Kapitel 3

Die Intuitionistische Typentheorie

Im vorhergehenden Kapitel haben wir die verschiedenen Arten des logischen Schließens besprochen, die beim Beweis mathematischer Aussagen und bei der Entwicklung korrekter Programme eine Rolle spielen. Wir haben die Prädikatenlogik, den λ-Kalkül und die einfache Typentheorie separat voneinander untersucht und festgestellt, daß jeder dieser drei Kalküle für jeweils ein Teilgebiet – logische Struktur, Werte von Ausdrücken und Eigenschaften (Typen) von Programmen – gut geeignet ist. Um nun *alle* Aspekte des Schließens über Programmierung innerhalb eines einzigen universellen Kalküls behandeln zu können, ist es nötig, diese Kalküle zu vereinigen und darüber hinaus das etwas zu schwache Typsystem der einfachen Typentheorie weiter auszubauen.

Möglichkeiten zur Integration der Logik und des λ-Kalküls in die Typentheorie haben wir im Abschnitt 2.4 bereits andiskutiert. Aufgrund der Curry-Howard Isomorphie läßt sich die Logik durch Typkonstrukte simulieren während der λ-Kalkül in einer Erweiterung des Typkonzepts zu einer typisierten Gleichheit aufgeht. Eine ausdrucksstarke Typentheorie ist somit in der Lage, alle drei bisher vorgestellten Konzepte zu subsumieren.

In der Literatur gibt es eine ganze Reihe von Ansätzen, eine solche Theorie zu entwickeln,[1] von denen bekannt ist, daß sie prinzipiell in der Lage sind, alle Aspekte von Mathematik und Programmierung zu formalisieren. Einen optimalen Kalkül gibt es hierbei jedoch nicht, da die Komplexität der Thematik es leider nicht erlaubt, einen Kalkül anzugeben, der zugleich einfach, elegant und leicht zu verwenden ist. So muß man bei der Entscheidung für die Vorteile eines bestimmten Kalkül immer gewisse Nachteile in Kauf nehmen, welche andere Kalküle nicht besitzen. In diesem Kapitel werden wir die *intuitionistische Typentheorie* des NuPRL Systems [Constable *et.al.*, 1986] vorstellen, die auf entsprechende Vorarbeiten von Per Martin-Löf [Martin-Löf, 1982, Martin-Löf, 1984] zurückgeht und auf eine Verarbeitung mit einem interaktiven Beweissystem angepaßt ist. Aus praktischer Hinsicht erscheint diese Theorie derzeit die geeigneteste zu sein, auch wenn man an dem zugehörigen System noch beliebig viel verbessern kann.

Da die intuitionistische Typentheorie gegenüber der einfachen Typentheorie eine erhebliche Erweiterung darstellt, werden wir zunächst in Abschnitt 3.1 die erforderlichen und wünschenswerten Aspekte diskutieren, die eine Theorie erfüllen sollte, um für das logische Schließen über alle relevanten Aspekte von Mathematik und Programmierung verwendbar zu sein. Aufgrund der Vielfalt der zu integrierenden (Typ-)konzepte werden wir im Anschluß daran (Abschnitt 3.2) eine Systematik für einen einheitlichen Aufbau der Theorie entwickeln und am Beispiel der grundlegendensten Typkonzepte illustrieren. Zugunsten einer in sich geschlossenen und vollständigen Präsentation werden wir dabei die gesamte Begriffswelt (Terme, Berechnung, Semantik, Beweise etc.) komplett neu definieren.[2] Wir müssen dabei etwas tiefer gehen als zuvor, um den gemeinsamen Nenner der bisherigen Kalküle uniform beschreiben zu können.

[1] Die wichtigsten Vertreter findet man beschrieben in
[Girard, 1971, Bruijn, 1980, Martin-Löf, 1982, Martin-Löf, 1984, Constable *et.al.*, 1986, Girard, 1986, Girard *et.al.*, 1989, Coquand & Huet, 1988, Paulin-Mohring, 1989, Constable & Howe, 1990a, Nordström *et.al.*, 1990]

[2] Die Konzepte des vorhergehenden Kapitels sind Spezialfälle davon, für die innerhalb des NuPRL Systems einige Details unterdrückt werden mußten.

Aufbauend auf dieser Systematik werden wir dann die Typentheorie von NuPRL in drei Phasen vorstellen. Zunächst betten wir über die Curry-Howard Isomorphie die Prädikatenlogik in die Typentheorie ein (Abschnitt 3.3) und zeigen, daß sich viele wichtige Eigenschaften der Prädikatenlogik unmittelbar aus grundlegenden Eigenschaften typisierbarer Terme ergeben. In Abschnitt 3.4 ergänzen wir die bis dahin besprochenen Typkonstrukte um solche, die für realistische Programme benötigt werden, und besprechen, wie die Entwicklung von korrekten Programmen aus einer gegebenen Spezifikation mit dem Konzept der Beweisführung zusammenhängt. Abschnitt 3.5 widmet sich der Frage, wie die Rekursion, die durch die Einführung des Typkonzeptes zunächst aus der Theorie verbannt wurde, unter Erhaltung der guten Eigenschaften wieder in die Typentheorie integriert werden kann. Im Abschnitt 3.6 stellen wir schließlich einige Ergänzungen des Inferenzsystems vor, die aus theoretischer Sicht zwar nicht erforderlich sind, das praktische Arbeiten mit dem Kalkül jedoch deutlich vereinfachen.

3.1 Anforderungen an einen universell verwendbaren Kalkül

Die Typentheorie erhebt für sich den Anspruch, alle in der Praxis relevanten Aspekte von Mathematik und Programmierung formalisieren zu können. Damit wird ihr die gleiche grundlegende Rolle für konstruktive formale Systeme beigemessen wie der Mengentheorie für die abstrakte Mathematik. Dies bedeutet insbesondere, daß jeder formale Ausdruck eine feste Bedeutung hat, die keiner weiteren Interpretation durch eine andere Theorie bedarf.

Wir wollen nun die Anforderungen diskutieren, die man an einen universell verwendbaren Kalkül stellen sollte, damit dieser seinem Anspruch einigermaßen gerecht werden kann. Diese Anforderungen betreffen die Semantik einer derart grundlegenden Theorie, ihre Ausdruckskraft sowie die Eigenschaften von Berechnungsmechanismen und Beweiskalkül.

3.1.1 Typentheorie als konstruktive mathematische Grundlagentheorie

Da der Typentheorie eine ähnlich fundamentale Rolle beigemessen werden soll wie der Mengentheorie, muß sie eine universelle Sprache bereitstellen, in der alle anderen Konzepte formuliert werden können. Die Bedeutung und Eigenschaften dieser Konzepte müssen sich dann sich ohne Hinzunahme weiterer Informationen aus den Gesetzen der Typentheorie ableiten lassen. Ihre eigenen Gesetze dagegen können nicht durch eine andere Theorie erklärt werden sondern müssen sich aus einem intuitiven Verständnis der verwendeten Begriffe ergeben. Natürlich müssen diese Gesetze konsistent sein, also einander nicht widersprechen, aber eine tiefergehende Abstützung kann es nicht geben.

Diese Sichtweise auf die Typentheorie erscheint auf den ersten Blick sehr ungewöhnlich zu sein, da man üblicherweise allen mathematischen Theorien eine Semantik zuordnen möchte, die auf mengentheoretischen Konzepten abgestützt ist. Die Mengentheorie selbst jedoch wird nicht weiter hinterfragt und besitzt genau die oben genannten Grundeigenschaften. Als eine grundlegende Theorie *kann* sie nur noch auf der Intuition abgestützt werden.[3] Sie ist der Ausgangspunkt für einen systematischen Aufbau der Mathematik und das einzige, was man fragen muß, ist, ob dieser Ausgangspunkt tatsächlich mächtig genug ist, alle mathematischen Konzepte zu beschreiben.

Die *intuitionistische Typentheorie* nimmt für sich nun genau die gleiche Rolle in Anspruch und man mag sich fragen, wozu man eine neue und kompliziertere Theorie benötigt, wenn doch die Mengentheorie mit dem Konzept der Mengenbildung und die Element-Beziehung “$x \in M$” bereits auskommt. Der Grund hierfür ist – wie wir es schon bei der Prädikatenlogik diskutiert hatten – eine unterschiedliche Sichtweise auf die Mathematik. Während die (klassische) Mengentheorie hochgradig unkonstruktiv ist und ohne die Hinzunahme

[3] Ansonsten müsste es ja eine andere Theorie geben, die noch fundamentaler ist, und wir stünden bei dieser Theorie wieder vor derselben Frage. An irgendeinem Punkt muß man nun einmal anfangen.

semantisch schwer zu erklärender Berechenbarkeitsmodelle nicht in der Lage ist, den Begriff der Konstruktion bzw. des Algorithmus zu erfassen, versteht sich die Typentheorie als eine *konstruktive Mengentheorie*. Ausgehend von dem Gedanken, daß Konstruktionen den Kern eines mathematischen Gedankens bilden, muß die Konstruktion natürlich auch den Kernbestandteil einer Formalisierung der mathematischen Grundlagen bilden, was in der klassischen Mengentheorie nicht der Fall ist. Dieses Konzept herunterzubrechen auf die Ebene der Mengenbildung würde es seiner fundamentalen Rolle berauben.[4] So gibt es in der Typentheorie zwar die gleichen Grundkonzepte wie in der Mengentheorie, also Mengen (Typen) und Elemente, aber bei der Erklärung, wie Mengen zu bilden sind, welche Elemente zu welchen Mengen gehören und was eine Menge überhaupt ausmacht, wird die Konstruktion und ihre formale Beschreibung eine zentrale Rolle spielen.

Wie jeder formale Kalkül besteht die Typentheorie aus einer formalen Sprache, deren *Syntax* und *Semantik* wir zu erklären haben, und einer Menge von *Inferenzregeln*, welche die Semantik der formalen Sprache wiederspiegeln sollen. Wegen ihres konstruktiven Aspekts müssen wir davon ausgehen, daß die formale Sprache der Typentheorie erheblich umfangreicher sein wird als die der Mengentheorie, zumal wir elementare Konstruktionen nicht weiter zerlegen werden. Zwar wäre es technisch durchaus möglich, Konzepte wie Zahlen weiter herunterzubrechen – z.B. durch eine Simulation durch Church-Numerals – aber dies wäre ausgesprochen unnatürlich, da Zahlen nun einmal ein elementares Grundkonzept sind und nicht umsonst "natürliche" Zahlen genannt werden. Andererseits sollte eine universell verwendbare Theorie auch einen *einheitlichen Aufbau* haben. Um dies sicherzustellen, benötigt die Theorie zusätzlich eine erkennbare *Systematik*, nach der formale Ausdrücke aufgebaut sind, wie ihre Semantik – also ihr Wert und der Zusammenhang zu anderen Ausdrücken – bestimmt werden kann und wie das Inferenzsystem aufzubauen ist. Diese Systematik werden wir in Abschnitt 3.2 entwickeln.

Die Entwicklung einer Grundlagentheorie, welche die gesamte intuitive – das steckt ja hinter dem Wort "intuitionistisch" – Mathematik und Programmierung beschreiben kann, ist eine komplizierte Angelegenheit. Der λ-Kalkül scheint mit seiner Turing-Mächtigkeit zu weit zu gehen, da er auch contra-intuitive Konstruktionen wie eine beliebige Selbstanwendbarkeit zuläßt. Eine Grundlagentheorie muß beschreiben können, was *sinnvoll* ist, und muß sich nicht auf alles einlassen, was *machbar* ist. Es ist natürlich die Frage, ob eine solche Theorie überhaupt formalisierbar ist. Die Erfahrungen zeigen, daß man mit einer einfache Formalisierung entweder weit über das Ziel hinausschießt oder wesentliche Aspekte nicht beschreiben kann. Aus diesem Grunde muß die Theorie *offen* (*open-ended*) gestaltet werden, also spätere Erweiterungen durch Hinzunahme neuer Konstrukte zulassen, die sich nicht simulieren lassen, aber essentielle mathematische Denkweisen wiederspiegeln.[5]

3.1.2 Ausdruckskraft

Die Anforderungen an die Ausdruckskraft der Typentheorie ergeben sich unmittelbar aus dem Anspruch, als grundlegende Theorie für die Formalisierung von Mathematik und Programmierung verwendbar zu sein. Dies verlangt, daß alle elementaren Konzepte von Mathematik und Programmiersprachen ein natürliches Gegenstück innerhalb der Typentheorie benötigen – entweder als fest vordefinierten Bestandteil der Theorie oder als leicht durchzuführende definitorische Erweiterung.

Natürlich wäre es wünschenswert, wirklich alle praktisch relevanten Konzepte direkt in die Theorie einzubetten. Dies aber ist unrealistisch, da hierdurch die Theorie extrem umfangreich würde und der Nachweis wichtiger Eigenschaften somit kaum noch zu führen wäre. So ist es sinnvoll, sich darauf zu beschränken, was für die Beschreibung der Struktur von Mengen grundlegend ist bzw. was in Programmiersprachen als zentraler (vorzudefinierender) Bestandteil anzusehen ist. Da die Theorie durch Definitionen jederzeit konservativ erweitert werden kann, ist dies keine essentielle Einschränkung.

[4] Der Wunsch, die gesamte Mathematik durch nur zwei oder drei Konstrukte zu beschreiben, ist zwar verständlich, aber es erhebt sich die Frage, ob das mathematische Denken im Endeffekt so einfach ist, daß es sich durch so wenige Konstrukte ausdrücken läßt. Dies zu glauben oder abzulehnen ist wiederum eine Frage der Weltanschauung. Eine absolute Wahrheit kann es hier nicht geben.

[5] Eine solche Denkweise ist zum Beispiel die *Reflektion*, also die Fähigkeit, über sich selbst etwas auszusagen, oder die *Analogie*.

Zu den grundlegenden Konstrukten gehören sicherlich alle im vorhergehenden Kapitel besprochenen Konzepte, also die Bestandteile der Prädikatenlogik, des λ-Kalküls und der einfachen Typentheorie. Dabei bilden Funktionsdefinition und -anwendung des λ-Kalküls und der Konstruktor $\rightarrow$ der einfachen Typentheorie die Kernbestandteile des Datentyps "Funktionenraum". Da wir uns bereits darauf festgelegt haben, die Logik dadurch zu integrieren, daß wir eine Formel mit dem Datentyp all ihrer Beweise (aufgeschrieben als Terme) identifizieren, müssen wir logische Operatoren durch entsprechende Typkonstruktoren simulieren und entsprechend durch formale Definitionen repräsentieren. Dabei haben wir bereits festgestellt, daß die Implikation sich genauso verhält wie der Operator $\rightarrow$ und angedeutet, daß die Konjunktion mit der Produktbildung verwandt ist. Weitere Zusammenhänge zu den im folgenden als grundlegend vorgestellten Typkonzepten werden wir in Abschnitt 3.3 ausführlich diskutieren. Wir können uns daher darauf beschränken, grundlegende Typkonstrukte zusammenzustellen – also Operatoren zur Konstruktion von Mengen, die zugehörigen kanonischen Elemente und die zulässigen Operationen auf den Elementen. Diese sind

- Der *Funktionen*raum $S{\rightarrow}T$ zweier Mengen S und T mit kanonischen Elementen der Form $\lambda x.t$ und Applikation $f\,t$ als zulässiger Operation.

- Der *Produkt*raum $S{\times}T$ zweier Mengen S und T mit kanonischen Elementen der Form $\langle s,t\rangle$ und den Projektionen bzw. allgemeiner dem spread-Operator $\texttt{let}\ \langle x,y\rangle = pair\ \texttt{in}\ u$ als zulässiger Operation.

  Die in der Programmierung benutzten Datentypen *Feld* (array) und *Verbund* (record) sind durch mehrfache Produktbildung leicht zu simulieren, wobei für Felder auch der Funktionenraum $\texttt{[1..n]}{\rightarrow}T$ benutzt werden kann.

- Die *Summe* (disjunkte Vereinigung) $S{+}T$ zweier Mengen S und T. Diese unterscheidet sich von der aus der Mengentheorie bekannten Vereinigung der Mengen S und T dadurch, daß man den Elementen von $S{+}T$ ansehen kann, ob sie nun aus S oder aus T stammen.[6] Kanonische Elemente bestehen also aus den Elementen von S und denen von T zusammen mit einer Kennzeichnung des Ursprungs, wofür sich als Notation $\texttt{inl}(s)$ bzw. $\texttt{inr}(t)$ (linke bzw. rechte Injektion) eingebürgert hat. Die einzig sinnvolle Operation auf diesem Typ besteht darin, eine Eingabe e zu analysieren und ihren Ursprung sowie das in der Injektion eingekapselte Element zu bestimmen. Abhängig von diesem Resultat können dann verschiedene Terme zurückgegeben werden. Als Notation hierfür kann man zum Beispiel die folgende, an Programmiersprachen angelehnte Bezeichnungsweise $\texttt{case}\ e\ \texttt{of}\ \texttt{inl}(x) \mapsto u\ \mid\ \texttt{inr}(y) \mapsto v$ verwenden.

  Der Datentyp $\mathbb{B}$ kann als ein Spezialfall hiervon angesehen werden, nämlich als Summe von Mengen, auf deren Elemente es nicht ankommt. **T** entspricht dann der linken und **F** der rechten Injektion. Das Conditional $\texttt{if}\ b\ \texttt{then}\ s\ \texttt{else}\ t$ analysiert den Ursprung eines booleschen Wertes b (aber nicht mehr das eingekapselte Element) und liefert dementsprechend einen der beiden Terme s und t als Ergebnis.

- *Zahlen* – zumindest natürliche ($\mathbb{N}$) oder ganze Zahlen ($\mathbb{Z}$) – bilden die Grundlage jeglicher Programmierung und müssen in der Theorie explizit enthalten sein. Kanonische Elemente sind `0,1,-1,2,-2,...` während die zulässigen Operationen aus den gängigen Funktionen `+,-,*,/` etc. und der Möglichkeit einer induktiven Verarbeitung von Zahlen bestehen müssen. Eine Simulation von Zahlen durch andere Konstruktionen ist aus praktischen Gründen abzulehnen.

- Für die Bildung endlicher, aber beliebig großer Strukturen sollte der Raum $S\ \texttt{list}$ der *Listen* (Folgen) über einer gegebenen Menge S und seine Operatoren (vergleiche Abschnitt 2.3.3, Seite 56) ebenfalls ein Grundbestandteil der Theorie sein. Auch hier wäre eine Simulation möglich, aber unpraktisch.

- Die Bildung von *Teilmengen* der Form $\{x{:}T \mid P(x)\}$ ist erforderlich, wenn ein Typ als Spezialfall eines anderen repräsentiert werden soll, wie zum Beispiel die natürlichen Zahlen als Spezialfall der ganzen Zahlen. An kanonischen Elementen und zulässigen Operatoren sollte sich hierdurch nichts ändern (die hierdurch entstehenden Schwierigkeiten diskutieren wir in Abschnitt 3.4.4).

[6] Eine beliebige Vereinigung zweier Mengen wie in der Mengentheorie ist hochgradig nichtkonstruktiv, da die beiden Mengen einfach zusammengeworfen werden. Es besteht somit keine Möglichkeit, die entstandene Menge weiter zu analysieren.

- Ebenso mag es notwendig sein, die Gleichheit innerhalb eines Typs durch Bildung von Restklassen umzudefinieren. Mit derartigen *Quotiententypen* (siehe Abschnitt 3.4.5) kann man zum Beispiel rationale Zahlen auf der Basis von Paaren ganzer Zahlen definieren.
- Über die Notwendigkeit, Textketten (strings) zusätzlich zu Listen in einen Kalkül mit aufzunehmen, mag man sich streiten. Für praktische Programme sind sie natürlich wichtig, aber sie haben wenig Einfluß auf das logische Schließen.
- Die Notwendigkeit *rekursiver Datentypen* und *partiell-rekursiver Funktionen* für die Typentheorie werden wir in Abschnitt 3.5 begründen.

Diese Liste ist bereits sehr umfangreich und man wird kaum Anwendungen finden, die sich mit den obengenannten Konstrukten nicht ausdrücken lassen. Dennoch reichen sie noch nicht ganz aus, da die bisherige Konzeption des Funktion- und Produktraumes zu einfach ist, um alle Aspekte von Funktionen bzw. Tupeln zu charakterisieren. Wir wollen hierzu ein paar Beispiele betrachten.

Beispiel 3.1.1

1. Innerhalb des λ-Kalküls beschreibt der Term $f \equiv \lambda\texttt{b}.\,\texttt{if b then}\ \lambda\texttt{x}.\lambda\texttt{y.x else}\ \lambda\texttt{x.x}$ eine durchaus sinnvolle[7] – wenn auch ungewöhnliche – Funktion, die bei Eingabe eines booleschen Wertes b entweder eine Projektionsfunktion oder die Identitätsfunktion als Ergebnis liefert.

   Versucht man, diese Funktion zu typisieren, so stellt man fest, daß der *Typ* des Ergebnisses vom *Wert* der Eingabe abhängt. Bei Eingabe von **T** hat der Ergebnistyp die Struktur S→T→S, während er die Struktur S→S hat, wenn **F** eingegeben wird.

   Eine einfache Typisierung der Art $f \in \mathbb{B}\to\texttt{T}$ ist also nicht möglich. Was uns bisher noch fehlt, ist eine Möglichkeit, diese Abhängigkeit zwischen dem Eingabewert $\texttt{b}:\mathbb{B}$ und dem Ausgabetyp $\texttt{T}[\texttt{b}]$[8] auszudrücken. Wir müssen also den Funktionenraumkonstruktor → so erweitern, daß derartige Abhängigkeiten erfaßt werden können. Man spricht in diesem Fall von einem *abhängigen Funktionenraum* (*dependent function type*), und verwendet als allgemeine Notation $\underline{x{:}S\to T}$ anstelle des einfacheren $S\to T$, um auszudrücken daß der Typ T vom Wert $x\in S$ abhängen kann.

2. In der Programmiersprache Pascal gibt es den sogenannten *variant record* – ein Konstrukt, daß es erlaubt, verschieden*artige* Tupel in einem Datentyp zusammenzufassen. Eine sinnvolle Anwendung ist z.B. die Beschreibung geometrischer Objekte, für die man – je nachdem ob es sich um Rechtecke, Kreise oder Dreiecke handelt – unterschiedlich viele Angaben benötigt.

```
RECORD
  CASE kind:(RECT,TRI,CIRC) of
     RECT: (länge, breite:real)
     TRI : (a,b,c:real)
     CIRC: (radius:real)
END
```

   Die Elemente dieses Datentyps bestehen also je nach Wert der ersten Komponente kind aus drei, vier oder zwei Komponenten. Der Datentyp ist also ein Produktraum der Gestalt $S{\times}T$, wobei der Typ T je nach Wert des Elements aus S entweder $\mathbb{R}\times\mathbb{R}$, $\mathbb{R}\times\mathbb{R}\times\mathbb{R}$ oder einfach nur $\mathbb{R}$ ist.

   Diese Abhängigkeit des Typs der zweiten Komponente vom Wert der ersten ist in dem üblichen Verständnis des Produktraumkonstruktors × nicht enthalten. Daher muß auch dieser entsprechend erweitert werden zu einem *abhängigen Produktraum* (*dependent product type*), den man mit $\underline{x{:}S{\times}T}$ bezeichnet.

[7] Funktionen dieser Art treten zum Beispiel auf, wenn man Lösungsstrategien für Suchprobleme in der KI programmieren möchte. Das Ergebnis eines Testes legt fest, wieviele Werte man noch abfragen muß, bevor man eine Lösung bestimmen kann.

[8] In Anlehnung an die in Definition 2.3.5 auf Seite 49 vereinbarte Notation kennzeichnet T[b] das Vorkommen der freien Variablen b im Typausdruck T.

3. Auch in der Mathematik kommen derartige Abhängigkeiten öfters vor. So werden zum Beispiel endliche Automaten als 5-Tupel (Q,Σ,q_0,δ,F) definiert, wobei Q und Σ endliche Mengen sind, $q_0\in Q$, $F\subseteq Q$ und $\delta{:}\ (Q{\times}\Sigma)\to Q$. Offensichtlich hängen also die Typen der dritten, vierten und fünften Komponente ab von den Werten der ersten beiden. Die Menge aller endlichen Automaten läßt sich somit nur durch die Verwendung abhängiger Produkträume präzise beschreiben.

4. Ein weiterer Grund, das Typsystem um abhängige Typkonstruktoren zu erweitern, ist die Tatsache, daß wir die Logik innerhalb der Typentheorie simulieren möchten. Aufgrund der Curry-Howard Isomorphie (siehe Abschnitt 2.4.7) können wir Formeln mit Typen identifizieren, deren Elemente genau den Beweisen für diese Formeln entsprechen. Wendet man diesen Gedanken nun auf die logischen Quantoren an, so ergeben sich folgende Zusammenhänge:

   - Um eine allquantifizierte Formel der Gestalt $\forall x{:}T.\,P(x)$ zu beweisen, müssen wir zeigen, wie wir für ein beliebiges Element $x\in T$ einen Beweis für $P(x)$ konstruieren. Genau besehen konstruieren wir also eine Funktion, die bei Eingabe eines Wertes $x\in T$ einen Beweis für $P(x)$ bestimmt. Der Ergebnis*typ* dieser Funktion – der mit $P(x)$ identifizierte Typ – hängt also ab vom *Wert* der Eingabe $x\in T$.
   - Um eine existentiell quantifizierte Formel der Gestalt $\exists x{:}T.\,P(x)$ zu beweisen, müssen wir ein Element $x\in T$ angeben und zeigen, daß für dieses Element die Eigenschaft $P(x)$ gilt. Genau besehen konstruieren wir also ein Tupel $\langle x,p\rangle$, wobei $x\in T$ und p ein Beweis für $P(x)$ ist. Der Typ der zweiten Komponente, also $P(x)$, hängt also ab vom *Wert* der ersten Komponente $x\in T$.

   Somit sind zur Simulation der Quantoren innerhalb der Typentheorie abhängige Typkonstruktoren unumgänglich.

Insgesamt sind also *abhängige Datentypen* ein notwendiger und sinnvoller Bestandteil einer ausdrucksstarken Theorie, die für sich in Anspruch nimmt, alle relevanten Aspekte von Mathematik und Programmierung formalisieren zu können.[9] Die Hinzunahme abhängiger Datentypen in die Typentheorie bringt jedoch auch einige Probleme mit sich. Während es in der einfachen Typentheorie möglich war, Typausdrücke und Objektausdrücke sauber voneinander zu trennen, verlangt die Abhängigkeit eines Typs T von einem gegebenen Wert $s\in S$, daß wir Objektausdrücke innerhalb von Typausdrücken zulassen müssen und eine *syntaktische* Trennung der beiden Ausdrucksarten kaum möglich sein wird. Die Identifizierung korrekter Objekt- bzw. Typausdrücke wird somit zu einem Problem der Semantik bzw. des Beweiskalküls.

3.1.3 Eigenschaften von Berechnungsmechanismen und Beweiskalkül

Bisher haben wir im wesentlichen über die Verwendbarkeit der Typentheorie als *mathematische* Grundlagentheorie gesprochen. Für ihre Anwendung als praktisch nutzbarer Beweiskalkül zum rechnergestützten Schließen über Programme müssen wir natürlich noch weitere Anforderungen stellen. Dazu gehören insbesondere die algorithmischen Eigenschaften des Reduktionsmechanismus, mit dem der Wert eines Ausdrucks bestimmt werden kann. Diese werden natürlich beeinflußt von der Ausdruckskraft des Kalküls: je mehr wir beschreiben können wollen, um so größere Abstriche müssen wir bei den Eigenschaften machen.

- Es ist offensichtlich, daß wir innerhalb des Schlußfolgerungskalküls *nur terminierenden Berechnungen* gebrauchen können. Andernfalls würde das Halteproblem jegliche Form einer automatischen Unterstützung

[9] Aus mathematischer Sicht kann man einen abhängigen Produktraum $x{:}S{\times}T$ auch als disjunkte Vereinigung aller Räume $T[x]$ mit $x\in S$ – bezeichnet mit $\sum_{x\in S}T$ – auffassen. Diese Notation, die von vielen Autoren benutzt wird, erfaßt aber nicht, daß es sich bei den Elementen des Datentyps nach wie vor um Tupel der Form $\langle s,t\rangle$ mit $s\in S$ und $t\in T[s]$ handelt, und erscheint daher weniger natürlich. Dasselbe gilt für die abhängigen Funktionenräume, die man entsprechend der Sicht von Funktionen als Menge von Paaren $\langle s,f(s)\rangle$ als unbeschränktes Produkt aller Räume $T[x]$ mit $x\in S$ – bezeichnet mit $\prod_{x\in S}T$ – auffassen kann. Diese Notation berücksichtigt nicht, daß es sich bei den Elementen des Datentyps nach wie vor um λ-Terme handelt.

ausschließen. Wir werden in Abschnitt 3.3 sehen, daß sich starke Normalisierbarkeit nicht mit extensionaler Gleichheit und dem Konzept der logischen Falschheit verträgt. Beide Konzepte sind aber von essentieller Bedeutung für das formale Schließen. Starke Normalisierbarkeit ist allerdings auch nicht unbedingt erforderlich, da es – wie bei allen Programmiersprachen – ausreicht, mit einem bestimmten Verfahren zu einem Ergebnis zu kommen.

- Auch wenn wir für automatische Berechnungen eine feste Reduktionsstrategie fixieren, werden wir auf *Konfluenz* nicht verzichten können. Der Grund hierfür ist, daß wir beim Führen formaler Beweise innerhalb des Kalküls mehr Freiheitsgrade lassen müssen als für die Reduktion. Es darf aber nicht angehen, daß wir durch verschiedene Beweise zu verschiedenen Ergebnissen kommen.
- Automatische *Typisierbarkeit* wäre wünschenswert, ist aber nicht zu erwarten. Durch das Vorkommen abhängiger Datentypen bzw. logischer Quantoren ist Typisierung in etwa genauso schwer wie automatisches Beweisen in der Prädikatenlogik. Man muß die Typisierung daher im Beweiskalkül unterbringen.

Eine letzte wünschenswerte Eigenschaft ergibt sich aus der Tatsache, daß die Prädikatenlogik innerhalb der Typentheorie simuliert wird. In seiner bisherigen Konzeption läßt der Kalkül nur die *Überprüfung* der Typkorrektheit zu, was der Kontrolle, ob ein vorgegebener Beweis tatsächlich ein Beweis für eine gegebene Formel ist, entspricht (*proof-checking*). Wir wollen Beweise jedoch nicht nur überprüfen sondern – wie wir es aus Abschnitt 2.2 gewohnt sind – mit dem Kalkül *entwickeln*. Der Kalkül muß daher so ausgelegt werden, daß er eine Beweis*konstruktion* und damit auch die Konstruktion von Programmen aus Spezifikationen zuläßt.

3.2 Systematik für einen einheitlichen Aufbau

Wegen der vorgesehenen Rolle als fundamentale Theorie für Mathematik und Programmierung ist die Intuitionistische Typentheorie als eine *konstruktive semantische Theorie* ausgelegt, also als eine Theorie, die nicht nur aus Syntax und Inferenzregeln besteht, sondern darüber hinaus auch noch die Semantik ihrer Terme beschreiben muß. Letztere wird erklärt über das Konzept der Auswertung (Reduktion) von Termen und durch eine methodische Anlehnung an die in Abschnitt 2.4.5.2 vorgestellte *TAIT computability method* (d.h. Abstützung auf 'berechenbare' Elemente) wird sichergestellt, daß diese Reduktionen immer terminieren.

In diesem Abschnitt werden wir die allgemeinen Prinzipien vorstellen, nach denen diese Theorie aufgebaut ist, und am Beispiel dreier Typkonstrukte – dem Funktionenraum, dem Produktraum und der disjunkten Vereinigung – erläutern. Diese Prinzipien regeln den allgemeinen Aufbau von Syntax, Semantik und Inferenzsystem und gehen im wesentlichen nach folgenden Gesichtspunkten vor.

Syntax: Um die formale Sprache festzulegen, sind zulässige Terme (Ausdrücke), bindende Vorkommen von Variablen in diesen Termen und die Ergebnisse von Substitutionen zu definieren. Eine syntaktische Unterscheidung von Objekt- und Typausdrücken wird nicht vorgenommen. Dies wird ausschließlich auf semantischer Ebene geregelt.

Semantik: Die Terme werden unterteilt in *kanonische* und *nichtkanonische* Terme. Der *Wert* eines kanonischen Terms ist der Term selbst, wobei man davon ausgeht, daß ein kanonischer Term immer mit einer intuitiven Bedeutung verbunden ist. Der Wert eines (geschlossenen) nichtkanonischen Ausdrucks wird durch *Reduktion* auf einen kanonischen Term bestimmt.[10]

Die grundsätzlichen Eigenschaften von Termen und die Beziehungen zwischen Termen werden durch *Urteile* (judgments) definiert. Innerhalb der Typentheorie regeln diese Urteile Eigenschaften wie Gleichheit, Typzugehörigkeit und die Tatsache, daß ein Term einen Typ beschreibt. Für kanonische Terme

[10]Hierbei ist anzumerken, daß sich das Konzept "kanonischer Term" oft nur auf die äußere Struktur eines Terms bezieht, der im Inneren durchaus noch reduzierbare Ausdrücke enthalten kann. Damit unterscheidet sich das Konzept der "Normalform" eines Terms von der in Abschnitt 2.3.7 definierten Form, die erst erreicht ist, wenn *alle* reduzierbaren Ausdrücke eliminiert sind.

wird die Semantik dieser Urteile explizit (rekursiv) definiert, für nichtkanonische Terme ergibt sie sich aus der Semantik der gleichwertigen kanonischen Terme.[11] Das Konzept des Urteils wird erweitert auf *hypothetische Urteile* um ein Argumentieren unter bestimmten Annahmen zu ermöglichen.

Inferenzsystem: Urteile und hypothetische Urteile werden syntaktisch durch Sequenzen repräsentiert[12] und die Semantik von Urteilen durch Inferenzregeln ausgedrückt. Dabei ist aber zu berücksichtigen, daß sich dieser Zusammenhang nicht vollständig beschreiben läßt, was insbesondere für die Eigenschaft, ein Typ zu sein, gilt.

Aus diesem Grunde muß die Syntax um einige Ausdrücke (wie Typuniversen und einen Gleichheitstyp) erweitert werden, deren einziger Zweck es ist, die in den Urteilen benannte Zusammenhänge innerhalb der formalen Sprache auszudrücken. Semantisch liefern diese Ausdrücke keinerlei neue Informationen, da ihre Semantik nahezu identisch mit der des zugehörigen Urteils ist.

An einigen Stellen mußten bei der Entwicklung der Typentheorie Entwurfsentscheidungen getroffen werden, die genausogut auch anders hätten ausfallen können. Wir werden diese und die Gründe, die zu dieser Entscheidung geführt haben, an geeigneter Stelle genauer erklären.

In der nun folgenden Beschreibung der Methodik für einen einheitlichen Aufbau der Theorie werden wir uns nicht auf alle Details einlassen, die bei der systematischen Konzeption des NuPRL Systems eine Rolle gespielt haben, sondern nur diejenigen Aspekte betrachten, die für einen eleganten Aufbau der eigentlichen Theorie und ihr Erscheinungsbild auf dem Rechner bedeutend sind.

3.2.1 Syntax formaler Ausdrücke

Bei der Festlegung der Syntax der formalen Sprache, also der zulässigen Ausdrücke, dem freien und gebundenen Vorkommen von Variablen und der Substitution von Variablen durch Terme könnte man einfach die Definitionen aus Abschnitt 2.4.1 übernehmen und entsprechend für die neu hinzugekommenen Konzepte erweitern. Für die Datentypen (abhängiger) Funktionenraum, (abhängiger) Produktraum und disjunkte Vereinigung und ihre zugehörigen Operationen ergäbe sich dann folgende Definition von Ausdrücken:

Es sei $\underline{\mathcal{V}}$ eine unendliche Menge von Variablen. Ausdrücke sind induktiv wie folgt definiert.

1. *Jede Variable $\underline{x}\in\mathcal{V}$ ist ein Ausdruck.*
2. *Ist t ein beliebiger Ausdruck, dann ist $\underline{(t)}$ ein Ausdruck.*
3. *Ist $x\in\mathcal{V}$ eine Variable und t ein beliebiger Ausdruck, dann ist $\underline{\lambda x.t}$ ein Ausdruck.*
4. *Sind t und f beliebige Ausdrücke, dann ist $\underline{f\ t}$ ein Ausdruck.*
5. *Ist $x\in\mathcal{V}$ eine Variable und sind S und T beliebige Ausdrücke, so sind $\underline{x{:}S\to T}$ und $\underline{S\to T}$ Ausdrücke.*
6. *Sind s und t beliebige Ausdrücke, dann ist $\underline{\langle s,t\rangle}$ ein Ausdruck.*
7. *Sind $x,y\in\mathcal{V}$ Variablen sowie e und u beliebige Ausdrücke, dann ist $\underline{\mathsf{let}\ \langle x,y\rangle = e\ \mathsf{in}\ u}$ ein Ausdruck.*
8. *Ist $x\in\mathcal{V}$ eine Variable und sind S und T beliebige Ausdrücke, so sind $\underline{x{:}S\times T}$ und $\underline{S\times T}$ Ausdrücke.*
9. *Sind s und t beliebige Ausdrücke, dann sind $\underline{\mathsf{inl}(s)}$ und $\underline{\mathsf{inr}(t)}$ Ausdrücke.*
10. *Sind $x,y\in\mathcal{V}$ Variablen sowie e, u und v beliebige Ausdrücke, dann ist $\underline{\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(x)\mapsto u\ \mid\ \mathsf{inr}(y)\mapsto v}$ ein Ausdruck.*
11. *Sind S und T beliebige Ausdrücke, dann ist $\underline{S+T}$ ein Ausdruck.*

[11]Hierdurch wird klar, warum Berechnungen terminieren müssen. Ohne diese Eigenschaft wäre die Semantik von Aussagen über nichtkanonische Terme nicht definiert.

[12]Es sei an dieser Stelle angemerkt, daß der Unterschied zwischen einem Urteil und seiner Repräsentation als Sequenz oft genauso verwischt wird wie der Unterschied zwischen einer Zahl und ihrer Dezimalschreibweise. Die Urteile beschreiben das semantische Konzept, während die Sequenzen die textliche Form sind, dies aufzuschreiben. Die Möglichkeit, durch die Einbettung der Sequenz in die formale Sprache wieder über die Semantik der Theorie selbst zu reflektieren ist durchaus gewollt.

Der Nachteil dieser Definition (die wir absichtlich nicht numeriert haben) ist, daß nicht nur eine Fülle von Ausdrücken definiert wird sondern auch völlig unterschiedliche Strukturen von Ausdrücken eingeführt werden, die deswegen jeweils eine unabhängige Zeile in der induktiven Definition verlangen. Dieser Nachteil wird noch deutlicher, wenn man sich die entsprechende Definition freier und gebundener Variablen ansieht.

Es seien $x,y,z \in \mathcal{V}$ Variablen sowie s, t, u, v, e, S und T Ausdrücke. Das freie und gebundene Vorkommen der Variablen x in einem Ausdruck ist induktiv durch die folgenden Bedingungen definiert.

1. *Im Ausdruck x kommt x frei vor. Die Variable y kommt nicht vor, wenn x und y verschieden sind.*
2. *In (t) bleibt jedes freie Vorkommen von x in t frei und jedes gebundene Vorkommen gebunden.*
3. *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.*
4. *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.*
5. *In $x{:}S \rightarrow T$ wird jedes freie Vorkommen von x in T gebunden. Freie Vorkommen von x in S bleiben frei. Gebundene Vorkommen von x in S oder T bleiben gebunden. Jedes freie Vorkommen der Variablen y in T bleibt frei, wenn x und y verschieden sind.*

   In $S \rightarrow T$ bleibt jedes freie Vorkommen von x in S oder T frei und jedes gebundene Vorkommen gebunden.
6. *In $\langle s,t\rangle$ bleibt jedes freie Vorkommen von x in s oder t frei und jedes gebundene Vorkommen gebunden.*
7. *In $\mathsf{let}\ \langle x,y\rangle = e\ \mathsf{in}\ u$ wird jedes freie Vorkommen von x und y in u gebunden. Freie Vorkommen von x oder y in e bleiben frei. Gebundene Vorkommen von x oder y in e oder u bleiben gebunden. Jedes freie Vorkommen der Variablen z in u bleibt frei, wenn z verschieden von x und y ist.*
8. *In $x{:}S \times T$ wird jedes freie Vorkommen von x in T gebunden. Freie Vorkommen von x in S bleiben frei. Gebundene Vorkommen von x in S oder T bleiben gebunden. Jedes freie Vorkommen der Variablen y in T bleibt frei, wenn x und y verschieden sind.*

   In $S \times T$ bleibt jedes freie Vorkommen von x in S oder T frei und jedes gebundene Vorkommen gebunden.
9. *In $\mathsf{inl}(s)$ bzw. $\mathsf{inr}(t)$ bleibt jedes freie Vorkommen von x in s bzw. t frei und jedes gebundene Vorkommen gebunden.*
10. *In $\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u \mid \mathsf{inr}(y) \mapsto v$ wird jedes freie Vorkommen von x in u bzw. von y in v gebunden. Freie Vorkommen von x oder y in e bleiben frei. Gebundene Vorkommen von x oder y in e, u oder v bleiben gebunden. Jedes freie Vorkommen der Variablen z in u bzw. v bleibt frei, wenn z verschieden von x bzw. y ist.*
11. *In $S+T$ bleibt jedes freie Vorkommen von x in S oder T frei und jedes gebundene Vorkommen gebunden.*

Wie man sieht, ufern die Definitionen sehr schnell aus, obwohl wir bisher nur drei der uns interessierenden Typkonstrukte beschrieben haben. Eine Systematik – besonders bei den bindenden Vorkommen von Variablen in Termen – ist ebenfalls nicht zu erkennen, da die Variablen innerhalb eines Termes an nahezu beliebigen Positionen vorkommen dürfen. Würden wir diese Vorgehensweise bei der Einführung der weiteren Typkonzepte beibehalten, so könnte alleine die Syntax der Sprache schnell unübersichtlich werden. Zudem wären wir gezwungen, die Definition der Ausdrücke ständig zu erweitern.

Das Hauptproblem der obigen Vorgehensweise liegt darin, daß die verschiedenen Ausdrücke in ihrer Struktur extrem unheitlich sind, da sie sich an lange vertraute Notationen anlehnen. Ein einheitlicher und systematischer Aufbau kann daher nur erreicht werden, wenn man von der Notation abstrahiert und die Beschreibung von Ausdrücken auf das eigentlich Wesentliche reduziert. Wir wollen hierfür einige Beispiele betrachten.

Beispiel 3.2.1

Die Paarbildung $\langle s,t\rangle$ ist eigentlich nur eine elegantere Notation für die Anwendung eines Tupeloperators auf die Ausdrücke s und t, also eine Abkürzung für einen Ausdruck der Form $\textbf{pair}(s;t)$.

Genauso ist die Applikation $f\ t$ nur eine Abkürzung für einen Ausdruck der Form $\textbf{apply}(f;t)$[13] und dasselbe gilt für den Ausdruck $\textbf{union}(S;T)$, der einen Ausdruck der Form $S+T$ ankürzt.

Eine wesentlich einheitlichere Notation für Ausdrücke ist also die aus Definition 2.2.2 (Seite 24) bekannte Termschreibweise. Zum Zwecke der Systematisierung der Theorie sollte ein Ausdruck (Term) immer die Gestalt $op(t_1;\ldots;t_n)$ haben, wobei op ein Operator – wie pair, apply oder union – ist und die t_i Ausdrücke sind.

Mit dieser Beschreibungsform können jedoch noch nicht alle Ausdrücke erfaßt werden, da sie nicht in der Lage ist, die *Bindung* von Variablen in Teiltermen zu kennzeichnen. Wir müssen daher eine gewisse Erweiterung vornehmen.

Beispiel 3.2.2

Die λ-Abstraktion $\lambda x.t$ beschreibt einen Ausdruck, in dem die Variable x innerhalb des Terms t gebunden werden soll. Eine Ausdruck der Form $\mathsf{lam}(x;t)$ würde diesen Zusammenhang nicht wiederspiegeln können, da hier x und t als unabhängige Teilterme auftreten, die jeweils ihren eigenen Wert haben. In Wirklichkeit wollen wir jedoch kennzeichnen, daß x eine *Variable* und nicht etwa ein Ausdruck ist, die innerhalb von t vorkommen kann und dort gebunden werden soll. x und t bilden also eine Einheit, die man als *gebundenen Term* bezeichnet und zum Beispiel mit $\underline{x.t}$ kennzeichnen kann. Eine angemessene Termschreibweise für $\lambda x.t$ wäre also $\textbf{lam}(x.t)$.

Die einheitliche Beschreibungsform für den abhängigen Funktionenraum $x{:}S \rightarrow T$ ist dementsprechend $\textbf{fun}(S;x.T)$ und die für den abhängigen Produktraum $x{:}S \times T$ ist $\textbf{prod}(S;x.T)$. Die einheitliche Notation macht übrigens auch deutlich, daß die Variable x im Typ T gebunden ist und nicht etwa in S. Die Tatsache, daß die Werte für x aus dem Typ S zu wählen sind, ist ein semantischer Zusammenhang und hat mit einer syntaktischen Codierung nichts zu tun.

Der Term, der sich hinter der Notation $\mathsf{let}\ \langle x,y\rangle = e\ \mathsf{in}\ u$ verbirgt, lautet $\textbf{spread}(e;x,y.u)$. Hier werden zwei Variablen innerhalb des Terms u gebunden, was durch die Notation $\underline{x,y.u}$ gekennzeichnet wird. Die Notation $\textbf{decide}(e;x.u;y.v)$ (für $\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u \mid \mathsf{inr}(y) \mapsto v$) macht ebenfalls deutlicher, welche Variablen in welchen Termen gebunden werden.

Entsprechend der obigen Beispiele müssen wir also die Beschreibungsform für Ausdrücke ausdehnen auf die Gestalt $op(b_1;\ldots;b_n)$, wobei op ein Operator ist und die b_i gebundene Terme sind. Gebundene Terme wiederum haben die Gestalt $x_1,\ldots,x_n.t$, wobei die x_i Variablen sind und t ein Term (Ausdruck) ist.

Die bisherigen Beispiele zeigen, wie wir aus Termen und Operatoren neue Terme aufbauen können. Womit aber fangen wir an? Was sind die Basisterme unserer Theorie? Auch hierzu geben wir ein Beispiel.

Beispiel 3.2.3

In allen bisherigen Definitionen der Syntax formaler Sprachen wurden Variablen $x \in \mathcal{V}$ als atomare Terme bezeichnet. Diese Gleichsetzung ist aber eine grobe Vereinfachung, denn genau besehen sind Terme strukturierte *Objekte*, die einen Wert besitzen können, während Variablen *Platzhalter* sind, deren wesentliches Merkmal ihr Name ist. Die Aussage "jede Variable $x \in \mathcal{V}$ ist ein Term" nimmt implizit eine Konversion vor. Bei einer präzisen Definition müßte man allerdings eine Unterscheidung vornehmen zwischen der Variablen x und dem Term, der nur die Variable x beinhaltet, und die Konversion explizit machen. Wenn wir unser bisheriges Konzept beibehalten und sagen, daß Terme nur durch Anwendung von Operatoren entstehen können, dann muß diese Konversion von Variablen in Terme einem Operator var entsprechen, der auf Variablen angewandt wird. Diese Anwendung des Operators var auf eine Variable ist allerdings etwas anderes als die Anwendung von Operatoren auf (gebundene) Terme, die als Argumente des Operators auftauchen. Deshalb soll diese Anwendungsform auch in ihrer Notation deutlich von der anderen

[13]Diese Abkürzung hat sich aber mittlerweile so sehr eingebürgert, daß sie oft verwechselt wird mit der Anwendung des Operators f auf den Ausdruck t. In Wirklichkeit wird aber nur ein neuer Ausdruck gebildet und die Funktion f wird erst auf t angewandt, wenn der Reduktionsmechanismus gestartet wird. Die Schreibweise $\textbf{apply}(f;t)$ bringt die Natur dieses Ausdrucks also auch viel deutlicher zum Vorschein.

unterschieden werden und wir schreiben var{x:variable}(), um den Term zu präzisieren, der gängigerweise einfach mit x bezeichnet wird. x ist also ein Parameter des Operators var und das Schlüsselwort variable kennzeichnet hierbei, daß es sich bei x um einen Variablennamen handelt.

Diese Präzisierung mag vielleicht etwas spitzfindig erscheinen, aber sie erlaubt eine saubere Trennung verschiedenartiger Konzepte innerhalb einer einheitlichen Definition des Begriffs "Term". Der Aufwand wäre sicherlich nicht gerechtfertigt, wenn Konversionen nur bei Variablen implizit vorgenomen würden. Bei Zahlen und anderen Konstanten, die wir als Terme ansehen wollen, stoßen wir jedoch auf dieselbe Problematik. Gemäß Definition 2.2.2 müssen alle Konstanten als Anwendungen nullstelliger Operatoren angesehen werden. Die Parametrisierung von Operatoren ermöglicht es nun, diese Operatoren unter *einem* Namen zu einer Familie gleichartiger Operatoren zusammenzufassen

Da es noch weitere, hier nicht betrachtete Gründe geben mag, Operatoren zu parametrisieren, dehnen wir die Beschreibungsform für Ausdrücke ein weiteres Mal aus. Insgesamt haben Ausdrücke nun die Gestalt

$$opid\{P_1, \ldots, P_n\}(b_1; \ldots; b_n),$$

wobei *opid* der Name für eine Operatorfamilie ist, die P_i seine Parameter (samt Kennzeichnung, um welche Art Parameter es sich handelt) sind und die b_i gebundene Terme. Gebundene Terme wiederum haben die Gestalt $x_1, \ldots, x_n.t$, wobei die x_i Variablen sind und t ein Term (Ausdruck) ist.

Diese einheitliche Schreibweise, die übrigens auch jede Art zusätzlicher Klammern überflüssig macht, beschreibt die wesentlichen Aspekte von Termen und ermöglicht sehr einfache Definitionen des Termbegiffs, gebundener und freier Variablen und von Substitutionen.

Natürlich wollen wir auf die vertrauten Notationen nicht ganz verzichten, denn dies wäre ja im Sinne einer praktisch nutzbaren Theorie ein ganz gewaltiger Rückschritt. Die einheitliche, abstrakte Notation hilft beim systematischen Aufbau einer Theorie und ihrer Verarbeitung mit dem Rechner. Für den Menschen, der mit den Objekten (Termen) dieser Theorie umgehen muß, ist sie nicht sehr brauchbar. Aus diesem Grunde wollen wir die textliche Darstellung eines Terms – die sogenannte *Display Form* – von seiner abstrakten Darstellung in der einheitlichen Notation – seine *Abstraktionsform* – unterscheiden und beide Formen simultan betrachten. In formalen Definitionen, welche die Theorie als solche erklären, verwenden wir die formal einfachere Abstraktionsform, während wir bei der Charakterisierung und Verarbeitung konkreter Terme soweit wie möglich auf die Display Form zurükgreifen werden. Die Eigenschaften dieser Terme ergeben sich dann größtenteils aus den allgemeinen Definitionen und dem Zusammenhang zwischen einer konkreten Abstraktion und ihrer Display Form.[14] Wir wollen diese Entscheidung als ein erstes wichtiges Prinzip der Theorie fixieren.

Entwurfsprinzip 3.2.4 (Trennung von Abstraktion und textlicher Darstellung)

In der Typentheorie werden die Abstraktionsform eines Terms und seine Darstellungsform getrennt voneinander behandelt aber simultan betrachtet.

Dieses Prinzip gilt gleichermaßen für die vordefinierten Terme der Typentheorie und für konservative Erweiterungen durch einen Anwender der Theorie.[15] Die Display Form beschreibt darüber hinaus auch alle wichtigen Eigenschaften, die für eine eindeutige Lesbarkeit der textlichen Darstellung nötig sind, wie zum Beispiel Prioritäten und Assoziativität. Durch die Trennung von Abstraktion und Darstellungsform, die vom NuPRL System voll unterstützt wird, gewinnen wir maximale Flexibilität bei der formalen Repräsentation von Konstrukten aus Mathematik und Programmierung innerhalb eines computerisierten Beweissystems.

[14] Im NuPRL System wird diese Vorgehensweise dadurch unterstützt, daß man üblicherweise die Display Form eines Termes gezeigt bekommt, die im Prinzip frei wählbar ist, während das System intern die Abstraktionsform verwaltet, die man nur auf Wunsch zu sehen bekommt.

[15] Dies hat auch Auswirkungen auf die Art, wie konservative Erweiterungen innerhalb der Theorie behandelt werden. Während im Abschnitt 2.1.5 eine formale Definition im wesentlichen als textliche Abkürzung verstanden wurde, die innerhalb eines Beweiseditors als Textmacro behandelt werden kann, besteht eine konservative Erweiterung nun aus zwei Komponenten: einer *Abstraktion*, die ein neues Operatorsymbol in die sogenannte Operatorentabelle einträgt, und einer *Display Form*, welche die textliche Darstellung von Termen beschreibt, die mit diesem Operator gebildet werden (siehe Abschnitt 4.1.7). Um also zum Beispiel den Typ IN der natürlichen Zahlen als Teilmenge der ganzen Zahlen zu definieren, definiert man zunächst als Abstraktion:
nat{}() ≡ {n:ℤ|n≥0}
Hierdurch wird ein Operator mit Namen nat definiert und seine Bedeutung erklärt.
Weiterhin wird festgelegt, wie dieser Operator üblicherweise zu schreiben ist.
IN ≡ nat{}()
Im Endeffekt bedeutet eine solche Definition also eine Erweiterung der Tabelle der vordefinierten Operatoren. Der einzige Unterschied zu diesen ist, daß aufgrund der Abstützung auf bereits bekannte Terme die Semantik und die Inferenzregeln des neu definierten Operators nicht neu erklärt und auf Konsistenz überprüft werden müssen. Beides ergibt sich automatisch aus der Semantik und den Regeln der Grundoperatoren und ist konsistent mit dem Rest der Theorie.

variable : Variablennamen (ML Datentyp var)
Zulässige Elemente sind Zeichenketten der Form [a−z A−Z 0−9 _ - %]+

natural : Natürliche Zahlen einschließlich der Null (ML Datentyp int).
Zulässige Elemente sind Zeichenketten der Form 0 + [1 − 9][0 − 9]*.

token : Zeichenketten für Namen (ML Datentyp tok).
Zulässige Elemente bestehen aus allen Symbolen des NuPRL Zeichensatzes mit Ausnahme der Kontrollsymbole.

string : Zeichenketten für Texte (ML Datentyp string).
Zulässige Elemente bestehen aus allen Symbolen des NuPRL Zeichensatzes mit Ausnahme der Kontrollsymbole.

level-expression : Ausdrücke für das Level eines Typuniversums (ML Datentyp level_exp)
Die genaue Syntax wird in Abschnitt 3.2.3.1 bei der Diskussion der Universenhierarchie besprochen.

Die Namen für Parametertypen dürfen durch ihre ersten Buchstaben abgekürzt werden.

Abbildung 3.1: Parametertypen und zugehörige Elemente

Nach all diesen Vorüberlegungen ist eine präzise Definition von Termen leicht zu geben.

Definition 3.2.5 (Terme)

1. *Es sei $\mathcal{I}$ eine Menge von Namen für Indexfamilien. Ein Parameter hat die Gestalt p:F, wobei $F \in \mathcal{I}$ eine Indexfamilie (Parametertyp) und p (Parameterwert) ein zulässiges Element von F ist.*

   Die Namen der vordefinierten Indexfamilien und ihrer zugehörigen Elemente sind den Einträgen der aktuellen Indexfamilientabelle zu entnehmen, die mindestens die Familie variable *enthalten muß.*

2. *Ein Operator hat die Gestalt $opid\{P_1, \ldots, P_n\}$ ($n \in \mathbb{N}$), wobei opid ein Operatorname und die P_i Parameter sind.*

3. *Es sei $\mathcal{V}$ eine unendliche Menge von Variablen (Elementen der Indexfamilie* variable*).*
   Terme und gebundene Terme sind induktiv wie folgt definiert.
   - *Ein gebundener Term hat die Gestalt $x_1, \ldots, x_m.t$ ($m \in \mathbb{N}$), wobei t ein Term ist und die x_i Variablen sind.*
   - *Ein Term hat die Gestalt $op(b_1; \ldots; b_n)$ ($n \in \mathbb{N}$), wobei op ein Operator und die b_i gebundene Terme sind.*

4. *Die Namen der vordefinieren Operatoren, ihre Parameter, Stelligkeiten und die zulässigen (gebundenen) Teilterme sind den Einträgen in der Operatorentabelle zu entnehmen, die mindestens den Operator* var *enthalten muß.*[16]

Diese Definition regelt die grundlegende Struktur von Termen, nicht aber die konkrete Ausprägung der vordefinierten Konzepte der Theorie. Sowohl die Indexfamilien als auch die Operatoren sind über Tabellen zu

[16] Durch diese Festlegung erhalten wir die Bedingung: '*Ist $x \in \mathcal{V}$, so ist* var{x:v}() *ein Term*' als einen Grundbestandteil jeder auf dieser Definition aufbauenden Theorie, also mehr oder weniger die Aussage '*jede Variable $x \in \mathcal{V}$ ist ein Term*'.

definieren, wobei in NuPRL als Einschränkung gilt, daß Operatornamen nichtleere Zeichenketten über dem Alphabet {`a–z A–Z 0–9 _ - !`} sein müssen. Die Operatorentabelle werden wir in diesem Kapitel schrittweise zusammen mit der Diskussion der verschiedenen Typkonstrukte aufbauen. Die aktuelle Tabelle der Indexfamilien (Parametertypen) ist dagegen relativ klein und komplett in Abbildung 3.1 zusammengestellt.

