Algebra, Logik und Beweisvisualisierung

Florent Schaffhauser, Vincent Voß und Katrin Weiß

Im Wintersemester 2024–2025 beschäftigte sich das studentische Seminar *Illustrating Mathematics* an der Universität Heidelberg mit Algebra, Logik und Beweisvisualisierung. Unter Verwendung des *Lean-theorem-provers* (lean-lang.org) entwickelten die Studierenden dieses Seminars ein mathematisches Spiel, in dem die Spielenden aufgefordert werden, Beweise unter Verwendung der Syntax von Lean zu schreiben. Dabei wurde für das Spiel bereits existierende Infrastruktur verwendet. Der mathematische Teil wurde jedoch vollständig von den Studierenden während des Semesters produziert und implementiert. Dies bot ihnen eine neue Möglichkeit, mit Mathematik zu experimentieren und einen Einblick in das Konzept des Beweises zu gewinnen. Im Folgenden wird erläutert, wie Beweisassistenten funktionieren und wie mathematische Aussagen in Beweisassistenten formalisiert und überprüft werden können. Abschließend wird kurz die Bedeutung dieser Werkzeuge für die zukünftige mathematische Ausbildung diskutiert.

Wie kann man einen mathematischen Beweis darstellen?

Das Konzept eines Beweises steht im Mittelpunkt der mathematischen Praxis. Ein Beweis ist das, was eine Vermutung in einen Satz verwandelt und, was noch wichtiger ist, was unser Verständnis dieses Satzes widerspiegelt und verkörpert [11]. Es besteht jedoch keine allgemeine Übereinstimmung, wie detailliert ein Beweis sein sollte, und infolgedessen stehen Studierende der Mathematik vor einem doppelten Herausforderung: man muss neues Material lernen, *indem man Theoreme beweist*, und gleichzeitig muss man lernen, *was ein Beweis ist und wann er als richtig und vollständig angesehen werden kann*. Es überrascht nicht, dass dies eine ernsthafte Hürde für Studierende darstellt und ein essenzieller Teil der Ausbildung eines jeden Mathematikers und einer jeden Mathematikerin ist. Als professionelle Praktiker und Lehrer fragen wir uns ständig: Gibt es irgendetwas, das wir tun können, um unseren Studierenden zu helfen, einen Beweis besser zu verinnerlichen? Wie sich herausstellt, ist es dank dem Fortschritt in der Ergonomie bestimmter *Softwaresysteme* möglich, eine optimistische und positive Antwort zu geben. Diese *Softwaresysteme* werden als *Beweisassistenten* bezeichnet.

Das Seminar, um das es in diesem Artikel geht, wurde im Rahmen der Aktivitäten des *Heidelberg Experimental Geometry Labs* (HEGL) durchgeführt, einer Komponente der Forschungsstelle Geometrie+Dynamik am Institut für Mathematik der Universität Heidelberg. An dem Seminar nahmen zehn Studierende teil, die sich größtenteils noch in einem frühen Stadium ihres Studiums befanden (2. bis 4. Semester). Neben den praktischen Aspekten des Seminars, die weiter unten ausgeführt werden, steht im Mittelpunkt dieses Experiments die Frage, ob der Einsatz von Beweisassistenten das Potenzial hat, die Art und Weise, wie Mathematik an der Universität unterrichtet wird, zu verändern. Werden in fünfzig Jahren Übungsblätter und Abschlussprüfungen mit solchen Werkzeugen erstellt? Werden Lehrbücher mit Hilfe eines Beweisassistenten geschrieben werden? Es ist noch zu früh, um das zu sagen, aber klar ist, dass Beweisassistenten die Studierenden auf vielfältige Weise ansprechen und sie dazu bringen, neue mathematische und programmiertechnische Fähigkeiten zu entwickeln sowie ihre Lernerfahrung zu verbessern.

Was ist ein Beweisassistent?

