Automatische Bewertung von Haskell-Programmieraufgaben

Marcellus Siegburg[1], Janis Voigtländer[1], Oliver Westphal[1]

Abstract: Wir beschreiben unsere Vorgehensweise bei der Durchführung von Online-Übungen zur Programmierung in Haskell. Der Fokus liegt insbesondere auf dem Zusammenspiel der verwendeten Sprachmittel, Programmbibliotheken und Tools, die es uns durch ihre Kombination erlauben, verschiedene Aspekte des Übungsbetriebs zu automatisieren bzw. zu erleichtern. Unser Ansatz erlaubt uns das automatische Bewerten von Einreichungen zu typischen Programmieraufgaben. Darüber hinaus sind wir in der Lage, Studierende durch geeignet gestaltete Aufgabenstellungen in Richtung bestimmter Lösungen zu führen, und währenddessen Hilfestellung durch entsprechendes Feedback zu geben.

1 Einführung

Wir führen an der Universität Duisburg-Essen eine Lehrveranstaltung „Programmierparadigmen“ durch, hauptsächlich als Pflichtveranstaltung in einem Kerninformatik-Bachelorstudiengang. Schwerpunkt ist funktionale Programmierung mit Haskell, zu einem geringeren Teil logische Programmierung mit Prolog. Wir möchten konkrete Programmierfertigkeiten und Anwendung der in der Vorlesung behandelten Programmiersprachenkonzepte vermitteln, daher haben die begleitenden Übungen einen hohen Anteil an Programmieraufgaben. Zur Unterstützung, der Studierenden wie auch der Tutoren, setzen wir Automatisierung ein. Dabei geht es sowohl um Bewertung der Korrektheit von finalen Einreichungen als auch um Hinweise während der Bearbeitung, zu Fehlern und Verbesserungsmöglichkeiten.

Unser Vorgehen ist gekennzeichnet durch ineinandergreifenden Einsatz verschiedener Systeme und Werkzeuge, die bei geeignet orchestriertem Zusammenspiel unsere Ziele realisieren, und uns gleichzeitig mit relativ wenig Neuentwicklung auf der technischen (statt inhaltlichen) Seite auskommen lassen. Den Studierenden wiederum wird eine Integration präsentiert, damit sie möglichst wenig Brüche wahrnehmen. Tatsächlich werden ihnen von den einzelnen Bestandteilen wohl nur die ersten beiden als eigenständige Entitäten bewusst:

- CodeWorld – eine Online-Lernumgebung zum Erzeugen von vorwiegend Bildern und Animationen, durch mathematische Ausdrücke und Funktionen
- Autotool – ein E-Learning-System zum Erzeugen, Stellen, Einreichen und Bewerten diverser Arten von Übungsaufgaben und Lösungen
- Typ- und Modulsystem von Haskell: Sprachfeatures, die bei Aufgabengestaltung sowie insbesondere Lenkung von Lösungsversuchen in bestimmte Bahnen hilfreich sind

[1] Universität Duisburg-Essen, {marcellus.siegburg, janis.voigtlaender, oliver.westphal}@uni-due.de

doi:10.18420/abp2019-3

- GHC – Haskell-Compiler, mit zusätzlichen Möglichkeiten der Überprüfung von Code über dessen Erfüllen des Sprachstandards hinaus
- diverse Haskell-Bibliotheken – statische Prüfung zum Beispiel eigener Syntaxbedingungen, sowie dynamische Ausführung von Code in abgesicherter Weise
- HLint – Linter für Haskell, mit reicher Regelsprache, konfigurier- und erweiterbar
- QuickCheck – deklaratives Testframework für pure Funktionen, eigenschaftsbasiert
- IOSpec – Möglichkeit des Testens auch effektbehafteten Codes innerhalb der Sprache, durch Reifikation als normale Datenstrukturen

