JFP **35**, e18, 61 pages, 2025.

doi:10.1017/S0956796825100075

Automatically testing console I/O behavior of student submissions in Haskell

OLIVER WESTPHAL
Universität Duisburg-Essen, Germany
(*e-mail:* oliver.westphal@uni-due.de)

JANIS VOIGTLÄNDER
Universität Duisburg-Essen, Germany
(*e-mail:* janis.voigtlaender@uni-due.de)

Abstract

Good test-suites are an important tool to check the correctness of programs. They are also essential in unsupervised educational settings, like automatic grading or for students to check their solution to some programming task by themselves. For most Haskell programming tasks, one can easily provide high-quality test-suites using standard tools like QuickCheck. Unfortunately, this is no longer the case once we leave the purely functional world and enter the lands of console I/O. Nonetheless, understanding console I/O is an important part of learning Haskell, and we would like to provide students the same support as with other subject matters. The difficulty in testing console I/O programs arises from the standard tools' lack of support for specifying intended console interactions as simple declarative properties. These interactions are however essential in order to determine whether a program behaves as desired. We describe the console interactions of a program by tracing its text input and output actions. In order to describe which traces match the intended behavior of the program under test, we present a formal specification language. The language is designed to capture interactive behavior found in commonly used textbook exercises and examples, or as much of it as possible, as well as in our own teaching, while at the same time retaining simplicity and clarity of specifications. We intentionally restrict the language, ensuring that expressed behavior is truly interactive and not simply a pure string-builder function in disguise. Based on this specification language, we build a testing framework that allows testing against specifications in an automated way. A central feature of the testing procedure is the use of a constraint solver in order to find meaningful input sequences for the program under test.

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

potentially error-prone, we employ Autotool, an e-learning system used in several German universities (Waldmann, 2017; Siegburg *et al*., 2019), for automatically testing submissions against a task specific test-suite, e.g., sets of QuickCheck properties (Claessen & Hughes, 2000), whenever possible. As an added benefit students get immediate feedback and can revise their submissions accordingly and incrementally.

Our approach relies on high-quality test-suites for each exercise task. For tasks about implementing pure functions this is easy to achieve using standard techniques for designing QuickCheck properties (Hughes, 2020). When switching from the pure context to console I/O, these techniques are unfortunately no longer directly applicable. The main problem is the non-availability of easy, lightweight and declarative techniques for specifying requirements on the interaction portion of console I/O programs. However, these interactions determine whether a program behaves as desired or not, especially for programs of type *IO* ().

Nonetheless, we would like our students to have access to the same level of support for console I/O tasks as they are used to for programming pure functions. Fortunately, we do not have to start completely from scratch. Swierstra & Altenkirch (2007) showed how one can change the monad underlying a Haskell I/O program in order to get an inspectable representation of a console I/O program (see Section 2). Using this technique, we can in principle check properties formulated over executions via QuickCheck or similar tools. Practical application, however, is cumbersome. Specifically, for every I/O exercise task we want to grade automatically, we need to implement three separate components:

1. A generator of input sequences suitable for testing the candidate program. As we will see later, this is not as easy as one might initially imagine. This is especially the case if we want to ensure certain coverage conditions.
2. A way of checking if a given execution trace exhibits the desired behavior for a given input sequence.
3. A means of providing feedback in case the behavior did not match the expectations, e.g., an explanation of the mismatch or an example of what would be correct behavior for the relevant input sequence.

These components have to cover a lot of different cases even for a single exercise task, since we typically allow students some degree of freedom when it comes to how their program should prompt for input values, in what form exactly it should print any computed result values, whether there are additional/optional output messages, etc. Moreover, the components are not directly related to each other, much less derived from a common source, thus leaving room for inconsistencies and other mistakes. Using this approach in practice made it clear to us that an overall framework is needed to effectively test I/O exercise tasks.

In previous work, we started building such a framework (Westphal & Voigtländer, 2020*a*, TFPIE'19). At the core of our approach is a small formal specification language for console I/O behavior. Specifications are structurally similar to lexical analysis regular expressions, but are augmented with global variables that track state and history of program runs and conditional branching and iterations based on that state. This allows us to express an interesting range of dynamic behavior. The language also supports encoding

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

of optional or ambiguous output behavior, to lessen the restrictions on how exactly programs communicate information. The language is kept intentionally minimal, enabling easy analysis of specifications and further automation. It is however expressive enough to cover almost all console I/O related programs and tasks commonly found in introductory Haskell textbooks. Given a specification of desired behavior, the three components needed for testing are automatically derived.

In this article, we present, from the ground up, an extended version of the specification language and testing framework. The language itself is largely the same. It is presented with an updated syntax and extended with ways of specifying dedicated handling of possible input errors as sketched informally in previous work (Westphal & Voigtländer, 2020*b*, FLOPS'20). The main shortcoming of the old system was a rather naive approach to input generation, interleaving random value generation with concrete execution. This restricted effective applications of the framework to only a small set of I/O behavior. We improve upon this by making the following contributions:

In order to improve the generation of input sequences, we use simple symbolic execution of specifications to generate constraints on input values. These constraints describe paths through the specified behavior, i.e., different abstract instances of the behavior. We search for input sequences that fulfill these constraints with the help of the Z3 SMT solver (de Moura & Bjørner, 2008). In order to generate multiple inputs for the same path, we use a sampling technique based on MaxSAT problems (Bjørner *et al*., 2015). Similarly to tools like SmallCheck (Runciman *et al*., 2008), we then test programs exhaustively on behavioral paths with lengths up to a certain cutoff. This ensures good coverage of different concrete variants of the specified behavior. If any counterexamples are found, we return the one with the shortest input sequence to make it easier to identify the underlying mistake. None of these improvements is novel on its own, but the combination and application of these techniques improves the quality of the framework's automated testing capabilities.

We implement the extended framework via an embedded domain-specific language (EDSL) in Haskell. We also give a precise description of the syntax and semantics of our language as well as a detailed description of trace comparison and constraint generation. Compared to earlier work, the formalization is refined and extended to incorporate the new input generation method. In terms of presentation, this leads to an untangling of input generation and specification interpretation and therefore an exposition more in line with how the implementation works. The formal syntax definition of specifications is refined to only allow a certain kind of well-behaved iterations, making the language's semantics more robust regarding issues of termination. Due to the simplicity of the language, the relevant property can be checked statically and therefore has benefits also during specification construction. Moreover, we realized that our previous notion of program correctness was too weak and replaced it with a more rigorous approach that now correctly accounts for the set of input sequences on which a program must behave in accordance with a given specification. We also relate the updated notion of program correctness to the actual testing procedure, thereby providing formal justifications for the overall correctness of our system that were not present in earlier work. For the sake of simplicity of the described formalism, we will only consider programs that read and write integers. It is however possible to generalize the approach to, for example, include string values, and the implemented EDSL

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

actually does so. Along with the focus on integers in the formalization, we will assume that I/O programs to test are written using the two primitive operations *readLn* and *print* that incorporate conversion from and to strings.[1]

Also, we previously reported on how to make sure the testing framework retains its desired semantics when the specification language is extended (Westphal & Voigtländer, 2020*b*, FLOPS'20). The new input generation method, the extended syntax of the specification language, and a general overhaul of the EDSL combinators all are examples of such changes and in fact those guardrails for preserving the language's semantics proved to be very useful. The basic idea is to leverage the fact that the semantics of the specification language is given via a relatively simple acceptance predicate on program traces but the actual testing is done by an independent more practical procedure. The acceptance predicate is easy to implement correctly, due to its simplicity. The more involved testing procedure is then validated against the acceptance predicate using a mix of QuickCheck properties and unit tests.

The rest of this article is structured as follows:

- Section 2 describes how we used to test console I/O programs in the past using the technique presented by Swierstra & Altenkirch (2007).
- Section 3 presents a general overview of our language and gives an intuitive description of the intended semantics.
- Section 4 showcases the problems of our old method of input generation from (Westphal & Voigtländer, 2020*a*, TFPIE'19).
- In Section 5, we then demonstrate how to better generate input sequences for testing with the help of constraint solving.
- Section 6 describes the full testing procedure.
- Section 7 showcases the implemented framework, including a short evaluation of expressiveness (culminating in Figure 5) and discussions on limitations of the new input generation method and on the usefulness of implementing a specification interpreter.
- In Section 8, we give the formal definitions for syntax and semantics of the specification language, as well as the definitions used for trace matching and input generation. Additionally, we give formal statements relating the semantics to the latter concepts in the form of a lemma and a conjecture. We provide evidence for this conjecture but do not present a complete proof, mainly because a full proof would be extremely tedious and without any new insights over the presented evidence. Our testing procedure is correct under the assumption that the conjecture holds, which we firmly believe to be the case.
- Section 9 reports on our experience using the implemented framework in the context of our programming paradigms course.
- We conclude after mentioning related work (Section 10) and possibilities to further build upon the presented language and testing framework (Section 11).

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

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

2 Ad-hoc testing of console I/O programs

Before we dive into the design of our specification language, let us first look at the situation we faced when testing console I/O programs before we first developed that language.

Consider the following verbal description of a non-I/O function one might give as an exercise task to a beginning programmer:

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

A simple Haskell solution could look like this:

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

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

```
propSing :: Int → Bool
propSing = λx → sum [x] == x

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

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

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

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

```
main :: IO ()                 loop :: Int → Int → IO ()
main = do n ← readLn          loop 0 res = print res
          loop n 0            loop n res = do x ← readLn
                                              loop (n − 1) (x + res)
```

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

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

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

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

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

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

Corresponding tests can be automated by following the approach presented by Swierstra & Altenkirch (2007). First, an alternative monad is defined that represents a semantic domain for console I/O programs:[2]

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

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

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

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

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

Now, any potential Haskell solution to the I/O task given further above, *main* = **do** ..., can not only be used at type *IO* (), but also at type IO_{rep} (). We can then "run" *main* in a kind of simulation mode that produces an explicit trace as a data structure when given a concrete input sequence, because values of type IO_{rep} are more inspectable than those of the normal *IO* type:[3]

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

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

Now we can define, per exercise task, a predicate *checkCorrectness* :: *Trace* → *Bool* that checks whether some trace exhibits the desired behavior. For our summation example this can look as follows:

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

[3] Note that run_{rep} completes successfully only when the provided user program consumes exactly all inputs offered to it. Exceptions thrown due to partiality here are already grounds for rejection of the supposed solution to the underlying I/O task.

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

```
checkCorrectness :: Trace → Bool
checkCorrectness (Read n t') =
  case readMaybe n of
    Just v → checkLoop v 0 t'
    Nothing → False
checkCorrectness _ = False

checkLoop :: Int → Int → Trace → Bool
checkLoop 0 s (Write text Stop) = text == show s
checkLoop n s (Read x t') =
  case readMaybe x of
    Just v → checkLoop (n − 1) (s + v) t'
    Nothing → False
checkLoop _ _ _ = False
```

This predicate exactly captures the set of desired traces given above. Next we define a generator of valid inputs for the specific exercise task like so:

```
validInputs :: Gen [String]
validInputs = do n ← chooseInt (0, 10)
                 xs ← vectorOf n $ chooseInt (−100, 100)
                 pure $ map show (n : xs)
```

With these two components we can use QuickCheck again to automatically test whether a submitted program has the intended behavior.

```
testProgram :: IO_rep () → IO ()
testProgram prog = quickCheck $ forAll validInputs $ checkCorrectness ∘ run_rep prog
```

This approach works reasonably well for our simple example. But writing such a generator and the correctness checking predicate is generally not as straightforward as one might hope. Ignoring for a moment the restriction concerning printing integers only, we usually want students to have some freedom when it comes to the formatting of outputs. In the above form, *checkCorrectness* will reject student programs that, e.g., prefix the sum with something like `"The sum is ..."`. Moreover, in many scenarios it makes sense to have other optional outputs. For the summation task this might be a "Welcome" message or some counter in between reading summands. Extending the checking predicate to account for these things quickly becomes difficult to maintain. And concerning input generation, one usually has to find a clever way to avoid naive "generate and test" loops for constraints rarely satisfied accidentally (above, the constraint would be that the length of the generated list is its first element plus one).

Figure 1 shows an input generator and a checking predicate for a variant of another simple task that we regularly use in our course:

> *Write a program that reads in integers until the two most recently entered integers sum up to zero. Then output the number of integers the program has read and stop. The program may write optional decorations and prompts.*

Even though the task and a sample solution are conceptually not too different from the summation one, generating test data and checking trace correctness is quite a bit more involved now. The code in Figure 1 is heavily optimized. Instead of dealing with the optional outputs directly, we first normalize the trace by dropping all outputs before the final input. Note that *checkCorrectness* receives the input sequence as a separate [*Int*] parameter now, so that we do have access to the expected number of read values and can use this information during the trace normalization. This works because we know the input

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

```
  -- trace argument comes from running the program on the given valid input sequence
checkCorrectness :: [Int] → Trace → Bool
checkCorrectness is t = let n = length is
                        in show n `isInfixOf` finalWrite (normalize n t)
  -- normalize trace (remove all optional writes and combine final writes into one)
normalize :: Int → Trace → Trace
normalize _ Stop          = Stop
normalize n (Read i t)    = Read i (normalize (n − 1) t)
normalize 0 (Write o t) = case normalize 0 t of
                            Write o' t' → Write (o ++ o') t'
                            t'          → Write o t'
normalize n (Write _ t) = normalize n t

finalWrite :: Trace → String
finalWrite Stop          = ""
finalWrite (Read _ t)    = finalWrite t
finalWrite (Write o _) = o

validInputs :: Gen [Int]
validInputs = do n ← chooseInt (0, 10)
                 xs ← vectorOf (n + 1) (chooseInt (−100, 100)) `suchThat` pred
                 return (xs ++ [negate $ last xs])
                 where
                   pred ys = and $ zipWith (λx y → x + y /= 0) ys (tail ys)
```

Fig. 1: Hand-written correctness checker and input generator.

sequence that produced the trace; it is the one generated by *validInputs*. Additionally, we normalize possible consecutive outputs at the end of the trace into a single output. This ensures that the computation result is part of the last output of the normalized trace, if the program correctly prints it. Without this, the last output might be decoration-only. Given that *validInputs* generates only sequences where no two adjacent values apart from the last two add up to 0, and knowing, due to the definition of run_{rep}, that the number of reads in the trace is exactly the length of the input sequence (see Footnote 3), we then only still need to check that the final output in the normalized trace contains the correct answer.

Note that the aggressive normalization of traces is possible only because all outputs before the last input are completely optional. If we want to check that these outputs contain certain specific information (in a slight variation of the task), we must instead inspect the trace in a step-by-step way similar to what we did in the summation example (some normalization of consecutive outputs can still be helpful, though). Lastly, providing feedback in case the predicate returns *False* is desirable, but will again add significant extra complexity and most likely will have to be tailored to that particular exercise task.

Generating test inputs and checking the correctness of traces often mimics the structure of the task's behavior, i.e., the different conceptual states the behavior induces. By extension, it therefore also mimics the structure of a sample solution. For example, comparing

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

the first *checkCorrectness* to *main*, it is not surprising to see that both first handle a single value and afterwards go through a recursively looping function. By squinting a little bit, we can even see parts of the alternative solution

