Animation strukturierter Beweise in der universitären Ausbildung

genehmigte

D i s s e r t a t i o n

von

Christian Pape

aus Lippstadt

Tag der mündlichen Prüfung:	23. November 1999
Erster Gutachter	Prof. Dr. P. Schmitt
Zweiter Gutachter	Prof. Dr. Th. Ottmann
Dritter Gutachter	Prof. Dr. P. Deussen

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Grundausbildung im Fach Informatik an Universitäten und Hochschulen beinhaltet als einen wesentlichen Teil eine Einführung in die theoretischen Konzepte der Informatik. An der Universität Karlsruhe wird dies durch die Vorlesung „Informatik III“ für Studierende im dritten Fachsemester realisiert. Themenschwerpunkt dieser Vorlesung sind Berechenbarkeit, Komplexitätstheorie und Formale Sprachen. Diese Themen werden betont mathematisch behandelt. Insbesondere spielen neben den dort vorgestellten Algorithmen Beweise eine große Rolle, um Studierenden Wissen aus diesem Bereich zu vermitteln, und sie zu befähigen, mathematisch gesicherte Erkenntnisse zu erarbeiten. Allerdings haben erfahrungsgemäß viele Studierende ernsthafte Probleme, Beweise zu verstehen oder selbst Beweise zu führen; hinzu kommt oftmals ein geringes studentisches Interesse an der Beweisführung.

In der Informatikausbildung spielt der Rechner als Werkzeug zur Vermittlung von Informatik-spezifischem Wissen eine immer größere Rolle: etwa bei der elektronischen Präsentation von Folien und Animationen in einer Vorlesung, bei der Nachbereitung einer Vorlesung oder beim ortsunabhängigen Selbststudium mit interaktiver Lernsoftware am heimischen Rechner. Diesem immer stärkeren Interesse an rechnergestützter Lehre, welches sich auch in einer Fülle staatlich geförderter Projekte widerspiegelt, steht ein Mangel an konkreten Methoden und Konzepten zur Erstellung derartiger Lernsoftware für viele Bereiche der Informatik entgegen.

Eine Ausnahme stellt die Animation von Algorithmen dar, also die dynamische, graphische Visualisierung des Ablaufs eines Algorithmus für Ausbildungszwecke. Sie ist kontinuierlich seit den achtziger Jahren Forschungsthema und eine Reihe von Konzepten und Systemen zur rechnergestützten Erstellung von Algorithmenanimationen ist aus ihr hervorgegangen. Aufgrund mittlerweile technisch fortschrittlich eingerichteter Hörsäle mit Netzanschlüssen für Rechner und festinstallierten „Beamern“, werden Algorithmenanimationen vermehrt in Informatikvorlesungen eingesetzt.

Für die anschauliche Vermittlung mathematischer Konzepte hingegen existieren derzeit nur wenige allgemein anwendbare Konzepte oder Methoden oder gar Autorensysteme, um gezielt zu mathematischen Inhalten rechnergestützt Lernsoftware zu entwickeln. Um das Verständnis von Beweisen zu verbessern, werden in Lehrbüchern

bisher oft bestimmte Aspekte eines Beweises mit zusätzlichen Abbildungen oder Diagrammen veranschaulicht. Diese Vorgehensweise wurde auch für einzelne Beweise in elektronischen Lehrtexten erprobt. Allerdings handelte es sich dabei immer um *Ad-hoc*-Lösungen, welche auf den individuellen Erfahrungen des Lehrenden und Autors beruhen und die nicht auf andere Beweise übertragbar sind. In systematischeren Ansätzen zur Beweisvisualisierung wird derzeit entweder lediglich der Beweistext durch eine betont strukturierte Darstellung lesbarer gestaltet, oder es wird versucht, für eine eingeschränkte Klasse von Beweisen den Beweistext ganz durch statische, visuelle Darstellungen — etwa Venn-Diagramme — zu ersetzen.

1.1 Ziel der Arbeit

In unserer Arbeit stellen wir neue Konzepte und Methoden vor, mit denen gezielt interaktive Lernsoftware für Beweise aus der Theoretischen Informatik entwickelt werden kann. Diese Lernsoftware soll Informatik-Studierenden helfen, ausgewählte Sachverhalte aus einer einführenden Vorlesung in die Theoretische Informatik am heimischen Rechner nachzubereiten. Gegenüber der traditionellen Nachbereitung mit Lehrbüchern und Vorlesungsmitschriften, zeichnet sich unsere Lernsoftware durch folgende Merkmale aus:

- Verwendung animierter Graphik, um die Beweise mit zusätzlicher visueller Anschauung zu vermitteln.
- Verwendung eines hohen Grads an Interaktivität, um Studierenden zum Experimentieren und zur intensiveren Auseinandersetzung mit Beweisen zu motivieren.

Die derzeitigen Ansätze zur graphischen Vermittlung von Beweisen ersetzen den Beweistext ganz durch visuelle Repräsentationen. Im wesentlichen hat dies zur Folge, daß die graphische Darstellung alle Eigenschaften eines Modells der bewiesenen Aussage reflektieren muß. Da dies meist nur für einfache oder endliche Modelle möglich ist, können nur eine sehr eingeschränkte Klasse von Beweisen visualisiert werden. Bei unserem Ansatz hingegen beschränken wir uns auf die graphische Darstellung von endlichen Teilen des Modells. Diese Visualisierung kann dann selbst zwar kein Beweise der Aussage mehr sein, es lassen sich aber sehr viel mehr Beweise graphisch veranschaulichen und es ist möglich, genauere Handlungsanweisungen zu erstellen, um zu einem Beweistext geeignete graphische Darstellungen zu konstruieren.

Im Gegensatz zu den bisherigen Ansätzen der Beweisvisualisierung entwickeln wir ein methodisches Konzept, mit dem eine große Klasse von Beweisen systematisch und mit Rechnerunterstützung visualisiert werden kann. Ein Ziel unserer Arbeit ist deswegen auch die Entwicklung einer Software, welche – ähnlich zu Algorithmenanimationssystemen – dazu verwendet werden kann, um Animationen von Beweisen zu erstellen.

1.2 Kurzüberblick

Wir beginnen unsere Arbeit mit einem kurzen, an zwei Beispielen aus der Theoretischen Informatik orientierten Überblick über Algorithmenanimationen und Systeme zu Erstellung derartiger Animationen. Die dabei vorgestellten Begriffe und Konzepte dienen uns in den darauffolgenden Kapiteln als Basis zur methodischen Erstellung von Beweisvisualisierungen.

In Kapitel 3 stellen wir unser Konzept vor, um für eine in der Theoretischen Informatik wichtige Klasse von Beweisen – Reduktionsbeweise – systematisch interaktive Lernsoftware zu erstellen. Diese kann von Studierenden der Informatik für ein selbstgesteuertes, erfahrungsbetontes Lernen („Learning by doing“) benutzt werden. Dabei stehen die einer Reduktion zugrundeliegenden algorithmischen Lösungsverfahren im Vordergrund, die Beweise selbst spielen noch eine untergeordnete Rolle. Am Ende das Kapitels berichten wir über eine Evaluation einiger Visualisierungen, die von uns im Rahmen der Übungen zur Vorlesung „Informatik III“ eingesetzt wurden.

In Kapitel 4 untersuchen wir den derzeitigen Stand der Forschung von Beweisvisualisierungen für den Ausbildungsbereich. Dazu benutzen wir eine von uns entwickelte Taxonomie zur Klassifizierung von Systemen zur Erstellung von Beweisvisualisierung und ordnen einige repräsentative Systeme grob in unser Klassifikationsschema ein. Das Klassifikationschema selbst stellen wir wegen seines Umfangs detailiert in Kapitel 8 vor.

Nach diesem Überblick stellen wir in Kapitel 5 unser Konzept zur Animation von Beweisen vor, bei der – ähnlich zu einer Algorithmenanimation – die logischen Aussagen und konstruktiven Argumente eines Beweistextes anhand von graphischen Darstellungen dynamisch veranschaulicht werden. Im Gegensatz zur unserer in Kapitel 3 vorgestellten Methodik, steht dabei der Beweistext im Zentrum. Wir zeigen, wie daraus systematisch eine den Beweistext begleitende Animation erstellt werden kann.

Basierend auf unserer Methode haben wir Werkzeuge entwickelt, mit denen sich WWW-basierte Beweisanimationen rechnergestützt entwickeln lassen. Wir stellen unser System — SCAPA — in Kapitel 6 vor. Wir beenden unsere Arbeit mit zwei ausführlichen Fallbeispielen von SCAPA-Animationen zweier Beweise, so wie sie typischerweise in einer Einführung in die Theoretische Informatik vorkommen.

Die von uns visuell umgesetzten Lerninhalte kommen in jeder Grundausbildung Informatik vor und lassen sich deshalb in vielen Lehrbüchern nachlesen. Wir haben bei der Darstellung der Lerneinheiten deswegen auf eine ausführliche Wiederholung der Inhalte verzichtet. Da aber in den Lehrbüchern nicht immer einheitliche Notationen existieren, führen die von uns benutzten Inhalte – so wie sie in der Vorlesung vermittelt werden – in Anhang A auf.

In Anhang 7 geben wir den ausführlichen Beweistext, die bewiesenen Aussagen und einige zusätzliche Erklärungen zum Beweise der beiden Fallbeispielen wieder. Die von uns entwickelten Werkzeuge stellen wir in Anhang C vor und die darauf basierende LaTeX-Stil-Datei und HTML-Erweiterung zur Erstellung der Beweistexte in Anhang D.1. Die Fragebögen der Evaluation gegen wir der Vollständigkeit halber in

Anhang E wieder.

1.3 Hinweis an den Leser

Bei den von uns entwickelten Animationen haben wir sehr oft Farbe benutzt. In dieser Arbeit kommen viele Bildschirmphotos dieser Animationen vor, die im Text mit Bezug auf die verwendeten Farben erläutert werden. Für die Leser, die nur eine reine schwarzweiß-Version dieser Arbeit in den Händen halten, haben wir an entsprechender Stelle im Text in Klammern angegeben, welchen Grauwert die im Text erwähnte Farbe ungefähr in der schwarzweiß-Abbildung besitzt.

Kapitel 2

Algorithmenanimation

Mit einer Algorithmenanimation läßt sich das dynamische Verhalten eines Algorithmus visualisieren, indem während des Ablaufs des Algorithmus graphische Sichten auf dessen Datenstrukturen dargestellt werden. Algorithmenanimationen werden hauptsächlich im Ausbildungsbereich eingesetzt, damit der Lernende die prinzipielle Funktionsweise eines Algorithmus besser erkennen und verstehen kann.

In diesem Kapitel legen wir dar, wie Algorithmenanimationen im Ausbildungsbereich – insbesondere im Bereich der Theoretischen Informatik – eingesetzt werden können und mit welchen Werkzeugen sich diese Animation erstellen lassen. Wir versuchen dabei nicht, einen vollständigen Überblick über alle derzeit verfügbaren Systeme und deren Leistungsmerkmale zu geben. Entsprechende Übersichten finden sich in [FRICK 1998] bzw. [PRICE et al. 1993]. Stattdessen reißen wir nur kurz die historische Entwicklung der bekanntesten Algorithmenanimationssysteme an und beschränken uns auf eine ausführlichere Darstellung eines einzelnen, von uns verwendeten Systems. Doch zuvor grenzen wir den Begriff Algorithmenanimation gegenüber anderen Visualisierungen in der Softwareentwicklung etwas ein. Wir adaptieren diese Begriffe später für eine ähnliche Abgrenzung im Bereich der Beweisvisualisierung.

2.1 Begriffsdefinitionen

Die Begriffe im Bereich der Softwarevisualisierung und deren Bedeutung werden in der Literatur nicht immer gleichbedeutend verwendet. Insbesondere wird die Algorithmenanimation oftmals als Teilbereich der Programmvisualisierung angesehen. Um keine Mißverständnisse über die Verwendung dieser Begriffe in dieser Arbeit aufkommen zu lassen, erläutern wir sie kurz etwas genauer. Wir folgen dabei der Darstellung in [PRICE et al. 1998], siehe auch Abbildung 2.1.

Softwarevisualisierung umfaßt den ganzen Bereich der visuellen Darstellung verschiedener Aspekte von Computer-Software mit Hilfe von Typographie, Graphik und Animationen. Zur Softwarevisualisierung gehört insbesondere die Visualisierung von *Algorithmen*, dies sind abstrakte Beschreibungen von Lösungs- oder Berechnungsver-

fahren, und von *Programmen*, dies sind konkrete Implementierungen von Algorithmen in einer bestimmten Programmiersprache:

> „The differentiation between program and algorithm is subtle and can best be desribed from a user perspective: if the system is designed to educate the user about a general algorithm, it falls into the class of *algorithm visualization*. If, however, the system is teaching the user about one particular implementation of an algorithm, it is more likely *program visualization*“ [PRICE et al. 1998]

Darüberhinaus können natürlich auch andere hier nicht aufgeführte Teile einer Software visuell dargestellt werden, etwa deren Spezifikation mit einem Klassendiagramm in UML (*unified model language*).

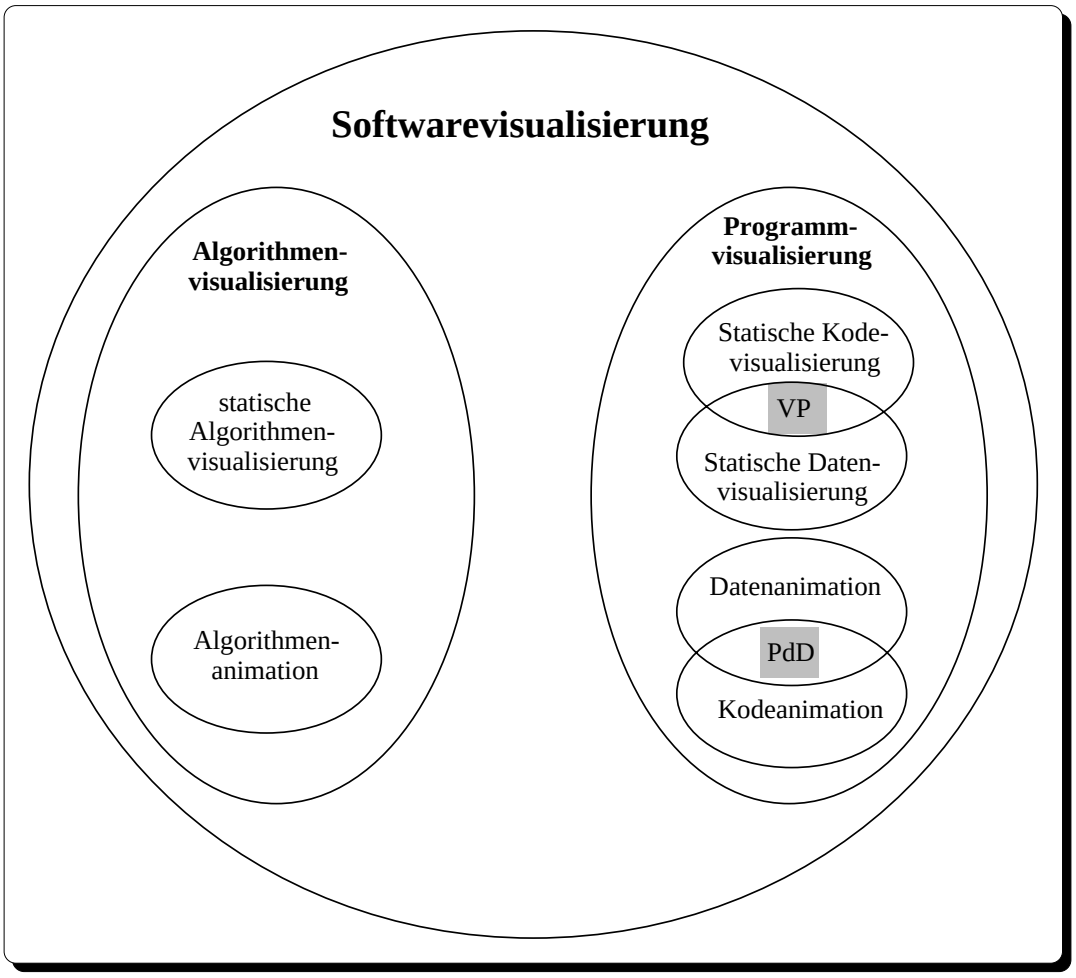


Abbildung 2.1: Zusammenhang zwischen einigen Begriffen aus der Softwarevisualisierung, nach [PRICE et al. 1998].

Algorithmen können statisch, zum Beispiel durch Angabe in Pseudokode, und dynamisch, durch eine Animation der im Algorithmus veränderten Datenstrukturen visualisiert werden. Bei einem Programm unterscheidet man noch genauer zwischen Quellkode und den Daten (Datenstrukturen). Der Kode kann statisch, zum Beispiel durch „pretty printing“, und die Daten, also der interne Zustand des ablaufenden Programms, durch Diagramme, wie etwa einer Darstellung einer verketten Listen mit Kästchen und Pfeilen, visuell dargestellt werden. Auch bei spezifikationsnahen Programmiersprachen wie „state charts“ kann deren graphische Angabe als endlicher Automat als statische Visualisierung angesehen werden. *Visuelles Programmieren* (VP) bezeichnet die Spezifikation eines Programms und dessen Daten mit Hilfe visueller Darstellungen. Dynamische Darstellungen spielen in diesem Bereich keine Rolle. Wird das Programm schrittweise ausgeführt und dabei visuell, zum Beispiel durch Hervorhebung der aktuellen Programmzeile, veranschaulicht, dann spricht man von *Kodeanimation*; werden bei Programmausführung die Änderungen der Datenstrukturen visuell dargestellt, dann spricht man von *Datenanimation*. Das *Programmieren durch Demonstration* (PdD), also die Spezifikation von Programmen und Daten durch Vorgaben von Beispielen durch den Benutzter, kann als Teil der Schnittmenge von Kode- und Datenanimation aufgefaßt werden.

In den folgenden Ausführungen interessieren uns im wesentlichen nur Systeme zur Erzeugung und Darstellung von Algorithmenanimationen.

2.2 Algorithmenanimationssysteme

Abbildung 2.2 zeigt die chronologische Entwicklung bekannter Algorithmenanimationssysteme und die Beziehungen zwischen ihnen. Die durchgezogenen Linien veranschaulichen eine starke Abhängigkeit der verbundenen Systeme, wie sie etwa durch eine evolutionäre Weiterentwicklung eines Systems innerhalb einer Forschergruppe entstanden ist. Die gestrichelten Linien veranschaulichen eine lose Abhängigkeit, bedingt durch wesentliche, gemeinsame Grundmerkmale, wie zum Beispiel die deklarative Beschreibung visueller Darstellungen. Der Übergang vom grau schattierten zum hellen Hintergrund markiert den derzeitigen Trend zu WWW-basierten Algorithmenanimationen mit Implementierungen von Systemen in Java.

Bei den folgenden Systemen wird der Algorithmus meist in einer bestimmten Programmiersprache implementiert, um dann daraus mit den speziellen Merkmalen des verwendeten Systems eine Animation zu erzeugen. Im Gegensatz zu Programmvisualisierungssystemen steht das Programm nicht im Vordergrund, sondern dient nur als Mittel, um eine animierte Sicht des Algorithmus zu generieren.

Die Entwicklung von Systemen zur Erzeugung und Präsentation von Algorithmenanimationen begann etwa 1984 mit BALSA (Brown ALgorithm Simulator and Animator) [BROWN und SEDGEWICK 1984], einem System zur Animation von Algorithmen, die in der Programmiersprache C implementiert werden. Das System unterstützt verschiedene, alternative Sichten auf einen Algorithmus. Mit der Vorstellung

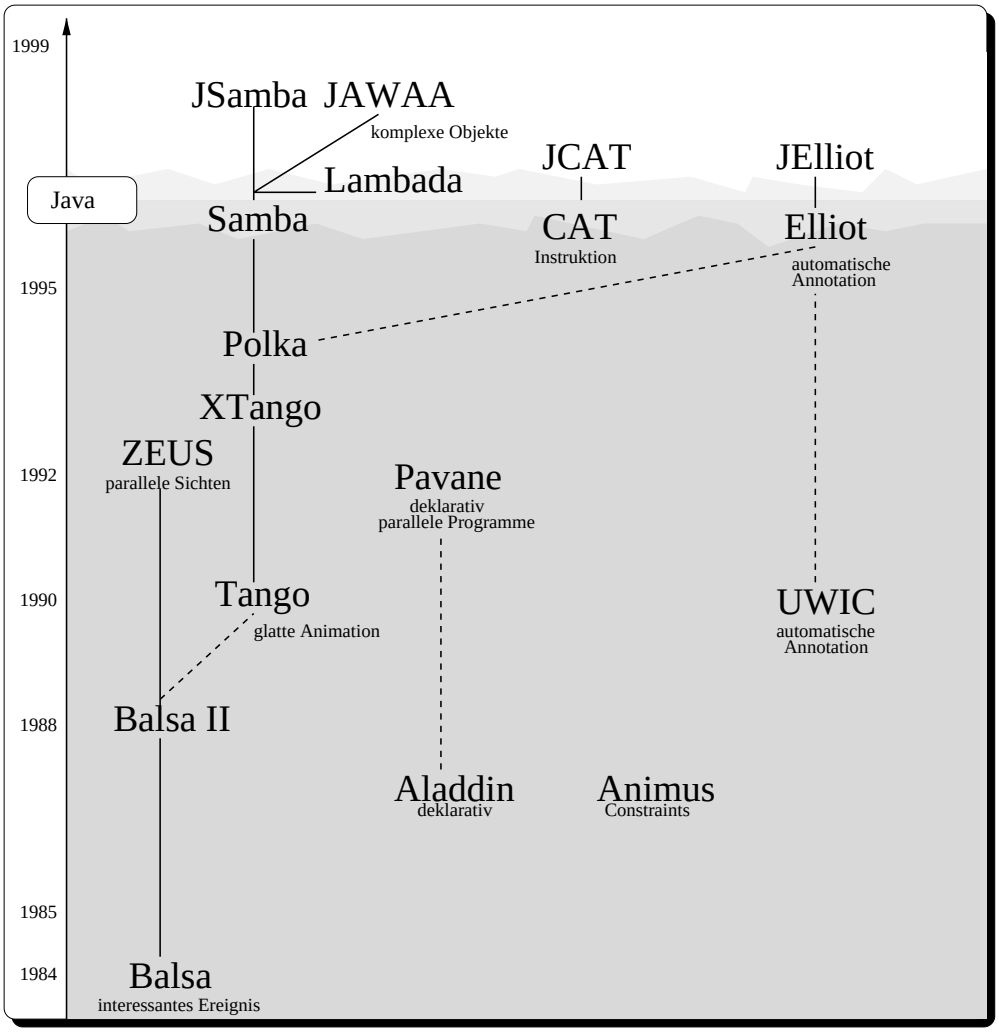


Abbildung 2.2: Historie von Algorithmenanimationssystemen

von BALSA wurde auch der Begriff eines *interessanten Ereignisses*[1] geprägt: Ein interessantes Ereignis ist ein Teil des Algorithmus, der eine für das Verständnis des Algorithmus wesentliche (interessante) Zustandsänderung (Ereignis) bewirkt. Zum Beispiel ist bei Sortieralgorithmen das Austauschen der Werte zweier Variablen ein interessantes Ereignis, während die Zustandsänderung der bei einer Implementierung eventuell dazu benutzte Hilfsvariable nicht interessant ist. Das Folgesystem BALSA-II [BROWN 1988b, BROWN 1988a] ist im wesentlichen eine Re-Implementierung von BALSA für den Apple-Macintosh. Mit BALSA-II können in Pascal implementierte Algorithmen animiert werden. Bei beiden Systemen müssen die interessanten Ereignisse manuell implementiert werden, das heißt, der Programmierer des Algorithmus

[1] Engl.: *interesting event*

wählt die interessanten Ereignisse aus und ergänzt die entsprechenden Programmteile durch Aufrufe geeigneter Funktionen aus einer Animationsbibliothek. Das so erweiterte Programm kann ganz normal übersetzt und gestartet werden. Mit Hilfe der Animationsbibliothek wird bei Ablauf des Programms eine visuelle Darstellung der interessanten Ereignisse erzeugt. Die in BALSA vorgestellten Konzepte werden in ZEUS [BROWN und HERSHBERGER 1992] auf simultane, mehrfache Sichten, der Verwendung von Farben und räumlichen Darstellungen erweitert. ZEUS animiert Algorithmen, die in einer Modula-3 ähnlichen Programmiersprache implementiert werden.

Ein weiterer Ableger der BALSA-Familie ist TANGO [STASKO 1990]. Dieses System unterstützt neben elementaren geometrischen Objekten die kontinuierliche Änderung der Position dieser Objekte entlang eines Pfades[2]. Die Quellprogramme müssen manuell modifiziert werden. XTANGO [STASKO 1992] ist lediglich eine X-Windows Version von TANGO. POLKA ist eine C++-Animationsbibliothek, welche die Funktionsmerkmale von TANGO implementiert. SAMBA [STASKO 1996a] ist auf Basis von POLKA entstanden. SAMBA erhält seine Animationsbefehle aus einem Skript (einem ASCII-Text), interpretiert die Befehle und erzeugt daraus eine Animation. Das Skript kann von jeder Programmiersprache mit Hilfe von Ausgabeanweisungen erzeugt werden. JSAMBA (Java-Samba) ist eine Re-Implementierung von SAMBA in der Programmiersprache Java. SAMBA hat die Entstehung weiterer Java-basierter Algorithmenanimationssysteme angeregt: JAWAA [PIERSON und RODGER 1998] basiert ebenfalls auf einem Animationsskript und unterstützt einen ähnlichen Befehlssatz wie SAMBA, erweitert diesen aber noch durch die Darstellung und Manipulation von Bäumen, Prioritätswarteschlangen, Listen, Schlangen und Graphen. Das System LAMBADA [MITCHENER et al. 1996] ist wie JSAMBA eine Re-Implementierung von SAMBA angereichert mit einigen zusätzlichen Befehlen.

Da die manuelle Implementierung der interessanten Ereignisse und zugehöriger visueller Darstellungen eine sehr zeitraubende Tätigkeit ist, wird dies bei den folgenden Systemen teilweise automatisiert. ANIMUS [DUISBERG 1987-1988] ist ein in Smalltalk implementiertes System, bei dem die visuellen Darstellungen sowie deren zeitliche Änderung mit Hilfe von *Constraints* beschrieben werden. Das System nutzt die Objekt-orientierten Eigenschaften von Smalltalk aus, um die darzustellenden Datenstrukturen (Objekte) über eine allen Objekten gemeinsame Basisklasse `Object` mit einer visuellen Sicht zu verbinden. ALADDIN [KYRSKYKARI und RAIHA 1987] ist ein System, um auf dem Apple-Macintosh in Pascal und Modula-2 Programm implementierte Algorithmen zu animieren. Die Animation des Algorithmus wird auf Basis einer deklarativen Spezifikation der Visualisierung automatisch erzeugt. PAVANE [ROMAN et al. 1992] ist ein weiteres System, mit dem die visuelle Darstellung von Datenstrukturen deklarativ beschrieben wird. Es können in C- oder C++ implementierte Algorithmen animiert werden. Der University of Washington Illustring Compiler (UWIC) [HENRY et al. 1990] visualisiert automatisch elementare Datentypen und Operationen in Pascal-Programmen mit heuristischen Verfahren. Darüberhinaus

[2] Engl.: *path-transition*

visualisiert das System noch die komplexeren Datenstrukturen Liste, Binärbaum und Graph.

ELIOT [LAHTINEN et al. 1996] animiert Algorithmen, die in einer C-ähnliche Programmiersprache (Eliot-C) implementiert wurden, weitgehend automatisch. Zur Erzeugung der Animationen, greift ELIOT auf die Animationsbibliothek von POLKA zu. Das System JELIOT (Java-Eliot) [HAAJANEN et al. 1997] ist eine Klient-Server-basierte Java-Implementierung von ELIOT. Über eine WWW-Schnittstelle kann der Benutzer Java-ähnliche Programme (EJava) eingeben; diese werden an einen Server geschickt, automatisch mit Graphikbefehlen erweitert und die resultierende Animation kann im WWW-Stöberer[3] vom Benutzer betrachtet werden.

Bei allen diesen Systemen hat der Benutzer die volle Kontrolle über den zu animierenden Algorithmus, dessen Eingabe und die Sicht auf ihn. Mit Collaborative Active Textbooks (CAT) [BROWN und NAJORK 1997] werden WWW-basierte Algorithmenanimationen innerhalb eines elektronischen Klassenzimmers zur Instruktion der Schüler durch einen Lehrer benutzt. Dabei wird der Algorithmus und dessen Eingaben vom Lehrer vorgegeben, die Schüler können lediglich unterschiedliche Sichten auf diesen Algorithmus produzieren. Die Algorithmen sind in der verteilten Programmiersprache Obliq implementiert [CARDELLI 1995]. Diese eignet sich besonders gut zur Implementierung des vornehmlich netzbasierten Systems CAT. Da CAT aufgrund der Verwendung von Obliq nur mit speziell angepaßten und nicht mit handelsüblichen WWW-Stöberern verwendbar ist, wurde eine eine Java-basierte Version von CAT entwickelt: JCAT [BROWN und RAISAMO 1996]. Die Algorithmen sind statt in Obliq in Java implementiert und die Kommunikation zwischen Server (Lehrer) und Klienten (Schülern) ist mit Javas Remote Method Invocation (RMI) realisiert.

Wie diese chronologischen Zusammenfassung zeigt, ist ein deutlicher Trend hin zu Java- und WWW-basierten Algorithmenanimationen zu erkennen. Desweiteren zeigt sich auch, daß Systeme, bei denen der Algorithmus in einer vorgegebenen Programmiersprache implementiert werden muß, meist recht kurzlebig sind und keinen sonderlichen Einfluß auf zukünftige Systeme ausüben.

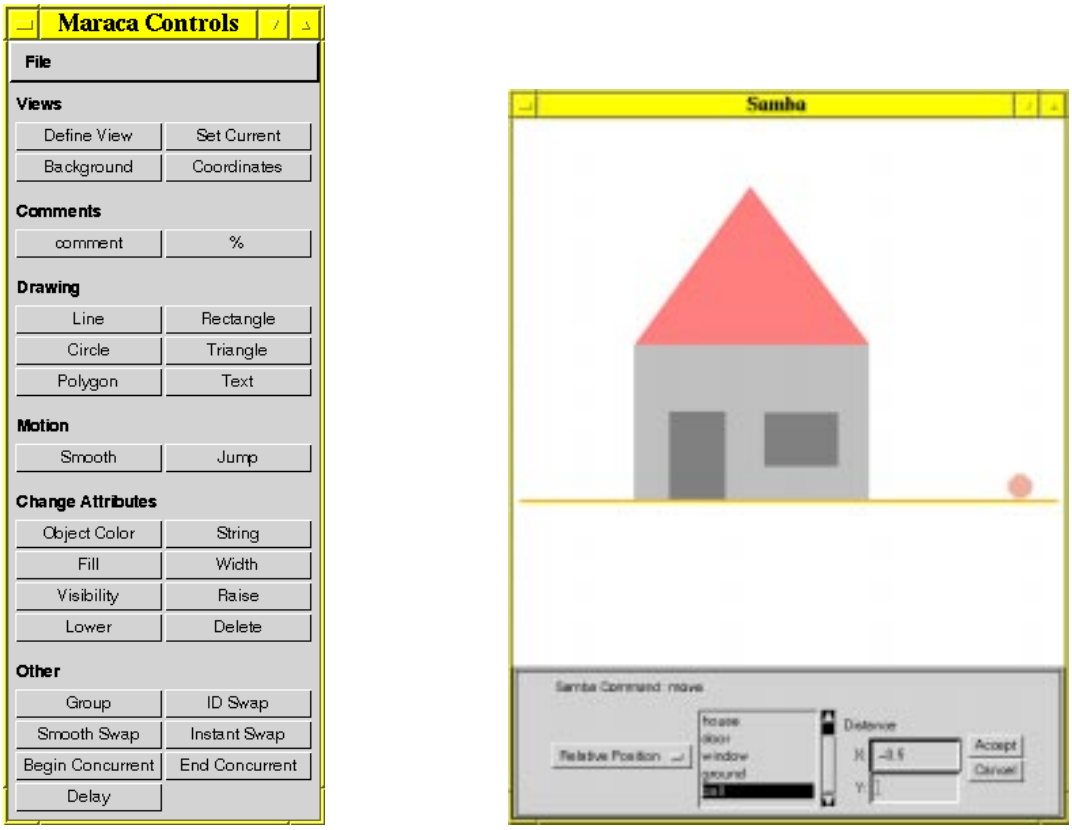
Wir betrachten im folgenden das Animationssystem SAMBA, einen Vertreter der BALSA-Familie, etwas genauer, weil es uns als Grundlage für die Erstellung einiger Algorithmenanimationen aus der Theoretischen Informatik dient. Wir stellen SAMBA hier auch vor, weil wir dessen Java-basierte Re-Implementierung LAMBADA als Basis zur Erstellung von Beweisanimationen in unserem System SCAPA verwenden. LAMBADA und SAMBA besitzen keinen auf die Animation von Algorithmen beschränkten Satz von Animationsbefehlen. Sie eignen sich daher auch für die Erstellung von Beweisanimationen.

2.2.1 Das Algorithmenanimationssystem Samba

SAMBA stellt eine einfache interpretierbare Sprache zur Verfügung, mit der Anima-

[3] Engl.: *WWW-Browser*

tionen beschrieben werden können. SAMBA ist in C++ implementiert und bildet die interpretierten Befehle auf die Animationsbibliothek von Polka ab. Die Befehle dieser Sprachen können als Ausgabe eines Programms einer beliebigen Programmiersprache erzeugt werden. Dazu muß der Autor die zu animierenden Programme von Hand geeignet mit Ausgabebefehlen erweitern. Man kann aber auch ohne Implementierung eines Algorithmus, durch direkte Angabe von Animationsbefehlen in eine Datei mit einem Texteditor ein Animationsskript erzeugen. Um diese Vorgehensweise komfortabel zu gestalten, ist SAMBA ein Java-Programm – MARACA – beigefügt, mit dem eine Folge von SAMBA-Befehlen über eine graphische Benutzungsschnittstelle eingegeben werden kann (Abbildung 2.3(a)). Die Animation wird dabei angezeigt und kann schrittweise erweitert werden (Abbildung 2.3(b)).



(a) Steuerung

(b) Animation

Abbildung 2.3: Erzeugen von SAMBA-Skripten mit MARACA: Die einzelnen Kommandos von SAMBA werden über die Steuerung von MARACA angewählt werden (a); im Animationsfenster werden die Parameter des Kommandos bestimmt und die Auswirkungen des letzten Kommandos lassen sich sofort betrachten (b).

SAMBA kennt Befehle zum Erzeugen von Linien, Polygone, Rechtecken, Kreisen, Dreiecken und Text. Deren Attribute, wie Farbe oder Position, lassen sich im Laufe einer Animation über einen eindeutigen Bezeichner ändern. Darüberhinaus erlaubt SAMBA die Darstellung dieser Objekte in verschiedenen Fenstern. Abbildung 2.4 zeigt ein mit MARACA erzeugtes Animationsskript.

```
triangle roof 0.4 0.872 0.146 0.588 0.688 0.588 red  half
rectangle base 0.148 0.29 0.534 0.298 gray  half
rectangle door 0.218 0.294 0.122 0.17 gray  solid
rectangle window 0.432 0.36 0.156 0.102 gray  solid
pointline ground 0.024 0.288 0.974 0.288 orange  medthick
circle ball 0.888 0.32 0.028425341 pink  solid
moverelative ball -0.5 0.0
```

Abbildung 2.4: Beispiel eines Animationsskripts für SAMBA

Wird SAMBA mit einem Animationsskript gestartet, so erscheint ein Steuerungsfenster („Polka Control Panel“) mit dem der Benutzer die Animation anhalten („Stop“), schrittweise ausführen („Step“) oder beenden („Quit“) sowie die Geschwindigkeit der Animation regeln kann („Slow“–„Fast“). Die Animation selbst erscheint in ein oder mehreren Fenstern („SAMBA“). Der Bildschirmbereich der Animation kann in alle vier Richtungen verschoben oder verkleinert werden (Abbildung 2.5).

Abbildung 2.5: Ausführen eines Animationskripts mit SAMBA

SAMBA ist gegenüber anderen Systemen flexibler einsetzbar. Es können Programme nahezu jeder Programmiersprache animiert werden. Voraussetzung ist, daß die Programmiersprache eine textuelle Ausgabe erzeugen und in eine Datei umleiten kann. Auf Basis der primitiven geometrischen Objekte lassen sich beliebige Datenstrukturen visualisieren. Dies ist allerdings auch ein wesentlicher Nachteil, da komplexe Animationen und visuelle Darstellungen mühsam aus den vorhandenen Objekten konstruiert werden müssen. Die Produktivität des Autors ist deswegen recht gering. Mit Systeme wie JELIOT können im Gegensatz dazu sehr schnell Animationen entwickelt werden. Dazu werden im wesentlichen die Datenstrukturen und Operationen automatisch während der Programmausführung in vorgefertigte graphische Darstellungen abgebildet. Dadurch hat der Autor dann allerdings oftmals wenig Einfluß auf die Gestaltung der Animation. Im Ausbildungsbereich kann dies sehr nachteilig sein, da man sich oft spezielle, auf den Algorithmus abgestimmte visuelle Darstellungen wünscht, um die Eigenschaften des Algorithmus besser vermitteln zu können.

2.3 Beispiele

In diesem Abschnitt geben wir zwei Beispiele von Algorithmenanimation zu typischen Themen aus der Theoretischen Informatik. Die Algorithmen sind von uns in Perl oder Java implementiert und manuell mit Animationsbefehlen für das Algorithmenanimationssystem SAMBA erweitert worden. Über eine HTML-basierte Schnittstelle kann der Benutzer Eingaben an den Algorithmus tätigen und das Animationsskript anfordern. SAMBA muß auf Klient-Seite installiert und als Hilfsanwendung[4] über die *MIME-types* im benutzten WWW-Stöberer integriert worden sein. Nach Anforderung des Animationsskripts, startet der Klient SAMBA mit dem Skript (Abbildung 2.6).

Diese Vorgehensweise ähnelt der Mocha-Architektur [BAKER et al. 1995], bei der ebenfalls eine Algorithmenanimation auf einem Server ausgeführt wird. Aber anstatt ein Animationsskript auf einmal zu übertragen, werden einzelnen Animationsbefehle über Unix-Ports sofort an den Klienten übermittelt. Außerdem kann der Benutzer auch während der Animation Daten eingeben, die an den Server geschickt und dort vom Algorithmus verarbeitet werden. Bei Mocha ist der Klient keine Hilfsanwendung, sondern ein Java-Applet, welches ebenfalls vom Server über das Netz geladen wird. Unser Ansatz hat den Vorteil, auf eine komplizierte Port-Anbindung zu verzichten und statt eines speziellen Protokolls für die Übermittlung der Animationsbefehle, das Animationsskript selbst dem Klienten zu übermitteln. Desweiteren entfällt die Implementierung eines Java-Applets, welches die Animationsbefehle empfängt und interpretiert.

Die im folgenden vorgestellten Algorithmenanimationen sind in einem HTML-basierten Hypertext enthalten, der von uns zur Nachbereitung der Studierenden der Vorlesung „Informatik III“ entwickelt wurde. Die Vorlesung richtet sich an Studierende der Informatik an der Universität Karlsruhe im dritten Fachsemester und führt

[4] Engl.: *Helper-Application*

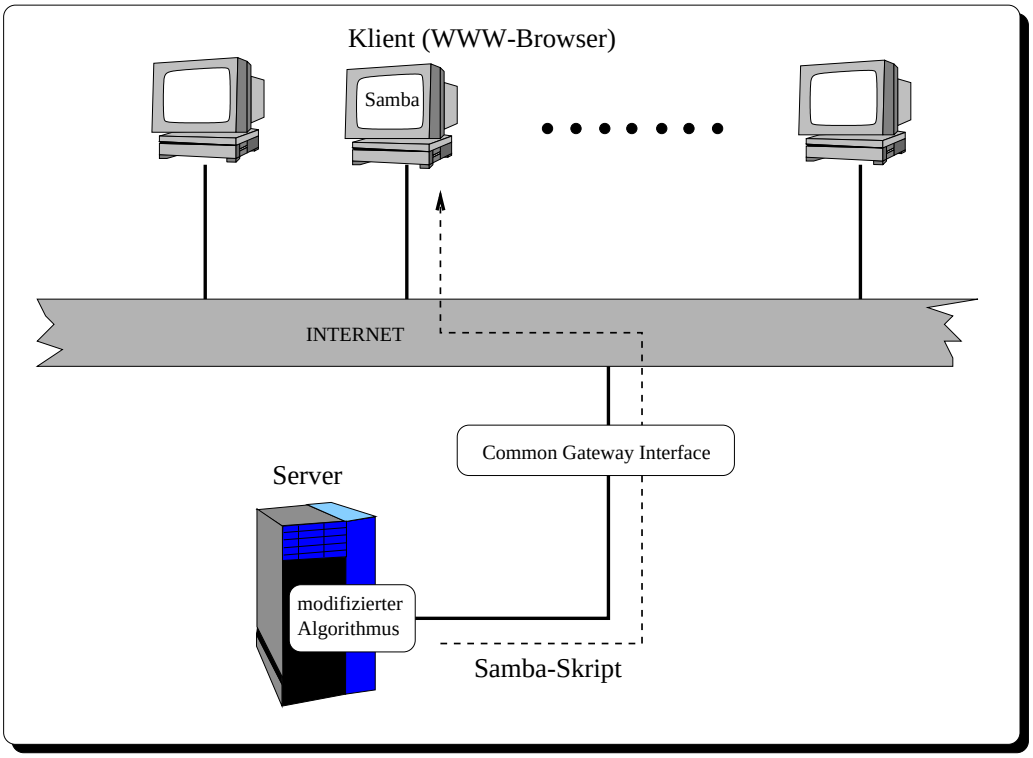


Abbildung 2.6: Anbindung von SAMBA an einen WWW-Stöberer

in die Themen Berechenbarkeit, Komplexitätstheorie, Automatentheorie und Formale Sprachen ein. Der Hypertext ist auch als Vorlesungsskriptum in einer „normalen“ Papierversion erhältlich. Als erstes Beispiel betrachten wir eine Animation des Cocke-Kasami-Younger-Algorithmus. Im Verlauf der Vorlesung wird auch dessen Korrektheit bewiesen. Eine Animation dieses Korrektheitsbeweises wird uns noch in einem späteren Kapitel begegnen. In Abschnitt 2.3.2 betrachten wir dann am Beispiel der Konstruktion kanonischer LR-Parser, wie Algorithmenanimationen dazu benutzt werden können, wenig algorithmisch dafür aber mathematisch und deklarativ beschriebene Verfahren anschaulich zu vermitteln.

2.3.1 Cocke-Kasami-Younger-Algorithmus

Der Cocke-Kasami-Younger- (CKY-)Algorithmus wird dazu benutzt, um in die Begriffe und Verfahren der Syntaxanalyse einzuführen. Er hat gegenüber den im Übersetzerbau eingesetzen Verfahren den Vorteil, noch sehr einfach und für beliebige kontextfreie Grammatiken in Chomsky-Normalform anwendbar zu sein. Er eignet sich deswegen besonders gut, um in die Begriffe und Konzepte des Parsings einzuführen. In der folgenden Darstellung des CKY-Algorithmus setzen wir die Vertrautheit mit einigen Begriffen aus dem Bereich der Formalen Sprachen, wie sie etwa in [MOLL et al. 1988]

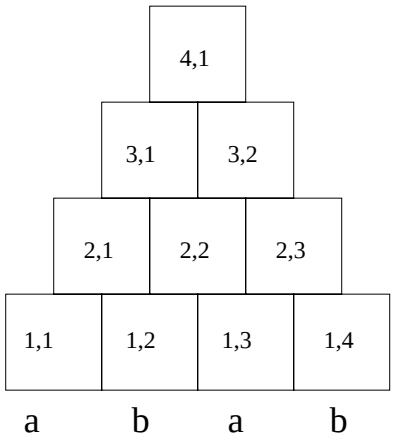


Abbildung 2.7: Beispielpyramide für den Cocke-Kasami-Younger-Algorithmus

vermittelt werden, voraus.

Der CKY-Algorithmus entscheidet zu gegebener, kontextfreien Grammatik G in Chomsky-Normalform[5] und Wort w, ob w in der von G erzeugten Sprache $L(G)$ enthalten ist. Es wird dabei folgende „bottom-up“ Strategie verfolgt: Ausgehend von w werden in einem ersten Schritt alle Nichtterminalzeichen bestimmt, aus denen alle Teilwörter der Länge 1 von w ableitbar sind. Mit Hilfe dieser Nichtterminalzeichen, bestimmt man dann alle Nichtterminalzeichen, aus denen alle Teilwörter der Länge 2 von w ableitbar sind, usw., bis die Menge M aller Nichtterminalzeichen bestimmt worden ist, aus denen w selbst ableitbar ist. Es gilt folgender Satz:

Theorem 1 *Es gilt $w \in L(G)$ genau dann, wenn das Startsymbol in M liegt.*

Die dem Algorithmus zugrundeliegende Datenstruktur läßt sich anschaulich als Pyramide darstellen. Die einzelnen Zellen der Pyramide korrespondieren zu bestimmten Teilwörtern von w und enthalten nach Terminierung des Algorithmus jeweils alle Nichtterminalzeichen, aus denen das zugehörige Wort ableitbar ist. Für die Formulierung des Algorithmus numeriert man die Zellen systematisch mit Paaren von Zahlen. Für das Wort $abab$ ergibt sich zum Beispiel eine Pyramide mit numerierten Zellen, wie sie Abbildung 2.7 zeigt.

Im Skriptum zur Vorlesung wird der Algorithmus wie folgt beschrieben und mit einem Beispiel – welches wir nicht präsentieren – erläutert.

Definition 1 (Cocke-Kasami-Younger-Algorithmus)
Gegeben sei eine kontextfreie Grammatik $G = (N, T, S, P)$ in Chomsky-Normalform und ein Wort $w = w_1 \dots w_n \in T^*$. Wir beginnen mit der Erzeugung einer Pyramide, deren Basis aus n Zellen besteht, unter welche von links nach rechts die Buchstaben

[5] G ist ϵ-frei und jede Produktion hat entweder die Form $X \to a$ oder $X \to YZ$.

Fehlerart	absolute Häufigkeit	relativ
Richtige Lösung	48	55.17%
3. Ebene zuviel	13	14.94%
3. Ebene zuviel/ 4. Ebene leer	8	9.20 %
3. und 4. Ebene leer	10	11.49%
total falsch	8	9.20%

Tabelle 2.1: Häufigkeiten von Fehlerarten bei einer Klausuraufgabe zum Cocke-Kasami-Younger-Algorithmus

von w geschrieben werden. Die Zellen der Pyramide werden nach folgender Vorschrift schrittweise mit Mengen von Nichtterminalzeichen von G gefüllt:

1. *Basis der Pyramide:* Für $1 \leq j \leq n$ schreibe $A \in N$ in die Zelle $(1, j)$ genau dann, wenn es eine Produktion $A \to w_j$ in P gibt.

2. *Weitere Reihen:* Alle Reihen kleiner gleich i seien schon eingetragen. Bestimme die Einträge in Zelle $(i + 1, j)$ für $1 \leq j \leq n - i$ wie folgt: Für $1 \leq m \leq i$ betrachte alle Paare X_1X_2 mit X_1 aus Zelle (m, j) und X_2 aus Zelle $(i - m + 1, j + m)$. Trage A in Zelle $(i + 1, j)$ ein, falls $A \to X_1X_2$ in P vorkommt.

Es gilt $w \in L(G)$ genau dann, wenn S in Zelle $(n, 1)$ vorkommt.

Probleme beim Verständnis des CKY-Algorithmus

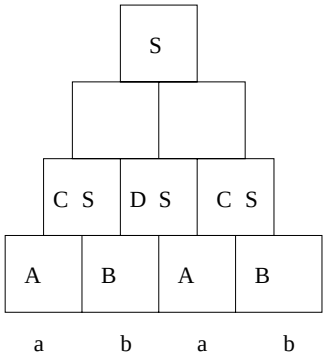
Bevor wir uns der Animation des Algorithmus zuwenden, belegen wir anhand von Daten einer Klausuraufgaben, daß die wesentlichen Probleme beim Verständnis des CKY-Algorithmus (bei einer traditionellen statischen Präsentation) in der richtigen Zuordnung der Zellen bestehen.

Bei der genaueren Untersuchung der Lösungen zu einer Aufgabe in der schriftlichen Prüfung zur Vorlesung „Informatik III“ im WS 96/97 an der Universität Karlsruhe zeigte sich, daß 39 von 89 (43,82%) der Studierenden, die diese Aufgabe bearbeitet haben, den Algorithmus nicht korrekt durchführen konnten. Die Studierenden mußten in dieser Aufgabe zeigen, daß $abab \in L(G)$ für die kontextfreie Grammatik

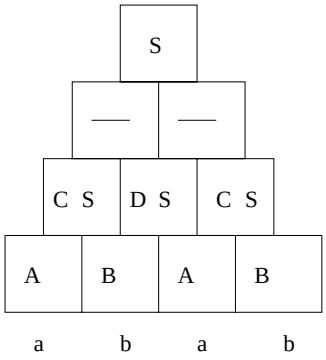
$$G = (\{S, A, B, C, D\}\{a, b\}, S, P) \text{ mit}$$

$$\begin{aligned} P = \{ \quad & S \to CS \mid DS \mid SC \mid SD \mid AB \mid BA, \\ & C \to AB, D \to BA, A \to a, B \to b \qquad \} \text{ gilt.} \end{aligned}$$

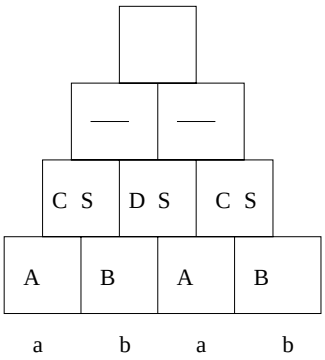
Dazu war die vorgegebene Pyramide entsprechend auszufüllen. Abbildung 2.8 zeigt die dabei am häufigsten gemachten Fehler. Die Fehlerhäufigkeiten sind in Tabelle 2.1 angegeben.



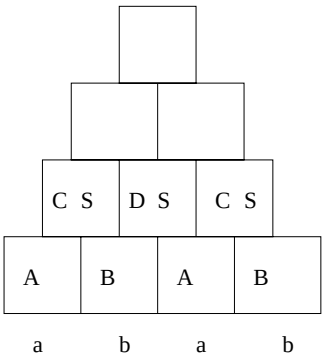
(a) Richtige Lösung



(b) 3. Ebene zuviel



(c) 3. Ebene zuviel und 4. Ebene leer



(d) 3. und 4. Ebene leer

Abbildung 2.8: Klassifikation von Fehlern bei einer Klausuraufgabe.

Die mit Terminalsymbolen ausgefüllten Kästen zeigen einen korrekten Eintrag, ein Minuszeichen zeigt eine fehlerhafte ausgefüllte Zelle. Pyramide 2.8(a) repräsentiert die einzig korrekte Lösung, bei der lediglich die 3. Ebene leer ist, die anderen drei Pyramiden repräsentieren häufig gemachte Fehler.

Bei Pyramide 2.8(b) ist die 3. Ebene fälschlicherweise ausgefüllt, alles andere ist aber richtig. Bei Pyramide 2.8(c) ist zusätzlich noch die 4. Ebene leer geblieben. Da man recht einfach durch „Hinsehen“, also ohne Ausführung des Algorithmus herausfinden konnten, ob $abab \in L(G)$ gilt, kann es sein, daß bei Pyramide 2.8(b) einige Studierenden „gemogelt“ haben und deswegen die obere Zelle korrekt ausfüllten. Wenn dies der Fall ist, dann sind sie eigentlich dem Fehlertyp 2.8(c) zuzuschlagen. Bei Pyramide 2.8(d) ist die 4. Ebene fälschlicherweise leer geblieben. Da bei allen diesen Studierenden (79 von 87, 90.80%) die 1. und 2. Ebene richtig ausgefüllt war, kann man davon ausgehen, daß sie – im Gegensatz zu acht völlig falschen, abgegebenen

Lösungsversuchen – den Algorithmus gelernt und ihn zumindestens teilweise verstanden haben. Viele Studierenden schienen sich allerdings davon verwirren zu lassen, daß die komplette 3. Ebene keine Einträge enthält. Dies läßt den Schluß zu, daß sie die korrekte Zuordnung der Zellen zu den Teilwörtern nicht wirklich begriffen haben. Die obige deklarative Beschreibung des Algorithmus, aber auch eine stärker algorithmische Darstellung, kann den genauen Ablauf der Zuordnung der Zellen zueinander nur implizit reflektieren. Zwar wird der Algorithmus in fast allen Lehrbüchern mit ein oder zwei Beispielen veranschaulicht. Aber schon bei kleinen Beispielen kann nicht jeder Zustand des Algorithmus dargestellt werden. Die folgende von uns entwickelte Animation des CKY-Algorithmus hingegen verdeutlicht die korrekte Zuordnung der Zellen sehr viel besser als eine statische Darstellung.

Animation des CKY-Algorithmus

Abbildung 2.9 zeigt einen Ausschnitt aus der Animation für die Grammatik und das Wort aus der Klausuraufgabe: Die aktuellen Zelle $(i+1, j)$ wird während der Animation gelb (hellgrau in der Abbildung) hervorgehoben. Die zugehörigen Zellen (m, j) und $(i-m+1, j+m)$ für $m = 1, \ldots, i-1$ werden der Reihe nach ebenfalls mit gelber Farbe markiert. Die Farbe signalisiert den Test, ob für jedes m und jede Produktion $A \to X_1 X_2$ die Nichtterminalzeichen X_1 in Zelle (m, j) und X_2 in Zelle $(i-m+1, j+m)$ enthalten sind. Ist dies der Fall, dann werden die Zellen mit grüner Farbe markiert (positiver Test, dunkelgrau in der Abbildung) und A in Zelle $(i+1, j)$ eingetragen; ist dies nicht der Fall, dann werden die Zellen mit roter Farbe markiert (negativer Test).

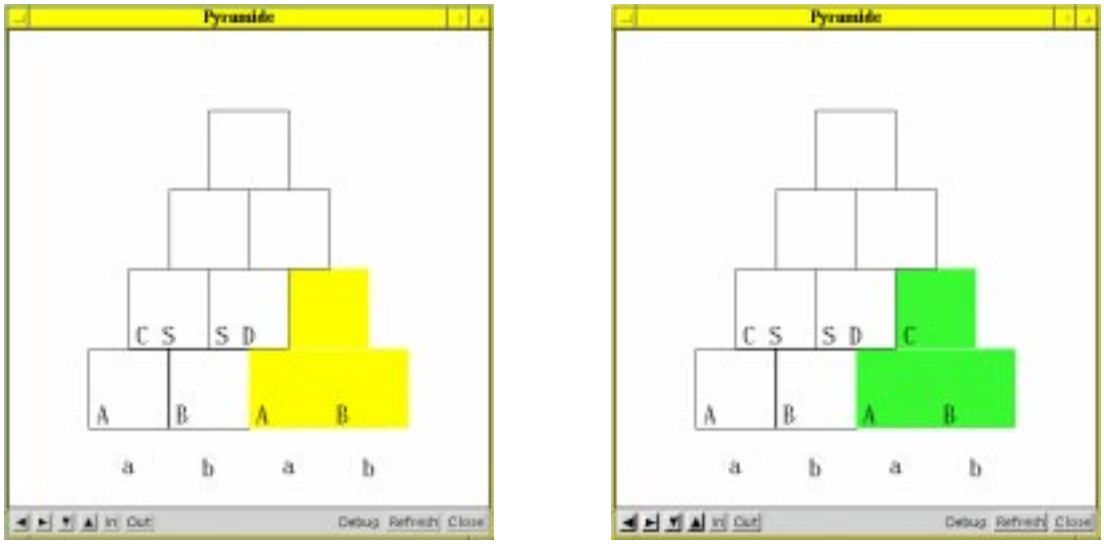


Abbildung 2.9: Animation des Cocke-Kasami-Younger-Algorithmus

Abbildung 2.10 zeigt alle drei Kombinationen für die obere Zelle $(4, 1)$. Bei der Animation wird besonders gut die genaue Zuordnung der Zellen mit dem schon bearbeiteten beiden Teillösungen und der gerade aktuellen Zelle sichtbar.

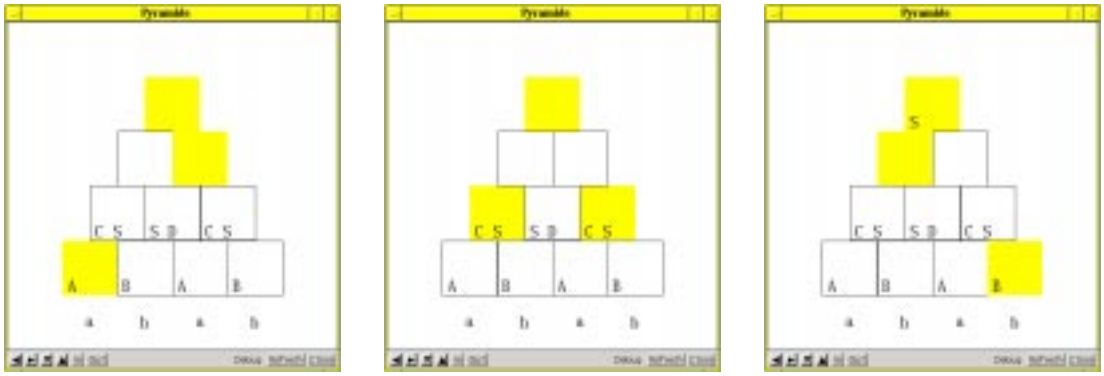


Abbildung 2.10: Zuordnung der Zellen bei der Animation des Cocke-Kasami-Younger-Algorithmus

Neben der vorgegebenen Grammatik, kann das Programm auch für eigene Beispiele genutzt werden. In der HTML-Version des Skriptums kann der Studierende über ein HTML-Formular eine Grammatik in Chomsky-Normalform, dessen Startsymbol und ein Terminalwort eingeben, um anschließend eine Algorithmenanimation des CKY-Algorithmus anzufordern.

Durch das Bearbeiten von Aufgaben oder Beispielen mit dieser Animation, erhält der Lernende eine sofortige Rückmeldung, ob die eigene Vorstellung vom Algorithmus korrekt ist. Bestimmte Fälle im Algorithmus, die nicht richtig verstanden wurden, lassen sich so vom Studierenden eigenmotiviert ergründen. Mit der Animation können auch sehr viel größere Beispiele bearbeitet werden, als das, wegen der Größe der resultierenden Pyramide, mit Papier und Bleistift in einem vertretbaren Zeitaufwand möglich ist. Gerade bei einer großen Pyramide fällt die korrekte Zuordnung und dessen Symmetrie noch viel stärker ins Auge, siehe Abbildung 2.11.

2.3.2 LR-Parsing

Neben dem Cocke-Kasami-Younger-Parsingalgorithmus, wird in der Vorlesung auch das für den Übersetzerbau wichtige LR-Parsing vorgestellt. Die Darstellung während der Vorlesung ist betont abstrakt und formal gehalten, um neben einer bloßen Erklärung des Verfahrens auch noch dessen Korrektheit beweisen zu können. Die Beweise sind im folgenden nicht aufgeführt, da uns nur das Verfahren selbst und dessen Präsentation interessiert. Die mathematischen Notationen erschweren es vielen Studierenden erfahrungsgemäß, die Konstruktion von LR-Parsern nachzuvollziehen oder selbst auszuführen. Deswegen haben wir einige Algorithmenanimationen erstellt,

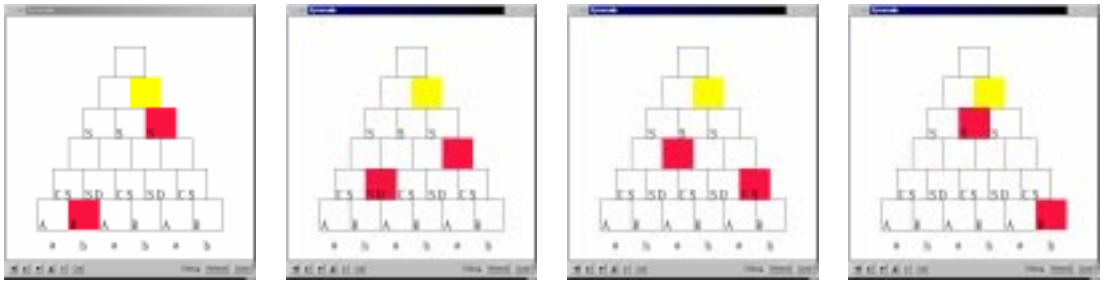


Abbildung 2.11: Die vier Zuordnungen zu Zelle $(5, 2)$ bei der Animation des Cocke-Kasami-Younger-Algorithmus für das Wort $ababab$ und für die Grammatik aus der Klausuraufgabe.

mit denen die Studierenden die Konstruktion eines LR-Parsers aus einer Grammatik Schritt für Schritt nachvollziehen und ausprobieren können. Die visuelle Darstellung hilft ihnen dabei, die abstrakten Notationen und formalen Begriffe mit konkreten Vorstellungen zu verknüpfen. Wie bei der Animation des CKY-Algorithmus, können sie eigene Beispiele eingeben. Die genauen Definitionen zum LR-Parsing, können von einem mit dem LR-Parsing nicht vertrauten Leser in Anhang A nachgelesen werden.

Verständnisprobleme beim LR-Parsing

Das LR-Parsing selbst ist ein algorithmischer Vorgang und die Ausführung einer LR-Kellerableitung mit Hilfe einer gegebenen LR-Parsertabelle zu einer Grammatik verursacht erfahrungsgemäß den Studierenden keine allzu großen Schwierigkeiten, wie auch unser Untersuchung zum CKY-Algorithmus in Abschnitt 2.3.1 zeigt, daß die Mehrheit der Studierenden ein gelerntes Verfahren korrekt durchführen können und die anderen zumindestens ein solides Grundverständnis des CKY-Algorithmus besitzen.

Im Gegensatz dazu verursacht die obige mathematische, deklarative Konstruktionsbeschreibung von kanonischen LR-Parsern den meisten Studierenden erhebliche Verständnisschwierigkeiten, da kein Verfahren algorithmisch angegebenen ist, um aus einer Grammatik über die LR-Kellermaschinen eine LR-Parsertabelle zu konstruieren. Die deklarative Beschreibung ist gewählt worden, da die darauf aufbauenden Korrektheitsbeweise eine wesentliche Rolle im Curriculum bilden: Die Studierenden sollen lernen, derartig beschriebene Algorithmen zu verstehen, um später vielleicht selbst welche zu entwerfen und um sich von deren Korrektheit mathematisch zu überzeugen.

Durch eine zusätzliche, stark algorithmische Beschreibung des Konstruktionsverfahrens, könnten vermutlich die studentischen Verständnisprobleme gemildert oder ganz ausgeräumt werden. Allerdings bekommt man dann zwei unterschiedliche Darstellungen: Eine, die den Studierenden hilft, LR-Parser zu konstruieren, und eine, die zum Führen von Beweisen benutzt wird.

Um zwei unterschiedliche Darstellungen zu vermeiden, haben wir mehrere Animationen erstellt, die auf einer Implementierung des Konstruktionsverfahrens beruhen.

Die Animationen können von den Studierenden benutzt werden, um die deklarativen Beschreibung besser zu verstehen und um sich selbst eine algorithmische Vorstellung zu bilden. Es existieren schon einige Programme zur Visualisierung von LR-Parsern. Diese ließen sich für unsere Zwecke leider nicht einfach übernehmen, da Implementierungen, wie etwa in [KHURI und WILLIAMS 1994, KHURI und SUGONO 1998] berichtet, sich im wesentlichen auf die Visualisierung des Parsing-Vorgangs und nicht der Konstruktion der Parsertabelle konzentrieren. Andere Lernprogramme unterstützen zwar die interaktive Konstruktion eines LR-Parsers aus einer Grammatik, aber nicht eines *kanonischen* LR-Parsers, sondern einer Variante [RODGER 1994]. Der aus dem Übersetzerbau bekannte LR-Parser-Generator YACC wird zwar in der Vorlesung beispielhaft vorgestellt, er eignet sich aber aufgrund fehlender visuellen Anschauung nicht zur Vermittlung des Konstruktionsverfahrens. Darüberhinaus erzeugt YACC keinen kanonischen LR-Parser, sondern eine weitere Variante: einen LALR-Parser.

Animation

Wie die Algorithmenanimation des Cocke-Kasami-Younger-Algorithmus werden die Animationen zum LR-Parsing mit Hilfe eines HTML-Formulars angefordert, siehe Abbildung 2.12.

Die Grammatik wird durch Angabe ihrer Produktionen in einem Textfenster eingegeben, als Startsymbol wird implizit S verwendet. Mit einem Auswahlschalter[6] kann der Benutzer den Parameter k einstellen, zur Auswahl stehen $k = 0$ und $k = 1$. Der Benutzer kann drei verschiedene Animationen anfordern:

- Mit dem Schalter „Indet. LR(k)-Maschine aus Grammatik“ wird eine Animation angefordert, welche die Konstruktion der nichtdeterministischen LR(k)-Kellermaschine zeigt.
- Mit dem Schalter „Det.LR(k)-Maschine“ wird die deterministischen LR(k)-Kellermaschine angezeigt.
- Mit dem Schalter „Parser-Tabelle aus det.LR(k)-Maschine“ bekommt man eine Animation, welche die Konstruktion der LR(k)-Parsertabelle aus der deterministischen LR(k)-Kellermaschine und der Grammatik zeigt.

Zusätzlich kann der Benutzer eine Postscript-Datei der nichtdeterministischen und deterministischen LR(k)-Kellermaschine anfordern („Indet.LR(k)-Maschine-Postscript“ und „Det.LR(k)-Maschine-Postscript“). Aufgrund ihrer Qualität eignet sich die Postscript-Darstellung wesentlich besser zum Überprüfen von eigenen (Papier)lösungen – zum Beispiel einer Übungsaufgabe – als die Animation am Bildschirm.

Das Konstruktionsverfahren ist in Java implementiert. Wie bei der Animation des CKY-Algorithmus, liefert das Programm über das *Common gateway interface* ein SAMBA-Animationsskript an den aufrufenden WWW-Stöberer. Das Layout der in der

[6] Engl.: *radio button*

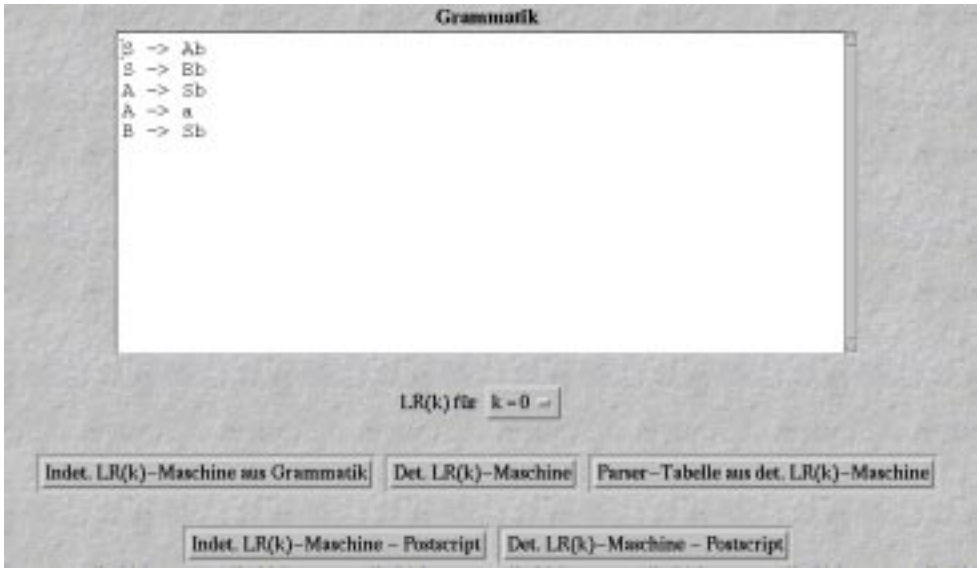


Abbildung 2.12: HTML-Formular zum Starten der Animationen zum LR-Parsing

Animation dargestellten endlichen Automaten wird auf Server-Seite mit dem frei verfügbaren Programmpaket „GraphViz“ [ELLSON et al. 1998] erzeugt. Es eignet sich besonders gut dafür, da es uns erlaubt, Einfluß auf die relative Anordnung der Knoten zu nehmen, und weil es das Layout für Graphen mit 20–30 Knoten in nur wenigen Sekunden berechnet. Dadurch ergibt sich (außer dem Laden des Animationsskript) keine merkliche Wartezeit bei der Anforderung einer Animation. Wir benutzen „GraphViz“ auch, um die Postscript-Versionen der Automaten zu produzieren.

Wir beschreiben die Animation am Beispiel der Grammatik

$$G = (\{a, b\}, \{A, B, S\}, S, \{S \to Ab, S \to Bb, A \to Sb, A \to a, B \to Sb\})$$

für $k = 1$ (siehe auch Beispiel 21).

Nichtdeterministische LR-Kellermaschine Abbildung 2.13 zeigt den Beginn der Konstruktion der nichtdeterministischen LR(1)-Kellermaschine.

Ausgehend vom Anfangszustand (dem LR(1)-Item $[S' \to \cdot S\$]$, gekennzeichnet durch einen eingehenden kurzen Pfeil) wird der Automat schrittweise um die Folgezustände mit LR(k)-Items und Übergängen erweitert (Definition 14). Die dabei noch nicht bearbeiteten Zustände sind weiß und mit einem cyan-farbigen Rand markiert (hellgrauer Rand in der Abbildung bei den Zustände mit LR(1)-Item $[S \to \cdot Bb, \$]$ und $[S \to \cdot Aa, \$]$). Bereits vollständig bearbeitete Zustände besitzen einen weißen Hintergrund und schwarzen Rand. Der gerade aktuelle Zustand (Anfangszustand $[S' \to \cdot S\$]$, der Punkt und das folgende Zeichen S sind hervorgehoben, der nicht interessierende Rest tritt durch das helle Grau in den Hintergrund), ist mit gelber Hintergrundfarbe

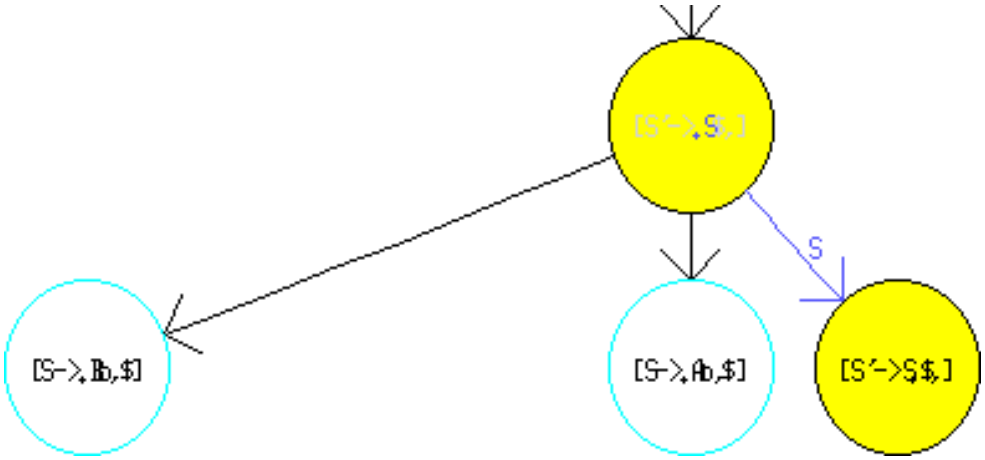


Abbildung 2.13: Anfang der Animation

markiert (hellgrau schattiert in der Abbildung). Der aktueller Übergang von diesem Zustand zu einem Folgezustand ist mit blauer Farbe hervorgehoben (dunkles Grau in der Abbildung), der Folgezustand ebenfalls mit Gelb (Transition vom Anfangszustand mit Zeichen S in den Zustand mit LR(1)-Item $[S' \rightarrow S \cdot \$]$). Das dabei eingelesene Symbol S hinter dem Punkt im LR(1)-Item des Anfangszustand ist ebenfalls blau markiert und durch die helle Schattierung der restlichen Teile des LR(1)-Items zusätzlich hervorgehoben. Wenn zum aktuellen Zustand alle Folgezustände berechnet werden, wird der nächste noch nicht bearbeitete Zustand betrachtet. Die zur Konstruktion benötigte Grammatik wird in einem separatem Fenster angezeigt (nicht dargestellt). Die Produktionen sind dort entsprechend der verwendeten bzw. erzeugten LR(k)-Items farblich hervorgehoben. Abbildung 2.3.2 zeigt die resultierende nichtdeterministische LR(1)-Maschine am Ende der Animation.

Deterministische LR-Kellermaschine Die Konstruktion der det. LR(k)-Kellermaschine wird nicht animiert, sondern lediglich angezeigt. Das Verfahren, um aus einem nichtdeterministischen Automaten (der nichtdeterministischen LR(k)-Maschine) einen äquivalenten deterministischen Automaten (der deterministischen LR(k)-Maschine) zu konstruieren ist Teil der in der Vorlesung behandelten Automatentheorie.

Für die wichtigsten Algorithmen aus der Automatentheorie wird vorlesungsbegleitend ein Applet angeboten, welches – wie die Algorithmenanimationen – im Hypertext integriert ist. Der Benutzer kann sich dort zu Automaten das Ergebnis diverser Algorithmen schrittweise anzeigen lassen, oder selbst eine Lösung eingeben und diese automatisch Schritt für Schritt überprüfen lassen. Dabei werden die eigenen Eingaben farblich mit den korrespondierenden Teilen des Automats verknüpft. Das Applet basiert auf Version 2.0 von JFlap [RODGER, GRAMOND und RODGER 1999], welches wir für unsere Zwecke angepaßt und um zusätzliche Verfahren erweitert haben [RESCH 1998]. Abbildung 2.15 zeigt einen Ausschnitt aus dem Programm zum Teilmengenkonstruk-

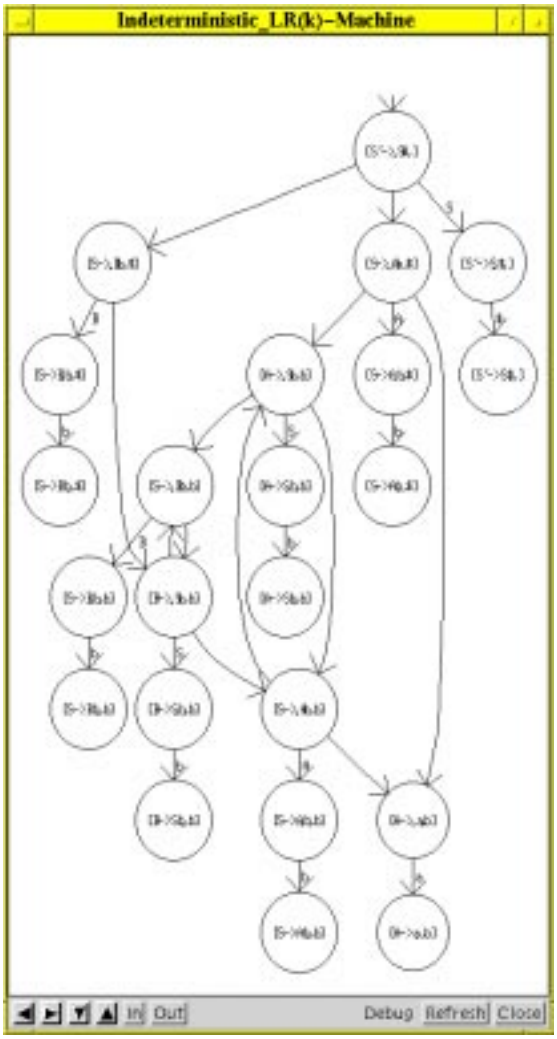


Abbildung 2.14: Ende der Animation

tionsverfahren, mit dem zu einem vorher eingegebenen nichtdeterministischen endlichen Automaten ein äquivalenter deterministischer endlicher Automat schrittweise berechnet und visualisiert wird.

