Kapitel 2

Formalisierung von Logik, Berechenbarkeit und Typdisziplin

In diesem Kapitel wollen wir die mathematischen Grundlagen vorstellen, die für ein Verständnis formaler Methoden beim Beweis mathematischer Aussagen und bei der Entwicklung korrekter Programme notwendig sind. Wir werden eine Einführung in die Grundkonzepte formaler Beweisführung geben und dies an drei fundamentalen Formalismen illustrieren.

Die *Prädikatenlogik* (siehe Abschnitt 2.2) ist den meisten als abkürzende Schreibweise für logische Zusammenhänge bekannt. Als formales System erlaubt sie, Aussagen zu beweisen, deren Gültigkeit ausschließlich aus der logischen Struktur folgt, ohne daß man dazu die Bedeutung der Einzelaussagen kennen muß. Sie ist ein sehr mächtiges Handwerkzeug, wenn es gelingt, alle notwendigen (und gültigen) Voraussetzungen als logische Teilaussagen zu formulieren. Gerechnet werden kann innerhalb der Prädikatenlogik allerdings nicht.

Der *λ-Kalkül* (siehe Abschnitt 2.3) ist eines der einfachsten Modelle für die Klasse aller berechenbaren Funktionen. Er erlaubt symbolisches Rechnen auf Termen und bietet gleichzeitig ein klar definiertes und einfaches Konzept, um logische Schlußfolgerungen über das Verhalten von Programmen zu ziehen.

Die *einfache Typentheorie* (siehe Abschnitt 2.4) wurde geschaffen, um den vielen Unentscheidbarkeitsfragen zu begegnen, die ein Turing-mächtiger Formalismus wie der λ-Kalkül mit sich bringt. Durch Hinzunahme einer Typdisziplin, welche die Selbstanwendbarkeit von Operationen – die Hauptursache der Unentscheidbarkeiten – deutlich einschränkt, lassen sich viele Programmeigenschaften typisierbarer Algorithmen leichter beweisen.

2.1 Formale Kalküle

Eines der größten Probleme, das normalerweise entsteht, wenn Mathematiker oder Programmierer versuchen, jemand anderen von der Richtigkeit ihrer Behauptungen zu überzeugen, ist die Mehrdeutigkeit der natürlichen Sprache, die sie hierbei verwenden. Dies liegt zum einen daran, daß in der natürlichen Sprache häufig Worte und Formulierungen verwendet werden, denen keine präzise, allgemeingültige Bedeutung zugewiesen werden kann. Das Wort *Baum*, zum Beispiel, hat sehr verschiedene Bedeutungen, je nachdem, in welchem Kontext es verwandt wird. Darüberhinaus gibt es aber auch Sätze, die mehrere Interpretationen zulassen. Was bedeutet zum Beispiel der folgende Satz?

Wenn eine gerade Zahl eingegeben wird, so gibt mein Programm eine Eins aus.

Man sollte meinen, daß dies eine präzise Beschreibung eines einfachen Sachverhalts ist. Aber können wir nun schließen, daß bei ungeraden Zahlen *keine* Eins ausgegeben wird? Das ist legitim, sofern mit *wenn* in Wirklichkeit *nur wenn* gemeint ist, ansonsten aber nicht. Gerade die bisherige Praxis der Validierung von Software zeigt, daß die Verwendung natürlicher Sprache beim logischen Schließen immer wieder die Gefahr von Mißverständnissen und Trugschlüssen in sich birgt. Um diesem Problem zu begegnen, hat man in der Mathematik schon frühzeitig *Kalküle* entwickelt.

Von ihrem ursprünglichen Sinn her sind Kalküle nichts anderes als formale Rechenvorschriften, welche es ermöglichen, bei der Lösung von Problemen nach einem einmal bewiesenen Schema vorzugehen, ohne über dessen Rechtfertigung weiter nachzudenken. Ein tieferes Verständnis ist nicht mehr nötig. Alles, was man zu tun hat, ist, stur einer vorgegebenen Menge von *Regeln* zu folgen, bis man am gewünschten Ziel ist.

Die meisten von Ihnen werden aus der Schule die Kalküle der Differential- und Integralrechnung kennen. Die Summenregel $d/dx(u+v) = du/dx + dv/dx$, die Produktregel $d/dx(u*v) = du/dx*v + u*dv/dx$ und andere Regeln zur Bestimmung von Ableitungsfunktionen wurden ein für allemal bewiesen und ab dann nur noch schematisch angewandt. Oft werden Sie die Rechtfertigung dieser Regeln längst vergessen haben und dennoch in der Lage sein, sie anzuwenden.

Genau besehen sind diese Regeln nichts anderes als syntaktische Vorschriften, wie man symbolische Objekte manipulieren bzw. erzeugen darf, um zum Ziel zu kommen. Auch wenn man diese Symbole im konkreten Fall mit einer Bedeutung verbindet – also bei Anwendung der Produktregel anstelle der Symbole u und v konkrete Funktionen im Blickfeld hat – so ist es bei der Verarbeitung eigentlich gar nicht notwendig, diese Bedeutung zu kennen. Alles, was man zu beachten hat, ist daß die Regeln korrekt angewandt werden, d.h. die Symbole korrekt manipuliert werden. So ist es möglich, *komplexere Aufgaben alleine durch symbolisches Rechnen zu lösen*, wie zum Beispiel bei der Berechnung der folgenden Ableitung.

$$\frac{d}{dx}(x^2+2x+4) = \frac{d}{dx}x^2 + \frac{d}{dx}2x + \frac{d}{dx}4 = \ldots = 2x+2$$

Dieses schematische Anwenden von Regeln zur Manipulation symbolischer Objekte als Ersatz für mathematische Schlußfolgerungen ist genau das Gebiet, für das Computer geschaffen wurden. Alle "Berechnungen", die ein Computerprogramm durchführt, sind im Endeffekt nichts anderes als Manipulationen von Bits (Worten, Symbolen), die mit bestimmten Bedeutungen (Zahlen, Texten etc.) in Verbindung gebracht werden. Die Idee, dieses Anwendungsgebiet auf das mathematische Schließen auszudehnen, um Unterstützung für komplexere Anwendungsgebiete zu schaffen, ist deshalb relativ naheliegend. Programme wie MACSYMA oder MATHEMATICA sind genau aus der Überlegung entstanden, daß für das Lösen bestimmter Aufgaben keine Intelligenz, sondern nur formale Umformungen erforderlich sind.[1]

Die Beschreibung eines formalen Kalküls gliedert sich in eine *formale Sprache*, bestehend aus *Syntax* und *Semantik*, und ein System von *(Inferenz-)Regeln*. Die Syntax der formalen Sprache legt fest, welche äußere Struktur formale Ausdrücke haben müssen, um – unabhängig von ihrer Bedeutung – überhaupt als solche akzeptiert werden zu können. Die Semantik der Sprache ordnet dann den syntaktisch korrekten Ausrücken eine Bedeutung zu. Die Regeln geben an, wie syntaktisch korrekte Grundausdrücke (*Axiome*, *Atome*, o.ä.) zu bilden sind bzw. wie aus bestehenden Ausdrücke neue Ausdrücke erzeugt werden können. Dabei ist beabsichtigt, daß alle Regeln die Semantik der Ausdrücke berücksichtigen. Im Falle logischer Kalküle heißt das, daß alle durch Regeln erzeugbare Ausdrücke *wahr* im Sinne der Semantik sein sollen. Mit Hilfe von formalen Kalkülen wird es also möglich, daß Computer mathematische Aussagen *beweisen*.

Im folgenden wollen wir nun diese Konzepte im Detail besprechen und am Beispiel der (hoffentlich) bekannten Prädikatenlogik erster Stufe, die im nächsten Abschnitt dann vertieft wird, illustrieren.

2.1.1 Syntax formaler Sprachen

Mit der Syntax wird die formale Struktur einer Sprache beschrieben. Sie legt fest, in welcher textlichen Erscheinungsform ein informaler Zusammenhang innerhalb einer formalen Sprache repräsentiert werden kann. Normalerweise verwendet man hierzu eine *Grammatik*, welche die "syntaktisch korrekten" Sätze über einem vorgegebenen *Alphabet* charakterisiert. Häufig wird diese Grammatik in *Backus-Naur Form* oder einem ähnlichen Formalismus notiert, weil dies eine Implementierung leichter macht. Ebenso gut – und für das Verständnis

[1] Die meisten der bisher bekannten Programmpakete dieser Art sind für Spezialfälle konzipiert worden. Wir werden uns im folgenden auf universellere Kalküle konzentrieren, die – zumindest vom Prinzip her – in der Lage sind, das logische Schließen als solches zu simulieren. Natürlich setzen wir dabei wesentlich tiefer an als bei den Spezialfällen und entsprechend dauert es länger bis wir zu gleich hohen Ergebnissen kommen. Dafür sind wir dann aber vielseitiger.

besser – ist aber auch die Beschreibung durch mathematische Definitionsgleichungen in natürlichsprachlichem Text. Wir werden diese Form im folgenden bevorzugt verwenden.

Beispiel 2.1.1 (Syntax der Prädikatenlogik – informal)

Die Sprache der Prädikatenlogik erster Stufe besteht aus Formeln, die aus Termen, Prädikaten, und speziellen logischen Symbolen (*Junktoren* und *Quantoren*) aufgebaut werden.

Terme bezeichnen mathematische *Objekte* wie Zahlen, Funktionen oder Mengen. Zu ihrer Beschreibung benötigt man eine (unendliche) Menge von *Variablen* sowie eine Menge von *Funktionssymbolen*, wobei jedes Funktionssymbol eine *Stelligkeit* $n \geq 0$ besitzt. Ein *Term* ist entweder eine Variable oder eine Funktionsanwendung der Form $f(t_1,\ldots,t_n)$, wobei f ein n-stelliges Funktionssymbol ist und alle t_i Terme sind. Nullstellige Funktionssymbole werden auch *Konstanten(-symbole)* genannt; der Term $c()$ wird in diesem Fall zur Abkürzung einfach als c geschrieben.

Formeln beschreiben mathematische *Aussagen* wie z.B. die Tatsache, daß das Quadrat einer geraden Zahl durch 4 teilbar ist. Hierzu benötigt man eine Menge von *Prädikatssymbolen*, wobei jedes Prädikatssymbol eine *Stelligkeit* $n \geq 0$ besitzt. Eine *atomare Formel* hat die Gestalt $P(t_1,\ldots,t_n)$, wobei P ein n-stelliges Prädikatssymbol ist und alle t_i Terme sind. Eine *Formel* ist entweder eine atomare Formel oder von der Gestalt $\neg A$, $A \wedge B$, $A \vee B$, $A \Rightarrow B$, $\forall x.A$, $\exists x.A$, oder (A), wobei A und B Formeln sind und x eine Variable ist. Jede Formel enthält somit zumindest ein Prädikatssymbol.

Ein paar *Konventionen* ermöglichen Abkürzungen beim Aufschreiben von Formeln. Bei Formeln mit nullstelligen Prädikatssymbolen wird wie bei den nullstelligen Funktionen die Klammer weggelassen. Statt $P()$ schreiben wir kurz P. *Prioritäten* sind ein weiteres wichtiges Hilfsmittel: $\neg$ bindet am stärksten, dann folgen $\wedge$, $\vee$, und $\Rightarrow$ in absteigender Reihenfolge; der *Bindungsbereich* eines Quantors reicht so weit nach rechts wie möglich, d.h. Quantoren binden am schwächsten. Diese Konventionen erlauben es, den syntaktischen Aufbau der Formel $\forall$`x. g(x)`$\wedge\neg$`v(x)` $\Rightarrow$ `v(sqr(x))` eindeutig zu rekonstruieren. Sie entspricht der Formel $\forall$`x. ( (g(x)` $\wedge$ `(`$\neg$`v(x)))` $\Rightarrow$ `v(sqr(x)) )`.

Die Sprache der Prädikatenlogik erster Stufe wird dazu verwandt, mathematische Aussagen zu präzisieren und auf ihren logischen Kern zu reduzieren. Diese Reduktion gilt insbesondere für Funktions- und Prädikatssymbole, die innerhalb dieser Sprache – im Gegensatz zu Junktoren und Quantoren – keine Bedeutung besitzen, selbst wenn die benutzten Namen eine solche suggerieren. Dies genau ist auch der Vorteil der Prädikatenlogik, denn bei den logischen Schlüssen darf es auf die konkrete Bedeutung der Terme und Prädikate nicht ankommen. Was zählt, ist ausschließlich die logische Struktur.[2]

Eine präzisierte Version dieser Beschreibung werden wir in Abschnitt 2.2.1 geben.

Als abkürzende Schreibweise für logische Zusammenhänge ist die Prädikatenlogik ein weit verbreitetes Mittel. Dennoch findet man eine Fülle von verschiedenen Notationen. So wird zuweilen auch $A\&B$ oder A `AND` B statt $A \wedge B$ geschrieben, $A|B$ oder A `OR` B statt $A \vee B$, und $A \rightarrow B$ oder A `IMPLIES` B statt $A \Rightarrow B$. Bei den Quantoren entfällt zuweilen der Punkt (wie in $\exists x\ A$) oder er wird durch Klammern ersetzt (wie in $(\exists x) A$). Der Bindungsbereich von Quantoren reicht manchmal nur bis zum *ersten* $\wedge$, $\vee$, oder $\Rightarrow$.

Dies macht deutlich, wie wichtig es ist, eine eindeutige Syntax für die verwendete formale Sprache zu vereinbaren, damit es beim Lesen von Formeln nicht zu Mißverständnissen kommen kann. Für den Einsatz von Computerunterstützung ist es ohnehin notwendig, die Syntax der Sprache präzise festzulegen.

2.1.2 Semantik formaler Sprachen

Die Semantik ordnet den syntaktisch korrekten Sätzen einer formalen Sprache eine Bedeutung zu. Sie gibt uns die Handhabe, darzustellen, daß wir den Satz $\forall$`x. teilbar(x,2)` $\Rightarrow$ `teilbar(x`^{2}`,4)` für wahr halten,

[2] Deshalb ist es durchaus legitim, den Satz "*das Quadrat einer geraden Zahl ist durch 4 teilbar*" auf verschiedene Arten zu formalisieren, z.B. als $\forall$`x. teilbar(x,2)` $\Rightarrow$ `teilbar(x`^{2}`,4)` oder als $\forall$`x. gerade(x)` $\Rightarrow$ `vierteilbar(x`^{2}`)` oder gar ganz kurz $\forall$`x. g(x)` $\Rightarrow$ `v(sqr(x))`. Die Gültigkeit dieses Satzes hängt nicht von der Wahl der Symbole ab, sondern davon, welche Formeln mit diesen Symbolen bereits als gültig vorausgesetzt werden.

den Satz $\forall$`x. teilbar(x,2)` $\Rightarrow$ `teilbar(x`^{2}`,5)` aber nicht. Die Semantik einer formalen Sprache wird üblicherweise durch eine *Interpretation* der einzelnen Symbole beschrieben.[3] Eine solche Interpretation stellt die Beziehung zwischen einer zu definierenden formalen Sprache (*Quellsprache*) zu einer zweiten (*Zielsprache*) her. In den meisten Fällen ist letztere eine weniger formale Sprache, von der man aber annimmt, daß sie eine klare Bedeutung besitzt.[4]

Fast alle mathematischen Theorien werden in der Cantorschen Mengentheorie interpretiert: jeder Term entspricht einer Menge von Objekten und jede Formel einer Aussage über solche Mengen. Die Mengentheorie selbst wird als Grundlage aller Mathematik betrachtet und nicht weiter in Frage gestellt. Ihre Axiome stützen sich auf informale, aber allgemein anerkannte Intuitionen wie "*Es gibt Mengen*" und "*die Vereinigung zweier Mengen ist wieder eine Menge*".[5]

Beispiel 2.1.2 (Semantik der Prädikatenlogik – informal)

Eine Interpretation der Prädikatenlogik weist jedem logischen Symbol ein mengentheoretisches Objekt zu. Variablen, Funktionen und Prädikate werden alle mit Bezug auf eine feste Menge, das sogenannte *Universum* erklärt. Jede Variable wird einem einem Element dieses Universums zugeordnet, jedes n-stellige Funktionssymbol einer konkreten n-stelligen Funktion auf Elementen des Universums und jedes n-stellige Prädikatssymbol einer n-stelligen Relation über Elementen des Universums. Vorschriften, *welche* Elemente, Funktionen oder Relationen den einzelen Symbolen zugeordnet werden müssen, gibt es nicht.

Eine Interpretation weist somit jedem Term einen Wert (ein Element) zu und macht jede atomare Formel entweder *wahr* oder *falsch*. Der Wahrheitsgehalt zusammengesetzter Formeln wird nach festen Regeln ermittelt, die unabhängig von der konkreten Interpretation gelten:

Die *Negation* $\neg A$ ist genau dann wahr, wenn A falsch ist.
Die *Konjunktion* $A \wedge B$ ist genau dann wahr, wenn sowohl A als auch B wahr sind.
Die *Disjunktion* $A \vee B$ ist genau dann wahr, wenn eine der beiden Formeln A oder B wahr ist.
Die *Implikation* $A \Rightarrow B$ ist genau dann wahr, wenn B wahr ist, wann immer A wahr ist.
Die *Universelle Quantifizierung* $\forall x.A$ ist genau dann wahr, wenn A für jedes mögliche x wahr ist.
Die *Existentielle Quantifizierung* $\exists x.A$ ist genau dann wahr, wenn A für mindestens ein x wahr ist.

Dabei sind A und B beliebige Formeln und x eine Variable.

2.1.3 Klassische und intuitionistische Mathematik

Die oben angegebene Semantik der Prädikatenlogik ist in der Mathematik allgemein akzeptiert. Jedoch läßt die natürlichsprachliche Formulierung einige Fragen offen, die sich vor allem die Art der Beweise auswirkt, welche als legitim anerkannt werden. Kann man zum Beispiel davon ausgehen, daß *jede* Formel entweder wahr oder falsch ist? Folgt die Wahrheit einer Formel $A \vee B$ aus der Tatsache, daß nicht beide falsch sein können? Ist es für die Wahrheit einer Formel der Gestalt $\exists x.A$ essentiell, das konkrete Element x benennen zu können, welches die Formel A wahr macht? Formal ausgedrückt ist das die Frage nach der Allgemeingültigkeit der folgenden logischen Gesetze:

[3] Dies ist die sogenannte *denotationale* Semantik. Es sei aber erwähnt, daß auch weitere Formen existieren wie z.B. die *operationale* Semantik des λ-Kalküls (siehe Abschnitt 2.3.2), welche beschreibt, wie λ-Terme auszurechnen sind.

[4] Um eine derartige Annahme wird man nicht herumkommen, auch wenn man sich bemüht, die Semantik der Zielsprache zu präzisieren. Im Endeffekt wird man sich immer auf eine Zielsprache abstützen müssen, die ihre Bedeutung der natürlichen Sprache entnimmt.

[5] Daß diese intuitive Abstützung nicht ohne Gefahren ist, zeigt der Zusammenbruch des gesamte Gedankengebäudes der Mathematik zu Beginn dieses Jahrhunderts, als das Russelsche Paradoxon das bis dahin vorherrschende Mengenkonzept zerstörte. Die Arbeit vieler Mathematiker – wie z.B. die von G. Frege [Frege, 1879, Frege, 1892], der die Mathematik zu formalisieren versuchte – wurde hierdurch zunichte gemacht. Bei der heutigen Formulierung der Mengentheorie ist man sich aber sehr sicher, daß sie keine Widersprüche zuläßt.

Es gilt $A \vee \neg A$ *(Gesetz vom ausgeschlossenen Dritten)*
Aus $\neg(\neg A)$ folgt A
Aus $\neg(\neg A \wedge \neg B)$ folgt $A \vee B$
Aus $\neg(\forall x.\neg A)$ folgt $\exists x.A$

Während diese Gesetze unstreitig für die *meisten* logischen Aussagen gültig sind, die man für A und B einsetzen kann, gibt es doch Fälle, in denen die Anwendung dieser Gesetze für viele "mathematisch unverbildete" Menschen zumindest problematisch ist.

Beispiel 2.1.3

In der Analysis wurde der Satz *"Es gibt zwei irrationale Zahlen x und y so, daß x^y rational ist"* zunächst wie folgt bewiesen

Die Zahl $\sqrt{2}^{\sqrt{2}}$ ist entweder rational oder irrational. Ist sie rational, so können wir $x{=}y{=}\sqrt{2}$ wählen, denn $\sqrt{2}$ ist bekanntermaßen irrational. Anderenfalls wählen wir $x{=}\sqrt{2}^{\sqrt{2}}$ und $y{=}\sqrt{2}$, die nun beide irrationale Zahlen sind, und bekommen $x^y{=}2$, also eine rationale Zahl.

Dieser Beweis macht massiv Gebrauch vom Gesetz des ausgeschlossenen Dritten, läßt aber die meisten Leser unzufrieden. Selbst wenn wir die Beweisführung akzeptieren, wissen wir immer noch nicht, welches die beiden Zahlen sind, deren Existenz bewiesen wurde.[6]

Intuitionismus (*intuitionistische Mathematik*) ist eine mathematische Denkrichtung, welche diese Art von Beweisführung und die Allgemeingültigkeit des Gesetzes vom ausgeschlossenen Dritten ablehnt. Anders als die sogenannte *klassische* Mathematik, die sich mittlerweile weitgehend in Schulen und Universitäten durchgesetzt hat, geht sie davon aus, daß mathematische Aussagen *konstruktiv* zu verstehen sind. In dieser Sicht besagt eine Existenzaussage der Form $\exists x.A$, daß man im Endeffekt ein konkretes Element angeben kann, welches die Aussage A erfüllt, und eine Disjunktion $A \vee B$, daß man sagen kann, welcher der beiden Fälle gültig ist.

Diese Forderung macht mathematische Beweise natürlich schwieriger als solche, bei denen es ausreicht, das Gegenteil einer Aussage zu widerlegen. In der Mathematik des Endlichen – solange man also ausschließlich über natürliche und rationale Zahlen, Listen, *endliche* Mengen (von Zahlen o.ä.) usw. spricht – ergeben sich hierduch auch keine anderen Ergebnisse,[7] so daß die Forderung eher als eine lästige Einschränkung erscheint. Sobald aber unendliche Objekte wie *unendliche* Mengen (wie z.B. die Menge der natürlichen Zahlen), reelle Zahlen, Funktionen oder Programme in den Aussagen vorkommen, werden auch die Ergebnisse unterschiedlich und der Streit über die Richtigkeit mancher Aussage ist heute noch nicht geklärt.

Für das Schließen über Programme wird dieses Problem besonders deutlich. Bei der Verifikation eines gegebenen Programmes mag es noch angehen, wegen der einfacheren Beweise klassische Logik zu verwenden, auch wenn dies in vielen Teilen bereits recht problematisch wird. Es macht aber wenig Sinn, die Existenz eines Programms mit einer gewünschten Eigenschaft dadurch zu beweisen, daß man die Behauptung widerlegt, alle Programme würden diese Eigenschaft nicht besitzen.

Daher werden wir uns im folgenden mit beiden Denkrichtungen, der klassischen und der intuitionistischen Logik beschäftigen müssen. Zum Glück ist der Unterschied *formal* sehr gering. Klassische Logik ergibt sich aus der intuitionistischen durch Hinzufügen eines einzigen Gesetzes, da alle logischen Gesetze, die einen Unterschied zwischen den beiden ausmachen, unmittelbare Folgerungen des Gesetzes vom ausgeschlossenen Dritten sind.

[6] Es gibt übrigens auch einen direkten Beweis mit $x = \sqrt{2}$, $y = 2 * ld3$ und $x^y = 3$.

[7] Im Endlichen ist es rein hypothetisch immer möglich, *alle* möglichen Fälle hinzuschreiben und einzeln zu untersuchen. Wenn ich also gezeigt habe, daß nicht alle Elemente x eine Eigenschaft A nicht haben (d.h. $\neg(\forall x.\neg A)$), dann weiß ich, daß ich bei der Untersuchung aller Möglichkeiten dieses Element finden werde. Sobald es aber unendlich viele Möglichkeiten gibt, besteht diese Chance im allgemeinenn nicht mehr.

2.1.4 Objekt- und Metasprache

In den vorhergehenden Teilabschnitten haben wir bereits öfter Aussagen über Formeln gemacht. Wir haben gesagt, daß $A \vee B$ eine Formel ist, wenn A und B Formeln sind, und über Gesetze wie das vom ausgeschlossenen Dritten gesprochen, was wir kurz mit "*Es gilt $A \vee \neg A$*" ausgedrückt haben. Dabei war klar, daß die Symbole A und B nicht selbst Formeln (also atomare Formeln, die nur das nullstellige Prädikatssymbol A bzw. B enthalten) sind, sondern *Platzhalter* für beliebige Formeln. Diese Platzhalter erlauben uns, Aussagen *über* logische Formeln – wie z.B. logische Gesetze – kurz und prägnant zu beschreiben, anstatt sie durch lange und kaum verständliche Sätze auszudrücken. Der Satz "*Aus der Negation der Konjunktion der Negation zweier Formeln folgt die Disjunktion der beiden Formeln*" besagt dasselbe wie "*aus $\neg(\neg A \wedge \neg B)$ folgt $A \vee B$*", ist aber erheblich schwerer zu verstehen.

Die Verwendung von Platzhaltern für Formeln, Terme und Variablen macht aber auch klar, daß man im Zusammenhang mit formalen Kalkülen zwei Ebenen, nämlich die *Objektsprache* und die *Metasprache* des Kalküls, unterscheiden muß.

- Die *Objektsprache* eines Kalküls ist die formale Sprache, in welcher die Objekte formalisiert werden, über die wir formale Schlüsse ziehen wollen. Bei der Prädikatenlogik ist dies also eine Sprache, die aus Termen und Formeln besteht.
- Die *Metasprache*[8] des Kalküls ist diejenige Sprache, die wir benutzen, wann immer wir Aussagen *über* den Kalkül und seine Objektsprache machen. Sie wird benötigt, um den Aufbau der Objektsprache und vor allem auch die Inferenzregeln zu beschreiben. Normalerweise wird hierfür eine natürliche Sprache verwendet. Wir benutzen einfaches Deutsch um über Kalküle zu reden. Da dies aber aus den obengenannten Gründen manchmal unverständlich wird, verwenden wir zusätzlich eine mathematische Notation, in der Symbole der Objektsprache (wie $\forall$ und $\wedge$) und Platzhalter für objektsprachliche Ausdrücke – sogenannte *syntaktische Metavariablen* – vorkommen.

Um Objekt- und Metasprache voneinander zu trennen, werden wir hierzu verschiedene Zeichensätze verwenden. Alle Ausdrücke der Objektsprache werden im `typewriter`-font geschrieben – so, wie sie auch im Computer Einsatz finden werden. Dabei ist jedoch zu bedenken, daß 8-bit fonts auch mathematische Symbole wie $\forall$ und $\wedge$ enthalten können. Zugunsten der besseren Lesbarkeit werden wir Schlüsselworte der Sprache durch **Fettdruck** in einem ähnlichen Zeichensatz hervorheben.

Syntaktische Metavariablen werden im mathematischen Zeichensatz *kursiv* gedruckt. Dabei werden wir bestimmte Symbole bevorzugt für bestimmte Arten von Objekten verwenden.

- Namen für Variablen der Objektsprache sind normalerweise $x, y, z, x_1, x_2, \ldots$
- Namen für Funktionssymbole sind $f, g, h, \ldots$
- Namen für Terme sind $t, u, v, r, s, \ldots$
- Namen für Prädikatssymbole sind $P, Q, R, \ldots$
- Namen für Formeln sind $A, B, C, \ldots$

Diese Konventionen[9] ersparen uns, syntaktische Metavariablen immer wieder neu deklarieren zu müssen.

2.1.5 Definitorische Erweiterung

Bei der Entwicklung eines formalen Kalküls versucht man normalerweise, mit einem möglichst kleinen Grundgerüst auszukommen. Dies erleichtert die Definition von Syntax und Semantik und macht vor allem den

[8] Die griechische Vorsilbe *meta* steht für "über", "jenseits von" und ähnliches.

[9] Diese Liste wird später implizit um weitere Konventionen erweitert. Die Symbole n, m, i, j, k, l bezeichnen Zahlen, griechische Buchstaben Γ, Δ, Θ stehen für Mengen und Listen von Formeln, T, S für Bereiche (Typen), usw.

Nachweis leichter, daß die Ableitungsregeln tatsächlich der Semantik der formalen Sprache entsprechen und nicht etwa Widersprüche enthalten. Bei der Formalisierung natürlichsprachlicher Zusammenhänge möchte man dagegen eine möglichst umfangreiche Sprache zur Verfügung haben, damit eine Formulierung nicht derart umfangreich wird, daß sie nicht mehr zu verstehen ist.

Ein einfaches Mittel, sowohl der theoretischen Notwendigkeit für einen kleinen Kalkül als auch der praktischen Anforderung nach einer großen Menge vordefinierter Konzepte gerecht zu werden ist die sogenannte *konservative Erweiterung* einer formalen Sprache durch *definitorische Abkürzungen*. Dieses in der Mathematik gängige Konzept, bei dem ein komplexer Zusammenhang durch einen neuen abkürzenden Begriff belegt wird, läßt sich größtenteils auch auf formale Theorien übertragen. Es wird einfach eine Definition aufgestellt, die einen Ausdruck der bestehenden formalen Sprache durch einen neuen abkürzt, wobei syntaktische Metavariablen eingesetzt werden dürfen.

Beispiel 2.1.4 (Definitorische Erweiterungen der Prädikatenlogik)

In der Syntax der Prädikatenlogik (siehe Beispiel 2.1.1 auf Seite 11) vermißt man die *Äquivalenz* "*A gilt, genau dann wenn B gilt*", die üblicherweise mit $A \Leftrightarrow B$ bezeichnet wird. Sie läßt sich aber auf zwei Implikationen zurückführen, nämlich "*A gilt, wenn B gilt*" und "*B gilt, wenn A gilt*". Damit können wir $A \Leftrightarrow B$ als Abkürzung für $(A \Rightarrow B) \wedge (B \Rightarrow A)$ betrachten. Wir benutzen hierfür folgende Schreibweise.

$$A \Leftrightarrow B \quad \equiv \quad (A \Rightarrow B) \wedge (B \Rightarrow A)$$

In ähnlicher Weise könnte man die Prädikatenlogik um eine exklusive Disjunktion $A \dot{\vee} B$ erweitern, die ausdrückt, daß genau eine der beiden Formeln A und B wahr ist.

Durch die konservative Erweiterung einer formalen Sprache gewinnen wir also Flexibilität – zumal für die textliche Form der Abkürzungen keine Einschränkungen gelten außer, daß es in einem Zeichensatz beschreibbar ist und alle syntaktischen Metavariablen der rechten Seite auch links vorkommen müssen. Die guten Eigenschaften des bisherigen Kalküls, an dem wir ja überhaupt nichts ändern, bleiben dagegen erhalten.

2.1.6 Inferenzsysteme

Durch die Definition der Semantik ist die Bedeutung von objektsprachlichen Ausdrücken eindeutig festgelegt. Rein hypothetisch wäre es möglich, diese Bedeutung dadurch zu bestimmen, daß man die Interpretation präzisiert und dann den "Wert" eines Ausdruck ausrechnet. Bei dieser Vorgehensweise wären formale Kalküle jedoch nur wenig mehr als eine abkürzende Schreibweise, da das Bewerten einer Aussage im wesentlichen doch wieder von Hand geschehen muß. Sinnvoller ist es, syntaktische Aussagen *direkt* zu manipulieren, ohne ihren Wert zu bestimmen, und dabei die Möglichkeiten für eine syntaktische Manipulation so einzugrenzen, daß dem Wert der Ausdrücke Rechnung getragen wird, selbst wenn sich das textliche Erscheinungsbild ändert.

Es gibt zwei Möglichkeiten zur syntaktischen Manipulation, *Konversion* und *Inferenz*. Bei der *Konversion* werden Ausdrücke der Objektsprache in semantisch äquivalente umgeformt.[10] Im Falle der Prädikatenlogik werden also logische Formeln in andere Formeln mit gleichem Wahrheitswert umgewandelt, also zum Beispiel $A \wedge (A \vee B)$ in A. Im Gegensatz dazu geht es bei der *Inferenz* darum, die Gültigkeit einer mathematischen Aussage zu *beweisen* ohne dabei über deren Bedeutung nachzudenken. Inferenzregeln haben daher die Gestalt "*aus A_1 und ... und A_n darf ich C schließen*", wie zum Beispiel "*aus A und $A \Rightarrow B$ folgt B*" ("*modus ponens*").

Auch wenn Konversionsregeln eher über *Werte* und Inferenzregeln eher über *Gültigkeit* reden, können Konversionsregeln meist auch als Inferenzregeln geschrieben werden. Jedoch kann man Konversionsregeln in beide Richtungen lesen, während Inferenzregeln nur in einer festen Richtung anwendbar sind. Für das logische Schließen sind Inferenzregeln vielseitiger, denn sie erlauben auch, Aussagen abzuschwächen.

Inferenzsysteme, oft auch einfach *Kalkül* genannt, bestehen aus einer Menge von Regeln zur Manipulation objektsprachlicher Ausdrücke. Eine Regel der Art "*aus A_1 und ... und A_n folgt C*" wird häufig in schematischer Form aufgeschrieben:

$$\frac{A_1, \ldots, A_n}{C}$$

[10] Der Kalkül der Differentialrechnung ist ein Konversionskalkül, bei dem Gleichheiten die Umformungsregeln bestimmen.

Dabei sind die Aussagen $A_1, \ldots, A_n$ die *Prämissen* dieser Regel und C die *Konklusion*. Die Anzahl n der Prämissen darf auch Null sein. In diesem Falle nennt man die Regel auch ein *Axiom*, da die Konklusion ohne jede Voraussetzung Gültigkeit hat. Axiome werden oft auch separat von den "eigentlichen" Regeln betrachtet, da sie mit allgemeingültigen Grundaussagen des Kalküls identifiziert werden können.

Genaugenommen bezeichnet die Schreibweise $\frac{A_1,\ldots,A_n}{C}$ ein Regel*schema*, da die Prämissen und die Konklusion, wie bereits erwähnt, syntaktische Metavariablen enthalten. Die Anwendung eines Regelschemas auf konkrete Prämissen wird mit $\underline{\Gamma \vdash_{rs} C}$ bezeichnet. Dabei ist Γ die Menge der konkreten Prämissen und C die konkrete Konklusion und rs die Bezeichnung für das jeweilige Regelschema. $\Gamma \vdash_{rs} C$ gilt, wenn die syntaktischen Metavariablen der Regel rs so durch Formeln ersetzt werden können, daß die Prämissen von rs in der Menge Γ enthalten sind und C die Konklusion ist.

Theoreme sind mathematische Aussagen, die sich durch Anwendung von endlich vielen Regeln *ableiten* lassen. Dabei ist Ableitbarkeit wie folgt definiert.

Definition 2.1.5 (Ableitbarkeit)

Es sei RG die Menge der Regeln und AX die Menge der Axiome eines Kalküls. Γ sei eine (endliche) Menge von Formeln der Logiksprache und C eine Formel der formalen Sprache. Dann heißt C aus Γ ableitbar, wenn es eine Folge $rs_1 \ldots rs_n$ von Regeln aus RG und eine eine Folge von Formeln $C_1 \ldots C_n$ mit $C_n = C$ gibt, derart, daß gilt

$$\Gamma_1 \vdash_{rs_1} C_1, \ \ldots, \ \Gamma_j \vdash_{rs_j} C_j, \ \ldots, \ \Gamma_n \vdash_{rs_n} C_n$$

wobei Γ_j eine Menge von Formeln bezeichnet, welche alle Axiome des Kalküls, alle Formeln aus Γ und alle bisher abgeleiteten Formeln C_l mit $1 \leq l < j$ enthält. (Also $\Gamma_j = AX \cup \Gamma \cup \{C_l | 1 \leq l < j\}$, $\Gamma_1 = AX \cup \Gamma$)

Falls Γ ausschließlich Axiome enthält, dann bezeichnen wir C als ableitbar oder als Theorem.

Bei einer formalen Darstellung läßt sich eine Ableitung als ein Baum beschreiben, an dessen Wurzel das Theorem steht und dessen Knoten durch Regeln markiert sind. Die Blätter dieses Baumes müssen entweder aus Axiomen – oder bei einer stärkeren Strukturierung – aus anderen Theoremen bestehen.

Mit Hilfe von formalen Kalkülen wird es also möglich, daß Computer mathematische Aussagen *beweisen*. Wir müssen aber dennoch eine scharfe Trennung zwischen *Wahrheit* und *Beweisbarkeit* ziehen. Im Idealfall sollte jedes bewiesene Theorem wahr im Sinne der Semantik sein und umgekehrt jede wahre Aussage beweisbar sein. Das ist jedoch nicht immer möglich. Deshalb müssen wir die folgenden zwei Begriffe prägen.

Definition 2.1.6 (Korrektheit und Vollständigkeit)

1. *Eine Inferenzregel heißt korrekt, wenn aus der Gültigkeit aller Prämissen immer auch die Gültigkeit der Konklusion (semantisch) folgt. Ein Kalkül heißt korrekt, wenn jede seiner Inferenzregeln korrekt ist.*

2. *Ein Kalkül heißt vollständig, wenn jede semantisch wahre Aussage ein Theorem, also mit den Inferenzregeln des Kalküls beweisbar ist.*

In korrekten Kalkülen ist also jedes Theorem des Kalküls semantisch wahr. Dies ist eine Grundvoraussetzung, die man an Kalküle stellen muß, denn Kalküle mit inkorrekten Regeln haben für die Beweisführung einen geringen praktischen Wert.[11] Vollständigkeit dagegen ist bei reichhaltigen formalen Sprachen praktisch nicht mehr zu erreichen. Wenn eine Theorie die gesamte Arithmetik umfaßt, dann kann es – so der Gödelsche Unvollständigkeitssatz – hierfür keine vollständigen Kalküle mehr geben.

[11] Da Korrektheit relativ zu einer Semantik definiert ist, muß der Begriff zuweilen relativiert werden, wenn die Semantik nicht eindeutig fixiert ist. Im Falle der Prädikatenlogik z.B. hängt die Semantik von der konkreten Interpretation ab. Ein korrekter Kalkül sollte für jede mögliche Interpretation korrekt sein. Ein Kalkül heißt *konsistent* (*widerspruchsfrei*), wenn die Regeln einander nicht widersprechen, also noch mindestens eine Interpretation zulassen. Konsistenz wird immer dann interessant, wenn sich die Semantik einer formalen Sprache noch nicht genau angeben läßt. In diesem Falle bestimmt der Kalkül, welche Semantik überhaupt noch möglich ist.

Axiomenschemata:

(A1)	$A \Rightarrow A$	(A11)	$(A\wedge B \ \vee\ C) \Rightarrow (A\vee C)\wedge(B\vee C)$
(A2)	$A \Rightarrow (B \Rightarrow A)$	(A12)	$(A\vee C)\wedge(B\vee C) \Rightarrow (A\wedge B \ \vee\ C)$
(A3)	$(A\Rightarrow B) \Rightarrow ((B\Rightarrow C) \Rightarrow (A\Rightarrow C))$	(A13)	$(A\vee B)\wedge C \Rightarrow (A\wedge C \ \vee\ B\wedge C)$
(A4)	$(A\Rightarrow (B\Rightarrow C)) \Rightarrow ((A\Rightarrow B) \Rightarrow (A\Rightarrow C))$	(A14)	$(A\wedge C \ \vee\ B\wedge C) \Rightarrow (A\vee B)\wedge C$
(A5)	$A \Rightarrow A\vee B$	(A15)	$(A\Rightarrow B) \Rightarrow (\neg B\Rightarrow\neg A)$
(A6)	$A \Rightarrow B\vee A$	(A16)	$A\wedge\neg A \Rightarrow B$
(A7)	$(A\Rightarrow C) \Rightarrow ((B\Rightarrow C) \Rightarrow (A\vee B \Rightarrow C))$	(A17)	$(A \wedge (A\Rightarrow B)) \Rightarrow B$
(A8)	$A\wedge B \Rightarrow A,$	(A18)	$(A\wedge C \Rightarrow B) \Rightarrow (C \Rightarrow (A\Rightarrow B))$
(A9)	$A\wedge B \Rightarrow B$	(A19)	$(A \Rightarrow (A\wedge\neg A)) \Rightarrow \neg A$
(A10)	$(C\Rightarrow A) \Rightarrow ((C\Rightarrow B) \Rightarrow (C \Rightarrow A\wedge B))$		

Ableitungsregel:

(mp) $\dfrac{A \ ,\ A\Rightarrow B}{B}$

Abbildung 2.1: Frege–Hilbert–Kalkül für die Aussagenlogik

Es sollte an dieser Stelle auch erwähnt werden, daß ein Kalkül nur die Regeln festlegt, mit denen Beweise geführt werden dürfen, um zu garantieren, daß alle formalen Beweise auch korrekten logischen Schlüssen entsprechen. Ein Kalkül ist dagegen keine Methode, um Beweise zu *finden*.[12]

Es gibt zwei Ausrichtungen bei Kalkülen. Die meisten von ihnen arbeiten *synthetisch* (bottom-up), d.h. sie beschreiben, wie man von einer vorgegebenen Menge von Axiomen auf eine neue Aussage schließen kann. Dies erscheint wenig zielgerichtet, aber dafür sind die Regeln oft einfacher formuliert, da man präzsiser beschreiben kann, wie man vorzugehen hat. Bei einem *analytischen* Kalkül beschreiben die Regeln den Weg von einer Zielaussage zu den Axiomen. Die Beweise sind zielgerichtet (top-down), aber die Regeln benötigen mehr Kontrolle, da festzulegen ist, welche Informationen noch gebraucht werden und welche weggeworfen werden dürfen. Generell ist zu berücksichtigen, daß es *den* Kalkül für eine formale Sprache nicht gibt. Regeln und Axiome können so gewählt werden, daß die gestellte Aufgabe bestmöglich gelöst werden kann. Eine andere Aufgabenstellung kann einen anderen Kalkül sinnvoll erscheinen lassen. Wir werden im folgenden die wichtigsten Kalkülarten kurz vorstellen.

2.1.6.1 Frege–Hilbert–Kalküle

Die ersten logischen Kalküle wurden von Gottlob Frege gleichzeitig mit der Entwicklung der Sprache der Prädikatenlogik entworfen. Diese wurden später von David Hilbert (siehe zum Beispiel [Hilbert & Bernays, 1934, Hilbert & Bernays, 1939]) weiterentwickelt, weshalb man sie heute als *Frege–Hilbert–Kalküle* oder auch als *Hilberttypkalküle* bezeichnet. Sie bestehen typischerweise aus relativ vielen Axiomen und wenigen "echten" Kalkülregeln. Sie sind synthetischer Natur, d.h. es werden Formeln ausgehend von Axiomen mittels Schlußregeln abgeleitet. Im Prinzip sind sie sehr mächtig, da sie kurze Beweise von Theoremen ermöglichen. Jedoch ist wegen der synthetischen Natur die *Suche* nach einem Beweis für ein gegebenes Problem recht aufwendig.

[12] Die rechnergestützte *Suche* nach Beweisen ist ein Thema der *Inferenzmethoden*, deren Schwerpunkte in gewissem Sinne komplementär zu denen dieser Veranstaltung sind. Die dort besprochenen Verfahren wie Konnektionsmethode und Resolution sind allerdings bisher nur auf die Prädikatenlogik erster Stufe (und Modallogiken) anwendbar. Zu komplexeren Kalkülen wie der Typentheorie ist bisher kein allgemeines Beweissuchverfahren bekannt und es ist auch unwahrscheinlich, daß es ein solches je geben wird. Wir werden uns daher damit begnügen müssen, zu jedem Teilkalkül/-aspekt methodische Hinweise zu geben, die wir später dann zum Teil als Beweistaktiken automatisieren.

Als Beispiel für einen Frege–Hilbert–Kalkül haben wir in Abbildung 2.1 einen Ableitungskalkül für die *Aussagenlogik* – also die Prädikatenlogik ohne Quantoren – zusammengestellt. Dabei haben wir bei den Axiomen darauf verzichtet, die leere Liste der Prämissen mit aufzuführen. Axiom (A1) schreiben wir daher kurz $A\Rightarrow A$ anstatt $\dfrac{}{A\Rightarrow A}$.

Der Kalkül in Abbildung 2.1 ist korrekt und vollständig für die *intuitionistische Aussagenlogik*. Für die klassische Aussagenlogik muß man die Axiome (A17) bis (A19) entfernen und stattdessen das Gesetz vom ausgeschlossenen Dritten $A \ \vee\ \neg A$ als neues Axiom (A17) hinzufügen. Es sei nochmals erwähnt, daß für die syntaktischen Metavariablen jede beliebige Formel eingesetzt werden darf.

2.1.6.2 Natürliche Deduktion

Mit den Frege–Hilbert–Kalkülen konnte das logische Schließen soweit präzisiert werden, daß Zweifel an der Gültigkeit von Aussagen zumindest im Prinzip ausgeschlossen werden können. Sie sollten es daher ermöglichen, mathematische Beweise auf eine völlig formale Gestalt zu bringen. Gerhard Gentzen hat jedoch darauf hingewiesen, daß Mathematiker eine Form von Beweisen verwenden, die sich von den Frege–Hilbert–Ableitungen wesentlich unterscheiden, so daß sich mathematische Beweise nicht leicht in solcher Weise formalisieren lassen. Von Gentzen stammt daher ein Kalkül [Gentzen, 1935], der dieses natürliche Schließen wesentlich unmittelbarer widerspiegelt. Bei diesem *Kalkül des natürlichen Schließens* (von Gentzen mit *NJ* bzw. *NK*[13] bezeichnet) handelt es sich ebenfalls um einen synthetischen Kalkül, der sich daher nicht besonders gut für die das Finden von Beweisen eignet. Die Nähe zu der "natürlichen" Form des mathematischen Schließens läßt ihn jedoch zu einem guten Bindeglied zwischen automatisch gestützten und menschlichen Schließen werden.

Die Grundidee dieses Kalküls ist die folgende. Wenn ein Mathematiker eine Formel der Gestalt $A\wedge B$ beweisen will, dann wird er hierzu separat A und B beweisen. Will er eine Formel der Gestalt $A\Rightarrow B$ beweisen, dann nimmt er zunächst an, A gelte, und versucht dann, aus dieser Annahme die Gültigkeit von B abzuleiten. Solche Annahmen spielen eine wichtige Rolle, denn sie enthalten Informationen, die bei der weiteren Beweisführung noch verwendet werden können. Hat eine Annahme zum Beispiel die Gestalt $B\wedge C$, so können wir sowohl die Formel B als auch C im Beweis benutzen. Gentzen hat den Kalkül des natürlichen Schließens sehr symmetrisch ausgelegt. Zu jedem logischen Symbol gibt es zwei Arten von Regeln, eine *Einführungsregel* (–I) und eine *Eliminationsregel* (–E):

- Die Einführungsregel beantwortet die Frage "*unter welchen Voraussetzungen kann ich auf die Gültigkeit einer Formel schließen?*". Die Einführungsregel $\wedge$–I für die Konjunktion $\wedge$ besagt zum Beispiel, daß man auf die Gültigkeit von $A\wedge B$ schließen kann, wenn man A und B als Prämissen hat: $\dfrac{A \quad B}{A\wedge B}$
- Die Eliminationsregel beantwortet die Frage "*welche Informationen folgen aus einer gegebenen Formel?*". Die beiden Eliminationsregeln $\wedge$–E besagen zum Beispiel, daß man aus $A\wedge B$ sowohl A als auch B folgern kann, was zu den zwei Regeln $\dfrac{A\wedge B}{A}$ und $\dfrac{A\wedge B}{B}$ führt.

Jedes logische Zeichen ist durch die Beantwortung dieser beiden Fragen eindeutig charakterisiert. Im Normalfall sind Einführungs- und Eliminationsregel invers zueinander, d.h. wenn wir eine Konjunktion $A\wedge B$ eliminieren und anschließend wieder einführen haben wir weder Informationen gewonnen noch verloren.

Anders als in Frege–Hilbert–Kalkülen behandeln alle Regeln jeweils nur *ein* logisches Zeichen. Dadurch wird der Kalkül sehr klar und einfach. Die Details des Kalküls *NJ* für die Aussagenlogik sind in Abbildung 2.2 zusammengestellt. Das Zeichen Λ steht hierbei für den logischen *Widerspruch* ("falsch").[14] Auch dieser Kalkül ist korrekt und vollständig für die intuitionistische Aussagenlogik. Bemerkenswerterweise benötigt er nicht ein einziges Axiom.[15] Der Grund hierfür ist die Tatsache, daß manche Regeln es ermöglichen, einmal getroffene Annahmen wieder zu entfernen.

[13] Die Abkürzungen stehen für die Natürliche Deduktion für intuitionistische (*J*) bzw. klassische (*K*) Logik.

[14] Es wird benötigt, um in dem von Gentzen angestrebten klaren Stil die Bedeutung der Negation zu erklären.

[15] Für die klassische Aussagenlogik müsste man das Axiom $A \ \vee\ \neg A$ ergänzen. Ansonsten sind die Kalküle identisch, was ein weiterer Vorteil der Gentzen'schen Kalküle des Natürlichen Schließens gegenüber Frege–Hilbert–Kalkülen ist.

