aus

Oliver J. Bott, Peter Fricke, Uta Priss, Michael Striewe (Hrsg.)

Automatisierte Bewertung in der Programmierausbildung

Digitale Medien in der Hochschullehre Band 6
2017, 420 Seiten, br., 42,90 €, ISBN 978-3-8309-3606-0

Waxmann Verlag GmbH
www.waxmann.com info@waxmann.com

8 Automatisierte Programmbewertung in der Mathematikausbildung

Uta Priss und Peter Riegler

Zusammenfassung
Insbesondere in Mathematikvorlesungen der ersten Semester mit hohen Teilnehmerzahlen ermöglichen automatisiert bewertete Programmieraufgaben regelmäßiges und zeitnahes Feedback. Zusammen mit einer geeigneten didaktischen Lehrmethode (in diesem Kapitel beruhend auf der APOS-Theorie) können Programmieraufgaben Verstehensprozesse in der Mathematiklehre unterstützen. Neben der APOS-Theorie werden Aufgabentypen vorgestellt, die sich hierfür besonders eignen, und es werden die benötigten Tests und erste Erfahrungen kurz erörtert.

8.1 Einleitung

Der Einsatz elektronischer Werkzeuge in der Mathematikausbildung ist nicht ungewöhnlich und aufgrund der Nähe von Mathematik und Informatik nahe liegend. Der Einsatz von Software als Berechnungswerkzeug ist mit Taschenrechner und zum Teil auch Computeralgebrasystemen (CAS) in der Mathematiklehre, wenn auch immer wieder kontrovers diskutiert [Hol95], weit verbreitet. CAS werden auch als eine Art Laborumgebung eingesetzt, in der Studierende Mathematik explorieren können. Hinzu kommt der Einsatz von Software meist in Form von *Applets*, um mathematische Zusammenhänge zu visualisieren.

Teile dieses Kapitels entstanden im Rahmen des Projekts eCULT, Teilvorhaben eAssessment, gefördert durch das Bundesministerium für Bildung und Forschung unter dem Förderkennzeichen 01PL16066H. Die Verantwortung für den Inhalt dieses Kapitels liegt bei der Autorin und dem Autor.

Im Vergleich zu den genannten Werkzeugen und Zielen scheint Programmieren als Tätigkeit, die das Erlernen von Mathematik begünstigt, bisher eher selten Teil der Mathematiklehre zu sein. Für einen solchen Einsatz sprechen aus didaktischer Sicht gute Gründe, die durchaus die Nachteile, die mit einem solchen Einsatz einhergehen (z. B. das Hinzufügen der Schwierigkeiten, die mit Programmieren verbunden sind, zu der ohnehin schon als herausfordernd empfundenen Lehre von Mathematik), kompensieren können. Mathematik wie auch andere MINT-Disziplinen erfordert erstens die koordinierte Nutzung verschiedener Repräsentationsformen (symbolisch, graphisch etc.) [WM06]. Programme stellen eine weitere Möglichkeit dar, Mathematik zu repräsentieren. Die Bedeutung dieser Repräsentationsmöglichkeit hat in akademischen Berufsfeldern auch außerhalb der Mathematik in den letzten Jahrzehnten deutlich zugenommen.

Zweitens erfordert Mathematik und das Erlernen von Mathematik wie jede wissenschaftliche Disziplin den Austausch mit anderen. Aus einer sozial-konstruktivistischen Perspektive betrachtet entstehen Fachkonzepte aus einer Konsensbildung in einer *Community*. Für das Erlernen dieser Konzepte ist Dialog hilfreich, wenn nicht sogar notwendig. Programmieren kann als Dialog aufgefasst werden, als Dialog mit einem gewissermaßen harten Gesprächspartner, der streng auf die Einhaltung vereinbarter (Syntax-)Regeln beharrt. Programmieren in der Mathematikausbildung bietet also den räumlich und zeitlich praktisch unbegrenzt verfügbaren Dialogpartner, der zudem sehr penetrant auf Strenge in der Konversation achtet.

Drittens kann Programmieren ein wertvoller Mediator der kognitiven Prozesse sein, die für das Verinnerlichen und Weiterentwickeln mathematischer Konzepte notwendig sind. In der Tat sprechen kognitionstheoretische Gründe für einen bestimmten Einsatz von Programmieraufgaben in der Mathematiklehre. Abschnitt 8.2 erläutert dies im Detail. Abschnitt 8.3 stellt Aufgabentypen vor, welche besonders für die Programmierung geeignet sind. Abschnitt 8.4 diskutiert technische Aspekte der automatischen Bewertung von Programmieraufgaben. Zum Schluss berichtet Abschnitt 8.5 von unseren Erfahrungen mit dem Einsatz von Programmieraufgaben in einer Lehrveranstaltung zur Diskreten Mathematik. Es sollte noch erwähnt werden, dass aus softwaretechnischer Sicht Programmieraufgaben in der Mathematik nicht anders zu implementieren sind als zum Beispiel Java-Programmieraufgaben. Wir selbst verwenden den Praktomat Grader (Kapitel 10), welchen wir um einen SetlX-Checker erweitert haben, in einer Kombination mit LON-CAPA wie in Kapitel 19 beschrieben.