Konstruktion der LR-Parser-Tabelle Die Animation des Konstruktionsverfahrens zur LR-Parsertabelle zeigt jeweils in einem Fenster die Grammatik, die dazugehörige deterministische LR(k)-Maschine und die zu erstellende LR(k)-Parsertabelle. Die Zustände der deterministischen LR(k)-Maschine sind durchnumeriert, sie dienen als Zeileneinträge der LR(k)-Parsertabelle. Die Tabelle wird während der Animation zeilenweise, Zustand für Zustand aufgebaut. Schon abgearbeitete Zustände und Übergänge werden in der deterministischen LR(k)-Maschine grau gefärbt.

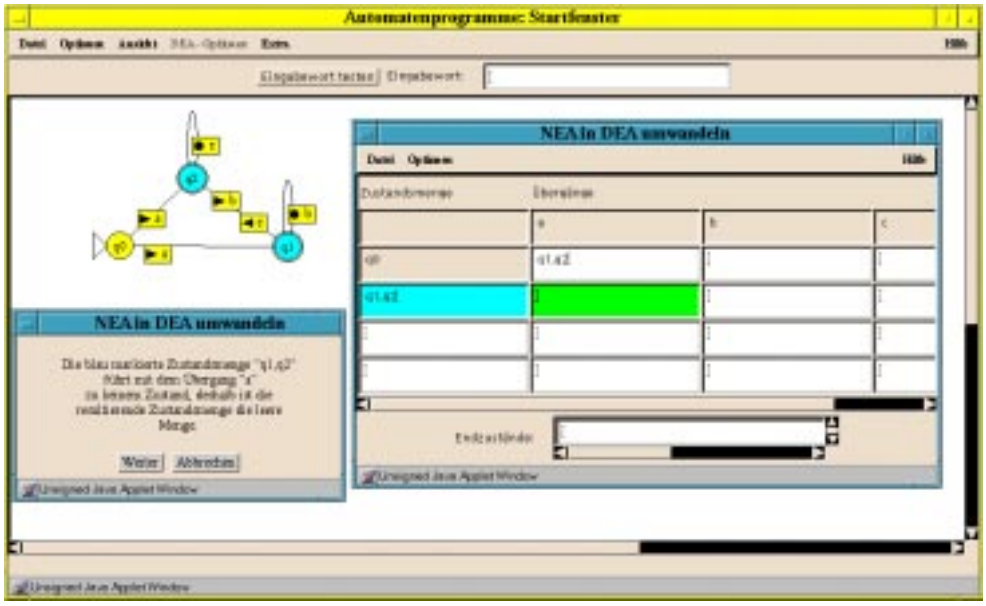


Abbildung 2.15: Interaktives Lernprogramm zum Bereich endliche Automaten.

Bei der Konstruktion der LR-Parser-Tabelle wird sowohl der aktuelle Zustand der deterministischen LR(k)-Maschine als auch der korrespondierende Zustand (Reihe) in der Tabelle mit gelber Farbe hervorgehoben (hellgrau in den Abbildungen). Bei einem shift- oder goto-Eintrag sind Übergang und Symbol der Transition und das zugehörige korrespondierende Symbol blau oder cyan markiert, siehe Abbildung 2.16.

Bei einem reduce-Eintrag wird die korrespondierende Produktion der Grammatik und das im Zustand befindliche LR(k)-Item analog hervorgehoben. Konflikte werden mit einem kurzen roten Text beschrieben und durch rote Einträge in Tabelle, Produktion und LR(k)-Item hervorgehoben.

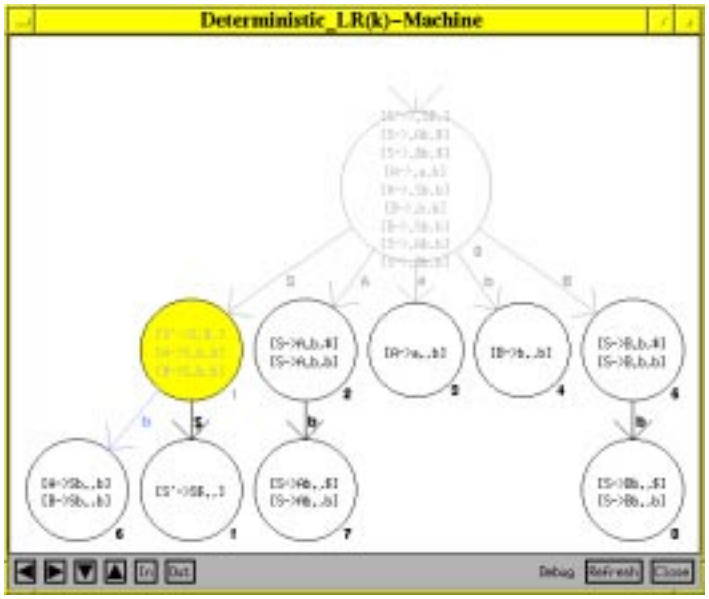
Abbildung 2.17 zeigt zwei LR(k)-Items $[A \to Sb\cdot, b]$ und $[B \to Sb\cdot b]$ im Zustand mit Nummer 6, die einen reduce-reduce-Konflikt verursachen: Die beiden zugehörigen reduce-Anweisungen sind in die Tabelle eingetragen und rot markiert; ein Text weist zusätzlich auf die Art des Konflikts hin (reduce-reduce oder shift-reduce Konflikt). Bei mehr als zwei Einträgen werden die weiteren Anweisungen nicht mehr eingetragen, sondern lediglich durch Punkte markiert.

Die Farben sind so gewählt, daß sich zusammengehörige Teile visuell schnell identifizieren lassen: Der aktuelle Zustand (Zeile) in der Tabelle und in der deterministischen LR-Kellermaschine ist jeweils gelb gefärbt. Das Eingabezeichen (Spalte) in der Tabelle und am Zustandsübergang im Automaten bei shift-Einträgen bzw. als Vorausschauzeichen im LR(k)-Item bei reduce-Einträgen ist jeweils blau oder cyan gefärbt. Mehr noch als bei der Animation des CKY-Algorithmus, erlaubt die Animation den Studierenden durch Angabe eigener Grammatiken bestimmte Aspekte oder Spezialfälle des Konstruktionsverfahrens (Auswirkung der Wahl von k, Konflikte) indivi-

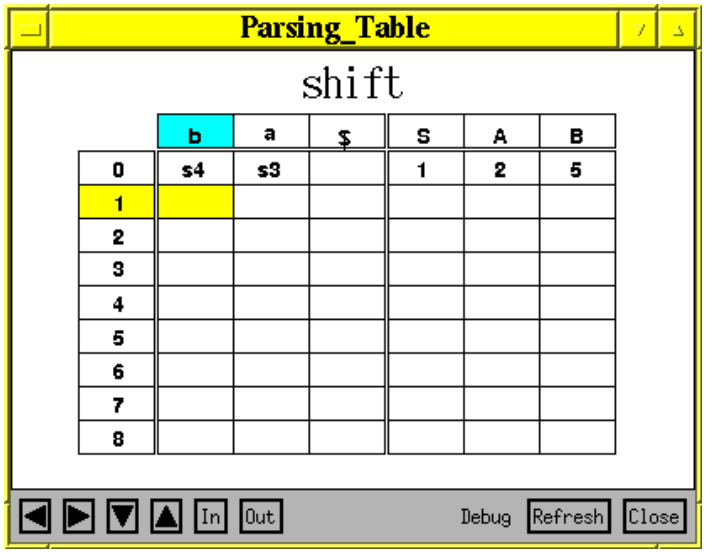
duell zu untersuchen. In einem Lehrbuch oder einer Vorlesung kann aus Platz- oder Zeitmangel nur unzureichend auf die besonderen Eigenheiten des Verfahrens eingegangen werden. Die Animation läßt sich auch benutzen, um eigene Lösungen zu entsprechenden Übungsaufgaben zu überprüfen, bevor ein studentischer Mitarbeiter die Lösung korrigiert.

2.4 Zusammenfassung

Wir haben in diesem Abschnitt – anhand zweier ausführlicher Beispiele und einem ausgewählten Animationssystem – den derzeitigen Stand der Technik demonstriert, Studierenden der Informatik Algorithmen mit Hilfe von Algorithmenanimationen anschaulich zu vermitteln. Im Gegensatz zu den Teilen der Grundausbildung Informatik, die sich ausgiebig mit dem Entwurf von Algorithmen beschäftigen, spielen in der Theoretischen Informatik mathematische Methoden und Konzepte, insbesondere Beweise, eine große Rolle bei der Vermittlung der dort behandelten Themen. Während die Animation von Algorithmen ein seit Beginn der 80er Jahre stark erforschtes Gebiet darstellt und eine Vielzahl verschiedener Systeme und Konzepte hervorgebracht hat, ist die Visualisierung mathematischer Konzepte speziell für den Ausbildungbereich ein nur wenig berücksichtigter Bereich. In den folgenden Kapiteln stellen wir unsere Konzepte vor, wie sich Beweise mit Hilfe des Rechner anschaulich und interaktiv Studierenden vermittelt lassen. Wir beginnen mit einer in der Theoretischen Informatik stark verbreiteten Beweistechnik: Reduktionsbeweise.

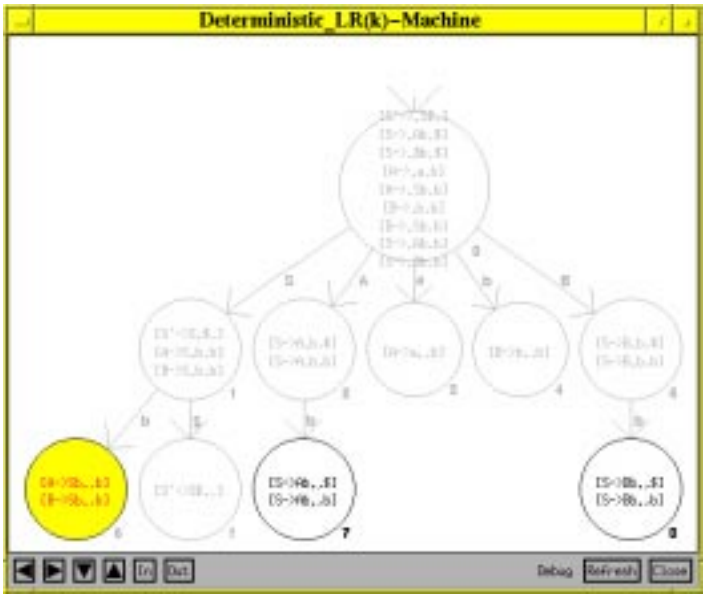


(a) Deterministische LR(1)-Maschine

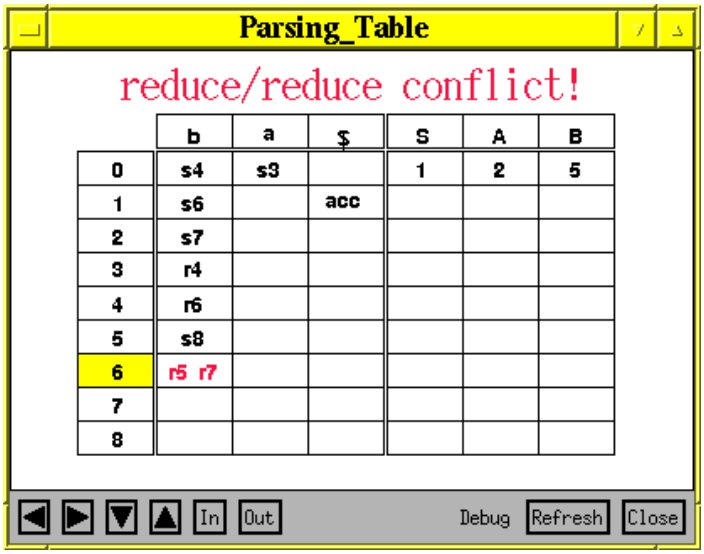


(b) LR(1)-Parsing-Tabelle

Abbildung 2.16: Gelb (helles Grau) markierter Zustand im Automaten mit blauen (dunkles Grau) Zustandsübergang (a) und analog hervorgehobenes Kästchen der Parsertabelle, in welcher der zugehörige shift-Eintrag s5 eingetragen wird (b).



(a) Deterministische LR(1)-Maschine



	b	a	$	S	A	B
0	s4	s3		1	2	5
1	s6		acc			
2	s7					
3	r4					
4	r6					
5	s8					
6	r5 r7					
7						
8						

(b) LR(1)-Parsing-Tabelle

Abbildung 2.17: Reduce-reduce Konflikt

Kapitel 3

Interaktive und visuelle Präsentation von Reduktionen

In diesem Kapitel wenden wir uns einer wichtigen Beweismethode aus der Theoretischen Informatik zu – der Reduktionstechnik: Viele Eigenschaften von (Entscheidungs)-Problemen lassen sich durch eine Abbildung eines bekannten Problems auf ein unbekanntes (oder umgekehrt) genauer untersuchen. Dabei sind sowohl die Eigenschaften der beteiligten Probleme als auch die der Abbildung relevant. Mit Hilfe dieser Technik können zum Beispiel untere Zeit- oder Speicherschranken von Problemen, deren Komplexität oder die Nichtberechenbarkeit bestimmter Funktionen bewiesen werden.

Die mathematische Behandlung dieser Themen erschwert vielen Studierenden den Zugang zu dieser wichtigen Technik. Wir stellen hier ein neues Konzept vor, mit dem sich zu einem Reduktionsbeweis gezielt ein interaktives Lernprogramm entwickeln läßt, mit der Studierende die für einen Beweis notwendigen Eigenschaften der behandelten Probleme und der Reduktionsabbildung visuell und interaktiv untersuchen können. Wir veranschaulichen unsere Methode anhand von vier von uns implementierten Beispielen. Während der Vorlesung „Informatik III“ im Rahmen der Übungen haben wir zwei dieser Programme eingesetzt. Sie konnten von den Studierenden zur Bearbeitung zweier Übungsaufgaben benutzt werden. Am Ende dieses Kapitels berichten wir über eine Evaluation dieses Einsatzes.

Unser Ansatz zur visuellen und interaktiven Präsentation von Reduktionen kann als eine Erweiterung der Technik der Algorithmenanimation interpretiert werden und läßt sich in Kürze wie folgt beschreiben:

1. Eine einzelne Instanz eines Problems kann meist auf eine natürliche Art und Weise visualisiert werden.

2. Die Reduktionsabbildung ist berechenbar und kann zu einer Algorithmenanimation ausgebaut werden, welche die Transformation der Instanzen eines Problems in Instanzen eines anderen Problems visualisiert.

3. Die Instanzen eines Problems lassen sich mit einem Algorithmus lösen. Dieser kann implementiert und zu einer visuellen Darstellung des Lösungsprozesses erweitert werden. Nichtdeterminismen oder zeitaufwendige Teile des Algorithmus lassen sich dabei durch Benutzerinteraktion realisieren.

4. Mit einer Koppelung der interaktiven Algorithmenanimationen aus 2 und 3 läßt sich ein simultaner, visueller Zusammenhang zwischen den Problemen und deren Eigenschaften herstellen.

Wir beginnen mit einer Darstellung der Reduktionstechnik, die sich an deren Anwendung in der NP-Vollständigkeitstheorie orientiert. Anschließend erläutern wir, wie Reduktionsbeweise in Lehrbüchern oder in einer Vorlesung traditionell präsentiert werden, um dann unseren Ansatz zur interaktiven Visualisierung von Reduktionen zu beschreiben.

3.1 Reduktionen

Die Grundstruktur einer Reduktion ist immer gleich: Ein Problem wird auf ein weiteres abgebildet, so daß bestimmte Eigenschaften erhalten bleiben, zum Beispiel, die Lösbarkeit eines Entscheidungsproblems, die Semantik eines übersetzten Programms oder, daß ein Folge von Zahlen mit einem bestimmten Zeitaufwand sortiert werden kann. Da wir normalerweise nur an Lösungen interessiert sind, die sich mit Hilfe eines Computers berechnen lassen, ist eine wesentliche Eigenschaft der Reduktionsabbildung ihre Berechenbarkeit.

Die zu transformierenden Objekte hängen vom Anwendungsbereich ab. Es kann sich dabei um Entscheidungsprobleme aber auch um Programme einer Programmiersprache handeln, die in die Maschinensprache eines Mikroprozessors übersetzt werden sollen. Wir können von den speziellen Objekten abstrahieren, indem wir sie auf geeignete Weise als Wörter über einem endlichen Alphabet kodieren, zum Beispiel als Sequenz von Nullen und Einsen, wie das bei jeder internen Repräsentation in einem Rechner der Fall ist. Deswegen wird üblicherweise eine formale Sprache zur abstrakten Beschreibung von Problemen verwendet.

Definition 2 (Reduktion)
Σ, Σ' seien endliche Alphabete und $L_1 \subseteq \Sigma^*, L_2 \subseteq \Sigma'^*$ zwei Sprachen über diesen Alphabeten. Eine *Reduktion* ist eine berechenbare Funktion $\tau : \Sigma^* \to \Sigma'^*$ mit

$$w \in L_1 \text{ genau dann, wenn } \tau(w) \in L_2 \text{ für alle } w \in \Sigma^* \quad .$$

In Definition 2 ist L_2 eine Menge von Problemen, für die wir eine Lösung kennen bzw. für die wir annehmen eine Lösung zu kennen. L_1 ist eine Menge von Problemen, für die wir bisher keine Lösung kennen. Wir betrachten Entscheidungsprobleme, daß

heißt, die Lösbarkeit eines Elements w von Σ besteht entweder in einer positiven ($w \in L_1$) oder einer negativen ($w \notin L_1$) Antwort. In der Situation von Def. 2 sagt man auch, daß L_1 *auf* L_2 *reduziert* wird.

Betrachten wir das Problem, C-Programme auf einem Rechner auszuführen. Dazu wird das C-Programm mit Hilfe eines Übersetzers in die Maschinensprache des Rechners transformiert und dort direkt ausgeführt. Dies ist eine Reduktion von C-Programmen auf Maschinenprogrammen. Der Übersetzer ist die Reduktionsabbildung. Die wesentliche Eigenschaft der Abbildung ist deren die Korrektheit. Dieser Sachverhalt läßt sich wie folgt mit einer Formulierung als Entscheidungsproblem beschreiben: Jedes Wort w aus Σ^* beschreibe ein C-Programm inklusive einer Ein- und Ausgabe und jedes Wort aus Σ'^* beschreibe ein Maschinenprogramm inklusive einer Ein- und Ausgabe. $L_1 \subseteq \Sigma^*$ sei die Menge aller C-Programme, die auf der gegebenen Eingabe die gegebene Ausgabe produzieren, analog sei $L_2 \subseteq \Sigma'^*$ die Menge aller Maschinenprogramme, die auf der gegebenen Eingabe die gegebene Ausgabe produzieren. Ein Beweis der Korrektheit des Übersetzers stellt dann in unserem Sinn einen Reduktionsbeweis dar. Einen derartigen Korrektheitsbeweis (für Turingmaschinen) wird uns zum Schluß dieser Arbeit wiederbegegnen. Der Beweis ist direkt: Eine Lösung des Entscheidungsproblem Σ'^* liegt in der Ausführung des Maschinenprogramms mit der Eingabe und Nachprüfung, ob die produzierte Ausgabe mit der gegebenen übereinstimmt. Das Entscheidungsproblem Σ^* kann dann durch die Reduktion ebenfalls gelöst werden.

Die Reduktionstechnik kann aber auch für indirekte Beweise benutzt werden, etwa um die Unentscheidbarkeit eines Problems zu beweisen: Wenn die Unentscheidbarkeit von Σ^* bekannt ist, kann durch Reduktion auf Problem Σ'^* die Unentscheidbarkeit von Σ'^* bewiesen werden.

Abbildung 3.1 veranschaulicht die Grundstruktur einer Reduktion.

3.1.1 Traditionelle Präsentation von Reduktionsbeweisen

Das Grundschema von Beweisen, die auf der Reduktionsmethode beruhen, sei es, um den Nachweis einer unteren Schranke eines Problems zu erbringen oder dessen Komplexität zu beweisen, ist immer sehr ähnlich und beruht auf den Nachweis bestimmter Eigenschaften der Reduktionsabbildung. Bevor ein Reduktionsbeweis durch den Lehrer präsentiert oder von den Lernenden ausgearbeitet werden soll, müssen zuerst die beiden Probleme selbst und dann die Reduktionsabbildung verstanden werden. Erst danach kann mit dem eigentlichen Beweis begonnen werden. Eine Präsentation eines Reduktionsbeweises besteht also aus:

1. der Definition und Erläuterung der beiden Entscheidungsprobleme bzw. Sprachen $L_1 \subseteq \Sigma^*$ und $L_2 \subseteq \Sigma'^*$,

2. der Definition und Erläuterung der Reduktionsabbildung $\tau : \Sigma^* \to \Sigma'^*$ und

3. dem Beweis von Eigenschaften von τ.

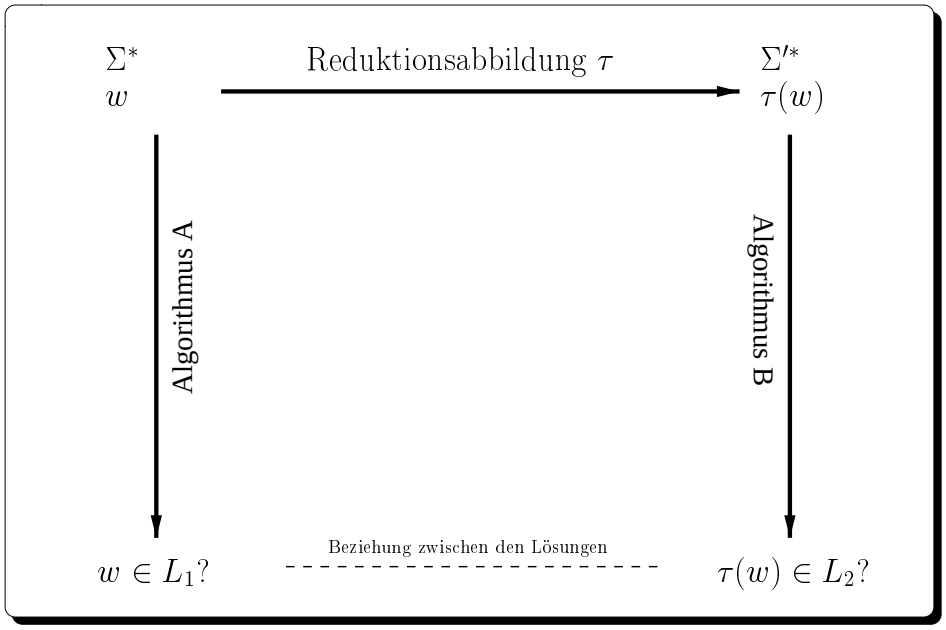


Abbildung 3.1: Schematisch dargestellte Beziehungen zwischen zwei Entscheidungsprobleme Σ^* und Σ'^*, Instanzen w und $\tau(w)$ und deren Lösungen in einem Reduktionsbeweis.

In Vorlesungen oder Lehrbüchern werden die beiden Entscheidungsprobleme neben einer rein textuellen Beschreibung oftmals durch die visuelle Darstellungen einer oder zweier Instanzen und deren Lösungen veranschaulicht. Formal besteht die Lösung zwar lediglich in einer Ja/Nein Antwort, aber in fast allen Fällen muß dazu eine Lösung konstruiert werden, die sich in der Regel visualisieren läßt. Bei dem Problem zu entscheiden, ob zum Beispiel ein gerichteter Graph einen Hamiltonschen Kreis besitzt oder nicht, läßt sich das Problem durch Zeichnen des Graphen und Markieren eines Hamiltonschen Kreises im Graphen veranschaulichen. In der Berechenbarkeitstheorie wird unter anderem mit der Reduktionstechnik die Äquivalenz zweier formaler Berechenbarkeitsmodelle gezeigt, etwa von Turingmaschinen und μ-rekursiven Funktionen. Das Problem, hier eine Menge von Turingmaschinen oder Funktionen, wird oft durch Angabe einzelner Berechnungsschritte veranschaulicht.

Die Abbildung τ wird normalerweise teils algorithmisch teils deklarativ angegeben und oftmals noch durch visuelle Darstellungen veranschaulicht, zum Beispiel, durch Angabe einer zu transformierenden Instanz und deren Bild. Damit sich der Lernende intensiver mit der Reduktionsabbildung auseinandersetzt, fordern viele Übungsaufgaben dazu auf, die Abbildung einer bestimmte Instanz selbst durchzuführen und das Ergebnis anzugeben. In vielen Fällen ist dies aufgrund der Größe der resultierenden Instanz nur für simple Beispiele sinnvoll.

Die Eigenschaften selbst werden bewiesen, aber für gewöhnlich nicht durch Bilder veranschaulicht. In einigen Fällen reicht den Beweisautoren oder Dozenten die Ver-

mittlung der Reduktionsabbildung und dessen Eigenschaften sogar schon als Beweis aus.

3.2 Interaktive Visualisierung von Reduktionen

Die bei einer Reduktion auftretenden Abbildungen können als Algorithmen betrachtet werden. Diese lassen sich mit Hilfe von Algorithmenanimationen anschaulicher vermitteln, als es traditionell in einer Vorlesung oder einem Buch möglich ist. Wie Abbildung 3.1 zeigt, kommen bei einer Reduktion drei Abbildungen vor: die berechenbare Reduktionsabbildung $\tau : \Sigma^* \to \Sigma'^*$ und zwei Lösungsalgorithmen $A : \Sigma^* \to \{\text{ja,nein}\}$ und $B : \Sigma'^* \to \{\text{ja,nein}\}$. A und B können deterministisch sein, sie können aber auch nichtdeterministische Berechnungsschritte enthalten oder lediglich rekursiv aufzählbar sein.

Wir beschreiben im folgenden, wie die Teile einer Reduktion – (1) die beiden Probleme und zugehörigen Lösungsverfahren, (2) die Reduktionsabbildung und (3) die Eigenschaften der Abbildung – auf Basis von Algorithmenanimationen und zusätzlichen Benutzereingaben, den Studierenden visuell und interaktiv vermittelt werden können und welche zusätzlichen pädagogischen Effekte sich daraus ergeben. Ausführliche Beispiele unseres Ansatzes behandeln wir in den nächsten Abschnitten.

3.2.1 Visualisierung der Entscheidungsprobleme

Bei den Entscheidungsproblemen handelt es sich um Sprachen, für die bestimmte algorithmische Lösungsverfahren vorhanden sind oder gesucht werden. Deswegen kann eine Instanz und der Lösungsprozeß mit Hilfe einer Algorithmenanimation des Verfahrens veranschaulicht werden. Dies geht über die rein statische Visualisierung einer Instanz und einer Lösung, wie sie in Lehrbüchern zu finden ist, hinaus. Diese Vorgehensweise ist allerdings nur praktikabel, wenn das Lösungsverfahren auf einem herkömmlichen Rechner berechenbar und effizient ist, also polynomiellen Zeitaufwand besitzt und keine nichtdeterministischen Berechnungschritte enthält.

Falls das Lösungsverfahren einen schlechten, etwa exponentiellen, Zeitaufwand besitzt, nichtdeterministische Berechnungsschritte enthält oder gar nicht entscheidbar, sondern nur rekursiv aufzählbar ist, dann ist eine Implementierung als Algorithmenanimation nicht mehr sinnvoll oder gar nicht mehr möglich (etwa bei nicht entscheidbaren Problemen). In diesen Fällen, läßt sich aber eine derartige Animation zu einer *interaktiven Emulation* des Problems ausbauen, indem der Benutzer folgende Berechnungsschritte des Algorithmus kontrolliert:

- Die *Angabe einer partiellen oder vollständigen Lösung* einer Instanz bei einem Problem mit hoher Zeitkomplexität, falls eine Lösung existiert.
- Die *nichtdeterministische Auswahl* einer Anweisung, etwa bei der Konstruktion einer Lösung von NP-vollständigen Problemen.

- Die *Auswahl eines Erweiterungsschritts* einer partiellen Lösung zu einer Instanz eines rekursiv aufzählbaren Problems, etwa beim Postschen Korrespondenzproblem.

In allen drei Fällen wird jeweils der komplexe Teil des Algorithmus durch den Benutzer ausgeführt, das Programm selbst prüft im wesentlichen nur noch die Korrektheit der Auswahl und stellt sie visuell dar, siehe Abbildung 3.2. Analog kann natürlich auch Problem L_2 visuell präsentiert werden.

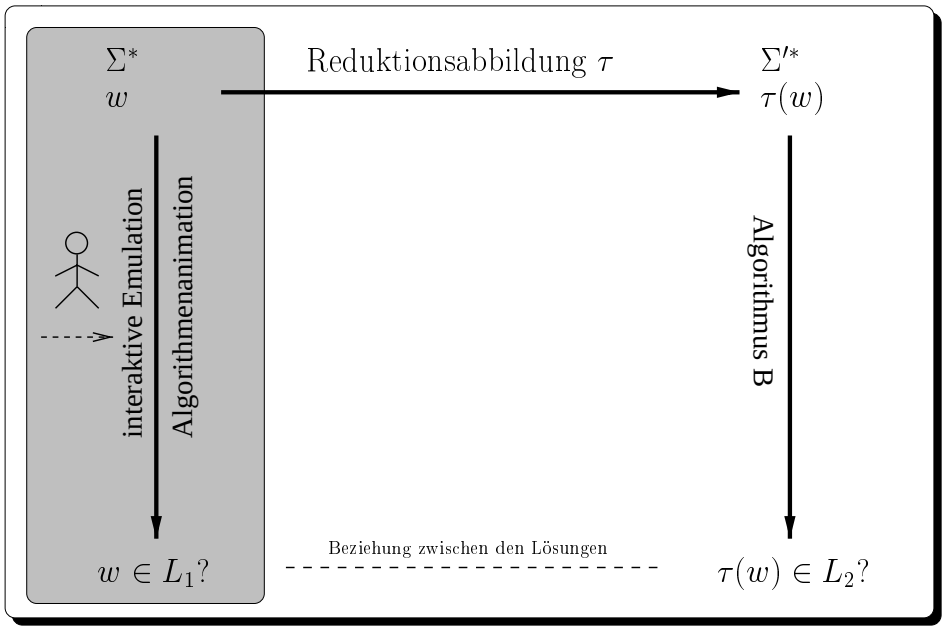


Abbildung 3.2: Interaktive Visualisierung von Probleminstanzen mit einer Algorithmenanimation und Benutzerinteraktionen (grau schattierter Teil der Reduktion).

Durch die Benutzerinteraktionen ergeben sich neben der visuellen Anschauung zusätzliche pädagogische Effekte, die sich mit einer reinen Algorithmenanimation nicht erreichen lassen:

1. Der Benutzer erhält eine sofortige Rückmeldung über den Erfolg des ausgewählten Berechnungsschritts. Wir erläutern dies exemplarisch am Beispiel eines NP-vollständigen Problems in Abschnitt 3.4.2. Es handelt sich dabei um eine interaktive Emulation von Problem L_2.

2. Durch die Schwierigkeiten, den nächsten Schritt im Lösungs- oder Aufzählungsverfahren auszuwählen, bekommt der Lernende ein Gefühl für die Komplexität des Problems, insbesondere wenn das Problem nicht entscheidbar ist. Wir erläutern dies anhand des Postschen Korrespondenzproblems in Abschnitt 3.4.1. Auch hier handelt sich um eine interaktive Emulation von Problem L_2.

3.2.2 Visualisierung der Reduktionsabbildung

Da die Reduktionsabbildung berechenbar ist, kann auch sie durch eine Algorithmenanimation präsentiert werden. Damit kann der Lernende auch Fälle oder Eigenschaften der Abbildung studieren, die bei einer traditionellen Darstellung nicht zur Geltung kommen. Insbesondere bei Reduktionsabbildungen, die schon bei einer kleinen Instanz w eine komplexe Instanz $\tau(w)$ liefern, läßt sich $\tau(w)$ mit Hilfe des Rechners auch dann noch graphisch veranschaulichen, wenn eine Darstellung durch den Autor oder durch den Studierenden zu mühsam oder zu aufwendig ist.

Wenn nicht nur die Präsentation einer bekannten Abbildung im Vordergrund steht, sondern der Lernenden zur Übung für einen eigenen Beweis eine geeignete Reduktionsabbildung finden soll, so kann bei Eingabe der Abbildung durch den Lernenden – anstatt einer Vorgabe durch den Autor – der Studierende die eingegebenen Abbildung anhand von Instanzen visuell überprüfen. Auf diese Weise können Fehler in der Reduktionsabbildung entdeckt werden und der Studierende bekommt Vertrauen in die Richtigkeit der eigenen Abbildung. In Abschnitt 3.4.2 präsentieren wir eine derartige interaktive Visualisierung eines Kachelungsproblems. Bei einer geeigneten Einschränkung der Eingabe kann in vielen Fällen sogar die Korrektheit der so eingegebenen Abbildung automatisch überprüft werden. Die Art des Korrektheitsbegriffs hängt dabei natürlich vom konkreten Anwendungsbereich ab. Die Korrektheit umfaßt aber immer die Berechenbarkeit von τ und dessen lösungserhaltende Eigenschaften. Bei einer eingegebenen Reduktionsabbildungen mit für den Beweis unerwünschten Eigenschaften lassen sich dem Studierenden sofort Hinweise auf den gemachten Fehler mitteilen. Wir stellen eine solche Implementierung in Abschnitt 3.4.3 vor. Siehe Abbildung 3.3 für eine schematische Darstellung.

3.2.3 Visualisierung der Beziehung zwischen den Lösungen

Bei einer simultanen Animation bzw. interaktiven Emulation der Entscheidungsverfahren A und B, können auch die Beziehungen zwischen der Lösung der Instanz aus L_1 zu der transformierten Instanz visuell dargestellt werden, zum Beispiel, wie eine Lösung bzw. die Konstruktion einer Lösung von w mit der Lösung von $\tau(w)$ zusammenhängt, oder umgekehrt. Dies kann den Studierenden helfen, den für alle Instanzen geführten Beweis dieser Eigenschaften anhand einiger Beispielinstanzen besser nachzuvollziehen. Insbesondere bei einer interaktiven Emulation der Probleme und einer Animation der Reduktion läßt sich so bei einer Änderung einer partiellen Lösung von w durch den Benutzer simultan der Zusammenhang mit der zugehörigen partiellen Lösung von $\tau(w)$ nachvollziehen. Siehe Abbildung 3.4 für eine schematische Darstellung.

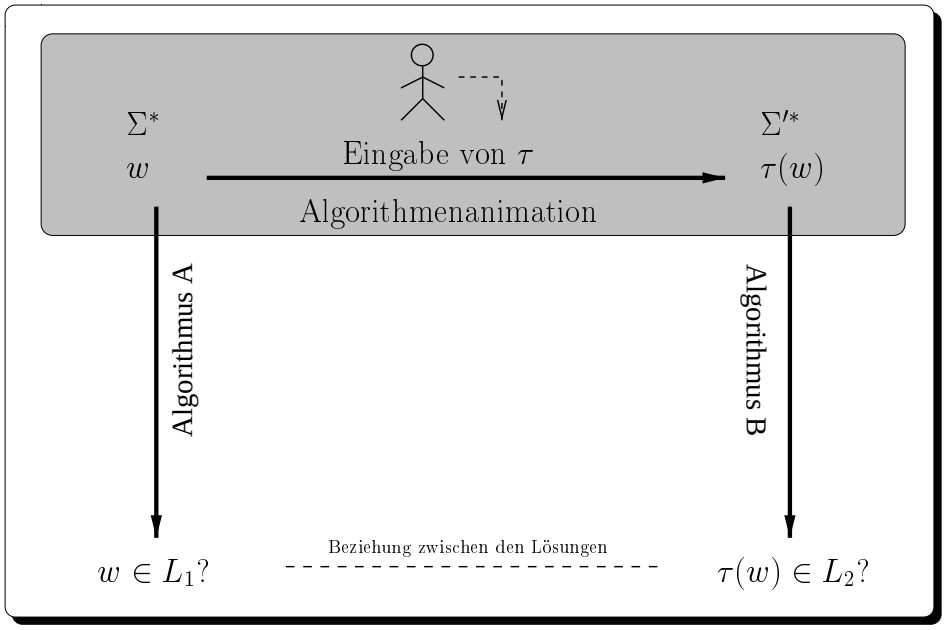


Abbildung 3.3: Algorithmenanimation der Reduktionsabbildung und deren Eingabe durch den Benutzer (grau schattierter Teil).

3.3 Auswahl der interaktiven und visuellen Teile einer Reduktion

Wie wir gesehen haben, lassen sich Reduktionen auf vielfältige Weise anschaulich und mit Benutzerinteraktion vermitteln. Dieser Vielfalt steht der Entwicklungsaufwand solcher Software entgegen. Nicht alles, was möglich ist, kann mit vertretbarem Zeitaufwand auch implementiert werden. Wir erörtern deswegen kurz, wann es sinnvoll ist, bestimmte Teile einer Reduktion interaktiv zu visualisieren und wann nicht.

Diese Entscheidung hängt im wesentlichen vom Einsatz der zu entwickelnden Lernsoftware ab. Die interaktiven Elemente sind besonders dann wichtig, wenn ein Dozent fehlt, der den Lernenden gezielt eine Reduktion vermittelt. Dies gilt ganz besonders für das Selbststudium, aber auch bei der Nachbereitung einer Vorlesung oder bei Lernsoftware, die im Rahmen einer Übungsaufgabe eingesetzt wird. Die visuelle Anschauung ist im Grunde in allen vier Bereichen sinnvoll, vor allem aber während einer Präsentation in der Vorlesung oder beim Selbststudium, wenn der Lernende die Reduktion zum ersten Male präsentiert bekommt. Bei der Nachbereitung oder im Rahmen von Übungsaufgaben kann der visuelle Anteil zugunsten der Interaktion reduziert werden. Die Eingabe und Verifikation der Reduktionsabbildung spielt in einer Vorlesung, bei deren Nachbereitung oder beim Selbststudium keine Rolle. Sie ist für eine Übungsaufgabe aber sehr wichtig, um eine Kontrolle und Rückmeldung über die eigene Lösung zu erhalten. Siehe Tabelle 3.1.

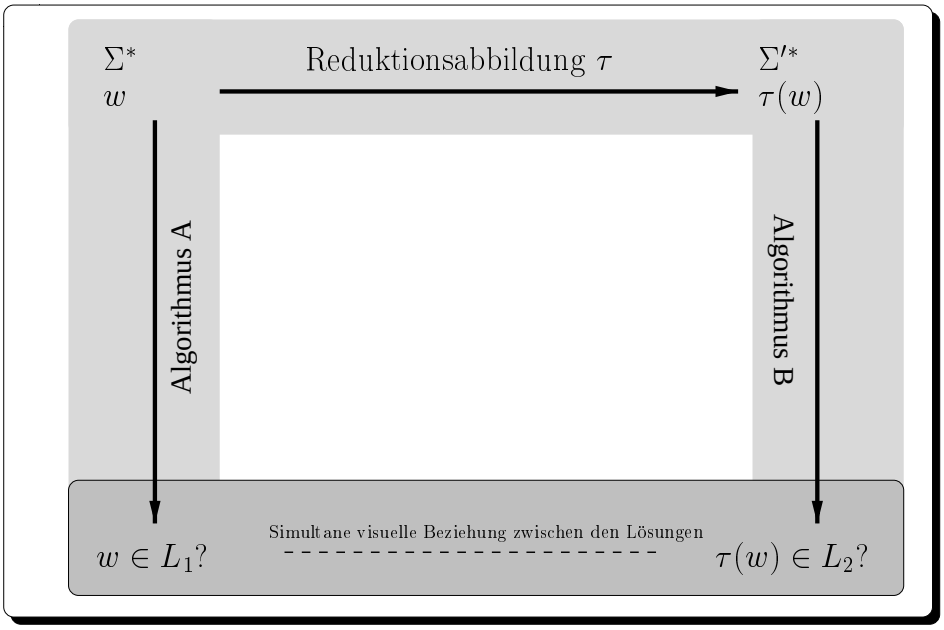


Abbildung 3.4: Visualisierung der Beziehung zwischen der Lösung einer Instanz und der Lösung der transformierten Instanz (dunkel grau schattierter Teil). Für eine simultane Visualisierung müssen auch die hellgrau schattierten Teile implementiert werden.

3.4 Beispiele

Wir fahren fort mit einer detaillierten Beschreibung von vier Reduktionen, an denen wir unser Konzept umgesetzt haben. Dabei haben wir im Einzelfall nicht alle beschriebenen Visualisierungs- und Interaktionsmöglichkeiten realisiert, sondern jeweils nur die für das betrachtete Beispiel interessanten Teile. Zwei der folgenden Lerneinheiten wurden von uns speziell für Übungsaufgaben entwickelt und konnten zur deren Bearbeitung von den Studierenden im Rahmen der Übungen zur Vorlesung „Informatik III“ benutzt werden. Wir berichten über diesen Einsatz und dessen Evaluation in Abschnitt 3.5. Unser erstes Beispiel visualisiert eine Reduktion des modifizierten Post-

	Vorlesung	Nachbereitung	Übung	Selbststudium
interaktiv	$\circ$	$+$	$++$	$++$
visuell	$++$	$+$	$+$	$++$
Korrektur	$-$	$-$	$+$	$\circ$

Tabelle 3.1: Auswahl interaktiver, visuelle Elemente oder einer automatischen Korrektur in Abhängigkeit vom Einsatz einer Reduktion: Nicht sinnvoll ($-$), bedingt sinnvoll ($\circ$), sinnvoll ($+$), sehr sinnvoll ($++$).

schen Korrespondenzproblems auf das Halteproblem. Wir beginnen mit einer kurzen Wiederholung einiger elementarer Begriffe aus der Berechenbarkeitstheorie.

3.4.1 Berechenbarkeit

In der Berechenbarkeitstheorie interessiert man sich unter anderem für die Beziehungen zwischen verschiedenen abstrakten Berechenbarkeitsmodellen: Zum Beispiel, ob zwei Berechenbarkeitsmodellen gleich mächtig sind, das heißt, ob mit ihnen jeweils die gleiche Menge von Funktionen berechnet werden kann, oder ob es Funktionen gibt, die nicht mit einem bestimmten Modell berechenbar sind. Ein wichtiges Berechenbarkeitsmodell ist das der Turingmaschine. Wir wiederholen die wesentlichen Begriffe im nächsten Abschnitt. Mit Hilfe eines solchen oder ähnlichen formalen Berechenbarkeitsbegriffs wird in einführenden Vorlesungen in die Theoretische Informatik gezeigt, daß es Funktionen (Probleme) gibt, die sich nicht berechnen (nicht lösen bzw. entscheiden) lassen. Es gehört mittlerweile zum kanonischen Ausbildungsstoff in der Theoretischen Informatik, zuerst die Unentscheidbarkeit des Halteproblems zu zeigen.

Auf Basis der Unentscheidbarkeit des Halteproblems, werden mit Hilfe der Reduktionsmethode weitere nicht berechenbare Funktionen bzw. unentscheidbare Probleme untersucht. Von größerer Bedeutung ist dabei das Postsche Korrespondenzproblem. Weiter unten zeigen wir, wie die Reduktion dieses Problems auf das Halteproblem mit einer interaktiven Emulation des Postschen Korrespondenzproblems visuell veranschaulicht werden kann. Im folgenden setzen wir die Vertrautheit mit den elementaren Begriffen einer Turingmaschine und dem Postschen Korrespondenzproblem voraus. Die vollständigen Definitionen finden sich in Anhang A.

Reduktion des modifizierten Postschen Korrespondenzproblems auf das Halteproblem

Der Beweis der Unentscheidbarkeit des modifizierten Postschen Korrespondenzproblems basiert auf einer Reduktion des modifizierten Postschen Korrespondenzproblems auf das Halteproblem. Dazu wird eine Transformation angegeben, die eine Turingmaschine mit Anfangskonfiguration so in eine Instanz des modifizierten Postschen Korrespondenzproblems abbildet, daß die Turingmaschine genau dann hält, wenn das zugehörige modifizierte Postsche Korrespondenzproblem lösbar ist.

Die grundlegende Idee bei der Reduktion ist es, mit Hilfe des modifizierte Postschen Korrespondenzproblems die Ausführungen der Turingmaschine zu simulieren. Dazu werden in geeigneter Weise die Anfangskonfiguration, die Zustandsübergänge und die Symbole der Turingmaschine in Wortpaare transformiert, so daß mit diesen Paaren Schritt für Schritt beginnend mit der Anfangskonfiguration die Konfigurationen der Turingmaschine erzeugt werden können (und nur diese). Falls ein Endzustand erreicht wird, sorgen spezielle Wortpaare dafür, daß sich die bisher erzeugte partielle Lösung zu einer Lösung des modifizierten Postschen Korrespondenzproblems vervollständigen läßt. Abbildung 3.5 veranschaulicht die Struktur dieser Reduktion und

die von uns visualisierten Teile.

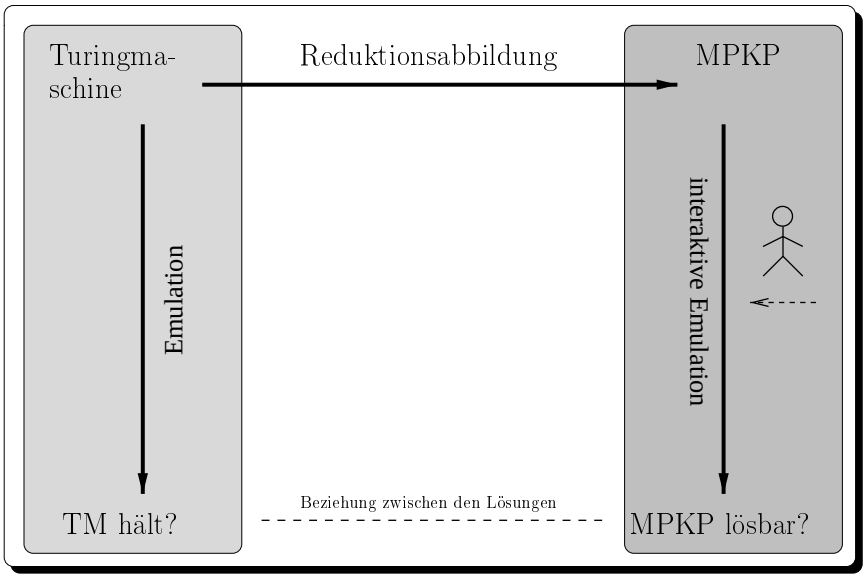


Abbildung 3.5: Beziehungen zwischen Lösungen des Halteproblems (Turingmaschine mit Eingabeband) und des modifizierten Postschen Korrespondenzproblems (MPKP)

Der hellgrau schattierte Teil repräsentiert die Darstellung von Turingmaschinen durch eine Java-Emulation, der dunkelgrau schattierte Teil repräsentiert die Darstellung des (modifizierten) Postschen Korrespondenzproblems durch eine weitere Java-Emulation. Diese beiden Darstellungen sind im Hypertext örtlich getrennt, es gibt keinen simultanen Zusammenhang zwischen den beiden Visualisierungen. Zur Visualisierung der Reduktion verwenden wir lediglich die Emulation des Postschen Korrespondenzproblems, da die Studierenden an diesem Punkt der Vorlesung schon hinreichend vertraut mit der Funktionsweise einer Turingmaschine sind. Die Reduktionsabbildung wird rein textuell erklärt.

Visualisierung einer Turingmaschine

Die Definition und Funktionsweise einer Turingmaschine wird durch eine von uns benutzte Java-Emulation visuell unterstützt. Die Studierenden können dort deterministische Turingmaschinen eingeben und schrittweise ausführen lassen. Die Zustandsänderungen und das aktuelle Band werden dabei angezeigt. Siehe Abbildung 3.6.

Diese Emulation ist insbesondere für Turingmaschinen geeignet, die viele Berechnungsschritte ausführen. Der Hypertext zur Vorlesung enthält einige Instanzen der Emulation mit „fleißigen Bibern“[1] [BRADY 1988]. Dies sind Turingmaschinen, die

[1] Engl.: *busy beaver*

Abbildung 3.6: Java-Emulation einer Turingmaschine

auf dem Bandalphabet $\{0, 1\}$ operieren und mit leerem Band (nur Nullen) starten. Fixiert man die Anzahl n der Zustände dieser Turingmaschinen und ordnet man jedem n die maximale Anzahl Einsen zu, welche das Band nach Halten dieser Turingmaschine enthält (falls sie hält), so bekommt man eine totale Funktion auf den natürlichen Zahlen. Von dieser Funktion – der „Biberfunktion“ – wird in der Vorlesung gezeigt, daß sie nicht berechenbar ist. Der Hypertext enthält einige Java-Emulationen fleißiger Biber, die den Studierenden anschaulich vermitteln, daß schon bei wenigen Zuständen die Anzahl Berechnungschritte und Einsen auf dem Band enorm ansteigt. Eine ausführliche Präsentation fleißiger Biber auf Papier würde in solch einem Fall Unmengen von Platz verbrauchen.

Visualisierung des Postschen Korrespondenzproblems

Mit unserer interaktiven Java-Emulation des Postschen Korrespondenzproblems können die Studierenden auch komplexere Instanzen ausprobieren, um so eine bessere Vorstellung von der Schwierigkeit des Postschen Korrespondenzproblems zu bekommen, siehe Abbildung 3.7.

Die Benutzungsschnittstelle der Simulation besteht aus drei Teilen: Die Menüleiste enthält alle wichtigen Kontrollfunktionen, wie Laden, Speichern und Eingabe von Instanzen sowie einer unbegrenzten „Undo“-Funktion. Unterhalb der Menüleiste werden die Wortpaare der Instanz eines Postschen Korrespondenzproblems aufgelistet, und zwar in einer für das Ausprobieren anschaulicheren Form: Das linke Wort eines Paars steht direkt über dem rechten Wort des Paars. Unterhalb dieser Wortpaare befindet sich eine partielle Lösung, wobei anstatt der Indizes der Wortpaare die Wortpaare selbst angegeben werden. Stimmen die beiden so gebildeten Wörter überein, so hat man eine Lösung gefunden. Der übereinstimmende Präfix der Wörter wird durch einen grünen Hintergrund hervorgehoben (dunkelgrau in der Abbildung). Der Studierende kann durch einfache Direktmanipulationen aus den Wortpaaren versuchen, eine Lösung zu konstruieren. Dazu muß mit der Maus ein Wortpaar ausgesucht und

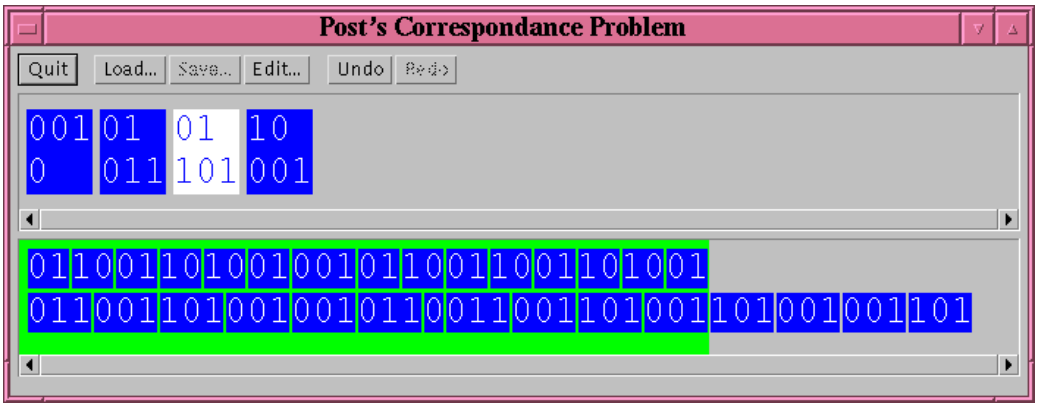


Abbildung 3.7: Emulation einer Instanz des Postschen Korrespondenzproblems

an eine Stelle in der partiellen Lösung verschoben werden, in der es dann automatisch eingefügt wird. Jeder dieser Schritte kann durch die „Undo“-Funktion wieder rückgängig gemacht werden. Mit Hilfe dieser Simulation können die Studierenden anhand von umfangreicheren Instanzen des Postschen Korrespondenzproblems ein besseres Verständnis der Komplexität des Postschen Korrespondenzproblems gewinnen.

Visualisierung der Reduktionsabbildung

Mit Hilfe der Emulation des Postschen Korrespondenzproblems und einer vorgegebenen transformierten Instanz einer Turingmaschine, können die Studierenden nachvollziehen, wie mit dem modifizierten Postschen Korrespondenzproblem, die Ausführungsschritte einer Turingmaschine simuliert werden. Abbildung 3.8 zeigt die Emulation einer Instanz eines modifizierten Postschen Korrespondenzproblems, welches zu einer Beispielinstanz einer Turingmaschine gehört. Da eine ausführliche Darstellung der Reduktionsabbildung hier zu viel Raum einnehmen würde, erläutern wir die Reduktion beispielhaft anhand dieser Emulation.

Das erste Wortpaar $\langle \#, q101\# \rangle$ zeigt die Kodierung der Anfangskonfiguration $Bq101B$ der Turingmaschine. Mit diesem Wortpaar muß begonnen werden. Das Symbol $\#$ dient zur Trennung der verschiedenen Konfigurationen. Der Zustandsübergang $\delta(q, 1) = (r, 0, R)$ wird durch das Wortpaar $\langle q1, 0r \rangle$ dargestellt. Da kein anderes Wortpaar paßt, muß die partielle Lösung damit erweitert werden. Danach werden die restlichen Bandzeichen mit Hilfe der Wortpaare $\langle 0, 0 \rangle, \langle 1, 1 \rangle, \langle B, B \rangle$ und zum Schluß mit $\langle \#, \# \rangle$ „kopiert“. Das Resultat ist eine partielle Lösung $\langle \#q101\#, \#q101\#0r01\# \rangle$, bei der die zweite Hälfte des unteren Wortes die Folgekonfiguration der Turingmaschine beschreibt. Nach einer weiteren Anwendung von $\langle 1, 1 \rangle$, einem Zustandsübergang und mehreren „Kopierregeln“ bekommt man die dritte Konfiguration, wie sie in obiger Abbildung dargestellt ist. Wird ein Finalzustand der Turingmaschine erreicht, dann sorgen entsprechende Wortpaare dafür, daß das obere Wort das untere „einholt“.

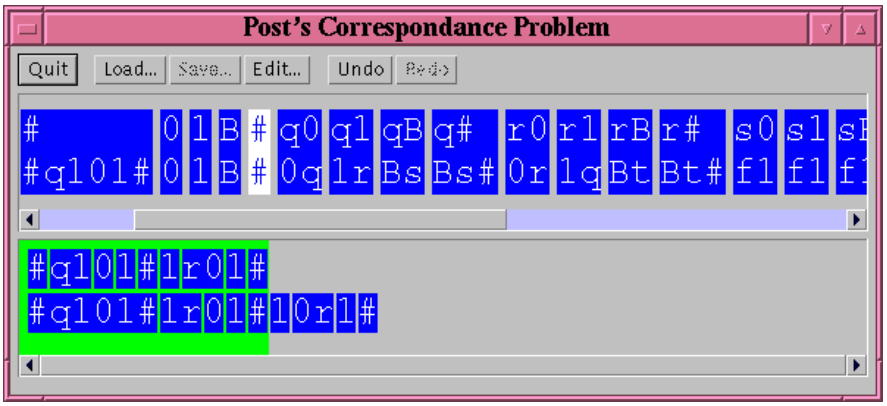


Abbildung 3.8: Simulation einer Turingmaschine mit einer Instanz des Postschen Korrespondenzproblems

Mit Hilfe dieses Beispiels, können die Studierenden die Transformation von Turingmaschinen in Instanzen des modifizierten Postschen Korrespondenzproblems, ausprobieren und dabei beispielhaft erfahren. Gebenüber einer traditionelle Darstellung des Beispiels in der Vorlesung (es nimmt unter Auslassung vieler Zwischenschritte 2 volle Folien in Anspruch), hat diese Form der Darstellung den Vorteil, die schrittweise Simulation der Turingmaschine mit dem modifizierten Postschen Korrespondenzproblem sehr viel deutlicher zu machen.

3.4.2 PUZZLE

Die folgenden drei Beispiele visualisieren Reduktionen NP-vollständiger Probleme. Doch zuvor wiederholen wir kurz die elementare Begriffe zur NP-Vollständigkeit. Ausführlicheres zur NP-Vollständigkeitstheorie findet sich in einer Vielzahl von Lehrbüchern oder in [GAREY und JOHNSON 1979]. Die in der Vorlesung benutzten Definitionen finden sich in Anhang A.

Es gibt zwei generelle Methoden, um die NP-Vollständigkeit eines Problems $L_2 \subseteq \Sigma'^*$ zu zeigen:

1. Durch Reduktion eines anderes Problem $L_1 \subseteq \Sigma^*$, dessen NP-Vollständigkeit schon bewiesen wurde, auf L_2 ($L_1 \leq_p L_2$).

2. Durch einen elementaren Beweis, das heißt, durch Angabe einer berechenbaren Funktion, die mit polynomiellen Zeitaufwand jede nichtdeterministische Turingmaschine mit einer Eingabe in eine Instanz $w_2 \in \Sigma'^*$ abbildet, so daß genau dann $w_2 \in L_2$ gilt, wenn die zugehörige nichtdeterministische Turingmaschine gestartet mit der Eingabe hält.

Wir erläutern im folgenden unser Konzept zur interaktiven Visualisierung von Reduktionen anhand drei Beispielen von Reduktionen NP-vollständiger Probleme. Die

ersten beiden wurden während der Vorlesung im Rahmen einer Übungsaufgabe behandelt.

1. Ein elementarer Beweis eines Kachelungsproblems – PUZZLE –, also eine Reduktion nichtdeterministischer Turingmaschinen auf PUZZLE. Unser Hauptaugenmerk liegt hier auf die Eingabe der Reduktionsabbildung und einer interaktiven Emulation des Kachelungsproblems.

2. Eine (einfache) Reduktion von 3SAT auf eine Variante von 3SAT – MONOTONE 3SAT. Hauptaugenmerk ist die Eingabe der Reduktionsabbildung und die automatische Verifikation einiger ihrer Eigenschaften.

3. Eine (komplexe) Reduktion von 3SAT auf das Problem Hamiltonscher Kreise – GHK. Hauptaugenmerk liegt hier auf die simultane und interaktive Visualisierung der beiden Probleme.

In der Vorlesung wird die NP-Vollständigkeit von SAT durch einen elementaren Beweis und von 3SAT durch eine Reduktion von SAT bewiesen. Im Skriptum zur Vorlesung ist zusätzlich eine Reduktion des Kachelungsproblems auf SAT angeben. Zusammengenommen bilden diese Probleme einen zusammenhängenden „Reduktionsbaum“, der den Anfang einer ganzen Hierarchie von Reduktionen darstellt, siehe Abbildung 3.9.

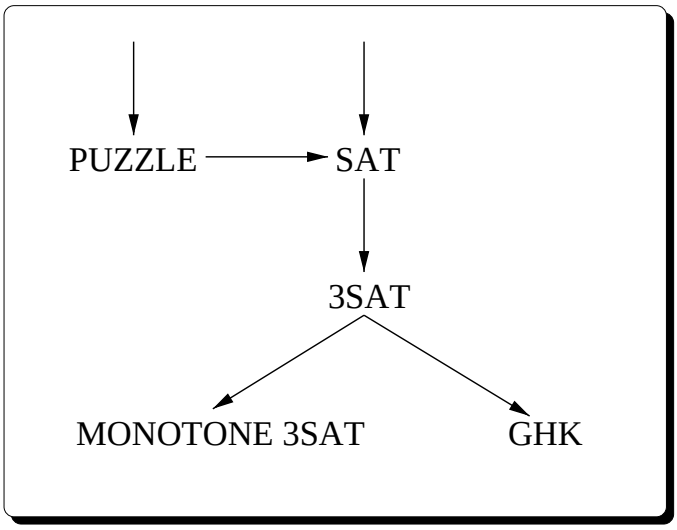


Abbildung 3.9: Hierarchie der zur Vorlesung behandelten Reduktionen

PUZZLE

Die folgende Darstellung des Kachelungsproblems orientiert sich an den Definitionen und dem Beweis in [WAGNER 1994].

Definition 3 (PUZZLE)

- INSTANZ: Σ sei ein endliches Alphabet, $T_1, \ldots, T_n \in \Sigma^4, n \geq 1$ seien Typen von „Kacheln“, wobei $\langle a, b, c, d \rangle$ den Typ einer Kachel mit oberer Seite a, rechter Seite b, unterer Seite c und linker Seite d bezeichnet, und $F \in \Sigma^{4t}, t \geq 1$ sei ein quadratischer Rahmen.
- FRAGE: Ist es möglich den Rahmen F so mit den Kacheln von Typ $T_1, \ldots, T_n$ auszufüllen, daß jede Seite einer Kachel an die Seite der benachbarten Kachel oder dem Rahmen paßt?

Dieses Entscheidungsproblem kann mit Hilfe einer visuellen Darstellung von Instanzen vermittelt werden, indem die Kacheln anschaulich als Quadrate mit Kantenmarkierungen und der Rahmen als Quadrat mit markierten Rechtecken dargestellt wird. Abbildung 3.10 zeigt eine derartige visuelle Darstellung der lösbaren Instanz $\Sigma = \{1, 2, 3, 4, 5\}, t = 3, F = \langle 1, 2, 3, 1, 1, 3, 2, 5, 2, 2, 3, 5 \rangle, T_1 = \langle 2, 3, 2, 4 \rangle, T_2 = \langle 2, 1, 2, 3 \rangle, T_3 = \langle 1, 4, 2, 2 \rangle, T_4 = \langle 4, 4, 5, 5 \rangle, T_5 = \langle 2, 5, 2, 5 \rangle$. Der quadratische Rahmen F einer Instanz ist dabei ausgehend von der linken oberen Ecke in Uhrzeigersinn visuell dargestellt.

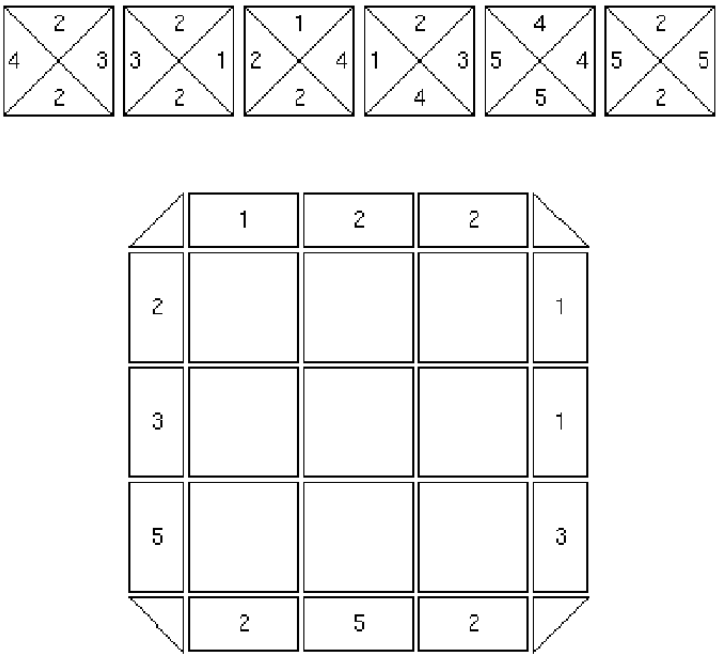


Abbildung 3.10: Visuelle Darstellung einer Beispielinstanz von PUZZLE

Für den Beweis der NP-Vollständigkeit von PUZZLE wird eine Turingmaschine mit einseitig-unendlichem Band betrachtet. Von dieser Variante von Turingmaschinen wird in der Vorlesung bewiesen, daß sie (mit lediglich polynomiellen Zusatzaufwand) die gleiche Menge von Funktionen berechnen, wie Turingmaschinen mit

zweiseitig-unendlichem Band. Aufgrund dieser Äquivalenz stellt dieser Ansatz keine Einschränkung zur Definition NP-vollständiger Probleme dar. Der Beweis der Äquivalenz wird uns noch in Kapitel 7 im Zusammenhang mit einer Beweisanimation beschäftigen.

Im zentralen Teil des Reduktionsbeweises ist zu zeigen, daß jedes Problem $L \in$ NP, $L \subseteq \Sigma^*$, polynomiell und lösungserhaltend auf PUZZLE abgebildet werden kann. Nach Definition von NP und Entscheidbarkeit (Def 18) existiert ein Polynom p und eine (einseitig-unendliche) nichtdeterministische Turingmaschine T, die genau dann nach $p(|w|)$ Schritten auf Eingabe $w \in \Sigma^*$ mit Ausgabe 1 hält, wenn $w \in L$ gilt. Für den elementaren Beweis muß also aus der Beschreibung von T und einer gegebenen Eingabe eine Instanz von PUZZLE konstruiert werden, welche genau dann lösbar ist, wenn T auf der Eingabe mit Ausgabe 1 hält.

Die Grundidee des Beweises ist es, die nichtdeterministische Turingmaschine T mit der zugehörigen Instanz von PUZZLE zu simulieren. Dazu wird eine Konfiguration von T und alle möglichen Übergänge in eine Reihe von Kacheln kodiert. Diese Kacheln müssen so gestaltet werden, daß eine Reihe r von Kacheln genau dann unter der darüberliegenden Reihe paßt, wenn r zu einer Folgekonfiguration von T korrespondiert. Die obere Seite einer Kachel enthält das Symbol der zugehörigen Zelle vom Band der Turingmaschine. Wenn der Schreib-/Lesekopf über einer Zelle steht, dann wird noch der aktuelle Zustand zur oberen Seite der zugehörigen Kachel hinzugefügt und die linke und rechte Seite der Kachel wird zur Simulation der Kopfbewegung über das Band verwendet. Die untere Seite der Kachel enthält die Information für die resultierende Konfiguration. Der Rahmen der Instanz hat die Größe $t = p(|w|)$. Er wird wie folgt erzeugt: Die obere Seite wird aus dem Zustand des initialen Bandes und dem aktuellen Zustand von T konstruiert; die untere Seite stellt das Ausgabeband mit Ausgabe 1 dar; die linke und rechte Seite des Rahmens wird mit einem speziellen Symbol (#) aufgefüllt, welches nicht in Σ vorkommen darf. Abbildung 3.11 veranschaulicht die Struktur dieser Reduktion und die von uns visualisierten Teile.

Der hellgrau schattierte Teil repräsentiert die Eingabe der Reduktionsabbildung, der dunkelgau schattierte Teil repräsentiert die interaktive Emulation von PUZZLE. Der Studierenden kann eine nichtdeterministische Turingmaschine eingeben, die von ihm genauer spezifizierte Reduktionsabbildung wird darauf angewendet und anhand der interaktiven Emulation läßt sich dann experimentell Vertrauen in die eigene Abbildung gewinnen oder der Studierende kann sich durch ein Gegenbeispiel davon überzeugen, daß er keine geeignete Abbildung eingegeben hat. Die Turingmaschine mit einseitig-unendlichem Band wird nicht visualisiert, da die Studierenden zu diesem Zeitpunkt der Vorlesung hinreichend damit vertraut sind.

Visualisierung von PUZZLE

Der Hypertext zur Vorlesung enthält eine Java-Emulation einer Instanz von PUZZLE (Abbildung 3.12). Der Lernende kann die Kacheln mit der Maus anwählen und versuchen damit den Rahmen zu füllen. Falls die benachbarten Seiten zweier Kacheln oder

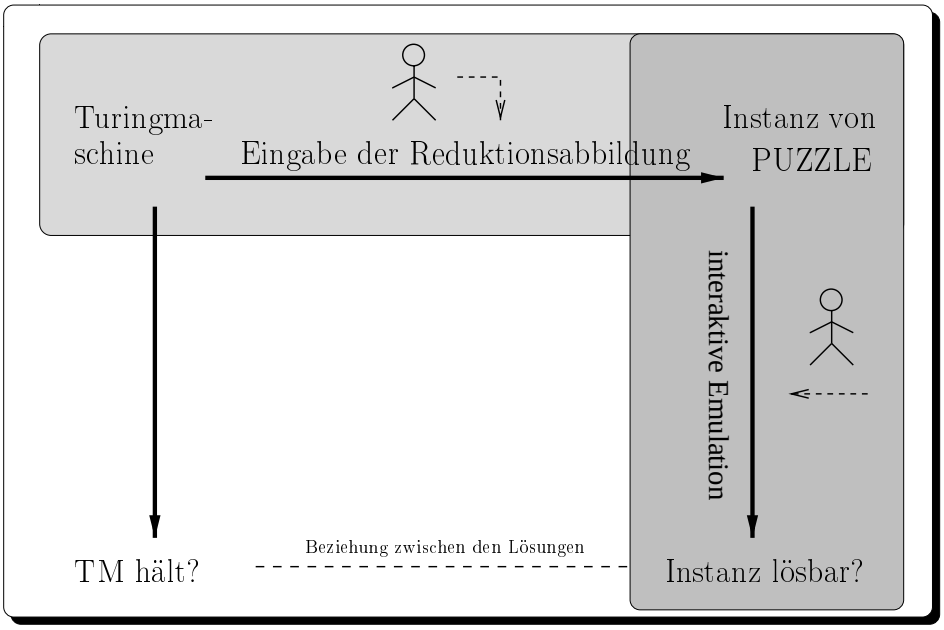


Abbildung 3.11: Visualisierung der Reduktion von PUZZLE

einer Kachel mit dem Rahmen nicht passen, dann werden diese Teile mit schwarzer Hintergrundfarbe hervorgehoben. Der Vorteil dieser Emulation gegenüber der Papierversion ist eine direkte visuelle Rückmeldung von falschen und korrekten Versuchen die Instanz zu lösen. Dies macht es einfacher für den Lernenden, das Problem im Detail zu verstehen. Nach kurzem Experimentieren sollten die meisten Benutzer mit dem Problem vertraut sein.

Visualisierung der Reduktionsabbildung

Abbildung 3.13 zeigt eine Instanz von PUZZLE die zu einer bestimmten, hier nicht näher erläuterten Turingmaschine gehört. Die obere Seite korrespondiert zur initialen Konfiguration $q_1 01B$ und die untere Seite korrespondiert zu der gewünschten Endkonfiguration $q_2 100B$. Die erste Reihe von Kacheln korrespondiert zu der initialen Konfiguration $q_1 01B$ (obere Seiten der Kacheln) und zur einer Folgekonfiguration $1q_1 1B$ (untere Seite der Kacheln); die Kachel in der oberen linken Ecke simuliert den Übergang $\langle q_1, 0\rangle \mapsto \langle q1, 1, R\rangle$.

Der schwierige Teil des Beweises besteht darin, die richtige Reduktionsabbildung zu finden. Für einen einführenden Kurs in die Theoretische Informatik ist diese Aufgabe zu komplex, um ohne Hinweise gelöst zu werden. Um die Suche nach der korrekten Transformation zu erleichtern, kann mit unserer interaktiven Visualisierung der Reduktion der Lernenden die Transformation selbst spezifizieren und eingeben. Nach der Eingabe einer Turingmaschine wird die Transformation automatisch angewendet,

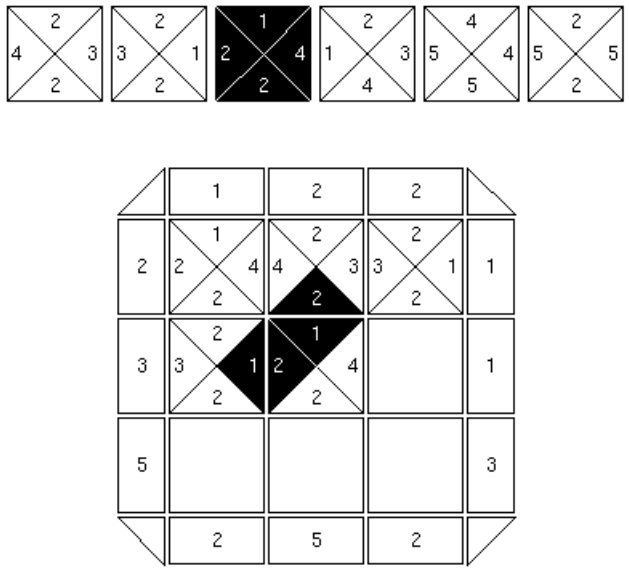


Abbildung 3.12: Bildschirmphoto der Java-basierten Emulation einer lösbaren Instanz von PUZZLE. Die Kachel in der Mitte paßt nicht. Die nicht passenden Teile der Kachel und ihrer Nachbarfelder werden automatisch schwarz hervorgehoben.

um die entsprechende Instanz zu erzeugen. Diese wird wie zuvor beschrieben simuliert. Damit kann der Lernenden anhand von Beispielinstanzen die Eigenschaften seiner Transformation validieren.

Der Rahmen wird automatisch aus dem Eingabeband und der gewünschten Ausgabe konstruiert. Es ist deswegen nicht notwendig, daß der Lernende selbst diesen Teil der Transformation spezifiziert. Desweiteren ist die graphische Benutzungsschnittstelle so gestaltet, daß völlig unsinnige Transformationen nicht eingegeben werden können.

Für die Eingabe der Transformation haben wir die Typen der Kacheln in Abhängigkeit der verschiedenen Aktionen und Zuständen der nichtdeterministische Turingmaschine aus „Sicht“ der Zellen in folgende sechs verschiedene Kategorien eingeteilt (beachte dabei, daß eine Kachel immer mit einer Zelle des Bandes korrespondiert):

1. Weder vor noch nach einem Schritt der nichtdeterministischen Turingmaschine ist der Schreib-/Lesekopf über dieser Zelle (Kachel).

