow_{\mathbf{E}}}$ and $(s \cdot s^{\rightarrow_{\mathbf{E}}} \cdot \mathbf{E})^{\rightarrow_{\mathbf{E}}}$. They also help tie together further components of the overall framework/system under development; see Sect. 5. We even envision to use randomly generated specifications for creating random exercise tasks, in particular in connection with other ideas from Sect. 6 and beyond. So let us make explicit our current strategy for randomly generating specifications here.

### 4.1 Randomly Generating Specifications

While the basic structure of specifications is fairly simple and does generally not need to fulfill any invariants, there are two non-trivial aspects to randomly generating specifications: terms in general and loops and their termination in particular. First of all, one needs to decide on a set of available functions that can appear in terms. Then, terms can be generated according to this grammar:[6]

$$
\begin{array}{rcl}
\langle term \rangle & ::= & \langle function \rangle\ \langle terms \rangle \\
& | & \langle var \rangle \\
& | & \langle literal \rangle
\end{array}
\qquad
\begin{array}{rcl}
\langle terms \rangle & ::= & \langle term \rangle\ \langle terms \rangle \\
& | & \langle term \rangle \\
\langle var \rangle & ::= & getAll\ \langle id \rangle \\
& | & getCurrent\ \langle id \rangle
\end{array}
$$

For loops, we can use the same grammar to generate a condition, but it might be necessary to restrict the set of available functions even further. Otherwise, the generated loop might not be guaranteed to make progress toward termination, therefore leaving us with a specification describing non-terminating behavior. Given the right constraints on the available functions, we can build a terminating loop from three random specifications $s_1$, $s_2$, $s_3$. Our loop skeleton has the form $(s_1 \cdot (s_2 \measuredangle \mathbf{?} \searrow s_3))^{\rightarrow_{\mathbf{E}}}$, i.e., $s_1$ is a common prefix for every iteration round. Note that we do not have a suffix sequentially after the branching. Since we will insert an exit-marker into one of the branches, a suffix would only ever be used after the branch without that marker. Therefore, it can always be suffixed to that branch. Now we generate a condition $c$ and a specification $s^*$ that guarantees progress toward evaluating $c$ to $\mathsf{True}$. Then our terminating loop is $(s_1 \cdot (s_2^* \measuredangle c \searrow (s_3 \cdot \mathbf{E})))^{\rightarrow_{\mathbf{E}}}$, where $s_2^*$ is the result of inserting $s^*$ into $s_2$ at a random position.[7] Alternatively, we can also negate the condition and use the

[6] Of course, one has to take scoping and types into account as well.

[7] More precisely, we choose a random and unconditionally reached position.

loop $(s_1 \cdot ((s_2 \cdot \mathbf{E}) \swarrow not(c) \searrow s_3^*))^{\rightarrow_{\mathbf{E}}}$. A simple way to generate the condition and the progressing specification is to manually define a set of condition and progression pairs and then choose elements of this set at random. For example, we could have a pair with condition $len(x_A) > n$, for some $n > 0$ and variable $x$, and specification $[\triangleright x]^{\mathbb{Z}}$. Reading into $x$ guarantees that $len(x_A)$ is increasing, eventually exceeding $n$. Choosing this condition-progression-pair and assuming $s_1 = [\triangleright y]^{\mathbb{N}}$, $s_2 = [\{\varepsilon, 2 \cdot y_C\} \triangleright][\{sum(y_A)\} \triangleright]$, and $s_3 = \mathbf{0}$, we could insert the progressing $[\triangleright x]^{\mathbb{Z}}$ between the two outputs of $s_2$ and get:

$$([\triangleright y]^{\mathbb{N}} \cdot (([\{\varepsilon, 2 \cdot y_C\} \triangleright][\triangleright x]^{\mathbb{Z}}[\{sum(y_A)\} \triangleright]) \swarrow len(x_A) > n \searrow \mathbf{0} \cdot \mathbf{E}))^{\rightarrow_{\mathbf{E}}}$$

