Es gibt selbstgefällige Programmierer, die die Vorstellung kultivieren, Informatik könne auf Mathematik oder Logik verzichten. Ebensogut kann ein Esel auf Beine verzichten. Er hat doch Zähne, sich vorwärtszuziehen.
(F.X. Reid)

Kapitel 1

Einführung

Seit mehr als 25 Jahren spricht man von einer Krise bei der Produktion kommerzieller Software. Auslöser dieser Krise ist genau diejenige Eigenschaft, welche den Einsatz von Software so attraktiv macht, nämlich die Vielseitigkeit und das komplexe Verhalten, das man erzeugen kann. Bis heute hat sich die Krise eher verschlimmert als verbessert, denn die Entwicklung von Methoden zur systematischen Softwareproduktion kann mit der stetig wachsenden Komplexität von Software, die auf dem Markt verlangt wird, nicht Schritt halten. Zwei Probleme treten dabei besonders hervor.

Das erste Problem sind die *Kosten von Software*, die in einem krassen Mißverhältnis zu dem Verfall der Hardwarepreise stehen. Im Gegensatz zu diesen entstehen Softwarekosten fast ausschließlich beim Entwurf — einer Phase, die Kreativität, Erfahrung und Disziplin verlangt. Zwar ist in allen Branchen die Entwurfsphase mit hohen Kosten verbunden. Bei der Softwareproduktion jedoch ist die Situation besonders schlecht, da die meisten Programme immer noch aus elementaren Konstrukten von Programmiersprachen anstatt aus größeren Modulen aufgebaut werden. Dies ist zu einem gewissen Teil eine Angewohnheit, die durch den wachsenden Einfluß objektorienterter Programmiersprachen abgebaut werden könnte. Prinzipiell aber liegt das Problem darin, daß selbst kleine Module bereits solch komplizierte Handlungsabläufe beschreiben müssen, daß hierzu bestehende Programmstücke nicht ohne Modifikationen übernommen werden können. Darüber hinaus werden die Kosten aber auch dadurch erhöht, daß der Erstentwurf eines Softwareprodukts nur selten die Absichten des Entwicklers erfüllt. Dies liegt daran, daß es – ganz im Gegensatz zur Hardwareentwicklung – bisher keinerlei Werkzeuge gibt, mit denen man ein bestimmtes Verhalten von Programmen erzwingen kann. Die notwendigen Tests und Korrekturen bis zur Erstellung des endgültigen Produkts dauern oft lange und machen dieses entsprechend teuer.

Aber auch nach der Auslieferung des Produkts entstehen weitere Kosten. Zum einen machen sich weitere Fehler in der Implementierung bemerkbar, da Tests niemals vollständig sein können. Zum anderen aber stellt sich nur allzu oft heraus, daß die gelieferte Software den wirklichen Wünschen der Kunden nicht vollständig entspricht, da diese Wünsche ursprünglich nicht genau genug formuliert waren. Aus diesem Grunde müssen nachträglich noch vielfältige Ergänzungen und Verbesserungen vorgenommen werden. Diese Modifikationen verlangen oft Änderungen, die mehr mit der abstrakten Entwurfsebene zu tun haben als mit dem unmittelbar vorliegenden Programmcode. Der ursprüngliche Entwicklungs*prozeß* ist jedoch nur selten ausführlich genug dokumentiert. Daher ist die Umsetzung dieser abstrakten Änderungen auf der Implementierungsebene fast genauso schwierig wie der Entwurf des ersten Prototyps. In der Tat sind *Wartung* von Software und eventuelle spätere *Erweiterungen* oft erheblich teurer als das Produkt selbst. All dies bezahlt im Endeffekt der Kunde – durch hohe Softwarepreise, Wartungsverträge oder die Notwendigkeit, "Upgrades" zu kaufen.

Das zweite und vielleicht bedeutendere Problem betrifft die *Zuverlässigkeit* von Software. Immer größer wird die Tendenz, die Steuerung komplizierter Prozesse in technischen Systemen einem Computerprogramm zu überlassen, da Menschen sie nicht mehr handhaben können oder wollen. Die Bandbreite der Anwendungen reicht von Waschmaschinenprogrammen und Autoelektronik (ABS, elektronische Motorsteuerung) bis hin zu Flugzeugsteuerung (Autopilot, "fly-by-wire"), Flugsicherung, U-Bahn-Netzen (automatischer Betrieb ohne Fahrer) und Sicherheitssystemen in Atomkraftwerken. Während man die Zuverlässigkeit der hierbei verwand-

ten Motoren, der mechanischen Steuerungen, der elektronischen Bauteile und der Computerhardware durch umfangreiche Tests sicherstellen kann, läßt die Unstetigkeit digitaler Systeme dies für die eingesetzte Software nicht zu. Bisher gibt es keinerlei praktikable Hilfsmittel, die Korrektheit von Software auf anderen Wegen zu garantieren. So verläßt man sich einfach darauf, daß die Produkte hinreichend gut sind, wenn während der Testphase keine der gängigen Fehler auftaucht.

