Schematische Konstruktion von Globalsuchalgorithmen

Die Anwendung der intuitionistischen Typentheorie als Basiskalkül des taktisch unterstützten Programmierens

29. Mai 1997

Diplomarbeit

angefertigt von	Gerd Stolpmann
betreut durch	Dr. habil. Christoph Kreitz
am	Fachbereich Informatik
der	Technischen Hochschule Darmstadt

Diplomarbeit angefertigt von

Gerd Stolpmann Viktoriastr. 100 64293 Darmstadt
stolpman@intellektik.informatik.th-darmstadt.de
stolpman@rbg.informatik.th-darmstadt.de
Matrikelnr. 815376

betreut durch Dr. habil. Christoph Kreitz am Fachbereich Informatik der Technischen Hochschule Darmstadt:

Schematische Konstruktion von Globalsuchalgorithmen

Die Anwendung der intuitionistischen Typentheorie als Basiskalkül des taktisch unterstützten Programmierens

29. Mai 1997

Erklärung

Hiermit erkläre ich an Eides statt, daß die vorliegende Arbeit von mir eigenständig und nur mit den angegebenen Hilfsmitteln angefertigt wurde.

Darmstadt, den 29. Mai 1997

In Erinnerung an die Liebe, die mir Thomas die ganze Zeit über entgegengebracht hat

Inhaltsverzeichnis

Einleitung

Was in dieser Arbeit unter Globalsuche firmiert, ist, was die Sache angeht, eigentlich ein alter Hut. Algorithmen, die systematisch Lösungen probieren, sind vor allem unter der Bezeichnung „Backtracking“ bekannt geworden. Die Globalsuche wird hier jedoch auf andere Weise thematisiert, als dies traditionell geschieht, nämlich als strenge Theorie, die sogar bis in alle Einzelheiten formalisierbar ist und als solche in dem Sinne implementierbar, daß es ein System gibt, das die Anwendung dieser Theorie unterstützt. Bevor wir diesen Ansatz näher charakterisieren, sind jedoch noch einige Anknüpfungspunkte zu klären, die diesen Ansatz allgemein in die Landschaft der Programmierung und, spezieller, in diejenige der Programmsynthese einordnen.

Es geht ganz allgemein darum, Programme herzustellen. Wenn wir uns die Praxis des Programmierens anschauen, stellen wir fest, daß der Bereich der Herstellung eigentlich nicht isoliert betrachtet werden kann, da an ihn Anforderungen gestellt werden und seine Folgen getragen werden müssen. Als Beispiel sei hier der *Zweck* eines Programms genannt. Er muß deutlich unterschieden werden von der Spezifikation des Programms, die zwar aus dem Zweck destilliert wurde aber nun reine Eigenschaft des Programms ist. Der Zweck ist demgegenüber eher die Erwartungshaltung, daß das Programm die Rolle, die ihm zugeschrieben wird, auch erfüllen kann. Die Ebene, auf der der Zweck gesetzt wird, ist im Grunde nicht einzugrenzen, da Zwecke immer weitere Kreise ziehen können. Eng gezogen, könnte etwa der Zweck sein, daß das reale Programm in seiner Umgebung funktioniert; wir könnten aber auch als Zweck festlegen, daß es Akzeptanz erwirbt oder daß es mein Ansehen als Produzent steigert oder daß es Käufer findet. Der Zweck wird von einem Subjekt gesetzt, stellt aber eine Beziehung zum Programm, zum Objekt her. Wenn man Programme erstellt, hat man immer mit Zwecken zu tun, allein schon deshalb, weil diese in der Regel von jemand anderem gesetzt werden. Dennoch ist es so, daß sich die Informatik in diesem Bereich als reine Ingenieurwissenschaft gibt, die alles, was jenseits des Objekts passiert, ignoriert, ja dafür noch nicht einmal Begriffe hat. Dem Zweck wird einfach stillschweigend zugestimmt.

Auch in dieser Arbeit werden Programme fast nur als Objekte behandelt. Dies liegt zunächst daran, daß aktuelle Fragestellungen in der Informatik sich rein objektiv geben (auch wenn sie es häufig nicht sind, beispielsweise von einem Geldgeber beeinflußt oder von einem öffentlichen Interesse gelenkt). Das hat allerdings seine große Berechtigung dadurch, daß wir erst durch die Kenntnis der Eigenarten der Objekte das Grundlagenwissen erwerben, wie wir am besten mit ihnen umgehen können oder wie wir sie in unser soziales Gefüge einbauen sollten. Wir können aber die Frage, wie sich Subjekte zu den Forschungsergebnissen verhalten, nur solange ignorieren, bis wir diese Ergebnisse aus unserem Verfügungskreis entlassen. Ein schlechtes Beispiel war in der letzten Zeit die Diskussion der Verschlüsselung im Zusammenhang mit dem Ausbau der Computernetze. Informatiker und Mathematiker haben zwar das Wissen gehabt, wie und warum Verschlüsselungsverfahren funktionieren, es aber sträflich vernachlässigt zu untersuchen, wie diese Verfahren sozialisiert werden. In der Öffentlichkeit wurden lange Zeit die krudesten Meinungen verbreitet, und von Regierungen bestellte Experten konnten sicher sein, daß Mangels öffentlicher Kompetenz ihre Autorität nicht hinterfragt werden würde. Nicht der Informatik, sondern engagierten Mitgliedern der „Netzgemeinde“ ist es zu verdanken, daß die Öffentlichkeit nun etwas aufgeklärter ist und sich getraut mitzureden.

Programme werden wir nicht im Sinne realer Software, sondern nur in Bezug auf abstrakte Berechenbarkeitsmodelle auffassen. Dies ist aber eher eine temporäre Selbstbegrenzung, die den zu untersuchenden Raum betretbarer machen soll. Letztlich wird angestrebt, daß die konstruierten Programme auch auf realen Maschinen ablaufen können. Die Analyse des Zwecks gewinnt dadurch eine gewisse Relevanz, weil man nun zumindest die Möglichkeit bedenken sollte, daß eine Entität oder eine Eigenschaft von einem Subjekt aufgefaßt werden kann und für dieses bedeutsam wird.

Seit Frege wird *Bedeutung* zugespitzt darauf, daß die Bedeutung eines Zeichens der (ggf. imaginäre) Gegenstand ist, für den das Zeichen steht. Die formale Logik hat diese Auffassung der Bedeutung tradiert ebenso wie die Informatik. Es ist allerdings zu fragen, ob diese enge Auffassung alleine ausreicht, um zu verstehen, was ein Zeichen ausmacht. In dieser Arbeit werden wir es in extremer Weise mit der Frege'schen Bedeutung zu tun haben, da ein Kalkül als formale Basis verwendet wird, und die Semantik eines solchen Kalküls nur auf diese Weise definiert werden kann. Audrücke eines solchen Kalküls kann man schlecht so stehen lassen wie sie sind; es wäre keinem Leser zuzumuten, sich in die Feinheiten des Zeichensystems eines Kalküls einzuarbeiten und gleichzeitig zu erwarten, sich einen komplexen Ausdruck des Kalküls als Abstraktion zu rekonstruieren. Es wird daher versucht, immer zunächst ein *Konzept* zu etablieren, das zwar in einer strengen Fachsprache definiert und erläutert wird, aber nicht der extremen Auffassung der Bedeutung ausgeliefert ist, bevor in einem zweiten Schritt dieses Konzept auf der Ebene des Kalküls nachgebildet wird. Beide Ebenen, die des Konzepts und die des Kalküls, haben je ihr eigenes Recht, und es wird nicht angestrebt, daß die eine Ebene die andere ersetzt.

Die hauptsächliche Eigenschaft der Programme, um die es hier geht, ist die der formalen Korrektheit. Das Verhältnis zwischen Programm und Korrektheit kann man hinsichtlich folgender Fragestellungen untersuchen:

- *Wodurch ist es gerechtfertigt, einem Programm die Eigenschaft der Korrektheit zuzuordnen?* Der Kalkül definiert, wie ein Beweis zu erbringen ist, der diese Zuordnung rechtfertigt. Dabei muß man natürlich den Kalkül als solchen akzeptieren.

  Der Kalkülbeweis ist wohl die artifiziellste Form des Beweises, die jemals erfunden worden ist. Er besteht nur noch aus schematischen Regeln; niemand scheint darin mehr involviert zu sein, sogar eine Maschine kann den Beweis führen. Es sieht so aus, daß der Kalkülbeweis die wissenschaftliche Neutralität und Objektivität in Person ist. Aber der Kalkülbeweis ist nur das Produkt einer sehr langen Entwicklung, und es lohnt sich vielleicht zu fragen, welche Anteile des Beweisens in die Form des Kalküls aufgehoben wurden.

  Dem Beweis geht der Zweifel voraus. Der Beweis ist allein dadurch motiviert, daß Menschen nicht die gleiche Einsichts- und Urteilsfähigkeit haben, und man also einem anderen Menschen erklären muß, wie man zu seinem Urteil gekommen ist. Der Kalkülbeweis läßt nun nicht mehr erkennen, daß gerade unterschiedliche Individualität den Beweis motiviert, sondern definiert im Gegenteil einen „Konsens“ auf der Ebene atomarer Urteilsformen. Dies heißt, man kann sich auf den Kalkül als Richtschnur für die Wahrheit einigen und alle Streitpunkte dadurch entscheiden, daß ein Beweis gegeben wird oder nicht. Wie man sich freilich einigt, liegt außerhalb des Kalküls.

  Es ist wichtig, daß man den Beweis als eine dialogische Ausdrucksform ansehen kann: jemand rechtfertigt sein Urteil gegenüber einem anderen. Für einen maschninell produzierten Beweis heißt dies, daß die Maschine gegenüber dem Menschen ihr Urteil rechtfertigt. Es ist noch zu überlgen, was dies für den Umgang mit maschinell produzierten Beweisen heißt. Ist der Benutzer z.B. gezwungen, den Beweis anzunehmen, nur weil der Kalkül korrekt anwendet wird, oder gibt es da noch andere Kriterien?

- *Wie können wir aufgrund einer Spezifikation ein Programm konstruieren?* Die Spezifikation beschreibt ja bloß die Wirkung eines Programms, ohne (vollständig) anzugeben, wie dies berechnet werden kann. Ein Programm zu einer Spezifikation zu erfinden, ist ein Akt, der in der Philosophie gemeinhin als *Spontaneität* firmiert. Dies ist das Vermögen, Daten zu einer formalen Einheit zusammenzuschließen; oder anders gesagt, dasjenige, was die Daten beschreiben, auf einer anderen Ebene als neue Einheit zu erfinden oder wiederzuerkennen. Hier ist die Spezifikation das Datum, das „Gegebene“, und auf der Ebene der Programme soll nun eine Entsprechung gefunden werden. Die Philosophie hat im Begriff der Spontaneität immer schon mitgedacht, daß es keinen direkten Weg gibt, wie die neue Einheit aus den Daten abgeleitet werden kann; es ist daher überraschend und faszinierend, daß eine Spontaneität glücken kann. Aus der theoretischen Informatik wissen wir, daß es keinen Algorithmus gibt, der beliebige Synthesen durchführen kann; insofern finden wir uns in einer Situation, für die die klassische philosophische Analyse zutreffend ist.

Indem wir solch einen „altehrwürdigen“ Begriff zur Orientierung heranziehen, können wir insbesondere vermeiden, einen irreführenden Anspruch für Synthesen zu setzen. Letztlich ist die Spontaneität nämlich nicht zu unterlaufen, dies resultiert aus ihrem Status als transzendentale Bedingung in einem Erkenntnisprozeß[1]; einem Programm spontane Fähigkeiten zu attestieren, hieße, seine eigene Rolle als Urteilender über dieses Programm zu verkennen. Wir befinden uns in der merkwürdigen Situation, uns ein Ziel zu stecken, das sich, was immer wir auch unternehmen, es zu erreichen, als zu weit herausstellt. Oder anders gesagt, ein Programm, das spezielle Synthesen berechnen kann, ist nur deshalb faszinierend, weil wir dies im Horizont des unlösbaren allgemeinen Syntheseproblems aufzufassen gewohnt sind.

- *Welche Programme können in dem Kalkül spezifiziert und welche können ausgedrückt werden?* Die Sprachen, die verwendet werden, sind nicht universell, d.h. nicht alles kann ausgedrückt werden. Der Grund dafür, von vorneherein nur eine eingeschränkte Ausdruckskraft anzustreben, liegt darin, daß damit einige Probleme entscheidbar werden. In der hier verwendeten Typentheorie kann man beispielsweise nur Programme schreiben, die für alle zulässigen Eingaben terminieren, also total sind. Dadurch sind Algorithmen möglich, die das extensionale Verhalten von Programmen an einzelnen Punkten entscheiden. Erst durch diese Voraussetzung ist es sinnvoll, Aussagen über das extensionale Verhalten überhaupt im Kalkül zuzulassen. – Ausdruckskraft und die Zahl der entscheidbaren Probleme verhalten sich umgekehrt proportional zueinander. In der Regel muß man den Sprachbereich erst so weit einschränken, daß die für die Anwendung relevanten Probleme entscheidbar werden. Natürlich ist es nun fraglich, ob auch noch genügend ausgedrückt werden kann; dies ist eine sehr wichtige Untersuchung, wenn man einen Kalkül benutzen möchte.

Die Methode der Globalsuche wurde zuerst in dem KIDS-System implementiert [smith91]. KIDS benutzt jedoch keinen strengen Kalkül als Grundlage, sondern eine Sprache, in der beliebige Annahmen über die Wirkung und Zusammenwirkung der Operationen gemacht werden können, ohne daß eine Rechtfertigung erfolgen muß. Der Wert der von KIDS produzierten Aussagen hängt sehr davon ab, ob diese Annahmen vernünftig sind oder nicht. Desweiteren sind die Schlüsse, die KIDS atomar

[1]Dies ist der kritische Teil der Philosophie Kants, der, auch wenn heute z.B. nicht mehr von einem einheitlichen Bewußtseinsstrom ausgegangen wird, die Zeit überdauert hat und immer noch als paradigmatisch gelten kann. „Wollen wir die *Rezeptivität* unseres Gemüts, Vorstellungen zu empfangen, sofern es auf irgendeine Weise affiziert wird, *Sinnlichkeit* nennen, so ist dagegen das Vermögen, Vorstellungen selbst hervorzubringen, oder die *Spontaneität* des Erkenntnisses, der *Verstand*.“ ([kant90], S. 95) Wie erkennt nun der Verstand? Kant denkt sich dies bereits so, daß man einer Regel, oder modern ausgedrückt einem Algorithmus folgt: „Der Begriff vom Hunde bedeutet eine Regel, nach welcher meine Einbildungskraft die Gestalt einer vierfüßigen Tieres allgemein verzeichnen kann, ohne auf irgendeine einzige besondere Gestalt, die mir die Erfahrung darbietet, oder auch ein jedes mögliche Bild, was ich in concreto darstellen kann, eingeschränkt zu sein.“ (S. 200) Kein Bild eines Hundes kann die Regel oder das Schema des Hundes ersetzen. Die Regel erlaubt aber, den Begriff des Hundes *spontan* zu setzen und mittels der Regel die Daten als Einheit, als Begriff zu denken; es ist überhaupt nicht zwingend, die Daten gerade als Hund aufzufassen.

durchführen kann, relativ großschrittig. Dies hat natürlich den Vorteil, daß nur wenige Schlüsse in folge gezogen werden müssen, um eine praktisch verwendbare Aussage zu beweisen. Nachteilig ist aber, daß das Risiko von Implementierungsfehlern bei großen Schritten wesentlich höher ist als bei kleinen. Wir sehen, daß das Design von KIDS nicht ausgerichtet ist auf eine höchstmögliche Absicherung der Korrektheit, sondern darauf, die höheren Schlußtechniken umstandslos verfügbar zu machen. Der Wert von KIDS liegt auch gerade darin, beispielhaft zu zeigen, wie höhere Schlußse genutzt werden können, um Synthese nach der Methode der Algorithmenschemata zu ermöglichen.

In dieser Arbeit wird als Basis-System dagegen NuPRL verwendet [constable et al.86]. NuPRL hat zunächst keinen bestimmten Anwendungszweck, sondern ist eine allgemeine Plattform, die den Kalkül der intuitionistischen Typentheorie implementiert, welcher nun von Anwendungen genutzt werden kann. Es wird gerade von der anderen Seite angesetzt als bei KIDS, indem eine sichere Grundlage geschaffen wird, auf die eine Anwendung aufsetzen kann. Mehr noch: NuPRL *verpflichtet* die Anwendung auf die Einhaltung der Kalkülregeln; jeder Verstoß wird entdeckt und führt zu einer Fehlermeldung[2]. NuPRL ist nicht nur ein reiner „proof checker", vielmehr wird der Datentyp des Beweises eingeführt, in dem die elementare Beweisüberprüfung eingebaut ist, und der nun nach außen hin auf mehrere Weisen verwendet werden kann. Zum einen kann man Beweise automatisch erzeugen, indem Beweisobjekte von einem Algorithmus zusammengesetzt werden, was auch als *Taktik* bezeichnet wird. NuPRL gibt die Programmiersprache vor, in der diese Algorithmen verfaßt werden müssen, da nur in dieser Zugriffe auf Beweisobjekte möglich sind; es handelt sich um ML („meta language"), der Meta-Sprache relativ zu der Sprache des Kalküls. Da die Überprüfung, ob die konstruierten Beweise den Kalkülregeln entsprechen, bereits innerhalb der Beweisobjekte geschieht, müssen Taktiken nicht dafür Sorge tragen, sondern sind davon entlastet. Algorithmische Fehler führen keinesfalls zu falschen Beweisen, sondern höchstens dazu, daß der Beweis unvollständig bleibt oder der Algorithmus vorzeitig abgebrochen wird. Außer durch Taktiken können die Beweisobjekte aber auch interaktiv bearbeitet werden; NuPRL wird insofern auch als Beweis-Editor bezeichnet. Es ist möglich, aus dem Beweiseditor Taktiken zu starten, die damit die Rolle einer automatischen Unterstützung für die an sich manuelle Beweisführung spielen. (Umgekehrt ist es nicht möglich, den Beweiseditor aus einer Taktik aufzurufen; dies ist eine Design-Schwäche, die hoffentlich bald überwunden wird. NuPRL sollte sich dahingehend öffnen, daß spezifische Applikationen sich ihre eigene Oberfläche definieren können.) Die meisten Taktiken benutzen daher auch Heuristiken, die für eine bestimmte Beweisaufgabe häufig funktionieren – jedoch ohne eine Garantie dafür abzugeben. In einer interaktiven Umgebung wie dem Beweis-Editor kann der Benutzer dann entscheiden, wie er damit umgehen soll, wenn die Taktik einmal scheitern sollte. Häufig reicht es etwa, gewisse Vorverarbeitungsschritte durchzuführen, damit die Taktik doch noch zum Erfolg führt. Anders gesagt: Durch eine Analyse der Ursache für das Scheitern kann die Taktik flexibel an den konkreten Fall angepaßt werden.

Obwohl grundsätzlich Taktiken als Heuristiken konzipiert werden können und durch die interaktiven Kooperationsmöglichkeiten fallweise auftretende Schwächen ausgebügelt werden können, wird man sicherlich nicht ohne Taktiken mit explizit definiertem Aufgabenbereich auskommen. Wir meinen damit Taktiken, bei denen man an der äußeren Form der Aufgabenstellung erkennen kann, ob sie die Aufgabe entscheiden können. Der Vorteil solcher Taktiken ist, daß sie sich auf sicherem Gebiet bewegen, und man vorhersagen kann, ob sie funktionieren werden. In NuPRL wurde etwa eine Entscheidungsprozedur integriert, die bestimmte Probleme mit ganzen Zahlen behandeln kann. Allerdings ist diese Prozedur keine Taktik, vielmehr wurde sie ob des elementaren Charakters der Probleme als zusätzliche Kalkülregel implementiert. Dies bietet sich allerdings nur bei wirklich elementaren Problemen an, da der Kalkülkern überschaubar bleiben sollte. In dieser Arbeit wurde an ein anderer Bereich angegangen: Es wird der Ansatz für eine Taktik vorgestellt, die bestimmte Typisierungsprobleme entscheiden

[2]In der aktuellen Version 4 von NuPRL gibt es freilich einige Sicherheitslücken; wir gehen zunächst aber von der Idealvorstellung aus.

kann. Zahlreiche praktisch vorkommende Typisierungsbeweise können damit automatisiert werden; allerdings gibt es genauso zahlreiche praktisch relevante Typisierungen, die nur mit Hilfestellung oder gar nicht erbracht werden können. Dies liegt eher daran, daß Typisierungsbeweise Teilaufgaben enthalten oder sogar mit Teilaufgaben regelrecht verzahnt sein können, die gewöhnliche Propositionen sind und schon von daher von der Taktik nicht bearbeitet werden können, als daran, daß in eine für Entscheidungsverfahren tödliche unbegrenzte Suche abgeglitten wird.

Mittels des Taktik-Konzepts ist es in NuPRL also dennoch möglich, große Beweisschritte zu machen, obwohl die automaren Schritte sehr klein sind. Freilich setzt dies voraus, daß es einen Algorithmus (eben die Taktik) gibt, der den großen Schritt auf mehrere kleine abbildet. Man kann dadurch grundsätzlich ein ähnliches Niveau erreichen, wie es im KIDS-System benutzt wird. Allerdings ist KIDS unbelastet von den formalen Details eines Kalküls; in NuPRL ist es nicht möglich, ein Detail als unbedeutend für das Niveau der gerade verfolgten Argumentation abzutun. Es scheint so, also ab sich das gewünschte Verhältnis genau umdreht: Der Basis-Kalkül bestimmt die höheren Konzepte, während eigentlich angestrebt wird, daß sich jedes vernünftige Konzept durch den Kalkül realisieren läßt. Beispielsweise muß man in NuPRL zwischen einer propositionalen Form und einer berechenbaren Form der Gleichheit streng unterscheiden, obwohl dieser Unterschied in den meisten Argumentationen unbedeutend ist. Er muß dennoch durch alle Schlüsse „hindurchgeschleppt“ werden, bis er an nur wenigen Stellen tatsächlich relevant wird. Alle höheren Konzepte müssen diesen Unterschied ebenfalls beachten. Eine Kleinigkeit, die nur hie und da zum Zuge kommt, kann so leicht das ganze Design von Theorien bestimmen. Es ist hier noch viel Arbeit zu leisten, um zu untersuchen, ob es nicht doch bessere Strukturierungsmöglichkeiten gibt.

NuPRL ist ein interaktives System, in dem die Rechtfertigung von Aussagen einen anderen Stellenwert gewinnt als in vollautomatischen Systemen. Der Benutzer kann sich jeden von der Maschine produzierten Beweis ansehen und für sich entscheiden, ob er der Argumentation zustimmt. Aber noch weiter: Normalerweise ist der Benutzer ja an der Entwicklung eines Beweises beteiligt und bestimmt wenigstens die grobe Richtung der Beweisführung. Es findet deshalb eine permanente Überprüfung statt, ob der Benutzer der Maschine noch folgen kann; auch der Weg des Beweises wird vom Benutzer geprüft. Es kann dadurch etwa nicht (oder kaum) passieren, daß der Benutzer zwar die bewiesene Aussage akzeptiert, weil sie ihm vernünftig vorkommt und der Kalkülbeweis diese Auffassung ja unterstützt, aber der Benutzer eine Zwischenaussage während des Beweises, den er sich nicht anschaut, ablehnen würde. In NuPRL kommt die intersubjektive Dimension des Beweises, die wir Rechtfertigung genannt haben, besser zum Zuge als in einem vollautomatischen System.

Beweise können in NuPRL auch bezüglich des Niveaus strukturiert werden. Dies heißt zunächst nur, daß man Taktiken bereitstellen kann, die in der Lage sind, Beweisschritte auf dem gewählten Niveau dadurch zu simulieren, daß eine Abbildung auf kleinere Schritte erfolgt. Auf der Seite der Repräsentation dieser Technik ist man frei, entweder die Abbildung sofort aufzulösen, so daß der Benutzer sofort die von der Taktik erzeugten Schritte sieht, oder diese Abbildung abstrakt zu lassen. In letzterem Fall sieht man nur, welche Taktik aufgerufen wird; die Schritte zur Simulation werden verborgen. Die Beweise erhalten damit eine Tiefenstruktur, die man Stufe für Stufe auflösen kann. (Der Beweiseditor erlaubt im Augenblick nicht, die einzelnen Stufen als solche sichtbar zu machen, man sieht immer nur die äußerste Stufe, und um an die inneren zu gelangen, muß man erst die abstrakten Taktikaufrufe auflösen.)

Bei der Implementierung der Globalsuche wird es nun darum gehen, zum einen die benötigten Konzepte und Theorien aufzubauen, zum anderen die taktischen Mittel bereitzustellen, damit diese Konzepte auf dem adäquaten Niveau angewandt werden können. Diese beiden Bereiche, Theorieaufbau und -anwendung, verlangen je unterschiedliche Methoden. Beim Aufbau geht es vor allem darum, die benötigten mathematischen Konzepte in der Sprache der Typentheorie auszudrücken, mittels dieser Konzepte Theoreme aufzustellen und diese schließlich zu beweisen. Es ist zu überlegen, welche inneren Ordnungsprinzipien für den strukturierten Aufbau erforderlich oder sinnvoll sind, um daraus für die spezifische Anwendung ein *Theoriedesign* zu entwickeln. Beim Beweis der Theoreme wird man wahrscheinlich die Methoden der Theorieanwendung nutzen können; es ist daher ratsam, diesen Teil der Theorieentwicklung so lange zu verschieben, bis die Theorieanwendung taktisch ausreichend unterstützt wird. Bei dieser Theorieanwendung geht es nun darum, Schlüsse dadurch zu ziehen, daß auf die Existenz eines Theorems verwiesen wird.

Um eine Theorie aufzubauen, die für die Globalsuche geeignet ist, mußte zunächst eine ganze Menge Vorarbeit geleistet werden. So benötigt man eine einigermaßen realistische Programmiersprache, um die für die Globalsuche erforderlichen Programme ausdrücken zu können. Desweiteren wird Wissen über diese Sprache gebraucht, d.h. es muß möglich sein, auf adäquatem Niveau über Ausdrücke der Sprache zu schließen. Zahlreiche Überlegungen hierzu findet man in meiner Studienarbeit [stolpmann96], die hier in Kapitel 2 aufgenommen und fortgeführt werden. Nun erst kann die Globalsuche in diese formale Sprache übersetzt werden, d.h. die angesprochene Programmiersprache wird in den Theoremen für die Globalsuche benutzt. In Kapitel 4 stellen wir die ausgearbeitete Theorie der Globalsuche in der Sprache des zugrunde gelegten Kalküls dar. Auf Beweise auf der Ebene des Kalküls mußte verzichtet werden, weil im Augenblick die taktische Unterstützung noch nicht ausreichend ist, um derart komplexe Beweise zu führen. Immerhin ergibt sich aber die Erkenntnis, daß die Theorie mittels des Kalküls überhaupt ausgedrückt werden kann.

Um Theoreme anzuwenden, stellt NuPRL bereits eine Reihe von Taktiken zur Verfügung. Es hat sich allerdings herausgestellt, daß die Taktiken zwar beispielhaft zeigen, durch welche Konstruktion von elementaren Kalkülregeln die Theoremanwendung simuliert werden kann, aber dabei häufig auf zusätzliches Wissen angewiesen sind, das erst gewonnen werden muß. NuPRL sieht hierfür bestimmte Heuristiken vor. Beispielsweise benötigt man des öfteren den Typ eines Ausdrucks, welchen man sich erst erschließen muß. Die NuPRL-Heuristik gerät dabei recht schnell an ihre Grenzen, gerade programmiersprachliche Ausdrücke, die ja oft Operatoren enthalten, die die Typen nicht identisch lassen, sind problematisch. Kapitel 2 beschäftigt sich daher besonders mit der Gewinnung von Typinformationen für Programmterme. In Kapitel 3 wird dann als die wichtigste Methode der Theoremanwendung die Termersetzung besprochen. NuPRL enthält zwar bereits ein Termersetzungspaket, allerdings gänzlich ohne die Fähigkeit, Typinformationen zu nutzen. Es werden daher Methoden vorgestellt, wie mittels dieser Informationen Termersetzungen durchgeführt werden können, die ansonsten scheitern würden.

Die Globalsuche selbst tritt ob dieser vielen Fragen, die bei der Implementierung auftauchen, beinahe in den Hintergrund. Für diese Arbeit ist sie auch nur ein Beispiel; es hätte im Grunde auch die konzeptuellen Probleme und die Implementierung einer beliebigen anderen Theorie untersucht werden können. Natürlich wäre dann ein anderer, auf die Eigenheiten der Anwendung bezogener Weg eingeschlagen worden. Die Idee der Algorithmenschemata, zu denen ja die Globalsuche zählt, hat aber eine gewisse Faszination ausgeübt, insofern sich hier möglicherweise ein neues Paradigma der Programmierung ankündigt. Das Bilden von Inferenzen über Programme wird nicht als intelligentes, sondern als schematisches Verfahren angesehen, allerdings als Schema ausgesuchter Abstraktionsqualität. Das Paradox der Künstlichen Intelligenz, die ja einerseits der „natürlichen“ Intelligenz nacheifert, andererseits aber schon immer besser sein will als die Natur, tritt erst gar nicht auf. Die KI wollte immer besondere Fähigkeiten, die als intelligent eingestuft wurden, adaptieren, etwa logisches Denken, zugleich aber besser sein als das Vorbild, etwa was Fehlschlüsse, Verfügbarkeit oder Effizienz betrifft. Man backt nun kleinere Brötchen, in der Hoffnung, daß sie endlich gar werden; die KI hat sicherlich sehr interessante Methoden hervorgebracht, allein die Ziele waren zu hochgesteckt. Der Ansatz der Algorithmenschemata ist nun sehr nüchtern und verspricht nicht mehr, als aufgrund einer ausgefeilten Klassifizierung von Algorithmen diejenigen Probleme lösen zu können, die in eine der Klassen fallen. Mensch und Maschine werden hier nicht von vorneherein als Konkurrenten gedacht, sondern als Experten auf je ihrem Gebiet: Die Maschine kann gut symbolische Ableitungen dargestellen und in begrenztem Umfang auch finden;

der Mensch kann seine Erfahrung im Programmieren in die Bildung von guten Algorithmenschemata umsetzen, sowie seine Intuition nutzen, um die grobe Richtung der Ableitungen zu steuern. Erst die Zusammenarbeit könnte vielleicht ermöglichen, was bisher undenkbar war, nämlich in größerem Stile verifizierte Programme zu schreiben. Es ist klar, daß auch dies ein hochgestecktes Ziel ist, das auch seine inneren Paradoxien hat: Wie vereinbart man zum Beispiel Kreativität und formale Korrektheit? Nun könnte sich aber gerade in der Kooperation zwischen Mensch und Maschine die Vermittlung der auseinanderstrebenden Momente abzeichnen, insofern für die Kreativität der Mensch zuständig ist, während die Details der formalen Korrektheit von der Maschine erledigt werden. Damit dies realisierbar ist, käme es vor allem darauf an, daß jede Seite die andere freiläßt, also etwa Darstellungen gefunden werden, die von der Maschine mit allen notwendigen Formalia ausgestattet werden können, aber auch für die menschliche Auffassung so aussagekräftig bleiben, daß sie kreative Problemlösungen anstoßen können.

Das Projekt, Algorithmenschemata und insbesondere die Globalsuche innerhalb der Typentheorie auszudrücken, wurde erstmals formuliert in der Habilitation [kreitz92]. Dort wird theoretisch auseinandergelegt, inwiefern die Typentheorie für Synthese geeignet ist; insbesondere wird aber die Möglichkeit erkundet, Synthese selbst zu ihrem eigenen Objekt zu machen[3]. Nun ist es allerdings nicht ausreichend, nur die Möglichkeit einer Form von Synthese aufzuzeigen; eine solche Analyse wahrt nämlich immer Distanz zu ihrem Gegenstand. Es könnte z.B. sein, daß eine wichtige Bedingung übersehen wurde oder daß an einer Stelle ungerechtfertigt simplifiziert wurde; Möglichkeitsuntersuchungen müssen, weil sie Zukunft antizipieren wollen, mit einer gewissen Unsicherheit rechnen. Erst eine Realisierung kann beweisen, daß ein Weg gangbar ist.

Diese Arbeit nimmt sich vor, die Globalsuche auf der NuPRL-Plattform zu implementieren. Ziel ist dabei aber nicht primär ein Software-Produkt gewesen, und es ist auch nicht so gekommen, daß ein Produkt in dem Sinne, daß alle ursprünglich gesetzten Ansprüche realisiert werden konnte, herausgekommen ist. Ziel war es eher, einige Meter auf das vorgestellte Produkt hinzuzulaufen, um auf diesem Weg Konzepte und ihre Realisierungsmöglichkeiten zu studieren. Ein solches Experiment ist etwas ganz anderes, als Software herzustellen, weil man immer damit rechnen muß, daß der Boden, auf dem man sich bewegt, mit einem Mal nicht mehr tragfähig ist. Man muß dann zurückgehen und die Grundlagen aufs Neue erkunden.

Zu dem NuPRL-System gibt es keine ausgezeichnete Referenz, weil auch dieses System sich ständig im Umbruch befindet. Man muß daher mit einem eilig zusammengestrickten Handbuch [jackson94b] und einer Kurzbeschreibung der Metasprache ML [jackson94a] auskommen, um das System zu verstehen. Es ist leider sehr viel Herumprobieren nötig, um manchen Phänomenen auf die Schliche zu kommen. Es wird nicht vorausgesetzt, daß der Leser dieser Arbeit NuPRL kennt; es wäre aber von Vorteil.

Für den zugrunde gelegten Kalkül, die Typentheorie von Martin-Löf, sieht die Verfügbarkeit von Literatur ein ganzes Stück besser aus. Zum einen gibt es die Originalarbeit [martin-löf84], zum anderen eine Einführung der NuPRL-Macher [constable et al.86]. Dem Autor dieser Arbeit ist der Einstieg durch ein exzellentes Skript leicht gemacht worden [kreitz94], das hoffentlich bald einer größeren Leserschaft zur Verfügung steht.

[3] Als philosophische Begleituntersuchung wäre hier hochinteressant, den Deutschen Idealismus zu studieren, der ja versucht hat, die Welt dadurch zu erklären, daß das Ich durch fortgesetzte reflexive Prozesse sich aus sich selbst entfaltet. Man könnte so etwa erfahren, welche Paradoxien man dabei zu erwarten hat.

Kapitel 1

Das Schema für die Globalsuche

In dieser Arbeit werden wir es mit einem bestimmten Typ von Algorithmen zu tun haben, den wir als *Globalsuche* bezeichnen. Es soll eine Methode vorgestellt werden, wie man grundsätzlich solche Algorithmen erzeugen und rechtfertigen kann. Diese Methode wird in den formalen Rahmen der Typentheorie eingefaßt werden, wodurch wir sie für die Implementierung im NuPRL-System vorbereiten. Letztendlich streben wir eine relativ weitgehende automatische Unterstützung zur Durchführung dieser speziellen Methode an. Wir zielen dabei freilich nicht auf ein allgemeines Prinzip zur Synthese, sondern wollen lediglich ein Verfahren zur Verfügung stellen, daß für Probleme adäquat ist, die mittels der Globalsuch-Rekursion gelöst werden können.

In den Lehrbüchern der Programmierung wird versucht, dem Lernenden dadurch einen besseren Überblick über die möglichen Algorithmen zu geben, indem Algorithmen, die miteinander verwandt sind, als Typus aufgefaßt werden. Der Vorteil davon ist, daß man lediglich einen repräsentativen Algorithmus lernen muß, um die Algorithmen eines solchen Typs im Prinzip verstanden zu haben. Liegt nun ein konkretes Problem vor, das algorithmisch gelöst werden soll, muß man nun nicht gleich eine neue Art der Rekursion erfinden; es reicht, sich einer der relativ wenigen Typen von Rekursion zu erinnern, die andere Informatiker bereits ausfindig gemacht haben. Dieser Vorgehensweise liegt die Abstraktion konkreter Algorithmen nach ausgesuchten Kriterien zu Grunde, die in erster Linie die jeweilige Funktionsweise bestimmen und damit Anlaß geben, die Algorithmen entsprechend zu ordnen. Dieses Wissen, das durch jahrzehntelange Arbeit von Informatikern angesammelt wurde, ist ausgesprochen wertvoll, weil die Erfahrungen, die mit Typisierungen von Algorithmen gemacht worden sind, immer wieder, z.B. beim Schreiben von Lehrbüchern, in die Typisierung zurückflossen; dieses Wissen weist dementsprechend eine hohe Dichte auf, was auf die Güte der Ordnugnskriterien zurückzuführen ist.

Lehrbücher sind nun für Menschen geschrieben worden; die dort gegebenen Darstellungen sind also vor allem geeignet, wenn Programmierer einen Algorithmus-Typ auf eine konkrete Aufgabenstellung übertragen möchten oder wenn neue Erfahrungen in dieses Wissen hineingebracht werden sollen. Die Darstellungen sind nicht direkt geeignet, wenn man die Übertragung maschinell unterstützen und ihre Korrektheit durch einen Kalkül verifizieren möchte. Das Wissen muß formalisiert, d.h. in eine andere Gestalt gebracht werden, die „kalkülgängig“ ist und die die Grundlage bildet, damit formale Methoden benutzbar sind. Statt mit Typisierungen haben wir es dann auch mit Klassen von Algorithmen zu tun, da nun die Grenzen der Klassen scharf gezogen sind. Durch die Ausgearbeitetheit formalisierter Algorithmenklassen sinkt freilich der Abstraktionsgrad, da der Formalisierung stets zusätzliche Annahmen geschuldet sind. Es bedarf daher Experimenten der Formalisierung, um herauszufinden, inwieweit relevante Algorithmen noch wiedergegeben werden können und wie hoch die Abstriche sind.

Eine scharfe Klassenbildung wird hier nun dadurch eingeführt, daß Algorithmen als Ausprägungen eines Schemas dargestellt werden; dies findet man bereits in den Lehrbüchern, die viele Teile eines Algorithmus, die man im konkreten Anwendungsfall ausprogrammieren müßte, weglassen oder nur

$$\begin{tikzcd}
\text{Problem } P_0 \arrow[r, "R\text{-reduzierbar auf}"] \arrow[d, dash, "\text{erfüllt von}"] & \text{Problem } P_{std} \arrow[d, dash, "\text{erfüllt von}"'] \\
\text{Algorithmus } f_0 & \text{Algorithmus } f_{std} \arrow[l, "R\text{-konstruierbar durch}"']
\end{tikzcd}$$

Abb. 1.1. *Allgemeine Reduktion*

durch einzelne Symbole repräsentieren. Das Schema beschreibt dabei ein Programmgerüst, das das grundsätzliche Ablaufprinzip festlegt, aber noch instanziiert und gegebenenfalls variiert werden muß, um ein konkretes Problem zu lösen. Die so zu Lehrzwecken gewonnene abstrakte Sicht auf Algorithmenklassen stellt freilich keine Anleitung dar, auf die sich ein Lernender blind verlassen könnte, um ein Programm zu konstruieren. Allerdings gelingt es immerhin, ein Konstruktionsprinzip freizulegen, das die Richtung der Problemlösung fixiert und das die Möglichkeiten weiterer Konstruktionsschritte stark reduziert. Desweiteren lassen sich an solche Klassen bestimmte Vorstellungen anknüpfen, die es dem Lernenden erlauben, sich die Funktionsweise solcher Algorithmen bildhaft zu machen.

Diese Beobachtungen lassen sich nun für die automatische Unterstützung der Programmkonstruktion nutzen. Falls es gelingt, formale Kriterien für die durch Schemata definierten Algorithmenklassen zu finden, ist eine Implementierung bestimmter Schemata nicht mehr fern. Attraktiv wird diese Vorgehensweise dadurch, daß ein Schema die Konstruktionsrichtung vorgibt und damit Anhaltspunkte liefert, wie der Algorithmus funktioniert, aber dabei genügend Flexibilität läßt, das Schema für viele Probleme zu verfeinern. Es ist gleichermaßen schlecht, wenn das Schema zu allgemein ist, weil dann der Weg bis zur Problemlösung zu lang ist, als auch, wenn es zu speziell ist, da dann zu wenige Algorithmen erzeugt werden könnten, als daß der immense Aufwand zur Verifikation des Schemas gerechtfertigt wäre. Schemata sind auch deshalb interessant, weil sie das tradierte Wissen der Informatiker aufnehmen und daher eine „natürliche" Bneutzerschnittstelle ermöglichen. Es wird damit einsichtig, welches Schema für welches Problem geeignet ist, außerdem wie ein Problem aufbereitet werden muß, damit ein Schema anwendbar ist, und es kann auch nachvollzogen werden, warum eine Lösung korrekt ist.

Wir beschäftigen uns nun ausschließlich mit dem Schema der Globalsuche, das Smith erstmalig formalisiert hat [smith87]. Zunächst ist aber erst zu definieren, was überhaupt ein Algorithmenschema ist, um dann die Globalsuche als ein solches Schema zu etablieren. Es zeigt sich, das sei schon vorweggenommen, daß aufgrund des Flexibilitätsgebots die geradlinige Schematisierung unpraktibel ist. Eine gleichermaßen richtungsweisende wie flexible Algorithmenkonstruktion wird allerdings durch den Einbau der Technik der *Spezialisierung* ermöglicht.

1.1 Algorithmenschemata

Ein Prinzip zur Algorithmenkonstruktion ist die *allgemeine Reduktion*, die einen bereits vorhandenen Algorithmus so benutzt, daß er auch für eine neue Problemstellung eingesetzt werden kann. Das neue Problem P_0 kann auf das Standardproblem P_{std}, dessen Algorithmus f_{std} bekannt ist und z.B. aus einer Wissensbank abgerufen werden kann, reduziert werden, wenn es ein Reduktionsschema R gibt, das eine Möglichkeit festlegt, wie man den neuen Algorithmus aus dem vorhandenen konstruiert. Dabei unterscheiden wir zwischen dem reinen Zusammensetzen des gesuchten Algorithmus f_0 unter Verwendung von f_{std} und der Rechtfertigung dieser Zusammensetzung. Letztere muß zeigen, daß die Anwendung der Reduktion tatsächlich zu einem Algorithmus führt, der das Problem P_0 löst. Es

$$
\begin{tikzcd}[column sep=huge, row sep=huge]
\text{Problem } P_0 \atop {\text{Eingabebedingung } I_0 \atop \text{Ausgabebedingung } O_0} \arrow[r, "\text{direkt reduzierbar, falls } I_0 \Rightarrow I_{std} \text{ und } O_0 \Leftarrow O_{std}"] \arrow[d, dash, "\text{erfüllt von}"] & \text{Problem } P_{std} \atop {\text{Eingabebedingung } I_{std} \atop \text{Ausgabebedingung } O_{std}} \arrow[d, dash, "\text{erfüllt von}"] \\
\text{Algorithmus } f_0 & \text{Algorithmus } f_{std} \arrow[l, "\text{direkt konstruierbar: } f_0 = f_{std}"']
\end{tikzcd}
$$

Abb. 1.2. *Direkte Reduktion*

ist der Schluß zu ziehen, daß aus der Annahme, daß f_{std} eine Lösung von P_{std} ist, folgt, daß der durch R gewonnene Algorithmus f_0 eben P_0 löst (Abbildung 1.1). Für konkretes R können wir ein *Reduktionsaxiom* angeben, das den Ansatz für die Rechtfertigung ausdrückt und das man erfüllen muß.

Die einfachste Weise der Reduktion ist die *direkte*, bei der das Programmschema praktisch die Identität ist: $f_0 = f_{std}$ (Abbildung 1.2). Diese Trivialform der Zusammensetzung kann natürlich nur in äußert günstigen Fällen gerechtfertigt werden. Bei der Formulierung des Axioms gehen wir davon aus, daß sowohl P_0 als auch P_{std} durch ein Paar aus Eingabe- und Ausgabebedingung angegeben werden. Es besagt nun, daß die Eingabebedingung von P_0 stärker und die Ausgabebedingung schwächer sein muß als die jeweiligen Bedingungen von P_{std}, damit die direkte Reduktion angewandt werden darf. Das Axiom muß für jeden konkreten Fall von Reduktion bewiesen werden. Ein zugrunde liegendes Reduktionstheorem garantiert dann, daß alle Fälle, die das Axiom erfüllen, korrekt reduziert werden.

Die direkte Reduktion kann übrigens verallgemeinert werden, indem zusätzliche Homomorphismen hinzugenommen werden, die die Eingabewerte bzw. Ausgabewerte zwischen den beiden Problemen abbilden. Für unser Projekt ist die direkte Reduktion jedoch nicht so interessant; eine weitere Form der Reduktion, die *Spezialisierung*, die wir weiter unten beschreiben, wird sich allerdings als relevant herausstellen.

Wir nennen hier die Reduktion auch, weil sie die bekannteste Methode zur schematischen Erzeugung von Algorithmen ist. An diesem ersten Beispiel sehen wir, daß wir zwei Ebenen unterscheiden müssen: Die *Theorieebene*, hier vertreten durch das Reduktionstheorem, sichert abstrakt die Korrektheit des Schemas, während auf der *Modellebene* konkrete Algorithmen zusammengesetzt werden, und zwar mittels des Programmschemas; das Axiom vermittelt beide Ebenen. Für die Implementierung eines Algorithmenschemas ist dieser Unterschied von größter Bedeutung:

- Die Theorie gibt den allgemeinen Rahmen vor, der das Schema behandelt; sie ist fallunabhängig und kann vorab aufgestellt und bewiesen werden. *Die Theorie legt die Allgemeinheit des Schemas fest.*

- Die Modelle werden erst in einer konkreten Anwendung verfügbar. Ihre Eigenschaften müssen zum Beweis des Axioms herangezogen werden, welcher fallweise angetreten wird. Hier wird erst klar, ob die Bedingungen, die die Theorie mittels des Axioms vorgibt, in konkreten Fällen überhaupt erfüllt werden können. *Auf der Modellebene entscheidet sich die Flexibilität des Schemas.*

Ein Schema kann sich als zu allgemein erweisen, weil die Bedingung, die im Axiom ausgedrückt wird, zwar theoretisch von vielen Algorithmen erfüllt wird, im Einzelfall aber nur schwer nachzuweisen

$$
\begin{tikzcd}
\text{Problemtheorie } \mathcal{P} \arrow[r, "\text{erweitert zur}"] \arrow[d, dash, "\text{hat Modell}"] & \text{Programmtheorie } \mathcal{F} \arrow[d, dash, "\text{hat Modell}"'] \\
\text{Problem } P \arrow[r, "\text{Programmschema ergibt}"] & \text{Programm } f \text{ korrekt für } P
\end{tikzcd}
$$

Abb. 1.3. *Schematische Generierung von Algorithmen*

ist. Die Grenze zwischen Theorie- und Modellebene ist damit außerordentlich kritisch für das Gelingen einer Synthesemethode.

Zur genaueren Benennung der Aspekte beider Ebenen führen wir folgende Begriffe informal ein, die allerdings aus der Algebra bekannt sind.

- Eine *Signatur* besteht aus einer Menge von Namen, denen ein Typ zugeordnet ist. Ist ein Name von einem Funktionstyp, sprechen wir auch von einem *Operator*.
- Eine *Struktur* besteht aus einer Signatur, dessen Namen jeweils ein Objekt passenden Typs zugeordnet ist. Ist ein Objekt von einem Funktionstyp, sprechen wir auch von einer *Operation*.
- Eine *Theorie* besteht aus einer Signatur und einer Menge von Axiomen, in der die Namen der Signatur typadäquat vorkommen dürfen[1].
- Ein *Modell für eine bestimmte Theorie* ist eine Struktur, dessen Objekte die Theorie erfüllen, wobei Struktur und Theorie dieselbe Signatur haben.

Wir interessieren uns für Signaturen, die ein Problem bzw. einen Algorithmus, der ein Problem lösen soll, benennen. Eine entsprechende Theorie heißt *Problemtheorie*, wenn die Signatur einen Namen eines Problems enthält. Die Signatur einer *Programmtheorie* enthält außerdem den Namen einer berechenbaren Funktion, die vom Typ her zu dem Problem paßt und die das Problem erfüllt. Programmtheorien sind also genauer gesagt Theorien *formal korrekter Programme*. Im Unterschied zu einfachen Problemtheorien enthalten sie das Korrektheitsaxiom, das eben die Korrektheit des Algorithmus postuliert.

Abbildung 1.3 legt nun den Unterschied zwischen Theorie- und Modellebene auseinander: oben wird der theoretische Zusammenhang zwischen Problem und Programm, unten der modellbezogene gezeigt. Die Problemtheorie wird nun im allgemeinen neben dem Problem weitere Größen einführen und, in Form von Axiomen, Annahmen über das Problem voraussetzen. Eine in dieser Weise ausgearbeitete Problemtheorie $\mathcal{P}$ heißt nun *Algorithmentheorie*, sofern ein Modell dieser Theorie, das also ein konkretes Problem und konkrete weitere Objekte enthält, mittels eines Programmschemas zu einem korrekten Programm erweitert werden kann.

Wir zeigen dies am Beispiel der direkten Reduktion: Eine entsprechende Algorithmentheorie der direkten Reduktion beschriebe außer dem zu lösenden Problem P_0 als sekundäre Größen das Problem P_{std} und den Algorithmus f_{std} sowie als Axiome die Korrektheit des Standard-Algorithmus und das Reduktionsaxiom. Um nun überhaupt ein Modell zu bilden, muß man neben dem konkreten Problem, das zu lösen ist, auch ein konkretes Standardproblem, auf das man sich beruft, und den Standardalgorithmus angeben; dabei müssen die Axiome erfüllt sein. Die Tatsache, daß wir eine Algorithmentheorie

[1] Häufig wird stattdessen von einem *Axiomensystem* gesprochen, und mit *Theorie* werden alle Folgerungen dieses System bezeichnet.

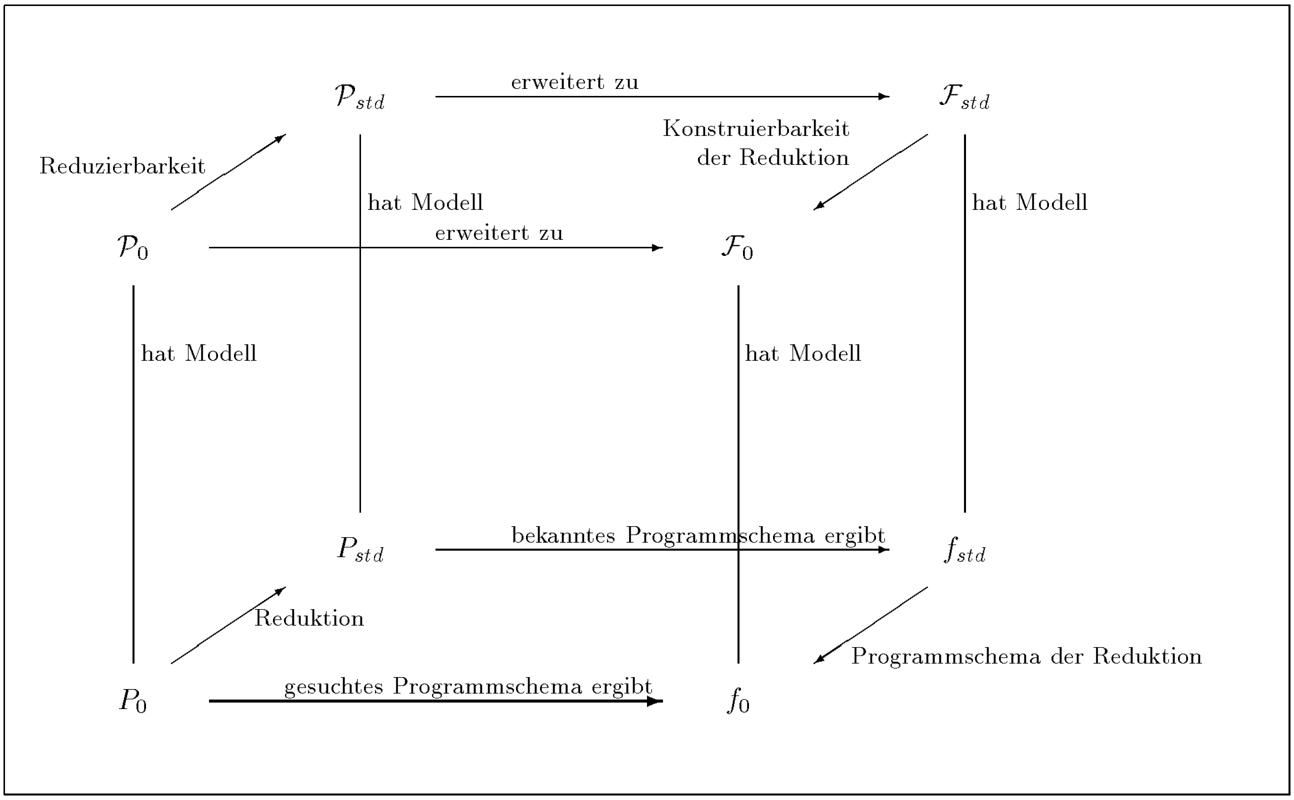


Abb. 1.4. *Hintereinanderschaltung von Reduktion und Algorithmenschema*

haben, liefert uns nun das Programmschema sowie die Gewißheit, daß dieses Schema zu korrekten Programmen führt.

Vergleichen wir die beiden Abbildungen 1.1 und 1.3: In ersterer wird eine spezielle Weise dargestellt, wie neues auf bekanntes Wissen zurückgeführt wird; in letzterer, wie allgemein ein konkretes Problem auf eine allgemeine und bekannte Theorie angewandt wird. Es ist ggf. attraktiv, beide Bezugsarten hintereinanderzuschalten, d.h. das gesuchte Problem zunächst via Reduktion auf ein Standardproblem zurückzuführen, welches dann mittels eines Algorithmenschemas gelöst wird, das sich selbst trägt und nicht auf einen vorhandenen Algorithmus abstützt. Abbildung 1.4 zeigt diese Kombination, dargestellt als Würfel. Links stehen die Probleme, rechts die Programme; oben die Theorien, unten die Modelle, ganz wie in Abbildung 1.3. Neu ist, daß die Tiefe den Unterschied zwischen gesuchtem Algorithmus (Index 0) und Standardalgorithmus (Index std) ausdrückt. Die gesuchte Verbindung ist die zwischen P_0 und f_0, die aber zunächst unbekannt ist. Es kann jedoch das Problem P_0 auf P_{std} reduziert werden, für das es eine Standard-Algorithmentheorie $\mathcal{P}_{std}$ gibt. Dadurch ist garantiert, daß man mittels eines bekannten Programmschemas das Modell f_{std} der Programmtheorie $\mathcal{F}_{std}$ erzeugen kann. Durch Anwendung des durch die Reduktion definierten Programmschemas erhält man nun daraus den gesuchten Algorithmus f_0. – Betrachtet man den Würfel übrigens von der Seite, kann man die Reduktion selbst als Algorithmentheorie deuten.

Bei der Implementierung einer Algorithmentheorie nach dem Würfel-Modell kommt es vor allem darauf an, an welcher Stelle Standard-Methoden benutzt werden und an welcher Stelle die Anpassung des Standards an das jeweilige konkrete Problem stattfindet. „Standard“ heißt dabei, daß einige Theorien oder Modelle fertig zur Auswahl stehen. Dabei nehmen wir eine *Wissensbank* an, die die bestehenden Theorien oder Modelle sowie eine Aussage über diese enthält. Über eine Theorie sollte ausgesagt werden, mit welchem Programmschema ein korrektes Programm generierbar ist; über Modelle sollte ausgesagt werden, welche Theorie erfüllt wird. Es gibt generell mehrere Möglichkeiten, die Theorien und Modelle aus Abbildung 1.4 als Standard anzusehen:

1. $\mathcal{P}_{std}$ und die Reduktionstheorie sind standardisiert. Dies ist die minimal erforderliche Standardisierung, da alle Bedingungen, die an das Modell P_{std} zu stellen sind, bekannt sind. Weitere Anhaltspunkte, wie dieses Modell konstruiert werden soll, gibt es jedoch nicht. Daher wird man eine der beiden folgenden Standardisierungen vorziehen.

2. $\mathcal{P}_{std}$, P_{std} und die Reduktionstheorie sind standardisiert. Genauer gesagt, ist die Beziehung von P_{std} zu $\mathcal{P}_{std}$ in der Wissensbank gespeichert. Dies heißt, daß man zu dem zu lösenden Problem P_0 ein Modell P_{std} in der Wissensbank sucht, von dem man bereits weiß, daß es Modell der Algorithmentheorie ist. Es ist nun nur noch die Reduktion von P_0 auf P_{std} zu bestimmen.

3. $\mathcal{P}_{std}$, die Reduktionstheorie und das Reduktionsmodell sind standardisiert. In der Wissensbank finden wir nun ebenfalls Kandidaten für P_{std}, jedoch nur insofern, als es sich um häufig vorkommende Reduktionsmöglichkeiten handelt. Um nun ein konkretes Problem zu lösen, wird man zunächst blind reduzieren, um dann zu hoffen, daß sich das erhaltene P_{std} als Modell erweist.

Welcher Anteil nun standardisiert wird, hängt sehr davon ab, wie die Ziele Allgemeinheit und Flexibilität dadurch erreichbar werden, und natürlich, ob die Gesamtmethode in ausreichender Weise die Problemlösung in eine bestimmte Richtung lenkt. In der weiteren Arbeit werden wir zeigen, daß die zweite Möglichkeit erfolgreich beschritten werden kann. Die dritte Möglichkeit ist damit aber nicht ganz bedeutungslos, weil man im Prinzip auch mehrere Reduktionen dem eigentlichen Algorithmenschema vorschalten könnte, und man dann weitere Anhaltspunkte bräuchte, welche Reduktionen in frage kommen.

1.2 Globalsuche

Bei der Globalsuche geht es darum, systematisch alle möglichen Kandidaten für die Lösung eines Problems auszuprobieren. Diese Kandidaten müssen dazu als ein (globaler) Suchraum organisiert werden, d.h. es muß einen Algorithmus geben, der alle Kandidaten aufzählt. Man kann auf diese Weise zwei Typen von Problemen behandeln: Ein *Punktproblem* gibt sich damit zufrieden, irgendeine Lösung innerhalb des Suchraums zu finden, während man bei einem *Mengenproblem* an allen Lösungen interessiert ist, die sich im Suchraum finden lassen.

Die Spezifikation des Punktproblems notieren wir formal als (man beachte, daß hier, wie auch beim Mengenproblem, die spezifische algorithmische Lösungsmethode natürlich nicht präjudiziert wird; die Beschreibung des Suchraums taucht also noch nicht auf):

$$\texttt{FUNCTION}\ f(x{:}\,D){:}\,R\ \texttt{WHERE}\ I(x)\ \texttt{RETURNS}\ (y \mid O(x,y))$$

Dies bedeutet, daß eine Funktion vom Typ $D \to R$ gesucht wird, wobei diese Funktion f für jede Eingabe x, für die die Eingabebedingung $I(x)$ erfüllt ist, eine Ausgabe y berechnet, die der Ausgabebedingung $O(x, y)$ genügt. Wie man sieht, ist die Form des Punktproblems die allgemeine Form von formalen Spezifikationen. Beim Mengenproblem geben wir uns nun nicht mit irgendeiner Lösung für y zufrieden, sondern suchen die endlichen Menge aller Lösungen $\{y \mid O(x, y)\}$:

$$\texttt{FUNCTION}\ f(x{:}\,D){:}\,\texttt{Set}(R)\ \texttt{WHERE}\ I(x)\ \texttt{RETURNS}\ \{y \mid O(x,y)\}$$

Dabei ist `Set` der Typkonstruktor für endliche Mengen eines Basistyps. Das Mengenproblem ist nicht immer erfüllbar, wenn für das Punktproblem eine Lösung gefunden werden kann, und damit, obwohl

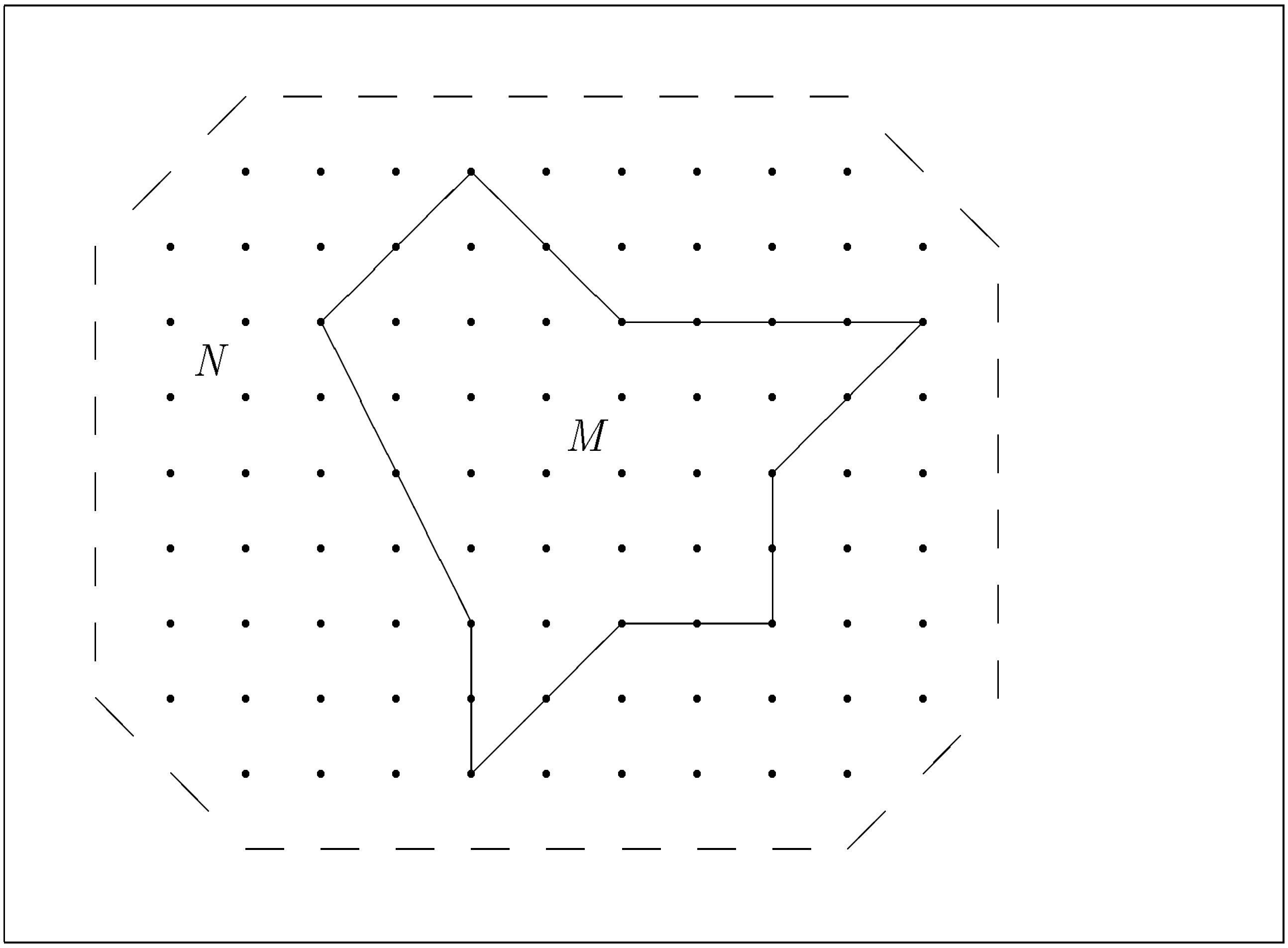


Abb. 1.5. *Globalsuche: Die gesuchte Menge M wird bestimmt, indem der (gestrichelte) Suchraum N systematisch durchsucht wird (angedeutet durch die Punkte)*

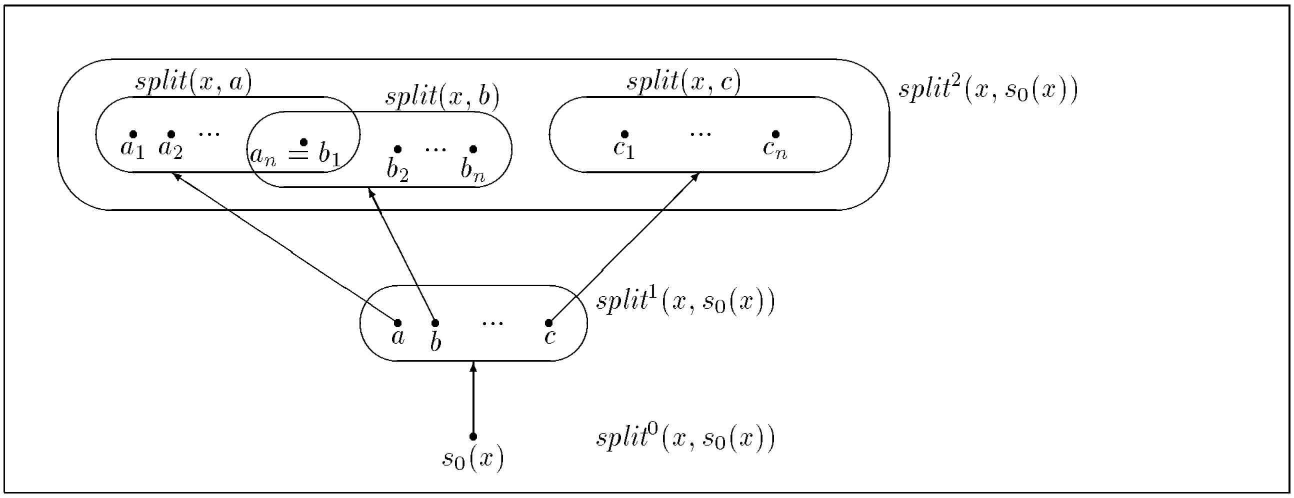


Abb. 1.6. *Aufspannung des Globalsuchraums durch die Funktionen s_0 und split*

auf den ersten Blick die Fragestellung ausgedehnt wird, ein spezielleres Problem als das Punktproblem. Falls es nämlich unendlich viele Lösungen für y gibt, werden wir sicherlich keinen Algorithmus finden, dessen Ausgabe dies als endliche Menge repräsentieren kann. Das Punktproblem kann in diesem Fall aber trotzdem erfüllbar sein. Umgekehrt kann man, wenn es keine Punktlösung gibt, immer die leere Menge zurückgeben. Für das weitere betrachten wir daher immer das Mengenproblem als die eigentliche Aufgabenstellung, vor allem deshalb, weil so das Ergebnis eines Absuchens eines endlichen Suchraums repräsentiert werden kann[2].

Die Idee der Globalsuche ist, (für jede Eingabe x) einen *Suchraum* N anzugeben, der die gesuchte Menge M mit Sicherheit überdeckt (Abbildung 1.5), der endlich ist und für den es eine Aufzählungsfunktion gibt. Wir können dann nämlich für eine Eingabe x alle Punkte y des Suchraums durchlaufen und für jeden Punkt die Bedingung $O(x, y)$ auswerten. Ergibt sich als Ergebnis `true`, so gehört y zu M, andernfalls nicht. Wir müssen dafür natürlich voraussetzen, daß die Bedingung O auswertbar ist oder zumindest eine Auswertungsfunktion gefunden werden kann.

Was wir suchen, ist eine Algorithmentheorie der Globalsuche, d.h. eine Problemtheorie, für deren Probleme mittels eines Programmschemas ein Programm generiert werden kann. Um dieses Schema und die Axiome der Theorie angeben zu können, müssen wir zunächst genauer angeben, wie die allgemeine Form des Suchraums aussieht.

Werte $y \in R$ haben nicht unbedingt eine günstige Darstellung, um den Suchraum aufzuzählen. Statt R wird daher ein *Deskriptortyp* S benutzt, die für diesen Zweck geeigneter ist. Der Suchraum wird nun mit den Funktionen s_0, *split* und *ext* wie folgt definiert:

- $s_0 \in D \to S$:
  $s_0(x)$ ist der initiale Deskriptor für die Eingabe x, d.h. $s_0(x)$ ist der erste Punkt der Aufzählung des Suchraums.

- $\mathit{split} \in D \times S \to \mathtt{Set}(S)$:
  $\mathit{split}(x, s)$ ergibt die Menge der direkten Folgedeskriptoren von s bei Eingabe x. Die Menge der k-ten Folgedeskriptoren split^k ist rekursiv definiert durch:
  - $\mathit{split}^0(x, s) = s$
  - $\mathit{split}^{k+1}(x, s) = \bigcup_{t \in \mathit{split}^k(x,s)} \mathit{split}(x, t)$

Ein Beispiel, wie durch s_0 und *split* ein Suchraum definiert wird, sehen wir in Abbildung 1.6. Jeweils ein Punkt wird durch *split* in eine Menge von Punkten abgebildet, hier durch die Ovale verdeutlicht. Die durch die Exponentiation split^k erzeugte Gesamtmenge enthält nun alle Punkte, die in genau k Abbildungsschritten vom Ausgangspunkt $s_0(x)$ erreichbar sind. Ein Punkt kann dabei grundsätzlich auch mehrfach erreichbar sein, wie hier a_n, der auch als b_1 referenziert werden kann.

Es ist jedoch nicht beabsichtigt, daß der Suchraum ins unendliche wächst, etwa durch Zyklenbildung. Hierzu definieren wir genauer: Der von s_0 und *split* für die Eingabe x aufgespannte Suchraum heißt endlich, falls es ein k gibt, so daß $\mathit{split}^k(x, s_0(x)) = \emptyset$. Nach k Schritten gibt es also keinen Nachfolgepunkt mehr, so daß die Suche abgebrochen werden kann.

Aus der Eigenschaft der Endlichkeit folgt sofort die Zyklenfreiheit, in Formeln: Für jedes $s \in \mathit{split}^k(x, s_0(x))$ gilt, daß für alle $k' > k$ der Deskriptor $s \notin \mathit{split}^{k'}(x, s_0(x))$. Ansonsten könnten wir nämlich durch beliebig häufiges Durchlaufen des Zyklus beliebig viele Abbildungsschritte ausführen.

Die Repräsentation des Suchraums muß sich wieder in die eigentlich geforderte Darstellung als Punkte aus R übersetzen lassen. Dazu:

[2] Es gibt natürlich benachbarte Problemstellungen. So könnte man auf die Endlichkeit des Suchraums verzichten, wenn es ein Kriterium gibt, das erkennt, ob das letzte zum Ergebnis gehörige Element bei der Suche gefunden wurde; auf die Terminierung im Fall der leeren Menge verzichtet man.

- $ext \in S \to \mathtt{Set}(R)$:
  $ext(s)$ ist die Menge der durch s direkt repräsentierten Punkte in R.

Diese drei Funktionen s_0, *split* und *ext* sollen berechenbar sein. Damit ist die Globalsuche mit folgendem Programmschema, in dem die Funktionen O, s_0, *split* und *ext* zu instanziieren sind, lösbar:

$\triangleright$ letrec $\mathtt{global\_search}(x) =$
  letrec $\mathtt{descriptor\_search}(s) =$
    $\{y \mid y \in ext(s) \wedge O(x,y)\}$
    $\cup \bigcup_{t \in split(x,s)} \mathtt{descriptor\_search}(t)$
  in $\mathtt{descriptor\_search}(s_0(x))$

Zum Beweis der Korrektheit dieses Schemas müssen wir allerdings, neben den bereits genannten, weitere Voraussetzungen machen. Das Schema enthält leider keinen direkten Zusammenhang zwischen x und den für x definierten Deskriptoren einerseits und auch nicht zwischen einem Deskriptor und der möglicherweise von ihm repräsentierten Punktmenge andererseits ($ext(s)$ ist davon abhängig, daß s durch s_0 und *split* erzeugt worden ist und damit indirekt auch von der Eingabe x abhängig). Aus diesem Grunde fordern wir die Existenz zweier Prädikate J und *sat*:

- $sat(y, s)$ ist das Erfüllbarkeits-Prädikat, das genau dann wahr ist, wenn y von s oder von einem Folgedeskriptor von s repräsentiert wird, d.h.

  $$sat(y,s) \iff \exists k.\, \exists t.\, t \in split^k(x, s_0(x)) \Rightarrow y \in ext(t)$$

  Man beachte, daß die Definition von *sat* unabhängig von der Eingabe x ist, d.h. für alle Eingaben gilt. Damit gibt *sat*, ohne darauf einzugehen, ob oder wie s zustande kommt, an, welche Punkte y durch s repräsentiert werden. *sat* kann man daher auch als die ausgabeseitige Definition des Suchraums ansehen.

- $J(x, s)$ ist die Invariante des Suchraums für x. $J(x, s)$ soll gelten für alle Deskriptoren, die für x durch s_0 und *split* erzeugt werden. J stellt den Zusammenhang der Eingabe x mit dem Deskriptor s her; es handelt sich also um die eingabeseitige Definition des Suchraums.

Mit diesen Hilfsmitteln können wir nun die Theorie der Globalsuche angeben. Zunächst definieren wir jedoch die zugrunde liegende Signatur.

Definition 1.2.1 (Signatur der Globalsuche)

Die Signatur (Vereinbarung von Namen mit Typ)

1. FUNCTION $f(x\colon D)\colon \mathtt{Set}(R)$ WHERE $I(x)$ RETURNS $\{y \mid O(x,y)\}$ *: Spezifikation, (wobei nur die Namen D, R, I, O vereinbart werden, und x, y, f als Hilfssymbole anzusehen sind)*
2. S *: Typ*
3. J *: Prädikat auf* $D \times S$
4. *sat : Prädikat auf* $R \times S$
5. s_0 *: berechenbare Funktion* $D \to S$
6. *split : berechenbare Funktion* $D \times S \to \mathtt{Set}(S)$
7. *ext : berechenbare Funktion* $S \to \mathtt{Set}(R)$

heißt Signatur der Globalsuche.

Sowohl die Theorie als auch die Modelle für die Theorie stützen sich auf diese Signatur. Die darin vorkommende Spezifikation definiert das zu lösende Problem, wobei wir, über allen Formalismen stehend, locker die Namen D, R, I, O vereinbaren, obwohl der Typ „Spezifikation" für das angegebene Problem-Schema, in dem diese Namen vorkommen, gelten soll.

Definition 1.2.2 (Algorithmentheorie der Globalsuche)

Die für die Signatur der Globalsuche definierte Theorie, die aus den Axiomen

1. *Verankerung der Invariante:*
   $\forall x{:}\, D.\; I(x) \Rightarrow J(x, s_0(x))$
2. *Induktionsschritt der Invariante:*
   $\forall x{:}\, D.\; \forall s{:}\, S.\; I(x) \wedge J(x,s) \quad \Rightarrow \quad (\forall t \in \mathit{split}(x,s).\; J(x,t))$
3. *Notwendigkeit des Suchraums:*
   $\forall x{:}\, D.\; \forall y{:}\, R.\; I(x) \wedge O(x,y) \quad \Rightarrow \quad \mathit{sat}(y, s_0(x))$
4. *Definition der Erfüllbarkeit:*
   $\forall x{:}\, D.\; \forall s{:}\, S.\; \forall y{:}\, R.\; I(x) \wedge J(x,s) \wedge I(x,y) \quad \Rightarrow$
   $\quad \mathit{sat}(y,s) \iff \exists k.\; \exists t{:}\, S.\; t \in \mathit{split}^k(x,s) \Rightarrow y \in \mathit{ext}(t)$
5. *Endlichkeit (Wohlfundiertheit) des Suchraums:*
   $\forall x{:}\, D.\; \forall s{:}\, S.\; I(x) \wedge J(x,s) \quad \Rightarrow \quad \exists k.\; \mathit{split}^k(x,s) = \emptyset$

besteht, heißt wohlfundierte Theorie der Globalsuche.

Wie man sieht, benötigen wir J eigentlich nur, um *sat* vernünftig definieren zu können, und *sat* nur, um das Notwendigkeitsaxiom anzugeben. Dieses ist eine Präzisierung der bereits eingangs gemachten Voraussetzung, daß der Suchraum die gesuchte Menge überdeckt.

Für eine wohlfundierte Globalsuchtheorie gilt nun der wichtige Satz, daß es sich bei ihr um eine Algorithmentheorie handelt. Finden wir also Instanzen für die Namen *D, R, I, O, J, sat,* s_0*, split, ext*, die die Axiome erfüllen (also ein Modell für die Theorie), so ergibt das weiter oben angegebene Programmschema einen korrekten Algorithmus, der das durch *D, R, I, O* spezifizierte Problem löst.

Lemma 1.2.3

Sei $\mathcal{P}$ *eine wohlfundierte Globalsuchtheorie. Sei* P *ein Modell für* $\mathcal{P}$*;*
$P = (D_P, R_P, I_P, O_P, J_P, \mathit{sat}_P, s_{0,P}, \mathit{split}_P, \mathit{ext}_P)$.

Dann erfüllt das Programm

letrec `global_search`$(x) =$
  letrec `descriptor_search`$(s) =$
    $\{y \mid y \in \mathit{ext}_P(s) \wedge O_P(x,y)\}$
    $\cup \bigcup_{t \in \mathit{split}_P(x,s)}$ `descriptor_search`(t)
  in `descriptor_search`$(s_{0,P}(x))$

die Spezifikation

`FUNCTION` $f(x{:}\, D_P)$`:Set`(R_P) `WHERE` $I_P(x)$ `RETURNS` $\{y \mid O_P(x,y)\}$

Den Beweis hierfür findet man in [smith87].

Nachdem jetzt alles wunderbar gelungen scheint, wollen wir die Probleme nicht verschweigen. Da lediglich die Theorie standardisiert ist, müssen wir, um ein konkretes Programm zu synthetisieren, das durch *D, R, I, O* gegebene Problem um Instanzen für *J, sat,* s_0*, split, ext* ergänzen und anschließend nachweisen, daß diese Struktur die Axiome der Theorie erfüllt, also ein Modell ist. Natürlich kann man versuchen, den Suchraum ebenfalls zu standardisieren, was uns allerdings nicht viel weiter bringt, da die Kenngrößen des Problems *D, R, I, O* in allen Axiomen vorkommen und wir daher keines der instanziierten Axiome aus der Wissensbank abrufen können.

Diese Schwierigkeit können wir erst meistern, wenn wir die Strukturierung nach Abbildung 1.3 verlassen und entsprechend Abbildung 1.4 einen Reduktionsschritt vorschalten. Dadurch müssen wir nicht das gegebene Problem $P_0 = (D, R, I, O)$ *direkt* als Modell der standardisierten Globalsuchtheorie $\mathcal{P}_{std}$ auffassen oder zu solch einem Modell ausbauen; wir haben die Chance, erst durch den Reduktionsschritt von P_0 auf ein geeignetes Modell P_{std} zu kommen. Grob gesagt, verlagern wir einen Großteil der Anpassung der Theorie an den konkreten Fall auf diesen Reduktionsschritt und beziehen P_{std} fix und fertig aus einer Wissensbank. Die Instanzen für *J, sat,* s_0, *split* und *ext*, die ja zu diesem Modell gehören, sind damit vorgegeben und brauchen nicht für jede Anwendung neu erfunden werden. Desweiteren können die Beweise der Axiome, die für jedes Modell durchgeführt werden müssen, ebenfalls aus der Wissensbank bezogen werden.

1.3 Spezialisierte Globalsuche

Zunächst ist zu klären, was es heißt, P_{std} zu standardisieren. Ausgangspunkt ist die Festlegung des Suchraums. Dieser richtet sich, vor allem was das Aufzählungsprinzip angeht, nur nach den Typen D und R. Die Wahl eines Suchraums und von D und R sollte daher möglichst in einem Schritt erfolgen, da beides zusammenpassen muß. Die Eingabebedingung I sollte nun schlicht so schwach wie möglich sein; auch dies richtet sich nach dem Suchraum. Die Wahl der Ausgabebedingung ist am problematischsten, da vor allem hier die eigentliche Aufgabenstellung ausgedrückt wird. Es hängt nun in hohem Maße von der Art des Reduktionsschritts ab, was hier sinnvoll ist. Die Spezialisierung, die wir benutzen wollen, ist gerade deswegen für das Zusammenspiel mit der Globalsuche geeignet, weil sie die Anpassung von O ermöglicht. Bei der Spezialisierung wird auch O möglichst schwach gewählt; die Gründe hierfür sind etwas komplizierter.