2 Umgebungen und Ausgangspunkte

CodeWorld [Co] ist zuvorderst ein edukatives Projekt zur Unterstützung der Mathematikausbildung im Kontext nordamerikanischer Mittelschulkurrikula, insbesondere zum Thema Algebra. Im Webbrowser können mittels mathematischer Ausdrücke und Funktionen eigene Bilder, Animationen, Interaktionen erstellt bzw. programmiert werden. Den Unterbau bildet dabei eine Haskell-Bibliothek für geometrische Objekte, und tatsächlich gibt es neben der an das Schulpublikum gerichteten Version, etwa mit stärker an mathematische Schreibweise statt Haskell-Programmierung angelehnter Syntax, auch eine Version der Plattform für „vanilla“ Haskell (`https://code.world/haskell#`). Wir haben diese sowie eine Vorgängerbibliothek bereits seit mehreren Jahren, auch an der Universität Bonn in einer ähnlichen Lehrveranstaltung, eingesetzt, um vor allem zu Beginn des Semesters den Paradigmenwechsel von anweisungsbasiertem zu ausdrucksbasiertem Programmieren möglichst intuitiv zu gestalten. Während die CodeWorld API auch offline als normale Bibliothek genutzt werden kann, scheinen die Studierenden zusätzliche Möglichkeiten der CodeWorld-Plattform zu schätzen, etwa die Online-IDE und interaktive Features wie Inspektion/Zerlegung eines Bildes mit direkter Rückkopplung zum erzeugenden Code.

Über Autotool wurde durch dessen Urheber bereits auf dem dritten ABP-Workshop berichtet [Wa17], wobei der Schwerpunkt auf programmiersprachenunabhängigen, algorithmischen Aufgaben, deren automatischer Überprüfung, und auf der Erstellung/Konfiguration/Randomisierung solcher Aufgaben lag. Autotool ist in Haskell implementiert, und so gab es naheliegenderweise schon früh auch einen Autotool-Aufgabentyp zur Haskell-Programmierung. Dieser wurde ebenso bereits an der Universität Bonn eingesetzt, und um diverse Aspekte erweitert, die wir auch nun weiter nutzen und im Folgenden mit beschreiben.

3 Eigenschaftsbasiertes Testen

Da anders als bei üblichen Autotool-Aufgaben zu formalsprachlichen Themen, Algorithmen oder Datenstrukturen eine Überprüfung von Einreichungen nicht durch Entscheidungsprozeduren geleistet werden kann, arbeiten wir mit Tests. Dabei wird statt Unit-Testing der QuickCheck-Ansatz verwendet [CH00], im Wesentlichen bestehend aus eingebetteten DSLs

für einerseits datentypgetriebene Erzeugung zufälliger Eingaben, andererseits deklarativ spezifizierte Eigenschaften von Funktionen. Für eine Aufgabe zur Ermittlung der Anzahl von Vorkommen eines Elementes in einer Liste, also das Schreiben einer Funktion mit der Signatur `countElem :: Int -> [Int] -> Int`, werden etwa folgende Tests verwendet:

```
main :: IO ()
main = do putStrLn "Singleton list [x] contains x once:"
          quickCheck $ \x -> countElem x [x] == 1
          putStrLn "Singleton list [y] does not contain x if x /= y:"
          quickCheck $ \x y -> x /= y ==> (countElem x [y] == 0)
          putStrLn "Occurrences in a combined list sum up:"
          quickCheck $ \x ys zs -> countElem x (ys ++ zs)
                                   == countElem x ys + countElem x zs
```

Hier werden durch die `quickCheck`-Aufrufe automatisch passende Zahlen(paare) und Listen von Zahlen erzeugt, die angegebenen Gleichungen überprüft, und gefundene Gegenbeispiele (gegebenenfalls nach per ebenfalls datentypgetriebenen Strategien versuchter automatischer Minimierung) rückgemeldet. Mithilfe einer solchen Testsuite kann nun eine studentische Lösung entweder als korrekt, das heißt, dem gewünschten Verhalten entsprechend, oder inkorrekt klassifiziert werden. Obwohl es prinzipiell möglich wäre, auf Basis der einzelnen, in der Testsuite kodierten Eigenschaften zwischen verschiedenen Stufen von Korrektheit zu unterscheiden, verwenden wir aktuell lediglich eine binäre Klassifizierung.

In der obigen Version ist die Testsuite für die Studierenden direkt in der Konsole ausführbar, auf Autotool-Seite erfolgt die Einbettung etwas anders, da zum einen mehr Kontrolle gewünscht ist (etwa eine Staffelung von Tests, oder explizite Timeouts), zum anderen Maßnahmen getroffen werden müssen, um Ausführung von problematischem, eventuell sogar malignem Code auf dem Server zu unterbinden. Exzessives Sandboxing etwa durch Auslagerung in separate Prozesse oder Virtualisierung ist dabei nicht nötig, da einerseits Haskells Semantik und Typsystem diverse potentiell schädliche Seiteneffekte bereits unter Kontrolle halten, andererseits es im Haskell-Ökosystem etablierte Verfahren zur sicheren eingebetteten Ausführung gibt.[2]

Dass die obige Testsuite den Studierenden mit der Aufgabenstellung zur Verfügung gestellt wird, ist ein wesentlicher Aspekt. Sie ist aussagekräftiger als Unit-Tests, stellt hier sogar eine vollständige formale Spezifikation dar, und erlaubt den Studierenden auch lokal eine gründliche Prüfung[3], ohne etwa zum Vergleich bereits Zugriff auf eine Musterlösung haben zu müssen.

