Universität Bielefeld

Diplomarbeit

In der Arbeitsgruppe Praktische Informatik,
an der Technischen Fakultät.
Betreut von Prof. Dr. Robert Giegerich

QUETZAL

Ein Beschreibungssystem für semantische Unifikationsverfahren

Stephan Thesing

Jakob-Kaiser-Str. 14b/App 311

33615 Bielefeld

Inhaltsverzeichnis

Danksagung

Es gibt viele Menschen, die am Zustandekommen dieser Diplomarbeit beteiligt waren: Zuerst sicher Herr Prof. Dr. Robert Giegerich, der mein Interesse an dem Themengebiet der logisch-funktionalen Programmierung geweckt hat und bereit war, diese Arbeit zu betreuen.
Dank schulde ich auch Herrn Dr. Enno Ohlebusch für Anregungen und Diskussionen über mathematische Strenge und Korrektheit in wissenschaftlicher Arbeit, ebenso wie allen anderen Mitgliedern der AG PI.
Besonders interessant, lehrreich und befriedigend war die Zusammenarbeit mit Dipl.-Inf. Bernd Bütow in dem Projekt "Abstrakte Inferenzmaschine"[1] während zweier spannender Jahre.
Der Prof.-Rhein-Stiftung danke ich für die gewährte Unterstützung.
Meinen Eltern (allen vieren) und Geschwistern gilt mein Dank für ihre Unterstützung über viele Jahre hindurch.
Als letztes, aber vor allem, gilt mein Dank meiner Frau, die bewiesen hat, daß das Sprichwort über 'asiatische Geduld' mehr als einen wahren Kern besitzt.

Für meine Großeltern, in memoriam.

[1] Gefördert von der DFG unter AZ Gi 178-1/2

Kapitel 1

Einleitung

Noch war nichts verbunden. Nichts gab Laut, nichts bewegte, nichts erschütterte, nichts brach des Himmels Schweigen. Noch gab es nichts Aufrechtes. Nur die ruhenden Wasser, das sanfte Meer, einsam und still. Nichts anderes.
Unbeweglich und stumm war die Nacht, die Finsternis. Aber im Wasser, umflossen von Licht, waren diese: Tzakól der Schöpfer; Bitól, der Former; der Sieger Tepëu und die Grünfederschlange Gucumátz; Alóm auch und Caholóm, die Erzeuger. Unter grünen und blauen Federn waren sie verborgen, darum sagt man Grünfederschlange.

Popol Vuh, das Buch des Rates der Quiché Maya

1.1 Imperative und deklarative Programmiersprachen

Man unterscheidet heute zwei Kategorien von Programmiersprachen: Einerseits die imperativen Sprachen, wie **Pascal, C**, andererseits die deklarativen Sprachen, wie **Miranda, Prolog, Haskell**.
Bis vor einigen Jahren hatten die imperativen Sprachen die unangefochtene Führungsrolle inne: Deklarative Sprachen führten größtenteils ein Schattendasein als Forschungsobjekte an Universitäten.
Die Semantik der imperativen Sprachen beruht auf dem Konzept eines globalen *Speichers* und der Veränderung seines Inhaltes. Jeder Befehl eines Programmes kann den Inhalt des Speichers verändern, daher ist die *Reihenfolge* dieser Befehle signifikant. Dieses Konzept ist primitiv und auf die realen Architekturen von Computern zugeschnitten. Es wird zwar durch einen mehr oder weniger ausgeprägten Grad an Abstraktion "verzuckert", bleibt aber letzten Endes als fundamentales Konzept übrig[1]
Die Führungsrolle der imperativen Sprache ist eine direkte Folge dieses Konzeptes; denn imperative Programme lassen sich relativ einfach und direkt in Assemblersprachenprogramme für irgendeine Architektur umsetzen. Daher kann ein imperatives Programm in ein kurzes und effizientes Assemblerprogramm umgesetzt werden.
Die primitive Semantik und direkte Korrespondenz zu realen Computerarchitekturen sind aber gleichzeitig der Schwachpunkt der imperativen Sprachen: Größere Programme für kompliziertere Aufgaben sind schwer bau- und wartbar. Die Lösung des gegebenen Problems wird dadurch erschwert, daß man den Lösungsweg in einem unzulänglichen, in-

[1] So ist die Kritik von **C** als "maschinenunabhängiger Assemblersprache" durchaus nicht unbegründet.

adequaten Formalismus ausdrücken muß, oder anders gesagt: Die bekannte Gleichung 'Algorithm=Logic+Control' wird zum Stolperstein, da nicht mehr die (wichtigere) Komponente der 'Logic' des Alogorithmus im Vordergrund steht, sondern sogar fast völlig in der Formulierung der 'Control' in einem primitiven Formalismus untergeht.
In der Anfangszeit der Informatik war diese Problematik relativ unwichtig: Computer waren langsam und hatten nur kleine Speicherkapazitäten. Komplexere Probleme liessen sich von vorneherein nicht formulieren, da es keine Rechner gab, die in der Lage gewesen wären, sie zu lösen.
Mit dem immensen Anstieg der Kapazitäten der Rechner wurden auch die gestellten Probleme immer umfangreicher und komplexer. Die Entwicklung der imperativen Sprachen versuchte, sich diesem Fortschritt anzupassen, indem abstraktere Konzepte eingeführt wurden[2]. Durch das oben erklärte fundamentale Defizit dieser Sprachen konnte aber der Graben zwischen Ausdrucksfähigkeit der Sprachen und Komplexität der zu lösenden Probleme nicht geschlossen werden und dies führte schließlich zur sogenannten 'Softwarekrise'. Die Hauptarbeit beim Erstellen eines Programmes wurde nicht mehr dazu aufgewandt, das Problem zu formulieren und den Programmtext zu schreiben, sondern in der Suche nach und im Beseitigen von Fehlern im Programm verschwendet. Diese Tatsache ist besonders gravierend, weil imperative Sprachen durch ihre primitive Semantik eine Vielzahl von Gelegenheiten bieten, Fehler einzubauen, ohne gleichzeitig Möglichkeiten zur formalen Verifikation von Programmen zu bieten, d.h. es ist praktisch sehr schwer, formal über die Korrektheit eines imperativen Programmes zu argumentieren.
Für die Zukunft ist auch kein Weg in Sicht, der dieses Dilemma beheben kann, nur Linderungen dieser 'Krankheit' sind möglich und werden praktiziert[3].
Daher hat sich in jüngster Zeit das Interesse wieder[4] den deklarativen Sprachen zugewandt. Die Idee, die allen ihren Vertretern zu Grunde liegt ist einfach: Man beschreibt nicht mehr, *wie* ein Programm etwas tun soll, sondern nur noch, *was* es tun soll. Man vernachläßigt also die 'Control' und konzentriert sich auf die 'Logic'.
Die Hauptrichtungen innerhalb der deklarativen Sprachen, wie sie heute anzutreffen sind, lassen sich als *funktionales Paradigma* und *relationales Paradigma* beschreiben.
Beim funktionalen Paradigma wird das Problem anhand von Gleichungen und einem auszuwertenden Ausdruck beschrieben. Dieser Ausdruck wird anhand der Gleichungen immer wieder umgeformt, bis er eine Endform erreicht hat.
Beim relationalen Paradigma, welches meistens als 'logisches Paradigma' bezeichnet wird, werden Relationen angegeben. Die Gesamtheit dieser Relationen werden als Axiome einer Logik aufgefaßt und das Lösen eines Problems wird auf das Auffinden eines Beweises für eine Aussage (Goal) zurückgeführt.
Beiden gemein ist die Ausdrucksfähigkeit ihrer Konzepte. Der Programmierer kann sich darauf konzentrieren, anzugeben, wie ein Problem aussieht. Die Frage, wie das Problem wirklich zu lösen ist, bleibt dem Computer überlassen. Dabei ist die Semantik beider Pa-

[2]Wie Unterprogramme, getypte Sprachen, etc.

[3]Etwa die Systematisierung des Entwicklungsprozesses von Software, oder die Einführung neuer Ideen, wie objektorientierte Programmierung.

[4]'Wieder', weil z.B. schon LISP deklarative Konzepte beinhaltete.

radigma einfach, aber keineswegs primitiv, wie die Semantik imperativer Sprachen. Dies erlaubt es sogar, formal über deklarative Programme zu argumentieren; Korrektheitsbeweise sind um einiges leichter, als bei imperativen Sprachen. Auch die Modularisierung und Kombination deklarativer Programme wird wesentlich erleichtert.

Es ist keineswegs so, daß die Probleme, die in ihrer Gesamtheit die 'Softwarekrise' ausmachen, mit der Verwendung deklarativer Sprachen einfach verschwinden würden. Deklarative Sprachen haben allerdings Eigenschaften, die diese Probleme stark vermindern und beherrschbarer machen können.

Die deklarativen Sprachen wurden (und werden teilweise immer noch) als langsam und speicherhungrig verschrien. In der Tat ist der Bedarf einer deklarativen Sprache an Speicherplatz zur Repräsentation ihrer Daten oft erheblich höher, als der einer imperativen Sprache. Dies ist der Preis, den man für die abstrakteren Formulierungsmöglichkeiten der deklarativen Sprachen zahlen muß. Aber im Vergleich zu früheren Implementierungen sind heutige deklarative Sprachen schon sehr viel effizienter geworden. Und die Forschung auf diesem Gebiet ist noch keineswegs als abgeschloßen zu betrachten. Außerdem sind Fragen des Speicherverbrauchs bei den heute verfügbaren Hauptspeichermengen nicht mehr ganz so brisant, wie früher.

1.2 Logisch–funktionale Sprachen

Nachdem die Forschung auf dem Gebiet dieser beiden Paradigmen lange Zeit getrennt vorangetrieben wurde, wird seit einiger Zeit darüber nachgedacht, die Vorteile beider Ansätze zu verbinden, um einen noch ausdrucksvolleren Formalismus zu erhalten.

Bei den relationalen Sprachen stört vor allem die umständliche Formulierung mittels Prädikaten (Relationen), während das selbstständige Auffinden einer Lösung zu den großen Stärken dieser Richtung gehört.

Andererseits glänzen die funktionalen Sprachen durch die Eleganz ihres Formalismusses, der Annotation eines Programmes über Funktionsgleichungen.

Daher ist ein naheliegender Ansatz für die Verbindung beider Paradigmen, von den funktionalen Sprachen das *Funktionskonzept*, von den relationalen Sprachen das *Programmieren als Lösen von Anfragen*, zu übernehmen.

Dabei werden Funktionen meistens durch *Termersetzungssysteme (TRS)* implementiert: d.h. statt Gleichungen verwendet man *Regeln*: gerichtete Gleichungen. Der Begriff der *Gleichheit* wird dann als 'Überführbarkeit durch Regelanwendung' definiert.

Die Semantik dieser Sprachen ist über *semantische Unifikation* gegeben: Zu einer Anfrage (Goal) $s =_? t$ werden Lösungen (Substitutionen) σ gesucht, so daß $\sigma(s) =_{\mathcal{R}} \sigma(t)$, wobei die Gleichheit $s' =_{\mathcal{R}} t'$ geben ist durch die syntaktische Gleichheit der Normalformen von s' und t'.

Beispiel 1.1: Addition auf $\mathbb{N}$

$$
\begin{array}{l}
\mathcal{R} = \{ \\
\ 0 + X \quad\ \ \rightarrow X \ , \\
\ s(X) + Y \rightarrow s(X+Y) \\
\} \\
\\
X + Y \quad\ \ =_? s(s(0))
\end{array}
$$

Beispiel 1.1 zeigt ein semantisches Unifikationsproblem: Gegeben die Regeln für Addition auf den natürlichen Zahlen (die dargestellt werden als 0 und $s(X)$), finde Lösungen für die Anfrage $X + Y =_? 2$.
Man könnte alle Lösungen finden, indem man für X, Y einfach alle Terme aufzählt, einsetzt und dann prüft, ob die beiden Seiten gleich sind, indem man sie zu ihren Normalformen reduziert.
Bis heute wurden eine ganze Reihe von semantischen Unifikationsverfahren untersucht, wobei die meisten auf dem *Narrowing* aufbauen, welches zuerst in [Hul80] beschrieben wurde.
Narrowing versucht, die Menge der Terme, die für Variablen eingesetzt werden können zu *verengen*[5], indem die Terme ausgeschloßen werden, die nicht passen können, weil sonst später eine Reduktion zu einer Normalform nicht möglich wäre.
Eine Art Endpunkt in der Untersuchung von Narrowing Verfahren wurde mit dem *Needed–Narrowing* [AEH94] erreicht, das anhand einer recht komplizierten Auswahlfunktion den Suchraum vermindert.
Eine andere Art von Verfahren bilden die *Dekompositionsverfahren*, die nicht mehr, wie die Narrowing Verfahren, auf einer direkten Korrespondenz zwischen Rewriteschritten und Narrowingschritten aufbauen, sondern ein Goal in kleinere Subgoals zerlegen, in denen nach Lösungen gesucht wird. Natürlich läßt sich immer noch eine Beziehung zwischen Rewrite- und Suchschritten herstellen, nur ist sie komplizierter als bei den Narrowing Verfahren.

Bis heute wurde eine ganze Reihe von Verfahren zur semantischen Unifikation vorgeschlagen, etwa [BGM88, BKW92, BGOT95, BT94, DG89, DMS92, Ech90, Fay79, Fri85, AEH94, Her86, Höl89, Hul80, MNKLRA90, NRS89, You89], siehe [Han94] für eine Zusammenfassung der Verfahren.
Daraus erwächst das Problem, daß zwar von vielen Autoren Verfahren vorgeschlagen werden, ein objektiver Vergleich der Stärken und Schwächen dieser Verfahren aber nicht möglich ist. Dies hat einige triviale Gründe:

- Theoretische Effizienzvergleiche sind schwer durchführbar, da zu viele Faktoren ein-

[5] Daher der Name: to narrow = verengen.

fließen. Z.B. das benutzte TRS, die Form des Goals, etc.pp. Daher werden (wenn überhaupt) nur einige kleine empirische Vergleiche mit anderen Verfahren gemacht. Es lassen sich also keine allgemeingültigen Aussagen über die Vorteile einer Strategie geben, im Gegensatz zu den funktionalen Sprachen.

- Es gibt kaum Implementierungen verschiedener Verfahren auf einer gemeinsamen Plattform[6]. Vergleiche mit wirklichkeitsnaheren, da größeren, Beispielen sind also praktisch unmöglich.

- Viele Verfahren sind nichtdeterministisch definiert: Die nötige Auflösung dieses Nichtdeterminismusses für eine Implementierung wird oft nicht angegeben. Auch werden die Auswirkungen dieser Auflösung auf die Performanz des Verfahrens nicht betrachtet.

Die vorliegende Diplomarbeit versucht dieses Manko zu beheben: Es wurde ein System, Quetzal[7], definiert und implementiert, das es erlaubt semantische Unifikationsverfahren mittels einer Beschreibungssprache zu definieren und zu Code für eine abstrakte Maschine zu übersetzen. Das System erlaubt es, alle relevanten Daten (TRSs, Goals, Signaturen, etc.) zu definieren und die Ergebnisse von Lösungssuchen zu protokollieren, inklusive relevanter Daten für den Effizienzvergleich von Verfahren[8].
Durch das System werden Effizienzvergleiche verschiedener Verfahren unter gleichen Voraussetzungen möglich. Durch die wirkliche Implementierung auf *einer* Maschine wird die praktische Realisierbarkeit der Verfahren testbar, sowie die Auswirkungen der Auflösung des Nichtdeterminismusses beobachtbar.

1.3 Überblick über das System

1.3.1 Idee

Die Idee hinter Quetzal besteht darin, den Prozeß des Findens von Lösungen zu einem Goal als *Manipulation einer Goalliste* aufzufassen. Eine Goalliste wird solange anhand (für jede Strategie) gegebener *Meta-Regeln* umgeformt, bis sie leer ist. Dann muß aus (explizit) mitberechneten Daten die Lösung ausgegeben werden.
Für Narrowing z.B. besteht die Transformation darin, einen Subterm mir der linken Seite einer Regel zu unifizieren und durch die rechte Regelseite zu ersetzen und den Unifikator anzuwenden. Die kombinierten Unifikatoren einer Ableitung werden als Daten mitberechnet und am Schluß ausgegeben.

[6] Eine Ausnahme ist das PaNaMa System [BT95].

[7] Quetzal: [ket-'sa^:l] lat. Pharomachrus mocinno. Ein Vogel aus der Familie der Trogone. Verbreitet in Mittelamerika, der männliche Vogel hat ein herrlich schillerndes Gefieder und sehr lange Schwanzfedern. Der Quetzal ist, insbesondere in Guatemala und Mexiko, ein seit langer Zeit von den Maya verehrtes Tier. Er wird mit dem Gott Quetzalcoatl, Grünfederschlange, assoziiert.

[8] Z.B. die Anzahl gerechneter Schritte, Rechenzeit, Reduktionsschritte, ...

Diese Idee erlaubt gleichermassen die Formulierung von Narrowing– und Dekompositionsverfahren.
Ein Verfahren wird einfach dadurch implementiert, daß

1. Eine explizite Auflösung des Nichtdeterminismusses in Form einer Ablaufsteuerung angegeben wird.

2. Die mitzuberechnenden Daten definiert werden.

3. Die Metaregeln gegeben werden.

Quetzal erlaubt es, mit einer Beschreibungssprache[9], sowohl Verfahren, als auch alle anderen benötigten Objekte, wie Signaturen, Regeln, Goals, TRS, etc., zu definieren.
Diese Programme in der $\mathcal{QDL}$ werden zu Code für eine *abstrakte Maschine*[10] compiliert und auf ihr ausgeführt.
Die $\mathcal{QDL}$ dient zur Definition der Objekte. Ihre *Manipulation*, etwa das Lösen eines Goals, geschieht über die Ausführung von *Aktionen*.
Es gibt Aktionen, um Goals zu lösen, Parameter des Systems zu beeinflußen, Objekte zu löschen oder anzuzeigen. Ebenfalls implementiert sind Aktionen zum Testen der Terminierung und Konfluenz eines TRS.
Die Semantik des Systems wird also durch die kombinierte Semantik der Definitionen und Aktionen definiert, die $\mathcal{QDL}$ dient zur Definition der Objekte, die Aktionen zur Erzeugung der Ergebnisse.

1.3.2 Systemaufbau

Das System besteht aus zwei Hauptbestandteilen:

1. Das *Frontend* (FE) regelt die Interaktion zwischen dem Benutzer und dem Backend. Es ist verantwortlich dafür, die Befehle des Benutzers in simple Kommandos an das Backend umzuwandeln und bereitet die Ausgaben des Backends für die Anzeige zum Benutzer auf.

2. Das *Backend* (BE) besteht aus den Teilen *Parser, Zwischendarstellung*[11]*, Compiler, Abstrakte Maschine ($\mathcal{QAM}$)*. Der Parser nimmt ein (vom FE vorverarbeitetes) Programm und erzeugt daraus eine Zwischendarstellung, die zu der schon vorhandenen Darstellung hinzugefügt wird. Sind alle Objekte definiert, wird die Compilation angestoßen. Diese erzeugt aus der $\mathcal{QIR}$ $\mathcal{QAM}$-Code und speichert ihn. Soll schließlich ein Goal gelöst werden, wird dieser Code auf der $\mathcal{QAM}$ ausgeführt.

[9] $\mathcal{QDL}$, für $\mathcal{Q}$uetzal $\mathcal{D}$escription $\mathcal{L}$anguage
[10] $\mathcal{QAM}$, für $\mathcal{Q}$uetzal $\mathcal{A}$bstract $\mathcal{M}$achine.
[11] $\mathcal{QIR}$, für $\mathcal{Q}$uetzal $\mathcal{I}$ntermediate $\mathcal{R}$epresentation.

Die Ausgaben des FE und des BE sind zweigeteilt: Einmal werden *Ergebnisse* geliefert, also die Lösungen des Goals. Zum anderen wird ein *Protokoll* erstellt, das darüber Auskunft gibt, *wie* gerechnet wurde. Z.B. sind darin auch Informationen über Sackgassen, angewandte Metaregeln, etc. enthalten.
Abbildung 1.1 verdeutlicht das Zusammenspiel der Teile.

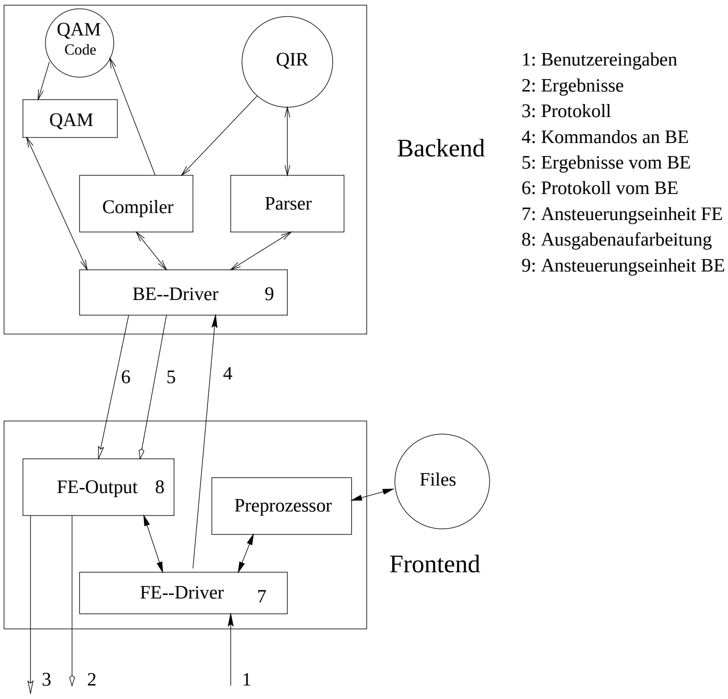


Abbildung 1.1: Systemaufbau

1.3.3 Ziele des Systems

Die Zielsetzung von Quetzal ist es,

- Vergleichbarkeit von Verfahren herzustellen, durch Implementierung auf ein und derselben (abstrakten) Maschine.
- 'Rapid Prototyping' möglich zu machen: Verfahrensimplementierungen sollen schnell verfügbar und die Verfahren damit schnell testbar sein.

Darüber hinaus erfüllt das System folgende Anforderungen:

- Effizienz. Bei Suchproblemen ist ein, durch eine Implementierung hinzukommender, (großer) Overhead nicht tragbar.
- Universalität. Möglichst alle Verfahren sollen sich implementieren lassen.

- Transparenz. Die Auswirkungen einer Implementierung sollen *vorher* überschaubar sein. Dies erfordert u.a. eine klare Definition der Semantik und eine hinreichende Spezifikation des Systems an sich.
- Leichte Bedienung. Der Umgang mit einem so komplexen System muß durch eine geeignete Benutzeroberfläche erleichtert werden.
- Portierbarkeit. Ist eine Standardanforderung und dringend notwendig, wenn das System Verbreitung finden soll.

Nach Abwägung der obigen Kriterien wurde als Implementierungssprache für Quetzal C gewählt. Funktionale Sprachen würden zwar eine schnellere und fehlerfreiere Implementierung erlauben, haben aber den Nachteil eines zu großen Overheads bei der Speicherverwaltung, was sie für dieses Projekt ungeeignet macht.
Quetzal ist weiterhin *modular* aufgebaut. Nicht nur sind das Frontend und Backend getrennt, sondern es ist auch möglich, *verschiedene* Frontends zu benutzen. Implementiert wurden:

1. Ein *Batch* Frontend, das er erlaubt, Quetzal als 'Compiler' zu benutzen, indem alle Kommandos in der Befehlszeile angegeben werden und die Ausgaben in Files erfolgt.
2. Ein *Terminal* Frontend, welches die Steuerung über ein normales Terminal erlaubt.

Darüber hinaus wurde versucht, die verschiedenen Komponenten des Systems, wie Parser, Compiler, etc., auch in der Implementierung möglichst zu trennen.
Die Protierbarkeit erlaubte es, Versionen für

1. SPARC1/5/10/20 Stations unter Solaris2.4
2. Amiga Rechner unter NetBSD1.1
3. Amiga Rechner unter AmigaDOS3.0

zu implementieren. Die Installation der einzelnen Versionen geschieht automatisch mittels von GNU **autoconf** ([Mac95]) erzeugten Konfigurationsskripts.

1.4 Aufbau der Arbeit

Die Arbeit ist in drei Teile aufgeteilt:

1. Theorie.

   Dieser Teil (I) handelt die nötige Theorie in ausreichendem Umfang ab.

   In Kapitel 3 werden alle relevanten Definitionen und Schreibweisen für die Theorie der semantischen Unifikation eingeführt.

Kapitel 4 folgt mit der Definition der Semantik der Beschreibungssprache unter den Aktionen.

In Kapitel 5 werden die abstrakte Maschine und die Übersetzung von $\mathcal{QIR}$ zu $\mathcal{QAM}$-Programmen besprochen.

In Kapitel 6 werden einige semantische Unifikationsverfahren vorgestellt und ihre Implementierung im Rahmen von Quetzal behandelt.

2. Implementierung.

   In diesem Teil der Arbeit (II) werden die Prinzipien, Verfahren und der Verlauf der Implementierung des Systems auf den drei unterstützten Architekturen beschrieben. Alle sich aus der Implementierung ergebenden Einschränkungen[12] werden aufgeführt und begründet.

3. Ergebnisse.

   In diesem Teil (III) werden die Erfahrungen mit dem System beschrieben, insbesondere werden die Implementierungen der Verfahren aus Abschnitt 6 verglichen.

Ein Ausblick auf künftige Erweiterungen schließt die Arbeit ab.

[12] Wie etwa der erlaubte Wertebereich einer ganzen Zahl.

Teil I

Theorie

[...] Von jedem Gegenstande des Gedankenkreises bilde dir einen genauen, bestimmten Begriff, so daß du denselben nach seiner wirklichen Beschaffenheit unverhüllt, ganz und nach allen seinen Bestandteilen anschaulich zu erkennen und ihn selbst sowohl, als auch die einzelnen Merkmale, aus denen er zusammengesetzt ist und in die er wieder aufgelöst wird, mit ihren richtigen Namen zu bezeichnen vermagst.

Marc Aurel, Selbstbetrachtungen III/11

Kapitel 2

Vorbemerkung

Für ein System, das zur Implementierung, Vergleich und Erprobung verschiedenster semantischer Unifikationsverfahren dienen soll, ist eine gründliche und saubere theoretische Verankerung unverzichtbar.
Die Effekte einer Verfahrensimplementierung müssen von vornherein absehbar sein, was sich letztendlich nur durch eine saubere mathematische Definition der Semantik des Systems realisieren läßt.
Deshalb werden sowohl die theoretischen Grundlagen der semantischen Unifikation, als auch die Semantik des Systems ausführlich vorgestellt.
Dies verhindert Mißverständnisse über z.B. die Details der Theorie der semantischen Unifikation, etwa die zulässige Gestalt der benutzten Regeln.
Im nächsten Kapitel werden die Grundlagen der semantischen Unifikation eingeführt. Leider gibt es keinen allgemein akzeptierten Standard der Notation, deshalb wurde versucht, auf unterschiedliche Schreibweisen in der Literatur hinzuweisen. Auf einige Stolpersteine in den Feinheiten und Implikationen der zugrundeliegenden Theorie wird ebenfalls hingewiesen.
Das darauf folgende Kapitel startet mit einer (intuitiv gehaltenen) Einführung in die Semantik des Systems. Anhand von Beispielen sollen sowohl die Ideen hinter, als auch die Wirkungsweisen des Systems erklärt werden. Da diese Einführung aus Verständnisgründen auf eine theoretische Fundierung verzichten muß, wird diese in den folgenden Abschnitten nachgeholt. Dabei wird sowohl die Sprache an sich definiert, als auch ihre Semantik. Die Definition der Semantik gliedert sich wie üblich in zwei Teile: zuerst wird die *statische Semantik* gegeben, d.h. die Zusatzbedingungen, die ein Programm über die Syntax hinaus erfüllen muß. Hierbei werden etwa die Sichtbarkeitsregeln für Namen und das verwendete *Typsystem* definiert. Als zweites folgt die *dynamische Semantik*, die jedem Programm, welches die Bedingungen der statischen Semantik erfüllt, seine Bedeutung zuordnet. Hier werden dann die Fragen nach der Auswertungsstrategie von Ausdrücken, der Reihenfolge von Suchprozessen, etc. geklärt.
Beide Semantiken werden als Sätze von *Inferenzregeln* gegeben, genauer als eine *natural semantics* [Kah87]. Diese Methode ist gleichermassen für beide Arten von Semantikdefinitionen geeignet. Die Lesbarkeit und Einfachheit ihres Formalismusses macht sie einer

denotationellen Semantik überlegen. Beide Semantiken beinhalten Konsistenzlemmata, die gewisse Eigenschaften von korrekten, d.h. über den Inferenzregeln ableitbaren, Programmen beweisen, etwa den Sichtbarkeitsbereich von Namen, etc.
Da Quetzal seine Programme zu Code für eine abstrakte Maschine übersetzt, wird im darauffolgendem Kapitel diese Maschine definiert. Auch hier und in der Angabe der Übersetzung zu Code für diese Maschine kommt wieder eine Natural Semantics zum Einsatz.
Zum Abschluß des Theorieteils werden einige semantische Unifikationsverfahren vorgestellt. Die Implementierung dieser Verfahren in der Beschreibungssprache wird jeweils angegeben.

Kapitel 3

Semantische Unifikation

In diesem Kapitel werden die grundlegenden Begriffe der semantischen Unifikation vorgestellt. Es werden die allgemein gebräuchlichen Begriffe verwandt, ggf. wird auf Unterschiede zu bekannteren Arbeiten hingewiesen. Im übrigen sei auf die "Standardwerke" [DJ90, Han94, Klo92] verwiesen.

3.1 Signaturen, Terme, Substitutionen

Wir beginnen mit der Definition des Begriffes einer *Signatur* und den darüber baubaren *Termen*.

Definition 3.1: (Signatur)
Eine *sortierte Signatur* Σ ist ein Tripel $(\mathcal{S}, \mathcal{OP}, <_\Sigma)$, wobei $\mathcal{S}$ eine endliche Menge von *Sorten* ist, $\mathcal{OP}$ eine endliche Menge von *Operatorsymbolen* der Form $f_{w \to s}$, wobei $w \in \mathcal{S}^*, s \in \mathcal{S}$ und $<_\Sigma$ eine strikte Partialordnung[1] auf $\mathcal{S}$ ist.

Es gilt $\mathcal{OP} \cap \mathcal{S} = \emptyset$. Die Menge $\mathcal{OP}$ wird weiter unterteilt in zwei disjunkte Mengen $\mathcal{C}, \mathcal{F}$, mit $\mathcal{OP} = \mathcal{C} \cup \mathcal{F}$ von *Constructoren* und *Functoren*.

Für alle $f_{w \to s} \in \mathcal{OP}$ gilt: $\nexists f_{w' \to s'} \in \mathcal{OP} : w' \neq w \vee s' \neq s$, d. h. Überladung von Operatorsymbolen ist nicht erlaubt. Im übrigen nennt man $w \to s$ die *Stelligkeit*, w die *Argumentsorten*, s die *Ergebnissorte* von f.

Zu dieser Definition sind einige Bemerkungen angebracht:

- Diese Definition einer Signatur ist eine Verallgemeinerung anderer Signaturdefinitionen. Insbesondere sind many-sorted Signaturen darin enthalten, wenn man $<_\Sigma = \emptyset$ wählt. Da sortenlose Signaturen ein Spezialfall der many-sorted Signaturen sind, sind sie ebenfalls in dieser Definition enthalten (Man hat dann nur eine Sorte, d. h. $\mathcal{S} = \{s\}$ und die einzige Information in der Stelligkeit eines Operatorsymbols $f_{w \to s}$ bleibt die Anzahl der Argumentsorten, d. h. die Länge von w.)

[1] D.h. irreflexiv und transitiv.

- Da $\mathcal{S}$ endlich ist und $<_\Sigma$ eine strikte Partialordnung ist, ist $<_\Sigma$ auch *wohlfundiert*, d. h. es gibt keine unendlichen Ketten der Form $s_1 <_\Sigma s_2 <_\Sigma \cdots$[2].
- Wir definieren $s \leq_\Sigma s' \Leftrightarrow s <_\Sigma s' \vee s = s'$. $\leq_\Sigma$ ist dann eine Partialordnung[3].
- Statt $f_{w \to s}$ schreiben wir oft nur f, wenn die Stelligkeit ohne Bedeutung ist.

Eine Signatur legt fest, welche *Terme* man bauen kann:

Definition 3.2: (Terme)

Gegeben eine Signatur Σ und eine abzählbar unendliche Menge von *Variablen* $\mathcal{X}$, wobei $\mathcal{X} = \bigcup\limits_{s \in \mathcal{S}} \mathcal{X}\!:\!s$ aus disjunkten Mengen $\mathcal{X}\!:\!s$ von Variablen der Sorte s besteht und immer $\mathcal{X} \cap (\mathcal{F} \cup \mathcal{S}) = \emptyset$ gilt.

Die Menge $\mathcal{T}(\Sigma, \mathcal{X}) : s$ der *Terme der Sorte s mit Variablen* ist definiert als die kleinste Menge, die folgende Bedingungen erfüllt:

1. $\forall_{x \in \mathcal{X}:s} : x \in \mathcal{T}(\Sigma, \mathcal{X}) : s$
2. $\forall_{f_{\to s} \in \mathcal{OP}} : f \in \mathcal{T}(\Sigma, \mathcal{X}) : s$
3. $\forall_{t_1 \in \mathcal{T}(\Sigma,\mathcal{X}):s_1, \ldots, t_n \in \mathcal{T}(\Sigma,\mathcal{X}):s_n, f_{s_1 \cdots s_n \to s} \in \mathcal{OP}, n \geq 1} : f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{X}) : s$
4. $\forall_{t \in \mathcal{T}(\Sigma,\mathcal{X}):s,\, s' \in \mathcal{S},\, s \leq_\Sigma s'} : t \in \mathcal{T}(\Sigma, \mathcal{X}) : s'$

Die *Menge aller Terme über* Σ, $\mathcal{T}(\Sigma, \mathcal{X})$, ist dann definiert als:

$\mathcal{T}(\Sigma, \mathcal{X}) := \bigcup\limits_{s \in \mathcal{S}} \mathcal{T}(\Sigma, \mathcal{X}) : s$

Ein Term $t \in \mathcal{T}(\Sigma, \mathcal{X})$ heißt *Constructorterm* gdw. in ihm keine Operatorsymbole aus $\mathcal{F}$ vorkommen.

Die *Wurzel* $root(t)$ eines Terms t ist definiert als:

i. $\forall_{x \in \mathcal{X}} : root(x) := x$
ii. $\forall_{f_{\to s} \in \mathcal{OP}} : root(f) := f$
iii. $\forall_{f_{s_1 \cdots s_n \to s} \in \mathcal{OP}, n \geq 1} : root(f(t_1, \ldots, t_n)) := f$

Dazu bleibt zu bemerken:

- 2.) ist eigentlich nur ein Spezialfall von 3.), erlaubt uns aber statt $f()$ einfach f zu schreiben. Im übrigen nennen wir so ein f dann *Konstante*.
- Über 4.) kommt die Sorteninklusion zum Tragen: Ein Term einer "kleineren" Sorte ist auch ein Term der "größeren" Sorte, aber nicht umgekehrt. D.h. $\mathcal{T}(\Sigma, \mathcal{X}) : s \subseteq \mathcal{T}(\Sigma, \mathcal{X}) : s' \Leftarrow s <_\Sigma s'$.

[2] Dies sieht man leicht ein: Gäbe es nämlich eine unendliche Kette, dann (da $\mathcal{S}$ endlich ist) gibt es s_i, s_j mit $i < j$, $s_i = s_j$. Daraus folgt, da $<_\Sigma$ transitiv ist: $s_i <_\Sigma s_j$. Das ergibt $s_i <_\Sigma s_i$, was aber die Irreflexivität von $<_\Sigma$ verletzt.

[3] D.h. reflexiv, antisymmetrisch und transitiv.

- Für jede Sorte $s \in \mathcal{S}$ verlangen wir, daß $\mathcal{T}(\Sigma, \mathcal{X}) : s \neq \emptyset$, es gibt also immer einen Term dieser Sorte.
- Falls jede Variable in einem Term t genau einmal vorkommt, nennt man t auch *linear*.

Zur Manipulation von Termen führen wir nun einige Begriffe ein.

Definition 3.3: (Vorkommen, Subterm, Subtermersetzung)
Sei $t \in \mathcal{T}(\Sigma, \mathcal{X})$, für eine Signatur Σ und Variablenmenge $\mathcal{X}$.

Die Menge $\mathcal{O}(t) \subset \mathbb{N}^*$ der *Vorkommen* (oder Positionen) in t ist wie folgt definiert[4]:

1. $\forall_{x \in \mathcal{X}} : \mathcal{O}(x) := \{\epsilon\}$
2. $\forall_{f_{\to s} \in \mathcal{OP}} : \mathcal{O}(f) := \{\epsilon\}$
3. $\forall_{f_{s_1 \cdots s_n \to s} \in \mathcal{OP}, n \geq 1} : \mathcal{O}(f(t_1, \ldots, t_n)) := \{i.p | 1 \leq i \leq n, p \in \mathcal{O}(t_i)\} \cup \{\epsilon\}$

Die Menge $\bar{\mathcal{O}}(t)$ der *nicht-variablen Vorkommen* des Terms t ist definiert als:

1. $\forall_{x \in \mathcal{X}} : \bar{\mathcal{O}}(x) := \emptyset$
2. $\forall_{f_{\to s} \in \mathcal{OP}} : \bar{\mathcal{O}}(f) := \{\epsilon\}$
3. $\forall_{f_{s_1 \cdots s_n \to s} \in \mathcal{OP}, n \geq 1} : \bar{\mathcal{O}}(f(t_1, \ldots, t_n)) := \{i.p | 1 \leq i \leq n, p \in \bar{\mathcal{O}}(t_i)\} \cup \{\epsilon\}$

Für zwei Vorkommen u, v bezeichnet $u.v$ auch die Konkatenation von Positionen. Der Operator . wird also überladen.

Die *Präfixordnung* $\leq_p \subset \mathbb{N}^* \times \mathbb{N}^*$ ist definiert als: $u \leq_p v \Leftrightarrow \exists w : u.w = v$. Wir schreiben $u <_p v$, gdw. $u \leq_p v \wedge u \neq v$. Die *Höhe* einen Terms t ist definiert als die Länge des längsten Vorkommens in $\mathcal{O}(t)$: $height(t) := \max\limits_{p_i \in \mathcal{O}(t)} \{\|p_i\|\}$

Für einen Term $t \in \mathcal{T}(\Sigma, \mathcal{X})$ definieren wir seine *Sorte* $sort(t)$:

1. $\forall_{x \in \mathcal{X}_s} : sort(x) := s$
2. $\forall_{f_{\to s} \in \mathcal{OP}} : sort(f) := s$
3. $\forall_{f_{s_1 \cdots s_n \to s} \in \mathcal{OP}, n \geq 1} : sort(f(t_1, \ldots, t_n)) := s$

Der *Subterm* t/p eines Terms t an einer Position $p \in \mathcal{O}(t)$ ist definiert über:

1. $t/\epsilon := t$
2. $f(t_1, \ldots, t_n)/i.p := t_i/p$

Die Ersetzung des Subterms t/p durch einen Term t' wird annotiert als $t[p \leftarrow t']$ und ist formal definiert als:

1. $t[\epsilon \leftarrow t'] := t'$

[4] Im Englischen wird $\mathcal{O}(t)$ als Menge der *occurences* von t bezeichnet.

2. $f(t_1, \ldots, t_n)[i.p \leftarrow t'] := f(t_1, \ldots, t_{i-1}, t_i[p \leftarrow t'], t_{i+1}, \ldots, t_n)$

Die Menge der in einem Term t vorkommenden Variablen $\mathcal{V}(t)$ ist definiert als:

1. $\forall_{x \in \mathcal{X}} : \mathcal{V}(x) := \{x\}$
2. $\forall_{f_{\rightarrow s} \in \mathcal{OP}} : \mathcal{V}(f) := \emptyset$
3. $\forall_{f_{s_1 \cdots s_n \rightarrow s} \in \mathcal{OP}, n \geq 1} : \mathcal{V}(f(t_1, \ldots, t_n)) := \bigcup_{1 \leq i \leq n} \mathcal{V}(t_i)$

Falls $\mathcal{V}(t) = \emptyset$ für einen Term t, dann nennen wir t einen *Grundterm*.

Eine Relation $R \subseteq \mathcal{T}(\Sigma, \mathcal{X}) \times \mathcal{T}(\Sigma, \mathcal{X})$ heißt *sorten-kompatibel*, dann, wenn $\forall_{t,t' \in \mathcal{T}(\Sigma,\mathcal{X})} : t\, R\, t' \Rightarrow \exists s \in \mathcal{S} : t, t' \in \mathcal{T}(\Sigma, \mathcal{X}) : s$.

Bemerkungen:

- Statt t/p wird in der Literatur oft $t_{|p}$ geschrieben.
- Für $t[p \leftarrow t']$ wird manchmal $t[t']_p$ geschrieben
- $\mathcal{T}(\Sigma, \emptyset)$ sind alle Grundterme über einer gegebenen Signatur Σ.

Variablen stehen in Termen als eine Art Platzhalter. *Substitutionen* ersetzen sie durch andere Terme:

Definition 3.4: (Substitution)

Eine *Substitution* ist eine Funktion σ, die Variablen auf Terme abbildet: $\sigma : \mathcal{X} \rightarrow \mathcal{T}(\Sigma, \mathcal{X})$, so daß die Menge $\mathcal{D}(\sigma) := \{x \in \mathcal{X} | \sigma(x) \neq x\}$ endlich ist. $\mathcal{D}(\sigma)$ heißt *Domäne*[5] von σ und enthält alle Variablen, die von σ nicht auf sich selbst abgebildet werden.

Da $\mathcal{D}(\sigma)$ endlich ist, kann man Substitutionen als eine endlich Menge schreiben der Form $\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$, mit $x_i \neq x_j \Leftrightarrow i \neq j$, $\forall_{1 \leq i \leq n} : \exists s_i \in \mathcal{S} : x_i, t_i \in \mathcal{T}(\Sigma, \mathcal{X}) : s_i$.

Die Menge der von einer Substitution σ *eingeführten Variablen*[6], $\mathcal{I}(\sigma)$, ist definiert als $\mathcal{I}(\sigma) := \bigcup_{x \in \mathcal{D}(\sigma)} \mathcal{V}(\sigma(x))$.

Die *Beschränkung* einer Substitution auf eine Menge $V \subseteq \mathcal{X}$, d. h. $\sigma_{|V}$ ist definiert durch: $\sigma_{|V} := \{x \leftarrow t \in \sigma | x \in V\}$.

Die Erweiterung von Substitutionen von Variablen auf Terme ist für eine Substitution σ definiert als:

1. $\forall_{x \in \mathcal{X}} : \sigma'(x) := \begin{cases} t\,, & \text{falls } \exists x \leftarrow t \in \sigma \\ x\,, & \text{sonst} \end{cases}$

[5] Englisch: *domain.*
[6] Englisch: *introduced variables.*

2. $\forall_{f_{\to s} \in \mathcal{OP}} : \sigma'(f) := f$

3. $\forall_{f_{s_1 \cdots s_n \to s} \in \mathcal{OP}, n \geq 1} : \sigma'(f(t_1, \ldots, t_n)) := f(\sigma'(t_1), \ldots, \sigma'(t_n))$

Im folgenden werden wir nicht mehr zwischen beiden unterscheiden und identifizieren σ und σ'. Auch schreiben wir statt $\sigma(t)$ nur $t\sigma$.

Eine Substitution σ heißt *Umbenennung*, gdw. sie, eingeschränkt auf Variablen, eine bijektive Abbildung ist, d.h. gdw.

$$\lambda : \mathcal{X} \to \mathcal{X},\ \lambda(x) := x\sigma,\ \text{falls } x\sigma \in \mathcal{X}$$

total und bijektiv ist.

Beispiel 3.5: Eine Signatur und einige Terme

$$\begin{aligned}
&\Sigma = (\{nat, list, element\},\\
&\qquad \{0 :\to nat, s : nat \to nat, nil :\to list, cons : element\ list \to list\} \cup\\
&\qquad \{+ : nat\, nat \to nat,\ app : list\, list \to list\}, \{nat <_\Sigma element\})\\
&\mathcal{X} = \{X, Y, \ldots\}\\
&\mathcal{T}(\Sigma, \mathcal{X}) \supset \{0, s(X), nil, 0 + s(0), cons(0, cons(s(0) + s(s(X)), Y))\}
\end{aligned}$$

3.2 TRS, Ableitung

In der Theorie der semantischen Unifikation implementieren Termersetzungssysteme die Funktionen im Rahmen logisch-funktionaler Sprachen.

Definition 3.6: (Termersetzungssysteme, Ableitung)
Ein *Termersetzungssystem (TRS)* $\mathcal{R}$ ist eine endliche Menge von *Regeln*. Eine *Regel* ist ein geordnetes Paar $l \to r$ aus $\mathcal{T}(\Sigma, \mathcal{X}) \times \mathcal{T}(\Sigma, \mathcal{X})$, so daß:

i. $sort(r) \leq_\Sigma sort(l)$

ii. $root(l) \in \mathcal{F}$.

iii. $\mathcal{V}(r) \subseteq \mathcal{V}(l)$.

Eine *Instanz* einer Regel $l \to r$ (eines Terms t) ist eine Regel $l' \to r'$ (ein Term t'), so daß $l' \equiv l\sigma, r' \equiv r\sigma$ ($t' \equiv t\sigma$) für irgendeine Umbenennung σ, wobei $\equiv$ *syntaktische* Gleichheit bezeichnet.

Ein Term t *leitet ab* an einer Position $p \in \mathcal{O}(t)$ zu einem Term t' unter Benutzung der Regel $l \to r$ und der *Matching-Substitution* σ (geschrieben $t \to_{l \to r, p, \sigma} t'$), gdw.