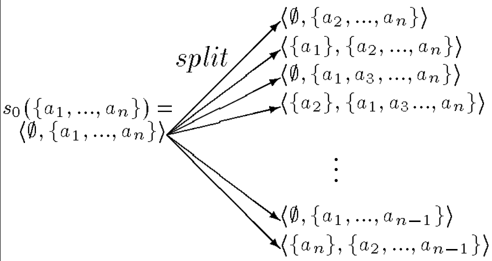
Die Standardisierung von P_{std} bringt die Schwierigkeit mit sich, daß man zwar auf allgemeine Weise ein Aufzählungsprinzip für den Suchraum angeben, den Suchraum jedoch schwerlich begrenzen kann. Wir wollen das an einem positiven und einem negativen Beispiel illustrieren.

Abbildung 1.7 zeigt den Versuch, einen Standard-Suchraum festzulegen, indem alle Teilmengen der Eingabe x, die hierbei ebenfalls eine Menge sein muß, durchprobiert werden. Die Deskriptoren haben die besondere Form von Paaren $\langle s_1, s_2 \rangle$, wobei $s_1 \cup s_2$ den restlichen Suchraum angibt, für den noch alle Teilmengen gebildet werden müssen, und s_1 die aktuelle Teilmenge ist. Die Details entnehme man der Abbildung. Wichtig ist nun, daß dieser Suchraum von vorneherein endlich ist, da jede endliche Menge natürlich nur endlich viele Teilmengen hat. – Abbildung 1.8 ist ein anderer Suchraum für eine mengenwertige Eingabe x. Hier werden alle Repräsentationen von Teilmengen von x als Listen durchprobiert. Im Unterschied zu vorhin ergibt sich das Problem, daß jede Menge durch unendlich viele Listen repräsentiert wird. Diese Art, den Suchraum aufzuspannen, ist also zunächst nicht wohlfundiert. Dieser Suchraum ist nun kein Einzelfall, sondern eher die Regel. Die Konsequenzen daraus sind:

- Die in Abbildung 1.7 angegebenen Instanzen $P_{\texttt{subset}}$ bilden von sich aus ein Modell einer wohlfundierten Algorithmentheorie, d.h. wir besitzen ein Standard-Modell für P_{std}. Wir entsprechen damit voll und ganz dem Entwurf von Abbildung 1.4 und sind sogar frei in der Wahl des vorgeschalteten Reduktionsverfahrens.

- Die Instanzen $P_{\texttt{sets\_as\_seqs}}$ aus Abbildung 1.8 erfüllen jedoch nicht das Wohlfundiertheitsaxiom der Globalsuche. Diese Instanzen taugen deshalb nicht für ein Standard-Modell. Ein zusätzli-

Suchraum aller Teilmengen einer endlichen Menge $P_{\texttt{subset}}$:



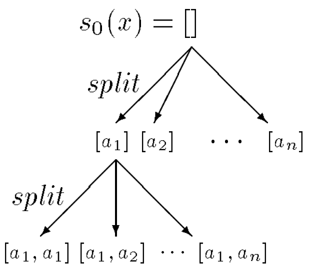
$$\begin{array}{rcll}
D &=& \texttt{Set}(\alpha) & \text{Endliche Menge über } \alpha \\
R &=& \texttt{Set}(\alpha) & \text{Endliche Menge über } \alpha \\
S &=& \texttt{Set}(\alpha)\times\texttt{Set}(\alpha) & \\
&& \text{Paare aus zwei endlichen Mengen} & \\
J(x,\langle s_1,s_2\rangle) &=& s_1\cap s_2=\emptyset \ \wedge\ s_1\cup s_2\subseteq x & \\
&& \text{Darstellungsinvariante der} & \\
&& \text{Deskriptoren} & \\
sat(y,\langle s_1,s_2\rangle) &=& s_1\subseteq y \ \wedge\ y\subseteq s_1\cup s_2 & \\
&& \text{Alle Teilmengen von } s_1\cup s_2 \text{ können} & \\
&& \text{noch erzeugt werden} & \\
s_0(x) &=& \langle\emptyset,x\rangle & \text{(s. links)} \\
split(x,\langle s_1,s_2\rangle) &=& \{\langle s_1,\ s_2\setminus\{a\}\rangle \ \mid a\in s_2\}\ \cup & \\
&& \{\langle s_1\cup\{a\},\ s_2\setminus\{a\}\rangle \ \mid a\in s_2\} & \\
ext(\langle s_1,s_2\rangle) &=& \text{if } s_2=\emptyset \text{ then } \{s_1\} \text{ else } \emptyset & \\
I(x) &=& \text{true} & \\
&& \text{schwächste Eingabebedingung} & \\
O(x,y) &=& y\subseteq x & \\
&& \text{schwächste Ausgabebedingung} &
\end{array}$$

Der Suchraum ist endlich. Es müssen keine Filter benutzt werden.

Abb. 1.7. *Beispiel für einen Standard-Suchraum*

Suchraum aller Listenrepräsentationen einer endlichen Menge $P_{\texttt{sets\_as\_seqs}}$:

$x=\{a_1,...,a_n\}$:



D	$=$	$\texttt{Set}(\alpha)$	Eingaben sind endliche Mengen über α.
R	$=$	$\texttt{Seq}(\alpha)$	Ausgabepunkte sind Listen über α
S	$=$	$\texttt{Seq}(\alpha)$	Deskriptoren ebenso
$J(x,s)$	$=$	$\texttt{range}(s)\subseteq x$	Deskriptorliste enthält nur Werte der Eingabe
$sat(y,s)$	$=$	s Präfix von y	Deskriptor erreicht noch alle Punkte, die den Deskriptor als Präfix haben.
$s_0(x)$	$=$	$[]$	siehe links
$split(x,s)$	$=$	$\{s\cdot a \mid a\in x\}$	siehe links
$ext(s)$	$=$	$\{s\}$	da R direkt als Suchtyp S benutzt wird
$I(x)$	$=$	true	schwächste Eingabebedingung für diesen Suchraum
$O(x,y)$	$=$	$\texttt{range}(y)\subseteq x$	schwächste Ausgabebedingung für diesen Suchraum

Der Suchraum ist unendlich. Es müssen Filter benutzt werden, um ihn zu begrenzen.

Abb. 1.8. *Beispiel für einen Standard-Suchraum* ches Konzept muß her, um die Wohlfundiertheit der Suche zu gewährleisten, womit wir uns im weiteren beschäftigen.

Aus diesem Grunde erweitern wir die Darstellung der Globalsuche noch um einen *Filter* ϕ, der getrennt von den anderen Anteilen von P_{std} aus der Wissensbank bezogen wird und die jeweilige Beschränkung des Suchraums vollzieht. Die Modifikation der bisherigen Suche braucht nur die Berechnung der Folgedeskriptormenge einzubeziehen. Sei $split_w$ die aus der Abfrage der Wissensbank erhaltene Funktion, so definieren wir das tatsächlich verwendete *split* unter Einbeziehung des Filters ϕ durch:

$$split(x,s) = \{t \mid t \in split_w(x,s) \wedge \phi(x,t)\}$$

Filterung ist nicht nur zur Wohlfundierung des Suchraums nützlich, sondern kann auch verwendet werden, um einen Standard-Suchraum besser an das konkrete Problem anzupassen, d.h. diesen knapper über die Ausgabemenge zu legen. Dieser Aspekt der Effizienzsteigerung macht Filterung auch für bereits wohlfundierte Suchräume interessant.

Zunächst wollen wir aber bloß wissen, ob wir mit Filtern tatsächlich das Wohlfundiertheitsaxiom erfüllen können.

Es zeigt sich, daß dies erreichbar ist, aber zu einem gewissen Preis. Beispielsweise können wir mehrere Filter für $P_{\texttt{sets\_as\_seqs}}$ angeben, die alle die Wohlfundierung erzielen:

▷ $\phi_1(x,s) \equiv |s| \leq k$
▷ $\phi_2(x,s) \equiv |s| \leq k \cdot |x|$
▷ $\phi_3(x,s) \equiv s$ enthält keine Duplikate

Durch die Begrenzung des Suchraums besteht nun aber die Gefahr, daß der Suchraum nicht mehr notwendig für das jeweilige Problem ist. Man sehe sich hierauf noch einmal $P_{\texttt{subset}}$ an: Das Notwendigkeitsaxiom kann hier bewiesen werden, weil O so gewählt ist, daß die von O spezifizierte Ausgabemenge immer vom Suchraum überdeckt wird; gleichzeitig ist O die schwächste Bedingung, für die das möglich ist. Bauen wir in $P_{\texttt{sets\_as\_seqs}}$ nun einen Filter in die *split*-Funktion ein, so wird damit O aber nicht verändert und ist für den resultierenden Suchraum zu schwach; d.h. die durch O spezifizierte Ausgabemenge (die sogar unendlich ist) wird nicht mehr durch den gefilterten Suchraum überdeckt. Die Notwendigkeit des Suchraums ist damit nicht mehr gegeben.

Eine Möglichkeit, die Notwendigkeit wieder herbeizuführen, besteht darin, mit der Modifikation der *split*-Operation beim „Einbau" des Filters auch die O-Bedingung zu verändern:

▷ $O(x,y) = O_w(x,y) \ \wedge \ \forall s{:}\, S.\ sat(y,s) \Rightarrow \phi(x,s)$

Diese Idee verfolgen wir hier nicht weiter, und zwar aus dem einfachen Grund, weil in der Literatur anders vorgegangen wird und erst noch erforscht werden müßte, ob sie zum Erfolg führt[3].

Der andere Weg, der auch in der Literatur verfolgt wird, besteht darin, nur besonders qualifizierte Filter zu verwenden, die *notwendigen Filter*.

---

[3] Sollte dies der Fall sein, würde dies zu einer wesentlichen Vereinfachung der Theorie führen, die dann strikt der Abbildung 1.4 folgen könnte; Globalsuche und Spezialisierung müßten nicht mehr zu einer einzigen Theorie zusammengezogen werden. Der Begriff des „notwendigen Filters" würde entfallen, da dies in der O-Bedingung bereits ausgedrückt ist. Bei der Spezialisierung steckt die Notwendigkeitsaussage daher in O und braucht nicht gesondert berücksichtigt werden. Für den Anwender der vielleicht etwas kohärenteren Theorie ändert sich jedoch nichts.

Definition 1.3.1 (Notwendiger Filter)

Ein Filter ϕ heißt notwendig, falls gilt:

$$\forall s{:}\, S.\; I(x) \wedge (\exists y{:}\, R.\; sat(y,s) \wedge O(x,y)) \;\Rightarrow\; \phi(x,s)$$

Im folgenden nennen wir eine Struktur für die Signatur der Globalsuche der Einfachheit halber Globalsuchraum; gleichgültig, ob Axiome erfüllt werden oder nicht. Einen solchen Globalsuchraum kann man, wie schon angegeben filtern und dann eine Aussage machen:

Definition 1.3.2 (Gefilterter Globalsuchraum)

Sei $gs = (D, R, I, O, J, sat, s_0, split_w, ext)$ eine Struktur passend zur Signatur der Globalsuche. Sei ϕ ein Filter. Dann heißt

$$\begin{array}{l} gs_\phi = (D, R, I, O, J, sat, s_0, split, ext) \\ mit\ split(x,s) = \{t \mid t \in split_w(x,s) \wedge \phi(x,t)\} \end{array}$$

der mit ϕ gefilterte Globalsuchraum.

Lemma 1.3.3 (Gefilterte Globalsuche)

Sei gs ein Globalsuchraum und ϕ ein für gs notwendiger Filter. Desweiteren erfülle gs die Axiome 1 bis 4 der Globalsuchtheorie. Falls nun gs_ϕ das 5. Axiom (Wohlfundiertheit) erfüllt, so ist gs_ϕ ein Modell der Theorie der Globalsuche.

Im weiteren werden wir nun auch noch die Spezialisierung hinzunehmen. Leider ergibt sich dabei, daß die Notwendigkeit, die durch den Filter gesichert wird, nicht relativ zum bloß gefilterten, sondern relativ zum gefilterten und dann für das jeweilige konkrete Problem spezialisierte Globalsuchraum gezeigt wird. Dieser Beweis ist daher bei jeder Anwendung neu zu führen.

Definition 1.3.4 (Spezialisierung eines Problems)

Das Problem

$$P = \texttt{FUNCTION}\; f_P(x{:}\, D_P){:}\, R_P\; \texttt{WHERE}\; I_P(x)\; \texttt{RETURNS}\; \{y \mid O_P(x,y)\}$$

heißt Spezialisierung des Problems

$$Q = \texttt{FUNCTION}\; f_Q(x{:}\, D_Q){:}\, R_Q\; \texttt{WHERE}\; I_Q(x)\; \texttt{RETURNS}\; \{y \mid O_Q(x,y)\}$$

falls folgende Bedingung (das Spezialisierungsaxiom) gilt:

$$\begin{array}{l} R_P = R_Q \wedge \\ \quad \exists \theta{:}\, D_P \to D_Q.\, \forall x{:}\, D_P. \\ \qquad I_P(x) \;\Rightarrow\; I_Q(\theta(x)) \wedge \forall y{:}\, R.\, O_P(x,y) \Rightarrow O_Q(\theta(x), y) \end{array}$$

Wir schreiben dies auch als $P \ll Q$. Falls es darauf ankommt, daß mit einer bestimmten Funktion θ spezialisiert wird, schreiben wir auch $P \ll_\theta Q$. Die Funktion θ heißt auch Spezialisierungstransformation.

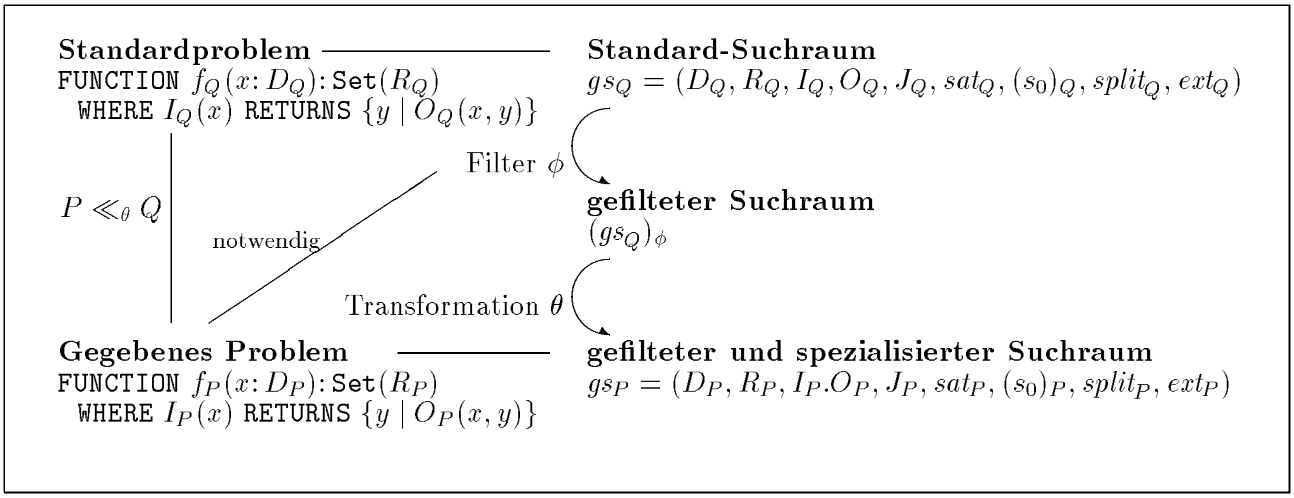


Abb. 1.9. *Verhältnisse auf der Modellebene bei der spezialisierten und gefilterten Globalsuche*

Die Idee bei der Spezialisierung ist ähnlich der Grundidee der Globalsuche. Es wird davon ausgegangen, daß die Ausgabemenge des Problems Q die Ausgabemenge von P überdeckt, weshalb man den Lösungsalgorithmus für Q auch für P verwenden kann, wenn man die überschüssigen Elemente aus der Ergebnismenge herausstreicht. Neu ist, daß die Eingabe x von P erst mittels einer Transformation θ in die Eingabe $\theta(x)$ von Q abgebildet wird.

Es ist nun auch klar, warum die Ausgabebedingung des Problems in der Standard-Globalsuchbeschreibung möglichst schwach sein sollte. Der Spezialisierungsschritt ist nämlich in der Lage, die durch die zu schwache Bedingung zu viel berechneten Elemente wieder zu entfernen.

Eine Reduktion wie die Spezialisierung ist selbst wieder ein Algorithmenschema. Das zugehörige Programmschema lautet:

$$f_P(x) = \{y \mid y \in f_Q(\theta(x)) \wedge O_P(x,y)\}$$

Lemma 1.3.5 (Algorithmenschema der Spezialisierung) *Seien P und Q Mengenprobleme mit $P \ll_\theta Q$. Die Ausgabebedingung O_P von P sei auswertbar. Es sei f_Q ein Programm, das Q erfüllt. Dann erfüllt (obiges) f_P die Spezifikation P.*

Die Spezialisierung läuft auf eine zweite Filterung mittels der gewünschten Ausgabebedingung hinaus. Es ist jetzt natürlich die Frage, was dies für die Notwendigkeit des ersten Filters bedeutet. Letztendlich ist ja unser Ziel, die Anteile der Globalsuche, die standardisiert sind, von den Anteilen zu trennen, die fallweise nachgewiesen werden müssen; die Notwendigkeit von ϕ muß nun dazu führen, daß der Suchraum die spezialisierte Ausgabemenge O_P überdeckt, d.h. sie wird fallweise gezeigt. Dies hat nun zur Folge, daß die Theorie eine recht eigentümliche Form annimmt.

In Abbildung 1.9 sehen wir die bereits recht komplizierten Verhältnisse auf der Modellebene. Zur Lösung des gegebenen Problems P soll das Standard-Problem Q benutzt werden, zu dem es in der Wissensbank einen Suchraum gs_Q gibt. Wir müssen nun zunächst nachweisen, daß $P \ll Q$ gilt. Desweiteren benötigen wir einen Notwendigkeitsbeweis für den noch hinzuzunehmenden Filter ϕ. Der gefilterte Suchraum $(gs_Q)_\phi$ ist nun wohlfundiert und damit ein Modell der Globalsuchtheorie. Die Notwendigkeit muß jedoch bezüglich des Problems P gezeigt werden und ist damit eine fallweise zu zeigende Eigenschaft. Der Grund dafür ist, daß bei der durchzuführenden Transformation mittels θ der Suchraum noch weiter eingeschränkt wird, wir aber letztlich daran interessiert sind, den dann transformierten Suchraum gs_P als Modell der Globalsuchtheorie zu verwenden. Letzterer soll ja die tatsächliche Lösung immer noch überdecken, damit alle Teile dieser Lösung gefunden werden können. Der Suchraum gs_P ist nun erstens ein Suchraum für das Problem P, das wir eigentlich lösen wollen, und zweitens ein Modell der Globalsuchtheorie. Damit können wir Lemma 1.2.3 anwenden und erhalten einen Algorithmus für das Problem P.

Wir geben nun den finalen Satz an, der die Algorithmentheorie für diese Kombination beschreibt:

Lemma 1.3.6 (Algorithmentheorie der gefilterten Globalsuche)

Sei

$$gs_Q = (D_Q, R_Q, I_Q, O_Q, J_Q, sat_Q, (s_0)_Q, split_Q, ext_Q)$$

ein Suchraum, der die Axiome 1 bis 4 der Globalsuche erfüllt. Wir bezeichnen mit

$$Q = \texttt{FUNCTION } f_Q(x\colon D_Q)\colon \texttt{Set}(R_Q) \texttt{ WHERE } I_Q(x) \texttt{ RETURNS } \{y \mid O_Q(x,y)\}$$

das in gs_Q definierte Problem.

Sei

$$P = \texttt{FUNCTION } f_P(x\colon D_P)\colon \texttt{Set}(R_P) \texttt{ WHERE } I_P(x) \texttt{ RETURNS } \{y \mid O_P(x,y)\}$$

ein Mengenproblem mit $P \ll_\theta Q$; dabei sei $R_Q = R_P$. Es sei

$$gs'_P = (D_P, R_P, I_P, O_P, J_P, sat_P, (s_0)_P, split'_P, ext_P)$$

mit

$$\begin{aligned} J_P(x,s) &= J_Q(\theta(x), s) \\ sat_P &= sat_Q \\ (s_0)_P(x) &= (s_0)_Q(\theta(x)) \\ split'_P(x,s) &= split_Q(x,s) \\ ext_P &= ext_Q \end{aligned}$$

der spezialisierte Suchraum.

Es sei ϕ_Q ein Filter für gs_Q, so daß $\phi_P(x,s) = \phi_Q(\theta(x), s)$ notwendig für gs'_P ist.

Es sei

$$split_P(x,s) = \{t \mid t \in split'_P(x,s) \wedge \phi_P(x,t)\}$$

Dann ist

$$gs_P = (D_P, R_P, I_P, O_P, J_P, sat_P, (s_0)_P, split_P, ext_P)$$

ein Modell der Globalsuchtheorie.

Folgender Algorithmus f_P erfüllt das Problem P:

$\triangleright$ letrec $\texttt{global\_search}(x) =$
  letrec $\texttt{descriptor\_search}(s) =$
    $\{y \mid y \in ext_P(s) \wedge O_P(x,y)\}$
    $\cup \bigcup_{t \in split_P(x,s)} \texttt{descriptor\_search}(t)$
  in $\texttt{descriptor\_search}((s_0)_P(x))$

Kapitel 2

Die Typentheorie als Programmiersprache

Die Taktik zur Synthese von Globalsuchalgorithmen kann auf vielerlei Weisen implementiert werden. Unterschiede gibt es dabei im Grad der Formalisierung und im Korrektheitsanspruch. Es ist naheliegend, einen Formalismus zu benutzen, der diejenigen mathematischen Aspekte gut beschreiben kann, die für die Globalsuche relevant sind. Um sich nicht mehr technische Schwierigkeiten zu bereiten als notwendig, ist es desweiteren möglich, den Formalismus so zu begrenzen, daß darüberhinausgehende Aspekte nicht berücksichtigt werden. Für erste Experimente mit einer Globalsuch-Taktik ist dieser Zuschnitt des Basis-Formalismus sicherlich angebracht. Genauso wird man den Korrektheitsanspruch sicherlich begrenzen wollen. Zwar gehört es zu den Zielen von Programmsynthese, die Korrektheit der generierten Programme zu garantieren, doch muß man dies nicht unbedingt auf die Spitze treiben. Globalsuche bietet ohnehin eine methodische Grundlage für die Synthese, so daß für die grundsätzliche Vorgehensweise verbürgt werden kann. Die Taktik kann sich darauf verlassen und muß nur die jeweils speziellen Details überprüfen, die bei der Durchführung der Methode anfallen. Insofern ist ein reduzierter Korrektheitsanspruch durchaus ausreichend; das KIDS-System verfährt entsprechend.

In dieser Arbeit wurden die Schwerpunkte jedoch bewußt anders gesetzt. Das KIDS-System hat bereits gezeigt, daß die Globalsuche implementierbar ist und damit erfolgreich praktisch relevante Algorithmen erzeugt werden können. Die Übertragung der Globalsuche auf das NuPRL-System hat daher primär andere Ziele.

Der in NuPRL verwendete Basiskalkül, die Intuitionistische Typentheorie nach Martin-Löf, hat den Anspruch, alle relevanten Aspekte der Mathematik und der Programmierung abzudecken. Er ist also das genaue Gegenteil eines auf eine bestimmte Anwendung zugeschnittenen Formalismus: Die Typentheorie ist universell einsetzbar, kann sogar als mathematische Grundlagentheorie (analog der Mengenlehre) verstanden werden. Die Übertragung der Globalsuche nach NuPRL testet u.a., ob dieser Anspruch überhaupt eingelöst werden kann. Es ist dabei zwischen theoretischer und praktischer Durchführbarkeit zu unterscheiden. Erstere entscheidet darüber, ob der Anspruch der Typentheorie für die mathematischen Aspekte, die bei der Globalsuche relevant sind, gerechtfertigt ist; die praktische Umsetzbarkeit fragt darüber hinaus auch nach den Kosten, die dabei entstehen.

Es ist zu erwarten, daß eine im genannten Sinne universelle Grundlagentheorie teurer ist als eine für den speziellen Zweck zugeschnittene. Dies ist nachteilig. Die Universalität einer solchen Grundlagentheorie bietet jedoch die Möglichkeit, verschiedenste Methoden zu kombinieren, um ein bestimmtes Ziel zu erreichen. Für die Programmsynthese können damit beispielsweise Ansätze, die Formeln mittels Äquivalenztransformationen in ausführbare Terme umformen und so Programme erzeugen, mit der Synthese durch Algorithmenschemata wie etwa der Globalsuche gekoppelt werden. Die Grundlagentheorie ist die einheitliche Plattform, auf die alle Einzelmethoden aufsetzen und die es zuläßt, Konzepte zu entwickeln, die es erlauben, die Einzelmethoden miteinander zu verkoppeln.

Die Typentheorie gilt als ein radikal-formalistischer Ansatz für eine solche Grundlagentheorie. Dies äußert sich insbesondere darin, daß die Überprüfung, ob ein Term eine Aussage oder einen Typ in konsistenter Weise ausdrückt, mit Hilfe der Schlußregeln kalkülimmanent stattfindet und nicht durch eine Prozedur auf der Metaebene des Kalküls. Diese Problematik der immanenten Absicherung der *Wohlgeformtheit* von Termen ist das größte Handicap bei der Anwendung der Typentheorie. Ein typentheoretischer Beweis mündet, so zeigen schon einfachste Beweisaufgaben, andauernd darin, daß eine solche Wohlgeformtheit nachzuweisen ist. Für die manuelle Beweisführung gibt es im NuPRL-System eine Reihe von Einzeltaktiken, die solche Nachweise in vielen Fällen automatisch erbringen können. Sie beruhen auf Heuristiken, die die Erfahrung der NuPRL-Entwickler und -Anwender in Algorithmen zur Verfügung stellen. Manchmal müssen jedoch zusätzliche Argumente angegeben oder Vorverarbeitungsschritte durchgeführt werden, damit diese Taktiken Erfolg haben. In einigen, durchaus vorkommenden Fällen funktionieren sie überhaupt nicht, weil der Wohlgeformtheitsnachweis zu kompliziert ist. Die NuPRL-Entwickler haben dieser Sammlung von Taktiken den etwas irreführenden Namen „Autotaktik" gegeben. Sie ist eine halbautomatische Unterstützung für die manuelle Beweisführung; für eine vollautomatische Beweisführung taugt sie nicht. Wir werden weiter unten sehen, daß wir für einen automatischen Wohlgeformtheitsbeweiser Abstriche bei der universellen Einsetzbarkeit[1] machen müssen.

Daß die Wohlgeformtheitsnachweise inhärent typentheoretisch geführt werden müssen, hat den Vorteil der größeren Ausdruckskraft. Die Wohlgeformtheit eines Terms kann nun von beliebigen, typentheoretisch ausdrückbaren Bedingungen abhängig sein. Die volle Kompliziertheit der möglichen Bedingungen zeigt sich vor allem beim abhängigen Funktionenraum. Es sind Funktionen erlaubt, deren Ergebnistyp vom Argument abhängt. Um den Typ eines Terms zu beweisen, in dem eine solche Funktion vorkommt, muß man über das extensionale Verhalten dieser Funktion schließen, welches beliebig sein kann. Ein universeller Beweiser für solche Wohlgeformtheitsprobleme ist mithin genauso schwierig wie ein universeller Beweiser für beliebige Aussagen der Typentheorie (Beispiele, die dies plausibel machen, werden in Abschnitt 2.2.3 gegeben.) – Bedingte Wohlgeformtheiten drücken einen für die Informatik wichtigen Aspekt der Formalisierung aus. Beispielsweise können wir dem Quotienten x/y nur dann den Typ der ganzen Zahlen $\mathbf{Z}$ zuordnen, wenn gewährleistet ist, daß y niemals Null werden kann[2], in Formeln: $\forall x, y{:}\,\mathbf{Z}.\ \ y \neq 0 \ \Rightarrow\ x/y \in \mathbf{Z}$. Aus Sicht der Mathematik sind solche Überprüfungen partiell definierter Terme zwar notwendig, aber ein Mathematiker würde niemals die Aufmerksamkeit auf ein solches Detail lenken, wenn er es nicht gerade für bedeutsam für die Gesamtargumentation hält. Insbesondere kann man Mathematik erfolgreich anwenden, ohne sich um solche Fragen zu kümmern; in den Natur- und Ingenieurwissenschaften als Anwendungsgebiete werden solche Fragen schlicht ignoriert. In der Informatik ist dies anders. Das Problem partiell definierter Terme wird in der praktischen Programmierung als das Problem von Laufzeitfehlern (Division durch Null,

[1] Den Begriff *Universalität* verwenden wir in einer starken und einer schwachen Bedeutung. Ein Algorithmus (oder eine Methode) funktioniert universell (im starken Sinn), wenn er für alle Elemente einer definierten Klasse funktioniert. Wenn wir auf Erfahrungswerte rekurrieren müssen, etwa bei heuristischen Algorithmen oder Methoden unklarer Reichweite, sprechen wir manchmal (im schwachen Sinn) von universeller Einsetzbarkeit (o.ä. Wortwahl). Dies soll heißen, daß es *a priori* keinen Grund gibt, den Algorithmus oder die Methode für Elemente einer Klasse *nicht* zu probieren, aber andererseits auch keine Erfolgsgarantie gegeben werden kann. Der Rückgriff auf eine derart schwach definierte Kategorie ist dem pragmatischen Zugang geschuldet, der in dieser Arbeit eingeschlagen wird.

[2] Man beachte den Unterschied: Wir ordnen nicht dem *Ergebnis* von x/y einen Typ zu; dies wäre unproblematisch, weil es schon voraussetzt, daß es ein Ergebnis überhaupt gibt. Vielmehr ordnen wir dem *Term* x/y einen Typ zu, also der Erzeugungsvorschrift für das Ergebnis. Dabei müssen wir die Bedingungen formulieren, die den Erfolg der Erzeugung garantieren. Die Typentheorie interpretiert die Typzuordnung $t \in T$ immer so, daß dies auch impliziert, daß t in jedem Fall ein sinnvoll definierter Term ist; $t \in T$ hat insofern eine doppelte Semantik, als zum einen alle Ergebnisse von t dem Typ T zugeordnet werden und zum anderen über den Term t selbst ausgesagt wird, daß er zu (sinnvollen) Ergebnissen führt.

Indexüberschreitung etc.) thematisiert; solche Details werden in der Praxis wichtig, weil reale Computer an ihnen scheitern können. Für die Programmverifikation ist sogar als Argument ins Feld geführt worden, daß Programmierer häufig Fehler wie das Übersehen von Definitionslücken machen, und solche Fehler beim Programmieren im Kleinen durch Programmbeweise vermieden werden könnten. Partiell definierte Terme sind ein besseres Modell für Algorithmen; die Beachtung dieser Eigenschaft führt zu Aussagen, die näher an der Realität liegen[3]. – Im weiteren werden wir Wohlgeformtheitsproblemen weniger in Gestalt abhängiger Funktionenräume als vielmehr in Form partiell definierter Terme begegnen.

Der Entwickler der Typentheorie, Per Martin-Löf, hat versucht, zwei widerstreitende Positionen in seinem Kalkül zu versöhnen: Formalismus und Intuitionismus. Erstere ist bestrebt, alle Erkenntnisse in Termen einer formalen Regeln gehorchenden Sprache ausdrücken zu können. Die Typentheorie folgt dieser Anschauung in geradezu radikaler Weise, indem auch Aspekte, vor allem die schon genannte Wohlgeformtheit, der Sprache inhärent gemacht werden, die üblicherweise auf der Metaebene behandelt werden. Natürlich gibt es dennoch Fragen und Erklärungsbedarf, die nicht innerhalb der Typentheorie besprochen werden können. Dies betrifft vorallem die Grundlegung der Sprache selbst. Martin-Löf erläutert daher seinen Kalkül ausführlich. Er könnte sich stattdessen auch darauf zurückziehen, lediglich die Widerspruchsfreiheit des Kalküls zu zeigen und keine weitere Bedeutungen zu assoziieren. Indem er den Kalkül aber durchsichtig macht und dabei darauf spekuliert, daß die Leser ihre eigenen Vorstellungen über Mathematik darin wiederfinden können, ist er durchaus ein Intuitionist. Formalismus und Intuitionismus geraten leicht dadurch in Widerspruch, daß Sprachen mit strengen Regeln die Vorstellungskraft einengen oder ihr in kontraintuitiver Weise widersprechen können. Martin-Löf versucht dem dadurch zu entgehen, daß er die Sprache in einer dem Intuitionismus entsprechenden Weise begründet, dabei aber ein streng formales Regelwerk aufstellt.

Der intuitionistische Moment der Typentheorie kann leicht übersehen werden, weil der formalistische Charakter viel offensichtlicher ist. Wir müssen jedoch unbedingt beide Züge berücksichtigen, um den Korrektheitsanspruch bestimmen zu können.

2.1 Präliminarien

Um für die folgenden Diskussionen gerüstet zu sein, benötigen wir einige Präzisierungen geläufiger Konzepte. Kenner von NuPRL können diese Einzelheiten überspringen.

2.1.1 Terme und Variablen

Term ist zunächst nur diejenige syntaktische Form, die die Typentheorie benutzt, um sich auszudrücken. Dies ist allerdings nur die Grundbedeutung der Terme, die im weiteren Verlauf erweitert wird; NuPRL verwendet Terme fast schon im Sinne einer universalen Struktur, die die mannigfaltig vorkommenden Ausdrücke in einer einheitlichen Form darstellen kann. Dies hat zum einen pragmatische Gründe: wenn man für eine bestimmte Aufgabe ohnehin Terme benutzt, weil an einer bestimmten Stelle eben ein Term eingesetzt werden muß, ist es naheliegend, die Termstruktur überhaupt zur Beschreibung der Aufgabe heranzuziehen. NuPRL bietet mächtige Werkzeuge zur Manipulation von Termen, etwa einen Termeditor, außerdem zahlreiche Hilfsfunktionen, die die Benutzung von Termen als Grundstruktur vorteilhaft machen. Ein anderer Grund für die universelle Bedeutung der Terme liegt darin, daß man die Termstruktur auch in der Typentheorie selbst nachbilden kann, und auf diese

[3] Wir weisen darauf hin, daß dies nicht selbstverständlich ist. Man könnte auch diesen Komplex ignorieren und pauschal unterstellen, daß alle Algorithmen terminieren. Wir sollten uns die Freiheit bewußt machen, daß wir Phänomene beachten oder unterschlagen können; dies heißt nämlich, daß wir nicht über Algorithmen *an sich* schließen, sondern nur über eine Modellvorstellung von Algorithmen. Wir sollten uns mit unseren Gegenständen nicht vorschnell identifizieren, sondern versuchen, sie kühl aus der Distanz zu betrachten.

Weise die Möglichkeit in Reichweite rückt, Konzepte der Metaebene der Typentheorie innerhalb der Typentheorie nachzuzeichnen.

Im folgenden werden wir über Terme sprechen müssen. Dabei haben wir einen folgenschweren Unterschied zu beachten, nämlich Terme als Terme oder als Metaterme auffassen zu können. Sich auf die Metaebene zu begeben, heißt, Ausdrücke der ursprünglichen Ebene zum verhandelten Gegenstand zu machen. Im folgenden wollen wir darüber sprechen, wie Terme aufgebaut sind; dies könnte man etwa dadurch erledigen, indem man bestimmte Mengen von Wörtern als Terme auszeichnet. Die Beschreibung als Mengen wäre dann die Metaebene der Terme. Zwar führt diese Methode mit Sicherheit zu einer sehr sauberen Definition, gerade weil auf eine anerkannte Sprache, nämlich die der Mengentheorie, zurückgegriffen wird; allerdings ist festzustellen, daß man sich damit von den Termen insofern sehr weit entfernt, als deren strukturellen Besonderheiten durch die mengentheoretische Ausdrucksweise in den Hintergrund geraten; etwa der rekursive Charakter.

Für unseren Zweck ist es im Grunde ausreichend, Terme nur insoweit zu definieren, als Mehrdeutigkeiten und Variationen geklärt werden; Termstrukturen sind wohlbekannt. (Die bekannte Art, wie in der Mathematik definiert wird, ist eigentlich ohnehin eine extreme Auffassung der Definition. Das definiendum ist dabei nur Abkürzung des definiens; das definiendum geht jeder Eigenart verlustig, weil es stets ohne Verkürzung durch sein definiens ersetzt werden kann. Das Definieren hat dann nur noch den didaktischen Wert, dem Leser einen Zugang zu weisen. Demgegenüber sollte man aber beachten, daß der Leser stets etwas mitbringt und daß es, damit der Leser eine Erkenntnis nachvollziehen und damit selbst haben kann, sehr darauf ankommt, auf diese Zutaten des Lesers einzugehen. Weitergedacht, könnte man sogar sagen, daß Erkenntnis überhaupt mehr darauf beruht, daß Vorhandenes moduliert als daß Neues geschaffen wird.)

Terme sind eine rekursive Struktur, die aus einem Operationsnamen (dem Operator) und einer (geordneten) Liste von Bindungsbereichen besteht; ggf. von zusätzlichen Parametern ergänzt. Die Bindungsbereiche bestehen wiederum aus einer Liste von Bindungsvariablen und einem Term. In der Liste der Bindungsvariablen darf kein Name doppelt vorkommen. – Ein atomarer Term, der keinen weiteren Term enthält, kann dadurch konstruiert werden, daß die Liste der Bindungsbereiche leer ist; der Term besteht dann nur noch aus einem Operator und ggf. Parametern. Ein Term drückt zum einen eine „Klammerstruktur“ aus, da die Anwendungsordnung der Operatoren festgelegt wird, zum anderen eine „Bindungsstruktur“, da Variablen eingeführt werden und jeweils die Reichweite einer Einführung darstellbar ist.

Eine Schreibweise für Terme, die sehr nah an der tatsächlichen NuPRL-Repräsentation liegt, ist die folgende:

$$\widehat{op}\{\hat{p}_1, ..., \hat{p}_m\}(\hat{V}_1.\hat{s}_1; ...; \hat{V}_n.\hat{s}_n)$$

Wir haben die Symbole mit einem „Hut“ versehen, damit klar wird, daß diese Symbole keine Variablen im Sinne des Terms sind, sondern Platzhalter der Metaebene, auf der wir den Aufbau der Terme besprechen. Für $\widehat{op}$ kann der Operatorname eingesetzt werden, für $\hat{p}_1$ bis $\hat{p}_m$ können Parameter eingesetzt werden; für $\hat{V}_1$ bis $\hat{V}_n$ können Listen von Bindungsvariablen eingesetzt werden; für $\hat{s}_1$ bis $\hat{s}_n$ können schließlich Terme eingesetzt werden. In dieser Schreibweise lassen wir den Teil in geschweiften Klammern weg, falls es keine Parameter gibt; desweiteren wird auch der Punkt jeweils weggelassen, wenn es keine Bindungsvariablen gibt.

Wir führen diese Schreibweise nur ein, um eine genaue Notation zu haben und um alle Bestandteile von Termen identifizieren zu können. Zum Lesen komplexer Terme ist diese Schreibweise ungeeignet. Daher benutzen wir parallel dazu eine laxe, sich an überkommene Notationen orientierende Schreibweise, z.B. $3 + 4$ statt $\texttt{add}(3; 4)$. NuPRL unterstützt dies in besonderer Weise, dazu später mehr.

Bestimmte atomare Terme können erst mit Hilfe von Parametern ausgedrückt werden, etwa Konstanten. So hat der Term, der die Konstante 5 beschreibt, wie alle Zahlkonstanten den Operatornamen `natural_number`, sowie als einzigen Parameter den Wert der Konstanten. Eine andere Art von atomaren Termen sind die Variablen; eine durch einen Bindungsbereich eingeführte Variable soll ja auch in seinem Inneren aufgerufen werden können. Dazu wird der Operatorname `variable` verwendet; als einziger Parameter erscheint der Variablenname. Es gilt die Regel, daß sich eine solche Variable, wenn möglich, immer auf die Einführung des innersten Bindungsbereichs bezieht, der sie enthält. Im folgenden meinen wir mit dem *Vorkommen* einer Variablen einen konkreten Term mit Operatornamen `variable`, während wir mit *Variable* häufig alle gleichnamigen Vorkommen eines Bindungsbereichs meinen, was ja heißen soll, das sich diese Vorkommen auf das selbe (hypothetische) Objekt beziehen. Statt der ausführlichen Notation `variable`$\{\hat{v}\}()$ für eine Variable mit dem Namen $\hat{v}$ schreiben wir meist nur kurz den Namen selbst: $\hat{v}$.

Folgende Termini über die Verwendung von Variablen seien vereinbart:

- „*Das Vorkommen $\hat{x}$ einer Variablen wird durch den Bindungsbereich $\hat{V}.\hat{s}$ gebunden*“ heißt: $\hat{x}$ liegt in $\hat{s}$ und $\hat{V}$ ist der innerste Bindungsbereich mit $\hat{x} \in \hat{V}$.
- „*Freies Vorkommen von $\hat{x}$*“ heißt: Kein Bindungsbereich, in dem das Vorkommen $\hat{x}$ liegt, bindet $\hat{x}$.
- *Free($\hat{t}$)*: Menge der freien Variablen des Terms $\hat{t}$, d.h. diejenigen Variablen, die in $\hat{t}$ ein freies Vorkommen haben.
- *Free($\hat{V}.\hat{t}$) = Free($\hat{t}$) $\setminus \hat{V}$*
- „*Geschlossener Term*“ heißt: Alle Vorkommen von Variablen sind in diesem Term gebunden.

Terme werden vor allem verwendet, um Ausdrücke der Typentheorie darzustellen. Dazu müssen jedoch eine Reihe von Bedingungen eingehalten werden: Nur bestimmte Operatoren dürfen verwendet werden. Jeder Operator bestimmt, wieviele Bindungsbereiche eingeführt werden und wieviele Bindungsvariablen jeweils angegeben werden; desweiteren wieviele Parameter benötigt werden und welcher Art diese sein müssen. Auch der Gebrauch der Variablen ist eingeschränkt.

Häufig werden wir den Gebrauch von Termen über den eigentlichen Bereich der Typentheorie hinaus ausdehnen. Dazu werden die Einschränkungen im Aufbau von Termen in der Regel gelockert. Ein Anwendungsgebiet dafür ist, den Aufbau von Termen selbst als Term darzustellen. Beispielsweise ist

$$\lambda x, y.\ a[x] + b[y]$$

ein Term, der im Sinne einer Grammatik den Aufbau möglicher (erzeugbarer) Terme beschreibt. Damit ein Term zu dieser Grammatik paßt, muß er zum einen auf dieselbe Weise aus einem λ-Operator, der eine Summe enthält, bestehen, darf aber anstelle von x und y andere Namen verwenden und anstelle von a und b andere Teilterme. Es gilt jedoch die Einschränkung, daß in a nur die Variable x und in b nur die Variable y frei vorkommen darf (dies sollen die eckigen Klammern ausdrücken). Beispielsweise fällt $\lambda h, g.\ 3 * h + g$ in die Menge der erzeugbaren Terme. Dieser grammatikalisch gemeinte Term ist nun, was wir einen Metaterm nennen, d.h. ein Ausdruck, der über Terme spricht, aber selbst als Term formuliert ist.

Als Metaterm gemeinte Terme benützen meist zusätzliche Operatoren, um die Metastrukturen darzustellen (dies müssen sie allerdings nicht, man könnte darum herumkommen; ein Metaterm wird erst in unserer Auffassung zum Metaterm). Für einen grammatikalischen Term werden etwa *Variablen zweiter Ordnung* zusätzlich verwendet, in unserem Beispiel a und b. Es handelt sich hierbei um Variablen, denen zusätzlich zum Namen eine Liste fester Länge zugeordnet ist: $v^k[s_1, ..., s_k]$. Die Länge k ist die Stelligkeit der Variablen. Variablen kann man in einer formalen und einer aktualen Weise verwenden; in einer formalen Variablen zweiter Ordnung gilt die Einschränkung, daß alle Stellen s_1 bis s_k gewöhnliche Variablen sind. Eine formale Verwendung ist etwa obiges Beispiel, dabei haben die Stellen die Rolle eines Musters für mögliche Instanzen; in einer aktualen Verwendung wird dagegen in den Stellen bezeichnet, welche Terme für welche Bindungsvariable eingesetzt wird.

In einem Term mit *konsistenten Variablen zweiter Ordnung* sind alle gebundenen Variablen solche erster Ordnung und alle Vorkommen von freien Variablen mit demselben Namen haben die gleiche Stelligkeit.

In NuPRL können Variablen zweiter Ordnung auf zwei Arten dargestellt werden. Zum einen verwendet man den Operator `variable` wie bei gewöhnlichen Variablen, speichert aber die Stellen nun in den Teiltermen, die ursprünglich ja nicht vorhanden waren. Diese Methode ist einfach, hat aber den Nachteil, daß die Restriktionen für Terme, die typentheoretisch benutzt werden, verletzt werden. Daher wurde eine zweite Darstellung entwickelt. Diese unterscheidet allerdings zwischen Instanzen und der Instanziierung. Eine Instanz kann man sich denken als den (anonymen) Wert, der für eine Variable zweiter Ordnung eingesetzt werden kann. Die Operation `so_lambda`$(x_1, ..., x_n.s)$ bezeichnet die Instanz einer Variablen zweiter Ordnung mit n Stellen; dabei soll der Term s für die Variable zweiter Ordnung eingesetzt werden, wobei in s aber die Variablen x_1 bis x_n vorkommen dürfen. Demgegenüber entspricht die Instanziierung einem formalen Muster oder aktualen Beispiel, wie die Variable verwendet wird. Die Operation `so_apply`$(s; t_1; ...; t_n)$ beschreibt die Instanziierung der Instanz s, die n Stellen haben muß, mit den konkreten Stellentermen t_1 bis t_n. Um eine Variable a formal zu verwenden, benutzt man nun einen Term der Form `so_apply`$(a; s_1; ...; s_k)$; dabei sind $a, s_1, ..., s_k$ jeweils gewöhnliche Variablen. Bei einer aktualen Verwendung kann man dagegen für s_1 bis s_k beliebige Terme einsetzen. Es ist nun in dieser Darstellung von Variablen zweiter Ordnung besonders einfach, von der formalen Verwendung zur aktualen überzugehen und den Instanziierungsprozeß zu beschreiben. Dazu setzt man nämlich für a lediglich eine Instanz ein, und erhält:

`so_apply`(`so_lambda`$(x_1, ..., x_k.s); s_1; ...; s_k)$

Die operationale Semantik dieses Konstrukts besteht darin, daß die freien Variablen x_1 bis x_k in s durch die Terme s_1 bis s_k ersetzt werden (s.u.). Ersetzt man also auch die s_1 bis s_k durch die aktualen Terme, ergibt sich durch die Substitution sofort der Term, der durch die Variable umschrieben wurde.

Die Operatoren `so_lambda` und `so_apply` können nun innerhalb der Typentheorie definiert werden (s.u.). Dadurch können nun grammatikalische Terme überall verwendet werden, beispielsweise bei der Formulierung eines Theorems, das nun so gedeutet wird, daß es die Grammatik von Theoremen angibt. Um ein solches höheres Theorem anzuwenden, muß man zunächst alle Variablen zweiter Ordnung instanziieren und erhält damit erst das eigentliche Theorem im Sinne der Typentheorie. Beispielhaft sei das Typisierungstheorem für den λ-Operator genannt:

$\forall s$: Term vom Typ B in Abhängigkeit einer Stelle vom Typ A.
 $(\lambda x.\, s[x]) \in A \to B$

Dabei ist s eine Variable zweiter Ordnung, die hier als Platzhalter für einen Term aufgefaßt wird, dessen erste Stelle vom Typ A ist und der als ganzes vom Typ B ist. Wird ein solcher Term in das λ-Konstrukt eingesetzt, entsteht ein Term, der ein Funktionraum von A nach B ist. Das Detail $s[x]$ dieses Theorems kann man nun durch `so_apply`$(s; x)$ darstellen. Instanzen für s werden durch den `so_lambda`-Operator angegeben; so wird durch Ersetzen von s durch `so_lambda`$(y.\, y+y)$ das konkrete Theorem $(\lambda x.\, x + x) \in \mathbf{Z} \to \mathbf{Z}$ generiert.

Die obige Standardnotation für Terme berücksichtigt in keiner Weise, daß es geläufige Schreibweisen für bestimmte Terme gibt. So möchte man statt

`all`$(\hat{T}; \hat{x}.\hat{P})$

lieber die gewohnte Notation

$$\forall \hat{x}{:}\hat{T}.\ \hat{P}$$

lesen. NuPRL bietet die Möglichkeit, für jeden Operator eine *Display-Form* zu vereinbaren, die beschreibt, auf welche Weise ein Term angezeigt werden soll. In die Display-Form können auch Klammer- und Umbruchregeln eingebaut werden. Im folgenden verwenden wir so weit wie möglich die Display-Form eines Terms.

Display-Formen machen Terme zwar lesbarer, haben aber auch den Nachteil, daß manchmal wichtige Details unsichtbar oder undeutlich bleiben. Beispielsweise haben $\forall x{:}\hat{T}.\ \hat{P}$ und $\forall x \in \hat{S}.\ \hat{B}$ trotz ihrer großen Ähnlichkeit verschiedene Operatoren (`all` bzw. `fset_all`), die zudem eine völlig unterschiedliche Semantik besitzen. Beispielsweise wird das Symbol „$\in$“ in mannigfaltiger Bedeutung verwendet. Natürlich ist es so, daß ein geübter Leser diese feinen Differenzen aus dem Kontext der Formel ersehen kann, in dem sie vorkommt, aber nichtsdestotrotz lassen sich alle Details erst klären, wenn ein Term in seiner strukturellen Form dargestellt wird. Dies ist auf dem Papier manchmal schwierig, und der Leser lasse sich bitte nicht dadurch verwirren, daß von einem bestimmten Operator gesprochen wird, dieser Name in der angegebenen Formel aber gar nicht vorkommt. NuPRL bietet dem Benutzer übrigens an, einzelne Terme statt in Display-Form in struktureller Form anzuzeigen, so daß interaktiv alle Unklarheiten beseitigt werden können[4].

2.1.2 Substitutionen

Eine *Substitution erster Ordnung* ist eine Abbildung σ von Variablen auf Terme. Die Substitution wird kanonisch wie folgt auf Terme fortgesetzt:

- $\bar{\sigma}(x) = \sigma(x), \qquad x$ Variable
- $\bar{\sigma}(op\{p_1, ..., p_m\}(V_1.s_1; ...; V_n.s_n)) = op\{p_1, ..., p_m\}(V_1'.\sigma_1(s_1); ...; V_n'.\sigma_n(s_n))$

  wobei
$$\begin{aligned} V_i' &= \{\mu_i(v) \mid v \in V_i\} \\ \mu_i(v) &= \begin{cases} \text{neue Variable} & , v \in \bigcup\{Free(\sigma(v)) \mid v \in V_i\} \\ v & ,\text{sonst} \end{cases} \\ \sigma_i(v) &= \begin{cases} \mu_i(v) & , v \in V_i \\ \sigma(v) & , v \notin V_i \end{cases} \end{aligned}$$

Eine konkrete Substitution $\bar{\sigma}(t)$ erster Ordnung notieren wir auch als

$$t\ [t_1, ..., t_k/v_1, ..., v_k]_1$$

wobei $\sigma(v_i) = t_i$ und für alle ungenannten Variablen v gilt, daß $\sigma(v) = v$.

Eine *Substitution zweiter Ordnung* ist eine Abbildung σ von Variablen (zweiter Ordnung) auf Bindungsbereiche[5]. Die Fortsetzung auf Terme mit konsistenten Variablen zweiter Ordnung lautet:

[4] Dieser Mechanismus könnte noch verbessert werden. Insbesondere sollte es möglich sein, Zwischenstufen zwischen struktureller und Display-Form zu wählen, in denen soviel Information mitgeteilt wird, wie gerade gewünscht wird. Z.B. sollte es möglich sein, Terme grundsätzlich vollgeklammert anzuzeigen oder die Operatornamen aller Teilterme auszugeben. Manche Operatoren, die zu einem der weiter unten eingeführten ADTs gehören, besitzen zusätzliche Typinformationen. Es sollte möglich sein, diese optional in der Darstellung zu unterdrücken. Hier könnte ein Mechanismus helfen, der es erlaubt, global eingestellte Schalter innerhalb der Definition einer Display-Form abzufragen, und jenachdem den Term angemessen anzuzeigen.

[5] In NuPRL werden Bindungsbereiche zumeist als `so_lambda`-Terme dargestellt.

- $\bar{\sigma}(x^n(s_1;...;s_n)) = \begin{cases} t\,[s_1,...,s_n/v_1,...,v_n]_1 & ,\sigma(x) = v_1,...,v_n.t \\ \text{undefiniert} & ,\text{sonst} \end{cases}$

- $\bar{\sigma}(op\{p_1,...,p_m\}(V_1.s_1;...;V_n.s_n)) = op\{p_1,...,p_m\}(V_1'.\sigma_1(s_1);...;V_n'.\sigma_n(s_n))$

  wobei
$$\begin{aligned} V_i' &= \{\mu_i(v) \mid v \in V_i\} \\ \mu_i(v) &= \begin{cases} \text{neue Variable} & , v \in \bigcup\{Free(\sigma(v)) \mid v \in V_i\} \\ v & ,\text{sonst} \end{cases} \\ \sigma_i(v) &= \begin{cases} \{\}.\mu_i(v) & , v \in V_i \\ \sigma(v) & , v \notin V_i \end{cases} \end{aligned}$$

Eine konkrete Substitution zweiter Ordnung $\bar{\sigma}(t)$ notieren wir auch als

$$t\,[V_1.t_1,...,V_k.t_k/v_1,...,v_k]_2$$

wobei $\sigma(v_i) = V_i.t_i$ und für alle ungenannten Variablen v gilt, daß $\sigma(v) = v$.

2.1.3 Typentheorie in Kürze

Wir versuchen uns an einer superkurzen Einführung in die Typentheorie. Wir können dabei nicht alle Aspekte behandeln, insbesonder lassen wir semantische Fragen völlig aus und vertrauen auf die intuitive Verständlichkeit.

Tabelle 2.1 zeigt alle Operatoren der Typentheorie. Es ist dabei zwischen Typ- und Elementoperatoren zu unterscheiden. Typoperatoren, die in der ersten Zeile jedes Kastens stehen, der einen Typ beschreibt, geben an, wie ein Typ aufgebaut wird, ggf. durch Zusammensetzung aus weiteren Typen. Die Metavariablen S und T in der Tabelle bedeuten, daß an dieser Stelle Typterme eingesetzt werden können. Der insgesamt erhaltene Typterm ist nun eine Art Konstruktionsanleitung für die Elemente, die der Typ enthält. Beispielsweise entsteht ein Element zu einem Typ der Form S list nur durch die Elementoperatoren, die für Listen vorgesehen sind; dies sind die Leere Liste und die Konstruktion einer Liste, indem einer bestehende Liste ein Glied vorangestellt wird. Im Gegensatz zur Mengentheorie müssen die Elemente eines Typs stets durch eine endliche Zusammensetzung der für die beteiligten Typkonstrukte vorgesehenen Elementoperatoren konstruierbar sein. – Die zu jedem Typ gehörigen Elementoperatoren sind in den eingerückten Spalten dargestellt. Links stehen die kanonischen Elementoperatoren, rechts die nichtkanonischen. „Kanonisch" heißt, daß Terme, die zuäußerst einen solchen Operator haben, einen zum jeweiligen Typ passenden *Wert* repräsentieren. Werte bilden nach außen hin die Einheit, die das Ergebnis einer Berechnung darstellt, auch wenn man sie in ihrem Inneren noch weiterhin auswerten kann. So ist z.B. $\lambda x.\ x * (1+2)$ ein Wert, obwohl man $1+2$ noch ausrechnen könnte. Dies ist reine Setzung; es ist möglich, daß Werte unterschiedlicher Gestalt dasselbe Element des Typs repräsentieren können[6]. Nichtkanonische Operatoren haben die Aufgabe, die Einheit von Werten aufzubrechen und mit ihren Bestandteilen Berechnungen zu beschreiben. So ist z.B. $(\lambda x.\ x * (1+2))\ 3$ ein Programm, das vorschreibt, die 3 für das x einzusetzen, d.h. nach einem Berechnungsschritt erhält man $3 * (1+2)$, was man nun weiter auswerten kann. Zusammengefaßt fassen kanonische Operatoren die Bestandteile für das Ergebnis einer Berechnung zusammen, während nichtkanonische Operatoren diese Bestandteile in neue Berechnungen einsetzen.

Tabelle 2.2 (Seite 51) beschreibt alle Möglichkeiten, wie nichtkanonische Operatoren Werte aufbrechen und ihre Bestandteile verrechnen können. Diese Tabelle zeigt allerdings nicht, welche Auswertungsstrategie in der Typentheorie benutzt wird; dies heißt, welche Reihenfolge benutzt wird,

---
[6]Daß ein solcher Unterschied gemacht werden muß, liegt in der Natur der Funktionenräume: Das gleiche extensionale Verhalten zweier Funktionen kann durch völlig unterschiedliche Berechnungsvorschriften erreicht werden. In der Typentheorie wird dieser Unterschied prinzipiell auf alle Typen erweitert.

wenn mehrere nichtkanonische Operatoren zugleich die Rechnung fortführen könnten. Zwar ist es stets möglich, innerhalb eines Beweises an irgendeiner Stelle auszuwerten, wobei der Wert der Berechnung, wenn es ihn gibt und er erreicht wird, derselbe bleibt. Man begibt sich jedoch in das Risiko von Endlosschleifen, die bei willkürlicher Auswertung auftreten können. Um eine Aussage über die Terminierung von Berechnungen machen zu können, legt die Typentheorie die Auswertungsstrategie „lazy evaluation" fest, die im wesentlichen besagt, daß kanonische Terme im Inneren nicht ausgewertet werden und daß bestimmte Teilterme (die *Hauptargumente*) vor den anderen ausgewertet werden müssen. Hauptargument ist immer der Teilterm einer nichtkanonischen Operation, in dem der zugegörige kanonische Operator erwartet wird, so daß ein Berechnungsschritt (auch: Reduktion, Kontraktion) möglich wird. Die Möglichkeit, auf diese Weise rechnen zu können, macht die Typentheorie von vornherein zu einer Programmiersprache, wenn auch eine mit sehr einfachen Operatoren.

Sowohl Elemente als auch Typen werden dargestellt, indem Terme zu ihrer Konstruktion angegeben werden. Beide Bereiche werden nicht syntaktisch getrennt; auf semantischer Seite existiert eine Hierarchie, in der die einfachen Elemente die unterste Stufe einnehmen; darauf folgen die einfachen Typen. Es gibt nun die Möglichkeit der Typbildung, indem Typen als Elemente höherer Typen aufgefaßt werden. Auf diese Weise ergibt sich eine beliebig fortsetzbare Hierarchie von *Universen*, wie die Gesamtheit aller Typen auf einer Ebene genannt wird.

Von Termen ist syntaktisch nicht ablesbar, ob sie Elemente oder Typen konstruieren; ja es ist noch nicht einmal ablesbar, ob der Term eine konsistente Konstruktionsvorschrift liefert. Wenn ein Term t gegeben ist, müssen wir daher erst nachweisen, daß er für einen Typ steht oder Element eines bestimmten Typs ist, in Zeichen $t \in T$. Durch die Universenhierarchie kann diese Notation sowohl für den Fall benutzt werden, daß t ein Element und T ein Typ (eines Universums) ist als auch dafür, daß t ein Typ und T ein Universum ist. Typ-Sein wird so als Frage nach dem Element-eines-Universums-Sein aufgefaßt. In dieser Hierarchie spielen Universen eine wichtige Rolle: das erste Universum $\mathbf{U}$ oder $\mathbf{U}_1$ enthält alle Typen, die keine Typen als Elemente haben; das Universum $\mathbf{U}_i$ enthält alle Typen, die das das Universum $\mathbf{U}_{i-1}$ *selbst* enthalten sowie alle Typen, die *in* $\mathbf{U}_{i-1}$ liegen. Z.B. gilt $\mathbf{U}_1 \times \mathbf{Z} \in \mathbf{U}_2$, wobei $\mathbf{U}_1 \times \mathbf{Z}$ ein Typ ist, der Paare aus einem Typ des ersten Universums und einer Zahl als Elemente enthält, etwa $\langle \mathbf{Atom}, 1 \rangle$. – Die Typzugehörigkeitsrelation $\cdot \in \cdot$ kann, wie gesagt, nicht durch syntaktische Kriterien entschieden werden. Aus diesem Grund muß dieser Aspekt der Sprache auf Beweisebene behandelt werden, d.h. wenn in einem Beweisschritt Annahmen über Typzugehörigkeit gemacht werden, müssen diese Annahmen explizit bewiesen werden.

Typen haben wir bis jetzt mehr oder weniger als Strukturierungsweise von Elementen aufgefaßt, was eng an programmiersprachliche Typkonzepte angelehnt ist. So ist der Produktraum $S \times T$ ein Typ, der als Struktur seiner Elemente vorgibt, Paare zu sein, d.h. durch den kanonischen Operator für Paare darstellbar zu sein. Es ist jedoch noch eine andere Auffassung bedeutend, die als Curry-Howard-Isomorphie bekannt ist. Hier wird nicht mehr primär nach der Struktur selbst gefragt, sondern nur danach, ob ein Typ mit einer bestimmten Strukturierungsweise überhaupt Elemente hat. Hat er diese, identifizieren wir diesen Typ mit logischer Wahrheit, ansonsten mit logischer Falschheit. Es ist dabei zu beachten, daß so definierte Wahrheit und Falschheit immer noch Typstruktur mit sich trägt, die wir jedoch ignorieren. Machen wir ein Beispiel: Der schon genannte Produktraum $S \times T$ hat genau dann Elemente, wenn die Typen S *und* T Elemente haben, da man, um ein Paar darzustellen, immer ein Element aus S und ein Element aus T benötigt. Den Produktraum kann man daher als eine Konjunktion deuten. Anders der Raum der disjunkten Vereinigung $S \mid T$: Er hat genau dann Elemente, wenn S *oder* T Elemente haben, da man, um ein Element aus $S \mid T$ darzustellen, nur entweder eines aus S oder eines aus T braucht. Es zeigt sich, daß man auf diese Weise alle logischen Operatoren simulieren kann und daß die Schlußregeln der Typentheorie ebenfalls als logische Schlußregeln interpretiert werden können; dabei ist natürlich zu beachten, daß dies nur funktioniert, weil die Randfälle der Elementkonstruktion sich eben isomorph zu den logischen Junktoren verhalten[7]. Da die Typentheorie konstruktiven Charakter hat, ergibt sich auf diese Weise eine intuitionistische Logik.

Wir verwenden im weiteren die Zeichen $\mathbf{F}, \mathbf{T}, \wedge, \vee, \neg, \Rightarrow, \Leftarrow, \iff, \forall, \exists$ für logische Operatoren, ohne anzugeben, wie sie simuliert werden. Die Typen $\mathbf{P}_i$ stehen für als Proposition gedeutete Typen, dabei gilt $\mathbf{P}_i \equiv \mathbf{U}_i$, d.h. $\mathbf{P}_i$ besteht aus allen Typen des i-ten Universums, insofern sie als Wahrheit (Elemente besitzend) oder Falschheit (keine Elemente besitzend) betrachtet werden. Einige der Typen aus Tabelle 2.1 sind bereits als Proposition intendiert, nämlich $u < v$ oder $u = v \in T$. Dies sind „künstliche" Typen, die entweder nur das Element „Axiom" enthalten oder leer sind. „Axiom" hat keinen weiteren Sinn außer dem, Element eines dieser Typen zu sein. Die Gleichheit $u = v \in T$ hat eine doppelte Bedeutung: Erstens steht sie für die Aussage, daß u und v gleich sind; zweitens steht sie für das Urteil, daß u und v dem Typ T angehören. Wir können $u \in T$ als Kurzform von $u = u \in T$ auffassen.

Die Typentheorie trennt strikt die Begriffe Proposition (Aussage) und Urteil. Der Unterschied ist der, daß ein Urteil $\vdash P$ auf einer anderen Ebene angesiedelt ist als die Proposition P. Während eine Proposition dadurch gekennzeichnet ist, der Zuordnung von wahr (beweisbar) oder falsch (widersprüchlich) fähig zu sein, drückt ein Urteil aus, daß eine Wahrheit evident ist, also daß eine Konstruktion für ein Element eines Typs (einer Proposition) existiert. Genauer gesagt, hat daher ein Urteil immer die Gestalt[8] $\vdash t \in P$, d.h. man muß immer angeben, welches Element t die Proposition P wahr macht. Aus pragmatischen Gründen spaltet man dieses Urteil jedoch in einen Behauptungsteil $\vdash P$, mit dem bereits ein Beweis angesetzt werden kann, und in die Evidenz t. Ein Beweis besteht (grundsätzlich) darin, eine Reihe von Urteilen zu bilden, die durch Schlüsse zustande kommen, d.h. durch einen Schluß wird die Richtigkeit eines Urteils A auf die Richtigkeit eines Urteils B zurückgeführt. Nun ist es sehr schwierig, gleich zu Anfang des Beweises schon die Evidenzen zu wissen; daher kann man die allermeisten Beweise[9] dadurch erbringen, daß man lediglich den Behauptungsteil der Urteile in den Schlüssen berücksichtigt. Ist der Beweis vollständig, kann man die Schlußketten in umgekehrter Richtung durchgehen und die Urteile durch die Evidenzen komplettieren.

Neben $\vdash P$ ist nun auch das Urteil $\vdash P \in \mathbf{P}_i$ möglich, das behauptet, daß P überhaupt eine Aussage ist. Der Term $P \in \mathbf{P}_i$ enthält dabei zweierlei Anteil: Erstens, daß P mit sich selbst identisch ist (was eine Aussage ist, wenn P eine Proposition ist, daher gilt $(P \in \mathbf{P}_i) \in \mathbf{P}_{i+1}$), und zweitens, daß P eine Proposition i-ter Ordnung ist, was nur als Urteil interpretierbar ist, ansonsten würden wir mittels einer Proposition einem Term zuschreiben, Proposition zu sein, wodurch ein definitorischer Zirkel entstünde. Urteile sind daher das grundlegendere Konzept (und das eingeschränktere Konzept, da man nicht in einem Urteil über Urteile reden kann, wodurch ein weiterer Zirkel unmöglich ist).

Ein Beweis der Typentheorie ist ein Baum, in dessen Knoten hypothetische Urteile als Sequenzen dargestellt sind. Ein hypothetisches Urteil hat die Form $H_1, ..., H_n \vdash C$ und bedeutet, daß C, die *Konklusion* behauptet wird, falls alle *Hypothesen* $H_1, ..., H_n$ wahr sind. C ist dabei eine Aussage (ein Typ), während die Hypothesen H_i Variablen als Typen deklarieren, von denen angenommen wird, daß sie nicht leer sind. $x : T \vdash P$ heißt also, daß die Proposition P behauptet wird (bzw. die Nicht-Leere des Typs P), falls der Typ T nicht leer ist, wobei x für ein beliebiges Element dieses Typs steht. (Falls T als Proposition gedeutet wird, sind konkrete Elemente uninteressant; man läßt in diesem Fall die

[7]Die Beobachtung von Curry und Howard ist also, daß man diese Randfälle so einrichten *kann*, daß sie den logischen Operatoren entsprechen, nicht aber, daß diese Randfälle zwingend zu der Semantik der beteiligten Typen gehören.

[8]Die Typentheorie betrachtet dies zwar als Sonderfall $t = t \in P$ des allgemeineren Gleichheitsurteils, doch ist in dieser elaborierteren Auffassung der Kern des Urteils, die Evidenz der Wahrheit auszudrücken, von der Gleichheitssemantik überlagert und nicht mehr so klar sichtbar. Man sollte vielmehr ausnutzen, daß der intuitionistische Urteilsbegriff relativ einfach einsichtig gemacht werden kann, weil die Evidenz ausgedrückt werden kann.

[9]In meiner Studienarbeit [stolpmann96] habe ich auf Seite 48 tatsächlich einen Fall gefunden, bei dem man die Evidenz a priori wissen muß. Dies ist aber die absolute Ausnahme.

Variable auch weg: $T \vdash P$.)

Einen Beweisbaum notieren wir so:

$$
\begin{array}{l}
H_1, ..., H_n \vdash C \\
\quad \text{BY Schlußregel} \\
\qquad J_{1,1}, ..., J_{1,m_1} \vdash D_1 \\
\qquad \vdots \\
\qquad J_{p,1}, ..., J_{p,m_p} \vdash D_p
\end{array}
$$

Dies ist zu lesen: Das Urteil $H_1, ..., H_n \vdash C$ ist zutreffend, falls alle Urteile $J_{i,1}, ..., J_{i,m_i} \vdash D_i$ zutreffend sind. Die Korrektheit dieses Schlusses wird durch die Korrektheit der Schlußregel abgesichert. Die Unter-Urteile (auch Untersequenzen, Unterziele) sind jeweils wiederum durch Beweisbäume zu zeigen.

Ein Beweis eines Urteils ist erbracht, wenn der Beweisbaum abgeschlossen worden ist, d.h. es gibt keine Untersequenz mehr, die nicht durch einen Schluß nachgewiesen wurde.

Zur Typentheorie gehört ein Satz Schlußregeln, die korrekt relativ zur Semantik der Typentheorie sind. Die Menge dieser Regeln ist derart umfangreich, daß wir sie hier unmöglich alle erläutern können. Abbildung 2.3 (Seite 53) gibt die acht Regeln, die speziell für den Produktraum vorgesehen sind, an. In diesem Schema stehen die Variablen Γ und Δ für beliebige Hypothesenlisten, die Variablen i und j, die als Argument mancher Regeln vorkommen, geben die Nummer der zu benutzenden Hypothese bzw. das Universenlevel an. Die anderen Metavariablen erschließen sich aus dem Schema. Zu weiteren Erklärungen dieser Regeln siehe unten.

Neben Regeln, die speziell für einen Typ zugeschnitten sind, gibt es noch einige allgemeine Regeln, etwa `hypothesis`, die triviale Urteile beweist, oder `cut`, welche einen Schnitt in den Beweis einführt. Die Bedeutung dieser Regeln ist sehr einsichtig, so daß wir auf eine Erläuterung verzichten.

2.2 Grundlinien einer Programmiersprache

Die Typentheorie stellt nur die Primitiven des Programmierens zur Verfügung, nicht jedoch problemadäquate Konstrukte. Mittels *Abstraktionen* können wir diesen Mangel ausbügeln. Dabei gibt es zum einen die technische Seite, den *Abstraktionsmechanismus*, zum anderen eine gestalterische Seite, da der Mechanismus weitgehende Entwurfsfreiheiten läßt. Beide Seiten werden wir diskutieren müssen.

Neben der Bereitstellung angemessener Konstrukte bleibt noch die Frage, auf welche Weise die dann erweiterte Sprache Objekt unserer Fragestellungen wird. Es zeigt sich, daß sich die Aspekte des Rechnens, des Typisierens und des Schließens als relevant herausstellen. Wir werden an einem Beispiel demonstrieren, inwiefern diese Aspekte in bestimmte Entscheidungen beim Entwurf der Sprache eingehen.

2.2.1 Abstraktion

Die primitiven Operatoren sind eine recht bescheidene Ausstattung für eine Programmiersprache. Sie erlauben im Wesentlichen nur die minimal notwendigen Operationen, um Werte des jeweiligen Typs zu bilden oder um mit den Werten zu rechnen. Solcher Minimalismus ist aus Sicht des Autors des Kalküls erstrebenswert, da die Eigenschaften des Kalküls schneller nachgewiesen werden können und der Kalkül konzeptuell durchsichtiger wird. Als Programmierer verlangen wir jedoch, die Sprache so reichhaltig an Konstrukten machen zu können, daß das Niveau dieser Konstrukte nicht zu stark unter dem Niveau der Probleme liegt, die wir programmieren wollen. Die NuPRL-Implementierung der Typentheorie bietet die Möglichkeit, konservative Erweiterungen vorzunehmen, d.h. neue Operatoren

Kanonische Element-Operatoren		Nichtkanonische Element-Operatoren	
Z *(Typ der ganzen Zahlen)*			
n:	Konstante n	$-u$:	Negation
$-n$:	Konstante $-n$	$u+v$, $u-v$,	
		$u*v$, $u \div v$:	Grundrechenarten
		u rem v:	Divisionsrest
		if $u=v$ then s else t:	Konditional
		if $u<v$ then s else t:	Konditional
		rec-case u of	
		$x<0 \mapsto r.s \mid$	
		$0 \mapsto b \mid$	
		$x>0 \mapsto q.t$:	Induktor
$u<v$ *(Kleiner-Proposition)*			
Axiom:	Beweisbarkeit		
void *(Leerer Typ, Falschheit)*			
		any(e)	Unmöglichkeit
Atom *(Typ der Symbole)*			
"Symbol":	Konstante	if $u=v$ then s else t	Konditional
$\mathbf{U_j}$ *(Universum)*			
• Alle Typoperatoren außer Universum			
• Universen $\mathbf{U_i}$ mit $i<j$			
$x{:}S \to T$ *(Funktionenraum)*			
$\lambda x.t$:	λ-Abstraktion	$s\ t$:	λ-Applikation
$x{:}S \times T$ *(Produktraum)*			
$\langle s,t\rangle$:	Paar konstruieren	let $\langle x,y\rangle = u$ in v	Paar zerlegen
$S \vert T$ *(disjunkte Vereinigung)*			
inl(s):	Linkes Element	case s of inl$(x) \mapsto u \mid$	
inr(s):	Rechtes Element	inr$(y) \mapsto v$:	Summe zerlegen
$s=t \in T$ *(Gleichheitsproposition)*			
Axiom:	Beweisbarkeit		
T list *(Liste)*			
$[]$:	Leere Liste	rec-case s of	Listeninduktor
$x{::}l$:	Liste konstruieren	$[] \mapsto b \mid x{::}l \mapsto r.t$	
$\{x{:}S \mid T\}$ *(Teiltyp)*			
• Elemente von S			
$x,y{:}T//E$ *(Quotiententyp)*			
• Elemente von T			
rectype $X = T_X$ *(induktiver Typ)*			
• Elemente gemäß T_X		let* $f(x)=t$ in $f(u)$:	Induktor
$S \not\to T$ *(partieller Funktionenraum)*			
letrec $f(x)=t$:	Funktionsabstraktion	dom(f):	Domain-Proposition
		$f(t)$:	Funktionsapplikation

Tab. 2.1. *Die primitiven Operatoren der Typentheorie*

zu vereinbaren, die konkrete Terme abkürzen. Die NuPRL-Terminologie nennt diese Erweiterungen auch *Abstraktionen*. (Der Name Abstraktion deutet freilich mehr auf den Zweck hin als auf das, was er bezeichnet.) In der Definition einer Abstraktion dürfen Metavariablen erster oder zweiter Ordnung verwendet werden (s.u.).

Neben dem technischen Hilfmittel der Abstraktionen, das weiter unten genauer erläutert wird, ist aber vor allem interessant, *welche* Terme abstrahiert werden sollen. Aus Sicht des Kalküls ist dies natürlich beliebig, da es sich nur um einen Abkürzungsmechanismus handelt. Wir sind daher weitgehend frei, auf welche Weise Abstraktionen gebildet werden. Denkbar sind z.B. folgende Kriterien:

- Die Begriffe einer schon bekannten Theorie werden zur Orientierung herangezogen.
- Intuitivität: Es werden solche Abstraktionen gebildet, die gut verstanden werden können.
- Operatibilität: Die Abstraktionen werden so gebildet, daß sie von einem ML-Programm (z.B. einer Taktik) möglichst gut verarbeitet werden können.
- Systematik: Es wird eine bestimmte Systematik beim Aufbau eines Abstraktionengefüges verfolgt.
- Kürze: Das Abstraktionengefüge soll möglichst klein sein.

Diese Kriterien sind nicht alle voneinander unabhängig. So werden in bereits aufgestellten Theorien selbstverständlich Intuitivität und Systematik berücksichtigt. Wirklich neu ist das Kriterium der Operatibilität, das nicht zu unterschätzen ist. Beispielsweise kann es von der Art der Abstraktionsbildung abhängen, wie gut die Autotaktik mit entsprechenden Wohlgeformtheitsbeweisen zurechtkommt.

Ein weiteres Kriterium erweist sich als ausgesprochen starke Idee, wie wir noch sehen werden:

- Abstraktheit eines Abstraktionensystems

Wie gesagt versteht NuPRL unter „Abstraktion“ wirklich nur einen Mechanismus; es ist also noch lange nicht gesagt, daß die abgekürzten Terme tatsächlich in einem weitergehenden Sinn abstrakt sind. Leider gibt es keinen wirklich allgemein anerkannten Begriff der Abstraktion, auf den wir uns hier berufen könnten; er scheint insbesondere sehr stark mit dem jeweils verfolgten erkenntnistheoretischen Ansatz verschränkt zu sein. Um eine nähere Vorstellung zu bekommen, was „abstrakt“ heißen könnte, müssen wir diesbezüglich Hypothesen annehmen.

Zu den Grundhypothesen dieser Arbeit gehört es, die typentheoretische Formalisierung stets als die Nachbildung eines vorgängigen Konzepts aufzufassen. So wäre etwa die typentheoretische Ausprägung der ganzen Zahlen nicht selbst als Zahlkonzept zu deuten, sondern als eine Struktur, die den Zahlen deshalb gleichkommt, weil wir als Interpreten diese Struktur als Zahlen erkennen können; d.h. diese Struktur ist nur Zahlkonzept, weil wir es als isomorph zu unserer bisherigen Vorstellung von Zahl begreifen.

Der Abstraktionsmechanismus ist nun nur ein Mittel der Formalisierung. Wir wollen aber nicht jeden Einsatz dieses Mittels als gleich gut gelten lassen. Eine Idee ist nun, die Wahl von Abstraktion zu bevorzugen, die der eigenen Struktur des vorgängigen Konzeptes folgt; beispielsweise, indem diejenigen Bezeichnungen, die auch zur Beschreibung des Konzepts benutzt werden, in die Abstraktion übertragen werden.

Nun ist es so, daß man bei einer Formalisierung in der Regel zusätzliche Annahmen machen muß, z.B. eine konkrete Repräsentation wählen muß. Unser Maßstab ist aber das vorgängige Konzept, und darin sind diese Annahmen nicht enthalten. Wenn nun abstrakt heißt, daß wir dem Eigensinn des Konzepts folgen wollen, soll dies nun auch implizieren, daß diese zusätzlichen Annahmen durch die Wahl der Abstraktion verborgen werden. Abstrakt heißt dabei letztlich, daß wir als Interpreten in den durch den Abstraktionsmechanismus abgekürzten Termen nicht die Abkürzung sehen wollen, sondern durch die Formalisierung „hindurchblicken“ und in ihnen nur eine Ausdrucksweise des eigentlich „gemeinten“ Konzepts sehen wollen. Daß formalisiert wurde, soll am besten gar nicht wahrgenommen werden; es reicht aber, wenn es gelingt, dies bei der Betrachtung eines Terms ausblenden zu können.

Ein System von Abstraktionen wird häufig um einen bestimmten Typ herum organisiert. Dieser Typ ist der Bezugspunkt der zusätzlichen Annahmen, von denen gesprochen wurde, etwa einer Datenrepräsentation. Wir sprechen dann von einem *abstrakten Typ*, wenn wir diesen Typ als Abkürzung einführen und damit seinen konzeptuellen Charakter deutlich machen. Beispiele dafür sind die Typen *Spezifikation* und *Globalsuchbeschreibung*, sowie die *abstrakten Datentypen*.