Bei anderen Aufgabenstellungen wäre es sogar unmöglich, Korrektheit durch Übereinstimmung mit einer (versteckten) Musterlösung zu definieren, da es kein eindeutiges Lösungsverhalten gibt. Zum Beispiel stellen wir Aufgaben, bei denen Serialisierung und

[2] `http://hackage.haskell.org/package/mueval`, `http://hackage.haskell.org/package/hint`

[3] Die den Studierenden zur Verfügung gestellte Testsuite kann je nach Aufgabe auch lediglich eine Untermenge der tatsächlich durchgeführten Tests sein.

Deserialisierung programmiert werden sollen. Etwa für einen endlichen Aufzählungstyp `data Color = Red | Blue | Yellow` bestünde die Aufgabe dann im Schreiben zweier Funktionen (die zweite per `Maybe` im Ergebnistyp explizit eine partielle Funktion):

```
encode :: [Color] -> [Bit]
decode :: [Bit] -> Maybe [Color]
```

welche in geeignetem Sinne invers zueinander sein müssen. Da es verschiedene Kodierungsstrategien gibt, kann eine Überprüfung einer studentischen Einreichung hier nicht im Vergleich der für `Color`-Listen erzeugten `Bit`-Listen mit den für dieselben `Color`-Listen durch eine vom Aufgabensteller hinterlegte Musterlösung erzeugten `Bit`-Listen bestehen. Vielmehr liegt der Schlüssel im direkten Ausdruck der (für funktionale Korrektheit einzig) relevanten Roundtrip-Eigenschaften, mit bei nicht korrekten Lösungsversuchen automatischem Finden, durch QuickCheck, von diese Eigenschaften verletzenden Gegenbeispielen:

```
main :: IO ()
main = do quickCheck $ \v -> decode (encode v) == Just v
          quickCheck $ \c -> let mv = decode c in
                             isJust mv ==> encode (fromJust mv) == c
```

Der Ansatz wie oben beschrieben ist direkt nur auf pure Funktionen (wie `countElem`, `encode`, `decode`) anwendbar, also Funktionen, die man zwar im Interpreter aufrufen kann, die aber keine vollständigen Programme sind. Ein Programm, das zum Beispiel Eingaben liest, diese verarbeitet, und Ergebnisse ausdruckt, hat in Haskell immer einen `IO`-Typ, verwendet Primitiven wie `putStrLn`, und hat gewöhnlich eine gegenüber purem Code andere Codestruktur (etwa oben in den `main`-Definitionen zu sehen an der imperativ programmiert anmutenden Sequenzialisierung von Anweisungen nach jeweils dem `do`-Schlüsselwort). Auch das Schreiben solcher Programme wollen wir in der Lehrveranstaltung üben, können für die Überprüfung entsprechender Aufgaben aber nicht einfach Eigenschaften der Art „Aussagen über Ergebniswerte für Funktionsaufrufe mit bestimmten Argumentwerten/-mustern“ verwenden, da es nun für die Korrektheit mindestens genauso sehr auf die `IO`-Effekte des Programms ankommt. Auf eine gesteuerte/überwachte Ausführung in einer Konsole mit tatsächlicher Auslösung der Effekte (und dann externe Beobachtung dieser, etwa Einlesen und Parsen der Programmausgabe zur Analyse in Tests) können wir jedoch verzichten dank eines Mechanismus zur Reifikation von Effekten als Datenstrukturen innerhalb Haskell selbst [SA07]. Dieser Ansatz, realisiert in der Bibliothek IOSpec, erlaubt uns zum Beispiel für eine Aufgabe wie „Lies eine Zahl n ein, dann weitere n Zahlen, und drucke deren Summe aus“ die Studierenden ein Programm wie

```
main :: IO ()
main = do putStr "Anzahl der Summanden: "
          n <- readLn
          while ...
            ...
          print ...
```