```
main :: IO_rep ()
main = do n ← readLn
          xs ← replicateM n readLn
          print (sum xs)
```

in the input generator. Keeping this common structure in sync when creating and adjusting exercise tasks is the main challenge of the "manual" approach. But the common structure also enables a different approach. Instead of trying to keep the different testing components in sync, we may specify the desired behavior we want for some task, in an appropriate language, and then the necessary components are derived automatically from the specification.

To summarize, we want to test student submissions to exercise tasks by treating the submitted programs as black-boxes. Similarly to how we would test pure functions with QuickCheck, we do not perform any analysis of the program code, type-checking aside. Instead, we want to simply write a specification of the desired behavior. Moreover, the components needed for testing should then be derived automatically. Included should be a mechanism to provide informative feedback in case the program's behavior differs from the specified behavior. The language of specifications should be expressive enough to capture usual exercise tasks on interactivity, including simple guessing games and other common exercises from Haskell textbooks (Thompson, 2011; Hutton, 2016; Schrijvers, 2023). Also, specifications should facilitate easy and intuitive adaptation of existing tasks. For example, we might want to change the summation in the first example task to a product or change that task to only read positive numbers for summands. In the second example task, we might want to require that the last three, or maybe even all, inputs sum up to 0 instead of the last two. Such simple adaptations should translate to equivalently simple changes to tests or to a specification for the original task. It is certainly not the case for the code in Figure 1.

As another case in point, note that while changing – in the earlier exercise task – the summation to a product computation might seem trivial at first, there exists a subtle issue with products commonly causing overflows of the bounded *Int* type. If the student program and the generator and checking predicate do not either all use *Int* or all use *Integer*, conceptually correct programs might be rejected. Making generation and checking agnostic to the choices of numerical types in the student program, though, again increases complexity and is thus a burden on the test writer. But in our framework we can provide a general solution for such issues (see Section 5.3.1).

3 Specifications

The main goal of the specification language is to describe the behavior that a correct solution for some task should have. We want such descriptions to be concise, intuitive and easily adaptable toward new tasks. The design currently does not include any abstraction

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

facilities or tries to achieve compositionality, since we focus only on specifying small scale exercise tasks. For the same reason, we do not care to capture the full range of I/O behavior, i.e., the behavior of every perceivable console I/O program. A lot of behavior is uninteresting or ill-suited for exercise tasks. For example, we want behavior that actually requires interaction, so that students must make repeated use of the I/O primitives to solve the task.

In order to facilitate these goals, we intentionally restrict the specification language in several ways both in terms of the overall design and also what specifications we consider well-formed. For example, the specification language has only a very restricted form of (global) state to enforce real interactivity of programs (see Section 3.5 for details). Essentially, the only state information accessible inside a specification are the values read in during the interaction so far. We also deliberately rule out general nondeterminism when expressing optionality and restrict the type of iterative behavior that can be expressed, to rule out specifications with certain unproductive cycles. As an additional benefit, such restrictions keep the syntax and semantics of the specification language simple and have the potential to enable additional reasoning about specifications. We will go into detail on these restrictions both in this section as well as in the formal definitions of syntax and semantics in Sections 8.1 and 8.2.

To motivate the design of our small DSL, we will go through the summation example task from Section 2 step by step and see what constructs are necessary to describe the intended behavior formally (Section 3.1). Additionally, we show how to express a very specific form of optional behavior in specifications (Section 3.2). We then explain how a specification is to be interpreted intuitively in terms of execution traces (Section 3.3) and sketch how, given a sequence of inputs, we can derive traces from the specification itself by treating it as a (non-deterministic) program (Section 3.4). Traces obtained this way are, for example, useful when providing students with feedback.

3.1 Describing behavior

Recall the second task description from Section 2:

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

We will now, step by step, construct a specification for this behavior, introducing the necessary specification language constructs as needed. The final specification will be shown in each step, but parts not yet discussed will remain in gray.

First off, since we want to speak about interactive behavior, we need notations for input and output primitives. We use square brackets to describe such atomic actions and distinguish inputs from outputs via a triangle arrow into something or out of something:

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

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

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

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

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

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

The middle part of our example specification should correspond to the reading-in of the n numbers we want to sum. Since n is determined by the first read value, we do not know up front (before a program runs) how many values we need to read overall. Therefore, we need some mechanism for flexible iteration (rather than just some fixed times concatenation of sub-specifications). We mark the part of a specification we would like to iterate with $\rightarrow^{\mathbf{E}}$ and introduce a marker **E** to indicate where/when the iteration process should finish:

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

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

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

Now we can fill in a condition that only when satisfied gives control to the left branch, leading in our case to the termination of the iteration process. Otherwise the right branch will be used. Even though our example here uses only a single exit marker and branching construct, the language can express much richer iteration schemata through the use of multiple exit markers[4] and nested branching.

We can now use branching and iteration to repeatedly read in a value until some condition is fulfilled:

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

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

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

One thing the verbal description states that is not yet present in the DSL expression is the fact that the first number should not be negative. This kind of restriction (in a task) is often useful when we do not care about ill-formed or otherwise undesirable inputs, especially in

[4] For an example of a specification where multiple exit markers arise naturally, see the variation of this example in Figure 9 in Section 8.2.

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

an educational setting where we usually introduce new concepts one step at a time. That is, in the beginning of a course, we might not want students to, for example, have to worry about checking inputs for correctness. But later on we might explicitly require them to do so. Our specification language therefore provides the necessary flexibility to go both ways. Each occurrence of the primitive for reading has to be annotated with the set of values we expect there. Additionally we state for each input action what behavior we expect in case the value is not from the given set. In the default case we do not expect any invalid values at all. That is, the behavior when reading an invalid value is irrelevant. This default decision corresponds to guaranteeing students that inputs will be well-formed. For our example, we get the following specification:

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

In case we want specific behavior upon reading an invalid value, we choose one of two modes of handling such values. Either the program has to stop (in a controlled fashion, not via a runtime error), or the program has to enter a loop, repeatedly reading further candidate inputs until a valid one is encountered, then to continue normally with that valid input. We denote the first case with $[\triangleright \cdot]^{\cdot}_{\top}$ and the second with $[\triangleright \cdot]^{\cdot}_{\circlearrowleft}$. In the second case, invalid values are discarded and are therefore not accessible in later parts of the specification. For the rest of this section, we will always assume valid inputs.

3.2 Optionality

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

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

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

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

While this way of expressing optionality can look a bit cumbersome compared to, for example, simply flagging an output as optional via a dedicated construct, it is far more expressive since the set we can give there is rather arbitrary. For example, we could allow the programs, for whatever reason, to output exactly any multiple of the result of some value computation.[5]

[5] This expressiveness really pays off if we generalize the language to output arbitrary strings, since we can then specify that we allow any output string as long as it contains the required result somewhere.

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

$[\triangleright n]^{\mathbb{N}}$	$([len(x_A) = n_C] \Rightarrow \mathbf{E} \triangle$	$[\{\varepsilon, n_C - len(x_A)\} \triangleright]$	$[\triangleright x]^{\mathbb{Z}}$	$)^{\rightarrow \mathbf{E}}$	$[\{sum(x_A)\} \triangleright]$
$2 \rightarrow n$	$len([\,]) = 2 : False$	choose ε	$5 \rightarrow x$	↻	
	$len([5]) = 2 : False$	choose ε	$3 \rightarrow x$	↻	
	$len([5,3]) = 2 : True$				$8 \in \{sum([5,3])\}$

Fig. 2: Successful matching of trace ?2 ?5 ?3 !8 *stop*.

We deliberately introduce only this specific kind of non-determinism, in outputs, and not, for example, a general non-deterministic choice operator. We have not yet encountered any tasks where such a general choice operator would be needed to encode the desired behavior as a specification, not in any textbook and not in our own teaching. Therefore, it seems adding general non-determinism would only increase the complexity of syntax and semantics without providing any actual benefit.

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

3.3 Valid program runs

Consider now the following trace we might get from a program: ?2 ?5 ?3 !8 *stop*. The program first reads in the numbers 2, 5, and 3, then prints 8 and stops. Does this trace match the specification developed above, i.e., could a program fulfilling the specification have such a run? If not, we have just found evidence that the program under consideration does not fulfill the specification.

We can check the validity of the trace by going from left to right (and possibly in loops) through the specification and seeing if the trace actions match the required actions, while keeping track of the contents of variables. Figure 2 illustrates this process.

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

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

For our current case, we have $[len(x_A) = n_C] \Rightarrow \mathbf{E} \triangle [\{\varepsilon, n_C - len(x_A)\} \triangleright][\triangleright x]^{\mathbb{Z}}$ as the iteration's body. So we have to check ?5 ?3 !8 *stop* against that. We first evaluate the

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

branching condition to determine which sub-specification we have to match against. Since x_A has length 0 at the moment, we choose the right branch, which means we have to check ?5 against $[\,\{\varepsilon, n_C - \mathit{len}(x_A)\} \triangleright\,]$. 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.[6] And indeed the next action required by the specification is $[\,\triangleright x\,]^{\mathbb{Z}}$, which matches ?5 and results in 5 being assigned to x (actually, to be assigned to x_C and appended to x_A). Since we have no specification left to check locally, but are inside an iteration, we again check against the whole iteration body. This results in 3 also being read into x, which now conceptually holds the list $[5, 3]$ (as x_A, with x_C being the 3 from the end of that list). Therefore, in the next round the branching condition evaluates to *True*, thus ending the loop due to the occurrence of **E** in the left branch, i.e., the middle argument of $([\cdot] \Rightarrow \cdot \vartriangle \cdot)$.

All that is left now is to check !8 *stop* against $[\,\{\mathit{sum}(x_A)\} \triangleright\,]$. Since we have $\mathit{sum}(x_A) = \mathit{sum}([5, 3]) = 8$, 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 specification.

3.4 Specifications as (non-deterministic) programs

When we match a trace against a specification, in order to compare the trace's actions with the expectations, we have to keep track of a variable environment and evaluate branching conditions and terms from output actions. This essentially is the equivalent of evaluating the specification itself on the given input sequence. However, in contrast to the evaluation of regular programs, we have to deal with the output actions' (potential) non-determinism, i.e., different possibilities for output values or completely optional outputs. So instead of a single trace, evaluating a specification on an input sequence results in multiple traces, one for each combination of output choices.

The matching procedure described in Section 3.3 can then be restated as checking whether a program's trace is equal to one of the traces resulting from the evaluation of the specification under the same input sequence. The traces a specification evaluates to, for a specific input sequence, differ only in the output values. In the matching example above we showed why ?2 ?5 ?3 !8 *stop* is a valid program run for the summation task. But so is ?2 !2 ?5 !1 ?3 !8 *stop*, i.e., when the program outputs the number of remaining needed inputs for the sum.

Recall that one of our stated requirements is a way to provide feedback in case of a mismatch between a program's behavior and a specification. Both of the above traces could be given as an example of a correct program run. None of them is an obvious best choice. Depending on whether students try to fulfill the optional requirement, either one or the other will be more useful for them to fix their mistake.

We can address this ambiguity by combining both traces into the representation ?2 !$\{\varepsilon, 2\}$?5 !$\{\varepsilon, 1\}$?3 !$\{8\}$ *stop* with sets of possible values for each output. Such a

[6] If both skipping and successfully matching against a non-ε output value is possible, checking trace validity can require backtracking. Section 3.4 hints at an optimization that remedies this problem. Section 8.3 will introduce this optimization in detail.

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

trace also "contains" the two additional possible traces for the input sequence [2, 5, 3]: ?2 !2 ?5 ?3 !8 *stop* and ?2 ?5 !1 ?3 !8 *stop*. We call this type of trace a generalized trace, since it is a generalized description of the behavior given by a specification, for a certain input sequence.

In case of a matching failure, we can provide such generalized traces (in some pretty-printed form) to students. This way we can showcase what are all possible runs a correct program can have on the input sequence for which their program behaved incorrectly. In case the generalized trace contains lots of optional outputs or outputs whose allowed values cannot be printed concisely, showing the complete generalized trace may lead to confusion on the side of the students. But turning the generalized trace back into a single ordinary one for display is always possible, for example, by ignoring all optional outputs and otherwise always choosing the smallest allowed output.

Moreover, we can compare the erroneous program's trace with the correct generalized trace for the respective input sequence to provide useful feedback beyond just displaying correct sample solution runs, namely by pinpointing how the expected and actual program behaviors differ. There are effectively two cases a mismatch can result from. Either the structure of the traces does not line up, e.g., at some point during the comparison one trace begins with an input and the other with an output and the respective step in the generalized trace cannot be skipped (because it is not of the form $!\{\varepsilon, \ldots\}$). Or, for some output step there is a mismatch between the expected and actual output. In both cases, a simple message like "*Expected: . . . , but got: . . .*" or "*Unexpected output value . . . , expected one of . . .*" can be generated and presented to the student, along with the input (sequence) that triggered the error.

To determine the complete set of expected outputs in the presence of optionality, some lookahead into immediately following outputs in the trace might be necessary. This can be avoided by combining consecutive outputs in traces into sequences of values. For example, the two consecutive outputs $!\{\varepsilon, 1\}$ $!\{\varepsilon, 2\}$ would be combined into $!\{\varepsilon, 1, 2, 1.2\}$, where we write 1.2 for a word over $\mathbb{Z}$ to distinguish it from a decimal representation for twelve.[7] A similar normalization of outputs was already present in the "example-specific" checking predicate in Figure 1. Now, the merging of consecutive outputs helps to avoid lookahead or backtracking when checking whether a program trace is covered by a generalized trace. More details on generalized traces are provided in Section 8.3.

3.5 Restrictions on expressiveness

As we already hinted at earlier, the expressiveness of the specification language is restricted at several points. That rules out specifications of certain kinds of behavior, for good or bad. Most notably, we deliberately ruled out general non-determinism, as already explained in Section 3.2. Other restrictions will follow.

When it comes to usefulness in our educational setting, we would, for example, like the specified pattern to enforce actual interactivity. That is, at its core the behavior should rely on a (somewhat alternating) sequence of reads and writes and should not be expressible in a different way. Consider, for example, the following Haskell program:

[7] The lowered dot also clearly separates $\mathbb{Z}$-word concatenation from our notations for explicit trace concatenation ($\cdot$) and integer multiplication ($*$) used later on.

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

```
main :: IO ()                loop :: Int → IO ()
main = do n ← readLn         loop n | n ⩽ 0 = return ()
          loop n             loop n         = print n ≫ loop (n − 1)
```

A specification corresponding to this program is not expressible in our DSL (reading and writing integer values), and that was a design goal. The non-expressibility is due to the facts that in our specifications an iteration process can only end based on some predicate over the global variable state (contents of variables, their history) and that only inputs can alter this state, leaving the above kind of "output-driven loops" impossible to encode. In other words, what is "missing" from the specification language are first-class value-variables and assignment statements that would allow us to create and manipulate arbitrary (global) state. More generally, specifications lack the ability to describe sequences of consecutive output actions whose length is dynamically determined. According to our motivation, this restriction is a good thing. We only want inherently interactive behavior to be expressible, whereas the above program can be rewritten as

```
main :: IO ()                loop :: Int → String
main = do n ← readLn         loop n | n ⩽ 0 = ""
          print (loop n)     loop n         = show n ++ "\n" ++ loop (n − 1)
```

with exactly one input action at the beginning, then all computation happening in a non-I/O loop[8], and exactly one output action at the end. Overall this is not an attractive teaching example when we actually want to cover interactive I/O in Haskell and how programs must be structured to organize sequences of input and output actions in interesting ways.

If we for a moment would lift our restriction to just use integers as inputs and outputs, we could write a specification like $[\triangleright n]^{\mathbb{Z}}[\{loop(n)\}\triangleright]$ for behavior as above, with the second version of *loop* above. From this it is immediately clear that the interactive core of the program/task here is almost trivial, so we do not want it. Put differently, we wanted to make sure that there are as few as possible ways in our DSL to encode essentially non-interactive computations in only seemingly interactive guise. Note that even if we do indeed allow strings for input and output, as we do for practical usage in our course, we can still prevent creation of such "boring tasks" via the DSL by controlling which functions are allowed in terms for conditions and outputs, preventing, for example, something like *loop* from appearing there as it does in the hypothetical specification $[\triangleright n]^{\mathbb{Z}}[\{loop(n)\}\triangleright]$. We will later, in the formal definition in Section 8.1, see that this is encoded in the definition of syntactically correct terms by parameterizing it over some set of available functions.

4 Challenges of finding good input sequences

So far we showed how to specify console I/O behavior and how to check whether the execution trace of a program matches the described behavior. But what input sequences should be used to produce the execution traces? Take the original example

[8] The *loop*'s structure can be viewed as a co-program producing a sequence of outputs (Gibbons, 2021), i.e., a co-recursive case distinction between ending and continuing the sequence. Iteration in our DSL cannot be used to issue output actions in a way that encodes such co-programs producing sequences of outputs with dynamic length.

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

from Section 3: $[\triangleright\, n\,]^{\mathbb{N}}([len(x_A) = n_C] \Rightarrow \mathbf{E} \triangle [\triangleright\, x\,]^{\mathbb{Z}})^{\rightarrow \mathbf{E}}[\,\{sum(x_A)\} \triangleright\,]$. This specification requires an input sequence starting with a non-negative integer and then as many arbitrary integers. It is unlikely that a completely random input sequence will have this property. In Section 2, we already saw how to write a hand-crafted input generator for this example. A naive solution for framework-provided instead of user-supplied input generation interleaves generation of random values with evaluation of the specification's behavior. At each input action $[\triangleright \cdot]^{\tau}$ we can generate a random value $v \in \tau$ and store it in a variable environment. For $[\triangleright \cdot]^{\tau}_{\top}$ and $[\triangleright \cdot]^{\tau}_{\circlearrowleft}$ we can generate either a value $v \in \tau$ or a value $v \notin \tau$, then proceed appropriately. Generally, we go through the specification similarly to how we did when matching against a trace. If we reach the end of the specification, the sequence of generated values is an input sequence suitable for testing. This is our previous approach as described in our first presentation of the specification language (Westphal & Voigtländer, 2020*a*, TFPIE'19). With a carefully chosen value range, the naive approach yields acceptable results for certain specifications with behavior that straightforwardly guarantees the termination of the generation process, like the one above.

However, the generated sequences can become very long. In the above summation example, the first generated natural number determines the length of the input sequence. Long sequences are not necessarily bad, but we would like to establish confidence that a program behaves correctly on input sequences of any length. Generating only longer input sequences can even hurt our ability to find mistakes. Take as an example the second task from Section 2 (this time without the optional outputs):

"Write a program that reads in integers until the two most recently entered integers sum up to zero. Then output the number of integers the program has read and stop."

This task can be specified as $[\triangleright\, x\,]^{\mathbb{Z}}([\triangleright\, x\,]^{\mathbb{Z}}[{x_C}^{-1} + x_C = 0] \Rightarrow \mathbf{E} \triangle \mathbf{0})^{\rightarrow \mathbf{E}}[\,\{len(x_A)\} \triangleright\,]$. Here ${x_C}^{-1}$ is short for the penultimate element of x_A, i.e., the value of x_C prior to the last input action on x. This is not an extension of the language itself, as ${x_C}^{-1} = last(init(x_A))$. If we run the naive generation approach on this specification and we draw values from $\{x \in \mathbb{Z} \mid |x| < n\} \subseteq \mathbb{Z}$, the expected length of an input sequence is $2n + 2$. Now, let us assume a student program for the stated task counts but otherwise ignores its first input, which may result in an execution trace like ?1 ?−1 ?1 !3. Such a mistake will not be detected if the program is not tested on small input sequences. After all, input sequences for the task's specification with length > 2 will never contain a pattern that triggers this error, because the first two values in these sequences will never sum up to zero. (Otherwise already these two-element *prefixes* would be the relevant sequences being looked at.) But the chance of never encountering a two-element input sequence is higher than 10% when testing the program on 100 sequences naively generated from the above specification while drawing values from the range $[-25\,..\,25]$.

Controlling the size of randomized inputs is not a problem unique to testing I/O programs. When using random inputs to test programs, the first random input that witnesses a property violation is usually not the most succinct one, if any is found at all. Therefore, most property-based testing frameworks employ techniques to shrink found counterexamples in an attempt to find the minimal input needed to reproduce an encountered problem (Pike, 2014; Claessen, 2012; de Vries, 2023). Some frameworks do not use shrinking and

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

instead systematically and exhaustively test the input domain up to some cutoff on the input's size (Runciman *et al*., 2008; Duregård *et al*., 2012).

We will employ a somewhat hybrid approach between exhaustive testing and random inputs with shrinking. We explore input sequences exhaustively with regard to their length but per realizable length we only test a few randomized sequences. Additionally, we do not attempt to shrink the counterexamples themselves. In case we find a counterexample, we narrow the search space going forward, to further include only input sequences shorter than the counterexample already found, and keep testing. In the end, we report the shortest counterexample found.

To implement this approach, we need a reliable way to exhaustively explore realizable (lengths of) input sequences for the behavior given by some specification expression. That is, we want to be able to systematically control the length of generated input sequences and to explore as many different combinations of the specification's branching choices as possible.[9] And in contrast to hand-crafted input generators or naive framework-provided input generation, the new approach should work for all specifications, yet not require any analysis of specifications by the user or overly careful choices of ranges for random values. Indeed, it should also work well if, for example, choosing certain input values can make it impossible to reach the end of the specification. We present a constraint-solving-based approach fulfilling these requirements in the next section and tie it all together into a fully formed testing procedure in Section 6.[10]

5 Input sequence generation through constraint solving

Our goal is to systematically search for input sequences that result in terminating traces, for a given specification. Additionally, we want to be able to control exactly what branching choices are taken in the specification per input sequence. To this end, we construct paths of symbolic constraints over input sequences. We call them specification paths. If such a path is satisfiable, finding a concrete input sequence for it then yields an input sequence that we can use for testing. Unsatisfiable paths correspond to impossible combinations of branching choices and are therefore not relevant for testing programs.

This is the essential idea behind test case generation through symbolic execution (King, 1976; Cadar & Sen, 2013). Usually, it is the tested program itself that is symbolically executed into constraints. We perform symbolic execution on specifications instead, since we choose to only look at the traces a student program produces and do not use its actual program code. In this section, we will first introduce specification paths in detail (Section 5.1). We then show how to systematically explore a specification's paths (Section 5.2). Finally, we describe how we use the Z3 SMT solver (de Moura & Bjørner, 2008) to determine satisfiability of specification paths and generate input sequences from satisfiable paths (Section 5.3).

[9] Note that branching conditions under an iteration induce a new independent choice in every cycle of the iteration.

[10] The naive generation method is still available in the implementation, as it can be significantly faster for specifications without very specific termination conditions.

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

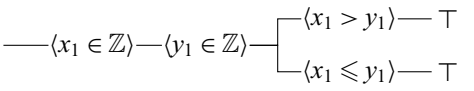
5.1 Specification paths

Specification paths are constructed by traversing the specification in the same way as when comparing against traces. But instead of working with concrete values we only gather up constraints on symbolic variables. Since we have no concrete values at hand we cannot evaluate branching conditions. We therefore simply choose one of the branches and record the necessary constraint to choose that branch on the path.

Take, for example, $[\triangleright x]^{\mathbb{Z}}[\triangleright y]^{\mathbb{Z}}([x_C > y_C] \Rightarrow [\{2 * x_C\} \triangleright] \triangle [\{3 * y_C\} \triangleright])$. This specification has two paths $\langle x_1 \in \mathbb{Z}\rangle\langle y_1 \in \mathbb{Z}\rangle\langle x_1 > y_1\rangle\top$ and $\langle x_1 \in \mathbb{Z}\rangle\langle y_1 \in \mathbb{Z}\rangle\langle x_1 \leqslant y_1\rangle\top$.

Constraints of the form $\langle x_i \in \tau\rangle$ introduce a new input value from set τ, specifically the ith value read into variable x. These constraints originate from the input actions of the specification. A constraint like $\langle x_1 > y_1\rangle$ states that the first value read into x should be greater than the first value read into y. Such constraints encode which branching choices are made on this path. The $\top$ symbol represents termination, i.e., reaching the end of the specification. Note that the output actions are unimportant for the constraints.

Both paths for $[\triangleright x]^{\mathbb{Z}}[\triangleright y]^{\mathbb{Z}}([x_C > y_C] \Rightarrow [\{2 * x_C\} \triangleright] \triangle [\{3 * y_C\} \triangleright])$ share a common prefix and differ only in the last constraint, i.e., in the decision which branch to take. This is not surprising, since we always explore the paths of a specification starting from the first action. We will therefore represent the paths of a specification as a tree. In this case the tree is:



To fully automate input generation, we use a constraint solver to search for input sequences that satisfy the constraints of a given path. For this to work, we need functions used in conditions to be expressible in the language of the constraint solver. In Section 3.5, we already mentioned parameterizing specifications over the set of available functions to guarantee interesting behavior. The same approach also works for guaranteeing specifications produce only constraints expressible in the solver's language. But this time we are interested in restricting available functions in branching conditions instead of output actions. We will go into more detail on how we use constraint solving in practice in a moment. First, we will look at some examples of how the structure of a specification and its branching conditions affect the shape, and especially the size, of the path tree.

5.2 Infinite path trees

Input sequences derived from terminating specification paths are guaranteed to produce well-formed, i.e., terminating traces for the underlying specification. However, this does not mean that we can always fully explore programs with regard to the specified behavior, since in general specifications can have infinitely many paths. This happens when a specification contains a "read until valid" input action $[\triangleright \cdot]^{\circlearrowleft}$ or an iteration.

As an example of the iteration case, let us consider the "sum up to zero"-example from Section 4 again: $[\triangleright x]^{\mathbb{Z}}([\triangleright x]^{\mathbb{Z}}[x_C{}^{-1} + x_C = 0] \Rightarrow \mathbf{E} \triangle \mathbf{0})^{\rightarrow \mathbf{E}}[\{len(x_A)\} \triangleright]$. Recall that $x_C{}^{-1}$ refers to the penultimate element of x_A. For this specification we have a path

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

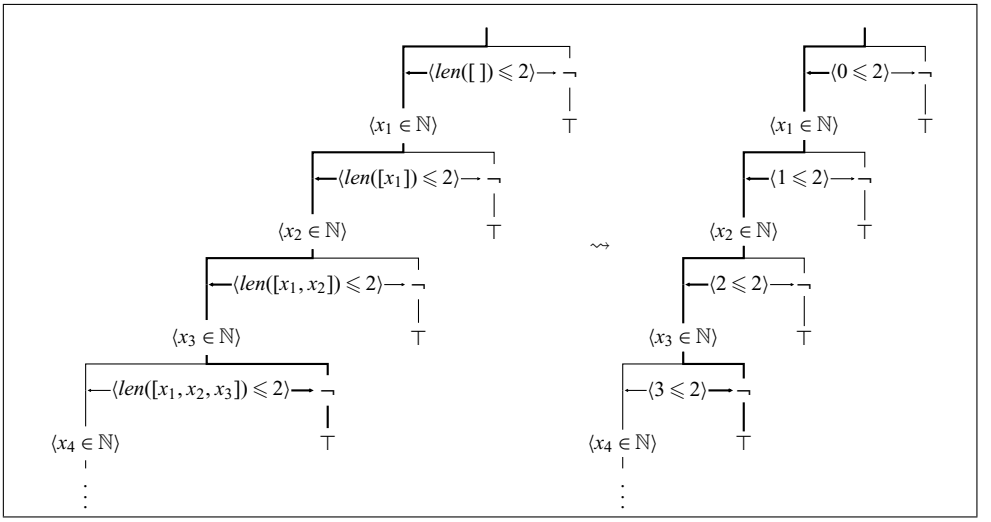


Fig. 3: Single satisfiable path (with *len* evaluated on the right).

$\langle x_1 \in \mathbb{Z}\rangle\langle x_2 \in \mathbb{Z}\rangle\langle x_1 + x_2 = 0\rangle\top$ describing immediate termination of the iteration. But of course there is also $\langle x_1 \in \mathbb{Z}\rangle\langle x_2 \in \mathbb{Z}\rangle\langle x_1 + x_2 \neq 0\rangle\langle x_3 \in \mathbb{Z}\rangle\langle x_2 + x_3 = 0\rangle\top$, i.e., the iteration stopping after reading one additional value into x. Continuing this pattern, we can always extend a path by one additional input, resulting in infinitely many paths:

$$\langle x_1 \in \mathbb{Z}\rangle\langle x_2 \in \mathbb{Z}\rangle \begin{cases} \langle x_1 + x_2 = 0\rangle\top \\ \langle x_1 + x_2 \neq 0\rangle\langle x_3 \in \mathbb{Z}\rangle \begin{cases} \langle x_2 + x_3 = 0\rangle\top \\ \langle x_2 + x_3 \neq 0\rangle\langle x_4 \in \mathbb{Z}\rangle\cdots \end{cases} \end{cases}$$

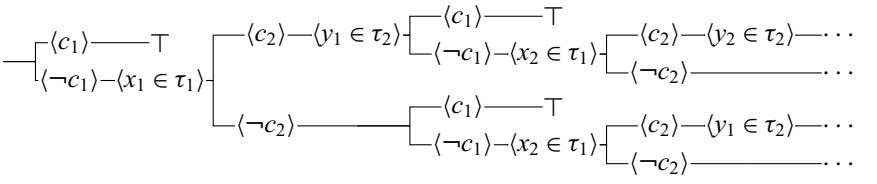
We impose an upper limit on the number of input constraints in paths to test, since we cannot test all possible paths for such a specification in finite time. But up to such a limit, we can now systematically search for all satisfiable paths of specific lengths and generate (multiple) input sequences for each such path.

Up until now, all paths in our examples were satisfiable. For more complex specifications this does not need to be the case. There are even situations where we have an infinite number of paths for a specification, but only one of them is satisfiable. That is, the specification requires very specific input sequences, all of the same fixed length. An example for such a specification would be $([\mathit{len}(x_A) \leqslant 2] \Rightarrow [\triangleright x]^{\mathbb{N}} \triangle \mathbf{E})^{\rightarrow \mathbf{E}}$. The paths for this specification are shown in Figure 3. The only satisfiable path $\langle \mathit{len}([\,]) \leqslant 2\rangle\langle x_1 \in \mathbb{N}\rangle\langle \mathit{len}([x_1]) \leqslant 2\rangle\langle x_2 \in \mathbb{N}\rangle\langle \mathit{len}([x_1, x_2]) \leqslant 2\rangle\langle x_3 \in \mathbb{N}\rangle\langle \neg(\mathit{len}([x_1, x_2, x_3]) \leqslant 2)\rangle\top$ is highlighted there. On this path the length of x_A increases by one each iteration cycle until the branching condition becomes true. All other paths are clearly unsatisfiable, but since paths are constructed purely symbolically they are nonetheless part of the tree. Note that we symbolically evaluated each occurrence of x_A in the specification's branching conditions to a list of input variables. We can evaluate the conditions even further, as already partially done in Figure 3, since the *len*-function does not require any knowledge about the list's elements'

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

actual values. The result would be "constant" constraints $\langle true \rangle$ or $\langle false \rangle$. But to emphasize where each constraint originates from, we do not fully do this simplification in the figure.

In general, path trees can have both infinitely many satisfiable and infinitely many unsatisfiable paths. This is especially the case if we have a tree that grows exponentially in the number of iteration rounds. For example, for specifications of the form $(([c_1] \Rightarrow \mathbf{E} \triangle \mathbf{0})[\triangleright x]^{\tau_1}[c_2] \Rightarrow [\triangleright y]^{\tau_2} \triangle \mathbf{0})^{\to \mathbf{E}}$ the path tree has the following shape:



Here, searching for satisfiable paths (up to a certain length of input sequences) can become expensive quickly. In general, due to the varying shape of the path tree and its ratio of satisfiable to unsatisfiable paths, the efficiency and effectiveness of our input generation method depends on the strategy we use to search the path tree for satisfiable paths. For example, identifying unsatisfiable subtrees early and pruning them from the search space can reduce the runtime significantly for exponential path trees with many times more unsatisfiable paths than satisfiable ones. See Section 7.3 for details on this and also the limitations of our implementation.

5.3 On solving specification paths

We use the Z3 SMT solver (de Moura & Bjørner, 2008) to find input sequences for satisfiable specification paths. Given a specification path, we construct a corresponding SMT formula and ask the solver to search for a model for that formula. For this to work, the functions used in branching conditions and the sets we draw inputs from need to be translatable into some logic supported by the SMT solver. At the moment we use QF_LIA, quantifier-free formulas over linear integer constraints, as our target logic. Even though this is a fairly restricting choice, all of the exercises we usually give to students are expressible within this theory, and so is the majority of programs and exercise tasks from popular textbooks (Thompson, 2011; Hutton, 2016; Schrijvers, 2023). This is in part because in many cases functions that fall outside of linear integer arithmetic are only used in output actions where they do not affect input generation.

Note that during translation, lists of symbolic values are translated to (in)equality constraints between expressions over input variables. Assume, for example, that we have a specification which describes reading in a natural number n and then repeatedly reading integers until the sum of these integers exceeds n. A path for such a specification is $\langle n_1 \in \mathbb{N} \rangle \langle \neg(sum([\,]) > n_1) \rangle \langle x_1 \in \mathbb{Z} \rangle \langle \neg(sum([x_1]) > n_1) \rangle \langle x_2 \in \mathbb{Z} \rangle \langle \neg(sum([x_1, x_2]) > n_1) \rangle \langle x_3 \in \mathbb{Z} \rangle \langle sum([x_1, x_2, x_3]) > n_1 \rangle \top$ and it is translated to the formula

$$(n_1 \geqslant 0) \wedge (0 \leqslant n_1) \wedge (x_1 \leqslant n_1) \wedge (x_1 + x_2 \leqslant n_1) \wedge (x_1 + x_2 + x_3 > n_1)$$

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

Checking for satisfiability of this constraint then gives us a model from which we then derive an input sequence that will test the specification path, in this case the sequence [0, 0, 0, 1].

Ideally, we want multiple input sequences for a single path to test the program with. This is especially important since the paths describe the behavior from the perspective of the specification and a program can potentially realize one specification path's behavior through more than one execution path. Unfortunately, simply running the solver multiple times on the same query will most likely result in the same model every time, since the solver usually works deterministically. To work around this, we encode paths as MaxSAT optimization problems (Bjørner *et al*., 2015). That is, we add additional constraints to the formula that the solver is allowed to ignore when searching for a model. In a MaxSAT problem the goal is to find a model that fulfills the maximum number of these soft constraints. We use these extra constraints to encode randomized suggestions for the different values of the input sequence. For our current example we, might suggest the input sequence [3, 7, 15, −4]:

$$(n_1 \geqslant 0) \wedge (0 \leqslant n_1) \wedge (x_1 \leqslant n_1) \wedge (x_1 + x_2 \leqslant n_1) \wedge (x_1 + x_2 + x_3 > n_1)$$
$$\wedge (n_1 = 3) \wedge (x_1 = 7) \wedge (x_2 = 15) \wedge (x_3 = -4)$$

The soft constraints we add, shown in gray, are always of the form $x_i = c_i$, where c_i is a concrete value randomly drawn from the set of allowed inputs for x_i. They encode a suggested input sequence and the solver is asked to find the closest input sequence satisfying the path constraint, with distance measured by the number of differences between the two sequences. Here, the solver would return the sequence [22, 7, 15, 1], satisfying two of the soft constraints. This is a common approach when sampling solutions of SAT and SMT problems (Dutra *et al*., 2018). Each new run for the same specification path is done with different random value suggestions.

5.3.1 Additional constraints

Sometimes straightforward solving of paths can be insufficient. Consider our running example, the summation of n numbers, but now we require the program to output the product instead of the sum: $[\triangleright n]^{\mathbb{N}}([len(x_A) = n_C] \Rightarrow \mathbf{E} \triangle [\triangleright x]^{\mathbb{Z}})^{\rightarrow \mathbf{E}}[\{product(x_A)\} \triangleright]$. For this specification, even relatively short input sequences can make $product(x_A)$ exceed the bounds of Haskell's *Int* type. If students would now write an otherwise correct program using the *Int* type instead of the unbounded *Integer* type, our testing framework would most likely reject it. When matching a program trace against a specification, we carry along a variable environment under which branching conditions and expected outputs are evaluated. This environment always stores unbounded *Integer* values to keep in line with the semantics of specifications that have no concept of bounded-size machine integers. In case of an overflow of the *Int* type, the corresponding output step of the program's trace and the evaluation of the expected outputs will most likely differ and cause a behavior mismatch.

In the past, we have given students a task based on this specification and in fact some students wrote programs that only worked as long as the *Int* type did not overflow. But since already a product of 10 relatively small numbers can cause an *Int*-overflow, these programs got rejected by the testing framework. In some contexts this might be the appropriate

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

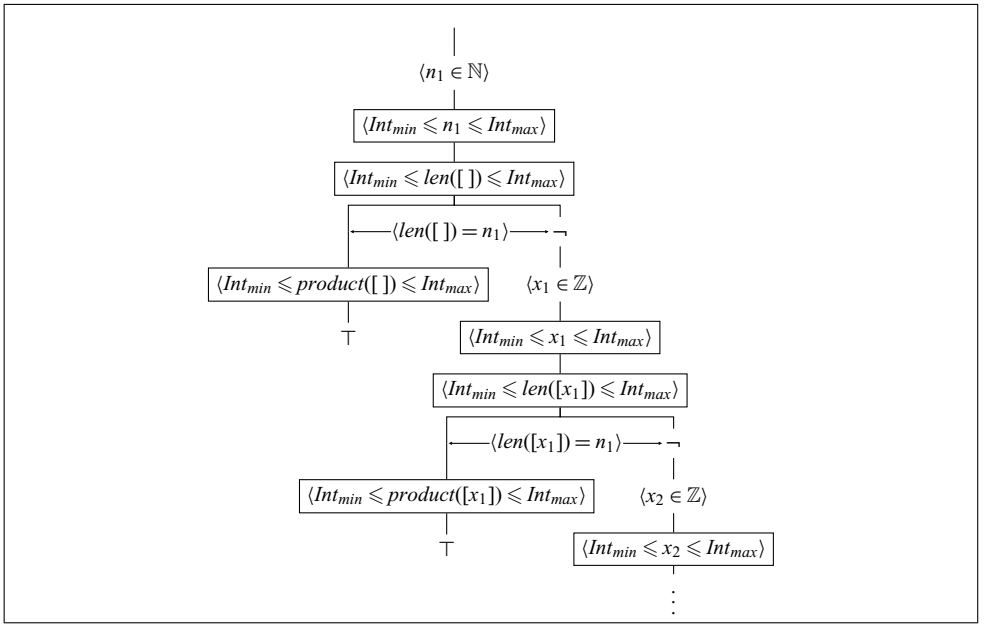


Fig. 4: Additional constraints to avoid *Int*-overflows.

outcome. But when the goal is to teach simple (recursive) I/O programs, such technical details get in the way of the desired educational goal. We cannot, though, simply require students to only use unbounded *Integer*s as some standard library functions just return *Int*s, most notably the ubiquitous function $length :: [a] \to Int$. Depending on the context, we therefore might require the solver to consider additional constraints when checking the satisfiability of paths. In the above example, we would like to make sure that ideally no (sub-)computation in conditions and outputs exceeds the bounds of the *Int* type. For every computation representable inside the solver we can insert respective constraints into the paths, without changing the structure of the specification's path tree. Figure 4 shows the modified paths for the product example. Here Int_{min} and Int_{max} are the lower and upper bound of the *Int* type. Note how the output actions now also contribute constraints to the paths.

By design, (sub-)terms in outputs do not have to be completely representable in the language of constraints. Our implementation provides an option to generate such additional constraints on a best-effort basis, thus reducing the possibilities of unwanted testing failures due to overflows. Additionally, we also produce a warning during the actual testing on input sequences whenever a computation, from the specification, would exceed the limits of the *Int* type. So even if the additional constraints were not able to prevent an overflow, students or lecturers are still alerted and can interpret the test's result accordingly.

6 Testing procedure

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

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

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

We described how to match program traces against specifications and how to provide feedback in Sections 3.3 and 3.4. Section 5 described our approach to input generation. We now test programs by repeatedly running the following steps:

1. Search for a satisfiable path for the given specification s,
2. find a valid input sequence for this path using random suggestions for each value in the sequence,
3. run the program under testing on that input sequence, resulting in trace t_p,
4. check whether t_p is a valid run for s.

The testing procedure accepts the program if we have successfully tested n input sequences for each satisfiable path up to length l_{max}, with the length of a path being the number of its input constraints. If we instead find a counterexample, i.e., an input sequence for which the program produces a trace that is not a valid run for s, we record the length of this sequence and continue testing on paths shorter than the counterexample's length only. This process continues until we have successfully tested n input sequences for each satisfiable path shorter than the last found counterexample, which we then report. Increasing n and l_{max} increases the probability that the program actually has the specified behavior in case we do not find a counterexample. Increasing l_{max} can, however, cause the number of tests performed to grow exponentially (see Section 7.3).

7 Implementation

We have built an EDSL for the designed language in Haskell and implemented the testing approach explained thus far.[11] An interactive sandbox for playing around with the implementation on various examples is available at `https://iotasks.fmi.uni-due.de/`.

Within the framework, we provide a data type for describing a *Specification*, the IO_{rep} type from Section 2, and a function $taskCheck :: IO_{rep}\ () \to Specification \to IO\ ()$ that tests a program against a specification according to the procedure described in Section 6, reporting the shortest counterexample found.

7.1 Example usage

To see the testing framework in action we once again use the summation example task from Section 2:

> *"Read a positive number n from* `stdin`*, then read n additional numbers and print the sum of those n numbers to* `stdout`*."*[12]

[11] The library containing the EDSL is available at `https://github.com/fmidue/IOTasks`.
[12] For a clearer presentation we here exclude the case of $n = 0$.

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

We specify this behavior as $[\triangleright\, n\,]^{\mathbb{N}^+}([len(x_A) = n_C] \Rightarrow \mathbf{E} \triangle [\triangleright\, x\,]^{\mathbb{Z}})^{\rightarrow \mathbf{E}}[\,\{sum(x_A)\} \triangleright\,]$ and a possible solution we presented in Section 2 is