Für syntaktische Manipulationen und die Beschreibung des Berechnungsmechanismus ist das Konzept der Substitution von frei vorkommenden Variablen(-termen) durch Terme von großer Bedeutuug. Aufgrund der einfachen und einheitlichen Notation für Terme ist es nicht schwer, die hierbei notwendigen Begriffe präzise zu definieren. Im wesentlichen läuft dies darauf hinaus, daß Terme alle vorkommenden Variablen frei enthalten, solange diese nicht innerhalb eines gebundenen Terms an die Variable vor dem Punkt gebunden sind.

Definition 3.2.6 (Freies und gebundenes Vorkommen von Variablen in Termen)

Es seien $x, x_1, \ldots, x_m, y \in \mathcal{V}$ Variablen, $t, b_1, \ldots, b_n$ Terme und op ein Operator. Das $\underline{\text{freie}}$ und $\underline{\text{gebundene}}$ Vorkommen der Variablen x in einem Term ist induktiv durch die folgenden Bedingungen definiert.

1. *Im Term $\mathsf{var}\{x\!:\!\mathtt{v}\}$ kommt x frei vor. y kommt nicht vor, wenn x und y verschieden sind.*
2. *In $op(b_1;\ldots;b_n)$ bleibt jedes freie Vorkommen von x in den b_i frei und jedes gebundene Vorkommen gebunden.*
3. *In $x_1,\ldots,x_m.t$ wird jedes freie Vorkommen der Variablen x_i in t gebunden. Gebundene Vorkommen von x_i in t bleiben gebunden. Jedes freie Vorkommen der Variablen y bleibt frei, wenn y verschieden von allen x_i ist.*

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

Eine Term ohne freie Variablen heißt $\underline{\text{geschlossen}}$.

Auch das Konzept der Substitution hat nun eine relativ einfache Definition, die den Definitionen 2.2.20 und 2.3.7 (Seite 37 bzw. 50) relativ nahe kommt. Substitution der Variablen x durch den Term t innerhalb eines Terms u beedeutet, daß jedes freie Vorkommen von x in u durch t ersetzt wird, während jedes gebundene Vorkommen von x in u unverändert bleibt. Hierbei ist jedoch zu beachten, daß nicht etwa die Variable x selbst durch t ausgetauscht wird[17] sondern der Term, der aus der Variablen x gebildet wird, also $\mathsf{var}\{x\!:\!\mathtt{v}\}()$.

Definition 3.2.7 (Substitution)

Eine $\underline{\text{Substitution}}$ ist eine endliche Abbildung σ von der Menge der Terme der Gestalt $\mathsf{var}\{x\!:\!\mathtt{v}\}()$ mit $x\in\mathcal{V}$ in die Menge der Terme.

Gilt $\sigma(\mathsf{var}\{x_1\!:\!\mathtt{v}\}())=t_1,\ \ldots,\ \sigma(\mathsf{var}\{x_n\!:\!\mathtt{v}\}())=t_n$, so schreiben wir kurz $\sigma = \underline{[t_1,\ldots,t_n/x_1,\ldots,x_n]}$.

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

$$
\begin{array}{lll}
\mathsf{var}\{x\!:\!\mathtt{v}\}()[t/x] & = t & \\
\mathsf{var}\{x\!:\!\mathtt{v}\}()[t/y] & = \mathsf{var}\{x\!:\!\mathtt{v}\}(), & \textit{wenn } x \textit{ und } y \textit{ verschieden sind.}\\
(op(b_1;\ldots;b_n))[t/x] & = op(b_1[t/x];\ldots;b_n[t/x]) & \\
(x_1,\ldots,x_n.u)[t/x] & = x_1,\ldots,x_n.u, & \textit{wenn } x \textit{ eines der } x_i \textit{ ist.}\\
(x_1,\ldots,x_n.u)[t/x] & = x_1,\ldots,x_n.u[t/x] & \\
\end{array}
$$

wenn x verschieden von allen x_i ist, und es der Fall ist, daß x nicht frei in u ist oder daß keines der x_i frei in t vorkommt.

$$(x_1,..,x_j,..,x_n.u)[t/x] \;=\; (x_1,..,z,..,x_n.u[z/x_j])[t/x]$$

wenn x verschieden von allen x_i ist, frei in u vorkommt und x_j frei in t erscheint. z ist eine neue Variable, die weder in u noch in t vorkommt.

Dabei sind $x, x_1, \ldots, x_m, y \in \mathcal{V}$ Variablen, $t, b_1, \ldots, b_n$ Terme und op ein Operator.

Die Anwendung komplexerer Substitutionen auf einen Term – $u[t_1,\ldots,t_n/x_1,\ldots,x_n]$ – wird entsprechend durch ein simultane Induktion definiert.

[17] Die Ersetzung von `x` durch `λx.x` im Term `x` würde in diesem Fall zu einem Konstrukt führen, dessen Abstraktionsform **var**{**lam**(`x`.**var**{`x:v`}())`:v`}() ist anstelle des gewünschten **lam**(`x`.**var**{`x:v`}()).

Operator und Termstruktur	*Display Form*
var{x:v}()	x
fun{}(S; $x.T$)	$x\!:\!S\!\to\!T$
lam{}($x.t$)	$\lambda x.t$
apply{}(f;t)	$f\ t$
prod{}(S; $x.T$)	$x\!:\!S\!\times\!T$
pair{}(s;t)	$\langle s,t\rangle$
spread{}(e; $x,y.u$)	let $\langle x,y\rangle = e$ in u
union{}(S;T)	$S+T$
inl{}(s), **inr**{}(t)	inl(s), inr(t)
decide{}(e; $x.u$; $y.v$)	case e of inl(x) $\mapsto u$ | inr(y) $\mapsto v$

Abbildung 3.2: Operatorentabelle für Funktionenraum, Produktraum und Summe

Wir wollen die Auswirkungen dieser Definitionen am Beispiel von Funktionenraum, Produktraum und disjunkter Vereinigung illustrieren.

Beispiel 3.2.8

Abbildung 3.2 zeigt die Einträge der Operatorentabelle für Variablen, Funktionenraum, Produktraum und Summe. Mit Ausnahme des Operators var sind alle Operatoren parameterlos. Ein Eintrag wie **fun**{}(S; $x.T$) besagt, daß ein Operator mit Namen fun definiert wird, der zwei Argumente besitzt, von denen das erste ein ungebundener Term und das zweite ein mit einer Variablen gebundener Term sein muß. x, S und T sind Meta-Variablen zur Beschreibung dieser Tatsache und ihr Vorkommen in der Display Form gibt an, an welcher Stelle die verschiedenen Komponenten der Argumente von fun in der textlichen Darstellung wiederzugeben sind.

Gemäß Definition 3.2.6 wird eine Variable in T gebunden, wenn sie mit der für x eingesetzten Variablen identisch ist. Alle anderen Variablen behalten ihren Status.

Die weiteren Einträge spiegeln genau die 'Definitionen' von Termen und freien/gebundenen Vorkommen von Variablen wieder, die wir zu Beginn dieses Abschnitts aufgelistet haben.

3.2.2 Semantik: Werte und Eigenschaften typentheoretischer Ausdrücke

Anders als bei den im vorigen Kapitel vorgestellten Kalkülen wollen wir die Semantik der Typentheorie nicht auf einer anderen Theorie wie der Mengentheorie abstützen, sondern auf der Basis eines intuitiven Verständnisses gewisser mathematischer Grundkonstrukte formal definieren. Dabei gehen wir davon aus, daß elementare Grundkonstrukte wie die natürliche Zahlen eine natürliche und unumstrittene Bedeutung haben und erklären die Bedeutung komplexerer Konstrukte durch (rekursive) Definitionen unter Verwendung einer mathematischen Sprache, die ebenfalls nicht weiter erklärt werden muß. Wir erklären zunächst, wie der Wert eines typentheoretischen Ausdrucks zu bestimmen ist, und beschreiben dann auf dieser Basis die semantischen Eigenschaften von Termen.

3.2.2.1 Der Wert eines Ausdrucks

Typentheoretische Ausdrücke werden als Repräsentanten mathematischer Objekte angesehen. Unter diesen gibt es Terme wie `0,1,-1,2,-2,...` oder `λx.4`, die wir als Standard-Darstellung eines Objektes ansehen und solche, die als noch auszuwertende Ausdrucke betrachtet werden. Bei den ersteren gehen wir davon aus, daß sie mehr oder weniger einen direkten Bezug zu einem Objekt haben, der intuitiv klar ist bzw. nur mit intuitiven mathematischen Konzepten erklärt werden kann. Diese *kanonischen* Terme können also als (Standard-Repräsentanten für) Werte der Theorie angesehen werden. Die Bedeutung anderer, *nichtkanonischer* Terme muß dagegen durch eine Auswertung zu einem kanonischen Term bestimmt werden.

Für eine autarke Beschreibung der Semantik der Typentheorie ist also die Auswertung oder *Reduktion* von essentieller Bedeutung. Um zu garantieren, daß diese Reduktion (für typisierbare Terme gemäß Abschnitt 3.2.2.2) immer terminiert, nehmen wir eine gewisse methodische Anlehnung an die in Abschnitt 2.4.5.2 vorgestellte *TAIT computability method* vor. Bestimmte Terme werden durch syntaktische Kriterien als kanonisch definiert und gelten als nicht weiter reduzierbare Werte. Für die nichtkanonischen Ausdrücke werden bestimmte Argumente als *Hauptargumente* gekennzeichnet und Reduktion (Berechenbarkeit) wird definiert durch das Ergebnis einer Auswertung nichtkanonischer Ausdrücke mit kanonischen Hauptargumenten. Dieses Ergebnis kann anhand einer Tabelle von *Redizes* und *Kontrakta* bestimmt werden, die sich auf das (syntaktische) Konzept der Substitution stützt. So ist zum Beispiel f das Hauptargument eines Ausdrucks **apply**$\{\}(f\,;t)$ und Reduktion wird erklärt durch Auswertung von Redizes der Art **apply**$\{\}($**lam**$\{\}(x\,.u)\,;t)$ (also $(\lambda x\,.u)\ t$) zum Term $u[t/x]$.

Dies alleine garantiert natürlich noch keine immer terminierenden Reduktionen. Es hängt davon ab, *welche* Terme als kanonisch bezeichnet werden. Für eine Theorie, die so ausdrucksstark ist wie in Abschnitt 3.1.2 gefordert, müssen wir dafür sorgen, daß Reduktionen frühzeitig gestoppt werden. Die einfachste Vorgehensweise, mit der dies erreicht werden kann, ist, die Eigenschaft "kanonisch" ausschließlich an der *äußeren* Struktur eines Terms festzumachen, also zum Beispiel eine λ-Abstraktion $\lambda x\,.t$ grundsätzlich als kanonischen Term zu deklarieren – unabhängig davon, ob der Teilterm t selbst kanonisch ist.[18] Der Auswerungsmechanismus, der sich hieraus ergibt, heißt in der Fachsprache *lässige Auswertung* (*lazy evaluation*): man reduziere die Hauptargumente eines Terms, bis sie in kanonischer Form sind, ersetzte das entstehende Redex durch sein Kontraktum und verfahre dann weiter wie zuvor, bis man eine kanonische Form erreicht hat.

Entwurfsprinzip 3.2.9 (Lazy Evaluation)

Die Semantik der Typentheorie basiert auf Lazy Evaluation

Wir wollen nun das bisher gesagte durch entsprechende Definitionen präzisieren.

Definition 3.2.10 (Werte)

Ein Term heißt <u>kanonisch</u>, wenn sein Operatorname in der Operatorentabelle als werteproduzierend *(kanonisch) gekennzeichnet wurde. Ansonsten heißt er <u>nichtkanonisch</u>.*

Ein geschlossener kanonischer Term wird als <u>Wert</u>[19] *bezeichnet.*

[18] Dies ist bei genauerem Hinsehen nicht weniger natürlich als ein Wertebegriff, der eine maximal mögliche Reduktion verlangt. Der wesentliche Aspekt eines Terms $\lambda x\,.t$ ist, daß er eine Funktion beschreibt. Die genaue Bedeutung dieser Funktion kann ohnehin erst erfaßt werden, wenn man sie auf einem Eingabeargument auswertet. Hierbei macht es keinen Unterschied, ob der Term t zuvor schon soweit wir möglich reduziert war, da man ohnehin mindestens einen Auswertungsschritt vornehmen muß. Zugegebenermaßen ist aber die reduzierte Form etwas leichter lesbar. Da man diese nicht immer erreichen kann, müssen wir hierauf allerdings verzichten.

[19] Das Wort *Wert* (englisch *value*) ist an dieser Stelle etwas irreführend, da die meisten Menschen ein anderes Verständnis von diesem Begriff haben. Das Problem liegt darin begründet, daß Menschen normalerweise keinen Unterschied machen zwischen einem semantischen Konzept und dem Text, der zur Beschreibung dieses Konzeptes benötigt wird. Der Wert im Sinne von Definition 3.2.10 ist ein "Beschreibungswert", also genau besehen nur ein Stück Text. Es ist daher immer noch möglich, daß verschiedene solcher Werte dasselbe semantische Objekt beschreiben, also *semantisch gleich* im Sinne von Definition 3.2.15 sind.

Warum ist es sinnvoll, in dieser scheinbar so verwirrenden Weise die Grundlagen einer formalen Theorie aufzubauen? Gibt es nicht immer nur *einen* Beschreibungswert für ein Objekt? Das Beispiel der reellen Zahlen zeigt, daß dies im Normalfall nicht mehr garantiert werden kann. Denn unter allen Beschreibungen einer irrationalen Zahl durch Folgen rationaler Zahlen kann man keine fixieren, die eine gute "Standard"-Beschreibung ist in die alle anderen Beschreibungen reeller Zahlen umgewandelt werden können. Selbst in den Einzelfällen, in denen dies möglich wäre, ist der Berechnungsaufwand für die Standardisierung zu hoch. In einer formalen Theorie aber müssen (Beschreibungs-)Werte für Objekte in endlicher Zeit berechnet und nicht nur abstrakt erklärt werden können.

Technisch ist Lazy Evaluation der einzig sinnvolle Weg, dies zu realisieren. Über Auswertung werden die Beschreibungswerte bestimmt, die man nicht weiter vereinfachen kann. Auf diese Werte kann man dann zurückgreifen, wenn man auf formale Weise die Semantik von Termen und Urteilen – also den Bezug zwischen einer Beschreibung und der Realität – festlegen will. Diese Semantik wird zwangsläufig etwas komplizierter ausfallen, da sie unter anderem präzisieren muß, wann zwei Werte dasselbe Objekt bezeichnen sollen, und somit die syntaktischen Restriktionen der lässigen Auswertung wieder aufhebt.

Wie die Semantik wird auch das Inferenzsystem der Typentheorie *nicht* den Einschränkungen der lässigen Auswertung unterliegen, da es – im Gegensatz zur Auswertung von Termen – nicht durch eine feste Strategie gesteuert werden muß.

Algorithmus $\boxed{\texttt{EVAL}(t)}$: *(t: beliebiger typentheoretischer Term)*

- Falls t in kanonischer Form ist, gebe t als Wert aus.
- Falls t nichtkanonisch ist, bestimme s_1:=`EVAL`(t_1), ..., s_n:=`EVAL`(t_n), wobei die t_i die Hauptargumente von t sind. Ersetze in t die Argumente t_i durch ihre Werte s_i und nenne das Resultat s
  - Falls s ein Redex ist und u das zugehörige Kontraktum, so gebe `EVAL`(u) als Wert aus.
  - Andernfalls stoppe die Reduktion ohne Ergebnis: t besitzt keinen Wert.

Abbildung 3.3: Die Auswertungsprozedur der Typentheorie

Die Definition des Begriffs *Wert* hängt also ausschließlich von der äußeren Struktur eines Terms – d.h. von seinem Operatornamen – ab. Diese Konzeption hat den Vorteil, daß Werte sehr leicht, nämlich durch Nachschlagen des Operatornamens in der Operatorentabelle, also solche identifiziert werden können. Natürlich muß man hierzu die Operatorentabelle, die wir in Abbildung 3.2 gegeben haben, um eine entsprechende Unterteilung erweitern. Wir wollen dies tun, nachdem wir die Semantik nichtkanonischer Terme erklärt haben.

Definition 3.2.11 (Reduktion)

Die <u>prinzipiellen Argumente</u> (Hauptargumente) eines nichtkanonischen Terms t sind diejenigen Teilterme, die in der Operatorentabelle als solche gekennzeichnet wurden.

Ein <u>Redex</u> ist ein nichtkanonischer Term t, dessen Hauptargumente in kanonischer Form sind und der in der <u>Redex–Kontrakta Tabelle</u> als Redex erscheint. Das <u>Kontraktum von t</u> ist der entsprechend zugehörige Eintrag in derselben Tabelle.

Ein Term t heißt β-<u>reduzierbar auf</u> u – im Zeichen $\underline{t \xrightarrow{\beta} u}$ –, wenn t ein Redex und u das zugehörige Kontraktum ist.

Definition 3.2.12 (Semantik typentheoretischer Terme)

Der <u>Wert eines geschlossenen Terms t</u> ist derjenige Wert, der sich durch Anwendung der Auswertungsprozedur `EVAL` *aus Abbildung 3.3 auf t ergibt.*

Ist u der Wert des Terms t, so schreiben wir auch $\underline{t \xrightarrow{l} u}$.

Wir wollen die Auswertung von Termen an einigen Beispielen illustrieren.

Beispiel 3.2.13

Abbildung 3.4 zeigt die Einträge der Operatorentabelle für Variablen, Funktionenraum, Produktraum und Summe, die wir um eine Klassifizierung in kanonische und nichtkanonische Terme erweitert haben. Die Hauptargumente nichtkanonischer Terme haben wir dabei durch eine Umrahmung gekennzeichnet.

kanonisch		**nichtkanonisch**
(Typen)	(Elemente)	
	var$\{x\!:\!\mathtt{v}\}()$ x	
fun$\{\}(S\,;x\,.T)$ $x\!:\!S\!\to\! T$	**lam**$\{\}(x\,.t)$ $\lambda x\,.t$	**apply**$\{\}(\boxed{f};t)$ $\boxed{f}\ t$
prod$\{\}(S\,;x\,.T)$ $x\!:\!S\!\times\! T$	**pair**$\{\}(s\,;t)$ $\langle s\,,t\rangle$	**spread**$\{\}(\boxed{e};\,x\,,y\,.u)$ let $\langle x\,,y\rangle = \boxed{e}$ in u
union$\{\}(S;T)$ $S\!+\!T$	**inl**$\{\}(s)$, **inr**$\{\}(t)$ inl(s), inr(t)	**decide**$\{\}(\boxed{e};\,x\,.u;\,y\,.v)$ case $\boxed{e}$ of inl$(x)\mapsto u$ | inr$(y)\mapsto v$

Abbildung 3.4: Vollständige Operatorentabelle für Funktionenraum, Produktraum und Summe

Redex		Kontraktum
$(\lambda x.u)\ t$	$\xrightarrow{\beta}$	$u[t/x]$
$\mathsf{let}\ \langle x,y\rangle = \langle s,t\rangle\ \mathsf{in}\ u$	$\xrightarrow{\beta}$	$u[s,t\,/\,x,y]$
$\mathsf{case}\ \mathsf{inl}(s)\ \mathsf{of}\ \mathsf{inl}(x)\mapsto u\ \mid\ \mathsf{inr}(y)\mapsto v$	$\xrightarrow{\beta}$	$u[s/x]$
$\mathsf{case}\ \mathsf{inr}(t)\ \mathsf{of}\ \mathsf{inl}(x)\mapsto u\ \mid\ \mathsf{inr}(y)\mapsto v$	$\xrightarrow{\beta}$	$v[t/y]$

Abbildung 3.5: Redex–Kontrakta Tabelle für Funktionenraum, Produktraum und Summe

Der besseren Lesbarkeit wegen wurde ein Sortierung in Typen und Elemente vorgenommen, die aber nur informativen Charakter hat, und die Display Form unter die Abstraktionsform geschrieben. So ist zum Beispiel jedes Tupel $\langle s,t\rangle$ kanonisch und gehört zu einem (kanonischen) Typ $x{:}S{\times}T$. Die Operation $\mathsf{let}\ \langle x,y\rangle = e\ \mathsf{in}\ u$ ist die zugehörige nichtkanonische Form, deren Hauptargument e ist.

Die zugehörigen Einträge der Redex–Kontrakta Tabelle sind in Abbildung 3.5 wiedergegeben. Entsprechend dieser Einträge liefert EVAL für einige Beispielterme die folgenden Resultate.

$(\lambda\mathtt{x}.\mathsf{inl}(\mathtt{x}))\ \mathtt{y}$	$\stackrel{l}{\longrightarrow}$	$\mathsf{inl}(\mathtt{y})$
$\mathsf{inl}((\lambda\mathtt{x}.\mathtt{x})\ \mathtt{y})$	$\stackrel{l}{\longrightarrow}$	$\mathsf{inl}((\lambda\mathtt{x}.\mathtt{x})\ \mathtt{y})$
$(\lambda\mathtt{x}.\mathsf{inl}(\mathtt{x}))\ (\lambda\mathtt{y}.\mathtt{y})\ \mathtt{z}$	$\stackrel{l}{\longrightarrow}$	$\mathsf{inl}((\lambda\mathtt{y}.\mathtt{y})\ \mathtt{z})$
$(\lambda\mathtt{x}.\mathsf{inl}(\mathtt{x}))\ ((\lambda\mathtt{y}.\mathtt{y}\ \mathtt{y})\ (\lambda\mathtt{y}.\mathtt{y}\ \mathtt{y}))$	$\stackrel{l}{\longrightarrow}$	$\mathsf{inl}((\lambda\mathtt{y}.\mathtt{y}\ \mathtt{y})\ (\lambda\mathtt{y}.\mathtt{y}\ \mathtt{y}))$
$\mathsf{let}\ \langle\mathtt{f},\mathtt{b}\rangle = \langle\lambda\mathtt{x}.\mathsf{inl}(\mathtt{x}),\mathtt{y}\rangle\ \mathsf{in}\ \mathtt{f}\ \mathtt{b}$	$\stackrel{l}{\longrightarrow}$	$\mathsf{inl}(\mathtt{y})$ (via $(\lambda\mathtt{x}.\mathsf{inl}(\mathtt{x}))\ \mathtt{y}$)
$\mathsf{let}\ \langle\mathtt{f},\mathtt{b}\rangle = \mathsf{inl}(\mathtt{x})\langle\lambda\mathtt{x}.\mathsf{inl}(\mathtt{x}),\mathtt{y}\rangle\ \mathsf{in}\ \mathtt{f}\ \mathtt{b}$	$\stackrel{l}{\longrightarrow}$	(kein Wert)

Man beachte, daß wegen der lässigen Auswertung der Wert eines Terms durchaus ein Term sein kann, welcher einen nichtreduzierenden Teilterm enthält.

Für Funktionen läßt sich die in der Prozedur EVAL enthaltene lässige Reduktionsrelation mit der in Definition 2.3.11 (Seite 51) gegebenen *strikten Reduktion* vergleichen. Der Unterschied besteht darin, daß zugunsten der Terminierungsgarantie auf ξ- und μ-Reduktion verzichtet wurde. Die anderen Vorschriften zur Auflösung der Termstruktur sind nach wie vor in Kraft.

Aufgrund der Verwendung lässiger Auswertung von Termen können wir die Gleichheit zweier Terme nicht mehr so einfach definieren wie in Definition 2.3.24 (Seite 58), in der es ausreichte, daß die beiden Terme zu demselben Term reduzieren. Lässige Auswertung hat zur Folge, daß die Terme $\lambda\mathtt{x}.5$ und $\lambda\mathtt{x}.3{+}2$ bereits einen Wert darstellen und *nicht* auf denselben Wert reduziert werden, obwohl sie offensichtlich gleich sind. Wir müssen daher die semantische Gleichheit von der Normalform entkoppeln und als Eigenschaft betrachten, die semantisch durch ein *Urteil* zu definieren ist.

3.2.2.2 Urteile

Urteile sind die semantischen Grundbausteine, mit denen Eigenschaften von Termen und die Beziehungen zwischen Termen formuliert werden. Sie legen fest, welche Grundeigenschaften innerhalb einer Theorie *gültig* sein sollen. In einem Beweis bilden sie die *Behauptung*, die als wahr nachzuweisen ist. Sie selbst sind aber kein Bestandteil der formalen Sprache, da *über* ein Urteil – also die Frage *ob* es gültig ist oder nicht – nicht geschlossen werden kann.[20] In der Prädikatenlogik war das Grundurteil eine Aussage, die durch Formeln beschrieben werden konnte. Im λ-Kalkül war es die Gleichheit von λ-Termen und in der einfachen Typentheorie die Typisierung und die Eigenschaft, Typausdruck zu sein. All diese Urteile müssen ein Gegenstück in den

[20] Allerdings werden Beweise und Inferenzregeln der Theorie so ausgelegt werden, daß sie die Bedeutung der Urteile respektieren. So wird bei einer groben Betrachtung kaum ein Unterschied zwischen Urteilen und Ausdrücken der formalen Sprache zu sehen sein. Der Unterschied ist subtil: eine logische Formel kann zu wahr oder zu falsch ausgewertet werden, da sie nur ein Ausdruck ist; ein Urteil dagegen legt fest, *wann* etwas wahr ist, und kann daher nicht ausgewertet werden, ohne die Theorie zu verlassen.

Urteilen der Typentheorie finden, wobei wir Formeln durch Typen ausdrücken, deren Elemente die Beweise beschreiben. Deshalb benötigen wir in der Typentheorie mehrere Arten von Urteilen.

Definition 3.2.14 (Urteile)

Ein Urteil in der Typentheorie hat eine der folgenden vier Gestalten.

- $\underline{T\ \mathrm{Typ}}$: *Der Term T ist ein Ausdruck für einen Typ (Menge).* *(Typ-Sein)*
- $\underline{S{=}T}$: *Die Terme S und T beschreiben denselben Typ.* *(Typgleichheit)*
- $\underline{t\in T}$: *Der Term t beschreibt ein Element des durch T charakterisierten Typs.* *(Typzugehörigkeit)*
- $\underline{s{=}t\in T}$: *Die Terme s und t beschreiben dasselbe Element des durch T charakterisierten Typs.* *(Elementgleichheit)*

Was soll nun die Bedeutung dieser Urteile sein? In unserer Diskussion zu Beginn von Abschnitt 2.4 hatten wir bereits festgelegt, daß ein Typ definiert ist durch seine kanonischen Elemente und die zulässigen Operationen auf seinen Elementen. Wir hatten ebenfalls angedeutet, daß die Gleichheit von Elementen von dem Typ abhängt, zu dem sie gehören. Somit muß die Eigenschaft, ein Typ zu sein, die Begriffe der Typzugehörigkeit und der typisierten Gleichheit mit enthalten. Zwei Typen können nur dann gleich sein, wenn sie dieselben Elemente enthalten und dieselbe Form von Gleichheit induzieren. Diese Bedingung ist allerdings nur notwendig und nicht unbedingt hinreichend, denn der strukturelle Aufbau spielt ebenfalls eine Rolle. Ebenso klar ist, daß die Gleichheitsurteile Äquivalenzrelationen auf Termen sein müssen und die Semantik von Ausdrücken zu respektieren haben: ein nichtkanonischer Term t muß genauso beurteilt werden wie der Wert, auf den er reduziert. Diese Überlegungen führen zu der folgenden allgemeinen Vorgehensweise, die bei einer präzisen Definition der Bedeutung von Urteilen befolgt werden sollte.

- $\underline{T\ \mathrm{Typ}}$: Der Begriff des Typ-Seins ist an kanonischen Termen festzumachen und für nichtkanonische Ausdrücke entsprechend an ihren Wert zu binden.

  Unter den kanonischen Ausdrücken werden bestimmte Ausdrücke als kanonische Typausdrücke gekennzeichnet, wobei einerseits ihre Syntax – also zum Beispiel $x{:}S{\to}T$ – eine Rolle spielt, andererseits aber auch gewisse semantische Vorbedingungen gestellt werden müssen – wie zum Beispiel, daß S und T selbst Typen sind und $T[s/x]$ und $T[s'/x]$ für gleiche Werte s, s' aus S auch denselben Typ bezeichnen.

  Ein Ausdruck T beschreibt nun genau dann einen Typ, wenn er auf einen kanonischen Typausdruck reduziert.

- $\underline{S{=}T}$: Auch die Typgleichheit muß zunächst relativ zu kanonischen Typen definiert werden. Dies erfordert allerdings eine komplexe induktive Definition, die von der Struktur der Typausdrücke abhängt.

  $S{=}T$ gilt, wenn S und T auf denselben kanonischen Typ reduzieren.

- $\underline{t\in T}$ gilt, wenn T ein Typ ist und t zu einem kanonischen Element von T reduziert, was wiederum induktiv zu definieren ist.

- $\underline{s{=}t\in T}$ gilt, wenn s und t zu demselben kanonischen Element von T reduzieren, was ebenfalls eine induktive Definition erfordert.

Es ist leicht zu sehen, daß die Typgleichheit das Typ-Sein subsumiert, da zwei Typen nur gleich sein können, wenn sie überhaupt Typen darstellen. Aus dem gleichen Grunde stellt die Elementgleichheit eine Verallgemeinerung der Typzugehörigkeit dar. Wir können die Bedeutung der Urteile also im wesentlichen dadurch erklären, daß wir die Gleichheit kanonischer Terme in der Form von Tabelleneinträgen fixieren und die anderen Konzepte hierauf abstützen. Dies macht natürlich nur Sinn für geschlossene Terme, da ansonsten kein Wert bestimmt werden kann.

Definition 3.2.15 (Bedeutung von Urteilen)

Die Semantik konkreter typentheoretischer Urteile ist für geschlossene Terme s, t, S und T induktiv wie folgt definiert.

- *T Typ gilt genau dann, wenn T=T gilt.*
- *S=T gilt genau dann, wenn es kanonische Terme S' und T' gibt, deren Typgleichheit S'=T' aus einem der Einträge in der Typsemantiktabelle folgt und für die $S \xrightarrow{l} S'$ und $T \xrightarrow{l} T'$ gilt*
- *$t \in T$ gilt genau dann, wenn t=$t \in T$ gilt.*
- *s=$t \in T$ gilt genau dann, wenn es kanonische Terme s', t' und T' gibt, deren Elementgleichheit s'=$t' \in T'$ aus einem der Einträge in der Elementsemantiktabelle folgt und für die $s \xrightarrow{l} s'$, $t \xrightarrow{l} t'$ und T=T' gilt.*

Man beachte hierbei, daß sich die einzelnen Definitionen der Urteile gegenseitig beeinflussen. Dies wird besonders deutlich in den Semantiktabellen, innerhalb derer eine Vielfalt von Querbezügen hergestellt wird. Die Bestimmung der Semantik ist somit ein ständiges Wechselspiel von Reduktion und der Konsultation von Tabelleneinträgen. Die Einträge in der Tabelle wiederum richten sich nach dem intuitiven Verständnis der entsprechenden Konzepte, das nur genügend präzisiert werden muß, wobei man der syntaktischen Struktur der Terme folgt. Wir wollen dies am Beispiel von Funktionenraum, Produktraum und Summe illustrieren.

Beispiel 3.2.16

Zwei Funktionenräume $S_1 \to T_1$ und $S_2 \to T_2$ sind gleich, wenn die Argumentebereiche S_1 und S_2 und die Wertebereiche T_1 und T_2 gleich sind. Verallgemeinert man dies auf abhängige Funktionenräume $x_1{:}S_1 \to T_1$ und $x_2{:}S_2 \to T_2$, so kommt zusätzlich hinzu, daß T_1 und T_2 auch dann gleich bleiben, wenn sie durch gleichwertige (aber syntaktisch verschiedene) Elemente von S_1 (bzw. S_2) beeinflußt werden. Abhängige und unabhängige Funktionenräume sind kompatibel in dem Sinne, daß $S_1 \to T_1$ identisch ist mit $x_1{:}S_1 \to T_1$, wobei x_1 eine beliebige Variable ist.[21] Für diese Definition, die in der Typsemantiktabelle in Abbildung 3.6 präzisiert wird, benötigt man die Typgleichheit der Teilterme und die Gleichheit von Elementen aus S_1, was zu einer rekursiven Verwendung von Definition 3.2.15 führt.

Zwei Funktionen $\lambda x_1.t_1$ und $\lambda x_2.t_2$ sind gleich in $x{:}S \to T$, falls $x{:}S \to T$ überhaupt ein Typ ist (was keineswegs sicher ist) und die Funktionskörper t_1 bzw. t_2 sich auf gleichen Eingaben aus S auch in T gleich verhalten.

Abbildung 3.6 stellt diese Urteile und die entsprechenden Urteile für Produkträume und disjunkte Vereinigungen zusammen.

Man beachte, daß die Semantik der Gleichheitsurteile – im Gegensatz zur Auswertung – *keine lässige Auswertung* zuläßt, sondern durch einen ständigen Wechsel von Reduktion und Tabellenverwendung einen Term bis in das letzte Detail analysiert. Aus der Definition 3.2.15 und den bisher angegebenen Semantiktabellen lassen sich folgende Eigenschaften der semantischen Urteile ableiten.

Lemma 3.2.17

1. *Typgleichheit S=T ist transitiv und symmetrisch (aber nicht reflexiv)*
2. *Für alle Terme S und T gilt S=T genau dann, wenn es einen Term S' gibt mit $S \xrightarrow{l} S'$ und S'=T.*
3. *Elementgleichheit s=$t \in T$ ist transitiv und symmetrisch (aber nicht reflexiv) in s und t.*
4. *Für alle Terme s, t und T gilt s=$t \in T$ genau dann, wenn es einen Term s' gibt mit $s \xrightarrow{l} s'$ und s'=$t \in T$.*
5. *Für alle Terme s, t und T folgt aus s=$t \in T$ immer, daß T ein Typ (T=T) ist.*
6. *Für alle Terme s, t, S und T folgt aus s=$t \in T$ und S=T, daß s=$t \in S$ gilt.*

[21] Da $S_1 \to T_1$ geschlossen sein muß, kann x_1 in T_1 nicht frei vorkommen und es entstehen keinerlei Probleme.

Typsemantik		
$x_1{:}S_1 \to T_1 \;=\; x_2{:}S_2 \to T_2$	falls	S_1=S_2 und $T_1[s_1/x_1]$=$T_2[s_2/x_2]$ für alle Terme s_1, s_2 mit s_1=$s_2 \in S_1$.
$T \;=\; S_2 \to T_2$	falls	$T \;=\; x_2{:}S_2 \to T_2$ für ein beliebiges $x_2 \in \mathcal{V}$.
$S_1 \to T_1 \;=\; T$	falls	$x_1{:}S_1 \to T_1 \;=\; T$ für ein beliebiges $x_1 \in \mathcal{V}$.
$x_1{:}S_1 \times T_1 \;=\; x_2{:}S_2 \times T_2$	falls	S_1=S_2 und $T_1[s_1/x_1]$=$T_2[s_2/x_2]$ für alle Terme s_1, s_2 mit s_1=$s_2 \in S_1$.
$T \;=\; S_2 \times T_2$	falls	$T \;=\; x_2{:}S_2 \times T_2$ für ein beliebiges $x_2 \in \mathcal{V}$.
$S_1 \times T_1 \;=\; T$	falls	$x_1{:}S_1 \times T_1 \;=\; T$ für ein beliebiges $x_1 \in \mathcal{V}$.
$S_1 + T_1 \;=\; S_2 + T_2$	falls	S_1=S_2 und T_1=T_2.
Elementsemantik		
$\lambda x_1.t_1 \;=\; \lambda x_2.t_2 \;\in\; x{:}S \to T$	falls	$x{:}S \to T$ Typ und $t_1[s_1/x_1] \;=\; t_2[s_2/x_2] \;\in\; T[s_1/x]$ für alle Terme s_1, s_2 mit s_1=$s_2 \in S$.
$\langle s_1, t_1 \rangle \;=\; \langle s_2, t_2 \rangle \;\in\; x{:}S \times T$	falls	$x{:}S \times T$ Typ und s_1=$s_2 \in S$ und t_1=$t_2 \in T[s_1/x]$.
$\mathsf{inl}(s_1) \;=\; \mathsf{inl}(s_2) \;\in\; S+T$	falls	$S+T$ Typ und s_1=$s_2 \in S$.
$\mathsf{inr}(t_1) \;=\; \mathsf{inr}(t_2) \;\in\; S+T$	falls	$S+T$ Typ und t_1=$t_2 \in T$.

Abbildung 3.6: Semantik der Urteile für Funktionenraum, Produktraum und Summe

Darüber hinaus kann man auch noch zeigen, daß die Semantik der Urteile die Reduktion auch dann respektiert, wenn sie sich nicht an lässige Auswertung hält. Wenn zwei Terme sich durch beliebige Reduktionen auf denselben Term reduzieren lassen (also *berechnungsmäßig äquivalent* sind), dann gilt Typgleichheit, falls einer von ihnen ein Typ ist, und Elementgleichheit in einem Typ T, falls einer von ihnen zu T gehört.

Diese Aussagen lassen sich unter Verwendung der konkreten Semantiktabellen beweisen und gelten somit zunächst nur für Funktionenraum, Produktraum und disjunkte Vereinigung. Die Erweiterungen der Semantiktabellen, die wir in den folgenden Abschnitten vornehmen werden, werden allerdings so ausgelegt, daß Lemma 3.2.17 und die obige Aussage weiterhin Gültigkeit behalten werden.

3.2.2.3 Hypothetische Urteile

Um ein Argumentieren unter vorgegebenen *Annahmen* zu ermöglichen, ist es nötig, das Konzept des Urteils auf *hypothetische Urteile* zu erweitern. Diese erlauben es, zu beschreiben, daß ein bestimmtes Urteil wahr ist, falls bestimmte Voraussetzungen erfüllt sind. In ihrer textlichen Beschreibungsform sind hypothetische Urteile sehr ähnlich zu Sequenzen:

$$A_1,\ldots,A_n \vdash J$$

Dies soll ausdrücken, daß das Urteil (judgment) J aus den Annahmen A_i folgt. Man beachte jedoch, daß im Unterschied zu einer Sequenz ein hypothetisches Urteil ein semantisches Konzept ist. Innerhalb der Typentheorie beschränken wir uns auf die vier in Definition 3.2.14 festgelegten Arten von Urteilen und auf Annahmen der Art $x_i{:}T_i$ – also auf Typdeklarationen. Das hypothetische Urteil

$$\underline{x_1{:}T_1,\ldots,x_n{:}T_n \vdash J}$$

ist also zu lesen als

Unter der Annahme, daß x_i Variablen vom Typ T_i sind, gilt das Urteil J.

Eine präzise Definition hypothetischer Urteile muß *Funktionalität* und *Extensionalität* garantieren. In erster Näherung ist dies die Eigenschaft, daß $J[t_i/x_i]$ für alle kanonischen Elemente $t_i \in T_i$ gelten muß, und daß für alle (auch nichtkanonischen) Terme t_i und t_i' folgt, daß $J[t_i/x_i]$ und $J[t_i'/x_i]$ gleich beurteilt werden, falls t_i=$t_i' \in T_i$ gilt.

Auf eine genaue Erklärung dieser Eigenschaften wollen wir an dieser Stelle verzichten. Interessierte können Details finden in [Constable *et.al.*, 1986, Seite 141] und [Martin-Löf, 1984, Seite 16ff].

3.2.3 Inferenzsystem

Durch Reduktion und Urteile ist die Semantik von Ausdrücken der Typentheorie eindeutig definiert. Wir wollen nun beschreiben, wie man diese Semantik durch rein syntaktische Mittel innerhalb eines Inferenzsystems für die Typentheorie so simulieren kann, daß ein formales logisches Schließen über alle Bestandteile der Theorie möglich ist. Wir müssen hierzu im wesentlichen eine syntaktische Repräsentation der (hypothetischen) Urteile angeben und Inferenzregeln entwerfen, die den Einträgen in der Semantiktabelle entsprechen.[22]

Entsprechend der textlichen Beschreibungsform hypothetischer Urteile sind Sequenzen das angemessenste Mittel einer syntaktischen Repräsentation innerhalb eines Kalküls.[23] Die bisherige Gestalt von Sequenzen war

$$x_1{:}T_1,\ldots,x_n{:}T_n \vdash C,$$

wobei die x_i Variablen und die T_i Ausdrücke sind und durch die Anforderungen an den Kalkül sichergestellt war, daß die T_i Typen darstellten. Welche Gestalt aber soll nun die Konklusion C – das Gegenstück zu den einfachen Urteilen – besitzen? Wir möchten hierfür nur ein einziges syntaktisches Konzept verwenden, müssen aber insgesamt vier Arten von Urteilen repräsentieren. Wir wollen daher vor einer formalen Definition von Sequenzen diskutieren, auf welche Art wir diese Anforderung erreichen können und welche Auswirkungen dies auf den Kalkül der Typentheorie hat.

3.2.3.1 Universen: syntaktische Repräsentation der Typ-Eigenschaft

Für die syntaktische Repräsentation der Typeigenschaft hatten wir bereits im Zusammenhang mit der einfachen Typentheorie ein *Typuniversum* eingeführt, das wir mit dem Symbol $\mathbb{U}$ gekennzeichnet hatten. Unter Verwendung dieses Symbols können wir das Urteil 'T *Typ*' durch '$T \in \mathbb{U}$' darstellen und '$S{=}T$' durch '$S{=}T \in \mathbb{U}$'. Damit hätten wir Typ-Sein und Typgleichheit auf die Urteile Typzugehörigkeit und Elementgleichheit zurückgeführt und brauchen uns nur noch um eine einheitliche Darstellung dieser beiden Urteile zu bemühen.

Welche Rolle aber spielt $\mathbb{U}$ in unserer formalen Theorie? Offensichtlich muß es ein Ausdruck der formalen Sprache sein und sein Verwendungszweck suggeriert, daß es einen Typausdruck kennzeichnet. Dies würde bedeuten, daß $\mathbb{U}$ als ein Typausdruck selbst zum Universum aller Typen gehören muß. Damit müßte das Urteil $\mathbb{U} \in \mathbb{U}$ gültig sein. Können wir dies nun vermeiden? Das folgende Beispiel zeigt, daß $\mathbb{U}$ in der Tat nicht nur aus formalen Gründen ein Typ sein muß.

Beispiel 3.2.18

In Beispiel 3.1.1 auf Seite 97 hatten wir die Notwendigkeit abhängiger Datentypen für eine hinreichend ausdrucksstarke Theorie begründet. Abhängigkeit bedeutet dabei zum Beispiel für einen Funktionenraum $x{:}S{\to}T$, daß der Datentyp T eine freie Variable x enthalten muß, die mit Werten aus S belegt werden darf. Die einzige Möglichkeit, dies syntaktisch korrekt zu beschreiben, ist die Verwendung einer Funktion $\hat{T}$, die bei Eingabe eines Wertes $s \in S$ genau auf den Wert $T[s/x]$ reduziert, was ein Typ – also ein Element von $\mathbb{U}$ – sein muß. $\hat{T}$ *muß* somit (mindestens) den Typ $S{\to}\mathbb{U}$ besitzen.

Der Ausdruck $\mathbb{U}$ wird also als Bestandteil von Typaudrücken vorkommen müssen und wenn wir kein neues Sonderkonzept zusätzlich zu Typen und Elementen[24] einführen wollen, dann müssen wir $\mathbb{U}$ selbst als Typaus-

[22] Für das intuitive Verständnis der Typentheorie ist dieser Teilabschnitt vielleicht der bedeutendste aber zugleich auch der schwierigste, da eine Reihe scheinbar willkürlicher "Entwurfsentscheidungen" getroffen werden müssen, deren Tragweite erst bei einer gewissen Erfahrung im Umgang mit dieser Theorie deutlich wird. Einen theoretisch zwingenden Grund, das Inferenzsystem in der hier vorgestellten Form auszulegen, gibt es nicht. Es ist eher das Resultat langjähriger Erfahrungen mit der Automatisierung einer mathematisch formalen Grundlagentheorie, bei denen die Vor- und Nachteile verschiedener Möglichkeiten sorgsam gegeneinander abgewägt werden mußten. Oft zeigten auch die Experimente, daß mancher Weg, der theoretisch so einfach erschien, zu unlösbaren praktischen Problemen führte. Daher werden manche Entscheidungen erst im Rückblick einsichtig.

[23] Die formale Ähnlichkeit ist in der Tat so groß, daß Sequenzen und hypothetische Urteile oft miteinander verwechselt werden. Der Unterschied besteht darin, daß die Bestandteile eines hypothetischen Urteils tatsächlich Annahmen (Deklarationen) und Urteile sind, während eine Sequenz eigentlich nur ein Stück Text ist.

[24] Diesen Weg hat Girard [Girard, 1971, Girard, 1986, Girard *et.al.*, 1989] in seinem System $\mathcal{F}$ eingeschlagen, um die ursprünglichen Paradoxien der frühen Martin-Löf'schen Ansätze [Martin-Löf, 1970] zu umgehen.

druck zulasssen. Das Russelsche Paradoxon der Mengentheorie wird hiervon nicht berührt, da es sich bei $\mathbb{U}$ um eine ganz spezielle "maximale" Klasse handelt, die ein Element von sich selbst ist, und diese Eigenschaft keineswegs für andere Mengen (Typen) erlaubt wird.

Die Wahl eines Typuniversums $\mathbb{U}$, das selbst als Typ verwandt werden darf, würde die Theorie übrigens auch sehr vereinfachen, da in diesem Fall viele der von uns gewünschten Typkonstrukte durch einen abhängigen Funktionenraum simuliert und damit als definitorische Abkürzung eingeführt werden könnten.[25] So könnten wir durch das Typuniversum $\mathbb{U}$ auf einfache Weise einen Kalkül höherer Ordnung einführen, der extrem ausdrucksstark ist. Leider ermöglicht die Kombination von abhängigen Typen und dem Axiom $\mathbb{U} \in \mathbb{U}$ jedoch ein anderes Paradoxon, das zwar sehr viel komplizierter ist als das Russelsche Paradoxon, aber dennoch die Widersprüchlichkeit einer solchen Theorie aufdeckt. Mit diesem Paradoxon (*Girard's Paradoxon*, entdeckt 1970) kann man den folgenden allgemeinen Satz nachweisen.

Satz 3.2.19

Jede Theorie, welche den abhängigen Funktionenraum $x{:}S{\to}T$ enthält und $\mathbb{U} \in \mathbb{U}$ zuläßt, ist inkonsistent.

Wir müssen also nach einem anderen Weg suchen, die Typeigenschaft des Typuniversums $\mathbb{U}$, um die wir aus den oben erwähnten Gründen nicht herumkommen, syntaktisch zu repräsentieren.

Die wohl einfachste Lösung für dieses Problem ergibt sich, wenn wir versuchen, die Natur des Universums $\mathbb{U}$ zu charakterisieren. Es ist gleichzeitig eine Menge von Typen und selbst ein Typ. Als Typ aber steht $\mathbb{U}$ in irgendeinem Sinne auf einer höheren Stufe als diejenigen Typen, die Elemente von $\mathbb{U}$ sind. Das Universum, zu dem $\mathbb{U}$ gehören muß, ist also nicht $\mathbb{U}$ selbst, sondern ein höher geartetes Universum, welches man zum Beispiel mit U_2 bezeichnen kann. $\mathbb{U}$ selbst ist also die Menge aller "kleinen" Typen und selbst ein "großer" Typ. Diesen Gedanken muß man natürlich weiterführen, denn auch U_2 muß wiederum ein Typ eines höheren Universums sein. So erhalten wir schließlich eine ganze Hierarchie von Universen $\mathbb{U}{=}\mathsf{U}_1$, U_2, U_3, ..., die das syntaktische Gegenstück zur Typeigenschaft bilden. Diese Universen dürfen allerdings nicht völlig getrennt voneinander erscheinen, wie das folgende Beispiel zeigt.

Beispiel 3.2.20

Im Beispiel 3.2.18 hatten wir erklärt, daß der Typ T im abhängigen Funktionenraum $x{:}S{\to}T$ im wesentlichen durch eine Funktion $\hat{T}$ gebildet werden kann, die vom Typ $S{\to}\mathbb{U}$ ist. Dabei ist S ein einfacher Typ aus U_1 und $\mathbb{U}$ ein großer Typ aus U_2.

Zu welchem Universum gehört nun $S{\to}\mathbb{U}$? Gemäß unseren bisherigen Regeln ist es das Universum, zu dem sowohl S als auch $\mathbb{U}$ gehören. Da S und $\mathbb{U}$ aber zu verschiedenen Universen gehören, müssen wir entweder die Funktioneraumbildung dahingehend abändern, daß wir die Universen der Teiltypen und des Gesamttyps als Indizes mitschleppen, oder zulassen, daß alle Elemente von U_1 auch Elemente von U_2 sind. Da die erste Alternative zu so komplizierten Konstrukten wie $S\overset{1,2}{\to}_2\mathbb{U}$ und einer Vielfalt nahezu identischer Regeln führen würde, erscheint die zweite Lösung sinnvoller.

Es ist also sinnvoll, die Hierarchie der Universen *kumulativ* zu gestalten: jedes Universum enthält das nächsttieferliegende Universum und alle seine Elemente.

$$\mathsf{U}_1 \overset{\in}{\subset} \mathsf{U}_2 \overset{\in}{\subset} \mathsf{U}_3 \overset{\in}{\subset} \ldots$$ [26]

[25] Als Beispiel seien die folgenden Typsimulationen genannt.

$x{:}S{\times}T$	$\equiv$	$\mathtt{X}{:}\mathbb{U} \to (x{:}S{\to}(T{\to}\mathtt{X})) \to \mathtt{X}$	$A \wedge B$	$\equiv$	$\forall\mathtt{P}{:}\mathbb{U}.(A \Rightarrow (B \Rightarrow \mathtt{P})) \Rightarrow \mathtt{P}$
$\forall x{:}T\,.\,P$	$\equiv$	$x{:}T{\to}P$	$A \vee B$	$\equiv$	$\forall\mathtt{P}{:}\mathbb{U}.\ ((A \Rightarrow \mathtt{P}) \Rightarrow (B \Rightarrow \mathtt{P})) \Rightarrow \mathtt{P}$
$\exists x{:}T\,.\,P$	$\equiv$	$x{:}T{\times}P$	Λ	$\equiv$	$\forall\mathtt{P}{:}\mathbb{U}.\mathtt{P}$
$A \Rightarrow B$	$\equiv$	$A{\to}B$	$\neg A$	$\equiv$	$A \Rightarrow \Lambda$
			$s{=}t \in T$	$\equiv$	$\forall\mathtt{P}{:}T{\to}\mathbb{U}.\ \mathtt{P}(s) \Rightarrow \mathtt{P}(t)$

Man kann relativ leicht überprüfen, daß die logischen Gesetze der simulierten Operationen sich tatsächlich aus denen des abhängigen Funktionenraumes ergeben. Auch die Probleme der natürlichen Zahlen innerhalb der einfachen Typentheorie lassen sich durch abhängige Funktionenraüme durch eine leichte Erweiterung der Church-Numerals lösen.

$\overline{n} \equiv \lambda\mathtt{X}.\lambda\mathtt{f}.\lambda\mathtt{x}.\ \mathtt{f}^n\,\mathtt{x}$	$\mathbb{N}$	$\equiv$	$\mathtt{X}{:}\mathbb{U} \to (\mathtt{X}{\to}\mathtt{X}) \to \mathtt{X}{\to}\mathtt{X}$
	$\mathbf{PRs}[base, h]$	$\equiv$	$\lambda\mathtt{n}.\ \mathtt{n}(\mathbb{N})\ h\ base$

Mit diesen Definitionen kann die primitive Rekursion problemlos und sinnvoll typisiert werden.

kanonisch (Typen)	kanonisch (Elemente)	nichtkanonisch
$\mathbf{U}\{j{:}\mathtt{l}\}()$	(alle kanonischen Typen)	
U_j		
$\mathbf{equal}\{\}(s;t;T)$	$\mathbf{Axiom}\{\}()$	
$s = t \in T$	Axiom	

Abbildung 3.7: Operatorentabelle für Universen und Gleichheit

Gemessen an ihrer Ausdruckskraft ist diese unendliche kumulative Universenhierarchie sehr ähnlich zu der Eigenschaft $\mathbb{U}{\in}\mathbb{U}$. Sie kann jedoch ihre Probleme vermeiden und ist somit für formale Systeme besser geeignet.

Es sei an dieser Stelle angemerkt, daß für die Universenhierarchie ein sehr wichtiges *Reduktionsprinzip* gilt, welches besagt, daß jedes Objekt eines höheren Typs durch ein Objekt aus $\mathbb{U}$ simuliert werden kann. Prinzipiell könnte man also auf die höheren Universen verzichten und immer die Einbettung in $\mathbb{U}$ vornehmen. Dies würde die Theorie jedoch unnötig verkomplizieren. Wir behalten daher die Universenhierarchie und werden uns die Möglichkeit einer Einbettung für eventuelle Selbstreflektion aufheben.

Entwurfsprinzip 3.2.21

Die Typeigenschaft wird dargestellt durch eine kumulative Hierarchie von Universen.

Um dieses Prinzip zu realisieren, müssen wir also unsere Syntax durch einen weiteren Eintrag in der Operatorentabelle (siehe Abbildung 3.7) ergänzen. Zugunsten einer größeren Flexibilität beim formalen Schließen lassen wir als Indizes neben den konstanten natürlichen Zahlen – welche zu kanonischen Universen führen – auch einfache arithmetische Ausdrücke zu, die aus Zahlen, Zahlenvariablen, Addition und Maximumbildung zusammengesetzt sind. Diese *level-expressions* bilden die erlaubten Parameter des Operators **U** in Abbildung 3.7. Spezielle nichtkanonische Formen für Universen gibt es nicht (und somit auch keine Einträge in die Redex–Kontrakta Tabelle).

Die Semantik der Universen (siehe Abbildung 3.8) ist einfach. Zwei Universen U_i und U_j sind als Typen gleich, wenn i und j als Zahlen gleich sind. Die kanonischen Elemente eines Universums sind genau die kanonischen Typen und somit ist die Gleichheit von Elementen aus U_j nahezu identisch mit der Typgleichheit. Die Kumulativität wird semantisch dadurch wiedergespiegelt, daß die Grundtypen wie $\mathbb{Z}$, aus denen jeder kanonische Term im Endeffekt aufgebaut wird, zu jedem Universum gehören werden.

3.2.3.2 Gleichheit als Proposition

Mithilfe der Universenhierarchie haben wir die syntaktische Repräsentation von Typ-Sein und Typgleichheit auf die Urteile Typzugehörigkeit und Elementgleichheit zurückgeführt. Wir wollen nun zeigen, wie wir auch für diese beiden Urteile eine einheitliche Darstellung finden können. Unsere Definition 3.2.15 der Semantik von Urteilen hat die Typzugehörigkeit mehr oder weniger als Spezialfall der Elementgleichheit charakterisiert. Dennoch gibt es Gründe, nicht die Elementgleichheit, sondern die Typzugehörigkeit als formalen Oberbegriff zu wählen und die semantische Gleichheit – ähnlich wie die Typeigenschaft – *innerhalb der Objektsprache* zu simulieren, also einen Gleichheitstyp einzuführen, welcher die semantischen Eigenschaften der Gleichheit syntaktisch wiederspiegelt.

In unserer Diskussion der Urteile hatten wir auf die Notwendigkeit des Gleichheits*urteils* hingewiesen. Unter allen elementaren logischen Aussagen ist die Gleichheit die wichtigste, da Schließen über Gleichheit in nahezu allen Teilen der Mathematik und Programmierung eine essentielle Rolle spielt. Was uns bisher jedoch fehlt, ist eine Möglichkeit, innerhalb von formalen Beweisen über Gleichheit zu *schließen*, da wir in

[26] Diese Idee ist eigentlich schon in der frühen Mathematik entstanden. In '*Principia Mathematicae*', dem Grundlagenbuch der modernen Mathematik [Whitehead & Russell, 1925], findet man eine hierarchische Typstruktur, wenn auch eine recht komplizierte. Sie wurde später durch Quine [Quine, 1963] vereinfacht, was zu einer kumulativen Theorie der Typen führte.