		Λ-E	$\dfrac{\Lambda}{A}$
$\neg$-I	$\dfrac{\begin{matrix}[A]\\ \Lambda\end{matrix}}{\neg A}$	$\neg$-E	$\dfrac{\neg A \quad A}{\Lambda}$
$\wedge$-I	$\dfrac{A \quad B}{A\wedge B}$	$\wedge$-E	$\dfrac{A\wedge B}{A} \qquad \dfrac{A\wedge B}{B}$
$\vee$-I	$\dfrac{A}{A\vee B} \qquad \dfrac{B}{A\vee B}$	$\vee$-E	$\dfrac{A\vee B \quad \begin{matrix}[A]\\ C\end{matrix} \quad \begin{matrix}[B]\\ C\end{matrix}}{C}$
$\Rightarrow$-I	$\dfrac{\begin{matrix}[A]\\ B\end{matrix}}{A\Rightarrow B}$	$\Rightarrow$-E	$\dfrac{A \quad A\Rightarrow B}{B}$

Abbildung 2.2: Kalkül des Natürlichen Schließens für die Aussagenlogik

Wir wollen dies am Beispiel der Einführungsregel für die Implikation erläutern. Diese Regel, die dem Modus Ponens gleichkommt, drückt aus, daß die Implikation $A\Rightarrow B$ genau dem Gedanken "*aus A folgt B*" entspricht. Sie verlangt als Prämisse nur, daß B *unter der Annahme*, daß A wahr ist, erfüllt sein muß (was wir dadurch kennzeichnen, daß wir $[A]$ in eckigen Klammern über B stellen). Hieraus darf man $A\Rightarrow B$ schließen, ohne daß es hierfür noch der Annahme A bedarf. Von den bisher getroffenen Annahmen kann A also entfernt werden.

Zur Illustration geben wir eine Ableitung der Formel `((A ⇒ B) ∧ (B ⇒ C)) ⇒ (A ⇒ C)` (dies entspricht im Prinzip dem Axiom (A3) im Frege–Hilbert–Kalkül) in diesem System an. Zunächst zeigen wir, wie der zugehörige Beweis eines Mathematikers in etwa aussehen würde.

1. Wir nehmen an `(A ⇒ B) ∧ (B ⇒ C)` sei erfüllt
2. Wir nehmen weiter an, daß `A` gilt.
3. Aus der ersten Annahme folgt `(A ⇒ B)`
4. und mit der zweiten dann auch `B`.
5. Aus der ersten Annahme folgt auch, daß `(B ⇒ C)` gilt
6. und mit der vierten dann auch `C`.
7. Es ergibt sich also, daß `C` unter der Annahme `A` gilt. Also folgt `A ⇒ C`
8. Insgesamt ergibt sich also, daß `A ⇒ C` unter der Annahme `(A ⇒ B) ∧ (B ⇒ C)` gilt.
   Damit folgt die Behauptung: es gilt `((A ⇒ B) ∧ (B ⇒ C)) ⇒ (A ⇒ C)`

Die entsprechende Ableitung im Kalkül des natürlichen Schließens ist die folgende:

(1)	`(A ⇒ B) ∧ (B ⇒ C)`	Annahme
(2)	`A`	Annahme
(3)	`(A ⇒ B)`	$\wedge$-E angewendet auf (1)
(4)	`B`	$\Rightarrow$-E angewendet auf (2) und (3)
(5)	`(B ⇒ C)`	$\wedge$-E angewendet auf (1)
(6)	`C`	$\Rightarrow$-E angewendet auf (4) und (5)
(7)	`(A ⇒ C)`	$\Rightarrow$-I angewendet auf (2) und (6) — *Annahme (2) entfällt*
(8)	`(A ⇒ B) ∧ (B ⇒ C) ⇒ (A ⇒ C)`	$\Rightarrow$-I angewendet auf (1) und (7) —*Annahme (1) entfällt*

Es ist offensichtlich, daß jeder der acht Schritte des natürlichen Beweises in unmittelbarer Weise von dem entsprechenden Schritt des formalen Beweises wiedergegeben wird und umgekehrt. Man kann diesen

noch weiter formalisieren und verkürzen, indem man eine Baumschreibweise verwendet, also einfach alle Ableitungsschritte übereinander schreibt. Der Beweis sieht dann wie folgt aus.

$$\dfrac{\dfrac{\dfrac{\dfrac{[A] \quad \dfrac{[(A\Rightarrow B)\wedge(B\Rightarrow C)]}{(A\Rightarrow B)}\,\wedge\text{-E}}{B}\,\Rightarrow\text{-E} \qquad \dfrac{[(A\Rightarrow B)\wedge(B\Rightarrow C)]}{(B\Rightarrow C)}\,\wedge\text{-E}}{C}\,\Rightarrow\text{-E}}{(A\Rightarrow C)}\,\Rightarrow\text{-I}}{((A\Rightarrow B)\wedge(B\Rightarrow C))\Rightarrow(A\Rightarrow C)}\,\Rightarrow\text{-I}$$

Auch wenn der Zusammenhang zum informalen mathematischen Schließen deutlich zu erkennen ist, ist der Kalkül des natürlichen Schließens nicht ganz leicht zu handhaben. Besonders die Verwendung von Regeln, die einmal getroffene Annahmen wieder entfernen können, bereitet Schwierigkeiten. Sie machen es notwendig, bei der Beweisführung jederzeit den gesamten bisherigen Beweisbaum zu berücksichtigen und eine Übersicht darüber zu behalten, welche Annahmen noch gelten und welche schon entfernt wurden.

2.1.6.3 Sequenzenkalküle

Gentzen selbst hatte bereits die globale Sicht mit der notwendigen Verwaltung von Annahmen als ein Problem bei den Kalkülen der natürlichen Deduktion erkannt. Er entwickelte daher in der gleichen Arbeit [Gentzen, 1935] einen Kalkül, der die Behandlung von Annahmen während einer Ableitung überflüssig machte, aber die grundsätzliche Sichtweise des natürlichen Schließens beibehielt. Die Modifikationen sind verhältnismäßig einfach. Anstatt Formeln als Kern des logischen Schließens zu betrachten, stellte Gentzen sogenannte *Sequenzen* in den Mittelpunkt, die aus einer Formel zusammen mit der Liste der aktuell gültigen Annahmen bestehen. Sie haben die äußere Form

$$A_1,\ldots,A_n \vdash C \qquad (n\geq 0)$$

und bedeuten, daß die Formel C von den Annahmen $A_1,\ldots,A_n$ abhängt.[16] Regeln handhaben nunmehr Sequenzen anstelle logischer Formeln. Auf den ersten Blick scheint dies nicht mehr als ein formaler Trick zu sein, der nur zusätzliche Schreibarbeit einbringt. Der Vorteil aber ist, daß *Sequenzenkalküle* eine *lokale* Sicht auf Beweise ermöglichen. In jedem Beweisschritt ist es nun nur noch notwendig, die aktuelle Sequenz zu kennen.[17]

Bis auf diese Modifikation sind Sequenzenbeweise praktisch identisch mit Beweisen im Kalkül des natürlichen Schließens und die Regeln des Sequenzenkalküls ergeben sich unmittelbar aus den Regeln des natürlichen Schließens. Wie zuvor dienen die Einführungsregeln für ein logisches Zeichen dazu, Formeln mit dem entsprechenden Zeichen zu erzeugen. Da manche Einführungsregeln aber auch Annahmen – also Formeln der linken Seite – entfernen, benötigen wir nun Regeln, welche zu jedem logischen Zeichen eine entsprechende Formel auf der Annahmenseite erzeugen. Diese Regeln übernehmen die Rolle der Eliminationsregeln, denn sie bearbeiten wie sie die Frage "*welche Informationen folgen aus einer gegebenen Formel?*".

Wir wollen dies am Beispiel der Regeln für die Konjunktion erläutern. Die Eliminationsregeln des logischen Schließens besagt, daß man aus $A\wedge B$ sowohl A als auch B folgern kann, also $\frac{A\wedge B}{A}$ und $\frac{A\wedge B}{B}$. Im Sequenzenkalkül wird dieser Gedanke nun mit dem Konzept der Konsequenzen von Annahmen verbunden. Da A aus $A\wedge B$ gefolgert werden darf, dürfen wir schließen, daß jede Aussage, die aus A und einer Menge anderer Annahmen – bezeichnet mit Γ – folgt, mit Sicherheit auch aus $A\wedge B$ und Γ folgt: $\frac{\Gamma, A \vdash C}{\Gamma, A\wedge B \vdash C}$.

[16] Semantisch gesehen entspricht eine Sequenz einem *Urteil*, welches besagt, daß eine mathematische Aussage (bezeichnet durch die Formel C) aus einer Reihe von anderen Aussagen folgt. Der Kalkül simuliert also eigentlich nicht das logische Schließen über Aussagen sondern liefert Begründungen für die Gültigkeit bestimmter logischer Schlüsse. Allerdings kann man mathematische Aussagen als spezielle Urteile ansehen, nämlich daß die Aussage ohne Voraussetzungen folgt.

[17] Sequenzenkalküle sind daher ideal für eine Kooperation von Mensch und Computer beim Führen formaler Beweise. Die lokale Sicht ermöglicht es dem Menschen, den Beweis zu steuern, während der Computer für die korrekte Anwendung der Regeln sorgt.

Axiom	$\dfrac{}{\Gamma,A \vdash A}$	Schnitt	$\dfrac{\Gamma \vdash A \quad \Delta,A \vdash C}{\Gamma,\Delta \vdash C}$
		Λ-E	$\dfrac{}{\Gamma,\ \Lambda \vdash A}$
$\neg$-I	$\dfrac{\Gamma,\ A \vdash \Lambda}{\Gamma \vdash \neg A}$	$\neg$-E	$\dfrac{\Gamma \vdash A}{\Gamma,\neg A \vdash \Lambda}$
$\wedge$-I	$\dfrac{\Gamma{\vdash}A \quad \Gamma{\vdash}B}{\Gamma \vdash A\wedge B}$	$\wedge$-E	$\dfrac{\Gamma,A \vdash C}{\Gamma,A\wedge B \vdash C} \quad \dfrac{\Gamma,B \vdash C}{\Gamma,A\wedge B \vdash C}$
$\vee$-I	$\dfrac{\Gamma \vdash A}{\Gamma \vdash A\vee B} \quad \dfrac{\Gamma \vdash B}{\Gamma \vdash A\vee B}$	$\vee$-E	$\dfrac{\Gamma,A \vdash C \quad \Gamma,B \vdash C}{\Gamma,A\vee B \vdash C}$
$\Rightarrow$-I	$\dfrac{\Gamma,\ A \vdash B}{\Gamma \vdash A\Rightarrow B}$	$\Rightarrow$-E	$\dfrac{\Gamma \vdash A \quad \Delta,B \vdash C}{\Gamma,\Delta,A\Rightarrow B \vdash C}$

Abbildung 2.3: Sequenzenkalkül für die Aussagenlogik

In erster Näherung werden Eliminationsregeln also genau in umgekehrter Reihenfolge aufgeschrieben. Statt die Formel auf der rechten Seite zu eliminieren wird sie auf der linken Seite erzeugt.[18] Abbildung 2.3 faßt die entsprechenden Regeln für die intuitionistische Aussagenlogik zusammen.[19] Man beachte dabei, daß die syntaktischen Metavariablen Γ und Δ beliebige *Mengen* von Formeln bezeichnen. Die hervorgehobenen Formeln einer Regel dürfen also an jeder beliebigen Stelle in der Liste der Annahmen stehen.[20] Die Axiom-Regel ist nötig, um Annahmen tatsächlich auch verwenden zu können. Die Schnittregel erlaubt eine bessere Strukturierung von Beweisen (man beweist erst ein Teilziel und verwendet es dann), ist aber nicht unbedingt erforderlich. In seinem Hauptsatz [Gentzen, 1935] zeigte Gentzen, daß alle Schnittregeln aus einem Beweis gefahrlos entfernt werden können. Allerdings vergrößert sich der Beweis dadurch erheblich.

Wie die bisher vorgestellten Kalküle sind Sequenzenkalküle eigentlich synthetischer Natur. Ein Beweis wird von den Axiomen ausgehend entwickelt bis man am gewünschten Ziel ist. Allerdings lassen sich die Regeln verhältnismäßig leicht so umwandeln, daß sie einen analytischen Sequenzenkalkül liefern. Man muß hierzu lediglich die Richtung der Regeln umkehren und gegebenenfalls Steuerungsparameter ergänzen, welche diejenigen Informationen geben, die nach einer synthetischen Anwendung einer Regel verlorengegangen sind.

So muß zum Beispiel bei der Umkehrung der Schnittregel $\frac{\Gamma \vdash A \quad \Delta,A \vdash C}{\Gamma,\Delta \vdash C}$ angegeben werden, welche Formel A für den Schnitt benutzt wird, da diese ja nachher nicht mehr vorhanden ist. Ebenso kann man in der Zielsequenz die Aufteilung der Annahmen in Γ und Δ nicht erkennen. Um den Kalkül nicht unnötig zu verkomplizieren, übernimmt man bei einem analytischen Vorgehen einfach alle Annahmen und entscheidet später, welche man davon benötigt. Insgesamt bekommt die Regel die Gestalt

$$\frac{\Gamma \vdash C}{\Gamma \vdash A \quad \Gamma,A \vdash C}\ \texttt{cut}\ A.$$

Weitere Details, eine computernähere Präsentation und eine semantische Rechtfertigung der Regeln des analytischen Sequenzenkalküls werden wir im folgenden zusammen mit der Prädikatenlogik besprechen.

[18] Entsprechend bezeichnete Gentzen die Regeln nun als Linksregeln (*-l*) und Rechtsregeln (*-r*). Diese Konvention wird von vielen Mathematikern übernommen während Computersysteme meist auf den Begriffen Einführung und Elimination basieren.

[19] Der ursprüngliche Sequenzenkalkül *LK* für die klassische Aussagenlogik ist etwas komplizierter als der hier angegebene intuitionistische Kalkül *LJ*, da er mehrere Formeln auch auf der rechten Seite zuläßt, die als mögliche Alternativen (also eine Art Disjunktion) betrachtet werden. Der Kalkül *LJ* kann allerdings auch durch Abschwächung der Regel Λ-E zu $\frac{\Gamma,\neg A \vdash \Lambda}{\Gamma \vdash A}$ in einen Kalkül für die klassische Aussagenlogik umgewandelt werden.

[20] Es sei angemerkt, daß Gentzens ursprüngliche Sequenzenkalküle *Folgen* von Annahmen verarbeiten, bei denen es von Bedeutung ist, wo und wie oft eine Formel in den Annahmen auftritt. Für die Vollständigkeit dieses Kalküls werden daher neben den in Abbildung 2.3 angegebenen logischen Regeln auch sogenannte *Strukturregeln* zur Vertauschung, Kontraktion und Verdünnung von Annahmen benötigt. Diese Vorgehensweise ist bei einer mechanischen Abarbeitung einer Sequenz als Text (z.B. in einem Computer) erforderlich, wird bei einer formalen Beweisführung durch Menschen jedoch eher als unnötiger Ballast angesehen.

2.2 Prädikatenlogik

Unter allen formalen Sprachen ist die *Prädikatenlogik* am weitesten verbreitet. Für den eingegrenzten Bereich logischer Schlüsse, deren Gültigkeit sich ausschließlich auf die Struktur mathematischer Aussagen stützt, ist sie ein überaus praktisches und mächtiges Handwerkzeug. Im Prinzip läßt sich jede mathematische Aussage in der Sprache der Prädikatenlogik formulieren. Man muß hierzu nur die entsprechenden mathematischen Begriffe in der Form von Prädikats- und Funktionssymbolen beschreiben. Da diese Symbole allerdings keine festgelegte Bedeutung haben – selbst wenn die Namensgebung dies suggerieren mag – ist nicht festgelegt, bis zu welchem Detail man die Aussage vor ihrer Formalisierung mit Hilfe logischer Strukturierungsmechanismen zerlegen sollte. Die optimale Form ergibt sich aus dem Verwendungszweck.

Beispiel 2.2.1 (Formalisierung mathematischer Aussagen)

Faßt man die Aussage "*Alle Studenten, die ihre Prüfungsleistungen innerhalb von zwei Jahren ablegen, haben die Diplomhauptprüfung bestanden*" als eine logische Einheit der Diplomprüfungsordnung auf, so wird man sie vielleicht nicht weiter unterstrukturieren und sie vielleicht durch das Prädikat

`DPO_18_4`

ausdrücken. Diese Formulierung enthält allerdings nicht die geringsten Hinweise auf die intendierte Bedeutung und wird nur sehr eingeschränkt verwendbar sein. Etwas mehr Klarheit gewinnt man schon, wenn man fixiert, daß es hier um eine Bestimmung geht, die für alle Studenten Gültigkeit hat und zudem eine "wenn-dann" Beziehung enthält. In diesem Sinne präzisiert lautet die Aussage "*Für alle Studenten gilt: wenn sie ihre Prüfungsleistungen innerhalb von zwei Jahren ablegen, dann haben sie die Diplomhauptprüfung bestanden*". Bei der Formalisierung verwendet man hierzu als logische Symbole den Allquantor $\forall$ sowie die Implikation $\Rightarrow$ und schreibt:

$\forall$`student. Prüfungen_in_zwei_Jahren_abgelegt(student)` $\Rightarrow$ `Diplom_bestanden(student)`

oder mit etwas kürzeren Prädikatsnamen

$\forall$`s. P_abgelegt(s)` $\Rightarrow$ `Diplom(s)`

Bei der zweiten Form zeigt sich ein Dilemma. Durch die Verwendung weniger suggestiver Variablen- und Prädikatsnamen geht ein Teil der Informationen verloren, die vorher vorhanden waren. Genau besehen waren diese jedoch niemals vorhanden, da die Prädikatenlogik über die Bedeutung ihrer Variablen-, Funktionen- und Prädikatsnamen nichts aussagt, wenn man nicht weitere Informationen beisteuert.

Wieviele dieser Informationen man explizit hinzugibt, ist reine Ermessenssache. So kann es für die Bestimmung relevant sein, daß es sich bei den Objekten, die mit dem Symbol `s` bezeichnet werden, um Studenten handelt. Es mag aber sein, daß die Bestimmung auf andere Objekte niemals angewandt werden wird, und dann ist es eigentlich unbedeutend, wofür `s` nun genau steht. Sicherlich von Interesse aber ist, auszudrücken, daß die Dauer der Prüfung nicht mehr als zwei Jahre betrug. So liest sich unsere obige Aussage nun wie folgt: "*Für alle Objekte* `s` *gilt: wenn* `s` *ein Student ist und* `s` *seine Diplomprüfungen abgelegt hat und die Dauer dieser Prüfungen maximal 2 Jahre war, dann hat* `s` *die Diplomhauptprüfung bestanden*". Hier wird einerseits ein funktionaler Zusammenhang (die Dauer der Prüfung) benutzt und eine Verkettung von Voraussetzungen. Bei der Formalisierung verwendet man hierzu Funktionssymbole sowie die Konjunktion $\wedge$ und schreibt:

$\forall$`s. Student(s)` $\wedge$ `abgelegt(s,prüfung)` $\wedge$ $\leq$`(dauer(prüfung), 2)` $\Rightarrow$ `Diplom(s)`

Man könnte die Strukturierung jetzt noch weitertreiben, da sie immer noch einige Fragen offen läßt. Endgültige Eindeutigkeit wird man jedoch nie erreichen können. An irgendeinem Punkt muß man sich darauf verlassen, daß die Formulierung auf der Basis der bisher gewählten Begriffe (und möglicher Zusatzinformationen über diese) für den Anwendungszweck hinreichend klar ist.[21]

[21] Die Typentheorie, die wir in Kapitel 3 vorstellen werden geht einen anderen Weg. Sie geht – wie die Mengentheorie – davon aus, daß wenige Objekte wie natürliche Zahlen und Funktionen elementarer Natur sind, deren Regeln erforscht werden müssen. Alle anderen Konzepte sind dann auf der Basis einer festen Formalisierung dieser Objekte auszudrücken. Das heißt, daß jede Formalisierung eine feste Bedeutung erhält (sofern sie nicht Variable eines Quantors ist).

Die übliche Prädikatenlogik verwendet zur Strukturierung mathematischer Aussagen die in Beispiel 2.1.2 auf Seite 12 erklärten Operatoren *Negation*, *Konjunktion*, *Disjunktion*, *Implikation*, *Universelle Quantifizierung* und *Existentielle Quantifizierung*, deren Bedeutung intuitiv leicht zu erklären ist. Nur bei der Bedeutung der Quantoren bleiben einige Fragen offen, da es erlaubt ist, über beliebige Objekte zu quantifizieren, deren Natur völlig im unklaren liegt. Dies ist sicherlich legitim, wenn man die Logik verwendet, um immer wieder über den gleichen Bereich – also zum Beispiel natürliche Zahlen – zu reden. Hier stimmt die Annahme, daß alle Variablen, Funktionen und Prädikate mit Bezug auf eine festes *Universum* erklärt werden können, und man ist sich darüber im klaren, daß der Satz $\forall$x. x=0 $\vee$ x$\geq$1 eine hinreichend präzise Formalisierung einer mathematischen Grundeigenschaft ist. Das wird aber sofort anders, wenn der gleiche Satz als Eigenschaft reeller Zahlen angesehen wird, denn nun fehlen ein paar Voraussetzungen. Denkt man bei der Interpretation gar an Objekte wie Graphen, Mengen oder Liste, so wird der Satz schlicht sinnlos.

In der Prädikatenlogik behilft man sich zuweilen damit, daß man Informationen über den Bereich, aus dem ein Objekt gewählt werden darf, als Voraussetzung in die Aussage hineinnimmt. Man schreibt zum Beispiel

$\forall$x. nat(x) $\Rightarrow$ (x=0 $\vee$ x$\geq$1)

und nimmt separat die Axiome der natürlichen Zahlen hinzu. Auf diese Art kann man die bisherige Sprache beibehalten, handelt sich aber für das formale Beweisen eine große Menge zu verarbeitender Formeln ein.

Eine Alternative hierzu ist die Erweiterung der Prädikatenlogik zu einer *Sortenlogik*, indem man die Informationen über die Natur der Objekte als eine Art *Typdeklaration* in den Quantor mit hineinnnimmt und dann die Formel wie folgt aufschreibt:

$\forall$x:$\mathbb{N}$. x=0 $\vee$ x$\geq$1

Die obige Schreibweise ist nicht nur eleganter, sondern ermöglicht auch eine klarere Charakterisierung der Semantik einer Aussage. Zudem wird sie der Denkweise moderner Programmiersprachen gerechter, bei denen man eine Typdisziplin für Variablen nicht mehr missen möchte. Dafür muß der Formalismus allerdings die Bezeichner für die Sorten mitschleppen, selbst wenn diese nicht weiter verarbeitet werden. Da wir spätestens bei der Besprechung der Typentheorie in Kapitel 3 die Bereiche, aus denen ein Objekt gewählt werden darf, ohnehin zum zentralen Thema machen werden, wollen wir bei der nun folgenden Präzisierung der Prädikatenlogik die Sorten mit in den Formalismus mit aufnehmen. Die bekannte unsortierte Prädikatenlogik, die wir im Abschnitt 2.1 bereits informal erklärt haben, ergibt sich hieraus einfach durch Herausstreichen jeglicher Sorteninformationen.

Ein weiterer Nachteil der Prädikatenlogik gegenüber der gängigen mathematischen Notation ist die strikte Verwendung der Präfixschreibweise auch dort, wo die Mathematik Infixoperatoren wie `+, -, *` usw. benutzt.[22] Die einzige Ausnahme von dieser Regelung wird zuweilen bei der *Gleichheit* gemacht, da Gleichheit ein derart fundamentales Konzept ist, daß man ihre Bedeutung nicht freistellen sollte. Deshalb nehmen wir – wie viele Formulierungen der Prädikatenlogik – auch die Gleichheit als fest definiertes Prädikatssymbol in Infixschreibweise mit in die formale Sprache auf, deren Syntax und Semantik wir nun festlegen wollen.

2.2.1 Syntax

Die Sprache der Prädikatenlogik besteht aus Formeln, die wiederum aus Termen, Prädikaten, und speziellen logischen Symbolen (*Junktoren* und *Quantoren*) aufgebaut werden. Zur Präzisierung von Termen benötigt man Funktions- und Variablensymbole, die aus bestimmten, nicht näher präzisierten Alphabeten ausgewählt werden.

[22] Auch dieses Problem werden wir erst im Zusammenhang mit der Besprechung der Typentheorie in Kapitel 3 lösen.

Definition 2.2.2 (Terme)

Es sei $\mathcal{V}$ ein Alphabet von Variablen(-symbolen). Es sei $\mathcal{F}^i$ ein Alphabet von i-stelligen Funktionssymbolen für jedes $i\geq0$ und $\mathcal{F} = \bigcup_{i=0}^{\infty}\mathcal{F}^i$.

Die Terme der Sprache der Prädikatenlogik sind induktiv wie folgt definiert.

- *Jede Variable $x \in \mathcal{V}$ ist ein Term.*
- *Ist $f \in \mathcal{F}^0$ ein beliebiges 0-stelliges Funktionssymbol, dann ist f ein Term (eine Konstante).*
- *Sind $t_1,\ldots,t_n$ Terme ($n\geq1$) und ist $f \in \mathcal{F}^n$ ein beliebiges n-stelliges Funktionssymbol, dann ist $f(t_1,\ldots,t_n)$ ein Term.*

Man beachte hierbei, daß Variablen- und Funktionssymbole nicht unbedingt aus einzelnen Buchstaben bestehen müssen. Es ist durchaus legitim, Namen wie `x14`, $\mathtt{f}_1$, `student` und ähnliches zu verwenden. Auch ist nicht unbedingt gefordert, daß die Alphabete $\mathcal{V}$ und $\mathcal{F}$ disjunkt sein müssen. Meist geht es aus dem Kontext hervor, um welche Art von Symbolen es sich handelt. Es hat sich jedoch eingebürgert, die auf Seite 14 eingeführten Konventionen für syntaktische Metavariablen in ähnlicher Form auch als Konventionen für die Namensgebung von Symbolen der Objektsprache zu verwenden, um Mißverständnisse zu vermeiden. Wir wollen die Bildung von Termen gemäß der obigen Regeln nun anhand einiger Beispiele veranschaulichen.

Beispiel 2.2.3

Die folgenden Ausdrücke sind korrekte Terme im Sinne von Definition 2.2.2

`x`, `24`, `peter`, `vater(peter)`, `max(2,3,4)`, `max(plus(4,plus(5,5)),23,5)`

Dabei ist die Rolle von `x`, `24`, `peter` ohne unsere Namenskonventionen nicht eindeutig festzustellen. Üblicherweise betrachten wir `x` als Variable und `24` genauso wie `peter` als nullstelliges Funktionssymbol, d.h. als *Konstante*. `vater`, `max` und `plus` müssen Funktionszeichen sein, wobei `vater` einstellig, `max` dreistellig, und `plus` zweistellig ist.

Der Ausdruck `max(2,3)` wäre an dieser Stelle erlaubt, aber problematisch. Zwar ist prinzipiell nicht ausgeschlossen, daß ein Symbol mit verschiedenen Stelligkeiten benutzt wird (die $\mathcal{F}^i$ müssen nicht disjunkt sein). Eine Verwendung von `max` als zwei- und als dreistelliges Symbol wird aber die meisten Inferenzsysteme überlasten, so daß man auf derartige Freiheiten besser verzichten sollte.

Die Definition prädikatenlogischer Formeln ist strukturell der eben aufgestellten Definition von Termen sehr ähnlich. Wir definieren sie als atomare und als induktiv zusammengesetzte Formeln.

Definition 2.2.4 (Formeln)

Es sei für jedes $i\geq0$ $\mathcal{P}^i$ ein Alphabet von i-stelligen Prädikatssymbolen, $\mathcal{P} = \bigcup_{i=0}^{\infty}\mathcal{P}^i$ und $\mathcal{T}$ ein Alphabet von Bereichssymbolen (Typen).

Die Formeln der Sprache der Prädikatenlogik sind induktiv wie folgt definiert.

- *Λ ist eine (atomare) Formel.*
- *Ist $P \in \mathcal{P}^0$ ein 0-stelliges Prädikatssymbol, dann ist P eine atomare Formel (oder Aussagenvariable).*
- *Sind t_1 und t_2 Terme, dann ist t_1 = t_2 eine atomare Formel.*
- *Sind $t_1,\ldots,t_n$ Terme ($n\geq1$) und ist $P \in \mathcal{P}^n$ ein beliebiges n-stelliges Prädikatssymbol, dann ist $P(t_1,\ldots,t_n)$ eine atomare Formel.*
- *Sind A und B Formeln, $x \in \mathcal{V}$ ein Variablensymbol und $T \in \mathcal{T}$ ein Bereichssymbol, dann sind*

  $\neg A$, $A \wedge B$, $A \vee B$, $A \Rightarrow B$, $\forall x{:}T.A$, $\exists x{:}T.A$ *und* (A) *Formeln.*

Man beachte, daß im Unterschied zur Definition 2.2.2 die Argumente von Prädikatssymbolen nicht Formeln, sondern Terme sind. Das Symbol Λ ist ein spezielles 0-stelliges Prädikatssymbol, das für die atomare Aussage "*falsch*" verwendet wird. Das Symbol = ist ein spezielles 2-stelliges Prädikatssymbol, das die gewöhnliche Gleichheit ausdrückt und in Infixnotation verwendet wird. Auch hierzu wollen wir ein paar Beispiele ansehen.

Beispiel 2.2.5

Die folgenden Ausdrücke sind korrekte Formeln im Sinne von Definition 2.2.4

4=plus(2,3), $\leq$(max(2,3,4),7), (4=5) $\Rightarrow \Lambda$, Sein $\vee\neg$Sein, lange_währt $\Rightarrow$ endlich_gut,
$\forall$x:$\mathbb{N}$.$\exists$x:$\mathbb{Z}$. x=x, $\forall$x:$\mathbb{N}$.$\exists$y:$\mathbb{Z}$. $\leq$(*(y,y), x) $\wedge$ <(x, *(plus(y,1),plus(y,1)))

Man beachte daß die syntaktische Korrektheit einer Formel nichts mit ihrer Wahrheit zu tun hat, denn es muß ja auch möglich sein, falsche Aussagen zu *formulieren*. Deshalb ist `4 = 5` eine korrekte Formel, auch wenn wir damit keine wahre Aussage verbinden. Keine korrekten Formeln sind dagegen

plus(plus(2,3),4), $\wedge$ so_weiter, $\forall$x:$\mathbb{N}$. x(4)=x, $\forall$x. x=x.

Der erste Ausdruck ist ein Term, aber keine Formel, im zweiten Ausdruck fehlt eine Formel auf der linken Seite des Symbols $\wedge$ und im dritten Beispiel wird eine Variable unzulässigerweise als Funktionszeichen benutzt. Das vierte Beispiel ist eine Formel der unsortierten Prädikatenlogik, die aber im Sinne der Definition 2.2.4 syntaktisch nicht korrekt ist.

Die Verwendung von nullstelligen Funktions- und Prädikatssymbolen als Terme bzw. atomare Formeln, ohne daß hierzu Klammern verwendet werden müssen, entspricht der in Beispiel 2.1.1 auf Seite 11 angesprochenen *Konvention* zur Abkürzung formaler Ausdrücke. Die anderen dort angesprochenen Konventionen sind für eine syntaktische Korrektheit von Formeln eigentlich überflüssig. Ausdrücke wie

$\exists$y:$\mathbb{N}$. gerade(y) $\wedge$ $\geq$(y,2) $\Rightarrow$ y=2 $\wedge$ >(y,20)

sind syntaktisch absolut korrekt. Jedoch läßt sich die Frage, welche Aussagen sie beschreiben, nicht eindeutig beantworten, da unklar ist, was die wirkliche Struktur dieses Ausdrucks ist. Sie könnte gelesen werden als

$\exists$y:$\mathbb{N}$. (gerade(y) $\wedge$ $\geq$(y,2)) $\Rightarrow$ (y=2 $\wedge$ >(y,20)) oder als

$\exists$y:$\mathbb{N}$. (gerade(y) $\wedge$ ($\geq$(y,2) $\Rightarrow$ y=2)) $\wedge$ >(y,20) oder als

$\exists$y:$\mathbb{N}$. gerade(y) $\wedge$ ($\geq$(y,2) $\Rightarrow$ (y=2 $\wedge$ >(y,20)))

Mit diesen drei Lesarten verbinden wir sehr unterschiedliche mathematische Aussagen. Die erste Aussage wird z.B. wahr, wenn wir eine ungerade Zahl für `y` einsetzen. Die zweite Aussage ist falsch, denn wir müssten eine gerade Zahl einsetzen, die größer als 20 ist, aber den Wert 2 annimmt, wenn sie größer oder gleich 2 ist. In der dritten können wir nur die Zahl `0` als Wert für `y` verwenden. Es wäre nun lästig, wenn wir die Eindeutigkeit einer Formalisierung nur durch die Verwendung von Klammern erreichen können. Aus diesem Grunde gibt es Konventionen, welche die Struktur ungeklammerter Ausdrücke eindeutig festlegen.[23]

Definition 2.2.6 (Konventionen zur Eindeutigkeit prädikatenlogischer Ausdrücke)

Für die Bindungskraft der logischen Operatoren gilt die folgende Reihenfolge:

$\neg$ bindet stärker als $\wedge$, dann folgt $\vee$, dann $\Rightarrow$, dann $\exists$ und zum Schluß $\forall$.

In einer Formel braucht eine durch einen stärker bindenden Operator gebildete Teilformel nicht geklammert zu werden. Bei gleicher Bindungskraft gilt (im Zweifel) Rechtsassoziativität.

Diese Vereinbarung erlaubt es uns, überflüssige Klammern fallen zu lassen. Die Vereinbarung der Rechtsassoziativität ist eigentlich nur für die Implikation von Bedeutung. Auch hier geben wir ein paar Beispiele.

Beispiel 2.2.7

A $\wedge$ $\neg$B	entspricht	A $\wedge$ ($\neg$B)
A $\wedge$ B $\vee$ C	entspricht	(A $\wedge$ B) $\vee$ C
A $\Rightarrow$ B $\vee$ C	entspricht	A $\Rightarrow$ (B $\vee$ C)
A $\Rightarrow$ B $\Rightarrow$ C	entspricht	A $\Rightarrow$ (B $\Rightarrow$ C)
$\exists$x:$\mathbb{N}$. A $\wedge$ B(x) $\vee$ C(x)	entspricht	($\exists$x:$\mathbb{N}$. (A $\wedge$ B(x)) $\vee$ C(x))

Bei der obigen Beispielformel entspricht die erste Lesart der Konvention aud Definition 2.2.6.

[23] Diese Konventionen sind besonders von großer Bedeutung, wenn man die formale Sprache mit dem Computer verarbeiten möchte und dennoch eine gewisse Freiheit bei der Formalisierung haben will. Zusammen mit den Definition 2.2.2 und 2.2.4 führen diese Konventionen zu einer *eindeutigen* Grammatik für die Prädikatenlogik, die sich von einem Parser leichter verarbeiten läßt.

2.2.2 Semantik

Obwohl es sich bei logischen Formeln eigentlich nur um reine Zeichenfolgen ohne jede Bedeutung handelt, assoziieren wir natürlich bestimmte Begriffe mit Konstanten wie `peter` oder `24` und Prädikaten wie $\leq$(4,29). Um dieser gedachten Zuordnung zwischen formalen Ausdrücken und semantischen Inhalten Ausdruck zu verleihen, bedient man sich einer *Interpretationsfunktion*, die jedem Variablen-, Funktions-, Prädikats- und Bereichssymbol ein mengentheoretisches Objekt zuweist, und setzt diese dann auf Terme und logische Formeln fort. In mathematischen Begriffen definiert man die Semantik von Formeln dann wie folgt.

Definition 2.2.8 (Interpretation prädikatenlogischer Formeln)

1. *Eine Interpretation der Prädikatenlogik ist ein Paar $(\iota,\mathcal{U})$, wobei $\mathcal{U}$ eine Menge (das Universum) und ι eine Abbildung ist, für die gilt*
   - *ι weist jeder Variablen $x\in\mathcal{V}$ ein Objekt aus $\mathcal{U}$ zu.*
   - *ι weist jedem Funktionssymbol $f\in\mathcal{F}^n$ eine n-stellige Funktion $\phi:\mathcal{U}^n\to\mathcal{U}$ zu.*
   - *ι weist jedem Bereichssymbol $T\in\mathcal{T}$ eine Menge zu, und es gilt $\bigcup_{T\in\mathcal{T}}\iota(T)\subseteq\mathcal{U}$.*
   - *ι weist jedem Prädikatssymbol $P\in\mathcal{P}^n$ eine charakteristische Funktion $\Pi:\mathcal{U}^n\to\{\mathsf{wahr},\mathsf{falsch}\}$ zu.*

2. *Eine Interpretationsfunktion ι wird auf prädikatenlogische Terme und Formeln gemäß der folgenden Regeln homomorph fortgesetzt.*
   - *Sind $t_1,\ldots,t_n$ Terme ($n\geq 1$) und ist $f\in\mathcal{F}^n$ ein beliebiges n-stelliges Funktionssymbol, dann ist* $\iota(f(t_1,\ldots,t_n)) = \iota(f)(\iota(t_1),\ldots,\iota(t_n))$.
   - $\iota(\Lambda) = \mathsf{falsch}$
   - *Sind t_1 und t_2 Terme, dann ist*
     $$\iota(t_1 = t_2) = \begin{cases} \mathsf{wahr} & \textit{falls die Objekte } \iota(t_1) \textit{ und } \iota(t_2) \textit{ in } \mathcal{U} \textit{ gleich sind} \\ \mathsf{falsch} & \textit{sonst} \end{cases}$$
   - *Sind $t_1,\ldots,t_n$ Terme ($n\geq 1$) und ist $P\in\mathcal{P}^n$ ein beliebiges n-stelliges Prädikatssymbol, dann ist* $\iota(P(t_1,\ldots,t_n)) = \iota(P)(\iota(t_1),\ldots,\iota(t_n))$.
   - *Sind A und B Formeln, $x\in\mathcal{V}$ ein Variablensymbol und $T\in\mathcal{T}$ ein Bereichssymbol, dann ist*
     $$\iota(\neg A) = \begin{cases} \mathsf{wahr} & \textit{falls } \iota(A) = \mathsf{falsch} \\ \mathsf{falsch} & \textit{falls } \iota(A) = \mathsf{wahr} \end{cases}$$
     $$\iota(A\wedge B) = \begin{cases} \mathsf{wahr} & \textit{falls } \iota(A) = \mathsf{wahr} \textit{ und } \iota(B) = \mathsf{wahr} \\ \mathsf{falsch} & \textit{falls } \iota(A) = \mathsf{falsch} \textit{ oder } \iota(B) = \mathsf{falsch} \end{cases}$$
     $$\iota(A\vee B) = \begin{cases} \mathsf{wahr} & \textit{falls } \iota(A) = \mathsf{wahr} \textit{ oder } \iota(B) = \mathsf{wahr} \\ \mathsf{falsch} & \textit{falls } \iota(A) = \mathsf{falsch} \textit{ und } \iota(B) = \mathsf{falsch} \end{cases}$$
     $$\iota(A\Rightarrow B) = \begin{cases} \mathsf{wahr} & \textit{falls aus } \iota(A) = \mathsf{wahr} \textit{ immer } \iota(B) = \mathsf{wahr} \textit{ folgt} \\ \mathsf{falsch} & \textit{falls } \iota(A) = \mathsf{wahr} \textit{ und } \iota(B) = \mathsf{falsch} \end{cases}$$
     $$\iota(\forall x{:}T.A) = \begin{cases} \mathsf{wahr} & \textit{falls } \iota_x^u(A) = \mathsf{wahr} \textit{ für \underline{alle} } u\in\iota(T) \textit{ ist} \\ \mathsf{falsch} & \textit{falls } \iota_x^u(A) = \mathsf{falsch} \textit{ für \underline{ein} } u\in\iota(T) \textit{ ist} \end{cases}$$
     $$\iota(\exists x{:}T.A) = \begin{cases} \mathsf{wahr} & \textit{falls } \iota_x^u(A) = \mathsf{wahr} \textit{ für \underline{ein} } u\in\iota(T) \textit{ ist} \\ \mathsf{falsch} & \textit{falls } \iota_x^u(A) = \mathsf{falsch} \textit{ für \underline{alle} } u\in\iota(T) \textit{ ist} \end{cases}$$
     $$\iota((A)) = \iota(A)$$
     Dabei bezeichnet ι_x^u diejenige Interpretationsfunktion, die identisch mit ι für alle von der Variablen x verschiedenen Symbole ist und für die $\iota_x^u(x)$ der Wert u ist.[24]

Wir sehen also, daß durch die Interpretation im Grunde nur von der Symbolwelt in eine andere mathematische Denkwelt, nämlich die Mengentheorie, abgebildet wird. Der Unterschied ist lediglich, daß wir von der Mengentheorie eine relativ klare Vorstellung haben, die nicht weiter erklärt werden muß. Die Interpretation versetzt uns also in die Lage, den Wahrheitsgehalt einer Formel genau zu beurteilen. Zu beachten ist dabei, daß nullstellige Funktionen mit Elementen ihres Bildbereiches identifiziert werden. Nullstelligen Funktionssymbolen weist eine Interpretation also ein konstantes Objekt aus $\mathcal{U}$ zu und nullstelligen Prädikatssymbolen einen Wahrheitswert aus {wahr,falsch}.

Das folgende Beispiel zeigt, wie man die Bedeutung eines Ausdrucks für eine vorgegebene Interpretation "ausrechnen" kann.

Beispiel 2.2.9

Die Interpretationsfunktion ι interpretiere Zahlsymbole in Dezimaldarstellung in der bekannten Weise, d.h. $\iota(\texttt{2})$ ist die Zahl *zwei*, $\iota(\texttt{12})$ ist die Zahl *zwölf* usw. $\iota(\texttt{max})$ sei die Funktion ϕ_{max}, welche das Maximum dreier Zahlen bestimmt. $\iota(\leq)$ sei die charakteristische Funktion $\Pi_{\leq}$ der "*kleiner-gleich*"-Relation, also genau dann wahr, wenn das erste Argument nicht größer als das zweite ist. $\iota(\mathbb{N})$ sei die Menge der natürlichen Zahlen.[25] Dann wird der Ausdruck $\leq\texttt{(max(2,3,4),7)}$ wie folgt interpretiert

$$\begin{array}{rl} & \iota(\leq\texttt{(max(2,3,4),7)}) \\ = & \iota(\leq)(\iota(\texttt{max(2,3,4)}),\iota(\texttt{7})) \\ = & \Pi_{\leq}(\iota(\texttt{max})(\iota(\texttt{2}),\iota(\texttt{3}),\iota(\texttt{4})),\ sieben)) \\ = & \Pi_{\leq}(\phi_{max}(zwei,drei,vier),\ sieben) \\ = & \Pi_{\leq}(vier,\ sieben) \\ = & \mathsf{wahr} \end{array}$$

Sei zusätzlich – der Vollständigkeit halber – $\iota(\texttt{x})$ die Zahl *dreizehn*. Dann wird der prädikatenlogische Ausdruck $\exists\texttt{x}:\mathbb{N}.\ \leq\texttt{(max(2,3,4),x)}$ wie folgt interpretiert:

$$\begin{array}{rl} & \iota(\exists\texttt{x}:\mathbb{N}.\ \leq\texttt{(max(2,3,4),x)}) \\ = & \mathsf{wahr}\text{ genau dann, wenn } \iota_x^u(\leq\texttt{(max(2,3,4),x)}) = \mathsf{wahr}\text{ für ein } u\in\iota(\mathbb{N})\text{ ist} \\ = & \vdots \\ = & \mathsf{wahr}\text{ genau dann, wenn } \Pi_{\leq}(vier,\iota_x^u(\texttt{x})) = \mathsf{wahr}\text{ für eine natürliche Zahl } u\text{ ist.} \\ = & \mathsf{wahr}\text{ genau dann, wenn } \Pi_{\leq}(vier,u) = \mathsf{wahr}\text{ für eine natürliche Zahl } u\text{ ist.} \\ = & \mathsf{wahr} \qquad\qquad (\text{Wir wählen } u \text{ als } \mathit{fünf}) \end{array}$$

Einen Sonderfall wollen wir hier noch ansprechen. Es ist durchaus legitim, ein und dieselbe Variable zweimal innerhalb von Quantoren zu verwenden. Der Ausdruck

$$\forall\texttt{x}:\mathbb{N}.\ \geq\texttt{(x,0)}\ \wedge\ \exists\texttt{x}:\mathbb{Z}.\ \texttt{x=x}$$

ist eine absolut korrekte, wenn auch ungewöhnliche Formel der Prädikatenlogik. Damit muß es auch möglich sein, dieser eine präzise Bedeutung zuzuordnen. Die Intuition sagt uns, daß der zuletzt genannte Quantor in einer Formel Gültigkeit haben muß, daß also das `x` in `x=x` sich auf den Existenzquantor bezieht. Der Gedanke dabei ist, daß es bei quantifizierten Formeln auf den konkreten Namen einer Variablen eigentlich nicht ankommen darf, sondern daß diese nur ein Platzhalter dafür ist, daß an ihrer Stelle etwas beliebiges eingesetzt werden darf. Die Formel $\exists\texttt{x}:\mathbb{Z}.\ \texttt{x=x}$ muß also dasselbe bedeuten wie $\exists\texttt{y}:\mathbb{Z}.\ \texttt{y=y}$ und damit muß obige Formel die gleiche Bedeutung haben wie

$$\forall\texttt{x}:\mathbb{N}.\ \geq\texttt{(x,0)}\ \wedge\ \exists\texttt{y}:\mathbb{Z}.\ \texttt{y=y}$$

Unsere Definition der Semantik wird dieser Forderung gerecht, denn sie besagt, daß *innerhalb* einer quantifizierten Formel eine Interpretation ι bezüglich der quantifizierten Variable x geändert werden darf. Welcher Wert außerhalb für x gewählt wurde, hat innerhalb also keine Gültigkeit mehr.

[24] Durch die Forderung, daß $\iota_x^u(A)$ den Wert wahr für alle bzw. ein $u\in\iota(T)$ haben muß, drücken wir aus, daß die Interpretation der Formel A wahr sein muß für alle Werte (bzw. einen Wert), die wir für (die Interpretation von) x einsetzen können.

[25] Es sei vereinbart, daß die natürlichen Zahlen die Null enthalten.

In Definition 2.2.8 haben wir die Semantik der logischen Symbole durch natürlichsprachliche Formulierungen erklärt, über deren Bedeutung es klare Vorstellungen zu geben scheint. Leider gehen im Falle der Disjunktion $\vee$, der Implikation $\Rightarrow$ und des Existenzquantors $\exists$ die Meinungen darüber auseinander, was die Worte "*oder*", "*folgt*" und "*es gibt ein...*" nun genau bedeuten. Viele Mathematiker würden die folgenden Charakterisierungen als gleichbedeutend zu denen aus Definition 2.2.8 ansehen:

$$\iota(A\vee B) = \begin{cases} \mathsf{falsch} & \text{falls } \iota(A) = \mathsf{falsch} \text{ und } \iota(B) = \mathsf{falsch} \\ \mathsf{wahr} & \text{sonst} \end{cases}$$

$$\iota(A\Rightarrow B) = \begin{cases} \mathsf{falsch} & \text{falls } \iota(A) = \mathsf{wahr} \text{ und } \iota(B) = \mathsf{falsch} \\ \mathsf{wahr} & \text{sonst} \end{cases}$$

$$\iota(\exists x{:}T.A) = \begin{cases} \mathsf{falsch} & \text{falls } \iota_x^u(A) = \mathsf{falsch} \text{ für } \underline{\text{alle}}\ u\in\iota(T) \text{ ist} \\ \mathsf{wahr} & \text{sonst} \end{cases}$$

Dahinter verbirgt sich die Vorstellung, daß *jede* mathematische Aussage einen eindeutig bestimmten Wahrheitsgehalt haben *muß*. Damit sind bei der Disjunktion und der Implikation jeweils nur vier Möglichkeiten zu betrachten und dies führt dann dazu, daß die obigen Charakterisierungen wirklich äquivalent zu den in Definition 2.2.8 gegebenen werden. Auch stimmt es, daß die Interpretation einer Formel $\exists x{:}T.A$ mit Sicherheit genau dann falsch ist, wenn für alle $u\in\iota(T)$ $\iota_x^u(A) = \mathsf{falsch}$ ist.

Im Endeffekt ist aber die Annahme, daß eine Aussage wahr ist, wenn sie nicht falsch ist, ein *Postulat* – also eine Forderung, die sich nicht beweisen läßt. Eine Möglichkeit, andere von der Richtigkeit oder Falschheit dieses Postulats zu überzeugen, gibt es nicht. Ob man die obengemachte Annahme akzeptiert oder ablehnt, ist also eher eine Frage der "Weltanschauung" und weniger die Frage danach, was denn nun wahr ist.[26] Da sich im formalen Kalkül der Unterschied zwischen diesen Weltanschauungen durch Hinzunahme oder Entfernung einer einzigen Regel ausdrücken läßt, werden wir die klassische und die intuitionistische Prädikatenlogik im folgenden simultan behandeln.

Es sei angemerkt, daß die klassische Prädikatenlogik gemäß der oben angegebenen äquivalenten Charakterisierungen eigentlich auf Disjunktion, Implikation und Existenzquantor verzichten könnte, da sich diese durch $\neg$, $\wedge$ und $\forall$ simulieren lassen und somit als definitorische Erweiterung im Sinne von Abschnitt 2.1.5 (Seite 14) angesehen werden können.