2. $(p, b, R) \in \delta(q, a)$, das heißt, der Schreib-/Lesekopf ist über dieser Zelle, ein neues Symbol a wird in diese Zelle geschrieben, der Schreib-/Lesekopf bewegt sich nach rechts und der interne Zustand der nichtdeterministische Turingmaschine wechselt nach q.

3. $(p, b, L) \in \delta(q, a)$, analog zu 2.

4. $(p, b, U) \in \delta(q, a)$, analog zu 2.

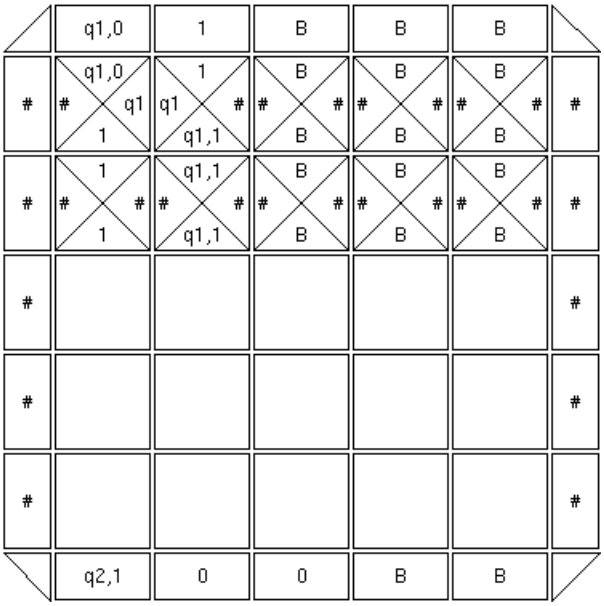


Abbildung 3.13: Eine teilweise ausgefüllte Instanz von PUZZLE, die zu einer nichtdeterministischen Turingmaschine gehört.

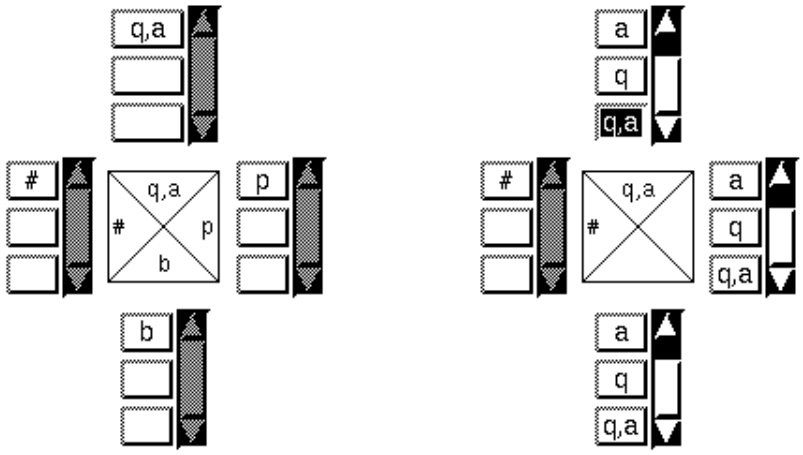
5. Der Schreib-/Lesekopf bewegt sich von links in die Zelle.

6. Der Schreib-/Lesekopf bewegt sich von rechts in die Zelle.

In diesen Kategorien sind a, b und q, p Platzhalter für die zugehörigen Symbole und Zustände der nichtdeterministischen Turingmaschine. Durch eine Mehrfachauswahl[2] der Symbole, Zustände oder Paaren von Symbolen und Zuständen kann der Benutzer die Transformation für eine Kategorie eingeben. Abbildung 3.14 zeigt die Eingabe für die Kategorie 2 und 5). Dieser Ansatz der Mehrfachauswahl erleichtert es den Lernenden, die korrekte Tranformation zu finden und verhindert unsinnige Eingaben.

Nachdem die Transformation auf diese Weise spezifiziert ist, kann der Studierende eine nichtdeterministische Turingmaschine eingeben und die automatisch erzeugte Instanz von PUZZLE eingehender untersuchen. Die Studierenden können mit Hilfe der Emulation der Instanz ihre Lösung experimentell überprüfen und Sicherheit in die Korrektheit der Transformation gewinnen oder sie durch ein Gegenbeispiel widerlegen: Zum Beispiel durch Eingabe einer Turingmaschine, die bei einer bestimmten Eingabe mit Ausgabe 1 anhält, ohne das die zugehörige Instanz von PUZZLE lösbar ist oder umgekehrt. Ist dies der Fall, dann kann der Studierende die Transformation modifizieren und dessen Korrektheit erneut durch Experimentieren validieren oder falsifizieren.

[2] Engl.: *multiple-choice*



(a) Vollständige, korrekte Eingabe zu Kategorie 2

(b) Partielle Eingabe zu Kategorie 5

Abbildung 3.14: Eingabe der Transformation: (a) zeigt die korrekte Eingabe für die Kategorie 2, (b) zeigt die teilweise eingegebene Kategorie 5.

3.4.3 MONOTONE 3SAT

Wir betrachten nun ein weiteres Entscheidungsproblem, dessen NP-Vollständigkeit nicht wie bei PUZZLE über einen generischen Beweis bewiesen wird, sondern durch eine Reduktion von 3SAT, dem Erfüllbarkeitsproblem aussagenlogischer Klauselmengen mit Klauseln, die aus höchstens drei Literalen bestehen.

Definition 4 (3SAT)

- INSTANZ: Eine Klauselmenge S, so daß jede Klausel in S aus höchstens drei Literalen besteht.
- FRAGE: Ist S erfüllbar?

Wie in Abschnitt 3.1 beschrieben, nehmen wir an, daß die Instanzen von 3SAT in einer Sprache $L \subseteq \Sigma^*$ kodiert sind. Desweiteren identifizieren wir den Namen des Problems 3SAT mit der Menge aller erfüllbaren Klauselmengen.

Um Erfahrungen mit dem Beweisen NP-vollständiger Probleme zu gewinnen, hatten die Studierenden während der Übungen zur Vorlesung „Informatik III“ die Gelegenheit, von folgender Variante von 3SAT (aus [GAREY und JOHNSON 1979]) zu zeigen, daß sie NP-vollständig ist:

Definition 5 (MONOTONE 3SAT)

- INSTANZ: eine Klauselmenge S, so daß jede Klausel in S aus höchstens drei Literalen besteht und entweder nur negative oder nur positive Literale enthält.
- FRAGE: Ist S erfüllbar?

Wir nennen Klauseln positiv (negativ), wenn sie nur aus negativen (positiven) Literalen bestehen. Der Beweis der NP-Vollständigkeit von MONOTONE 3SAT basiert auf einer polynomiellen Reduktion von 3SAT auf MONOTONE 3SAT. Klauseln einer Instanz von 3SAT, die nicht der Form von MONOTONE 3SAT genügen, werden in eine Menge von Klauseln transformiert, die nur positive und negative Klauseln enthält. Dazu wird in einer nicht negativen (nicht positiven) 3SAT-Klausel ein positives (negatives) Literal x ($\neg x$) durch ein neues negatives (positives) Literal $\neg u$ (u) in der positiven (negativen) Klausel ersetzt und eine neue Klausel $\neg u \vee \neg x$ ($u \vee x$) der Klauselmenge hinzugefügt. Klauseln einer Instanz von 3SAT, die schon die in MONOTONE 3SAT verlangte Form besitzen, werden nicht verändert. Im Beweis ist dann zu zeigen, daß eine Instanz von 3SAT genau dann erfüllbar ist, wenn die transformierte Instanz erfüllbar ist.

Die Hauptschwierigkeit der Studierenden bestand darin, eine solche geeignete Abbildung zu finden. Die Idee, ein Literal durch ein neues Literal zu ersetzen, wurde schon bei der Reduktion von SAT auf 3SAT in der Vorlesung benutzt. In Rahmen der Übungsaufgabe sollten die Studierenden diese Idee aufgreifen, um einen Reduktionsbeweis für MONOTONE 3SAT zu schreiben. Der Beweis wurde von studentischen Mitarbeitern (Tutoren) korrigiert. Wir haben eine Software entwickelt, mit dem die Studierenden ihre Abbildung eingeben und automatisch überprüfen lassen konnten. Das Programm verifiziert die Reduktionsabbildung automatisch und gibt detaillierte Hinweise, falls die Transformation fehlerhaft ist. Die Instanzen selbst werden dabei nicht visualisiert. Korrekt bedeutet in diesem Fall, daß die Transformation polynomiell berechenbar und lösungserhaltend ist. Abbildung 3.15 zeigt schematisch die Reduktion.

Visualisierung der Reduktionsabbildung

Durch eine automatische Überprüfung der Reduktionsabbildung konnten die Studierenden vor Abgabe und Korrektur ihrer Lösung erfahren, ob die ihrem Beweis zugrunde liegende Reduktionsabbildung korrekt war. Der Beweis selbst konnte natürlich noch Fehler enthalten und wurde deswegen noch traditionell vom Tutor überprüft. Abbildung 3.4.3 zeigt die Eingabemaske dieses automatischen Tutors im Hypertext zur Vorlesung.

Mit Hilfe dieser Eingabemaske kann eine Transformation τ von 3SAT auf MONOTONE 3SAT durch Eingabe der Bilder von τ auf zwölf verschiedene Typen von Klauseln definiert werden. Diese Typen charakterisieren alle möglichen Klauseln einer Instanz von MONOTONE 3SAT. Zu jeder Klausel C (erste Spalte „Urbildbereich“ in Abbildung 3.4.3) kann eine Menge von Klauseln eingegeben werden („Bild“). Klau-

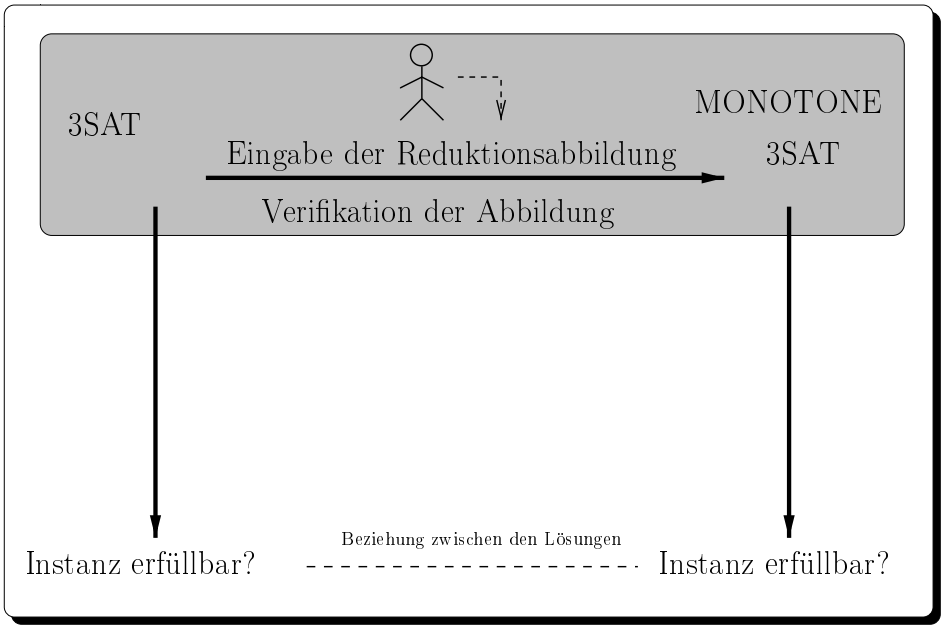


Abbildung 3.15: Eingabe und Verifikation der Reduktionsabbildung bei einer polynomiellen Reduktion von 3SAT auf MONOTONE 3SAT.

seln werden als Folgen von Literalen getrennt durch „|“ (logische Disjunktion) eingeben. Literale sind entweder aussagenlogische Variablen (gegeben durch kleine Buchstaben) oder Variablen mit voranstehender Negation „-“. Eine Menge von Klauseln ist eine durch Komma getrennte Folge von Klauseln. Die vierte Reihe in Abbildung 3.4.3 definiert zum Beispiel $\tau(x \vee \neg y)$ als die Menge $\{x \vee u, \neg u \vee \neg y\}$ von nicht tautologischen Klauseln mit einer positiven und einer negativen Klausel. u ist eine dabei neu eingeführte aussagenlogische Variable.

Nachdem ein Student die Transformation vollständig oder partiell eingegeben und „Abbildung Überprüfen“ gedrückt hat, wird automatisch überprüft, ob die Transformation eine Abbildung von 3SAT nach MONOTONE 3SAT darstellt, und ob sie erfüllbarkeitserhaltend ist oder nicht. Die polynomielle Berechenbarkeit der Transformation wird implizit durch das Eingabeformat sichergestellt: nicht polynomielle berechenbare Funktionen lassen sich nicht eingeben.

Falls, wie in unserem Beispiel, die Transformation nicht korrekt ist, dann gibt das Programm detaillierte Meldungen über die gemachten Fehler zurück (Abbildung 3.17). Zum Beispiel, wurde in der siebten Reihe für die Klausel $\neg x \vee \neg y \vee \neg z$ ($-x \mid -y \mid -z$ bei ersten Spalte „Urbildbereich“) kein Bild angegeben (zweite Spalte „Bildbereich“). Die Information über eine fehlerhafte Reduktion ist durch rote Farbe (dunkler Grauton) in der dritten Spalte „Bemerkungen“ hervorgehoben. Falls die Reduktion nicht erfüllbarkeitserhaltend ist, gibt das Programm zu jedem entsprechenden Teil einer Klausel ein Gegenbeispiel in Form einer Interpretation zurück, welche beschreibt,

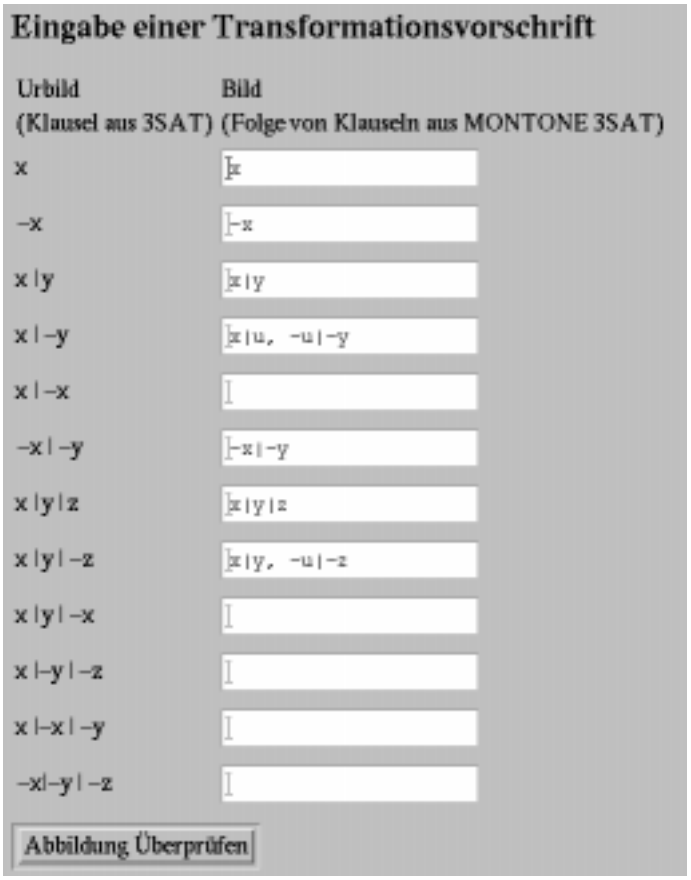


Abbildung 3.16: Eingabe einer Transformation

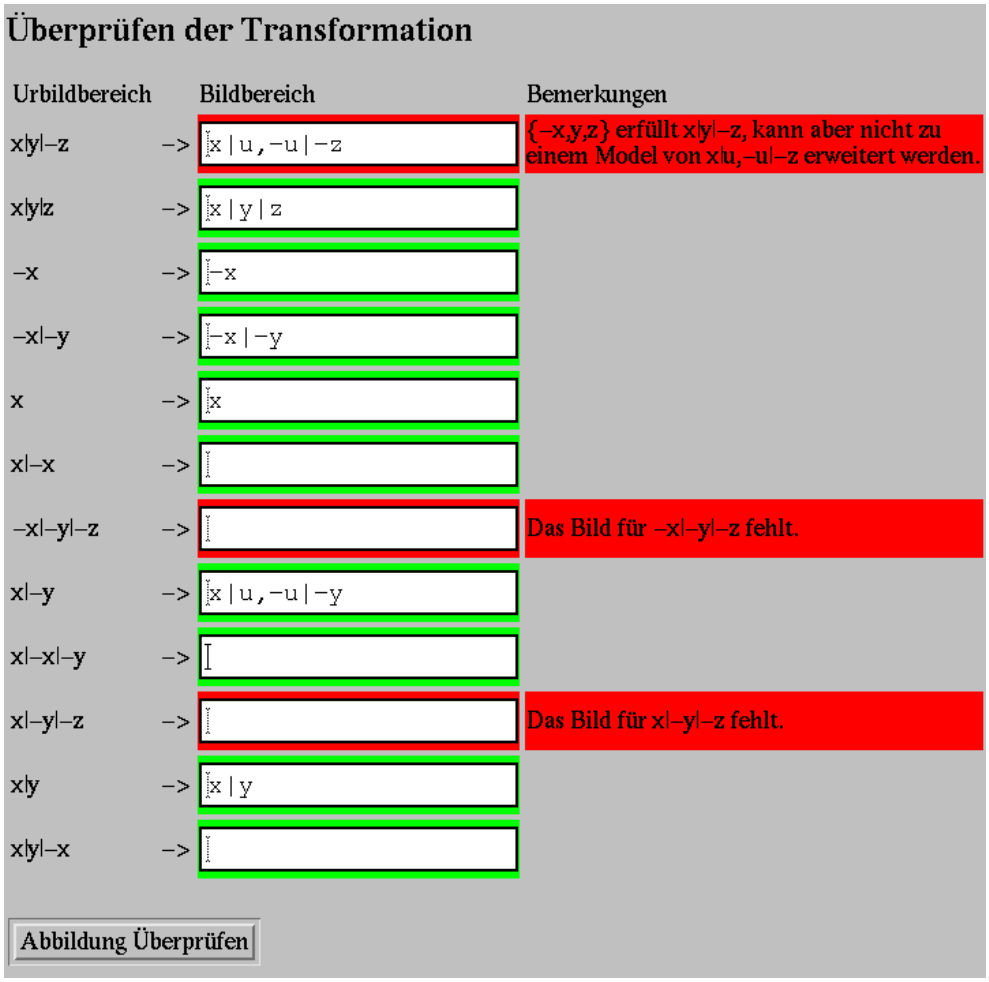


Abbildung 3.17: Ausgabe des automatischen Tutors

warum die Reduktion an dieser Stelle zu einem fehlerhaften, das heißt, nicht erfüllbarkeitserhaltendem Ergebnis führt. (Erste Reihe in Abbildung 3.17). Mit Hilfe dieser Unterstützung, kann eine korrekte Reduktion gefunden werden, bevor sie und der zugehörige Reduktionsbeweis von den Tutoren korrigiert wird.

Da es im allgemeinen unentscheidbar ist, ob eine Funktion τ erfüllbarkeitserhaltend ist oder nicht, überprüft unser Programm eine stärkere, entscheidbare Bedingung:

- Jedes Modell einer Instanz S von 3SAT kann zu einem Modell der transformierten Instanz $\tau(S)$ erweitert werden und
- Jedes Modell einer Instanz $\tau(S)$ ist ein Modell von S.

Aufgrund dieser Einschränkung können nicht alle erfüllbarkeitserhaltenden Reduktionen automatisch als korrekt identifiziert werden. Falls τ zum Beispiel eine erfüllbarkeitserhaltenden Abbildung ist, die obigen Bedingungen genügt, dann kann daraus eine weitere erfüllbarkeitserhaltenden Reduktion konstruiert werden, indem die Vorzeichen der Literale in der resultierenden Klauselmenge invertiert werden. Die so erzeugte Abbildung ist erfüllbarkeitserhaltend, genügt aber nicht mehr obiger Bedingung! Die so konstruierten Abbildungen sind allerdings etwas „unnatürlich“, insbesondere kommen Studierenden normalerweise nicht auf solche komplizierten Lösungen. Deswegen ist obige Einschränkung beim Einsatz in der Ausbildung nicht sonderlich hinderlich. Sie ermöglicht es, polynomiell berechenbare Transformation auf obige beschriebene einfache Art und Weise einzugegeben. Die einfache Eingabe entläßt den Studierenden allerdings nicht aus der Verpflichtung, im anschließenden schriftlichen Beweis zu zeigen, daß die Reduktion polynomiell berechenbar und erfüllbarkeitserhaltend ist.

Unser Programm ist in der Skriptsprache Perl implementiert und in unserem Hypertext mit Hilfe des Common Gateway Interface eingebunden. Das Programm selbst ist recht klein: es besteht aus ungefähr 500 Zeilen (ohne Kommentare). Der Hauptteil des Programms ist eine Implementierung eines Tableau-basierten automatischen Beweisers für Aussagenlogik. Er wird dazu verwendet, um sowohl obige eingeschränkte erfüllbarkeitserhaltenden Eigenschaften zu testen als auch die Gegenbeispiele zu erzeugen.

3.4.4 Gerichteter Hamiltonkreis

Das im Jahre 1859 von Sir William Hamilton (1805–1865) nach ihm benannte *Hamiltonsche Problem* gehört zum Standardrepertoire der algorithmischen Graphentheorie. Es wird im Grundstudium der Informatik im Zusammenhang mit Graphenproblemen sowie der dort behandelten Graphenalgorithmen vorgestellt und später in der Komplexitätstheorie eingehender untersucht. Wir betrachten eine Formulierung des Hamiltonschen Problems für gerichtete Graphen. Eine analoge Variante läßt sich auch für ungerichtete Graphen aufstellen.

Definition 6 (Gerichteter Hamiltonkreis (GHK))

- INSTANZ: Ein gerichteter Graph $G = (V, E)$ mit Knotenmenge V und Menge $E \subseteq V \times V$ von Kanten.
- FRAGE: Besitzt G einen Rundweg durch den Graphen, so daß jeder Knoten genau einmal besucht wird ?

GHK ist NP-vollständig. Bewiesen wird dies üblicherweise mit einer Reduktion von 3SAT. Der Beweis und die dabei verwendete Reduktionsabbildung ist im Gegensatz zur Reduktion von SAT auf 3SAT oder von 3SAT auf MONOTONE 3SAT nicht einfach. Die meisten Lehrbücher, die diese Reduktion behandeln, begnügen sich mit einer umgangssprachlichen Definition der Reduktionsabbildung und einer knappen Erläuterung, warum sie erfüllbarkeitserhaltend ist. Da die Definition der Reduktionsabbildung recht unüberschaubar ist und der durch die Reduktion entstehende Graph sehr schnell anwächst, haben viele Studierende erfahrungsgemäß große Probleme, die Reduktion zu verstehen. Ist andererseits die Reduktionsabbildung und der Zusammenhang zwischen den Lösungen der Instanzen gut verstanden worden, dann braucht der Beweis auch nicht mehr besonders ausführlich zu sein, um die NP-Vollständigkeit von GHK einzusehen.

Unsere folgende Beschreibung der Reduktionsabbildung ist weitgehend umgangssprachlich gehalten, in einer Form, so wie sie in den meisten Standardlehrbüchern vorkommt. Für weitere Informationen über den Beweis, sei hier auf ein Lehrbuch verwiesen, etwa [WEGENER 1993] oder [SPERSCHNEIDER und HAMMER 1996].

$S = \{c_1, \ldots, c_m\}$ sei eine Menge mit m aussagenlogischen Klauseln $c_i = z_{i1} \vee z_{i2} \vee z_{i3}$ mit $z_{ij} \in x_1, \neg x_1, ..., x_n, \neg x_n$. Wir beschreiben die Reduktionsabbildung τ informell. Man konstruiert aus S einen gerichteten Graphen $G = \tau(S) = (V, E)$ in mehreren Schritten. Jede Variable in S wird durch jeweils einen Knoten in $\tau(S)$ und jede Klausel in S wird durch jeweils einen Teilgraphen bestehend aus sechs Knoten in $\tau(S)$ repräsentiert. Die genaue Struktur der Teilgraphen für die Klausel $x_1 \vee \neg x_2 \vee \neg x_1$ zeigt Abb. 3.18. Es ist hilfreich, sich diese Teilgraphen zunächst als „black box“ mit drei eingehenden und drei ausgehenden Kanten vorzustellen - jeweils eine eingehende und ausgehende Kante für eines der drei Literale der Klausel. Jeder Variablenknoten besitzt zwei eingehende und zwei ausgehende Kanten. Die horizontale Eingangskante der Knoten auf der linken Seite des Teilgraphen repräsentieren eine positive Belegung der Variablen, die vertikale Eingangskante innerhalb des Teilgraphen eine negative. Analog repräsentieren die horizontale Ausgangskante der Knoten auf der rechten Seite des Teilgraphen eine positive, die vertikale Ausgangskante innerhalb des Teilgraphen eine negative Belegung der Variablen.

Die restlichen Kanten spiegeln die Struktur der Formel im Graphen wieder: Für jede Variable werden zwei Pfade eingefügt, die die positiven und negativen Vorkommen der Variablen in den einzelnen Klauseln der Reihe nach besuchen. Schließlich werden die beiden Pfade mit der nächsten Variablen verbunden bzw. der ersten, falls es keine

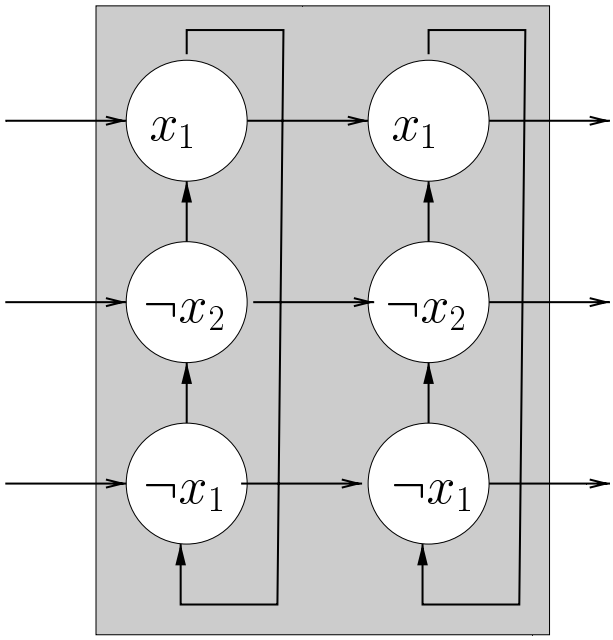


Abbildung 3.18: Reduktionsabbildung: Struktur der Teilgraphen

weitere Variable mehr gibt. Etwas detaillierter bedeutet dies: Startpunkt für den positiven Pfad zur i-ten Variablen ist der Variablenknoten i. Kommt die zugehörige Variable im l-ten Literal der k-ten Klausel positiv vor, so wird der positive Pfad um eine Kante vom bisherigen Endpunkt zum l-ten Eingangsknoten des k-ten Klauselteilgraphen erweitert. Der neue Endpunkt des positiven Pfades wird dann der l-te Ausgangsknoten des k-ten Klauselteilgraphen. Sind alle positiven Vorkommen der Variablen besucht, so wird der Endpunkt des Pfades mit dem nächsten Variablenknoten $i + 1$ verbunden. Analog verfährt man für die negativen Vorkommen der Variablen.

Abbildung 3.19 zeigt schematisch die Reduktion und unsere Visualisierung dazu. Mit dem von uns entwickelten Java-Programm können die Studierenden Instanzen von 3SAT eingeben. Die Reduktionsabbildung wird automatisch auf diese Instanzen angewendet und der zugehörige Graph wird graphisch dargestellt (oberer hellgrau schattierter Teil). Darüberhinaus kann der Studierende (partielle) Lösungen der eingegebenen Instanz oder der transformierten Instanz angeben (dunkelgrau schattierte Teile). Spezielles Augenmerk wurde dabei auf die Veranschaulichung der besonderen Eigenschaften einer Reduktion gelegt, nämlich die simultane Korrespondenz von Lösungen zu Instanzen der beiden zugehörigen, in Zusammenhang gebrachten Problemstellungen (unterer hellgrau schattierter Teil). Dadurch wird der Zusammenhang zwischen einer erfüllenden Belegung der Klauselmenge und einem Hamiltonkreis im erzeugten Graphen durch das Applet graphisch deutlich gemacht.

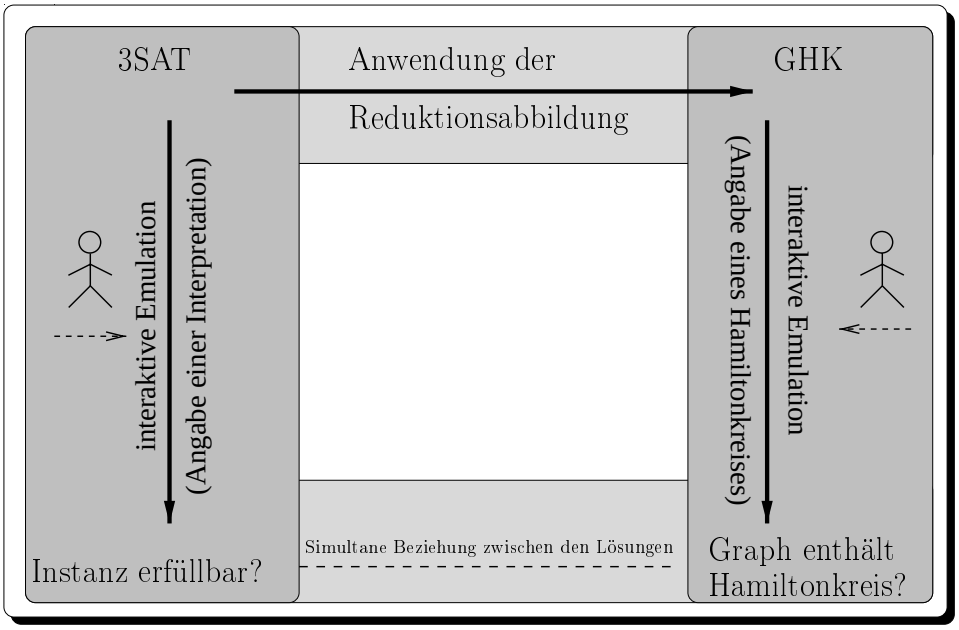


Abbildung 3.19: Visualisierung der Reduktion von 3SAT auf GHK

Visualisierung der Reduktion

Nach dem Start des Applets erscheint eine Eingabemaske, in der eine eigene Klauselmenge eingegeben oder aus einer Liste voreingestellter Klauseln ausgewählt werden kann. Hat man die Eingabe abgeschlossen, so läßt sich durch Betätigen eines Schalters der zur Klauselmenge gehörige Graph der Reduktion berechnen und anzeigen. Die Anwendung der Reduktionsabbildung selbst wird nicht visualisiert, sondern nur die Eingabeinstanz und der resultierende Graph.

Visualisierung der Probleme

3SAT Die Klauselmenge wird im wesentlichen textuell dargestellt. Abbildung 3.20 zeigt die Darstellung für die erfüllbare Menge $\{x_1 \vee \neg x_1 \vee \neg x_2, x_3 \vee x_1 \vee \neg x_2\}$.

Die in der Menge vorkommenden Variablen sind am unteren Rand der Reihe nach aufgelistet („Variablen der Formeln“). Darüber ist die Menge selbst im speziellen Eingabeformat des Applets dargestellt („angegebene Formel“): Eine Klausel besteht dabei aus höchstens drei Literalen, die durch ein Komma getrennt werden. Sie wird durch eine öffnende („{“) und eine schließende („}“) geschweifte Klammer begrenzt. Ein negatives Literal wird durch ein vor dem Bezeichner gestelltes „˜“ gekennzeichnet. Am oberen Rand sind von oben nach unten die einzelnen Klauseln der Menge aufgelistet. Links im Eingabeformat („Klauseln der Formeln“) und jeweils rechts daneben sind die drei Literale der Klausel separat aufgeführt („Literale der Klausel“). Die Farben markieren die von einer partiellen Interpretation erfüllten (grün, mittleres Grau in der

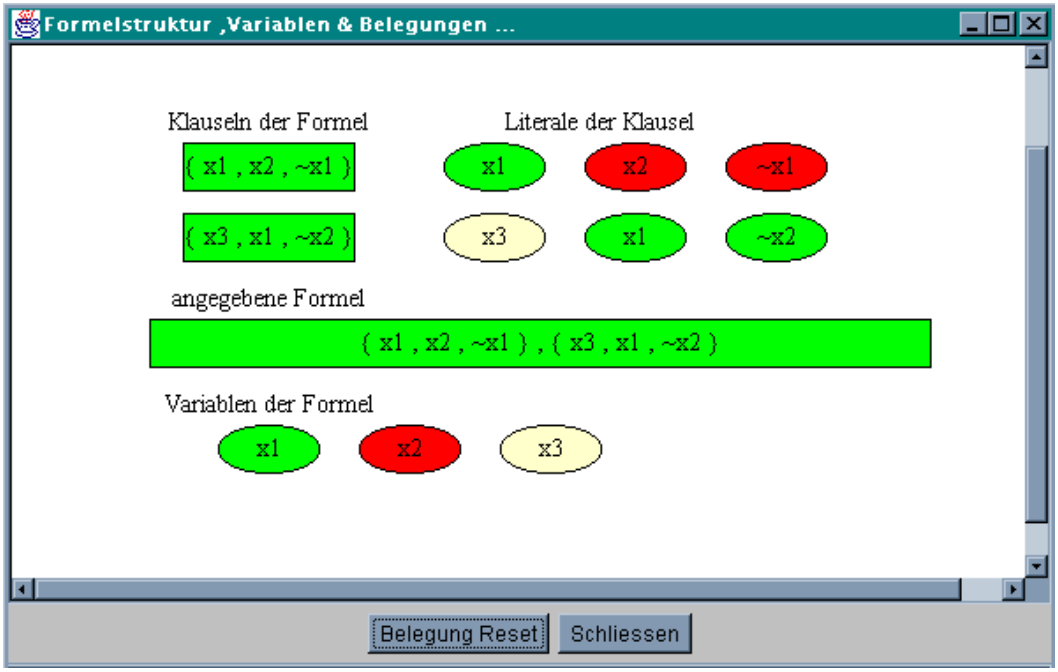


Abbildung 3.20: Visualisierung einer Instanz von 3SAT und eines partiellen Modells der Instanz

Abbildung), nicht erfüllten (rot, dunkles Grau) und noch nicht genauer spezifizierten Teile der Klauselmenge (gelb, helles Grau). Die partielle Interpretation wird interaktiv durch Anwählen der aufgelisteten Variablen und Literale mit der Maus angegeben: Durch einmaliges Anwählen einer unspezifizierten Variablen bzw. Literal (gelb) wird es mit dem Wahrheitswert WAHR belegt (grün), alle Literale, die diese Variable enthalten bzw. die Variable, die in diesem Literal enthalten ist, und alle weitere Vorkommen des Literals ändern simultan ihre Farbe. Durch weiteres Anwählen werden sie entsprechend mit dem Wahrheitswert FALSCH (rot) und dann wieder als unspezifiziert (gelb) markiert. Auf diese Weise kann der Studierende sehr leicht eine Belegung der Variablen angeben und sofort visuell überprüfen, ob die Klauselmenge oder Teile davon erfüllt sind oder nicht.

Hamiltonkreis Das Hamiltonkreisproblem wird nicht separat, sondern nur in Zusammenhang mit der Reduktion visualisiert. Dargestellt werden also nicht beliebige gerichtete Graphen, sondern lediglich der zu einer Instanz von 3SAT gehörige Graph. Abbildung 3.21 zeigt den zur Klauselmenge $\{x_1 \vee \neg x_1 \vee \neg x_2, x_3 \vee x_1 \vee \neg x_2\}$ gehörigen Graphen.

Die beiden Blocks von jeweils sechs Knoten repräsentieren die zwei Klauseln der Beispielinstanz. Sie sind gemäß der Definition der Reduktionsabbildung wie in Abbildung 3.18 skizziert mit Kanten untereinander verbunden. Die drei Knoten ober-

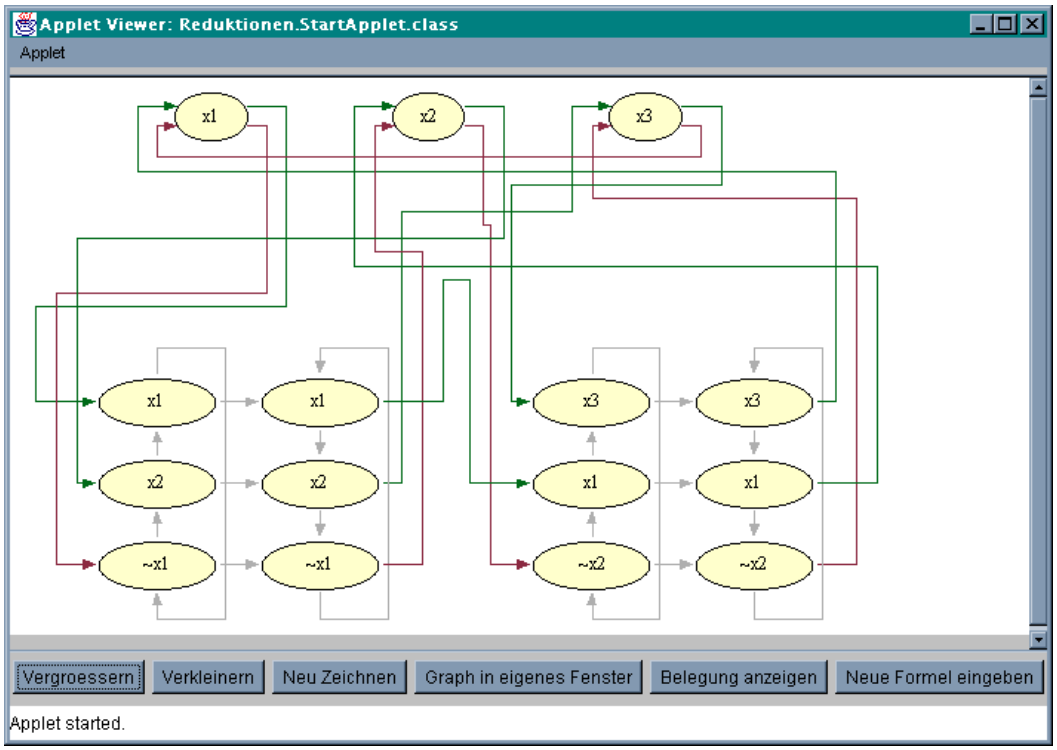


Abbildung 3.21: Visualisierung einer transformierten Instanz

halb davon entsprechen den in der Beispielinstanz vorkommenden Variablen. Sie sind ebenfalls gemäß der Definition der Reduktionsabbildung mit Kanten untereinander verknüpft. Dabei entspricht eine grüne Kante einem Pfad zu einem positiven Vorkommen und eine rote Kante einem Pfad zu einem negativen Vorkommen der Variablen in einem zu den Klauseln gehörigen Knoten (in der schwarzweiß-Abbildung leider nicht zu erkennen).

Zu jedem der einzelnen Graphelemente existieren Hilfetexte, die bei längerem Verbleiben des Mauszeigers über dem entsprechenden Element erscheinen. Wie bei der Darstellung einer Instanz von 3SAT, repräsentiert die Färbung der Graphknoten den Wahrheitswert der zugehörigen Variablen, die zum jeweilige Knoten gehört. Analog kann durch Klicken auf den Knoten dieser Wahrheitswert verändert werden.

Visualisierung der Beziehungen zwischen den Lösungen

Um die Beziehung zwischen einer eingegebenen Klauselmenge und dem zugehörigen Graphen besser verdeutlichen und experimentell erfahren zu können, besteht ein simultaner Zusammenhang zwischen einer eingegebenen partiellen Interpretation im Klauselfenster und einem partiellen Hamiltonkreis im Graphenfenster. Abbildung 3.22 zeigt die Färbung des Graphen zur in Abbildung 3.20 durch den Benutzer angegebenen

Interpretation.

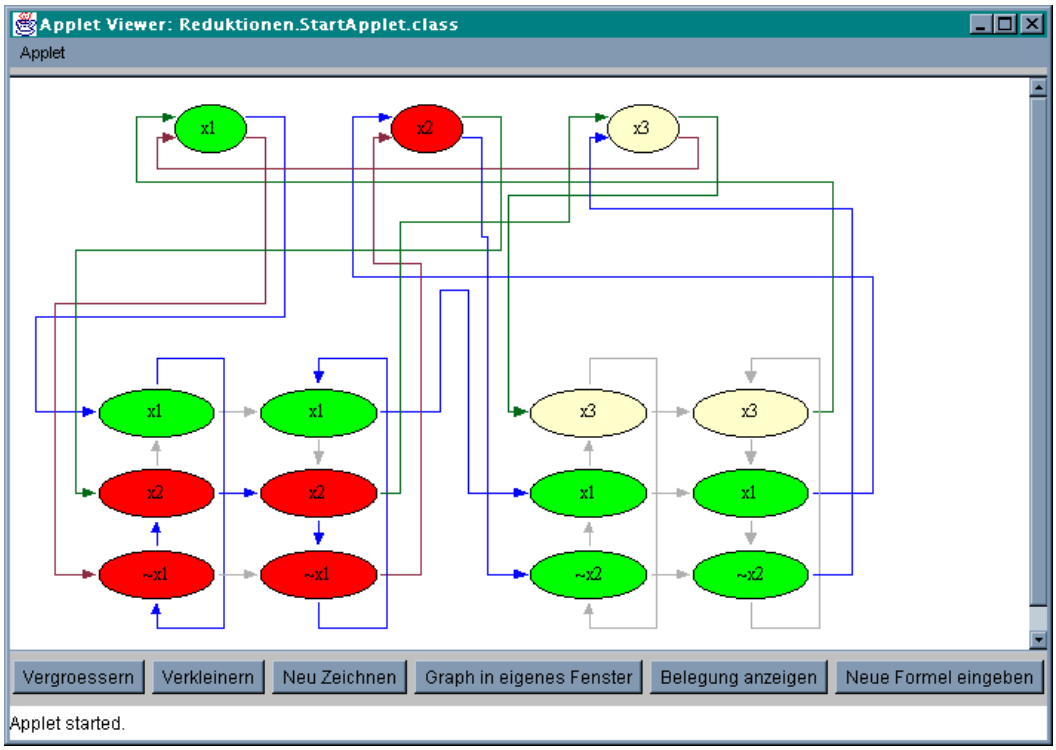


Abbildung 3.22: Beispielbelegung der Literale

Eine blaue Färbung einer Kante verdeutlicht, das diese zu dem gesuchten bzw. konstruierten Pfad dazugehört, der bei einer erfüllenden Belegung der Formel einen Kreis bildet bzw. bilden soll (in schwarzweiß-Abbildung leider nicht zu erkennen). Ein Kante kann durch Klicken zum Pfad hinzugenommen oder aus dem Pfad entfernt werden. Wird ein möglicher Teilweg des Kreises durch die Auswahl einer Kante vervollständigt, so belegt das Applet die zugehörige Variable entsprechend den Aussagen im Beweis. Dies bedeutet jedoch nicht, daß der vervollständigte Teilweg auch tatsächlich zu einem Kreis im Graphen gehört. So können durch das einfache Klicken auf die Knoten und Kanten im Graphen wahlweise beide Implikationen der Äquivalenzeigenschaft der Reduktion getrennt voneinander oder gleichzeitig geprüft werden: Man kann einen Kreis im Graphen suchen und nach dem Finden eines solchen eine erfüllende Belegung der Formel ablesen, oder man sucht eine erfüllende Belegung und sieht nach dem Bestimmen einer solchen einen Kreis im Graphen, oder aber man führt einfach beides gleichzeitig durch, um einen Kreis oder eine erfüllende Belegung zu bestimmen.

3.5 Einsatz und Evaluation

Während der Übungen zur Vorlesung „Informatik III“ im WS 97/97, haben wir die in Abschnitt 3.4.2 (PUZZLE) und Abschnitt 3.4.3 (MONOTONE 3SAT) erläuterten interaktiven Lerneinheiten eingesetzt und deren Verwendung seitens der Studierenden auf Basis eines Fragebogens ausgewertet. Im Rahmen der regulären, freiwilligen Übungen, waren zwei Reduktionsaufgaben zu bearbeiten: Ein elementarer NP-Vollständigkeitsbeweis von PUZZLE und eine Reduktion von 3SAT auf MONOTONE 3SAT. Letztere war einfach zu lösen, falls der zuvor in der Vorlesung behandelte analoge Reduktionsbeweis von SAT auf 3SAT verstanden wurde. Die Aufgabe zu PUZZLE war hingegen umfangreicher und schwieriger. Bei der Aufgabenbeschreibung und dem zugehörigen Applet wurde deswegen als Lösungshinweis ein Kacheltyp vorgegeben. Die Lösung zu den Aufgaben war schriftlich abzugeben und wurde von studentischen Mitarbeitern (Tutoren) korrigiert. Die Software konnte benutzt werden, um die eigene Lösung genauer zu untersuchen, bevor sie zu Papier gebracht wurde. Die Software war auf den für die Studierenden eingerichteten Rechnern im Rechenzentrum (AB-Pool) installiert und konnten mit einem WWW-Stöberer benutzt werden. Das PUZZLE-Applet mit HTML-Anleitung konnte als Datei vom Netz geladen und alternativ von den Studierenden auf einem heimischen Rechner benutzt werden. Da die Software zu MONOTONE 3SAT Server-basiert ist, konnten die Studierenden darauf lediglich *on-line* über das WWW zugreifen.

Einige Tage, nachdem die Studierenden ihre korrigierten Lösungen zurückerhalten hatten, wurden die Fragebögen in der Vorlesung verteilt, ausgefüllt und anschließend wieder eingesammelt. Mit der Evaluation sollte herausgefunden werden, wie viele Studierenden das zusätzliche multimediale Angebot nutzen und – basierend auf einer Selbsteinschätzung –, ob es ihnen bei der Bearbeitung der Aufgaben geholfen hat.

Weil die Antworten auf eine reine Selbsteinschätzung beruhen, sind die Ergebnisse nicht objektiv und daher mit Vorsicht zu interpretieren. Die Evaluation ist auch nicht repräsentativ, da viele Studierende nicht regelmäßig (einige leider überhaupt nicht) zur Vorlesung erscheinen. Diese Studierenden nehmen erfahrungsgemäß aber auch gar nicht an den Übungen teil, so daß dieses zusätzliche Angebot von ihnen nicht wahrgenommen wurde. Die zur Umfrage verwendeten Fragebögen haben wir in Anhang E aufgeführt.

3.5.1 Auswertung zu MONOTONE 3SAT

58 Studierenden haben den Fragebogen beantwortet. Von diesen haben 43 die Aufgabe bearbeitet. 10 davon haben die Software benutzt. Tabelle 3.2 zeigt die Auswertung zu den Fragen, welche von allen 43 Studierenden beantwortet wurden. Die erste Spalte beinhaltet alle Ergebnisse der 10 Studierenden, die zusätzlich die Software benutzt haben, die zweite Spalte beinhaltet die Ergebnisse der anderen 33 Studierenden, die dritte Spalte beinhaltet die Summe.

Die durchschnittlich aufgewendete Zeit für die Bearbeitung der Aufgabe ist bei

	Software benutzt		
	Ja	Nein	Gesamt
Anzahl	10	33	43
Zeit [h]	1.43	1.15	1.22
Schwierigkeitsgrad	3.77	3.31	3.43
korrekte Lösung	4	19	23
teilweise korrekt	6	5	11
falsche Lösung	1	8	9

Tabelle 3.2: Ergebnisse zur Evaluation von MONOTONE 3SAT. Der Schwierigkeitsgrad konnte im Bereich 1 (viel zu niedrig) bis 5 (viel zu hoch) angegeben werden. Weitere Erklärungen im Text.

denjenigen, welche die Software benutzt haben, etwas höher. Dies ist nicht sonderlich verwunderlich, da sie zusätzliche Zeit benötigten, um sich mit der Software vertraut zu machen und sie zu bedienen. Erstaunlicherweise wird der Schwierigkeitsgrad der Aufgabe von den 10 Studierenden etwas höher eingeschätzt als von diesen 33 anderen. Dies mag seine Ursache darin haben, daß viele von den 33 Studierenden die Aufgabe sofort lösen konnten und deswegen die Software nicht benutzt haben. Insbesonderen wird dies der Fall gewesen sein, wenn sie die analoge, lokale Reduktion von SAT auf 3SAT, wie sie in der Vorlesung ausführlich behandelt wurde, vorher verstanden hatten. Sie schätzen die Aufgabe deswegen vermutlich als etwas einfacher ein. Die nächsten Zeilen zeigen, daß von den 33 Studierenden, sehr viele die Aufgabe richtig lösen konnten und von den 10 Studierenden verhältnismäßig viele die Aufgabe nicht ganz korrekt gelöst haben. Dies zusammen mit der vorigen Überlegung legt nahe, daß bei den 10 Studierenden relativ mehr schwächere Studierende dabei sind als bei den 33 Studierenden. Bemerkenswert ist, daß nur einer von den 10 (10%) aber 8 von 33 (24%) eine falsche Lösung abgaben. Es scheint, daß die Software den schwächeren Studierenden geholfen hat, eine richtige Transformation zu finden, auch wenn der anschließende Beweis nicht ganz richtig ist (bei 6 Studierenden ist er lediglich teilweise korrekt).

Vorgaben in Anleitung geholfen	3.18
Fehler entdeckt	0
Fehler teilweise entdeckt	4
Anleitung gelesen	3.23

Tabelle 3.3: Ergebnisse zur Evaluation von MONOTONE 3SAT. Nur Studierende, die die Software benutzt haben, sind berücksichtigt. Bei den Vorgaben im Anleitungstext konnten Zahlen von 1 (sehr viel) bis 5 (gar nicht) und bei der Frage zur Bedienanleitung von 1 (vollständig gelesen) bis 5 (gar nicht gelesen) angegeben werden. Weitere Erklärungen im Text.

Nach der Selbsteinschätzung dieser 10 Studierenden, haben die Vorgaben (d.h. die

Beschreibung der Einschränkung bei der Transformation) nur mäßig geholfen (Tabelle 3.3). Die Vermutung, daß die Software half, die Aufgabe besser zu bearbeiten, wird durch die Selbsteinschätzung der Studierenden etwas bestätigt: vier von den 10 haben teilweise Fehler in ihrer Lösung gefunden. Auch wenn diese Ergebnisse keinen genauen Schluß zulassen, glauben wir doch, daß die Software mit ihren zusätzlichen Hilfestellungen einen Teil der (schwächeren) Studierenden geholfen hat, ihre Lösung vor deren Abgabe zu verbessern. Insbesondere zeigen die Resultate keinerlei negativen Auswirkungen bei Verwendung der Software auf die endgültige Lösung der Aufgabe.

Die Anleitung zur Software (eine Bildschirm füllende HTML-Seite) wurde lediglich zum Teil gelesen.

3.5.2 Auswertung zu PUZZLE

Auch bei dieser Aufgabe wurden 58 Fragebögen abgegeben. 29 Studierende haben die Aufgabe bearbeitet, davon 28, die auch die andere Aufgabe bearbeiteten. Das Puzzle-Applet benutzt haben 9 Personen, 8 davon haben auch die andere Aufgabe bearbeitet, 5 von diesen 8 haben dabei die Software benutzt. Tabelle 3.4 zeigt den genauen Zusammenhang zwischen der Bearbeitung von der Aufgabe zu MONOTONE 3SAT und zu PUZZLE.

MONOTONE 3SAT	PUZZLE Aufgabe bearbeitet			
bearbeitet	Nein	Ja	Ja mit Applet	Gesamt (Ja,Nein)
Nein	14	1	1	15
Ja	15	28	8	43
Ja mit Software	1	10	5	11
Gesamt (Ja,Nein)	29	29	9	58

Tabelle 3.4: Zusammenhang der Bearbeitung von PUZZLE und MONOTONE 3SAT

Tabelle 3.5 zeigt die Ergebnisse der Fragen, welche von allen 29 Studierenden beantwortet wurden. Die erste Spalte beinhaltet alle Ergebnisse der 9 Studierenden, die zusätzlich das Applet benutzt haben, die zweite Spalte beinhaltet die Ergebnisse der anderen 20 Studierenden, die dritte Spalte beinhaltet die Summe.

Wie zuvor, liegt der Zeitaufwand der 9 Studierenden, welche das Applet benutzten, über denjenigen der 20 anderen. Der Schwierigkeitsgrad wird bei beiden Gruppen als nahezu gleich eingeschätzt. Dies mag daran liegen, daß auch einige bessere Studierende, die die erste Aufgabe leichter einschätzten und ohne die zusätzliche Software gelöst haben, sich bei der schwierigeren PUZZLE Aufgabe mit Hilfe des Applets vergewissern wollten.

Es haben verhältnismäßig weniger von den 9 Studierende eine falsche Lösung abgegeben als von den 20 anderen.

	Software benutzt		
	Ja	Nein	Gesamt
Anzahl	9	20	29
Zeit [h]	2.19	1.83	1.94
Schwierigkeitsgrad	3.33	3.3	3.31
korrekte Lösung	5	8	13
teilweise korrekt	3	7	10
falsche Lösung	1	5	6
Papier oder Software	2.44	1.25	1.62

Tabelle 3.5: Ergebnisse zur Evaluation von PUZZLE. Der Schwierigkeitsgrad konnten im Bereich 1 (viel zu niedrig) bis 5 (viel zu hoch) angegeben werden. Bei der letzten Zeile konnte angegeben werden, ob bei der Aufgabe eher mit „Papier und Bleistift“ (1, sehr viel) oder dem Applet (5, nur Applet) gearbeitet wurde. Weitere Erklärungen im Text.

	Ja	Nein	Geholfen
Transformation eingegeben	5 (7)	2	2
An Beispiel getestet	5 (6)	3	1
Eigenes Beispiel	1 (3)	6	2

Tabelle 3.6: Ergebnisse zur Evaluation von PUZZLE. Nur Studierende, die das Applet benutzt haben, sind berücksichtigt. Die Zahlen in Klammern in der ersten Spalte beinhalten zusätzlich die Ergebnisse der dritten Spalte. Weitere Erklärungen im Text.

Von den 9 Studierenden, die das Applet benutzten, haben die meisten eine Transformationsvorschrift eingegeben und anhand der im Applet voreingestellten Turingmaschine getestet (siehe Tabelle 3.6).

Nur drei Studierende haben eine zusätzliche Turingmaschine eingegeben. Die meisten haben sich damit begnügt, ihre Transformation am im Applet voreingestellten Beispiel zu testen. Diese Turingmaschine sollte also mit Bedacht gewählt werden, so daß möglichst viele falsche Transformationen damit erkannt werden können. Die umfangreiche Bedienungsanleitung (mehrere Seiten) des Applets wurde noch weniger gut gelesen (Durchschnitt 2.72 bei möglichen Werten von 1 (vollständig gelesen) bis 5 (gar nicht gelesen)) als die Anleitung der anderen Software.

3.5.3 Freie Antworten

Zu beiden Programmen gab es je zwei Fragen zur deren Bedienbarkeit. Sie wurden leider nur sehr unzureichend beantwortet, so daß sich kein klares Bild ergibt. Es kann aber auch einfach sein, daß es bei der Bedienung keine besonderen Probleme auftraten. Hier wäre eine Befragung in Form eines Interviews hilfreicher gewesen.

Beim PUZZLE Applet gab es Probleme bei der Bildschirmdarstellung in verschie-

denen Stöberern (insbesonderen dem Appletviewer). Ein bekanntes Java Phänomen das unter dem geflügelten Begriff „write once, debug everywhere“ mittlerweile schon zum Allgemeingut bei der Entwicklung Java-basierter Software gehört.

3.5.4 Freie Kommentare

Neben den konkret vorgegebenen Fragen, konnten die Studierenden auf den Fragebögen auch eigene Kommentare zu den beiden Werkzeugen abgeben. Dies hilft erfahrungsgemäß, Probleme zu berücksichtigen, die bei Durchführung der Befragung nicht erkannt oder als Problem betrachtet wurden. Leider werden diese Kommentare von den Studierenden nicht besonders stark genutzt. Trotzdem ergab sich in einigen Punkten eine klare Häufung der Kommentare, die auf wesentliche Probleme beim Einsatz der Software schließen läßt.

Vier Studierende bemängelten den schlechten Zugang zur Software: „Im AB-Pool sind meistens keine Rechner verfügbar (zu voll).“. Weitere vier Studierende schrieben, insgesamt „verschlingt das Besorgen und Einarbeiten“ in die Software „zu viel Zeit“. Diese Aussage wird durch die Auswertung der Daten zur aufgewendeten Zeit untermauert. Einige empfanden die Anleitung und Aufgabenbeschreibung zu PUZZLE als zu lang. Dies wird eine wesentliche Ursache dafür gewesen sein, daß nur wenige die Bedienanleitungen vollständig gelesen haben.

Einige Bemerkungen von den Tutoren zu den abgegebenen Lösungen offenbarte eine unerwartete Kuriosität: Einige Studierenden hatten den Reduktionsbeweis zu MONOTONE 3SAT mit einer Fallunterscheidung von allen 12 Fällen geführt. Dies zeigt, daß die alternative Präsentation – in diesem Fall die redundante Eingabe der Transformation – einen nicht zu unterschätzenden Einfluß auf die Gestaltung der studentischen Lösungen hat.

3.5.5 Zusammenfassung der Evaluation

Die Auswertung zeigt, daß nur wenige Studierenden das zusätzliche Angebot angenommen haben, sie aber wahrscheinlich von dessen Nutzung profitieren konnten, während bei den anderen mehr fehlerhafte Lösungen zu finden waren. Die geringe Akzeptanz liegt im wesentlichen im zusätzlichen Zeitaufwand und den schlechten Zugriff auf die Software begründet. Letzteres kann durch eine off-line Architektur – zum Beispiel eine reine Java Implementierung mit HTML – in den Griff bekommen werden.

Da die Software zusätzlich zum gewohnten Lehrbetrieb eingesetzt wird, läßt sich der zusätzliche Zeitaufwand nicht reduzieren. Um auch dieses Hindernis zur Akzeptanz aus dem Weg zu räumen, könnten die Werkzeuge direkt im Unterricht eingesetzt werden. Diese wird zum Beispiel für einen Kurs „Formale Sprachen und endliche Automaten“ angewendet [RODGER 1995]. Allerdings ist eine solche Vorgehensweise eher bei einer instruktionalen Vermittlung von Wissen durch einen Lehrer geeignet,

als zur nachbereitenden Einübung der in der Vorlesung behandelten Themen anhand neuer, ähnlicher Probleme, so wie wir es durchgeführt haben.

Die Auswertung hat auch gezeigt, daß selbst geringfügige Dokumentationen zu den Lernprogrammen von Informatikstudenten nicht ausreichend gelesen werden. Ein Hauptaugenmerk bei der Entwicklung von Lernsoftware für diesen Anwenderkreis sollte deswegen auf eine möglichst intuitive Benutzerführung liegen. Dies ist bei der Komplexität der Software (vor allem des PUZZLE-Applets) leider nicht immer möglich. Die geringe Bereitschaft, Softwaredokumentation zu lesen, kann aber auch eine besondere Eigenart von Informatikstudierenden sein, die mit dem Umgang von Software vertraut sind und diese deswegen ohne langwieriges Lesen von Bedienanleitungen benutzen.

3.6 Zusammenfassung

Wir haben in diesem Kapitel eine Konzept vorgestellt, mit dem in der Theoretischen Informatik vorkommende Reduktionsabbildungen und (Entscheidungs-)Probleme sowie deren Lösungsverfahren visuell und interaktiv vermittelt werden können. Eine Evaluation zweier von uns exemplarisch für den Bereich NP-Vollständigkeit entwickelten Lernprogramme zeigt, daß dessen Einsatz im Rahmen der Übungen einigen Studierenden half, die für einen Reduktionsbeweis notwendige Transformation besser zu finden und damit eher einen korrekten Reduktionsbeweis auszuführen.

Die Reduktionsbeweise selbst spielten bisher eine untergeordnete Rolle. Die Software hilft nicht den Beweistext besser zu vermitteln oder ihn zu gestalten. Erfahrungsgemäß haben sehr viele Studierende Probleme, Beweise in lesbarer Form zu notieren. Siehe dazu auch Abschnitt 4.4 am Ende des folgenden Kapitels. Uns stellte sich deswegen die Frage, wie noch weitergehend visuelle Darstellungen zur Präsentation von Beweisen benutzt werden können. Wir entwickeln in den weiteren Kapiteln eine grundlegende Methodik, zu konstruktiven Beweisen systematisch eine Animation zu entwerfen, mit welcher wesentliche Teile des Beweises zusätzlich auf visueller Ebene vermittelt werden können. Unsere Vorgehensweise kann als eine Übertragung des Paradigma und der Verfahren zur Algorithmenanimation auf Beweise verstanden werden.

Bevor wir aber zum eigentlichen Kern dieser Arbeit kommen, geben wir im nächsten Kapitel einen Überblick über existierenden Ansätze und Methoden der Visualisierung von Beweisen. Unser Hauptaugenmerk gilt dabei insbesondere der Darstellung von Beweisen für Ausbildungszwecken.

Kapitel 4

Beweisvisualisierung

In diesem Kapitel geben wir einen Überblick über derzeit existierende Konzepte und Systeme, mit denen Beweise visuell dargestellt werden können. Dabei interessieren uns insbesondere Ansätze zur visuellen Vermittlung von Beweisen für Ausbildungszwecke. Im nächsten Abschnitt entwickeln wir Begriffe, mit denen sich Beweisvisualisierungen und Systeme grob klassifizieren lassen. Wir adaptieren dazu im wesentlichen die für den Bereich Softwarevisualisierung existierenden Begriffe. Diese Nähe zur Softwarevisualisierung nutzen wir aus, um eine feinere Klassifikation von Beweisvisualisierunssystemen basierend auf der Taxonomie für Softwarevisualisierung aus [PRICE et al. 1998] zu entwickeln. Da das Klassifikationeschema sehr allgemein und umfangreich ist, stellen wir es im Detail erst in Kapitel 8 vor und begnügen uns in diesem Kapitel mit einer groben Klassifizierung 4.2. Die Tragfähigkeit unseres Schemas erweist sich anschließend bei einer Klassifikation existierender Beweisvisualisierungssysteme (Abschnitt 4.3). Am Ende dieses Kapitels berichten wir über einen von uns durchgeführten Vergleich zweier Klausuraufgaben, der die praktischen Probleme der Studierenden mit dem Führen eigener Beweise aufdeckt, auf die derzeitigen Beweisvisualisierungen und Systeme nur unzureichend eingehen.

4.1 Begriffsdefinitionen

Im Bereich der Visualisierung von Beweisen gibt es im Gegensatz zur Softwarevisualisierung noch keine genaue Abgrenzung zwischen den verschiedenen existierenden Formen von Visualisierung. Um in den folgenden Abschnitten bisherige Ansätze zur Beweisvisualisierung genauer untersuchen und klassifizieren zu können, definieren und beschreiben wir einige Begriffe zur Beweisvisualisierung und ordnen existierende Begriffe in diesen Kontext ein. Siehe Abbildung 4.1 für eine schematische Darstellung der im folgenden erklärten Begriffe.

Mit *Beweisvisualisierung* bezeichnen wir den ganzen Bereich der visuellen Darstellung verschiedener Aspekte aller Arten von Beweisen mit Hilfe von Typographie, Graphik und Animationen. Wir unterscheiden zwischen Visualisierungen *nicht forma-*

ler Beweise – dies sind Beweise mit Argumentationen auf abstraktem Niveau, so wie sie in Lehrbüchern zu finden sind – und *formaler Beweise* – dies sind Beweise, die in einem formalen Kalkül ausgeführt sind und bei denen logische Aussagen Wörter einer formalen Sprache sind.

Ein nicht formaler Beweis verhält sich zu einem formalen Beweise ähnlich wie ein in Pseudokode notierter Algorithmus zu einem in einer bestimmten Programmiersprache geschriebenen Programm: Pseudokode besitzt im Gegensatz zu Programmiersprachen weder eine genaue Syntax noch eine Semantik, insbesondere kann der Algorithmus nicht auf einem Rechner ausgeführt werden. Die genau Bedeutung eines so beschriebenen Algorithmus wird dem Menschen erst durch zusätzliche umgangssprachliche Erklärungen offenbar. Analog dazu ist ein nicht formaler Beweis lediglich eine auf hohem abstrakten Niveau geführte aber wenig detaillierte umgangssprachliche Begründung für die Wahrheit einer Aussage. Einem formalen Beweis hingegen liegt ein formaler Beweiskalkül mit festgelegtem Satz von Beweisregeln zugrunde. Die zu beweisende Aussage und die in einem formalen Beweis auftretenden logischen Formeln halten sich an eine formale Syntax. Formale Beweise können automatisch vom Rechner überprüft oder mit dessen Unterstützung konstruiert werden. Wie bei einem Rechnerprogramm sind Syntax und Semantik bei einem formalen Beweise genau definiert. Diese Analogie läßt sich auch genauer als Isomorphie zwischen bestimmten konstruktiven Beweisen und typisierten Lambda-Ausdrücken beschreiben (also einer funktionalen Programmiersprache); siehe zum Beispiel [TROELSTRA 1999]. Wir begnügen uns im folgenden mit einer laxen Identifizierung von Programm und Beweis sowie Daten(strukturen) und Termen bzw. Formeln. Auf Basis dieser Analogie bilden wir im folgenden Begriffe für den Bereich Beweisvisualisierung, die eine Entsprechung im Bereich der Softwarevisualisierung haben. Siehe dazu auch die Ausführungen in Abschnitt 2.1, Seite 5.

Eine *statische Visualisierung formaler Beweise* findet sich derzeit hauptsächlich bei Systemen zur formalen Spezifikation und Verifikation von Soft- oder Hardware. [SETZER] gibt eine (nicht erschöpfende) Übersicht der gebräuchlichsten Systeme. Beweise werden hier hauptsächlich durch eine Baum- oder Graphendarstellung statisch visualisiert. In diesen meist interaktiven Systemen liegt der Schwerpunkt auf der rechnergestützten Konstruktion von noch nicht gefundenen Beweisen und weniger auf der Präsentation bereits vorliegender Beweise. Die Visualisierung hat dabei die Aufgabe, die Komplexität formaler Beweise zu reduzieren, um dem Menschen nur die gerade interessierenden Details – zum Beispiel das aktuelle Beweisziel – präsentiert werden. So ist die Termtiefe bei der Darstellung von Formeln und Termen meist beschränkt. Diese Form der visuellen Darstellung ist mit dem „pretty printing“ von Programmen vergleichbar. Eine Ausnahme von diesen rein baumartigen Beweisdarstellungen machen das ILF- [DAHN et al. 1997] und Ω-System [BENZMÜLLER et al. 1997]. Mit ihnen lassen sich formale Beweise in eine dem Menschen lesbare umgangssprachliche Darstellung bringen.

Die Spezifikation und Klassifikation der Syntax und Semantik diagrammatischer

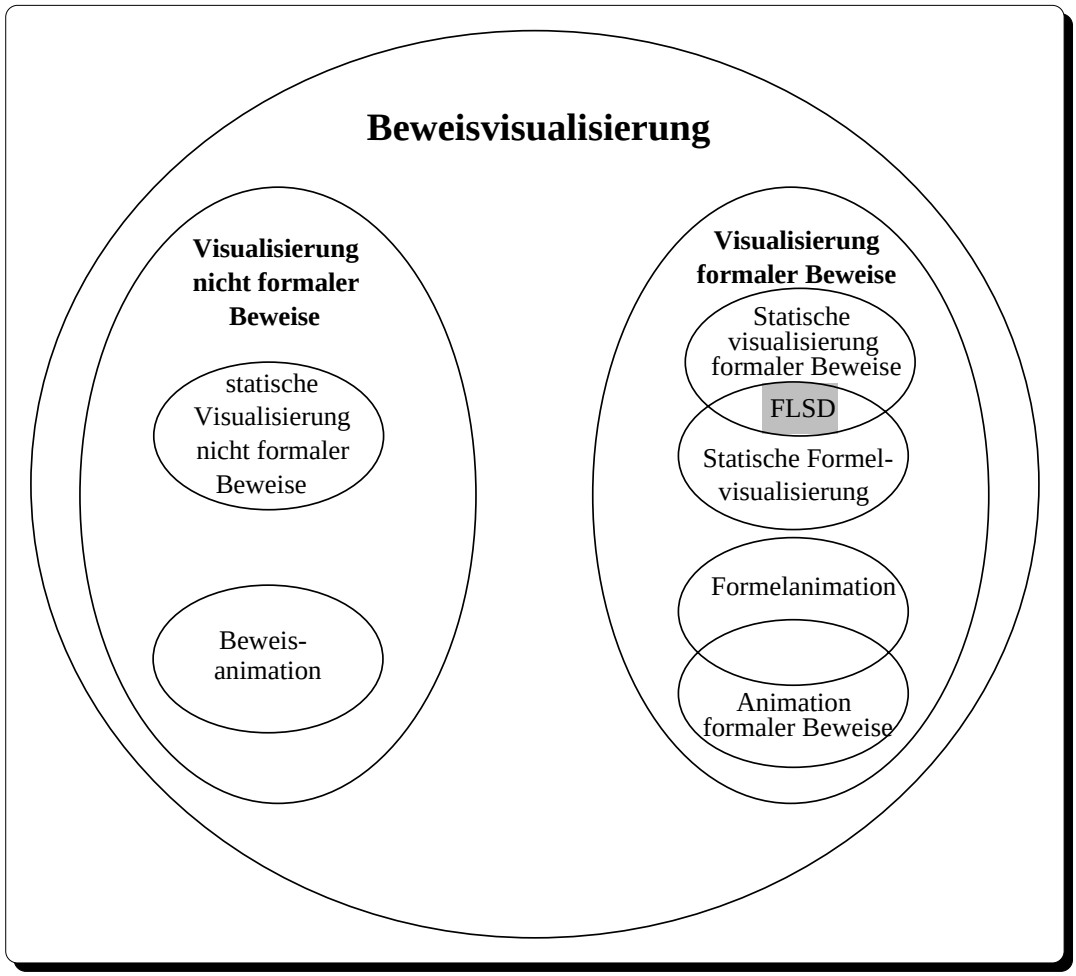


Abbildung 4.1: Zusammenhang zwischen Begriffen im Bereich der Beweisvisualisierung.

Repräsentationen ist Gegenstand der Forschung im Bereich *visueller Sprachen*[1]. Visuelle Sprachen umfassen einen sehr weiten Bereich visueller Darstellung für die Kommunikation mit und unter Menschen:

> „They [visual languages] cover the entire spectrum of human expression ranging from fine art, such as an abstract expressionist's private language, to precise technical communication using rigorously defined notation, such as musical notation, mathematical notation, or street maps." [MARRIOT und MEYER 1998]

Visuelle Sprachen umfassen im Bereich Softwarevisualisierung auch das visuelle Programmieren, enthalten in der Schnittmenge statischer Programm- und Datenvisualisierung. In der Beweisvisualisierung gibt es einen dem visuellen Programmieren entspre-

[1] Engl.: *visual languages*

chenden Forschungszweig, der als Teilbereich visueller Sprachen aufgefaßt werden kann:

> „A recent ... use of visuale languages is to formalize and support so-called diagrammatic reasoning, that is, reasoning about diagrams and reasoning by visual manipulation of diagrams.“ [MARRIOT und MEYER 1998]

Der Begriff „reasoning“ ist in diesem Zusammenhang noch sehr allgemein und wird eher im Sinne eines vernünftigen Argumentierens oder Nachdenkens gebraucht. Man kann etwa das Interpretieren diagrammatisch aufbereiteter statistischer Daten in diesem Sinne als „diagrammatic reasoning“ bezeichnen. Dies wird bei der Beweisführung mit Hilfe von Diagrammen einschränkender als *logisches Schließen mit Diagrammen*[2] [ALLWEIN und BARWISE 1996] bezeichnet und umfaßt formale und nicht formale Beweisführung mit Hilfe statischer, visueller Darstellungen, wie etwa Venn-Diagramme für das Beweisen mengentheoretischer Aussagen. Kennzeichnend für dieses Forschungsgebiet ist aber das Bestreben, formale diagrammatische Darstellungen mit genau festgelegter Syntax und Semantik zu erhalten. Wir wollen dies genauer als *formales logisches Schließen mit Diagrammen* (FLSD) bezeichnen. Dieser Bereich ist in der Schnittmenge zur Visualisierung formaler Beweise und Formeln enthalten. Insbesondere steht die Realisierung von Systemen zur visuellen Beweisführung im Vordergrund. Am weitesten fortgeschritten ist diese Technik im Bereich geometrischer Beweise: Aus Punkte und Linien bestehende Diagramme, die Beweisskizzen repräsentieren, werden formal interpretiert und haben Eingang in Systeme zur rechnergestützten Beweisführung [BARKER-PLUMMER und BAILIN 1992] und in intelligenten Tutorsystemen gefunden [ANDERSON et al. 1985, MATSUDA und OKAMOTO 1998].

Die Animation formaler Beweise und Formeln ist ein von der Forschung nur wenig berücksichtigter Bereich. Animationen könnten etwa zur dynamischen Darstellung von Formeln und insbesondere von Regel*anwendungen* benutzt werden. Bei Systemen zur Hard- und Softwareverifikation macht dies allerdings nur wenig Sinn, da die dort auftretenden Formeln und Terme nur wenig Struktur besitzen, die der menschliche Betrachter meist leicht erfaßt. Der Schwerpunkt dieser Systeme liegt in der Unterstützung des Menschen, eine bestimmte Beweisregel aus einer großen Menge potentiell anwendbarer Regeln auszuwählen. Ein animierte Darstellung einer Regelanwendung, die dem Verständnis oder der Erläuterung dieses Beweisschritts dienen soll, hilft dem Menschen wenig, einen partiellen Beweis zu einem korrekten formalen Beweis zu vervollständigen, da die weitere Beweissuche hauptsächlich vom Ergebnis der Regelanwendung aber weniger vom Verständnis der Anwendung abhängt. Das System selbst überprüft die Korrektheit des Beweises. Beim formalen logischen Schließen mit Diagrammen können animierte Darstellungen von Regelanwendungen hingegen Sinn machen. So läßt sich die Konstruktion von Venn-Diagrammen durch eine animierte Darstellung der dabei angewendeten visuellen Beweisregeln sehr viel anschaulicher darstellen, als es mit einer statischen Repräsentation möglich ist.

[2] Engl.: *logical reasoning with diagrams*

Bei nicht formalen Beweisen unterscheiden wir zwischen dem Gebrauch *statischer* und *animierter* Darstellungen. Statische visuelle Darstellungen werden im Ausbildungsbereich oft als zusätzliche Erklärungen bei nicht formalen Beweisen in Lehr- und Schulbüchern verwendet. Zur statischen Visualisierung gehört auch die strukturierte Darstellung nicht formaler Beweise, wie etwa das *Calculational proof format* [DIJKSTRA und SCHOLTEN 1990], eine strukturierte Variante [BACK et al. 1996] oder ein vergleichbarer, von Leslie Lamport entwickelter Beweisstil [LAMPORT 1995]. Auf der Ebene der Softwarevisualisierung entsprechen diese stilistischen Notationsformen der strukturierte Darstellung von Algorithmen in Pseudokode.