Auch wenn es bisher kaum Unfälle gegeben hat, die eindeutig auf Softwarefehler zurückzuführen sind, glauben nur noch wenige Leute, daß im Angesicht der gegenwärtige Produktionsweise Software-verursachte Katastrophen gänzlich ausgeschlossen werden können. Spektakuläre Flugzeugunfälle zu Anfang der 90er Jahre wie der Absturz der Lauda Air Boeing 767 (Umkehrschub in der Luft) und der Air Inter French A 320 (unerklärlich zu tief geflogen) haben zu langen und gefährlichen Spekulationen über die tatsächlichen Ursachen geführt und Zweifel genährt, ob Software in sicherheitsrelevanten Bereichen überhaupt eingesetzt werden sollte.

Seit dem Aufkommen der Softwarekrise gibt es Bemühungen, zuverlässigere Methoden der Softwareentwicklung zu entwerfen, um den Einsatz von Software in diesen ökonomisch so wichtigen Bereichen zu ermöglichen. Methodiker beschrieben Programmierung als Kunst, Disziplin, Handwerk und Wissenschaft, um die verschiedenen Aspekte des Programmierprozesses hervorzuheben, zu systematisieren und somit zur Entwicklung besserer Software beizutragen.[1] Programmierung, so wird immer wieder hervorgehoben, ist weit mehr als die Beherrschung aller Feinheiten einer konkreten Programmiersprache, da beim eigentlichen Entwurf ganz andere Fähigkeiten eine Rolle spielen. Es kommt darauf an, kreativ Ideen zu entwerfen, sie auszuarbeiten und sich darüber klar zu werden, *warum* diese Idee eine Lösung des Problems darstellt. Neben Programmiererfahrung und Disziplin beim Präzisieren der Idee verlangt dies vor allem ein logisches Durchdenken und Analysieren von Problemen und Lösungen.

1.1 Programmierung als logischer Prozeß

Die Fähigkeit, zu *beweisen*, daß eine Idee tatsächlich auch funktioniert, spielt dabei im Hinblick auf die Zuverlässigkeit der erzeugten Programme ein fundamentale Rolle. Hierin aber liegt oft auch das größte Problem. Viele Programmierer haben keine klare Vorstellung davon, was Korrektheit wirklich bedeutet und wie man beweisen kann, daß ein Programm tatsächlich korrekt ist. Zudem hat das Wort "Beweis" für die meisten einen unangenehmen Beigeschmack, der viel Mathematik assoziiert. Sie sehen darin eher eine Behinderung ihrer Arbeit, die keinerlei Nutzen mit sich bringt. Wozu ein Programm beweisen, wenn es doch "funktioniert"?

Dies aber ist kurzsichtig gedacht. Zwar ist es richtig, daß es dem Entwickler wenig bringt, einen Beweis *nachträglich* zu führen. Es ist einfach zu schwer und oft auch nicht mehr praktisch durchführbar, denn bei der Beweisführung müssen alle Ideen, die zur Entwicklung des Programms beigetragen haben, rekonstruiert und mit dem endgültigen Programm in Beziehung gesetzt werden. Gewöhnt man sich aber an, bei der Implementierung *gleichzeitig* auch schon an einen möglichen Korrektheitsbeweis zu denken, so wird dies die Qualität des erstellten Programms erheblich steigern. Auch ist es effizienter, die Einsichten, die sich aus den Beweisideen ergeben, in die Implementierung einfließen zu lassen. Dies wollen wir an einem Beispiel illustrieren, auf das wir öfter noch zurückkommen werden.