Wir wollen nochmals darauf hinweisen, daß – mit Ausnahme der Gleichheit – alle Prädikats- und Funktionssymbole (insbesondere die vertrauten Symbole aus der Arithmetik) keine feste Bedeutung besitzen. Die Bedeutung dieser Symbole hängt stattdessen von der konkreten Interpretation ab und wird bei der Frage nach der Gültigkeit logischer Gesetze und ihrer Darstellung als Inferenzregeln keine Rolle spielen. In der Prädikatenlogik kann man nur strukturelle Informationen verarbeiten, aber nicht rechnen.

[26] In der Tat hat es über diese Frage seit Anfang dieses Jahrhunderts unter den Mathematikern Streitigkeiten gegeben, die schon fast religiöse Züge angenommen haben, da sowohl die klassische als auch die intuitionistische Mathematik für sich in Anspruch nahmen, die *reine Wahrheit zu verkünden* und die Vertreter der jeweils anderen Denkrichtung als Ketzer ansahen.

Für die *klassische* – Ihnen aus der Schule vertrautere – Mathematik ist die Wahrheit einer Aussage dann gesichert, wenn man weiß, daß sie nicht falsch ist. Insbesondere werden Objekte mit einer bestimmten Eigenschaft als existent angesehen, wenn man weiß, daß nicht alle Objekte diese Eigenschaft nicht besitzen. Das strapaziert zuweilen zwar die Anschauung, macht aber mathematische Beweise so viel einfacher, daß sich die klassische Mathematik weitestgehend durchgesetzt hat.

Die *intuitionistische* Mathematik dagegen geht davon aus, daß nur die natürlichen Zahlen – wie der Name schon sagt – etwas Naturgegebenes sind, von dem jeder Mensch ein Verständnis besitzt. Alle anderen mathematischen Objekte dagegen – also ganze Zahlen, Brüche, reelle und komplexe Zahlen usw. – sind reine Hilfskonstruktionen des Menschen, deren einzige Aufgabe es ist *bestimmte natürliche Phänomene eleganter zu beschreiben*. Es sind also abkürzende *Definitionen* oder – wie man sagt – *mentale Konstruktionen*. Die Frage nach der Existenz von abstrakten Objekten, die sich nicht einmal gedanklich konstruieren lassen, ist in diesem Sinne reine Spekulation, deren Beantwortung etwas "metaphysisches" beinhaltet. Religion aber – so sagen die Intuitionisten – ist Privatsache des einzelnen Menschen und hat mit Mathematik nichts zu tun.

Beide Argumente sind an und für sich einsichtig, vertragen einander aber nicht. Für das logische Schließen über Algorithmen, deren Natur ja gerade die Konstruktion ist, scheint jedoch die intuitionistische angemessener zu sein, auch wenn wir dadurch auf einen Teil der Eleganz der klassischen Mathematik verzichten. Im Zusammenhang mit der Besprechung der Typentheorie in Kapitel 3 werden wir allerdings auch zeigen, wie wir beide Konzepte in einem Formalismus gleichzeitig behandeln können. Wir können in der intuitionistischen Typentheorie klassisch schließen, müssen dies aber explizit sagen.

2.2.3 Analytische Sequenzenbeweise

Formale Kalküle sollen, wie in Abschnitt 2.1.6 erwähnt, durch syntaktische Manipulation logisch korrekte Schlußfolgerungen simulieren. Ein Kalkül für die Prädikatenlogik muß also Formeln so in andere Formeln umwandeln, daß der Semantik aus Definition 2.2.8 Rechnung getragen wird. Dabei sollte der Kalkül natürlich unabhängig von der konkreten Interpretation sein, denn er muß ja alleine auf Formeln operieren.

In diesem Sinne sind logisch korrekte Schlüsse solche, die wahre Aussagen wieder in wahre Aussagen überführen. Für den Kalkül bedeutet dies, daß Formeln, die für jede Interpretation wahr sind, wieder in "allgemeingültige" Formeln umgewandelt werden müssen. Um dies zu präzisieren, wollen wir ein paar Begriffe prägen.

Definition 2.2.10 (Modelle und Gültigkeit)

Es sei A eine beliebige prädikatenlogische Formel.

1. *Eine Interpretation $(\iota,\mathcal{U})$ mit $\iota(A)$ = wahr heißt Modell von A*
2. *A heißt*
   - *erfüllbar, wenn es ein Modell für A gibt,*
   - *widerlegbar, wenn es ein Modell für $\neg A$ gibt,*
   - *widersprüchlich oder unerfüllbar, wenn es für A kein Modell gibt,*
   - *(allgemein)gültig, wenn jede Interpretation ein Modell für A ist.*

Wir wollen diese Begriffe an einigen Beispielen erläutern.

Beispiel 2.2.11

1. Die Formel $(\leq(4,+(3,1)) \Rightarrow \leq(+(3,1),4)) \Rightarrow \leq(+(3,1),4)$ ist erfüllbar aber nicht gültig.

   Interpretieren wir nämlich $\leq$ durch die übliche "kleiner-gleich" Relation, + durch die Addition und die Zahlsymbole wie gewöhnlich, dann erhalten wir eine wahre Aussage. Interpretieren wir dagegen $\leq$ durch die Relation "kleiner", so wird die Aussage falsch. Denn die innere Implikation wird wahr (aus $4 < 4$ folgt $4 < 4$), aber die rechte Seite der äußeren Implikation ($4 < 4$) ist falsch.

2. Die Formel $\leq(4,+(3,1)) \wedge \neg\leq(4,+(3,1))$ ist unerfüllbar, da die beiden Aussagen der Konjunktion einander widersprechen und nie gleichzeitig wahr sein können.

3. Die Formel $(\leq(4,+(3,1)) \wedge \leq(+(3,1),4)) \Rightarrow \leq(+(3,1),4)$ ist allgemeingültig.

   Nach der Semantik der Konjunktion in Definition 2.2.8 gilt nämlich für jede Interpretation ι, daß $\iota(\leq(+(3,1),4))$ = wahr ist, wenn $\iota(\leq(4,+(3,1)) \wedge \leq(+(3,1),4))$ = wahr ist. Damit ist auch die Voraussetzung für die Wahrheit der gesamten Implikation erfüllt.

Gültige Formeln spielen bei der Simulation logisch korrekter Folgerungen eine Schlüsselrolle. Beim Entwurf der Inferenzregeln eines Kalküls muß man sicherstellen, daß aus gültigen Formeln wieder gültige Formeln hergeleitet werden. Dieser Aspekt wird bei der Begründung der Regeln von besonderer Bedeutung sein.

Wir wollen im folgenden nun einen analytischen Sequenzenkalkül (vgl. Abschnitt 2.1.6.3, Seite 20 ff) für die Prädikatenlogik entwickeln. Dabei werden wir auf eine computergerechte Formulierung der Inferenzregeln achten, damit diese Regeln auch in einem Beweiseditor wie dem NuPRL System Verwendung finden können. Die wesentliche Änderung betrifft dabei die Menge Γ der Annahmen in einer Sequenz $\Gamma \vdash C$. Sie wird nun als Liste dargestellt und erlaubt somit, auf einzelne Annahmen über deren Position in der Liste zuzugreifen. Wir wollen als erstes den Begriff der Sequenzen und Beweise definieren.

Definition 2.2.12 (Sequenzen, Regeln und Beweise)

1. *Eine Deklaration hat die Gestalt $x{:}T$, wobei $x \in \mathcal{V}$ eine Variable und $T \in \mathcal{T}$ ein Bereichsbezeichner ist.*
2. *Eine Sequenz hat die Gestalt $\Gamma \vdash C$, wobei Γ – die Hypothesenliste – eine Liste von durch Komma getrennten Formeln oder Deklarationen und C – die Konklusion – eine Formel ist.*
3. *Eine (analytische) Inferenzregel ist eine Abbildung, welche eine Sequenz – das Beweisziel – in eine endliche Liste von Sequenzen – die Unter- oder Teilziele – abbildet.*
4. *Ein Beweis ist ein Baum, dessen Knoten eine Sequenz und eine Regel enthalten. Dabei müssen die Sequenzen der Nachfolger eines Knotens genau die Teilziele sein, welche sich durch Anwendung der Regel auf die Knotensequenz ergeben.*

   In einem unvollständigen Beweis darf es vorkommen, daß die Blätter nur eine Sequenz, aber keine Regel enthalten. Ein Beweis ist vollständig, wenn seine Blätter Regeln enthalten, welche bei Anwendung auf die zugehörigekn Sequenzen keine neuen Teilziele erzeugen.
5. *Eine Formel C ist ein Theorem, wenn es einen vollständigen Beweis gibt, dessen Wurzel die Sequenz '$\vdash C$' – also eine Sequenz ohne Hypothesen – enthält.*

Semantisch besehen ist eine Sequenz $A_1,\ldots,A_n \vdash C$ keine Formel, sondern ein *Urteil* über eine logische Folgerung. Üblicherweise liest man diese Sequenz als

Aus den Annahmen $A_1,\ldots,A_n$ folgt die Konklusion C.

Die folgende Definition der Semantik einer Sequenz entspricht dieser Lesart.

Definition 2.2.13 (Interpretation von Sequenzen)

Eine Interpretationsfunktion ι wird auf Sequenzen wie folgt fortgesetzt

$$\iota(A_1,\ldots,A_n \vdash C) = \begin{cases} \mathsf{wahr} & \text{falls aus } \iota(A_1) = \mathsf{wahr} \text{ und } \ldots \iota(A_n) = \mathsf{wahr} \text{ immer } \iota(C) = \mathsf{wahr} \text{ folgt} \\ \mathsf{falsch} & \text{sonst} \end{cases}$$

Dabei gilt – der Einfachheit halber – $\iota(x{:}T)$ = wahr für jede Deklaration $x{:}T$.[27]

Zu beachten ist, daß diese Definition der Rolle einer Sequenz als Formalisierung von logischen Urteilen nicht ganz gerecht wird, da genaugenommen ein Wahrheitsbegriff auf Urteile nicht anwendbar ist. Wir haben in Definition 2.2.13 implizit das sogenannte *Deduktionstheorem* angewandt, welches besagt, daß ein logischer Schluß von den Annahmen $A_1,\ldots,A_n$ auf eine Konklusion C genau dann korrekt ist, wenn die Formel $A_1 \wedge \ldots \wedge A_n \Rightarrow C$ gültig ist. Syntaktisch betrachtet spielt innerhalb eines Kalküls der Unterschied zwischen einem Urteil und einer Implikation keine besondere Rolle.

Inferenzregeln werden üblicherweise nicht als Funktionen beschrieben, sondern als Regel*schemata*, welche die Syntax des Beweiszieles und der entstehenden Teilziele mit Hilfe syntaktischer Metavariablen angeben. Die Regel ist auf eine gegebene Sequenz anwendbar, wenn sich dieses aus dem schematischen Beweisziel durch Einsetzen konkreter Formeln anstelle der syntaktischen Metavariablen ergibt. Die neuen Unterziele ergeben sich entsprechend aus den schematischen Teilzielen. Neben einigen strukturellen Regeln – wie zum Beispiel der Regel, daß eine Konklusion C gilt, wenn sie in der Hypothesenliste auftaucht – gibt es im Sequenzenkalkül für jedes logische Symbol zwei Arten von Regelschemata.

[27] Deklarationen haben im Rahmen der Prädikatenlogik vorerst keine semantische Bedeutung. Diese wird erst im Zusammenhang mit der Diskussion der Typentheorie, beginnend ab Abschnitt 2.4 deutlich werden. Bis dahin sind sie nicht mehr als ein syntaktischer Kontrollmechanismus für die Namensgebung von Variablen.

- Einführungsregeln beantworten die Frage "*was muß man zeigen, um eine Konklusion C zu beweisen?*" und haben üblicherweise die Gestalt

  $\Gamma \vdash C$ **by** *symbolname*`_i` *parameter*
  $\quad\Gamma,\ \ldots \vdash C_1$
  $\quad\quad\vdots$
  $\quad\Gamma,\ \ldots \vdash C_n$

  symbolname`_i` ist der Name der Regel. Er ist zum Beispiel anzugeben, wenn man die Regel mit Hilfe eines Beweiseditors ausführen möchte. *parameter* beschreibt eine (meist leere) Liste von Kontrollparametern, die in manchen Fällen – wie bei dem Existenzquantor – erforderlich sind, um die Regel ausführen zu können. $\Gamma \vdash C$ ist das Beweisziel und darunter stehen eingerückt die sich ergebenden Teilziele. In manchen Fällen – wie bei der Implikation – werden dabei Annahmen in die Hypothesenliste verlagert.

- Eliminationsregeln beantwortet die Frage "*welche Annahmen folgen aus einer gegebenen Formel A in der Hypothesenliste?*" und haben üblicherweise die Gestalt

  $\Gamma,\ A,\ \Delta \vdash C$ **by** *symbolname*`_e` i *parameter*
  $\quad\Gamma,\ \ldots,\ \Delta \vdash C_1$
  $\quad\quad\vdots$
  $\quad\Gamma,\ \ldots,\ \Delta \vdash C_n$

  Dabei ist *symbolname*`_e` wieder der Name der auszuführenden Regel, ggf. gesteuert durch einige Parameter. Da es sich bei den Hypothesen um Listen handelt, können wir die zu eliminierende Formel durch die Angabe ihrer Position i in dieser Liste bestimmen.[28]

Analytische Einführungsregeln werden also dazu benutzt, um die Konklusion in Teilformeln zu zerlegen während man mit Eliminationsregeln eine bestehende Annahme zerlegt, um neue Annahmen zu generieren.

2.2.4 Strukturelle Regeln

Zusätzlich zu den sogenannten "logischen" Regeln, welche die Semantik einzelner logischer Symbole charakterisieren, benötigt der Sequenzenkalkül eine Regel, welche den Zusammenhang zwischen den Hypothesen einer Sequenz und ihrer Konklusion beschreibt. Diese Regel besagt, daß eine Sequenz $\Gamma \vdash C$ gültig ist, wann immer die Konklusion C in den Annahmen auftaucht. Diese Regel, deren Rechtfertigung sich aus dem oben erwähnten Deduktionstheorem ableitet, ist praktisch das Axiom des Sequenzenkalküls. Formal drückt man dies so aus, daß die Hypothesenliste sich aufspalten läßt in drei Teile Γ,C,Δ, wobei Γ und Δ beliebig lange (auch leere!) Hypothesenlisten sind. Ist C nun die i-te Hypothese in der gesamten Liste (d.h. Γ hat i-1 Elemente) und die Konklusion der Sequenz, dann können wir sagen, daß die Konklusion aus der i-ten Hypothese folgt. Als formale Inferenzregel schreibt sich dies wie folgt:

$\Gamma,C,\Delta \vdash C$ **by** `hypothesis` i

Die *Hypothesenregel* `hypothesis` ist die einzige strukturelle Regel, die für den Sequenzenkalkül unbedingt erforderlich ist. Auf die folgenden beiden Regeln – *Schnitt* (`cut`) und *Ausdünnung* (`thin`) – könnte man im Prinzip auch verzichten. Sie tragen jedoch dazu bei, Beweise besser zu strukturieren und durch Entfernen nicht mehr benötigter Annahmen lesbarer zu machen.

Die *Schnittregel* besagt, daß man Beweise durch Einfügen von Zwischenbehauptungen zerlegen darf. Dies entspricht der Methode, mathematische Beweise durch Lemmata zu strukturieren und übersichtlicher zu gestalten. Man stellt dazu eine Formel A auf, beweist deren Gültigkeit und darf diese dann innerhalb der Annahmen zum Beweis des eigentlichen Beweiszieles weiterverwenden:

$\Gamma,\Delta \vdash C$ **by** `cut` i A
$\quad\Gamma,\Delta \vdash A$
$\quad\Gamma,A,\Delta \vdash C$

Dabei ist i die neue Position der Formel A in der Hypothesenliste.

[28] Deklarationen können im Rahmen der Prädikatenlogik nicht eliminiert werden. Sie werden nur benötigt, um die Variablennamen in Hypothesen und Konklusion zu verwalten und tauchen entsprechend nur über Einführung bei Allquantoren und Elimination von Existenzquantoren auf, wo sie die sogenannte "*Eigenvariablenbedingung*" von Genzten [Gentzen, 1935] ersetzen.

Die *Ausdünnungsregel* ermöglicht es, überflüssige Annahmen aus der Hypothesenliste zu entfernen. Dies ist besonders dann sinnvoll, wenn die Liste der Annahmen in großen Beweisen unüberschaubar zu werden droht und das zu beweisende Teilziel mit manchen Hypothesen gar nichts zu tun hat. Zum Ausdünnen gibt man einfach die Position i der zu entfernende Annahme an.[29]

$\Gamma,A,\Delta \vdash C$ **by** `thin` i
$\quad\Gamma,\Delta \vdash C$

Die Ausdünnungsregel ist jedoch mit Sorgfalt einzusetzen. Unterziele könnten unbeweisbar werden, wenn man unvorsichtigerweise Annahmen herausnimmt, die doch noch relevant sind.

2.2.5 Aussagenlogische Regeln

Konjunktion

Eine Konjunktion $A \wedge B$ ist genau dann wahr, wenn A und B wahr sind. Um also $A \wedge B$ zu beweisen, müssen wir A und B separat zeigen können. Die entsprechende Regel des Sequenzenkalküls lautet:

$\Gamma \vdash A \wedge B$ **by** `and_i`
$\quad\Gamma \vdash A$
$\quad\Gamma \vdash B$

Die Anwendung der Einführungsregel `and_i` für die Konjunktion auf die Sequenz $\Gamma \vdash A \wedge B$ läßt uns also als noch zu beweisende Unterziele zwei Sequenzen, nämlich $\Gamma \vdash A$ und $\Gamma \vdash B$.

Umgekehrt können wir aus einer Annahme $A \wedge B$ folgern, daß sowohl A als auch B gültige Annahmen sind. Wenn i die Position der Formel $A \wedge B$ in der Hypothesenliste kennzeichnet, dann können wir diese Schlußfolgerung durch Ausführung der Eliminationsregel "`and_e` i" simulieren.

$\Gamma,A \wedge B,\Delta \vdash C$ **by** `and_e` i
$\quad\Gamma,A,B,\Delta \vdash C$

Beispiel 2.2.14

Durch Anwendung der Regeln für die Konjunktion läßt sich sehr leicht zeigen, daß die Konjunktion kommutativ ist. Zu beweisen ist hierbei die Sequenz

$A \wedge B \vdash B \wedge A$

wobei A und B Platzhalter für beliebige Formeln sind (d.h. das Beweismuster ist für alle Konjunktionen gleich[30]). Im ersten Schritt eliminieren wir die Annahme $A \wedge B$, welche die erste und einzige Formel in unserer Hypothesenliste ist. Nach Anwendung der Regel `and_e 1` erhalten wir

$A,\ B \vdash B \wedge A$

Nun zerlegen wir das Beweisziel durch die Einführungsregel `and_i` für die Konjunktion und erhalten zwei Teilziele

$A,\ B \vdash B$ und $A,\ B \vdash A$

Beide Teilziele lassen sich durch Anwendung von `hypothesis 2` bzw. `hypothesis 1` vollständig beweisen. Der Beweis, den wir in Abbildung 2.4[31] zusammenfassend dargestellt haben, ist damit abgeschlossen.

[29] Deklarationen von Variablen, die noch verwendet werden, können nicht ausgedünnt werden. Die entsprechende Regel des NuPRL Systems würde bei einem derartigen Versuch eine Fehlermeldung auslösen.

[30] Wir könnten daher das Kommutativgesetz als eine "kombinierte" Regel hinzufügen, die im Endeffekt dasselbe tut, wie die hier vorgestellten vier Beweisschritte. Mit dem Taktik-Konzept, das wir aber erst in Kapitel 4 besprechen werden, können wir derartige Ergänzungen sogar als konservative Erweiterung des Regelsystems durchführen.

[31] Jede Zeile repräsentiert einen Bestandteil eines Knotens im Beweisbaum, also die Sequenz oder die Beweisregel. Das Resultat der Anwendung einer Beweisregel erscheint eingerückt unter dieser Regel und ist mit ihr graphisch verbunden. Das Maß der Einrückung kennzeichnet auch die Tiefe des Knotens im Beweisbaum.

```
A∧B  ⊢  B∧A
by and_e 1
 \
  A, B     ⊢  B∧A
  by and_i
  |\
  | A, B ⊢  B
  | by hypothesis 2
   \
    A, B ⊢  A
    by hypothesis 1
```

Abbildung 2.4: Sequenzenbeweis für die Kommutativität der Konjunktion

Disjunktion

Eine Disjunktion $A \vee B$ ist wahr, wenn A oder B wahr sind. Um also $A \vee B$ zu beweisen, muß man entweder A beweisen oder B beweisen können. Die Entscheidung, welches der beiden Ziele gezeigt werden soll, muß durch die Wahl einer der beiden Einführungsregeln getroffen werden.

$$\begin{array}{ll} \Gamma \vdash A \vee B \quad \textbf{by}\ \texttt{or\_i1} & \Gamma \vdash A \vee B \quad \textbf{by}\ \texttt{or\_i2} \\ \quad\quad \Gamma \vdash A & \quad\quad \Gamma \vdash B \end{array}$$

Umgekehrt folgt eine Konklusion C aus $A \vee B$ wenn sie aus A folgt und aus B folgt. Die Elimination einer Disjunktion entspricht also einer Fallunterscheidung.

$$\begin{array}{l} \Gamma, A \vee B, \Delta \vdash C \quad \textbf{by}\ \texttt{or\_e}\ i \\ \quad\quad \Gamma, A, \Delta \vdash C \\ \quad\quad \Gamma, B, \Delta \vdash C \end{array}$$

Beispiel 2.2.15

Hier ist ein Beweis für ein Distributivgesetz zwischen Konjunktion und Disjunktion:

```
A ∧ (B∨C) ⊢ (A∧B) ∨ (A∧C)
by and_e 1
 \
  A, (B∨C) ⊢ (A∧B) ∨ (A∧C)
  by or_e 2
  |\
  | A, B ⊢ (A∧B) ∨ (A∧C)
  | by or_i1
  |  \
  |   A, B ⊢ A∧B
  |   by and_i
  |   |\
  |   | A, B ⊢ A
  |   | by hypothesis 1
  |    \
  |     A, B ⊢ B
  |     by hypothesis 2
   \
    A, C ⊢ (A∧B) ∨ (A∧C)
    by or_i2
     \
      A, C ⊢ A∧C
      by and_i
      |\
      | A, C ⊢ A
      | by hypothesis 1
       \
        A, C ⊢ C
        by hypothesis 2
```


Implikation

Eine Implikation $A \Rightarrow B$ ist wahr, wenn B aus der Annahme A folgt. Um also $A \Rightarrow B$ zu beweisen, muß man A in die Annahmen verlagern und auf dieser Basis dann B zeigen. Umgekehrt folgt eine Konklusion C aus $A \Rightarrow B$, wenn sie aus B folgt und man A beweisen kann (man beachte, daß dies keine "genau dann, wenn" Beziehung ist).

$$\begin{array}{ll} \Gamma \vdash A \Rightarrow B \quad \textbf{by}\ \texttt{imp\_i} & \Gamma, A \Rightarrow B, \Delta \vdash C \quad \textbf{by}\ \texttt{imp\_e}\ i \\ \quad\quad \Gamma, A \vdash B & \quad\quad \Gamma, A \Rightarrow B, \Delta \vdash A \\ & \quad\quad \Gamma, \Delta, B \vdash C \end{array}$$

Die Eliminationsregel entspricht dem modus ponens "*aus A und $A \Rightarrow B$ folgt B*", ist aber kaum noch als dieser wiederzuerkennen. Liest man sie aber wie folgt

> *Wenn A unter der Annahme $A \Rightarrow B$ sowie C unter der Annahme B folgt und $A \Rightarrow B$ gilt, dann folgt C.*

so wird klar, warum sie korrekt ist. Die Annahme $A \Rightarrow B$ muß im ersten Unterziel erhalten bleiben, da sie möglicherweise im Beweis noch benötigt wird.[32] Im zweiten Teilziel ist sie dagegen redundant und kann daher fallengelassen werden.

Beispiel 2.2.16

Hier ist ein Beweis für einen wichtigen Zusammenhang zwischen Implikation und Konjunktion:

```
A∧B ⇒ C  ⊢ A ⇒ B ⇒ C
by imp_i
 \
  A∧B ⇒ C, A  ⊢ B ⇒ C
  by imp_i
   \
    A∧B ⇒ C, A, B  ⊢ C
    by imp_e 1
    |\
    | A∧B ⇒ C, A, B  ⊢ A∧B
    | by and_i
    | |\
    | | A∧B ⇒ C, A, B  ⊢ A
    | | by hypothesis 2
    |  \
    |   A∧B ⇒ C, A, B  ⊢ B
    |   by hypothesis 3
     \
      A, B, C  ⊢ C
      by hypothesis 3
```

Man beachte hierbei die Prioritätsregeln ungeklammerter Ausdrücke: $\wedge$ bindet stärker als $\Rightarrow$ und die Implikation ist rechtsassoziativ zu lesen.

Negation

Eine Negation $\neg A$ ist wahr, wenn A falsch ist. Was könnte einfacher und zugleich problematischer sein als Falschheit? In Anlehnung an Prawitz [Prawitz, 1965] stellen wir Falschheit als eine separate atomare Formel Λ dar und betrachten $\neg A$ als Abkürzung für $A \Rightarrow \Lambda$. Das Symbol Λ drückt den Gedanken des *Widerspruchs* aus und wird separat diskutiert. Diese Vorgehensweise hat den Vorteil, daß sie sowohl für klassische als auch für intuitionistische Logik eine einheitliche korrekte Sichtweise der Negation ermöglicht. Die Regeln der Negation ergeben sich unmittelbar aus denen der Implikation.

$$\begin{array}{ll} \Gamma \vdash \neg A \quad \textbf{by}\ \texttt{not\_i} & \Gamma, \neg A, \Delta \vdash C \quad \textbf{by}\ \texttt{not\_e}\ i \\ \quad\quad \Gamma, A \vdash \Lambda & \quad\quad \Gamma, \neg A, \Delta \vdash A \\ & \quad\quad \Gamma, \Delta, \Lambda \vdash C \end{array}$$

[32] Unter den Annahmen könnte sich z.B. bereits die Annahme $(A \Rightarrow B) \Rightarrow A$ befinden. Dann folgt A aus dieser Annahme und der verbliebenen Implikation.

2.2.6 Quantoren, Variablenbindung und Substitution

Die bisherigen Inferenzregeln waren verhältnismäßig leicht zu erklären, da sie sich nur auf aussagenlogische Junktoren bezogen. Mit den Quantoren $\forall$ und $\exists$ wird jedoch eine größere Dimension des logischen Schließens eröffnet und eine Reihe von Themen treten auf, die bisher nicht behandelt werden mußten: *Substitution*, *freies* und *gebundenes Vorkommen* von Variablen. Die Aussagenlogik – also die Logik der Symbole $\wedge$, $\vee$, $\Rightarrow$ und $\neg$ – ist *entscheidbar*, aber für die Prädikatenlogik gilt dies nicht mehr.

Bei der Simulation der logischen Gesetze für Quantoren spielt die Ersetzung von Variablen einer Formel durch Terme – die sogenannte *Substitution* – eine Schlüsselrolle: eine Formel $\forall x{:}T.A$ gilt als bewiesen, wenn man in A die Variable x durch einen beliebigen Term ersetzen und dann beweisen kann. Das Gleiche gilt für $\exists x{:}T.A$, wenn man *einen* solchen Term findet.[34]

Um präzise zu definieren, was Substitution genau bedeutet, führen wir zunächst ein paar Begriffe ein, welche klarstellen, ob eine Variable in einer Formel substituiert werden darf oder nicht. Wir nennen ein Vorkommen einer Variablen x innerhalb einer Formel der Gestalt $\forall x{:}T.A$ bzw. $\exists x{:}T.A$ *gebunden*, denn in diesen Formeln ist der Name x nur ein Platzhalter. Die Formel würde sich nicht ändern, wenn man jedes Vorkommen von x durch eine andere Variable y ersetzt. In der Teilformel A dagegen tritt x *frei* auf, denn hier würde eine Umbenennung die Bedeutung der gesamten Formel ändern.

Definition 2.2.18 (Freies und gebundenes Vorkommen von Variablen)

Es sei $x,y \in \mathcal{V}$ Variablen, $f \in \mathcal{F}$ ein Funktionssymbol, $P \in \mathcal{P}$ ein Prädikatssymbol, $t_1,\ldots,t_n$ Terme und A,B Formeln. Das <u>freie</u> und <u>gebundene</u> Vorkommen der Variablen x in einem Ausdruck ist induktiv durch die folgenden Bedingungen definiert.

1. *In dem Term x kommt x frei vor. Die Variable y kommt überhaupt nicht vor, wenn x und y verschieden sind.*
2. *In $f(t_1,...,t_n)$ bleibt jedes freie Vorkommen von x in einem Term t_i frei und jedes gebundene Vorkommen von x in einem Term t_i gebunden.*
3. *In Λ kommt x nicht vor.*
4. *In t_1=t_2 bleibt jedes freie Vorkommen von x in einem Term t_i frei und jedes gebundene Vorkommen von x in einem Term t_i gebunden.*
5. *In $P(t_1,...,t_n)$ bleibt jedes freie Vorkommen von x in einem Term t_i frei und jedes gebundene Vorkommen von x in einem Term t_i gebunden.*
6. *In $\neg A$, $A \wedge B$, $A \vee B$, $A \Rightarrow B$ und (A) bleibt jedes freie Vorkommen von x in den Formeln A und B frei und jedes gebundene Vorkommen von x in den Formeln A und B gebunden.*
7. *In $\forall x{:}T.A$ und $\exists x{:}T.A$ wird jedes freie Vorkommen von x in A gebunden. Gebundene Vorkommen von x in A bleiben gebunden. Jedes freie Vorkommen der Variablen y bleibt frei, wenn x und y verschieden sind.*

Das freie Vorkommen der Variablen $x_1,\ldots,x_n$ in einem Term t bzw. einer Formel A wird mit $\underline{t[x_1,\ldots,x_n]}$ bzw. $\underline{A[x_1,\ldots,x_n]}$ gekennzeichnet.

Eine Term bzw. eine Formel ohne freie Variablen heißt <u>geschlossen</u>.

[34] Man beachte jedoch, daß sich die Semantik des Allquantors in einem formalen System nur zum Teil wiederspiegeln läßt. In der Semantik ist die Formel $\forall x{:}T.A$ wahr, wenn die durch A dargestellte Aussage für alle möglichen Interpretationen der Variablen x wahr ist. In einem formalen Kalkül kann man dagegen nur prüfen, ob für jeden *Term* t gilt, daß die Formel A *beweisbar* ist, wenn man jedes Vorkommen der Variablen x durch t ersetzt.

Das ist nicht dasselbe, da ein Modell Werte enthalten mag, die sich nicht durch einen Term ausdrücken lassen. Wenn das Bereichssymbol T zum Beispiel als Menge der reellen Zahlen interpretiert wird, so gibt es überabzählbar viele Werte, die man als Interpretation für x einsetzen kann, aber nur abzählbar viele Terme.

Dieses Problem gilt allerdings nur in der klassischen Mathematik, welche die Existenz reeller Zahlen durch das Axiom der kleinsten oberen Schranke postuliert. In der intuitionistischen Mathematik befaßt man sich nur mit Objekten, die – zumindest gedanklich – konstruierbar sind, also mit endlichen Mitteln beschrieben werden können.

Wir haben die Definition des freien und gebundenen Vorkommens von Variablen bewußt ausführlicher gestaltet, als dies für die reine Prädikatenlogik notwendig ist. Vorerst kann eine Variable in einem Term oder einer atomaren Formel nicht gebunden vorkommen. Dies wird sich aber schon im Zusammenhang mit dem λ-Kalkül in Abschnitt 2.3 ändern, wo zusätzlich zu den Quantoren weitere Möglichkeiten entstehen, frei vorkommende Variablen zu *binden*. Wichtig ist auch, daß Variablen in einem Ausdruck sowohl frei als auch gebunden vorkommen können, wie das folgende Beispiel illustriert.

Beispiel 2.2.19

Wir wollen das Vorkommen der Variablen `x` im Term `(∀x:T. P(x) ∧ Q(x)) ∧ R(x)` analysieren

- Die Variable `x` tritt frei auf im Term `x`.
- Nach (5.) ändert sich daran nichts, wenn die Terme `P(x)`, `Q(x)` und `R(x)` gebildet werden.
- Nach (6.) ist `x` auch frei in `P(x) ∧ Q(x)`.
- In `∀x:T. P(x) ∧ Q(x)` ist `x` gemäß (7.) – von außen betrachtet – gebunden.
- Dies bleibt so auch nach der Klammerung in `(∀x:T. P(x) ∧ Q(x))`.
- In der Konjunktion `(∀x:T. P(x) ∧ Q(x)) ∧ R(x)` tritt `x` gemäß (6.) nun sowohl frei als auch gebunden auf.

Insgesamt haben wir folgende Vorkommen von `x` innerhalb des Terms `(∀x:T. P(x) ∧ Q(x)) ∧ R(x)` und seiner Teilterme.

$$\overbrace{\overbrace{(\forall \mathtt{x}{:}\mathtt{T}.\ \underbrace{\mathtt{P(x)} \wedge \mathtt{Q(x)}}_{\mathtt{x}\ \textit{frei}})}^{\mathtt{x}\ \textit{gebunden}} \wedge \underbrace{\mathtt{R(x)}}_{\mathtt{x}\ \textit{frei}}}^{\mathtt{x}\ \textit{frei und gebunden}}$$

Mit Hilfe der eben definierten Konzepte "freies" bzw. "gebundenes Vorkommen" können wir nun die Substitution einer Variablen x durch einen Term t innerhalb einer Formel A genau definieren. Dabei soll die Substitution das syntaktische Gegenstück zum Einsetzen von Werten in Teilformeln quantifizierter Formeln darstellen. Es ist leicht einzusehen, daß hierfür nur die *freien* Vorkommen von x in A zur Verfügung stehen. Die gebundenen Vorkommen von x sind nämlich, wie oben erwähnt, austauschbare Platzhalter, bei denen der tatsächliche Name keine Rolle spielt. Deshalb sollte eine Substitution von x durch t bei den beiden Formeln $\exists x{:}T.P(x)$ und $\exists y{:}T.P(y)$ denselben Effekt haben, d.h. gar keinen. Die folgende Definition der Substitution trägt diesem Gedanken Rechnung.

Definition 2.2.20 (Substitution)

- *Eine <u>Substitution</u> ist eine endliche Abbildung σ von der Menge $\mathcal{V}$ der Variablen in die Menge der Terme (d.h. $\sigma(x) \neq x$ gilt nur für endlich viele Variablen $x \in \mathcal{V}$).*
- *Gilt $\sigma(x_1)$=t_1, …, $\sigma(x_n)$=t_n, so schreiben wir kurz $\sigma = \underline{[t_1,\ldots,t_n/x_1,\ldots,x_n]}$.*[35]
- *Die Anwendung einer Substitution σ=$[t/x]$ auf einen prädikatenlogischen Ausdruck A wird mit $\underline{A[t/x]}$ bezeichnet und ist induktiv wie folgt definiert.*

  1. *$x[t/x] \ = \ t$;*
     $x[t/y] \ = \ x$, wenn x und y verschieden sind.
  2. *$f(t_1,...,t_n)[t/x] \ = \ f(t_1[t/x],...,t_n[t/x])$.*
  3. *$\Lambda[t/x] \ = \ \Lambda$.*
  4. *$(t_1$=$t_2)[t/x] \ = \ t_1[t/x]$=$t_2[t/x]$.*
  5. *$P(t_1,...,t_n)[t/x] \ = \ P(t_1[t/x],...,t_n[t/x])$.*

6.
$$
\begin{aligned}
(\neg A)[t/x] &= \neg A[t/x],\\
(A \wedge B)[t/x] &= A[t/x] \wedge B[t/x],\\
(A \vee B)[t/x] &= A[t/x] \vee B[t/x],\\
(A \Rightarrow B)[t/x] &= A[t/x] \Rightarrow B[t/x],\\
\texttt{(}A\texttt{)}[t/x] &= \texttt{(}A[t/x]\texttt{)}.
\end{aligned}
$$

7.
$$
\begin{aligned}
(\forall x{:}T.A)[t/x] &= \forall x{:}T.A & (\exists x{:}T.A)[t/x] &= \exists x{:}T.A\\
(\forall x{:}T.A)[t/y] &= (\forall z{:}T.A[z/x])[t/y] & (\exists x{:}T.A)[t/y] &= (\exists z{:}T.A[z/x])[t/y],
\end{aligned}
$$

wenn x und y verschieden sind, y frei in A vorkommt und x frei in t vorkommt. z ist eine neue Variable, die von x und y verschieden ist und weder in A noch in t vorkommt.

$$
(\forall x{:}T.A)[t/y] = \forall x{:}T.A[t/y] \qquad (\exists x{:}T.A)[t/y] = \exists x{:}T.A[t/y],
$$

wenn x und y verschieden sind und es der Fall ist, daß y nicht frei in A ist oder daß x nicht frei in t vorkommt.

Dabei sind $x, y \in \mathcal{V}$ Variablen, $f \in \mathcal{F}$ ein Funktionssymbol, $P \in \mathcal{P}$ ein Prädikatssymbol, $t, t_1, \ldots, t_n$ Terme und A, B Formeln.

- *Die Anwendung komplexerer Substitutionen auf einen Ausdruck, $PA[t_1, \ldots, t_n/x_1, \ldots, x_n]$, wird entsprechend definiert.*

In Definition 2.2.20(7.) mußten wir für die Anwendung von Subtitutionen auf quantifizierte Formeln einige Sonderfälle betrachten. Die Formel ändert sich gar nicht, wenn die zu ersetzende Variable gebunden ist. Andernfalls müssen wir vermeiden, daß eine freie Variable in den Bindungsbereich eines Quantors gerät ("*capture*"), daß also zum Beispiel der Allquantor nach dem Ersetzen eine Variable bindet, die in dem eingesetzten Term t frei war. Die folgende Substitution entspräche nämlich nicht dem Gedanken einer Ersetzung:

$$(\forall \texttt{y}{:}\mathbb{N}.\texttt{x<y})[\texttt{y}/\texttt{x}] = \forall \texttt{y}{:}\mathbb{N}.\texttt{y<y}$$

Deshalb wird zunächst eine Umbenennung der gebundenen Variablen vorgenommen zu einer Variablen, die bisher überhaupt nicht vorkam, und die Ersetzung geht in zwei Phasen vor sich

$$(\forall \texttt{y}{:}\mathbb{N}.\texttt{x<y})[\texttt{y}/\texttt{x}] = (\forall \texttt{z}{:}\mathbb{N}.\texttt{x<z})[\texttt{y}/\texttt{x}] = \forall \texttt{z}{:}\mathbb{N}.\texttt{y<z}$$

Der letzte Fall der Definition 2.2.20 beschreibt die normale Situation: ersetzt wird in den Unterformeln. Wir wollen nun die Substitution an einem Beispiel erklären.

Beispiel 2.2.21

$$
\begin{aligned}
&((\forall \texttt{y}{:}\mathbb{N}.\ \texttt{R(+(x,y))} \wedge \exists \texttt{x}{:}\mathbb{N}.\texttt{x=y}) \wedge \texttt{P(x)})[\texttt{-(y,4)}/\texttt{x}]\\
=\ &(\forall \texttt{y}{:}\mathbb{N}.\ \texttt{R(+(x,y))} \wedge \exists \texttt{x}{:}\mathbb{N}.\texttt{x=y})[\texttt{-(y,4)}/\texttt{x}] \wedge \texttt{P(x)}[\texttt{-(y,4)}/\texttt{x}]\\
=\ &(\forall \texttt{z}{:}\mathbb{N}.\ \texttt{R(+(x,z))} \wedge \exists \texttt{x}{:}\mathbb{N}.\texttt{x=z})[\texttt{-(y,4)}/\texttt{x}] \wedge \texttt{P(-(y,4))}\\
=\ &(\forall \texttt{z}{:}\mathbb{N}.\ \texttt{R(+(x,z))}[\texttt{-(y,4)}/\texttt{x}] \wedge (\exists \texttt{x}{:}\mathbb{N}.\texttt{x=z})[\texttt{-(y,4)}/\texttt{x}]) \wedge \texttt{P(-(y,4))}\\
=\ &(\forall \texttt{z}{:}\mathbb{N}.\ \texttt{R(+(-(y,4),z))} \wedge \exists \texttt{x}{:}\mathbb{N}.\texttt{x=z}) \wedge \texttt{P(-(y,4))}
\end{aligned}
$$

Mit diesen Begriffen sind wir nun in der Lage, die noch ausstehenden Regeln für die Quantoren aufzustellen und zu begründen.

[35] Leider gibt es eine Fülle von Notationen für die Substitution. Anstelle von $\sigma(x)=t$ schreiben manche $\sigma=[x\backslash t]$ andere $\sigma=\{x\backslash t\}$ oder $\sigma=\{t/x\}$. Die in diesem Skriptum verwendete Konvention $\sigma=[t/x]$ entspricht derjenigen, die in der Literatur zu konstruktiven Kalkülen und zum λ-Kalkül üblich ist. All diesen Schreibweisen ist aber gemein, daß die zu ersetzende Variable "unterhalb" des Terms steht, der sie ersetzt. Auch hat es sich eingebürgert, die Anwendung einer Substitution auf einen Ausdruck dadurch zu kennzeichnen, daß man die Substitution *hinter* den Ausdruck schreibt.

Universelle Quantifizierung

Entsprechend unserer Semantikdefinition 2.2.8 auf Seite 26 ist eine Formel $\forall x{:}T.A$ unter einer Interpretation genau dann wahr, wenn A für jeden möglichen Wert wahr ist, den wir für x einsetzen können. Da eine allgemeingültige Formel für jede beliebige Interpretation ι wahr sein muß, folgt hieraus, daß $\forall x{:}T.A$ genau dann allgemeingültig ist, wenn die Teilformel A allgemeingültig ist – unabhängig davon, ob x in A überhaupt vorkommt oder nicht. Damit genügt es, A zu zeigen, wenn man $\forall x{:}T.A$ beweisen möchte.

Für Sequenzen, welche ja auch die Abhängigkeit der Konklusion von den Annahmen beschreiben, muß man allerdings berücksichtigen, daß x in den bisherigen Annahmen bereits frei vorkommen könnte. Eine Regel

aus $\Gamma \vdash A$ *folgt* $\Gamma \vdash \forall x{:}T.A$

würde also eine *unzulässige Verallgemeinerung* darstellen, da in Γ Annahmen über das spezielle x stehen könnten, das durch den Quantor gebunden ist. Ein Schluß von $\texttt{x=4} \vdash \texttt{x=4}$ auf $\texttt{x=4} \vdash \forall \texttt{x}{:}\mathbb{N}.\texttt{x=4}$ ist sicherlich nicht akzeptabel. Die Grundlage für eine korrekte Einführungsregel für den Allquantor liefert das folgende Lemma.

Lemma 2.2.22

Wenn die Variable x in der Hypothesenliste Γ nicht frei vorkommt, dann ist $\Gamma \vdash \forall x{:}T.A$ genau dann gültig, wenn $\Gamma \vdash A$ gültig ist.

In Gentzens Originalarbeit [Gentzen, 1935] hat diese Feststellung zu einer sogenannten *Eigenvariablenbedingung* bei der Einführungsregel für den Allquantor geführt. Die Regel darf nur angewandt werden, wenn die quantifizierte Variable nicht in der Hypothesenliste vorkommt. Andernfalls ist ihre Anwendung verboten. Diese Einschränkung ist allerdings etwas zu restriktiv. Es genügt zu fordern, daß die quantifizierte Variable umbenannt werden muß, wenn sie bereits in der Hypothesenliste erscheint. Dies führt zu der folgenden Einführungsregel (*Verallgemeinerung*) für den Allquantor

$\Gamma \vdash \forall x{:}T.A$ **by** `all_i` *Umbenennung $[x'/x]$, wenn x in Γ frei vorkommt*
$\quad\Gamma, x'{:}T \vdash A[x'/x]$

Aus Gründen, die im Zusammenhang mit der Typentheorie deutlicher werden, fügt die Einführungsregel eine Deklaration einer Variablen zur Hypothesenliste hinzu. Dabei wird der Name x durch eine *neue* Variable x' ersetzt, die bisher nirgends vorkam, wenn x in Γ frei vorkommt (dies ersetzt die Eigenvariablenbedingung).[36]

Die Eliminationsregel (*Spezialisierung*) besagt, daß jeder Spezialfall $A[t/x]$ gültig ist, wenn $\forall x{:}T.A$ gültig ist. Um also eine Konklusion C unter der Annahme $\forall x{:}T.A$ zu zeigen, reicht es zu zeigen, daß C unter den bisherigen Hypothesen und der zusätzlichen Annahme $A[t/x]$ gilt. Der Term t für diese *Instantiierung* der universell quantifizierten Formel muß hierbei als Parameter der Regel angegeben werden.

$\Gamma, \forall x{:}T.A, \Delta \vdash C$ **by** `all_e` i t
$\quad\Gamma, \forall x{:}T.A, \Delta, A[t/x] \vdash C$

Die Annahme $\forall x{:}T.A$ bleibt hierbei erhalten, da sie im Beweis noch einmal benötigt werden könnte wie z.B. in einem Beweis für $\forall \texttt{x}{:}\mathbb{N}.\texttt{>(x,0)} \vdash \texttt{>(1,0)} \wedge \texttt{>(2,0)}$.

Beispiel 2.2.23

Abbildung 2.5 zeigt einen Beweis für das Distributivgesetz zwischen Allquantor und Konjunktion.

$$(\forall x{:}T.P(x) \wedge Q(x)) \quad\Rightarrow\quad (\forall x{:}T.P(x)) \wedge (\forall x{:}T.Q(x))$$

In diesem Beweis ist die Reihenfolge der Regeln für den Allquantor sehr wichtig. Der Term x kann erst in der Regel `all_e 1 x` benutzt werden, nachdem x durch `all_i` freigegeben wurde.

[36] Üblicherweise wird der Kalkül benutzt, um geschlossene Theoreme ohne freie Variablen zu beweisen. Das bedeutet, daß die Ausgangssequenz die Gestalt '$\vdash C$' hat, wobei C keine freie Variable besitzt. Im Verlauf des Beweises können freie Variablen dann nur in die Hypothesenliste geraten, wenn sie zuvor deklariert wurden. Für die Regel `all_i` reicht es daher, das Vorkommen einer Deklaration von x zu überprüfen.

```
∀x:T.P(x)∧Q(x)  ⊢  (∀x:T.P(x)) ∧ (∀x:T.Q(x))
by and_i
|\
| ∀x:T.P(x)∧Q(x)  ⊢ ∀x:T.P(x)
| by all_i
|  \
|   ∀x:T.P(x)∧Q(x), x:T  ⊢  P(x)
|   by all_e 1 x
|    \
|     ∀x:T.P(x)∧Q(x), x:T, P(x)∧Q(x)  ⊢  P(x)
|     by and_e 3
|      \
|       ∀x:T.P(x)∧Q(x), x:T, P(x), Q(x)  ⊢  P(x)
|       by hypothesis 3
 \
  ∀x:T.P(x)∧Q(x)  ⊢ ∀x:T.Q(x)
  by all_i
   \
    ∀x:T.P(x)∧Q(x), x:T  ⊢  Q(x)
    by all_e 1 x
     \
      ∀x:T.P(x)∧Q(x), x:T, P(x)∧Q(x)  ⊢  Q(x)
      by and_e 3
       \
        ∀x:T.P(x)∧Q(x), x:T, P(x), Q(x)  ⊢  Q(x)
        by hypothesis 4
```

Abbildung 2.5: Sequenzenbeweis für das Distributivgesetz zwischen Allquantor und Konjunktion

Würde man versuchen, den Beweis dadurch zu verkürzen, daß man die Annahmen $P(x)$, $Q(x)$ erzeugt, *bevor* man die Einführungsregel `all_i` verwendet, so würde folgende Situation entstehen:

```
∀x:T.P(x)∧Q(x)  ⊢  (∀x:T.P(x)) ∧ (∀x:T.Q(x))
by all_e 1 x
 \
  ∀x:T.P(x)∧Q(x), P(x)∧Q(x)  ⊢  (∀x:T.P(x)) ∧ (∀x:T.Q(x))
  by and_e 2
   \
    ∀x:T.P(x)∧Q(x), P(x), Q(x)  ⊢  (∀x:T.P(x)) ∧ (∀x:T.Q(x))
    by and_i
    |\
    | ∀x:T.P(x)∧Q(x), P(x), Q(x)  ⊢  ∀x:T.P(x)
    | by all_i
    |  \
    |   ∀x:T.P(x)∧Q(x), P(x), Q(x), x':T  ⊢  P(x')
    |   by ?????
     \
      ∀x:T.P(x)∧Q(x), P(x), Q(x)  ⊢  ∀x:T.Q(x)
      by all_i
       \
        ∀x:T.P(x)∧Q(x), P(x), Q(x), x':T  ⊢  Q(x')
        by ?????
```

Da bei der Anwendung von `all_i` die Variable x in den Annahmen frei vorkommt, muß eine Umbenennung stattfinden. Damit sind jedoch die Hypothesen $P(x)$ bzw. $Q(x)$ für den Beweis unbrauchbar.[37]

Existentielle Quantifizierung

Eine Formel $\exists x{:}T.A$ ist genau dann wahr, wenn wir für x mindestens einen Wert einsetzen können, der A wahr macht. Um also $\exists x{:}T.A$ zeigen zu können, reicht es zu zeigen, daß $A[t/x]$ für einen Term t gilt. Dieser

[37] Prinzipiell wäre es möglich, den Beweis dennoch zu Ende zu führen, da der Weg des Beweises aus Abbildung 2.5 immer noch verfolgt werden kann. Der Beweis würde nur einige unnötige Schritte enthalten. Sowohl in Gentzens Originalkalkül [Gentzen, 1935] als auch im NuPRL System ist dieser Beweisversuch jedoch völlig undurchführbar. Die Eigenvariablenbedingung von Gentzen würde die Anwendung von `all_i` verbieten. In NuPRL wäre bereits die Regel `all_e 1` x nicht erlaubt, da der Term x Variablen enthält, die noch nicht deklariert waren.