Unter *Beweisanimation* verstehen wir sowohl die dynamische Darstellung der *Änderungen* logischer Aussagen während eines Beweises als auch die Animation der logischen Aussagen selbst. Dieser Ansatz zur Vermittlung nicht formaler Beweise ist bisher nur bei einigen wenigen *Ad-hoc*-Implementierungen durchgeführt worden, wie etwa bei einer mit ZEUS erstellten Animation eines geometrischen Beweises des Satzes von Phytagoras [BROWN 1993, NELSON und GLASSMAN]. Eine Methodik oder ein Konzept, mit der auch andere Beweise systematisch animiert werden können, existiert nicht.

4.2 Grobes Klassifikationsschema

Unser Klassifikationschema besteht wie das Schema von Price, aus sechs obersten Kategorien, die wir im folgenden kurz erläutern. Zu Details siehe bitte Kapitel 8.

A. *Anwendungsbereich*: Der Anwendungsbereich umfaßt alle Merkmale eines Beweisvisualisierungssystems, welche für den Beweisautor wichtig sind. Dazu gehören Hardware und Betriebssystem, verwendbare Logik und Beweiskalül, Einschränkungen, Grenzen und Stärken des Systems hinsichtlichs der zu visualisierenden Beweise.

B. *Inhalt*: Hiermit wird beschrieben welche Teile (Formeln, Regeln) eines Beweises und auf welcher Abstraktionsebene (formal, nicht formal) ein Beweis visualisiert wird. Zu dieser Kategorie gehört auch die Art der Visualisierungerzeugung („live“ oder in einer Vorverarbeitung).

C. *Form*: Mit dieser Kategorie wird die visuelle Ausgabe beschrieben: welches Zielmedium (Bildschirm, Papier) für Visualisierung verwendet wird; Farben und Raumdimensionen; graphische Primitive; Animationen oder statische Bilder; Auslassung von Details und Verwenden verschiedener Sichten.

D. *Methode*: Die Methode beschreibt alle wesentlichen Merkmale eines Softwarevisualisierungssystems, die der Visualisierer zur Erzeugung der Visualisierung berücksichtigt und verwendet: Spezifikationsstil der Visualisierung; Automatisierungsgrad; Anpassbarkeit der Visualisierung durch den Benutzer; Verbindung zwischen Beweis und Visualisierung.

E. *Interaktion*: Die Interaktion des *Benutzers* mit dem Visualisierungssystem wird durch die allgemeine Merkmale einer graphischer Benutzungsschnittstellen und die spezielle Merkmale eines Animationssystems beschrieben.

F. *Wirksamkeit*: Hiermit werden Zweck, Einsatz und Nutzen des Beweisvisualisierungssystems klassifiziert.

4.3 Klassifikation von Beweisvisualisierungssystemen

Mit Hilfe unseres Klassifikationsschemas (Details siehe Kapitel 8) untersuchen wir im folgenden einige Beweisvisualisierungssysteme etwas genauer. Theorembeweiser, die zur formalen Beweisführung benutzt werden, berücksichtigen wir dabei nicht, da sie sich nicht zur Vermittlung nicht formaler Beweise eignen. Als einzige Ausnahme betrachten wir in Abschnitt 4.3.3 das Lernprogramm HYPERPROOF, da es speziell für den Ausbildungsbereich konzipiert wurde und einige auch für die Visualisierung nicht formaler Beweise interessante Merkmale aufweist. Ebenso berücksichtigen wir keine spezialisierten Systeme zur geometrischen Beweisführung, da sie weniger die anschauliche visuelle Darstellung von Beweisen sondern mehr dessen visuelle Konstruktion in den Vordergrund rücken. Als einzigen Vertreter aus diesem Bereich untersuchen wir in Abschnitt 4.3.1 lediglich eine mit dem Algorithmenanimationssystem ZEUS erstellte Animationen eines geometrischen Beweises des Satzes von Pythagoras. In Abschnitt 4.3.2 stellen wir Gloors System zur Visualisierung nicht formalen Korrektheitsbeweise von Graphenalgorithmen vor. Und als letztes System betrachten wir in Abschnitt 4.3.4 PROOFVIEWS, mit dem im *Structured calculational proof format* geschriebene Beweise in einen interaktiven Hypertext transformiert werden können.

Wir beschreiben die verschiedenen Aspekte der Systeme absatzweise in der Reihenfolge der Taxonomie (A–F), ohne jeweils dabei die einzelnen Teilkategorien zu benennen. Teile der Taxonomie, die nicht zutreffen, sind in den Ausführungen auch nicht aufgeführt.

4.3.1 Zeus

A: Anwendungsbereich Bei diesem System handelt es sich nicht um ein spezielles Beweisvisualisierungssystem, sondern um eine mit dem Algorithmenanimationssystem ZEUS [BROWN und HERSHBERGER 1992] erstellte Animation eines Beweises des Satzes von Pythagoras [NELSON und GLASSMAN]. Sie wurde im Rahmen eines Algorithmenanimations-„Festivals“ entwickelt [BROWN 1993]. ZEUS animiert Programme, die in einem Dialekt von Modula-2 geschrieben sind. Die interessanten Ereignisse und Sichten auf das Programm werden manuell mit Hilfe einer Animationsbibliothek implementiert.

Anstatt Algorithmen können mit ZEUS natürliche beliebige implementierbare Ablaufvisualisierungen realisiert werden. In diesem speziellen Beispiel wurde ein Pro-

gramm implementiert, das die konstruktiven Vorschriften eines Beweises zum Satz von Pythagoras reflektiert. Der Beweis selbst ist als eine Sicht implementiert, welche die gerade betrachteten Aussagen und Beweisregeln umgangssprachlich zeigt („Euclid's Proof“, siehe Abbildung 4.2). Es werden Teile der Aussagen (über das rechtwinklige Dreieck) graphisch in einer zweiten Sicht veranschaulicht („Euclid View“).

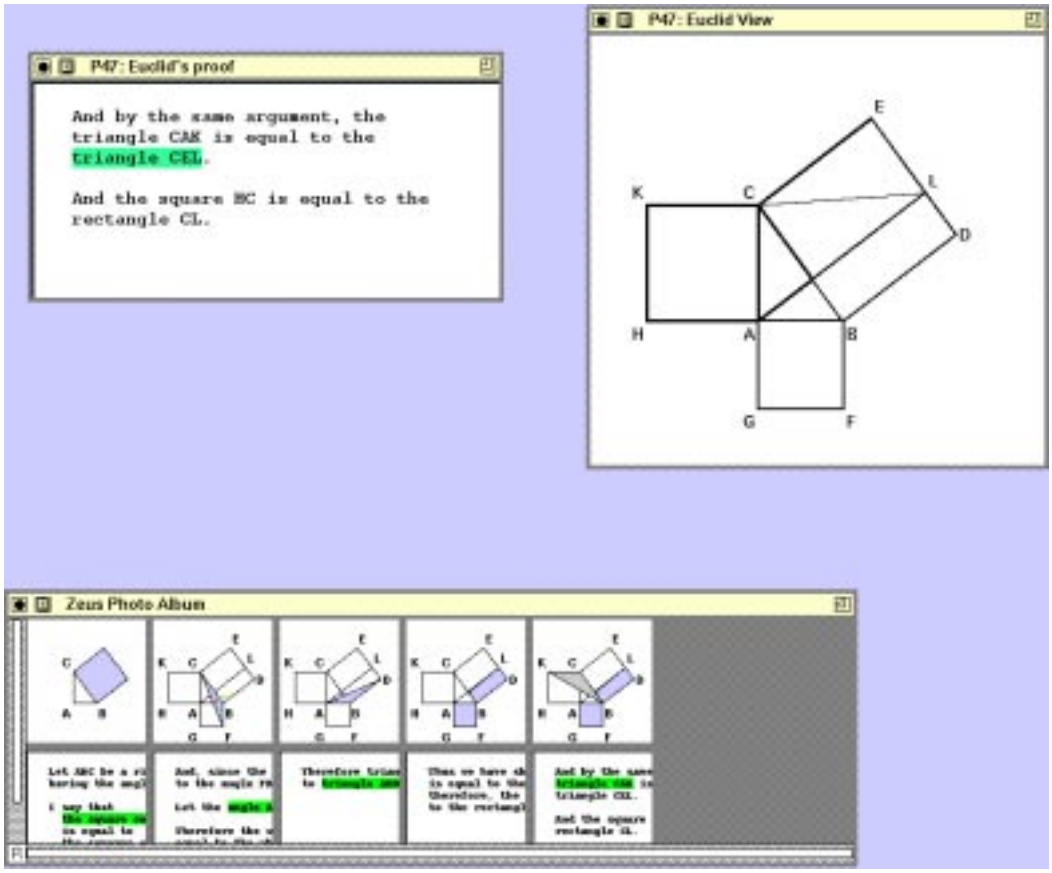


Abbildung 4.2: Bildschirmphoto einer ZEUS-Animation eines Beweises des Satzes von Pythagoras (aus [NELSON und GLASSMAN]).

B: Inhalt Beweisregeln werden nur implizit durch ihre Auswirkungen auf das dargestellte rechtwinklige Dreieck visualisiert. Die Visualisierung wird live während der Laufzeit des Programms erzeugt.

C: Form Als Darstellungsmedium dient der Bildschirm. Alle graphische Elemente stammen aus einer Animationsbibliothek. Mit grüner Farbe (hellgrau in der Abbildung) sind Teilaussagen im Beweis markiert, die durch das rechtwinklige Dreieck visualisiert werden. Ansonsten wird Farbe nur zur Kennzeichnung des Flächeninhalts des Dreiecks benutzt.

D: Methode Die Visualisierung ist handkodiert und fest in das System integriert. Der Autor muß den Beweis sehr gut kennen, um den zugehörigen Algorithmus und dessen Animation zu erstellen.

E: Interaktion ZEUS wird über verschiedene Fenster und Schalter gesteuert. Der Benutzer kann das Tempo der Animation beeinflussen. Details der Visualisierung lassen sich nicht ausschalten. Der Benutzer kann die visuelle Darstellung nicht an seine Bedürfnisse anpassen. Ein später abspielbares Animationsskript wird nicht erstellt. Bei der Animation handelt sich um eine Art „Dia-Schau“. Einzelne „Dias“ lassen sich über ein spezielles Fenster direkt anwählen („Zeus Photo Album“).

F: Wirksamkeit Die Animation wurde nicht im Unterricht eingesetzt. Sie stellt lediglich eine Studie dar, die auf einem Algorithmenanimations-„Festival“ entstanden ist.

4.3.2 Gloors System

A: Anwendungsbereich Das in [GLOOR und STREITZ 1990] beschriebene System zur Visualisierung von Korrektheitsbeweisen von Algorithmen ist eine nur auf dem Apple Macintosh lauf—fähige Implementierung in HyperCard. Eine spezielle Logik oder ein konkreter Beweiskalkül ist vom System nicht vorgegeben, da der Beweis lediglich als umgangssprachlicher Text formuliert wird. Es wurde bisher nur ein Korrektheitsbeweis eines bestimmten Graphenalgorithmus visualisiert. Die Autoren sehen den Einsatzschwerpunkt in der Visualisierung von Korrektheitsbeweisen von Graphenalgorithmen. Die Größe der Beweise ist nur durch die Implementierung in HyperCard eingeschränkt, eine konkrete Beschränkung ist uns nicht bekannt.

B: Inhalt Formale Beweise können nicht visualisiert werden. Nicht formale Beweise werden als HyperCard-Anwendung implementiert. Die Regelanwendungen sowie die Aussagen werden als umgangssprachlicher Text formuliert und mit Hyperverweisen miteinander verbunden. Beweisregeln werden nicht visualisiert. Auf Basis von Animationen des Algorithmus wird im wesentlichen die Induktionsaussage im Beweis, die mit der Invariante des Algorithmus korrespondiert visualisiert (siehe Abbildung 4.3). Die Visualisierungen werden mit HyperCard vor der Laufzeit entwickelt.

C: Form Der Beweis wird auf einem Computer-Bildschirm dargestellt. Die Granularität ist im voraus durch eine fixe Implementierung festgelegt und kann durch den Benutzer nicht geändert werden.

D: Methode Die Visualisierung ist eine HyperCard Anwendung und fest in das System integriert. Beim vorliegenden Beispiel wurde eine zur Visualisierung des Algorithmus erstellte Algorithmenanimation zur Visualisierung des Korrektheitsbeweises

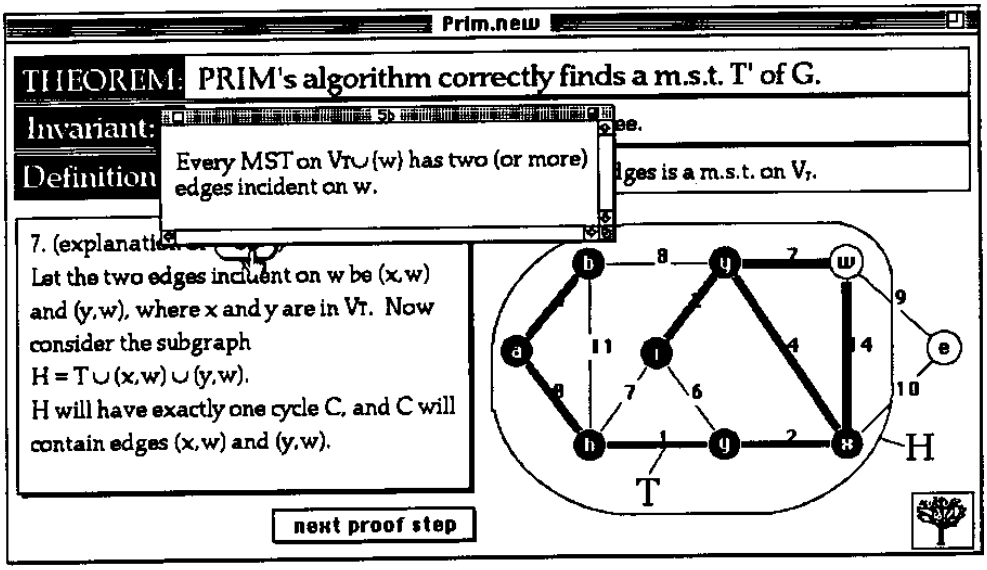


Abbildung 4.3: Bildschirmphoto von Gloors System

verwendet. Der Autor muß den Beweis sehr gut kennen, um die Animation zu erstellen. Der Benutzer kann die Visualisierung nicht an seine Bedürfnisse anpassen.

E: Interaktion Das Visualisierungssystem ist Teil von ANIMATED ALGORITHMS und verwendet daher dessen Steuerung. ANIMATED ALGORITHMS ist eine integrierte Hypermedia-Anwendung. Einzelne Beweisteile und die dazugehörigen visuellen Darstellungen werden über Hyperverweise angewählt. Details des Beweises können ausgelassen werden, indem nicht allen Hyperverweisen gefolgt wird. Einfluß auf den zeitlichen Verlauf der Visualisierung läßt sich nehmen.

F: Wirksamkeit Die Visualisierung ist ein „Abfallprodukt“ von ANIMATED ALGORITHMS. Sie wurde nicht im Unterricht eingesetzt. Im Vordergrund stand das Interesse der Autoren, ihr Animationssystem auch für den Bereich der Beweisvisualisierung zu erproben. Sie hoffen, daß „[ihre Arbeit] will stimulate research in the area and influence the generation of more systems for proof visualization“ [GLOOR et al. 1992].

4.3.3 Hyperproof

A: Anwendungsbereich Hyperproof [BARWISE und ETCHEMENDY 1998] ist ein Lernprogramm für den Apple Macintosh, mit dem Studierenden das Führen formaler Beweise erlernen können. Dem Ganzen ist ein erfahrungsbetontes Lernen („Learning by doing“) zugrunde gelegt. Das System ist von seiner Konzeption ein interaktiver Theorembeweiser. Nicht formale Beweise sind nicht Gegenstand des Systems.

Es lassen sich formale Aussagen der Prädikatenlogik mit einem Kalkül im Fitch-Stil [BARWISE und ETCHEMENDY 1993] beweisen. Die zugrundeliegende prädikatenlogische Sprache erlaubt lediglich, Aussagen über eine vorgegebene geometrische Welt zu formulieren und diese zu beweisen. Einschränkungen der Beweisgröße und Formeln sind uns nicht bekannt.

B: Inhalt Die Beweise werden textuell dargestellt und hierarchisch strukturiert. Der Name der bei einem Beweisschritt verwendete Beweisregel ist an der entsprechenden Stelle im Beweis vermerkt. Neben dieser rein textuellen Darstellung sind die atomare Formeln visuell mit geometrischen Objekten, wie Quadern, Tetraedern und Pyramiden, und deren räumliche Lage und Größe in Bezug gesetzt: Auf jedes graphische Objekt kann in einer Formeln mit einer Konstanten verwiesen werden, und die Größe und Lage dieser Objekte kann durch Prädikatensymbole beschrieben werden. Diese geometrische Welt läßt sich als Visualisierung eines prädikatenlogischen Modells für die gegebene formale Sprache auffassen. Die Beweisregeln und komplexere Formeln werden nicht visualisiert. Die Visualisierung wird live während der Laufzeit erstellt. Siehe auch Abbildung 4.4.

C: Form Als Darstellungsmedium dient der Bildschirm. Alle graphischen Elemente stammen aus einer fixen Menge geometrischer Objekte und deren räumlicher Beziehung. Farben werden nur zur besseren Unterscheidung der geometrischen Objekte verwendet. Es existiert sowohl eine zwei-dimensionale Sicht als auch eine alternative drei-dimensionale Sicht auf die Blockwelt. Details des Beweises und der Visualisierung lassen sich nicht ausblenden. Die Visualisierung ist statisch, insbesondere gibt es keine Animationen. Alternative Sichten auf den Beweis oder die Formeln (außer den drei Raumdimensionen) werden ebenfalls nicht unterstützt.

D: Methode Die Visualisierung ist handkodiert und fest in das System integriert. Mit Hilfe der Maus kann der Benutzer die geometrische Lage und Anzahl der geometrischer Objekte verändern.

E: Interaktion Das System wird über Schalter und Menüs gesteuert. Der Benutzer kommuniziert über verschiedenen Dialogfenster, mit denen zum Beispiel Formeln eingegeben werden können.

F: Wirksamkeit HYPERPROOF wird von seinen Autoren zur Ausbildung ihrer Studierenden in formaler Logik benutzt. Eine Studie über die Wirksamkeit existiert nicht. Es sind lediglich einige ausgewählte studentische Kommentare auf der WWW-Seite zum Programm zu finden.

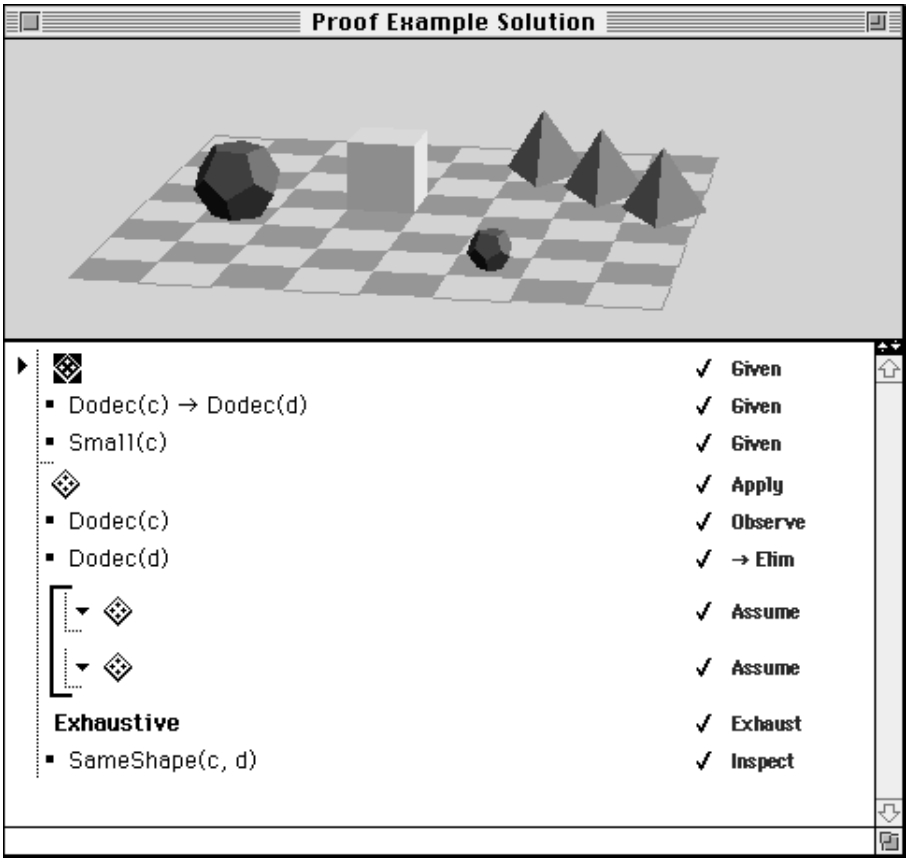


Abbildung 4.4: Ein Beweis in HYPERPROOF. Der Beweis ist hierarchisch strukturiert. Der Benutzer kann den Wahrheitsgehalt der Formeln visuell anhand der graphischen Darstellung eines Modells überprüfen. Geometrische Objekte in dieser Darstellung lassen sich mit den im Beweis verwendeten Konstantensymbole versehen (kommt in der Abbildung nicht vor).

4.3.4 Proofviews

A: Anwendungsbereich PROOFVIEWS [GRUNDY 1996a] stellt in HTML geschriebene strukturierte Beweise hierarchisch dar. PROOFVIEWS wird vornehmlich zur Visualisierung nicht formaler Beweiser benutzt, es wurde aber auch zur Visualisierung formaler Beweise im HOL System verwendet [GRUNDY und LÅNGBACKA 1997]. Zur Betrachtung der Beweisvisualisierungen genügt ein WWW-Stöberer. Die Beweisvisualisierungen werden unter dem Betriebssystem Unix erzeugt. Mit dem System werden Beweise von Aussagen in Prädikatenlogik erster Stufe behandelt. Die Beweise müssen im *Structured calculational proof format* geschrieben sein [BACK et al. 1997], einem Notationsstil, der eng an Kalküle des natürlichen Schließens angelehnt ist. Da lediglich die Struktur des Beweises visualisiert wird, handelt es sich bei PROOFVIEWS um eine Art „pretty printer". Es können nur Beweise mit wenigen geschachtelten Teilbeweisen visualisiert werden, da der Speicherbedarf exponentiell in der Anzahl der Teilbeweise anwächst.

B: Inhalt Der Beweistext wird als Hypertext dargestellt. Geschachtelte Teilbeweise lassen sich im WWW-Stöberer über Hyperverweise auf- und wieder verdecken. Dies ermöglicht es dem Betrachter, Details des Beweises individuell auszublenden und nur bei Bedarf genauer zu inspizieren. Siehe dazu auch Abbildung 4.5. Diese Form des hierarchischen Beweis-„browsing" wurde erstmals von Leslie Lamport vorgeschlagen [LAMPORT 1995]. Sie erlaubt dem Leser, eine ihm angemessene Sicht des Beweises zu wählen, um zum Beispiel alle uninteressanten Details eines Beweises zu verdecken.

C: Form Da der Beweis lediglich als HTML-Text am Bildschirm visualisiert wird, gibt es kein graphisches Vokabular. Ebenso werden keine Animationen eingesetzt. Farben werden lediglich zur Hervorhebung einiger HTML-Verweise, insbesondere zum Auf- und Verdecken von Teilbeweisen benutzt. Das System unterstützt keine alternativen Sichten auf den Beweis.

D: Methode Um mit PROOFVIEWS einen in HTML vorliegenden bereits strukturierten Beweis in eine stöberbare Version zu transformierten, markiert man mit speziellen Anweisungen im ursprünglichen HTML-Text die Teilbeweise per Hand.

E: Interaktion Die Beweisvisualisierung kann mit jedem WWW-Stöberer betrachtet werden. Der Benutzer interagiert über die Navigationsleiste des Stöberers und über die Verweise im Beweistext, mit denen Teilbeweise verdeckt und wieder aufgedeckt werden können.

F: Wirksamkeit PROOFVIEWS wurde bisher nicht im Ausbildungsbereich eingesetzt. Es ist Teil einer Erweiterung des HOL-Theorembeweiser und wird dort zur bes-

```
  A ==> ⌊A ∨ B⌋
= {Assuming the hypothesis, we can simplify the consequent.}
  A ==> ⌜true⌝
= {Boolean simplification.}
  true
```

(a) Verdeckter Teilbeweis

```
  A ==> ⌊A ∨ B⌋
= {Assuming the hypothesis, we can simplify the consequent.}
    ▷ <A>
      A ∨ B
    = {Replace A with true, since it is an assumption.}
      true ∨ B
    = {Boolean simplification.}
      true
  A ==> ⌜true⌝
= {Boolean simplification.}
  true
```

(b) Aufgedeckter Teilbeweis

Abbildung 4.5: Darstellung eines strukturierten Beweises mit PROOFVIEWS. Die blau hervorgehobenen Textteile erläutern jeweils informell die Anwendung einer Beweisregel auf die diesem Textteil voranstehenden logischen Aussagen (in der schwarzweiß-Abbildung nicht zu erkennen). Unterstrichene Beweisregeln weisen auf einen verdeckten Teilbeweis hin (a), der durch Anwählen mit der Maus aufgedeckt werden kann (b).

seren Darstellung formaler Beweise benutzt.

4.4 Studentische Probleme beim Verständnis von Beweisen

Meine zweijährige Erfahrung bei der Betreuung der Übungen zu den Pflichtvorlesungen „Informatik III“ und „Informatik IV“ an der Universität Karlsruhe und der Korrektur der zugehörigen Klausuren hat bei mir den bleibenden Eindruck hinterlassen, daß die meisten Studierenden kaum in der Lage sind, einen Beweis präzise und weitgehend exakt zu formulieren, auch wenn sie eine richtige Beweisidee gefunden haben. Andererseits haben sie keine ernsthaften Probleme, algorithmische Verfahren zu verstehen, auszuarbeiten oder anzuwenden. Die folgende kurze Analyse von Klausurergebnissen

soll dies anhand konkreter Zahlen verdeutlichen.

Die Klausur zur Vorlesung „Informatik III“ im Februar 1997 enthielt eine Aufgabe aus dem Bereich Formale Sprachen zum Thema Grammatiken und kontextfreie Sprachen. In einer Teilaufgabe sollten die Studierenden mit vollständiger Induktion beweisen, daß die von der in der Aufgabe vorgegebenen kontextfreien Grammatik erzeugten Sprache Teilmenge von der in der Teilaufgabe angegebenen Sprache ist. In der darauffolgenden Teilaufgabe war zu einem gegebenen Wort und einer Grammatik in Chomsky-Normalform der Cocke-Kasami-Younger-Algorithmus (CKY-Algorithmus) auszuführen.[3] Die dabei auszufüllende Pyramide war im Klausurtext vorgegeben. Die erste Teilaufgabe wurde mit 6 Punkten, die zweite mit 3 Punkten bewertet. Beide Problemstellungen wurden zuvor in der Vorlesung und in den Übungen mit etwa derselben Intensität behandelt. Die Schwierigkeit in der ersten Teilaufgabe bestand weniger darin, auf die richtige Beweisidee zu kommen (diese war simpel), sondern in der Formulierung des Beweises. Letzteres wurde exemplarisch für eine ähnliche Aufgabe in den Übungen zur Vorlesung behandelt. Bei der zweiten Teilaufgabe kam es darauf an, den Algorithmus mit Hilfe der visuellen Darstellung auf eine bestimmte Eingabe anwenden zu können. Auch dieses wurde in den Übungen behandelt. Abbildung 4.6 zeigt die Verteilung der Differenz der erzielten Punkte der Studierenden in der Beweisaufgabe zur CKY-Aufgabe. Bei der Bildung der Differenz wurden die Punkte der zweiten Teilaufgabe verdoppelt, um das Resultat besser interpretieren zu können.

Die Punkte (Kreise und *) rechts von 0 auf der unteren Achse entsprechen Studierenden, die bei der Beweisaufgabe tendenziell besser als bei der CKY-Aufgabe abschnitten. Punkte links davon entsprechen Studierenden, die beim CKY-Algorithmus tendenziell besser als bei der Beweisaufgabe abschnitten. Die mit einem Stern (*) bezeichneten Punkte sind Studierenden, die mindestens 1 Punkte (von 3 Punkten) in der CKY-Aufgabe und mindestens 1.5 Punkte (von 6 Punkten) in der Beweisaufgabe erzielt haben. Der Mittelwert von allen Studierenden liegt bei -2.544. Dies belegt, daß die Studierenden mit der CKY-Aufgabe erheblich weniger Probleme hatten als mit der Beweisaufgabe.

Die Sterne repräsentieren Studierende, bei denen anzunehmen ist, daß sie für beide Aufgaben ein gewisses Grundverständnis besitzen. Man kann bei diesen Studierenden davon ausgehen, daß sie beide Teilaufgaben verstanden haben und eine Idee zu deren Lösung besitzen. Die Häufung der Sterne (Mittelwert -1.631) zeigt, daß auch diese Studierenden besser bei der Anwendung eines zuvor gelernten und visuell geeignet darstellbaren Algorithmus abschneiden, als beim Ausführen einfacher Induktionsbeweise. Eine genauere Inspektion der gemachten Fehler in der Beweisaufgabe zeigte, daß die meisten dieser Studierenden eine konstruktive Idee hatten, wie der Beweis verlaufen müßte, aber bei der Umsetzung in eine Induktion scheiterten.

[3] Es handelt sich um die gleiche Aufgabe, von der die Werte in Tabelle 2.1 stammen.

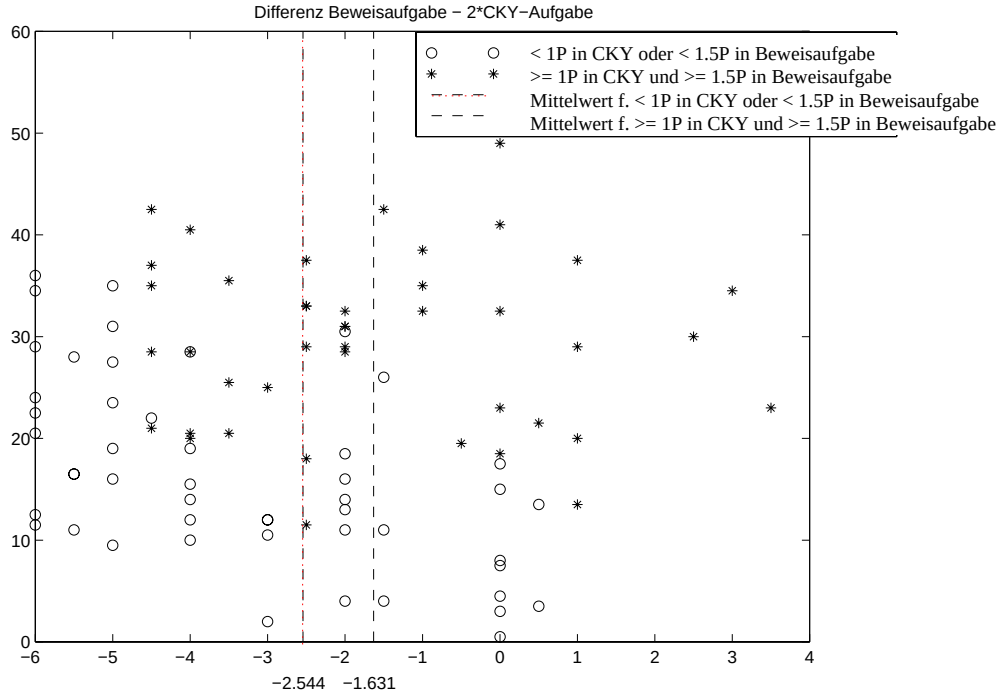


Abbildung 4.6: Beziehung zwischen den erzielten Punkte bei zwei Klausuraufgaben

4.5 Zusammenfassung und Ausblick auf das nächste Kapitel

Die hier vorgestellten Systeme stellen eine repräsentative Auswahl von Beweisvisualisierungen auf dem Rechner für den Ausbildungsbereich dar. Fast alle weiteren Versuche, wie die Animation zum Satz des Phytagoras oder eine unserer früheren Arbeiten [PAPE und SCHMITT 1997], sind lediglich *Ad-hoc*-Implementierungen eines bestimmte Beweises, die ohne methodische Grundlage oder gar ein spezielles Autorenwerkzeug entstanden sind. Ausnahmen sind lediglich PROOFVIEWS und HYPERPROOF. Allerdings haben beide Systeme nur einen eingeschränkten Anwendungsbereich: Mit PROOFVIEWS werden Beweise lediglich strukturiert dargestellt, und mit HYPERPROOF können lediglich formale Beweise zu einen ganz engen Anwendungsbereich visualisiert werden.

Die bisherigen Ansätze zur Beweisvisualisierung beruhen fast ausschließlich auf einer Koppelung zwischen dem Beweistext und zusätzlichen *statischen* Bildern. Sinnvoller wäre es, die Fähigkeiten der Studierenden aus dem Bereich der Algorithmen für das Lernen von Beweisen auszunutzen. Dies würde ihnen helfen, ungewohnte und als zu schwierig empfundene Beweise mit bekannten und verstandenen Konzepten zu verbinden. Von solch einem Ansatz berichtete zum Beispiel [WAGENKNECHT 1998]: Hörer einer Einführung in die Theoretische Informatik implementieren im Rahmen

der Übungen zur Vorlesung die dort vorgeführten Beweise als ausführbare Programme in der funktionalen Programmiersprache Scheme. Dies hilft zwar nicht, die Beweise direkt besser zu verstehen, zwingt die Studierenden aber, sich intensiver mit den Beweisen auseinanderzusetzen. In unserer Arbeit gehen wir einen Schritt weiter und betrachten den Beweistext selbst als eine Art „Algorithmus“, bei dem Schritt für Schritt logische Aussagen transformiert werden. Wir zeigen, wie sich ein Beiweistext mit Hilfe zusätzlicher graphischer Darstellungen der im Beweistext auftretenden Objekte veranschaulichen läßt. Dabei fassen wir folgende Ziele ins Auge:

- Entwicklung einer Ablaufvisualisierung eines deklarativen Beweises, um die algorithmischen Fähigkeiten der Studierenden für das Verständnis deklarativer Beweise auszunutzen.
- Heranführen der Studierenden an das hohe Abstraktionsniveau eines exakten Beweises durch graphische Veranschaulichung der im Beweis auftretenden Objekte anhand konkreter Instanzen.
- Fördern eines explorativen Lernstils, indem die Animation sich an die vom Benutzer eingegebenen Instanzen anpaßt.

Bei unserem Konzept steht eine systematische und methodische Entwicklung von Beweisvisualisierungen im Vordergrund, denn nur so läßt sich der ganze Entwicklungsprozeß vom rein umgangssprachlich formulierten Beweis bis zur Erstellung der Beweisvisualisierung mit spezieller Software unterstützen. Der derzeitige Mangel an solchen Systemen ist wohl der wesentliche Grund dafür, daß Beweisvisualisierungen auf dem Rechner – im Gegensatz zu Algorithmenanimationen – derzeit selten entwickelt und fast gar nicht im Ausbildungsbereich eingesetzt werden.

Kapitel 5

Animation strukturierter Beweise

Unser Ansatz zur Animation von Beweisen läßt sich in drei abgegrenzte, aber aufeinander aufbauende Teile untergliedern. Wir formulieren diese drei Teile als Fragen, die ein Autor zu beantworten hat, wenn er zu einem Beweis schrittweise eine begleitende Animation erstellen möchte.

1. Wie schreibe ich den Beweistext?
2. Wie finde ich zu den Objekten im Beweistext geeignete visuelle Darstellungen?
3. Wie finde ich eine Ablaufbeschreibung des Beweistextes, mit der sich die visuellen Darstellungen zu einer Animation zusammenfügen lassen?

Die erste Frage betrifft den Beweistext selbst: Soll er vollständig durch graphische Darstellungen ersetzt werden? Diese Frage läßt sich für uns mit „Nein“ beantworten, da derartige Visualisierungen in den Bereich logischen Schließens mit Diagrammen fallen und die dortigen Ansätze noch weit entfernt sind von einer umfassenden Visualisierung umfangreicher Beweise. Zu klären ist dann allerdings, wie der Beweistext zu schreiben ist: Bei einer rein umgangssprachlichen Formulierung kann eine Beweisanimation nur unzureichend mit Softwareunterstützung erzeugt werden. Der Beweistext muß also formale und semantisch genau spezifizierte Elemente enthalten, um diese automatisch mit dem Rechner bearbeiten zu können. Darüberhinaus soll der Beweisautor möglichst frei bei der Gestaltung und Formulierung des Beweistextes sein. Insbesondere wollen wir nicht von ihm verlangen, einen formalen Beweis zu schreiben. Für eine rechnergestützt Beweisvisualisierung ist also die Frage zu klären, welche formale Elemente der Beweistext enthalten soll und welche nicht. Siehe dazu Abschnitt 5.1. Es wird sich herausstellen, das grobe Strukturmerkmale für eine rechnergestützte Entwicklung ausreichend sind. Wir sprechen deswegen in unserer Arbeit von der Animation *strukturierter* Beweise. Auch wenn sich unsere Beispiele auf nicht formale Beweise beschränken, ist unser Ansatz ebenso auf formale Beweise übertragbar, sofern die formalen Beweise die dafür notwendige Strukturmerkmale besitzen.

Die zweite Frage betrifft die visuelle Darstellung zum Beweis: Wie finde ich für bestimmte Beweisteile geeignete Visualisierungen? Nach [GLOOR und STREITZ 1990]

liegen hier teilweise die Gründe für den derzeitigen Mangel an Beweisvisualisierungen:

> „The lack of previous work in the area is partly due to the fact that proofs often involve hypothetical and abstract objects created for the purpose of the proof. Such objects are difficult to visualize."

In Abschnitt 5.2.2 zeigen wir, wie zumindest für konstruktive Beweise aus weiten Bereichen der Theoretischen Informatik Visualisierung systematisch gefunden werden können. Basis unseres Ansatzes ist eine visuelle Darstellung eines Modells der bewiesenen Aussage. Dies ähnelt dem in HYPERPROOF an einem fixen, endlichen Modell verwirklichten Konzept. Im Gegensatz zu der dortigen statischen Visualisierung des Modells benutzen wir zusätzlich Animationen, um die konstruktiven Teile eines Beweises zu veranschaulichen.

In Abschnitt 5.2.4 befassen wir uns dann mit der Frage, wie sich aus den visuellen Darstellungen der Objekte im Beweistext eine Animation erzeugen läßt. Dazu muß zum Beweistext eine Ablaufstruktur gefunden werden, mit der sich die Visualisierungen zu einer Animation zusammengefügen lassen.

Unser Konzept zur Beweisanimation kann – wie schon bei den Begriffsdefinitionen und der Taxonomie im vorangegangenen Kapitel geschehen – in Analogie zur Algorithmenanimation interpretiert werden: So wie bei einer Algorithmenanimation der dynamische Ablauf des Algorithmus anhand einer graphischen Darstellung der Datenstrukturen und ihrer Änderungen veranschaulicht wird, veranschaulichen unsere Beweisanimationen den dynamischen Ablauf eines Beweistextes anhand einer graphischen Darstellung der logischen Aussagen und ihrer Änderungen bei Beweisregelanwendungen.

5.1 Wie schreibe ich den Beweistext?

Die Frage „Wie schreibe ich den Beweistext?" wollen wir durch die Angabe eines konkreten Beweisformats klären. Das Beweisformat muß dabei möglichst den folgenden Anforderungen genügen.

Der *Autor* des Beweises soll Details des Beweistextes weitgehend frei formulieren können, ohne durch das Beweisformat stark eingeschränkt zu werden. Insbesondere wollen wir keine formale Sprache vorgegeben, in der die logischen Aussagen geschrieben werden müssen, also keine Vorgabe von logischen Konnektiven, Prädikaten oder Sorteninformationen. Da wir eine umfassende Menge von Beweisen abdecken wollen und nicht, wie beim logischen diagrammatischen Schließen nur einen spezialisierten Bereich, muß das Beweisformat eine möglichst flexible Beweisführung erlauben. Insbesondere darf kein fester, einschränkender Satz von Beweisregeln vorgegeben werden. Wie ein Beweisschritt begründet wird, soll in Verantwortung des Autors bleiben.

Für den *Leser* soll das Beweisformat möglichst lesbar sein. Es soll ihm helfen, den Beweistext zu überschauen und besser nachzuvollziehen.

Um das Erstellen der Beweisanimation teilweise zu *automatisieren*, muß das Format auch einige formale Element enthalten, allerdings nur so viele, daß der Autor nicht zu stark in seinen Gestaltungsmöglichkeiten eingeschränkt wird. Ein weiteres Kriterium bei einer rechnergestützten Erstellung von Beweisanimationen ist die Wartbarkeit des Beweises und der zugehörigen Animation: Kleine Änderungen im Beweistext sollen später keine oder nur wenige Änderungen bei dessen Visualisierung nach sich ziehen. Dies ist insbesondere bei einer Teilautomatisierung der Beweisvisualisierung wichtig, weil dabei einige Elemente manuell vom Autor implementiert werden müssen.

Das Resultat unserer Bemühungen, die *Beweisanimation*, veranschaulicht die konstruktiven Elemente des Beweistextes. Das Beweisformat sollte deswegen schon selbst einen eher betont konstruktiven Charakter haben.

Es gibt in der Literatur einige Beweisformate, die entwickelt wurden, um das Schreiben korrekter (nicht formaler) Beweise zu fördern. Formate, die auf Beweiskalkülen des natürlichen Schließen basieren, zum Beispiel [BACKHOUSE 1989], haben wir nicht weiter betrachtet, da sie zu stark an bestimmte formale Beweisregeln gekoppelt sind. Darüberhinaus sind derartig aufbereitete Beweise unserer Meinung nach nicht gut lesbar.

Mit dem Beweisformat von Leslie Lamport [LAMPORT 1995] wird ausgehend von einer Beweisskizze durch Dekomposition in immer kleinere Teilbeweise, schrittweise ein stark formalisierter Beweis entwickelt. Das Beweisformat dient vornehmlich dem Autor dazu, sich selbst von der Korrektheit einer Aussage zu überzeugen. Die Beweise in diesem Format sind nicht immer gut lesbar, da sie aufgrund der Methodik sehr stark ineinander verschachtelte Teilbeweise enthalten.

Das unseren Anforderungen am nächsten kommende existierende Beweisformat ist das *Structured calculational proof format*, welches uns schon in Zusammenhang mit PROOFVIEWS im vorangegangen Kapitel begegnet ist. Da es bis auf wenige Details alle für uns notwendigen Merkmale besitzt, haben wir es zur Grundlage unserer Beweisanimationen gemacht. Wir betrachten zuerst im nächsten Abschnitt dessen unstrukturierte Variante, dann das *Structured calculational proof format* selbst und anschließend einige von uns hinzugefügten Erweiterungen.

5.1.1 „Calculational proof format“

Das *Calculational proof format* [DIJKSTRA und SCHOLTEN 1990] ist ein von Edsger W. Dijkstra entwickelter Formalismus, der ursprünglich dazu diente, um die systematische Entwicklung eines Programms aus dessen formaler Spezifikation zu unterstützen. Das Beweisformat wird mittlerweile auch in einer Einführung in die formale Prädikatenlogik und deren Anwendungen in der systematischen Programmentwicklung benutzt [GRIES und SCHNEIDER 1994]. Wir orientieren uns in unserer Darstellung im wesentlichen an den in [GRIES und SCHNEIDER 1994] verwendeten Notationen.

Die Grundidee beim *Calculational proof format* ist es, Beweisschritte als syntaktische Transformationen zwischen Booleschen Formeln aufzufassen, das heißt, diese

Transformationen sind Berechnungen (*calculations*) auf Formeln (*structures*, Strukturen). Im *Calculational proof format* wird eine Aussage (Struktur) bewiesen bzw. widerlegt, indem ausgehend von der Aussage eine Folge von Beweisschritten (Berechnungen) im Wahrheitswert `true` oder `false` endet. Dies betont den für uns wichtigen konstruktiven Charakter von Beweisen. Die Beweisschritte sind linear angeordnet, um den Beweis für den Menschen lesbarer zu gestalten. Die Granularität der Beweisschritte sollte so gewählt sein, daß der Mensch jeden Beweisschritt rein syntaktisch nachvollziehen kann. Um ihm dies zu erleichtern, ist an jedem Beweisschritt eine Rechtfertigung in Form eines Kommentars, der zum Beispiel den Namen der verwendeten Beweisregel angibt beigefügt. Rechtfertigungen werden in geschweiften Klammern an den zugehörigen Beweisschritt geschrieben. Der Wahrheitswert einer Booleschen Struktur A wird durch den Operator $[\cdot]$ mit $[A]$ „ausgewertet“. Die folgenden schematischen Beweisschritte illustrieren einen „Beweis“ einer Booleschen Struktur A:

$$
\begin{array}{ll}
 & [A] \\
\equiv & \quad \{ \text{ Rechtfertigung für } [[A] \equiv [B]] \ \} \\
 & [B] \\
\equiv & \quad \{ \text{ Rechtfertigung für } [[B] \equiv [C]] \ \} \\
 & [C] \\
\equiv & \quad \{ \text{ Rechtfertigung für } [[C] \equiv \texttt{true}] \ \} \\
 & \texttt{true}
\end{array}
$$

Der Operator $[\cdot]$ besitzt noch eine zweite Bedeutung, die ihm seinen Namen – *Überall-Operator*[1] – gegeben hat: Sind A und B reelle Matrizen, so definiert $C = A + B$ die durch elementweise Addition resultierende reelle Matrix C. Sind hingegen A und B Boolesche Matrizen, so bezeichnet $A = B$ normalerweise nicht die Boolesche Matrix, die durch elementweise Boolesche Verknüpfung von A und B entsteht, sondern den Wahrheitswert `true` (`false`), falls A und B elementweise gleich sind (oder nicht). Damit alle Operatoren uniform gehandhabt werden, wird durch den Überall-Operator mit $[A = B]$ angezeigt, daß es sich um einen elementweisen Vergleich handelt, der in einem Wahrheitswert resultiert, d.h., A und B müssen *überall* übereinstimmen, damit $[A = B] = \texttt{true}$ gilt. Ansonsten bezeichnet $A = B$ analog zu $A + B$ die elementweise Verknüpfung mit $=$.

Zusätzlich zum Gleichheitssymbol $=$ wird im Fall von Booleschen Strukturen ein weiteres Symbol $\equiv$ für Gleichheit zugelassen. Bei Gleichheit auf Booleschen Strukturen handelt es sich um die logische Äquivalenz. Da diese assoziativ ist, ist $\equiv$ im Gegensatz zu $=$ immer assoziativ. Desweiteren hat $\equiv$ eine kleinere Präzedenz als alle anderen Operatoren und erlaubt deswegen, bei komplexeren Ausdrücken Klammern einzusparen.

Bei einem „richtigen“ Beweis wird der explizite Bezug zu einem Beweisschritt in einer Rechtfertigung weggelassen, im obigen Beispiel etwa $[A] \equiv [B]$. Die Rechtfertigung muß sich nicht notwendigerweise auf eine vorher festgelegte Beweisregel

[1] Engl.: *everywhere operator*

beziehen, sondern kann auch ein kurzer informeller Kommentar sein. Ebenso ist es nicht zwingend, die zu transformierenden Booleschen Strukturen als stark formalisierte Aussagen zu notieren. Der Autor ist weitgehend frei in der Wahl des Formalisierunggrads.

Ist die zu beweisende Behauptung die Äquivalenz zweier Booleschen Strukturen A und D, so muß $[A = D] \equiv \texttt{true}$ gezeigt werden. Ein Beweis davon stellt sich schematisch etwa wie folgt dar:

$$
\begin{array}{ll}
 & [A = D] \\
\equiv & \quad \{ \text{ Rechtfertigung für } [A = D] \equiv [B = D] \ \} \\
 & [B = D] \\
\equiv & \quad \{ \text{ Rechtfertigung für } [B = D] \equiv [C = D] \ \} \\
 & [C = D] \\
\equiv & \quad \{ \text{ Rechtfertigung für } [C = D] \equiv \texttt{true} \ \} \\
 & \texttt{true}
\end{array}
$$

Dieser Beweis ist durch die häufige Verwendung von D etwas schwer zu lesen. Das *Calculational proof format* bietet deswegen eine verkürzte Schreibweise an, bei der die Transformation von $[A = D]$ in `true` abkürzend als Transformation von A in D notiert wird. Es werden also nicht skalare Werte, sondern Strukturen transformiert. Obiger Beweis kann dann abkürzend wie folgt notiert werden.

$$
\begin{array}{ll}
 & A \\
\equiv & \quad \{ \text{ Rechtfertigung für } [A \equiv B] \ \} \\
 & B \\
\equiv & \quad \{ \text{ Rechtfertigung für } [B \equiv C] \ \} \\
 & C \\
\equiv & \quad \{ \text{ Rechtfertigung für } [C \equiv D] \ \} \\
 & D
\end{array}
$$

Quantoren haben das Format $\forall : V : B : T$ mit einer Variablen V (oder einer Folge V von Variablen), einer Booleschen Struktur B, die den quantifizierten Bereich angibt und einer Booleschen Struktur T, welche die Aussage repräsentiert, über die quantifiziert wird. $\forall : V : B : T$ ist selbst wieder eine Boolesche Struktur. Der Ausdruck ist ein Skalar, falls B und T skalar sind. In unseren Beispielen weiter unten benutzen wir ein etwas weniger formales Format für die Quantifizierung und schreiben „$\forall x \in T : B$:“ statt $\forall : x : B : T$. Ein Beweis von $\forall x :: [A.x \equiv D.x])$ sieht schematisch etwa wie folgt aus (Bereich und Typ sind weggelassen):

$$
\begin{array}{ll}
 & \forall : x :: A.x \\
\equiv & \quad \{ \text{ Rechtfertigung für } (\forall : x :: [A.x \equiv B.x]) \ \} \\
 & \forall : x :: B.x \\
\equiv & \quad \{ \text{ Rechtfertigung für } (\forall : x :: [B.x \equiv C.x]) \ \} \\
 & \forall : x :: C.x \\
\equiv & \quad \{ \text{ Rechtfertigung für } (\forall : x :: [C.x \equiv D.x]) \ \} \\
 & \forall : x :: D.x
\end{array}
$$

Um die Beweisführung zu vereinfachen, können im *calculcational proof format* freie Variablen benutzt werden, so daß sich der vorangehender Beweis per Konvention auch in folgender vereinfachter Form notieren läßt (Bereich und Typ weggelassen):

Wir stellen für jedes x folgendes fest:

$A.x$
$\equiv$ { Rechtfertigung für $(\forall : x :: [A.x \equiv B.x])$ }
$B.x$
$\equiv$ { Rechtfertigung für $(\forall : x :: [B.x \equiv C.x])$ }
$C.x$
$\equiv$ { Rechtfertigung für $(\forall : x :: [C.x \equiv D.x])$ }
$D.x$

Analog kann natürlich auch mit dem Existenzquantor verfahren werden.

Beispiel 1

Der folgende stark formalisierte Beweis soll einen Eindruck darüber geben, wie das *Calculational proof format* für das Erstellen exakter, nach unseren Definitionen aber nicht formaler Beweise benutzt werden kann. Wir geben einen Beweis der simplen Aussage

$$\forall a, b, c, d \in \mathbb{N} : a \leq b \wedge c \leq d \Rightarrow a + c \leq b + d \ .$$

Im Beweis verwenden wir zwei spezielle Regeln, um die in der Aussage vorkommende Ungleichheit $\leq$ zweier natürlicher Zahlen zu behandeln.

1. Die Symmetrie von $\leq$ für einen beliebigen Term t, der eine natürliche Zahl repräsentiert: `true` $\equiv [t \leq t]$, Regel (1), und

2. es ist erlaubt einen Term s, der eine natürliche Zahl repräsentiert, auf der rechten Seite einer Ungleichung $\leq$ durch einen Term t zu ersetzen, für den $[s \leq t] \equiv$ `true` bewiesen wurde: $[l \leq r] \Rightarrow [l \leq r[t/s]]$, Regel (2), wobei $r[t/s]$ die Substitution von s durch t in r bezeichnet. Da wir keine Subtraktion betrachten, ist diese Regel für natürliche Zahlen korrekt.

Desweiteren verwenden wir noch zwei einfache aussagenlogische Umformungen.

3. Die Abschwächung von $\wedge$: $[A \wedge B] \Rightarrow [A]$, Regel (3), und

4. die Äquivalenz $[A] \equiv [A \wedge true]$, Regel (4), für alle Booleschen Strukturen A und B.

Für alle $a, b, c, d \in \mathbb{N}$ gilt:

$[a \leq b \wedge c \leq d]$
$\equiv$ { Regel (4) }
$[a \leq b \wedge c \leq d \wedge true]$

$\equiv$ { Symmetrie von $\leq$ für $a + c$, Regel (1) }
$[a \leq b \wedge c \leq d \wedge a + c \leq a + c]$
$\Rightarrow$ { Ersetze a durch b auf der rechten Seite von $a + c \leq a + c$, Regel (2) }
$[a \leq b \wedge c \leq d \wedge a + c \leq b + c]$
$\Rightarrow$ { Abschwächung von $\wedge$, Regel (3) }
$[c \leq d \wedge a + c \leq b + c]$
$\Rightarrow$ { Ersetze c durch d auf der rechten Seite von $a + c \leq b + c$, Regel (2) }
$[c \leq d \wedge a + c \leq b + d]$
$\Rightarrow$ { Abschwächung von $\wedge$, Regel (3) }
$[a + c \leq b + d]$

Dieser Beweis ist für alle natürlichen Zahlen a, b, c und d eine Transformation vom Booleschen Wert $[a \leq b \wedge c \leq d]$ in den Booleschen Wert $[a + c \leq b + d]$. Aufgrund der Semantik der Implikation $\Rightarrow$ und des Quantors $\forall$, stellt diese Transformation einen Beweis für die Aussage $\forall a, b, c, d : \mathbb{N} : a \leq b \wedge c \leq d \Rightarrow a + c \leq b + d$ dar. Der Detaillierungsgrad des Beweises ist recht hoch. Dies erlaubt ein einfaches, schrittweises Überprüfen der einzelnen Beweisschritte durch den Schreiber oder durch einen Leser des Beweises. Für fortgeschrittene Leser könnten die einfachen aussagenlogischen Umformungen weggelassen werden.

5.1.2 „Structured calculational proof format“

Das *Calculational proof format*, stößt – was die Lesbarkeit der Beweise angeht – an Grenzen, wenn Beweise geführt werden, die aus vielen Teilbeweisen bestehen. Das *Structured calculcational proof format* [BACK et al. 1996, BACK et al. 1997] erweitert das Beweisformat um zusätzliche hierarchische Strukturierungsmittel, die es erlauben, Teilbeweise an der Stelle im Beweis auszuführen, an der sie verwendet werden. Betrachte folgenden schematischen Beweis im *Calculational proof format* für die Aussage $[A \wedge P \equiv Z] = \texttt{true}$ (aus [BACK et al. 1996, Seite 3]).

$A \wedge P$
$\equiv$ { Rechtfertigung für $[P \equiv Q]$ }
$A \wedge Q$
$\equiv$ { Rechtfertigung für $[Q \equiv R]$ }
$A \wedge R$
$\equiv$ { Rechtfertigung für $[A \equiv B]$ }
$B \wedge R$
$\equiv$ { Rechtfertigung für $[B \equiv C]$ }
$C \wedge R$
$\equiv$ { Rechtfertigung für $[C \wedge R \equiv Y]$ }
Y
$\equiv$ { Rechtfertigung für $[Y \equiv Z]$ }
Z

Beweise dieser Gestalt werden durch das wiederholte Auftreten ganzer Teilformeln wie A oder R sehr schnell für den Menschen unübersichtlich. Um die Wiederholungen zu vermeiden und solche Beweise trotzdem linear darzustellen, müssen im *Calculational proof format* die Teilbeweise ausgegliedert und getrennt bewiesen werden. Sie können dann im nachfolgenden Beweis innerhalb einer Rechtfertigung als ein Lemma benutzt werden.

Lemma 2 $[P \equiv R] \equiv \texttt{true}$.

$$
\begin{array}{ll}
 & P \\
\equiv & \quad \{ \text{ Rechtfertigung für } [P \equiv Q] \ \} \\
 & Q \\
\equiv & \quad \{ \text{ Rechtfertigung für } [Q \equiv R] \ \} \\
 & R
\end{array}
$$

Lemma 3 $[A \equiv C] \equiv \texttt{true}$

$$
\begin{array}{ll}
 & A \\
\equiv & \quad \{ \text{ Rechtfertigung für } [A \equiv B] \ \} \\
 & B \\
\equiv & \quad \{ \text{ Rechtfertigung für } [B \equiv C] \ \} \\
 & C
\end{array}
$$

Theorem 4 *Es gilt* $[A \wedge P \equiv Z]$.

$$
\begin{array}{ll}
 & A \wedge P \\
\equiv & \quad \{ \text{ Lemma 2 } \} \\
 & A \wedge R \\
\equiv & \quad \{ \text{ Lemma 3 } \} \\
 & C \wedge R \\
\equiv & \quad \{ \text{ Rechtfertigung für } [C \wedge R \equiv Y] \ \} \\
 & Y \\
\equiv & \quad \{ \text{ Rechtfertigung für } [Y \equiv Z] \ \} \\
 & Z
\end{array}
$$

Bei dieser Umstrukturierung des Beweises, die unternommen wurde, um seine Lesbarkeit zu erhöhen, werden nun Teilbeweise vorweg geführt, ohne daß der Leser Kenntnis über den Beweis der eigentlich interessierenden Aussage besitzt. Dies kann in vielen Situationen das Verständnis der ausgegliederten Teilbeweise behindern, insbesondere wenn es sich um rein technische Aussagen handelt, die sich nicht sinnvoll als eigenständiges Lemma formulieren lassen. Wird hingegen der Hauptbeweis vorweg betrachtet, dann werden darin Rechtfertigungen benutzt, die auf Ergebnisse verweisen,

die erst hinterher bewiesen werden. In beiden Fällen ist die Präsentation des Beweis für den menschlichen Leser eher schlechter geworden als besser.

Statt nun Teilbeweise ganz auszugliedern, werden sie im *Structured calculational proof format* hierarchisch an der Stelle ihrer Verwendung in den Beweis eingegliedert und durch Einrücken und vorangestellten $\bullet$ und nachgestellten $\cdot$ als Teilbeweis kenntlich gemacht. Ferner wird im Antezedent des Beweisschritts die im Teilbeweis transformierte Teilformel P durch $\llcorner P \lrcorner$ und in der Konklusion die resultierende Teilformel R durch $\ulcorner R \urcorner$ markiert:

$$
\begin{array}{ll}
 & A \wedge \llcorner P \lrcorner \\
\equiv & \quad \{ \text{ Teilbeweis für } [P \equiv R] \ \} \\
 & \quad\quad \bullet \quad P \\
 & \quad\quad \equiv \quad \{ \text{ Rechtfertigung für } [P \equiv Q] \ \} \\
 & \quad\quad\quad Q \\
 & \quad\quad \equiv \quad \{ \text{ Rechtfertigung für } [Q \equiv R] \ \} \\
 & \quad\quad\quad R \\
 & \quad\quad \cdot \quad \llcorner A \lrcorner \wedge \ulcorner R \urcorner \\
\equiv & \quad \{ \text{ Teilbeweis für } [A \equiv C] \ \} \\
 & \quad\quad \bullet \quad A \\
 & \quad\quad \equiv \quad \{ \text{ Rechtfertigung für } [A \equiv B] \ \} \\
 & \quad\quad\quad B \\
 & \quad\quad \equiv \quad \{ \text{ Rechtfertigung für } [B \equiv C] \ \} \\
 & \quad\quad\quad C \\
 & \quad\quad \cdot \quad \ulcorner C \urcorner \wedge R \\
\equiv & \quad \{ \text{ Rechtfertigung für } [C \wedge R \equiv Y] \ \} \\
 & Y \\
\equiv & \quad \{ \text{ Rechtfertigung für } [Y \equiv Z] \ \} \\
 & Z
\end{array}
$$

Bei der Verwendung von Teilbeweisen ist Vorsicht geboten. Falls der Teilbeweis keine äquivalente Beziehung zwischen zwei Booleschen Strukturen beweist, hängt die korrekte Verwendung des Teilbeweises von den Monotonieeigenschaften des Operators ab, dessen Argument durch den Teilbeweis ersetzt wird. Soll zum Beispiel aus $\llcorner A \lrcorner \Rightarrow P$ die Aussage $\ulcorner C \urcorner \Rightarrow P$ gefolgert werden, so muß wegen der Antimonotonität des erstens Arguments von $\Rightarrow$ ein Teilbeweis für $\neg A \Rightarrow \neg C$, also $C \Rightarrow A$ bzw. $A \Leftarrow C$, geführt werden.

$$
\begin{array}{ll}
 & \llcorner A \lrcorner \Rightarrow P \\
\equiv & \quad \{ \text{ Teilbeweis für } [A \Leftarrow C] \ \}
\end{array}
$$

- A
  $\Leftarrow$ { Rechtfertigung für $[A \Leftarrow B]$ }
  B
  $\equiv$ { Rechtfertigung für $[B \equiv C]$ }
  C

. $\ulcorner C \urcorner \Rightarrow P$

Es ist also darauf zu achten, daß die Ersetzung wahrheitserhaltend ist. Die korrekte Verwendung eines Teilbeweises ist, wie auch bei den anderen im Beweis auftretenden Begründungen, dem Autor überlassen.

Beim Führen eines Teilbeweises kann der Autor oftmals einen Vorteil aus der Semantik des logischen Operators ziehen. Wird zum Beispiel der Antezedent A einer Implikation $A \Rightarrow B$ in einem Teilbeweis umgeformt, so kann der Autor aufgrund der Semantik von $\Rightarrow$ im Teilbeweis $[A] \Rightarrow \texttt{true}$ annehmen. Diese Art der Ausnutzung der Semantik logischer Operatoren in Booleschen Strukturen beim Führen von Teilbeweisen hilft oftmals, um kurze und lesbare Beweise zu schreiben. Sie wurde zum Beispiel explizit in das hierarchische Beweisformat von Lamport integriert [LAMPORT 1995]. Unser vorangegangenes Beispiel kann im *Structured calculational proof format* wie folgt notiert werden:

Beispiel 2
Mit Hilfe eines Teilbeweises und der Ausnutzung der Semantik von $\Rightarrow$ kann der Beweis aus Beispiel 1 wie folgt umgeschrieben werden:

Für alle natürlichen Zahlen a, b, c, d gilt:
$\llcorner [a \leq b \wedge c \leq d] \lrcorner$
$\Rightarrow$ { Teilbeweis von $[a \leq b \wedge c \leq d] \Rightarrow [a + c \leq b + d]$ }

Annahme: $[a \leq b \wedge c \leq d] \equiv \texttt{true}$, also $[a \leq b] \equiv [c \leq d] \equiv \texttt{true}$
wegen der Semantik von $\wedge$
- `true`
  $\equiv$ { Symmetrie von $\leq$ für $a + c$, Regel (1) }
  $[a + c \leq a + c]$
  $\Rightarrow$ { Regel (2) und Annahme $[a \leq b] \equiv \texttt{true}$ }
  $[a + c \leq b + c]$
  $\Rightarrow$ { Regel (2) und Annahme $[c \leq d] \equiv \texttt{true}$ }
  $[a + c \leq b + d]$

. $\ulcorner [a + c \leq b + d] \urcorner$

Im Gegensatz zu Lamports Beweisformat, enthalten Beweise, die im *Structured calculational proof format* geschrieben sind, meist weniger tief verschachtelte Teilbeweise. Aber auch schon bei ein oder zwei Teilbeweisen leidet die Lesbarkeit des

Beweistextes etwas. In [GRUNDY 1996b] wird deswegen eine Hypertextdarstellung für das *Structured calculational proof format* vorgestellt, mit dem sich geschachtelte Teilbeweise individuell durch den Leser auf- und wieder verdecken lassen, siehe dazu auch die Erklärungen zu PROOFVIEWS im vorangegangenen Kapitel.

5.1.3 Erweiterungen des Beweisformats

Das *Calculational proof format* trennt logische Aussagen (dort Boolesche Strukturen genannt) von den mit einem Kommentar versehenen Regelanwendungen. Die strukturierte Variante erweitert das Format um geschachtelte Teilbeweise; Quantoren werden informell behandelt. Weitere oft vorkommende Strukturierungsmittel wie Induktion und Fallunterscheidungen sind nicht explizit Bestandteil des Beweisformats. Diese Konstrukte sind für uns aber von Bedeutung, da sie angeben, auf welche Art die logischen Aussagen während des Beweises transformiert werden. Für unsere rechnergestützte Erstellung einer Ablaufvisualisierung legen wir deswegen folgende Notation für Induktion und Fallunterscheidung fest.

Fallunterscheidung

In einer Fallunterscheidung wird, etwa in einem Beweis von $[A \vee B \vee C \Rightarrow D]$, nach den disjunktiv verknüpften logischen Aussagen A, B und C unterschieden. Dies kann zwar im *Structured calculational proof format* durch drei aufeinanderfolgende Teilbeweise notiert werden, ist aber dann nur schwer für den Leser als Fallunterscheidung erkennbar, wie folgendes schematisches Beispiel veranschaulichen soll:

$\llcorner A \lrcorner \vee B \vee C$
$\Rightarrow$ { Teilbeweis für $[A \Rightarrow D]$ }

- A
  $\Rightarrow$ { ... }
  D

$\ulcorner D \urcorner \vee \llcorner B \lrcorner \vee C$
$\Rightarrow$ { Teilbeweis für $[B \Rightarrow D]$ }

- B
  $\Rightarrow$ { ... }
  D

$[D \vee \ulcorner D \urcorner \vee C]$
$\Rightarrow$ { Teilbeweis für $[C \Rightarrow D]$ }

- C
  $\Rightarrow$ { ... }
  D

$\cdot \quad D \vee D \vee \ulcorner D \urcorner$
$\Rightarrow$ { Einfache Aussagenlogik }
D

Durch explizite Angabe der Fälle läßt sich ein derartiger Beweis lesbarer gestalten. Die Fallunterscheidung führen wir als Teilbeweis, die Fälle sind fortlaufend numeriert, und jeder Fall besteht aus einem eigenen Beweis. Die letzte Boolesche Struktur dieser Beweise ist in jedem der Fälle identisch.

$\llcorner A \vee B \vee C \lrcorner$
$\Rightarrow$ { Teilbeweis für $[A \vee B \vee C \Rightarrow D]$ }

- Beweis durch Fallunterscheidung nach A, B und C
  1. A:
     A
     $\Rightarrow$ { ... }
     D
  2. B:
     B
     $\Rightarrow$ { ... }
     D
  3. C:
     C
     $\Rightarrow$ { ... }
     D

$\cdot \quad \ulcorner D \urcorner$

Bei folgendem Spezialfall einer Fallunterscheidung wird nach den Eigenschaften einer Variablen unterschieden, die in einem Term oder einer logischen Aussage enthalten ist. Die spezifische Eigenschaft der Variablen wird im zugehörigen Teilbeweis ausgenutzt. Wir betrachten als Beispiel den schematischen Beweis der Aussage $\forall n \in \mathbb{Z} : P(n) \Rightarrow Q(n)$ und machen eine Fallunterscheidung nach $n > 0, n = 0$ und $n < 0$:

Für alle $n \in \mathbb{Z}$ gilt:
$\llcorner P(n) \lrcorner$
$\Rightarrow$ { Teilbeweis für $[P(n) \Rightarrow Q(n)]$ }

- Beweis durch Fallunterscheidung nach $n > 0, n = 0$ und $n > 0$
  1. $n > 0$:
     $P(n)$
     $\Rightarrow$ { Weil $[n > 0]$ gilt }
     $Q(n)$

2. $n = 0$:

   $P(n)$

   $\Rightarrow$ { Weil $[n = 0]$ gilt }

   $Q(n)$

3. $n < 0$:

   $P(n)$

   $\Rightarrow$ { Weil $[n < 0]$ gilt }

   $Q(n)$

. $\ulcorner Q(n) \urcorner$

In diesem Beweis ist die zu beweisende Aussage $\forall n \in \mathbb{N} : P(n) \Rightarrow Q(n)$ gewissermaßen implizit in $\forall n \in \mathbb{N} : (n > 0 \wedge P(n) \vee n = 0 \wedge P(n) \vee n < 0 \wedge P(n)) \Rightarrow Q(n)$ umgeformt und dann obiges Schema für eine Fallunterscheidung angewendet worden.

Für die Lesbarkeit eines Beweis mit Fallunterscheidung ist wichtig, daß zu Anfang der Fallunterscheidung dem Leser klar ist, daß die Fallunterscheidung erschöpfend ist: In unserem ersten Beispiel durch Angabe der expandierten Teilstrukturen A, B und C, im zweiten durch Angabe aller möglichen Fälle $n > 0, n = 0, n < 0$.

Induktion

Eine Induktion besteht aus einer Induktionshypothese, dem Induktionsparameter, dem Induktionsanfang und einem Induktionsschluß mit der Verwendung der Induktionshypothese. Die Induktionshypothese ist in vielen Fällen einfach die zu beweisende Aussage. Sie braucht deswegen nicht explizit im Beweis vorzukommen. Unsere Erweiterung des Beweisformates sieht eine explizite Angabe des Induktionsparameters, des Induktionsanfangs und des Induktionsschlus (inklusive Angabe der Art des Induktionsschritts und Verwendung der Induktionshypothese) vor. David Gries benutzt in seiner Variante des *Calculational proof formats* [GRIES und SCHNEIDER 1994] eine ähnliche Syntax. Wir stellen das Konstrukt beispielhaft in einem Beweis der simplen Aussage $[\forall n \in \mathbb{N} : n! = n \cdot n - 1 \cdot \ldots \cdot 2 \cdot 1]$ vor. Die Fakultät einer natürlichen Zahl sei dabei rekursiv durch $1! = 1$ und $n! = n \cdot (n - 1)!$ definiert.

Beweis durch vollständige Induktion nach $n \in \mathbb{N}$:
Induktionsanfang

1. $n = 1$:

   $n!$

   $=$ { $n = 1$ }

   $1!$

   $=$ { Def. von ! }

   1

$$
\begin{array}{ll}
= & \{\ n = 1\ \} \\
 & n \cdot \ldots \cdot 1
\end{array}
$$

Induktionsschritt von $n-1$ auf n

$$
\begin{array}{ll}
 & n! \\
= & \{\ \text{Def. von !}\ \} \\
 & n \cdot (n-1)! \\
= & \{\ \text{Induktionsvoraussetzung für } n-1\ \} \\
 & n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1
\end{array}
$$

Der Induktionsbeweis besteht also aus zwei Teilen, die durch Angabe von Induktionsanfang und Induktionsschluß markiert sind. Die Beweise in diesen beiden Teilen sind im *Structured calculational proof format* notiert. Die Induktionshypothese wird als Regelanwendung mit einem informellen Kommentar angegeben. Bei einem Basisfall mit mehreren Fällen kann einfach unser vorangegangenes Konstrukt zur Fallunterscheidung verwendet werden.

5.2 Wie finde ich geeignete visuelle Darstellungen?

Wie wir bisher gesehen haben, ist das wesentliche Strukturierungsinstrument in unserem auf dem *Structured calculational proof format* basierenden Beweisformat eine Unterscheidung in logische Aussagen (Formeln, Booleschen Strukturen) einerseits und Anwendungen von Beweisregeln auf diese Aussagen andererseits. Bevor wir uns der Visualisierung von Beweisregel zuwenden, legen wir in diesem Abschnitt dar, wie sich die logischen Aussagen sinnvoll graphisch veranschaulichen lassen. Unser Ziel ist es dabei, den Wahrheitswert einer logischen Formel, also deren Semantik, visuell zu veranschaulichen. Der Betrachter soll anhand von Beispielinstanzen sehen können, daß die gerade im Beweis manipulierte Aussage wahr (oder falsch) ist.

Der Wahrheitswert einer logischen Aussage wird formal durch ein Modell einer Logik bestimmt. Als Logik betrachten wir im folgenden die zweiwertige Prädikatenlogik; unsere Vorgehensweise ist aber auch auf andere Logiken mit Modellsemantik übertragbar. Der folgende Ansatz ist unabhängig vom verwendeten Kalkül und einem bestimmten Beweisformat. Insbesondere spielen Beweisregeln noch keine Rolle.

Das Modell ist für eine Beweisführung durch den Menschen wichtig. Wird ein Beweis durch den Leser nachvollzogen, so kann dies einerseits durch eine rein syntaktische Überprüfung der Regelanwendungen geschehen oder durch eine Überprüfung auf semantischer Ebene. Gerade letzteres ist für ein tiefes Verständnis des Beweises von grundlegender Bedeutung.

Wir legen eine Vorgehensweise dar, wie zu einer logischen Aussage mit einer Art „bottom-up“ Methode systematisch eine sinnvolle visuelle Darstellung gefunden werden kann. Ziel ist es, die Interpretation einer logischen Formel durch ein Modell mit Hilfe einer visuellen Darstellung (von Teilen) des Modells sichtbar zu machen. Im Gegensatz zum Bereich des diagrammatischen logischen Schließen ist es nicht unser

Ziel, eine graphische Repräsentation des Modells zu konstruieren, die einen visuellen formalen Beweis der zu visualisierende Aussage darstellt. Dies würde nur in Ausnahmefällen für die meist unendliche Modelle gelingen, mit denen wir es in der Theoretischen Informatik zu tun haben.

Wir beschreiben unseren Ansatz formal und wiederholen deswegen kurz die Definition der modellbasierten Semantik einer zweiwertigen Prädikatenlogik.

5.2.1 Modellsemantik der Prädikatenlogik

Wir betrachten im folgenden nur Aussagen, die in einer Sprache der zweiwertige Prädikatenlogik erster Stufe (PL1) formuliert sind. Terme werden wie gewohnt aus gegebenen Variablen-, Konstanten- und Funktionensymbolen; Prädikate aus Termen und einem Prädikatensymbol; und logische Aussagen aus Prädikaten und den üblichen logischen Konnektiven und Quantoren gebildet. Ersetzungen von nicht quantifizierten Variablen durch Terme werden durch Substitutionen beschrieben. Da die formale Syntax solch einer Sprache für unsere weiteren Betrachtungen nur zweitrangig ist, verzichten wir auf eine genaue Darstellung der Syntax; sie kann vom interessierten Leser zum Beispiel in [FITTING 1996, Seiten 97–103] nachgelesen werden.

