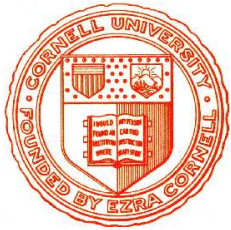


Automatisierte Logik und Programmierung

Wintersemester 2008/09



Christoph Kreitz

Theoretische Informatik, Raum 1.18, Telephon 3060

kreitz@cs.uni-potsdam.de

<http://www.cs.uni-potsdam.de/ti/lehre/08-ALuP-I>

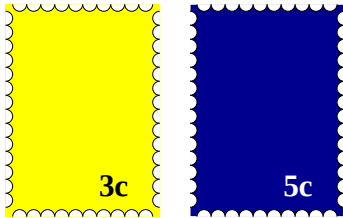


1. Automatisches Schließen – wozu?
2. Was ist Automatisierte Logik?
3. Anwendungen und Erfolge
4. Aufbau & Organisation der Veranstaltung

AUTOMATISCHES SCHLIESSEN – WOZU?

- **Es gibt zu viele Fehler in wissenschaftlicher Arbeit**
 - Mathematische Beweise sind oft komplex
 - Menschen machen Fehler bei der Ausarbeitung von Details
 - Fehler werden auch beim Reviewprozeß von Publikationen übersehen
 - 40–50% aller veröffentlichten Resultate sind falsch
- **Fast alle Softwareprodukte sind unzuverlässig**
 - Softwareprodukte und ihre Anforderungen sind extrem komplex
 - Auch sorgfältig getestete Programme enthalten größere Fehler
- **Informales Vorgehen hat sich nicht bewährt**
 - Wir konzentrieren uns auf Ideen und machen Fehler bei der Ausführung
 - Wir machen implizite Annahmen, die nicht immer stimmen
 - Wir verwenden Analogien, die nicht wirklich greifen
 - Wir vertrauen “Autoritäten” ohne nachzuprüfen

BEISPIEL: DAS BRIEFMARKEN PROBLEM



Ist es möglich jedes Porto ab 8 Cent nur mit 3c und 5c Briefmarken zu erzeugen?

$$8c = \begin{array}{|c|c|} \hline 5c & 3c \\ \hline \end{array}, \quad 9c = \begin{array}{|c|c|c|} \hline 3c & 3c & 3c \\ \hline \end{array}, \quad 10c = \begin{array}{|c|c|} \hline 5c & 5c \\ \hline \end{array}, \quad 11c = \begin{array}{|c|c|c|} \hline 5c & 3c & 3c \\ \hline \end{array}, \quad \dots$$

● Einfacher Induktionsbeweis

- Zeige: für alle $n \geq 8$ gibt es $i, j \in \mathbb{N}$ mit $n = i \cdot 3 + j \cdot 5$
 - Basisfälle 8, 9, 10 wie oben illustriert.
 - Induktionsschritt erzeugt Lösung für $n+1$ aus der für $n-2$.

● Gibt es andere Paare mit derselben Eigenschaft?

- Offensichtlich 1c und jede andere Zahl, 2c und jede ungerade Zahl
- **Kann man beweisen, daß dies alle Möglichkeiten sind?**

Sei $a < b \in \mathbb{N}$. Ist jedes $n \geq a+b$ darstellbar als $n = i \cdot a + j \cdot b$ mit $i, j \in \mathbb{N}$, dann ist $a=1$ oder $a=2$ und b ist ungerade oder $a=3$ und $b=5$.