1. $l \to r$ ist eine Instanz einer Regel aus $\mathcal{R}$.
2. $\mathcal{V}(l) \cap \mathcal{V}(t) = \emptyset$
3. $t/p \equiv l\sigma$
4. $t' \equiv t[p \leftarrow r\sigma]$
5. σ ist die kleinste Substitution, die diese Bedingungen erfüllt.

Ein Term t, der nicht mehr zu einem anderen Term t' abgeleitet werden kann, ist in *Normalform*.

Bemerkungen:

- i.) stellt sicher, daß durch Ableitung aus wohlgetypten Terme wieder wohlgetypte Terme entstehen.
- In den geläufigeren Definitionen werden 2.) und 5.) meistens nicht gefordert. Es erleichtert jedoch einige Argumentationen und beseitigt Unklarheiten.
- Es existiert immer eine Instanz einer Regel aus $\mathcal{R}$, die 2.) erfüllt.
- Manchmal schreiben wir einfach $\to$ oder $\to_{\mathcal{R}}$, statt $\to_{l\to r,p,\sigma}$, wenn die Position, Regel oder Substitution ohne Bedeutung sind.
- Statt nur einem Ableitungsschritt kann man auch Folgen von Ableitungsschritten definieren: $\to_{\mathcal{R}}^*$ bezeichnet Null oder mehrere Ableitungsschritte[7]. $\to_{\mathcal{R}}^{\leq k}$ bezeichnet Null bis maximal k Ableitungsschritte, $\to_{\mathcal{R}}^n$ bezeichnet genau n Ableitungsschritte.
- Gilt $t \to_{\mathcal{R}}^* t'$ und ist t' in Normalform, dann heißt t' auch *Normalform von t*. Mit $\mathcal{N}(t)$ bezeichnet man die Menge aller Normalformen von t. Falls $\mathcal{R}$ terminierend ist, dann ist die Menge $\mathcal{N}(t)$ für jeden Term s endlich und berechenbar, da ein Term nur zu endlich vielen Termen angeleitet werden kann und jede Ableitung irgendwann terminiert.
- Man beachte, daß *Umbennenung* der Variablen in einer Regel keinen Einfluß auf die erzeugte Relation $\to_{\mathcal{R}}$ hat.
- $\to_{\mathcal{R}}$ ist abgeschloßen unter Substitutionen und Subtermersetzung, d. h. : Falls $s \to_{\mathcal{R}}^* t$, dann auch $s\sigma \to_{\mathcal{R}}^* t\sigma$ und $t'[p \leftarrow s] \to_{\mathcal{R}}^* t'[p \leftarrow t]$ $\forall_{t' \in \mathcal{T}(\Sigma,\mathcal{X}), p \in \mathcal{O}(t'), \sigma}$.
- $\to_{\mathcal{R}}^*$ is sorten-kompatibel.

Lemma 3.7: (Matchingsubstitution)
Sei $\mathcal{R}$ gegeben. Für jeden Ableitungsschritt $t \to_{l\to r,p,\sigma}$ gilt, daß $\mathcal{D}(\sigma) = \mathcal{V}(l)$.

[7] Und ist damit die reflexive/transitive Hülle der Relation $\to_{\mathcal{R}}$

Beweis: Da σ nur Variablen in l belegen muß, folgt aus der Minimalität von σ, daß $\mathcal{D}(\sigma) \subseteq \mathcal{V}(l)$.

Wäre $\mathcal{D}(\sigma) \neq \mathcal{V}(l)$, so gibt es ein $x \in \mathcal{V}(l)$, $x \notin \mathcal{D}(\sigma)$. Es gilt immer, daß $l\sigma \equiv t/p$. Sei dann p_l ein Vorkommen der Variable x in l, d. h. $l/p_l \equiv x$. Dann gilt $l\sigma/p_l \equiv t/(p.p_l) \equiv x$, im Widerspruch zu $\mathcal{V}(l) \cap \mathcal{V}(t) = \emptyset$.
□

Für TRS definiert man die folgenden Begriffe:

Definition 3.8: (Eigenschaften von TRS)
Ein TRS $\mathcal{R}$ heißt *terminierend*, falls es keine unendlichen Ableitungen der Form $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \ldots$ gibt. Es heißt *konfluent*, falls $\forall_{t,t_1,t_2} : t \to^*_{\mathcal{R}} t_1 \wedge t \to^*_{\mathcal{R}} t_2 \Rightarrow \exists t_3 : t_1 \to^*_{\mathcal{R}} t_3 \wedge t_2 \to^*_{\mathcal{R}} t_3$. Diese Eigenschaft ist gleichbedeutend mit der *Church-Rosser* Eigenschaft[8]. Im Unterschied dazu heißt ein TRS *lokal konfluent*, wenn $\forall_{t,t_1,t_2} : t \to_{\mathcal{R}} t_1 \wedge t \to_{\mathcal{R}} t_2 \Rightarrow \exists t_3 : t_1 \to^*_{\mathcal{R}} t_3 \wedge t_2 \to^*_{\mathcal{R}} t_3$. D.h. alle in einem Schritt auseinanderlaufenden Rechnungen sind wieder zusammenführbar. Offensichtlich ist ein konfluentes TRS auch lokal konfluent. Ein TRS heißt *kanonisch* (konvergent), falls es sowohl terminierend, als auch konfluent ist.

$\mathcal{R}$ heißt *eindeutig normalisierend*, falls jeder Term t eine eindeutige Normalform hat, die wir dann mit $t\downarrow$ bezeichnen.

Es heißt *constructor-basiert*, falls für jede linke Seite einer Regel ($l \equiv f(t_1, \ldots, t_n)$) die Terme $t_1, \ldots, t_n$ Constructorterme sind.

Ein TRS $\mathcal{R}$ heißt *variablen-erhaltend*, wenn $\forall_{l \to r \in \mathcal{R}} : \mathcal{V}(l) = \mathcal{V}(r)$, d.h. die linken und rechten Seiten der Regeln enthalten die gleichen Variablen. Falls $\forall_{l \to r \in \mathcal{R}} : l$ ein linearer Term ist, dann heißt $\mathcal{R}$ *links-linear*.

Ein Ersetzungsschritt $t \to_{l \to r, p, \sigma}$ heißt *innermost* (*outermost*), falls kein Ableitungsschritt an einer Position p', mit $p <_p p'$ ($p' <_p p$) möglich ist. Eine ganze Ableitung heißt *innermost* (*outermost*), falls jeder Schritt in ihr innermost (outermost) ist.

Lemma 3.9:
Ein kanonisches TRS $\mathcal{R}$ ist eindeutig normalisierend.

Beweis: Da $\mathcal{R}$ terminierend ist, gibt es keine unendlichen Ableitungen. Daher kommt jede Ableitungsfolge ausgehend von einem Term s zu einer Normalform. Für jeden Term s gibt es keine zwei verschiedenen Normalformen t_1, t_2, da sonst $s \to^*_{\mathcal{R}} t_1$, $s \to^*_{\mathcal{R}} t_2$, $\nexists t_3 : t_1 \to^*_{\mathcal{R}} t_3, t_1 \to^*_{\mathcal{R}} t_3$. Dies würde aber die Konfluenzeigenschaft verletzen.
□

[8]Sie garantiert, daß die Reihenfolge einer Rechnung ohne Bedeutung ist, da man von verschiedenen Teilergebnissen (t_1, t_2) immer wieder zu einem gemeinsamen Teilergebniss (t_3) kommen kann.

Beispiel 3.10: $\mathcal{R}$ für Addition auf $\mathbb{N}$

$$
\begin{array}{l}
\mathcal{R} = \{ \\
\quad 0 + X \to X, \\
\quad s(X) + Y \to s(X+Y) \\
\ \} \\
s(0) + 0 \to_{s(X)+Y \to s(X+Y),\, \epsilon,\, \{X \leftarrow 0, Y \leftarrow 0\}} s(0+0) \\
\to_{0+X \to X,\, 1,\, \{X \leftarrow 0\}} s(0)
\end{array}
$$

Wie Beispiel 3.10 zeigt, ist es oft lästig, die kompletten Regeln an einen Ableitungsschritt zu schreiben. Wir werden daher, wenn wir die Regel überhaupt angeben, *Namen* für Regeln vergeben und diese dann schon im TRS schreiben, z.B. $r_1 : 0 + X \to X$.

3.3 Gleichheit, Goals, Lösungen

Da Programmierung in logisch-funktionalen Sprachen darauf hinausläuft, Lösungen für Gleichungen zu finden, muß erst mal definiert werden, wann zwei Terme (semantisch) gleich sind:

Definition 3.11: (Semantische Gleichheit)
Gegeben seien ein eindeutig normalisierendes TRS $\mathcal{R}$ und Terme t, t'. t, t' heißen *semantisch gleich (unter $\mathcal{R}$)*, geschrieben $t =_{\mathcal{R}} t'$, gdw. $t\!\downarrow\, \equiv t'\!\downarrow$.

Beispiel 3.12: Gleichheit

$$
\begin{array}{l}
s(0) =_{\mathcal{R}} s(0) + 0 \\
s(X) =_{\mathcal{R}} s(0) + X \\
s(X) \neq_{\mathcal{R}} s(X) + 0
\end{array}
$$

Bemerkungen:

- Offensichtlich gilt $\equiv \subseteq =_{\mathcal{R}}$.
- $=_{\mathcal{R}}$ ist ebenfalls abgeschlossen unter Substitutionen und Subtermersetzung, d. h. falls $s =_{\mathcal{R}} t$, dann auch $s\sigma =_{\mathcal{R}} t\sigma$ und $t'[p \leftarrow s] =_{\mathcal{R}} t'[p \leftarrow t]$, $\forall_{t' \in \mathcal{T}(\Sigma,\mathcal{X}), p \in \mathcal{O}(t'), \sigma}$. Dies folgt direkt aus den gleichen Eigenschaften von $\to_{\mathcal{R}}$.

Da wir daran interessiert sind, Anfragen (Goals) zu lösen, definieren wir nun, was ein Goal und was eine Lösung ist.

Definition 3.13: (Goal)
Ein *Goal* ist ein Paar $(s,t) \in \mathcal{T}(\Sigma,\mathcal{X}) : s' \times \mathcal{T}(\Sigma,\mathcal{X}) : s', s' \in \mathcal{S}$, geschrieben $s =_? t$.

Definition 3.14: (Semantisches Unifikationsproblem)
Gegeben seien $\mathcal{T}(\Sigma,\mathcal{X})$ und ein TRS $\mathcal{R}$, sowie ein Goal $s =_? t$. Das Auffinden aller Substitutionen (*Lösungen*) σ , so daß $s\sigma =_{\mathcal{R}} t\sigma$, bezeichnet man als das *semantische Unifikationsproblem* (SUP).

Das *syntaktische Unifikationsproblem* ist ein, in dem semantischen Unifikationsproblem enthaltender, Spezialfall:

Definition 3.15: (Syntaktisches Unifikationsproblem)
Gegeben $\mathcal{T}(\Sigma,\mathcal{X})$ und zwei Terme $s,t \in \mathcal{T}(\Sigma,\mathcal{X})$. Das Auffinden einer Substitution σ, so daß $s\sigma \equiv t\sigma$ nennt man das *syntaktische Unifikationsprobelm* (SYUP). Ein σ mit $s\sigma \equiv t\sigma$ nennen wir *Unifikator*, weiter nennen wir es *allgemeinsten Unifikator* (mgu[9]), falls für alle anderen Unifikatoren θ gilt: $\exists\lambda : \theta = \sigma\lambda$. Gibt es ein σ mit $s\sigma \equiv t\sigma$, so nennt man s,t *unifizierbar*.

Beispiel 3.16: SUP

$$X + Y =_? s(0)$$
$$X + X =_? X + Y$$

Nimmt man das TRS aus Beispiel 3.10, so hat das erste Goal in Beispiel 3.16 die Lösungen $\sigma_1 = \{X \leftarrow 0, Y \leftarrow s(0)\}$ und $\sigma_2 = \{X \leftarrow s(0), Y \leftarrow 0\}$.

3.4 Entscheidbarkeiten, Berechenbarkeiten

Bisher wurde noch (fast) kein Wort darüber verloren, ob (und wie) verschiedene Eigenschaften entscheidbar sind.
Dieser Abschnitt holt das jetzt nach, vermeidet aber Beweise, wo sie leicht in der Literatur gefunden werden können und begnügt sich dann mit einem Verweis.
Wir beginnen mit den positiven Resultaten, dazu vorweg die

Definition 3.17: (Matchingproblem)
Für zwei Terme l,t, mit $\mathcal{V}(l) \cap \mathcal{V}(t) = \emptyset$ ist das *Matchingproblem* (MP) gegeben durch: 'Gibt es eine Substitution σ, so daß $l\sigma \equiv t$?'

[9]Für 'most general unifier'.

I. Das Matchingproblem MP ist entscheidbar. Darüber hinaus gilt:

   - Es existiert genau eine minimale Matchingsubstitution σ.
   - Sie ist aus l, t berechenbar.

   Vgl. dazu [Ave95, Lemma 3.4.2].

II. Es ist entscheidbar, ob für ein gegebenes TRS $\mathcal{R}$ und einen Term t, $t \rightarrow_{\mathcal{R}} t'$ für irgendeinen Term t' gilt. Das Grund hierfür ist die *Endlichkeit* von $\mathcal{R}$. An jeder Position $p \in \mathcal{O}(t)$, von denen es nur endlich viele gibt, und jede der (endlich vielen) Regeln $l \rightarrow r \in \mathcal{R}$ probiert man, ob sich das Matchingproblem für $l, t/p$ lösen läßt. Wenn ja, kann man t' berechnen als $t' \equiv t[p \leftarrow r\sigma]$.

    Ist $\mathcal{R}$ unendlich, ist es i.A. nicht entscheidbar, ob $t \rightarrow_{\mathcal{R}} t'$ gilt, siehe dazu auch [Ohl94, S. 22].

III. Aus II folgt sofort: Es ist entscheidbar, ob ein Term t in Normalform ist. Denn das ist gerade dann der Fall, wenn *keine* Regel anwendbar ist.

IV. Das syntaktische Unifikationsproblem (SYUP) ist entscheidbar. Es gilt sogar:

    - Wenn ein Unifikator existiert, dann existiert immer auch ein allgemeinster Unifikator, der aus s, t berechenbar ist.
    - Allgemeinste Unifikatoren sind eindeutig, bis auf Variablenumbennenung, d.h. sind σ, λ allgemeinste Unifikatoren, so existsiert immer eine Umbennenung θ, so daß $\sigma = \lambda\theta$. D.h., daß man immer einen mgu als *den* mgu wählen kann.

    Für einen Beweis siehe [MM82].

Nach den erfreulichen Ergebnissen folgen nun die weniger erfreulichen. Aber zuvor noch die

Definition 3.18: (Kritisches Paar)
Sei $\mathcal{R}$ ein TRS und $l_1 \rightarrow r_1$, $l_2 \rightarrow r_2$, Varianten von Regeln aus $\mathcal{R}$, so daß $\mathcal{V}(l_1) \cap \mathcal{V}(l_2) = \emptyset$. Wenn es ein $p \in \bar{\mathcal{O}}(l_1)$ gibt, so daß l_1/p und l_2 unifizierbar sind mit dem mgu σ, dann heißt $(l_1[p \leftarrow r_2]\sigma, r_1\sigma)$ ein *kritisches Paar*, es sei denn, die beiden Regeln sind Varianten derselben Regel aus $\mathcal{R}$ und $p = \epsilon$. Die Menge aller kritischen Paare bezeichnen wir mit $\mathcal{CP}(\mathcal{R})$.

Kritische Paare formulieren das Problem konkurrierender Regelanwendung und werden benötigt für das wichtige

Lemma 3.19: (Knuth-Bendix)
Sei $\mathcal{R}$ ein TRS. $\mathcal{R}$ ist lokal konfluent, gdw. alle kritischen Paare $(s, t) \in \mathcal{CP}(\mathcal{R})$ zusammenführbar sind, d.h. $\exists t' : s \rightarrow^*_{\mathcal{R}} t'$, $t \rightarrow^*_{\mathcal{R}} t'$.

Beweis: Siehe [Klo92, Lemma 2.4.11].
□

Ebenso interessant ist folgendes

Lemma 3.20: (Newman)
Sei $\mathcal{R}$ ein terminierendes TRS. Dann gilt: $\mathcal{R}$ ist konfluent, gdw. es lokal konfluent ist.

Beweis: Siehe [Ohl94, Lemma 2.1.12],[Ave95, Satz 1.2.9].
□

V. Für irgendein TRS $\mathcal{R}$ ist nicht entscheidbar, ob es terminiert. Denn man kann jede Turing-Maschine als ein TRS darstellen. Aus der Unentscheidbarkeit des Halteproblems einer Turing-Maschine folgt dann sofort die Unentscheidbarkeit der Terminerung eines TRS. Den Beweis hierzu kann man in [Dau92] finden.

VI. Es ist nicht entscheidbar, ob ein TRS konfluent ist. Beweis: Siehe [Ave95, Satz 2.4.7].

VII. Es ist nicht entscheidbar, ob ein TRS lokal konfluent ist. Denn nach Lemma 3.19 ist dies äquivalent zu der Entscheidbarkeit, ob zwei Terme zusammenführbar sind, was für nicht terminierende TRS ebenfalls nicht entscheidbar ist.

Erfreulicherweise gilt aber noch folgende Feststellung:

VIII. Falls ein TRS terminierend ist, dann ist lokale Konfluenz entscheidbar.

Beweis: Die Menge $\mathcal{CP}(\mathcal{R})$ ist endlich, da $\mathcal{R}$ endlich ist. Es gibt also nur endlich viele Paare $(t, t') \in \mathcal{CP}(\mathcal{R})$. Nach Lemma 3.19 ist $\mathcal{R}$ lokal konfluent, gdw. $\forall_{(t,t') \in \mathcal{CP}(\mathcal{R})} : t, t'$ zusammenführbar sind. Da $\mathcal{R}$ nun terminierend ist, ist $\mathcal{N}(s)$ für jeden Term s endlich und berechenbar. Um zu überprüfen, ob zwei Terme zusammenführbar sind, muß man nur noch prüfen, ob $\mathcal{N}(t) \cap \mathcal{N}(t') \neq \emptyset$ ist. Dies kann man für alle (endlich vielen) $(t, t') \in \mathcal{CP}(\mathcal{R})$ tun.
□

IX. Darüber hinaus gilt für ein terminierendes TRS nach Lemma 3.20: Es ist entscheidbar, ob das TRS konfluent ist.

Als schlechten Abschluß die entscheidende Feststellung:

X. Das semantische Unifikationsproblem ist nicht entscheidbar. D.h. es gibt kein Verfahren, das immer stoppt und alle Lösungen eines Goals $s =_? t$ liefert.

Beweis: Wir zeigen, daß das 10te Hilbert'sche Problem (Lösen diophantischer Gleichungen [Hil00, Hil02]) auf ein semantisches Unifikationsproblem zurückgeführt werden kann. Aus der Unentscheidbarkeit des 10ten Hilbert'schen Problems folgt dann sofort die Unentscheidbarkeit von SUP. Ein Beweis für die Unentscheidbarkeit des Problems

Beispiel 3.21: Nullstellensuche

$$
\begin{array}{ll}
\mathcal{R} = \{ & \\
\quad pn(0,X) & \to X \\
\quad pn(s(X),Y) & \to s(pn(X,Y)) \\
\quad mn(0,X) & \to 0 \\
\quad mn(s(X),Y) & \to pn(mn(X,Y),Y) \\
\quad (S,X)+(S,Y) & \to (S,pn(X,Y)) \\
\quad (p,0)+(m,X) & \to (m,X) \\
\quad (p,s(X))+(m,0) & \to (p,s(X)) \\
\quad (p,s(X))+(m,s(Y)) & \to (p,X)+(m,Y) \\
\quad (m,0)+(p,X) & \to (p,X) \\
\quad (m,s(X))+(p,0) & \to (m,s(X)) \\
\quad (m,s(X))+(p,s(Y)) & \to (m,X)+(p,Y) \\
\quad (S,X)*(S,Y) & \to (p,mn(X,Y)) \\
\quad (p,X)*(m,Y) & \to (m,mn(X,Y)) \\
\quad (m,X)*(p,Y) & \to (m,mn(X,Y)) \\
\quad \} &
\end{array}
$$

ist z.B. in [HI73, JM91] zu finden. Siehe auch [DMR76] für eine Zusammenfassung der Auswirkungen der Unentscheidbarkeit dieses Problems.

Man betrachte das TRS in Beispiel 3.21. Jede Zahl $n \in \mathbb{N}$ läßt sich eineindeutig darstellen als $s^n(0)$ Man kann leicht sehen (und beweisen), daß jede ganze Zahl z sich darstellen läßt, als $(v, s^n(0))$, mit $v = \begin{cases} p, & \text{falls } z \geq 0 \\ m, & \text{falls } z < 0 \end{cases}$ und $n = |z|$. Ebenso läßt sich der Grundterm $(v, s^n(0))$, $n \geq 0$ der ganzen Zahl $z = \begin{cases} n, & \text{falls } v = p \\ -n, & \text{falls } v = m \end{cases}$ zuordnen. Jeder Grundterm läßt sich somit einer Zahl zuordnen und jede Zahl läßt sich einem Grundterm zuordnen.

Man kann sehen, daß die Reduktion von $pn(s,t)$, $s,t \in \mathcal{T}(\Sigma,\emptyset)$ gerade den Grundterm liefert, der $n+n'$ entspricht, wobei n,n' die natürlichen Zahlen sind, die durch s,t repräsentiert werden. Gleiches gilt für mn und die Multiplikation auf natürlichen Zahlen, sowie $+,*$ und Addition, Multiplikation auf ganzen Zahlen.

Eine diophantische Gleichung über Variablen $x_1,\ldots,x_l$, $\sum\limits_{1\leq j\leq l}\sum\limits_{0\leq i\leq m} k_{i,j}x_j^i = 0$ läßt sich als ein Goal $G =_? 0$ darstellen, wobei jeder Summand $S_{i,j}$ die Form $S_{i,j} \equiv k'_{i,j} * \overbrace{X_j * \cdots * X_j}^{i \text{ mal}}$ hat. Damit kann man dann schreiben: $G \equiv S_{0,1} + \cdots + S_{m,1} + S_{0,2} + \cdots + \cdots + S_{m,l}$. Bei dieser Darstellung sind X_j Variablen, $k'_{i,j}$ die Darstellung von $k_{i,j}$.

Jede Lösung dieses Goals $\sigma = \{X_j \leftarrow t'_j | 1 \leq j \leq l\}$, $t'_j \in \mathcal{T}(\Sigma, \emptyset)$ korrespondiert zu einer Lösung $\{x_j = t_j | 1 \leq j \leq l\}$ der diophantischen Gleichung, wobei t_j die ganze Zahl ist, die zu t'_j korrespondiert.

Wäre SUP entscheidbar, gäbe es ein Verfahren, das immer stoppt und alle Lösungen liefert. Dann könnten wir aber auch das 10te Hilbert'sche Problem entscheiden, indem wir einfach das Goal, das die fragliche Gleichung repräsentiert, von dem Verfahren bearbeiten lassen. Daher kann es kein solches Verfahren geben.

□

Siehe auch [WZ95] für andere TRSe, die $*, +$ auf den ganzen Zahlen implementieren.

Kapitel 4

Definition des Systems

In diesem Kapitel werden die wichtigsten Komponenten von Quetzal definiert:

1. Die Beschreibungssprache $\mathcal{QDL}$.
2. Die Aktionen und ihre Semantik.

Die abstrakte Maschine und die Übersetzung von $\mathcal{QIR} \Rightarrow \mathcal{QAM}$-Code werden in Kapitel 5 behandelt.
Die Definition des Systemaufbaus und des Zusammenspiels der einzelnen Komponenten geschieht in Kapitel 8, dort werden auch die einzelnen Frontends behandelt.
Der Rest dieses Kapitels ist folgendermaßen aufgebaut:

- 4.1 besteht aus einer Einführung. Dort werden die Konzepte und Ideen von Quetzal anhand von Beispielen eingeführt.
- In 4.2 wird die Grammatik der $\mathcal{QDL}$ und der Aktionen gegeben.
- Danach wird in 4.3 die Umformung eines $\mathcal{QDL}$ Programmes in die abstrakte Darstellung ($\mathcal{QIR}$) angegeben. Die Semantikdefinitionen arbeiten ausschließlich auf dieser Darstellung.
- Als nächstes werden einige Konventionen und Notationen eingeführt, welche in der Semantikdefinition benutzt werden (4.4).
- Darauf folgt in 4.5 die Definition der statischen Semantik.
- In 4.7 schließlich wird die dynamische Semantik des Systems unter den Aktionen gegeben.

Zur Definition von $\mathcal{QDL}$ wird eine kontextfreie Grammatik verwandt, die durch Zusatzbedingungen ergänzt wird.
Die Definition der statischen und dynamischen Semantik bedient sich eines Satzes von Inferenzregeln und ist an [MTH90, MT90] angelehnt. Die Semantik wird ergänzt durch einige Lemmata, die Aussagen über ihre Konsistenz beinhalten.

4.1 Die Grundzüge

Quetzal basiert auf der Idee, daß sich verschiedenste semantische Unifikationsverfahren einfach als Manipulation einer Liste von Goals auffassen lassen. Dabei wird das erfolgreiche Lösen eines Goals als eine Folge von Transformationsschritten aufgefaßt, welche bei der Liste, die nur dieses Goal enthält, beginnt und bei der leeren Liste endet. Damit kann jedoch nur erkannt werden, daß es eine Lösung gibt, wie diese Lösung aber nun aussieht muß noch explizit angegeben werden.
Dies geschieht dadurch, daß in jedem Transformationsschritt Daten mitberechnet werden können, die zum nächsten Transformationsschritt weitergereicht werden. Beim Erreichen der leeren Liste muß dann noch aus den vorliegenden Daten die Lösung ausgerechnet werden.
Jede Transformation beginnt damit, daß aus der aktuellen Goalliste ein Goal herausgenommen wird, welches zum *aktuellen Goal* wird. Diese Auswahl wird explizit als eine Funktion angegeben, die aus der aktuellen Goalliste und den Daten der Transformation das neue Goal und eine restliche Goalliste berechnet. Es wird dann versucht, das aktuelle Goal zu einer Goalliste zu transformieren, die wieder mit der Restliste vereinigt wird.
Im allgemeinen sind jedoch gleichzeitig mehrere Transformationsschritte möglich. Zum Erhalt der Vollständigkeit muß natürlich jeder Transformationsmöglichkeit nachgegangen werden. Um diesen Nichtdeterminismus zu implementieren, wird er sequentialisiert: Es wird eine Ablaufsteuerung angegeben, die genau festlegt, welcher Möglichkeit zuerst nachgegangen werden soll. Nun beschränkt sich dies aber nicht nur auf eine Anordnung von Möglichkeiten, sondern es ist möglich, Bedingungen in diese Ablaufsteuerung einzuarbeiten. Es ist etwa möglich zu sagen: "Wenn diese Möglichkeit nicht funktioniert, dann nimm diese, sonst nicht". Auch eine einfache Art der Iteration ist eingebaut. Die Ablaufsteuerung hat Zugriff auf die Daten eines Transformationsschrittes und kann sie in ihre Entscheidungen einbeziehen.
In Quetzal wird ein einzelner Transformationsschritt als *Metaregel* bezeichnet. Eine Metaregel kann, neben dem aktuellen Goal, auch noch andere Daten von der Ablaufsteuerung erhalten. Beim Probieren einer Metaregel werden erst die Parameter der Regeldefinition gegen die aktuellen Argumente gematcht. Ist dies nicht möglich, schlägt die Metaregel fehl und die Ablaufsteuerung fährt mit der nächsten Anweisung fort. Andernfalls wird das aktuelle Goal gegen das *Goal-Pattern* in der Metaregel gematcht. Ist dies erfolgreich, muß noch die *Bedingung der Metaregel* zu 'wahr' auswerten. Erst danach wird das Ergebnis aus der Metaregel berechnet und die Veränderung der Daten, die ebenfalls durch die Metaregel notiert wird, ausgeführt. Danach wird die *komplette* Ablaufsteuerung mit der neuen Goalliste wiederum ausgeführt.
Jede Strategie definiert außerdem, was beim Auftreten bestimmter Ereignisse als Ergebnis oder Protokoll auszugeben ist. Dies wird einfach durch Terme annotiert, die mit einem Ereignis gekoppelt werden. Ereignisse können hierbei sein: 'solved' für eine leere Goalliste, also eine Lösung. 'dead' für das Erreichen einer Sackgasse. Eine Sackgasse ist dann erreicht, wenn die komplette Ablaufsteuerung abgearbeitet wurde, aber keine Metaregel erfolgreich angewandt werden konnte. 'bound' steht für das Überschreiten der maximalen Suchtiefe.

Ein wesentlicher Bestandteil von Quetzal ist, daß man Terme ausrechnen kann und auch eigene Funktionen angeben kann. Damit ist sichergestellt, daß sich viele Strategien implementieren lassen. Ein weiterer Vorteil ist, daß Quetzal beim Ausrechnen von Funktionsausdrücken (größtenteils) eine *needed-evaluation* Strategie benutzt, d.h. Ausdrücke werden nur dann ausgerechnet, wenn sie benötigt werden und auch dann nur einmal, d.h. Unterausdrücke werden geteilt. Einige Konstrukte und Bedingungen zwingen allerdings dazu, daß bestimmte Ausdrücke in Metaregeln komplett ausgewertet werden müssen, um die Konsistenz der Daten zu bewahren. Näheres dazu findet sich später in der Definition der dynamischen Semantik.
Natürlich wäre die Möglichkeit eigener Funktionsdefinitionen mangelhaft, wenn man nicht auch eigene Datentypen definieren könnte. Quetzal bietet diese Möglichkeit ebenso. Es können neue Konstruktoren eingeführt werden, die dann auch in Funktions-/Strategiedefinitionen auftauchen können. Selbstverständlich sind schon einige primitive Datentypen eingebaut: Natürliche Zahlen, Zeichen, Listen, Tupel, Bool'sche Werte, Substitutionen.
Um semantische Korrektheit zu bewahren, müssen alle Definitionen wohlgetypt sein. Das verwendete Typsystem ist im wesentlichen das von Milner[Mil78]. Da der Typcheck zur Compilezeit stattfindet und aus Effizienzgründen auf einen Runtime-typecheck verzichtet wird, muß man bei der Implementierung von Strategien mehr Verantwortung übernehmen: Denn es ist nicht möglich, zur Compilezeit zu überprüfen, ob z.B. die Manipulation eines Goals in einer Metaregel aus wohlgeformten Goals[1] wieder wohlgeformte Goals baut, da die Signatur, über der diese Goals Terme darstellen, noch nicht bekannt ist und erst später vom Benutzer gegeben wird (s.u.).
Neben der Möglichkeit, Datentypen, Funktionen und Strategien zu definieren tritt noch unabhängig die Verwendung von Quetzal, auch alle Objekte zu definieren, die man zum tatsächlichen Lösen eines Goals braucht: Signaturen, TRSs, Präzedenzen, Goals, Regeln, Taktiken. Signaturen, Regeln, TRSs und Goals werden in einer natürlichen Schreibweise annotiert. In Quetzal hat jedes dieser Objekte einen einzigartigen Namen und wird auch über ihn manipuliert. Präzedenzen dienen zur Angabe einer Ordnung über den Operatorsymbolen einer Signatur, die für einen (ebenfalls im System eingebauten) Terminierungscheck eines TRS verwendet wird. Taktiken dienen dazu, Regeln in einem TRS mit Prioritäten und Attributen auszustatten. Sie annotieren eine Reihenfolge unter Regeln, die bei der Bildung des aktuellen TRS aus einem TRS berücksichtigt wird: In dem TRS, das als *Liste* von Regeln angesehen wird, werden die Regeln anhand der Vorgaben der Strategie geordnet. Gleichzeitig kann eine Regel durch eine Taktik mit einem oder mehreren *Attributen* versehen werden. Attribute sind Bool'sch: Eine Regel hat ein Attribut, oder sie hat es nicht. Weitere Information liegt nicht vor.
In Daten-/Funktions- und Strategiedefinitionen können Terme über einer Signatur als Argumente verwandt werden. Da aber bei der Definition dieser Objekte die Signatur der Terme i.A. noch nicht bekannt ist, können diese Terme nur über generelle Pattern eingeführt werden. Pattern und Terme mit bestimmten Operatorsymbolen können nicht gebaut wer-

[1] D.h. solchen, die unter einer gegebenen Signatur wohlgetypt sind.

den. Gleichwohl stehen eine Reihe von vordefinierten Funktionen zur Manipulation dieser Objekte zur Verfügung. Weiter unten wird dies anhand eines Beispielprogrammes näher erläutert. In Quetzal werden solche Signaturterme durch den eingebauten Typ `term` repräsentiert.

Obwohl Namen in Daten-/Funktions-/Strategiedefinitionen einerseits und den anderen Definitionen andererseits, einzigartig sein müssen gilt dies nicht für das Ganze: So kann es z.B. eine Funktion und ein TRS mit demselben Namen geben.

4.1.1 Ein Beispiel

Anhand der folgenden Programmbeispiele sollen die oben angesprochenen Konzepte verdeutlicht werden:

```
sorts
    foo<reply;  response;  reply<response;
.

cons
    a: foo;   b: foo;   fail: response;   r: foo foo -> reply;
.

funcs
    f: foo -> response;   g: foo foo -> foo;
.

rules
    r1: g(X,X) -> X;   r2: g(a,b) -> b;   r3: g(b,a) -> a;
    r4: f(a) -> fail;   r5: f(b) -> r(g(a,b),b);
.

trs
    t1: r1 r2 r3 r4 r5;
    t2: r2 r4;
.

tactics
    tac1: r1>r3>@, r1 _RWOnly;  tac2: r1>r5;
.

goals
    g1: g(X,Y) =? a;
    g2: f(X) =? b;
.
```

Das Beispiel besteht aus 8 Definitionsblöcken für 20 Objekte; jeder Definitionsblock beginnt mit einem Schlüsselwort und endet mit einem Punkt. Ein Definitionblock kann mehrere Objekte definieren, deren Definitionen durch '_;_' getrennt werden.

Die ersten drei Definitionsblöcke definieren eine Signatur. Es werden nacheinander Sorten, Constructoren und Functoren definiert. Die Ordnungsrelation $<_\Sigma$ wird durch eine Reihe von Angaben der Form `s < s'` gegeben.
Die `rules` Definitionen führen 5 Regeln `r1`–`r5` ein, wobei Variablen in den linken/rechten Seiten dadurch gekennzeichnet werden, daß sie großgeschrieben werden (`X, Y`). TRSe werden durch das Aufzählen der Regeln, die sie beinhalten sollen, definiert.
Taktiken werden definiert, indem zum einen eine Reihenfolge festgelegt wird, zum anderen Attributnamen an Regelnamen gebunden werden. Die Taktik `tac1` legt fest, daß Regel `r1` vor Regel `r3` auftauchen soll. Das Zeichen '`@`' ein Platzhalter und steht für die restlichen Regeln des aktuellen TRS, d.h. beim Berechnen des TRS zum Lösen eines Goals werden für '`@`' alle Regeln des aktuellen TRS, in der Reihenfolge, wie sie dort auftauchen, eingesetzt, die in der Taktik nicht explizit auftauchen. Wäre das aktuelle TRS z.B. `t1`, dann würde mit `tac1` folgendes TRS entstehen: `[ r1, r3, r2, r4, r5]`, mit `tac2` das TRS `[r1, r5, r2, r3, r4]`, da ein implizites '`@`' an die Taktik angehängt wird, wenn sie keins enthält.
Zum Abschluß werden noch zwei Goals definiert.
Das nächste Beispiel definiert eine Strategie, zusammen mit einigen Datentypen und Funktionen.

```
data
   fail: result;   ok:substitution --> result;
.

functions
 equal: (s,t) = fail, if res ==%
               where res=unify(s,t);;
        (s,t) = ok(s), if res!=%
              where res=unify(s,t);;
        ;
 chop:  #x = x, if argsnum(x)==0;
        #x = x/[1], if argsnum(x)>0;
        ;
 selfunc: x:xs = (x,xs);
          ;
.

strategy bar
 data local res = fail;
       global foo =0;
 select selfunc();
  mr1();
 else
  mr2();
output
   dead: "Sackgasse";
   bound: "Zu tief, total: "++show(foo);
   solved: "Geloest: "++show(res);
rules
   mr1(): x=?y --> [], if result!=fail;
```

```
      res:=result;
      foo:=foo+1;
      output "Geloest.....";
      where result=equal(x,y);
    ;
  mr2(): #x=?#y -->[x1 =? y1], if (x1!=x) & (y1!=y);
      res:= ok({});
      foo:=foo+1;
      where x1=chop(x); y1=chop(y);
    ;
.
```

Dieses Beispiel besteht aus drei Definitionsblöcken. Zuerst werden zwei Datenconstructoren definiert: `fail` und `ok`. `fail` ist eine Konstante des neuen Typs `result`. `ok` hat ein Argument vom Typ `substitution`, dem eingebauten Substitutionstyp.
Es folgenden die Definitionen von drei Funktionen: `equal, chop, selfunc`. Die Definition von Funktionen erfolgt durch Pattern; es werden alle Pattern der Argumente aufgezählt und das zugehörige Ergebnis geschrieben, begleitet von einem optionalen Guard. So sagt das (einzige) Pattern für die Funktion `selfunc`, daß, wenn das Argument eine Liste mit Kopfelement `x` und Restliste `xs` ist, dann ist das Ergebnis das Paar `(x,xs)`.
Funktionsdefinitionen müssen immer 'geblockt' werden, d.h. alle Fälle für die Funktion müssen zusammen angegeben werden und dürfen nicht in seperaten Definitionen erfolgen. Die Definition von `equal` zeigt, daß durch `data` Definitionen eingeführte Constructoren frei verwendet werden dürfen[2].
Das Pattern '`#x`' in der Definition von `chop` hat eine besondere Bedeutung. Wie erwähnt, können Funktionen auch auf Termen über einer Signatur arbeiten, etwa einem Goal. Zur Bearbeitung dieser Terme muß man oft unterscheiden, ob sie aus einer Variablen[3] oder aus einem Operatorterm bestehen. Dazu dient das Pattern '`#x`': Es paßt nur auf Signaturterme, die keine Variable sind, also ein Operatorsymbol an der Wurzel haben. Ebenso gibt es ein Pattern, daß nur auf Variablen paßt, z.B. `X`, also ein großgeschriebener Bezeichner.
Die in `chop` benutzten Funktionen `argsnum, /` sind eingebaute Funktionen, vom Typ `term -> nat, (term, [nat]) -> term`, die die Anzahl von Argumenten bzw. den Subterm an einer bestimmten Stelle eines Signaturterms zurückgeben. Der Typ `term` ist der eingebaute Typ von Signaturtermen.
Als letztes wird eine Strategie definiert. Die Definition wird vom Namen der neuen Strategie eingeleitet: `bar`. Da eine Strategie beim Lösen Daten mitberechnen muß, wird mit den folgenden Definitionen deklariert, welche Daten diese Strategie benötigt. Dabei wird zwischen *globalen* und *lokalen* Daten unterschieden. Lokale Daten werden beim Lösungsprozeß zurückgesetzt, wenn ein Suchzweig abgearbeitet wurde. Ihre Veränderung wirkt sich also nur auf den von diesem Knoten ausgeheneden Suchbaum aus, nicht auf den Rest der Suche. Im Gegensatz dazu bleiben Veränderungen der globalen Daten sichtbar.

[2] '`%`' steht dabei für den eingebauten Constructor, der eine fehlgeschlagene Substitution repräsentiert, also das eingebaute Äquivalent zu dem benutzerdefiniertem '`fail`'.
[3] Einer Variablen über der Signatur, *nicht* einer Variablen, wie `s` in der Def. von `equal`!!

Diese Strategie definiert ein lokales Datum mit Namen `res`. Es folgt der Initialwert dieses Datums, also der Wert, der ihm beim Anfang der Suche zugewiesen wird. In diesem Falle ist dies der oben eingeführte Datenconstructor `fail`. Man beachte, daß die Initialisierung den Typ des Datums festlegt. Die Strategie hat auch ein lokales Datum: `foo`, das die Anzahl gerechneter Knoten enthalten soll.
Als nächstes wird angegeben, wie ein Goal aus der aktuellen Goalliste auszuwählen ist. Dies geschieht mittels der Angabe eines Ausdruckes, der den Typ `[goal] -> (goal,[goal])` haben muß. D.h. in diesem Falle, daß aktuelle Goal (und die neue Restliste) ergeben sich durch die Auswertung des Ausdruckes '`selfunc (`*goals*`)`', wobei *goals* die aktuelle Goalliste repräsentiert[4].
Als nächstes folgt die *Ablaufsteuerung*. In diesem Beispiel besteht sie aus folgendem: `mr1();` besagt, daß die Metaregel `mr1` versucht werden soll. Durch das `else` wird angegeben, daß beim Fehlschlag von `mr1` die nachfolgende Anweisung versucht werden soll, also die Metaregel `mr2`.
Nun folgt die Definition der Ausgaben, die beim Auftreten gewisser Ereignisse erfolgen sollen. Die Funktion `show` wandelt einen beliebigen Term[5] in seine Darstellung als Zeichenkette um.
Als letztes folgen nun die Definitionen der Metaregeln. Als erstes wird die Metaregel `mr1` definiert. Dies geschieht wieder über Pattern. Falls das aktuelle Goal auf das Muster `x=?y` paßt, dann wird die Bedingung `result!=fail` ausgewertet. Ergibt dies `True`, dann wird das Ergebnis `[]`, also die leere Liste zurückgegeben. Danach werden die Daten `res, foo` verändert[6] und der Suchbaum mit erneuter Abarbeitung der Ablaufsteuerung fortgesetzt[7]. Diese Regel enthält auch noch eine Ausgabeanweisung. Der Term dort wird in das *Protokoll* des Suchprozesses aufgenommen.
Die andere Metaregel `mr2` besteht aus ähnlichen Komponenten. Ihr Zweck ist die Verkleinerung der Terme `x, y`. Beachte auch hier die Verwendung des Patterns `#`.

4.2 Grammatik

Zuerst werden nun, nachdem einige Konventionen eingeführt wurden, die Grammatik und Zusatzbedingungen der $\mathcal{QDL}$ gegeben. Danach werden die Aktionen definiert.
Die Übersetzung von $\mathcal{QDL}$ nach $\mathcal{QIR}$ wird im nächsten Abschnitt gegeben. Die Semantikdefinitionen in den folgenden Abschnitten bedienen sich ausschließlich der $\mathcal{QIR}$, da sie ein wesentlich genaueres Beschreibungsmittel darstellt, als die $\mathcal{QDL}$, welche auf der anderen Seite für den menschlichen Benutzer einfacher zu lesen sein dürfte.

[4] Anmerkung: Der Operator '`=?`' besitzt den Typ `(term, term)->goal`.
[5] Genauer: Seine Normalform
[6] Man beachte, daß die *alten* Werte der Daten auf den rechten Seiten auftauchen dürfen!.
[7] Wobei die neue Goalliste aus dem Ergebnis, konkateniert mit der alten Restliste besteht.

4.2.1 Konventionen

Die Grammatik folgt der üblichen Schreibweise, wobei folgende Operatoren verwendet werden:

- Terminalsymbole werden unterstrichen gesetzt: `Terminal`
- Nichtterminale werden kursiv gesetzt: *Nichtterminal*
- Auswahl zwischen zwei Alternativen: A | B.
- Auswahl zwischen einer oder beiden Komponenten: A || B steht für eine der Sequenzen {A, B, AB}.
- Optionales Auftauchen: [A]
- Null oder mehrmaliges Auftauchen: $\{A\}^*$
- Schließlich steht $\{A\}^+$ für $A\{A\}^*$
- Kommentare zu einzelnen Produktionen: %Ein Kommentar

Die Operatoren | und || sind rechts–assoziativ. Um Klammern zu sparen, wird vereinbart, daß Sequenzierung die höchste Priorität hat und || eine höhere als | .