Auch in der folgenden detaillierteren Darstellung der Semantik von Ausdrücken einer Sprache der zweiwertigen Prädikatenlogik halten wir uns sehr stark an die in [FITTING 1996] gegebenen Definitionen.

Definition 7 (Prädikatenlogisches Modell)
Ein *Modell* $M = \langle D, I \rangle$ für eine Sprache der zweiwertigen Prädikatenlogik erster Stufe besteht aus:

1. Einer nicht leeren Menge D, der *Domäne* von M.
2. Einer Abbildung I, der *Interpretation* von M, die
   - jeder Konstanten c ein Element $c^I \in D$,
   - jedem n-stelligen Funktionssymbol f eine n-stellige Funktion $f^I : D^n \to D$ und
   - jedem n-stelligen Prädikatensymbol P eine n-stellige Relation $P^I \subseteq D^n$ zuordnet.

Definition 8 (Variablenbelegung)
Eine *Variablenbelegung* in einem Modell $\langle D, I \rangle$ ist eine Abbildung β von der Menge der Variablen nach D. Das Bild einer Variablen v unter der Abbildung β bezeichnen wir mit v^β.

Mit Hilfe eines Modells $\langle D, I \rangle$ und einer Variablenbelegung kann jedem Term ein Element aus der Domäne D zugeordnet werden.

Definition 9 (Auswertung eines Terms)
$M = \langle D, I\rangle$ sei ein Modell und β eine Variablenbelegung in M, dann wird jedem Term t ein Wert $t^{I,\beta}$ wie folgt rekursiv zugeordnet:

1. Für jede Konstante c gilt $c^{I,\beta} = c^I$.
2. Für jede Variable v gilt $v^{I,\beta} = v^\beta$.
3. Für ein n-stelliges Funktionssymbol f und Terme $t_1, \ldots, t_n$ gilt

$$[f(t_1, \ldots, t_n)]^{I,\beta} = f^I(t_1^{I,\beta}, \ldots, t_n^{I,\beta}) \ .$$

Beispiel 3 (Natürliche Zahlen)
Die in Beispiel 2 bewiesene Aussage

$$\forall a, b, c, d : \mathbb{N} : a \leq b \wedge c \leq d \Rightarrow a + c \leq b + d$$

wird durch das Modell $\langle \mathbb{N}, I\rangle$ mit $(n + m)^I = n^I + m^I$ und $\leq^I = \{(n, m) \mid n \leq m, n, m \in \mathbb{N}\}$ erfüllt.[2]

Einer Formel aus der gegebenen Sprache wird wie folgt einer der Wahrheitswerte `w` und `f` zugeordnet. Wir gehen davon aus, daß auf Basis von Wahrheitstafeln für die logische Konnektiven, logische Ausdrücke über den Wahrheitswerten `w` und `f` ausgewertet werden können.

Definition 10 (Auswertung einer Formel)
$M = \langle D, I\rangle$ sei ein Modell für eine Sprache der zweiwertigen Prädikatenlogik erster Stufe und β eine Variablenbelegung. Den Formeln Φ, $\neg\Phi$, $\Phi \wedge \Psi$, $\Phi \vee \Psi$, $(\forall x)\Phi$ und $(\exists x)\Phi$ wird wie folgt ein Wahrheitswert $\Phi^{I,\beta}$ (`w` oder `f`) zugeordnet:

1. Für atomare Formeln:

$$[P(t_1, \ldots, t_n)]^{I,\beta} = \texttt{w} \text{ genau dann, wenn } \langle t_1^{I,\beta}, \ldots, t_n^{I,\beta}\rangle \in P^I$$
$$\texttt{true}^{I,\beta} = \texttt{w}$$
$$\texttt{false}^{I,\beta} = \texttt{f}$$

2. $[\neg\Phi]^{I,\beta} = \begin{cases} \texttt{w} & \text{, falls } \Phi^{I,\beta} = \texttt{f} \\ \texttt{f} & \text{, sonst} \end{cases}$
3. $[\Phi \wedge \Psi]^{I,\beta} = \begin{cases} \texttt{w} & \text{, falls } \Phi^{I,\beta} = \texttt{w} \text{ und } \Psi^{I,\beta} = \texttt{w} \\ \texttt{f} & \text{, sonst} \end{cases}$
4. $[\Phi \vee \Psi]^{I,\beta} = \begin{cases} \texttt{w} & \text{, falls } \Phi^{I,\beta} = \texttt{w} \text{ oder } \Psi^{I,\beta} = \texttt{w} \\ \texttt{f} & \text{, sonst} \end{cases}$

[2] Dabei sind $0, 1, 2, \ldots$ lediglich Abkürzungen für deren formale Darstellung $0, s(0), s(s(0))$ mit Hilfe einer Nachfolgefunktion.

5. $[\Phi \Rightarrow \Psi]^{I,\beta} = [\neg\Phi \vee \Psi]^{I,\beta}$

6. $[(\forall x)\Phi]^{I,\beta} = \begin{cases} \mathsf{w} & \text{, falls für jedes } \beta' \text{ mit } \beta(y) = \beta'(y) \text{ für jedes } y \neq x \\ & \Phi^{I,\beta'} = \mathsf{w} \text{ gilt} \\ \mathsf{f} & \text{, sonst} \end{cases}$

7. $[(\exists x)\Phi]^{I,\beta} = \begin{cases} \mathsf{w} & \text{, falls für ein } \beta' \text{ mit } \beta(y) = \beta'(y) \text{ für jedes } y \neq x \\ & \Phi^{I,\beta'} = \mathsf{w} \text{ gilt} \\ \mathsf{f} & \text{, sonst} \end{cases}$

Beispiel 4 (Hyperproof)
In HYPERPROOF enthält die Domäne des dort fixierten Modells geometrischen Objekte. Die Interpretation I bildet Konstanten ($a, b, c, \ldots$) auf die entsprechenden geometrischen Objekte ab. Funktionssymbole gibt es nicht. Die Prädikatensymbole werden von I auf Relationen abgebildet, deren Wahrheitswert von der Lage und Größe der geometrischen Objekte abhängt.

Betrachte Abbildung 4.4, Seite 77. Das große Dodekaeder wird mit d und das kleine mit d bezeichnet (siehe Beweistext). Für das gegebene Modell gilt also $d \in Large^I$ und $Dodec^I = \{c, d\}$, und damit $Large(d)^I = w$ und $Large(c)^I = f$.

5.2.2 Modellbasierte Visualisierung logischer Formeln

Ein von einem Menschen umgangssprachlich formulierter Beweis enthält üblicherweise abstrakte Schlußfolgerungen, die auf den semantischen Eigenschaften der im Beweis auftretenden Objekte basieren. Rein syntaktische Schlußfolgerungen mit einem formalen Beweiskalkül kommen eher selten vor. Im Gegensatz zu einer Maschine führt ein Mensch den Beweis einer Aussage meist mit einem bestimmten Modell vor Augen. Um umgekehrt einen Leser die Schlußfolgerungen des Beweises zu vermitteln, ist es deswegen sinnvoll die im Beweis auftretenden logischen Aussagen und Argumentationen auf Basis dieses Modells zu veranschaulichen. Als ersten Schritt zur Erstellung von Visualisierungen logischer Aussagen fixieren wir deshalb ein Modell der bewiesenen Aussage. Wir erläutern dann, wie auf Basis solch eines Modells die logischen Aussagen systematisch visualisiert werden können. Dabei konstruieren wir ausgehend vom fixierten Modell schrittweise für die Konstantensymbole, Terme, Prädikate und letztendlich für die Formeln eine visuelle Darstellung, bei der jeweils die inm vorangehenden Schritt festgelegten graphischen Darstellungen für den nächsten Schritt übernommen werden. Als Resultat dieser hierarchischen Vorgehensweise „enthält“ die Visualisierung einer Formeln (eines Terms) die Visualisierung von Teilformeln (Teiltermen). Wie wir noch sehen werden, ist diese Eigenschaft wichtig für die Animation der Beweisregeln. In der Praxis wird man natürlich nicht derart formal wie im folgenden vorgehen, sondern lediglich informell die interessierenden Teile des Modells fixieren. Wie die Objekte des Beweises dann konkret dargestellt werden sollen, hängt stark vom Einzelfall ab. Wir stellen unser Konzept deswegen an Beispielen orientiert

vor. Unsere Methode dient als eine Art Leitfaden zur Konstruktion visueller Darstellungen der Interpretation logischer Aussagen.

Visualisierung der Domäne

Der erste Schritt, die in einem Beweis auftretenden logischen Formeln visuell darzustellen, ist festzulegen, wie die Elemente der fixierten Domäne D graphisch veranschaulicht werden sollen.

Definition 11 (Visualisierung einer Domäne)
$\langle D, I \rangle$ sei ein prädikatenlogisches Modell. Eine Abbildung V, die jedem $d \in D$ eine graphische Darstellung d^V zuordnet, heißt *Visualisierung von* D.

Eine sinnvolle Wahl von V hängt von dem im Beweis verwendeten Eigenschaften der Elemente von D ab. Die Visualisierung der Domäne eines Modells sollte die wesentlichen Eigenschaften der Elemente aus der Domäne visuell wiedergeben. „Wesentlich“ heißt, daß die in einem Beweis verwendeten Eigenschaften von und Beziehungen zwischen den Elementen visuell vorliegen, aber die zu beweisenden Eigenschaften nicht visualisiert werden.

Wir unterscheiden zwei grundlegende Typen der Visualisierung einer Domäne, die wir als *konkrete* und *abstrakte* Visualisierung bezeichnen:

- *Konkrete Visualisierung:* Die graphischen Darstellungen der Elemente der Domäne sind verschieden. Die Darstellung gibt genau die individuellen Eigenschaften eines Elements wieder.

- *Abstrakte Visualisierung:* Die graphischen Darstellungen der Elemente der Domäne sind nicht verschieden. Viele Elemente (insbesondere unendliche viele) besitzen dieselbe graphische Darstellung. Die Darstellung gibt einige allen Elementen gemeinsame Eigenschaften wieder.

Als Beispiele betrachten wir im folgenden auch die in Abschnitt 2.2 vorgestellten Systeme und interpretieren deren visuelle Darstellungen unter dem Aspekt obiger Begriffe.

Beispiel 5 (Gloors System)
Gloors System wurde zur Animation eines Korrektheitsbeweises eines Graphenalgorithmus benutzt. Die Domäne D des Modells des bewiesenen Theorems enthält (bezeichnete) Knoten, (gewichtete) Kanten, natürliche Zahlen (zur Gewichtung der Kanten) und Graphen.

Ein Graph wird auf Basis seiner Kanten und Knoten mit einer in der Informatik üblichen Form graphisch dargestellt: Jeder Knoten wird durch einen Kreis visuell mit alphanumerischen Bezeichner dargestellt, jede Kante durch eine Gerade, welche die

beiden zugehörigen Knoten verbindet, jedes Gewicht wird als Folge arabischer Ziffern an die zugehörigen Kante geschrieben. Ein aus Knoten und gewichteten Kanten bestehender Graph wird auf analoge Weise visualisiert.

Jede Kante, jeder Knoten, jede Zahl und jeder Graph der Domäne D hat eine eigene eindeutige graphische Darstellung, das heißt, in unserer Terminologie verwendet Gloors System eine konkrete Visualisierung von D.

Beispiel 6 (Satz des Pythagoras)
Die Domäne eines natürlichen Modells des Satzes von Pythagoras, enthält rechtwinklige Dreiecke. Diese können auf vielfältige Art und Weise mathematisch modelliert werden: Als Menge dreier Punkte im euklidischen Raum oder als Tripel der Kantenlänge der Hypothenuse und zweier Winkel. Die Details interessieren uns hier nicht.

Die ZEUS-Animation verwendet eine abstrakte Visualisierung eines rechtwinkligen Dreiecks. Jedes Dreieck der Domäne D wird durch eine identische graphische Darstellung visualisiert, die von den konkreten Maßen und Winkeln der Dreiecke abstrahiert. In unserer Terminologie verwendet die ZEUS-Animation eine abstrakte Visualisierung von D.

Beispiel 7 (Hyperproof)
In HYPERPROOF besteht die Domäne aus drei-dimensionalen geometrischen Objekten (Pyramiden, Quader, Dodekaeder).

Sie werden hauptsächlich konkret als zwei-dimensionale Projektionen graphisch dargestellt. Nur für manche Lernsituationen wird bei einem Objekt die Form unbestimmt gelassen: Der Benutzer soll als Übung aus vorgegebenen Aussagen Form und Größe deduktiv bestimmen. Zu sehen ist dann ein Zylinder als abstrakte Darstellung des Objekts.

Konkrete Visualisierungen entsprechen in der Programmvisualisierung in etwa der graphischen Darstellung von Datenstrukturen. Ein konkrete Darstellung der Elemente einer Domäne ist meist einfacher zu finden, als eine stark abstrahierte. Abstrakte Darstellungen reflektieren weniger Eigenschaften als konkrete Darstellungen und können deswegen irreführend sein, wenn eine für einen Beweis wichtige Eigenschaft nicht dargestellt wird. Da abstrakte Darstellungen aber weniger Details zeigen, sind sie für den Betrachter meist kognitiv leichter zu erfassen. Siehe Tabelle 5.1 für eine Gegenüberstellung dieser Vor- und Nachteile. Für die Praxis ist es sinnvoll, erst eine konkrete Darstellung zu wählen, und erst später zu versuchen, von unwichtigen Details in der Visualisierung zu abstrahieren (etwa den inneren Knoten von Bäumen und Graphen oder dem konkreten Winkel eines Dreiecks).

Beispiel 8 (Natürliche Zahlen)
Wir betrachten noch einmal das in Beispiel 3 fixierte Modell $\langle \mathbb{N}, I \rangle$. Der Beweis nutzt die relative Größe einer natürlichen Zahl als ihr wesentliches Merkmal aus. Diese läßt sich auf vielfältige Weise visualisieren, zum Beispiel:

konkret	abstrakt
leichter zu finden	schwierig zu finden
allgemein für verschiedene Beweis zu verwenden	Abstraktion meist speziell für einen Beweis
meist detailreiche Darstellung	oft kompakte Darstellung
kognitiv schwerer zu erfassen	kognitiv leichter zu erfassen
meist alle Eigenschaften dargestellt	wichtige Eigenschaften könnten fehlen

Tabelle 5.1: Vor- und Nachteile konkreter und abstrakter Visualisierungen

- Durch einen (horizontalen) Balken, dessen Länge proportional zur Größe der dargestellten Zahl ist.
- Durch einen Kreis, dessen Durchmesser proportional zur Größe der dargestellten Zahl ist.

Welche der beiden Varianten – Darstellung natürlicher Zahlen als Balken oder Kreise – die sinnvollere ist, hängt neben der Domäne D auch von der fixierten Interpretation I ab. Letztere ordnet den in einem Beweis auftretenden syntaktischen Konstrukten, Konstanten und Termen Elemente aus D und Prädikaten Wahrheitswerten zu. Wir erweitern nun die Visualisierung von D zu einer Visualisierung von Konstanten und Termen. Dabei gehen wir davon aus, daß die Elemente von D von V für einen vorliegenden Beweis sinnvoll graphisch dargestellt werden (also alle für den Beweis wesentlichen Eigenschaften reflektieren).

Visualisierung von Konstanten

Die im Beweis auftretenden Konstantensymbole c werden sinnvollerweise durch $(c^I)^V$ visualisiert, das heißt, eine Konstante c wird durch die Visualisierung ihres durch die Interpretation zugeordneten Elements c visualisiert.

Definition 12 (Visualisierung von Konstanten)
$\langle D, I\rangle$ sei ein prädikatenlogisches Modell und V eine Visualisierung von D. V läßt sich vermittels $(c^I)^V$ zu einer *Visualisierung von Konstanten* c fortsetzen.

Beispiel 9 (Hyperproof)
In Hyperproof existieren mit kleinen alphanumerischen Symbolen bezeichnete Konstanten ($a, b, c, \ldots$). Sie werden als geometrische Objekte des fixierten Modells interpretiert und mit Hilfe deren zwei-dimensionale Projektionen visualisiert. Zur besseren Identifizierung einer zu einer Konstanten gehörigen graphischen Darstellung, werden die Konstanten teilweise zur zusätzlichen Kennzeichnung verwendet.

Die visuelle Darstellung von Konstantensymbolen entspricht in der Programmvisualisierung in etwa der visuellen Darstellung von Werten elementarer Datentypen, wie ganzen Zahlen oder Zeichenketten.

Visualisierung von Termen

Die einfachste Variante, Terme auf Basis der vorangehenden Visualisierungen zu veranschaulichen, ist analog zu Konstanten eine Fortsetzung von V vermittels $t^V = (t^I)^V$. Allerdings geht dabei in einige Fällen wichtige Information über den Termaufbau verloren. Wir diskutieren dies anhand zweier Beispiele etwas genauer.

Beispiel 10 (Binärbäume)
In einem Beweis der Korrektheit eines Algorithmus, der Binärbäume über natürlichen Zahlen manipuliert, seien Binärbäume syntaktisch wie folgt beschrieben:

- nil ist der leere Binärbaum
- Wenn l und r Binärbäume sind, dann ist auch $b(l, r)$ ein Binärbaum.

Das natürliche Modell $\langle D, I\rangle$ der Korrektheitsaussage enthält eine mathematische Beschreibung von Binärbäume. Das zweistellige Funktionssymbol b hat bei einem Term $b(l, r)$ einen rein *kompositionalen* Charakter: Es fügt die Binärbäume l und r zu einem neuen Binärbaum mit einem (nicht näher bestimmten) Wurzelknoten und linken Teilbaum l sowie rechten Teilbaum r zusammen. Im zugehörigen Modell sind die beiden mathematischen Modellierungen l^I und r^I der Teilbäume l und r integraler Bestandteil von $b(l, r)^I$.

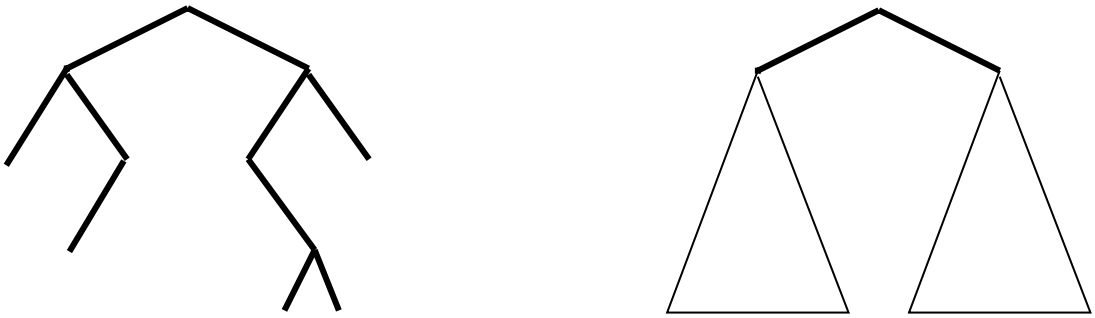
Diese Elemente von D können durch eine gewohnte graphische Darstellung von Binärbäumen konkret visualisiert werden, die graphischen Darstellungen von l und r sind dann Teil der graphischen Darstellung von $b(l, r)$. Sei V solch eine Visualisierung von D. Obiger Term läßt sich analog zu einer Konstanten ohne Informationsverlust vermittels $(b(l, r)^I)^V$ visualisieren. Bei einer abstrakten Visualisierung kann auch von den konkreten Termen l und r abstrahiert werden. Siehe Abbildung 5.1 für solche zwei unterschiedliche Darstellungen von

$$b(b(b(nil, nil), b(b(nil, nil), nil)), b(b(nil, b(b(nil, nil), b(nil, nil))), b(nil, nil))) \quad .$$

Bei Funktionssymbolen mit kompositionalem Charakter sind die so konstruierten Objekte oftmals eindeutig gegeben, das heißt, ein Element von D hat in der logischen Sprache eine eindeutige syntaktische Darstellung als Term. Anders verhält es sich im folgenden Beispiel.

Beispiel 11 (Natürliche Zahlen)
Die Addition $+$ auf den natürlichen Zahlen besitzt einen *operationalen* Charakter. Sie weist zwei natürlichen Zahlen eine neue Zahl zu, ohne daß die beiden Argumente Teil der gebildeten Summe sind. Darüberhinaus ist die Darstellung nicht eindeutig: $1 + 3$ bezeichnet dieselbe Zahl wie $2 + 2$.

Stellt eine Visualisierung V von $\mathbb{N}$ die natürlichen Zahlen als horizontale Balken dar, so sind die Darstellungen von $((1 + 3)^I)^V$ und $((2 + 2)^I)^V$ identisch. Die Information über den Termaufbau geht dabei verloren. Abhilfe kann hier eine Animation



(a) Konkrete Visualisierung

(b) Abstrakte Visualisierung

Abbildung 5.1: Beispiel einer konkreten (a) und abstrakte (b) Visualisierung eines Binärbaums

schaffen, welche die Konstruktion von $(1+3)^I$ veranschaulicht und in der visuellen Darstellung $((1+3)^I)^V$ endet. Abbildung 5.2 skizziert solch eine Animation, bei der die zu den Argumenten gehörigen Balken zu einem neuen Balken vereinigt werden, der deren Summe visualisiert.

Werden natürliche Zahlen durch Kreise mit einem Durchmesser proportional zum Wert der Zahl dargestellt, so läßt sich die Addition nicht mehr derart intuitiv visualisieren. Die Balkendarstellung ist in diesem Fall sinnvoller.

(a) Animation von $1+3$

(b) Animation von $2+2$

Abbildung 5.2: Animation der Summe $1+3$ (a) und $2+2$ (b)

Falls ein Term $f(t_1,\ldots,t_n)$, der mehrere Teilterme $t_1,\ldots,t_n$ mit operationalem Charakter enthält, durch eine Animation visualisiert werden soll, ist es sinnvoll, zuvor die Teilterme in einer bestimmten Reihenfolge zu animieren, um dann die jeweils resultierenden statischen Abbildungen als Ausgangspunkt der Animation von f zu benutzen. Eine simultane Animation der Teilterme ist zu vermeiden, da dann der kognitive Aufwand für den Betrachter zu groß wird. Man kann als einfache Auswertungsstrategie den am weitesten links stehenden Teilterm zuerst (rekursiv) animieren, dann den davon rechts stehenden, usw.

Damit diese hierarchische Form der Visualisierung von Teiltermen systematisch durchgeführt werden kann, sollte bei der Animation eines Terms t immer das letzte Bild mit der Visualisierung $(t^I)^V$ übereinstimmen. Ist diese wichtige Eigenschaft erfüllt, dann kann der Betrachter die Interpretation des Terms Schritt für Schritt anhand der Animation visuell nachvollziehen.

Beispiel 12 (Natürliche Zahlen)
Wir visualisieren, wie in Beispiel 11, natürliche Zahlen als horizontale Balken und die Addition zweier Zahlen als Aneinanderfügen von deren zugehörigen Balken. Dann endet die Animation $1 + 1$ in einer Visualisierung der resultierenden natürlichen Zahl 2, und die Summe $((1 + 1) + 1) + 1$ kann dann systematisch durch eine Sequenz der drei Animationen $1 + 1$, $2 + 1$ und $3 + 1$ visualisiert werden. Siehe Abbildung 5.2 für eine Skizze dieser Animation.

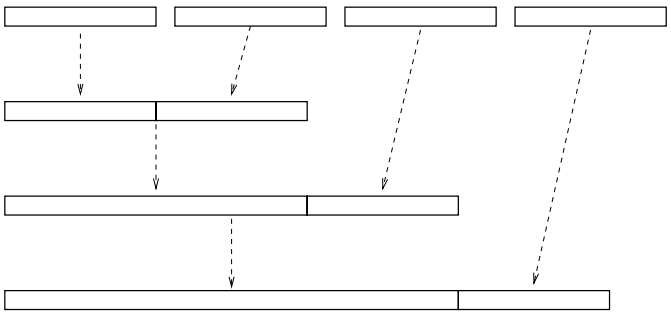


Abbildung 5.3: Animation der Summe $((1 + 1) + 1) + 1$

Ein Term, der aus verschiedenen Funktionssymbole aufgebaut ist, kann natürlich beide Formen – kompositional und operational – enthalten und wie oben beschrieben visualisiert werden (eine statische Visualisierung läßt sich dabei als eine sehr simple Animation auffassen). In manchen Fällen ist es sinnvoller, einen Term t mit operationalem Charakter nur einmal durch eine Animation zu visualisieren, und in weiteren Beweisschritten, in denen t vorkommt, lediglich das Resultat der Animation zu verwenden. Dies reduziert den kognitiven Aufwand des Betrachters.

Die visuelle Darstellung kompositionaler Terme entspricht in der Programmvisualisierung grob der visuellen Darstellung zusammengesetzter Datenstrukturen; und die von operationalen Termen entspricht der visuellen Darstellung eines (arithmetischen) Ausdrucks.

Visualisierung von Prädikaten

Wenn die Visualisierung V von Termen festgelegt ist, so muß als nächstes überlegt werden, wie auf Basis von V die Prädikate der logischen Sprache geeignet veranschaulicht werden können. Prädikate treffen Aussagen über die Eigenschaften von

oder Beziehungen zwischen Elementen der Domäne. Eine Visualisierung sollte diese Beziehung für den menschlichen Betrachter sichtbar machen.

Bei Prädikaten, die individuelle Eigenschaften der Elemente der Domäne beschreiben, sollten diese Eigenschaften schon bei der Visualisierung der Elemente mit veranschaulicht werden. Eine explizite Visualisierung des Prädikatensymbols entfällt dann.

Beispiel 13 (Hyperproof)
In HYPERPROOF existieren Prädikatensymbole, um die Form und absolute oder relative Größe der geometrischen Objekte syntaktisch zu beschreiben. Diese Eigenschaften werden implizit durch die graphische Darstellung der Elemente als Tetraeder, Quader oder Dodekaeder mit genau sichtbarer Größe visualisiert.

In nicht formalen Beweisen kommen Gleichheitsrelation, Ungleichheit und Enthaltenseinbeziehungen sehr häufig vor. Prädikate, die Elemente derart vergleichen, können meist durch einfaches Gegenüberstellen der visuellen Darstellungen der verglichenen Elemente visualisiert werden.

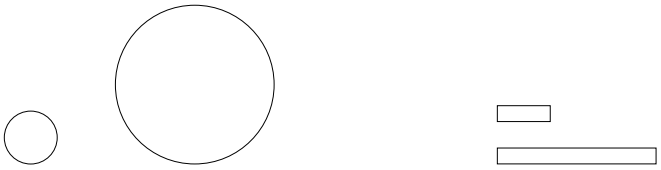
Beispiel 14 (Natürliche Zahlen)
Der Beweis aus Beispiel 2 enthält logische Aussagen, in denen natürliche Zahlen größenmäßig verglichen werden, etwa $n \leq m$. Werden, wie in Beispiel 8 vorgeschlagen, natürliche Zahlen als horizontale Balken bzw. Kreise mit einer Länge oder einem Durchmesser proportional zum Wert der Zahl visualisiert, so kann das Prädikat $\leq$ durch Gegenüberstellen der Balken oder Kreisen visualisiert werden. Die graphischen Objekte sollten so plaziert werden, daß der menschliche Betrachter den Größenunterschied leicht sieht – etwa durch Übereinanderstellen der beiden Balken und gleicher Ausrichtung an deren linken Positionen; siehe Abbildung 5.4. Welche der beiden Darstellungen sinnvoller ist, hängt davon ab, welche weiteren Eigenschaften natürlicher Zahlen im Beweise visualisiert werden sollen.

Man beachte, daß auf dieselbe Weise auch die Relationen $<$, $\geq$, $>$ und $=$ dargestellt werden können.

Enthaltenseinrelationen, wie die Mengenrelationen $\supset$, $\supseteq$, $\subset$ und $\subseteq$, lassen sich – wie bei Venn-Diagrammen üblich – durch geometrisches Enthaltensein einer Visualisierung in einer anderen visualisieren. Dies trifft auch für das Enthaltensein von Teilbäumen in Bäumen, partiellen Ableitungen in Ableitungsfolgen usw. zu. Zu bemerken ist, daß bei einer guten Visualisierung einer Relation oftmals auch deren Umkehrung mit visualisiert wird, so wie bei $\leq$ und $\subset$.

Visualisierung von Formeln

Logische Formeln setzen sich aus atomaren Formeln und (rekursiv) aus Formeln mit Hilfe logischer Konnektive und Quantoren zusammen. Die Semantik dieser Formeln hängt von der verwendeten Logik ab. Wir beschränken uns hier auf die gebräuchlichen Konnektive $\neg$ (Negation), $\wedge$ (Konjunktion), $\vee$ (Disjunktion), $\Rightarrow$ (Implikation)



(a) Kreise (b) Balken

Abbildung 5.4: Beispiel einer Visualisierung von $1 \leq 3$, einmal durch Kreise und einmal durch horizontale Balken. Der Durchmesser der Kreise ist identisch zur Länge der horizontalen Balken. Die Größe läßt sich in beiden Fällen gut vergleichen. Der Platzbedarf ist bei Balken allerdings erheblich geringer.

und $\Leftrightarrow$ (Äquivalenz) sowie die Quantoren $\forall$ (Allquantor) und $\exists$ (Existenzquantor) mit der bekannten zweiwertigen Semantik. Im folgenden können wir kein allgemeinverbindliches Rezept bieten, um beliebige Formeln zu visualisieren. Wir beschränken uns wieder auf eine beispielorientierte Darstellung. Die Erfahrungen mit unseren Beispielanimationen zeigen, daß die logischen Aussagen in den nicht formalen Beweisen fast keine logischen Konnektive oder Quantoren enthalten (so wie bei Gleichheitsbeweisen). Die wenigen Konnektive lassen sich meist wie in den im folgenden vorgestellten Beispielen visualisieren.

Visualisierung logischer Konnektive Eine Konjunktion $\Phi \wedge \Psi$ ist wahr, wenn Φ *und* Ψ wahr ist. Dies läßt sich am sinnvollsten visualisieren, wenn der Betrachter die Visualisierung von Φ *und* von Ψ sehen kann. Eine Konjunktion sollte deswegen durch eine simultane Visualisierung der beiden Teilformeln dargestellt werden, etwa durch Nebeneinanderstellen oder einer Darstellung in zwei verschiedenen Fenstern, siehe Abbildung 5.5 für ein Beispiel.

Abbildung 5.5: Beispiel einer Visualisierung von $1 \leq 3 \wedge 2 \leq 4$.

Eine Disjunktion $\Phi \vee \Psi$ ist wahr, wenn Φ *oder* Ψ wahr sind. Dies läßt sich am sinnvollsten visualisieren, wenn der Betrachter die Visualisierung von Φ *oder* von Ψ sehen kann, je nachdem, welche der beiden Formeln vom Modell erfüllt wird. Auch dies läßt sich wie bei der Konjunktion durch eine simultane Visualisierung der erfüllten Teilformeln erreichen. Werden konjunktiv verknüpfte Formeln vertikal und disjunktiv verknüpfte Formeln horizontal ausgerichtet, so lassen sich beide Konnektive optisch unterscheiden. Diese Form der Darstellung findet sich beim automatischen

Beweisen zur Darstellung von Formeln in Negationsnormalform beim Dissolutionskalkül [MURRAY und ROSENTHAL 1987] bzw. für Formeln in konjunktiver Normalform bei der Matrixmethode [BIBEL 1987], dort werden die Teilformeln allerdings nur syntaktisch dargestellt.

Die logische Negation braucht bei atomaren Formeln normalerweise nicht explizit visualisiert zu werden, da bei guter Visualisierung des Prädikats auch dessen Negation sichtbar wird, so wie das in unseren Beispielen der Fall ist.

Wenn Konjunktion, Disjunktion und Negation so wie erörtert visualisiert werden, kann aus der Visualisierung selbst die logische Aussage nicht mehr rekonstruiert werden: Abbildung 5.5 könnte auch eine Visualisierung von $1 \leq 3 \wedge 2 \leq 4$ oder $\neg(1 \leq 3) \vee \neg(2 \leq 4)$ zeigen. Im Bereich des logischen Schließens mit Diagrammen wäre dies fatal, da die Visualisierung den Beweistext ersetzt und in unserem Beispiel die Visualisierung nicht mehr als eindeutiger Beweis einer bestimmten Aussage gelten kann. Weil wir die Visualisierung lediglich begleitend zum nach wie vor existierenden Beweistext verwenden, haben wir diese Probleme allerdings nicht.

Da in der zweiwertigen Prädikatenlogik alle anderen logischen Konnektive mit Hilfe der Konjunktion, Disjunktion und Negation dargestellt werden können, lassen sie sich auch auf analoge Weise visualisieren. In vielen Fällen besitzt die Implikation allerdings einen eher konstruktiven Charakter, der für das Verständnis der Aussage wichtiger ist als das rein logische Äquivalent $\neg\Phi \vee \Psi$ von $\Phi \Rightarrow \Psi$. Falls eine Implikation eher den Charakter einer Konstruktionsvorschrift besitzt, so kann es sinnvoll sein, zu visualisieren, wie aus dem Wahrheitswert von Φ der Wahrheitswert von Ψ konstruiert wird. Dies läßt sich, wie bei Funktionen mit operationalem Charakter, mit Hilfe einer Animation bewerkstelligen. Ausgangspunkt der Animation ist die Visualisierung von Φm und Endpunkt ist die Visualisierung von Ψ.

Beispiel 15 (Natürliche Zahlen)
Wir betrachten die Formel $1 \leq 2 \Rightarrow 1 \leq 2+1$. Die Implikation läßt sich als Konstruktion der Konklusion $1 \leq 2 + 1$ aus der Prämisse $1 \leq 2$ interpretieren.

Dies läßt sich auf Basis der wie in den vorangehenden Beispielen visualisierten Zahlen, dem Additionsoperator und dem Prädikat $\leq$ durch eine Animation visualisieren, die aus der Visualisierung von $1 \leq 2$ durch Addition von 1 auf der rechten Seite der Ungleichung die Visualisierung von $1 \leq 2 + 1$ konstruiert, siehe Abbildung 5.6.

Visualisierung von Quantoren Bei den bisherigen Betrachtungen spielten Variablen – ob gebunden oder ungebunden – noch keine Rolle: es handelte sich um Grundterme und Grundformeln.

Eine existenzielle Aussage $\exists x \in M : \Phi$ ist genau dann wahr, wenn es ein $d \in M$ gibt, so daß $\Phi[x/d]$ wahr ist. Dieser Interpretationsvorgang kann durch Angabe eines solchen Elements d dem menschlichen Betrachters veranschaulicht werden. $\exists x \in M : \Phi$ wird also wie die Formeln $\Phi[x/d]$ visualisiert. Der Bezug zum Quantor wird (wie bei den logischen Konnektiven) durch die textuelle Darstellung hergestellt. Die Wahl

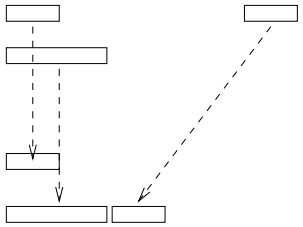


Abbildung 5.6: Beispiel einer Visualisierung von $1 \leq 2 \Rightarrow 1 \leq 2 + 1$ durch eine Animation.

von d hängt von dessen weiterer Verwendung im Beweis ab. Es gibt unter Umständen unendlich viele Wahlmöglichkeiten.

Beispiel 16 (Natürliche Zahlen)
Betrachte die Formel $\exists n : \mathbb{N} : 2 \leq 1 + n$. Sie ist wahr für jedes $n \geq 1$. Die Formeln kann etwa durch eine Visualisierung von $2 \leq 1 + 3$ veranschaulicht werden.

Eine universell quantifizierte Formel $\forall x \in M : \Phi$ ist genau dann wahr, wenn für jedes $d \in M$ die Formel $\Phi[x/d]$ wahr ist. Analog zu existenziellen Quantifizierungen könnte ein menschlicher Betrachter diese Interpretation visuell nachvollziehen, indem er für jedes $d \in M$ die Visualisierung von $\Phi[x/d]$ betrachtet. Dies wäre allerdings bei einem unendlichen Grundbereich M eine sehr zeitraubende Tätigkeit und ist daher nur bei endlichen Mengen M praktikabel. Bei Mengen mit unendlich vielen Elementen bieten sich folgende Lösungen an:

- Wähle eine abstrakte Visualisierung der Elemente von M, so daß sich $\forall x \in M : \Phi$ durch eine endliche Aufzählung geeigneter Repräsentanten $\bar{d}$ von M vermittels $\Phi[x/\bar{d}]$ visualisieren läßt. Am günstigsten ist es, alle Elemente von M durch eine einzige Darstellung zu visualisieren. Allerdings gelingt dies nur in Ausnahmefällen.

- Wähle ein Element $d \in M$ aus, und visualisiere $\forall x \in M : \Phi$ vermittels der (konkreten) Visualisierung von $\Phi[x/d]$. Diese Lösung ist in der Regel immer möglich, da die so resultierende Grundformel eine Aussage über endlich viele Elemente trifft. Bei einer Implementierung der Visualisierung auf einem Rechner kann d etwa durch den Betrachter der Visualisierung selbst gewählt werden, so wie bei einer Algorithmenanimation die Eingabe des Algorithmus oft vom Betrachter ausgewählt werden kann.

Beispiel 17 (Natürliche Zahlen)
Betrachte die Aussage $\forall n \in \mathbb{N} : n \leq n + 1$ Wir nehmen an, daß natürliche Zahlen durch horizontale Balken, die Addition zweier Zahlen durch Zusammenlegen ihrer visuellen Darstellungen und die Ungleichung durch Übereinanderstellen visualisiert werden.

Die Aussage kann dann durch eine Iteration über alle natürlichen Zahlen visualisiert werden. Dies resultiert in der (unendlichen) Folge der Visualisierungen von $1 \leq 1+1$, $2 \leq 2+1$, $3 \leq 3+1$, etc. Dies ist offenbar nicht praktikabel. Statt dessen kann ein geeigneter Repräsentant für n gewählt werden, etwa 2, und die Aussage wird durch die Visualisierung von $2 \leq 2+1$ veranschaulicht.

Bei einer konkreten Visualisierung führt die Iteration zu einer komplexen animierten Darstellung der logischen Aussage. Bei einer Folge verschiedener allquantifizierter Formeln in einem (linearen) Beweis, ist der Zusammenhang zwischen den visuellen Darstellungen der einzelnen Formel für den Betrachter nur noch sehr schwer nachvollziehbar. Deswegen sollten Quantifizierungen möglichst früh im Beweis durch freie Variablen ersetzt werden. Dann kann anstatt über jede einzelne Formel, über den quantifizierten Teil des Beweises iteriert werden. Für den Betrachter hat dies den Vorteil, den quantifizierten Teil schrittweise an einer Instanz der Variablen nachvollziehen zu können.

5.2.3 Zusammenfassung

Wir fassen noch einmal schematisch zusammen, wie zu logischen Aussagen auf Basis eines Modells visuelle Darstellungen konstruiert werden sollten.

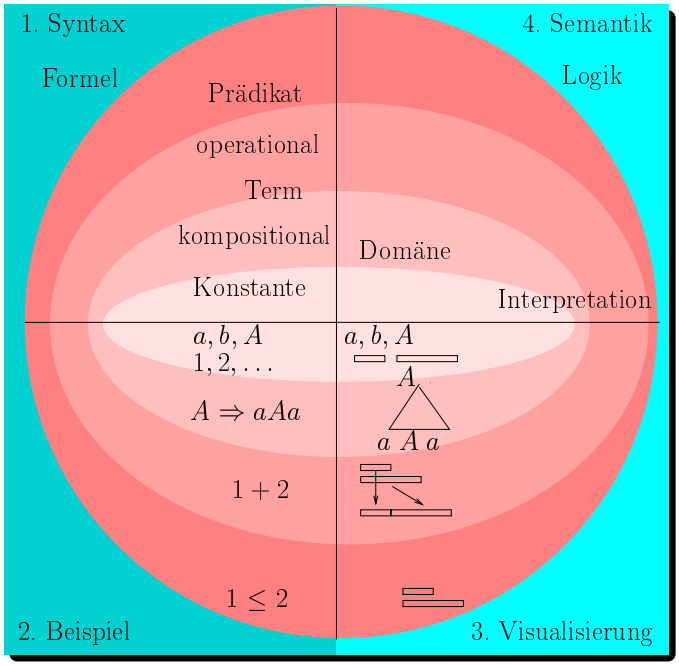


Abbildung 5.7: Schematische Vorgehensweise bei der Visualisierung von Formeln

Abbildung 5.7 zeigt konzentrische Ellipsen, die in vier Quartale unterteilt sind; die

linken beiden Viertel beschreiben die Syntaxebene; die rechten beiden die semantische Ebene. Die Viertel bedeuten im einzelnen (gegen den Uhrzeigersinn von links, oben):

1. Die syntaktische Beschreibung logischer Aussagen durch Konstanten-, Funktions-, Prädikatensymbolen und die logische Sprache.

2. Beispiele von Termen und atomaren Formeln.

3. Visuelle Darstellungen der Beispiele auf Basis einer Visualisierung eines Modells.

4. Die Semantik logischer Aussagen, gegeben durch ein prädikatenlogisches Modell bestehend aus einer Domäne und einer Interpretation.

Die zu visualisierenden logischen Aussagen sind im Beweistext enthalten. Die bewiesene Aussage wird durch das Modell erfüllt. Um systematisch zu den logischen Aussagen geeignete visuelle Darstellungen zu finden, wird von „innen“ nach „außen“ vorgegangen: Zuerst werden die visuellen Darstellungen der Konstanten festgelegt (der elementaren Objekte der Domäne); diese werden verwendet, um die Darstellungen von Termen mit kompositionalem Charakter festzulegen (komplexe, zusammengesetzte Objekte der Domäne); diese werden verwendet, um die restlichen Termen mit überwiegend operationalem Charakter festzulegen; die visuellen Darstellungen der Terme werden verwendet, um die durch Prädikatensymbole beschriebenen Beziehungen zwischen den Termen zu visualisieren. Als Effekt dieser Vorgehensweise „enthalten“ die Visualisierungen von Formeln und Termen die Visualisierungen ihrer Teilformeln und Teilterme. Die konkrete Bedeutung von „enthalten“ hängt von der Wahl der visuellen Darstellung ab, gemeint ist aber durchaus ein weitgehend geometrisches Enthaltensein. Die so konstruierte visuelle Darstellung einer logischen Aussage kann auch als deren visuelle „Interpretation“ angesehen werden. Resultat dieser „Interpretation“ ist ein Diagramm, welches den Wahrheitswert der logischen Aussage sichtbar macht.

Im Gegensatz zu den Arbeiten im Fachgebiet „logisches Schließen mit Diagrammen“, bei denen fast ausschließlich eine isomorphe Beziehung zwischen zu beweisender Aussage und diagrammatischer Darstellung eines Modells im Vordergrund steht, begnügen wir uns mit einer injektiven Abbildung von logischen Aussagen auf visuelle Darstellungen von Teilen eines Modells. Unser Ansatz hat den Vorteil, auch Modelle, die keine (anschauliche) endliche diagrammatische Darstellung besitzen, durch konkrete Visualisierungen ihrer Elemente zu veranschaulichen. Allerdings stellt die Visualisierung dann kein Modell (also Beweis) der Aussage mehr dar. Da wir den Beweistext aber nicht durch diese Darstellungen ersetzen, stellt dies für unsere Methode kein Problem dar.

5.2.4 Animation von Beweisregelanwendungen

Wir kommen nun zur Visualisierung der Beweisregelanwendungen in einem Beweis. Eine Beweisregel im *Calculational proof format* ist eine syntaktische Transformati-

on einer Booleschen Struktur Φ in eine neue Boolesche Struktur Ψ. Ein Beweis ist dann eine Folge solcher Transformationsschritte. Die Regelanwendung kann als syntaktische Substitution eines Teilterms (Teilstruktur) s in Φ durch einen Term (eine Struktur) t beschrieben werden. Wir schreiben $\Phi[s/t]$ für solch eine Substitution. Es gilt also $\Psi = \Phi[s/t]$. Natürlich kann eine Regelanwendung Φ auch ganz ersetzen.

Die wichtigste Eigenschaft einer Beweisregel ist dessen Korrektheit. Im *Calculational proof format* bedeutet dies, daß die Regel den Booleschen Wahrheitswert der transformierten Booleschen Struktur erhält: Eine wahre (falsche) Aussage wird wieder in eine wahre (falsche) Aussage transformiert.

Im vorangegangene Abschnitt haben wir gezeigt, wie auf Basis eines Modells der bewiesenen Aussage der Wahrheitswert einer logischen Aussage, also hier einer Booleschen Struktur, visualisiert werden kann. Um eine Regelanwendung in einem Beweis anschaulich nachvollziehen und verstehen zu können, muß eine Visualisierung der Regelanwendung deutlich machen, *wie* die visuelle Darstellung des Wahrheitswerts der transformierten Struktur Φ in die visuelle Darstellung des Wahrheitswerts der resultierenden Struktur Ψ transformiert wird. Dies ist ein konstruktiver Prozeß, der am besten durch eine Animation verdeutlicht wird. Ausgangspunkt der Animation ist die Visualisierung von Φ und Endpunkt der Animation ist die Visualisierung von Ψ. Weitere Visualisierungen von Beweisregelanwendungen können so leicht hintereinander ausgeführt werden. Der Betrachter kann die Korrektheit der Regelanwendungen anhand der modellbasierten visuellen Darstellungen der Booleschen Strukturen anschaulich nachvollziehen.

Zu einer sinnvollen Animation gelangt man durch eine Untersuchung der Beweisregel auf syntaktischer Ebene: Welcher Teil s der Booleschen Struktur Φ wird durch einen anderen Term oder eine Boolesche Struktur t ersetzt, und wie wirkt sich diese Änderung auf die Visualisierung aus? Für die Animation der Regelanwendung nutzen wir die „Enthaltensein“-Eigenschaft der Visualisierung der Booleschen Strukturen aus: Die Visualisierung von s wird durch die Visualisierung von t ersetzt. Dies funktioniert, da die visuelle Darstellung von s in der von Φ geometrisch lokalisierbar ist. Die Animation endet dann in der visuelle Darstellung von $\Phi[s/t]$; siehe Abbildung 5.8 für eine schematische Darstellung.

Die Visualisierung von s verschwindet bei der Animation. Die Visualisierung von t wird entweder durch die zu animierenden Regel neu eingeführt oder ist schon auf Grund einer vorangegangene Regelanwendung vorhanden.

Beispiel 18 (Natürliche Zahlen)
Im Beweis aus Beispiel 2 wird im Teilbeweis in der Booleschen Struktur $[a+c \leq a+c]$ der Term a auf der rechten Seite der Ungleichung durch b ersetzt. Die resultierende Boolesche Struktur ist $[a + c \leq b + c]$. Wird die Addition selbst nur einmal beim ersten Auftreten im Beweis durch eine Animation visualisiert und in allen weiteren Vorkommen durch deren Ergebnis, dann läßt sich der Ersetzungsprozeß durch simples Austauschen des horizontalen Balkens von a durch den Balken von b veranschaulichen. Siehe Abbildung 5.9 für ein skizzenhaftes Beispiel anhand der Instanzen $a = 1$,

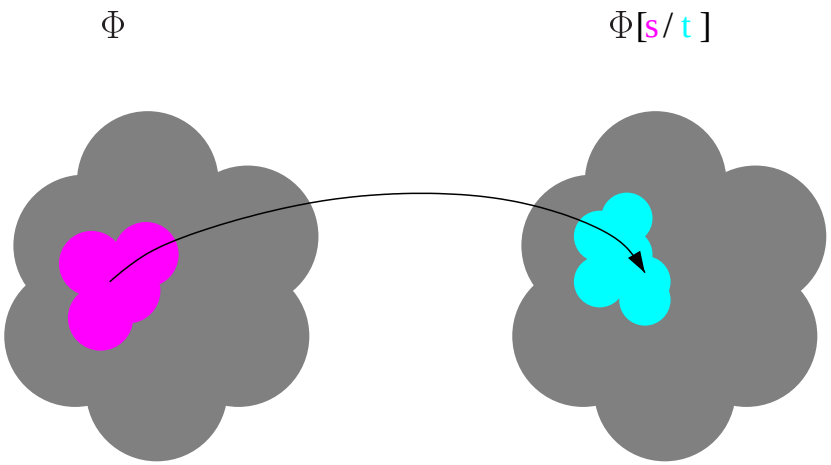


Abbildung 5.8: Schematische Darstellung einer Beweisregelanimation. Der in Φ vorkommende Teilterm s wird durch einen Teilterm t ersetzt. Die zugehörige Animation verdeutlicht diesen Ersetzungsprozeß durch Ersetzen der visuellen Darstellung von s durch die Darstellung von t.

$b = 2$ und $c = 3$.

Bei der Ersetzung von t durch s sind auch Teile der Visualisierung betroffen, die nicht zu s oder t gehören. Im vorangehenden Beispiel wurden durch den Ersetzungsprozeß die in der Visualisierung von $b + c$ enthaltende Visualisierung von c etwas nach rechts verschoben. Derartige Veränderungen sollten möglichst vermieden werden, damit der Betrachter sich allein auf die von der Beweisregel modifizierten Teile konzentrieren kann.

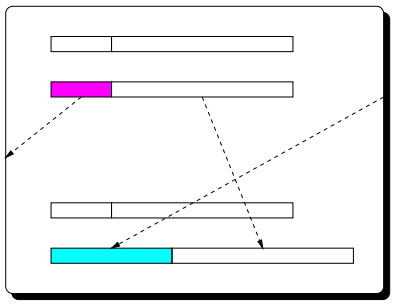


Abbildung 5.9: Beispiel einer Beweisregelanimation, die $1+3 \leq 1+3$ in $1+3 \leq 2+3$ durch Ersetzen von 1 durch 2 auf der rechten Seite der Booleschen Struktur transformiert.

Falls t in weiteren Beweisschritten verwendet wird oder, wie im folgenden Beispiel, Teil eines kompositionalen Terms u ist, sollte eine Kopie der Visualisierung von t zur Animation verwendet werden. Die originale Visualisierung von t läßt sich dann für weitere Beweisschritte oder zur Visualisierung von u verwenden.

Beispiel 19 (Pumpinglemma)
Pumpinglemma für kontextfreie Sprachen: Für jede kontextfreie Sprache L gibt es ein $n \in \mathbb{N}$, so daß es für jedes $z \in L$ mit $|z| \geq n$ eine Zerlegung $z = uvwxy$ gibt mit:

1. $|uv| \leq n$
2. $|v| > 1$
3. $uv^iwx^iy \in L$ für alle $i \in \mathbb{N}$

In einem Beweis dieses Lemmas nutzt man aus, daß es zu jeder kontextfreien Sprache L eine kontextfreie Grammatik G in Chomsky-Normalform mit $L = L(G)$ existiert. Der Beweis beruht dann darauf, in einer Ableitung $S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow uvwxy$ eines Wortes $uvwxy$ den Ableitungsschritt $A \Rightarrow^* vAx$ wegzulassen oder durch beliebig viele weitere Ableitungsfolgen $A \Rightarrow^* vAx$ zu erweitern. Die ursprüngliche Ableitungsfolge und deren Visualisierung sollte im letzten Fall erhalten bleiben.

Bei einer Visualisierung einer Ableitungsfolge durch einen Syntaxbaum, kann dieser Ersetzungsvorgang durch eine Animation veranschaulicht werden, bei welcher der zu $A \Rightarrow vAy$ gehörige Teilbaum i-mal kopiert und an entsprechender Stellte eingefügt wird. Abbildung 5.10 zeigt wie bei einer abstrakten Visualisierung der Teilbaum zweimal zur Erweiterung des Syntaxbaums verwendet wird.

In der strukturierten Variante des *Calculational proof formats* kommen als spezielle Beweisregeln noch Teilbeweise hinzu. Das Beweisformat besitzt keinerlei Einschränkungen, wie das Ergebnis eines Teilbeweises zur Transformation der vorangehenden Booleschen Struktur verwendet werden kann. Die häufigste Variante sind Teilbeweise, bei denen eine Teilstruktur ϕ einer Booleschen Struktur Φ (in mehreren Einzelschritten) in eine andere Teilstruktur ψ transformiert wird (durch äquivalente Umformung oder Implikation), um ϕ in Φ durch ψ zu ersetzen. In diesen Fällen visualisiert die Animation des Teilbeweises, den Ersetzungsprozeß von ϕ durch ψ. Aufgrund der „Enthaltensein"-Eigenschaft wird implizit auch die Ersetzung $\Phi[\phi/\psi]$ visualisiert. Die Anwendung des Teilbeweises braucht also nicht explizit berücksichtigt werden. Falls im Teilbeweis $\neg\phi$ in $\neg\psi$ transformiert wird, hängt es von der Visualisierung der Negation ab, wie der Teilbewes geeignet visualisiert wird. Falls – wie bei unseren Beispielen – die Negation nicht explizit visualisiert wird, braucht die Anwendung des Teilbeweises auch nicht explizit berücksichtigt werden.

5.3 Wie finde ich eine Ablaufstruktur des Beweises?

Wir nehmen im folgenden an, daß der zu animierende Beweis im *Structured calculational proof format* vorliegt und vom Autor geklärt wurde, wie die dort enthaltenen logischen Aussagen und Regelanwendungen gemäß obiger Methode graphisch dargestellt und animiert werden. Die Frage, mit der wir uns nun beschäftigen, ist, wie sich

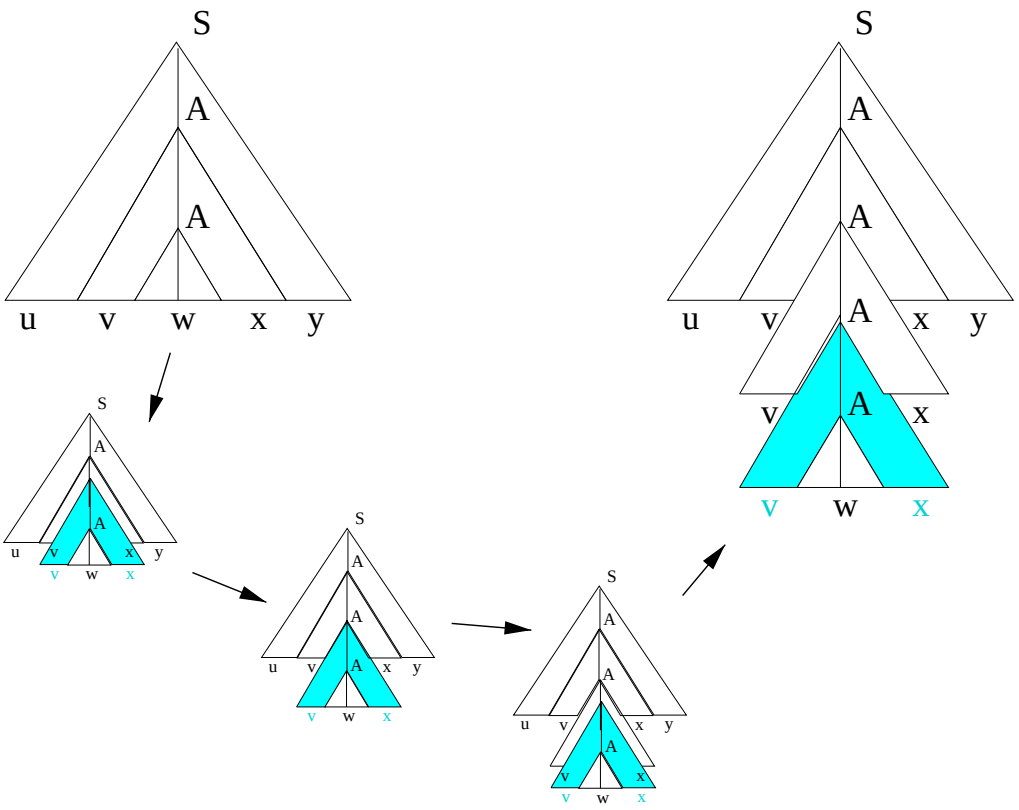


Abbildung 5.10: Animation einer Regelanwendung im Beweis des Pumpinglemmas für kontextfreie Sprache.

die visuellen Darstellungen der logischen Aussagen und die Animation der Regelanwendungen zu einer Gesamtanimation des Beweistextes zusammenfügen lassen.

Die Ablaufstruktur eines Beweistextes, insbesondere von lineare Beweisen im *Calculational proof format*, ist im wesentlichen durch den Autor vorgegeben: Ein Leser liest den Beweis ausgehend von der ersten logischen Aussage schrittweise von oben nach unten bis zur letzten logischen Aussage durch und vollzieht dabei die einzelnen Transformationschritte nach. Wenn der Beweis – wie beim von uns verwendeten *Structured calculational proof format* – geschachtelte Teilbeweise enthält, so wird bei Erreichen eines Teilbeweises dieser vom Leser ebenfalls schrittweise von oben nach unten nachvollzogen.

Basierend auf dieser Ablaufstruktur startet eine Beweisanimation mit der Visualisierung der ersten logischen Aussage und animiert die einzelnen Regelanwendungen. Die Animation endet dann mit der Visualisierung der letzten logischen Aussage. Teilbeweise entsprechen dabei Regelanwendungen, die – wie im vorangehenden Abschnitt diskutiert – implizit mit animiert werden.

Problematisch wird diese Form des Ablaufs, wenn der Beweis auch Allquantifizierungen, Induktion oder Fallunterscheidungen enthält. Existenzquantifizierungen

können durch Angabe des existierenden Elementes (das natürlich berechenbar sein muß) aufgelöst werden.

5.3.1 Ablauf einer Allquantifizierung

Der Ablauf eines allquantifizierten (Teil-)Beweises kann analog zur Visualisierung einer allquantifizierten Formel realisiert werden. Der wesentliche Unterschied ist, daß sich bei einem allquantifizierten Teilbeweis die Quantifizierung nicht nur auf eine Formel sondern auf mehrere Formeln und auf Beweisregeln bezieht.

Ein Teilbeweis, der eine allquantifizierte Variable enthält, macht eine Aussage über alle quantifizierten Elemente einer Teilmenge einer Domäne. Dies können unendliche viele sein, etwa bei einer Quantifizierung über alle natürlichen Zahlen. Beim logischen Schließen mit Diagrammen versucht man für spezielle Bereiche, etwa Beweise mengentheoretischer Aussagen, derartige Quantifizierungen in den Diagrammen mit darzustellen, etwa durch Darstellung einer Menge als Kreis oder Region. Bei derartigen Diagrammen wird von den konkreten Elementen abstrahiert, es handelt sich dabei also um abstrakte Visualisierungen von Teilen der Domäne. Falls der Autor eine derartige Visualisierung fixiert hat, braucht ein Allquantor bei der Ablaufbeschreibung nicht explizit berücksichtigt zu werden: der quantifizierte Beweis kann anhand der abstrakten Darstellung animiert werden.

Falls die Domäne konkret visualisiert wird, funktioniert dieser Ansatz nicht mehr. Hier gibt es zwei verschiedenen Möglichkeiten, bei der Ablaufstruktur den Allquantor zu behandeln. Welche Wahl getroffen wird, hängt von der Verwendung des Allquantors im Beweis ab:

1. Einsetzen einer konkreten Instanz bei Erreichen des allquantifizierten Teilbeweises. Diese Vorgehensweise bietet sich besonders für unendlichen Bereiche an: So wie bei einer Algorithmenanimation die graphischen Darstellungen von einer konkreten Eingabe des Programms abhängen, welches den Algorithmus implementiert, können auch die Elemente, über welche in einem Theorem eine Allaussage getroffen wird, im Beweis anhand eines konkreten Elements veranschaulicht werden.

2. Iteration über alle quantifizierten Elemente bei einer Quantifizierung über einen endlichen Bereich. In einigen Fällen kommen Allquantifizierungen in Beweisen vor, die keine Korrespondenz zu einem Allquantor im zu beweisenden Theorem besitzen, sondern zur Dekomposition oder Konstruktion komplexerer, aber endlicher Objekte verwendet werden, wie etwa die Indizierung im Korrektheitsbeweis des Cocke-Kasami-Younger-Algorithmus, siehe Anhang B.1. In diesen Fällen sollte in der Ablaufstruktur über alle quantifizierten Elemente iteriert werden, um den konstruktiven Charakter zu betonen. Der quantifizierte Teilbeweis wird also mehrfach durchlaufen.

5.3.2 Ablauf einer Induktion

Ein Induktion verhält sich ähnlich wie eine Allquantifizierung über einen endlichen Bereich. Wenn eine abstrakte Visualisierung gewählt wurde, dann können erst der Induktionsanfang und darauf—folgend der Induktionsschritt anhand der gewählten graphischen Darstellungen der Domäne visualisiert werden. Eine Iteration ist nicht notwendig.

Eine Induktion (über natürlichen Zahlen) besitzt etwa folgendes Schema:

$$(P(0) \wedge (\forall n \in \mathbb{N} : P(n) \Rightarrow P(n+1))) \Rightarrow \forall n \in \mathbb{N} : P(n)$$

In unserem Beweisformat kommen Induktionsanfang $P(0)$ und Induktionsschluß $\forall n \in \mathbb{N} : P(n) \Rightarrow P(n+1)$ vor. Obige Konklusion ist die zu beweisende Induktionshypothese.

Bei einer konkreten Visualisierung wird ein bestimmter Wert des Induktionsparameters n festgelegt, der von anderen allquantifizierten Variablen abhängen kann, die vorher vom Betrachter mit einem konkreten Wert belegt wurden. n kann etwa die Länge einer Zeichenkette sein. Die Induktion beschreibt dann eine Konstruktion ausgehend vom Induktionsanfang, über eine mehrfache Anwendung der Induktionsvoraussetzung im Induktionsschluß bis der Wert des Induktionsparameters erreicht wurde. Bei konkreten Visualisierungen läßt sich der Ablauf eine Induktion also durch eine Iteration von 0 bis zum konkreten Wert von n beschreiben. Es ist auch möglich die Induktion rückwärtsgerichtet zu lesen: Ausgehend vom konkreten Wert des Induktionsparameters wird zurückgezählt bis der Induktionsanfang erreicht wurde. Diese Sichtweise entspricht bei einem Algorithmus einer Rekursion. Eine sinnvolle Wahl der Ablaufrichtung – vorwärts oder rückwärts – hängt vom konkreten Beweis ab. Wir halten einen vorwärtsgerichteten Ablauf allerdings für geeigneter, weil sie analog zu einer Iteration in einem Algorithmus für den Menschen leichter nachzuvollziehen ist, als ein rückwärtsgerichteter Ablauf wie bei einer Rekursion.

5.3.3 Ablauf einer Fallunterscheidung

Wir haben zwei Formen von Fallunterscheidungen kennengelernt: Eine allgemeine, bei der für disjunktiv verknüpfte Teilstrukturen jeweils getrennte Teilbeweise geführt wird, und eine spezielle, bei der nach verschiedenen Instanzen einer allquantifizierten Aussage unterschieden wird.

Programme enthalten gewöhnlich Verzweigungen im Programmfluß. Eine Verzweigung wird während der Ausführung deterministisch in Abhängigkeit vom aktuellen Programmzustand ausgewählt. Dieser Ablauf läßt sich derart allgemein nicht auf eine Fallunterscheidung übertragen, da alle oder mehrere Fälle simultan zutreffen können. Dies hängt im wesentlichen von der Wahl der Visualisierung – abstrakt oder konkret – ab:

1. Bei einer abstrakten Visualisierung trifft – nach Definition – die Visualisierung

immer auf jeden Teilbeweis in einer Fallunterscheidung zu (falls nicht, dann ist die visuelle Darstellung falsch gewählt worden).

2. Bei einer konkreten Visualisierung kann die konkrete Instanz auch mehr als nur einen Fall erfüllen.

Man kann diesen Indeterminismus im Beweisablauf auf folgende Weisen begegnen:

1. Durch Auswahl eines zutreffenden Falls aus mehreren möglichen. Dies kann entweder durch den Beweisautor a priori oder durch den Benutzer während des Beweisablaufs geschehen.

2. Durch sequentielle Ausführung jedes zutreffenden Falls.

3. Durch simultane Ausführung jedes zutreffenden Falls: Ablauf des ersten zutreffenden Falls, dann des nächsten usw.

Letztere Variante ist aus didaktischer Sicht ungeeignet, da der Betrachter einen mitunter komplizierten parallelen Beweisablauf und dabei mehrere simultane Visualisierungen verfolgen muß. Eine sequentielle Ausführung ist einfacher nachzuvollziehen, könnte den Betrachter aber verwirren, da unmotiviert nach Ablauf des erste Falls wieder zurück zu einem weiteren gesprungen wird. Insbesondere widerspricht dies der algorithmischen Erfahrung unserer Studierenden. Die Auswahl eines zutreffenden Falls durch den Betrachter selbst stellt unserer Ansicht nach die sinnvollste Variante dar: Zum einen bekommt der Studierenden nicht den irreführenden Eindruck eines Determinismuses und zum anderem wird er gezwungen, sich genauer mit dem Beweis auseinanderzusetzen.

Kapitel 6

SCAPA

In diesem Abschnitt beschreiben wir SCAPA – **S**tructured **CA**lculational **P**roof **A**nimator – unserem System, mit dem Beweise im *Structured calculational proof format* und zugehörige Beweisanimationen rechnergestützt erzeugt und präsentiert werden können. SCAPA besteht hauptsächlich aus zwei Teilen:

1. Einem WWW-basierten System, das dem Benutzer erlaubt, mit einem WWW-Stöberer durch hierarchisch strukturierte Beweise zu stöbern[1] sowie die Animation zum Beweis zu starten und zu steuern. Der Benutzer kann den Beweis schrittweise inspizieren, während er die Gültigkeit der einzelnen Regeln anhand der begleitenden Animation visuell nachvollziehen kann. Wir stellen diesen Teil von SCAPA aus Sicht des Benutzers in Abschnitt 6.1 vor.

2. Einer Sammlung von Werkzeugen, die vom Autor eines in LaTeX geschriebenen Beweises benutzt werden können, um diesen automatisch in einen hierarchisch aufgebauten HTML-Beweistext und in Java-Quellkode transformieren zu können. Der Javakode dient als Rahmen, um die Beweisanimation zu implementieren und um sie mit dem Stöberer zu verbinden und zu synchronisieren. Wir stellen diesen Teil von SCAPA aus der Sicht des Autors in Abschnitt 6.2 vor.

6.1 Benutzersicht

SCAPA kann mit jedem verfügbaren Java- und JavaScript-fähigen WWW-Stöberer, wie Netscape oder dem Internet Explorer, ohne Installation zusätzlicher Software benutzt werden. Eine Beweisanimation besteht aus dem HTML-Beweistext und einem Java-Applet, welches die zugehörige Animation implementiert. Abbildung 6.1 zeigt ein Bildschirmphoto von SCAPA, das die HTML-Version des in Abschnitt 2 ausgeführten strukturierten Beweises zeigt.

Der Teilbeweis ist verdeckt, nur der Hinweis zum Teilbeweis und ein kleines Dreieck, welches die Position des Teilbeweises angibt, sind sichtbar. Wenn der Benutzer

[1] Engl.: *to browse*

auf dieses Dreieck klickt, dann wird der zugehörige Teilbeweis sichtbar. In gleicher Weise kann der Benutzer auch Teilbeweise wieder verdecken. Wir haben diese Form des „Stöberns“ im wesentlichen von PROOFVIEWS übernommen.

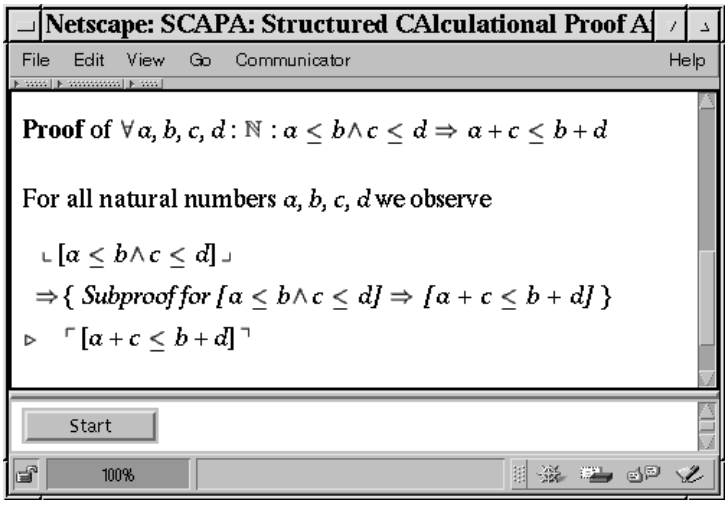


Abbildung 6.1: Ein Bildschirmphoto eines Beweises mit einem verdeckten Teilbeweis

Falls der Benutzer eine ihm genehme Sicht des Beweises ausgewählt hat, dann kann er die Animation durch Anklicken des Knopfes „Start“ starten. Dieser Knopf befindet sich in einem separatem HTML-Rahmen unter dem HTML-Rahmen des Beweistextes. Wenn der Knopf angewählt wurde, erscheint ein neues Fenster („Lambda Control“), das einige Dialogelemente zur Steuerung der Animation enthält, und es erscheinen ein oder mehrere Fenster, die die Animation enthalten („Animation“). Während der Beweisanimation wird die zum aktuellen Zustand der Animation gehörige Boolesche Struktur bzw. der zugehörige Hinweis simultan zur Animation durch Farbumschlag hervorgehoben. Abbildung 6.2 zeigt das System, nachdem der Teilbeweis vom Benutzer aufgedeckt und einige Beweisschritte ausgeführt wurden. Es zeigt ein Bildschirmphoto aus einer Animation zum Beweis der Aussage $\forall a, b, c, d \in \mathbb{N} : a \leq b \wedge c \leq d \Rightarrow a + c \leq b + d$, wie sie in Abschnitt 5.2.2, Beispiel 18 beschrieben wurden: Die Domäne des natürlichen Modells der Aussage besteht aus den natürlichen Zahlen. Sie werden als horizontale Balken visualisiert, die Addition zweier Zahlen wird als Animation visualisiert, bei der die zugehörigen Balken aneinandergefügt werden und die Ungleichheit zweier Zahlen wird durch Übereinanderstellen der horizontalen Balken mit Ausrichtung auf die linke Ecke visualisiert. Die im Beweis verwendeten Regeln werden ebenfalls durch eine Animation visualisiert, bei der die beteiligten horizontalen Balken geeignet verschoben werden.

Mit dem Steuerungsfenster kontrolliert der Benutzer die Geschwindigkeit der Animation, hält sie an oder führt sie schrittweise fort. Die Fenster, die die Animationen enthalten, besitzen Schalter, um den Fensterinhalt nach oben („up“), unten („down“),

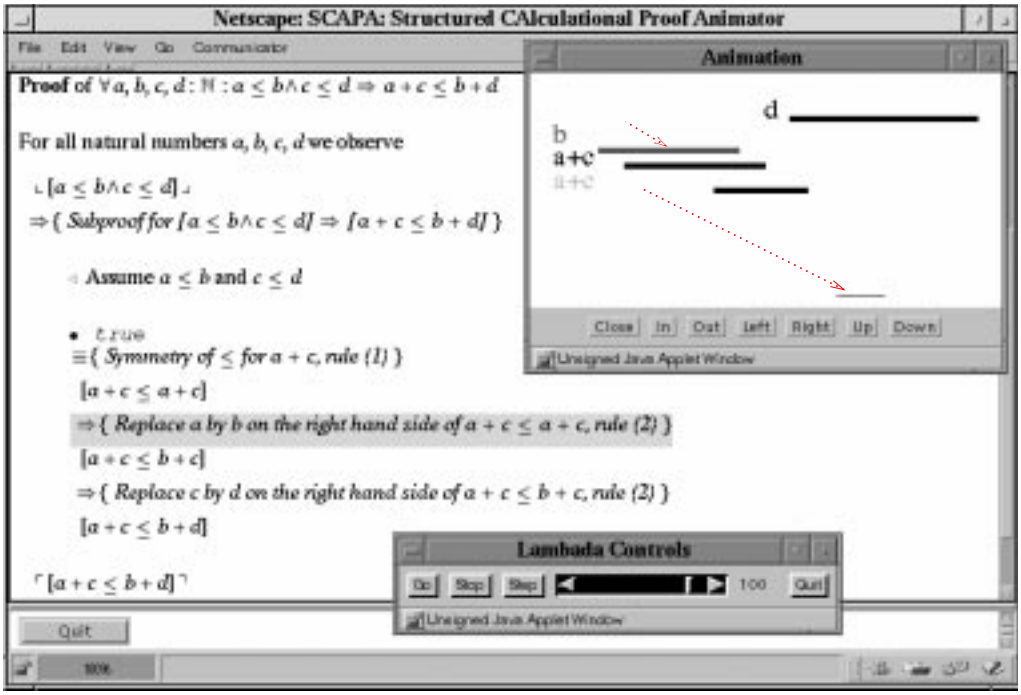


Abbildung 6.2: Ein Bildschirmphoto mit einem Ausschnitt einer Beweisanimation

rechts („right“) oder links („left“) zu verschieben, um den Fensterinhalt zu vergrößern („in“) oder zu verkleinern („out“) und um das Fenster zu schließen („close“). Während der Animation wird die gerade aktuelle Boolesche Struktur im Beweistext hervorgehoben: Durch einen grünen Hintergrund, falls die ausgewertete Boolesche Struktur den Wahrheitswert `true` besitzt und durch einen roten Hintergrund, falls sie den Wahrheitswert `false` besitzt. Abbildung 6.2 zeigt den Zustand der Beweisanimation, bei der Regel (2) (Ersetzung eines Term s, der eine natürliche Zahl repräsentiert, auf der rechten Seite einer Ungleichung $\leq$ durch einen Term t für den $[s \leq t] \equiv \texttt{true}$ gilt) auf die Ungleichung $a + c \leq a + c$ angewendet wurde. Der Hinweis zur Regelanwendung im Beweistext ist mit gelber Farbe hervorgehoben, und simultan dazu zeigt die Animation wie in der visuelle Darstellung a durch b auf der rechten Seite der Ungleichung ersetzt wird. Abbildung 6.2 zeigt die Animation in einem Zustand nach etwa der Hälfte der abgelaufenen Zeit. Die roten gepunkteten Pfeile sind nicht Teil dieser Animation, sondern deuten nur an, wie sich die zu b und a gehörigen Balken bewegt haben. Nach der Regelanwendung ist die Animation in einem Zustand, der die resultierende Boolesche Struktur $a + c \leq b + c$ visualisiert.

Die Instanzen a, b, c und d können in diesem einfachen Beispiel nicht durch den Benutzer vor Beginn der Animation selbst eingegeben werden, sondern sind durch den Autor der Animation vorgegeben worden. Die konkreten Werte der Variablen sind für das Verständnis der Argumentationen im Beweistext nicht von Belang, sondern nur

der relativer Größenunterschied der Variablen. Es handelt sich bei der Animation um eine abstrakte Visualisierung des Modells.

