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

Oliver Westphal   Janis Voigtländer

University of Duisburg-Essen

Context

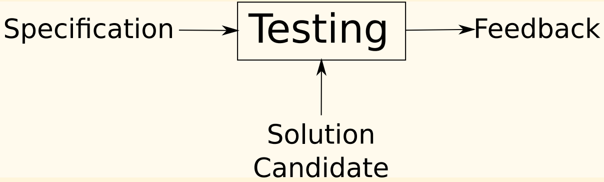
- Programming paradigm course including teaching Haskell
- Automatic grading system for exercise tasks
- Domain specific language embedded in Haskell to easily specify and test correctness of solution candidates for I/O-Exercises[1]
- **Goal**: Ensure the framework works as expected and keeps working when we extend it in the future
- Two perspectives
  - As implementer: Technical correctness
  - As instructor: Consistency of formulated exercises

[1] Westphal, O., & Voigtländer, J. (2020). Describing Console I/O Behavior for Testing Student Submissions in Haskell, TFPIE 2019.

Specification Framework

Main Framework Concepts

1. Language of specifications
2. Notion of program traces
3. Acceptance criterion, relating specifications and traces
4. Testing procedure, checking adherence of programs to specifications



Check whether the set of traces described by a specification contains all traces produced by a program

Specifications

- Read values, storing them in history-valued *variables*
- Output result of computation over *variables*
- Access *variables* either as a list of all values (A) or the most current value (C)
- Basic branching and iteration

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

$$\updownarrow$$

"Read a positive integer n from the console, and then read n integers one after the other and finally output their sum."

Traces

Definition (Trace)

A trace is a sequence $m_0 v_0, m_1, v_1, \ldots, m_n, v_n, \mathsf{stop} \in Tr$, where $n \in \mathbb{N}$, $m_i \in \{?, !\}$ and $v_i \in \mathbb{Z}$.

Example: $?2\ !3\ !8\ \mathsf{stop}$

Definition (Generalized Trace)

A trace is a sequence $m_0 V_0, m_1, V_1, \ldots, m_n, V_n, \mathsf{stop} \in Tr_G$, where $n \in \mathbb{N}$, $m_i \in \{?, !\}$ and $V_i \subseteq \mathbb{Z} \cup \{\varepsilon\}$.

Example: $?2\ !\{3, 6\}\ !\{\varepsilon, 7\}\ !\{8\}\ \mathsf{stop}$

Traces

Covering Relation

$\prec \subseteq Tr \times Tr_G$

$t \prec t_g$ iff t can be obtained by choosing one of the output options from each V_i in t_g.

Examples:

$?2\ !3\ !8\ \text{stop} \prec\ ?2\ !\{3,6\}\ !\{\varepsilon,7\}\ !\{8\}\ \text{stop}$

$?2\ !3\ !8\ \text{stop} \not\prec\ ?2\ !\{3,6\}\ !\{7\}\quad !\{8\}\ \text{stop}$

$?2\ !3\ !8\ \text{stop} \not\prec\ ?4\ !\{3,6\}\ !\{\varepsilon,7\}\ !\{8\}\ \text{stop}$

Acceptance Criterion

accept (simpl.)

Let $s \in Spec$ and $t \in Tr$,
$accept(s, t) = \text{True}$ iff t is a valid program run with regard to the behavior specified by s.

Examples:

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

$accept(s, ?2\ ?3\ ?12\ !15\ \text{stop}) = \text{True}$
$accept(s, ?2\ ?3\ ?12\ !7\ \text{stop}) = \text{False}$
$accept(s, ?3\ ?3\ ?12\ !15\ \text{stop}) = \text{False}$

Acceptance Criterion

Definition (accept)