```
program :: IO_rep ()
program = do n ← readLn
             loop n 0

loop :: Integer → Integer → IO_rep ()
loop 0 res = print res
loop n res = do x ← readLn
                loop (n − 1) (x + res)
```

Now how can we use our EDSL in order to check the correctness of this solution?

Since our main goal was to have everything needed for testing to be derived from the specification of the desired behavior, all we need to do is express the behavior's specification in our EDSL:

```
specification :: Specification
specification =
  readInput n pos AssumeValid <>                          -- [▷ n ]^ℕ⁺
  tillExit (                                              -- (
  branch (length* (allValues x) ·==· currentValue n) --     [len(x_A) = n_C] ⇒
    exit                                              --       E △
    (readInput x ints AssumeValid)                    --       [▷ x ]^ℤ
  ) <>                                                    -- )^→E
  writeOutput [resultOf (sum* $ allValues x)]             -- [ {sum(x_A)} ▷ ]
  where
    n, x :: Var Integer
    n = intVar "n"
    x = intVar "x"
pos, ints :: ValueSet Integer
pos = greaterThan 0
ints = complete
```

For every construct in our specification language there is a corresponding constructor in our EDSL which creates that particular kind of specification, possibly from smaller specifications.

```
readInput    :: Var a → ValueSet a → InputMode → Specification
writeOutput  :: [OutputPattern] → Specification
branch       :: Term Bool → Specification → Specification → Specification
tillExit     :: Specification → Specification
exit         :: Specification
nop          :: Specification
```

Since the *Specification* type forms a *Monoid* with regard to sequential composition (and neutral element **0**, now written *nop*), we use (<>) to glue multiple specifications together.

Note that *Var* and *ValueSet* are polymorphic over the type of values they stand for or talk about. This is due to the fact that the implementation is not restricted to *Integer* inputs, but also supports *String*-valued variables. Similarly, instead of simply a list of *Term*s, the *writeOutput*-function takes an [*OutputPattern*] argument, allowing for more flexible

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

outputs than simply printing (integer) results of computations. *OutputPattern*s are built from the primitives

$$\begin{aligned}&\mathit{resultOf} :: \mathit{Show}\ a \Rightarrow \mathit{Term}\ a \to \mathit{OutputPattern}\\&\mathit{text} :: \mathit{String} \to \mathit{OutputPattern}\\&\mathit{wildcard} :: \mathit{OutputPattern}\end{aligned}$$

and can be combined through a *Monoid* instance (with neutral element *text* `""`). For example, in practice we often want to allow results to be printed with some optional embellishments. The output pattern $\mathit{wildcard} \diamond \mathit{resultOf}\ (\mathit{sum}^\star\ \$\ \mathit{allValues}\ x) \diamond \mathit{wildcard}$ could then be used in the above specification to accept programs that use something more descriptive like

$$\mathit{putStrLn}\ \$\ \texttt{"The sum of all inputs is "} +\!\!+ \mathit{res} +\!\!+ \texttt{"."}$$

instead of the simple *print res* call in the sample solution further above.

When it comes to values of *Term* types, for output and branching, we construct them from syntactically inspectable versions of some standard *Prelude* functions.

$$\begin{aligned}&\textbf{data}\ \mathit{Term}\ a \quad \text{-- abstract}\\&(\cdot{=}{=}\cdot) :: \mathit{Eq}\ a \Rightarrow \mathit{Term}\ a \to \mathit{Term}\ a \to \mathit{Term}\ \mathit{Bool}\\&(\cdot{>}\cdot) :: \mathit{Ord}\ a \Rightarrow \mathit{Term}\ a \to \mathit{Term}\ a \to \mathit{Term}\ \mathit{Bool}\\&(\cdot{<}\cdot) :: \mathit{Ord}\ a \Rightarrow \mathit{Term}\ a \to \mathit{Term}\ a \to \mathit{Term}\ \mathit{Bool}\\&(\cdot{+}\cdot) :: \mathit{Term}\ \mathit{Integer} \to \mathit{Term}\ \mathit{Integer} \to \mathit{Term}\ \mathit{Integer}\\&(\cdot\&\&\cdot) :: \mathit{Term}\ \mathit{Bool} \to \mathit{Term}\ \mathit{Bool} \to \mathit{Term}\ \mathit{Bool}\\&\mathit{not}^\star :: \mathit{Term}\ \mathit{Bool} \to \mathit{Term}\ \mathit{Bool}\\&\mathit{length}^\star :: \mathit{Term}\ [a] \to \mathit{Term}\ \mathit{Integer}\\&\mathit{sum}^\star :: \mathit{Term}\ [\mathit{Integer}] \to \mathit{Term}\ \mathit{Integer}\\&\mathit{product}^\star :: \mathit{Term}\ [\mathit{Integer}] \to \mathit{Term}\ \mathit{Integer}\\&\ldots\end{aligned}$$

The special term constructors *currentValue* and *allValues* are used to access specification variables, corresponding to the subscript notation $_C$ and $_A$.

$$\begin{aligned}&\mathit{currentValue} :: \mathit{Var}\ a \to \mathit{Term}\ a\\&\mathit{allValues} :: \mathit{Var}\ a \to \mathit{Term}\ [a]\end{aligned}$$

We cannot build terms from arbitrary Haskell functions in general, as we need to translate terms into constraints that we can then hand to the SMT solver. For terms used in outputs this restriction can be loosened, though. Apart from checking for overflows, as described in Section 5.3.1, we do not need to translate output terms into SMT-expressible constraints. The implementation therefore allows the use of opaque Haskell functions in outputs, but for simplicity we disregard this detail here.

Now that we have written the specification, we can check whether the program has the desired behavior simply by running the $\mathit{taskCheck} :: \mathit{IO}_{\mathit{rep}}\ () \to \mathit{Specification} \to \mathit{IO}\ ()$ function:

```
> taskCheck program specification
generated 125 input sequences covering 25 satisfiable paths
+++ OK, passed 125 tests.
```

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

By default, we explore all paths that unfold iterations at most 25 times (globally). Therefore, we find 25 satisfiable paths, namely for $n \in \{1, \ldots, 25\}$, i.e., with 2 to 26 inputs in length. The number of different sequences tested per satisfiable path defaults to 5, so we end up with 125 tests in total.

What happens if a program does not have the specified behavior? For example, what if the program we are testing reads one value less than it should:

$$
\begin{array}{ll}
\mathit{wrong1} :: \mathit{IO}_{\mathit{rep}}\ () & \mathit{loop} :: \mathit{Integer} \to \mathit{Integer} \to \mathit{IO}_{\mathit{rep}}\ () \\
\mathit{wrong1} = \mathbf{do}\ n \leftarrow \mathit{readLn} & \mathit{loop}\ 0\ \mathit{res} = \mathit{print}\ \mathit{res} \\
\qquad \mathit{loop}\ (n-1)\ 0 & \mathit{loop}\ n\ \mathit{res} = \mathbf{do}\ x \leftarrow \mathit{readLn} \\
 & \qquad \mathit{loop}\ (n-1)\ (x + \mathit{res})
\end{array}
$$

In such a case, we get an error message like this:

```
> taskCheck wrong1 specification
generated 1 input sequence covering 1 satisfiable path
*** Failure
Input sequence: ?1 ?57
Expected run: ?1 ?57 !57 stop
Actual run: ?1 !0 stop
Error:
  AlignmentMismatch:
    Expected:
      ?57
    Got:
      !0
```

When checking whether the generalized trace covers the program trace here, we get stuck when checking the program's !0 *stop* against the expected ?57 !{57} *stop*. Such an alignment mismatch, as already mentioned in Section 3.4, is one of two possible error causes when comparing ordinary and generalized traces in our setting. The other possible source of a mismatch (not an alignment issue) manifests when the program writes an output that is not part of the set of valid outputs at the respective position, i.e., the actual output is not covered by the expected outputs. For this example, that could be the case because a program does not include the first read number into the summation:

$$
\begin{array}{ll}
\mathit{wrong2} :: \mathit{IO}_{\mathit{rep}}\ () & \mathit{loop} :: \mathit{Integer} \to \mathit{Integer} \to \mathit{IO}_{\mathit{rep}}\ () \\
\mathit{wrong2} = \mathbf{do}\ n \leftarrow \mathit{readLn} & \mathit{loop}\ 0\ \mathit{res} = \mathit{print}\ \mathit{res} \\
\qquad x \leftarrow \mathit{readLn} & \mathit{loop}\ n\ \mathit{res} = \mathbf{do}\ x \leftarrow \mathit{readLn} \\
\qquad \mathit{loop}\ (n-1)\ 0 & \qquad \mathit{loop}\ (n-1)\ (x + \mathit{res})
\end{array}
$$

For such a program the error message looks like this:

```
> taskCheck wrong2 specification
generated 1 input sequence covering 1 satisfiable path
*** Failure
Input sequence: ?1 ?37
```

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