2.2.2 Der Abstraktionsmechanismus

Durch eine Gleichsetzung der Art

$$\hat{op}\{\hat{p}_1, ..., \hat{p}_m\}(\hat{V}_1.\hat{s}_1; ...; \hat{V}_n.\hat{s}_n) \quad \equiv \quad \hat{t}$$

wird der Operator $\hat{op}$ definiert. Wir haben hier einen besonders interessanten Fall eines Metaterms vorliegen. Die Gleichsetzung selbst muß NuPRL nämlich als Term mitgeteilt werden, d.h man gibt einen Term mit dem Operator $\equiv$ ein, der zwei Argumente hat. Das linke Argument ist der Musterterm für den neuen Operator $\hat{op}$, der mit genau so vielen Parametern und inneren Argumenten angegeben wird, wie der neue Operator haben soll. Die Parameter werden wir im folgenden ignorieren; die Argumente dürfen jedenfalls ausschließlich formale Variablen zweiter Ordnung enthalten, die jeweils als Stellen die Bindungsvariablen besitzen. Das rechte Argument des $\equiv$-Terms ist der Term, der vom neuen Operator abgekürzt werden soll. In ihm dürfen die links benutzten Variablen zweiter Ordnung aktual verwendet werden.

Informal bedeutet die Gleichsetzung, daß künftig in Termen der Operator $\hat{op}$ in genau der Weise benutzt werden darf, wie es der Musterterm angibt, und daß diese Benutzung u äquivalent zu dem Term v ist, der entsteht, wenn man die linke Seite gegen u matcht und das Resultat in die freien Variablen der rechten Seite einsetzt. Die Semantik wird durch diese Operation des *Auffaltens* festgelegt.

Nun zu den genauen Bedingungen des $\equiv$-Terms:

- Die Terme, die für die $\hat{s}_i$ eingesetzt werden, müssen (disjunkte) Variablen zweiter Ordnung sein, die formal verwendet werden und die genau von den in $\hat{V}_i$ genannten Bindungsvariablen abhängig sind.
- Der Term, der für $\hat{t}$ eingesetzt wird, muß konsistent für die Variablen zweiter Ordnung sein.
- Alle freien Variablen von $\hat{t}$ müssen in einem der $\hat{s}_i$-Terme vereinbart sein.

Die Auffaltung eines Terms u wird durch den *Unfold*-Metaoperator definiert:

Definition 2.2.1 (*Unfold*-Metaoperator)

Sei für den Operator $\hat{op}$ die Gleichsetzung

$$\hat{op}(\hat{V}_1.\hat{s}_1; ...; \hat{V}_n.\hat{s}_n) \quad \equiv \quad \hat{t}$$

unter Berücksichtigung obiger Bedingungen vereinbart. Dann ist

$$Unfold(\hat{op}(\hat{V}'_1.\hat{s}'_1; ...; \hat{V}'_n.\hat{s}'_n)) = \hat{t}[\hat{V}'_1.\hat{s}'_1, ..., \hat{V}'_n.\hat{s}'_n / v_1, ..., v_n]_2$$

wobei $\hat{s}_i = v_i^{|\hat{V}_i|}[\hat{V}_i]$ Variablen zweiter Ordnung sind.

Es gibt auch die inverse Operation des *Rückfaltens*, die durch den *Fold*-Metaoperator definiert wird. Wir lassen die Einzelheiten aus.

Unfold und *Fold* können auf beliebige Teilterme eines Terms angewandt werden; die diesbezügliche Fortsetzungen der Operationen sind problemlos.

Innerhalb eines typentheoretischen Beweises können die beiden Operationen *Unfold* und *Fold* in beliebiger Weise benutzt werden. Da es sich lediglich um einen Abkürzungsmechanismus handelt, sind so eingeführte neue Operationen semantisch problemlos. Ihre Semantik besteht einfach darin, daß sie identisch ist mit derjenigen des aufgefalteten Terms. Desweiteren ist es so, daß der Abstraktionsmechanismus verträglich ist mit den Schlußregeln der Typentheorie, d.h. es kommt stets auf desselbe hinaus, ob ich nun eine Schlußregel direkt auf eine Sequenz anwende, in der abstrakte Operationen vorkommen, oder ob ich die abstrakten Operationen zuerst komplett auffalte, dann die Schlußregel anwende und zu guter Letzt wieder zurückfalte.

Meistens kann man daher in Beweisen die abstrakten Operationen stehen lassen. Manchmal ist es jedoch notwendig, die Abstraktion aufzuheben; dazu gibt es die speziellen Schlußregeln fold, foldHyp, unfold und unfoldHyp, die das Auffalten und Rückfalten erlauben. Diese Regeln anzuwenden ist meist eine Entscheidung innerhalb des Beweises, die diesem eine komplett neue Richtung gibt; sie sollten stets sehr bewußt eingesetzt werden. Insbesondere sollten Taktiken so programmiert werden, daß sie nicht von selbst Faltungen vornehmen, oder wenn sie es denn tun, auch die entsprechende entgegengesetzte Faltung beinhalten. Der Wechsel der Abstraktionsebene, die mit einer Faltung einhergeht, dürfte für einen automatischen Beweiseransatz so ziemlich die schwierigste Entscheidung sein, die zu treffen ist.

Simulation von Abstraktionen zweiter Ordnung durch solche erster Ordnung

Bei der Definition von Abstraktionen konnten wir den Variablen zweiter Ordnung eine Bedeutung geben, weil der Unfold-Metaoperator dies in sinnvoller Weise zuließ. In Termen, die in Beweisen vorkommen, dürfen keine Variablen zweiter Ordnung benutzt werden, da die Typentheorie diese gar nicht kennt. Manchmal benötigt man innerhalb von Beweisen jedoch einen ähnlichen Effekt, Variablen als Platzhalter für Terme anzusehen, in denen bestimmte Variablen (Stellen) vorkommen können. In der Denkweise der Typentheorie würden wir dies jedoch anders beschreiben: Der „Term“ beschreibt eine funktionale Abhängigkeit, nämlich die Abhängigkeit seines Wertes von seinen „Stellen“. Es gibt also eine Isomorphie zwischen Variablen zweiter Ordnung und Funktionen. Wir sehen uns dies genauer an.

In Abstraktionen haben wir Variablen zweiter Ordnung in zwei Bedeutungen verwendet: als formale Deklaration links und als formale Instanz rechts des $\equiv$-Zeichens. Desweiteren gibt es immer einen Bindungsbereich, der für die Variable bei der Anwendung der Abstraktion eingesetzt wird: Dies ist die aktuelle Instanz.

Die Typentheorie simuliert diese drei Formen durch die Deklaration einer Variablen (erster Ordnung) als Funktion, die (ggf. mehrstellige) Funktionsapplikation und die (mehrstellige) Funktionsabstraktion. Eine Abstraktion, z.B. $\forall x{:}T.\ P^1[x] \ \equiv\ x{:}T \rightarrow P^1[x]$, kann nun bequemer in Beweisen thematisiert werden. Das Theorem

- $(\forall x{:}T.\ P^1[x] \wedge Q^1[x]) \iff (\forall x{:}T.\ P^1[x]) \wedge (\forall x{:}T.\ Q^1[x])$

wird etwa repräsentiert werden, indem P^1 und Q^1 als Funktionen $T \rightarrow \mathbf{P}$ deklariert werden und die Verwendungen $P^1[x]$ bzw. $Q^1[x]$ durch Applikationen $P^1\ x$ bzw. $Q^1\ x$ simuliert werden. Die tatsächliche Gestalt ist somit

- $\forall T{:}\,\mathbf{U}.\ \forall P^1, Q^1{:}\,T \rightarrow \mathbf{P}.\quad (\forall x{:}T.\ P^1\ x \wedge Q^1\ x) \iff (\forall x{:}T.\ P^1\ x) \wedge (\forall x{:}T.\ Q^1\ x)$

Wenn wir dieses Theorem instanziieren wollen, etwa durch die Substitution $[\mathbf{Z},\ \lambda x.x \geq 0,\ \lambda x.x < 0\ /\ T, P^1, Q^1]_1$, so benutzen wir λ-Abstraktionen.

Um mehrstellige Abhängigkeiten zu unterstützen, sind in NuPRL die beiden Operatoren `so_apply` und `so_lambda` vordefiniert:

- `so_lambda:` $\lambda_2 x_1, ..., x_n.\, s^n[x_1, ..., x_n] \;\equiv\; \lambda x_1. ... \lambda x_n.\, s^n[x_1, ..., x_n]$
- `so_apply:` $s\texttt{[}t_1, ..., t_n\texttt{]} \;\equiv\; (\cdots(s\,t_1)t_2\cdots)t_n$

Man beachte, daß die eckigen Klammern in der unteren Zeile zur Display-Form gehören, daß also die Applikation zweiter Ordnung als $s[t_1, ..., t_n]$ geschrieben wird[10]. Um deutlich zu machen, daß mittels einer Variablen Abhängigkeiten zweiter Ordnung simuliert werden sollen, benutzen wir für die Deklaration das Zeichen $\stackrel{2}{\rightarrow}$, wie in $S \stackrel{2}{\rightarrow} T$. (Leider gibt es dafür keinen zusätzlichen Operator, man muß dies als gewöhnlichen Funktionenraum eingeben. In einer zukünftigen NuPRL-Version sollte dies vorgesehen werden, etwa als Operator `so_type`.)

2.2.3 Aspekte des Programmierens

In konventionellen Programmiersprachen geht es fast ausschließlich darum, die Arten des Rechnens in geordneter Weise zu organisieren. Viele Sprachen überprüfen (statisch oder dynamisch) Typbedingungen, was man bereits als einfache automatische Bildung von Inferenzen ansehen kann. Die Programmiersprache, die hier entworfen wird, strebt sogar an, Theoreme über die Wirkung von Programmen aufzustellen und die Inferenztechniken bereitzustellen, diese Theoreme beweisen zu können, und zwar möglichst in automatischer Weise.

Unsere Sprache muß dazu folgende Aspekte des Programmierens unterstützen:

- *Rechnen:* Sie muß natürlich Terme, die Programmtext darstellen, ausrechnen (normalisieren) können.

  Für die primitiven Operationen der Typentheorie ist dies gegeben. Während eines Beweises kann man mit den Regeln compute, computeHyp, rev_compute und rev_computeHyp beliebige Redizes (Termformen, die durch einen Berechnungsschritt aufgelöst werden können), die innerhalb eines Terms in einer Sequenz vorkommen, kontrahieren. Tabelle 2.2 zeigt, welche dieser *primitiven* Redizes es gibt. Diese Tabelle ist so zu lesen, daß Terme, die eine der Redex-Formen auf der linken Seite haben, durch eine Reduktion (synonym: Kontraktion) in den entsprechenden Kontraktum-Term der rechten Seite umgeformt werden können. Die Tabelle zeigt alle Reduktionsmöglichkeiten und damit auch alle Formen von Berechnungsschritten.

  Natürlich existieren zunächst keine Redizes, die für Operatoren funktionieren, die per Abstraktion definiert wurden. Solche *abstrakten Redizes* sind jedoch ungemein wichtig. Ein Beispiel soll dies illustrieren.

  Für den Produktraum $x{:}S \times T$ gibt es lediglich eine primitive Operation $\langle s, t\rangle$, die ein Element konstruiert, und eine primitive Operation let $\langle x, y\rangle = s$ in u, die ein Element zerlegt. Es ist sinnvoll, die beiden Projektionen `pi1` und `pi2` zu definieren, die die linke bzw. rechte Hälfte eines Paares zurückgeben:

  ▷ `pi1:` $u.1 \;\equiv\;$ let $\langle x, y\rangle = u$ in x
  ▷ `pi2:` $u.2 \;\equiv\;$ let $\langle x, y\rangle = u$ in y

  Die entsprechenden abstrakten Redizes sind $\langle s, t\rangle.1$ und $\langle s, t\rangle.2$, die zu s bzw. t kontrahieren sollen. Um dies zu erreichen, sind die Abstraktionen aufzufalten (mittels Metaoperation *Unfold*), wodurch sich primitive Redizes ergeben, die man dann reduzieren kann (Metaoperation *Compute*):

[10] Hat jemand eine Idee, wie der Wirrwarr der eckigen Klammern vermieden werden kann?

Metavariablen mit besonderer Bedeutung: In der folgenden Tabelle steht π für eine positive Zahlkonstante, μ und ν stehen für eine Zahlkonstante, α und β stehen für Symbolkonstanten. Variablen mit einem „Hut" stehen für Terme.

$(\lambda x.\hat{s})\hat{t}$	$\leadsto \hat{s}[\hat{t}/x]_1$
let $\langle x,y\rangle = \langle\hat{s},\hat{t}\rangle$ in $\hat{u}$	$\leadsto \hat{u}[\hat{s},\hat{t}/x,y]_1$
case inl$(\hat{s})$ of inl$(x) \mapsto \hat{u}$ | inr$(y) \mapsto \hat{v}$	$\leadsto \hat{u}[\hat{s}/x]_1$
case inr$(\hat{t})$ of inl$(x) \mapsto \hat{u}$ | inr$(y) \mapsto \hat{v}$	$\leadsto \hat{v}[\hat{t}/y]_1$
rec-case 0 of $x<0 \mapsto r.\hat{s} \mid 0 \mapsto \hat{b} \mid x>0 \mapsto q.\hat{t}$	$\leadsto \hat{b}$
rec-case π of $x<0 \mapsto r.\hat{s} \mid 0 \mapsto \hat{b} \mid x>0 \mapsto q.\hat{t}$	$\leadsto \hat{t}[\pi,(\text{rec-case } \pi-1 \text{ of } x<0 \mapsto r.\hat{s} \mid 0 \mapsto \hat{b} \mid x>0 \mapsto q.\hat{t})$ $/x,q]_1$
rec-case $-\pi$ of $x<0 \mapsto r.\hat{s} \mid 0 \mapsto \hat{b} \mid x>0 \mapsto q.\hat{t}$	$\leadsto \hat{s}[-\pi,(\text{rec-case } -\pi+1 \text{ of } x<0 \mapsto r.\hat{s} \mid 0 \mapsto \hat{b} \mid x>0 \mapsto q.\hat{t})$ $/x,r]_1$
$-\mu$	$\leadsto$ Der Term, der die Konstante mit Wert $-\mu$ darstellt.
$\mu+\nu$	$\leadsto$ Der Term, der die Konstante mit Wert $\mu+\nu$ darstellt.
$\mu-\nu$	$\leadsto$ Der Term, der die Konstante mit Wert $\mu-\nu$ darstellt.
$\mu*\nu$	$\leadsto$ Der Term, der die Konstante mit Wert $\mu*\nu$ darstellt.
$\mu\div\nu$	$\leadsto$ Der Term, der die Konstante mit Wert $\mu\div\nu$ darstellt.
μ rem ν	$\leadsto$ Der Term, der die Konstante mit Wert μ rem ν darstellt.
if $\mu=\nu$ then $\hat{s}$ else $\hat{t}$	$\leadsto \hat{s}$, falls μ und ν den gleichen Wert darstellen, sonst $\hat{t}$.
if $\mu<\nu$ then $\hat{s}$ else $\hat{t}$	$\leadsto \hat{s}$, falls μ einen Wert kleiner ν darstellt, sonst $\hat{t}$.
rec-case [] of $[] \mapsto \hat{b} \mid x{::}l \mapsto r.\hat{t}$	$\leadsto \hat{b}$
rec-case $\hat{u}{::}\hat{v}$ of $[] \mapsto \hat{b} \mid x{::}l \mapsto r.\hat{t}$	$\leadsto \hat{t}[\hat{u},\hat{v},(\text{rec-case } \hat{v} \text{ of } [] \mapsto \hat{b} \mid x{::}l \mapsto r.\hat{t})/x,l,r]_1$
if $\alpha=\beta$ then s else t	$\leadsto \hat{s}$, falls α und β den gleichen Wert darstellen, sonst $\hat{t}$.
let* $f(x)=\hat{t}$ in $f(\hat{e})$	$\leadsto \hat{t}[\lambda x.\text{let* } f(x)=\hat{t} \text{ in } f(\hat{e}),e/f,x]_1$
(letrec $f(x)=\hat{t}$) $(\hat{u})$	$\leadsto \hat{t}[\text{letrec } f(x)=\hat{t},\hat{u}/f,x]_1$
dom(letrec $f(x)=\hat{t}$)	$\leadsto$ Eine Proposition, die einen Definitionsbereich angibt.

Tab. 2.2. *Redizes und Kontrakta für die Primitiven der Typentheorie*

$$\begin{array}{lcl}
\langle s,t\rangle.1 \overset{\textit{Unfold}}{\leadsto} \text{let } \langle x,y\rangle = \langle s,t\rangle \text{ in } x & \overset{\textit{Compute}}{\leadsto} & s \\
\langle s,t\rangle.2 \overset{\textit{Unfold}}{\leadsto} \text{let } \langle x,y\rangle = \langle s,t\rangle \text{ in } y & \overset{\textit{Compute}}{\leadsto} & t
\end{array}$$

Die Wirkung der abstrakten Reduktion kann also durch Auffaltung und primitive Reduktion simuliert werden. Es ist naheliegend, einen Mechanismus zur Verfügung zu stellen, der diese Simulation automatisiert. Man könnte z.B. für jede abstrakte Operation eine Taktik angeben, die die Simulation durchführt und damit ein Detail der Darstellung für den Benutzer unsichtbar macht.

- *Typisieren:* Die Programmiersprache muß nicht nur Typbedingungen überprüfen, sondern sogar über alle Terme strenge Typaussagen machen.

  Die Überprüfung der Typisierung ist, eine Besonderheit der Typentheorie, in die Inferenzregeln selbst eingebaut. Andere Kalküle prüfen die Typisierung schon bei der Eingabe einer Formel. Der Grund dafür, daß die Typentheorie anders verfährt, liegt darin, daß dadurch die Typisierungsmöglichkeiten wesentlich erweitert werden, also auch Terme behandelt werden können, die ansonsten zurückgewiesen werden müßten. Der Preis dafür ist, daß die Typisierung nun in alle Beweise mit eingebaut werden muß.

  In Abbildung 2.3 sehen wir die primitven Inferenzregeln des Produktraums. Vier von acht Regeln befassen sich mit dem kombinierten Gleichheits-/Typ-Urteil $\vdash s = t \in T$; diese Regeln sind am

$\Gamma \vdash x_1{:}S_1{\times}T_1 = x_2{:}S_2{\times}T_2 \in \mathbf{U}_j$
BY productEq
$\quad\Gamma \vdash S_1 = S_2 \in \mathbf{U}_j$
$\quad\Gamma,\ x' : S_1 \vdash T_1[x'/x_1]_1 = T_2[x'/x_2]_1 \in \mathbf{U}_j$

$\Gamma \vdash \langle s_1, t_1\rangle = \langle s_2, t_2\rangle \in x{:}S{\times}T$
BY pairEq j
$\quad\Gamma \vdash s_1 = s_2 \in S$
$\quad\Gamma \vdash t_1 = t_2 \in T[s_1/x]_1$
$\quad\Gamma,\ x' : S \vdash T[x'/x]_1 \in \mathbf{U}_j$

$\Gamma \vdash \langle s_1, t_1\rangle = \langle s_2, t_2\rangle \in S \times T$
BY pairEq_indep
$\quad\Gamma \vdash s_1 = s_2 \in S$
$\quad\Gamma \vdash t_1 = t_2 \in T$

$\Gamma \vdash \text{let } \langle x_1, y_1\rangle = e_1 \text{ in } t_1$
$\quad = \text{let } \langle x_2, y_2\rangle = e_2 \text{ in } t_2 \in C[e_1/z]_1$
BY spreadEq $z\ C\ x{:}S{\times}T$
$\quad\Gamma \vdash e_1 = e_2 \in x{:}S{\times}T$
$\quad\Gamma,\ s : S,\ t : T[s/x]_1,\ y : e_1 = \langle s, t\rangle \in x{:}S{\times}T$
$\quad\vdash t_1[s, t/x_1, y_1]_1 = t_2[s, t/x_2, y_2]_1 \in C[\langle s, t\rangle/z]_1$

$\Gamma \vdash x{:}S{\times}T$
BY pairI $j\ s$
$\quad\Gamma \vdash s \in S$
$\quad\Gamma \vdash T[s/x]_1$
$\quad\Gamma,\ x' : S \vdash T[x'/x]_1 \in \mathbf{U}_j$

$\Gamma \vdash S \times T$
BY pairI_indep
$\quad\Gamma \vdash S$
$\quad\Gamma \vdash T$

$\Gamma,\ z : x{:}S{\times}T,\ \Delta \vdash C$
BY productE i
$\quad\Gamma,\ z : x{:}S{\times}T,\ x : S,\ t : T[s/x]_1,$
$\quad\Delta[\langle s, t\rangle/z]_1 \vdash C[\langle s, t\rangle/z]_1$

$\Gamma \vdash (\text{let } \langle x, y\rangle = \langle s, t\rangle \text{ in } u) = t_2 \in T$
BY spreadRed
$\quad\Gamma \vdash u[s, t/x, y]_1 = t_2 \in T$

Abb. 2.3. *Inferenzregeln des Produktraums*

„Eq" im Namen zu erkennen. Im speziellen Fall, daß dieses Urteil die Form $\vdash s = s \in T$ hat, das wir auch kurz $\vdash s \in T$ schreiben, wird der Gleichheitsaspekt, da trivial, irrelevant, und wir können dies als reines Typurteil ansehen.

Für den Produktraum gibt es nur zwei primitive Operationen. Einerseits ist dies die Paarbildung `pair`, notiert als $\langle s, t\rangle$, die zwei Elemente s und t, die zwei Typen S und T entnommen sind, zu einer Einheit zusammenfügt. Das Paar hat dann den Typ $S{\times}T$. Um ein Paar wieder zu zerlegen, gibt es den Paaranalyse-Operator `spread`, notiert als let $\langle x, y\rangle = s$ in t, der ein Element s eines Produktraums in seine zwei Anteile x und y zerteilt, die in t zur weiteren Berechnung verwendet werden dürfen. – Die Typentheorie hat den Produktraum noch etwas verallgemeinert, da es ihn auch in einer *abhängig* genannten Variante unterstützt. In diesem Fall ist der Typ des Produkts $x{:}S \times T$, was bedeutet, daß der Typ der rechten Paarhälfte davon abhängt, welches Element in der linken Hälfte steht.

$\vdash$ (let $\langle x,y\rangle = \langle 1,\text{"ABC"}\rangle$ in x) $\in \mathbf{Z}$
BY spreadEq 'z' '$\mathbf{Z}$' '$\mathbf{Z}\times\mathbf{Atom}$' – *let Herunterbrechen*

$\vdash \langle 1,\text{"ABC"}\rangle \in \mathbf{Z}\times\mathbf{Atom}$
BY pairEq '$\mathbf{U}_1$' – *Paar herunterbrechen*

$\vdash 1 \in \mathbf{Z}$
BY natnumEq – *Zahl typisieren*

$\vdash \text{"ABC"} \in \mathbf{Atom}$
BY tokEq – *Symbol typisieren*

$x' : \mathbf{Z} \quad \vdash \mathbf{Z} \in \mathbf{U}_1$
BY intEq – *Typ der Zahlen typisieren*

$s:\mathbf{Z},\ \ t:\mathbf{Atom},\ \ \langle 1,\text{"ABC"}\rangle = \langle s,t\rangle \in \mathbf{Z}\times\mathbf{Atom} \quad \vdash s \in \mathbf{Z}$
BY hypEq – wahr nach Hypothese

Abb. 2.4. *Typisierungsbeweis*

Wie man sieht, erzeugen die Eq-Regeln, wenn man sie auf ein reines Typurteil anwendet, ausschließlich Untersequenzen, die wiederum Typurteile sind. Wir können diese Eq-Regeln daher in dem speziellen Sinn auffassen, daß sie die Inferenzregeln für die Typisierung sind. Dies gilt übrigens auch für fast alle Eq-Regeln, die zur Typisierung der anderen Typen benutzt werden; nur ganz wenige Ausnahmen erzeugen, auf ein Typurteil angewendet, Untersequenzen mit anderem Muster. Schauen wir uns ein erstes Beispiel für einen Typisiserungsbeweis an.

Abbildung 2.4 zeigt den Beweis des Typurteils $\vdash$ (let $\langle x,y\rangle = \langle 1,\text{"ABC"}\rangle$ in x) $\in \mathbf{Z}$. Der Beweis kann überhaupt nur angefangen werden, wenn man schon vorab weiß, zu welchem Typ der fragliche Term gehört; es ist nicht so, daß sich der Typ erst im Laufe des Beweises ergibt. Der Term und der behauptete Typ werden durch die angewandten Inferenzregeln sukzessive in ihre Bestandteile zerlegt, so daß in den Untersequenzen Typurteile stehen, die die entsprechenden Teile von Term und Typ in Beziehung setzen. Bei genauem Hinsehen merken wir, daß wir nicht nur zu Beginn des Beweises eine Typvorausschau benötigen, auch die spreadEq-Regel hat als Argument den Typ von $\langle 1,''ABC''\rangle$, den man angeben muß, damit der Beweis weitergehen kann.

Wir halten fest, daß in Typisierungsbeweisen an manchen Stellen der Typ von Termen bestimmt werden muß. Dies erschwert natürlich die Automatisierung, da man nicht nur nach passenden Inferenzregeln schauen, sondern diese ggf. mit richtigen Typargumenten versehen muß.

Es ist natürlich die Frage, ob die Typbestimmung algorithmisch lösbar ist, möglichst in praktisch vertretbarer Zeit. Wir werden diesen Problemen weiter unten etwas genauer nachgehen; an dieser Stelle soll nur die Schwierigkeit des Unterfangens anhand von Beispielen illustriert werden.

Im folgenden verwenden wir neben Produkten auch Listen. Zur Konstruktion einer Liste gibt es zum einen die leere Liste nil, zum anderen die Operation x::L, die vor die Liste L das Element x setzt. Der Typ einer Liste mit Elementen vom Typ S wird mit S list angegeben. Weitere Operationen erläutern wir bei Bedarf.

Wir betrachten folgenden Typ T:

$$n{:}\,\mathbf{Z} \times \{L{:}\,\mathbf{Atom}\text{ list} \mid \texttt{length}(L) = n \in \mathbf{Z}\}$$

Dies ist der Typ aller Paare $\langle n, L\rangle$ einer ganzen Zahl n und einer Symbolliste L, deren Länge, die durch die Abstraktion `length` festgestellt wird, gleich n ist. Um nun beispielsweise das Typurteil $\vdash \langle 2, \text{"A"::"B"::nil} \rangle \in T$ zu beweisen, reicht es nicht mehr aus, nur Eq-Regeln zu benutzen. Im Laufe des Beweises werden wir eine Sequenz zu zeigen haben, die ausdrückt, daß die Länge der gegebenen Liste in der Tat zwei ist, also eine Sequenz, die kein Typurteil mehr, sondern eine konkrete Eigenschaft der `length`-Abstraktion ist. Der Typbeweis mutiert in einen Eigenschaftsbeweis!

Anstelle von $\texttt{length}(L) = n$ hätten wir auch irgendeine andere Proposition schreiben können; die Form des Typs T, bestehend aus einem echten abhängigen Produkt und einem Teiltyp, kann also benutzt werden, um beliebige propositionale Forderungen in Typen einzubauen, die man dann bei einem konkreten Typbeweis nachweisen muß. Ein automatischer Typbeweiser schlösse damit einen automatischen Propositionsbeweiser (des ersten Universums) ein. Letzterer ist praktisch unmöglich.

Immerhin haben wir es uns bei diesem Beispiel mit dem Teilproblem der Typbestimmung noch nicht ganz verdorben. Wir können nämlich, etwa für das konkrete Element $\langle 2, \text{"A"::"B"::nil} \rangle$, durchaus Typen angeben, und zwar mehrere. Neben T z.B. auch den einfachsten Typ $\mathbf{Z} \times \mathbf{Atom}$ list. Genauer betrachtet, haben wir sogar die Auswahl zwischen unendlich vielen Typen, denen dieses Element tatsächlich angehört. Die Frage ist jetzt natürlich, welcher in der konkreten Situation der richtige ist. Dies kann man dann entscheiden, wenn man in der Lage ist, Typgleichungen $X = Y$, in denen unbekannte Typvariablen vorkommen können, zu lösen. Betrachten wir dazu noch einmal den Beweis in Abbildung 2.4 und stellen uns vor, das zu zeigende Urteil wäre folgendes gewesen:

$$\vdash (\text{let } \langle x, y \rangle = \langle 2, \text{"A"::"B"::nil} \rangle \text{ in } \texttt{rest}(y)) \in \{L\colon \mathbf{Atom} \text{ list} \mid \texttt{length}(L) = 1 \in \mathbf{Z}\}$$

Dabei gibt die `rest`-Abstraktion, angewandt auf eine Liste, diese Liste verkürzt um das erste Element zurück. Wie wären wir nun an den Typ gekommen, den wir der spreadEq-Regel als Argument übergeben? Es muß dabei der Typ des Teilterms $\langle 2, \text{"A"::"B"::nil} \rangle$ angegeben werden. Die einfachste Wahl $\mathbf{Z} \times \mathbf{Atom}$ list führt dummerweise nicht zum Ziel, da dann das zweite Unterziel von spreadEq unbeweisbar wäre. (Um dies einzusehen, probiere der Leser den Beweis analog Abbildung 2.4.)

Wir gehen nun systematisch an die Lösung des Problems heran. Dazu vereinbaren wir für alle Teilterme Symbole für deren Typ:

$$\underbrace{\text{let } \langle \underbrace{x}_{T_4}, \underbrace{y}_{T_3} \rangle = \underbrace{\langle \underbrace{2}_{T_6}, \underbrace{\underbrace{\text{"A"}}_{T_8}::\underbrace{\text{"B"}}_{T_9}::\underbrace{\text{nil}}_{T_{10}}}_{T_5} \rangle}_{T_7} \text{ in } \underbrace{\texttt{rest}(\underbrace{y}_{T_3})}_{T_2}}_{T_1}$$

Aus der Typsemantik der einzelnen Operatoren können wir nun zahlreiche Typgleichungen gewinnen. Beispielsweise können wir den Zusammenhang $T_7 = T_6 \times T_5$ aufstellen, weil der Typ eines Paars eben das Produkt der Typen der Elemente ist[11]. Im einzelnen erhalten wir folgende Typgleichungen:

[11] Man beachte, daß es sich hierbei nur um eine hinreichende Bedingung handelt. Es wird nicht berücksichtigt, daß auch ein abhängiges Produkt möglich wäre, d.h. man könnte auch anders ansetzen.

Semantik von `spread`:	$T_1 = T_2$, $T_7 = T_4 \times T_3$
Semantik von `pair`:	$T_7 = T_6 \times T_5$
Semantik von `rest`:	$T_3 = T_2$
Semantik des ::-Operators:	$T_8 = T_9$, $T_5 = T_8$ list, $T_{10} = T_9$ list
Semantik von Zahlkonstanten:	$T_6 = \mathbf{Z}$
Semantik von Symbolkonstanten:	$T_8 = \mathbf{Atom}$, $T_9 = \mathbf{Atom}$
Ansatz:	$T_1 = \{L\colon \mathbf{Atom}\text{ list} \mid \texttt{length}(L) = 1 \in \mathbf{Z}\}$

Der „Ansatz" ergibt sich durch die obige Aufgabenstellung.

Offensichtlich kann man aus diesem Gleichungssystem die Lösung T_7 wie folgt ableiten:

$$T_7 = T_4 \times T_3 = T_6 \times T_5 \quad \Rightarrow T_4 = T_6$$
$$T_7 = T_4 \times T_3 = T_6 \times T_3 = \mathbf{Z} \times T_1 = \mathbf{Z} \times \{L\colon \mathbf{Atom}\text{ list} \mid \texttt{length}(L) = 1 \in \mathbf{Z}\}$$

Natürlich funktioniert dies, weil das Beispiel hinreichend einfach gestrickt ist. Allerdings begegnen wir bereits einem Problem dieses Gleichungsansatzes: Es kann nämlich auch eine widersprüchliche Gleichung abgeleitet werden, nämlich

$$\mathbf{Atom}\text{ list} = \{L\colon \mathbf{Atom}\text{ list} \mid \texttt{length}(L) = 1 \in \mathbf{Z}\}$$

– was sich freilich vermeiden läßt, wenn man nicht überall Gleichungen verwendet, sondern auch Teiltypbeziehungen $S \subseteq T$ erlaubt, oder indem man versucht, die Bedeutung der Gleichheit umzudefinieren[12].

Das gerade noch behandelbare Problem kann man wiederum ins praktisch Unlösbare steigern. Dazu betrachten wir den folgenden Typ W:

$$n\colon \mathbf{Z} \times (\text{ if } n \text{ rem } 2 = 0 \text{ then } \mathbf{Atom} \text{ else } \mathbf{Atom}\text{ list})$$

Im Unterschied zu vorhin hängt jetzt von n nicht nur eine zusätzliche Bedingung für das rechte Element des Paares ab, vielmehr bestimmt n den Typ des rechten Elementes. Ist n gerade, so ist dieses ein Symbol, sonst eine Symbolliste. Der Typ des rechten Elementes wird berechnet! Für solche Konstruktionen ist nun auch die Typbestimmung extrem schwierig, da über das Ergebnis einer Berechnung geschlossen werden muß, um den Typ zu erhalten. Es ist nun sogar problematisch, überhaupt einen Kandidaten für einen Typ zu bekommen.

Wie wir gesehen haben, ermöglicht die Typentheorie durch bestimmte Konstrukte, daß Aussagen über den Wert von Berechnungen Eingang in einen Typ finden, entweder als Zusatzbedingungen oder in der noch schwieriger zu behandelnden Form, daß je nach Ergebnis der Berechnung unterschiedliche Typen zum Zuge kommen. Beides führt dazu, daß die Automatisierung der Typisierung praktisch unlösbar ist. Da diese Schwierigkeiten durch ganz bestimmte Konstrukte entstehen, können wir aber diese Konstrukte aus unseren Automatisierungsbemühungen ausklammern. Wir werden sehen, wie dies funktionieren könnte.

Zunächst aber noch zu einer anderen Anforderung an automatische Typbeweiser. Wir benötigen ein Konzept, mit dessen Hilfe wir Abstraktionen, die in Termen benutzt werden, in ähnlicher

[12] Beispielsweise, indem zwei Typen als gleich betrachtet werden, wenn ihre Elemente in gleicher Weise strukturiert sind – auch wenn möglicherweise einer der beiden Typen mehr Elemente enthält als der andere. Dann ist die angegebene Gleichung auch nicht mehr widersprüchlich. Möglicherweise ergibt sich dann aber ein Problem mit dem leeren Typ: Gelten Gleichungen, deren eine Seite leer ist?

Weise typisieren können wie die primitiven Operatoren. Die Autotaktik beinhaltet einen interessanten Mechanismus, der in dieser Hinsicht richtungsweisend ist.

Zu jedem abstrakten Operator, der definiert wird, muß zusätzlich ein *Wohlgeformtheitslemma* (Wf-Lemma) angegeben und bewiesen werden. Dieses Lemma muß die Form einer einfachen universellen Formel haben (s.u.), was wir an zwei Beispielen zeigen wollen.

Für beliebige Listen L sind folgende Operationen sinnvoll: `lenght`(L) berechnet die Zahl der Elemente von L und `rest`(L) gibt L verkürzt um das erste Element zurück:

$\triangleright$ `length`(L) $\equiv$ rec-case L of $[] \mapsto 0 \mid h{::}t \mapsto length_t.length_t + 1$
$\triangleright$ `rest`(L) $\equiv$ rec-case L of $[] \mapsto \infty \mid h{::}t \mapsto x.t$

Dabei bewirkt die rec-case-Operation, daß die angegebene Liste rekursiv zerlegt wird.

Die dazugehörigen Wf-Lemmata lauten nun:

$\triangleright$ $\forall T{:}\,\mathbf{U}_i.\ \forall L{:}\,T$ list. `length`(L) $\in \mathbf{Z}$
$\triangleright$ $\forall T{:}\,\mathbf{U}_i.\ \forall L{:}\,T$ list. $L \neq [] \Rightarrow$ `rest`(L) $\in T$ list

Auf der Ebene von Lemmata wird nun ausgedrückt, was die Eq-Regeln für die Primitiven leisten: Ein Typurteil, z.B. $\vdash$ `length`("A"::nil) $\in \mathbf{Z}$, kann in seine Bestandteile heruntergebrochen werden (hier durch die Taktik MemCD, die zur Autotaktik gehört):

$\vdash$ `length`("A"::nil) $\in \mathbf{Z}$
BY MemCD
  $\vdash$ **Atom** $\in$ **U**
  $\vdash$ "A"::nil $\in$ **Atom** list

MemCD sucht nun in der Bibliothek nach einem passenden Wf-Lemma für die `length`-Abstraktion, und wendet dieses automatisch an. Die Instanzen für die Variablen T und L werden dabei durch spezielle Matching-Methoden gefunden, der Universen-Level i wird durch eine Heuristik bestimmt.

`rest` hat im Gegensatz zu `length` ein *bedingtes* Wf-Lemma. Die Bedingung $L \neq []$ wird beim Anwenden des Lemmas in einem eigenen Unterziel zurückgegeben. Damit ist sichergestellt, daß niemals der ∞-Term, der symbolisch für die Definitionslücke steht, ausgewertet wird[13].

Solche Bedingungen sind in zweierlei Hinsicht schwierig:

1. Eigentlich gehört der Beweis der Bedingungen zum Typbeweis dazu. Es handelt sich jedoch um den Nachweis einer Proposition und nicht um ein Typurteil. Typbeweiser wie die Autotaktik kommen mit ihnen nicht zurecht und geben sie als offene Ziele zurück.
2. Um diese Bedingung zu beweisen, benötigt man in der Regel *Kontextbedingungen*, also z.B. Kenntnis, in welcher Weise L verwendet wird. Beispielsweise könnte ja `rest` in einem Konditional vorkommen:

   if $L = []$ then ... else `rest`(L)

   Da `rest` im else-Zweig vorkommt, kann L unmöglich leer sein; der Operator `rest` wird daher niemals auf die Definitionslücke angewendet.

[13] ∞ kann man als einen beliebigen Term definieren, der unter keinen Umständen typisierbar ist, z.B. $\infty \equiv (\lambda x.x\,x)\,(\lambda x.x\,x)$.

Die bisher betrachtete Struktur von Typbeweisen interessiert sich nur für das Zerlegen einer Formel in Typurteile, nicht dafür, während des Zerlegens kontextuelle Eigenschaften aufzusammeln. Obwohl Typbeweiser bedingte Wohlgeformtheiten nicht selbst nachweisen können, müssen sie jedoch wärend des reinen Typbeweises zugleich alle relevanten Kontextbedingungen erzeugen. Dies kann sehr schwierig sein, wenn man daran denkt, daß es sich bei Kontextbedingungen auch um Invarianten handeln kann.

Bedingte Wf-Lemmata sind von der Semantik recht ähnlich zu den mittels Teiltyp ausgedrückten Typzusatzbedingungen. Dennoch sind bedingte Wf-Lemmata einfacher zu handhaben, da zum einen bestimmte Schwierigkeiten des Teiltypkonstrukts entfallen und zum anderen eine klarere Trennung zwischen Typbeweis und Bedingungsbeweis existiert.

Nun aber, nach den Ausführungen über das *Rechnen* und das *Typisieren*, endlich zum dritten Aspekt des Programmierens, dem ...

- *Schließen:* Unsere neue Programmiersprache, die wir aus der Typentheorie entwickeln, muß es natürlich erlauben, Theoreme über die Wirkung von Programmen aufzustellen und zu beweisen. Von der Typentheorie verlangen wir, sollte sie tatsächlich als Grundlagentheorie taugen, daß sie alle für unseren Zweck relevanten Schlußweisen unterstützt. Wir können dabei zwei Bereiche ausmachen:

  1. *Logisches Schließen:* Es sollen Propositionen (Aussagen) P gebildet werden können, so daß möglichst alle (semantisch) wahren Urteile $\vdash P$ beweisbar sind. Die Typentheorie geht hier den Weg, Propositionen durch Typen zu simulieren. Dabei kommt eine konstruktive Prädikatenlogik beliebiger Stufe heraus, für die alle in der (konstruktiven) Mathematik üblichen Beweistechniken benutzt werden können. Die Simulation stützt sich dabei auf den intuitionistischen Sequenzenkalkül, dessen Regeln von Taktiken nachgeahmt werden. Logische Schlüsse können mit diesen simulierten Logikregeln durchgeführt werden.
  2. *Typbezogenes Schließen:* Propositionen können vor allem durch die vordefinierten Prädikate gebildet werden, insbesondere der Gleichheit, die für jeden Typ definiert ist. Für jeden Typ gibt es einen Satz von Inferenzregeln, die spezifische Schlußweisen für Urteile vorsehen, an denen der jeweilige Typ beteiligt ist. Abbildung 2.4 zeigt z.B. die Inferenzregeln des Produktraums. Sie lassen sich in vier Gruppen einteilen: Gleichheitsregeln (erkennbar am „Eq“ im Namen), Einführungsregeln („I“), Eliminationsregeln („E“) und Reduktionsregeln („Red“). Sie entsprechen vier grundlegenden Aspekten des typbezogenen Schließens: Behandlung der Gleichheit, Existenz (Konstruktion) von Elementen, Analyse von Elementen bzw. Durchführung eines Berechnungsschritts.

Wir haben also einen Satz von Logikregeln sowie für jeden Typ einen Satz von spezifischen Inferenzregeln, die die Basis aller Schlüsse sind. Abbildung 2.5 zeigt exemplarisch den Beweis des Urteils $\langle x.1, x.2\rangle = x$ unter der Annahme, daß x vom Typ $S \times T$ für zwei beliebige Typen S und T ist. Der Beweis hat, trotz der Einfachheit des Urteils, schon beachtliche Länge; die Inferenzregeln sind also sehr kleinschrittig.

Welche Möglichkeiten gibt es, zu größeren Schritten zu kommen? NuPRL erlaubt nicht nur, Beweise durch primitive Regeln zu verfeinern, sondern ebenfalls mittels Taktiken, die sich nach außen hin wie Regeln verhalten. Die Taktik konstruiert dabei, in der Regel für den Benutzer nicht sichtbar, einen Beweis, der die fehlende Regel nachbildet. Wir studieren dies wieder an einem Beispiel.

$S:\mathbf{U}_j,\ T:\mathbf{U}_j,\ x:S{\times}T$
$\quad\vdash \langle x.1, x.2\rangle = x \in S{\times}T$
BY productE 3 – *Analyse von x als Paar $\langle s,t\rangle$*

$\quad S:\mathbf{U}_j,\ T:\mathbf{U}_j,\ x:S{\times}T,\ s:S,\ t:T$
$\quad\quad\vdash \langle\langle s,t\rangle.1, \langle s,t\rangle.2\rangle = \langle s,t\rangle \in S{\times}T$
$\quad$BY pairEq j – *Zur Gleichheit der Komponenten übergehen*

$\quad\quad S:\mathbf{U}_j,\ T:\mathbf{U}_j,\ x:S{\times}T,\ s:S,\ t:T$
$\quad\quad\quad\vdash \langle s,t\rangle.1 = s \in S$
$\quad\quad$BY unfold `pi1` – *Auffalten der* `pi1`*-Abstraktion*

$\quad\quad\quad S:\mathbf{U}_j,\ T:\mathbf{U}_j,\ x:S{\times}T,\ s:S,\ t:T$
$\quad\quad\quad\quad\vdash \text{let } \langle x,y\rangle = \langle s,t\rangle \text{ in } x = s \in S$
$\quad\quad\quad$BY spreadRed – *Kontraktion des Redex'*

$\quad\quad\quad\quad S:\mathbf{U}_j,\ T:\mathbf{U}_j,\ x:S{\times}T,\ s:S,\ t:T$
$\quad\quad\quad\quad\quad\vdash s = s \in S$
$\quad\quad\quad\quad$BY hypEq 4 – *Wahr laut Annahme 4 ($s:S$)*

$\quad\quad S:\mathbf{U}_j,\ T:\mathbf{U}_j,\ x:S{\times}T,\ s:S,\ t:T$
$\quad\quad\quad\vdash \langle s,t\rangle.2 = t \in T$
$\quad\quad$BY unfold `pi2` – *Auffalten der* `pi2`*-Abstraktion*

$\quad\quad\quad S:\mathbf{U}_j,\ T:\mathbf{U}_j,\ x:S{\times}T,\ s:S,\ t:T$
$\quad\quad\quad\quad\vdash \text{let } \langle x,y\rangle = \langle s,t\rangle \text{ in } y = t \in T$
$\quad\quad\quad$BY spreadRed – *Kontraktion des Redex'*

$\quad\quad\quad\quad S:\mathbf{U}_j,\ T:\mathbf{U}_j,\ x:S{\times}T,\ s:S,\ t:T$
$\quad\quad\quad\quad\quad\vdash t = t \in T$
$\quad\quad\quad\quad$BY hypEq 5 – *Wahr laut Annahme 5 ($t:T$)*

$\quad\quad S:\mathbf{U}_j,\ T:\mathbf{U}_j,\ x:S{\times}T,\ s:S,\ t:T,\ x':S$
$\quad\quad\quad\vdash T \in \mathbf{U}_j$
$\quad\quad$BY hypEq 2 – *Wahr laut Annahme 2*

Abb. 2.5. *Exemplarischer Beweis mittels der primitiven Inferenzregeln*

Die primitiven Regeln für den Produktraum ermöglichen es, eine Paargleichung $\langle s_1,t_1\rangle = \langle s_2,t_2\rangle$, die in der Konklusion einer Sequenz steht, in zwei Komponentengleichungen $s_1 = s_2$ und $t_1 = t_2$ zu zerlegen. Es gibt jedoch keine Regel, die eine Paargleichung, die in einer Hypothese steht, in gleicher Weise zerlegt. Eine taktische Regel pair_eqE_indep[14] soll diese Lücke füllen. Im oberen Teil von Abbildung 2.6 sehen wir das *Verhalten* der Taktik. Sie wird auf eine Sequenz angesetzt, in deren i-ter Hypothese die Paargleichung steht; die anderen Hypothesen Γ und Δ sind beliebig. Sie konstruiert nun für sich den Beweis weiter, läßt jedoch alle Details im Verborgenen, und gibt nur diejenigen Sequenzen zurück, die sie nicht weiter verfeinert (offene Ziele). Neben einer Hauptsequenz, hier die Komponentengleichungen enthaltend, werden in der Regel noch Hilfssequenzen erzeugt, die die Annahmen, die während in die Beweiskonstruktion eingingen, explizit

[14] Der Suffix „indep“ deutet an, daß diese Taktik nur für unabhängige Produkte funktioniert. Eine entsprechende Taktik für abhängige Produkte ist wesentlich komplizierter. Es liegt jedoch eine Implementierung für pair_eqE vor.

$\Gamma,\ \langle s_1,t_1\rangle = \langle s_2,t_2\rangle \in S{\times}T,\ \Delta \quad\vdash\quad C$

BY pair_eqE_indep i

$\Gamma,\ \langle s_1,t_1\rangle = \langle s_2,t_2\rangle \in S{\times}T,\ s_1 = s_2 \in S,\ t_1 = t_2 \in T,\ \Delta \quad\vdash\quad C$

...

... } *Ggf. Hilfsziele*

$\Gamma,\ \langle s_1,t_1\rangle = \langle s_2,t_2\rangle \in S{\times}T,\ \Delta \quad\vdash C$

BY cut ‘$\langle s_1,t_1\rangle.1 = \langle s_2,t_2\rangle.1 \in S$‘ – *Schnitt*

$\Gamma,\ \langle s_1,t_1\rangle = \langle s_2,t_2\rangle \in S{\times}T,\ \Delta$

$\vdash \langle s_1,t_1\rangle.1 = \langle s_2,t_2\rangle.1 \in S$

BY unfold `pi1` – *Auffalten der* `pi1`*-Abstraktion*

$\Gamma,\ \langle s_1,t_1\rangle = \langle s_2,t_2\rangle \in S{\times}T,\ \Delta$

$\vdash (\text{let } \langle x,y\rangle = \langle s_1,t_1\rangle \text{ in } x) = (\text{let } \langle x,y\rangle = \langle s_2,t_2\rangle \text{ in } x) \in S$

BY spreadEq ‘z‘ ‘S‘ ‘$S{\times}T$‘ – *Zur Komponentengleichheit übergehen*

$\Gamma,\ \langle s_1,t_1\rangle = \langle s_2,t_2\rangle \in S{\times}T,\ \Delta$

$\vdash \langle s_1,t_1\rangle = \langle s_2,t_2\rangle \in S{\times}T$

BY hypothesis i – *Wahr nach Hypothese i*

$\Gamma,\ \langle s_1,t_1\rangle = \langle s_2,t_2\rangle \in S{\times}T,\ \Delta,\ s:S,\ t:T,\ h:\langle s_1,t_1\rangle = \langle s,t\rangle \in S{\times}T$

$\vdash s = s \in S$

BY hypEq ... – *Wahr nach einer der Hypothesen*

$\Gamma,\ \langle s_1,t_1\rangle = \langle s_2,t_2\rangle \in S{\times}T,\ \langle s_1,t_1\rangle.1 = \langle s_2,t_2\rangle.1 \in S,\ \Delta \quad\vdash C$

BY *Abstrakte Reduktion der beiden* `pi1`*-Abstraktionen*

$\Gamma,\ \langle s_1,t_1\rangle = \langle s_2,t_2\rangle \in S{\times}T,\ s_1 = s_2 \in S,\ \Delta \quad\vdash C$

BY cut ‘$\langle s_1,t_1\rangle.2 = \langle s_2,t_2\rangle.2 \in T$‘ – *Schnitt*

... *analog dem oberen Schnitt, jedoch für* `pi2`

Abb. 2.6. *oben: Verhalten der taktischen Regel pair_eqE_indep. Unten: Schema des Beweises, den die Taktik intern konstruiert.*

machen. Der untere Teil der Abbildung 2.6 zeigt das Schema, nach dem die Taktik einen Beweis konstruiert.

Auf diese Weise kann man nun Taktiken benutzen, um höhere Schlußregeln zur Verfügung zu stellen. Dabei muß es sich nicht um einen (wie in unserem Beispiel) vorab schematisierten Beweis handeln, vielmehr sind Taktiken grundsätzlich auch in der Lage, Beweise nach beliebigen Erwägungen zu konstruieren. Insbesondere liegt also auch die *automatische Suche* nach Beweisen im Bereich des Möglichen.

Neben der Konstruktion durch primitive oder taktische Regeln, die kleinere oder größere Schritte des Schließens erlauben, kann man auch beweisen, indem man auf bereits fertiges Wissen zurückgreift. In NuPRL können bewiesene Sätze in einer Bibliothek abgelegt und in neu angefangenen Beweisen zitiert werden. Letzteres wird durch eine spezielle Inferenzregel `lemma` bewirkt, die anwendbar ist, wenn die zu beweisende Konklusion mit dem Theorem der Bibliothek genau übereinstimmt.

Dies ist wiederum nur ein sehr primitiver Schritt. Mittels Taktiken können wir aber flexiblere Aufrufe von Theoremen programmieren. Zwei (aufeinander aufbauende) Ideen zeigen, wie dies aussehen kann:

1. *Universelle Formeln:* Die Theoreme aus der Bibliothek müssen eine bestimmte Form haben, die grob
$$\forall x_1{:}T_1.\ \dots \forall x_n{:}T_n,\ \ A_1 \Rightarrow \dots \Rightarrow A_m \Rightarrow C$$
ist. Die zu beweisende Sequenz habe eine Formel C' in der Konklusion. Dabei muß sich C' aus C ergeben, indem konkrete Terme für die Variablen x_i eingesetzt werden. Die Taktik liefert für jede der Annahmen A_j eine noch zu beweisende Sequenz zurück.
2. *Termersetzung:* Die Theoreme aus der Bibliothek müssen ebenfalls eine bestimmte Form haben:
$$\forall x_1{:}T_1.\ \dots \forall x_n{:}T_n,\ \ A_1 \Rightarrow \dots \Rightarrow A_m \Rightarrow s\ R\ t$$
Dabei ist R eine zweistellige Relation mit bestimmten Eigenschaften. Die zu beweisende Sequenz habe eine Formel C in der Konklusion. Es wird nun durch die Taktik eine Hauptsequenz erzeugt, in der ein Teilterm s_C von C, der sich aus s durch Instanziieren der Variablen x_i ergibt, durch einen Term t_C ersetzt wurde, in dem die gleichen Instanzen für die x_i-Variablen in t eingesetzt sind. Neben der Hauptsequenz werden noch zahlreiche Hilfssequenzen erzeugt, die die impliziten Annahmen und die A_j enthalten.

Durch diese Techniken der Lemmaanwendung ergibt sich eine vielversprechende Möglichkeit, allgemein formuliertes Wissen für den bestimmten Fall zu konkretisieren, der gerade bewiesen werden soll.

*

Die drei Aspekte Rechnen, Typisieren und Schließen haben zwar konzeptionell verschiedene Ursprünge, fließen aber in realen Beweisen ineinander. Vor allem zwei Verhältnisse springen ins Auge:

1. *Rechnen und Schließen:* Oftmals werden einzelne Berechnungsschritte innerhalb von Beweisen durchgeführt. Dadurch haben sie den Charakter von Schlüssen. Dieses Verhältnis wird interessanter, wenn man bedenkt, daß es auch die Möglichkeit gibt, Berechnungen in umgekehrter Richtung durchzuführen, d.h. von einem gegebenen Ergebnis auf eine mögliche Eingabe der Rechnung zu schließen. Diese Richtung ist nicht deterministisch, weshalb man, wie bei den universellen Formeln, bestimmte Variablen instanziieren muß.

2. *Typisieren und Schließen:* Typurteile scheinen dieselbe Form zu haben wie Aussagenurteile, ja der Operator $\cdot = \cdot \in \cdot$ wird sogar für beide Zwecke, Typzugehörigkeit und Gleichheit, verwendet. Es ist jedoch unbedingt zu beachten, daß Typurteile fundamentaler sind, da durch Typschlüsse erst festgestellt wird, ob eine Aussage überhaupt eine Aussage ist. Dies äußert sich auch darin, daß über Typzugehörigkeit nicht in der allgemeinen Weise geschlossen werden kann wie über propositionale Urteile, insbesondere kann über Typen nur apodiktisch, nicht jedoch hypothetisch

geurteilt werden, d.h. man kann keine Annahmen über Typzugehörigkeit machen. So ist z.B. $x \in S \Rightarrow x \in T$ keine Aussage, da das Typurteil $\vdash x \in T$ unter der Annahme der Gültigkeit des Urteils $\vdash x \in S$ behauptet wird (es sei denn, $\vdash x \in T$ wäre ohnehin wahr).

Allerdings haben wir an Beispielen gesehen, daß man Typen konstruieren kann, die Zusatzbedingungen, also Propositionen, enthalten. Für einen konkreten Typisiserungsbeweis heißt dies, daß dieser in einen Propositionsbeweis übergeht. Umgekehrt kommt es sehr häufig vor, daß Propositionsbeweise, da Typannahmen gemacht werden, als Typisierungsbeweise fortgesetzt werden. Die Typentheorie trennt also beide Konzepte strikt, um sie an genau kontrollierten Übergängen ineinander überführbar zu machen.

Bildlich gesprochen kann man sich Typisierungen als Fundament eines Hauses vorstellen, auf dem die Propositionen gemauert werden. Es ist möglich, höhere Typen zu bilden, die von Propositionen abhängen, also Zwischendecken einzuziehen, auf denen man wieder Mauern errichten kann. Die Typentheorie zwingt uns dazu, entweder einen Typbeweis zu führen, der auf die tieferen Etagen des Hauses zurückgehen kann, oder einem Propositionsbeweis zu folgen, dessen Mauern aber nur auf einem tragfähigen (Typ-)Boden errichtet werden dürfen.

Soweit eine erste Inspektion, inwieweit die Typentheorie zu einer praktisch benutzbaren Programmiersprache ausgebaut werden kann.

2.3 Simulation abstrakter Datentypen

Ein abstrakter Datentyp (ADT) ist dadurch gekennzeichnet, daß keine Festlegung auf einen bestimmten Trägertyp getroffen wird, sondern die Semantik des Typs nur über die Wirkung der Operationen des ADTs angegeben wird. In meiner Studienarbeit [stolpmann96] habe ich ausführlich diskutiert, wie man das Konzept ADT in NuPRL realisieren kann. Im folgenden werden kurz die wichtigsten Ergebnisse in Hinblick auf das Projekt Globalsuche dargestellt.

Es wird davon ausgegangen, daß eine axiomatische Spezifikation des ADT vorliegt, d.h. es werden die Namen der Operationen, ihre Signaturen und eine Menge von Axiomen angegeben. Um einen ADT zu implementieren, muß es mindestens typentheoretische Theoreme geben, die die Existenz von Funktionen garantieren, die diese Spezifikation erfüllen. Desweiteren muß die Existenz eines Trägertyps gesichert sein, der den ADT repräsentiert.

Es zeigt sich, daß es zwar hochinteressant, aber auch hochproblematisch ist, bei dieser Minimalforderung der bloßen Existenz von Trägertyp und Funktionen zu bleiben. Aus praktischen Gründen ist es angebracht, die Abstraktheit des Datentyps aufzugeben und stattdessen konkrete Funktionen und Trägertypen zu fordern, d.h. wir verlangen nicht nur die Existenz, sondern einen konkreten Term. Dieser Schritt der Konkretisierung ist damit zunächst derselbe wie in hergebrachten Programmiersprachen: Der abstrakte Typ wird durch einen konkreten ersetzt, der aber nur in Hinblick auf seine abstrakten Eigenschaften benutzt wird. Im Unterschied zu anderen Programmiersprachen werden in der Typentheorie jedoch Theoreme aufgestellt, die diese Eigenschaften explizit machen. Diese Theoreme füllen nun die Rolle einer Schnittstelle für den ADT aus.

Da die konkreten Funktionen und Trägertypen eingesetzt werden, gibt es keine Gewähr, daß diese nicht auch benutzt werden. NuPRL kennt leider keinen Kapselungsmechanismus, der den Zugriff auf verborgene Details regelt. Wir können jedoch durch eine disziplinierte Vorgehensweise sichern, daß nur abstrakte Eigenschaften verfügbar sind. Eine Eigenschaft ist dann abstrakt, wenn sie zur ADT-Theorie gehört, d.h. aus den Axiomen des ADT gefolgert werden kann. Es bieten sich zwei methodische Vorgehensweisen an:

1. *Die Methode der Simulation:* Nachdem konkrete Terme für Trägertypen und Funktionen definiert sind, werden alle Axiome des ADT für diese Konkretisierung bewiesen. Alle anderen

Eigenschaften dürfen nur die Axiome benutzen, jedoch nicht die konkreten Terme.

2. *Die Methode der Generierung:* Es wird ein konkreter Term für den Trägertyp definiert. Die Funktionen werden nur spezifiziert, wobei die Spezifikationen Umformungen der Axiome sind. Es wird gezeigt, daß die Spezifikationen erfüllbar sind; in diesem Beweis müssen dann die konkreten Terme für die Funktionen angegeben werden.

Die Methode der Simulation funktioniert immer, wogegen bei der Generierung vorausgesetzt wird, daß die Funktionen seriell spezifizierbar sind. Letzteres ist für einen ADT als ganzes in der Regel nicht der Fall, jedoch für einzelne Funktionen schon. Die Methode der Generierung hat den Vorteil, daß man sich nicht auf einen konkreten Term für eine Funktion festlegen muß, sondern mehrere anbieten kann. Desweiteren können die Terme für die Funktionen, da ja Spezifikationen vorliegen, grundsätzlich durch Synthesetaktiken gewonnen werden.

Um die Vorteile beider Methoden zum Zuge kommen lassen zu können, bietet es sich an, den ADT in zwei Teile zu zerlegen. Der Kern des ADT, gegeben durch seine Grundoperationen, wird durch Simulation implementiert. Der Kern enthält alle Voraussetzungen, damit die restlichen, sog. Erweiterungsoperationen spezifizierbar sind. Diese können nun generiert werden.

2.3.1 Implementierung des Datentyp-Konzepts mit Hilfe von `TYPES`

Ein ADT setzt abstrakte Eigenschaften und läßt zunächst offen, ob er überhaupt implementierbar ist. Wir fordern natürlich zusätzlich, daß die Operationen des ADT nicht nur durch irgendwelche Funktionen, sondern durch Programm-Funktionen konkretisiert werden (A). Genauso fordern wir vom Trägertyp, daß er ein Daten-Typ ist (B). Wir verstehen darunter, daß die Gleichheit zweier Werte dieses Typs entscheidbar ist. Dies ist eigentlich eine intuitiv recht eingängige Forderung, obwohl sich scheinbar bisher niemand dafür interessiert hat.

Was Daten, besonders im Unterschied zu Programmen, im Kern sind, wird nicht häufig gefragt, obwohl allgemein geteilte Auffassungen immer wieder in Systeme oder Sprachen Eingang finden. Gerade funktionale Programmiersprachen, wie etwa ML, versuchen, prinzipiell keinen Unterschied zwischen Daten und Programmen zu machen. An einer Stelle wird aber dennoch deutlich, daß es sich dennoch um zwei Konzepte handelt. ML stellt die Gleichheit generisch für alle Typen zur Verfügung; ist am Typ jedoch der Funktionenraum beteiligt, ist die Wirkung der Gleichheit undefiniert. Der Grund dafür ist, daß die extensionale Gleichheit von Funktionen unentscheidbar ist, und es einer ML-Implementierung daher freigestellt wird, die Gleichheit in irgendeinem intensionalen Sinn zu interpretieren. Die Typentheorie geht, um konsistent zu bleiben, noch einen Schritt weiter und kennt gar nicht die generische Verwendung der Gleichheit als Entscheidungsfunktion. In der Typentheorie ist daher tatsächlich kein prinzipieller Unterschied zwischen Daten und Programmen feststellbar. Der Vergleich mit ML legt aber den Verdacht nahe, daß dies für eine praktisch verwendbare Programmiersprache unhaltbar ist.

Der Hintergrund dafür, daß wir ein Konzept für Daten benötigen, ist, daß einige Objekte dadurch herausgestellt sind, systematisch in endlicher Zeit extensional bestimmbar zu sein. Dies ist für Programme im allgemeinen nämlich nicht möglich, jedoch der Wesenskern von Daten. Eine äquivalente Formulierung dafür ist, daß die Gleichheit eines Datentyps entscheidbar ist.

Die beiden Forderungen A und B müssen nicht unbedingt in die Implementierung von abstrakten Datentypen Eingang finden, sondern könnten auch als Meta-Forderungen außen vor bleiben. Da wir Datentypen jedoch mit generischen Parametern ausstatten wollen, muß B tatsächllich berücksichtigt werden. Beispielsweise ist $\texttt{Seq}(\alpha)$ der Datentyp aller Listen mit Elementtyp α. Um nun bestimmte Operationen für `Seq` auch generisch formulieren zu können, benötigen wir den Zugriff auf die Gleichheit für beliebige Instanzen von α, d.h. generischen Zugriff. Dazu muß das Konzept Datentyp in besonderer Weise typentheoretisch verankert werden.

Folgende Abstraktion modelliert Datentypen:

$\triangleright$ `TYPES` $\equiv$ $T{:}\,\mathbf{U} \times \{eq{:}\,T{\times}T{\rightarrow}\mathbf{B} \mid \forall x,y{:}T.\ (x=y \in T) \iff (eq\langle x,y\rangle = \text{true} \in \mathbf{B})\}$

Ein Datentyp ist somit ein Paar $\langle T, eq\rangle$ aus einem Typ T und einer boole'schen Funktion eq, die die Gleichheit für Werte von T entscheidet. Zusätzlich zu diesem Typ vereinbaren wir folgende Zugriffsoperationen (für Datentypen benutzen wir griechische Buchstaben):

$\triangleright$ α^T $\equiv$ $\alpha.1$
$\triangleright$ `eq`(α) $\equiv$ $\alpha.2$

Um nun z.B. eine Operation $x \in L$ zu programmieren, die angibt, ob x in der Liste L enthalten ist, vergleichen wir x nacheinander mit allen Elementen aus L. Jeder Einzelvergleich benutzt dabei die mittels `eq` aus dem Datentyp der Listenelemente extrahierte Gleichheit.

Im folgenden verwenden wir `TYPES` ähnlich wie $\mathbf{U}$ quasi als weiteres Universum.

2.3.2 Zur Implementierung einzelner Datentypen

Zur Zeit sind fünf Datentypen teilweise implementiert worden. Es handelt sich um

- *Boole'sche Algebra:* Rechnen mit „wahr" und „falsch". Dieser Datentyp darf nicht mit dem Typ der Propositionen ($\mathbf{P}$ bzw. $\mathbf{P}_j$ verwechselt werden. Der Hauptunterschied ist, daß der Wert eines Ausdrucks der boole'schen Algebra durch Berechnung bestimmt wird, während ein propositionaler Ausdruck bewiesen oder falsifiziert wird. Eine genauere Diskussion des Unterschieds findet sich weiter unten.
- *Ganze Zahlen*
- *Unabhängige Produkte:* Dies ist ein Teilkonzept des typentheoretischen Produktraums, die die unabhängige Typisierbarkeit der Elemente fordert. Der Grund dafür ist die einfachere Typisierbarkeit (s.u.)
- *Endliche Sequenzen:* Eine Erweiterung des primitiven Listentyps.
- *Endliche Mengen:* Es können Mengen mit endlich vielen Elementen gebildet und manipuliert werden.

Die Tabellen der nächsten Seiten geben einen Überblick über diese Datentypen. Es werden jeweils die Namen für Träger- und Datentyp und die Namen der Grundoperationen genannt, und auf welche Weise sich diese Typen bzw. Operationen typisieren lassen. Desweiteren werden die Axiome aufgeführt. In den Tabellen sind absichtlich keine Implementierungsdetails genannt, um zu verdeutlichen, daß die Semantik der Datentypen ausschließlich durch den entsprechenden ADT festgelegt wird.

Die Unterscheidung zwischen „Grundoperationen" und „abgeleiteten Grundoperationen" beruht darauf, daß minimal zwar nur die ersteren „Grundoperationen" ausreichend wären, um den Datentyp zu definieren, es sich aber als vorteilhaft herausgestellt hat, weitere Operationen zur Grundlegung des Typs zu benutzen. Daneben gibt es noch, in den Tabellen nicht aufgeführt, zahlreiche Erweiterungsoperationen, die durch Zusammensetzung aus Grundoperationen gewonnen werden.

Konkret müssen zur Implementierung eines Datentyps folgende Bestandteile konstruiert werden; wir gehen dabei davon aus, daß die Methode der Simulation benutzt wird:

- Die Abstraktionen des Datentyps werden definiert, d.h. dem Abstraktionsmechanismus wird mitgeteilt, welche Abkürzungen für welche Terme stehen sollen.
- Es werden Display-Formen für die Abstraktionen vereinbart.

Trägertyp: $\mathbf{B}$
Typisierung: $\mathbf{B} \in \mathbf{U}$

Datentyp: $\mathbf{B}\!\uparrow$
Typisierung: $\mathbf{B}\!\uparrow \in \texttt{TYPES}$

Grundoperationen:

▷ Wahrheit: `true`
▷ Falschheit: `false`
▷ Konditional: $\texttt{if } B \texttt{ then } s \texttt{ else } t$

Typisierungen:

1. $\texttt{true} \in \mathbf{B}$
2. $\texttt{false} \in \mathbf{B}$
3. $\forall T{:}\,\mathbf{U}.\ \forall B{:}\,\mathbf{B}.\ \forall s{:}\,T.\ \forall t{:}\,T.$
   $\texttt{if } B \texttt{ then } s \texttt{ else } t \in T$

Abgeleitete Grundoperationen:

▷ Konjunktion: $A \wedge B$
▷ Disjunktion: $A \vee B$
▷ Negation: $\neg A$
▷ Implikation: $A \Rightarrow B$
▷ Implikation: $A \Leftarrow B$
▷ Aequivalenz/Gleichheit: $A \iff B$
▷ Konversion: $\lceil A \rceil$

Typisierungen:

1. $\forall A, B{:}\,\mathbf{B}.\ A \wedge B \in \mathbf{B}$
2. $\forall A, B{:}\,\mathbf{B}.\ A \vee B \in \mathbf{B}$
3. $\forall A{:}\,\mathbf{B}.\ \neg A \in \mathbf{B}$
4. $\forall A, B{:}\,\mathbf{B}.\ A \Rightarrow B \in \mathbf{B}$
5. $\forall A, B{:}\,\mathbf{B}.\ A \Leftarrow B \in \mathbf{B}$
6. $\forall A, B{:}\,\mathbf{B}.\ A \iff B \in \mathbf{B}$
7. $\forall A{:}\,\mathbf{B}.\ \lceil A \rceil \in \mathbf{P}$

Axiome: Es gelten die üblichen Axiome der boole'schen Algebra.

Tab. 2.7. *Datentyp: Boole'sche Algebra*

Trägertyp: $\mathbf{Z}$
Typisierung: $\mathbf{Z} \in \mathbf{U}$

Datentyp: $\mathbf{Z}\!\uparrow$
Typisierung: $\mathbf{Z}\!\uparrow \in \mathtt{TYPES}$

Grundoperationen:

- ▷ Zahlkonstanten: 0, 1, -1, 2, -2, ...
- ▷ Negation: $-a$
- ▷ Addition: $a+b$
- ▷ Subtraktion: $a-b$
- ▷ Multiplikation: $a*b$
- ▷ Division: a/b
- ▷ Divisionsrest: $a \ \mathtt{rem}\ b$
- ▷ Induktion: $\text{rec-case } a \text{ of } x<0 \mapsto r.n[x,r] \mid 0 \mapsto b \mid x>0 \mapsto r.p[x,r]$
- ▷ Kleiner-Proposition: $a<b$

Typisierungen:

1. Für alle Zahlkonstanten n: $n \in \mathbf{Z}$
2. $\forall a{:}\mathbf{Z}.\ -a \in \mathbf{Z}$
3. $\forall a,b{:}\mathbf{Z}.\ a+b \in \mathbf{Z}$
4. $\forall a,b{:}\mathbf{Z}.\ a-b \in \mathbf{Z}$
5. $\forall a,b{:}\mathbf{Z}.\ a*b \in \mathbf{Z}$
6. $\forall a,b{:}\mathbf{Z}.\ b \neq 0 \Rightarrow a/b \in \mathbf{Z}$
7. $\forall a,b{:}\mathbf{Z}.\ b \neq 0 \Rightarrow a \ \mathtt{rem}\ b \in \mathbf{Z}$
8. $\forall T{:}\mathbf{U}.\ \forall a{:}\mathbf{Z}.\ \forall b{:}T.\ \forall n{:}\mathbf{Z} \xrightarrow{2} T.\ \forall p{:}\mathbf{Z} \xrightarrow{2} T.$
   $(\text{rec-case } a \text{ of } x<0 \mapsto r.n[x,r] \mid 0 \mapsto b \mid x>0 \mapsto r.p[x,r]) \in T$
9. $\forall a,b{:}\mathbf{Z}.\ (a<b) \in \mathbf{P}$

Abgeleitete Grundoperationen:

- ▷ Falls gleich: $\mathtt{if}\ a=b\ \mathtt{then}\ s\ \mathtt{else}\ t$
- ▷ Falls kleiner: $\mathtt{if}\ a<b\ \mathtt{then}\ s\ \mathtt{else}\ t$
- ▷ Boole'sche Gleichheit: $a=b$
- ▷ Boole'sches Kleiner: $a<b$

Typisierungen:

1. $\forall T{:}\mathbf{U}.\ \forall a,b{:}\mathbf{Z}.\ \forall s,t{:}T.\ \mathtt{if}\ a=b\ \mathtt{then}\ s\ \mathtt{else}\ t \in T$
2. $\forall T{:}\mathbf{U}.\ \forall a,b{:}\mathbf{Z}.\ \forall s,t{:}T.\ \mathtt{if}\ a<b\ \mathtt{then}\ s\ \mathtt{else}\ t \in T$
3. $\forall a,b{:}\mathbf{Z}.\ (a=b) \in \mathbf{B}$
4. $\forall a,b{:}\mathbf{Z}.\ (a<b) \in \mathbf{B}$

Axiome: Es gelten die üblichen Axiome für ganze Zahlen.

Tab. 2.8. *Datentyp: Ganze Zahlen*

Trägertyp: $\alpha \times \beta$

Typisierung: $\forall\alpha\colon \mathtt{TYPES}.\ \forall\beta\colon \mathtt{TYPES}.\ \alpha \times \beta \in \mathbf{U}$

Datentyp: $\alpha \boldsymbol{\times} \beta$

Typisierung: $\forall\alpha\colon \mathtt{TYPES}.\ \forall\beta\colon \mathtt{TYPES}.\ \alpha \boldsymbol{\times} \beta \in \mathtt{TYPES}$

Grundoperationen:

▷ Paarbildung: $\langle a, b\rangle$

▷ Paaranalyse: $\mathtt{let}\ \langle x, y\rangle = u\ \mathtt{in}\ v[x, y]$

Typisierungen:

1. $\forall\alpha\colon \mathtt{TYPES}.\ \forall\beta\colon \mathtt{TYPES}.\ \forall a\colon \alpha.\ \forall b\colon \beta.\ \langle a, b\rangle \in \alpha \times \beta$
2. $\forall\alpha\colon \mathtt{TYPES}.\ \forall\beta\colon \mathtt{TYPES}.\ \forall T\colon \mathbf{U}.\ \forall u\colon \alpha \times \beta.\ \forall v\colon \alpha^T \xrightarrow{2} \beta^T \xrightarrow{2} T.$
   $\mathtt{let}\ \langle x, y\rangle = u\ \mathtt{in}\ v[x, y] \in T$

Abgeleitete Grundoperationen:

▷ Projektion links: $u.1$

▷ Projektion rechts: $u.2$

▷ Boole'sche Gleichheit: $u = v$

Typisierungen:

1. $\forall\alpha\colon \mathtt{TYPES}.\ \forall\beta\colon \mathtt{TYPES}.\ \forall u\colon \alpha \times \beta.\ \ u.1 \in \alpha^T$
2. $\forall\alpha\colon \mathtt{TYPES}.\ \forall\beta\colon \mathtt{TYPES}.\ \forall u\colon \alpha \times \beta.\ \ u.2 \in \beta^T$
3. $\forall\alpha\colon \mathtt{TYPES}.\ \forall\beta\colon \mathtt{TYPES}.\ \forall u\colon \alpha \times \beta.\ \forall v\colon \alpha \times \beta.\ \ (u = v) \in \mathbf{B}$

Axiome:

$\forall\alpha\colon \mathtt{TYPES}.\ \forall\beta\colon \mathtt{TYPES}.\ \forall a, b, c, d\colon \alpha \times \beta.\ \forall T\colon \mathbf{U}.\ \forall u\colon \alpha^T \to \beta^T \to T.$

1. $\text{let } \langle x, y\rangle = \langle a, b\rangle \text{ in } u[x, y] \ = \ u[a, b]$
2. $\langle a, b\rangle.1 = a$
3. $\langle a, b\rangle.2 = b$
4. $\langle a, b\rangle = \langle c, d\rangle \iff (a = c \wedge b = d)$

Tab. 2.9. *Datentyp: Unabhängige Produkte*

Trägertyp: $\texttt{seq}(\alpha)$
Typisierung: $\forall\alpha{:}\texttt{TYPES}.\ \texttt{seq}(\alpha) \in \mathbf{U}$

Datentyp: $\texttt{Seq}(\alpha)$
Typisierung: $\forall\alpha{:}\texttt{TYPES}.\ \texttt{Seq}(\alpha) \in \texttt{TYPES}$

Grundoperationen:

▷ leere Sequenz: $[]$
▷ Voranstellen: $a \cdot L$
▷ Induktion: $\texttt{seqrec-case}\ L\ \texttt{of}\ \texttt{[]}\ \mapsto b\ \mid\ a \cdot L' \mapsto FL'.F[a, L', FL']$

Typisierungen:

1. $\forall\alpha{:}\texttt{TYPES}.\ [] \in \texttt{seq}(\alpha)$
2. $\forall\alpha{:}\texttt{TYPES}.\ \forall a{:}\,\alpha^T.\ \forall L{:}\,\texttt{seq}(\alpha).\ \ a \cdot L \in \texttt{seq}(\alpha)$
3. $\forall\alpha{:}\texttt{TYPES}.\ \forall T{:}\,\mathbf{U}.\ \forall L{:}\,\texttt{seq}(\alpha).\ \forall b{:}\,T.\ \forall F{:}\,\alpha^T \xrightarrow{2} \texttt{seq}(\alpha) \xrightarrow{2} T \to T.$
   $(\texttt{seqrec-case}\ L\ \texttt{of}\ \texttt{[]}\ \mapsto b \mid a \cdot L' \mapsto FL'.F[a, L', FL']) \in \texttt{seq}(\alpha)$

Abgeleitete Grundoperationen:

▷ Null-Abfrage: $\texttt{null?}(L)$
▷ Erstes Element: $\texttt{first}(L)$
▷ Außer erstem Element: $\texttt{rest}(L)$
▷ Boole'sche Gleichheit: $L = L'$

Typisierungen:

1. $\forall\alpha{:}\texttt{TYPES}.\ \forall L{:}\,\texttt{seq}(\alpha).\ \ \texttt{null?}(L) \in \mathbf{B}$
2. $\forall\alpha{:}\texttt{TYPES}.\ \forall L{:}\,\texttt{seq}(\alpha).\ \ \texttt{first}(L) \in \alpha^T$
3. $\forall\alpha{:}\texttt{TYPES}.\ \forall L{:}\,\texttt{seq}(\alpha).\ \ \texttt{rest}(L) \in \texttt{seq}(\alpha)$
4. $\forall\alpha{:}\texttt{TYPES}.\ \forall L{:}\,\texttt{seq}(\alpha).\ \forall L'{:}\,\texttt{seq}(\alpha).\ \ (L = L') \in \mathbf{B}$

Axiome:

$\forall\alpha{:}\texttt{TYPES}. \forall L, L'{:}\,\texttt{seq}(\alpha). \forall a, a'{:}\,\alpha^T. \forall T{:}\,\mathbf{U}. \forall b{:}\,T.$
$\forall F{:}\,\alpha \xrightarrow{2} \texttt{seq}(\alpha) \xrightarrow{2} T \to T. \forall P{:}\,\texttt{seq}(\alpha) \to \mathbf{P}_1.$

1. $(\texttt{seqrec-case}\ []\ \texttt{of}\ [] \mapsto b \mid a \cdot L' \mapsto FL'.F[a, L', FL']\ ) = b \in T$
2. $(\texttt{seqrec-case}\ a \cdot L\ \texttt{of}\ [] \mapsto b \mid a' \cdot L' \mapsto FL'.F[a', L', FL']\ ) =$
   $F[a, L, \texttt{seqrec-case}\ L\ \texttt{of}\ [] \mapsto b \mid a' \cdot L' \mapsto FL'.F[a', L', FL']\ ] \in T$
3. $[] \neq a \cdot L$
4. $a \cdot L = a' \cdot L' \iff (a = a' \wedge L = L')$
5. $\texttt{first}(a \cdot L) = a$
6. $\texttt{rest}(a \cdot L) = L$
7. $\texttt{null}([])$
8. $\neg\texttt{null}(a \cdot L)$
9. $(P[] \wedge \forall R{:}\,\texttt{seq}(\alpha). \forall y{:}\,\alpha^T.\ P\ R \Rightarrow P(y \cdot R)) \ \Rightarrow\ \forall R{:}\,\texttt{seq}(\alpha).\ P\ R$

Tab. 2.10. *Datentyp: Endliche Sequenzen (Listen)*

Trägertyp: $\mathtt{set}(\alpha)$
Typisierung: $\forall\alpha{:}\,\mathtt{TYPES}.\ \mathtt{set}(\alpha) \in \mathbf{U}$

Datentyp: $\mathtt{Set}(\alpha)$
Typisierung: $\forall\alpha{:}\,\mathtt{TYPES}.\ \mathtt{Set}(\alpha) \in \mathtt{TYPES}$

Grundoperationen:

▷ leere Menge: $\{\}$
▷ Addition: $S + x$
▷ Mengeninduktion: $\mathtt{setrec\text{-}case}\ S\ \mathtt{of}\ \{\} \mapsto b \mid x \mapsto r.t[x,r]$

Typisierungen:

1. $\forall\alpha{:}\,\mathtt{TYPES}.\ \{\} \in \mathtt{set}(\alpha)$
2. $\forall\alpha{:}\,\mathtt{TYPES}.\ \forall S{:}\,\mathtt{set}(\alpha).\ \forall x{:}\,\alpha^T.\ S + x \in \mathtt{set}(\alpha)$
3. $\forall\alpha{:}\,\mathtt{TYPES}.\ \forall T{:}\,\mathbf{U}.\ \forall S{:}\,\mathtt{set}(\alpha).\ \forall b{:}\,T.\ \forall t{:}\,\alpha^T \xrightarrow{2} T \xrightarrow{2} T.$
   t erfüllt spezielle Axiome
   $\Rightarrow\ \mathtt{setrec\text{-}case}\ S\ \mathtt{of}\ \{\} \mapsto b \mid x \mapsto r.t[x,r] \in T$

Abgeleitete Grundoperationen:

▷ Mengen-Allquantor: $\forall x \in S.\ P[x]$
▷ Mengen-Existenzquantor: $\exists x \in S.\ P[x]$
▷ Elementabfrage: $x \in S$
▷ Boole'sche Gleichheit: $S = S'$

Typisierungen:

1. $\forall\alpha{:}\,\mathtt{TYPES}.\ \forall S{:}\,\mathtt{set}(\alpha).\ \forall P{:}\,\alpha^T \to \mathbf{B}.\ (\forall x \in S.\ P[x]) \in \mathbf{B}$
2. $\forall\alpha{:}\,\mathtt{TYPES}.\ \forall S{:}\,\mathtt{set}(\alpha).\ \forall P{:}\,\alpha^T \to \mathbf{B}.\ (\exists x \in S.\ P[x]) \in \mathbf{B}$
3. $\forall\alpha{:}\,\mathtt{TYPES}.\ \forall S{:}\,\mathtt{set}(\alpha).\ \forall x{:}\,\alpha^T.\ (x \in S) \in \mathbf{B}$
4. $\forall\alpha{:}\,\mathtt{TYPES}.\ \forall S{:}\,\mathtt{set}(\alpha).\ \forall S'{:}\,\mathtt{set}(\alpha).\ (S = S') \in \mathbf{B}$

Axiome:

$\forall\alpha{:}\,\mathtt{TYPES}.\ \forall S,S'{:}\,\mathtt{set}(\alpha).\ \forall x,a{:}\,\alpha^T.\ \forall T{:}\,\mathbf{U}.\ \forall b{:}\,T.\ \forall f{:}\,\alpha \times T \to T.\ \forall P{:}\,\mathtt{set}(\alpha) \to \mathbf{P}_1.$

1. $\neg(a \in \{\})$
2. $x \in (S + a) \Leftrightarrow (x = a \vee x \in S)$
3. $(S + a) + x\ =\ (S + x) + a$
4. $(S + a) + a\ =\ S + a$
5. $\mathtt{setrec\text{-}case}\ \{\}\ \mathtt{of}\ \{\} \mapsto b\ \mid\ x \mapsto r.\ f\langle x,r\rangle\ =\ b \in T$
6. $(\forall z,z'{:}\,\alpha^T.\ \forall t{:}\,T.\ f\langle z, f\langle z',t\rangle\rangle = f\langle z', f\langle z,t\rangle\rangle \in T)$
   $\wedge(\forall z{:}\,\alpha^T.\ \forall t{:}\,T.\ f\langle z, f\langle z,t\rangle\rangle = f\langle z,t\rangle \in T)$
   $\Rightarrow \mathtt{setrec\text{-}case}\ S + a\ \mathtt{of}\ \{\} \mapsto b\ \mid\ x \mapsto r.\ f\langle x,r\rangle\ =$
   $f\langle a, \mathtt{setrec\text{-}case}\ S\ \mathtt{of}\ \{\} \mapsto b\ \mid\ x \mapsto r.\ f\langle x,r\rangle\rangle \in T$
7. $(P\{\} \wedge \forall R{:}\,\mathtt{set}(\alpha).\ \forall y{:}\,\alpha^T.\ P\ R \Rightarrow P(R + y))\ \Rightarrow\ \forall R{:}\,\mathtt{set}(\alpha).\ P\ R$

Tab. 2.11. *Datentyp: Endliche Mengen*

- Die Abstraktionen werden typisiert.
- Es werden die abstrakten Redizes definiert und Taktiken aufgestellt, wie die Kontraktion dieser Redizes durch primitive Redizes simuliert werden kann.
- Es werden die Axiome des Datentyps aufgestellt und bewiesen.
- Es werden Lemmata aufgestellt und bewiesen, die weitere Eigenschaften des Typs angeben.