Ein Beweisassistent ist ein *Softwaresystem*, das auf einer Programmiersprache basiert. Es wird als Beweisassistent bezeichnet, weil es einen sogenannten *Type-Checker* enthält, dessen Ziel es ist, zu überprüfen, ob die vom Benutzer geschriebenen Programme tatsächlich ihren Spezifikationen entsprechen. Wenn man zum Beispiel eine Liste konstruieren soll und das Ergebnis der Konstruktion eine natürliche Zahl ist, wird der Typprüfer dies nicht akzeptieren und den Benutzer darauf hinweisen. Diese Art der *zertifizierten Programmierung* ist von breiterem Interesse, aber interessant ist, dass sie zur *Formalisierung der Mathematik* verwendet werden kann, wobei wir mit Formalisierung den Prozess der Darstellung mathematischer Konzepte mit Hilfe einer Programmiersprache meinen. Funktionale Programmiersprachen wie Agda, Rocq oder Lean sind dafür besonders gut geeignet, da sie sogenannte *abhängige Typen* enthalten.[1] Ein zentraler Aspekt von Beweisassistenten ist, dass die Kommunikation zwischen dem Benutzer und dem Typprüfer durch die Wahl einer formalen Logik ermöglicht wird. Im Fall von Agda, Rocq und Lean ist die zugrundeliegende Logik eine Erweiterung von Martin-Löfs Typentheorie, die als *Kalkül induktiver Konstruktionen* [7] bekannt ist. Letzteres basiert auf der Arbeit von Thierry Coquand [4], Gérard Huet und Christine Paulin-Mohring [5], und wurde zuerst in dem implementiert, was jetzt der *Rocq-Prover* (rocq-prover.org) geworden ist.

Konkret lässt ein Beweisassistent den Benutzer mathematische Aussagen in Form eines Typs implementieren. Die Typentheorie hat ihren Ursprung in der Arbeit von Bertrand Russell, der Typen als Quantifizierungsbereiche einführte, um das berühmte Paradoxon zu vermeiden, das er im mengentheoretischen Ansatz zu den Grundlagen der Mathematik aufgedeckt hatte. Später wurden Typen von

DOI 10.1515/dmvm-2025-0058

Alonso Church übernommen (und die Typentheorie präzisiert), als er den *Lambda-Kalkül* einführte, der die frühe theoretische Grundlage für die Implementierung funktionaler Programmiersprachen bildet [2]. Typen werden in der Programmierung verwendet, um zu *spezifizieren*, was von einem Programm erwartet wird. Die Verbindung zur Mathematik wird durch die berühmte *Propositionen-als-Typen-Korrespondenz* hergestellt [12], die auf die Arbeit vieler Logiker und Mathematiker zurückgeht und auch als Curry-Howard-Korrespondenz bekannt ist [6]. In dieser Korrespondenz wird erklärt, wie man Typen bildet, die mathematische Sätze *darstellen*. Die entscheidende Erkenntnis ist, dass ein Programm, das eine solche Spezifikation erfüllt, dann per Definition einen Beweis für diesen Satz darstellt. Von diesem Punkt an ist die einzige Zutat, die noch fehlt, um einen Beweisassistenten zu bauen, ein Typprüfer. Das heißt, ein Algorithmus, der in der Lage ist, zu bestätigen, dass der Typ eines bestimmten Programms seiner Spezifikation entspricht. Ein solcher Algorithmus wurde erstmals 1967 von Nicolaas G. de Bruijn vorgestellt, woraus die Programmiersprache Automath [1] hervorging, und kurz darauf folgte das Mizar-System von Andrzej Trybulec (1973).