Typsemantik		
$s_1 = t_1 \in T_1 \;=\; s_2 = t_2 \in T_2$	falls	$T_1{=}T_2$ und $s_1{=}s_2 \in T_1$ und $t_1{=}t_2 \in T_1$.
$\mathsf{U}_{j_1} = \mathsf{U}_{j_2}$	falls	$j_1{=}j_2$ *(als natürliche Zahl)*
Elementsemantik		
$\mathsf{Axiom} = \mathsf{Axiom} \in s = t \in T$	falls	$s{=}t \in T$
$x_1{:}S_1{\to}T_1 = x_2{:}S_2{\to}T_2 \in \mathsf{U}_j$	falls	$S_1{=}S_2 \in \mathsf{U}_j$ und $T_1[s_1/x_1]{=}T_2[s_2/x_2] \in \mathsf{U}_j$ für alle Terme s_1, s_2 mit $s_1{=}s_2 \in S_1$.
$T = S_2{\to}T_2 \in \mathsf{U}_j$	falls	$T = x_2{:}S_2{\to}T_2 \in \mathsf{U}_j$ für ein beliebiges $x_2 \in \mathcal{V}$.
$S_1{\to}T_1 = T \in \mathsf{U}_j$	falls	$x_1{:}S_1{\to}T_1 = T \in \mathsf{U}_j$ für ein beliebiges $x_1 \in \mathcal{V}$.
$x_1{:}S_1{\times}T_1 = x_2{:}S_2{\times}T_2 \in \mathsf{U}_j$	falls	$S_1{=}S_2 \in \mathsf{U}_j$ und $T_1[s_1/x_1]{=}T_2[s_2/x_2] \in \mathsf{U}_j$ für alle Terme s_1, s_2 mit $s_1{=}s_2 \in S_1$.
$T = S_2{\times}T_2 \in \mathsf{U}_j$	falls	$T = x_2{:}S_2{\times}T_2 \in \mathsf{U}_j$ für ein beliebiges $x_2 \in \mathcal{V}$.
$S_1{\times}T_1 = T \in \mathsf{U}_j$	falls	$x_1{:}S_1{\times}T_1 = T \in \mathsf{U}_j$ für ein beliebiges $x_1 \in \mathcal{V}$.
$S_1{+}T_1 = S_2{+}T_2 \in \mathsf{U}_j$	falls	$S_1{=}S_2 \in \mathsf{U}_j$ und $T_1{=}T_2 \in \mathsf{U}_j$.
$s_1 = t_1 \in T_1 \;=\; s_2 = t_2 \in T_2 \;\in \mathsf{U}_j$	falls	$T_1{=}T_2 \in \mathsf{U}_j$ und $s_1{=}s_2 \in T_1$ und $t_1{=}t_2 \in T_1$.
$\mathsf{U}_{j_1} = \mathsf{U}_{j_2} \in \mathsf{U}_j$	falls	$j_1{=}j_2{<}j$ *(als natürliche Zahl)*

Abbildung 3.8: Semantik der Urteile für Universen und Gleichheit

den Hypothesen einer Sequenz bisher nur (Typ-)Ausdrücke, aber keine Urteile zugelassen haben. Wir stehen daher vor der Wahl, entweder die Struktur der Sequenzen zu erweitern und Urteile und Typausdrücke simultan in den Hypothesen zu verwalten, oder einfach nur die formale Sprache um einen Gleichheitstyp zu ergänzen.

Die Entscheidung für den ersten Weg würde eine Menge formaler Komplikationen mit sich bringen. Der zweite Weg dagegen wirft philosophische Probleme auf: *Gleichheit ist eine logische Aussage und nicht etwa ein Typ.* Konzeptuell ist dieser Unterschied in der Tat von großer Bedeutung. Man sollte allerdings im Auge behalten, daß die Typentheorie das logische Schließen so formalisieren soll, daß es von Computern verarbeitet werden kann. Der Unterschied zwischen Typen und logischen Aussagen verschwindet also spätestens auf der Ebene der Verarbeitung der Symbole, die zur Repräsentation verwendet werden. Somit ist er für die *formale* Theorie von untergeordneter Bedeutung und da eine formale Trennung einen erheblichen Mehraufwand mit sich bringen würde, entscheiden wir uns dafür, logische Aussagen bereits *innerhalb der Theorie* durch Typen zu simulieren. Diese Vorgehensweise, deren Berechtigung durch die Curry-Howard Isomorphie gestützt wird, die wir in Abschnitt 3.3 genauer besprechen, wird in der Literatur als Prinzip der Propositionen als Typen (*propositions as types principle*) bezeichnet

Entwurfsprinzip 3.2.22 (Propositionen als Typen)

In der Typentheorie werden logische Aussagen dargestellt durch Typen, deren Elemente den Beweisen der Aussagen entsprechen.

Für die Darstellung der Gleichheit ergänzen wir unsere formale Sprache also um einen weiteren vordefinierten Term $\mathbf{equal}\{\}(s;t;T)$ (siehe Abbildung 3.7), der das Gleichheitsurteil $s{=}t \in T$ syntaktisch simuliert. In seiner Display Form werden wir diesem Term daher dieselbe Gestalt $s = t \in T$ geben.[27] Das kanonische Element dieses Typs wird mit Axiom bezeichnet. Alle Beweise (Elemente des Typs) werden im Endeffekt auf Axiom reduzieren und somit Evidenz liefern, daß das Urteil $s{=}t \in T$ tatsächlich Gültigkeit hat. In der Semantiktabelle 3.8 stellen wir genau dieser Zusammenhang zwischen kanonischen Elementen des *Typs* $s = t \in T$ und dem *Urteil* $s{=}t \in T$ her. Auch für die Gleichheit gibt es keine speziellen nichtkanonischen Ausdrücke und dementsprechend keine Einträge in der Redex–Kontrakta Tabelle.

Durch die Verwendung der Gleichheit als Datentyp sind wir nun in der Lage, alle vier Arten von Urteilen durch ein einziges Urteil, nämlich die Typzugehörigkeit, zu simulieren und Gleichheiten auch in den Hypothesen eines Beweises zu verwenden. Welche weiteren Vorteile die Wahl des Typzugehörigkeitsurteils als zentrales Beweiskonzept anstelle der Elementgleichheit mit sich bringt, zeigt der nun folgende Abschnitt.

[27] Man achte auf die feinen Unterschiede in den Zeichensätzen.

3.2.3.3 Entwicklung von Beweisen statt Überprüfung

In unserer Diskussion der Anforderungen an einen praktisch verwendbaren Kalkül im Abschnitt 3.1.3 hatten wir bereits darauf hingewiesen, daß wir Kalküle zum *Entwickeln* formaler Beweise einsetzen wollen und nicht etwa nur zum Überprüfen eines bereits gefundenen Beweises. Da wir nun gemäß dem Prinzip "Propositionen als Typen" logische Aussagen als Typen repräsentieren, deren Elemente genau die Beweise dieser Aussagen sind, ist eine *direkte* Darstellung des Typzugehörigkeitsurteils durch ein syntaktisches Konstrukt der Gestalt $t \in T$ (wobei t und T Ausdrücke sind) wenig geeignet, um dieses Ziel zu erreichen. Wir wollen dies an einem Beispiel erläutern.

Beispiel 3.2.23

Eine direkte syntaktische Darstellung des Urteils $s{=}t \in T$ müßte entsprechend der im vorigen Abschnitt beschriebenen Methodik die Gestalt

$$p \in s = t \in T$$

haben, wobei p ein Term ist, der im Endeffekt zu Axiom reduzieren muß. Dieser Term hängt aber von dem Beweis ab, der geführt werden muß, um diese konkrete Gleichheit nachzuweisen. Wir müßten also bereits *im voraus* wissen, wie $s{=}t \in T$ zu beweisen ist, um p angeben zu können. Dann aber ist der Kalkül von geringem Nutzen, da wir nur noch überprüfen können, ob unsere Beweisführung stimmt.

Das Problem wird noch größer, wenn wir bedenken, wie viele logische Regeln wir beim Beweis einer logischen Aussage verwenden können. Wir wären gezwungen, den Beweis zunächst von Hand zu führen, aus dem vollständigen Beweis einen Beweisterm zusammenzusetzen und danach den Beweis zu überprüfen, indem wir ihn anhand des Beweisterms noch einmal durchgehen. Die Rechnerunterstützung bei der Beweisführung würde dadurch ihren Sinn verlieren, da sie keinerlei Vorteile gegenüber der Handarbeit mit sich brächte.

Der Sinn einer automatisierbaren formalen Logik ist jedoch nicht, die Korrektheit *bestimmter* Beweise nachzuweisen, sondern ein Hilfsmittel dafür zu bieten, die Gültigkeit einer *Aussage* zu beweisen. Mit anderen Worten: wir wollen den Beweisterm aufbauen *während* wir den Beweis entwickeln und nicht etwa vorher. Um dies zu erreichen, müssen wir den formalen Kalkül, mit dem wir Beweise für Urteile syntaktisch simulieren, so gestalten, daß er die *Entwicklung* eines Terms, der zu einem vorgegebenen Typ gehören soll, unterstützt und nicht nur die Korrektheit eines gegebenen Typzugehörigkeitsurteils nachzuweisen hilft. Dies verlangt eine andere Form der Darstellung eines Urteils. Anstelle von $p \in s = t \in T$ müssen wir eine Form finden, die es erlaubt, p erst dann niederzuschreiben, wenn der Beweis beendet ist und den Term in unvollständigen Beweisen einfach offenzulassen. Formal werden wir dies durch die Notation

$$s = t \in T \quad \lfloor\mathsf{ext}\ p\rfloor$$

kennzeichnen, die inhaltlich dasselbe Urteil repräsentiert wie zuvor, aber durch die Notation $\lfloor\mathsf{ext}\ p\rfloor$ andeutet, daß wir den Beweisterm p zu verstecken gedenken und aus einem vollständigen Beweis *extrahieren* können. Dies hat keinerlei Einfluß auf die Semantik der Theorie und die theoretische Bedeutung der Inferenzregeln, bewirkt aber einen gravierenden praktischen Unterschied für das Arbeiten mit diesen Regeln innerhalb eines interaktiven Beweissystems für die Typentheorie. Die Anwendung einer Regel scheint nämlich zunächst einmal nur den Typ selbst zu betreffen, dessen Element wir innerhalb des Beweises konstruieren wollen. Daß diese Regel implizit natürlich auch sagt, was der Zusammenhang zwischen den "*Extrakt-Termen*" der Teilziele und dem des ursprünglichen Beweisziels ist, spielt zunächst erst einmal keine Rolle. Erst, wenn der Beweis beendet ist, spielt diese Information eine Rolle, denn sie kann dazu verwendet werden, die jeweiligen Extrakt-Terme automatisch zu konstruieren. Die Entscheidung, einen Kalkül zur interaktiven Entwicklung von Beweisen zu entwerfen, führt also zu dem folgenden Gestaltungsmerkmal der intuitionistischen Typentheorie von NuPRL.

Entwurfsprinzip 3.2.24 (Implizite Darstellung von Elementen)

Urteile der Typentheorie werden dargestellt durch Konklusionen der Form $\underline{T\ \lfloor\mathsf{ext}\ p\rfloor}$.

Es sei angemerkt, daß wir durch diese Entwurfsentscheidung, die den Kalkül von NuPRL deutlich von den meisten anderen Formulierungen der Typentheorie abhebt, keineswegs die Möglichkeit zu einem expliziten Schließen über Elemente eines gegebenen Typs verlieren. Der Datentyp der typabhängigen Gleichheit, den wir oben eingeführt haben, kann genau für diesen Zweck genutzt werden. Somit bietet uns der Kalkül insgesamt sechs grundsätzliche Arten des formalen Schließens innerhalb eines einheitlichen Formalismus.

- Explizites Schließen (Überprüfung eines Urteils):
  - *Typ-Sein* "T Typ": Zeige die Existenz eines Terms p und eines Levels j mit $p \in T = T \in \mathsf{U}_j$.
  - *Typgleichheit* "$S{=}T$": Zeige die Existenz eines Terms p und eines Levels j mit $p \in S = T \in \mathsf{U}_j$.
  - *Typzugehörigkeit* "$t \in T$": Zeige die Existenz eines Terms p mit $p \in t = t \in T$.
  - *Elementgleichheit* "$s{=}t \in T$": Zeige die Existenz eines Terms p mit $p \in s = t \in T$.

  Die zugehörigen Beweisziele lauten $\underline{\vdash T = T \in \mathsf{U}_j}$, $\underline{\vdash S = T \in \mathsf{U}_j}$, $\underline{\vdash t = t \in T}$ bzw. $\underline{\vdash s = t \in T}$.
- Implizites Schließen (Konstruktion von Termen, die ein Urteil erfüllen):
  - *Konstruktion von Datentypen*: Zeige die Existenz eines Terms T mit $T \in \mathsf{U}_j$.
  - *Konstruktion von Elementen* eines Typs T: Zeige die Existenz eines die Terms t mit $t \in T$.
    Diese Form wird benutzt um logische Beweise zu führen (Konstruktion von Beweistermen) und vor allem auch, um Algorithmen zu konstruieren, von denen man nur die Spezifikation gegeben hat.

  Die entsprechenden Beweisziele sind $\underline{\vdash \mathsf{U}_j}$ bzw. $\underline{\vdash T}$.

3.2.3.4 Sequenzen, Regeln und formale Beweise

Nach all diesen Vorüberlegungen sind wir nun in der Lage, präzisen Definition der im Beweiskalkül von NuPRL relevanten Konzepte zu geben. Als syntaktische Repräsentation von hypothetischen Urteilen bilden *Sequenzen* die Grundkonstrukte formaler Beweise. Ihre Gestalt ergibt sich aus dem Entwurfsprinzip 3.2.24.

Definition 3.2.25 (Sequenzen)

1. *Eine* *__Deklaration__* *hat die Gestalt* $x{:}T$*, wobei* x *eine Variable und* T *ein Term ist.*
2. *Eine* *__Hypothesenliste__* *ist eine Liste* $\Gamma = x_1{:}T_1, \ldots, x_n{:}T_n$ *von durch Komma getrennten Deklarationen.*
3. *Eine* *__Konklusion__* *ist ein Term im Sinne von Definition 3.2.5.*
4. *Eine* *__Sequenz__* *(oder* *__Beweisziel__**) hat die Gestalt* $\Gamma \vdash C$*, wobei* Γ *eine Hypothesenliste und* C *eine Konklusion ist.*
5. *Eine Sequenz* $Z = x_1{:}T_1, \ldots, x_n{:}T_n \vdash C$ [28] *ist* *__geschlossen__**, wenn jede in* C *oder einem der* T_i *frei vorkommende Variable* x *zuvor durch* $x{:}T_j$ $(j < i)$ *deklariert wurde.*

   Eine Sequenz ist *__rein__**, wenn sie geschlossen ist und jede Variable nur einmal deklariert wurde.*
6. *Eine (reine) Sequenz* $\Gamma \vdash C$ *ist* *__gültig__**, wenn es einen Term t gibt, für den* $\Gamma \vdash t \in C$ *ein hypothetisches Urteil ist. Ist der Term t bekannt, der die Sequenz gültig macht, so schreiben wir* $\underline{\Gamma \vdash C\ \lfloor\mathsf{ext}\ t\rfloor}$.

Eine Sequenz hat also insgesamt die Form $\underline{x_1{:}T_1,\ldots,x_n{:}T_n \vdash C}$, wobei die x_i Variablen sind und C sowie die T_i Terme. Eine solche Sequenz ist zu lesen als die *Behauptung*

Unter der Annahme, daß x_i *Variablen vom Typ* T_i *sind, kann ein Element des Typs* C *konstruiert werden.*

[28] Um Verwechslungen mit Typen zu vermeden, die wir mit dem Symbol S kennzeichnen, verwenden wir für Sequenzen das Symbol Z, welches die Anwendung als Beweis<u>z</u>iel heraushebt.

In der textlichen Erscheinungsform sind Sequenzen also sehr ähnlich zu den hypothetischen Urteilen, die zur Erklärung der Gültigkeit von Sequenzen herangezogen werden (und nun einmal durch irgendeine Form von Text aufgeschrieben werden müssen). Man beachte jedoch, daß hypothetische Urteile semantische Konzepte sind, die gemäß Definition 3.2.15 einen Sachverhalt als wahr "beurteilen". Im Gegensatz dazu ist eine Sequenz nur die syntaktische Repräsentation einer Behauptung, die auch ungültig – insbesondere also auch unbeweisbar – sein kann.

Es sei angemerkt, daß man auch bei der Definitionen von Sequenzen die Sprache der Terme aus Definition 3.2.5 verwenden und zum Beispiel $\mathsf{declaration}\{\}(x.T)$ oder $\mathsf{sequent}\{\}(\Gamma;C)$ schreiben könnte. Dies würde die Möglichkeit eröffnen, Sequenzen durch Terme der Objektsprache auszudrücken und somit innerhalb der Typentheorie über den eigenen Beweismechanismus formal zu schließen. Eine solche *Selbstreflektion* ist durchaus wünschenswert, würde zum gegenwärtigen Zeitpunkt jedoch zu weit führen. Wir beschränken uns daher auf eine textliche Definition der Beweiskonzepte und werden das Thema der Reflektion erst in späteren Kapiteln wieder aufgreifen.

Die äußere Form, die wir für Sequenzen gewählt haben, hat auch Auswirkungen auf die Struktur der Regeln. Während eine Inferenzregel bisher nur zur Zerlegung von Beweiszielen in Unterziele verwandt wurde, macht die implizite Verwaltung von Extrakt-Termen es nötig, daß die Regeln nun zwei Aufgaben übernehmen: Sie zerlegen Ziele in Teilziele und müssen gleichzeitig angeben, wie Extrakt-Terme der Teilziele – also die Evidenzen für ihre Gültigkeit – zu einem Extrakt-Term des ursprünglichen Ziels zusammengesetzt werden können. Diese beiden Teilaufgaben nennt man *Dekomposition* und *Validierung*.

Definition 3.2.26 (Regeln)

1. *Eine Dekomposition ist eine Abbildung, welche eine Sequenz – das Beweisziel – in eine endliche Liste (möglicherweise leere) von Sequenzen – die Unter- oder Teilziele – abbildet.*

2. *Eine Validierung ist eine Abbildung, welche eine endliche Liste von (Paaren von) Sequenzen und Termen in einen Term[29] abbildet.*

3. *Eine Inferenzregel hat die Gestalt (dec,val), wobei dec eine Dekomposition und val eine Validierung ist.*

4. *Eine Inferenzregel r = (dec,val) ist korrekt, wenn für jede reine Sequenz $Z = \Gamma \vdash C$ gilt:*
   - *Alle durch Anwendung von dec erzeugten Teilziele $Z_1 = \Gamma_1 \vdash C_1, \ldots, Z_n = \Gamma_n \vdash C_n$ sind rein.*
   - *Aus der Gültigkeit der Teilziele $Z_1,\ldots Z_n$ folgt die Gültigkeit des Hauptzieles Z.*
   - *Gilt $\Gamma_i \vdash C_i$ $\lfloor$**ext** $t_i\rfloor$ für alle Teilziele Z_i, so folgt $\Gamma \vdash C$ $\lfloor$**ext** $t\rfloor$, wobei t derjenige Term ist, der durch Anwendung von val auf $[(Z_1,t_1),\ldots,(Z_n t_n)]$ entsteht.*

Eine korrekte Inferenzregel zerlegt also ein Beweisziel so, daß seine Korrektheit aus der Korrektheit der Teilziele folgt, wobei die Validierungsfunktion die Evidenzen für die Gültigkeit der Ziele entsprechend zusammensetzen kann. Damit ist es möglich, in einem Beweis eine Initialsequenz durch Anwendung von Inferenzregeln schrittweise zu verfeinern und nach Abschluß des Beweises den Term, welcher die Sequenz gültig macht, automatisch – nämlich durch Zusammensetzen der Validierungen – aus dem Beweis zu extrahieren. Aus diesem Grunde wird dieser Term auch als Extrakt-Term des Beweises bzw. der Sequenz bezeichnet.

Formale Beweise sind Bäume, deren Knoten mit Sequenzen und Regeln markiert sind, wobei die Sequenzen der Nachfolger eines Knotens genau die Teilziele sind, welche sich durch Anwendung der Regel auf die Knotensequenz ergeben. Unvollständige Beweise enthalten Blätter, die nur mit einer Sequenz markiert – also *unverfeinert* – sind. Die Blätter eines vollständigen Beweises müssen notwendigerweise Regeln enthalten, welche keine Teilziele erzeugen.

[29]Um einen Extrakt-Term einer Sequenz zu bilden benötigt eine Validierung also eine neben den Extrakt-Termen der Teilziele auch die Teilziele selbst, da in deren Annahmen zuweilen notwendige Komponenten enthalten sind. Da die Sequenz zusammen mit ihrem Extrakt-Term alle Informationen des eigentlichen Beweises enthält, werden im NuPRL-System Validierungen der Einfachheit als Abbildungen dargestellt, die Listen von Beweisen in Beweise abbilden können (siehe Abschnitt 4.1.2).

Definition 3.2.27 (Beweise und Extrakt-Terme)

1. *Beweise sind induktiv wie folgt definiert*
   - *Jede Sequenz $Z = \Gamma \vdash C$ ist ein unvollständiger Beweis mit Wurzel Z*
   - *Ist $Z = \Gamma \vdash C$ eine Sequenz, r = (dec,val) eine korrekte Inferenzregel und sind $\pi_1,\ldots,\pi_n$ ($n{\geq}0$) Beweise, deren Wurzeln die durch Anwendung von dec auf Z erzeugten Teilziele sind, so ist das Tupel $(Z,\ r,\ [\pi_1,\ldots,\pi_n])$ ein Beweis mit Wurzel Z.*

2. *Ein Beweis ist vollständig, wenn er keine unvollständigen Teilbeweise enthält.*

3. *Der Extrakt-Term $\mathtt{EXT}(\pi)$ eines vollständigen Beweises $\pi = (Z,\ (dec,val),\ [\pi_1,\ldots,\pi_n])$ rekursiv definiert als $val([\,(Z_1,\mathtt{EXT}(\pi_1)),\ldots,(Z_n,\mathtt{EXT}(\pi_n))\,])$, wobei die Z_i die Wurzeln der Beweise π_i sind.*

4. *Eine Initialsequenz ist eine geschlossene Sequenz, deren Hypothesenliste leer ist.*

5. *Ein Theorem ist ein vollständiger Beweis, dessen Wurzel eine Initialsequenz ist.*

Die rekursive Definition von Extrakt-Termen schließt den Fall mit ein, daß ein Beweis nach Anwendung der Regeln keine weiteren Unterziele mehr enthält und die Validierungsfunktion entsprechend einen Extrakt-Term aus einer leeren Liste erzeugt. Daher ist eine Fallunterscheidung in der Definition überflüssig. Da die Verwendung abhängiger Datentypen eine (vollständige) automatische Typüberprüfung unmöglich macht, müssen die Initialsequenzen von Beweisen eine leere Hypothesenliste besitzen. Auf diese Art kann durch das Regelsystem sichergestellt werden, daß in den Unterzielen jede Deklaration auf der rechten Seite nur Typen enthält.

Das folgende Lemma zeigt, daß die obigen Definitionen tatsächlich zu Beweisen führen, welche die durch Urteile gegebene Semantik respektieren. Damit ist sichergestellt, daß jeder Beweis im Sinne der Semantik korrekt ist und dem intuitiven Verständnis der dargestellten Sequenzen entspricht. Es sei jedoch angemerkt, daß wegen der reichhaltigen Ausdruckskraft der Theorie nicht alles, was semantisch gültig ist, auch formal beweisbar sein kann.

Lemma 3.2.28

Es sei π ein Theorem mit Wurzel $\vdash T$. Dann gilt

1. *T ist ein Typ (d.h. es gilt das Urteil 'T Typ').*
2. *Für $t := \mathtt{EXT}(\pi)$ ist $t \in T$ ein Urteil*
3. *Für jeden Teilbeweis π' von π mit Wurzel $Z = x_1{:}T_1,\ldots,x_n{:}T_n \vdash C$ gilt:*
   – *Z ist eine reine Sequenz.*
   – *C und alle T_i sind Typen.*
   – *$\Gamma \vdash \mathtt{EXT}(\pi') \in C$ ist ein hypothetisches Urteil.*
   – *Z ist eine gültige Sequenz.*

Auch bei der Definitionen von Beweisen, Extrakttermen und Regeln könnte man die Sprache der Terme aus Definition 3.2.5 verwenden und zum Beispiel $\mathsf{unrefined}\{\}(Z)$ und $\mathsf{refined}\{\}(Z,\ r,\ [\pi_1,\ldots,\pi_n])$ als Notation verwenden. Damit ist eine Selbstreflektion innerhalb der Typentheorie tatsächlich durchführbar.[30]

3.2.3.5 Das Regelsystem im Detail

Die Beweisregeln für die einzelnen Konstrukte der formalen Theorie sind so anzulegen, daß sie die durch Urteile gegebene Semantik – also die Einträge der Typsemantiktabelle und der Elementsemantiktabelle – respektieren. Wir werden sie im folgenden gruppenweise entsprechend dem Typkonstruktor auflisten. Jede dieser Gruppen wird normalerweise vier Arten von Regeln enthalten, die wir am Beispiel der Regeln für den Funktionenraum (siehe Abbildung 3.9 auf Seite 122) erläutern wollen.

[30]Die gegenwärtige Konzeption des NuPRL Systems unterstützt diese Selbstreflektion, da Regeln und viele andere Metakonzepte – wie zum Beispiel der Editor – durch explizite Objekte der Theorie in der Syntax der Terme beschrieben werden.

1. *Formationsregeln* legen fest, wie ein Typ aus anderen Typen zusammengesetzt wird. Sie unterstützen das Schließen über Typgleichheit und die Eigenschaft, ein Typ zu sein, und könnten als Regeln über die kanonischen Elemente der Universen betrachtet werden. Die hier betrachteten Beweisziele haben die Form $\Gamma \vdash S = T \in U_j$, wobei S und T kanonische Typausdrücke sind (, die gleich sind, wenn über Typ-Sein geschlossen werden soll) und der Regelname folgt dem Muster <u>*typ*`Eq`</u>, wobei *typ* der Name des Typs ist.[31] Für den Funktionenraum ist dies zum Beispiel die folgende Regel, die genau der in Beispiel 3.2.16 auf Seite 111 erklärten Typgleichheit von Funktionenräumen entspricht.

   $\Gamma \vdash x_1{:}S_1{\to}T_1 = x_2{:}S_2{\to}T_2 \in U_j$ ⌊Ax⌋
   by `functionEq`
   $\Gamma \vdash S_1 = S_2 \in U_j$ ⌊Ax⌋
   $\Gamma,\ x{:}S_1 \vdash T_1[x/x_1] = T_2[x/x_2] \in U_j$ ⌊Ax⌋

2. *Kanonische* Regeln erklären, wie die kanonischen Elemente eines Typs zusammengesetzt werden und wann sie gleich sind. Die Beweisziele haben die Gestalt $\Gamma \vdash s = t \in T$ bzw. $\Gamma \vdash T$ ⌊**ext** t⌋, wobei T ein kanonischer Typausdruck ist und s bzw. t die zugehörigen kanonischen Elementausdrücke. Die Regeln haben üblicherweise den Namen <u>*element*`Eq`</u> bzw. <u>*element*`I`</u>, wobei *element* der Name des kanonischen Elements ist. Als Beispiel seien hier die kanonischen Regeln des Funktionenraums genannt.

   $\Gamma \vdash \lambda x_1.t_1 = \lambda x_2.t_2 \in x{:}S{\to}T$ ⌊Ax⌋
   by `lambdaEq` j
   $\Gamma,\ x'{:}S \vdash t_1[x'/x_1] = t_2[x'/x_2] \in T[x'/x]$ ⌊Ax⌋
   $\Gamma \vdash S \in U_j$ ⌊Ax⌋

   $\Gamma \vdash x{:}S{\to}T$ ⌊**ext** $\lambda x'.t$⌋
   by `lambdaI` j
   $\Gamma,\ x'{:}S \vdash T[x'/x]$ ⌊**ext** t⌋
   $\Gamma \vdash S \in U_j$ ⌊Ax⌋

3. *Nichtkanonische* Regeln erklären, wann ein nichtkanonischer Term zu einem vorgegebenen Typ gehört. Die hier betrachteten Beweisziele haben die Gestalt $\Gamma \vdash s = t \in T$ bzw. $\Gamma,\ x{:}S,\Delta \vdash T$ ⌊**ext** t⌋, wobei T ein beliebiger Typausdruck ist, S ein kanonischer Typausdruck, x eine Variable und s bzw. t nichtkanonische Terme sind. Im ersten Falle erklärt die Regel (<u>*nichtkanonisch*`Eq`</u>), wie die nichtkanonischen Ausdrücke zu zerlegen sind, während im zweiten Fall (<u>*typ*`E`</u>) die in der Hypothesenliste deklarierte Variable x des zugehörigen kanonischen Typs S ‘eliminiert’ wird, um den nichtkanonischen Extrakt-Term zu generieren. Die nichtkanonischen Regeln des Funktionenraums lauten

   $\Gamma \vdash f_1\,t_1 = f_2\,t_2 \in T[t_1/x]$ ⌊Ax⌋
   by `applyEq` $x{:}S{\to}T$
   $\Gamma \vdash f_1 = f_2 \in x{:}S{\to}T$ ⌊Ax⌋
   $\Gamma \vdash t_1 = t_2 \in S$ ⌊Ax⌋

   $\Gamma,\ f{:}\ x{:}S{\to}T,\ \Delta \vdash C$ ⌊**ext** $t[f\,s, \mathsf{Axiom}/y,z]$⌋
   by `functionE` $i\ s$
   $\Gamma,\ f{:}\ x{:}S{\to}T,\ \Delta \vdash s \in S$ ⌊Ax⌋
   $\Gamma,\ f{:}\ x{:}S{\to}T,\ y{:}T[s/x],$
   $\quad z{:}\ y = f\,s \in T[s/x],\ \Delta \vdash C$ ⌊**ext** t⌋

   Die Eliminationsregel `functionE` muß $y = f\,s \in T[s/x]$ zu den Hypothesen hinzunehmen, um einen Bezug zwischen der neuen Variablen y und möglichen Vorkommen von f und s in Δ und C herzustellen.

4. Die bisherigen Regeln erlauben nur die Untersuchung einer *strukturellen Gleichheit* von Termen, bei der ausschließlich die äußere Form von Bedeutung ist. Nicht möglich ist dagegen ein Vergleich von Termen, die strukturell verschieden aber semantisch gleich sind. Aus diesem Grunde ist es nötig, Regeln mit aufzunehmen, welche den Auswertungsmechanismus widerspiegeln.

   Berechnungsregeln (<u>*nichtkanonisch*`Red`</u>) stellen den Bezug zwischen kanonischen und nichtkanonischen Ausdrücken eines Datentyps her. Sie sind anwendbar auf Beweisziele der Form $\Gamma \vdash s = t \in T$, wobei T ein beliebiger Typausdruck ist und einer der beiden Terme s und t ein Redex ist, das durch Anwendung der Regel zu seinem Kontraktum reduziert wird.[32] Für den Funktionenraum ist dies die folgende Regel:

[31] Es gibt auch entsprechende Regeln zu den impliziten Beweiszielen der Form $\Gamma \vdash U_j$ ⌊**ext** T⌋. Diese werden aber praktisch überhaupt nicht mehr benutzt und daher im folgenden nicht aufgeführt.

[32] Diese typspezifischen Reduktionsregeln sind innerhalb von NuPRL eigentlich redundant, da zugunsten einer einheitlicheren Beweisführung sogenannte ‘***direct computation rules***’ eingeführt wurden, die es ermöglichen, den allgemeinen Auswertungsalgorithmus aus Abbildung 3.3 (Seite 108) auf jeden beliebigen Teilterm eines Ausdrucks bei gleichzeitiger Kontrolle der Anzahl der Reduktionsschritte anzuwenden. Die Details dieser typunabhängigen Berechnungsregeln werden wir in Abschnitt 3.6 besprechen.

$\Gamma \vdash x_1{:}S_1{\to}T_1 = x_2{:}S_2{\to}T_2 \in U_j$ ⌊Ax⌋
by `functionEq`
$\Gamma \vdash S_1 = S_2 \in U_j$ ⌊Ax⌋
$\Gamma,\ x{:}S_1 \vdash T_1[x/x_1] = T_2[x/x_2] \in U_j$ ⌊Ax⌋

$\Gamma \vdash \lambda x_1.t_1 = \lambda x_2.t_2 \in x{:}S{\to}T$ ⌊Ax⌋
by `lambdaEq` j
$\Gamma,\ x'{:}S \vdash t_1[x'/x_1] = t_2[x'/x_2] \in T[x'/x]$ ⌊Ax⌋
$\Gamma \vdash S \in U_j$ ⌊Ax⌋

$\Gamma \vdash x{:}S{\to}T$ ⌊**ext** $\lambda x'.t$⌋
by `lambdaI` j
$\Gamma,\ x'{:}S \vdash T[x'/x]$ ⌊**ext** t⌋
$\Gamma \vdash S \in U_j$ ⌊Ax⌋

$\Gamma \vdash f_1\,t_1 = f_2\,t_2 \in T[t_1/x]$ ⌊Ax⌋
by `applyEq` $x{:}S{\to}T$
$\Gamma \vdash f_1 = f_2 \in x{:}S{\to}T$ ⌊Ax⌋
$\Gamma \vdash t_1 = t_2 \in S$ ⌊Ax⌋

$\Gamma,\ f{:}\ x{:}S{\to}T,\ \Delta \vdash C$ ⌊**ext** $t[f\,s, \mathsf{Axiom}\,/\,y, v]$⌋
by `functionE` $i\ s$
$\Gamma,\ f{:}\ x{:}S{\to}T,\ \Delta \vdash s \in S$ ⌊Ax⌋
$\Gamma,\ f{:}\ x{:}S{\to}T,\ y{:}T[s/x],$
$\quad v{:}\ y = f\,s \in T[s/x],\ \Delta \vdash C$ ⌊**ext** t⌋

$\Gamma \vdash (\lambda x.t)\,s = t_2 \in T$ ⌊Ax⌋
by `applyRed`
$\Gamma \vdash t[s/x] = t_2 \in T$ ⌊Ax⌋

$\Gamma \vdash f_1 = f_2 \in x{:}S{\to}T$ ⌊**ext** t⌋
by `functionExt` $j\ x_1{:}S_1{\to}T_1\ x_2{:}S_2{\to}T_2$
$\Gamma,\ x'{:}S \vdash f_1\,x' = f_2\,x' \in T[x'/x]$ ⌊**ext** t⌋
$\Gamma \vdash S \in U_j$ ⌊Ax⌋
$\Gamma \vdash f_1 \in x_1{:}S_1{\to}T_1$ ⌊Ax⌋
$\Gamma \vdash f_2 \in x_2{:}S_2{\to}T_2$ ⌊Ax⌋

$\Gamma,\ f{:}\ S{\to}T,\ \Delta \vdash C$ ⌊**ext** $t[f\,s,/y]$⌋
by `functionE_indep` i
$\Gamma,\ f{:}\ S{\to}T,\ \Delta \vdash S$ ⌊**ext** s⌋
$\Gamma,\ f{:}\ S{\to}T,\ y{:}T,\ \Delta \vdash C$ ⌊**ext** t⌋

Abbildung 3.9: Regeln für Funktionenraum

$\Gamma \vdash (\lambda x.t)\,s = t_2 \in T$ ⌊Ax⌋
by `applyRed`
$\Gamma \vdash t[s/x] = t_2 \in T$ ⌊Ax⌋

– In manchen Gruppen kommen noch *besondere Regeln* zu diesen vier Regelarten hinzu. So gilt im Falle des Funktionenraumes die Gleichheit zweier Funktionen nicht nur, wenn die Regel `lambdaEq` anwendbar ist, sondern auch dann, wenn die Funktionen sich auf allen Argumenten gleich verhalten. Diese *Extensionalität* ist eine besondere Eigenschaft von Funktionen, die deutlich macht, daß nicht die innere Struktur, sondern das äußere Verhalten das wesentliche Charakteristikum einer Funktion ist. Die formale Regel berücksichtigt auch, daß die beiden genannten Funktionen zu völlig verschiedenen Funktionenräumen gehören können, die eine “Obermenge” des genannten Funktionenraums $x{:}S{\to}T$ sind.

$\Gamma \vdash f_1 = f_2 \in x{:}S{\to}T$ ⌊**ext** t⌋
by `functionExt` $j\ x_1{:}S_1{\to}T_1\ x_2{:}S_2{\to}T_2$
$\Gamma,\ x'{:}S \vdash f_1\,x' = f_2\,x' \in T[x'/x]$ ⌊**ext** t⌋
$\Gamma \vdash S \in U_j$ ⌊Ax⌋
$\Gamma \vdash f_1 \in x_1{:}S_1{\to}T_1$ ⌊Ax⌋
$\Gamma \vdash f_2 \in x_2{:}S_2{\to}T_2$ ⌊Ax⌋

$\Gamma,\ f{:}\ S{\to}T,\ \Delta \vdash C$ ⌊**ext** $t[f\,s,/y]$⌋
by `functionE_indep` i
$\Gamma,\ f{:}\ S{\to}T,\ \Delta \vdash S$ ⌊**ext** s⌋
$\Gamma,\ f{:}\ S{\to}T,\ y{:}T,\ \Delta \vdash C$ ⌊**ext** t⌋

Hinzu kommt eine besondere Behandlung des unabhängigen Funktionenraums $S{\to}T$, der als Spezialfall eines abhängigen Funktionenraums gesehen werden kann, bei dem die Bindungsvariable keine Rolle spielt. Die meisten Regeln des unabhängigen Funktionenraums sind direkte Spezialisierungen der obengenannten Regeln. Bei der Eliminationsregel ergibt sich allerdings eine so starke Vereinfachung, daß hierfür eine gesonderte Regel existiert, bei der ein Anwender einen Parameter weniger angeben muß.

$\Gamma \vdash x_1{:}S_1{\times}T_1 = x_2{:}S_2{\times}T_2 \in \mathsf{U}_j$ ⌊Ax⌋
by productEq
$\Gamma \vdash S_1 = S_2 \in \mathsf{U}_j$ ⌊Ax⌋
$\Gamma,\ x'{:}S_1 \vdash T_1[x'/x_1] = T_2[x'/x_2] \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma \vdash \langle s_1,t_1\rangle = \langle s_2,t_2\rangle \in x{:}S{\times}T$ ⌊Ax⌋
by pairEq j
$\Gamma \vdash s_1 = s_2 \in S$ ⌊Ax⌋
$\Gamma \vdash t_1 = t_2 \in T[s_1/x]$ ⌊Ax⌋
$\Gamma,\ x'{:}S \vdash T[x'/x] \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma \vdash x{:}S{\times}T$ ⌊**ext** $\langle s,t\rangle$⌋
by pairI j s
$\Gamma \vdash s \in S$ ⌊Ax⌋
$\Gamma \vdash T[s/x]$ ⌊**ext** t⌋
$\Gamma,\ x'{:}S \vdash T[x'/x] \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma \vdash \mathsf{let}\ \langle x_1,y_1\rangle = e_1\ \mathsf{in}\ t_1$
$= \mathsf{let}\ \langle x_2,y_2\rangle = e_2\ \mathsf{in}\ t_2 \in C[e_1/z]$ ⌊Ax⌋
by spreadEq z C $x{:}S{\times}T$
$\Gamma \vdash e_1 = e_2 \in x{:}S{\times}T$ ⌊Ax⌋
$\Gamma,\ s{:}S,\ t{:}T[s/x],\ y{:}\ e_1{=}\langle s,t\rangle \in x{:}S{\times}T$
$\vdash t_1[s,t/x_1,y_1] = t_2[s,t/x_2,y_2] \in C[\langle s,t\rangle/z]$ ⌊Ax⌋

$\Gamma,\ z{:}\ x{:}S{\times}T,\ \Delta \vdash C$ ⌊**ext** $\mathsf{let}\ \langle s,t\rangle = z\ \mathsf{in}\ u$⌋
by productE i
$\Gamma,\ z{:}\ x{:}S{\times}T,\ s{:}S,\ t{:}T[s/x],\ \Delta[\langle s,t\rangle/z]$
$\vdash C[\langle s,t\rangle/z]$ ⌊**ext** u⌋

$\Gamma \vdash \mathsf{let}\ \langle x,y\rangle = \langle s,t\rangle\ \mathsf{in}\ u = t_2 \in T$ ⌊Ax⌋
by spreadRed
$\Gamma \vdash u[s,t/x,y] = t_2 \in T$ ⌊Ax⌋

$\Gamma \vdash \langle s_1,t_1\rangle = \langle s_2,t_2\rangle \in S{\times}T$ ⌊Ax⌋
by pairEq_indep
$\Gamma \vdash s_1 = s_2 \in S$ ⌊Ax⌋
$\Gamma \vdash t_1 = t_2 \in T$ ⌊Ax⌋

$\Gamma \vdash S{\times}T$ ⌊**ext** $\langle s,t\rangle$⌋
by pairI_indep
$\Gamma \vdash S$ ⌊**ext** s⌋
$\Gamma \vdash T$ ⌊**ext** t⌋

Abbildung 3.10: Regeln für Produktraum

Wir haben diese Regeln wie immer in ihrer allgemeinsten Form durch Regelschemata beschrieben, welche auf die konkrete Situation angepaßt werden muß. Dabei haben wir im wesentlichen die Dekomposition – also die Form, die ein Anwender der Theorie beim Arbeiten mit NuPRL sehen wird – hervorgehoben während das Validierungsschema in eckigen Klammern (⌊**ext** t⌋) neben den entsprechenden Sequenzenschemata steht. Zur Vereinfachung der textlichen Präsentation haben wir Extrakt-Terme der Gestalt ⌊**ext** Axiom⌋ durch ⌊Ax⌋ und Teilziele der Gestalt $s{=}s \in T$ durch $s \in T$ abgekürzt.[33]

Die neuen Variablen in den Unterzielen werden automatisch generiert, falls Umbenennungen erforderlich sind. Es gibt jedoch noch eine Reihe weiterer Parameter, die von manchen Regeln zu Steuerungszwecken benötigt werden und vom Benutzer anzugeben sind. Dies sind

1. Die Position i einer zu elimierenden Variablen innerhalb der Hypothesenliste wie zum Beispiel bei der Regel `functionE_indep` i.
2. Ein Term, der als einzusetzender Wert für die weitere Verarbeitung benötigt wird, wie zum Beispiel s in der Regel `functionE` i s.
3. Das Level j des Universums, was innerhalb eines Unterziels benötigt wird, um nachzuweisen, daß ein bestimmter Teilterm ein Typ ist. Dies muß zum Beispiel in `lambdaEq` j angegeben werden, während es im Falle der Regel `functionEq` bereits bekannt ist.
4. Der Typ T eines Teilterms des zu analysierenden Beweiszieles, welcher in einem Unterziel isoliert auftritt. Dies wird vor allem beim Schließen über nichtkanonische Formen benötigt, in denen der Typ des Hauptargumentes nicht eindeutig aus dem Kontext hervorgeht. So muß zum Beispiel bei `applyEq` $x{:}S{\to}T$ der Typ der Funktion f_1 genannt werden, die als Hauptargument der Applikation auftritt.

[33] Diese Abkürzungen sind jedoch eine reine Notationsform für die Regeln und nicht etwa ein Teil der Typentheorie selbst. In den Regelschemata von NuPRL steht ⌊**ext** Axiom⌋, wo in diesem Skript ⌊Ax⌋ steht und $s = s \in T$, wo $s \in T$ steht.

$\Gamma \vdash S_1{+}T_1 = S_2{+}T_2 \in \mathsf{U}_j$ ⌊Ax⌋
by unionEq
$\Gamma \vdash S_1 = S_2 \in \mathsf{U}_j$ ⌊Ax⌋
$\Gamma \vdash T_1 = T_2 \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma \vdash \mathsf{inl}(s_1) = \mathsf{inl}(s_2) \in S{+}T$ ⌊Ax⌋
by inlEq j
$\Gamma \vdash s_1 = s_2 \in S$ ⌊Ax⌋
$\Gamma \vdash T \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma \vdash S{+}T$ ⌊**ext** $\mathsf{inl}(s)$⌋
by inlI j
$\Gamma \vdash S$ ⌊**ext** s⌋
$\Gamma \vdash T \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma \vdash \mathsf{inr}(t_1) = \mathsf{inr}(t_2) \in S{+}T$ ⌊Ax⌋
by inrEq j
$\Gamma \vdash t_1 = t_2 \in T$ ⌊Ax⌋
$\Gamma \vdash S \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma \vdash S{+}T$ ⌊**ext** $\mathsf{inr}(t)$⌋
by inrI j
$\Gamma \vdash T$ ⌊**ext** t⌋
$\Gamma \vdash S \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma \vdash \mathsf{case}\ e_1\ \mathsf{of}\ \mathsf{inl}(x_1) \mapsto u_1 \mid \mathsf{inr}(y_1) \mapsto v_1$
$= \mathsf{case}\ e_2\ \mathsf{of}\ \mathsf{inl}(x_2) \mapsto u_2 \mid \mathsf{inr}(y_2) \mapsto v_2$
$\in C[e_1/z]$ ⌊Ax⌋
by decideEq z C $S{+}T$
$\Gamma \vdash e_1 = e_2 \in S{+}T$ ⌊Ax⌋
$\Gamma,\ s{:}S,\ y{:}\ e_1{=}\mathsf{inl}(s) \in S{+}T$
$\vdash u_1[s/x_1] = u_2[s/x_2] \in C[\mathsf{inl}(s)/z]$ ⌊Ax⌋
$\Gamma,\ t{:}T,\ y{:}\ e_1{=}\mathsf{inr}(t) \in S{+}T$
$\vdash v_1[t/y_1] = v_2[t/y_2] \in C[\mathsf{inr}(t)/z]$ ⌊Ax⌋

$\Gamma,\ z{:}S{+}T,\ \Delta \vdash C$
⌊**ext** $\mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u \mid \mathsf{inr}(y) \mapsto v$⌋
by unionE i
$\Gamma,\ z{:}S{+}T,\ x{:}S,\ \Delta[\mathsf{inl}(x)/z]$
$\vdash C[\mathsf{inl}(x)/z]$ ⌊**ext** u⌋
$\Gamma,\ z{:}S{+}T,\ y{:}T,\ \Delta[\mathsf{inr}(y)/z]$
$\vdash C[\mathsf{inr}(y)/z]$ ⌊**ext** v⌋

$\Gamma \vdash \mathsf{case}\ \mathsf{inl}(s)\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u \mid \mathsf{inr}(y) \mapsto v$
$= t_2 \in T$ ⌊Ax⌋
by decideRedL
$\Gamma \vdash u[s/x] = t_2 \in T$ ⌊Ax⌋

$\Gamma \vdash \mathsf{case}\ \mathsf{inr}(t)\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u \mid \mathsf{inr}(y) \mapsto v$
$= t_2 \in T$ ⌊Ax⌋
by decideRedR
$\Gamma \vdash v[t/y] = t_2 \in T$ ⌊Ax⌋

Abbildung 3.11: Regeln für disjunkte Vereinigung (Summe)

5. Ein Term C und eine Variable z, die in C frei vorkommt. Dies ist in manchen Fällen nötig, um die Abhängigkeit eines Teilterms der Konklusion von einem anderen Teilterm deutlich zu machen. Dies wird zum Beipiel in der nichtkanonischen Regel `spreadEq` z C $x{:}S{\times}T$ des Produktraumes benötigt, welche in Abbildung 3.10 aufgeführt ist.

   Im Typ der Gleichheit $\mathsf{let}\ \langle x_1,y_1\rangle = e_1\ \mathsf{in}\ t_1 = \mathsf{let}\ \langle x_2,y_2\rangle = e_2\ \mathsf{in}\ t_2 \in C[e_1/z]$ kann es sein, daß bestimmte Vorkommen des Teilterms e_1 in C mit dem e_1 des let-Konstruktes in Verbindung zu bringen sind und andere nicht. Ist zum Beispiel C der Term $e_1{=}e_1 \in T$, so könnte dies zum Beispiel eine Instanz von $e_1{=}z \in T$ sein oder eine von $z{=}z \in T$.

   Da die Konklusion C üblicherweise in instantiierter Form vorliegt, muß die intendierte Abhängigkeit zwischen C und dem Teilterm e_1 explizit angegeben werden. Dies kann dadurch geschehen, daß man die Konlusion C dadurch verallgemeinert, daß man e_1 an den gewünschten Stellen durch eine neue Variable z ersetzt (also die Konklusion als Instanz $C[e_1/z]$ des allgemeineren Terms auffaßt) und sowohl z als auch die verallgemeinerte Konklusion als Steuerungsparameter für die Regel mit angibt. Damit wird zum Beispiel bei Anwendung von `spreadEq` festgelegt, e_1 ersetzt werden und welche bestehen bleiben, also ob im zweiten von `spreadEq` erzeugten Teilziel $e_1{=}\langle s,t\rangle \in T$ oder $\langle s,t\rangle{=}\langle s,t\rangle \in T$ steht.

In den meisten Fällen können die letzten drei Parameter automatisch berechnet werden, da sie aus dem Beweiskontext eindeutig hervorgehen. Da dies aber nicht in allen Beweisen der Fall ist, müssen sie als separate Bestandteile der Regeln mit aufgeführt werden. Wir wollen im folgenden nun alle Beweisregeln der Typentheorie tabellarisch zusammenstellen, die bis zu diesem Zeitpunkt relevant sind und die Besonderheiten, die sich nicht unmittelbar aus den Semantiktabellen ergeben, kurz erläutern.

$\Gamma \vdash \mathsf{U}_j \;\in\; \mathsf{U}_k \;\lfloor\mathsf{Ax}\rfloor$ **by** `univEq` *	$\Gamma \vdash T \;\in\; \mathsf{U}_k \;\lfloor\mathsf{Ax}\rfloor$ **by** `cumulativity` j * $\Gamma \vdash T \;\in\; \mathsf{U}_j \;\lfloor\mathsf{Ax}\rfloor$
$\Gamma \vdash\; s_1 = t_1 \in T_1 \;=\; s_2 = t_2 \in T_2 \;\in\; \mathsf{U}_j \;\lfloor\mathsf{Ax}\rfloor$ **by** `equalityEq` $\Gamma \vdash T_1 = T_2 \;\in\; \mathsf{U}_j \;\lfloor\mathsf{Ax}\rfloor$ $\Gamma \vdash s_1 = s_2 \;\in\; T_1 \;\lfloor\mathsf{Ax}\rfloor$ $\Gamma \vdash t_1 = t_2 \;\in\; T_1 \;\lfloor\mathsf{Ax}\rfloor$	$\Gamma \vdash \mathsf{Axiom} \;\in\; s = t \in T \;\lfloor\mathsf{Ax}\rfloor$ **by** `axiomEq` $\Gamma \vdash s = t \;\in\; T \;\lfloor\mathsf{Ax}\rfloor$ $\Gamma,\; z{:}\; s = t \in T,\; \Delta \vdash C \;\lfloor\textbf{ext } u\rfloor$ **by** `equalityE` i $\Gamma,\; z{:}\; s = t \in T,\; \Delta[\mathsf{Axiom}/z]$ $\vdash C[\mathsf{Axiom}/z] \;\lfloor\textbf{ext } u\rfloor$
$\Gamma,\; x{:}T,\; \Delta \vdash x \;\in\; T \;\lfloor\mathsf{Ax}\rfloor$ **by** `hypEq` i	$\Gamma \vdash\; s = t \in T \;\lfloor\mathsf{Ax}\rfloor$ **by** `symmetry` $\Gamma \vdash\; t = s \in T \;\lfloor\mathsf{Ax}\rfloor$
$\Gamma \vdash\; s = t \in T \;\lfloor\mathsf{Ax}\rfloor$ **by** `transitivity` t' $\Gamma \vdash\; s = t' \in T \;\lfloor\mathsf{Ax}\rfloor$ $\Gamma \vdash\; t' = t \in T \;\lfloor\mathsf{Ax}\rfloor$	$\Gamma \vdash C[s/x] \;\lfloor\textbf{ext } u\rfloor$ **by** `subst` $j\;\; s = t \in T\;\; x\;\; C$ $\Gamma \vdash\; s = t \in T \;\lfloor\mathsf{Ax}\rfloor$ $\Gamma \vdash C[t/x] \;\lfloor\textbf{ext } u\rfloor$ $\Gamma,\; x{:}T \vdash C \;\in\; \mathsf{U}_j \;\lfloor\mathsf{Ax}\rfloor$

*: *Die Regel ist nur anwendbar, wenn $j < k$ ist.*

Abbildung 3.12: Regeln für Universen und Gleichheit

Die Regeln für den *Produktraum* in Abbildung 3.10 sind strukturell sehr ähnlich zu denen des Funktionenraumes. Um die Gleichheit zweier Tupel nachzuweisen, muß man sie komponentenweise untersuchen und – im Falle abhängiger Produkte $x{:}S{\times}T$ – zusätzlich die allgemeine Typ-Eigenschaft von T nachweisen. Um ein Element von $x{:}S{\times}T$ einführen zu können, muß man im abhängigen Fall die erste Komponente explizit angeben, da der Typ der zweiten davon abhängt. Bei unabhängigen Produkten sind die entsprechenden Regeln erheblich einfacher und daher separat aufgeführt. Die Gleichheit der nichtkanonischen Formen (`spreadEq`) wird ebenfalls durch strukturelle Dekomposition in Teilausdrücke untersucht, wobei die obenerwähnten Komplikationen zu berücksichtigen sind. Die Elimination einer Produktvariablen z führt zur Einführung zweier Variablen für die Einzelkomponenten, die als Tupel an die Stelle von z treten.

Die meisten Regeln für die *disjunkte Vereinigung* in Abbildung 3.11 sind naheliegend. Bei den kanonischen Regeln für inl und inr muß jedoch sichergestellt werden, daß der *gesamte* Ausdruck $S{+}T$ einen Typ darstellt. Da jeweils nur noch einer der beiden Teiltypen weiter betrachtet wird, müssen die Teilziele $T \;\in\; \mathsf{U}_j$ bzw. $S \;\in\; \mathsf{U}_j$ in diese Regeln explizit hineingenommen werden. Die Regel `decideEq` entspricht der Fallanalyse.

Wie in Abschnitt 3.2.3.1 besprochen, dient eine kumulative Hierarchie von *Universen* der syntaktischen Repräsentation der Typeigenschaft. Alle Typkonstruktoren – einschließlich U_j für $j < k$ – erzeugen somit kanonische Elemente eines Universums U_k. Da die entsprechenden kanonischen Regeln für Universen bereits als Formationsregeln des entsprechenden Typs aufgeführt wurden, gibt es keine speziellen kanonischen Regeln für Universen. Nichtkanonische Ausdrücke für Universen gibt es ebenfalls nicht und damit entfällt auch die Berechnungsregel. Bedeutend ist jedoch eine Regel zur Darstellung der Kumulativität, die es ermöglicht, Ziele wie $T{\to}\mathsf{U}_1 \;\in\; \mathsf{U}_2$ zu beweisen, wenn T nur ein Element von U_1 ist.

Der Typ der *Gleichheit* besitzt nur ein kanonisches Element Axiom und keine nichtkanonischen Ausdrücke. Aus diesem Grunde erzeugt jede Regel, deren Beweisziel eine Gleichheit ist, den Term Axiom als Extrakt-Term. Die Regel `equalityE` wird relativ selten eingesetzt, da sie wenig praktische Auswirkungen hat. Die einzig bedeutenden Regeln sind die bekannten Regeln zum Schließen über Gleichheit. Dabei ist `hypEq` eine Art bedingte Reflexivitätsregel. Allgemeine Reflexivität (`x = x ∈ T`) gilt nicht, da Typzugehörigkeit geprüft

$\Gamma,\; x{:}T,\; \Delta \vdash T \;\lfloor\textbf{ext } x\rfloor$ **by** `hypothesis` i	$\Gamma \vdash T \;\lfloor\textbf{ext } t\rfloor$ **by** `intro` t $\Gamma \vdash t \;\in\; T \;\lfloor\mathsf{Ax}\rfloor$
$\Gamma, \Delta \vdash C \;\lfloor\textbf{ext } (\lambda x.t)\, s\rfloor$ **by** `cut` $i\;\; T$ $\Gamma, \Delta \vdash T \;\lfloor\textbf{ext } s\rfloor$ $\Gamma, x{:}T, \Delta \vdash C \;\lfloor\textbf{ext } t\rfloor$	$\Gamma,\; x{:}T,\; \Delta \vdash C \;\lfloor\textbf{ext } t\rfloor$ **by** `thin` i $\Gamma, \Delta \vdash C \;\lfloor\textbf{ext } t\rfloor$

Abbildung 3.13: Strukturelle Regeln

werden muß. Die Regeln `symmetry` und `transitivity` sind – wie bereits in Beispiel 2.2.26 auf Seite 42 – redundant, da sie durch die Substitutionsregel simuliert werden können.[34] Die Regeln für Universen und Gleichheit sind in Abbildung 3.12 zusammengestellt.

Neben den Regeln, welche die Semantik einzelner Ausdrücke der Typentheorie wiederspiegeln, benötigen wir noch eine Reihe *struktureller Regeln*, von denen wir einige bereits aus Abschnitt 2.2.4 kennen.

