Intuitionistische und konstruktive Logik

Simona Cabel
simona.cabel@gmx.de

Frankfurt am Main
Mai 2000

Dieser Vortrag wurde im Rahmen des Seminars *Methoden und Werkzeuge zur Verifikation von Software* im Sommersemester 2000 an der Johann Wolfgang Goethe - Universität Frankfurt am Main von mir angefertigt und gehalten.

Inhaltsverzeichnis

1 Was ist konstruktive Mathematik?

Die Bezeichnung *konstruktive Mathematik* wird benutzt, wenn die Existenz eines mathematischen Objekts explizit demonstriert werden muß, indem man das Objekt konstruiert. (In der klassischen Mathematik kann man indirekt beweisen, man muß das Objekt nicht konstruieren). Die *konstruktive oder intuitionistische Logik* ist ein Teil der klassischen Logik.
Für die Computerwissenschaft ist die konstruktive Mathematik interessant, weil ein konstruktiver Beweis, dass ein Objekt (Spezifikation einer Funktion) existiert, dieses Objekt (Funktion) liefert. D. h., gegeben ein konstruktiver Beweis für eine Spezifikation, kann man automatisch die Funktion angeben, die diese Spezifikation erfüllt. Es gibt mehrere Arten von Konstruktivismus:

- Bishop's konstruktive Mathematik
- Intuitionismus
- Rekursive Mathematik

Für diesen Vortrag interessant ist der **Intuitionismus (INT)**:
Der Intuitionismus ist der älteste Zweig der konstruktiven Mathematik. Die Idee ist: Man zeigt, dass ein Objekt existiert, indem man eine endliche Routine (Algorithmus) angibt, die das Objekt findet. Diese Routine soll nicht mit allgemeinen rekursiven Funktionen identifiziert werden. Die Beweise müssen direkt und konstruktiv sein.

2 Der intuitionistische Prädikatenkalkül

Eine Programmierungs-Theorie besteht aus einer Theorie für Funktionen und einer Theorie für Typen. Die Theorie für Funktionen sagt uns welche Funktionen es gibt und welche Eigenschaften sie haben. Das gleiche tut die Theorie für Typen. Solche Theorien müssen formell durch eine Sprache beschrieben werden, und das ist die Hauptaufgabe des Prädikatenkalküls. Der Kalkül ermöglicht die Formulierung von Axiomen über Funktionengleichheit und Typzugehörigkeit.
Mit Hilfe des Prädikatenkalküls werden Programme spezifiziert: Jedes Programm erfüllt eine Aufgabe. Um sagen zu können, ob ein Programm richtig läuft oder nicht, muß man das Ziel präzise und unabhängig beschreiben können. Dafür wurden Sprachen und Theorien für die Programm-Spezifikation entwickelt. Viele dieser Theorien basieren auf dem Prädikatenkalkül. Der Prädikatenkalkül ermöglicht eine abstrakte Beschreibung der Aufgabe, die das Programm zu erfüllen hat, ohne genauer zu sagen, wie diese Aufgabe gelöst werden kann. Wiederum kann man mit Hilfe der Prädikatenlogik feststellen, ob das Programm die Spezifikation erfüllt oder nicht. Das *intuitionistische Prädikatenkalkül* erleichtert die Konstruktion des Programms, weil man es aus dem Beweis der Spezifikation herausnehmen kann.

2.1 Die Sprache des Prädikatenkalküls

Der Prädikatenkalkül, das hier vorgestellt wird, ermöglicht die Beschreibung von Funktionen und ihren Typen. Er ermöglicht das Handeln mit Termen und Typen, Typzugehörigkeit, Gleichheit zwischen Termen, und evt. Programm-Terminierung. Es gibt daher drei syntaktische Klassen:

1. Wohl-definierte Formeln (wff)
2. Terme
3. Typen

2.1.1 Wohl-definierte Formeln (wff)

Die Sprache der wff besteht aus zwei Arten von Aussagen: Elementaraussagen und komplexe Aussagen, gebildet mit Hilfe von Junktoren $\wedge, \vee, \rightarrow$ und Quantoren $\forall, \exists$. Es gilt folgende Syntax:

$$\phi ::= \alpha|(\phi \wedge \psi)|(\phi \vee \psi)|(\phi \rightarrow \psi)|(\forall x \phi)|(\exists x \phi)$$

wobei (α) Elementaraussagen und alle anderen komplexe Aussagen sind. ϕ, ψ... bezeichnen wohl-definierte Formeln (wff), $u, v, x, y, z, f, g, ...$ sind Variablen. Wenn keine Klammerung vorhanden ist, dann setzt man Rechtsassoziativität voraus: $\phi \rightarrow \psi \rightarrow \eta$ entspricht $\phi \rightarrow (\psi \rightarrow \eta)$.
Es gibt drei Sorten von Elementaraussagen: die erste besagt, dass zwei Terme gleich sind, die zweite dass ein Term einen bestimmtem Typ hat, und die dritte ist eine Aussage über das Unmögliche:

$$\alpha ::= (t = s)|(t \in T)|\Omega$$

Die wff Ω ist für die Definition der Negation wichtig.

2.1.2 Terme

Die Sprache der Terme besteht aus Variablen und Konstanten:

$$t ::= x|c|...$$

wobei c, d, e Konstanten bezeichnen.