Let $s \in \mathit{Spec}$ and $t \in \mathit{Tr}$,

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

$$\dots$$

k: continuation handling iteration contexts
Δ: variable store for read values

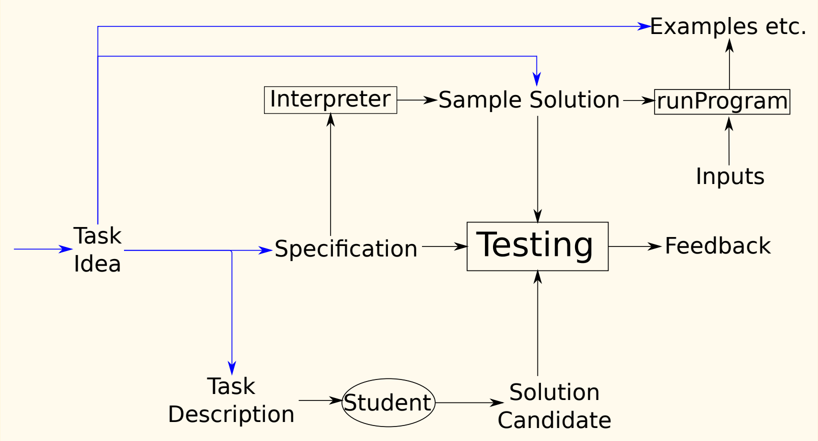
cf. our TFPIE 2019 paper

Testing procedure

- Derive a function *traceSet* from *accept* by "solving" $accept(s, k_I)(t, \Delta_I) = \text{True}$ for t
- $traceSet(s, k_I^T)(\Delta_I) \subseteq Tr_G$ contains all generalized traces valid for s
- Testing a program p against specification s:
  1. Sample $t_g \in Tr_G$ from $traceSet(s, k_I^T)(\Delta_I)$
  2. Extract input sequence from t_g
  3. Run p on these inputs $\rightarrow t \in Tr$
  4. Check whether $t \prec t_g$
- *traceSet* is more complicated to implement than *accept* but we need it for testing

Validating the Implementation

System Overview



Consistency of the framework depends on correctness of testing procedure

Goal

1. Establish correctness of testing procedure
2. Provide means to ensure overall consistency of tasks
   - Task Idea $\leftrightarrow$ specification
   - Correctness of sample solution
   - Automatically create supporting material for verbal task descriptions
   - etc.

Correctness of Testing Procedure

Correctness of testing

1. Translate accept to Haskell (almost verbatim)
2. Establish correctness by code inspection
3. Validate (through testing) more involved testing procedure against accept-semantics:

Theorem

Let $s \in Spec$ and $t \in Tr$, then

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

$$\textit{iff}$$

$$\textit{there exists a } t_g \in traceSet(s, k_I^T)(\Delta_I) \textit{ such that } t \prec t_g.$$

Validation through testing is not a replacement for a proof but much easier to set up.

Test cases

Case 1: "$\Rightarrow$"

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

$\Rightarrow \exists\, t_g \in traceSet(s, k_I^T)(\Delta_I).\, t \prec t_g.$

Hard due to distribution of positive and negative cases.
$\longrightarrow$ Unit testing needed.

Case 2: "$\Leftarrow$"

$\exists\, t_g \in traceSet(s, k_I^T)(\Delta_I).\, t \prec t_g$

$\Rightarrow accept(s, k_I)(t, \Delta_I) = \text{True}$

No restrictions on s required, t with $t \prec t_g$ easily obtainable from t_g.
$\longrightarrow$ Property based testing possible.

Side Note: Random Specifications

Main problem: Generating terminating iterations

- loop skeleton: $(s_1 \cdot (s_2 \measuredangle \mathbf{?} \triangleright s_3))^{\rightarrow \mathbf{E}}$
- condition-progress pair: (c, s^*)
- two possible loops:
  - $(s_1 \cdot (s_2^* \measuredangle c \triangleright (s_3 \cdot \mathbf{E})))^{\rightarrow \mathbf{E}}$
  - $(s_1 \cdot ((s_2 \cdot \mathbf{E}) \measuredangle not(c) \triangleright s_3^*))^{\rightarrow \mathbf{E}}$

(s_i^*: insert s^* (randomly) into s_i)

Examples

$$([\,\{len(y_A)\} \triangleright\,][\,\triangleright z\,]^{\mathbb{Z}}\,(\mathbf{E} \angle not(len(x_A) > 1) \lrcorner\, [\,\triangleright x\,]^{\mathbb{Z}}))^{\rightarrow \mathbf{E}}$$