- Mit der *Hypothesenregel* `hypothesis` können wir Konklusionen beweisen, die bereits als Typausdruck in der Hypothesenliste erscheinen. Extrakt-Term ist in diesem Falle die an dieser Stelle deklarierte Variable.

- Eine *explizite Einführungsregel* `intro` ermöglicht es, den Extrakt-Term für eine Konklusion direkt anzugeben. In diesem Falle muß natürlich noch seine Korrektheit überprüft werden.

- Die *Schnittregel* `cut` erlaubt das Einfügen von Zwischenbehauptungen. Diese sind in einem Teilziel zu beweisen und dürfen in einem anderen zum Beweis weiterverwendet werden. Die gewünschte Position der Zwischenbehauptung in der Hypothesenliste kann angegeben werden, wobei aber darauf zu achten ist, daß alle freien Variablen der Zwischenbehauptung in früheren Hypothesen deklariert sein müssen.

- Mit der *Ausdünnungsregel* `thin` werden überflüssige Annahmen aus der Hypothesenliste entfernt, um den Beweis überschaubarer zu machen. Deklarationen von noch benutzten freien Variablen können jedoch nicht ausgedünnt werden.

3.2.4 Grundprinzipien des systematischen Aufbaus

In diesem Abschnitt haben wir die allgemeinen Prinzipien und Entwurfsentscheidungen vorgestellt, die beim formalen Aufbau der Typentheorie eine zentrale Rolle spielen und am Beispiel dreier Typkonstrukte erläutert. Der wesentliche Grundsatz war dabei, Definitionen bereitzustellen, welche die formale Struktur von Syntax, Semantik und Inferenzsystem *unabhängig von der konkreten Theorie* eindeutig festlegen und es dann erlauben, die eigentliche Theorie kurz und einfach durch Einträge in diversen Tabellen zu definieren, welche die Anforderungen der allgemeinen Definitionen respektieren. Dies erleichtert sowohl ein Verständnis als auch eine Implementierung der formalen Theorie.

Zugunsten einer einheitlichen Syntax wurde entschieden, die *Abstraktionsform* eines Terms und seine *Darstellungsform* getrennt voneinander zu behandeln. Dies ermöglicht, eine einfache Definition für Terme, Variablenbindung und Substitution zu geben und in einer *Operatorentabelle* die konkrete Syntax der Theorie anzugeben. Hierdurch gewinnen wir maximale Flexibilität bei der formalen Repräsentation von Konstrukten aus Mathematik und Programmierung innerhalb eines computerisierten Beweissystems.

[34]In NuPRL gibt es anstelle der beiden Regeln `symmetry` und `transitivity` eine komplexere `equality` Regel. Diese Regel ist als Entscheidungsprozedur zum allgemeinen Schließen über Gleichheit implementiert und kann eine vielfache Anwendung von (bedingter) Reflexivität, Symmetrie und Transitivität in einem Schritt durchführen. Wir werden diese Regel und den zugrundeliegenden Algorithmus bei der Besprechung der Automatisierungsmöglichkeiten im nächsten Kapitel ausführlicher vorstellen.

Die Semantik der Typentheorie basiert auf *Lazy Evaluation*. Bestimmte Terme werden anhand ihres Operatornamens als kanonisch deklariert und nichtkanonische Terme werden nur soweit ausgewertet, bis ihre äußere Form kanonisch ist. Neben dem allgemeinen Auswertungsalgorithmus benötigt die konkrete Ausprägung der Theorie hierfür nur eine *Tabelle von Redizes und Kontrakta*. Die eigentliche Semantik wird nun durch *Urteile* über Terme festgelegt, wobei es nur vier Formen von Grundurteilen gibt. In einer allgemeinen Definition wird fixiert, wie für gegebene Terme die entsprechenden Urteile zu bestimmen sind, und für eine konkrete Ausprägung dieser Urteile werden wieder Einträge in einer Semantiktabelle vorgenommen.

Die syntaktische Repräsentationsform der semantischen Urteile innerhalb des Inferenzsystems bilden die *Sequenzen*. Zugunsten einer einheitlichen Darstellung wurde ein Gleichheitstyp und der Typ der Universen eingeführt. Hierbei wurde entschieden, logische Aussagen durch Typen darzustellen, deren Elemente ihren Beweisen entsprechen (*Propositionen als Datentypen*), und eine *kumulative Hierarchie von Universen* zu verwenden, um Widersprüche zu vermeiden. Der Kalkül wurde so ausgelegt, daß eine interaktive *Entwicklung* von Beweisen und Programmen unterstützt wird und nicht nur ihre Überprüfung. Dementsprechend müssen Inferenzregeln nicht nur Beweisziele in Teilziele zerlegen, sondern auch implizit *Extrakt-Terme* verwalten.

Auf dieser Basis ist bei (nicht-konservativer) Einführung eines neuen Konstrukts wie folgt vorzugehen.

- Die Syntax und die Darstellungsform der neuen Terme ist in die Operatortabelle einzutragen.
- Kanonische und nichtkanonische Terme sind zu trennen. Die Redex-Kontrakta Tabelle ist entsprechend zu erweitern.
- Die Semantik (Typgleichheit und Elementegleichheit) ist durch Einträge in der Typ- bzw. Elementsemantiktabelle zu fixieren. Hierbei ist darauf zu achten, daß keine Widersprüchlichkeiten entstehen.
- Es sind Inferenzregeln aufzustellen, welche diese Semantik weitestgehend wiederspiegeln.

Bei einer konservativen Erweiterung durch einen Benutzer der Theorie geht man im Prinzip genauso vor. Allerdings wird hierbei die Abstraktionsform als Abkürzung für einen bereits bestehenden komplexeren Term definiert. Daher ergibt sich die (widerspruchsfreie!) Semantik und die zugehörigen Inferenzregeln mehr oder weniger automatisch aus der Definition.

3.3 Logik in der Typentheorie

Mit den bisher vorgestellten Konstrukten können wir die elementaren Grundkonzepte der Programmierung innerhalb der Typentheorie repräsentieren. Wir haben dabei allerdings schon öfter angedeutet, daß wir die Prädikatenlogik über das Prinzip "Propositionen als Datentypen" einbetten wollen. Dies wollen wir nun weiter ausarbeiten und die Konsequenzen der Einbettung untersuchen.

3.3.1 Die Curry-Howard Isomorphie

In Abschnitt 2.4.7 (Seite 87) hatten wir die *Curry-Howard Isomorphie* zwischen der einfachen Typentheorie und dem Implikationsfragment der Prädikatenlogik aufgedeckt. Wir wollen nun zeigen, daß auch die anderen logischen Operationen ein isomorphes Gegenstück innerhalb der Typentheorie besitzen. Wir bedienen uns hierzu einer konstruktiven Interpretation der logischen Symbole, die davon ausgeht, daß eine logische Aussage wahr ist, wenn wir einen Beweis – also ein Element des entsprechenden Datentyps – finden können.

Konjunktion: Um $A \wedge B$ zu beweisen, müssen wir in der Lage sein, A und B unabhängig voneinander zu beweisen. Ist also a ein Beweis für A und b einer für B, dann liefern a und b zusammen – also zum Beispiel das Paar $\langle a,b\rangle$ – alle Evidenzen, die wir benötigen, um die Gültigkeit von $A \wedge B$ nachzuweisen. Damit verhält sich die Konjunktion im wesentlichen wie ein *(unabhängiger) Produktraum*. Dies wird

besonders deutlich, wenn man die entsprechenden Inferenzregeln für die Konjunktion mit den impliziten Regeln des unabhängigen Produkts vergleicht.[35]

$\Gamma \vdash A \wedge B$	$\Gamma \vdash A \times B$ $\lfloor$**ext** $\langle a,b\rangle\rfloor$
by `and_i`	**by** `pairI_indep`
$\Gamma \vdash A$	$\Gamma \vdash A$ $\lfloor$**ext** $a\rfloor$
$\Gamma \vdash B$	$\Gamma \vdash B$ $\lfloor$**ext** $b\rfloor$

Die Einführungsregel für die Konjunktion ist nichts anderes als ein Spezialfall der Regel `pairI_indep`, bei dem man die Validierung unterdrückt hat. Ähnlich sieht es bei den Eliminationsregeln aus.

$\Gamma, A \wedge B, \Delta \vdash C$	$\Gamma,\ z{:}\ A \times B,\ \Delta \vdash C$ $\lfloor$**ext** let $\langle a,b\rangle = z$ in $u\rfloor$
by `and_e` i	**by** `productE` i
$\Gamma, A, B, \Delta \vdash C$	$\Gamma,\ z{:}\ A \times B,\ a{:}A,\ b{:}B,\ \Delta[\langle a,b\rangle/z]$
	$\vdash C[\langle a,b\rangle/z]$ $\lfloor$**ext** $u\rfloor$

Der Unterschied besteht hier darin, daß innerhalb der Typentheorie alle Formeln (Typen) in den Hypothesen mit Variablen in Verbindung gebracht werden und mögliche Abhängigkeiten der Hypothesen und der Konklusion von diesen Variablen ebenfalls berücksichtigt sind. Streicht man diese Abhängigkeiten aus der Regel wieder heraus, weil sie innerhalb der Prädikatenlogik nicht auftauchen können, und dünnt danach[36] die redundante Hypothese $z{:}\ A \times B$ aus, so erhält man folgende Spezialisierung von `productE` i, die den Zusammenhang zu `and_e` i deutlich macht.

$\Gamma,\ z{:}\ A \times B,\ \Delta \vdash C$ $\lfloor$**ext** let $\langle a,b\rangle = z$ in $u\rfloor$
by `productE` i `THEN thin` i
$\Gamma,\ a{:}A,\ b{:}B,\ \Delta \vdash C$ $\lfloor$**ext** $u\rfloor$

Aus dieser Regel wird auch deutlich, welche Evidenzen sich ergeben, wenn man eine Konjunktion eliminiert: ist bekannt, wie aus Beweisen a für A und b für B ein Beweis u für C aufgebaut werden kann, so beschreibt let $\langle a,b\rangle = z$ in u, wie der Beweis für C aus einem Beweis z für $A \wedge B$ zu konstruieren ist.

Disjunktion: Um $A \vee B$ zu beweisen, müssen wir entweder A oder B beweisen können. Die Menge aller Beweise für $A \vee B$ ist damit die *disjunkte Vereinigung* der Beweise für A und der Beweise für B. Auch hier zeigt ein Vergleich der entsprechenden Beweisregeln die genauen Zusammenhänge auf.

$\Gamma \vdash A \vee B$	$\Gamma \vdash A+B$ $\lfloor$**ext** inl$(a)\rfloor$
by `or_i1`	**by** `inlI` j
$\Gamma \vdash A$	$\Gamma \vdash A$ $\lfloor$**ext** $a\rfloor$
	$\Gamma \vdash B \in \mathsf{U}_j$ $\lfloor$Ax$\rfloor$

Der Unterschied zwischen diesen beiden Regeln besteht nur darin, daß bei der disjunkten Vereinigung das Level des Universums von B angegeben werden muß. Innerhalb der Prädikatenlogik *erster Stufe* steht eigentlich fest, daß immer das Level 1 zu wählen ist, da Typen (Propositionen) höherer Universen (Stufen) nicht erlaubt sind. Mit dem zweiten Ziel ist also nur nachzuweisen, daß B tatsächlich eine Formel ist. Wegen der wesentlich einfacheren Syntax der Prädikatenlogik, die eine Mischung von Formeln und Termen verbietet (siehe Definition 2.2.4 auf Seite 24), ist diese Überprüfung jedoch automatisierbar und könnte ganz entfallen. Wir werden daher in den folgenden Vergleichen derartige *Wohlgeformtheitsziele* unterdrücken[37] und erhalten als zweite Einführungsregel für die Disjunktion

$\Gamma \vdash A \vee B$	$\Gamma \vdash A+B$ $\lfloor$**ext** inr$(b)\rfloor$
by `or_i2`	**by** `inrI 1 THEN ...`
$\Gamma \vdash B$	$\Gamma \vdash B$ $\lfloor$**ext** $b\rfloor$

Die spezialisierte Eliminationsregel `unionE` i erweitert die Regel `or_e` i ebenfalls wieder nur um die Konstruktion der Evidenzen. Hierdurch wird auch deutlicher, daß die Elimination einer Disjunktion im wesentlichen eine Fallunterscheidung ist.

[35] Zum Zwecke der Vergleichbarkeit haben wir in den typentheoretischen Regelschemata S durch A und T durch B ersetzt.
[36] Das *Tactical* `THEN` (siehe Abschnitt 4.2.4) erlaubt es, mehrere Regeln direkt miteinander zu verbinden.
[37] In einer Logik höherer Stufe, in der auch über beliebig zusammengesetzte Formeln geschlossen werden darf, muß das Wohlgeformtheitsziel dagegen bestehen bleiben, da hier eine vollständige Automatisierung nicht möglich ist.

$\Gamma, A \vee B, \Delta \vdash C$ **by** `or_e` i $\Gamma, A, \Delta \vdash C$ $\Gamma, B, \Delta \vdash C$	$\Gamma,\ z{:}A{+}B,\ \Delta \vdash C$ $\lfloor$**ext** case z of inl$(a) \mapsto u$ | inr$(b) \mapsto v\rfloor$ **by** `unionE` i `THEN thin` i $\Gamma,\ a{:}A,\ \Delta \vdash C$ $\lfloor$**ext** $u\rfloor$ $\Gamma,\ b{:}B,\ \Delta \vdash C$ $\lfloor$**ext** $v\rfloor$

Auch zeigt die Regel `unionE`, welche Evidenzen sich bei der Elimination ergeben: ist bekannt, wie aus einem Beweis a für A bzw. aus b für B ein Beweis u (bzw. v) für C aufgebaut werden kann, so beschreibt case z of inl$(a) \mapsto u$ | inr$(b) \mapsto v$ einen Beweis für C auf der Grundlage eines Beweises z für $A \vee B$.

Implikation: Der grundsätzliche Zusammenhang zwischen der Implikation und dem *unabhängigen Funktionenraum* ist bereits aus Abschnitt 2.4.7 bekannt. Auch hier machen die spezialisierten Regeln (mit Unterdrückung der Wohlgeformtheitsziele) deutlich, wie die Beweise zusammengesetzt werden.

$\Gamma \vdash A \Rightarrow B$ **by** `imp_i` $\Gamma, A \vdash B$	$\Gamma \vdash A {\to} B$ $\lfloor$**ext** $\lambda x.b\rfloor$ **by** `lambdaI 1 THEN ...` $\Gamma,\ x{:}A \vdash B$ $\lfloor$**ext** $b\rfloor$
$\Gamma, A \Rightarrow B, \Delta \vdash C$ **by** `imp_e` i $\Gamma, A \Rightarrow B, \Delta \vdash A$ $\Gamma, \Delta, B \vdash C$	$\Gamma,\ pf{:}\ A {\to} B,\ \Delta \vdash C$ $\lfloor$**ext** $b[pf\,a,/y]\rfloor$ **by** `functionE_indep` i `THEN ...` $\Gamma,\ pf{:}\ A {\to} B,\ \Delta \vdash A$ $\lfloor$**ext** $a\rfloor$ $\Gamma,\ y{:}B,\ \Delta \vdash C$ $\lfloor$**ext** $b\rfloor$

In der spezialisierten Regel wird die Hypothese $pf{:}\ A{\to}B$ nur im zweiten Teilziel ausgedünnt.

Negation: $\neg A$ kann gemäß unserer Definiton auf Seite 34 als Abkürzung für $A \Rightarrow \Lambda$ angesehen werden. Damit ergibt sich die Repräsentation der Negation aus dem unabhängigen Funktionenraum und der Darstellung der Falschheit.

Falschheit: Λ ist eine logische Aussage, für die es keine Beweise geben darf. Der zugehörige Datentyp muß also ein Typ ohne Elemente – ein Gegenstück zur leeren Menge sein. Diesen Datentyp void werden wir im folgenden Abschnitt genauer besprechen.

Universelle Quantifizierung: Um $\forall x{:}T.A$ zu beweisen, müssen wir eine allgemeine Methode angeben, wie wir $A[x]$ für ein beliebiges $x \in T$ beweisen, ohne weitere Informationen über x zu besitzen. Damit ist diese Methode im wesentlichen eine Funktion, welche für alle Eingaben $x \in T$ einen Beweis von $A[x]$ liefert. Der Ausgabetyp dieser Funktion – die Proposition $A[x]$ – hängt allerdings von x ab und deshalb ist das typentheoretische Gegenstück zur universellen Quantifizierung ein *abhängiger Funktionenraum*. Auch hier zeigen die spezialisierten Regeln die genauen Zusammenhänge.

$\Gamma \vdash \forall x{:}T.A$ **by** `all_i` * $\Gamma, x'{:}T \vdash A[x'/x]$	$\Gamma \vdash x{:}T {\to} A$ $\lfloor$**ext** $\lambda x'.a\rfloor$ **by** `lambdaI 1 THEN ...` $\Gamma,\ x'{:}T \vdash A[x'/x]$ $\lfloor$**ext** $a\rfloor$

Man beachte, daß die Eigenvariablenbedingung * in `all_i` "*Die Umbenennung* $[x'/x]$ *erfolgt, wenn* x *in* Γ *frei vorkommt*" innerhalb der typentheoretischen Regeln durch ein wesentlich einfacheres Kriterium dargestellt wird. Um die Reinheit der Sequenz zu erhalten, erfolgt eine Umbenennung $[x'/x]$, wenn x in Γ bereits deklariert wurde.

$\Gamma, \forall x{:}T.A, \Delta \vdash C$ **by** `all_e` i t $\Gamma, \forall x{:}T.A, \Delta, A[t/x] \vdash C$	$\Gamma,\ pf{:}\ x{:}T {\to} A,\ \Delta \vdash C$ $\lfloor$**ext** $u[pf\,t\,/\,y]\rfloor$ **by** `functionE` i t $\Gamma,\ pf{:}\ x{:}T {\to} A,\ \Delta \vdash t \in T$ $\lfloor$Ax$\rfloor$ $\Gamma,\ pf{:}\ x{:}T {\to} A,\ y{:}A[t/x],\ \Delta \vdash C$ $\lfloor$**ext** $u\rfloor$

Das zusätzliche erste Teilziel von `functionE` i t stellt sicher, daß es sich bei dem angegebenen Term t tatsächlich um einen Term vom Typ T handelt. Innerhalb der einfachen Prädikatenlogik kann dieser Zusammenhang nicht überprüft werden.

Existentielle Quantifizierung: Um $\exists x{:}T.A$ zu beweisen, müssen wir in der Lage sein, ein Element $t \in T$ anzugeben und für dieses Element einen Beweis a für $A[t/x]$ zu konstruieren. Damit besteht der gesamte Beweis im wesentlichen aus dem Paar $\langle t,a\rangle$, wobei $t \in T$ und $a \in A[t/x]$ gilt. Das bedeutet, daß zur Darstellung der existentiellen Quantifizierung ein *abhängiger Produktraum* genau das geeignete Konstrukt ist. Auch hier zeigen die Regeln wieder die genauen Zusammenhänge, wobei für die Typzugehörigkeit im ersten Teilziel von `pairI` dasselbe gilt wie im Falle von `functionE` und anstelle der Eigenvariablenbedingung in `ex_e` i wiederum das einfachere Kriterium bei der typentheoretischen Regel `productE` i verwendet werden kann.

$\Gamma \vdash \exists x{:}T.A$ **by** `ex_i` t $\Gamma \vdash A[t/x]$	$\Gamma \vdash x{:}T {\times} A$ $\lfloor$**ext** $\langle t,a\rangle\rfloor$ **by** `pairI 1` t `THEN ...` $\Gamma \vdash t \in T$ $\lfloor$Ax$\rfloor$ $\Gamma \vdash A[t/x]$ $\lfloor$**ext** $a\rfloor$
$\Gamma, \exists x{:}T.A, \Delta \vdash C$ **by** `ex_e` i ** $\Gamma, x'{:}T, A[x'/x], \Delta \vdash C$	$\Gamma,\ z{:}\ x{:}T {\times} A,\ \Delta \vdash C$ $\lfloor$**ext** let $\langle x',a\rangle = z$ in $u\rfloor$ **by** `productE` i `THEN thin` i $\Gamma,\ x'{:}T,\ a{:}A[x'/x],\ \Delta \vdash C$ $\lfloor$**ext** $u\rfloor$

Damit lassen sich alle logischen Operatoren mit Ausnahme der Falschheit Λ durch die bereits bekannten Konstrukte Funktionenraum, Produkt und Summe innerhalb der Typentheorie repräsentieren. Die so erweiterte Curry-Howard Isomorphie ist eine weitere praktische Rechtfertigung des Prinzips, Propositionen innerhalb der Typentheorie als Datentypen anzusehen. Dieses ermöglicht zudem eine einheitliche Behandlung von Logik und Berechenbarkeit in der intuitionistischen Typentheorie. Wir müssen also nur noch eine geeignete Beschreibung eines leeren Datentyps finden und erhalten dann den Kalkül der Prädikatenlogik mehr oder weniger umsonst.

3.3.2 Der leere Datentyp

Um logische Falschheit darzustellen, benötigen wir, wie soeben angedeutet, einen Datentyp, der im wesentlichen der leeren Menge entspricht. Prinzipiell gibt uns der Gleichheitstyp die Möglichkeit, diesen Datentyp, den wir mit dem Namen void bezeichnen, als konservative Erweiterung der bisherigen Theorie einzuführen, indem wir void auf einem Datentyp abstützen, von dem wir *wissen*, daß er leer sein muß. So könnte man zum Beispiel definieren

$$\texttt{void} \equiv \texttt{X}{:}\mathsf{U}_1 \to \texttt{X} = (\texttt{X}{\to}\texttt{X}) \in \mathsf{U}_1 .$$

Dennoch gibt es eine Reihe von Gründen, die dafür sprechen, void explizit einzuführen.

- Das Konzept der logischen Falschheit bzw. des leeren Datentyps ist *primitiv* – also ein grundlegendes mathematisches Konzept – und sollte daher als ein einfacher Typ des Universums U_1 dargestellt werden. Jede Simulation mit den bisherigen Typkonstrukten[38] gehört jedoch mindestens zum Universum U_2.
- Die Besonderheiten des leeren Datentyps, die bei einer Simulation eine untergeordnete Rolle spielen, sollten durch eine explizite Definition von Semantik und Inferenzregeln hervorgehoben werden.
- Der *Gleichheitstyp* besitzt keine nichtkanonischen Ausdrücke, die es erlauben, Gleichheiten zu analysieren. Insbesondere gibt es keine Möglichkeit, in formalen Beweisen die Aussage zu verwenden, daß eine Gleichheit *nicht* gilt. Dies zu dem Gleichheitstyp hinzunehmen wäre aber in etwa genauso aufwendig wie eine explizite Definition des Typs void[39] und wesentlich unnatürlicher, denn die Tatsache, daß void keine Elemente enthält, ist das wesentliche Charakteristikum dieses Typs.

[38] Bei Verwendung des Datentyps $\mathbb{Z}$ oder $\mathbb{B}$ wäre natürlich auch eine Simulation innerhalb des ersten Universums möglich, etwa durch $0 = 1 \in \mathbb{Z}$ oder $\mathsf{T} = \mathsf{F} \in \mathbb{B}$.

[39] Die umgekehrte Richtung, Ungleichheit durch Verwendung der logischen Falschheit $\Lambda \equiv$ void zu definieren, also zum Beispiel durch $s \neq t \in T \equiv s = t \in T \to$ void, kommt dem allgemeinen mathematischen Konsens erheblich näher.

Eine Simulation von void mit dem Typ der Gleichheit ist also möglich, aber aus praktischen Gründen nicht sinnvoll. Es sei jedoch hervorgehoben, daß alle Eigenschaften der Typentheorie, die sich aus der Hinzunahme von void ergeben – wie zum Beispiel der Verlust der starken Normalisierbarkeit – im Prinzip auf dem Gleichheitstyp beruhen bzw. auf der Tatsache, daß logische Propositionen innerhalb der Typentheorie dargestellt werden müssen. Es besteht daher keine Möglichkeit, diese Eigenschaften zu vermeiden.

So zwingt uns zum Beispiel die Hinzunahme eines leeren Datentyps, auch mit Konstrukten wie void$\times T$, void$+T$, $T\rightarrow$void und void$\rightarrow T$ umzugehen. Dabei sind die ersten beiden Typen relativ leicht zu interpretieren. Da void keine Elemente besitzt, muß void$\times T$ ebenfalls leer und void$+T$ isomorph zu T sein. Was aber ist mit den anderen beiden Konstrukten?

- $T\rightarrow$void beschreibt die Menge aller totalen Funktionen von T in den leeren Datentyp void. Wenn nun T nicht leer ist – also ein Element t besitzt – dann würde jede Funktion f aus $T\rightarrow$void ein Element $f\ t\ \in$ void beschreiben, dessen Existenz aber nun einmal ausgeschlossen sein soll. Folglich darf $T\rightarrow$void in diesem Fall keine Elemente besitzen. $T\rightarrow$void muß also leer sein, wann immer T Elemente besitzt.[40] Diese Erkenntnis deckt sich auch mit unserem Verständnis der logischen Falschheit. Eine Aussage der Form $A \Rightarrow \Lambda$ kann nicht beweisbar sein, wenn es einen Beweis für A gibt.

- Weitaus komplizierter ist die Interpretation von Typen der Art void$\rightarrow T$. Solche Typen besitzen in jedem Fall mindestens ein Element – selbst dann, wenn T ein leerer Datentyp ist – denn die Semantik von Funktionen (siehe Abbildung 3.6 auf Seite 112) besagt nun einmal, daß ein Ausdruck $\lambda x.t$ genau dann ein Element von void$\rightarrow T$ ist, wenn das Urteil $t[s/x] \in T$ für alle konkreten Elemente s von void gilt. Da void aber keine konkreten Elemente besitzt, die anstelle der Variablen x eingesetzt werden könnten, ist diese Bedingung in jedem Fall erfüllt. Auch diese Erkenntnis deckt sich mit unserem Verständnis der logischen Falschheit. Eine Aussage der Form $\Lambda \Rightarrow A$ ist immer wahr, denn wenn wir erst einmal gezeigt haben, daß ein Widerspruch (d.h. logische Falschheit) gilt, dann ist jede beliebige Aussage gültig.[41]

  Wie stellen wir dies nun syntaktisch dar? Aus dem oben Gesagten folgt, daß ein Beweisziel der Form

  Γ, z:void, $\Delta \vdash T$

  mit einer Regel voidE beweisbar sein muß, die – wie die äquivalente logische Regel false_e – keine weiteren Teilziele mehr erzeugt.

  Welchen Extrakt-Term aber soll diese Regel generieren? Es ist ein Term, der einerseits von z abhängen sollte und andererseits ein Element eines jeden beliebigen Datentyps T liefert, sofern z ein Element von void ist. Auch wenn wir wissen, daß wir für z niemals einen Term einsetzen können, müssen wir eine syntaktische Beschreibung angeben, welche den obigen Sachverhalt wiederspiegelt. Wir werden hierfür die Bezeichnung any(z) verwenden, um zu charakterisieren, daß hier ein Element eines jeden (englisch "*any*") Typs generiert würde, sofern es gelänge, für z ein Element von void einzusetzen. any(z) ist folglich ein nichtkanonischer Term, der zum Datentyp void gehört.

Auf den ersten Blick erscheint die Einführung eines nichtkanonischen Terms, der ein Element eines beliebigen Datentyps sein soll, etwas Widersinniges zu sein. Bedenkt man jedoch, daß – entsprechend unserem Beispiel 2.3.25 auf Seite 59 – **T = F** $\in \mathbb{B}$ eine gute Simulation für den Typ void ist, so kann man leicht einsehen, daß die gesamte Typenhierarchie ohnehin kollabieren würde, wenn es gelingen würde, ein Element von void zu finden: alle Typen wären gleich und jeder Term wäre auch ein Element eines jeden Typs. Die Forderung "any(z) $\in T$ falls $z \in$void" ist also etwas sehr Sinnvolles.

Die Hinzunahme des Terms any(z) – so seltsame Eigenschaften auch damit verbunden sind – ändert also nicht im Geringsten etwas an unserer bisherigen Theorie. Sie macht nur deutlich, welche Konsequenzen mit

[40] Ist T dagegen selbst ein leerer Datentyp, so ist es durchaus möglich, daß $T\rightarrow$void Elemente besitzt. So ist zum Beispiel λx.x ein zulässiges Element von void$\rightarrow$void.

[41] Vergleiche hierzu unsere Diskussion auf Seite 35 und das Beispiel 2.3.25 auf Seite 59, in dem wir aus der Gleichheit **F = T** die Gleichheit aller λ-Terme gefolgert haben.

kanonisch (Typen)	(Elemente)	nichtkanonisch
void{}()		**any**{}(e)
void		any(e)

Zusätzliche Einträge in die Operatorentabelle

Redex	Kontraktum
— *entfällt* —	

Zusätzliche Einträge in die Redex–Kontrakta Tabelle

Typsemantik		
void = void		
Elementsemantik		
$s = t \in$ void		*gilt niemals !*
void = void $\in$ U$_j$	falls	$j\geq 1$ (*als natürliche Zahl)*

Zusätzliche Einträge in den Semantiktabellen

$\Gamma \vdash$ void $\in$ U$_j$ [Ax]
by voidEq

$\Gamma \vdash$ any(s) = any(t) $\in T$ [Ax]
by anyEq
$\Gamma \vdash s = t \in$ void [Ax]

Γ, z:void, $\Delta \vdash C$ [ext any(z)]
by voidE i

Inferenzregeln

Abbildung 3.14: Syntax, Semantik und Inferenzregeln des Typs void

dem mathematischen Konzept des Widerspruchs verbunden sind. Es ist klar, daß der Term any(z) in der *Semantik* keine Rolle spielen darf, denn diese drückt ja nur aus, was gültig ist, und enthält in sich keine Widersprüche. Da void keine kanonischen Elemente enthält, wird any(z) auch niemals reduzierbar sein und somit überhaupt nicht zur Semantik eines Ausdrucks beitragen: *wenn Unsinn eingegeben wird, kann auch nichts Sinnvolles herauskommen.*

Insgesamt scheint die angesprochene Interpretation des Typs void ein angemessenes typentheoretisches Gegenstück sowohl zum logischen Konzept der Falschheit als auch zur leeren Menge zu liefern. Wir erweitern daher unsere bisherige Theorie wie folgt (siehe Abbildung 3.14).

- Die Operatorentabelle um zwei neue Operatoren **void**{}() und **any**{}(z) erweitert.
- Die Redex–Kontrakta Tabelle bleibt unverändert, da any(z) nicht reduzierbar ist. Entsprechend gibt es auch keine Reduktionsregel.
- In den Semantiktabellen wird festgelegt, daß void ein Typ (jedes Universums) ohne Elemente ist.[42]
- Die Tatsache daß void keine Elemente hat, wird dadurch widergespiegelt, daß es keine kanonischen Regeln gibt und daß die nichtkanonischen Regeln festlegen, daß any(z) zu jedem Typ gehört, wenn $z \in$void gilt.

Durch die Hinzunahme des leeren Datentyps wird deutlich, daß die Einbettung von Logik in eine Typentheorie und speziell die Einbettung der Gleichheit – die man ja zur Simulation von void einsetzen könnte – zu einigen Komplikationen in der Theorie führt, derer man sich bewußt sein sollte. So führt die Verwendung von Widersprüchen in den Annahmen (also void oder Λ) zu gültigen Beweisen einiger sehr seltsamer Aussagen.

[42] Der Eintrag 'void = void' bedeutet, daß void ein Typ ist und ein anderer Typ nur dann gleich ist zu void, wenn er sich mittels lässiger Auswertung direkt zu dem Term void reduzieren läßt.

Beispiel 3.3.1

Für $u \equiv$ `λX.λx.λy.y` können wir void→u ∈ U_1 beweisen, obwohl u selbst überhaupt kein Typ ist. Der formale Beweis arbeitet wie folgt

```
⊢ void→(λX.λx.λy.y) ∈ U1
by functionEq
|\
| ⊢ void ∈ U1
| by voidEq
 \
  x:void ⊢ λX.λx.λy.y ∈ U1
  by voidE 1
```

Als Konsequenz davon ist, daß in vollständigen und korrekten Beweisen durchaus Sequenzen mit unsinnigen Hypothesenlisten auftauchen können wie zum Beispiel `x:void, z:λX.λx.λy.y, ...`.

Es gibt keinen Weg, derartigen Unsinn in Beweisen auszuschließen. Es sei jedoch angemerkt, daß eine Deklaration der Art `x:void` immer auftreten *muß*, um so etwas zu erzeugen. Dies bedeutet, daß in einem solchen Fall die gesamte Hypothesenmenge ohnehin widersprüchlich ist, und es ist nicht ganz so schmerzhaft, unsinnige Deklarationen als Folge von Annahmen hinzunehmen, die niemals gültig sein können.

Es gibt jedoch noch eine zweite, wesentlich gravierendere Art von Problemen, die Anlaß dafür waren, lässige Auswertung als Reduktionsmechanismus zu verwenden. Es ist nämlich möglich, typisierbare Terme aufzustellen, die einen nichtterminierenden Teilterm besitzen. Ein Beispiel hierfür ist der Term

`λT.λz.(λx.x x)(λx.x x)`

der sich, wie in Abbildung 3.15 gezeigt, als Element des Typs `T:U1→void→T` nachweisen läßt. Die Konsequenz hiervon ist, daß im Gegensatz zur einfachen Typentheorie eine starke Normalisierbarkeit – im Sinne von "jede beliebige Folge von Reduktionen in beliebigen Teiltermen eines Terms terminiert" – im Allgemeinfall nicht mehr gilt. Es sei aber noch einmal angemerkt, daß dies nicht ein Fehler bei der Beschreibung des Typs void ist, sondern unausweichlich mit der Möglichkeit verbunden ist, Datentypen wie `X:U1 → X=X→X ∈ U1` zu formulieren, die keine Elemente besitzen. [43]

Korollar 3.3.2

Eine Typentheorie, in welche die Prädikatenlogik eingebettet werden kann, kann keine stark normalisierbare Reduktionsrelation besitzen.

Diese Tatsache führte im Endeffekt dazu, lässige Auswertung als grundlegende Reduktionsrelation für die (Martin-Löf'sche) Typentheorie zu verwenden. Der Term

`λz.(λx.x x)(λx.x x)`

ist bereits in kanonischer Form und braucht daher nicht weiter reduziert zu werden. Auch eine Applikation, welche zur Freisetzung des Teilterms `(λx.x x)(λx.x x)` führen würde, ist nicht durchführbar, da hierfür ein Term $t \in$void anstelle von `z` eingesetzt werden müßte. Einen solchen Term aber kann es nicht geben.

Durch den leeren Datentyp void wird somit der Unterschied zwischen Variablen eines Typs und Termen desselben Typs besonders deutlich. Die Beschreibung einer Funktion $f \equiv \lambda x.b \in S{\rightarrow}T$ suggeriert zwar, daß f Werte $b[x]$ vom Typ T produziert, aber dies kann auch ein Trugschluß sein. Ist nämlich die Eingabevariable vom Typ $S \equiv$void, so gibt es keine Möglichkeit, für x einen Wert einzusetzen.[44]

[43] Terme wie `λT.λz.(λx.x x)(λx.x x)` tauchen allerdings nicht von selbst in einem Beweis auf. Es ist notwendig, sie explizit als Teil des Beweisziels oder mit der Regel `intro` in den Beweis hineinzunehmen. Wenn wir unsere Theorie auf Terme einschränken, welche als Extrakt-Terme in Beweisen *generiert* werden können, so haben wir nach wie vor die starke Normalisierbarkeit.

[44] Dieser Unterschied erklärt auch die zum Teil sehr vorsichtige und kompliziert erscheinende Beweisführung zu den Eigenschaften der einfachen Typentheorie im Abschnitt 2.4.5. Will man diese Beweise auf allgemeinere Konstrukte übertragen, so muß man berücksichtigen, daß Variablen auch zu einem leeren Typ gehören können.


```
⊢ λT.λz.(λx.x x)(λx.x x) ∈ T:U1→void→T
by lambdaEq 2
|\
| T:U1 ⊢ λz.(λx.x x)(λx.x x) ∈ void→T
| by lambdaEq 1
| |\
| | T:U1, z:void ⊢ (λx.x x)(λx.x x) ∈ T
| | by voidE 2
| \
|  T:U1 ⊢ void→T ∈ U1
|  by functionEq
|  |\
|  | T:U1 ⊢ void ∈ U2
|  | by voidEq
|   \
|    T:U1, x:void ⊢ T ∈ U1
|    by hypEq 1
 \
  T:U1→void→T ∈ U2
  by functionEq
  |\
  | ⊢ U1 ∈ U2
  | by univEq
   \
    T:U1 ⊢ void→T ∈ U2
    by cumulativity 1
     \
      T:U1 ⊢ void→T ∈ U1
      by functionEq
      |\
      | T:U1 ⊢ void ∈ U2
      | by voidEq
       \
        T:U1, x:void ⊢ T ∈ U1
        by hypEq 1
```

Abbildung 3.15: Typisierungsbeweis für `λT.λz.(λx.x x)(λx.x x)`

3.3.3 Konstruktive Logik

Die Hinzunahme des Datentyps void liefert uns das letzte noch fehlende Konstrukt das wir für eine Einbettung der Prädikatenlogik in die Typentheorie mittels des Prinzips "Propositionen als Datentypen" benötigen. Die Regel `voidE` deckt sich mit der entsprechenden Regel `false_e` für die logische Falschheit und damit können alle Regeln des analytischen Sequenzenkalküls für die Prädikatenlogik (siehe Abbildung 2.6 auf Seite 43) als Spezialfälle typentheoretischer Regeln betrachtet werden. Deshalb sind wir in der Lage, die Prädikatenlogik als *konservative Erweiterung* (siehe Abschnitt 2.1.5) zu der bisher definierten Theorie hinzuzunehmen und können auf eine explizite Definition von Semantik und Inferenzregeln verzichten. Zu diesem Zweck führen wir die folgenden definitorischen Abkürzungen ein.[45]

Definition 3.3.3 (Logikoperatoren)

$A \wedge B$	$\equiv$ **and**$\{\}(A;B)$	$\equiv A{\times}B$
$A \vee B$	$\equiv$ **or**$\{\}(A;B)$	$\equiv A{+}B$
$A \Rightarrow B$	$\equiv$ **implies**$\{\}(A;B)$	$\equiv A{\rightarrow}B$
$\neg A$	$\equiv$ **not**$\{\}(A)$	$\equiv A{\rightarrow}$void
Λ	$\equiv$ **false**$\{\}()$	$\equiv$ void
$\forall x{:}T.A$	$\equiv$ **all**$\{\}(T;x.A)$	$\equiv x{:}T{\rightarrow}A$
$\exists x{:}T.A$	$\equiv$ **exists**$\{\}(T;x.A)$	$\equiv x{:}T{\times}A$
$\mathbb{P}_i$	$\equiv$ **prop**$\{i{:}1\}()$	$\equiv U_i$

[45] Im Prinzip sind derartige Abkürzungen nichts anderes als textliche Ersetzungen. Innerhalb des NuPRL Systems sind sie allerdings gegen ungewollte Auflösung geschützt. Man muß eine Definition explizit "auffalten" mit der Regel <u>`Unfold` *'name'* *i*</u>, wobei *name* der name der Definition ist (meist der Operatorname) und *i* die Hypothese, in der die Definition aufgelöst werden soll. Die Konklusion wird als Hypothese 0 angesehen. Eine eventuell nötige Rückfaltung wird mit <u>`Fold` *'name'* *i*</u> erzeugt. Die logischen Regeln aus Abbildung 2.6 wurden mit diesen zwei Regeln und den entsprechenden typentheoretischen Regeln simuliert.

Wir wollen uns nun die logischen Konsequenzen dieser Definitionen etwas genauer ansehen. Als erstes ist festzustellen, daß das natürliche Gegenstück der typentheoretischen Konstrukte die *intuitionistische* Prädikatenlogik ist. Alle Regeln des intuitionistischen Sequenzenkalküls haben ein natürliches Gegenstück in der Typentheorie, aber die klassische Negationsregel `classical_contradiction` gilt im Allgemeinfall nicht, da die Formel $\vdash\ \neg(\neg A)\ \Rightarrow\ A$ nicht für alle Formeln A beweisbar ist. Ein Beweisversuch für das typentheoretische Gegenstück – also $\vdash\ ((A\to\texttt{void})\to\texttt{void})\to A$ – würde erfolglos stecken bleiben.

> Der einzig mögliche erste Schritt wäre `lambdaI 1`, was zu $\texttt{f}:(A\to\texttt{void})\to\texttt{void}\ \vdash\ A$ führt. Da A beliebig sein soll, können wir kein Element von A direkt angeben. Wir müssen also `functionE 1` s anwenden und hierzu als Argument für f einen Term $s\in A\to\texttt{void}$ angeben. Dies aber ist nicht möglich, da $A\to\texttt{void}$ – wie oben argumentiert – leer sein muß, wenn A Elemente haben (also wahr sein) soll. Damit bleibt uns keine Möglichkeit, den Beweis ohne spezielle Kenntnisse über A zu Ende zu führen.

Die Curry-Howard Isomorphie und das Prinzip "Propositionen als Datentypen" führen also zu einer konstruktiven Logik, obwohl wir uns bei der Erklärung der Semantik der Typentheorie nicht auf eine intuitionistische Denkweise abgestützt haben. Die Tatsache, daß wir zur Semantikdefinition nur Elemente benutzt haben, die man explizit angeben kann, reicht aus, der Typentheorie einen konstruktiven Aspekt zu verleihen.

Satz 3.3.4

> *Die Logik, welche durch das Prinzip "Propositionen als Datentypen" definiert wird, ist die intuitionistische Prädikatenlogik* ($\underline{\mathcal{J}}$).

Aufgrund von Theorem 3.3.4 sind wir in der Lage, die Eigenschaften der intuitionistischen Prädikatenlogik $\mathcal{J}$ und des intuitionistischen Sequenzenkalküls $\mathcal{LJ}$ mit den Mitteln der Typentheorie genauer zu untersuchen. Einige sehr interessante Metatheoreme über $\mathcal{LJ}$, die ursprünglich recht aufwendig zu beweisen waren, erhalten dadurch erstaunlich einfache Beweise. Wir werden hierfür einige Beispiele geben.

Die Schnittregel `cut` ist im Sequenzenkalkül ein sehr wichtiges Hilfsmittel zur Strukturierung von Beweisen durch Einfügung von Zwischenbehauptungen (Lemmata). Für eine interaktive Beweiskonstruktion ist diese Regel unumgänglich. Für eine automatische Suche nach Beweisen stellt sie jedoch ein großes Hindernis dar. Außerdem taucht diese Regel im Kalkül des natürlichen Schließens nicht auf, was den Beweis der Äquivalenz der beiden Kalküle erheblich erschwert. Es ist jedoch möglich, Anwendungen der Schnittregel aus Beweisen zu eliminieren, wobei man allerdings ein eventuell exponentielles Wachstum der Beweise in Kauf nehmen muß. Dieser in der Logik sehr aufwendig zu beweisende Satz (Genztens *Hauptsatz* in [Gentzen, 1935]) läßt sich in der Typentheorie durch eine einfache Verwendung der Reduktion beweisen.

Satz 3.3.5 (Schnittelimination)

> *Wenn* $\Gamma\ \vdash\ C$ *innerhalb von* $\mathcal{LJ}$ *bewiesen werden kann, dann gibt es auch einen Beweis ohne Verwendung der Schnittregel.*

Beweis: Wenn wir C als Typ betrachten und innerhalb der Typentheorie mit den zu $\mathcal{LJ}$ korrespondierenden Regeln beweisen, dann können wir aus dem Beweis einen Term t extrahieren, der genau beschreibt, welche Regeln zum Beweis von C in welcher Reihenfolge angewandt werden müssen. Da für extrahierte Terme innerhalb der Typentheorie starke Normalisierbarkeit gilt (siehe Fußnote Seite 133), können wir t so lange reduzieren, bis keine Redizes mehr vorhanden sind. Der resultierende Term t' liefert ebenfalls einen Beweis für C, der aber keine Schnittregel `cut` mehr enthalten kann, da diese im Extrakt-Term Teilterme der Art $(\lambda x.u)\ s$ erzeugt. □

Das Schnitteliminationstheorem erleichtert auch einen Beweis für die *Konsistenz* des Kalküls $\mathcal{LJ}$, also die Tatsache, daß nicht gleichzeitig eine Aussage und ihr Gegenteil beweisen werden kann.[46]

Satz 3.3.6

> $\mathcal{LJ}$ *ist konsistent.*

[46] Die logische Konsistenz eines Kalküls kann unabhängig von einer konkreten Semantik überprüft werden und ist daher etwas schwächer als *Korrektheit* (*soundness*). Ist der Kalkül allerdings auch *vollständig*, so fallen die beiden Begriffe zusammen.

Beweis: Wenn es auch nur eine einzige Formel A gäbe, für die in $\mathcal{LJ}$ sowohl ein Beweis für $\vdash\ A$ als auch für $\vdash\ \neg A$ geführt werden könnte, dann könnte man $\vdash\ \Lambda$ wie folgt beweisen:

```
⊢ Λ
by cut 1 A
|\
| ⊢ A
| by  – fester Beweis –
 \
  x:A ⊢ Λ
  by cut 2 ¬A
  |\
  | ⊢ ¬A
  | by  – fester Beweis –
   \
    x:A, y:¬A ⊢ Λ
    by functionE_indep 2           (Unfold 'not' 2)
    |\
    | x:A, y:¬A ⊢ A
    | by hypEq 1
     \
      x:A, y:¬A, z:void ⊢ Λ
      by voidE 3                   (Unfold 'false' 0)
```

Nach Theorem 3.3.5 gäbe es dann aber auch einen Beweis für $\vdash\ \Lambda$, der ohne Schnitte auskommt. Da jedoch alle anderen Regeln auf dieses Ziel nicht anwendbar sind (es gibt keine Regel `falseI`), kann dies nicht stimmen. Also kann es keine einander widersprechende Aussagen geben, die beide in $\mathcal{LJ}$ beweisbar sind. □

Die Isomorphie zwischen der Typentheorie und der konstruktiven Logik ist übrigens nicht beschränkt auf Logik erster Stufe. Die Kombination von Universenhierarchie und abhängigem Funktionenraum ermöglicht die formale Darstellung aller Quantoren höherer Stufe. Insbesondere erhalten wir

- Einen Funktionalkalkül höherer Ordnung, in dem über beliebige Funktionen quantifiziert werden darf wie zum Beispiel in $\forall \texttt{f}:S\to T.\ t[\texttt{f}]$. (Stufe ω)

- Die Prädikatenlogik beliebiger fester Stufe `i+1`, in der über beliebige Prädikatsymbole der Stufe `i` quantifiziert werden darf wie zum Beispiel in $\forall \texttt{P}:\mathbb{P}_{\texttt{i}}.\ t[\texttt{P}]$.[47]

3.3.4 Klassische Logik

Soweit die intuitionistische Logik. Wie aber sieht es mit der klassischen Logik aus, in der Theoreme bewiesen werden können, die nicht unbedingt konstruktiv gelten? Können wir diese ebenfalls in der Typentheorie darstellen? Die Lösung hierfür ist einfach, da die klassische Logik sich in die intuitionistische einbetten läßt, indem wir einfach jede Formel durch doppelte Negation ihres konstruktiven Anteils berauben. Wir definieren hierzu die klassischen logischen Operatoren wie folgt.

Definition 3.3.7 (Klassische Logikoperatoren)

Λ	$\equiv$ **false**$\{\}()$	$\equiv$ void
$\neg A$	$\equiv$ **not**$\{\}(A)$	$\equiv A\to$void
$A\wedge B$	$\equiv$ **and**$\{\}(A;B)$	$\equiv A\times B$
$\forall x:T.A$	$\equiv$ **all**$\{\}(T;x.A)$	$\equiv x:T\to A$
$A\vee_c B$	$\equiv$ **orc**$\{\}(A;B)$	$\equiv \neg(\neg A\wedge\neg B)$
$A\Rightarrow_c B$	$\equiv$ **impc**$\{\}(A;B)$	$\equiv \neg A\vee_c B$
$\exists_c x:T.A$	$\equiv$ **exc**$\{\}(T;x.A)$	$\equiv \neg(\forall x:T.\neg A)$

[47] Es gibt Versuche, dies auf Stufe ω auszuweiten, in der die Stufe des Prädikats nicht mehr verwaltet werden muß. Diese bergen jedoch einige Schwierigkeite in sich.

Wie man sieht, bleiben Negation, Konjunktion und Allquantor unverändert während die anderen drei Operatoren implizit schon die nichtkonstruktive Sicht wiederspiegeln. Mit dieser Übersetzung der klassischen Operatoren in die intuitionistischen, deren homomorphe Fortsetzung auf Formeln und Sequenzen wir mit $^\circ$ bezeichnen (die sogenannte *Gödel-Transformation*), läßt sich folgendes Einbettungstheorem beweisen.

Satz 3.3.8

Es sei $'$ die Transformation, welche in einer Formel bzw. Sequenz jede atomare Teilformel A durch $\neg\neg A$ ersetzt. Wenn innerhalb des klassischen Sequenzenkalküls die Sequenz $\Gamma \vdash C$ bewiesen werden kann, dann gibt es für $\Gamma^{\circ\prime} \vdash C^{\circ\prime}$ einen Beweis im intuitionistischen Sequenzenkalkül $\mathcal{LJ}$.

Statt eines Beweises von Theorem 3.3.8 wollen wir die Gültigkeit der (transformierten) klassischen Regel `classical_contradiction` nachweisen, welche die Grundlage aller Widerspruchsbeweise ist.

```
Γ ⊢ C°'    by classical_contradiction
     Γ, ¬C°' ⊢ Λ
```

Wir skizzieren einen $\mathcal{LJ}$-Beweis für den Fall, daß C atomar – also $C^{\circ\prime} \equiv \neg\neg C$ – ist.

```
[Γ ⊢ C°']
by not_i                                (mit Unfold)
 \
  Γ, ¬C ⊢ Λ
  by cut -1 ¬C°'
  |\
  | Γ, ¬C ⊢ ¬C°'
  | by not_i                            (mit Unfold)
  |  \
  |   Γ, ¬C, ¬¬C ⊢ Λ
  |   by not_e -1
  |   |\
  |   | Γ, ¬C, ¬¬C ⊢ ¬C
  |   | by hypothesis -2
  |    \
  |     Γ, ¬C, Λ ⊢ Λ
  |     by false_e -1
   \
    Γ, ¬C, ¬C°' ⊢ Λ
    by thin -1
    [Γ, ¬C°' ⊢ Λ]
```

Auf ähnliche Weise ließe sich übrigens das Gesetz vom Ausgeschlossenen Dritten $\vdash C \vee_c \neg C$ für beliebige Formeln C direkt nachweisen, was die Transformation $'$ in den meisten Fällen überflüssig macht.

Damit kann auch die klassische Logik in der Typentheorie behandelt werden. Man muß sich bei der Wahl der logischen Operatoren allerdings festlegen, ob man ein klassisches oder ein intuitionistisches Verständnis der Formel anlegen möchte. Im Gegensatz zu anderen Kalkülen sind Mischformen ebenfalls möglich.

3.4 Programmierung in der Typentheorie

Bei unseren bisherigen Betrachtungen der Typentheorie haben stand vor allem die Ausdruckskraft im Vordergrund. Mit den vorgestellten Konstrukten können wir einen Großteil der Mathematik und Programmierung formal repräsentieren und Eigenschaften von Programmen und mathematischen Konstrukten formal nachweisen. Wir wollen nun beginnen, Konzepte zu diskutieren, welche eine effiziente *Anwendung* der Typentheorie unterstützen. Hierzu gehört insbesondere der Aspekt der *Programmentwicklung*, den wir in diesem Abschnitt besprechen wollen.

$T \in \mathsf{U}_j$	$t \in T$	T $\lfloor$**ext** $t\rfloor$
T ist ein Typ (eine Menge)	t ist ein Element der Menge T	T ist nicht leer *(inhabited)*
T ist eine Proposition	t ist ein(e) Beweis(konstruktion) für T	T ist wahr (beweisbar)
T ist eine Intention	t ist eine Methode, um T zu erfüllen	T ist erfüllbar (realisierbar)
T ist ein Problem (Aufgabe)	t ist eine Methode, um T zu lösen	T ist lösbar

Abbildung 3.16: Interpretationen der typentheoretischen Urteile

3.4.1 Beweise als Programme

Für die typentheoretischen Urteile bzw. ihre Darstellung durch Sequenzen kann man – je nachdem, welche Problemstellung man mit formalen Theorien angehen möchte – eine Reihe verschiedener semantischer Interpretationen geben. Wir haben die wichtigsten Sichtweisen in Abbildung 3.16 zusammengestellt. Die erste Zeile beschreibt die Lesart, die wir bei der Einführung der Typentheorie als Motivation verwandt haben: Typen sind Mengen mit Struktur und Elementen. Die zweite Zeile entspricht einer *logischen Sicht*, die – unterstützt durch die Curry-Howard Isomorphie – zu dem Prinzip "Propositionen als Datentypen" geführt hatte. Die dritte, eher philosophische Sichtweise geht zurück auf Heyting [Heyting, 1931] und soll hier nicht vertieft werden.

Die letzte Interpretation wurde von Kolmogorov [Kolmogorov, 1932] vorgeschlagen, der die Mathematik als eine Sammlung von Problemstellungen verstand, für die es *Lösungsmethoden* zu konstruieren galt. In der Denkweise der Programmierung würde man heute den Begriff "*Spezifikation*" anstelle von "Problem" und das Wort "*Programm*" anstelle von "Methode" verwenden. Damit würde '$t \in T$' interpretiert als "t ist ein Programm, welches die Spezifikation T löst" und das Beweisziel '$\vdash T$ $\lfloor$**ext** $t\rfloor$' könnte verstanden werden als die *Aufgabe, ein Programm zu konstruieren*, welches eine gegebene Spezifikation T erfüllt. Der Kalkül der Typentheorie läßt sich daher prinzipiell zur Konstruktion von Programmen aus ihren Spezifikationen verwenden. Wir wollen nun untersuchen, wie dies geschehen kann, und betrachten hierzu ein einfaches Beispiel.

Beispiel 3.4.1

Wir nehmen einmal an, wir hätten innerhalb der Typentheorie ein Theorem beweisen, welches besagt, daß jede natürliche Zahl eine ganzzahlige Quadratwurzel besitzt.[48]

$$\vdash \forall \mathtt{x}{:}\mathbb{N}.\exists \mathtt{y}{:}\mathbb{N}.\ \mathtt{y}^2 \mathtt{\le x} \ \wedge\ \mathtt{x<(y+1)}^2$$

Dann liefert der Beweis dieses Theorems einen Extrakt-Term p, den wir als Evidenz für seine Gültigkeit verstehen können. Setzen wir nun anstelle der logischen Operatoren die entsprechenden Typkonstruktoren ein, durch die sie definiert wurden (vergleiche Definition 3.3.3), so sehen wir, daß p den Typ

$$\mathtt{x}{:}\mathbb{N} \to \mathtt{y}{:}\mathbb{N} \times \mathtt{y}^2 \mathtt{\le x} \times \mathtt{x<(y+1)}^2$$

hat – wobei wir für $\le$ und `<` vorerst keine Interpretation geben. Damit ist p also eine Funktion, die eine natürliche Zahl n in ein Tupel $\langle y, \langle pf_1, pf_2\rangle\rangle$ abbildet, wobei y die Integerquadratwurzel von n ist und pf_1 sowie pf_2 Beweise für $y^2 \le n$ bzw. $n\mathtt{<}(y\mathtt{+1})^2$ sind. p enthält also sowohl den Berechnungsmechanismus für die Integerquadratwurzel als auch einen allgemeinen Korrektheitsbeweis für diesen Mechanismus. Beide Anteile kann man voneinander durch Anwendung des `spread`-Operators trennen und erhält somit ein allgemeines Programm zur Berechnung der Integerquadratwurzel durch

$$\mathtt{sqrt} \ \equiv\ \lambda\mathtt{n}.\ \mathsf{let}\ \langle\mathtt{y},\mathtt{pf}\rangle = p\,\mathtt{n}\ \mathsf{in}\ \mathtt{y}.$$

Die Korrektheit dieses Programms folgt unmittelbar aus dem obigen Theorem.

Dies bedeutet also, daß uns ein Beweis für eine mathematische Aussage der Form $\forall x{:}T.\ \exists y{:}S.\ spec[x,y]$ – ein sogenanntes *Spezifikationstheorem* – automatisch ein Programm liefert, welches die Spezifikation *spec* erfüllt. Wir können die Typentheorie daher nicht nur für die mathematische Beweisführung sondern auch für die Entwicklung von Programmen mit garantierter Korrektheit verwenden. Diese Tatsache läßt sich sogar *innerhalb* der Typentheorie nachweisen: wenn ein Spezifikationstheorem der Form $\forall x{:}T.\ \exists y{:}S.\ spec[x,y]$ beweisbar ist, dann gibt es auch eine Funktion $f{:}T{\to}S$, welche die Spezifikarion *spec* erfüllt.