4.2.2 Die Grammatik

QDL-Prog ::= { *ObjDef* }$^+$

ObjDef ::= *SortDef* | *FunctorDef* | *ConsDef* | *PrecDef* | *TRSDef* |
TacticDef | *RuleDef* | *GoalDef* | *DataDef* | *FunctionDef* |
StrategyDef

SortDef ::= `sorts` { *Name* [`<` *Name*] `;` }$^+$ `.`

FunctorDef ::= `funcs` { *Name* `:` [{ *Name* }* `->`] *Name* `;` }$^+$ `.`

ConsDef ::= `cons` { *Name* `:` [{ *Name* }* `->`] *Name* `;` }$^+$ `.`

PrecDef ::= `precs` { *Name* `:` *PrecChain* {`,` *PrecChain* }* `;` }$^+$ `.`

PrecChain ::= *Name* {`<` *Name* }$^+$

TRSDef ::= `trs` { *Name* `:` { *Name* }$^+$ `;` }$^+$ `.`

TacticDef ::= `tactics` { *Name* `:` { *TacticChain* || {`,` *TacticAttributes* } } `;` }$^+$ `.`

TacticChain ::= { *Name* | `@` } {`>`{ *Name* | `@` } }$^+$

TacticAttributes ::= { *Name* *Name* } {`,` *Name* *Name* }*

RuleDef ::= `rules` { *Name* `:` *OpTerm* `->` *OpTerm* `;` }$^+$ `.`

```
GoalDef        ::= goals { Name: OpTerm =? OpTerm ; }+ .

DataDef        ::= data { Name :{Name | {Type -->Name }  } ; }+ .

Type           ::= Name |                                                  %Typname
                   [Type ] |                                 %Liste über einem Typ
                   (Type{,Type }+ ) |                                    %Tupeltyp
                   Type ->Type |                                    %Funktionstyp
                   VarName |                                        %Typvariable
                   (Type )                                          %Klammerung

FunctionDef    ::= functions
                   { Name: { {Pattern | () } = Term[, if Term]
                                                  [where LocalDefs] ;
                           }+
                   }+ .

StrategyDef    ::= strategy Name
                   [data {LocalData || GlobalData } ]
                   select Term ;
                   {Control }+
                   [output {Name: Term; }+ ]
                   [where LocalDefs]
                   rules {MetaRule; }+
                   .

MetaRule       ::= Name([Pattern {, Pattern }* ] ): Pattern --> Term [, if Term] ;
                                                    {Name := Term ; }*
                                                    [output Term;]
                                                    [where LocalDefs]

LocalData      ::= local {Name = Term; }+           % Lokale Daten und Initialwerte

GlobalData     ::= global {Name = Term; }+          % Global Daten und Initialwerte

Control        ::= break; |                                             % Abbruch
                   Name([Term{, Term }* ] ); |             % Aufruf einer Metaregel
                   Control else Control |                               % Auswahl
                   {{ {Control }* } |                % Block, inkl. leerem Statement
                   for Pattern in Term {{ Control }* } |           % Listeniteration
                   if ( Term ) {{Control }* } |                % Bedingte Abarbeitung
                   Control => {{Name := Term; }+ }                 % Datenzuweisung

LocalDefs      ::= { Name = Term; }+                                 % Nur Abkürzung

OpTerm         ::= Name[( OpTerm {, OpTerm }* )]  |
                   VarName
```

```
Pattern   ::= Number |                                    % Natürliche Zahlen
              Char |                                      % Zeichen
              String |                                    % Zeichenketten
              True | False |                              %Bool'sche Werte
              [[Pattern {, Pattern }* ] ] |               % Listen
              Pattern : Pattern |                         % Listen
              ( Pattern {, Pattern }+  ) |                % Tupel
              Name |                                      %Variable
              Name([ Pattern {, Pattern }* ] ) |          % Datenconstructoren
              VarName |                                   %Signaturtermvariablen
              #Name |                                     %Signatur-Operatorterme
              Pattern =? Pattern |                        %Goals
              Pattern -> Pattern |                        %Regeln
              ( Pattern )                                 %Klammerung

Term      ::= Number |                                    % Natürliche Zahlen
              Char |                                      % Zeichen
              String |                                    % Zeichenketten
              True | False |                              %Bool'sche Werte
              [[Term{, Term }* ] ] |                      % Listen
              ( Term{, Term }+ ) |                        % Tupel
              Name |                                      % Variablen
              Name( [Term{, Term }* ] ) |   % Fktsaufrufe oder Datenconstruktor
              VarName |                                   % Variablen
              Term : Term |                               % Listen
              Term BinOp Term |                           % Binäre Operatoren
              ~ Term |                                    % Bool'sche Negation
              BinOp |                                     % Sections
              ~ |                                         % Section
              % |                                         %Undefinierte Substitutionen
              {[ Term <- Term{, Term <- Term }* ] } |     % Substitutionen
              ( Term ) |                                  % Klammerung
              Term[Term <- Term]                          %Subtermersetzung

BinOp     ::= + |                                         % Addition
              - |                                         % Subtraktion
              * |                                         % Multiplikation
              \ |                                         % Division
              == | != |                                   % Gleichheit, Ungleichheit
              >| >=| <| <= |                              % Zahlenvergleich
              & | | |                                     % Logische UND, ODER
              =? | ->|                                    % Goal-, Regeloperator
              / |                                         % Subtermoperator
              ++                                          % Listenkonkatenation
```

Action	::= solve *Name* |	% Ein Goal lösen
	evaluate *Term* |	% Ausdruck auswerten
	confluent *Name* |	% Konfluenz testen
	terminates *Name Name*	% Terminierung testen
Name	::= {a| ... | z| _ } {*CharOrDigit* }*	
VarName	::= {A| ... | Z } {*CharOrDigit* }*	
CharOrDigit	::= {a| ... | Z| _| 0| ... | 9 }	
Number	::= {0| ... | 9 } {0| ... | 9 }*	
Char	::= ' *ASCIIChar* '	
String	::= " {*ASCIIChar* }* "	

4.2.3 Anmerkungen

- *DataDef:* Datentypen können nicht prametrisiert werden, also kann nicht geschrieben werden '`foo: [X]->foo_typ(X);`', wie z.B. in *Haskell*. Eine begrenzte Unterstützung für Parametrisierung wird durch das Frontend bereitgestellt: da es jedes Programm durch einen Präprozessor schickt (s. dort), kann man z.B. folgendes schreiben: '`#define foo_typ(X,N) foo:[X]->_foo_typ_/**/N`' dann führt '`foo_typ([A],ListOf_A);`' den neuen Typ ein.
  Man beachte, das Konstanten nie generisch sind, '`foo: X`' ist verboten.

- *Type:* Typvariablen werden durch Variablennamen notiert. So ist z.B. `foo: [X]->foo_typ;` zu lesen als `foo:` ∀ `X: [X]->foo_typ`. Der Operator `->` ist rechts-assoziativ.

- *FunctionDef:* Funktionen, die kein Argument haben, werden über das Pattern `()` definiert.

- *Strategydef:* Die Reichweite der lokalen Definitionen, die durch die '`where`' Klausel eingeführt werden, erstreckt sich auf die '`select`', '*Control*' und die '`output`' Abschnitte.

- *MetaRule:* Wie bei Funktionsdefinitionen steht das Pattern '`()`' für eine Metaregel ohne Argumente.

- *Control:* Die einzelnen Konstrukte haben folgende Prioritäten, geordnet von der niedrigsten zur höchsten.
  - Sequenzierung hat die niedrigste Priorität und ist rechts-assoziativ.
  - '`else`' ist rechts–assoziativ.
  - '`for`' und '`if`' haben dieselbe Priorität.

– '=>' ist nicht assoziativ.

- *Pattern:*
  - ':' ist rechts–assoziativ.
  - '[a,b,c]' steht als Abkürzung für 'a:b:c[]'.
  - *String*: '"ab"' steht für ''a':'b':[]'.
- *Term:* '[a,b,c]' und Strings sind genauso Abkürzungen, wie bei *Pattern.*
  - Funktionsaufrufe. Beim Aufruf von Funktionen ohne Argumente ist immer das leere Argumenttupel '()' anzugeben.
  - *VarName:* Kommen aus einem Pattern. So stellt 'X' keineswegs die Signaturvariable `X` dar, sondern die Signaturvariable, die als Argument von dem Pattern gematcht wurde.
  - ':' Der Listenconstructor ist rechts–assoziativ und hat eine höhere Priorität als '/' und '++' (s.u.).
  - Sections: Operatoren können ebenfalls als Argumente verwandt werden. So hat z.B. '=?' den Typ '`(Term, Term)->Goal`'.
  - Subtermersetzung. Diese hat den Typ '`(term,term,[nat]) -> term`'. Positionen in Signaturtermen[8] werden als Liste von Zahlen repräsentiert, z.B. '`f(X,Y)[[1]<-g(a)] = f(g(a),Y)`'.
- *BinOp:* Die Operatoren nach aufsteigender Priorität.
  - '/' ist links–assoziativ.
  - '++' ist rechts–assoziativ.
  - ':' ist rechts–assoziativ.
  - '~' ist rechts–assoziativ.
  - '&' und '|' sind rechts–assoziativ.
  - '==' und '!=' sind rechts–assoziativ.
  - '>', '>=', '<' und '<=' sind nicht–assoziativ.
  - '+' und '-' sind links–assoziativ.
  - '*' und '\' sind links–assoziativ.
- *Action:*
  - '`solve` GoalName': Löst das Goal 'GoalName'.
  - '`eval` t': Wertet den Term 't' zu Normalform aus und zeigt das Ergebnis.

[8] In anderen Termen ist der Begriff der Position nicht definiert.

– '`confluent` TRSName': Prüft Konfluenz des TRS 'TRSName'.
– '`terminates` TRSName PrecName': Prüft Terminierung des TRS 'TRSName' unter der Präzedenz 'PrecName'.

4.2.4 Nebenbedingungen

Außer den Nebenbedingungen, die sich aus der statischen Sematik ergeben, gelten noch folgende:

- Reservierte Wörter dürfen nicht als Namen verwandt werden: `sorts`, `funcs`, `cons`, `precs`, `trs`, `tactics`, `rules`, `goals`, `data`, `functions`, `where`, `strategy`, `local`, `global`, `select`, `output`, `break`, `if`, `for`, `do`, `else`, `True`, `False`
- Ausnahmen: `goals`, `trs` dürfen in Strategien als Name auftauchen, allerdings nicht auf der linken Seite der lokalen Definition (`where` Klausel) und nicht als Namen eines Datums oder der Strategie. In Metaregeln dürfen sie ebenso auftauchen, allerdings wiederum nicht als linke Seite einer Definition und nicht als Namen einer Metaregel.
- Wenn in Termen Datenconstructoren auftreten, so müssen sie vorher definiert worden sein, da sie andernfalls als Aufruf einer (noch nicht definierten) Funktion verstanden werden.
- In den Datendefinitionen einer Strategie müssen alle eingeführte Namen disjunkt sein.
- Die Namen auf den linken Seiten in einer lokalen Definition müssen untereinander verschieden sein.
- Gleiches gilt für Zuweisungsblöcke.
- Die Namen im '`output`' Statement einer Strategie müssen untereinander disjunkt sein.
- In einer *RuleDef* darf die linke Seite keine Variable sein, sondern muß ein Operatorterm sein.

4.3 $\mathcal{QIR}$

In diesem Abschnitt wird zuerst die $\mathcal{QIR}$ gegeben. Danach wird beschrieben, wie ein Programm in der $\mathcal{QDL}$ in einen Term der $\mathcal{QIR}$ übersetzt wird. Im folgenden wird dann immer davon ausgegangen, daß ein $\mathcal{QIR}$–Term durch die Übersetzung aus einem $\mathcal{QDL}$–Programm hervorgegangen ist.
Zur Definition wird ein ähnlicher Formalismus benutzt, wie zur Beschreibung der Grammatik. Allerdings werden einige primitive Konstrukte verwandt:

- Die Notation '[foo]' beschreibt Listen über dem Konstrukt 'foo'.
- '(a,b,...)' bezeichnet Tupel über den Konstrukten 'a', 'b', ...
- Natürliche Zahlen, Zeichen sind ebenfalls Bestandteil des Formalismusses und werden als $\mathbb{N}$, $\mathcal{A}$ notiert.
- Jedes Konstrukt definiert eine *Menge*. Die Notation 'cds∈[(Name,ConsObj)]' z.B. bedeutet, daß 'cds' eine Liste über Paaren über Namen und 'ConsObj' ist.
- Im Gegensatz zu der Grammatik sind Nebenbedingungen in den Definitionen erlaubt. Auch die direkte Angabe der Elemente mit zwischenliegender Ellipse '...' ist möglich: '$n_1, \ldots, n_m$'. Ein Index (hier m) gibt an, wie viele Elemente auftauchen.
- Wie bei der Grammatik sind Kommentare mit einem % gekennzeichnet.

4.3.1 Definition der $\mathcal{QIR}$

Program ::= `Prog`(*sds, cds, fds, rds, gds, tds, tads, pds, dads, fuds, stds*)
    wobei *sds* ∈ `[(`*Name,SortObj*`)]`   *cds*∈`[(`*Name,ConsObj*`)]`
    fds∈`[(`*Name,FunctorObj*`)]`   *rds*∈`[(`*Name,RuleObj*`)]`
    gds∈`[(`*Name,GoalObj*`)]`   *tds*∈`[(`*Name,TRSObj*`)]`
    tads∈`[(`*Name,TacticObj*`)]`   *pds*∈`[(`*Name,PrecObj*`)]`
    dads∈`[(`*Name,DataObj*`)]`   *fuds*∈`[(`*Name,FunctionObj*`)]`
    stds∈`[(`*Name,StrategyObj*`)]`

SortObj ::= `Sort`| `SubSort(`*Name*`)`

ConsObj ::= `Constr([`*Name*`],`*Name*`)`

FunctObj ::= `Functor([`*Name*`],`*Name*`)`

RuleObj ::= `Rule(App(DOp("->"),T([`*LOpPattern, OpPattern*`])))`

GoalObj ::= `Goal(App(DOp("=?"),T([`*OpPattern, OpPattern*`])))`

TRSObj ::= `TRS([`*Name*`])`

TacticObj ::= `Tactic([`*TacOrder*`],[(`*Name, Name*`)])`

PrecObj ::= `Prec([[`*Name*`]])`

DataObj ::= `Data(Simple(`*Name*`))` |
    `Data(Func(`*Typ*`,Simple(`*Name*`)))`

FunctionObj ::= `Function([(`*FOP, Term , Term, ld_i*`)])` |
    `Function([(App(`*FOP,SPattern*`),` *Term, Term, ld_i*`)])`
    wobei $ld_i = [(\mathtt{TV}(n_1^i), t_1^i), \ldots, (\mathtt{TV}(n_m^i), t_m^i)]$
    $m \geq 0$
    $\forall_{i,j,k} : n_j^i \in Name,\ t_j^i \in Term,\ n_j^i \neq n_k^i \iff j \neq k$

$StratObj$ ::= `Strategy([`$(gn_1,gt_1),\ldots,(gn_n,gt_n)$`],[`$(ln_1,lt_1),\ldots,(ln_m,lt_m)$`],`$Term$`,`
$Control$`,[(TV(`n_1`),`t_1`),`$\ldots$`,(TV(`n_k`),`t_k`)] ,[`$(on_1,ot_1),\ldots,(on_l,ot_l)$`],`
`[`$(mrn_1,mro_1),\ldots,(mrn_p,mro_p)$`])`
wobei $n,m,k,l \geq 0,\ p \geq 1$ $\quad n_i \in Name$
$gn_i, ln_i, on_i, mrn_i \in Name$
$gt_i, lt_i, t_i, ot_i \in Term$
$mro_i \in AMetaRule$
$\forall_{i,j,i \neq j}:$ $\quad gn_i \neq gn_j$
$\forall_{i,j,i \neq j}:$ $\quad ln_i \neq ln_j$
$\forall_{i,j,i \neq j}:$ $\quad n_i \neq n_j$
$\forall_{i,j,i \neq j}:$ $\quad on_i \neq on_j$
$\forall_{i,j,i \neq j}:$ $\quad mrn_i \neq mrn_j$

$LOpPattern$ ::= UOP |
`App(`UOP`,T([`$a_1,\ldots,a_n$`]))`
wobei $n \geq 1,\ \forall_{1 \leq i \leq n} : a_i \in OpPattern$

$OpPattern$ ::= UOP|
`App(`UOP`,T([`$a_1,\ldots,a_n$`]))`, wobei $n \geq 1,\ \forall_{1 \leq i \leq n} : a_i \in OpPattern$|
`V(`$Name$`)`

$TacOrder$ ::= `Wildcard`|
`TacName(`$Name$`)`

$SPattern$	::= `App(`DOP`,`$SPattern$`)`|	% Datenconstructor
	`T([`$p_1,\ldots,p_n$`])`, wobei $n \geq 2,\ \forall_{1 \leq i \leq n} : p_i \in SPattern$|	% Tupel
	TV |	% Variable
	`Var(`TV`)`|	% Signaturvariable
	`Op(`TV`)`|	% Signaturterm
	`True`| `False`|	% Bool'sche Werte
	`Nil`|	% Leere Liste
	`N(`$\mathbb{N}$`)`|	% Zahl
	`Char(`$\mathcal{A}$`)`|	% Zeichen
	`C(`SPattern,SPattern`)`|	% Liste
	DOP	% Datenconstruktorkonstante
UOP	::= `UOp(`Name`)`	% Ein Operator aus einer Signatur
FOP	::= `FOp(`Name`)`	% Ein Name einer Funktion
DOP	::= `DOp(`Name`)`	% Datenconstructor
TV	::= `TV(`Name`)`	%Variable, eingeführt über Pattern oder lokale Definition
Typ	::= `Simple(`Name`)`|	% Einfacher Typ, inkl. primitiver Typen
	`List(`Typ`)`|	% Listentyp
	`Tupel([`Typ`])`|	% Tupel
	`TypV(`Name`)`|	% Typvariable

		`Func`(Typ,Typ)|	% Funktionstyp
		`Unit`	% Typ von ()
GTyp	::=	$\forall$[Name].Typ	% Quantifizierter Typ
Term	::=	`App`(FOP $\cup$ TV $\cup$ DOP,Term)|	% Anwendung auf Argumente
		`T`([t_1,...,t_n]), wobei $n \geq 2$, $\forall_{1 \leq i \leq n} : t_i \in Term$|	% Tupel
		TV|	% Variable
		FOP|	% Funktionssymbol
		DOP|	% Datenconstructor
		`True`| `False`|	% Bool'sche Werte
		`Nil`|	% Leere Liste
		`N`($\mathbb{N}$)|	% Zahl
		`Char`($\mathcal{A}$)|	% Zeichen
		`C`(Term,Term)|	% Liste mit Kopfelement
		`Failed`|	% Fehlgeschlagene Substitution
		`Subst`([(Term, Term)])	%Substitution
Control	::=	`Break`|	%Abbruch
		`For`(SPattern,Term,Control)|	% Listeniteration
		`Call`(Name,Term $\cup$\{()\})|	% Metaregelaufruf
		`If`(Term,Control)|	% Bedingte Ausführung
		`Else`(Control,Control)|	%Alternative Ausführung
		`Seq`(Control,Control)|	% Sequentialisierung
		`Assign`(Control,[(n_1,t_1),...,(n_m,t_m)]),	
		wobei $m \geq 0$	% Zuweisung
		$n_i \in Name$ $\quad t_i \in Term$	
		$\forall_{i,j,i \neq j} : n_i \neq n_j$	
		|	
		`Empty`	% Leere Anweisung
AMetaRule	::=	`MetaRule`(SPattern $\cup$\{()\},Term,SPattern,	
		Term,[(an_1,at_1),...,(an_n,at_n)],Term,	
		[(`TV`(n_1),t_1),...,(`TV`(n_m),t_m)])	
		wobei $n, m \geq 0 \quad n_i \in \mathcal{N}$	
		$an_i \in Name \quad at_i, t_i \in Term$	
		$\forall_{i,i,i \neq j} : \quad n_i \neq n_j$	
		$\forall_{i,i,i \neq j} : \quad an_i \neq an_j$	

4.3.2 Anmerkungen

- *Program:* Die Definitionen eines Programmes werden in Listen gesammelt, wobei für jeden Objekttyp eine eigene Liste angelegt wird. Die Listen bestehen jeweils aus Paaren, wobei das erste Element den Namen des Objektes enthält, das zweite Element die Definition.

- *ConsObj:* Die Namen der Sorten, die in der Stelligkeit auftreten werden als Liste gehalten. Das zweite Element gibt den Namen der Zielsorte an. Für Konstanten ist die Liste natürlich leer.

- *RuleObj:* Die linke und rechte Seite der Regel werden zusammen mit dem Datenconstructor '`->`' zu einem Term zusammengefaßt. '`App`' steht dabei für die Anwendung eines Operators auf einen Term. Durch diese Definition ist sichergestellt, daß die linke Seite einer Regel keine Variable sein kann.

- *TRSObj:* Die Namen der Regeln werden als Liste repräsentiert.

- *TacticObj:* Die Ordnung der Regeln wird als erstes Element in Form einer Liste angegeben. Die Attribute sind eine Liste von Namenspaaren, wobei der erste Name den Namen einer Regel repräsentiert.

- *PrecObj:* Die Präzedenz wird als einer Liste von Ketten repräsentiert, wobei jede Kette wiederum als Liste repräsentiert wird.

- *DataObj:* Die Ähnlichkeit mit der Grammatikregel ist evident.

- *FunctionObj:* Eine Funktion wird repräsentiert, indem in einer Liste aus Quadrupeln alle Definitionsgleichungen gegeben werden. Jedes Quadrupel besteht aus dem Pattern der Regel, dem Ergebnis und dem Guard, sowie den lokalen Definitionen. Diese bestehen aus der Variablen, die definiert wird, und dem Term, an den die Variable gebunden wird.

- *StratObj:* Eine Strategie besteht aus der Definition der globalen/lokalen Daten in Form einer Liste, ähnlich der einer lokalen Definition. Als drittes Argument folgt der Selektionsausdruck, danach die Ablaufsteuerung, die lokalen Definitionen. Als sechstes folgt die Ausgabedefinition als Liste von Namen/Term–Paaren. Den Abschluß bilden die Metaregeln, die wiederum als Liste von Paaren repräsentiert werden, wobei der Name der Metaregel das erste Element, die Definition der Metaregel das zweite darstellt.

- *SPattern:* Die einzelnen Fälle, die ein Pattern annehmen kann, sollten nach Vergleich mit den Beispielen und der Grammatik verständlich sein.

- *GTyp:* Ein generalisierter Typ quantifiziert einige (oder alle) der Typvariablen in seinem Typargument. Die Intention dabei ist, daß nur die Typvariablen mit beliebigen Typen instanziiert werden können, die quantifiziert sind. Dies ist jedoch nur bei der Typinferenz eines Programmes von Bedeutung, da alle Funktionen und Datenconstructoren der 'obersten Schicht' immer alle Typvariablen in dem Typausdruck quantifizieren. Für eine genauere Begründung der Notwendigkeit teilweiser Quantifizierung, siehe [Mil78, Sch94, PJ86].

- *Term:* Die erste Regel benötigt Erklärung: Die Anwendung eines Operators auf einen Term, etwa eines Funktionsoperators, wie '+', wird mit dem 'App' Konstrukt beschrieben. Als erstes Argument nimmt es eines Operator, als zweites einen Term, auf den dieser Anzuwenden ist. Als Operator kann auch eine Variable 'TV(n)' auftreten. Selbstverständlich muß diese Variable dann zu einem Operator auswerten[9]. Der Rest der Regeln ist äquivalent zu den Produktionen der Grammatik, mit der Ausnahme das die dort auftauchenden Operatoren '+,*,/,...' nicht erscheinen. Sie werden als spezielle 'FOp' Operatoren repräsentiert, wie man nachher in der Übersetzung sehen wird.

- *Control:* Die einzelnen Konstrukte aus der Grammatik finden sich hier wieder, mit der Ausnahmen, daß das 'Seq' Konstrukt die Blöcke ersetzt. D.h., eine Reihenfolge von *Control* Statements wird durch eine geschachtelte Anordnung von 'Seq' Konstrukten ersetzt. Der leere Block wird dabei durch das Empty Konstrukt beschrieben.

- *AMetaRule:* Die Elemente der Definition repräsentieren nacheinander das Argumentpattern der Metaregel ('()' steht dabei für eine argumentlose Metaregel), die Bedingung der Regel, das Goalpattern, das Ergebnis, die Zuweisungen, die Ausgabe der Regel und die lokalen Definitionen.

4.3.3 Übersetzung $\mathcal{QDL} \rightarrow \mathcal{QIR}$

Hier wird beschrieben, wie ein korrektes $\mathcal{QDL}$ Programm in einen $\mathcal{QIR}$–Term überführt werden kann. Die Transformationsregeln bedienen sich einfacher Bedingungen und sollten intuitiv verständlich sein.
Die Übersetzung wird durch $\rightsquigarrow$ bezeichnet. Links steht ein Konstrukt der $\mathcal{QDL}$, rechts das entsprechende Konstrukt der $\mathcal{QIR}$.
Zur Abkürzung wird folgende Konvention eingeführt: '$\texttt{[}o_i\texttt{]}_{1\leq i\leq k}$' steht für '$\texttt{[}o_1, o_2, \ldots, o_k\texttt{]}$'.

$\mathcal{QDL}$-Prog:

$$\frac{[ObjDef_i \rightsquigarrow (sd_i, cd_i, \ldots, funcd_i, std_i)]^{1\leq i\leq n}}{ObjDef_1 \cdots ObjDef_n \rightsquigarrow \texttt{Prog}(sds, cds, \ldots, funcds, stds)}$$

$$\text{, wobei } \begin{aligned} sds &= sd_1 ++ \cdots ++ sd_n \\ cds &= cd_1 ++ \cdots ++ cd_n \\ &\cdots \\ funcds &= funcd_1 ++ \cdots ++ funcd_n \\ stds &= std_1 ++ \cdots ++ std_n \end{aligned}$$

ObjDef:

$$\frac{SortDef \rightsquigarrow sds}{SortDef \rightsquigarrow (sds, [], [], \ldots)}$$

$$\frac{ConsDef \rightsquigarrow cds}{ConsDef \rightsquigarrow ([], cds, [], \ldots)}$$

[9] Dies wird in der statischen Semantik expliziert.

...

$$\frac{FunctionDef \leadsto fuds}{FunctionDef \leadsto ([], \ldots, fuds, [])}$$

$$\frac{StrategyDef \leadsto std}{StrategyDef \leadsto ([], \ldots, [std])}$$

SortDef: $\underline{\texttt{sorts}}\ sd_1\underline{;}\ \ldots\ sd_n\underline{;}\ \underline{.}\ \leadsto [sd'_i]_{1 \le i \le n}$,

wobei $sd_i = \begin{cases} (name, \texttt{Sort}) & , sd_i = Name \\ (name_1, \texttt{SubSort}(name_2)), & sd_i = Name_1\underline{<}Name_2 \end{cases}$

FunctorDef: $\underline{\texttt{funcs}}\ fd_1\underline{;}\ \ldots\ fd_n\underline{;}\ \underline{.}\ \leadsto [fd'_i]_{1 \le i \le n}$, wobei

$$fd'_i = \begin{cases} (name_1, \texttt{Functor}([], name_2)), \ fd_i = Name_1\underline{:}\underline{[\texttt{->}]}Name_2 \\ (name_1, \texttt{Functor}([name_i]_{2 \le i \le k}, name_{k+1})), \\ \quad fd_i = Name_1\underline{:}Name_2 \ldots Name_k\underline{\texttt{->}}Name_{k+1}\ (k \ge 2) \end{cases}$$

ConsDef: Analog zu *FunctorDef.*

PrecDef:
$$\frac{[pc_{(i,j)} \leadsto pc'_{(i,j)}]^{(1 \le i \le n,\, 1 \le j \le k_i)}}{\underline{\texttt{precs}}\ pd_1\underline{;}\ \ldots\ pd_n\underline{;}\underline{.}\ \leadsto [pd'_i]_{1 \le i \le n}}$$

, wobei $pd_i = Name_i\underline{:}\ pc_{(i,1)}\underline{,}\ \ldots\ \underline{,}pc_{(i,k_i)}$
$pd'_i = (name_i, \texttt{Prec}([pc'_{(i,1)}, \ldots, pc'_{(i,k_i)}]))$

PrecChain: $Name_1\underline{<} \cdots \underline{<}Name_n \leadsto [name_i]_{1 \le i \le n}$

TRSDef: $\underline{\texttt{trs}}\ td_1\underline{;}\ \ldots\ td_n\underline{;}\ \underline{.}\ \leadsto [td'_i]_{1 \le i \le n}$

, wobei $td_i = Name_i^1\underline{:}\ Name_i^2\ \ldots\ Name_i^{k_i}$
$td'_i = (name_i^1, \texttt{TRS}([name_i^j]_{2 \le j \le k_i}))$

TacticDef:
$$\frac{[tad_i \leadsto (n_i, to_i, ta_i)]^{1 \le i \le n}}{\underline{\texttt{tactics}}\ tad_1\underline{;}\ \ldots\ tad_n\underline{;}\ \underline{.}\ \leadsto [tad'_i]_{1 \le i \le n}}$$

,wobei $tad'_i = (n_i, \texttt{Tactic}(to_i, ta_i))$

$$\frac{TacticChain \leadsto tc}{Name\underline{:}\ TacticChain \leadsto (name, tc, [])}$$

$$\frac{TacticAttribute \leadsto ta}{Name\underline{:}\ \underline{,}TacticAttribute \leadsto (name, [], ta)}$$

$$\frac{\begin{array}{c} TacticChain \leadsto tc \\ TacticAttribute \leadsto ta \end{array}}{Name\underline{:}\ TacticChain\ \underline{,}TacticAttribute \leadsto (name, tc, ta)}$$

TacticChain: $nw_1\underline{>} \cdots \underline{>}mw_n \leadsto [nw'_i]_{1 \le i \le n}$, wobei

$$nw'_i = \begin{cases} \texttt{Wildcard} & , \text{falls } nw_i = \underline{\texttt{@}} \\ \texttt{TacName}(nw_i) & , \text{falls } nw_i = Name \end{cases}$$

TacticAttributes: $Name_1\ Name'_1\underline{,}\ldots\underline{,}\ Name_k\ Name'_k \rightsquigarrow [(name_i, name'_i)]_{1\le i\le k}$

RuleDef:

$$\frac{[OpTerm_i \rightsquigarrow l_i,\ \ OpTerm'_i \rightsquigarrow r_i]^{1\le i\le n}}{\underline{\texttt{rules}}\ rd_1\underline{;}\ \ldots\ rd_n\underline{;}\ \underline{.} \rightsquigarrow [rd'_i]_{1\le i\le n}}$$

, wobei $rd_i = Name_i\underline{:}\ OpTerm_i\ \underline{\texttt{->}}\ OpTerm'_i$
$rd'_i = (name_i, \texttt{Rule(App(DOp("->"),T(}[l_i, r_i]\texttt{))))}$

GoalDef: Analog.

DataDef:

$$\frac{[dd_i \rightsquigarrow dd'_i]^{1\le i\le n}}{\underline{\texttt{data}}\ dd_1\underline{;}\ \ldots\ dd_n\underline{;}\ \underline{.} \rightsquigarrow [dd'_i]_{1\le i\le n}}$$

$$Name_1\underline{:}\ Name_2 \rightsquigarrow (name_1, \texttt{Data(Simple(}name_2\texttt{))})$$

$$\frac{Type \rightsquigarrow type}{Name_1\underline{:}\ Type\underline{\texttt{-->}}Name_2 \rightsquigarrow (name_1, \texttt{Data(}res\texttt{)})}$$

,wobei $res = \texttt{Func(}type\texttt{,Simple(}name_2\texttt{))}$

Type: $Name \rightsquigarrow \texttt{Simple(}name\texttt{)}$

$$\frac{Type \rightsquigarrow type}{\underline{[}Type\underline{]} \rightsquigarrow \texttt{List(}type\texttt{)}}$$

$$\frac{[Type_i \rightsquigarrow type_i]^{1\le i\le n}}{\underline{(}Type_1\underline{,}\ldots\underline{,}Type_n\underline{)} \rightsquigarrow \texttt{Tupel(}[type_i]_{1\le i\le n}\texttt{)}}$$

$$\frac{Type_1 \rightsquigarrow type_1,\ Type_2 \rightsquigarrow type_2}{Type_1\underline{\texttt{->}}Type_2 \rightsquigarrow \texttt{Func(}type_1\texttt{,}type_2\texttt{)}}$$

$$VarName \rightsquigarrow \texttt{TypV(}varname\texttt{)}$$

$$\frac{Type \rightsquigarrow type}{\underline{(}Type\underline{)} \rightsquigarrow type}$$

FunctionDef:

$$\frac{[fd_i \rightsquigarrow fd'_i]^{1\le i\le n}}{\underline{\texttt{functions}} fd_1\underline{;}\ \ldots\ fd_n\underline{;}\underline{.} \rightsquigarrow [fd'_i]_{1\le i\le n}}$$

$$\frac{[Name\underline{:}\,fdeq_i \rightsquigarrow fdeq'_i]^{1\le i\le n}}{Name\underline{:}\ fdeq_1\underline{;}\ \ldots\ fdeq_n\underline{;}\ \underline{;} \rightsquigarrow (name, fres)}$$

, wobei $fres = \texttt{Function(}[fdeq'_i]_{1\le i\le n}\texttt{)}$
, falls $\forall_i : fdeq_i = (\texttt{FOp(}name\texttt{)}, t_i, c_i, lds_i)$
oder $\forall_i : fdeq_i = (\texttt{App(FOp(}name\texttt{)},p_i\texttt{)}, t_i, c_i, lds_i)$

$$\frac{\begin{array}{c} pat \leadsto p \\ cond \leadsto ct \\ ldef \leadsto lds \\ Term \leadsto t \end{array}}{Name\,\underline{:}\,pat\,\underline{=}\,Term\;cond\;ldef \;\leadsto\; (p', t, ct, lds)}$$

$$,\text{wobei } p' = \begin{cases} \texttt{FOp(}name\texttt{)} & \text{, falls } p = \texttt{()} \\ \texttt{App(FOp(}name\texttt{),}p\texttt{)} & \text{, sonst} \end{cases}$$

ldef: $\leadsto []$

$$\frac{LocalDefs \leadsto lds}{\underline{\texttt{where}}\, LocalDefs \leadsto lds}$$

cond: $\leadsto \texttt{True}$

$$\frac{Term \leadsto t}{\underline{\texttt{, if}}\, Term \leadsto t}$$

pat: $\underline{\texttt{()}} \leadsto \texttt{()}$

$$\frac{Pattern \leadsto p}{Pattern \leadsto p}$$

StrategyDef:

$$\frac{\begin{array}{c} data \leadsto (gd, ld),\ Term \leadsto st \\ [Control_i \leadsto cntrl_i]^{1 \le i \le n},\ ldef \leadsto ldefs \\ output \leadsto outs,\ [mr_i \leadsto (n_i, mr'_i)]^{1 \le i \le k} \end{array}}{strategy \leadsto (name, res)}$$

$$\begin{array}{lll} \text{, wobei} & strategy = & \underline{\texttt{strategy}}\, Name\ data\ \underline{\texttt{select}}\, Term\,\underline{;} \\ & & Control_1 \ldots Control_n\ output\ ldef \\ & & \underline{\texttt{rules}}\, mr_1\underline{;} \ldots mr_k\underline{;}\ \underline{.} \\ & cntr = \texttt{Seq(}cntrl_1\texttt{,Seq(}\ldots\texttt{,Seq(}cntrl_{n-1}\texttt{,}cntrl_n\texttt{)))} & \\ & res = \texttt{Strategy(}gd\texttt{,}ld\texttt{,}st\texttt{,}cntr\texttt{,}ldefs\texttt{,}outs\texttt{,}[(n_i, mr'_i)]_{1 \le i \le k}\texttt{)} & \end{array}$$

$$\begin{array}{ll} \text{, falls} & \forall_{i,j,i \ne j} : gn_i \ne gn_j,\ ln_i \ne ln_j \\ & \forall_{i,j} : gn_i \ne ln_j \\ & \forall_{i,j,i \ne j} : n_i \ne n_j \\ \text{, wobei} & [(gn_i, gt_i)]_{1 \le i \le n_g} = gd \\ & [(ln_i, gt_i)]_{1 \le i \le n_l} = ld \end{array}$$

data: $\leadsto ([], [])$

$$\frac{LocalData \leadsto ld}{\underline{\texttt{data}}\, LocalData \leadsto ([], ld)}$$

$$\frac{GlobalData \leadsto gd}{\underline{\texttt{data}}\, GlobalData \leadsto (gd, [])}$$

$$\frac{LocalData \rightsquigarrow ld,\ GlobalData \rightsquigarrow gd}{\underline{\texttt{data}}\, LocalData\ GlobalData \rightsquigarrow (gd, ld)}$$

ldef: Wie bei *FunctionDef.*

output: $\rightsquigarrow []$

$$\frac{[Term_i \rightsquigarrow t_i]^{1 \le i \le n}}{\underline{\texttt{output}}\, o_1 \underline{;}\ \ldots\ o_n \underline{;} \rightsquigarrow [(name_i, t_i)]_{1 \le i \le n}}$$

, wobei $o_i = Name_i \underline{:} Term_i$

, falls $\forall_{i,j,i \ne j} : Name_i \ne Name_j$

MetaRule:

$$\frac{\begin{array}{c} patterns \rightsquigarrow rp,\ Pattern \rightsquigarrow gp \\ Term \rightsquigarrow rest,\ cond \rightsquigarrow ct \\ assigns \rightsquigarrow as,\ out \rightsquigarrow ot \\ ldef \rightsquigarrow ldefs \end{array}}{mr \rightsquigarrow (name, \texttt{MetaRule(}rp\texttt{,}ct\texttt{,}gp\texttt{,}rest\texttt{,}as\texttt{,}ot\texttt{,}ldefs\texttt{))}}$$

, wobei $mr = \ Name \underline{(} patterns \underline{)} \underline{:} Pattern \underline{\texttt{-->}} Term\ cond \underline{;}$
$assigns\ out\ ldef \underline{;}$

patterns: $\rightsquigarrow \texttt{()}$

$$\frac{Pattern \rightsquigarrow p}{Pattern \rightsquigarrow p}$$

$$\frac{[Pattern_i \rightsquigarrow p_i]^{1 \le i \le n}}{Pattern_1 \underline{,}\ \ldots\ \underline{,} Pattern_n \rightsquigarrow \texttt{T(}[p_i]_{1 \le i \le n}\texttt{)}}, n > 1$$

cond: Wie bei *FunctionDef.*

assigns: $\rightsquigarrow []$

$$\frac{[Term_i \rightsquigarrow t_i]^{1 \le i \le n}}{a_1 \underline{;}\ \ldots\ a_n \underline{;} \rightsquigarrow [(n_i, t_i)]_{1 \le i \le n}}$$

, wobei $a_i = Name_i \underline{:} Term_i$

, falls $\forall_{i,j,i \ne j} : Name_i \ne Name_j$

out: $\rightsquigarrow \texttt{Nil}$

$$\frac{Term \rightsquigarrow t}{\underline{\texttt{output}}\, Term \underline{;} \rightsquigarrow t}$$

ldef: Wie bei *FunctionDef.*

LocalData:

$$\frac{[Term_i \rightsquigarrow t_i]^{1 \le i \le n}}{\underline{\texttt{local}}\, d_1 \underline{;}\ \ldots\ d_n \underline{;} \rightsquigarrow [(\texttt{TV(}name_i\texttt{)}, t_i)_{1 \le i \le n}]}$$

, wobei $d_i = Name_i \underline{=} Term_i$

GlobalData: Analog zu *LocalData*.

Control: $\underline{\texttt{break;}} \rightsquigarrow \texttt{Break}$

$$\frac{args \rightsquigarrow t}{Name\underline{\texttt{(}}\,args\underline{\texttt{)}}\,\underline{\texttt{;}} \rightsquigarrow \texttt{Call(}name\texttt{,}t\texttt{)}}$$

, wobei args: $\rightsquigarrow \texttt{()}$

$$\frac{Term \rightsquigarrow t}{Term \rightsquigarrow t}$$

$$\frac{[Term_i \rightsquigarrow t_i]^{1 \leq i \leq n}}{Term_1\underline{\texttt{,}}\ \ldots\ \underline{\texttt{,}}Term_n \rightsquigarrow \texttt{T(}[t_i]_{1 \leq i \leq n}\texttt{)}}, n > 1$$

$$\frac{Control_1 \rightsquigarrow c_1,\ Control_2 \rightsquigarrow c_2}{Control_1\ \underline{\texttt{else}}\ Control_2 \rightsquigarrow \texttt{Else(}c_1\texttt{,}c_2\texttt{)}}$$

$\underline{\texttt{\{\}}} \rightsquigarrow \texttt{Empty}$

$$\frac{Control \rightsquigarrow c}{\underline{\texttt{\{}}\,Control\,\underline{\texttt{\}}} \rightsquigarrow c}$$

$$\frac{[Control_i \rightsquigarrow c_i]^{1 \leq i \leq n}}{\underline{\texttt{\{}}\,Control_1\ \ldots\ Control_n\underline{\texttt{\}}} \rightsquigarrow res}, n > 1$$

, wobei $res = \texttt{Seq(}c_1\texttt{,Seq(}\ldots\texttt{,Seq(}c_{n-1}\texttt{,}c_n\texttt{))}\cdots\texttt{)}$

$$\frac{Pattern \rightsquigarrow p,\ Term \rightsquigarrow t\ cntrls \rightsquigarrow cs}{\underline{\texttt{for}}\ Pattern\ \underline{\texttt{in}}\ Term\ cntrls \rightsquigarrow \texttt{For(}p\texttt{,}t\texttt{,}cs\texttt{)}}$$

$$\frac{Term \rightsquigarrow t\ cntrls \rightsquigarrow cs}{\underline{\texttt{if}}\,\underline{\texttt{(}}\,Term\,\underline{\texttt{)}}\,cntrls \rightsquigarrow \texttt{If(}t\texttt{,}cs\texttt{)}}$$

$$\frac{Control \rightsquigarrow c\ assigns \rightsquigarrow as}{Control\,\underline{\texttt{=>}}\,\underline{\texttt{\{}}\,assigns\underline{\texttt{\}}} \rightsquigarrow \texttt{Assign(}c\texttt{,}as\texttt{)}}$$

assigns: Wie bei *MetaRule*.

LocalDefs:

$$\frac{[Term_i \rightsquigarrow t_i]^{1 \leq i \leq n}}{a_1\underline{\texttt{;}}\ \ldots\ a_n\underline{\texttt{;}} \rightsquigarrow [(name_i, t_i)]_{1 \leq i \leq n}}$$

, wobei $a_i = Name_i\,\underline{\texttt{=}}\,Term_i$

, falls $\forall_{1 \leq i,j \leq n} : Name_i \neq Name_j \Leftrightarrow i \neq j$

OpTerm: $Name \rightsquigarrow \texttt{UOp}(name)$

$$\frac{OpTerm \rightsquigarrow ot}{Name\,\underline{(}\,OpTerm\,\underline{)} \rightsquigarrow \texttt{App(UOp(}name\texttt{)},ot\texttt{)}}$$

$$\frac{[OpTerm_i \rightsquigarrow ot_i]^{1\leq i\leq n}}{Name\,\underline{(}\,OpTerm_i\,\underline{)}_{1\leq i\leq n} \rightsquigarrow \texttt{App(UOp(}name\texttt{)},\texttt{T(}[ot_i]_{1\leq i\leq n}\texttt{))}}, n>1$$

Pattern:

$$Number \rightsquigarrow \texttt{N(}number\texttt{)}$$

$$Char \rightsquigarrow \texttt{Char(}char\texttt{)}$$

$$String \rightsquigarrow \texttt{C(Char(}c_1\texttt{)},\texttt{C(Char(}c_2\texttt{)},\ldots,\texttt{C(Char(}c_n\texttt{)},\texttt{Nil))}\cdots\texttt{)}$$
, wobei $string = \underline{"}c_1\cdots c_n\underline{"}$

$$\underline{\texttt{True}} \rightsquigarrow \texttt{True},\ \underline{\texttt{False}} \rightsquigarrow \texttt{False}$$

$$\frac{[Pattern_i \rightsquigarrow p_i]^{1\leq i\leq n}}{\underline{[}\,Pattern_i\,\underline{]}_{1\leq i\leq n} \rightsquigarrow \texttt{C(}p_1,\ldots,\texttt{C(}p_n,\texttt{Nil))}}, n\geq 0$$

$$\frac{[Pattern_i \rightsquigarrow p_i]^{1\leq i\leq 2}}{Pattern_1\,\underline{:}\,Pattern_2 \rightsquigarrow \texttt{C(}p_1,p_2\texttt{)}}$$

$$\frac{[Pattern_i \rightsquigarrow p_i]^{1\leq i\leq n}}{\underline{(}Pattern_i\underline{)}_{1\leq i\leq n} \rightsquigarrow \texttt{T(}[p_i]_{1\leq i\leq n}\texttt{)}}, n>1$$

$$Name \rightsquigarrow \texttt{TV(}name\texttt{)}$$

$$Name\underline{()} \rightsquigarrow \texttt{DOp(}name\texttt{)}$$

$$\frac{Pattern \rightsquigarrow p}{Name\underline{(}\,Pattern\,\underline{)} \rightsquigarrow \texttt{App(DOp(}name\texttt{)},p\texttt{)}}$$

$$\frac{[Pattern_i \rightsquigarrow p_i]^{1\leq i\leq n}}{Name\,\underline{(}\,Pattern_i\,\underline{)}_{1\leq i\leq 2\leq n} \rightsquigarrow \texttt{App(DOp(}name\texttt{)},\texttt{T(}[p_i]_{1\leq i\leq n}\texttt{))}}$$

$$VarName \rightsquigarrow \texttt{Var(TV(}name\texttt{))}$$

$$\underline{\#}Name \rightsquigarrow \texttt{Op(TV(}name\texttt{))}$$

$$\frac{[Pattern_i \rightsquigarrow p_i]^{1\le i\le 2}}{Pattern_1 \,\underline{\texttt{=?}}\, Pattern_2 \rightsquigarrow \texttt{App(DOp("=?"),T(}[p_1,p_2]\texttt{))}}$$

$Pattern_1 \,\underline{\texttt{->}}\, Pattern_2$ analog.

$$\frac{Pattern \rightsquigarrow p}{\underline{(}\, Pattern \,\underline{)} \rightsquigarrow p}$$

Term:

$Number$, $\underline{\texttt{False}}$, $\underline{\texttt{True}}$, $Char$, $String$ wie bei $Pattern$.

$$\underline{\texttt{[]}} \rightsquigarrow \texttt{Nil}$$

$$\frac{[Term_i \rightsquigarrow t_i]^{1\le i\le n}}{\underline{[}\, Term_i \,\underline{]}_{1\le i\le n} \rightsquigarrow \texttt{C(}t_1\texttt{,C(}t_2\texttt{,}\cdots\texttt{,C(}t_n\texttt{,Nil))}\cdots\texttt{)}}$$

$$\frac{[Term_i \rightsquigarrow t_i]^{1\le i\le n}}{\underline{(}\, Term_i \,\underline{)}_{1\le i\le n} \rightsquigarrow \texttt{T(}[t_i]_{1\le i\le n}\texttt{)}},\ n>1$$

$$Name \rightsquigarrow \texttt{TV(}name\texttt{)}$$

$$Name\,\underline{\texttt{()}} \rightsquigarrow atv$$
$$,\ atv = \begin{cases} \texttt{TV(}name\texttt{)} & \text{, falls } Name \in \{\texttt{"trs"}, \texttt{"goal"}, \texttt{"goals"}\} \\ \texttt{DOp(}name\texttt{)} & \text{, falls es Def. } (name, \texttt{Data(}dd\texttt{)}) \text{ gibt.} \\ \texttt{FOp(}name\texttt{)} & \text{, sonst.} \end{cases}$$

$$\frac{Term \rightsquigarrow t}{Name\,\underline{(}\, Term \,\underline{)} \rightsquigarrow \texttt{App(}op\texttt{,}t\texttt{)}}$$
$$,\ op = \begin{cases} \texttt{TV(}name\texttt{)} & \text{, falls es lokale Variable } name \text{ gibt} \\ & \quad \text{oder } Name \in \{\texttt{"trs"}, \texttt{"goal"}, \texttt{"goals"}\} \\ \texttt{DOp(}name\texttt{)} & \text{, falls es Def. } (name, \texttt{Data(}dd\texttt{)}) \text{ gibt.} \\ \texttt{FOp(}name\texttt{)} & \text{, sonst.} \end{cases}$$

$$\frac{[Term_i \rightsquigarrow t_i]^{1\le i\le 2\le n}}{Name\,\underline{(}\, Term_i \,\underline{)}_{1\le i\le n} \rightsquigarrow \texttt{App(}op\texttt{,T(}[t_i]_{1\le i\le n}\texttt{))}},\ op \text{ wie oben.}$$

$$VarName \rightsquigarrow \texttt{TV(}varname\texttt{)}$$

$$\frac{[Term_i \rightsquigarrow t_i]^{1\le i\le 2},\ BinOp \rightsquigarrow op}{Term_1\ BinOp\ Term_2 \rightsquigarrow \texttt{App(}op\texttt{,T(}[t_1,t_2]\texttt{))}}$$

$$\frac{[Term_i \rightsquigarrow t_i]^{1\leq i\leq 2}}{Term_1\ \underline{\texttt{:}}\ Term_2 \rightsquigarrow \texttt{C(}t_1\texttt{,}t_2\texttt{)}}$$

$$\frac{Term \rightsquigarrow t}{\underline{\texttt{\~{}}}Term \rightsquigarrow \texttt{App(FOp("\~{}"),}t\texttt{)}}$$

$$\underline{\texttt{\%}} \rightsquigarrow \texttt{Failed}$$

$$\frac{[Term_i \rightsquigarrow t_i]^{1\leq i\leq n},\ [Term'_i \rightsquigarrow t'_i]^{1\leq i\leq n}}{\underline{\texttt{\{}}\,Term_i\ \underline{\texttt{<-}}\ Term'_i\ \underline{\texttt{\}}}_{1\leq i\leq n} \rightsquigarrow \texttt{Subst(}[(t_i,t'_i)]_{1\leq i\leq n}\texttt{)}}$$

$$\frac{Term \rightsquigarrow t}{\underline{\texttt{(}}\,Term\ \underline{\texttt{)}} \rightsquigarrow t}$$

$$\frac{[Term_i \rightsquigarrow t_i]^{1\leq i\leq 3}}{Term_1\ \underline{\texttt{[}}\,Term_2\ \underline{\texttt{<-}}\ Term_2\ \underline{\texttt{]}} \rightsquigarrow \texttt{App(FOp("[<-]"),T(}[t_1,t_2,t_3]\texttt{))}}$$

BinOp:

$$\underline{\texttt{+}} \rightsquigarrow \texttt{FOp("+")}$$

$\underline{\texttt{-}}$,$\underline{\texttt{*}}$,$\underline{\texttt{\textbackslash}}$,$\underline{\texttt{==}}$,$\underline{\texttt{!=}}$, $\underline{\texttt{>}}$, $\underline{\texttt{>=}}$, $\underline{\texttt{<}}$, $\underline{\texttt{<=}}$, $\underline{\texttt{\&}}$, $\underline{\texttt{|}}$, $\underline{\texttt{/}}$, $\underline{\texttt{++}}$ analog.

$$\underline{\texttt{=?}} \rightsquigarrow \texttt{DOp("=?")}$$

$$\underline{\texttt{->}} \rightsquigarrow \texttt{DOp("->")}$$

4.3.4 Anmerkungen zur Übersetzung

Die meisten Übersetzungsregeln sind offensichtlich. Daher nur einige Bemerkungen zu bestimmten Regeln.