8.2 APOS-Theorie

Die APOS-Theorie [Arn+13] ist eine Kognitionstheorie, die beschreibt und erklärt, wie mathematische Konzepte gelernt werden. Das Akronym APOS steht für *Action*, *Process*, *Object*, *Schema* und benennt charakteristische, in der Regel sukzessive Stadien im Verständnis eines mathematischen Konzepts. Konzeptverständnis unterliegt also einer Entwicklung. Studierende haben ein mathematisches Konzept nicht einfach verstanden oder eben nicht, sondern Verständnis hat charakteristische, aufeinander folgende Ausprägungen zunehmender Abstraktion.

Diese Ausprägungen sollen im Folgenden sowohl allgemein als auch am Beispiel des Konzepts Funktion an einer konkreten Fragestellung und charakteristischen studentischen Antworten erläutert werden. Die Aufgabenstellung besteht in der Frage, ob es eine Funktion gibt, die aus jeder Buchstabenfolge, die eine Zahl bezeichnet (z. B. einhundertdreiundzwanzig), die entsprechende Ziffernfolge (z. B. 123) gewinnen kann. Typische studentische Antworten auf diese Fragestellung sind von der Art [MR11]:

> Student A1: „Ich würde die Buchstabenfolge zuerst in Zahlworte wie Eins, Zwei, Drei, Hundert, Tausend usw. zerlegen und die Zahlworte dann mithilfe einer Tabelle in Ziffern übersetzen."
>
> Student A2: „Mir fehlt der Eingabewert, der in die Funktion gesteckt wird. Eine Funktion braucht eine Eingabe, den ersten Schritt. Der fehlt mir gerade."
>
> Student P: „Da man die Buchstabenfolgen 1:1 auf Ziffernfolgen abbilden kann, sehe ich da keine Schwierigkeiten."

Alle drei Studierenden bejahen letztendlich die Existenz einer solchen Funktion. Die Art ihrer Antworten weist jedoch auf charakteristische Unterschiede hinsichtlich ihres Verständnisses des Konzepts Funktion hin. Für die Studenten A1 und A2 ist es für die Antwortfindung wichtig, ob es einen konkreten Algorithmus gibt, während Student P alleine mit der Eindeutigkeit als charakteristische Eigenschaft einer Funktion argumentiert.

Im *Action*-Stadium verstehen Studierende ein Konzept als eine Aktion. Aktionen sind wiederholbare Abfolgen von Objektmanipulationen. Manipulierbar sind dabei in der Regel nur konkret erfahrbare Objekte, wie physikalische Objekte oder auf Papier geschriebene mathematische Symbole. Studierende benötigen ein explizites, schrittweises Rezept (wie Studenten A1 und A2) oder eine Formel, die solche Aktionen beschreibt, um ein Konzept anwenden zu können. Einzelne

Schritte können dabei nicht alleine in Gedanken ausgeführt werden oder mental übersprungen werden.

Im *Process*-Stadium dagegen können Studierende die Transformationen ausführen ohne jeden dazu notwendigen Schritt explizit ausführen zu müssen (vgl. Student P). Dafür charakteristisch ist die Fähigkeit die Transformation allein im Geiste ausführen zu können. Dabei müssen keine konkreten Objekte manipuliert werden. Der Mechanismus, der vom *Action*- zum *Process*-Stadium führt, wird in der APOS-Theorie als Internalisierung bezeichnet. Das Verständnis eines mathematischen Konzepts wandelt sich dabei von einer Abfolge extern durchzuführender Aktionen hin zu einem geistigen, internen Prozess. Im Fall des Konzepts Funktion werden Funktionen im *Process*-Stadium als eindeutige Abbildungen von Objekten aus einer Klasse auf Objekte einer möglicherweise anderen Klasse verstanden, ohne dass dazu die Abbildungsvorschrift explizit bekannt sein muss.

Charakteristisch für ein *Object*-Verständnis ist, dass Prozesse zu Objekten werden, die selbst wieder manipuliert werden können. Beispielsweise im Fall des Funktionenkonzepts erfordert das Konzept der Ableitung ein Objektverständnis von Funktionen, denn die Ableitung transformiert eine Funktion in eine neue Funktion. Das Konzept der Ableitung setzt also mit dem *Object*-Verständnis ein ziemlich elaboriertes Verständnis des Konzepts Funktion voraus. Der Mechanismus, der vom *Process*- zum *Object*-Stadium führt, wird als Kapselung bezeichnet.

Das *Schema*-Stadium schließlich ist dadurch charakterisiert, dass ein Konzept mit weiteren mathematischen Konzepten verbunden wird und daraus ein neues Konzept entsteht, das jeweils wieder als *Action*, *Process* und *Objekt* konzeptualisiert werden kann.