[48] Die hierzu nötigen Konstrukte werden wir im Abschnitt 3.4.2 und einen Beweis im Beispiel 3.4.9 auf Seite 154 vorstellen.

Satz 3.4.2 (Konstruktives Auswahlaxiom / Beweise als Programme)

Für alle Universen U_j *kann das folgende Theorem in der Typentheorie beweisen werden.*

$$\vdash \forall\mathtt{T}{:}\mathsf{U}_j.\forall\mathtt{S}{:}\mathsf{U}_j.\forall\mathtt{spec}{:}\,\mathtt{T}{\times}\mathtt{S}{\to}\mathbb{P}_j.\ (\forall\mathtt{x}{:}\mathtt{T}.\ \exists\mathtt{y}{:}\mathtt{S}.\ \mathtt{spec}\langle\mathtt{x},\mathtt{y}\rangle)\ \Rightarrow\ \exists\mathtt{f}{:}\mathtt{T}{\to}\mathtt{S}.\forall\mathtt{x}{:}\mathtt{T}.\ \mathtt{spec}\langle\mathtt{x},\mathtt{f\ x}\rangle$$

Beweis: Der formale Beweis spiegelt genau das Argument aus Beispiel 3.4.1 wieder.
Der Übersichtlichkeit halber werden wir ihn nur skizzieren und dabei Wohlgeformtheitsziele unterdrücken.

```
⊢ ∀T:U_j.∀S:U_j.∀spec:T×S→IP_j. (∀x:T.∃y:S.spec⟨x,y⟩)  ⇒  ∃f:T→S.∀x:T.spec⟨x,f x⟩
by Repeat all_i                                              (d.h. function_i -- 3 mal)
 \
  T:U_j, S:U_j, spec:T×S→IP_j ⊢ (∀x:T.∃y:S.spec⟨x,y⟩)  ⇒  ∃f:T→S.∀x:T.spec⟨x,f x⟩
  by imp_i                                                   (d.h. function_i  mit Variable)
   \
    T:U_j, S:U_j, spec:T×S→IP_j, p:(∀x:T.∃y:S.spec⟨x,y⟩) ⊢ ∃f:T→S.∀x:T.spec⟨x,f x⟩
    by ex_i λx.let ⟨y,pf⟩=p x in y                            (d.h. pair_i )
    |\
    | T:U_j, S:U_j, spec:T×S→IP_j, p:(∀x:T.∃y:S.spec⟨x,y⟩) ⊢ λx.let ⟨y,pf⟩=p x in y ∈ T→S
    |  by  – Wohlgeformtheitsbeweis –
     \
      T:U_j, S:U_j, spec:T×S→IP_j, p:(∀x:T.∃y:S.spec⟨x,y⟩)
      ⊢ ∀x:T.spec⟨x,(λx.let ⟨y,pf⟩=p x in y) x⟩
      by  – explizite Reduktion, siehe Abschnitt 3.6
       \
        T:U_j, S:U_j, spec:T×S→IP_j, p:(∀x:T.∃y:S.spec⟨x,y⟩) ⊢ ∀x:T.spec⟨x,let ⟨y,pf⟩=p x in y⟩
        by all_i
         \
          T:U_j, S:U_j, spec:T×S→IP_j, p:(∀x:T.∃y:S.spec⟨x,y⟩), x:T ⊢ spec⟨x,let ⟨y,pf⟩=p x in y⟩
          by intro let ⟨y,pf⟩=p x in pf                   (Siehe Abbildung 3.13, Seite 126)
           \
            T:U_j, S:U_j, spec:T×S→IP_j, p:(∀x:T.∃y:S.spec⟨x,y⟩), x:T
          ⊢ let ⟨y,pf⟩=p x in pf  ∈  spec⟨x,let ⟨y,pf⟩=p x in y⟩
            by spreadEq  z  spec⟨x,let ⟨y,pf⟩=z in y⟩  y:S × spec⟨x,y⟩
            |\
            | T:U_j, S:U_j, spec:T×S→IP_j, p:(∀x:T.∃y:S.spec⟨x,y⟩), x:T ⊢ p x ∈ y:S × spec⟨x,y⟩
            | by  – Wohlgeformtheitsbeweis mit Hypothese 4 –
             \
              T:U_j, S:U_j, spec:T×S→IP_j, p:(∀x:T.∃y:S.spec⟨x,y⟩), x:T, s:S,
                t:spec⟨x,s⟩, z:p x=⟨s,t⟩ ∈ y:S × spec⟨x,y⟩ ⊢ t ∈ spec⟨x,let ⟨y,pf⟩=⟨s,t⟩ in y⟩
              by  – explizite Reduktion, siehe Abschnitt 3.6
               \
                T:U_j, S:U_j, spec:T×S→IP_j, p:(∀x:T.∃y:S.spec⟨x,y⟩), x:T, s:S,
                  t:spec⟨x,s⟩, z:p x=⟨s,t⟩ ∈ y:S × spec⟨x,y⟩ ⊢ t ∈ spec⟨x,s⟩
                by hypEq 7
```

□

In der Mengentheorie ist die Aussage des Theorems 3.4.2 als eine Variante des *Auswahlaxioms* bekannt geworden. In keiner Formulierung der (nichtkonstruktiven) Mengentheorie (z.B. Zermelo-Fraenkel) ist es möglich, diese Aussage zu beweisen, und man müß sie deshalb immer als Axiom zur Theorie hinzunehmen. Die Rechtfertigung dieses Axioms ist allerdings nicht sehr einfach. In der Typentheorie ist die Aussage dagegen ein Theorem, welches unmittelbar aus der Darstellung von Propositionen als Datentypen folgt und uns eine generelle Methode zur Entwicklung von Programmen aus gegebenen Spezifikationen liefert.

Entwurfsprinzip 3.4.3 (Beweise als Programme)

In der Typentheorie ist Programmentwicklung dasselbe wie Beweisführung

Dieses Prinzip, welches unter dem Namen "*proofs as programs*" [Bates & Constable, 1985] bekannt geworden ist, ermöglicht es, auf systematische Weise Programme zu entwickeln, die *garantiert korrekt sind*: man muß einfach nur einen Beweis für ein Spezifikationstheorem führen und hieraus ein Programm extrahieren. Auf den ersten Blick scheint diese Methode unnatürlich und sehr ineffizient zu sein – was hat Programmierung

mit dem Nachweis der Erfüllbarkeit einer Spezifikation zu tun? Es hat sich jedoch herausgestellt, daß nahezu alle Programmiermethoden, die von Systematikern wie Hoare [Hoare, 1972], Dijkstra [Dijkstra, 1976] und anderen aufgestellt wurde, im wesentlichen nichts anderes sind als spezielle Neuformulierungen von bekannten Beweismethoden. Daher ist es durchaus legitim, Programmierung als eine Art Spezialfall von Beweisführung anzusehen und Entwurfstechniken, die ursprünglich von Methodikern wie Polya [Polya, 1945] als mathematische Methoden für Beweise vorgeschlagen wurden, auf die Programmentwicklung zu übertragen.[49] Hierfür ist es allerdings notwendig, Konstrukte in die Typentheorie hineinzunehmen, die notwendig sind, um "echte" Mathematik und Programmierung zu beschreiben. Dies wollen wir nun in den folgenden Abschnitten tun.

3.4.2 Zahlen und Induktion

In der praktischen Mathematik und den meisten "realen" Programmen spielen arithmetische Konstrukte eine zentrale Rolle. Aus diesem Grunde ist es notwendig, das Konzept der Zahl und die elementare Arithmetik – einschließlich der primitiven Rekursion und der Möglichkeit einer induktiven Beweisführung – in die Typentheorie mit aufzunehmen. Wir hatten in Abschnitt 2.3.3 auf Seite 54 bereits gezeigt, daß natürliche Zahlen prinzipiell durch Church Numerals simuliert werden können. Eine solche Simulation würde aber bereits die einfachsten Konstrukte extrem aufwendig werden lassen und ein formales Schließen über Arithmetik praktisch unmöglich machen. Zugunsten einer effizienten Beweisführung ist es daher notwendig, Zahlen und die wichtigsten Grundoperationen sowie die zugehörigen Inferenzregeln explizit einzuführen.

Wir werden dies in zwei Phasen tun. Zunächst werden wir eine eher spartanische Variante der Arithmetik – die *primitiv rekursive Arithmetik* PRA – betrachten, die nur aus den natürlichen Zahlen, der Null, der Nachfolgerfunktion und der primitiven Rekursion besteht. Dies ist aus theoretischer Sicht der einfachste Weg, die Typentheorie um eine Arithmetik zu erweitern, und ermöglicht es, die Grundeigenschaften einer solchen Erweiterung gezielt zu erklären. Aus praktischen Gesichtspunkten ist es jedoch sinnvoller, die Typentheorie um eine erheblich umfangreichere Theorie der Arithmetik – die konstruktive Peano Arithmetik oder *Heyting Arithmetik* HA – zu erweitern. Diese ist bezüglich ihrer Ausdruckskraft genauso mächtig wie die primitiv rekursive Arithmetik. Jedoch werden ganze Zahlen anstelle der natürlichen Zahlen betrachtet, die Zahlen wie $\mathtt{0},\mathtt{1},\mathtt{2},\ldots$ und eine Fülle von elementaren arithmetischen Operationen wie $\mathtt{+},\mathtt{-},\mathtt{*},\div,\mathsf{rem},\mathtt{<}$ und arithmetische Fallunterscheidungen sind bereits vordefiniert und das Regelsystem ist entsprechend umfangreich. Die Hinzunahme von Zahlen in die Typentheorie von NuPRL gestaltet sich daher verhältnismäßig aufwendig.

3.4.2.1 Die primitiv rekursive Arithmetik

Die primitiv rekursive Arithmetik besteht aus den minimalen Bestandteilen, die zum Aufbau einer arithmetischen Theorie notwendig sind. Als Typkonstruktor wird der Typ $\underline{\mathbb{N}}$[50] der natürlichen Zahlen eingeführt, der als kanonische Elemente die Null ($\underline{\mathtt{0}}$) besitzt und alle Elemente, die durch Anwendung der Nachfolgerfunktion $\underline{\mathsf{s}}{:}\mathbb{N}{\to}\mathbb{N}$ auf kanonische Elemente entstehen. Die zugehörige nichtkanonische Form analysiert den induktiven Aufbau einer natürlichen Zahl aus $\mathtt{0}$ und s und baut hieraus rekursiv einen Term zusammen: der Term $\underline{\mathsf{ind}(n;\,base;\,m,f_m.t)}$ beschreibt die Werte einer Funktion f, die bei Eingabe der Zahl n=$\mathtt{0}$ das Resultat *base* und bei Eingabe von n=$\mathsf{s}(m)$ den Term $t[m,f_m]$ liefert, wobei f_m ein Platzhalter für $f(m)$ ist. Dieser nichtkanonische Term ist also ein typentheoretisches Gegenstück zur primitiven Rekursion.

Die Semantik der obengenannten Terme ist relativ leicht zu formalisieren. Entsprechend der intuitiven Erklärung gibt es zwei Arten, den Induktionsterm zu reduzieren.

$$\begin{array}{lcl} \mathsf{ind}(\mathtt{0};\,base;\,m,f_m.t) & \xrightarrow{\beta} & base \\ \mathsf{ind}(\mathsf{s}(n);\,base;\,m,f_m.t) & \xrightarrow{\beta} & t[n,\mathsf{ind}(n;\,base;\,m,f_m.t)\ /\ m,\ f_m] \end{array}$$

[49] Dieser Zusammenhang wurde von David Gries [Gries, 1981] herausgestellt. Ein Beispiel für die Vorteile dieser Sichtweise haben wir bereits in Abschnitt 1.1 vorgestellt.

[50] Auf die Darstellung in der einheitlichen Syntax wollen wir an dieser Stelle verzichten, da wir anstelle der natürlichen Zahlen die ganzen Zahlen als Grundkonstrukt zur Typentheorie hinzunehmen. Die natürlichen Zahlen können hieraus mit dem Teilmengenkonstruktor, den wir im Abschnitt 3.4.4 vorstellen werden, als definitorische Erweiterung gebildet werden.

Unter den kanonischen Termen ist $\mathbb{N}$ ein Typ, der nur identisch ist mit sich selbst (d.h. es gilt das Urteil '$\mathbb{N}$=$\mathbb{N}$') und $\mathtt{0}$ ein Element von $\mathbb{N}$, das nur mit sich selbst identisch ist. Die Nachfolger zweier natürlicher Zahlen n und m sind genau dann gleich, wenn n und m semantisch gleich sind. In den formalen Semantiktabellen wären also die folgenden Einträge zu ergänzen.

Typsemantik	
$\mathbb{N} = \mathbb{N}$	

Elementsemantik	
$\mathtt{0} = \mathtt{0} \in \mathbb{N}$	
$\mathbf{s}(n) = \mathbf{s}(m) \in \mathbb{N}$	falls $n = m \in \mathbb{N}$
$\mathbb{N} = \mathbb{N} \in \mathsf{U}_j$	

Man beachte, daß hieraus folgt, daß $\mathbf{s}(n)$=$\mathtt{0} \in \mathbb{N}$ *nicht* gelten kann, da es keinen Tabelleneintrag gibt, der dises Urteil unterstützt. Ebenso gilt, daß aus $\mathbf{s}(n) = \mathbf{s}(m) \in \mathbb{N}$ auch $n = m \in \mathbb{N}$ folgen muß, weil es keinen anderen Weg gibt, die Gleichheit von $\mathbf{s}(n)$ und $\mathbf{s}(m)$ semantisch zu begründen. Diese beiden Grundeigenschaften der natürlichen Zahlen müssen daher nicht explizit in den Tabellen aufgeführt werden.
Die zugehörigen Inferenzregeln müssen die soeben gegebene Semantik respektieren.

- Als Formationsregel erhalten wir somit, daß $\mathbb{N}$ ein Typ jeden Universums ist, da an die Typeigenschaft von $\mathbb{N}$ ja keine Anforderungen gestellt wurden.

  $\Gamma \vdash \mathbb{N} \in \mathsf{U}_j \;\lfloor\mathsf{Ax}\rfloor$
  by `natEq`

- Die kanonischen Gleichheitsregeln spiegeln genau die obigen Gleichheitsurteile wieder. Die entsprechenden impliziten Varianten zur Erzeugung kanonischer Terme ergeben sich direkt aus diesen.

  $\Gamma \vdash \mathtt{0} \in \mathbb{N} \;\lfloor\mathsf{Ax}\rfloor$
  by `zeroEq`

  $\Gamma \vdash \mathbb{N} \;\lfloor\textbf{ext }\mathtt{0}\rfloor$
  by `zeroI`

  $\Gamma \vdash \mathbf{s}(t_1) = \mathbf{s}(t_2) \in \mathbb{N} \;\lfloor\mathsf{Ax}\rfloor$
  by `sucEq`
  $\quad\Gamma, \vdash t_1 = t_2 \in \mathbb{N} \;\lfloor\mathsf{Ax}\rfloor$

  $\Gamma \vdash \mathbb{N} \;\lfloor\textbf{ext }\mathbf{s}(t)\rfloor$
  by `sucI`
  $\quad\Gamma \vdash \mathbb{N} \;\lfloor\textbf{ext }t\rfloor$

- In den nichtkanonischen Regeln legen wir fest, daß zwei Induktionsterme nur gleich sein können, wenn die Hauptargumente gleich sind, die Basisfälle übereinstimmen und die rekursive Abarbeitung sich gleich verhält. Dies deckt natürlich nicht alle Fälle ab, denn die Gleichheit kann ja auch aus anderen Gründen gelten – so wie zum Beispiel $4{+}7 = 2{+}9$ ist. Derartige Gleichheiten hängen jedoch von den konkreten Werten ab und können nur durch Auswertung der Argumente und Reduktion (siehe unten) nachgeweisen werden. Die implizite Variante ergibt sich entsprechend.

  $\Gamma \vdash \mathsf{ind}(n_1; base_1; m_1, f_1.t_1)$
  $\quad = \mathsf{ind}(n_2; base_2; m_2, f_2.t_2) \in T[n_1/z] \;\lfloor\mathsf{Ax}\rfloor$
  by `indEq` z T
  $\quad\Gamma \vdash n_1 = n_2 \in \mathbb{N} \;\lfloor\mathsf{Ax}\rfloor$
  $\quad\Gamma \vdash base_1 = base_2 \in T[\mathtt{0}/z] \;\lfloor\mathsf{Ax}\rfloor$
  $\quad\Gamma, m{:}\mathbb{N}, f{:}T[m/z]$
  $\quad\quad\vdash t_1[m, f / m_1, f_1] = t_2[m, f / m_2, f_2]$
  $\quad\quad\quad\in T[\mathbf{s}(m)/z] \;\lfloor\mathsf{Ax}\rfloor$

  $\Gamma, n{:}\mathbb{N}, \Delta \vdash C \;\lfloor\textbf{ext }\mathsf{ind}(n; base; m, f.t)\rfloor$
  by `natE` i
  $\quad\Gamma, n{:}\mathbb{N}, \Delta \vdash C[\mathtt{0}/n] \;\lfloor\textbf{ext }base\rfloor$
  $\quad\Gamma, n{:}\mathbb{N}, \Delta, m{:}\mathbb{N}, f{:}C[m/n]$
  $\quad\quad\vdash C[\mathbf{s}(m)/n] \;\lfloor\textbf{ext }t\rfloor$

- Zwei Reduktionsregeln werden benötigt, um die oben angesprochenen Berechnungen zu repräsentieren.

  $\Gamma \vdash \mathsf{ind}(\mathtt{0}; base; m, f.t) = t_2 \in T \;\lfloor\mathsf{Ax}\rfloor$
  by `indRedBase`
  $\quad\Gamma \vdash base = t_2 \in T \;\lfloor\mathsf{Ax}\rfloor$

  $\Gamma \vdash \mathsf{ind}(\mathbf{s}(n); base; m, f.t) = t_2 \in T \;\lfloor\mathsf{Ax}\rfloor$
  by `indRedUp`
  $\quad\Gamma \vdash t[n, \mathsf{ind}(n; base; m, f.t) \,/\, m, f]$
  $\quad\quad = t_2 \in T \;\lfloor\mathsf{Ax}\rfloor$

Soweit die Standardregeln des Datentyps $\mathbb{N}$. Was bisher zu fehlen scheint, ist eine Regel, welche die Beweisführung durch Induktion wiederspiegelt: "*Um einer Eigenschaft P für alle natürlichen Zahlen nachzuweisen, reicht es aus, $P(\mathtt{0})$ zu beweisen und zu zeigen, daß aus $P(n)$ immer auch $P(n{+}1)$ folgt*". Dies wäre also eine Regel der folgenden Art.

$\Gamma, n{:}\mathbb{N}, \Delta \vdash P(n)$
by `induction` n
$\quad\Gamma, n{:}\mathbb{N}, \Delta \vdash P(\mathtt{0})$
$\quad\Gamma, n{:}\mathbb{N}, \Delta, P(n) \vdash P(n{+}1)$

Welche Evidenzen werden nun durch diese Regel konstruiert? Nehmen wir einmal an, wir hätten einen Beweisterm b für $P(\mathtt{0})$ im ersten Teilziel. In den Hypothesen des zweiten Teilziels sei x ein Bezeichner für die Annahme $P(n)$. Dann wird ein Beweisterm t für $P(n{+}1)$ bestenfalls von x und n abhängen können. Für ein beliebiges $m \in \mathbb{N}$ wird also der Beweisterm für $P(m)$ schrittweise aus b und t aufgebaut werden: für m=0 ist er b, für m=1 ist er $t[0, b\,/\,n, x]$, für m=2 ist er $t[1, t[0, b\,/\,n, x]\,/\,n, x]$ usw. Dies aber ist genau der Term, der sich durch $\mathsf{ind}(m; b; n, x.t)$ beschreiben läßt. Damit ist aber die Induktionsregel nichts anderes als ein Spezialfall der Eliminationsregel `natE` für natürliche Zahlen und es ergibt sich folgendes Prinzip:

Entwurfsprinzip 3.4.4 (Induktion)

In der Typentheorie ist induktive Beweisführung dasselbe wie die Konstruktion rekursiver Programme.

Dieses Prinzip vervollständigt das Prinzip "Beweise als Programme" um das Konzept der Rekursion, welches in der Curry-Howard-Isomorphie nicht enthalten war.

Auf der Basis der primitiv rekursiven Arithmetik kann nun im Prinzip die gesamte Arithmetik und Zahlentheorie entwickelt werden. So könnte man zum Beispiel Addition, Vorgängerfunktion ($\mathbf{p}$), Subtraktion, Multiplikation, Quadrat und eine Fallunterscheidung mit Test auf Null durch die üblichen aus der Theorie der primitiv rekursiven Funktionen bekannten Formen definieren.

$n{+}m$	$\equiv$	$\mathsf{ind}(m; n; \mathtt{k}, \mathtt{n}_{+k}.\mathbf{s}(\mathtt{n}_{+k}))$
$\mathbf{p}(n)$	$\equiv$	$\mathsf{ind}(n; \mathtt{0}; \mathtt{k}, \_.\mathtt{k})$
$n{-}m$	$\equiv$	$\mathsf{ind}(m; n; \mathtt{k}, \mathtt{n}_{-k}.\mathbf{p}(\mathtt{n}_{-k}))$
$n{*}m$	$\equiv$	$\mathsf{ind}(m; \mathtt{0}; \mathtt{k}, \mathtt{n}_{*k}.\,\mathtt{n}_{*k}{+}n)$
n^2	$\equiv$	$n{*}n$
$\mathsf{if}\ n{=}0\ \mathsf{then}\ s\ \mathsf{else}\ t$	$\equiv$	$\mathsf{ind}(n; s; \_, \_.t)$

Die definitorischen Erweiterungen der primitiv rekursiven Arithmetik um die wichtigsten arithmetischen Operationen sind also noch relativ einfach durchzuführen. Es ist jedoch verhältnismäßig mühsam, die bekannten Grundeigenschaften dieser Operationen nachzuweisen. Ein Beweis für die Kommutativität der Addition würde zum Beispiel wie folgt beginnen

```
 ⊢ ∀n:ℕ.∀m:ℕ. n+m = m+n ∈ ℕ
 by all_i THEN all_i
  \
  n:ℕ, m:ℕ ⊢ n+m = m+n ∈ ℕ
  by natE 2
  |\
  | n:ℕ, m:ℕ ⊢ n+0 = 0+n ∈ ℕ
  | by indRedBase
  |   \
  |   n:ℕ, m:ℕ ⊢ n = 0+n ∈ ℕ
  |   by natE 1
  |   |\
  |   | n:ℕ, m:ℕ ⊢ 0 = 0+0 ∈ ℕ
  |   | by symmetry THEN indRedBase
  |   |   \
  |   |   n:ℕ, m:ℕ ⊢ 0 = 0 ∈ ℕ
  |   |   by zeroEq
  |    \
  |     n:ℕ, m:ℕ, k:ℕ, f_k:k=0+k ∈ ℕ ⊢ s(k) = 0+s(k) ∈ ℕ
  |     by symmetry THEN indRedUp THEN symmetry
  |       \
  |       n:ℕ, m:ℕ, k:ℕ, f_k:k=0+k ∈ ℕ ⊢ s(k) = s(0+k) ∈ ℕ
  |       by sucEq THEN hypothesis 4
   \
   n:ℕ, m:ℕ, k:ℕ, f_k:n+k=k+n ∈ ℕ ⊢ n+s(k) = s(k)+n ∈ ℕ
   by  ⋮
```

Wie man sieht, ist schon der Beweis einer einfachen Eigenschaft in der primitiv rekursiven Arithmetik so aufwendig, daß diese Theorie für praktische Zwecke einfach unbrauchbar ist. Wir müssen daher die Erweiterung der Typentheorie um das Konzept der Zahlen auf eine tragfähigere Grundlage stellen.

kanonisch (Typen)	kanonisch (Elemente)	nichtkanonisch
int{}()	**natnum**{n:n}() **minus**{}(**natnum**{n:n}())	**ind**{}($\boxed{u}$; x, f_x.s; $base$; y, f_y.t) **minus**{}($\boxed{u}$), **add**{}($\boxed{u}$;$\boxed{v}$), **sub**{}($\boxed{u}$;$\boxed{v}$) **mul**{}($\boxed{u}$;$\boxed{v}$), **div**{}($\boxed{u}$;$\boxed{v}$), **rem**{}($\boxed{u}$;$\boxed{v}$) **int_eq**($\boxed{u}$; $\boxed{v}$; s; t), **less**($\boxed{u}$; $\boxed{v}$; s; t)
$\mathbb{Z}$	n $-n$	ind($\boxed{u}$; x, f_x.s; $base$; y, f_y.t) -$\boxed{u}$, $\boxed{u}$+$\boxed{v}$, $\boxed{u}$-$\boxed{v}$ $\boxed{u}$*$\boxed{v}$, $\boxed{u}$÷$\boxed{v}$, $\boxed{u}$rem$\boxed{v}$ if $\boxed{u}$=$\boxed{v}$ then s else t, if $\boxed{u}$<$\boxed{v}$ then s else t
lt{}(u;v) u<v	**Axiom**{}() Axiom	

Zusätzliche Einträge in die Operatorentabelle

Redex		Kontraktum
ind(0; x, f_x.s; $base$; y, f_y.t)	$\xrightarrow{\beta}$	$base$
ind(n; x, f_x.s; $base$; y, f_y.t)	$\xrightarrow{\beta}$	$t[n$,ind(n-1; x, f_x.s; $base$; y, f_y.t) $/y, f_y]$, $(n>0)$
ind(-n; x, f_x.s; $base$; y, f_y.t)	$\xrightarrow{\beta}$	$s[-n$,ind(-n+1; x, f_x.s; $base$; y, f_y.t) $/x, f_x]$, $(n>0)$
$-i$	$\xrightarrow{\beta}$	*Die Negation von i (als Zahl)*
i+j	$\xrightarrow{\beta}$	*Die Summe von i und j*
i-j	$\xrightarrow{\beta}$	*Die Differenz von i und j*
i*j	$\xrightarrow{\beta}$	*Das Produkt von i und j*
i÷j	$\xrightarrow{\beta}$	0, *falls j=0; ansonsten die Integer-Division von i und j*
i rem j	$\xrightarrow{\beta}$	0, *falls j=0; ansonsten der Rest der Division von i und j*
if i=j then s else t	$\xrightarrow{\beta}$	*s, falls $i = j$; ansonsten t*
if i<j then s else t	$\xrightarrow{\beta}$	*s, falls $i < j$; ansonsten t*

Zusätzliche Einträge in die Redex–Kontrakta Tabelle

Typsemantik		
$\mathbb{Z} = \mathbb{Z}$		
i_1<j_1 = i_2<j_2	falls	$i_1 = i_2 \in \mathbb{Z}$ und $j_1 = j_2 \in \mathbb{Z}$
Elementsemantik		
$i = i \in \mathbb{Z}$		
Axiom = Axiom $\in$ s<t	falls	es ganze Zahlen i und j gibt, wobei i kleiner als j ist, für die gilt $s \xrightarrow{l} i$ und $t \xrightarrow{l} j$
$\mathbb{Z} = \mathbb{Z} \in U_j$		
i_1<j_1 = i_2<j_2 $\in U_j$	falls	$i_1 = i_2 \in \mathbb{Z}$ und $j_1 = j_2 \in \mathbb{Z}$

Zusätzliche Einträge in den Semantiktabellen

Abbildung 3.17: Syntax und Semantik des Typs $\mathbb{Z}$

3.4.2.2 Der NuPRL-Typ der ganzen Zahlen

Beim Entwurf der Typentheorie von NuPRL hat man sich dafür entschieden, anstelle der natürlichen Zahlen die *ganzen* Zahlen ($\underline{\mathbb{Z}}$) als grundlegenden Datentyp zu verwenden. Der Grund hierfür ist, daß das formale Schließen über ganze Zahlen nur unwesentlich komplexer ist als Schließen in $\mathbb{N}$ während umgekehrt eine Simulation von $\mathbb{Z}$ – etwa durch $\mathbb{N}$+$\mathbb{N}$ – das Umgehen mit negativen Zahlen erheblich verkomplizieren würde. Ebenso wurde aus praktischen Gesichtspunkten festgelegt, einen Großteil der elementaren arithmetischen Operationen als vordefinierte Terme in die Theorie mit aufzunehmen.

Die kanonischen Elemente von $\mathbb{Z}$ sind die ganzen Zahlen 0, 1, -1, 2, -2, ... in der üblichen Dezimaldarstellung.[51] Die oben beschriebene Induktion, der zentrale nichtkanonische Term der ganzen Zahlen, wird erweitert auf die negativen Zahlen, was zu dem Term ind(u; x, f_x.s; $base$; y, f_y.t) führt. Standardoperationen wie das einstellige - (Negation) and die binären Operation + (*Addition*), - (*Subtraktion*), * (*Multiplikation*), ÷ (*Integerdivision*, d.h. Division abgerundet auf die nächstkleinere ganze Zahl), rem (*Divisionsrest / remainder*) und das Prädikat < (*"kleiner"-Relation*) können in der vertrauten Infix-Notation benutzt werden.

Zur konkreten Analyse von Zahlen werden zwei arithmetische Fallunterscheidungen if u=v then s else t und if u<v then s else t eingeführt. Man beachte jedoch, daß dies nichtkanonische Terme zum Typ $\mathbb{Z}$ sind, die berechenbare Enscheidungsprozeduren darstellen, während das Gleichheits*prädikat* $u = v \in \mathbb{Z}$ und die Relation u<v *Typen* sind, die elementare Propositionen (mit Axiom als kanonischem Element) repräsentieren.

Bei der präzisen Beschreibung der Semantik wird davon ausgegangen, daß die Bedeutung arithmetischer Grundoperationen auf ganzen Zahlen nicht weiter erklärt werden muß, da jeder Mensch ein intuitives Verständnis davon besitzt. Deshalb stützt sich die Reduktion von Redizes mit kanonischen Elementen von $\mathbb{Z}$ weniger auf Termersetzung als auf eine Auswertung arithmetischer Berechnungen auf den zugrundeliegenden Parametern des Operators **natnum**. So liefert zum Beispiel eine Reduktion des Terms 4+5, dessen Abstraktionsform **add**{}(**natnum**{4:n}(); **natnum**{5:n}()) ist, den Term **natnum**{9:n}(), weil die Zahl *neun* genau die Summe der Zahlen *vier* und *fünf* ist. In der Display Form sieht dies genauso aus, wie man es üblicherweise hinschreiben würde: 4+5 $\xrightarrow{l}$ 9.[52] Die vollständige Syntax und Semantik der formalen Repräsentation ganzer Zahlen und ihrer Operationen innerhalb von NuPRL's Typentheorie ist in Abbildung 3.17 zusammengefaßt.

Die große Anzahl der vordefinierten Operationen hat natürlich auch eine umfangreiche Tabelle von zugehörigen Inferenzregeln zur Folge. Neben erweiterten Versionen der im Abschnitt 3.4.2.1 angesprochenen Regeln – also Typformationsregeln, kanonische Regeln[53] sowie nichtkanonische und Reduktionsregeln für den Induktionsterm – kommen jetzt Einführungsregeln für die strukturelle Analyse der arithmetischen Operationen und Reduktionsregeln für die arithmetischen Fallunterscheidungen hinzu. Zusätzlich müssen Regeln für den Vergleichstyp **lt** eingeführt werden.

Da Reduktion nicht auf Termersetzung sondern auf arithmetischer Berechnung beruht, können die Reduktionsregeln von $\mathbb{Z}$ im Gegensatz zu denen der anderen Typen auch auf nichtkanonische Elemente angewandt

[51] Hier gibt es allerdings eine geringe Inkonsistenz in der einheitlichen Abstraktionsform. Während nichtnegative ganze Zahlen intern durch natnum{n:n}() beschrieben werden, hat die abstrakte Darstellung der negativen Zahlen die Form minus{}(natnum{n:n}()). Dabei wird der Operator minus allerdings als *nichtkanonisch* deklariert, wenn sein Argument *nicht* die Form natnum{n:n}() besitzt. Der Grund für diese Doppelrolle des Operators minus ist, daß einerseits nur Zahlen ohne Vorzeichen als Parametertyp verwendet werden sollten andererseits aber alle ganzen Zahlen als kanonische Elemente von $\mathbb{Z}$ zu gelten haben. Die Einführung eines Zusatzoperators negnum{n:n}() mit Darstellungsform '-n', die aus Gründen einer klareren Trennung vorsuziehen wäre, wurde unterlassen, da der Operator minus ohnehin benötigt wird.

[52] Etwas trickreicher ist der Umgang mit negativen kanonischen Elementen von $\mathbb{Z}$, da ein Term der Gestalt -n intern durch minus{}(natnum{n:n}()) beschrieben wird. Um diese Form aufrecht zu erhalten, muß bei der Auswertung von Operationen, in denen negative Zahlen vorkommen, eine Fallunterscheidung vorgenommen werden. So liefert zum Beispiel die Negation einer positiven ganzen Zahl n intern überhaupt keine Veränderung, da der Term minus{}(natnum{n:n}()) bereits in kanonischer Form ist. Die Negation einer negativen Zahl dagegen muß zwei minus-Operatoren entfernen. Es gilt also

minus{}(natnum{n:n}()) $\xrightarrow{\beta}$ minus{}(natnum{n:n}())   minus{}(minus{}(natnum{n:n}())) $\xrightarrow{\beta}$ natnum{n:n}()

In den anderen Fällen ist Reduktion intern etwas komplizierter zu beschreiben, ist aber ebenfalls durch eine simple Fallunterscheidung zu programmieren. Aus theoretischer Hinsicht ist diese Komplikation allerdings unbedeutend.

[53] Man beachte, daß der Term n in den kanonischen Regeln eine *natürliche* Zahl sein muß, also die interne Form natnum{n:n}() haben muß. Bei negativen kanonischen Termen ist zuvor die Regel minusEq bzw. minusI anzuwenden.

werden. Dabei entstehen dann als Teilziele Behauptungen, die den Voraussetzungen für die Reduktion in der Redex–Kontrakta Tabelle entsprechen. Zur Erhöhung der Lesbarkeit dieser Regeln machen wir Gebrauch von den folgenden einfachen (nichtinduktiven) definitorischen Abkürzungen.

Definition 3.4.5 (Arithmetische Vergleiche)

$$\begin{array}{lll}
i\texttt{=}j & \equiv \textbf{eqi}\{\}(i;j) & \equiv i \texttt{ = } j \in \mathbb{Z} \\
i\neq j & \equiv \textbf{neqi}\{\}(i;j) & \equiv \neg(i\texttt{=}j) \\
i\leq j & \equiv \textbf{le}\{\}(i;j) & \equiv \neg(j\texttt{<}i) \\
i\geq j & \equiv \textbf{ge}\{\}(i;j) & \equiv j\leq i \\
i\texttt{>}j & \equiv \textbf{gt}\{\}(i;j) & \equiv j\texttt{<}i
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash \mathbb{Z} \in \mathsf{U}_j \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{intEq}}
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash n \in \mathbb{Z} \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{natnumEq}}
\end{array}
\qquad
\begin{array}{l}
\Gamma \vdash \mathbb{Z} \;\lfloor\textbf{ext } n\rfloor \\
\quad \textbf{by } \underline{\texttt{natnumI } n}
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash \texttt{-}s_1 \texttt{ = } \texttt{-}s_2 \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{minusEq}} \\
\quad\; \Gamma \vdash s_1 \texttt{ = } s_2 \;\lfloor\text{Ax}\rfloor
\end{array}
\qquad
\begin{array}{l}
\Gamma \vdash \mathbb{Z} \;\lfloor\textbf{ext } \texttt{-}s\rfloor \\
\quad \textbf{by } \underline{\texttt{minusI}} \\
\quad\; \Gamma \vdash \mathbb{Z} \;\lfloor\textbf{ext } s\rfloor
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash s_1\texttt{+}t_1 \texttt{ = } s_2\texttt{+}t_2 \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{addEq}} \\
\quad\; \Gamma \vdash s_1 \texttt{ = } s_2 \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash t_1 \texttt{ = } t_2 \;\lfloor\text{Ax}\rfloor
\end{array}
\qquad
\begin{array}{l}
\Gamma \vdash \mathbb{Z} \;\lfloor\textbf{ext } s\texttt{+}t\rfloor \\
\quad \textbf{by } \underline{\texttt{addI}} \\
\quad\; \Gamma \vdash \mathbb{Z} \;\lfloor\textbf{ext } s\rfloor \\
\quad\; \Gamma \vdash \mathbb{Z} \;\lfloor\textbf{ext } t\rfloor
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash s_1\texttt{-}t_1 \texttt{ = } s_2\texttt{-}t_2 \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{subEq}} \\
\quad\; \Gamma \vdash s_1 \texttt{ = } s_2 \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash t_1 \texttt{ = } t_2 \;\lfloor\text{Ax}\rfloor
\end{array}
\qquad
\begin{array}{l}
\Gamma \vdash \mathbb{Z} \;\lfloor\textbf{ext } s\texttt{-}t\rfloor \\
\quad \textbf{by } \underline{\texttt{subI}} \\
\quad\; \Gamma \vdash \mathbb{Z} \;\lfloor\textbf{ext } s\rfloor \\
\quad\; \Gamma \vdash \mathbb{Z} \;\lfloor\textbf{ext } t\rfloor
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash s_1\texttt{*}t_1 \texttt{ = } s_2\texttt{*}t_2 \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{mulEq}} \\
\quad\; \Gamma \vdash s_1 \texttt{ = } s_2 \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash t_1 \texttt{ = } t_2 \;\lfloor\text{Ax}\rfloor
\end{array}
\qquad
\begin{array}{l}
\Gamma \vdash \mathbb{Z} \;\lfloor\textbf{ext } s\texttt{*}t\rfloor \\
\quad \textbf{by } \underline{\texttt{mulI}} \\
\quad\; \Gamma \vdash \mathbb{Z} \;\lfloor\textbf{ext } s\rfloor \\
\quad\; \Gamma \vdash \mathbb{Z} \;\lfloor\textbf{ext } t\rfloor
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash s_1\div t_1 \texttt{ = } s_2\div t_2 \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{divEq}} \\
\quad\; \Gamma \vdash s_1 \texttt{ = } s_2 \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash t_1 \texttt{ = } t_2 \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash t_1\neq\texttt{0} \;\lfloor\text{Ax}\rfloor
\end{array}
\qquad
\begin{array}{l}
\Gamma \vdash \mathbb{Z} \;\lfloor\textbf{ext } s\div t\rfloor \\
\quad \textbf{by } \underline{\texttt{divI}} \\
\quad\; \Gamma \vdash \mathbb{Z} \;\lfloor\textbf{ext } s\rfloor \\
\quad\; \Gamma \vdash \mathbb{Z} \;\lfloor\textbf{ext } t\rfloor
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash s_1 \;\mathsf{rem}\; t_1 \texttt{ = } s_2 \;\mathsf{rem}\; t_2 \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{remEq}} \\
\quad\; \Gamma \vdash s_1 \texttt{ = } s_2 \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash t_1 \texttt{ = } t_2 \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash t_1\neq\texttt{0} \;\lfloor\text{Ax}\rfloor
\end{array}
\qquad
\begin{array}{l}
\Gamma \vdash \mathbb{Z} \;\lfloor\textbf{ext } s \;\mathsf{rem}\; t\rfloor \\
\quad \textbf{by } \underline{\texttt{remI}} \\
\quad\; \Gamma \vdash \mathbb{Z} \;\lfloor\textbf{ext } s\rfloor \\
\quad\; \Gamma \vdash \mathbb{Z} \;\lfloor\textbf{ext } t\rfloor
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash \mathsf{if}\; u_1\texttt{=}v_1 \;\mathsf{then}\; s_1 \;\mathsf{else}\; t_1 \\
\quad \texttt{=}\; \mathsf{if}\; u_2\texttt{=}v_2 \;\mathsf{then}\; s_2 \;\mathsf{else}\; t_2 \in T \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{int_eqEq}} \\
\quad\; \Gamma \vdash u_1\texttt{=}u_2 \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash v_1\texttt{=}v_2 \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma,\; v\!:u_1\texttt{=}v_1 \vdash s_1 \texttt{ = } s_2 \in T \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma,\; v\!:u_1\neq v_1 \vdash t_1 \texttt{ = } t_2 \in T \;\lfloor\text{Ax}\rfloor
\end{array}
\qquad
\begin{array}{l}
\Gamma \vdash \mathsf{if}\; u_1\texttt{<}v_1 \;\mathsf{then}\; s \;\mathsf{else}\; t \\
\quad \texttt{=}\; \mathsf{if}\; u_2\texttt{<}v_2 \;\mathsf{then}\; s_2 \;\mathsf{else}\; t_2 \in T \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{lessEq}} \\
\quad\; \Gamma \vdash u_1\texttt{=}u_2 \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash v_1\texttt{=}v_2 \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma,\; v\!:u_1\texttt{<}v_1 \vdash s_1 \texttt{ = } s_2 \in T \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma,\; v\!:u_1\geq v_1 \vdash t_1 \texttt{ = } t_2 \in T \;\lfloor\text{Ax}\rfloor
\end{array}$$

Abbildung 3.18: Inferenzregeln des Typs $\mathbb{Z}$ (1)

$$\begin{array}{l}
\Gamma \vdash \mathsf{ind}(u_1;\, x_1,f_{x1}.s_1;\, base_1;\, y_1,f_{y_1}.t_1) \\
\quad \texttt{=}\; \mathsf{ind}(u_2;\, x_2,f_{x2}.s_2;\, base_2;\, y_2,f_{y_2}.t_2) \\
\quad \in T[u_1/z] \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{indEq } z\; T} \\
\quad\; \Gamma \vdash u_1\texttt{=}u_2 \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma,\; x\!:\!\mathbb{Z},\; v\!:x\texttt{<0},\; f_x\!:\!T[(x\texttt{+1})/z] \\
\quad\quad \vdash s_1[x,f_x\,/\,x_1,f_{x1}] \texttt{ = } s_2[x,f_x\,/\,x_2,f_{x2}] \\
\quad\quad\quad \in T[x/z] \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash base_1 \texttt{ = } base_2 \in T[\texttt{0}/z] \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma,\; x\!:\!\mathbb{Z},\; v\!:\texttt{0<}x,\; f_x\!:\!T[(x\texttt{-1})/z] \\
\quad\quad \vdash t_1[x,f_x\,/\,y_1,f_{y_1}] \texttt{ = } t_2[x,f_x\,/\,y_2,f_{y_2}] \\
\quad\quad\quad \in T[x/z] \;\lfloor\text{Ax}\rfloor
\end{array}
\qquad
\begin{array}{l}
\Gamma,\; z\!:\!\mathbb{Z},\; \Delta \vdash C \\
\lfloor\textbf{ext } \mathsf{ind}(z;\, x,f_x.s[\mathsf{Axiom}/v];\, base;\, x,f_x.t[\mathsf{Axiom}/v])\rfloor \\
\quad \textbf{by } \underline{\texttt{intE } i} \\
\quad\; \Gamma,\; z\!:\!\mathbb{Z},\; \Delta,\; x\!:\!\mathbb{Z},\; v\!:x\texttt{<0},\; f_x\!:\!C[(x\texttt{+1})/z] \\
\quad\quad \vdash C[x/z] \;\lfloor\textbf{ext } s\rfloor \\
\quad\; \Gamma,\; z\!:\!\mathbb{Z},\; \Delta \vdash C[\texttt{0}/z] \;\lfloor\textbf{ext } base\rfloor \\
\quad\; \Gamma,\; z\!:\!\mathbb{Z},\; \Delta,\; x\!:\!\mathbb{Z},\; v\!:\texttt{0<}x,\; f_x\!:\!C[(x\texttt{-1})/z] \\
\quad\quad \vdash C[x/z] \;\lfloor\textbf{ext } t\rfloor
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash \mathsf{ind}(i;\, x,f_x.s;\, base;\, y,f_y.t) \texttt{ = } t_2 \in T \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{indRedDown}} \\
\quad\; \Gamma \vdash t[i,\mathsf{ind}(i\texttt{+1};\, x,f_x.s;\, base;\, y,f_y.t)\;/x,f_x] \\
\quad\quad \texttt{=}\; t_2 \in T \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash i\texttt{<0} \;\lfloor\text{Ax}\rfloor
\end{array}
\qquad
\begin{array}{l}
\Gamma \vdash \mathsf{ind}(i;\, x,f_x.s;\, base;\, y,f_y.t) \texttt{ = } t_2 \in T \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{indRedUp}} \\
\quad\; \Gamma \vdash t[i,\mathsf{ind}(i\texttt{-1};\, x,f_x.s;\, base;\, y,f_y.t)\;/y,f_y] \\
\quad\quad \texttt{=}\; t_2 \in T \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash \texttt{0<}i \;\lfloor\text{Ax}\rfloor
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash \mathsf{ind}(i;\, x,f_x.s;\, base;\, y,f_y.t) \texttt{ = } t_2 \in T \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{indRedBase}} \\
\quad\; \Gamma \vdash base \texttt{ = } t_2 \in T \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash i\texttt{=0} \;\lfloor\text{Ax}\rfloor
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash \mathsf{if}\; u\texttt{=}v \;\mathsf{then}\; s \;\mathsf{else}\; t \texttt{ = } t_2 \in T \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{int_eqRedT}} \\
\quad\; \Gamma \vdash s \texttt{ = } t_2 \in T \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash u\texttt{=}v \;\lfloor\text{Ax}\rfloor
\end{array}
\qquad
\begin{array}{l}
\Gamma \vdash \mathsf{if}\; u\texttt{=}v \;\mathsf{then}\; s \;\mathsf{else}\; t \texttt{ = } t_2 \in T \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{int_eqRedF}} \\
\quad\; \Gamma \vdash t \texttt{ = } t_2 \in T \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash u\neq v \;\lfloor\text{Ax}\rfloor
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash \mathsf{if}\; u\texttt{<}v \;\mathsf{then}\; s \;\mathsf{else}\; t \texttt{ = } t_2 \in T \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{lessRedT}} \\
\quad\; \Gamma \vdash s \texttt{ = } t_2 \in T \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash u\texttt{<}v \;\lfloor\text{Ax}\rfloor
\end{array}
\qquad
\begin{array}{l}
\Gamma \vdash \mathsf{if}\; u\texttt{<}v \;\mathsf{then}\; s \;\mathsf{else}\; t \texttt{ = } t_2 \in T \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{lessRedF}} \\
\quad\; \Gamma \vdash t \texttt{ = } t_2 \in T \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash u\geq v \;\lfloor\text{Ax}\rfloor
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash \texttt{0} \leq s \;\mathsf{rem}\; t \;\wedge\; s \;\mathsf{rem}\; t < t \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{remBounds1}} \\
\quad\; \Gamma \vdash \texttt{0} \leq s \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash \texttt{0} < t \;\lfloor\text{Ax}\rfloor
\end{array}
\qquad
\begin{array}{l}
\Gamma \vdash \texttt{0} \leq s \;\mathsf{rem}\; t \;\wedge\; s \;\mathsf{rem}\; t < \texttt{-}t \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{remBounds2}} \\
\quad\; \Gamma \vdash \texttt{0} \leq s \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash t < \texttt{0} \;\lfloor\text{Ax}\rfloor
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash s \;\mathsf{rem}\; t \leq \texttt{0} \;\wedge\; s \;\mathsf{rem}\; t > \texttt{-}t \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{remBounds3}} \\
\quad\; \Gamma \vdash s \leq \texttt{0} \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash \texttt{0} < t \;\lfloor\text{Ax}\rfloor
\end{array}
\qquad
\begin{array}{l}
\Gamma \vdash s \;\mathsf{rem}\; t \leq \texttt{0} \;\wedge\; s \;\mathsf{rem}\; t > t \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{remBounds4}} \\
\quad\; \Gamma \vdash s \leq \texttt{0} \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash t < \texttt{0} \;\lfloor\text{Ax}\rfloor
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash s \texttt{ = } (s\div t)\texttt{*}t + (s \;\mathsf{rem}\; t) \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{divremSum}} \\
\quad\; \Gamma \vdash s\in\mathbb{Z} \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash t\neq\texttt{0} \;\lfloor\text{Ax}\rfloor
\end{array}$$

$$\begin{array}{l}
\Gamma \vdash s_1\texttt{<}t_1 \texttt{ = } s_2\texttt{<}t_2 \in \mathsf{U}_j \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{ltEq}} \\
\quad\; \Gamma \vdash s_1 \texttt{ = } s_2 \;\lfloor\text{Ax}\rfloor \\
\quad\; \Gamma \vdash t_1 \texttt{ = } t_2 \;\lfloor\text{Ax}\rfloor
\end{array}
\qquad
\begin{array}{l}
\Gamma \vdash \mathsf{Axiom} \in s\texttt{<}t \;\lfloor\text{Ax}\rfloor \\
\quad \textbf{by } \underline{\texttt{axiomEq_lt}} \\
\quad\; \Gamma \vdash s\texttt{<}t \;\lfloor\text{Ax}\rfloor
\end{array}$$

Abbildung 3.19: Inferenzregeln des Typs $\mathbb{Z}$ (2)

Neben den in den Abbildungen 3.18 und 3.19 zusammengestellten Regeln für den Typ der ganzen Zahlen gibt es eine weitere Regel, die in praktischer Hinsicht extrem wichtig ist, aber nur algorithmisch erklärt werden kann. Die Regel `arith` j ist eine arithmetische *Entscheidungsprozedur*, die es erlaubt, in einem einzigen Schritt die Gültigkeit eines arithmetischen Vergleichs zu beweisen, der aus den Hypothesen durch eine eingeschränkte Form arithmetischen Schließens folgt. Die Konklusion muß dabei entweder die Form 'void' haben (falls die Hypothesen arithmetische Widersprüche enthalten) oder disjunktiv aus $=,\neq,<,>,\leq,\geq$ und jeweils zwei Termen des Typs $\mathbb{Z}$ zusammengesetzt sein. Den Algorithmus, der hinter dieser Regel steht, seinen Anwendungsbereich und seine Rechtfertigung als Bestandteil eines formalen Schlußfolgerungssystems werden wir im Kapitel 4.3 besprechen.

3.4.3 Listen

So wie die natürlichen Zahlen gehören Listen zu den grundlegensten Konstrukten der 'realen' Programmierwelt. Sie werden benötigt, wenn endliche, aber beliebig große Strukturen von Datensätzen – wie lineare Listen, Felder, Bäume etc. – innerhalb von Programmen aufgebaut werden müssen. Rein hypothetisch wäre es möglich, den Datentyp der Listen mit Elementen eines Datentyps T in der bisherigen Typentheorie zu simulieren.[54] Eine solche Simulation wäre allerdings verhältnismäßig aufwendig und würde die wesentlichen Eigenschaften von Listen nicht zur Geltung kommen lassen. Aus diesem Grunde wurde der Datentyp T list der Listen (Folgen) über einem beliebigen Datentyp T und seine Operatoren explizit zur Typentheorie hinzugenommen und direkt auf einer Formalisierung der gängigen Semantik von Listen abgestützt.

Wie in Abschnitt 2.3.3 auf Seite 56 bereits angedeutet wurde, können Listen als eine Art Erweiterung des Konzepts der natürlichen Zahlen – aufgebaut durch Null und Nachfolgerfunktion – angesehen werden. Die kanonischen Elemente werden gebildet durch die leere Liste [] und durch Anwendung der Operation **cons**$\{\}(t;l)$, welche ein Element $t \in T$ vor eine bestehende Liste l anhängt. Der nichtkanonische Term ist ein Gegenstück zum Induktionsterm ind: list_ind$(L;\, base;\, x,l,f_l\,.t)$ beschreibt die Werte einer Funktion f, die ihr Ergebnis durch eine Analyse des induktiven Aufbaus der Liste L bestimmt. Ist L die leere Liste, so lautet das Ergebnis *base*. Hat L dagegen die Gestalt **cons**$\{\}(x;l)$, so ist $t[x,l,f_l]$ das Ergebnis, wobei f_l ein Platzhalter für $f(l)$ ist. Listeninduktion ist also eine Erweiterung der primitiven Rekursion.

Im Gegensatz zu den ganzen Zahlen beschränkt man sich auf diese vier Grundoperationen und formuliert alle anderen Listenoperationen als definitorische Erweiterungen. Die Semantik und die Inferenzregeln von Listen sind leicht zu formalisieren, wenn man die Formalisierung von natürlichen Zahlen aus Abschnitt 3.4.2.1 als Ausgangspunkt nimmt. In Abbildung 3.20 ist die komplette Erweiterung des Typsystems zusammengefaßt.

Wir wollen nun anhand eines größeren Beispiels zeigen, daß das Prinzip, Programme durch Beweisführung zu entwickeln (Entwurfsprinzip 3.4.3), in der Tat ein sinnvolles Konzept ist. Die Extraktion von Programmen aus dem Beweis eines Spezifikationstheorems garantiert nicht nur die Korrektheit des erstellten Programms sondern führt in den meisten Fällen auch zu wesentlich besser durchdachten und effizienteren Algorithmen. Wir hatten bereits im Einführungskapitel im Abschnitt 1.1 mit dem Beispiel der maximalen Segmentsumme illustriert, daß mathematisches Problemlösen und Programmierung sehr ähnliche Aufgaben sind. Wir wollen nun zeigen, wie dieses Beispiel innerhalb eines formalen Schlußfolgerungssystems behandelt werden kann.

Beispiel 3.4.6 (Maximale Segmentsumme)

In Beispiel 1.1.1 auf Seite 2 hatten wir das Problem der Berechnung der maximalen Summe von Teilsegmenten einer gegebenen Folge ganzer Zahlen aufgestellt.