Beispiel 1.1.1 (Maximale Segmentsumme)

Betrachten wir einmal das folgende einfache, aber doch realistische Programmierproblem

> *Zu einer gegebenen Folge $a_1, a_2, \ldots, a_n$ von n ganzen Zahlen soll die Summe $m = \sum_{i=p}^{q} a_i$ einer zusammenhängenden Teilfolge bestimmt werden, die maximal ist im Bezug auf alle möglichen Summen zusammenhängender Teilfolgen $a_j, a_{j+1} \ldots, a_k$.*

[1] Die Lehrbücher von Knuth [Knuth, 1968, Knuth, 1972, Knuth, 1975], Dijkstra [Dijkstra, 1976], Reynolds [Reynolds, 1981] und Gries [Gries, 1981] spielen in den Kursen über Programmierung heute noch eine wichtige Rolle. Lesenswert sind auch Abhandlungen von Wirth [Wirth, 1971] sowie Dahl, Dijkstra und Hoare [Dahl *et.al.*, 1972].

```
maxseg:INTEGER
  is
  local p, q, i, sum :INTEGER
  do
    from p := lower              -- variiere untere Grenze p
         Result := item(lower);
      until p >= upper
      loop
        from q := p              -- variiere obere Grenze q
          until q > upper
          loop
            from i := p ;        -- Berechne ∑_{i=p}^{q} a_i
                 sum := item(i) -- Initialwert der Summe
              until i = q
              loop
                  i := i+1;
                  sum := sum+item(i)
            end -- sum = ∑_{i=p}^{q} a_i
            if sum > Result
              then Result := sum
            end
            q := q+1
        end;
        p := p+1
    end
end
```

Abbildung 1.1: Berechnung der maximalen Segmentsumme: direkte Lösung

Derartige zusammenhängende Teilfolgen heißen *Segmente* und das Problem ist deshalb als das Problem der *maximalen Segmentsumme* bekanntgeworden. Für die Folge `-3, 2,-5, 3,-1, 2` ist zum Beispiel die maximale Segmentsumme die Zahl `4` und wird erreicht durch das Segment `3,-1, 2`.

Die vielleicht naheliegenste Lösung dieses Problems wäre, einfach alle möglichen Segmente und ihre Summen zu bestimmen und dann die größte Summe auszuwählen. Diese Lösung – in Abbildung 1.1 durch ein Eiffel-Programm beschrieben – ist jedoch weder elegant noch effizient, da die Anzahl der Rechenschritte für Folgen der Länge n in der Größenordnung von n^3 liegt – also 1 000 Schritte für Folgen der Länge 10 und schon 1 000 000 für Folgen der Länge 100. Es lohnt sich daher, das Problem systematisch anzugehen. Dazu formulieren wir es zunächst einmal in mathematischer Notation.

Zu einer Folge a der Länge n soll $\underline{M_n} = \texttt{max}(\{\sum_{i=p}^{q} a_i \mid 1 \leq \texttt{p} \leq \texttt{q} \leq \texttt{n}\})$ *berechnet werden.*

Da eine Lösung ohne Schleifen nicht möglich sein wird, bietet es sich an, nach einer induktiven Lösung zu suchen, bei der die Länge der Folge (wie in der direkten Lösung) natürlich eine Rolle spielen wird. Für einelementige Folgen ist nämlich die Lösung trivial – die maximale Segmentsumme ist der Wert des einzigen Elementes – und es besteht eine gewisse Hoffnung, daß wir das Problem einheitlich lösen können, wenn wir die (betrachtete) Folge um ein weiteres Element ergänzen.[2]

Nehmen wir also an, wir hätten für eine Folge der Länge n die maximale Segmentsumme M_n bereits bestimmt. Wenn wir nun ein neues Element a_{n+1} hinzufügen, dann ist die neue maximale Segmentsumme M_{n+1} entweder die Summe eines Segmentes, welches a_{n+1} enthält oder die Summe eines Segmentes, welches a_{n+1} nicht enthält.

Im ersten Fall müssen wir wissen, was die maximale Summe eines Segments ist, welches das letzte Element a_{n+1} enthält. Wir nennen diese Summe L_{n+1} und untersuchen sie später. Im zweiten Fall ist die Lösung einfach, da das letzte Element keine Rolle spielt – sie ist identisch mit M_n. Insgesamt wissen wir also, daß M_{n+1} das Maximum von L_{n+1} und M_n ist.