This method of generating termination conditions relies heavily on the set of hand-written conditions and their respective progressing specifications. In general, the ability to catch implementation errors through testing with randomly generated specifications depends on the possible terms we generate. Consider, for instance, these five specifications generated by our implementation as described:

$$([\{len(y_A)\} \triangleright][\triangleright z]^{\mathbb{Z}}(\mathbf{E} \swarrow not(len(x_A) > 1) \searrow [\triangleright x]^{\mathbb{Z}}))^{\rightarrow_{\mathbf{E}}}$$

$$[\triangleright n]^{\mathbb{Z}}[\{n_C\} \triangleright][\{n_C - n_C, n_C\} \triangleright](\mathbf{0} \swarrow null(x_A) \searrow [\triangleright m]^{\mathbb{Z}})$$

$$[\triangleright m]^{\mathbb{Z}}([\triangleright n]^{\mathbb{Z}} \swarrow len(n_A) > 0 \searrow \mathbf{E})^{\rightarrow_{\mathbf{E}}}[\{\varepsilon, sum(m_A), m_C\} \triangleright]$$

$$[\{sum(m_A)\} \triangleright](([\triangleright m]^{\mathbb{Z}} \swarrow null(m_A) \searrow \mathbf{0}) \swarrow len(x_A) = len(n_A) \searrow [\triangleright m]^{\mathbb{Z}})$$

$$[\{\varepsilon, sum(z_A)\} \triangleright][\triangleright n]^{\mathbb{Z}}(\mathbf{0} \swarrow len(x_A) < n_C * n_C \searrow ([\triangleright y]^{\mathbb{Z}} \swarrow n_C = n_C * n_C \searrow \mathbf{0}))$$

Clearly, most of the generated specifications do not resemble anything a user of the language would write. But for testing an implementation such specifications are precisely what we want, as they potentially trigger edge cases outside the implementer's imagination. We could, of course, attach additional constraints to the generation of specifications. However, too much restricting might lead to some errors never being triggered. On the other hand, restrictions can lead to more useful specifications that resemble actual use cases. For example, it might be a good idea to not allow terms like $x_C = x_C$ in a branching condition when generating specifications for usage in exercise tasks (see Sect. 6), but during validation of an implementation one might explicitly want such edge cases.

## 5 Empowering the Educator: An Interpreter Semantics

A central problem an educator might have when writing specifications in the DSL so far is the fact that there is no direct way to inspect what behavior a specification represents. When writing normal programs, we are used to a fundamentally different situation: during development we can execute a candidate (the current program version) and play around with different inputs to confirm that we are actually on the right track.

In order to get the same possibility when developing specifications, we wanted an interpreter that given a specification behaves exactly like a program matching that specification would. Essentially, the desire is for a function of the following type: $[\![\cdot]\!] :: \mathsf{Specification} \to \mathsf{IO}\ ()$. Since specifications conceptually use a global variable environment, our interpreter has to be stateful as well, even beyond the "I/O state". Also, in order to correctly terminate loops, some way to abort a running program part and to recover from such an abort would be handy, to emulate behavior similar to a `break` command. Thus motivated, and in order to keep the interpreter's structure itself simple, we do not target $\mathsf{IO}\ ()$ directly but instead the interpreter produces a value of type $\mathsf{Semantics}\ ()$, which is declared as follows:

$$
\begin{aligned}
&\textbf{newtype}\ \mathsf{Semantics}\ a = \mathsf{Semantics}\ \{\mathit{runSemantics} \\
&\qquad :: \mathsf{Environment} \to \mathsf{IO}\ (\mathsf{Either}\ \mathsf{Exit}\ a, \mathsf{Environment})\} \\
&\textbf{data}\ \mathsf{Exit} = \mathsf{Exit} \\
&\textbf{type}\ \mathsf{Environment} = [(\mathsf{String}, [\mathsf{Int}])]
\end{aligned}
$$