schreiben zu lassen, welches sie lokal kompiliert in einer Konsole ausführen und mit tatsächlichen Ein- und Ausgabeeffekten walten lassen könnten, für welches wir auf Autotool-Seite jedoch via stiller Ersetzung des „echten“ `IO`-Typs durch eine IOSpec-Typannotation (aber ohne irgendeine weitere Änderung des Programmcodes) eine als pure Funktion testbare Version des Programms erhalten. Diese Version löst keine Effekte mehr aus, sondern berechnet einfach aus Werten einer Eingabestrom-Datenstruktur einen wiederum als Datenstruktur weiterverarbeitbaren Strom von Ausgabewerten, samt Informationen zum Interleaving der Ein- und Ausgaben. Überprüfung der Korrektheit von Einreichungen, und Erzeugung von Feedback etwa in Form von Gegenbeispielen, erfolgt dann konzeptionell nicht anders als für von vornherein pure Funktionen wie `countElem`, `encode`, `decode` oben erläutert. Statt QuickCheck etwa Zahlen, Listen von Zahlen, oder `Color`- bzw. `Bit`-Listen erzeugen zu lassen, verwenden wir nun einen Generator für (zur Aufgabenstellung passende) Eingabeströme, und die überprüften Gleichungen/Eigenschaften nehmen zusätzlich auf den jeweils erhaltenen Ausgabestrom Bezug.

Bezüglich CodeWorld-Aufgaben ist zu sagen, dass wir aktuell keine automatischen Funktionstests durchführen. Es wäre bereits realisierbar, bestimmte dynamische Aspekte zu prüfen, etwa ob Bilder erzeugende Funktionen für relevante Eingaben zumindest in vorgegebener Zeit terminieren. Prinzipiell wäre auch ein QuickCheck-Zugang zur Überprüfung der erzeugten Bilder oder Animationen, auf Passung zur Aufgabenstellung, möglich, da CodeWorld intern mit einer expliziten Repräsentation von Vektorgrafiken als Datenstruktur arbeitet, auf der man Eigenschaften formulieren könnte. Bisher wird die Testsuite für CodeWorld-Aufgaben in Autotool aber einfach leer gelassen. Das Einreichenlassen auch dieser Aufgaben per Autotool (statt etwa Moodle) lohnt dennoch, da damit alle in den folgenden beiden Abschnitten beschriebenen Möglichkeiten zum Tragen kommen.

4 Vorgaben und Einschränkungen

Neben im einfachsten Fall „nur Vorgabe von Typsignaturen“ nutzen wir diverse weitere Mittel, um Aufgabenstellungen näher zu umreißen, Lösungsversuche in eine gewünschte Richtung zu bringen, oder aus verschiedenen Gründen nicht intendierte Wege auch auszuschließen. Ein vorhandenes Sprachmittel sind selektive Imports. Wollen wir etwa in der `countElem`-Aufgabe darauf hinwirken, dass ein Lösungsweg gewählt wird, der möglichst deklarativ die gewünschte Berechnung auf der Eingabeliste als Ganzes ausdrückt (etwa durch eine Comprehension und/oder durch Komposition existierender Transformations- und Aggregierungsfunktionen auf Listen), statt kleinteilig in Anlehnung an imperative Schleifenprogrammierung und mit Indexzugriff auf einzelne Elemente vorzugehen, dann können wir die Standardbibliothek genau minus den Indexzugriffsoperator importieren:

```
import Prelude hiding ((!!))
```

Wollen wir zusätzlich ausschließen, dass Lösungen ergoogelt und eventuell unverstanden übernommen werden, können wir auch noch alle in der zur Aufgabe passenden Antwort auf StackOverflow vorkommenden Higher-Order Funktionen verbieten:

```
import Prelude hiding ((!!), filter, map, fmap, foldl, foldr)
```

Wollen wir umgekehrt, an einem späteren Punkt in der Lehrveranstaltung, gerade den Einsatz einer solchen Funktion üben, also etwa erzwingen, dass eine Lösung per `foldr` geschrieben wird, dann können wir nicht auf ein vorhandenes Sprachmittel zurückgreifen. Es dürfte keine Programmiersprache geben, in der man rein durch Typ- oder Modulkonzepte die Verwendung einer bestimmten Funktion in der Implementierung einer anderen Funktion erzwingen kann. Stattdessen muss konkret die Syntax des eingereichten Programms geprüft werden. Wir verwenden für solche Zwecke eine in der Haskell-Community als Standard geltende Syntax-Bibliothek[4]. Für die meisten Aufgabenstellungen verlangen wir mindestens, dass die Einreichungen einen „Mustervergleich“ gegenüber einem vorgegebenen Programmtemplate erfüllen. Dieses dient sozusagen als Lückentext, der zu füllen/ergänzen ist. Die Syntax von Haskell als ausdrucksbasierter Sprache, in der fast alle bedeutungstragenden Teile kompositionell aufgebaute Ausdrücke sind (statt getrennter syntaktischer Konzepte für Ausdrücke, Anweisungen, Kontrollstrukturen), ist solch einer Formulierung besonders zugänglich. Als „Lücke“ dient der vordefinierte Ausdruck `undefined`. Eine Aufgabenstellung, die etwa `countElem` einfach nur irgendwie implementieren lassen will, würde als Template

```
countElem :: Int -> [Int] -> Int
countElem = undefined
```