```
Expected run: ?1 ?37 !37 stop
Actual run: ?1 ?37 !0 stop
Error:
  OutputMismatch:
    !0 is not covered by !37
```

7.1.1 Counterexamples with naive input generation

The implementation also includes the naive input generation method as an opt-in fallback. As discussed in Section 4, this input generation method only works for a small set of specifications. And even then, the problems with coverage heavily impact the quality of error messages. On average, input sequences provided in error messages are significantly longer. In the summation example, the termination condition is simple enough that we can use naive input generation. Doing so, for the *wrong1* program, we get an error message like this:

```
> Naive.taskCheckWith stdArgs{inputRange = 15} wrong1 specification
*** Failure
Input sequence: ?9 ?-12 ?12 ?12 ?8 ?2 ?5 ?9 ?-6 ?13
Expected run: ?9 ?-12 ?12 ?12 ?8 ?2 ?5 ?9 ?-6 ?13 !43 stop
Actual run: ?9 ?-12 ?12 ?12 ?8 ?2 ?5 ?9 ?-6 !30 stop
Error:
  AlignmentMismatch:
    Expected:
      ?13
    Got:
      !30
```

And for *wrong2*:

```
> Naive.taskCheckWith stdArgs{inputRange = 15} wrong2 specification
*** Failure
Input sequence: ?11 ?8 ?-11 ?-15 ?-2 ?2 ?12 ?15 ?4 ?15 ?-7 ?1
Expected run: ?11 ?8 ?-11 ?-15 ?-2 ?2 ?12 ?15 ?4 ?15 ?-7 ?1 !22 stop
Actual run: ?11 ?8 ?-11 ?-15 ?-2 ?2 ?12 ?15 ?4 ?15 ?-7 ?1 !14 stop
Error:
  OutputMismatch:
    !14 is not covered by !22
```

Notice how we restricted the *inputRange* parameter to 15. This restricts the absolute value of randomly generated integers. We will therefore only draw values from $\{-15, \ldots, 15\}$, or $\{1, \ldots, 15\}$ for the first input. The default *inputRange* used for the constraint-based examples is 100.

Without the *inputRange* restriction the counterexamples generated with the naive method can become very large here as the length of the input sequence depends on the first input value. Alternatively, we could have changed the specification itself, to directly

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

restrict the set of allowed values for the first input. Either way, some ad-hoc restriction is necessary when we use naive generation and thus eschew exploiting path coverage information. There is also no immediately obvious way to shrink a counterexample into a smaller one, as it is not clear which input controls the length of the overall sequence from the error-producing sequence alone. We could try to brute-force a smaller sequence that also triggers the error, by trying to shrink individual values, but this is clearly infeasible in general. The constraint-based generation approach, with its knowledge of paths, can systematically search for errors on shorter paths and thereby on shorter input sequences, reducing the size of counterexamples considerably and reliably even when inputs are drawn from larger value ranges.

7.1.2 Derived combinators

Since specifications are embedded as ordinary Haskell values, one can use the full power of Haskell to build more complex combinators from the basic specification constructors. This allows for higher-level abstractions and consequently more succinctly written specifications. For example, the implementation provides a combinator to capture the structure of what is conceptually a while loop in our summation specification:

```
while :: Term Bool → Specification → Specification
while c body = tillExit (branch c body exit)
```

With this newly defined combinator, the complete specification becomes shorter and more declarative:

```
specification :: Specification
specification =
  readInput n pos AssumeValid <>
  while (not* $ length* (allValues x) ·==· currentValue n)
    (readInput x ints AssumeValid) <>
  writeOutput [resultOf (sum* $ allValues x)]
  where
    n = intVar "n"
    x = intVar "x"
```

Moreover, the implementation's version of *while* is additionally enriched with useful sanity checks. For example, we check that there is no top-level exit marker in the loop body, and that at least one input action in the body modifies one of the variables in the condition. Such checks then help to not write ill-formed or otherwise bad specifications.

7.2 Expressiveness

So far, we have seen only relatively simple examples of behavior. However, specifications can express much richer behavior. Starting from the simplest behavior, we can express things like reading some value and then passing that value to some function, printing back the result. A specification for such behavior consists of a single input action followed by an output action. Even though this is not particularly interesting behavior, in the form of

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

a greeter program where the user is asked to enter their name and then is greeted with a custom message this interaction pattern occurs in multiple textbooks (Thompson, 2011; Schrijvers, 2023).

Moving up the complexity ladder, we have behaviors that require some form of repeated actions until a certain condition is met. The simplest condition just checks whether the last read value fulfills some predicate. Examples are reading values until the read value is 0 or until a certain string input like `"stop"` is read. Some behaviors require checking more than just a single value. For example, we already saw a specification waiting for the last two values read to sum up to 0. As long as we guarantee that enough values have been read, we can express conditions not only on the last two values but on any fixed number of recent inputs. We can also use a condition over all values read into some variable, like in our summation running example (where the condition is length-related) or in the example from Section 5.3. In fact, the summation task is the single most common task we found in textbooks.

I/O behavior is all about interactivity, so it is not surprising that sooner or later one wants to implement simple interactive games. Our framework can capture common simple games found in textbooks. Since string inputs are only supported in a very rudimentary fashion, we usually need to reframe these games in terms of reading integers instead of more complicated data like strings or game moves (but see Footnote 15).

The easiest games are simple single player guessing games. One example is a game where the user has to guess a hidden number by repeatedly trying numbers and getting feedback as to whether the hidden number is higher or lower (Schrijvers, 2023). For these single player games, the initial setup of the game is not part of the specification itself. The (randomized) selection of the hidden number is outside of the scope of the specification language. Therefore, we end up with an EDSL expression that is parameterized over the hidden number.

```
specification :: Integer → Specification
specification target =
  ...
  readInput guess ints AssumeValid <>
  branch (currentValue guess ·==· intLit target)
  ...
```

Consequently, the program tested against such a specification must have the corresponding type *main* :: *Integer* → *IO* () and testing is then done like this:

let *target* = ... **in** *taskCheck* (*main target*) (*specification target*)

We might need to test a program with multiple (random) initializations to make sure it has the desired behavior.

A slightly more involved game is a variant of the classic hangman game, where the goal is to guess a word of length n by guessing a single letter in each try until the complete word is revealed (a string-based variant is used by Hutton (2016)). Again, as our framework deals mainly with integers, the most straightforward specification for a hangman game also requires guessing a sequence of numbers instead of a dictionary word (for example, trying to guess the digits of an unknown prime number that the game host is revealing

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

piece by piece). The win condition for this game is more complicated than for the high-low guessing game. We need to check that all numbers in the target sequence are contained in the guessed values. This condition depends on the target sequence, so we define a function to construct the win condition from an integer sequence.