2.1.3 Typen

Typen werden benutzt, um Objekte als natürliche Zahlen, boolsche Werte, Buchstaben, oder komplexere Typen, zu beschreiben. Ein Beispiel für die Syntax ist:

$$T ::= Bool|Char|N|List[N]|(T \oplus S)|(T \otimes S)|(T \Rightarrow S)$$

Die drei Typenkonstruktoren sind: disjunkte Vereinigung $\oplus$, kartesisches Produkt $\otimes$ und Funktionenraum $\Rightarrow$. $T \otimes S$ ist der Typ von geordneten Paaren von Objekten, das erste vom Typ T und das zweite von Typ S. $T \oplus S$ ist der Typ von Objekten, die entweder von Typ T oder von Typ S sind. $T \Rightarrow S$ ist der Typ von Objekten, die Funktionen von Objekten von Typ T zu Objekten von Typ S sind.

2.1.4 Definierte Konstrukte

Die Negation definiert man wie folgt:

$$\sim \phi =_{def} \phi \rightarrow \Omega$$

Die Negation einer wff ϕ ist die Aussage, dass ϕ einen Widerspruch oder das Unmögliche impliziert.
Eine weitere Erweiterung der Sprache ist die Äquivalenz:

$$\phi \leftrightarrow \psi =_{def} (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$$

Die Einführung von Typen stellt ein Mittel zur Verfügung, um *gebundene Quantifizierung* auszudrücken. Für jeden Typ T, definiert man den *gebundenen Quantor* wie folgt:

$$(\forall x \in T)\phi =_{def} \forall x(x \in T \rightarrow \phi)$$

$$(\exists x \in T)\phi =_{def} \exists x(x \in T \wedge \phi)$$

Damit können All- und Existenzquantoren auf Elementen von einem besonderen Typ beschränkt werden.

2.1.5 Freie und gebundene Variablen

Bevor man Regeln für den Kalkül einführen kann, braucht man noch einige Begriffe, wie *Bindungsbereich eines Quantors*, *freie* und *gebundene* Variablen und *Substitution.*

Definition 1 *Jede wff die man benutzt hat, um eine wff ϕ zu bilden, heißt* **Teilformel** *von ϕ. Sei ϕ irgendeine wff mit Teilformel $(Qx\psi)$, wobei Q ($Q = \forall oder \exists$) ein Quantor ist. Der* **Bindungsbereich** *dieses Vorkommen von Q ist die Teilformel ψ.*

Zum Beispiel der Bindungsbereich des ersten Existenzquantors in der wff $[\exists x(x = y)] \vee [\exists x(x \in T)]$ ist die wff $x = y$; der Bindungsbereich des zweiten ist die wff $x \in T$.

Definition 2 *Ein Vorkommen einer Variable x in einer wff ϕ nennt man* **gebunden***, wenn das Vorkommen in einem Quantor Qx oder in dem Bindungsbereich eines Quantors mit der gleichen Variable ist. Ansonsten heißt das Vorkommen* **frei***. Wenn eine Variable ein freies Vorkommen in einer wff hat, dann sagt man, dass sie eine* **freie Variable** *dieser wff ist. Um zu zeigen, dass x und y frei in ϕ sind, schreibt man $\phi[x, y]$.*

Zum Beispiel ist x in der wff $\forall x(x = y)$ gebunden und y frei. In einer wff kann eine Variable sowohl frei als auch gebunden sein: x in $(\exists x(x = y)) \vee (x \in T)$

Definition 3 *Die* **Substitution** *einer Variable x durch einen Term t in einer wff ϕ besteht aus dem gleichzeitigen Ersetzen der freien Vorkommen von x in ϕ durch t. Man schreibt dies als $\phi[t/x]$ (oder einfach $\phi[t]$). Ein Term t ist* **frei für x** *in $\phi[x]$, wenn jedes freie Vorkommen von x in ϕ durch t substituiert werden kann, ohne dass unerwünschte Quantorbindungen auftreten.*

Zum Beispiel wenn $\phi[x, y]$ die wff $(\exists x(x = y)) \wedge (x \in T)$, dann ist $\phi[t, y]$ die wff $(\exists x(x = y)) \wedge (t \in T)$. x ist nicht frei für y in $\forall x \phi[x, y]$.

2.2 Intuitionistische Logik

Um den Kalkül zu benutzen braucht man eine *Logik* für die Bindewörter. Mit jedem Bindewort assoziert man zwei Beweisregeln. Die erste (Einfügeregel (i)) ermöglicht das Einfügen einer komplexen wff als Folgerung einer Herleitung, indem ein besonderes Bindewort benutzt wird. Die zweite (Eliminationsregel (e)) ermöglicht das Eliminieren eines Bindewortes aus der Folgerung.

2.2.1 Konjunktion

Die Einfüge- und Eliminationsregeln für die Konjunktion sind die einfachsten Regeln. Um die Regeln anzuzeigen benutzt man $\wedge i$ bzw. $\wedge e$:

Einfügen der Konjunktion	Elimination der Konjunktion
$(\wedge i) \quad \dfrac{\begin{matrix}\pi & \pi' \\ \phi & \psi\end{matrix}}{\phi \wedge \psi}$	$(\wedge e) \quad \dfrac{\begin{matrix}\pi \\ \phi \wedge \psi\end{matrix}}{\phi} \qquad \dfrac{\begin{matrix}\pi \\ \phi \wedge \psi\end{matrix}}{\psi}$

Bei der ersten Regel $(\wedge i)$ hat man Herleitungen π und π' mit den Folgerungen ϕ bzw. ψ, woraus man $\phi \wedge \psi$ schließen kann. D.h. gegeben Herleitungen für die Voraussetzung, generiert die Einfügeregel eine neue Herleitung mit der Schlussfolgerung $\phi \wedge \psi$. Die Eliminationsregel operiert entgegengesetzt: gegeben eine Herleitung von $\phi \wedge \psi$ kann man ϕ (oder ψ) folgern.