- Die Prioritätenregeln für die Operatoren, wie sie mit der Grammatik gegeben wurden, haben die Mehrdeutigkeiten der Grammatik aufgehoben. Die Übersetzung kann sich also einfach an den Regeln orientieren.
- *Term.* Wird ein *Name*$\underline{\texttt{()}}$ übersetzt, so kann es drei Möglichkeiten geben, in welches Zielkonstrukt er übersetzt werden soll:
  1. Eine lokale Variable[10], die einen Funktionstyp hat.
  2. Einen Datenconstructor.

[10] Oder eine Patternvariable

3. Eine Funktion.

Diese Mehrdeutigkeit kann aber aufgelöst werden. Ist der Name in einer aktiven lokalen Variabalendefinition, wozu auch die Variablen in Funktionspattern gehören, enthalten, dann wird er zu einer Variablen übersetzt. Ebenfalls als Variablen werden die Datennamen aufgefaßt, inklusive der vordefinierten.

Wenn dies nicht der Fall ist, aber eine Datendefinition für diesen Namen schon abgearbeitet wurde, dann wird dieser Name als der Name dieses Constructors aufgefaßt und entsprechend übersetzt. Sonst ist er als Name einer (ggf. noch undefinierten) Funktion aufzufassen und zu übersetzen.

Die Entscheidung, welche Alternative zutrifft ist nicht lokal treffbar. Man braucht dazu eine Liste aller bisher definierten lokalen Variablen und Datenconstructoren. Es wäre kein Problem, den Übersetzungsregeln diese Liste als Parameter mitzugeben, außer das dies die Regeln unnötig verkompliziert und überlädt. Deshalb wurde bei der Präsentation der Übersetzungsregeln darauf verzichtet.

- Ebenfalls bemerkenswert ist, daß der Listenconstructor nicht als Section existiert.

4.4 Konventionen

Die Semantik der Konstrukte wird wiederum als natural semantics gegeben. Dabei treten nun Ableitungen der Form $s \overset{\mathsf{Regelsatz}}{\vdash} s'$ auf. Dies besagt einfach, daß $s \vdash s'$ mit den Inferenzregeln aus dem Abschnitt **Regelsatz** hergeleitet werden soll. Dies ermöglicht eine Modularisierung der Inferenz.
In den Regeln tauchen, neben Konstrukten aus der $\mathcal{QIR}$, noch *Umgebungen* auf. Umgebungen dienen zur Protokollierung von Bindungen von Namen (oder Konstrukten) an Konstrukte.

- Γ protokolliert Bindungen von Namen an Objekte. Die einzelnen Bindungen sind von der Form $n \mapsto o$, $\mathtt{TV}(n) \mapsto o$, etc.
- Θ, Ξ protokollieren Bindungen von Namen an Typen, etwa $\Theta = \{\mathtt{TV}(n) \mapsto \mathtt{Simple("Bool")}\}$.
- Δ protokolliert Bindungen von Namen an Objekte in der dynamischen Semantik.

Wir betrachten solche Umgebungen auch als Funktionen und schreiben etwa $\Gamma(n)$, was mit $\Gamma = \{n \mapsto \mathtt{Data}(n')\}$ gleichbedeutend ist, mit $\mathtt{Data}(n')$. Der Begriff der Domain ($\mathcal{D}(\Gamma)$) überträgt sich ebenfalls. Oft benutzen wir die Einschränkung einer Umgebung auf bestimmte Konstrukte: $\Gamma_{|\mathtt{Function}}$ steht für die Menge $\{n \mapsto \mathtt{Function}(ds) | n \mapsto \mathtt{Function}(ds) \in \Gamma\}$.
Man beachte, daß eine Umgebung nicht unbedingt eine Funktion sein muß, so erzeugt Regel (5) etwa keine Funktion, da die beiden Paare $n \mapsto \mathtt{Sort}$ und $n \mapsto \mathtt{SubSort}(n')$ in

die Umgebung aufgenommen werden. Wird eine Umgebung als Funktion verwandt, ist immer sichergestellt, daß dies eindeutig ist, d.h. das es zu einem n nur genau ein $n \mapsto o$ in der Umgebung gibt.
Um eine Umgebung zu modifizieren stehen zwei Möglichkeiten zur Verfügung. Einmal können mit der normalen Mengenvereinigung ($\cup$) neue Elemente aufgenommen werden, zum anderen gibt es die Operation '$\oplus$', welche die neu hinzukommenden Elemente an die Stelle der alten setzt, z.B.: $\{n \mapsto o, n' \mapsto o'\} \cup \{n \mapsto o''\} = \{n \mapsto o, n \mapsto o'', n' \mapsto o'\}$, aber $\{n \mapsto o, n' \mapsto o'\} \oplus \{n \mapsto o''\} = \{n \mapsto o'', n' \mapsto o'\}$.
In den Regeln tauchen oft Variablen auf, die für Typen stehen (z.B. in **termtype**). Dabei steht τ für einen unquantifizierten Typ, wie `Simple("Bool")` und σ für einen quantifizierten Typ, wie etwa `∀["a"].List(TypV("a"))`. Die Funktion $Instanz(\tau)$ instantiiert solche quantifizierten Typen: Für die Typvariablen werden Typen eingesetzt. Die Funktion $Generalisiere(\tau, \Theta)$ generalisiert die Typvariablen in dem Typ τ, die nicht frei in Θ vorkommen. Eine Typvariable heißt dabei frei, wenn sie nicht durch Quantifizierung gebunden ist. Mit $FV(\sigma)$ bezeichnet man dabei die freien Typvariablen in σ. Siehe im übrigen [Sch94, Mil78] für eine ausführlichere Beschreibung dieser Sachverhalte.
Die Regeln benutzen eine Reihe von Initialumgebungen, in denen die primitiven Objekte der Sprache enthalten sind. Dies sind im einzelnen:

- Die primitiven Typnamen.

  Sie bezeichnen die Namen der primitiven Datentypen.
  $PrimDataNames' = \{$`"nat"`, `"bool"`, `"term"`, `"char"`, `"substitution"`, `"arg"`$\}$
  $PrimDataNames = PrimDataNames' \cup \{$`"rule"`, `"goal"`$\}$
  $PrimDataNames'$ enthält die Namen, die nicht als Zieltyp eines benutzerdefinierten Datenconstructors auftauchen können, d.h. `"goal"` und `"rule"` dürfen als einzige dieser primitiven Datentypen auch als Zieltyp eines benutzerdefinierten Datenconstructors auftauchen.

- Die primitiven Datenconstructoren.

  Es sind nur die Constructoren für eine Regel und ein Goal vorgegeben.
  $PrimDOPs = \{$ `"=?"` $\mapsto$ `Data("term"` $\times$ `"term"` $\rightarrowtail$ `"goal")`, .
  `"->"` $\mapsto$ `Data("term"` $\times$ `"term"` $\rightarrowtail$ `"rule")`$\}$
  In dieser Definition wurden einige Abkürzungen verwandt, etwa $a \times b$ für `Tupel(`$[a, b]$`)` und `"goal"` für `Simple("goal")` und $a \rightarrowtail b$ für `Func(`a`,`b`)`. Diese Abkürzungen werden sich auch weiterhin als nützlich erweisen, wo noch $[a]$ für `List(`a`)` verwandt wird.

- Die primitiven Funktionen.

  Sie werden durch die Inferenzregeln **prim** definiert, da sie nicht als Funktionsdefinitionen in einem Programm gegeben werden können. Die Umgebung enthält für sie deshalb auch nur "dummy" Einträge der Form `FOp(`n`)` $\mapsto$ `Function([])`.

$$PrimFOPs = \{\texttt{FOp}(n_i) \mapsto \texttt{Function}([])|n_i \in \{\ \texttt{"/"}, \texttt{"[<-]"}, \texttt{"+"}, \texttt{"-"}, \texttt{"*"}, \texttt{"\textbackslash"}, \texttt{">"}, \texttt{"<"}, \texttt{"<="}, \texttt{">="}, \texttt{"=="}, \texttt{"!="}, \texttt{"argsnum"}, \texttt{"opof"}, \texttt{"attributes"}, \texttt{"newvar"}, \texttt{"show"}, \texttt{"dom"}, \texttt{"apply"}, \texttt{"compose"}\}\}$$

- Die Namen der primitiven Funktionen, die kein Argument haben.
  $Prim^0 = \{\texttt{"newvar"}, \texttt{"attributes"}\}$

- Die Namen der primitiven Funktionen, die mindestens ein Argument haben.
  $Prim^1 =$ alle Namen primitiver Funktionen, die nicht in $Prim^0$ auftauchen.

- Die vordefinierten, nicht-primitiven Funktionen: $PreFOPs$.

  Diese könnten auch über ein Programm definiert werden[11]. Die Menge $PreFOPs$ ergäbe sich durch die Übersetzung des nachstehenden Programmes:

```
functions
 ~: True = False;
    False = True;
    ;

 |: (True,x) = True;
    (False,x) = x;
    ;

 &: (False,x) = False;
    (True,x)  = True;
    ;

 ++: ([],ys)    = ys;
     (x:xs,ys)  = x:(xs++ys);
    ;
.
```

- Die Typen der primitiven und vordefinierten Funktionen.
  $PrimTypEnv =$

[11]Sind sie nicht, da ihre Namen in einem Programm nicht in einer Definition auftauchen könnten.

$$
\begin{array}{l}
\{\ \texttt{FOp("/")} \mapsto \texttt{"term"} \times [\texttt{"nat"}] \rightarrowtail \texttt{"term"},\\
\quad \texttt{FOp("[<-]")} \mapsto \texttt{"term"} \times [\texttt{"nat"}] \times \texttt{"term"} \rightarrowtail \texttt{"term"},\\
\quad \texttt{FOp("+")} \mapsto \texttt{"nat"} \times \texttt{"nat"} \rightarrowtail \texttt{"nat"},\\
\quad \texttt{FOp("-")} \mapsto \texttt{"nat"} \times \texttt{"nat"} \rightarrowtail \texttt{"nat"},\\
\quad \texttt{FOp("*")} \mapsto \texttt{"nat"} \times \texttt{"nat"} \rightarrowtail \texttt{"nat"},\\
\quad \texttt{FOp("\textbackslash")} \mapsto \texttt{"nat"} \times \texttt{"nat"} \rightarrowtail \texttt{"nat"},\\
\quad \texttt{FOp(">")} \mapsto \texttt{"nat"} \times \texttt{"nat"} \rightarrowtail \texttt{"bool"},\\
\quad \texttt{FOp("<")} \mapsto \texttt{"nat"} \times \texttt{"nat"} \rightarrowtail \texttt{"bool"},\\
\quad \texttt{FOp(">=")} \mapsto \texttt{"nat"} \times \texttt{"nat"} \rightarrowtail \texttt{"bool"},\\
\quad \texttt{FOp("<=")} \mapsto \texttt{"nat"} \times \texttt{"nat"} \rightarrowtail \texttt{"bool"},\\
\quad \texttt{FOp("==")} \mapsto \forall[\texttt{"a"}].\texttt{TypV("a")} \times \texttt{TypV("a")} \rightarrowtail \texttt{"bool"},\\
\quad \texttt{FOp("!=")} \mapsto \forall[\texttt{"a"}].\texttt{TypV("a")} \times \texttt{TypV("a")} \rightarrowtail \texttt{"bool"},\\
\quad \texttt{FOp("\textasciitilde")} \mapsto \texttt{"bool"} \rightarrowtail \texttt{"bool"},\\
\quad \texttt{FOp("\&")} \mapsto \texttt{"bool"} \times \texttt{"bool"} \rightarrowtail \texttt{"bool"},\\
\quad \texttt{FOp("|")} \mapsto \texttt{"bool"} \times \texttt{"bool"} \rightarrowtail \texttt{"bool"},\\
\quad \texttt{FOp("++")} \mapsto \forall[\texttt{"a"}].[\texttt{TypV("a")}] \times [\texttt{TypV("a")}] \rightarrowtail [\texttt{TypV("a")}],\\
\quad \texttt{FOp("argsnum")} \mapsto \texttt{"term"} \rightarrowtail \texttt{"nat"},\\
\quad \texttt{FOp("opof")} \mapsto \texttt{"term"} \rightarrowtail (\texttt{"arg"} \rightarrowtail \texttt{"term"}),\\
\quad \texttt{FOp("attributes")} \mapsto [([\texttt{"char"}],[[\texttt{"char"}]])],\\
\quad \texttt{FOp("newvar")} \mapsto \texttt{"term"},\\
\quad \texttt{FOp("show")} \mapsto \forall[\texttt{"a"}].\texttt{TypV("a")} \rightarrowtail [\texttt{"char"}],\\
\quad \texttt{FOp("dom")} \mapsto \texttt{"substitution"} \rightarrowtail [\texttt{"term"}],\\
\quad \texttt{FOp("compose")} \mapsto \texttt{"substitution"} \times \texttt{"substitution"} \rightarrowtail \texttt{"substitution"},\\
\quad \texttt{FOp("apply")} \mapsto \texttt{"substitution"} \times \texttt{"term"} \rightarrowtail \texttt{"term"}\}
\end{array}
$$

Als nächstes wird nun die statische Semantik von Quetzal gegeben. Im Anschluß finden sich einige Kommentare dazu. Daraufhin erscheint die dynamische Semantik. Den Abschluß bilden Kommentare dazu und einige Lemmata.

4.5 Statische Semantik

4.5.1 Programm (prog)

$$
(1)\ \dfrac{\begin{array}{c}
\emptyset \overset{\text{sorts}}{\vdash} sds : \Gamma_1 \quad \Gamma_1 \overset{\text{cons}}{\vdash} cds : \Gamma_2 \quad \Gamma_2 \overset{\text{functor}}{\vdash} fds : \Gamma_3\\
\Gamma_3 \overset{\text{rule}}{\vdash} rds : \Gamma_4 \quad \Gamma_4 \overset{\text{goal}}{\vdash} gds : \Gamma_5 \quad \Gamma_5 \overset{\text{trs}}{\vdash} tds : \Gamma_6\\
\Gamma_6 \overset{\text{tactic}}{\vdash} tads : \Gamma_7 \quad \Gamma_7 \overset{\text{prec}}{\vdash} pds : \Gamma_8\\
(\Gamma_9, PrimDataNames) \overset{\text{data}}{\vdash} dds : \Gamma_{10}\\
(\Gamma_{10}, PrimTypEnv) \overset{\text{function}}{\vdash} fuds : (\Gamma_{11}, \Theta_1)\\
(\Gamma_{11}, \Theta_1) \overset{\text{strategy}}{\vdash} stds : \Gamma_{12}
\end{array}}{\vdash \texttt{Prog}(sds, cds, fds, rds, gds, tds, tads, pds, pds, dds, fuds, stds) : (\Gamma_8, \Gamma_{12}, \Theta_1)}
$$

, falls $GenTran(GenOrdering(\Gamma_1))$ irreflexiv ist.
, wobei $\Gamma_9 = PrimDOPs \cup PrimFOPs \cup PreFOPs$

4.5.2 Sorten

sorts

$$(2) \Gamma \vdash [] : \Gamma \qquad (3)\frac{\Gamma \vdash o : \Gamma_1 \quad \Gamma_1 \vdash os : \Gamma_2}{\Gamma \vdash [o \,|\, os] : \Gamma_2}$$

(4) $\Gamma \vdash (n, \texttt{Sort}) : \Gamma \cup \{n \mapsto \texttt{Sort}\}$

(5) $\Gamma \vdash (n, \texttt{SubSort}(n')) : \Gamma \cup \{n \mapsto \texttt{Sort}, n \mapsto \texttt{SubSort}(n')\}$, falls $n' \in \mathcal{D}(\Gamma_{|\texttt{Sort}})$

4.5.3 Constructoren

cons

$$(6) \Gamma \vdash [] : \Gamma \qquad (7)\frac{\Gamma \vdash o : \Gamma_1 \quad \Gamma_1 \vdash os : \Gamma_2}{\Gamma \vdash [o \,|\, os] : \Gamma_2}$$

(8) $\Gamma \vdash (n, \texttt{Constr}([n_1, \ldots, n_k], n')) : \Gamma \cup \Gamma'$
, falls $n \notin \mathcal{D}(\Gamma)$
und $\forall_{1 \leq i \leq k} : n_i \in \mathcal{D}(\Gamma_{|\texttt{Sort}})$
und $n' \in \mathcal{D}(\Gamma_{|\texttt{Sort}})$
, wobei $\Gamma' = \{n \mapsto \texttt{Constr}([n_1, \ldots, n_k], n')\}$

4.5.4 Functoren

functor

$$(9) \Gamma \vdash [] : \Gamma \qquad (10)\frac{\Gamma \vdash p : \Gamma_1 \quad \Gamma_1 \vdash os : \Gamma_2}{\Gamma \vdash [o \,|\, os] : \Gamma_2}$$

(11) $\Gamma \vdash (n, \texttt{Functor}([n_1, \ldots, n_k], n')) : \Gamma \cup \Gamma'$
, falls $n \notin \mathcal{D}(\Gamma)$
und $\forall_{1 \leq i \leq k} : n_i \in \mathcal{D}(\Gamma_{|\texttt{Sort}})$
und $n' \in \mathcal{D}(\Gamma_{|\texttt{Sort}})$
, wobei $\Gamma' = \{n \mapsto \texttt{Functor}([n_1, \ldots, n_k], n')\}$

4.5.5 Regeln

rule

$$(12) \Gamma \vdash [] : \Gamma \qquad (13)\frac{\Gamma \vdash p : \Gamma_1 \quad \Gamma_1 \vdash os : \Gamma_2}{\Gamma \vdash [o \,|\, os] : \Gamma_2}$$

$$(14) \ \frac{(\Gamma, \emptyset) \overset{\text{sort-of}}{\vdash} \texttt{UOp}(n') : (s_1, \Theta_1) \quad (\Gamma, \Theta_1) \overset{\text{sort-of}}{\vdash} r : (s_2, \Theta_1) \quad \Gamma \overset{\text{sort-lt}}{\vdash} s_1 : s_2}{\Gamma \vdash (n, \texttt{Rule(App(DOp("->"),T(}[\texttt{UOp}(n'), r]\texttt{))))} : \Gamma \cup \Gamma'}$$

, falls $n \notin \mathcal{D}(\Gamma)$ und $n' \in \mathcal{D}(\Gamma_{|\texttt{Functor}})$
, wobei $\Gamma' = \{n \mapsto \texttt{Rule(App(DOp("->"),T(}[\texttt{UOp}(n'), r]\texttt{))))}\}$

$$(15)\ \frac{(\Gamma,\emptyset)\overset{\textsf{sort-of}}{\vdash}\texttt{App(UOp(}n'\texttt{)}a\texttt{, :)}(s_1,\Theta_1)\quad(\Gamma,\Theta_1)\overset{\textsf{sort-of}}{\vdash}r:(s_2,\Theta_1)\quad\Gamma\overset{\textsf{sort-lt}}{\vdash}s_1:s_2}{\Gamma\vdash(n,\texttt{Rule(App(DOp("->"),T([App(UOp(}n'\texttt{),}a\texttt{)},r\texttt{]))))}:\Gamma\cup\Gamma'}$$

, falls $n\notin\mathcal{D}(\Gamma)$ und $n'\in\mathcal{D}(\Gamma_{|\texttt{Functor}})$
, wobei $\Gamma'=\{n\mapsto\texttt{Rule(App(DOp("->"),T([App(UOp(}n'\texttt{),}a\texttt{)},r\texttt{]))))}\}$

4.5.6 Goals

goal

$$(16)\Gamma\vdash[]:\Gamma\qquad(17)\frac{\Gamma\vdash o:\Gamma_1\quad\Gamma_1\vdash os:\Gamma_2}{\Gamma\vdash[o\,|\,os]:\Gamma_2}$$

$$(18)\ \frac{(\Gamma,\emptyset)\overset{\textsf{sort-of}}{\vdash}l:(s_1,\Theta_1)\quad(\Gamma,\Theta_1)\overset{\textsf{sort-of}}{\vdash}r:(s_2,\Theta_2)\quad\Gamma\overset{\textsf{sort-lt}}{\vdash}s_1:s_2}{\Gamma\vdash(n,\texttt{Goal(App(DOp("=?"),T([}l,r\texttt{]))))}:\Gamma\cup\Gamma'}$$

, falls $n\notin\mathcal{D}(\Gamma)$
, wobei $\Gamma'=\{n\mapsto\texttt{Goal(App(DOp("=?"),T([}l,r\texttt{]))))}\}$

$$(19)\ \frac{(\Gamma,\emptyset)\overset{\textsf{sort-of}}{\vdash}l:(s_1,\Theta_1)\quad(\Gamma,\Theta_1)\overset{\textsf{sort-of}}{\vdash}r:(s_2,\Theta_2)\quad\Gamma\overset{\textsf{sort-lt}}{\vdash}s_2:s_1}{\Gamma\vdash(n,\texttt{Goal(App(DOp("=?"),T([}l,r\texttt{]))))}:\Gamma\cup\Gamma'}$$

, falls $n\notin\mathcal{D}(\Gamma)$
, wobei $\Gamma'=\{n\mapsto\texttt{Goal(App(DOp("=?"),T([}l,r\texttt{]))))}\}$

4.5.7 Sortentyping und –inklusion (sort-of, sort-lt)

sort-lt

(20) $\Gamma\vdash s:s$

(21) $\Gamma\vdash s_1:s_2$, falls $\{s_1\mapsto\texttt{Sort},s_2\mapsto\texttt{Sort},s_2\mapsto\texttt{SubSort(}s_1\texttt{)}\}\subseteq\Gamma$

$$(22)\ \frac{\Gamma\vdash s_1:s_2\quad\Gamma\vdash s_2:s_3}{\Gamma\vdash s_1:s_3}$$

sort-of

(23) $(\Gamma,\Theta)\vdash\texttt{V(}n\texttt{)}:(s,\Theta)$, falls $n\mapsto s\in\Theta$

(24) $(\Gamma,\Theta)\vdash\texttt{V(}n\texttt{)}:(s,\Theta\cup\{n\mapsto s\})$, falls $n\notin\mathcal{D}(\Theta)$ und $s\in\mathcal{D}(\Gamma_{|\texttt{Sort}})$

(25) $(\Gamma,\Theta)\vdash\texttt{UOp(}n\texttt{)}:s$, falls $\exists n\mapsto\texttt{Constr([],}s\texttt{)}\in\Gamma$ oder $\exists n\mapsto\texttt{Functor([],}s\texttt{)}\in\Gamma$

$$(26)\ \frac{\begin{array}{ccc}(\Gamma,\Theta)\vdash a_1:(s_1,\Theta_1) & \cdots & (\Gamma,\Theta_{k-1})\vdash a_k:(s_k,\Theta_k)\\ \Gamma\overset{\textsf{sort-lt}}{\vdash}s'_1:s_1 & \cdots & \Gamma\overset{\textsf{sort-lt}}{\vdash}s'_k:s_k\end{array}}{(\Gamma,\Theta)\vdash\texttt{App(UOp(}n\texttt{),T([}a_1,\ldots,a_k\texttt{]))}:(s',\Theta_k)}$$

, falls $\exists n\mapsto\texttt{Constr([}s'_1,\ldots,s'_k\texttt{],}s'\texttt{)}\in\Gamma$
oder $\exists n\mapsto\texttt{Functor([}s'_1,\ldots,s'_k\texttt{],}s'\texttt{)}\in\Gamma$

4.5.8 TRSs

trs

$$(27)\Gamma \vdash [] : \Gamma \qquad (28)\frac{\Gamma \vdash o : \Gamma_1 \quad \Gamma_1 \vdash os : \Gamma_2}{\Gamma \vdash [o \,|\, os] : \Gamma_2}$$

$$(29)\ \Gamma \vdash (n, \texttt{TRS}([n_1, \ldots, n_k])) : \Gamma \cup \Gamma'$$
$$\text{, falls } n \notin \mathcal{D}(\Gamma) \text{ und } \forall_{1 \leq i \leq k} : n_i \in \mathcal{D}(\Gamma_{|\texttt{Rule}})$$
$$\text{, wobei } \Gamma' = \{n \mapsto \texttt{TRS}([n_1, \ldots, n_k])\}$$

4.5.9 Taktiken (tactic, tactic1, tactic2)

tactic

$$(30)\Gamma \vdash [] : \Gamma \qquad (31)\frac{\Gamma \vdash o : \Gamma_1 \quad \Gamma_1 \vdash os : \Gamma_2}{\Gamma \vdash [o \,|\, os] : \Gamma_2}$$

$$(32)\ \frac{\Gamma \overset{\mathsf{tactic1}}{\vdash} ts : (wn, rs) \quad \Gamma \overset{\mathsf{tactic2}}{\vdash} as}{\Gamma \vdash (n, \texttt{Tactic}(ts, as)) : \Gamma \cup \Gamma'}$$
$$\text{, falls } n \notin \mathcal{D}(\Gamma) \text{ und } wn \leq 1$$
$$\text{, wobei } \Gamma' = \{n \mapsto \texttt{Tactic}(ts, as)\}$$

tactic1

$$(33)\Gamma \vdash [] : (0, \emptyset) \qquad (34)\frac{\Gamma \vdash o : (m, r) \quad \Gamma \vdash os : (m', rs)}{\Gamma \vdash [o \,|\, os] : (m + m', r \cup rs)} \text{, falls } r \cap rs = \emptyset$$

$$(35)\Gamma \vdash \texttt{Wildcard} : (1, \emptyset) \qquad (36)\Gamma \vdash \texttt{TacName}(n) : (0, \{n\}) \text{, falls } n \in \mathcal{D}(\Gamma_{|\texttt{Rule}})$$

tactic2

$$(37)\Gamma \vdash [] \qquad (38)\frac{\Gamma \vdash os}{\Gamma \vdash [(n, n') \,|\, os]} \text{, falls } n \in \mathcal{D}(\Gamma_{|\texttt{Rule}})$$

4.5.10 Präzedenzen (prec, prec')

prec

$$(39)\Gamma \vdash [] : \Gamma \qquad (40)\ \frac{\Gamma \vdash o : (\Gamma_1, ord) \quad \Gamma_1 \vdash os : \Gamma_2}{\Gamma \vdash [o \,|\, os] : \Gamma_2}$$
$$\text{, falls } GenTran(ord) \text{ irreflexiv ist.}$$

$$(41)\ \frac{\Gamma \overset{\mathsf{prec'}}{\vdash} p_1 : o_1 \quad \ldots \quad \Gamma \overset{\mathsf{prec'}}{\vdash} p_k : o_k}{\Gamma \vdash (n, \texttt{Prec}([p_1, \ldots, p_k])) : (\Gamma \cup \Gamma', o_1 \cup \cdots \cup o_k)}$$
$$\text{, falls } n \notin \mathcal{D}(\Gamma)$$
$$\text{, wobei } \Gamma' = \{n \mapsto \texttt{Prec}([p_1, \ldots, p_k])\}$$

prec'

$(42) \Gamma \vdash [] : \{\}$ $\qquad$ $(43) \Gamma \vdash [n] : \{\}$, falls $n \in \mathcal{D}(\Gamma_{|\texttt{Constr}}) \cup \mathcal{D}(\Gamma_{|\texttt{Functor}})$

$$(44)\ \frac{\Gamma \vdash [n_2 \,|\, ns] : o_1}{\Gamma \vdash [n_1, n_2 \,|\, ns] : \{(n_1, n_2)\} \cup o_1}, \text{ falls } n_1 \in \mathcal{D}(\Gamma_{|\texttt{Constr}}) \cup \mathcal{D}(\Gamma_{|\texttt{Functor}})$$

4.5.11 Datentypen (data, data')

data

$$(45)\ \frac{(\Gamma, s) \overset{\text{data'}}{\vdash} os : (\Gamma_1, s_1, s_2)}{(\Gamma, s) \vdash os : \Gamma_1}\ , \text{ falls } s_2 \subseteq s_1$$

data'

$(46) (\Gamma, s) \vdash [] : (\Gamma, s, \emptyset)$ $\qquad$ $(47) \dfrac{(\Gamma, s) \vdash o : (\Gamma_1, s_1, s_2) \quad (\Gamma_1, s_1) \vdash os : (\Gamma_2, s_3, s_4)}{(\Gamma, s) \vdash [o \,|\, os] : (\Gamma_2, s_1 \cup s_3, s_2 \cup s_4)}$

$(48)\ (\Gamma, s) \vdash (n, \texttt{Data(Simple(}n'\texttt{))}) : (\Gamma \cup \Gamma', s \cup \{n'\}, \emptyset)$
, falls $n \notin \mathcal{D}(\Gamma)$ und $n' \notin PrimDataNames'$
, wobei $\Gamma' = \{n \mapsto \texttt{Data(Simple(}n'\texttt{))}\}$

$$(49)\ \frac{\vdash t : s_1}{(\Gamma, s) \vdash (n, \texttt{Data(Func(}t\texttt{,Simple(}n'\texttt{)))}) : (\Gamma \cup \Gamma', s \cup \{n'\}, s_1)}$$
, falls $n \notin \mathcal{D}(\Gamma)$ und $n' \notin PrimDataNames'$
, wobei $\Gamma' = \{n \mapsto \texttt{Data(Func(}t\texttt{,Simple(}n'\texttt{)))}\}$

$(50) \vdash \texttt{Simple(}n\texttt{)} : \{n\}$ $\qquad$ $(51) \vdash \texttt{TypV(}n\texttt{)} : \emptyset$

$(52) \dfrac{t : s}{\texttt{List(}t\texttt{)} : s}$ $\qquad$ $(53) \dfrac{\vdash t_1 : s_1 \quad \cdots \quad \vdash t_k : s_k}{\vdash \texttt{Tupel(}[t_1, \ldots, t_k]\texttt{)} : s_1 \cup \cdots \cup s_k}$

$$(54)\ \frac{\vdash t_1 : s_1 \quad \vdash t_2 : s_2}{\vdash \texttt{Func(}t_1\texttt{,}t_2\texttt{)} : s_1 \cup s_2}$$

4.5.12 Funktionen (function, fusynt, futyp, futyp')

function

$$(55)\ \frac{\Gamma \overset{\text{fusynt}}{\vdash} fuds : (\Gamma_1, ns) \quad (\Gamma_1, \Theta) \overset{\text{futyp}}{\vdash} fuds : \Theta_1}{(\Gamma, \Theta) \vdash fuds : (\Gamma_1, \Theta_1)}, \text{ falls } ns \subseteq \mathcal{D}(\Gamma_{1|\texttt{Function}})$$

fusynt

$$(56)\Gamma \vdash [] : (\Gamma, \emptyset) \qquad (57)\frac{\Gamma \vdash o : (\Gamma_1, ns_1) \quad \Gamma_1 \vdash os : (\Gamma_2, ns_2)}{\Gamma \vdash [o|os] : (\Gamma_2, ns_1 \cup ns_2)}$$

$$(58)\ \frac{\left[\begin{array}{cc} \Gamma \overset{\mathsf{varinpat}}{\vdash} p_i : pv_i & (pv_i', \Gamma) \overset{\mathsf{termsynt}}{\vdash} t_i : nt_i \\ (pv_i', \Gamma) \overset{\mathsf{termsynt}}{\vdash} c_i : nc_i & \left[(pv_i', \Gamma) \overset{\mathsf{termsynt}}{\vdash} t_j^i : n_j^i\right]^{1 \le j \le k_i} \end{array}\right]^{1 \le i \le m}}{\Gamma \vdash (n, \mathtt{Function}(ds)) : (\Gamma \cup \{n \mapsto \mathtt{Function}(ds)\}, \bigcup_{1 \le i \le m} ns_i)}$$

, falls $n \notin \mathcal{D}(\Gamma)$ und $pv^i \cap \{\mathtt{TV}(n_j^i) | 1 \le j \le k_i\} = \emptyset$ und $n_j^i \ne n_k^i \iff j \ne k$

, wobei $ds = [(p_i, t_i, c_i, ld_i)]_{1 \le i \le m}$
$ld_i = [(\mathtt{TV}(n_j^i), t_j^i)]_{1 \le j \le k_i}$
$ns_i = nt_i \cup nc_i \cup \bigcup_{1 \le j \le k_i} n_j^i$
$pv_i' = pv_i \cup \{\mathtt{TV}(n_j^i) | 1 \le j \le k_i\}$

futyp

$$(59)\ \frac{\left[(\Gamma, \Theta \cup \bigcup_{1 \le j < i} \Theta_j') \overset{\mathsf{futyp'}}{\vdash} K_i : \Theta_i\right]^{1 \le i \le k}}{(\Gamma, \Theta) \vdash fuds : \Theta \cup \bigcup_{1 \le i \le k} \Theta_i'}$$

, wobei $[K_1, \ldots, K_k] = ZerlegeinZHK(fuds)$
$\Theta_j' = Generalisiere(\Theta_j, \emptyset)$

futyp'

$$(60)\ \frac{(\Gamma, \Theta') \vdash d_1 \quad \cdots \quad (\Gamma, \Theta') \vdash d_k}{(\Gamma, \Theta) \vdash [d_1, \ldots, d_k] : \Theta'}$$

, wobei $d_i = (n_i, \mathtt{Function}(fd_i))$
$\Theta' = \{n_i \mapsto \tau_i | 1 \le i \le k\} \cup \Theta$

$$(61)\ \frac{\left[\begin{array}{cc} (\Gamma, \Theta) \overset{\mathsf{pattype}}{\vdash} p_i : (\tau, \Xi_i) & \left[(\Gamma, \Theta_i) \overset{\mathsf{termtype}}{\vdash} t_j^i : \tau_j^i\right]^{1 \le j \le k_i} \\ (\Gamma, \Theta_i') \overset{\mathsf{termtype}}{\vdash} t_i : \tau & (\Gamma, \Theta_i') \overset{\mathsf{termtype}}{\vdash} c_i : \mathtt{Simple("bool")} \end{array}\right]^{1 \le i \le m}}{(\Gamma, \Theta) \vdash (n, \mathtt{Function}(ds))}$$

, wobei $ds = [(p_i, t_i, c_i, lds_i)]_{1 \le i \le m}$
$lds_i = [(\mathtt{TV}(n_j^i), t_j^i)]_{1 \le j \le k_i}$
$\Theta_i = \Theta \cup \Xi_i \cup \{\mathtt{TV}(n_j^i) \mapsto \tau_j^i | 1 \le j \le k_i\}$
$\Theta_i' = Generalisiere(\Theta_i, \Xi_i)$

4.5.13 Variablen in Pattern, Syntax, Typing von Pattern und Termen (varinpat, termsynt, pattype, termtype)

varinpat

(62) $\Gamma \vdash t : \emptyset$
, falls $t \in \{\texttt{True}, \texttt{False}, \texttt{()}, \texttt{Nil}, \texttt{N}(i), \texttt{Char}(c), \texttt{FOp}(n)\}$

(63) $\Gamma \vdash \texttt{TV}(n) : \{\texttt{TV}(n)\}$

(64) $\Gamma \vdash \texttt{Var(TV}(n)\texttt{)} : \{\texttt{TV}(n)\}$

(65) $\Gamma \vdash \texttt{Op(TV}(n)\texttt{)} : \{\texttt{TV}(n)\}$

(66) $\Gamma \vdash \texttt{DOp}(n) : \emptyset$, falls $n \in \mathcal{D}(\Gamma_{|\texttt{Data}})$

(67) $$\frac{\Gamma \vdash t_1 : n_1 \quad \Gamma \vdash t_2 : n_2}{\Gamma \vdash \texttt{C}(t_1\texttt{,}t_2) : n_1 \cup n_2}$$

(68) $$\frac{\Gamma \vdash f : n_1 \quad \Gamma \vdash p : n_2}{\Gamma \vdash \texttt{App}(f\texttt{,}p) : n_1 \cup n_2}$$

(69) $$\frac{\Gamma \vdash p_1 : n_1 \quad \cdots \quad \Gamma \vdash p_k : n_k}{\Gamma \vdash \texttt{T}([p_1, \ldots, p_k]) : n_1 \cup \cdots \cup n_k}$$

termsynt

(70) $(pv, \Gamma) \vdash t : \emptyset$
, falls $t \in \{\texttt{True}, \texttt{False}, \texttt{Nil}, \texttt{Failed}, \texttt{N}(i), \texttt{Char}(c)\}$

(71) $(pv, \Gamma) \vdash \texttt{TV}(n) : \emptyset$, falls $\texttt{TV}(n) \in pv$

(72) $(pv, \Gamma) \vdash \texttt{FOp}(n) : \{n\}$

(73) $(pv, \Gamma) \vdash \texttt{DOp}(n) : \emptyset$, falls $n \in \mathcal{D}(\Gamma_{|\texttt{Data}})$

(74) $(pv, \Gamma) \vdash \texttt{UOp}(n) : \emptyset$,falls $\texttt{UOp}(n) \in \mathcal{D}(\Gamma)$

(75) $$\frac{(pv, \Gamma) \vdash t_1 : n_1 \quad \cdots \quad (pv, \Gamma) \vdash t_k : n_k}{(pv, \Gamma) \vdash \texttt{T}([t_1, \ldots, t_k]) : n_1 \cup \cdots \cup n_k}$$

(76) $$\frac{(pv, \Gamma) \vdash f : n_1 \quad (pv, \Gamma) \vdash t : n_2}{(pv, \Gamma) \vdash \texttt{App}(f\texttt{,}t) : n_1 \cup n_2}$$

(77) $$\frac{(pv, \Gamma) \vdash t_1 : n_1 \quad (pv, \Gamma) \vdash t_2 : n_2}{(pv, \Gamma) \vdash \texttt{C}(t_1\texttt{,}t_2) : n_1 \cup n_2}$$

(78) $$\frac{(pv, \Gamma) \vdash t_1 : n_1 \quad (pv, \Gamma) \vdash t'_1 : n'_1 \quad \cdots \quad (pv, \Gamma) \vdash t_k : n_k \quad (pv, \Gamma) \vdash t'_k : n'_k}{(pv, \Gamma) \vdash \texttt{Subst}([(t_1, t'_1), \ldots, (t_k, t'_k)]) : n_1 \cup n'_1 \cup \cdots \cup n_k \cup n'_k}$$

pattype

(79) $(\Gamma, \Theta \cup \{n \mapsto \tau\}) \vdash \texttt{FOp}(n) : (\tau, \emptyset)$

(80) $(\Gamma, \Theta \cup \{\texttt{TV}(n) \mapsto \tau\}) \vdash \texttt{TV}(n) : (\tau, \emptyset)$

(81) $(\Gamma, \Theta) \vdash \texttt{TV}(n) : (\tau, \{\texttt{TV}(n) \mapsto \tau\})$, falls $\texttt{TV}(n) \notin \mathcal{D}(\Theta)$

(82) $(\Gamma, \Theta) \vdash \texttt{True} : (\texttt{Simple("bool")}, \emptyset)$ (83) $(\Gamma, \Theta) \vdash \texttt{False} : (\texttt{Simple("bool")}, \emptyset)$

(84) $(\Gamma, \Theta) \vdash \texttt{N(}m\texttt{)} : (\texttt{Simple("nat")}, \emptyset)$

(85) $(\Gamma, \Theta) \vdash \texttt{Char(}c\texttt{)} : (\texttt{Simple("char")}, \emptyset)$

(86) $(\Gamma, \Theta) \vdash \texttt{Nil} : (\texttt{List(}\tau\texttt{)}, \emptyset)$

(87) $(\Gamma, \Theta) \vdash \texttt{()} : (\texttt{Unit}, \emptyset)$

(88) $$\frac{(\Gamma, \Theta) \vdash a : (\tau, \Theta_1) \quad (\Gamma, \Theta \cup \Theta_1) \vdash b : (\texttt{List(}\tau\texttt{)}, \Theta_2)}{(\Gamma, \Theta) \vdash \texttt{C(}a\texttt{,}b\texttt{)} : (\texttt{List(}\tau\texttt{)}, \Theta_1 \cup \Theta_2)}$$

(89) $$\frac{(\Gamma, \Theta) \vdash a : (\texttt{Func(}\tau_1\texttt{,}\tau_2\texttt{)}, \Theta_1) \quad (\Gamma, \Theta \cup \Theta_1) \vdash b : (\tau_1, \Theta_2)}{(\Gamma, \Theta) \vdash \texttt{App(}a\texttt{,}b\texttt{)} : (\tau_2, \Theta_1 \cup \Theta_2)}$$

(90) $$\frac{(\Gamma, \Theta) \vdash \texttt{TV(}n\texttt{)} : (\texttt{Simple("term")}, \Theta_1)}{(\Gamma, \Theta) \vdash \texttt{Op(TV(}n\texttt{))} : (\texttt{Simple("term")}, \Theta_1)}$$

(91) $$\frac{(\Gamma, \Theta) \vdash \texttt{TV(}n\texttt{)} : (\texttt{Simple("term")}, \Theta_1)}{(\Gamma, \Theta) \vdash \texttt{Var(TV(}n\texttt{))} : (\texttt{Simple("term")}, \Theta_1)}$$

(92) $(\Gamma \cup \{n \mapsto \texttt{Data(}\tau\texttt{)}\}, \Theta) \vdash \texttt{DOp(}n\texttt{)} : (Instanz(\tau), \emptyset)$

(93) $$\frac{\left[(\Gamma, \Theta \cup \bigcup_{1 \leq j < i} \Theta_j) \vdash t_i : (\tau_i, \Theta_i)\right]^{1 \leq i \leq k}}{(\Gamma, \Theta) \vdash \texttt{T(}[t_1, \ldots, t_k]\texttt{)} : (\texttt{Tupel(}[\tau_1, \ldots, \tau_k]\texttt{)}, \bigcup_{1 \leq i \leq k} \Theta_i)}$$

termtype

(94) $(\Gamma, \Theta \cup \{\texttt{TV(}n\texttt{)} \mapsto \sigma\}) \vdash \texttt{TV(}n\texttt{)} : Instanz(\sigma)$

(95) $(\Gamma, \Theta) \vdash \texttt{N(}m\texttt{)} : \texttt{Simple("nat")}$

(96) $(\Gamma, \Theta) \vdash \texttt{Char(}c\texttt{)} : \texttt{Simple("char")}$

(97) $(\Gamma, \Theta) \vdash \texttt{True} : \texttt{Simple("bool")}$ (98) $(\Gamma, \Theta) \vdash \texttt{False} : \texttt{Simple("bool")}$