The Semantics type constructor is a monad, in fact, an inlined version of the following monad transformer stack [6]: ExceptT Exit (StateT Environment IO). Thus, it provides us with at least the operations *readLn*, *print*, *gets*, *modify*, *throwError*, and *catchError*. These operations give us everything we need to manage a global state and to abort loops in a convenient way. Additionally using the function *evalTerm* discussed in Sect. 3, as well as

$$\mathit{store} :: \mathsf{String} \to \mathsf{Int} \to \mathsf{Environment} \to \mathsf{Environment}$$

for the actual updating of environments, the interpreter is then defined thus:

$$
\begin{aligned}
&[\![\cdot]\!] :: \mathsf{Specification} \to \mathsf{Semantics}\ () \\
&[\![\mathbf{0}]\!] &&= \mathit{return}\ () \\
&[\![s_1 \cdot \ldots \cdot s_n]\!] &&= \textbf{do}\ [\![s_1]\!] \\
&&&\quad\ \vdots \\
&&&\quad\ [\![s_n]\!] \\
&[\![[\triangleright x\,]^{\tau}]\!] &&= \mathit{modify} \circ \mathit{store}\ x =\!\!\ll \mathit{readLn} \\
&[\![[\,\{\varepsilon, \ldots\} \triangleright\,]]\!] &&= \mathit{return}\ () \\
&[\![[\,\{t, \ldots\} \triangleright\,]]\!] &&= \mathit{print} =\!\!\ll \mathit{gets}\ (\mathit{evalTerm}\ t) \\
&[\![s^{\rightarrow^{\mathbf{E}}}]\!] &&= \textbf{let}\ \mathit{loop} = \textbf{do}\ [\![s]\!] \\
&&&\qquad\qquad\qquad\ \ \mathit{loop} \\
&&&\quad\ \textbf{in}\ \mathit{catchError}\ \mathit{loop}\ (\lambda \mathsf{Exit} \to \mathit{return}\ ()) \\
&[\![s_1 \angle c \searrow s_2]\!] &&= \mathit{ifM}\ (\mathit{gets}\ (\mathit{evalTerm}\ c))\ [\![s_2]\!]\ [\![s_1]\!] \\
&[\![\mathbf{E}]\!] &&= \mathit{throwError}\ \mathsf{Exit}
\end{aligned}
$$

To get a runnable IO computation from a specification, we can start the interpreter with an empty environment and ignore both a potential Exit value and the final environment as follows:

```
buildComputation :: Specification → IO ()
buildComputation s = void (runSemantics ⟦s⟧ (map (,[]) (vars s)))
```

Interestingly, and usefully, the interpreter can also be seen as an alternative formulation of the semantics of specifications in the first place: to understand what behavior a specification is representing, understanding either the *accept*-function or the interpreter semantics suffices. For someone already with a good grasp of Haskell and monads, the latter option could be substantially more attractive. That is, the target audience for the interpreter semantics are certainly not our students taking the course, but neither is that the case for the *accept*-function. Consider, though, the situation of handing the job of creating new exercise tasks to teaching assistants, which in our case could be advanced students from previous years. They need to be able to very clearly understand the semantics of the DSL in order to be successful at task creation. The *accept*-function is probably not for them, but they can certainly work informed by $[\![\cdot]\!]$ as given above. The *accept*-function, on the other hand, as the more mathematical and less programmatical foundation, is relevant in the background when extending the overall framework, devising new testing and feedback methods, etc.

These considerations rely on the *accept*-function and the interpreter semantics being equivalent, and obviously just claiming that they are is not very convincing. But fortunately we can formulate a simple, mechanically checkable, property that relates the interpreter to the *accept*-function. This property states that every interpretation of a specification has to lead to a computation that can only produce traces acceptable by that specification. We can capture it thus:

```
prop :: Specification → Property
prop s = buildComputation s `satisfiesAccept` s
```

Of course, it is basically just a play on normal correctness checking of candidate solutions (the candidate now not being a student submission but an interpreter call). In order to check this property automatically, randomly generated specifications (see Sect. 4.1) come in handy again.

Note that the above property covers only the soundness of the interpreter. It does not test whether every valid trace for the given specification can be generated by the interpretation result. Looking at the interpreter's definition as a deterministic definition, the resulting program clearly cannot, in general, produce all traces the specification would accept, since we always choose one particular value to output and discard all other possibilities. In order for the interpreter to act as a semantics alternative to *accept*, we need to view it as containing some form of non-determinism. For example, we can interpret the selection of an element from the set of possible outputs as a random choice or change the definition to ultimately produce a list of all possible combinations of choices. (We have not done any of that yet.)