2.2.2 Implikation

Es gibt wieder zwei Regeln. Einfügeregel: Angenommen man hat ψ durch wiederholten Aufruf der Voraussetzung ϕ gebildet. Dann hat man eigentlich gezeigt, wie man eine Herleitung von ψ aus einer hypothetischen für ϕ konstruiert. Die Eliminationregel ermöglicht es, aus zwei Herleitungen (eine für $\phi \to \psi$ und eine für ϕ) ψ zu folgern.

Einfügen der Implikation	Elimination der Implikation
$(\to i)\ \dfrac{\begin{matrix}[\phi]\\ \pi\\ \psi\end{matrix}}{\phi\to\psi}$	$(\to e)\ \dfrac{\begin{matrix}\pi\\ \phi\end{matrix}\quad\begin{matrix}\pi'\\ \phi\to\psi\end{matrix}}{\psi}$

Bei der Einfügeregel ist ϕ wie eine temporäre Annahme, aus der man eine Herleitung für ψ bilden kann. Aus dieser Herleitung kann man $\phi \to \psi$ folgern, während man die temporäre Annahme ϕ entläd (verwirft). Die Notation $[\phi]$ zeigt an, dass ϕ entladen ist. Das Entladen von Annahmen ist optional [1]. Anders gesagt: gegeben eine Herleitung von ψ, kann man $\phi \to \psi$ herleiten und gleichzeitig einige (oder alle) Vorkommen von ϕ entladen. Folgender Fall von $(\to i)$ gilt auch, da ψ eine Herleitung ist:

$$\frac{\psi}{\phi \to \psi}$$

2.2.3 Disjunktion

Einfügen der Disjunktion	Elimination der Disjunktion
$(\vee i)\ \dfrac{\begin{matrix}\pi\\ \phi\end{matrix}}{\phi\vee\psi}\ \dfrac{\begin{matrix}\pi\\ \psi\end{matrix}}{\phi\vee\psi}$	$(\vee e)\ \dfrac{\begin{matrix}\pi\\ \phi\vee\psi\end{matrix}\quad\begin{matrix}[\phi]\\ \pi'\\ \eta\end{matrix}\quad\begin{matrix}[\psi]\\ \pi''\\ \eta\end{matrix}}{\eta}$

Für die Einfügeregel gilt: Wenn man eine Herleitung von ϕ oder eine von ψ hat, dann hat man eine von $\phi \vee \psi$. Die Eliminationsregel besagt: Angenommen, man kann aus der Voraussetzung ϕ und ψ je eine Herleitung η bilden. Dann kann man η aus $\phi \vee \psi$ folgern, und die Voraussetzungen ϕ und ψ entladen. D.h., wenn man η aus ϕ oder ψ folgern kann, dann auch aus $\phi \vee \psi$.

2.2.4 Negation und das Unmögliche

Die Negation ist definiert als: $\sim \phi =_{def} \phi \to \Omega$, wobei Ω das Unmögliche ist. Die Regeln für die Implikation liefern zwei hergeleitete Regeln für die Negation:

Einfügen der Negation	Elimination der Negation
$(\sim i)\ \dfrac{\begin{matrix}[\phi]\\ \pi\\ \Omega\end{matrix}}{\sim\phi}$	$(\sim e)\ \dfrac{\begin{matrix}\pi\\ \phi\end{matrix}\quad\begin{matrix}\pi'\\ \sim\phi\end{matrix}}{\Omega}$

Die hergeleitete Eliminationsregel erlaubt eine Folgerung Ω aus Herleitungen einer wff und ihrer Negation (ϕ und $\sim \phi$). Die Einfügeregel ähnelt der klassischen Regel für Negation, was man nicht verwechseln sollte. In der klassischen Logik ist die Negation wie folgt definiert:

klassische Negation

$$(NEG)\ \frac{\begin{matrix}[\sim\phi]\\ |\\ \Omega\end{matrix}}{\phi}$$

[1] Wenn man alle temporären Annahmen entlädt, dann bekommt man einen Satz

Die nächste Regel ist für das Unmögliche:

Das Unmögliche

$$(absurd) \quad \frac{\begin{matrix}\pi\\ \Omega\end{matrix}}{\phi}$$

Diese Regel besagt, dass man aus einem Widerspruch (oder dem Unmöglichen) alles folgern kann. Wenn man in einem Beweis Ω folgert, dann hat man einen Fehler gemacht, oder man kann folgern, was man will.
Diese Regeln werden nicht oft benutzt.