Für das 2024–2025 HEGL-*Illustrating Mathematics*-Seminar über Logik, Algebra and Beweisvisualisierung haben wir den *Lean-Theorem-Prover* verwendet. Lean ist eine Programmiersprache, die im Jahr 2013 von Leonardo de Moura entwickelt wurde und deren aktuelle Version, Lean 4, im Jahr 2021 eingeführt wurde [9]. Dank der Bemühungen seiner Nutzergemeinde erfreut sich Lean zunehmender Beliebtheit in der Mainstream-Mathematik. Die wichtigste mathematische Bibliothek von Lean namens *Mathlib* (github.com/leanprover-community/mathlib4) ist einer der Gründe für diesen Erfolg. Mathlib war insbesondere bei der formalen Überprüfung eines Satzes von Peter Scholze behilflich (Scholze hatte die Forschungsgemeinschaft herausgefordert, den Beweis zu überprüfen, den er in seinen Vorlesungsunterlagen gegeben hatte [10]). Diese Geschichte wurde bereits ausführlich erzählt, insbesondere von einem ihrer Hauptprotagonisten [3]. Sie ist auch heute noch relevant, weil sie die Fähigkeiten der Beweisassistenten, insbesondere von Lean, als Werkzeuge für die laufende mathematische Forschung unterstreicht. Dies wiederum macht Lean zu einer beliebten Wahl, um Studierende in den Bereich des *formalen Beweises* einzuführen. Genau das haben wir im HEGL-Seminar getan.

Konstruktive Algebra

Ein auffälliges Merkmal dieser Beweisassistenten, das wir bemerken, wenn wir sie zum ersten Mal verwenden, ist, dass sie uns zwingen, die in einer mathematischen Aussage enthaltene *Bedeutung* im Detail zu analysieren. Die formale Logik, die wir bereits erwähnt haben, ist in der Tat eine *konstruktive* Logik: Obwohl es möglich ist, klassische Axiome wie das Auswahlaxiom in unsere Bibliotheken der formalisierten Mathematik einzuführen, ergeben sie sich nicht direkt aus der Syntax des Typisierungssystems. Wenn wir also einen Beweisassistenten öffnen und versuchen, zum Beispiel den Fundamentalsatz der Algebra zu beweisen, kann uns das System helfen, etwas zu erkennen. Die Aussage, dass jedes nichtkonstantes Polynom mit komplexen Koeffizienten mindestens eine Wurzel zulässt, kann durch die Aussage ersetzt werden, dass eine gegenteilige Annahme zu einer Absurdität führt. Der Assistent lässt uns aber sehen, dass die logische Äquivalenz zwischen den beiden Aussagen zwar klassisch, jedoch nicht konstruktiv gilt, wie man oft sagt. Der Unterschied zwischen der klassischen und der konstruktiven Herangehensweise war der Ausgangspunkt für das Seminar, in dem wir zunächst die grundlegende Algebra aus konstruktiver Sicht in Anlehnung an das Lehrbuch von Mines, Richman und Ruitenburg betrachteten [8]. Ein Vorteil dabei ist, dass die Studierenden etwas mathematisch Neues lernen und gleichzeitig der Weg zur Implementierung in eine Maschine verkürzt wird, insbesondere für den Beweis grundlegender Existenzaussagen.

Wenn man mit der Formalisierung von Mathematik in einer Sprache wie Lean beginnt, ist es tatsächlich empfehlenswert, sich mit den Grundlagen der Typentheorie und der funktionalen Programmierung vertraut zu machen und dann zu lernen, diese beiden Fähigkeiten zu kombinieren, um mathematische Aussagen in Form einer Programmspezifikation zu erstellen. Zum Beispiel würde die Programmspezifikation (das heißt der Typ), die dem Fundamentalsatz der Algebra entspricht, den Benutzer auffordern, eine Funktion zu konstruieren, die ein gegebenes nichtkonstantes Polynom mit komplexen Koeffizienten P auf einen Beweis dafür abbildet, dass dieses spezifische P eine komplexe Wurzel hat. Dies ist ein Beispiel für eine so genannte *abhängige Funktion*, bei der der Rückgabetyp der Funktion vom Eingabewert (hier dem Polynom P) abhängt. Dank des Buches von Mines, Richman und Ruitenburg waren die im Seminar betrachteten Beispiele einfacher und wurden von den Studierenden im Allgemeinen gut aufgenommen. Sie hatten Spaß daran, die Theoreme von Kronecker über die subtilen Unterschiede zwischen einem faktoriellen Ring und einem Ring mit eindeutiger Zerlegung in irreduzible Faktoren zu lernen, die beim konstruktiven Denken auftreten (siehe [8, Theoreme 4.8 und 4.9, S. 125–126]).