Ohne Beschränkung der Allgemeinheit ist $1 < a$.

$$\exists i, j. a+b+1 = i \cdot a + j \cdot b \quad \mapsto \quad a \mid (b+1) \text{ oder } b=a+1 \quad (1)$$

$$\exists i, j. a+b+2 = i \cdot a + j \cdot b \quad \mapsto \quad a=2 \text{ oder } a \mid (b+2) \text{ oder } b=a+2 \quad (2)$$

Falls $a=2$, dann ist b ungerade wegen (1)

Falls $a > 2$, dann ist $b > 3$ und (1) liefert zwei Fälle

$a \mid (b+1)$: wegen $a > 2$ kann a nicht $b+2$ teilen und wegen (2) gilt $b=a+2$.

$$\exists i, j. a+b+3 = i \cdot a + j \cdot b \quad \mapsto \quad a=3 \text{ oder } a \mid (b+3) \text{ oder } b=a+3 \quad (3)$$

– $b=a+3$ ist unmöglich, da $b=a+2$.

– $a \mid (b+3)$ ist unmöglich, da $a \mid (b+1)$ und $a > 2$.

Also gilt $a = 3$ und $b = 5$



$b=a+1$: wegen (2) folgt wie oben $a \mid (a+3)$ oder $a+1 = a+2$.

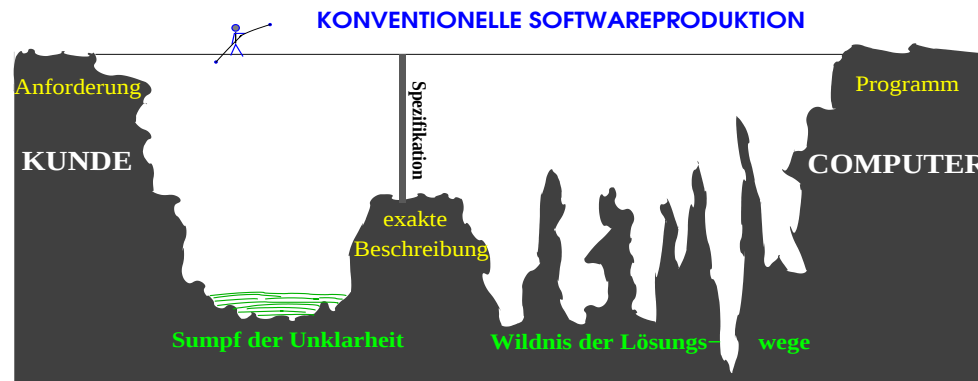
~~Beides ist unmöglich.~~ **Möglich für $a=3$ und $b=4$**

Formales Vorgehen hilft, solche Fehler zu vermeiden

WIR KÖNNEN UNS SOFTWAREFEHLER NICHT ERLAUBEN

- **Software ist integraler Bestandteil unseres Lebens**
 - Steuerungsmodule in Alltagsprodukten, e-Commerce, Telephonnetze, Automobilkonstruktion, Luftfahrtkontrolle, ...
- **Softwarefehler sind lästig**
 - Reboot, Datenverlust, Viren, ...
- **Softwarefehler sind teuer**
 - 1994: **Pentium I Prozessor** liefert falsche Resultate bei Division
 - 1996: **ESA Ariane 501** explodiert wegen Überlauf der 16bit Arithmetik
 - 1999: **Mars Polar Lander & Climate Orbiter** stürzen ab (Einheitenfehler)
 - 2000: Barclays Bank erlebt **unbefugten Online-Zugriff** auf Fremdkonten
- **Softwarefehler kosten Menschenleben**
 - 1988: **Air France A320** streift Bäume bei Flugshow (Einheitenfehler)
 - 1993: **Lufthansa A320** verweigert Umkehrschub bei Landung im Regen
 - 1995: **Boeing 757** prallt auf Berge auf (Fehler in Navigationsdaten)

ES LÄUFT ETWAS FALSCH IN DER SOFTWAREPRODUKTION



● Konventionelle Softwareproduktion ist ineffizient

- Entwurf und Implementierung fokussiert auf Modellierungs- und Programmiersprachen anstatt auf Eigenschaften des Problembereichs
- Codierung ‘von Hand’ führt zu hohen Kosten für Erstellung und Wartung
- Resultate sind oft suboptimal und funktionieren selten auf Anhieb
- Programmierer geben keine Begründung für Korrektheit ihres Programms
- Probleme führen zu ad hoc Änderungen statt Revision des Gesamtentwurfs