2.2.5 Allquantor

Die Regeln für Quantoren sind etwas schwieriger. Für den Allquantor gibt es folgende zwei:

Einfügen des Allquantors	Elimination des Allquantors
$(\forall i) \quad \frac{\begin{matrix}\pi\\ \phi\end{matrix}}{\forall x\phi}$	$(\forall e) \quad \frac{\begin{matrix}\pi\\ \forall x\phi\end{matrix}}{\phi[t/x]}$

Bei der Einfügeregel muß die Herleitung von ϕ so sein, dass die Variable x nicht frei ist in irgendeiner nicht entladenen wff, von welcher ϕ abhängt (d.h. in einer offenen Annahme von welcher $\phi[x]$ abhängt); die Herleitung ist dann unabhängig von x und die wff ist für jedes Objekt gültig. Die Eliminationsregel erlaubt als Folgerung einer universellen Aussage einen besonderen Fall dieser Aussage. Bei dieser Regel muß t frei für x in ϕ sein, um bei der Substitution unerwünschte Bindungen der Variablen zu vermeiden.

2.2.6 Existenzquantor

Diese Regeln benutzen die Entladung von temporären Annahmen (Voraussetzungen):

Einfügen des Existenzquantors	Elimination des Existenzquantors
$(\exists i) \quad \frac{\begin{matrix}\pi\\ \phi[t/x]\end{matrix}}{\exists x\phi}$	$(\exists e) \quad \frac{\begin{matrix} & [\phi]\\ \pi & \pi'\\ \exists x\phi & \eta\end{matrix}}{\eta}$

In der Einfügeregel muß t frei für x in ϕ sein, um unerwünschte Bindungen zu vermeiden. Bei der Eliminationsregel wird eine neue wff eingeführt, die nicht Teil der wff ist, die man untersucht. Beim Folgern kann ϕ entladen werden. x darf nicht frei sein in η und in jeder Annahme (außer ϕ), von welcher die oberen Vorkommen von η abhängen. Das wird verlangt, damit die Herleitung von η unabhängig von x ist. Die Eliminationsregel besagt: wenn man η aus ϕ folgern kann (die Einschränkung garantiert, dass die Herleitung unabhängig von x ist), dann kann man η auch aus der Aussage, dass manche x existieren, die ϕ erfüllen, folgern.

Das sind alle Beweisregeln für Junktoren und Quantoren. Die Entladung wurde induktiv definiert.
Eine Herleitung ist entweder eine Elementaraussage (eine atomische wff), oder man bekommt sie aus einer vorhandenen Herleitung, indem man die Regeln für Konjunktion, Disjunktion, usw. anwendet. Diese Beweisregeln bilden die *intuitionistische Prädikatenlogik* (IPC). Aus der intuitionistischen Prädikatenlogik erhält man die klassische Prädikatenlogik (CPC), wenn man eine der folgenden drei Regeln hinzunimmt:

1. Klassische Negation (NEG)
2. Das Gesetz der ausgeschlossenen Mitte (LEM)
3. Doppelte Negation (DN)

$$\begin{array}{ccc} \text{NEG} & \text{LEM} & \text{DN} \\ \dfrac{\begin{array}{c}[\sim\phi] \\ | \\ \Omega\end{array}}{\phi} & \dfrac{}{\sim\phi\vee\phi} & \dfrac{\sim\sim\phi}{\phi} \end{array}$$

IPC und eine dieser Regeln ermöglichen die Herleitung der anderen zwei. Diese drei Regeln machen den Unterschied zwischen IPC und CPC aus. (Dadurch, dass in der IPC die doppelte Negation fehlt, ist es nicht möglich einen Beweis durch Widerspruch zu führen; man muß konstruktiv beweisen.)

3 Spezifikation des Programms

Wenn man eine Funktion spezifiziert, dann stellt man zwei Bedingungen: eine pre- und eine post-Bedingnug der Funktion. In der IPC werden Funktionen wie folgt spezifiziert:

$$\forall x(\phi[x] \to \exists y \psi[x, y])$$

wobei ϕ die pre-Bedingung und ψ die post-Bedingung ist. Die Spezifikation verlangt, dass für jedes Objekt x, das ϕ erfüllt, ein Objekt y existiert, was mit x zusammen ψ erfüllt.
Als Beispiel die Spezifikation einer Funktion zum Sortieren von Listen von natürlichen Zahlen:

$$\forall x[x \in List[N] \to \exists y(y \in List[N] \wedge O[y] \wedge P[x, y])]$$

Die wff $O[y]$ zeigt, ob die Liste y sortiert ist. Die wff $P[x, y]$ zeigt, dass die Liste x eine Permutation von y ist. Die Formel besagt, dass zu jeder Liste eine geordnete Permutation dieser Liste existiert. Die Sortier-Funktion ist dadurch implizit definiert. Um sicher zu sein, dass diese Funktion existiert, muß man beweisen, dass die Spezifikation wahr ist.

4 Realisierbarkeit

Für jede Behauptung in der intuitionistischen Logik braucht man ein Mittel, um sie zu verifizieren. Ein solches Mittel sind Programme. Um sie benutzen zu können, muß man zuerst erklären, unter welchen notwendigen und hinreichenden Bedingungen ein Programm die Spezifikation realisiert (Wie muß das Programm aussehen?). Man muß also eine Semantik beschreiben.