Wie funktioniert das in der Praxis?

Neben den konstruktiven Aspekten ist ein charakteristisches Merkmal der formalisierten Mathematik der Status von Beweisen, d. h. die Art und Weise, wie Beweise im System formal dargestellt werden. Wie jedes bekannte mathematische Objekt (z. B. natürliche Zahlen, ganze Zahlen, Polynome, Vektorräume usw.) treten Beweise in Lean als *Terme* auf, wobei wir mit Terme Ausdrücke meinen, deren Syntax sie zu Instanzen eines bestimmten Typs macht. Wenn wir beispielsweise mit 1 die natürliche Zahl eins bezeichnen, dann wird $1 + 1$ vom Typprüfer als natürliche Zahl interpretiert. Dies erscheint intuitiv einleuchtend. Weniger klar ist zunächst die Tatsache, dass auch Beweise Terme eines bestimmten Typs sind. Zum Beispiel sollte die Aussage $x = y$ als Typ angesehen werden, und die Terme dieses Typs sind Beweise dafür, dass x gleich y ist. Was bedeutet es also, zu

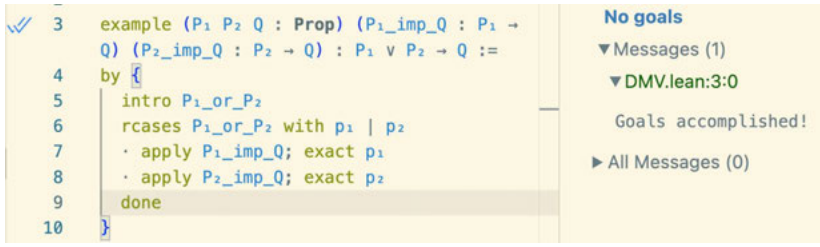

Abbildung 1. Die Eliminationsregel für propositionale Disjunktion (taktischer Beweis)

beweisen, dass der Satz $x = y$ den Satz $y = x$ impliziert? Im Hinblick auf das *Propositions-as-types*-Paradigma bedeutet es einfach, ein Programm zu schreiben, das einen Beweis für $x = y$ als Eingabe annimmt und einen Beweis für $y = x$ als Ausgabe erzeugt. Ein solches Programm ist per Definition eine Funktion vom Typ $x = y$ zum Typ $y = x$, und diese Funktion ist wiederum per Definition ein Beweis für die Implikation $x = y \to y = x$. Wie sich herausstellt, hat dies fundamentale Auswirkungen auf alles, zum Beispiel auf die Definition grundlegender algebraischer Strukturen. Wir können dies bereits bei der Definition einer Halbgruppe sehen, die nicht mehr nur ein Paar (S, m) ist, das aus einer Menge S und einer Operation $m : S \times S \to S$ mit einer abstrakten Assoziativitätseigenschaft besteht. Stattdessen ist eine Halbgruppe ein Tripel $(S, m, m - assoc)$, wobei *m-assoc* ein Beweis dafür ist, dass die Operation m assoziativ ist. Der Punkt ist, dass die Assoziativitätseigenschaft durch den Typ $\forall x, y, z \in S, m(m(x, y), z) = m(x, m(y, z))$ dargestellt wird. Per Definition sind die Terme dieses Typs abhängige Funktionen, die ein Tripel x, y, z von Elementen von S zu einem Beweis der Gleichheit $m(m(x, y), z) = m(x, m(y, z))$ abbilden. Das ist also die formale Bedeutung von *m-assoc*, einer konkreten abhängigen Funktion mit Werten in einem Gleichheitstyp und nicht mehr eine abstrakte Eigenschaft, die zu einer nicht näher spezifizierten Umgebung gehört. Das ist die Art von Überlegungen, die man anstellen muss, wenn man die Mathematik mit einem Beweisassistent formalisiert. Im Seminar konzentrierten wir uns auf alle oben genannten Aspekte, angefangen bei der Logik, grundlegenden algebraischen Strukturen wie Gruppen, Ringen usw. bis hin zur Teilbarkeit in diskreten Integritätsringen im Sinne von Mines, Richman und Ruitenburg (d. h. Integritätsringe mit entscheidbarer Gleichheit: $\forall x, y \in X, x = y \vee x \neq y$). Dies half den Studierenden zu erkennen, dass die Art und Weise, wie wir einen Beweis schreiben, uns manchmal durch die Syntax aufgezwungen wird.