(99) $(\Gamma, \Theta) \vdash \texttt{Failed} : \texttt{Simple("substitution")}$ (100) $(\Gamma, \Theta) \vdash \texttt{Nil} : \texttt{List(}\tau\texttt{)}$

(101) $(\Gamma, \Theta \cup \{n \mapsto \sigma\}) \vdash \texttt{FOp(}n\texttt{)} : Instanz(\sigma)$

(102) $(\Gamma \cup \{n \mapsto \texttt{Data(}\tau\texttt{)}\}, \Theta) \vdash \texttt{DOp(}n\texttt{)} : Instanz(\forall \vec{\alpha} : \tau)$, wobei $\vec{\alpha} = FV(\tau)$

(103) $$\frac{(\Gamma, \Theta) \vdash a : \tau \quad (\Gamma, \Theta) \vdash b : \texttt{List(}\tau\texttt{)}}{(\Gamma, \Theta) \vdash \texttt{C(}a\texttt{,}b\texttt{)} : \texttt{List(}\tau\texttt{)}}$$

(104) $(\Gamma,\Theta) \vdash \texttt{()} : \texttt{Unit}$

$$(105)\ \frac{(\Gamma,\Theta) \vdash a : \texttt{Func(Simple("arg"),Simple("term"))} \quad (\Gamma,\Theta) \vdash b : \tau}{(\Gamma,\Theta) \vdash \texttt{App(}a\texttt{,}b\texttt{)} : \texttt{Simple("term")}}$$

, falls $\tau = \texttt{Simple("term")}$
oder $\tau = \texttt{Tupel([Simple("term")}, \ldots, \texttt{Simple("term")])}$

$$(106)\ \frac{(\Gamma,\Theta) \vdash a : \texttt{Func(}\tau_1\texttt{,}\tau_2\texttt{)} \quad (\Gamma,\Theta) \vdash b : \tau_1}{(\Gamma,\Theta) \vdash \texttt{App(}a\texttt{,}b\texttt{)} : \tau_2}$$

, falls $\tau_1 \neq \texttt{Simple("arg")}$

$$(107)\ \frac{[(\Gamma,\Theta) \vdash t_i : \tau_i]^{1 \leq i \leq k}}{(\Gamma,\Theta) \vdash \texttt{T(}[t_1, \ldots, t_k]\texttt{)} : \texttt{Tupel(}[\tau_1, \ldots, \tau_k]\texttt{)}}$$

$$(108)\ \frac{[(\Gamma,\Theta) \vdash t_i : \texttt{Simple("term")} \quad (\Gamma,\Theta) \vdash t'_i : \texttt{Simple("term")}]^{1 \leq i \leq k}}{(\Gamma,\Theta) \vdash \texttt{Subst(}[(t_1, t'_1), \ldots, (t_k, t'_k)]\texttt{)} : \texttt{Simple("substitution")}}$$

(109) $(\Gamma,\Theta) \vdash \texttt{V(}n\texttt{)} : \texttt{Simple("term")}$

(110) $(\Gamma,\Theta) \vdash \texttt{UOp(}n\texttt{)} : \tau$

$$, \text{wobei } \tau = \begin{cases} \texttt{Simple("term")} \text{, falls } \exists \texttt{UOp(}n\texttt{)} \mapsto \left\{ \begin{array}{l} \texttt{Functor} \\ \texttt{Constr} \end{array} \right\} \texttt{([],}n'\texttt{)} \in \Gamma \\ \texttt{Func(Simple("arg"),Simple("term"))} \text{, falls} \\ \qquad \exists \texttt{UOp(}n\texttt{)} \mapsto \left\{ \begin{array}{l} \texttt{Functor} \\ \texttt{Constr} \end{array} \right\} \texttt{(}[n_1, \ldots, n_k]\texttt{,}n'\texttt{)} \in \Gamma \end{cases}$$

4.5.14 Strategien (strategy, stsynt, sttype)

strategy

(111) $(\Gamma,\Theta) \vdash [] : \Gamma$

$$(112)\ \frac{(\Gamma,\Theta) \vdash o : \Gamma_1 \quad (\Gamma_1,\Theta) \vdash os : \Gamma_2}{(\Gamma,\Theta) \vdash [o \,|\, os] : \Gamma_2}$$

$$(113)\ \frac{\Gamma \overset{\textsf{stsynt}}{\vdash} o : \Gamma_1 \quad (\Gamma_1,\Theta) \overset{\textsf{sttype}}{\vdash} o}{(\Gamma,\Theta) \vdash o : \Gamma_1}$$

stsynt

$$(114)\quad \frac{\begin{array}{c}[(\Gamma, pres) \vdash gt_i]^{1 \le i \le n} \quad [(\Gamma, pres) \vdash lt_i]^{1 \le i \le m} \\ (\Gamma, ns) \vdash sel \qquad [(\Gamma, ns) \vdash t_i]^{1 \le i \le k} \\ (\Gamma, ns, gns \cup lns, mrns) \vdash cntrl \\ [(\Gamma, ns) \vdash ot_i]^{1 \le i \le l} \quad [(\Gamma, ns') \vdash mro_i]^{1 \le i \le p}\end{array}}{(\Gamma \vdash (n, \texttt{Strategy}(gd, ld, sel, cntrl, lds, os, mrs)) : \Gamma \cup \Gamma'}$$

$$\begin{array}{ll}
\text{, falls} & nns \cap (gns \cup lns \cup pres) = gns \cap lns = pres \cap (gns \cup lns) = \emptyset \\
& n \notin \mathcal{D}(\Gamma) \\
\text{, wobei} & pres = \{\texttt{TV("goal")}, \texttt{TV("goals")}, \texttt{TV("trs")}, \texttt{TV("!attributes")}\} \\
& nns = \{\texttt{TV}(n_i) | 1 \le i \le k\} \\
& gns = \{\texttt{TV}(gn_i) | 1 \le i \le n\} \\
& lns = \{\texttt{TV}(ln_i) | 1 \le i \le m\} \\
& ons = \{\texttt{TV}(on_i) | 1 \le i \le l\} \\
& mrns = \{mrn_i | 1 \le i \le p\} \\
& gd = [(gn_1, gt_1), \ldots, (gn_n, gt_n)] \\
& ld = [(ln_1, lt_1), \ldots, (ln_m, lt_m)] \\
& lds = [(\texttt{TV}(n_1), t_1), \ldots, (\texttt{TV}(n_k), t_k)] \\
& os = [(on_1, ot_1), \ldots, (on_l, ot_l)] \\
& mrs = [(mrn_1, mro_1), \ldots, (mrn_p, mro_p)] \\
& ns = gns \cup lns \cup pres \cup nns \\
& ns' = gns \cup lns \cup pres \\
& \Gamma' = \{n \mapsto \texttt{Strategy}(gd, ld, sel, cntrl, lds, os, mrs)\}
\end{array}$$

(115) $(\Gamma, ns) \vdash \texttt{DOp}(n)$, falls $n \in \mathcal{D}(\Gamma_{|\texttt{Data}})$

(116) $(\Gamma, ns) \vdash \texttt{FOp}(n)$, falls $n \in \mathcal{D}(\Gamma_{|\texttt{Function}})$

(117) $(\Gamma, ns) \vdash t$
, falls $t \in \{\texttt{True}, \texttt{False}, \texttt{Nil}, \texttt{Failed}, \texttt{N}(i), \texttt{Char}(c), \texttt{()}\}$

(118) $(\Gamma, ns) \vdash \texttt{TV}(n)$, falls $\texttt{TV}(n) \in ns$

$$(119)\quad \frac{(\Gamma, ns) \vdash a \quad (\Gamma, ns) \vdash b}{(\Gamma, ns) \vdash \texttt{C}(a, b)}$$

$$(120)\quad \frac{(\Gamma, ns) \vdash a_1 \quad \cdots \quad (\Gamma, ns) \vdash a_k}{(\Gamma, ns) \vdash \texttt{T}([a_1, \ldots, a_k])}$$

$$(121)\quad \frac{(\Gamma, ns) \vdash a \quad (\Gamma, ns) \vdash b}{(\Gamma, ns) \vdash \texttt{App}(a, b)}$$

$$(122)\quad \frac{(\Gamma, ns) \vdash a_1 \quad (\Gamma, ns) \vdash a'_1 \quad \cdots \quad (\Gamma, ns) \vdash a_k \quad (\Gamma, ns) \vdash a'_k}{(\Gamma, ns) \vdash \texttt{Subst}([(a_1, a'_1), \ldots, (a_k, a'_k)])}$$

(123) $(\Gamma, ns, dns, ns') \vdash \texttt{Break}$

(124) $(\Gamma, ns, dns, ns') \vdash \texttt{Empty}$

$$(125)\ \frac{(\Gamma, ns, dns, ns') \vdash a \quad (\Gamma, ns, dns, ns') \vdash b}{(\Gamma, ns, dns, ns') \vdash \texttt{Seq(}a\texttt{,}b\texttt{)}}$$

$$(126)\ \frac{(\Gamma, ns, dns, ns') \vdash a \quad (\Gamma, ns, dns, ns') \vdash b}{(\Gamma, ns, dns, ns') \vdash \texttt{Else(}a\texttt{,}b\texttt{)}}$$

$$(127)\ \frac{(\Gamma, ns) \vdash cond \quad (\Gamma, ns, dns, ns') \vdash c}{(\Gamma, ns, dns, ns') \vdash \texttt{If(}cond\texttt{,}a\texttt{)}}$$

$$(128)\ \frac{(\Gamma, ns) \vdash t}{(\Gamma, ns, dns, ns') \vdash \texttt{Call(}n\texttt{,}t\texttt{)}}, \text{ falls } n \in ns'$$

$$(129)\ \frac{\Gamma \overset{\mathsf{varinpat}}{\vdash} p : pv \quad (\Gamma, ns) \vdash t \quad (\Gamma, ns \cup pv, dns, ns') \vdash c}{(\Gamma, ns, dns, ns') \vdash \texttt{For(}p\texttt{,}t\texttt{,}c\texttt{)}}$$

$$(130)\ \frac{(\Gamma, ns, dns, ns') \vdash c \quad [(\Gamma, ns) \vdash t_i]^{1 \le i \le k}}{(\Gamma, ns, dns, ns') \vdash \texttt{Assign(}c\texttt{,}[(n_i, t_i)]_{1 \le i \le k}\texttt{)}}, \text{ falls } \forall_i : \texttt{TV(}n_i\texttt{)} \in dns$$

$$(131)\ \frac{\begin{array}{c} \Gamma \overset{\mathsf{varinpat}}{\vdash} p : pv \quad \Gamma \overset{\mathsf{varinpat}}{\vdash} gp : gpv \\ (\Gamma, ns') \vdash cond \quad (\Gamma, ns') \vdash res \\ [(\Gamma, ns') \vdash at_i]^{1 \le i \le n} \\ (\Gamma, ns') \vdash ot \\ [(\Gamma, ns') \vdash t_i]^{1 \le i \le m} \end{array}}{(\Gamma, ns) \vdash \texttt{MetaRule(}p\texttt{,}cond\texttt{,}gp\texttt{,}res\texttt{,}as\texttt{,}ot\texttt{,}lds\texttt{)}}$$

$$\begin{array}{ll} \text{, falls} & pv \cap gpv = \emptyset,\ lns \cap (pv \cup gpv \cup ns) = \emptyset \\ & ns \cap (pv \cup gpv) = \emptyset \\ & \{\texttt{TV(}an_i\texttt{)} | 1 \le i \le n\} \subseteq ns \\ \text{, wobei} & as = [(an_1, at_1), \ldots, (an_n, at_n)] \\ & lds = [(\texttt{TV(}n_1\texttt{)}, t_1), \ldots, (\texttt{TV(}n_m\texttt{)}, t_m)] \\ & lns = \{\texttt{TV(}n_i\texttt{)} | 1 \le i \le m\} \\ & ns' = ns \cup pv \cup gpv \cup lns \end{array}$$

sttype

$$\frac{\begin{array}{cc}\left[(\Gamma,\Theta_1)\overset{\textsf{termtype}}{\vdash} t_i : \tau_i\right]^{1\leq i\leq k} & \left[(\Gamma,\Theta_2)\overset{\textsf{termtype}}{\vdash} gt_i : \tau_i^g\right]^{1\leq i\leq n}\\ \left[(\Gamma,\Theta_2)\overset{\textsf{termtype}}{\vdash} lt_i : \tau_i^l\right]^{1\leq i\leq m} & (\Gamma,\Theta_3)\overset{\textsf{termtype}}{\vdash} sel : \tau^{sel}\\ \left[(\Gamma,\Theta_3)\overset{\textsf{termtype}}{\vdash} ot_i : \tau_i^o\right]^{1\leq i\leq l} & \left[(\Gamma,\Theta_4)\vdash mro_i : \tau_i^m\right]^{1\leq i\leq p}\\ \multicolumn{2}{c}{(\Gamma,\Theta_3,\Theta_5)\vdash cntrl}\end{array}}{(\Gamma,\Theta)\vdash (n, \texttt{Strategy(}gs\texttt{,}ls\texttt{,}sel\texttt{,}cntrl\texttt{,}lds\texttt{,}os\texttt{,}mrs\texttt{))}} \tag{132}$$

, wobei
$$\begin{aligned}
gs &= [(gn_1, gt_1), \ldots, (gn_n, gt_n)]\\
ls &= [(ln_1, lt_1), \ldots, (ln_m, lt_m)]\\
lds &= [(\texttt{TV(}n_1\texttt{)}, t_1), \ldots, (\texttt{TV(}n_k\texttt{)}, t_k)]\\
os &= [(on_1, ot_1), \ldots, (on_l, ot_l)]\\
mrs &= [(mrn_1, mro_1), \ldots, (mrn_p, mro_p)]\\
\Theta_1 &= \Theta \cup \{\texttt{TV(}n_i\texttt{)} \mapsto \tau_i | 1 \leq i \leq k\} \cup\\
&\quad \{\texttt{TV("goal")} \mapsto \texttt{Simple("goal")},\\
&\quad \texttt{TV("goals")} \mapsto \texttt{List(Simple("goals"))},\\
&\quad \texttt{TV("trs")} \mapsto [([\texttt{"Char"}], \texttt{"Rule"})],\\
&\quad \texttt{TV("!attributes")} \mapsto [([\texttt{"Char"}], [[\texttt{"Char"}]])]\}\\
\Theta_2 &= Generalisiere(\Theta_1, \emptyset)\\
\Theta_3 &= \Theta_2 \cup Generalisiere(\{\texttt{TV(}gn_i\texttt{)} \mapsto \tau_i^g | 1 \leq i \leq n\} \cup\\
&\qquad \{\texttt{TV(}ln_i\texttt{)} \mapsto \tau_i^l | 1 \leq i \leq m\}, \emptyset)\\
\Theta_4 &= \Theta_3 \setminus \{\texttt{TV(}n_i\texttt{)} \mapsto \sigma_i | 1 \leq i \leq k\}\\
\Theta_5 &= Generalisiere(\{mrn_i \mapsto \tau_i^m | 1 \leq i \leq p\}, \emptyset)\\
\tau^{sel} &= \texttt{Func(List(Simple("goal")),}\\
&\quad \texttt{Tupel(}[\texttt{Simple("goal")}, \texttt{List(Simple("goal"))}]\texttt{))}
\end{aligned}$$

$$\frac{\begin{array}{c}(\Gamma,\Theta)\overset{\textsf{pattype}}{\vdash} p : (\tau^p, \Theta^p)\\ (\Gamma,\Theta\cup\Theta^p)\overset{\textsf{pattype}}{\vdash} gp : (\texttt{Simple("Goal")}, \Theta^{gp})\\ \left[(\Gamma,\Theta_1)\overset{\textsf{termtype}}{\vdash} t_i : \tau_i\right]^{1\leq i\leq k}\\ (\Gamma,\Theta_2)\overset{\textsf{termtype}}{\vdash} cond : \texttt{Simple("Bool")}\\ (\Gamma,\Theta_2)\overset{\textsf{termtype}}{\vdash} res : \texttt{List(Simple("Goal"))}\\ (\Gamma,\Theta_2)\overset{\textsf{termtype}}{\vdash} ot : \tau^{ot}\\ \left[(\Gamma,\Theta_2)\overset{\textsf{termtype}}{\vdash} at_i : Instanz(\Theta(\texttt{TV(}an_i\texttt{)}))\right]^{1\leq i\leq m}\end{array}}{(\Gamma,\Theta)\vdash \texttt{MetaRule(}p\texttt{,}cond\texttt{,}gp\texttt{,}res\texttt{,}as\texttt{,}ot\texttt{,}lds\texttt{)} : restyp} \tag{133}$$

, wobei
$$\begin{aligned}
\Theta_1 &= \Theta \cup \Theta^p \cup \Theta^{gp} \cup \Theta^l\\
\Theta^l &= \{\texttt{TV(}n_i\texttt{)} \mapsto \tau_i | 1 \leq i \leq k\}\\
\Theta_2 &= Generalisiere(\Theta_1, \Theta^p \cup \Theta^{gp})\\
as &= [(an_1, at_1), \ldots, (an_m, at_m)]\\
lds &= [(\texttt{TV(}n_1\texttt{)}, t_1), \ldots, (\texttt{TV(}n_k\texttt{)}, t_k)]\\
restyp &= Generalisiere(\{\tau^p\}, \emptyset)
\end{aligned}$$

$$(134)\quad (\Gamma,\Theta,\Theta_1) \vdash \texttt{Break}$$

$$(135)\quad \frac{(\Gamma,\Theta,\Theta_1) \vdash c_1 \quad (\Gamma,\Theta,\Theta_1) \vdash c_2}{(\Gamma,\Theta,\Theta_1) \vdash \texttt{Else}(c_1,c_2)}$$

$$(136)\quad \frac{(\Gamma,\Theta,\Theta_1) \vdash c_1 \quad (\Gamma,\Theta,\Theta_1) \vdash c_2}{(\Gamma,\Theta,\Theta_1) \vdash \texttt{Seq}(c_1,c_2)}$$

$$(137)\quad (\Gamma,\Theta,\Theta_1) \vdash \texttt{Empty}$$

$$(138)\quad \frac{(\Gamma,\Theta) \overset{\mathsf{termtype}}{\vdash} t : Instanz(\Theta_1(n))}{(\Gamma,\Theta,\Theta_1) \vdash \texttt{Call}(n,t)}$$

$$(139)\quad \frac{(\Gamma,\Theta) \overset{\mathsf{termtype}}{\vdash} cond : \texttt{Simple("Bool")} \quad (\Gamma,\Theta,\Theta_1) \vdash c}{(\Gamma,\Theta,\Theta_1) \vdash \texttt{If}(cond,c)}$$

$$(140)\quad \frac{\begin{array}{c}(\Gamma,\Theta,\Theta_1) \vdash c \\ \left[(\Gamma,\Theta) \overset{\mathsf{termtype}}{\vdash} t_i : Instanz(\Theta(\texttt{TV}(n_i)))\right]^{1 \leq i \leq k}\end{array}}{(\Gamma,\Theta,\Theta_1) \vdash \texttt{Assign}(c,[(n_1,t_1),\ldots,(n_k,t_k)])}$$

$$(141)\quad \frac{(\Gamma,\Theta) \overset{\mathsf{pattype}}{\vdash} p : (\tau^p,\Theta^p) \quad (\Gamma,\Theta) \overset{\mathsf{termtype}}{\vdash} t : \texttt{List}(\tau^p) \quad (\Gamma,\Theta_2,\Theta_1) \vdash c}{(\Gamma,\Theta,\Theta_1) \vdash \texttt{For}(p,t,c)}$$

, wobei $\Theta_2 = \Theta \oplus \Theta^p$

4.6 Kommentar zur statischen Semantik

4.6.1 prog

Diese Regel inferiert die statische Korrektheit für alle Komponenten eines Programmes. Die Trennung zwischen Benutzerobjekten (Sorten, Constructoren, Regeln, etc.) und den Datenconstructoren, Funktionen und Strategien[12] findet sich auch in dieser Regel wieder. Die statische Korrektheit der beiden Objektklassen wird unabhängig voneinander abgeleitet: Die Umgebung Γ_9, welche die Ableitung der Korrektheit der Metaobjekte einleitet, enthält keine der Benutzerobjekte. In jeder Klasse werden jedoch die Abhängigkeiten innerhalb der Klassen berücksichtigt. Da z.B. Functordefinitionen von den Sortendefinitionen abhängen, werden sie erst geprüft, wenn die Prüfung der Sorten gelungen ist[13]. Die Klauseln für die Benutzerobjekte haben alle die Form $\Gamma \vdash objs : \Gamma'$. Die Intention dabei ist: In der Umgebung Γ ergibt eine erfolgreiche Prüfung der Objekte $objs$ eine neue Umgebung Γ'. Die Umgebung enthält dabei die Bindungen aller bisher geprüften Objekte in der Form $n \mapsto o$, wobei n ein Name und o ein Objekt ist.
Die Prüfung der Metaobjekte ist komplizierter, als die der Benutzerobjekte. Zuerst werden die Datenconstructordefinitionen überprüft, dann die Funktionsdefinitionen und zum Schluß die Strategiedefinitionen. Die Funktions- und Strategiedefinitionen benötigen neben den Bindungen der Namen an Objekte noch eine Typ-Umgebung, in welcher Bindungen von Namen an Typen protokolliert werden.
Die einzige Bedingung der Regel stellt sicher, daß die Ordnung, welche durch die Sortendefinitionen beschrieben wird, irreflexiv ist. Die Funktion $GenOrdering(\Gamma)$ berechnet aus den Bindungen der Sortennamen an Sortenobjekte die zugehörige Ordnung.
Die Funktion $GenTran(o)$ liefert den transitiven Abschluß einer Ordnung, so daß die Bedingung schließlich sicherstellt, daß die so definierte Relation $<_\Sigma$ (siehe Abschnitt 3) irreflexiv ist.

4.6.2 sorts

Die ersten beiden Regeln definieren die Abarbeitung der Objektliste, die von links nach rechts erfolgt. Man beachte die Notation `[head|tail]` für eine Liste mit Kopfelement head und Restliste tail. Die dritte Regel (4) behandelt den Fall einer einfachen Sortendefinition. Die Umgebung wird um die Bindung des Namens n an das Sortenobjekt erweitert. Spätere Regeln werden eine solche Erweiterung nur zulassen, wenn der entsprechende Name noch nicht in der Umgebung gebunden ist. Da dies hier nicht gefordert wird, kann ein Programm mehrere Definitionen derselben Sorte enthalten.
Die letzte Regel (5) behandelt den Fall der Sorteninklusion. Die Bedingung $n' \in \mathcal{D}(\Gamma_{\texttt{Sorts}})$ fordert, daß die Obersorte bereits definiert wurde. Die Umgebung wird um zwei Bindun-

[12] Welche man auch als Metaobjekte bezeichnen kann.

[13] Obwohl die Reihenfolge von Klauseln in einer Inferenzregel keine Bedeutung hat, ergibt sich eine Reihenfolge dadurch, daß Klauseln von Variablen aus anderen Klauseln abhängen, etwa einer inferierten Umgebung.

gen erweitert, einmal um die Bindung des Namens an ein einfaches Sortenobjekt `Sorts`, andererseits um die Bindung desselben Namens an ein Subsortenobjekt. Dies ist der einzige Fall, in dem mehrere Bindungen für denselben Namen in der Umgebung auftauchen können.
Man beachte, daß diese Regel zur Folge hat, daß in jedem $\mathcal{QDL}$ Programm die Obersorte vor der Definition einer Sorteninklusion bereits definiert worden sein muß: `sorts a<b; b; .` ist kein zulässiges Programm.

4.6.3 cons, functor

Diese beiden Fälle unterscheiden sich kaum. Die Bedingung in der jeweils dritten Regel stellt sicher, daß die vorkommenden Namen an Sortenobjekte gebunden sind.

4.6.4 rule

Die Regeln (14/15) behandeln die beiden Fälle, wie eine Regeldefinition aussehen kann. Zuerst wird der Fall behandelt, daß die linke Seite der Regel aus einer Konstanten besteht, dann der Fall eines Terms mit mehreren Argumenten.
Die Bedingung $n' \in \mathcal{D}(\Gamma_{|\texttt{Functor}})$ stellt sicher, daß die Wurzel der linken Seite ein Functor ist. Die Überprüfung der linken und rechten Seite geschieht mittels des Inferenzschematas sort-of. Falls $(\Gamma, \Theta) \overset{\textsf{sort-of}}{\vdash} t : (s, \Theta')$, dann enthält der Term t nur bereits bekannte Operatorsymbole und ist wohlgetypt bzgl. der Signatur, die durch die Sorten-, Constructor- und Functordefinitionen beschrieben wird. Θ enthält die Bindungen für die Variablen. s ist der Name der Sorte, die den Typ dieses Terms darstellt. Durch das Auftreten derselben Umgebung Θ_1 am Ende der ersten und zweiten Klausel ist sichergestellt, daß durch die Überprüfung der rechten Regelseite keine neuen Bindungen von Variablen entstehen können. Als Folge davon können in der rechten Regelseite keine Variablen auftauchen, die nicht schon in der linken Seite vorgekommen sind.
Die Überprüfung, ob $sort(r) \leq_\Sigma sort(l)$ für eine Regel $l \to r$ gilt, geschieht mittels $\Gamma \overset{\textsf{sort-lt}}{\vdash} s_1 : s_2$. Ist dies nämlich ableitbar, dann gilt eben gerade $s_2 \leq_\Sigma s_1$.

4.6.5 goal

Diese Inferenzregeln ähneln denen für rule, mit den Unterschieden, daß keine Beschränkung über das Auftauchen von Variablen existiert und die linke und rechte Gleichungsseite einfach nur eine gemeinsame Obersorte haben müssen, also falls s_1, s_2 die Sorten der linken, bzw. rechten Gleichungsseite sind, dann muß entweder $s_1 \leq_\Sigma s_2$ oder $s_2 \leq_\Sigma s_1$ gelten. Die beiden letzten Regeln prüfen jeweils einen dieser Fälle.

4.6.6 sort-lt, sort-of

Regel (20) behandelt den Fall der Reflexivität. Die nächste Regel trifft, wenn es eine direkte Subsortenbeziehung gibt, d.h., falls das Programm eine Definition der Form $s_2 \mathtt{<} s_1$

enthielt. Die letzte Regel schließlich berücksichtigt die Transitivität der Sorteninklusion.
Regel (23) behandelt den Fall, daß eine Variable mit schon bekanntem Typ angetroffen wird. Die nächste Regel nimmt eine unbekannte Variable in die Typumgebung auf und bindet sie an einen Sortennamen. Die anschließende Regel behandelt Konstanten. Die letzte Regel schließlich behandelt Operatorterme mit Argumenten. Zuerst werden die Sorten der Argumentterme hergeleitet, dann wird überprüft, daß diese mit den in der Definition des Wurzeloperators gegebenen Sorten verträglich sind, d.h. Untersorten davon sind.

4.6.7 trs

Die letzte Regel stellt einfach sicher, daß alle auftretenden Namen an Regelobjekte gebunden sind.

4.6.8 tactic, tactic1, tactic2

Regel (32) prüft zuerst die Ordnung der Taktik: $\Gamma \overset{\mathsf{tactic1}}{\vdash} ts : (wn, rs)$, falls die Ordnung auf den Regeln (ts) wn Wildcards enthält und rs die Regelnamen enthält, die in der Ordnung vorkommen. Die Bedingung der Inferenzregel garantiert, daß höchstens ein Wildcard in der Ordnung vorkommt. Als zweites werden die Attributzuordnungen der Taktik überprüft.
Die Regel (33) behandelt den Fall einer leeren Ordnung, die folgende Regel den Fall einer nicht leeren. Die Bedingung $r \cap rs = \emptyset$ stellt sicher, daß ein Regelname höchstens einmal in der Ordnung auftaucht. Die dritte/vierte Regel behandeln die Elemente einer Ordnung. Die Bedingung der vierten Regel stellt sicher, daß der vorkommende Name auch wirklich ein Regelname ist.
Die Bedingung bei Regel (38) garantiert, daß der erste Name in einem Attributpaar der Name einer Regel ist.

4.6.9 prec, prec'

Die Regel (40) unterscheidet sich von den bisher vorkommenden Regeln zur Abarbeitung der Objektliste. Hier wird zusätzlich geprüft, ob die transitive Hülle der durch die Präzedenz beschriebenen Ordnung *ord* irreflexiv ist. Die nächste Regel leitet für jede Kette p_i eine Ordnung o_i ab und gibt die Vereinigung dieser Ordnungen zurück.
Regel (42) und die beiden folgenden Regel leiten aus einer Kette eine Ordnung ab, wobei jeweils geprüft wird, ob die auftretenden Namen zu einem Constructor oder Functor gehören.

4.6.10 data, data'

Regel (45) überprüft die Objektliste os: Wenn $(\Gamma, s) \overset{\mathsf{data'}}{\vdash} os : (\Gamma_1, s_1, s_2)$, dann enthält Γ_1 die Bindungen der Objekte an die Namen, s_1 enthält die Namen, welche als Ziel einer Datenconstructordefinition auftauchen, s_2 die Namen, die als Argumente auftauchen. Die

Bedingung stellt sicher, daß alle als Argument auftauchenden Namen auch als Ziel auftauchen. s sind vorgegebene Zielnamen, es gilt: $s \subseteq s_1$.
Die 9 Regeln ab Regel (46) behandeln die Definitionen. Die ersten beiden Regeln arbeiten die Liste der Definitionen ab und sorgen für die Propagation der Typnamen. Die folgenden beiden Regeln behandeln die beiden möglichen Formen eines Datenconstructorobjektes: Zuerst eine Konstante, dann ein Constructor mit Argumenten. Die Bedingung $n' \notin PrimDataNames'$ stellt sicher, daß der Benutzer keine Constructoren zu einem primitiven Typ definieren kann[14].
Die 5 Regeln ab (50) liefern die vorkommenden Typnamen in einem Typ.

4.6.11 function, fusynt, futyp, futyp'

Da Funktionsdefinitionen voneinander abhängen können, werden sie zusammen überprüft. Zuerst findet die syntaktische Überprüfung statt: $\Gamma \overset{\mathsf{fusynt}}{\vdash} fuds : (\Gamma_1, ns)$. Die Menge ns enthält die Namen aller Funktionen, die in den Definitionen aufgerufen werden. Die Bedingung $ns \subseteq \mathcal{D}(\Gamma_{|\mathtt{Function}})$ stellt sicher, daß alle benutzten Funktionen auch definiert sind. Nach der syntaktischen Prüfung erfolgt das Typechecking der Funktionsdefinitionen. Regel (58) prüft eine einzelne Definition. Für alle Gleichungen zu der Funktion werden zuerst die in dem Pattern p_i eingeführten Variablen abgeleitet: $\Gamma \overset{\mathsf{varinpat}}{\vdash} p_i : pv_i$. Dann wird die rechte Seite, die Bedingung der Regel und die rechten Seiten der lokalen Definitionen überprüft. Die Bedingung der Inferenzregel stellt sicher, daß die Variablen in dem Pattern und die durch die lokalen Definitionen eingeführten Variablen verschieden sind.
Regel (59) bedarf ausführlicher Erklärung. Das Typsystem von Quetzal ist dasselbe, das allen[15] funktionalen Sprachen zugrundeliegt[16]. Funktionen können generische Typen haben. D.h. sie können Argumente verschiedener Typen haben, ein Beispiel dafür ist die Listenkonkatenation `"++"`. Die Wohlgetyptheit der Funktionen garantiert dabei, daß Funktionen nie auf Argumente eines unpassenden Typs angewandt werden[17], so ist etwa der Term `1++[]` nicht wohlgetypt. Im Typsystem von Milner unterscheidet man generische Typen und spezielle Typen. Generische Typen haben die Form $\forall \vec{\alpha}.\tau$, wobei alle Typvariablen in $\vec{\alpha}$ instantiiert werden können, um einen speziellen Typ τ' zu gewinnen. Im Unterschied zu generischen Typen können spezielle Typen nicht geändert (instantiiert) werden, außer durch Generalisierung. Generalisierung bedeutet, daß alle Typvariablen in einem Typ τ quantifiziert werden, die nicht frei in anderen Typen auftreten.
Um die Wohlgetyptheit von Funktionsdefinitionen, die voneinander abhängen, zu gewährleisten werden in solchen Definitionen den Funktionen spezielle Typen zugeordnet. Nachdem alle Funktionsdefinitionen getypt wurden, werden diese Typen generalisiert. Lokale Definitionen in Funktionen sind implizit rekursiv voneinander abhängig und werden deshalb ähnlich getypt.

[14] Mit Ausnahme von `"rule"` und `"goal"`.
[15] Oder doch den meisten.
[16] Und zwar das von Milner [Mil78]
[17] "well-typed terms can't go wrong"

Dies fordert, daß man die Funktionsdefinitionen anhand ihrer Abhängigkeiten sortiert und den Typcheck mit den Funktionen beginnt, die nur von primitiven Funktionen abhängen. Man erreicht das, indem man die Funktionsdefinitionen in 'starke Zusammenhangskomponenten' sortiert, siehe [PJL92, Abschnitt 6.8]. Regel (59) leistet gerade dies. Die Funktion $ZerlegeinZHK$ nimmt die Liste der Funktionsdefinitionen und gibt eine Liste von Funtionsdefinitionslisten $[K_1, \ldots, K_k]$ zurück. K_i enthält dabei alle Funktionen, die nur von primitiven Funktionen oder Funktionen in $K_j, j \leq i$ abhängen. Die Definitionen in K_i werden zusammen geprüft, mit den Typbindungen Θ_i als Ergebnis. Diese Bindungen werden danach generalisiert und zwar bezüglich einer leeren Umgebung, da die Typbindungen in Θ_i keine Typvariablen beinhalten, die woanders frei vorkommen[18].
Regel (60) weist jeder Regel einen speziellen Typ τ_i zu. Das Typchecking geschieht immer in einer Umgebung, die bereits Typzuweisungen für die zu prüfenden Funktionen enthält. Regel (61) nun prüft eine einzelne Funktionsdefinition. Zuerst wird das Pattern jeder Gleichung geprüft. Ξ_i enthält die Typbindungen für die Variablen in dem Pattern p_i. der Typ des Pattern muß der Typ der Funktion sein und für alle Pattern derselbe: τ. Danach werden die rechten Seiten der lokalen Definitionen geprüft, wobei den Variablen der linken Seiten spezielle Typen τ_j^i zugeordnet werden, die natürlich mit den Typen der korrespondierenden rechten Seiten übereinstimmen müssen. Nachdem alle lokalen Definitionen getypt sind, wird die Typumgebung generalisiert und zwar im Bezug auf die Variablen, die im Pattern vorkommen. Diesen wird während der gesamten Definition nur ein spezieller Typ zugeordnet. Die Prüfung der rechten Seite der Gleichung, die denselben Typ, wie das Pattern haben muß, und der Bedingung der Regel bergen dann keine Überraschungen mehr.

4.6.12 varinpat, termsynt, pattype, termtype

Die Regeln für **varinpat**, **termsynt** sollten klar sein. Die Regeln (79)-(93) für **pattype** typen ein Pattern. $(\Gamma, \Theta) \vdash p : (\tau, \Theta')$ bedeutet, daß das Pattern p unter den Bindungen Γ und den Typbindungen Θ den (speziellen) Typ τ hat und zusätzliche Typbindungen Θ' impliziert.
Die zweite und dritte Regel behandeln Variablen, einmal, wenn schon eine Tybindung für die Variable vorliegt und wenn eine Typbindung erst eingeführt werden muß. Die Typregeln sollten an sich klar sein, so daß nur noch einige spezielle Fälle kommentiert werden müssen. Regel (87) behandelt das spezielle Pattern `()`, das bei Metaregeldefinitionen auftreten kann. Regeln (90/91) behandeln den Fall eines Usertermpatterns. Regel (92) behandelt Datenconstructoren. Im Gegensatz zu Funktionen darf jedes Auftreten eines Datenconstructoren einen Typ haben, der eine Instanz des generischen Typs des Constructors ist.
Das Typing eines Terms wird von den Regeln (94)-(110) behandelt. Die erste Regel behandelt Variablen. Im Gegensatz zu einem Pattern ist der Typ einer Variablen hier eine Instanz ihres (generischen) Typs σ. Da spezielle Typen generischen Typen ohne Quantifizierung entsprechen, ändert eine Instantiierung an einem speziellen Typ nichts. Ähnliches gilt für

[18] Da Θ anfangs nur generische Typen enthält.

Funktionen in Regel (101). Der spezielle Term `()` kann in Aufrufen einer Metaregel auftreten, falls diese keine Argument hat, Regel (104). Regel (105) bedarf einiger Erklärung. Da Userterme über einer Signatur auch von Funktionen verarbeitet werden können, muß ein Weg gefunden werden, ihren Typ, der erst zur Aufrufzeit bekannt ist, herzuleiten. Quetzal macht einfach den Ansatz, daß alle Userterme den Typ `"term"` haben. Jetzt tritt das Problem auf, die Anwendung eines (User)Operators auf Terme zu typen. Da wir nicht wissen können, welcher Useroperator (zur Laufzeit) angewandt werden wird, können wir auch die Anzahl seiner Argumente nicht kennen. Wir wissen nur, daß sie alle den Typ `"term"` haben müssen. Dafür wird der Typ `"arg"` eingeführt. Er steht für ein Tupel von Termen des Typs `"term"`, ohne das man die Anzahl der Terme festlegen müsste. Diese Regel nun behandelt gerade den Fall, daß ein Useroperator[19] auf ein Tupel unbekannter Länge angewandt wird. Die Regel (109) behandelt das Auftreten einer Variable in einem Term. Dies kann nur passieren, wenn ein Term geprüft wird, der durch die Aktion eval ausgewertet werden soll, da Variablen sonst nicht in Termen auftreten können. Die folgende Regel behandelt in der gleichen Situation Useroperatoren. Dabei ist einmal der Fall einer Konstanten und der Fall eines Operators mit Argumenten zu unterscheiden.

4.6.13 strategy, stsynt, sttype

Regel (113) prüft zuerst die Syntax, dann das Typing einer Strategiedefinition.
Die Überprüfung der Syntax geschieht mit den Regeln (114) bis (131). Die erste Regel prüft für ein Strategieobjekt nacheinander die Terme der Datendefinitionen, des Auswahlterms, der rechten Seiten der lokalen Definitionen, die Ablaufsteuerung, die Ausgabeterme und die Metaregeln. Falls $(\Gamma, ns) \vdash t$, dann ist der Term t korrekt, wobei die Variablen in ns bekannt sind. Die Form der Regel legt fest, das die globalen/lokalen Datennamen in den Initialisierungstermen der Daten (lt_i, gt_i) unbekannt sind, diese sich also nicht darauf beziehen können. Man beachte ebenfalls, daß die lokalen Definitionen in den Metaregeln und Dateninitialisierungen *nicht* sichtbar sind. Die Bedingungen der Regel stellen sicher, daß die lokalen Definitionen, die globalen, lokalen Daten und die vordefinierten Namen $pres$ nicht kollidieren.
Die Regeln (115) bis (122) prüfen Terme und sollten keine Überraschung darstellen. Die Überprüfung der Ablaufsteuerung geschieht mittels der Regeln (123) bis (130). $(\Gamma, ns, ds, ns') \vdash c$ bedeutet dabei, daß c unter der Umgebung Γ und den Variablen ns wohlgetypt ist. $ds \subseteq ns$ enthält die Datennamen, die nur in Regel (130) zur Prüfung der Zuweisung benötigt werden. ns' enthält die Namen der Metaregeln.
Die Regel (131) dient zur Prüfung einer Metaregel. Zuerst werden die Variablen des Regelpatterns und des Goalpatterns (p, gp) abgeleitet. Danach werden die Terme, die in der Metaregel vorkommen, geprüft. Die Bedingungen der Regel garantieren, daß die Pattern, die lokalen Definitionen und die Datennamen (aus ns) keine gemeinsame Namen definieren.
Die Typprüfung einer Strategie wird von den Regeln (132) bis (141) gehandhabt. Regel (132) typt zuerst die rechten Seiten der lokalen Definitionen, und zwar in einer Typum-

[19] Der den Typ `"arg"` $\rightarrowtail$ `"term"` hat.

gebung, die bereits Bindungen für die linken Seiten der lokalen Definitionen enthält[20]. Zusätzlich enthält die Umgebung noch Typbindungen für die vordefinierten Bezeichner. Nach dem Typing der lokalen Definitionen werden deren Typen generalisiert. Mit dieser neuen Typumgebung Θ_2 werden dann die Initalisierungsterme der globalen und lokalen Daten gt_i, lt_i getypt. Die Typen der Daten werden dann ebenfalls generalisiert. Danach erfolgt das Typen des Auswahlterms sel, der den gegebenen Typ haben muß. Danach werden die Ausgabeterme getypt. Anschließend erfolgt das Typen der Metaregeln, wobei aus der Typumgebung die Bindungen der lokalen Definitionen entfernt werden. Zum Abschluß wird dann noch die Wohlgetyptheit der Ablaufsteuerung überprüft.

Regel (133) prüft eine Metaregel. Zuerst wird das Regel- und das Goalpattern getypt, was Typbindungen Θ^p, Θ^{gp} für die Variablen in den Pattern liefert. Der Typ der Metaregel ist der Typ des Regelpatterns. Das Goalpattern muß natürlich den Typ `"goal"` haben. Danach werden, wie gehabt, die rechten Seiten der lokalen Definitionen getypt. Die Typen der lokalen Definitionen werden generalisiert, wobei natürlich Typvariablen, die in den Pattern vorkommen nicht generalisiert werden dürfen (Θ_2). Daraufhin werden die Bedingung der Metaregel ($cond$), das Ergebnis und der Ausgabeterm getypt. Zum Abschluß werden die rechten Seiten der Zuweisungen getypt. Deren Typ muß eine Instanz des Typs des zugehörigen Datums sein. Der Typ der Metaregel $restyp$ ist eine Verallgemeinerung des Typs des Regelpatterns (τ^p).

Regeln (134)-(141) typen die Ablaufsteuerung. Θ_1 enthält dabei die Typbindungen der Metaregeln. In den Regeln (138/140) muß der Typ eine Instanz des entsprechenden Typs für die Metaregel/das Datum sein. Die letzte Regel (141) leitet für das Pattern p erst einen Typ und neue Typbindungen ab, der Term muß dann den Typ $[\tau^p]$ haben. In der Ablaufsteuerung c sind dann die Typbindungen der Variablen aus p wirksam.