4.1 Definition der Realisierbarkeit

Aus der Prädikatenlogik erhält man eine Semantic, indem man mit jeder wff ϕ eine neue wff $z\rho\phi$ assoziiert, wobei z eine neue, in ϕ nicht freie Variable ist. $z\rho\phi$ heißt dann *z realisiert ϕ*. Die Assoziation verhält sich wie eine rekursive Translation zwischen den wff.
Für jedes Bindewort gibt es eine genaue Translationsregel:

(a) Konjunktion Angenommen man muß ein Programm schreiben, das ein Problem löst, welches als eine Konjunktion von zwei Teilproblemen spezifiziert ist. Dieses Programm muß dann aus ein Paar von Programmen bestehen, eins zur Lösung des ersten und eins zur Lösung des zweiten Teilproblems. Diese Tatsache führt zur folgenden Translationsregel:

$$\text{R1. } z\rho(\phi \wedge \psi) \text{ ist } [z \in \text{PAIR} \wedge lz\rho\phi \wedge rz\rho\psi]$$

(b) Disjunktion Ein Programm, das ein als Disjunktion zweier Teilprobleme gegebenes Problem $\phi \vee \psi$ löst, ist eine Injektion. Je nachdem, ob das Programm eine Links- oder Rechtsinjektion ist, ist seine Projektion eine Lösung für ϕ oder ψ. Die Art der Injektion zeigt an, welches Teilproblem man meint:

$$\text{R2. } z\rho(\phi \vee \psi) \text{ ist } [z \in \text{INJ} \wedge (z \in \text{INL} \rightarrow (\text{proj } z)\rho\phi) \wedge (z \in \text{INR} \rightarrow (\text{proj } z)\rho\psi)]$$

(c) Implikation Um ein als Implikation gegebenes Problem zu lösen, muß man eine Prozedur finden, die aus einer Lösung für den Vorgänger eine Lösung für den Nachfolger macht.

$$\text{R3. } z\rho(\phi \rightarrow \psi) \text{ ist } \forall x(x\rho\phi \rightarrow zx\rho\psi)$$

(d) Allquantor Ein Programm zur Lösung oder Verifikation des Problems $\forall x\phi$ muß eine Funktion sein, die gegeben irgendein Element x, ein Programm erzeugt, das die entsprechende Instanz von ϕ löst.

$$\text{R4. } z\rho(\forall x\phi) \text{ ist } \forall x(zx\rho\phi)$$

(e) Existenzquantor Um das Problem $\exists x\phi$ zu lösen, muß man das Element finden das ϕ erfüllt und auch zeigen, dass das gefundene Element ϕ erfüllt.

$$\text{R5. } z\rho(\exists x\phi) \text{ ist } [z \in \text{PAIR} \wedge lz\rho\phi[rz/x]]$$

(f) Elementaraussagen Elementaraussagen sind nur dann realisierbar, wenn sie wahr sind.

$$\text{R6. } z\rho\phi \text{ ist } \phi \text{ für Elementaraussagen } \phi$$

Die Regeln R1 bis R6 ermöglichen eine rekursive Translation zwischen wff's. Zu jeder wff ϕ wird rekursiv eine neue wff $z\rho\phi$ assoziiert. Die neue wff hat als freie Variable z (ausgenommen bei Elementarsussagen) und die freien Variablen von ϕ. Eine weitere nützliche Regel liefert der nächste Satz:

Satz 1 *Für jede wff ϕ, Variable x und Term s gilt:*

$$z\rho(\phi[s/x]) \leftrightarrow (z\rho\phi)[s/x]$$

Diese Interpretation der intuitionistischen Logik entspricht nicht genau dem, was man ursprünglich mit der logischen Formel meinte. Deshalb gibt es mehrere Arten von Realisierbarkeit, auf denen ich hier nicht näher eingehen werde. (siehe Kap. 11 im Buch)
Die Realisierbarkeit selbst liefert kein Schema mit dem man aus einem Beweis ein Programm bekommt. Sie beschreibt nur Kriterien für Korrektheit.

4.2 Strukturelle induktive Typen

Der interessanteste Fall von Realisierbarkeit betrifft induktive Typen. Hier wird nur ein spezial Fall betrachtet, weil der allgemeine Fall zu unübersichtlich ist. Man betrachte den induktiven Typ I, gegeben wie folgt:

Abschluß (Axiom für I)

$$\forall x \in T(cx \in I) \wedge \forall u \in I \forall v \in I(duv \in I)$$

$$\text{Induktion}\quad \frac{\forall x{\in}T(\phi[cx]) \quad \forall u{\in}I\forall v{\in}I((\phi[u]\wedge\phi[v])\rightarrow\phi[duv])}{\forall y{\in}I(\phi[y])}$$

Das Induktionsaxiom wurde einfachheitshalber wie eine Regel geschrieben.
Wenn man die Realisierbarkeit dieser zwei Formeln betrachtet, merkt man zuerst, dass das Abschluß-Axiom eine negative Formel [2] ist. Negative Formeln haben die Eigenschaft, dass sie wahr sind, wenn sie realisierbar sind, und wenn sie realisierbar sind, dann durch einen kannonischen Term, der nur von der Formel abhängt. Also muß man für die erste Formel nichts beweisen. Für die Induktion nehme man an, dass die Voraussetzungen durch f bzw. g realisiert werden. Man betrachte dann folgende Funktion:

$$\begin{aligned}
\text{rec }(cx)z &= fx(t[x \in T]) \\
\text{rec }(duw)z &= gu(t[u \in I])w(t[w \in I])((\text{rec } uz), (\text{rec } wz))
\end{aligned}$$