[2]Wir haben daher den Maximalwert M_n mit einem Index n versehen, der auf die Abhängigkeit von der Länge hinweist.

```
maxseg:INTEGER
  is
  local n, L_n :INTEGER
  do
    from n := lower;
         Result := item(n);
         L_n := item(n)
      until n >= upper
      invariant n<=upper                  -- ∧ Result=M_n ∧ L_n=L_n
      variant  upper - n
      loop
         if L_n > 0
            then L_n := L_n + item(n+1)
            else L_n := item(n+1)
         end;                          --  L_n = L_{n+1}
         if L_n > Result
            then  Result := L_n
         end;                          -- Result = M_{n+1}
         n := n+1
      end
end
```

Abbildung 1.2: Berechnung der maximalen Segmentsumme: systematisch erzeugte Lösung

Nun müssen wir noch L_{n+1} bestimmen, also die maximale Summe einer Teilfolge $a_i, a_{i+1} \ldots, a_{n+1}$. Da wir induktiv vorgehen, können wir davon ausgehen, daß L_n bereits aus dem vorhergehenden Schritt bekannt ist. Ist L_n negativ, dann ist die maximale Summe einer Folge, welche a_{n+1} enthält, die Summe der einelementigen Folge a_{n+1} (jede längere Folge hätte nur eine kleinere Summe). Andernfalls können wir "gewinnen", wenn wir die Folge a_{n+1} um das Segment ergänzen, dessen Summe L_n war, und erhalten L_{n+1} als Summe von L_n und a_{n+1}. Da L_1 offensichtlich a_1 ist, haben wir die Induktion auch verankert.

Dieses Argument sagt uns genau, wie wir eine Lösung berechnen und als korrekt nachweisen können. Der Algorithmus, der sich hieraus ergibt, ist in Abbildung 1.2 wiederum durch ein Eiffel-Programm beschrieben. Er läßt sich jetzt leicht verifizieren (wir müssen nur obiges Argument präzisieren) und ist darüber hinaus auch viel effizienter als der vorherige. Die Anzahl der Rechenschritte für Folgen der Länge n liegt nun in der Größenordnung von $3n$ – also nur etwa 300 für Folgen der Länge 100.

Dieses Beispiel zeigt, daß es sehr sinnvoll ist, in Programm und seinen Korrektheitsbeweis gleichzeitig zu entwickeln, wobei der *Beweis* die Vorgehensweise bestimmen sollte. Dabei muß ein Beweis im Prinzip weder mathematisch noch formal sein, sondern im wesentlichen nur *ein Argument, das den Leser von der Wahrheit eines Sachverhalts überzeugt*. Die Tatsache, daß Programmierer einen Großteil ihrer Zeit damit verbringen, Fehler aus ihren Programmen zu eliminieren – *obwohl* sie von der Richtigkeit ihrer Ideen überzeugt waren – zeigt jedoch, daß die derzeit vorherrschende 'Methode', sich von der Korrektheit eines Programms zu überzeugen, völlig unzureichend ist.

Der Grund hierfür wird relativ schnell klar, wenn man sich anschaut, welche Fähigkeiten bei der Entwicklung zuverlässiger Programme eigentlich gefragt sind. Das Beispiel zeigt, daß dies – neben Kreativität – vor allem die Fähigkeit ist, logische Schlußfolgerungen zu ziehen und dabei Wissen über Programme, Anwendungsbereiche und Programmiermethoden zu verwenden. Angesichts der Komplexität der zu erzeugenden Software ist diese Aufgabe für Menschen kaum zu bewältigen, denn sie neigen dazu, Flüchtigkeitsfehler in scheinbar unbedeutenden Details zu machen, die sich nachher fatal auswirken.

1.2 Maschinell unterstützte Softwareentwicklung

Die Aufgabe, komplexe Zusammenhänge fehlerfrei bis ins letzte Detail auszuarbeiten, stellt für menschliche Programmierer ein fast unlösbares Problem dar. Da Präzision bei der Behandlung komplexer Details aber

genau eine der wesentlichen Stärken von Computern ist, liegt der Gedanke nahe, den Programmierprozeß durch Computer unterstützen zu lassen und auf diese Art sicherzustellen, daß ein Programm die gewünschten Eigenschaften tatsächlich auch besitzt.

Ein erster Schritt im Hinblick auf eine rechnergestützte Entwicklung von Programmen ist das sogenannte *Computer-aided software engineering* (CASE). In speziellen Anwendugen können *Code-Generatoren* bereits benutzt werden, um Routinesoftware zu erzeugen. Werkzeuge für *Analyse und Entwurf* helfen, die Konsistenz modularer Systeme sicherzustellen. *Projekt Management Systeme* können dafür genutzt werden, größere Projekte unter Berücksichtigung aller Zusammenhänge zu planen und ihren Verlauf zu überwachen. Für die Praxis sind CASE Tools etwas sehr Nützliches, zumal die Technologie mittlerweile schon recht weit ausgereift ist. Dennoch sind sie keine Antwort auf das eigentliche Problem. Sie unterstützen nur spezielle Problemstellungen, nicht aber eine *logische* Verarbeitung von Wissen auf dem Weg von der Problemstellung zur Lösung. Es besteht somit keine Möglichkeit, zu dokumentieren, welche Ideen und Techniken bei der Entwicklung des Programms eine Schlüsselrolle gespielt haben. Derartiges Wissen aber ist wichtig, um die Korrektheit eines Softwaresystems zu überprüfen, Änderungen und Erweiterungen durchzuführen und einmal gewonnene Erkenntnisse bei der Entwicklung neuer Software wiederzuverwenden. Man benötigt weit mehr als die CASE Technologie, um eine wirkliche Unterstützung bei der Produktion zuverlässiger Software bereitzustellen.