```
hangmanSpec :: [Integer] → Specification
hangmanSpec target =
  ...
  branch (winCond $ allValues guess)
  ...
  where
    winCond :: Term [Integer] → Term Bool
    winCond guesses = foldr (λa b → (intLit a `elem*` guesses) ·&&· b) true target
```

Here $elem^{\star} :: Term\ Integer \rightarrow Term\ [Integer] \rightarrow Term\ Bool$ is the lifted membership test on lists.[13]

Finally, there are games that are usually played with two players but often feature a computer opponent in the context of programming exercises. Tasks for these games usually have a pure component that implements a strategy for the computer opponent and an interactive part for the main game loop. The pure part of the game logic can be tested with regular property-based tests. Again, the win condition needs to be expressed in the EDSL's term language and the game state must be computable from the inputs alone. This means that the computer opponent's strategy must be deterministic.

A basic example is the rock-paper-scissors game, where the game loop consists of playing multiple rounds against a certain computer strategy with the overall winner determined by who won the most rounds (Thompson, 2011).[14] The fourth and final game is a number picking game where players take turns picking numbers from 1 to 9, each number can be picked only once, and whoever picked three numbers that add up to 15 wins. This game is equivalent to a game of Tic-tac-toe on a 3×3 grid but without the need to handle positional moves.[15]

All of the above examples and some variations can be found in our example collection at `https://github.com/fmidue/IOTasks-collection` (see also Figure 5) and the smaller examples are also available as templates in the interactive demo (`https://iotasks.fmi.uni-due.de/`).

7.3 Practical limitations of input generation

Most of the concrete examples we presented so far contain only the simplest form of iteration expressible with our specifications (all but the sketches of games): a single iteration with exactly one branching construct and therefore two paths inside its body, one leading

[13] List membership is translated to a disjunction of equalities when creating constraints for the SMT solver.

[14] The example presented by Thompson models multiple different opponent strategies, including one that uses *unsafePerformIO* to implement a completely random strategy. For the reasons stated in the previous paragraph this non-deterministic strategy is not compatible with our framework.

[15] We have actually also used a task that from the students' perspective looked exactly like real Tic-tac-toe, including string input/output of positional moves, but internally used a numeric encoding hidden by clever name-shadowing and pretty-printing.

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

Task	Origin	Restrictions	Variations
Check whether user input is a palindrome.	**T**	–	Compute some other function on user input.
Greet user with their entered name.	**T**, **S**	–	repeating user input (multiple times)
Read two numbers and output their sum. Repeat until the first entered number is 0, then print the number of performed additions.	**O**	–	no negative numbers allowed (on negative numbers, repeat reading until non-negative number is entered)
Read natural number, then that many additional integers and output their sum.	**T**, **H**, **S**	–	Compute product, sort, etc. instead of sum; With extra prompts; Read values until 0 is entered instead of reading the number of values upfront.
High-low guessing game	**S**	one specification per secret	–
Hangman	**H**	guess individual letters (digits) instead of whole words; one specification per different secret needed	–
Rock-paper-scissors	**T**	game moves encoded as numbers; one specification per strategy needed	–
Number scrabble	**O**	–	Tic-tac-toe; against CPU; with two human players

Origins **T**: (Thompson, 2011); **H**: (Hutton, 2016); **S**: (Schrijvers, 2023); **O**: our own course material

Fig. 5: Some expressible exercise tasks.

to termination and one to continuing the iteration. This structure ensures that the number of paths inside the iteration is linear with regard to the maximum length of the input sequence, since only one of the two paths, after a fork, forks again. However, as already shown in Section 5.2, in general the tree of specification paths can be up to exponentially large in the maximum desired input length. The specifications for simple games often exhibit this problematic structure. For example, for hangman, there is usually a reaction of the program telling the user whether a guess was correct or not, introducing two different paths through the iteration encoding the game loop. In many cases, large parts of such trees can consist solely of unsatisfiable paths. The implementation, therefore, regularly tests prefixes of paths for satisfiability to potentially prune subtrees of unsatisfiable paths early.

Pruning is most effective when unsatisfiability can be determined from a short common prefix of some paths. For example, a win or termination condition becoming irrefutable after a certain number of read inputs will result in prunable subtrees. Pruning is therefore often effective in reducing the amount of paths to explore and in fact was introduced

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

precisely to make testing games like hangman more efficient. We have observed speed-up between 1.5× and 4× when searching for satisfiable paths of suitable specifications, for example, hangman games with shorter secrets. On specifications where unsatisfiable paths do not have short unsatisfiable prefixes in common, the pruning checks only incur a small extra runtime cost of around 10–20%.

But for some specifications pruning is not possible at all, as all paths are in fact satisfiable. Since we only start looking for prunable subtrees after finding the first unsatisfiable path, pruning has no performance penalty on such specifications. A simple specification for which all paths are satisfiable is the following one:

$$[\triangleright x]^{\mathbb{Z}}([sum(x_A) > 0] \Rightarrow \mathbf{E} \triangle ([\triangleright x]^{\mathbb{Z}}[x_C > 0] \Rightarrow [\,\{1\} \triangleright\,] \triangle [\,\{0\} \triangleright\,]))^{\rightarrow \mathbf{E}}$$

Here the nested branching constructs result in two paths that both continue the iteration. Therefore, the overall number of paths for this specification is $2^{l_{max}} - 1$, where l_{max} is the maximum length of input sequences. Our current implementation can handle such specifications only for relatively small numbers of iteration unfoldings.

7.4 Specification interpreter

In addition to the core functionality of testing programs against specifications, the implementation also provides means to run specifications as if they were programs. The most straightforward way is a function $runSpecification :: Specification \rightarrow [String] \rightarrow Trace$, producing a (generalized) trace given a specification and an input sequence. Such a function is already part of the core framework's internals, as it is exactly what we use to find the generalized trace for a specific input sequence (see Figure 11 in Section 8.3.1).

Additionally, we also provide a function $interpret :: Specification \rightarrow [IO\,()]$ that turns a specification into actual I/O programs. This function resolves the potential nondeterminism of output actions by returning a list of all meaningful variants of resolving non-deterministic outputs. Every such variant results from first choosing a specific combination of output options, i.e., reducing every set of outputs to some singleton set, and then interpreting the specification.

Interpretation is an important tool when designing a new specification. Through running the specification itself on different input sequences, we can make sure that a specification matches our mental model of what we want to encode. In case of exercise task design, this also means checking the verbal task description against the specification used for testing solution candidates.

Additionally, with the ability to interpret specifications, we can also test the implementation of the testing framework itself, to some degree. As interpreting specifications as programs is independent, in terms of the implementation, from testing programs against specifications, we can programmatically interpret a specification and check the resulting program against that specification. This raises confidence in the internal consistency of the implementation. Moreover, via this connection, correctness arguments, e.g., by code inspection, in one part also bridge and carry over to other parts to some extent. We have presented a detailed exploration of this idea in a previous article (Westphal & Voigtländer, 2020*b*, FLOPS'20).

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

$$\frac{\tau \subseteq \mathbb{Z} \quad x \in Var}{[\triangleright x]^{\tau} \in Spec}\ (\text{Input1}) \qquad \frac{\tau \subseteq \mathbb{Z} \quad x \in Var \quad m \in \{\top, \circlearrowleft\}}{[\triangleright x]^{\tau}_{m} \in Spec}\ (\text{Input2})$$

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

$$\frac{s_1 \in Spec \quad s_2 \in Spec \quad c \in T_{\mathbb{B}}}{[c] \Rightarrow s_1 \triangle s_2 \in Spec}\ (\text{Branch})$$

$$\frac{\circ \notin effects(s) \quad s \in Spec}{s^{\rightarrow \mathbf{E}} \in Spec}\ (\text{Till-}\mathbf{E}) \qquad \frac{}{\mathbf{E} \in Spec}\ (\text{LoopExit}) \qquad \frac{}{\mathbf{0} \in Spec}\ (\text{Nop})$$

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

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

Fig. 6: Syntax of specifications (top) and terms (bottom).

8 Formal definitions

After presenting the basic concepts and ideas of our approach informally, we will now give formal definitions for the specification language and everything else needed for testing.

We start by defining the syntax of specifications (Section 8.1). We then give a precise semantics of when a trace is accepted by a specification and a corresponding notion of program correctness (Section 8.2). Next, we define generalized traces and what it means for a generalized trace to cover a program trace (Section 8.3). Then, we modify the acceptance criterion to compute generalized traces from a specification by essentially evaluating a specification on an input sequence (Section 8.3.1). We define another modification of the acceptance criterion that, given a specification, computes the tree of all specification paths for input generation (Section 8.4). Finally, we relate the producers of generalized traces and of specification paths, and the testing procedure using them, to our original notions of trace acceptance and program correctness, by giving a lemma and a conjecture that together ensure agreement on which programs adhere to a specification's behavior and which do not (Section 8.5).

8.1 Syntax

As described in Section 3, specifications are essentially built from primitives for specifying input and output actions, together with a branching and an iteration construct. Figure 6 gives the full syntax of our language by defining the set *Spec* of all specifications as well as the term language used for the description of output values and branching conditions.

We distinguish different subsets of the set of all terms by a subscript indicating the type of value a term evaluates to. For example, $T_{\mathbb{Z}}$ denotes the set of all terms that evaluate to an integer and $T_{\mathbb{B}}$ the set of terms evaluating to a Boolean value. We write $[\mathbb{Z}]$ instead of $\mathbb{Z}^*$

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

$$effects([\,\Theta \triangleright\,]) = effects(\mathbf{0}) = \{\circ\}$$
$$effects([\,\triangleright\, x\,]^{\tau}) = effects([\,\triangleright\, x\,]^{\tau}_{\top}) = effects([\,\triangleright\, x\,]^{\tau}_{\circlearrowleft}) = \{\uparrow\}$$
$$effects(\mathbf{E}) = \{\downarrow\}$$
$$effects(s_1 \cdot s_2) = \bigcup_{X \in effects(s_1)} \begin{cases} \{X \sqcup Y \mid Y \in effects(s_2)\} & \text{if } X = \circ \vee X = \uparrow \\ \{X\} & \text{if } X = \downarrow \vee X = \updownarrow \end{cases}$$
$$effects([c] \Rightarrow s_1 \triangle s_2) = effects(s_1) \cup effects(s_2)$$
$$effects(s^{\rightarrow \mathbf{E}}) = \bigcup_{X \in effects(s)} \begin{cases} \varnothing & \text{if } X = \uparrow \\ \{\circ\} & \text{if } X = \downarrow \\ \{\uparrow\} & \text{if } X = \updownarrow \end{cases}$$

Note that no case $X = \circ$ exists in the last equation, since we inductively already know that for any $s^{\rightarrow \mathbf{E}}$ in the syntax it holds $\circ \notin effects(s)$.

Fig. 7: Loop body effects.

for sequences of integers here, emphasizing that we are dealing with list values as opposed to words over integers.

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

In the case of iterations, we require guaranteed *progress* in the sense that each path through the iteration body reaches an exit marker or contains at least one input action, i.e., the path alters the global variable state. To check this requirement, the function *effects* defined in Figure 7 computes abstractions of the different possible effects an iteration body can have. There are two separate effects we are interested in: firstly, reading something into some variable at least once, and secondly, finishing the iteration. A single path can therefore have four different possible combinations of presence/absence of these effects: $\circ$, representing paths that neither read any input nor terminate the iteration; $\uparrow$ and $\downarrow$ for either paths that read or ones that terminate; and $\updownarrow$ for paths that both read and terminate the iteration. We use $\sqcup$ for the obvious join operation on this four element lattice. Depending on the branching structure a (sub-)specification can feature any collection from those four effect abstractions. The *effects*-function computes the set of (combined) effects for all paths of the given specification. For example, the set $\{\circ, \updownarrow\}$ encodes that a specification has at least one path that neither reads nor terminates the iteration, at least one path that both reads and terminates, and no paths with any other combination of effects.

The condition $\circ \notin effects(s)$ in the (Till-**E**) rule therefore exactly describes our desired progress condition: every path through a loop must terminate the iteration ($\downarrow$ or $\updownarrow$) or read at least one new value ($\uparrow$ or $\updownarrow$). Note that progress of iterations in general does not imply termination. For example, $([sum(x_A) < 100] \Rightarrow [\,\triangleright\, x\,]^{\mathbb{Z}} \triangle \mathbf{E})^{\rightarrow \mathbf{E}}$ has the progress property

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

but a program with this behavior does not always terminate (and yet, programs can be effectively tested against this specification, see below). On the other hand, a specification of definitely terminating behavior will always also have an equivalent variant that satisfies the progress condition.

As a consequence of requiring progress, building specification paths as introduced in Section 5 will never result in an infinite path with only a finite number of input constraints. This is because for a path that leads to repeating the iteration's body its combined effect is exactly ↑ (since ∘ is outlawed in the (Till-**E**) rule and ↓ or ↕ would imply the path is not one of those leading to repetition). So every repetition of the iteration's body will add at least one additional input constraint. This is an important property as it enables us to find all specification paths up to a specified cutoff length (see Section 8.4 for details). A similar argument also guarantees termination of the semantic functions for trace acceptance and for the computation of generalized traces, as they will consume finite traces and finite input sequences, respectively, in lockstep with specification input actions (see Sections 8.2 and 8.3.1).

The syntactic definition of specifications is parameterized over a not further specified set *Func* of functions and some variable set *Var*. In principle, we could choose any set of functions we want, as long as both concrete and symbolic evaluation of terms is well-defined. The formalization does not require or assume any other property of *Func*. Strictly speaking, we require symbolic evaluation for only terms used in branching conditions as only those are used to create path constraints.

In addition to the syntactic definition of specification expressions, we make the following assumptions regarding structure and semantic well-formedness. A specification $s \in \mathit{Spec}$ is called *well-formed* exactly if it has the following properties:

1. A variable x_C does not occur in a term before x occurred in an input action, since this would make the evaluation of that term fail. A corresponding issue does not exist for x_A since we can define it to initially evaluate to the empty list.
2. The exit marker **E** never occurs outside of an iteration, i.e., never at the top-level of a specification expression.
3. Every loop eventually terminates, i.e., it reaches an occurrence of **E** (given the right sequence of input values). If we are not interested in termination in such a strict fashion, we can alternatively loosen the requirement so that for every loop $(s')^{\rightarrow \mathbf{E}}$ inside s there has to exist an exit marker somewhere in s' and outside another iteration, but we do not analyze the branching conditions to reach it.

As a side note, we observe that both the second and the looser version of the third condition can be expressed in terms of *effects* as "$\mathit{effects}(s) \subseteq \{\circ, \uparrow\}$", and "for every loop $(s')^{\rightarrow \mathbf{E}}$ inside s, it holds $\mathit{effects}(s') \neq \{\uparrow\}$", respectively. After some consideration, we might even go further and require that the *effects* set of every loop body must consist exactly of ↑, one of ↓ and ↕ or both, and no ∘ (the latter of course already known from the (Till-**E**) rule).

The purpose of the third property above, in its strict form, is to let specifications only ever describe finite behavior. In practice, the three properties do not necessarily have to be checked statically. When creating new specifications for exercise tasks, we cross-validate the specification against other artifacts (see Section 7.4 and (Westphal & Voigtländer,

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

2020*b*, FLOPS'20)). So even if we do not check the properties above explicitly, cross-validation usually imposes enough constraints such that accidentally violating one of the properties becomes unlikely.

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

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

8.2 Semantics

We give the semantics of our specification language by defining which program traces are instances of the described behavior. That is, which traces are accepted by a given specification.

In Section 2, we gave a data type for representing such traces and also informally introduced a compact notation for traces specialized to reading and writing integer values. A trace then 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*. We use *Tr* to denote the set of all traces (regardless of a certain program or specification). We mostly denote traces simply as *stop*-terminated sequences without any explicit concatenation symbol, but for visual clarity we sometimes write $x \cdot t$ to describe a trace's structure, where x is a trace prefix, i.e., a trace without final *stop*, and $t \in Tr$.

In order to determine whether a given trace is valid for a given specification, we introduce a function *accept* such that $accept(s, k_I)(t, \Delta_I) = True$ exactly if a given trace $t \in Tr$ exhibits behavior specified by $s \in Spec$. Figure 8 gives the definition of this function. Other than the trace and the specification that the trace is checked against, the function also takes a variable environment Δ and a continuation function k as additional inputs (with k_I and Δ_I being the initial continuation and empty variable environment). The continuation k takes care of managing the current iteration context, as informally described when discussing how to check an iteration in Section 3.3. It encodes how to proceed if we `Exit` from the current context to an outer one or if we just `End` a round inside the current context and continue with another round of the iteration. The functions *eval* and *store* evaluate terms and store values in the environment, respectively. Their definitions are straightforward and are therefore omitted here. We write $eval(\Theta, \Delta)$ for evaluating, under Δ, every term in a set Θ.

At its core, *accept* traverses a specification from left to right, consuming matching trace elements and updating variables along the way. If we are left with exactly the empty specification or if we encounter an exit marker of some iteration, we call the current continuation k with the appropriate argument to indicate whether we want to continue the iteration process or exit from it, and pass the remaining trace and current variable environment along. Note that this also covers the case where we completely consumed the specification in the outermost context, i.e., the initial one. If the trace is then also fully consumed already, the

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

$$accept([\triangleright x]^{\tau} \cdot s', k)(t, \Delta) = \begin{cases} accept(s', k)(t', store(x, v, \Delta)) & \text{, if } t = ?v \cdot t' \wedge v \in \tau \\ False & \text{, otherwise} \end{cases} \tag{8.1}$$

$$accept([\triangleright x]^{\tau}_{\top} \cdot s', k)(t, \Delta) = \begin{cases} accept(s', k)(t', store(x, v, \Delta)) & \text{, if } t = ?v \cdot t' \wedge v \in \tau \\ True & \text{, if } t = ?v\ stop \wedge v \notin \tau \\ False & \text{, otherwise} \end{cases} \tag{8.2}$$

$$accept([\triangleright x]^{\tau}_{\circlearrowleft} \cdot s', k)(t, \Delta) = \begin{cases} accept(s', k)(t', store(x, v, \Delta)) & \text{, if } t = ?v \cdot t' \wedge v \in \tau \\ accept([\triangleright x]^{\tau}_{\circlearrowleft} \cdot s', k)(t', \Delta) & \text{, if } t = ?v \cdot t' \wedge v \notin \tau \\ False & \text{, otherwise} \end{cases} \tag{8.3}$$

$$accept([\,(\Theta \setminus \{\varepsilon\}) \triangleright\,] \cdot s', k)(t, \Delta) = \begin{cases} accept([\,(\Theta \setminus \{\varepsilon\}) \triangleright\,] \cdot s', k)(t, \Delta) & \text{, if } \varepsilon \in \Theta \\ \quad \vee\ accept(s', k)(t, \Delta) & \\ accept(s', k)(t', \Delta) & \text{, if } \varepsilon \notin \Theta \wedge t = !o \cdot t' \\ & \quad \wedge o \in eval(\Theta, \Delta) \\ False & \text{, otherwise} \end{cases} \tag{8.4}$$

$$accept(([c] \Rightarrow s_1 \triangle s_2) \cdot s', k)(t, \Delta) = \begin{cases} accept(s_1 \cdot s', k)(t, \Delta)\text{ , if } eval(c, \Delta) = True \\ accept(s_2 \cdot s', k)(t, \Delta)\text{ , otherwise} \end{cases} \tag{8.5}$$

$$accept(s^{\rightarrow \mathbf{E}} \cdot s', k)(t, \Delta) = accept(s, k')(t, \Delta) \quad \text{with } k'(cont) = \begin{cases} accept(s, k')\text{ , if } cont = \texttt{End} \\ accept(s', k)\text{ , if } cont = \texttt{Exit} \end{cases} \tag{8.6}$$

$$accept(\mathbf{E} \cdot s', k)(t, \Delta) = k(\texttt{Exit})(t, \Delta) \tag{8.7}$$

$$accept(\mathbf{0}, k)(t, \Delta) = k(\texttt{End})(t, \Delta) \tag{8.8}$$

$$k_I(cont)(t, \Delta) = \begin{cases} True & \text{, if } cont = \texttt{End} \wedge t = stop \\ False & \text{, if } cont = \texttt{End} \wedge t \neq stop \\ \texttt{error} & \text{, if } cont = \texttt{Exit} \end{cases}$$

Fig. 8: Trace acceptance.

acceptance match is successful. The initial continuation k_I is defined such that in the case of `End` it performs exactly this check (see Figure 8) and therefore finishes the computation.

Note that specifications that violate one of the first two of the three well-formedness properties from the previous subsection would lead to errors when evaluating *accept*. Namely, this happens if in equations (8.4) or (8.5) we evaluate x_C before any $[\triangleright x]$ or $[\triangleright x]_{\top/\circlearrowleft}$ occurred, and if we encounter an **E**-marker at the top-level.

It might seem strange not to mention the third property, on termination, here, as one could expect that a non-terminating specification might also cause *accept* to fail to terminate. But the progress condition $\circ \notin effects(s)$ we enforce on iteration bodies makes sure that even for an infinite iteration process we consume at least one input from the trace we are matching against during each round of such an iteration (except if the match exits with a negative result, i.e., *False* or `error`). Thereby, in the absence of errors caused by *eval*, the *accept*-function is guaranteed to terminate by reaching the end/exit of the specification or of the given trace. Note that for just ensuring termination of *accept*, the progress condition is even stricter than necessary, as an input or non-optional output action or **E** in each iteration round would suffice.

It is important to note that the equations (8.1) to (8.7) are all defined with the associativity of sequential composition in mind. We do not have a case for something like $accept((s \cdot s') \cdot s'', k)(t, \Delta)$ since this can always be rewritten as $accept(s \cdot (s' \cdot s''), k)(t, \Delta)$. Moreover, the fact that **0** is the neutral element of sequential composition means that the

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

rules can also be applied to specifications like $[\triangleright x]^{\tau}$ by expanding them to $[\triangleright x]^{\tau} \cdot \mathbf{0}$. By the same reasoning, we do not need an explicit rule like $accept(\mathbf{0} \cdot s', k)(t, \Delta) = accept(s', k)(t, \Delta)$. And crucially, equation (8.8) is only applicable if the specification is exactly $\mathbf{0}$. Otherwise this would allow for the initiation of another round of iteration at any point in the specification, since, for example, $s \cdot s'$ can always be rewritten as $s \cdot \mathbf{0} \cdot s'$.