Die Implementierung der Datentypen hat mit einer ganzen Reihe von Schwierigkeiten zu rechnen. In [stolpmann96] wird dies expemplarisch am Beispiel der endlichen Mengen diskutiert, der sich von den genannten als der mit Abstand schwierigste Datentyp herausgestellt hat. Schon die genaue Formalisierung der Simulation des Datentyps ist mitunter recht kompliziert. Dabei ist nicht nur problematisch, die einzelnen NuPRL-Abstraktionen zu definieren, sondern auch die Abfolge dieser Abstraktionen auszutüfteln. Einzubeziehen ist dabei, daß die einzelnen Abstraktionen stets typisierbar sein sollen und dies auch in Form von Wohlgeformtheitslemmata bewiesen wird. Einige Wohlgeformtheiten stellen sich dabei als schwierig heraus, es müssen zur Unterstützung dieser Beweise weitere Lemmata herangezogen werden. Es entsteht auf diese Weise ein Gefüge von Abhängigkeiten, das schwer zu überschauen ist. NuPRL bietet hierfür keinerlei Unterstützung, vielmehr muß durch eine geplante und disziplinierte Vorgehensweise dieses Gefüge in eine lineare Form gebracht werden, die NuPRL wieder versteht.

Im Augenblick ist die Implementierung der Datentypen so weit gediehen, daß sie grundsätzlich benutzt werden können. Dabei wurden längst nicht alle Lemmata bewiesen, die aufgestellt wurden; diese Lücken sind irgendwann noch zu schließen. Es gibt bisher kaum Theoreme, die über die Axiome hinausgehende Eigenschaften der Datentypen aussagen. Der systematische Aufbau einer Lemmadatenbank wäre hier angebracht.

2.3.3 Zwei Typen für die Wahrheit

Nachdem wir den boole'schen Typ $\mathbf{B}$ eingeführt haben, gibt es nun einen zweiten Typ neben $\mathbf{P}$, um Wahrheit auszudrücken. Beiden Typen haben dabei einen grundsätzlich verschiedenen Ursprung und einen jeweils anderen Anwendungsbereich:

- $\mathbf{B}$ hat Operationen, die berechenbar sind, während für die Operationen von $\mathbf{P}$ keine Reduktionen möglich sind. Der Wahrheitswert einer Proposition P wird statt durch Ausrechnen durch einen Beweis für $\vdash P$ (oder $\vdash \neg P$) bestimmt.
- $\mathbf{P}$ ist wesentlich ausdrucksstärker als $\mathbf{B}$. Zum einen kann jeder boole'sche Wert B mittels $B = \text{true} \in \mathbf{B}$ als Proposition aufgefaßt werden, zum anderen können Propositionen über unendliche Bereiche sprechen (etwa mittels All- oder Existenzquantor), was mit boole'schen Ausdrücken nicht möglich ist.

Die rein aussagenlogischen Operatoren $\wedge, \vee, \neg, \Rightarrow, \iff$ sind sowohl für $\mathbf{B}$ als auch für $\mathbf{P}$ möglich und haben in beiden Fällen eine vergleichbare Bedeutung. Mittels folgender Lemmata läßt sich jedes Vorkommen eines dieser Operatoren in das Pendant des anderen Wahrheitstyps umwandeln; dabei indizieren wir die Operatoren, um sie unterscheiden zu können, mit einem Index B oder P:

Lemma 2.3.1

Für beliebige $b, b' \in \mathbf{B}$ gilt

1. $\lceil b \rceil \wedge_P \lceil b' \rceil \;\;\Leftrightarrow\;\; \lceil b \wedge_B b' \rceil$

2. $\lceil b \rceil \vee_P \lceil b' \rceil \quad \Leftrightarrow \quad \lceil b \vee_B b' \rceil$
3. $\lceil b \rceil \Rightarrow_P \lceil b' \rceil \quad \Leftrightarrow \quad \lceil b \Rightarrow_B b' \rceil$
4. $(\lceil b \rceil \Leftrightarrow_P \lceil b' \rceil) \quad \Leftrightarrow \quad \lceil b \Leftrightarrow_B b' \rceil$
5. $\neg_P \lceil b \rceil \quad \Leftrightarrow \quad \lceil \neg_B b \rceil$

Die Operation, die durch die Klammern $\lceil \cdots \rceil$ notiert wird, wandelt dabei einen boole'schen Wert in einen propositionalen um.

2.4 Implementierungsebene, Bedingungsebene und Metaebene

Es ist zu erwarten, daß sich die Probleme, die mit den Aspekten des Rechnens, Typisierens oder Schließens zusammenhängen, von jeweils bestimmten Konstrukten der Typentheorie verursacht werden. Eine Strategie, damit umzugehen, besteht darin, Teilsprachen der Typentheorie durch syntaktische Restriktion zu definieren, die bestimmte Konstruktionen verbieten, so daß damit einige der Probleme gelöst werden können. Im folgenden wird ein Modell von drei Teilsprachen vorgestellt: APM, APM+$\mathbf{P_1}$ und Meta-APM. Dabei gelten die Inklusionen

$$\text{APM} \subset \text{APM+}\mathbf{P_1} \subset \text{Meta-APM}$$

weswegen wir auch von drei Sprachebenen sprechen.

Die Unterteilung in diese Ebenen werden wir durch ein Beispiel begründen und damit der Intuition zugänglich machen. Im weiteren Verlauf wird sich die genaue Abgrenzung immer weiter herauskristallisieren.

2.4.1 Leitbeispiel: Mischen zweier Listen

Um herauszufinden, welche Sprachebenen sinnvoll betrachtet werden können, spielen wir die Programmierung an einem Beispiel durch. Es sollen zwei Listen ganzer Zahlen, die bereits sortiert sind, gemischt werden. Dies kann wie folgt formal spezifiziert werden:

$\texttt{FUNCTION}\ \mathit{merge}(L_1\texttt{:Seq(}\mathbf{Z}\texttt{)},\ L_2\texttt{:Seq(}\mathbf{Z}\texttt{))}\texttt{:Seq(}\mathbf{Z}\texttt{)}$
$\quad\texttt{WHERE}\ L_1\ \text{sortiert} \wedge L_2\ \text{sortiert}$
$\quad\texttt{RETURNS}\ (L' \mid L_1 \cdot L_2\ \text{ist Permutation von}\ L' \wedge L'\ \text{sortiert}\ )$

Wie benötigen also eine Abstraktion, die eine in dieser Weise formalisierte Spezifikation ausdrücken kann. „Abstraktion" heißt dabei zum einen, mittels des NuPRL-Abstraktionsmechanismus einen neuen Operator zu definieren, der den Term, der allgemein Spezifikationen darstellt, abkürzt; es heißt aber vor allem, die Bedeutung einer Spezifikation von der Termdarstellung abzulösen und sie als eigenständigen Begriff aufzufassen (analog den ADT). Ersteres betrifft die NuPRL-Implementierung, letzteres unsere intuitive Auffassung. An die Abstraktion für die Spezifikation werden, um eine relative Eigenständigkeit zu ermöglichen, daher zwei weitere Abstraktionen direkt angeschlossen:

Das Programm P erfüllt die Spezifikation S

und

Spezifikation S ist erfüllbar (es gibt ein Programm P, das S erfüllt)

Diese drei Abstraktionen (Spezifikation, „erfüllt", „erfüllbar") sind kaum voneinander zu trennen und sind daher, obwohl jede für sich implementiert wird, als Einheit aufzufassen. Erst in ihrem Zusammenspiel werden Spezifikationen in abstrakter Weise verfügbar. Um die Einheit zu benennen, sprechen wir auch vom *abstrakten Typ* der Spezifikation. (Es ist schade, daß NuPRL den Terminus „Abstraktion" vorschnell für eine bestimmte Technik verwendet, ohne daß geklärt wäre, worin die Abstraktheit bestehen soll.)

In der obigen Spezifikation für das Mischen kommen Operatoren vor, die Eigenschaften von Listen angeben: „L sortiert" und „L ist Permutation von L'". Auch diese Operatoren werden durch NuPRL-Abstraktionen definiert, sind jedoch nicht an der Konstruktion eines abstrakten Typs beteiligt. Wir nennen solche Abstraktionen, die einen Aspekt schon bestehender Typen herausstellen, auch *singuläre Abstraktionen.*

Um nun zu einem Algorithmus für das Mischen zu kommen, benutzen wir die Induktion über die Listenelemente von L_1. Man kann das Axiom für die Induktion so formulieren, daß es mittels Spezifikationen ausgedrückt wird:

> Die Spezifikation
>
> `FUNCTION` f`(`x`:Seq(`α`),` y`:`β`):`γ `WHERE` $I(x,y)$ `RETURNS` $(z \mid O(x,y,z))$
>
> ist erfüllbar, falls die Spezifikation des Basisfalls
>
> `FUNCTION` f_b`(`y`:`β`):`γ `WHERE` $I([],y)$ `RETURNS` $(z \mid O([],y,z))$
>
> erfüllbar ist und die Spezifikation des Induktionsschritts
>
> `FUNCTION` f_s`(`x_h`:`α`,` x_t`:Seq(`α`),` z_t`:`γ`,` y`:`β`):`γ `WHERE` $I(x_h \cdot x_t, y) \quad \wedge \quad (I(x_t,y) \Rightarrow O(x_t,y,z_t))$
> `RETURNS` $(z \mid O(x_h \cdot x_t, y, z))$
>
> erfüllbar ist.

In diesem typentheoretischen Satz steckt implizit das Konstruktionsverfahren für den Algorithmus, den man dem Extraktterm entnehmen kann, wenn der Beweis abgeschlossen ist. Die Anwendung dieses Satzes auf das Mischproblem reduziert dies auf zwei Teilprobleme:
Spezifikation des Basisfalls:

> `FUNCTION` $merge_b$`(`L_2`:Seq(`$\mathbf{Z}$`)):Seq(`$\mathbf{Z}$`)`
> `WHERE` $[]$ sortiert $\wedge$ L_2 sortiert
> `RETURNS` $(L' \mid [] \cdot L_2$ ist Permutation von $L' \wedge L'$ sortiert $)$

Spezifikation des Induktionsschritts:

> `FUNCTION` $merge_s$`(`L_h`:`$\mathbf{Z}$`,` L_t`:Seq(`$\mathbf{Z}$`),` L_t'`:Seq(`$\mathbf{Z}$`),` L_2`:Seq(`$\mathbf{Z}$`)) :Seq(`$\mathbf{Z}$`)`
> `WHERE` $(L_h \cdot L_t$ sortiert $\wedge$ L_2 sortiert$)$ $\wedge$
> L_t sortiert $\wedge$ L_2 sortiert $\Rightarrow$ $L_t \cdot L_2$ ist Permutation von $L_t' \wedge L_t'$ sortiert
> `RETURNS` $(L' \mid L_h \cdot L_t \cdot L_2$ ist Permutation von $L' \wedge L'$ sortiert $)$

Der Basisfall wird nun durch den trivialen Algorithmus $merge_b(L_2) = L_2$ gelöst, was man allerdings noch zeigen muß. Neben sehr einfachen logischen Schlüssen stützt sich dieser Beweis im Wesentlichen auf die Gültigkeit von

> $[] \cdot L_2$ ist Permutation von L_2

ab. Dies kann durch die Gesetze, daß [] neutrales Element ist und daß die Permutationseigenschaft reflexiv ist, leicht gezeigt werden. – Wir sehen, daß wir bereits zwei Arten von Sätzen benutzt haben. Auf der einen Seite haben wir die ursprüngliche Spezifikation in zwei einfachere Spezifikationen transformiert, zum anderen Terme umgeformt, die Bedingungen in Programmen ausdrücken.

Der Algorithmus für den Induktionsschritt $merge_s$ benötigt wiederum eine Induktion, die nach dem gleichen Muster abläuft. Zuletzt erhält man durch Extraktion folgende Programme (die zweite Induktion ist dabei nicht weiter aufgeschlüsselt):

$$\begin{aligned} merge(L_1, L_2) \equiv\ & \texttt{seqrec-case}\ L_1\ \texttt{of} \\ & \quad [\,] \mapsto merge_b(L_2) \\ & \quad L_h \cdot L_t \mapsto L_t'.merge_s(L_h, L_t, L_t', L_2) \\ merge_b(L_2) \equiv\ & L_2 \end{aligned}$$

$$\begin{aligned} merge_s(h, t, z', L) \equiv\ & \texttt{seqrec-case}\ L\ \texttt{of} \\ & \quad [\,] \mapsto [h] \\ & \quad L_h \cdot L_t \mapsto L_t'.\texttt{if}\ L_h < h\ \texttt{then}\ L_h \cdot L_t'\ \texttt{else}\ h \cdot L_h \cdot L_t \end{aligned}$$

Wir haben gesehen, daß wir verschiedenartige Sprachelemente benutzt haben, die in eigentümlicher Beziehung zu den allgemeinen Aspekten Rechnen, Typisierung und Schließen stehen:

1. Der abstrakte Typ der Spezifikationen spricht einem hypothetischen Programm bestimmte Eigenschaften zu. Es gibt sinnvolle Sätze, die Spezifikationen in Teilspezifikationen herunterbrechen, d.h. man kann über Spezifikationen vor allem *schließen.*

2. In den Spezifikationen stehen Bedingungen, in denen Programmfragmente logisch verknüpft werden. Diese Fragmente können ggf. reduziert werden; es können Inferenzen auf die Bedingungen als ganzes angewandt werden.

3. Zuletzt erhält man einen Programmtext sowie ein Theorem, das garantiert, daß dieses Programm die Spezifikation erfüllt. Der Programmtext kann für konkrete Eingaben ausgewertet werden.

Diese drei Arten von Termen haben wir mit folgenden, definierten Sprachbereichen identifiziert. Jeder Sprachbereich wird durch syntaktische Restriktionen für die Element- und Typterme festgelegt.

3. APM, das abstrakte Programmier-Modell: Durch Elementterme von APM sollen Programmtexte ausgedrückt werden. Es werden nur Typterme zugelassen, die in Programmen sinnvoll sind und die, noch zu besprechende, Eigenschaften von APM bezüglich der Automatisierung des Typisierens ermöglichen.

2. APM+$\mathbf{P}_1$, die Erweiterung um die Prädikatenlogik 1. Stufe: Elementterme dieses Sprachbereichs können zusätzlich Operatoren enthalten, die prädikatenlogische Formeln ausdrücken. Dazu gehören Junktoren ($\wedge$, $\vee$, $\neg$ usw.), Quantoren ($\forall$, $\exists$) sowie bestimmte Relationen ($=$, $<$).

1. Meta-APM, welches APM-Terme thematisiert und Bedingungen aus APM+$\mathbf{P}_1$ enthält: Wir machen keine Einschränkungen für den Gebrauch der Typentheorie.

Im folgenden untersuchen wir, wie sich die Aspekte des Programmierens Rechnen, Typisieren und Schließen zu den drei Sprachebenen verhalten. Eine Zusammenfassung findet sich in Tabelle 2.12.

		Rechnen	**Typisieren**			**Schließen**	
	Restriktion		Bestimmung von Typkandidaten	Typprüfung	Typisierungsbeweis	Instanzierungsproblem	Vollständigkeit
Meta-APM	keine	marginale Bedeutung	heuristisch	nicht möglich	halbautomatisch mit starker Lemma-Unterstützung	praktisch unlösbar	nein
APM $+\mathbf{P_1}$	auf APM und den Typ $\mathbf{P}_1$ mit prädikatenlogischen Operatoren	wie APM	ja	ja	ja, bis auf Aspekt der bedingten Typisierung	lösbar	sofern die benutzten Theorien vollständig sind
APM	ausgewälte Typkonstrukte und ausgewählte Operatoren	alle typisierbaren APM-Terme sind normalisierbar (total)	ja	ja	ja, bis auf Aspekt der bedingten Typisierung	nicht sinnvoll	

Tab. 2.12. *Unterschiede der Sprachebenen in Bezug auf Programmieraspekte*

2.4.2 Problematisierung des Rechnens für die Sprachebenen

Hauptsächlich spielen sich Berechnungen im Sprachbereich APM ab, aber dies muß nicht so sein. Die Erweiterung auf APM+$\mathbf{P_1}$ bringt zunächst keine neuen Möglichkeiten, da sich Propositionen in der Typentheorie grundsätzlich nicht auswerten lassen. Oder anders gesagt: Bevor man auswerten kann, muß zunächst der Schluß gezogen werden, daß Auswertung möglich ist, z.B. durch Anwendung der Lemmata 2.3.1 (s.o.). Innerhalb von Meta-APM spielen Berechnungen eine geringe Rolle, meistens haben sie nur den Sinn, eine bessere Darstellung zu ermöglichen. Beispielsweise kann man mittels des Produktraums zusammengehörige Typen, Aussagen oder andere Entitäten zu einer Einheit zusammenschließen; man sehe sich hierauf einmal den Typ der Spezifikationen an.

Grundsätzlich sind typisierbare Terme der Typentheorie normalisierbar, d.h. es gibt stets einen Auswertungsweg, der zu einer Normalform führt, die nicht weiter auswertbar ist. Die Normalform ist eindeutig. Nicht jede Strategie zur Auswertung findet jedoch die Normalform, da es Fälle typisierbarer Terme gibt, bei denen nichtterminierende Teilterme vorkommen können. Es gibt eine Auswertungsstrategie (*lazy evaluation*), die dies vermeidet.

Die Normalisierbarkeit von Termen ist von deren Typisierbarkeit abhängig. Damit ist Typisierbarkeit das wichtigere Kriterium, dem wir uns jetzt zuwenden.

2.4.3 Problematisierung der Typisierbarkeit für die Sprachebenen

Wie wir bereits in den vorangegangenen Beispielen gesehen haben, sind mehrere Begriffe im Zusammenhang mit Typisierbarkeit sinnvoll. In der folgenden Definition wird offengelassen, ob Restriktionen für t oder T vorausgesetzt werden oder nicht.

Definition 2.4.1 (Typisierbarkeit)

Ein Term t heißt typisierbar für einen Typ T, falls es einen Beweis für $\vdash t \in T$ gibt.

Definition 2.4.2 (Typprüfung)

Wir sprechen von einer Typprüfung $t \in_? T$, falls nur bestimmte Aspekte der behaupteten Typzugehörigkeit von t zu T überprüft werden. Ein entsprechender Typprüfungsalgorithmus gibt „wahr" oder „falsch" zurück, wenn die genannten Aspekte die Typzugehörigkeit zulassen bzw. ihr widersprechen.

Eine Typprüfung $t \in_? T$ heißt falsifizierend, sofern der negative Ausgang des Typprüfungsalgorithmus impliziert, daß $\vdash t \in T$ nicht gilt. Mit anderen Worten: Falls der Algorithmus „falsch" berechnet, stimmt dies auch, wenn er „wahr" berechnet, heißt dies nur, daß er keinen Einwand gefunden hat.

Umgekehrt heißt eine Typprüfung verifizierend, wenn aus dem positiven Ausgang des Algorithmus' folgt, daß $\vdash t \in T$ gilt.

Eine Typprüfung, die falsifizierend und verifizierend ist, heißt streng.

Definition 2.4.3 (Typbestimmung)

Wir sprechen von einer Typbestimmung eines Terms t, falls ein Algorithmus einen zu t in bestimmter Hinsicht passenden Typ T berechnet oder t zurückweist.

Eine Typbestimmung heißt falsifizierend, falls aus der Zurückweisung folgt, daß es keinen Typ T gibt mit $\vdash t \in T$.

Eine Typbestimmung heißt verifizierend, falls aus der Berechnung eines Typs T folgt, daß $\vdash t \in T$ gilt.

Eine Typbestimmung heißt streng, wenn sie falsifizierend und verifizierend ist.

Eine Typbestimmung, die weder falsifizierend noch verifizierend ist, nennen wir auch heuristisch.

Treibend für die Unterscheidung der Sprachebenen ist die Idee, für APM nur solche Typkonstruktoren zuzulassen, die möglichst positive Ergebnisse für die gerade aufgeworfenen Typisierungsfragen erlauben. Es ist natürlich noch offen, welche Aspekte in praktisch benutzbare Typprüfung- und Typbestimmungsalgorithmen eingebaut werden sollten, und ob falsifizierende oder verifizierende Algorithmen anzustreben sind.

In Hinblick auf die Möglichkeit, Algorithmen zu finden, sind die folgenden Punkte zu beachten:

- *Das Problem der Extensionalität:* Es ist möglich, daß zwei unterschiedliche Terme T und T' denselben Typ bezeichnen ($T = T'$). Dies muß natürlich in alle Algorithmen eingebaut sein. Offensichtlich darf man nicht zu stark von der Extensionalität Gebrauch machen, es muß zum einen immer entscheidbar bleiben, ob $T = T'$ gilt, zum anderen müssen wichtige Problemstellungen (z.B. Matching, Unifikation) modulo der Typ-Extensionalität lösbar sein.

  Durch die `TYPES`-Abstraktion von Datentypen gibt es für diese Typen immer zwei darstellende Terme. Die dadurch hervorgerufene Extensionalität ist allerdings harmlos (obwohl lästig).

  Zu umfangreiche Extensionalität bedeutet, daß die Typprüfungs- und -bestimmungsalgorithmen nicht mehr falsifizieren können, da sie, wenn sie mit zwei Darstellungen T und T' desselben Typs konfrontiert werden und dies nicht erkennen, sie diese falsche Entscheidung in die Irre führt und sie scheitern, obwohl die Typisierbarkeit gegeben ist.

- *Das Problem bedingter Typzuschreibungen:* Wie wir am Beispiel von Wf-Lemmata schon gesehen haben, sind manche Typisierungsmöglichkeiten davon abhängig, daß bestimmte Bedingungen erfüllt werden. Es ist offensichtlich unpraktikabel, diese Bedingungen innerhalb von Typprüfungs- oder -bestimmungsalgorithmen nachzuweisen, da es sich hier um ein Problem des Schließens handelt. Läßt man die Bedingungen beseite, so sind verifizierende Algorithmen bereits nicht mehr möglich.

- *Das Problem multipler Typzuschreibungen:* Wenn bedingte Typisierungen möglich sind, so spricht zunächst nichts dagegen, einen Term t mit T_1 zu typisieren, falls B_1 erfüllt ist und mit T_2 zu typisieren, falls B_2 gilt. Damit erkauft man sich jedoch wieder Extensionalität, da $\lambda x.t$ nur mit einem abhängigen Funktionenraum typisiert werden kann und in diesem B_1 und B_2 extensional (in anderer Darstellung) vorkommen können. Multiple bedingte Typisierbarkeiten sind also nur eine implizite Form von Extensionalität.

Sollen nun falsifizierende oder verifizierende Algorithmen bevorzugt werden? Da, wenn man bedingte Typzuschreibungen erlaubt, letztere gar nicht möglich sind, aber Bedingungen notwendig für die beim Programmieren häufigen partiell definierten Terme sind, muß das Ziel der (vollautomatischen) Verifikation wohl oder übel aufgegeben werden. Es bleibt noch die Falsifikation, die durch die Extensionalität berührt wird.

Extensionalität wird durch die Konstrukte des abhängigen Funktionen- und Produktraums in beliebiger Weise möglich. Die Verwendung dieser Konstrukte kann aber beschränkt werden. Beispielsweise ist es tolerierbar, daß die abhängigen Variablen Elemente von Universen sind, da dann diese Variablen nicht in Berechnungen eingehen können. Ob und wie weit diese Beschränkungen gelockert werden können, ohne zu viel Extensionalität zu ermöglichen, die der Falsifizierbarkeit entgegenstünde, ist offen.

Für den Zweck des Programmierens haben die abhängigen Typen keinen großen Wert, so daß man auf sie leicht verzichten kann. Die Falsifizierbarkeit ist in jedem Fall eine wichtigere Forderung.

Fassen wir also zusammen:

- Innerhalb von APM sollen falsifizierende Typprüfungs- und Typbestimmungsalgorithmen möglich sein.
- Die vollautomatische Verifikation wird nicht angestrebt, jedoch sollen Taktiken, die Typbeweise konstruieren, die von ihnen nicht erfaßten Annahmen als Unterziele zurückgeben, und zwar möglichst so, daß diese beweisbar sind, wenn Typisierbarkeit gegeben ist.

Bezüglich der Typisierbarkeit ist die Hinzunahme von $\mathbf{P_1}$ problemlos, so daß diese Ziele auch für APM+$\mathbf{P_1}$ gelten.

In Meta-APM können wir die Beschränkung der abhängigen Räume nicht hinnehmen, da diese Konstrukte hier sehr wertvoll sind. Es bleibt damit nur, auch das Ziel der Falsifikation fallen zu lassen. Für die Praxis heißt dies, daß keine vollautomatischen Taktiken mehr sinnvoll sind und man ein Zusammenspiel von manueller Beweisführung mit automatischer Unterstützung ermöglichen sollte. Die Autotaktik ist hierfür ein gutes Beispiel.

2.4.4 Problematisierung des Schließens für die Sprachebenen: Das Instanziierungsproblem

Manche Schlüsse sind daran gekoppelt, den Typ eines Terms zu wissen, der daran beteiligt ist. Zwei Fälle sind interessant:

- *Taktische Regeln mit Typargument:* Wir wir schon am Beispiel des Produktraums gesehen haben, benötigen manche primitiven Inferenzregeln ein Argument, das den Typ eines Terms enthält.

- *Instanziierung von Lemmata:* Es soll ein Lemma benutzt werden, das folgende Form hat:

$$\forall T\colon \mathbf{U}_i.\ \forall x\colon X_{\{T\}}.\ \ldots \Rightarrow A_{\{x\}}$$

  Dabei ist entscheidend, daß im Typ X die Typvariable T vorkommt, in A, der Konsequenz des Lemmas, aber nur x und nicht T.

  Es sei A' die mittels des Lemmas zu beweisende Formel. Die beiden Variablen x und T müssen dazu so mit Termen t_x und t_T instanziiert werden, daß $A' = A[t_x/x]_1$ syntaktisch gilt. Allerdings kommt in dieser Gleichung T nicht vor, weshalb man t_T auf diese Weise nicht bestimmen kann.

  Falls eine Typbestimmung möglich ist, bietet sich jedoch ein Ausweg. Man bestimmt den konkreten Typ T_x von t_x, und versucht, mittels der Gleichung $T_x = X[t_T/T]$ den Term t_T zu bekommen.

  Wir bezeichnen als *Instanziierungsproblem*, wenn man für ein gegebenes Lemma obiger Form alle Typvariablen und diejenigen Elementvariablen, die in der Konsequenz des Lemmas vorkommen, so instanziiert, daß das Lemma angewandt werden kann.

Es lassen sich zwei Arten von Lemmata in Hinblick auf das Instanziierungsproblem unterscheiden:

1. Lemmata, die dem Bereich APM+$\mathbf{P_1}$ zugeordnet werden können: Diese Lemmata zeigen Eigenschaften der Elementoperatoren dieses Bereichs. Da eine Typbestimmung möglich ist, kann das Instanziierungsproblem berechnet werden.

2. Lemmata, die dem Bereich Meta-APM zugeordnet werden können: Hierunter fallen insbesondere alle Sätze, die Spezifikationen transformieren oder zeigen, daß Programmschemata bestimmte Spezifikationen erfüllen. Eine Typbestimmung ist nur heuristisch möglich, weswegen das Instanziierungsproblem ungelöst bleibt. Der Beweis mittels Lemmata läuft immer Gefahr, bereits an dieser ersten Hürde zu scheitern.

   Übrigens gehören auch einige Eigenschaften von Elementoperatoren zum Bereich Meta-APM, z.B. Induktionsaxiome. Dies liegt daran, daß über propositionale Funktionen quantifiziert wird: $\forall P\colon X \to \mathbf{P}_1.\ \ldots$ Ggf. ist daran zu denken, daß speziell für diesen Zweck ein weiterer Sprachbereich isoliert wird (APM+$\mathbf{P_2}$), da diese Funktionen eigentlich noch recht gut typisierbar sind.

Summa sumarum sind die Chancen für eine Automatisierbarkeit der Lemmaanwendung im Bereich APM+$\mathbf{P_1}$ sehr gut, während im Bereich Meta-APM nur darauf gehofft werden kann, daß alle Instanzen bestimmbar sind.

2.4.5 Resümee

Wir haben gesehen, daß es wichtige Zusammenhänge zwischen den Aspekten Rechnen, Typisieren und Schließen gibt, wobei insbesondere das Typisieren eine fundamentale Rolle hat. Die Trennung in Sprachbereiche ergibt, daß für den Programmierkern APM+$\mathbf{P_1}$ positive Ergebnisse in punkto Typisierbarkeit erzielt werden können. Insbesondere rückt die vollautomatische Anwendung von Lemmata dieses Bereichs in greifbare Nähe.

2.5 Das abstrakte Programmier-Modell: Ebenen APM und APM+P_1

Die nächsten beiden Abschnitte stellen einen Ansatz dar, Typprüfung für einen restringierten Sprachbereich durchzuführen. Dabei werden Systeme von Typgleichungen eine besondere Rolle spielen.

Die eigentliche Prüfung besteht darin, den zu prüfenden Term aufgrund vorgegebener Typisierungsansätze in ein Typgleichungssystem (TGS) zu überführen und dies aufzulösen. Der Hindley-Milner-Algorithmus kann auf diese Weise als TGS-Lösungsalgorithmus gedeutet werden. Im Anschluß an diese Typprüfungsmethode werden wir die konkreten Restriktionen diskutieren, mittels derer der Bereich APM+$\mathbf{P_1}$ definiert wird, so daß für ihn Typprüfung möglich wird.

2.5.1 Typgleichungssysteme

Vorbemerkung für diesen Abschnitt: Ich habe hier versucht, die Typprüfung mit algebraischen Mitteln darzustellen. Der Grund dafür ist, daß man dadurch nicht sofort Algorithmen angeben muß, sondern eine theoretische Haltung gegenüber der Problemstellung einnehmen kann. Allerdings handelt es sich nicht um mehr als eine Skizze; die Theorie ist nicht bis ins Letzte durchdacht. Ich kann noch nicht einmal garantieren, daß sinnvoll definiert wurde; Beweise fehlen ja ohnehin. Man behandle diesen Ansatz als eine Darstellung, die mehr eine Richtung zur Lösung der Probleme angibt als die Lösung selbst.

In der folgenden Definition gehen wir davon aus, daß die Typterme in einem noch zu fassenden Sinne wohlgeformt sind. Grundsätzlich soll damit sichergestellt werden, daß mittels dieser Terme nur Typen ausgedrückt werden können; die genaue Definition der Wohlgeformtheit hängt von der beabsichtigten Restriktion ab.

Definition 2.5.1 (Typvariablendeklaration, Typgleichung)

> *Eine Typvariablendeklaration D ist eine Liste $T_1 : \hat{U}_1, \ldots T_n : \hat{U}_n$ von Paaren $T_i : \hat{U}_i$, wobei T_i ein Variablenname ist und $\hat{U}_i$ ein Universum oder* `TYPES`. *In der Liste darf nicht zweimal derselbe Name vorkommen. Es ist $V(D)$ die Menge der Variablennamen, die in D deklariert werden.*
>
> *Eine Typgleichung für eine Typvariablendeklaration D ist ein Paar $\langle \hat{S}, \hat{S}' \rangle$ von wohlgeformten Typtermen mit $Free(\hat{S}) \subseteq V(D)$ und $Free(\hat{S}') \subseteq V(D)$. Eine solche Typgleichung notieren wir auch als*
>
> $$\hat{S} \stackrel{*}{=} \hat{S}'$$
>
> *Einige Variablen aus $V(D)$ haben nur die Rolle von Typkonstanten, die anderen die Rolle von Unbekannten, die gesucht sind. Um dies zu unterscheiden, verwenden wir für erstere Namen $R, S, T, \ldots$ und für letztere Namen X, Y, Z.*
>
> *Die Lösung einer Typgleichung $\hat{S} \stackrel{*}{=} \hat{S}'$ ist eine Substitution σ für die Variablen $V(D)$, so daß $\sigma(\hat{S})$ und $\sigma(\hat{S}')$ denselben Term ergeben (bis auf α-Umbenennungen). σ kann eine Substitution erster oder zweiter Ordnung sein.*

Wir bemerken, daß die Lösung einer Typgleichung natürlich auf die Unbekannten beschränkt bleiben sollte. Es kann durchaus vorkommen, daß nicht alle Unbekannten (nichtidentisch) substituiert werden, z.B. kann $X \to Y \stackrel{*}{=} X \to \mathbf{Z}$ durch $\sigma_0 = [\mathbf{Z}/Y]_1$ gelöst werden. Eine besondere Rolle spielen Lösungen, die nicht mehr substituieren als notwendig. Beispielsweise ist σ_0 sogar eine allgemeinste (*prinzipielle*) Lösung, weil X nicht unnötig konkretisiert wird.

Definition 2.5.2 (Prinzipielle Lösung)

> *Eine Lösung σ einer Typgleichung $\hat{S} \stackrel{*}{=} \hat{S}'$ heißt prinzipiell, falls für jede Lösung τ der Gleichung gilt:*
>
> $$\tau(\sigma(\hat{S})) \stackrel{*}{=} \tau(S) \text{ und } \tau(\sigma(\hat{T})) \stackrel{*}{=} \tau(T)$$

Ohne Beweis geben wir an, daß jede lösbare Typgleichung genau eine prinzipielle Lösung (bis auf α-Umbenennungen) hat.

Nun ist diese Art von Typgleichung festgelegt im Hinblick auf die unterliegende Gleichheit von Typen: Zwei Typen gelten genau dann als gleich, wenn sie durch denselben Term repräsentiert werden. Dies ist zu starr; wir möchten, daß wir flexibel festlegen können, welche Typen gleich sind. Insbesondere möchten wir zusätzliche Gleichheiten fordern können. Folgende Definition ist etwas unsauber, da wir nicht alle benutzten Termini eingeführt haben. Eine präzise Fassung würde den Rahmen dieser Arbeit jedoch sprengen.

Definition 2.5.3 (Gleichungstheorie)

Es sei $\mathcal{G}$ eine (endliche) Menge von Typgleichungen einer Deklaration D. Dann bezeichnen wir mit $Th(\mathcal{G})$ die (ggf. unendliche) Menge von Typgleichungen, die aus $\mathcal{G}$ semantisch folgen, wobei die übliche Semantik der Gleichheit gilt. (Also: Eine Gleichung gilt für eine Struktur, falls sie für alle Belegungen erfüllt ist; $Mod(\mathcal{G})$ sei die Menge aller Modellstrukturen, d.h. aller Strukturen, für die alle Gleichungen $\mathcal{G}$ gelten; $Eq_D(M)$ seien alle Gleichungen mit Variablen aus D, die in allen Strukturen aus M gelten; dann ist $Th(\mathcal{G}) = Eq_D(Mod(\mathcal{G}))$.)

Unter einer Lösung einer Typgleichung modulo einer Gleichungstheorie verstehen wir nun eine Substitution σ, die nicht mehr unbedingt beide Seiten der Gleichung identisch machen muß, sondern die ausreicht, eine zur Gleichungstheorie gehörige Gleichung zu bilden:

Definition 2.5.4 (Lösung modulo einer Gleichungstheorie)

Eine Lösung einer Typgleichung $\hat{S} \stackrel{*}{=} \hat{S}'$ modulo einer Gleichungstheorie für eine Gleichungsmenge $\mathcal{G}$, die allesamt für eine Typvariablendeklaration D gegeben sind, ist eine Substitution σ von Variablen aus $V(D)$, so daß

$$\sigma(\hat{S}) \stackrel{*}{=} \sigma(\hat{S}') \in Th(\mathcal{G})$$

Ein einfaches Beispiel für $\mathcal{G}$ ist die Beachtung der durch die Einführung der Datentypen `TYPES` hervorgerufene Verdoppelung der Typkonstruktoren. Es gibt in der Regel zwei Konstruktoren, einen für den Trägertyp und einen für den Datentyp. Seien die beiden Konstruktoren mit op_T und op_D bezeichnet, dann gilt eine Gleichung der Art

$$op_T(\langle\text{Teilterme}\rangle) = (op_D(\langle\text{Teilterme}\rangle))^T \in \mathbf{U}_j$$

innerhalb der Typentheorie; dabei wird mit $\cdot^T$ die Operation bezeichnet, die den Trägertyp aus einem Datentyp zurückgibt. Für jedes Paar von Konstruktoren ist es daher gerechtfertigt, eine entsprechende Typgleichung in die Gleichungsmenge mit aufzunehmen. Die so erhaltene Gleichungsmenge nennen wir auch $\mathcal{G}_{\texttt{TYPES}}$.

Um anzudeuten, daß wir uns für die Lösungen modulo $\mathcal{G}$ interessieren, notieren wir Typgleichungen auch in der Form

$$\hat{S} \stackrel{*}{=} \hat{S}' \;/\; \mathcal{G}$$

Definition 2.5.5 (Typgleichungssystem (TGS))

Ein Typgleichungssystem $\mathcal{S}$ für eine Typvariablendeklaration D ist eine Menge von Gleichungen für D.

Eine Substitution σ von Variablen aus $V(D)$ heißt Lösung von $\mathcal{S}$, falls σ für alle Gleichungen $G \in \mathcal{S}$ Lösung ist. Entsprechend überträgt sich auch die Lösung modulo einer Gleichungstheorie auf TGS.

Eine Lösung σ heißt prinzipiell, falls für alle Substitutionen τ von Variablen aus $V(D)$ gilt, daß $\tau(\sigma(\hat{S})) \overset{*}{=} \tau(\hat{S})) \in Th(\mathcal{G})$ und $\tau(\sigma(\hat{S}')) \overset{*}{=} \tau(\hat{S}')) \in Th(\mathcal{G})$ für jede Gleichung $\hat{S} \overset{*}{=} \hat{S}' \in \mathcal{S}$, wobei $\mathcal{G}$ die zugrundeliegende Gleichungstheorie ist.

Wir notieren für $\mathcal{S} = \{G_1, ..., G_n\}$ auch $\begin{pmatrix} G_1 \\ \vdots \\ G_n \end{pmatrix}$.

2.5.2 Typisierungsansätze und Aufstellung eines TGS

Betrachten wir zunächst an Beispielen, wie wir zu typisierende Terme in TGS umsetzen. Zu Behandlung von Teiltermen ist dabei die λ-Applikation $s_1\ s_2$ recht typisch, während wir die Wirkung von Bindungsvariablen gut an der λ-Abstraktion $\lambda x.s$ studieren können. Beide Aspekte sind zusammenzuführen und zu verallgemeinern.

Um einer Applikation $s_1\ s_2$ einen Typ Z zuzuordnen, gehen wir davon aus, daß wir die Typen von s_1 und s_2 schon kennen; seien dies Z_1 bzw. Z_2. Dabei möge es sich um prinzipielle Lösungen der entsprechenden Teilprobleme handeln. In dem gesuchten TGS werden Z_1 und Z_2 durch einen Teil der Gleichungen festgelegt, den wir hier als berechnet voraussetzen und nicht weiter angeben.

Um den Ansatz zu beschreiben, benötigen wir neue Variablen Y_1 und Y_2. Folgende Typgleichungen geben an, welche Bedingungen erfüllt werden müssen, damit die Applikation durch einen unabhängigen Funktionenraum typisierbar ist:

1. $Z_1 \overset{*}{=} Y_1 \rightarrow Y_2$
2. $Z_2 \overset{*}{=} Y_1$
3. $Z \overset{*}{=} Y_2$

Diese Gleichungen sind nun zum TGS hinzuzufügen. Die Auflösung nach Z ermittelt den Typ der Applikation.

Bei der Abstraktion haben wir zusätzlich noch eine Bindungsvariable. Wir nehmen an, X sei der Typ dieser Variable und Z_1 der Typ von s unter der Voraussetzung, daß $x \in X$. Wir erhalten als Typgleichungen:

1. $Z_1 \overset{*}{=} Y_1$
2. $Z \overset{*}{=} X \rightarrow Y_1$

Wir sehen an diesem Beispiel, daß wir eine Verwaltung der freien Variablen einführen müssen. Analog der Typvariablendeklaration sei F eine Deklaration freier Variablen. Bei der Bestimmung des Gleichungssystemteils, der Z_1 festlegt, nehmen wir $x : X$ in die Deklaration F auf; die Variable x kann nun adäquat in Typgleichungen eingehen, da wir X als Symbol für ihren Typ benutzen dürfen.

Unter einem *Typisierungsansatz* verstehen wir ein Schema, das angibt, unter welchen Voraussetzungen für die Bindungsvariablen und Teilterme ein Term, der mittels eines bestimmten Operators gebildet ist, zu typen ist. Wir verwenden die Notation D: Annahmen $\overset{*}{\Longrightarrow}$ Typisierung, z.B.

- für die λ-Applikation: $Y_1 : \mathbf{U},\ Y_2 : \mathbf{U} :\quad s_1 \in Y_1 \rightarrow Y_2, s_2 \in Y_1 \overset{*}{\Longrightarrow} (s_1\ s_2) \in Y_2$

- für die λ-Abstraktion: $Y_1 : \mathbf{U},\ X : \mathbf{U} : \quad s^1 \in X \stackrel{2}{\rightarrow} Y_1 \stackrel{*}{\Longrightarrow} (\lambda x.s^1[x]) \in X \rightarrow Y_1$

Man beachte, daß im Ansatz der λ-Abstraktion die Variable s^1 zweiter Ordnung ist. Die Annahme, daß x vom Typ X ist, wird ausgedrückt, indem x den Platz der ersten Stelle von s^1 einnimmt und die erste Stelle mit X typisiert wird.

Im weiteren gehen wir davon aus, daß es für jeden vorkommenden Operator (außer `variable`) einen solchen Ansatz gibt, und nennen dies ein Ansatzsystem. Dabei bleiben Bedingungen, die nicht in das Schema eines Ansatzes passen, außen vor; diese Bedingungen werden durch die Typprüfung auch nicht überprüft.

Um ein Verfahren zur Konstruktion eines TGS angeben zu können, benötigen wir zunächst einmal eine Präzisierung des Typisierungsansatzes.

Definition 2.5.6 (Typisierungsansatz)

Ein Typisiserungsansatz besteht aus einer Typvariablendeklaration D, einer Liste von Annahmen $A = v_1 \in \hat{V}_1, ..., v_n \in \hat{V}_n$ und einer Typzuschreibung $B = \hat{s} \in \hat{S}$. Die Variablen $W = \{v_i \mid 1 \leq i \leq n\}$ sind Variablen zweiter Ordnung, und die von ihnen angenommenen Typen $\hat{V}_i$ sind Typterme zweiter Ordnung, die der Stelligkeit von v_i entsprechen müssen. Die Variablen W sind genau die Variablen, die in $\hat{s}$ frei vorkommen. $\hat{S}$ ist ein wohlgeformter Typterm, in dem die Variablen W frei vorkommen dürfen.

Wir notieren den Typisierungsansatz auch als

$$D: \quad v_1 \in \hat{V}_1, ..., v_n \in \hat{V}_n \stackrel{*}{\Longrightarrow} \hat{s} \in \hat{S}$$

Nun können wir den oben frei durchgeführten Algorithmus zur Aufstellung des TGS angeben (Abbildung 2.13).

Wie man sieht, ist die Idee des Verfahrens, für die Typen der Teilterme und den Typ des Gesamtterms neue Typvariablen $Z_1, ..., Z_n$ sowie Z einzuführen. Diese Variablen werden mit den im Ansatz vorgesehenen Typtermen gleichgesetzt. Die Funktion von Z_i ist dabei, die Verbindung zum Teil-TGS G_i für den i-ten Teilterm herzustellen; die Funktion von Z ist, einen solchen Verbindungspunkt nach außen bereitzustellen.

Die Typausdrücke in den Listen L_i stehen für die angenommenen Typen der Bindungsvariablen. Es ist zu beachten, daß $\hat{S}_i$ ein Typterm zweiter Ordnung ist: $\hat{S}_i = T_1 \stackrel{2}{\rightarrow} T_2 \stackrel{2}{\rightarrow} ... \stackrel{2}{\rightarrow} T_k \stackrel{2}{\rightarrow} T$, wobei T_j der Typ der j-ten Stelle und T der resultierende Typ ist. Mit $\mathit{Result}(\hat{S}_i)$ wird nur der Anteil T zurückgegeben. Mit $\mathit{Type}(j; \hat{S}_i)$ wird T_j zurückgegeben.

In Tabelle 2.14 ist ein Trace des Algorithmus' für eine Beispielformel abgebildet. Die Teilterme der Formel sind mit t_1 bis t_{11} durchnumeriert; wie man sieht, traversiert der Algorithmus die Teilterme. Wir gehen davon aus, daß mit $\mathcal{A}_{std}$ ein System von Typisierungsansätzen existiert, das für die primitven Operatoren der Typentheorie funktioniert (siehe Abb. 2.16). Der Typ des i-ten Teilterms wird symbolisch mit Z_i bezeichnet. Daneben werden noch Typvariablen X_j benutzt, die ebenfalls global durchnumeriert sind; dies sind die in den Typisierungsansätzen benutzten Variablen. Es ist darauf zu achten, daß bei jedem Teilterm die Typvariablen des Ansatzes neue Namen erhalten; beispielsweise wird hier der Ansatz für den Operator `pair` zweimal benutzt, aber mit je anderen Typvariablen (siehe t_5 und t_8). Die X_j werden in den Ansätzen benutzt, um die Typen der Teilterme und den Typ des Ergebnisses eines Operators allgemein formulieren zu können; in zwei Anwendungen desselben Operators werden in der Regel implizit unterschiedliche Bedingungen an diese Typen gestellt, so daß je eigene Variablen erforderlich sind.

Typprüfungs-TGS$(\hat{t}; \mathcal{A}; Z; F)$
mit $\hat{t}$: zu typisierender Term
$\mathcal{A}$: zu benutzendes System von Typisierungsansätzen
Z: zu verwendende Lösungsvariable (die nirgendwo sonst vorkommt)
F: Deklarationen für die freien Variablen in $\hat{t}$

Falls $\hat{t} = v$, (v Variable)
schlage den Typ $\hat{T}$ von v in F nach und gib das TGS
$(Z \stackrel{*}{=} \hat{T})$
zurück.

Falls $\hat{t} = op(V_1.\hat{t}_1; ...; V_n.\hat{t}_n)$ (dabei V_i: Bindungsvariablen, $\hat{t}_i$: Teilterme)
Sei A_{op} der in $\mathcal{A}$ für op vorgesehene Typisierungsansatz.
Sei
$$A_{op} = D: \quad s_1^{\mu_1} : \hat{S}_1,\ ...,\ s_n^{\mu_n} : \hat{S}_n \stackrel{*}{\Longrightarrow} op(V_1.s_1^{\mu_1}[V_1];\ ...;\ V_n.s_n^{\mu_n}[V_n]) \in \hat{S}$$
(wobei ggf. die Typvariablen in neue Typvariablen umbenannt sind)

Sei $L_i = [\mathit{Type}(j, \hat{S}_i) \mid 1 \leq j \leq \mu_i]$
Seien $Z_1, ..., Z_n$ neue Typvariablen für die Lösungen der Teil-TGS
Sei $F_i = \{v : \hat{X} \mid (v : \hat{X}) \in F,\ v \notin V_i\} \cup$
$\{v : \hat{X} \mid v \in V_i,\ \hat{X}$ ist Typausdruck aus L_i für $v\}$
Sei G_i = Typprüfungs-TGS$(\hat{t}_i; \mathcal{A}; Z_i; F_i)$

Ergebnis ist folgendes TGS:
$$\begin{pmatrix} Z \stackrel{*}{=} \hat{S} \\ Z_1 \stackrel{*}{=} \mathit{Result}(\hat{S}_1) \\ \vdots \\ Z_n \stackrel{*}{=} \mathit{Result}(\hat{S}_n) \end{pmatrix} \bigcup G_1 \bigcup \cdots \bigcup G_n$$

Abb. 2.13. *Algorithmus zur TGS-Berechnung*

Der Aufruf der Funktion *Typprüfungs-TGS* enthält neben den bereits erläuterten Argumenten (t_i, $\mathcal{A}_{std}$ und Z_i) noch ein viertes Argument. Hierbei handelt es sich darum, daß für die jeweils freien Variablen des Teilterms Typannahmen gemacht werden; diese können dem Typisierungsansatz entnommen werden.

Die Behandlung des Teilterms schließt jeweils auf zweierlei Weise an die weiteren Berechnungen an: Erstens wird mit Z_i die Verbindung zum umliegenden Teilterm hergestellt, indem die neue Gleichung aufgenommen wird, daß Z_i gleich dem Typ ist, den der Typisierungsansatz für diesen Term fordert. Zweitens werden Verbindungen zu den nächstinneren Teiltermen aufgebaut. Dies geschieht zum einen darüber, daß die Z_i dieser Teilterme wiederum denjenigen Typen gleichgesetzt werden, die der Ansatz postuliert; zum anderen werden die Typen für die in den Teiltermen neu vorkommenden freien Variablen über das vierte Argument des Funktionsaufrufs übergeben. Der Typ einer Variablen irgendwo im Inneren dieses Teilterms kann nun einfach dadurch bestimmt werden, daß im Wert dieses Arguments nachgeschlagen wird. – Die erzeugten Gleichungen werden aufgesammelt und vereinigt. Zum Schluß erhält man das Typgleichungssystem für den gesamten Term.

In der Trace-Tabelle wurden gegenüber dem Algorithmus einige Vereinfachungen durchgeführt. Erstens werden die Variablen Z, G und X global indiziert. Die Variablen F_i und L_i sind nur Zwi-

Formel: $\lambda x.\,\underbrace{\text{let}\ \underbrace{\langle a,b\rangle = \underbrace{(\text{rec-case}\ \underbrace{x}_{t_4}\ \text{of}\ [\,] \mapsto \underbrace{\langle \underbrace{x}_{t_6}, \underbrace{0}_{t_7}\rangle}_{t_5} \ \mid\ h{::}x' \mapsto r.\ \underbrace{\langle \underbrace{x}_{t_9}, \underbrace{h}_{t_{10}}\rangle}_{t_8})}_{t_3}\ \text{in}\ \underbrace{a}_{t_{11}}}_{t_2}}_{t_1}$

Typprüfungs-TGS$(t_1, \mathcal{A}_{std}, Z_1, \{\})$:

- $A_{\texttt{lambda}} =$ "$s^1 \in X_1 \xrightarrow{2} X_2 \overset{*}{\Longrightarrow} (\lambda x.s^1[x]) \in X_1 \to X_2$"
- $G_1 =$ Typprüfungs-TGS$(t_2, \mathcal{A}_{std}, Z_2, \{x\colon X_1\})$:
  - $A_{\texttt{spread}} =$ "$e^0 \in X_3 \times X_4,\ u^2 \in X_3 \xrightarrow{2} X_4 \xrightarrow{2} X_5 \overset{*}{\Longrightarrow} (\text{let}\ \langle a,b\rangle = e^0\ \text{in}\ u^2[a,b]) \in X_5$"
  - $G_2 =$ Typprüfungs-TGS$(t_3, \mathcal{A}_{std}, Z_3, \{x\colon X_1\})$:
    - $A_{\texttt{list\_ind}} =$ "$s^0 \in X_6\ \text{list},\ b^0 \in X_7,\ u^3 \in X_6 \xrightarrow{2} X_6\ \text{list} \xrightarrow{2} X_7 \xrightarrow{2} X_7$
      $\overset{*}{\Longrightarrow} (\text{rec-case}\ s^0\ \text{of}\ [\,] \mapsto b^0 \ \mid\ h{::}x' \mapsto r.\ u^3[h,x',r]) \in X_7$"
    - $G_3 =$ Typprüfungs-TGS$(t_4, \mathcal{A}_{std}, Z_4, \{x\colon X_1\})$:
      - Ergebnis: $(Z_4 \overset{*}{=} X_1)$
    - $G_4 =$ Typprüfungs-TGS$(t_5, \mathcal{A}_{std}, Z_5, \{x\colon X_1\})$:
      - $A_{\texttt{pair}} =$ "$s^0 \in X_8,\ u^0 \in X_9 \overset{*}{\Longrightarrow} \langle s^0, u^0\rangle \in X_8 \times X_9$"
      - $G_5 =$ Typprüfungs-TGS$(t_6, \mathcal{A}_{std}, Z_6, \{x\colon X_1\})$: • Ergebnis: $(Z_6 \overset{*}{=} X_1)$
      - $G_6 =$ Typprüfungs-TGS$(t_7, \mathcal{A}_{std}, Z_7, \{x\colon X_1\})$:
        - $A_{\texttt{natural\_number}} =$ "$\overset{*}{\Longrightarrow} 0 \in \mathbf{Z}$" • Ergebnis: $(Z_7 \overset{*}{=} \mathbf{Z})$
      - Ergebnis: $\left(\begin{array}{lcl} Z_5 & \overset{*}{=} & X_8 \times X_9 \\ Z_6 & \overset{*}{=} & X_8 \\ Z_7 & \overset{*}{=} & X_9 \end{array}\right) \cup G_5 \cup G_6$
    - $G_7 =$ Typprüfungs-TGS$(t_8, \mathcal{A}_{std}, Z_8, \{x\colon X_1,\ h\colon X_6,\ x'\colon X_6\ \text{list},\ r\colon X_7\})$:
      - $A_{\texttt{pair}} =$ "$s^0 \in X_{10},\ u^0 \in X_{11} \overset{*}{\Longrightarrow} \langle s^0, u^0\rangle \in X_{10} \times X_{11}$"
      - $G_8 =$ Typpr.-TGS$(t_9, \mathcal{A}_{std}, Z_9, \{x\colon X_1,\ h\colon X_6,\ x'\colon X_6\ \text{list},\ r\colon X_7\})$:
        - Ergebnis: $(Z_9 \overset{*}{=} X_1)$
      - $G_9 =$ Typpr.-TGS$(t_{10}, \mathcal{A}_{std}, Z_{10}, \{x\colon X_1,\ h\colon X_6,\ x'\colon X_6\ \text{list},\ r\colon X_7\})$:
        - Ergebnis: $(Z_{10} \overset{*}{=} X_6)$
      - Ergebnis: $\left(\begin{array}{lcl} Z_8 & \overset{*}{=} & X_{10} \times X_{11} \\ Z_9 & \overset{*}{=} & X_{10} \\ Z_{10} & \overset{*}{=} & X_{11} \end{array}\right) \cup G_8 \cup G_9$
    - Ergebnis: $\left(\begin{array}{lcl|lcl} Z_3 & \overset{*}{=} & X_7 & Z_5 & \overset{*}{=} & X_7 \\ Z_4 & \overset{*}{=} & X_6\ \text{list} & Z_8 & \overset{*}{=} & X_7 \end{array}\right) \cup G_3 \cup G_4 \cup G_7$
  - $G_{10} =$ Typprüfungs-TGS$(t_{11}, \mathcal{A}_{std}, Z_{11}, \{x\colon X_1,\ a\colon X_3,\ b\colon X_4\})$: • Ergebnis: $(Z_{11} \overset{*}{=} X_3)$
  - Ergebnis: $\left(\begin{array}{lcl|lcl|lcl} Z_2 & \overset{*}{=} & X_5 & Z_3 & \overset{*}{=} & X_3 \times X_4 & Z_{11} & \overset{*}{=} & X_5 \end{array}\right) \cup G_2 \cup G_{10}$
- Ergebnis: $\left(\begin{array}{lcl} Z_1 & \overset{*}{=} & X_1 \to X_2 \\ Z_2 & \overset{*}{=} & X_2 \end{array}\right) \cup G_1 = \left(\begin{array}{lcl|lcl|lcl} Z_1 & \overset{*}{=} & X_1 \to X_2 & Z_5 & \overset{*}{=} & X_7 & Z_7 & \overset{*}{=} & \mathbf{Z} \\ Z_2 & \overset{*}{=} & X_2 & Z_8 & \overset{*}{=} & X_7 & Z_8 & \overset{*}{=} & X_{10} \times X_{11} \\ Z_2 & \overset{*}{=} & X_5 & Z_4 & \overset{*}{=} & X_1 & Z_9 & \overset{*}{=} & X_{10} \\ Z_3 & \overset{*}{=} & X_3 \times X_4 & Z_5 & \overset{*}{=} & X_8 \times X_9 & Z_{10} & \overset{*}{=} & X_{11} \\ Z_{11} & \overset{*}{=} & X_5 & Z_6 & \overset{*}{=} & X_8 & Z_9 & \overset{*}{=} & X_1 \\ Z_3 & \overset{*}{=} & X_7 & Z_7 & \overset{*}{=} & X_9 & Z_{10} & \overset{*}{=} & X_6 \\ Z_4 & \overset{*}{=} & X_6\ \text{list} & Z_6 & \overset{*}{=} & X_1 & Z_{11} & \overset{*}{=} & X_3 \end{array}\right)$

Tab. 2.14. *oben: typgeprüfte Formel, unten: Trace der Funktionsaufrufe von Typprüfungs-TGS*

Ausgangs-TGS:

$$
G = \left(\begin{array}{clcl|clcl|clcl}
1 & Z_1 & \overset{*}{\doteq} & X_1 \to X_2 & 8 & Z_5 & \overset{*}{\doteq} & X_7 & 15 & Z_7 & \overset{*}{\doteq} & \mathbf{Z} \\
2 & Z_2 & \overset{*}{\doteq} & X_2 & 9 & Z_8 & \overset{*}{\doteq} & X_7 & 16 & Z_8 & \overset{*}{\doteq} & X_{10} \times X_{11} \\
3 & Z_2 & \overset{*}{\doteq} & X_5 & 10 & Z_4 & \overset{*}{\doteq} & X_1 & 17 & Z_9 & \overset{*}{\doteq} & X_{10} \\
4 & Z_3 & \overset{*}{\doteq} & X_3 \times X_4 & 11 & Z_5 & \overset{*}{\doteq} & X_8 \times X_9 & 18 & Z_{10} & \overset{*}{\doteq} & X_{11} \\
5 & Z_{11} & \overset{*}{\doteq} & X_5 & 12 & Z_6 & \overset{*}{\doteq} & X_8 & 19 & Z_9 & \overset{*}{\doteq} & X_1 \\
6 & Z_3 & \overset{*}{\doteq} & X_7 & 13 & Z_7 & \overset{*}{\doteq} & X_9 & 20 & Z_{10} & \overset{*}{\doteq} & X_6 \\
7 & Z_4 & \overset{*}{\doteq} & X_6\ \mathrm{list} & 14 & Z_6 & \overset{*}{\doteq} & X_1 & 21 & Z_{11} & \overset{*}{\doteq} & X_3
\end{array}\right)
$$

Auflösung:

Nummer	Gleichung (nach Substitution)		neue Substitution / Änderung der Substitution
21:	$Z_{11} \overset{*}{\doteq} X_3$	•	$Z_{11} \setminus X_3$
20:	$Z_{10} \overset{*}{\doteq} X_6$	zusätzlich	$Z_{10} \setminus X_6$
19:	$Z_9 \overset{*}{\doteq} X_1$	zusätzlich	$Z_9 \setminus X_1$
18:	$X_6 \overset{*}{\doteq} X_{11}$	geändert:	$Z_{10} \setminus X_{11},\quad X_6 \setminus X_{11}$
17:	$X_1 \overset{*}{\doteq} X_{10}$	geändert:	$Z_9 \setminus X_{10},\quad X_1 \setminus X_{10}$
16:	$Z_8 \overset{*}{\doteq} X_{10} \times X_{11}$	zusätzlich	$Z_8 \setminus X_{10} \times X_{11}$
15:	$Z_7 \overset{*}{\doteq} \mathbf{Z}$	zusätzlich	$Z_7 \setminus \mathbf{Z}$
14:	$Z_6 \overset{*}{\doteq} X_{10}$	zusätzlich	$Z_6 \setminus X_{10}$
13:	$\mathbf{Z} \overset{*}{\doteq} X_9$	zusätzlich	$X_9 \setminus \mathbf{Z}$
12:	$X_{10} \overset{*}{\doteq} X_8$	zusätzlich	$X_8 \setminus X_{10}$
11:	$Z_5 \overset{*}{\doteq} X_{10} \times \mathbf{Z}$	zusätzlich	$Z_5 \setminus X_{10} \times \mathbf{Z}$
10:	$Z_4 \overset{*}{\doteq} X_{10}$	zusätzlich	$Z_4 \setminus X_{10}$
9:	$X_{10} \times X_{11} \overset{*}{\doteq} X_7$	zusätzlich	$X_7 \setminus X_{10} \times X_{11}$
8:	$X_{10} \times \mathbf{Z} \overset{*}{\doteq} X_{10} \times X_{11}$	•	$Z_{11} \setminus X_3,\ Z_{10} \setminus \mathbf{Z},\ Z_9 \setminus X_{10},\ X_6 \setminus \mathbf{Z},\ X_1 \setminus X_{10},\ Z_8 \setminus X_{10} \times \mathbf{Z},\ Z_7 \setminus \mathbf{Z},\ Z_6 \setminus X_{10},\ X_9 \setminus \mathbf{Z},\ Z_5 \setminus X_{10} \times \mathbf{Z},\ Z_4 \setminus X_{10},\ X_7 \setminus X_{10} \times \mathbf{Z},\ X_{11} \setminus \mathbf{Z}$
7:	$X_{10} \overset{*}{\doteq} \mathbf{Z}\ \mathrm{list}$	•	$Z_{11} \setminus X_3,\ Z_{10} \setminus \mathbf{Z},\ Z_9 \setminus \mathbf{Z}\ \mathrm{list},\ X_6 \setminus \mathbf{Z},\ X_1 \setminus \mathbf{Z}\ \mathrm{list},\ Z_8 \setminus \mathbf{Z}\ \mathrm{list} \times \mathbf{Z},\ Z_7 \setminus \mathbf{Z},\ Z_6 \setminus \mathbf{Z}\ \mathrm{list},\ X_9 \setminus \mathbf{Z},\ Z_5 \setminus \mathbf{Z}\ \mathrm{list} \times \mathbf{Z},\ Z_4 \setminus \mathbf{Z}\ \mathrm{list},\ X_7 \setminus \mathbf{Z}\ \mathrm{list} \times \mathbf{Z},\ X_{11} \setminus \mathbf{Z},\ X_{10} \setminus \mathbf{Z}\ \mathrm{list}$
6:	$Z_3 \overset{*}{\doteq} \mathbf{Z}\ \mathrm{list} \times \mathbf{Z}$	zusätzlich	$Z_3 \setminus \mathbf{Z}\ \mathrm{list} \times \mathbf{Z}$
5:	$X_3 \overset{*}{\doteq} X_5$	zusätzlich	$X_5 \setminus X_3$
4:	$\mathbf{Z}\ \mathrm{list} \times \mathbf{Z} \overset{*}{\doteq} X_3 \times X_4$	•	$Z_{11} \setminus \mathbf{Z}\ \mathrm{list},\ Z_{10} \setminus \mathbf{Z},\ Z_9 \setminus \mathbf{Z}\ \mathrm{list},\ X_6 \setminus \mathbf{Z},\ X_1 \setminus \mathbf{Z}\ \mathrm{list},\ Z_8 \setminus \mathbf{Z}\ \mathrm{list} \times \mathbf{Z},\ Z_7 \setminus \mathbf{Z},\ Z_6 \setminus \mathbf{Z}\ \mathrm{list},\ X_9 \setminus \mathbf{Z},\ Z_5 \setminus \mathbf{Z}\ \mathrm{list} \times \mathbf{Z},\ Z_4 \setminus \mathbf{Z}\ \mathrm{list},\ X_7 \setminus \mathbf{Z}\ \mathrm{list} \times \mathbf{Z},\ X_{11} \setminus \mathbf{Z},\ X_{10} \setminus \mathbf{Z}\ \mathrm{list},\ Z_3 \setminus \mathbf{Z}\ \mathrm{list} \times \mathbf{Z},\ X_5 \setminus \mathbf{Z}\ \mathrm{list},\ X_3 \setminus \mathbf{Z}\ \mathrm{list}$
3:	$Z_2 \overset{*}{\doteq} \mathbf{Z}\ \mathrm{list}$	zusätzlich	$Z_2 \setminus \mathbf{Z}\ \mathrm{list}$
2:	$\mathbf{Z}\ \mathrm{list} \overset{*}{\doteq} X_3$	zusätzlich	$X_2 \setminus \mathbf{Z}\ \mathrm{list}$
1:	$Z_1 \overset{*}{\doteq} \mathbf{Z}\ \mathrm{list} \to \mathbf{Z}\ \mathrm{list}$	zusätzlich	$Z_1 \setminus \mathbf{Z}\ \mathrm{list} \to \mathbf{Z}\ \mathrm{list}$
		Ergebnis:	$Z_{11} \setminus \mathbf{Z}\ \mathrm{list},\ Z_{10} \setminus \mathbf{Z},\ Z_9 \setminus \mathbf{Z}\ \mathrm{list},\ X_6 \setminus \mathbf{Z},\ X_1 \setminus \mathbf{Z}\ \mathrm{list},\ Z_8 \setminus \mathbf{Z}\ \mathrm{list} \times \mathbf{Z},\ Z_7 \setminus \mathbf{Z},\ Z_6 \setminus \mathbf{Z}\ \mathrm{list},\ X_9 \setminus \mathbf{Z},\ Z_5 \setminus \mathbf{Z}\ \mathrm{list} \times \mathbf{Z},\ Z_4 \setminus \mathbf{Z}\ \mathrm{list},\ X_7 \setminus \mathbf{Z}\ \mathrm{list} \times \mathbf{Z},\ X_{11} \setminus \mathbf{Z},\ X_{10} \setminus \mathbf{Z}\ \mathrm{list},\ Z_3 \setminus \mathbf{Z}\ \mathrm{list} \times \mathbf{Z},\ X_5 \setminus \mathbf{Z}\ \mathrm{list},\ X_3 \setminus \mathbf{Z}\ \mathrm{list},\ Z_2 \setminus \mathbf{Z}\ \mathrm{list},\ X_2 \setminus \mathbf{Z}\ \mathrm{list},\ Z_1 \setminus \mathbf{Z}\ \mathrm{list} \to \mathbf{Z}\ \mathrm{list}$

Tab. 2.15. *Auflösung des Typgleichungssystems durch sukzessives Bestimmen der prinzipiellen Lösung*

schengrößen und wurden weggelassen. Der Typisierungsansatz wurde gleich in der Form notiert, die sich durch Umbenennungen ergibt. Dies betrifft sowohl die Typvariablen X_i als auch die Bindungsvariablen, die denjenigen der Originalformel angeglichen wurden.

2.5.3 Algorithmen zur Lösung eines TGS

Wir können uns nun der Auflösung der Typgleichungssysteme widmen. Erst an dieser Stelle müssen wir eventuell zu Grunde gelegte Gleichungstheorien beachten, da bis jetzt Typgleichungen nur syntaktisches Konstrukt waren, ohne Bedeutung, nun aber festgelegt werden muß, was eine Lösung ist und was nicht.

Zur Auflösung einzelner Gleichungen bieten sich die bekannten Unifikationsalgorithmen an, in die man auch die Berücksichtigung von Gleichungstheorien einbauen kann. Insbesondere eignet sich der Martelli-Montanari-Algorithmus, der die Unifikation auf Gleichungsbasis durchführt und um Termersetzungsregeln erweitert werden kann. Vorausgesetzt wird dabei, daß die Gleichungstheorie mittels des Knuth-Bendix-Verfahrens in ein Termersetzungssystem umgewandelt wird.

Die aktuelle Implementierung berücksichtigt ohnehin nur die Theorie $\mathcal{G}_{\mathbf{TYPES}}$, die auch sehr gut in die Herbrand-Robinson-Unfikation eingebaut werden kann.

Wie geht man nun bei der Auflösung eines ganzen Gleichungssystems G vor? Dazu nutzen wir aus, daß wir einzelne Gleichungen prinzipiell lösen können. Wir spalten also eine Gleichung ab:

$$G \quad = \quad G' \ \dot{\cup} \ \hat{S} \stackrel{*}{=} \hat{S}'$$

Sei nun σ die prinzipielle Lösung der abgespaltenen Gleichung. Es sei $\sigma(G')$ das Restgleichungssystem, auf dessen Terme diese Lösung angewandt wurde.

Es gilt nun, daß die prinzipielle Lösung τ von $\sigma(G')$ auch die prinzipielle Lösung von $\sigma(G)$ ist, d.h. $\tau \circ \sigma$ ist die prinzipielle Lösung von G:

- *τ ist eine Lösung von $\sigma(G)$:* Dies liegt daran, daß σ prinzipiell war, denn deshalb löst $\tau \circ \sigma$ auch die abgespaltene Gleichung.

- *τ ist prinzipielle Lösung von $\sigma(G)$:* Dies liegt daran, daß τ prinzipielle Lösung von G' ist und zum anderen $\sigma(\hat{S} \stackrel{*}{=} \hat{S}')$ tautologisch ist, also keine Bedingung ausdrückt.

- *$\tau \circ \sigma$ ist prinzipielle Lösung von G:* ist nur eine andere Formulierung des Vorherigen.

Damit ist bereits ein rekursiver Algorithmus zur Auflösung des TGS gegeben: Man spaltet nacheinander alle Gleichungen ab und setzt die Lösung der abgespaltenen Gleichung in den verbliebenen Rest ein.

Tabelle 2.15 setzt das angefangene Beispiel fort. Die Gleichungen des TGS wurden durchnumeriert und in umgekehrter Reihenfolge abgespalten; dies ergibt meist eine relativ günstige Substitutionsbehandlung. Die rechte Spalte gibt den jeweiligen Stand der Substitution wider, der, bevor die nächste Gleichung gelöst werden kann, auf die Gleichung angewendet werden muß. Die dadurch ggf. modifizierte Gleichung wurde daher auf der linken Seite ebenfalls notiert.

Den Typ der gesamten Formel kann man nun ablesen, indem das Substitut für Z_1 bestimmt wird: $\mathbf{Z}$ list $\rightarrow$ $\mathbf{Z}$ list. Wie man sieht, handelt es sich bei dem typisierten Term um die identische Funktion, die jedoch aufgrund der implizierten Nebenbedingungen für die Typen nur für Listen ganzer Zahlen angewendet werden darf.

2.5.4 APM+P$_1$: Restriktionen, Typsystem, Ansätze

Nachdem wir nun eine Methode haben, mit der wir den Typ eines Terms bestimmen und überprüfen können, ist jetzt noch festzulegen, auf welche Weise diese angewandt wird. Desweiteren wollen wir natürlich wissen, welche Aspekte der Typisierung überhaupt überprüft werden und welche nicht.

Eng mit der Anwendbarkeit dieser Methode hängt die genaue Definition des Sprachbereichs zusammen, für den sie zuständig sein soll. Der Sprachbereich wird durch syntaktische Restriktion festgelegt, und zwar zum einen für die Typterme, zum anderen für die Elementterme.

Die Restriktion der Typterme ist dabei die entscheidendere Voraussetzung, da von ihr abhängt, in welchem Grad die Extensionalität für Typen zugelassen wird. In die Typprüfung geht diese Extensionalität als Gleichungstheorie $\mathcal{G}$ ein, und wir sollten nur solche $\mathcal{G}$ erlauben, für die die prinzipielle Lösung von Typgleichungen modulo $\mathcal{G}$ noch möglich ist.

Die Restriktion für die Elementterme wird dadurch definiert, daß wir nur solche Operatoren erlauben, für die ein Typisierungsansatz gegeben ist. Desweiteren soll dieser Ansatz natürlich nicht beliebig sein, sondern möglichst mit der tatsächlichen Typisierung des Operators übereinstimmen. Worin diese Übereinstimmung besteht, ist noch zu klären.

Um Typterme zu bilden, werden nur folgende Typkonstruktoren zugelassen: Die primitiven Konstruktoren **Z**, **Atom**, $S \to T$, $S \times T$, $S|T$, T list, $S \not\to T$ sowie die abstrakten Konstruktoren **B**, **P**, $\mathtt{seq}(\alpha)$, $\mathtt{set}(\alpha)$. Für alle Konstruktoren außer den Funktionenräumen $S \to T$ und $S \not\to T$ werden außerdem die entsprechenden Datentypkonstruktoren erlaubt, sowie $\cdot^T$, die Berechnung des Trägertyps eines Datentyps. Weitere Operatoren sind in Typtermen nicht zugelassen (s.u. für eine Diskussion weiterer Möglichkeiten).

Die Extensionalität der Typen ist mit dieser Auswahl beschränkt auf die Gleichheitstheorie $\mathcal{G}_{\mathtt{TYPES}}$, die, wie wir bereits wissen, problemlos implementiert werden kann.

Welche Aspekte der Typisierung können und sollen nun in die Typisierungsansätze aufgenommen werden? Die Möglichkeiten, die diese Ansätze bieten, sind zu schwach und zu stark zugleich:

- Bedingungen für die Typisierung können nicht ausgedrückt werden. Die Aussage des Ansatzes ist insofern zu schwach und muß durch den noch zu führenden Beweis der Bedingungen verstärkt werden.

- Multiple Ansätze für denselben Operator werden nicht unterstützt. Wie wir wissen, ist dies nicht weiter schlimm, da multiple Ansätze erst benötigt werden, wenn man die Extensionalität der Typen weiter ausbaut. Trotzdem gibt es eine „unechte“ Form multipler Ansätze, die auch mittels der restringierten Typen auftreten kann:

  $$\text{if } b \text{ then } 1 \text{ else } \text{"ABC"}$$

  Im allgemeinen ist solch ein Term nicht mehr typisierbar, da man keinen der Restriktion gemäßen Typterm findet, der die Abhängigkeit des Typs des Ergebnisses von b ausdrückt. Ist b jedoch, ggf. im Kontext, stets wahr oder stets falsch, so ist der Typ dieses Terms **Z** bzw. **Atom**. Der gewöhnliche Ansatz für den if-Operator ist jedoch zu stark, um diese Möglichkeit zu sehen: Er würde den Term als fehlerhaft getypt zurückweisen. – Dieses Phänomen ist natürlich nur von theoretischem Interesse, da man stets einen äquivalenten Term finden wird, der typprüfbar ist.

Damit haben wir ein Wahrheitskriterium für die Ansätze: Ein Ansatz für einen Operator ist korrekt, wenn jeder typisierbare Term mit diesem Operator, bei dem eine unechte multiple Typisierung nicht auftritt, dem Schema des Ansatzes entspricht[15].

[15] Eine genauere Definition dieses Kriteriums müßte rekursiv von der Korrektheit der Teilterm-Typprüfung auf die Korrektheit der Gesamtterm-Typprüfung schließen. Die Definition hätte mit einigen formalen Problemen zu kämpfen, deren Lösung ich im Augenlick nicht als produktiv ansehe.