Wir illustrieren an einem Beispiel, wie ein Beweis mit Lean aussehen kann und welche Werkzeuge zur Verfügung stehen. Wenn man beispielsweise beweisen will, dass der Satz $P_1 \vee P_2$ den Satz Q impliziert, genügt es zu beweisen, dass der Satz P_1 den Satz Q impliziert und der Satz P_2 den Satz Q impliziert. Um einen Beweis für den Satz $P_1 \vee P_2 \to Q$ zu konstruieren, nimmt man nämlich $P_1 \vee P_2$ an und versucht, Q abzuleiten. Wenn wir also $P_1 \to Q$ und $P_2 \to Q$ beweisen können, dann haben wir in jedem Fall einen Beweis für Q. In Lean wird das entsprechende Programm wie in Abbildung 1 geschrieben. Dieses Programm wurde unter Verwendung des *Taktikmodus* von Lean geschrieben, der

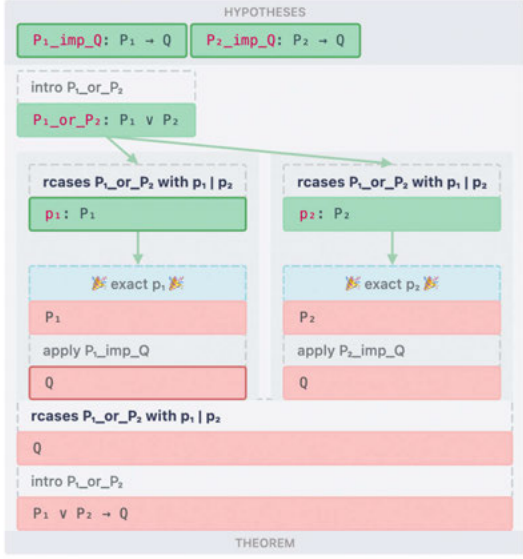

Abbildung 2. Grafische Darstellung eines Beweisbaums mit Paperproof (ein Tool von Evgenia Karunus und Anton Kovsharov)

dem Benutzer hilft, einen Term des richtigen Typs zu konstruieren (d. h. ein Programm, das der gegebenen Spezifikation entspricht), indem er das *Ziel* des Beweises (das der in der Spezifikation angegebene Typ ist) in ein Ziel umwandelt, das wir erreichen können. In der Praxis geschieht dies in der Regel am Ende einer Reihe von Schritten, nach denen der Benutzer jeweils eine Rückmeldung über den aktuellen *Kontext* und das Ziel erhält (im obigen Bild auf der rechten Seite sichtbar). Das resultierende Programm kann anfangs sicherlich etwas trocken aussehen, daher wurden neue Schnittstellen entwickelt, die bei der Visualisierung des entsprechenden Beweistermes helfen. Im Seminar haben wir eine solche Schnittstelle namens *Paperproof* (paperproof.brick.do) verwendet, die von Evgenia Karunus und Anton Kovsharov entwickelt wurde. Paperproof bietet eine grafische Darstellung des *Beweisbaums* eines bestimmten Programms, die dem Benutzer eine dynamischere Präsentation des Beweises ermöglicht. In der Abbildung 2 stellt der untere Teil (in rosa) unser Ziel und der obere Teil (in grün) unseren Kontext dar.