Note that for $\mathbf{E}$ in equation (8.7) the situation is different than that. Even though the pattern $\mathbf{E} \cdot s'$ in it might seem strange at first, since one would probably never write a specification containing "dead code" s' like this, cases exist where a specification of the form $\mathbf{E} \cdot s'$ does arise. Consider, for example, $(([c_1] \Rightarrow ([c_2] \Rightarrow \mathbf{E} \triangle s_1) \triangle s_2) \cdot s_3)^{\rightarrow \mathbf{E}}$, which is a perfectly reasonable specification to write. Now in order to leave the loop via that $\mathbf{E}$, both conditions must evaluate to *True*, and by equation (8.5) we are now left with matching against $\mathbf{E} \cdot s_3$. So in order to correctly handle such specifications, equation (8.7) needs to discard everything following an occurrence of $\mathbf{E}$.

See Figure 9 for a detailed step-by-step application of the *accept*-function on a variation of the "read a natural number and then as many integers"-specification.

8.2.1 Program correctness

Using the *accept*-function, we can now formulate a notion of program (trace) correctness. First, we define the set of all traces that are accepted by specification s as $Tr^s = \{t \in Tr \mid accept(s, k_I)(t, \Delta_I) = True\}$, where Δ_I is the initial environment containing no values, i.e., $eval(x_A, \Delta_I)$ evaluates to the empty list for every variable x occurring in s, and k_I is the initial continuation as shown in Figure 8. Next, we derive from this set the set of relevant input sequences $inputs(Tr^s)$ with $inputs : Tr \to [\mathbb{Z}]$ mapping a trace to its sequence of values marked as inputs. Note that *inputs* implicitly induces a partition of Tr^s into pairwise disjoint sets (called "clusters" henceforth) of s-accepted traces with identical input sequences.

Definition (Program correctness)**.** *Treating a program as a (potentially) partial function* $prog : [\mathbb{Z}] \rightharpoonup Tr$ *with* $inputs \circ prog \subseteq id_{[\mathbb{Z}]}$*, it is considered to have the behavior given by specification* s *if and only if: For every* $vs \in inputs(Tr^s)$*, it holds that* $prog(vs) \in Tr^s$*.*[16]

That is, we consider a program correct with regard to a specification s if on every input sequence that specification defines as relevant, the program does give an execution trace t and it holds that $accept(s, k_I)(t, \Delta_I) = True$. Programs that do not read all provided inputs or do attempt to read more than the given inputs, fall into the partiality case, i.e., their trace (on a relevant input sequence) becomes undefined and they therefore cannot be correct (see Footnote 3).

8.3 Generalized traces

The semantics of specifications and the correctness of programs are both defined through the *accept*-function. But the actual testing procedure does not use *accept* directly,

[16] Of course, $prog(vs)$ will then fall into the one "cluster" from the partition mentioned above whose elements all have vs as input sequence.

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

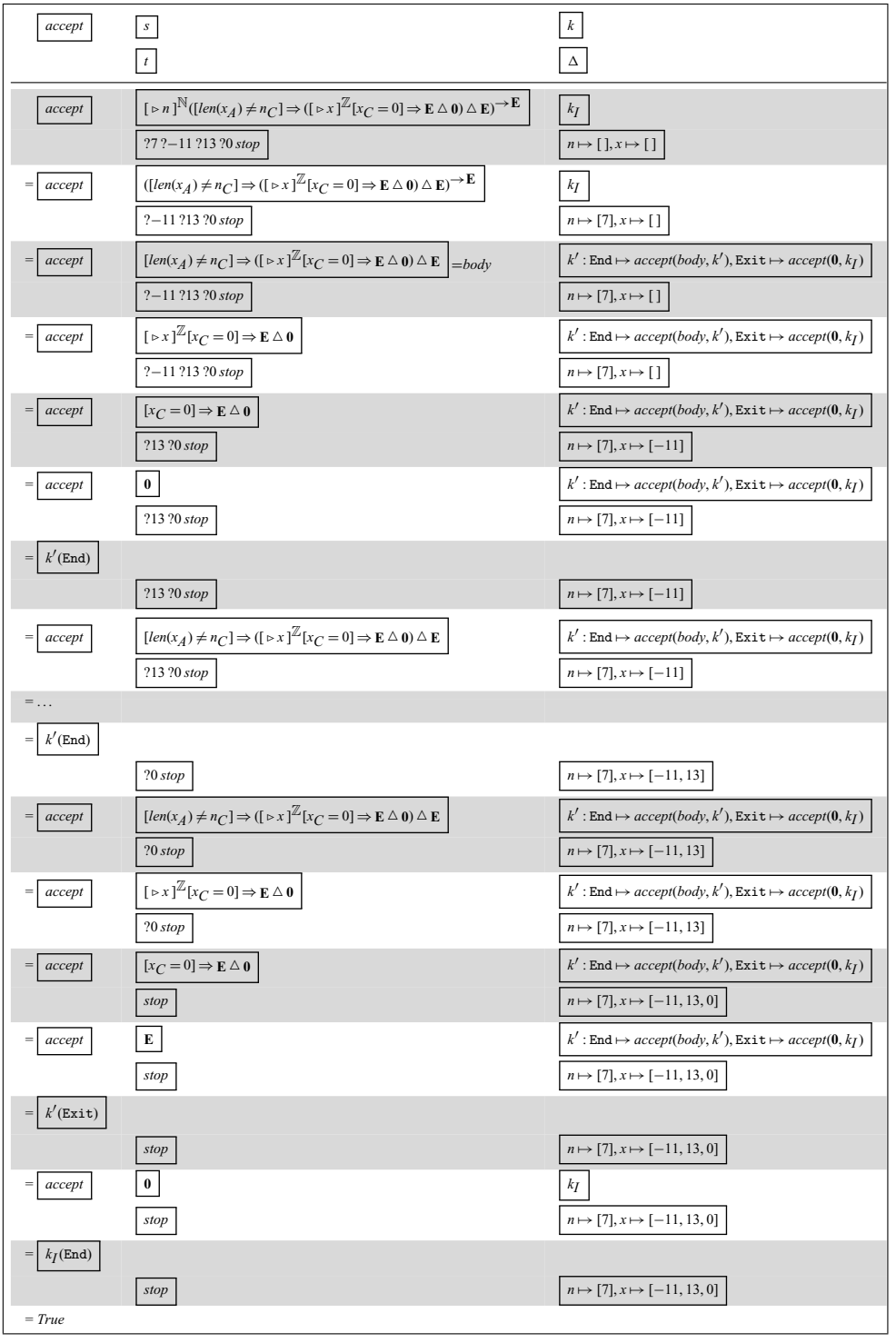


Fig. 9: Acceptance example
(Read natural number and then as many integers but stop on reading 0).

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

since for practical purposes it is much more convenient to compute generalized traces, as introduced in Section 3.4, from specifications and to have a notion of coverage relating program traces and generalized traces. To see this, consider the specification $[\triangleright\, x\,]^{\mathbb{Z}}[\,\{\varepsilon, x_C\}\triangleright\,][\,\{\varepsilon, x_C\}\triangleright\,][\,\{x_C\}\triangleright\,]$ and the matching trace ?1 !1 !1 *stop*. Each time we encounter $[\,\{\varepsilon, x_C\}\triangleright\,]$ when matching the trace against the specification, we need to decide whether to "consume" the next !1 output or to choose to skip it. It turns out that with the *accept*-function we cannot simply match the trace against the specification in a single left to right traversal. No matter whether we prioritize evaluating the left or the right side of the disjunction in the first case of equation (8.4), i.e., choosing x_C or ε, we always have to backtrack at the end and change one of our choices. For longer traces and more complicated specifications, it becomes ever more likely that this backtracking causes a runtime blowup of the matching procedure.

We can avoid this problem by not matching program traces against specifications directly. Instead, we match traces against other traces that we derive from the specification. Such a derived generalized trace represents exactly one "cluster" from the partition that *inputs* induces on Tr^s.

Analogously to how we allowed different potential output values in a single output action in the specification language, we now allow different values in each output step of a (generalized) trace. Moreover, and unlike for specifications, we fuse sequences of adjacent output steps into a single output step, then containing (possibly various) sequences of values.

Before we look at how to compute generalized traces from specifications, we will first define what generalized traces are exactly and how they relate to ordinary traces.

The set of all *generalized* traces (regardless of a certain program or specification), Tr_G, is defined by the following rules:

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

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

Instead of a single value o as in an ordinary trace, an output step in a generalized trace consists of a *set* of *sequences* of values. The sequences encode the fusing of consecutive output steps and the sets encode non-determinism, i.e., the output of one of the sequences from the given set. These sets are not allowed to be empty or singleton sets containing only ε. Similar to the discussion on $[\,\{\}\triangleright\,]$ and $[\,\{\varepsilon\}\triangleright\,]$, the form $!\{\}$ is not realizable by any program in a meaningful way and $!\{\varepsilon\}$ is always skipped. Additionally, the two separate rules for $!O$ ensure that generalized traces do not have consecutive output actions. This corresponds to always fusing as many output steps as possible.

It is quite straightforward to check whether an ordinary program trace is *covered* by a generalized trace. For example, for $[\triangleright\, x\,]^{\mathbb{Z}}[\,\{\varepsilon, x_C\}\triangleright\,][\,\{\varepsilon, x_C\}\triangleright\,][\,\{x_C\}\triangleright\,]$ the generalized trace for input 1 is ?1 !{1, 1.1, 1.1.1} *stop*. We write *x.y* to distinguish words over $\mathbb{Z}$ from decimal representations of integer numbers. Comparing this generalized trace to a program's trace ?1 !1 !1 *stop*, we can determine that the latter is covered by the former and is therefore a valid program run for the underlying specification. But, unlike previously,

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

we now do not need to speculate how to relate the two outputs !1 with the expected outputs. We can simply gather up all consecutive outputs of the program and then check whether this output sequence is an element of the set of expected output sequences.

To simplify the formal definition of this process, we first normalize ordinary traces by way of the function $\lceil\cdot\rceil : Tr \to Tr_G$ that embeds Tr into Tr_G, the image being exactly the subset of Tr_G with only singleton sets, of non-empty words, in output steps:

$$\lceil t\rceil = \lceil t\rceil_\varepsilon \qquad \lceil !o \cdot t'\rceil_w = \lceil t'\rceil_{w.o}$$

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

This normalization is exactly the fusion of consecutive definite outputs into a single output containing a sequence of values, e.g., $\lceil ?1\ !1\ !1\ stop\rceil = ?1\ !\{1.1\}\ stop$.

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

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

A normalized ordinary trace is covered by a generalized trace if the input sequences of both traces match exactly and it holds that whenever the generalized trace has an output step between two input actions, the normalized ordinary trace has a corresponding output step with one of the value sequences "required" by the generalized trace's set of value sequences at that point; if there is no corresponding output step in the ordinary trace, the trace can only be covered if the generalized trace's output options include ε. Note that, due to the typing of the relation, neither the trace on the left nor that on the right of any occurrence of $\prec$ can contain directly consecutive output steps. Having no consecutive outputs in the right argument means that in the third rule we do not need to check that t_1 does not begin with an output. When checking $t_1 \prec t_2$ in this context, t_2 either starts with an input or is *stop*. So the check on the shape of t_1 is performed implicitly in the next step.

8.3.1 Computing generalized traces

Equipped with the definition of generalized traces, we now need a way to actually compute them for a given specification. The basic idea of generalized traces was to find, for a fixed input sequence, a representation of all program runs that match the behavior of the given specification. In order to get a single *non*-generalized trace that fulfills a specification, we can leverage the definition of *accept* to search for an accepted trace with a specific input sequence. Basically, we want to solve for t in the equation $accept(s, k_I)(t, \Delta_I) = True$. We can do this by evaluating *accept* with t unfixed – up to $inputs(t) = vs$ – and on demand extending the trace with the appropriate steps such that we never fall into a *False*-case. That is, we take the next unused value from the desired input sequence at every input action and choose one of the possible outputs of each output action. Figure 10 exemplifies this for a certain specification and singleton input sequence. This process results in a particular non-generalized trace (normalized or not) that matches the specification, provided

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

Let $s = [\triangleright x]^{\tau}_{\top}[\{x_C, 2 * x_C\} \triangleright][\{1, \varepsilon\} \triangleright]$ and fix $inputs(t) = [v_1]$.

Assuming $v_1 \notin \tau$: $accept(s, k_I)(t, \Delta_I) = True \iff t = ?v_1\ stop = \lceil t \rceil$

Assuming $v_1 \in \tau$:

$accept(s, k_I)(t, \Delta_I) = True$

$\iff t = ?v_1 \cdot t' \wedge accept([\{x_C, 2 * x_C\} \triangleright][\{1, \varepsilon\} \triangleright], k_I)(t', store(x, v_1, \Delta_I)) = True$

$\iff t = ?v_1\ !o_1 \cdot t'' \wedge o_1 \in \{v_1, 2 * v_1\} \wedge accept([\{1, \varepsilon\} \triangleright], k_I)(t'', store(x, v_1, \Delta_I)) = True$

then alternatively, making output choices:

$\iff t = ?v_1\ !o_1 \cdot t'' \wedge o_1 = v_1 \wedge accept([\{1\} \triangleright], k_I)(t'', store(x, v_1, \Delta_I)) = True$

$\iff t = ?v_1\ !o_1\ !o_2\ stop \wedge o_1 = v_1 \wedge o_2 = 1 \iff \lceil t \rceil = ?v_1\ !\{v_1.1\}\ stop$

or:

$\iff t = ?v_1\ !o_1 \cdot t'' \wedge o_1 = v_1 \wedge accept(\mathbf{0}, k_I)(t'', store(x, v_1, \Delta_I)) = True$

$\iff t = ?v_1\ !o_1\ stop \wedge o_1 = v_1 \iff \lceil t \rceil = ?v_1\ !\{v_1\}\ stop$

or:

$\iff t = ?v_1\ !o_1 \cdot t'' \wedge o_1 = 2 * v_1 \wedge accept([\{1\} \triangleright], k_I)(t'', store(x, v_1, \Delta_I)) = True$

$\iff t = ?v_1\ !o_1\ !o_2\ stop \wedge o_1 = 2 * v_1 \wedge o_2 = 1 \iff \lceil t \rceil = ?v_1\ !\{(2 * v_1).1\}\ stop$

or:

$\iff t = ?v_1\ !o_1 \cdot t'' \wedge o_1 = 2 * v_1 \wedge accept(\mathbf{0}, k_I)(t'', store(x, v_1, \Delta_I)) = True$

$\iff t = ?v_1\ !o_1\ stop \wedge o_1 = 2 * v_1 \iff \lceil t \rceil = ?v_1\ !\{2 * v_1\}\ stop$

Fig. 10: Solving *accept* for t with desired input sequence.

the desired input sequence is an element of the set of relevant input sequences for the given specification. If we want to get a *generalized* trace instead, all we need to do is to not choose a single output per output action but rather extend the trace with all possible outputs each time.

In Figure 11, the definition of this execution function is given. The most notable conceptual deviation from the *accept*-function, apart from turning an acceptor into an evaluator, is equation (11.4). It avoids the case distinction from the corresponding part in Figure 8 since we consider all possible output values of an output action and combine consecutive output values into single words. For notational simplicity, we assume that $eval(\varepsilon, \Delta) = \varepsilon$. Figure 12 shows the evaluation of *gtrace* in correspondence to the solving of "$accept = True$" in Figure 10. Note how all output alternatives are now explored in one go.

Similarly to *accept*, the function $gtrace(s, k_I^T)(\cdot, \Delta_I)$ can produce an error if we try to evaluate an x_C "too early" or encounter a top-level **E**. But compared to *accept*, the *gtrace*-function has a lot more cases where instead of a result, i.e., a generalized trace, it will return an error. For example, this happens when we encounter an input action after we ran out of input values or when an input value is not part of the required value set for the input mode that assumes only valid inputs. Additionally, and equally as important, *gtrace* also results in an error if there are still input values left after the end of the specification is reached. In fact, *gtrace* returns an error in those places in its definition where *accept* would return `error` or *False*. The only exception to this is *accept*'s case for checking outputs,

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

$$gtrace([\triangleright x]^{\tau} \cdot s', k)(vs, \Delta) = \begin{cases} ?v \cdot gtrace(s', k)(vs', store(x, v, \Delta)) & \text{, if } vs = v : vs' \wedge v \in \tau \\ \texttt{error} & \text{, otherwise} \end{cases} \quad (11.1)$$

$$gtrace([\triangleright x]^{\tau}_{\top} \cdot s', k)(vs, \Delta) = \begin{cases} ?v \cdot gtrace(s', k)(vs', store(x, v, \Delta)) & \text{, if } vs = v : vs' \wedge v \in \tau \\ ?v\ stop & \text{, if } vs = [v] \wedge v \notin \tau \\ \texttt{error} & \text{, otherwise} \end{cases} \quad (11.2)$$

$$gtrace([\triangleright x]^{\tau}_{\bigcirc} \cdot s', k)(vs, \Delta) = \begin{cases} ?v \cdot gtrace(s', k)(vs', store(x, v, \Delta)) & \text{, if } vs = v : vs' \wedge v \in \tau \\ ?v \cdot gtrace([\triangleright x]^{\tau}_{\bigcirc} \cdot s', k)(vs', \Delta) & \text{, if } vs = v : vs' \wedge v \notin \tau \\ \texttt{error} & \text{, otherwise} \end{cases} \quad (11.3)$$

$$gtrace([\,\Theta\triangleright\,] \cdot s', k)(vs, \Delta) = eval(\Theta, \Delta) \odot gtrace(s', k)(vs, \Delta) \quad (11.4)$$

$$\text{with } O \odot t = \begin{cases} !\{o.o' \mid o \in O, o' \in O'\} \cdot t' & \text{, if } t = !O' \cdot t' \\ !O \cdot t & \text{, otherwise} \end{cases}$$

$$gtrace(([c] \Rightarrow s_1 \triangle s_2) \cdot s', k)(vs, \Delta) = \begin{cases} gtrace(s_1 \cdot s', k)(vs, \Delta)\text{ , if } eval(c, \Delta) = True \\ gtrace(s_2 \cdot s', k)(vs, \Delta)\text{ , otherwise} \end{cases} \quad (11.5)$$

$$gtrace(s^{\rightarrow \mathbf{E}} \cdot s', k)(vs, \Delta) = gtrace(s, k')(vs, \Delta) \quad (11.6)$$

$$\text{with } k'(cont) = \begin{cases} gtrace(s, k')\text{ , if } cont = \texttt{End} \\ gtrace(s', k)\text{ , if } cont = \texttt{Exit} \end{cases}$$

$$gtrace(\mathbf{E} \cdot s', k)(vs, \Delta) = k(\texttt{Exit})(vs, \Delta) \quad (11.7)$$

$$gtrace(\mathbf{0}, k)(vs, \Delta) = k(\texttt{End})(vs, \Delta) \quad (11.8)$$

$$k_I^T(cont)(vs, \Delta) = \begin{cases} stop & \text{, if } cont = \texttt{End} \wedge vs = [\,] \\ \texttt{error} & \text{, otherwise} \end{cases}$$

Fig. 11: Specification execution (differences to Figure 8 are in gray).

Assuming $\underline{v_1 \notin \tau}$: $gtrace([\triangleright x]^{\tau}_{\top}[\,\{x_C, 2 * x_C\} \triangleright\,][\,\{1, \varepsilon\} \triangleright\,], k_I^T)([v_1], \Delta_I) = ?v_1\ stop$

Assuming $\underline{v_1 \in \tau}$:

$$\begin{aligned} & gtrace([\triangleright x]^{\tau}_{\top}[\,\{x_C, 2 * x_C\} \triangleright\,][\,\{1, \varepsilon\} \triangleright\,], k_I^T)([v_1], \Delta_I) \\ &= ?v_1 \cdot gtrace([\,\{x_C, 2 * x_C\} \triangleright\,][\,\{1, \varepsilon\} \triangleright\,], k_I^T)([\,], store(x, v_1, \Delta_I)) \\ &= ?v_1 \cdot (\{v_1, 2 * v_1\} \odot gtrace([\,\{1, \varepsilon\} \triangleright\,], k_I^T)([\,], store(x, v_1, \Delta_I))) \\ &= ?v_1 \cdot (\{v_1, 2 * v_1\} \odot (\{1, \varepsilon\} \odot gtrace(\mathbf{0}, k_I^T)([\,], store(x, v_1, \Delta_I)))) \\ &= ?v_1 \cdot (\{v_1, 2 * v_1\} \odot (\{1, \varepsilon\} \odot stop)) \\ &= ?v_1 \cdot (\{v_1, 2 * v_1\} \odot !\{1, \varepsilon\}\ stop) \\ &= ?v_1\ !\{v_1.1, v_1, (2 * v_1).1, 2 * v_1\}\ stop \end{aligned}$$

Fig. 12: Evaluating *gtrace* for the example from Figure 10.

equation (8.4). There, the *False*-case is now dropped instead as *gtrace* computes the correct outputs itself and so nothing can go wrong at that point. The replacing of *accept*'s other *False*-cases by `error` is by design, as we intend *gtrace* to only ever be called on value sequences that are known to originate from accepted traces, i.e., on sequences from the set of valid input sequences for the given specification.

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

$$paths([\triangleright x]^{\tau} \cdot s', k)(\Delta) = \text{——}\langle v \in \tau\rangle\text{——}paths(s', k)(store_{SYM}(x, v, \Delta)) \quad (\star) \quad (13.1)$$

$$paths([\triangleright x]^{\tau}_{\top} \cdot s', k)(\Delta) = \begin{array}{l} \text{——}\langle v \in \tau\rangle\text{——}paths(s', k)(store_{SYM}(x, v, \Delta)) \\ \text{——}\langle v \notin \tau\rangle\text{——}\top \end{array} \quad (\star) \quad (13.2)$$

$$paths([\triangleright x]^{\tau}_{\circlearrowleft} \cdot s', k)(\Delta) = \begin{array}{l} \text{——}\langle v \in \tau\rangle\text{——}paths(s', k)(store_{SYM}(x, v, \Delta)) \\ \text{——}\langle v \notin \tau\rangle\text{——}paths([\triangleright x]^{\tau}_{\circlearrowleft} \cdot s', k)(\Delta) \end{array} \quad (\star) \quad (13.3)$$

$$paths([\,\Theta \triangleright\,] \cdot s', k)(\Delta) = paths(s', k)(\Delta) \quad (13.4)$$

$$paths(([c] \Rightarrow s_1 \triangle s_2) \cdot s', k)(\Delta) = \begin{array}{l} \text{——}\langle eval_{SYM}(c, \Delta)\rangle\text{———}paths(s_1 \cdot s', k)(\Delta) \\ \text{——}\langle \neg eval_{SYM}(c, \Delta)\rangle\text{———}paths(s_2 \cdot s', k)(\Delta) \end{array} \quad (13.5)$$

$$paths(s^{\rightarrow \mathbf{E}} \cdot s', k)(\Delta) = paths(s, k')(\Delta) \quad \text{with } k'(cont) = \begin{cases} paths(s, k') \text{ , if } cont = \texttt{End} \\ paths(s', k) \text{ , if } cont = \texttt{Exit} \end{cases} \quad (13.6)$$

$$paths(\mathbf{E} \cdot s', k)(\Delta) = k(\texttt{Exit})(\Delta) \quad (13.7)$$

$$paths(\mathbf{0}, k)(\Delta) = k(\texttt{End})(\Delta) \quad (13.8)$$

$$k^P_I(cont)(\Delta) = \begin{cases} \text{——}\top \text{ , if } cont = \texttt{End} \\ \texttt{error} \text{ , if } cont = \texttt{Exit} \end{cases}$$

($\star$) with fresh variable v

Fig. 13: Specification paths (differences to Figure 8 are in gray).

Due to similar reasoning as for *accept*, the *gtrace*-function will terminate even if the (well-formed) specification contains an infinite iteration. However, in contrast to *accept*, we now really need exactly the progress condition on iterations in order to guarantee termination of *gtrace*. This results from the fact that *gtrace* cannot guarantee that its specification- or its input-sequence-argument will permanently decrease in size on consuming an output action (inside an iteration).

8.4 Specification paths

The last concept we introduce formally are the specification paths we discussed in Section 5.1. Specification paths are sequences of symbolic constraints on input values. Constraints either describe to what set a certain input value needs to belong or to what set it must not, or they describe whether a specific branching condition should evaluate to *True* or to *False* at a specific point in time. Constraints of single input values are denoted as $\langle v \in \tau\rangle$ or $\langle v \notin \tau\rangle$, for some symbolic variable v. Conditions are denoted as $\langle c\rangle$ or $\langle \neg c\rangle$ with c being the result of symbolically evaluating a term from $T_{\mathbb{B}}$. Each specification path ends with $\top$ indicating termination of the underlying behavior.

Once again, we modify the *accept*-function, this time to generate the tree containing all specification paths for the argument specification (see Figure 13). For *gtrace* we essentially dropped the outputs from *accept*'s trace argument and turned its *False*-cases into errors (except for the case where *accept* checks the trace's outputs). Now to generate the path

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

$$gtrace([\triangleright x]^{\tau}_{\top}[\{x_C, 2 * x_C\} \triangleright][\{1, \varepsilon\} \triangleright], k_I^T)([v_1, \ldots, v_n], \Delta_I)$$
$$= \begin{cases} ?v_1\ !\{v_1.1, v_1, (2 * v_1).1, 2 * v_1\}\ stop & , \text{if } n = 1 \wedge v_1 \in \tau \\ ?v_1\ stop & , \text{if } n = 1 \wedge v_1 \notin \tau \\ \texttt{error} & , \text{if } n \neq 1 \end{cases}$$

$$paths([\triangleright x]^{\tau}_{\top}[\{x_C, 2 * x_C\} \triangleright][\{1, \varepsilon\} \triangleright], k_I^P)(\Delta_I)$$
$$= \begin{array}{l} \langle v_1 \in \tau \rangle \text{—} paths([\{x_C, 2 * x_C\} \triangleright][\{1, \varepsilon\} \triangleright], k_I^P)(store_{SYM}(x, v, \Delta_I)) \\ \langle v_1 \notin \tau \rangle \text{—} \top \end{array}$$
$$= \begin{array}{l} \langle v_1 \in \tau \rangle \text{—} \top \\ \langle v_1 \notin \tau \rangle \text{—} \top \end{array}$$

Fig. 14: Comparing *gtrace*, as per Figure 12, and *paths*.

tree, we drop the trace argument completely and instead introduce fresh symbolic variables when we need to access input values, and gather up constraints on these variables. We replace the functions *store* and *eval* by symbolic versions, and Δ now holds lists of symbolic variables instead of concrete values as before. For example, the term $sum(x_A) > 0$ is evaluated symbolically to $v_1 + v_2 + v_3 > 0$ under the environment $[x \mapsto [v_1, v_2, v_3]]$. Here v_1, v_2 and v_3 are pairwise distinct elements from a set of symbolic variables. Moreover, we turn *accept*'s case distinctions over input values and branching conditions into forks in the tree with one branch for every case in *accept* that does not immediately return *False*. So while in *gtrace* we turned *False*-cases into errors, for *paths* we now simply elide them completely. In fact, a direct comparison between *gtrace* and *paths* is also illustrative, both on the level of definitions (Figure 11 vs. Figure 13) and based on an example. Regarding the latter, consider again the specification used in Figures 10 and 12. We only looked at it for $vs = [v_1]$ so far, but from the definition of *gtrace* it is not difficult to see that exactly all input sequences of other lengths would be rejected as erroneous.[17] As it happens, the *paths*-function implicitly reveals that information as well, see Figure 14.

The above modifications turn *accept*, an inductively defined consumer of concrete traces, into a co-inductively defined producer of symbolic constraints. Note that $paths(s, k_I^P)(\Delta_I)$ can result in a tree with infinite paths. From equation (13.3), it is clear that this happens if s has an input action with the $\circlearrowleft$-mode. But it can also happen when s contains an iteration that can repeat its body, i.e., there is at least one path with no occurrence of **E** in the iteration body. For the $\circlearrowleft$-mode, the infinite paths will have infinitely many different input constraints of the form $\langle v_i \notin \tau \rangle$. Infinite paths arising from iterations also have infinitely many different input constraints, possibly interspersed with branching constraints, because the progress condition in the (Till-**E**) rule from Figure 6 ensures that we never repeat the iteration body before producing at least one new input constraint.

We can now define the set of specification paths P^s for some specification s to be all finite paths from the root to a $\top$-labeled leaf that are contained in $paths(s, k_I^P)(\Delta_I)$. That

[17] More explicit and general calculations for the same example in Section 8.5.2, concerning both *gtrace* and *accept*, confirm this aspect.

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

is, $P^s = \{R\, c_1 \ldots c_n \top \in \mathit{paths}(s, k_I^P)(\Delta_I)\}$, with R denoting the root of the tree of specification paths. We can also, for every $l \in \mathbb{N}$, compute the finite subset $P_l^s = \{p \in P^s \mid |p| \leqslant l\}$ where $|p|$ denotes the length of path p defined as the number of input constraints in p. The presence of infinite paths does not interfere with the ability to compute this set. By the arguments above, every infinite path has infinitely many input constraints and checking whether $|p| > l$ is decidable in finite time even if p is an infinite path.

For some path $p \in P^s$ and input sequence $vs \in [\mathbb{Z}]$, we write $vs \models p$ if $vs = [v_1, \ldots, v_{|p|}]$ and all constraints on p are fulfilled by the values of vs. Note that the input sequence must have exactly as many elements as the number of input values introduced by the path in order for $vs \models p$ to hold.

8.5 Correctness of testing

Our testing procedure, as introduced in Section 6, uses *paths* and evaluation to generalized traces (*gtrace*), but the semantics of our specification language and the notion of program correctness are given by the definition of *accept*. We therefore relate the testing procedure to the definition of program correctness to show that we actually test in accordance with the defined semantics.

Recall from Section 8.2.1 that we have the following definition for when a program *prog* is correct, given well-formed specification s:

$$\text{For every } vs \in \mathit{inputs}(Tr^s), \text{ it holds that } \mathit{prog}(vs) \in Tr^s.$$

By contrast, our testing procedure searches for a counterexample to the following statement:

$$\text{For every } p = R\, c_1 \ldots c_n \top \in \mathit{paths}(s, k_I^P)(\Delta_I) \text{ and } vs \in [\mathbb{Z}] \text{ with } vs \models p, \text{ it holds that } \lceil \mathit{prog}(vs) \rceil \prec \mathit{gtrace}(s, k_I^T)(vs, \Delta_I).$$

In order for the testing procedure to be in line with the *accept*-based notion of program correctness, we need these two statements to be equivalent. In particular, we aim for two facts:

1. The input generation using *paths* and constraint solving produces exactly the set of relevant input sequences, i.e., $\mathit{inputs}(Tr^s)$.
2. The combination of *gtrace*, $\lceil \cdot \rceil$, and $\prec$ can be used to decide $t \in^? Tr^s$ for each trace $t \in Tr$ with $\mathit{inputs}(t) \in \mathit{inputs}(Tr^s)$.

We will make the connections precise by introducing a lemma and a conjecture, each of which a strengthening of the corresponding desired fact above.

8.5.1 Input generation

We need to show that from using paths and constraint solving we get exactly the set of relevant input sequences for the given specification. So every relevant input sequence must satisfy some path and no input sequence that satisfies a path shall be irrelevant.

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

Lemma 1. *Let $s \in Spec$ be well-formed. It holds $inputs(Tr^s) = \biguplus_{p \in P^s} \{vs \in [\mathbb{Z}] \mid vs \models p\}$.*

Proof Recall that $P^s = \{R\, c_1 \ldots c_n \top \in paths(s, k_I^P)(\Delta_I)\}$. Instead of just stating that with *paths* we get exactly the set of relevant input sequences, the lemma additionally states that input sequences found by different paths must be different too. The latter follows directly from the definition of *paths*. If a path is split in equations (13.2), (13.3) or (13.5), the next constraints in the resulting two paths are each others' negations. So no two paths $p, q \in P^s$ with $p \neq q$ exist that have a common satisfying sequence (with the exact required length), i.e., a sequence $vs \in [\mathbb{Z}]$ such that $vs \models p$ and $vs \models q$. This means that all subsets $\{vs \in [\mathbb{Z}] \mid vs \models p\}$ in the right-hand side's union are disjoint.

Now we establish both directions of the equality:

- $inputs(Tr^s) \subseteq \biguplus_{p \in P^s} \{vs \in [\mathbb{Z}] \mid vs \models p\}$

  For any $t \in Tr^s$ we know that $accept(s, k_I)(t, \Delta_I) = True$ and thus can obtain satisfied constraints on the input sequence of t from the side conditions in equations (8.1), (8.2), (8.3), and (8.5). The condition for *True* in k_I then corresponds to the end of the path. The other side conditions in Figure 8, namely in equation (8.4) for handling an output action, do not constrain the input sequence and are therefore unimportant here.

  From the gathered conditions and the corresponding cases in Figure 13, we deduce that there exists a corresponding (unique) path $p \in P^s = \{R\, c_1 \ldots c_n \top \in paths(s, k_I^P)(\Delta_I)\}$ whose symbolic constraints are made true when concrete values are taken from the sequence $inputs(t)$, i.e., $inputs(t) \models p$.

  Therefore, $inputs(t) \in \biguplus_{p \in P^s} \{vs \in [\mathbb{Z}] \mid vs \models p\}$.
- $inputs(Tr^s) \supseteq \biguplus_{p \in P^s} \{vs \in [\mathbb{Z}] \mid vs \models p\}$

  Given $p \in P^s$ and $vs \in [\mathbb{Z}]$ with $vs \models p$, we can construct a trace $t \in Tr$ with $inputs(t) = vs$ and with outputs such that $accept(s, k_I)(t, \Delta_I) = True$, i.e., even $t \in Tr^s$. We do this, likewise to Figure 10, by going through the specification as if we were to evaluate *accept*, building up the trace along the way, starting from an empty trace prefix. When we encounter an input action in the specification, we take the next unused input v from vs, extending the trace prefix with $?v$. We know from the definition of *paths* that the constraints satisfied on vs/v guarantee we do not fall into any *False*-cases in the definition of *accept*. For branching, we can simply use all the values read in until that point to determine which branch to pick. This choice necessarily agrees with the choice made at the corresponding position in p. On output actions in the specification, we simply choose any $o \in eval(\Theta, \Delta)$ and extend the trace prefix with $!o$ if $o \neq \varepsilon$ or leave it as is otherwise. Upon reaching the end of the evaluation, i.e., when reaching $k_I(\texttt{End})$, we complete the trace prefix to a full trace by adding the *stop*. The constraints in path p guarantee that we have exactly enough input values in the input sequence to actually reach the end.[18]

[18] Of course, all this is essentially the same as what *gtrace* does, but here we end up with an ordinary trace instead of a generalized trace that encodes all possible outputs in a normalized form.

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

Therefore, for every sequence in $\biguplus_{p \in P^s} \{vs \in [\mathbb{Z}] \mid vs \models p\}$ there exists a trace in Tr^s with that sequence as input sequence.

■

8.5.2 Deciding $t \in^? Tr^s$

Our testing procedure uses the result of evaluating *gtrace* on a relevant input sequence in combination with $\prec$ and $\lceil \cdot \rceil$ to judge concrete traces for correctness/acceptance. To better see the connections between *gtrace* and *accept*, we first look at the evaluations of both functions on a small but representative (except for the aspect of iterations, which we will look at later) example specification, and an arbitrary $t \in Tr$ (not necessarily known to also be $\in Tr^s$):

Let $s = [\triangleright x]^{\tau}_{\top}[\{x_C, 2 * x_C\} \triangleright][\{1, \varepsilon\} \triangleright]$, the specification we already used in Figures 10, 12 and 14. We require τ to be a non-trivial set of allowed values here, i.e., we want neither $\tau = \varnothing$ nor $\tau = \mathbb{Z}$. The reason for this is simply that we want conditions like $v \in \tau$ and $v \notin \tau$ to both be satisfiable. Additionally, let $t = o^0\, ?v_1\, o^1 \ldots ?v_n\, o^n\, stop \in Tr$ and $vs = inputs(t) = [v_1, \ldots, v_n] \in [\mathbb{Z}]$ with $o^i = !o^i_1 \ldots !o^i_{n_i}$ being potentially empty sequences of output actions. Crucially, unlike in the earlier considerations of *accept*- and *gtrace*-evaluations for this example specification, we now do not start with fixing $n = 1$, instead staying more general. We can already describe the shape of t after normalization: $\lceil t \rceil = !\{w_0\}\, ?v_1\, !\{w_1\} \ldots ?v_n\, !\{w_n\}\, stop$, with $w_i = o^i_1 \ldots o^i_{n_i}$. For notational simplicity, we treat $!\{w_i\}$ as ε in case of $w_i = \varepsilon$. This happens if $n_i = 0$, i.e., $o^i = \varepsilon$.

We now look at the possible values of $t_g = gtrace(s, k^T_I)(vs, \Delta_I)$ and compare them with the evaluation of $accept(s, k_I)(t, \Delta_I)$, i.e., with checking which traces are in Tr^s. For *gtrace*, we will ultimately get the result already displayed earlier in Figure 14, but now we more carefully explore the genesis of the `error`-cases. Apart from this aspect, the calculation is largely a repeat from Figure 12, though this time around we do not start with $vs = [v_1]$ outright but instead let n be arbitrary initially. For cases $n \geqslant 1$ we abbreviate $store(x, v_1, \Delta_I)$ to Δ'.

$$t_g = gtrace([\triangleright x]^{\tau}_{\top}[\{x_C, 2 * x_C\} \triangleright][\{1, \varepsilon\} \triangleright], k^T_I)(vs, \Delta_I)$$

$$= \begin{cases} ① \; ?v_1 \cdot gtrace([\{x_C, 2 * x_C\} \triangleright][\{1, \varepsilon\} \triangleright], k^T_I)([v_2, \ldots, v_n], \Delta') \\ \qquad\qquad , \text{if } n \geqslant 1 \wedge v_1 \in \tau \\ ② \; ?v_1\, stop \quad , \text{if } n = 1 \wedge v_1 \notin \tau \\ ③ \; \texttt{error} \quad , \text{if } n = 0 \vee (n > 1 \wedge v_1 \notin \tau) \end{cases}$$

$$= \begin{cases} ① \; ?v_1 \cdot (\{v_1, 2 * v_1\} \odot (\{1, \varepsilon\} \odot gtrace(\mathbf{0}, k^T_I)([v_2, \ldots, v_n], \Delta'))) \text{ , if } n \geqslant 1 \wedge v_1 \in \tau \\ ② \\ ③ \end{cases}$$

$$= \begin{cases} ④ \; ?v_1 \cdot (\{v_1, 2 * v_1\} \odot (\{1, \varepsilon\} \odot stop)) \\ \qquad\qquad , \text{if } n = 1 \wedge v_1 \in \tau \\ ⑤ \; \texttt{error} \quad , \text{if } n > 1 \wedge v_1 \in \tau \\ ② \\ ③ \end{cases}$$

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

$$= \begin{cases} ④\ ?v_1\ !\{v_1.1, v_1, (2 * v_1).1, 2 * v_1\}\ stop \\ \qquad\qquad , \text{if } n = 1 \wedge v_1 \in \tau \\ ② \\ ⑥\ \texttt{error} \quad , \text{if } n = 0 \vee n > 1 \end{cases}$$

Only the lines with the markers ③ and ⑤ introduce new `error`-cases, in the steps at which they first appear, whereas the line with the marker ⑥ is simply the merger of the two. Analogous marking is also used in the following *accept*-evaluation. The calculation is again similar to the derivation of fulfilling conditions shown in Figure 10, but now we evaluate *accept* with a completely unfixed trace, not even assuming $inputs(t) = [v_1]$, and thus discovering all relevant conditions on the trace structure as well as exploring the *False*-cases explicitly. This time, we see that only the lines with the markers ③, ⑧ and ⑪ introduce new *False*-cases, whereas the line with the marker ⑥ at the end simply merges all three (while ④ merges ⑨ and ⑩). The markers appearing in both calculations (i.e., above and also below) are used for lines related to each other, while ⑤ in the first calculation would be related to the merger of ⑧ and ⑪ in the second calculation.

$$\begin{aligned}
&accept([\,\triangleright\, x\,]^{\tau}_{\top}[\,\{x_C, 2 * x_C\} \triangleright\,][\,\{1, \varepsilon\} \triangleright\,], k_I)(t, \Delta_I) \\
&= \begin{cases} ①\ accept([\,\{x_C, 2 * x_C\} \triangleright\,][\,\{1, \varepsilon\} \triangleright\,], k_I)(t', \Delta'), \text{ where } t' = o^1\ ?v_2\ o^2 \ldots ?v_n\ o^n\ stop \\ \qquad\qquad , \text{if } o^0 = \varepsilon \wedge n \geqslant 1 \wedge v_1 \in \tau \\ ②\ True \quad , \text{if } o^0 = \varepsilon \wedge n = 1 \wedge v_1 \notin \tau \wedge o^1 = \varepsilon \\ ③\ False \quad , \text{if } o^0 \neq \varepsilon \vee n = 0 \vee (n > 1 \wedge v_1 \notin \tau) \vee (n = 1 \wedge v_1 \notin \tau \wedge o^1 \neq \varepsilon) \end{cases} \\
&= \begin{cases} ⑦\ accept([\,\{1, \varepsilon\} \triangleright\,], k_I)(t'', \Delta'), \text{ where } t'' = !o^1_2 \ldots !o^1_{n_1}\ ?v_2\ o^2 \ldots ?v_n\ o^n\ stop \\ \qquad\qquad , \text{if } o^0 = \varepsilon \wedge n \geqslant 1 \wedge v_1 \in \tau \wedge n_1 \geqslant 1 \wedge o^1_1 \in \{v_1, 2 * v_1\} \\ ⑧\ False \quad , \text{if } o^0 = \varepsilon \wedge n \geqslant 1 \wedge v_1 \in \tau \wedge (n_1 = 0 \vee o^1_1 \notin \{v_1, 2 * v_1\}) \\ ② \\ ③ \end{cases} \\
&= \begin{cases} ⑦\ accept([\,\{1\} \triangleright\,], k_I)(t'', \Delta') \vee accept(\mathbf{0}, k_I)(t'', \Delta') \\ \qquad , \text{if } o^0 = \varepsilon \wedge n \geqslant 1 \wedge v_1 \in \tau \wedge n_1 \geqslant 1 \wedge o^1_1 \in \{v_1, 2 * v_1\} \\ ⑧ \\ ② \\ ③ \end{cases} \\
&= \begin{cases} ⑨\ True \quad , \text{if } o^0 = \varepsilon \wedge v_1 \in \tau \wedge o^1_1 \in \{v_1, 2 * v_1\} \wedge (t'' = !1\ stop, \text{ i.e., } n = 1 \wedge n_1 = 2 \wedge o^1_2 = 1) \\ ⑩\ True \quad , \text{if } o^0 = \varepsilon \wedge v_1 \in \tau \wedge o^1_1 \in \{v_1, 2 * v_1\} \wedge (t'' = stop, \text{ i.e., } n = 1 \wedge n_1 = 1) \\ ⑪\ False \quad , \text{if } o^0 = \varepsilon \wedge v_1 \in \tau \wedge o^1_1 \in \{v_1, 2 * v_1\} \wedge (n > 1 \vee n_1 > 2 \vee (n_1 = 2 \wedge o^1_2 \neq 1)) \\ ⑧ \\ ② \\ ③ \end{cases} \\
&= \begin{cases} ④\ True \quad , \text{if } \lceil t \rceil = ?v_1\ !\{w_1\}\ stop \wedge v_1 \in \tau \wedge w_1 \in \{v_1.1, v_1, (2 * v_1).1, 2 * v_1\} \\ ②\ True \quad , \text{if } \lceil t \rceil = ?v_1\ stop \wedge v_1 \notin \tau \\ ⑥\ False \quad , \text{if } o^0 \neq \varepsilon \vee n \neq 1 \ \vee (v_1 \in \tau \wedge o^1 \notin \{!v_1\ !1, !v_1, !(2 * v_1)\ !1, !(2 * v_1)\}) \\ \qquad\qquad\qquad\qquad \vee (v_1 \notin \tau \wedge o^1 \neq \varepsilon) \end{cases}
\end{aligned}$$

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

We now investigate the connections between *accept*'s results and the generalized traces produced by *gtrace*, by case analysis on the just calculated results of *gtrace*:

Case ④ $n = 1 \wedge v_1 \in \tau$: We have $t_g = ?v_1\, !\{v_1.1, v_1, (2 * v_1).1, 2 * v_1\}\, stop$, also $t = o^0\, ?v_1\, o^1 stop$ and $\lceil t \rceil = !\{w_0\}\, ?v_1\, !\{w_1\}\, stop$. So the testing procedure checks this:

$$!\{w_0\}\, ?v_1\, !\{w_1\}\, stop \overset{?}{\prec} ?v_1\, !\{v_1.1, v_1, (2 * v_1).1, 2 * v_1\}\, stop$$

If $w_0 \neq \varepsilon$, then we have $\lceil t \rceil \not\prec t_g$. If $w_0 = \varepsilon$ but $w_1 \notin \{v_1.1, v_1, (2 * v_1).1, 2 * v_1\}$, then we also have $\lceil t \rceil \not\prec t_g$. Only when $w_0 = \varepsilon$ and $w_1 \in \{v_1.1, v_1, (2 * v_1).1, 2 * v_1\}$, does $\lceil t \rceil \prec t_g$ hold. These – respectively $o^0 = \varepsilon$ and $o^1 \in \{!v_1\, !1, !v_1, !(2 * v_1)\, !1, !(2 * v_1)\}$ – are precisely the remaining side conditions for which $accept(s, k_I)(t, \Delta_I) = True$ when $n = 1$ and $v_1 \in \tau$. So if $n = 1 \wedge v_1 \in \tau$, then $t \in Tr^s$ iff $\lceil t \rceil \prec t_g$.

Case ② $n = 1 \wedge v_1 \notin \tau$ and thus $t_g = ?v_1\, stop$: Analogously to above, we again need $w_0 = o^0 = \varepsilon$ and this time also $w_1 = o^1 = \varepsilon$ in order to fulfill $\lceil t \rceil \prec t_g$. And those are again precisely the remaining side conditions for which $accept(s, k_I)(t, \Delta_I) = True$ when $n = 1$ and $v_1 \notin \tau$. So if $n = 1 \wedge v_1 \notin \tau$, then $t \in Tr^s$ iff $\lceil t \rceil \prec t_g$.

Case ⑥ $gtrace(s, k_I^T)(vs, \Delta_I) = \texttt{error}$: This happens exactly when $n \neq 1$. But from the *accept*-evaluation above we also know that then $accept(s, k_I)(t, \Delta_I) = False$, i.e., $t \notin Tr^s$. Put differently, no trace with zero or more than one inputs can be in Tr^s, and hence neither $vs = [\,]$ nor any $vs = [v_1, v_2, \ldots]$ in $inputs(Tr^s)$. So we would not even consider such input sequences as relevant to begin with in the context of testing for program correctness.

Note that computing *accept* introduces *False*-cases (or rather conditions for them) in greater volume than *gtrace* does with `error`, but only as regards outputs. This is because the *False*-cases in *accept* stem from both ill-formed inputs and ill-formed outputs, whereas for *gtrace* the `error`-cases only come from ill-formed inputs, as *gtrace* computes the output sets itself. The additional *False*-cases are therefore exactly those introduced by equation (8.4), the case that we did not turn into an `error` in *gtrace*. In the end this is not surprising, since *gtrace*'s role is to compute and not to check outputs.

In a nutshell: The conditions of *accept* on inputs are obviously all contained in checking $\lceil t \rceil \prec t_g$ (the inputs are unaffected by trace normalization, are directly reproduced by *gtrace* in non-`error`-cases, and are checked for exact agreement in the coverage relation), and the "missing" checks on outputs are also reintroduced when we check $\lceil t \rceil \prec t_g$. Essentially, if the conditions for $gtrace(s, k_I^T)(vs, \Delta_I) \neq \texttt{error}$ are fulfilled, then the (remaining) conditions for $accept(s, k_I)(t, \Delta_I) = True$ are identical to those for $\lceil t \rceil \prec t_g$.

We generalize the observations from the given example into the following conjecture. The important bit about the above calculations actually is that not only do the results "agree", but also the calculations themselves are in suitable correspondence. The way in which steps in one are related to steps in the other is essentially what would drive a proof of the conjecture by cases and induction. In fact, the example specification used was chosen precisely to cover representative structural cases, minus any iteration for simplicity.

Conjecture 2. *Let $s \in Spec$ be well-formed. It holds for every $vs \in [\mathbb{Z}]$,*

a. $gtrace(s, k_I^T)(vs, \Delta_I) = \texttt{error}$ iff $vs \notin inputs(Tr^s)$, and

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

b. for every $t \in Tr$ with $inputs(t) = vs$,
if $gtrace(s, k_I^T)(vs, \Delta_I) = t_g \in Tr_G$, then $t \in Tr^s$ iff $\lceil t \rceil \prec t_g$.

Note that the conjecture would allow us to decide $t \in^? Tr^s$ for arbitrary $t \in Tr$. By choosing $vs = inputs(t)$, we either get $gtrace(s, k_I^T)(inputs(t), \Delta_I) = \texttt{error}$, in which case $inputs(t) \notin inputs(Tr^s)$ and thus $t \notin Tr^s$, or, if $gtrace(s, k_I^T)(inputs(t), \Delta_I) = t_g$, we can check $\lceil t \rceil \prec t_g$. Note also that for our program correctness testing procedure, $\lceil prog(vs) \rceil \prec gtrace(s, k_I^T)(vs, \Delta_I)$, we only ever use input sequences from $inputs(Tr^s)$, so by Conjecture 2a. we have that *gtrace* will never throw an error during testing.[19]

Instead of a complete proof of the statement of the conjecture, we will use the example considered at length above and an additional one given below as a sketch. We believe these two examples to be representative of the cases that would constitute the essential steps in a full induction proof. So we perform(ed) the analysis of these examples in the hope that this sufficiently illustrates how one can go about actually proving the statement of the conjecture. We are convinced a rigorous formal proof is possible, but extremely tedious and boring.

For the second example, we use a specification with an iteration and branching other than via input mode: $s = ([\triangleright x]^{\mathbb{Z}}[x_C = 0] \Rightarrow \mathbf{E} \triangle \mathbf{0})^{\rightarrow \mathbf{E}}$. Again, we compute *accept* and *gtrace* for the general trace form $t = o^0\ ?v_1\ o^1 \ldots ?v_n\ o^n\ stop \in Tr$ and its corresponding input sequence $vs = inputs(t) = [v_1, \ldots, v_n] \in [\mathbb{Z}]$, and use markers to indicate lines related to each other.

$$
\begin{aligned}
t_g &= gtrace(([\triangleright x]^{\mathbb{Z}}[x_C = 0] \Rightarrow \mathbf{E} \triangle \mathbf{0})^{\rightarrow \mathbf{E}}, k_I^T)(vs, \Delta_I) = gtrace([\triangleright x]^{\mathbb{Z}}[x_C = 0] \Rightarrow \mathbf{E} \triangle \mathbf{0}, k'_T)(vs, \Delta_I) \\
&= \begin{cases} \textcircled{1}\ ?v_1 \cdot gtrace([x_C = 0] \Rightarrow \mathbf{E} \triangle \mathbf{0}, k'_T)([v_2, \ldots, v_n], \Delta') & , \text{if } n \geqslant 1 \\ \textcircled{2}\ \texttt{error} & , \text{if } n = 0 \end{cases} \\
&= \begin{cases} \textcircled{3}\ ?v_1 \cdot gtrace(\mathbf{E}, k'_T)([v_2, \ldots, v_n], \Delta') & , \text{if } n \geqslant 1 \wedge v_1 = 0 \\ \textcircled{4}\ ?v_1 \cdot gtrace(\mathbf{0}, k'_T)([v_2, \ldots, v_n], \Delta') & , \text{if } n \geqslant 1 \wedge v_1 \neq 0 \\ \textcircled{2} \end{cases} \\
&= \begin{cases} \textcircled{3}\ ?v_1 \cdot gtrace(\mathbf{0}, k_I^T)([v_2, \ldots, v_n], \Delta') & , \text{if } n \geqslant 1 \wedge v_1 = 0 \\ \textcircled{4}\ ?v_1 \cdot gtrace([\triangleright x]^{\mathbb{Z}}[x_C = 0] \Rightarrow \mathbf{E} \triangle \mathbf{0}, k'_T)([v_2, \ldots, v_n], \Delta') & , \text{if } n \geqslant 1 \wedge v_1 \neq 0 \\ \textcircled{2} \end{cases} \\
&= \begin{cases} \textcircled{5}\ ?v_1\ stop & , \text{if } n = 1 \wedge v_1 = 0 \\ \textcircled{6}\ \texttt{error} & , \text{if } n > 1 \wedge v_1 = 0 \\ \textcircled{4} \\ \textcircled{2} \end{cases}
\end{aligned}
$$

[19] Incidentally, we also institute a corresponding guarantee about running the program itself on the input sequence. Our setup in Section 8.2.1 uses partiality in that it considers $prog(vs)$ to be undefined when *prog* tries to consume too few or too many inputs, but in practice we actually still use the above check even then: We compute $prog(vs)$ using a less strict version of run_{rep}, possibly producing a special "read beyond provided inputs" action, and then check $\lceil prog(vs) \rceil \prec gtrace(s, k_I^T)(vs, \Delta_I)$ as usual. The generalized trace computed by *gtrace* here will definitely have exactly *vs* as its sub-sequence of input actions, resulting in a mismatch with $\lceil prog(vs) \rceil$, and a somewhat informative one at that. In fact, we have already seen this at work in the testing and error reporting for *wrong1* in Section 7.1.

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

$$= \begin{cases} ⑤ \\ ⑦\ ?v_1\ ?v_2\ stop & \text{, if } n = 2 \wedge v_1 \neq 0 \wedge v_2 = 0 \\ ⑧\ ?v_1\ ?v_2 \cdot gtrace([\triangleright x]^{\mathbb{Z}}[x_C = 0] \Rightarrow \mathbf{E} \triangle \mathbf{0}, k'_T)([v_3, \ldots, v_n], store(x, v_2, \Delta')) & \\ \qquad\qquad & \text{, if } n \geqslant 2 \wedge v_1 \neq 0 \wedge v_2 \neq 0 \\ ⑨\ \texttt{error} & \text{, if } n = 1 \wedge v_1 \neq 0 \\ ⑩\ \texttt{error} & \text{, if } n > 2 \wedge v_1 \neq 0 \wedge v_2 = 0 \\ ②, ⑥ & \end{cases}$$

$\ldots$

$$= \begin{cases} ?v_1 \ldots ?v_n\ stop & \text{, if } n \geqslant 1 \wedge v_{1 \leqslant i < n} \neq 0 \wedge v_n = 0 \\ \texttt{error} & \text{, otherwise} \end{cases}$$

$accept(([\triangleright x]^{\mathbb{Z}}[x_C = 0] \Rightarrow \mathbf{E} \triangle \mathbf{0})^{\rightarrow \mathbf{E}}, k_I)(t, \Delta_I) = accept([\triangleright x]^{\mathbb{Z}}[x_C = 0] \Rightarrow \mathbf{E} \triangle \mathbf{0}, k')(t, \Delta_I)$

$$= \begin{cases} ①\ accept([x_C = 0] \Rightarrow \mathbf{E} \triangle \mathbf{0}, k')(t', \Delta'), \text{ where } t' = o^1\ ?v_2\ o^2 \ldots ?v_n\ o^n\ stop & \\ \qquad\qquad & \text{, if } o^0 = \varepsilon \wedge n \geqslant 1 \\ ②\ False & \text{, if } o^0 \neq \varepsilon \vee n = 0 \end{cases}$$

$$= \begin{cases} ③\ accept(\mathbf{E}, k')(t', \Delta') & \text{, if } o^0 = \varepsilon \wedge n \geqslant 1 \wedge v_1 = 0 \\ ④\ accept(\mathbf{0}, k')(t', \Delta') & \text{, if } o^0 = \varepsilon \wedge n \geqslant 1 \wedge v_1 \neq 0 \\ ② & \end{cases}$$

$$= \begin{cases} ③\ accept(\mathbf{0}, k_I)(t', \Delta') & \text{, if } o^0 = \varepsilon \wedge n \geqslant 1 \wedge v_1 = 0 \\ ④\ accept([\triangleright x]^{\mathbb{Z}}[x_C = 0] \Rightarrow \mathbf{E} \triangle \mathbf{0}, k')(t', \Delta') & \text{, if } o^0 = \varepsilon \wedge n \geqslant 1 \wedge v_1 \neq 0 \\ ② & \end{cases}$$

$$= \begin{cases} ⑤\ True & \text{, if } o^0 = \varepsilon \wedge n = 1 \wedge v_1 = 0 \wedge o^1 = \varepsilon \\ ⑪\ False & \text{, if } o^0 = \varepsilon \wedge n = 1 \wedge v_1 = 0 \wedge o^1 \neq \varepsilon \\ ⑥\ False & \text{, if } o^0 = \varepsilon \wedge n > 1 \wedge v_1 = 0 \\ ④ & \\ ② & \end{cases}$$

$$= \begin{cases} ⑤ & \\ ⑦\ True & \text{, if } o^0 = \varepsilon \wedge n = 2 \wedge v_1 \neq 0 \wedge o^1 = \varepsilon \wedge v_2 = 0 \wedge o^2 = \varepsilon \\ ⑧\ accept([\triangleright x]^{\mathbb{Z}}[x_C = 0] \Rightarrow \mathbf{E} \triangle \mathbf{0}, k')(o^2\ ?v_3\ o^3 \ldots ?v_n\ o^n\ stop, store(x, v_2, \Delta')) & \\ \qquad\qquad & \text{, if } o^0 = \varepsilon \wedge n \geqslant 2 \wedge v_1 \neq 0 \wedge o^1 = \varepsilon \wedge v_2 \neq 0 \\ ⑨\ False & \text{, if } o^0 = \varepsilon \wedge n = 1 \wedge v_1 \neq 0 \\ ⑩\ False & \text{, if } o^0 = \varepsilon \wedge n > 2 \wedge v_1 \neq 0 \wedge o^1 = \varepsilon \wedge v_2 = 0 \\ ⑫\ False & \text{, if } o^0 = \varepsilon \wedge n \geqslant 1 \wedge v_1 \neq 0 \wedge o^1 \neq \varepsilon \\ ⑬\ False & \text{, if } o^0 = \varepsilon \wedge n = 2 \wedge v_1 \neq 0 \wedge o^1 = \varepsilon \wedge v_2 = 0 \wedge o^2 \neq \varepsilon \\ ②, ⑥, ⑪ & \end{cases}$$

$\ldots$

$$= \begin{cases} True & \text{, if } n \geqslant 1 \wedge v_{1 \leqslant i < n} \neq 0 \wedge v_n = 0 \wedge o^{0 \leqslant i \leqslant n} = \varepsilon \\ False & \text{, otherwise} \end{cases}$$

Comparing $\lceil t \rceil = !\{w_0\}\ ?v_1\ !\{w_1\} \ldots ?v_n\ !\{w_n\}\ stop$, with $w_i = o^i_1 \ldots o^i_{n_i}$, to t_g, analogously to the previous example, we now find that $\lceil t \rceil \prec t_g$ exactly when $o^i = \varepsilon$ for all $0 \leqslant i \leqslant n$, which again is precisely the remaining part of the side condition for the ultimate *True*-case

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

of *accept*. This should not be too surprising here either, as the only cases that have an impact on this correspondence are the ones regarding input actions and output actions in the specification, as well as the conditions for `error`-free termination in the initial continuations k_I and k_I^T. All of these conditions were already present in the previous example. Neither branching nor iterations add any new side conditions to the evaluation.

The novel part, highlighted by this example, is the fact that the same argument we used for the first example also works in the presence of loops (and branching). The ellipsis in both derivations abstracts two things that make this work. First, the possibility that t can have an arbitrary finite length and still get accepted. And second, the fact that eventually the computation will terminate due to the progress requirement on loop bodies in well-formed specifications.

8.5.3 Wrap-Up

Combining the lemma and the conjecture above gives us the desired statement on the correctness of our testing procedure. Even though we did not fully prove the conjecture, we hope the justifications to be enough to convince the reader it actually holds, and therefore, by extension, that our testing procedure actually tests program correctness as defined in terms of *accept*.

9 Retrospective comments