● Softwareentwicklung ist mehr als Codierung

- Logische Verarbeitung von Wissen + kreative Umsetzung von Ideen

Automatisierte formale Logik kann diesen Prozeß unterstützen

BEISPIEL: MAXIMALE SEGMENTSUMME EINER LISTE

Gegeben eine Folge $a_1, a_2, \dots, a_n \in \mathbb{Z}$ bestimme die Summe $\sum_{i=p}^q a_i$ eines Segmentes, die maximal bezüglich aller möglicher Segmentsummen ist

2	3	-6	4	5	-3	8	-2	-1	5	-9	2	3	16
---	---	----	---	---	----	---	----	----	---	----	---	---	----

- Direkte Lösung leicht zu programmieren

```
let maxseg a ≡  
  result := a[1]  
  from p = 1 to length(a) do  
    from q = i to length(a) do  
      sum := 0  
      from i = p to q do sum := sum+a[i] end  
      if sum > result then result := sum  
    end  
  end  
end
```

- Algorithmus ist ineffizient ($\mathcal{O}(n^3)$)

- Wie kann man eine bessere Lösung erzeugen?

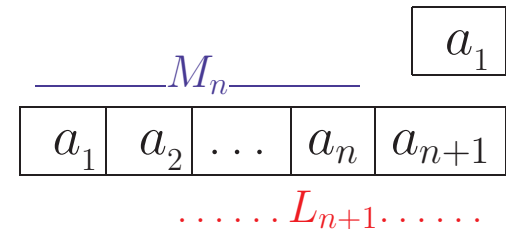
MAXIMALE SEGMENTSUMME: SYSTEMATISCHE LÖSUNG

Betrachte Eigenschaften von $M_n \equiv \max\{\sum_{i=p}^q a_i \mid 1 \leq p \leq q \leq n\}$

- Induktive Analyse liefert

- $M_1 = a_1$

- $M_{n+1} = \max(M_n, \max\{\sum_{i=p}^{n+1} a_i \mid 1 \leq p \leq n\})$



- Definiere $L_n \equiv \max\{\sum_{i=p}^n a_i \mid 1 \leq p \leq n\}$

- $L_1 = a_1, \quad L_{n+1} = \max(L_n + a_{n+1}, a_{n+1})$

- Analyse liefert eleganten, effizienten Algorithmus

```
let maxseg a ≡
```

```
  Mn := a[1]
```

```
  Ln := a[1]
```

```
  from n = 1 to length(a) do
```

```
    if Ln > 0 then Ln := Ln + a[n] else Ln := a[n]
```

```
    if Ln > Mn then Mn := Ln
```

```
  end
```

AUTOMATISCHES SCHLIESSEN LIEFERT LÖSUNGEN

- **Logische Analyse führt zu besserer Software**
 - Wissen wird systematisch verarbeitet
- **Formalisierung logischer Schlüsse eliminiert Fehler**
 - Zusammenhang zwischen Aufgabe und Lösung wird überprüfbar
 - Effektive Kooperation zwischen Mensch und Maschine möglich
 - Kreative Steuerung des Entwicklungsprozesses durch Menschen
 - Computer kontrolliert Korrektheit der Herleitung in logischem Kalkül
- **Automatisierte Logik reduziert Aufwand**
 - Computer führt “triviale” logische Schlüsse von selbst aus
 - Computer synthetisiert Code aus formaler Analyse des Problems

AUTOMATISIERTE LOGIK – LEIBNIZ’ TRAUM ERFÜLLT?