4.7 Dynamische Semantik

4.7.1 Auswerten von Ausdrücken zu Normalform: **NF**

$$(142)\quad \frac{\Delta \overset{\mathsf{HNF}}{\vdash} t \Rightarrow t' : \Delta_1}{\Delta \vdash t \Rightarrow t' : \Delta_1}$$

, falls $t' \in \{\mathtt{N}(i), \mathtt{Char}(c), \mathtt{True}, \mathtt{False}, \mathtt{Nil}, \mathtt{Failed}, \mathtt{UOp}(n), \mathtt{DOp}(n), \mathtt{V}(n)\}$

$$(143)\quad \frac{\Delta \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \mathtt{C}(t_1,t_2) : \Delta_1 \quad \Delta_1 \vdash t_1 \Rightarrow t_1' : \Delta_2 \quad \Delta_2 \vdash t_2 \Rightarrow t_2' : \Delta_3}{\Delta \vdash t \Rightarrow \mathtt{C}(t_1',t_2') : \Delta_3}$$

$$(144)\quad \frac{\begin{array}{c}\Delta \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \mathtt{Subst}([(t_1,t_1'),\ldots,(t_k,t_k')]) : \Delta_1 \\ [\Delta_i \vdash t_i \Rightarrow \mathtt{V}(n_i) : \Delta_i' \;\; \Delta_i' \vdash t_i' \Rightarrow u_i' : \Delta_{i+1}]^{1\le i\le k}\end{array}}{\Delta \vdash t \Rightarrow \mathtt{Subst}([(\mathtt{V}(n_1),u_1'),\ldots,(\mathtt{V}(n_k),u_k')]) : \Delta_{k+1}}$$

, falls $\forall_{i\neq j} : n_i \neq n_j$

[20]Siehe futyp

$$(145)\quad \frac{\Delta \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \mathtt{T}([t_1,\ldots,t_k]) : \Delta_1 \quad [\Delta_i \vdash t_i \Rightarrow t'_i : \Delta_{i+1}]^{1\leq i\leq k}}{\Delta \vdash t \Rightarrow \mathtt{T}([t'_1,\ldots,t'_k]) : \Delta_{k+1}}$$

$$(146)\quad \frac{\Delta \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \mathtt{App(DOp(}n\mathtt{)},ts\mathtt{)} : \Delta_1 \quad \Delta_1 \vdash ts \Rightarrow ts' : \Delta_2}{\Delta \vdash t \Rightarrow \mathtt{App(DOp(}n\mathtt{)},ts'\mathtt{)} : \Delta_2}$$

$$(147)\quad \frac{\Delta \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \mathtt{App(UOp(}n\mathtt{)},ts\mathtt{)} : \Delta_1 \quad \Delta_1 \vdash ts \Rightarrow ts' : \Delta_2}{\Delta \vdash t \Rightarrow \mathtt{App(UOp(}n\mathtt{)},ts'\mathtt{)} : \Delta_2}$$

$$(148)\quad \frac{\begin{array}{c}\Delta \overset{\mathsf{HNF}}{\vdash} t \Rightarrow t' : \Delta_1 \quad \Delta_1 \overset{\mathsf{share}}{\vdash} t' \Rightarrow t'' : \Delta_2 \\ \Delta_2 \cup \{\mathtt{TV(}n\mathtt{)} \mapsto t''\} \vdash t'' \Rightarrow t_3 : \Delta_3\end{array}}{\Delta \cup \{\mathtt{TV(}n\mathtt{)} \mapsto t\} \vdash \mathtt{TV(}n\mathtt{)} \Rightarrow t_3 : \Delta_3 \oplus \{\mathtt{TV(}n\mathtt{)} \mapsto t_3\}}$$

4.7.2 Reduktion auf Kopfnormalform: HNF

(149) $\Delta \vdash t \Rightarrow t : \Delta$

, falls $t \in \{\ \mathtt{N(}i\mathtt{)}, \mathtt{Char(}c\mathtt{)}, \mathtt{True}, \mathtt{False}, \mathtt{Nil}, \mathtt{Failed}, \mathtt{UOp(}n\mathtt{)}, \mathtt{DOp(}n\mathtt{)}, \mathtt{V(}n\mathtt{)}, \mathtt{C(}t_1\mathtt{,}t_2\mathtt{)}, \mathtt{Subst(}ts\mathtt{)}, \mathtt{T(}ts\mathtt{)}, \mathtt{App(UOp(}n\mathtt{)},ts\mathtt{)}, \mathtt{App(DOp(}n\mathtt{)},ts\mathtt{)}\}$

$$(150)\quad \frac{\Delta \vdash t \Rightarrow t' : \Delta_1 \quad \Delta_1 \overset{\mathsf{share}}{\vdash} t' \Rightarrow t'' : \Delta_2}{\Delta \cup \{\mathtt{TV(}n\mathtt{)} \mapsto t\} \vdash \mathtt{TV(}n\mathtt{)} \Rightarrow t'' : \Delta_2 \oplus \{\mathtt{TV(}n\mathtt{)} \mapsto t''\}}$$

$$(151)\quad \frac{\Delta \overset{\mathsf{prim}}{\vdash} t \Rightarrow t' : \Delta_1 \quad \Delta_1 \vdash t' \Rightarrow t'' : \Delta_2}{\Delta \vdash t \Rightarrow t'' : \Delta_2}$$

, falls $t = \mathtt{FOp(}n\mathtt{)}$ und $n \in Prim^0$ oder $t = \mathtt{App(FOp(}n\mathtt{)},t'\mathtt{)}$ und $n \in Prim^1$

(152) $\Delta \vdash \mathtt{FOp(}n\mathtt{)} \Rightarrow \mathtt{FOp(}n\mathtt{)} : \Delta$

, falls $n \in Prim^1$ oder $\Delta(n) = \mathtt{Function(}[\mathtt{App(FOp(}n\mathtt{)},as\mathtt{)}\,|\,ds]\mathtt{)}$

$$(153)\quad \frac{\Delta, ds \overset{\mathsf{rewrite}}{\vdash} t \Rightarrow t' : \Delta_1 \quad \Delta_1 \vdash t' \Rightarrow t'' : \Delta_2}{\Delta \vdash t \Rightarrow t'' : \Delta_2}$$

, falls $n \notin Prim^0 \cup Prim^1$ und $\begin{array}{l} t = \mathtt{FOp(}n\mathtt{)} \text{ oder} \\ t = \mathtt{App(FOp(}n\mathtt{)},t'''\mathtt{)}\end{array}$ und

$$\Delta(n) = \mathtt{Function}\left(\begin{array}{ll}[(\mathtt{FOp(}n\mathtt{)}, t, c, lds)\,|\,ds'] & \text{, falls } t = \mathtt{FOp(}n\mathtt{)} \\ [(\mathtt{App(FOp(}n\mathtt{)},as\mathtt{)}, t, c, lds)\,|\,ds'], & \text{sonst}\end{array}\right)$$

, wobei $\Delta(n) = \mathtt{Function(}ds\mathtt{)}$

$$(154)\quad \frac{\Delta \vdash f \Rightarrow f' : \Delta_1 \quad \Delta_1 \vdash \mathtt{App(}f'\mathtt{,}t\mathtt{)} \Rightarrow t' : \Delta_2}{\Delta \vdash \mathtt{App(}f\mathtt{,}t\mathtt{)} \Rightarrow t' : \Delta_2}$$

, falls f nicht in HNF.

4.7.3 Sharing erhalten: share

(155) $\Delta \vdash t \Rightarrow t : \Delta$

, falls $t \in \{$ $\texttt{Nil}, \texttt{True}, \texttt{False}, \texttt{Failed}, \texttt{N(}i\texttt{)}, \texttt{Char(}c\texttt{)}, \texttt{V(}n\texttt{)}, \texttt{UOp(}n\texttt{)}, \texttt{DOp(}n\texttt{)}, \texttt{FOp(}n\texttt{)}\}$

(156) $\Delta \vdash \texttt{C(}a\texttt{,}b\texttt{)} \Rightarrow \texttt{C(}a'\texttt{,}b'\texttt{)} : \Delta \cup \Delta'$

, wobei $a' = \texttt{TV(}n'\texttt{)},\ b' = \texttt{TV(}n''\texttt{)},\ n', n''$ sind neue Namen.
$\Delta' = \{\texttt{TV(}n'\texttt{)} \mapsto a, \texttt{TV(}n''\texttt{)} \mapsto b\}$

(157) $\Delta \vdash \texttt{App(}a\texttt{,}b\texttt{)} \Rightarrow \texttt{App(}a'\texttt{,}b'\texttt{)} : \Delta \cup \Delta'$

, wobei $a' = \texttt{TV(}n'\texttt{)},\ b' = \texttt{TV(}n''\texttt{)},\ n', n''$ sind neue Namen.
$\Delta' = \{\texttt{TV(}n'\texttt{)} \mapsto a, \texttt{TV(}n''\texttt{)} \mapsto b\}$

(158) $\Delta \vdash \texttt{T(}[t_1, \ldots, t_k]\texttt{)} \Rightarrow \texttt{T(}[t'_1, \ldots, t'_k]\texttt{)} : \Delta \cup \Delta'$

, wobei $t'_i = \texttt{TV(}n'_i\texttt{)}$ n'_i ist ein neuer Name.
$\Delta' = \{\texttt{TV(}n'_i\texttt{)} \mapsto t_i | 1 \leq i \leq k\}$

(159) $\Delta \vdash \texttt{Subst(}[(s_1, t_1), \ldots, (s_k, t_k)]\texttt{)} \Rightarrow \texttt{Subst(}[(s'_1, t'_1), \ldots, (s'_k, t'_k)]\texttt{)} : \Delta \cup \Delta'$

, wobei $s'_i = \texttt{TV(}n'_i\texttt{)},\ t'_i = \texttt{TV(}n''_i\texttt{)}$, n'_i, n''_i sind neue Namen
$\Delta' = \{\texttt{TV(}n'_i\texttt{)} \mapsto s_i, \texttt{TV(}n''_i\texttt{)} \mapsto t_i | 1 \leq i \leq k\}$

4.7.4 Rewriteschritt: rewrite

$$(160)\quad \frac{\Delta \overset{\mathsf{matches}}{\vdash} p' \downarrow t : OK(\Delta_1, \bar{t}) \quad \Delta_1 \cup \Delta^l \overset{\mathsf{NF}}{\vdash} cond' \Rightarrow \texttt{True} : \Delta_2}{\Delta, [(p, t', cond, lds)\,|\,ds] \vdash t \Rightarrow t'' : \Delta_2}$$

, wobei $t = \texttt{FOp(}n\texttt{)}$ oder $t = \texttt{App(FOp(}n\texttt{)}\texttt{,}t'''\texttt{)}$
$\{p', t'', cond', lds'\} = Rename(\{p, t', cond, lds\})$
$lds = [(\texttt{TV(}n_1\texttt{)}, t_1), \ldots, (\texttt{TV(}n_k\texttt{)}, t_k)]$
$\Delta^l = \{\texttt{TV(}n'_i\texttt{)} \mapsto t'_i | 1 \leq i \leq k\}$

$$(161)\quad \frac{\begin{array}{l}\Delta \overset{\mathsf{matches}}{\vdash} p' \downarrow t : OK(\Delta_1, \bar{t}) \quad \Delta_1 \cup \Delta^l \overset{\mathsf{NF}}{\vdash} cond' \Rightarrow \texttt{False} : \Delta_2 \\ \Delta_2, ds \vdash \bar{t} \Rightarrow t'' : \Delta_3\end{array}}{\Delta, [(p, t', cond, lds)\,|\,ds] \vdash t \Rightarrow t'' : \Delta_3}$$

, wobei $t = \texttt{FOp(}n\texttt{)}$ oder $t = \texttt{App(FOp(}n\texttt{)}\texttt{,}t'''\texttt{)}$
$\{p', cond', lds'\} = Rename(\{p, cond, lds\})$
$lds = [(\texttt{TV(}n_1\texttt{)}, t_1), \ldots, (\texttt{TV(}n_k\texttt{)}, t_k)]$
$\Delta^l = \{\texttt{TV(}n'_i\texttt{)} \mapsto t'_i | 1 \leq i \leq k\}$

$$(162)\quad \frac{\Delta \overset{\mathsf{matches}}{\vdash} p' \downarrow t : FAIL(\Delta_1, \bar{t}) \quad \Delta_1, ds \vdash \bar{t} \Rightarrow t'' : \Delta_2}{\Delta, [(p, t', cond, lds)\,|\,ds] \vdash t \Rightarrow t'' : \Delta_2}$$

, wobei $t = \texttt{FOp(}n\texttt{)}$ oder $t = \texttt{App(FOp(}n\texttt{)}\texttt{,}t'''\texttt{)}$
$p' = Rename(p)$

4.7.5 Matching: matches

(163) $\Delta_1 \vdash \mathtt{FOp(}n\mathtt{)} \downarrow \mathtt{FOp(}n\mathtt{)} : OK(\Delta_1, \mathtt{FOp(}n\mathtt{)})$

$$(164)\ \frac{\Delta_1 \overset{\mathsf{HNF}}{\vdash} t \Rightarrow t' : \Delta_2}{\Delta_1 \vdash p \downarrow t : r}$$

, falls $p \in \{\mathtt{True}, \mathtt{False}, \mathtt{Nil}, \mathtt{N(}i\mathtt{)}, \mathtt{Char(}c\mathtt{)}, \mathtt{DOp(}n\mathtt{)}, \mathtt{()}\}$

, wobei $r = \begin{cases} OK(\Delta_2, t') & \text{, falls } t' = p \\ FAIL(\Delta_2, t') & \text{, falls } t' \neq p \end{cases}$

$$(165)\ \frac{\Delta_1 \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \mathtt{C(}t_1\mathtt{,}t_2\mathtt{)} : \Delta_2 \quad \Delta_2 \vdash p_1 \downarrow t_1 : OK(\Delta_3, \bar{t_1}) \quad \Delta_3 \vdash p_2 \downarrow t_2 : r}{\Delta_1 \vdash \mathtt{C(}p_1\mathtt{,}p_2\mathtt{)} \downarrow t : r'}$$

, wobei $r' = \begin{array}{c} OK \\ FAIL \end{array} (\Delta_4, \mathtt{C(}\bar{t_1}\mathtt{,}\bar{t_2}\mathtt{)})$, falls $r = \begin{array}{c} OK \\ FAIL \end{array} (\Delta_4, \bar{t_2})$

$$(166)\ \frac{\Delta_1 \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \mathtt{C(}t_1\mathtt{,}t_2\mathtt{)} : \Delta_2 \quad \Delta_2 \vdash p_1 \downarrow t_1 : FAIL(\Delta_3, \bar{t_1})}{\Delta_1 \vdash \mathtt{C(}p_1\mathtt{,}p_2\mathtt{)} \downarrow t : FAIL(\Delta_3, \mathtt{C(}\bar{t_1}\mathtt{,}t_2\mathtt{)})}$$

$$(167)\ \frac{\Delta_1 \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \mathtt{Nil} : \Delta_2}{\Delta_1 \vdash \mathtt{C(}p_1\mathtt{,}p_2\mathtt{)} \downarrow t : FAIL(\Delta_2, \mathtt{Nil})}$$

$$(168)\ \frac{\Delta_1 \vdash ps \downarrow ts : r}{\Delta_1 \vdash \mathtt{App(FOp(}n\mathtt{),}ps\mathtt{)} \downarrow \mathtt{App(FOp(}n\mathtt{),}ts\mathtt{)} : r'}$$

, wobei $r' = \begin{array}{c} OK \\ FAIL \end{array} (\Delta_2, \mathtt{App(FOp(}n\mathtt{),}\bar{ts}\mathtt{)})$, falls $r = \begin{array}{c} OK \\ FAIL \end{array} (\Delta_2, \bar{ts})$

$$(169)\ \frac{\Delta_1 \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \mathtt{App(DOp(}n\mathtt{),}ts\mathtt{)} : \Delta_2 \quad \Delta_2 \vdash ps \downarrow ts : r}{\Delta_1 \vdash \mathtt{App(DOp(}n\mathtt{),}ps\mathtt{)} \downarrow t : r'}$$

, wobei $r' = \begin{array}{c} OK \\ FAIL \end{array} (\Delta_3, \mathtt{App(DOp(}n\mathtt{),}\bar{ts}\mathtt{)})$, falls $r = \begin{array}{c} OK \\ FAIL \end{array} (\Delta_3, \bar{ts})$

$$(170)\ \frac{\Delta_1 \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \mathtt{T(}[t_1, \ldots, t_k]\mathtt{)} : \Delta_2 \quad \Delta_2 \vdash p_1 \downarrow t_1 : FAIL(\Delta_3, \bar{t_1})}{\Delta_1 \vdash \mathtt{T(}[p_1, \ldots, p_k]\mathtt{)} \downarrow t : FAIL(\Delta_3, \mathtt{T(}[\bar{t_1}, t_2, \ldots, t_k]\mathtt{)})}$$

(171) $\Delta_1 \vdash \mathtt{T(}[]\mathtt{)} \downarrow \mathtt{T(}[]\mathtt{)} : OK(\Delta_1, \mathtt{T(}[]\mathtt{)})$

$$(172)\ \frac{\begin{array}{c}\Delta_1 \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \mathtt{T(}[t_1, \ldots, t_k]\mathtt{)} : \Delta_2 \quad \Delta_2 \vdash p_1 \downarrow t_1 : OK(\Delta_3) \\ \Delta_3 \vdash \mathtt{T(}[p_2, \ldots, p_k]\mathtt{)} \downarrow \mathtt{T(}[t_2, \ldots, t_k]\mathtt{)} : r\end{array}}{\Delta_1 \vdash \mathtt{T(}[p_1, \ldots, p_k]\mathtt{)} \downarrow t : r'}$$

, wobei $r' = \begin{array}{c} OK \\ FAIL \end{array} (\Delta_4, \mathtt{T(}[\bar{t_1} \mid \bar{ts}]\mathtt{)})$, falls $r = \begin{array}{c} OK \\ FAIL \end{array} (\Delta_4, \mathtt{T(}\bar{ts}\mathtt{)})$

(173) $\Delta_1 \vdash \mathtt{TV(}n\mathtt{)} \downarrow t : OK(\Delta_1 \oplus \{\mathtt{TV(}n\mathtt{)} \mapsto t\}, \mathtt{TV(}n\mathtt{)})$, falls $\mathtt{TV(}n\mathtt{)} \notin \mathcal{D}(\Delta_1)$

$$(174)\quad \frac{\Delta_1 \overset{\mathsf{NF}}{\vdash} \Delta(\mathtt{TV}(n)) \Rightarrow t_2 : \Delta_2 \quad \Delta_2 \overset{\mathsf{NF}}{\vdash} t \Rightarrow t_3 : \Delta_3}{\Delta_1 \vdash \mathtt{TV}(n) \downarrow t : r}$$

$$\text{, falls } \mathtt{TV}(n) \in \mathcal{D}(\Delta_1)$$

$$\text{, wobei } r = \begin{cases} OK(\Delta_3 \cup \{\mathtt{TV}(n) \mapsto t_2\}, t_3) & \text{, falls } t_2 = t_3 \\ FAIL(\Delta_3, t_3) & \text{, falls } t_2 \neq t_3 \end{cases}$$

$$(175)\quad \frac{\Delta_1 \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \mathtt{V}(n') : \Delta_2}{\Delta_1 \vdash \mathtt{Var(TV(}n\mathtt{))} \downarrow t : OK(\Delta_2 \oplus \{\mathtt{TV}(n) \mapsto \mathtt{V}(n')\}, \mathtt{TV}(n))}$$

$$\text{, falls } \mathtt{TV}(n) \notin \mathcal{D}(\Delta_1)$$

$$(176)\quad \frac{\Delta_1 \overset{\mathsf{NF}}{\vdash} t \Rightarrow \mathtt{V}(n') : \Delta_2 \quad \Delta_2 \overset{\mathsf{NF}}{\vdash} \Delta(\mathtt{TV}(n)) \Rightarrow t' : \Delta_3}{\Delta_1 \vdash \mathtt{Var(TV(}n\mathtt{))} \downarrow t : r}$$

$$\text{, falls } \mathtt{TV}(n) \in \mathcal{D}(\Delta_1)$$

$$\text{, wobei } r = \begin{cases} OK(\Delta_3 \oplus \{\mathtt{TV}(n) \mapsto \mathtt{V}(n')\}, \mathtt{TV}(n)) & \text{, falls } t' = \mathtt{V}(n') \\ FAIL(\Delta_3, \mathtt{V}(n')) & \text{, falls } t' \neq \mathtt{V}(n') \end{cases}$$

$$(177)\quad \frac{\Delta_1 \overset{\mathsf{NF}}{\vdash} t \Rightarrow t' : \Delta_2}{\Delta_1 \vdash \mathtt{Var(TV(}n\mathtt{))} \downarrow t : FAIL(\Delta_2, t')}$$

$$\text{, falls } t' \neq \mathtt{V}(n')$$

$$(178)\quad \frac{\Delta_1 \overset{\mathsf{HNF}}{\vdash} t \Rightarrow t' : \Delta_2}{\Delta_1 \vdash \mathtt{Op(TV(}n\mathtt{))} \downarrow t : r}$$

$$\text{, falls } \mathtt{TV}(n) \notin \mathcal{D}(\Delta_1)$$

$$\text{, wobei } \quad r = \begin{cases} OK(\Delta_2 \cup \Delta_3, \mathtt{TV}(n)) & \text{, falls } \quad t' = \mathtt{UOp}(n') \\ & \quad \text{oder } t' = \mathtt{App(UOp(}n'\mathtt{)},ts\mathtt{)} \\ FAIL(\Delta_2, t') & \text{, sonst} \end{cases}$$

$$\Delta_3 = \{\mathtt{TV}(n) \mapsto t'\}$$

$$(179)\quad \frac{\begin{array}{c}\Delta_2 \vdash \mathtt{Op(TV(}n\mathtt{))} \downarrow t : OK(\Delta_3 \cup \{\mathtt{TV}(n) \mapsto t_2\}, \bar{t}) \quad \Delta_3 \overset{\mathsf{NF}}{\vdash} t_2 \Rightarrow t_3 : \Delta_4 \\ \Delta_4 \overset{\mathsf{NF}}{\vdash} t' \Rightarrow t_4 : \Delta_5\end{array}}{\Delta_1 \vdash \mathtt{Op(TV(}n\mathtt{))} \downarrow t : r}$$

$$\text{, falls } \exists \mathtt{TV}(n) \mapsto t' \in \Delta_1$$

$$\text{, wobei } \quad \Delta_2 = \Delta_1 \setminus \{\mathtt{TV}(n) \mapsto t'\}$$

$$r = \begin{cases} OK(\Delta_5 \oplus \{\mathtt{TV}(n) \mapsto t_3\}, \mathtt{TV}(n)) & \text{, falls } t_3 = t_4 \\ FAIL(\Delta_5, t_3) & \text{, sonst} \end{cases}$$

$$(180)\quad \frac{\Delta_2 \vdash \mathtt{Op(TV(}n\mathtt{))} \downarrow t : FAIL(\Delta_3, \bar{t})}{\Delta_1 \vdash \mathtt{Op(TV(}n\mathtt{))} \downarrow t : FAIL(\Delta_3, \bar{t})}$$

$$\text{, falls } \exists \mathtt{TV}(n) \mapsto t' \in \Delta_1$$

$$\text{, wobei } \Delta_2 = \Delta_1 \setminus \{\mathtt{TV}(n) \mapsto t'\}$$

4.7.6 Primitive Funktionen: **prim**

$$(181)\quad \frac{\Delta \overset{\mathsf{NF}}{\vdash} t \Rightarrow \mathtt{T([N(}i\mathtt{),N(}i'\mathtt{)])} : \Delta'}{\Delta \vdash \mathtt{App(FOp("+"),}t\mathtt{)} \Rightarrow \mathtt{N(}i+i'\mathtt{)} : \Delta'}$$

$$(182)\quad \frac{\Delta \overset{\mathsf{NF}}{\vdash} t \Rightarrow \mathtt{T([N(}i\mathtt{),N(}i'\mathtt{)])} : \Delta'}{\Delta \vdash \mathtt{App(FOp("-"),}t\mathtt{)} \Rightarrow \mathtt{N(}r\mathtt{)} : \Delta'}$$

$$\text{, wobei } r = \begin{cases} 0 & \text{, falls } i' > i \\ i - i' & \text{, sonst} \end{cases}$$

$$(183)\quad \frac{\Delta \overset{\mathsf{NF}}{\vdash} t \Rightarrow \mathtt{T([N(}i\mathtt{),N(}i'\mathtt{)])} : \Delta'}{\Delta \vdash \mathtt{App(FOp("*"),}t\mathtt{)} \Rightarrow \mathtt{N(}i*i'\mathtt{)} : \Delta'}$$

$$(184)\quad \frac{\Delta \overset{\mathsf{NF}}{\vdash} t \Rightarrow \mathtt{T([N(}i\mathtt{),N(}i'\mathtt{)])} : \Delta'}{\Delta \vdash \mathtt{App(FOp("\backslash"),}t\mathtt{)} \Rightarrow \mathtt{N(}i \ \mathsf{div}\ i'\mathtt{)} : \Delta'}$$

, falls $i' \neq 0$

$$(185)\quad \frac{\Delta \overset{\mathsf{NF}}{\vdash} t \Rightarrow \mathtt{T([N(}i\mathtt{),N(}i'\mathtt{)])} : \Delta'}{\Delta \vdash \mathtt{App(FOp(}n\mathtt{),}t\mathtt{)} \Rightarrow r : \Delta_2}$$

, falls $n \in \{\mathtt{"<"}, \mathtt{">"}, \mathtt{"<="}, \mathtt{">="}\}$

$$\text{, wobei } r = \begin{cases} \mathtt{True} & \text{, falls } n = \mathtt{"<"} \text{ und } i < i' \\ \mathtt{False} & \text{, falls } n = \mathtt{"<"} \text{ und } i \geq i' \\ & \quad\ldots \\ \mathtt{True} & \text{, falls } n = \mathtt{">="} \text{ und } i \geq i' \\ \mathtt{False} & \text{, falls } n = \mathtt{">="} \text{ und } i < i' \end{cases}$$

$$(186)\quad \Delta \vdash \mathtt{FOp("newvar")} \Rightarrow \mathtt{V(}n\mathtt{)} : \Delta$$

, wobei n ein neuer Name ist.

$$(187)\quad \frac{\Delta \overset{\mathsf{NF}}{\vdash} t \Rightarrow t' : \Delta_1}{\Delta \vdash \mathtt{App(FOp("argsnum"),}t\mathtt{)} \Rightarrow \mathtt{N(0)} : \Delta_1}$$

, falls $t' = \mathtt{V(}n\mathtt{)}$ oder $t' = \mathtt{UOp(}n\mathtt{)}$

$$(188)\quad \frac{\Delta \overset{\mathsf{NF}}{\vdash} t \Rightarrow \mathtt{App(UOp(}n\mathtt{),}t'\mathtt{)} : \Delta_1}{\Delta \vdash \mathtt{App(FOp("argsnum"),}t\mathtt{)} \Rightarrow \mathtt{N(1)} : \Delta_1}$$

, falls $t' \neq \mathtt{T(}ts\mathtt{)}$

$$(189)\quad \frac{\Delta \overset{\mathsf{NF}}{\vdash} t \Rightarrow \mathtt{App(UOp(}n\mathtt{),T([}t_1,\ldots,t_k\mathtt{]))} : \Delta_1}{\Delta \vdash \mathtt{App(FOp("argsnum"),}t\mathtt{)} \Rightarrow \mathtt{N(}k\mathtt{)} : \Delta_1}$$

$$(190)\quad \frac{\Delta \overset{\mathsf{NF}}{\vdash} t \Rightarrow \mathtt{UOp(}n\mathtt{)} : \Delta_1}{\Delta \vdash \mathtt{App(FOp("opof"),}t\mathtt{)} \Rightarrow \mathtt{UOp(}n\mathtt{)} : \Delta_1}$$

$$(191)\quad \frac{\Delta \overset{\mathsf{NF}}{\vdash} t \Rightarrow \texttt{App(UOp(}n\texttt{),}t'\texttt{)} : \Delta_1}{\Delta \vdash \texttt{App(FOp("opof"),}t\texttt{)} \Rightarrow \texttt{UOp(}n\texttt{)} : \Delta_1}$$

$$(192)\quad \Delta \vdash \texttt{FOp("attributes")} \Rightarrow t : \Delta$$

$$\text{, falls } \exists \texttt{TV("!attributes")} \mapsto t \in \Delta$$

$$(193)\quad \frac{\Delta \overset{\mathsf{NF}}{\vdash} t \Rightarrow \texttt{T(}[t', \texttt{Nil}]\texttt{)} : \Delta'}{\Delta \vdash \texttt{App(FOp("/"),}t\texttt{)} \Rightarrow t' : \Delta'}$$

$$(194)\quad \frac{\Delta \overset{\mathsf{NF}}{\vdash} t \Rightarrow \texttt{T(}[t', \texttt{C(N(}i\texttt{),}l'\texttt{)}]\texttt{)} : \Delta_1 \qquad \Delta_1 \overset{\mathsf{NF}}{\vdash} \texttt{App(FOp("/"),T(}[t'', l']\texttt{))} \Rightarrow res : \Delta_2}{\Delta \vdash \texttt{App(FOp("/"),}t\texttt{)} \Rightarrow res : \Delta_2}$$

$$\text{, wobei } t'' = \begin{cases} t_3 & \text{, falls } t' = \texttt{App(UOp(}n\texttt{),}t_3\texttt{)} \text{ und } i = 1 \text{ und } t_3 \neq \texttt{T(}ts\texttt{)} \\ t'_i & \text{, falls } t' = \texttt{App(UOp(}n\texttt{),T(}[t'_1, \ldots, t'_k]\texttt{))} \text{ und } 0 < i \leq k \end{cases}$$

$$(195)\quad \frac{\Delta \overset{\mathsf{NF}}{\vdash} t \Rightarrow \texttt{T(}[t', \texttt{Nil}, t_2]\texttt{)} : \Delta_1}{\Delta \vdash \texttt{App(FOp("[<-]"),}t\texttt{)} \Rightarrow t_2 : \Delta_1}$$

$$(196)\quad \frac{\Delta \overset{\mathsf{NF}}{\vdash} t \Rightarrow \texttt{T(}[t', \texttt{C(N(}i\texttt{),}l\texttt{)}, t_2]\texttt{)} : \Delta_1 \qquad \Delta_1 \overset{\mathsf{NF}}{\vdash} \texttt{App(FOp("[<-]"),T(}[t'', l, t_2]\texttt{))} \Rightarrow res' : \Delta_2}{\Delta \vdash \texttt{App(FOp("[<-]"),}t\texttt{)} \Rightarrow \texttt{App(UOp(}n\texttt{),}res\texttt{)} : \Delta_2}$$

$$\text{, wobei } t'' = \begin{cases} t_3 & \text{, falls } i = 1 \text{ und } t' = \texttt{App(UOp(}n\texttt{),}t_3\texttt{)} \text{ und } t_3 \neq \texttt{T(}ts\texttt{)} \\ t'_i & \text{, falls } t' = \texttt{App(UOp(}n\texttt{),T(}[t'_1, \ldots, t'_k]\texttt{))} \text{ und } 0 < i \leq k \end{cases}$$

$$res = \begin{cases} res', \text{ falls } i = 1 \text{ und } t' = \texttt{App(UOp(}n\texttt{),}t_3\texttt{)} \text{ und } t_3 \neq \texttt{T(}ts\texttt{)} \\ \texttt{T(}[t'_1, .., t'_{i-1}, res', t'_{i+1}, .., t'_k]\texttt{)}, \text{ falls } t' = \texttt{App(UOp(}n\texttt{),T(}[t'_1, \ldots, t'_k]\texttt{))} \end{cases}$$

$$(197)\quad \frac{\Delta \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \texttt{T(}[a, b]\texttt{)} : \Delta_1 \qquad \Delta_1 \overset{\mathsf{HNF}}{\vdash} a \Rightarrow t : \Delta_2 \quad \Delta_2 \overset{\mathsf{HNF}}{\vdash} b \Rightarrow t' : \Delta_3}{\Delta \vdash \texttt{App(FOp("=="),}t\texttt{)} \Rightarrow res : \Delta_3}$$

$$\text{, falls } t \text{ oder } t' \in \{\ \texttt{Nil}, \texttt{Failed}, \texttt{True}, \texttt{False}, \texttt{N(}i\texttt{)}, \texttt{Char(}c\texttt{)}, \texttt{DOp(}n\texttt{)}, \texttt{UOp(}n\texttt{)}, \texttt{V(}n\texttt{)}, \texttt{FOp(}n\texttt{)} \}$$

$$\text{, wobei } res = \begin{cases} \texttt{True} & \text{, falls } t' = t \\ \texttt{False} & \text{, falls } t' \neq t \end{cases}$$

$$(198)\quad \frac{\Delta \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \texttt{T(}[a, b]\texttt{)} : \Delta_1 \qquad \Delta_1 \overset{\mathsf{HNF}}{\vdash} a \Rightarrow \texttt{Subst(}[t_1, \ldots, t_k]\texttt{)} : \Delta_2 \quad \Delta_2 \overset{\mathsf{HNF}}{\vdash} b \Rightarrow \texttt{Subst(}[t'_1, \ldots, t'_{k'}]\texttt{)} : \Delta_3}{\Delta \vdash \texttt{App(FOp("=="),}t\texttt{)} \Rightarrow \texttt{False} : \Delta_3}$$

$$\text{, falls } k \neq k'$$

$$(199)\quad \frac{\Delta \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \texttt{T(}[a, b]\texttt{)} : \Delta_1 \qquad \Delta_1 \overset{\mathsf{HNF}}{\vdash} a \Rightarrow \texttt{Subst(}[t_1, \ldots, t_k]\texttt{)} : \Delta_2 \quad \Delta_2 \overset{\mathsf{HNF}}{\vdash} b \Rightarrow \texttt{Subst(}[t'_1, \ldots, t'_k]\texttt{)} : \Delta_3 \qquad \Delta_3 \vdash \texttt{App(FOp("=="),T(}[t_1, t'_1]\texttt{))} \Rightarrow \texttt{False} : \Delta_4}{\Delta \vdash \texttt{App(FOp("=="),}t\texttt{)} \Rightarrow \texttt{False} : \Delta_4}$$

$$\frac{\begin{array}{c}\Delta \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \mathtt{T}([a,b]) : \Delta_1 \\ \Delta_1 \overset{\mathsf{HNF}}{\vdash} a \Rightarrow \mathtt{Subst}([t_1,\ldots,t_k]) : \Delta_2 \qquad \Delta_2 \overset{\mathsf{HNF}}{\vdash} b \Rightarrow \mathtt{Subst}([t'_1,\ldots,t'_k]) : \Delta_3 \\ \Delta_3 \vdash \mathtt{App(FOp("==")},\mathtt{T}([t_1,t'_1])) \Rightarrow \mathtt{True} : \Delta_4 \\ \Delta_4 \vdash \mathtt{App(FOp("==")},[\mathtt{T(Subst}([t_2,\ldots,t_k]),\mathtt{Subst}([t'_2,\ldots,t'_k]))]) \Rightarrow res : \Delta_5\end{array}}{\Delta \vdash \mathtt{App(FOp("==")},t) \Rightarrow res : \Delta_5} \tag{200}$$

$$\frac{\begin{array}{c}\Delta \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \mathtt{T}([a,b]) : \Delta_1 \\ \Delta_1 \overset{\mathsf{HNF}}{\vdash} a \Rightarrow \mathtt{C}(t_1,t_2) : \Delta_2 \quad \Delta_2 \overset{\mathsf{HNF}}{\vdash} b \Rightarrow \mathtt{C}(t'_1,t'_2) : \Delta_3 \\ \Delta_3 \vdash \mathtt{App(FOp("==")},\mathtt{T}([t_1,t'_1])) \Rightarrow \mathtt{False} : \Delta_4\end{array}}{\Delta \vdash \mathtt{App(FOp("==")},t) \Rightarrow \mathtt{False} : \Delta_4} \tag{201}$$

$$\frac{\begin{array}{c}\Delta \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \mathtt{T}([a,b]) : \Delta_1 \\ \Delta_1 \overset{\mathsf{HNF}}{\vdash} a \Rightarrow \mathtt{C}(t_1,t_2) : \Delta_2 \quad \Delta_2 \overset{\mathsf{HNF}}{\vdash} b \Rightarrow \mathtt{C}(t'_1,t'_2) : \Delta_3 \\ \Delta_3 \vdash \mathtt{App(FOp("==")},\mathtt{T}([t_1,t'_1])) \Rightarrow \mathtt{True} : \Delta_4 \\ \Delta_4 \vdash \mathtt{App(FOp("==")},\mathtt{T}([t_2,t'_2])) \Rightarrow res : \Delta_5\end{array}}{\Delta \vdash \mathtt{App(FOp("==")},t) \Rightarrow res : \Delta_5} \tag{202}$$

Die Fälle für $\mathtt{T}(ts)$ und $\mathtt{App}(a,b)$ analog.

$$\frac{\Delta \overset{\mathsf{NF}}{\vdash} \mathtt{App(FOp("\tilde{}")},\mathtt{App(FOp("==")},t)) \Rightarrow res : \Delta_1}{\Delta \vdash \mathtt{App(FOp("!=")},t) \Rightarrow res : \Delta_1} \tag{203}$$

$$\frac{\Delta \overset{\mathsf{NF}}{\vdash} t \Rightarrow t' : \Delta_1}{\Delta \vdash \mathtt{App(FOp("show")},t) \Rightarrow \bar{t} : \Delta_1} \tag{204}$$

, wobei $\bar{t}$ die Repräsentation von t' als ($\mathcal{QIR}$) Zeichenkette ist.

$$\frac{\Delta \overset{\mathsf{NF}}{\vdash} s \Rightarrow \mathtt{Subst}([(\mathtt{V}(n_i),t_i)]_{1\leq i\leq m}) : \Delta_2}{\Delta \vdash \mathtt{App(FOp("dom")},s) \Rightarrow res : \Delta_2} \tag{205}$$

, falls $\forall_{i\neq j} : n_i \neq n_j$

, wobei $res = \mathtt{C}(\mathtt{V}(n_1),\cdots,\mathtt{C}(\mathtt{V}(n_m),\mathtt{Nil})\cdots)$

$$\frac{\Delta \overset{\mathsf{NF}}{\vdash} t \Rightarrow \mathtt{T}([\mathtt{Subst}([(\mathtt{V}(n_i),a_i)]_{1<i<k_1}),\mathtt{Subst}([(\mathtt{V}(n'_i),b_i)]_{1<i<k_2})]) : \Delta_1}{\Delta \vdash \mathtt{App(FOp("compose")},t) \Rightarrow \mathtt{Subst}(as) : \Delta_1} \tag{206}$$

, falls $\forall_{i\neq j} : n_i \neq n_j,\ n'_i \neq n'_j$

, wobei $as = [(\mathtt{V}(n_1),a'_1),\ldots,(\mathtt{V}(n_{k_1}),a'_{k_1}),(\mathtt{V}(n'_{i_1}),b_{i_1}),\ldots,(\mathtt{V}(n'_{i_l}),b_{i_l})]$

$a'_i = a_i$, wobei jedes Vorkommen von $\mathtt{V}(n'_i)$ durch b_i ersetzt ist.

$\{n'_{i_1},\ldots,n'_{i_l}\} = \{n'_j | n'_j \notin \{n_1,\ldots,n_k\}\},\ n'_{i_j} \neq n'_{i_k} \forall j \neq k$

$$\frac{\Delta \overset{\mathsf{NF}}{\vdash} t \Rightarrow \mathtt{T}([\mathtt{Subst}([(\mathtt{V}(n_i),t_i)]_{1<i<m}),t']) : \Delta_1}{\Delta \vdash \mathtt{App(FOp("apply")},t) \Rightarrow \bar{t} : \Delta_1} \tag{207}$$

, falls $n_i \neq n_j \forall i \neq j$ und $t' = \mathtt{V}(n)$ oder $t' = \mathtt{UOp}(n')$

, wobei $\bar{t} = \begin{cases} t_j, \text{ falls } \exists j : n_j = n \text{ und } t' = \mathtt{V}(n) \\ t', \text{ sonst}\end{cases}$

$$(208)\quad \frac{\Delta \overset{\mathsf{NF}}{\vdash} t \Rightarrow \texttt{T([}s', \texttt{App(UOp(}n\texttt{)},t'\texttt{)])} : \Delta_1 \qquad \Delta_1 \overset{\mathsf{NF}}{\vdash} \texttt{App(FOp("apply"),T(}[s',t']\texttt{))} \Rightarrow \bar{t} : \Delta_2}{\Delta \vdash \texttt{App(FOp("apply")},t\texttt{)} \Rightarrow \texttt{App(UOp(}n\texttt{)},\bar{t}\texttt{)} : \Delta_2}$$

, falls $t' \neq \texttt{T}(ts)$

$$(209)\quad \frac{\Delta \overset{\mathsf{NF}}{\vdash} t \Rightarrow \texttt{T([}s', \texttt{App(UOp(}n\texttt{)},\texttt{T(}[t_1,\ldots,t_k]\texttt{)))])} : \Delta_1 \qquad \left[\Delta_i \overset{\mathsf{NF}}{\vdash} \texttt{App(FOp("apply")},t_i\texttt{)} \Rightarrow \bar{t}_i : \Delta_{1+i}\right]^{1\leq i\leq k}}{\Delta \vdash \texttt{App(FOp("apply")},t\texttt{)} \Rightarrow \texttt{App(UOp(}n\texttt{)},\texttt{T(}[\bar{t}_1,\ldots,\bar{t}_k]\texttt{))} : \Delta_{1+k}}$$

4.7.7 Strategiesemantik

strategy

$$(210)\quad \frac{\Gamma \overset{\mathsf{calctrs}}{\vdash} (trs, tact) : \Delta^{trs} \qquad (\Delta_1, s, 0, maxd) \overset{\mathsf{cntrl}}{\vdash} cnt : (\Delta_2, res, prot)}{(\Gamma, \Delta, trs, tac, maxd, goal) \vdash s : (res, prot)}$$

, wobei
$$\begin{aligned}
s &= \texttt{Strategy(}gd\texttt{,}ld\texttt{,}sel\texttt{,}cnt\texttt{,}lds\texttt{,}ots\texttt{,}mrs\texttt{)}\\
gd &= [(gn_1, gt_1), \ldots, (gn_m, gt_m)]\\
ld &= [(ln_1, lt_1), \ldots, (ln_n, lt_n)]\\
lds &= [(\texttt{TV(}n_1\texttt{)}, t_1), \ldots, (\texttt{TV(}n_k\texttt{)}, t_k)]\\
\Delta_1 &= \ \Delta \oplus \{\texttt{TV("goals")} \mapsto \texttt{C(}goal\texttt{,Nil)}, \texttt{TV("goal")} \mapsto goal\} \cup\\
&\qquad \Delta^g \cup \Delta^l \cup \Delta^{trs} \cup \Delta^{ld} \cup \Delta^u\\
\Delta^g &= \{\texttt{TV(}gn_i\texttt{)} \mapsto gt_i | 1 \leq i \leq m\}\\
\Delta^l &= \{\texttt{TV(}ln_i\texttt{)} \mapsto lt_i | 1 \leq i \leq n\}\\
\Delta^{ld} &= \{\texttt{TV(}n_i\texttt{)} \mapsto t_i | 1 \leq i \leq k\}\\
\Delta^u &= \{\texttt{UOp(}n\texttt{)} \mapsto ut | n \mapsto ut \in \Gamma_{|\texttt{Functor}} \cup \Gamma_{|\texttt{Constr}}\}
\end{aligned}$$

getout

$$(211)\quad \Delta \vdash str, s : (\Delta, \texttt{""})$$

, falls $str \notin [on_i]_{1\leq i\leq k}$

, wobei $s = \texttt{Strategy(}dds\texttt{,}lds\texttt{,}sel\texttt{,}cntrl\texttt{,}ldefs\texttt{,}[(on_i, ot_i)]_{1\leq i\leq k}\texttt{,}mrs\texttt{)}$

$$(212)\quad \frac{\Delta \overset{\mathsf{NF}}{\vdash} ot \Rightarrow res : \Delta'}{\Delta \vdash str, s : (\Delta', show(res))}$$

, falls $(str, ot) \in [(on_i, ot_i)]_{1\leq i\leq k}$

, wobei s wie oben.

calctrs

$$(213)\quad \frac{\vdash (rns, to', [], []) : [n_1, \ldots, n_p]}{\Gamma \vdash (\texttt{TRS(}rns\texttt{)}, \texttt{Tactic(}to\texttt{,}as\texttt{)}) : \Delta}$$

, wobei $to' = \begin{cases} to & \text{, falls } \texttt{Wildcard} \text{ in } to \text{ vorkommt.} \\ to \mathbin{+\!\!+} [\texttt{Wildcard}] & \text{, sonst.} \end{cases}$

$trs = \texttt{C(}t_1\texttt{,} \cdots \texttt{,C(}t_p\texttt{,Nil)} \cdots \texttt{)}$

$t_i = \texttt{T(}[n'_i, r_i]\texttt{)}$

$\Gamma(n_i) = \texttt{Rule(}r_i\texttt{)}$

$n'_i =$ der in $\mathcal{QIR}$ umgewandelte Name n_i

$attrs = \texttt{C(}a_1\texttt{,} \cdots \texttt{,C(}a_p\texttt{,Nil)} \cdots \texttt{)}$

$a_i = \texttt{T(}[n'_i, as'_i]\texttt{)}$

$as'_i = \texttt{C(}as'^1_i\texttt{,} \cdots \texttt{,C(}as'^{j_i}_i\texttt{,Nil)} \cdots \texttt{)}$

$[as^1_i, \ldots, as^{j_i}_i] = [an_j | (n_i, an_j) \in as]$

$as'^j_i =$ der in $\mathcal{QIR}$ umgewandelte Name as^j_i

$\Delta = \{\texttt{TV("trs")} \mapsto trs, \texttt{TV("!attributes")} \mapsto attrs\}$

$$(214)\quad \vdash (rns, [], pres, posts) : pres \mathbin{+\!\!+} rns \mathbin{+\!\!+} posts$$

$$(215)\quad \frac{\vdash (rns, ts, pres, posts) : ns}{\vdash (rns, [\texttt{Wildcard|}ts], pres, posts) : ns}$$

$$(216)\quad \frac{\vdash (rns, ts, pres, posts) : ns}{\vdash (rns, [\texttt{TacName(}n\texttt{)|}ts], pres, posts) : ns}$$

, falls $n \notin rns$

$$(217)\quad \frac{\vdash (rns -\!- [n], ts, pres', posts') : ns}{\vdash (rns, [\texttt{TacName(}n\texttt{)|}ts], pres, posts) : ns}$$

, falls $n \in rns$

, wobei $pres' = \begin{cases} pres & \text{, falls } \texttt{Wildcard} \notin ts \\ pres \mathbin{+\!\!+} [n] & \text{, sonst.} \end{cases}$

$posts' = \begin{cases} posts & \text{, falls } \texttt{Wildcard} \in ts \\ posts \mathbin{+\!\!+} [n] & \text{, sonst.} \end{cases}$

cntrl

$$(218)\quad \frac{(\Delta \cup \{\texttt{TV("goals")} \mapsto \texttt{Nil}\}) \overset{\mathsf{getout}}{\vdash} \texttt{"solved"}, s : (\Delta', res)}{(\Delta \cup \{\texttt{TV("goals")} \mapsto \texttt{Nil}\}, s, d, md) \vdash cnt : (\Delta', res, \texttt{"Solved\textbackslash 0"})}$$

$$(219)\quad \frac{(\Delta \cup \Delta_g) \overset{\mathsf{getout}}{\vdash} \texttt{"bound"}, s : (\Delta', pr)}{(\Delta \cup \Delta_g, s, d, md) \vdash cnt : (\Delta', \texttt{""}, \texttt{"Bound:"} \mathbin{+\!\!+} pr \mathbin{+\!\!+} \texttt{"\textbackslash 0"})}$$

, falls $d > md$

, wobei $\Delta_g = \{\texttt{TV("goals")} \mapsto \texttt{C(}t_1\texttt{,}t_2\texttt{)}\}$

$$(220)\quad \frac{\Delta \cup \Delta_g \overset{\mathsf{NF}}{\vdash} \texttt{App(}sel\texttt{,TV("goals"))} \Rightarrow \texttt{T(}[g,gs]\texttt{)} : \Delta' \qquad (\Delta'', d, md, s) \overset{\mathsf{cntrl'}}{\vdash} cnt : (\Delta_3, res, prot, DEAD, st) \quad \Delta_3 \overset{\mathsf{getout}}{\vdash} \texttt{"dead"}, s : (\Delta_4, pr)}{(\Delta \cup \Delta_g, s, d, md) \vdash cnt : (\Delta_4, res, prot +\!\!+ \texttt{"Dead:"} +\!\!+ pr +\!\!+ \texttt{"\textbackslash 0"})}$$

, falls $d \leq md$

, wobei $\Delta_g = \{\texttt{TV("goals")} \mapsto \texttt{C(}t_1\texttt{,}t_2\texttt{)}\}$

$\Delta'' = \Delta' \oplus \{\texttt{TV("goal")} \mapsto g, \texttt{TV("goals")} \mapsto gs\}$