We have used different versions of the presented framework in our course over the last years, with a disruption of Autotool usage in one academic year when our university was suffering and then recovering from a hacking incident that had brought the whole digital infrastructure to a halt and then much of it kept offline for several months. For our students, the instant online feedback seems to be a key motivating factor for working on the exercise tasks (which are neither mandatory in the course nor part of the final grade in any way). Both in the course period when we could not use Autotool and instead improvised a workflow where students would get the same automated feedback on their submissions, but only once the exercise week was completed and thus only on one solution attempt, as well as when we have posed tasks that came without instant feedback for other reasons, student participation went down. We interpret this as there being distinctive value in a setup where students can in principle work in a test-driven-development style, getting feedback at each step in the process. That does not mean we do not also inspect submitted programs individually to provide feedback on programming style and other aspects that black-box testing cannot cover. But such manual inspection happens only once per week and only on the last submission per student.

We can also report anecdotally on the value of using property-based testing in particular. With unit tests, as we sometimes have for simple tasks or when the overall test-suite is partitioned into a student visible and a hidden one, at least some in our audience have a tendency to try to circumvent the overall task intent and instead address just the special cases contained in the unit tests. Even though there is no harm arising from such behavior in our non-graded exercise setting, we still find it educationally beneficial to make it so that students cannot game the system in that way. This applies already to the exercise tasks

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

on pure functional programs (transforming data structures etc.), and since students thus become used to getting a fresh counterexample whenever they make progress on the task solution by covering more of its semantics correctly but not yet all of it, it only makes sense to us to realize the same experience in the I/O setting.

In that context, the quality of the counterexamples produced is highly important, of course. Comparing the counterexamples displayed in Section 7.1.1, produced with the old input generation method, to the ones displayed in the subsection before that, shows the improvements exemplarily. We have first used the new input generation method in the Summer 2024 edition of the course and observed the impact. Most of the time students are given exactly the smallest input sequence for which their program does not conform to the required behavior. Even so, the trace output can become overwhelming due to the all-possibilities-in-one aspect of generalized traces but also due to the pattern output we use for display in the case of string values instead of the simplified integer-only values handled in the examples in this article. We have started to experiment with heuristics for turning generalized traces into single ordinary traces for display, as well as even more mundane variations such as laying out long traces horizontally (as in all the examples printed in this article) vs. laying them out vertically with additional alignment.

One instance where in the past we have posed I/O programming tasks without automated feedback, as alluded to above, was the interactive part of Tic-tac-toe (in preceding weeks students implement the pure game logic), since we had not worked out how to encode it into previous versions of the framework. Now we support that task, as indicated in Section 7.2. Games also appear to profit particularly from programmatic generation of tests instead of simply using unit tests, because it becomes easier to account for different game strategies. Generally, we are satisfied with the expressiveness of the framework. Every behavior that we want to test, we can express in the language; and additionally, we have not found any textbook example that was not expressible, apart from what we call "output-driven loops", but we do not want these anyway.

Of course, some bias could be at play here: we have our own notions of what we consider interesting or non-interesting I/O programming tasks, and our specification language ends up being a very good fit to express the former ones. In any case, for us as educators it is almost as important, beside the ability to express relevant tasks in the first place, to be able to make modifications to them without breaking anything. Before using the framework, we almost never changed the tasks and their manually written generators and checkers, and even with the initial version of the framework supporting the naive-but-automated input generation method as well as full trace match checking, we were still reluctant to make changes, since the input generation was fragile and without coverage guarantees. Now we regularly experiment with new tasks and can produce variations on them quickly.

Sometimes, varying a task leads to surprises. As reported in Section 5.3.1, we had an occasion where a seemingly innocent change from "sum up the read values" to "build the product of read values" led the testing to become unreliable due to overflow problems, and that is what motivated our introduction of extra constraints into the framework. Other occasions where we still need to adapt the framework in response to wanting to realize a particular exercise task are largely in the context of supporting new functions in conditional expressions in specifications. As indicated in Section 7.1, we support certain functions from the standard library out of the box. Whenever a new function should be used for

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

branching decisions, ad-hoc extension of the framework is currently needed, since the SMT solver also needs to be taught about this function. We have plans to streamline this process in the sense that an educator/user of the framework could themselves provide the relevant SMT code.

Both when using the framework as educators and when extending the framework as developers, we have substantially profited from the specification interpreter as elaborated on in Section 7.4.

10 Related work

Due to the large number of existing automatic task grading and assessment tools, we cannot give a complete overview here. A list of many different published automatic grading systems for programming tasks is maintained by Strickroth & Striewe (2022). A survey (with a focus on feedback generation) of different automatic assessment tools for programming tasks is presented by Keuning *et al*. (2019). Most tools use some form of automatic testing against specified test cases or compare submissions to the results of sample solutions. Additionally, a number of tools use program transformation or static analysis techniques to determine how a program deviates from a sample solution. Task specification is usually done through unit or property tests or by providing sample solutions in the respective programming language. As far as we can tell, no existing system defines intended behavior using a formal specification language designed expressly for that purpose.

Similarly, there exists a large body of research on program test case generation with symbolic execution (King, 1976; Cadar & Sen, 2013). Compared to the literature, our approach is very simple. A major reason is that we do not execute actual programs but rather the specifications, which are syntactically much simpler than a full-featured programming language. Additionally, we construct path constraints purely from the specifications and do not incorporate information from running the specified behavior on concrete inputs. This is sometimes called static symbolic execution, as opposed to dynamic symbolic execution where dynamic, i.e. runtime, information is included (Cadar & Engler, 2005; Godefroid *et al*., 2005; Sen, 2007; Godefroid, 2011; Giantsios *et al*., 2015). Dynamic symbolic execution makes it easier to find very specific inputs that pure symbolic execution cannot find, e.g., because some condition is not expressible symbolically and rarely satisfied by random inputs. Concrete execution can simplify conditions such that symbolic reasoning is possible again and can enable much longer paths to be explored. This sometimes includes lifting information about complex functions to the level of the SMT solver in the form of additional constraints on uninterpreted functions (Anand *et al*., 2008). Some authors have also used machine learning techniques to improve path coverage (Liu *et al*., 2021). Due to the very restricted nature of our specification language, such optimizations were not yet necessary in our application context.

Reach problems are another approach to finding inputs that exercise specific program paths (Naylor & Runciman, 2007; Fowler & Hutton, 2016). In a reach problem an expression in some program is marked as a target and then a reach solver tries to find inputs to a source function such that the target expression is evaluated. Reach solvers can, for example, be implemented using lazy narrowing, which is a form of symbolic evaluation developed in the context of functional-logic programming.

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

Searching for values that satisfy some predicate does not necessarily require a constraint solving approach. Claessen *et al*. (2014) automatically derive generators for values of an algebraic data type from a Boolean predicate on that type alone.

The general mechanism for building inspectable representations of effectful programs (Swierstra & Altenkirch, 2007) is provided in the Haskell IOSpec library.[20] It supports not only console I/O but also forking processes, mutable references, and software transactional memory. However, its API is very minimal and no higher-level abstractions exist.

Quviq QuickCheck (Hughes, 2007, 2016) is a variation of the original QuickCheck that deals with stateful computations in the context of the Erlang programming language.[21] Instead of testing specific programs, like we do, they test whole stateful APIs. A specification of such an API is a semantic model, given in Erlang, of the API together with pre- and post-conditions for each stateful action. Testing is then done by generating random sequences of actions based on the pre-conditions and checking the result of the actual API calls against the model and post-conditions. Any found sequence of API calls that do not behave in accordance with the semantic model is simplified via shrinking to obtain a small counterexample.

Many approaches for formal descriptions of behavior exist. For example, session types describe communication protocols between multiple parties of some (distributed) system (Honda *et al*., 1998; Vasconcelos, 2012). Programs with compatible session types are statically guaranteed to adhere to the specified protocol. In contrast to our specifications, session types usually do not inspect communicated values. They "only" describe the communication pattern, but the range of expressible patterns far exceeds the simple interaction patterns of console I/O programs. Dynamic session type variants exist that are not checked statically and allow for some access to communicated data (Neykova, 2013; Fowler, 2016).

Software contracts are another way to specify behavior of programs (Hinze *et al*., 2006; Dimoulas *et al*., 2016). These contracts are akin to complex dynamically checked types but often exist as first-class entities inside a programming language. Various different flavours of contracts exist, including contracts capable of describing effects like I/O (Moy *et al*., 2024).

Elsewhere, we have shown how, in principle, we can automatically generate, for every expression in our specification language, source code of a (Haskell) program with the respective behavior (Westphal & Voigtländer, 2020*b*, FLOPS'20). Such programs can be used as sample solutions for (automatically generated) tasks, or even as part of task descriptions. Advances in generative language models may also lead to more ways of profiting from such code artifacts (Sarsa *et al*., 2022). Other applications of machine learning are conceivable in the context of feedback generation, to help explain the root causes of detected behavior mismatch in student programs, as Seidel *et al*. (2017) do for type errors.

Additionally, using a slightly earlier version of the language presented in this article, we have previously built a separate EDSL for specifying templates for automatic task generation (Westphal, 2021, WFLP'20). Templates represent a diverse range of tasks and are centered around deriving various artifacts, like example runs and code fragments, from

[20] `https://hackage.haskell.org/package/IOSpec`
[21] See `http://hackage.haskell.org/package/quickcheck-state-machine` for a Haskell version.

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

specifications in our language. For each given specification, the templates can then be instantiated further to possibly randomized variations of a basic task design, significantly reducing time spent on manually creating such variations.

11 Future work

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

- **Automatic generation of meaningful specifications.** Using the presented language of specifications as a basis for automatic exercise generation requires a generation method for meaningful specifications, possibly randomized. Generated specifications should describe behavior that can be expressed verbally in a compact and/or simple natural language description or at least somewhat resembles interactions occurring in real-world programs. Also, the overall complexity of the specified behavior needs to be controllable in order to not generate overwhelmingly difficult tasks. When multiple specifications are needed as the basis for individualized tasks, uniform complexity is equally important to ensure fairness among students. Blindly building specification expressions just constrained by syntactic correctness and well-formedness will likely only rarely fulfill this criterion, so a more structured approach is needed. We have taken first steps to develop such an approach (Westphal, 2025).
- **Generation of helpful feedback.** Currently the feedback we gain from a failing test case contains no real pointers to the root of the problem but just a (short) counterexample for which the program behaves in a certain wrong way. Inferring, for example, a specification for the wrongly behaving program and comparing it to the target specification might help us identify behavioral mismatches at a higher level of abstraction.

Additionally, the language itself and the testing framework as a whole can be improved in various ways or built upon. For example:

- **Improvements to constraint solving.** The presented method of input generation works well for most of our relatively small examples. But it is generally not hard to find examples where it performs sub-optimally, both in terms of runtime as well as achievable search depth. It might be worth exploring if different variations of symbolic execution from the literature, especially those including runtime information, yield overall better or simply faster generation procedures.
- **Additional I/O capabilities.** I/O programs can do much more than reading and writing from and to the console. For example, a lot of traditional (imperative) exercise tasks for novice programmers include reading and writing files as well. The specification language could be extended to express such behavior, allowing more diverse behavior to be tested. We could introduce atomic actions on files like moving files around or appending two files and writing the result into a new file. Filenames could be values from outside the program, i.e., be arguments on the command line as

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

opposed to values read interactively (or both). Having these features in the language would allow testing of behavior like "Swap the contents of two files" or "Copy all files in folder X to folder Y".

- **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. For example, the approach could be adapted to other domains like transformations of lists using a certain set of predefined combinators or declarative descriptions of simple images composed of geometric primitives. Specification languages for such other domains might use very different structuring principles than we do for our I/O specifications. For example, they might need non-tail recursion and thus be more akin to context-free grammars than to regular expressions, which served as inspiration for our dealing with iterations.

12 Conclusion

We presented a formal language for specifying the interactive behavior of console I/O programs. By doing so, we gain the ability to automatically generate test cases from intuitive declarative specifications. Tests are generated with a mix of probabilistic and exhaustive testing techniques combined with constraint solving. We can therefore easily pose new tasks and check the correctness of program submissions. The presented framework is a significant improvement to the ability to state and automatically grade exercises compared to ad-hoc testing with preexisting tools and techniques. Constraint-driven input generation improves the quality of generated test cases, compared to our initial version of the framework. Additionally, the fact that we can manipulate the formal descriptions of behavior programmatically opens up a wide range of possibilities for further automation and analyses.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions that helped us develop the material in several ways beyond what was originally submitted.

Conflicts of interest

None.

References

Anand, S., Godefroid, P. & Tillmann, N. (2008) Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer Berlin, Heidelberg, Germany, pp. 367–381.

Bjørner, N., Phan, A.-D. & Fleckenstein, L. (2015) νZ - An optimizing SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer Berlin, Heidelberg, Germany, pp. 194–199.

Cadar, C. & Engler, D. (2005) Execution generated test cases: How to make systems code crash itself. In *Model Checking of Software (SPIN)*. Springer Berlin, Heidelberg, Germany, pp. 2–23.

Cadar, C. & Sen, K. (2013) Symbolic execution for software testing: Three decades later. *Commun. ACM*. **56**(2), 82–90.

https://doi.org/10.1017/S0956796825100075 Published online by Cambridge University Press

Claessen, K. (2012) Shrinking and showing functions (functional pearl). In *2012 Haskell Symposium*. Association for Computing Machinery, New York, NY, USA, pp. 73–80.

Claessen, K., Duregård, J. & Pałka, M. H. (2014) Generating constrained random data with uniform distribution. In *Functional and Logic Programming (FLOPS)*. Springer International Publishing, Switzerland, pp. 18–34.

Claessen, K. & Hughes, J. (2000) QuickCheck: A lightweight tool for random testing of Haskell programs. In *5th International Conference on Functional Programming (ICFP)*. Association for Computing Machinery, New York, NY, USA, pp. 268–279.

de Moura, L. & Bjørner, N. (2008) Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer Berlin, Heidelberg, Germany, pp. 337–340.

de Vries, E. (2023) falsify: Internal Shrinking Reimagined for Haskell. In *2023 Haskell Symposium*. Association for Computing Machinery, New York, NY, USA, pp. 97–109.

Dimoulas, C., New, M. S., Findler, R. B. & Felleisen, M. (2016) Oh Lord, please don't let contracts be misunderstood (functional pearl). In *21st International Conference on Functional Programming (ICFP)*. Association for Computing Machinery, New York, NY, USA, pp. 117–131.

Duregård, J., Jansson, P. & Wang, M. (2012) Feat: Functional enumeration of algebraic types. In *2012 Haskell Symposium*. Association for Computing Machinery, New York, NY, USA, pp. 61–72.

Dutra, R., Laeufer, K., Bachrach, J. & Sen, K. (2018) Efficient sampling of SAT solutions for testing. In *40th International Conference on Software Engineering (ICSE)*. Association for Computing Machinery, New York, NY, USA, pp. 549–559.

Fowler, J. & Hutton, G. (2016) Towards a theory of reach. In *Trends in Functional Programming (TFP)*. Springer International Publishing, Switzerland, pp. 22–39.

Fowler, S. (2016) An Erlang implementation of multiparty session actors. In *9th Interaction and Concurrency Experience Workshop (ICE)*. Open Publishing Association, pp. 36–50.

Giantsios, A., Papaspyrou, N. & Sagonas, K. (2015) Concolic testing for functional languages. In *17th International Symposium on Principles and Practice of Declarative Programming (PPDP)*. Association for Computing Machinery, New York, NY, USA, pp. 137–148.

Gibbons, J. (2021) How to design co-programs. *J. Funct. Program*. **31**, e15.

Godefroid, P. (2011) Higher-order test generation. In *32nd Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, New York, NY, USA, pp. 258–269.

Godefroid, P., Klarlund, N. & Sen, K. (2005) DART: Directed automated random testing. In *26th Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, New York, NY, USA, pp. 213–223.

Hinze, R., Jeuring, J. & Löh, A. (2006) Typed contracts for functional programming. In *Functional and Logic Programming (FLOPS)*. Springer Berlin, Heidelberg, Germany, pp. 208–225.

Honda, K., Vasconcelos, V. T. & Kubo, M. (1998) Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems (ESOP)*. Springer Berlin, Heidelberg, Germany, pp. 122–138.

Hughes, J. (2007) QuickCheck testing for fun and profit. In *Practical Aspects of Declarative Languages (PADL)*. Springer Berlin, Heidelberg, Germany, pp. 1–32.

Hughes, J. (2016) Experiences with QuickCheck: Testing the hard stuff and staying sane. In *A List of Successes That Can Change the World – Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Springer International Publishing, Switzerland, pp. 169–186.

Hughes, J. (2020) How to specify it! In *Trends in Functional Programming (TFP)*. Springer Nature, Switzerland, pp. 58–83.

Hutton, G. (2016) *Programming in Haskell, Second Edition*. Cambridge University Press, Cambridge, UK.

Keuning, H., Jeuring, J. & Heeren, B. (2019) A systematic literature review of automated feedback generation for programming exercises. *ACM Trans. Comput. Educ. (TOCE)*. **19**(1), 3:1–3:43.

King, J. C. (1976) Symbolic execution and program testing. *Commun. ACM*. **19**(7), 385–394.

Liu, D., Ernst, G., Murray, T. & Rubinstein, B. I. P. (2021) Legion: Best-first concolic testing. In *35th International Conference on Automated Software Engineering (ASE)*. Association for Computing Machinery, New York, NY, USA, pp. 54–65.

Moy, C., Dimoulas, C. & Felleisen, M. (2024) Effectful software contracts. In *Proc. ACM Program. Lang*. **8**(POPL), 2639–2666.

Naylor, M. & Runciman, C. (2007) Finding inputs that reach a target expression. In *7th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, Washington, DC, USA, pp. 133–142.

Neykova, R. (2013) Session types go dynamic or how to verify your Python conversations. In *5th Workshop on Programming Language Approaches to Concurrency and Communication-centric Software (PLACES)*. Open Publishing Association, pp. 95–102.

Pike, L. (2014) SmartCheck: Automatic and efficient counterexample reduction and generalization. In *2014 Haskell Symposium*. Association for Computing Machinery, New York, NY, USA, pp. 53–64.

Runciman, C., Naylor, M. & Lindblad, F. (2008) SmallCheck and Lazy SmallCheck – Automatic exhaustive testing for small values. In *2008 Haskell Symposium*. Association for Computing Machinery, New York, NY, USA, pp. 37–48.

Sarsa, S., Denny, P., Hellas, A. & Leinonen, J. (2022) Automatic generation of programming exercises and code explanations using large language models. In *2022 Conference on International Computing Education Research (ICER)*. Association for Computing Machinery, New York, NY, USA, pp. 27–43.

Schrijvers, T. (2023) *Soar with Haskell*. Packt Publishing, Birmingham, UK.

Seidel, E. L., Sibghat, H., Chaudhuri, K., Weimer, W. & Jhala, R. (2017) Learning to blame: localizing novice type errors with data-driven diagnosis. In *Proc. ACM Program. Lang*. **1**(OOPSLA), 1–27.

Sen, K. (2007) Concolic testing. In *22nd International Conference on Automated Software Engineering (ASE)*. Association for Computing Machinery, New York, NY, USA, pp. 571–572.

Siegburg, M., Voigtländer, J. & Westphal, O. (2019) Automatische Bewertung von Haskell-Programmieraufgaben. In *4th Workshop "Automatische Bewertung von Programmieraufgaben" (ABP)*. Gesellschaft für Informatik, Bonn, Germany, pp. 19–26.

Strickroth, S. & Striewe, M. (2022) Building a corpus of task-based grading and feedback systems for learning and teaching programming. *Int. J. Eng. Pedagog. (iJEP)*. **12**(5), 26–41.

Swierstra, W. & Altenkirch, T. (2007) Beauty in the beast – A functional semantics for the Awkward squad. In *2007 Haskell Workshop*. Association for Computing Machinery, New York, NY, USA, pp. 25–36.

Thompson, S. (2011) *Haskell: The Craft of Functional Programming*. Pearson Education, Essex, UK.

Vasconcelos, V. T. (2012) Fundamentals of session types. *Inform. Comput*. **217**, 52–70.

Waldmann, J. (2017) Automatische Erzeugung und Bewertung von Aufgaben zu Algorithmen und Datenstrukturen. In *3rd Workshop "Automatische Bewertung von Programmieraufgaben" (ABP)*. CEUR-WS.org, Aachen, Germany.

Westphal, O. (2021) A framework for generating diverse Haskell-I/O exercise tasks. In *Functional and Constraint Logic Programming (WFLP)*. Springer Nature, Switzerland, pp. 97–114.

Westphal, O. (2025) Intent preserving generation of diverse and idiomatic (code-)artifacts. *Electronic Proceedings in Theoretical Computer Science (EPTCS)* **424**, 109–129. doi: 10.4204/EPTCS.424.6.

Westphal, O. & Voigtländer, J. (2020a) Describing console I/O behavior for testing student submissions in Haskell. In *8th and 9th International Workshop on Trends in Functional Programming in Education (TFPIE)*. Open Publishing Association, pp. 19–36.

Westphal, O. & Voigtländer, J. (2020b) Implementing, and keeping in check, a DSL used in E-learning. In *Functional and Logic Programming (FLOPS)*. Springer Nature, Switzerland, pp. 179–197.