Specifying Teletype Behavior for Automated Handling of Exercises on Interaktive Haskell Programs

Oliver Westphal **Janis Voigtländer**

University of Duisburg-Essen

Motivation

- We teach Haskell in a course on programming paradigms
- Each week students solve small programming tasks
- We want to automatically test submissions for correctness
  - Easy for pure functions (QuickCheck etc.)
  - But there is also IO
- Goal:
  - Flexible and easy to use way to specify and automatically test Haskell IO programs
  - Enable further optimization and analysis

Teletype IO Programs

- Two basic operations:

```
readLn :: Read a => IO a
print :: Show a => a -> IO ()
```

- Results of IO computations do not contain information on side effects
- For programs with type `IO ()`, comparing against the result only checks termination
- Therefore we capture a programs monadic effects with traces over basic operations

Current Solution - Algebraic Effects

Swiestra and Altenkirch, Haskell Workshop 2007:

"By constructing an internal model of such concepts [teletype IO, ...] within our programming language, we can test, debug and reason about programs that perform IO as if they were pure."

```
data Teletype a = ReadLine (String -> Teletype a)
                | WriteLine String (Teletype a)
                | Return a

instance Monad Teletype

readLn :: Read a => Teletype a
print :: Show a => a -> Teletype ()

runTeletype :: Teletype a -> [String] -> Trace a

data Trace a = Read String (Trace a)
             | Write String (Trace a)
             | Result a
```

Current Solution - Implementing Tasks

(Silently) hand students this verison of IO

```
type IO = Teletype

type Input = String {- or somthing else -}

validInputs :: Gen [String]
checkCorrectness :: Trace -> Bool
correctTrace :: [Input] -> Trace {- maybe from sample solution -}

prog :: IO () {- student solution -}
check = quickCheck $ forAll validInputs
  \xs -> counterexample (show $ correctTrace xs) $
  checkCorrectness (runTeletype prog xs)
```

For each task implement `validInputs`, `checkCorrectness` and `correctTrace`

Current Solution - Problems

- Implementing new tasks is hard
  - especially checking correctness
  - and accounting for optionality
- Inconsistencies between components possible
- explicit (ad hoc) checking instead of declarative description
- Automatization beyond testing not possible (e.g. task generation)

Specification Language

- Build sufficiently expressive specification language
  - Describe basic IO behavior
  - Leave room for optional extra information (e.g. additional output)
- Automatically derive components for testing
- Keep design minimal to enable more automatization and analysis in the future
  - Task generation
  - Deriving (idiomatic) sample solutions
  - Generate helpful feedback

Example specification

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

Example specification

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

- primitive oprtations
- flow control & iteration
- backwards references/variables

Primitive Operations

- Read a value of "type" τ into x: $[\triangleright x]^{\tau}$
- Print the value of a term t: $[t \triangleright]$
  - Term language can vary depending on what kind of expressions are needed

Flow Control & Iteration

- Branch based on a predicate p: $s_1 \angle p \rangle s_2$
- Iterate (sub-)specification: $s^{\rightarrow \mathbf{E}}$
- Terminate iteration: **E**

Variables

- With conventional variables, iterating over reads shadows references to old values
- Therefore **all** variables store **lists**!
  - Read = append new value to list
  - No value is ever overwritten
  - Two access methods for each variable
    - Last/current value: x_C
    - All values: x_A

Example Task

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

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

Optionality

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

Extend writes to specify a set of possible outputs

$$[\,\{t_1, t_2, \dots\}\triangleright\,]$$

Encode optionality / no output with special "empty" term ε

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

Testing Behavior

- Given program *p* and specification *s*
- Does *p* match the behavior described by *s*?
  1. Define what it means for a single program run/trace to match the specification
  2. Does this hold for all traces *p* can produce?
  3. If yes, then *p* fulfills the specification

Example

- Trace: ?2 ?5 ?3 !8
- Match against specification by left-to-right traversal

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

- Successful match sine the trace is consumed completely

Automatic Testing

- Remember that we need at least the following components

```
validInputs :: Gen [Input]
checkCorrectness :: Trace -> Bool
```

- We can reverse trace matching to sample valid program runs from specifications
- Such a run contains a valid `Input` sequence
- Generalizing runs w.r.t. optionality lets us check if a run of an actual program is contained in the general case
- If this is not the case we found a counterexample
- So the program des not have the described behavior

Example

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

Sample trace: ?3 !$\{\varepsilon, 3\}$?1 !$\{\varepsilon, 2\}$?7 !$\{\varepsilon, 1\}$?4 !$\{12\}$
Matching:

- ?3 !3 ?1 !2 ?7 !1 ?4 !12
- ?3 !3 ?1 ?7 !1 ?4 !12

Non-Matching: ?3 !**4** ?1 !**3** ?7 !**2** ?4 !12

EDSL Implementation

Embedding specifications in Haskell

```
exampleTask :: Specification
exampleTask =
  readInput "n" NatTy <>
  tillE (
  branch
    ((\xs n -> length xs == n) <$> getAll @Int "x" <*> getCurrent "n")
    (readInput "x" IntTy)
    e
  ) <>
  writeOutput [sum <$> getAll @Int "x"]

test :: IOtt () -> IO ()
test prog = quickCheck $ prog `fulfills` exampleTask
```

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

Conclusion

- Simple framework for testing teletype IO behavior
- Clear and declarative specifications
- Potential uses beyond testing:
  - Provide better feedback
  - Specifications can be randomly generated
  - Automatic generation of (idiomatic) sample solutions
  - Maybe even automatic verbalization (of tasks)