Wenn wir eine Taktik wie `intro` oder `rcases` verwenden, können sich das Ziel und der Kontext ändern. Nachdem wir im Beispiel `intro` $P_1\_or\_P_2$ verwendet haben, ändert sich unser Ziel von $P_1 \vee P_2 \to Q$ zu Q, und ein Term namens $P_1\_or\_P_2$, der vom Typ $P_1 \vee P_2$ ist, wird dem Kontext hinzugefügt. Anschließend führen wir mit `rcases` $P_1\_or\_P_2$ eine Fallunterscheidung für den Typ $P_1 \vee P_2$ durch. Es sind zwei Fälle zu berücksichtigen, je nachdem, ob $P_1 \vee P_2$ von P_1 oder P_2 stammt. Daraus ergeben sich die beiden Verzweigungen. Man beachte, dass das Ziel immer noch Q ist. Wir müssen also Q zuerst aus P_1 und dann aus P_2 beweisen. Dies geschieht, indem wir die Hypothesen anwenden, um unser Ziel für den jeweiligen Fall weiter zu reduzieren, entweder auf P_1 oder P_2. Der Beweis ist vollständig, wenn wir einen Term eingeben können, dessen Typ mit dem Typ des Ziels übereinstimmt, was mit der `exact`-Taktik geschieht. Dies funktioniert, weil der Typprüfer erkennen kann, dass der Typ des nach der `exact`-Taktik eingegebenen Terms mit dem des Ziels übereinstimmt. Genauer gesagt, haben wir im ersten Fall (linke Spalte) unser Ziel auf P_1 reduziert und geben einen Term p_1 an, der tatsächlich vom Typ P_1 ist, womit der Beweis dieses Teiles abgeschlossen ist.

Ein „Heidelberg Lean Game“

Bis zum Ende des Semesters hatten die Seminarteilnehmer alle oben vorgestellten Grundkonzepte erlernt, von der konstruktiven Algebra à la Mines-Richman-Ruitenburg bis zur Verwendung abhängiger Funktionen in der formalen Mathematik. Dann begann der spannendste Teil, der darin bestand, ein mathematisches Lean-Spiel zu schreiben, das online von jedem gespielt werden kann, der mit Lean experimentieren möchte. Dabei profitierten wir von der vorhandenen Infrastruktur des *Lean Game Servers* (adam.math.hhu.de), die durch das Projekt ADAM an der Heinrich-Heine-Universität Düsseldorf (ADAM steht für *Anticipating the Digital Age of Mathematics*) zur Verfügung gestellt wurde.

Da unser Spiel eher klein sein sollte, haben wir uns dafür entschieden, es vor Ort in Heidelberg zu hosten. Dies bietet für uns die Möglichkeit, das Spiel in zukünftigen Sitzungen des Seminars erweitern und z. B. weitere Level hinzuzufügen. Aus unserer Sicht dient das Spiel einem doppelten Zweck: mit Mathematik zu experimentieren und sich mit Lean als Programmiersprache vertraut zu machen. Aber in der Tat ist es für diejenigen, die das Spiel schreiben, auch ein großartiger Einstieg in die Formalisierung, da es erforderlich ist, den geeigneten Ansatz zu finden, um bestimmte mathematische Konzepte darzustellen. Als spezifisches Thema für das Spiel entschieden die Seminarteilnehmer, sich auf die Teilbarkeit zu konzentrieren. Genauer gesagt haben wir den Beweis, dass jede Primzahl irreduzibel ist, in ein Spiel verwandelt und ihn in verschiedene Level aufgeteilt.