Term muß als Parameter der Einführungsregel für den Existenzquantor angegeben werden.

$\Gamma \vdash \exists x{:}T.A$ **by** `ex_i` t
$\quad\Gamma \vdash A[t/x]$

Die Eliminationsregel ist etwas komplizierter, weil hier ähnliche Probleme auftreten wie bei der Einführungsregel für den Allquantor. Da eine allgemeingültige Formel für jede beliebige Interpretation ι wahr sein muß, folgt eine Konklusion C aus $\exists x{:}T.A$ genau dann, wenn sie aus A folgt. Allerdings ist hierbei wieder zu beachten, daß die freie Variable x nicht auch in C oder Γ frei ist. Ansonsten würde $A \vdash C$ leichter zu beweisen sein als $\exists x{:}T.A \vdash C$. Das folgende Lemma ist das Analogon zu Lemma 2.2.22 auf Seite 39.

Lemma 2.2.24

Wenn die Variable x in der Konklusion C und der Hypothesenliste Γ nicht frei vorkommt, dann ist Γ, $\exists x{:}T.A \vdash C$ genau dann gültig, wenn Γ, $A \vdash C$ gültig ist.

In Gentzens Sequenzenkalkül gilt deshalb auch bei der Eliminationsregel für den Existenzquantor eine Eigenvariablenbedingung. Wir ersetzen diese wiederum durch die Forderung nach einer Variablenumbenennung.

$\Gamma,\exists x{:}T.A,\Delta \vdash C$ **by** `ex_e` i *Umbenennung $[x'/x]$, wenn x in C oder Γ frei vorkommt*
$\quad\Gamma,x'{:}T,A[x'/x],\Delta \vdash C$

Beispiel 2.2.25

Hier ist ein Beweis für die Vertauschbarkeit von Existenzquantoren.

```
∃x:T.∃y:S.P(x,y)  ⊢  ∃y:S.∃x:T.P(x,y)
by ex_e 1
 \
  x:T, ∃y:S.P(x,y)  ⊢  ∃y:S.∃x:T.P(x,y)
  by ex_e 2
   \
    x:T, y:S, P(x,y)  ⊢  ∃y:S.∃x:T.P(x,y)
    by ex_i y
     \
      x:T, y:S, P(x,y)  ⊢  ∃x:T.P(x,y)
      by ex_i x
       \
        x:T, y:S, P(x,y)  ⊢  P(x,y)
        by hypothesis 3
```

2.2.7 Gleichheit

Mit den Regel für die Quantoren und Junktoren ist der Kalkül für die Prädikatenlogik abgeschlossen. Die Behandlung von Gleichheit gehört eigentlich nicht zur Logik dazu. Dennoch ist dies eine derart fundamentale mathematische Beweistechnik, daß wir bereits jetzt auf ihre Repräsentation in formalen Kalkülen eingehen wollen.

Eine Formel t_1=t_2 ist genau dann wahr, wenn t_1 und t_2 die gleichen Objekte beschreiben. Um dies zu beweisen, müßte man eigentlich auf die konkreten Eigenschaften der Terme eingehen und erklären, wie man den "Wert" eines Terms bestimmt – also, daß zum Beispiel `+(4,5)` und `9` das gleiche Objekt kennzeichnen. Innerhalb der Prädikatenlogik ist das nicht möglich, da wir die Bedeutung von Funktions- und Prädikatssymbolen ja gerade *nicht* berücksichtigen wollen.[38] Nichtsdestotrotz gibt es einige logische Grundregeln zur Verarbeitung von Gleichheit, die von der konkreten Bedeutung eines Terms unabhängig sind.

[38] Die *Auswertung* von Termen innerhalb eines formalen Kalküls ist genau das zentrale Thema des Abschnitts 2.3. Der λ-Kalkül beschreibt, wie sogenannte *Normalformen* eines Terms – das syntaktische Gegenstück zu einem *Wert* – zu *berechnen* sind. Die Frage nach der *semantischen Gleichheit* wird damit prinzipiell durch Berechnung der Normalform und einen Test auf syntaktische Gleichheit (die Reflexivitätsregel) gelöst. Die hierbei entstehenden Probleme werden in Abschnitt 2.3 ausführlich besprochen.

Zwei Terme t_1 und t_2 sind mit Sicherheit gleich, wenn sie syntaktisch identisch sind. Dies ist das Gesetz der *Reflexivität*.

$\Gamma \vdash t$=t **by** `reflexivity`

Offensichtlich spielt bei der Gleichheit die Reihenfolge keine Rolle. t_1=t_2 ist wahr, wenn dies für t_2=t_1 gilt. Gleichheit ist also eine *symmetrische* Relation.

$\Gamma \vdash t_1$=t_2 **by** `symmetry`
  $\Gamma \vdash t_2$=t_1

Das letzte fundamentale Gesetz der Gleichheit ist *Transitivität*. Zwei Terme t_1 und t_2 sind gleich, wenn beide gleich sind mit einem dritten Term u. In der entsprechenden Inferenzregel ist u als Parameter anzugeben.

$\Gamma \vdash t_1$=t_2 **by** `transitivity` u
  $\Gamma \vdash t_1$=u
  $\Gamma \vdash u$=t_2

Diese drei Regeln besagen, auf welche Arten man – neben dem Ausrechnen – die Gleichheit zweier Terme beweisen kann. Sie entsprechen also den Einführungsregeln für die Gleichheit. Wie aber kann man Gleichheiten verwenden? Dem intuitiven Verständnis nach kann man zwei gleiche Objekte durch nichts unterscheiden. Das bedeutet, daß jede Eigenschaft, die für das eine Objekt gilt, auch für das andere gilt.[39] Dieses *Kongruenzgesetz* wird formal durch eine Substitutionsregel ausgedrückt. Um eine Formel A zu beweisen, in der ein Term t_1 vorkommt, genügt es t_1 durch einen gleichwertigen Term t_2 zu ersetzen und die entsprechend modifizierte Formel zu beweisen. Die Gleichheit t_1=t_2 muß hierzu angegeben werden.

$\Gamma \vdash A[t_1/x]$ **by** `substitution` t_1=t_2
  $\Gamma \vdash t_1$=t_2
  $\Gamma \vdash A[t_2/x]$

Man hätte diese Regel auch als echte Eliminationsregel formulieren können, bei der die Gleichheit t_1=t_2 bereits in den Annahmen steckt. Die oben angegebene Form ist aber in praktischen Anwendungen handlicher.

Beispiel 2.2.26

Die Substitutionsregel macht die Transitivitätsregel eigentlich überflüssig. Hier ist ein Beweis für eine Simulation der Transitivitätsregel.

```
t1=u ∧ u=t2  ⊢  t1=t2
by and_e 1
 \
  t1=u,  u=t2  ⊢  t1=t2
  by substitution t1=u
  |\
  | t1=u,  u=t2  ⊢  t1=u
  | by hypothesis 1
   \
    t1=u,  u=t2 ⊢  u=t2
    by hypothesis 2
```

Die Symmetrieregel läßt sich in ähnlicher Weise simulieren.

Abbildung 2.6 faßt alle Regeln des analytischen Sequenzenkalküls für die Prädikatenlogik zusammen.

2.2.8 Eigenschaften des Kalküls

Die wichtigsten Eigenschaften, die man von einem Kalkül fordern sollte, sind Korrektheit und Vollständigkeit (vgl. Definition 2.1.6 auf Seite 16), also die Eigenschaft, daß alle wahren Aussagen beweisbar sind und keine falschen. Mit Hilfe der in Definition 2.2.10 und 2.2.13 (siehe Seite 29 und 30) geprägten Begriffe lassen sich diese Forderungen nun wie folgt für die Prädikatenlogik erster Stufe präzisieren.

[39] In der Prädikatenlogik *höherer* Stufe drückt man dies so aus, daß man über alle Prädikate quantifiziert:
t_1=t_2 $\equiv$ $\forall$P. P(t_1) $\Leftrightarrow$ P(t_2).

$\Gamma, C, \Delta \vdash C$ **by** `hypothesis` i	

$\Gamma, \Delta \vdash C$ **by** `cut` i A
  $\Gamma, \Delta \vdash A$
  $\Gamma, A, \Delta \vdash C$
	$\Gamma, A, \Delta \vdash C$ **by** `thin` i
	  $\Gamma, \Delta \vdash C$

$\Gamma \vdash A \wedge B$ **by** `and_i`
  $\Gamma \vdash A$
  $\Gamma \vdash B$
	$\Gamma, A \wedge B, \Delta \vdash C$ **by** `and_e` i
	  $\Gamma, A, B, \Delta \vdash C$

$\Gamma \vdash A \vee B$ **by** `or_i1` $\Gamma \vdash A$	

$\Gamma \vdash A \vee B$ **by** `or_i2`
  $\Gamma \vdash B$
	$\Gamma, A \vee B, \Delta \vdash C$ **by** `or_e` i
	  $\Gamma, A, \Delta \vdash C$
	  $\Gamma, B, \Delta \vdash C$

$\Gamma \vdash A \Rightarrow B$ **by** `imp_i`
  $\Gamma, A \vdash B$
	$\Gamma, A \Rightarrow B, \Delta \vdash C$ **by** `imp_e` i
	  $\Gamma, A \Rightarrow B, \Delta \vdash A$
	  $\Gamma, \Delta, B \vdash C$

$\Gamma \vdash \neg A$ **by** `not_i`
  $\Gamma, A \vdash \Lambda$
	$\Gamma, \neg A, \Delta \vdash C$ **by** `not_e` i
	  $\Gamma, \neg A, \Delta \vdash A$
	  $\Gamma, \Delta, \Lambda \vdash C$
	
	$\Gamma, \Lambda, \Delta \vdash C$ **by** `false_e` i

$\Gamma \vdash \forall x{:}T.A$ **by** `all_i` *
  $\Gamma, x'{:}T \vdash A[x'/x]$
	$\Gamma, \forall x{:}T.A, \Delta \vdash C$ **by** `all_e` i t
	  $\Gamma, \forall x{:}T.A, \Delta, A[t/x] \vdash C$

$\Gamma \vdash \exists x{:}T.A$ **by** `ex_i` t
  $\Gamma \vdash A[t/x]$
	$\Gamma, \exists x{:}T.A, \Delta \vdash C$ **by** `ex_e` i **
	  $\Gamma, x'{:}T, A[x'/x], \Delta \vdash C$

$\Gamma \vdash t$=t **by** `reflexivity`	$\Gamma \vdash t_1$=t_2 **by** `symmetry` $\Gamma \vdash t_2$=t_1

$\Gamma \vdash t_1$=t_2 **by** `transitivity` u
  $\Gamma \vdash t_1$=u
  $\Gamma \vdash u$=t_2
	$\Gamma \vdash A[t_1/x]$ **by** `substitution` t_1=t_2
	  $\Gamma \vdash t_1$=t_2
	  $\Gamma \vdash A[t_2/x]$

$\Gamma \vdash C$ **by** `classical_contradiction` $\Gamma, \neg C \vdash \Lambda$	

*: *Die Umbenennung $[x'/x]$ erfolgt, wenn x in Γ frei vorkommt*

**: *Die Umbenennung $[x'/x]$ erfolgt, wenn x in C oder Γ frei vorkommt.*

Abbildung 2.6: Der analytische Sequenzenkalkül für die Prädikatenlogik erster Stufe

Definition 2.2.27

Ein Kalkül für die Prädikatenlogik heißt

- *korrekt, wenn mit seinen Inferenzregeln nur gültige Formeln ableitbar sind,*
- *vollständig, wenn jede gültige Formel in dem Kalkül ableitbar ist.*

Auf die einzelnen Regeln des analytischen Sequenzenkalküls bezogen bedeutet Korrektheit also folgendes.

Eine Regel des Sequenzenkalküls ist korrekt, wenn die Allgemeingültigkeit eines Beweisziels aus der Gültigkeit der erzeugten Teilziele folgt.

Diese Eigenschaften läßt sich für jede einzelne Regel relativ leicht zeigen. Wir wollen dies am Beispiel der Einführungsregel für die Implikation zeigen.

Lemma 2.2.28

Die Einführungsregel `imp_i` *für die Implikation ist korrekt*

Beweis: Die Regel `imp_i` hat die Gestalt $\begin{array}{l}\Gamma \vdash A \Rightarrow B \quad \textbf{by } \texttt{imp\_i} \\ \Gamma, A \vdash B\end{array}$

Zu zeigen ist, daß aus der Gültigkeit des von `imp_i` erzeugten Teilziels $\Gamma, A \vdash B$ die Allgemeingültigkeit des Beweisziels $\Gamma \vdash A \Rightarrow B$ folgt.

Es sei $\Gamma = A_1, \ldots, A_n$ eine beliebige Folge von Formeln, A, B Formeln und $\Gamma, A \vdash B$ eine allgemeingültige Sequenz. Dann ist $\iota(\Gamma, A \vdash B) = \mathsf{wahr}$ für jede Interpretationsfunktion ι.

Es sei also ι eine beliebige Interpretationsfunktion. Nach Definition 2.2.13 auf Seite 30 folgt dann, daß $\iota(B)=\mathsf{wahr}$ ist, wann immer $\iota(A_1)=\mathsf{wahr}, \ldots \iota(A_n)=\mathsf{wahr}$ und $\iota(A)=\mathsf{wahr}$ ist.

Es sei also $\iota(A_1)=\mathsf{wahr}, \ldots$ und $\iota(A_n)=\mathsf{wahr}$. Dann ist $\iota(B)=\mathsf{wahr}$, wann immer $\iota(A)=\mathsf{wahr}$ ist. Nach Definition 2.2.8 ist somit $\iota(A \Rightarrow B)=\mathsf{wahr}$. Damit gilt, daß $\iota(A \Rightarrow B)=\mathsf{wahr}$ ist, wann immer $\iota(A_1)=\mathsf{wahr}, \ldots$ und $\iota(A_n)=\mathsf{wahr}$ ist. Somit ist $\iota(\Gamma \vdash A \Rightarrow B) = \mathsf{wahr}$. Da ι beliebig gewählt war, haben wir gezeigt, daß $\Gamma \vdash A \Rightarrow B$ eine allgemeingültige Sequenz ist. Damit ist die Behauptung bewiesen. □

Die Korrektheitsbeweise für die anderen Regeln des Sequenzenkalküls verlaufen ähnlich und folgen dabei den bei der Präsentation gegebenen Begründungen.

Die Vollständigkeit des Kalküls ist nicht so leicht zu beweisen, da hierzu komplexe Induktionen über den Wahrheitswert von Formeln geführt werden müssen. Wir wollen an dieser Stelle auf den Beweis verzichten und verweisen interessierte Leser auf [Richter, 1978, Seite 132].

Insgesamt ist also der in Abbildung 2.6 zusammmengestellte Sequenzenkalkül korrekt und vollständig für die intuitionistische Prädikatenlogik erster Stufe, wenn man die Regel `classical_contradiction` wegläßt, und korrekt und vollständig für die klassische Prädikatenlogik erster Stufe, wenn man sie hinzunimmt.

2.2.9 Beweismethodik

Der Sequenzenkalkül gibt uns Regeln an die Hand, mit denen wir die Korrektheit jeder wahren prädikatenlogischen Aussage auf rein formale Weise beweisen können. Er garantiert die *Korrektheit* von Beweisen, die nach seinen Regeln aufgebaut sind. Diese sind analytisch formuliert, unterstützen also eine zielorientierte Vorgehensweise bei der Beweisführung. Beweiseditoren wie das NuPRL System ermöglichen es, Beweise schrittweise nur durch die Angabe von Regelnamen und Kontrollparametern zu konstruieren.

Der Sequenzenkalkül beinhaltet jedoch keine Methode, *wie* man Beweise für eine vorgegebene Formel führen kann. So ist man hierbei im wesentlichen immer noch auf den eigenen Einfallsreichtum angeweisen. Nichtsdestotrotz lassen sich einige Leitlinien angeben, welche das Finden eines formalen Beweises erleichtern.[40]

[40] Diese Leitlinien sind auch der Kern von Taktiken (vgl. Kapitel 4.2), welche einen Großteil der Beweise automatisch finden.

- Wende die Hypothesenregel `hypothesis` und die Widerspruchsregel `false_e` an, wann immer dies möglich ist, da sie Teilbeweise (Zweige im Beweisbaum) abschließen.
- Verwende Einführungsregeln, um die Konklusion in kleinere Teile aufzubrechen, und Eliminationsregeln, um Annahmen aufzubrechen (*Dekomposition*).
- Wenn mehr als eine aussagenlogische Regel anwendbar ist, spielt bei klassischer Logik die Reihenfolge keine Rolle. Bei intuitionistischem Schließen sollte die Eliminationsregel für die Disjunktion `or_e` *vor* der Einführungsregel `or_i` (in der man sich für einen Fall entscheidet) angewandt werden. Andernfalls mag es sein, daß unbeweisbare Teilziele erzeugt werden.
- Die Regel `ex_i` und `all_e` sollten niemals vor den Regeln `ex_e` und `all_i` angewandt werden. Wie das Beispiel 2.2.23 auf Seite 39 zeigt, könnten die ersten Regeln freie Variablen in der Konklusion bzw. den Annahmen erzeugen, welche bei der Anwendung der zweiten zu Umbenennungen führen.[41]
- Wenn die obigen Richtlinien immer noch Wahlmöglichkeiten lassen, sollte die Regel benutzt werden, welche die wenigsten Teilziele erzeugt.

Der kritische Punkt bei der Erstellung von Sequenzenbeweisen ist meist die Auswahl eines Terms t, der bei der Auflösung eines Quantors für eine gebundene Variable x substituiert werden soll. Dieser Term kann eigentlich erst dann bestimmt werden, wenn man den Beweis zu Ende geführt hat. Auf der anderen Seite kann der Beweis nicht fortgesetzt werden, solange der Term t nicht angegeben wurde. Diese Problematik macht eine automatische Konstruktion von Sequenzenbeweisen sehr schwer und führt dazu, daß beim Schließen mit Quantoren eine Interaktion zwischen Mensch und Computer notwendig ist.

In den letzten Jahrzehnten wurden jedoch eine Reihe von Beweissuchverfahren zu maschinennäheren Kalkülen wie der *Resolution* oder den *Matrixkalkülen* (*Konnektionsmethode*) entwickelt. Sie erlauben es, zunächst die Anforderungen an Terme, welche für quantifizierte Variablen eingesetzt werden müssen, zu sammeln und dann den Term durch *Unifikation* zu bestimmen. Für die klassische Logik sind auf dieser Basis eine Fülle von Theorembeweisern implementiert worden. Für die intuitionistische Logik gibt es Ansätze, diese Methoden entsprechend zu übertragen [Wallen, 1990], was sich jedoch als erheblich komplizierter erweist.

Der Nachteil dieser Techniken ist, daß die erzeugten Beweise schwer zu verstehen sind und wieder in lesbare Beweise in natürlicher Deduktion oder im Sequenzenkalkül übersetzt werden müssen. Auch hier sind bereits erste Ansätze untersucht worden, die wir im zweiten Teil dieser Veranstaltung näher ansehen wollen.

2.3 Der λ-Kalkül

Wir haben im letzten Abschnitt bereits darauf hingewiesen, daß eine der großen Schwächen der Prädikatenlogik erster Stufe darin besteht, daß es keine (direkten) Möglichkeiten gibt, Schlüsse über den Wert von Termen zu führen. Die Tatsache, daß die Terme `+(4,7)`, `+(5,6)` und `11` gleich sind, läßt sich nicht vernünftig ausdrücken, da es an Mechanismen fehlt, den Wert dieser Terme auf rein syntaktischem Wege auszurechnen.

Um derartige Berechnungen durchführen zu können, wurde seit Beginn dieses Jahrhunderts eine Vielfalt von mathematischen Modellen entwickelt. Manche davon, wie die Turingmaschinen oder die Registermaschinen, orientieren sich im wesentlichen an dem Gedanken einer maschinellen Durchführung von Berechnungen. Andere, wie die μ-rekursiven Funktionen, basieren auf einem Verständnis davon, was intuitiv berechenbar ist und wie sich berechenbare Funktionen zu neuen zusammensetzen lassen.

Unter all diesen Kalkülen ist der λ-Kalkül der einfachste, da er nur drei Mechanismen kennt: die Definition einer Funktion (*Abstraktion*), die Anwendung einer Funktion auf ein Argument (*Applikation*) und die Auswertung einer Anwendung, bei der die Definition bekannt ist (*Reduktion*). Dabei werden bei der Auswertung

[41] NuPRL würde eine Anwendung von `ex_i` bzw. `all_e` mit freien Variablen als Kontrollparameter ohnehin verweigern.

ausschließlich Terme manipuliert, ohne daß hierbei deren Bedeutung in Betracht gezogen wird (was einem Rechner ohnehin unmöglich wäre). Die Syntax und der Berechnungsmechanismus sind also extrem einfach. Dennoch läßt sich zeigen, daß man mit diesen Mechanismen alle berechenbaren Funktionen simulieren kann.

Wir wollen die Grundgedanken des λ-Kalküls an einem einfachen Beispiel erläutern.

Beispiel 2.3.1

Es ist allgemein üblich, Funktionen dadurch zu charakterisieren, daß man ihr Verhalten auf einem beliebigen Argument x beschreibt. Um also zum Beispiel eine Funktion zu definieren, welche ihr Argument zunächst verdoppelt und anschließend die Zahl *drei* hinzuaddiert, schreibt man kurz:

```
f(x) = 2*x+3
```

Diese Notation besagt, daß das Symbol `f` als Funktion zu interpretieren ist, welche jedem Argument `x` den Wert der Berechnung zuordnet, die durch den Ausdruck `2*x+3` beschrieben ist. Dabei kann für `x` jede beliebige Zahl eingesetzt werden. Um nun einen konkreten Funktionswert wie zum Beispiel `f(4)` auszurechnen, geht man so vor, daß zunächst jedes Vorkommen des Symbols `x` durch `4` ersetzt wird, wodurch sich der Ausdruck `2*4+3` ergibt. Dieser wird anschließend ausgewertet und man erhält `11` als Resultat.

Bei diesem Prozeß spielt das Symbol `f` eigentlich keine Rolle. Es dient nur als Abkürzung für eine Funktionsbeschreibung, welche besagt, daß jedem Argument `x` der Wert `2*x+3` zuzuordnen ist. Im Prinzip müßte es also auch möglich sein, ohne dieses Symbol auszukommen und die Funktion durch einen Term zu beschreiben, der genau die Abbildung `x` $\mapsto$ `2*x+3` wiederspiegelt.

Genau dies ist die Grundidee des λ-Kalküls. Funktionen lassen sich eindeutig durch einen Term beschreiben, welcher angibt, wie sich die Funktion auf einem beliebigen Argument verhält. Der Name des Arguments ist dabei nur ein Platzhalter, der – so die mathematische Sicht – zur *Abstraktion* der Funktionsbeschreibung benötigt wird. Um diese abstrakte Platzhalterrolle zu kennzeichnen, hat man ursprünglich das Symbol '\' benutzt und geschrieben

```
\x. 2*x+3
```

Dies drückt aus, daß eine Abbildung mit formalem Argument `x` definiert wird, welche als Ergebnis den Wert des Ausdrucks rechts vom Punkt liefert. Später wurde – der leichteren Bezeichnung wegen – das Symbol '\' durch den griechischen Buchstaben λ (*lambda*) ersetzt. In heutiger Notation schreibt man

```
λx. 2*x+3
```

Diese Abstraktion von Ausdrücken über formale Parameter ist eines der beiden fundamentalen Grundkonzepte des λ-Kalküls. Es wird benutzt, um Funktionen zu *definieren*. Natürlich aber will man definierte Funktionen auch auf bestimmte Eingabewerte *anwenden*. Die Notation hierfür ist denkbar einfach: man schreibt einfach das Argument hinter die Funktion und benutzt Klammern, wenn der Wunsch nach Eindeutigkeit dies erforderlich macht. Um also obige Funktion auf den Wert `4` anzuwenden schreibt man einfach:

```
(λx. 2*x+3)(4)
```

Dabei hätte die Klammerung um die `4` auch durchaus entfallen können. Diese Notation – man nennt sie *Applikation* – besagt, daß die Funktion `λx. 2*x+3` auf das Argument `4` angewandt wird. Sie sagt aber nichts darüber aus, welcher Wert bei dieser Anwendung herauskommt. Abstraktion und Applikation sind daher nichts anderes als syntaktische *Beschreibungsformen* für Operationen, die auszuführen sind.

Es hat sich gezeigt, daß durch Abstraktion und Applikation alle Terme gebildet werden können, die man zur Charakterisierung von Funktionen und ihrem Verhalten benötigt. Was bisher aber fehlt, ist die Möglichkeit, Funktionsanwendungen auch *auszuwerten*. Auch dieser Mechanismus ist naheliegend: um eine Funktionsanwendung auszuwerten, muß man einfach die Argumente anstelle der Platzhalter einsetzen. Man berechnet den Wert einer Funktionsanwendung also durch *Reduktion* und erhält

```
2*4+3
```


Bei der Reduktion verschwindet also die Abstraktion `λx` und die Anwendung `(4)` und stattdessen wird im inneren Ausdruck der Platzhalter `4` durch das Argument `4` ersetzt. Diese Operation ist nichts anderes als eine simple Ersetzung von Symbolen durch Terme, also eine *Substitution* im Sinne von Definition 2.2.20 (siehe Seite 37). Damit ist die Auswertung von Termen ein ganz schematischer Vorgang, der sich durch eine einfache Symbolmanipulation beschreiben läßt.

`(λx. 2*x+3)(4)` $\longrightarrow$ `(2*x+3)` $[$`x`$/$`4`$]$ = `2*4+3`.

Anders als Abstraktion und Applikation ist diese durch das Symbol $\longrightarrow$ gekennzeichnete Reduktion kein Mechanismus um Terme zu bilden, sondern einer um Terme in andere Terme umzuwandeln, welche im Sinne der vorgesehenen Bedeutung den gleichen Wert haben. Reduktion ist also eine Konversionsregel im Sinne von Abschnitt 2.1.6 (Seite 15).

Durch Abstraktion, Applikation und Reduktion ist der λ-Kalkül vollständig charakterisiert. Weitere Operationen sind nicht erforderlich. So können wir uns darauf konzentrieren, präzise Definitionen für diese Konzepte zu liefern und einen Kalkül zu erstellen, der es ermöglicht, Aussagen über den *Wert* eines λ-Terms zu beweisen. Der λ-Kalkül erlaubt symbolisches Rechnen auf Termen und geht somit weit über die Möglichkeiten der Prädikatenlogik hinaus.[42]

Anders als diese gibt der λ-Kalkül jedoch eine *intensionale* Sicht auf Funktionen. λ-Terme beschreiben uns die *innere* Struktur von Funktionen, nicht aber ihr äußeres (*extensionales*) Verhalten. Im λ-Kalkül werden Funktionen als eine *Berechnungsvorschrift* angesehen. Diese erklärt, wie der Zusammenhang zwischen dem Argument einer Funktion und ihrem Resultat zu bestimmen ist, nicht aber, welche mengentheoretischen *Objekte* hinter einem Term stehen. Der λ-Kalkül ist also eine Art *Logik der Berechnung*.

Eine weitere Änderung gegenüber der Prädikatenlogik erster Stufe ist, daß im λ-Kalkül Funktionen selbst wieder Argumente von anderen Funktionen sein dürfen. Ausdrücke wie `(λf.λx. f(x))(λx. 2*x)` werden im λ-Kalkül durchaus sehr häufig benutzt (man könnte fast nichts beschreiben ohne sie), während sie in der Prädikatenlogik erster Stufe verboten sind. In diesem Sinne ist der λ-Kalkül eine *Logik höherer Ordnung*.

Aus der Berechnungsvorschrift des λ-Kalküls ergibt sich unmmittelbar auch ein logischer Kalkül zum Schließen über den Wert eines λ-Ausdrucks. Logisch betrachtet ist damit der λ-Kalkül ein *Kalkül der Gleichheit* und er bietet ein klar definiertes und einfaches Konzept, um logische Schlußfolgerungen über das Verhalten von Programmen zu ziehen. Es muß nur sichergestellt werden, daß sich die Gleichheit zweier Terme genau dann beweisen läßt, wenn die Berechnungsvorschrift bei beiden zum gleichen Ergebnis führt.

Die Semantik von λ-Ausdrücken mengentheoretisch zu beschreiben ist dagegen verhältnismäßig schwierig, da – anders als bei der Prädikatenlogik – eine konkrete mengentheoretische Semantik bei der Entwicklung des λ-Kalküls keine Rolle spielte. Zwar ist es klar, daß es sich bei λ-Ausdrücken im wesentlichen um Funktionen handeln soll, aber die zentrale Absicht war die Beschreibung der *berechenbaren* Funktionen. Eine *operationale* Semantik, also eine Vorschrift, wie man den Wert eines Ausdrucks ausrechnet, läßt sich daher leicht angeben. Was aber die berechenbaren Funktionen im Sinne der Mengentheorie genau sind, das kann man ohne eine komplexe mathematische Theorie kaum angeben. Es ist daher kein Wunder, daß die (extensionale) Semantik erst lange nach der Entwicklung des Kalküls gegeben werden konnte.

Wir werden im folgenden zuerst die Syntax von λ-Ausdrücken sowie ihre operationale Semantik beschreiben und die mengentheoretische Semantik nur am Schluß kurz skizzieren. Wir werden zeigen, daß der λ-Kalkül

[42] Aus der Sicht der Logiker kann man den λ-Kalkül als einen Mechanismus zur Erzeugung von Funktionszeichen ansehen. Der Term `λx. 2*x` ist *einer* der vielen Namen, die man für die Verdoppelungsfunktion nehmen kann. Gegenüber anderen hat er den Vorteil, daß man ihm die intendierte Bedeutung besser ansehen kann. Aus diesem Gedankengang der *Namens*abstraktion ergab sich übrigens auch das Symbol \ als Kennzeichnung, daß jetzt ein neuer Name generiert wird. Durch die Verwendung von λ-Termen kann in der Logik also das Alphabet $\mathcal{F}$ entfallen.

Der λ-Kalkül hat ansonsten sehr viel mit der Prädikatenlogik gemeinsam. Die Alphabete $\mathcal{F}$, $\mathcal{P}$ und $\mathcal{T}$ entfallen und übrig bleiben nur die Mechanismen zur Verarbeitung von Variablen. Für diese verhält sich der λ-Operator wie ein neuer Quantor, der Variablen bindet. In der Semantik kann man ihn als Mengenabstraktion interpretieren: der Term `λx.x+2` steht für die Menge $\{(x,x+2) \mid x \in \mathcal{U}\}$. Aus dieser Betrachtungsweise sind viele Definitionen wie Syntax, Substitution etc. relativ naheliegend, da sie sehr ähnlich zu denen der Prädikatenlogik sind.

tatsächlich genauso ausdruckstark ist wie jedes andere Berechenbarkeitsmodell und hierfür eine Reihe von Standardoperationen durch λ-Ausdrücke beschreiben. Die Turing-Mächtigkeit des λ-Kalküls hat natürlich auch Auswirkungen auf die Möglichkeiten einer automatischen Unterstützung des Kalkül zum Schließen über die Werte von λ-Ausdrücken. Dies werden wir am Ende dieses Abschnitts besprechen.

2.3.1 Syntax

Die Syntax von λ-Ausdrücken ist sehr leicht zu beschreiben, da es nur Variablen, Abstraktion und Applikation gibt. Stilistisch ist sie so ähnlich wie die Definition der Terme der Prädikatenlogik (Definition 2.2.2, Seite 24)

Definition 2.3.2 (λ-Terme)

Es sei $\underline{\mathcal{V}}$ ein Alphabet von Variablen(-symbolen)

Die Terme der Sprache des λ-Kalküls – kurz λ-Terme sind induktiv wie folgt definiert.
- *Jede Variable $\underline{x} \in \mathcal{V}$ ist ein λ-Term.*
- *Ist $x \in \mathcal{V}$ eine Variable und t ein beliebiger λ-Term, dann ist die λ-Abstraktion $\underline{\lambda x.t}$ ein λ-Term.*
- *Sind t und f beliebige λ-Terme, dann ist die Applikation $\underline{f\ t}$ ein λ-Term.*
- *Ist t ein beliebiger λ-Term, dann ist $\underline{(t)}$ ein λ-Term.*

Wie bei der Prädikatenlogik gibt es für die Wahl der Variablennamen im Prinzip keinerlei Einschränkungen bis auf die Tatsache, daß sie sich im Computer darstellen lassen sollten und keine reservierten Symbole wie λ enthalten. Namenskonventionen lassen sich nicht mehr so gut einhalten wie früher, da bei einem Term wie `λx.x` noch nicht feststeht, ob die Variable `x` ein einfaches Objekt oder eine Funktion beschreiben soll. Für die Verarbeitung ist dies ohnehin unerheblich. Wir wollen die Bildung von Termen gemäß der obigen Regeln nun anhand einiger Beispiele veranschaulichen.

Beispiel 2.3.3

Die folgenden Ausdrücke sind korrekte λ-Terme im Sinne von Definition 2.3.2.

`x`, `pair`, `x(x)`, `λf.λg.λx. f g (g x)`, `λf.λx.f(x)`, `(λx.x(x))(λx.x(x))`

Die Beispiele zeigen insbesondere, daß es erlaubt ist, Terme zu bilden, bei denen man Funktionen *höherer Ordnung* – also Funktionen, deren Argumente wiederum Funktionen sind, wie z.B. `f` in `f g (g x)` – assoziiert, und Terme, die *Selbstanwendung* wie in `x(x)` beschreiben. Dies macht die Beschreibung einer mengentheoretischen Semantik (siehe Abschnitt 2.3.6) natürlich sehr viel schwieriger als bei der Prädikatenlogik.

Die Definition läßt es zu, λ-Terme zu bilden, ohne daß hierzu Klammern verwendet werden müssen. Wie bei der Prädikatenlogik erhebt sich dadurch jedoch die Frage nach einer eindeutigen Decodierbarkeit ungeklammerter λ-Terme. Deshalb müssen wir wiederum Prioritäten einführen und vereinbaren, daß der "λ-Quantor" schwächer bindet als die Applikation.

Definition 2.3.4 (Konventionen zur Eindeutigkeit von λ-Termen)

Die Applikation bindet stärker als die λ-Abstraktion und ist linksassoziativ.[43] *In einem λ-Term braucht ein durch einen stärker bindenden Operator gebildeter Teilterm nicht geklammert zu werden.*

Gemäß dieser Konvention ist also der Term `f a b` gleichbedeutend mit `(f(a))(b)` und nicht etwa mit `f(a(b))`. Die Konvention, die Applikation linksassoziativ zu interpretieren, liegt darin begründet, daß hierdurch der Term `f a b` etwa dasselbe Verhalten zeigt wie die Anwendung einer Funktion `f` auf das Paar `(a,b)`. Im zweiten Fall werden die Argumente auf einmal verarbeitet, während sie im λ-Kalkül der Reihe nach abgearbeitet werden. Die Anwendung von `f` auf `a` liefert eine neue Funktion, die wiederum auf `b` angewandt wird. Die Restriktion der λ-Abstraktionen auf *einstellige* Funktionen ist also keine wirkliche Einschränkung.[44]

[43] Eine Assoziativitätsvereinbarung für die Abstraktion braucht – wie bei den logischen Quantoren – nicht getroffen zu werden. `λx.λy.t` ist gleichwertig mit `λx.(λy.t)`, da die alternative Klammersetzung `(λx.λy).t` kein syntaktisch korrekter λ-Term ist.

[44] Die Umwandlung einer Funktion f, die auf Paaren von Eingaben operiert, in eine einstellige Funktion höherer Ordnung F mit der Eigenschaft $F(x)(y) = f(x,y)$ nennt man – in Anlehnung an den Mathematiker Haskell B. Curry – *currying*. Es sei allerdings angemerkt, daß diese Technik auf den Mathematiker Schönfinkel und nicht etwa auf Curry zurückgeht.

2.3.2 Operationale Semantik: Auswertung von Termen

In Beispiel 2.3.1 haben wir bereits angedeutet, daß wir mit jedem λ-Term natürlich eine gewisse Bedeutung assoziieren. Ein Term `λx.2*x` soll eine Funktion beschreiben, die bei Eingabe eines beliebigen Argumentes dieses verdoppelt. Diese Bedeutung wird im λ-Kalkül durch eine *Berechnungsvorschrift* ausgedrückt, welche aussagt, auf welche Art der Wert eines λ-Terms zu bestimmen ist. Diese Vorschrift verwandelt den bisher bedeutungslosen λ-Kalkül in einen Berechnungsformalismus.

Es ist offensichtlich, daß eine Berechnungsregel rein syntaktischer Natur sein muß, denn Rechenmaschinen können ja nur Symbole in andere Symbole umwandeln. Im Falle des λ-Kalküls ist diese Regel sehr einfach: wird die Apllikation einer Funktion $\lambda x.t$ auf ein Argument s ausgeweetet, so wird der formale Parameter x der Funktion – also die Variablen der λ-Abstraktion – im Funktionskörper t durch das Argument s ersetzt. Der Ausdruck $(\lambda x.t)(s)$ wird also zu $t[s/x]$ *reduziert*. Um dies präzise genug zu definieren müssen wir die Definitionen der freien und gebundenen Vorkommen von Variablen (vergleiche Definition 2.2.18 auf Seite 36) und der Substitution (Definition 2.2.20 auf Seite 37) entsprechend auf den λ-Kalkül anpassen.

Definition 2.3.5 (Freies und gebundenes Vorkommen von Variablen)

Es seien $x, y \in \mathcal{V}$ Variablen sowie f und t λ-Terme. Das freie und gebundene Vorkommen der Variablen x in einem λ-Term ist induktiv durch die folgenden Bedingungen definiert.

1. *Im λ-Term x kommt x frei vor. Die Variable y kommt nicht vor, wenn x und y verschieden sind.*
2. *In $\lambda x.t$ wird jedes freie Vorkommen von x in t gebunden. Gebundene Vorkommen von x in t bleiben gebunden. Jedes freie Vorkommen der Variablen y bleibt frei, wenn x und y verschieden sind.*
3. *In $f\ t$ bleibt jedes freie Vorkommen von x in f oder t frei und jedes gebundene Vorkommen von x in f oder t gebunden.*
4. *In (t) bleibt jedes freie Vorkommen von x in t frei und jedes gebundene Vorkommen gebunden.*

Das freie Vorkommen der Variablen $x_1, \ldots, x_n$ in einem λ-Term t wird mit $\underline{t[x_1, \ldots, x_n]}$ gekennzeichnet.

Eine λ-Term ohne freie Variablen heißt geschlossen.

Man beachte, daß im Unterschied zur Prädikatenlogik Variablen jetzt auch innerhalb des "Funktionsnamens" einer Applikation $f\ t$ frei oder gebunden vorkommen sind, da f seinerseits ein komplexerer λ-Term sein kann. Das folgende Beispiel illustriert die Möglichkeiten des freien bzw. gebundenen Vorkommens von Variablen.

Beispiel 2.3.6

Wir wollen das Vorkommen der Variablen `x` im λ-Term `λf.λx.(λz. f x z) x` analysieren
- Die Variable `x` tritt frei auf im λ-Term `x`.
- Nach (3.) ändert sich daran nichts, wenn die λ-Terme `f x` und `f x z` gebildet werden.
- Nach (2.) bleibt `x` frei im λ-Term `λz. f x z` .
- Nach (3.) bleibt `x` frei im λ-Term `(λz. f x z) x` .
- In `λx.(λz. f x z) x` ist `x` gemäß (2.) – von außen betrachtet – gebunden.
- Nach (2.) bleibt `x` gebunden im gesamten Term `λf.λx.(λz. f x z) x`.

Insgesamt haben wir folgende Vorkommen von `x`: $\overbrace{\lambda \mathtt{f}.\lambda \mathtt{x}.\underbrace{(\lambda \mathtt{z}.\ \mathtt{f\ x\ z})\ \mathtt{x}}_{\mathtt{x}\ \mathit{frei}}}^{\mathtt{x}\ \mathit{gebunden}}$

Auch das Konzept der Substitution muß geringfügig geändert werden. Während in Definition 2.2.20 nur innerhalb der Argumente einer Funktionsanwendung ersetzt werden konnte, können im λ-Kalkül auch innerhalb der Funktion einer Applikation (die ebenfalls ein Term ist) Ersetzungen vorgenommen werden. Die λ-Abstraktion dagegen verhält sich wie ein Quantor. Entsprechend müssen wir wieder drei Fälle betrachten.

Definition 2.3.7 (Substitution)

Die Anwendung einer Substitution $\sigma=[t/x]$ auf einen λ-Term u – bezeichnet durch $\underline{u[t/x]}$ – ist induktiv wie folgt definiert.

$$\begin{array}{lll}
x[t/x] & = & t \\
x[t/y] & = & x, \text{ wenn } x \text{ und } y \text{ verschieden sind.} \\
(\lambda x\,.\,u)[t/x] & = & \lambda x\,.\,u \\
(\lambda x\,.\,u)[t/y] & = & (\lambda z\,.\,u[z/x])[t/y] \\
 & & \text{wenn } x \text{ und } y \text{ verschieden sind, } y \text{ frei in } u \text{ vorkommt und } x \text{ frei in } t \text{ vorkommt. } z \text{ ist eine} \\
 & & \text{neue Variable, die von } x \text{ und } y \text{ verschieden ist und weder in } u \text{ noch in } t \text{ vorkommt.} \\
(\lambda x\,.\,u)[t/y] & = & \lambda x\,.\,u[t/y] \\
 & & \text{wenn } x \text{ und } y \text{ verschieden sind und es der Fall ist, daß } y \text{ nicht frei in } u \text{ ist oder daß } x \\
 & & \text{nicht frei in } t \text{ vorkommt.} \\
(f\ u)[t/x] & = & f[t/x]\,u[t/x] \\
(u)\,[t/x] & = & (u[t/x])
\end{array}$$

Dabei sind $x, y \in \mathcal{V}$ Variablen sowie f, u und t λ-Terme.

Die Anwendung komplexerer Substitutionen auf einen λ-Term, $u[t_1,\ldots,t_n/x_1,\ldots,x_n]$, wird entsprechend definiert. Wir wollen die Substitution an einem einfachen Beispiel erklären.

Beispiel 2.3.8

$$\begin{array}{ll}
 & (\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{n\ f\ (f\ x)})\,[\lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{x}\,/\,\mathtt{n}] \\
= & \lambda \mathtt{f}.\ (\lambda \mathtt{x}.\ \mathtt{n\ f\ (f\ x)})\,[\lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{x}\,/\,\mathtt{n}] \\
= & \lambda \mathtt{f}.\lambda \mathtt{x}.\ (\mathtt{n\ f\ (f\ x)})\,[\lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{x}\,/\,\mathtt{n}] \\
= & \lambda \mathtt{f}.\lambda \mathtt{x}.\ (\lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{x})\ \mathtt{f\ (f\ x)}
\end{array}$$

Bei der Anwendung einer Substitution auf eine λ-Abstraktion müssen wir also in besonderen Fällen wiederum auf eine Umbenennung gebundener Variablen zurückgreifen, um zu vermeiden, daß eine freie Variable ungewollt in den Bindungsbereich der λ-Abstraktion gerät. Diese Umbenennung gebundener Variablen ändert die Bedeutung eines Terms überhaupt nicht. Terme, die sich nur in den Namen ihrer gebundenen Variablen unterscheiden, müssen also im Sinne der Semantik als gleich angesehen werden, auch wenn sie textlich verschieden sind. Zur besseren Unterscheidung nennt man solche Termpaare daher *kongruent*.

Definition 2.3.9 (α-Konversion)

Eine Umbenennung gebundener Variablen (oder α-Konversion) in einem λ-term t ist die Ersetzung eines Teilterms von t mit der Gestalt $\lambda x\,.\,u$ durch den Term $\lambda z\,.\,u[z/x]$, wobei z eine Variable ist, die in u bisher nicht vorkam.

Zwei λ-Terme t und u heißen kongruent oder α-konvertibel, wenn u sich aus t durch endlich viele Umbenennungen gebundener Variablen ergibt.

Umbennenung kann also als eine erste einfache Rechenvorschrift angesehen werden, welche Terme in andere Terme umwandelt (konvertiert), die syntaktisch verschieden aber gleichwertig sind. Eine wirkliche Auswertung eines Termes findet jedoch nur statt, wenn reduzierbare Teilterme durch ihre kontrahierte Form ersetzt werden.

Definition 2.3.10 (Reduktion)

Die Reduktion eines λ-terms t ist die Ersetzung eines Teiltermes von t mit der Gestalt $(\lambda x\,.\,u)(s)$ durch den Term $u[s/x]$. Der Term $(\lambda x\,.\,u)(s)$ wird dabei als Redex bezeichnet und $u[s/x]$ als sein Kontraktum.[45]

Ein λ-Terme t heißt reduzierbar auf einen λ-Term u – im Zeichen $\underline{t \xrightarrow{*} u}$ –, wenn u sich aus t durch endlich viele Reduktionen und α-Konversionen ergibt.

[45] Das Wort *Redex* steht für einen reduzierbaren Ausdruck (reducible expression) und *Kontraktum* für die zusammengezogene Form dieses Ausdrucks.

Für die Auswertung von λ-Termen durch einen Menschen würden diese beiden Definitionen vollkommen ausreichen. Da wir den Prozeß der Reduktion jedoch als Berechnungsmechanismus verstehen wollen, den eine Maschine ausführen soll, geben wir zusätzlich eine detaillierte formale Definition.

Definition 2.3.11 (Formale Definition der Reduzierbarkeit)

- *Konversion von λ-Termen ist eine binäre Relation $\cong$, die induktiv wie folgt definiert ist.*
  1. *$\lambda x\,.\,u \cong \lambda z\,.\,u[z/x]$, falls z eine Variable ist, die in u nicht vorkommt.* — *α-Konversion*
  2. *$f\ t \cong f\ u$, falls $t \cong u$* — *μ-Konversion*
  3. *$f\ t \cong g\ t$, falls $f \cong g$* — *ν-Konversion*
  4. *$\lambda x\,.\,t \cong \lambda x\,.\,u$, falls $t \cong u$* — *ξ-Konversion*
  5. *$t \cong t$* — *ρ-Konversion*
  6. *$t \cong u$, falls $u \cong t$* — *σ-Konversion*
  7. *$t \cong u$, falls es einen λ-Term s gibt mit $t \cong s$ und $s \cong u$* — *τ-Konversion*
- *Reduktion von λ-Termen ist eine binäre Relation $\longrightarrow$, die induktiv wie folgt definiert ist.*
  1. *$(\lambda x\,.\,u)\ s \longrightarrow u[s/x]$* — *$\beta$-Reduktion*
  2. *$f\ t \longrightarrow f\ u$, falls $t \longrightarrow u$*
  3. *$f\ t \longrightarrow g\ t$, falls $f \longrightarrow g$*
  4. *$\lambda x\,.\,t \longrightarrow \lambda x\,.\,u$, falls $t \longrightarrow u$*
  5. *$t \longrightarrow u$, falls es einen λ-Term s gibt mit $t \longrightarrow s$ und $s \cong u$ oder $t \cong s$ und $s \longrightarrow u$*
- *Reduzierbarkeit von λ-Termen ist eine binäre Relation $\xrightarrow{*}$, die wie folgt definiert ist*

  $t \xrightarrow{*} u$, falls $t \xrightarrow{n} u$ für ein $n \in \mathbb{N}$.

  Dabei ist die Relation $\xrightarrow{n}$ induktiv wie folgt definiert
  - *$t \xrightarrow{0} u$, falls $t \cong u$*
  - *$t \xrightarrow{1} u$, falls $t \longrightarrow u$*
  - *$t \xrightarrow{n+1} u$, falls es einen λ-Term s gibt mit $t \longrightarrow s$ und $s \xrightarrow{n} u$*

Die Regeln, die zusätzlich zur zur β-Reduktion (bzw. zur α-Konversion) genannt werden, drücken aus, daß Reduktion auf jedes Redex innerhalb eines Termes angewandt werden darf, um diesen zu verändern. Unter all diesen Regeln ist die β-Reduktion die einzige 'echte' Reduktion. Aus diesem Grunde schreibt man oft auch $\underline{t \xrightarrow{\beta} u}$ anstelle von $t \longrightarrow u$. Wir wollen nun die Reduktion an einigen Beispielen erklären.