Es wird jeweils die Typvariablendeklaration $R, S, T : \mathbf{U}$ vorausgesetzt:

- Alle Zahlkonstanten n: $\stackrel{*}{\Longrightarrow}$ $n \in \mathbf{Z}$
- $-, *, \div$, rem wie:
  $u \in \mathbf{Z},\ v \in \mathbf{Z} \stackrel{*}{\Longrightarrow} (u + v) \in \mathbf{Z}$
- $u \in \mathbf{Z},\ v \in \mathbf{Z},\ t_1 \in T,\ t_2 \in T \stackrel{*}{\Longrightarrow} (\text{if } u = v \text{ then } t_1 \text{ else } t_2) \in T$
- $u \in \mathbf{Z},\ v \in \mathbf{Z},\ t_1 \in T,\ t_2 \in T \stackrel{*}{\Longrightarrow} (\text{if } u < v \text{ then } t_1 \text{ else } t_2) \in T$
- $u \in \mathbf{Z},\ t_1^2 \in \mathbf{Z} \stackrel{2}{\to} T \stackrel{2}{\to} T,\ b \in T,\ t_2^2 \in \mathbf{Z} \stackrel{2}{\to} T \stackrel{2}{\to} T$
  $\stackrel{*}{\Longrightarrow} (\text{rec-case } u \text{ of } x < 0 \mapsto r.t_1^2[x, r] \mid 0 \mapsto b \mid x > 0 \mapsto r.t_2^2[x, r]) \in T$
- Alle Symbolkonstanten s: $\stackrel{*}{\Longrightarrow}$ $s \in \mathbf{Atom}$
- $u \in \mathbf{Atom},\ v \in \mathbf{Atom},\ t_1 \in T,\ t_2 \in T \stackrel{*}{\Longrightarrow} (\text{if } u = v \text{ then } t_1 \text{ else } t_2) \in T$
- $t^1 \in S \stackrel{2}{\to} T \stackrel{*}{\Longrightarrow} (\lambda x.t^1[x]) \in S \to T$
- $f \in S \to T,\ s \in S \stackrel{*}{\Longrightarrow} (f\ s) \in T$
- $s \in S,\ t \in T \stackrel{*}{\Longrightarrow} \langle s, t\rangle \in S \times T$
- $e \in S \times T,\ u^2 \in S \stackrel{2}{\to} T \stackrel{2}{\to} R \stackrel{*}{\Longrightarrow} (\text{let } \langle x, y\rangle = e \text{ in } u^2[x, y]) \in R$
- $s \in S \stackrel{*}{\Longrightarrow} (\text{inl } s) \in S|T$
- $t \in T \stackrel{*}{\Longrightarrow} (\text{inr } t) \in S|T$
- $e \in S|T,\ s^1 \in S \stackrel{2}{\to} R,\ t^1 \in T \stackrel{2}{\to} R$
  $\stackrel{*}{\Longrightarrow} (\text{case } e \text{ of inl } x \mapsto s^1[x] \mid \text{inr } x \mapsto t^1[x]) \in R$
- $\stackrel{*}{\Longrightarrow} [] \in T \text{ list}$
- $s \in T,\ t \in T \text{ list} \stackrel{*}{\Longrightarrow} (s{::}t) \in T \text{ list}$
- $s \in T \text{ list},\ b \in R,\ t^3 \in T \stackrel{2}{\to} T \text{ list} \stackrel{2}{\to} R \stackrel{2}{\to} R$
  $\stackrel{*}{\Longrightarrow} (\text{rec-case } s \text{ of } [] \mapsto b \mid x{::}l \mapsto r.t^3[x, l, r]) \in R$
- $t^2 \in (S \not\to T) \stackrel{2}{\to} S \stackrel{2}{\to} T \stackrel{*}{\Longrightarrow} (\text{letrec } f(x) = t^2[f, x]) \in S \not\to T$
- $f \in S \not\to T \stackrel{*}{\Longrightarrow} (\text{dom f}) \in S \to \mathbf{P}$
- $f \in S \not\to T,\ s \in S \stackrel{*}{\Longrightarrow} f(t) \in T$

Abb. 2.16. *Typisierungsansätze für die primitiven Operatoren*

Die auf den obigen Restriktionen basierende Typprüfung ist damit beinahe falsifizierend; genauer: falls keine unechten multiple Typisierungen vorkommen, ist sie falsifizierend.

Die Ansätze für die primitiven Operatoren der Typentheorie zeigt Abbildung 2.16. Die Ansätze für die abstrakten Operatoren können meistens ziemlich direkt aus dem entsprechenden Wf-Lemma abgelesen werden.

In der Praxis interessiert man sich übrigens nicht nur für die geschilderte Form der Typprüfung und -bestimmung. Vor allem interessiert häufig nicht der Typ des ganzen Terms, sondern der Typ eines Teilterms im Kontext des ganzen Terms. Beispielsweise ist der Typ der λ-Abstraktion in $(\lambda x.x)\ 1$ für sich genommen $T \to T$, im Kontext des Gesamtterms jedoch $\mathbf{Z} \to \mathbf{Z}$. Die beschriebene Methode zur Typbestimmung kann leicht abgewandelt werden, um auch solche Probleme lösen zu können; man muß nur das TGS nach der Variable auflösen, die für den interessanten Teilterm steht.

Desweiteren möchte man natürlich auf der Basis dieser Typbestimmung eine Taktik haben, die diese verläßliche Typinformationen zur Beweiskonstruktion benutzt. Dies ist grundsätzlich möglich, hat aber mit dem Problem zu kämpfen, daß die Typisierungsbedingungen, die man bei der reinen Typbestimmung unter den Tisch fallen lassen kann, nun zumindest in einer Form aufbereiten muß, daß diese möglichst, sofern sie wahr sind, auch bewiesen werden können. Im Augenblick gibt es dazu noch keine Methode.

Spezifikationen:

$\triangleright$ $\mathtt{SPECS} \equiv D{:}\,\mathtt{TYPES} \times (R{:}\,\mathtt{TYPES} \times ((D^T \to \mathbf{P}) \times (D^T \times R^T \to \mathbf{P})))$
$\triangleright$ $\mathtt{D}(spec) \equiv$ let $\langle D, R, I, O\rangle = spec$ in D
$\triangleright$ $\mathtt{R}(spec) \equiv$ let $\langle D, R, I, O\rangle = spec$ in R
$\triangleright$ $\mathtt{I}(spec) \equiv$ let $\langle D, R, I, O\rangle = spec$ in I
$\triangleright$ $\mathtt{O}(spec) \equiv$ let $\langle D, R, I, O\rangle = spec$ in O
$\triangleright$ $\mathtt{FUNCTION}\ f(x{:}\,D){:}\,R\ \mathtt{WHERE}\ I^1[x]\ \mathtt{RETURNS}\ (y \mid O^2[x,y])$
$\equiv \langle D, R, (\lambda x.I^1[x]), (\lambda\langle x,y\rangle.O^2[x,y])\rangle$

Mengenwertige Spezifikationen:

$\triangleright$ $spec* \equiv \langle \mathtt{D}(spec), \mathtt{Set}(\mathtt{R}(spec)), \mathtt{I}(spec), \lambda\langle x,z\rangle.\ \forall y{:}\,\mathtt{R}(spec).\ y \in z \iff \mathtt{O}(spec)\langle x,y\rangle\ \rangle$
$\triangleright$ $\mathtt{FUNCTION}\ f(x{:}\,D){:}\,\mathtt{Set}(R)\ \mathtt{WHERE}\ I^1[x]\ \mathtt{RETURNS}\ \{y \mid O^2[x,y]\}$
$\equiv (\langle D, R, (\lambda x.I^1[x]), (\lambda\langle x,y\rangle.O^2[x,y])\rangle)*$

Programme und Korrektheit:

$\triangleright$ $\mathtt{BODYS}(spec) \equiv \mathtt{D}(spec)^T \not\to \mathtt{R}(spec)^T$
$\triangleright$ $body\ \mathtt{halts\ on}\ x \equiv \mathrm{dom}(body)(x)$
$\triangleright$ $\mathtt{PROGS} \equiv spec{:}\,\mathtt{SPECS} \times \mathtt{BODYS}(spec)$
$\triangleright$ $prog\ \mathtt{terminates}$
$\equiv$ let $\langle spec, body\rangle = prog$ in $\forall x{:}\,\mathtt{D}(spec).\ \mathtt{I}(spec)(x) \Rightarrow body\ \mathtt{halts\ on}\ x$
$\triangleright$ $prog\ \mathtt{is\ p\text{-}correct}$
$\equiv$
let $\langle spec, body\rangle = prog$ in $\forall x{:}\,\mathtt{D}(spec).\ \mathtt{I}(spec)(x) \Rightarrow body\ \mathtt{halts\ on}\ x \Rightarrow \mathtt{O}(spec)\langle x, body\ x\rangle$
$\triangleright$ $prog\ \mathtt{is\ correct}$
$\equiv prog\ \mathtt{terminates} \land prog\ \mathtt{is\ p\text{-}correct}$
$\triangleright$ $body\ \mathtt{satisfies}\ spec \equiv \langle spec, body\rangle\ \mathtt{is\ correct}$
$\triangleright$ $spec\ \mathtt{is\ satisfiable} \equiv \exists body{:}\,\mathtt{BODYS}(spec).\ body\ \mathtt{satisfies}\ spec$

Abb. 2.17. *Abstraktionen von Spezifikationen und Programmen*

2.6 Das abstrakte Programmier-Modell: Ebene Meta-APM

Kommen wir nun zu dem Sprachbereich, der Terme aus APM zum Objekt macht: Meta-APM. In diesem Kapitel beschäftigen wir uns dabei lediglich mit den Abstraktionen, die zu diesem Zweck zu definieren sind; der eigentliche Sinn von Meta-APM ist ja, über Programme zu schließen, was wir am Beispiel der Globalsuche in der ganzen restlichen Arbeit thematisieren.

2.6.1 Spezifikationen und Programme

Als eine Spezifikation betrachten wir formal ein Vier-Tupel $\langle D, R, I, O\rangle$, wobei D und R Datentypen sind, I die Eingabebedingung $I \in D^T \to \mathbf{P}$ und O die Ausgabebedingung $O \in D^T \times R^T \to \mathbf{P}$ ist. Wir sehen, daß die Typen von I und O von den Werten von D und R abhängig sind; daher ist dieses Vier-Tupel nur mit einem *abhängigen* Produktraum zu typisieren. In Abbildung 2.17 definieren wir die Abstraktion SPECS, die diesen Produktraum abkürzt, sowie vier Zugriffsoperatoren, die jeweils ein Element des Vier-Tupels zurückgeben, und außerdem eine lesbarere Notation für die Vier-Tupel. Es werden dazu einige nicht-primitive Operatoren verwendet: Ein Vierfach-let sowie eine spezielle λ-Abstraktion, die zugleich ihr Argument, welches ein Paar sein muß, in seine Elemente zerlegt. Diese Operatoren machen erhebliche Typisierungsprobleme.

Eine Variante von Spezifikation spielt bei der Globalsuche eine besondere Rolle: die *mengenwertige* Spezifikation. Zu einer gegebenen Spezifikation *spec* spezifiziert das zugehörige mengenwertige Problem ein Programm, das zu einer gegebenen Eingabe x die endliche Menge *aller* Ergebnisse y, die nach *spec* möglich sind, berechnet. Beispielsweise ist

$$\texttt{FUNCTION sqrt}(x{:}\,\mathbf{Z}){:}\,\mathbf{Z}\ \texttt{WHERE}\ x \geq 0\ \texttt{RETURNS}\ (y \mid y * y = x)$$

das Wurzelproblem. Einem Programm, das eine Lösung berechnet, ist freigestellt, ob es die positive oder negative Wurzel zurückgibt. Ein Programm, das das zugehörige mengenwertige Problem berechnet, gibt eine Menge $\{w, -w\}$ mit beiden (allen) Lösungen zurück.

Abbildung 2.17 zeigt die $\cdot*$-Abstraktion, die eine Spezifikation mengenwertig transformiert sowie eine Notation für mengenwertige Probleme.

Ein Programmkörper ist eine partielle Funktion, die zu einer Spezifikation paßt. Ein Programm ist ein Paar aus Spezifikation und passendem Programmkörper. Ein Programm *terminiert*, wenn der Programmkörper für alle erlaubten Eingaben hält. Ein Programm ist *partiell korrekt*, wenn, sofern man Terminierung annimmt, der Programmkörper für alle erlaubten Eingaben eine Ausgabe berechnet, die der Spezifikation entspricht. Ein Programm ist *korrekt*, wenn es terminiert und partiell korrekt ist. Der untere Teil von Abbildung 2.17 zeigt die zugehörigen Abstraktionen.

Häufig werden wir auch den Terminus „erfüllbar“ verwenden: Ein Programmkörper erfüllt eine Spezifikation, wenn beide ein korrektes Programm bilden. Eine Spezifikation nennen wir erfüllbar, wenn es einen erfüllenden Programmkörper gibt. Diese Existenz ist konstruktiv zu interpretieren!

Noch eine Bemerkung dazu, daß wir für Programmkörper partielle Funktionen verwenden. Genauer gesagt, gibt es zwei Gründe, die diesen Typ interessant machen:

1. Es können Funktionen mittels partiell definierter Terme gebildet werden.

2. Der Y-Kombinator kann benutzt werden, um Rekursionen zu definieren.

Der erste Grund zwingt noch nicht zu partiellen Funktionen, da man auch eine Umschreibung mittels auf einem Teiltyp definierter totaler Funktionen verwenden könnte:

$$\texttt{BODYS}'(\mathit{spec}) \quad \equiv \quad \{x{:}\,\texttt{D}(\mathit{spec})^T \mid \texttt{I}(\mathit{spec})(x)\} \rightarrow \texttt{R}(\mathit{spec})^T$$

Der zweite Grund ist nicht zu unterschätzen, da man auf diese Weise zum einen realistischere, desweiteren effizientere und zum dritten einfachere Programme schreiben kann als ohne Y-Kombinator.

2.6.2 Programmschemata

Zur schematischen Generierung verwenden wir Programme, die variable Unterprogramme enthalten, und schreiben ein solches Schema als:

$$\begin{array}{l}
\texttt{SCHEME} \\
\quad b_1\ \texttt{satisfies}\ S_1 \\
\quad \ldots \\
\quad b_n\ \texttt{satisfies}\ S_n \\
\Rightarrow \\
\quad b_{\{b_1,\ldots,b_n\}}\ \texttt{satisfies}\ S \\
\texttt{END}
\end{array}$$

Dabei sind $b_1, ..., b_n$ Variablen, die für Programmkörper stehen, die die Spezifikationen $S_1, ..., S_n$ (respektive) erfüllen. Die Aussage dieses Schemas ist, daß b, ein Term, in dem die Variablen $b_1, ..., b_n$ durch Programmkörper ersetzt sind, die Spezifikation S erfüllt, sofern der für b_i eingesetzte Körper die Spezifikation S_i erfüllt.

Zu Taktiken, die das Schließen mittels Programmschemata unterstützen, siehe Anhang B.

2.6.3 Spezialisierung

Die in der Globalsuche wichtige Spezialisierungsaussage, die besagt, daß ein Programm für ein mengenwertiges Problem $A*$ durch Filterung auch ein Problem $B*$ lösen kann, das bei gleicher (homomorpher) Eingabe eine Teilmenge der Ausgabe von $A*$ ergibt, wird in mehreren Schritten formalisiert. Dabei kommt es vor allem auf den Zusammenhang zwischen A und B (oder $A*$ und $B*$) an, der Voraussetzung für die Spezialisierung ist.

Wir nennen B eine Spezialisierung von A (oder A eine Generalisierung von B), in Zeichen $A \ll B$, wenn wir ein Programm, das $A*$ löst, zur Berechnung von $B*$ benutzen können, indem wir alle Ergebnismengen des Programmes für $A*$ durch einen Filter schicken, der aus der jeweiligen Ergebnismenge alle nicht zum Ergebnis von $B*$ gehörigen Elemente herausstreicht. Die einfachste Bedingung für diesen Filter besteht darin, für jedes Element abzuprüfen, ob die Ausgabebedingung für B erfüllt ist, die ja entscheidet, ob das Element zur Ergebnismenge von $B*$ gehört oder nicht, und falls dies nicht der Fall ist, dieses Element zu streichen. Diese Mindestanforderung benutzen wir zur Definition der Spezialisierung. Folgende Funktion (die punktuelle Spezialisierungsbedingung) ist definiert für die beiden Probleme A und B und hat als ihre beiden Eingaben x bzw. x', wobei x die Eingabe bezüglich $B*$ und x' die analoge Eingabe bezüglich $A*$ ist:

$$\triangleright\ \texttt{Specl\_cond}(B;A) \quad \equiv \quad \lambda\langle x,x'\rangle.\ \texttt{I}(A)(x') \wedge (\forall y{:}\,\texttt{R}(B)^T.\ \texttt{O}\langle x,y\rangle \quad \Rightarrow \quad \texttt{O}(A) = \langle x',y\rangle)$$

Diese Funktion ergibt *wahr*, wenn zur Lösung von $B*$ für die Eingabe x die Lösung von $A*$ für die Eingabe x' benutzt werden kann.

Im weiteren gehen wir davon aus, daß es einen Algorithmus θ gibt, der für zwei bestimmte Probleme die Abbildung von $x \mapsto x'$ berechnet. Damit ist es auch möglich, eine globale Spezialisierungsbedingung aufzustellen, die für alle Eingaben eine Aussage macht:

$$\begin{aligned}\triangleright\ B \ll A\ \texttt{with}\ \theta \quad \equiv \quad & \texttt{R}(B) = \texttt{R}(A) \wedge \\ & (\forall x{:}\,\texttt{D}(B)^T.\ \texttt{I}(B)(x) \quad \Rightarrow \quad \theta\ \texttt{halts on}\ x \wedge \texttt{Specl\_cond}(B;A)\langle x, \theta x\rangle\end{aligned}$$

Wir notieren kurz, sofern es nur auf die Existenz von θ ankommt:

$$\triangleright\ B \ll A \quad \equiv \quad \exists\theta{:}\,\texttt{D}(B)^T \not\to \texttt{D}(A)^T.\ B \ll A\ \texttt{with}\ \theta$$

Wir können die Spezialisierung auch so auffassen, daß sie das Problem ist, die Funktion θ für gegebene Spezifikationen A und B zu finden:

$$\texttt{FUNCTION SpeclP}(x{:}\,\texttt{D}(B)^T){:}\,\texttt{D}(A)^T\ \texttt{WHERE I}(B)(x)\ \texttt{RETURNS}\ (x' \mid \texttt{Specl\_cond}(B;A)\langle x,x'\rangle)$$

Die Aussage, daß `SpeclP` für bestimmte A, B erfüllbar ist, ist äquivalent zu $B \ll A$.

2.6.4 Typisierung von Meta-APM-Operatoren

Als recht schwierig zu typisieren hat sich die Verwendung des abhängigen Produktraums erwiesen. Diese wird vor allem bei Spezifikationen benutzt. Insbesondere die Wf-Lemmata für die Zugriffsoperatoren `D, R, I, O` sind recht problematisch. Dies konnte etwas entschärft werden, indem Taktiken entworfen wurden, die das Niveau der Inferenzregeln etwas anheben. In Anhang C sind diese Taktiken beschrieben.

Eine ähnliche Problematik findet man bei den Zugriffsoperatoren für Globalsuchräume. In Abschnitt 4.1.1, Seite 113, wird etwas ausführlicher darauf eingegangen.

2.7 Offene Probleme

Zum Schluß noch ein Überblick über die wichtigsten offenen Fragestellungen im Bereich der Typprüfung.

2.7.1 Kontextbedingungen

Wie bereits gesagt, müssen in einem Typisierungsbeweis ggf. Bedingungen über den Wert von Ausdrücken gezeigt werden. Dazu benötigt man allerdings das Wissen, in welchem Zusammenhang dieser Ausdruck auftaucht, um zu zeigen, daß in diesem Kontext die Bedingung erfüllt ist.

In Abbildung 2.18 wurde ein Typisierungsbeweis angefangen, der nicht beendet werden konnte, da versäumt wurde, in den Beweis Kontextbedingungen einzuflechten. Für den if- und den rest-Operator wird dabei je ein Wf-Lemma vorausgesetzt, das in der Bibliothek steht und bereits bewiesen wurde, und nun wie angegeben instanziiert wird. Der rest-Term benötigt als Voraussetzung für seine Typisierbarkeit, daß die Argument-Liste nicht leer ist. Diese Bedingung wird durch den Kontext sichergestellt, was im angegebenen Beweis aber nicht ausgenutzt wird.

Wir sprechen von einer Kontextbedingung für einen Teilterm t eines auswertbaren Terms s, falls diese Bedingung B_t beim Auswerten via *lazy evaluation* stets eingehalten wird, sobald t in die Berechnung eingeht (analysiert wird). Es ist entscheidend, daß wir uns hier auf das Auswertungsverfahren *lazy evaluation* festlegen, da dies die Strategie ist, die die Regeln der Typentheorie von sich aus annehmen ist; die so erhaltenen Kontexte sind die schwächsten, auf die man sich verlassen kann.

Dabei stellt sich nun die Frage, wie man systematisch solche Kontextbedingungen aufstellt und wie man den Beweis dieser Bedingungen mit dem Typisierungsbeweis verzahnt. Wir präsentieren im folgenden eine Idee, die allerdings nicht zu Ende gedacht ist.

Wir nehmen an, der fragliche Teilterm t befinde sich im i-ten Teilterm von $op(V_1.s_1;...;V_i.s_i;...;V_n.s_n)$, d.h. t ist in s_i enthalten. Zu op und i gibt es einen Kontextkonstruktor $K_{op,i}$, der für eine beliebige Bedingung B, die an einen Teil von s_i gestellt wird, einen Kontext erzeugt. Dieser Kontextkonstruktor hat die folgende Gestalt: $K_{op,i}(V_1.v_1[V_1];...;V_i.B[V_i];...;V_n.v_n[V_n])$ und gibt die Gesamtbedingung zurück, die die Gültigkeit von B in diesem Kontext ausdrückt. Der Konstruktor ist nicht nur von der Bedingung B abhängig, für die ein Kontext erstellt werden soll, sondern auch von übrigen Teiltermen von op; die Variablen $v_k[V_k]$ stehen hier formal für die Teilterme s_k.

Wenden wir dies auf unser Beispiel an. Für den if-Operator finden wir folgende Kontextkonstruktoren (dabei sei `ifnull`$(S;x;y)$ die Termdarstellung für die Display-Form if $S=[]$ then x else y):

- $K_{\text{ifnull},1}(B_S;x;y) \equiv B_S$
- $K_{\text{ifnull},2}(S;B_x;y) \equiv S=[] \Rightarrow B_x$
- $K_{\text{ifnull},3}(S;x;B_y) \equiv S\neq[] \Rightarrow B_y$

Falls also ein Kontext für den ersten Teilterm erzeugt werden soll, wird die Bedingung einfach durchgereicht. Beim zweiten Teilterm ist klar, daß dieser nur ausgewertet wird, wenn die Liste S leer

$\vdash$ if $L = []$ then $[]$ else $\texttt{rest}(L) \in T$ list
BY Wf-Lemma $\forall X, Y \in \mathbf{U}_j.\ \forall S \in X$ list. $\forall s, t \in Y.$
(if $S = []$ then s else t) $\in Y$
mit Instanziierung $[T,\ T\,\text{list},\ L,\ [],\ \texttt{rest}(L)/X, Y, S, s, t]_1$

$\vdash$ $T \in \mathbf{U}_j$
...

$\vdash$ $T\,\text{list} \in \mathbf{U}_j$
...

$\vdash$ $L \in T\,\text{list}$
...

$\vdash$ $[] \in T\,\text{list}$
...

$\vdash$ $\texttt{rest}(L) \in T\,\text{list}$
BY Wf-Lemma $\forall X \in \mathbf{U}_j.$
$\forall S \in X$ list.
$S \neq [] \Rightarrow \texttt{rest}(S) \in X$ list
mit Instanziierung $[T, L/X, S]_1$

$\vdash$ $T \in \mathbf{U}_j$
...

$\vdash$ $L \in T\,\text{list}$
...

$\vdash$ $L \neq []$
Problematisches Ziel

Abb. 2.18. *Dieser Typisierungsbeweis kann nicht beendet werden, da Kontextwissen fehlt*

ist; also wird $S = []$ zum Kontext addiert. Beim dritten Teiilterm wird entsprechend $S \neq []$ zum Kontext hinzugefügt.

Um nun die Kontextbedingung für if $L = []$ then $[]$ else $\texttt{rest}(L)$, unserem Beispiel, zu erzeugen, gehen wir wie folgt vor: Wir stellen fest, daß die Bedingung an das zweite L gestellt werden soll, und zwar $B(L) = L \neq []$. Wir gehen nun von innen nach außen die Operatoren entlang und fügen die Kontexte hinzu, die von diesen Operatoren erzeugt werden. Der innerste Operator ist `rest`, der freilich nichts zum Kontext addiert, also wie $K_{\text{ifnull},1}$ nur die Bedingung durchreicht. Der dann äußere Operator ist `ifnull`, der einen Kontext mittels $K_{\text{ifnull},3}$ hinzufügt. Wir erhalten im Ergebnis:

$$
\begin{aligned}
&K_{\text{ifnull},3}(L; []; K_{\text{rest},1}(L \neq [])) = \\
&K_{\text{ifnull},3}(L; []; L \neq []) = \\
&L \neq [] \Rightarrow L \neq []
\end{aligned}
$$

Diese Bedingung kann nun in der Tat bewiesen werden.

In Abbildung 2.19 sehen wir nun den Beweis, der Kontexte berücksichtigt. Die Gesamtbedingung wird gleich am Anfang mittels eines Schnittes eingeführt. Die Typisierungstaktik kann auf diese Weise die Bedingung komplett nach außen weitergeben sowie im weiteren Verlauf des Beweises die Bedingung

$\vdash$ if $L = []$ then $[]$ else $\texttt{rest}(L) \in T$ list
BY cut '$K_{\text{ifnull},3}(L; []; K_{\text{rest},1}(L \neq []))$'

$\vdash \quad K_{\text{ifnull},3}(L; []; K_{\text{rest},1}(L \neq []))$
...

$K_{\text{ifnull},3}(L; []; K_{\text{rest},1}(L \neq []))$
$\vdash$ if $L = []$ then $[]$ else $\texttt{rest}(L) \in T$ list
BY Taktik, die Kontexte transformiert

$\vdash \quad T \in \mathbf{U}_j$
...

$\vdash \quad T\,\text{list} \in \mathbf{U}_j$
...

$\vdash \quad L \in T\,\text{list}$
...

$\vdash \quad [] \in T\,\text{list}$
...

$K_{\text{rest},1}(L \neq []) \quad \vdash \quad \texttt{rest}(L) \in T\,\text{list}$
BY Taktik, die Kontexte transformiert

$\vdash \quad T \in \mathbf{U}_j$
...

$\vdash \quad L \in T\,\text{list}$
...

$L \neq [] \quad \vdash \quad L \neq []$
Nun beweisbar!

Abb. 2.19. *In diesen Typisierungsbeweis ist ein Kontextnachweis eingebaut*

simultan mit dem zu typisierenden Term zerlegen. Zur Zerlegung können wir nun leider nicht mehr einfach Lemmata benutzen, da man in Lemmata keine Kontextanalyse ausdrücken kann; es muß eine Taktik benutzt werden, deren Arbeitsweise noch unbekannt ist.

Dies ist nun ein bewußt einfach gewähltes Beispiel. In allgemeinen ist nicht davon auszugehen, daß Kontexte durch Schemata erzeugt werden können, da, sofern Induktoren vorkommen, auch spezifische Invarianten eine Rolle spielen können. Diese Invarianten müssen vom Benutzer angegeben werden.

Desweiteren gingen wir davon aus, daß alle Terme in der Kontextbedingung selbst problemlos typisierbar sind. Allgemein muß jedoch damit gerechnet werden, daß die Typisierung dieser Bedingungen wieder Kontexte notwendig macht. Es ist darauf zu achten, daß Zirkel vermieden werden.

Es ist kein Zufall, daß wir in dem Beispiel abstrakte Operatoren verwendet haben. Die Typisierungsregeln für die Primitiven erzeugen von sich aus Kontexte und lassen das Problem gar nicht erst virulent werden. Da wir für abstrakte Operatoren jedoch Wf-Lemmata benutzen und diese keine Kontexte erzeugen, besteht das Problem, eine Taktik zu entwerfen, die analog den Wf-Lemmas typisiert und dabei zugleich Kontexte generiert.

2.7.2 Erweiterung des typprüfbaren Sprachbereichs

Die Restriktion für APM schließt im Augenblick einige Typen aus. Wir wollen hier kurz diskutieren, welche dieser Typen noch erlaubt werden könnten:

- Teiltyp $\{x{:}S \mid T\}$: Der Teiltyp enthält alle Elemente von S, für die $T[x]$ nicht leer ist. Die einfachste Möglichkeit, Teiltypen zu berücksichtigen, besteht darin, in die Gleichungstheorie $\{x{:}S \mid T\} \stackrel{*}{=} S$ aufzunehmen und die durch T ausgedrückte Bedingung analog den bedingten Typisierungsansätzen zu behandeln.

- Induktiver Typ rectype $X = T_X$: Auch kann man durch eine Erweiterung der Gleichungstheorie diesen Typ berücksichtigen. Es ist rectype $X = T_X \stackrel{*}{=} T_{\mathrm{rectype}\, X=T_X}$ die neue Typgleichung.

- Quotiententyp $x, y; T//E$: Dieser Typ sollte am Besten nicht direkt erlaubt werden, da die Intention ja ist, T und den Quotienten voneinander zu trennen. Eine Typgleichung würde dieser Trennung nicht angemessen sein. Stattdessen sollte man Werkzeuge anbieten, die die Konstruktion konkreter Quotienten unterstützen, die dann als eigene Typkonstruktoren neu eingeführt werden (analog den endlichen Mengen).

- Höhere Universen $\mathbf{U}_i$: Im Prinzip spricht nichts dagegen, auch Universen zu erlauben, solange man keine abhängigen Typen zuläßt. Der Universenlevel kann in diesem Fall automatisch bestimmt werden.

- Abhängige Räume: Diese Typen erlauben grundsätzlich die Vermengung von Typausdrücken und Elementausdrücken, was unbedingt vermieden werden muß. Es ist möglich, bestimmte Varianten abhängiger Räume zuzulassen, sofern eben sichergestellt ist, daß diese Vermischung nicht stattfindet.

Grundsätzlich ist bei jeder dieser Erweiterungen festzustellen, ob es noch Algorithmen gibt, die die (eine) prinzipielle Lösung von Typgleichungen oder TGS berechnen können.

Kapitel 3

Inferenzen

3.1 Termersetzung: Einführung

Bei der Termersetzung geht es darum, die Ersetzung eines Teilterms, die in der Formel der Konklusion oder einer Hypothese steht, automatisch durch einen typentheoretischen Beweis zu rechtfertigen. Wir werden uns zunächst einen Überblick verschaffen, wie solche Beweise konstruiert werden können und uns erst danach einer Taktik zuwenden, die in der Lage ist, eine solche Konstruktion durchzuführen. Das „rewrite package" des derzeitigen NuPRL (Version 4.x) enthält eine Implementierung einer solchen Taktik. Es wird sich herausstellen, daß diese Implementierung für unsere Zwecke zu schwach ist, da sie insbesondere keinen Zugang zu Typinformation hat. Wir haben daher einen Aufsatz programmiert, die die vorhandene Implementierung so erweitert, daß sie Typisierungsalgorithmen aufrufen und diese Information nutzen können.

Eine typische Aufgabe, die mit Termersetzung gelöst werden kann, ist: Beweise $x + y + 1 \geq x + 6$ unter der Annahme, daß $y \geq 5$ gilt. Der Ansatz besteht darin, das y in der zu beweisenden Formel durch 5 zu ersetzen, also die Annahme $y \geq 5$ als Ersetzung $y \mapsto 5$ zu verstehen.

Wir betrachten im folgenden Formeln immer als Formelbäume (Abb. 3.1). Jeder Teilterm der Formel wird nun durch einen Teilbaum repräsentiert, und wir können dadurch jedem Teilterm eine *Adresse* zuordnen. Eine Adresse ist eine Liste $[a_1, ..., a_n]$, die beschreibt, wie man von der Wurzel zu dem gemeinten Teilterm kommt, indem man zunächst in den a_1-ten Teilbaum der Wurzel, dann in dessen a_2-ten Teilbaum, usw absteigt, bis man den Teilbaum erreicht, der den Teilterm repräsentiert. In unserem Beispiel hat der Teilterm y die Adresse $[1, 2, 1]$.

Um eine Termersetzung zu rechtfertigen, werden wir eine Reihe von (binären) Relationen zu betrachten haben. Als Basisrelation $\trianglelefteq_B$ bezeichnen wir den Zusammenhang von zu ersetzendem Teilterm t zum ersetzten Term t'. In unserem Beispiel ist $\trianglelefteq_B = \geq$, da wir davon ausgehen, daß $y \geq 5$ gilt. Allgemein können wir immer voraussetzen, daß $t \trianglelefteq_B t'$ gilt.

Die Klauselrelation $\trianglelefteq_K$ ist diejenige Relation, die den Rahmen der Termersetzung definiert. Wir haben generell zu unterscheiden, ob wir vorwärts oder rückwärts schließen; es kommt darauf an, in welcher Klausel die Termersetzung stattfindet. Handelt es sich um eine Hypothese s, in der der zu ersetzende Term t vorkommt, so muß letztlich $s \Rightarrow s'$ gelten, wobei s' die Hypothese mit ersetzem Term ist. Falls s jedoch eine Konklusion ist, müssen wir uns auf $s \Leftarrow s'$ abstützen. Die Klauselrelation ist daher für Termersetzung in einer Hypothese fest $\trianglelefteq_K = \Rightarrow$ und für Ersetzung in einer Konklusion fest $\trianglelefteq_K = \Leftarrow$.

Mit Hilfe der Basis- und der Klauselrelation können wir nun die eigentliche Aufgabe für die Rechtfertigung formulieren: Beweise die Formel

$$t \trianglelefteq_B t' \;\Rightarrow\; s \trianglelefteq_K s'$$

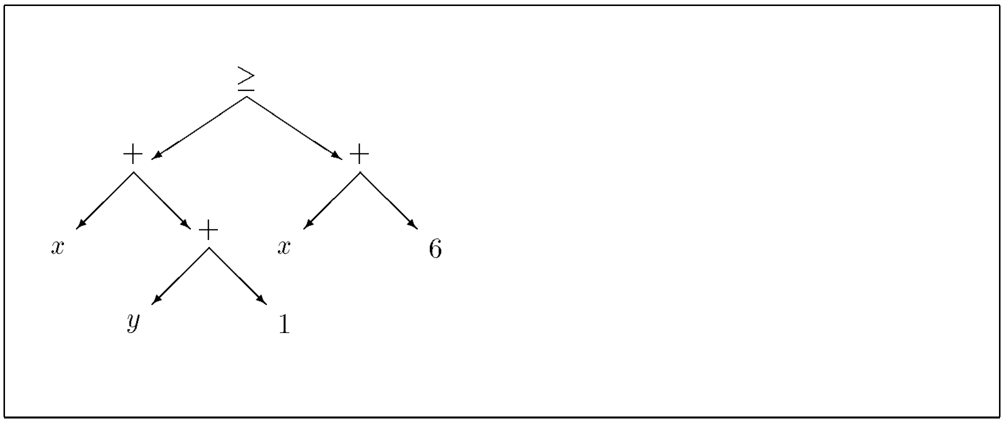


Abb. 3.1. *Formelbaum*

In unserem Beispiel ist dies

$$y \geq 5 \;\Rightarrow\; (x + y + 1 \geq x + 6 \Leftarrow x + 5 + 1 \geq x + 6)$$

da die zu ersetzende Formel in einer Konklusion vorkommt.

Die Relationen und die Operatoren, die in dieser Aufgabe vorkommen, sind bekannt; es ist daher sinnvoll, ihre Eigenschaften in eine Wissensbank einzutragen, und bei der Konstruktion des Rechtfertigungsbeweises auf diese Eigenschaften zurückzugreifen.

Was die Relationen angeht, so sind vor allem zwei Sätze von Eigenschaften sinnvoll:

- *Transitive unsymmetrische Relationen:* Diese Relationen sind vor allem transitiv; ggf. sind sie zusätzlich reflexiv. Hierunter fallen insbesondere Ordnungsrelationen wie $\geq$.

- *Äquivalenzrelationen:* Diese Relationen sind transitiv, reflexiv und symmetrisch.

Aus Termersetzungssicht ist der primäre Unterschied, daß sich aufgrund der Symmetrie Äquivalenzausdrücke invertieren lassen.

Häufig wollen wir einen Ausdruck $x \trianglelefteq y$ auf die Form $x \trianglelefteq' y$ bringen, da die weitere Konstruktion eben die Relation $\trianglelefteq'$ verlangt. Dazu tragen wir in die Wissensbank ebenfalls ein, wie sich die Relationen untereinander verhalten, und zwar als *Abschächbarkeit.* Die Relation $\trianglelefteq$ wird auf die Relation $\trianglelefteq'$ abgeschwächt, falls für alle x und y gilt: $x \trianglelefteq y \;\Rightarrow\; x \trianglelefteq' y$.

Beispielsweise können wir $>$ auf $\geq$ abschwächen, oder $\Longleftrightarrow$ auf $\Leftarrow$.

Was die Operatoren angeht, so benötigen wir insbesondere Eigenschaften, die es erlauben, von der Rechtfertigung für die Ersetzung in einem (direkten) Teilterm auf die Ersetzung im Gesamtterm zu schließen. Solche Eigenschaften werden als *Funktionalität* bezeichnet und haben stets die Form

$$x_i \trianglelefteq_i y_i \;\Rightarrow\; op(x_1, ..., x_i, ..., x_n) \trianglelefteq op(x_1, ..., x_{i-1}, y_i, x_{i+1}, ..., x_n)$$

Vertrauter ist uns dies als Monotonie, falls Ordnungsrelationen benutzt werden, oder Kongruenz, falls Äquivalenzrelationen benutzt werden.

Für den Operator $+$ haben wir in unserem Beispiel die Monotonien:

- $a \geq a' \Rightarrow a + b \geq a' + b$
- $b \geq b' \Rightarrow a + b \geq a + b'$

Wir können $\geq$ statt als Relation auch als Operator auffassen und etwa für die inverse Implikation folgende Monotonien aufstellen:

- $a \geq a' \;\Rightarrow\; (a \geq b \Leftarrow a' \geq b)$

$\vdash x+y+1 \geq x+6 \;\Leftarrow\; x+5+1 \geq x+6$
BY Monotonie für $\geq$

$\quad \vdash x+y+1 \geq x+5+1$
$\quad$ BY Monotonie für $+$

$\quad\quad \vdash y+1 \geq 5+1$
$\quad\quad$ BY Monotonie für $+$

$\quad\quad\quad \vdash y \geq 5$
$\quad\quad\quad$ BY Annahme

$\vdash x+y+1 \geq x+6 \;\Leftarrow\; x+5+1 \geq x+6$
BY Schnitt

$\quad \vdash y \geq 5$
$\quad$ BY Annahme

$\quad y \geq 5 \;\vdash\; -\text{"}-$
$\quad$ BY Monotonie für $+$

$\quad\quad y+1 \geq 5+1 \;\vdash\; -\text{"}-$
$\quad\quad$ BY Monotonie für $+$

$\quad\quad\quad x+y+1 \geq x+5+1 \;\vdash\; -\text{"}-$
$\quad\quad\quad$ BY Monotonie für $\geq$

$\quad\quad\quad\quad x+y+1 \geq x+6 \;\Leftarrow\; x+5+1 \geq x+6$
$\quad\quad\quad\quad \vdash x+y+1 \geq x+6 \;\Leftarrow\; x+5+1 \geq x+6$
$\quad\quad\quad\quad$ BY Hypothese

Abb. 3.2. *links: Top-Down-Beweis; Rechts: Bottom-Up-Beweis*

- $b' \geq b \;\Rightarrow\; (a \geq b \Leftarrow a \geq b')$

Für den generellen Ablauf der Termersetzungs-Rechtfertigung gibt es zwei Möglichkeiten:

- *Top-Down:* Wir sorgen dafür, daß die Aufgabe $\vdash s \trianglelefteq_K s'$ in der Konklusion steht, und steigen nun analytisch in die Teilterme von s und s' simultan ab, wobei wir die Eigenschaften der Relationen und Operatoren benutzen. Zuletzt erhalten wir ein Unterziel $\vdash t \trianglelefteq_B t'$, welches nach Annahme gilt.
- *Bottom-Up:* Wir sorgen dafür, daß unsere Annahme $t \trianglelefteq_B t'$ in einer Hypothese steht, und konstruieren nun synthetisch neue relationale Formeln, und zwar der Reihe nach für die äußeren Terme, die t bzw. t' enthalten, bis wir beim äußersten Term s bzw. s' angekommen sind. Es sollte dann in der (einer) Hypothese die Formel $s \trianglelefteq_K s'$ stehen, aus der die Aufgabe $\vdash s \trianglelefteq_K s'$ trivialerweise folgt.

Beide Methoden sind gleich gut geeignet, um reine Termersetzung durchzuführen. Die Top-Down-Ersetzung paßt jedoch besser zur ohnehin analytischen Typentheorie, und ist daher etwas einfacher zu implementieren. Die Bottom-Up-Methode hat den Vorteil, daß man hier den Beweis für die Termersetzung mit anderen Beweisen für die jeweiligen Teilterme verzahnen kann, da in einer anderen Hypothese als derjenigen, die für die Durchführung der Ersetzung benutzt wird, simultan eine weitere Eigenschaft von $s \trianglelefteq_K s'$ bewiesen werden kann. Wir weisen darauf hin, weil dies eventuell zum Beweis von Kontextbedingungen nützlich sein könnte.

In Abbildung 3.2 sehen wir links den Top-Down-Beweis für unser Beispiel. Durch dreimalige Anwendung einer Monotonieeigenschaft erhält man schließlich eine Konklusion, die $t \trianglelefteq_B t'$ entspricht. Die Monotonien werden dabei, da in jedem Schritt die Voraussetzung der letzten Behauptung gesucht wird, rückwärts, also von rechts nach links angewendet.

Auf der rechten Seite sehen wir den etwas umfänglicheren Bottom-Up-Beweis. Zunächst muß mit Hilfe eines Schnittes die Formel $t \trianglelefteq_B t'$ in eine Hypothese gebracht werden. Auf diese Hypothese wird nun dreimalig eine Monotonieeigenschaft vorwärts, von links nach rechts angewendet. Die Konklusion bleibt dabei die ganze Zeit über identisch. Schließlich wurde in der Hypothese die Behauptung zusammengesetzt, so daß eine triviale Sequenz entsteht.

Das Beispiel funktioniert nun alleine mit Hilfe von Monotonien, oder allgemeiner, von Funktionalitäten. Häufig ist es jedoch notwendig, weitere Eigenschaften zu benutzen. Beispielsweise findet man in der Wissensbank die Monotonien zu $+$ nicht für die $\geq$-Relation, sondern nur für die $\leq$-Relation. Ein Zwischenschritt, der die beiden Seiten der Relation umdreht ($a \geq b \iff b \leq a$), schafft Abhilfe. Im Falle von Äquivalenrelationen kann zudem das Symmetriegesetz zum Invertieren benutzt werden.

Eine Abschwächung wäre z.B. notwendig, wenn man die etwas stärkere Voraussetzung $y > 5$ gemacht hätte. In diesem Fall hätte man (beim Top-Down-Verfahren zuletzt, beim Bottom-Up-Prozedere zuerst) die Eigenschaft $y > 5 \Rightarrow y \geq 5$ benutzen müssen.

Die Transitivitätsgesetze werden gebraucht, wenn man mehrere Termersetzungen nacheinander durchführen möchte. Angenommen, ein Teilterm von s ist ersetzt worden und man erhielt s', d.h. man hat $s \trianglelefteq_K s'$, und in s' ist ein weiterer Teilterm ersetzt worden, so daß s'' herauskommt. Letzteres wird in analoger Weise durch $s' \trianglelefteq_K s''$ gerechtfertigt. Um nun die Gesamt-Termersetzung $s \trianglelefteq_K s''$ beweisen zu können, benötigen wir ein Transitivitätsgesetz:

$$x \trianglelefteq_K y \wedge y \trianglelefteq_K z \Rightarrow x \trianglelefteq_K z$$

Die Hintereinanderschaltung auf Klauselebene ist eigentlich uninteressant, da man ja einfach auch zwei Einzelbeweise verkoppeln könnte. Nun funktioniert dies aber ganz genauso für jeden Teilterm, an dem man während des Ersetzungsprozesses vorbeikommt. Der Vorteil ist dann, daß der Abstieg bis zu diesem Teilterm für beide Ersetzungen nur einmal durchgeführt werden muß, und man erst ab dieser Stelle im Formelbaum, bei der die beiden Ersetzungen in jeweils unterschiedliche Teilbäume hineinlaufen, mittels des Transitivitätsgesetzes den Beweis für jede Ersetzung separieren muß. Durch die Anwendung der Transitivität werden also optimalere Beweise konstruiert. Wir werden uns weiter unten mit einer Methode beschäftigen, die automatisch die günstigste Verwendung von Transitivitäten berechnet.

Zunächst aber zu der einfachen Taktik für die Termersetzung, wie sie im „rewrite package" enthalten ist. Im Anschluß an deren Darstellung werden wir uns eine Möglichkeit überlegen, wie die Schwäche dieser Taktik, keine lokalen Umgebungen erzeugen zu können, mit Hilfe von Typinferenzen ausbügeln können. Diese verbesserte Termersetungstaktik werden wir dann bezüglich der Transitivitäten optimieren.

3.2 Termersetzung: Taktik

Wir stellen zunächst das Interface der Taktik vor, um erst anschließend auf die Realisierung einzugehen.

3.2.1 Das Interface für die einfache Termersetzung

Ausgangspunkt jeder Termersetzung ist die taktische Regel `Rewrite`, kurz `RW`, die aus einer Angabe, welche Termersetzung wie durchgeführt und auf welche Klausel angewandt werden soll, den Termersetzungsbeweis konstruiert. Diese Regel hat die Argumente:

`RW` C i

Dabei ist C eine *Konversion* und i die Nummer der Klausel (d.h. 0 für die Konklusion und $i > 0$ für die Hypothese i). Eine Konversion beschreibt zugleich, welche Termersetzung durchgeführt und auf welche Weise der Beweis konstruiert werden soll. Wir unterscheiden im folgenden *atomare* Konversionen von *Konversionalen*; letztere haben Konversionen als Argumente, erstere nicht.

Paradigmatisch für eine atomare Konversion ist `HypC`, die eine universelle Formel, die in einer anzugebenden Hypothese steht, als Termersetzungsregel auffaßt. Hat die Formel die Struktur

$$\forall x_1, ..., x_n.\ A \ \Rightarrow\ t \trianglelefteq t'$$

so wird sie von `HypC` als Anleitung zur Termersetzung von t durch t' gelesen, wobei A Voraussetzungen enthält und die x_i instanziierbare Variablen darstellen. (Umgekehrt liest das Pendant `Rev_HypC` diese Formel als Ersetzung von t' durch t.) Wir bemerken, daß hier wieder das Instanziierungsproblem auftritt, wie es in Abschnitt 2.4.4, Seite 76 beschrieben wurde.

Auf welchen Teilterm bezieht sich jetzt die durch eine Konversion wie etwa `HypC` beschriebene Termsersetzung? Das Konzept des *aktuellen Teilterms* soll uns dabei helfen – man beachte, daß der aktuelle Teilterm nicht direkt in der Implementierung der Konversionen repräsentiert wird und nur dem besseren Verständnis dient.

Die taktische Regel `RW` bezieht die angegebene Konversion C stets auf die ganze Klausel, d.h. der aktuelle Teilterm *ist* die Klausel. Um innere Terme zu ersetzen, muß man mit Hilfe von Konversionalen den Fokus der Termersetzung verschieben.

Das allgemeinste Konversional zur Fokussierung innerer Terme ist `SubIfC`. Es hat folgende Argumente:

`SubIfC` P C

Sei s der aktuelle Teilterm, auf den `SubIfC` angewendet wird. Dieses Konversional geht nun alle Teilterme von s durch und wendet auf alle diese Teilterme, für die die Bedingung P wahr ist, die Konversion C an. Wollen wir etwa den vierten Teilterm einer Klausel mittels `HypC` ersetzen, könnten wir ein Konstrukt der Art

`RW (SubIfC` „vierter Teilterm“ `(HypC` j`))` i

benutzen. Es ist nicht schwer zu sehen, daß man, wenn man Adressen zur Angabe des Ortes der Termersetzung verwenden will, aus mehreren `SubIfC`-Konversionen eine adressierende Konversion zusammensetzen kann. `AddrC` macht dies automatisch:

`AddrC` a C

Dies wendet die Konversion C auf den Teilterm mit der Adresse a relativ zum aktuellen Teilterm an.

Um auf unsere Beispiel von der Einführung zurückzukommen: Die Ersetzung von y durch 5 wird durch folgende Konversion durchgeführt:

$y \geq 5 \quad \vdash \quad x + y + 1 \geq x + 6$
BY RW (AddrC [1;2;1] (HypC 1)) 0
Ersetze bei Adresse [1;2;1] in Klausel 0 entsprechend Hypothese 1
$y \geq 5 \quad \vdash \quad x + 5 + 1 \geq x + 6$

Die Taktik `RW` konstruiert dabei einen Beweis, der Abbildung 3.2 links entspricht. Jede beteiligte Konversion produziert dabei einen Teil dieses Beweises: Die drei `SubIfC`-Aufrufe, die sich ergeben, wenn `AddrC` eine dreistelligen Adresse übergeben wird, erzeugen die drei Rückgriffe auf Monotoniegesetze; `HypC` konstruiert den Bezug auf die Annahme. – Die Taktik `RW` selbst erzeugt einen Beweisteil, der bislang noch nicht abgebildet wurde, nämlich den Übergang von der Situation bei ihrem Aufruf zu dem Beweis nach Abbildung 3.2. Dieser ist jedoch relativ einfach (s.u.).

Die wichtigsten weiteren Konversionale sind in Abbildung 3.3 zusammengefaßt.

- `IdC`: Identische Konversion (keine Ersetzung)
- `FailC`: Konversion, die immer fehlschlägt.
- C_1 `ANDTHENC` C_2: Wende auf den aktuellen Teilterm zuerst C_1 und dann C_2 an.
- C_1 `ORELSEC` C_2: Versuche, auf den aktuellen Teilterm C_1 anzuwenden. Scheitert dies, wende C_2 an.
- C_1 `ORTHENC` C_2: Probiere, auf den aktuellen Teiltem C_1 anzuwenden. Gelingt dies, wende anschließend C_2 an. Ergab sich ein Fehlschlag, wende nur C_2 an.
- `TryC` C: Versuche, auf den aktuellen Teilterm C anzuwenden. Scheitert dies, lasse den Teilterm unverändert.

Abb. 3.3. *Konversionale*

3.2.2 Implementierung der einfachen Termersetzung

Wir kommen nun zur Implementierung der Konversionen. Diese stützt sich auf den ML-Datentyp `convn`, der eine Abkürzung für folgenden Typ ist:

```
convn = env → term → (term # reln # just)
```

Dies ist so zu verstehen: Eine Konversion wird stets mit einer Umgebung (vom Typ `env`) und dem aktuellen Teilterm aufgerufen und berechnet ein Tripel, das den Teilterm mit durchgeführter Ersetzung, die Basisrelation (`reln`) und eine Rechtfertigung (`just`) enthält. Die Umgebung deklariert freie Variablen des aktuellen Teilterms. Die Rechtfertigung ist wiederum eine Taktik, die den Beweisteil, für den die Konversion zuständig ist, konstruiert.

Wollen wir nur wissen, welche Termersetzung durch eine Konversion ausgedrückt wird, interessiert uns nur das erste Element des Tripels, der Ersatzterm, die anderen Komponenten könne wir ignorieren. Wollen wir jedoch auch den rechtfertigenden Beweis konstruieren, benötigen wir auch die Basisrelation und die Rechtfertigungstaktik.

Diese Taktik erwartet, daß sie auf eine Sequenz angesetzt wird, die folgende Form hat; wir gehen dabei davon aus, daß die Konversion C entsprechend $C\,e\,t = (t', \unlhd, J)$ rechnet:

$$\Gamma \vdash t \unlhd t'$$

Die Rechtfertigung J hat diese Behauptung zu beweisen. Die Hypothesen Γ müssen dabei die Variablen genauso deklarieren wie in e vorgegeben[1].

Atomare Konversionen haben nun Rechtfertigunstaktiken, die diese Behauptung eigenständig beweisen können. `HypC` benutzt dazu etwa eine Taktik, die die universelle Formel instanziiert (`BackThruHyp`). Konversionale erzeugen dagegen nur Beweisabschnitte, da sie die Behauptung auf Teilbehauptungen reduzieren, für die die als Argument übergebenen Konversionen zuständig sind. So produziert etwa `SubIfC` P C einen Beweisabschnitt der Art

$\Gamma \vdash t \unlhd t'$
BY Anwendung einer Funktionalität
$\Gamma_1 \vdash t_1 \unlhd_1 t'_1$
$\vdots$
$\Gamma_n \vdash t_n \unlhd_n t'_n$

[1] Dabei ergibt sich freilich das Problem, daß in e der selbe Name mehrfach deklariert werden darf, es zählt dann jeweils der „lokalere“ Name; in einer Hypothesenliste darf jedoch nur jede Variable einmal deklariert werden. Die Rechtfertigung muß also damit rechnen, daß in der Hypothesenliste Umbenennungen stattgefunden haben.

Dabei sind t_1 bis t_n diejenigen Teilterme von t, für die die Bedingung P wahr ist. Desweiteren seien e_1 bis e_n die für die jeweiligen Teilterme erweiterten Umgebungen. `SubIfC` ruft nun für jeden Teilterm C auf: $(t'_i, \trianglelefteq_i, J_i) = C\, e_i\, t_i$. t' berechnet sich nun, indem die ausgewählten Teilterme in t durch die Ersatzterme $t'_1, ..., t'_n$ ersetzt werden. Damit weiß `SubIfC`, was durch die Funktionalität geleistet werden muß und kann ein entsprechendes Lemma aus der Wissensbank bestimmen, das zur Anwendung kommen soll. Auf jedes so erhaltene Unterziel $\Gamma_i \vdash t_i \trianglelefteq_i t'_i$ wird nun die von C berechnete Rechtfertigung J_i angesetzt. Damit ist nun klar, wie die von `SubIfC` selbst zurückgegebene Rechtfertigungstaktik auszusehen hat: Wende das Funktionalitätslemma an und dann auf die i-te Untersequenz J_i.

Man vergleiche dieses Schema für `SubIfC` mit den einzelnen Schritten in Abbildung 3.2 links. Wie man sieht, paßt das Schema mit $n = 1$.

3.2.3 Kritik an der einfachen Termersetzung

Wie gesagt, haben wir das bisher dargestellte Schema als bereits in NuPRL implementiert vorgefunden. Ein Kritikpunkt deutete sich bereits an: Dieses Schema unterstützt allein die Top-Down-Methode der Termersetzung. Wir sollten jedoch im Auge behalten, daß die Bottom-Up-Methode ggf. Vorteile bringt, wenn während der Termersetzung simultan Eigenschaften dieser Ersetzung bewiesen werden sollen. Möglicherweise ist dies für die Synthese von Kontextbedingungen wichtig, wie sie für bedingte Termersetzungen benötigt werden.

Ein weiteres Problem ist nicht so offensichtlich: Das Konversional `SubIfC` hat zwar die Möglichkeit, die Umgebung für die Teilterme zu erweitern, d.h. die Bindungsvariablen dieser Teilterme zu deklarieren. Gleichzeitig fehlt es jedoch an einem Mechanismus, der es erlaubt, den Bindungsvariablen Typen zuzuordnen. Dies führt dazu, daß die Typen der Variablen während der Ersetzung in der Regel unbekannt sind. Viele Schlüsse sind jedoch erst möglich, wenn typisierte Variablen vorliegen.

Betrachte etwa das Assoziativitätsgesetz für die Listenkonkatenation:

$$\forall T{:}\,\mathbf{U}.\ \forall L_1, L_2, L_3{:}\,T \text{ list}.\quad (L_1 \cdot L_2) \cdot L_3 = L_1 \cdot (L_2 \cdot L_3)$$

Um dieses Gesetz, das als Lemma in einer Wissensbank gespeichert sein könnte, auf einen konkreten Fall anzuwenden, müssen nicht nur die drei Listen L_1, L_2 und L_3 instanziiert werden, sondern auch der Typ T. Nehmen wir einmal an, wir wollen dieses Gesetz auf folgende Formel anwenden, d.h. die linke Seite der Gleichung entsprechend der Assoziativität umschreiben:

$$\forall x \in \{1..100\}.\ ([x;x] \cdot [x;x]) \cdot [x;x] = [x;x] \cdot [x;x;x;x]$$

Die Instanzen für die drei Listen zu bestimmen, ist kein Problem, man kann sie ablesen. Aber daß für T der Typ $\mathbf{Z}$ einzusetzen ist, ist nicht so offenkundig.

Im bisherigen Schema ist `SubIfC` nicht in der Lage, der Variablen x beim Abstieg in den $\forall$-Teilterm einen Typ zuzuordnen. Es ist daher ebenfalls nicht möglich, den Typ der drei Listen zu bestimmen, da darin ja x vorkommt und kein weiterer Anhaltspunkt existiert. Damit kann aber auch T nicht bestimmt werden.

Wäre der Typ von x jedoch bekannt, so könnten die drei Listen durch $\mathbf{Z}$ list typisiert werden. Die Gleichung T list $= \mathbf{Z}$ list könnte aufgelöst werden, und $T = \mathbf{Z}$ wäre die gesuchte Instanz.

Das hier geschiderte Problem ist dem Instanziierungsproblem (siehe Abschnitt 2.4.4, Seite 76) vorgelagert. Ohne den Typ von x zu wissen, braucht man sich gar nicht erst Gedanken zu machen, wie man das Lemma zu instanziieren hat.

3.2.4 Termersetzung mit Typbestimmung

Da wir für den Sprachbereich APM einen Algorithmus zur Typbestimmung haben, ist es naheliegend, die Termersetzung so zu verbessern, daß dieser Algorithmus benutzt werden kann. Das Interface für die Termersetzung soll dabei weitgehend beibehalten werden, so daß es nun primär um eine verbesserte Implementierung geht.

Kernstück ist wiederum eine Typdefinition, um die alles andere herum organisiert ist:

```
typed_convn = env → (term # term # gettype) → (term # gettype # reln # just)
```

Der Konversion wird nun ein Typbestimmungs-Algorithmus g vom Typ `gettype` übergeben: $C\ e\ (t, T, g) = (t', g', \trianglelefteq, J)$. Dieser Algorithmus ist fähig, die Typen aller (oder ausgewählter) Teilterme oder Bindungsvariablen von t zu bestimmen. Wir sind dabei nicht auf einen bestimmten Algorithmus festgelegt, sondern können einen geeigneten einsetzen. Wir setzen desweiteren voraus, daß T der Typ von t ist.

Das Ergebnis der Konversion besteht nun aus einem Quadrupel, das als neue Komponente g' enthält. Hierbei handelt es sich um den modifizierten Algorithmus g, der an den resultierenden Term t' der Termersetzung angepaßt ist. Wir werden etwas weiter unten diskutieren, was es damit auf sich hat.

Die Implementierung der getypten Konversionen ist nun ein Aufsatz auf die bestehenden Konversionen. Dies zeigt sich daran, daß es kein getyptes Analogon zur `Rewrite`-Taktik gibt, da die Einbindung in das System mittels der einfachen Konversionen erfolgt. Die Funktion

`TCtoC` TC

übersetzt die getypte Konversion TC in eine einfache Konversion; während die Umkehrung

`CtoTC` C

die einfache Konversion C in eine getypte Konversion übersetzt. Bei letzterem entfällt bei der Auswertung von C natürlich der Zugang zum Typisierungsalgorithmus. Der Aufruf einer getypten Konversion muß nun in jedem Fall die Funktion `TCtoC` benutzen, da dies das einzige Tor ist, das die taktische Anbindung erlaubt. Innerhalb von `TCtoC` erfolgt die Einsetzung des konkreten Typisierungsalgorithmus, der dann während der Durchführung der Konversion weiterpropagiert wird.

Alle Konversionen und Konversionale sind nun auf der Ebene der getypten Konversionen bereitzustellen. Wir kennzeichnen diese Ebene, indem wir das Suffix „TC“ für die Namen der Konversionen verwenden: `SubIfTC`, `HypTC`, `ANDTHENTC` usw.

Die Konversion `SubIfTC`, mit der wir einzelne Teilterme adressieren können, ist nun in der Lage, die Bindungsvariablen dieser Teilterme mit einem Typ zu deklarieren und diese Deklaration an die nachfolgende Konversion mittels der Umgebung e durchzureichen. Dadurch erreichen wir, daß alle Variablen stets deklariert sind und diese Information in weitere Typisierungen eingehen kann. Das Problem in obigem Beispiel, daß der Typ von x unbekannt ist, kann nun nicht mehr auftreten.

Die Konversion `HypTC` kommt in den Genuß des speziellen Vorteils, daß die Typinformation nun zur Instanziierung der Termersetzungsregel benutzt werden kann. Um das Beispiel des Assoziativitätsgesetzes noch einmal aufzurollen:

$$\forall T{:}\,\mathbf{U}.\ \forall L_1, L_2, L_3{:}\,T\ \text{list}.\quad (L_1 \cdot L_2) \cdot L_3 = L_1 \cdot (L_2 \cdot L_3)$$

Das Problem hieran ist, daß nicht nur die Objektvariablen L_1, L_2, L_3 instanziiert werden müssen, sondern auch die Typvariable T. Diese ist jedoch nur zu erschließen, wenn der Typ der Objektvariablen bekannt ist. In der Formel, auf die die Ersetzung angewendet werden soll, müssen dazu die den Objektvariablen entsprechenden Teilterme typisiert werden. Damit dies gelingt, sollten die Typen aller Variablen bekannt sein, und es sollte ein Algorithmus, der zur Typisierung dieser Terme adäquat ist, zur Verfügung stehen. Getypten Konversionen haben Zugang zu diesen Informationen.

Wir haben noch kein Wort über die innere Struktur der Algorithmen g : `gettype` verloren. Diese entscheidet sehr stark über die Effizienz der getypten Konversionen, da hier insbesondere vermieden werden kann, denselben Term immer wieder neu zu typisieren. Die augenblickliche Implementierung speichert innerhalb von g immer gleich alle Typen aller Teilterme, da diese Information beim Typisieren ohnehin immer anfällt. Beim Abstieg in Teilterme braucht dann gar nichts neu typisiert zu werden; dies wird erst dann notwendig, wenn ein Term tatsächlich ersetzt wurde. Der Typ des Ersatzterms selbst ist natürlich derselbe geblieben wie der des Originalterms, aber seine Teilterme können andere sein. Diese Änderungen müssen dann in den Baum der Typen eingetragen werden. Diese Modifikationen werden in g' zurückgegeben, das zum Resultat einer Konversion gehört.

In Anhang E wird die Anwendung getypter Konversionen im Sinne von `HypTC` diskutiert. Man findet dort noch weitere Informationen, wie das Instanziierungsproblem gelöst wird; und auch eine weitere Anwendung von Konversionen, um feine syntaktische Differenzen zu behandeln.

3.2.5 Optimierte Termersetzung

Wir diskutieren im folgenden eine Möglichkeit, wie eine *Folge* von Termersetzungen optimiert werden kann. Die Unterscheidung zwischen atomaren Konversionen und Konversionalen wird jetzt äußerst wichtig, da zwei Zuständigkeiten getrennt werden: Konversionale haben die Aufgabe, die atomaren Konversionen zu adressieren, d.h. sie auf die gewünschten Teilterme in der gewünschten Reihenfolge anzuwenden. Die Adressierung mittels `SubIfTC` und `ANDTHENTC` legt jedoch nicht nur dieses fest, sondern zugleich auch die konkrete Reihenfolge, in der die einzelnen Teilterme durchlaufen werden. Für den Benutzer einer Termersetzung ist diese Konkretionsstufe jedoch uninteressant, er erwartet, daß der Mechanismus eine möglichst optimale Traversierung wählt. Im folgenden werden wir das Problem auseinanderlegen, haben aber noch keine Lösung parat.

Wir gehen davon aus, daß die Folge der Termersetzungen wie folgt angegeben wird:

$$(\ (a_1, C_1);\ (a_2, C_2);\ \ldots (a_n, C_n)\ )$$

Dabei ist a_i die Adresse des Teilterms, auf den die atomare Konversion C_i angewendet werden soll. Die Folge legt zugleich auch die Reihenfolge fest: Die Termersetzungen sind von links nach rechts durchzuführen.

In solch einer Folge dürfen wir nun die Reihenfolge in gewissen Grenzen ändern, da es nur darauf ankommt, daß die Durchführung der Ersetzung dasselbe Ergebnis liefert wie durch die Originalfolge festgelegt ist. Genauer gesagt, können wir für je zwei Paare eine Relation $\prec$ festlegen, wobei $(a_i, C_i) \prec (a_j, C_j)$ heißt, daß die Konversion bei a_i ausgeführt sein muß, bevor diejenige bei a_j an die Reihe kommt.

Die $\prec$-Relation wird nun durch die Bedingung definiert, daß die Konversion (a_i, C_i) vor allen Konversionen (a_j, C_j) mit $j > i$ ausgeführt werden muß, für die a_j einen Teilterm des durch a_i gegebenen Teilterms adressiert:

$$(a_i, C_i) \prec (a_j, C_j) \iff i < j \ \wedge\ a_i \text{ Präfix von } a_j$$

Es ist nun erlaubt, eine Umordnung der Folge mittels der Permutation ν

$$(\ (a_{\nu_1}, C_{\nu_1});\ (a_{\nu_2}, C_{\nu_2});\ \ldots (a_{\nu_n}, C_{\nu_n})\ )$$

durchzuführen, wenn für die umgeordneten Termersetzungen nur gilt, daß die $<$-Relation (genauer: deren transitive Hülle) erhalten bleibt:

$$\forall 1 \leq i \leq n-1.\ (a_{\nu_i}, C_{\nu_i}) \prec (a_{\nu_{i+1}}, C_{\nu_{i+1}})$$

```
letrec traverseTC ((a_1,C_1);...;(a_n,C_n)) =
  if n = 0 then
    IdTC
  else
    if a_1 = [] then
      C_1 ANDTHENTC traverseTC ((a_2,C_2);...;(a_n,C_n))
    else
      let a.a' = a_1 in
      let k = max { i | ∀2 ≤ j ≤ i. hd a_j = a } in
        AddrTC [a] (traverseTC ((tl a_1,C_2);...;(tl a_k,c_k)))
        ANDTHENTC traverseTC ((a_{k+1},C_{k+1});...;(a_n,C_n))
;;
```

Abb. 3.4. *Traversierung zu einer Folge*

Es ist natürlich die Frage, welche Permutation zu optimaleren Traversierungen führt. Wir definieren dazu eine Kostenfunktion K:

$K(\nu) =$ Anzahl der `SubIfTC`- und `ANDTHENTC`-Aufrufe zur Durchführung der Termersetzung nach der mit ν permutierten Folge

Wieviele `SubIfTC`- und `ANDTHENTC`-Aufrufe benötigt werden, hängt nun aber davon ab, welcher Traversierungsalgorithmus zugrunde gelegt wird. Bevor wir die Frage der optimalen Permutation beantworten können, müssen wir erst einmal wissen, wie eine gegebene Folge optimal traversiert wird. Abbildung 3.4 zeigt diesen Algorithmus. Die Idee ist, den Abstieg in einen Teilterm, der für die erste Konversion der Folge erforderlich ist, gleich für möglichst viele weitere Konversionen zu nutzen. Sei a der Anfang der Adresse von a_1, d.h. die Nummer des Teilterms, in den zunächst abgestiegen werden muß. Es wird das größte k bestimmt, für das die Adressen a_2 bis a_k sich ebenfalls auf diesen Teilterm beziehen. Es wird nun mittels `AddrTC [`a`]` in diesen Teilterm hinabgestiegen und diese Teilfolge traversiert; anschließend (`ANDTHENTC`) wird der Rest traversiert.

Die optimierende Permutation ν sollte nun ähnlich konstruiert werden: Nachdem in einen Teilterm hinabgestiegen wurde, werden alle gemäß $\prec$ möglichen Konversionen vorgezogen, die sich auf diesen Teilterm beziehen. Im Augenblick gibt es dafür noch keinen Algorithmus, geschweige denn eine Untersuchung über die Optimalität der erhaltenen Folgen. Die aktuelle Implementierung enthält nur die optimale Traversierung nach Abbildung 3.4.

Die aktuelle Implementierung enthält dafür eine andere Idee, die noch wert ist, hier vorgestellt zu werden. Damit für den Benutzer möglichst alles beim Alten bleibt, wurde das Interface für die optimierten Konversionen beibehalten, den *adressierenden getypten Konversionen.* Das Suffix für diese Konversionen ist nun „ATC“: `SubIfATC`, `HypATC`, `ANDTHENATC` usw. Diese Konversionen werden jedoch nicht direkt ausgeführt, vielmehr sind sie so programmiert, daß sie die oben vorausgesetzte Folge sukzessive aufbauen. Dies heißt, daß das zunächst berechnete Ergebnis der Anwendung einer ATC-Konversion nicht etwa eine Beweistaktik ist, die die vorgenommene Termersetzung rechtfertigt, sondern nur eine Folge von mit Adressen versehenen TC-Konversionen. Alle Konversionale sind so programmiert, daß sie ihre Wirkung über diese Folge beschreiben; beispielsweise macht `ANDTHENATC` nicht anderes, als zwei Folgen zu konkatenieren, `SubIfATC` readressiert eine Folge (unter der gegebenen Bedingung). Atomare Konversionen wie etwa `HypATC` geben schlicht eine einelementige Folge zurück. Mittels des Konverters `TCtoATC` kann man eine beliebige TC-Konversion als eine solche einelementige Folge auffassen.

Irgendwann muß die Folge interpretiert werden, d.h. sie muß in eine Beweistaktik überführt werden.

Dies ist nur möglich mittels der Übersetzungsfunktion `ATCtoTC`, die die Folge als TC-Konversion deutet; im Grunde passiert *von der Wirkung her* nichts anderes, als daß die Folge $((a_1, C_1), ..., (a_n, C_n))$ ausgeführt wird durch das TC-Konversional

$$\texttt{AddrTC}\ a_1\ C_1\ \texttt{ANDTHENTC} \cdots \texttt{ANDTHENTC}\ \texttt{AddrTC}\ a_n\ C_n$$

Allerdings finden bei dieser Ausführung gar keine Optimierungen bei der Adressierung statt; im Gegenteil, jeder `AddrTC`-Aufruf bewirkt, daß die Adressierung durch die komplette Formel hindurch stattfinden muß. Die aktuelle Implementierung verwendet dagegen den Traversierungsalgorithmus aus Abbildung 3.4, so daß immerhin vermieden wird, daß derselbe Teilterm bei hintereinanderfolgenden Ersetzungen mehrfach adressiert wird. Weitere Optimierungen sind derzeit nicht eingebaut.

In Anhang D findet der Leser eine Liste mit allen im Augenblick implementierten Konversionen und Konversionalen nach dem TC- und ATC-Schema.

Ein besonderes Detail der ATC-Konversionen sei noch hervorgehoben; dies wird in Anhang F ausgenutzt, um eine durch eine Gleichheit gerechtfertigte Substitution *ganz ohne* Adressierungsaufwand durchführen zu können. Da sich in der aufgesammelten Folge immer die Adresse findet, an der die Konversion angewendet werden soll, kann man optional auch *selbstadressierende* Konversionen angeben. Dies sind Konversionen, die einen eigenen effizienten Algorithmus haben, um den gewünschten Teilterm zu adressieren. Die Adresse wird der Konversion als Argument mitgeteilt.

3.3 Termersetzung: Strategie

Nachdem wir geschildert haben, wie sich Termersetzungen durchführen lassen, ist nun natürlich die Frage naheliegend, auf welche Weise die Termersetzungen gesteuert werden sollen, um einen bestimmten Beweis zu führen. Damit hängt zusammen, zu klären, auf welche Formeln Termersetzungen überhaupt anzuwenden sind und bei welchen Formeln man andere Methoden vorziehen sollte.

Diese Fragen sind im Grunde nicht allgemein zu beantworten, allgemein in dem Sinne, eine für alle Fälle optimale Lösung zu geben. Wir befinden uns in einem hochkomplexen Gebiet, in dem niemand mehr mit Bestimmtheit sagen kann, welche Methode allgemein gut ist, weil niemand mehr einen Überblick über die überhaupt vorkommenden Fälle hat. Es ist schließlich das Gebiet, in dem Mathematiker ihre Geschicklichkeit, ihre Erfahrung und ihre Kreativität einbringen, um immer wieder zu staunen, welche Beweismöglichkeiten es gibt.

Wir benötigen eine definierte Erfahrungsbasis, auf deren Grundlage wir die Güte von Methoden beurteilen können. In unserem Falle ist diese Basis glücklicherweise näher bestimmbar: Es sind die Fälle zu untersuchen, die sich bei der Globalsuche als noch zu lösende Beweisziele ergeben. Damit kennen wir immerhin die grobe Struktur der Formeln, denn diese ist (siehe Kapitel 4) festgelegt; desweiteren wissen wir, aus welchen Quellen die Teile der Formeln stammen. Weiteres sicheres Wissen gibt es nicht.

Natürlich ist dies noch viel zu wenig, um als Grundlage dienen zu können. Wir brauchen weiteres Material, das freilich nur aus konkreten Anwendungen der Globalsuchen stammen kann, um auf diesem Material Abstraktionen zu errichten, die dann die Basis der Beurteilung abgeben können. Diese einigermaßen verläßliche Basis gibt es noch nicht, da erst wenige Anwendungen der Globalsuche durchgeführt wurden. Dennoch finden wir bei [smith91] einige Hinweise. So werden etwa, für die von Smith geführten Beispielbeweise, vor allem distributive Termersetzungsregeln benötigt, und zwar potentiell Distributivgesetze, in denen jeder Operator mit jedem Operator distribuiert. Dies heißt, daß in diesen Beispielen Formeln vorkommen, zu deren Beweis es nur auf die relative Bedeutung der Operatoren untereinander ankommt. Insbesondere sind höhere Beweistechniken, etwa Schnitte oder Induktion unnötig.

Wir wollen diesem Hinweis nachgehen und lassen dies vorläufig als unsere Erfahrungsbasis gelten, wohlwissend, daß sie eigentlich noch völlig unzureichend ist.

Als Distributivgesetz wollen wir jede Formel der Art $op_1(op_2(x)) \trianglelefteq t$ betrachten, wobei entscheidend ist, daß die Zusammenwirkung zweier Operatoren op_1 und op_2 beschrieben wird[2]. Beispielsweise können wir für das Zusammenspiel von Elementoperation und Konkatenation für Listen folgendes Gesetz aufstellen:

- $x \in (L \cdot L') \iff x \in L \vee x \in L'$

Auf diese Weise können wir für jedes Paar von Operatoren, die vom Typ her zusammenpassen, ein oder mehrere Gesetze aufstellen, die wir als Termersetzungsregeln deuten.

Gegeben sei nun eine zu beweisende Formel F. Wir können nun diese Formel untersuchen und aufgrund der vorgefundenen Operatorpaare eine Menge von Termersetzungen angeben, die möglicherweise anwendbar sind. Es ist noch zu beachten, daß wir jede Regel vorwärts oder rückwärts lesen können, aber häufig nur eine der beiden Richtungen erlaubt ist, manchmal aber auch beide. Die nun erhaltene Menge von Regeln mit Anwendungsrichtung können wir nun benutzen, um die Regeln der Reihe nach auszuprobieren und auf diese Weise einen Suchraum aufzuspannen.

Neben den vielen Details, die bei der Implementierung eines solchen Suchraums anfallen würden, ist von Interesse,

a) welche Formeln sich überhaupt so beweisen lassen,

b) wie groß der Suchraum dabei typischerweise wird,

c) wann eine Formel dabei als bewiesen gilt.

Auf a) und b) können wir hier keine Antwort geben, weil wir keine Implementierung durchgeführt haben. Es ist jedoch zu beachten, daß wir den Beweiser jederzeit unterstützen können, indem wir zusätzliche Regeln aufstellen, beweisen und in die Wissensbank stellen, so daß bei der Suche diese Regel beachtet wird. Das Wissen des Beweisers ist also nicht fest, sondern erweiterbar; desweiteren können wir Benutzer, die wir vielleicht eine bessere Intuition haben, dem Beweiser Tips geben, bestimmte Regeln früh anzuwenden, damit der Suchraum verkleinert wird.

Zu c) ist zu sagen, daß wir irgendwann eine Formel bekommen, die bereits rein logisch wahr ist. Es ist dann besser, die Termersetzung zu stoppen und eine andere Methode zu wählen, die logische Beweise viel besser durchführen kann. In den folgenden beiden Abschnitten stellen wir einen einfachen logischen Beweiser vor, der vielleicht für diese Aufgabe geeignet ist.

3.4 Einfache logische Beweise: Kalkül

Problematisch bei herkömmlichen Sequenzenbeweisen sind die Regeln zum Einführen von Existenzquantoren bzw. zum Eliminieren von Allquantoren. Man muß den Term, der jeweils für die Variable instanziiert wird, beim Aufruf einer der beiden Regeln angeben. Dies ist aus Sicht der Beweisüberprüfung geboten, für die automatische Suche nach einem Beweis jedoch außerordentlich unpraktisch.

Betrachten wir etwa den Beweis für die Sequenz

$$a + 1 \geq 1 \quad \vdash \quad \exists x{:}\, \mathbf{Z}.\, x < (-1) \;\vee\; x \geq 1$$

[2] Assoziativ- und Kommutativgesetze kann man in diesem Sinne als Distributivgesetze eines Operators mit sich selbst verstehen.

Der Beweis hierfür sieht wie folgt aus, man beachte, daß bei der ex_i-Regel der Term angegeben werden muß, bevor klar ist, welcher Term sinnvoll ist:

$a+1 \geq 1 \quad \vdash \quad \exists x{:}\mathbf{Z}.\ x < (-1) \ \vee \ x \geq 1$
BY ex_i '$a+1$'

  $a+1 \geq 1 \quad \vdash \quad a+1 < (-1) \ \vee \ a+1 \geq 1$
  BY or_i2

    $a+1 \geq 1 \quad \vdash \quad a+1 \geq 1$
    BY hypothesis

Für die reine Suche nach diesem Beweis hätten wir auch x bei der Einführung des Existenzquantors uninstanziiert lassen können und uns lediglich merken müssen, daß eine Instanz für dieses x noch gesucht wird. Die Anwendung von or_i2 kann diesen Unterschied ignorieren. Erst die hypothesis-Regel funktioniert nur, wenn ein Term für x eingesetzt wird. An dieser Stelle müssen wir uns in der Tat entscheiden, welcher Term instanziiert wird.

Im folgenden präsentieren wir einen Kalkül für den Zweck der Beweissuche, in dem die verzögerte Instanziierung eingebaut ist. Der Unterschied zum Sequenzenkalkül besteht im wesentlichen darin, daß Variablen nicht nur mit einem Typ, sondern auch mit ihrem Status deklariert werden: Entweder wird *angenommen*, daß die Variable einen Wert ihres Typs darstellt (also eine Deklaration im herkömmlichen Sinn), oder es ist noch offen, ob es einen Wert für diesen Typ gibt, und wir *suchen* einen solchen Wert. Desweiteren benötigen wir, wenn wir einen Term für eine gesuchte Variable gefunden haben, einen Mechanismus, der die vorgenommenen Instanziierungen verwaltet.

Eine Sequenz hat nun folgende Form:

$$d_1, ..., d_m; \ \ H_1, ..., H_n \ \vdash \ C \ \text{ subst } \sigma$$

Die Hypothesen H_i enthalten nun keine Deklarationen mehr; diese werden in den d_j vorgenommen. C ist wie üblich die Konklusion. In der Substitution σ werden vorgenommene Instanziierungen verwaltet.

Jede Deklaration d_j ist ein Tripel aus dem Variablennamen, seinem Typ und der Markierung, ob es sich um eine angenommene oder eine gesuchte Variable handelt. Wir notieren dies auch als $x\,!{:}\,T$ bzw. $x\,?{:}\,T$. Die Deklarationen sind geordnet: Wenn d_j eine gesuchte Variable deklariert, so darf dafür nur eine Instanz eingesetzt werden, in denen alle freien Variablen in d_k mit $k < j$ deklariert werden.