Die didaktische Konsequenz der APOS-Theorie besteht darin, dass Studierenden Gelegenheit gegeben werden muss ihr Verständnis mathematischer Konzepte entlang der namensgebenden Phasen weiterzuentwickeln. Studierenden muss also die Gelegenheit zu Internalisierung bzw. Kapselung gegeben werden. Internalisieren von Aktionen zu mentalen Prozessen kann dadurch begünstigt werden, dass Studierende Aktionen wiederholt ausführen und ihr Vorgehen reflektieren. Dies kann dadurch geschehen, dass Studierende Code, der spezifische Berechnungen ausführt, durch Code ersetzen, der die Berechnung für unspezifizierte Werte ausführt. Um beispielsweise die Internalisierung des Konzepts Assoziativität für eine bestimmte Operation $\otimes$ zu unterstützen, kann eine Programmiertätigkeit hilfreich sein, in der Studierende für die Operation $\otimes$ ausgehend von Befehlen wie `(1⊗2)⊗3==1⊗(2⊗3)` und `(0⊗2)⊗42==0⊗(2⊗42)` Programmcode schreiben, der die Assoziativitätseigenschaft nicht nur anhand konkreter Fälle, sondern für alle möglichen Belegungen aus einer Grundmenge überprüft (vgl. auch Abschnitt 8.3.2).

Aus Sicht der APOS-Theorie hat Programmieren in der Mathematik jenseits der in Abschnitt 8.1 genannten Funktionen die wesentliche Funktion die Entstehung mentaler Strukturen bei Studierenden zu unterstützen. Dies erfordert zunächst keine automatisierte Bewertung der von Studierenden erstellten Programme. Wenn Programmieren allerdings zu einem wesentlichen Bestandteil der Mathematiklehre wird und Studierenden durch Programmieraktivitäten die Gelegenheit gegeben werden soll, ihr Verständnis mathematischer Konzepte zu entwickeln und weiterzuentwickeln, muss dies unter anderem an prominenter Stelle, das heißt während der Kontaktzeit, geschehen. Die für die Hochschullehre typischen hohen Teilnehmerzahlen erlauben es jedoch nicht solche kritischen und denkintensiven Aktivitäten von Studierenden in angemessener Intensität zu betreuen. Dabei kann (Teil-) Automatisiertes Feedback wertvolle Unterstützung leisten. Zudem ermöglicht eine automatisierte Programmbewertung Lehrszenarien, die auf regelmäßigem und zeitnahem Feedback basieren [BT11].

8.3 Besonders geeignete Aufgabentypen

In diesem Abschnitt werden Aufgabentypen vorgestellt, welche sich besonders für eine auf APOS-Theorie beruhende Lehre eignen. Normale Rechenaufgaben lassen sich mit gängigen Lernmanagementsystemen einsetzen, indem die Studierenden eine Zahl oder Formel eingeben, welche dann direkt mit dem Ergebnis verglichen wird oder über ein CAS ausgewertet wird. Aus Sicht der APOS-Theorie sind aber Aufgaben interessanter, in denen Studierende mathematisch modellieren und selbst konstruieren, wie zum Beispiel Aufgaben, in denen Studierende eine Datenstruktur (wie beispielsweise eine Menge) definieren. Diese lassen sich zum Teil auch noch mit einem CAS überprüfen. Noch interessanter sind Aufgaben, bei denen ein mathematischer Sachverhalt als Funktion[1] modelliert wird, da damit mathematische Eigenschaften definiert und angewendet werden können. Funktionen können im Allgemeinen nicht mit einem CAS überprüft werden, da es viele verschiedene Lösungsmöglichkeiten geben kann, welche nicht durch einen einfachen Vergleich oder über eine Formel auf Richtigkeit geprüft werden können. Funktionen können aber mit Unit-Tests evaluiert werden auf ähnliche Art wie diese Technik im Software-Engineering eingesetzt wird. Man testet also, ob für eine Auswahl von Eingabewerten die richtigen Ausgabewerte produziert werden. Die Tests müssen gut gewählt sein, damit mögliche Fehler auch tatsächlich gefunden

1 Wir bezeichnen in diesem Kapitel alles als Funktion, was Eingabe- und Ausgabewerte hat, und unterscheiden nicht zwischen Methoden, Subroutinen, Prozeduren, Funktionen und Abbildungen.

werden. Außerdem benötigt man für viele Aufgaben noch weitere Tests, die ausschließen, dass die Studierenden „schummeln“ und zum Beispiel in der Programmiersprache schon vordefinierte Funktionen verwenden oder fundamental andere Lösungswege wählen, als sie es bei einer bestimmten Aufgabe tun sollen. Gegebenenfalls bietet sich auch eine Plagiatsüberprüfung an. Diese stößt aber insbesondere bei kurzen mathematischen Aufgaben dann an ihre Grenzen, wenn es nur eine begrenzte Anzahl verschiedener Lösungsansätze gibt und die Lösungen sich somit zwangsläufig ähnlich sehen.