## 6 Further Support: Validation and Program Generation

Recall that for each exercise task there are five artifacts that an educator might need to keep consistent (see Sect. 3): the general idea of what the task is to be

about, the specification expression used for testing, a task description for students, a sample solution, and additional material like example runs of programs.

In the previous section we have shown how to run a specification as if it were a program. Thus we already have an automated way to create a correctly behaving computation from a given specification and use it to drive example runs. This interpreting of specifications does not yet cover our need for a sample solution, though, since we do not directly get any actual program code that could be shown to students. What we have instead is a way for an educator to validate their own sample solution by comparing it to both the interpreted and the actual specification (via the testing procedure). By doing so, the educator can validate that their idea for the task matches the written specification, since a mismatch between these two might manifest as a mismatch in observed behavior between "solutions".

Now the last artifact (as per Sect. 3, Fig. 2) that is not yet connected to the others in any systematic way is the description of the task as handed to students. Up front, it might seem quite impossible to automatically generate any reasonably formulated task description. But if we shift our focus away from classic verbal descriptions, instead to tasks that require the re-implementation of some (imperative) program with Haskell, then generating task descriptions gets way easier. For example, we might want to pose tasks of the form "Write a Haskell program that has the same behavior as the following Python program ..." (building on the students' knowledge from their introductory programming course), and then we just need to be able to automatically generate Python code from a specification expression. Such tasks are not generally what one always wants, but they work well when the goal is to highlight the differences between I/O in Haskell and in languages with mainly ambient effects [2]; concerning type distinctions between pure and impure expressions, syntactic differences like the two relevant forms of binding in Haskell's **do**-blocks, **let** vs. ←, etc.

To actually generate the Python code needed, we can define a translation function similar in structure to our interpreter from the previous section:

$$
\begin{aligned}
[\![\mathbf{0}]\!] &= \mathbf{pass} \\
[\![s_1 \cdot \ldots \cdot s_n]\!] &= [\![s_1]\!] \\
&\quad \ldots \\
&\quad [\![s_n]\!] \\
[\![[\triangleright x\,]^{\tau}]\!] &= x_A \mathrel{+}= [\mathbf{int}(\mathbf{input}())] \\
[\![[\,\{\varepsilon, \ldots\} \triangleright\,]]\!] &= \mathbf{pass} \\
[\![[\,\{t, \ldots\} \triangleright\,]]\!] &= \mathbf{print}(\lfloor t \rfloor) \\
[\![s^{\rightarrow^{\mathbf{E}}}]\!] &= \mathbf{while}\ \mathrm{True}: \\
&\quad [\![s]\!] \\
[\![s_1 \angle c \searrow s_2]\!] &= \mathbf{if}\ \lfloor c \rfloor: \\
&\quad [\![s_2]\!] \\
&\ \mathbf{else}: \\
&\quad [\![s_1]\!] \\
[\![\mathbf{E}]\!] &= \mathbf{break}
\end{aligned}
$$

The target domain is now not a type of I/O semantics but actual syntactic program text. Additionally to the program text emitted by the above translation function, we also need to prepend an initialization $x_A = []$ for each variable $x$ used in the specification. Moreover, $\lfloor t \rfloor$ as used above means to replace all variable occurrences of the form $x_C$ in a term $t$ by $x_A[-1]$. Note that in order for the translation to be carried out automatically, we need access to the structure of terms used for outputs and for branching. Otherwise we could not generate textual descriptions, including the replacement of certain variable occurrences via $\lfloor t \rfloor$. Our implementation of the specification language therefore provides an appropriate representation of terms, as mentioned in Sect. 3.