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

[54] Man kann eine Liste mit Elementen aus T als eine endliche Funktion $l \in \{1..n\} \to T$ interpretieren. Wenn wir die endliche Menge $\{1..n\}$ wiederum durch $x{:}\mathbb{Z} \times (1 \leq x \wedge x \leq n)$ repräsentieren, so können wir den Datentyp T list wie folgt simulieren
$T \text{ list} \equiv n{:}\mathbb{N} \times (x{:}\mathbb{Z} \times (1 \leq x \wedge x \leq n)) \to T$
Eine andere Möglichkeit ergibt sich aus der Darstellung von Listen im λ-Kalkül wie in Abschnitt 2.3.3.

kanonisch (Typen)	kanonisch (Elemente)	nichtkanonisch
list$\{\}(T)$	**nil**$\{\}()$, **cons**$\{\}(t;l)$	**list_ind**$\{\}(\boxed{s};\, base;\, x,l,f_{xl}\,.t)$
T list	[], $t\,.l$	list_ind$(\boxed{s};\, base;\, x,l,f_{xl}\,.t)$

Zusätzliche Einträge in die Operatorentabelle

Redex		Kontraktum
list_ind$([];\, base;\, x,l,f_{xl}\,.t)$	$\xrightarrow{\beta}$	$base$
list_ind$(s\,.u;\, base;\, x,l,f_{xl}\,.t)$	$\xrightarrow{\beta}$	$t[s,u,\text{list\_ind}(u;\, base;\, x,l,f_{xl}\,.t)\ /x,l,f_{xl}]$

Zusätzliche Einträge in die Redex–Kontrakta Tabelle

Typsemantik	
T_1 list $= T_2$ list	falls $T_1 = T_2$
Elementsemantik	
[] $=$ [] $\in T$ list	falls T Typ
$t_1\,.l_1 = t_2\,.l_2 \in T$ list	falls T Typ und $t_1 = t_2 \in T$ und $l_1 = l_2 \in T$ list
T_1 list $= T_2$ list $\in \mathsf{U}_j$	falls $T_1 = T_2 \in \mathsf{U}_j$

Zusätzliche Einträge in den Semantiktabellen

$\Gamma \vdash T_1 \text{ list} = T_2 \text{ list} \in \mathsf{U}_j$ $\lfloor\mathsf{Ax}\rfloor$
by `listEq`
$\Gamma \vdash T_1 = T_2 \in \mathsf{U}_j$ $\lfloor\mathsf{Ax}\rfloor$

$\Gamma \vdash [] \in T \text{ list}$ $\lfloor\mathsf{Ax}\rfloor$
by `nilEq` j
$\Gamma \vdash T \in \mathsf{U}_j$ $\lfloor\mathsf{Ax}\rfloor$

$\Gamma \vdash T \text{ list}$ $\lfloor$**ext** []$\rfloor$
by `nilI` j
$\Gamma \vdash T \in \mathsf{U}_j$ $\lfloor\mathsf{Ax}\rfloor$

$\Gamma \vdash t_1\,.l_1 = t_2\,.l_2 \in T \text{ list}$ $\lfloor\mathsf{Ax}\rfloor$
by `consEq`
$\Gamma \vdash t_1 = t_2 \in T$ $\lfloor\mathsf{Ax}\rfloor$
$\Gamma \vdash l_1 = l_2 \in T \text{ list}$ $\lfloor\mathsf{Ax}\rfloor$

$\Gamma \vdash T \text{ list}$ $\lfloor$**ext** $t\,.l\rfloor$
by `consI`
$\Gamma \vdash T$ $\lfloor$**ext** $t\rfloor$
$\Gamma \vdash T \text{ list}$ $\lfloor$**ext** $l\rfloor$

$\Gamma \vdash \text{list\_ind}(s_1;\, base_1;\, x_1,l_1,f_{xl1}\,.t_1)$
$= \text{list\_ind}(s_2;\, base_2;\, x_2,l_2,f_{xl2}\,.t_2)$
$\in T[s_1/z]$ $\lfloor\mathsf{Ax}\rfloor$
by `list_indEq` z T S list
$\Gamma \vdash s_1 = s_2 \in S \text{ list}$ $\lfloor\mathsf{Ax}\rfloor$
$\Gamma \vdash base_1 = base_2 \in T[[]/z]$ $\lfloor\mathsf{Ax}\rfloor$
$\Gamma,\ x{:}S,\ l{:}S \text{ list},\ f_{xl}{:}T[l/z] \vdash$
$t_1[x,l,f_{xl}\,/\,x_1,l_1,f_{xl1}] =$
$t_1[x,l,f_{xl}\,/\,x_2,l_2,f_{xl2}]$
$\in T[x\,.l/z]$ $\lfloor\mathsf{Ax}\rfloor$

$\Gamma,\ z{:}T \text{ list},\ \Delta \vdash C$
$\lfloor$**ext** $\text{list\_ind}(z;\, base;\, x,l,f_{xl}\,.t)\rfloor$
by `listE` i
$\Gamma,\ z{:}T \text{ list},\ \Delta \vdash C[[]/z]$ $\lfloor$**ext** $base\rfloor$
$\Gamma,\ z{:}T \text{ list},\ \Delta,\ x{:}T,\ l{:}T \text{ list},\ f_{xl}{:}C[l/z]$
$\vdash C[x\,.l/z]$ $\lfloor$**ext** $t\rfloor$

$\Gamma \vdash \text{list\_ind}([];\, base;\, x,l,f_{xl}\,.t) = t_2 \in T$ $\lfloor\mathsf{Ax}\rfloor$
by `list_indRedBase`
$\Gamma \vdash base = t_2 \in T$ $\lfloor\mathsf{Ax}\rfloor$

$\Gamma \vdash \text{list\_ind}(s\,.u;\, base;\, x,l,f_{xl}\,.t) = t_2 \in T$ $\lfloor\mathsf{Ax}\rfloor$
by `list_indRedUp`
$\Gamma \vdash t[s,u,\text{list\_ind}(u;\, base;\, x,l,f_{xl}\,.t)\ /x,l,f_{xl}]$
$= t_2 \in T$ $\lfloor\mathsf{Ax}\rfloor$

Inferenzregeln

Abbildung 3.20: Syntax, Semantik und Inferenzregeln des Listentyps

```
⊢ ∀a:ℤ list .∀a1:ℤ.∃L:ℤ.∃M:ℤ. L=maxbeg(a1.a) ∧ M=maxseq(a1.a)
by all_i THEN listE 1 THEN all_i                                   (Induktion auf  a)
|\
| a:ℤ list , a1:ℤ ⊢ ∃L:ℤ.∃M:ℤ. L=maxbeg(a1.[]) ∧ M=maxseq(a1.[])
| by ex_i a1 THEN ex_i a1 THEN and_i
| |\
| | a:ℤ list , a1:ℤ ⊢ a1=maxbeg(a1.[])
| | by ...  Anwendung arithmetischer Lemmata ...
| \
|   a:ℤ list , a1:ℤ ⊢ a1=maxseq(a1.[])
|   by ...  Anwendung arithmetischer Lemmata ...
 \
  a:ℤ list , x:ℤ, l:ℤ list , v:∀a1:ℤ.∃L:ℤ.∃M:ℤ. L=maxbeg(a1.l) ∧ M=maxseq(a1.l), a1:ℤ
  ⊢ ∃L:ℤ.∃M:ℤ. L=maxbeg(a1.(x.l)) ∧ M=maxseq(a1.(x.l))
  by all_e 4 x THEN thin 4
   \
    a:ℤ list , x:ℤ, l:ℤ list , a1:ℤ, v1:∃L:ℤ.∃M:ℤ. L=maxbeg(x.l) ∧ M=maxseq(x.l)
    ⊢ ∃L:ℤ.∃M:ℤ. L=maxbeg(a1.(x.l)) ∧ M=maxseq(a1.(x.l))
    by ex_e 5 THEN ex_e 6 THEN and_e 7
     \
      a:ℤ list , x:ℤ, l:ℤ list , a1:ℤ, L:ℤ, M:ℤ, v2:L=maxbeg(x.l), v3:M=maxseq(x.l)
      ⊢ ∃L:ℤ.∃M:ℤ. L=maxbeg(a1.(x.l)) ∧ M=maxseq(a1.(x.l))
      by ex_i  max(L+a1,a1)  THEN  ex_i  max(M, max(L+a1,a1))  THEN and_i
      |\
      | a:ℤ list , x:ℤ, l:ℤ list , a1:ℤ, L:ℤ, M:ℤ, v2:L=maxbeg(x.l), v3:M=maxseq(x.l)
      | ⊢  max(L+a1,a1)=maxbeg(a1.(x.l))
      | by ...  Anwendung arithmetischer Lemmata ...
       \
        a:ℤ list , x:ℤ, l:ℤ list , a1:ℤ, L:ℤ, M:ℤ, v2:L=maxbeg(x.l), v3:M=maxseq(x.l)
        ⊢   max(M, max(L+a1,a1))=maxseq(a1.(x.l))
        by ...  Anwendung arithmetischer Lemmata ...
```

Abbildung 3.21: Formaler Induktionsbeweis für die Existenz der maximalen Segmentsumme

Unsere Analyse ergab, daß die naheliegende iterative Lösung (kubische Laufzeit!) durch mathematisch-induktive Betrachtungen zu einem linearen Algorithmus verbessert werden kann. Bestimmt man nämlich parallel zu der maximalen Segmentsumme M_k einer Teilliste von k Elementen auch noch die maximalen Summe L_k von Segmenten, die das letzte Element enthalten, dann ist $M_1{=}L_1{=}a_1$. L_{k+1} ist das Maximum von a_{k+1} und $a_{k+1}{+}L_k$. M_{k+1} ist schließlich das Maximum von L_{k+1} und M_k.

Da ganze Zahlen und Listen vordefiniert sind, ist es nicht schwierig, die obige Problemstellung innerhalb der Typentheorie formal zu beschreiben. Allerdings ist zu berücksichtigen, daß Listen schrittweise nach *vorne* erweitert werden, während in unserer Argumentation die Erweiterung zum rechten Ende hin betrachtet wurde. Es empfiehlt sich daher, zugunsten einer einfacheren Beweisführung das Argument umzudrehen und von der eingebauten Listeninduktion Gebrauch zu machen.

Zudem geht das Argument davon aus, daß die betrachteten Listen immer mindestens ein Element enthalten. Wir müssen unsere Induktion also bei der Länge 1 verankern, denn ansonsten müßten wir $L_0{=}0$ und $M_0{=}{-}\infty$ wählen. In Abbildung 3.21 haben wir eine Formalisierung des zentralen Arguments innerhalb eines NuPRL-Beweises skizziert.[55] Wir verwenden dabei die folgenden definitorischen Abkürzungen.

[55] Ohne eine Unterstützung durch Lemmata und Beweistaktiken, welche die Anwendung von Regeln zum Teil automatisieren (siehe Abschnitt 4.2) kann die arithmetische *Rechtfertigung* der in Beispiel 1.1.1 gegebenen Argumentation nicht praktisch formalisiert werden. Der vollständige Beweis macht daher ausgiebig Gebrauch von beidem und soll hier nicht weiter betrachtet werden. Die mathematischen Gleichungen (Lemmata), welche zur Unterstützung verwandt werden, sind die folgenden.

$$\begin{aligned}
M_1 &= \mathtt{max}(\{\textstyle\sum_{i=p}^{q} a_i \mid 1{\le}p{\le}q{\le}1\}) = \sum_{i=1}^{1} a_i = a_1\\
M_{n+1} &= \mathtt{max}(\{\textstyle\sum_{i=p}^{q} a_i \mid 1{\le}p{\le}q{\le}n{+}1\}) = \mathtt{max}(\{\sum_{i=p}^{q} a_i \mid 1{\le}p{\le}q{\le}n\} \cup \{\sum_{i=p}^{q} a_i \mid 1{\le}p{\le}q{=}n\})\\
&= \mathtt{max}(\mathtt{max}(\{\textstyle\sum_{i=p}^{q} a_i \mid 1{\le}p{\le}q{\le}n\}),\ \mathtt{max}(\{\sum_{i=p}^{n} a_i \mid 1{\le}p{\le}n\})) = \mathtt{max}(M_n, L_{n+1})\\
L_1 &= \mathtt{max}(\{\textstyle\sum_{i=p}^{1} a_i \mid 1{\le}p{\le}1\}) = \sum_{i=1}^{1} a_i = a_1\\
L_{n+1} &= \mathtt{max}(\{\textstyle\sum_{i=p}^{n+1} a_i \mid 1{\le}p{\le}n{+}1\}) = \mathtt{max}(\{\sum_{i=p}^{n+1} a_i \mid 1{\le}p{\le}n\} \cup \{\sum_{i=p}^{n+1} a_i \mid 1{\le}p{=}n{+}1\})\\
&= \mathtt{max}(\{\textstyle\sum_{i=p}^{n} a_i{+}a_{n+1} \mid 1{\le}p{\le}n\} \cup \{\sum_{i=n+1}^{n+1} a_i\}) = \mathtt{max}(\mathtt{max}(\{\sum_{i=p}^{n} a_i{+}a_{n+1} \mid 1{\le}p{\le}n\}),\ \mathtt{max}(\{a_{n+1}\}))\\
&= \mathtt{max}(\mathtt{max}(\{\textstyle\sum_{i=p}^{n} a_i \mid 1{\le}p{\le}n\}){+}a_{n+1},\ a_{n+1}) = \mathtt{max}(L_n{+}a_{n+1}, a_{n+1})
\end{aligned}$$

$$\begin{array}{lll}
\mathsf{max}(i,j) & \equiv \ \mathbf{max}\{\}(i;j) & \equiv \ \mathsf{if}\ i{<}j\ \mathsf{then}\ \mathtt{j}\ \mathsf{else}\ i\\
|a| & \equiv \ \mathbf{length}\{\}(a) & \equiv \ \mathsf{list_ind}(a;\, 0;\, \mathtt{x,a',lg}_{a'}.\mathtt{lg}_{a'}{+}1)\\
\mathsf{hd}(a) & \equiv \ \mathbf{head}\{\}(a) & \equiv \ \mathsf{list_ind}(a;\, 0;\, \mathtt{x,a',hd}_{a'}.\mathtt{x})\\
\mathsf{tl}(a) & \equiv \ \mathbf{tail}\{\}(a) & \equiv \ \mathsf{list_ind}(a;\, [\,];\, \mathtt{x,a',tl}_{a'}.\mathtt{a'})\\
a_i & \equiv \ \mathbf{select}\{\}(a;i) & \equiv \ \mathsf{hd}(\mathsf{ind}(i;\, _,_.a;\, a;\, \mathtt{x,a'}.\mathsf{tl}(\mathtt{a'})\mathtt{tl(s)}))\\
\sum_{i=p}^{q} a_i & \equiv \ \mathbf{sum}\{\}(a;p;q) & \equiv \ \mathsf{ind}(q{-}p;\, _,_,0;\ a_p;\ \mathtt{i,sum.\,sum}{+}a_{p+i+1})\\
M = \mathtt{maxseq}(a) & \equiv \ \mathbf{maxseq}\{\}(a;M) & \equiv \ \exists \mathtt{k,j}{:}\mathbb{Z}.\ (1{\le}\mathtt{k} \wedge \mathtt{k}{\le}\mathtt{j} \wedge \mathtt{j}{\le}|a|)\ \wedge\ M{=}\sum_{i=k}^{j} a_i\\
 & & \quad \wedge\ \forall \mathtt{p,q}{:}\mathbb{Z}.\ (1{\le}\mathtt{p} \wedge \mathtt{p}{\le}\mathtt{q} \wedge \mathtt{q}{\le}|a|)\ \Rightarrow\ M{\ge}\sum_{i=p}^{q} a_i\\
L = \mathtt{maxbeg}(a) & \equiv \ \mathbf{maxbeg}\{\}(a;L) & \equiv \ \exists \mathtt{j}{:}\mathbb{Z}.\ (1{\le}\mathtt{j} \wedge \mathtt{j}{\le}|a|)\ \wedge\ L{=}\sum_{i=1}^{j} a_i\\
 & & \quad \wedge\ \forall \mathtt{q}{:}\mathbb{Z}.\ (1{\le}\mathtt{q} \wedge \mathtt{q}{\le}|a|)\ \Rightarrow\ L{\ge}\sum_{i=1}^{q} a_i
\end{array}$$

Der Algorithmus, der in diesem Beweis enthalten ist, hat die Form

$$\begin{aligned}
&\lambda \mathsf{a}.\lambda \mathsf{a}_1.\ \ \mathsf{list_ind}(\mathsf{a};\ \langle \mathsf{a}_1, \mathsf{a}_1, pf_{base}\rangle;\\
&\qquad\qquad \mathtt{x,l,v}.\ \ \lambda \mathsf{a}_1.\mathsf{let}\ \langle \mathtt{L}, \langle \mathtt{M}, \mathtt{v}_2, \mathtt{v}_3\rangle\rangle = \mathtt{v\ x}\ \mathtt{in}\ \langle \mathtt{max(L{+}a_1,a_1)}, \mathtt{max(M,\ max(L{+}a_1,a_1))}, pf_{ind}\rangle)
\end{aligned}$$

wobei pf_{base} und pf_{ind} Terme sind, welche die Korrektheit der vorgegebenen Lösungen nachweisen. Durch den formalen Beweis ist der Algorithmus, der nur eine einzige Induktionsschleife beinhaltet, als korrekt nachgewiesen und somit erheblich effizienter als eine ad hoc Lösung des Problems.

3.4.4 Teilmengen

Das Beispiel der formalen Herleitung eines Algorithmus zur Bestimmung der maximalen Segmentsumme in einer Liste zeigt nicht nur die Vorteile einer mathematischen Vorgehensweise sondern auch einige Schwächen in der praktischen Ausdruckskraft der bisherigen Theorie.

- Zum einen enthält ein Programm, welches aus dem Beweis eines Spezifikationstheorems extrahiert wird, Teilterme, die nur zum Nachweis der Korrektheit der Resultate, nicht aber zu ihrer Berechnung benötigt werden. Der Grund hierfür liegt in der Tatsache, daß die die Standard-Einbettung der konstruktiven Logik den Existenzquantor $\exists x{:}T.A[x]$ mit dem Produktraum $x{:}T{\times}A[x]$ identifiziert und somit ein Beweisterm nicht nur das Element x selbst sondern auch einen Nachweis für $A[x]$ konstruiert. Während aus beweistheoretischer Sicht diese Komponente unbedingt nötig ist, bedeutet sie für den Berechnungsprozeß nur eine unnötige Belastung. Natürlich könnten wir versuchen, sie im nachhinein mithilfe des spread-Konstrukts wieder auszublenden. Bei komplexeren Induktionsbeweisen wie dem in Beispiel 3.4.6 wird dies jedoch kaum (automatisch) durchführbar sein und zudem wäre es ohnehin effizienter, diese Anteile erst gar nicht in den Algorithmus einzufügen. Dies verlangt jedoch eine andersartige Formulierung des Spezifikationstheorems, bei der Existenzquantoren mit Typkonstrukten identifiziert werden, die *nur* das gewünschte Element als Evidenz enthalten und den ungewünschten Beweisterm unterdrücken.

- Ein zweites Problem war die etwas umständliche Formulierung der Tatsache, daß wir nichtleere Listen – also eine Teilmenge des Listentyps – als Eingabebereich der Spezifikation betrachten wollten. Natürlich hätten wir auch formulieren können

  $$\vdash\ \forall \mathtt{a}{:}\mathbb{Z}\,\mathsf{list}.\neg(\mathtt{a}{=}[\,] \in \mathbb{Z}\,\mathsf{list})\ \Rightarrow\ \exists \mathtt{L}{:}\mathbb{Z}.\exists \mathtt{M}{:}\mathbb{Z}.\ \mathtt{L}{=}\mathtt{maxbeg(a)}\ \wedge\ \mathtt{M}{=}\mathtt{maxseq(a)}.$$

  In diesem Fall wären wir jedoch gezwungen, als Eingabe für den extrahierten Algorithmus nicht nur eine Liste a, sondern auch einen Beweis dafür, daß a nicht leer ist, einzugeben. Was wir in Wirklichkeit formulieren wollen, ist daß der Eingabebereich die Menge $\{\mathtt{a}{:}\mathbb{Z}\,\mathsf{list} \mid \neg(\mathtt{a}{=}[\,] \in \mathbb{Z}\,\mathsf{list})\}$ ist. Mit den bisherigen Mitteln aber können wir das nicht ausdrücken, da es uns die Konstrukte fehlen, einen gegebenen Typ auf eine Teilmenge *einzuschränken*.

Wir benötigen also aus mehreren Gründen einen *Teilmengentyp* der Form $\{x{:}S \mid P[x]\}$, der aus allen Elementen von S besteht, von denen wir wissen, daß sie die Eigenschaft P besitzen – ohne daß wir hierzu einen separaten Beweis als Evidenz hinzugeben müssen. Ohne ein solches Konstrukt ist eine Formalisierung vieler Standardkonstrukte der Mathematik nicht praktisch durchführbar. Ohne einen Teilmengentyp müßten wir ein so einfaches Problem wie das in Beispiel 3.4.1 vorgestellte Spezifikationstheorem für die Berechnung der Integerquadratwurzel sehr umständlich formulieren, nämlich als

$$\vdash\ \forall \mathtt{x}{:}\mathbb{Z}.0\leq\mathtt{x}\ \Rightarrow\ \exists \mathtt{y}{:}\mathbb{Z}.\ 0\leq\mathtt{y}\ \wedge\ \mathtt{y}^2\leq\mathtt{x}\ \wedge\ \mathtt{x}<(\mathtt{y}+1)^2$$

und die eigentlich interessierende Fragestellung würde von den Randbedingungen völlig verschleiert. Miteinander verwandte Typen wie die ganzen und die natürlichen Zahlen können ohne ein Teilmengenkonstrukt nicht wirklich miteinander in Beziehung gesetzt werden und die Tatsache, daß jede natürliche Zahl auch als ganze Zahl zu verstehen ist, würde durch die explizit vorzunehmenden Konversionen völlig verschleiert.

Eine Simulation des Konzepts der Teilmengen durch bisherige Typkonstrukte ist nicht sinnvoll möglich. Als einzig denkbare Abstützung käme der Produkttyp in Frage – also eine Definition der Art $\{x{:}S \mid P\} \equiv x{:}S{\times}P$. Dies würde aber die Intuition, die hinter dem Konzept der Teilmenge steht, nur unvollständig wiederspiegeln. Zwar ist oberflächlich eine Ähnlichkeit vorhanden, weil die Werte, die man für x in $x{:}S{\times}P$ einsetzen darf, tatsächlich genau die Elemente von $\{x{:}S \mid P\}$ sind. Jedoch muß im Produkttyp immer noch eine zweite Komponente hinzukommen, nämlich ein Term, welcher $P[x]$ *beweist*. Diese Komponente aber wollen wir im Teilmengenkonstrukt gerade nicht haben. Es reicht uns, zu *wissen*, daß jedes Element x von $\{x{:}S \mid P\}$ die Eigenschaft $P[x]$ besitzt, aber umgehen wollen wir ausschließlich mit dem Element x selbst. Insbesondere in Algorithmen, die auf Teilmengen operieren, wollen wir nicht gezwungen sein, den Beweisterm als Eingabe mitzuliefern, der vom Algorithmus dann ohnehin weggeworfen wird. Dies wäre eine sehr unnatürliche Form von Algorithmen, die in der Welt der Programmierung niemals Fuß fassen könnte. Anstatt einer Simulation müssen wir daher nach einem eleganteren Weg suchen, Teilmengen zu beschreiben.

Diese Überlegungen führten schließlich dazu, die Typentheorie um einen Teilmengenkonstruktor $\{x{:}S \mid P\}$ zu ergänzen, welcher das Konzept der Teilmengenbildung direkt wiederspiegelt. Dies ist weniger eine Frage nach der syntaktischen Repräsentation als nach der semantischen Elementbeziehung, die präzise fixiert werden muß, und nach einer geeigneten Darstellung von *implizit vorhandenem Wissen* innerhalb der Inferenzregeln. Das Wissen, daß ein Element s von $\{x{:}S \mid P\}$ die Eigenschaft $P[s]$ besitzt, für die aber keine Evidenz – also kein Beweisterm – explizit mitgeliefert wird, muß so verwaltet werden, daß es in Beweisen verwendet werden kann, innerhalb der extrahierten Algorithmen aber keine Rolle spielt.[56]

Die formale Beschreibung des Teilmengentyps – insbesondere der Inferenzregeln – ist angelehnt an den Produkttyp $x{:}S{\times}T$, weicht aber an den Stellen davon ab, wo es um die explizite Benennung der zweiten Komponente t eines Paars $\langle s,t\rangle \in x{:}S{\times}T$ geht. Für den Teilmengentyp $\{x{:}S \mid T\}$ reicht es zu wissen, daß es ein solches $t \in T[s/x]$ gibt, denn entsprechend dem intuitiven Verständnis sind seine Elemente genau diejenigen Elemente s von S, für die der Typ $T[s/x]$ nicht leer – also beweisbar – ist. Eigene kanonische oder nichtkanonische Terme gibt es für den Teilmengentyp nicht und auch die Gleichheitsrelation auf den Elementen von S wird unverändert übernommen.

Die Verwaltung des Wissens, daß für ein Element s von $\{x{:}S \mid T\}$ der Typ $T[s/x]$ nicht leer ist, verlangt eine leichte Modifikation von Sequenzen und Beweisen. Einerseits müssen die Regeln dafür sorgen, daß zum Nachweis von $s \in \{x{:}S \mid T\}$ die Eigenschaft $T[s/x]$ überprüft wird. Andererseits aber ist dieses Wissen nicht im Term s selbst enthalten. Somit darf bei der Elimination von Mengenvariablen eine Evidenz für die Eigenschaft $T[s/x]$ nicht in den erzeugten Algorithmus eingehen.[57] Wir wollen dies an einem einfachen Beispiel erläutern.

[56] Es sei angemerkt, daß dies keine Aufgabe ist, für die es eine eindeutig optimale Antwort gibt, da die Mathematik mit dem Konzept der Teilmenge relativ nachlässig umgeht und es in einem hochgradig unkonstruktiv Sinne verwendet. In einer konstruktiven Denkwelt, die im Zusammenhang mit der Automatisierung des Schließens nun einmal nötig ist, kann diese Vorgehensweise, die bedenkenlos einem Objekt Eigenschaften zuweist, die aus diesem nur unter Hinzunahme zusätzlicher Informationen herleitbar ist, nur unvollständig nachgebildet werden. In verschiedenen Theorien sind hierfür unterschiedliche Lösungen vorgeschlagen worden, die jeweils ihre Stärken und haben. In [Salvesen & Smith, 1988] werden diese Unterschiede ausführlicher beleuchtet.

[57] Eine Ausnahme davon bilden natürlich diejenigen Eigenschaften, die wir *direkt* aus s ableiten können. So kann zum Beispiel für jedes konkrete Element $s \in \{\mathtt{i}{:}\mathbb{Z} \mid \mathtt{i}\geq 0\}$ die Eigenschaft $\mathtt{i}\geq 0[s/\mathtt{i}]$ mit der Regel `arith` bewiesen werden.

kanonisch (Typen)	(Elemente)	nichtkanonisch
$\mathbf{set}\{\}(S;\,x.T)$ $\{x{:}S \mid T\}$		

Zusätzliche Einträge in die Operatorentabelle

Redex	Kontraktum
— *entfällt* —	

Zusätzliche Einträge in die Redex–Kontrakta Tabelle

Typsemantik		
$\{x_1{:}S_1 \mid T_1\} = \{x_2{:}S_2 \mid T_2\}$	falls	$S_1{=}S_2$ und es gibt eine Variable x, die weder in T_1 noch in T_2 vorkommt, und zwei Terme p_1 und p_2 mit der Eigenschaft $p_1 \in \forall x{:}S_1.\,T_1[x/x_1] \Rightarrow T_2[x/x_2]$ und $p_2 \in \forall x{:}S_1.\,T_2[x/x_2] \Rightarrow T_1[x/x_1]$.
$T = \{S_2 \mid T_2\}$	falls	$T = \{x_2{:}S_2 \mid T_2\}$ für ein beliebiges $x_2 \in \mathcal{V}$.
$\{S_1 \mid T_1\} = T$	falls	$\{x_1{:}S_1 \mid T_1\} = T$ für ein beliebiges $x_1 \in \mathcal{V}$.
Elementsemantik		
$s = t \in \{x{:}S \mid T\}$	falls	$\{x{:}S \mid T\}$ Typ und $s = t \in S$ und es gibt einen Term p mit der Eigenschaft $p \in T[s/x]$
$\{x_1{:}S_1 \mid T_1\} = \{x_2{:}S_2 \mid T_2\} \in \mathsf{U}_j$	falls	$S_1{=}S_2 \in \mathsf{U}_j$ und $T_1[s/x_1] \in \mathsf{U}_j$ sowie $T_2[s/x_2] \in \mathsf{U}_j$ gilt für alle Terme s mit $s \in S_1$ und es gibt eine Variable x, die weder in T_1 noch in T_2 vorkommt, und zwei Terme p_1 und p_2 mit der Eigenschaft $p_1 \in \forall x{:}S_1.\,T_1[x/x_1] \Rightarrow T_2[x/x_2]$ und $p_2 \in \forall x{:}S_1.\,T_2[x/x_2] \Rightarrow T_1[x/x_1]$
$T = \{S_2 \mid T_2\} \in \mathsf{U}_j$	falls	$T = \{x_2{:}S_2 \mid T_2\} \in \mathsf{U}_j$ für ein beliebiges $x_2 \in \mathcal{V}$.
$\{S_1 \mid T_1\} = T \in \mathsf{U}_j$	falls	$\{x_1{:}S_1 \mid T_1\} = T \in \mathsf{U}_j$ für ein beliebiges $x_1 \in \mathcal{V}$.

Zusätzliche Einträge in den Semantiktabellen

$\Gamma \vdash \{x_1{:}S_1 \mid T_1\} = \{x_2{:}S_2 \mid T_2\} \in \mathsf{U}_j$ ⌊Ax⌋
by `setEq`
  $\Gamma \vdash S_1 = S_2 \in \mathsf{U}_j$ ⌊Ax⌋
  $\Gamma,\ x{:}S_1 \vdash T_1[x/x_1] = T_2[x/x_2] \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma \vdash s = t \in \{x{:}S \mid T\}$ ⌊Ax⌋
by `elementEq` j
  $\Gamma \vdash s = t \in S$ ⌊Ax⌋
  $\Gamma \vdash T[s/x]$ ⌊Ax⌋
  $\Gamma,\ x'{:}S \vdash T[x'/x] \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma \vdash \{x{:}S \mid T\}$ ⌊**ext** s⌋
by `elementI` $j\ s$
  $\Gamma \vdash s \in S$ ⌊Ax⌋
  $\Gamma \vdash T[s/x]$ ⌊Ax⌋
  $\Gamma,\ x'{:}S \vdash T[x'/x] \in \mathsf{U}_j$ ⌊Ax⌋

$\Gamma,\ z{:}\{x{:}S \mid T\},\ \Delta \vdash C$ ⌊**ext** $(\lambda y.t)\,z$⌋
by `setE` i
  $\Gamma,\ z{:}\{x{:}S \mid T\},\ y{:}S,\ [\![v]\!]{:}T[y/x],\ \Delta[y/z]$
    $\vdash C[y/z]$ ⌊**ext** t⌋

$\Gamma \vdash s = t \in \{S \mid T\}$ ⌊Ax⌋
by `elementEq_indep`
  $\Gamma \vdash s = t \in S$ ⌊Ax⌋
  $\Gamma \vdash T$ ⌊Ax⌋

$\Gamma \vdash \{S \mid T\}$ ⌊**ext** s⌋
by `elementI_indep`
  $\Gamma \vdash S$ ⌊**ext** s⌋
  $\Gamma \vdash T$ ⌊Ax⌋

Inferenzregeln

Abbildung 3.22: Syntax, Semantik und Inferenzregeln des Teilmengentyps

Beispiel 3.4.7

Es gibt zwei naheliegende Möglichkeiten, den Typ aller ganzzahligen Funktionen, welche eine Nullstelle besitzen, zu beschreiben.

$$F_0 \;\equiv\; \texttt{f}:\mathbb{Z}\to\mathbb{Z} \times \exists\texttt{y}:\mathbb{Z}.\,\texttt{f y = 0} \qquad\qquad \overline{F_0} \;\equiv\; \{\texttt{f}:\mathbb{Z}\to\mathbb{Z} \mid \exists\texttt{y}:\mathbb{Z}.\,\texttt{f y = 0}\}$$

Es ist relativ einfach, einen Algorithmus zu beschreiben, welcher für jedes Element g von F_0 eine Nullstelle bestimmt. Man muß hierzu nur bedenken, daß g in Wirklichkeit aus einem Paar $\langle f,p\rangle$ besteht, wobei p ein Beweis dafür ist, daß f eine Nullstelle besitzt. p selbst hat wiederum die Form $\langle y,p'\rangle$ und y ist genau die gesuchte Nullstelle. Kurzum, der Algorithmus muß nur auf die erste Komponente y der zweiten Komponente p von g zugreifen um das gewünschte Ergebnis zu liefern.

Eine solche Möglichkeit gibt es für die Elemente von $\overline{F_0}$ nicht. Statt eines direkten Zugriffs muß nun die Nullstelle *gesucht* werden, wobei für die Suche keine obere Grenze angegeben werden kann. In der bisherigen Typentheorie kann der allgemeine Algorithmus zur Bestimmung von Nullstellen daher nicht formuliert werden.[58]

In den Teilen eines Beweises, die zum extrahierten Algorithmus etwas beitragen, darf die Eigenschaft $T[s/x]$ also nicht benutzt werden. Die volle Bedeutung der Menge $\{x:S \mid T\}$ würde aber nicht erfaßt werden, wenn wir die Verwendung von $T[s/x]$ generell verbieten würden. In diesem Fall würde eine Deklaration der Form $s:\{x:S \mid T\}$ nämlich nicht mehr Informationen enthalten als die Deklaration $s:S$. Deshalb ist es nötig, die Information $T[s/x]$ bei der Elimination von Mengenvariablen in die Hypothesen mit aufzunehmen, aber in allen Teilbeweisen zu *verstecken*, die einen algorithmischen Anteil besitzen. Nur in 'nichtkonstruktiven' Teilbeweisen, also solchen, die – wie zum Beispiel alle direkten Beweise für Gleichheiten – Axiom als Extrakt-Term liefern, kann die versteckte Hypothese wieder freigegeben werden.

Um dies zu realisieren, müssen wir das Konzept der Sequenz um die Möglichkeit erweitern, Hypothesen zu verstecken und wieder freizugeben. Als Notation für eine versteckte Hypothese schließen wir die zugehörige Variable in (doppelte) eckige Klammern ein, wie zum Beispiel in $\underline{[\![v]\!]:T[y/x]}$. Dies kennzeichnet, daß das *Wissen* $T[y/x]$ vorhanden ist, aber die *Evidenz* v nicht konstruktiv verwendet werden kann. Versteckte Hypothesen werden von der Eliminationsregel für Mengen `setE` erzeugt (und später auch von der Eliminationsregel `quotient_eqE`für die Gleichheit im Quotiententyp). Versteckte Hypothesen können nur durch die Anwendung von Regeln, welche Axiom als Extrakt-Term erzeugen, wieder *freigegeben* werden. In den erzeugten Teilzielen trägt die Evidenz v nichts zum gesamten Extrakt-Term bei und darf deshalb wieder benutzt werden.

Zugunsten einer einheitlicheren Darstellung wird neben dem üblichen (abhängigen) Teilmengenkonstruktor $\{x:S \mid T\}$ auch eine unabhängige Version $\{S \mid T\}$ eingeführt. Dieser Typ besitzt entweder dieselben Elemente wie S, wenn T nicht leer ist, und ist leer im anderen Fall. Abbildung 3.22 beschreibt die entsprechenden Erweiterungen von Syntax, Semantik und Inferenzregelsystem der Typentheorie.[59] Der besseren Lesbarkeit wegen haben wir in der Beschreibung der Semantik von den in Definition 3.3.3 auf Seite 134 vorgestellten Logikoperatoren gemacht. Dies trägt dem Gedanken Rechnung, daß T_1 bzw. T_2 Typen sind, die als Propositionen aufgefaßt werden. Die in [Constable *et.al.*, 1986, Kapitel 8.2] gegebene Originaldefinition der Semantik von Mengentypen verwendet stattdessen die entsprechenden Funktionstypen.

Zum Abschluß dieses Abschnittes wollen wir das auf Seite 138 begonnene Beispiel der Integerquadratwurzel zu Ende führen. Wir ergänzen hierzu zunächst einige definitorische Abkürzungen, die für eine natürliche Behandlung von Teilbereichen der ganzen Zahlen von Bedeutung sind.

[58]Der Grund hierfür ist, daß die Typentheorie zugunsten der Beweisbarkeit von Eigenschaften nur terminierende Funktionen enthalten darf. Da die Menge der total-rekursiven Funktionen aber unentscheidbar ist, sind die bisher repräsentierbaren Funktionen nur primitiv rekursiv. Das Teilmengenkonstrukt liefert allerdings die Möglichkeit, *partiell-rekursive* Funktionen zu betrachten, die auf einer bekannten Teilmenge des Eingabetyps terminieren. Hierzu müssen wir das Prinzip der rekursiven Definition, welches wir im Prinzip mit dem **Y**-Kombinator (siehe Definition 2.3.22 auf Seite 57) simulieren könnten, genauer untersuchen und seine Eigenschaften so fixieren, daß wir Suchalgorithmen der gewünschten Art formalisieren und ihre Terminierung mithilfe der Information $g \in \{\texttt{f}:\mathbb{Z}\to\mathbb{Z} \mid \exists\texttt{y}:\mathbb{Z}.\,\texttt{f y = 0}\}$ nachweisen können. Dieses Thema werden wir im Abschnitt 3.5.2 vertiefen.

[59]Man beachte, daß durch die Hinzunahme des Teilmengenkonstruktors jetzt mehrere verschiedene Möglichkeiten zur Charakterisierung eines Termes bestehen. Während zuvor die Struktur der kanonischen Elemente bereits festlegte, zu welcher Art Typ sie gehören können, bleibt jetzt immer noch die Option offen, daß sie auch zu einem anderen Typ gehören, der über den Teilmengentyp gebildet wurde. So ist zum Beispiel `0` sowohl ein Element von $\mathbb{Z}$ als auch eines von $\mathbb{N} \equiv \{\texttt{n}:\mathbb{Z} \mid \texttt{n}\geq\texttt{0}\}$.

Definition 3.4.8 (Teilbereiche der ganzen Zahlen und zusätzliche Operationen)

$\mathbb{N}$	$\equiv$	**nat**$\{\}()$	$\equiv$	$\{\texttt{n}:\mathbb{Z} \mid \texttt{n}\geq\texttt{0}\}$
$\mathbb{N}^+$	$\equiv$	**nat_plus**$\{\}()$	$\equiv$	$\{\texttt{n}:\mathbb{N} \mid \texttt{n>0}\}$
$\{i..j\}$	$\equiv$	**int_iseg**$\{\}(i;j)$	$\equiv$	$\{\texttt{n}:\mathbb{Z} \mid i\leq\texttt{n} \wedge \texttt{n}\leq j\}$
n^2	$\equiv$	**square**$\{\}(n)$	$\equiv$	$n{*}n$

Beispiel 3.4.9 (Bestimmung der Integerquadratwurzel)

Die einfachste Form, die Existenz einer Integerquadratwurzel über das Theorem

$$\vdash \forall\texttt{x}:\mathbb{N}.\exists\texttt{y}:\mathbb{N}.\ \texttt{y}^2\leq\texttt{x} \ \wedge\ \texttt{x<(y+1)}^2$$

zu beweisen, besteht in einer simplen Induktion über der Zahl `x`. Falls `x=0` ist, so müssen wir ebenfalls `y=0` wählen. Andernfalls können wir davon ausgehen, daß wir die Integerquadratwurzel von `x-1`, die wir mit `z` bezeichnen, bereits kennen, d.h. es gilt $\texttt{z}^2\leq\texttt{x-1} \wedge \texttt{x-1<(z+1)}^2$. Gilt nun auch $\texttt{x<(z+1)}^2$, dann können wir `z` unverändert als Integerquadratwurzel von `x` übernehmen. Ansonsten müssen wir die nächstgrößere Zahl, also `z+1` als Integerquadratwurzel von `x` angeben. Der folgende formale NuPRL Beweis spiegelt dieses Argument genau wieder.

```
⊢ ∀x:ℕ.∃y:ℕ. y^2≤x ∧ x<(y+1)^2
by all_i
 \
  x:ℕ ⊢ ∃y:ℕ. y^2≤x ∧ x<(y+1)^2
  by natE 1
  |\
  | x:ℕ ⊢ ∃y:ℕ. y^2≤0 ∧ 0<(y+1)^2
  | by ex_i 0
  | |\
  | | x:ℕ ⊢ 0^2≤0 ∧ 0(0+1)^2
  | | by ......... and_i, arith
  |  \
  |   x:ℕ ⊢ 0 ∈ ℕ
  |   by elementEq 1
  |   |\
  |   | x:ℕ ⊢ 0 ∈ ℤ
  |   | by natnumEq
  |   |\
  |   | x:ℕ ⊢ 0≥0
  |   | by arith 1
  |    \
  |     x:ℕ, i:ℤ ⊢ i≥0 ∈ U1
  |     by ......... funEq, voidEq, ltEq, natnumEq, hypEq
   \
    x:ℕ, n:ℕ, 0<n, v:∃y:ℕ. y^2≤n-1 ∧ n-1<(y+1)^2 ⊢ ∃y:ℕ. y^2≤n ∧ n<(y+1)^2
     \
      by ex_e 4 THEN and_e 5
       \
        x:ℕ,n:ℕ,0<n,y:ℕ,y^2≤n-1,n-1<(y+1)^2 ⊢ ∃y:ℕ. y^2≤n ∧ n<(y+1)^2
        by cut 6  n<(y+1)^2 ∨ n=(y+1)^2
        |\            | x:ℕ, n:ℕ, 0<n, y:ℕ,  y^2≤n-1, n-1<(y+1)^2 ⊢ n<(y+1)^2 ∨ n=(y+1)^2
        | \ by ... arith
         \
          x:ℕ,n:ℕ,0<n,y:ℕ,y^2≤n-1,n-1<(y+1)^2, n<(y+1)^2 ∨ n=(y+1)^2 ⊢ ∃y:ℕ. y^2≤n ∧ n<(y+1)^2
          by or_e 7
          |\
          | x:ℕ,n:ℕ,0<n,y:ℕ,y^2≤n-1,n-1<(y+1)^2, n<(y+1)^2 ⊢ ∃y:ℕ. y^2≤n ∧ n<(y+1)^2
          | by ex_i y
          | |\
          | | x:ℕ,n:ℕ,0<n,y:ℕ,y^2≤n-1,n-1<(y+1)^2, n<(y+1)^2 ⊢  y^2≤n ∧ n<(y+1)^2
          | | by ......... and_i, arith, hyp
          |  \
          |   x:ℕ,n:ℕ,0<n,y:ℕ,y^2≤n-1,n-1<(y+1)^2, n<(y+1)^2 ⊢ y∈ℕ
          |   by hypEq 4
           \
            x:ℕ,n:ℕ,0<n,y:ℕ,y^2≤n-1,n-1<(y+1)^2, n=(y+1)^2 ⊢ ∃y:ℕ. y^2≤n ∧ n<(y+1)^2
           by ex_i y+1
            |\
            | x:ℕ,n:ℕ,0<n,y:ℕ,y^2≤n-1,n-1<(y+1)^2, n=(y+1)^2 ⊢ (y+1)^2≤n ∧ n<(y+1+1)^2
            | by ......... and_i, arith
             \
              x:ℕ,n:ℕ,0<n,y:ℕ,y^2≤n-1,n-1<(y+1)^2, n<(y+1)^2 ⊢ y+1∈ℕ
              by ...... addEq, hypEq, natnumEq
```

In dem formalen Beweis müssen wir berücksichtigen, daß der Typ $\mathbb{N}$ als Teilmenge der ganzen Zahlen dargestellt wird und daß die Induktion über ganzen Zahlen auch die negativen Zahlen betrachtet, die in $\mathbb{N}$ gar nicht vorkommen. Die Induktion über `x` – also die Regel `natE` aus Abschnitt 3.4.2.1 – besteht daher aus mehreren Einzelschritten: der Elimination der Menge mit `setE`, der Elimination der ganzen Zahl mit `intE` und der vollständigen Behandlung des negativen Induktionsfalls, der durch arithmetisches Schließen mittels `arith` als widersprüchlich nachgewiesen wird. Hierfür sind weitere Detailsschritte erforderlich, welche die versteckte Hypothese freigeben und zu der Induktionsannahme in Beziehung setzen.

Die Schritte, die hierbei ausgeführt werden, sind in allen Fällen dieselben – hängen also nicht vom konkreten Beweisziel ab. Wir können daher die Regel `natE` als eine definitorische Abkürzung für diese Folge von Einzelschritten betrachten, wobei die Zwischenergebnisse nicht gezeigt werden. Wir hätten, wie man leicht sieht, auch noch weitere Schritte des Beweises zusammenfassen können. So ist zum Beispiel die Typüberprüfung $0 \in \mathbb{N}$, die wir etwas ausführlicher beschrieben haben, eine logische Einheit. Das gleiche gilt für die Fallanalyse, die wir mit der `cut`-Regel eingeleitet haben. Faßt man diese Schritte zu einer jeweils größeren Regel zusammen so ist es möglich, den NuPRL Beweis lesbarer und ähnlicher zum informalen Beweis zu gestalten. Den 'Taktik'-Mechanismus, der es uns erlaubt, elementare Inferenzregeln auf diese Art zu kombinieren, werden wir in Abschnitt 4.2 ausführlich besprechen.

Der obige Beweis liefert uns einen Algorithmus, der im wesentlichen die Gestalt

$$\lambda \mathtt{x}.\ \mathtt{ind(x;\ \_,\_,\_;\ 0;\ n,y.if\ n<(y+1)}^2\ \mathtt{then\ y\ else\ y+1)}$$

hat. Dieser Algorithmus ist linear in `x` und der effizienteste Algorithmus, den man mit Mitteln der Induktion bzw. primitiven Rekursion (also FOR-Schleifen) erreichen kann. Es gibt jedoch auch wesentlich effizientere Algorithmen zur Berechnung der Integerquadratwurzel, nämlich zum Beispiel die lineare Suche nach dem kleinsten Wert `y`, für den $\mathtt{(y+1)}^2\ \mathtt{>\ x}$ gilt. Ein solcher Algorithmus verwendet jedoch eine allgemeinere Form der Rekursion, die mit einfacher Induktion nicht mehr dargestellt werden kann. Die bisherige Ausdruckskraft der Typentheorie reicht daher für eine Betrachtung und Erzeugung realistischer Programme immer noch nicht aus und wir werden sie um eine allgemeinere Form der Rekursion erweitern müssen. Diese Erweiterung werden wir im Abschnitt 3.5 besprechen.

3.4.5 Quotienten

Das Teilmengenkonstrukt des vorhergehenden Abschnitts ermöglicht uns, die Typzugehörigkeitsrelation eines vorgegebenen Datentyps durch Restriktion zu verändern. Auf diese Art können wir verschiedenartige Typen wie die ganzen und die natürlichen Zahlen miteinander in Verbindung bringen, ohne eine Konversion von Elementen vornehmen zu müssen. Es gibt jedoch noch eine andere sinnvolle Möglichkeit, zwei verschieden Typen in Beziehung zu setzen, die im Prinzip doch die gleichen Elemente besitzen. So werden zum Beispiel in der Analysis die rationalen Zahlen mit Paaren ganzer Zahlen und die rellen Zahlen mit konvergierenden unendlichen Folgen rationaler Zahlen identifiziert. Ein Unterschied besteht jedoch darin, wann zwei Elemente als gleich zu gelten haben. Während die Gleichheit von Paaren von Zahlen elementeweise bestimmt wird werden zwei rationale Zahlen als gleich betrachtet, wenn die entsprechenden gekürzten Formen gleich sind. Zwei Folgen rationaler Zahlen sind gleich als relle Zahl, wenn sie gegen den gleichen Grenzwert konvergieren.[60] Somit werden zwar jeweils dieselben Elemente betrachtet, aber die Gleichheitsrelation ist verändert.

Mathematisch betrachtet besteht diese Veränderung in einer Restklassenbildung. Der Typ T der betrachteten Elemente bleibt im Prinzip unverändert, aber er wird ergänzt um eine neue Gleichheitsrelation E (*'equality'*), welche ab nun die semantische Gleichheit des neuen Typs bestimmt. Natürlich muß E hierfür tatsächlich eine *Äquivalenzrelation* sein. Die Schreibweise für diesen Typ ist $\underline{x,y : T/\!/E}$, wobei T ein Typ ist und E ein Äquivalenzprädikat (also auch ein Typ), welches von den Variablen $x, y \in T$ abhängen kann. Die Elemente des so entstandenen Typs sind die Elemente von T. Zwei Elemente s und t gelten als gleich, wenn sie in der Relation E stehen, also wenn es einen Beweis für (ein Element von) $E[s,t\,/\,x,y]$ gibt.

[60] Diese Gleichheiten können natürlich alleine auf der Basis der Eigenschaften von Zahlenpaaren bzw. von Folgen rationaler Zahlen definiert werden: $\langle$`6,4`$\rangle$=$\langle$`9,6`$\rangle$ gilt, weil `6*6=4*9` ist.

kanonisch (Typen)	(Elemente)	**nichtkanonisch**
quotient{}$(T\,;x\,,y\,.E)$ $x\,,y : T/\!/E$		

Zusätzliche Einträge in die Operatorentabelle

Redex	**Kontraktum**
— entfällt —	

Zusätzliche Einträge in die Redex–Kontrakta Tabelle

Typsemantik		
$x_1,y_1 : T_1/\!/E_1 \ = x_2,y_2 : T_2/\!/E_2$	falls	$T_1 = T_2$ und es gibt (verschiedene) Variablen x, y, z, die weder in E_1 noch in E_2 vorkommen, und Terme p_1, p_2, r,s und t mit der Eigenschaft $p_1 \in \forall x{:}T_1 . \forall y{:}T_1 .\ E_1[x,y/x_1,y_1] \Rightarrow E_2[x,y/x_2,y_2]$ und $p_2 \in \forall x{:}T_1 . \forall y{:}T_1 .\ E_2[x,y/x_2,y_2] \Rightarrow E_1[x,y/x_1,y_1]$ und $r \in \forall x{:}T_1 . E_1[x,x/x_1,y_1]$ und $s \in \forall x{:}T_1 . \forall y{:}T_1 .\ E_1[x,y/x_1,y_1] \Rightarrow E_1[y,x/x_1,y_1]$ und $t \in \forall x{:}T_1 . \forall y{:}T_1 . \forall z{:}T_1 .$ $E_1[x,y/x_1,y_1] \Rightarrow E_1[y,z/x_1,y_1] \Rightarrow E_1[x,z/x_1,y_1]$

Elementsemantik		
$s = t \ \in \ x,y : T/\!/E$	falls	$x,y : T/\!/E$ Typ und $s \in T$ und $t \in T$ und es gibt einen Term p mit der Eigenschaft $p \in E[s,t/x,y]$
$x_1,y_1 : T_1/\!/E_1 \ = x_2,y_2 : T_2/\!/E_2 \ \in \mathsf{U}_j$	falls	$T_1 = T_2 \in \mathsf{U}_j$ und für alle Terme s,t mit $s \in T_1$ und $t \in T_1$ gilt $E_1[s,t/x_1,y_1] \in \mathsf{U}_j$ sowie $E_2[s,t/x_2,y_2] \in \mathsf{U}_j$ und es gibt (verschiedene) Variablen x, y, z, die weder in E_1 noch in E_2 vorkommen, und Terme p_1, p_2, r,s und t mit der Eigenschaft $p_1 \in \forall x{:}T_1 . \forall y{:}T_1 .\ E_1[x,y/x_1,y_1] \Rightarrow E_2[x,y/x_2,y_2]$ und $p_2 \in \forall x{:}T_1 . \forall y{:}T_1 .\ E_2[x,y/x_2,y_2] \Rightarrow E_1[x,y/x_1,y_1]$ und $r \in \forall x{:}T_1 . E_1[x,x/x_1,y_1]$ und $s \in \forall x{:}T_1 . \forall y{:}T_1 .\ E_1[x,y/x_1,y_1] \Rightarrow E_1[y,x/x_1,y_1]$ und $t \in \forall x{:}T_1 . \forall y{:}T_1 . \forall z{:}T_1 .$ $E_1[x,y/x_1,y_1] \Rightarrow E_1[y,z/x_1,y_1] \Rightarrow E_1[x,z/x_1,y_1]$

Zusätzliche Einträge in den Semantiktabellen

Abbildung 3.23: Syntax und Semantik des Quotiententyps

Bei der Formalisierung derartiger *Quotiententypen* ist jedoch zu beachten, daß die Evidenz für die veränderte Gleichheit – wie die Evidenz für die Einschränkung bei Teilmengen – nicht als Bestandteil des Elements selbst mitgeführt wird, sondern eine 'abstrakte' Eigenschaft ist. Das Wissen um die veränderte Gleichheit ist in $s{=}t \ \in \ x,y : T/\!/E$ also nur implizit enthalten und darf nicht zu einem Algorithmus beitragen, der aus einem entsprechenden Beweis enthalten ist. Daher muß die Regel, welche Gleichheiten in Kombination mit Quotiententypen analysiert (`quotient_eqE`) ebenfalls eine versteckte Hypothese generieren, welche erst durch Regeln freigegeben wird, deren Extrakt-Term Axiom ist. Alle anderen Bestandteile des Quotiententyps, die in den Abbildungen 3.23 und 3.24 beschrieben sind, ergeben sich direkt aus einer Ausformulierung des Konzeptes der Restklassen unter gegebenen Äquivalenzrelationen.[61]

Ein typisches Beispiel für die Anwendung des Quotiententyps ist die Formalisierung der rationalen Zahlen. Die folgende Definition entstammt [Constable *et.al.*, 1986, Kapitel 11.5] und wird dort ausführlicher diskutiert.

[61] Es sei angemerkt, daß zugunsten des einfacheren Übergangs zwischen T und $x,y : T/\!/E$ zwei "schwache" Regeln ergänzt wurden, in denen die Eigenschaften der Restklassenbildung nur zu einem geringen Teil ausgenutzt werden. Diese Regeln "vergeuden" Informationen in dem Sinne, daß die erzeugten Teilziele viel mehr beweisen als im ursprünglichen Ziel gefordert wurde.

$\Gamma \vdash x_1,y_1:T_1/\!/E_1 = x_2,y_2:T_2/\!/E_2 \in U_j$ ⌊Ax⌋
by quotientEq_weak
  $\Gamma \vdash T_1 = T_2 \in U_j$ ⌊Ax⌋
  $\Gamma,\ x{:}T_1,\ y{:}T_1 \vdash E_1[x,y/x_1,y_1] = E_2[x,y/x_2,y_2] \in U_j$ ⌊Ax⌋
  $\Gamma,\ x{:}T_1,\ y{:}T_1 \vdash E_1[x,x/x_1,y_1]$ ⌊Ax⌋
  $\Gamma,\ x{:}T_1,\ y{:}T_1,\ v{:}E_1[x,y/x_1,y_1] \vdash E_1[y,x/x_1,y_1]$ ⌊Ax⌋
  $\Gamma,\ x{:}T_1,\ y{:}T_1,\ z{:}T_1,\ v{:}E_1[x,y/x_1,y_1],\ v'{:}E_1[y,z/x_1,y_1] \vdash E_1[x,z/x_1,y_1]$ ⌊Ax⌋

$\Gamma \vdash x_1,y_1:T_1/\!/E_1 = x_2,y_2:T_2/\!/E_2 \in U_j$ ⌊Ax⌋
by quotientEq
  $\Gamma \vdash x_1,y_1:T_1/\!/E_1 \in U_j$ ⌊Ax⌋
  $\Gamma \vdash x_2,y_2:T_2/\!/E_2 \in U_j$ ⌊Ax⌋
  $\Gamma \vdash T_1 = T_2 \in U_j$ ⌊Ax⌋
  $\Gamma,\ v{:}T_1{=}T_2 \in U_j,\ x{:}T_1,\ y{:}T_1 \vdash E_1[x,y/x_1,y_1] \Rightarrow E_2[x,y/x_2,y_2]$ ⌊Ax⌋
  $\Gamma,\ v{:}T_1{=}T_2 \in U_j,\ x{:}T_1,\ y{:}T_1 \vdash E_2[x,y/x_2,y_2] \Rightarrow E_1[x,y/x_1,y_1]$ ⌊Ax⌋

$\Gamma \vdash s = t \in x,y:T/\!/E$ ⌊Ax⌋
by memberEq_weak j
  $\Gamma \vdash x,y:T/\!/E \in U_j$ ⌊Ax⌋
  $\Gamma \vdash s = t \in T$ ⌊Ax⌋

$\Gamma \vdash x,y:T/\!/E$ ⌊**ext** t⌋
by memberI j
  $\Gamma \vdash x,y:T/\!/E \in U_j$ ⌊Ax⌋
  $\Gamma \vdash T$ ⌊**ext** t⌋

$\Gamma \vdash s = t \in x,y:T/\!/E$ ⌊Ax⌋
by memberEq j
  $\Gamma \vdash x,y:T/\!/E \in U_j$ ⌊Ax⌋
  $\Gamma \vdash s \in T$ ⌊Ax⌋
  $\Gamma \vdash t \in T$ ⌊Ax⌋
  $\Gamma \vdash E[s,t/x,y]$ ⌊Ax⌋

$\Gamma,\ v{:}s{=}t \in x,y:T/\!/E,\ \Delta \vdash C$ ⌊**ext** u⌋
by quotient_eqE $i\ j$
  $\Gamma,\ v{:}s{=}t \in x,y:T/\!/E,\ [\![v']\!]{:}E[s,t/x,y],\ \Delta \vdash C$ ⌊**ext** u⌋
  $\Gamma,\ v{:}s{=}t \in x,y:T/\!/E,\ \Delta,\ x'{:}T,\ y'{:}T \vdash E[x',y'/x,y] \in U_j$ ⌊Ax⌋

$\Gamma,\ z{:}x,y:T/\!/E,\ \Delta \vdash s{=}t \in S$ ⌊Ax⌋
by quotientE $i\ j$
  $\Gamma,\ z{:}x,y:T/\!/E,\ \Delta,\ x'{:}T,\ y'{:}T \vdash E[x',y'/x,y] \in U_j$ ⌊Ax⌋
  $\Gamma,\ z{:}x,y:T/\!/E,\ \Delta \vdash S \in U_j$ ⌊Ax⌋
  $\Gamma,\ z{:}x,y:T/\!/E,\ \Delta,\ x'{:}T,\ y'{:}T,\ v{:}E[x',y'/x,y] \vdash s[x'/z] = t[y'/z] \in S[x'/z]$ ⌊Ax⌋

Abbildung 3.24: Inferenzregeln des Quotiententyps

Definition 3.4.10 (Rationale Zahlen)

$$
\begin{array}{lll}
x_1 =_q x_2 & \equiv \ \mathbf{rat\_equal}\{\}(x_1;x_2) & \equiv \ \mathsf{let}\ \langle z_1,n_1\rangle = x_1 \ \mathsf{in}\ \mathsf{let}\ \langle z_2,n_2\rangle = x_2 \ \mathsf{in}\ z_1{*}n_2{=}z_2{*}n_1 \\
\mathbb{Q} & \equiv \ \mathbf{rat}\{\}() & \equiv \ \mathtt{x,y} : \mathbb{Z}\times\mathrm{IN}^{+}/\!/_{\mathtt{x}=_q\mathtt{y}} \\
x_1 + x_2 & \equiv \ \mathbf{rat\_add}\{\}(x_1;x_2) & \equiv \ \mathsf{let}\ \langle z_1,n_1\rangle = x_1 \ \mathsf{in}\ \mathsf{let}\ \langle z_2,n_2\rangle = x_2 \ \mathsf{in}\ \langle z_1{*}n_2{+}z_2{*}n_1,\ n_1{*}n_2\rangle \\
x_1 - x_2 & \equiv \ \mathbf{rat\_sub}\{\}(x_1;x_2) & \equiv \ \mathsf{let}\ \langle z_1,n_1\rangle = x_1 \ \mathsf{in}\ \mathsf{let}\ \langle z_2,n_2\rangle = x_2 \ \mathsf{in}\ \langle z_1{*}n_2{-}z_2{*}n_1,\ n_1{*}n_2\rangle \\
x_1 * x_2 & \equiv \ \mathbf{rat\_mul}\{\}(x_1;x_2) & \equiv \ \mathsf{let}\ \langle z_1,n_1\rangle = x_1 \ \mathsf{in}\ \mathsf{let}\ \langle z_2,n_2\rangle = x_2 \ \mathsf{in}\ \langle z_1{*}z_2,\ n_1{*}n_2\rangle
\end{array}
$$

Aus mathematischer Sicht ist der Quotiententyp ein sehr mächtiger Abstraktionsmechanismus, denn er ermöglicht, *benutzerdefinierte Gleichheiten* in die Theorie auf eine Art mit aufzunehmen, daß alle Gleichheitsregeln – insbesondere die Substitutionsregel und eine Entscheidungsprozedur für Gleichheit (die `equality` Regel in Abbildung 3.28) – hierauf anwendbar werden. Dieser Mechanismus muß jedoch mit großer Sorgfalt eingesetzt werden, da die Gleichheit von kanonischen Elementen eines Quotiententyps – im Gegensatz zur Elementgleichheit bei anderen Typkonstrukten – nicht mehr von der Struktur dieser Elemente abhängt sondern von der benutzerdefinierten Gleichheitsrelation.[62] Somit müssen Konstruktionen, an denen Objekte eines Quotiententyps beteiligt sind, unabhängig von der speziellen Darstellung dieser Objekte sein. Genauer gesagt, ein Typ $T[z]$, der von einer Variablen z eines Quotiententyps $Q \equiv x,y : S/\!/E$ abhängt, muß so gestaltet sein, daß eine spezielle Instanz $T[s/z]$ nicht davon abhängt, welchen Term $s \in S$ man gewählt hat: für gleiche Elemente s_1, s_2 von Q – also Elemente, für die $E[s_1, s_2 \,/\, x, y]$ gilt – müssen $T[s_1/z]$ und $T[s_2/z]$ gleich sein.