~1700: **“Logisches Schließen präzisieren”**

- Universelle, akkurate Wissenschaftssprache
 - Regeln für Lösung wissenschaftlicher Dispute
- } $\mapsto$ “*Calculus*”

1890: **Formale Logik**

- Gesetze der Logik simuliert durch mechanische Textmanipulation

1941: **Computer – ideal für symbolische Manipulation**

- Fehlerfreie Anwendung von Regeln
- Suche nach Lösungen durch Untersuchung ‘unzähliger’ Möglichkeiten



Simuliere logisches Schließen auf dem Computer

Simulation mathematisch-semantischer Argumente

- **Anwendung formaler Regeln ohne Nachdenken**

- Umgeht Mehrdeutigkeiten der natürlichen Sprache
- Erlaubt schematische Lösung mathematischer Probleme

- **Kernbestandteile:**

- Formale Sprache (Syntax + Semantik)
- Ableitungssystem (Axiome + Inferenzregeln)

- **Wichtige Eigenschaften**

- Korrekt, vollständig, automatisierbar (notwendig)
- Leicht verständlich (für Interaktion)
- Konstruktiv, ausdrucksstark (für Programmierung)

Nicht jeder Kalkül hat all diese Eigenschaften gleichzeitig

THEORETISCHE GRENZEN

- **Arithmetik ist nicht voll axiomatisierbar**
- **Kein allgemeines Verfahren kann entscheiden**
 - ob eine gegebene logische Formel gültig ist
 - ob ein gegebenes Programm terminiert
 - ob ein gegebenes Programm korrekt ist
 - ob zwei Programme dieselbe Funktionalität haben



Beweisverfahren müssen nach Beweisen suchen

- Unendliche Suchbäume — keine Antwort im Mißerfolgsfall
- Suche erfordert Benutzersteuerung oder intelligente Strategien

● Interaktive Beweiseditoren

- Benutzer konstruieren Beweise durch Anwendung von Regeln
- Computer führt Regeln aus und zeigt ungelöste Teilprobleme
- Mechanismus: Pattern Matching + Term Rewriting

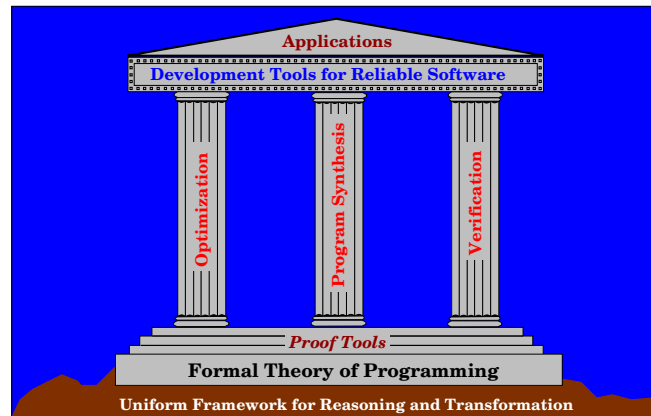
● Automatische Beweisprozeduren

- Taktiken: programmierte Anwendung von Inferenzregeln
- Entscheidungsprozeduren for entscheidbare Teilprobleme
- Beweissuchverfahren für eingeschränkte Anwendungsbereiche
 - Prädikatenlogik, Gleichheit, Induktion, Spezialanwendungen, ...
- Beweisplaner, Model Checking, Computer Algebra, ...

● Integrierte Systeme

- Interaktive Beweiseditoren mit externen Steuerungsmechanismen

ANWENDUNGSGEBIETE



- **Mathematische Beweisführung**

- Aufdeckung und Korrektur von Fehlern
- Automatische Suche nach neuen Beweisen

(Beweisprüfung)

(Theorembeweisen)

- **Unterstützung für Entwurf zuverlässiger Software**

- Fehlersuche und Korrektheitsbeweise
- Verbesserung der Performanz
- Erzeugung aus Spezifikationen

(Verifikation)

(Optimierung)

(Synthese)

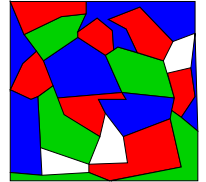
- **Inferenzmaschine für KI-Systeme**

- Problemlöser und Planer für Roboter, ...

ERFOLGE DER VERGANGENHEIT MACHEN MUT ...