Verschiedene Programmiersprachen kommen für Mathematikaufgaben in Frage. Ein Vorteil von imperativen Sprachen ist, dass Studierende oft schon damit vertraut sind oder diese parallel lernen. Aus technischer Sicht ist es relativ belanglos, welche Sprache verwendet wird, solange es möglich ist, Tests zu spezifizieren, welche in Batchverarbeitung auf die studentische Einreichung angewendet werden können. Aus pädagogischer Sicht ist es natürlich von Vorteil, wenn die Notation der Programmiersprache möglichst mathematiknah ist. Daher verwenden wir in diesem Kapitel die Sprache SetlX[2], die eine Weiterentwicklung von Jack Schwartz' SETL (Set Language) ist, welche auch sonst in auf APOS-Theorie beruhenden Textbüchern eingesetzt wird [LD95]. Im Prinzip könnte man auch Python verwenden, da sich SetlX und Python recht ähnlich sind. Ein Vorteil von Python ist, dass es sich um eine bekanntere Sprache handelt, für die es auch viele andere Anwendungsmöglichkeiten gibt. Allerdings dürfen Mengen in Python keine „mutablen Objekte“ (wie zum Beispiel Mengen oder Listen) enthalten. Daher kann man in Python Mengen von Mengen nur dann darstellen, wenn die innere Menge als unveränderlich deklariert wird. Außerdem sind die Darstellungsformen von SetlX noch näher an der mathematischen Notation als die von Python. Für eine einführende Mathematikvorlesung ist SetlX daher die geeignetere Sprache.

Im Folgenden stellen wir verschiedene Aufgabentypen vor, die insbesondere in Vorlesungen zur Einführung in die Mathematik und zu Diskreten Strukturen verwendet werden können. Unsere Liste beruht auf eigenen Erfahrungen, auf der Liste von Farahani & Uhlig [FU09] und auf Aufgaben aus Textbüchern, welche die APOS-Theorie einsetzen [LD95].

8.3.1 Mengendefinition durch Set-Comprehension

Mengen können bekanntermaßen entweder durch Aufzählung der Elemente oder durch die sogenannte Set-Builder- (oder Set-Comprehension-) Notation beschrie-

2 http://www.randoom.org/Software/SetlX

ben werden. Zum Beispiel kann die Menge der geraden Zahlen mathematisch durch $\{n \mid n \in \mathbb{Z} \wedge n \bmod 2 = 0\}$ und in der Programmiersprache SetlX durch `{n : n in [1..100]| n % 2 == 0};` beschrieben werden. Ein Unterschied zwischen der mathematischen und der programmiersprachlichen Notation ist, dass in der Programmiersprache nur endliche Mengen verwendet werden können. Ansonsten sind sich die beiden Notationen aber recht ähnlich. Ein wesentliches Einsatzgebiet für Aufgaben, in denen die Studierenden Set-Comprehension Notation verwenden sollen, ist das Üben von logischen Ausdrücken einschließlich von Quantoren (forall und exists). Um die Mengen korrekt zu bilden, müssen die Studierenden logische Ausdrücke richtig anwenden. Tupel (oder Listen) lassen sich ebenso definieren, da sie sich in SetlX von Mengen nur durch die Art der Klammern unterscheiden. Da Listen und Mengen auch summiert werden können, lassen sich auch Folgen und Reihen auf diese Art darstellen.

8.3.2 Mathematische Eigenschaften verstehen

Ein Vorteil des Einsatzes von Programmiersprachen ist, dass sich viele mathematische Eigenschaften (oder Axiome) direkt darstellen und überprüfen lassen. Das Assoziativgesetz kann beispielsweise wie folgt implementiert werden:

```
istAssoziativ := procedure(set,operat){
   return forall(a in set, b in set, c in set|
   operat(a,operat(b,c)) == operat(operat(a,b),c));
};
```

Der Anhang dieses Kapitels enthält für dieses Beispiel eine mögliche Aufgabenstellung und Unit-Tests. Eine komplexere Struktur „istGruppe“ kann dann durch Zusammenfügen der Axiome (Assoziativität usw.) definiert werden. Die Eingaben für die Funktion „istAssoziativ“ sind eine Menge und eine Operation (zum Beispiel `addition := procedure(a,b){return a+b;};`). Die Studierenden können zum einen dadurch, dass sie die Eigenschaften selbst implementieren, sehen, wie diese genau funktionieren. Zum anderen können sie mit Beispielen experimentieren, auf die die Eigenschaften zutreffen oder nicht zutreffen, und somit durch Programmieren ein Gefühl dafür entwickeln, wie sich die Strukturen verhalten. Das Übergeben der Operationen als Parameter fördert außerdem ein abstraktes Verständnis des Operationsbegriffs.

8.3.3 Ein abstrakter Funktionsbegriff

Operationen, die wie im vorherigen Beispiel als Parameter übergeben werden, sind aus Programmiersprachensicht als Funktionen implementiert. Diese Darstellung kann im Unterricht thematisiert werden. Neben mehrstelligen, partiellen, rekursiven und verketteten Funktionen können Funktionen auch anonym definiert, verschachtelt und als Ausgabewert zurückgegeben werden. Damit sind Lambda-Ausdrücke darstellbar und insgesamt ein sehr abstrakter Umgang mit Funktionen möglich. Für mathematische Grundlagenveranstaltungen sind davon vermutlich hauptsächlich die rekursiven Funktionsdefinitionen von Interesse.