Die Ablaufstruktur des Beweistextes und der zugehörigen Animation ist bei SCAPA im wesentlichen wie in Abschnitt 5.3 dargelegt realisiert. Die Ablaufstruktur hängt von der Wahl der Visualisierung ab, ob konkret oder abstrakt, und muß deswegen im voraus durch den Animationsautor festgelegt werden. Generell wird der Beweis schrittweise von Anfang bis Ende durchlaufen. Der Ablauf bei einem Allquantor wird bei Erstellung der Animation festgelegt. Es gibt zwei Möglichkeiten:

1. Durch Instantiierung der Variablen durch den Benutzer oder Beweisautor oder

2. durch eine fortgesetzte Iteration über den quantifizierten Bereich.

Bei einer Induktion wird ausgehend vom Basisfall (oder den Basisfällen) schrittweise solange durch den Induktionsschritt iteriert, bis der Wert der Induktionsvariablen erreicht worden ist. Bei einer Fallunterscheidung wird während der Animation basierend auf dem aktuellen Zustand ein zutreffender Fall deterministisch ausgewählt. Eine Auswahl durch den Benutzer ist nicht vorgesehen (aber technisch mit unserem System möglich).

Die in diesen Abschnitt von uns zur Illustration verwendete Animation eines Beweises der sehr simplen Aussage $\forall a, b, c, d \in \mathbb{N} : a \leq b \wedge c \leq d \Rightarrow a + c \leq b + d$ stellt noch keine besonderes realistische Anwendung unseres Systems dar. In Kapitel 7 zeigen wir zwei ausführliche Beispiele von Animationen zu Beweisen von Aussagen, wie sie in der Ausbildung in der Theoretischen Informatik vorkommen. Im folgenden konzentrieren wir uns auf die Erstellung derartiger Animationen mit SCAPA.

6.2 Autorensicht

Aus Sicht des Autors ist SCAPA eine Sammlung von Werkzeugen, um strukturierte Beweise zu schreiben, diese in Javakode und eine HTML-Version des Beweises zu transformieren, und um die zugehörige Beweisanimation zu implementieren. Abbildung 6.3 zeigt die einzelnen Schritte, um eine Beweisanimation systematisch mit SCAPA zu erstellen (wir erläutern die Details in den nächsten Abschnitten): Schreiben des Beweises in LaTeX (1); Konvertieren (oder direktes Schreiben) des Beweises in HTML mit Hilfe von „Pseudo-Tags“ (2); Transformieren der so annotierten HTML-Version in reines HTML (3) und Erzeugen von Javakode (4); Erweitern des Javakodes durch Überschreiben von Methoden zu einer vollständigen Implementierung der Booleschen Strukturen und der Hinweise (5); und Erweiterung dieser Implementierung mit Anweisungen für das Algorithmenanimationssystem LAMBADA (6). Der resultierende HTML-Beweistext (3) kann mit einen WWW-Stöberer betrachtet werden, und die zugehörige Animation (6) kann gestartet werden.

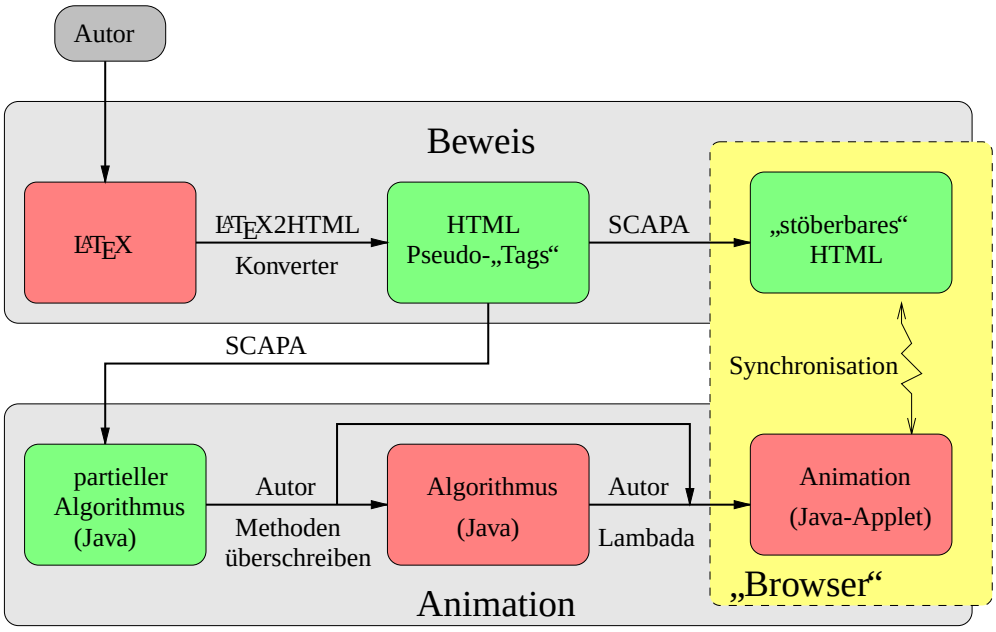


Abbildung 6.3: Entwickeln einer Beweisanimation mit SCAPA

6.2.1 Schreiben des Beweistextes

Der Beweistext kann entweder in LaTeX geschrieben werden, einem Textsatzsystem im Stil einer *Mark-up*-Sprache [LAMPORT 1986], oder direkt in HTML mit Hilfe spezieller „Pseudo-Tags“.

LaTeX wird im universitären Bereich häufig benutzt, um technische Dokumente oder Texte zu erstellen, die viele (mathematische) Symbole enthalten, so wie das im Bereich der Theoretischen Informatik der Fall ist. Wir haben einen LaTeX-Makro-Paket (LaTeX-Stil) zum Setzen von Beweisen im *Structured calculational proof format* entwickelt, mit dem die Strukturinformationen des Beweises, wie in Abschnitt 5.1 erläutert, explizit gekennzeichnet werden. In Anhang D.1 beschreiben wir detailliert dieses LaTeX-Stil.

Die LaTeX-Version des Beweises wird automatisch mit dem frei erhältlichen LaTeX-2HTML-Konverter [DRAKOS 94] nach HTML transformiert. Der LaTeX2HTML-Konverter sucht zu jedem im Dokument verwendeten LaTeX-Stil (eine Datei) nach einer Perl-Datei gleichen Namens. Diese Perl-Datei beschreibt im einzelnen, wie die speziellen Kommandos und Umgebungen des zugehörigen LaTeX-Stils nach HTML konvertiert werden. Wir haben diesen Mechanismus benutzt, um den Konverter so zu erweitern, daß die speziellen LaTeX-Kommandos zum Setzen des Beweises in HTML-„Pseudo-Tags“ übersetzt werden. Diese „Tags“ enthalten alle Strukturinformationen des Beweises. Wir benutzen diese „Pseudo-Tags“, um daraus eine reine HTML-Version des Beweistextes zu erzeugen (siehe nächsten Abschnitt).

Diese zweistufige Vorgehensweise erlaubt es, einen Beweis anstatt in LaTeX auch direkt in HTML zu schreiben (zum Beispiel mit einem HTML-Editor). Da HTML keine speziellen mathematischen Zeichen und das Setzen von mathematischen Formeln unterstützt, bevorzugen wir den Ansatz der Konvertierung von LaTeX. Unsere bisherigen Erfahrungen (und Bildschirmphotos) zeigen, daß die resultierende HTML-Version von guter Qualität ist. Desweiteren bleibt SCAPA aufgrund dieses zweistufigen Ansatzes offen für weitere Textformate wie Framemaker oder Word-Doc, falls mit den dort zur Verfügung stehenden Konvertierungswerkzeugen die entsprechenden „Pseudo-Tags“ erzeugt werden können.

Abbildung 6.4 zeigt einen Ausschnitt aus der LaTeX-Version des Beweises aus Abbildungen 6.1 und 6.2. Die dunkler schattierten Textteile markieren die Strukturinformationen, die anderen realisieren den Beweistext in einer für den LaTeX erfahrenen Autor gewohnten Form. Eine detaillierte Beschreibung unserer Stildatei geben wir in Anhang D.1. Abbildung 6.5 zeigt einen Ausschnitt der zugehörgigen mit dem Konverter erzeugten HTML-Version. Zur besseren Übersicht sind die meisten automatisch erzeugten HTML Teile nur mit Punkten angedeutet und teilweise von uns per Hand eingerückt. Die Strukturinformationen sind in „Pseudo-Tags“ von der Form <CPROOF ... > ... < \CPROOF> kodiert. Wir erläutern diese „Pseudo-Tags“ genauer in Anhang D.2.

6.2.2 Erzeugen des HTML-Beweistextes

Zwei von SCAPAs Werkzeugen – `cproof2html` und `fold` (siehe Anhänge C.2 und C.3) – produzieren aus dem annotiertem HTML-Beweistext eine reine HTML-Version. Mit `cproof2html` werden die in den „Pseudo-Tags“ enthaltenden Strukturinformationen des HTML-Beweistextes zur Formatierung der HTML-Version des Beweises benutzt. Das Resultat ist ein reiner HTML-Text, mit Ausnahme der Kennzeichnung von Teilbeweisen. Das zweite Werkzeug `fold` nutzt diese Information, um den HTML-Text in eine stöberbare Version zu transfomieren. Anstatt, wie bei PROOFVIEWS implementiert, für jede mögliche Kombination von verdeckten und sichtbaren Teilbeweisen (exponentiell viele) jeweils eine HTML-Datei zu erstellen, erzeugen wir nur eine einzige HTML-Datei und nutzen JavaScript, um Teilbeweise zu verdecken oder wieder sichtbar zu machen. Das Resultat von `fold` ist eine einzelne, kompakte HTML-Datei.

Wann immer der Benutzer die Sicht des HTML-Beweistextes ändert, wird der aktuelle Status der verdeckten und sichtbaren Teilbeweise dem automatisch erzeugten Java-Programm mitgeteilt. Dort kann diese Information zur adaptiven Gestaltung der Animation benutzt werden. Zum Beispiel können die Teile einer Animation, die zu verdeckten Teilbeweisen gehören, abgekürzt oder ausgelassen werden.

```
\begin{cproof}
  F"ur alle nat"urlichen Zahlen $a,b,c,d$ gilt:
  \begin{cstructure}
  $\cexpand{ [a\leq b\land c\leq d] }$
  \end{cstructure}
  \begin{cbecause}{$\Rightarrow$}
  Teilbeweis von $[a\leq b\land c\leq d]\Rightarrow
                  [a+c\leq b+d]$
  \end{cbecause}
  \begin{csubproof}
    ...
    \begin{cbecause}{$\equiv$}
    Symmetrie von $\leq$ f"ur $a+c$, Regel (1)
    \end{cbecause}
    \begin{cstructure}
    $[a+c\leq a+c]$
    \end{cstructure}
    ...
  \end{csubproof}
  \begin{cstructure}
  $\cresult{ [a+c\leq b+d] }$
  \end{cstructure}
\end{cproof}
```

Abbildung 6.4: Ausschnitte aus der LaTeX-Version eines Beweistextes

6.2.3 Erzeugen des Java-Quelltextes

Ein drittes von SCAPAs Werkzeugen – `cproof2java`, siehe Anhang C.1 – benutzt die HTML-ähnlichen „Pseudo-Tags“, um aus der textuellen Beschreibung des Beweises Java Quelltext zu extrahieren. Dieser Quelltext dient zur Synchronisation der Animation und zu dessen Verbindung mit dem HTML-Beweistext. Der Autor einer Beweisanimation braucht nichts über diese internen Details von SCAPA zu wissen. Desweiteren dient der Quelltext als Rahmen für den Algorithmus, der zur Implementierung der Animation dient: Alle explizit mit den „Pseudo-Tags“ angegebenen Beweisregeln – Induktion, Fallunterscheidung, etc. – werden benutzt, um aus dem Beweis einen partiellen Algorithmus zu synthetisieren. Folgende, dabei nicht automatisch erzeugten Teile werden durch (leere) Methodenaufrufe implementiert:

```
<CPROOF>
  F&uuml;r alle nat&uuml;rlichen Zahlen <I>a< \I>, ...
    <CPROOF CMD=STRUCTURE ID=ID1>
    <IMG ... >[<I>a< \I> <IMG ...
  < \CPROOF>
  <CPROOF CMD=BECAUSE ID=ID2>
  <OPERATOR> <IMG ... SRC="img5.gif" ...> < \OPERATOR>
    Teilbeweis von ...
  < \CPROOF>
  <CPROOF CMD=SUBPROOF ID=0>
    <FOLDME ID=0> ... < \FOLDME>
  < \CPROOF>
    <CPROOF CMD=STRUCTURE ID=ID10>
    <IMG ... SRC="tl-blue.gif" >[<I>a< \I> + ...
  < \CPROOF>
< \CPROOF>
```

Abbildung 6.5: Ausschnitte aus der HTML-Version des Beweistextes

- Für jede Boolesche Struktur existiert jeweils eine Methode, die den Wahrheitswert dieser Struktur berechnet und eine, welche die Struktur selbst implementiert.
- Für jeden Hinweis existiert eine Methode, die die zugehörige Regelanwendung implementiert.
- Für jede Quantifizierung (∀,∃, aber auch Induktionsvariablen) existiert eine Methode, um die Variable einzuführen und wieder zu zerstören (∀) oder deren Wert aus den gerade sichtbaren Werten zu berechnen (∃, Induktionsvariablen).

Diese Methoden werden an entsprechenden Stellen im Algorithmus aufgerufen. Der Autor der Animation muß diese Methoden durch Überschreiben geeignet implementieren (Punkt (5) in Abbildung 6.3). Dazu braucht er den automatisch erzeugten Quelltext nicht zu ändern, sondern kann durch Erweitern dieser Klasse und Überschreiben der Methoden die offenen Teile implementieren. Der so implementierte Algorithmus wird mit Befehlen für das von uns verwendete Animationssystem LAMBADA angereichtert, um eine Animation des Beweises zu erzeugen. Unsere Erfahrungen mit der Erstellung von Beweisanimationen zeigen, daß die Annotation des Algorithmus bzw. der Methoden am besten durch ein weiteres Überschreiben der Methoden implementiert werden sollte, da dies eine klare Trennung von dem zum Beweis gehörigen Algorithmus und dessen Animation erlaubt. Durch diese objektorientierten Ansatz kann das verwen-

dete Animationssystem dann leichter durch ein anderes Java-basiertes Animationssystem ersetzt werden, ohne den einmal implementierten Algorithmus wieder ändern zu müssen. Zusätzlich erlaubt es dieser Ansatz, den Beweis(text) und die zugehörige Animation einfacher zu warten: Werden im LaTeX-Beweistext nach Fertigstellung der Animation nur geringfügige Änderungen, wie Ändern von Indizes oder Übersetzung der verwendeten natürlichen Sprache (Deutsch) in eine andere (Englisch), vorgenommen, so ist es nicht notwendig, die vom Benutzer implementierten Methoden und die Animation zu ändern. Die fertige Java-basierte Animation wird als Applet in den Beweistext integriert.

Um die Zuordnung der automatisch erzeugten Methoden zum Beweistext zu erleichtern, benutzen wir das `javadoc`-Werkzeug des Java Developer Kits (JDK). Für jede Methode, die potentiell vom Autor überschrieben werden kann, wird in der von `javadoc` erstellten HTML-Dokumentation der entsprechende Teil des Beweises (die Boolesche Struktur oder der Hinweis) zur Dokumentation der Methode benutzt (Abbildung 6.6).

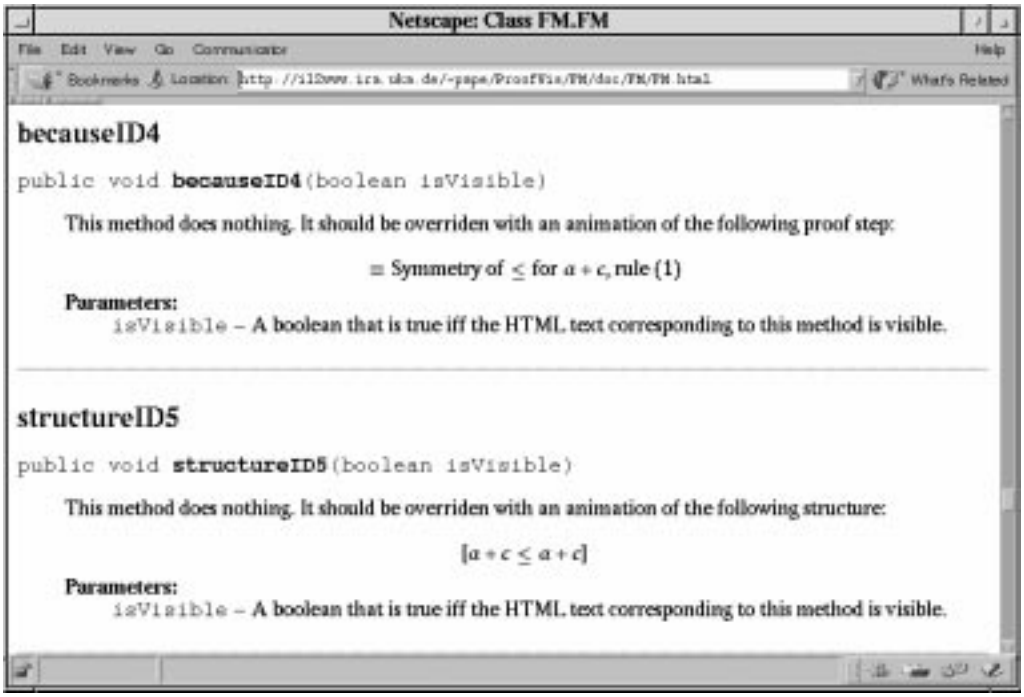


Abbildung 6.6: Beispiel einer mit `javadoc` erstellen Dokumentation des automatisch erzeugten Javakodes

6.2.4 Implementierung der Animation

Zur Erzeugung der Beweisanimation benutzen wir das frei verfügbare Algorithmenanimationssystem LAMBADA [MITCHENER et al. 1996]. Wir haben LAMBADA um

spezielle Kommandos erweitert, mit denen der HTML-Text im Stöberer vom Animationssystem aus während der Animation farblich markiert werden kann. LAMBADA ist vollständig in Java implementiert und ist kompatibel mit dem Algorithmenanimationssystem Samba [STASKO 1996b]. Analog zum Schreiben einer Algorithmenanimation mit LAMBADA, muß der Autor einer Beweisanimation die interessanten Ereignisse (Boolesche Strukturen und Hinweise) des automatisch erzeugten und manuell erweiterten Algorithmus mit LAMBADA-spezifischen Animationsbefehlen implementieren. Diese Befehle erlauben es, zwei-dimensionale geometrische Objekte wie Kreise, Linien oder Text zu erzeugen und zu manipulieren (Farbänderungen, fließenden Bewegungen).

Abbildung 6.7 zeigt die derzeitige Architektur der Darstellungskomponente von SCAPA: Mit der HTML-Darstellungskomponente des Stöberers („Pagelayouter“) und JavaScript haben wir den stöberbaren Beweistext realisiert. Die Beweisanimation ist als Applet bestehend aus dem Algorithmus und dem Animationssystem implementiert. Beide Teile sind durch Implementierung als Java-Threads voneinander entkoppelt. Der Zustand der vom Benutzer auf- und zugedeckten Teilbeweise wird von JavaScript vor Start der Animation an den Algorithmus übermittelt. Nach Start der Animation erzeugt der Algorithmus basierend auf diesen Informationen und auf optionalen Benutzereingaben ein Animationsskript und übermittelt es innerhalb des Applets dem Animationssystem. Das Animationssystem übernimmt die Kontrolle und markiert während des Ablaufs der Animation die zugehörigen Teile des Beweistextes. Beweistext und Animation sind über „Live Connect“ – einer Schnittstelle um Daten zwischen Java-Script und Java-Applet auszutauschen – gekoppelt. Im Gegensatz zu dieser proprietären aber nicht standardisierten Schnittstelle, stellt das vom WWW-Konsortium festgelegt „Document Objekt Modell“ (DOM) eine alternative Schnittstelle dar, ist aber derzeit nur in Teilen beim Internet Explorer implementiert (Version 4.0).

Wir haben LAMBADA verwendet, da dessen Befehlssatz nicht auf die Animation von Programmen, die in einer bestimmten Programmiersprache implementiert wurden, beschränkt ist, sondern sehr flexibel zur Animation beliebiger Objekte eingesetzt werden kann. Desweiteren kann ein vollständig Java- (und JavaScript-)basiertes System off-line benutzt werden. Diese Anforderung ergab sich aus unserer frühen Erfahrungen mit dem Einsatz CGI-basierter Programme, die gezeigt hat, daß Studierende Lernsoftware nur unzureichend benutzen, wenn sie nur im Rechnerlabor oder nach langwieriger Installation auf dem eigenen Rechner oder langen Ladezeiten über das WWW benutzt werden kann. Siehe dazu die Diskussion in Abschnitt 3.5.

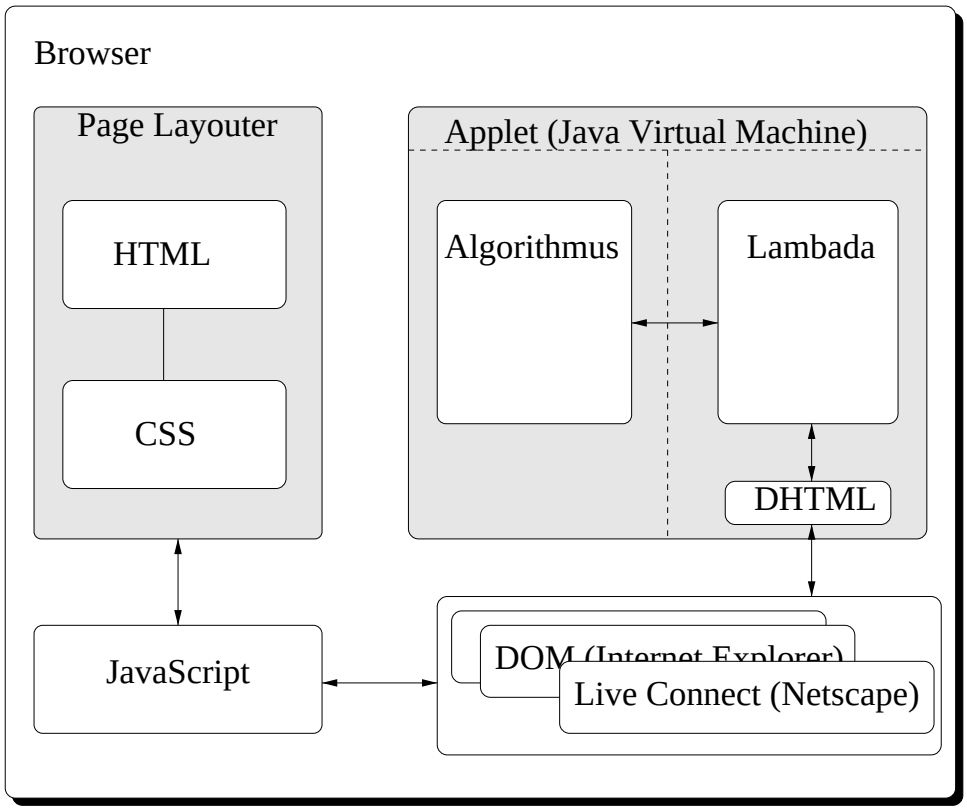


Abbildung 6.7: Archiktektur der Darstellungskomponente von SCAPA

Kapitel 7

Fallbeispiele

In diesem Kapitel beschließen wir unsere Arbeit mit zwei ausführlichen Beispielen von SCAPA-Beweisanimationen realistischer Theoreme. Das erste Beispiel stammt aus dem Bereich Formale Sprachen. Es veranschaulicht einen Korrektheitsbeweis des in einführenden Vorlesungen gerne behandelten Cocke-Kasami-Younger-Parsingalgorithmus. Das zweite Beispiel stammt aus dem Bereich Berechenbarkeit. Es veranschaulicht eine Richtung in einem Beweis zur Äquivalenz zweier verschiedener Typen von Turingmaschinen. Die ausführlichen Beweistexte mit einigen zusätzlichen Erläuterungen befinden sich in Anhang B.

7.1 Korrektheit des Cocke-Kasami-Younger-Algorithmus

Den Cocke-Kasami-Younger-Parsingalgorithmus und eine zugehörige Algorithmenanimation haben wir in Abschnitt 2.3.1 vorgestellt. Um die Studierenden davon zu überzeugen, daß der Algorithmus für eine Grammatik G entscheidet, ob ein Wort w in $L(G)$ ist oder nicht, wird in der Vorlesung die Korrektheit des Algorithmus bewiesen. Dazu ist zu zeigen, daß $w \in L(G)$ genau dann gilt, wenn das Startsymbol in der obersten Zelle der Pyramide enthalten ist. In der Vorlesung wird folgende etwas stärkere Aussage bewiesen:

Lemma 5 *$G = (N, \Sigma, S, P)$ sei eine Grammatik in Chomsky-Normal-Form, $w \in \Sigma^*$ mit $n = |w|$ und $zelle$ die vom CKY-Algorithmus berechnete Funktion, dann gilt für alle $1 \leq i \leq n, 1 \leq j \leq n - i$:*

$$zelle(i, j) = \{X \in N \mid X \Rightarrow^* w_j \ldots w_{j+i-1}\}$$

Falls der Algorithmus selbst gut verstanden wurde, ist die Aussage des Theorems einsichtig. Der Korrektheitsbeweis ist überschaubar und unkompliziert. Er eignet sich deswegen gut für ein erstes realistisches Fallbeispiel.

Die Domäne des natürliche Modells des Satzes besteht aus folgenden Mengen:

- Grammatiken in Chomsky-Normalform bestehend aus Terminal- und Nichtterminalzeichen. Der Einfachheit halber legen wir — wie bei der Algorithmenanimation — $N = \{A, B, \ldots, Z\}, \Sigma = \{a, b, \ldots, z\}$ und S als Startsymbol fest.
- Folgen von Terminal- und Nichtterminalzeichen (Wörter) und den Produktionen,
- der dazu gehörigen vom Algorithmus berechneten Funktion $zelle$ und
- Ableitungsfolgen.

Wir visualisieren diese Elemente (bis auf Ableitungsfolgen) wie folgt mit konkreten Darstellungen:

1. Die Terminalzeichen textuell als einzelne, kleine Buchstaben.
2. Die Nichtterminalzeichen textuell als einzelne große Buchstaben.
3. Die Wörter werden textuell als Folgen der Visualisierungen der einzelnen Zeichen dargestellt.
4. Die Produktionen textuell als mit einem Pfeil verbundene rechte und linke Seite der Produktion. Die Visualisierung enthält also die Visualisierung von Nichtterminalzeichen und Wörtern.
5. Die Grammatiken durch vertikale Auf—listung ihrer Produktionen. Die Visualisierung einer Grammatik enthält die Visualisierung der Produktionen.
6. Die Funktion $zelle$ wird durch die von der Algorithmenanimation gewohnten Darstellung als Pyramide visualisiert. Die von $zelle$ berechneten Mengen von Nichtterminalzeichen werden textuell in die zugehörige Zelle geschrieben. Eine Zelle enthält die Visualisierung der Nichtterminalzeichen.
7. Ableitungsfolgen werden durch eine abstrakte Visualisierung des Syntaxbaums dargestellt. Es wird vom linken und rechten Teilbaum der Wurzel abstrahiert, da deren konkreten Elemente für das Verständnis der Induktion nebensächlich sind. Der Syntaxbaum enthält die Visualisierung der Nichtterminal- und Terminalzeichen.

Der Beweis ist ein Gleichheitsbeweis, der in der Kurzform des Beweisformats niedergeschrieben ist. Die Booleschen Strukturen (der imaginären, zugehörigen Langform) sind alle atomar. Einziges Prädikatensymbol ist die Gleichheit, ansonsten treten nur Terme auf. Die Gleichheit wird durch Angabe der verglichenen Terme visualisiert, in diesem Fall sind dies Terme mit dem obersten Funktionssymbol $zelle$ (Pyramide)

und Mengen von Ableitungen. Da eine Angabe aller Ableitungsbäume aber zu aufwendig und unübersichtlich ist[1], stellen wir nur die jeweilige im Teilbeweise verwendete Ableitung durch einen Ableitungsbaum dar.

Der Beweis zeigt, wie eine vom Algorithmus berechnete Zelle zu Ableitungen korrespondiert. Durch Anwenden einiger elementarer Definitionen und der Induktionsvoraussetzungen werden diese Ableitungen konstruiert. Da eine ausführliche Erläuterung der Animation hier zu viel Raum einnehmen würde, konzentrieren wir uns auf den wichtigsten Teil, den Teilbeweis im Induktionsschluß. Siehe Anhang B.1 für den kompletten Beweis. Abbildung 7.1 zeigt einen Ausschnitt aus der Animation für den inneren Teil des Induktionsschlusses.

$$
\begin{array}{ll}
& \text{Für jedes } 1 \leq m < i \quad \text{und jede Produktion } X \to YZ \in P \text{ gilt:} \\
& Y \in zelle(m,j), Z \in zelle(i-m, j+m) \\
\equiv & \{\ m, i-m \leq n-i+1 \text{ und Induktionsvoraussetzung } \} \\
& Y \Rightarrow^* w_j \ldots w_{j+m-1},\ Z \Rightarrow^* w_{j+m} \ldots w_{j+i-1} \\
\equiv & \{\ \text{Def. von } \Rightarrow \text{ und } X \to YZ\ \} \\
& X \Rightarrow^1 YZ \Rightarrow^* w_j \ldots w_{j+m-1} Z \Rightarrow^* w_j \ldots w_{j+m-1} w_{j+m} \ldots w_{j+i-1} \\
\equiv & \{\ \text{Def. von } \Rightarrow^*\ \} \\
& X \Rightarrow^* w_j \ldots w_{j+i-1}
\end{array}
$$

Dies ist ein Beweis für $Y \in zelle(m,j), Z \in zelle(i-m, j+m) \equiv X \Rightarrow^* w_j \ldots w_{j+i-1}$ für $zelle(i,j)$ und alle Produktionen $X \to YZ$.[2] Die Nichtterminalzeichen von $zelle(i,j)$ werden während der Animation vor Erreichen des Teilbeweises an entsprechender Stelle in die Pyramide eingesetzt. Noch nicht betrachtete Zellen sind leer. Siehe Abbildung 7.1(a). Der Teilbeweis veranschaulicht die Beziehung zwischen den Nichtterminalzeichen von $zelle(i,j)$, $zelle(m,j)$ und $zelle(i-m, j+m)$ und den zugehörigen Ableitungen der entsprechenden Teilwörter.

Die im Teilbeweis gerade betrachtete Produktion wird in der Auflistung der Produktionen farblich markiert (nicht in der Abbildung gezeigt). Die Aussage

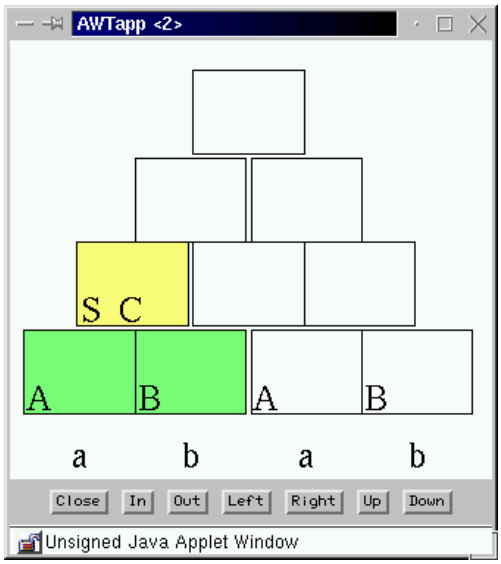
$$Y \in zelle(m,j), Z \in zelle(i-m, j+m)$$

wird nicht explizit visualisiert, da man direkt sehen kann, daß die entsprechenden Symbole in der Zelle vorhanden sind. Es wäre aber möglich, die betrachteten Nichtterminalzeichen noch zusätzlich hervorzuheben. Die Werte von m, j und $i-m, j+m$ werden wie bei der Algorithmenanimation durch farbliches Hervorheben der zugehörigen Zellen visualisiert: Gelb für $zelle(i,j)$ und Grün für die beiden anderen.

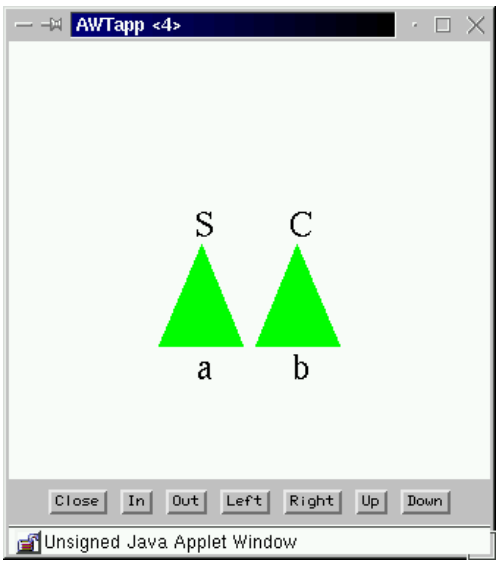
Die Induktionsvoraussetzung führt zwei vorher berechnete Ableitungen der betrachtete Teilwörter ein. Sie werden in der Animation bei Anwendung der Induktionsvoraussetzung jeweils als abstrakter Syntaxbaum dargestellt, siehe Abbildung 7.1(b).

[1] Zu einer Zelle gehört eine Menge von Ableitungen, zur Pyramide also eine Menge von Mengen von Ableitungen.

[2] Man beachte, daß X, Y und Z quantifizierte Variablen sind und nicht etwa Nichtterminalzeichen aus der Domäne des Modells.



(a) Darstellung der Funktion $zelle$ als Pyramide



(b) Darstellung der Ableitung $Y \Rightarrow^* w_j \dots w_{j+m-1}$, $Z \Rightarrow^* w_{j+m} \dots w_{j+i-1}$ als abstrakte Syntaxbäume

Abbildung 7.1: Pyramidendarstellung (a) und zu grünen Zellen gehörige Syntaxbäume (b)

Die Zugehörigkeit eines Teilbaums zu einer Zelle wird durch die grüne Farbe und die Stellung veranschaulicht. Aus diesen beiden Ableitungen und der betrachteten Produktion wird unter Zuhilfenahme der Definition der Ableitbarkeitsrelation $\Rightarrow^*$ die Ableitung $X \Rightarrow^* w_j \dots w_{j+i-1}$ konstruiert. Dieser Konstruktionsprozeß wird durch eine Animation veranschaulicht. Abbildung 7.2(a) zeigt den Anfang der Animationssequenz und Abbildung 7.2(b) das Ende. Die Animation schiebt die zu den Teilbäumen gehörigen Teilwörter am unteren Ende des neuen Syntaxbaums zum gerade betrachteten Teilwort zusammen, und die inneren Details werden ausgeblendet. Das Resultat ist eine abstrakte Visualisierung von $X \Rightarrow^* w_j \dots w_{j+i-1}$, also dem Ergebnis der Regelanwendung durch den resultierenden abstrakten Syntaxbaum visualisiert. Um den Zusammenhang mit der zugehörigen Zelle deutlich zu machen, wird der Syntaxbaum ebenfalls gelb markiert.

Der gesamte Ablauf der Animation wird durch die Grobstruktur des Beweises bestimmt. Die beiden äußeren Allquantoren des Beweises werden durch eine Eingabe realisiert: Der Benutzer kann vor Ablauf der Animation analog zur Algorithmenanimation selbst eine Grammatik und ein Wort wählen, anhand deren der Beweis veranschaulicht wird. Die Induktion über i und die Allquantifizierung von j und m wird durch eine (endliche) Aufzählung der Werte realisiert. Der sich so ergebene Ablauf der Animation entspricht dem Verlauf der Algorithmenanimation. Dies ist nicht zufällig, da die Induktion den auf dynamischem Programmieren beruhenden Kern des Cocke-Kasami-

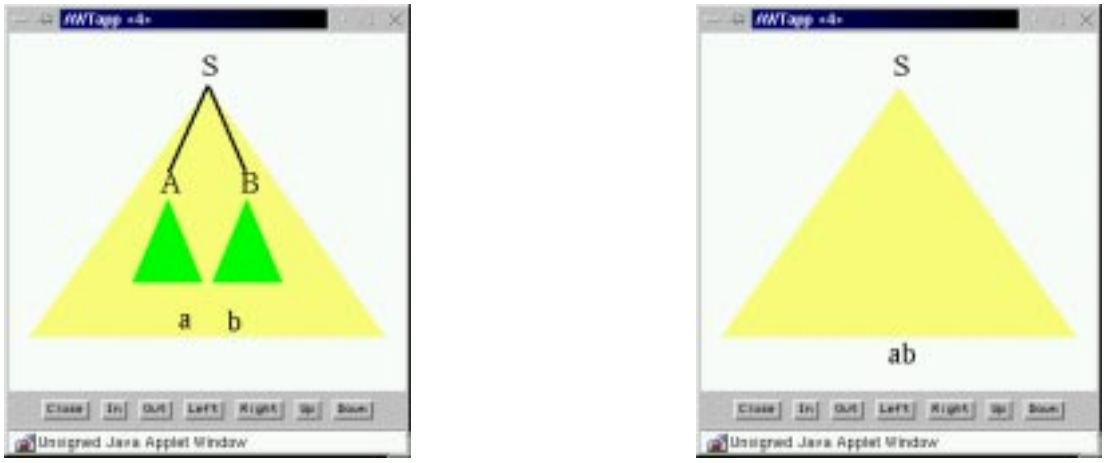


Abbildung 7.2: Bildschirmausschnitte aus der Beweisanimation zum CKY-Algorithmus: Beginn (a) und Ende (b) der Animation, welche die Konstruktion des Syntaxbaums zu $X \Rightarrow^* w_j \ldots w_{j+i-1}$ zeigt.

Younger-Algorithmus reflektiert und die Allquantifizierungen den inneren Schleifen des Algorithmus entsprechen.

So wie die Algorithmenanimation das dynamische Verhalten des CKY-Algorithmus besser veranschaulicht, werden mit der Animation des Korrektheitsbeweises die Konstruktion der Ableitungen und deren Zusammenhang mit der berechneten Funktion $zelle$ anschaulicher vermittelt als es mit einer rein statischen, graphischen Darstellung möglich wäre.

7.2 Turingmaschinen

Im folgenden präsentieren wir eine Re-Implementierung mit SCAPA unserer *Ad-hoc*-Implementierung einer Visualisierung eines Äquivalenzbeweises zweier verschiedener Typen von Turingmaschinen [PAPE und SCHMITT 1997].

Der Beweis kann auch als Reduktionsbeweis betrachtet werden, bei der Turingmaschinen mit zweiseitig unendlichem Band auf Turingmaschinen mit einseitig unendlichem Band reduziert werden, so daß beide Turingmaschinen die gleiche Funktion berechnen. Im Sinne des in Kapitel 3 vorgestellten Konzepts handelt es sich bei den Turingmaschinen um die „Probleme“, die Lösung eines solchen Problems ist die Berechnung einer Funktion, das heißt die Ausführung der Turingmaschine. Die Reduktionsabbildung übersetzt — ähnlich einem Compiler für herkömmliche Programmiersprachen — eine Turingmaschine mit zweiseitig unendlichem Band in eine Turingmaschine mit einseitig unendlichem Band, so daß die Semantik der Turingmaschinen erhalten bleibt. Siehe Abbildung 7.3.

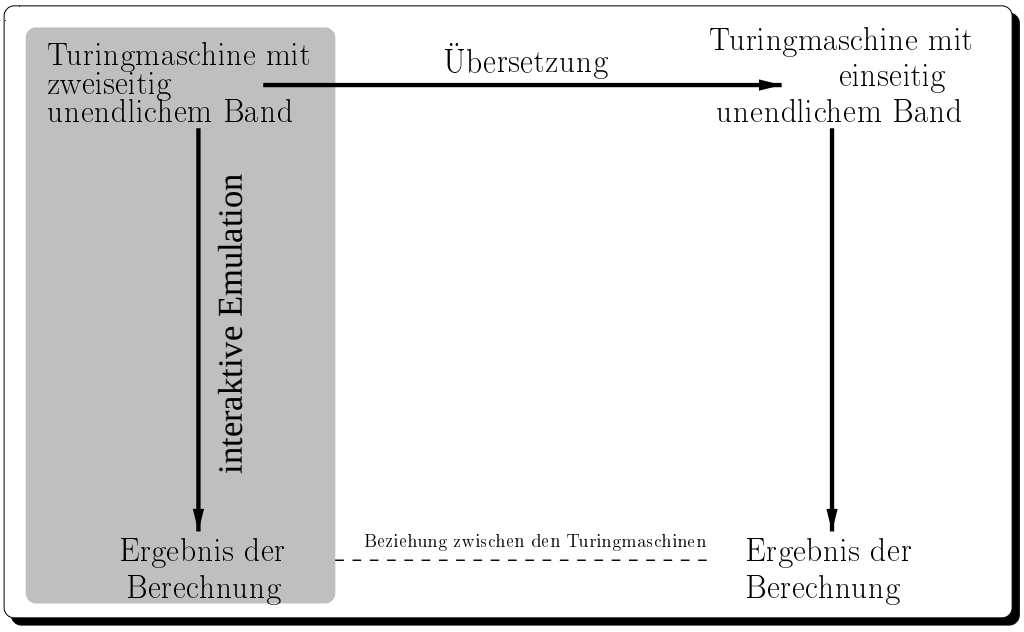


Abbildung 7.3: Reduktion einer Turingmaschine mit zweiseitig unendlichem Band auf eine Turingmaschine mit einseitig unendlichem Band

Die in Kapitel 3 vorgestellten Konzepte können zur visuellen und interaktiven Vermittlung dieses Sachverhalts eingesetzt werden. Im Hypertext zur Vorlesung können die Studierenden anhand einer Java-Emulation die Funktionsweise einer Turingmaschine mit zweiseitig unendlichem Band erfahren. Siehe Abschnitt A.2 für die Definition von Turingmaschinen. und Abbschnitt 3.4.1 für eine kurze Erklärung der Emulation.

Turingmaschinen mit einseitig unendlichen Band unterscheiden sich von Turingmaschinen mit zweiseitig unendlichen Band in zweierlei Hinsicht:

1. Ihr Band ist nach links beschränkt, besitzt also einen Anfang, und ist lediglich nach rechts hin unbeschränkt. Das linke Ende wird durch ein spezielles, sonst nicht vorkommendes Bandsymbol markiert. Wir benutzten dazu ►.

2. Wir vereinbaren, daß die Turingmaschine anhält, wenn der Schreib-/Lesekopf über das linke Ende hinauswandert.

Turingmaschinen mit einseitig unendlichem Band werden in der Vorlesung eingeführt, um zu zeigen, daß Variationen von Turingmaschinen (es gibt noch weitere) sich nicht auf die Menge der von ihnen berechenbaren Funktionen auswirkt. Ein Studierender könnte zuerst glauben, daß mit Turingmaschinen mit einseitig unendlichem Band weniger Funktionen berechnet werden können. Um diesen möglichen Eindruck zu widerlegen, wird bewiesen, daß beide Typen von Turingmaschinen die gleiche Menge

von Funktionen berechnet. Die Implikation von einseitig unendlichen zu zweiseitig unendlichen Turingmaschinen ist intuitive und technisch einfacher als die umgekehrte Richtung, die wir im folgenden betrachten werden.

Die grundlegende Idee basiert auf einer schrittweisen Simulation einer zweiseitig unendlichen Turingmaschine auf einer einseitig unendlichen: Wir stellen uns die Zellen des einseitigen Bandes von links nach rechts numeriert vor. Eine imaginäre Mitte auf dem zweiseitig unendlichen Band wird fixiert und die Zellen auf deren rechter Seite werden fortlaufend von der Mitte aus auf die geraden Zellen des einseiten Bandes und die Zellen der linken Seite werden auf die ungeraden Zellen abgebildet. Nennen wir diese Abbildung sim. Dann wird eine Transformation (ein „Compiler“) angegeben, welche die Zustandübergangsfunktion δ einer zweiseitig unendlichen Turingmaschine M auf eine Übergangsfunktion δ' einer einseitig unendlichen Turingmaschine M' abbildet, so daß M' die Turingmaschine M simuliert. Bei der Transformation werden Zustände und Symbole von M umbenannt, und es kommen einige neue hinzu, um zum Beispiel auf dem einseitigen Band das linke Ende zu markieren, damit der Kopf nicht über das Ende hinaus läuft. Bewiesen wird folgendes „Simulationslemma“.

Lemma 6 *K_n sei das Band der zweiseitig unendlichen Turingmaschine M nach n Schritten und K^1_m das einseitig unendliche Band der simulierenden Turingmaschine M nach m Schritten. Dann gilt:*

$$\exists m \in \mathbb{N} : m \geq n \wedge \mathit{sim}(K_n) = K^1_m$$

Die Domäne des prädikatenlogischen Modells des Theorems enthält beiden Typen von Turingmaschinen, diese bestehen jeweils aus

1. einer Menge Γ von Bandsymbolen,
2. dem Band, einer Folge von Bandsymbolen (zweiseitig oder einseitig unendlich)
3. einer Menge Q von Zuständen, einer Menge $F \subseteq Q$ von Endzuständen,
4. der Zustandsübergangsfunktion $\delta : \Gamma \times Q \to \Gamma \times Q \times \{L, R, U\}$,
5. der Position des Schreib-/Lesekopfes auf dem Band und
6. dem aktuellem Zustand q der Turingmaschine.

Wir visualisieren diese Elemente mit konkreten Darstellungen wie folgt:

1. Die Bandsymbole textuell als beliebige Zeichenketten. Das spezielle Bandsymbol ▶ wird durch das Zeichen $>$ visualisiert.
2. Das Band durch einen beschränkten (wählbaren) Ausschnitt um den Schreib-/Lesekopf herum. Jede Zelle des Bandes wird durch ein Rechteck markiert, in dem das Bandsymbol eingetragen ist. Zusätzlich werden Farben benutzt, um den Zusammenhang zwischen den geraden und ungeraden Zellen des einseitig unendlichen Bandes und den Zellen auf der linken und rechten Seite der imaginären Mitte des zweiseitig unendlichen Bandes hervorzuheben.

3. Die Zustände textuell als beliebige Zeichenfolgen.

4. Die Zustandsübergangsfunktion wird nicht explizit visualisiert, es wird lediglich der jeweils folgende Zustandsübergang textuell angezeigt. Dies ist für das Verständnis des Beweises ausreichend.

5. Die Position des Schreib-/Lesekopfes wird durch ein blaues Rechteck um die zugehörige Zelle visualisiert.

6. Der aktuelle Zustand ist Teil der Visualisierung des nächsten Zustandsübergangs.

Der Beweis basiert auf vollständiger Induktion nach der Anzahl n von Ausführungsschritten von M. Er enthält drei ineinander geschachtelte Fallunterscheidungen für die verschiedenen Bewegungen des Schreib-/Lesekopfes, welche selbst je noch einmal zwei Fälle enthalten, nämlich daß der Kopf sich links oder rechts der imaginären Mitte befindet. Dazu kommen dann im Fall einer Bewegung über die Mitte hinweg noch jeweils ein weiterer dazu. Insgesamt also acht zu betrachtende Fälle.

Im Beweis werden die einzelnen Simulationsschritte und der Zusammenhang zwischen den beiden Bändern detailliert nachvollzogen. Obwohl die einzelnen Beweisschritte im wesentlichen nur auf der Definition der Tranformation basieren, lassen die Fallunterscheidungen den Beweis stark wachsen. Die vollständigen Definitionen und der Beweis finden sich in Anhang B.2. Die Hypertextdarstellung des Beweises bietet hier den Studierenden die Möglichkeit, sich erst eine grobe Sicht des Beweises anzuschauen, ohne die inneren Fallunterscheidungen zu betrachten. Die Animation zum Beweis zeigt die einzelnen Simulationsschritte. Der Studierende kann sich während des Beweises visuell von der „Gleichheit“ der beiden Bänder überzeugen. Falls ein Teilbeweis verdeckt wird, zum Beispiel der Fall, daß der Schreib-/Lesekopf sich nach Links bewegt, dann zeigt die Animation die zugehörigen zwei bis drei Simulationsschritte nicht, sondern lediglich das resultierende Band. Die Animation paßt sich so den individuellen Bedürfnissen eines Studenten an.

Dieses Beispiel zeigt, daß in der Hypertextdarstellung durch die verdeckbaren Teilbeweise auch größere Beweise noch lesbar sind. Der vollständige Beweis nimmt ca. fünf Papierseiten ein. Die meisten Fälle im Beweis werden auf analoge Weise geführt. In einer Vorlesung oder einem Lehrbuch würden nur zwei oder drei der insgesamt acht Fällen wirklich betrachtet. Die Auswahl liegt dabei in den Händen des Dozenten oder Autors. Dahingegen erlaubt unsere Beweisanimation dem Studierende selbst, die ihn interessierenden Fälle mit Hilfe der Visualisierung genau zu untersuchen.

7.3 Erfahrungen

Zum Abschluß berichten wir noch kurz über unsere Erfahrungen über den Zeitaufwand, den das Schreiben einer Beweisanimation verursacht, und wie sich die Entwicklung einer Beweisanimation auf die Gestaltung des Beweistextes auswirken kann.

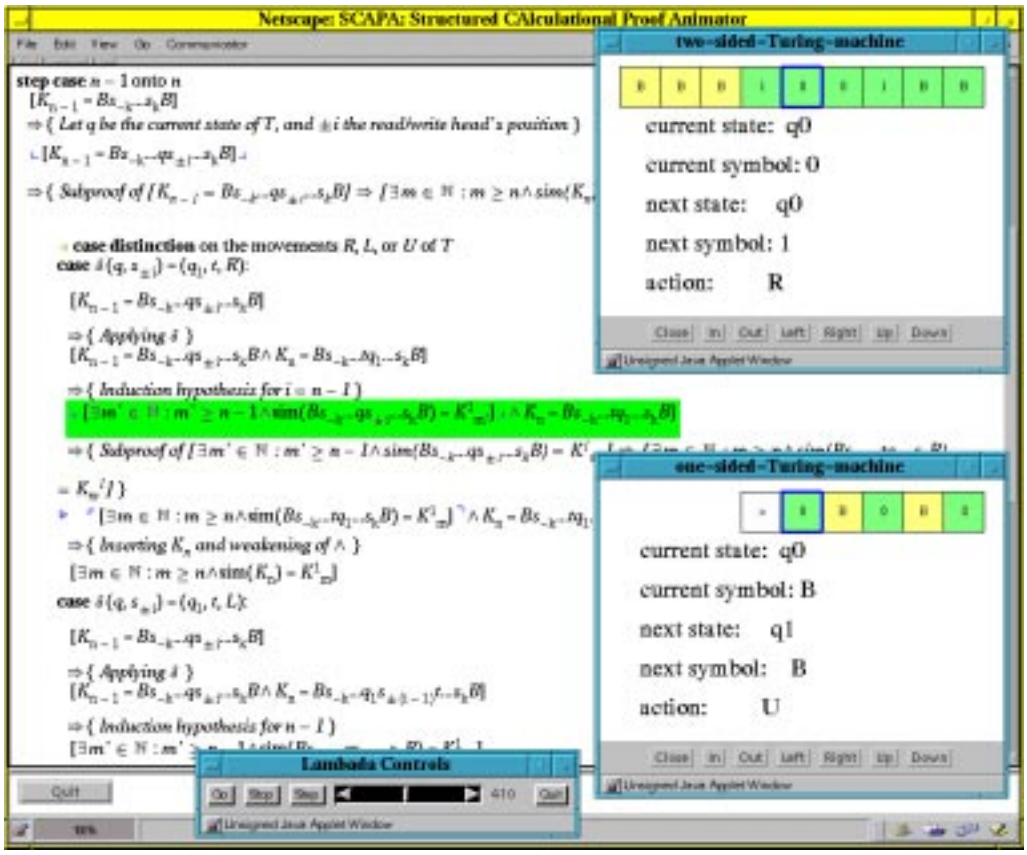


Abbildung 7.4: Bildschirmphoto einer Beweisanimation zur Äquivalenz zweier Typen von Turingmaschinen

Der Zeitaufwand hängt im wesentlichen von der Größe des Beweises (Zeitaufwand zum Schreiben des Beweises) und von der Anzahl verschiedener Beweisobjekte ab (Zeitaufwand zum Implementieren dieser Objekte und deren Animation). Da unsere Beweise stärker formalisiert und strukturiert sind als herkömmliche Beweistexte, dauert das Schreiben des Beweises auch länger. Bei unseren beiden Beispielen lag jeweils ein umgangssprachlich formulierter Beweis vor. Das Umschreiben in das *Calculational proof format* hat etwa ein (CKY-Korrektheitsbeweis) bis zwei (Turingmaschine) Tage gedauert. Es ist anzunehmen, daß mit einer zunehmenden Gewöhnung an das spezielle Beweisformat, dieser Zeitaufwand etwas reduziert werden kann. Den zum Beweis gehörigen Algorithmus anschließend in Java zu implementieren, hat in den Beispielen weniger Zeit gekostet als von uns erwartet: etwas über einen Tag. Das liegt zum einen daran, daß die zu implementierenden Datenstrukturen sehr einfach sind, und zum anderen, daß die Zeitkomplexität der Algorithmen keine große Rolle spielt (die Eingabeinstanzen sind klein). Die Implementierung der Animation hingegen ist recht aufwendig. Qualität und Zeitaufwand hängen stark vom verwendeten Animati-

onssystem ab. Das von uns verwendete LAMBADA stellt keine allzu mächtigen Animationsbefehle zur Verfügung. Wir haben ungefähr zwei bis drei Tage benötigt, um die optisch eher einfachen Animationen zu implementieren.

Die Entwicklung einer Beweisanimation hat sich positiv auf den Beweistext ausgewirkt: Zum einen erschwert die feinere Granularität des verwendeten Beweisformats es, wesentliche Beweisschritte auszulassen und zum anderen deckt die anschließende Implementierung oftmals noch unbemerkte Fehler im Beweistext auf. Der im Vorlesungsskriptum enthaltene Korrektheitsbeweis des CKY-Algorithmus enthielt im Induktionsanfang zum Beispiel keinen Hinweis auf die wesentliche Voraussetzung, daß die Produktionen in Chomsky-Normalform (also terminierend sind). Desweiteren konnten wir bei der Implementierung des Beweises zur Äquivalenz zweier Turingmaschinen viele falsche Indizierungen im Beweistext entfernen. Insgesamt werden die Beweistexte also korrekter. Das Schreiben eines Beweises mit dem Ziel einer anschließenden Beweisanimation hatte auch Auswirkung auf die strukturelle Gestaltung der Beweistexte. Zum Beispiel haben wir im Korrektheitsbeweis des CKY-Algorithmus bewußt im Induktionsanfang und -schluß einen Teilbeweis eingeführt, um den Kern der Argumentation auf eine terminierende Produktion und die zugehörige Ableitung zu isolieren und sie so leichter visualisieren zu können. In der Version im Skriptum wurde lediglich eine Gleichheitsbeziehung zwischen der *Menge* aller terminierenden Produktionen und der zugehörigen *Menge* von Ableitungen benutzt.

Kapitel 8

Taxonomie von Beweisvisualisierungen

In [HARRISON 1996] wird untersucht, wie in interaktiven Beweissystemen maschinenlesbare Beweise dem Menschen zur Eingabe, Modifikation und Überprüfung präsentiert werden. Dabei geht es weniger um die Darstellung eines existierenden Beweises, sondern hauptsächlich um dessen rechnergestützte Konstruktion durch den Menschen. Harrison beschreibt folgende sechs Kriterien, nach denen sich Beweisdarstellungen in interaktiven Theorembeweisern klassifizieren lassen:

1. Automatisierungsgrad: In welchem Maße wird der Benutzer vom System bei der Beweissuche unterstützt?

2. Benutzerinteraktion: In welchem Maße muß der Benutzer dem System Anweisungen erteilen?

3. Erweiterbarkeit: In welchem Maße kann der Benutzer die Darstellung des Beweises modifizieren?

4. Beweisstil: Werden Beweise eher deklarativ oder prozedural notiert?

5. Beweisrichtung: Welche Beweisformen, wie Widerspruchsbeweise oder direkte Beweise, unterstützt das System?

6. Verarbeitungsparadigma: Werden Beweise interaktiv oder Stapel-orientiert[1] verarbeitet?

Er wendet Kriterien 1 (niedrig, hoch) und 4 (prozedural, deklarativ) auf einige existierenden Systeme an. Diese Einteilung ist sehr grob und reflektiert den Schwerpunkt dieser Systeme auf das möglichst automatische *Finden* und *Überprüfen* eines Beweises und nicht auf dessen verständliche Darstellung für einen menschlichen Betrachter.

[1] Engl.: *batch oriented*

Die Stärken und Schwächen eines Beweisstils werden nach folgenden Gesichtspunkten konkret anhand einiger Systeme und abstrakter anhand einer Unterscheidung nach prozeduralen oder deklarativen Beweisstilen erörtert:

1. *Schreibbarkeit*: Wie aufwendig ist es, dem System einen korrekten Beweis einer Aussage einzugeben?
2. *Lesbarkeit*: Werden die Beweise dem Menschen verständlich präsentiert?
3. *Wartbarkeit*: Können Beweise leicht wiederverwendet oder adaptiert werden?
4. *Verarbeitungsgeschwindigkeit*: Kann der Beweis vom System effizient verarbeitet werden?

Weitere Klassifikationsschemata finden sich zum Thema Beweisvisualisierung derzeit im Gegensatz zu Softwarevisualisierungssystemen nicht. Da die Taxonomie von Harrison zu grob ist und lediglich einschränkend den Bereich formaler Beweise abdeckt, entwickeln wir auf Basis der Taxonomie von Softwarevisualisierungssystemen nach [PRICE et al. 1998] eine eigene Klassifikation von Beweisvisualisierungsystemen. Da das Schema von [PRICE et al. 1998] sich in den meisten Teilen nicht konkret auf Software bezieht, sondern allgemeine Kriterien von Visualisierungssystemen umfaßt, sind nur geringfügige Modifikationen nötig.

Die Taxonomie für Softwarevisualisierungssysteme nach [PRICE et al. 1998] besteht auf der ersten Ebenen aus sechs Kategorien:

A. *Anwendungsbereich*[2]: Welche Programme können mit dem Softwarevisualisierungssystem (SV-System) visualisiert werden?
B. *Inhalt*: Welche Aspekte eines Programmes werden von dem SV-System visualisiert?
C. *Form*: Was sind die Merkmale der erzeugten Visualisierung?
D. *Methode*: Wie wird die Visualisierung spezifiziert?
E. *Interaktion*: Wie interagiert der Benutzer mit dem SV-System?
F. *Wirksamkeit*[3]: Wie gut übermittelt das SV-System Informationen dem Benutzer?

Diese Unterscheidung in sechs oberste Kategorien basiert auf einem in der Softwarevisualisierung weit verbreiteten Modell, bei der zu jeder Kategorie verschiedene Personen beigetragen haben: Der *Programmierer* produziert den Algorithmus oder das Programm, welches gemäß dem Anwendungsbereich des Systems visualisiert werden kann (A: Anwendungsbereich). Der *Entwickler des Softwarevisualisierungssystems* bestimmt, welche Aspekte des Algorithmus oder Programms visualisiert werden

[2] Engl.: *scope*
[3] Engl.: *effectiveness*

können (B: Inhalt). Das *Softwarevisualisierungssystem* produziert eine visuelle Ausgabe mit bestimmten Merkmalen (C: Form). Der *Visualisierer* spezifiziert die Visualisierung (D: Methode). Der *Benutzer* interagiert mit dem System, welches bestimmte Auswirkungen auf ihn hat (E: Interaktion, F: Wirksamkeit).

Wir stellen im folgenden diese Taxonomie für den Bereich Softwarevisualisierung vor (jeweils in einem eigenen Unterabschnitt) und adaptieren sie zu einer Taxonomie von Beweisvisualisierungssystemen (ebenfalls ein eigener Unterabschnitt). Da sehr viele Merkmale, wie zum Beispiel *Interaktion* und *Wirksamkeit*, der Taxonomie unabhängig vom Visualisierungbereich sind, ergeben sich nur marginale Unterschiede. Falls sich eine Kategorie nicht oder nur geringfügig ändert, verzichten wir auf eine genauere Beschreibung für den Bereich Softwarevisualisierung. Basis der Adaption wird, wie bei der Begriffsbildung im Bereich Beweisvisualisierung, die Analogie Algorithmus und nicht formaler Beweis, Programm und formaler Beweis, Daten und Formeln sein.

8.1 A: Anwendungsbereich

8.1.1 Softwarevisualisierung

Der Anwendungsbereich umfaßt alle Merkmale eines Softwarevisualisierungssystems, welche für den Programmierer des zu visualisierenden Programms wichtig sind.

A.1. *Allgemeine Anwendbarkeit*[4]: Kann das System einen allgemeinen Bereich von Programmen verarbeiten oder stellt es nur eine feste Menge von Beispielen dar?

- A.1.1 *Hardware*: Auf welcher Hardware läuft das System?
- A.1.2 *Betriebssystem*: Welches Betriebssystem wird benötigt?
- A.1.3 *Sprache*: In welcher Programmiersprache müssen die Programm geschrieben sein?
  - A.1.3.1. *Nebenläufigkeit*[5]: Unterstützt das System Nebenläufigkeiten der Programmiersprache (falls vorhanden)?
- A.1.4 *Anwendung*[6]: Welche Einschränkungen in der Art der zu visualisierenden Programme gibt es?
  - A.1.4.1. *Spezialgebiet*: Welche Programme können besonders gut mit dem System visualisiert werden?

A.2. *Skalierbarkeit*: In welchem Maße verarbeitet das System große Programme?

- A.2.1 *Programm*: Was ist das größte verarbeitbare Programm?
- A.2.2 *Datenmenge*: Was ist die größte verarbeitbare Datenmenge?

[4]Engl.: *generality*
[5]Engl.: *cuncurrency*
[6]Engl.: *application*

8.1.2 Beweisvisualisierung

Hardware und Betriebssystem sind allgemeine Merkmale eines jeden Systems und bleiben deswegen unverändert. Die anderen Punkte beziehen sich auf Programme. Da wir Beweise statt Programme visualisieren wollen, ist zu untersuchen, ob die Klassifizierungsmerkmale auch für Beweise sinnvoll sind, wenn wir Programm durch Beweis ersetzen. Dabei sollten möglichst alle wichtigen Merkmale des Systems aus Sicht des *Autors* eines Beweises erfaßt werden. Insbesondere gehen wir davon aus, daß ein Beweis nicht erst gefunden werden muß, sondern schon vorliegt. Die wichtigsten Merkmale eines Beweises sind die Logik, in welchem die bewiesene Aussage formuliert ist, und der Beweiskalkül, mit dem die Aussage bewiesen wurde. Nebenläufigkeit ist bei der Darstellung eines gefundenen Beweises nicht von Bedeutung; deswegen lassen wir diesen Punkt im Klassifikationsschema weg.[7] Die „Daten“ eines Beweises sind die in ihm auftretenden Variablen, Terme und Formeln bzw. logischen Aussagen. Wir werden im weiteren Verlauf unserer Erörterungen bei Beweisen allgemein den Begriff Formeln statt Daten benutzten. Dies soll Variablen und Terme immer mit einbeziehen. In Theorembeweisern ist die Größe der verarbeitbaren Beweise und Formeln normalerweise nicht beschränkt. Im Ausbildungsbereich macht es oftmals allerdings Sinn, Beschränkungen einzuführen, da mit der anwachsenden Größe von Beweisen und Formeln auch die Menge der visuellen Darstellungen zunimmt und damit unübersichtlich werden kann. Der Anwendungsbereich – bezogen auf Beweisvisualisierung – stellt sich dann wie folgt dar (siehe auch Abbildung 8.1).

A.1. *Allgemeine Anwendbarkeit*: Kann das System einen allgemeinen Bereich von Beweisen verarbeiten oder stellt es nur eine feste Menge von Beispielen dar?

- A.1.1 *Hardware*: Auf welcher Hardware läuft das System?
- A.1.2 *Betriebssystem*: Welches Betriebssystem wird benötigt?
- A.1.3 *Sprache*: In welcher Logik und in welchem Kalkül sind die Beweise geschrieben? Ist es ein deklarativer oder prozeduraler Beweis?
- A.1.4 *Anwendung*: Welche Einschränkungen auf die Art der zu visualisierenden Beweise gibt es?
  - A.1.4.1. *Spezialgebiet*: Welche Beweise können besonders gut mit dem System visualisiert werden?

A.2. *Skalierbarkeit*: In welchem Maße verarbeitet das System große Beispiele?

- A.2.1 *Beweis*: Was ist der größte verarbeitbare Beweis?
- A.2.2 *Datenmenge*: Was ist die größte verarbeitbare Datenmenge (Variablen, Terme, Formeln)?

[7] Bei interaktiven Theorembeweisen kann Nebenläufigkeit jedoch in Form von Indeterminismen des Kalküls oder bei paralleler Beweissuche auftreten.

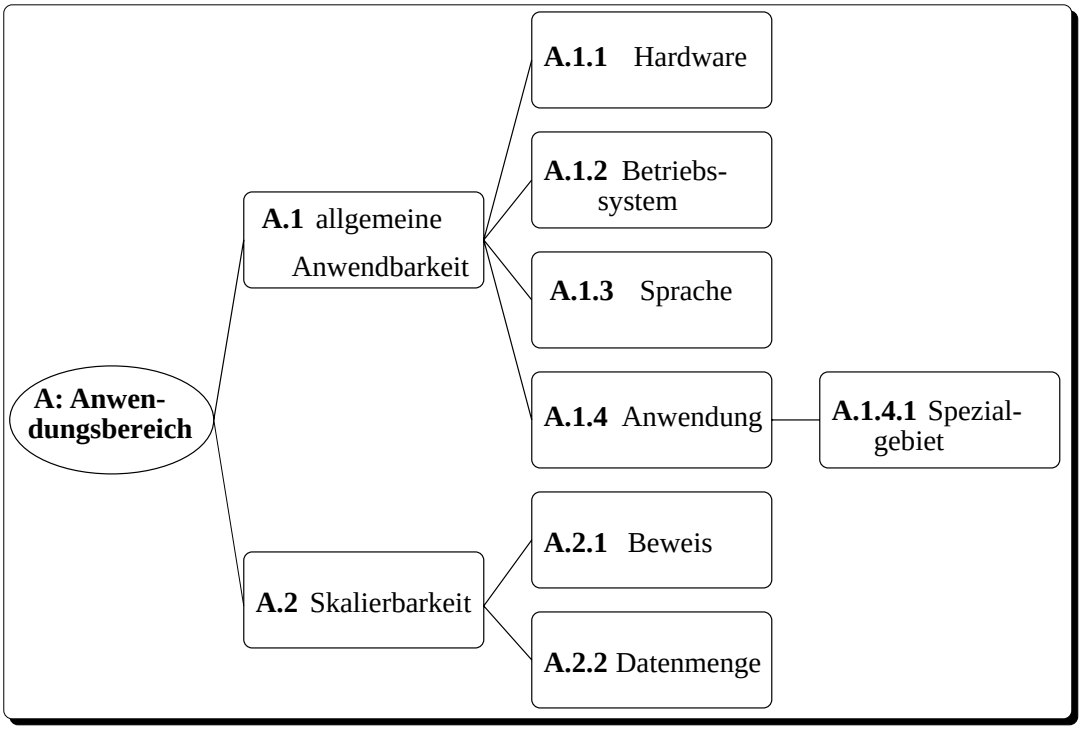


Abbildung 8.1: Hierarchie zum Anwendungsbereich

8.2 B: Inhalt

8.2.1 Softwarevisualisierung

Die beiden wesentlichen Teile einer Software sind das implementierte *Programm* (der Quelltext) und eine abstrakte Beschreibung der Software durch einen *Algorithmus*. Manche Systeme sind flexibel genug, beide Aspekte darzustellen.

B.1. *Programm*: In welchem Maße visualisiert das System das implementierte Programm?

B.1.1 *Kode*: In welchem Maße visualisiert das System die Anweisungen des Quellprogramms[8]?

B.1.1.1. *Ablauffluß*: In welchem Maße visualisiert das System den Ablauf der Instruktionen des Quellprogramms?

B.1.2 *Daten*: In welchem Maße visualisiert das System die Datenstrukturen im Quellprogramm?

B.1.2.1. *Datenfluß*: In welchem Maße visualisiert das System den Datenfluß im Quellprogramm?

[8]Engl.: *program source code*

B.2. *Algorithmus*: In welchem Maße visualisiert das System den abstrakten Algorithmus[9] des Programms?

B.2.1 *Anweisungen*: In welchem Maße visualisiert das System die Anweisungen des Algorithmus?

B.2.1.1. *Ablaufffluß*: In welchem Maße visualisiert das System den Ablauf der Anweisungen des Algorithmus?

B.2.2 *Daten*: In welchem Maße visualisiert das System die abstrakten Datenstrukturen[10] im Algorithmus?

B.2.2.1. *Datenfluß*: In welchem Maße visualisiert das System den Datenfluß im Algorithmus?

B.3. *Treue und Vollständigkeit*[11]: Repräsentieren die visuellen Darstellungen[12] das wahre und vollständige Verhalten der zugrundeliegenden virtuellen Maschine?

B.3.1 *Durchdringung*[13]: Falls das System zur Visualisierung nebenläufiger Programme benutzt werden kann, zerstört[14] es die Ausführungsreihenfolge des Programms?

B.4. *Zeitpunkt der Datenerzeugung*[15]: Werden die Daten, von denen die Visualisierung abhängt, während Übersetzungszeit oder Laufzeit erzeugt?

B.4.1 *Abbildung der Programmzeit*[16]: Was ist die Abbildung zwischen der „Programmzeit“ und der „Visualisierungszeit“?

B.4.2 *Zeitpunkt der Visualisierungserzeugung*[17]: Wird die Visualisierung nach Programmablauf mit dabei gespeicherten Daten oder „live“ während der Programmausführung erzeugt?

8.2.2 Beweisvisualisierung

Die Darstellungen des Quelltextes eines Programmes umfaßt „pretty printing“, Strukturdiagramme und Funktionsgraphen[18]. Übertragen auf formale Beweise umfaßt dieser Bereich eine strukturierte, meist baumförmige Darstellung der verwendeten *Beweisregeln* (den Beweisbaum). Sie kommt bei vielen Systemen zur interaktiven formalen

[9] Engl.: *high level algorithm*
[10] Engl.: *high level data structures*
[11] Engl.: *fidelity and completeness*
[12] Engl.: *metaphors*
[13] Engl.: *invasiveness*
[14] Engl.: *disrupt*
[15] Engl.: *data gathering time*
[16] Engl.: *temporal control mapping*
[17] Engl.: *visualization generation time*
[18] Engl.: *call tree*

Beweisführung, wie beim *Karlsruhe Interactive Verifier* [REIF et al. 1997] oder bei der graphischen Benutzungsschnittstelle des HOL-Systems [SYME 1995], aber auch bei automatischen Theorembeweisern vor, wie etwa beim Tableau-basierten Beweiser ${}_3T^AP$ [BECKERT et al. 1996]. Mittlerweile werden auch Hypertext-Darstellungen von Beweisbäumen verwendet, bei denen der Benutzer ineinander geschachtelte Teilbeweise ausblenden kann, um so adaptiv die Komplexität formaler Beweise zu reduzieren [GRUNDY und LÅNGBACKA 1997].

Der Ablaufffluß eines Programms kann durch verschiedene Formen von Flußdiagrammen statisch dargestellt werden. Ebenso gehören visuelle Repräsentationen der Rekursionstiefe in diesen Bereich oder das simples Hervorheben der gerade ausgeführten Instruktion im Quelltext. Bei Beweisen entspricht der Ablaufffluß der Reihenfolge der Regelanwendungen. Wir bezeichnen dies als *Regelfluß*. Dessen visuelle Darstellung spielt bei den derzeit existierenden Theorembeweisern so gut wie keine Rolle. Sie ist lediglich implizit in der Baumdarstellung des Beweises als lineare Folge von Regelanwendungen (Knoten) auf einem Ast gegeben. Bei Verzweigungen ist hingegen keine Reihenfolge der Äste mehr gegeben. Visuelle Darstellungen wie Flußdiagramme könnten auch für Beweise verwendet werden, indem sie strukturierende Beweisregeln wie Induktion, Teilbeweise oder Fallunterscheidungen hervorheben. Analog könnten gebräuchliche Visualisierungen für die Rekursionstiefe zur Visualisierung einer prozedural interpretierten Induktion benutzt werden. Dies hat bisher aber keine Eingang in die Beweisvisualisierung gefunden, weil der deklarative Charakter von Beweisen bei fast allen Systemen und Beweisnotationen im Vordergrund steht.

Bei den Daten eines Programmes handelt es sich um die bei einem Programmablauf erzeugten Datenstrukturen, die zusammen den aktuellen Zustand des Programms bilden. Bei einer Programmvisualisierung können zusammengesetzte Datenstrukturen auf Basis visueller Darstellungen elementarer Daten in vielfältiger Form veranschaulicht werden. Oftmals werden gerichtete oder ungerichtete Graphen zur visuellen Darstellung komplexer Datenstrukturen verwendet. Dies trifft insbesondere auf mit Zeigern realisierte Datenstrukturen zu. Die in einem Beweis auftretenden Regeln manipulieren *Formeln*, die den Daten eines Programms entsprechen. Darüberhinaus können auch freie Variablen auftreten, die den Zustandsvariablen eines Programmes entsprechen. Sie können im Verlauf eines Beweises durch Werte instantiiert werden. Ein wesentliches Charakteristikum eines Beweisvisualisierungssystems ist die visuelle Darstellung der Formeln und der in ihnen enthaltenden Variablen und Terme. Theorembeweiser begnügen sich normalerweise mit einer typographischen Darstellung von Formeln. Bei dem Lernprogramm HYPERPROOF [BARWISE und ETCHEMENDY 1998] hingegen sind zusätzlich Konstanten als geometrische Objekte sowie atomare Aussagen über deren örtliche Beziehung durch die Position der geometrischen Objekte visualisiert. Venn-Diagramme stellen zum Beispiel formale logische Aussagen über Mengen dar, die mit formalen Beweisregeln manipuliert werden.

Mit Datenfluß bezeichnet man abstrakt die Beziehung von Ein- und Ausgabedaten zwischen Objekten. Der Datenfluß repräsentiert Zwischenergebnisse, die während einer Berechnung auftreten. Der Datenfluß kann bei Programmen diagrammartig mit

einem Graphen statisch repräsentiert werden, etwa als Auswertungsbaum bei einem arithmetischen Ausdruck. Eine visuelle Sicht des Laufzeitkellers beschreibt den Datenfluß zwischen Funktionen und Prozeduren. Wir bezeichnen mit *Formelfluß* konsequenterweise die Beziehung zwischen der Prämisse und der Konklusion von Beweisregeln. Diese wird noch weniger als der Regelfluß derzeit zur visuellen Darstellung von Beweisen herangezogen. Bei einer prozeduralen Sichtweise einer Induktion, kann – ähnlich einem Laufzeitkeller – der Formelfluß, also die induktiv konstruierten Formeln, zur Ablaufzeit visuell dargestellt werden.