$s = \texttt{Strategy(}gd\texttt{,}ld\texttt{,}sel\texttt{,}cntrl\texttt{,}lds\texttt{,}os\texttt{,}mrs\texttt{)}$

$$(221)\quad \frac{\Delta \cup \Delta_g \overset{\mathsf{NF}}{\vdash} \texttt{App(}sel\texttt{,TV("goals"))} \Rightarrow \texttt{T(}[g,gs]\texttt{)} : \Delta' \qquad (\Delta'', d, md, s) \overset{\mathsf{cntrl'}}{\vdash} cnt : (\Delta_3, res, prot, ALIVE, st)}{(\Delta \cup \Delta_g, s, d, md) \vdash cnt : (\Delta_3, res, prot)}$$

, falls $d \leq md$

, wobei $\Delta_g = \{\texttt{TV("goals")} \mapsto \texttt{C(}t_1\texttt{,}t_2\texttt{)}\}$

$\Delta'' = \Delta' \oplus \{\texttt{TV("goal")} \mapsto g, \texttt{TV("goals")} \mapsto gs\}$

$s = \texttt{Strategy(}gd\texttt{,}ld\texttt{,}sel\texttt{,}cntrl\texttt{,}lds\texttt{,}os\texttt{,}mrs\texttt{)}$

cntrl'

$$(222)\quad (\Delta, d, md, s) \vdash \texttt{Break} : (\Delta, \texttt{""}, \texttt{""}, DEAD, BROKEN)$$

$$(223)\quad \frac{(\Delta, d, md, s) \vdash c_1 : (\Delta', res, pr, dd, BROKEN)}{(\Delta, d, md, s) \vdash \texttt{Seq(}c_1\texttt{,}c_2\texttt{)} : (\Delta', res, pr, dd, BROKEN)}$$

$$(224)\quad \frac{(\Delta, d, md, s) \vdash c_1 : (\Delta', res, pr, dd, OK) \qquad (\Delta', d, md, s) \vdash c_2 : (\Delta'', res', pr', dd', st')}{(\Delta, d, md, s) \vdash \texttt{Seq(}c_1\texttt{,}c_2\texttt{)} : (\Delta'', res +\!\!+ res', pr +\!\!+ pr', d_3, st')}$$

, wobei $d_3 = \begin{cases} DEAD & \text{, falls } dd = dd' = DEAD \\ ALIVE & \text{, sonst} \end{cases}$

$$(225)\quad \frac{(\Delta, d, md, s) \vdash c_1 : (\Delta', res, pr, dd, st)}{(\Delta, d, md, s) \vdash \texttt{Else(}c_1\texttt{,}c_2\texttt{)} : (\Delta', res, pr, dd, st)}$$

, falls $st = BROKEN$ oder $dd = ALIVE$

$$(226)\quad \frac{(\Delta, d, md, s) \vdash c_1 : (\Delta', res, pr, DEAD, OK) \qquad (\Delta', d, md, s) \vdash c_2 : (\Delta'', res', pr', dd', st')}{(\Delta, d, md, s) \vdash \texttt{Else(}c_1\texttt{,}c_2\texttt{)} : (\Delta'', res +\!\!+ res', pr +\!\!+ pr', dd', st')}$$

$$(227)\quad \frac{\Delta \overset{\mathsf{NF}}{\vdash} cond \Rightarrow \texttt{False} : \Delta'}{(\Delta, d, md, s) \vdash \texttt{If(}cond\texttt{,}c\texttt{)} : (\Delta', \texttt{""}, \texttt{""}, DEAD, OK)}$$

$$(228)\quad \frac{\Delta \overset{\mathsf{NF}}{\vdash} cond \Rightarrow \texttt{True} : \Delta' \quad (\Delta', d, md, s) \vdash c : (\Delta'', res, pr, dd, st)}{(\Delta, d, md, s) \vdash \texttt{If(}cond\texttt{,}c\texttt{)} : (\Delta'', res, pr, dd, st)}$$

(229) $(\Delta, d, md, s) \vdash \texttt{Empty} : (\Delta, \texttt{""}, \texttt{""}, ALIVE, OK)$

$$(230)\quad \frac{(\Delta, d, md, s) \vdash c : (\Delta', res, pr, dd, st)}{(\Delta, d, md, s) \vdash \texttt{Assign(}c\texttt{,}as\texttt{)} : (\Delta', res, pr, dd, st)}$$

, falls $st = BROKEN$ oder $dd = DEAD$

$$(231)\quad \frac{(\Delta, d, md, s) \vdash c : (\Delta', res, pr, ALIVE, OK)}{(\Delta, d, md, s) \vdash \texttt{Assign(}c\texttt{,}as\texttt{)} : (\Delta'', res, pr, ALIVE, OK)}$$

, wobei $\Delta'' = \Delta' \oplus \Delta^{as}$
$as = [(an_i, at_i)]_{1 \leq i \leq m}$
$\Delta^{as} = \{\texttt{TV(}an_i\texttt{)} \mapsto at_i | 1 \leq i \leq m\}$

$$(232)\quad \frac{(\Delta, d, md, s) \overset{\mathsf{mr}}{\vdash} (mr_i, t) : (\Delta', res, pr, dd)}{(\Delta, d, md, s) \vdash \texttt{Call(}n\texttt{,}t\texttt{)} : (\Delta', res, pr, dd, OK)}$$

, wobei $s = \texttt{Strategy(}gd\texttt{,}ld\texttt{,}sel\texttt{,}cnt\texttt{,}lds\texttt{,}ots\texttt{,}mrs\texttt{)}$
$mrs = [(n_1, mr_1), \ldots, (n, mr_i), \ldots, (n_m, mr_m)]$

$$(233)\quad \frac{\Delta \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \texttt{Nil} : \Delta'}{(\Delta, d, md, s) \vdash \texttt{For(}p\texttt{,}t\texttt{,}c\texttt{)} : (\Delta', \texttt{""}, \texttt{""}, DEAD, OK)}$$

$$(234)\quad \frac{\Delta \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \texttt{C(}t_1\texttt{,}t_2\texttt{,:)}\Delta' \quad \Delta' \overset{\mathsf{matches}}{\vdash} p' \downarrow t_1 : FAIL(\Delta'', \bar{t_1}) \quad (\Delta'', d, md, s) \vdash \texttt{For(}p'\texttt{,}t_2\texttt{,}c'\texttt{)} : (\Delta_3, res, pr, dd, st)}{(\Delta, d, md, s) \vdash \texttt{For(}p\texttt{,}t\texttt{,}c\texttt{)} : (\Delta_3, res, pr, dd, st)}$$

, wobei $\{p', c'\} = Rename(\{p, c\})$

$$(235)\quad \frac{\Delta \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \texttt{C(}t_1\texttt{,}t_2\texttt{)} : \Delta' \quad \Delta' \overset{\mathsf{matches}}{\vdash} p' \downarrow t_1 : OK(\Delta'', \bar{t_1}) \quad (\Delta'', d, md, s) \vdash c' : (\Delta_3, res, pr, dd, BROKEN)}{(\Delta, d, md, s) \vdash \texttt{For(}p\texttt{,}t\texttt{,}c\texttt{)} : (\Delta_3, res, pr, dd, OK)}$$

, wobei $\{p', c'\} = Rename(\{p, c\})$

$$(236)\quad \frac{\Delta \overset{\mathsf{HNF}}{\vdash} t \Rightarrow \texttt{C(}t_1\texttt{,}t_2\texttt{)} : \Delta' \quad \Delta' \overset{\mathsf{matches}}{\vdash} p' \downarrow t_1 : OK(\Delta'', \bar{t_1}) \quad (\Delta'', d, md, s) \vdash c' : (\Delta_3, res, pr, dd, OK) \quad (\Delta_3, d, md, s) \vdash \texttt{For(}p'\texttt{,}t_2\texttt{,}c'\texttt{)} : (\Delta_4, res', pr', dd', st')}{(\Delta, d, md, s) \vdash \texttt{For(}p\texttt{,}t\texttt{,}c\texttt{)} : (\Delta_4, res \mathbin{+\!\!+} res', pr \mathbin{+\!\!+} pr', dd'', OK)}$$

, wobei $\{p', c'\} = Rename(\{p, c\})$

$$dd'' = \begin{cases} ALIVE & \text{, falls } dd = ALIVE \text{ oder } dd' = ALIVE \\ DEAD & \text{, sonst} \end{cases}$$

mr

$$(237)\quad \frac{\Delta \overset{\mathsf{matches}}{\vdash} rp' \downarrow t : FAIL(\Delta', \bar{t})}{(\Delta, s, d, md) \vdash (mr, t) : (\Delta', \texttt{""}, \texttt{""}, DEAD, OK)}$$

, wobei $\{rp', gp', lds', res', ot', as', cond'\} = Rename(\{rp, gp, lds, res, ot, as, cond\})$
$mr = \texttt{MetaRule(}rp\texttt{,}cond\texttt{,}gp\texttt{,}res\texttt{,}as\texttt{,}ot\texttt{,}lds\texttt{)}$

$$(238)\quad \frac{\Delta \overset{\textsf{matches}}{\vdash} rp' \downarrow t : OK(\Delta', \bar{t}) \quad \Delta' \overset{\textsf{matches}}{\vdash} gp' \downarrow \texttt{TV("goal")} : FAIL(\Delta'', \bar{g})}{(\Delta, s, d, md) \vdash (mr, t) : (\Delta'', \texttt{""}, \texttt{""}, DEAD, OK)}$$

, wobei $\{rp', gp', lds', res', ot', as', cond'\} = Rename(\{rp, gp, lds, res, ot, as, cond\})$
$mr = \texttt{MetaRule}(rp\texttt{,}cond\texttt{,}gp\texttt{,}res\texttt{,}as\texttt{,}ot\texttt{,}lds)$

$$(239)\quad \frac{\begin{array}{c}\Delta \overset{\textsf{matches}}{\vdash} rp' \downarrow t : OK(\Delta', \bar{t}) \quad E' \overset{\textsf{matches}}{\vdash} gp' \downarrow \texttt{TV("goal")} : OK(\Delta'', \bar{g}) \\ \Delta_3 \overset{\textsf{NF}}{\vdash} cond' \Rightarrow \texttt{False} : \Delta_4\end{array}}{(\Delta, s, d, md) \vdash (mr, t) : (\Delta_4, \texttt{""}, \texttt{""}, DEAD, OK)}$$

, wobei $\{rp', gp', lds', res', ot', as', cond'\} = Rename(\{rp, gp, lds, res, ot, as, cond\})$
$mr = \texttt{MetaRule}(rp\texttt{,}cond\texttt{,}gp\texttt{,}res\texttt{,}as\texttt{,}ot\texttt{,}lds)$
$\Delta_3 = \Delta'' \oplus lds'$

$$(240)\quad \frac{\begin{array}{c}\Delta \overset{\textsf{matches}}{\vdash} rp' \downarrow t : OK(\Delta', \bar{t}) \quad E' \overset{\textsf{matches}}{\vdash} gp' \downarrow \texttt{TV("goal")} : OK(\Delta'', \bar{g}) \\ \Delta_3 \overset{\textsf{NF}}{\vdash} cond' \Rightarrow \texttt{True} : \Delta_4 \qquad\qquad \Delta_4 \overset{\textsf{NF}}{\vdash} ot' \Rightarrow otr : \Delta_5 \\ \Delta_5 \overset{\textsf{NF}}{\vdash} \texttt{App(FOp("++"),T(}[res', \texttt{TV("goals")}]\texttt{))} \Rightarrow resr : \Delta_6 \\ \left[\Delta_{6+i} \overset{\textsf{NF}}{\vdash} at_i \Rightarrow atr_i : \Delta_{7+i}\right]^{1 \le i \le m} \\ (\tilde{\Delta}, s, d+1, md) \overset{\textsf{cntrl}}{\vdash} cnt : (\tilde{\tilde{\Delta}}, res, pr)\end{array}}{(\Delta, s, d, md) \vdash (mr, t) : (\hat{\Delta}, res, \texttt{"MR:"} +\!\!+ show(otr) +\!\!+ \texttt{"\textbackslash 0"} +\!\!+ pr, ALIVE, OK)}$$

, wobei $\{rp', gp', lds', res', ot', as', cond'\} = Rename(\{rp, gp, lds, res, ot, as, cond\})$
$mr = \texttt{MetaRule}(rp\texttt{,}cond\texttt{,}gp\texttt{,}res\texttt{,}as\texttt{,}ot\texttt{,}lds)$
$\Delta_3 = \Delta'' \oplus lds'$
$\Delta_7 = \Delta_6 \oplus \{\texttt{TV("goals")} \mapsto resr\}$
$\tilde{\Delta} = \Delta_{7+m} \oplus \{\texttt{TV(}an_i\texttt{)} \mapsto atr_i | 1 \le i \le m\}$
$as' = [(an_1, at_1), \ldots, (an_m, at_m)]$
$\hat{\Delta} = \tilde{\tilde{\Delta}} \oplus \{\texttt{TV(}ln_i\texttt{)} \mapsto \Delta_6(\texttt{TV(}ln_i\texttt{)}) | 1 \le i \le l\} \oplus$
$\{\texttt{TV("goals")} \mapsto \Delta_6(\texttt{TV("goals")})\} \oplus$
$\{\texttt{TV("goal")} \mapsto \Delta_6(\texttt{TV("goal")})\}$
$s = \texttt{Strategy(}gd\texttt{,}[(ln_1, lt_1), \ldots, (ln_l, lt_l)]\texttt{,}a\texttt{,}b\texttt{,}c\texttt{,}d\texttt{,}e\texttt{)}$

4.7.8 Aktionssemantik: solve

$$(241)\quad \frac{\begin{array}{c}\overset{\textsf{prog}}{\vdash} p : (\Gamma, \Delta, \Theta) \\ (\Gamma, \Delta, \Gamma(trsn), \Gamma(tacn), depth, \Gamma(goaln)) \overset{\textsf{strategy}}{\vdash} \Delta(stratn) : (res, prot)\end{array}}{(p, trsn, tacn, stratn, goaln, depth) \vdash (res, prot)}$$

, falls $trsn \in \mathcal{D}(\Gamma_{|\texttt{TRS}})$, $tacn \in \mathcal{D}(\Gamma_{|\texttt{Tactic}})$
$goaln \in \mathcal{D}(\Gamma_{|\texttt{Goal}})$, $stratn \in \mathcal{D}(\Delta_{|\texttt{Strategy}})$

4.7.9 Aktionssemantik: eval

$$(242)\quad \frac{\overset{\mathsf{prog}}{\vdash} p : (\Gamma, \Delta, \Theta) \quad (\emptyset, \Gamma) \overset{\mathsf{termsynt}}{\vdash} t : fns \quad (\Gamma, \Theta) \overset{\mathsf{termtype}}{\vdash} t : \tau \quad \Delta \overset{\mathsf{NF}}{\vdash} t \Rightarrow t' : \Delta'}{p \vdash t \Rightarrow show(t')}$$

$$\text{, falls } fns \subseteq \mathcal{D}(\Delta_{|\texttt{Function}})$$

4.8 Kommentar zur dynamischen Semantik

4.8.1 NF

Diese Regeln beschreiben die Auswertung eines Terms auf Normalform. Eine Normalform ist dabei

- Eine Konstante, wie `N(`i`)`
- Ein Datenconstructor oder ein Useroperator: `DOp(`n`)`,`UOp(`n`)`
- Die Applikation eines Datenconstructors/Useroperators auf Argumente, die in Normalform sind.
- Ein Cons, dessen beide Argumente Normalformen sind: `C(`a`,`b`)`.
- Eine Substitution, deren Argumente Normalformen sind: `Subst(`ts`)`.
- Ein Tupel, deren Komponenten Normalformen sind: `T(`ts`)`.

Die Klausel $\Gamma \vdash t \Rightarrow t' : \Gamma'$ besagt, daß unter den Bindungen Γ der Term t zu t' auswertet, mit neuen Bindungen Γ'. Die Bindungen repräsentieren den *Heap* funktionaler Sprachen. Terme der Form `TV(`n`)`, bezeichnen Verweise auf den Term, der in Γ an `TV(`n`)` gebunden ist. Derselbe Ausdruck kann also geteilt (geshared) werden, indem mehrere Vorkommen von `TV(`n`)` in einem Term existieren. Dies ist die einzige Möglichkeit, Sharing mit *Termen* zu beschrieben, da Sharing an sich Term*graphen* entspricht, die nur so mit Termen beschrieben werden können. Da Quetzal *needed-evaluation* zur Auswertung von Ausdrücken benutzt, muß sichergestellt sein, daß ein gegebenes Sharing nicht durchbrochen werden kann: Ein Term, der selbst wieder reduzibel ist, darf nicht "verdoppelt" werden. Wir werden später sehen, wo dies die Auswertungsregeln beeinflußt. Aus diesem Grund liefert eine Auswertung eines Terms auch immer eine neue Umgebung Γ': Die Bindungen von `TV(`n`)` Termen könnten sich verändert haben.
Die Regeln (142) bis (148) beschreiben die Auswertung eines Terms t zu Normalform. Dazu wird der Term erst auf Kopfnormalform (HNF) gebracht, danach weiter behandelt. Die erste Regel handelt den Fall eines Terms ab, der zu einer Konstanten auswertet. Die Regeln (143-147) betrachten die Fälle, eines Terms mit Argumenten. Dabei ist in Regel (144) zu beachten, daß für eine Substitution der erste Term eines Paares immer zu einer Variablen auswerten muß und alle Variablen paarweise verschieden sein müssen.

Die letzte Regel nun ist die Quelle der needed-evaluation. Falls der auszuwertende Term ein Verweis auf einen (möglicherweise) gesharten Term ist ($\texttt{TV}(n)$), dann wird zuerst der Term, auf den verwiesen wird (t') zu HNF ausgewertet. Und zwar in einer Umgebung, *die keine Bindung für* $\texttt{TV}(n)$ *enthält*. Der Grund dafür ist einfach: Wenn t' zu HNF auswertet, ohne daß der Wert von $\texttt{TV}(n)$ benötigt wird, macht es keinen Unterschied, ob eine Bindung existiert. Wenn aber der Wert von $\texttt{TV}(n)$ benötigt wird, dann liegt der Fall eines "black holes" vor: Die Auswertung wird auf keinen Fall terminieren. Damit kann mit der Semantik ein "black hole" erkannt werden: Die Ableitung ist unmöglich, weil es keine Bindung für $\texttt{TV}(n)$ in Γ gibt. Nun ist t' entweder eine Konstante, in welchem Fall wir einfach mit der Auswertung unter der neuen Bindung $\texttt{TV}(n) \mapsto t'$ fortfahren können, oder ein Term mit Argumenten, wie etwa $\texttt{C}(a,b)$. Im letzteren Fall muß man sicherstellen, daß Zugriffe auf die Argumente von t' wieder geshared werden[21]. Die Klausel $\Gamma \overset{\mathsf{share}}{\vdash} t \Rightarrow t' : \Gamma'$ sorgt gerade dafür. Die letzte Klausel wertet nun den (neu gesharten) Term t'' zu Normalform aus. Das Ergebnis ist ein Term t_3 und die Bindung von $\texttt{TV}(n)$ an t_3.

4.8.2 HNF

Ein Term ist in Kopfnormalform, wenn er

- Eine Konstante, ein Datenconstructor oder ein Useroperator ist.
- An der Wurzel ein `C`, `Subst`, `T` stehen hat, gleichgültig, ob die Argumente in Kopfnormalform sind, oder nicht.
- Ein nicht reduzierbarer Funktionsterm ist.

Die Regeln (149) bis (154) beschreiben die Ableitung zur HNF. Die erste Regel trifft auf Terme, die bereits in HNF sind. Die Regel (150) ist wiederum für den Erhalt des Sharings zuständig und ähnelt der Regel aus NF. Die folgende Regel ist für die Reduktion von primitiven Funktionen zuständig. Wenn der Term t die Anwendung einer primitiven Funktion ist, dann wird er reduziert. Diese Reduktion endet nicht notwendigerweise in einer HNF, also wird der resultierende Term wiederum auf HNF reduziert. Die nächste Regel (152) trifft zu, falls t eine Funktion mit zuwenig Argumenten ist. Diese kann nicht ausgewertet werden, bis sie ihre Argumente bekommt. Regel (153) betrachtet den Fall, daß der Term t mittels einer nicht-primitiven Funktion reduziert werden kann. Der sich ergebende Term wird auf HNF gebracht und zurückgegeben. Die Unterscheidung in der Regel betrifft Funktionen ohne Argument und Funktionen mit Argumenten. Da alle Gleichungen einer Funktion entweder alle eine argumentlose Funktion definieren, oder alle eine Funktion mit einem Argument (das auch ein Tupel sein kann), kann man hier entscheiden, ob die Funktion hier angewandt werden kann, oder nicht.
Die letzte Regel (154) wertet eine Anwendung aus, deren Kopfelement nicht in HNF ist. Zuerst wird das Kopfelement auf HNF gebracht, dann die Anwendung ausgewertet.

[21]Sonst würde z.B. in $t' = $ `C(FOp("newvar"),Nil)` der Term `FOp("newvar")` vielleicht zweimal ausgewertet.

4.8.3 share

Die Regeln (155) bis (159) beschreiben das Sharing eines Terms. Jedes Argument des Terms muß durch `TV(`n`)` ersetzt werden, wobei n ein noch nie verwandter, frischer Name ist. Die Umgebung muß um die Bindungen dieser neuen Verweise an die alten Argumente ergänzt werden.

4.8.4 rewrite

Die 3 Regeln (160) bis (162) behandeln die Reduktion einer Funktion: $\Gamma, ds \vdash t \Rightarrow t' : \Gamma'$, falls der Term t an der Wurzel durch eine Gleichung aus ds zu t' reduziert werden kann.
Jede Gleichung für eine Funktion ist ein Quadrupel $(p, t, c, ldefs)$ aus einem Pattern p, einem Ergebnis t, einer Bedingung c und lokalen Definitionen $ldefs$.
Um einen Term mit einer Gleichung zu reduzieren, muß zuerst das Pattern der Gleichung auf den Term gematcht werden. Das Ergbnis dieses Matchversuchs kann erfolgreich ($OK(\Gamma, \bar{t})$) sein, oder fehlschlagen ($FAIL(\Gamma, \bar{t})$). Γ enthält dabei Bindungen für die Variablen in p. Da der Term t beim Matchen ggf. reduziert werden muß, um das Matching zu prüfen, ist $\bar{t}$ noch der reduzierte Term. Um Zusammenstöße von Namen in einer Gleichung und bereits existierenden Namen zu vermeiden, müssen die Variablen einer Gleichung konsistent umbenannt werden. In den Regeln kommt dies durch die Einführung eines neuen Quadrupels $(p', t'', c', ldefs')$ zum Ausdruck.
Regel (160) behandelt den Fall einer erfolgreichen Reduktion. Zuerst wird das Pattern erfolgreich gematcht, danach die Bedingung der Regel zu `True` ausgewertet, und zwar in einer Umgebung, die die lokalen Definitionen beinhaltet. Der Term t reduziert damit zum Term t'', dem umbenannten Ergebnis der Regel.
Die nächste Regel (161) behandelt den Fall, daß die Bedingung zu `False` auswertet, die Gleichung also nicht anwendbar ist. Mit der dritten Klausel wird dann versucht, den Term t mit einer der übrigen Gleichungen zu reduzieren.
Die letzte Regel schließlich trifft zu, wenn das Matching des Patterns bereits fehlschlug. Auch hier wird versucht, den Term mit einer der verbleibenden Regeln zu reduzieren.
Man beachte, daß es keine Regel für den Fall gibt, daß die Liste der Gleichungen leer ist. Dieser Fall repräsentiert einen Laufzeitfehler[22] und somit hat ein solcher Reduktionsversuch auch keine Semantik.

4.8.5 matches

Regel (163) betrachtet den Fall, daß das Funktionssymbol einer Regel gematcht werden muß. Die nächste Regel behandelt den Fall eines konstanten Patterns. Der Term t wird auf HNF reduziert und dann mit dem Pattern verglichen.
Die Regeln (165) bis (167) betreffen ein Cons Pattern `C(`a`,`b`)`. Wie man sehen kann, wird zuerst das Kopfelement gematcht, dann die Restliste.

[22]"Missing case in the definition of.."

Regel (168) behandelt den Fall der Anwendung einer Funktion auf Argumente, wie er in einem Pattern einer Gleichung auftreten kann.
Die nächste Regel kümmert sich um das Matching von Datenconstructoren mit Argumenten.
Die Regeln (170) bis (172) behandeln das Matching von Tupeln. Quetzal kennt keine lazy Produktpattern (siehe [PJ86]). Auch hier wird von links nach rechts gematcht.
Die beiden folgenden Regeln matchen Variablen. Da Quetzal auch nicht-links-lineare Pattern erlaubt, muß man zwei Fälle unterscheiden: Das erste Matchen der Variable und weitere Matches. Bei weiteren Matchen müssen die beiden Terme gleiche Normalformen haben, was durch die zweite Regel ausgedrückt wird.
Die Regeln (175) bis (177) betreffen Termvariablenpattern. Zusätzlich zu den beiden Fällen, daß `TV(`n`)` zum ersten mal gematcht werden soll oder schon gematcht wurde, muß hier auch noch geprüft werden, ob t zu einer Termvariablen auswertet.
Die letzten drei Regeln ab (178) betreffen Usertermpattern in ähnlicher Weise.

4.8.6 prim

Die Regeln (181) bis (184) definieren Addition, Subtraktion, Multiplikation und Division, wenn die Argumente zu Zahlen auswerten. Da Quetzal nur natürliche Zahlen kennt, werden Subtraktion und Division modifiziert, damit das Ergebnis wiederum eine ganze Zahl ist.
Regel (185) behandelt die Vergleichsoperationen `<`, `>`, `<=` und `>=`, und bietet keine Überraschungen.
Die nächste Regel definiert die primitive Funktion `"newvar"`, die eine neue Termvariable liefert.
Die folgenden drei Regeln (187) bis (189) sind die Implementierung der `"argsnum"` Funktion, die die Anzahl der Argumente eines Userterms zurückgibt.
Die nächsten beiden Regeln definieren `"opof"`: Die Funktion, die zu einem Userterm den Wurzeloperator zurückgibt.
Regel (192) ist die Schnittstelle zu den Attributen der Regeln. Das Ergebnis ist die Attributliste, wie sie an die Variable `TV("!attributes")` gebunden ist. Diese Variable wird von der Strategiesemantik initialisiert.
Der Subtermoperator `"/"` wird durch die nächsten beiden Regeln implementiert, deren Ähnlichkeit zur normalen Definition des Subterms in Abschnitt 3 einen weiteren Kommentar überflüssig macht.
Gleiches gilt für die beiden Regeln (195/196) und die Subtermersetzung.
Die Regeln (197) bis (202) definieren die Gleichheitsfunktion `"=="`. Da Gleichheit in Quetzal lazy ist, müssen die beiden Terme a, b auf Kopfnormalform gebracht und dann verglichen werden. Die Regeln tun dies für alle möglichen Fälle.
Die Regel (203) definiert Ungleichheit über die Boolsche Negation `"~"` und die Gleichheitsfunktion `"=="`.
Die Funktion `"show"` erzeugt zu jedem Term die Darstellung seiner Normalform als $\mathcal{QIR}$ Zeichenkette (Liste von Zeichen). Sie wird in Regel (204) eingeführt.

Die Funktion "dom", welche die Domain einer Substitution liefert wird in Regel (205) definiert.
Die Komposition zweier Substitutionen wird mit der Regel (206) beschrieben. Die zweite Substitution t wird dabei auf die Ergebnisse der ersten angewandt (a'_i).
Zuletzt kommt die Beschreibung der Anwendung einer Funktion auf einen Term in den Regeln (207-209). Die erste Regel behandelt dabei den Fall, daß der Term zu einer Konstanten oder einer Variablen auswertet. In der zweiten und dritten Regel wird jeweils der Fall behandelt, daß der Term nur ein Argument oder mehrere Argumente hat.

4.8.7 strategy

Die Semantik einer Strategie (Regel (210)) ist an sich einfach. Zuerst wird aus dem TRS-Objekt trs und dem Taktikobjekt tac eine Umgebung Δ^{trs} berechnet, welche Bindungen für TV("trs") und TV("!attributes") enthält. Dann wird die Ablaufsteuerung mit den Bindungen für die Daten, die lokalen Definitionen und der Bindung TV("goals") $\mapsto$ C($goal$,Nil), der initialen Goalliste, die nur das übergebene Goal $goal$ enthält, aufgerufen. Das Ergebnis ist, neben einer neuen Umgebung Δ_2 ein Resultat res und ein Protokoll $prot$, beides Zeichenketten. Das $maxd$ in der zweiten Klausel bezeichnet die maximale Tiefe, bis zu der gerechnet werden soll, 0 ist die Anfangstiefe. Man beachte, daß es auch eine Bindung TV("goal") $\mapsto$ $goal$ in Δ_1 gibt. Im Gegensatz zu später, wo ein Bezug auf "goal" immer das aktuelle Goal referenziert, kann also in dieser initialen Umgebung das anfängliche Goal referenziert werden.

getout

Die Regeln für **getout**, (211/212) werden später zur Erzeugung der Protokolle/Ergebnisse für die Ereignisse "Lösung", "Sackgasse" und "Tiefenüberschreitung" gebraucht. Sie werten für ein Ereignis str den zugehörigen Ausgabeterm aus, falls in der Strategie einer dafür existiert. Ansonsten ist das Ergebnis die leere Zeichenkette.

calctrs

Die Regeln (213) bis (217) berechnen aus einem TRS- und einem Taktikobjekt die Bindungen für TV("trs") und TV("!attributes"). Zuerst werden die Regelnamen, die im entgültigen TRS vorkommen sollen, abgeleitet: $[n_1, \ldots, n_p]$. Die Vorgehensweise dabei ist: Die Ordnung der Taktik auf die Form 'Name>···>Name>Wildcard> Name>···>Name' bringen, in dem ggf. ein Wildcard an die Ordnung angehängt wird[23]. Dann wird die Ordnung von vorne durchgegangen, bis ein Wildcard erreicht ist. Dabei werden alle Namen übernommen, wenn sie im TRS vorkommen. Danach wird die Ordnung nach dem Wildcard durchgegangen und ebenfalls alle Namen in einer separaten Liste übernommen, die im TRS auftreten. Beim Übernehmen eines Namens wird dieser aus dem TRS mittels $rns - -[n]$ entfernt. Zum Schluß werden die Listen konkateniert: $pres \mathbin{+\!\!+} rns \mathbin{+\!\!+} posts$. In $pres$ wurden

[23]Jede Folge von Regelnamen darf auch leer sein.

die Namen vor dem Wildcard gesammelt, in $posts$ die nach dem Wildcard. rns enthält die noch im TRS verbliebenden Namen.
Die Attributliste berechnet sich einfach: Aus der Attributkomponente der Taktik werden alle Paare herausgenommen, deren erste Komponente ein Name aus dem entgültigen TRS ist. $[as_i^1, \ldots, as_i^{j_i}]$ ist dann die Liste aller Attribute für den Namen n_i. Diese Liste wird in eine $\mathcal{QIR}$ Liste transformiert, ebenso die Namen darin. Das Ergebnis ist dann eine Liste von Paaren `T(`$[n_i', as_i']$`)` von Namen und Attributlisten. Das entgültige TRS ist eine Liste von Namen-/Regelpaaren, die ebenso durch eine Transformation zu $\mathcal{QIR}$ Namen/Listen entstanden ist.

cntrl

Die Regeln (218) bis (221) behandeln die grobe Ablaufsteuerung. Hier werden die Ereignisse 'Goal gelöst', 'Sackgasse' und 'Tiefenüberschreitung' erkannt.
Die erste Regel erkennt, wann ein Goal gelöst ist: Wenn `TV("goals")` an die leere Liste gebunden ist. In diesem Fall wird mittels **getout** die dazugehörige Ausgabe berechnet. Im Protokoll wird `"Solved\0"` vermerkt. Das Zeichen '`\0`' dient zur Kennung des Endes eines Protokolleintrags. Das Ergebnis ist das von **getout** gelieferte.
Regel (219) behandelt den Fall einer Tiefenüberschreitung: $d > md$. Das Protokoll wird mittels **getout** geholt und ausgegeben.
Die folgende Regel behandelt den Fall einer Sackgasse. Wichtig ist hier, daß nun der Selektionsterm sel ausgewertet wird: Er wird auf die Goalliste angewandt und auf Normalform gebracht. Das Ergebnis ist ein Paar (g, gs) aus aktuellem Goal und neuer Restliste. Mit den neuen Bindungen für `TV("goal")` und `TV("goals")` wird nun die Ablaufsteuerung abgearbeitet: $(\Delta'', d, md, s) \overset{\mathsf{cntrl'}}{\vdash} cnt : (\Delta_3, res, prot, dd, st)$. res und $prot$ sind das Ergebnis, bzw. Protokoll der Abarbeitung. dd ist der 'Lebendigkeitszustand' der Abarbeitung. Er kann $DEAD$ sein, falls keine Anweisung der Ablaufsteuerung erfolgreich war, oder $ALIVE$, falls mindestens eine erfolgreich war. st ist ein Flag, um anzuzeigen, ob die Anweisung `Break` aktiv ist. `Break` zeigt an, daß nachfolgende Anweisungen ignoriert werden sollen, bis die nächst umschließende `For` Anweisung wieder erreicht ist. Ergebnis und Protokoll dieser Regel setzen sich zusammen aus dem Ergebnis/Protokoll der Ablaufsteuerung und der Ausgabe von **getout** `"dead"`.
Die letzte Regel (221) behandelt den Fall einer erfolgreichen Abarbeitung der Ablaufsteuerung. Hier ist dann das Ergebnis/Protokoll gleich dem Ergebnis/Protokoll der Ablaufsteuerung.

cntrl'

Die Regeln (222) bis (236) beschreiben die Semantik der Ablaufsteuerungsanweisungen.
Die erste Regel betrifft die `Break` Anweisung. Ihre einzige Auswirkung ist das Setzen des Flags $BROKEN$. Regeln (223/224) betreffen die `Seq` Anweisung, die Sequenzierung zweier Ablaufsteuerungen. Die erste Regel behandelt den Fall, daß die erste Ablaufsteuerung c_1

abgebrochen wurde. Dann wird c_2 einfach ignoriert. Falls c_1 nicht abgebrochen wurde, wird auch c_2 abgearbeitet. Die Sequenz ist dabei 'tot', falls beide 'Controls' tot waren.
Die beiden folgenden Regeln (225/226) behandeln das `Else(`c_1`,`c_2`)` Konstrukt. Wenn c_1 abgebrochen, oder erfolgreich ist, wird c_2 nicht ausgewertet. Wenn c_1 nicht erfolgreich und nicht abgebrochen war, wird c_2 ausgewertet.
Die bedingte Ausführung von Controls wird durch die Regeln (227/228) beschrieben. Die Bedingung *cond* wird auf NF ausgewertet. Ist das Ergebnis `True`, wird die Control c abgearbeitet. Ist das Ergebnis `False`, geschieht nichts weiter.
Die leere Ablaufsteuerung `Empty` hat (verwunderlicherweise) keinerlei Auswirkungen, außer daß sie immer erfolgreich ist.
Die Zuweisung von Werten an Daten wird mit den Regeln (230/231) erledigt. Zuerst wird die Control c abgearbeitet. Ist sie erfolgreich und nicht abgebrochen, werden die Bindungen der Daten entsprechend den Angaben as verändert.
Die nächste Regel (232) behandelt den Fall des Aufrufs einer Metaregel. Das Ergebnis ist einfach das Ergebnis der Semantik der Metaregel, die aufgerufen wird.
Die letzten drei Regeln (233-236) betreffen die `For(`p`,`t`,`c`)` Anweisung. Ihre Semantik ist es, nacheinander die Element der Liste t gegen p zu matchen und im Erfolgsfall mit den sich ergebenden zusätzlichen Bindungen für die Variablen in p, die Control c abzuarbeiten, bis die Liste t leer ist, oder c abgebrochen wurde.
Die erste Regel (233) trifft den Fall, daß die Liste t leer ist. Das bedeutet den Fehlschlag der Control.
Wenn die Liste zwar nicht leer ist, das Matchen von p auf das Kopfelement der Liste aber nicht möglich ist, wird der Vorgang mit der Restliste wiederholt. Man beachte, daß die Variablen in p und c umbenannt werden müssen, um Kollisionen mit schon vorhandenen Variablen zu vermeiden.
Regel (235) betrifft den Fall, daß der Match erfolgreich war, die Control c' jedoch abgebrochen wurde. Jetzt wird die Restliste nicht mehr betrachtet, sondern die Ablaufsteuerung endet.
Die letzte Regel behandelt den verbleibenden Fall eines erfolgreichen Matches/Abarbeitung von c'. Jetzt wird der Vorgang mit der Restliste wiederholt.

mr

Bei der Semantik einer Metaregel muß man 4 Fälle unterscheiden:

- Das Regelpattern paßt nicht auf den Argumentterm.
- Das Regelpattern paßt, aber das Goalpattern paßt nicht auf `TV("goal")`.
- Regel- und Goalpattern passen, aber die Bedingung der Regel wertet zu `False` aus.
- Regel-/Goalpattern passen, die Bedingung wertet zu `True` aus. Dann ist die Metaregel erfolgreich und kann abgearbeitet werden.

Die Regeln (237) bis (240) behandeln gerade diese 4 Fälle.
Die erste Regel trifft den Fall eines nicht passenden Regelpatterns, die zweite den eines passenden Regelpatterns und nicht passenden Goalpatterns.
In Regel (239) passen beide Pattern, allerdings wertet die Bedingung zu `False` aus. Man beachte, daß die Variablen der Metaregel immer konsistent umbenannt werden müssen, um Kollisionen zu vermeiden. Die lokalen Definitionen sind erst ab der Auswertung der Bedingung bekannt.
Die letzte Regel betrifft den interessanten Fall einer anwendbaren Metaregel. Zuerst wird jetzt der Ausgabeterm berechnet: $\Delta_4 \overset{\mathsf{NF}}{\vdash} ot' \Rightarrow otr : \Delta_5$. Danach wird die Konkatenation des Ergebnisses der Metaregel und der aktuellen Restgoalliste zu NF ausgewertet. Mit der neuen Bindung `TV("goals")` $\mapsto resr$ werden dann die Zuweisungen zu Normalform gerechnet. Danach werden die Bindungen der Daten aktualisiert: $\tilde{\Delta}$. Mit dieser neuen Umgebung wird dann die Ablaufsteuerung wiederum abgearbeitet, nur mit einer um eins erhöhten Tiefe. Da die lokalen Daten davon unbeeinflußt bleiben müssen, werden sie durch die Definition von $\hat{\Delta}$ wieder auf die Werte gesetzt, die sie vor Auswertung der Ablaufsteuerung hatten. Man beachte, daß `"goals"` und `"goal"` ebenfalls als lokale Daten angesehen werden, nicht jedoch `"trs"`, `"!attributes"`. Das Ergebnis/Protokoll setzt sich aus dem Protokoll/Ergebnis der Ablaufsteuerung und dem berechneten Ausgabeterm zusammen. $show(otr)$ wandelt dabei den $\mathcal{QIR}$-Term zu einem String um.
Bemerkenswert ist, daß bei der Berechnung der Zuweisungen die *alten* Werte der Daten[24] verfügbar sind.

4.8.8 solve

Die Semantik des Lösens eines Goals prüft erst die statische Korrektheit des Programms und wendet dann die Semantik der übergebenen Strategie an.

4.8.9 eval

Die Semantik der Auswertung eines Terms prüft erst die statische Korrektheit des Programms und des Terms und reduziert diesen dann auf NF.

4.9 Lemmata

Dieser Abschnitt enthält einige Lemmata, die Eigenschaften eines $\mathcal{QIR}$-Programmes betreffen.
Wir gehen im folgenden immer davon aus, daß ein $\mathcal{QIR}$-Programm p gegeben ist, für das $\overset{\mathsf{prog}}{\vdash} p : (\Gamma, \Delta, \Theta)$ gilt.
Wenn nun von der Signatur Σ die Rede ist, dann ist die Signatur gemeint, die sich ganz natürlich aus der Gesamtheit der Definitionen für Sorten, Constructoren und Functoren

[24] Abgesehen von `TV("goals")`

ergibt. Ebenso betrachten wir dann $\mathcal{QIR}$-Terme $t \in OpPattern$ als Gebilde im Kontext einer Signatur.

Lemma 4.1: (Σ-Terme)

Falls $\Gamma \overset{\textsf{sort-lt}}{\vdash} s_1 : s_2$, dann gilt $s_2 \leq_\Sigma s_1$. Weiter: Falls $(\Gamma, \Theta_1) \overset{\textsf{sort-of}}{\vdash} t : (s, \Theta_2)$, dann gilt: $t \in \mathcal{T}(\Sigma, \mathcal{X})$, $sort(t) = s$. Hierbei wird $\mathcal{X} : s' := \{n | n \mapsto s' \in \Theta_2\}$ definiert. Auch gilt: $\mathcal{D}(\Theta_2) \setminus \mathcal{D}(\Theta_1) = \mathcal{V}(t) \setminus \mathcal{D}(\Theta_1)$.

Beweis: Die erste Behauptung ist nach Betrachtung der Regeln (20ff.) für sort-lt sofort einsichtig. Die zweite Behauptung läßt sich per Induktion über t beweisen:

- $t = \texttt{V(}n\texttt{)}$. Folgt sofort mit Hilfe der Regeln (23f.).
- $t = \texttt{UOp(}n\texttt{)}$. Offensichtlich.
- $t = \texttt{App(UOp(}n\texttt{),T(}[a_1, \ldots, a_k]\texttt{))}$. Per Induktionsannahme haben wir, daß $a_i \in \mathcal{T}(\Sigma, \mathcal{X})$, $sort(a_i) = s_i$. Aus $\Gamma \overset{\textsf{sort-lt}}{\vdash} s'_i : s_i$ folgt dann: $a_i \in \mathcal{T}(\Sigma, \mathcal{X}) : s'_i$. Damit ergibt sich die Behauptung.

Lemma 4.2: (Konsistenz)

Für alle Objekte in p gilt:

1. Alle Userobjekte referenzieren nur Objekte der richtigen Kategorie, die auch definiert sind.
2. Für Regeln/Goals gilt außerdem:
   (a) Die Argumentterme $l \in LOpPattern$, $r \in OpPattern$ repräsentieren Terme aus $\mathcal{T}(\Sigma, \mathcal{X})$.
   (b) Für eine Regel $l \to r$ gilt: $sort(r) \leq_\Sigma sort(l)$, $l \notin \mathcal{X}$, $root(l) \in \mathcal{F}$
   (c) Für ein Goal $l =_? r$ gilt: $sort(l) \leq_\Sigma sort(r) \vee sort(r) \leq_\Sigma sort(l)$
3. Funktionen referenzieren nur Objekte, die auch eine Definition haben.
4. Strategien referenzieren nur Objekte, die eine Definition haben. Namen, denen zugewiesen wird, bezeichnen Daten.
5. Ein Term t mit $(\emptyset, \Gamma) \overset{\textsf{termsynt}}{\vdash} t : fns$, $(\Gamma, \Theta) \overset{\textsf{termtype}}{\vdash} t : \tau$, $fns \subseteq \mathcal{D}(\Delta_{|\texttt{Function}})$ referenziert nur definierte Objekte.

Beweis:

1. Der Beweis ergibt sich sofort aus den Bedingungen der Inferenzregeln.
2. (a) Mit Lemma 4.1 klar.
   (b) Anhand der Bedingung der Regeln (14f.) folgt sofort $root(l) \in \mathcal{F}$. Offensichtlich $sort(l) = s_1, sort(r) = s_2$ und ebenso nach Lemma 4.1 $\mathcal{V}(r) \setminus \mathcal{D}(\Theta_1) = \emptyset$. Da $\mathcal{V}(l) = \mathcal{D}(\Theta_1)$, folgt $\mathcal{V}(r) \subseteq \mathcal{V}(l)$, wie behauptet. Aus $\Gamma \overset{\textsf{sort-lt}}{\vdash} s_1 : s_2$ folgt dann schließlich $sort(r) = s_2 \leq_\Sigma s_1 = sort(r)$.

(c) Ähnlich trivial anhand der Regeln (16ff.).

3. Da mit $(v,\Gamma)\overset{\mathsf{termsynt}}{\vdash} t : ns$ sichergestellt ist, daß t nur Objekte aus Γ, v und FOPs aus ns referenziert, ist mit Regeln (55-58) die Behauptung klar.
4. Ähnlich. Für Assigns gilt die Behauptung anhand Regeln (130/131).
5. Offensichtlich.

Das folgende Lemma beschreibt die semantische Korrektheit einer Rechnung: Ein reduzierter Term hat denselben Typ, wie der ursprüngliche Term. Der Beweis vermeidet eine detailierte Argumentation. Wer daran interessiert ist, schlage in [Sch94, S. 175] oder [Mil78] nach, wo diese Eigenschaft für den λ-Kalkulus bewiesen wird. Die Eigenschaft überträgt sich auf das in Quetzal verwandte Reduktionssystem.

Lemma 4.3: (Subject-Reduction)
Falls t ein wohlgeformter Term ist, $(\Gamma,\Theta)\overset{\mathsf{termtype}}{\vdash} t : \tau$ und $\Gamma\overset{foo}{\vdash} t \Rightarrow t' : \Gamma'$, dann gilt $(\Gamma',\Theta')\overset{\mathsf{termtype}}{\vdash} t' : \tau$, wobei Θ' nur Bindungen für $\mathtt{TV}(n) \notin t$ enthält. $foo \in \{\mathsf{NF}, \mathsf{HNF}, \mathsf{prim}, \mathsf{rewrite}, \ldots\}$.

Beweis: Offensichtlich reicht es dies für $t \neq t'$ und $foo \in \{\mathsf{rewrite}, \mathsf{share}, \mathsf{prim}\}$ zu zeigen.