Außerdem kann es zumindest für Informatikstudierende hilfreich sein, ein Funktionsverständnis zu entwickeln, welches sowohl mathematische Funktionen umfasst als auch die vielfältigen Anwendungen von Funktionen in Programmiersprachen, die sie schon aus anderen Vorlesungen kennen. Damit kann auch ein Beitrag geschaffen werden, Studierenden zumindest die Möglichkeit zu geben Studieninhalte miteinander zu vernetzen. Die wichtige Aufgabe „Wissensinseln“ zu einem kohärenten Ganzen zu verbinden wird vielleicht selten in Lehrveranstaltungen aktiv unterstützt. Es ist zumindest fraglich, ob Lehre davon ausgehen kann, dass ein Großteil der Studierenden diese Integration ohne Hilfe leisten kann oder ohne Anlass leistet.

8.3.4 Algorithmisches Denken in der Mathematik

Algorithmisches Denken hat in der Informatik vermutlich einen ähnlichen Stellenwert wie logisches Denken in mathematischen Beweisen. Algorithmen lassen sich gut mit automatisch bewerteten Programmieraufgaben üben – für Beweisverfahren, die nicht nur aus Umformen bestehen, ist das schwieriger oder unmöglich. In einigen mathematischen Gebieten, wie zum Beispiel in der Graphentheorie, sind aber Algorithmen auch von Bedeutung. Studierenden können durch das Programmieren von Algorithmen kreativer tätig werden, als das bei den anderen schon vorgestellten Aufgabentypen der Fall ist. Zumindest Informatikstudierende kann man eventuell für Algorithmen eher begeistern als für Beweisverfahren. Außerdem lassen sich durch Algorithmen auch gut Grenzen erfahren. Zum Beispiel ist es bei Primzahlen schon so, dass nur relativ effiziente Algorithmen noch 4-stellige Primzahlen bestimmen können.

8.3.5 Weitere Aufgabentypen

Viele weitere Aufgabentypen sind denkbar. Farahani & Uhlig [FU09] erwähnen Aufgaben zu grundlegenden Abzählverfahren der Kombinatorik. Zum Beispiel kann das Auswählen von drei Elementen aus einer Menge durch drei geschachtelte Schleifen dargestellt werden. Dass beim Auswählen ohne Zurücklegen die Elemente verschieden sein müssen, kann dann über eine if-Abfrage hinzugefügt werden. Ebenso kann über eine Änderung des Datentyps (Liste oder Menge) oder durch Sortierung der Elemente gesteuert werden, ob die Reihenfolge der Elemente beachtet werden soll oder nicht.

In der Programmiersprache Python gibt es durch die Verbindung zum CAS SageMath eine immense Auswahl von vorprogrammierten Funktionen für alle Gebiete der Mathematik. Studierende können das Anwenden solcher Funktionen im Zusammenhang komplexer Problemstellungen üben. In SetlX gibt es nur eine geringere Auswahl vorprogrammierter Funktionen. Daher eignet sich SetlX eher für das Üben mathematischer Grundbegriffe, während Python auch für weiterführende Anwendungen verwendbar ist. Es ist aber nicht schwierig für Lehrende, auch in SetlX selbst Funktionen zu definieren, welche von den Studierenden verwendet werden können, zum Beispiel um graphische Darstellungen zu ermöglichen.

8.4 Die Erstellung von Tests

Wie bereits erwähnt, reicht bei einigen Aufgabentypen ein CAS zur Bewertung aus – bei Funktionsaufrufen benötigt man aber Unit-Tests. Für den Einsatz von Unit-Tests muss den Studierenden der Name der zu testenden Funktion und die Liste der Eingabeparameter vorgegeben werden und vereinbart werden, welche Datentypen die Funktion zurückgeben soll (siehe Anhang). Der Test führt dann die eingereichte Funktion mit den Eingabewerten aus und vergleicht die Ausgabewerte mit den Ausgabewerten einer Musterlösung. Einige Aspekte des Modellierens mathematischer Probleme werden den Studierenden durch die benötigten detaillierten Aufgabenstellungen daher vorenthalten.

Für Lehrende stellt sich die Herausforderung, die Tests so vorzubereiten, dass es tatsächlich für jeden möglichen Fehler eine Überprüfung durch einen Test gibt. Bei einigen Aufgaben ist das leicht, weil man die Eingabewerte so typisieren kann, dass alle Fälle getestet werden können (wobei Extremfälle wie zum Beispiel die leere Menge besonders beachtet werden sollten, da die Studierenden diese oft übersehen). Bei einigen Aufgaben ist das Erstellen guter Tests aber auch schwie-

riger. Wenn zum Beispiel die Gruppenaxiome implementiert werden sollen, benötigt man als Testfälle jeweils Strukturen, die alle Axiome bis auf eins nicht erfüllen, damit man sieht, ob jedes Axiom korrekt implementiert ist. Das Axiom bezüglich inverser Elemente bedingt aber, dass ein neutrales Element existiert. Es gibt also keine Struktur, welche inverse Elemente ohne ein neutrales Element enthält. Ein Beispiel einer algebraische Struktur, die alle Gruppenaxiome außer der Assoziativität erfüllt, sind die Oktonionen. Insbesondere, wenn die Tests durch studentische Hilfskräfte geschrieben werden, kann man aber davon ausgehen, dass diese die Oktonionen nicht kennen und vermutlich den Test für das Assoziativgesetz weglassen werden.