[62] Die uns vertraute Mathematik geht mit dem Konzept der Restklassen – wie im Falle der Teilmengen – leider etwas zu sorglos um. Bei einer vollständigen Formalisierung fallen die kleinen Unstimmigkeiten allerdings auf und müssen entsprechend behandelt werden. Deshalb sind in beiden Fällen die Formalisierungen komplizierter als dies dem intuitiven Verständnis nach sein müsste.

Bei Typen $T[z]$, die tatsächlich einen Datentyp darstellen, ist dies normalerweise kein Problem, da sich hier das intuitive Verständnis mit der formalen Repräsentation deckt. Anders wird dies jedoch, wenn $T[z]$ eine logische Aussage darstellt. Üblicherweise verbindet man hiermit dann nur einen Wahrheitswert und vergißt, daß die formale Repräsentation auch die *Struktur* der Beweise wiedergibt, die durchaus von der speziellen Instanz für die Variable z abhängen kann. Wir wollen dies an einem Beispiel illustrieren.

Beispiel 3.4.11

In Definition 3.4.10 haben wir die rationalen Zahlen über Paare von ganzen und positiven Zahlen definiert, wobei wir die Definition der Gleichheit durch "Ausmultiplizieren" auf die Gleichheit ganzer Zahlen zurückgeführt haben. In gleicher Weise könnte man nun versuchen, andere wichtige Vergleichsoperationen wie die Relation $<$ auf rationale Zahlen fortzusetzen, was zu folgender Definition führen würde:

$$x_1{<}x_2 \ \equiv\ \mathsf{let}\ \langle z_1,n_1\rangle = x_1\ \mathsf{in}\ \mathsf{let}\ \langle z_2,n_2\rangle = x_2\ \mathsf{in}\ z_1{*}n_2{<}z_2{*}n_1$$

Im Gegensatz zu dem intuitiven Verständnis der Relation $<$ haben wir bei dieser Definition jedoch eine Struktur aufgebaut und nicht etwa nur ein Prädikat, das nur wahr oder falsch sein kann. Wie sieht es nun mit der Unabhängigkeit dieser Struktur von der Darstellung rationaler Zahlen aus? Gilt das Urteil $x_1{<}x_2 = x_1'{<}x_2'$, wenn $x_1{=}x_1' \in \mathbb{Q}$ und $x_2{=}x_2' \in \mathbb{Q}$ gilt?

Um dies zu überprüfen, müssen wir kanonische Terme von $\mathbb{Q}$ betrachten, das `spread`-Konstrukt reduzieren und dann die Typsemantik der Relation `<` auf ganzen Zahlen berücksichtigen. Gemäß dem Eintrag in Abbildung 3.17 gilt jedoch

$$z_1{*}n_2{<}z_2{*}n_1 \ =\ z_1'{*}n_2'{<}z_2'{*}n_1'$$

nur, wenn die jeweiligen Terme links und rechts vom Symbol `<` als ganze Zahlen gleich sind, also wenn

$$z_1{*}n_2 \ =\ z_1'{*}n_2' \ \in\ \mathbb{Z} \quad \text{und} \quad z_2{*}n_1 \ =\ z_2'{*}n_1' \ \in\ \mathbb{Z} \quad \text{gilt.}$$

Daß dies nicht immer der Fall ist, zeigt das Beispiel $x_1{=}\langle 2,1\rangle$, $x_1'{=}\langle 4,2\rangle$, $x_2{=}x_2'{=}\langle 1,1\rangle$, denn es gilt weder $2{*}1 = 4{*}1 \in \mathbb{Z}$ noch $1{*}1 = 1{*}2 \in \mathbb{Z}$

Diese semantische Analyse macht klar, daß Beweisziele der Art $x_1{:}\mathbb{Q},\ x_2{:}\mathbb{Q} \vdash x_1{<}x_2 \in U_1$ unter Verwendung der obigen Definition nicht bewiesen werden können, weil die Regel `quotientE`, die irgenwann im Verlauf des Beweises angewandt werden *muß*, genau das obige Problem aufdeckt.

Wie kann man dieses Problem nun lösen? Sicherlich wäre es nicht sinnvoll, die Semantik der Relation `<` auf ganzen Zahlen so abzuschwächen, daß Typgleichheit $i_1{<}j_1 = i_2{<}j_2$ gilt, wenn entweder beide Relationen erfüllt oder beide Relationen falsch sind, denn dies würde nur das spezielle Problem aus Beispiel 3.4.11 lösen und wir hätten dasselbe Problem bei anderen Prädikaten, die über die Relation `<` auf rationalen Zahlen definiert werden.[63] Wir müssen stattdessen überlegen, wie wir die Überstrukturierung vermeiden können, die wir bei der Definition in Beispiel 3.4.11 erzeugt haben.

Einen sehr einfachen Weg, einen strukturierten Datentyp P, der eigentlich nur eine logische Aussage repräsentieren soll, in einen unstrukturierten Datentyp umzuwandeln, bietet die Verwendung des unabhängigen Teilmengenkonstrukts. Der Typ $\{0 \in \mathbb{Z} \mid P\}$ besteht aus den Elementen von $0 \in \mathbb{Z}$ (d.h. aus Axiom), falls P Elemente hat, und ist andernfalls leer. Die spezifische Information, auf welche Art P die Elemente von P zu konstruieren sind, wird unterdrückt und spielt entsprechend der Semantik des Teilmengenkonstrukts in Abbildung 3.22 auch bei der Typgleichheit keine Rolle mehr. Dies deckt sich genau mit der Sichtweise von P als eine logische Aussage, die entweder wahr (beweisbar) oder falsch (unbeweisbar) ist. Diese Technik, durch Verwendung des unabhängigen Teilmengenkonstrukts die Struktur aus einem Datentyp P zu entfernen und diesen auf 'Wahrheit' zu reduzieren, nennt man *Type-Squashing* (Zerdrücken eines Typs). Sie wird durch die folgende konservative Erweiterung des Typsystems unterstützt.

Definition 3.4.12 (Type-Squashing)

$$\|T\| \ \equiv\ \mathbf{squash}\{\}(T) \ \equiv\ \{0 \in \mathbb{Z} \mid T\}$$

[63] Außerdem würde eines der Grundkonzepte der Typentheorie verletzt, Gleichheit *strukturell* zu definieren, d.h. von der Gleichheit aller beteiligter Teilterme abhängig zu machen.

Type-Squashing bietet sich an, wenn wir Prädikate über einem Quotiententyp $x,y : S/\!/E$ definieren wollen. Durch die Verwendung des Quotiententyps haben wir als Benutzer der Theorie die Gleichheit auf dem zugrundeliegenden Datentyp S verändert. Daher müssen wir als Benutzer bei der Einführung von Prädikaten auch selbst dafür sorgen, daß diese tatsächlich Prädikate über dem selbstdefinierten Quotiententyp bilden.[64]

Wir wollen die Verwendung von Type-Squashing am Beispiel einer Formalisierung der reellen Zahlen in der Typentheorie illustrieren. Diese kann man durch Cauchy-Folgen rationaler Zahlen beschreiben, also durch unendliche Folgen rationaler Zahlen (Funktionen von $\mathbb{N}^+$ nach $\mathbb{Q}$), deren Abstände gegen Null konvergieren. Um dies zu präzisieren, müssen wir zunächst die Relation < auf rationalen Zahlen geeignet definieren.

Definition 3.4.13 (Reelle Zahlen)

$x_1 < x_2$	$\equiv$ **rat_lt**$\{\}(x_1;x_2)$	$\equiv \Vert \text{let } \langle z_1,n_1\rangle = x_1 \text{ in let } \langle z_2,n_2\rangle = x_2 \text{ in } z_1 * n_2 < z_2 * n_1 \Vert$
$x_1 \leq x_2$	$\equiv$ **rat_le**$\{\}(x_1;x_2)$	$\equiv x_1 < x_2 \ \vee\ x_1 = x_2 \in \mathbb{Q}$
z/n	$\equiv$ **rat_frac**$\{\}(z;n)$	$\equiv \langle z,n\rangle$
$\lvert x\rvert$	$\equiv$ **rat_abs**$\{\}(x)$	$\equiv \text{let } \langle z,n\rangle = x \text{ in if } z<0 \text{ then } \langle -z,n\rangle \text{ else } \langle z,n\rangle$
$\mathbb{R}_{pre}$	$\equiv$ **real_pre**$\{\}()$	$\equiv \{\mathtt{f}:\mathbb{N}^+ \to \mathbb{Q} \mid \forall \mathtt{m},\mathtt{n}:\mathbb{N}^+.\ \lvert\mathtt{f(n)} - \mathtt{f(m)}\rvert \leq \mathtt{1/m+1/n}\}$
$x_1 =_r x_2$	$\equiv$ **real_equal**$\{\}(x_1;x_2)$	$\equiv \forall \mathtt{n}:\mathbb{N}^+.\ \lvert x_1(\mathtt{n}) - x_2(\mathtt{n})\rvert \leq \mathtt{2/n}$
$\mathbb{R}$	$\equiv$ **real**$\{\}()$	$\equiv \mathtt{x},\mathtt{y} : \mathbb{R}_{pre}/\!/\mathtt{x} =_r \mathtt{y}$
$x_1 + x_2$	$\equiv$ **real_add**$\{\}(x_1;x_2)$	$\equiv \lambda\mathtt{n}.\ x_1(\mathtt{n}) + x_2(\mathtt{n})$
$x_1 - x_2$	$\equiv$ **real_sub**$\{\}(x_1;x_2)$	$\equiv \lambda\mathtt{n}.\ x_1(\mathtt{n}) - x_2(\mathtt{n})$
$\lvert x\rvert$	$\equiv$ **real_abs**$\{\}(x)$	$\equiv \lambda\mathtt{n}.\ \lvert x(\mathtt{n})\rvert$

Auch den Datentyp $\mathbb{B}$ könnte man als Restklasse der ganzen Zahlen betrachten und definieren:

$$\mathbb{B} \equiv \mathtt{i},\mathtt{j} : \mathbb{Z}/\!/(\mathtt{i}\ \mathrm{rem}\ 2 = \mathtt{j}\ \mathrm{rem}\ 2)$$

Gerade Zahlen stünden in diesem Fall für **T** und ungerade Zahlen für **F**. Wegen der gedanklichen Verwandschaft zur disjunkten Vereinigung wird $\mathbb{B}$ jedoch üblicherweise als Summe zweier einelementiger Typen definiert.

3.4.6 Strings

Der Vollständigkeit halber enthält die Typentheorie einen Datentyp der Strings, der mit Atom bezeichnet wird. Dadurch wird es möglich, Programme zu betrachten, die feste Textketten als Antworten oder Meldungen ausgeben, ohne diese weiter zu verarbeiten. Die kanonischen Elemente des Typs Atom sind daher einfache Textketten, die in (doppelte) Anführungszeichen gesetzt sind.[65] Die Gleichheit zweier Strings ist leicht zu entscheiden: sie müssen textlich identisch sein. Zur Analyse steht ein Test auf Gleichheit (if u=v then s else t) zur Verfügung. Die zugehörigen Ergänzungen des Typsystems sind in Abbildung 3.25 zusammengestellt.

3.5 Rekursion in der Typentheorie

Im vorigen Abschnitt haben wir den Zusammenhang zwischen der Konstruktion formaler Beweise und der Entwicklung von Programmen hervorgehoben und die Vorteile dieser Denkweise illustriert. Dabei stellte sich natürlich heraus, daß die Mächtigkeit und Eleganz der so erzeugten Programme durch die Ausdruckskraft des zugrundeliegenden Inferenzkalküls beschränkt ist. Solange man sich auf reine Prädikatenlogik beschränkt, kann man nur Programme generieren, die aus elementaren Substitutionen und Fallunterscheidungen durch einfache Kompositionen zusammengesetzt sind. Durch die Hinzunahme von Induktionsbeweisen auf Zahlen

[64] Es sei allerdings angemerkt, daß das Mittel des Type-Squashing sehr grob ist und eigentlich zu viele Informationen unterdrückt. Wenn wir Teilinformationen eines Prädikats in Analysen verwenden wollen, die zu einem extrahierten Algorithmus beitragen, dann können wir Type-Squashing in seiner allgemeinen Form nicht verwenden sondern müssen die Struktur zum Teil freigeben und nur die Teile unterdrücken, die wirklich nicht benötigt werden und Störeffekte hervorrufen.

[65] Diese '*String-quotes*' ¨ sind zu unterscheiden von den sogenannten '*Token-quotes*' ', welche bei der Angabe von Namen in manchen Regeln benötigt werden.

kanonisch (Typen)	kanonisch (Elemente)	nichtkanonisch
Atom$\{\}()$	**token**$\{string:\mathtt{t}\}()$	**atom_eq**$(\boxed{u};\boxed{v};s;t)$
Atom	¨$string$¨	if $\boxed{u}$=$\boxed{v}$ then s else t

Zusätzliche Einträge in die Operatorentabelle

Redex		Kontraktum
if u=v then s else t	$\xrightarrow{\beta}$	s, *falls* $u = v$; *ansonsten* t

Zusätzliche Einträge in die Redex–Kontrakta Tabelle

Typsemantik
Atom = Atom
Elementsemantik
¨$string$¨ = ¨$string$¨ $\in$ Atom
Atom = Atom $\in$ U_j

Zusätzliche Einträge in den Semantiktabellen

$\Gamma \vdash$ Atom = Atom $\in U_j$ ⌊Ax⌋
  by atomEq

$\Gamma \vdash$ ¨$string$¨ $\in$ Atom ⌊Ax⌋
  by tokEq

$\Gamma \vdash$ Atom ⌊**ext** ¨$string$¨⌋
  by tokI ¨$string$¨

$\Gamma \vdash$ if u_1=v_1 then s_1 else t_1
  = if u_2=v_2 then s_2 else t_2 $\in T$ ⌊Ax⌋
  by atom_eqEq
  $\Gamma \vdash u_1$=u_2 $\in$ Atom ⌊Ax⌋
  $\Gamma \vdash v_1$=v_2 $\in$ Atom ⌊Ax⌋
  Γ, v: u_1=v_1 $\in$ Atom $\vdash s_1 = s_2 \in T$ ⌊Ax⌋
  Γ, v: $\neg$(u_1=v_1 $\in$ Atom) $\vdash t_1 = t_2 \in T$ ⌊Ax⌋

$\Gamma \vdash$ if u=v then s else t = t_2 $\in T$ ⌊Ax⌋
  by atom_eqRedT
  $\Gamma \vdash s = t_2 \in T$ ⌊Ax⌋
  $\Gamma \vdash u$=v $\in$ Atom ⌊Ax⌋

$\Gamma \vdash$ if u=v then s else t = t_2 $\in T$ ⌊Ax⌋
  by atom_eqRedF
  $\Gamma \vdash t = t_2 \in T$ ⌊Ax⌋
  $\Gamma \vdash \neg$(u=v $\in$ Atom) ⌊Ax⌋

Inferenzregeln

Abbildung 3.25: Syntax, Semantik und Inferenzregeln des Typs Atom und Listen kommen elementare arithmetische Operationen und primitive Rekursion hinzu. Über das Niveau der primitiv-rekursiven Funktionen kommt man mit den bisherigen Typkonzepten jedoch nicht hinaus.

Rein theoretisch ist dies zwar keine signifikante Einschränkung, da es so gut wie keine (terminierenden) Programme gibt, deren Effekt sich nicht auch mit den Mitteln der primitiv-rekursiven Funktionen beschreiben läßt. Jedoch haben primitiv-rekursive Funktionen gegenüber den Programmen, die in der Praxis eingesetzt werden, den Nachteil einer erheblich geringeren Eleganz und – wie am Ende von Beispiel 3.4.9 (Seite 154) bereits angedeutet – einer wesentlich schlechteren Komplexität. Dies liegt daran, daß primitiv-rekursive Funktionen – zu der Abarbeitung einer Zählschleife – ausschließlich eine Reduktion des Eingabearguments in Einzelschritten zulassen. Komplexere Schritte, eine vorzeitige Beendigung der Rekursion, eine Rekursion auf der Ergebnisvariablen oder gar eine Rekursion ohne eine vorher angegebene obere Schranke für die Anzahl der Rekursionsschritte – also die Charakteristika der üblicherweise eingesetzten Schleifen – lassen sich nicht unmittelbar ausdrücken. Zwar ist es möglich, die ersten drei Aspekte zu simulieren, aber diese Simulation bringt nur scheinbare Vorteile, da die ausdrucksstärkeren Programmierkonstrukte nur sehr ineffizient dargestellt werden können.[66]

Die primitive Rekursion trägt daher ihren Namen "primitiv" zu recht. Die allgemeine Rekursion, die in der Mathematik als auch in der Programmierung verwendet wird, ist an die obengenannten Einschränkungen nicht gebunden. Im Bezug auf Ausdruckskraft, Eleganz und Effizienz wird sie daher von keinem anderen mathematisch-programmiertechnischen Konstrukt übertroffen. Die Möglichkeit, Algorithmen oder mathematische Objekte durch rekursive Gleichungen zu definieren ist essentiell für den Aufbau mathematischer Theorien und die Entwicklung realistischer Programme.

Rekursion birgt jedoch auch Gefahren und Trugschlüsse in sich. Algorithmen können sich endlos rekursiv aufrufen, ohne jemals ein Ergebnis zu liefern. Mathematische Konstruktionen können nicht 'wohlfundiert' sein, weil die Rekursion in sich zwar schlüssig ist, es aber keinen Anfangspunkt gibt, auf dem man sie abstützen kann. Da es bekanntermaßen keine allgemeinen Verfahren gibt, derartige Gefahren auszuschließen,[67] ist ein das logische Schließen über Ausdrücke, die allgemeine Rekursionen enthalten, relativ schwer zu formalisieren, wenn wir zugunsten einer Rechnerunterstützbarkeit des Inferenzsystems garantieren wollen, daß jeder (typisierbare) Ausdruck einem Wert entspricht. Eine (aus theoretischer Sicht) vollständige Einbettung von Rekursion in formale Kalküle ist bisher nicht bekannt und wird vielleicht auch nie zu erreichen sein.

Im folgenden werden wir drei Erweiterungen der Typentheorie um rekursive Definitionen vorstellen, von denen zur Zeit allerdings nur die ersten beiden auch innerhalb des NuPRL Systems implementiert wurden. Die Unterschiede liegen in der Art der Objekte, die durch eine rekursive Gleichung definiert werden.

- Die *induktiven Typkonstruktoren*, die wir in Abschnitt 3.5.1 vorstellen, ermöglichen ein Schließen über rekursiv definierte Datentypen und deren Elemente. Hierbei muß die Rekursionsgleichung allerdings *wohlfundiert* sein, d.h. es muß möglich sein, aus der Rekursionsgleichung 'auszusteigen' und hierbei einen wohldefinierten Typ zu erhalten. Ein typisches Beispiel hierfür ist die Definition 2.2.4 von Formeln der Prädikatenlogik (Seite 24), welche besagt, daß Formeln entweder atomare Formeln sind, oder aus anderen Formeln durch Negation, Disjunktion, Konjunktion, Implikation oder Quantoren zu bilden sind.

- Die Behandlung *partiell rekursiver Funktionen* innerhalb einer Theorie, welche keine nichtterminierenden Reduktionen zuläßt, ist das Thema des Abschnitts 3.5.2. Die Schlüsselidee lautet hierbei, ausschließlich solche Funktionen zu betrachten, bei denen man auf der Grundlage ihrer rekursiven Definition einen Definitionsbereich bestimmen kann, auf dem sie garantiert terminieren. So wird eine Möglichkeit eröffnet, die volle Ausdruckskraft und Eleganz der üblichen Programmiersprachen innerhalb der Typentheorie wiederzuspiegeln, und dennoch die Probleme einzuschränken, die der allgemeine λ-Kalkül mit sich bringt.

[66] Da die primitive Rekursion nach wie vor die Reduktion einer Eingabe in einzelnen Schritten verlangt und keine der fest vordefinierten arithmetischen Operationen mehr als eine konstante Verringerung der Eingabe ermöglicht, können primitiv-rekursive Funktionen bestenfalls in linearer Zeit, gemessen an der Größe der Eingabe, berechnet werden. Logarithmische Zeit ist unerreichbar, obwohl viele praktische Probleme in logarithmischer Zeit lösbar sind.

[67] Die Unentscheidbarkeit des Halteproblems macht es unmöglich, ein allgemeines Verfahren anzugeben, welches nichtterminierende Algorithmen identifiziert bzw. unfundierte Rekursionen als solche entdeckt.

- Die *lässigen Typkonstruktoren*, die wir in Abschnitt 3.5.3 kurz andiskutieren werden, eröffnen eine Möglichkeit zum Schließen über unendliche Objekte wie zum Beispiel Prozesse, die ständig aktiv sind. Der Unterschied zu den induktiven Typkonstruktoren liegt darin, daß auch Gleichungen betrachtet werden können, die keine Ausstiegsmöglichkeit bieten. Ein typisches Beispiel hierfür ist die Definition eines Datenstroms (stream), die besagt, daß ein Datenstrom aus einem Zeichen, gefolgt von einem Datenstrom, besteht. Eine solche Definition kann nur durch unendliche Objekte interpretiert werden.

Induktive und lässige Datentypen sind eine Formalisierung der Grundideen rekursiver Datenstrukturen, die zum Beispiel in [Hoare, 1975] und [Gordon *et.al.*, 1979] ausführlich diskutiert werden. Wir werden im folgenden nur die Grundgedanken dieser drei Ansätze und das zugehörige Inferenzsystem vorstellen. Eine Vertiefung und weitere Details findet man in [Constable & Mendler, 1985, Mendler, 1987a, Constable & Smith, 1987, Constable & Smith, 1988, Mendler *et.al.*, 1986, Mendler, 1987b, Smith, 1988].

3.5.1 Induktive Typen

In den bisherigen Abschnitten haben wir bereits eine Reihe rekursiver Definitionen zur Beschreibung der formalen Sprache der Kalküle benutzt. So war zum Beispiel die Prädikatenlogik (Definition 2.2.4 auf Seite 24) rekursiv über atomare Formeln sowie Negation, Disjunktion, Konjunktion, Implikation und Quantoren definiert. λ-Terme (Definition 2.3.2 auf Seite 48) waren rekursiv über Variablen, Abstraktion und Applikation eingeführt worden. Die Terme der Typentheorie (Definition 3.2.5 auf Seite 104) wurden rekursiv über Operatoren und gebundene Terme erklärt. All diesen Definitionen ist gemeinsam, daß sie die zu erklärende Klasse von Objekten rekursiv durch eine Reihe von Bedingungen definieren. Dabei wird implizit festgelegt, daß die kleinstmögliche Klasse betrachtet werden soll, welche diese Bedingungen erfüllt. Dies bedeutet also, daß das die Klasse *induktiv* definiert wird: es wird eine Art Basisfall definiert und angegeben, wie aus bereits konstruierten Objekten ein neues Objekt aufgebaut werden darf. Für die Klasse der Formeln sind zum Beispiel die atomaren Formel der Basisfall. Für die Klasse der λ-Terme sind es die Variablen.

Die bisherigen rekursiven Definitionen waren informaler Natur. Will man nun typentheoretische Objekte durch rekursive Definitionen erklären, so bietet es sich an, hierzu rekursive Gleichungen zu verwenden und festzulegen, daß hierbei das kleinste Objekt definiert werden soll, welches diese Gleichungen erfüllt. Wir wollen dies an einem einfachen Beispiel illustrieren.

Beispiel 3.5.1 (Binärbäume über ganzen Zahlen)
Eines der Standardbeispiele für rekursiv definierte Datentypen sind Binärbäume über den ganzen Zahlen. Üblicherweise definiert man sie wie folgt.

Ein Binärbaum besteht entweder aus einer ganzen Zahl i oder einem Tripel (i,t_l,t_r), *wobei* t_l *und* t_r *Binärbäume sind und i eine ganze Zahl ist.*

Damit besteht der Datentyp `bintree` aller Binärbäume entweder aus dem Typ $\mathbb{Z}$ der ganzen Zahlen oder dem Produktraum $\mathbb{Z} \times \texttt{bintree} \times \texttt{bintree}$. Dies läßt sich formal durch die Gleichung

$$\texttt{bintree} = \mathbb{Z} + \mathbb{Z} \times \texttt{bintree} \times \texttt{bintree}$$

ausdrücken. Im Gegensatz zu dem oben angegegebenen Text läßt diese Gleichung aber noch mehrere Interpretationen zu, da sie ja nicht festlegt, daß wir die *kleinste* Menge `bintree` betrachten wollen, welche sie erfüllt. Auch unendliche Binärbäume erfüllen diese Gleichung, denn sie bestehen aus einer Integerwurzel und zwei unendlichen Binärbäumen. Deshalb muß bei der Einführung rekursiver Datentypen nicht nur die syntaktische Form sondern gleichzeitig die Semantik fixiert werden. Die naheliegendste Semantik ist die obenangegebene Interpretation als die kleinste Menge, welche eine Gleichung erfüllt. Dies bedeutet in unserem Fall, daß sich jedes Objekt in endlich vielen Schritten durch die in der Gleichung vorkommenden Fälle erzeugen läßt. In NuPRL bezeichnen wir den Term, der mit dieser Semantik identifiziert wird, als *induktiven* Datentyp und schreiben

$$\texttt{rectype bintree} = \mathbb{Z} + \mathbb{Z} \times \texttt{bintree} \times \texttt{bintree}$$

Terme, die eine andere Semantik rekursiver Gleichungen repräsentieren, werden wir in Abschnitt 3.5.3 kurz ansprechen.

Induktive Datentypen der Form $\underline{\mathsf{rectype}\ X = T_X}$ besitzen also eine Semantik, die – ähnlich wie bei der Definition rekursiver Funktionen im λ-Kalkül auf Seite 57 – durch den *kleinsten* Fixpunkt des Terms T_X auf der rechten Seite der Definitionsgleichung erklärt ist. Fast alle rekursiven Typgleichungen werden in dieser Weise verstanden und deshalb sind induktive Datentypen das wichtigste Konstrukt zur Einführung rekursiv definierter Konzepte.

Anders als bei den bisherigen Typkonstruktoren beschreiben induktive Datentypen zwar eine neue Klasse von Elementen, erzeugen hierfür aber *keine neuen kanonischen* Elemente. Stattdessen wird durch die rekursive Gleichung eine Art Rezept angegeben, wie Elemente des induktiven Typs zu konstruieren sind. So enthält zum Beispiel die oben angegebene Typdefinition

$$\mathsf{rectype}\ \mathtt{bintree} = \mathbb{Z}\ +\ \mathbb{Z} \times \mathtt{bintree} \times \mathtt{bintree}$$

die Vorschrift, daß die Elemente von `bintree` entweder Elemente von $\mathbb{Z}$ sein müssen, oder aus einem Tripel bestehen, dessen erste Komponente ein Element von $\mathbb{Z}$ ist und dessen andere beiden Komponenten jeweils Elemente von `bintree` sein müssen. Dabei kann im letzteren Fall nur auf bereits erklärte Elemente von `bintree` zurückgegriffen werden.

Um Elemente eines induktiven Datentyps weiterverwenden zu können, müssen wir eine *nichtkanonische Form* bereitstellen, welche besagt, wie der induktiven Aufbau eines solchen Elementes zu analysieren ist. Dies geschieht ähnlich wie im Falle der natürlichen Zahlen, die wir in Abschnitt 3.4.2.1 auf Seite 140 besprochen haben: ein Term $\underline{\mathsf{let}^*\, f(x) = t\ \mathsf{in}\ f(e)}$[68] beschreibt den Funktionswert einer Funktion f bei Eingabe eines Elementes e von $\underline{\mathsf{rectype}\ X = T_X}$, wobei für f die rekursive Funktionsgleichung $f(x)=t$ gelten soll. Diese Form läßt sich eigentlich auch unabhängig von induktiven Datentypen verwenden (siehe Abschnitt 3.5.2), führt im Zusammenhang mit diesen jedoch zu *immer terminierenden rekursiven Funktionen*, sofern sie mithilfe einer ensprechenden Eliminationsregel generiert wurde. In diesem Fall wird nämlich der induktive Aufbau des Elementes e schrittweise rekursiv abgebaut bis der “Basisfall” erreicht ist und eine feste Antwort berechnet werden kann. Wir wollen dies an einem Beispiel illustrieren

Beispiel 3.5.2

Wenn wir für einen Binärbaum b die Summe der Knoten berechnen wollen, dann reicht es, die in Beispiel 3.5.1 gegebene Struktur zu analysieren. Falls b ein Element von $\mathbb{Z}$ ist, dann ist b selbst die Summe. Andernfalls müssen wir rekursiv die Summe der beiden Teilbäume bestimmen und zum Wert der Wurzel hinzuaddieren. Die Fallanalyse ist durch den Vereinigungstyp bereits implizit vorgegeben. Wir erhalten also folgende formale Beschreibung eines Algorithmus $\mathtt{binsum}(t)$:

```
let* sum(b-tree)  =
  case b-tree  of inl(leaf) ↦ leaf
               | inr(triple) ↦ let ⟨num,pair⟩ = triple
                                in  let ⟨left,right⟩ = pair
                                    in  num+sum(left)+sum(right)
in sum(t)
```

Durch die nichtkanonische Form $\mathsf{let}^*\, f(x) = t\ \mathsf{in}\ f(e)$ sind wir nun in der Lage, zur Beschreibung von Funktionen eine unbeschränkte Rekursion verwenden zu dürfen, was die praktische Ausdruckskraft und *Effizienz* von NuPRL Programmen deutlich steigert. Falls keine freien Variablen in dieser Form vorkommen, kann sie als Redex betrachtet werden. Eine Reduktion entspricht in diesem Falle der Auswertung eines einzelnen Rekursionsschrittes: $\mathsf{let}^*\, f(x) = t\ \mathsf{in}\ f(e)$ reduziert zu $t[\lambda y.\mathsf{let}^*\, f(x) = t\ \mathsf{in}\ f(y),\ e\ /\ f, x]$. Um dies tun zu können, muß allerdings der induktive Aufbau des Elements e vorliegen, wodurch auch die Terminierung der Rekursion gesichert wird. Deshalb ist aus theoretischer Sicht keine Ausdruckskraft gegenüber der primitiven Rekursion gewonnen worden. Eine wirkliche Steigerung ist erst dann möglich, wenn man die Rekursion von der Analyse eines vorgegebenen induktiven Aufbaus entkoppelt und das Risiko nichtterminierender Algorithmen eingeht. Die hierbei entstehenden Probleme und einen Ansatz zu ihrer Lösung werden wir in Abschnitt 3.5.2 besprechen.

[68] Man beachte, daß durch $\mathsf{rectype}\ X = T_X$ und $\mathsf{let}^*\, f(x) = t\ \mathsf{in}\ f(e)$ nur ein Term erklärt wird, nicht aber der *Name* X oder f vergeben wird. X und f sind – wie die Details in Abbildung 3.26 zeigen – nichts anderes als bindende Variablen

kanonisch (Typen)	(Elemente)	nichtkanonisch
$\mathsf{rec}\{\}(X.T_X)$ $\mathsf{rectype}\ X = T_X$		$\mathsf{rec\_ind}\{\}(\boxed{e}; f, x.t)$ $\mathsf{let}^*\, f(x) = t\ \mathsf{in}\ f(\boxed{e})$

Zusätzliche Einträge in die Operatorentabelle

Redex		Kontraktum
$\mathsf{let}^*\, f(x) = t\ \mathsf{in}\ f(e)$	$\xrightarrow{\beta}$	$t[\lambda y.\mathsf{let}^*\, f(x) = t\ \mathsf{in}\ f(y),\ e\ /\ f, x]$

Zusätzliche Einträge in die Redex–Kontrakta Tabelle

Typsemantik		
$\mathsf{rectype}\ X_1 = T_{X1}\ =\ \mathsf{rectype}\ X_2 = T_{X2}$	falls	$T_{X1}[X/X_1]\ =\ T_{X2}[X/X_2]$ für alle Typen X
Elementsemantik		
$s\ =\ t\ \in\ \mathsf{rectype}\ X = T_X$	falls	$\mathsf{rectype}\ X = T_X$ Typ und $s\ =\ t\ \in\ T_X[\mathsf{rectype}\ X = T_X\ /\ X]$
$\mathsf{rectype}\ X_1 = T_{X1}\ =\ \mathsf{rectype}\ X_2 = T_{X2}\ \in\ \mathsf{U}_j$	falls	$T_{X1}[X/X_1]\ =\ T_{X2}[X/X_2]\ \in\ \mathsf{U}_j$ für alle Terme X mit $X\ \in\ \mathsf{U}_j$

Zusätzliche Einträge in den Semantiktabellen

$$\begin{array}{l}
\Gamma\ \vdash\ \mathsf{rectype}\ X_1 = T_{X1}\ =\ \mathsf{rectype}\ X_2 = T_{X2}\ \in\ \mathsf{U}_j\quad \lfloor\mathsf{Ax}\rfloor\\
\quad \textbf{by}\ \underline{\mathtt{recEq}}\\
\quad \Gamma,\ X:\mathsf{U}_j\ \vdash\ T_{X1}[X/X_1]\ =\ T_{X2}[X/X_2]\ \in\ \mathsf{U}_j\quad \lfloor\mathsf{Ax}\rfloor
\end{array}$$

$$\begin{array}{l}
\Gamma\ \vdash\ s\ =\ t\ \in\ \mathsf{rectype}\ X = T_X\quad \lfloor\mathsf{Ax}\rfloor\\
\quad \textbf{by}\ \underline{\mathtt{rec_memEq}\ j}\\
\quad \Gamma\ \vdash\ s\ =\ t\ \in\ T_X[\mathsf{rectype}\ X = T_X\ /\ X]\quad \lfloor\mathsf{Ax}\rfloor\\
\quad \Gamma\ \vdash\ \mathsf{rectype}\ X = T_X\ \in\ \mathsf{U}_j\quad \lfloor\mathsf{Ax}\rfloor
\end{array}$$

$$\begin{array}{l}
\Gamma\ \vdash\ \mathsf{rectype}\ X = T_X\quad \lfloor\textbf{ext}\ t\rfloor\\
\quad \textbf{by}\ \underline{\mathtt{rec_memI}\ j}\\
\quad \Gamma\ \vdash\ T_X[\mathsf{rectype}\ X = T_X\ /\ X]\quad \lfloor\textbf{ext}\ t\rfloor\\
\quad \Gamma\ \vdash\ \mathsf{rectype}\ X = T_X\ \in\ \mathsf{U}_j\quad \lfloor\mathsf{Ax}\rfloor
\end{array}$$

$$\begin{array}{l}
\Gamma\ \vdash\ \mathsf{let}^*\, f_1(x_1) = t_1\ \mathsf{in}\ f_1(e_1)\\
\qquad =\ \mathsf{let}^*\, f_2(x_2) = t_2\ \mathsf{in}\ f_2(e_2)\ \in\ T[e_1/z]\quad \lfloor\mathsf{Ax}\rfloor\\
\quad \textbf{by}\ \underline{\mathtt{rec_indEq}\ z\ T\ \mathsf{rectype}\ X = T_X\ j}\\
\quad \Gamma\ \vdash\ e_1\ =\ e_2\ \in\ \mathsf{rectype}\ X = T_X\quad \lfloor\mathsf{Ax}\rfloor\\
\quad \Gamma\ \vdash\ \mathsf{rectype}\ X = T_X\ \in\ \mathsf{U}_j\quad \lfloor\mathsf{Ax}\rfloor\\
\quad \Gamma,\ P:(\mathsf{rectype}\ X = T_X)\to\mathbb{P}_j,\\
\qquad f:(y:\{x:\mathsf{rectype}\ X = T_X \mid P(x)\}\to T[y/z]),\\
\qquad x:T_X[\{x:\mathsf{rectype}\ X = T_X \mid P(x)\}\ /\ X]\\
\qquad \vdash\ t_1[f,x\ /\ f_1,x_1] = t_2[f,x\ /\ f_2,x_2] \in T[x/z]\quad \lfloor\mathsf{Ax}\rfloor
\end{array}$$

$$\begin{array}{l}
\Gamma,\ z:\mathsf{rectype}\ X = T_X,\ \Delta\ \vdash\ C\\
\qquad \lfloor\textbf{ext}\ \mathsf{let}^*\, f(x) = t[\lambda y.\Lambda\ /\ P]\ \mathsf{in}\ f(z)\rfloor\\
\quad \textbf{by}\ \underline{\mathtt{recE}\ i\ j}\\
\quad \Gamma,\ z:\mathsf{rectype}\ X = T_X,\ \Delta\\
\qquad \vdash\ \mathsf{rectype}\ X = T_X\ \in\ \mathsf{U}_j\quad \lfloor\mathsf{Ax}\rfloor\\
\quad \Gamma,\ z:\mathsf{rectype}\ X = T_X,\ \Delta\\
\qquad P:(\mathsf{rectype}\ X = T_X)\to\mathbb{P}_j,\\
\qquad f:(y:\{x:\mathsf{rectype}\ X = T_X \mid P(x)\}\to C[y/z]),\\
\qquad x:T_X[\{x:\mathsf{rectype}\ X = T_X \mid P(x)\}\ /\ X]\\
\qquad\quad \vdash\ C[x/z]\quad \lfloor\textbf{ext}\ t\rfloor
\end{array}$$

$$\begin{array}{l}
\Gamma,\ z:\mathsf{rectype}\ X = T_X,\ \Delta\ \vdash\ C\quad \lfloor\textbf{ext}\ t[z/x]\rfloor\\
\quad \textbf{by}\ \underline{\mathtt{recE_unroll}\ i}\\
\quad \Gamma,\ z:\mathsf{rectype}\ X = T_X,\ \Delta,\\
\qquad x:T_X[\mathsf{rectype}\ X = T_X\ /\ X],\\
\qquad v:\ z = x \in T_X[\mathsf{rectype}\ X = T_X\ /\ X]\\
\qquad\quad \vdash\ C[x/z]\quad \lfloor\textbf{ext}\ t\rfloor
\end{array}$$

Inferenzregeln

Abbildung 3.26: Syntax, Semantik und Inferenzregeln induktiver Typen

Induktive Datentypen können im Prinzip simuliert werden, da wir einen induktiven Datentyp – die natürlichen Zahlen – bereits kennen. So könnte zum Beispiel eine Simulation des Typs der Binärbäume aus Beispiel 3.5.1 zunächst induktiv Binärbäume der maximalen Tiefe i definieren durch

$$\texttt{Bintree}(i) \;\equiv\; \mathsf{ind}(i;\ \_,\_.\_;\ \mathbb{Z};\ \texttt{j},\texttt{bin}_j.\ \texttt{bin}_j\ +\ \mathbb{Z}\times\texttt{bin}_j\times\texttt{bin}_j)$$

und dann festlegen

$$\texttt{Bintree} \;\equiv\; \texttt{i}:\mathbb{N}\ \times\ \texttt{Bintree(i)}.$$

Eine derartige Simulation wäre allerdings verhältnismäßig umständlich und würde die natürliche Form der rekursiven Definition völlig entstellen. Nichtsdestotrotz zeigt die Simulation, daß die Hinzunahme der induktiven Datentypen keine Erweiterung der Ausdruckskraft sondern nur eine naturgetreuere Formalisierung für eine bestimmte Klasse von Datentypen liefert.[69]

Abbildung 3.26 faßt die Syntax, Semantik und Inferenzregeln induktiver Datentypen zusammen. Man beachte, daß die Semantik des nichtkanonischen Terms durch "Aufrollen" (Englisch "*unroll*") der Rekursion erklärt wird, wobei jedoch nur ein einziger Rekursionsschritt durchgeführt wird. Ist der entstehende Term nicht in kanonischer Form, so muß weiter reduziert werden, um die Bedeutung des Ausdrucks zu erhalten. Hierdurch wird sichergestellt, daß im Kalkül selbst keine nichtterminierenden Bestandteile auftauchen obwohl der implizit in $\mathsf{let}^*\ f(x) = t\ \ \mathsf{in}\ \ f(e)$ enthaltene Algorithmus (Auflösen der Rekursion bis zu einem Terminierungspunkt) durchaus nicht zu einem Ende kommen muß.

Die meisten Inferenzregeln stützen sich ebenfalls auf das Aufrollen einer Rekursion ab. Da es keine kanonischen Elemente gibt, muß zum Nachweis der Typzugehörigkeit die rekursive Typdefinition schrittweise aufgefaltet und analysiert werden. Ebenso kann man induktive Typen durch einfaches Aufrollen eliminieren (`recE_unroll`). Im Normalfall (`recE`) ist die Elimination induktiver Typen allerdings komplexer, da hierdurch ein nichtkanonischer Term der Form $\mathsf{let}^*\ f(x) = t\ \ \mathsf{in}\ \ f(e)$ erzeugt werden soll. Um die oben beschriebene Bedeutung (Funktionswert einer rekursiv durch t definierte Funktion f mit Argument x bei Eingabe eines Elementes z) zu erzielen, müssen wir f als eine Funktion auf einer beliebigen Teilmenge von $\mathsf{rectype}\ X = T_X$ und x als ein Element von T_X – wobei für X diese Teilmenge eingesetzt wird – voraussetzen und zeigen, wie wir hieraus den Term t konstruieren. Hierdurch wird sichergestellt, daß der entstehende rekursive Algorithmus wohldefiniert ist und terminiert, wenn man mit der leeren Menge als Ausgangspunkt anfängt.[70]

Beispiel 3.5.3

Als Anwendungsbeispiel wollen wir die Konstruktion eines Algorithmus skizzieren, welcher bei Eingabe eines Binärbaumes und einer ganzen Zahl entscheidet, ob diese Zahl im Baum erscheint oder nicht. Da dieser Algorithmus genauso grundlegend ist, wie die Eigenschaft, die ihn spezifiziert, geben wir als Beweisziel nur eine Typisierung des Algorithmus an und konstruieren diesen dann implizit durch unsere Entscheidungen im Laufe des Beweises. Aus Gründen der Übersichtlichkeit werden wir uns auf eine Skizze der Kernbestandteile des Beweises beschränken. Als definitorische Abkürzung verwenden wir

$$\texttt{Bintree} \;\equiv\; \mathsf{rectype}\ \texttt{bintree} = \mathbb{Z}\ +\ \mathbb{Z}\times\texttt{bintree}\times\texttt{bintree}$$

[69] Umgekehrt macht die Existenz der induktiven Datentypen natürlich auch die explizite Definition konkreter induktiver Typen wie natürlicher Zahlen bzw. Listen hinfällig. Man könnte nämlich simulieren:

$$\mathbb{N} \;\equiv\; \mathsf{rectype}\ \texttt{N} = \texttt{Unit}\ +\ \texttt{N}$$

wobei `Unit` ein vorgegebener einelementiger Datentyp ist und eine Zahl n durch n Rekursionen dargestellt wird. Ebenso lassen sich Listen über dem Typ T beschreiben durch

$$T\ \mathsf{list} \;\equiv\; \mathsf{rectype}\ \texttt{T\_list} = T + \texttt{T\_list}.$$

Da jedoch auch diese Simulation unnatürlich wäre, ist eine explizite Formalisierung dieser Datentypen in jedem Fall vorzuziehen.

[70] Diese Vorgehensweise spiegelt die '*Fixpunktinduktion*' wieder. Der induktive Typ $\mathsf{rectype}\ X = T_X$ kann semantisch als Grenzwert der Folge $\emptyset,\ T_X(\emptyset),\ T_X(T_X(\emptyset)),\ \ldots$ angesehen werden und jedes Element z von $\mathsf{rectype}\ X = T_X$ gehört zu einer dieser Stufen. Eine rekursive Analyse von z wird also schrittweise diese Stufen abbauen und bei $\emptyset$ terminieren. Da die Stufe von z aber nicht bekannt ist, muß der Analyseterm t auf beliebigen Teilmengen von $\mathsf{rectype}\ X = T_X$ operieren können und sich im Endeffekt auf $\emptyset$ abstützen.

Die genauen Regeln sind das Ergebnis mühevoller Feinarbeit, bei der es darum ging, mögliche Trugschlüsse zu vermeiden. Sie sind daher normalerweise nicht sofort intuitiv klar sondern müssen gründlich durchdacht werden.


```
⊢ Bintree → ℤ → 𝔹
by lambdaI 1                          ⌊ext λB.λz.let* f(x) = case x of inl(i) ↦ ... | inr(tree) ↦ ...  in f(B)⌋
|\
| B:Bintree ⊢ ℤ → 𝔹
| by lambdaI 1                          ⌊ext λz.let* f(x) = case x of inl(i) ↦ ... | inr(tree) ↦ ...  in f(B)⌋
| |\
| | B:Bintree, z:ℤ ⊢ 𝔹
| | by recE 1 1                           ⌊ext let* f(x) = case x of inl(i) ↦ ... | inr(tree) ↦ ...  in f(B)⌋
| | |\
| | | B:Bintree, z:ℤ ⊢ Bintree ∈ U1
| | |   siehe unten
| |  \
| |   B:Bintree, z:ℤ, P:Bintree→ℙ1, f:{x:Bintree | P(x)} → 𝔹,
| |     x:ℤ + ℤ × {x:Bintree | P(x)} × {x:Bintree | P(x)}
| |   ⊢ 𝔹
| |   by unionE 5                     ⌊ext case x of inl(i) ↦ if z=i then T else F | inr(tree) ↦ let .... ⌋
| |    |\
| |    | .... i:ℤ ⊢ 𝔹
| |    | by intro if z=i then T else F                                  ⌊ext if z=i then T else F⌋
| |    |     ⋮
| |     \
| |      .... tree:ℤ × {x:Bintree | P(x)} × {x:Bintree | P(x)} ⊢ 𝔹
| |      by productE 6 THEN productE 8          ⌊ext let ⟨i,B1,B2⟩ = tree in if z=i then T else ...⌋
| |       \
| |        .... i:ℤ, pair:{x:Bintree | P(x)} × {x:Bintree | P(x)},
| |          B1:{x:Bintree | P(x)}, B2:{x:Bintree | P(x)}
| |        ⊢ 𝔹
| |        by intro if z=i then T else if f(B1) then T else f(B2)       ⌊ext if z=i then T else ...⌋
| |             ⋮
|  \
|   ⊢ ℤ ∈ U1
|   by intEq
 \
  ⊢ Bintree ∈ U1
  by recEq
   \
    bintree:U1 ⊢ ℤ + ℤ × bintree × bintree ∈ U1
    by ..... unionEq, productEq, intEq, hypEq
```

Aus dem Beweis können wir den folgenden rekursiven Algorithmus extrahieren.

```
λB.λz. let* f(x) =
          case x  of inl(i) ↦ if z=i then T else F
                 | inr(tree) ↦ let ⟨i,B1,B2⟩ = tree in if z=i then T else if f(B1) then T else f(B2)
        in f(B)
```

Dieser Algorithmus analysiert den rekursiven Aufbau eines Binärbaumes, und vergleicht die gesuchte Zahl mit der Wurzel des Baumes (bzw. dem ganzen Baum im Basisfall). Sind diese Zahlen identisch, so ist die Suche beendet. Ansonsten wird zunächst der linke und danach der rechte Teilbaum durchsucht.

Induktive Datentypen sind ein mächtiges Hilfsmittel zum Schließen über wohlfundierte, rekursive definierte Konstrukte. Sie sind allerdings auch mit Vorsicht einzusetzen, da – im Gegensatz zu den bisher eingeführten Typkonstrukten – nicht jeder formulierbare induktive Datentyp auch sinnvoll ist.

Beispiel 3.5.4

Wir betrachten den Term $T \equiv \mathsf{rectype}\ \texttt{X} = \texttt{X}\to\mathbb{Z}$. Die hierin enthaltene Rekursionsgleichung legt fest, daß die Elemente von T Funktionen von T in die Menge der ganzen Zahlen sein müssen. Daß dies zu widersprüchlichen Situationen führt, zeigt das folgende Argument.

Es sei $t \in T$. Dann ist t auch ein Element von $T\to\mathbb{Z}$ und somit ist $t\,t$ ein wohldefiniertes Element von $\mathbb{Z}$. Abstrahieren wir nun über t, so erhalten wir $\lambda t.\,t\,t\ \in\ T\to\mathbb{Z}$ bzw. $\lambda t.\,t\,t\ \in\ T$.

Wenn wir also $\mathsf{rectype}\ \texttt{X} = \texttt{X}\to\mathbb{Z}$ als korrekten induktiven Datentyp innerhalb der Typentheorie zulassen würden, dann würde ein wohlbekannter nichtterminierender Term typisierbar werden und dies – im

Gegensatz zu unserem Beispiel aus Abbildung 3.15 (Seite 134) – sogar ohne jede Voraussetzung. Wir wären sogar in der Lage, diesen Term aus dem folgenden Beweis zu extrahieren.

```
⊢ T      ⌊ext λt. t t⌋
by rec_memI 1
|\
| ⊢ T→ℤ      ⌊ext λt. t t⌋
|  by lambdaI 1
|   |\
|   | t:T ⊢ ℤ    ⌊ext t t⌋
|   | by recE_unroll 1
|   |  \
|   |   t:T, y:T→ℤ, v: t=y∈T→ℤ ⊢ ℤ    ⌊ext y y⌋
|   |   by functionE 2 y
|   |    |\
|   |    | t:T, y:T→ℤ, v: t=y∈T→ℤ ⊢ y∈T
|   |    | Substitution etc.
|   |     \
|   |       t:T, y:T→ℤ, v: t=y∈T→ℤ, z:ℤ, v': z = y y ∈ ℤ ⊢ ℤ    ⌊ext z⌋
|   |       by hyp 4
|    \
|     ⊢ T ∈ U1
|     siehe unten
 \
  ⊢ T ∈ U1
  by recEq
   \
    X:U1 ⊢ X→ℤ ∈ U1
    by functionEq
     |\
     | X:U1 ⊢ X ∈ U1
     | by hypEq 1
      \
       X:U1 ⊢ ℤ ∈ U1
       by intEq
```

Dieses Beispiel zeigt, daß wir Terme der Form rectype X = X→T nicht zulassen dürfen, da wir auf die schwache Normalisierbarkeit typisierbarer Terme nicht verzichten wollen. Diese Einschränkung ist durchaus sehr natürlich, da auch intuitiv ein Typ nicht sein eigener Funktionenraum sein darf. Was aber genau ist die Ursache des Problems?

Im Beispiel haben wir gesehen, daß die Kombination der Einführungsregel für λ-Terme und der unroll-Regel für induktive Datentypen die Einführung des Terms $\lambda t.\, t\, t$ ermöglicht. Dies ist aber nur dann der Fall, wenn der induktive Datentyp T ein Funktionenraum ist, der sich selbst im Definitionsbereich enthält. Ein Datentyp der Form rectype X = T→X ist dagegen durchaus unproblematisch.

Wie können wir nun *syntaktische* Einschränkungen an induktive Datentypen geben, welche die problematischen Fälle schon im Vorfeld ausschließen, ohne dabei übermäßig restriktiv zu sein? Eine genaue syntaktische Charakterisierung der zulässigen Möglichkeiten konnte bisher noch nicht gefunden werden. Die bisher beste Restriktion ist die Forderung, daß in einem induktiven Datentyp rectype $X = T_X$ die Typvariable X in T_X nur *positiv* vorkommen darf, was in etwa[71] dasselbe ist wie die Bedingung, daß X nicht auf der linken Seite eines Funktionenraumkonstruktors in T_X erscheinen darf. Diese Bedingung, die sich syntaktisch relativ leicht überprüfen läßt, stellt sicher, daß nur induktive Datentypen mit einer wohldefinierten Semantik in formalen Definitionen und Beweisen verwendet werden können.

Der bisher vorgestellte induktive Datentyp reicht aus, um die meisten in der Praxis vorkommenden rekursiven Konzepte zu beschreiben. In manchen Fällen ist es jedoch sinnvoll, die rekursive Definition zu *parametrisieren*.

[71] Eine ausführliche Definition dieses Begriffs findet man in [Constable & Mendler, 1985, Mendler, 1987a].

Beispiel 3.5.5

Um den Datentyp der logischen Propositionen erster Stufe innerhalb der Typentheorie präzise zu beschreiben, müßte man eine entsprechende Einschränkung des Universums U_1 charakterisieren, die nur solche Terme zuläßt, deren äußere Gestalt einer logischen Formel im Sinne von Definition 2.2.4 entspricht:

$$\mathbb{P} \equiv \{\texttt{F}:U_1 \mid \texttt{is\_formula(F)}\}$$

Dabei soll is_formula(F) das Prädikat (den Datentyp) repräsentieren, welches beschreibt, daß F eine Formel ist. Dieses Prädikat hat – entsprechend der Definition 2.2.4 – eine rekursive Charakterisierung:

```
is_formula(F)
  ⇔   atomic(F)
     ∨  F = Λ ∈ U1
     ∨  ∃A:U1.∃B:U1. is_formula(A) ∧ is_formula(A)
          ∧  F = ¬A ∈ U1
          ∨  F = A∧B ∈ U1
          ∨  F = A∨B ∈ U1
          ∨  F = A⇒B ∈ U1
     ∨  ∃T:U1.∃A:T→U1. ∀x:T.is_formula( A(x) )
          ∧  F = ∀x:T.A(x) ∈ U1
          ∨  F = ∃x:T.A(x) ∈ U1
```

Will man dies nun in einen induktiven Datentyp umsetzen, so stellt man fest, daß die Rekursion nicht nur von is_formula abhängt, sondern auch das Argument des Prädikats sich ständig ändert. Mit dem bisherigen – einfachen – Konzept der induktiven Datentypen läßt sich dies nicht mehr auf natürliche Art ausdrücken.

Eine Simulation parametrisierter induktiver Datentypen mithilfe der einfachen Version und des Produkttyps ist prinzipiell möglich, führt aber zu größeren Komplikationen. Es gibt daher Überlegungen, parametrisierte induktive Datentypen als Grundform in die Typentheorie mit aufzunehmen.[72]

Neben der Parametrisierung gibt es noch eine weitere Erweiterungsmöglichkeit für induktive Datentypen. So hatten wir in der Definition der Terme der Typentheorie (siehe Seite 104) das Konzept der Terme auf gebundene Terme und dieses wieder auf das der Terme abgestützt. Eine solche simultane (Englisch *mutual*=gegenseitig) rekursive Definition zweier Konzepte taucht auch in modernen Programmiersprachen relativ häufig auf: zwei Prozeduren können sich wechselseitig immer wieder aufrufen, bis ein Ergebnis geliefert wird. Zwar gibt es auch hierfür eine Simulation (siehe z.B. [Mendler, 1987a, Seite 17]), aber die Natürlichkeit der simultanen Rekursion macht es sinnvoll, diese ebenfalls explizit in die Typentheorie zu integrieren.

Die allgemeinste Form des induktiven Datentyps würde also n durch simultane Induktion definierte Datentypen X_i enthalten, die durch Paramter x_i parametrisiert sind. Zusätzlich müßte angegeben werden, welcher dieser Datentypen X_i nun gefragt ist und mit welchem Wert a_i der Parameter x_i initialisiert werden soll. Als Term wird hierfür vorgeschlagen:

$$\textbf{mrec}\{\}(X_1, x_1.T_{X_1}; \ldots; X_n, x_n.T_{X_n};\ X_i; a_i)$$ [73]

Eine vollständige Formalisierung derartiger induktiver Datentypen sowie ihre Semantik und die zugehörigen Regeln wird in [Constable & Mendler, 1985, Mendler, 1987a] ausführlich behandelt.