Man benutzt die Induktion I, um zu beweisen, dass $\forall x \in I((\text{rec } xz)\rho\phi[x])$. Die Induktionsformel sei $(\text{rec } xz)\rho\phi[x]$. Zuerst muß man zeigen, dass $\forall x \in T(fx(t[x \in T])\rho\phi[cx])$. Das folgt, weil f die erste Voraussetzung realisiert und weil $t[x \in T]$ der kannonische Realisierer für die Zugehörigkeit in T ist. Induktionsschritt: Man nehme an, dass $(\text{rec } uz)\rho\phi[u]$ und $(\text{rec } wz)\rho\phi[w]$ gelten. Durch die Vereinbarung für g gilt $(\text{rec } (duw)z)\rho\phi[duw]$. Also gilt durch Induktion, dass $\forall x \in I((\text{rec } xz)\rho\phi[x])$. Aus der Definition der Realisierbarkeit erhält man dann die Funktion $\lambda xy.(\text{rec } xz)\rho\forall x \in I(\phi[x])$, wobei y eine neue Variable ist.
Wenn man natürliche Zahlen und Listen betrachtet, bekommt man die folgenden rekursiven Funktionen:

$$\begin{aligned}
\text{rec } 0z &= f \\
\text{rec } x'z &= gx(t[x \in N])(\text{rec } xz) \\
\\
\text{rec } [\,]z &= f \\
\text{rec }(u : x)z &= gu(t[u \in T])x(t[x \in \text{List}[T]])(\text{rec } xz)
\end{aligned}$$

Im Folgenden wird die *konstruktive funktionale Programmierung* mit Hilfe der intuitionistischen Logik eingeführt, indem man einen Algorithmus für **insertion sort** entwickelt.

5 Einführung in die konstruktive funktionale Programmierung

In der intuitionistischen Logik ist eine Programm-Spezifikation eine implizite Behauptung mit dem Inhalt, dass eine Funktion existiert. Aus einem konstruktiven Beweis einer solchen Spezifikation, kann die benötigte Funktion automatisch entnommen werden. Dies wird bei der konstruktiven funktionalen Programmierung gemacht. Beim Programmieren geht es also darum, eine Konstruktion in der intuitionistischen Logik zu beweisen. Zum Programm kommt man in drei Schritten:

1. Spezifikation des Programms

2. Existenzbeweis

3. Extraktion des Programms

[2]Eine Formel ϕ heißt negativ (oder $\exists, \vee$ frei), wenn ϕ nicht $\exists$ oder $\vee$ enthält.

Einige Worte zu diesen Schritten:
Spezifikation des Programms Eine Funktion wird üblicherweise durch zwei Bedingungen spezifiziert, die so genannten Vor- und Nachbedingung einer Funktion. Die Nachbedingung ist eine Aussage über die benötigte Beziehung zwischen der Eingabe und Ausgabe der Funktion, unter der Beachtung der Vorbedingung. Die Spezifikation einer Funktion sieht dann so aus:

$$\forall x(\phi[x] \rightarrow \psi[x, fx])$$

wobei ϕ die Vorbedingung und ψ die Nachbedingung ist. (ϕ und ψ sind Formeln)
Wenn man die Realisierbarkeit betrachtet, genügt es die Funktion implizit zu spezifizieren:

$$\forall x(\phi[x] \rightarrow \exists y\psi[x, y])$$

Im Vergleich zur üblicheren Spezifikation muß hier die Vorbedingung negativ sein. Für Funktionen, die mehr als ein Argument haben, sieht es dann so aus:

$$\forall x_1 \in T_1 \cdots x_n \in T_n(\phi[x_1, \cdots, x_n] \rightarrow \exists y \in S\psi[x_1, \cdots, x_n, y])$$

Existenzbeweis In zweiten Schritt betrachtet man die Spezifikation als ein Theorem, was bewiesen werden muß; der Beweis des Theorems beläuft sich zu einer Rechtfertigung, dass die Spezifikation möglich ist. Dies ist der schwierigste Schritt. Von Hand ist die Konstruktion der Beweise eine ziemlich ermüdende Tätigkeit. Generell bestimmt die Art des Beweises die Art des Programms, das im letzten Schritt extrahiert wird.
Extraktion des Programms Dieser letzte Schritt ist streng genommen nicht Teil der Programmierarbeit; es ist automatisch, wenn man die Ausgabe vom zweiten Schritt hat. Es gibt eine Extraktion-Funktion F, die wie ein Compiler arbeitet und das benötigte Programm hervorbringt, wenn der Beweis gegeben ist. Programmieren ist dadurch reduziert zum Beweisen der Existenz. Vielmehr, der Existenzbeweis im Zusammenhang mit dem Korrektheitsbeweis für F garantiert automatisch, dass das hervorgebrachte Programm die Spezifikation erfüllt.

5.1 Insertion Sort

Im Allgemeinen finden Programm-Spezifikationen nicht isoliert statt. Normalerweise sind sie von bestimmten vordefinierten Funktionen und Relationen verfolgt. Diese Funktionen können selbst spezifiziert und hergeleitet sein, aber wir beschäftigen uns jetzt mit der Entwicklung des Sortier-Algorithmus, und nehmen an, dass bestimmte grundlegende Funktionen schon vorhanden sind.
Insbesondere brauchen wir zwei boolsche Funktionen. Die erste bestimmt, ob eine Liste geordnet ist oder nicht, und die zweite ob eine Liste eine Permutation einer anderen Liste ist. Um es einfacher zu machen, arbeiten wir mit Listen von Zahlen mit der üblichen Ordnung.