Im Prinzip müßte man eine Automatisierung des gesamten Programmierprozesses anstreben und Werkzeuge für *wissensbasiertes Software Engineering*[3] entwickeln. Neben der Erzeugung *formaler* Spezifikationen aus informalen Anforderungen, was wir hier nicht weiter betrachten wollen,[4] bedeutet dies eine rechnergestützte *Synthese* von Programmen aus formalen Spezifikationen. Da Computer aber nur mit formalen Objekten operieren können, müssen hierzu im wesentlichen zwei Teilaufgaben bewältigt werden. Zum einen ist es nötig, jegliche Form von Wissen, das bei der Programmentwicklung eine Rolle spielt, zu *formalisieren* und *Kalküle* bereitzustellen, welche erlauben, das logische Schließen durch eine Manipulation formaler Objekte zu simulieren. Zum zweiten ist es sinnvoll, *Strategien* zu entwickeln, welche das formalisierte Wissen verwenden, um Schlüsse zum Teil automatisch auszuführen und einen Programmierer bei der Bearbeitung von formalen Detailfragen und Routineproblemen zu entlasten. Dabei ist es durchaus wünschenswert, einfache Programme vollautomatisch aus ihren Spezifikationen erzeugen zu können. Mit beiden Teilaufgaben werden wir uns im folgenden auseinandersetzen.

1.3 Formale Kalküle für Mathematik und Programmierung

Für eine rechnergestützte Synthese von Programmen ist es notwendig, ein breites Spektrum von Wissen aus Mathematik und Programmierung in formalisierter Form bereitzustellen. Ein Programmsynthesesystem sollte daher auf einem universell verwendbaren logischen Kalkül aufbauen, in dem man – zumindest im Prinzip – jegliche mathematische Aussage, einschließlich solcher über das Verhalten von Programmen, formal ausdrücken und ihren Wahrheitsgehalt untersuchen kann.

Die Entwicklung eines derartigen Kalküls, der es ermöglicht, natürlichsprachliche Aussagen in eine formale Sprache (*lingua rationalis*) zu übertragen und ihre Korrekheit mittels eines Berechnungsschemas (*calculus rationaticor*) zu überprüfen ist ein uralter Traum der Mathematiker, der mindestens auf die Zeit von G. W. Leibnitz (1646–1716) zurückgeht. Die Hoffnung war, auf diese Art Streitigkeiten unter Wissenschaftlern

[3] Dieser Begriff wurde erstmals geprägt in [Balzer *et.al.*, 1983].

[4] Es sei an dieser Stelle angemerkt, daß die Erstellung formaler Spezifikationen keineswegs trivial ist und daß eine Menge wichtiger Arbeiten auf diesem Gebiet geleistet wurden. Detaillierte Informationen hierzu kann man z.B. in [Hayes, 1987, Jones & Shaw, 1990, Kelly & Nonnenmann, 1991, Johnson & Feather, 1991] und [Lowry & Duran, 1989, Abschnitt B und D] finden. Wenn wir Softwareentwicklung als die Aufgabe betrachten, eine Brücke zwischen den Anforderungen und dem Programmcode zu bauen, so kann man die formale Spezifikation als einen Stützpfeiler in der Mitte ansehen. Indem wir uns die Konstruktion der zweiten Hälfte zum Ziel setzen, welche sich leichter formalisieren und durch Computer unterstützen läßt, unterstützen wir den Bau der ersten, da Zuverlässigkeitsanalyse und Wartung nun auf der Ebene der Spezifikation anstelle des Programmcodes durchgeführt werden können.