vorgeben, eine spezifischer auf `foldr` abstellende Aufgabe stattdessen folgendes Template:

```
countElem :: Int -> [Int] -> Int
countElem value = foldr undefined undefined
```

Sind über einen Lückentext/Mustervergleich hinausgehende Forderungen oder Einschränkungen angezeigt, können wir speziellere Syntaxprädikate verwenden. Auch mittels haskell-src-exts implementiert haben wir so etwa die Möglichkeit der Überprüfung, ob eine eingereichte Funktion ein bestimmtes Fallunterscheidungsmittel nutzt oder nicht, rekursiv definiert ist oder nicht, eine bestimmte andere Funktion aufruft (ggfs. auch tiefer geschachtelt als oben mittels des `foldr undefined undefined` ausdrückbar) oder nicht, etc.

5 Statische Analyse und Hinweise

Neben syntaktischer und Typkorrektheit kann der Haskell-Compiler GHC weitere statische Analysen von Code durchführen. So lassen sich etwa Warnungen ausgeben, wenn bestimmte Arten von Fallunterscheidungen nicht vollständig sind oder sich darin Fälle überlappen, wenn Variablenbindungen zu Name Shadowing führen, wenn für eine Definition keine Typsignatur angegeben ist, wenn toter Code existiert, etc. Unser Haskell-Aufgabentyp in Autotool erlaubt selektives, einzelnes An- und Ausschalten dieser Warnungen, und lässt uns außerdem konfigurieren, ob sie nur als zusätzliches Feedback an die Studierenden

[4] `http://hackage.haskell.org/package/haskell-src-exts`

gegeben werden, oder ob Einhalten der Warnungen, bzw. Entfernen ihrer Ursachen im Code, erzwungen wird, bevor eine Einreichung als korrekt akzeptiert werden kann. So können wir einerseits gezielt typische Fehlerursachen abfangen, andererseits bestimmte Richtlinien durchsetzen.

Darüber hinaus setzen wir mit HLint [HL] ein Tool ein, das spezifische Vorschläge zur Vereinfachung und Verbesserung von Haskell-Code macht. Es weist zum Beispiel auf die Möglichkeit der Verwendung von Standardfunktionen hin, merkt überflüssige Klammerung oder anderweitig ungeschickte Syntaxverwendung an, erfasst aber auch tiefergehende Potentiale für Umstrukturierung oder Zusammenfassung von Ausdrücken, bis hin zu Fällen, in denen eine Datenstruktur unnötigerweise mehrfach traversiert wird (analog zu Loop Fusion). Nicht alle möglichen Hinweise sind für den Code Studierender sinnvoll, gegebenenfalls auch abhängig vom Fortschritt in der Lehrveranstaltung, etwa weil durch HLint vorgeschlagene Higher-Order Funktionen noch gar nicht behandelt wurden. Günstigerweise ist auch HLint feinkörnig konfigurierbar, so dass wir auch hier für jede Aufgabenstellung gezielt entscheiden können, welcher Fundus von Regeln in Anschlag gebracht werden soll. Ebenso behalten wir uns wieder vor, Hinweise nur als Feedback zu geben oder ihre Einhaltung zu erzwingen.

Ein weiterer nützlicher Aspekt von HLint ist, dass es relativ einfach erweiterbar ist. Davon haben wir einerseits bereits Gebrauch gemacht, indem wir basierend auf vergangenen Einreichungen Studierender zusätzliche Regeln abgeleitet haben, die nun Eingang in HLint gefunden haben und in Zukunft automatisch mit geprüft werden. Zum anderen sind auch in einer lokalen Konfiguration von HLint zusätzliche domänenspezifische Regeln angebbar, etwa für CodeWorld-Ausdrücke.