1977: **Vier-Farben Problem**

- Spezialsoftware überprüft tausende kritischer Fälle



1993: **Synthese von Scheduling Algorithmen**

- **KIDS** erzeugt korrekte Algorithmen in wenigen Stunden
- Erzeugter Lisp Code 2000 mal schneller als existierende ADA Software

1995: **Pentium Bug**

- Model Checking findet Fehler in Hardwaretabellen für Division



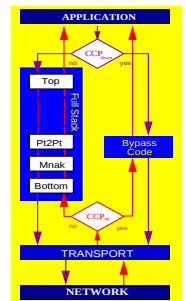
1996: **Robbins Hypothese**

The New York Times

- Theorembeweiser **EQP** findet (lesbaren) Beweis in 7 Tagen

1998: **Netzwerkverifikation, -optimierung, -entwurf**

- Verifikation findet subtilen Fehler in Kommunikationsprotokoll
- Logische Optimierung steigert Performanz um Faktor 3–10
- Formaler Entwurf eines verifizierten adaptiven Protokolls



- **Formale Logiken**

- Klassische, intuitionistische, modale, lineare, temporale, ...
- Logik erster Stufe, höherer Stufe, Typentheorien, ...
- Formalisierung von Objekten, Vererbung, Nebenläufigkeit, Echtzeit, ...

- **Automatische Beweisprozeduren**

- Matrixmethoden, Induktionsbeweisen, Rewriting, Beweisplaner, ...
- Entscheidungsprozeduren, kooperierende & verteilte Beweiser, ...

- **Beweisentwicklungssysteme**

- Interaktive Beweiseditoren, Beweispräsentation, Wissensverwaltung, ...
- Web-Einbindung, GUI's, Systemschnittstellen, ...

- **Anwendung: Logik in die Software bringen**

- Verifikation formalen Wissens zu Daten- und Algorithmenstrukturen
- Strategien für Synthese, Verifikation und Optimierung von Algorithmen
- Entwicklung und Verifikation sicherheitskritischer Systeme

⋮

THEMEN DIESER VERANSTALTUNG

● Formale Kalküle

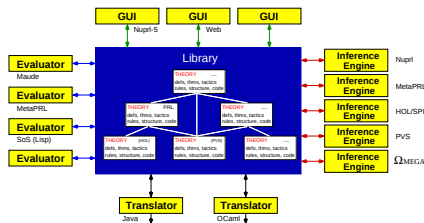
(Wintersemester)

- Logisches Schließen: Prädikatenlogik
- Programmierung: λ -Kalkül
- Programmeigenschaften: einfache Typentheorie
- Ein uniformer Kalkül: Konstruktive Typentheorie

● Beweisentwicklungssysteme

(Sommersemester)

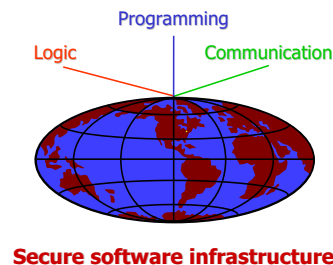
- Das NUPRL Logical Programming Environment
- Taktische Beweisführung
- Entscheidungsprozeduren
- Integration externer Beweissysteme



● Anwendungen

(Sommersemester)

- Aufbau formaler Theorien
- Programmsynthese
- Optimierung von (verteilter) Software



⋮

- **Zuordnung:** **theoretische Informatik**
angewandte Informatik
- **Veranstaltungstermine**
 - Di 11:00–12:30 – Vorlesung
 - Mi 13:30–15:00 – Vorlesung/Übung im Wechsel
 - **Keine Veranstaltung am 6./7. Januar 2009**
- **Hilfreiche Vorkenntnisse:**
 - Formale Logik, Funktionale Programmierung, KI
- **Lehrmaterialien**
 - Vorlesungsfolien, Skript (1995), Fachbücher im Web
- **Erfolgskriterien**
 - **Abschlußprüfung** (mündlich/schriftlich, je nach Teilnehmerzahl)
 - **Aktive** Teilnahme an Übungen ist sehr hilfreich