Applying the code generation procedure to our earlier example specification $[\triangleright n]^{\mathbb{N}}([\triangleright x]^{\mathbb{Z}} \measuredangle len(x_A) = n_C \searrow \mathbf{E})^{\rightarrow^{\mathbf{E}}}[\{sum(x_A)\} \triangleright]$, we obtain the following Python program:

```
n_A = []
x_A = []

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

Even though this program has the intended behavior, it is not an ideal program to hand to students. Due to the compositional nature of the translation, the resulting program code does not generally exploit any information regarding the overall structure of the specification. Thus, we end up with programs that do not necessarily adhere to good programming practice. It is possible to manipulate such programs further or to optimize the process generating them. For example, if for some variable $x$ the operation $\lfloor \cdot \rfloor$ never (throughout the whole processing above) encountered variant $x_A$, then for that variable we can use a simple version $x_C$ only, without initialization and without altering it in $\lfloor \cdot \rfloor$, and $x_C = \mathbf{int}(\mathbf{input}())$ instead of $x_A \mathrel{+}= [\mathbf{int}(\mathbf{input}())]$ in translations of input operations for it. Doing so for the example used above, we would end up with:

```
x_A = []

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

This program is still not ideal and further improvements are needed before we can use such artifacts for programming education. One of the most obvious improvements would be to detect special cases like "**while** True: **if** ... **break else**: ..." and to transform them into other **while**-loops with an explicit loop condition. We have not done any deeper investigations into this area, as of yet, but definitely plan to do so in the future since we see a lot of potential for automatic task generation based on this approach.

An important difference to the previous section is that the target audience for the generated Python code are indeed the students taking the course. And certainly the code generation approach advocated here is not limited to Python programs. Alternatively to interpreting a specification as a Haskell value in a semantics type as in the previous section, we could also emit the actual program text *there*, given a printable representation of the Terms used by the specification exists. We have not implemented this Haskell version of the code generation yet, but by looking at the definition of the interpreter one could imagine translating the DSL expression

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

to the following program:

```
prog :: Semantics ()
prog = do
  modify ∘ store "n" =≪ readLn
  let loop = do ifM (gets (evalTerm (
                                length (getAll "x") == getCurrent "n")))
                  (throwError Exit)
                  (modify ∘ store "x" =≪ readLn)
                loop
  catchError loop (λExit → return ())
  print =≪ gets (evalTerm (sum (getAll "x")))
```

In principle, this translation could let an educator automatically generate a correct sample solution. However, in the above form this approach does not lend itself directly for generating sample solutions presentable to students, even less so than for the case of targeting Python. Not only is the usage of the state and exception monads nowhere near an idiomatic solution for such a simple specification; they also make it somewhat difficult to identify the actually interesting part of the computation here. But by inlining the monad transformer operations and simplifying the resulting program, the educator could systematically derive a presentable program. It might even be possible to do this derivation fully automatically with the use of some program analysis and transformations, also including specialized simplification strategies as mentioned for Python above.

Again, we have not done any deeper exploration on such transformations yet, but will do so in the future.

## 7 Putting It All Together

To summarize, let us once again consider our running example and go through the steps we would take to implement the task inside the presented framework.

The first step is coming up with an idea for what the task should be. For our summation example, the idea might be formulated as follows: *"We want students to realize a simple I/O loop, so they should write a program that reads a number and then as many further numbers and finally prints a sum."* Next, we write a task description, a specification, and a sample solution based on this idea:

*"Write a program which first reads a positive integer n from the console, then reads n integers one after the other, and finally outputs their sum."*

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

```
main = do
  n ← readLn
  let loop xs =
        if length xs == n
          then print (sum xs)
          else do
            x ← readLn
            loop (x : xs)
  loop [ ]
```

To verify that these components are consistent with each other and our idea, we now use the different connections shown in Fig. 2 to relate them to each other. First off, we can run both our sample solution and the specification, using the interpreter, on some sample inputs to see if their behavior matches our idea of the task as well as each other. Next, we use the testing procedure to make sure that the sample solution fulfills the specification. If all these checks are successful, we can be confident that the idea, the sample solution, and the specification are indeed consistent. What is left is validating the written task description. Without the ability to automatically generate useful descriptions, or when hand-written descriptions are preferable, this has to be done by the usual careful inspection of the description.