6 Diskussion und Ausblick

Die Bewertung von Programmieraufgaben automatisch durchführen zu lassen und dafür verschiedene Tools zur Bewertung zu verwenden, erscheint naheliegend. Ein zu unserem Vorgehen sehr ähnliches gibt es für OCaml [HP19], jedoch ohne Compilerwarnungen als Teil der Rückmeldung. Ein weiterer Fokus liegt dort zusätzlich darauf, die Studierenden zu motivieren, Unit-Tests für ihre Funktionen zu schreiben. Auch zu Lehrveranstaltungen mit Haskell-Programmierung gibt es bereits entsprechende Erfahrungen, wie zum Beispiel von Blanchette et al. [Bl14] berichtet. Sie setzen weniger Automatisierung ein als wir, im Wesentlichen beschränken sie sich dahingehend auf die Ausführung von Testfällen. Es gibt zahlreiche weitere Ansätze für automatische Bewertung von Programmieraufgaben mit unterschiedlichen Graden an Interaktivität und Rückmeldung, die auf Grund der hohen Anzahl hier nicht alle betrachtet werden können. Keuning et al. [KJH18] geben einen zusammenfassenden Überblick über entsprechende Veröffentlichungen.

Wir haben die Verwendung etablierter Tools für Haskell aufeinander abgestimmt, dabei haben wir insbesondere eine Reihenfolge des Ablaufs festgelegt, so dass die Rückmeldung

bei der schrittweisen Behebung von Problemen hilft und nicht mit zu vielen unterschiedlichen Meldungen überfordert. Die Konfiguration der Tools erfolgt für jede gestellte Aufgabe individuell, dies ermöglicht gezieltere Vorgaben für die Bearbeitung und eine bessere Feinabstimmung der Rückmeldungen. Durch unterschiedliche Vorgaben können die Einreichungen automatisch hinsichtlich verschiedener Kriterien überprüft und bewertet werden. Das ermöglicht es, Richtlinien ohne erhöhten Korrekturaufwand durchzusetzen, außerdem erfolgt die Rückmeldung vom System umgehend.

Es gibt jedoch auch Erweiterungsmöglichkeiten für das entwickelte Vorgehen. Beispielsweise ermöglicht es HLint zwar, strukturelle Vorgaben zu forcieren, allerdings sind diese Mittel eingeschränkt. Auch über unseren Mustervergleich sind bisher nur bestimmte strukturelle Überprüfungen möglich, er unterstützt keine aufgabenspezifischen Anforderungen. Individuelle strukturelle Vorgaben, wie zum Beispiel Überprüfung der Verwendung von Tail-Rekursion, könnten durch erweiterte Analysen des Syntaxbaums der Einreichung forciert werden. Solche Möglichkeiten gibt es zum Beispiel bereits in Learn-OCaml [HP19]. Als gänzliche Neuerung entwickeln wir gerade eine DSL zur intentionellen Beschreibung von (im Moment speziell `IO`-)Aufgaben [WV19], um dann benötigte QuickCheck-Artefakte wie Generatoren, zu testende Eigenschaften und andere Funktionalität automatisch zu erzeugen.

Literatur

[Bl14] Blanchette, J. C.; Hupel, L.; Nipkow, T.; Noschinski, L.; Traytel, D.: Experience Report: The Next 1100 Haskell Programmers. In: Proc. Haskell. S. 25–30, 2014.

[CH00] Claessen, K.; Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: Proc. ICFP. ACM, S. 268–279, 2000.

[Co] CodeWorld, `https://github.com/google/codeworld`, Accessed: Sept. 2019.

[HL] HLint, `https://github.com/ndmitchell/hlint`, Accessed: Sept. 2019.

[HP19] Hameer, A.; Pientka, B.: Teaching the Art of Functional Programming Using Automated Grading (Experience Report). Proc. ACM Program. Lang. 3/ICFP, Article 115, 2019.

[KJH18] Keuning, H.; Jeuring, J.; Heeren, B.: A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. ACM Trans. Comput. Educ. 19/1, Article 3, 2018.

[SA07] Swierstra, W.; Altenkirch, T.: Beauty in the beast: A Functional Semantics for the Awkward Squad. In: Proc. Haskell. ACM, S. 25–36, 2007.

[Wa17] Waldmann, J.: Automatische Erzeugung und Bewertung von Aufgaben zu Algorithmen und Datenstrukturen. In: Proc. ABP, CEUR WS vol. 2015. 2017.

[WV19] Westphal, O.; Voigtländer, J., Describing Textual I/O Behavior for Testing Student Submissions in Haskell. Submitted: Post-Proc. TFPIE, 2019.