In Abbildung 3.5 ist der entsprechend modifizierte Kalkül dargestellt. Wie man sieht, hat die neue Struktur der Sequenzen auf die meisten Regeln keinen Einfluß, sofern man den Substitutionsteil außer acht läßt. Bei den Regeln exists_i und all_e muß nun der einzustetzende Term nicht mehr angegeben werden, stattdessen wird die Quantorvariable einfach als gesuchte Variable markiert.

Eine Besonderheit ist ferner, daß die or_i-Regel gleich für ein ganzes „Oder-Gebirge" funktioniert; ein Zugeständnis an die Taktik, die diesen Suchkalkül benutzt (s.u.). Die Elimination für die Implikation fehlt, da diese Taktik im Augenblick ohnehin nicht in der Lage ist, sie zu benutzen.

Die Regeln können zunächst angewendet werden, ohne daß die Substitutionen berechnet werden. Dies erfolgt erst, nachdem der Beweisbaum komplett ist; analog der Extraktion der Typentheorie können nun die Substitutionen von unten nach oben an die Knoten des Baums geschrieben werden. Eventuell mißlingt dies, dann scheitert der Beweis.

Erzeugt werden die Substitutionen nur durch die hypothesis- und die contradiction-Regel, die sie durch eine Variation der Unifikation bestimmen, bei der die Reihenfolge der Deklarationen beachtet wird. Die meisten anderen Regeln reichen die Substitutionen einfach nach oben durch. Die Regeln mod_and_i und mod_or_e führen zwei Zweige zusammen, bei ihnen müssen die beiden Substitutionen verknüpft werden. Die Regeln mod_exists_i und mod_all_e überprüfen die Substitution, die für die Quantorvariable gefunden wurde, und streichen sie heraus, bevor sie die Substitution weiterreichen.

$V;\quad \Gamma \quad \vdash A \wedge B \quad \text{subst } \sigma_1 \circ \sigma_2$
BY mod_and_i
$\quad V;\quad \Gamma \quad \vdash A \quad \text{subst } \sigma_1$
$\quad V;\quad \Gamma \quad \vdash B \quad \text{subst } \sigma_2$

$V;\quad \Gamma, A \wedge B, \Delta \quad \vdash C \quad \text{subst } \sigma$
BY mod_and_e_ i
$\quad V;\quad \Gamma, A, B, \Delta \quad \vdash C \quad \text{subst } \sigma$

$V;\quad \Gamma \quad \vdash \vee(A_1, ..., A_n) \quad \text{subst } \sigma$
BY mod_multi_or_i i
$\quad V;\quad \Gamma \quad \vdash A_i \quad \text{subst } \sigma$

$V;\quad \Gamma, A \vee B, \Delta \quad \vdash C \quad \text{subst } \sigma_1 \circ \sigma_2$
BY mod_or_e i
$\quad V;\quad \Gamma, A, \Delta \quad \vdash C \quad \text{subst } \sigma_1$
$\quad V;\quad \Gamma, B, \Delta \quad \vdash C \quad \text{subst } \sigma_2$

$V;\quad \Gamma \quad \vdash A \Rightarrow B \quad \text{subst } \sigma$
BY mod_imp_i
$\quad V;\quad \Gamma, A \quad \vdash B \quad \text{subst } \sigma$

$V;\quad \Gamma \quad \vdash \neg A \quad \text{subst } \sigma$
BY mod_not_i
$\quad V;\quad \Gamma, A \quad \vdash \mathbf{F} \quad \text{subst } \sigma$

$V;\quad \Gamma \quad \vdash \mathbf{T} \quad \text{subst } \emptyset$
BY mod_true_i

$V;\quad \Gamma, \mathbf{F}, \Delta \quad \vdash C \quad \text{subst } \emptyset$
BY mod_false_e i

$V;\quad \Gamma \quad \vdash \forall x{:}T.\, A \quad \text{subst } \sigma$
BY mod_all_i
$\quad V, x'\,!{:}\,T;\quad \Gamma \quad \vdash A[x'/x]_1 \quad \text{subst } \sigma$

$V;\quad \Gamma, \forall x{:}T.\, A, \Delta \quad \vdash C \quad \text{subst } \sigma - x'$
BY mod_all_e i
$\quad V, x'\,?{:}\,T;\quad \Gamma, A[x'/x]_1, \Delta \quad \vdash C \quad \text{subst } \sigma$

$V;\quad \Gamma \quad \vdash \exists x{:}T.\, A \quad \text{subst } \sigma - x'$
BY mod_exists_i
$\quad V, x'\,?{:}\,T;\quad \Gamma \quad \vdash A[x'/x]_1 \quad \text{subst } \sigma$

$V;\quad \Gamma, \exists x{:}T.\, A, \Delta \quad \vdash C \quad \text{subst } \sigma$
BY mod_exists_e i
$\quad V, x'\,!{:}\,T;\quad \Gamma, A[x'/x]_1, \Delta \quad \vdash C \quad \text{subst } \sigma$

$V;\quad \Gamma, A, \Delta \quad \vdash B \quad \text{subst } \sigma$
BY mod_hypothesis i
falls $A\sigma = B\sigma$

$V;\quad \Gamma, A, \Delta, \neg B, \Pi \quad \vdash C \quad \text{subst } \sigma$
BY mod_contradiction i j
falls $A\sigma = B\sigma$

Abb. 3.5. *Modifizierter Sequenzenkalkül*

Der Beweis für das obige Beispiel lautet im modifizierten Sequenzenkalkül nun wie folgt:

$a\,!\!:\mathbf{Z};\quad \mathbf{a+1\geq 1} \quad\vdash\quad \exists \mathbf{x}\!:\mathbf{Z}.\,\mathbf{x} < (\mathbf{-1}) \vee \mathbf{x} \geq \mathbf{1} \quad \mathbf{subst}\ []$
BY mod_ex_i

$a\,!\!:\mathbf{Z},\ \mathbf{x}\,?\!:\mathbf{Z};\quad \mathbf{a+1\geq 1} \quad\vdash\quad \mathbf{x} < (\mathbf{-1}) \vee \mathbf{x} \geq \mathbf{1} \quad \mathbf{subst}\ [\mathbf{a+1/x}]$
BY mod_multi_or_i 2

$a\,!\!:\mathbf{Z},\ \mathbf{x}\,?\!:\mathbf{Z};\quad \mathbf{a+1\geq 1} \quad\vdash\quad \mathbf{x} \geq \mathbf{1} \quad \mathbf{subst}\ [\mathbf{a+1/x}]$
BY hypothesis

Es ist einsichtig, daß man diesen Beweis leicht in einen Beweis für den originären Sequenzenkalkül verwandeln kann, indem bei der Einführung des Existenzquantors der zuständige Eintrag in der Substitution genutzt wird, um den Term zu bestimmen, der die Existenz evident macht.

Die Korrektheit des modifizierten Kalküls stützt sich allein darauf, daß seine Beweise in Sequenzenbeweise übersetzt werden können. Für einen Kalkül, der lediglich zur Suche benutzt wird, ist dies ausreichend; letztlich können wir immer einen echten Sequenzenbeweis bekommen.

Eine Taktik, die diesen Kalkül nun zur Beweissuche verwendet, besteht demnach aus zwei Teilen:

- Die Suche des Beweises im modifizierten Kalkül, und
- die Transformation des gefundenen Beweises in einen Beweis, der von NuPRL akzeptiert wird.

Bevor wir uns der Taktik zuwenden, wollen wir uns noch einige Gedanken über die Leistungsfähigkeit des modifizierten Kalküls machen. Diese ist offensichtlich, da es keine Implikations-Elimination gibt, stark eingeschränkt. Es ist damit praktisch nicht möglich, innerhalb dieses Kalküls Lemmata anzuwenden. Der Kalkül ist jedoch gedacht als Ergänzung der Termersetzung, die ja nichts anderes ist als eine elaborierte Lemmaanwendung. Insofern haben wir uns diese Möglichkeit immer noch, es ist nur die Frage, wie sich das Zusammenspiel von Termersetzung und Logikbeweis auf die Beweisstärke auswirkt.

Grundidee des Zusammenspiels ist ein Nacheinander: Zuerst mehrere Termersetzungen, bis sich eine Formel ergibt, die nur noch Logik benötigt, um bewiesen zu werden. Wir zeigen dies an einem Beispiel (Abbildung 3.6). Die zu beweisende Formel wird zunächst durch drei Anwendungen von Termersetzungsregeln soweit umgeformt, bis durch den Suchkalkül eine Instanz für S gefunden werden kann. Wie man sieht, liegt die Substanz des Beweises im Termersetzungsteil, der die Lösung präjudiziert; durch andere Termersetzungen ergäben sich andere Instanzen für S (jede Teilmenge von $\{1..n\}$ löst die Formel). Der Logikteil ist nur ein Anhang, der die Richtigkeit der Termersetzungen bestätigt.

Der abgebildete Beweis hat dennoch einen Haken: In der zweiten Termersetzung sind die Instanzen von k und l vom Himmel gefallen. Im Prinzip ist jedoch die gleiche Methode anwendbar, wie bei der Erschließung von S: Man läßt k und l solange unbestimmt, bis mögliche Instanzen offensichtlich werden. Dies bedeutet aber, daß der Termersetzungsteil ebenso wie der Logikteil eine vorausschauende Suche benötigt, da die Typentheorie verzögerte Instanziierung nicht erlaubt. Derzeit ist dies noch nicht implementiert.

3.5 Einfache logische Beweise: Taktik

Ziel dieser Logiktaktik ist nun nicht, alle wahren Formeln zu beweisen; dies würde weit über die Notwendigkeiten hinausschießen. Die Suche nach dem Beweis soll von vorneherein stark begrenzt sein; da diese Taktik als Ergänzung einer systematischen Termersetzungssuche gedacht ist, muß sie schnell entscheiden, ob sie den Beweis finden kann oder ob nicht. Beweise, die einen großen logischen

$\vdash \quad \forall n{:}\,\mathbf{Z}.\; n \geq 1 \Rightarrow \forall T{:}\,\mathtt{Set}(\alpha).\; \exists S{:}\,\mathtt{Set}(\mathbf{Z}).\;\; |S \cup \{n\}| \geq |T| \Rightarrow |\{1..n\}| \geq |T|$

BY *Ersetze via* $\forall i{:}\,\mathbf{Z}.\; \{i\} = \{i..i\}$

  $\vdash \quad \forall n{:}\,\mathbf{Z}.\; n \geq 1 \Rightarrow \forall T{:}\,\mathtt{Set}(\alpha).\; \exists S{:}\,\mathtt{Set}(\mathbf{Z}).\;\; |S \cup \{n..n\}| \geq |T| \Rightarrow |\{1..n\}| \geq |T|$

  BY *Ersetze via* $\forall i,j,k,l{:}\,\mathbf{Z}.\; k \leq i \wedge j \leq l \;\Rightarrow\; \{i..j\} \subseteq \{k..l\}$ *mit* $k = 1$ *und* $l = n$

    $--$ antecedent $--$

    $\ldots \vdash \quad 1 \leq n \wedge n \leq n$

    $\vdash \quad \forall n{:}\,\mathbf{Z}.\; n \geq 1 \Rightarrow \forall T{:}\,\mathtt{Set}(\alpha).\; \exists S{:}\,\mathtt{Set}(\mathbf{Z}).\;\; |S \cup \{1..n\}| \geq |T| \Rightarrow |\{1..n\}| \geq |T|$

    BY *Ersetze via* $\forall M{:}\,\mathtt{Set}(\beta).\; M = \emptyset \cup M$

      $\vdash \quad \forall n{:}\,\mathbf{Z}.\; n \geq 1 \Rightarrow \forall T{:}\,\mathtt{Set}(\alpha).\; \exists S{:}\,\mathtt{Set}(\mathbf{Z}).\;\; |S \cup \{1..n\}| \geq |T| \Rightarrow |\emptyset \cup \{1..n\}| \geq |T|$

$\vdash \quad \forall n{:}\,\mathbf{Z}.\; n \geq 1 \Rightarrow \forall T{:}\,\mathtt{Set}(\alpha).\; \exists S{:}\,\mathtt{Set}(\mathbf{Z}).$
$\quad |S \cup \{1..n\}| \geq |T| \Rightarrow |\emptyset \cup \{1..n\}| \geq |T| \;\; \mathrm{subst}\; []$

BY mod_all_i

  $n\,!{:}\,\mathbf{Z};\;\; \vdash \quad n \geq 1 \Rightarrow \forall T{:}\,\mathtt{Set}(\alpha).\; \exists S{:}\,\mathtt{Set}(\mathbf{Z}).$
  $\quad |S \cup \{1..n\}| \geq |T| \Rightarrow |\emptyset \cup \{1..n\}| \geq |T| \;\; \mathbf{subst}\; []$

  BY mod_imp_i

    $n\,!{:}\,\mathbf{Z};\;\; \mathbf{n \geq 1}$
    $\vdash \quad \forall T{:}\,\mathtt{Set}(\alpha).\; \exists S{:}\,\mathtt{Set}(\mathbf{Z}).\;\; |S \cup \{1..n\}| \geq |T| \Rightarrow |\emptyset \cup \{1..n\}| \geq |T| \;\; \mathrm{subst}\; []$

    BY mod_all_i

      $n\,!{:}\,\mathbf{Z},\; \mathbf{T}\,\mathbf{!{:}}\,\mathtt{Set}(\alpha);\;\; \mathbf{n \geq 1}$
      $\vdash \quad \exists S{:}\,\mathtt{Set}(\mathbf{Z}).\;\; |S \cup \{1..n\}| \geq |T| \Rightarrow |\emptyset \cup \{1..n\}| \geq |T| \;\; \mathrm{subst}\; []$

      BY mod_ex_i

        $n\,!{:}\,\mathbf{Z},\; \mathbf{T}\,\mathbf{!{:}}\,\mathtt{Set}(\alpha),\; \mathbf{S}\,\mathbf{?{:}}\,\mathtt{Set}(\mathbf{Z});\;\; \mathbf{n \geq 1}$
        $\vdash \quad |S \cup \{1..n\}| \geq |T| \Rightarrow |\emptyset \cup \{1..n\}| \geq |T| \;\; \mathrm{subst}\; [\emptyset/S]$

        BY mod_imp_i

          $n\,!{:}\,\mathbf{Z},\; \mathbf{T}\,\mathbf{!{:}}\,\mathtt{Set}(\alpha),\; \mathbf{S}\,\mathbf{?{:}}\,\mathtt{Set}(\mathbf{Z});\;\; \mathbf{n \geq 1},\; |\mathbf{S} \cup \{\mathbf{1..n}\}| \geq |\mathbf{T}|$
          $\vdash \quad |\emptyset \cup \{1..n\}| \geq |T| \;\; \mathrm{subst}\; [\emptyset/S]$
          BY mod_hypothesis

Abb. 3.6. *oben: Termersetzungsteil des Beweises, unten: Logikteil*

Anteil haben (der nicht Termersetzung ist), können somit nicht gefunden werden. Implizit machen wir damit die Annahme, daß das Wissen, was über Termersetzung in einen Beweis einfließt, eine viel größere Bedeutung hat als das logische Wissen. Die Zukunft mag zeigen, ob diese Annahme für Synthesetaktiken wie die Globalsuche gerechtfertigt ist oder nicht.

Diese Taktik hier benutzt nun zur Beweisdarstellung den modifizierten Sequenzenkalkül. Der generelle Ansatz ist: Um eine Sequenz zu beweisen, probiere alle Regeln aus; für jede durch Regelanwendung erhaltene neue Sequenz wird das Beweisverfahren rekursiv wiederholt. Die Tiefe der ineinander verschachtelten Regelanwendungen wird begrenzt. – Es ist klar, daß man diesen naiven Ansatz verbessern kann. Dazu versuchen wir, die Eigenschaften der Regeln zu nutzen.

- Zum einen gibt es Regeln, die aus der gegebenen Sequenz nur eine Untersequenz erzeugen, wobei in dieser noch dieselben Informationen enthalten sind, nur in etwas anderer Darstellung. Beispiele hierfür sind mod_and_e, mod_imp_i, mod_not_i, mod_all_i, mod_all_e, mod_exists_i, mod_exists_e. Wenn man diese Regeln anwendet, die wir *linear* nennen wollen, hält man sich noch alle Optionen offen, und man vergrößert den Suchraum nicht.

- Manche Regeln erzeugen mehr als eine Untersequenz, wobei aber durch die Summe aller Untersequenzen alle Möglichkeiten abgedeckt sind. Diese Regeln sollen *hyperlinear* heißen. Man hält sich immer noch alle Optionen offen, vergrößert aber den Suchraum. Hierzu gehören mod_and_i und mod_or_e.

- Bei manchen Regeln muß man sich jedoch für eine Beweisfortsetzung unter mehreren entscheiden. Diese *disjunktiven* Regeln vergrößern nicht einfach den Suchraum, sondern unterteilen diesen in mehrere Segmente. Hierzu gehören mod_multi_or_i, mod_hypothesis und mod_contradiction (die letzten beiden deshalb, weil man durch eine unterschiedliche Wahl der Hypothesen in der Regel unterschiedliche Substitutionen erhält, unter denen man sich entscheiden muß).

- Desweiteren natürlich die *finalen* Regeln:
  mod_true_i, mod_false_e, mod_hypothesis und mod_contradiction.

- Wir können die Regeln aber auch nach der Behandlung der Substitutionen klassifizieren. Das Gros der Regeln reicht einfach die Subtitution nach oben weiter.

- Einige Regeln (mod_and_i und mod_or_e) verknüpfen jedoch zwei Substitutionen. Diese Verknüpfung kann scheitern, weil sie widersprüchlich ist.

- Einige Regeln entfernen lediglich einen Eintrag aus der Substitution (mod_all_e und mod_exists_i). Dies ist völlig unbedenklich.

- Zwei Regeln (mod_hypothesis und mod_contradiction) sind desweiteren noch dafür zuständig, Substitutionen überhaupt zu erzeugen. Es kann nun passieren, daß mehrere Substitutionen möglich sind, aber nur eine zum Beweiserfolg führt, während die anderen in einem höheren Beweisknoten zum Widerspruch führen.

Die augenblickliche Implementierung verfährt so, daß zunächst alle linearen und hyperlinearen Regeln angewendet werden, die möglich sind, dann maximal eine Anwendung von mod_multi_or_i und schließlich ggf. weitere lineare und hyperlineare Regeln. Es wird versucht, finale Regeln möglichst früh zu finden. Der Suchraum ist auf diese Weise recht knapp bemessen.

Es wird jedoch nicht probiert, mehrere Substitutionen auszuprobieren, um eine passende zu finden; vielmehr wird stets die erstbeste genommen. (Um dies besser in den Griff zu bekommen, sollte man überlegen, den Kalkül noch einmal so zu verändern, daß mehrere Kandidaten für die Substitution mitgeführt werden können.)

Kapitel 4

Implementierte Globalsuche

Nachdem wir in den vorangegangenen zwei Kapiteln eine Grundlage gelegt haben, um Algorithmentheorien zu formalisieren und zu operationalisieren, kommen wir zu unserem ursprünglichen Vorhaben zurück, die Globalsuche exemplarisch zu implementieren. Wir wollen an dieser Stelle, an der mehrere von uns eingeschlagene Wege wieder zusammenfinden, kurz zusammenfassen, was wir bisher untersucht haben.

Die Beschreibung der Globalsuche in Kapitel 1 orientierte sich noch nicht an einem Kalkül, sondern eher an der Intuition, obgleich natürlich eine Art Vorstrukturierung im Hinblick auf die Formalisierung stattgefunden hat. Dies ermöglicht uns, zunächst unser Vorhaben an der Sache selbst zu demonstrieren, und erst in einem anschließenden Schritt eine Formalisierung zu finden, die der Sache und der Intuition gerecht wird. Der radikale Formalismus der Typentheorie läßt gerade die Notwendigkeit der intuitiven Vorarbeit umso deutlicher werden, weil die Gefahr droht, orientierungslos in einem Sumpf aus Formeln, die niemand mehr versteht, steckenzubleiben. Die Theorie der Globalsuche ist bereits relativ groß und unübersichtlich, und man kann eigentlich nicht erwarten, daß eine derart umfängliche Theorie begriffen wird, wenn sie nur in detailversessenen Formeln vorgestellt wird. In Kapitel 1 wurden deshalb bewußt Bilder[1] benutzt, um die Globalsuche zu erklären; Formeln dienten nur der Präzisierung, und nur, wenn dies das Verständnis erleichterte. Diese Bilder haben nicht nur didaktischen Wert; sie drücken eine bestimmte Perspektive auf die Sache aus, die Formeln nicht einnehmen können. Formeln sind hervorragend geeignet, um Ergebnisse zu fixieren; Bilder erlauben vor allem, das Vorstellungsvermögen in Bewegung zu setzen und z.B. nachzuzeichnen, welche Bewegung zu dieser bestimmten Theorie geführt hat. Dies ist auch ein Teil der Wahrheit.

In der folgenden Formalisierung werden wir eine typentheoretische Fassung der Globalsuche vorstellen. Der Leser oder die Leserin möge einmal versuchen, diese Fassung zu modifizieren. Er oder sie wird wahrscheinlich die Erfahrung machen, daß bereits kleine Änderungen an einer Stelle Auswirkungen auf beinahe die ganze Theorie haben. Dabei geht es fast nur um Details der Formulierung, kaum um Substantielles. Zugleich wird man, für sich, stets von diesen Details absehen, um überhaupt den Überblick zu behalten. Wir können daran sehen, wir starr eine Theorie wird, wenn sie formalisiert wird, und wie wenig wir uns als verstehende Wesen daran halten (können). – Diese Problematik im

[1] *Bild* ist hier auf keinen Fall wörtlich zu nehmen; es ist vielmehr alles gemeint, was die reine Ebene der Symbole überschreitet. Ein Bild ist insbesondere dadurch ausgezeichnet, daß man nicht mehr frei (wie bei einem Symbol) ist, Bedeutungen zuzuordnen, sondern vom Bild in bestimmter Weise vereinnahmt wird. Es mag für Informatiker befremdlich sein, Bildern einen Erkenntniswert zuzugestehen, weil man an die (symbolische) Nominaldefinition gewohnt ist. Bilder verweisen auch stets auf einen sozial-psychologischen Kontext, der außerhalb der Disziplin steht. Immer wenn sich die Informatik der Welt zuwendet, muß sie sich aber ohnehin mit den dortigen Bildern beschäftigen oder schafft sogar, in der Regel unwillkürlich, selbst Bilder. Es wäre ein Fortschritt, diesen Aspekt in die Wissenschaft selbst mit aufzunehmen. Dies kann darin bestehen, Bilder, um die man nicht umhin kommt, bewußt zu schaffen, oder darin, zu untersuchen, wo wie in der Informatik mit Bildern zu rechnen haben und welche Erkenntnis wir daraus ableiten können.

Auge behaltend, können wir eine wichtige Regel für die Formalisierung aufstellen: Formalisiere so, daß die Formeln in die eigene Vorstellung (bzw. die kollektive Vorstellung der Wissenschaftler) übersetzbar bleiben! Durch die Übersetzbarkeit können wir nämlich die Starre der Formalisierung jederzeit aufbrechen.

Die Implementierung der Globalsuche, wie sie im folgenden vorgestellt wird, wendet nun Grundlagen an, die in den vorangegangenen Kapitel gelegt wurden. Wir haben jetzt mit den Problemen einer konkreten Formalisierung einer Theorie zu kämpfen, die zum einen verständlich bleiben muß und zum anderen in bestimmter Weise operationalisierbar sein soll. Letzteres heißt hier, die Theorie auf ein bestimmtes Problem anzuwenden und die damit verbundenen Aufgaben zu lösen.

4.1 Formalisierung der Globalsuche

Die Formalisierung werden wir in zwei Phasen vornehmen: Zunächst die einfache, ggf. mit einem Filter versehene Globalsuche, darauffolgend die spezialisierte Globalsuche. Dieser Aufbau entspricht der in Kapitel 1 vorgegebenen Strukturierung.

Die folgenden Erläuterungen gehen top-down vor, weil dadurch klarer wird, daß die Formalisierung zielorientiert durchgeführt wird. Am Anfang steht immer eine Aussage, die ausgedrückt werden soll; im Verlauf werden wir uns der Ausdrucksmittel versichern, die wir bereits verwendet haben.

Diese Richtung, von oben her an die Sache heranzugehen, ist der tatsächlichen Implementierungsrichtung genau entgegengesetzt. NuPRL verlangt von uns, zuerst grundlegende Abstraktionen einzugeben, um diese dann in höheren Abstraktionen benutzen zu dürfen. Hier ist noch viel Arbeit in die Verbesserung von NuPRL zu stecken, damit dieser *Editor* wirklich die *Entwicklung* der Formalisierung unterstützt, und nicht, wie zur Zeit, voraussetzt, daß diese bis ins Detail vorausgeplant ist.

Wir bemerken, daß die folgenden Definitionen und Sätze der NuPRL-Bibliothek direkt entnommen sind. Die Formatierung entspricht im wesentlichen derjenigen, die auch ein NuPRL-Bneutzer zu sehen bekäme (bis auf einige der schlimmsten Fehltritte, die manuell ausgebügelt wurden). Die Organisation des Umbruchs und der Klammersetzung ist eine Wissenschaft für sich; im ganzen muß der jetzige Stand schon als recht brauchbar angesehen werden.

4.1.1 Einfache und gefilterte Globalsuche

Zunächst das Basis-Theorem, auf dem alles andere aufbaut:

gs_synth_filterless:

```
∀gs:GS
 (gs fulfils search axioms)
 ⇒ (gs fulfils wf axioms)
 ⇒ SCHEME
    ∀
      O SATISFIES odec_problem(specification(gs))
    ⇒
      filterless_body(gs; O) SATISFIES (specification(gs)*)
   END
```

Dieses Theorem hat die Struktur, daß zwei axiomatische Voraussetzungen erfüllt sein müssen, damit das Programmschema SCHEME $\cdots$ END gilt. Formuliert ist dieses Theorem für einen Globalsuchraum gs, was dadurch ausgedrückt wird, daß gs als Typ GS aller Globalsuchräume deklariert wird. Wir können uns unter gs das Tupel $\langle spec, S, J, s_0, sat, split, ext, Phi\rangle$ vorstellen, das wir schon aus Kapitel 1 kennen.

Die genaue Darstellung haben wir noch festzulegen. An dieser Stelle müssen wir nur wissen, daß mit `specification`(gs) der *spec*-Teil dieses Tupels referenziert wird.

Damit zu einem Globalsuchraum ein Algorithmus generierbar ist, müssen (siehe Definition 1.2.2 und Lemma 1.2.3, Seite 29) fünf Axiome gelten. Diese wurden hier in zwei Gruppen aufgeteilt, weil dies für die Hinzunahme der Filterung hilfreich ist. Die Axiome 1 bis 4 sind die „`search axioms`", das Wohlfundiertheitsaxiom 5 firmiert unter „`wf axioms`".

Das Programmschema beschreibt die eigentliche Generierung. Es besteht aus zwei Teilen: zunächst werden die Programme angegeben, deren Existenz vorausgesetzt wird (hier nach dem $\forall$-Zeichen), dann das Programm, das generiert werden soll (hier nach dem $\Rightarrow$-Zeichen). Hier wird nun O vorausgesetzt als ein Programm, das die Spezifikation `odec_problem`(...) erfüllt. Die Details siehe unten; im Augenblick reicht es zu wissen, daß hiermit eine Entscheidungsfunktion O für die in der gs-Spezifikation enthaltene Ausgabebedingung gefordert wird. – Generiert wird `filterless_body`($gs; O$), wobei `filterless_body` das Programmgerüst wie in Lemma 1.2.3 angibt, das nun mit gs und O instanziiert wird. Wichtig ist nun noch, daß nicht direkt die gs-Spezifikation, die ja nur das Punktproblem der Globalsuche angibt, sondern das entsprechende Mengenproblem, erkennbar am $*$-Zeichen, synthetisiert wird.

Erläuterungen zum Konzept des Programmschemas findet man auch in Abschnitt 2.6.2, Seite 89 und im Anhang B.

Erläuterungen zu den Konzepten Programm und Spezifikation findet man in Abschnitt 2.6.1, Seite 88.

Darstellung des Globalsuchraums

Als gs darf nun nicht ein beliebiges Tupel eingesetzt werden, sondern nur eines, das bestimmte Nebenbedingungen erfüllt. Aus diesem Grunde erfolgt die Defionition des zugehörigen Typs `GS` in zwei Stufen:

1. Definiere den unterliegenden Typ des Tupels `GS0`. Hier wird nur festgelegt, welche Typen für die Komponenten des Tupels vorgesehen sind.
2. Definiere den eigentlichen Typ `GS` als den Teiltyp derjenigen Elemente von `GS0`, die die Nebenbedingungen erfüllen.

Die explizite Definition `GS0` empfiehlt sich auch deshalb, weil in den Nebenbedingungen ja über Tupel vom Typ `GS0` gesprochen wird, und man eine Bezeichnung wie `GS0` vorsehen sollte, damit man den Typ leicht angeben kann (etwa in Typisierungsbeweisen). Es ist davon abzuraten, `GS` in einem Schritt zu definieren.

Hier also `GS0` als abhängiger Produktraum für Tupel der Art $\langle spec, S, J, s_0, sat, split, ext, Phi\rangle$, und, darauf aufbauend, `GS` als Teiltyp mit Bedingung `GS-Domain`:

gs_descr0:

$$
\begin{aligned}
&\mathtt{GS0}\\
&\equiv gsspec{:}\mathtt{SPECS}\\
&\quad\times\ S{:}\mathtt{TYPES}\\
&\quad\times\ \mathtt{D}(gsspec)^T \times S^T \not\to \mathbf{P}\\
&\quad\times\ \mathtt{D}(gsspec)^T \not\to S^T\\
&\quad\times\ ((\mathtt{R}(gsspec)^T \times S^T) \to \mathbf{P})\\
&\quad\times\ \mathtt{D}(gsspec)^T \times S^T \not\to \mathtt{set}(S)\\
&\quad\times\ (S^T \to \mathtt{set}(\mathtt{R}(gsspec)))\\
&\quad\times\ \mathtt{D}(gsspec)^T \times S^T \not\to \mathbf{B}
\end{aligned}
$$

gs_descr:

$$\begin{array}{l} \texttt{GS} \\ \quad \equiv \{gs\texttt{:GS0}\,|\ \texttt{GS}-\texttt{Domain}(gs)\} \end{array}$$

Man beachte, daß (im Gegensatz zur Darstellung in Kapitel 1) der Filter *Phi* mit in den Globalsuchraum aufgenommen wurde. Für die ungefilterte Globalsuche kann diese Komponente ignoriert werden; der Übergang zur gefilterten Globalsuche erfolgt dadurch, daß der Filter in die übrigen Komponenten eingebaut wird. Auf diese Weise kann jeder Globalsuchraum ungefiltert oder gefiltert interpretiert werden, je nach dem, ob man die *Phi*-Komponente beachtet oder nicht.

Bei den Nebenbedingungen geht es darum, daß die Komponenten J, *split*, *Phi* und s_0 als partielle Funktionen eingeführt worden sind und noch eine Aussage über deren Definitionsbereich gemacht werden muß. Diese besteht darin, daß diese Funktionen jeweils terminieren müssen, falls die Eingabebedingung I in *gs*-Spezifikation erfüllt ist. Formalisiert wird dies wie folgt:

gs_domain:

$$\begin{array}{l} \texttt{GS}-\texttt{Domain}(gs) \\ \quad \equiv \texttt{let}\ gsspec, S, J, s0, sat, split, ext, Phi = gs \\ \qquad \texttt{in let}\ D, R, I, O = gsspec \\ \qquad\quad \texttt{in xy}-\texttt{Domain}(D^T, S^T, I, J) \\ \qquad\qquad\qquad \wedge\ \texttt{xy}-\texttt{Domain}(D^T, S^T, I, split) \\ \qquad\qquad\qquad \wedge\ \texttt{xy}-\texttt{Domain}(D^T, S^T, I, Phi) \\ \qquad\qquad\qquad \wedge\ \texttt{x}-\texttt{Domain}(D^T, I, s0) \end{array}$$

gs_xy_domain:

$$\begin{array}{l} \texttt{xy}-\texttt{Domain}(X, Y, P, f) \\ \quad \equiv \forall x{:}X.\ P\ x \Rightarrow (\forall y{:}Y.\ f\ \texttt{halts on}\ \langle x, y\rangle) \end{array}$$

gs_x_domain:

$$\begin{array}{l} \texttt{x}-\texttt{Domain}(X, P, f) \\ \quad \equiv \forall x{:}X.\ P\ x \Rightarrow f\ \texttt{halts on}\ x \end{array}$$

Es wird zunächst das Tupel *gs* in seine Komponenten zerlegt; hierzu dient eine spezielle `let`-Variante, die wir gleich näher betrachten. Für drei der betrachteten Funktionen gilt, daß sie von $D^T \times S^T$ aus abbilden; für diese wird die Bedingung mittels `xy-Domain` angegeben. Die Funktion s_0 bildet jedoch nur aus D^T ab, daher muß eine gesonderte Bedingung mittels `x-Domain` formuliert werden. Die Bedingungen sind jeweils Terminierungsaussagen.

Die Zerlegung von *gs* in Komponenten bedient sich folgender Abstraktion:

gs_spread:

$$\begin{array}{l} \texttt{let}\ gsspec, S, J, s0, sat, split, ext, Phi = gs \\ \texttt{in}\ term[gsspec; S; J; s0; sat; split; ext; Phi] \\ \quad \equiv \texttt{gs\_spread\_0}(gs \\ \qquad ; \lambda gsspec, S, J, s0, sat, split, ext, Phi.\ term[gsspec; S; J; s0; sat; split; ext; Phi]) \end{array}$$

gs_spread_0:

$$\begin{array}{l} \texttt{gs\_spread\_0}(gs; f) \\ \quad \equiv \texttt{let}\ gsspec, S, J, s0, sat, split, ext, Phi = gs \\ \qquad \texttt{in}\ f\ gsspec\ S\ J\ s0\ sat\ split\ ext\ Phi \end{array}$$

Dies sieht in der Summe ziemlich tautologisch aus, hat aber dennoch seinen Sinn. Die *gs_spread*-Abstraktion benutzt *gs_spread_0*, die eine äquivalente Abstraktion ist, jedoch nur Variablen 1. Ordnung verwendet. Darin wird wiederum auf das allgemeine `let`-Konstrukt zurückgegriffen, genauer: auf die Abstraktion **spreadn**, die in Anhang C im Einzelnen besprochen wird. Der Effekt der *gs_spread*-Abstraktion ist nun genau derselbe wie derjenige des allgemeinen `let`-Konstrukts; weshalb also der Aufwand?

Der Unterschied liegt, darin, daß man für das allgemeine `let` kein Wf-Lemma angeben kann, zum einen wegen der Variablen Komponentenzahl dieses `let`s, zum anderen, weil in dem allgemeinen `let` keine Hinweise auf den Typ des zu zerlegenden Tupels vorhanden sind. Da wir Globalsuchräume mit Hilfe von abhängigen Produkträumen darstellen, können wir hier auch keine Hilfe von Typinferenz-Algorithmen erwarten; diese sind chancenlos. Die pragmatische Lösung besteht darin, den Typ des Tupels dadurch zu codieren, daß eine besondere Abstraktion eingeführt wird. Dies heißt, wir sollten stets die Zerlegung eines *gs*-Tupels durch die spezielle *gs_spread*-Abstraktion vornehmen lassen, weil dies gleichzeitig der Hinweis ist, daß nicht irgendein Tupel, sondern ein `GS`-Tupel analysiert wird.

Für *gs_spread* kann nun leider ebenfalls kein Wf-Lemma angegeben werden, da dies eine Abstraktion zweiter Ordnung ist. Indem wir aber eine äquivalente Abstraktion erster Ordnung *gs_spread_0* einschieben, können wir zum einen indirekt doch ein Wf-Lemma formulieren, und zum anderen *gs_spread* speziell markieren (als *soft abstraction*), so daß diese Abstraktion in Typisierungsbeweisen automatisch aufgefaltet wird und die nun sichtbare *gs_spread_0*-Abstraktion typisiert werden kann.

Zur Rückführung von Abstraktionen zweiter Ordnung auf solche erster Ordnung siehe auch den Abschnitt 2.2.2, Seite 49.

Das geschilderte Problem der Typisierung von Tupelzerlegungen tritt auch bei Spezifikationen auf und kann in derselben Weise gelöst werden, indem eine spezielle Zerlegung nur für Spezifikationen definiert wird.

Um auf die Komponenten von *gs* besser zugreifen zu können, gibt es noch folgende spezielle Zugriffsoperationen:

gs_spec:

$$\texttt{specification}(gs)$$
$$\equiv \texttt{let}\ gsspec, S, J, s0, sat, split, ext, Phi = gs\ \texttt{in}\ gsspec$$

gs_S:

$$\texttt{S}(gs)$$
$$\equiv \texttt{let}\ gsspec, S, J, s0, sat, split, ext, Phi = gs\ \texttt{in}\ S$$

Analog zu *gs_S* werden Zugriffsoperationen für die übrigen Komponenten definiert: `J`(gs), `s0`(gs), `sat`(gs), `split`(gs), `ext`(gs) und `Phi`(gs).

Man beachte, daß der Beweis der zughörigen Wf-Lemmata ausgesprochen schwierig ist. Siehe hierzu Anhang C.

Axiome der einfachen Globalsuche

Nun zur genauen Gestalt der Axiome in obigem Theorem, die ja in zwei Gruppen eingeteilt sind. Die vier Suchaxiome werden je in eigenen Abstraktionen definiert. Die Wf-Lemmata für die Axiome sind übrigens nichttrivial, was daran liegt, daß die Nebenbedingungen des Typs `GS` herangezogen werden müssen. Die Axiome stimmen mit denjenigen überein, die in Kapitel 1 angegeben wurden, so daß wir uns hier auf die Kommentierung einiger Besonderheiten der Formalisierung beschränken.

gs_fulfils_search_axioms:

$$
\begin{array}{l}
(gs\ \texttt{fulfils search axioms}) \\
\quad \equiv \texttt{gs\_fulfils\_axiom\_1}(gs) \\
\qquad \wedge\ \texttt{gs\_fulfils\_axiom\_2}(gs) \\
\qquad \wedge\ \texttt{gs\_fulfils\_axiom\_3}(gs) \\
\qquad \wedge\ \texttt{gs\_fulfils\_axiom\_4}(gs)
\end{array}
$$

gs_fulfils_axiom_1:

$$
\begin{array}{l}
\texttt{gs\_fulfils\_axiom\_1}(gs) \\
\quad \equiv \forall x{:}\texttt{D(specification(}gs\texttt{))}^T \\
\qquad\quad \texttt{I(specification(}gs\texttt{))}\ x \Rightarrow \texttt{J(}gs\texttt{)<}x\texttt{,s0(}gs\texttt{)(}x\texttt{)>}
\end{array}
$$

Man beachte in dem vorigen Axiom, daß es sich bei $\texttt{J}(gs)$ um eine Funktion handelt, auf die ein Paar $\langle\cdots,\cdots\rangle$ angewendet wird. Ergebnis der Anwendung ist eine Proposition.

gs_fulfils_axiom_2:

$$
\begin{array}{l}
\texttt{gs\_fulfils\_axiom\_2}(gs) \\
\quad \equiv \forall x{:}\texttt{D(specification(}gs\texttt{))}^T \\
\qquad\quad \forall s{:}\texttt{S(}gs\texttt{)}^T \\
\qquad\qquad \texttt{I(specification(}gs\texttt{))}\ x \\
\qquad\qquad \Rightarrow \texttt{J(}gs\texttt{)<}x\texttt{,}s\texttt{>} \\
\qquad\qquad \Rightarrow (\forall t{:}\texttt{S(}gs\texttt{)}^T \\
\qquad\qquad\qquad \lceil t{:}\texttt{S(}gs\texttt{)} \in \texttt{split(}gs\texttt{)<}x\texttt{,}s\texttt{>}\rceil \Rightarrow \texttt{J(}gs\texttt{)<}x\texttt{,}t\texttt{>})
\end{array}
$$

Die Funktion $\texttt{split}(gs)$ ergibt nun, wenn das Paar angewendet wird, eine endliche Menge von Deskriptoren (aus S). Der Operator $\cdot{:}\cdot \in \cdot$ beschreibt, daß die (hintere) Menge das (vordere) Element enthält, wobei die Elemente der Menge den (mittleren) Typ haben.

gs_fulfils_axiom_3:

$$
\begin{array}{l}
\texttt{gs\_fulfils\_axiom\_3}(gs) \\
\quad \equiv \forall x{:}\texttt{D(specification(}gs\texttt{))}^T \\
\qquad\quad \forall z{:}\texttt{R(specification(}gs\texttt{))}^T \\
\qquad\qquad \texttt{I(specification(}gs\texttt{))}\ x \\
\qquad\qquad \Rightarrow \texttt{O(specification(}gs\texttt{))}\ \texttt{<}x\texttt{,}z\texttt{>} \\
\qquad\qquad \Rightarrow \texttt{sat(}gs\texttt{)}\ \texttt{<}z\texttt{,s0(}gs\texttt{)(}x\texttt{)>}
\end{array}
$$

Die Funktionen O und sat ergeben Propositionen.

gs_fulfils_axiom_4:

$$
\begin{array}{l}
\texttt{gs\_fulfils\_axiom\_4}(gs) \\
\quad \equiv \forall x{:}\texttt{D(specification(}gs\texttt{))}^T \\
\qquad\quad \forall s{:}\texttt{S(}gs\texttt{)}^T \\
\qquad\qquad \forall z{:}\texttt{R(specification(}gs\texttt{))}^T \\
\qquad\qquad\quad \texttt{I(specification(}gs\texttt{))}\ x \\
\qquad\qquad\quad \Rightarrow \texttt{J(}gs\texttt{)<}x\texttt{,}s\texttt{>} \\
\qquad\qquad\quad \Rightarrow \texttt{O(specification(}gs\texttt{))}\ \texttt{<}x\texttt{,}z\texttt{>} \\
\qquad\qquad\quad \Rightarrow (\texttt{sat(}gs\texttt{)}\ \texttt{<}z\texttt{,}s\texttt{>} \\
\qquad\qquad\qquad \Longleftrightarrow (\exists k{:}\mathbf{N} \\
\qquad\qquad\qquad\qquad \exists t{:}\texttt{S(}gs\texttt{)}^T \\
\qquad\qquad\qquad\qquad\ \lceil t{:}\texttt{S(}gs\texttt{)} \in (\texttt{split(}gs\texttt{)} \uparrow k\texttt{)<}x\texttt{,}s\texttt{>}\rceil \\
\qquad\qquad\qquad\qquad\ \Rightarrow \lceil z{:}\texttt{R(specification(}gs\texttt{))} \in \texttt{ext(}gs\texttt{)}\ t\rceil))
\end{array}
$$

`ext` dagegen ist eine Funktion, die wieder eine endliche Menge ergibt, diesmal allerdings eine Menge von Ergebnissen vom Typ `R`. Man beachte, daß in diesem Axiom die k-te Folgedeskriptormenge bestimmt wird.

Nun fehlt nur noch das Wohlfundiertheitsaxiom, das ja gesondert definiert wird. Die `empty?`-Operation ergibt „wahr", falls die angegebene endliche Menge leer ist. Dieses Axiom ist deshalb separiert, weil es das einzige ist, das im weiteren Verlauf so modifiziert werden wird, daß die gefilterte `split`-Funktion eingesetzt wird; siehe weiter unten.

gs_fulfils_wf_axioms:

$$\begin{array}{l} (gs\ \texttt{fulfils wf axioms}) \\ \equiv \forall x{:}\texttt{D(specification(}gs\texttt{))}^T \\ \quad \forall s{:}\texttt{S(}gs\texttt{)}^T \\ \quad\ \texttt{I(specification(}gs\texttt{))}\ x \\ \quad\ \Rightarrow \texttt{J(}gs\texttt{)}{<}x,s{>} \\ \quad\ \Rightarrow (\exists k{:}\mathbf{N}.\ \lceil \texttt{empty?((split(}gs\texttt{)}\ \uparrow\ k\texttt{)}{<}x,s{>}\texttt{)}\rceil) \end{array}$$

Es bleibt noch, das `odec_problem` zu präzisieren, welches die entscheidbare Version der Ausgabebedingung O spezifiziert. Im Augenblick findet man in der Bibliothek folgende Definition:

odec_problem:

$$\begin{array}{l} \texttt{odec\_problem(}spec\texttt{)} \\ \equiv {<}\texttt{D(}spec\texttt{)} \times \texttt{R(}spec\texttt{)},\mathbf{B}\!\uparrow,\texttt{always},\lambda{<}x,y{>}.\lceil y\rceil \Longleftrightarrow \texttt{O(}spec\texttt{)}\ x{>} \end{array}$$

Es ist allerdings ein Unding, Spezifikationen als 4-Tupel anzugeben. Wir geben daher noch eine ordentliche Notation für die Spezifikation:

$$\begin{array}{l} \texttt{FUNCTION}\ \mathrm{Obool}(\langle x,z\rangle{:}\,\texttt{D(}spec\texttt{)} \times \texttt{R(}spec\texttt{)}){:}\,\mathbf{B} \\ \texttt{RETURNS}\ (y \mid \lceil y\rceil \iff \texttt{O(}spec\texttt{)}\,\langle x,z\rangle) \end{array}$$

Dies sagt nun aus, das eine Funktion existiert, die für jedes Paar $\langle x, z\rangle$ einer Eingabe und einer Ausgabe berechnet, ob dieses Paar die Ausgabebedingung `O` erfüllt oder nicht.

Programmschema der einfachen Globalsuche

In obigem Theorem wird das Programm mit Hilfe von `filterless_body` generiert. Das so erhaltene Programm erfüllt die Mengenspezifikation `specification(`gs`)*`, die ja ausgehend von der einfachen, durch *D, R, I, O* ausgedrückten Spezifikation dasjenige Problem meint, welches die *Menge* aller Werte, die die einfache Spezifikation erfüllen, berechnet. Dieses Mengenproblem wird nun, falls die in dem Theorem angegebenen Voraussetzungen gegeben sind, durch folgendes Programm erfüllt (vgl. mit dem Algorithmus in Kapitel 1):

filterless_body:

$$\begin{array}{l} \texttt{filterless\_body(}gs; \mathit{Obool}\texttt{)} \\ \equiv\ \texttt{letrec}\ \mathit{search}(x)\ = \\ \qquad \texttt{letrec}\ h(\texttt{xs})\ = \\ \qquad\quad \texttt{let}\ x,s\ =\ xs \\ \qquad\quad \texttt{in}\ \ \{\ z \mid z \in \texttt{ext}(gs)\,s\ \ \wedge \mathit{Obool}{<}x,z{>}\ \} \\ \qquad\qquad \cup \bigcup\ \{\ h{<}x,t{>} \mid t \in \texttt{split}(gs){<}x,s{>}\ \} \\ \qquad ({<}x, \texttt{s0}(gs)(x){>}) \end{array}$$

Der gefilterte Globalsuchraum

Der Einbau eines Filters gestaltet sich nun recht einfach. Das durch *gs* beschriebene Modell, in dem der Filter bisher unberücksichtigt blieb, wird nun so abgewandelt, das der Filter in die `split`-Funktion Eingang findet:

filtered_splitting:

```
filtered_splitting(gs)
 ≡ <specification(gs)
   ,S(gs)
   ,J(gs)
   ,s0(gs)
   ,sat(gs)
   ,letrec h(xs) = let x,s = xs
                   in {t
                      | t∈split(gs)<x,s>
                      ∧ Phi(gs)<x,t>
                      }
   ,ext(gs)
   ,letrec h(xs) = t
   >
```

Dies ist die einzige Stelle, an der die Filterkomponente *Phi* des *gs*-Tupels interpretiert wird. Ziel ist nun, das durch `filtered_splitting` transformierte *gs* möglichst wieder als einfachen Suchraum zu behandeln. Dies gelingt im wesentlichen, als Ausnahme erhält man jedoch die Besonderheit, daß an den Filter eine eigene Notwendigkeitsbedingung gestellt werden muß.

Bevor wir jedoch diese Zusatzbedingung formalisieren, zunächst das modifizierte Theorem.

Das Theorem der gefilterten Globalsuche

Das Theorem, das die Filterung berücksichtigt, sieht sehr ähnlich zu dem obigen aus. Die beiden Axiome sind jedoch andere; und auch die Generierung erfolgt durch ein anderes Programm.

gs_synth_wf_filtered:

```
∀gs:GS
 (gs describes global search)
 ⇒ (gs has necessary filter)
 ⇒ SCHEME
     ∀
       O SATISFIES odec_problem(specification(gs))
     ⇒
       filtered_body(gs;O) SATISFIES (specification(gs)*)
   END
```

Man beachte, daß für *gs* die gleichen Globalsuchräume eingesetzt werden wie zuvor, nur daß nun auch die Filterkomponente interpretiert wird.

Die Filterung bedeutet auf der Seite der Generierung einfach, daß die modifizierte `split`-Funktion zum Durchlaufen des Suchraums benutzt wird; formal:

filtered_body:

```
filtered_body(gs;Obool)
 ≡ filterless_body(filtered_splitting(gs);Obool)
```

Die Axiome der gefilterten Globalsuche

Auf der Seite der Axiome sind die Änderungen eigentlich bedeutender. Die beiden Axiomgruppen werden nun durch folgende Definition zusammengefaßt:

gs_describes_global_search:
$(gs$ `describes global search`$)$
$\equiv (gs$ `fulfils search axioms`$) \wedge (gs$ `has wf filter`$)$

gs_has_wf_filter:
$(gs$ `has wf filter`$)$
$\equiv ($`filtered_splitting`(gs) `fulfils wf axioms`$)$

Die Änderung beschränkt sich also in der Tat auf das Wohlfundiertheitsaxiom, bei dem man die Modifikation der `split`-Funktion berücksichtigen muß. Die anderen Axiome können übernommen werden (dies wird durch ein Lemma gerechtfertigt, s.u.).

Die Filterung muß die weitere Voraussetzung machen, daß der Filter *notwendig* ist, was durch folgendes ausgedrückt wird:

gs_has_necessary_filter:
$(gs$ `has necessary filter`$)$
$\equiv \forall x$`:D(specification(`gs`))`^{T}. $(gs$ `has filter which is necessary at` $x)$

gs_has_nec_filter_at:
gs `has filter which is necessary at` x
$\equiv \forall s$`:S(`gs`)`^{T}
`I(specification(`gs`))` x
$\Rightarrow (\exists z$`:R(specification(`gs`))`^{T}
`sat(`gs`)` $<z,s> \wedge$ `O(specification(`gs`))` $<x,z>)$
$\Rightarrow \lceil$`Phi(`gs`)`$<x,s>\rceil$

Die Separation dieser Definition in zwei Teile hat wiederum Gründe, die erst weiter hinten eine Rolle spielen. Die Notwendigkeitsbedingung relativ zur Eingabe x benötigt man, wenn man die Spezialisierungsbedingung als Spezifikation formulieren will, siehe unten.

Rückführung der gefilterten Globalsuche auf die einfache

Um das Theorem mit Filterung auf das einfache Theorem zurückzuführen, reicht es aus, daß man es so umschreiben kann, daß überall dort, wo im einfachen Theorem gs steht, nun `filtered_splitting`(gs) steht. Da der Operator `filtered_splitting` den Spezifikationsanteil von gs unverändert läßt (was wir uns erspart haben, als Lemma zu formulieren), ist das einzige Problem dabei, diesen Schluß bezüglich der Suchaxiome durchzuführen. Folgendes Lemma drückt aus, was wir wollen:

filtered_gs:
$\forall gs$`:GS`
$(gs$ `has necessary filter`$)$
$\Rightarrow (gs$ `fulfils search axioms`$)$
$\Rightarrow ($`filtered_splitting`(gs) `fulfils search axioms`$)$

Dies ist der einzige Satz, den man zur Rückführung benötigt. Er ist zugleich die Stelle, an der die zusätzliche Voraussetzung von der Notwendigkeit des Filters eingeht. Genauer betrachtet, sind die

Axiome 1 und 3 ohnehin übertragbar, da in ihnen kein `split`-Aufruf vorkommt. In Axiom 4 sehen wir, daß `split` nur auf der linken Seite eines ⇒-Zeichens vorkommt; die Verkleinerung des Suchraums ist daher ebenfalls unproblematisch. Lediglich bei Axiom 2 ist die Sache komplizierter; man benötigt nun tatsächlich die Notwendigkeit des Filters. Wir gehen auf die Einzelheiten nicht weiter ein.

Damit haben wir bereits eine Algorithmentheorie für die einfache und die gefilterte Globalsuche. Wir müssen nur die Sätze *gs_synth_filterless* oder *gs_synth_wf_filtered* anwenden, wobei wir eine Instanz *gs* einsetzen. Diese Instanz beschreibt den Suchraum, und wir müssen jeweils zeigen, daß diese Instanz auch die Axiome, die ein Suchraum erfüllen muß, einhält. Ergebnis der Anwendung ist ein Algorithmus, der diesen Suchraum durchkämmt. Freilich müssen wir auf dieser Stufe der Entwicklung jeweils spezielle Suchräume einsetzen, die für das zu lösende Problem adäquat sind. Wir sind noch nicht so weit, Standard-Suchräume zu benutzen, die an das konkrete Problem lediglich angepaßt werden brauchen. Diese Verfeinerung ergibt sich erst unter Benutzung der Spezialisierung, die wir nun formalisieren wollen.

4.1.2 Spezialisierte Globalsuche

Das Synthese-Theorem sieht unter Einbeziehung der Spezialisierung nun so aus:

gs_synth_specl:
```
     ∀gs:GS
      ∀spec:SPECS
       (gs describes global search)
       ⇒ (R(spec)=R(specification(gs)))
       ⇒ SCHEME
            ∀
              O SATISFIES odec_problem(spec)
              theta SATISFIES FUNCTION theta WHICH spec
                                   << specification(gs)
                                   AND WHICH guarantees necessity for Phi(gs)

            ⇒
              filtered_body(gs specialized to spec using theta
              ;O)
               SATISFIES (spec*)
           END
```

Es wird nun nicht mehr ein Programm für die in *gs* enthaltene Spezifikation synthetisiert, sondern für das gesondert anzugebende Problem *spec*. Der Suchraum und die zu lösende Spezifikation müssen nun nicht mehr identisch sein; allerdings muß sich *spec* auf die Spezifikation des Suchraums reduzieren lassen.

Wie man sieht, kommt im Programmschema nun ein neues Programm *theta* vor, was zunächst geschrieben sein will, damit die Generierung funktioniert. Bei *theta* handelt es sich um die Spezialisierungstransformation, also die Abbildung der Eingaben der Suchraum-Spezifikation auf die Eingaben von *spec*. Da *theta* berechenbar sein muß, wurden alle Bedingungen, die an *theta* zu stellen sind, in einer speziellen Spezifikation `FUNCTION theta` gebündelt. Dabei handelt es sich um eine allgemeine Voraussetzung zur Spezialisierung (*spec* ≪ `specification`(*gs*)), sowie um eine modifizierte Formulierung der Filternotwendigkeit.

Das generierte Programm wird wiederum mit Hilfe von `filtered_body` konstruiert, jedoch wird nicht der Suchraum *gs* selbst, sondern der spezialisierte Suchraum eingesetzt, also derjenige Suchraum, der an das Problem *spec* angepaßt ist.

Transformation des Suchraums durch Spezialisierung

Der spezialisierte Suchraum wird dadurch berechnet, daß in die Funktionen, die vom Definitionsbereich `D(`*gs*`)` aus abbilden, die Transformation *theta* eingesetzt wird.

gs_specl:

```
gs specialized to spec using theta
 ≡ <spec
   ,S(gs)
   ,J(gs) | theta
   ,s0(gs) | theta
   ,sat(gs)
   ,split(gs) | theta
   ,ext(gs)
   ,Phi(gs) | theta
   >
```

gs_specl_1:

```
f | theta
 ≡ letrec h(x) = f(theta(x))
```

gs_specl_2:

```
f | theta
 ≡ letrec h(xy) = let x,y = xy in f<theta(x),y>
```

Dies ist auch im Grunde alles, was man unternehmen muß, damit man, wenn man die Funktion *theta* bereits kennt, ein spezialisiertes Programm generieren kann. Es bleibt noch die wichtigere Frage zu klären, welche Bedingungen denn *theta* überhaupt erfüllen muß.

Die Spezialisierungsbedingung als Spezifikation

Die Spezifikation von *theta* wird durch folgendes beschrieben:

specl_problem_with_necessity:

```
FUNCTION theta WHICH spec
<< specification(gs)
AND WHICH guarantees necessity for Phi(gs)
 ≡ FUNCTION theta(x:D(spec)):D(specification(gs))
   WHERE I(spec) x
   RETURNS (y | Specl_cond(spec;specification(gs)) x y
              ∧ gs
                specialized to spec
                has nec filter at x,y)
```

Wir sehen, daß *theta* eine partielle Funktion ist, die vom *spec*-Definitionsbereich zum *gs*-Definitionsbereich abbildet. Desweiteren muß die Abbildung zwei Bedingungen erfüllen, von der eine mittels `Specl_cond` ausgedrückt wird. Dieses Prädikat wurde bereits in Abschnitt 2.6.3, Seite 90, definiert und beschreibt die generelle Spezialisierbarkeit *spec* $\ll$ `specification(`*gs*`)`.

Die andere Bedingung ist die Einbettung der Filternotwendigkeit in den Rahmen der spezifikatorischen Spezialisierungsauffassung. Die Bedingung entspricht im wesentlichen der bereits bekannten Notwendigkeit:

gs_specld_has_nec_filter:

```
gs specialized to spec has nec filter at x,y
 ≡ ∀s:S(gs)^T
    I(spec) x
    ⇒ (∃z:R(spec)^T. sat(gs) <z,s> ∧ O(spec) <x,z>)
    ⇒ ⌈Phi(gs)<y,s>⌉
```

Die Spezifikation von *theta* ist äquivalent einer etwas freieren Beschreibung, die zudem die schon eingeführten Abstraktionen aufnimmt, was durch folgendes Lemma ausgedrückt wird:

specl_problem_with_necessity_ch:

```
∀gs:GS
 ∀spec:SPECS
  (R(spec)=R(specification(gs)))
  ⇒ (FUNCTION theta WHICH spec
     << specification(gs)
     AND WHICH guarantees necessity for Phi(gs)
     is satisfiable
     ⇐⇒ (∃theta:spec to specification(gs)
           (spec << specification(gs)with theta)
           ∧ (gs specialized to spec using theta has necessary filter)))
```

Man benötigt dieses Lemma, um die spezialisierte Globalsuche auf die gefilterte zurückzuführen; im wesentlichen passiert nicht mehr, als die Besonderheiten, die durch die Rahmung als Spezifikation notwendig geworden sind, zurückzudrängen und mit einfacheren Mitteln dasselben auszudrücken.

Rückführung der spezialisierten Globalsuche auf die gefilterte

Wir wollen hier nur die beiden wesentlichen Lemmata angeben, die für die Rückführung erforderlich sind. Die Details lassen wir wieder aus.

gs_search_axioms_specl:

```
∀gs:GS
 ∀spec:SPECS
  ∀theta:spec to specification(gs)
   (gs fulfils search axioms)
   ⇒ (spec << specification(gs)with theta)
   ⇒ (gs specialized to spec using theta fulfils search axioms)
```

gs_description_specl:

```
∀gs:GS
 ∀spec:SPECS
  ∀theta:spec to specification(gs)
   (gs describes global search)
   ⇒ (spec << specification(gs)with theta)
   ⇒ (gs specialized to spec using theta describes global search)
```

4.2 Operationalisierung der Globalsuche

Nachdem nun der Rahmen für eine formale Theorie vorgegeben ist (ihre Ausführung als Beweis der angegebenen Theoreme müßte natürlich noch erfolgen, sprengt aber den Umfang dieser Arbeit bei weitem), ist noch zu klären, auf welche Weise die Theoreme angewendet werden. Die Theoreme, oder besser, die Beweise, die konstruiert werden, um diese Theoreme anzuwenden, werden damit zum Objekt einer höheren Manipulationsebene, die in NuPRL durch die Metasprache ML repräsentiert wird. In ML sind alle Bestandteile der Typentheorie als ML-Datentypen dargestellt. Dies eröffnet, daß

- jene Beweise, die die Globalsuch-Theoreme auf konkrete Instanzen anwenden, auf beliebige algorithmische Weise erzeugt werden können und diese Algorithmen (Taktiken) in Form von ML-Programmen direkt in der NuPRL-Umgebung ablaufen können, sowie daß
- ein höherer Begriff des Beweises möglich wird, in dem nicht nur die Primitiven der Typentheorie erlaubt sind, sondern ebenso taktische Regeln, sofern sich jede Instanz einer taktischen Regel auf einen korrekten primitiven Beweis abstützen kann.

Die Korrektheit von Taktiken oder taktischen Regeln wird durch eine besondere Kapselung des Datentyps „Beweis“ (`proof`) sichergestellt. Die technischen Einzelheiten sollen uns hier nicht weiter interessieren; es ist jedenfalls Vorsorge getragen, daß Taktiken nur solche Beweise erzeugen können, die von einem NuPRL-Modul auf Korrektheit geprüft sind, und daß, wer diesem Modul mißtraut, auch jederzeit einen Beweis in Termen der primitiven Regeln ausgeben und anderweitig prüfen kann. Diese Möglichkeiten der Gegenprüfung entlasten uns als Taktik-Designer, da wir nicht verpflichtet sind, die Taktiken selbst einem penibel Korrektheitsbeweis zu unterziehen.

Ausgangspunkt der Globalsuch-Taktik, die wir nun entwerfen, ist eine Behauptung der Form:

(1) $\vdash$ *sp* `is satisfiable`

Dabei ist *sp* eine Metavariable, die für die Spezifikation steht, deren algorithmische Lösung wir anstreben. Werfen wir nun einen Blick auf das Theorem *gs_synth_specl*, welches wir zu benutzen gedenken. Es ist zunächst zu klären, auf welche Weise die allquantifizierten Variablen *gs* und *spec* zu instanziieren sind; desweiteren muß ggf. die Behauptung auf eine bestimmte Form gebracht werden, um das Theorem anzuwenden; schließlich sind die Voraussetzungen zu untersuchen, die das Theorem macht und die natürlich bei einer Anwendung gegeben sein müssen. Die spezielle Form typentheoretischer Beweise als eines Baumes, dessen Wurzel Konsequenz seiner Söhne ist, führt dazu, daß wir den Beweis ausgehend von der Behauptung in Richtung von Voraussetzungen konstruieren, die die Behauptung evident machen. In diesem Sinne denken wir uns die Behauptung als Wurzel, zu der wir nun eine evidente Voraussetzung finden müssen, die Konsequenz des Globalsuch-Theorems ist, welche Anwendung uns wiederum Voraussetzungen nennt, die noch zu prüfen sind.

Algorithmentheorien haben nun gegenüber anderen Synthesestrategien den Vorteil, daß bei auf ihnen basierenden Beweisen, gedacht als Voraussetzungsanalysen, die Entscheidungen, die die Beweisrichtung relevant bestimmen, sehr weit oben im Beweisbaum geschehen. Die Formierung der Behauptung, also deren Umformung, damit man überhaupt ein Theorem anwenden kann, ist ein sehr kleiner und eher unbedeutender Schritt. Es kommt vielmehr auf die richtige Instanziierung des Theorems an (hier vor allem *gs*), damit man zu weiteren Voraussetzungen kommt, die beweisbar sind. Diese Voraussetzungen, die im Beweisbaum also die unmittelbaren Söhne der Theoremanwendung sind, sind Formeln von einer Art, die eine konkrete Richtung vorgeben, was die weitere Analyse betrifft.

Wie man sieht, folgt die Beweiskonstruktion einem bestimmten Schema, das an jeder Stelle die Gestalt des Beweises in bestimmter, häufig beinah durchgängiger Weise fixiert, aber jeweilig Spielräume läßt, die eine Anpassung an die konkreten Gegegebenheiten erlaubt. Wir werden nun einen Vorschlag für ein solches Schema für das Theorem *gs_synth_specl* vorstellen, wobei wir jeweils die fixen und variablen Anteile besonders herausarbeiten. Wir stellen uns natürlich die Frage, ob die einzelnen Schritte automatisierbar sind; insgesamt also, ob es eine Taktik gibt, die dem Schema entspricht. Das Verhältnis von festen und veränderlichen Anteilen ist dafür besonders wichtig, weil feste Vorgaben eine hohe algorithmische Nutzbarkeit mit sich bringen und Variabilitäten das Schema trotzdem so elastisch machen, daß es auf eine Vielzahl von Problemen paßt.

Bevor wir zu den Details eines solchen Schemas kommen, noch die Bemerkung, daß die aktuelle Implementierung noch nicht so weit ist, das Schema von einer Taktik vollständig konstruieren zu lassen. Viele Teile des Schemas werden aber bereits taktisch aufgebaut; die Auswahl der passenden Teile ist jedoch manchmal problematisch. Das Schema muß an bestimmten Stellen eben *gesteuert* werden; hierin drückt sich seine Modifizierbarkeit aus, die zur Anpassung an konkrete Probleme unerläßlich ist. Wir werden weiter unten diese offene Frage der Steuerung noch etwas weiter diskutieren.

Den Ausgangspunkt (1) fassen wir zunächst noch etwas anders:

(2) $\vdash$ $sp*$ `is satisfiable`

Dadurch können wir im weiteren die punktweise Spezifikation des Globalsuchproblems unter *sp* ansprechen, während die mengenwertige Spezifikation als $sp*$ verfügbar ist. Desweiteren paßt so der Ausgangspunkt genau auf das Globalsuchtheorem *gs_synth_specl*, was wir im folgenden ausnutzen werden. (Liegt das Problem dennoch in der Form (1) vor, müßten wir es in einem Vorverarbeitungsschritt nach (2) umformen; wie dies funktionieren könnte, sei dahingestellt.)

Wir kommen nun direkt zur Anwendung des Globalsuchtheorems. Dessen beiden Variablen *gs* und *spec* sind zu instanziieren. Für *spec* wird nun unmittelbar *sp* eingesetzt, da so die Konsequenz des Theorems und die Ausgangsformel übereinstimmen. Die Instanz gs_0 für *gs* ist dagegen vom Benutzer zu bestimmen. (Eine automatische Auswahl der Instanz ist, zumindest im Augenblick, nicht empfehlenswert, da dies die wesentlichste Richtungsentscheidung der Synthese ist. Das Wissen des Anwenders über die Art des zu lösenden Problems kann bei der Auswahl des Suchraums in prägnanter Weise genutzt werden, um in einem frühen Stadium der Synthese ihre Richtung festzulegen und damit die Tragweite der weiteren, untergeordneten Entscheidungen zu begrenzen. Die Chancen der Automatisierung dieser Unterentscheidungen steigen gerade wegen der frühen Interaktion mit dem Benutzer. Für eine weitere Diskussion dieses interessanten Kooperationsansatzes zwischen Anwender und Maschnine verweisen wir auf das fünfte Kapitel.) Die Taktik InstLemma ruft ein Lemma mit gegebenen Instanzen auf und separiert, sofern möglich, die Voraussetzungen einer universellen Formel von der Konsequenz. Wir nutzen sie, um das Globalsuchtheorem anzuwenden:

$\vdash$ $sp*$ `is satisfiable`
| BY InstLemma 'gs_synth_specl' [gs_0; sp]

  $\vdash$ gs_0 `describes global search`

  $\vdash$ `R(`sp`) = R(specification(`gs_0`))`

(3) 1. `SCHEME`
  O `SATISFIES odec_problem(`sp`)`
  theta `SATISFIES` $\langle$ *theta-Spezifikation* $\rangle$
 $\Rightarrow$
  `filtered_body(`gs_0 `spezialized to` sp `using` *theta*`;` O`)`
   `SATISFIES` $sp*$
 `END`
 $\vdash$ $sp*$ `is satisfiable`

Wie man sieht, werden zwei Voraussetzungen des Globalsuchtheorems abgetrennt und in eigenen Untersequenzen zurückgegeben; die Konsequenz, die hier das Programmschema ist, kann als Hypothese zum Beweis der Ausgangsformel benutzt werden. Das Unterziel, welches behauptet, gs_0 sei ein Global-Suchraum, kann man, falls diese Instanz ein Standard-Suchraum ist, durch Verweis auf ein Lemma der Wissensbank beweisen. Die andere Voraussetzung, daß die Typen der Wertebereiche von dem Suchraum und der Spezifikation übereinstimmen, ist relativ leicht zu zeigen, indem die Gleichung soweit vereinfacht wird, daß eine Identität $R = R$ vorliegt.

Im dritten Unterziel geht der Hauptstrang des Beweisschemas weiter. In der Hypothese befindet sich das Programmschema, dessen Folgerung besagt, daß der `filtered_body`-Term ein Programm ist, das das gestellte Problem erfüllt. Die Konklusion der Sequenz behauptet, daß es ein Programm gibt, welches das Problem löst. Unter Benutzung des Programmschemas kann diese Behauptung eingelöst werden, und zwar mit dem genannten Lösungsprogramm. Allerdings stellt das Programmschema Bedingungen an die Variablen O und *theta*, für die zunächst Programme gefunden werden müssen, damit das `filtered_body`-Programm das gestellte Problem erfüllt. Im Beweisschema sieht diese Analyse des Programmschemas wie folgt aus:

(4)

1. SCHEME
   O SATISFIES odec_problem(sp)
   theta SATISFIES ⟨ *theta-Spezifikation* ⟩
   $\Rightarrow$
   filtered_body(gs_0 spezialized to sp using *theta*; O)
   SATISFIES $sp*$
   END
$\vdash$ $sp*$ is satisfiable
BY *geeignete Taktik*
  $\vdash$ odec_problem(sp) is satisfiable
  $\vdash$ ⟨ *theta-Spezifikation* ⟩ is satisfiable

Man beachte, daß hierbei die konstruktive Deutung des Existenzquantors, der ja in den `is satisfiable`-Abstraktionen enthalten ist, vorausgesetzt wird. Dadurch kann ein Beweis für eine solche Existenz, hier also für eines der Unterziele, als ein Konstruktionsverfahren für ein konkretes Objekt, das die Existenz erfüllt, angesehen werden. So ist immer gewährleistet, daß es ein konkretes Objekt für O und für *theta* gibt, weil die Beweise der Unterziele solche Objekte angeben müssen. Die Typentheorie enthält den Mechanismus der Extraktion, mit dem die Benutzung der in einem Beweis konstruierten Objekte nachvollzogen werden kann. Mit Hilfe der Extraktion können so, falls die Unterziele vollständig bewiesen sind, die konkreten Terme für O und *theta* bestimmt werden; im weiteren auch der komplette Term, der das gestellte Problem löst.

Wir wollen an dieser Stelle nicht weiter analysieren, wie man die Behauptung, es gebe eine entscheidbare Form O der Ausgabebedingung, im einzelnen beweist. Wichtiger ist, wie man an die *theta*-Transformation kommt. Dazu analysieren wir die Spezifikation für *theta*, die wir bislang noch gar nicht als formalen Term notiert hatten.

$\vdash$ `FUNCTION theta`
`WHICH` $sp \ll$ `specification(`gs_0`)`
`AND WHICH guarantees necessity for Phi(`gs_0`)`
`is satisfiable`

(5) BY *geeignete Taktik*

$\vdash\ \exists\, theta{:}\langle\ theta\text{-}Typ\ \rangle.$
$\langle$ `FUNCTION theta ...`, $theta\ \rangle$ `terminates`
$\wedge\ \langle$ `FUNCTION theta ...`, $theta\ \rangle$ `is partially correct`

Soll ein Term *theta* gefunden werden, der die theta-Spezifikation erfüllt, sind zwei Aspekte zu berücksichtigen: Daß er auf allen zulässigen Eingaben terminiert und daß er für alle zulässigen Eingaben korrekte Ergebnisse berechnet (partielle Korrektheit).

Wir haben an dieser Stelle zwei Möglichkeiten: Entweder wir lassen den Existenzquantor stehen, falten die Abstraktionen für Terminierung und partielle Korrektheit auf und führen weitere Schlüsse durch, bis wir ein Instanz für *theta* an der Formel ablesen können, oder wir haben bereits eine Ahnung, welcher Term für *theta* eingesetzt werden muß und führen diese Instanz schon jetzt ein. Ersteres ist angebracht, wenn wir eine Methode haben, die die geeigneten Schlüsse automatisch findet; da wir noch nicht soweit sind, bleibt uns zunächst nichts anderes übrig, als unser eigenes Wissen zu nutzen und den *theta*-Term anzugeben. Wir nehmen im folgenden an, wir wüßten, daß *th* die richtige Instanz ist, und hätten diese schon eingesetzt. Desweiteren interessiert uns im Augenblick nicht so sehr die Terminierung, sondern die partielle Korrektheit. Wir betrachten deshalb im weiteren nur, wie dieser Teil der Behauptung bewiesen werden kann.

Faltet man die beteiligten Abstraktionen auf, nimmt die Behauptung der partiellen Korrektheit folgende Form an:

$$
\begin{array}{lll}
(6) & \vdash\ \forall x{:}\mathtt{D}(sp). & \\
& \quad \mathtt{I}(sp)(x) & \\
& \quad \Rightarrow th\ \mathtt{halts\ on}\ x & \\
& \quad \Rightarrow \mathtt{I}(\mathtt{specification}(gs_0))(th(x)) & \text{(a)} \\
& \qquad \wedge\ \forall y{:}\mathtt{R}(sp).\ \ O(sp)(x,y) \Rightarrow O(\mathtt{specification}(gs_0))(th(x),y) & \text{(b)} \\
& \qquad \wedge\ \forall s{:}\mathtt{S}(gs_0). & \\
& \qquad\quad \mathtt{I}(sp)(x) & \text{(c)} \\
& \qquad\quad \Rightarrow \exists z{:}\mathtt{R}(sp).\ \ \mathtt{sat}(gs_0)(s,z) \wedge O(sp)(x,z) & \\
& \qquad\quad \Rightarrow \mathtt{Phi}(gs_0)(th(x),s) &
\end{array}
$$

Diese Formel kann man wiederum in drei Einzelbehauptungen zerlegen, hier als (a), (b) und (c) markiert. (a) besagt, daß die Transformation eingabeseitig funktioniert, (b) ausgabeseitig. Diese Teile gehen direkt aus der Spezialisierungsbedingung hervor. Teil (c) ist die Notwendigkeitsbedingung des benutzten Filters.

An dieser Stelle endet das allgemeine Schema für den Erfüllbarkeitsbeweis, da die weiteren Verfeinerungen der drei Einzelbehauptungen nur möglich sind, wenn konkrete Instanzen für sp, gs_0 und th bekannt sind. Die äußere Form dieser Behauptungen steht jedoch fest. Insbesondere sehen wir, daß es sich jeweils um Implikationen handelt, was für die automatische Behandlung von Vorteil ist. Die Beweissuche kann nämlich zielorientiert vorgehen, also entweder vorwärts von der Voraussetzung zur Folgerung oder entsprechend rückwärts Zwischenschritte suchen. Als erfolgversprechende Technik bietet sich die in Kapitel 3 ausgeführte Termersetzung mit anschließendem Logikteil an. Die Details der Beweissuche konnten leider nicht zum Gegenstand gemacht werden, weil die Implementierung der Grundlagen nicht zuverlässig genug gemacht werden konnte.

Das hier vorgestellte allgemeine Schema für Globalsuchbeweise ist natürlich noch lange nicht perfekt. Zahlreiche Umstellungen sind möglich, etwa zur Optimierung, um ansonsten mehrfach vorkommende Sequenzen an einer Stelle zusammenzufassen, desweiteren zur besseren Automatisierbarkeit, man denke hier vor allem an die vom Himmel gefallene *theta*-Instanz, oder zur verbesserten Benutzbarkeit. Der Beweis sollte so strukturiert sein, daß die Punkte, an denen Entscheidungen vom Benutzer verlangt werden, an einer sinnvollen Stelle liegen. Dies betrifft nicht so sehr die dargestellten Entscheidungen, allerdings die hier nicht verfolgte Möglichkeit, zusätzliche Filter zur Effizienzsteigerung anzugeben. Im Augenblick müßte man diese Filter zusammen mit dem Suchraum eingeben, also an einer Stelle, an der man noch gar nicht absehen kann, welche Filter möglich und welche gut und welche schlecht sind. Besser wäre es, diese Festlegung analog O und *theta* nach hinten zu verschieben.

Wie wir sehen, haben wir es mit einem Beweis zu tun, der nicht bloß erbracht werden muß, sondern dessen Gelingen stark von seinem prozessualen Verlauf abhängt. Wir geben nicht nur ein Ziel vor, sondern gleich auch zahlreiche Zwischenstationen des Weges, einerseits um die Erreichbarkeit theoretisch wie praktisch zu erhöhen, andererseits aber auch, um Qualitätskriterien an das Produkt (etwa gute oder schlechte Filter) und an den Syntheseprozeß (Laufzeit, Erwartungen an den Benutzer usw.) stellen zu können.

Um die Implementierung der Globalsuche abzuschließen, müßte man nun eine Taktik ins Auge fassen, die einen Beweis, der diesem Schema folgt, konstruiert. Dabei können Teile nach einem festen Muster erzeugt werden oder durch Suche nach einer Lösung. Benutzerentscheidungen können prinzipiell eingebaut werden[2].

Ist der Beweis vollendet, interessiert uns natürlich das konkrete Programm, das dabei synthetisiert wurde. Die Typentheorie unterstützt dies mit Hilfe der Extraktterme: Das in einem Beweis benutzte Konstruktionsverfahren, um die Existenz einer Variablen zu erweisen, kann im nachhinein extrahiert werden. Wir interessieren uns natürlich für die Funktion, die das gestellte Problem berechnet. Eine Behauptung $\vdash sp*$ `is satisfiable` generiert dabei einen Extraktterm der Art $\langle \text{letrec } f(x) = t,\ \text{Ax}\rangle$ (ggf. nach einigen Reduktionsschritten). Das linke Element dieses Paares ist die gesuchte Funktion.

4.3 Architektur der implementierten Globalsuche

Wir sind nun an eine Position gelangt, von der aus deutlich wird, warum wir bestimmte Techniken untersucht haben und welche Bedeutung diesen für das Gesamtprojekt zugemessen werden kann. Der Aufbau dieser Arbeit ist dergestalt, daß mit den Grundlagen und ihrer Versicherung angefangen und zu immer höheren, zusammengesetzteren und komplexeren Phänomenen und Techniken aufgestiegen wurde. Wir betrachten nun die zurückgelegte Wegstrecke in umgekehrter Richtung und versuchen, von dem erreichten Niveau aus die Relevanz der grundlegenderen Schichten zu verstehen.

Die Formalisierung der Globalsuche und ihre Operationalisierung scheint zunächst recht abgehoben vom Rest zu sein; jedenfalls entfaltet sich ein ganz eigenständiger Bedeutungszusammenhang. Die zum Zuge kommenden Konzepte sind dabei stets sehr besondere und immer ganz konkret; die darauf basierenden Techniken sind nur für die spezielle Konzeptualisierung gemacht. Dies sind Kennzeichen einer auf eine bestimmte Anwendung zielenden Theoriebildung sowie einem Typus von Operationalisierung, deren Erfolg sich nicht an einer abstrakten Universalität, sondern nur an einer pragmatischen Benutzbarkeit mißt. Es geht also nicht darum, allgemeine Konzepte und Techniken zusammenzusetzen, sondern solche, die bereits auf einen bestimmten Zweck hin zugeschnitten sind und die dann als gelungen gelten, wenn es genügend Beispiele gibt, die die Zweckerfüllung demonstrieren können.

Ganz anders unsere Diskussion über die allgemeinen Techniken zur Bildung von Abstraktionen,

[2]Im derzeitigen NuPRL sind leider für Taktiken keine eigenen Interaktionsmöglichkeiten vorgesehen. Im Fall von Transformationstaktiken verstoßen solche Interaktionen jedoch nicht gegen Kapselungsgebote, so daß eigentlich nichts im Wege steht.