ord[ ]	=	true	
ord[n]	=	true	
ord m:n:x	=	ord n:x	wenn $m \leq n$
ord m:n:x	=	false	wenn $n < m$
perm x [ ]	=	empty x	
perm x (m:y)	=	(member m x) und (perm(remove m x)y)	
wobei			
remove m [ ]	=	[ ]	
remove m (n:x)	=	x	wenn m=n
remove m (n:x)	=	n:(remove m x)	wenn nicht(m=n)

Im Folgenden bezeichen wir $(ordx) = true$ mit $O(x)$ und $(perm\ xy) = true$ mit $P(x, y)$. Diese Relationen brauchen wir für die Spezifikationen von *insertion sort*. Im Laufe der Programmentwicklung werden wir bestimmte hergeleitete Eigenschaften dieser Relationen benutzen. Insbesondere benutzen wir folgende Eigenschaften von P:

(a) $\forall x \in List[N]\{P(x, x)\}$
(b) $\forall x \in List[N] \forall y \in List[N]\{P(x, y) \rightarrow P(y, x)\}$
(c) $\forall x \in List[N] \forall y \in List[N] \forall z \in List[N]\{P(x, y) \wedge P(y, z) \rightarrow P(x, z)\}$
(d) $\forall x \in List[N] \forall y \in List[N] \forall u \in List[N] \forall v \in List[N]\{P(x, y) \wedge P(u, v)$
$\rightarrow P(x * u, y * v)\}$

Wenn man ab jetzt auf solche hergeleitete Eigenschaften zurückgreift, zeigt man das, indem man $P \wedge O$ schreibt. Diese Schritte spielen die Hintergrundrolle in der aktuellen Entwicklung von Sortier-Algorithmen und deren Beweise. Man könnte auch anders vorgehen, indem man eine abstrakte Spezifikation für diese Relationen liefert, in welchem Fall dann (a) bis (d) ein Teil der Spezifikation bilden. Da wir uns vorrangig mit der Entwicklung des Sortier-Algorithmuses beschäftigen, ist es unwichtig welchen Weg wir nehmen.

5.1.1 Die Spezifikation

Die Spezifikation für *insertion sort* ist:

$$\forall x \in List[N] \exists y \in List[N]\{O(y) \wedge P(x, y)\} \quad \text{(a)}$$

d.h. für jede Liste gibt es eine geordnete Permutation dieser Liste. Bemerke: weil $List[N]$ total ist, spezifiziert sie eine totale Funktion.
Um die Behauptung (a) zu beweisen, muß man sich für eine Beweisstrategie entscheiden. Am leichtesten ist es mit Listeninduktion. Für diesen Beweis braucht man noch folgendes Lemma:

$$\forall n \in N \forall y \in List[N]\{O(y) \rightarrow \exists z \in List[N](P(n : y, z) \wedge O(z))\} \quad \text{(b)}$$

D. h. für jede natürliche Zahl n und jede geordnete Liste y, gibt es eine geordnete Liste, die eine Permutation von n:y ist.
Dies beendet die Spezifikationsphase.

5.1.2 Existenzbeweis

In diesem konstruktiven Beispiel ist der Hauptschritt der Programmentwicklung die Konstruktion der Beweise. Im Grunde genommen zeigt man, dass solche Spezifikationen erfüllt werden können. Für dieses Beispiel müssen zwei Beweise gemacht werden: für (a) und für (b), wobei (b) zuerst bewiesen wird.

Satz 2 $\forall n \in N \forall y \in List[N]\{O[y] \rightarrow \exists z \in List[N](P(n : y, z) \wedge O(z))\}$

Beweis Einfachheitshalber wird jetzt List[N] mit L abgekürzt. Der Hauptteil des Beweises besteht aus einer Listeninduktion, mit der Induktionsformel:

$$\phi[y] = \{O[y] \rightarrow \exists z \in List[N](P(n : y, z) \wedge O(z))\}$$

$[n \in N]$	(1)	Voraussetzung
Grundschritt		
$[n] \in L \wedge n : [\,] = [n] \wedge P(n : [\,], [n]) \wedge O([n])$	(2)	$P \wedge O$
$\exists z \in L(P(n : [\,], z) \wedge O(z))$	(3)	$\exists i, (2)$
$O([\,]) \rightarrow \exists z \in L(P(n : [\,], z) \wedge O(z))$	(4)	$\rightarrow i$, (3)
Induktionsschritt		
$[m \in N]$	(5)	Voraussetzung
$[y \in L]$	(6)	Voraussetzung
$[\phi[y]]$	(7)	Induktionsannahme
$[O(m : y)]$	(8)	Voraussetzung
$n \leq m \vee m < n$	(9)	Satz
$[m < n]$	(10)	Voraussetzung
$O[y]$	(11)	(8), $P \wedge O$
$\exists z \in L(P(n : y, z) \wedge O(z))$	(12)	$\rightarrow e$, (7)
$[\exists z \in L \wedge (P(n : y, z) \wedge O(z))]$	(13)	Voraussetzung
$(m : z) \in L \wedge P(n : m : y, m : z) \wedge O(m : z)$	(14)	(13), (10), (8), $P \wedge O$
$\exists z \in L(P(n : m : y, z) \wedge O(z))$	(15)	$\exists i$, (14)
$\exists z \in L(P(n : m : y, z) \wedge O(z))$	(16)	$\exists e$, (15), (13), (12)
$[n \leq m]$	(17)	Voraussetzung
$O(n : m : y)$	(18)	$P \wedge O$, (8), (17)
$\exists z \in L(P(n : m : y, z) \wedge O(z))$	(19)	$\exists i$, (18), $P \wedge O$
$\exists z \in L(P(n : m : y, z) \wedge O(z))$	(20)	$\wedge e$, (19), (16), (9)
$O(m : y) \rightarrow \exists z \in L(P(n : m : y, z) \wedge O(z))$	(21)	$\rightarrow i$, (20), (8)
$\phi[y] \rightarrow \phi[m : y]$	(22)	$\rightarrow i$, (21), (7)
$\forall m \in N \forall y \in L(\phi[y] \rightarrow \phi[m : y])$	(23)	$\rightarrow i, \forall i$, (22), (5), (6)
Der Induktionsschritt ist jetzt beendet.		
$\forall y \in L\{\phi[y]\}$	(24)	(4), (23), Induktion
$\forall n \in N \forall y \in L\{\phi[y]\}$	(25)	$\rightarrow i, \forall i$, (1), (24)