Die analogen Charakterisierungen der Softwarevisualisierung existieren auch für Algorithmen. Der wesentliche Unterschied besteht lediglich darin, daß die visuellen Darstellungen eines Algorithmus eine größere Abstraktionsstufe gegenüber den Details einer Programmvisualisierung besitzen. Analog lassen sich die für die Beweisvisualisierungen erläuterten Merkmale problemlos auf nicht formale Beweise übertragen. Statt von Formeln sprechen wir in diesem Fall aber lieber etwas vager von *Aussagen*.

Die visuellen Darstellungen einer Software variieren zwischen den verschiedenen Systemen. Wie gut eine visuelle Darstellung das Verhalten und den Zustand der virtuellen Maschine widerspiegelt, ist deswegen ein wichtiges Beurteilungskriterum.

> „Automatic systems . . . may produce a misleading abstraction for a data structure while visualizers using a system like TANGO may only animate a particular part of an algorithm for expository purpose.-'[PRICE et al. 1998]

Die größte *Treue und Vollständigkeit* besitzen meist Systeme, die stark mit dem Quelltext eines Programmes verbunden sind. Dieser Aspekt spielt auch bei der Charakterisierung eines Systems zur Visualisierung von Beweisen eine Rolle. Hier stellt sich die Frage, wie gut die Visualisierung die Eigenschaften des Beweiskalküls widerspiegeln.

Da *Durchdringung* bei der Softwarevisualisierung nur ein Aspekt paralleler Programmausführung ist, die auf der Ebene von Beweisen keine Entsprechung besitzt, lassen wir diesen Punkt in unserer Klassifizierung weg. Unser Klassifikationsschema für Systeme zur Beweisvisualisierung zum Bereich Inhalt stellt sich dann wie folgt dar, siehe auch Abbildung 8.2:

B.1. *Formaler Beweis*: In welchem Maße visualisiert das System den formalen Beweis?

- B.1.1 *Beweisregeln*: In welchem Maße visualisiert das System die Beweisregeln im formalen Beweis?
  - B.1.1.1. *Regelfluß*: In welchem Maße visualisiert das System den Regelfluß des formalen Beweises?
- B.1.2 *Formeln*: In welchem Maße visualisiert das System die Formeln im formalen Beweis?

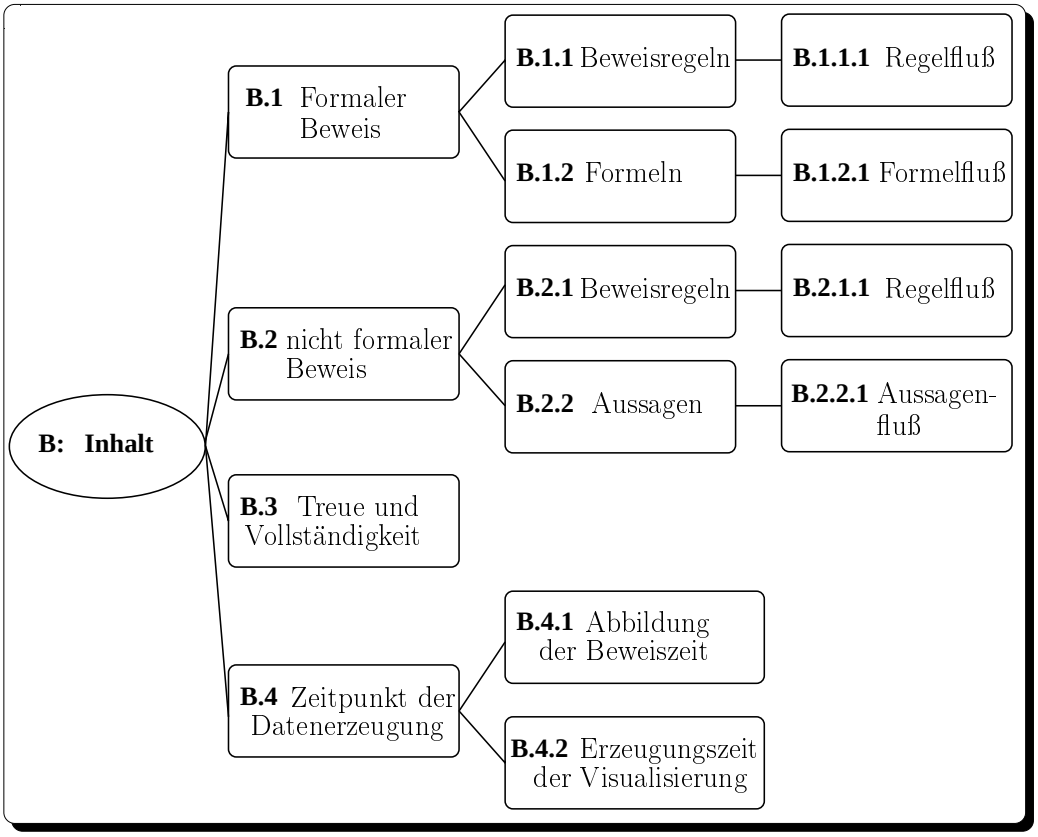


Abbildung 8.2: Hierarchie zu **Inhalt**

B.1.2.1. *Formelfluß*: In welchem Maße visualisiert das System Formelfluß im formalen Beweis? Zum Beispiel den Zustand der Daten während einer Induktion?

B.2. *Nicht formaler Beweis*: In welchem Maße visualisiert das System den nicht formalen Beweis?

B.2.1 *Beweisregeln*: In welchem Maße visualisiert das System die Beweisregeln im nicht formalen Beweis?

B.2.1.1. *Regelfluß*: In welchem Maße visualisiert das System den Regelfluß des nicht formalen Beweises?

B.2.2 *Aussagen*: In welchem Maße visualisiert das System die abstrakten Aussagen im nicht formalen Beweis?

B.2.2.1. *Aussagenfluß*: In welchem Maße visualisiert das System den Aussagenfluß im nicht formalen Beweis?

B.3. *Treue und Vollständigkeit*: Repräsentieren die visuellen Darstellungen die wahre und vollständige Bedeutung des zugrundeliegenden Kalküls?

B.4. *Zeitpunkt der Datenerzeugung*: Werden die Daten, von denen die Visualisierung abhängt, während Übersetzungszeit oder Laufzeit erzeugt?

B.4.1 *Abbildung der „Beweiszeit“*: Was ist die Abbildung zwischen „Beweiszeit“ und „Visualisierungszeit“?

B.4.2 *Erzeugungszeit der Visualisierung*: Wird die Visualisierung nach Beweisablauf mit dabei gespeicherten Daten oder „live“ während der Beweisausführung erzeugt?

8.3 C: Form

Dieser Teil der Taxonomie beschreibt die Charakteristik der Ausgabe des Visualisierungssystems. Da sie allgemeine Visualisierungsmerkmale erfaßt, kann sie ohne große Änderung übernommen werden. Wir stellen diesen Teil der Taxonomie deswegen ohne Beispiele aus der Softwarevisualisierung vor. Siehe Abbildung 8.3. Das Merkmal C.5 *Programmsynchronisation* aus der Taxonomie von [PRICE et al. 1998] beschreibt die Fähigkeit eines Softwarevisualisierungssystems, mehrere unterschiedliche Programme – geschrieben in einer gemeinsamen Programmiersprache – synchron zu visualisieren, zum Beispiel, um das Zeitverhalten verschiedener Sortiertalgorithmen zu veranschaulichen. Wir haben diesen Punkt weggelassen, weil es unserer Meinung nach selten vorkommt, daß verschiedene Beweise derselben Aussage – geschrieben in einem gemeinsamen Kalkül – parallel visualisiert werden sollen.[19]

C.1. *Medium*: Was ist das Zielmedium für die visuelle Darstellung?

C.2. *Präsentationsstil*: Wie ist das generelle Erscheinungsbild der visuellen Darstellung?

C.2.1 *Graphisches Vokabular*: Welche graphischen Elemente werden zur Produktion der Visualisierung benutzt?

C.2.1.1. *Farbe*: In welchem Maße werden Farben benutzt?

C.2.1.2. *Dimension*: In welchem Maße werden zusätzliche Dimensionen für die Visualisierung benutzt?

C.2.1.1. *Animation*: Wenn das System Daten zur Laufzeit erzeugt, in welchem Maße enthält die resultierende Visualisierung Animationen?

C.2.1.1. *Ton*: In welchem Maße verwendet das System Ton zur Übertragung von Informationen?

C.3. *Granularität*: In welchem Maße kann das System feine Details filtern?

[19] In diesem Zusammenhang denkbar ist auch eine parallele Visualisierung eines Beweises, der in verschiedenen Kalkülen ausgeführt wurde; zum Beispiel, um zwei Kalküle zu vergleichen.

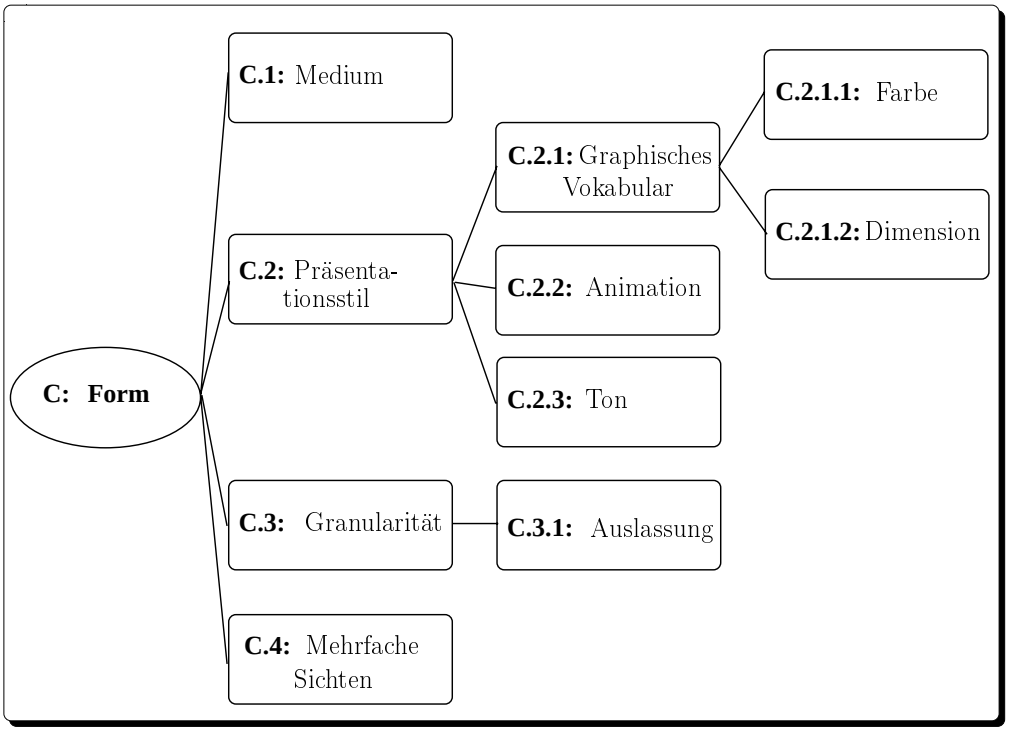


Abbildung 8.3: Hierarchie zu **Form**

C.3.1 *Auslassung*: In welchem Maße erlaubt das System Informationen auszulassen?

C.4. *Mehrfache Sichten*: In welchem Maße stellt das System mehrfache, synchronisierte Sichten dar?

Das *Zielmedium* der erzeugten Visualisierung kann vielfältig sein und Film, Video oder auch einfach nur Papier umfassen. Die gebräuchlichsten Medien bei einer Beweisvisualisierung auf dem Rechner dürften jedoch Farb- oder schwarzweiß-Bildschirme sein. Das graphische Vokabular gibt Auskunft über die graphischen Primitive (Punkte, Linien, Polygone, Farben, Texturen), aus denen die graphischen Darstellungen zusammengesetzt sind. Mit *Farbe* können bestimmte Teile einer Beweisvisualisierung hervorgehoben werden, zum Beispiel zur Markierung einer Teilformeln in einem Beweistext, die gleichzeitig in einer Animation graphisch dargestellt wird. Neben zweidimensionalen Darstellungen, wie sie bei Visualisierungen zu Beweisen in traditionellen Lehrtexten vorkommen, können moderne Rechner darüberhinaus Informationen über einen Beweis drei-dimensional darstellen. HYPERPROOF etwa stellt Konstanten in Formeln alternativ als zwei- oder drei-dimensionale geometrische Objekte dar. *Animationen* werden in Softwarevisualisierungssystemen benutzt, um die zeitlichen Aspekte bei einer Programmausführung zu erfassen. In derzeit existierenden Theorembeweisern werden Animationen bisher noch nicht zur Visualisierung formaler Beweise

herangezogen. Bei nicht formalen Beweisen sind bisher nur in einigen Ausnahmefällen dynamische Aspekte eines Beweises visualisiert worden. Zu letzteren gehört die schon erwähnte ZEUS-Animation des Satzes von Pythagoras. Statt statischer Darstellungen können Animationen von Beweisen sehr viel besser die konstruktiven Aspekte in einem Beweis visualisieren, zum Beispiel eine Regelanwendung. *Ton* wurden noch nicht für Beweisvisualisierung benutzt. Selbst im Bereich der Softwarevisualisierung sind Töne bisher rar. Töne – insbesondere gesprochener Text – könnten dazu benutzt werden, während des Ablaufs einer Beweisvisualisierung die graphischen Darstellungen zusätzlich zu erläutern.

Die *Granularität* beschreibt, welche Details eines Beweises visualisiert werden und inwieweit das System erlaubt, bestimmte Informationen gezielt *auszulassen* oder zu filtern. Theorembeweiser stellen normalerweile die Details eines Beweises selektiv dar, da der Benutzer nicht immer alle verfügbaren Informationen zur interaktiven Beweisführung benötigt. Zum Beispiel können bei ${}_3T^AP$ die Sorteninformationen bei der Darstellung von Formeln ausgeblendet werden. Strukturierte Beweisformate wie das *Structured calculational proof format* ermöglichen eine Hypertextdarstellung eines nicht formalen Beweises, bei dem sich Teilbeweise vom Benutzer gezielt ausblenden lassen.

Mehrfache Sichten erlauben die simultane visuelle Darstellung eines Beweises mit verschiedenen synchronen Sichten. Bei der Visualisierung eines Beweises können zum Beispiel Formeln auf unterschiedliche Weise visualisiert werden, etwa eine grobe von den Details abstrahierende Sicht und eine feine alle Details zeigende Sicht. Dies Merkmal wird bisher aber von keinem uns bekannten System verwendet, außer zur groben und detaillierten Sicht auf einen Beweisbaum.

8.4 D: Methode

Die *Methode* beschreibt alle wesentlichen Merkmale eines Softwarevisualisierungssystems, die der Visualisierer zur Erzeugung der Visualisierung berücksichtigt und verwendet. Die Taxonomie enthält auch hier vom Bereich der Softwarevisualisierung weitgehend unabhängige Merkmale. Wir haben deswegen auch diesen Teil fast ohne Modifikation übernommen. Wo sich etwas gegenüber dem Anwendungsbereich Software geändert hat, haben wir den ursprünglichen auf Softwarevisualisierung zielenden Teil in Klammern hinzugefügt, der Rest ist unverändert; siehe Abbildung 8.4.

D.1. *Spezifikationsstil*: Welcher Stil zur Spezifikation der Visualisierung wird verwendet?

D.1.1 *Intelligenz*: Falls die Visualisierung automatisch erstellt wird, wie fortgeschritten ist die Software aus Sicht der „Künstlichen Intelligenz“?

D.1.2 *Anpaßbarkeit*[20]: In welchem Maße kann der Benutzer die Visualisierung

[20] Engl.: *tailorability*

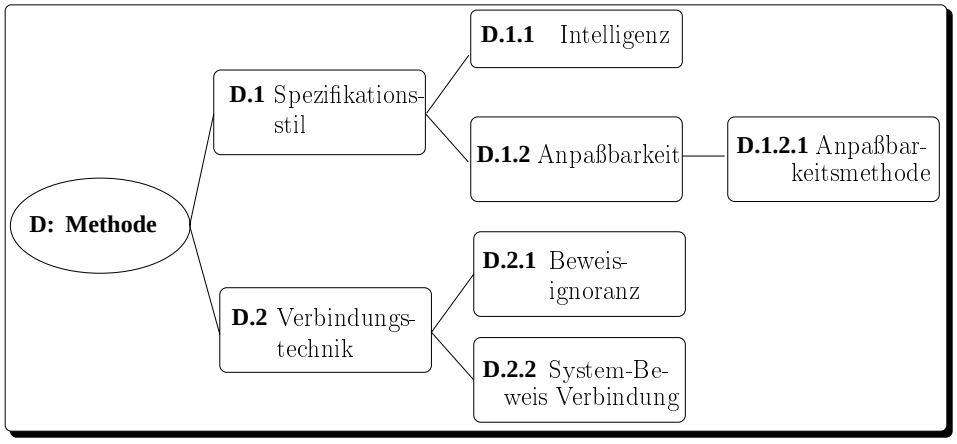


Abbildung 8.4: Hierarchie zu **Methode**

anpassen?

D.1.2.1. *Anpaßbarkeitsmethode*: Wie kann der Benutzer die Visualisierung anpassen?

D.2. *Verbindungstechnik*: Wie sind Visualisierung und Beweis (Software) verbunden?

D.2.1 *Beweisignoranz (Kodeignoranz)*[21]: Falls das System nicht vollautomatisch ist, wieviel Kenntnis muß der Visualisierer über den Beweis (Programmkode) besitzen, um die Visualisierung zu erzeugen?

D.2.2 *System-Beweis(-Kode)-Verbindung*: Wie stark ist das System mit dem Beweis (Kode) verbunden?

Der *Spezifikationsstil* umfaßt die Art und Weise, wie die Visualisierung erstellt wird. Bei den derzeit existierenden Softwarevisualisierungssystemen kann der Stil *unveränderbar* sein. Dazu gehören statische Visualisierungen in Büchern und Filme, wie beispielsweise „Sorting out Sorting“ [BAECKER und SHERMAN 1981]. Visualisierungen können *handkodiert* sein, durch Schreiben eines speziellen Programms, welches die Animation erzeugt, so wie es bei den von uns erzeugten und in Kapitel 2 vorgestellten Algorithmenanimationen der Fall ist. Bei BALSA, ZEUS und TANGO wird die Animation auf Basis spezieller Befehle aus einer *Animationsbibliothek* erstellt. Es gibt auch *automatisch* erzeugte Visualisierungen, etwa bei (J)ELIOT und bei allen Quelltext-„Debuggern“. Das Maß an „*Intelligenz*“, mit dem die Systeme dabei vorgehen, mag zwar für den Entwickler von Softwarevisualisierungswerkzeugen interessant sein, ist unserer Meinung nach für den Gebrauch des Systems durch den Visualisierer aber unwichtig. Wir führen diesen Punkt nur der Vollständigkeit halber auf. Bei machen Systemen kann der Benutzer die Visualisierung in bestimmtem

[21] Engl.: *code ignorance allowance*

Maße anpassen. So lassen sich Fenstergröße und -inhalt bei SAMBA verschieben und vergrößern. Darüberhinaus kann der Benutzer auch die Darstellung der Daten beeinflussen, etwa durch eine Spezifikation der Abbildung von Programmdaten auf visuelle Darstellungen bei Pavane. Bei den meisten „pretty printern“ läßt sich durch Angabe von Parametern die Darstellung beeinflussen. Bei Beweisvisualisierungssystemen sind dieselben Merkmale zu beobachten: So wie „pretty printing“-Programmen eine strukturierte Sicht auf den Quelltextes erlauben, erzeugen fast alle interaktiven Theorembeweiser automatisch eine hierarchische Darstellung eines formalen Beweises. Die Anpaßbarkeit durch den Benutzer beschränkt sich meist auf alternative Darstellungen von Formeln, wie zum Beispiel die Einstellung der Termtiefe oder das Ausschalten von Sorteninformationen beim automatischen Theorembeweiser ${}_3T^AP$. Eine spezielle Spezifikationssprache gibt es nicht, alternative Darstellungen lassen sich meist über die Schalter einer graphischen Benutzungsschnittstelle anwählen. Beim ILF-System hingegen kann der Benutzer über eine Spezifikationssprache die Darstellung der Formeln beschreiben. Bei den derzeit existierenden Visualisierungen nicht formaler Beweise überwiegt ein unveränderlicher oder handkodierter Stil zur Spezifikation der Visualisierung. Er ist oftmals durch das verwendete Animationssystem festgelegt, etwa durch die Verwendung von ZEUS bei der Animation des Beweises des Satzes von Pythagoras oder durch ANIMATED ALGORITHMS [GLOOR et al. 1993] bei der Visualisierung eines Korrektheitsbeweises [GLOOR und STREITZ 1990, GLOOR et al. 1992]. Lediglich PROOFVIEWS erstellt automatisch einen strukturierten Hypertext aus einem Beweis ausgeführt im *Calculational proof format*, der in HTML geschrieben und mit speziellen Strukturierungsanweisungen versehen sein muß.

Bei einem Softwarevisualisierungssystem charakterisiert die *Verbindungstechnik*, wie Programm und zugehörige Visualisierung zusammenhängen. Unsere Algorithmenanimationen sind manuell mit Ausgabebefehlen angereichert, die bei Ausführen des Programms ein Animationsskript für SAMBA erstellen. Manche Systeme wie TANGO (mit einer entsprechenden Erweiterung) stellen einen speziellen Editor zur Verfügung, mit dem die interessanten Ereignisse im Quelltext markiert (annotiert) werden, ohne den ursprünglichen Kode selbst zu verändern. Der so annotierte Quelltext wird dann automatisch in ein mit Animationsanweisungen angereicherten Kode transformiert, der anschließend übersetzt und ausgeführt wird. (J)ELIOT annotiert den Kode automatisch. Daneben existieren noch andere Ansätze, die meist von spezifischen Eigenheiten der Quellsprache Gebrauch machen: Animus nutzt die objektorientierten Merkmale von Smalltalk aus, um die Änderung beliebiger Objekte zu überwachen und darauf mit einer Modifikation der Visualisierung zu reagieren. Aus Sicht des Visualisierers ist die *Kodeignoranz* ein wichtiger Aspekt von Softwarevisualisierungssystemen: Während bei automatischen Systemen normalerweise keine Kenntnisse des Quelltextes vorhanden sein müssen, muß der Visualisierer bei manuell modifizierten Programmen in der Regel den Kode genauer verstehen. Die Kodeignoranz ist ein wichtiges Kriterium bei Systemen, mit denen der Quelltext nicht modifiziert sondern annotiert wird. Ebenso von Bedeutung ist die Frage nach der Verbindung des Visualisie-

rungssystems mit dem Quellkode. Bei BALSA sind Quelltext und Visualisierung so stark verschränkt, daß Programm und Visualisierung in derselben Umgebung und auf dem gleichen Rechner ablaufen müssen, während unsere SAMBA-Animationen durch das Skript so vom Programm entkoppelt sind, daß die Animation auf einem anderen Rechner ablaufen kann. Werden die Skripte gespeichert, so sind Programmablauf und Visualisierung zeitlich entkoppelt. Animationssysteme wie (J)CAT und MOCHA entkoppeln Programm und Visualisierung örtlich mit Hilfe einer Klient-Server Architektur.

Analog stellt sich bei Beweisvisualisierungen die Frage nach der Verbindung zwischen dem Beweis und seiner Visualisierung. Bei fast allen Theorembeweisern sind formaler Beweis und visuelle Darstellung so stark verschränkt, daß sie zu einem integrierten Gesamtsystem gehören. Dies gilt für interaktive Theorembeweiser wie HOL, automatische Beweiser wie ${}_3T^AP$, bis hin zum Lernprogramm HYPERPROOF. Da die Visualisierungen in diesem Fall automatisch generiert werden, sind zu deren Erzeugung keine Kenntnisse über den Beweis notwendig. Bei einigen Theorembeweiser ist die Visualisierung vom Beweissystem entkoppelt: Für HOL gibt es auf Basis von PROOFVIEWS eine HTML-Ausgabe von Beweisen (ausgeführt in einem bestimmten Kalkül) im *Structured calculational proof format*, welche dann mit einem WWW-Stöberer inspiziert werden können, und bei ${}_3T^AP$ lassen sich Tableaubeweise nachträglich mit dem speziellen Programm MORETAB schrittweise nachvollziehen. Die derzeit existierenden Visualisierungen nicht formaler Beweise wurden bisher immer per Hand kodiert. Dies liegt im wesentlichen am Mangel an jeglichen formalen Komponenten der visualisierten Beweise: In Gloors System besteht der Beweis aus verschiedenen mit HyperCard verknüpften Textbausteinen; der Visualisierer stellt die Visualisierung und deren Verknüpfung mit dem Beweistext ebenfalls manuell mit HyperCard her. In PROOFVIEWS werden alle zur strukturierten Darstellung benötigten Informationen per Hand in den Beweistext eingefügt. Der Visualisierer muß also sehr genaue Kenntnisse über den geführten Beweis besitzen, um eine sinnvolle Visualisierung zu erzeugen.

8.5 E: Interaktion

Die Interaktion des *Benutzers* mit dem Visualisierungssystem wird in der Taxonomie weitgehend durch allgemeine Merkmale einer graphischer Benutzungsschnittstellen und spezielle Merkmale eines Animationssystems charakterisiert. Da dieser Teil ebenfalls unabhängig vom Anwendungsbereich Softwarevisualisierung ist, übernehmen wir ihn vollständig für eine Charakterisierung der Interaktion bei Beweisvisualisierungssystemen; siehe Abbildung 8.5

E.1. *Stil*: Wie interagiert der Benutzer mit dem System und wie steuert er es?

E.2. *Navigation*: In welchem Maße unterstützt das System eine Navigation durch die Visualisierung?

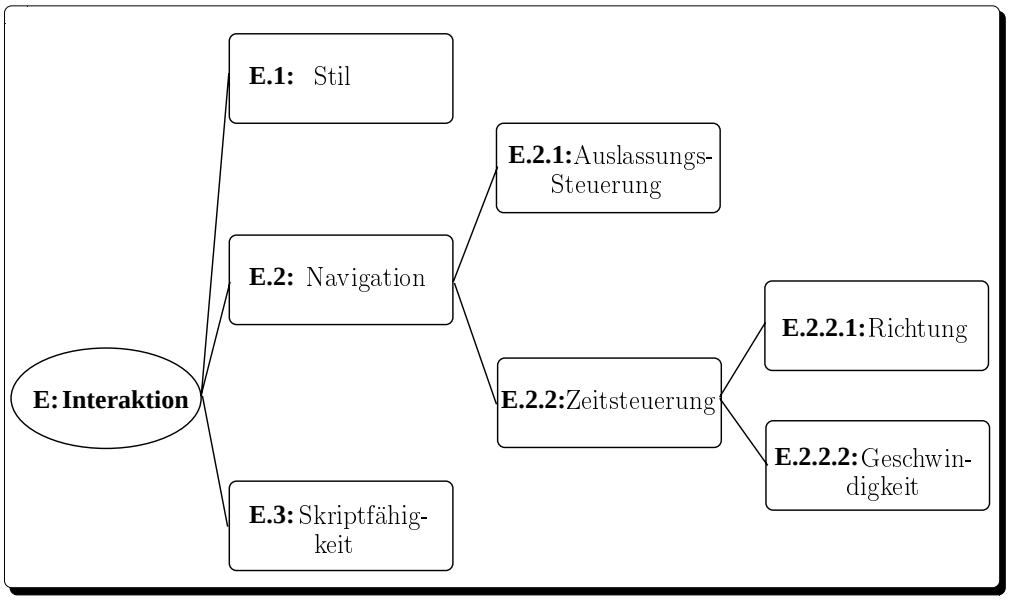


Abbildung 8.5: Hierarchie zu **Interaktion**

E.2.1 *Auslassungssteuerung*: Kann der Benutzer Details auslassen?

E.2.2 *Zeitliche Steuerung*: In welchem Maße kann der Benutzer den zeitlichen Verlauf der Visualisierung steuern?

E.2.2.1. *Richtung*: Kann der Benutzer die zeitliche Richtung umkehren?

E.2.2.2. *Geschwindigkeit*: In welchem Maße kann der Benutzer die Ausführungsgeschwindigkeit steuern?

E.3. *Skriptfähigkeit*: Unterstützt das System das Speichern und Wiederholen von Visualisierungen?

Der *Stil* charakterisiert die Benutzungsschnittstelle des Beweisvisualisierungssystems aus dem Blickpunkt des Betrachters der Visualisierung. Der Stil ist nicht auf Visualisierungssysteme begrenzt, sondern ist ein allgemeines Merkmal von Software. [PRICE et al. 1998] unterscheiden drei Interaktionstypen:

- *Kommandozeilen*: Das System wird durch textuelle Eingabe einzelner Kommandos interaktiv oder durch Angabe von Parametern beim Programmstart gesteuert.
- *Skriptbasierte Steuerung*: Die Steuerungsanweisungen für das System sind vollständig in einem Skript enthalten. Eine weitere Interaktion mit dem System ist normalerweise nicht vorgesehen.

- *Graphische Benutzungschnittstellen*: Der Benutzer steuert das System mit der Maus durch Anwählen von Schaltern und Menüs. Insbesondere gehören Fensterbasierte Systeme in diesen Bereich.

Bei den derzeit verfügbaren Beweisvisualisierungssystemen überwiegen graphischen Benutzungschnittstellen. Lediglich bei Programmen, die formale Beweise strukturiert darstellen, finden sich Kommandozeilen-orientierte Systeme. Dazu gehören zum Beispiel PROOFVIEWS, welches aus einer Shell gestartet wird, und MORETAB, das durch Tastaturkürzel gesteuert wird. Eine Skript-basierte Steuerung von Beweisvisualisierungen ist uns nicht bekannt. Fast alle Systeme besitzen eine graphische Benutzungsschnittstelle und sind Fenster-basiert.

Die *Navigation* beschreibt Funktionsmerkmale des Systems mit denen der Benutzer Beweise und Formeln betrachten kann. Allgemein verbreitete Merkmale sind zum Beispiel Änderungen der Größe und Auflösung der visuellen Darstellung oder eine Abstraktion von den Details eines Beweises. So zeigt das KIV-System den Beweisbaum eines formalen Sequenzenbeweises ohne die zugehörigen Sequenzen an, und der Baum kann vergrößert oder verkleinert dargestellt werden. Die *Auslassungsteuerung* beschreibt die Fähigkeit des Systems, auf Benutzeranforderung hin Details des Beweises oder bestimmte Informationen zu unterdrücken. Zum Beispiel kann der Benutzer bei mit PROOFVIEWS erzeugten HTML-Beweisen ineinander verschachtelte Teilbeweise beliebig verdecken und wieder aufdecken. Falls das System Beweise nicht nur statisch visualisiert, sondern animiert, ist die *zeitliche Steuerung* ein wichtiges Kriterium, um zu beurteilen, wie der Benutzer den zeitlichen Verlauf der Animation steuern kann. Bei der mit dem Algorithmenanimationssystem ZEUS erstellten Animation des Satzes von Pythagoras kann der Benutzer die Animation starten, anhalten, wieder fortführen und die Ausführungs*geschwindigkeit* modifizieren. Die *Richtung* der Zeit läßt sich hingegen nicht umkehren.

Die *Skriptfähigkeit* beschreibt, ob mit dem Beweisvisualisierungssystem eine erzeugte Visualisierung abgespeichert und später wiederholt von verschiedenen Personen abgerufen werden kann. Der automatischen Theorembeweiser ${}_3T^AP$ erzeugt während der Beweissuche ein Skript, das MORETAB zur Darstellung des gefundenen Beweises benutzt. Bei anderen Theorembeweisern gibt es diese Fähigkeit nicht, da der Beweis und dessen Visualisierung zu stark verschränkt sind. Skriptfähigkeit findet man bei Systemen zur Visualisierung von nicht formalen Beweisen noch vergebens, wenn man von der unveränderlichen ZEUS-Animation zum Satz von Pythagoras absieht.

8.6 F: Wirksamkeit

Die bisher aufgeführten Punkte der Taxonomie berücksichtigen hauptsächlich technische Merkmale von Beweisvisualisierungssystemen. Darüberhinaus ist für den praktischen Einsatz auch die *Wirksamkeit* der entwickelten Visualisierung auf den Benutzer für die Beurteilung eines Systems von Bedeutung. Dieser Teil der Taxonomie ist eben-

falls von so allgemeiner Natur, so daß wir ihn ohne jegliche Änderung übernehmen. Siehe Abbildung 8.6.

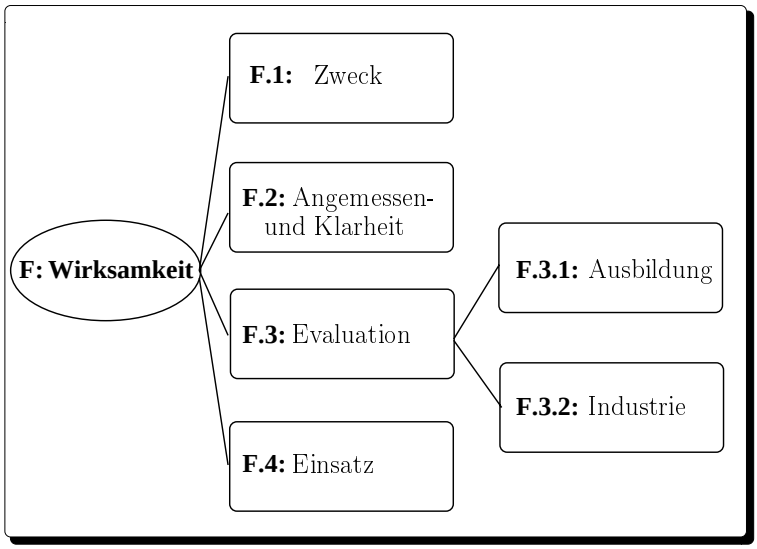


Abbildung 8.6: Hierarchie zur **Wirksamkeit**

F.1. *Zweck*: Für welchen Zweck ist das System geeignet?

F.2. *Angemessenheit und Klarheit*: Falls automatisch erzeugte Visualisierungen vom System verwendet werden, wie gut vermitteln sie die zu visualisierenden Informationen?

F.3. *Evaluation*: In welchem Maße wurde das System evaluiert?

F.4. *Einsatz*: In welchem Zeitraum und wie wurde das System bisher verwendet?

F.4.1 *Ausbildung*: Wurde das System zur Ausbildung an (Hoch-)Schulen benutzt?

F.4.2 *Industrie*: Wurde das System in der Industrie eingesetzt?

Der *Zweck* von Visualisierungen formaler Beweise, die in Theorembeweisern integriert sind, liegt in der Unterstützung des Benutzers bei der Beweissuche und weniger in der Vermittlung von Kenntnissen und Fähigkeiten im Führen formaler Beweise. Im Gegensatz dazu dient das Lernprogramm HYPERPROOF Studierenden, einfache logische Aussagen formal zu beweisen und mit Hilfe zusätzlicher anschaulicher graphischer Darstellungen besser zu verstehen.

Die *Angemessenheit und Klarheit* charakterisiert bei automatisch erzeugten Beweisvisualisierungen, wie gut sie die dem Betrachter helfen, einen Beweis zu verstehen. Verstehen bedeutet dabei nicht so sehr das rein syntaktische Nachvollziehen einzelner Beweisschritte, sondern ein gesamtheitliches mentales Verständnis des Beweises. Letzteres wird durch die rein baumartige Darstellung formaler Beweise bei Theorembeweisern nur unzureichend gefördert (ist allerdings auch nicht deren Zweck). Bei den existierenden Systemen zur Visualisierung nicht formaler Beweise, wird die Visualisierung nicht automatisch sondern manuell durch den Visualisierer generiert. Die Visualisierungen sind meist für einen begrenzten Themenbereich konzipiert und speziell dafür entworfen. Mit diesen Systemen kann der Benutzer deswegen die visualisierten Beweise vermutlich besser verstehen.

Ob dies der Fall ist, läßt sich empirisch *evaluieren*. Der Effekt von Beweisvisualisierungen auf den Betrachter ist (im Gegensatz zu Algorithmenanimationen) allerdings bisher noch gar nicht untersucht worden. Die Ursache dürfte im bisher nur spärlichen Einsatz solcher Systeme und der Schwierigkeit einer repräsentativen Evaluation liegen. Meistens begnügt man sich mit individuellen Erfahrungen von Benutzern, wie etwa Kommentare von Studierenden, die HYPERPROOF im Unterricht benutzten.

Beweisvisualisierungssysteme sind bisher lediglich im Forschungs- und Ausbildungsbereich eingesetzt worden. Dies trifft insbesondere für nicht formale Beweise zu, während sich Theorembeweiser mit deren integrierten Visualisierungkomponente zur Soft- und Hardwareverifikation mittlerweile an der Schwelle zum industriellen Einsatz befinden.

Kapitel 9

Ausblick

Wir sind in dieser Arbeit der Frage nachgegangen, wie man Beweise für den Informatikunterricht multimedial aufbereiten kann. Zum einen haben wir ein Konzept vorgestellt, um interaktive Lernprogramme für die wichtige Klasse der Reduktionsbeweise zu entwickeln, und zum anderen haben wir eine zur Algorithmenanimation analoge Methode – Beweisanimation – entworfen, um Beweistexte mit einer begleitenden Animation visuell zu veranschaulichen.

Die von uns entwickelten interaktiven Lernprogramme einiger Reduktionsbeweise decken einen wichtigen Teil der Beweise ab, wie sie in einer einführenden Informatikvorlesung zum Thema NP-Vollständigkeitstheorie an den Universitäten behandelt werden. Unsere auf einer Selbsteinschätzung der Studierenden basierende Evaluierung zweier dieser Programme während eines vorlesungsbegleitenden Einsatzes in den Übungen zur Vorlesung „Informatik III“ an der Universität Karlsruhe offenbarte, daß die Akzeptanz derartiger Lernprogramme unter dem zusätzlichen Zeitaufwand leidet, den die Benutzung der Programme den Studierenden verursacht. Unsere Auswertung zeigte aber auch, daß die Studierenden, die die Lernprogramme benutzten, korrektere Lösungen bei den zugehörigen Übungsaufgaben abgaben, als die Studierenden, die die Lernprogramme nicht benutzten.

Unser zweites Konzept zur multimedialen Aufbereitung von Beweisen – die Beweisanimation – erlaubt die Visualisierung einer großen Klasse von konstruktiven Beweisen. Die Beweisanimation ist der erste systematische und rechnergestützte Ansatz zur Visualisierung von Beweisen für Ausbildungszwecke mit Hilfe einer den Beweistext begleitenden Animation. So wie bei einer Algorithmenanimation der Lernende die einzelnen Operationen eines Algorithmus schrittweise anhand graphischer Darstellungen der Datenstrukturen nachvollziehen kann, werden bei unseren Beweisanimationen die konstruktiven Argumentationen eines Beweistextes mit Hilfe graphischer Darstellungen der im Beweistext manipulierten logischen Aussagen veranschaulicht. Mit unserem Beweisanimationssystem SCAPA lassen sich derartige Beweisanimationen erstellen. Die so implementierten Beweisanimationen folgen keinem vorher festgelegten starren Ablauf, sondern passen sich den Eingaben des Benutzers an. Sie ermöglichen den Studierenden ein selbstgesteuertes und erfahrungsbetontes Lernen von Beweisen.

Im Gegensatz zu den Lernprogrammen aus dem Bereich der Reduktionsbeweise, wurden die Beweisanimationen von uns bisher noch nicht im Rahmen einer Vorlesung eingesetzt. Unsere gegenwärtigen Arbeiten beschäftigen sich deswegen mit dem Einsatz und der Evaluierung von Beweisanimationen begleitend zur Vorlesung „Informatik III“ im WS 99/00. Unser Ziel ist es, die Akzeptanz der Beweisanimationen und deren Auswirkung auf die Studierenden zu messen.

Zukünftige Arbeiten auf dem Gebiet der Beweisanimation sollten sich – neben der Erstellung weiterer Animationen und deren Einsatz und Evaluierung – mit der Verbesserung der Werkzeuge zur Erstellung von Beweisanimationen beschäftigen. Mit vergleichbar geringem Aufwand können unsere Werkzeuge zusammen mit dem mausgesteuerten Animationssystem MARACA und einer graphischen Benutzungsschnittstelle zu einem Autorensystem integriert werden. Dies würde die Entwicklung (graphisch einfacher) Beweisanimationen, die einem starren Ablauf folgen, drastisch erleichtern.

Weit aus schwieriger ist eine Weiterentwicklung unseres Ansatzes, um Beweisanimationen *automatisch* aus einem Beweis zu erstellen. Dazu ist es notwendig, die strukturierten, nicht formalen Beweise systematisch in formale Beweise zu transformieren. Auf Basis des formalen Beweises könnte dann ein zugehöriges Programm vollautomatisch synthetisiert werden, und aus diesem Programm kann dann ebenso automatisch eine Animation erstellt werden. Die sich daraus ergebenen Forschungsfragen sind ebenso vielfältig wie komplex: Wie erstelle ich aus einem strukturierten, nicht formalen Beweis systematisch einen formalen Beweis? Wie sieht ein formaler Beweiskalkül aus, der die Strukturinformationen und nicht formalen Komponenten des ursprünglichen Beweises im formalen Beweis erhält? Wie kann aus dem formalen Beweis ein Algorithmus synthetisiert werden, der als Basis für eine automatisch erstellte Animation dienen kann? Wie kann diese Animation mit dem ursprünglichen nicht formalen Beweistext gekoppelt werden?

Letztendlich hängt der Einsatz von Beweisanimationen und die Weiterentwicklung von Werkzeugen zur Erstellung von Beweisanimationen stark davon ab, wie und ob die Studierenden das zusätzliche Angebot nutzten und wie weit sie davon profitieren können.

Anhang A

Definitionen und Algorithmen

In diesem Kapitel wiederholen wir eine Reihe bekannter Definitionen und Verfahren, die zum detaillierten Verständnis der vorangegangenen Kapiteln beitragen. Die Darstellung beruht, soweit nicht anders angegeben, auf dem Vorlesungskriptum. Wir haben nur an manchen Stelle den Text noch etwas gekürzt und einige Kommentare von uns in Klammern innerhalb des Textes eingefügt.

A.1 LR-Parsing

Das LR-Parsing ist ein „bottom-up"-Verfahren, bei der ausgehend von einem Terminalwort mit Hilfe von Regeln schrittweise eine Rechtsableitung bezüglich einer ϵ-freien kontextfreien Grammatik konstruiert wird. Damit dabei der Parser das Ende des Wortes erkennt, wird vereinbart, das es immer mit dem Sonderzeichen $\$$ endet. Dabei wird vorausgesetzt, daß $\$$ nicht in der Menge der Nichtterminal oder Terminalzeichen vorkommt. Es werden nur Grammatiken von folgender Form betrachtet:

Definition 13 (Erweiterte Grammatik)
Zu einer kontextfreien Grammatik $G = (N, T, S, P)$ entsteht die *erweiterte Grammatik*

$$G' = (N \cup \{S'\}, T \cup \{\$\}, S', P \cup \{S' \to S\$\})$$

durch Hinzunahme eines neuen Startsymbols und einer neuen Produktion.

Wir erklären die Funktionsweise des LR-Parsings im folgenden anhand der Beispielgrammatik G_{AEXP}.

Beispiel 20 (Die Grammatik G_{AEXP})

1	$S \to S + T$	4	$T \to F$
2	$S \to T$	5	$F \to (S)$
3	$T \to T \times F$	6	$F \to id$

Um festzustellen, ob ein Wort w im Sprachschatz von G_{AEXP} liegt, kann man die LR(1)-Parsertabelle aus Abbildung A.1 benutzen. Die systematische Konstruktion solch einer Tabelle aus einer Grammatik, wird uns weiter unten beschäftigen.

Zustand	Aktion						Sprung		
	id	+	×	(	)	$	S	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Abbildung A.1: Parsertabelle für G_{AEXP}

Ein *LR-Parser* kann mehrere Zustände haben. Diese Zustände werden während des Parsings in einem Keller abgelegt und zwar immer abwechselnd ein Buchstabe des Kelleralphabets, ein Zustandssymbol, ein Buchstabe des Kelleralphabets usw. Das oberste Kellerzeichen ist immer ein Zustandssymbol: der *aktuelle Zustand*. Der Einfachheit halber benutzen wir Zahlen zur Bezeichnung von Zuständen. Der Anfangszustand wird durch 0 dargestellt. Ein LR-Parser startet in der Konfiguration mit leerem Keller und dem zu parsenden Wort w als Eingabe. Eine solche Konfiguration schreiben wir als $0 \mid w$. Die nächste anzuwendende Regel wird durch die Einträge der Parsertabelle bestimmt. In Abhängigkeit von dem ersten noch nicht gelesenen Buchstaben des Eingabewortes und dem aktuellen Zustand wird aus der Parsertabelle die nächste auszuführende Aktion ermittelt. Steht in dem entsprechenden Feld der LR-Parsertabelle kein Eintrag, so ist der Parsingversuch fehlgeschlagen und das Eingabewort gehört nicht zu $L(G_{AEXP})$. Steht dort acc, so ist das Eingabewort akzeptiert. Ein Eintrag si signalisiert, daß eine shift Operation ausgeführt wird: das nächste Zeichen von w wird auf dem Keller abgelegt. Der neue aktuelle Zustand ist i. Die Zahl i wird jetzt das oberste Kellerzeichen. Steht in dem Feld der LR-Parsertabelle der Eintrag rj, so ist eine reduce-Aktion mit der j-ten Regel, $A \to v$, der Grammatik G_{AEXP} auszuführen: Der Keller weist in diesem Fall das Endstück von der Form $s_0 b_1 s_1 \ldots s_{k-1} b_k j$ auf. Zunächst wird $b_1 s_1 \ldots s_{k-1} b_k j$ durch die linke Seite der Produktion ersetzt, so daß jetzt der Keller auf $s_0 A$ endet. Der neue aktuelle Zustand wird jetzt wieder aus der Parsertabelle ermittelt unter dem Zustand s_0 und der Spalte für A. Wir wollen im folgenden $S' \mid \varepsilon$ als Endkonfiguration für LR-Parser ansehen. Tabelle A.1 zeigt eine

LR-Kellerableitung mit dem Parser für das Wort id $\times$ id $+$ id\$.

Zeile	Keller	Eingabe	Ableitung	
1	0	id $\times$ id $+$ id\$		id $\times$ id $+$ id
2	0id5	$\times$ id $+$ id\$	$\Leftarrow^0$	id $\cdot \times$ id $+$ id
3	0F3	$\times$ id $+$ id\$	$\Leftarrow^1$	F $\cdot \times$ id $+$ id
4	0T2	$\times$ id $+$ id\$	$\Leftarrow^1$	T $\cdot \times$ id $+$ id
5	0T2$\times$ 7	id $+$ id\$	$\Leftarrow^0$	T $\times \cdot$ id $+$ id
6	0T2$\times$ 7id5	$+$ id\$	$\Leftarrow^0$	T $\times$ id $\cdot +$ id
7	0T2$\times$ 7F10	$+$ id\$	$\Leftarrow^1$	T $\times$ F $\cdot +$ id
8	0T2	$+$ id\$	$\Leftarrow^1$	T $\cdot +$ id
9	0S1	$+$ id\$	$\Leftarrow^1$	S $\cdot +$ id
10	0S1$+$6	id\$	$\Leftarrow^0$	S $+ \cdot$ id
11	0S1$+$6id5	\$	$\Leftarrow^0$	S $+$ id $\cdot$
12	0S1$+$6F3	\$	$\Leftarrow^1$	S $+$ F $\cdot$
13	0S1$+$6T9	\$	$\Leftarrow^1$	S $+$ T $\cdot$
14	0S1	\$	$\Leftarrow^1$	S $\cdot$

Tabelle A.1: Beispiel einer LR-Kellerableitung des Worts id $\times$ id $+$ id\$. Die letzte Spalte zeigt, wie bei einer erfolgreichen LR-Kellerableitung im „bottom-up“ Verfahren ein Rechtsableitung aus dem Startsymbol S konstruiert wird. Der Punkt „$\cdot$“ markiert dabei die Position des Präfixes, der bei der Kellerableitung schon verarbeitet wurde.

Den Zeichen auf dem Keller kommt eine besondere Bedeutung zu, denn sie kommen als Präfix der einzelnen Zwischenwörter bei der Rechtsableitung vor. Beim LR-Parsing wird dies ausgenutzt, um unsinnige Ableitungen während des Parsens zu vermeiden: Aktionen, die zu einer Ableitung führen, bei der ein Präfix entsteht, welcher bei *keiner* Rechtsableitung aus dem Startsymbol entstehen, kommen nicht der LR-Parsertabelle vor. Präfixe, die bei einer Rechtsableitung vorkommen, werden *lebensfähige Präfixe* genannt. Der erste Schritt zur LR-Parsertabelle ist die Konstruktion eines endlichen Automaten – die nichtdeterministische LR-Kellermaschine –, welcher alle lebensfähigen Präfixe erkennt. Dazu wird ein sogenanntes *LR(k)-Element* eingeführt. Es ist von der Form $[A \to u \cdot v, y]$, wobei $y \in T^*$ und $A \to uv$ eine Ableitung ist.

Definition 14 (Die nichtdet. LR(k)-Maschine)
Sei $G = (N, T, S, P)$ eine kontextfreie Grammatik, G' ihre \$-Erweiterung, dann ist die LR(k)-Maschine $M_{G,k} = (Q, V, q_0, \delta, F)$ ein nichtdeterministischer endlicher Automat mit spontanen Übergängen, wie folgt definiert:

Q ist die Menge aller LR(k)-Elemente von G',
$V = N \cup T$,
$q_0 = [S' \to \cdot S\$, \varepsilon]$,
$F = Q$,
$[A \to uX \cdot v, y] \in \delta([A \to u \cdot Xv, y], X)$ für $X \in V$
$[B \to \cdot v_1, z] \in \delta([A \to u \cdot Bv, y], \varepsilon)$, falls $B \in N, z \in \{u \in T^* \mid |u| \leq k, vy \Rightarrow u\}$ und $B \to v_1 \in P$

(Im Skriptum und in der Vorlesung wird noch bewiesen, daß dieser Automat alle lebensfähigen Präfixe der Grammatik akzeptiert. Wir lassen diesen Beweis hier aus, da uns im folgenden nur die Konstruktion der LR(k)-Maschine interessiert.)

Beispiel 21
Der auf der folgenden Seite abgebildete endliche Automat stellt die nichtdeterministische LR(1)-Kellermaschine zur Grammatik $G = (\{a, b\}, \{A, B, S\}, S, \{S \to Ab, S \to Bb, A \to Sb, A \to a, B \to Sb\})$ dar. Der Automat wird ausgehend vom Startzustand q_0 schrittweise mit Hilfe von Definition 14 konstruiert.

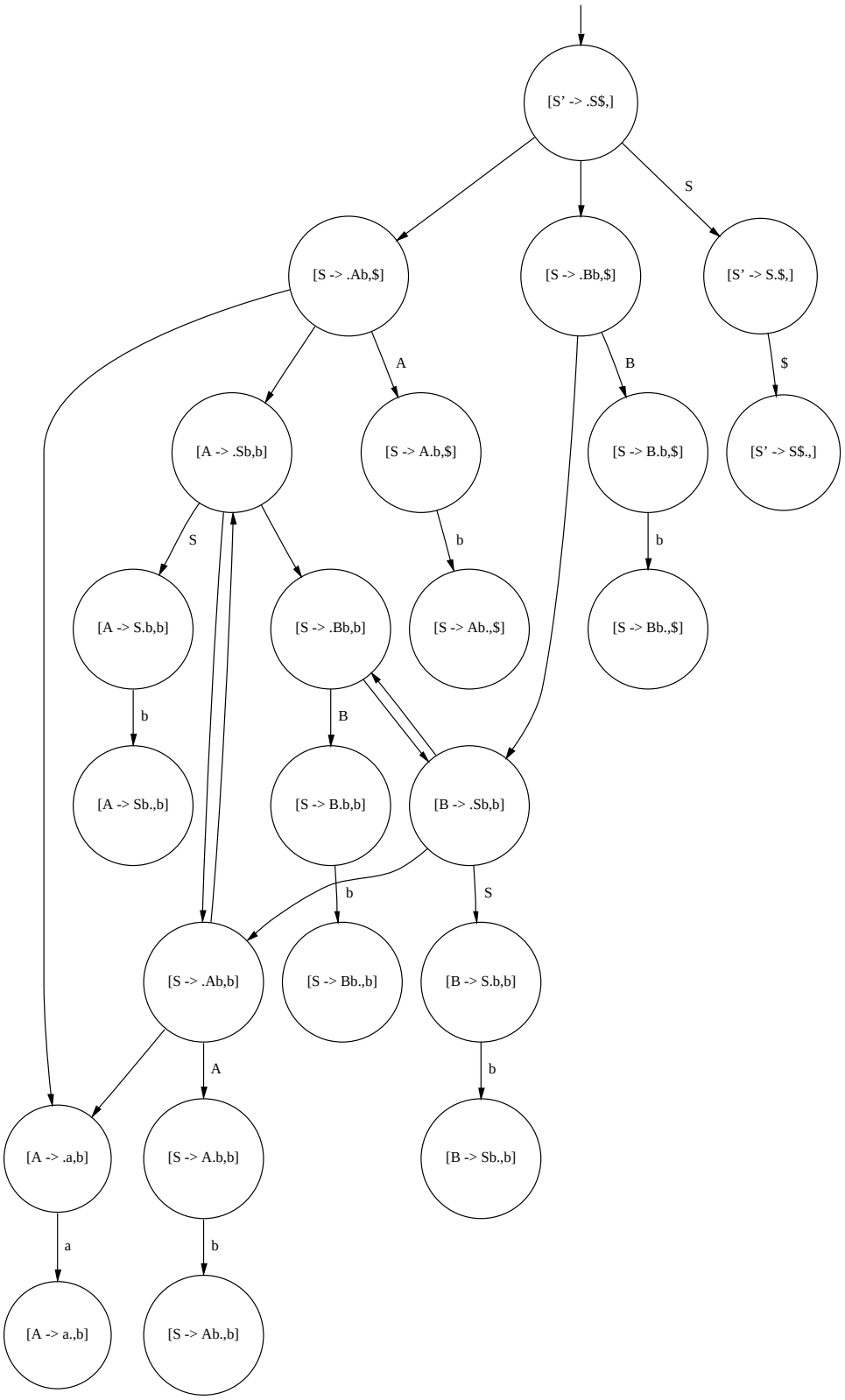
(Aus diesem Automaten wird durch das zuvor in der Vorlesung behandelte Teilmengenkonstruktionsverfahren ein äquivalenter deterministischer Automat – die deterministische LR(k)-Kellermaschine – erzeugt. Abbildung A.2 zeigt den resultierenden Automaten für die Grammatik G.)

Mit Hilfe der deterministische LR(k)-Kellermaschine ist man in der Lage, zu einer kontextfreien Grammatik eine Parsertabelle zu konstruieren. Aus praktischen Gründen betrachten wir nur den Fall $k = 0$ und $k = 1$.

Definition 15 (Der kanonische LR(k)-Parser)
$G = (N, T, S, P)$ sei eine kontextfreie Grammatik, $D_{G,k} = (Q, V, q_0, \Delta, F)$ die zugehörige deterministische LR(k)-Maschine, $k \in \{0, 1\}$. Der kanonische LR(k)-Parser $K = K_{G,k}$ für G besitzt als Zustände die Zustände von $D_{G,k}$. Sei $q \in Q$, $c \in T$ und $A \in N$. Die Parsertabelle für K wird durch die folgenden Regeln konstruiert:

- falls $\Delta(q, c)$ definiert ist, kommt $s\Delta(q, c)$ im Feld (q, c) vor (shift-Anweisung).
- falls $\Delta(q, A)$ definiert ist, kommt $\Delta(q, A)$ im Feld (q, A) vor (goto-Anweisung).
- falls ein Element $[A \to u \cdot , y]$ in q liegt, kommt rj für die mit j numerierte Ableitung $A \to u$ im Feld (q, y) vor (reduce-Anweisung).
- falls $[S' \to S \cdot \$, y]$ in q vorkommt, steht acc im Feld $(q, \$)$.

Es kann passieren, daß die LR(k)-Parsertabelle zu einem Zustand und einem Symbol mehrere Einträge enthält. In solch einem Fall können beim LR-Parsing die Aktionen nicht mehr deterministisch ausgewählt werden. Es handelt sich dabei um einen Konflikt. In der Praxis interessiert man sich nur für LR-Parser ohne solche Konflikte.


[S' -> .S$,]
S
[S -> .Ab,$]
[S -> .Bb,$]
[S' -> S.$,]
A
B
$
[A -> .Sb,b]
[S -> A.b,$]
[S -> B.b,$]
[S' -> S$.,]
S
b
b
[A -> S.b,b]
[S -> .Bb,b]
[S -> Ab.,$]
[S -> Bb.,$]
b
B
[A -> Sb.,b]
[S -> B.b,b]
[B -> .Sb,b]
b
S
[S -> .Ab,b]
[S -> Bb.,b]
[B -> S.b,b]
A
b
[A -> .a,b]
[S -> A.b,b]
[B -> Sb.,b]
a
b
[A -> a.,b]
[S -> Ab.,b]

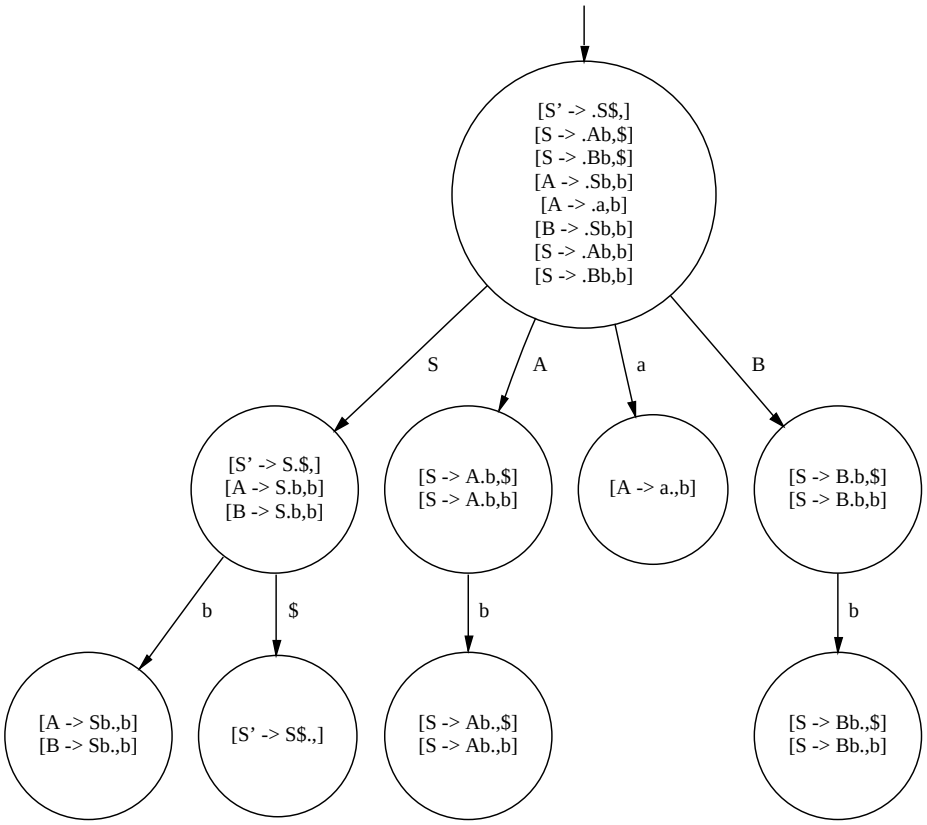


Abbildung A.2: Beispiel einer deterministische LR(1)-Kellermaschine

Definition 16 (Konflikte)
Sei $D_{G,k}$ der deterministische LR(k)-Automat zur Grammatik G. Ein Zustand q von $D_{G,k}$ enthält einen *reduce-reduce-Konflikt*, wenn es zwei verschiedene Produktionen $A \to v$, $B \to u$ und ein Wort $y \in k : T^*$ mit $[A \to v\cdot, y] \in q$ und $[B \to u\cdot, y] \in q$ gibt. Ein Zustand q von $D_{G,k}$ enthält einen *shift-reduce-Konflikt*, wenn es zwei Produktionen $A \to vcw$, $B \to u$ mit $c \in T$ und Wörter $y, z \in k : T^*$ mit $1 : y = c$ mit $[A \to v \cdot cw, z] \in q$ und $[B \to u \cdot , y] \in q$ gibt.

A.2 Turingmaschinen

Eine Turingmaschine [TURING 1936] ist eine abstrakte Maschine, um den Begriff der Berechenbarkeit einer Funktion zu formalisieren. Wir geben im folgenden eine gegenüber dem Vorlesungsskriptum oder einem Lehrbuch stark verkürzte und weniger formale Darstellung der wesentlichen Begriffe.

Eine Turingmaschine besteht aus:

1. Einem zweiseitig unendlichen Band, das in einzelne Zellen aufgeteilt ist. Jede Zelle enthält ein Symbol aus einer endlichen Menge Γ. Γ enthalte immer ein spezielles Symbol B.

2. Einem Schreib-/Lesekopf, der sich immer genau über eine Zelle des Bandes befindet und in der Lage ist, nach links und nach rechts über das Band zu wandern.

3. Einer Kontrolleinheit, die sich zu jeder Zeit in einem internen Zustand befindet, der aus einer endlichen Zustandsmenge Q entnommen ist.

Das Verhalten einer Turingmaschine wird, wie folgt, durch eine Zustandsübergangsfunktion $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, U\}$ beschrieben: Wenn $q \in Q$ der aktuelle Zustand der Turingmaschine ist, $a \in \Gamma$ das Symbol der Zelle unterhalb des Schreib-/Lesekopfs und $\delta(q, a) = \langle r, b, L\rangle$ ($\delta(q, a) = \langle r, b, R\rangle$, $\delta(q, a) = \langle r, b, U\rangle$) gilt, dann überschreibt die Turingmaschine das Symbol in der Zelle unterhalb des Schreib-/Lesekopfs mit b, der Kopf bewegt sich eine Zelle weiter nach links (rechts, bewegt sich nicht) über das Band und die Turingmaschine geht in den internen Zustand r über. Eine Turingmaschine *hält*, falls der interne Zustand in einen ausgezeichneten Endzustand $q_f \in Q$ übergeht.

Eine *Konfiguration* einer Turingmaschine besteht aus der Position des Schreib-/Lesekopfes, dem Band und dem Zustand der Turingmaschine. Eine Konfiguration wird abkürzend durch den relevanten Teil des Bandes notiert: gegeben als Zeichenkette, mit dem aktuellen Zustand geschrieben vor dem Zeichen der Zelle unter dem Schreib-/Lesekopf. Zum Beispiel beschreibt $B00q_1 1B$ eine Konfiguration einer Turingmaschine, die sich im Zustand q_1 befindet, mit einem Band, das die Symbole $0, 0, 1$ enthält, wobei sich links und rechts davon unendlich viele Leerzeichen B befinden und der Schreib-/Lesekopf über der Zelle mit dem Symbol 1 steht.

Mit Turingmaschinen können, ähnlich zu einem gewöhnlichen Rechner, Funktionen berechnet werden. Dazu werden die Funktionsargumente geeignet kodiert und auf das Band geschrieben. Dann wird die Turingmaschine gestartet und wenn sie hält, kann der Funktionswert vom Band abgelesen werden.

Eine *nichtdeterministische Turingmaschine* ist analog zu einer deterministischen Turingmaschine definiert, mit dem einzigen Unterschied, daß die Übergangsfunktion einen Zustand in eine *Menge* von möglichen Aktionen abbildet:

$$\delta : \Gamma \times \Sigma \to 2^{\Gamma \times \Sigma \times \{L,R,U\}}$$

Das Verhalten einer nichtdeterministische Turingmaschine ist wie folgt definiert: Wenn q der aktuelle Zustand der nichtdeterministischen Turingmaschine ist und s das sich unter dem Schreib-/Lesekopf befindliche Symbol, dann wählt die nichtdeterministische Turingmaschine ein Tripel $\langle r, t, d\rangle \in \delta(q, s)$ und der interne Zustand der Maschine ändert sich dementsprechend.

Turingmaschinen stellen ein elegantes und einfaches Modell dar, um den Begriff einer Berechnung und eines Algorithmus zu formalisieren. Mit Hilfe diese Modells,

läßt sich auf elementare Weise nachweisen, daß es Funktionen gibt, die nicht berechenbar sind. Normalerweise wird dies in einer Anfängervorlesung vom *allgemeinen Halteproblem* für Turingmaschine gezeigt, d.h. es gibt keinen Algorithmus (keine Turingmaschine), der bei Eingabe einer Turingmaschine und einer Anfangskonfiguration entscheidet, ob die Turingmaschine hält oder nicht. Die Unentscheidbarkeit des allgemeinen Halteproblems dient dann als Ausgangspunkt für einen Reduktionsbeweis der Unentscheidbarkeit des folgenden Problems.

A.3 Postsche Korrespondenzproblem

Beim Postschen Korrespondenzproblem [POST 1946] handelt es sich um ein Wortproblem, welches in fast jeder in die Theoretische Informatik einführende Vorlesung behandelt wird. Das Postsche Korrespondenzproblem dient als Ausgangspunkt für viele wichtige Resultate in der Informatik: zum Beispiel der Unterscheidbarkeit der Prädikatenlogik erster Stufe.

Definition 17 (Postsches Korrespondenzproblem)

Eine Instanz des Postschen Korrespondenzproblems ist eine Folge von Wortpaaren $\langle (x_1, y_1), \ldots, (x_k, y_k) \rangle, x_i, y_i \in \Sigma^*$ über einem endlichen Alphabet Σ.

Eine *Lösung* einer Instanz des Postschen Korrespondenzproblems ist eine Folge $\langle i_1, \ldots, i_n \rangle$ von Indizes mit $n > 0$, so daß $x_{i_1} x_{i_2} \ldots x_{i_n} = y_{i_1} y_{i_2} \ldots y_{i_n}$ gilt.

Zum Beispiel besitzt die Instanz $\langle (0, 01), (10, 0) \rangle$ die Lösung $\langle 1, 2 \rangle$ (bzw. 010 als Lösungswort), während die Instanz $\langle (0, 1) \rangle$ keine Lösung besitzt. Nicht immer sind diese Beispiele und deren Lösungen — sofern sie existieren — derart einfach: die kürzeste Lösungfolge der „harmlos“ aussehend Instanz

$$\langle (001, 0), (01, 011), (01, 101), (10, 001) \rangle$$

besteht aus 66 Indizes.

Die Unentscheidbarkeit des Postschen Korrespondenzproblems wird durch Reduktion auf eine modifizierte Variante bewiesen. Beim *modifizierten Postschen Korrespondenzproblem* muß eine Lösung mit dem ersten Wortpaar der Folge beginnen. Ansonsten gleichen sich die Definitionen.

A.4 NP-Vollständigkeit

Als mathematische Basis zur Beschreibung von Lösungsverfahren wird überlicherweise das Modell der Turingmaschine herangezogen, siehe dazu den früheren Abschnitt A.2. Eine (deterministische oder nichtdeterministische) Turingmaschine *entscheidet* eine Sprache $L \subseteq \Sigma^*$ genau dann, wenn die Maschine bei jeder Eingabe

$w \in \Sigma^*$ anhält und die finale Konfiguration $Bq1B$ ist, falls $w \in L$ gilt, und $Bq0B$ sonst. Dabei sei $q \in \Gamma$ ein Endzustand.

Um von der nicht formalen Beschreibung von Entscheidungsproblemen zu abstrahieren, werden diese – wie wir schon bei der Erläuterung der Reduktionsmethode gesehen haben – in der Komplexitätstheorie als Formale Sprachen $L \subseteq \Sigma^*$ über einer endlichen Signatur Σ kodiert. Eine *Instanz* eines Entscheidungsproblems ist dann ein Wort $w \in \Sigma^*$ und die Entscheidungsfrage ist, ob $w \in L$ gilt oder nicht. Dieser Ansatz vereinfacht die Untersuchung von Entscheidungsproblemen erheblich und erlaubt es, sie auf einem abstraktem mathematischen Niveau zu untersuchen. Eine Sprache L heißt *entscheidbar* genau dann, wenn die Frage $w \in L$? für jedes $w \in \Sigma^*$ von einer Turingmaschine entschieden werden kann. Etwas formaler läßt sich dies wie folgt formulieren:

Definition 18
Eine Sprache $L \subseteq \Sigma^*$ heißt *entscheidbar*, wenn es eine nichtdeterministische Turingmaschine gibt, die genau dann auf Eingabe $w \in L$ mit Ausgabe 1 hält, wenn $w \in L$ gilt.

Die Klassen P bzw. NP sind die Mengen aller Probleme $L \subseteq \Sigma^*$, die mit einer deterministischen bzw. nichtdeterministischen Turingmaschine T in höchstens $p(n)$ Schritten entschieden werden können. Dabei ist p ein Polynom und n die Länge des Eingabeworts $w \in L$.

Unser allgemeiner Begriff der Reduzierbarkeit wird im Fall NP-vollständiger Probleme durch eine zusätzliche Anforderung an die Reduktionsabbildung eingeschränkt: Ein Problem $L_1 \subseteq \Sigma^*$ ist *polynomiell reduzierbar* auf ein Problem $L_2 \subseteq \Sigma'^*$, bezeichnet mit $L_1 \leq_p L_2$, wenn es eine deterministische Turingmaschine gibt, die in polynomieller Zeit eine Funktion $\tau : \Sigma^* \to \Sigma'^*$ berechnet, so daß gilt: $w_1 \in L_1$ genau dann, wenn $\tau(w_1) \in L_2$.

Ein Problem $L_2 \in$ NP heißt *NP-vollständig*, wenn jedes Problem $L_1 \in$ NP polynomiell reduzierbar auf L_2 ist.

Anhang B

Beweise zu den Beispielen

B.1 Korrektheit des Cocke-Kasami-Younger-Algorithmus

Lemma 7 *$G = (N, \Sigma, S, P)$ sei eine Grammatik in Chomsky-Normal-Form, $w \in \Sigma^*$ mit $n = |w|$ und $zelle$ die vom CKY-Algorithmus berechnete Funktion, dann gilt für alle $1 \leq i \leq n, 1 \leq j \leq n - i$:*

$$zelle(i, j) = \{X \in N \mid X \Rightarrow^* w_j \ldots w_{j+i-1}\}$$

Für jede Grammatik $G = (N, \Sigma, S, P)$ in Chomsky-Normalform und jedes $w \in \Sigma^+$ gilt:

Beweis durch vollständige Induktion nach i:$\mathbb{N}$

Induktionsanfang

1. $i = 1$:

   Für jedes $1 \leq j \leq n$ gilt:

   $zelle(1, j)$

   $=$ { Def. von $zelle$ }

   $\{X \in N \mid \llcorner X \to w_j \in P \lrcorner\}$

   $=$ { Teilbew. für $X \in N$ }

   Für jede Produktion $X \to w_j \in P$ gilt:

   - $X \to w_j \in P$

     $\equiv$ { Def. von $\Rightarrow^1$ }

     $X \Rightarrow^1 w_j$

     $\equiv$ { Def. von $\Rightarrow^*$ und Grammatik in CNF }

     $X \Rightarrow^* w_j$

   $\cdot$ $\{X \in N \mid \ulcorner X \Rightarrow^* w_j \urcorner\}$

Induktionsschritt von $i-1$ auf i

Für jedes $1 \leq j \leq n-i+1$ gilt:

$zelle(i,j)$

$=$ { Def. von $zelle$ }

$\bigcup_{1\leq m<i}\{X \in N \mid X \to YZ \in P, \llcorner Y \in zelle(m,j), Z \in zelle(i-m, j+m)\lrcorner\}$

$=$ { Teilbew. für $X \to YZ \in P$ und $1 \leq m < i$ }

Für jedes $1 \leq m < i$ und jede Produktion $X \to YZ \in P$ gilt:

- $Y \in zelle(m,j), Z \in zelle(i-m, j+m)$

  $\equiv$ { $m, i-m \leq n-i+1$ und Induktionsvoraussetzung }

  $Y \Rightarrow^* w_j \ldots w_{j+m-1}$, $Z \Rightarrow^* w_{j+m} \ldots w_{j+i-1}$

  $\equiv$ { Def. von $\Rightarrow$ und $X \to YZ$ }

  $X \Rightarrow^1 YZ \Rightarrow^* w_j \ldots w_{j+m-1}Z \Rightarrow^* w_j \ldots w_{j+m-1}w_{j+m} \ldots w_{j+i-1}$

  $\equiv$ { Def. von $\Rightarrow^*$ }

  $X \Rightarrow^* w_j \ldots w_{j+i-1}$

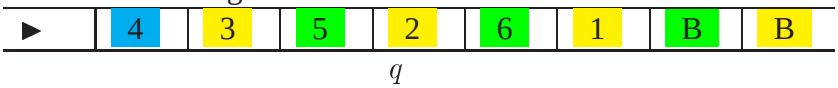
$\cdot$ $\bigcup_{1\leq m<i}\{X \in N \mid \ulcorner X \Rightarrow^* w_j \ldots w_{j+i-1}\urcorner\}$

B.2 Turingmaschinen

Wir geben hier eine genauere Schilderung des Beweises der Aussage, daß jede Berechnung einer Turingmaschine mit zweiseitig unendlichem Band sich auch auf einer Turingmaschine mit einseitig unendlichem Band durchführen läßt, genauer: Sei $T = (\Sigma, \Gamma, Q, q_0, \delta)$ eine Turingmaschine mit zweiseitig unendlichem Band, welche die Funktion $g : \Sigma^* \to \Sigma^*$ berechnet. Dann gibt es eine Turingmaschine $T_1 = (\Sigma, \Gamma, Q_1, q_0, \delta_1)$ mit einseitig unendlichen Band, welche ebenfalls g berechnet. Um diese Behauptung zu beweisen, sind einige Vorbemerkungen und Notationen notwendig, auf dem dann das zu beweisende Lemma beruht. Wir folgen im wesentlichen der Darstellung des Vorlesungsskriptums.

Die Idee des Beweises besteht darin, ein zweiseitig unendliches Band

durch das einseitig unendliche Band



zu simulieren. Dabei ist $\downarrow$ eine Markierung für die Bandzelle, die in der Anfangskonfiguration fokussiert wurde.