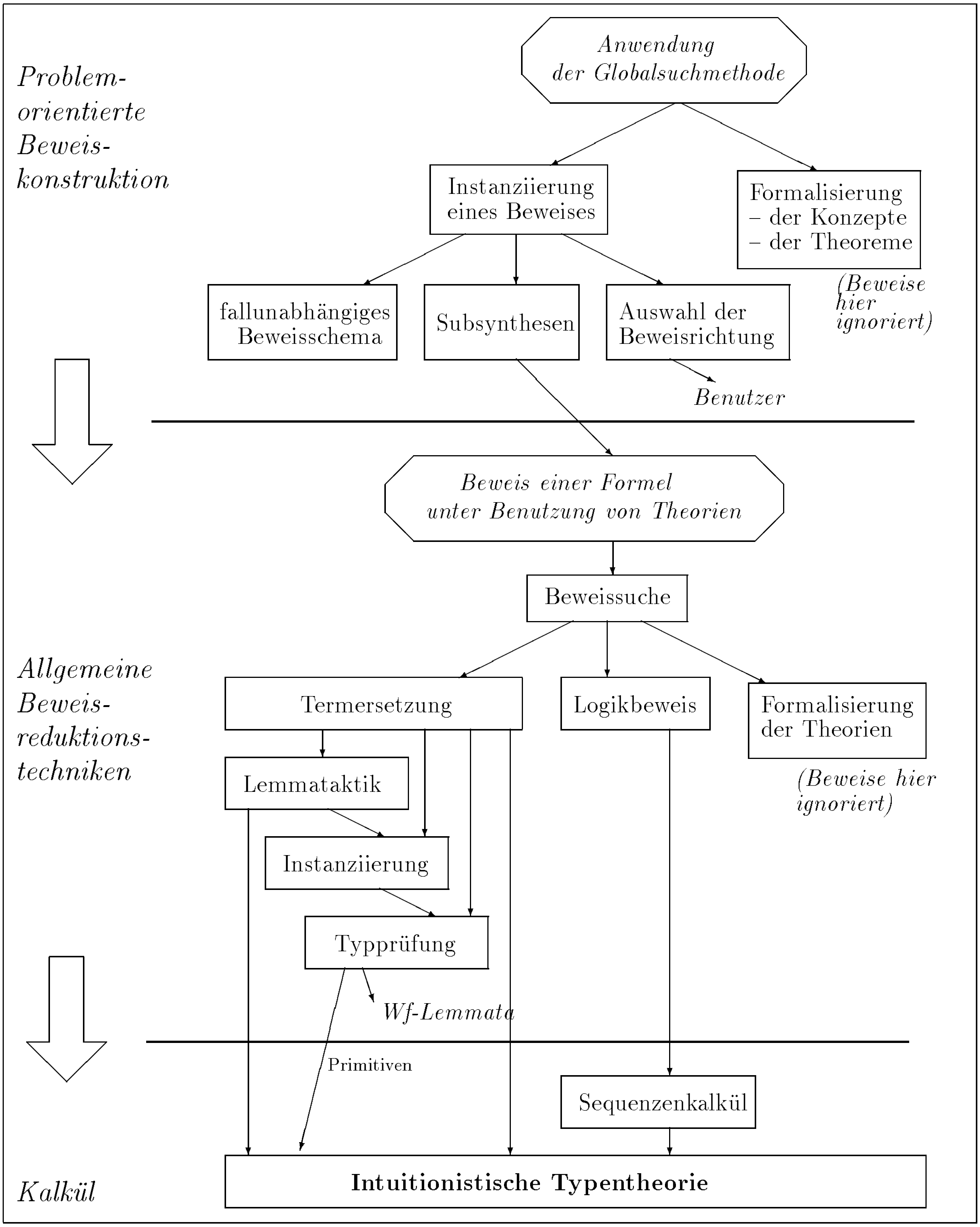


Abb. 4.1. *Die Reduktion der Globalsuche auf die Typentheorie*

über Typprüfung, abstrakte Datentypen, Termersetzung und logische Beweise: Diese Konzepte und Techniken sind weit weniger speziell, weil sie die Grundlage für viele mögliche Anwendungen bilden sollen. Die Frage der Universalität wird hier zumindest gestellt, und eine Antwort sollte zumindest den Bereich charakterisieren, innerhalb dessen diese Techniken zum Erfolg führen. Es soll einigermaßen berechenbar sein, ob in einer konkreten Situation die Benutzung einer der Techniken erfolgversprechend ist oder nicht. Natürlich sind diese Konzepte nicht völlig universell, sondern immer nur eingeschränkt; dies liegt daran, daß der ganze Umfang des in der Typentheorie Ausdrückbaren meistens zu groß ist, um dabei auftretende Probleme noch entscheiden zu können; folglich muß soweit begrenzt werden, daß die Problemlösung gewährleistet ist. Damit ergibt sich eine bereichsspezifische Universalität, die für diesen Typus der Konzeptionalisierung kennzeichnend ist.

Zuletzt kann man diesen beiden Typen der Begriffsbildung noch die Grundlage in Form des typentheoretischen Kalküls gegenüberstellen. Diese bildet den Maßstab des Ausdrückbaren und damit denjenigen der technischen Universalität. Er ist jedoch andererseits wieder bereits eine bestimmte Ausformung; gibt vor, was benennbar ist. Die Anwendung wie etwa die Globalsuchmethode ist aber nicht darauf fixiert, mittels gerade dieses Kalküls ausgedrückt zu werden; mithin ist der Horizont, vor der die Frage der Formalisierbarkeit einer Anwendung gestellt werden muß, nicht ein einzelner Kalkül, sondern die Gesamtheit der Kalküle, also genau genommen eine unbekannte Größe. Fragen wir nun danach, inwieweit in der Methode der Anwendung selbst und nicht in der spezifischen formalen Gestalt, die sie für einen Kalkül angenommen hat, zur Erfüllung des gestellten Zwecks geeignet ist, müssen wir vor einem offenen Horizont argumentieren. Es ist klar, daß es darauf keine letzte Antwort geben kann.

In Abbildung 4.1 sind die wichtigsten Konzepte der implementierten Globalsuche so angeordnet, daß nachvollzogen werden kann, welches Konzepts auf welchem aufbaut oder welche Konzept in ein anderes überführt werden kann. Die Abbildung ist dreigeteilt entsprechend den drei Typen der Konzpetualisierung.

Um nun die Methode der Globalsuche anzuwenden, benötigen wir zum einen eine Formalisierung der Abstraktionen und Theoreme. Wir haben in dieser Arbeit eine Ausprägung der Formalisierung dargestellt, jedoch es unterlassen, die Theoreme zu beweisen; dies müßte noch unternommen werden. Um die Beweise führen zu können, benötigt man mit Sicherheit sehr elaborierte allgemeine Beweistechniken, die von ihrem Status her dem entsprechen, was im mittleren Teil der Abbildung zu sehen ist. – Zum anderen muß man, um Globalsuche betreiben zu können, eine Vorstellung haben, wie die Theoreme angewendet werden, um einen Beweis für konkrete Problemstellungen zu erzeugen. Dabei gibt es einen fallunabhängigen Teil, der also in jeder Beweisinstanz gleich ausfällt, und einen spezifischen Teil. Dieser besteht darin, die Beweisrichtung zu steuern, etwa durch Auswahl brauchbarer Globalsuchräume durch den Benutzer, und darin, die anfallenden Subprobleme zu lösen. In unserem Fall sind dies die berechenbare Version von O sowie die Synthese der Transformation *theta*.

Für die Subprobleme haben wir nun einen Apparat allgemeiner Beweistechniken, die zum Einsatz kommen können. Ziel ist es dabei, eine Formel beweisen zu können, wobei vorgefertigtes Wissen angewendet werden soll. Dieses Wissen muß in formalisierten Theorien bereitstehen; diese lagen wieder außerhalb des Fokus' dieser Arbeit und wurden deshalb nicht behandelt. In Form der abstrakten Datentypen haben wir jedoch ein Strukturierungsprinzip vorgestellt. – Der Beweis muß gesucht werden; die Möglichkeiten der Beweiskonstruktion sollen also systematisch durchforstet werden. Wir konnten darauf leider nur am Rande eingehen, weil die unterliegenden Techniken noch nicht reif genug sind, um spezifische Suchansätze zu erproben. Zwei dieser Techniken kommen vor allem in betracht: Termersetzungen sowie einfache Sequenzenbeweise. Die Termersetzung basiert auf den grundlegenderen Techniken der Lemmaanwendung, der Instanziierung und der Typprüfung. Zu guter Letzt sind die Techniken genau dann erfolgreich, wenn es gelingt, die höheren Konzepte auf die Primitiven der Typentheorie zu reduzieren.

4.4 Ausführliches Beispiel: Die Costas-Arrays

Beim Costas-Arrays-Problem geht es darum, zu einer gegebenen Liste $L = [1; ...; x]$, $x \geq 1$, alle permutierten Listen L' zu finden, die eine bestimmte Bedingung hinsichtlich ihrer *Differenzentafel* erfüllen. Diese Tafel hat x Zeilen, die wiederum Listen sind; in der ersten Zeile D_1 befinden sich die Differenzen von in L' benachbarten Elementen; in der zweiten Zeile D_2 die Differenzen von in L' ein Element auseinander liegenden Elementen; usw. Etwas genauer: Ist

$$L' = [l_1; l_2; ...; l_x]$$

so ist die i-te Zeile der Differenzentafel definiert durch

$$D_i = [l_1 - l_{1+i};\ \ l_2 - l_{2+i};\ \dots ; l_{x-i} - l_{(x-i)+i}].$$

Eine Liste L' gehört nun zur Menge der gesuchten Listen, wenn in allen Zeilen der Differenzentafel kein Element mehrfach vorkommt.

Wir setzen nun den Hebel zur Synthese eines korrekten Programmes an, indem wir das Auffinden der Costas-Arrays als Globalsuchproblem begreifen. Zunächst unterscheiden wir das Punktproblem vom Mengenproblem:

- *Punktproblem:* Finde zu einem ganzzahligen x mit $x \geq 1$ eine Liste L', die Permutation der Liste $[1; ...; x]$ ist und für die die Zeilen der Differenzentafel keine mehrfachen Elemente enthalten.
- *Mengenproblem:* Finde zu einem ganzzahligen x mit $x \geq 1$ eine Menge $\{L' \mid L' \text{ löst das Punktproblem für } x\}$.

Die formalisierte Fassung des Punktproblems lautet (das Mengenproblem ist dann `costas_spec*`):

costas_spec:
```
costas_spec()
 ≡ FUNCTION Costas(n:Z↑):Seq(Z↑)
   WHERE n ≥ 1
   RETURNS (p | ⌈permutation(p:seq(Z↑);{1..n}:set(Z↑))
                ∧ (∀j∈{1..|p|}. nodups(dtrow(p;j):seq(Z↑)))
               ⌉)
```

costas_dtrow:
```
dtrow(L;j)
 ≡ [(L_(i)⁻) - (L_(i + j)⁻) | i∈[1..|L| - j]]
```

seq_permutation:
```
permutation(L:seq(alpha);S:set(alpha))
 ≡ nodups(L:seq(alpha)) ∧ (range(L) ≐ S : Set(alpha))
```

seq_nodups:
```
nodups(L:seq(alpha))
 ≡ ∀x·L'∈L. ¬(∃y∈L'. (x eq(alpha) y))
```

seq_range:
```
range(L)
 ≡ seqrec_case L of [] => {} | x·_ => R.R+x
```

Wie man sieht, benötigen wir eine Reihe von Operationen für den Listentyp `seq` und einige für den Mengentyp `set`, um das Problem ausdrücken zu können. Diese Operationen sind natürlich selbst kleine Programme; um ein Problem zu spezifizieren, für das ein Programm synthetisiert werden soll, müssen wir also bereits im Besitz von Programmen sein. Syntheseverfahren fangen in der Regel nicht bei Null an, sondern setzen voraus, daß grundlegendere Programme bereits erzeugt worden sind.

Neben den in den Tabellen 2.7 (Seite 64) bis 2.11 aufgeführten Operationen werden hier noch $\{x..y\}$ für das Intervall von x bis y als Menge, $[x..y]$ für das Intervall als Liste, $|L|$ für die Länge der Liste L, $L\_(i)^-$ für das i-te Element von L von hinten[3], $[t_i \mid i \in L]$ für die Abbildung der Listenelemente von L durch $i \mapsto t_i$ und $\forall x \cdot L \in L'. P$ für die boole'sche Aussage benutzt, daß P für alle x und L gilt, so daß $x \cdot L$ Suffix von L' ist.

Wir bemerken an dieser Stelle, daß wir nicht vorhaben, den komplexen Weg zu durchlaufen, den ein Entwickler antritt, um ausgehend von einer Problembeschreibung einen Globalsuchalgorithmus zu synthetisieren. Es gibt eine ganze Menge Entscheidungen, die dieser treffen muß; eine davon haben wir soeben schlicht ergebnishaft dargestellt, nämlich die Formalisierung des Problems. Es ist beispielsweise gar nicht selbstverständlich, die Differenzentafel zeilenweise aufzubauen, auch Spalten wären möglich. Auch im weiteren Verlauf werden Entscheidungen getroffen, die wir hier präsentieren, als ob eine natürlich auf die andere folgte, obwohl es sich in Wirklichkeit um ein Geflecht wechselseitiger Abhängigkeiten handelt. Die Reihung der Entscheidungen hat den Vorzug besserer Nachvollziehbarkeit, was für einen *Text* ausschlaggebend ist. Die maschinelle Unterstützung eines interaktiven Syntheseprozesses sollte die nicht-lineare Struktur der Entwicklung jedoch berücksichtigen.

Die Globalsuche benötigt im weiteren einen Suchraum, der vorgibt, wie die Kandidaten für L' aufgezählt werden. Um kurz zu beschreiben, auf welchen Algorithmus die Suche hinausläuft: Daß die Werte des Suchraums Listen sind, steht fest, da der Wertetyp des Suchraums und der Wertetyp des Punktproblems immer übereinstimmen müssen. Es bleibt zu klären, welche Bedingung die aufgesuchten Listen erfüllen müssen. Hierzu sind die zur Verfügung stehenden Standard-Suchräume zu sichten und auf ihre Tauglichkeit zu überprüfen. Es bieten sich natürlich vor allem Suchräume an, die es erlauben, Permutationen aufzuzählen. Aus den zahlreichen Möglichkeiten wählen wir den Suchraum, der alle Listendarstellungen einer endlichen Menge durchläuft. Verschärft man nämlich die Listendarstellung um die Forderung, keine mehrfachen Elemente zu erlauben, erhält man nur Permutationslisten der Menge. Diesen Suchraum haben wir bereits in Abbildung 1.8, Seite 31, dargestellt.

Es bleibt noch die Anpassung dieses Standard-Suchraums an das Problem mittels Filterung und Spezialisierung. Naheliegenderweise wählen wir zur Wohlfundierung den Filter, der Mehrfachnennungen in der Liste ausschließt. Als Spezialisierungstransformation θ kommt eigentlich nur $\theta(x) = \{1..x\}$ in frage, da $\{1..x\}$ die Menge ist, deren Permutationslisten aufgezählt werden. Damit ergibt sich ein erster, unoptimierter Algorithmus:

> Sei x die Eingabe. Zähle alle Listen L' auf, die die Menge $\{1..x\}$ repräsentieren, wobei von vorneherein alle L' mit mehrfach vorkommenden Elementen herausgefiltert werden. Für die übriggebliebenen, endlich vielen Listen nimm alle diejenigen in die Ausgabemenge M auf, die die Ausgabebedingung des Punktproblems erfüllen, d.h. für die insbesondere gilt, daß die Differenzentafel nur Zeilen hat, in denen keine Mehrfachelemente vorkommen.

Im wesentlichen läuft also alles darauf hinaus, den Aufzählungsalgorithmus des Standard-Suchraums mit einigen, den Suchraum genau auf die Lösungsmenge M beschränkenden Bedingungen zu versehen, sowie eine Transformation von x auf $\{1..x\}$ durchzuführen.

Bevor wir die Synthese formal darstellen, noch ein kurzer Blick auf die hier nicht verfolgten Möglichkeiten zur Optimierung. Wichtigstes Instrument dabei ist die Einführung eines zusätzlichen Filters,

[3] Listen sind etwas einfacher von hinten zu numerieren als von vorne, da der Listeninduktor eine Liste stets von hinten nach vorne durchläuft. Es spielt normalerweise keine Rolle, in welcher Richtung Listen numeriert werden.

der weitere Kandidaten möglichst streicht, bevor die aufwendige Auswertung der Ausgabebedingung des Punktproblems durchgeführt wird. Ein Filter ϕ ist von der transformierten Eingabe $\theta(x)$ und dem aktuellen Deskriptor s abhängig. Interessant ist nun, ob man an einem Deskriptor bereits ablesen kann, ob in dem Teilsuchraum, für den er steht, noch Lösungen existieren. Falls nein, könnte dieser Bereich von der weiteren Suche ausgespart bleiben.

Der von uns gewählte Suchraum benutzt als Deskriptoren Präfixe, d.h. ein Deskriptor s steht für alle Kandidaten $s{\cdot}q$, so daß $s{\cdot}q$ eine Permutationsliste von $\{1..x\}$ ist. Die Idee ist nun, bereits für diesen Präfix s eine Differenzentafel aufzustellen: Jede Zeile der Differenzentafel von s ist nämlich Präfix der Differenzentafeln aller Kandidaten, für die s steht. Finden wir in diesen Zeilen-Präfixen mehrfache Elemente, kommen diese auch in den entsprechenden Zeilen aller Kandidaten vor. Also ist keiner dieser Kandidaten eine Lösung; folglich kann der gesamte Teilsuchraum, der durch s bezeichnet wird, gekappt werden. Als zusätzlicher, effizienzsteigernder Filter kommt daher in Betracht (man beachte, daß ϕ dann `true` berechnet, wenn *nicht* herausgefiltert wird):

$$\phi(\{1..x\}, s) = \forall k \in \{1..|s|\}.\ \mathtt{nodups}(\mathtt{dtrow}(s;k))$$

Die Überlegungen, die wir gerade angestellt haben, um diesen Filter zu rechtfertigen, tauchen im formalen Beweis als Nachweis der Notwendigkeit des Filters wieder auf.

Weitere Optimierungen sind möglich, betreffen aber nicht mehr die Einengung des Suchraums, sondern seine geschickte Berechnung. Beispielsweise ist es überflüssig, die Differenzentafel für jeden Deskriptor neu zu berechnen. Ist $s' = s \cdot a$ ein Nachfolgedeskriptor von s, so muß die Differenzentafel nicht komplett neu berechnet, sondern lediglich erweitert werden. (Tatsächlich wird die Differenzentafel also spaltenweise aufgebaut! Daß wir oben die Tafel über ihre Zeilen definiert haben, ist also vom Standpunkt des Algorithmus' aus eine abseitige Methode.)

Wenden wir nun, um zu unserem Experimentierfeld zurückzukommen, die Methode der Globalsuche streng formal auf das Problem der Costas-Arrays an. Um den Eindruck möglichst unverfälscht zu lassen, haben wir den Beweis für die Lösbarkeit direkt aus NuPRL übernommen. Der durchgängige NuPRL-Beweis wurde dazu in Stücke zerlegt, die wiederum geringfügig editiert wurden, damit uninteressante Details nicht zuviel Platz beanspruchen. Desweiteren wird nur der Hauptstrang des Beweises dargestellt; die Nebenziele beschäftigen sich nur mit Pedanterien, die aus der typentheoretischen Formalisierung entspringen. Diese haben sich als problemlos beweisbar herausgestellt, so daß lediglich der Beweisaufwand der Nebenziele von Interesse ist; wir werden dies nebenbei erwähnen. Die Struktur des Beweises folgt genau dem im vorangegangenen Abschnitt vorgeschlagenen Aufbau (kein Wunder, wurde dieser doch aus dem Costas-Arrays-Beispiel destilliert). Die angegebenen Nummern (n) beziehen sich auf diesen Abschnitt.

Die ersten Beweisschritte zeigen die Abbildungen 4.2 bis 4.4, die im allgemeinen Schema (3) entsprechen. Aus der Wissensbank wird das Globalsuchtheorem zitiert, das mit dem Suchraum *gs_seq_set_3* und eben mit dem Costas-Arrays-Punktproblem instanziiert wird. Hierbei handelt es sich um den Suchraum, der Mengen als Listen aufzählt und durch den Wohlfundiertheitsfilter beschränkt ist, der doppelte Elemente in den Listen verbietet. Daß dieser Suchraum auch einer ist, entnehmen wir wieder der Wissensbank (Lemma *gs_seq_set_3_ax*). Das Nebenziel, daß die Wertetypen von Suchraum und Problem übereinstimmen, kann durch Auffaltung und Reduktion gezeigt werden; es taucht im weiteren Beweis immer wieder auf (hier in den gekürzten Teilen), so daß es beweisoptimierend wäre, dieses Ziel per Schnitt dem Restbeweis voranzustellen, um diese Aussage in einer Hypothese verfügbar zu halten. Wir erwähnen dies bloß; solche Optimierungen verkürzen zwar die Rechenzeit, erschweren aber das Verständnis des Beweises[4].

[4] Idee am Rande: Es würde die Lesbarkeit als auch die Editierbarkeit von Beweisen verbessern, wenn in Beweisbäumen unter bestimmten Umständen Querverweise erlaubt wären. Dies ginge z.B., wenn die zu beweisende Sequenz identisch ist mit der Sequenz, auf die verwiesen wird, und wenn dadurch kein Zirkelschluß entsteht. Ähnlich wie die Regel lemma auf

```
⊢ (costas_spec()*) is satisfiable

| BY InstLemma `gs_synth_specl` [gs_seq_set_3(Z↑); costas_spec()]
|    THENW Auto
|\
|  ⊢ (gs_seq_set_3(Z↑) describes global search)
|
|  <not refined>
|\
|  ⊢ R(costas_spec())=R(specification(gs_seq_set_3(Z↑)))
|
|   BY Unfolds "costas_spec gs_seq_set_3 same_range" 0
|       THEN
|        Reduce_gs_access 0
|       THEN
|        Reduce_specification 0
 \

    1. SCHEME
        ∀
          O SATISFIES odec_problem(costas_spec())
          theta SATISFIES FUNCTION theta WHICH costas_spec()
                          << specification(gs_seq_set_3(Z↑))
                          AND WHICH guarantees necessity for Phi(gs_seq_set_3(Z↑))

        ⇒
          filtered_body(gs_seq_set_3(Z↑) specialized to costas_spec() using theta
          ; O)
           SATISFIES (costas_spec()*)
       END
   ⊢ (costas_spec()*) is satisfiable

    BY pschemeConstrHyp 1
```

Abb. 4.2. *Teil 1 des Costas-Arrays-Beispiels.*

```
⊢ (gs_seq_set_3(Z↑) describes global search)

 BY BackThruLemma `gs_seq_set_3_ax`
     THENW HM
```

Abb. 4.3. *Teil 2 des Costas-Arrays-Beispiels.*

```
⊢ R(costas_spec())=R(specification(gs_seq_set_3(Z↑)))

| BY Unfolds "costas_spec gs_seq_set_3 same_range" 0
|       THEN
|        Reduce_gs_access 0
|       THEN
|        Reduce_specification 0
 \
   ⊢ Seq(Z↑) = Seq(Z↑) ∈ TYPES

    BY Folds "member" 0 THEN HM
```

Abb. 4.4. *Teil 3 des Costas-Arrays-Beispiels.*

```
1. SCHEME
     ∀
       O SATISFIES odec_problem(costas_spec())
       theta SATISFIES FUNCTION theta WHICH costas_spec()
                          << specification(gs_seq_set_3(Z↑))
                          AND WHICH guarantees necessity for Phi(gs_seq_set_3(Z↑))

     ⇒
       filtered_body(gs_seq_set_3(Z↑) specialized to costas_spec() using theta
       ; O)
        SATISFIES (costas_spec()*)
   END
⊢  (costas_spec()*) is satisfiable

  BY pschemeConstrHyp 1

     1. SCHEME
          ∀
            O SATISFIES odec_problem(costas_spec())
            theta SATISFIES FUNCTION theta WHICH costas_spec()
                               << specification(gs_seq_set_3(Z↑))
                               AND WHICH guarantees necessity for Phi(gs_seq_set_3(Z↑))

          ⇒
            filtered_body(gs_seq_set_3(Z↑) specialized to costas_spec() using theta
            ; O)
             SATISFIES (costas_spec()*)
        END
     ⊢  odec_problem(costas_spec()) is satisfiable

     <not refined>

     1. SCHEME
          ∀
            O SATISFIES odec_problem(costas_spec())
            theta SATISFIES FUNCTION theta WHICH costas_spec()
                               << specification(gs_seq_set_3(Z↑))
                               AND WHICH guarantees necessity for Phi(gs_seq_set_3(Z↑))

          ⇒
            filtered_body(gs_seq_set_3(Z↑) specialized to costas_spec() using theta
            ; O)
             SATISFIES (costas_spec()*)
        END
     ⊢  FUNCTION theta WHICH costas_spec()
        << specification(gs_seq_set_3(Z↑))
        AND WHICH guarantees necessity for Phi(gs_seq_set_3(Z↑))

        is satisfiable

     <not refined>

     1−5.
        body,b1 : ''deklariert mit bestimmten Eigenschaften''
     ⊢
        filtered_body(gs_seq_set_3(Z↑) specialized to costas_spec() using b1;body)
        ∈ BODYS((costas_spec()*))

     <not refined>

     1. b2 : BODYS((costas_spec()*))
     ⊢  b2 satisfies (costas_spec()*) ∈ P

     <not refined>
```

Abb. 4.5. *Teil 4 des Costas-Arrays-Beispiels.*

```
 1. SCHEME
      ∀
        O SATISFIES odec_problem(costas_spec())
        theta SATISFIES FUNCTION theta WHICH costas_spec()
                          << specification(gs_seq_set_3(Z↑))
                          AND WHICH guarantees necessity for Phi(gs_seq_set_3(Z↑))

      ⇒
        filtered_body(gs_seq_set_3(Z↑) specialized to costas_spec() using theta
        ; O)
         SATISFIES (costas_spec()∗)
    END
⊢── FUNCTION theta WHICH costas_spec()
    << specification(gs_seq_set_3(Z↑))
    AND WHICH guarantees necessity for Phi(gs_seq_set_3(Z↑))

    is satisfiable
 |  BY thin 1 THEN
 |     unfold_theta 0 THEN
 |     RW (AddrC [2;2]
 |        (unfold_and_simplify_formulaC “costas_spec gs_seq_set_3“)) 0
 \
  \
    ⊢─
        ∃body:⌈D(costas_spec())↛⌈D(specification(gs_seq_set_3(Z↑)))
         <FUNCTION theta
          (x:D(costas_spec()))
          :D(specification(gs_seq_set_3(Z↑)))
          WHERE I(costas_spec()) x
          RETURNS (y | Specl_cond(costas_spec();specification(gs_seq_set_3(Z↑))) x y
                                   ∧ gs_seq_set_3(Z↑)
                                     specialized to costas_spec()
                                     has nec filter at x
                                     ,y)

         ,body
         >
         terminates
         ∧ (∀x:⌈Z↑
              x ≥ 1
              ⇒ body halts on x
              ⇒ (T
                 ∧ (∀y:⌈Seq(Z↑)
                      ⌈permutation(y:seq(Z↑);{1..x}:set(Z↑))
                       ∧ (∀j∈{1..|y|}. nodups(dtrow(y;j):seq(Z↑)))
                      ⌉
                      ⇒ ⌈range(y) ⊆ body(x) : Set(Z↑)⌉))
                 ∧ (∀s:⌈Seq(Z↑)
                      x ≥ 1
                      ⇒ (∃z:⌈Seq(Z↑)
                           ⌈(s ⊑ z  : Seq(Z↑))⌉
                           ∧ ⌈permutation(z:seq(Z↑);{1..x}:set(Z↑))
                              ∧ (∀j∈{1..|z|}. nodups(dtrow(z;j):seq(Z↑)))
                             ⌉)
                      ⇒ ⌈nodups(s:seq(Z↑))⌉))

    <not refined>
```

Abb. 4.6. *Teil 5 des Costas-Arrays-Beispiels.*

```
⊢  ∃body:⌈D(costas_spec())↛⌈D(specification(gs_seq_set_3(Z↑)))
     ''eingesetztes theta−Programm'' terminates
     ∧ (∀x:⌈Z↑
          x ≥ 1
          ⇒ body halts on x
          ⇒ (T
             ∧ (∀y:⌈Seq(Z↑)
                  ⌈permutation(y:seq(Z↑);{1..x}:set(Z↑))
                   ∧ (∀j∈{1..|y|}. nodups(dtrow(y;j):seq(Z↑)))
                  ⌉
                  ⇒ ⌈range(y) ⊆ body(x) : Set(Z↑)⌉))
             ∧ (∀s:⌈Seq(Z↑)
                  x ≥ 1
                  ⇒ (∃z:⌈Seq(Z↑)
                        ⌈(s ⊑ z  : Seq(Z↑))⌉
                        ∧ ⌈permutation(z:seq(Z↑);{1..x}:set(Z↑))
                           ∧ (∀j∈{1..|z|}. nodups(dtrow(z;j):seq(Z↑)))
                          ⌉)
                  ⇒ ⌈nodups(s:seq(Z↑))⌉))

BY With letrec theta(x) = {1..x} (D 0)

  ⊢
     letrec theta(x) = {1..x} ∈ ⌈D(costas_spec())↛⌈D(specification(gs_seq_set_3(Z↑)))
  <not refined>

  ⊢  ''eingesetztes theta−Programm'' terminates
     ∧ (∀x:⌈Z↑
          x ≥ 1
          ⇒ letrec theta(x) = {1..x} halts on x
          ⇒ (T
             ∧ (∀y:⌈Seq(Z↑)
                  ⌈permutation(y:seq(Z↑);{1..x}:set(Z↑))
                   ∧ (∀j∈{1..|y|}. nodups(dtrow(y;j):seq(Z↑)))
                  ⌉
                  ⇒ ⌈range(y) ⊆ letrec theta(x) = {1..x}(x) : Set(Z↑)⌉))
             ∧ (∀s:⌈Seq(Z↑)
                  x ≥ 1
                  ⇒ (∃z:⌈Seq(Z↑)
                        ⌈(s ⊑ z  : Seq(Z↑))⌉
                        ∧ ⌈permutation(z:seq(Z↑);{1..x}:set(Z↑))
                           ∧ (∀j∈{1..|z|}. nodups(dtrow(z;j):seq(Z↑)))
                          ⌉)
                  ⇒ ⌈nodups(s:seq(Z↑))⌉))

     BY and_i

       ⊢  ''eingesetztes theta−Programm'' terminates
       <not refined>

       ⊢  ∀x:⌈Z↑. ''... wie oben''
       <not refined>
```

Abb. 4.7. *Teil 6 des Costas-Arrays-Beispiels.*

Im Schritt (3) wird die wichtigste Entscheidung des Beweises getroffen: Die Auswahl des (gefilterten) Suchraums. Die weiteren Schritte (im Hauptstrang des Beweises) bis zum Unterziel der Form (5) sind lediglich mechanische Zerlegungen, bei denen keine weitere Interaktion erforderlich ist. Die Abbildungen 4.5 und 4.6 zeigen diesen Teil für das Costas-Arrays-Beispiel. Wir bemerken, daß das erste Unterziel in Abbildung 4.5 nicht ganz uninteressant ist, weil hier die Subsynthese von O betrieben werden müßte. Im Costas-Arrays-Beispiel ist dies nicht besonders schwer, da die Ausgabebedingung in der Form $\lceil O' \rceil$ angegeben ist und diese Subsynthese darauf hinausläuft, das Weglassen der Klammern $\lceil \cdots \rceil$ zu rechtfertigen und O' als Ergebnis zurückzugeben. Falls die Ausgabebedingung solche Klammern hat, kann die Subsynthese sogar mechanisch erfolgen; hat sie keine, muß diese Subsynthese explizit durchgeführt werden. Im allgemeinen ist die Subsynthese von O also nicht zu vernachlässigen.

Wir sind nun an der Stelle, an der wir gemäß dem Schema eine Instanz für *theta* einsetzen müssen, um auf die Formel (6) zu kommen. Abbildung 4.7 zeigt, wie letrec $theta(x) = \{1..x\}$ instanziiert wird. Abbildung 4.8 stellt die Zerlegung von (6) in ihre Teile (a), (b), (c) dar.

Es ist klar, daß wir eigentlich nicht wollen, daß in einem Beweis eine Instanz einfach so „vom Himmel fällt". Dies ist zur Zeit der unvollständigen Implementierung geschuldet. Wir können uns aber versuchen vorzustellen, wie es ablaufen müßte: Die folgenden Beweisschritte, die hier für konkretes *theta* durchgeführt werden, funktionieren, bis auf den letzten, auch für allgemeines *theta*. Daher ist eine Taktik möglich, die diese Schritte vorausplant und die Instanz erst bestimmt und einsetzt, wenn sie evident ist. Als Ergebnis dieses Plans erhält man die einzelnen Schritte sowie insbesondere die Instanz. Anschließend, nachdem die Planung abgeschlossen ist, kann der Beweis genau so erzeugt werden, wie wir ihn jetzt manuell entwickeln, d.h. die Instanz wird zuallererst eingesetzt und die weiteren Beweisschritte werden als Rechtfertigung nachgereicht.

In Abbildung 4.9 sehen den Beweis für Formelteil (b). Es handelt sich um die Aussage, daß mittels *theta* der Suchraum für das konkrete Problem spezialisiert werden kann. Es handelt sich um eine Auffaltung, eine Lemmaanwendung, eine Umwandlung von boole'schen Operatoren in propositionale und einen einfachen logischen Sequenzenbeweis. Die Auffaltung bezieht sich auf die Abstraktion `permutation`, wodurch sichtbar wird, daß sich darin der `range`-Operator verborgen hält. Auf diesen kann man nämlich im zweiten Schritt das Lemma „a4_9_13" anwenden:

lemma_a4_9_13:
$\forall alpha$:`TYPES`
$\forall S$:`set`($alpha$)
$\forall S'$:`set`($alpha$)
$S \doteq S'$: `Set`($alpha$)
$= S \subseteq S'$: `Set`($alpha$) $\wedge$ $S' \subseteq S$: `Set`($alpha$)

Die Gleichheit wird damit durch gegenseitige Inklusion ausgedrückt. Die Anwendung erfolgt als Termersetzung (siehe Kapitel 3).

Der dritte Schritt hat nicht ganz geklappt, wie man sieht. Die Termersetzung durch lift_all_downATC konnte nicht fehlerfrei gerechtfertigt werden, eine Variable y wird irgendwo instanziiert, obwohl sie nicht deklariert ist. Die Termersetzungstaktik erzeugt daher neben dem (unten dargestellten) Hauptziel ein weiteres Ziel, das die nicht gerechtfertigte Behauptung enthält, hier die Äquivalenz der Terme mit boole'schen und mit propositionalen Operatoren. Da diese Äquivalenz offensichtlich gilt, lassen wir uns von dem Fehler, der hoffentlich irgendwann gefunden wird, nicht beeindrucken und tun so, als ob die Termersetzung geklappt hätte.

bereits bewiesene Theoreme verweist, wären dann Bezugnahmen auf andere Äste desselben Beweises möglich. Beweisteile können nun mehrfach verwendet werden, ohne daß Kopien angefertigt würden und ohne daß der Beweis umgestellt werden müßte.

```
⊢  ∀x:⌈Z↑
     x ≥ 1
     ⇒ letrec theta(x) = {1..x} halts on x
     ⇒ (T
        ∧ (∀y:⌈Seq(Z↑)
              ⌈permutation(y:seq(Z↑);{1..x}:set(Z↑))
               ∧ (∀j∈{1..|y|}. nodups(dtrow(y;j):seq(Z↑)))
              ⌉
              ⇒ ⌈range(y) ⊆ letrec theta(x) = {1..x}(x) : Set(Z↑)⌉))
        ∧ (∀s:⌈Seq(Z↑)
              x ≥ 1
              ⇒ (∃z:⌈Seq(Z↑)
                    ⌈(s ⊑ z  : Seq(Z↑))⌉
                    ∧ ⌈permutation(z:seq(Z↑);{1..x}:set(Z↑))
                       ∧ (∀j∈{1..|z|}. nodups(dtrow(z;j):seq(Z↑)))
                      ⌉)
              ⇒ ⌈nodups(s:seq(Z↑))⌉)

   BY all_i THEN D 0 THENM D 0 THENM (and_i THENL [and_i;Id])

       1. x : ⌈Z↑
       2. x ≥ 1
       3. letrec theta(x) = {1..x} halts on x
     ⊢  ∀y:⌈Seq(Z↑)
           ⌈permutation(y:seq(Z↑);{1..x}:set(Z↑))
            ∧ (∀j∈{1..|y|}. nodups(dtrow(y;j):seq(Z↑)))
           ⌉
           ⇒ ⌈range(y) ⊆ letrec theta(x) = {1..x}(x) : Set(Z↑)⌉

       BY RW (HigherC Y_redexC) 0

          ⊢  ∀y:⌈Seq(Z↑)
                ⌈permutation(y:seq(Z↑);{1..x}:set(Z↑))
                 ∧ (∀j∈{1..|y|}. nodups(dtrow(y;j):seq(Z↑)))
                ⌉
                ⇒ ⌈range(y) ⊆ {1..x} : Set(Z↑)⌉

            <not refined>

       1. x : ⌈Z↑
       2. x ≥ 1
       3. letrec theta(x) = {1..x} halts on x
     ⊢  ∀s:⌈Seq(Z↑)
           x ≥ 1
           ⇒ (∃z:⌈Seq(Z↑)
                 ⌈(s ⊑ z  : Seq(Z↑))⌉
                 ∧ ⌈permutation(z:seq(Z↑);{1..x}:set(Z↑))
                    ∧ (∀j∈{1..|z|}. nodups(dtrow(z;j):seq(Z↑)))
                   ⌉)
           ⇒ ⌈nodups(s:seq(Z↑))⌉

     <not refined>
```

Abb. 4.8. *Teil 7 des Costas-Arrays-Beispiels.*

```
 1. x : ⌈Z↑
 2. x ≥ 1
 3. letrec theta(x) = {1..x} halts on x
⊢  ∀y:⌈Seq(Z↑)
     ⌈permutation(y:seq(Z↑);{1..x}:set(Z↑)) ∧ (∀j∈{1..|y|}. nodups(dtrow(y;j):seq(Z↑)))⌉
     ⇒ ⌈range(y) ⊆ {1..x} : Set(Z↑)⌉

 BY Unfold 'seq_permutation' 0

   ⊢  ∀y:⌈Seq(Z↑)
        ⌈nodups(y:seq(Z↑)) ∧ (range(y) ≐ {1..x} : Set(Z↑))
         ∧ (∀j∈{1..|y|}. nodups(dtrow(y;j):seq(Z↑)))
        ⌉
        ⇒ ⌈range(y) ⊆ {1..x} : Set(Z↑)⌉

    BY RW
         (TCtoC P
          (ATCtoTC
           (AddrATC [2;1;1;1;2] (apm_gen_LemmaATC 'lemma_a4_9_13' [] IdEQ 'LR'))
          )
         ) 0
        THENW ...

      ⊢  ∀y:⌈Seq(Z↑)
           ⌈nodups(y:seq(Z↑))
            ∧ range(y) ⊆ {1..x} : Set(Z↑) ∧ {1..x} ⊆ range(y) : Set(Z↑)
            ∧ (∀j∈{1..|y|}. nodups(dtrow(y;j):seq(Z↑)))
           ⌉
           ⇒ ⌈range(y) ⊆ {1..x} : Set(Z↑)⌉

       BY RW
            (TCtoC P
             (ATCtoTC (AddrATC [2;1] lift_all_downATC))
            ) 0
           THENW ...

          4. x1 : ⌈Seq(Z↑)
         ⊢
              ...refine: ... -> Free Variable y occurs in conclusion.
              ... ⌈nodups(x1:seq(Z↑))
                    ∧ range(x1) ⊆ {1..x} : Set(Z↑)
                      ∧ {1..x} ⊆ range(x1) : Set(Z↑)
                    ∧ (∀j∈{1..|x1|}. nodups(dtrow(x1;j):seq(Z↑)))
                  ⌉
                  ⇐⇒ (⌈nodups(x1:seq(Z↑))⌉
                         ∧ ⌈range(x1) ⊆ {1..x} : Set(Z↑)⌉
                         ∧ ⌈{1..x} ⊆ range(x1) : Set(Z↑)⌉)
                         ∧ ⌈(∀j∈{1..|x1|}. nodups(dtrow(x1;j):seq(Z↑)))⌉

         BY % fehlerhafte Ziel, siehe Begleittext %

         ⊢  ∀y:⌈Seq(Z↑)
              (⌈nodups(y:seq(Z↑))⌉
              ∧ ⌈range(y) ⊆ {1..x} : Set(Z↑)⌉
              ∧ ⌈{1..x} ⊆ range(y) : Set(Z↑)⌉)
              ∧ ⌈(∀j∈{1..|y|}. nodups(dtrow(y;j):seq(Z↑)))⌉
              ⇒ ⌈range(y) ⊆ {1..x} : Set(Z↑)⌉

         BY search_and_transform (\x. IdC)
              THENW
               HM
```

Abb. 4.9. *Teil 8 des Costas-Arrays-Beispiels.*

```
  1. x : ⌈Z↑
  2. x ≥ 1
  3. letrec theta(x) = {1..x} halts on x
⊢ ∀s:⌈Seq(Z↑)
    x ≥ 1
    ⇒ (∃z:⌈Seq(Z↑)
         ⌈(s ⊑ z  : Seq(Z↑))⌉
         ∧ ⌈permutation(z:seq(Z↑);{1..x}:set(Z↑))
           ∧ (∀j∈{1..|z|}. nodups(dtrow(z;j):seq(Z↑)))
          ⌉)
    ⇒ ⌈nodups(s:seq(Z↑))⌉

  BY Unfold 'seq_permutation' 0

    ⊢ ∀s:⌈Seq(Z↑)
        x ≥ 1
        ⇒ (∃z:⌈Seq(Z↑)
             ⌈(s ⊑ z  : Seq(Z↑))⌉
             ∧ ⌈nodups(z:seq(Z↑)) ∧ (range(z) ≐ {1..x} : Set(Z↑))
               ∧ (∀j∈{1..|z|}. nodups(dtrow(z;j):seq(Z↑)))
              ⌉)
        ⇒ ⌈nodups(s:seq(Z↑))⌉

      BY RW
           (TCtoC P
            (ATCtoTC (AddrATC [2;2;1;2;2] lift_all_downATC))
           ) 0
          THENW ...

        ⊢ ''% ebenfalls fehlerhaft, siehe oben %''

        <not refined>

        ⊢ ∀s:⌈Seq(Z↑)
            x ≥ 1
            ⇒ (∃z:⌈Seq(Z↑)
                 ⌈(s ⊑ z  : Seq(Z↑))⌉
                 ∧ (⌈nodups(z:seq(Z↑))⌉ ∧ ⌈(range(z) ≐ {1..x} : Set(Z↑))⌉)
                 ∧ ⌈(∀j∈{1..|z|}. nodups(dtrow(z;j):seq(Z↑)))⌉)
            ⇒ ⌈nodups(s:seq(Z↑))⌉

          BY RW
               (TCtoC P
                (ATCtoTC
                 (AddrATC [2;2;1;2;2;1;1]
                  (apm_gen_LemmaATC 'lemma_prefix_nodups' ['x',s] FailEQ 'LR'))
                )
               ) 0    THENW ...

            ⊢ ''zwei weitere fehlerhafte Ziele''

            <not refined>

            ⊢ ∀s:⌈Seq(Z↑)
                x ≥ 1
                ⇒ (∃z:⌈Seq(Z↑)
                     ⌈(s ⊑ z  : Seq(Z↑))⌉
                     ∧ (⌈nodups(s:seq(Z↑))⌉ ∧ ⌈(range(z) ≐ {1..x} : Set(Z↑))⌉)
                     ∧ ⌈(∀j∈{1..|z|}. nodups(dtrow(z;j):seq(Z↑)))⌉)
                ⇒ ⌈nodups(s:seq(Z↑))⌉

              BY search_and_transform (\x. FailC)
                 THENW HM
```

Abb. 4.10. *Teil 9 des Costas-Arrays-Beispiels.*

Der letzte Schritt ist ein einfacher logischer Schluß, der hier von der Taktik search_and_transform gefunden und bewiesen wird.

Nun zu Abbildung 4.10, die den Beweis für Formelteil (c) darstellt. Die Schritte sind den vorangegangenen ähnlich, nur daß ein anderes Lemma benutzt wird:

lemma_prefix_nodups:
$\forall alpha$:TYPES
$\forall X$:set($alpha$)
$\forall Y$:set($alpha$)
$X \sqsubseteq Y$
$\Rightarrow$ nodups(Y)
$\Rightarrow$ nodups(X)

Die Termersetzung funktioniert hier noch weniger als zuvor, die aufgetretenen Fehler sind jedoch gleich dem vorigen. Wir sehen, daß die Termersetzungstaktik noch nicht so weit entwickelt ist, daß sie in einem hohen Grad korrekt funktioniert, und man sich deshalb im Augenblick hüten sollte, sie anders als manuell zu benutzen.

Der tatsächlich in NuPRL geführte Beweis für das Costas-Arrays-Beispiel ist komplett, d.h. auch die hier nicht gezeigten Äste sind vorhanden. Daher kann man sich prinzipiell den Extraktterm ausgeben lassen, der dann die synthetisierte Funktion enthält.

4.5 Offene Probleme

Der derzeitige Stand der Implementierung der Globalsuche ist mehr eine Skizze eines Projekts als seine Realisierung. Im gehen im folgenden die wichtigsten Punkte durch, die noch verbessert, verfeinert, restrukturiert oder nachgereicht werden müssen.

4.5.1 Automatisierung der *theta*-Bestimmung

Im Augenblick wird die Transformationsfunktion *theta* an der Stelle des Beweises, an dem deren Existenz behauptet wird, einfach als „vom Himmel gefallener“ Term eingesetzt. Dies ist natürlich eine Vorwegnahme eines Wissens, das bei einer Automatisierung des ganzen Beweisschemas nicht gegeben ist. Es ist daher zu klären, wie man an diesen *theta*-Term kommt und durch welches Verfahren die Beweisschritte, die ihn rechtfertigen, gefunden werden.

Man sehe sich daraufhin noch einmal den entsprechenden Teil des Costas-Arrays-Beispiels an. Der *theta*-Term wird durch vier Schritte bewiesen: (i) Eine Auffaltung, (ii) eine Termersetzung, (iii) eine Konversion boole'scher Operatoren in propositionale und (iv) einen sehr einfachen logischen Beweis. Die Schritte (i) bis (iii) können ohne Kenntnis von *theta* durchgeführt werden; in (iv) muß spätestens ein konkreter Term vorliegen. Schritt (iv) ist dabei rein logisch, d.h. es kommt nur auf die Identität von $theta(x)$ mit dem Term $\{1..x\}$ an, nicht jedoch auf den inneren Aufbau von *theta*.

Wir nehmen nun an, wir könnten diese Situation verallgemeinern, und machen daher folgenden Ansatz zur Automatisierung der Bestimmung von *theta*: Wir setzen *theta* zunächst nicht ein, entfernen aber zu Zwecken der Beweisplanung den Existenzquantor, ersetzen alle Vorkommen von $theta(x)$ durch t (die Funktion kommt nur in dieser Form vor) und denken uns die Variable t als implizit existenzquantifiziert. Dann suchen wir geeignete Termersetzungsschritte (1) bis $(n-1)$, so daß der abschließende rein logische Schritt (n) möglich ist. In diesem logischen Schritt muß nun eine Instanz der implizit existenzquantifizierten Variablen t gefunden werden. Für *theta* wurde so insgesamt die Funktion letrec $theta(x) = t$ bestimmt.

Es ist klar, daß wir die Schritte (1) bis (n) zunächst nicht als reale Beweisschritte, sondern als Planungsschritte durchführen. Erst anschließend können die vorausgeplanten Schritte in einen Beweis übersetzt werden, da nun die Instanz für *theta* bekannt ist.

4.5.2 Optimierung: Filter zur Effizienzsteigerung

Wie bereits angemerkt, kann man zusätzliche Filter einbauen, die den Suchraum von vorneherein stark begrenzen, was zu effizienteren Algorithmen führt. Es ist angebracht, diese Möglichkeit weiterer Filter direkt in der Theorie vorzusehen, da man nur so Beweisstrukturen erhält, die eine nachträgliche Effizienzverbesserung erlauben. Wir stellen uns dies so vor, daß es im Globalsuchbeweis ein neues Unterziel gibt, daß die Existenz eines solchen zusätzlichen Filters behauptet. Interessiert man sich zunächst nur dafür, ob die Synthese überhaupt funktioniert, setzt man für diesen Filter die Identität ein, schaltet ihn sozusagen ab, und versucht, den restlichen Beweis zu führen. Hat man damit Erfolg, wird man die Möglichkeiten zur Effizienzsteigerung untersuchen und den zusätzlichen Filter stärker machen.

Die automatische Unterstützung für effizienzsteigernde Filter könnte aufgrund der Notwendigkeitsbedingung, die solche Filter einhalten müssen, eine Reihe von Kandidaten ableiten, die alle korrekt sind. Der Benutzer müßte sich dann ein Urteil über die Effizienz dieser Kandidaten bilden und einen auswählen.

4.5.3 Weitere Optimierungsmöglichkeiten

Der von der Globalsuchsynthese erzeugte Algorithmus ist noch lange nicht optimal. Es bieten sich insbesondere nachträgliche Optimierungsschritte an, die den bereits generierten Algorithmus verbessern. Im KIDS-System sind folgende Optimierungen vorgesehen:

- Simplifikation: Versuche Terme mittels der Lemmata der Wissensbank durch einfachere äquivalente Terme zu ersetzen.

- Partielle Auswertung: Es wird versucht, einen Teil des synthetisierten Programms vorab (also ohne konkrete Eingabe) auszuwerten.

- Endliche Differenzierung: Versuche in einer rekursiven Berechnung eine Variable in folgender Weise zu optimieren: Statt sie in jedem Schritt der Rekursion vollständig neu zu berechnen, benutze ihren letzten Wert, um ihren neuen Wert daraus inkrementell zu bestimmen.

- Fallanalyse: Versuche zusätzliche Bedingungen während des Programmlaufs zu bestimmen, um bestimmte Programmteile nur dann berechnen zu müssen, wenn dies erforderlich ist.

4.5.4 Interaktive Taktiken

NuPRL gibt derzeit die Interaktionsmöglichkeiten strikt vor. Taktiken müssen in einem Beweisfenster gestartet werden und haben selbst keine weiteren Möglichkeiten, mit dem Benutzer zu kommunizieren. Dies ist keine schöne Oberfläche für Synthese.

Neben den Beweisfenstern sollten NuPRL-Anwender die Möglichkeit erhalten, eigene Interaktionsformen zur Beweisführung zu programmieren. Da Beweise mittels eines gekapselten Typs dargestellt werden, bestehen von daher keine Einwände, die Oberfläche der Beweise aus dem Systemkern herauszunehmen und der Anwendungsprogrammierung zugänglich zu machen.

Eine Oberfläche für die Synthese würde den Beweis, der während der Synthese konstruiert wird, zunächst unsichtbar lassen und nur die vom Benutzer zu treffenden Entwurfentscheidungen darstellen. Die Oberfläche würde von sich aus Vorschläge machen, welche Entscheidung unter den Möglichen die wahrscheinlich beste ist. Der Benutzer kann sich den Beweis, wenn er will, auch anzeigen lassen. Eine weitere Interaktionsform ist, den Extraktterm zu berechnen und das synthetisierte Programm mit Testeingaben ablaufen zu lassen oder Effizienzanalysen durchzuführen.

4.5.5 Beweise der Theorie der Globalsuche und der Hintergrundtheorien

Die Theorie der Globalsuche wurde zwar aufgestellt, jedoch nicht bewiesen. Es ist damit zu rechnen, daß diese Beweise recht kompliziert sind. Sie zu finden ist ein eigenes Forschungsprojekt.

Die Hintergrundtheorien, etwa für die einzelnen Datentypen, sind wesentlich einfacher gestrickt. Einzelne Sätze dieser Theorien zu beweisen, ist bereits jetzt möglich. Allerdings bestehen diese Theorien aus sehr vielen Sätzen, so daß eine automatische Unterstützung wünschenswert ist. Zum Teil können ähnliche Taktiken eingesetzt werden wie bei der Bestimmung der *theta*-Funktion, zum Teil sind wohl eher Induktionsbeweiser angebracht.

Kapitel 5

Zusammenfassung, Bewertung und Ausblick

Wir wollen hier die wichtigsten Ergebnisse der Untersuchung zusammentragen und einer Bewertung unterziehen. Dabei kommen zunächst die Ansprüche in betracht, die in der Einleitung als Arbeitshypothesen formuliert wurden. Eine weitere Bewertung soll hinsichtlich der Geeignetheit des NuPRL-Ansatzes der taktisch konstruierten Beweise gegeben werden. Wir beschließen die Arbeit mit der Vision eines Synthesesystems, das undogmatisch die verschiedensten Methoden vereinigen kann.

5.1 Zusammenfassung der Ergebnisse

Die Arbeit untersucht eine spezielle Konstellation, die aus folgenden Beiträgen zusammengesetzt ist:

- einem grundlegenden Kalkül, der intuitionistischen Typentheorie
- einem Abstraktionsmechanismus, der die Erweiterung der typentheoretischen Sprache erlaubt
- einem konkreten Erweiterungsvorschlag, der auf einer Strukturierung im Sinne abstrakter Datentypen beruht, und auf eine für praktische Probleme verwendbare Programmiersprache zielt
- einer Einteilung in drei Sprachbereiche (APM, APM+P_1, Meta-APM), um bestimmte Probleme, vor allem die Typprüfung, lösen zu können
- einer Konzeption, um über Programme Aussagen machen zu können; hierzu gehören Spezifikationen, Korrektheit, Programmschemata
- einem taktischen Beweissystem NuPRL, das vor allem für folgende Eigenschaften verantwortlich zeichnet: taktische Beweiskonstruktion und -verifikation, Interaktivität, Bibliothekskonzept (und vieles mehr)
- dem Aufbau einer Termersetzungstaktik, die Typinformationen nutzen kann und Termersetzungen optimiert
- der Implementierung einer einfachen logischen Taktik, die Sequenzenbeweise finden kann
- der Globalsuch-Methode, die ein Algorithmenschema ist, das Probleme mittels des Durchkämmens eines Suchraums löst
- der Formalisierung der Globalsuch-Methode mittels des Abstraktionsmechanismus und des Bibliothekskonzepts in dem Kalkül der Typentheorie
- einer Konzeption zur Operationalisierung der Globalsuch-Methode

Einige dieser Beiträge gelten dabei als feste Grundlage, deren Annahme zunächst außerhalb der Diskussion steht, weil nur relativ zu dieser Grundlage die anderen Punkte bewertet werden können. In diesem Sinne grundlegend sind die Typentheorie, der Abstraktionsmechanismus, NuPRL und die Globalsuch-Methode. Wir können *hier* höchstens feststellen, daß diese Grundlagen entweder taugen oder ggf. in der Weise kritisieren, daß bestimmte Phänomene auf Unzulänglichkeiten der Grundlage zurückgeführt werden, sie nicht aber grundsätzlich in frage stellen.

Einige weitere Beiträge stehen gewissermaßen im Schatten, weil sie nur Hilfskonzeptionen in Bezug auf andere sind; hierzu gehören die abstrakten Datentypen und die Konzeptionen für die Aussagen über Programme. Wir können nur feststellen, daß sie ihren Dienst geleistet haben.

Nun aber zu dem eigentlichen Inhalt der Arbeit. Hauptziel war die Implementierung der Globalsuch-Methode im NuPRL-System. Diese zerteilt sich in drei große Untersuchungsgebiete: einer Absicherung der Grundlagen, indem die Sprache zum Gegenstand gemacht wird; des Entwurfs von allgemeinen Techniken zur Beweiskonstruktion; schließlich die Anwendung auf die formalisierte und operationalisierte Methode der Globalsuche.

Ausgehend von einem konkreten Erweiterungsvorschlag der typentheoretischen Sprache in Richtung einer Programmiersprache wird vorgeschlagen, bei der Spracherweiterung drei Sprachbereiche zu unterscheiden. Die Merkmale zur Unterscheidung sind den bei Syntheseaufgaben vorkommenden sprachlichen Ausdrucksmitteln entnommen. Demnach gibt es einen Bereich (APM für „abstraktes Programmier-Modell" genannt), der für die Formulierung des ausführbaren Codes zuständig ist; einen weiteren, den ersten enthaltenden Bereich, der dafür da ist, Bedingungen des Programmablaufs auszudrücken (APM+P_1, weil schlicht die Prädikatenlogik hinzugenommen wird); schließlich eine dritte, beide vorherigen Bereiche enthaltene Ausdrucksebene, die Programme im ganzen zum Thema macht, etwa, indem deren Korrektheit konstatiert wird (Meta-APM). Die Abgrenzung zwischen den Bereichen wird präzisiert, indem syntaktische Kriterien angegeben werden. Desweiteren werden die Bereiche hinsichtlich ihrer Eigenschaften bezüglich des Rechnens, des Typisierens und des Schließens untersucht. Insbesondere ergibt sich dabei, daß es möglich ist, die Bereiche APM und APM+P_1 so zu verfassen, daß eine falsifizierende Typprüfung möglich ist. Diese Aussage wird leider nur dadurch gewonnen, daß typische und bekannte Probleme, die dieser Typprüfung entgegenstehen, diskutiert werden; nicht jedoch, indem eine theoretische Untersuchung darüber angestellt wird. Letzteres hat noch zu erfolgen, um die Ergebnisse aussagekräftig zu machen. – Die Möglichkeit der Typprüfung spielt im weiteren Verlauf eine große Rolle, können doch Probleme der Instanziierung von getypten universellen Formeln sowie der Umgebungserweiterung bei der Termersetzung gelöst werden.

Im Hinblick auf die Unterteilung in Sprachbereiche konnten auch gewisse Problematisierungen der Typentheorie selbst vorgenommen werden. Zunächst zeigt sich deren relative Ausdrucksstärke als unverzichtbare Eigenschaft, mit der erstens Phänomene der Programmierung wie etwa Definitionslücken und zweitens Sprachkonstrukte ausdrückbar sind, mit denen über Programme gesprochen werden kann. Die Ausdrucksstärke ist jedoch unvereinbar mit der Entscheidbarkeit vor allem der Typisierung. Durch die Einteilung in Bereiche kann man jedoch diese Vor- und Nachteile in gewissen Grenzen manövrieren.

Die Implementierung der Globalsuch-Methode macht von dem Konzept der Taktiken sehr stark Gebrauch. NuPRL erlaubt es, daß Taktiken, also Algorithmen, die Kalkülbeweise erzeugen, in Form einer Regelanwendung aufgerufen werden können. Damit existiert ein Werkzeug, mit dem die primitiven Kalkülregeln um leistungsstärkere taktische Regeln erweitert werden können, ohne Einbußen hinsichtlich der Korrektheit zu riskieren. Diese Technik wird nun angewandt, um auf einem angemessenen Abstraktionsniveau Beweistechniken zur Verfügung stellen zu können, die zur Konstruktion von Beweisen benötigt werden, die bei der Instanziierung der Globalsuch-Methode anfallen. Insbesondere kann man so Theoreme der Bibliothek automatisch instanziieren, man kann Termersetzungen innerhalb von Formeln auf der Basis von Theoremen durchführen, und man kann einfache Sequenzenbeweise automatisch finden. Die Möglichkeit einer Typprüfung führt bei den Techniken, die eine Theoreminstanziierung durchführen, dazu, daß einige zusätzliche, praktisch jedoch vorkommende Fälle gelöst werden können, weil Typvariablen bei der Instanziierung durch Typinferenzen behandelt werden können. Die vollständige Automatisierung dieser Techniken rückt damit einen bedeutenden Schritt näher. Es zeigte sich jedoch, daß vor allem bedingte Termersetzungen die Analyse von Kontextbedingungen erforderlich machen, wofür es noch keinen Ansatz gibt.

Es wurde ein Vorschlag gemacht, wie die Globalsuch-Methode formalisiert werden kann. Dabei zeigte sich, daß man beim Aufbau einer Theorie sowie des zugrundeliegenden Abstraktionensystems viele Freiheiten hat. Es stellt sich damit die Frage, wie diese genutzt werden können, welche Ziele erreicht werden sollen. Man kann auf der einen Seite eine Theorie so strukturieren, daß ihr innerer Aufbau gut nachvollziehbar ist (sich mit intuitiven Vorstellungen deckt), auf der anderen Seite aber auch so, daß eine Theorie möglichst gut von Operationen (wie etwa der Beweiskonstruktion) genutzt werden kann. Beides läßt sich nicht immer unter einen Hut bringen. Es ist klar, daß bei der Implementierung einer Theorie dem Kriterium der Operatibilität immer die höhere Priorität eingeräumt wird. – Die Formalisierung der Globalsuche entfernt sich damit beachtlich von einer Beschreibung, die mit den traditionellen Ausdrucksmitteln der Mathematik dargestellt wird. Daneben ging es aber zunächst einmal darum herauszufinden, ob eine Formulierung gefunden werden kann, die sich auch bei der Eingabe in das reale NuPRL-System als vereinbar mit den Typregeln der Typentheorie herausstellt. Dies konnte positiv beantwortet werden. Offen bleibt aber, ob aufgrund dieser Formulierung die Theorme der Globalsuche innerhalb des Kalküls bewiesen werden können.

Nachdem die Formalisierung der Globalsuch-Methode zur Verfügung steht und die Techniken zur Beweiskonstruktion eingesetzt werden können, stellt sich noch die Frage, ob tatsächlich Instanzen von Globalsuchsynthesen beweisbar sind. Dazu wurde ein Beispiel erarbeitet (das Costas-Arrays-Problem), für das eine Lösung mittels der Globalsuche bekannt ist und versucht, hierfür einen Beweis zu finden. Dies ist (im großen und ganzen) gelungen, wenn auch bei der Implementierung des Beweises einige Detailfehler nicht beseitigt werden konnten. Anhand dieses Beispiels wurde dann analysiert, wie im allgemeinen ein Instanzenbeweis aussieht und welche Teile von Fall zu Fall anzupassen sind. Daß eine solche Unterscheidung bei der Globalsuche möglich und erfolgreich ist, wurde bereits in der Literatur gezeigt; hier ging es darum, ob und wie bei der gewählten NuPRL-Implementierung die Beweisteile unterscheidbar sind und ob die Anpassung an den konkreten Fall gut operationalisierbar ist. Es konnten Vorschläge gemacht werden, wie dies erfolgen kann.

Die Untersuchung der Operationalisierbarkeit blieb leider im Beispielhaften stecken. Das Fernziel, eine Taktik zu schaffen, die die Methode der Globalsuche weitgehend automatisch, ggf. mit geringfügiger interaktiver Hilfe durchführt, wie es etwa das KIDS-System vormacht, konnte daher nicht erreicht werden. Die Techniken zur Beweiskonstruktion haben bereits einen Komplexionsgrad erreicht, der es schwierig macht, Fehlern der Implementierung nachzuspüren. Es konnte so nicht die notwendige Zuverlässigkeit der Basistechniken sichergestellt werden, auf die eine weitgehend automatische Taktik angewiesen ist.

5.2 Überprüfung der Ansprüche des Resultats

Wir rollen an dieser Stelle noch einmal die Diskussion über einige Ansprüche und Hypothesen auf, die in der Einleitung bereits angesprochen wurden.

Zum Korrektheitsanspruch: Das NuPRL-System zeigt, daß ein formalistischer und ein intersubjektiver Korrektheitsanspruch sich nicht ausschließen müssen, sondern in der praktischen Umsetzung vermittelbar sind. NuPRL gewärleistet, daß nur Beweise akzeptiert werden, die den Regeln des Kalküls entsprechen; die Beweise sind also stets formal korrekt. Wir können nun aber auch sehen, daß wir in der Lage sind, (in gewissen Grenzen) *unsere Vorstellungen* über Mathematik in die Ausprägung des

Kalküls hineinzutragen. Wir haben dies in dieser Arbeit dadurch bewerkstelligt, daß wir unter einer Abstraktion stets die Kalkül-Repräsentation eines vorgängigen Konzepts verstanden haben. So konnten wir stets Ausdrücke des Kalküls relativ zu unserer Vorstellungswelt interpretieren und dabei sicher sein, keine allzu großen Fehler zu machen. In NuPRL kann desweiteren jeder formale Beweis vom Benutzer betrachtet und bewertet werden; der Übereinstimmung der formalen Korrektheit mit der intersubjektiven wird bereits im prozessualen Verlauf der Beweiserzeugung Rechnung getragen. Auf diese Weise kann man stets sicherstellen, daß der formale Beweis nie seinen Bezug zu den allgemein geteilten Vorstellungen über Mathematik verliert.

NuPRL bietet bestimmte Techniken, die diese „Brücke" zwischen formaler und intersubjektiver Welt herzustellen erleichtern: Alle Beweise können in einer verständlichen Form gegengelesen werden; durch Benutzung taktischer Regeln ist es möglich, den Beweisen eine Tiefenstruktur zu geben, die die Details unterordnet und somit zur besseren Verständlichkeit beiträgt; Beweisen wird als ein interaktiver Prozeß verstanden.

In dieser Arbeit haben wir nun gezeigt, daß es tatsächlich möglich ist, die Formalisierung der Konzeptualisierung unterzuordnen. Es hat sich aber auch gezeigt, daß die Formalisierung ihre eigenen Gesetzmäßigkeiten hat. Beispielsweise hat sich die formale Globalsuch-Theorie recht weit von dem zuvor dargestellten Konzept entfernt; dies ist vor allem der besseren Operationalisierbarkeit geschuldet. In der Tat muß man also vom Ideal Abstriche machen; durch ein geschicktes Vorgehen kann man jedoch erreichen, daß die Zugeständnisse nicht allzu hoch ausfallen.

Zur Spontaneität der Synthese: Wie bereits in der Einleitung ausgeführt, nimmt der Autor hier eine recht kritische Position ein. Es ist zwar faszinierend, eine Problemlösung als spontanen Ausdruck zu verstehen, doch muß vor einer Strapazierung dieser Betrachtung gewarnt werden. Um es zu wiederholen: Die Faszination rührt nur daher, daß eine allgemeine, eigentlich unlösbare Aufgabe den Hintergrund abgibt für eine den besonderen Umständen geschuldete Lösbarkeit. Wenn also das Costas-Arrays-Problem mittels der Globalsuche der Synthese zugänglich gemacht werden kann, ist dies nur deshalb aufregend, weil wir dabei Wörter wie etwa „Synthese" verwenden, die suggerieren, man diskutiere im Zusammenhang eines universellen Problem-Synthetisierers.

Bei der Globalsuche ist auffallend, daß die Methode an sich nicht auf eine bestimmte Klasse von Problemen beschränkt ist. Tatsächlich kann man *theoretisch* jedes Problem als Globalsuche auffassen; damit wird sozusagen eine Kulisse aufgebaut, die Erwartungen weckt. Natürlich sagt dies noch nichts über die praktische Durchführbarkeit der Methode; diese ist jedoch von einige glücklichen Zufällen abhängig. Man kann leider keine Theorie darüber aufstellen, die es erlauben würde, die Klasse der Probleme, für die die Methode durchführbar ist, zu charakterisieren; der Erfolg ist also unbestimmt, und es ist besser, sich als Optimisten zu geben. So können wir uns denn ganz der Faszination hingeben, daß wir eine Methode besitzen, die hin und wieder funktioniert.

Die Wissenschaft hat leider keine Kategorie für dieses Phänomen hervorgebracht, die uns zu vernünftigen Vorstellungen anleiten könnte. Um seriös zu bleiben, können wir nur in Beispielen reden: Hier funktioniert die Globalsuche gut, und dort ebenfalls. Es ist nicht möglich, den Status solcher Erkenntnisse genauer zu fixieren und ihre Tragweite zu beurteilen. Als letzte Zuflucht bleibt uns nur die Praxis: Etwa Untersuchungen darüber, wie bedeutend die Probleme sind, die bisher gelöst worden sind.

5.3 Zum taktischen Beweisansatz

Taktische Regeln ermöglichen es uns, zwei eigentlich divergierende Ziele auf einen Nenner zu bringen: Zum einen nur Beweise gelten zu lassen, die den Primitiven der Typentheorie entsprechen, zum anderen die Beweise auf einem der Problemstellung angemessenen Abstraktionsniveau zu formulieren. Taktiken sind damit das wichtigste Hilfsmittel, um von Menschen gemachte Konzepte in eine maschinengängige Form zu bringen. Beweise können somit in ähnlicher Weise strukturiert werden, wie dies aus der Mathematik bekannt ist. Desweiteren haben wir so eher die Chance, den Erfahrungsschatz der Mathematik für die interaktive und sogar automatische Beweiskonstruktion zugänglich zu machen.

Die Verwendung von Taktiken hat leider einen erheblichen Nachteil, was die Rechenzeit angeht. Jeder höhere Beweisschritt muß auf eine Unzahl primitiver Regelanwendungen abgebildet werden; häufig muß diese Abbildung sogar zunächst geplant werden, um bestimmte Anteile der späteren Konstruktion des Primitivbeweises zu berechnen. Die lange Rechendauer ist vor allem ein Problem während der interaktiven Entwicklung eines Beweises, da dem Benutzer das Ausprobieren, ob eine Taktik weiterhilft, auf diese Weise verleidet wird.

Eine Lösung könnte sein, die Abbildung auf die Primitiven nur optional durchzuführen, und gewöhnlich nur so zu tun, als ob abgebildet worden wäre. Die taktische Regel wird also doppelt implementiert: Zum einen als Abbildung, zum anderen als Simulation dieser Abbildung. Dadurch hat man die Möglichkeit, während der Beweisentwicklung die schnellere Simulation vorzuziehen und erst, wenn der Beweis bereits auf diese Weise komplettiert worden ist, die Simulation abzuschalten und die Korrektheit des Beweises dadurch gegenzuprüfen, daß die zeitraubende Abbildung auf die Primitiven durchgeführt wird. Auf diese Weise könnte Effizienz gewonnen werden, ohne auf die formale Korrektheit zu verzichten.

Freilich setzen wir dabei voraus, daß die Simulation wirklich exakt ist. Bereits eine unbedeutende kleine Veränderung, etwa die Wahl eines anderen Variablennamens, kann dazu führen, daß die Gegenprüfung scheitert. Dies ist unbedingt auszuschließen. Gewisse Veränderungen könnte man zwar bei der Gegenprüfung tolerieren, aber es sollte kein Risiko bestehen, daß ein eigentlich korrekter Beweis auf diese Weise fehlschlägt.

Kandidaten für solche Taktiken sind insbesondere:

- Typisierungstaktiken, deren Simulation mittels Typprüfungsalgorithmen stattfinden könnte
- Lemmainstanziierung
- Termersetzung (das NuPRL-Termersetzungspaket erlaubt bereits jetzt eine, freilich unzureichende Simulation)
- logische Beweistaktiken, bei denen man in die Simulation auf die Transformation der Darstellung, in der der Beweis gefunden wurde, in einen NuPRL-Beweis verzichtet

Die Möglichkeit der Taktiksimulation sollte fest in die NuPRL-Konzeption integriert werden. Dies kann z.B. bedeuten, daß für jedes Theorem in der Bibliothek zusätzlich gespeichert wird, mit welchem Simulationsansatz der Beweis geführt wurde. Besser ist noch zu erlauben, denselben Beweis simultan mit unterschiedlichen Simulationen speichern zu können. Derselbe Regelbaum, der den Beweis ausmacht, kann dann mehrfach instanziiert (ausgeführt) werden, wobei zu jeder Instanz der benutzte Simulationsansatz gehört.

Kommen wir noch zu einem anderen Kritikpunkt. Normalerweise ist es nicht so, daß eine Taktik direkt primitive Regeln erzeugt, sondern zunächst wiederum taktische Regeln, die dann weiter heruntergebrochen werden. Auf diese Weise besitzt ein Beweis auch eine Tiefenstruktur. Das aktuelle NuPRL erlaubt zwar, Beweise mit sichtbarer Tiefenstruktur zu erzeugen, bietet aber noch nicht ausreichend Möglichkeiten an, diese Tiefe zu handhaben.

Zunächst sollte es wesentlich einfacher sein, eine neue Tiefenebene in den Beweis einzuziehen. Im Augenblick ist es dazu erforderlich, zunächst eine textuelle Darstellung der taktischen Regel zu erzeugen. Es ist sehr mühsam, sich als Taktikprogrammierer auch noch darum zu kümmern. Besser wäre es, zwei Ausführungsmodi von Taktiken zu unterscheiden. Der eine Modus entspricht dem status quo und bricht die Anwendung der taktischen Regel auf elementarere Regeln herunter. Der andere Modus erzeugt nun zunächst nur eine (z.B. textuelle) Darstellung der Taktikanwendung, die dazu führt, daß diese Anwendung den Übergang zu einer neuen Tiefenebene markiert. Die Pflicht zur Erzeugung dieser Darstellung wird nun von der aufrufenden zur aufgerufenen Taktik verlagert; der Aufwand zur Programmierung dieses Effektes wird dadurch deutlich geringer. Nicht jede Taktik muß den neuen Ausführungsmodus unterstützen, es wäre dann einfach so, daß eine Taktik, die dies nicht vorsieht, den speziellen Ablaufmodus an die unterliegenden Taktiken weitergibt. Auf diese Weise kann man Taktiken so entwerfen, daß die Einfügung der Tiefenstruktur automatisch an den sinnvollen Stellen geschieht.

Um etwas konkreter zu werden: Eine Taktik T kann man nun im textuellen Modus ausführen, wenn man sie so aufruft: `TextualMode` T. Der Unterschied ist nun der folgende:

- *Ohne* textuellen Modus würde T intern zwar weitere Taktiken benutzen, dies würde aber nicht sichtbar werden, da deren Anteile am Gesamtbeweis sofort in diesen integriert werden. Man erhielte einen Beweis, der sich aus primitiven Regeln zusammensetzt.

- *Mit* textuellem Modus erzeugt T zunächst eine Darstellung seines eigenen Aufrufs und gibt dies als Ergebnis zurück. Diese Darstellung wird nun in den Beweisbaum eingetragen. Um die Wirkung von T zu erhalten, wird diese Darstellung dann aber sofort wieder interpretiert, wodurch T im normalen Modus ausgeführt wird. Dabei wird nun ein innerer Beweisbaum erzeugt, der aber nur hinsichtlich seiner Wirkung, d.h. seiner offenen Unterziele, in den ursprünglichen Baum eingetragen wird.

Eventuell möchte der Benutzer nun auch die inneren Beweisbäume ansehen. Der Beweiseditor muß dazu so abgewandelt werden, daß man auch in die Richtung der Tiefe den Betrachtungsfokus verschieben kann, etwa indem eine weitere Editorinstanz mit dem inneren Baum erzeugt wird (diese Instanz wäre dann aber „read only“).

Wenn man dies schon vorsieht, könne man auch gleich eine weitere Verbesserung einfügen. Häufig ist es so, daß man einen Beweisteil entwickelt, aber diesen Teil der besseren Lesbarkeit wegen wieder in einer Regel zusammenfassen möchte. Hierzu sollte es erlaubt sein, Unterbeweisbäume interaktiv zu erzeugen und anstatt einer Regel einzusetzen.

5.4 Ausblick: Eine Plattform zur interaktiven Synthese

Der Kalkül der Typentheorie ist universell einsetzbar; wir haben ihn in dieser Arbeit zwar für eine bestimmte Anwendung genutzt, er ist jedoch darauf nicht fixiert. Für unseren Zweck, verifizierte Programme zu generieren, heißt dies, daß wir grundsätzlich auch in der Lage sind, andere Methoden als die Globalsuche dazu zu benutzen und diese typentheoretisch darzustellen. Das NuPRL-System könnte auf diese Weise zu einer vielseitigen Plattform ausgebaut werden, die die Anwendung der unterschiedlichsten Methoden erlaubt.

Wir wollen diesen Gedanken noch etwas weitertreiben. Wir haben bei der Globalsuche gesehen, daß diese Methode alleine noch nicht zum Erfolg führt; man erhält Unterprobleme, die wiederum zu lösen sind. An dieser Stelle können wir ansetzen, um die verschiedenen Methoden miteinander zu kombinieren. Dazu benötigen wir allerdings eine einheitliche Darstellung für Probleme; ein Kandidat dazu ist unsere Formalisierung einer Spezifikation. Wenn man nun eine Synthesemethode anwendet, ist der Ausgangspukt immer eine Spezifikation, die zu erfüllen ist, und die Methode endet an definierten Stellen, an denen sie wiederum Spezifikationen erzeugt, die von anderen Methoden zu lösen sind.

Wir haben die Globalsuche bereits in diese Form gebracht. Die Beschreibung der noch zu findenden Algorithmen für O und *theta* wurde explizit jeweils als Spezifikation notiert. Damit sind hier bereits die Voraussetzungen gegeben, die die Einbettung in den Plattformgedanken ermöglicht.

NuPRL besitzt im Augenblick lediglich einen Beweiseditor als die einzige Möglichkeit zur Interaktion mit dem Benutzer. Für ein Synthesesystem ist dies unzulänglich; es werden bessere Interaktionsmöglichkeiten (z.B. Menüs) benötigt, die den Benutzer anleiten. Desweiteren ist zu überlegen, ob es nicht besser wäre, einen eigenen Datentyp zur Darstellung von Synthesen anstelle von `proof` zu verwenden (der freilich auf `proof` abbildbar bleibt). Dies könnte etwa eine Art Skript sein, das eine Mitschrift der Interaktionen darstellt. Der Benutzer kann nun in den Syntheseprozeß, der ja nach wie vor eine Taktik ist, die einen Beweis konstruiert, an bestimmten Punkten eingreifen und auswählen, was er an dieser Stelle für richtig hält. Das Skript protokolliert diese Entscheidung. Damit kann im Nachhinein stets nachvollzogen werden, wie ein Verifikationsbeweis zustande gekommen ist.

Ein Skript würde nun immer damit beginnen, daß die zu lösende Spezifikation protokolliert würde. Es wäre damit eine Ausprägung, wie die Probleme auf einheitliche Weise dargestellt werden können. Das Skript mündet in die Beschreibungen der Subprobleme, die durch Subskripte gelöst werden. Auf dem Weg von der Ausgangsspezifikation zu den Unterspezifikationen werden die benutzten Taktiken sowie die interaktiven Eingriffe des Benutzers notiert.

Skripte wären damit eine Möglichkeit, etwas freiere Interaktionen zu erlauben, ohne auf die Reproduzierbarkeit des Ergebnisses zu verzichten.

Wie man sieht, gibt es auf unterschiedlichen Ebenen noch Möglichkeiten, das derzeitige System zu verbessern. Die Kritik der NuPRL-Basis lag in dieser Diplomarbeit verständlicherweise die ganze Zeit außerhalb der Betrachtung, weil eine bestimmte Anwendung verfolgt wurde und natürlich die dieser Benutzung von NuPRL eigenen Probleme am brennendsten sind. Wenn wir hier auch NuPRL selbst kritisieren, dann sicherlich auch in der Hoffnung, daß einige Punkte in zukünftigen Versionen berücksichtigt werden.

Abbildungsverzeichnis

Literaturverzeichnis

[constable et al.86] Constable, Robert L.; Allen, Stuart F.; Bromley, H. Mark; Cleaveland, W. Rance; Cremer, J. F.; Harper, Robert W.; Howe, Douglas J.; Knoblock, Todd B.; Mendler, Nax Paul; Panangaden, Prakash; Sasaki, Jim T. and Smith, Scott F. Implementing Mathematics with the NuPRL proof development system, Englewood Cliffs NJ 1986

[jackson94b] Jackson, Paul. The Nuprl Proof Development System. Reference manual and Users's Guide (Version 4.1), 1994

[jackson95] Jackson, Paul. The Nuprl Proof Development System. Reference manual and Users's Guide (Version 4.2), 1995

[jackson94a] Jackson, Paul. Nuprl's Metalanguage ML. Reference Manual and User's Guide, 1994

[kant90] Kant, Immanuel. Kritik der reinen Vernunft. 3. Auflage, Meiner Hamburg 1990. Neuausgabe herausgegeben von Raymond Schmidt.