dadurch beseitigen zu können, daß man einfach durch Ausrechnen entscheidet, welche Aussagen wahr sind und welche nicht. In der Denkweise der heutigen Zeit bedeutet dies, daß man versuchte Maschinen zu bauen, die *entscheiden*, ob eine mathematische Aussage wahr ist oder nicht. Seit den dreißiger Jahren dieses Jahrhunderts weiß man jedoch, daß dies im Allgemeinfall nicht möglich ist. Probleme wie die Terminierung von Algorithmen oder das Entscheidungsproblem selbst sind unentscheidbar. Beweisverfahren, die mathematische Probleme entscheiden, sind also nur für spezielle Teiltheorien der Mathematik möglich. Will man dagegen *jegliche* Art mathematischer Aussagen formal untersuchen, so muß man sich darauf beschränken, Kalküle zu entwickeln, in denen formale Beweise geführt und durch Rechner *überprüft* werden können.[5]

Die axiomatische Mengentheorie, die Grundlage der modernen Mathematik, ist für diese Zwecke leider ungeeignet, da sie hochgradig unkonstruktiv ist, also den Begriff des *Algorithmus* kaum erfassen kann. Aus diesem Grunde wurden seit den siebziger Jahren *konstruktive Kalküle* entwickelt, die in Hinblick auf Universalität und Anwendungsbereich der klassischen Mengentheorie gleichkommen. Die wichtigsten Vertreter sind Martin-Löf's *intuitionistische prädikative Mengentheorie* (siehe [Martin-Löf, 1982, Martin-Löf, 1984]), der *Kalkül der Konstruktionen* von Coquand und Huet [Coquand & Huet, 1985, Coquand & Huet, 1988], Platek und Scott's *Logik berechenbarer Funktionen* und die *Lineare Logik* von Girard [Girard, 1987]. Im Gegensatz zur Mengentheorie, die im wesentlichen nur den Begriff der Menge und ihrer Elemente kennt, enthalten diese Theorien bereits formale Gegenstücke für die wichtigsten Standardkonzepte aus Mathematik und Programmierung – einschließlich eines Klassifizierungskonzepts wie Datentypen bzw. Sorten – und ermöglichen eine direkte Formalisierung von mathematischen Aussagen und Aussagen über Programme *und* ihre Eigenschaften.

Parallel zur Entwicklung von Kalkülen hat man sich bemüht, Rechnerunterstützung für das Erstellen und Überprüfen formaler Beweise zu schaffen. Beginnend mit dem *AUTOMATH* Projekt in Eindhoven (siehe [Bruijn, 1980]) wurden generische *proof-checker* implementiert, mit denen man der Automatisierung der Mathematik wieder ein Stück näher kam. In späteren Systemen wie *Edinburgh LCF* [Gordon *et.al.*, 1979], *PP*λ [Paulson, 1987]), *NuPRL* [Constable *et.al.*, 1986] und *ISABELLE* [Paulson, 1990] wurde dieses Konzept schrittweise zu *Beweiseditoren* ausgebaut, die man zur interaktiven Entwicklung von Beweisen verwenden kann, wobei es möglich ist, *Taktiken* (d.h. Suchprogramme) bei der Suche nach Beweisen zuhilfe zu nehmen.

Beweiseditoren für ausdruckstarke Kalküle, die es erlauben, jegliche Form logischen Schließens über mathematische Aussagen und Eigenschaften von Programmen auf dem Computer zu simulieren, gekoppelt mit der Möglichkeit, das logische Schließen durch programmierte Taktiken zum Teil zu automatisieren, scheinen für die rechnergestützte Entwicklung zuverlässiger Software das ideale Handwerkzeug zu sein.

Im Laufe dieser Veranstaltung werden wir uns im wesentlichen mit der *intuitionistischen Typentheorie* des NuPRL Systems befassen, die eine Weiterentwicklung der Martin-Löf'schen Theorie ist.[6] Im Verhältnis zu anderen ist die Theorie sehr reichhaltig und vor allem das zugehörige System ist schon sehr weit entwickelt, auch wenn es immer noch viele Möglichkeiten zu weiteren Verbesserungen gibt.

1.4 Strategien: Automatisierung von Entwurfsmethoden

Für die Implementierung von Strategien, die logische Schlüsse zum Teil selbständig ausführen und Routineprobleme eventuell vollautomatisch lösen können, spielt das eben erwähnte Konzept der *Taktiken* eine wesentliche Rolle. Taktiken sind ("Meta"-)Programme, welche die Anwendung elementarer Schlußregeln in einem ansonsten voll interaktiven Beweiseditor steuern und einen Anwender von der Bearbeitung trivialer, aber aufwendiger Detailprobleme entlasten. Diese Vorgehensweise sichert die Korrektheit aller ausgeführten logischen Schritte

[5] Durch den Gödelschen Unvollständigkeitssatz wurde nachgewiesen, daß es auch hier Grenzen gibt: zu jedem formalen System, das eine Formalisierung der *gesamten* Arithmetik erlaubt, kann man Aussagen formulieren, die in diesem System weder bewiesen noch widerlegt werden können. Diese Aussagen haben zum Glück wenig praktische Auswirkungen.