With confidence in the consistency established, we can then generate supporting material; for example, we can give a run of the sample solution on some specific input. For instance, we can add the following line to our task description: *"Example: After reading 2, 7, and 13, your program should print 20."*

We cannot yet report on any concrete experience using this workflow, as we will only start using it in the upcoming iteration of our Haskell course.

One interesting detail to note is the fact that the presented approach is, in principle, not limited to tasks dealing with I/O. Given a suitable specification language and testing framework, the basic idea of (semi-)automatically generating artifacts and cross-validating them against manually created ones, and each other, is certainly applicable in other settings as well.

## 8 Related Work

As mentioned in Sect. 3, our implementation builds upon an inspectable representation of side-effecting programs [10]. The Haskell IOSpec library[8] implements such representations not only for console I/O but supports also forking processes, mutable references, and software transactional memory. However, it only features a very minimal API. Also, no higher-level abstractions currently exist.

Another tool for testing stateful computations is the state machine version of QuickCheck for Erlang [4,5].[9] Instead of testing specific programs, like we do, it can be used to test stateful APIs. Behavior is specified as 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 differs from the semantic model is shrunk to provide a small counterexample.

## 9 What Next?

We have an implementation of the specification language from earlier work [12] along with supporting components, correctness checkers for both student submissions and the framework itself, and semantics/code generators. Our hope is to benefit from this investment when we grow the specification language, and with it the overall framework, to accommodate further needs on the education side. Being able to safely grow the framework is precisely what the "keeping in check" part of this article's title refers to: when the expressiveness of the underlying DSL is to be extended, different parts of the implementation have to be revisited as well, and we expect that the thoughts and work put in now at the beginning will pay off in the sense of maintainability and certain guarantees. By way of an outlook, let us discuss a concrete extension we have in mind.

At the moment, the specification language does not yet talk about how a program (e.g., a student submission) should cope with possible input errors. For example, in $[\triangleright n]^{\mathbb{N}}(\dots)^{\rightarrow^{\mathbf{E}}}\dots$ we expressed that the first number that is read in should not be negative, but what happens otherwise is left completely unspecified. Of course, the *accept*-function is formulated in such a way that a trace starting with a negative input value would be rejected here, but the actual testing of student submissions deliberately only presents inputs that are well-formed according to the specification. From a different perspective, the interpreter given in Sect. 5 (which builds computations that also serve as possible sample solutions; see the second half of that section as well as Sect. 6) completely ignores the $\tau$ argument in this line:

[8] https://hackage.haskell.org/package/IOSpec.

[9] A Haskell version can be found at http://hackage.haskell.org/package/quickcheck-state-machine.

$$[\![ [\triangleright x]^{\tau} ]\!] = \mathit{modify} \circ \mathit{store}\ x =\!\ll \mathit{readLn}$$

If we were to instead write

$$\begin{aligned}[\![ [\triangleright x]^{\tau} ]\!] = \mathbf{do}\ & v \leftarrow \mathit{readLn} \\ & \mathit{when}\ (v \notin \tau)\ (\mathit{error}\ \texttt{"blow up"}) \\ & \mathit{modify}\ (\mathit{store}\ x\ v)\end{aligned}$$

then the potential runtime error added there would never actually be triggered during any automatic test runs, simply because a decision was made to not subject student submissions to ill-formed or otherwise undesirable inputs. The rationale for that decision is that students just beginning to learn I/O programming in Haskell should not have to worry about checking inputs for correctness. But what about later? At some point we might want to explicitly require them to do so, that is, to turn the management of expectations about input values from a job of the testing framework into a job of the students. And we might want to be able to be selective about at which input actions such checks are required, and at which not, as well as to retain flexibility concerning how exactly student submissions should deal with incorrect inputs.