[kreitz92] Kreitz, Christoph. METASYNTHESIS: Deriving Programs that Develop Programs. Habilitationsschrift, Fachgebiet Intellektik TH Darmstadt 1992. Forschungsbericht AIDA-93-03.

[kreitz94] Kreitz, Christoph. Automatisierte Logik und Programmierung, Vorlesungsskript WS 1994/95 und SS 1995, Technische Hochschule Darmstadt (unveröffentlicht)

[martin-löf84] Martin-Löf, Per. Intuitionistic Type Theory, Napoli 1984

[smith87] Smith, Douglas R. Structure and design of global search algorithms. Technical Report KES.U.87.12, Kestrel Institute, November 1987.

[smith91] Smith, Douglas R. KIDS – a knowledge-based software development system. In: Lowry, Michael R. and McCartney, Robert D. (eds), Automating Software Design, pp. 483-514, Menlo Park CA 1991

[stolpmann96] Stolpmann, Gerd. Datentypen und Programmsynthese. Untersuchungen zur typentheoretischen Konzeption von Datentypen und die exemplarische Implementierung im Beweiseditor NuPRL. Studienarbeit am Fachgebiet Intellektik, TH Darmstadt 1996

Anhang A

Taktiken zur Typprüfung und Typbestimmung

Die Implementierung der Typprüfung teilt sich in zwei Sektionen: Zum einen die Verwaltung bestimmter Ressourcen, zum anderen den eigentlichen Algorithmus. Wir skizzieren beides nur kurz.

A.1 Verwaltung von Datentypen, Typkonstruktoren und Typisierungsansätzen

A.1.1 Datentypen

Als ein Datentyp wird ein Paar $\langle T, eq\rangle$ eines Trägertyps T und eines Algorithmus' eq zur Entscheidung der Gleichheit für Werte dieses Typs bezeichnet. Im weiteren gehen wir davon aus, daß erstens jeder Trägertyp durch einen eigenen Operator op_R notiert wird und daß zweitens für das Paar $\langle T, eq\rangle$ eine eigene Abstraktion op_D definiert wird. D.h. wir wollen erkennen können, daß der spezifische Datentyp benutzt wird, und zwar daran, daß einer der Operatoren op_R oder op_D in einem Ausdruck vorkommt.

Die Verwaltung der Datentypen hat nun den eigentlichen Zweck, die bijektive Abbildung von op_R auf op_D u.u. durchzuführen. Dazu wird eine Tabelle angelegt, die für jeden Datentyp folgende Einträge aufweist:

1. op_R
2. op_D
3. eine Konversion `dtrepC`
4. die dazu inverse Konversion `repdtC`

Die Konversion `dtrepC` wandelt einen Term der Bauart $(op_D(...))^T$ in einen äquivalenten der Bauart $op_R(...)$ um. Die Konversion `repdtC` führt die Umwandlung in umgekehrter Richtung durch.

Der Zweck dieser beiden Konversionen ist, zwei Terme ineinander überführbar zu machen, die denselben Typ auf verschiedene Weise notieren, indem der eine Term an entsprechender Stelle einen Datentypoperator benutzt, an der der andere einen Trägeroperator verwendet. Diese Überführung kann von zwei Konversionen, die auf der Datentyptabelle aufbauen, geleistet werden: `norm_type_at_toplevelC` und `rev_norm_type_at_toplevelC`. Erstere wandelt einen Typausdruck so um, daß der äußerste Operator derjenige für $\cdot^T$ ist. Der nächstinnere ist dann ein Datentypoperator. Zweitere wandelt einen Typausdruck so um, daß der äußerste Operator ein Trägeroperator ist.

Zur Zeit sind folgende Datentypen in die Tabelle eingetragen:

Bezeichung	Datentypoperator	Trägeroperator
ganze Zahlen	$\mathbf{Z}\uparrow$ (int_type)	$\mathbf{Z}$ (int)
boole'sche Algebra	$\mathbf{B}\uparrow$ (bool_type)	$\mathbf{B}$ (bool)
unabhängige Produkte	$\alpha \times \beta$ (product_type)	$S \times T$ (product)
Sequenzen	Seq(α) (seq_type)	seq(α) (seq)
endliche Mengen	Set(α) (fset_type)	set(α) (fset)

A.1.2 Typkonstruktoren

In dieser Tabelle werden die Daten bereitgehalten, die zum Typisieren von Typausdrücken benötigt werden. Der Typ eines Typs ist stets ein Universum U_j oder TYPES, der Typ der Datentypen. TYPES hat eine Bedeutung, die den Universen im engeren Sinne gleichkommt und soll daher hier ebenfalls als Universum bezeichnet werden.

Typkonstruktoren sind zum einen die Operatoren der Datentyptabelle (und zwar jeweils sowohl Datentyp- als auch Trägeroperator), als auch diejenigen Operatoren von Typtermen, die nicht zu den Datentypen gehören, wie etwa der Funktionenraum. Es können nur Typkonstruktoren berücksichtigt werden, die keine Bindungsvariablen einführen, d.h. abhängige Typen werden hier nicht betrachtet[1].

Ein Eintrag in der Typkonstruktor-Tabelle enthält nun folgende Angaben:

1. den Operator *op* des Typkonstruktors
2. eine Liste *arg_univ*, die für jeden Teilterm des Typkonstruktors angibt, welchem Universum er zugerechnet werden muß
3. die Angabe *op_univ*, die das Universum bezeichnet, in dem der Typkonstruktor dann liegt
4. einen Generator g, der in der Lage ist, ein Element des beschriebenen Typs zu erzeugen, wenn er Elemente für die Typen aller Teilterme als Argumente bekommt

Im Augenblick müssen für *arg_univ* und *op_univ* jeweils feste Universen angegeben werden, obwohl viele Typkonstruktoren auch zuließen, daß der Level des Ergebnisuniversums in Abhängigkeit von den Leveln der Teiltermuniversen spezifiziert werden. In einer zukünftigen Version der Tabelle müßte dieses Feature ergänzt werden.

Mit den Einträgen der Tabelle kann nun überprüft werden, ob Typausdrücke korrekt gebildet sind. Dazu wird versucht, allen Teiltermen des Ausdrucks ein Universum zuzuordnen. Gelingt dies, kann der Ausdruck passieren. Ansonsten wird er zurückgewiesen; freilich ist diese Prüfung strenger als notwendig, auch korrekte Typausdrücke werden zurückgewiesen, wenn sie nicht der Tabelle entsprechen.

Mittels des jeweils vorhandenen Generators kann zu jedem passablen Typ ein konkretes Element erzeugt werden. Dies ist zum Beispiel notwendig, wenn man eine Formel der Art $\exists x{:}T....$ beweisen will und x im Inneren gar nicht mehr vorkommt. Ist T ein passabler Typ, kann man sich nun ein Element x_0 generieren lassen und für x einsetzen.

A.1.3 Typisierungsansätze

Zu jedem Elementoperator wird in dieser Tabelle ein Typisierungsansatz bereitgehalten. Die Operatoren können sowohl primitive als auch abstrakte Operationen bezeichnen. Die Speicherung der Ansätze folgt im wesentlichn dem im Haupttext angegebenen Schema; wir verzichten hier daher auf eine nochmalige Erläuterung.

Jeder Eintrag der Tabelle enthält nun folgende Angaben:

[1]Die unabhängigen Varianten des Funktionen- und Produktraums verwenden in NuPRL stets eine spezielle Bindungsvariable, die *Nullvariable*, die selbst nicht eingegeben werden kann. So ist es möglich, diese Varianten sofort zu erkennen.

1. den Namen des Operators op
2. den Typisierungsansatz
   $A = D: \quad v_1 \in \hat{V}_1, ..., v_m \in \hat{V}_m, s_1 \in \hat{S}_1, ..., s_n \in \hat{S}_n \stackrel{*}{\Longrightarrow} op(v_1, \ldots .s_1; ...; ...v_m.s_n) \in \hat{S}$
   (wobei natürlich nur die Typterme $\hat{V}_i$, $\hat{S}_i$ und $\hat{S}$ gespeichert werden, die ja bereits durch die Position innerhalb des op-Terms in ihren Zusammenhang rücken; die Variablen v_i und s_i sind hier nur der lesbaren Darstellung geschuldet.)
3. die Universenergänzung U des Ansatzes: zu jedem Term $\hat{V}_i$, $\hat{S}_i$ und $\hat{S}$ wird das Universum gespeichert
4. die Typisierungstaktik T
5. die Liste der Hauptargumente H
6. das Niveau N des Operators

Es ist noch zu beachten, daß einige Operatoren nicht ganz in das Schema für die Typisierungsansätze passen. So hat etwa die Operation für das Enthaltensein eines Elementes in einer endlichen Menge $a \in M : \texttt{Set}(\alpha)$ neben den gewöhnlichen Argumenten a und M noch eine Typisierungsangabe α. Gemäß dem Schema müßte man als α-Typ `TYPES` einsetzen; dadurch wäre es jedoch nicht mehr möglich, auf α selbst zu referieren, etwa im Typ für M, der ja als $\texttt{Set}(\alpha)$ angegeben werden muß. Im Augenblick wird dies dadurch gelöst, daß man ein Argument des Operators speziell markieren kann. Die Typangabe S_i, die für dieses Argument gespeichert wird, wird dann nicht mehr so interpretiert, daß eine Instanz des Arguments in diesem Typ enthalten sein muß, sondern so, daß die Instanz diesen Typ selbst ausdrückt. Eine andere Möglichkeit wäre, den Instanzen selbst auch Namen geben zu können, was in dem Fall, daß es sich um Typen handelt, ja durchaus Sinn macht.

Im Augenblick ist es so, daß in dieser Tabelle wie auch der für die Typkonstruktoren nur feste Universen gespeichert werden können. Eine Erweiterung tut dringend not.

Es wäre auch zu überlegen, beide Tabellen zu einer einzigen zusammenzufassen. Manche Angaben können dann zwar nur für Element-, manche nur für Typoperatoren gemacht werden, aber die meisten wären für beide Fälle möglich.

Die Typisierungstaktik wird nur benötigt, wenn ein Typisierungsbeweis konstruiert werden soll. Diese Taktik wird mit Argumenten p_1,...,p_k aufgerufen, die die konkreten Instanzen für die Typvariablen angeben, die in der Deklaration D des Typisierungsansatzes vorkommen. Die Taktik wird auf eine Sequenz der Gestalt ... $\vdash op(B_1.t_1, ..., B_n.t_n) \in T$ angesetzt; sie kann Untersequenzen erzeugen, die mit 'type-wf', 'subterm i' oder 'antecedent' markiert sind. Die mit 'type-wf' gekennzeichneten Ziele sollten die Form ... $\vdash u \in U$, wobei U ein Universum ist, haben; die Ziele 'subterm i' sollten die Form ... $\vdash t_i \in T_i$ haben, wobei in den Hypothesen die Bindungsvariablen aus B_i geeignet deklariert sind; konkrete Propositionen, die vorausgesetzt werden müssen, damit die Typisierung klappt, sollten mit 'antecedent' markiert sein.

Falls es für den Operator op ein Wf-Lemma gibt, kann man eine generische Typisierungstaktik aufrufen, die dieses Lemma automatisch anwendet. Auch für primitive Operatoren gibt es eine Sammlung von Taktiken, die diese typisieren können.

Die Hauptargumente H sind diejenigen, die nach der Auswertungsstrategie *lazy evaluation* bevorzugt reduziert werden. H hat eigentlich nichts in dieser Tabelle zu suchen, sondern sollte einer eigenen Tabelle assoziiert werden.

Mit den Niveaus N verhält es sich so, daß Operatoren mit höherem Niveau mittels solcher Operatoren mit niedrigerem Niveau definiert werden. Primitive Operatoren haben stets das Niveau 0. Die Zuordnung von Niveaus ist nützlich, wenn man abstrakte Operatoren dadurch bearbeiten will, daß man sie auffaltet; dies kann sowohl für Typisierungen nützlich sein als auch für andere Zwecke.

Überigens kann man alle für die Typisierung erforderlichen Angaben aus dem jeweiligen Wf-Lemma ableiten. Es ist daher möglich, diesen Teil der Tabelle automatisch aufzubauen, d.h. wann immer ein unbekannter Operator entdeckt wird, sucht die Typprüfungstaktik nach einem passenenden Wf-Lemma, mit dessen Hilfe der fehlende Eintrag gemacht wird.

A.2 Typprüfung, Typbestimmung

Basis aller Typprüfung ist eine Funktion, die, ein gegebenes Environment (Deklaration freier Variablen) vorausgesetzt, jedem Teilterm des zu typenden Terms einen Typ zuordnet. Noch mehr, wird auch jeder vorkommenden Bindungsvariable ein Typ zugewiesen. Dabei kann es passieren, daß nicht alle Typteile konkretisiert werden können; in diesem Fall bleiben in den Typen noch Typvariablen stehen, die mit einem Universum deklariert sind.

Basis dieses Algorithmus' sind die vorangehend erläuterten Tabellen.

Die augenblickliche Implementation ist noch nicht so strukturiert, wie im Haupttext beschrieben. Die gefundenen Typgleichungen werden sofort als Substitutionen aufgefaßt, die immer sofort angewendet werden. Aufstellung und Lösung des TGS sind mithin nicht getrennt. In zukünftigen Implementierungen sollte diese Strukturierung jedoch eingehalten werden, schon deshalb, weil man dann den Teil für die Auflösung des TGS optimieren kann. Weitere Gründe sind die Entzerrung des Algorithmus, seine Erweiterbarkeit (z.B. um Typungleichungen oder ganz andere Sorten von Bedingungen) und seine Modularisierung (so könnte das TGS vorverarbeitet werden, bevor es aufgelöst wird; oder zwei TGS könnten vereinigt werden u.ä.).

Für den Algorithmus' stehen zahlreiche Interface-Funktionen zur Verfügung, die selektiv einige Informationen auswählen, die der Algorithmus erzeugt.

Idee für Neuimplementierung: Es sollte ein ML-Typ `typed_term` *eingeführt werden, der einen Term mit angefügten Typen aller Teilterme darstellt. Einige aufbauende ML-Typen, wie etwa getypte Konversionen und Equalizer, könnten so auf abstrakte Weise Typinformationen nutzen, ohne einen bestimmten Algorithmus zur Typprüfung kennen zu müssen.*

A.3 Konstruktion von Typisierungsbeweisen

In der Tabelle für die Typisiserungsansätze wird auch eine Taktik eingetragen, die spezifisch für jeden Operator den entsprechenden Teil eines Typisierungsbeweises konstruiert. Mit Hilfe der Informationen, die der Typprüfungsalgorithmus liefert ist es nun auch möglich, einen kompletten Typisierungsbeweis zu erzeugen.

Es wird dabei freilich angenommen, daß Kontextbedingungen keine Rolle spielen; diese werden zur Zeit nämlich nicht generiert. Die von den Einzeltaktiken zurückkommenden 'antecedent'-Ziele müssen daher ohne Kontextwissen beweisbar sein, damit der Einsatz dieser Taktik sinnvoll ist. Wie bereits erläutert, muß die Bereitstellung des Kontextwissens mit dem Typisierungsbeweis verzahnt werden, was im Augenblick nicht geleistet wird.

Es ist ggf. zu überlegen, ob nicht es besser ist, statt den Typisierungsbeweis sofort aus den Typprüfungsinformationen zu erzeugen, zunächst einen Zwischenschritt zu machen, indem nur notiert wird, wie die zu den Operatoren gehörigen Einzeltaktiken aufgerufen werden müßten, ohne dies jedoch direkt zu tun. Dadurch hätte man die Chance, den Beweis zunächst umzustrukturieren, um etwa Kontextkonstruktionen einzuflechten, bevor man ihn in seine endgültige Form gießt.

Anhang B

Taktiken für Programmschemata

Vorbemerkung: Die im folgenden beschriebenen Taktiken sind nur zum Teil implementiert worden. Die Details können dabei geringfügig abweichen; es geht hier mehr darum, die Idee dieser Taktiken darzustellen.

B.1 Definition

Ein Programmschema ist eine Formel der Art

$$\forall f_1.\ f_1 \text{ satisfies } S_1 \Rightarrow \forall f_2.\ f_2 \text{ satisfies } S_2 \Rightarrow ...\forall f_n.\ f_n \text{ satisfies } S_n \Rightarrow t[f_1, ..., f_n] \text{ satisfies } S$$

Dabei sind $S_1, ..., S_n$ Terme von Spezifikationen für die mit den Variablen $f_1, ..., f_n$ bezeichneten Programmkörper. Ein Programmschema drückt aus, daß unter der Annahme, daß f_i die Spezifikation S_i erfüllt (für alle i), der Term t ein korrekter Programmkörper für S ist.

Die Variablen $f_1, ..., f_n$ dürfen nur in t vorkommen, jedoch nicht in den Spezifikationen.

Es wird folgende spezielle Schreibweise für Programmschemata vorgeschlagen:

$$\begin{array}{l}
\texttt{SCHEME} \\
\quad f_1 \text{ satisfies } S_1 \\
\quad f_2 \text{ satisfies } S_2 \\
\quad \ldots \\
\quad f_n \text{ satisfies } S_n \\
\Rightarrow \\
\quad t[f_1, ...f_n] \text{ satisfies } S \\
\texttt{END}
\end{array}$$

B.2 Abstrakte taktische Regeln

Schreibweise: $^{(i:)}$ bezeichnet die Position der i-ten Hypothese; (i) meint die i-te Hypothese das Hauptziels.

B.2.1 Grundregeln: pschemeI, pschemeI_triv, pschemeE, pschemeE_triv

$\Gamma \quad \vdash \quad$ `SCHEME`
$\quad f_1$ satisfies S_1
$\quad f_2$ satisfies S_2
$\quad \ldots$
$\quad f_n$ satisfies s_n
$\Rightarrow$
$\quad t$ satisfies S
`END`

BY pschemeI

Γ, f_1 satisfies $S_1 \quad \vdash \quad$ `SCHEME`
$\quad f_2$ satisfies S_2
$\quad \ldots$
$\quad f_n$ satisfies s_n
$\Rightarrow$
$\quad t$ satisfies S
`END`

– – wf – –
$\Gamma \quad \vdash \quad S_1 \in$ `SPECS`

$\Gamma \quad \vdash \quad$ `SCHEME`
$\quad -$
$\Rightarrow$
$\quad t$ satisfies S
`END`

BY pschemeI_triv

$\Gamma \quad \vdash \quad t$ satisfies S

$\Gamma, {}^{(i:)}$`SCHEME` $, \Delta \quad \vdash \quad C$
$\quad f_1$ satisfies S_1
$\quad f_2$ satisfies S_2
$\quad \ldots$
$\quad f_n$ satisfies s_n
$\Rightarrow$
$\quad t$ satisfies S
`END`

BY pschemeE i s

– – antecedent – –
$\Gamma, (i), \Delta \quad \vdash \quad s$ satisfies S_1

$\Gamma, (i), \Delta,$ `SCHEME` $\quad \vdash \quad C$
$\quad f_2$ satisfies S_2
$\quad \ldots$
$\quad f_n$ satisfies s_n
$\Rightarrow$
$\quad t[s/f_1]$ satisfies S
`END`

– – wf – –
$\Gamma, (i), \Delta \quad \vdash \quad s \in$ `BODYS`(S_1)

$\Gamma,$ `SCHEME` $\quad \vdash \quad C$
$\quad -$
$\Rightarrow$
$\quad t$ satisfies S
`END`

BY pschemeE_triv i

Γ, t satisfies $S \quad \vdash \quad C$

B.2.2 Allgemeine Instanziierungsregel: pschemeInstHyp

Mit folgender Regel kann man in einem Programmschema an beliebiger Stelle Algorithmen einsetzen:

$\Gamma, ^{(i:)}$`SCHEME` $, \Delta \quad \vdash \quad C$

f_1 satisfies S_1
f_2 satisfies S_2
...
f_n satisfies S_n
$\Rightarrow$
t satisfies S
`END`

BY pschemeInstHyp $[f_{\lambda_1}, s_1, ...f_{\lambda_m}, s_m]\ i$

– – antecedent – –
$\Gamma, (i), \Delta \quad \vdash \quad s_1$ satisfies S_{λ_1}
$\vdots$

– – antecedent – –
$\Gamma, (i), \Delta \quad \vdash \quad s_m$ satisfies S_{λ_m}

$\Gamma, (i), \Delta,$ `SCHEME` $\quad \vdash \quad C$
...
f_k satisfies $S_k; \quad k \neq \lambda_j$
...
$\Rightarrow$
$t[s_1, ..., s_m/f_{\lambda_1}, ..., f_{\lambda_m}]$ satisfies S
`END`

– – wf – – (i.a. mehrere wf-Ziele des Typs:)
$\Gamma, (i), \Delta \quad \vdash \quad S_k \in$ `SPECS`

– – wf – – (i.a. mehrere wf-Ziele des Typs:)
$\Gamma, (i), \Delta \quad \vdash \quad s_k \in$ `BODYS`(S_{λ_k})

B.2.3 Automatische Instanziierung: pschemeBackThruHyp, pschemeConstrHyp, pschemeFwdThruHyp

Analog zu BackThruHyp/Lemma und FwdThruHyp/Lemma kann man auch für Programmschemata Taktiken entwerfen, die eine gegebene Konklusion oder Hypothese gegen ein Programmschema matchen und dieses automatisch instanziieren. Es ist vermutlich am klügsten, diese Taktiken gleich so auszulegen, daß sie mit beliebigen Mischungen aus universeller Formel und Programmschema zurechtkommen. In diesem Fall ergibt sich die verlockende Möglichkeit, die Gültigkeit eines Programmschemas von Axiomen abhängig zu machen, d.h. zu Algorithmenschemata zu verallgemeinern. Die Axiome könnten in der universellen Formel als Voraussetzungen genannt werden und das Programmschema ihr Resultat sein. Automatische Instanziierung hieße nun, daß ein gegebenes Problem gegen das Algorithmenschema gematcht wird und daß man als Unterziele die noch zu lösenden Teilprobleme und die vorausgesetzten Axiome erhält.

Unterscheide:

- pschemeBackThruHyp: Anwendbar auf Ziele $\Gamma \vdash t$ satisfies S, matcht dieses Ziel gegen die gennante Hypothese und erzeugt als Unterziele Teilprobleme und Axiome
- pschemeConstrHyp: Anwendbar auf Ziele $\Gamma \vdash S$ is satisfiable
- pschemeFwdThruHyp: Erzeugt eine neue vereinfachte Hypothese, in denen die gefundenen Voraussetzungen nicht mehr vorkommen; genauer: Alle Axiome und eine Auswahl der Teilprobleme müssen gefunden werden

Anhang C

Taktiken für den Produktraum

C.1 Der spreadn-Operator

NuPRL stattet den Operator `spreadn` mit einer speziellen Bedeutung aus, die fest definiert ist und die nicht durch eine gewöhnliche Abstraktion erreicht werden kann. Dieser Operator unterstützt in besonderer Weise Mehrfachprodukte, die einen Typ der Form

$$x_1{:}\,S_1 \times (x_2{:}\,S_2 \times (\cdots \times (x_{n-1}{:}\,S_{n-1} \times S_n)\cdots))$$

haben.

Mehrfachprodukte sind zunächst unproblematisch: Ihre Typen können durch Verschachtelung des einfachen Produkts leicht angegeben werden; Mehrfachpaare (n-Tupel) können durch entsprechende Verschachtelung des `pair`-Operators ebenso simpel erzeugt werden. Umständlich ist jedoch die Analyse eines n-Tupels. Soll der `spread`-Operator hierzu herangezogen werden, müssen jeweils zusätzlich Hilfsvariablen benutzt werden, um das Tupel sukzessive zu zerlegen.

Es bietet sich daher an, für bestimmte n erweiterte `spread`-Abstraktionen zu definieren, die diese Details verbergen. Beispielsweise sähe für $n = 3$ der `spread3`-Operator wie folgt aus:

$$\texttt{spread3}(e;\ x_1, x_2, x_3.t^3[x_1, x_2, x_3]) \quad \equiv \quad \text{let } \langle x_1, h_1\rangle = e \text{ in let } \langle x_2, x_3\rangle = h_1 \text{ in } t^3[x_1, x_2, x_3]$$

Eine Display-Form für diesen Operator wäre naheliegend: let $\langle x_1, x_2, x_3\rangle = e$ in $t^3[x_1, x_2, x_3]$. Wie man sieht, reicht hier eine Hilfsvariable h_1 aus.

Damit man nun nicht für jedes n eine eigene Abstraktion angeben muß, wurde nun `spreadn` vorab so definiert, daß Analysen für beliebiges n durchgeführt werden können. Man kann sich diesen Operator als wie folgt definiert vorstellen:

$$\begin{array}{l}
\texttt{spreadn}(e;\ x_1, ..., x_n.t^n[x_1, ..., x_n]) \\
\equiv\ \ \text{let } \langle x_1, h_1\rangle = e \text{ in} \\
\qquad \text{let } \langle x_2, h_2\rangle = h_1 \text{ in} \\
\qquad\quad \cdots \\
\qquad\qquad \text{let } \langle x_{n-1}, x_n\rangle = h_{n-2} \text{ in } t^n[x_1, ..., x_n]
\end{array}$$

Wendet man den *Unfold*-Metaoperator auf einen `spreadn`-Term an, wird diese Definition herangezogen, um den aufgefalteten Term zu bestimmen. Im Gegensatz zu einer üblichen Definition besitzt dieser Operator als zweites Argument eine variable Zahl von Bindungsvariablen. Dies wird vom NuPRL-Abstraktionsmechanismus eigentlich nicht unterstützt und ist hier nur in Form einer Ausnahme implementiert.

(Leider beschränkt sich dies i.w. auf den `spreadn`-Operator, noch nicht einmal die „nächsten Verwandten" werden unterstützt. So gibt es keine allgemeinen Projektionsfunktionen a la `pi1` und `pi2`.)

C.2 Taktische Regel: Elimination einer Paargleichung

Folgende Regel erweitert den Regelsatz für den einfachen Produktraum. Siehe auch im Haupttext Abschnitt 2.2.3, insbesondere Seite 58.

$\Gamma,\ \langle s_1, t_1\rangle = \langle s_2, t_2\rangle \in x{:}S \times T,\ \Delta \quad \vdash \quad C$
BY pair_eqE i U_j

$\Gamma,\ \langle s_1, t_1\rangle = \langle s_2, t_2\rangle \in x{:}S \times T,\ \Delta \quad \vdash \quad S \in U_j$

$\Gamma,\ \langle s_1, t_1\rangle = \langle s_2, t_2\rangle \in x{:}S \times T,\ s_1 = s_2 \in S,\ \Delta \quad \vdash \quad T[s_1/x] \in U_j$

$\Gamma,\ \langle s_1, t_1\rangle = \langle s_2, t_2\rangle \in x{:}S \times T,\ \Delta \quad \vdash \quad s_1 \in S$

$\Gamma,\ \langle s_1, t_1\rangle = \langle s_2, t_2\rangle \in x{:}S \times T,\ s_1 = s_2 \in S,\ \Delta \quad \vdash \quad t_1 \in T[s_1/x]$

$\Gamma,\ \langle s_1, t_1\rangle = \langle s_2, t_2\rangle \in x{:}S \times T,\ s_1 = s_2 \in S,\ t_1 = t_2 \in T[s_1/x],\ \Delta \quad \vdash \quad C$

C.3 Taktische Regeln für Mehrfachprodukte

Die Regeln `spreadEq`, `productE` und `pair_eqE` wurden auf folgende Weise für Mehrfachprodukte erweitert:

C.3.1 Regel `spreadnEq`

$\Gamma \quad \vdash \quad \texttt{spreadn}(e;\ x_1, ..., x_n.s^n[x_1, ..., x_n]) = \texttt{spreadn}(f;\ y_1, ..., y_n.t^n[y_1, ..., y_n]) \in S$
BY spreadnEq '$v_1{:}T_1 \times v_2{:}T_2 \times \cdots \times v_{n-1}{:}T_{n-1} \times T_n$' U_j

$\Gamma \quad \vdash \quad e = f \in (v_1{:}T_1 \times v_2{:}T_2 \times \cdots \times v_{n-1}{:}T_{n-1} \times T_n)$

$\Gamma,\ u_1{:}T_1,\ ...,\ u_n{:}T_n,\ h{:}\, e = \langle u_1, ..., u_n\rangle \in (v_1{:}T_1 \times v_2{:}T_2 \times \cdots \times v_{n-1}{:}T_{n-1} \times T_n)$
$\vdash \quad s^n[u_1, ..., u_n/x_1, ..., x_n] = t^n[u_1, ..., u_n/y_1, ..., y_n] \in S$

weitere Ziele, die die Wohlgeformtheit sicherstellen

C.3.2 Regel `productnE`

$\Gamma,\ z{:}\,(x_1{:}S_1 \times \cdots x_{n-1}{:}S_{n-1} \times S_n),\ \Delta \quad \vdash \quad C$
BY productnE i n

$\Gamma,\ z{:}\,(x_1{:}S_1 \times \cdots \times x_{n-1}{:}S_{n-1} \times S_n),$
$s_1{:}S_1,\ ...,\ s_n{:}S_n([s_1/x_1]\cdots[s_{n-1}/x_{n-1}]),\ \Delta[\langle s_1, ..., s_n\rangle/z]$
$\vdash \quad C[\langle s_1, ..., s_n\rangle/z]$

C.3.3 Regel pairn_eqE

$\Gamma,\ \langle s_1, ..., s_n\rangle = \langle t_1, ..., t_n\rangle \in (x_1{:}\, t_1 \times \cdots \times x_{n-1}{:}\, T_{n-1} \times T_n),\ \Delta \quad \vdash \quad C$

BY pairn_eqE $i\ n\ U_j$

zahlreiche Wohlgeformtheitsziele

$\Gamma,\ \langle s_1, ..., s_n\rangle = \langle t_1, ..., t_n\rangle \in (x_1{:}\, t_1 \times \cdots \times x_{n-1}{:}\, T_{n-1} \times T_n),$
$s_1 = t_1 \in T_1,\ ...,\ s_n = t_n \in T_n([s_1/x_1] \cdots [s_{n-1}/x_{n-1}]),\ \Delta$
$\vdash \quad C$

C.4 Anwendungen: Zugriffsoperationen für Spezifikationen und Globalsuchräume

Produkte kommen in dieser Arbeit vor allem an zwei Stellen vor: Bei der Darstellung von Spezifikationen und von Globalsuchräumen. Die folgenden Ausführungen beziehen sich nur auf Spezifikationen, können jedoch leicht auf den anderen Fall übertragen werden.

Eine konkrete Spezifikation wird als 4-Tupel $\langle D, R, I, O\rangle$ repräsentiert. Um auf die vier Anteile zugreifen zu können, werden vier spezielle Zugriffsoperationen (Projektionen) definiert: `D`(*spec*), `R`(*spec*), `I`(*spec*), `O`(*spec*). Dies ist leicht, wenn man den `spreadn`-Operator benutzt, beispielsweise

$$\texttt{I}(spec) \quad \equiv \quad \texttt{spreadn}(spec;\ D, R, I, O.I)$$

Es stellt sich allerdings heraus, daß die Sache noch einen Haken hat. Geht man nämlich naiv an den Beweis des entsprechenden Wf-Theorems $\forall spec.\ \texttt{I}(spec) \in \texttt{D}(spec) \to \mathbf{P}$ heran, bleibt man schnell stecken. Insbesondere die Autotaktik berücksichtigt nämlich nicht, daß es hier darauf ankommt, die in einer Spezifikation vorhandene Abhängigkeit der hinteren beiden Elemente I und O von den vorderen beiden D und R auszunutzen. Der Leser probiere nun einmal, auf der Ebene der primitiven Inferenzregeln oder sogar den Autotaktik-Regeln dieses Theorem zu beweisen!

Mit Hilfe der neuen Regeln für den Produktraum kann jedoch leicht wie im folgenden Beweis argumentiert werden (dabei wird $\texttt{SPECS} \equiv D{:}\, \mathbf{U} \times R{:}\, \mathbf{U} \times (D \to \mathbf{P}) \times (D{\times}R \to \mathbf{P})$ vorausgesetzt). Wie man sieht, besteht die Hauptschwierigkeit darin, daß *spec* zweimal durch ein Quadrupel konkretisiert werden muß, nämlich sowohl beim Herunterbrechen des äußersten `spread`-Operators durch `spreadnEq` als auch in der allgemeinen Verwendung in der Konklusion (durch `productnE`). Die beiden Quadrupel $\langle D, R, I, O\rangle$ und $\langle D', R', I', O'\rangle$ sind zwar im Ganzen gleichgesetzt, aber dies läßt sich nicht so einfach nutzen. Durch `pairn_eqE` kann man diese Gleichung jedoch in vier einzelne Gleichungen aufspalten. Dennoch erhält man ein „Mischmasch" von Bedingungen, in denen Anteile beider Quadrupel vorkommen. Man beachte, daß in diesem Beweis die Anwendung der `subst`-Regel der Typentheorie folgenlos ist, man dreht sich nur im Kreis. Dies ist im allgemeinen in Wohlgeformtheitsnachweisen so, weil `subst` immer schon davon ausgeht, daß die Formeln wohlgeformt sind. Dieser Beweis hier wird daher durch `replaceHyp` weitergeführt; das sieht etwas umständlich aus, ist aber die einzige Möglichkeit, weiterzukommen. `replaceHyp` ersetzt eine Hypothese durch eine andere, wobei nachzuweisen ist, daß diese gleichwertig ist. Da eine Hypothese ersetzt wird, muß nicht erst nachgewiesen werden, daß diese wohlgeformt ist; Hypothesen sind dies immer.

Dieser Beweis kann als Beispiel dafür dienen, wie kompliziert das Schließen über Gleichheiten werden kann, wenn Typen mit inneren Abhängigkeiten vorkommen.

$spec$: $\mathtt{SPECS}$
$\vdash\quad (\text{let } \langle D,R,I,O\rangle = spec \text{ in } I) \in (\text{let } \langle D,R,I,O\rangle = spec \text{ in } D) \to \mathbf{P}$
BY **spreadnEq** ‘$\mathtt{SPECS}$‘

- $\ldots \vdash\quad spec \in \mathtt{SPECS}$

- $spec\colon \ldots,\ D\colon \mathbf{U},\ R\colon \mathbf{U},\ I\colon D \to \mathbf{P},\ O\colon D{\times}R \to \mathbf{P},\ h\colon spec = \langle D,R,I,O\rangle \in \mathtt{SPECS}$
  $\vdash\quad I \in (\text{let } \langle D,R,I,O\rangle = spec \text{ in } D) \to \mathbf{P}$
  BY **productnE** 1 4

  - $spec\colon \ldots,\ D\colon \ldots,\ R\colon \ldots,\ I\colon \ldots,\ O\colon \ldots,$
    $D'\colon \mathbf{U},\ R'\colon \mathbf{U},\ I'\colon D' \to \mathbf{P},\ O'\colon D'{\times}R' \to \mathbf{P},$
    $h\colon \langle D',R',I',O'\rangle = \langle D,R,I,O\rangle \in \mathtt{SPECS}$
    $\vdash\quad I \in (\text{let } \langle D,R,I,O\rangle = \langle D',R',I',O'\rangle \text{ in } D) \to \mathbf{P}$
    BY *Reduktion der Konklusion*

    - $spec\colon \ldots,\ D\colon \ldots,\ R\colon \ldots,\ I\colon \ldots,\ O\colon \ldots,$
      $D'\colon \ldots,\ R'\colon \ldots,\ I'\colon \ldots,\ O'\colon \ldots,\ h\colon \ldots$
      $\vdash\quad I \in D' \to \mathbf{P}$
      BY **pairn_eqE** 10 4 $\mathbf{U}_2$

      - $\ldots \vdash$ Wf-Ziele

      - $spec\colon \ldots,\ D\colon \ldots,\ R\colon \ldots,\ I\colon \ldots,\ O\colon \ldots,$
        $D'\colon \ldots,\ R'\colon \ldots,\ I'\colon \ldots,\ O'\colon \ldots,\ h\colon \ldots$
        $h_1\colon D' = D \in \mathbf{U},\ h_2\colon R' = R \in \mathbf{U},\ h_3\colon I' = I \in D' \to \mathbf{P},$
        $h_4\colon O' = O \in D'{\times}R' \to \mathbf{P}$
        $\vdash\quad I \in D' \to \mathbf{P}$
        BY **replaceHyp** 4 ‘$D' \to \mathbf{P}$‘ $\mathbf{U}_2$

        - $\ldots,\ I\colon D' \to \mathbf{P},\ \ldots$
          $\vdash\quad I \in D' \to \mathbf{P}$
          trivial

        - $\ldots \vdash\quad (D \to \mathbf{P}) = (D' \to \mathbf{P})$
          relativ einfach

Anhang D

Liste der getypten Konversionen

Im folgenden wird eine Aufstellung der implementierten getypten Konversionen geboten. Insgesamt gibt es ja nun drei Konversionsarten: Standard-Konversionen (Suffix C), getypte Konversionen (TC) und addressierende getypte Konversionen (ATC). Zunächst die Umwandlungskonversionen, die zwischen den Arten vermitteln; dann die Liste mit den TC- und ATC-Konversionen.

Folgende Buchstaben werden mit bestimmten ML-Typen benutzt:

c	`convn` (Standard-Konversion)
tc	`typed_convn` (getypte Konversion)
atc	`addr_typed_convn` (addressierende getypte Konversion)
tcl	Liste von getypten Konversionen
$atcl$	Liste von addressierenden getypten Konversionen
T	`term` (Term, der für einen Typausdruck steht)

D.1 Umwandlungskonversionen

`CtoTC` c	faßt c als getypte Konversion auf. Bei der Ausführung von c kann dabei natürlich nicht auf Typinformationen zugegriffen werden.
`TCtoC` T tc	faßt tc als gewöhnliche Konversion auf. Es wird dabei angenommen, daß der Term, auf den tc angewendet wird, den Typ T hat.
`TCtoATC` tc	faßt tc als addressierende Konversion auf.
`addrTCtoATC` f	die Funktion f wird mit $f\ a$ aufgerufen, wobei a die Adresse des Teilterms ist, auf den sich die Konversion bezieht; $f\ a$ muß eine getypte Konversion ergeben.
`ATCtoTC` atc	faßt atc als normale getypte Konversion auf. Hier wird versucht, die Reihe der in atc aufgesammelten Einzelkonversionen zu optimieren.

D.2 Liste der Konversionen

Die Namen richten sich nach den Pendants bei den Standard-Konversionen, nur daß der Suffix geändert wurde. Die Typen der Prädikate, die hin und wieder anzugeben sind, wurden auch geändert (hier nicht dokumentiert).

• `FailTC` `FailATC`	schlägt immer fehl
• `IdTC` `IdATC`	die identische Konversion
• `showTC` tc `showATC` atc	wendet tc (bzw. atc) an und gibt aus, welche Termersetzung durchgeführt wurde.
• `IfTC` p tc `IfATC` p atc	falls das Prädikat p für den Term wahr ist, wird tc/atc darauf angewendet
• `IfIsTC` t tc `IfIsATC` t atc	falls der zu ersetzende Term t ist, wird tc/atc darauf angewendet, sonst Fehlschlag.
• tc_1 `ORELSETC` tc_2 atc_1 `ORELSEATC` atc_2	wendet tc_1/atc_1 an. Nur wenn dies fehlschlägt, wird tc_2/atc_2 angewendet.
• tc_1 `ANDTHENTC` tc_2 atc_1 `ANDTHENATC` atc_2	wendet tc_1/atc_1 zuerst an und dann tc_2/atc_2. Schlägt fehl, wenn eine der Anwendungen fehlschlägt.
• tc_1 `ORTHENTC` tc_2 atc_1 `ORTHENATC` atc_2	wendet tc_1/atc_1 an. Schlägt dies fehl, wird *stattdessen* tc_2/atc_2 angewendet; ansonsten wird *zusätzlich* diese zweite Konversion durchgeführt.
• `TryTC` tc `TryATC` atc	wendet tc/atc an. Schlägt dies fehl, wird stattdessen die Identität durchgeführt.
• `ProgressTC` tc `ProgressATC` atc	wendet tc/atc an. Schlägt fehl, falls dadurch keine Veränderung des Terms erreicht wurde.
• `RepeatTC` tc `RepeatATC` atc	wendet tc/atc wiederholt an, bis keine Veränderung mehr stattfindet oder ein Fehlschlag auftritt, der allerdings absorbiert wird. Erlaubt die nullmalige Wiederholung.
• `Repeat1TC` tc `Repeat1ATC` atc	entsprechend, es muß jedoch wenigstens eine Anwendung stattfinden, sonst Fehlschlag.
• `RepeatForTC` n tc `RepeatForATC` n atc	wendet tc/atc genau n mal wiederholt an.
• `FirstTC` tcl `FirstATC` $atcl$	probiert der Reihe nach alle Konversionen aus $tcl/atcl$ aus. Es wird die erste Konversion durchgeführt, die nicht fehlschlägt.
• `SomeTC` tcl `SomeATC` $atcl$	führt alle Konversionen nacheinander aus, die keinen Fehlschlag ergeben.
• `AllTC` tcl `AllATC` $atcl$	führt alle Konversionen nacheinander aus.

• `SubIfTC` *p tc* –	wendet die Konversion auf alle Teilterme an, für die *p* wahr ist.
• `SubTC` *tc* `SubATC` *atc*	wendet die Konversion auf alle Teilterme an.
• `NthSubTC` *n tc* `NthSubATC` *n atc*	wendet die Konversion auf den *n*-ten Teilterm an.
• `AddrTC` *a tc* `AddrATC` *a atc*	wendet die Konversion auf den mit *a* adressierten Teilterm an.
• – `SweepDnATC` *atc*	wendet die Konversion von außen nach innen auf alle Teilterme an.
• – `SweepUpATC` *atc*	wendet die Konversion von innen nach außen auf alle Teilterme an.
• – `HigherATC` *atc*	wendet die Konversion auf alle äußersten Teilterme an, d.h. falls ein Teilterm konvertiert werden konnte, wird nicht mehr in seinem Inneren angewendet.
• – `LowerATC` *atc*	wendet die Konversion auf alle innersten Teilterme an, d.h. falls ein Teilterm konvertiert werden konnte, wird nicht mehr in der Umgebung des Teilterms angewendet.

Anhang E

Instanziierung, Termersetzung und Angleichung

E.1 Taktik zur Instanziierung eines Lemmas

Die Taktik

`apm_InstHyp` *instances i*

bildet die Grundlage für die Beweiskonstruktionen, die auch die folgenden Taktiken durchführen. Es wird angenommen, daß Hypothese i eine einfache universelle Formel der Gestalt

$$\forall x_1 : T_1 . \forall x_2 : T_2[x_1] . \cdots \forall x_n : T_n[x_1, ..., x_{n-1}] .\ A_1[x_1, ..., x_n] \Rightarrow \cdots \Rightarrow A_m[x_1, ..., x_n] \Rightarrow P[x_1, ..., x_n]$$

ist. In die Variablen x_1 bis x_n werden nun die in der Termliste $instances = [t_1; ...; t_n]$ angegebenen Instanzen eingesetzt. Die Taktik erzeugt für jede Annahme A_i eine Untersequenz, die mit 'antecedent' markiert ist und die Form $... \vdash A_i[t_1, ..., t_n / x_1, ..., x_n]_1$ hat. Desweiteren wird natürlich eine Hauptsequenz erzeugt, die die Gestalt $..., P[t_1, ..., t_n / x_1, ..., x_n]_1 \vdash C$ hat; dabei ist C die ursprüngliche Konklusion.

Anders als die in NuPRL bereits vorhandene Taktik zur Instanziierung arbeitet `apm_InstHyp` nur mit Substitutionen erster Ordnung. Dies heißt, daß man auch `so_lambda`-Terme instanziieren kann, ohne daß diese gesondert behandelt werden. NuPRL dagegen würde solche Instanzen, sofern sie gerade in einen `so_apply`-Term eingesetzt werden, sofort reduzieren.

Die Taktik

`apm_BackThruHyp` *hints eq i*

bestimmt automatisch die Instanzen für die einfache universelle Formel i, indem die Formel C in der Konklusion gegen $P[x_1, ..., x_n]$ gematcht wird, d.h. es wird versucht, die syntaktische Gleichung

$$P[x_1, ..., x_n] \doteq C$$

aufzulösen. Mehr dazu siehe unten. Diese Taktik erzeugt nun keine Hauptsequenz, da die mittels `apm_InstHyp` zwischenzeitlich erzeugte die triviale Form $..., C \vdash C$ hat.

In der Liste *hints* können Instanzen manuell angegeben werden, die automatisch nicht bestimmbar sind. Beispielsweise kann man in einem Transitivitätslemma der Form $\forall x_1 . \forall x_2 . \forall x_3 .\ x_1 \leq x_2 \Rightarrow x_2 \leq x_3 \Rightarrow x_1 \leq x_3$ die Instanz für x_2 nicht durch Matching bestimmen, da sie im P-Teil der Formel gar nicht vorkommt. Durch den Parameter *hints* = ['x_2', s] kann man einen Term s beliebig angeben, der nun für x_2 eingesetzt wird.

(Wird *hints* zur Typbestimmung herangezogen?)

Das Argument *eq* dient dazu, während des Matchings bestimmte Terme als gleich anzusehen, die syntaktisch unterschiedlich ausgedrückt werden. Oder anders gesagt: Mit *eq* wird eine Definition der verwendeten Term-Gleichheit $\doteq$ angegeben. Mehr dazu siehe unten.

Die Taktik

`apm_BackThruLemma` *hints lname eq*

benutzt statt einer Hypothese die einfache universelle Formel, die im Theorem mit Namen *lname* in der Bibliothek abgelegt ist.

E.2 Konversion zur Instanziierung eines Lemmas

Die obigen Taktiken bilden die Grundlage für folgende Konversionen. Hier geht es nun darum, einen Teilterm s, der in einer Formel vorkommmt, durch einen Term s' zu ersetzen, wobei die Ersetzbarkeit durch eine spezielle einfache universelle Formel abgesichert wird. Diese Formel kann wiederum als Hypothese oder als Lemma referenziert werden.

Die Formel muß nun folgende Gestalt haben:

$$\begin{array}{l}\forall x_1{:}\,T_1.\forall x_2{:}\,T_2[x_1].\cdots\forall x_n{:}\,T_n[x_1,...,x_{n-1}].\\ \quad A_1[x_1,...,x_n] \Rightarrow \cdots \Rightarrow A_m[x_1,...,x_n] \Rightarrow\\ \qquad u[x_1,...,x_n] \trianglelefteq v[x_1,...,x_n]\end{array}$$

Dabei ist $\trianglelefteq$ die Relation, die die Ersetzung rechtfertigt. Die Formel kann nun *von links nach rechts* interpretiert werden, dann ist u durch v zu ersetzen, oder *von rechts nach links*, dann umgekehrt v durch u.

Instanziierung heißt hier, daß bei Links-nach-rechts-Ersetzung der u-Term gegen s gematcht wird, und die Instanzen benutzt werden, um durch deren Substitution in v den Term s' zu erhalten. Bei Rechts-nach-links-Ersetzung sind die Rollen von u und v vertauscht.

Die Konversionen, die dies bewerkstelligen, lauten nun:

`apm_HypATC` *hints eq direction i*
`apm_LemmaATC` *lname hints eq direction*

Das Argument *direction* gibt dabei an, ob von links nach rechts oder von rechts nach links substituiert werden soll. Im ersteren Fall ist 'LR', im zweiteren 'RL' anzugeben. Die anderen Argumente haben die gleiche Bedeutung wie schon bei der Instanziierungstaktik.

E.3 Wie die Instanziierung gefunden wird

In der einfachen universellen Formel muß jede der Variablen x_i durch einen Typ T_i deklariert werden. Desweiteren ist bekannt, daß P eine Proposition sein muß, oder im Falle der Termersetzung, daß u und v der Typisierung der Relation $\trianglelefteq$ entsprechen müssen. Diese Typinformationen werden nun genutzt, um die Instanzen zu finden.

Zunächst wird versucht, per Matching die Instanzen zu bestimmen (s.u.). Nun können einige Variablen uninstanziiert bleiben. Entweder wird nun das Argument *hints* benutzt, oder, falls die Instanz selbst ein Typ muß, wird folgendes Verfahren angewendet.

Es werden Typen S_i für die Instanzen bestimmt, und zwar im Kontext der Formel. Diese Typen werden mit den Deklarationen T_i verglichen, um den unbestimmten Instanzen auf die Spur zu kommen. Dazu werden die Gleichungen $S_i \stackrel{*}{=} T_i$ aufgestellt und als zusätzliche Bedingungen in die Typprüfung hineingenommen. Dadurch sollte das TGS alle Informationen enthalten, die zur Bestimmung der Unbekannten notwendig sind. Gelingt dies nicht, ist davon auszugehen, daß die Instanz keine Rolle spielt und beliebig sein kann; man kann dann z.B. $\mathbf{Z}$ einsetzen.

Im Augenblick verwendet der Typprüfungsalgorithmus leider keine expliziten TGS. Daher wird das Verfahren durchgeführt, indem die zusätzlichen Gleichungen per Unifikation aufgelöst werden und die so erhaltenen Substitutionen zu der schon bestehenden addiert werden.

Beispiel für ein Lemma, daß dieses Instanziierungsverfahren benötigt

Folgendes Theorem soll instanziiert werden:

$$\forall \alpha\colon \mathtt{TYPES}.\ \forall S\colon \mathtt{set}(\alpha).\ \forall a\colon \alpha^T.\ \forall P^1\colon \alpha^T \xrightarrow{2} \mathbf{P}\quad (\exists x \in (S+a).P^1[x]) \iff (\exists x \in S.P^1[x]) \vee P^1[a]$$

Wie man sieht, tauchen zwar die Variablen S, a und P^1 in der eigentlichen Formel auf, jedoch nicht die Typvariable α. Dennoch ist für jede Anwendung des Lemmas diese Variable eindeutig bestimmt: Für sie muß derjenige Datentyp eingesetzt werden, über dem die endlichen Menge S gebildet wird. Um an α zu kommen, reicht es also aus, den Typ der Instanz für S aufzuspüren, der die Form $\mathtt{set}(\alpha)$ haben muß, und α aus diesem Typ abzulesen. Obiges Verfahren nutzt jedoch alle Informationen, die es kriegen kann; auch die Typen der anderen Instanzen werden bestimmt und werden in Bedingungen überführt. Es sollte jedoch dasselbe Ergebnis herauskommen, da sich diese Bedingungen nicht widersprechen dürfen; ansonsten ist von einem Typisierungsfehler auszugehen.

E.4 Matching

Ein Matching-Problem besteht darin, daß ein Term A, in dem Variablen $x_1, ..., x_n$ frei vorkommen (die *Matchingvariablen*), einem Term B gleichgesetzt wird:

$$A[x_1, ..., x_n] \doteq B$$

Kommt eine Variable x_i auch in B vor, so soll dieses Vorkommen unterschieden werden. Die Lösung des Matching-Problems besteht darin, Instanzen für $x_1, ..., x_n$ zu finden, etwa $t_1, ..., t_n$, die die beiden Terme syntaktisch gleichmachen:

$$A[t_1, ..., t_n/x_1, ..., x_n]_1 \doteq B$$

Das Matching-Problem kann leicht auf den Fall ausgedehnt werden, daß die Variablen x_i zweiter Ordnung sind. In diesem Fall werden nicht Terme t_i instanziiert, sondern Bindungsbereiche $B_i = y_{i,1}, ..., y_{i,m}.t_i$, die allerdings von der Stelligkeit m her passen müssen.

Schwieriger ist es zu behandeln, daß häufig für ein und dasselbe Konstrukt syntaktisch leicht verschiedene Notationen existieren. Beispiele hierfür sind u.a.

- Typangaben α^T bzw. T, wenn α durch einen Datentypoperator und T durch einen Trägeroperator desselben Typs angegeben werden.
- Boole'sche Operatoren, die einmal in einer Fassung für den Typ $\mathbf{B}$ und einmal in einer Fassung für den Typ $\mathbf{P}$ existieren. Beide Formen können leicht ineinander überführt werden.

Es wurde eine Matching-Funktion implementiert, die grundsätzlich mit solchen feinen Differenzen umgehen kann. Dazu dienen sogenannte *Equalizer*, spezielle Konversionen, die den Zweck haben, die Notation eines Terms an die eines anderen Terms anzugleichen, so daß beide Terme syntaktisch identisch sind. Equalizer wurden in obigen Taktiken jeweils als *eq*-Argument übergeben.

Beim Matching mit Angleichen entsteht als Ergebnis nun nicht mehr nur eine Liste von Instanzen, sondern zusätzlich eine Konversion δ, die, wenn man sie auf den B-Term anwendet, diesen identisch zum instanziierten A-Term macht:

$$A[t_1, ..., t_n/x_1, ..., x_n]_1 \doteq B\delta$$

E.5 Terme angleichen

Als ML-Typ für Equalizer wird ungefähr folgendes eingeführt:

$\texttt{equalizer} = \texttt{term} \rightarrow \texttt{addr\_typed\_convn}$

Ein angewendeter Equalizer *eq s* soll bedeuten, daß eine Konversion berechnet wird, die einen Term an s angleicht. Dies heißt zunächst nur, daß die äußersten Operatoren beider Terme identisch gemacht werden; im Inneren dürfen sie sich noch unterscheiden. Dies ist die Minimalbedingung, die an einen Equalizer gestellt wird. Ein vollständig angleichender Equalizer erzeugt dagegen einen Term, der im Ganzen identisch bis auf α-Umbennungen ist.

Die Anwendung eines Equalizers, die nur mit der Funktion `EQrunATC` (s.u.) erfolgen kann, ist kontrolliert. Es wird nachgeprüft, ob die beiden Operatoren identisch gemacht wurden; ansonsten schlägt die Angleichung fehl.

Es gibt nun folgende Funktionen, die mit Equalizern hantieren:

`MakeEQ` f	erzeugt aus der Funktion f des Typs `term` $\rightarrow$ `addr_typed_convn` einen äquivalenten Equalizer
`EQrunATC` *eq s*	wendet den Equalizer *eq* auf *s* an; Ergebnis ist eine Konversion
`FailEQ`	Equalizer, der stets fehlschlägt
`IdEQ`	der identische Equalizer; er funktioniert nur, wenn die äußersten Operatoren bereits identisch sind
eq_1 `ANDTHENEQ` eq_2	gleicht zunächst mittels eq_1 an, um auf das Resultat sofort eq_2 anzuwenden. Nach der ersten Anwendung muß bereits der äußerste Operator übereinstimmen; die zweite Anwendung kann weitere Angleichungen im Inneren durchführen.
eq_1 `ORELSEEQ` eq_2	es wird zunächst versucht, mittels eq_1 anzugleichen. Schlägt dies fehl, wird stattdessen eq_2 benutzt.
`OnDemandEQ` *eq*	führt *eq* nur aus, falls die äußersten Operatoren noch unterscheiden.

`SubEQ` *eq*	Dies kann nur angewendet werden, falls die äußersten Operatoren bereits übereinstimmen. Um die sich entsprechenden Teilterme anzugleichen, wird *eq* benutzt.
`AlphaSubEQ` *aeq*	Dies kann nur angewendet werden, falls die äußersten Operatoren bereits übereinstimmen. Wie bei `SubEQ` sollen auch nun die Teilterme angeglichen werden. Es handelt sich bei *aeq* jedoch um eine Funktion des Typs `var`×`var` list → `equalizer`, die also eine Liste von Variablenpaaren als Argument hat. Diese Liste enthält die sich entsprechenden Bindungsvariablen der jeweils anzugleichenden Teilterme. Der Equalizer kann damit die unterschiedliche Benennung der Bindungsvariablen verfolgen.
`SweepDnEQ` *eq*	Wendet den Equalizer *eq* zunächst auf die äußersten Operatoren an, sofern sich diese unterscheiden. Es wird dann rekursiv in die sich entsprechenden Teilterme von außen nach innen hinabgestiegen, um diese ebenfalls anzugleichen. Ist dies erfolgreich, haben beide Terme an sich entsprechenden Stellen identische Operatoren, können sich aber noch in den Variablen und Parametern unterscheiden.
`AlphaSweepDnEQ` *eq*	Im Gegensatz zu `SweepDnEQ` wird nun auch überprüft, ob die Bindungsvariablen modulo Umbennung übereinstimmen. Diese Funktion ist nur erfolgreich, wenn die Terme α-kompatibel sind.

Mittels dieser Funktionen kann ein Equalizer, der nur die Angleichung auf äußerster Ebene durchführen kann, leicht auf eine vollständige Angleichung fortgesetzt werden.

Equalizer für Typterme

Es wurde ein Equalizer `typeEQ` implementiert, der Typausdrücke, die auf unterschiedliche Weise Datentyp- und Trägeroperationen anwenden, auf Grundlage der Datentyptabelle angleichen kann. `typeEQ` bezieht sich natürlich nur auf die äußersten Typoperatoren; durch `AlphaSweepDnEQ typeEQ` kann man aber einen Equalizer bauen, der Typausdrücke komplett angleicht.

Equalizer für boole'sche Operatoren

Die Implementierung dieser Equalizer ist noch nicht abgeschlossen. Es hat sich als problematisch erwiesen, daß Equalizer keinen Zugriff auf Typinformation haben. In einer Neuauflage des Konzepts sollte dies ergänzt werden. Einige Lemmata, die für diese Angleichungen benutzt werden müssen, benötigen nämlich, um überhaupt instanziierbar zu sein, diese Informationen.

Anhang F

Effiziente Substitution gleicher Terme

Die Typentheorie besitzt mit der `subst`-Regel eine einfache und schnelle Möglichkeit, einen Term an beliebiger Position in der Konklusion durch einen gleichen anderen Term zu ersetzen. Würde man diesen Effekt durch das Schema der Konversionen simulieren, müßte man zunächst den zu ersetzenden Teilterm freilegen, ihn substituieren und anschließend den umliegenden Term wieder zusammensetzen. Tatsächlich ist man jedoch auf diese Simulation angewiesen, wenn man solche Gleichheitsersetzungen mit anderen Substitutionen mischen möchte. Im folgenden präsentieren wir eine Möglichkeit, wie man mit Hilfe der `subst`-Regel direkt eine adressierende getype Konversion programmieren kann, die *nicht* den umständlichen Weg der Analyse geht.

Grundlage einer solchen Konversion ist eine Taktik, die die `subst`-Regel auf Substitutionen zweiter Ordnung erweitert. Diese Taktik hat folgendes Verhalten:

$\Gamma \;\vdash\; C[t[x_1, ..., x_n]@a]$

BY **so_substitute_concl_at_addr** $a\;\; [x_1; ...; x_n]$ ’`so_lambda`$(y_1, ..., y_n.t'[y_1, ..., y_n])$’
$z_1\colon T_1 \to \cdots \to z_n\colon T_n \to T \quad U_j$

– – antecedent – –
$\Gamma \;\vdash\;$ `so_lambda`$(x_1, ..., x_n.t[x_1, ..., x_n]) =$
`so_lambda`$(y_1, ..., y_n.t'[y_1, ..., y_n]) \in z_1\colon T_1 \to \cdots \to z_n\colon T_n \to T$

– – wf – –
$\Gamma,\; z\colon T \;\vdash\; C[z@a] \in U_j$

– – main – –
$\Gamma \;\vdash\; C[t'[x_1, ..., x_n/y_1, ..., y_n]@a]$

Dabei ist a die Adresse, bei der in C der zu ersetzende Term t steht. In diesem Term t kommen die Variablen x_1,...,x_n frei vor. An seine Stelle wird nun der Term t' eingesetzt, der in Form eines Bindungsbereich mit den Stellen $y_1, ..., y_n$ formuliert ist. Beim Einsetzen sind die x_i für die formalen Stellen y_i zu substituieren. Wie man sieht, muß noch der Typ des Bindungsbereichs angegeben werden; dabei handelt es sich um einen abhängigen Funktionenraum. Die ’antecedent’-Untersequenz drückt aus, daß der ersetzte Bindungsbereich den Term t wegen Gleichheit auch gerechtfertigt werden kann. Im ’wf’-Ziel erhalten wir eine Sequenz, die aus der originalen `subst`-Regel stammt und ausdrückt, daß in C bei Adresse a allgemein substituiert werden darf.

Diese neue taktische Regel kann man nun durch eine trickreiche Simulation auf primitive Regeln, insbesondere `subst`, abbilden. Die Details lassen wir an dieser Stelle aus.

Die ATC-Konversionen haben nun die Fähigkeit, auch selbst die Adressierung innerhalb der Formel durchzuführen. Damit kann man Alternativen zum Standardverfahren programmieren, das, wie geschildert, die Formel sukzessive analysiert und wieder zusammensetzt. Wird diese Option benutzt, erhält die elementare ATC-Konversion beim Aufruf zusätzlich die Adresse a, an der die Ersetzung stattfinden soll. Die Konversion muß nun selbst die Adressierung des Formelteils durchführen; obige Taktik kann dies ohnehin, weshalb diese Option hier vorteilhaft ist.

Achtung: Für die folgende Konversion existiert noch keine Implementierung. Ob das Schema „durchgeht“, ist deshalb nicht abschließend zu beurteilen.

Folgende Konversion leistet das gewünschte:

$$\texttt{substATC}\ a\ \ [x_1;...;x_n]\ \ \text{'}\texttt{so\_lambda}(y_1,...,y_n.t'[y_1,...,y_n])\text{'}$$
$$z_1{:}T_1 \to \cdots \to z_n{:}T_n \to T \qquad U_j$$

Es wird wiederum erwartet, daß bei Adresse a des vorgesetzten Terms die zu ersetzende Teilformel t steht, in der wieder die Variablen $x_1,...,x_n$ frei vorkommen. Die übrigen Argumente haben ebenfalls dieselbe Bedeutung wie zuvor.

Die Wirkung dieser Konversion ist nun, daß t durch t' ersetzt wird, wobei der angegeben Bindungsbereich beachtet wird. Man erhält aber ein 'antecedent'-Ziel, das wie oben aussieht und noch bewiesen werden muß.

Anwendung: Optimierung von Gleichheitstheoremen

Die Konversionen `apm_HypATC` und `apm_LemmaATC` sind in der Lage, Ersetzungen aufgrund universeller Formeln durchzuführen, die in einer Hypothese oder einem Theorem bereitstehen. Falls die Relation die Gleichheit ist, kann man nun aber statt des allgemeinen Schemas auf `substATC` zurückgreifen und wesentliche Effizienzgewinne erzielen. Dies heißt nämlich, daß bei der Gleichheitsersetzung praktisch kein Adressierungsaufwand anfällt!

Folgendes ist noch nicht implementiert: Die Konversionen `apm_HypATC` *bzw.* `apm_LemmaATC` *sollten so verbessert werden, daß sie automatisch erkennen, wenn modulo der Gleichheit ersetzt wird und dann das optimierte Schema auswählen.*

Restrukturierung des Termersetzungspakets

Die bei ATC-Konversionen mögliche Form der Selbstadressierung ist eine verallgemeinerte Form der im Standard-Termersetzungspaket vorgesehenen „direct computation conversions“. Diese werden nämlich speziell gespeichert, und zwar als Berechnungsfolgen; damit entfällt nämlich der Adressierungsaufwand, weil die in der Typentheorie enthaltenen primitiven Regeln zur Berechnung ebenfalls selbst adressieren können.

Falls einmal das Termersetzungspaket reimplementiert werden sollte, kann man die Spezialbehandlung der „direct computation conversions“ zugunsten selbstadressierender Konversionen aufgeben.

Anhang G

Konversionen zur Reduktion

Im Standard-Termersetzungspaket sind bereits zahlreiche Konversionen vorhanden, die das Rechnen an beliebiger Stelle in einer Formel ausdrücken können. Freilich bilden diese Konversionen nur einen Grundstock des Rechnen; für ernsthafte Anwendungen muß noch stark erweitert werden.

Folgende zwei Kritikpunkte sollten dabei maßgeblich sein:

1. Es sollte das Rechnen auf abstraktem Niveau ermöglicht werden. Im Augenblick ist es nur möglich, primitive Redizes auszurechnen.

2. Es sollte neben dem gewöhnlichen Vorwärtsrechnen auch das Rückwärtsrechnen unterstützt werden. Dies ist insbesondere für Schlüsse wichtig.

Im Augenblick liegen noch keine konkreten Entwürfe vor, wie diese Punkte ausgearbeitet werden könnten. Folgendes sollte aber bedacht werden.

Erstens sollte es möglich sein, analog der Anwendung eines Theorems, einen Rechenschritt als einen Schluß aufzufassen. Es ergeben sich dabei genau die gleichen Probleme wie bei den Theoremen, vor allem dasjenige der Instanziierung (beim Rückwärtsrechnen).

Das manuelle Programmieren von abstrakten Reduktionsschritten ist, wie einige Experimente ergaben, sehr aufwendig und fehleranfällig. Es sollte daher die Möglichkeit ins Auge gefaßt werden, diese Rechenschritte generieren zu lassen. Meistens setzen sich die Schritte nur aus wenigen primitiven Reduktionen zusammen, und durch eine stark begrenzte Suche könnte man dies auch automatisch herausfinden.

Bei der abstrakten Reduktion ist desweiteren die Teilreduktion von der vollständigen Reduktion zu unterscheiden. Letztere ist nur möglich, wenn alle Komponenten in concreto bekannt sind, und führt zu einem Endergebnis. Erstere wertet nur soweit aus, wie dies nach dem Stand der Dinge gerade möglich (oder gewünscht) wird. Dies ist vor allem innerhalb von Beweisen nützlich.

Anhang H

Details der Logiktaktik

H.1 Beweisdarstellung

Kernstück der Darstellung ist diejenige der Sequenz. Es wird folgender ML-Datentyp benutzt:

$$\begin{array}{lll} \texttt{search\_sequence} = & (\texttt{var} \times \texttt{term} \times \texttt{bool})\ \text{list} & (*\ \textit{Deklarationen}\ *) \\ & \times\ \texttt{term}\ \text{list} & (*\ \textit{Hypothesen}\ *) \\ & \times\ \texttt{term} & (*\ \textit{Konklusion}\ *) \\ & \times\ \texttt{bool} & (*\ \textit{Status: ist die Seqeunz validiert?}\ *) \\ & \times\ (\texttt{var} \times \texttt{term})\ \text{list} & (*\ \textit{Substitution}\ *) \end{array}$$

Eine Deklaration ordnet jeder Variable, die deklariert werden soll, einen als Term dargestellten Typ zu. Desweiteren muß boole'sch angegeben werden, ob die Deklaration angenommen (false) oder gesucht (true) wird. Die Hypothesen werden der Reihe nach in ihrer Termdarstellung gespeichert, ebenso die Konklusion. Eine Sequenz kann validiert sein oder nicht. Ersteres heißt, daß in einem Beweis aus den Substitutionen der Untersequenzen eine Substitution für die gegebene Sequenz erfolgreich berechnet wurde. Bei nicht-validierten Sequenzen ist dies noch nicht geschehen oder nicht möglich.

Ein Beweis ist nun ein geordneter Baum, in dessen Knoten je eine Sequenz und maximal eine Regelanwendung enthalten ist. Dies wird so repräsentiert:

$$\texttt{search\_proof} = (\texttt{search\_sequence} \times (\texttt{tok} \times \texttt{int} \times \texttt{int})\ \text{list})\ \texttt{ordered\_tree}$$

Dabei kann durch $(\texttt{tok} \times \texttt{int} \times \texttt{int})$ list entweder eine oder keine Regelanwendung angegeben werden. Knoten ohne Regelanwendung dürfen keine Söhne haben. Knoten mit Regelanwendung müssen genau so viele Söhne haben, wie die Anwendung ergibt.

Zur Konstruktion eines Beweises dient die Funktion

`refine_search_proof` *rule sp*

Mit *sp* muß ein Beweis übergeben werden, dessen Wurzelsequenz hierdurch verfeinert wird. Es wird die Regel *rule* angewendet; es werden genau so viele Sequenzen als Söhne der Wurzel erzeugt, wie die Regel vorschreibt. Die Wurzelsequenz und ihre Söhne werden als nicht-validiert markiert.

Mittels dieser Funktion kann man nun einen Beweis konstruieren, der aber, um vollständig zu sein, nicht nur alle Beweisäste abschließen muß, sondern auch eine Validierung bestehen muß. Hierzu dient die Funktion

`validate_search_proof` *sp*

Es wird versucht, einen zu *sp* äquivalenten Beweis zu konstruieren, der jedoch validiert ist. Diesen neuen Beweis erhält man als Funktionsergebnis.

H.2 Suchtaktiken

Mit den bisher vorgestellten Funktionen ist noch kein Staat zu machen; sie bilden nur die Basisimplementierung für komplexere Taktiken. Das NuPRL-Taktik-Konzept wird daher auf den Datentyp `search_proof` übertragen.

Kernstück ist hier wieder ein Datentyp, um den alles herum organisiert ist:

`search_tactic` = `search_proof` → `search_proof`

Man kann Taktiken mit zweierlei Absichten versehen:

- Zur Konstruktion eines Beweises erhält die Taktik T als Argument einen Beweis, von dem aber nur die Wurzelsequenz als Beweisziel aufgefaßt wird. Ergebnis ist ein umfangreicherer Beweis.
- Zur Validierung eines Beweises erhält die Taktik T als Argument einen Beweis, der keine offenen Beweisziele mehr enthält und erzeugt einen validierten Beweis.

Die atomare Konstruktionstaktik ist

`SearchRefine` *rule*

die wie `refine_search_proof` funktioniert. Die grundlegende Validierungstaktik ist

`SearchValidate` T

die zunächst die Taktik T anwendet und anschließend das Ergebnis validiert.

Desweiteren gibt es folgende höheren Taktiken und Taktikale:

`SearchHypothesis` i	wende mod_hypothesis i an und validiere sofort
`SearchContradiction` i j	wende mod_contradiction i j an und validiere sofort
`SearchId`	die identische Taktik läßt den Beweis wie er ist
`SearchFail`	diese Taktik schlägt immer fehl
T_1 `SEARCH_ORELSE` T_2	probiert zunächst T_1 und nur, wenn dies fehlschlägt, auch T_2
T_1 `SEARCH_THEN` T_2	führt T_1 aus, und auf alle offenen Teilziele des Ergebnisses T_2.
`SearchTry` T	= T `SEARCH_ORESLE SearchId`
`SearchComplete` T	führt T aus. Falls sich dadurch nicht ein Beweisbaum ergibt, in dem alle Äste abgeschlossen ist, schlägt die Taktik fehl.
`SearchRawRepeat` T	führt T wiederholt aus, bis T fehlschlägt
`SearchProgress` T	führt T aus. Schlägt jedoch fehl, wenn sich keine Veränderung des Beweises ergibt.
`SearchRepeat` T	= `SearchRawRepeat (SearchProgress` T`)`
`SearchOnFirstHyp` f	führt für die kleinste Hypothese i, für die folgendes nicht fehlschlägt, die Taktik f i t aus, wobei t der Term der i-ten Hypothese ist

Mittels dieser Taktiken ist es möglich, auf einem etwas höheren Niveau den Algorithmus zur Beweissuche zusammenzusetzen. Man beachte, daß diese Taktiken, wenn nicht anders angegeben, keine Validierung durchführen. Auf diese Weise kann man die Validierung zu günstigen Zeitpunkten zusammenfassen.

H.3 Bemerkungen zum Suchalgorithmus

Der Suchalgorithmus sieht nun relativ simpel aus:

```
let search_the_proof =
  SearchValidate ( SearchRepeat ( search_final_rules
                                  SEARCH_ORELSE
                                  search_linear_rules
                                )
                   SEARCH_THEN
                   search_multi_or_i_rules
                   SEARCH_THEN
                   ( SearchValidate ( SearchRepeat ( search_final_rules
                                                     SEARCH_ORELSE
                                                     search_linear_rules
                                                   )
                                    )
                   )
                 )
;;
```

Dabei ist noch zu ergänzen, daß `search_final_rules` alle möglichen finalen Regeln ausprobiert, bis eine paßt, daß `search_linear_rules` alle linearen und hyperlinearen Regeln ausprobiert, und außerdem daß `search_multi_or_i_rules` alle Alternativen einer ggf. möglichen mod_multi_or_i-Anwendung prüft.

In der aktuellen Implementierung ist zudem noch eine Optimierung eingebaut. Es müssen nicht immer alle finalen Regeln ausprobiert werden, sondern nur diejenigen, die sich aufgrund der geänderten Sequenz ergeben.

H.4 Übersetzung von Suchbeweisen in NuPRL-Beweise

Nur validierte Beweise können in NuPRL-Beweise übersetzt werden. Die Übertragung erfolgt im wesentlichen 1:1, die meisten Regeln des modifizierten Sequenzenkalküls können in den originalen Sequenzenkalkül ohne Änderung übernommen werden. Lediglich bei all_e und exists_i ist zusätzlich der Term anzugeben, der eingesetzt werden soll. Dieser kann in der Regel der Substitution entnommen werden; kommt er dort nicht vor, spielt er für den Beweis keine Rolle. Man kann dann einen beliebigen Term passenden Typs angeben.

Leider ergibt sich bei der Übersetzung noch ein Umstand, der recht knifflig ist. Die Hypothesen müssen nämlich neu numeriert werden. Während der Übersetzung muß daher die Abbildung der modifizierten Hypothesen auf die NuPRL-Hypothesen mitgeführt werden.

Die Implementierung der Übersetzung kann einfach als NuPRL-Taktik aufgerufen werden:

`transform_search_proof` *sp*

Dabei ist *sp* der validierte Suchbeweis.

Die NuPRL-Taktik `search_and_transform` führt zunächst den Suchalgorithmus durch und übersetzt anschließend sofort den Beweis in einen NuPRL-Beweis.

Anhang I

Übertragung des Costas-Arrays-Algorithmus' nach ML

I.1 Einfacher Costas-Arrays-Algorithmus

Im folgenden präsentieren wir eine Übersetzung des generierten Costas-Arrays-Algorithmus in den CAML-Light-Dialekt der Sprache ML. Es wurde versucht, sich relativ eng an die Strukturierung der Theorie zu halten; mehr oder weniger wurde dies dadurch erreicht, daß Konzepte übertragen wurden, indem diejenigen Anteile, die zur Berechnung beitragen, beibehalten und diejenigen Anteile, die nur theoretische Bedeutung haben, entfernt wurden.

```
(*************************
 * COSTAS ARRAYS         *
 *                       *
 * Einfacher Algorithmus *
 *************************)


(********************************)
(* Grundfunktionen Listen/Mengen *)
(********************************)


let rec filter f L =
  match L
   with []   -> []
      | h::t -> if f h then h::(filter f t)
                       else filter f t
;;


let rec nodups L =
  match L
   with []   -> true
      | h::t -> not (mem h t) & nodups t
;;
```

```
let rec upto a b =
  if b < a then
    []
  else
    a::(upto (a+1) b)
;;


let permutation L1 L2 =
  for_all (fun a -> mem a L2) L1
  & for_all (fun a -> mem a L1) L2
  & nodups L1
;;


(*******************)
(* BASISALGORITHMUS *)
(*******************)


type ('D,'R,'S) GS =
 { s0    : 'D -> 'S
 ; split : 'D -> 'S -> 'S list
 ; ext   : 'S -> 'R list
 ; Phi   : 'D -> 'S -> bool
 }
;;


let filtered_splitting gs =
  let new_split x s =
  ( let subspace = gs.split x s in
    filter (gs.Phi x) subspace
  )
  in
    { s0    = gs.s0
    ; split = new_split
    ; ext   = gs.ext
    ; Phi   = fun _ _ -> true
    }
;;


let add_filter gs addPhi =
  { s0    = gs.s0
  ; split = gs.split
  ; ext   = gs.ext
  ; Phi   = (fun x s -> gs.Phi x s & addPhi x s)
  }
;;
```

```
let print_descriptor s =
  print_string "[";
  do_list (fun x -> print_string (string_of_int x ^ " "))
          s;
  print_string "]\n"
;;



let filterless_body (gs:(int,int list,int list) GS) Obool =
  let search x =
  ( let rec h s =
    ( (* print_descriptor s; *)
      let extraction = filter (Obool x) (gs.ext s) in
      let subspace = gs.split x s in
      let rest = flat_map h subspace in
        extraction @ rest
    )
    in h (gs.s0 x)
  )
  in search
;;



let filtered_body gs Obool =
  filterless_body (filtered_splitting gs) Obool
;;



(******************)
(* SPEZIALISIERUNG *)
(******************)



let gs_specialized gs theta =
  { s0    = (fun x   -> gs.s0 (theta x))
  ; split = (fun x s -> gs.split (theta x) s)
  ; ext   = gs.ext
  ; Phi   = (fun x s -> gs.Phi (theta x) s)
  }
;;



(********************)
(* STANDARD-SUCHRAUM *)
(********************)
```

```
let gs_seq_set_3 =
  { s0    = (fun x   -> [])
  ; split = (fun x s -> map (fun a -> s @ [a]) x)
  ; ext   = (fun x   -> [x])
  ; Phi   = (fun x s -> nodups s)
  }
;;


(************************)
(* COSTAS-ARRYS-Beispiel *)
(************************)


let rec del_first p j =
  if j = 0 then
    p
  else
    del_first (tl p) (j-1)
;;


let dtrow p j =
  let q = del_first p j in

  let rec diff p1 q1 =
  ( if q1=[] then
      []
    else
      (hd p1 - hd q1)::(diff (tl p1) (tl q1))
  )

  in diff p q
;;


let CA_0bool n p =
    permutation p (upto 1 n)
  & for_all (fun j -> nodups (dtrow p j))
            (upto 1 n)
;;


let CA_theta x =
  upto 1 x
;;
```

```
let Costas =
  filtered_body (gs_specialized gs_seq_set_3 CA_theta) CA_Obool
;;
```

I.2 Costas-Arrays mit Optimierungen

Hier nun modifizierte Algorithmen, die zwei Optimierungen einführen. Zum einen wird eine zusätzliche Filterung durchgeführt; zum anderen wird die Berechnung mittels „endlicher Differenzierung“ optimiert.

```
(*********************************************)
(* Costas-Arrays mit zusaetzlicher Filterung *)
(*********************************************)



let CA_opt_phi theta_x s =
  for_all (fun k -> nodups (dtrow s k)) (upto 1 (list_length s))
;;



let Costas_opt1 =
  filtered_body (add_filter (gs_specialized gs_seq_set_3 CA_theta)
                            CA_opt_phi
                )
                CA_Obool
;;



(***********************************************)
(* Costas-Arrays mit endlicher Differenzierung *)
(***********************************************)



(* Achtung: Costas-Arrays-Zeilen sowie Deskriptoren
   sind umgekehrt gespeicherte Listen! *)



let add_dt dt s a =
  ( map2 (fun dt_row x -> (x-a)::dt_row)
         dt
         s
  )
  @ [ [] ]
;;
```

```
let dt_nodups dt =
  for_all (fun dt_row -> match dt_row
                         with []   -> true
                            | h::t -> not (mem h t))
          dt
;;



let map_except f L =
  let rec R L1 L2 =
  ( match L2
     with []   -> []
        | h::t -> (f h (L1 @ t))::(R (h::L1) t)
  )
  in R [] L
;;



let Costas_opt2 n =
  let theta_n = upto 1 n in
  ( let rec h (s,s_compl,dt_s) =
    ( let extraction = if list_length s = n then
                         [rev s]
                       else []
                       in
      let subspace0 = map_except (fun a ac -> (a::s, ac, add_dt dt_s s a))
                                 s_compl in
      let subspace = filter (fun (s1,s_compl1,dt_s1) -> dt_nodups dt_s1)
                            subspace0 in
      let rest = flat_map h subspace in
        extraction @ rest
    )
    in
        h ( [], theta_n, [] )
  )
;;
```