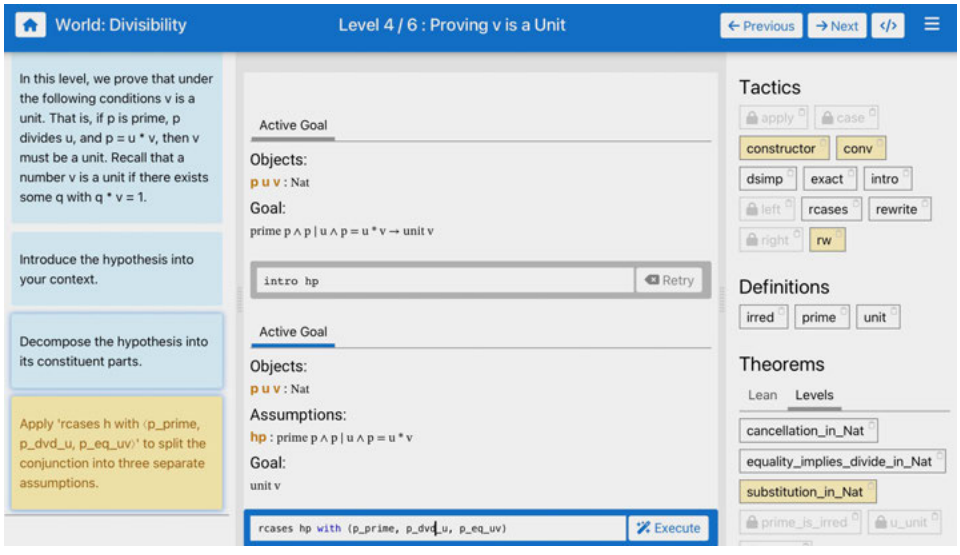

Abbildung 3. Die Benutzeroberfläche des Lean Game Servers (ADAM Projekt, Heinrich-Heine-Universität Düsseldorf)

Auf diese Weise kann die begriffliche Struktur des Nachweises erklärt und eine nützliche Einführung in die Syntax von Lean gegeben werden.

Wir haben schließlich sechs Level geschrieben. In jedem Level bahnt man sich seinen Weg zu einem Teil des Beweises und erhält dabei Zugang zu neuen Taktiken und Funktionen aus der Bibliothek. Man kann auch eine „regellose" Version des Spiels spielen, bei der man von Anfang an auf jede Taktik und jede Funktion zugreifen kann. Jedoch muss man beachten, dass das Spiel mit dem Ziel konzipiert ist, die Funktionsweise der Syntax zu erklären, so dass es in der Tat einfacher ist, am Anfang weniger Wahlmöglichkeiten zu haben. Während man das Spiel spielt, bietet die Benutzeroberfläche Hinweise, die durch den Beweis führen, sowie Tipps, wenn man nicht weiterkommt. In der Mitte des Bildschirms wird der Satz angezeigt, den man beweisen soll. Dort kann die Lösung eingeben werden. Die Erklärungen und Hinweise erscheinen auf der linken Seite (in verschiedenen Farben), während auf der rechten Seite die verfügbaren Taktiken und Funktionen angezeigt werden, die man zur Lösung eines bestimmten Levels verwenden kann. Hier sind auch weitere Erklärungen zu finden, indem man auf den Namen der Taktik oder die Funktion klickt.

Schlussbemerkungen

Die Integration von Beweisassistenten in die universitäre Mathematiklehre steht noch am Anfang, birgt jedoch schon jetzt ein enormes Potenzial. *Interactive Theorem Provers* sind innovative Werkzeuge, die eine einzigartige Möglichkeit bieten, mathematische Konzepte nicht nur zu verstehen, sondern auch aktiv anzuwenden und zu überprüfen. Der Enthusiasmus und das Engagement der Studierenden im HEGL-*Illustrating Mathematics*-Seminar 2024–2025 in Heidelberg zeigen deutlich, dass Beweisassistenten eine wertvolle Bereicherung für den Lernprozess darstellen. Indem wir diese Technologien in den Lehrplan integrieren, können wir den Studierenden eine dynamischere und interaktivere Lernerfahrung bieten, die sie auf die Herausforderungen der modernen mathematischen Forschung vorbereitet.