Tests können also aus verschiedenen Gründen unvollständig oder auch fehlerhaft sein. Eine gute Möglichkeit fehlende oder falsche Tests zu entdecken besteht darin, dass man die studentischen Einreichungen zumindest beim ersten Einsatz der Aufgaben auch manuell kontrolliert und aufpasst, ob es Einreichungen gab, die als korrekt bewertet wurden, obwohl sie fehlerhaft waren. Gleichzeitig kann man damit auch feststellen, ob es vielleicht Fehler aufgrund einer unklaren Aufgabenstellung gab, welche verbessert werden sollte. Es empfiehlt sich also, die Aufgaben und Tests einem Entwicklungszyklus zu unterstellen.

8.5 Einsatzszenarien und Erfahrungen

8.5.1 Erfahrungen mit Programmieraufgaben

Wir haben Programmieraufgaben bereits mehrfach in Mathematikveranstaltungen eingesetzt. Die Aufgaben wurden von den Studierenden gut angenommen und in studentischen Evaluationen zumindest nicht als schlechter als traditionelle Übungsaufgaben bewertet. Ein Test auf Plagiate ergab, dass im Durchschnitt 50% der Einreichungen plagiiert sein könnten. Den Studierenden wurde aber nicht explizit verboten, die Aufgaben vor der Einreichung in Lerngruppen zu diskutieren. Außerdem können insbesondere bei einfachen Aufgaben Übereinstimmungen auch zufällig sein. Bei schwierigeren Aufgaben scheint auf jeden Fall mehr plagiiert zu werden als bei einfacheren Aufgaben, da bei solchen Aufgaben die Anzahl der verschiedenen Einreichungen abnimmt, obwohl zufällige Ähnlichkeiten weniger wahrscheinlich sind. Laut einer anonymen Umfrage in einer Veranstaltung (mit 70 Studierenden) besprachen 65% der Studierenden die Programmieraufgaben miteinander, aber kaum jemand gab an, nur abzuschreiben. Wir wissen nicht, inwieweit sich dieses Verhalten vom Verhalten bei der Bearbeitung traditioneller Hausaufgaben unterscheidet, bei denen vermutlich auch abgeschrieben oder in

Lerngruppen gearbeitet wird. In der gleichen Veranstaltung haben wir auch die Klausur, in der die Studierenden auch Programmcode schreiben mussten, analysiert. Die Auswertung ergab, dass 10% der Studierenden, die die Hausaufgaben regelmäßig eingereicht hatten, überhaupt keinen Programmcode schreiben konnten. Es könnte also sein, dass 10% die tatsächliche Quote der Studierenden darstellt, welche die Programmieraufgaben als Lernmethode nicht annehmen. Wir wissen nicht, ob andere Methoden bei diesen Studierenden erfolgreicher wären oder nicht.

Im Allgemeinen zeigte die Klausur eine Korrelation zwischen Klausurnote und Programmierfähigkeit. Im Sinne der APOS-Theorie enthielt die Klausur eine geringe Anzahl von Aufgabenteilen, welche rezeptartig (im *Action*-Stadium) lösbar waren. Da es sich um eine Erstsemesterveranstaltung handelte, gab es auch nur eine geringe Anzahl von Aufgaben, welche ein *Object*-Stadium voraussetzten, und gar keine zum *Schema*-Stadium. Die meisten Aufgabenteile überprüften also das Verstehen mathematischer Konzepte im *Process*-Stadium. Interessanterweise gab es einen Aufgabenteil zum *Object*-Verständnis von Gruppen, welcher mit dem Erreichen der Note 3 korrelierte, und zwar nicht aufgrund der Bepunktung, da diese nicht höher war als für andere Aufgabenteile. Fast alle Studierenden mit Note 3 und fast keine Studierenden mit einer schlechteren Note hatten diesen Aufgabenteil richtig gelöst. Im Großen und Ganzen bestätigte die Klausur damit unsere Erwartungen zur APOS-Theorie und zum Einsatz von Programmieraufgaben.

Für die meisten Studierenden scheint der Einsatz von Programmieraufgaben somit hilfreich oder zumindest nicht nachteilig zu sein. Wir haben aber in der Klausur beobachtet, dass es einerseits einige wenige Studierende zu geben scheint, die mathematische Notation wesentlich besser lesen und schreiben können als Programmcode, aber andererseits auch Studierende, die Programmieren können, aber keine mathematische Notation lesen und schreiben. Es wäre interessant, genauer zu untersuchen, warum das der Fall ist. Möglicherweise gibt es einerseits selbst bei Aufgaben, die auf *Process*-Verständnis zielen, noch Studierende, die diese Aufgaben ohne Verständnis anhand von Regeln und Strategien lösen. Und zwar gibt es bei einigen Themen nur eine begrenzte Art von möglichen Aufgabenstellungen, so dass Klausuraufgaben eventuell vorher behandelten Aufgaben ähneln und trainiert werden können. Andererseits ist vielleicht die imperative Struktur von Programmcode leichter verständlich für einige Studierende als die mathematische Notation. Es gibt also noch viele offene Fragen bezüglich des Einsatzes von Programmieraufgaben mit (oder auch ohne) Berücksichtigung der APOS-Theorie in der Mathematiklehre.