[6] Der Name resultiert daher, daß bei dieser Theorie die Datentypen (bzw. die Klassifizierung eines mathematischen Objektes) eine fundamentale Rolle spielen und bewußt eine Anlehnung an die elementaren Datenstrukturen von Programmiersprachen vorgenommen wurde.

und gibt dennoch eine große Flexibilität zum Experimentieren mit Beweis- und Programmentwicklungsstrategien.[7]

Die meisten Beweis- und Programmsyntheseverfahren wurden bisher allerdings unabhängig von den im letzten Abschnitt genannten Kalkülen entwickelt und implementiert. *Theorembeweiser*, die in der Lage sind, mathematische Beweise eigenständig zu finden, sind in ihrem Anwendungsbereich eingeschränkt auf Probleme, die sich gut in der Prädikatenlogik – eventuell ergänzt durch Gleichheit und einfache Induktion (siehe Abschnitt 2.2) – formulieren lassen. Viele *Programmsynthesestrategien* wurden zunächst als eher informale Methoden auf dem Papier entwickelt und dann als eigenständige Programme zur Manipulation von Formeln implementiert. Diese Vorgehensweise ist aus praktischen Gesichtspunkten zunächst einfacher, effizienter und erfolgversprechender. Das KIDS System [Smith & Lowry, 1990, Smith, 1991a, Smith & Parra, 1993] ist sogar schon nahe daran, für Routineprogrammierung verwendbar zu sein. Dennoch bleibt bei einer von formalen Kalkülen losgelösten Implementierung immer die Gefahr, daß Strategien, die auf dem Papier nachweisbar gute Eigenschaften besitzen, bei ihrer Implementierung Fehler enthalten und somit nicht die zuverlässige Software generieren, die man von ihnen erwartet. Es besteht daher die Notwendigkeit, diese Strategien durch eine Kombination von Taktiken und einer Formalisierung der "Theorie der Programmierung" in das Konzept der formalen Kalküle zu integrieren. Auf welche Art dies sicher und effizient geschehen kann, wird derzeit noch erforscht.

1.5 Aufbau der Veranstaltung

Die *Automatisierte Logik und Programmierung* ist als zweiteilige Veranstaltung ausgelegt. Der Schwerpunkt des ersten Teils wird das theoretische Fundament sein, also formale Kalküle, ihre Eigenschaften und der Bau von interaktiven Beweissystemen.

Im Kapitel 2 werden wir formale Kalküle als solche vorstellen und zur Formalisierung von Logik, Berechnung und Datentypen zunächst drei separate Kalküle vorstellen, die unabhängig voneinander entstanden sind. Wir werden dabei einige grundlegende Eigenschaften von Kalkülen besprechen, die notwendig sind, um die Sicherheit zu garantieren, die wir von formalen Systemen erwarten.

Die Typentheorie des NuPRL Systems, die wir im Kapitel 3 ausführlich diskutieren, bringt dann alles in einem einheitlichen Rahmen zusammen und ergänzt es um Formalisierungen der meisten Grundkonzepte, die in Programmiersprachen eine Rolle spielen. Da diese Theorie verhältnismäßig umfangreich ist, werden wir auch über eine Systematik beim Aufbau formaler Konzepte sprechen und über das sensible Gleichgewicht zwischen der Eleganz und der "Kontrollierbarkeit" formaler Theorien.

Im Kapitel 4 werden wir dann besprechen, wie auf der Basis eines formalen Kalküls zuverlässige interaktive Systeme gebaut werden können, mit denen ein Anwender beweisbar korrekte Schlüsse über Programme und ihre Eigenschaften ziehen kann.

In diesem ersten Teil werden theoretische Aspekte, Formalismen und Logik eine große Rolle spielen. Sie sollen lernen, Stärken und Schwächen sowie Grenzen und Möglichkeiten von Kalkülen einzuschätzen, Probleme zu formalisieren, formale Beweise zu führen und Beweiseditoren zu benutzen, auszubauen und im Hinblick auf Automatisierung zu ergänzen. In einem zweiten Teil wird es dann darum gehen, wie man auf der Grundlage dieses Fundaments mathematisches Wissen und eine formale Theorie der Programmierung "implementieren" kann und mit welchen konkreten Strategien sich eine Entwicklung korrekter Programme aus Spezifikationen unterstützen läßt.