Anmerkung

1. Aber abhängige Typen sind nicht unbedingt notwendig, um Mathematik zu formalisieren: Andere Beweisassistenten wie Isabelle oder Mizar haben sehr große Bibliotheken mit formalisierter Mathematik und werden bis heute aktiv weiterentwickelt.

Literatur

[1] N. G. de Bruijn, *The Mathematical Language Automath, its Usage and Some of its Extensions*, Symposium on automatic demonstration. Springer, 1970. alexandria.tue.nl/repository/freearticles/597618.pdf

[2] A. Church, A formulation of the simple theory of types. *Journal of Symbolic Logic* 5 (1940) 56–68.

[3] J. Commelin, Liquid tensor experiment. *Mitteilungen der Deutschen Mathematiker-Vereinigung* 30, no. 3, (2022), 166–170. DOI 10.1515/dmvm-2022-0058

[4] T. Coquand and G. Huet, *The Calculus of Constructions*, INRIA Rapport 530 (1986). inria.hal.science/file/index/docid/76024/filename/RR-0530.pdf

[5] T. Coquand and C. Paulin-Mohring, Inductively defined types. In P. Martin-Löf and G. Mints (eds.) *COLOG-88. COLOG 1988*. Lecture Notes in Computer Science, vol. 417. Springer, 1990. DOI 10.1007/3-540-52335-9_47

[6] W. A. Howard, The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley (eds.), *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980, pp. 479–490. [Original paper manuscript from 1969] www.cs.cmu.edu/~crary/819-fo9/Howard80.pdf

[7] P. Martin-Löf, *Constructive Mathematics and Computer Programming*, Studies in Logic and the Foundations of Mathematics, vol. 104. Elsevier, 1982, pp. 153–175.

[8] R. Mines, F. Richman, and W. Ruitenburg, *A course in Constructive Algebra*. Springer, Universitext, 1988. DOI 10.1007/978-1-4419-8640-5

[9] L. de Moura and S. Ullrich, The lean 4 theorem prover and programming language. In A. Platzer and G. Sutcliffe (eds.), *Automated Deduction, CADE 28*, pages 625–635, Springer International Publishing, 2021.

[10] P. Scholze, Liquid tensor experiment. Guest post on *Xena*, December 5, 2020. xenaproject.wordpress.com/2020/12/05/liquid-tensor-experiment/

[11] W. P. Thurston, On proof and progress in mathematics. *For the Learning of Mathematics 15* (1) (1995), 29–37. www.jstor.org/stable/40248168

[12] P. Wadler, Propositions as types. *Commun. ACM* 58 (12) (2015), 75–84. DOI 10.1145/2699407

Prof. Dr. Florent Schaffhauser
Universität Heidelberg, Mathematisches Institut
Im Neuenheimer Feld 205, 69120 Heidelberg
fschaffhauser@mathi.uni-heidelberg.de

Vincent Voß
vincent.voss@stud.uni-heidelberg.de

Katrin Weiß
k.weiss@stud.uni-heidelberg.de

Florent Schaffhauser ist seit 2023 Geschäftsführer des Heidelberg Experimental Geometry Labs *an der Universität Heidelberg. Von 2010 bis 2022 war er Professor für Mathematik an der Universität von Los Andes in Bogotá (Kolumbien) und von 2022 bis 2023 war er Vertretungsprofessor an der Universität Heidelberg. Er interessiert sich für Higgsbündel, Charaktervarietäten, reelle algebraische Geometrie und Homotopietypentheorie.*

Vincent Voß und Katrin Weiß studieren gymnasiales Lehramt im Fach Mathematik an der Universität Heidelberg. Beide interessieren sich sehr für die Verknüpfung von Mathematik und Lehre, die auch in der Schule Anwendung finden kann. Vincent schließt aktuell das Bachelorstudium mit zweitem Fach Spanisch und Erweiterungsfach Englisch ab und wird in Heidelberg in den Master of Education übergehen. Katrin ist am Ende ihres Studiums im Erweiterungsfach Mathematik und beginnt anschließend das Referendariat in den Fächern Mathematik, Wirtschaft und Politik.