Beispiel 2.3.12

1.
$$\begin{array}{llll}
 & (\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{n\ f\ (f\ x)})\ (\lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{x}) & \longrightarrow & (\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{n\ f\ (f\ x)})[\lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{x}\,/\,\mathtt{n}] \\
= & \lambda \mathtt{f}.\lambda \mathtt{x}.\ (\lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{x})\ \mathtt{f\ (f\ x)} & & \textit{(vergleiche Beispiel 2.3.8 auf Seite 50)} \\
 & (\lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{x})\ \mathtt{f} & \longrightarrow & \lambda \mathtt{x}.\mathtt{x} \\
\mathtt{also} & (\lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{x})\ \mathtt{f\ (f\ x)} & \longrightarrow & (\lambda \mathtt{x}.\mathtt{x})\ \mathtt{(f\ x)} \\
\mathtt{also} & \lambda \mathtt{x}.(\lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{x})\ \mathtt{f\ (f\ x)} & \longrightarrow & \lambda \mathtt{x}.(\lambda \mathtt{x}.\mathtt{x})\ \mathtt{(f\ x)} \\
\mathtt{also} & \lambda \mathtt{f}.\lambda \mathtt{x}.(\lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{x})\ \mathtt{f\ (f\ x)} & \longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.(\lambda \mathtt{x}.\mathtt{x})\ \mathtt{(f\ x)} \\
 & (\lambda \mathtt{x}.\mathtt{x})\ \mathtt{(f\ x)} & \longrightarrow & \mathtt{f\ x} \\
\mathtt{also} & \lambda \mathtt{x}.(\lambda \mathtt{x}.\mathtt{x})\ \mathtt{(f\ x)} & \longrightarrow & \lambda \mathtt{x}.\ \mathtt{f\ x} \\
\mathtt{also} & \lambda \mathtt{f}.\lambda \mathtt{x}.(\lambda \mathtt{x}.\mathtt{x})\ \mathtt{(f\ x)} & \longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f\ x}
\end{array}$$

Diese ausführliche Beschreibung zeigt alle Details einer formalen Reduktion einschließlich des Hinabsteigens in Teilterme, in denen sich die zu kontrahierenden Redizes befinden. Diese Form ist für eine

Reduktion "von Hand" jedoch zu ausführlich, da es einem Menschen nicht schwerfällt, einen zu reduzierenden Teilterm zu identifizieren und die Reduktion durchzuführen. Wir schreiben daher kurz

$$
\begin{array}{ll}
 & (\lambda\mathtt{n}.\lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{n\ f\ (f\ x)})\ (\lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{x}) \\
\longrightarrow & \lambda\mathtt{f}.\lambda\mathtt{x}.\ (\lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{x})\ \mathtt{f\ (f\ x)} \\
\longrightarrow & \lambda\mathtt{f}.\lambda\mathtt{x}.\ (\lambda\mathtt{x}.\mathtt{x})\ \mathtt{(f\ x)} \\
\longrightarrow & \lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{f\ x}
\end{array}
$$

oder noch kürzer: $(\lambda\mathtt{n}.\lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{n\ f\ (f\ x)})\ (\lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{x}) \xrightarrow{3} \lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{f\ x}$

2.
$$
\begin{array}{ll}
 & (\lambda\mathtt{n}.\lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{n\ f\ (f\ x)})\ (\lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{f\ x}) \\
\longrightarrow & \lambda\mathtt{f}.\lambda\mathtt{x}.\ (\lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{f\ x})\ \mathtt{f\ (f\ x)} \\
\longrightarrow & \lambda\mathtt{f}.\lambda\mathtt{x}.\ (\lambda\mathtt{x}.\ \mathtt{f\ x})\ \mathtt{(f\ x)} \\
\longrightarrow & \lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{f\ (f\ x)}
\end{array}
$$

3. Bei der Reduktion des Terms $(\lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{f\ x\ (f\ f)})\ (\lambda\mathtt{x}.\lambda\mathtt{y}.\ \mathtt{y})$ gibt es mehrere Möglichkeiten vorzugehen. Die vielleicht naheliegenste ist die folgende:

$$
\begin{array}{ll}
 & (\lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{f\ x\ (f\ f)})\ (\lambda\mathtt{x}.\lambda\mathtt{y}.\ \mathtt{y}) \\
\longrightarrow & \lambda\mathtt{x}.\ (\lambda\mathtt{x}.\lambda\mathtt{y}.\ \mathtt{y})\ \mathtt{x}\ ((\lambda\mathtt{x}.\lambda\mathtt{y}.\ \mathtt{y})(\lambda\mathtt{x}.\lambda\mathtt{y}.\ \mathtt{y})) \\
\longrightarrow & \lambda\mathtt{x}.\ (\lambda\mathtt{y}.\ \mathtt{y})\ ((\lambda\mathtt{x}.\lambda\mathtt{y}.\ \mathtt{y})(\lambda\mathtt{x}.\lambda\mathtt{y}.\ \mathtt{y})) \\
\longrightarrow & \lambda\mathtt{x}.\ ((\lambda\mathtt{x}.\lambda\mathtt{y}.\ \mathtt{y})(\lambda\mathtt{x}.\lambda\mathtt{y}.\ \mathtt{y})) \\
\longrightarrow & \lambda\mathtt{x}.\ \lambda\mathtt{y}.\ \mathtt{y})
\end{array}
$$

Wir hätten ab dem zweiten Schritt aber auch zunächst das rechte Redex reduzieren können und folgende Reduktionskette erhalten:

$$
\begin{array}{ll}
 & (\lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{f\ x\ (f\ f)})\ (\lambda\mathtt{x}.\lambda\mathtt{y}.\ \mathtt{y}) \\
\longrightarrow & \lambda\mathtt{x}.\ (\lambda\mathtt{x}.\lambda\mathtt{y}.\ \mathtt{y})\ \mathtt{x}\ ((\lambda\mathtt{x}.\lambda\mathtt{y}.\ \mathtt{y})(\lambda\mathtt{x}.\lambda\mathtt{y}.\ \mathtt{y})) \\
\longrightarrow & \lambda\mathtt{x}.\ (\lambda\mathtt{x}.\lambda\mathtt{y}.\ \mathtt{y})\ \mathtt{x}\ (\lambda\mathtt{y}.\ \mathtt{y})) \\
\longrightarrow & \lambda\mathtt{x}.\ (\lambda\mathtt{y}.\ \mathtt{y})(\lambda\mathtt{y}.\ \mathtt{y})) \\
\longrightarrow & \lambda\mathtt{x}.\ \lambda\mathtt{y}.\ \mathtt{y})
\end{array}
$$

Das dritte Beispiel zeigt deutlich, daß Reduktion keineswegs ein deterministischer Vorgang ist. Unter Umständen gibt es mehrere Redizes, auf die eine β-Reduktion angewandt werden kann, und damit auch mehrere Arten, den Wert eines Terms durch Reduktion auszurechnen. Um also den λ-Kalkül als eine Art Programmiersprache verwendbar zu machen, ist es zusätzlich nötig, eine *Reduktionsstrategie* zu fixieren. Diese Strategie muß beim Vorkommen mehrerer Redizes in einem Term entscheiden, welches von diesen zuerst durch ein Kontraktum ersetzt wird. Die Auswirkungen einer solchen Festlegung und weitere Reduktionseigenschaften des λ-Kalküls werden wir im Abschnitt 2.3.7 diskutieren.

2.3.3 Standard-Erweiterungen

Bisher haben wir den λ-Kalkül im wesentlichen als ein formales System betrachtet, welches geeignet ist, Terme zu manipulieren. Wir wollen nun zeigen, daß dieser einfache Formalismus tatsächlich geeignet ist, bekannte berechenbare Funktionen auszudrücken. Dabei müssen wir jedoch beachten, daß die einzige Berechnungsform die Reduktion von Funktionsanwendungen ist. Zahlen, boolesche Werte und ähnliche aus den meisten Programmiersprachen vertraute Operatoren gehören nicht zu den Grundkonstrukten des λ-Kalküls. Wir müssen sie daher im λ-Kalkül *simulieren*.[46]

Aus theoretischer Sicht bedeutet dies, den λ-Kalkül durch Einführung abkürzender Definitionen im Sinne von Abschnitt 2.1.5 *konservativ* zu erweitern. Auf diese Art behalten wir die einfache Grundstruktur des

[46] In *realen* Computersystemen ist dies nicht anders. Auch ein Computer operiert nicht auf Zahlen sondern nur auf Bitmustern, welche Zahlen *darstellen*. Alle arithmetischen Operationen, die ein Computer ausführt, sind nichts anderes als eine Simulation dieser Operationen durch entsprechende Manipulationen der Bitmuster.

λ-Kalküls, wenn es darum geht, grundsätzliche Eigenschaften dieses Kalküls zu untersuchen, erweitern aber gleichzeitig die Ausdruckskraft der vordefinierten formalen Sprache und gewinnen somit an Flexibilität.

Höhere funktionale Programmiersprachen wie LISP oder ML können in diesem Sinne als konservative Erweiterungen des λ-Kalküls betrachtet werden.[47] Sie unterscheiden sich von diesem nur durch eine große Menge vordefinierter Konstrukte, die ein einfacheres Programmieren in diesen Sprachen erst möglich machen.

Boolesche Operatoren

Im reinen λ-Kalkül ist die Funktionsanwendung (Applikation) die einzige Möglichkeit, "Programme" und "Daten" in Verbindung zu bringen. In praktischen Anwendungen besteht jedoch oft die Notwendigkeit, bestimmte Programmteile nur auszuführen, wenn eine Bedingung erfüllt ist. Um dies zu simulieren, benötigt man boolesche Werte und ein Konstrukt zur Erzeugung *bedingter Funktionsaufrufe*.

Definition 2.3.13 (Boolesche Operatoren)

$$
\begin{array}{lcl}
\mathsf{T} & \equiv & \lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{x} \\
\mathsf{F} & \equiv & \lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{y} \\
\mathbf{\mathsf{cond}}(b;s;t) & \equiv & b\,s\,t
\end{array}
$$

Die Terme **T** und **F** sollen die Wahrheitswerte *wahr* und *falsch* simulieren. Der Term $\mathsf{cond}(b;s;t)$ beschreibt ein sogenanntes *Conditional*. Es nimmt einen booleschen Ausdruck b als erstes Argument, wertet diesen aus und berechnet dann – je nachdem ob diese Auswertung **T** oder **F** ergab – den Wert von s oder den von t. Es ist nicht schwer zu beweisen, daß **T**, **F** und cond sich tatsächlich wie die booleschen Operatoren verhalten, die man hinter dem Namen vermutet.

Beispiel 2.3.14

Wir zeigen, daß $\mathsf{cond}(\mathsf{T};s;t)$ zu s reduziert und $\mathsf{cond}(\mathsf{F};s;t)$ zu t

$$
\begin{array}{lclclclcl}
\mathsf{cond}(\mathsf{T};s;t) & \equiv & \mathsf{T}\,s\,t & \equiv & (\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{x})\,s\,t & \longrightarrow & (\lambda\mathtt{y}.s)\,t & \longrightarrow & s \\
\mathsf{cond}(\mathsf{F};s;t) & \equiv & \mathsf{F}\,s\,t & \equiv & (\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{y})\,s\,t & \longrightarrow & (\lambda\mathtt{y}.\mathtt{y})\,t & \longrightarrow & t
\end{array}
$$

Damit ist das Conditional tatsächlich invers zu den booleschen Werten **T** und **F**.[48] Im Beispiel 2.3.25 (Seite 59) werden wir darüber hinaus auch noch zeigen, daß **T** und **F** auch fundamental unterschiedliche Objekte beschreiben und somit tatsächlich dem Gedanken entsprechen, daß **T** und **F** einander widersprechen.

Die bisherige Präsentationsform des Conditionals als Operator cond, der drei Eingabeparameter erwartet, entspricht immer noch der Denkweise von Funktionsdefinition und -anwendung. Sie hält sich daher an die syntaktischen Einschränkungen an den Aufbau von Termen, die uns aus der Prädikatenlogik (vergleiche Definition 2.2.2 auf Seite 24) bekannt und für einen Parser leicht zu verarbeiten ist. Für die meisten Programmierer ist diese Präsentationsform jedoch sehr schwer zu lesen, da sie mit der Schreibweise "if b then s else t" vertrauter sind. Wir ergänzen daher die Definitionen boolescher Operatoren um eine verständlichere Notation für das Conditional.

$$\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t \ \equiv\ \mathsf{cond}(b;s;t)$$

Im folgenden werden wir für die meisten Operatoren immer zwei Formen definieren: eine mathematische 'Termform' und eine leichter zu lesende "Display Form" dieses Terms.[49]

[47] Man beachte, daß für die die textliche Form der abkürzender Definitionen keinerlei Einschränkungen gelten – bis auf die Forderung, daß sie in einem festen Zeichensatz beschreibbar sind und alle syntaktischen Metavariablen ihrer rechten Seiten auch links vorkommen müssen.

[48] In der Denkweise des Sequenzenkalküls können wir **T** und **F** als Operatoren zur *Einführung* boolescher Werte betrachten und cond als Operator zur *Elimination* boolescher Werte. Diese Klassifizierung von Operatoren – man sagt dazu auch *kanonische* und *nichtkanonische* Operatoren – werden wir im Abschnitt 2.4 weiter aufgreifen.

[49] Im NuPRL System und der Typentheorie, die wir im Kapitel 3 vorstellen, wird diese Idee konsequent zu Ende geführt. Wir unterscheiden dort die sogenannte *Abstraktion* (nicht zu verwechseln mit der λ-Abstraktion), welche einen neuen Operatornamen wie cond einführt, von der *Display Form*, die beschreibt, wie ein Term textlich darzustellen ist, der diesen Operatornamen als Funktionszeichen benutzt. Auf diese Art erhalten wir eine einheitliche Syntaxbeschreibung für eine Vielfalt von Operatoren und dennoch Flexibilität in der Notation.

Paare und Projektionen

Boolesche Operationen wie das Conditional können dazu benutzt werden, 'Programme' besser zu strukturieren. Aber auch bei den 'Daten' ist es wünschenswert, Strukturierungsmöglichkeiten anzubieten. Die wichtigste dieser Strukturierungsmöglichkeiten ist die Bildung von Tupeln, die es uns erlauben, `f(a,b,c)` anstelle von `f a b c` zu schreiben. Auf diese Art wird deutlicher, daß die Werte `a`, `b` und `c` zusammengehören und nicht etwa einzeln abgearbeitet werden sollen.

Die einfachste Form von Tupeln sind Paare, die wir mit $\langle \mathtt{a},\mathtt{b}\rangle$ bezeichnen. Komplexere Tupel können durch das Zusammensetzen von Paaren gebildet werden. `(a,b,c)` läßt sich zum Beispiel als $\langle \mathtt{a},\langle \mathtt{b},\mathtt{c}\rangle\rangle$ schreiben. Neben der *Bildung* von Paaren aus einzelnen Komponenten benötigen wir natürlich auch Operatoren, die ein Paar *p analysieren*. Üblicherweise verwendet man hierzu *Projektionen*, die wir mit $p.1$ und $p.2$ bezeichnen.

Definition 2.3.15 (Operatoren auf Paaren)

$$
\begin{array}{lcl}
\mathbf{pair}(s,t) & \equiv & \lambda \mathtt{p}.\ \mathtt{p}\,s\,t \\
\mathbf{pi1}(pair) & \equiv & pair\ (\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{x}) \\
\mathbf{pi2}(pair) & \equiv & pair\ (\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y}) \\
\mathbf{spread}(pair;\, x,y.t) & \equiv & pair\ (\lambda x.\lambda y.t) \\
\langle s,t\rangle & \equiv & \mathbf{pair}(s,t) \\
pair.1 & \equiv & \mathbf{pi1}(pair) \\
pair.2 & \equiv & \mathbf{pi2}(pair) \\
\mathsf{let}\ \langle x,y\rangle = pair\ \mathsf{in}\ t & \equiv & \mathbf{spread}(pair;\, x,y.t)
\end{array}
$$

Der spread-Operator beschreibt eine *einheitliche* Möglichkeit, Paare zu analysieren: ein Paar p wird aufgespalten in zwei Komponenten, die wir mit x und y bezeichnen und in einem Term t weiter verarbeiten.[50] Die Projektionen können als Spezialfall des spread-Operators betrachtet werden, in denen der Term t entweder x oder y ist.

In seiner ausführlicheren Notation $\mathsf{let}\ \langle x,y\rangle = p\ \mathsf{in}\ t$ wird dieses Konstrukt zum Beispiel in der Sprache ML benutzt. Bei seiner Ausführung wird der Term p ausgewertet, bis seine beiden Komponenten feststehen, und diese dann anstelle der Variablen x und y im Term t eingesetzt. Das folgende Beispiel zeigt, daß genau dieser Effekt durch die obige Definition erreicht wird.

Beispiel 2.3.16

$$
\begin{array}{lcl}
\mathsf{let}\ \langle x,y\rangle = \langle u,v\rangle\ \mathsf{in}\ t & \equiv & \langle u,v\rangle(\lambda x.\lambda y.t) \ \equiv\ (\lambda \mathtt{p}.\ \mathtt{p}\,u\,v)(\lambda x.\lambda y.t) \\
& \longrightarrow & (\lambda x.\lambda y.t)\,u\,v \\
& \longrightarrow & (\lambda y.t[u/x])\,u\,v \\
& \longrightarrow & t[u,v/x,y])
\end{array}
$$

Auf ähnliche Weise kann man zeigen $\langle u,v\rangle.1 \longrightarrow u$ und $\langle u,v\rangle.2 \longrightarrow v$.

Damit ist der spread-Operator also tatsächlich invers zur Paarbildung.

Natürliche Zahlen

Es gibt viele Möglichkeiten, natürliche Zahlen und Operationen auf Zahlen im λ-Kalkül darzustellen. Die bekannteste dieser Repräsentationen wurde vom Mathematiker Alonzo Church entwickelt. Man nennt die entsprechenden λ-Terme daher auch *Church Numerals*. In dieser Repräsentation codiert man eine natürliche Zahl n durch einen λ-Term, der zwei Argumente `f` und `x` benötigt und das erste insgesamt n-mal auf das zweite anwendet. Wir bezeichnen diese Darstellung einer Zahl n durch einen λ-Term mit $\overline{n}$. Auf diese Art wird eine Verwechselung der *Zahl n* mit ihrer Darstellung als *Term* vermieden.

[50] Aus logischer Sicht sind x und y zwei Variablen, deren Vorkommen in t durch den spread-Operator gebunden wird. Zusätzlich wird festgelegt, daß x an die erste Komponente des Paares p gebunden wird und y an die zweite.

Definition 2.3.17 (Darstellung von Zahlen durch Church Numerals)

$$
\begin{array}{lcl}
f^n\ t & \equiv & \underbrace{f\ (f\ \ldots (f\,t)\ \ldots)}_{n-mal} \\
\overline{n} & \equiv & \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x} \\
\mathbf{s} & \equiv & \lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{n\ f\ (f\ x)} \\
\mathbf{add} & \equiv & \lambda \mathtt{m}.\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{m\ f\ (n\ f\ x)} \\
\mathbf{mul} & \equiv & \lambda \mathtt{m}.\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{m\ (n\ f)\ x} \\
\mathbf{exp} & \equiv & \lambda \mathtt{m}.\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{n\ m\ f\ x} \\
\mathbf{zero} & \equiv & \lambda \mathtt{n}.\ \mathtt{n}\ (\lambda \mathtt{n}.\mathbf{F})\ \mathbf{T} \\
\mathbf{p} & \equiv & \lambda \mathtt{n}.\ (\mathtt{n}\ (\lambda \mathtt{fx}.\ \langle \mathtt{s}, \mathsf{let}\ \langle \mathtt{f},\mathtt{x}\rangle = \mathtt{fx}\ \mathsf{in}\ \mathtt{f\ x}\rangle)\ \langle \lambda \mathtt{z}.\overline{0}, \overline{0}\rangle).2 \\
\mathbf{PRs}[base,h] & \equiv & \lambda \mathtt{n}.\ \mathtt{n}\ h\ base
\end{array}
$$

Die Terme **s**, **add**, **mul**, **exp**, **p** und **zero** repräsentieren die Nachfolgerfunktion, Addition, Multiplikation, Exponentiation, Vorgängerfunktion und einen Test auf Null. **PRs** ist eine einfache Form der Primitiven Rekursion. Diese Repräsentationen hängen natürlich von der Zahlendarstellung durch Church Numerals ab.

So muß die *Nachfolgerfunktion* **s** dafür sorgen, daß bei Eingabe eines Terms $\overline{n}{=}\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x}$ das erste Argument `f` einmal öfter als bisher auf das zweite Argument `x` angewandt wird. Dies wird dadurch erreicht, daß durch geschickte Manipulationen das zweite Argument durch `(f x)` ausgetauscht wird. Bei der Simulation der *Addition* benutzt man die Erkenntnis, daß $\mathtt{f}^{m+n}\,\mathtt{x}$ identisch ist mit $\mathtt{f}^m\,(\mathtt{f}^n\,\mathtt{x})$ und ersetzt entsprechend das zweite Argument von $\overline{m}{=}\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^m\,\mathtt{x}$ durch $\mathtt{f}^n\,\mathtt{x}$. Bei der *Multiplikation* muß man – gemäß der Erkenntnis $\mathtt{f}^{m*n}\,\mathtt{x} = (\mathtt{f}^m)^n\,\mathtt{x}$ – das erste Argument modifizieren und bei der *Exponentiation* muß man noch einen Schritt weitergehen. Der *Test auf Null* ist einfach, da $\overline{0}{=}\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{x}$ ist. Angewandt auf $(\lambda \mathtt{n}.\mathbf{F})$ und **T** liefert dies **T** während jedes andere Church Numeral die Funktion $(\lambda \mathtt{n}.\mathbf{F})$ auswertet, also **F** liefert.

Die *Vorgängerfunktion* ist verhältnismäßig kompliziert zu simulieren, da wir bei Eingabe des Terms der Form $\overline{n}{=}\lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{f}^n\,\mathtt{x}$ eine Anwendung von `f` *entfernen* müssen. Dies geht nur dadurch, daß wir den Term komplett neu aufbauen. Wir tun dies, indem wir den Term $\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x}$ schrittweise abarbeiten und dabei die Nachfolgerfunktion **s** mit jeweils einem Schritt Verzögerung auf $\overline{0}$ anwenden. Bei der Programmierung müssen wir hierzu ein Paar $\langle \mathtt{f},\mathtt{x}\rangle$ verwalten, wobei `x` der aktuelle Ausgabewert ist und `f` die *im nächsten Schritt* anzuwendende Funktion. Startwert ist also $\overline{0}$ für `x` und $\lambda \mathtt{z}.\overline{0}$ für `f`, da im ersten Schritt ja ebenfalls $\overline{0}$ als Ergebnis benötigt wird. Ab dann wird für `x` immer der bisherige Wert benutzt und **s** für `f`.

Wir wollen am Beispiel der Nachfolgerfunktion und der Addition zeigen, daß die hier definierten Terme tatsächlich das Gewünschte leisten.

Beispiel 2.3.18

Es sei $n \in \mathbb{N}$ eine beliebige natürliche Zahl. Wir zeigen, daß $\mathbf{s}\ \overline{n}$ tatsächlich reduzierbar auf $\overline{n+1}$ ist.

$$
\begin{array}{lll}
\mathbf{s}\ \overline{n} \ \equiv & (\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{n\ f\ (f\ x)})\ (\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x}) & \\
\longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ (\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x})\ \mathtt{f\ (f\ x)} & \\
\longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ (\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x})\ \mathtt{(f\ x)} & \\
\longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\ \mathtt{(f\ x)} & \\
\longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^{n+1}\ \mathtt{x} & \equiv\ \overline{n+1}
\end{array}
$$

Es seien $m,n \in \mathbb{N}$ beliebige natürliche Zahlen. Wir zeigen, daß $\mathbf{add}\ \overline{m}\ \overline{n}$ reduzierbar auf $\overline{m+n}$ ist.

$$
\begin{array}{lll}
\mathbf{add}\ \overline{m}\ \overline{n} \ \equiv & (\lambda \mathtt{m}.\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{m\ f\ (n\ f\ x)})\ \overline{m}\ \overline{n} & \\
\longrightarrow & (\lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \overline{m}\ \mathtt{f\ (n\ f\ x)})\ \overline{n} & \\
\longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ \overline{m}\ \mathtt{f}\ (\overline{n}\ \mathtt{f\ x}) & \equiv\ \lambda \mathtt{f}.\lambda \mathtt{x}.\ (\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^m\,\mathtt{x})\ \mathtt{f}\ (\overline{n}\ \mathtt{f\ x}) \\
\longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ (\lambda \mathtt{x}.\ \mathtt{f}^m\,\mathtt{x})\ (\overline{n}\ \mathtt{f\ x}) & \\
\longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^m\,(\overline{n}\ \mathtt{f\ x}) & \equiv\ \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^m\,((\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x})\ \mathtt{f\ x}) \\
\longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^m\,((\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x})\ \mathtt{x}) & \\
\longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^m\,(\mathtt{f}^n\,\mathtt{x}) & \\
\longrightarrow & \lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^{m+n}\,\mathtt{x} & \equiv\ \overline{m+n}
\end{array}
$$

Ähnlich leicht kann man zeigen, daß **mul** die Multiplikation, **exp** die Exponentiation und **zero** einen Test auf Null repräsentiert. Etwas schwieriger ist die Rechtfertigung der Vorgängerfunktion **p**. Die Rechtfertigung von **PRs** läßt sich durch Abwandlung der Listeninduktion aus Beispiel 2.3.20 erreichen.

Listen

Endliche Listen von Termen lassen sich mathematisch als eine Erweiterung des Konzepts natürlicher Zahlen ansehen. Bei Zahlen startet man mit der Null und kann jede weitere Zahl dadurch bilden, daß man schrittweise die Nachfolgerfunktion **s** anwendet. Bei endlichen Listen startet man enstprechend mit einer leeren (nullelementige) Liste – bezeichnet durch `[]` – und bildet weitere Listen dadurch, daß man schrittweise Elemente $\mathtt{a}_i$ vor die bestehende Liste `L` anhängt. Die entsprechende Operation – bezeichnet durch $\mathtt{a}_i\mathtt{.L}$ – wird durch eine Funktion **cons** – das Gegenstück zur Nachfolgerfunktion **s** ausgeführt.

Definition 2.3.19 (Operatoren auf Listen)

$$\begin{array}{lcl}
\mathtt{[]} & \equiv & \lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{x} \\
\textbf{cons}(t,list) & \equiv & \lambda\mathtt{f}.\lambda\mathtt{x}.\,\mathtt{f}\ t\ (list\ \mathtt{f}\ \mathtt{x}) \\
t.list & \equiv & \textbf{cons}(t,list) \\
\textbf{list_ind}[base,h] & \equiv & \lambda\mathtt{list}.\ \mathtt{list}\ h\ base
\end{array}$$

Die leere Liste ist also genauso definiert wie die Repräsentation der Zahl 0 während der Operator `cons` nun die Elemente der Liste – jeweils getrennt durch die Variable `f` – nebeneinanderstellt. Die Liste $\mathtt{a}_1.\mathtt{a}_2\ \ldots\mathtt{a}_n$ wird also dargestellt durch den Term

$$\lambda\mathtt{f}.\lambda\mathtt{x}.\,\mathtt{f}\ \mathtt{a}_1\ (\mathtt{f}\ \mathtt{a}_2\ \ldots(\mathtt{f}\ \mathtt{a}_n\ \mathtt{x})\ \ldots)$$

Die Induktion auf Listen $\textbf{list\_ind}[base,h]$ ist die entsprechende Erweiterung der einfachen primitiven Rekursion. Ihr Verhalten beschreibt das folgende Beispiel.

Beispiel 2.3.20

Es sei f definiert durch $f\equiv\textbf{list\_ind}[base,h]$. Wir zeigen, daß f die Rekursionsgleichungen

$$f(\mathtt{[]})\ \mathtt{=}\ base \quad\text{und}\quad f(t.list)\ \mathtt{=}\ h\ (t)\ (f(list))$$

erfüllt. Im Basisfall ist dies relativ einfach

$$\begin{array}{lcl@{\quad}cl}
f(\mathtt{[]}) & \equiv & \textbf{list_ind}[base,h]\ \mathtt{[]} & \equiv & (\lambda\mathtt{list}.\ \mathtt{list}\ h\ base)\ \mathtt{[]} \\
& \longrightarrow & \mathtt{[]}\ h\ base & \equiv & (\lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{x})\ h\ base \\
& \longrightarrow & (\lambda\mathtt{x}.\mathtt{x})\ base & & \\
& \longrightarrow & base & &
\end{array}$$

Schwieriger wird es im Rekursionsfall:

$$\begin{array}{lcl@{\quad}cl}
f(t.list) & \equiv & \textbf{list_ind}[base,h]\ t.list & \equiv & (\lambda\mathtt{list}.\ \mathtt{list}\ h\ base)\ t.list \\
& \longrightarrow & t.list\ h\ base & \equiv & (\lambda\mathtt{f}.\lambda\mathtt{x}.\,\mathtt{f}\ t\ (list\ \mathtt{f}\ \mathtt{x}))\ h\ base \\
& \longrightarrow & (\lambda\mathtt{x}.\,h\ t\ (list\ h\ \mathtt{x}))\ base & & \\
& \longrightarrow & h\ t\ (list\ h\ base) & &
\end{array}$$

Dies ist offensichtlich nicht der gewünschte Term. Wir können jedoch zeigen, daß sich $h\ (t)\ (f(list))$ auf denselben Term reduzieren läßt.

$$\begin{array}{lcl@{\quad}cl}
h\ (t)\ (f(list)) & \equiv & h\ t\ (\textbf{list_ind}[base,h]\ list) & \equiv & h\ t\ ((\lambda\mathtt{list}.\ \mathtt{list}\ h\ base)\ list) \\
& \longrightarrow & h\ t\ (list\ h\ base) & &
\end{array}$$

Die Rekursionsgleichungen von f beziehen sich also auf *semantische Gleichheit* und nicht etwa darauf, daß die linke Seite genau auf die rechte reduziert werden kann.

Rekursion

Die bisherigen Konstrukte erlauben uns, bei der Programmierung im λ-Kalkül Standardkonstrukte wie Zahlen, Tupel und Listen sowie eine bedingte Funktionsaufrufe zu verwenden. Uns fehlt nur noch eine Möglichkeit *rekursive Funktionsaufrufe* – das Gegenstück zur Schleife in imperative Programmiersprachen – auf einfache Weise zu beschreiben. Wir benötigen also einen Operator, der es uns erlaubt, eine Funktion f durch eine rekursive Gleichung der Form

$$f(x)\ =\ t[f,x]^{51}$$

zu definieren, also durch eine Gleichung, in der f auf beiden Seiten vorkommt. Diese Gleichung an sich beschreibt aber noch keinen Term, sondern nur eine *Bedingung*, die ein Term zu erfüllen hat, und ist somit nicht unmittelbar für die Programmierung zu verwenden. Glücklicherweise gibt es jedoch ein allgemeines Verfahren, einen solchen Term direkt aus einer rekursiven Gleichung zu erzeugen. Wenn wir nämlich in der obigen Gleichung den Term t durch die Funktion $T\equiv\lambda f.\lambda x.t[f,x]$ ersetzen, so können wir die Rekursionsgleichung umschreiben als $f(x)\ =\ T\ \ f\ \ x$ bzw. als

$$f\ =T\ \ f$$

Eine solche Gleichung zu lösen, bedeutet, einen *Fixpunkt* der Funktion T zu bestimmen, also ein Argument f, welches die Funktion T in sich selbst abbildet. Einen Operator R, welcher für beliebige Terme (d.h. also Funktionsgleichungen) deren Fixpunkt bestimmt, nennt man *Fixpunktkombinator* (oder *Rekursor*).

Definition 2.3.21 (Fixpunktkombinator)

Ein Fixpunktkombinator ist ein λ-Term R mit der Eigenschaft, daß für jeden beliebigen λ-Term T die folgende Gleichung erfüllt ist:

$$R\ \ T\ =T(\ R\ \ T)\ )$$

Ein Fixpunktkombinator R liefert bei Eingabe eines beliebigen Terms T also eine Funktion $f=R(T)$, für welche die rekursive Funktionsgleichung $f=T\ f$ erfüllt ist.

Natürlich entsteht die Frage, ob solche Fixpunktkombinatoren überhaupt existieren. Für den λ-Kalkül kann diese Frage durch die Angabe konkreter Fixpunktkombinatoren positiv beantwortet werden. Der bekannteste unter diesen ist der sogenannte **Y**-Kombinator.

Definition 2.3.22 (Der Fixpunktkombinator Y)

$$\begin{array}{lcl}
\mathbf{Y} & \equiv & \lambda\mathtt{f}.\,(\lambda\mathtt{x}.\,\mathtt{f}\ (\mathtt{x}\ \mathtt{x}))(\lambda\mathtt{x}.\,\mathtt{f}\ (\mathtt{x}\ \mathtt{x})) \\
\mathsf{letrec}\ \ f(x)=t & \equiv & \mathbf{Y}(\lambda f.\lambda x.t)
\end{array}$$

Lemma 2.3.23

Y *ist ein Fixpunktkombinator*

Beweis: Wie im Falle der Listeninduktion (Beispiel 2.3.20) ergibt sich die Gleichung $\mathbf{Y}(t)\ =\ t(\mathbf{Y}(t))$ nur dadurch, daß $\mathbf{Y}(t)$ und $t(\mathbf{Y}(t))$ auf denselben Term reduziert werden können.[52]

$$\begin{array}{lcl}
\mathbf{Y}\ t & \equiv & \lambda\mathtt{f}.\,(\lambda\mathtt{x}.\,\mathtt{f}\ (\mathtt{x}\ \mathtt{x}))(\lambda\mathtt{x}.\,\mathtt{f}\ (\mathtt{x}\ \mathtt{x}))\ t \\
& \longrightarrow & (\lambda\mathtt{x}.\,t\ (\mathtt{x}\ \mathtt{x}))\ (\lambda\mathtt{x}.\,t\ (\mathtt{x}\ \mathtt{x})) \\
& \longrightarrow & t\ (\ (\lambda\mathtt{x}.\,t\ (\mathtt{x}\ \mathtt{x}))\ (\lambda\mathtt{x}.\,t\ (\mathtt{x}\ \mathtt{x}))\ ) \\
t\ (\mathbf{Y}\ t) & \equiv & t\ (\lambda\mathtt{f}.\,(\lambda\mathtt{x}.\,\mathtt{f}\ (\mathtt{x}\ \mathtt{x}))(\lambda\mathtt{x}.\,\mathtt{f}\ (\mathtt{x}\ \mathtt{x}))\ t) \\
& \longrightarrow & t\ (\ (\lambda\mathtt{x}.\,t\ (\mathtt{x}\ \mathtt{x}))\ (\lambda\mathtt{x}.\,t\ (\mathtt{x}\ \mathtt{x}))\ )
\end{array}$$

□

Fixpunktkombinatoren wie **Y** erzeugen also aus jeder beliebigen rekursiven Funktionsgleichung der Form $f(x)\ \mathtt{=}\ t$ einen λ-Term, welcher – wenn eingesetzt für f – diese Gleichung erfüllt. Fixpunktkombinatoren können also zur “Implementierung” rekursiver Funktionen eingesetzt werden. Man beachte aber, daß der Ausdruck $\mathsf{letrec}\ \ f(x)=t$ einen Term beschreibt und nicht etwa eine Definition ist, die den *Namen* f mit diesem Term verbindet.

[51] $t[f,x]$ ist ein Term, in dem die Variablen f und x frei vorkommen (vergleiche Definition 2.2.18 auf Seite 36).

[52] Ein Fixpunktkombinator, der sich auf seinen Fixpunkt reduzieren läßt, ist $(\lambda\mathtt{x}.\lambda\mathtt{y}.\,\mathtt{y}\ (\mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ (\lambda\mathtt{x}.\lambda\mathtt{y}.\,\mathtt{y}\ (\mathtt{x}\ \mathtt{x}\ \mathtt{y}))$

$\Gamma \vdash f\,t$ = $g\,u$ **by** apply_eq
  $\Gamma \vdash f$ = g
  $\Gamma \vdash t$ = u
	$\Gamma \vdash \lambda x.t$ = $\lambda y.u$ **by** lambda_eq*
	  Γ, x':U $\vdash t[x'/x]$ = $u[x'/y]$

$\Gamma \vdash (\lambda x.u)\,s$ = t **by** reduction $u[s/x] = t$	
$\Gamma \vdash t$=t **by** reflexivity	$\Gamma \vdash t_1$=t_2 **by** symmetry $\Gamma \vdash t_2$=t_1

$\Gamma \vdash t_1$=t_2 **by** transitivity u
  $\Gamma \vdash t_1$=u
  $\Gamma \vdash u$=t_2	

*: *Die Umbenennung $[x'/x]$ erfolgt, wenn x in Γ frei vorkommt.*

Abbildung 2.7: Sequenzenkalkül für die Gleichheit von λ-Termen

2.3.4 Ein Kalkül zum Schließen über Berechnungen

Bisher haben wir uns im wesentlichen mit mit der Auswertung von λ-Termen beschäftigt. Die Reduktion von λ-Termen dient dazu, den Wert eines gegebenen Termes zu bestimmen. Zum Schließen über Programme und ihr Verhalten reicht dies jedoch nicht ganz aus, da wir nicht nur Terme untersuchen wollen, die – wie 4+5 und 9 – aufeinander reduzierbar sind, sondern auch Terme, die – wie 4+5 und 2+7 – den gleichen Wert haben. Mit den bisher eingeführten Konzepten läßt sich Werte-Gleichheit relativ leicht definieren.

Definition 2.3.24 (Gleichheit von λ-Termen)

> *Zwei λ-Terme heißen (semantisch) gleich (<u>konvertierbar</u>), wenn sie auf denselben λ-Term reduziert werden können:*
>
> <u>t=u</u> *gilt genau dann, wenn es einen Term v gibt mit $t \xrightarrow{*} v$ und $u \xrightarrow{*} v$.*

Man beachte, daß Werte-Gleichheit weit mehr ist als nur syntaktische Gleichheit und daß sich das Gleichheitssymbol = auf diese Werte-Gleichheit bezieht.

Da wir für die Reduktion von λ-Termen in Definition 2.3.11 bereits eine sehr präzise operationalisierbare Charakterisierung angegeben haben, ist es nunmehr nicht sehr schwer, einen Kalkül aufzustellen, mit dem wir formale Schlüsse über die Resultate von Berechnungen – also die Gleichheit zweier λ-Terme – ziehen können. Wir formulieren hierzu die "Regeln" der Definition 2.3.11 im Stil des Sequenzenkalküls und ergänzen eine Regel für Symmetrie.[53]

Abbildung 2.7 faßt die Regeln des Sequenzenkalküls für die Gleichheit von λ-Termen zusammen. Sie ergänzen die bekannten Gleichheitsregeln der Prädikatenlogik (siehe Abschnitt 2.2.7) um die β-Reduktion und zwei Regeln zur Dekomposition von λ-Termen. Letztere machen die Substutionsregel innerhalb des reinen λ-Kalküls überflüssig.

Die Regel `lambda_eq` verdient besondere Beachtung, da ihre Formulierung auch die α-Konversion beinhaltet. Zwei λ-Abstraktionen $\lambda x.t$ und $\lambda y.u$ sind gleich, die Terme t und u nach einer Umbenennung der gebundenen Variablen x und y zu einer gemeinsamen Abstraktionsvariablen gleich sind. Auch hier gilt eine Art Eigenvariablenbedingung: eine Umbenennung ist in jedem Fall erforderlich, wenn die Abstraktionsvariable (des ersten Terms) bereits in der Hypothesenliste vorkommt.

Mit dem folgenden Beispiel wollen wir nun zeigen, daß die booleschen Werte **T** und **F** tatsächlich 'gegenteilige' Objekte beschreiben. Wären sie nämlich gleich, dann würde folgen, daß *alle* λ-Terme gleich sind.

[53]Der Beweisbegriff ergibt sich unmittelbar aus dem der Prädikatenlogik (siehe Definition 2.2.12 auf Seite 30). Die einzige Änderung besteht darin, daß die Konklusion nun immer eine Gleichheitsformel ist. Eine vollständige Definition werden wir erst im Kapitel 3 für die Typentheorie aufstellen, welche alle Kalküle dieses Kapitels umfaßt.

Beispiel 2.3.25

Es seien s und t beliebige λ-Terme. Ein Beweis für s=t könnte wie folgt verlaufen:

```
⊢  s=t
by transitivity  if F then s else t
|\
| ⊢ s = if F then s else t
| by symmetry
|  \
|   ⊢ if F then s else t = s
|   by transitivity  if T then s else t
|   |\
|   | ⊢ if F then s else t =  if T then s else t
|   | by apply_eq        (das schließt eine Auflösung der Definitionen mit ein)
|   | |\
|   | | ⊢ F s  = T s
|   | | by apply_eq
|   | | |\
|   | | | ⊢ [F = T]
|   | |  \
|   | |   ⊢ s = s
|   | |   by reflexivity
|   |  \
|   |   ⊢ t = t
|   |   by reflexivity
|    \
|     ⊢ if T then s else t  = s
|     by reduction
|      \
|       ⊢ (λy.s) t = s
|       by reduction
|        \
|         ⊢ s = s
|         by reflexivity
 \
  if F then s else t = t
  by reduction
   \
    ⊢ (λy.y) t = t
    by reduction
     \
      ⊢ t = t
      by reflexivity
```

Dieses Beispiel gibt uns auch eine Handhabe, wie wir die Ungleichheit zweier Terme s und t beweisen können. Wenn wir mit den Regeln aus Abbildung 2.7 zeigen können, daß aus s=t die Gültigkeit von **F** = **T** folgt, dann wissen wir, daß s und t nicht gleich sein können. Diese Erkenntnis ist allerdings nicht direkt im Kalkül enthalten.

2.3.5 Die Ausdruckskraft des λ-Kalküls

Zu Beginn dieses Abschnitts haben wir die Einführung des λ-Kalküls als Formalismus zum Schließen über Berechnungen damit begründet, daß der λ-Kalkül das einfachste aller Modelle zur Erklärung von Berechenbarkeit ist. Wir wollen nun zeigen, daß der λ-Kalkül *Turing-mächtig* ist, also genau die Klasse der rekursiven Funktionen beschreiben kann. Gemäß der Church'schen These ist er damit in der Lage, jede berechenbare Funktion auszudrücken.

Wir machen uns bei diesem Beweis zunutze, daß Turingmaschinen, imperative Programmiersprachen, μ-rekursive Funktionen etc. bekanntermaßen äquivalent sind.[54] Der Begriff der Berechenbarkeit wird dabei in allen Fällen auf den natürlichen Zahlen abgestützt. Um also einen Vergleich durchführen zu können, definieren wir zunächst die λ-Berechenbarkeit von Funktionen auf natürlichen Zahlen.

[54]Diese Tatsache sollte aus einer Grundvorlesung über theoretische Informatik bekannt sein.

Definition 2.3.26 (λ-**Berechenbarkeit**)

Eine (möglicherweise partielle) Funktion f:$\mathbb{N}^n \to \mathbb{N}$ heißt λ-berechenbar, wenn es einen λ-Term t gibt mit der Eigenschaft, daß für alle $x_1, ..., x_n, m \in \mathbb{N}$ gilt:

$$f(x_1, ..., x_n) = m \quad \text{genau dann, wenn} \quad t\ \overline{x_1}\ \ldots \overline{x_n}\ = \overline{m}$$

Es ist leicht einzusehen, daß man λ-berechenbare Funktionen programmieren kann. Um die Umkehrung zu beweisen – also die Behauptung, daß jede rekursive Funktion auch λ-berechenbar ist – zeigen wir, daß jede μ-rekursive Funktion im λ-Kalkül simuliert werden kann. Wir fassen zu diesem Zweck die Definition der μ-rekursiven Funktionen kurz zusammen.

Definition 2.3.27 (μ-**rekursive Funktionen**)

Die Klasse der μ-rekursiven Funktionen ist induktiv durch die folgenden Bedingungen definiert.

1. *Alle Konstanten $0, 1, 2, \ldots \in \mathbb{N}$ sind (nullstellige) μ-rekursive Funktionen.*
2. *Die Nullfunktion $z : \mathbb{N} \to \mathbb{N}$ – definiert durch $z(n) = 0$ für alle $n \in \mathbb{N}$ – ist μ-rekursiv.*
3. *Die Nachfolgerfunktion $s : \mathbb{N} \to \mathbb{N}$ – definiert durch $s(n) = n + 1$ für alle $n \in \mathbb{N}$ – ist μ-rekursiv.*
4. *Die Projektionsfunktionen $pr^n_m : \mathbb{N}^n \to \mathbb{N}$ $(m \leq n)$ – definiert durch $pr^n_m(x_1, ..., x_n) = x_m$ für alle $x_1, ..., x_n \in \mathbb{N}$ – sind μ-rekursiv.*
5. *Die Komposition $Cn[f,\ g_1...g_n]$ der Funktionen $f, g_1...g_n$ ist μ-rekursiv, wenn $f, g_1...g_n$ μ-rekursive Funktionen sind. Dabei ist $Cn[f,\ g_1...g_n]$ die eindeutig bestimmte Funktion h, für die gilt*
   $$h(\vec{x}) = f(\ g_1(\vec{x}), ..., g_n(\vec{x})\ )^{55}$$
6. *Die primitive Rekursion $Pr[f, g]$ zweier Funktionen f und g ist μ-rekursiv, wenn f und g μ-rekursiv sind. Dabei ist $Pr[f, g]$ die eindeutig bestimmte Funktion h, für die gilt*
   $$h(\vec{x}, 0) = f(\vec{x}) \quad \text{und} \quad h(\vec{x}, y + 1) = g(\vec{x}, y,\ h(\vec{x}, y))$$
7. *Die Minimierung $Mn[f]$ einer Funktion f ist μ-rekursiv, wenn f μ-rekursiv ist. Dabei ist $Mn[f]$ die eindeutig bestimmte Funktion h, für die gilt*
   $$h(\vec{x}) = \begin{cases} min\{y \mid f(\vec{x}, y) = 0\} & \text{falls dies existiert und alle } f(\vec{x}, i), i < y \text{ definiert sind} \\ undefiniert & \text{sonst} \end{cases}$$

Aufgrund der konservativen Erweiterungen des λ-Kalküls, die wir im Abschnitt 2.3.3 gegeben haben, ist es nun nicht mehr schwer, den Äquivalenzbeweis zu führen.

Satz 2.3.28

Die Klasse der λ-berechenbaren Funktionen ist identisch mit der Klasse der μ-rekursiven Funktionen.

Beweis: Die λ-berechenbaren Funktionen lassen sich offensichtlich durch Programme einer der gängigen imperativen Programmiersprachen simulieren. Da diese sich wiederum durch Turingmaschinen beschreiben lassen und die Klasse der μ-rekursiven Funktionen identisch sind mit der Klasse der Turing-berechenbaren Funktionen, folgt hieraus, daß alle λ-berechenbaren Funktionen auch μ-rekursiv sind.

Wir können uns daher auf den interessanten Teil des Beweises konzentrieren, nämlich dem Nachweis, daß man mit einem so einfachen Berechnungsmechanismus wie dem λ-Kalkül tatsächlich alle berechenbaren Funktionen repräsentieren können. Wir weisen dazu nach, daß alle sieben Bedingungen der μ-rekursiven Funktionen aus Definition 2.3.27 durch λ-Terme erfüllt werden können.

1. Gemäß Definition 2.3.26 werden Konstanten $0, 1, 2, \ldots \in \mathbb{N}$ genau durch die Church-Numerals $\overline{0}, \overline{1}, \overline{2}, \ldots$ aus Definition 2.3.17 repräsentiert. Damit sind alle Konstanten auch λ-berechenbar.
2. Die Nullfunktion z läßt sich darstellen durch den Term λ`n.`$\overline{0}$ und ist somit λ-berechenbar.
3. Die Nachfolgerfunktion s haben wir in Beispiel 2.3.18 auf Seite 55 untersucht. Sie wird dargestellt durch den Term **s** und ist somit auch λ-berechenbar.

[55] $\vec{x}$ ist abkürzend für ein Tupel $(x_1, ..., x_m)$

4. Die Projektionsfunktionen pr^n_m sind leicht zu repräsentieren. Man wähle für pr^n_m den Term $\lambda\mathtt{x}_1 \ldots \lambda\mathtt{x}_n.\mathtt{x}_m$.
5. Die Komposition läßt sich genauso leicht direkt nachbilden. Definieren wir
   $$\mathbf{Cn} \equiv \lambda\mathtt{f}.\lambda\mathtt{g}_1 \ldots \lambda\mathtt{g}_n.\ \lambda\vec{\mathtt{x}}.\ \mathtt{f}\ (\mathtt{g}_1\ \vec{\mathtt{x}})\ \ldots(\mathtt{g}_n\ \vec{\mathtt{x}})$$
   so simuliert **Cn** den Kompositionsoperator Cn.[56]

   Sind also $f, g_1...g_n$ λ-berechenbare Funktionen und $F, G_1...G_n$ die zugehörigen λ-Terme, so repräsentiert **Cn** $F\ G_1\ \ldots G_n$ – wie man durch Einsetzen leicht zeigen kann – die Komposition $Cn[f,\ g_1...g_n]$. Damit ist gezeigt, daß die Komposition λ-berechenbarer Funktionen wieder eine λ-berechenbare Funktion ist.
6. Mithilfe der Fixpunktkombinatoren läßt sich die primitive Rekursion zweier Funktionen auf einfache und natürliche Weise nachbilden. Wir müssen hierzu nur die Rekursionsgleichungen von $Pr[f, g]$ in eine einzige Gleichung umwandeln. Für $h = Pr[f, g]$ gilt
   $$h(\vec{x}, y) = \begin{cases} f(\vec{x}) & \text{falls } y = 0 \\ g(\vec{x}, y{-}1,\ h(\vec{x}, y{-}1)) & \text{sonst} \end{cases}$$
   Wenn wir die rechte Seite dieser Gleichung durch Conditional und Vorgängerfunktion beschreiben und hierauf anschließend den **Y**-Kombinator anwenden, so haben wir eine Beschreibung fur die Funktion h durch einen λ-Term. Definieren wir also
   $$\mathbf{Pr} \equiv \lambda\mathtt{f}.\lambda\mathtt{g}.\ \mathbf{Y}\ (\lambda\mathtt{h}.\lambda\vec{\mathtt{x}}.\lambda\mathtt{y}.\ \text{if } \mathbf{zero}\mathtt{y} \text{ then } \mathtt{f}\ \vec{\mathtt{x}} \text{ else } \mathtt{g}\ \vec{\mathtt{x}}\ (\mathbf{p}\ \mathtt{y})\ (\mathtt{h}\ \vec{\mathtt{x}}\ (\mathbf{p}\ \mathtt{y}))\ )$$
   so simuliert **Pr** den Operator der primitiven Rekursion Pr. Damit ist gezeigt, daß die primitive Rekursion λ-berechenbarer Funktionen wieder eine λ-berechenbare Funktion ist.
7. Auch die Minimierung kann mithilfe der Fixpunktkombinatoren als λ-berechenbar nachgeweisen werden. Minimierung $Mn[f](\vec{x})$ ist im Endeffekt eine unbegrenzte Suche nach einem Wert y, für den $f(\vec{x}, y) = 0$ ist. Startwert dieser Suche ist die Zahl 0.