Formal definieren wir zu jeder Konfiguration

$$K = Bs_{-k} \ldots s_{-1} \downarrow s_1 \ldots q s_i \ldots s_k B,$$

von T die Konfiguration

$$\text{sim}(K) = \ \blacktriangleright\, s_1 s_{-1} \ldots q s_i s_{-i} \ldots s_k s_{-k} B$$

von T_1. Befindet sich der Schreib- und Lesekopf links der $\downarrow$-Markierung, also

$$K = B s_{-k} \ldots q s_{-i} \ldots s_{-1} \downarrow s_1 \ldots s_k B$$

so setzten wir

$$\text{sim}(K) = \ \blacktriangleright\, s_1 s_{-1} \ldots s_i q^{-1} s_{-i} \ldots s_k s_{-k} B$$

Das neu einzuführende Zustandssymbol q^{-1} kodiert dabei die Information, daß in der Ausgangskonfiguration q links von $\downarrow$ auftrat. Das Symbol $\downarrow$ dient allein zur Markierung der Anfangszelle. Eine Bewegung von s_1 um einen Schritt nach links führt also zu Zelle s_{-1}. Ist die Konfiguration K bezüglich der von B verschiedenen Bandbeschriftung nicht symmetrisch mit Zentrum $\downarrow$, dann füllt man die Konfiguration durch $s_i = B$ oder $s_{-i} = B$ entsprechend auf. Die Anfangskonfiguration für die Turingmaschine T_1 zur Berechnung des Funktionswertes $g(w)$ für $w = w_1 \ldots w_k$ ist jetzt

$$\text{sim}(K_0) = \ \blacktriangleright\, q_0 w_1 B \ldots w_{k-1} B w_k B$$

Ebenso ändert sich auch die Anweisung, wie der Funktionswert nach Terminierung aus der Bandinschrift abgelesen werden soll. Im einzelnen ist das schwierig zu beschreiben, aber die folgende intentionale Beschreibung vermittelt die Idee: von der fokussierten Zelle ausgehend (in der Konfiguration gekennzeichnet durch das Auftreten des Zustands q_f) liest man die Bandinschrift nach links bis zum nächsten B. Auf dem einseitigen Band heißt „lesen nach links“ auf dem rechts von $\downarrow$ stehenden Teil „lesen jedes zweiten Feldes nach links“ und auf dem links von $\downarrow$ stehenden Teil „lesen jedes zweiten Feldes nach rechts“.

Nach diesen einleitenden Erklärungen können wir jetzt zu gegebener zweiseitig unendlicher Turingmaschine $T = (\Sigma, \Gamma, Q, q_0, \delta)$ die einseitig unendliche Turingmaschine $T_1 = (\Sigma, \Gamma, Q_1, q_0, \delta_1)$ angeben:

$$\begin{aligned} Q_1 \ = \ & Q \cup \{q^{-1} \mid q \in Q\} \cup \\ & \{q_R \mid q \in Q\} \cup \{q_L \mid q \in Q\} \\ & \{q_R^{-1} \mid q \in Q\} \cup \{q_L^{-1} \mid q \in Q\} \end{aligned}$$

$$\begin{array}{lcll}
\delta_1(q, s) & = & (q_1, s_1, U) & \text{falls} \quad \delta(q, s) = (q_1, s_1, U) \\
\delta_1(q^{-1}, s) & = & (q_1^{-1}, s_1, U) & \text{falls} \quad \delta(q, s) = (q_1, s_1, U) \\
\delta_1(q, s) & = & (q_{1,R}, s_1, R) & \text{falls} \quad \delta(q, s) = (q_1, s_1, R) \\
\delta_1(q^{-1}, s) & = & (q_{1,L}^{-1}, s_1, L) & \text{falls} \quad \delta(q, s) = (q_1, s_1, R) \\
\delta_1(q, s) & = & (q_{1,L}, s_1, L) & \text{falls} \quad \delta(q, s) = (q_1, s_1, L) \\
\delta_1(q^{-1}, s) & = & (q_{1,R}^{-1}, s_1, R) & \text{falls} \quad \delta(q, s) = (q_1, s_1, L) \\
\delta_1(q_R, s) & = & (q, s, R) & \\
\delta_1(q_L, s) & = & (q, s, L) & \\
\delta_1(q_R^{-1}, s) & = & (q^{-1}, s, R) & \\
\delta_1(q_L^{-1}, s) & = & (q^{-1}, s, L) & \\
\delta_1(q_L, \blacktriangleright) & = & (q_R^{-1}, \blacktriangleright, R) & \\
\delta_1(q^{-1}, \blacktriangleright) & = & (q, \blacktriangleright, R) &
\end{array}$$

wobei stets $s \neq \blacktriangleright$ und $q, q_1 \in Q$ gilt.

Der Superskript -1 an einem Zustand zeigt an, daß sich die fokussierte Zelle des zweiseitig unendlichen Bandes links von der Anfangsmarkierung befindet, die Subskripte R und L dienen dazu, das jeweilige Zwischenfeld des einseitig unendlichen Bandes zu überbrücken.

Mit dieser Definition von T_1 kann nun das folgende Lemma bewiesen werden:

Lemma 8 (Simulationslemma)
Seien die Turingmaschinen T und T_1 und die Übersetzung sim wie oben definiert, K_n die Konfiguration, die T nach n Schritten erreicht und K_m^1 die Konfiguration, die T_1 nach m Schritten erreicht. Dann gilt für jedes $n \geq 0$:

$$\exists m \in \mathbb{N} : m \geq n \wedge \mathit{sim}(K_n) = K_m^1 \quad .$$

Beweis durch vollständige Induktion nach n:$\mathbb{N}_0$
Induktionsanfang

1. **Für** $n = 0$ **gilt:**:

   `true`

   $\Rightarrow$ { Konvention zur Berechnung eines Wertes $w_1 \ldots w_n$ }

   $K_0 = Bq_0w_1 \ldots w_nB$

   $\Rightarrow$ { Anwenden von sim }

   $\text{sim}(K_0) = \text{sim}(Bq_0w_1 \ldots w_nB)$

   $\Rightarrow$ { Def. von sim }

   $\text{sim}(K_0) = \blacktriangleright\, q_0w_1Bw_2B \ldots w_nB$

   $\Rightarrow$ { Def. von K_0^1 }

   $\text{sim}(K_0) = K_0^1$

   $\Rightarrow$ { $m := 0$ }

   $\exists m \in \mathbb{N} : m \geq n \wedge \text{sim}(K_n) = K_m^1$

Induktionsschritt von $n-1$ auf n
$K_{n-1} = Bs_{-k} \ldots s_kB$
$\Rightarrow$ { Aktueller Zustand von T sei q, Position des Kopfes $\pm i$ }
$\llcorner K_{n-1} = Bs_{-k} \ldots qs_{\pm i} \ldots s_kB \lrcorner$
$\Rightarrow$ { Teilbeweis für $[K_{n-1} = Bs_{-k} \ldots qs_{\pm i} \ldots s_kB \Rightarrow \exists m \in \mathbb{N} : m \geq n \wedge \text{sim}(K_n) = K_m^1$ }

Beweis durch Fallunterscheidung nach den Aktionen der Turingmaschine T

1. $\delta(q, s_{\pm i}) = (q_1, t, R)$:

   - $K_{n-1} = Bs_{-k} \dots qs_{\pm i} \dots s_k B$

   $\Rightarrow$ { Anwendung von δ }

   $K_{n-1} = Bs_{-k} \dots qs_{\pm i} \dots s_k B \wedge K_n = Bs_{-k} \dots tq_1 \dots s_k B$

   $\Rightarrow$ { Induktionsvoraussetzung für $i = n-1$ }

   $\llcorner \exists m' \in \mathbb{N} : m' \geq n-1 \wedge \text{sim}(Bs_{-k} \dots qs_{\pm i} \dots s_k B) = K^1_{m'} \lrcorner \wedge K_n = Bs_{-k} \dots tq_1 \dots s_k B$

   $\Rightarrow$ { Teilbeweis für $\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \text{sim}(Bs_{-k} \dots qs_{\pm i} \dots s_k B) = K^1_{m'} \Rightarrow \exists m \in \mathbb{N} : m \geq n \wedge \text{sim}(Bs_{-k} \dots tq_1 \dots s_k B) = K^1_m$ }

   Beweis durch Fallunterscheidung nach Position $\pm i > 0, \pm i < 0$ des Kopfes

   (a) $\pm i > 0$:

   - $\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \text{sim}(Bs_{-k} \dots qs_{\pm i} \dots s_k B) = K^1_{m'}$

   $\Rightarrow$ { $\pm i > 0$ nach Annahme }

   $\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \text{sim}(Bs_{-k} \dots s_{-1} \downarrow \dots qs_i \dots s_k B) = K^1_{m'}$

   $\Rightarrow$ { Def. von sim }

   $\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright \dots qs_i s_{-i} \dots s_k s_{-k} B = K^1_{m'}$

   $\Rightarrow$ { $\delta_1(q, s_i) = (q_{1,R}, t, R)$ }

   $\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright \dots tq_{1,R} s_{-i} s_{i+1} \dots s_k s_{-k} B = K^1_{m'+1}$

   $\Rightarrow$ { $\delta_1(q_{1,R}, s_{-i}) = (q_1, s_{-i}, R)$ }

   $\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright \dots ts_{-i} q_1 s_{i+1} \dots s_k s_{-k} B = K^1_{m'+2}$

   $\Rightarrow$ { $m := m' + 2$ und $m \geq n$ }

   $\exists m \in \mathbb{N} : m \geq n \wedge \blacktriangleright \dots tq_1 s_{i+1} \dots s_k s_{-k} B = K^1_m$

   $\Rightarrow$ { Def. von sim }

   $\exists m \in \mathbb{N} : m \geq n \wedge \text{sim}(Bs_{-k} \dots tq_1 s_{i+1} \dots s_k B) = K^1_m$

   (b) $\pm i < 0$:

   $\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \text{sim}(Bs_{-k} \dots qs_{\pm i} \dots s_k B) = K^1_{m'}$

   $\Rightarrow$ { $\pm i < 0$ nach Annahme }

   $\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \text{sim}(Bs_{-k} \dots qs_{-i} \dots \downarrow s_1 \dots s_k B) = K^1_{m'}$

   $\Rightarrow$ { Def. von sim }

   $\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright \dots s_i q^{-1} s_{-i} \dots s_k s_{-k} B = K^1_{m'}$

   $\Rightarrow$ { $\delta_1(q^{-1}, s_{-i}) = (q^{-1}_{1,L}, t, L)$ }

   $\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright \dots q^{-1}_{1,L} s_i t \dots s_k s_{-k} B = K^1_{m'+1}$

$\Rightarrow$ { Teilbeweis für $[\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright \ldots q_{1,L}^{-1} s_i t \ldots s_k s_{-k} B = K^1_{m'+1} \Rightarrow \exists m \in \mathbb{N} : m \geq n \wedge \blacktriangleright \ldots q_1^{-1} s_{-(i-1)} s_{i-1} t \ldots s_k s_{-k} B = K^1_m]$ }

Beweis durch Fallunterscheidung nach Position $\pm i < -1, \pm i = -1$ des Kopfes

b.1. $\pm i < -1$:

- $\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright \ldots q_{1,L}^{-1} s_i t \ldots s_k s_{-k} B = K^1_{m'+1}$

$\Rightarrow$ { $\delta_1(q_{1,L}^{-1}, s_{-(i-1)}) = (q_1^{-1}, s_{-(i-1)}, L)$ }

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright \ldots q_1^{-1} s_{-(i-1)} s_{i-1} t \ldots s_k s_{-k} B = K^1_{m'+2}$

$\Rightarrow$ { $m := m'+2$ und $m \geq n$ }

$\exists m \in \mathbb{N} : m \geq n \wedge \blacktriangleright \ldots q_1^{-1} s_{-(i-1)} s_{i-1} t \ldots s_k s_{-k} B = K^1_m$

b.2. $\pm i = -1$:

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright q_{1,L}^{-1} s_1 t s_2 \ldots s_k s_{-k} B = K^1_{m'+1}$

$\Rightarrow$ { $\delta_1(q_{1,L}^{-1}, s_1) = (q_1^{-1}, s_1, L)$ }

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge q_1^{-1} \blacktriangleright s_1 t s_2 \ldots s_k s_{-k} B = K^1_{m'+2}$

$\Rightarrow$ { $\delta_1(q_1^{-1}, \blacktriangleright) = (q_1, \blacktriangleright, R)$ }

$\exists m \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright q_1 s_1 t s_2 \ldots s_k s_{-k} B = K^1_{m'+3}$

$\Rightarrow$ { $m := m'+3$ und $m \geq n$ }

$\exists m \in \mathbb{N} : m \geq n \wedge \blacktriangleright q_1 s_1 t s_2 \ldots s_k s_{-k} B = K^1_m$

- $\ulcorner \exists m \in \mathbb{N} : m \geq n \wedge \blacktriangleright s_1 s_{-1} \ldots t s_{-i} q_1 \ldots s_k s_{-k} B = K^1_m \urcorner$

$\Rightarrow$ { Def. von sim }

$\exists m \in \mathbb{N} : m \geq n \wedge \text{sim}(B s_{-k} \ldots t q_1 \ldots s_k B) = K^1_m$

- $\ulcorner \exists m \in \mathbb{N} : m \geq n \wedge \text{sim}(B s_{-k} \ldots t q_1 \ldots s_k B) = K^1_m \urcorner \wedge K_n = B s_{-k} \ldots t q_1 \ldots s_k B$

$\Rightarrow$ { Einsetzen von K_n und Abschwächung }

$\exists m \in \mathbb{N} : m \geq n \wedge \text{sim}(K_n) = K^1_m$

2. $\delta(q, s_{\pm i}) = (q_1, t, L)$:

$K_{n-1} = B s_{-k} \ldots q s_{\pm i} \ldots s_k B$

$\Rightarrow$ { Anwendung von δ }

$K_{n-1} = B s_{-k} \ldots q s_{\pm i} \ldots s_k B \wedge K_n = B s_{-k} \ldots q_1 s_{\pm(i-1)} t \ldots s_k B$

$\Rightarrow$ { Induktionsvoraussetzung für $n-1$ }

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \text{sim}(B s_{-k} \ldots q s_{\pm i} \ldots s_k B) = K^1_{m'}$

$\Rightarrow$ { Teilbeweis für $[\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \text{sim}(B s_{-k} \ldots q s_{\pm i} \ldots s_k B) = K^1_{m'}] \Rightarrow [\exists m \in \mathbb{N} : m \geq n \wedge \text{sim}(B s_{-k} \ldots q_1 s_{\pm i-1} t \ldots s_k B) = K^1_m]$ }

Beweis durch Fallunterscheidung nach Position $\pm i > 0, \pm i < 0$ des Kopfes

(a) $\pm i > 0$:

- $\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \text{sim}(Bs_{-k} \dots qs_{\pm i} \dots s_k B) = K^1_{m'}$

$\Rightarrow$ { $\pm i > 0$ nach Annahme }

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \text{sim}(Bs_{-k} \dots s_{-1} \downarrow \dots qs_i \dots s_k B) = K_{m'}$

$\Rightarrow$ { Def. von sim }

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright \dots qs_i \dots s_k s_{-k} B = K_{m'}$

$\Rightarrow$ { Teilbeweis für $[\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright \dots qs_i \dots s_k s_{-k} B = K_{m'} \Rightarrow \exists m \in \mathbb{N} : m \geq n \wedge \blacktriangleright \dots s_{-1} \dots q_1 s_{i-1} s_{-i} t \dots s_k s_{-k} B = K_m$ }

Beweis durch Fallunterscheidung nach Position $\pm i > 1, \pm i = 1$ des Kopfes

a.1. $\pm i > 1$:

- $\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright \dots qs_i \dots s_k s_{-k} B = K_{m'}$

$\Rightarrow$ { $\delta_1(q, s_i) = (q_{1,L}, t, L)$ }

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright \dots q_{1,L} s_{-i} t \dots s_k s_{-k} B = K_{m'+1}$

$\Rightarrow$ { $\delta_1(q_{1,L}, s_{-i}) = (q_1, s_{-i}, L)$ }

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright \dots s_{-1} \dots q_1 s_{i-1} s_{-i} t \dots s_k s_{-k} B = K_{m'+2}$

$\Rightarrow$ { $m := m' + 2$ und $m \geq n$ }

$\exists m \in \mathbb{N} : m \geq n \wedge \blacktriangleright \dots s_{-1} \dots q_1 s_{i-1} s_{-i} t \dots s_k s_{-k} B = K_m$

a.2. $\pm i = 1$:

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright qs_1 q_{-1} \dots s_k s_{-k} B = K_{m'}$

$\Rightarrow$ { $\delta_1(q, s_1) = (q_{1,L}, t, L)$ }

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge q_{1,L} \blacktriangleright ts_{-1} \dots s_k s_{-k} B = K_{m'+1}$

$\Rightarrow$ { $\delta_1(q_{1,L}, \blacktriangleright) = (q^{-1}_{1,R}, \blacktriangleright, R)$ }

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright q^{-1}_{1,R} t \dots s_k s_{-k} B = K_{m'+2}$

$\Rightarrow$ { $\delta_1(q^{-1}_{1,R}, t) = (q_1^{-1}, t, R)$ }

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright q^{-1} s_{-i} t \dots s_k s_{-k} B = K_{m'+3}$

$\Rightarrow$ { $m := m' + 3$ und $m = m \geq n$ }

$\exists m \in \mathbb{N} : m \geq n \wedge \blacktriangleright tq^{-1} s_{-1} \dots s_{-k} s_k B = K_m$

$\cdot$ $\ulcorner \exists m \in \mathbb{N} : m \geq n \wedge \blacktriangleright tq^{-1} s_{-1} \dots s_{-k} s_k B = K_m \urcorner$

$\Rightarrow$ { Def. von sim }

$\exists m \in \mathbb{N} : m \geq n \wedge \text{sim}(Bs_{-k} \ldots tq^{-1}s_{-1} \downarrow s_1 \ldots s_k B) = K_m$

(b) $\pm i < 0$:

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \text{sim}(Bs_{-k} \ldots qs_{\pm i} \ldots s_k B) = K^1_{m'}$

$\Rightarrow$ { $\pm i < 0$ nach Annahme }

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \text{sim}(Bs_{-k} \ldots qs_{-i} \ldots \downarrow s_1 \ldots s_k B) = K^1_{m'}$

$\Rightarrow$ { Def. von sim }

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright \ldots q^{-1}s_{-i} \ldots s_k s_{-k} B = K^1_{m'}$

$\Rightarrow$ { $\delta_1(q^{-1}, s_{-i}) = (q^{-1}_{1,R}, t, R)$ }

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright \ldots tq^{-1}_{1,R} s_{i+1} s_{-(i+1)} \ldots s_k s_{-k} B = K^1_{m'+1}$

$\Rightarrow$ { $\delta_1(q^{-1}_{1,R}, s_{i+1}) = (q_1^{-1}, s_{i+1}, R)$ }

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright \ldots ts_{i+1}q_1^{-1} s_{-(i+1)} \ldots s_k s_{-k} B = K^1_{m'+2}$

$\Rightarrow$ { $m := m'+2$ und $m \geq n$ }

$\exists m \in \mathbb{N} : m \geq n \wedge \blacktriangleright \ldots ts_{i+1}q_1^{-1} s_{-(i+1)} \ldots s_k s_{-k} B = K^1_m$

$\Rightarrow$ { Def. von sim }

$\exists m \in \mathbb{N} : m \geq n \wedge \text{sim}(s_{-k} \ldots q_1^{-1} s_{-(i+1)} t \ldots s_k B) = K^1_m$

· $\ulcorner \exists m \in \mathbb{N} : m \geq n \wedge \text{sim}(s_{-k} \ldots q_1^{-1} s_{\pm i-1} t \ldots s_k B) = K^1_m \urcorner \wedge K_n = Bs_{-k} \ldots q_1 s_{\pm(i-1)} t \ldots s_k B$

$\Rightarrow$ { Einsetzen von K_n und Abschwächung }

$\exists m \in \mathbb{N} : m \geq n \wedge \text{sim}(K_n) = K_m$

3. $\delta(q, s_{\pm i}) = (q_1, t, U)$:

$K_{n-1} = Bs_{-k} \ldots qs_{\pm i} \ldots s_k B$

$\Rightarrow$ { Anwendung von δ }

$K_{n-1} = Bs_{-k} \ldots qs_{\pm i} \ldots s_k B \wedge K_n = Bs_{-k} \ldots q_1 t \ldots s_k B$

$\Rightarrow$ { Induktionsvoraussetzung für $n-1$ }

$\llcorner \exists m' \in \mathbb{N} : m' \geq n-1 \wedge \text{sim}(Bs_{-k} \ldots qs_{\pm i} \ldots s_k B) = K^1_{m'} \lrcorner \wedge K_n = Bs_{-k} \ldots q_1 t \ldots s_k B$

$\Rightarrow$ { Teilbeweis für $[\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \text{sim}(Bs_{-k} \ldots qs_{\pm i} \ldots s_k B) = K^1_{m'} \Rightarrow \exists m \in \mathbb{N} : m \geq n \wedge \text{sim}(Bs_{-k} \ldots q_1 t \ldots s_k B) = K^1_m]$ }

Beweis durch Fallunterscheidung nach $\pm i > 0, \pm i < 0$

(a) $\pm i > 0$:

• $\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \text{sim}(Bs_{-k} \ldots qs_{\pm i} \ldots s_k B) = K^1_{m'}$

$\Rightarrow$ { $\pm i > 0$ nach Annahme }

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \text{sim}(Bs_{-k}\ldots s_{-1} \downarrow qs_i\ldots s_kB) = K^1_{m'}$

$\Rightarrow$ { Def. von sim }

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright \ldots qs_i\ldots s_{-k}s_kB = K^1_{m'}$

$\Rightarrow$ { $\delta_1(q,s_i) = (q_1,t,U)$ }

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright \ldots q_1t\ldots s_{-k}s_kB = K^1_{m'+1}$

$\Rightarrow$ { $m := m'+1$ und $m \geq n$ }

$\exists m \in \mathbb{N} : m \geq n \wedge \blacktriangleright \ldots q_1t\ldots s_{-k}s_kB = K^1_m$

(b) $\pm i < 0$:

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \text{sim}(Bs_{-k}\ldots qs_{\pm i}\ldots s_kB) = K^1_{m'}$

$\Rightarrow$ { $\pm i < 0$ nach Annahme }

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \text{sim}(Bs_{-k}\ldots qs_{-i} \downarrow s_1\ldots s_kB) = K^1_{m'}$

$\Rightarrow$ { Def. von sim }

$\exists m' \in \mathbb{N} : m' \geq n-1 \wedge \blacktriangleright \ldots q^{-1}s_{-i}\ldots s_{-k}s_{-k}B = K^1_{m'}$

$\Rightarrow$ { $\delta_1(q^{-1},s_{-i}) = (q_1^{-1},t,U)$ }

$\exists m' \in \mathbb{N} : m'-1 \geq n-1 \wedge \blacktriangleright \ldots q_1^{-1}t\ldots s_{-k}s_{-k}B = K^1_{m'+1}$

$\Rightarrow$ { $m := m'+1$ und $m \geq n$ }

$\exists m \in \mathbb{N} : m \geq n \wedge \blacktriangleright \ldots q_1^{-1}t\ldots s_{-k}s_kB = K^1_m$

- $\ulcorner \exists m \in \mathbb{N} : m \geq n \wedge \text{sim}(Bs_{-k}\ldots q_1t\ldots s_kB) = K^1_m \urcorner \wedge K_n = Bs_{-k}\ldots q_1t\ldots s_kB$

$\Rightarrow$ { Einsetzen von K_n und Abschwächung }

$\exists m \in \mathbb{N} : m \geq n \wedge \text{sim}(K_n) = K^1_m$

- $\ulcorner \exists m \in \mathbb{N} : m \geq n \wedge \text{sim}(K_n) = K^1_m \urcorner$

Anhang C

Werkzeuge

C.1 cproof2java

Name cproof2java – erzeugt Java-Quellkode aus Beweisen im *calculational proof format*

Synopsis cproof2java [options] file(s)

Beschreibung cproof2java erzeugt ein Java Kodefragment aus einem Beweis im *caculational proof format*, der in HTML mit Hilfe zusätzlicher Pseudo Tags geschrieben ist, siehe AbschnittD.2 für eine Beschreibung dieser Tags. cproof2java benutzt perl. Anstatt den Beweis direkt in HTML, kann man ihn auch in LaTeX mit dem cproof-Stil schreiben (siehe Abschnitt D.1) und ihn mit LaTeX2HTML in das von cproof2java geforderte HTML Format konvertieren.

Optionen

-extends <classname> Spezifiziert den Namen der Oberklasse aus dem die erzeugten Java Klasse abgeleitet wird. Voreinstellung ist 'java.applet.Applet'. Falls <classname> leer ist, dann beitzt die erzeugte Java Klasse keine Oberlasse.

-configfile <file> Spezifiziert den Name der Konfigurationsdatei. Voreinstellung ist es, .\.cproof-init zu laden, falls sie existiert, oder die Konfigurationdatei aus dem Installationverzeichnis zu laden, falls sie nicht existiert.

-d <dir> Spezifiziert das Verzeichnis in dem die Java Klasse geschrieben wird. Voreinstellung ist \. . Diese Option wird ignoriert, falls kein Verzeichnis angegeben wird.

-o <file> Spezifiziert den Namen der erzeugten Java Klasse. Voreinstellung ist es, den Quelltext der Klasse auf die Standard Ausgabe zu schreiben.

C.2 cproof2html

Name cproof2html – konvertiert spezielle Pseudo Tags nach HTML.

Synopsis cproof2html [options] file

Beschreibung cproof2html konvertiert einen Beweis im *calculational proof format*, der mit zusätzlichen Pseudo Tags in HTML geschrieben ist in (fast) reines HTML. cproof2html benutzt perl. Nur die speziellen Pseudo Tags werden durch HTML Kode ersetzt. Alles anderen wird einfach übernommen. cproof2html gibt reinen HTML Kode aus mit der Ausnahmen, daß die geschachtelte Teilbeweise mit dem Pseudo Tag <FOLDME>..</FOLDME> markiert sind. Benutzen Sie fold, um einen stöberbaren HTML Beweistext zu erzeugen.

Optionen

-configfile <file> Spezifiziert den Name der Konfigurationsdatei. Voreinstellung ist es, .\.cproof-init zu laden, falls sie existiert, oder die Konfigurationdatei aus dem Installationverzeichnis zu laden, falls sie nicht existiert.

-d <dir> Spezifiziert das Verzeichnis, in dem die erzeugte HTML Datei geschrieben wird. Voreinstellung ist \. .

-o <file> Spezifiziert den Name der erzeugten HTML Datei Voreinstellung ist <file>p.<ext> wobei <file>.<ext> der Name der Eingabedatei ist.

C.3 fold

Name fold – erzeugt stöberbares HTML für geschachtelte Teilbeweise

Synopsis fold [options] file(s)

Beschreibung fold konvertiert eine HTML Datei, bei der die geschachtelte Teilbeweise mit dem Pseudo Tag <FOLDME>..</FOLDME> markiert wurden, in stöberbares HTML. fold benutzt perl. fold erzeugt drei Dateien: Den HTML Beweistext (proof.html), eine HTML Datei mit einem Applet, welches die Beweisanimation erzeugt (animapplet.html), und eine dritte, die zwei HTML Rahmen enthält, in welchem jeweils die beiden anderen Dateien eingebettet sind (index.html).

Optionen

`-applet` Fügt das Applet am Ende des HTML Beweistextes (`proof.html`) an. Voreinstellungsmäßig wird das Applet nicht am Ende der Datei gehängt.

`-appparam <string>` Spezifiziert die Parameter des Applet Tags. Voreinstellung ist

```
'CODEBASE="../.." CODE="Proof/Anim.class WID-
                HT=95 HEIGHT=30' .
```

`-appletname <string>` Spezifiziert den Namen des Applet. Voreinstellung ist `animapplet`.

`-configfile <file>` Spezifiziert den Name der Konfigurationsdatei. Voreinstellung ist es, `.\.cproof-init` zu laden, falls sie existiert, oder die Konfigurationdatei aus dem Installationverzeichnis zu laden, falls sie nicht existiert.

`-d <dir>` Spezifiziert das Verzeichnis, in welches die Ausgabedateien geschrieben werden. Voreinstellung ist `\.` .

`-extension <string>` Spezifiziert die Dateierweiterung der Erzeugten HTML Dateien. Voreinstellung ist `.html` .

`-foldtag <string>` Spezifiziert den Namen des Pseudo Tags, der zur Markierung der Teilbeweise verwendet wurde. Voreinstellung ist `FOLDME`.

`-nojavascript` Erzeuge für jede mögliche Kombination sichtbarer und verdeckter Teilbeweise jeweils eine Datei ohne JavaScript Kommandos. Voreinstellung ist es, nur ein Datei zu erzeugen und den jeweiligen Status sichtbarer und verdeckter Teile mit JavaScript und mit Hilfe der HTML Rahmendatei zu erzeugen.

`-noframes` Es wird keine Datei mit HTML Rahmen erzeugt. Voreinstellung ist diese Datei zu erzeugen. Diese Option macht nur Sinn, wenn `-nojavascript` gesetzt wird.

`-noscript` Füge keine JavaScript Kommandos am Ende jedes HTML Beweistextes an. Diese Kommandos werden für die Kommunikation mit dem Applet verwendet. Voreinstellung ist es, die Kommandos zu erzeugen.

`-numbering` Die Namen der HTML Beweisdateien werden aus dem Name `<file>.<text>` der Eingabedatei durch Anfügen einer fortlaufenden Numerierung am Ende von `<file>`. Die Numerierung startet mit 1 (`<file>1.html`). Die Datei `<file>0.html` ist der HTML Beweistext, bei dem alle Teilbeweise verdeckt sind. Falls diese Option nicht angegeben ist, werden die Namen der HTML Dateien aus den in ihnen enthaltenen Teilbeweisen erzeugt: Ein `H` für einen verdeckten Teilbeweis, ein `V` für einen sichtbaren Teilbeweis. `VVH.html` ist der Name der HTML Datei, in welche die ersten beiden Teilbeweise sichtbar sind und der dritte verdeckt. Die Namen geben nicht die hierarchische Struktur der Teilbeweis wieder, es ist lediglich eine falsche Repräsentation des Zustands der Teilbeweise. Diese Option macht nur Sinn, wenn `-nojavascript` gesetzt wird.

Anhang D

LaTeX Stil-Dateien und HTML-Pseudo-Tags

D.1 `cproof`-Stil

Dieser Abschnitt beschreibt LaTeX-Kommandos und -Umgebungen, mit denen Beweise im *Structured calculational proof format* in LaTeX gesetzt werden können. Java-spezifische Informationen sind dabei ausschließlich in optionalen Parametern unterzubringen, um die Kommandos auch ohne das Ziel einer Programmsynthese benutzen zu können. Wir benutzen das in [LAMPORT 1986] verwendete Format, um die LaTeX-Kommandos und -Umgebungen zu beschreiben:

> Everything in typewriter font, such as "`\newcommand{`" represents material that appear in the input file exactly as shown. The italicized parts *cmd*, *args*, and *def* represent items that vary; the command's description explains their function. Arguments enclosed in square brackets [ ] are optional; they (and the brackets) may be omitted" [LAMPORT 1986, Seite 150]

Optionale Parameter verweisen im `cproof`-Stil auf Java-spezifischen Kode, der zur Erzeugung der Programmteile für den synthetisierten Algorithmus notwendig ist, die sich nicht aus den LaTeX-Anweisungen erzeugen lassen. Die meisten der im folgenden beschriebenen Kommandos und Umgebungen starten mit einem `c` (für `c`alculational).

D.1.1 Markieren des Beweises

`\begin{cproof}`[*package*] *proof* `\end{cproof}`

Diese Umgebung markiert den Anfang und das Ende eines Beweises. Alle Anweisungen außerhalb dieser Umgebung werden von SCAPAs Werkzeugen ignoriert. *package* ist ein Bezeichner im Format [a–zA–Z_0–9]+. Es gibt den Namen des Java-*package* an, der den synthetisierten Javakode enthält. Voreinstellung ist `Proof`.

D.1.2 Boolesche Strukturen

`\begin{cstructure}` *structure* `\end{cstructure}`

Jede der im Beweis verwendeten Boolesche Strukturen muß mit dieser Umgebung markiert werden. Die Boolesche Struktur *structure* darf keine anderen Kommandos oder Umgebungen des `cproof`-Stil enthalten, mit Ausnahme der Kommandos `\cexpand`, `\cresult`, `\ctrue` und `\cfalse`. Die letzten beiden produzieren lediglich `true` und `false`. `\cexpand` markiert die Teilstruktur, die in einem nachfolgenden geschachtelten Teilbeweis transformiert wird. Die resultierende Boolesche Struktur wird in der dem geschachtelten Teilbeweise nachfolgenden Booleschen Struktur mit `\cresult` markiert.

D.1.3 Beweisschritt

`\begin{cbecause}`{*op*} *hint* `\end{cbecause}`

Mit dieser Umgebung wird ein Hinweis zu einem Beweisschritt markiert, der sich zwischen Booleschen Strukturen oder einer Booleschen Struktur und einem geschachtelten Teilbeweis befindet. *op* gibt die Art der Beweisregel an, etwa $=$ bzw. $\equiv$ (Äquivalenz) oder $\Rightarrow$ (Implikation). *hint* ist der Kommentar für diesen Beweisschritt.

D.1.4 Teilbeweise

`\begin{csubproof}` *subproof* `\end{csubproof}`

Ähnlich zur Markierung eines ganzen Beweises, umschließt die `csubproof`-Umgebung einen geschachtelten Teilbeweis. Der Teilbeweis *subproof* sollte sollte mit einer Booleschen Struktur beginnen und enden. Manchmal ist es allerdings vorteilhaft, zu Beginn des Teilbeweises einige informelle Erklärungen voranzustellen. Ein Teilbeweis kann selbst wieder einen Teilbeweis enthalten.

Die Boolesche Struktur, die während des Teilbeweises transformiert wird, sollte ein Teilstruktur der dem Teilbeweis vorausgehenden Booleschen Struktur sein. Der transformierte Teil dieser Struktur sollte mit dem `\cexpand`-Kommando kenntlich gemacht werden. In der gleichen Weise, sollte die letzte Boolesche Struktur des Teilbeweises eine Teilstruktur der dem Teilbeweis nachfolgenden Booleschen Struktur sein und mit dem `\cresult`-Kommando kenntlich gemacht werden.

D.1.5 Quantifizierung

Jeder Beweis oder Teilbeweis kann eine oder mehrere Variablen enthalten, die universell bzw. existenziell quantifiziert sind. Ein so markiert Beweis ist ein Beweis für alle

bzw. eine Instanz der quantifizierten Variable. Im synthetisierten Javakode wird für jede quantifizierte Variable eine Java-Variable erzeugt. Dessen Javatyp kann nicht automatisch ermittelt werden und muß deswegen vom Autor angegeben werden. Zusätzlich ist vom Autor anzugeben, wie eine universell quantifizierte Variable im Programm behandelt wird: durch eine Aufzählung aller Instanzen aus dem quantifizierten Bereich, durch Angabe einer Instanz zu Beginn des Algorithmus oder durch einen konstanten Wert. Um später in optionalen Parametern auf die Variable zugreifen zu können, kann ein eindeutiger Bezeichner angegeben werden.

Universelle Quantifizierung

`\begin{cforall}`[*type*][*iterate*][*ref*]{*var*}{*domain*} *proof* `\end{cforall}`

var ist der Name der Variable, so wie sie im markierten Beweis verwendet wird. Die Variable ist über den angegebenen Bereich *domain* quantifiziert.

Der optionale Typ *type* gibt den Typ der zugehörigen Java-Variablen an. Voreinstellung ist `int`. Mit *iterate* gibt der Autor an, wie die Quantifizierung im Javakode implementiert wird:

- `user`: Der Wert der Variablen wird zu Beginn der Programmausführung angegeben. Vorzugsweise sollte der Autor des Algorithmus und der Animation dem späteren Benutzer diesen Wert, etwa über ein Dialogfenster, eingeben zu lassen. Alternativ kann ein konstanter Wert im Algorithmus verwendet werden.
- `for=value;expr;next`: Alle Instanzen der Variable werden mit einer `for`-Schleife aufgezählt. Dabei ist `value` der initiale Wert der Java-Variablen, `expr` ist ein Boolescher Ausdruck, der genau dann wahr wird, wenn die Schleife abbrechen soll (alle Werte aufgezählt sind). Die jeweils nächste Instanz der Java-Variablen wird in einer Zuweisung innerhalb von `next` erzeugt.

Voreinstellung für `iterate` ist `user`. Das dritte optionale Argument *ref* spezifiziert einen eindeutigen Bezeichner, etwa `x`, für die Java-Variable. Er kann benutzt werden, um mit `<x>` innerhalb von `expr` oder allen anderen optionalen Argumenten auf die Java-Variable zu verweisen.

Existenzielle Quantifizierung

`\begin{cexists}`[*type*][*ref*]{*var*}{*domain*} *proof* `\end{cexists}`

var ist der Name der Variable, so wie sie im markierten Beweis verwendet wird. Die Variable ist über den angegebenen Bereich *domain* quantifiziert.

Der optionale Typ *type* gibt den Typ der zugehörigen Java-Variablen an. Voreinstellung ist `int`. Das zweite optionale Argument *ref* spezifiziert einen eindeutigen Bezeichner, etwa `x`, für die Java Variable. Er kann benutzt werden, um mit `<x>` innerhalb von `expr` oder allen anderen optionalen Argumenten auf die Java-Variable zu verweisen.

D.1.6 Induktion

Beruht ein Beweis auf vollständiger Induktion, dann müssen der zur Induktion gehörige Teil, der Basisfall sowie der Induktionsschritt markiert werden. Die in der Induktion verwendete Induktionsvariable wird, ähnlich zu einer universell quantifizierten Variable, mit einer `for`-Schleife aufgezählt.

Induktion

`\begin{cinduction}`[*type*][*for=expr;cond;state*]{*var*}{*domain*}
proof `\end{cinduction}`

var ist der Name der Variable, so wie sie im markierten Beweis verwendet wird. Die Variable ist über den angegebenen Bereich *domain* quantifiziert.

Der optionale Typ *type* gibt den Typ der zugehörigen Java-Variablen an. Voreinstellung ist `int`. Mit `<i>` kann auf den aktuellen Wert der Induktionsvariablen verwiesen werden. Mit `<n>` kann auf den ursprünglichen Werte der Induktionsvariablen verwiesen werden. Der zweite optional Parameter `for=value;expr;next` gibt an, wie der Induktionsverlauf implementiert wird: Die Werte der Variable werden mit einer `for`-Schleife aufgezählt. `value` spezifiziert den initialen Wert der Java-Variablen. `expr` ist ein Boolescher Ausdruck, der genau dann wahr wird, wenn die Schleife stoppen soll. Die jeweils nächste Instanz der Java-Variablen wird in einer Zuweisung innerhalb von `next` erzeugt. Voreinstellung für `for=value;expr;next` ist `for=0;<i> < <n>;<i>++`

Basisfall

`\begin{cbasecase}[{\it expr}]` *cases* `\end{cbasecase}`

Der Basisfall *muß* vor dem Induktionsschritt angegeben werden. *expr* ist ein Boolescher Ausdruck, der genau dann wahr sein soll, wenn der aktuelle Zustand der Induktionsvariablen durch einen Basisfall behandelt wird. Man kann `<i>` in *expr* auf den aktuellen Wert der Variablen verweisen. (und analog auf andere quantifizierte Variablen). Voreinstellung von *expr* ist `<i>==0`. *cases* enthält alle Basisfälle. Jeder Basisfall enthält einen Teilbeweis, der mit `case` markiert sein muß.

`\begin{case}`[*expr*]{*thecase*} *proof* `\end{case}`

Das optionale Argument *expr* ist ein Boolescher Ausdruck, der genau dann wahr sein sollte, wenn der aktuelle Zustand der Induktionsvariable vom zugehörigen Basisfall behandelt wird. Man kann mit `<i>` in *expr* auf den aktuellen Wert der Induktionsvariablen zugreifen. Voreinstellung von *expr* ist `<i>==0`. *thecase* ist ein LaTeX-Text der dem Ausdruck `expr` entspricht. *proof* ist der Beweis des Basisfalls (ohne umschließende `proof`- oder `subproof`-Umgebung).

Induktionsschritt

`\begin{cstepcase}`{*reduction*} *proof* `\end{cstepcase}`

Ein Induktionsbeweis muß genau einen Induktionsschritt enthalten. Der erste Parameter *reduction* ist eine kurze Beschreibung, wie die Induktionsvariable reduziert wird. *proof* ist der Beweis des Induktionsschrittes.

Verwendung der Induktionshypothese

`\begin{cbecauseind}`[*expr*]{*op*} *hint* `\end{cbecauseind}`

Diese Umgebung sollte anstatt der `cbecause`-Umgebung zur Angabe eines Beweisschrittes verwendet werden, der von der Induktionshypothese Gebrauch macht. Das optional Argument *expr* ist ein Java Ausdruck, der den reduzierten Wert aus dem Induktionsparameter berechnet (zum Beispiel `<n>-1`). Dieser Parameter wird derzeit ignoriert. Er kann aber vielleicht in zukünftigen Version für eine rekursive Implementierung der Induktion verwendet werden. *op* zeigt die Art des Beweisschrittes an, etwa $=$ bzw- $\equiv$ (Äquivalenz) oder $\Rightarrow$ (Implikation) *hint* ist der Kommentar für diesen Beweisschritt.

D.1.7 Fallunterscheidung

`\begin{ccase}`*cases proof* `\end{ccases}`

Ein Fallunterscheidung im Beweis muß mit dieser Umgebung markiert werden. *cases* ist eine kurze Angabe aller in *proof* aufgezählten Fälle. Jeder einzelne Fall in *proof* besteht aus einem separaten Beweis, der mit der Umgebung `case` markiert werden muß.

`\begin{case}`[*expr*]{*thecase*} *proof* `\end{case}`

Das optionale Argument *expr* ist ein Boolescher Ausdruck, der genau dann wahr sein sollte, wenn der aktuelle Zustand des Beweises vom zugehörigen Fall behandelt wird. Man kann in *expr* auf die Java-Variablen wie oben beschrieben zugreifen. *thecase* ist ein LaTeX-Text der dem Ausdruck `expr` entspricht. *proof* ist ein Beweis des Basisfalls (ohne umschließende `proof`- oder `subproof`-Umgebung).

D.2 HTML-Pseudo-Tags

Anstatt den Beweis von LaTeX nach HTML zu konvertieren, kann er auch direkt in HTML gesetzt werden. Mit einem speziellen HTML-Pseudo-Tags werden dabei die Informationen über den Beweis markiert. Der Pseudo-Tag besitzt folgendes Format:

<CPROOF CMD=command *arguments*> *text* </CPROOF>

`command` spezifiziert den Typ (Boolesche Struktur, Hinweis, Induktion, etc) des umschlossenen Textes. Jedes Kommando kann einige zusätzliche Argumente besitzen. Die folgenden Auflistung zeigt alle Kommandos mit ihren Argumenten in der gleichen Reihenfolge wie ihre zugehörigen LaTeX-Befehle.

D.2.1 CPROOF

<CPROOF CMD=CPROOF PACKAGE=*package*> *proof* </CPROOF>

Dieser Tag markiert den Anfang und das Ende eines Beweises. Alles außerhalb wird von SCAPAs Werkzeugen ignoriert. Das optionale Argument *package* ist ein Bezeichner im Format [a–zA–Z_0–9]+. Er gibt den Namen des Java package des Java Kodefragments für diesen Beweis an. Voreinstellung ist `Proof`.

D.2.2 STRUCTURE

<CPROOF CMD=STRUCTURE ID=*id*> *structure* </CPROOF>

Jede Boolesche Struktur *structure* im Beweis muß mit diesem Tag markiert werden. `structure` darf keine anderen Pseudo-Tags enthalten.

Der (nicht optionale) Parameter *id* ist ein eindeutiger Bezeichner. Basierend auf diesem Bezeichner erzeugt `cproof2java` zwei Java Methoden, die als Rahmen für die Implementierung der Booleschen Struktur im Algorithmus dienen:

- `structure`*id*: Dies implementiert die Boolesche Struktur.
- `truthValueOfstructure`*id*: Hiermit wird der Wahrheitswert der Booleschen Struktur zurückgegeben. Voreinstellung ist es, den Wahrheitswert der zuletzt ausgeführten Booleschen Struktur zurückzugeben. Diese Methode sollte nur dann überschrieben werden, wenn sich der Wahrheitswert während der Ausführung der Booleschen Struktur `structure`*id* ändern kann.

D.2.3 BECAUSE

<CPROOF CMD=BECAUSE ID=*id*> *hint* </CPROOF>

Ein Hinweis zu einem Beweisschritt wird immer zwischen zwei Booleschen Strukturen oder einer Booleschen Struktur und einem geschachtelten Teilbeweis plaziert. *hint* ist der Hinweis für den Beweisschritt. Der Operator für diesen Beweisschritt und die Klammern, die den Kommentar einschließen müssen in HTML manuell angegeben werden.

Der (nicht optionale) Parameter *id* ist ein eindeutiger Bezeichner. Basierend auf diesem Bezeichner erzeugt `cproof2java` eine Java Methode, die als Rahmen für die Implementierung der Booleschen Struktur im Algorithmus dient:

- `because`*id*: Implementiert die Transformation der vorangestellten Booleschen Struktur in die nachfolgende.

D.2.4 SUBPROOF

`<CPROOF CMD=subproof>` *subproof* `</CPROOF>`

Der geschachtelte Teilbeweis *subproof* sollte mit einer Booleschen Struktur beginnen und in einer enden. Manchmal ist es allerdings sinnvoll mit einigen informellen Erklärungen zu beginnen. Der Teilbeweis kann selbst wieder eine beliebige Anzahl geschachtelter Teilbeweise enthalten.

D.2.5 Quantifizierung

`FORALL`

`<CPROOF CMD=FORALL TYPE=`*type* `ITERATE=`*it* `REF=`*ref*`>` *proof* `</CPROOF>`

Der optionale Parameter `type` gibt den Typen der zugehörigen Java-Variablen im Java Kode Fragment an. Voreinstellung ist `int`. Mit *it* spezifiziert der Autor, wie die Quantifizierung im Kode Fragment implementiert wird:

- `user`: Der Wert der Variablen wird zu Beginn der Programmausführung angegeben. Vorzugsweise sollte der Autor des Algorithmus und der Animation dem späteren Benutzer diesen Wert, etwa über ein Dialogfenster, eingeben zu lassen. Alternativ kann ein konstanter Wert im Algorithmus verwendet werden.

- `for=value;expr;next`: Alle Instanzen der Variable werden mit einer `for`-Schleife aufgezählt. Dabei ist `value` der initiale Wert der Java-Variablen, `expr` ist ein Boolescher Ausdruck, der genau dann wahr wird, wenn die Schleife abbrechen soll (alle Werte aufgezählt sind). Die jeweils nächste Instanz der Java-Variablen wird in einer Zuweisung innerhalb von `next` erzeugt.

Voreinstellung für `iterate` ist `user`. Das dritte optionale Argument *ref* spezifiziert einen eindeutigen Bezeichner, etwa `x`, für die Java Variable. Er kann benutzt werden, um mit `<x>` innerhalb von `expr` oder allen anderen optionalen Argumenten auf die Java-Variable zu verweisen.

`cproof2java` erzeugt zwei Java Methoden, die als Rahmen für die Implementierung der Quantifizierung dienen.

- `variableintroduction`*name*: Diese Methode wird im Java Kode Fragment immer dann aufgerufen, wenn der Algorithmus den quantifizierten Teil des Beweises erreicht. Die Methode kann zu Initialisierungen benutzt werden.

- `variabledestruction`*name*: Diese Methode wird im Java Kode Fragment immer dann aufgerufen, wenn der Algorithmus den quantifizierten Teil des Beweises verlaßt.

Dabei ist *name* ein eindeutiger Bezeichner, der aus *type* gebildet wird.

`EXISTS`

`<CPROOF CMD=EXISTS TYPE=`*type* `REF=`*ref*`>` *proof* `</CPROOF>`

Der optionale Parameter `type` gibt den Typen der zugehörigen Java-Variablen im Java Kode Fragment an. Voreinstellung ist `int`. Der zweite optionale Parameter *ref* spezifiziert einen eindeutigen Bezeichner, etwa `x`, für die Java Variable. Er kann benutzt werden, um mit `<x>` innerhalb von `expr` oder allen anderen optionalen Argumenten auf die Java-Variable zu verweisen.

`cproof2java` erzeugt zwei Java Methoden, die als Rahmen zur Implementierung der Quantifizierung dienen.

- `variableintroduction`*name*: Diese Methode wird im Java Kode Fragment immer dann aufgerufen, wenn der Algorithmus den quantifizierten Teil des Beweises erreicht. Die Methode kann zu Initialisierungen benutzt werden.
- `variabledestruction`*name*: Diese Methode wird im Java Kode Fragment immer dann aufgerufen, wenn der Algorithmus den quantifizierten Teil des Beweises verlaßt.

Dabei ist *name* ein eindeutiger Bezeichner, der aus *type* gebildet wird.

D.2.6 Induktion

INDUCTION

`<CPROOF CMD=INDUCTION TYPE=`*type* `ITERATE=`*it*`>` *proof* `</CPROOF>`

type gibt den Java Typen der Induktionsvariablen im Java Kode Fragment an. Mit dem zweiten Parameter *it* spezifiziert der Autor, wie der induktive Prozeß im Kode Fragment implementiert wird. *it* ist von der Form `for=value;expr;next`. Die Variable wird in einer `for`-Schleife iteriert. `value` gibt den initialen Wert der Java-Variablen an, `expr` ist ein Boolescher Ausdruck, der genau dann wahr wird, wenn die Schleife stoppen soll, und die jeweils nächste Instanz der Java-Variablen wird innerhalb einer Zuweisung in `next` erzeugt.

`cproof2java` erzeugt drei Java Methoden, die als Rahmen zur Implementierung der Induktion dienen.

- `induction`*name*: Liefert den initialen Wert der Induktionsvariablen, wenn der Algorithmus die Induktion erreicht.

- `variableintroduction`*name*: Diese Methode wird im Java Kode Fragment immer dann aufgerufen, wenn der Algorithmus den quantifizierten Teil des Beweises erreicht. Die Methode kann zu Initialisierungen benutzt werden.
- `variabledestruction`*name*: Diese Methode wird im Java Kode Fragment immer dann aufgerufen, wenn der Algorithmus den quantifizierten Teil des Beweises verlaßt.

Dabei ist *name* ein eindeutiger Bezeichner, der aus *type* gebildet wird.

BASECASE

`<CPROOF CMD=BASECASE EXPR=`*expr*`>` *cases* `</CPROOF>`

Der Basisfall der Induktion muß immer vor dem Induktionsschritt im Text vorkommen. *expr* ist ein Boolescher Java Ausdruck, der genau dann wahr ist, wenn der aktuelle Zustand der Induktionsvariablen vom Basisfall behandelt wird. Auf dem aktuellen Zustand kann mit `<n>` in *expr* zugegriffen werden (analog auch auf alle anderen sichtbaren Variablen). *cases* enthält alle Basisfäle. Jeder einzelne Fall muß in einem separatem Beweis enthalten sein, der mit einen Pseudo-Tag mit Kommando `CASE` markiert ist.

`<CPROOF CMD=CASE EXPR=`*expr*`>` *cases* `</CPROOF>`

expr ist ein Boolescher Java Ausdruck, der genau dann wahr ist, wenn der aktuelle Zustand der Induktionsvariablen von diesem Fall behandelt wird. Auf dem aktuellen Zustand kann mit `<n>` in *expr* zugegriffen werden (analog auch auf alle anderen sichtbaren Variablen).

STEPCASE

`<CPROOF CMD=STEPCASE>` *proof* `</CPROOF>`

Es muß genau ein Induktionsschritt vorkommen. *proof* ist der Beweis des Induktionsschritt. (ohne Pseudo-Tag mit `PROOF` oder `SUBPROOF` Kommando).

`BECAUSEIND`

`<CPROOF CMD=BECAUSEIND EXPR=`*expr*`ID=`*id*`>` *hint* `</CPROOF>`

Dieses Pseudo-Tag sollte anstatt des Kommandos `CBECAUSE` verwendet werden, um in einem Induktionsschritt zu markieren wo und wie die Induktionshypothese verwendet wurde. *expr* ist ein Boolescher Java Ausdruck, der beschreibt wie die Induktionsvariable reduziert wird (etwa `<n>-1`). Dieser Parameter wird momentan

ignoriert. Er kann in zukünftigen Versionen benutzt werden, um eine rekursive Variante der Induktion zu implementierten. Der Operator für diesen Beweisschritt und die Klammern, die den Kommentar einschließen müssen in HTML manuell angegeben werden.

Der (nicht optionale) Parameter *id* ist ein eindeutiger Bezeichner. Basierend auf diesem Bezeichner erzeugt `cproof2java` eine Java Methode, die als Rahmen für die Implementierung der Booleschen Struktur im Algorithmus dient:

- `because`*id*: Implementiert die Transformation der vorangestellten Booleschen Struktur in die nachfolgende.

D.2.7 Fallunterscheidung

`<CPROOF CMD=CCASE=`*expr*`>` *proof* `</CPROOF>`

Wenn ein Beweis aus unterschiedlichen Fällen besteht, dann sollte dieses Pseudo-Tag benutzt werden, um alle diese Fälle im Beweise zu markieren. Jeder einzelne Fall in *proof* besteht aus einem separaten Beweise, der mit dem Pseudo-Tag `CASE` markiert ist.

`<CPROOF CMD=CASE EXPR=`*expr*`>` *cases* `</CPROOF>`

expr ist ein Boolescher Java Ausdruck, der genau dann wahr ist, wenn der aktuelle Zustand des Beweises von diesem Fall behandelt wird. Innerhalb von `expr` kann auf die sichtbaren Java-Variablen mit dem optionalen Bezeichner zugegriffen werden (siehe Quantifizierungen).

Anhang E

Fragebögen

Fragebogen zur Übungsaufgabe „PUZZLE-Problem“

1. Haben Sie die Aufgabe 1 auf dem 4. Übungsblatt („PUZZLE-Problem“) bearbeitet? ja ☐ nein ☐

Falls *ja*, bitten wir Sie, Angaben zu den folgenden Fragen zu machen (alle Angaben sind freiwillig):

2. Wieviele Stunden haben Sie sich insgesamt mit der Übungsaufgabe beschäftigt? _____ h

3. Wie beurteilen Sie den Schwierigkeitsgrad der Aufgabe?
(1: viel zu niedrig, 2: etwas zu niedrig, 3: richtig, 4: etwas zu hoch, 5: viel zu hoch) _____

4. Haben Sie die richtige Transformationsvorschrift gefunden? ja ☐ teilweise ☐ nein ☐

5. Haben Sie dabei mehr mit Papier und Bleistift (P.u.B) oder mehr mit dem Puzzle-Applet gearbeitet? (1: nur P.u.B., 2: überwiegend P.u.B., 3: beides gleich,
4: überwiegend Applet, 5: nur Applet) _____

Falls Sie das Puzzle-Applet benutzt haben, bitten wir Sie, auch folgende Fragen zu beantworten:

6. Wieviel haben Ihnen die Vorgaben beim Puzzle-Applet zu einem Teil der Transformationsvorschrift geholfen? (1: sehr viel, 2: viel, 3: mäßig, 4: wenig, 5: gar nicht) _____

7. Bitte geben Sie an, wie Sie das Puzzle-Applet benutzt haben, und ob Ihnen das geholfen hat, einen Fehler in ihrer Transformationsvorschrift zu finden:

   Haben Sie eine Transformationsvorschrift eingegeben? ja ☐ nein ☐ geholfen ☐

   Haben Sie die Transformation an der Beispiel-TM getestet und die erzeugten Puzzleteile in den Rahmen eingesetzt? ja ☐ nein ☐ geholfen ☐

   Haben Sie eine eigene Turingmaschine eingegeben? ja ☐ nein ☐ geholfen ☐

8. Welche zusätzlichen Hilfestellungen durch das Puzzle-Applet hätten Sie sich gewünscht?

   _____

   _____

9. Welche Probleme hatten Sie mit der Bedienung des Puzzle-Applets?

   _____

   _____

   _____

10. Haben Sie die Bedienungsanleitung zum Puzzle-Applet gelesen?
(1: vollständig, 2: fast vollst., 3: zum Teil, 4: nur kurz, 5: gar nicht) _____

Fragebogen zur Übungsaufgabe „MONOTONE 3SAT“

1. Haben Sie Aufgabe 2 auf dem 4. Übungsblatt („MONOTONE 3SAT“) bearbeitet? ja ☐ nein ☐

Falls *ja*, bitten wir Sie, Angaben zu den folgenden Fragen zu machen (alle Angaben sind freiwillig):

2. Wieviele Stunden haben Sie sich insgesamt mit der Übungsaufgabe beschäftigt? _____ h

3. Wie beurteilen Sie den Schwierigkeitsgrad der Aufgabe?
   (1: viel zu niedrig, 2: etwas zu niedrig, 3: richtig, 4: etwas zu hoch, 5: viel zu hoch) _____

4. Haben Sie die richtige Transformationsvorschrift gefunden? ja ☐ teilweise ☐ nein ☐

5. Haben Sie die bei der Aufgabe angegebenen HTML-Seiten benutzt, um Ihre Transformationsvorschrift zu überprüfen? ja ☐ nein ☐

Falls *ja*, bitten wir Sie, auch die folgenden Fragen zu beantworten:

6. Wieviel haben Ihnen die Vorgaben bei den HTML-Seiten bei der Lösung der Aufgabe geholfen? (1: sehr viel, 2: viel, 3: mäßig, 4: wenig, 5: gar nicht) _____

7. Haben Ihnen die HTML-Seiten geholfen, Fehler in Ihrer Lösung zu entdecken? ja ☐ teilweise ☐ nein ☐

8. Welche zusätzlichen Hilfestellungen durch die HTML-Seiten hätten Sie sich gewünscht?

   _____

   _____

9. Welche Probleme hatten Sie mit der Bedienung der HTML-Seiten?

   _____

   _____

   _____

10. Haben Sie die Bedienungsanleitung zu den HTML-Seiten gelesen?
    (1: vollständig, 2: fast vollst., 3: zum Teil, 4: nur kurz, 5: gar nicht) _____

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[ALLWEIN und BARWISE 1996] ALLWEIN, G. und J. BARWISE, Hrsg. (1996). *Logical Reasoning with Diagrams*. Oxford University Press.

[ANDERSON et al. 1985] ANDERSON, JOHN R., C. F. BOYLE und G. YOST (1985). *The Geometry Tutor*. In: JOSHI, ARAVIND, Hrsg.: *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, S. 1–7, Los Angeles, CA. Morgan Kaufmann.

[BACK et al. 1996] BACK, RALPH, J. GRUNDY und J. VON WRIGHT (1996). *Structured Calculational Proof*. Joint Computer Science Technical Report TR-CS-96-09, The Australian National University, Department of Computer Science, Canberra ACT 0200, Australia.

[BACK et al. 1997] BACK, RALPH, J. GRUNDY und J. VON WRIGHT (1997). *Structured Calculational Proof*. Formal Aspects of Computing, 9(5–6):469–483.

[BACKHOUSE 1989] BACKHOUSE (1989). *Making Formality Work For Us*. BEATCS: Bulletin of the European Association for Theoretical Computer Science, 38.

[BAECKER und SHERMAN 1981] BAECKER, R. und D. SHERMAN (1981). *Sorting out Sorting*. Dynamic Graphics Project, University of Toronto. 30 minütiger farbiger Tonfilm.

[BAKER et al. 1995] BAKER, J. E., I. F. CRUZ, G. LIOTTA und R. TAMASSIA (1995). *A New Model for Algorithm Animation Over the WWW*. ACM Computing Surveys, 27(4):568–572.

[BARKER-PLUMMER und BAILIN 1992] BARKER-PLUMMER, D. und S. C. BAILIN (1992). *Graphical Theorem Proving: An Approach to Reasoning with the Help of Diagrams*. In: NEUMANN, BERND, Hrsg.: *Proceedings of the 10th European Conference on Artificial Intelligence*, S. 55–59, Vienna, Austria. John Wiley & Sons.

[BARWISE und ETCHEMENDY 1993] BARWISE, JON und J. ETCHEMENDY (1993). *The Language of First-Order Logic: Including the Macintosh Program Tarski's World 4.0*. Center for the Study of Language and Information (CSLI), Stanford, California, Dritte überarbeitete und erweiterte Aufl.

[BARWISE und ETCHEMENDY 1998] BARWISE, JON und J. ETCHEMENDY (1998). *Hyperproof*. Center for the Study of Language and Information (CSLI), Stanford, California.

[BECKERT et al. 1996] BECKERT, BERNHARD, R. HÄHNLE, K. GEISS, P. OEL, C. PAPE und M. SULZMANN (1996). *The Many-Valued Tableau-Based Theorem Prover $_3T^AP$, Version 4.0*. Interner Bericht 3/96, Universität Karlsruhe, Fakultät für Informatik.

[BENZMÜLLER et al. 1997] BENZMÜLLER, C., L. CHEIKHROUHOU, D. FEHRER, A. FIEDLER, X. HUANG, M. KERBER, M. KOHLHASE, K. KONRAD, A. MEIER, F. MELIS, W. SCHAARSCHMIDT, J. SIEKMANN und V. SORGE (1997). *ΩMEGA: Towards a Mathematical Assistant*. In: MCCUNE, WILLIAM, Hrsg.: *Proceedings of the 14th International Conference on Automated deduction*, Bd. 1249 d. Reihe *LNAI*, S. 252–255, Berlin. Springer.

[BIBEL 1987] BIBEL, W. (1987). *Automated Theorem Proving*. Vieweg Verlag.

[BRADY 1988] BRADY, ALLEN H. (1988). *The Busy Beaver Game and the Meaning of Life*. In: *The Universal Turing Machine. A Half-Century Survey*, S. 259–277. Kammerer & Unverzagt.

[BROWN 1988a] BROWN, M. H. (1988a). *Exploring Algorithms Using Balsa-II*. IEEE Computer, 21(5):14–38.

[BROWN und HERSHBERGER 1992] BROWN, M. H. und J. HERSHBERGER (1992). *Color and Sound in Algorithm Animation*. Computer, 25(12):52–63.

[BROWN und SEDGEWICK 1984] BROWN, M. H. und R. SEDGEWICK (1984). *A system for algorithm animation*. Computer Graphics, 18(3):177–186.

[BROWN 1988b] BROWN, MARC H. (1988b). *Algorithm Animation*. MIT Press, Cambridge.

[BROWN 1993] BROWN, MARC H. (1993). *The 1992 SRC Algorithm Animation Festival*. Technischer Bericht 98, Compaq Systems Research Center.

[BROWN und NAJORK 1997] BROWN, MARC H. und M. A. NAJORK (1997). *Collaborative Active Textbooks*. J. Visual Languages and Computing, 8(4):453–486.

[BROWN und RAISAMO 1996] BROWN, MARC H. und R. RAISAMO (1996). *JCAT: Collaborative Active Textbooks Using Java*. In: *CompuGraphics'96*.

[CARDELLI 1995] CARDELLI, LUCA (1995). *A Language with Distributed Scope*. Computing Systems, 8(1):27–59.

[DAHN et al. 1997] DAHN, B. I., J. GEHNE, T. HONIGMANN und A. WOLF (1997). *Integration of Automated and Interactive Theorem Proving in ILF*. In: MCCUNE, WILLIAM, Hrsg.: *Proceedings of the 14th International Conference on Automated deduction*, Bd. 1249 d. Reihe *LNAI*, S. 57–60, Berlin. Springer.

[DIJKSTRA und SCHOLTEN 1990] DIJKSTRA, EDSGER W. und C. S. SCHOLTEN (1990). *Predicate Calculus and Program Semantics*. Springer, New York.

[DRAKOS 94] DRAKOS, NIKOS (94). *From Text to Hypertext: A Post-Hoc Rationalisation of LaTeX2HTML*. In: *The Proceedings of the First WorldWide Web Conference*, CERN, Geneva, Switzerland.

[DUISBERG 1987-1988] DUISBERG, ROBERT ADAMY (1987-1988). *Animation Using Temporal Constraints: An Overview of the Animus System*. Human-Computer Interaction, 3(3):275–307.

[ELLSON et al. 1998] ELLSON, JOHN, E. GANSNER, E. KOUTSOFIOS und S. NORTH (1998). *GraphViz*. Elektronisch erhältlich im World Wide Web unter `http://www.research.att.com/sw/tools/graphviz`.

[FITTING 1996] FITTING, MELVIN C. (1996). *First-Order Logic and Automated Theorem Proving*. Springer, New York, zweite Aufl.

[FRICK 1998] FRICK, ARNE (1998). *Visualisierung von Programmabläufen*, Bd. 10 d. Reihe *Forschritts-Berichte VDI*. VDI, Düsseldorf.

[GAREY und JOHNSON 1979] GAREY, MICHAEL R. und D. S. JOHNSON (1979). *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York.

[GLOOR und STREITZ 1990] GLOOR, P. A. und N. A. STREITZ, Hrsg. (1990). *Hypertext und Hypermedia*. Springer.

[GLOOR et al. 1993] GLOOR, PETER, S. DYNES und I. LEE (1993). *Animated algorithms*. MIT Press, Cambridge, MA. (CD-ROM).

[GLOOR et al. 1992] GLOOR, PETER A., D. B. JOHNSON, F. MAKEDON und P. METAXAS (1992). *A Visualization System for Correctness Proofs of Graph Algorithms*. Technischer BerichtTR92-180, Dartmouth College, Computer Science.

[GRAMOND und RODGER 1999] GRAMOND, E. und S. H. RODGER (1999). *Using JFLAP to Interact with Theorems in Automata Theory*. Thirtieth SIGCSE Technical Symposium on Computer Science Education.

[GRIES und SCHNEIDER 1994] GRIES, DAVID und F. B. SCHNEIDER (1994). *A Logical Approach to Discrete Math*. Texts and Monographs in Computer Science. Springer, New York, NY.

[GRUNDY 1996a] GRUNDY, JIM (1996a). *A Browsable Format for Proof Presentation*. Mathesis Universalis, 1(2).

[GRUNDY 1996b] GRUNDY, JIM (1996b). *A Browsable Format for Proof Presentation*. Technical Report TUCS-TR-22, Turku Centre for Computer Science, Finland.

[GRUNDY und LÅNGBACKA 1997] GRUNDY, JIM und T. LÅNGBACKA (1997). *Recording HOL Proofs in a Structured Browsable Format*. In: JOHNSON, MICHAEL, Hrsg.: *Alebraic Methodology and Software Technology: 6th International Conference, AMAST'97*, Bd. 1349 d. Reihe *Lecture Notes in Computer Science*, S. 567–571, Sydney, Australia. Springer.

[HAAJANEN et al. 1997] HAAJANEN, J., M. PESONIUS, E. SUTINEN, J. TARHIO, T. TERÄSVIRTA und P. VANNINEN (1997). *Animation of user algorithms on the Web*. In: *Proc. VL '97, IEEE Symposium on Visual Languages*, S. 360–367. IEEE.

[HARRISON 1996] HARRISON, JOHN (1996). *Proof Style*. In: GIMÉNEZ, EDUARDO und C. PAULIN-MOHRING, Hrsg.: *Selected Papers 4th Intl. Workshop on Types for Proofs and Programs, TYPES'96, Aussois, France, 15–19 Decemeber 1996*, Bd. 1512 d. Reihe *LNCS*, S. 154–172, Berlin. Springer.

[HENRY et al. 1990] HENRY, ROBERT R., K. M. WHALEY und B. FORSTALL (1990). *University of Washington illustrating compiler*. ACM SIGPLAN Notices, 25(6):223–233.

[KHURI und WILLIAMS 1994] KHURI, SAMI und J. WILLIAMS (1994). *Understanding the Bottom-Up SLR Parser*. In: JOYCE, DANIEL, Hrsg.: *Proceedings of the 25th Technical Symposium on Computer Science Education*, Bd. 26 d. Reihe *SIGCSE Bulletin*, S. 339–343, New York, NY, USA. ACM Press.

[KHURI und SUGONO 1998] KHURI, SAMIT und Y. SUGONO (1998). *Animating Parsing Algorithms*. SIGSCE, ??(3):232–236.

[KYRSKYKARI und RAIHA 1987] KYRSKYKARI, A. und K.-J. RAIHA (1987). *Aladdin: A Tool for Generating Algorithm Animations*. Technischer Bericht A-1987-6, Department of Computer Science, University of Tampere, Tampere, Finland.

[LAHTINEN et al. 1996] LAHTINEN, S.-P., T. LAMMINJOKI, E. SUTINEN, J. TARHIO und A. P. TUOVINEN (1996). *Towards Automated Animation of Algorithms*. In: *Winter School of Computer Graphics 1996*. University of West Bohemia, Plzen, Czech Republic, 12-16 February 1996.

[LAMPORT 1986] LAMPORT, LESLIE (1986). *LaTeX: A Document Preparation System - User's Guide & Reference Manual*. Addison-Wesley, Reading.

[LAMPORT 1995] LAMPORT, LESLIE (1995). *How to write a proof*. American Mathematical Monthly, 102(7):600–608.

[MARRIOT und MEYER 1998] MARRIOT, KIM und B. MEYER (1998). *Introduction*. In: MARRIOT, KIM und B. MEYER, Hrsg.: *Visual Language Theory*, Kap. 1, S. 1–4. Springer.

[MATSUDA und OKAMOTO 1998] MATSUDA, N. und T. OKAMOTO (1998). *Diagrammatic Reasoning for Geometry ITS to Teach Auxiliary Line Construction Problems*. Lecture Notes in Computer Science, 1452:244–??

[MITCHENER et al. 1996] MITCHENER, GARRET, T. SELBY und J. UNGER (1996). *Using Lambada v. 1.0*. Elektronisch erhältlich im World Wide Web unter `http://www.cs.duke.edu/~garrett/Lambada/Release-1.0.html`.

[MOLL et al. 1988] MOLL, R. A., A. J. KFOURY und M. A. ARBIB (1988). *An Introduction to Formal Language Theory*. Texts and Monographs in Computer Science. Springer-Verlag.

[MURRAY und ROSENTHAL 1987] MURRAY, NEIL V. und E. ROSENTHAL (1987). *Inference with path resolution and semantic graphs*. Journal of the ACM, 34(2):225–254.

[NELSON und GLASSMAN] NELSON, GREG und S. GLASSMAN. *An Animation of the proof of Euclid's Proposition 47*. Elektronisch erhältlich unter `http://www.research.digital.com/SRC/zeus/euclid.html`.

[PAPE und SCHMITT 1997] PAPE, CHRISTIAN und P. H. SCHMITT (1997). *Visualizations for Proof Presentation in Theoretical Computer Science Education*. In: HALIM, Z., T. OTTMANN und Z. RAZAK, Hrsg.: *Proceedings of International Conference on Computers in Education, Kuching, Sarawak, Malaysia, December 2–6*, S. 229–236. Association for the Advancement of Computing in Education.

[PIERSON und RODGER 1998] PIERSON, W. und S. H. RODGER (1998). *Web-based Animation of Data Structures Using JAWAA*. In: *Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*.

[POST 1946] POST, E. (1946). *A variant of a recursivley unsolvable problem*. Bulletin of the American Mathematical Society, 53:264–268.

[PRICE et al. 1993] PRICE, BLAINE A., R. BAECKER und I. SMALL (1993). *A Principled Taxonomy of Software Visualization*. Journal of Visual Languages and Computing, 4(3):211–266.

[PRICE et al. 1998] PRICE, BLAINE A., R. BAECKER und I. SMALL (1998). *An Introduction to Software Visualization*. In: STASKO, JOHN, J. DOMINGUE, M. H. BROWN und B. A. PRICE, Hrsg.: *Software Visualization: Programming as a Multimedia Experience*, Kap. 1, S. 3–27. M.I.T. Press.

[REIF et al. 1997] REIF, W., G. SCHELLHORN und K. STENZEL (1997). *Proving System Correctness with KIV 3.0*. In: MCCUNE, WILLIAM, Hrsg.: *Proceedings of the 14th International Conference on Automated deduction*, Bd. 1249 d. Reihe *LNAI*, S. 69–72, Berlin. Springer.

[RESCH 1998] RESCH, ULLA (1998). *Entwurf und Implementierung einer Lernsoftware zu ausgewählten Algorithmen aus der Automatentheorie*. Diplomarbeit, Universität Karlsruhe.

[RODGER 1994] RODGER, S.H. (1994). *LLparse and LRparse: Visual and Interactive Tools for Parsing*. SIGCSE'94, 3:208–211.

[RODGER 1995] RODGER, S.H. (1995). *An Interactive Lecture Approach to Teaching Computer Science*. SIGCSE'95, 3:278–282.

[RODGER] RODGER, SUSAN H. *JFlap 3.0*. Elektronisch erhältlich unter `http://www.cs.duke.edu/~rodger/tools/jflap/index.html`.

[ROMAN et al. 1992] ROMAN, G.-C., K. COX, C. WILCOX und J.Y.PLUN (1992). *Pavane: a System for Declarative Visualization of Concurrent Computations*. Journal of Visual Languages and Computing, 3(2):161–193.

[SETZER] SETZER, ANTON. *List of Research Groups in Logic and Theoretical Computer Science World-wide*. Elektronisch erhältlich im World Wide Web unter `http://www.math.uu.se/logik/logic-server/software.html`.

[SPERSCHNEIDER und HAMMER 1996] SPERSCHNEIDER, VOLKER und B. HAMMER (1996). *Theoretische Informatik: eine problemorientierte Einführung*. Springer Verlag.

[STASKO 1996a] STASKO, JOHN (1996a). *SAMBA Animation Designer's Package*. `ftp://ftp.cc.gatech.edu/pub/gvu/tech-reports/96-19.ps.Z`.

[STASKO 1990] STASKO, JOHN T. (1990). *Tango: A Framework and System for Algorithm Animation*. Computer, 23(9):27–39.

[STASKO 1992] STASKO, JOHN T. (1992). *Animating Algorithms with XTANGO*. SIGACT News, 23(2):67–71.

[STASKO 1996b] STASKO, JOHN T. (1996b). *Using Student-Built Algorithm Animations as Learning Aids*. Technischer Bericht GIT-GVU-96-19, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA.

[SYME 1995] SYME, D. (1995). *A New Interface for HOL — Ideas, Issues and Implementation*. Lecture Notes in Computer Science, 971:324–340.

[TROELSTRA 1999] TROELSTRA, ANNE SJERP (1999). *From Constructivism to Computer Science*. Theoretical Computer Science, 211(1–2):233–252.

[TURING 1936] TURING, ALAN M. (1936). *On computable numbers, with an application to the Entscheidungsproblem*. Proc. London Mathematical Society, 42:230–265.

[WAGENKNECHT 1998] WAGENKNECHT, CHRISTIAN (1998). *Theoretische Informatik mit SCHEME — Ein Erfahrungsbericht*. In: CLAUS, VOLKER, Hrsg.: *Informatik und Ausbildung*, Informatik aktuell, S. 92–101. Springer.

[WAGNER 1994] WAGNER, K. W. (1994). *Einführung in die Theoretische Informatik. Grundlagen und Modelle*. Springer.

[WEGENER 1993] WEGENER, INGO (1993). *Theoretische Informatik*. B.G.Teubner.

Index