[7] Dieses Konzept wurde erstmalig für *Edinburgh LCF* [Gordon *et.al.*, 1979] entwickelt und später in vielen Systemen übernommen und erfolgreich angewandt wie z.B. in *Cambridge LCF* [Paulson, 1987], *NuPRL* [Constable *et.al.*, 1986], *λ-PROLOG* [Felty & Miller, 1988, Felty & Miller, 1990], *OYSTER* [Bundy, 1989, Bundy *et.al.*, 1990], *ISABELLE* [Paulson, 1989, Paulson, 1990] *KIV* [Heisel *et.al.*, 1988, Heisel *et.al.*, 1990, Heisel *et.al.*, 1991], und *DEVA* [Lafontaine, 1990, Weber, 1990].

1.6 Literaturhinweise

Zur Typentheorie und ihren Wurzeln gibt es eine Fülle von Literatur, aber so gut wie keine Lehrbücher. Wir stützen uns im wesentlichen auf zwei zusammenfassende Darstellungen:

- Per Martin-Löf, *Intuitionistic Type Theory*, Bibliopolis, Napoli, 1984.
- R.L. Constable et.al, *Implementing Mathematics with the NuPRL Proof Development System*, Prentice Hall, Englewood Cliffs, 1986.

In beiden Büchern werden die Grundgedanken und viele Details der Typentheorie und des NuPRL Systems ausführlich erklärt. In den letzten 10 Jahren haben sich aber sowohl die Theorie als auch das System erheblich weiterentwickelt. Detailaspekte sind in vielen verschiedenen Schriften dokumentiert [Constable, 1984, Constable & Mendler, 1985, Mendler, 1987b, Constable & Smith, 1987, Allen, 1987a, Constable & Smith, 1988, Constable, 1989, Constable & Howe, 1990b]. Der aktuelle Stand ist derzeit nur in diesem Buch und den folgenden Manuals zu finden.

- The Nuprl Proof Development System, Version 4.1: Introductory Tutorial
- The Nuprl Proof Development System, Version 4.1: Reference Manual and User's Guide
- NuPRL's Metalanguage ML: Reference Manual and User's Guide

Es mag sich herausstellen, daß die Manuals noch Fehler aufweisen oder für Sie unverständlich sind. Für entsprechende konkrete Hinweise wäre ich sehr dankbar.

Für Interessierte lohnt es, die Entwicklung der Typentheorie, der konstruktiven Mathematik und der Logik weiter zurückzuverfolgen. Die Typentheorie von Martin-Löf [Martin-Löf, 1970, Martin-Löf, 1982] geht in ihren Zielen zurück auf die konstruktive Mathematik von L.E.J. Brouwer (siehe hierzu [Brouwer, 1908b, Brouwer, 1908a, Brouwer, 1924b, Brouwer, 1924a, Brouwer, 1925, Brouwer, 1926, Brouwer, 1927] – zu finden in [Brouwer, 1975]), A. Heyting [Heyting, 1934, Heyting, 1971], E. Bishop [Bishop, 1967, Bridges, 1979] und A.S. Troelsta [Troelsta, 1969, Troelsta, 1977]. Lesern, die sich für die historische Entwicklung von Mathematik und Logik als solche interessieren, sei die Lektüre von [Kneale & Kneale, 1962] nahegelegt.

Die frühesten Wurzeln der Typentheorien findet man schon bei Russel [Russel, 1908] und später in konstruktiver Form bei Church [Church, 1940]. Dennoch hat sich aufgrund der verschiedenen Detailprobleme immer noch keine "optimale" Formulierung herauskristallisiert. Die Unterschiede der verschiedenen konstruktiven Theorien liegen in Notationen, Effizienz der herleitbaren Algorithmen, Ausdruckskraft der Basissprache und Größe des Notwendigen formalen Apparates. Wir werden kaum dazu kommen, alle Varianten vorzustellen, die in manchen Bereichen sicherlich eleganter sind als die Typentheorie von NuPRL, in anderen dafür aber wiederum Schwächen aufweisen. Die wichtigsten Veröffentlichungen hierzu sind [Smith, 1984, Andrews, 1986, Girard, 1986, Coquand & Huet, 1988, Girard *et.al.*, 1989, Horn, 1988, Backhouse *et.al.*, 1988a, Backhouse *et.al.*, 1988b, Backhouse, 1989, Paulin-Mohring, 1989, Coquand, 1990, Galmiche, 1990] und das Buch [Nordström *et.al.*, 1990].

Weitere Literaturhinweise werden wir zu den einzelnen Themen am Ende jedes Kapitels geben.