8.5.2 Studentisches Verständnis mathematischer Ausdrücke

Wie in der Einleitung geschildert verstehen wir Programmieren auch als Dialog. Aus dieser Perspektive betrachtet schaffen Programmierübungen für Lehrende die Gelegenheit Studierenden bei solchen Dialogen „zuzuhören“ und dabei herauszufinden, welche Aspekte des Stoffes diesen schwerfallen und womit diese Schwierigkeiten haben. Die Präzision, die eine Programmierspache hinsichtlich des Formulierens mathematischer Sachverhalte fordert, erlaubt es mitunter studentische Schwierigkeiten ebenso präzise wahrzunehmen.

In einer unserer Lehrveranstaltungen ist dieses Diagnosepotenzial beispielsweise bei der folgenden Aufgabenstellung zutage getreten. Studierende sollten mittels SetlX überprüfen, ob eine gegebene Funktion f mit Definitionsmenge $\mathbb{D}$ injektiv ist, ob also für alle $x_1, x_2 \in \mathbb{D}$ gilt, dass aus $x_1 \neq x_2$ auch $f(x_1) \neq f(x_2)$ folgt. Ein merklicher Teil der Studierenden hat als Lösung einen Ausdruck der Art

```
forall (x1 in domain, x2 in domain | f(x1) != f(x2) );
```

formuliert (wobei `domain` hier für $\mathbb{D}$ steht). In diesem Ausdruck ist die erforderliche Bedingung $x_1 \neq x_2$ nicht genannt. Diese Studierenden gehen also potenziell davon aus, dass $x_1 \neq x_2$ automatisch erfüllt ist. Gespräche mit den Studierenden haben diese Vermutung bestätigt wobei sich herausgestellt hat, dass für sie $x_1 \neq x_2$ eine Folge von $x_1, x_2 \in \mathbb{D}$ ist, zum Beispiel weil sie sich vorstellen, dass ein Element aus $\mathbb{D}$ ohne Zurücklegen entnommen wird und x_1 als Wert zugewiesen wird, und daher für x_2 nicht mehr als Wert zur Verfügung stehen kann.

8.5.3 Erfahrungen der Lehrenden

Es ist sicherlich keine neue Erkenntnis, dass die Sprache der Mathematik nicht immer alles im Detail expliziert. Beispielsweise ist eine gängige Formulierung der Assoziativität einer Operation $\otimes$:

$$a \otimes (b \otimes c) = (a \otimes b) \otimes c.$$

Dabei wird nicht explizit genannt, dass dies für alle Werte von a, b und c aus einer ebenfalls nicht benannten Menge gelten muss. Experten können damit umgehen, dass nicht alles explizit genannt ist. Bei Studierenden ist es allerdings zweifelhaft, ob diese den Aussagegehalt verstehen können, ohne dass Lehrende sie darauf aufmerksam machen, was implizit ist und wie man das Implizite herausarbeiten kann.

Uns als Lehrende hat die Formulierung von Programmieraufgaben immer wieder vor Augen geführt, wie viel Implizites in mathematischen Aussagen enthalten

ist. Unabhängig vom tatsächlichen Einsatz der Programmieraufgaben hatten wir so häufig Gelegenheit, potenzielle Schwierigkeiten der Studierenden als Experten selbst wieder wahrzunehmen. Gleichzeitig hat uns dies angeregt, derartige Verständnisschwierigkeiten in der Lehrveranstaltung zu thematisieren.

8.6 Fazit und Ausblick

In diesem Beitrag haben wir verschiedene Aspekte des Einsatzes von automatisch bewertbaren Programmieraktivitäten in der Mathematikausbildung beleuchtet. Wir haben einerseits aus der Perspektive der APOS-Theorie heraus Argumente genannt, die für einen solchen Einsatz sprechen, und andererseits potenzielle Folgen benannt und von ersten Erfahrungen berichtet.

Aus technologischer Sicht besteht eine besondere Herausforderung im Erstellungsprozess der Aufgaben. Das Formulieren der Unit-Tests kann sich auch für elementare Aufgabenstellungen als äußerst herausfordernd erweisen, wenn potenzielle Fehler trennscharf identifiziert werden sollen.

Aus didaktischer Sicht erweisen sich Programmieraufgaben für uns in zweifacher Hinsicht als wertvoll. Zum einen erlauben sie Einblicke in konzeptuelle Verständnisschwierigkeiten der Studierenden. Wenn Lehre auch als Identifikation charakteristischer studentischer Schwierigkeiten gepaart mit Anstrengungen, dabei zu helfen, diese zu überwinden, verstanden wird [Rie14], braucht es Gelegenheiten, stoffbezogene Verständnisschwierigkeiten zu identifizieren. Programmierübungen können dies leisten. Zum anderen hat das Formulieren von Programmierübungen uns sehr deutlich vor Augen geführt, wie sehr mathematische Aussagen in gängigen Lehrbüchern nicht explizit genannte Dinge voraussetzen. Insofern hat uns die Beschäftigung mit Programmieraufgaben unabhängig von deren tatsächlichem Einsatz wertvolle Einblicke in die Herausforderungen gegeben, die mit der Lehre von Mathematik verbunden sind.