Our suggestion now is to extend the specification language by two additional atomic forms: $[\triangleright x]^{\tau}_{\perp}$ and $[\triangleright x]^{\tau}_{\circlearrowleft}$. The intuitive semantics of the first variant is that if an input outside the set $\tau$ is read, the program stops (in a controlled fashion, not via a runtime error), while that of the second variant is that if an input outside the set $\tau$ is read, the user is prompted again (and possibly again and again) for an input until the value read is indeed in $\tau$. In the *accept*-function, these new forms would be defined as follows:

$$\mathit{accept}([\triangleright x]^{\tau}_{\perp} \cdot s', k)(t, \Delta) = \begin{cases} \mathsf{True} & , \text{if } t = ?v\,\mathsf{stop} \wedge v \notin \tau \\ \mathit{accept}(s', k)(t', \mathit{store}(x, v, \Delta)) & , \text{if } t = ?v\,t' \wedge v \in \tau \\ \mathsf{False} & , \text{otherwise} \end{cases}$$

$$\mathit{accept}([\triangleright x]^{\tau}_{\circlearrowleft} \cdot s', k)(t, \Delta) = \begin{cases} \mathit{accept}([\triangleright x]^{\tau}_{\circlearrowleft} \cdot s', k)(t', \Delta) & , \text{if } t = ?v\,t' \wedge v \notin \tau \\ \mathit{accept}(s', k)(t', \mathit{store}(x, v, \Delta)) & , \text{if } t = ?v\,t' \wedge v \in \tau \\ \mathsf{False} & , \text{otherwise} \end{cases}$$

and the remaining components of the framework, the checkers, generators, etc., would be extended as well, while relying on existing invariants and established connections/correspondences.

## References

1. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: International Conference on Functional Programming, Proceedings, pp. 268–279. ACM (2000). https://doi.org/10.1145/351240.351266

2. Filinski, A.: Controlling effects. Ph.D. thesis, Carnegie Mellon University (1996)
3. Gibbons, J., Wu, N.: Folding domain-specific languages: deep and shallow embeddings (Functional Pearl). In: International Conference on Functional Programming, Proceedings, pp. 339–347. ACM (2014). https://doi.org/10.1145/2628136.2628138
4. Hughes, J.: QuickCheck testing for fun and profit. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 1–32. Springer, Heidelberg (2006). https://doi.org/10.1007/978-3-540-69611-7_1
5. Hughes, J.: Experiences with QuickCheck: testing the hard stuff and staying sane. In: Lindley, S., McBride, C., Trinder, P., Sannella, D. (eds.) A List of Successes That Can Change the World. LNCS, vol. 9600, pp. 169–186. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30936-1_9
6. Liang, S., Hudak, P., Jones, M.P.: Monad transformers and modular interpreters. In: Principles of Programming Languages, Proceedings, pp. 333–343. ACM (1995). https://doi.org/10.1145/199448.199528
7. McBride, C., Paterson, R.: Applicative programming with effects. J. Funct. Program. **18**(1), 1–13 (2008). https://doi.org/10.1017/S0956796807006326
8. Peyton Jones, S.L., Wadler, P.: Imperative functional programming. In: Principles of Programming Languages, Proceedings, pp. 71–84. ACM (1993). https://doi.org/10.1145/158511.158524
9. Siegburg, M., Voigtländer, J., Westphal, O.: Automatische Bewertung von Haskell-Programmieraufgaben. In: Proceedings of the Fourth Workshop "Automatische Bewertung von Programmieraufgaben", pp. 19–26. GI (2019). https://doi.org/10.18420/abp2019-3
10. Swierstra, W., Altenkirch, T.: Beauty in the beast – a functional semantics for the awkward squad. In: Haskell Workshop, Proceedings, pp. 25–36. ACM (2007). https://doi.org/10.1145/1291201.1291206
11. Waldmann, J.: Automatische Erzeugung und Bewertung von Aufgaben zu Algorithmen und Datenstrukturen. In: Proceedings of the Third Workshop "Automatische Bewertung von Programmieraufgaben", CEUR Workshop Proceedings, vol. 2015. CEUR-WS.org (2017)
12. Westphal, O., Voigtländer, J.: Describing console I/O behavior for testing student submissions in Haskell. In: Eighth and Ninth International Workshop on Trends in Functional Programming in Education, Proceedings, EPTCS, vol. 321, pp. 19–36. EPTCS (2020). https://doi.org/10.4204/EPTCS.321.2