[72] In früheren Versionen des NuPRL Systems waren diese parametrisierten Versionen auch enthalten. Da sie aber erheblich aufwendiger sind und so gut wie keine praktische Anwendung fanden, wurden sie zugunsten der einfacheren Handhabung durch die hier vorgestellten "einfachen rekursiven Datentypen" ersetzt.

[73] Da dieser Typ bisher kaum Anwendung fand, gibt es eine bisher keine sinnvolle Displayform. In Anlehnung an die Programmiersprache ML könnte man vielleicht schreiben: rectype $X_1(x_1) = T_{X_1}$ and ... and $X_n(x_n) = T_{X_n}$ select $X_i(a_i)$

3.5.2 (Partiell) Rekursive Funktionen

Im vorhergehenden Abschnitt haben wir gesehen, daß die nichtkanonischen Terme, die wir im Zusammenhang mit induktiven Datentypen eingeführt haben, eine Möglichkeit bieten, allgemeine Rekursion in die Typentheorie mit aufzunehmen. Der Term $\mathsf{let}^*\ f(x) = t\ \ \mathtt{in}\ \ f(e)$ beschreibt eine rekursive Funktion in Abhängigkeit von einem Element eines induktiven Datentyps. Hierdurch gewinnen wir die Eleganz und Effizienz der allgemeinen Rekursion und behalten dennoch die Eigenschaft, daß alle typisierbaren Algorithmen terminieren. Dennoch bleibt etwas Unnatürliches in diesem Ansatz, da zugunsten der Wohlfundiertheit die in $\mathsf{let}^*\ f(x) = t\ \ \mathtt{in}\ \ f(e)$ definierte Funktion f an einen vorgegebenen rekursiven Datentyp gebunden ist, welcher ihren Definitionsbereich beschreibt. In vielen Fällen wird jedoch eine rekursive Funktion *nicht* innerhalb eines Beweises als Extraktterm eines induktiven Datentyps konstruiert. Stattdessen ist oft nur der Algorithmus gegeben, ohne daß der konkrete Definitionsbereich bekannt ist. Wir wollen hierfür einige Beispiele geben.

Beispiel 3.5.6

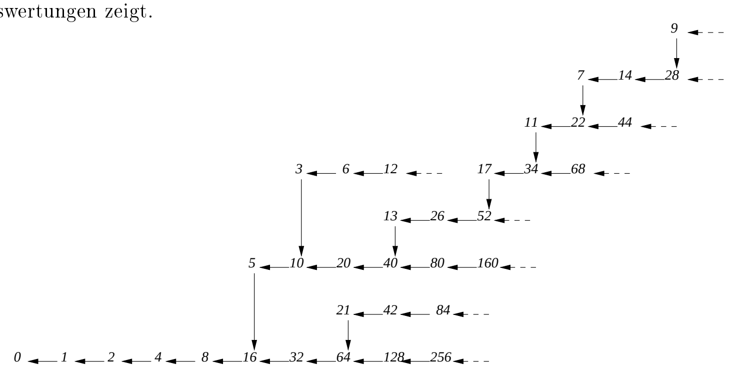
1. In der Mathematik ist die sogenannte `3x+1`-Funktion ein beliebtes Beispiel für eine leicht zu definierende rekursive Funktion, deren Definitionsbereich man nicht auf natürliche Art beschreiben kann. Sie ist definiert durch

$$f(x) = \begin{cases} 0 & \text{falls } x = 1, \\ f(x/2) & \text{falls } x \text{ gerade ist,} \\ f(3x+1) & \text{sonst} \end{cases}$$

   Löst man die Koppelung der rekursiven Funktionsdefinition von der Vorgabe eines induktiven Definitionsbereichs, so läßt sich diese Funktion formalisieren durch

   $\lambda\mathtt{x}.\ \mathsf{let}^*\ \mathtt{f(y)} = \mathsf{if}\ \mathtt{y=1\ then\ 0\ else\ if\ y\ rem\ 2\ =\ 0\ then\ f(x\div 2)\ else\ f(3*x+1)\ \ \ in\ f(x)}$

   Ihr Definitionsbereich ist allerdings relativ kompliziert strukturiert, wie die folgende Skizze des Verlaufes einiger Auswertungen zeigt.



   Zwar ist es prinzipiell möglich, diesen Definitionsbereich mithilfe eines parametrisierten induktiven Datentyps zu beschreiben, welcher genau der Funktionsdefinition entspricht, aber dies ist nicht nur relativ kompliziert, sondern auch unnatürlich. Hierdurch wir nämlich nicht ausgedrückt, daß es sich um eine – möglicherweise partielle – Funktion auf natürlichen Zahlen handelt, sondern der Bereich der natürlichen Zahlen muß entsprechend obiger Abbildung in seltsamer Weise neu angeordnet werden.

2. Unbeschränkte Suche nach der Nullstelle einer beliebigen Funktion auf den natürlichen Zahlen (vergleiche Beispiel 3.4.7 auf Seite 153) läßt sich ebenfalls leicht als rekursive Funktion beschreiben.

   $\lambda f.\ \mathsf{let}^*\ \mathtt{min}_f\mathtt{(y)} = \mathsf{if}\ f\mathtt{(y)=0\ then\ y\ else\ min}_f\mathtt{\ (y+1)\ \ in\ \ min}_f\mathtt{(0)}$

   Diese Funktion ist wohldefiniert und terminiert offensichtlich auch auf allen Elementen der Funktionenmenge $\{\mathtt{f}:\mathbb{N}\to\mathbb{Z}\mid \exists\mathtt{y}:\mathbb{N}.\ \mathtt{f\ (y)\ =0}\}$. Nichtsdestotrotz haben wir keinerlei Informationen über die rekursive Struktur dieses Definitionsbereiches.

3. In Beispiel 3.4.9 auf Seite 154 haben wir einen linearen Algorithmus zur Berechnung der Integerquadratwurzel einer natürlichen Zahl `x` hergeleitet und darauf hingewiesen, daß dies der effizienteste Algorithmus ist, den man mit dem Mittel der Induktion bzw. der primitiven Rekursion erzeugen kann. Natürlich aber gibt es erheblich effizientere Algorithmen, wenn man anstelle einer Rekursion auf der Eingabe `x` eine Rekursion auf dem Ergebnis verwenden kann. In diesem Falle ist es nämlich möglich, nach dem kleinsten Wert `y` zu suchen, für den $\mathtt{(y+1)}^2\mathtt{>x}$ gilt. Diese Vorgehensweise führt zum Beispiel zu dem folgenden rekursiven Algorithmus

   $\lambda\mathtt{x}.\ \mathsf{let}^*\ \mathtt{sq\text{-}search(y)} = \mathsf{if}\ \mathtt{x<(y+1)}^2\ \mathtt{then\ y\ else\ sq\text{-}search\ (y+1)\ \ in\ \ sq\text{-}search(0)}$

   Auch dieser Algorithmus ist sehr leicht zu verstehen und terminiert auf allen natürlichen Zahlen. Eine Koppelung an eine rekursive Datenstruktur müßte die natürlichen Zahlen jedoch in einer ganz anderen, dem Problem angepaßten Weise strukturieren.

Diese Beispiele zeigen, daß eine Koppelung der allgemeinen Rekursion an einen induktiven Datentyp eine massive Einschränkung der praktischen Anwendbarkeit rekursiver Funktionsdefinitionen mit sich bringt. Man muß nämlich immer erst den Definitionsbereich so umstrukturieren, daß der vorgesehene Algorithmus genau auf dieser Struktur arbeiten kann. Erst danach kann man dann den Algorithmus entwickeln.

In der Praxis geht man jedoch andersherum vor. Im Vordergrund steht der Algorithmus, der auf einem fest vorgegebenen Definitionsbereich oder einer Teilmenge davon operieren soll. Die induktive Struktur dieses Definitionsbereiches ergibt sich dann implizit aus der Abarbeitungsstruktur des Algorithmus, wird aber nicht als eigentlicher Definitionsbereich angesehen. So ist zum Beispiel der Definitionsbereich der Integerquadratwurzelfunktion die Menge der natürlichen Zahlen und nicht etwa ein komplizierter induktiver Typ, in dem natürliche Zahlen blockweise strukturiert sind. Ähnliches gilt für das Nullstellensuchprogramm, dessen Definitionsbereich eine Teilmenge der ganzzahligen Funktionen ist.

Es ist also wünschenswert, die Typentheorie um ein Konzept zu ergänzen, welches erlaubt, *alle* rekursiven Algorithmen zu betrachten, ohne daß dafür im Voraus die genaue Struktur des Definitionsbereiches bekannt sein muß. Dies würde erlauben, innerhalb der Typentheorie sehr effiziente Algorithmen zu programmieren und zu analysieren und weit über die Ausdruckskraft der primitiv-rekursiven Funktionen hinauszugehen. Erst die Entkoppelung der rekursiven Funktionen von den induktiven Datentypen macht es möglich, formal über "reale" Programme zu argumentieren. Damit wäre die Typentheorie nicht nur theoretisch dazu geeignet, das Schließen über Programmierung zu formalisieren, sondern auch ein praktisch adäquater Formalismus in dem Sinne, daß sie tatsächlich auch als reale Programmiersprache eingesetzt werden kann. Entsprechend den Erkenntnisssen der Theorie der Berechenbarkeit müßten wir hierfür allerdings auch Algorithmen akzeptieren, die von ihrer Natur her auch partiell sein könnten

Wie können wir nun allgemeine Rekursion in die Typentheorie mit aufnehmen, wenn doch gerade die unbeschränkte Rekursion die Ursache aller Probleme ist, die wir in Kapitel 2.3.7 feststellen mußten? Wie können wir also die Ausdruckskraft der Typentheorie auf elegante Art um ein Konzept partiell rekursiver Funktionen erweitern, ohne dabei das Inferenzsystem mit nichtterminierenden Bestandteilen zu belasten?

Die naheliegendste Idee ist, partiell rekursive Funktionen von einem Typ S in einen Typ T durch totale Funktionen zu repräsentieren, die auf einer Teilmenge von S operieren. Man könnte also den Datentyp $\underline{S\not\to T}$ der partiellen Funktionen simulieren durch

$$S\not\to T\ \equiv\ \mathtt{DOM}{:}S\to\mathbb{P}_1\ \times\ \{\mathtt{x}{:}S\mid \mathtt{DOM(x)}\}\to T.$$

Diese Darstellung ist sehr einfach und leicht zu handhaben, hat aber den Nachteil, daß man zu jedem Programm eine Beschreibung des Definitionsbereiches gleich mitliefern muß. $S\not\to T$ wäre also nicht ein Datentyp von partiellen Funktionen im eigentlichen Sinne.

Ein wesentlich natürlicherer und ebenso einfacher Ansatz, partielle Funktionen in die Typentheorie zu integrieren ist es, Rekursion zunächst einmal als ein *unabhängiges* Berechnungskonzept zu betrachten und die Definitionsbereiche rekursiver Funktionen aus ihrem Algorithmus *herzuleiten*. Dabei ist natürlich klar, daß es nicht immer möglich ist, den genauen Bereich zu bestimmen, auf dem eine Funktion terminiert. Der hergeleitete Definitionsbereich muß sich daher daran orientieren, was über den gegebenen Algorithmus mit Sicherheit bewiesen werden kann.[74] Dieser Weg macht das Schließen über partielle Funktionen zwar etwas aufwendiger, als wenn der Definitionsbereich vorgegeben ist, entspricht aber dem intuitiven Verständnis partieller Funktionen erheblich besser.

Formal bedeutet dies, daß jeder Funktion $f \in S \not\to T$ ein *Domain-Prädikat* $\underline{\mathsf{dom}(f)} \in S \to \mathbb{P}_1$ zugeordnet wird, welches für Elemente von S angibt, ob sie zum Definitionsbereich von S gehören oder nicht. Die Konsequenz dieser Vorgehensweise ist, daß eine partielle Funktion $f \in S \not\to T$ in der Typentheorie tatsächlich wie eine *totale* Funktion behandelt werden kann, nämlich als ein Element von $\{\mathsf{x}:S \mid \mathsf{dom}(f)(\mathsf{x})\} \to T$.

In Anlehnung an die Notation der Programmiersprache ML bezeichnen wir die kanonischen Elemente des Typs $S \not\to T$ mit $\underline{\mathsf{letrec}\ f(x) = t}$.[75] Dabei ist f allerdings kein Name, der hierdurch neu eingeführt wird, sondern wie x nur eine Variable, die in t gebunden wird. Die nichtkanonische Funktionsapplikation bezeichnen wir wie bei den 'normalen' Funktionen mit $\underline{f\ (t)}$, wobei jedoch zu bedenken ist, daß hinter dieser Darstellungsform ein völlig anderer interner Term steht. Die Reduktion einer Applikation rekursiv definierter Funktionen $(\mathsf{letrec}\ f(x) = t)\ (u)$ wird definiert durch Ausführung eines Rekursionsschrittes und ergibt $t[\mathsf{letrec}\ f(x) = t,\ u\ /\ f, x]$. Der Term $\mathsf{letrec}\ f(x) = t$ zeigt somit bei der Reduktion dasselbe Verhalten wie die Funktion $\lambda \mathsf{y}.\mathsf{let}^*\ f(x) = t\ \mathsf{in}\ f(\mathsf{y})$. Da aber die Koppelung an einen fest vorstrukturierten Definitionsbereich entfällt, können wir $\mathsf{letrec}\ f(x) = t$ als eine Erweiterung der rekursiven Induktion betrachten.

Prinzipiell könnten wir bereits alleine auf der Basis der Reduktionsregel definieren, wann eine rekursive Funktion auf einer Eingabe $s \in S$ terminiert und einen Wert $t \in T$ liefert. In Einzelfällen ist es durchaus möglich, durch eine Reihe von Reduktionsschritten zu beweisen, daß $(\mathsf{letrec}\ f(x) = t)\ (u)\ =\ t' \in T$ gilt. So können wir zum Beispiel problemlos beweisen, daß

$$[\mathsf{letrec}\ \mathtt{f(y) = if\ y=1\ then\ 0\ else\ if\ y\ rem\ 2\ =0\ then\ f(x\div 2)\ else\ f(3*x+1)}]\ (1)\ =\ 0\ \in\ \mathbb{Z}$$

gilt. Ähnlich kann man dies für viele andere Eingaben rekursiv definierter Funktionen tun. Das Problem ist jedoch, daß diese Beweise für jede Eingabe einzeln geführt werden müssen. Ein induktives Schließen über den *gesamten* Definitionsbereich $\{\mathsf{y}:S \mid (\mathsf{letrec}\ f(x) = t)\ (u)\ \in\ T\}$, also über die exakte Menge aller Eingaben, auf denen die Funktion terminiert, ist dagegen im Allgemeinfall nicht möglich, obwohl diese eine induktive Struktur besitzt.

Genau aus diesem Grunde wurde das oben erwähnte Domain-Prädikat als weiterer nichtkanonischer Term partiell-rekursiver Funktionen eingeführt. $\mathsf{dom}(f)$ beschreibt einerseits eine Menge von Eingaben, auf denen f garantiert terminiert und bietet andererseits eine Möglichkeit, auf die induktive Struktur des Definitionsbereiches von f zuzugreifen. Dies geschieht dadurch, daß für kanonische Elemente von $S \not\to T$ das Redex $\mathsf{dom}(\mathsf{letrec}\ f(x) = t)$ zu einem induktiven Datentyp-Prädikat reduziert. Wegen der komplexen Struktur der Definitionsbereiche partiell-rekursiver Funktionen kann diese Reduktion jedoch nicht durch ein einfaches Termschema beschrieben werden, sondern muß durch einen Algorithmus berechnet werden. Das Kontraktum hat daher die Gestalt $\lambda x.\mathsf{rectype}\ F = \mathcal{E}[[t]]$, wobei $\mathcal{E}$ kein Ausdruck der Typentheorie ist sondern einen Meta-Algorithmus zur Transformation von NuPRL-Termen beschreibt, der bei der Ausführung der Reduktion aufgerufen wird.[76]

[74] Dieser Ansatz entstammt einer Idee, die Herbrand [van Heijenoort, 1967] bei einer Untersuchung von Algorithmen im Zusammenhang mit konstruktiver Mathematik und Logik gewonnen hat. Man verzichtet auf den genauen Definitionsbereich, den man wegen des Halteproblems nicht automatisch bestimmen kann, und schränkt sich ein auf diejenigen Elemente ein, bei denen eine Terminierung aus der syntaktischen Struktur des Algorithmus gefolgert werden kann. Zugunsten der Beweisbarkeit werden also unter Umständen einige Eingaben, auf denen der Algorithmus terminiert, als unzulässig erklärt. Dies öffnet einen Weg, partielle Funktionen in ein immer terminierendes Beweiskonzept zu integrieren.

[75] Die interne Bezeichnung $\mathbf{fix}\{\}(f, x.t)$ ist an die LCF Tradition [Gordon *et.al.*, 1979] angelehnt und soll daran erinnern, daß die Semantik rekursiver Funktionen durch den kleinsten Fixpunkt der Rekursionsgleichung erklärt ist.

[76] Dieser Algorithmus muß natürlich immer terminieren, da wir verlangen, daß einzelne Reduktionsschritte in der Typentheorie immer zu einem Ergebnis führen. Zudem muß der Definitionsbereich für kanonische Elemente immer wohldefiniert sein, um ein formales Schließen zu ermöglichen. Aufgrund dieser Forderung fällt die Beschreibung des Definitionsbereichs etwas grober aus, als eine optimale Charakterisierung. Eine ausführlichere Beschreibung von $\mathcal{E}$ findet man in [Constable & Mendler, 1985] und [Constable *et.al.*, 1986, Seite 249].

kanonisch (Typen)	kanonisch (Elemente)	nichtkanonisch
$\mathbf{pfun}\{\}(S;T)$	$\mathbf{fix}\{\}(f, x.t)$	$\mathbf{dom}\{\}(\boxed{f})$, $\mathbf{apply\_p}\{\}(\boxed{f};t)$
$S \not\to T$	$\mathsf{letrec}\ f(x) = t$	$\mathsf{dom}(\boxed{f})$, $\boxed{f}\ (t)$

Zusätzliche Einträge in die Operatorentabelle

Redex		Kontraktum
$(\mathsf{letrec}\ f(x) = t)\ (u)$	$\xrightarrow{\beta}$	$t[\mathsf{letrec}\ f(x) = t,\ u\ /\ f, x]$
$\mathsf{dom}(\mathsf{letrec}\ f(x) = t)$	$\xrightarrow{\beta}$	$\lambda x.\mathsf{rectype}\ F = \mathcal{E}[[t]]$

Zusätzliche Einträge in die Redex–Kontrakta Tabelle

Typsemantik	
$S_1 \not\to T_1 = S_2 \not\to T_2$	falls $S_1 = S_2$ und $T_1 = T_2$
Elementsemantik	
$\mathsf{letrec}\ f_1(x_1) = t_1 = \mathsf{letrec}\ f_2(x_2) = t_2\ \in\ S \not\to T$	falls $S \not\to T$ Typ und $\{x:S \mid \mathsf{dom}(\mathsf{letrec}\ f_1(x_1) = t_1)(x)\}$ $= \{x:S \mid \mathsf{dom}(\mathsf{letrec}\ f_2(x_2) = t_2)(x)\}$ und $t_1[\mathsf{letrec}\ f_1(x_1) = t_1,\ s_1\ /\ f_1, x_1]$ $= t_2[\mathsf{letrec}\ f_2(x_2) = t_2,\ s_2\ /\ f_2, x_2]\ \in T$ für alle Terme s_1 und s_2 mit $s_1 = s_2 \in S$.
$S_1 \not\to T_1 = S_2 \not\to T_2\ \in\ \mathsf{U}_j$	falls $S_1 = S_2\ \in\ \mathsf{U}_j$ und $T_1 = T_2\ \in\ \mathsf{U}_j$

Zusätzliche Einträge in den Semantiktabellen

$\Gamma \vdash S_1 \not\to T_1\ =\ S_2 \not\to T_2\ \in\ \mathsf{U}_j\ \lfloor\mathsf{Ax}\rfloor$
  by $\underline{\mathtt{pfunEq}}$
  $\Gamma \vdash S_1\ =\ S_2\ \in\ \mathsf{U}_j\ \lfloor\mathsf{Ax}\rfloor$
  $\Gamma \vdash T_1\ =\ T_2\ \in\ \mathsf{U}_j\ \lfloor\mathsf{Ax}\rfloor$

$\Gamma \vdash (\mathsf{letrec}\ f_1(x_1) = t_1)$
  $=\ (\mathsf{letrec}\ f_2(x_2) = t_2)\ \in\ S \not\to T\ \lfloor\mathsf{Ax}\rfloor$
  by $\underline{\mathtt{fixEq}\ j}$
  $\Gamma \vdash \mathsf{letrec}\ f_1(x_1) = t_1\ \in\ S \not\to T\ \lfloor\mathsf{Ax}\rfloor$
  $\Gamma \vdash \mathsf{letrec}\ f_2(x_2) = t_2\ \in\ S \not\to T\ \lfloor\mathsf{Ax}\rfloor$
  $\Gamma \vdash \{x:S \mid \mathsf{dom}(\mathsf{letrec}\ f_1(x_1) = t_1)(x)\}$
    $=\ \{x:S \mid \mathsf{dom}(\mathsf{letrec}\ f_2(x_2) = t_2)(x)\} \in \mathsf{U}_j\ \lfloor\mathsf{Ax}\rfloor$
  $\Gamma,\ y:\{x:S \mid \mathsf{dom}(\mathsf{letrec}\ f_1(x_1) = t_1)(x)\}$
    $\vdash\ (\mathsf{letrec}\ f_1(x_1) = t_1)\ (y)$
    $=\ (\mathsf{letrec}\ f_2(x_2) = t_2)\ (y)\ \in\ T\ \lfloor\mathsf{Ax}\rfloor$

$\Gamma \vdash \mathsf{letrec}\ f(x) = t\ \in\ S \not\to T\ \lfloor\mathsf{Ax}\rfloor$
  by $\underline{\mathtt{fixMem}\ j}$
  $\Gamma \vdash S \not\to T\ \in\ \mathsf{U}_j\ \lfloor\mathsf{Ax}\rfloor$
  $\Gamma,\ f':S \not\to T,\ x':S \vdash \mathcal{E}[[\,t[f', x'\ /\ f, x]\,]]\ \in \mathsf{U}_j\ \lfloor\mathsf{Ax}\rfloor$
  $\Gamma,\ f':S \not\to T,\ x':S,\ \mathcal{E}[[\,t[f', x'\ /\ f, x]\,]]$
    $\vdash\ t[f', x'\ /\ f, x]\ \in\ T\ \lfloor\mathsf{Ax}\rfloor$

$\Gamma \vdash f_1\ (t_1)\ =\ f_2\ (t_2)\ \in\ T\ \lfloor\mathsf{Ax}\rfloor$
  by $\underline{\mathtt{apply\_pEq}\ S \not\to T}$
  $\Gamma \vdash f_1\ =\ f_2\ \in\ S \not\to T\ \lfloor\mathsf{Ax}\rfloor$
  $\Gamma \vdash t_1\ =\ t_2\ \in\ \{x:S \mid \mathsf{dom}(f_1)(x)\}\ \lfloor\mathsf{Ax}\rfloor$

$\Gamma,\ f:S \not\to T,\ \Delta \vdash C\ \lfloor\mathbf{ext}\ t[f\ (s), \mathsf{Axiom}\ /\ y, v]\rfloor$
  by $\underline{\mathtt{pfunE}\ i\ s}$
  $\Gamma,\ f:S \not\to T,\ \Delta$
    $\vdash\ s\ \in\ \{x:S \mid \mathsf{dom}(f_1)(x)\}\ \lfloor\mathsf{Ax}\rfloor$
  $\Gamma,\ f:S \not\to T,\ \Delta,\ y:T,\ v:\ y = f\ (s)\ \in\ T$
    $\vdash\ C\ \lfloor\mathbf{ext}\ t\rfloor$

$\Gamma \vdash (\mathsf{letrec}\ f(x) = t)\ (u)\ =\ t_2\ \in\ T\ \lfloor\mathsf{Ax}\rfloor$
  by $\underline{\mathtt{apply\_pRed}}$
  $\Gamma \vdash t[\mathsf{letrec}\ f(x) = t,\ u\ /\ f, x]\ =\ t_2\ \in\ T\ \lfloor\mathsf{Ax}\rfloor$

$\Gamma \vdash \mathsf{dom}(f_1)\ =\ \mathsf{dom}(f_2)\ \in\ S \to \mathbb{P}_j\ \lfloor\mathsf{Ax}\rfloor$
  by $\underline{\mathtt{domEq}\ S \not\to T}$
  $\Gamma \vdash f_1\ =\ f_2\ \in\ S \not\to T\ \lfloor\mathsf{Ax}\rfloor$
  $\Gamma \vdash S \not\to T\ \in\ \mathsf{U}_j\ \lfloor\mathsf{Ax}\rfloor$

Inferenzregeln

Abbildung 3.27: Syntax, Semantik und Inferenzregeln partiell rekursiver Funktionen

Im Falle der oben erwähnten `3x+1`-Funktion liefert der Algorithmus $\mathcal{E}$ zum Beispiel folgendes Domain-Prädikat:

$$
\begin{array}{ll}
\lambda x.\ \mathsf{rectype}\ \mathtt{D(y)} = & \mathtt{y}=1 \in \mathbb{Z} \\
& \vee\ 1\mathtt{<y}\ \wedge\ \mathtt{y}\,\mathsf{rem}\,2\ =\ 0 \in \mathbb{Z}\ \wedge\ \mathtt{D(y\div 2)} \\
& \vee\ 1\mathtt{<y}\ \wedge\ \mathtt{y}\,\mathsf{rem}\,2\ =\ 1 \in \mathbb{Z}\ \wedge\ \mathtt{D(3*y+1)} \\
\mathsf{select}\ \mathtt{D}(x) &
\end{array}
$$

Erwartungsgemäß besitzt dieses Domain-Prädikat, das wir zugunsten der natürlicheren Darstellung durch einen parametrisierten induktive Datentyp beschreiben (, in dem wir Datentypkonstrukte durch ihr logisches Gegenstück ersetzt haben), eine sehr große Ähnlichkeit zu der ursprünglichen Funktionsdefinition, da es die rekursive Abarbeitung des Algorithmus in einer induktiven Typstruktur wiederspiegeln muß.

Syntax, Semantik und Regeln der partiellen Funktionen sind in Abbildung 3.27 zusammengefaßt.[77] Man beachte hierbei, daß das Domain-Prädikat beim Schließen über rekursive Funktionen eine zentrale Rolle spielt, sowohl was die Einführung als auch was die Applikation betrifft. Um also $f\ (t)$ untersuchen zu können, muß insbesondere überprüft werden, daß t tatsächlich auch zum Definitionsbereich von f gehört. Dies garantiert, daß f tatsächlich auf t terminiert. Es sei allerdings nochmals darauf hingewiesen, daß f auch auf Eingaben t' terminieren kann, für die $\mathsf{dom}(f)(t')$ *nicht* nachgewiesen werden kann. In diesen Fällen kann man allerdings keinerlei Eigenschaften von $f\ (t')$ formal beweisen.

Partiell rekursive Funktionen können nicht als Extrakt-Term einer Einführungsregel `fixI` erzeugt werden, da hierfür die rekursive Struktur des Definitionsbereiches bereits bekannt sein muß.[78] In diesem Fall kann man auf die Regel `recE` der induktiven Datentypen zurückgreifen, die das wohlfundierte Gegenstück zu einer rekursiven Funktion erzeugt.

Im Gegensatz zu totalen Funktionenräumen dürfen partiell-rekursive Funktionenräume ohne Einschränkung innerhalb von induktiven Datentypen vorkommen. So ist zum Beispiel $T\ \equiv\ \mathsf{rectype}\ \mathtt{X} = \mathtt{X} \not\to \mathbb{Z}$ ein erlaubter Datentyp, da der Term $\mathsf{letrec}\ \mathtt{f(x)} = \mathtt{x\ (x)}$, dessen Gegenstück $\lambda\mathtt{x}.\ \mathtt{x\,x}$ in Beispiel 3.5.4 so viele Probleme erzeugte, ein legitimes Element von T ist. Grund hierfür ist, daß $\mathsf{letrec}\ \mathtt{f(x)} = \mathtt{x\ (x)}$ eine *partielle* Funktion beschreibt, die nur auf Elementen definiert ist, welche selbst wiederum partielle Funktionen sind und auf sich selbst angewandt werden dürfen – für die also $\mathsf{dom}\mathtt{(x)(x)}$ gilt). Ein solches Element des Definitionsbereiches von $\mathsf{letrec}\ \mathtt{f(x)} = \mathtt{x\ (x)}$ ist zum Beispiel die Identitätsfunktion $\mathsf{letrec}\ \mathtt{f(x)} = \mathtt{x}$, die ohne Bedenken auf sich selbst angewandt werden kann.

Zum Abschluß sei bemerkt, daß wie bei den induktiven Datentypen auch bei partiellen Funktionen ist eine simultane Rekursion sinnvoll ist. Die allgemeinste Form einer rekursiven Funktionsdefinition lautet daher

$$\mathsf{letrec}\ f_1(x_1)\ =\ t_1\ \mathsf{and}\ \ldots\ \mathsf{and}\ f_n(x_n)\ =\ t_n\ \mathsf{select}\ f_i$$

3.5.3 Unendliche Objekte

Bei der Untersuchung rekursiv definierter Datentypen in Abschnitt 3.5.1 haben wir die kleinste Lösung einer rekursiven Typgleichung betrachtet. Die Gleichung

$$\mathtt{bintree}\ =\ \mathbb{Z}\ +\ \mathbb{Z} \times \mathtt{bintree} \times \mathtt{bintree}$$

wurde als Beschreibung aller Terme interpretiert, die sich in endlich vielen Schritten durch die beiden Alternativen der rechten Seite ausdrücken lassen. Dies führte zu einer Interpretation von `bintree` als dem Datentyp

[77] Es sei an dieser Stelle erwähnt, daß das gegenwärtige NuPRL System die partiellen Funktionen nicht explizit unterstützt sondern durch den Fixpunktkombinator $\mathbf{Y}$ (siehe Definition 2.3.22 auf Seite 57) simuliert. Der Algorithmus $\mathcal{E}$ zur bestimmung des Definitionsbereiches ist allerdings auf der Meta-Ebene eingebaut. Man kann daher im System durch ein Kommando `add_recdef` auf der Meta-Ebene eine rekursive Funktionsgleichung eingeben und das System generiert sowohl die rekursive Funktion f als auch eine Beschreibung des Definitionsbereiches, indem es ein Theorem der Form

$$\vdash\ f\ \in\ \{\mathtt{x}\!:\!S \mid \mathsf{rectype}\ F = \mathcal{E}[[t]]\ \} \to T$$

generiert. Diese Simulation hat allerdings den Nachteil, daß der Typ $S \not\to T$ nicht explizit in Beweisen benutzt werden kann.

[78] In `fixMem` wird der Algorithmus $\mathcal{E}$ eingesetzt, der eine vorgegebene Funktionsdefinition benötigt. Diese Regel ist also nicht so leicht umkehrbar in eine implizite Erzeugungsregel, wie dies bei anderen Typkonstrukten der Fall ist.

aller endlichen Binärbäume über ganzen Zahlen. Genauso aber hätten wir auch nach der *größten* Lösung dieser Gleichung suchen können. Ein unendlicher Binärbaum erfüllt nämlich ebenfalls die Rekursionsgleichung, da er aus einer Wurzel und zwei unendlichen Binärbäumen zusammengesetzt ist. Somit könnten wir `bintree` genausogut auch als Datentyp aller *endlichen und unendlichen* Binärbäume auffassen. Um diese Semantik innerhalb der Typentheorie zu fixieren, müssen wir einen neuen Term einführen, dessen Syntax ähnlich zu derjenigen der induktiven Typen ist, aber doch auf den Unterschied hinweist. Um auszudrücken, daß wir `bintree` als den *größten* Fixpunkt interpretieren wollen, der die obige Gleichung erfüllt schreiben wir daher

$$\mathsf{inftype}\ \mathtt{bintree} = \mathbb{Z}\ +\ \mathbb{Z} \times \mathtt{bintree} \times \mathtt{bintree}.$$

Eine derartige maximale Fixpunktsemantik ist durchaus von praktischer Bedeutung für die Programmierung. Will man zum Beispiel dauernd aktive Prozesse wie Betriebssysteme oder Editoren programmieren, so muß man prinzipiell davon ausgehen, daß ein unendlicher Datenstrom von Eingaben verarbeitet werden muß. Um derartige Datenströme präzise zu beschreiben, braucht man rekursive Gleichungen, die unendliche Lösungen zulassen. Eine NuPRL Formalisierung wäre zum Beispiel:

$$\mathsf{inftype}\ \mathtt{stream} = \mathsf{Atom}\ \times\ \mathtt{stream}$$

Unendliche Datentypen können maschinenintern zum Beispiel relativ einfach durch zyklische Pointerstrukturen (rückwärts verkettete Listen) realisiert werden, die nicht vollständig sondern nur bedarfsweise verarbeitet werden. Aus diesem Grunde werden sie auch als *lässige Datentypen* angesehen.

Derzeit sind unendliche Datentypen noch nicht als fester Bestandteil in die Typentheorie aufgenommen worden und sollen deshalb hier auch nicht weiter vertieft werden. Eine ausführlichere Abhandlung über unendliche Datentypen kann man in [Mendler, 1987a] finden.

3.6 Ergänzungen zugunsten der praktischen Anwendbarkeit

In den bisherigen Abschnitten haben wir die Grundkonzepte der intuitionistischen Typentheorie besprochen und Inferenzregeln vorgestellt, die eine formales Schließen über diese Konzepte ermöglichen. Wir haben uns dabei bewußt auf einen relativ kleinen Satz von Regeln für jedes einzelne Typkonstrukt eingeschränkt, um die ohnehin schon sehr große Anzahl formaler Regeln nicht völlig unüberschaubar werden. Daher ist in vielen Einzelfällen das formale Schließen über relativ einfache Zusammenhänge bereits sehr aufwendig. Deshalb wurde das Inferenzsystem von NuPRL um eine Reihe von Regeln ergänzt, die aus theoretischer Sicht nicht nötig (weil redundant) sind, das praktische Arbeiten mit dem NuPRL System aber erheblich erleichtern. Die meisten dieser Regeln, die wir im folgenden kurz beschreiben wollen, beinhalten etwas komplexere Algorithmen oder beziehen sich auf *Objekte* wie Theoreme und Definitionen, die innerhalb der Bibliothek des NuPRL Systems unter einem bestimmten Namen abgelegt sind.

- Bei der Fixierung der Bedeutung typentheoretischer Ausdrücke haben wir in Definition 3.2.15 Urteile über nichtkanonische Terme mithilfe von Urteilen über die Werte dieser Terme definiert. Es ist also legitim, einen nichtkanonischen Term t in einer Konklusion oder einer Annahme durch einen anderen Term zu ersetzen, welcher zu dem gleichen Wert reduziert werden kann, insbesondere also einen Teilterm von t zu reduzieren oder eine Reduktion, die zu einem Teilterm von t geführt hat, wieder rückgängig zu machen. Zu diesem Zweck wurden insgesamt vier *Berechnungsregeln* eingeführt, deren Anwendung durch eine *Markierung* (tagging) der entsprechenden Teilterme kontrolliert wird.

  Eine solche Markierung eines Teilterms s von t geschieht dadurch, daß s durch $\mathtt{[[*}\!:\!s\mathtt{]]}$ bzw. $\mathtt{[[}n\!:\!s\mathtt{]]}$ ersetzt wird, wobei n die Anzahl der durchzuführenden Reduktionsschritte angibt. Eine Markierung mit $*$ bedeutet, daß so viele (lässige) Reduktionsschritte wie möglich ausgeführt werden sollen. Im Falle einer Rückwärtsberechnung muß anstelle von s der markierte Term angegeben werden, der zu s reduziert. Für die Anwendung der Regel ist der gesamte Term mit seinen markierten Teiltermen anzugeben. Die Berechnungsregel führt die entsprechenden Reduktionen durch und ersetzt t dann durch den reduzierten bzw. rückreduzierten Term (bezeichnet durch $t\!\downarrow_{tagt}$ bzw. $t\!\uparrow_{tagt}$).

$\Gamma \vdash C \;\lfloor \textbf{ext}\ t \rfloor$ **by** `compute` *tagC* $\Gamma \vdash C\!\downarrow_{tagC} \;\lfloor \textbf{ext}\ t \rfloor$	$\Gamma \vdash C \;\lfloor \textbf{ext}\ t \rfloor$ **by** `rev_compute` *tagC* $\Gamma, \vdash C\!\uparrow_{tagC} \;\lfloor \textbf{ext}\ t \rfloor$
$\Gamma,\ z{:}T,\ \Delta \vdash C \;\lfloor \textbf{ext}\ t \rfloor$ **by** `computeHyp` *i tagT* $\Gamma,\ z{:}T\!\downarrow_{tagT},\ \Delta \vdash C \;\lfloor \textbf{ext}\ t \rfloor$	$\Gamma,\ z{:}T,\ \Delta \vdash C \;\lfloor \textbf{ext}\ t \rfloor$ **by** `rev_computeHyp` *i tagT* $\Gamma,\ z{:}T\!\uparrow_{tagT},\ \Delta \vdash C \;\lfloor \textbf{ext}\ t \rfloor$
$\Gamma \vdash C \;\lfloor \textbf{ext}\ t \rfloor$ **by** `unfold` *def-name* $\Gamma \vdash C_{\downarrow} \;\lfloor \textbf{ext}\ t \rfloor$	$\Gamma \vdash C_{\downarrow} \;\lfloor \textbf{ext}\ t \rfloor$ **by** `fold` *def-name* $\Gamma \vdash C \;\lfloor \textbf{ext}\ t \rfloor$
$\Gamma,\ z{:}T,\ \Delta \vdash C \;\lfloor \textbf{ext}\ t \rfloor$ **by** `unfoldHyp` *i def-name* $\Gamma,\ z{:}T_{\downarrow},\ \Delta \vdash C \;\lfloor \textbf{ext}\ t \rfloor$	$\Gamma,\ z{:}T_{\downarrow},\ \Delta \vdash C \;\lfloor \textbf{ext}\ t \rfloor$ **by** `foldHyp` *i def-name* $\Gamma,\ z{:}T,\ \Delta \vdash C \;\lfloor \textbf{ext}\ t \rfloor$
$\Gamma,\ z{:}T,\ \Delta \vdash C \;\lfloor \textbf{ext}\ t \rfloor$ **by** `replaceHyp` *i S j* $\Gamma,\ z{:}S,\ \Delta \vdash C \;\lfloor \textbf{ext}\ t \rfloor$ $\Gamma,\ z{:}T,\ \Delta \vdash T = S \in \mathbb{U}_j \;\lfloor \text{Ax} \rfloor$	
$\Gamma \vdash C \;\lfloor \textbf{ext}\ t \rfloor$ **by** `lemma` *theorem-name*	$\Gamma \vdash t \in T \;\lfloor \text{Ax} \rfloor$ **by** `extract` *theorem-name*
$\Gamma \vdash C \;\lfloor \textbf{ext}\ t[\sigma] \rfloor$ **by** `instantiate` $\Gamma'\ C'\ \sigma$ $\Gamma' \vdash C' \;\lfloor \textbf{ext}\ t \rfloor$	$\Gamma \vdash C \;\lfloor \textbf{ext}\ t[y/x] \rfloor$ **by** `rename` *y x* $\Gamma[x/y] \vdash C[x/y] \;\lfloor \textbf{ext}\ t \rfloor$
$\Gamma \vdash s = t \in T \;\lfloor \text{Ax} \rfloor$ **by** `equality`	$\Gamma \vdash C \;\lfloor \textbf{ext}\ t \rfloor$ **by** `arith` *j* $\Gamma \vdash s_1 \in \mathbb{Z} \;\lfloor \text{Ax} \rfloor$ $\vdots$

Abbildung 3.28: Zusätzliche Inferenzregeln von NuPRL

Die Berechnungsregeln machen die einzelnen Reduktionsregeln eigentlich hinfällig, da sie diese subsumieren. Aus theoretischen Gründen müssen diese jedoch in der Theorie enthalten bleiben. Für partiell rekursive Funktionen gibt es *keine direkten Berechnungsregeln*, da für diese eine Terminierung nicht garantiert wäre.

- Die Existenz von benutzerdefinierten Erweiterungen wie zum Beispiel Logik-Operatoren gemäß Definition 3.3.3 auf Seite 134 macht es zuweilen nötig, zwischen dem neu eingeführten Term und seiner ausführlichen Definition hin- und herzuwechseln. Dies ist insbesondere dann wichtig, wenn die Grundeigenschaften des neuen Konzeptes bewiesen werden sollen. Aus diesem Grunde wurden Regeln hinzugenommen, die in einer Konklusion bzw. einer Hypothese derartige Definitionen auflösen (`unfold`) oder zurückfalten (`fold`). Anzugeben ist hierbei jeweils der Name, unter dem die Definition in der Bibliothek abgelegt ist.[79]

- Die `replaceHyp`-Regel ist eine Variante der Substitutionsregel. Sie erlaubt es, eine Hypothese T durch eine gleichwertige Hypothese S zu ersetzen

- Die `lemma`- und `extract`-Regeln erlauben einen Zugriff auf bereits bewiesene Theoreme der NuPRL-Bibliothek. Die `lemma`-Regel entspricht der `hypothesis`-Regel in dem Sinne, daß sie eine Konklusion

[79] Intern wird die Auflösung von Definitionen genauso gehandhabt wie die Durchführung eines Reduktionsschrittes.

dadurch beweist, daß auf eine bereits bekannte Tatsache – nämlich das bewiesene Theorem – verwiesen wird. Die `extract`-Regel entspricht in ähnlicher Weise der `hypEq`-Regel. hier wird eine Konklusion $t \in T$ dadurch bewiesen, daß ein Theorem genannt wird, dessen Beweisziel T und dessen Extraktterm t ist. In beiden Fällen ist der Name des entsprechenden Theorems anzugeben.

- Die `instantiate`-Regel ermöglicht eine Verallgemeinerung des Beweiszieles. $\Gamma \vdash C$ wird als Spezialfall (Instanz) eines allgemeineren Zieles $\Gamma' \vdash C'$ angesehen und folgt deshalb hieraus. Als Parameter sind Γ', C' und die Substitution σ anzugeben, welche Γ' zu Γ und C' zu C instantiiert.

- Mit der `rename`-Regel können Variablen innerhalb eines Beweiszieles umbenannt werden. Dies ist nützlich, wenn die Variablennamen mit einer Bedeutung assoziiert werden sollen oder wenn unsichtbare Variablen (wie bei unabhängigen Funktionenräumen) sichtbar gemacht werden sollen.

- Die `equality`- und `arith`-Regeln schließlich sind dafür geschaffen worden, in zwei speziellen Anwendungsbereichen dem Benutzer die Anwendung vieler einzelner Regeln zum Beweis relativ trivialer Aussagen zu ersparen. Hinter diesen Regeln steckt nicht mehr ein einzelner Beweisschritt sondern ein aufwendiger, theoretisch abgesicherter Algorithmus, welcher Entscheidungen über die Gültigkeit von Gleichheitsaussagen bzw. von einfachen arithmetische Aussagen treffen kann.

  So subsumiert die `equality`-Regel die Anwendung vieler Einzelschritte, die nur auf Kommutativität, (typisierter) Reflexivität oder Transitivität beruhen, in einem einzigen Schritt. Die `arith`-Regel beinhaltet alle Regeln über Addition, Subtraktion, Multiplikation, Division, Divisionsrest, arithmetische Tests sowie diverse Monotoniegesetze und kann praktisch alle arithmetischen Aussagen in einem Schritt beweisen, die keine Induktion benötigen.

  Die Algorithmen, die hinter diesen *Entscheidungsprozeduren* stehen und ihre Rechtfertigung werden wir im Kapitel 4.3 ausführlich besprechen.

3.7 Diskussion

Wir haben in diesem Kapitel die intuitionistische Typentheorie als einen Formalismus vorgestellt, der für sich in Anspruch nimmt, alle Aspekte von Mathematik und Programmierung zuverlässig repräsentieren zu können. Dabei sind wir davon ausgegangen, daß es für gewisse mathematische Grundkonstrukte ein intuitives Verständnis gibt, was sich nicht weiter in sinnvolle elementarer Bestandteile zerlegen läßt. Beim Aufbau der formalen Theorie ging es deshalb darum, die “natürlichen” Gesetze der zentralen in Mathematik und Programmierung vorkommenden Grundkonzepte in Form einer präzisen Semantikdefinition auszuformulieren und darüber hinaus auch für einen praktisch einsetzbaren Inferenzkalkül verwendbar zu machen. Auf eine minimale Theorie, in der kein Konzept durch ein anderes simuliert werden kann, wurde zugunsten einer natürlicheren Formalisierung der Grundkonstrukte verzichtet. Aus diesem Grunde ist die Theorie relativ umfangreich und enthält mehr als 100 Inferenzregeln.[80] Um die Theorie dennoch überschaubar zu halten und eine spätere Automatisierung der Beweisführung zu unterstützen, haben wir zu Beginn dieses Kapitels eine Systematik für einen einheitlichen Aufbau von Syntax, Semantik und Inferenzregeln entwickelt und anschließend die eigentliche Theorie in 4 Phasen – mathematische Grundkonstrukte, Logik, Grundkonstrukte der Programmierung und Rekursion – entwickelt.

Im Laufe dieser Entwicklung sind einige Prinzipien wichtig geworden, welche die konkrete Ausgestaltung der Theorie entscheidend beeinflußt hatten.

[80] Dies ist aber nur ein scheinbarer Nachteil gegenüber minimalen Theorien. Bei letzteren müßten Konzepte wie Zahlen, Listen auf die bestehende Theorie aufgesetzt und ihre Grundgesetze als Theoreme bewiesen werden, bevor man sie benutzen kann. Andernfalls würde alles formale Schließen über Zahlen auf einer Ebene unterhalb der Grundgesetze von Zahlen stattfinden, was eine (interaktive) Beweisführung durch einen Menschen praktisch ausschließt.

1. Die Typentheorie ist als eine konstruktive mathematische *Grundlagentheorie* ausgelegt, deren Konzepte sich direkt aus einem intuitiven mathematischen Verständnis ableiten lassen und nicht formal auf einer anderen Theorie abgestützt werden (siehe Abschnitt 3.1.1, Seite 94).

2. Die wichtigsten Grundkonstrukte der Mathematik und Programmierung sowie ihre Gesetze und Regeln sind als feste Bestandteile der Theorie formalisiert (Abschnitt 3.1.2, Seite 95).

3. Die Abstraktionsform eines Terms und seine Darstellungsform werden getrennt voneinander behandelt aber simultan betrachtet (Entwurfsprinzip 3.2.4, Seite 103).

4. Semantik und Inferenzregeln basieren auf einem immer terminierenden Reduktionskonzept. Als Berechnungsverfahren wurde *Lazy Evaluation* fixiert (Entwurfsprinzip 3.2.9, Seite 107).

5. *Urteile* sind die semantischen Grundbausteine, mit denen Eigenschaften von Termen und die Beziehungen zwischen Termen formuliert werden (Abschnitt 3.2.2.2, Seite 109).

6. Die semantische Typeigenschaft wird innerhalb des Inferenzsystems durch eine kumulative Hierarchie von Universen syntaktisch repräsentiert (Entwurfsprinzip 3.2.21, Seite 115).

7. Die Typentheorie unterstützt explizites und implizites Schließen, also die Überprüfung eines Urteils und die Konstruktion von Termen, die ein Urteil erfüllen. Zu diesem Zweck werden Urteile syntaktisch immer implizit dargestellt (Entwurfsprinzip 3.2.24, Seite 117).

8. Logische Aussagen werden dargestellt durch Typen, deren Elemente den Beweisen der Aussagen entsprechen ("*Propositionen als Typen*"-Prinzip 3.2.22, Seite 116).

   Konsequenterweise ist die Prädikatenlogik kein Grundbestandteil der Typentheorie sondern ein simulierbares Konzept, welches als *konservative Erweiterung* hinzugenommen werden kann. Hierbei spielt die Curry-Howard Isomorphie zwischen Logik und Typentheorie, die im Prinzip auch eine Logik höherer Stufe ermöglicht, eine fundamentale Rolle (Abschnitt 3.3.1, Seite 127).

9. Aufgrund des Prinzips "*Propositionen als Typen*" ist Programmentwicklung dasselbe wie Beweisführung ("*Beweise als Programme*"-Prinzip 3.4.3, Seite 139). Insbesondere ist induktive Beweisführung dasselbe wie die Konstruktion rekursiver Programme (Entwurfsprinzip 3.4.4, Seite 142).

Hinter vielen dieser Prinzipien steckt eine bewußte Entwurfsentscheidung, bei denen die absehbaren Nachteile zugunsten der erwarteten Vorteile in Kauf genommen wurden. Die Entscheidungen hätten durchaus auch anders gefällt werden können und hätten dann zu anderen Ausformulierungen der Typentheorie geführt.[81] Aus praktischer Hinsicht sind besonders die letzten beiden Prinzipien bedeutend, denn sie ermöglichen es, innerhalb eines einzigen in sich geschlossenen Kalküls über Programme und Beweise zu schließen und diese aus einer vorgegebenen Problemstellung auch zu konstruieren. Die Typentheorie eignet sich damit für die mathematische Beweisführung, Programmverifikation und die Synthese von Programmen aus ihren Spezifikationen.

Aus theoretischer Hinsicht ist die Typentheorie ein extrem mächtiger Formalismus, da es nun prinzipiell möglich ist, formale Schlüsse über alle Aspekte der Mathematik und Programmierung zu führen. Dennoch zeigen die Beispiele relativ schnell Grenzen des praktischen Einsatzes der bisherigen Theorie auf. Bereits einfache Aufgaben – wie zum Beispiel der Nachweis der Existenz einer Integerquadratwurzel (siehe Beispiel 3.4.9 auf Seite 154) – können nur mit einem relativ großen Aufwand gelöst werden und lassen sich kaum vollständig präsentieren. Zwei wesentliche Probleme machen sich hierbei bemerkbar.

[81] Diese Wege wurden durchaus beschritten und zu anderen Typtheorien mit entsprechend anderen Schwerpunkten geführt haben. Die wichtigsten Vertreter sind Girard's *System* $\mathcal{F}$ und $\mathcal{F}_\omega$ [Girard, 1971, Girard, 1972, Girard, 1986, Girard *et.al.*, 1989], der *Kalkül der Konstruktionen* von Coquand und Huet [Coquand & Huet, 1985, Coquand & Huet, 1988, Harper & Pollack, 1989, Coquand, 1990] sowie seine Erweiterung *ECC* [Luo, 1989, Luo, 1990, Pollack, 1994] die *lineare Logik* [Girard, 1987] sowie diverse leicht modifizierte Formulierungen von Martin-Löf's Typentheorie.

- Die Entwicklung eines formalen Beweises ist für einen Menschen mit extrem viel Schreibarbeit verbunden, bei der immer eine gewisse Gefahr für Schreibfehler besteht, welche den geführten Beweis verfälschen könnten. Dies ist nicht verwunderlich, denn der Zweck formaler Kalküle ist ja eigentlich, eine schematische Beweisführung zu unterstützen, die nicht unbedingt von einem Menschen ausgeführt werden muß. Die Ausführung formaler Beweise "von Hand" dient eigentlich nur Kontroll- und Übungszwecken, um ein gewisses Gefühl für den Kalkül zu entwickeln. Wirkliche Beweise sollten allerdings interaktiv mit Hilfe von Rechnern geführt werden, welche die Ausführung von Regeln übernehmen und vom Menschen nur die Angabe der auszuführenden Regel erwarten. Ein praktischer Umgang mit der Typentheorie macht also den Bau *interaktiver Beweissysteme* unbedingt erforderlich.

- Dies aber ist noch nicht genug, denn auch interaktive geführte Beweise enthalten eine große Menge von Details, welche sie sehr unübersichtlich werden lassen. Dies liegt zum einen an der großen Menge von Regeln, die nötig ist, um einen Beweis vollständig zu führen und zum anderen an den vielen Parametern, die für die Ausführung einer Regel erforderlich sind. Aus diesem Grunde ist es notwendig, eine Rechnerunterstützung anzubieten, die über die Möglichkeiten interaktiver Beweissysteme hinausgeht. Es müssen Techniken bereitgestellt werden, mit denen die Beweisführung zumindest zum Teil automatisiert werden kann – sowohl was die Auswahl der Regeln als auch die Bestimmung ihrer Parameter angeht.

Mit der Implementierung zuverlässiger interaktiver Beweissysteme (für die intuitionistische Typentheorie) und den prinzipiellen Möglichkeiten der Automatisierung der Beweisführung werden wir uns im Kapitel 4 befassen. Konkrete Formalisierungen von mathematischen Theorien und Programmierproblemen sowie Strategien für die Suche nach Beweisen und die Entwicklung von Programmen aus formalen Spezifikationen werden uns dann in den darauffolgenden Kapiteln beschäftigen.

3.8 Ergänzende Literatur

Zur intuitionistischen Typentheorie gibt es noch kein zusammenfassendes Lehrbuch sondern nur eine Reihe von Büchern und Fachartikeln, in denen komplette Formalisierungen vorgestellt oder spezielle Ergänzungen im Detail untersucht werden. Die Bandbreite der Notationen ist noch sehr breit und konnte bisher nicht standardisiert werden. Die hier betrachtete Formalisierung der Martin-Löf'schen Typentheorie lehnt sich an die im NuPRL-System verwandte Syntax an und ist weitestgehend in [Constable *et.al.*, 1986] beschrieben.

Lesenswert sind auch die Einführungskapitel einiger Bücher und Tutorials, die allgemeine Einführungen in die Martin-Löf'sche Typentheorie geben wie [Martin-Löf, 1973, Martin-Löf, 1984, Nordström *et.al.*, 1990, Backhouse *et.al.*, 1988a, Backhouse *et.al.*, 1988b, Backhouse, 1989]. Von Bedeutung für das Verständnis sind auch Martin-Löf's frühere Originalarbeiten [Martin-Löf, 1970, Martin-Löf, 1982]. Eine alternative Typentheorie beschreibt das Buch von Girard [Girard *et.al.*, 1989]. Es enthält aber eine Reihe detaillierter Beweise über Grundeigenschaften der Typentheorie, von denen sich fast alle direkt auf die hier vorgestellte Theorie übertragen lassen. Eine andere gute Quelle ist das Buch von Andrews [Andrews, 1986].

Der Aspekt der Programmentwicklung in der Typentheorie ind die Extraktion von Programmen aus Beweisen ist das Thema vieler Fachartikel wie [Bates & Constable, 1985, Backhouse, 1985, Constable, 1983, Constable, 1984, Constable, 1985, Constable, 1988, Constable & Howe, 1990b, Hayashi, 1986, Chisholm, 1987, Smith, 1983b, Nordström & Smith, 1984, Nordström *et.al.*, 1990, Paulin-Mohring, 1989, Leivant, 1990]. Jede Methode bringt vor und Nachteile mit sich. Eine optimale Extraktionsform konnte bisher allerdings noch nicht gefunden werden.

Ähnlich aufwendig sind rekursive Datentypen und Funktionen, für die ebenfalls noch keine optimale Darstellung gefunden werden konnte. Der gegenwärtige Stand der Forschung ist in [Constable *et.al.*, 1986, Kapitel 12] und Arbeiten wie [Constable & Mendler, 1985, Constable & Smith, 1987, Constable & Smith, 1988, Constable & Smith, 1993, Mendler, 1987a, Mendler, 1987b, Smith, 1988, Coquand & Paulin, 1988, Freyd, 1990, Martin-Löf, 1988] dokumentiert. Weitere Betrachtungen zu einzelnen Typkonstrukten und ihren Anwendungen sind in [Amadio & Cardelli, 1990, Backhouse, 1984, Chisholm, 1985, Cleaveland, 1987, Felty, 1991, Paulson, 1986, Pitts, 1989, Salvesen & Smith, 1988] zu finden.

Semantische Aspekte der Typentheorie werden in [Aczel, 1978, Allen, 1987a, Allen, 1987b, Allen *et.al.*, 1990, Constable, 1989, Constable & Howe, 1990b, Mendler, 1987a, Smith, 1984, Schwartzbach, 1986] betrachtet. Diese nicht immer ganz leicht zu lesenden Arbeiten geben wertvolle Einsichten in das Selbstverständnis der Typentheorie als formaler Grundlagentheorie und ihrer konkreten Ausformulierungen. Das Buch von Bishop [Bishop, 1967] liefert weitere Hintergründe zur Denkweise der konstruktiven Mathematik. Für ihr klassisches Gegenstück – die klassische Mengentheorie – lohnt es sich [Suppes, 1972, Quine, 1963] zu konsultieren.