   Die bei einem gegebenen Startwert y beginnende Suche nach einer Nullstelle von f läßt sich wie folgt durch durch eine rekursive Gleichung ausdrücken.
   $$\mathsf{min}_f(\vec{x}, y) = \begin{cases} y & \text{falls } f(\vec{x}, y) = 0 \\ \mathsf{min}_f(\vec{x}, y + 1) & \text{sonst} \end{cases}$$
   Da f und $\vec{x}$ für diese Gleichung als Konstante aufgefaßt werden können, läßt sich diese Gleichung etwas vereinfachen, bevor wir auf sie den **Y**-Kombinator anwenden. Definieren wir also
   $$\mathbf{Mn} \equiv \lambda\mathtt{f}.\lambda\vec{\mathtt{x}}.(\ \mathbf{Y}\ (\lambda\mathtt{min}.\lambda\mathtt{y}.\ \text{if } \mathbf{zero}(\mathtt{f}\ \vec{\mathtt{x}}\ \mathtt{y}) \text{ then } \mathtt{y} \text{ else } \mathtt{min}\ (\mathbf{s}\ \mathtt{y})\ )\ )\overline{0}$$
   so simuliert **Mn** den Minimierungsoperator Mn. Damit ist gezeigt, daß die Minimierung λ-berechenbarer Funktionen wieder eine λ-berechenbare Funktion ist.

Wir haben somit bewiesen, daß alle Grundfunktionen λ-berechenbar sind und die Klasse der λ-berechenbaren Funktionen abgeschlossen ist unter den Operationen Komposition, primitive Rekursion und Minimierung. Damit sind alle μ-rekursiven Funktionen auch λ-berechenbar. □

2.3.6 Semantische Fragen

Wir haben in den bisherigen Abschnitten die Syntax und die Auswertung von λ-Termen besprochen und gezeigt, daß man mit λ-Termen alle berechenbaren Funktionen ausdrücken kann. Offensichtlich ist auch, daß jeder λ-Term der Gestalt $\lambda x\,.\,t$ eine Funktion beschreiben muß. Jedoch ist unklar, mit welchem mathematischen Modell man eine solche Funktion als mengentheoretisches Objekt beschreiben kann.

Es ist nicht sehr schwer, ein sehr abstraktes und an Termen orientiertes Modell für den λ-Kalkül anzugeben (siehe [Church & Rosser, 1936]). Jeder Term t beschreibt die Menge M_t der Terme, die im Sinne von Definition 2.3.24 gleich sind zu t. Jede Abstraktion $\lambda x\,.\,t$ repräsentiert eine Funktion $f_{\lambda x.t}$, welche eine Menge M_u in die Menge $M_{t[u/x]}$ abbildet. Diese Charakterisierung bringt uns jedoch nicht weiter, da sie keine Hinweise auf den Zusammenhang zwischen Funktionen des λ-Kalküls und 'gewöhnlichen' mathematischen Funktionen liefert.

[56] Genau besehen haben wir hier beschrieben, wie wir für jede feste Anzahl n von Funktionen $\mathtt{g}_1 \ldots \mathtt{g}_n$ den Kompositionsoperator simulieren.

Einfache mathematische Modelle, in denen λ-Terme als Funktionen eines festen Funktionenraumes interpretiert werden können, sind jedoch ebenfalls auszuschließen. Dies liegt an der Tatsache, daß λ-Terme eine Doppelrolle spielen: sie können als Funktion und als Argument einer anderen Funktion auftreten. Daher ist es möglich, λ-Funktionen zu konstruieren, die in sinnvoller Weise auf sich selbst angewandt werden können.

Beispiel 2.3.29

Wir betrachten den Term

$$\texttt{twice} \equiv \lambda\texttt{f}.\lambda\texttt{x.f (f x)}.$$

Angewandt auf zwei Terme f und u produziert dieser Term die zweifache Anwendung von f auf u

$$\texttt{twice}\ f\ u \xrightarrow{*} f\ (f\ u)$$

Damit ist `twice` als eine Funktion zu verstehen. Andererseits darf `twice` aber auf sich selbst angewandt werden, wodurch eine Funktion entsteht, die ihr erstes Argument viermal auf das zweite anwendet:

$$\begin{array}{rl}
\texttt{twice twice} \equiv & (\lambda\texttt{f}.\lambda\texttt{x.f (f x)) twice} \\
\xrightarrow{*} & \lambda\texttt{x. twice (twice x)} \\
\equiv & \lambda\texttt{x.}(\lambda\texttt{f}.\lambda\texttt{x.f (f x)) (twice x)} \\
\xrightarrow{*} & \lambda\texttt{x.}\lambda\texttt{x'. (twice x) ((twice x) x')} \\
\cong & \lambda\texttt{f}.\lambda\texttt{x. (twice f) ((twice f) x)} \\
\xrightarrow{*} & \lambda\texttt{f}.\lambda\texttt{x. (twice f) (f (f x))} \\
\xrightarrow{*} & \lambda\texttt{f}.\lambda\texttt{x. f (f (f (f x)))}
\end{array}$$

Die übliche mengentheoretische Sichtweise von Funktionen muß die Selbstanwendung von Funktionen jedoch verbieten. Sie identifiziert nämlich eine Funktion f mit der Menge $\{(x,y) \mid y = f(x)\}$, also dem Graphen der Funktion. Wenn eine selbstanwendbare Funktion wie `twice` mengentheoretisch interpretierbar wäre, dann müsste `twice` als eine solche Menge aufgefaßt werden und würde gelten, daß $(\texttt{twice}, y)$ für irgendein y ein Element dieser Menge ist, weil nun einmal `twice` ein legitimes Argument von `twice` ist. Dies aber verletzt ein fundamentales Axiom der Mengentheorie, welches besagt, daß eine Menge sich selbst nicht enthalten darf.[57] Wir können daher nicht erwarten, 'natürliche' Modelle für den λ-Kalkül angeben zu können. Dies wird erst möglich sein, wenn wir die zulässigen Terme auf syntaktischem Wege geeignet einschränken.[58] Diese Problematik ist jedoch nicht spezifisch für den λ-Kalkül, sondern betrifft *alle* Berechenbarkeitsmodelle, da diese – wie wir im vorigen Abschnitt gezeigt hatten – äquivalent zum λ-Kalkül sind.

Das erste mathematische Modell für berechenbare Funktionen ist die *Domain-Theorie*, die Anfang der siebziger Jahre von Dana Scott [Scott, 1972, Scott, 1976] entwickelt wurde. Diese Theorie basiert im wesentlichen auf topologischen Begriffen wie Stetigkeit und Verbänden. Sie benötigt jedoch ein tiefes Verständnis komplexer mathematischer Theorien. Aus diesem Grunde werden wir auf die *denotationelle* Semantik des λ-Kalküls nicht weiter eingehen.

2.3.7 Eigenschaften des λ-Kalküls

Bei den bisherigen Betrachtungen sind wir stillschweigend davon ausgegangen, daß jeder λ-Term einen Wert besitzt, den wir durch Reduktion bestimmen können. Wie weit aber müssen wir gehen, bevor wir den Prozeß der Reduktion als beendet erklären können? Die Antwort ist eigentlich naheliegend: wir hören erst dann auf, wenn nichts mehr zu reduzieren ist, also der entstandene Term kein Redex im Sinne der Definition 2.3.10 mehr

[57] Ein Verzicht auf dieses Axiom würde zu dem *Russelschen Paradox* führen:
Man betrachte die Mengen $M = \{X \mid X \notin X\}$ und untersuche, ob $M \in M$ ist oder nicht. Wenn wir $M \in M$ annehmen, so folgt nach Definition von M, daß M – wie jedes andere Element von M – nicht in sich selbst enthalten ist, also $M \notin M$. Da M aber die Menge *aller* Mengen ist, die sich selbst nicht enthalten, muß M ein Element von M sein: $M \in M$.

[58] Ein einfaches aber doch sehr wirksames Mittel ist hierbei die Forderung nach Typisierbarkeit, die wir im nächsten Abschnitt diskutieren werden. Die Zuordnung von Typen zu Termen beschreibt bereits auf syntaktischem Wege, zu welcher Art von Funktionenraum eine Funktion gehören soll.

enthält. Diese Überlegung führt dazu, eine Teilklasse von λ-Termen besonders hervorzuheben, nämlich solche, die nicht mehr reduzierbar sind. Diese Terme repräsentieren die Werte, die als Resultat von Berechnungen entstehen können.

Definition 2.3.30 (Normalform)

Es seien s und t beliebige λ-Terme.

1. *t ist in Normalform, wenn t keine Redizes enthält.*
2. *t ist normalisierbar, wenn es einen λ-Term in Normalform gibt, auf den t reduziert werden kann.*
3. *t heißt Normalform von s, wenn t in Normalform ist und $s \xrightarrow{*} t$ gilt.*

Diese Definition wirft natürlich eine Reihe von Fragen auf, die wir im folgenden diskutieren wollen.

Hat jeder λ-Term eine Normalform?

In Anbetracht der Tatsache, daß der λ-Kalkül Turing-mächtig ist, muß diese Frage natürlich mit *nein* beantwortet werden. Bekanntermaßen enthalten die rekursiven Funktionen auch die partiellen Funktionen – also Funktionen, die nicht auf allen Eingaben definiert sind – und es ist nicht möglich, einen Formalismus so einzuschränken, daß nur totale Funktionen betrachtet werden, ohne daß dabei gleichzeitig manche berechenbare totale Funktion nicht mehr beschreibbar ist.[59] Diese Tatsache muß sich natürlich auch auf die Reduzierbarkeit von λ-Termen auswirken: es gibt Reduktionsketten, die nicht terminieren. Wir wollen es jedoch nicht bei dieser allgemeinen Antwort belassen, sondern einen konkreten Term angeben, der nicht normalisierbar ist.

Lemma 2.3.31

Der Term $(\lambda\texttt{x. x x})\ (\lambda\texttt{x. x x})$ besitzt keine Normalform.

Beweis: Bei der Reduktion von $(\lambda\texttt{x. x x})\ (\lambda\texttt{x. x x})$ gibt es genau eine Möglichkeit. Wenn wir diese ausführen, erhalten wir denselben Term wie zuvor und können den Term somit unendlich lange weiterreduzieren. □

Führt jede Reduktionsfolge zu einer Normalform, wenn ein λ-Term normalisierbar ist?

Im Beispiel 2.3.12 (Seite 51) hatten wir bereits festgestellt, daß es unter Umständen mehrere Möglichkeiten gibt, einen gegebenen λ-Term zu reduzieren. Da wir bereits wissen, daß nicht jede Reduktionskette terminieren muß, erhebt sich natürlich die Frage, ob es etwa Terme gibt, bei denen eine Strategie, den zu reduzierenden Teilterm auszuwählen, zu einer Normalform führt, während eine andere zu einer nichtterminierenden Reduktionsfolge führt. Dies ist in der Tat der Fall.

Lemma 2.3.32

Es gibt normalisierbare λ-Terme, bei denen nicht jede Reduktionsfolge zu einer Normalform führt.

Beweis: Es sei $\mathsf{W} \equiv \lambda\texttt{x. x x x}$ und $\mathsf{I} \equiv \lambda\texttt{x. x}$. Wir betrachten den Term

$$(\lambda\texttt{x.}\lambda\texttt{y.y})\ (\mathsf{WW})\ \mathsf{I}.$$

Es gibt zwei Möglichkeiten, diesen Term zu reduzieren. Wählen wir die am meisten links-stehende, so ergibt sich als Reduktionsfolge:

$$(\lambda\texttt{x.}\lambda\texttt{y.y})\ (\mathsf{WW})\ \mathsf{I} \xrightarrow{*} (\lambda\texttt{y.y})\ \mathsf{I} \xrightarrow{*} \mathsf{I}$$

und wir hätten eine Normalform erreicht. Wenn wir dagegen den Teilterm (WW) zuerst reduzieren, dann erhalten wir (WWW). Reduzieren wir dann wieder im gleichen Bereich, so erhalten wir folgende Reduktionskette

$$(\lambda\texttt{x.}\lambda\texttt{y.y})\ (\mathsf{WW})\ \mathsf{I} \xrightarrow{*} (\lambda\texttt{x.}\lambda\texttt{y.y})\ (\mathsf{WWW})\ \mathsf{I} \xrightarrow{*} (\lambda\texttt{x.}\lambda\texttt{y.y})\ (\mathsf{WWWW})\ \mathsf{I} \xrightarrow{*} \ldots$$

Diese Kette erreicht niemals eine Normalform. □

[59] In der Sprache der theoretischen Informatik heißt dies: "die Menge der total-rekursiven Funktionen ist nicht rekursiv-aufzählbar"

Wie kann man eine Normalform finden, wenn es eine gibt?

Wir wissen nun, daß die Bestimmung einer Normalform von der Reduktionsstrategie abhängen kann. Die Frage, die sich daraus unmittelbar ergibt, ist, ob es denn wenigstens eine einheitliche Strategie gibt, mit der man eine Normalform finden kann, wenn es sie gibt? Die Beantwortung dieser Frage ist von fundamentaler Bedeutung für die praktische Verwendbarkeit des λ-Kalküls als Programmiersprache. Ohne eine Reduktionsstrategie, mit der man garantiert eine Normalform auch finden kann, wäre der λ-Kalkül als Grundlage der Programmierung unbrauchbar. Glücklicherweise kann man diese Frage positiv beantworten

Lemma 2.3.33 (Leftmost-Reduktion)

Reduziert man in einem λ-Term immer das jeweils am meisten links stehende (äußerste) Redex, so wird man eine Normalform finden, wenn der Term normalisierbar ist.

Intuitiv läßt sich der Erfolg dieser Strategie wie folgt begründen. Der Beweis von Lemma 2.3.32 hat gezeigt, daß normalisierbare Terme durchaus Teilterme enthalten können, die nicht normalisierbar sind. Diese Teilterme können nun zur Bestimmung der Normalform nichts beitragen, da ihr Wert ja nicht festgestellt werden kann. Wenn wir daher die äußerste Funktionsanwendung zuerst reduzieren, werden wir feststellen, ob ein Teilterm überhaupt benötigt wird, bevor wir ihn unnötigerweise reduzieren.

Die Strategie, zuerst die Funktionsargumente einzusetzen, bevor sie deren Wert bestimmt, entspricht der *call-by-name* Auswertung in Programmiersprachen. Sie ist die sicherste Reduktionsstrategie, aber die Sicherheit wird oft auf Kosten der Effizienz erkauft. Es mag nämlich sein, daß durch die Reduktion ein Argument verdoppelt wird und daß wir es somit zweimal reduzieren müssen. Eine *call-by-value* Strategie hätte uns diese Doppelarbeit erspart, aber diese können wir nur anwenden, wenn wir wissen, daß sie garantiert terminiert. Da das Halteproblem jedoch unentscheidbar ist, gibt es leider keine Möglichkeit, dies für beliebige λ-Terme im Voraus zu entscheiden.[60] Ein präziser Beweis für diese Aussage ist verhältnismäßig aufwendig. Wir verweisen daher auf Lehrbücher über den λ-Kalkül wie [Barendregt, 1981, Stenlund, 1972] für Details.

Ist die Normalform eines λ-Terms eindeutig?

Auch hierauf benötigen wir eine positive Antwort, wenn wir den λ-Kalkül als Programmiersprache verwenden wollen. Wenn nämlich bei der Auswertung eines λ-Terms verschiedene Reduktionsstrategien zu verschiedenen Ergebnissen (Normalformen) führen würden, dann könnte man den Kalkül zum Rechnen und zum Schließen über Programme nicht gebrauchen, da er nicht eindeutig genug festlegt, was der Wert eines Ausdrucks ist.[61]

Rein syntaktisch betrachtet ist der Reduktionsmechanismus des λ-Kalküls ein *Termersetzungssystem*, welches Vorschriften angibt, wie Terme in andere Terme umgeschrieben werden dürfen (engl. *rewriting*). In der Denkweise der Termersetzung ist die Frage nach der Eindeutigkeit der Normalform ein Spezialfall der Frage nach der *Konfluenz* eines Regelsystems, also der Frage, ob zwei Termersetzungsketten, die im gleichen Term begonnen haben, wieder zusammengeführt werden können.

Die genauen Definitionen der Konfluenz werden wir im Abschnitt 2.4.5.3 geben, wo wir für typisierbare λ-Terme ein Konfluenztheorem beweisen werden. Auch für den uneingeschränkten λ-Kalkül kann man Konfluenz beweisen. Der Beweis dieses Theorems, das nach den Mathematikern A. Church und B. Rosser benannt wurde, ist allerdings relativ aufwendig. Für Details verweisen wir daher wiederum auf Lehrbücher wie [Barendregt, 1981, Hindley & Seldin, 1986, Stenlund, 1972].

Satz 2.3.34 (Church-Rosser Theorem)

Es seien t, u und v beliebige λ-Terme und es gelte $t \xrightarrow{*} u$ und $t \xrightarrow{*} v$.
Dann gibt es einen λ-Term s mit der Eigenschaft $u \xrightarrow{*} s$ und $v \xrightarrow{*} s$.

[60] Diese Möglichkeit wird uns nur bei typisierbaren λ-Termen geboten, die wir im folgenden Abschnitt besprechen.

[61] Rein theoretisch gäbe es aus einem solchen Dilemma immer noch einen Ausweg. Man könnte den Kalkül von vorneherein mit einer Reduktionsstrategie koppeln und hätte dann eine eindeutige Berechnungsvorschrift. Auf den Kalkül zum Schließen über Programme hätte dies aber negative Auswirkunge, da wir auch hier die Reduktionsstrategie mit einbauen müßten.

Eine unmittelbare Konsequenz dieses Theorems ist, daß die Normalformen eines gegebenen λ-Terms bis auf α-Konversionen eindeutig bestimmt sind und daß der in Abbildung 2.7 auf Seite 58 angegebenen Kalkül zum Schließen über die Gleichheit von λ-Termen korrekt ist im Sinne von Definition 2.3.24.

Korollar 2.3.35

Es seien t, u und v beliebige λ-Terme.

1. *Wenn u und v Normalformen von t sind, dann sind sie kongruent im Sinne von Definition 2.3.9.*
2. *Der Kalkül zum Schließen über die Gleichheit ist korrekt: Wenn $u=v$ bewiesen werden kann, dann gibt es einen λ-Term s mit der Eigenschaft $u \xrightarrow{*} s$ und $v \xrightarrow{*} s$.*
3. *Gilt $u=v$ und v ist in Normalform, so folgt $u \xrightarrow{*} v$.*
4. *Gilt $u=v$, so haben u und v dieselben Normalformen oder überhaupt keine.*
5. *Wenn u und v in Normalform sind, dann sind sie entweder kongruent oder nicht gleich.*

Es ist also legitim, λ-Terme als Funktionen anzusehen und mit dem in Abbildung 2.7 angegebenen Kalkül Schlüsse über die Resultate von Berechnungen zu ziehen.

Welche extensionalen Eigenschaften von λ-Termen kann man automatisch beweisen?

Die bisherigen Fragestellungen richteten sich im wesentlichen auf den Reduktionsmechanismus des λ-Kalküls. Wir wollen mit dem λ-Kalkül jedoch nicht nur rechnen, sondern auch extensionalen Eigenschaften von Programmen – also das nach außen sichtbare Verhalten – mit formalen Beweismethoden untersuchen. Die Frage, welche dieser Eigenschaften mithilfe fester Verfahren überprüft werden können, liegt also nahe.

Leider gibt uns die Rekursionstheorie (siehe z.B. [Rogers, 1967]) auf diese Frage eine sehr negative Antwort. Der Preis, den wir dafür bezahlen, daß der λ-Kalkül genauso mächtig ist wie die rekursiven Funktionen, ist – neben der Notwendigkeit, auch partielle Funktionen zu betrachten – die Unentscheidbarkeit *aller* extensionaler Eigenschaften von Programmen.

Satz 2.3.36 (Satz von Rice)

Keine nichttriviale extensionale Eigenschaft von λ-Termen kann mithilfe eines allgemeinen Verfahrens entschieden werden.

Die Bedeutung dieser Aussage wird klar, wenn wir uns ein paar typische Fragen ansehen, die man gerne mit einem festen Verfahren entscheiden können würde:

- *Terminiert die Berechnung einer Funktion f bei Eingabe eines Argumentes x?* (Halteproblem)
- *Ist die durch einen λ-Term f beschriebene Funktion total?*
- *Gehört ein Wert y zum Wertebereich einer durch einen λ-Term f beschriebenen Funktion?*
- *Berechnet die durch einen λ-Term f beschriebene Funktion bei Eingabe von x den Wert y?*
- *Sind zwei λ-berechenbare Funktionen f und g gleich?* (Dies ist nicht einmal beweisbar)

Die Liste könnte beliebig weiter fortgesetzt werden, da der Satz von Rice tatsächlich besagt, daß nicht eine einzige interessante Fragestellung mit einem universellen Verfahren entschieden werden kann. Aus diesem Grunde ist der allgemeine λ-Kalkül zum Schließen über Programme und ihre Eigenschaften ungeeignet. Essentiell für eine derart negative Antwort ist dabei jedoch die Forderung nach einem *universellen* Verfahren, das für jede beliebige Eingabe die Frage beantwortet. Verzichtet man jedoch auf die Universalität und schränkt die zu betrachtenden λ-Terme durch syntaktische Mittel ein, so kann es durchaus möglich sein, für diese kleinere Klasse von Programmen Entscheidungsverfahren für die wichtigsten Fragen zu konstruieren.

2.4 Die einfache Typentheorie

Im vorhergehenden Abschnitt haben wir gesehen, daß der λ-Kalkül einen einfachen und zugleich sehr mächtigen Formalismus zum Schließen über Programme darstellt. Die große Ausdruckskraft bringt jedoch eine Reihe von Problemen mit sich, welche eine Automatisierung des Schlußfolgerungsprozesses erheblich erschweren. Es gibt keine natürlichen mengentheoretischen Interpretationen für λ-Terme; keine extensionale Eigenschaft von λ-Termen kann vollautomatisch entschieden werden; in formalen Beweisen müssen wir mit partiellen Funktionen rechnen und bei totalen Funktionen hängt die Terminierung von Reduktionen immer noch davon ab, an welcher Stelle wir reduzieren.

Viele dieser Probleme haben ihre Ursache in der Universalität des λ-Kalküls, die in der Programmierpraxis gar nicht voll ausgeschöpft wird. Operationen wie ein universeller Fixpunktkombinator tauchen in realistischen Programmen niemals auf und werden nur benötigt, um komplexere Berechnungs*mechanismen* innerhalb des λ-Kalküls zu *erklären*. Es ist also durchaus möglich, auf Universalität zu verzichten, ohne daß der Kalkül – im Hinblick auf praktische Verwendbarkeit – an Ausdruckskraft verliert. Unser Ziel ist daher, durch syntaktische Einschränkungen des ansonsten sehr gut geeigneten λ-Kalküls ein schwächeres Berechenbarkeitsmodell zu entwickeln, welches ermöglicht, wichtige Eigenschaften der betrachteten Programme automatisch zu beweisen, und dennoch ausdrucksstark genug ist, alle in der Praxis auftretenden Probleme zu handhaben.

Der zentrale Grund für die Unentscheidbarkeit von Programmeigenschaften im λ-Kalkül ist die Tatsache, daß λ-Terme auf sich selbst angewandt werden dürfen.[62] Es ist möglich, einen λ-Term ganz anders zu verwenden als für den Zweck, für den er ursprünglich geschaffen wurde. So ist es zum Beispiel legitim, das λ-Programm $\overline{2}\,\overline{2}$ aufzustellen und zu $\overline{4}$ auszuwerten. Auch sind die λ-Terme für $\overline{0}$, **T** und [] (vgl. Abschnitt 2.3.3) identisch und somit könnte man auch Ausdrücke wie **add T []** zu $\overline{0}$ auswerten, was semantisch ziemlich unsinnig ist. Die uneingeschränkte Anwendbarkeit von λ-Termen auf Argumente ist – wie in Abschnitt 2.3.6 angedeutet – auch der Grund dafür, daß es keine einfachen mengentheoretischen Modelle für den λ-Kalkül geben kann. Um diese Probleme zu lösen, liegt es nahe, einen Formalismus zu entwickeln, durch den die Anwendbarkeit von λ-Termen auf sinnvolle Weise beschränkt werden kann.

Mathematisch besehen ist die Unentscheidbarkeit von Programmeigenschaften verwandt mit dem Russelschen Paradox der frühen Mengentheorie (siehe Fußnote 57 auf Seite 62). So wie die Möglichkeit einer *imprädikativen* Beschreibung von Mengen[63] in der Mengentheorie zu paradoxen Situationen führt, so ermöglicht die Selbstanwendbarkeit in der Programmierung die Konstruktion von Diagonalisierungsbeweisen, welche die Entscheidbarkeit von Programmeigenschaften ad absurdum führen. In seiner *Theorie der Typen* [Russel, 1908] hat Russel bereits im Jahre 1908 als Lösung vorgeschlagen, durch eine *Typdisziplin* jedem mathematischen Symbol einen Bereich zuzuordnen, zu dem es gehören soll. Diese Typdisziplin verbietet die Bildung von Mengen der Art $\{X \mid X \not\in X\}$, da das Symbol '$\in$' nunmehr Objekte verschiedener Typen in Beziehung setzt.

In gleicher Weise kann man den λ-Kalkül durch eine Typdisziplin ergänzen, welche eine stärkere Strukturierung von λ-Programmen bereits auf der syntaktischen Ebene ermöglicht. Im *getypten λ-Kalkül* [Church, 1940] werden Typen dazu benutzt, um die Menge der zulässigen λ-Terme syntaktisch stärker einzuschränken als dies im allgemeinen Kalkül gemäß Definition 2.3.2 der Fall ist. Darüber hinaus dienen sie aber auch als Bezeichner für die Funktionenräume, zu denen ein getypter λ-Term gehören soll. Die Typdisziplin verbietet viele der seltsam anmutenden Konstruktionen[64] des allgemeinen λ-Kalküls und sorgt somit dafür, daß viele Eigenschaften getypter λ-Terme algorithmisch entscheidbar werden.

[62] Im Konzept der Turingmaschinen und imperativen Programmiersprachen entspricht die Selbstanwendbarkeit der Existenz einer universellen Maschine, auf die man ungern verzichten möchte.

[63] Unter einer *imprädikativen* Beschreibung versteht man die uneingeschränkte abstrakte Definition einer Menge M durch $M = \{x \mid x \text{ hat Eigenschaft } P\}$.

[64] Die Tendenz in modernen Programmiersprachen geht – wenn auch verspätet – in die gleiche Richtung. Während es in vielen älteren Programmiersprachen wie Fortran, C, Lisp, etc. erlaubt ist, Variablen völlig anders zu verwenden als ursprünglich vorgesehen (man darf z.B. einen Integer-Wert als boolesche Variable oder als Buchstaben weiterverwenden, wenn es der Effizienz dient), haben fast alle neueren Programmiersprachen wie Pascal, Eiffel, ML, etc. ein strenges Typsystem, welches derartige Mißbräuche verbietet. Diese Typdisziplin bringt zwar einen gewissen Mehraufwand bei der Programmierung mit sich, sorgt aber auch für einen strukturierteren und übersichtlicheren Programmierstil.

Für einen formalen Kalkül zum Schließen über Programmeigenschaften bietet eine Typdisziplin aber noch mehr als nur eine Einschränkung von Termen auf solche, die natürliche Modelle und ein automatisiertes Schließen zulassen. Der Typ eines Terms kann darüber hinaus auch als eine Beschreibung der *Eigenschaften* des Terms angesehen werden und in diese Beschreibung könnte man weit mehr aufnehmen als nur die Charakterisierung der Mengen, auf denen der Term operiert. Eine Typisierung der Art

$$\texttt{sqrt x} \in \{\texttt{y:}\mathbb{N} \mid \texttt{y*y} \le \texttt{x < (y+1)*(y+1)}\}$$

würde zum Beispiel festlegen, daß die Funktion `sqrt` für jedes `x` eine (eindeutig bestimmte) Integerquadratwurzel von `x` bestimmt. Die Typdisziplin bietet also eine einfache Möglichkeit an, syntaktisch über den Bezug zwischen einem Programm und seiner *Spezifikation* zu schließen.

Es gibt zwei Wege, eine Typdisziplin über λ-Termen aufzubauen. Die ursprüngliche Vorgehensweise von Church's *getyptem λ-Kalkül* ('typed λ-calculus', siehe [Church, 1940]) nimmt die Typen direkt in die Terme mit auf – wie zum Beispiel in $\lambda \texttt{f}^{\texttt{S}\to\texttt{T}}.\lambda \texttt{x}^{\texttt{S}}.\,\texttt{f}^{\texttt{S}\to\texttt{T}}\,\texttt{x}^{\texttt{S}}$. Im wesentlichen muß man hierzu die Definition 2.3.2 der λ-Terme auf Seite 48 in eine für *getypte λ-Terme abwandeln*.[65] Ein zweiter Ansatz betrachtet λ-Terme und Typen als zwei unabhängige Konzepte. Dies bedeutet, daß der λ-Kalkül unverändert übernommen wird und eine (einfache) *Typentheorie* als separater Kalkül hinzukommt, mit dem nachgewiesen werden kann, daß ein λ-Term *typisierbar* ist, also zu einem bestimmten Typ gehört. In diesem Ansatz würde man zum Beispiel $\lambda \texttt{f}.\lambda \texttt{x}.\,\texttt{f}\,\texttt{x} \in (\texttt{S}\to\texttt{T}) \to \texttt{S} \to \texttt{T}$ nachweisen.

Beide Vorgehensweisen führen im Endeffekt zu den gleichen Resultaten, obwohl sie von ihrem Ansatz her recht verschieden sind. So werden im getypten λ-Kalkül die einzelnen Variablen getypt, während die Typentheorie jeweils nur einem gesamten Ausdruck einen Typ zuweist. Wir werden daher im die Begriffe *getypter λ-Kalkül* und *Typentheorie* als synonym ansehen und nicht weiter unterscheiden..

Aus praktischer Hinsicht ist der Weg der Typentheorie sinnvoller, da er bei der Formulierung und Berechnung von Programmen die volle Freiheit des λ-Kalküls behält und die Typisierung als ein statisches Konzept hinzunimmt, welches nur bei der Beweisführung hinzugenommen wird. Damit bleibt die Syntax von Programmen unkompliziert, da sie nicht immer wieder durch Typinformationen überfrachtet wird.[66] Erkauft wird diese Freiheit durch einen gewissen Mehraufwand bei der Beweisführung, da für einzelne Teilausdrücke eine Typisierung rekonstruiert werden muß, die man im getypten λ-Kalkül direkt aus dem Programm ablesen könnte (weil man sie dort bereits angeben *mußte*).

Wir werden im folgenden nun die sogenannte *Theorie der einfachen Typen* (*simply typed λ-calculus*) vorstellen, welche nur diejenigen Typisierungskonzepte enthält, die für eine natürliche Interpretation von λ-Termen unumgänglich sind. Wir werden am Ende dieses Abschnitts sehen, daß wir zugunsten einer hinreichenden Ausdruckskraft noch weitere Typkonstrukte hinzunehmen müssen, aber die hier entwickelten Konzepte fortführen können. Bevor wir die formalen Details besprechen, wollen wir zwei grundsätzliche Fragen klären.

Was sind die Charakteristika eines Typs?

Typen können in erster Näherung als das syntaktische Gegenstück zu Mengen angesehen werden. In der Mathematik ist eine Menge eindeutig dadurch charakterisiert, daß man angibt, welche *Elemente* zu dieser Menge gehören. So besteht zum Beispiel die Menge $\mathbb{N}$ der natürlichen Zahlen aus den Elementen `0,1,2,3,4,...`, die Menge $\mathbb{Z}$ der ganzen Zahlen aus den Elementen `0,1,-1,2,-2,...`, die Menge $\mathbb{Q}$ der Rationalzahlen aus $\texttt{0,1,-1,}\frac{1}{2}\texttt{,-}\frac{1}{2}\texttt{,2,-2,...}$ und die Menge $\mathbb{B}$ der booleschen Werte aus wahr und falsch.

Etwas komplizierter wird es, wenn man Mengen*konstruktoren* wie das Produkt $M \times M'$ zweier Mengen M und M', den Funktionenraum $M \to M'$ oder den Raum M `list` aller Listen von Elementen aus M charakterisieren möchte. In diesem Falle ist es nicht möglich, die konkreten Elemente anzugeben. Man muß sich daher

[65] Aus diesem Ansatz stammt der Begriff der *getypten Variablen*, den wir benutzen werden, um auszudrücken, daß eine Variable ein Platzhalter für Terme eines bestimmten Typs ist. In $\lambda \texttt{f}^{\texttt{S}\to\texttt{T}}.\lambda \texttt{x}^{\texttt{S}}.\texttt{f}^{\texttt{S}\to\texttt{T}}\,\texttt{x}^{\texttt{S}}$ ist z.B. die Variable `f` eine Variable des Typs $\texttt{S}\to\texttt{T}$.

[66] Moderne Programmiersprachen gehen den gleichen Weg. Das Typkonzept steht außerhalb des eigentlichen auszuführenden Programms und wird statisch vom Compiler überprüft. Hierdurch entfällt die Notwendigkeit von Kontrollen zur Ausführungszeit, wodurch ein Programm deutlich effizienter wird.

damit behelfen, zu beschreiben, wie die Elemente dieser Mengen zu *bilden* sind. So ist zum Beispiel $M \times M'$ genau die Menge aller Paare (a,b), wobei a ein Element von M und b eines von M' ist.

Genau besehen haben wir auch bei der Charakterisierung der Mengen $\mathbb{N}$, $\mathbb{Z}$ und $\mathbb{Q}$ nur textliche Beschreibungen der Elemente verwendet und nicht etwa die Elemente selbst. So benutzen wir bei den größeren natürlichen Zahlen wie `12` eine Aneinanderreihung von Symbolen. Bei den ganzen Zahlen erscheint das Symbol '`-`', um negative Zahlen zu beschreiben. Bei den Rationalzahlen behelfen wir uns mit der Bruchschreibweise, wobei es sogar vorkommen kann, daß wir dasselbe Element auf verschiedene Arten beschreiben, wie etwa durch $\frac{129}{21}$ und $\frac{86}{14}$. Bei den Rationalzahlen hat man sich darauf geeinigt, unter allen Darstellungsmöglichkeiten einer Zahl diejenige besonders herauszuheben, die sich nicht weiter (durch Kürzen) vereinfachen läßt. Eine solche standardisierte Darstellung nennt man auch *kanonisch* und man spricht der Einfachheit halber von den kanonischen Elementen der Menge $\mathbb{Q}$.

Bei den bisherigen Betrachtungen haben wir uns hauptsächlich mit den mathematischen Aspekten von Typen beschäftigt. Typen spielen jedoch auch in der Programmierung eine wesentliche Rolle. Mithilfe einer Typdisziplin kann man statisch überprüfen, ob die Anwendung von Operationen wie `+`,`*`,`-`,`/`,$\wedge$, $\vee$ auf bestimmte Objekte überhaupt sinnvoll ist und was sie im Zusammenhang mit diesem Objekt überhaupt bedeutet. So wird zum Beispiel ein Compiler Ausdrücke wie `1` $\vee$ `5` oder a`/`b als unzulässig erklären, wenn a und b als boolesche Werte deklariert sind. Ausdrücke wie `1+2` und $\frac{1}{2}+\frac{2}{3}$ sind dagegen zulässig, werden jedoch trotz des gleichen Operationssymbols zur Ausführung verschiedener Operationen führen. Die zulässigen Operationen auf den Elementen eines Datentyps müssen daher als untrennbarer Bestandteil dieses Typs angesehen werden. Wir fassen diese Erkenntnis als einen ersten wichtigen Grundsatz zusammen:

Ein Typ besteht aus kanonischen Elementen und zulässigen Operationen auf seinen Elementen.

Die kanonischen Elemente eines Typs geben an, wie Elemente zu *bilden* sind. Die zulässigen Operationen dagegen beschreiben, auf welche Arten Elemente für eine weitere Verarbeitung *verwendet* werden dürfen.[67]

Diese Charakterisierung von Typen ist zum Teil noch semantischer Natur. Da wir Typen jedoch als Bestandteil von Kalkülen verwenden wollen, die sich beim logischen Schließen ausschließlich auf die Syntax eines Ausdrucks stützen können, müssen wir den obigen Grundsatz mithilfe von syntaktischen Konzepten neu formulieren. Dabei stützen wir uns auf den Gedanken, daß jede Operation – ob sie kanonische Elemente erzeugt oder Elemente verwendet – durch einen Term beschrieben wird. Alles, was wir zu tun haben, ist, diese Terme im Zusammenhang mit dem Typkonzept zu klassifizieren in *kanonische Terme des Typs* – also Terme, die als Standardbeschreibung von Elementen gelten *sollen* – und in *nicht-kanonische Terme des Typs* – also Terme, die Elemente eines Typs als Bestandteile verwenden.[68] Für die Beschreibung eines Typs ist es natürlich auch notwendig, ihn zu benennen, wobei wir ebenfalls einen syntaktischen (Typ-)Ausdruck verwenden müssen. Insgesamt erhalten wir also:

Ein Typ wird definiert durch einen Typ-Ausdruck und seine kanonischen und nichtkanonischen Terme.

Welche Arten von Typen sind zu betrachten?

Wir wollen die Typdisziplin in erster Linie dazu verwenden, um eine natürliche Semantik für λ-Terme zu ermöglichen. λ-Terme sollen als Funktionen eines geeigneten Funktionenraumes interpretiert werden können. Wir benötigen also ein Konstrukt der Form $S{\rightarrow}T$, welches den Typ aller Funktionen vom Typ S in den Typ T bezeichnet. Da λ-Terme nur auf zwei Arten gebildet werden können, ist auch die Aufteilung in kanonische und nichtkanonische Terme einfach. Alle λ-Abstraktionen der Form $\lambda x\,.\,t$ gelten als kanonisch[69] während alle Applikationen der Form $f\ t$ als nichtkanonische Terme eingestuft werden.

[67] In modernen Programmiersprachen, in denen sich die objektorientierte Denkweise immer weiter durchsetzt, wird dieser Gedanke durch das Konzept der Klasse (Modul Unit o.ä.) realisiert. Hier wird angegeben, welche Operationen zulässig sind, um Elemente der Klasse zu erzeugen bzw. zu verändern, und mit welchen Operationen man auf sie zugreifen kann.

[68] Diese Trennung in kanonische und nichtkanonische Terme entspricht, wie wir später deutlicher sehen werden, der Auftrennung eines Kalküls in Einführungs- und Eliminationsregeln. Einführungsregeln sagen, wie kanonische Elemente eines Typs (Beweise einer Formel) aufzubauen sind, während Eliminationsregel sagen, wie man sie (nichtkanonisch) verwendet.

Außer dem Funktionenraumkonstruktor werden für den λ-Kalkül keine weiteren Typkonstrukte benötigt. Wir erhalten daher zunächst eine sehr einfache Typentheorie. Wir werden natürlich versuchen, für die im Abschnitt 2.3.3 angegebenen Simulationen gängiger Programmierkonstrukte auch entsprechende Typkonstrukte als definitorische Erweiterungen der einfachen Typentheorie zu geben. Dies ist aber, wie wir im Abschnitt 2.4.8 zeigen werden, nur bis zu einer gewissen Grenze möglich.

Wir werden im folgenden nun präzisieren, wie Typkonstrukte zu bilden sind, wie einem λ-Term ein geeigneter Typ zugeordnet werden kann, wie man formal beweisen kann, daß eine solche Zuweisung korrekt ist, und welche Eigenschaften typisierbare λ-Terme besitzen.

2.4.1 Syntax

Die Syntax der einfachen Typentheorie ist denkbar einfach zu definieren, da es nur einen einzigen Konstruktor gibt. In ihrem Aufbau folgt sie dem aus der Prädikatenlogik und dem λ-Kalkül bekannten Schema.

Definition 2.4.1 (Typ-Ausdrücke)

Es sei $\underline{\mathcal{V}}$ ein Alphabet von Variablen(-symbolen) und $\underline{\mathcal{T}}$ ein Alphabet von Bereichssymbolen (Typen).

Typ-Ausdrücke – kurz Typen – sind induktiv wie folgt definiert.

- *Jede Variable $\underline{T} \in \mathcal{T}$ ist ein (atomarer) Typ.*
- *Sind S und T beliebige Typen, so ist auch $\underline{S{\rightarrow}T}$ ein Typ.*
- *Ist T ein beliebiger Typ, dann ist $\underline{(T)}$ ein Typ*

Der Operator $\rightarrow$ ist rechtsassoziativ

Objekt-Ausdrücke sind λ-Terme im Sinne von Definition 2.3.2

- *Jede Variable $\underline{x} \in \mathcal{V}$ ist ein λ-Term.*
- *Ist $x \in \mathcal{V}$ eine Variable und t ein beliebiger λ-Term, dann ist $\underline{\lambda x\,.\,t}$ ein λ-Term.*
- *Sind t und f beliebige λ-Terme, dann ist $\underline{f\ t}$ ein λ-Term.*
- *Ist t ein beliebiger λ-Term, dann ist $\underline{(t)}$ ein λ-Term.*

Die Applikation bindet stärker als λ-Abstraktion und ist linksassoziativ.

Man beachte hierbei wiederum, daß Variablen- und Typsymbole nicht unbedingt aus einzelnen Buchstaben bestehen müssen und daß sich die Alphabete $\mathcal{V}$ und $\mathcal{T}$ überlappen dürfen. Meist geht es aus dem Kontext hervor, um welche Art von Symbolen es sich handelt. Um die Unterscheidung zu erleichtern, benutzen wir Großbuchstaben wie `S,T,X` etc. für Typsymbole und Kleinbuchstaben wie `x,y,z,f,g` für Objektvariablen.

Die Rechtsassoziativität des Funktionenraumkonstruktes und die Linksassoziativität der Applikation ergibt sich aus dem gewünschten Zusammenhang zwischen Typen und λ-Termen. Ein ungeklammerter Typ der Form $S_1{\rightarrow}S_2{\rightarrow}T$ sollte zu einem ungeklammerten λ-Term der Form $\lambda x_1\,.\,\lambda x_2\,.\,t$ passen, der notwendigerweise rechtsassoziativ zu klammern ist. Wendet man nun eine Funktion $f \in S_1{\rightarrow}S_2{\rightarrow}T$ auf zwei Argumente t und u an, so erwartet man, daß t ein Element von S_1, u eines von S_2 ist und daß als Ergebnis ein Element von T herauskommt. Ein rechtsassoziative Klammerung $f\ (t(u))$ würde dagegen verlangen, daß $t(u)$ ein Element von $S_1{\rightarrow}S_2$ ist, was nur zu einer Linksassoziativität von $\rightarrow$ paßt.

[69] Es macht hierbei wenig Sinn, zu fordern, daß t in Normalform sein muß, da dies die Auftrennung zwischen kanonischen und nichtkanonischen Termen sehr verkomplizieren würde. Wir werden in späteren Abschnitten und besonders im Kapitel 3 sehen, daß diese grobe Einteilung auch wichtig für eine effektive Begrenzung von Reduktionsstrategien (*lazy evaluation*) ist, ohne die ein formales Schließen über Programme ständig durch irrelevante Terminierungsprobleme belastet würde.

2.4.2 Semantik und Typzugehörigkeit

Bei der Definition der Typ- und Objekt-Ausdrücke hatten wir eine sehr einfache und natürliche Semantik vor Augen, die wir im folgenden nur informal beschreiben wollen. Typausdrücke sollen *Mengen* beschreiben und das Funktionssymbol $\rightarrow$ soll, wie in der gewöhnlichen Mathematik, einen *Funktionenraum* charakterisieren.

In Anlehnung an Definition 2.2.8 (Seite 26) interpretieren wir jedes Typsymbol $T \in \mathcal{T}$ durch eine Teilmenge $\iota(T) \subseteq \mathcal{U}$. Jeder Funktionentyp $S \rightarrow T$ wird dann durch den entsprechenden Funktionenraum interpretiert:

$$\iota(S \rightarrow T) = \iota(S) \rightarrow \iota(T)$$

Im folgenden werden wir λ-Termen einen Typ zuordnen. Terme, bei denen dies möglich ist, lassen sich ebenfalls sehr leicht interpretieren. Wird einem Term t der Typ T zugeordnet, dann können wir t als ein Element $\iota(t) \in \iota(T)$ interpretieren.

Dieser semantische Zusammenhang läßt sich verhältnismäßig einfach sicherstellen. Jede Variable x muß zu einem Typ T gehören. Jede λ-Abstraktion $\lambda x.t$ muß zu einem Raum $S \rightarrow T$ gehören, wobei x zu S und t zu T gehört. Auf diese Art wird garantiert, daß $\lambda x.t$ als Funktion eines entsprechenden Funktionenraums $\iota(S) \rightarrow \iota(T)$ interpretiert werden kann. Schließlich gehört eine Applikation $f\,t$ zum Typ T, wenn t zu einem Typ S gehört und f zu $S \rightarrow T$. In diesem Fall wird t immer durch einen Wert interpretiert, welcher im Definitionsbereich der durch f dargestellten Funktion liegt. Wir wollen wir diesen Zusammenhang zunächst an einem Beispiel erläutern.

Beispiel 2.4.2

Wir versuchen, dem Term λ`f.`λ`x.f x` einen Typ zuzuordnen.

Die Variable `x` muß zu einem Typ gehören, den wir der Einfachheit halber zunächst mit `S` bezeichnen. Da `f` auf `x` angewandt wird, muß `f` einen Typ der Gestalt `S`$\rightarrow$`T` haben, wobei `T` ebenfalls noch nicht näher festgelegt ist. Insgesamt hat der Ausdruck `f x` dann den Typ `T`.

Nun haben wir die Typen der Variablen `f` und `x` im Prinzip bereits festgelegt. Damit ergibt sich `S`$\rightarrow$`T` als Typ des Ausdrucks λ`x. f x` und `(S`$\rightarrow$`T)` $\rightarrow$ `(S`$\rightarrow$`T)` als Typ des gesamten Ausdrucks.

Dieses Beispiel zeigt, daß der Typ eines λ-Terms keineswegs eindeutig festliegt. Abgesehen davon, daß wir statt `S` und `T` auch andere Variablennamen hätten verwenden können, könnten wir an ihrer Stelle auch komplexere Typausdrücke einsetzen. Es wäre durchaus korrekt, dem Ausdruck λ`f.`λ`x.f x` den Typ

$$(S \rightarrow (T \rightarrow S)) \rightarrow (S \rightarrow (T \rightarrow S)))$$

oder den Typ

$$(S \rightarrow S) \rightarrow (S \rightarrow S)$$

zuzuordnen. Beide Ausdrücke sind aber unnötig spezialisiert: der erste gibt zu viel Struktur vor und der zweite identifiziert die Ein- und Ausgabetypen von `f` miteinander. Üblicherweise versucht man, einem λ-Term das einfachste (allgemeinste!) aller Typschemata zuzuordnen, also eines, was durch seine einfache Struktur noch die meisten Freiheitsgrade läßt. Ein solches Schema nennt man *prinzipielles Typschema*.

Wir haben am Ende des vorigen Abschnitts gesehen, daß es nicht möglich ist, der Klasse aller λ-Terme eine einfache mengentheoretische Sematik zu geben. Da Typen aber eine einfache Semantik geradezu suggerieren, muß es λ-Terme geben, die nicht *typisierbar* sind. Auch hierfür wollen wir ein Beispiel geben.

Beispiel 2.4.3

Wir betrachten den Term λ`x. x x`.

Die Variable `x` muß zu einem Typ S gehören. Da `x` aber auch auf `x` angewandt wird, muß `x` einen Typ der Gestalt $S \rightarrow T$ haben. Um also dem Term λ`x. x x` einen Typ zuordnen zu können, müssen wir Typen S und T bestimmen, welche die Gleichung $S = S \rightarrow T$ erfüllen. Da aber (semantisch) keine Menge identisch ist mit ihrem eigenen Funktionenraum, wird dies nicht möglich sein.

Nach diesen Beispielen dürfte die präzise Definition der Typzugehörigkeit recht naheliegend sein.

Definition 2.4.4 (Typzugehörigkeit)

Typzugehörigkeit ist eine Relation $\in$ zwischen Objekt-Ausdrücken und Typausdrücken, die induktiv wie folgt definiert ist.