Damit ist (b) bewiesen. Nun kann man (a) beweisen.

Satz 3 $\forall x \in L \exists z \in L\{P(x, z) \wedge O(z)\}$

Beweis Durch Induktion nach x. Als Induktionsformel wählen wir:

$$\phi[x] = \exists z \in L\{P(x, z) \wedge O(z)\}$$

Grundschritt		
$P([\,], [\,]) \wedge O([\,])$	(1)	$P \wedge O$
$[\,] \in L \wedge \{P([\,], [\,]) \wedge O([\,])\}$	(2)	$\wedge i, (1)$, Folgerung
$\exists z \in L\{P([\,], z) \wedge O(z)\}$	(3)	$\exists i$, (2)
Induktionsschritt		
$[m \in N]$	(4)	Voraussetzung
$[x \in L]$	(5)	Voraussetzung
$\forall n \in N \forall y \in L\{O[y]$ $\quad \rightarrow \exists z \in L(P(n : y, z) \wedge O(z))\}$	(6)	Satz 13.1
$[\phi[x]]$	(7)	Induktionsannahme

$[z \in L \wedge P(x,z) \wedge O(z)]$	(8)	Voraussetzung
$\exists u \in L(Pm : z,u) \wedge O(u))$	(9)	$\forall e, \rightarrow e, \wedge e$, (6), (8), (4)
$[u \in L \wedge P(m : z,u) \wedge O(u)]$	(10)	Voraussetzung
$P(m : x, m : z)$	(11)	(8), $\wedge e, P \wedge O$
$u \in L \wedge P(m : x,u) \wedge O(u)$	(12)	$P \wedge O$, (10), (11), $\wedge e, \wedge i$
$\exists z \in L\{P(m : x,z) \wedge O(z)\}$	(13)	$\exists i$, (12)
$\exists z \in L\{P(m : x,z) \wedge O(z)\}$	(14)	$\exists e$, (13), (10), (9)
$\exists z \in L\{P(m : x,z) \wedge O(z)\}$	(15)	$\exists e$, (14), (8)
$\phi[x] \rightarrow \phi[m : x]$	(16)	$\rightarrow i$, (15), (7)
$\forall m \in N \forall x \in L\{\phi[x] \rightarrow \phi[m : x]\}$	(17)	$\rightarrow i, \forall i$, (4), (5), (16)

Das vervollständigt den Beweis.
Man beachte, dass (an mehreren Stellen) mehrere Schritte zu einem zusammengefaßt wurden. Die Zwischenschritte werden dem Leser überlassen. (Dazu gehören die hergeleiteten Eigenschaften von $P \wedge O$) Die wichtigste Arbeit zur Programmkonstruktion ist damit erledigt. Das benötigte Programm ist implizit in dem obigen Beweis.

5.1.3 Extraktion des Programms

Um das Sortierprogramm zu bekommen, braucht man einen Compiler, der aus dem Beweis das Programm extrahiert. Der Compiler ist eine Extraktion-Funktion F, die aus Herleitungen ein Programm macht. Von Hand ist die Programmextraktion sehr mühsam, interessierte können sich aber ein Beispiel dazu im Buch anschauen (Kap. 13). Nachdem man die Funktion F angewendet hat, erhält man evt. folgendes Programm:

```
     insert n[ ]  =  [n]
   insert n(m:y)  =  n:m:y                 wenn n ≤ m
   insert n(m:y)  =  m:(insert n y)        wenn m < n

   insertsort [ ]  =  [ ]
 insertsort (m:x)  =  insert m(insertsort x)
```

Damit das ganze nicht nur auf Listen von Zahlen beschränkt ist, definiert man Typen für alle totalen Ordnungen auf ein Typ X wie folgt:
$TotOrd(X) = \{f \in List[X] \otimes List[X] \Rightarrow Bool : f \text{ ist eine totale Ordnug auf } X\}$
Damit kann man die ursprüngliche Spezifikation erweitern zu:
$\forall x \in List[X] \forall f \in TotOrd(X) \exists y \in List[X]\{O(y,f) \wedge P(x,y)\}$
wobei mit $O(y,f)$ gemeint ist, dass y geordnet relativ zu f ist. Wir erhalten dann folgende Funktion (mit einem fast gleichen Beweis):

```
   insertsort [ ] f  =  [ ]
 insertsort (m:x) f  =  insert m f(insertsort x f)
```

6 Quellen

Raymond Turner Constructive Foundations For Functional Languages

Kapitel 2, 11.1, 11.2, 13.