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

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

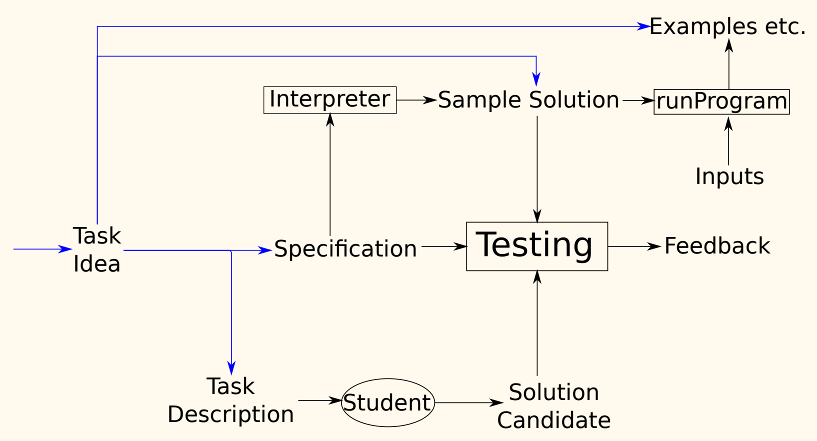
$$[\,\{sum(m_A)\} \triangleright\,]((([\,\triangleright m\,]^{\mathbb{Z}} \angle null(m_A) \lrcorner\, \mathbf{0}) \angle len(x_A) = len(n_A) \lrcorner\, [\,\triangleright m\,]^{\mathbb{Z}})$$

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

- Quality clearly depends on available functions and condition-progress-pairs
- But, unusual specifications can be desirable for testing

Further Checks

System Overview



Overall consistency of the framework allows for cross validation of different artifacts

Exercise Creation Workflow

Creating Artifacts

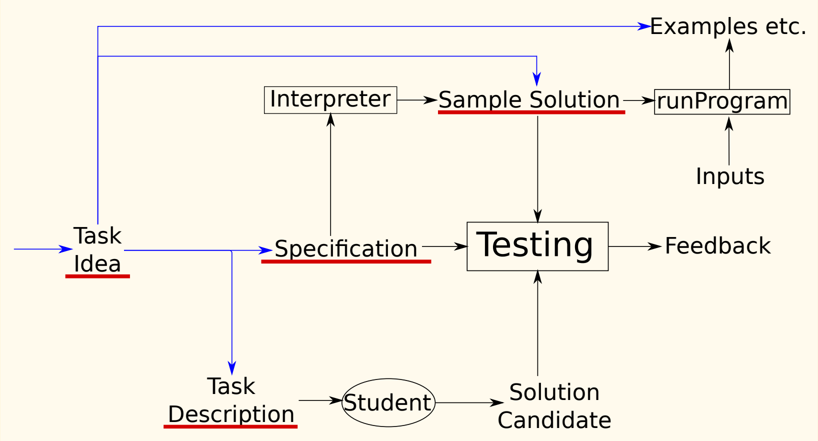
- Task idea: *"We want students to realize a simple I/O loop, so they should write a program that reads a number and then as many further numbers and finally prints a sum."*
- Task description: *"Write a program which first reads a positive integer n from the console, then reads n integers one after the other, and finally outputs their sum"*
- Specification:

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

- Sample solution:

```
main :: IO ()
main = ...
```

System Overview



Validating Artifacts

- Check whether sample solution and interpretation of the specification have matching behavior on some sample inputs
- Use testing procedure to check sample solution against specification itself
- Carefully inspect the task description
- Generate supporting material, e.g. example runs and add to task description:
  “Example: After reading 2, 7, and 13, your program should print 20.”

Conclusion

- Validation of the implementation's core by relating it to the much simpler *accept*-function
- Cross validation of artifacts ensures consistent exercise tasks
- Both approaches still work if the framework is extended (e.g. enriching the specification language)

Conclusion

- Validation of the implementation's core by relating it to the much simpler *accept*-function
- Cross validation of artifacts ensures consistent exercise tasks
- Both approaches still work if the framework is extended (e.g. enriching the specification language)
- Implementation available at `https://github.com/fmidue/IOTasks`