- $x \in T$, *falls $x \in \mathcal{V}$ eine Variable und T ein beliebiger Typ ist.*
- $\lambda x.t \in S \rightarrow T$, *falls $t \in T$ gilt, wann immer $x \in S$ ist.*
- $f\,t \in T$, *falls es einen Typ S gibt mit $f \in S \rightarrow T$ und $t \in S$.*
- $(t) \in T$, *falls $t \in T$*
- $t \in (T)$, *falls $t \in T$*

Ein λ-Term t heißt typisierbar, wenn es einen Typ T gibt mit $t \in T$.

Das prinzipielle Typschema eines typisierbaren λ-Terms t ist der kleinste[70] *Typ T, für den $t \in T$ gilt.*

In der obigen Definition haben wir das Konzept der Typzugehörigkeit durch rein syntaktische Mittel beschrieben. Dies erlaubt uns, Typzugehörigkeit durch symbolische Manipulationen wie z.B. durch den im Abschnitt 2.4.3 vorgestellten Kalkül zu überprüfen. Dennoch hängt der Typ eines λ-Terms t nicht von seinem syntaktischen Erscheinungsbild, sondern nur von seinen *Wert* ab. λ-Terme, die semantisch gleich (im Sinne von Definition 2.3.24) sind, gehören auch zum gleichen Typ. Damit untermauert Definition 2.4.4 tatsächlich die intuitive Interpretation von λ-Termen als Funktionen eines bestimmten Funktionenraumes.

Satz 2.4.5

Es seien t und t' beliebige typisierbare λ-Terme und T ein Typausdruck.

Wenn t und t' semantisch gleich sind, dann gilt $t \in T$ genau dann, wenn $t' \in T$ gilt.

Beweis:

Wir zeigen zunächst, daß die Typzugehörigkeit bei Reduktionen erhalten bleibt.

Aus $t \xrightarrow{\beta} t'$ folgt: $t \in T$ gilt genau dann, wenn $t' \in T$ gilt.

Wir beweisen dies durch eine Induktion über die Termstruktur (Tiefe) von t

- Ist t ein Term der Tiefe 1, so ist t eine Variable $x \in \mathcal{V}$. Da t bereits in Normalform ist, muß ein Term t' mit $t \xrightarrow{*} t'$ α-konvertibel[71] zu t, also ebenfalls eine Variable sein. Gemäß Definition 2.4.4 gilt dann $t \in T$ und $t' \in T$ für beliebige Typausdrücke T.
- Es sei für alle Terme u der Tiefe n, alle λ-Terme u' und alle Typen S gezeigt
  aus $u \xrightarrow{\beta} u'$ folgt: $u \in S$ gilt genau dann, wenn $u' \in S$ gilt.
- Es sei t' ein Term der Tiefe $n+1$ und es gelte $t \xrightarrow{*} t'$.
  - Falls t die Gestalt $\lambda x.u$ hat, so muß t' die Form $\lambda x.u'$ haben, wobei u' ein λ-Term mit $u \xrightarrow{\beta} u'$ ist. Gilt nun $t \in T$, so muß T von der Form $S_1 \rightarrow S_2$ sein und es ist $u \in S_2$, wann immer $x \in S_1$. Aufgrund der Induktionsannahme gilt somit $u' \in S_2$, wann immer $x \in S_1$ ist und somit $t' \equiv \lambda x.u' \in S_1 \rightarrow S_2 \equiv T$. Da die gleiche Argumentationskette auch umgekehrt geführt werden kann, folgt: $t \in T$ gilt genau dann, wenn $t' \in T$ gilt.
  - Falls t die Gestalt $(\lambda x.u)\ v$ hat und t' durch Reduktion des äußeren Redex entsteht, so ist $t' \equiv u[v/x]$. Gilt nun $t \in T$, so gibt es einen Typ S mit $\lambda x.u \in S \rightarrow T$ und $v \in S$. Nach Definition 2.4.4 ist nun $u \in T$, wann immer $x \in S$ gilt. Wegen $v \in S$ folgt hieraus $t' \equiv u[v/x] \in T$.
    Ist umgekehrt $t' \equiv u[v/x] \in T$, so gibt es einen Typ S mit $v \in S$ und der Eigenschaft, daß $u \in T$ ist, wann immer $x \in S$ gilt. Damit folgt $t \equiv (\lambda x.u)\ v \in T$.

[70] Unter dem kleinsten Typ verstehen wir einen Typ, dessen struktureller Aufbau im Sinne von Definition 2.4.1 mit dem geringsten Aufwand und der geringsten Spezialisierung verbunden ist. Man könnte auch sagen, daß jeder andere Typ T', für den $t \in T'$ gilt, sich durch Instantiierung der Typvariablen aus dem prinzipiellen Typschema ergeben muß.

[71] Genau besehen kann es einen Term t' mit $t \xrightarrow{\beta} t'$ (genau ein Reduktionsschritt!) nicht geben und damit folgt die Behauptung aus der Ungültigkeit der Annahme

– Falls t die Gestalt $f\,u$ hat, und t' nicht durch äußere Reduktion (wie oben) entsteht, so hat t' die Gestalt $f\,u'$, wobei u' ein λ-Term ist, für den $u \xrightarrow{\beta} u'$ gilt, oder die Form $f'\,u$, wobei $f \xrightarrow{\beta} f'$ gilt. Nach Induktionsannahme gehören u und u' (bzw. f und f') nun zum gleichen Typ und wie im ersten Fall folgt die Behauptung

Damit ist die Aussage bewiesen. Durch eine Induktion über die Länge der Ableitung kann man hieraus schließen:
Aus $t \xrightarrow{*} t'$ folgt: $t \in T$ gilt genau dann, wenn $t' \in T$ gilt.

Es seien nun t und t' semantisch gleich. Dann gibt es gemäß Definition 2.3.24 einen λ-Term u mit $t \xrightarrow{*} u$ und $t' \xrightarrow{*} u$. Es folgt: $t \in T$ gilt genau dann, wenn $u \in T$ gilt und dies ist genau dann der Fall, wenn $t' \in T$ gilt. □

2.4.3 Formales Schließen über Typzugehörigkeit

In Definition 2.4.4 haben wir bereits recht genaue Vorschriften dafür angegeben, wie einem λ-Term ein Typausdruck zuzuweisen ist. Wir wollen diese Vorschriften nun in der Form von syntaktischen Regeln präzisieren, um innerhalb eines formalen Kalküls *beweisen* zu können, daß eine gegebene Typzuweisung korrekt ist. Da wir die einfache Typisierung im Kapitel 3 zu einer Formalisierung beliebiger Programm*eigenschaften* ausbauen werden, ist dieser sehr einfache Kalkül ein wesentlicher Schritt in Richtung auf ein formales System zum automatisierten Schließen über die *Eigenschaften* von Programmen.

Wie in den vorhergehenden Abschnitten werden wir für die einfache Typentheorie einen Sequenzenkalkül aufstellen, dessen Regeln die semantischen Anforderungen wiederspiegeln. Wir müssen hierzu das in Definition 2.2.12 gegebene Grundkonzept geringfügig modifizieren, da nun in den Hypothesen die Deklarationen die zentrale Rolle spielen werden und in der Konklusion eine Typisierung anstelle der Formeln auftreten wird. Die Regeln, welche *hinreichende* Bedingungen an die Typzugehörigkeit eines λ-Terms angeben müssen, ergeben sich ganz natürlich aus Definition 2.4.4.

Ein Term der Form $\lambda x.t$ kann nur zu einem Typ gehören, der die Gestalt $S \to T$ hat. Um nachzuweisen, daß dies tatsächlich der Fall ist, nehmen wir an, daß x vom Typ S ist und zeigen dann, daß $t \in T$ gilt. Hierzu müssen wir die Hypothesenliste um die Deklaration $x{:}S$ erweitern und, falls x schon deklariert war, eine entsprechende Umbenennung vornehmen, was zu folgender Regel führt.

$\Gamma \vdash \lambda x.t \in S \to T$ **by** `???_i`
  $\Gamma,\ x'{:}S \vdash t[x'/x] \in T$

Bei Anwendung einer solchen Regel wandert der Term S ungeprüft in die Hypothesenliste, selbst wenn er kein korrekter Typ-Ausdruck ist. Natürlich könnte man dies vermeiden, indem man zu Beginn eines Beweises überprüft, ob der Ausdruck auf der rechten Seite des $\in$-Symbols überhaupt einen Typ-Ausdruck darstellt. Für die einfache Typentheorie ist ein solcher Test ohne Probleme durchführbar. Die Erweiterungen der Typentheorie, die wir in Kapitel 3 besprechen werden, lassen dies jedoch nicht mehr zu.[72] Deshalb werden wir die Überprüfung der *Wohlgeformtheit* eines Typausdrucks ebenfalls in unseren Kalkül mit aufnehmen. Formal geschieht dies dadurch, daß wir ein neues Symbol $\mathbb{U}$ einführen, welches als *Universum aller Typen* interpretiert wird. Die Wohlgeformtheit eines Typausdrucks S werden wir dann nachweisen, indem wir zeigen, daß S zu diesem Universum gehört – also, daß $S \in \mathbb{U}$ gilt. Entsprechend ergänzen wir die obige Regel um ein weiteres Unterziel: wird $x{:}S$ als Deklaration in die Hypothesenliste aufgenommen, so ist zu zeigen, daß $S \in \mathbb{U}$ gilt.

$\Gamma \vdash \lambda x.t \in S \to T$ **by** `lambda_i`
  $\Gamma,\ x'{:}S \vdash t[x'/x] \in T$
  $\Gamma \vdash S \in \mathbb{U}$

Durch die Hinzunahme des Universums $\mathbb{U}$ entfällt übrigens auch die Notwendigkeit, in formalen Beweisen zwischen Variablensymbolen aus $\mathcal{V}$ und Typsymbolen aus $\mathcal{T}$ zu unterscheiden. Eine Deklaration der Form $x{:}\mathbb{U}$ deklariert x als als *Typsymbol*, während jede andere Deklaration $x{:}Y$ das Symbol x als *Variable* deklariert. Wir können daher ab sofort für alle Variablen beliebige Bezeicher verwenden.

[72] Unter anderem wird die Möglichkeit entstehen, Abstraktionen und Applikationen in Typausdrücken zu verwenden und diese zu reduzieren. Die Frage, ob ein Ausdruck zu einem Typausdruck reduziert werden kann, ist jedoch unentscheidbar.

$\Gamma \vdash \lambda x.t \in S \to T$ **by** `lambda_i`* $\Gamma,\ x'{:}S \vdash t[x'/x] \in T$ $\Gamma \vdash S \in \mathbb{U}$	$\Gamma \vdash f\,t \in T$ **by** `apply_i` S $\Gamma \vdash f \in S \to T$ $\Gamma \vdash t \in S$
$\Gamma \vdash S \to T \in \mathbb{U}$ **by** `function_i` $\Gamma \vdash S \in \mathbb{U}$ $\Gamma \vdash T \in \mathbb{U}$	$\Gamma,\ x{:}T,\ \Delta \vdash x \in T$ **by** `declaration` i

*: *Die Umbenennung $[x'/x]$ erfolgt, wenn x in Γ frei vorkommt.*

Abbildung 2.8: Sequenzenkalkül für die einfache Typentheorie

Die Regel für die Applikation ist sehr einfach anzugeben: um $f\,t \in T$ zu zeigen, müssen wir zeigen, daß das Argument t von f zu einem bestimmten Typ S gehört und daß f eine Funktion von S nach T ist. Da der Typ S nicht automatisch[73] bestimmt werden kann, muß er als Parameter der Regel mit angegeben werden.

$\Gamma \vdash f\,t \in T$ **by** `apply_i` S
  $\Gamma \vdash f \in S \to T$
  $\Gamma \vdash t \in S$

Durch die Aufnahme der Wohlgeformtheit eines Typausdrucks in den Beweis ergibt sich die Notwendigkeit einer weiteren Regel. Um nachzuweisen, das $S \to T$ wohlgeformt ist, müssen wir nur nachweisen, daß dies für S und T gilt. Andere Möglichkeiten, Typausdrücke mit Regeln zu zerlegen, gibt es nicht.

$\Gamma \vdash S \to T \in \mathbb{U}$ **by** `function_i`
  $\Gamma \vdash S \in \mathbb{U}$
  $\Gamma \vdash T \in \mathbb{U}$

Die drei (Einführungs-)Regeln zur Analyse von Ausdrücken werden ergänzt um eine Regel, die besagt, daß man Deklarationen der Form $x{:}T$ zum Nachweis von $x \in T$ heranziehen kann. Diese Regel ist eine leichte Modifikation der Regel `hypothesis` aus Abschnitt 2.2.4. Sie kann sowohl auf Objektvariablen als auch auf Typvariablen angewandt werden.

$\Gamma,\ x{:}T,\ \Delta \vdash x \in T$ **by** `declaration` i

Alle Inferenzregeln der einfachen Typentheorie sind in Abbildung 2.8 zusammengefaßt. Da sich mittlerweile im Bezug auf die konkrete Ausgestaltung von Sequenzen gegenüber der Definition 2.2.12 (Seite 30) eine Reihe von Änderungen ergeben haben, geben wir der Vollständigkeit halber die entsprechend modifizierte Definition für die einfache Typentheorie.

Definition 2.4.6 (Deklaration, Typisierungen und Sequenzen)

1. *Eine <u>Deklaration</u> hat die Gestalt $x{:}X$ (<u>Variablendeklaration</u>) oder $T{:}\mathbb{U}$ (<u>Typdeklaration</u>), wobei x und T beliebige Bezeichner, X ein Ausdruck und $\mathbb{U}$ ein festes Symbol (<u>Universum</u>) ist.*

2. *Eine <u>Typisierung</u> hat die Gestalt $t \in T$ oder $T \in \mathbb{U}$, wobei t und T beliebige Ausdrücke sind.*

3. *Eine <u>Sequenz</u> hat die Gestalt $\Gamma \vdash C$, wobei die <u>Hypothesenliste</u> Γ eine Liste von durch Komma getrennten Deklarationen und die <u>Konklusion</u> C eine Typisierung ist.*

4. *Eine Hypothesenliste ist <u>rein</u>, wenn jeder Bezeichner genau einmal deklariert ist und jeder Bezeichner T, der auf der rechten Seite einer Deklaration erscheint, zuvor durch $T{:}\mathbb{U}$ deklariert wurde.*

5. *Eine <u>Initialsequenz</u> ist eine Sequenz der Gestalt $\Gamma \vdash t \in T$, deren Hypothesenliste rein ist, eine Typdeklaration für alle in T vorkommenden Bezeichner enthält und sonst keine Variablen deklariert.*

[73] Im Falle der einfachen Typentheorie könnte man für S durch den in Abschnitt 2.4.4 angegebenen Typechecking Algorithmus das prinzipielle Typschema bestimmen. Dies läßt sich jedoch nur begrenzt verallgemeinern und würde zudem die schematische Regel in eine Regel verwandeln, die nur im Zusammenhang mit diesem Algorithmus erklärt werden kann.

```
T:𝕌 ⊢ (λf.λx.f x)(λz.z) ∈ T→T
by apply_i  T→T
|\
| T:𝕌 ⊢ λf.λx.f x ∈ (T→T) → (T→T)
| by lambda_i
| |\
| | T:𝕌, f:T→T ⊢ λx.f x ∈ T→T
| | by lambda_i
| | |\
| | | T:𝕌, f:T→T, x:T ⊢ f x ∈ T
| | | by apply_i  T
| | | |\
| | | | T:𝕌, f:T→T, x:T ⊢ f ∈ T→T
| | | | by declaration 2
| | |  \
| | |   T:𝕌, f:T→T, x:T ⊢ x∈T
| | |   by declaration 3
| |  \
| |   T:𝕌, f:T→T ⊢ T∈𝕌
| |   by declaration 1
|  \
|   T:𝕌 ⊢ T→T ∈ 𝕌
|   by function_i
|   |\
|   | T:𝕌 ⊢ T∈𝕌
|   | by declaration 1
|    \
|     T:𝕌 ⊢ T∈𝕌
|     by declaration 1
 \
  T:𝕌 ⊢ λz.z ∈ T→T
  by lambda_i
  |\
  | T:𝕌, z:T ⊢ z ∈ T
  | by declaration 2
   \
    T:𝕌 ⊢ T∈𝕌
    by declaration 1
```

Abbildung 2.9: Sequenzenbeweis für eine Typisierung von $(\lambda\texttt{f}.\lambda\texttt{x}.\texttt{f x})(\lambda\texttt{z}.\texttt{z})$

Der eigentliche Beweisbegriff wird aus Definition 2.2.12 unverändert übernommen. Dementsprechend gilt eine Typisierung $t\in T$ als bewiesen, wenn es einen vollständigen Beweis für die (eindeutig bestimmte) zu $t\in T$ gehörende Initialsequenz gibt. Wir wollen dies an einem einfachen Beispiel illustrieren.

Beispiel 2.4.7

Wir wollen zeigen, daß $(\lambda\texttt{f}.\lambda\texttt{x}.\texttt{f x})(\lambda\texttt{z}.\texttt{z}) \in \texttt{T}\to\texttt{T}$ gilt.

Die äußere Operation des gegebenen Terms ist eine Applikation. Wir müssen daher als erstes die Regel `apply_i` angeben und hierbei einen Typen für das Argument $\lambda\texttt{z}.\texttt{z}$ angeben. Eine solche Angabe erfordert eine Reihe von Vorüberlegungen. Der Typ für $\lambda\texttt{z}.\texttt{z}$ wird gleichzeitig der Typ der Variablen `f` werden und $\lambda\texttt{x}.\texttt{f x}$ wird den Typen $\texttt{T}\to\texttt{T}$ erhalten müssen. Sinnvoll ist es also $\texttt{x}\in\texttt{T}$ und $\texttt{f}\in\texttt{T}\to\texttt{T}$ anzusetzen. Diese Überlegungen führen zur Anwendung der Regel `apply_i` $\texttt{T}\to\texttt{T}$, welche uns folgende Teilziele hinterläßt.

$$\lambda\texttt{f}.\lambda\texttt{x}.\texttt{f x} \in (\texttt{T}\to\texttt{T}) \to (\texttt{T}\to\texttt{T})$$

und

$$\lambda\texttt{z}.\texttt{z} \in \texttt{T}\to\texttt{T}$$

Während das zweite Teilziel nun direkt durch `lambda_i` und die Regel `declaration` bewiesen werden kann, ist für das erste eine Reihe weiterer Schritte erforderlich. Hierbei bestimmt jeweils der äußere Operator (λ-Abstraktion, Applikation oder Funktionenraumbildung) eindeutig die anzuwendende Regel und nur bei der Applikation müssen wieder Parameter angegeben werden. Diese ergeben sich aber aus obigen Vorbetrachtungen und führen zu dem in Abbildung 2.9 angegebenen Beweis.

Man beachte, daß der in Abbildung 2.8 angegebene Kalkül ausschließlich das Schließen über Typzugehörigkeit unterstützt. Die semantische Gleichheit von λ-Termen wird gemäß Theorem 2.4.5 zwar respektiert, aber das Schließen über Werte ist nicht in den Kalkül der Typentheorie integriert. Dies werden wir im Kapitel 3 nachholen, wenn wir Logik, Berechnung und Typdisziplin in einem Kalkül vereinen.

2.4.4 Ein Typechecking Algorithmus

Mit dem im vorhergehenden Abschnitt angegebenen Kalkül können wir eine vorgegebene Typisierung eines λ-Terms *überprüfen*. Wie aber *finden* wir eine solche Typisierung? Ist es möglich, den Typ eines λ-Terms durch eine Analyse der in Definition 2.4.4 angegebenen Vorschriften automatisch zu bestimmen?

Für die einfache Typentheorie und geringfügige Erweiterungen davon kann diese Frage positiv beantwortet werden. Es ist in der Tat möglich, einen (immer terminierenden) Algorithmus anzugeben, welcher zu jedem typisierbaren Term das prinzipielle Typschema angibt. Dieser Algorithmus basiert auf einem Typechecking Algorithmus, der von Hindley und Milner[74] entwickelt wurde und einen wesentlichen Bestandteil des reichhaltigen Typkonzepts der funktionalen Programmiersprache ML[75] bildet.

Die Grundidee dieses Verfahrens ist simpel. Um einem Term t einen Typ T zuzuweisen, ordnen wir t einfach einen Typen zu und beginnen mit der Typprüfung. Wann immer der Beweis dies nötig macht, werden wir T weiter verfeinern, bis die Prüfung erfolgreich beendet wurde oder offensichtlich ist, daß kein Typ zugeordnet werden kann. Bevor wir den Algorithmus im Detail vorstellen wollen wir ihn an einem Beispiel erläutern.

Beispiel 2.4.8

Um dem Term $\lambda\texttt{f}.\lambda\texttt{x}.\texttt{f}\,\texttt{x}$ einen Typ zuzuordnen (vgl. Beispiel 2.4.2), beginnen wir mit der Behauptung

$$\lambda\texttt{f}.\lambda\texttt{x}.\ \texttt{f x} \in \texttt{X}_0$$

Aufgrund der λ-Abstraktion muß $\texttt{X}_0$ die Gestalt $\texttt{X}_1\to\texttt{X}_2$ haben und

$$\lambda\texttt{x}.\ \texttt{f x} \in \texttt{X}_2$$

gelten, wobei `f` vom Typ $\texttt{X}_1$ ist. Genauso folgern wir, daß $\texttt{X}_2$ die Gestalt $\texttt{X}_3\to\texttt{X}_4$ haben muß und

$$\texttt{f x} \in \texttt{X}_4$$

gilt, wobei `x` vom Typ $\texttt{X}_3$ ist. Da der Typ des Argumentes von `f` feststeht, muß $\texttt{X}_1$ – der Typ von `f` – identisch mit $\texttt{X}_3\to\texttt{X}_4$ sein. Damit sind alle Bestandteile des Terms typisiert und es ist

$$\lambda\texttt{f}.\lambda\texttt{x}.\ \texttt{f x} \in (\texttt{X}_3\to\texttt{X}_4) \to (\texttt{X}_3\to\texttt{X}_4)$$

Da die Namen der Typvariablen $\texttt{X}_3$ und $\texttt{X}_4$ nur Platzhalter im prinzipiellen Typschema sind, haben wir nun den Typ von $\lambda\texttt{f}.\lambda\texttt{x}.\ \texttt{f x}$ bestimmt.

Das folgende Beispiel zeigt, wie das angedeutete Verfahren feststellt, daß eine Typisierung unmöglich ist.

Beispiel 2.4.9

Um dem Term $\lambda\texttt{x}.\texttt{x}\,\texttt{x}$ einen Typ zuzuordnen (vergleiche Beispiel 2.4.3), beginnen wir mit $\lambda\texttt{x}.\texttt{x}\,\texttt{x} \in \texttt{X}_0$. Aufgrund der λ-Abstraktion muß $\texttt{X}_0$ die Gestalt $\texttt{X}_1\to\texttt{X}_2$ haben und $\texttt{x x} \in \texttt{X}_2$ gelten, wobei `x` vom Typ $\texttt{X}_1$ ist. Da der Typ des Argumentes von `x` feststeht, muß $\texttt{X}_1$ – der Typ von `x` – identisch mit $\texttt{X}_1\to\texttt{X}_2$ sein. Der Versuch, $\texttt{X}_1$ und $\texttt{X}_1\to\texttt{X}_2$ durch *Unifikation*[76] gleichzumachen, wird jedoch fehlschlagen. Daher ist der Term $\lambda\texttt{x}.\texttt{x}\,\texttt{x}$ nicht typisierbar.

[74] Die entsprechenden Originalarbeiten sind in [Hindley, 1969, Milner, 1978, Damas & Milner, 1982] dokumentiert. Weitere Darstellungen dieses Verfahrens im Kontext verschiedener Formalismen findet man in [Hindley, 1983, Hindley & Seldin, 1986, Cardone & Coppo, 1990].

[75] Die Metasprache des NuPRL Systems – eine der ersten Versionen der Programmiersprache ML – enthält diesen Algorithmus. Es ist daher relativ einfach, das prinzipielle Typschema eines λ-Terms mit dem System zu finden. Um zum Beispiel den Term $\lambda\texttt{f}.\lambda\texttt{x}.\texttt{f}\,\texttt{x}$ zu typisieren, muß man diesen nur im ML top-loop eingeben. Man beachte hierbei jedoch, daß es sich um einen ML-Ausdruck und nicht etwa einen objektsprachlichen Ausdruck der Typentheorie von NuPRL handelt. Man gebe also ein:
`\f.\x. f x` (ohne 'Term-quotes')
Im Emacs Fenster wird dann als Antwort die Typisierung `\f.\x. f x : (*->**) -> (*->**)` erscheinen, wobei `*` und `**` Platzhalter für Typvariablen sind.

Algorithmus $\boxed{\texttt{TYPE-SCHEME-OF}(t)}$: *(t: (geschlossener) λ-Term)*

Initialisiere die Substituition σ (eine globale Variable) als identische Abbildung und rufe `TYPE-OF([],` t`)` auf.

Falls der Algorithmus fehlschlägt ist t nicht typisierbar.
Andernfalls ist das Resultat T das prinzipielle Typschema von t.

Hilfsalgorithmus $\underline{\texttt{TYPE-OF}(Env,t)}$: *(Env: Liste der aktuellen Annahmen, t: aktueller Term)*

- Falls t die Gestalt $\underline{x}$ hat, wobei $x \in \mathcal{V}$ eine Variable ist:
  Suche in Env die (eindeutige) Deklaration der Gestalt $x{:}T$ und gebe T aus.
- Falls t die Gestalt $\underline{f\ u}$ hat:
  Bestimme S_1:=`TYPE-OF(`Env`,`f`)` und dann S_2:=`TYPE-OF(`Env`,`u`)`*. Wähle eine neue Typvariable $\mathtt{X}_{i+1}$ und versuche, $\sigma(S_1)$ mit $S_2 \to \mathtt{X}_{i+1}$ zu unifizieren. Falls die Unifikation fehlschlägt, breche mit einer Fehlermeldung ab. Andernfalls ergänze σ um die bei der Unifikation erzeugte Substitution σ' ($\sigma := \sigma' \circ \sigma$). Ausgabe ist $\sigma(\mathrm{X}_{i+1})$
- Falls t die Gestalt $\underline{\lambda x.u}$ hat: Wähle eine neue Typvariable $\mathtt{X}_{i+1}$
  $$\text{Bestimme } S_1 := \begin{cases} \texttt{TYPE-OF}(Env\cdot[x{:}\mathtt{X}_{i+1}],u) & \text{falls } x \text{ nicht in } Env \text{ vorkommt}^* \\ \texttt{TYPE-OF}(Env\cdot[x'{:}\mathtt{X}_{i+1}],u[x'/x]) & \text{sonst } (x' \text{ neue Objektvariable})^* \end{cases}$$
  Ausgabe ist $\sigma(\mathrm{X}_{i+1}) \to S_1$

*: Beachte, daß σ beim Aufruf von `TYPE-OF` ergänzt worden sein kann.

Abbildung 2.10: Typechecking Algorithmus für die einfache Typentheorie

Der in Abbildung 2.10 vorgestellte Algorithmus ist beschrieben für geschlossene λ-Terme, also λ-Terme ohne freie Variablen. Die Typen eventuell vorkommender freier Variablen können dadurch bestimmt werden, daß man den Term durch entsprechende λ-Abstraktionen abschließt.

Satz 2.4.10 (Hindley / Milner)

Es ist effektiv entscheidbar, ob ein λ-Term typisierbar ist oder nicht.

Beweis: Der in Abbildung 2.10 Algorithmus bestimmt das prinzipielle Typschema eines gegebenen λ-Terms t oder endet mit einer Fehlermeldung, falls keine Typisierung möglich ist. Einen Beweis für seine Korrektheit, der sich eng an den Kalkül aus Abschnitt 2.4.3 anlehnt findet man in [Milner, 1978]. □

Da der Hindley-Milner Algorithmus für jeden typisierbaren λ-Term das prinzipielle Typschema bestimmen kann, könnte man innerhalb der einfachen Typentheorie eigentlich auf den Typüberprüfungskalkül verzichten. Anstatt $t \in T$ zu beweisen, könnte man genauso gut das Typschema von t zu bestimmen und zu vergleichen, ob T diesem Schema entspricht. Da dieses Verfahren jedoch nicht bei allen notwendigen Erweiterungen der Typentheorie funktioniert, muß man sich darauf beschränken, den Hindley-Milner Algorithmus als unterstützende Strategie einzusetzen, die in vielen – aber eben nicht in allen – Fällen zum Ziele führt.

Wir wollen den (zeitlichen) Ablauf des Algorithmus an zwei Beispielen erläutern. Die Beispiele zeigen, wie ein Term durch den Algorithmus zunächst zerlegt wird, durch einen rekursiven Aufruf die Typen der Teilterme ermittelt werden und dann der Typ des zusammengesetzten Terms gebildet wird. Da innerhalb eines rekursiven

[76] *Unifikation* ist ein Verfahren zur Bestimmung einer Substitution σ, welche freie Variablen in zwei vorgegebenen Ausdrücken so ersetzt, daß die beiden Ausdrücke gleich werden. Das allgemeine Verfahren wurde in [Robinson, 1965] wurde ursprünglich für prädikatenlogische Terme entwickelt, ist aber genauso auf Typausdrücke anwendbar, solange Reduktion keine Rolle spielt. Es zerlegt simultan die syntaktische Struktur der beiden Terme solange, bis ein äußerer Unterschied auftritt. An dieser Stelle muß nun die Variable des einen Termes durch den entsprechenden Teilterm des anderen ersetzt werden, wobei ein *Occur*-check sicherstellt, daß die ursprünglichen Variablen nicht in dem eingesetzten Teilterm enthalten sind. Das Verfahren schlägt fehl, wenn eine solche Ersetzung nicht möglich ist.

Aufrufes weitere Zerlegungen stattfinden können, werden im Laufe der Zeit verschiedene Teilterme ab- und wieder aufgebaut. Zur Beschreibung der einzelnen Schritte geben wir in einer Tabelle jeweils die Werte der Variablen *Env*, t und σ beim Aufruf von `TYPE-OF` an, sowie die durchgeführte Unifikation, die im folgenden Schritt zu einer veränderten Substitution σ führt. Die letzte Spalte benennt den berechneten Typ.

Beispiel 2.4.11

1. Typisierung des Terms `λf.λx.f(f(f x))` mit dem Hindley-Milner Algorithmus:

Env	*Aktueller Term t*	σ	`UNIFY`	*Typ*
	`λf.λx.f(f(f x))`			
$f{:}X_0$	`λx.f(f(f x))`			
$f{:}X_0,\ x{:}X_1$	`f(f(f x))`			
$f{:}X_0,\ x{:}X_1$	`f`			X_0
$f{:}X_0,\ x{:}X_1$	`f(f x)`			
$f{:}X_0,\ x{:}X_1$	`f`			X_0
$f{:}X_0,\ x{:}X_1$	`f x`			
$f{:}X_0,\ x{:}X_1$	`f`			X_0
$f{:}X_0,\ x{:}X_1$	`x`			X_1
$f{:}X_0,\ x{:}X_1$	`f x`		$X_0 = X_1 \to X_2$	X_2
$f{:}X_0,\ x{:}X_1$	`f(f x)`	$[X_1 \to X_2 / X_0]$	$X_1 \to X_2 = X_2 \to X_3$	X_3
$f{:}X_0,\ x{:}X_1$	`f(f(f x))`	$[X_3, X_3, X_3 \to X_3 \,/\, X_2, X_1, X_0]$	$X_3 \to X_3 = X_3 \to X_4$	X_4
$f{:}X_0$	`λx.f(f(f x))`	$[X_4, X_4, X_4, X_4 \to X_4 \,/\, X_3, X_2, X_1, X_0]$		$X_4 \to X_4$
	`λf.λx.f(f(f x))`	$[X_4, X_4, X_4, X_4 \to X_4 \,/\, X_3, X_2, X_1, X_0]$		$(X_4 \to X_4) \to X_4 \to X_4$

2. Typisierung des Terms `(λf.λx.f x) (λx.x)` mit dem Hindley-Milner Algorithmus:

Env	*Aktueller Term t*	σ	`UNIFY`	*Typ*
	`(λf.λx.f x)(λx.x)`			
	`λf.λx.f x`			
$f{:}X_0$	`λx.f x`			
$f{:}X_0,\ x{:}X_1$	`f x`			
$f{:}X_0,\ x{:}X_1$	`f`			X_0
$f{:}X_0,\ x{:}X_1$	`x`			X_1
$f{:}X_0,\ x{:}X_1$	`f x`		$X_0 = X_1 \to X_2$	X_2
$f{:}X_0$	`λx.f x`	$[X_1 \to X_2 / X_0]$		$X_1 \to X_2$
	`λf.λx.f x`	$[X_1 \to X_2 / X_0]$		$(X_1 \to X_2) \to X_1 \to X_2$
	`λx.x`			
$x{:}X_3$	`x`			X_3
	`λx.x`			$X_3 \to X_3$
	`(λf.λx.f x)(λx.x)`	$[X_1 \to X_2 / X_0]$	$(X_1 \to X_2) \to X_1 \to X_2 = (X_3 \to X_3) \to X_4$	$X_3 \to X_3$

Der Term des ersten Beispiels kann im Prinzip wie wie Funktion `twice` aus Beispiel 2.3.29 (Seite 62) auf sich selbst angewandt werden. Ein Ausdruck wie

`(λf.λx.f(f(f x))) (λf.λx.f(f(f x)))`

ist legal und kann auch typisiert werden. Dennoch wird durch die Typisierung die Möglichkeit der Selbstanwendung stark eingeschränkt. Es zeigt sich nämlich, daß jedes der beiden Vorkommen von `λf.λx.f(f(f x))` anders typisiert wird. Das zweite erhält den Typ $(X \to X) \to X \to X$, während dem ersten Vorkommen der Typ das $((X \to X) \to X \to X) \to (X \to X) \to X \to X$, also ein Typ einer wesentlich höheren Ordnung zugewiesen wird. Diese Konsequenz einer Typisierung wird sich besonders stark bei der Rekursion auswirken.

2.4.5 Eigenschaften typisierbarer λ-Terme

Wir wollen nun nachweisen, daß sich durch die Hinzunahme der Typdisziplin zum λ-Kalkül eine Reihe günstiger Eigenschaften ergeben, welche die Auswertung von λ-Termen und das automatische Schließen über ihre Eigenschaften erheblich vereinfachen. Im Gegensatz zum ungetypten λ-Kalkül besitzt jeder typisierbare λ-Term eine Normalform (siehe Abschnitt 2.4.5.1). Dabei ist es unerheblich, welche Reduktionsstrategie man verwendet, denn jede Reduktionsfolge wird terminieren (siehe Abschnitt 2.4.5.2). Das hat zur Folge, daß man in der Typentheorie erheblich effizientere Reduktionsstrategien einsetzen kann. Außerdem ermöglicht diese Eigenschaft auch einen wesentlich einfacheren Beweis für das Church-Rosser Theorem, welches die Eindeutigkeit der Normalform eines λ-Terms zum Inhalt hat (siehe Abschnitt 2.4.5.3). Eine Konsequenz hiervon ist, daß semantische Gleichheit von λ-Termen entscheidbar wird (siehe Abschnitt 2.4.5.4). Insgesamt liefert uns also die Typsdisziplin eine *entscheidbare Theorie totaler Funktionen.*

2.4.5.1 Schwache Normalisierbarkeit

Wir wollen als erstes zeigen, daß jeder typisierbare λ-Term normalisierbar ist – daß es also mindestens eine Reduktionsstrategie gibt, die in jedem Fall zu einer Normalform führt. Aufgrund von Lemma 2.3.33 (Seite 64) wissen wir bereits, daß die Reduktion des jeweils äußersten Redex (call-by-name) immer zum Ziel führt, wenn es überhaupt eine Normalform gibt. Diese Strategie ist allerdings relativ ineffizient. Eine wesentlich bessere Strategie wird uns der Beweis der sogenannten *schwachen Normalisierbarkeit* typisierbarer λ-Terme liefern, da er sich auf kombinatorische Argumente und die Komplexität der Typstruktur eines λ-Terms stützen kann.

Die Grundidee ist verhältnismäßig einfach. Ein Funktionenraum der Gestalt $(S_1 \to S_2) \to (T_1 \to T_2)$ beschreibt Funktionen, welche wiederum einfache Funktionen als Argumente und Ergebnis verwenden. Die Funktionenraumstruktur hat also die *Tiefe* 2, da der Raum in zwei Ebenenen aufgebaut wurde. Wenn wir es nun erreichen würden, durch eine Reduktionsstrategie die Tiefe des Typs eines gegebenen λ-Terms schrittweise zu reduzieren, dann muß diese Reduktionskette irgendwann einmal terminieren.[77] Wir geben hierzu eine Definition der Tiefe von Typausdrücken und typisierbaren λ-Termen.

Definition 2.4.12

Die Tiefe $d(T)$ eines Typausdrucks T ist induktiv wie folgt definiert.

- $d(T) = 0$, *falls T ein atomarer Typ ist* $(T \in \mathcal{T})$
- $d(S \to T) = 1 + max(d(S), d(T))$.

Die Tiefe eines Redex $(\lambda x.t)\, u$ ist die Tiefe des Typs von $\lambda x.t$.

Die Tiefe eines typisierbaren λ-Terms t ist die maximale Tiefe der in t enthaltenen Redizes (und 0, wenn t keine Redizes enthält).

Wie können wir nun eine Strategie entwickeln, die mit Sicherheit die Tiefe eines λ-Terms verringern wird? Die naheliegenste Idee ist natürlich, ein Redex maximaler Tiefe zu reduzieren, da hierdurch ja ein solches Redex aus dem Term entfernt wird. Diese Überlegung alleine reicht jedoch nicht aus, wie das folgende Beispiel zeigt.

Beispiel 2.4.13

Betrachte den Term `trice(trice(λx.x))`, wobei `trice` $\equiv$ `λf.λx.f(f(f x))` (vgl. Beispiel 2.4.11.1).

Da $(X \to X) \to (X \to X)$ der prinzipielle Typ von `trice` ist, folgt gemäß Definition 2.4.12, daß sowohl `trice(λx.x)` als auch `trice(trice(λx.x))` Redizes der Tiefe 2 sind. Würden wir nun (entsprechend der Leftmost-Reduktionsstrategie) das äußerste Redex maximaler Tiefe, also `trice(trice(λx.x))`, zuerst reduzieren, so erhielten wir

`λx. (trice(λx.x))((trice(λx.x))((trice(λx.x))x))`

also einen Term, welcher nunmehr 3 Redizes der Tiefe 2 enthält.

[77] Bei ungetypten λ-Termen ist dieses Argument nicht anwendbar, da die Tiefe des zugehörigen Funktionenraumes nicht festgestellt werden kann.

Die Reduktion eines Redex maximaler Tiefe kann also dazu führen, daß mehr Redizes maximaler Tiefe entstehen als zuvor vorhanden waren. Mit kombinatorischen Argumenten könnte die Terminierung der eben angedeuteten Vorgehensweise daher nicht bewiesen werden. Dieser unerwünschte Effekt kann jedoch vermieden werden, wenn wir das am weitesten rechts stehende Redex maximaler Tiefe verringern, da dieses keine Teilterme maximaler Tiefe mehr enthält.

Lemma 2.4.14 (Rightmost-Maxdepth Strategie)

Reduziert man in einem typisierbaren λ-Term t das am weitesten rechts stehende Redex maximaler Tiefe, so verringert sich die Anzahl der Redizes maximaler Tiefe.

Beweis: Es sei t ein typisierbarer λ-Term und $r{=}(\lambda x.u)\, v$ das am weitesten rechts stehende Redex maximaler Tiefe. Weder u noch v enthalten ein Redex maximaler Tiefe und die Reduktion von r hat die folgenden Effekte.

- Alle Redizes von t außerhalb von r bleiben unverändert.
- Ein Redex maximaler Tiefe, nämlich r, wird entfernt.
- Der entstehende Term $u[v/x]$ enthält keine Redizes maximaler Tiefe (auch wenn es durchaus möglich ist, daß andere Redizes vervielfältigt werden).

Damit wird durch die Reduktion von r die Anzahl der Redizes maximaler Tiefe verringert, während die Anzahl der Redizes geringerer Tiefe wachsen kann. □

Wir wollen den Effekt der Rightmost-Maxdepth Strategie an dem obigen Beispiel illustrieren.

Beispiel 2.4.15

Wenn wir in `trice(trice(λx.x))` das am weitesten rechts stehende Redex maximaler Tiefe, also `trice(λx.x)`, reduzieren, so entsteht der Term

`trice(λx.(λx.x)((λx.x)((λx.x)x)))`

Insgesamt ist hierbei die Anzahl der Redizes gewachsen, aber die Anzahl der Redizes maximaler Tiefe ist gesunken. Reduzieren wir nun wiederum das am weitesten rechts stehende Redex maximaler Tiefe, so wird der Term in seiner Größe zwar gewaltig anwachsen, aber es gibt nur noch Redizes der Tiefe 1.

```
λx. (λx.(λx.x)((λx.x)((λx.x)x)))
      ( (λx.(λx.x)((λx.x)((λx.x)x)))
            ( (λx.(λx.x)((λx.x)((λx.x)x))) x) )
```

Ab nun werden schrittweise alle Redizes der Tiefe 1 abgebaut, wobei keine neuen Redizes mehr entstehen können, da es Redizes der Tiefe 0 nicht geben kann.

Das obige Beispiel deutet bereits an, warum die Rightmost-Maxdepth Strategie terminieren muß. In jedem Schritt verringert sich die Anzahl der Redizes maximaler Tiefe, bis diese auf Null gesunken ist. Anschließend werden die Redizes der nächstgeringeren Teife abgebaut und das Ganze so lange fortgeführt bis kein Redex der mehr übrig ist und der Term in Normalform ist.

Satz 2.4.16 (Terminierung der Rightmost-Maxdepth Strategie)

Zu jedem typisierbaren λ-Term t kann man eine Anzahl n von Schritten bestimmen, innerhalb derer die Rightmost-Maxdepth-Strategie *bei Anwendung aus t terminiert.*

Beweis: Durch eine Doppelinduktion über d und m zeige man unter Verwendung von Lemma 2.4.14:

Für alle $d \in \mathbb{N}$ und alle $m \in \mathbb{N}$ gilt: wenn t ein λ-Term der Tiefe d ist und maximal m Redizes der Tiefe d enthält, dann gibt es eine Normalform t' von t und es gilt $t \xrightarrow{n} t'$ für ein $n \in \mathbb{N}$.

Die Ausführung des Induktionsbeweises sei dem Leser als Übung überlassen. □

Da wir nun eine Reduktionsstrategie kennen, die auf allen typsisierbaren λ-Termen terminiert, wissen wir auch, daß alle typisierbaren λ-Terme eine Normalform besizten müssen.

Korollar 2.4.17 (Schwache Normalisierbarkeit typisierbarer λ-Terme)

Jeder typisierbare λ-Term ist normalisierbar.

2.4.5.2 Starke Normalisierbarkeit

Wir haben soeben gezeigt, daß jeder typisierbare λ-Term eine terminierende Reduktionsfolge besitzt und daß es eine Strategie gibt, diese zu finden. Für einfach getypte λ-Terme kann man jedoch noch erheblich mehr zeigen, nämlich daß *jede* Reduktionsfolge terminieren muß. Diese Eigenschaft nennt man in der Sprache der Termersetzung *starke Normalisierbarkeit*.

Definition 2.4.18 (Starke Normalisierbarkeit)

Es sei $\xrightarrow{r}$ eine Reduktionsrelation und t ein Term.

1. *t heißt <u>stark normalisierbar</u> (SN), wenn jede mit t beginnende Reduktionsfolge endlich ist.*
2. *$\xrightarrow{r}$ ist stark normalisierbar, wenn jeder Term t unter $\xrightarrow{r}$ stark normalisierbar ist.*

Da jeder Term t nur an endlich vielen Stellen reduziert werden kann, bedeutet starke Normalisierbarkeit insbesondere, daß es für diesen Term eine Anzahl von Schritten gibt, innerhalb derer *alle* in t beginnenden Reduktionsfolgen terminieren. Es ist daher legitim, bei stark normalisierbaren Termen von der (maximalen) Anzahl der Reduktionsschritte zu sprechen. Eine Reduktionsrelation $\xrightarrow{r}$ ist stark normalisierbar, wenn jeder beliebige Term – unabhängig von der Reduktionsstrategie – nach endlich vielen Schritten in einer nicht weiter reduzierbaren Form ist.

Wir wollen nun zeigen, daß die β-Reduktion $\xrightarrow{\beta}$ auf typisierbaren λ-Termen genau diese Eigenschaft besitzt. Prinzipiell wäre es möglich, für die Theorie der einfachen Typen wie bei der schwachen Normalisierbarkeit ein kombinatorisches Argument zu konstruieren, also zu zeigen, daß irgendeine Größe durch *jede* Reduktion eines typisierbaren λ-Terms verringert wird. Wir werden im folgenden jedoch eine Beweismethode vorstellen, die sich auf aufwendigere Typsysteme, Reduktion höherer Ordnung und Polymorphie verallgemeinern läßt. Im Beweis werden wir daher Situationen betrachten, die im gegenwärtigen Kontext trivial erscheinen mögen, aber in allgemeineren Situationen von Bedeutung sein werden.

Die nach ihrem Erfinder benannte *TAIT computability method* [Tait, 1967] (mit technischen Verbesserungen von Girard [Girard, 1972]) ist im wesentlichen ein Induktionsbeweis, der in zwei Phasen vorgeht. Im ersten Schritt wird nachgewiesen, daß typisierbare λ-Terme eine wesentlich stärkere Eigenschaft besitzen als starke Normalisierbarkeit. Diese Eigenschaft, die als *Berechenbarkeit* bezeichnet wird, liefert erheblich stärkere Annahmen beim Führen des Induktionsschrittes und ist deshalb besser für eine Beweisführung geeignet. Im zweiten Schritt wird dann gezeigt, daß Berechenbarkeit tatsächlich stärker ist als starke Normalisierbarkeit.

Definition 2.4.19

<u>Berechenbare</u> λ-Terme sind induktiv wie folgt definiert.

– *Wenn $t \in T$ für einen atomaren Typ T gilt und t stark normalisierbar ist, dann ist t berechenbar.*
– *Gilt $t \in S \rightarrow T$ und ist $t\,s$ berechenbar für jeden berechenbaren Term $s \in S$, dann ist t berechenbar.*

Ein λ-Term ist <u>neutral</u>, wenn er nicht die Gestalt $\lambda x.t$ hat.

Der Unterschied zwischen Berechenbarkeit und starker Normalisierbarkeit liegt vor allem bei der Betrachtung von durch λ-Abstraktion definierten Funktionen, also den *kanonischen* Termen des λ-Kalküls. Während es bei starker Normalisierbarkeit ausreicht, daß alle Teilterme dieses Terms stark normalisierbar sind, verlangt die Berechenbarkeit, daß dies (d.h. Berechenbarkeit) auch dann gilt, wenn man für die gebundene Variable einen beliebigen berechenbaren Term einsetzt. Technisch ausgedrückt heißt dies, daß die Bildung eines Terms mit dem zugehörigen nichtkanonischen Operator (Applikation) zu einem berechenbaren Term führen muß. Diese Sprechweise deutet auch an, auf welche Art die obigen Begriffe auf andere Typkonstrukte verallgemeinert werden können.

Lemma 2.4.20

Es sei T ein beliebiger Typausdruck. t und t' seien beliebige λ-Terme vom Typ T. Dann gilt

1. *Wenn t berechenbar ist, dann ist t stark normalisierbar.*
2. *Wenn t berechenbar ist und $t \xrightarrow{\beta} t'$ gilt, dann ist t' berechenbar.*
3. *Wenn t neutral ist und jede β-Reduktion eines Redex in t zu einem berechenbaren Term t' führt, dann ist t berechenbar.*
4. *Wenn t neutral und in Normalform ist, dann ist t berechenbar.*

Beweis: Wir beweisen die Behauptungen durch eine simultane Induktion über die Struktur des Typs T.

Atomare Typen: Falls T atomar ist, so ist gemäß Definition 2.4.19 ein Term vom Typ T genau dann berechenbar, wenn er stark normalisierbar ist.
Es seien t und t' beliebige λ-Terme vom Typ T.

1. Wenn t berechenbar ist, dann ist t gemäß Definition 2.4.19 stark normalisierbar.
2. Wenn t berechenbar ist und $t \xrightarrow{\beta} t'$ gilt, dann sind alle Reduktionsfolgen von t' auch Teil einer in t beginnenden Reduktionsfolge. Da t auch stark normalisierbar ist, müssen all diese Reduktionsfolgen terminieren. Dies bedeutet, daß t' stark normalisierbar und damit auch berechenbar ist.
3. Wenn t neutral ist und jede β-Reduktion eines Redex in t zu einem berechenbaren Term t' führt, dann passiert jede in t beginnende Reduktionsfolge im ersten Schritt einen stark normalisierbaren Term t' vom Typ T. Dies bedeutet, daß diese Reduktionsfolgen terminieren muß, woraus die starke Normalisierbarkeit von t, also auch die Berechenbarkeit folgt.
4. Die Behauptung folgt unmittelbar aus der dritten, da auf einen Term in Normalform keine β-Reduktion mehr angewandt werden kann.

Funktionenräume: Falls T die Gestalt $T_1 \rightarrow T_2$ hat, dann ist gemäß Definition 2.4.19 ein Term t vom Typ T genau dann berechenbar, wenn jede Applikation auf berechenbare Terme wieder einen berechenbaren Term liefert. Als Induktionsannahme setzen wir voraus, daß die Behauptungen 1. –4. für alle Terme des Typs T_1 bzw. T_2 bereits bewiesen sind.
Es seien t und t' beliebige λ-Terme vom Typ $T = T_1 \rightarrow T_2$.