Der Aspekt der Wirksamkeit steht für uns dabei nicht, beziehungsweise zu diesem Zeitpunkt noch nicht, im Vordergrund. Aus einer Sicht des *Constructive Alignment* [BT11], also der Wechselwirkung von Lernzielen, Lehrmethode und Prüfung, hat der Einsatz von Programmieraufgaben in Bezug auf die Didaktik für uns einen Reflexionsprozess angestoßen: Die Veränderung in der Lehrmethode hat zu wertvollen Erkenntnissen unter anderem hinsichtlich der Schwierigkeit des Lernstoffs und charakteristischer Probleme von Studierenden mit diesem Stoff geführt. Diese Erkenntnisse fordert uns auf, über unsere Lernziele nachzudenken und zu entscheiden, welches Gewicht diese Aspekte in unseren Lehrveranstaltungen haben sollen.

Anhang: Beispiel einer Programmieraufgabe

Der Aufgabentext muss genau spezifizieren, was implementiert werden soll, einschließlich der Namen von Variablen oder Funktionen, die in den Tests aufgerufen werden. Wie viel sonstige Hilfestellung (mathematische Definitionen, Tipps zur Implementierung) gegeben wird, hängt davon ab, was die Studierenden schon vorher im Unterricht gelernt haben und welche anderen Aufgaben sie schon programmiert haben.

Beispiel einer Aufgabenstellung:

> Eine binäre Verknüpfung $* : A \times A \to A$ auf einer Menge A heißt *assoziativ*, wenn für alle $a, b, c \in A$ gilt: $a * (b * c) = (a * b) * c$.
> Schreiben Sie in SetlX eine Funktion **istAssoziativ**, die (in dieser Reihenfolge) eine Menge und eine Verknüpfung entgegennimmt und entscheidet, ob die Verknüpfung bezüglich der Menge assoziativ ist.
> Tipps: Die Verknüpfung können Sie auch als Funktion definieren, z. B. `add := procedure(a,b) {return a+b;};`. Ein Beispiel eines Funktionsaufrufs, der True zurückgibt ist dann: `istAssoziativ({1,2,3},add)`. Überlegen Sie sich auch weitere Beispiele für Funktionsaufrufe, die jeweils True oder False zurückgeben.

Musterlösung (zum Testen der Tests):

```
istAssoziativ := procedure(set,operat){
 return forall(a in set, b in set, c in set|
 operat(a,operat(b,c)) == operat(operat(a,b),c));
};
```

Für die Tests müssen Beispielmengen und Verknüpfungen definiert werden. Jedes print-Statement enthält zwischen den $-Zeichen einen Ausdruck, der True zurückgibt, wenn der Test bestanden ist, und sonst False. Es ist eine Herausforderung an Autoren, sich genau zu überlegen, welche Arten von Tests wirklich alle möglichen Fehlerquellen abdecken. Die ersten beiden Testfälle enthalten jeweils ein positives und negatives Beispiel der Assoziativität. Im dritten Test wird Assoziativität für einen String-Datentyp getestet, um sicherzustellen, dass die Funktion abstrakt genug definiert ist. Im letzten Testfall wird die leere Menge getestet, da nach unserer

Erfahrung Studierende manchmal Lösungen mit anderen Programmierkonstrukten implementieren, bei denen die Grenzfälle nicht funktionieren.

Testskript:

```
set1 := {1,2,3,4};
set2 := {};
set3 := {"a","b","c"};
add := procedure(a,b) {return a+b;};
minus := procedure(a,b) {return a-b;};
concat := procedure(a,b) {return a+" "+b;};
print("Test Addition:$istAssoziativ(set1,add) == true$");
print("Test Subtraktion:$istAssoziativ(set1,minus) == false$");
print("Test anderer Datentyp:$istAssoziativ(set3,concat) == true$");
print("Test Leere Menge:$istAssoziativ(set2,minus) == true$");
```

Literatur für dieses Kapitel

[Arn+13] Ilana Arnon u. a. *APOS theory: A framework for research and curriculum development in mathematics education*. Springer Science & Business Media, 2013.

[BT11] John Biggs und Catherine Tang. *Teaching for quality learning at university: What the student does*. McGraw-Hill Education (UK), 2011.

[FU09] Ali Farahani und Ronald P. Uhlig. „Use of Python in Teaching Discrete Mathematics“. In: *American Society for Engineering Education Annual Conference and Exposition*. American Society for Engineering Education. 2009.

[Hol95] Judy Holdener. „Calculus&Mathematica: Instructors' Perspectives on Continuing Controversies“. In: *Mathematica in Education and Research* 6 (1995), S. 6–10.

[LD95] Uri Leron und Ed Dubinsky. „An abstract algebra story“. In: *The American Mathematical Monthly* 102.3 (1995), S. 227–242.

[MR11] Philipp Marwan und Peter Riegler. „Entwicklung des Funktionenkonzepts bei Studierenden der Informatik“. In: *Wismarer Frege-Reihe* 2 (2011).

[Rie14] Peter Riegler. „Schwellenkonzepte, Konzeptwandel und die Krise der Mathematikausbildung“. In: *Zeitschrift für Hochschulentwicklung* 9.4 (2014), S. 241–257.

[WM06] Anne Watson und John Mason. *Mathematics as a constructive activity: Learners generating examples*. Routledge, 2006.