1. Es sei t berechenbar und x eine Variable vom Typ T_1. Dann ist x neutral und in Normalform und somit gemäß Induktionsannahme 4. auch berechenbar. Damit ist nach Definition 2.4.19 auch $t\,x$ ein berechenbarer Term vom Typ T_2. Gemäß Induktionsannahme 1. ist $t\,x$ auch stark normalisierbar.
Es sei nun $t \xrightarrow{\beta} t_1 \xrightarrow{\beta} t_2 \xrightarrow{\beta} \ldots$ eine beliebige in t beginnende Reduktionsfolge. Betrachten wir die hieraus entstehende Reduktionsfolge $t\,x \xrightarrow{\beta} t_1\,x \xrightarrow{\beta} t_2\,x \xrightarrow{\beta} \ldots$ für $t\,x$, so wissen wir, daß diese wegen der starken Normalisierbarkeit terminieren muß. Da diese Folge sich direkt durch eine zusätzliche Applikation aus der in t beginnenden Originalfolge ergibt, muß die Originalfolge ebenfalls terminieren. Damit terminiert jede in t beginnende Reduktionsfolge, d.h. t ist stark normalisierbar.
2. Es sei t berechenbar und es gelte $t \xrightarrow{\beta} t'$. Sei s ein beliebiger berechenbarer Term vom Typ T_1. Dann ist nach Definition der Berechenbarkeit $t\,s$ ein berechenbarer Term vom Typ T_2 und es gilt $t\,s \xrightarrow{\beta} t'\,s$. Nach Induktionsannahme 2. ist dann $t'\,s$ berechenbar. Nach Definition 2.4.19 ist damit t' berechenbar.
3. Es sei t neutral und jede β-Reduktion eines Redex in t führe zu einem berechenbaren Term t'. Sei s ein beliebiger berechenbarer Term vom Typ T_1. Dann ist s nach Induktionsannahme 1. auch stark normalisierbar. Um Induktionsannahme 3. verwenden zu können, zeigen wir nun durch Induktion über die Anzahl der Reduktionsschritte von s, daß jede β-Reduktion eines Redex in $t\,s$ zu einem berechenbaren Term t^* führt.
   - Falls s bereits in Normalform ist, dann hat t^* die Gestalt $t'\,s$ und es gilt $t \xrightarrow{\beta} t'$. Nach Voraussetzung ist t' berechenbar und damit auch $t'\,s \equiv t^*$.
   - Wir nehmen an, daß für alle in n Schritten normalisierenden berechenbaren Terme s' vom Typ T_1 gezeigt ist, daß jede β-Reduktion eines Redex in $t\,s'$ zu einem berechenbaren Term t^+ führt.
   - s reduziere in n+1 Schritten zur Normalform und es gelte $t\,s \xrightarrow{\beta} t^*$. Da t neutral ist, kann $t\,s$ selbst kein Redex sein. Es bleiben zwei Möglichkeiten:

(a) t^* hat die die Gestalt $t'\,s$ und es gilt $t \xrightarrow{\beta} t'$. Nach Voraussetzung ist t' berechenbar und damit auch $t'\,s \equiv t^*$.

(b) t^* hat die die Gestalt $t\,s'$ und es gilt $s \xrightarrow{\beta} s'$. Dann normalisiert s' in n Schritten und gemäß Induktionsannahme 2. ist s' berechenbar. Damit ist die innere Induktionsannahme anwendbar: jede β-Reduktion eines Redex in $t\,s'$ führt zu einem berechenbaren Term t^+.
Nun können wir die äußere Induktionsannahme 3. auf $t\,s'$, einen neutralen Term vom Typ T_2 anwenden und es folgt, daß $t\,s' \equiv t^*$ berechenbar ist.

In beiden Fällen ist also t^* berechenbar.

Damit ist gezeigt, daß jede β-Reduktion eines Redex in $t\,s$ zu einem berechenbaren Term t^* führt. Da $t\,s$ einen neutraler Term vom Typ T_2 ist, können wir Induktionsannahme 3. auf $t\,s$ anwenden und folgern, daß $t\,s$ berechenbar ist.
Da s beliebig gewählt war, sind die Voraussetzungen der Definition 2.4.19 erfüllt und t ist berechenbar.

4. Wie zuvor folgt die Behauptung unmittelbar aus der dritten, da auf einen Term in Normalform keine β-Reduktion mehr angewandt werden kann. □

In Definition 2.4.19 wurde die Berechenbarkeit von Funktionen über ihr Verhalten auf *berechenbaren* Argumenten definiert. Mit dem folgenden Lemma zeigen wir, daß wir von diesem Spezialfall abstrahieren können: anstelle von berechenbaren Argumenten dürfen auch beliebige Variablen des gleichen Typs verwendet werden. Der Vorteil hiervon ist, daß eine Variable *alle* Terme eines Typs darstellt und auch den Sonderfall handhaben kann, daß ein Typ überhaupt keine Elemente enthält. Die Verwendung von Variablen erlaubt also eine einheitliche Behandlung aller Typen.[78]

Lemma 2.4.21

Es sei t ein beliebiger typisierbarer λ-Term.
Wenn $t[s/x]$ berechenbar ist für alle berechenbaren λ-Terme s, dann ist $\lambda x\,.\,t$ berechenbar.

Beweis: Es sei t ein beliebiger λ-Term und $t[s/x]$ berechenbar für alle berechenbaren λ-Terme s. Um zu beweisen, daß $\lambda x\,.\,t$ berechenbar ist, müssen wir gemäß Definition 2.4.19 zeigen, daß $(\lambda x\,.\,t)\;s$ berechenbar ist für alle berechenbaren λ-Terme s.

Die Variable x ist neutral und in Normalform, also berechenbar nach Lemma 2.4.20.4. Damit muß $t{\equiv}t[x/x]$ nach Voraussetzung ebenfalls berechenbar sein. Es sei nun s ein berechenbarer Term vom Typ T_1. Dann sind nach Lemma 2.4.20.1. s und t stark normalisierbar.

Wir zeigen nun durch eine Doppelinduktion über die Anzahl der Reduktionssschritte von t und s, daß $(\lambda x\,.\,t)\;s$ berechenbar ist. Hierbei werden wir Gebrauch machen von Lemma 2.4.20.3. Da $(\lambda x\,.\,t)\;s$ neutral ist, reicht es zu zeigen, daß jeder Term t^* berechenbar ist, für den gilt $(\lambda x\,.\,t)\;s \xrightarrow{\beta} t^*$.

Es gelte also $(\lambda x\,.\,t)\;s \xrightarrow{\beta} t^*$. Dann gibt es insgesamt 3 Möglichkeiten für t^*:

1. t^* hat die Gestalt $t[s/x]$. Dann ist t^* nach Voraussetzung berechenbar.
(*Dies deckt auch den Basisfall der Induktion ab*)

2. t^* hat die Gestalt $(\lambda x.t')\,s$, wobei $t \xrightarrow{\beta} t'$ gilt. Da t berechenbar ist, gilt nach Lemma 2.4.20.2., daß auch t' berechenbar ist und in weniger Schritten normalisiert als t. Wir verwenden nun die Induktionsannahme für t' und folgern hieraus, daß $(\lambda x\,.\,t')\;s \equiv t^*$ berechenbar ist.
(*Dies deckt den Schritt der äußeren Induktion für t zusammen mit dem Basisfall von s ab.*)

3. t^* hat die Gestalt $(\lambda x.t)\,s'$, wobei $s \xrightarrow{\beta} s'$ gilt. Da s berechenbar ist, gilt nach Lemma 2.4.20.2., daß auch s' berechenbar ist und in weniger Schritten normalisiert als s. Wir verwenden nun die Induktionsannahme für s' und folgern hieraus, daß $(\lambda x\,.\,t)\;s' \equiv t^*$ berechenbar ist.
(*Dies deckt den Schritt der inneren Induktion für s im Basisfall und im Induktionsschritt für t ab.*)

Damit sind die Voraussetzungen von Lemma 2.4.20.3. erfüllt und wir dürfen folgern, daß $(\lambda x\,.\,t)\;s$ berechenbar ist. Aus Definition 2.4.19 folgt nun die Berechenbarkeit von $\lambda x\,.\,t$. □

[78] Für die Theorie der einfachen Typen wäre dies – wie bereits erwähnt – sicherlich nicht notwendig. Die hier präsentierte Beweismethodik läßt sich aber leichter generalisieren, wenn wir sie bereits im Spezialfall möglichst universell formulieren.

Das folgende Lemma ist eine weitere Verstärkung des angestrebten Theorems. Es läßt sich leichter in einem Induktionsschritt verwenden und hat die gewünschte Aussage als Spezialfall.

Lemma 2.4.22

Es sei t ein beliebiger typisierbarer λ-Term. $x_1 \ldots x_n$ seien die freien Variablen von t und $b_1 \ldots b_n$ seien beliebige berechenbare λ-Terme, wobei b_i denselben Typ wie die Variable x_i in t hat.
Dann ist der Term $t[b_1 \ldots b_n \,/\, x_1 \ldots x_n]$ berechenbar.

Beweis: Wir führen einen Induktionsbeweis über die Struktur von $t{=}t[x_1 \ldots x_n]$.

- Falls t eine Variable x_i ist, dann $t[b_1 \ldots b_n \,/\, x_1 \ldots x_n] = b_i$ und somit nach Voraussetzung berechenbar.
- Es sei die Behauptung gezeigt für typisierbare λ-Terme u und f.
- Es gibt zwei Möglichkeiten für die Struktur von t.
  – Falls t die Gestalt $\lambda x\,.\,u$ hat, dann ist $t[b_1 \ldots b_n \,/\, x_1 \ldots x_n] = \lambda x\,.\,u[b_1 \ldots b_n \,/\, x_1 \ldots x_n]$. (Die Variable x ist *nicht* frei in t). Nach Induktionsannahme ist für alle berechenbaren Terme b der Term $u[b_1 \ldots b_n, b \,/\, x_1 \ldots x_n, x]$ ein berechenbarer Term. Mit Lemma 2.4.21 folgt hieraus die Berechenbarkeit von $t[b_1 \ldots b_n \,/\, x_1 \ldots x_n]$.
  – Falls t die Gestalt $f\;u$ hat, dann ist $t[b_1 \ldots b_n \,/\, x_1 \ldots x_n] = f[b_1 \ldots b_n \,/\, x_1 \ldots x_n]\;\; u[b_1 \ldots b_n \,/\, x_1 \ldots x_n]$. Nach Induktionsannahme sind beide Teilterme einzeln berechenbar, woraus gemäß Definition 2.4.19 die Berechenbarkeit der Applikation $f[b_1 \ldots b_n \,/\, x_1 \ldots x_n]\;\; u[b_1 \ldots b_n \,/\, x_1 \ldots x_n]$. □

Nach all diesen – zugegebenermaßen recht aufwendigen – Vorarbeiten ist der Beweis der starken Normalisierbarkeit typisierbarer λ-Terme relativ einfach.

Satz 2.4.23 (Starke Normalisierbarkeit typisierbarer λ-Terme)

1. *Alle typisierbaren λ-Terme sind berechenbar.*
2. *Alle typisierbaren λ-Terme sind stark normalisierbar.*

Beweis: Es sei t ein beliebiger typisierbaren λ-Term.

1. Sind $x_1 \ldots x_n$ die freien Variablen von t, so ist $t = t[x_1 \ldots x_n \,/\, x_1 \ldots x_n]$. Da alle x_i berechenbar sind (Lemma 2.4.20.4.), folgt mit Lemma 2.4.22 die Berechenbarkeit von t.
2. Nach Teil 1. ist t berechenbar. Mit Lemma 2.4.20.1. folgt hieraus, daß t auch stark normalisierbar ist. □

Die starke Normalisierbarkeit typisierbarer λ-Terme hat zur Folge, daß wir zur Auswertung typisierbarer Terme sehr effiziente Reduktionsstrategien einsetzen können, deren Verwendung im ungetypten λ-Kalkül Gefahr laufen würde, zu nichtterminierenden Reduktionsfolgen zu führen. Das folgende Beispiel zeigt, wie sich dieser Unterschied auswirkt.

Beispiel 2.4.24

Wir reduzieren den Term `trice(trice(λx.x))`, wobei `trice` $\equiv$ `λf.λx. f(f(f x))` (vgl. Beispiel 2.4.13) mit drei verschiedenen Strategien.

Rightmost (Call-by-value):

$\quad$ `trice(trice(λx.x))`
$\xrightarrow{\beta}$ `trice (λx. (λx.x) ( (λx.x) ( (λx.x) x)))`
$\xrightarrow{\beta}$ `trice (λx. (λx.x) ( (λx.x) x))`
$\xrightarrow{\beta}$ `trice (λx. (λx.x) x)`
$\xrightarrow{\beta}$ `trice (λx.x)`
$\xrightarrow{\beta}$ `λx. (λx.x) ( (λx.x) ( (λx.x) x))`
$\xrightarrow{\beta}$ `λx. (λx.x) ( (λx.x) x)`
$\xrightarrow{\beta}$ `λx. (λx.x) x`
$\xrightarrow{\beta}$ `λx.x`

Rightmost-maxdepth:

trice(trice(λx.x))
$\xrightarrow{\beta}$ trice (λx. (λx.x) ((λx.x) ((λx.x) x)))
$\xrightarrow{\beta}$ λx. (λx. (λx.x) ((λx.x) ((λx.x) x)))
((λx. (λx.x) ((λx.x) ((λx.x) x)))
((λx. (λx.x) ((λx.x) ((λx.x) x))) x))
$\xrightarrow{\beta}$ λx. (λx. (λx.x) ((λx.x) ((λx.x) x)))
((λx. (λx.x) ((λx.x) ((λx.x) x)))
((λx.x) ((λx.x) ((λx.x) x))))
$\xrightarrow{3}$ λx. (λx. (λx.x) ((λx.x) ((λx.x) x))) ((λx. (λx.x) ((λx.x) ((λx.x) x))) x)
$\xrightarrow{\beta}$ λx. (λx. (λx.x) ((λx.x) ((λx.x) x))) ((λx.x) ((λx.x) ((λx.x) x)))
$\xrightarrow{3}$ λx. (λx. (λx.x) ((λx.x) ((λx.x) x))) x
$\xrightarrow{\beta}$ λx. (λx.x) ((λx.x) ((λx.x) x))
$\xrightarrow{3}$ λx.x

Leftmost (Call-by-name):

trice(trice(λx.x))
$\xrightarrow{\beta}$ λx. (trice (λx.x)) ((trice (λx.x)) ((trice (λx.x)) x))
$\xrightarrow{\beta}$ λx. (λx. (λx.x) ((λx.x) ((λx.x) x))) ((trice (λx.x)) ((trice (λx.x)) x))
$\xrightarrow{\beta}$ λx. ((λx.x) ((λx.x) ((λx.x) (trice (λx.x))))) ((trice (λx.x)) x)
$\xrightarrow{\beta}$ λx. ((λx.x) ((λx.x) (trice (λx.x)))) ((trice (λx.x)) x)
$\xrightarrow{\beta}$ λx. ((λx.x) (trice (λx.x))) ((trice (λx.x)) x)
$\xrightarrow{\beta}$ λx. (trice (λx.x)) ((trice (λx.x)) x)
$\xrightarrow{\beta}$ λx. (trice (λx.x)) ((trice (λx.x)) x)
$\xrightarrow{6}$ λx. (trice (λx.x)) x
$\xrightarrow{6}$ λx. x

2.4.5.3 Konfluenz

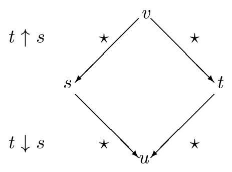
Für den ungetypten λ-Kalkül ist bekannt, daß jede terminierende Reduktionsfolge eines λ-Terms zu derselben Normalform führt. Der Beweis dieser Aussage (Das Church-Rosser Theorem auf Seite 64) ist wegen der Möglichkeit nichtterminierender Reduktionen allerdings ausgesprochen kompliziert. Für typisierbare λ-Terme ist wegen der starken Normalisierbarkeit der Beweis des Church-Rosser Theorem, den wir in diesem Abschnitt präsentieren, erheblich einfacher. Wir werden zu diesem Zwecke auf die Terminologie der Termersetzungssysteme zurückgreifen. In dieser Denkweise bedeutet die Eindeutigkeit der Normalform eines λ-Terms t, daß alle in t beginnenden Reduktionsfolgen wieder zu demselben Term zusammengeführt werden können, also *konfluent* sind. Um dies präzise zu definieren, führen wir einige Notationen ein.

Definition 2.4.25

Es seien t und s Terme und $\xrightarrow{r}$ eine Reduktionsrelation.

1. $\underline{t \xrightarrow{n} s}$ *gilt, falls s sich aus t durch n Reduktionsschritte ergibt.*
   – $t \xrightarrow{0} s$, *falls* t=s
   – $t \xrightarrow{n+1} s$, *falls es einen Term u gibt mit $t \xrightarrow{r} u$ und $u \xrightarrow{n} s$*
2. $\underline{t \xrightarrow{*} s}$ *gilt, falls s sich aus t durch endlich viele Reduktionen ergibt (d.h. $t \xrightarrow{n} s$ für ein $n \in \mathbb{N}$).*
3. $\underline{t \xrightarrow{+} s}$ *gilt, falls $t \xrightarrow{n} s$ für ein n>0.*
4. $\underline{t{\uparrow}s}$, *falls es einen Term u gibt mit $u \xrightarrow{*} t$ und $u \xrightarrow{*} s$*
5. $\underline{t{\downarrow}s}$, *falls es einen Term u gibt mit $t \xrightarrow{*} u$ und $s \xrightarrow{*} u$*

Diese Definitionen gelten für beliebige Reduktionsrelation, die – wie die β-Reduktion – nicht notwendigerweise transitiv sein müssen. $t{\uparrow}s$ bedeutet, daß t und s aus demselben Term u entstanden sind, während $t{\downarrow}s$ besagt, daß t und s zu demselben Term u zusammenfließen können.



Konfluenz bedeutet nun, daß das nebenstehende Diagramm kommutieren muß. Um diese Eigenschaft für die Typentheorie nachzuweisen, führen wir zusätzlich noch den Begriff der *lokalen Konfluenz* ein. Er besagt, daß t und s zu demselben Term u zusammenfließen können, wenn sie in einem Reduktionsschritt aus demselben Term entstanden sind.

Definition 2.4.26 (Konfluenz)

Es sei $\xrightarrow{r}$ eine Reduktionsrelation.

1. $\xrightarrow{r}$ *ist* $\underline{\textit{konfluent}}$, *wenn für alle Terme t und s gilt: aus $t{\uparrow}s$ folgt $t{\downarrow}s$*
2. $\xrightarrow{r}$ *ist* $\underline{\textit{lokal konfluent}}$, *wenn für alle Terme t und s gilt:*
   falls es einen Term u gibt mit $u \xrightarrow{r} t$ und $u \xrightarrow{r} s$, dann folgt $t{\downarrow}s$

Für stark normalisierbare Reduktionsrelationen kann man nun zeigen, daß lokale Konfluenz und Konfluenz identisch sind.

Lemma 2.4.27

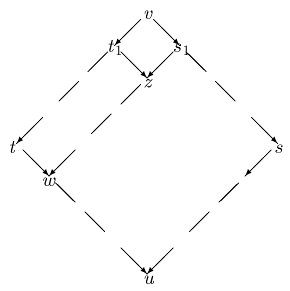
Es sei $\xrightarrow{r}$ eine stark normalisierbare und lokal konfluente Reduktionsrelation.
Dann ist $\xrightarrow{r}$ auch konfluent.

Beweis: Da $\xrightarrow{r}$ stark normalisierbar ist, gibt es für jeden Term v eine Anzahl n derart, daß jede in v beginnende Reduktionsfolge in maximal n Schritten terminiert. Wir zeigen durch Induktion über die Anzahl der Reduktionsschritte von v: *Gilt $v \xrightarrow{*} t$ und $v \xrightarrow{*} s$, dann folgt $t{\downarrow}s$.*

- Falls v in 0 Schritten reduziert, so folgt aus $v \xrightarrow{*} t$ und $v \xrightarrow{*} s$, daß v=t=s sein muß. $t{\downarrow}s$ gilt somit trivialerweise.
- Es sei für alle Terme r, die in n Schritten reduzieren gezeigt, daß aus $r \xrightarrow{*} t$ und $r \xrightarrow{*} s$ folgt $t{\downarrow}s$.
- v reduziere in n+1 Schritten und es gelte $v \xrightarrow{*} t$ und $v \xrightarrow{*} s$.

  Falls $v \xrightarrow{0} t$ oder $v \xrightarrow{0} s$, so folgt v=t oder v=s und $t{\downarrow}s$ gilt trivialerweise.

  Andernfalls gibt es Terme t_1 und s_1 mit $v \xrightarrow{r} t_1 \xrightarrow{*} t$ und $v \xrightarrow{r} s_1 \xrightarrow{*} s$. Da v in n+1 Schritten reduziert, müssen t_1 und s_1 notwendigerweise in maximal n Schritten terminieren und die Induktionsannahme ist anwendbar.

  Wegen der lokalen Konfluenz gilt $t_1{\downarrow}s_1$, d.h. es gibt einen Term z mit $t_1 \xrightarrow{*} z$ und $s_1 \xrightarrow{*} z$.

  Aufgrund der Induktionsannahme für t_1 folgt aus $t_1 \xrightarrow{*} t$ und $t_1 \xrightarrow{*} z$, daß $t{\downarrow}z$ gilt. Es gibt also einen Term w mit $t \xrightarrow{*} w$ und $z \xrightarrow{*} w$.

  Für den Term w gilt $s_1 \xrightarrow{*} z \xrightarrow{*} w$. Da außerdem $s_1 \xrightarrow{*} s$ gilt, folgt mit der Induktionsannahme für s_1, daß $w{\downarrow}s$ gilt, d.h. daß es ein u gibt mit $w \xrightarrow{*} u$ und $s \xrightarrow{*} u$.

  Für diesen Term gilt auch $t \xrightarrow{*} w \xrightarrow{*} u$ und damit folgt $t{\downarrow}s$.

Damit ist die Induktionsbehauptung bewiesen: aus $t{\uparrow}s$ folgt immer $t{\downarrow}s$. □

Lemma 2.4.28

Die β-Reduktion $\xrightarrow{\beta}$ des λ-Kalküls ist lokal konfluent.

Beweis:

Es seien s, t zwei verschiedene λ-Terme und es gelte $u \xrightarrow{\beta} t$ und $u \xrightarrow{\beta} s$ für einen λ-Term u. Da s und t durch verschiedene β-Reduktionen aus u entstehen müssen, gibt es in u einen Teilterm der Gestalt $v\,w$ oder $(\lambda x.v)\ w$, so daß s durch Reduktion von v, w oder des äußeren β-Redex und t durch eine andere Reduktion entsteht.
Das nebenstehende Diagramm zeigt, wie für alle drei Möglichkeiten, einen Term der Form $(\lambda x.v)\ w$ zu reduzieren, die Reduktionskette wieder zusammengeführt werden kann. (Die Betrachtung von Termen der Gestalt $v\,w$ ist implizit darin enthalten.) Damit ist die β-Reduktion lokal konfluent.

$$\begin{tikzcd}
(\lambda x.v)\ w \arrow[rr] \arrow[dr] \arrow[dd] & & v[w/x] \arrow[dr] \arrow[dd] & \\
& (\lambda x.v)\ w' \arrow[rr] \arrow[dd] & & v[w'/x] \arrow[dd, dashed] \\
(\lambda x.v')\ w \arrow[rr] \arrow[dr] & & v'[w/x] \arrow[dr] & \\
& (\lambda x.v')\ w' \arrow[rr, dashed] & & v'[w'/x]
\end{tikzcd}$$

□

Um das Church-Rosser Theorem für typisierbare λ-Terme zu beweisen, brauchen wir jetzt nur noch die beiden Lemmata 2.4.27 und 2.4.28 zusammenzusetzen.

Satz 2.4.29 (Church-Rosser Theorem der Typentheorie)

In der Typentheorie ist die β-Reduktion $\xrightarrow{\beta}$ konfluent.

Als ein Korollar erhalten wir die Eindeutigkeit der Normalform typisierbarer λ-Terme.

Korollar 2.4.30

Jeder typisierbare λ-Term hat eine eindeutig bestimmte Normalform.

2.4.5.4 Entscheidbarkeit

Aufgrund der starken Normalisierbarkeit und durch das Church-Rosser Theorem wissen wir, daß typisierbare λ-Terme sich sehr gut als Basis einer effizient zu verarbeitenden Programmiersprache eignen. Darüber hinaus folgt aber auch noch, daß die Typentheorie sehr gut geeignet ist für das automatisierte Schließen über das Verhalten von Programmen. Denn die Gleichheit zweier typisierbarer λ-Terme s und t läßt sich nun dadurch beweisen, daß man zunächst mit dem Hindley-Milner Algorithmus (Abbildung 2.10) ihr prinzipielles Typschema bestimmt, dann die beiden Terme normalisiert und dann gemäß Korollar 2.3.35 (Seite 65) auf Kongruenz überprüft.

Satz 2.4.31

Die Gleichheit typisierbarer λ-Terme ist entscheidbar.

Insgesamt wissen wir also, daß alle wesentlichen extensionalen Eigenschaften typisierbarer λ-Terme – im Gegensatz zu den untypisierten λ-Termen (vgl. Abschnitt 2.3.7) – entscheidbar sind.

- Das Halteproblem – die *Terminierung* von Algorithmen für ein gegebenes Argument – ist trivialerweise entscheidbar, da alle Funktionen terminieren.
- *Totalität* von typisierbaren λ-Funktionen ist aus dem gleichen Grunde entscheidbar.
- Die *Korrektheit* einer Funktion, also die Frage '$f\,u = t$' ist wegen Theorem 2.4.31 entscheidbar.
- Das gleiche gilt für die *Äquivalenz* von Programmen: '$f = g$'.
- Darüber hinaus lassen sich alle Eigenschaften eines Programms automatisch beweisen, die durch seinen Typ ausgedrückt werden können, da Typkorrektheit gemäß Theorem 2.4.10 ebenfalls entscheidbar ist.

Durch die Hinzunahme der Typdisziplin haben wir also die Möglichkeiten einer automatischen Überprüfung von Programmeigenschaften erheblich gesteigert.

2.4.6 Schließen über den Wert getypter λ-Terme

Bei unseren bisherigen Betrachtungen haben wir den λ-Kalkül und die Typdisziplin als separate Kalküle behandelt, um ihre Eigenschaften isoliert voneinander untersuchen zu können. Es ist jedoch nicht sehr schwer, den λ-Kalkül in die Typentheorie zu integrieren, also das Schließen über Gleichheit mit dem Schließen über Typzugehörigkeit zu verbinden. Man muß hierzu nur einen gemeinsamen Oberbegriff schaffen, nämlich das Schließen über die *Gleichheit zweier Terme innerhalb eines Typs*. Statt s=t und $t \in T$ hätte man als Konklusion einer Sequenz also ein Urteil der Form

$$s = t \ \in\ T$$

welches besagt, daß s und t Elemente von T sind und innerhalb dieses Typs als gleich anzusehen sind.[79] Die Typzugehörigkeitsrelation $t \in T$ ergibt sich hieraus, indem man s und t identisch wählt. Man kann also $t \in T$ als definitorische Abkürzung für $t = t \in T$ ansehen. Dementsprechend kann man auch die Regeln des λ-Kalküls und die der Typentheorie vereinigen und erhält zum Beispiel die folgenden Regeln:

$\Gamma \vdash f\,t = g\,u \ \in\ T$ **by** `applyEq` S
  $\Gamma \vdash f = g \ \in\ S{\to}T$
  $\Gamma \vdash t = u \ \in\ S$

$\Gamma \vdash \lambda x.t = \lambda y.u \ \in\ S{\to}T$ **by** `lambdaEq`
  $\Gamma,\ x'{:}S \vdash t[x'/x] = u[x'/y] \ \in\ T$
  $\Gamma \vdash S \in \mathbb{U}$

$\Gamma \vdash (\lambda x.u)\ s = t \ \in\ T$ **by** `reduction`
  $u[s/x] = t \ \in\ T$

Diese Integration von Typdisziplin und Berechnung in einem Kalkül werden wir im Kapitel 3 weiter vertiefen.

2.4.7 Die Curry-Howard Isomorphie

Bisher haben wir Typen im wesentlichen als Ausdrücke angesehen, welche die Bereiche kennzeichnen, zu denen ein durch einen λ-Term beschriebenes Objekt gehört. Wir haben soeben gezeigt, wie man λ-Kalkül und Typentheorie auf relativ einfache Art integrieren kann. Aber auch die Prädikatenlogik läßt sich im Prinzip mit der Typentheorie vereinigen. Vergleicht man nämlich die Regeln der Typentheorie mit denen der Prädikatenlogik aus Abbildung 2.6 (Seite 43), so entdeckt man eine gewisse Verwandschaft zwischen den Regeln für die Implikation und denen für die Einführung kanonischer und nichtkanonischer Terme in der Typentheorie. Diese Analogie wurde erstmalig von H. Curry [Curry *et.al.*, 1958], W. Tait [Tait, 1967] und W. Howard [Howard, 1980] beschrieben und wird in der Literatur als *Curry-Howard Isomorphie* bezeichnet.

Die Regel `lambda_i` der λ-Abstraktion besagt, daß ein λ-Term $\lambda x.t$ des Typs $S{\to}T$ dadurch aufgebaut werden kann, daß man im Beweis angibt, wie aus einer Variablen x vom Typ S ein Term t vom Typ T aufgebaut wird. Dies ist sehr ähnlich zu der Einführungsregel `imp_i` für die Implikation: um einen Beweis für $A \Rightarrow B$ aufzubauen, muß man zeigen wie aus der Annahme A ein Beweis für B aufgebaut wird.

$\Gamma \vdash \lambda x.t \in S{\to}T$ **by** `lambda_i`
  $\Gamma,\ x'{:}S \vdash t[x'/x] \in T$
  $\Gamma \vdash S \in \mathbb{U}$

$\Gamma \vdash A \Rightarrow B$ **by** `imp_i`
  $\Gamma, A \vdash B$

Von der Struktur her kann `lambda_i` also als Verallgemeinerung der Regel `imp_i` angesehen werden. Die Typen S und T entsprechen den Formeln A und B und der Funktionenraumkonstruktor $\to$ der Implikation $\Rightarrow$. Allerdings verarbeitet die Typentheorie noch weitere Informationen, nämlich die Elemente der entsprechenden Typen und ihren strukturellen Aufbau. Dieser Zusammenhang gilt in ähnlicher Form auch für die Regel `apply_i` für die Applikation und die Eliminationsregel der Implikation, wenn man sie in ihrer ursprünglichen Form als Regel des *modus ponens* niederschreibt, welche auf der Konklusion statt auf den Hypothesen arbeitet.[80] Um B zu beweisen, reicht es aus, $A \Rightarrow B$ und A zu zeigen.

[79] Daß es sinnvoll sein kann, die Gleichheit vom Typ abhängig zu machen, zeigt das Beispiel der Darstellung von Rationalzahlen als Paare ganzer Zahlen. Verschiedene Paare können durchaus dieselbe rationale Zahl bezeichnen.

[80] Daß diese Form im Sequenzenkalkül nicht benutzt wird, hat strukturelle Gründe. Die Regel `imp_e` verwendet existierende Hypothesen, während `modus ponens` die Implikation als Konklusion eines Teilziels erzeugt und hierzu die Formel A benötigt.

$\Gamma \vdash f\, t \in T$ **by** `apply_i` S
  $\Gamma \vdash f \in S \to T$
  $\Gamma \vdash t \in S$

$\Gamma \vdash B$ **by** `modus ponens` A
  $\Gamma \vdash A \Rightarrow B$
  $\Gamma \vdash A$

Wieder ist die typentheoretische Regel allgemeiner, da sie neben der Angabe, wie denn die Formeln (Typen) aufzubauen sind, auch noch sagt, wie sich die entsprechenden Terme zusammensetzen.

Der Zusammenhang zwischen Logik und Typentheorie wird noch deutlicher, wenn wir logische Aussagen nicht nur von ihrem Wahrheitsgehalt her betrachten, sondern uns auch noch für die Menge aller Beweise dieser Aussage interessieren. Wenn wir zum Beispiel bei der Verwendung von `imp_i` zeigen, wie wir aus der Annahme A die Formel B beweisen, dann geben wir genau genommen eine Konstruktion an, wie wir aus einem beliebigen Beweis a für A einen Beweis b für B erzeugen. Da hierbei a mehr oder weniger eine freie Variable in b ist, können wir einen Term der Art $\lambda a.b$ als Beweis für $A \Rightarrow B$ ansehen. Wenn wir umgekehrt bei der Verwendung von `modus ponens` einen Beweis pf_{AB} von $A \Rightarrow B$ und einen Beweis a von A kennen, dann wissen wir genau, wie wir A beweisen: wir wenden einfach den Beweis von $A \Rightarrow B$ auf den von A an, was wir kurz mit $pf_{AB}\ a$ bezeichnen können.

Zwischen der einfachen Typentheorie und dem Implikationsfragment der Logik (der Kernbestandteil des logischen Schließens) besteht also eine *Isomorphie*, wenn wir folgende Konzepte miteinander identifizieren.

Typ	Formel
Variable vom Typ S	Annahme der Formel S
Term vom Typ T (mit freien Variablen vom Typ S_i)	Beweis von T (unter den Annahmen S_i)
Typechecking	Beweisführung
(Regel der) λ-Abstraktion	$\Rightarrow$-Einführung
(Regel der) Applikation	$\Rightarrow$-Elimination

Diese Isomorphie zeigt, daß die Typentheorie nicht nur für das Schließen über Werte und Typzugehörigkeit von Programmen geeignet ist, sondern im Prinzip auch die Prädikatenlogik simulieren kann.[81] Damit bietet sie einen vielversprechenden Ansatz im Hinblick auf einen einheitlichen Formalismus für mathematisches Schließen und Programmierung.

2.4.8 Die Ausdruckskraft typisierbarer λ-Terme

Wir haben die Typdisziplin auf λ-Termen eingeführt, um den λ-Kalkül besser kontrollieren zu können, und bewußt in Kauf genommen, einen Teil der Ausdruckskraft des λ-Kalküls zu verlieren. Aus der theoretischen Informatik ist bekannt, daß jedes entscheidbare Berechenbarkeitskonzept, das – wie die Typentheorie – nur totale Funktionen zuläßt, dazu führt, daß manche berechenbare totale Funktion nicht mehr beschreibbar ist. Gewisse Einbußen müssen wir also hinnehmen. In diesem Abschnitt wollen wir nun untersuchen, wie viel Ausdruckskraft wir im typisierten λ-Kalkül behalten haben. Wie das folgende Beispiel zeigt, sind die elementarsten Funktionen auf natürlichen Zahlen leicht zu typisieren.

Beispiel 2.4.32

In Definition 2.3.17 haben wir natürliche Zahlen durch Church-Numerals beschrieben. Dabei war das Church-Numeral einer Zahl n definiert durch

$$\overline{n} \equiv \lambda f.\lambda x.\ f^n\, x$$

Es läßt sich relativ leicht zeigen, daß jedes Church-Numeral den prinzipiellen Typ $(X \to X) \to X \to X$ besitzt (vergleiche Beispiel 2.4.11.1 auf Seite 77). Wir könnten daher den Datentyp $\mathbb{N}$ als definitorische Abkürzung ansehen.

[81] So kann man zum Beispiel die Konjunktion $A \wedge B$ mit der Bildung eines Produktraumes $S \times T$ vergleichen: Ist a ein Beweis für A und b einer für B, so kann das Paar $\langle a,b\rangle$ als Beweis für $A \wedge B$ angesehen werden. Wieder entspräche die Einführungsregel der Regel für die kanonischen Elemente und die Eliminationsregel der Regel für die nichtkanonischen Elemente. Ähnlich sieht es aus mit den anderen logischen Operatoren. Da sich allerdings nicht alle hierzu nötigen Typkonstrukte in der einfachen Typentheorie simulieren lassen, werden wir hierauf erst im Kapitel 3 im Detail zurückkommen.

$$\mathbb{N} \equiv (X \to X) \to X \to X.$$

Mit dieser Abkürzung können wir überprüfen, daß gilt

$$\begin{array}{lll} \lambda n.\lambda f.\lambda x.\ n\ f\ (f\ x) & \equiv \mathbf{s} & \in \mathbb{N} \to \mathbb{N} \\ \lambda m.\lambda n.\lambda f.\lambda x.\ m\ f\ (n\ f\ x) & \equiv \mathbf{add} & \in \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ \lambda m.\lambda n.\lambda f.\lambda x.\ m\ (n\ f)\ x & \equiv \mathbf{mul} & \in \mathbb{N} \to \mathbb{N} \to \mathbb{N} \end{array}$$

Auch die in Definition 2.3.15 angegebenen Erweiterungen für Paare können durch den Hindley-Milner Algorithmus ohne Probleme typisiert werden.

Beispiel 2.4.33

$$\begin{array}{ll} \lambda p.\ p\, s\, t & \equiv \langle s,t\rangle \ \in\ (X \to Y \to Z) \to Z \\ pair\ (\lambda x.\lambda y.u) & \equiv \mathsf{let}\ \langle x,y\rangle = pair\ \mathsf{in}\ u\ \in\ Z \end{array}$$

Dabei ist $s \in X$, $t \in Y$, $pair \in (X \to Y \to Z) \to Z$ und $u \in Z$. Es erscheint damit legitim, den Produktraum $S \times T$ zweier Typen S und T wie folgt zu definieren:

$$S \times T \equiv (S \to T \to Z) \to Z$$

Auch die beiden Projektionen arbeiten in diesem Sinne korrekt. Bei Eingabe von $\langle s,t\rangle \in (S \to T \to Z) \to Z$ liefert die erste Projektion den Term $s \in S$ und die zweite den Term $t \in T$. Dies deckt sich mit der prinzipiellen Typisierung der beiden Funktionsanwendungen.

$$\begin{array}{l} \langle s,t\rangle\ (\lambda x.\lambda y.x) \equiv \langle s,t\rangle.1 \in S \\ \langle s,t\rangle\ (\lambda x.\lambda y.y) \equiv \langle s,t\rangle.2 \in T \end{array}$$

Man beachte, daß hierbei der Typ Z eine freie Variable ist, während die Typen S und T als die Typen von s und t bereits festliegen.

Typisierungen sind ohne weiteres auch möglich für boolesche Operatoren und Listen. Dennoch zeigen sich bei etwas näherem Hinsehen, daß diese Typisierung nicht ganz dem entspricht, was man gerne erreichen möchte.

Beispiel 2.4.34

Die Typisierung der booleschen Operatoren aus Definition 2.3.13 ergibt

$$\begin{array}{ll} \lambda x.\lambda y.x & \equiv \mathbf{T} \in X \to Y \to X \\ \lambda x.\lambda y.y & \equiv \mathbf{F} \in X \to Y \to Y \\ b\, s\, t & \equiv \mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t \in Z \end{array}$$

Dabei ist im letzten Fall $b \in X \to Y \to Z$, $s \in X$ und $t \in Y$. Wenn nun **T** und **F** zum gleichen Typ gehören sollen und als vorgesehene Eingaben im Conditional auftauchen sollen, so müssen X, Y und Z identisch sein und wir bekommen

$$\mathbb{B} \equiv X \to X \to X.$$

Dies würde aber verlangen, daß die Terme s und t innerhalb des Conditional immer vom gleichen Typ sein müssen.

Während die Typeinschränkungen bei den Termen des Conditionals noch akzeptabel erscheinen mögen (sie gelten für die meisten typisierten Sprachen), zeigt das folgende Beispiel, daß die bisherigen Typkonstrukte für praktische Zwecke nicht ganz ausreichen.

Beispiel 2.4.35

1. Die Exponentiierungsfunktion **exp** war in Definition 2.3.17 angegeben als

   $$\mathbf{exp} \equiv \lambda m.\lambda n.\lambda f.\lambda x.\ n\ m\ f\ x$$

   Der Prinzipielle Typ dieser Funktion ist

   $$(X \to Y \to Z \to T) \to X \to Y \to Z \to T$$

   wobei $m \in X \to Y \to Z \to T$, $n \in X$, $f \in Y$ und $x \in Z$ ist. Das Dilemma entsteht nun dadurch, daß der Typ von m und n gleich – nämlich $\mathbb{N}$ – sein soll, was aber auf keinen Fall gewährleistet werden kann.

2. Das gleiche Problem entsteht bei der Typisierung der einfachen primitiven Rekursion.

   $\mathsf{PRs}[base,h] \equiv \lambda\mathtt{n}.\ \mathtt{n}\ h\ base$

   Der Prinzipielle Typ dieser Funktion ist

   $(\mathtt{X}\to\mathtt{Y}\to\mathtt{Z}) \to \mathtt{X} \to \mathtt{Y} \to \mathtt{Z}$

   Der Konzeption nach sollte *base* vom Typ $\mathbb{N}$ sein, h vom Typ $\mathbb{N}\to\mathbb{N}$, die Eingabe `n` vom Typ $\mathbb{N}$ und ebenso das Ergebnis. Setzt man dies ein, so erhält man als Typ für die einfache primitiven Rekursion

   $((\mathbb{N}\to\mathbb{N})\to\mathbb{N}\to\mathbb{N}) \to (\mathbb{N}\to\mathbb{N}) \to \mathbb{N} \to \mathbb{N}$

   Dies bedeutet, daß die Eingabe `n` gleichzeitig den Typ $\mathbb{N}$ und $(\mathbb{N}\to\mathbb{N})\to\mathbb{N}\to\mathbb{N}$ haben muß. Dies aber ist nicht möglich, da die Gleichung

   $\mathbb{N} = (\mathbb{N}\to\mathbb{N})\to\mathbb{N}\to\mathbb{N}$

   keine Lösung hat.

Das Hauptproblem im letzten Beispiel ist die uneingeschränkte polymorphe Verwendung von `n` als Element von $\mathbb{N} \equiv (\mathtt{X}\to\mathtt{X}) \to \mathtt{X}\to\mathtt{X}$ für *verschiedene* Instanzen der freien Variablen `X` innerhalb *derselben* Anwendung. Es ist daher nicht möglich, die Variable `X` mit einem einheitlichen Wert zu belegen. Stattdessen müßte für jede Zahl n dem Church-Numeral $\overline{n}$ eine andere Instanz von $(\mathtt{X}\to\mathtt{X}) \to \mathtt{X}\to\mathtt{X}$ zugewiesen werden. Damit ist es nicht möglich, die primitive Rekursion ihrem Sinn gemäß zu typisieren.

Dies bedeutet, daß die Ausdruckskraft der einfachen Typentheorie unter derjenigen der primitiv rekursiven Funktionen liegt, was für praktische Anwendungen völlig unakzeptabel ist.[82] Somit ist die *einfache* Typentheorie als Kalkül zum Schließen über Programme ebenso ungeeignet wie der ungetypte λ-Kalkül. Dies liegt aber nicht so sehr am Typkonzept als solches, sondern daran, daß das bisher betrachtete Typkonstrukt – die simple Funktionenraumbildung – nicht ausdrucksstark genug ist. Im Anbetracht der vielen guten Eigenschaften, die wir von der einfachen Typentheorie nachweisen konnten, lohnt es sich, nach einer Erweiterung des Typsystems zu suchen, welche aus praktischer Hinsicht zufriedenstellend ist.

2.5 Diskussion

Wir haben in diesem Kapitel formale Kalküle zum Schließen über Logik, Berechnung und Typzugehörigkeit vorgestellt und ihre Eigenschaften untersucht. Mit der *Prädikatenlogik* können wir die *Struktur* logischer Aussagen analysieren und diejenigen unter ihnen beweisen, deren Wahrheitsgehalt ausschließlich aus dieser Struktur folgt. Mit dem *λ-Kalkül* können wir den *Wert* eines durch einen Ausdruck beschriebenen Objektes bestimmen und entsprechend über die Gleichheit von Werten Schlüsse ziehen. Die *einfache Typentheorie* stellt die Beziehung zwischen einem Programm (λ-Term) und seinen Eigenschaften (Typ) her und wird darüber hinaus dazu benutzt, um durch Typrestriktionen diejenigen Probleme zu umgehen, welche sich aus der Mächtigkeit des λ-Kalküls ergeben. Ihre bisherige Formulierung ist jedoch noch zu einfach und muß zugunsten einer verbesserten Ausdruckskraft um weitere Konzepte erweitert werden.

Jeder einzelne der drei Kalküle ist für ein einzelnes Teilgebiet des formalen Schließens über Programme sehr gut geeignet. In allen erwies sich der Stil des Sequenzenkalküls als geeignetes Mittel, die semantischen Zusammenhänge durch formale Regeln auszudrücken. Prinzipiell wäre es nun möglich, einen einheitlichen Kalkül für das Schließen über Programme durch eine simple Vereinigung aller Regeln der drei Kalküle zu erreichen. Dies würde jedoch die Schnittstelle *zwischen* den Kalkülen völlig ungeklärt belassen. Eine sehr interessante Möglichkeit, einen integrierten und leistungsfähigen Kalkül zum einheitlichen Schließen über alle Aspekte der Mathematik und Programmierung aufzubauen, ergibt sich aus den am Ende des letzten Abschnitts andiskutierten Ideen.

[82] Dies betrifft weniger die Klasse der Funktionen, die man im einfach typisierten λ-Kalkül ausdrücken kann, als die Programmiertechniken, die man einsetzen darf. Schon die einfachste Form der Schleife ist nicht erlaubt.

- Durch die Einführung einer *typisierten Gleichheit* (siehe Abschnitt 2.4.6), die ohnehin einem natürlichen Verständnis der Gleichheit näher kommt als die ungetypte Gleichheit, können wir das Schließen über Berechnung (d.h. den λ-Kalkül) in die Typentheorie *integrieren*.
- Die in Abschnitt 2.4.7 angesprochene Curry-Howard Isomorphie zwischen Logik und Typkonstrukten ermöglicht es, die Prädikatenlogik innerhalb der Typentheorie zu *simulieren*.

Eine entsprechend ausdrucksstarke Typentheorie wird also für das formal-mathematische Schließen und die Programmierung eine ähnlich grundlegende Rolle bekommen wie die Mengentheorie für die abstrakte Mathematik. Um dies zu erreichen, müssen wir allerdings die einfache Typentheorie gehörig ausdehnen.

Dabei können wir zwar die grundlegenden *Ideen* der einfachen Typentheorie beibehalten, müssen uns aber von dem Gedanken lösen, die Typentheorie als eine reine Ergänzung zum einfachen λ-Kalkül zu betrachten. So ist zum Beispiel eine naive Typisierung von natürlichen Zahlen auf der Basis der Church Numerals (vergleiche Abschnitt 2.4.8) problematisch. Die Diskussion zu Beginn des folgenden Kapitels wird deutlich machen, daß wir parallel mit der Erweiterung des Typsystems auch den λ-Kalkül ausdehnen müssen.

Die Entwicklung einer formalen Theorie, mit der wir alle Aspekte der Programmierung formalisieren können, ist also in einem gewissen Sinne ein inkrementeller Prozeß. Der ungetypte λ-Kalkül war konzeptionell der einfachste Programmierkalkül, aber zu mächtig, um ein automatisiertes Schließen zu ermöglichen. Die einfache Typentheorie war wiederum der einfachste Weg, den λ-Kalkül auf syntaktischem Wege einzugrenzen, aber ihre Restriktionen waren zu stark, um noch genügend Ausdruckskraft übrig zu lassen. Die Lösung liegt, wie immer, in der Mitte. Da hierfür aber genauere Abgrenzungen nötig sind, wird der integrierte Kalkül zum Schließen über Programme und ihre Eigenschaften insgesamt komplexer werden müssen als die bisher besprochenen Einzelkalküle.

2.6 Ergänzende Literatur

Gute Einführungen in elementare Logik für Informatiker findet man in Lehrbüchern wie [Boyer & Moore, 1979, Gallier, 1986, Manna & Waldinger, 1985, Turner, 1984]. Leser, die an stärker mathematischen Zugängen zu logischen Kalkülen interessiert sind, sollten Bücher wie [Richter, 1978, Schütte, 1977, Takeuti, 1975] konsultieren, in denen viele Aspekte gründlich ausgearbeitet sind. Lesenswert sind auch die Einführungskapitel von [Andrews, 1986, Bibel, 1987, Girard *et.al.*, 1989, Lakatos, 1976, Paulson, 1987], in denen viele Beispiele zu finden sind. Das Buch von Prawitz [Prawitz, 1965] kann als Standardreferenz für natürliche Deduktion angesehen werden. In [Dummett, 1977] findet man eine gute Beschreibung von Philosophie, Semantik und Inferenzsystemen der intuitionistischen Logik. Lesenswert sind auch [Curry, 1970, Heyting, 1971, van Dalen, 1986, Nerode, 1988].

Eine gute Einführung in den λ-Kalkül wurde von Hindley und Seldin [Hindley & Seldin, 1986] und von Huet [Huet, 1986] geschrieben. Das Buch von Barendregt [Barendregt, 1981] dagegen ist sehr ausführlich und detailliert. Wertvolle Details findet man auch im Buch von Stenlund [Stenlund, 1972].

Einführungen in die einfache Typentheorie findet man meist im Zusammenhang mit der Abhandlung komplexerer Theorien wie zum Beispiel in [Martin-Löf, 1984, Constable *et.al.*, 1986, Nordström *et.al.*, 1990, Backhouse *et.al.*, 1988a, Backhouse *et.al.*, 1988b]. Viele der hier vorgestellten Beweise findet man auch in den Büchern von Stenlund [Stenlund, 1972] und Girard [Girard *et.al.*, 1989].