1. share. Offensichtlich. Man führt in Θ' einfach die Bindungen der neuen $\mathtt{TV}(n)$ an die Typen der Ausdrücke, die sie ersetzen, ein.
2. rewrite. Nur der Fall eines erfolgreichen Rewrites muß betrachtet werden. Zuerst kann man zeigen: Wenn $\Gamma\overset{\mathsf{matches}}{\vdash} p \downarrow t : OK(\Gamma', \bar{t})$, dann $(\Gamma, \bar{\Theta})\overset{\mathsf{pattype}}{\vdash} p : (\tau, \Theta^p)$. Dies ist nach Betrachtung der Regeln für matches klar. Man muß dies übrigens nur für den Fall zeigen, daß $t = \mathtt{FOp}(n)$, oder $t = \mathtt{App(FOp(}n\mathtt{)},t'\mathtt{)}$, wobei t' in NF ist. Da die Gleichung, die zur Reduktion verwandt wird, wohlgeformt ist, hat das Ergebnis denselben Typ, wie das Pattern, also wird der Term t durch einen Term t'' desselben Typs ersetzt, womit die Behauptung bewiesen wäre.
3. prim. Einfache Durchsicht durch die Regeln. Das Ergebnis stimmt immer mit dem geforderten Typ überein.

Lemma 4.4: (Needed-evaluation)
Falls ein Term geshart ist, d.h. es gibt eine Bindung $\mathtt{TV}(n) \mapsto t' \in \Gamma$, dann wird bei der Auswertung jedes Terms t, der Verweise $\mathtt{TV}(n)$ enthält, t' höchstens einmal ausgewertet.

Beweis: Da man nie gleichzeitig zwei Verweise auswerten kann, muß ein Verweis in t der erste sein, der ausgewertet wird. Dies geschieht immer, indem t' auf HNF gebracht wird, in einer Umgebung, die keine Bindung für $\mathtt{TV}(n)$ enthält. Eine erfolgreiche Reduktion auf HNF kann also keinen Zugriff auf $\mathtt{TV}(n)$ beinhalten. Nach der Reduktion auf HNF gibt es zwei Fälle: Einmal kann der reduzierte Term eine irreduzible

Konstante sein. Zum anderen kann er ein Term mit Argumenten sein. Im ersten Fall kann jede weitere Auswertung des Verweises `TV(`n`)` nichts mehr auswerten, da `TV(`n`)` jetzt an die Konstante gebunden ist. Im zweiten Fall werden die Argumente durch neue Verweise ersetzt, und die Behauptung ergibt sich durch Induktion.

Kapitel 5

Die Abstrakte Maschine

Dieses Kapitel beschreibt sowohl die Abstrakte Maschine, als auch die Übersetzung von $\mathcal{QIR}$ zu Code für diese Maschine. Das Kapitel gliedert sich, wie folgt: In Abschnitt 5.1 werden die Komponenten der $\mathcal{QAM}$ vorgestellt. Der nächste Abschnitt enthält die Definition der Semantik von $\mathcal{QAM}$-Programmen und Befehlen. In Abschnitt 5.3 wird dann die Übersetzung von der $\mathcal{QIR}$ zu $\mathcal{QAM}$-Programmen vorgestellt, womit das Kapitel endet.

5.1 Der Aufbau der $\mathcal{QAM}$

5.1.1 Die Komponenten der $\mathcal{QAM}$

Die $\mathcal{QAM}$ besteht aus sieben Komponenten:

1. Dem *Heap*. In ihm werden alle Objekte angelegt, die ein $\mathcal{QAM}$-Programm erzeugt.
2. Dem *Stack*. Auf ihm werden lokale Daten und Informationen verwaltet.
3. Dem *Code*. Er enthält den gesamten $\mathcal{QAM}$-Code.
4. Dem *Code-Pointer* CP. Er zeigt auf die aktuelle Anweisung im Code.
5. Dem *Global-Pointer* GP. Er zeigt auf die globalen Daten im Stack.
6. Dem *Local-Pointer* LP. Er zeigt auf die lokalen Daten im Stack.
7. Dem *Stack-Pointer* SP. Er zeigt auf die oberste Position im Stack.

Der Stack wächst zu höheren Adressen hin, d.h. um etwas auf den Stack zu legen, muß SP erhöht werden.
Auf dem Stack werden abgelegt:

- *Basiswerte*: Zahlen, Zeichen, Bool'sche Werte: $\mathsf{num}(i)$, $\mathsf{char}(c)$, true, false

- *Heapadressen*: p
- *Codelabel*: l
- *Stackadressen*: s
- *'Schwarze Löcher'*: ?

Der Heap enthält *Zellen*. Jede Zelle wird über ihre *Adresse* angesprochen. Eine Zelle kann enthalten:

- Einen Basiswert.
- Eine leere Liste: nil
- Eine Applikation: app(p_1,p_2)
- Ein Cons: c(p_1,p_2)
- Ein Tupel: t($[p_1, \ldots, p_k]$)
- Eine fehlgeschlagene Substitution: failed
- Eine Substitution: subst($[(p_1, p'_1), \ldots, (p_k, p'_k)]$)
- Einen Constructor: dop(n)
- Einen Useroperator: uop(n)
- Eine Uservariable: v(n)
- Ein Funktionssymbol: fop(n,a,cl)
- Eine Indirektion: ind(p), ind(?)
- Ein 'schwarzes Loch': ?

Hierbei stehen p, p_i, p'_i für Heapadressen, n für natürliche Zahlen und cl für Codelabel.
Die Ähnlichkeit der möglichen Zelleninhalte zu Konstrukten aus der $\mathcal{QIR}$ ist evident. Nur `FOp(n)`, `DOp(n)`, `UOp(n)` und `V(n)` Konstrukte werden anders repräsentiert. Die Namen der Konstrukte in der $\mathcal{QIR}$ werden auf Zahlen abgebildet[1]. Ein fop(n,a,cl) hat zwei zusätzliche Argumente: a, die Stelligkeit des Funktionssymbols (0 oder 1) und cl, ein Codelabel, das den Code für die Funktion referenziert.
Ein schwarzes Loch '?' steht für eine noch unbekannte Heapadresse, die erst später eingesetzt werden kann. Dies wird später bei der Konstruktion von lokalen Variablen bei der Auswertung von Funktionen benutzt.

[1]Indexe in eine Symboltabelle, wenn man so will.

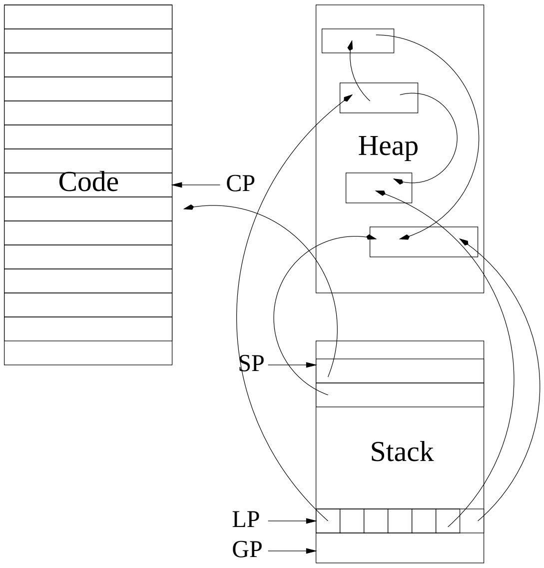


Abbildung 5.1: Aufbau der $\mathcal{QAM}$

LP und GP zeigen im Stack auf die lokalen/globalen Daten, die als Folge von Stackinhalten vorliegen. CP zeigt auf den aktuellen $\mathcal{QAM}$-Befehl im Code.
Ein Befehl im Code kann auch ein *Label* bekommen. Dies dient einfach zur Markierung des Befehls[2], etwa: evalnf: push(2). Sprünge können sich ebenfalls auf diese Label beziehen, anstatt die Codeadresse anzugeben: call evalnf.
Abbildung 5.1 verdeutlicht noch einmal den Aufbau der $\mathcal{QAM}$.

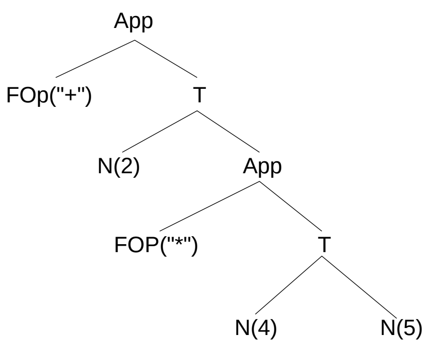
5.1.2 Erläuterung der $\mathcal{QAM}$

Der Aufbau der $\mathcal{QAM}$ ist an verschiedene Maschinen angelehnt, die zur Implementierung von funktionalen Sprachen benutzt werden, etwa die G-Maschine ([PJ86, PJL92]), oder die in [WM92, Kap. 3.5] vorgestellte Maschine. Zusätzlich enthält sie noch einige Erweiterungen, die zur Implementierung der globalen/lokalen Daten einer Strategie und zum Handling von Substitutionen dienen. Der Gedanke hinter der $\mathcal{QAM}$ ist einfach: Jede Funktion wird zu einem kleinen Programm übersetzt, das immer dann ausgeführt wird, wenn diese Funktion reduziert wird. Das Programm baut dann explizit das Ergebnis auf, die $\mathcal{QAM}$ besitzt Kommandos, die Objekte im Heap konstruieren und deren Adressen auf den Stack legen, und ersetzt dann den zu reduzierenden Term durch das so aufgebaute Ergebnis. Die einzelnen Befehle sind sehr simpel und können schnell ausgeführt werden. Dabei sind sie aber noch so abstrakt, daß die Übersetzung zu solchen Programmen einfach bleibt. Ein weiterer Vorteil dieser Vorgehensweise ist, daß man altbekannte Optimierungen, wie sie für die Übersetzung imperativer Sprachen benutzt werden, ebenfalls anwenden

[2] Oder des hier beginnenden Programmstücks

kann, um die Effizienz noch zu steigern.
Die $\mathcal{QAM}$ hat einige etwas komplexere Befehle, um Kopien von Strukturen anzulegen, Substitutionen anzuwenden, etc. Die Semantik ist aber immer noch einfacher, als die ursprüngliche Semantik dieser Operationen in der dynamischen Semantik. Alles in allem erlaubt der Befehlssatz eine effiziente Implementierung, die aber dennoch 'straight' ist, in dem Sinne, daß sie unkompliziert ist und keine verwickelten Abhängigkeiten berücksichtigen muß.
Ein $\mathcal{QIR}$-Programm wird in ein großes $\mathcal{QAM}$-Programm übersetzt, wobei jede Funktion und jede Strategie als ein Unterprogramm realisiert sind. Zusätzlich gibt es ein fixes Stück Programmcode, das den Code für die primitiven Funktionen und einige Routinen enthält, welche etwa die Reduktion auf Normalform leisten.
Soll z.B. ein Ausdruck t ausgewertet werden (Semantik von eval), dann wird dieser Ausdruck ebenfalls in ein $\mathcal{QAM}$-Programm übersetzt, welches gerade den Term t im Heap baut und den Verweis darauf auf den Stack legt, anschließend wird dir Routine evalnf aufgerufen, die die Reduktion zu Normalform erledigt, das Ergebnis liegt danach anstelle des alten Verweises auf dem Stack und wird mit dem $\mathcal{QAM}$-Befehl out(1,"","") ausgegeben, danach wird die Maschine angehalten. Beispiel 5.1 zeigt etwa den resultierenden Code, wie er für den Ausdruck '$2 + 4 * 5$' erzeugt wird, oder in $\mathcal{QIR}$-Darstellung: `App(FOp("+"),T([N(2),App(FOp("*"),T([N(4),N(5)]))]))`, Beispiel 5.1 zeigt diesen Term etwas anschaulicher in Baumform.

Beispiel 5.1: $\mathcal{QDL}$-Term,$\mathcal{QAM}$-Code




```
mkconst(num(5))
mkconst(num(4))
pushb(num(2))
mktupel
mkconst(fop(5,1,function_5))
mkapp
mkconst(num(2))
pushb(num(2))
mktupel
mkconst(fop(3,1,function_3))
mkapp
call evalnf
out(1,"","")
stop
```

Die ersten vier Befehle konstruieren die Argumente für den '*' Operator, beginnend mit dem letzten Argument. Das pushb(num(2)) gibt dabei die Anzahl der Argumente, die

in dem `T(...)` stehen. Der fünfte Befehl (mkconst(fop(5,1,function_5))) konstruiert den Funktionsoperator. Nämliches geschieht in den folgenden vier Befehlen, das letzte **mkapp** baut dann die Wurzel des Terms. Der folgende Aufruf der Routine evalnf reduziert den Term auf seine Normalform. Anschließend wird er mit out(1,"","") in den Protokollkanal ausgegeben[3]. Das abschließende stop hält die Maschine an.
Die Befehle legen jeweils die Adressen der von ihnen konstruierten Heapobjekte auf dem Stack ab und nehmen auch Argumente vom Stack. Bei der Abarbeitung dieses Programms verändert sich der Stack, wie aus der folgenden Tabelle ersichtlich:

Befehl	Stack nach dem Befehl
mkconst(num(5))	p_{n5}
mkconst(num(4))	$p_{n4} \cdot p_{n5}$
pushb(num(2))	num(2)$\cdot p_{n4} \cdot p_{n5}$
mktupel	p_t
mkconst(fop(5,1,function_5))	$p_{fop} \cdot p_t$
mkapp	p_{app}
mkconst(num(2))	$p_{n2} \cdot p_{app}$
pushb(num(2))	num(2)$\cdot p_{n2} \cdot p_{app}$
mktupel	$p_{t'}$
mkconst(fop(3,1,function_3))	$p_{fop'} \cdot p_{t'}$
mkapp	$p_{app'}$
call evalnf	$p_{app'}$
out(1,"","")	
stop	

Die Befehle der $\mathcal{QAM}$ gliedern sich in 9 Gruppen:

- Konstruktion von Heapobjekten
- Destruktion/Traversion von Heapobjekten
- Rechnen
- Sprünge
- Stackmanipulation/Datenzugriff
- Reduktion
- Substitutionen
- Ausgabe/Zeichenumwandlung
- Sonstiges

[3]Genauer gesagt, wird seine Darstellung als String ausgegeben.

5.1.3 Konstruktion von Heapobjekten

mkconst(c) Kreiert ein neues Objekt c im Heap, hinterlegt dessen Adresse auf dem Stack.

mkapp Baut eine app Zelle im Heap, nimmt 2 Argumente vom Stack.

mknum Liefert die Adresse einer neuen num Zelle im Heap, Argument liegt auf dem Stack.

mkuop(n) Konstruiert eine uop(n) Zelle im Heap.

mkdop(n) Baut dop(n) Zelle.

mkfop(n,a,cl) Ebenso für eine fop(n,a,cl) Zelle.

mkv Nimmt Argument num(n) vom Stack, baut eine v(n) Zelle.

mkcons Konstruiert eine c(p_1,p_2) Zelle aus den Argumenten vom Stack.

mktupel Dient zur Konstruktion einer t($[p_1, \ldots, p_n]$) Zelle. Nimmt num(n) und $p_1, \ldots, p_n$ Argumente vom Stack.

mksubst Baut eine Substitutionszelle: subst($[(p_1, p'_1), \ldots, (p_n, p'_n)]$). Nimmt num($2n$) und $p_1, p'_1, \ldots, p_n, p'_n$ Argumente vom Stack.

5.1.4 Destruktion/Traversierung von Heapobjekten

destruct Zerlegt die Zelle, deren Adresse auf dem Stack liegt und legt statt dessen die Anzahl der Argumente der Zelle, sowie die Argumente selbst auf den Stack.

update Dient zur Ersetzung eines Zelleninhaltes durch eine Indirektion auf eine andere Zelle.

clone Nimmt Adresse einer Zelle und konstruiert daraus eine Kopie mit neu alloziierten Zellen im Heap. Die Adresse der Kopie liegt danach auf dem Stack.

match(c) Prüft, ob eine Zelle vom gegebenem Typ c ist.

cmp Prüft, ob der Inhalt zweier Zellen übereinstimmt.

ishnf Prüft, ob der Ausdruck, dessen Adresse auf dem Stack liegt in Kopfnormalform ist.

5.1.5 Rechnen

add Addiert zwei Zahlen, die auf dem Stack liegen und ersetzt sie durch das Ergebnis.

sub Subtraktion zweier Zahlen.

mul Multiplikation.

div Division.

dec Erniedrigt die Zahl, die auf dem Stack liegt um eins, falls sie nicht 0 ist.

inc Erhöht eine Zahl um eins.

eq Liefert *wahr*, falls die beiden Basiswerte auf dem Stack gleich sind.

neq Liefert *wahr*, falls die beiden Basiswerte auf dem Stack nicht gleich sind.

lt Vergleicht die beiden oben liegenden Zahlen anhand der $<$ Relation.

gt Vergleich anhand $>$.

leq Vergleich anhand $\leq$.

geq Vergleich anhand $\geq$.

neg Negiert den oben liegenden Warheitswert.

or Liefert das logische Oder der beiden auf dem Stack liegenden Warheitswerte.

and Liefert das logische Und der beiden Werte.

5.1.6 Sprünge

jt l Springt zu l, falls das oberste Element des Stacks true ist.

jf l Springt zu l, falls das oberste Element false ist.

jump l Springt zu l.

call l Legt die Adresse des nächsten Befehls auf den Stack, springt zu l.

ret Nimmt Adresse l vom Stack, springt dahin.

5.1.7 Stackmanipulation/Datenzugriff

pop(n) Entfernt die obersten n Elemente vom Stack.

popi Entfernt n Elemente vom Stack, wobei num(n) als oberstes auf dem Stack liegt.

swap Tauscht die beiden obersten Elemente aus.

slide(n) Befördert das oberste Stackelement n Positionen nach unten, wobei diese n Elemente entfernt werden.

pushb(c) Legt den Basiswert c auf den Stack.

pushi Legt das n-te Element des Stacks oben auf den Stack. Dabei enthält der Stack vorher als oberstes Element num(n).

push(n) Legt das n-te Element des Stacks wiederum oben auf den Stack.

pushLP Legt den Wert von LP auf den Stack.

load Legt den Inhalt der Adresse, die oben auf dem Stack liegt, statt dessen auf den Stack.

loadLP(n) Legt das n-te Element, ab der Position des LP auf den Stack.

loadGP(n) Nämliches für das n-te Element ab GP.

popLP Legt den obersten Wert auf dem Stack in LP ab.

args(i,j) Legt j Verweise '?' für Patternvariablen auf den Stack, gefolgt von i Adressen neuer Zellen, die mit '?' gefüllt sind. Die i Zellen dienen zur Aufnahme der lokalen Definitionen einer Funktion/Strategie, etc.

store(n) Speichert den obersten Wert auf dem Stack in der n-ten Position.

storei Nimmt num(n) vom Stack und speichert den jetzt oben liegenden Wert in der jetzt $n-1$-ten Position.

storeLP(n) Speichert den oben liegenden Wert in der n-ten Position ab LP.

storeGP(n) Analog.

setLP Lädt den aktuellen Wert von SP in LP.

setGP Analog.

5.1.8 Reduktion

unwind Führt eine Reduktion in dem Ausdruck aus, dessen Adresse auf dem Stack liegt, falls dieser noch nicht in Kopfnormalform ist.

5.1.9 Substitutionen

dom Nimmt die Adresse einer subst$([(p_1,p'_1),\ldots,(p_n,p'_n)])$ Zelle vom Stack und ersetzt sie durch die Adresse einer neu konstruierten Liste c$(p_1,\cdots,$ c$(p_n,$nil$)\cdots)$.

repl Auf dem Stack liegen die Adresse einer subst-Zelle und die Adresse eines Terms. In dem Term wird jedes Vorkommen einer Variablen, die von der Substitution verändert wird, durch den entsprechenden Term ersetzt. Der Term wird nicht kopiert, sondern überschrieben. Sharing wird dabei berücksichtigt, d.h. geteilte Subterme werden nur einmal ersetzt.

compose Auf dem Stack liegen die Adressen zweier subst-Zellen. Sie werden durch die Adresse einer neu konstruierten Substitution ersetzt, welche die Komposition der beiden Substitutionen darstellt.

5.1.10 Ausgabe/Zeichenumwandlung

show Wandelt den Ausdruck, dessen Adresse auf dem Stack liegt in seine Zeichenrepräsentation und baut einen Ausdruck, der eben dieser Zeichenrepräsentation als Liste von Zeichen entspricht. Die Adresse des neuen Ausdrucks liegt danach auf dem Stack.

out(n,pre,$post$) Schickt die Zeichenrepräsentation des Ausdruckes, dessen Adresse auf dem Stack liegt, in Kanal n. Bei $n = 0$: Resultatskanal, $n = 1$: Protokollkanal. Vor dieser Ausgabe wird der String '*pre*' in den Protokollkanal ausgegeben, nach der Ausgabe des Ausdrucks wird der String '*post*' in den Protokollkanal ausgegeben.

5.1.11 Sonstiges

stop Dieser Befehl hält die Maschine an.

error(n) Hält die Maschine mit Fehlerstatus n an,

Die genaue Semantik der Befehle wird im folgenden Abschnitt definiert.

5.2 Semantik

5.2.1 Konventionen

Die Semantik der Befehle wird wieder über die Angabe einer Natural Semantics gegeben, dabei werden die Komponenten der $\mathcal{QAM}$ folgendermaßen repräsentiert:

- Der Stack wird als Folge $e_1 \cdot e_2 \cdot \cdots \cdot e_n \cdot R$ von Elementen e_i und Rest R dargestellt. e_1 ist das oberste Element des Stacks, auf welches SP zeigt. Durch diese Repräsentation sind der Stack und SP zusammengelegt.

- Der Heap wird als (eindeutige) Zuordnung von Adressen auf Objekte geschrieben: $\Gamma = \{p_1 \mapsto o_1, \ldots\}$, wie gehabt.

- GP, LP, CP werden als natürliche Zahlen repräsentiert.

Der Zugriff auf Elemente im Stack, die mittles GP, LP adressiert werden, werden als $[GP/LP - i]$ geschrieben, wobei $i \in \mathbb{N}$. Dies beschreibt das $i + 1$-te[4] Element von LP/GP aus.
Der Zugriff auf Objekte im Heap geschieht über ihre Adresse: $\Gamma(p)$, wie gehabt. Da eine Adresse aber auf eine Indirektion $\mathsf{ind}(p')$ verweisen kann, wird die Funktion $follow : \mathbb{N} \rightarrow \mathbb{N}$ definiert[5]:

$$follow(p) = \begin{cases} follow(p'), \text{ falls } \Gamma(p) = \mathsf{ind}(p') \\ p \qquad\qquad\quad , \text{ sonst} \end{cases}$$

[4]Wobei es natürlich kein '0-tes' Element gibt.

[5]Wobei ihr Ergebnis für Strukturen der Form $\{p \mapsto \mathsf{ind}(p'), p' \mapsto \mathsf{ind}(p)\}$ unbestimmt ist.

Wenn klar ist, welcher Heap Γ gemeint ist, schreiben wir $@p$ für $\Gamma(follow(p))$.
Oft benötigt man eine neue, unbenutzte Adresse, zur Kreierung neuer Objekte. Wir schreiben $p = new(\Gamma)$ für eine Adresse p, die in Γ nicht vorkommt, bzw. $\{p_1, \ldots, p_n\} = new(\Gamma)$ für n neue Adressen, $p_i \neq p_j \iff i \neq j$.
Die Funktion *root* liefert die "Wurzel" eines Zelleninhaltes. Sie ist definiert über:

$$root(c) = \begin{cases} c & \text{, falls } c \in \{\mathsf{num}(n), \mathsf{char}(c), \mathsf{true}, \mathsf{false}, \mathsf{nil}, \mathsf{failed}, \mathsf{dop}(n), \mathsf{uop}(n)\} \\ \mathsf{fop} & \text{, falls } c = \mathsf{fop}(n,a,cl) \\ \mathsf{v} & \text{, falls } c = \mathsf{v}(n) \\ \mathsf{c} & \text{, falls } c = \mathsf{c}(p_1,p_2) \\ \mathsf{app} & \text{, falls } c = \mathsf{app}(p_1,p_2) \\ \mathsf{t} & \text{, falls } c = \mathsf{t}([p_1, \ldots, p_k]) \\ \mathsf{subst} & \text{, falls } c = \mathsf{subst}([p_1, \ldots, p_k]) \end{cases}$$

Um den Befehl im Code *code* an Position CP zu bezeichnen, verwenden wir die Notation $code[CP]$.
Die Semantikregeln beinhalten als impliziten Parameter immer den Programmcode *code*. Da sich dieser aber während der Ausführung des Programmes nicht ändert, wird er nicht explizit mitgeführt.
Als nützlich erweist sich die Funktion if:

$$if(c, e_1, e_2) = \begin{cases} e_1, \text{ falls } c \text{ wahr} \\ e_2, \text{ sonst} \end{cases}$$

5.2.2 Regeln

Programm: progr

(243) $(\Gamma, st, GP, LP) \vdash CP : (\texttt{""}, \texttt{""}, 0)$, falls $code[CP] = \mathsf{stop}$

(244) $(\Gamma, st, GP, LP) \vdash CP : (\texttt{""}, \texttt{""}, n)$, falls $code[CP] = \mathsf{error}(n)$

(245) $$\frac{(\Gamma, st, CP, GP, LP) \overset{\mathsf{com}}{\vdash} com : (\Gamma', st', CP', GP', LP', res, prot) \qquad (\Gamma', st', GP', LP') \vdash CP' : (res', prot', n)}{(\Gamma, st, GP, LP) \vdash CP : (res +\!\!+ res', prot +\!\!+ prot', n)},$$
falls $com = code[CP] \notin \{\mathsf{error}(n), \mathsf{stop}\}$

com

(246) $(\Gamma, st, CP, GP, LP) \vdash \mathsf{mkconst}(c) : (\Gamma', p{\cdot}st, CP+1, GP, LP, \texttt{""}, \texttt{""})$,
wobei $\Gamma' = \Gamma \cup \{(p, c)\}$, $p = new(\Gamma)$

(247) $(\Gamma, p_1{\cdot}p_2{\cdot}st, CP, GP, LP) \vdash \mathsf{mkapp} : (\Gamma', p{\cdot}st, CP+1, GP, LP, \texttt{""}, \texttt{""})$,
wobei $\Gamma' = \Gamma \cup \{(p, \mathsf{app}(p_1,p_2))\}$, $p = new(\Gamma)$

(248) $(\Gamma, \mathsf{num}(i)\cdot st, CP, GP, LP) \vdash \mathsf{mknum} : (\Gamma', p\cdot st, CP+1, GP, LP, "", "")$,
wobei $\Gamma' = \Gamma \cup \{(p, \mathsf{num}(i))\}$, $p = new(\Gamma)$

(249) $(\Gamma, st, CP, GP, LP) \vdash \mathsf{mkuop}(n) : (\Gamma', p\cdot st, CP+1, GP, LP, "", "")$,
wobei $\Gamma' = \Gamma \cup \{(p, \mathsf{uop}(n))\}$, $p = new(\Gamma)$

(250) $(\Gamma, st, CP, GP, LP) \vdash \mathsf{mkdop}(n) : (\Gamma', p\cdot st, CP+1, GP, LP, "", "")$,
wobei $\Gamma' = \Gamma \cup \{(p, \mathsf{dop}(n))\}$, $p = new(\Gamma)$

(251) $(\Gamma, st, CP, GP, LP) \vdash \mathsf{mkfop}(n,a,cl) : (\Gamma', p\cdot st, CP+1, GP, LP, "", "")$,
wobei $\Gamma' = \Gamma \cup \{(p, \mathsf{fop}(n,a,cl))\}$, $p = new(\Gamma)$

(252) $(\Gamma, \mathsf{num}(n)\cdot st, CP, GP, LP) \vdash \mathsf{mkv} : (\Gamma', p\cdot st, CP+1, GP, LP, "", "")$,
wobei $\Gamma' = \Gamma \cup \{(p, \mathsf{v}(n))\}$, $p = new(\Gamma)$

(253) $(\Gamma, p_1\cdot p_2\cdot st, CP, GP, LP) \vdash \mathsf{mkcons} : (\Gamma', p\cdot st, CP+1, GP, LP, "", "")$,
wobei $\Gamma' = \Gamma \cup \{(p, \mathsf{c}(p_1,p_2))\}$, $p = new(\Gamma)$

(254) $(\Gamma, \mathsf{num}(n)\cdot p_1\cdot \cdots \cdot p_n\cdot st, CP, GP, LP) \vdash \mathsf{mktupel} : (\Gamma', p\cdot st, CP+1, GP, LP, "", "")$,
wobei $\Gamma' = \Gamma \cup \{(p, \mathsf{t}([p_1, \ldots, p_n]))\}$, $p = new(\Gamma)$

(255) $(\Gamma,\ \mathsf{num}(2n)\cdot p_1\cdot p_1'\cdot \cdots \cdot p_n\cdot p_n'\cdot st,\ CP,\ GP,\ LP) \vdash \mathsf{mksubst} : (\Gamma',\ p\cdot st,\ CP+1,\ GP,\ LP, "", "")$,
wobei $\Gamma' = \Gamma \cup \{(p, \mathsf{subst}([(p_1, p_1'), \ldots, (p_n, p_n')]))\}$, $p = new(\Gamma)$

(256) $(\Gamma, p\cdot st, CP, GP, LP) \vdash \mathsf{destruct} : (\Gamma', \mathsf{num}(n)\cdot p_1\cdot \cdots \cdot p_n\cdot st, CP+1, GP, LP, "", "")$,

$$\text{wobei}\quad n = \begin{cases} 0\ , \text{falls } @p \in & \{\mathsf{true}, \mathsf{false}, \mathsf{nil}, \mathsf{failed}, \mathsf{char}(c), \mathsf{num}(i), \\ & \mathsf{dop}(n), \mathsf{fop}(n,a,cl), \mathsf{uop}(n)\} \\ 1\ , \text{falls } @p = \mathsf{v}(n),\ @p_1 = \mathsf{num}(n),\ p_1 = new(\Gamma) & \\ 2\ , \text{falls } @p = \mathsf{c}(p_1,p_2) & \\ 2\ , \text{falls } @p = \mathsf{app}(p_1,p_2) & \\ m, \text{falls } @p = \mathsf{t}([p_1, \ldots, p_m]) & \\ m, \text{falls } @p = \mathsf{subst}([(p_1, p_2), \ldots (p_{m-1}, p_m)]) & \end{cases}$$

$$\Gamma' = \begin{cases} \Gamma \cup \{(p_1, \mathsf{num}(n))\}, & \text{falls } @p = \mathsf{v}(n) \\ \Gamma & \text{, sonst} \end{cases}$$

(257) $(\Gamma, p_1\cdot p_2\cdot st, CP, GP, LP) \vdash \mathsf{update} : (\Gamma', st, CP+1, GP, LP, "", "")$,
wobei $\Gamma' = \Gamma \oplus \{(follow(p_1), \mathsf{ind}(p_2))\}$

(258) $(\Gamma, p\cdot st, CP, GP, LP) \vdash \mathsf{clone} : (\Gamma', p'\cdot st, CP+1, GP, LP, "", "")$,
wobei $(\Gamma', p') = clone(\Gamma, p)$

(259) $(\Gamma, p\cdot st, CP, GP, LP) \vdash \mathsf{match}(c) : (\Gamma, b\cdot st, CP+1, GP, LP, "", "")$,
wobei $b = if(root(@p) = c, \mathsf{true}, \mathsf{false})$

(260) $(\Gamma, p_1 \cdot p_2 \cdot st, CP, GP, LP) \vdash \mathsf{cmp} : (\Gamma, b \cdot st, CP+1, GP, LP, "", "")$,
wobei $b = if(root(@p_1) = root(@p_2), \mathsf{true}, \mathsf{false})$

(261) $(\Gamma, p \cdot st, CP, GP, LP) \vdash \mathsf{ishnf} : (\Gamma, b \cdot st, CP+1, GP, LP, "", "")$,
wobei $b = if(ishnf(@p), \mathsf{true}, \mathsf{false})$

(262) $(\Gamma, \mathsf{num}(a) \cdot \mathsf{num}(b) \cdot st, CP, GP, LP) \vdash \mathsf{add} : (\Gamma, \mathsf{num}(a+b) \cdot st, CP+1, GP, LP, "", "")$

(263) $(\Gamma, \mathsf{num}(a) \cdot \mathsf{num}(b) \cdot st, CP, GP, LP) \vdash \mathsf{sub} : (\Gamma, \mathsf{num}(max(0, a-b)) \cdot st, CP+1, GP, LP, "", "")$

(264) $(\Gamma, \mathsf{num}(a) \cdot \mathsf{num}(b) \cdot st, CP, GP, LP) \vdash \mathsf{mul} : (\Gamma, \mathsf{num}(a * b) \cdot st, CP+1, GP, LP, "", "")$

(265) $(\Gamma, \mathsf{num}(a) \cdot \mathsf{num}(b) \cdot st, CP, GP, LP) \vdash \mathsf{div} : (\Gamma, \mathsf{num}(a \,\mathbf{div}\, b) \cdot st, CP+1, GP, LP, "", "")$,
falls $b \neq 0$

(266) $(\Gamma, \mathsf{num}(a) \cdot st, CP, GP, LP) \vdash \mathsf{dec} : (\Gamma, \mathsf{num}(max(0, a-1)) \cdot st, CP+1, GP, LP, "", "")$

(267) $(\Gamma, \mathsf{num}(a) \cdot st, CP, GP, LP) \vdash \mathsf{inc} : (\Gamma, \mathsf{num}(a+1) \cdot st, CP+1, GP, LP, "", "")$

(268) $(\Gamma, e_1 \cdot e_2 \cdot st, CP, GP, LP) \vdash \mathsf{eq} : (\Gamma, b \cdot st, CP+1, GP, LP, "", "")$,
wobei $b = if(e_1 = e_2, \mathsf{true}, \mathsf{false})$

(269) $(\Gamma, e_1 \cdot e_2 \cdot st, CP, GP, LP) \vdash \mathsf{neq} : (\Gamma, b \cdot st, CP+1, GP, LP, "", "")$,
wobei $b = if(e_1 = e_2, \mathsf{false}, \mathsf{true})$

(270) $(\Gamma, \mathsf{num}(a) \cdot \mathsf{num}(b) \cdot st, CP, GP, LP) \vdash \mathsf{lt} : (\Gamma, b \cdot st, CP+1, GP, LP, "", "")$,
wobei $b = if(a < b, \mathsf{true}, \mathsf{false})$

(271) $(\Gamma, \mathsf{num}(a) \cdot \mathsf{num}(b) \cdot st, CP, GP, LP) \vdash \mathsf{gt} : (\Gamma, b \cdot st, CP+1, GP, LP, "", "")$,
wobei $b = if(a > b, \mathsf{true}, \mathsf{false})$

(272) $(\Gamma, \mathsf{num}(a) \cdot \mathsf{num}(b) \cdot st, CP, GP, LP) \vdash \mathsf{leq} : (\Gamma, b \cdot st, CP+1, GP, LP, "", "")$,
wobei $b = if(a \leq b, \mathsf{true}, \mathsf{false})$

(273) $(\Gamma, \mathsf{num}(a) \cdot \mathsf{num}(b) \cdot st, CP, GP, LP) \vdash \mathsf{geq} : (\Gamma, b \cdot st, CP+1, GP, LP, "", "")$,
wobei $b = if(a \geq b, \mathsf{true}, \mathsf{false})$

(274) $(\Gamma, b \cdot st, CP, GP, LP) \vdash \mathsf{neg} : (\Gamma, b' \cdot st, CP+1, GP, LP, "", "")$,
wobei $b' = if(b = \mathsf{true}, \mathsf{false}, \mathsf{true})$

(275) $(\Gamma, b_1 \cdot b_2 \cdot st, CP, GP, LP) \vdash \mathsf{or} : (\Gamma, b' \cdot st, CP+1, GP, LP, "", "")$,
wobei $b' = if(b_1 = \mathsf{true}, \mathsf{true}, b_2)$

(276) $(\Gamma, b_1 \cdot b_2 \cdot st, CP, GP, LP) \vdash$ and : $(\Gamma, b' \cdot st, CP+1, GP, LP, "", "")$,
wobei $b' = if(b_1 = \mathsf{false}, \mathsf{false}, b_2)$

(277) $(\Gamma, b \cdot st, CP, GP, LP) \vdash$ jt l : $(\Gamma, st, CP', GP, LP, "", "")$,
wobei $CP' = if(b = \mathsf{true}, l, CP+1)$

(278) $(\Gamma, b \cdot st, CP, GP, LP) \vdash$ jf l : $(\Gamma, st, CP', GP, LP, "", "")$,
wobei $CP' = if(b = \mathsf{false}, l, CP+1)$

(279) $(\Gamma, st, CP, GP, LP) \vdash$ jump l : $(\Gamma, st, l, GP, LP, "", "")$

(280) $(\Gamma, st, CP, GP, LP) \vdash$ call l : $(\Gamma, (CP+1) \cdot st, l, GP, LP, "", "")$

(281) $(\Gamma, l \cdot st, CP, GP, LP) \vdash$ ret : $(\Gamma, st, l, GP, LP, "", "")$

(282) $(\Gamma, e_1 \cdot \cdots \cdot e_n \cdot st, CP, GP, LP) \vdash$ pop(n) : $(\Gamma, st, CP+1, GP, LP, "", "")$

(283) $(\Gamma, \mathsf{num}(n) \cdot e_1 \cdot \cdots \cdot e_n \cdot st, CP, GP, LP) \vdash$ popi : $(\Gamma, st, CP+1, GP, LP, "", "")$

(284) $(\Gamma, e_1 \cdot e_2 \cdot st, CP, GP, LP) \vdash$ swap : $(\Gamma, e_2 \cdot e_1 \cdot st, CP+1, GP, LP, "", "")$

(285) $(\Gamma, e \cdot e_1 \cdot \cdots \cdot e_n \cdot st, CP, GP, LP) \vdash$ slide(n) : $(\Gamma, e \cdot st, CP+1, GP, LP, "", "")$

(286) $(\Gamma, st, CP, GP, LP) \vdash$ pushb(c) : $(\Gamma, c \cdot st, CP+1, GP, LP, "", "")$

(287) $(\Gamma, \mathsf{num}(n) \cdot e_1 \cdot \cdots \cdot e_{n-1} \cdot st, CP, GP, LP) \vdash$ pushi : $(\Gamma, e_{n-1} \cdot e_1 \cdot \cdots \cdot e_{n-1} \cdot st, CP+1, GP, LP, "", "")$

(288) $(\Gamma, e_1 \cdot \cdots \cdot e_n \cdot st, CP, GP, LP) \vdash$ push(n) : $(\Gamma, e_n \cdot e_1 \cdot \cdots \cdot e_n \cdot st, CP+1, GP, LP, "", "")$

(289) $(\Gamma, st, CP, GP, LP) \vdash$ pushLP : $(\Gamma, LP \cdot st, CP+1, GP, LP, "", "")$

(290) $(\Gamma, p \cdot st, CP, GP, LP) \vdash$ load : $(\Gamma, @p \cdot st, CP+1, GP, LP, "", "")$,
falls $@p$ ein Basiswert ist.

(291) $(\Gamma, st, CP, GP, LP) \vdash$ loadLP(n) : $(\Gamma, [LP-n+1] \cdot st, CP+1, GP, LP, "", "")$

(292) $(\Gamma, st, CP, GP, LP) \vdash$ loadGP(n) : $(\Gamma, [GP-n+1] \cdot st, CP+1, GP, LP, "", "")$

(293) $(\Gamma, s \cdot st, CP, GP, LP) \vdash$ popLP : $(\Gamma, st, CP+1, GP, s, "", "")$

(294) $(\Gamma, st, CP, GP, LP) \vdash$ args(i,j) : $(\Gamma', p_1 \cdot \cdots \cdot p_i \cdot e_1 \cdot \cdots \cdot e_j \cdot st, CP+1, GP, LP, "", "")$,
wobei $\{p_1, \ldots, p_i\} = new(\Gamma)$
$e_k = ?$
$\Gamma' = \Gamma \cup \{(p_k, ?) | 1 \leq k \leq i\}$

(295) $(\Gamma, e_1\cdot \cdots \cdot e_n \cdot st, CP, GP, LP) \vdash$ $\mathsf{store}(n)$: $(\Gamma, e_2\cdot \cdots \cdot e_{n-1}\cdot e_1\cdot st, CP + 1, GP, LP,$ "", "")

(296) $(\Gamma,$ $\mathsf{num}(n)\cdot e_2\cdot \cdots \cdot e_n\cdot st,$ $CP,$ $GP,$ $LP) \vdash$ storei : $(\Gamma, e_3\cdot \cdots \cdot e_{n-1}\cdot e_2\cdot st, CP + 1, GP, LP,$ "", "")

(297) $(\Gamma, p\cdot st, CP, GP, LP) \vdash \mathsf{storeLP}(n) : (\Gamma, st', CP + 1, GP, LP,$ "", ""),
wobei st', mit $[LP - n + 1] := p$

(298) $(\Gamma, p\cdot st, CP, GP, LP) \vdash \mathsf{storeGP}(n) : (\Gamma, st', CP + 1, GP, LP,$ "", ""),
wobei st', mit $[GP - n + 1] := p$

(299) $(\Gamma, st, CP, GP, LP) \vdash \mathsf{setLP} : (\Gamma, st, CP + 1, GP, SP,$ "", "")

(300) $(\Gamma, st, CP, GP, LP) \vdash \mathsf{setGP} : (\Gamma, st, CP + 1, SP, LP,$ "", "")

(301) $$\frac{(\Gamma, st, CP) \overset{\mathsf{unwind}}{\vdash} p : (st', CP')}{(\Gamma, p\cdot st, CP, GP, LP) \vdash \mathsf{unwind} : (\Gamma, st', CP', GP, LP, \text{"", ""})}$$

(302) $(\Gamma, p\cdot st, CP, GP, LP) \vdash \mathsf{out}(n, pre, post) : (\Gamma, st, CP + 1, GP, LP, res, prot),$
wobei $res = if(n = 0, out,$ "")
$prot = if(n = 1, pre +\!\!+ out +\!\!+ post, pre +\!\!+ post)$
out = Stringrepräsentation von $@p$

(303) $(\Gamma, p\cdot st, CP, GP, LP) \vdash \mathsf{show} : (\Gamma', p'\cdot st, CP + 1, GP, LP,$ "", ""),
wobei $(\Gamma', p') = show(\Gamma, p)$

(304) $(\Gamma, p\cdot st, CP, GP, LP) \vdash \mathsf{dom} : (\Gamma, \mathsf{num}(n)\cdot p_1\cdot \cdots \cdot p_n\cdot st, CP + 1, GP, LP,$ "", ""),
falls $@p = \mathsf{subst}([(p_1, p'_1), \ldots, (p_n, p'_n)])$ und $\forall_i : @p_i = \mathsf{v}(n_i),\ n_i \neq n_j \iff i \neq j$

(305) $(\Gamma, p_s\cdot p_t\cdot st, CP, GP, LP) \vdash \mathsf{repl} : (\Gamma', p_t\cdot st, CP + 1, GP, LP,$ "", ""),
wobei $\Gamma' = if(p' \neq p_t, \Gamma_r \oplus \{(p_t, \mathsf{ind}(p'))\}, \Gamma_r)$
$(\Gamma_r, p') = replace([(pv_1, pt_1), \ldots, (pv_n, pt_n)], p_t, \Gamma)$
$@p_s = \mathsf{subst}([(pv_1, pt_1), \ldots, (pv_n, pt_n)])$

(306) $(\Gamma, p_1\cdot p_2\cdot st, CP, GP, LP) \vdash \mathsf{compose} : (\Gamma', p\cdot st, CP + 1, GP, LP,$ "", ""),
wobei $\Gamma' = \Gamma_r \cup \{(p, \mathsf{subst}(ps))\}$
$p = new(\Gamma_r)$
$@p_1 = \mathsf{subst}([(pv_1, pt_1), \ldots, (pv_{n_1}, pt_{n_1})])$
$@p_2 = \mathsf{subst}([(pv'_1, pt'_1), \ldots, (pv'_{n_2}, pt'_{n_2})])$
$ts = [(pv_1, \bar{pt}_1), \ldots, (pv_{n_1}, \bar{pt}_{n_1}), (pv'_{i_1}, pt'_{i_1}), \ldots, (pv'_{i_j}, pt'_{i_j})]$
$(\Gamma_r, [\bar{pt}_1, \ldots, \bar{pt}_{n_1}]) = compose(\Gamma, [pt_1, \ldots, pt_{n_1}], [(pv'_1, pt'_1), \ldots, (pv'_{n_2}, pt'_{n_2})])$
$[pv'_{i_1}, \ldots, pv'_{i_j}] = [pv'_k | @pv'_k \notin [@pv_i | 1 \leq i \leq n_1]]$

unwind

(307) $(\Gamma, st, CP) \vdash p : (p{\cdot}st, CP+1)$, falls
$ishnf(@p)$

(308) $(\Gamma, st, CP) \vdash p : ((CP+1){\cdot}p{\cdot}st, l)$, falls
$@p = \mathsf{fop}(n{,}0{,}l)$
oder $@p = \mathsf{app}(p_1{,}p_2)$ und $@p_1 = \mathsf{fop}(n{,}1{,}l)$

(309) $$\frac{(\Gamma, (CP+1){\cdot}p{\cdot}st, \mathsf{code\_traverse}-1) \vdash p_1 : (st', CP')}{(\Gamma, st, CP) \vdash p : (st', CP')}$$, falls
$@p = \mathsf{app}(p_1{,}p_2)$ und $@p_1 \neq \mathsf{fop}(n{,}1{,}l)$

5.2.3 Hilsfunktionen

Die obigen Regeln benutzen einige Hilfsfunktionen, die hier definiert werden.

clone

$clone(\Gamma, p) = (\Gamma \cup \{(p', @p)\}, p')$, falls $@p \in \{\mathsf{num}(n), \mathsf{char}(c), \mathsf{true}, \mathsf{false}, \mathsf{nil}, \mathsf{failed}, \mathsf{uop}(n), \mathsf{v}(n), \mathsf{dop}(n), \mathsf{fop}(n{,}a{,}cl)\}$
, wobei $p' = new(\Gamma)$

$clone(\Gamma, p) = (\Gamma_3, p')$, falls $@p = \mathsf{app}(p_1{,}p_2)$
, wobei $p' = new(\Gamma)$
$(\Gamma_1, p'_1) = clone(\Gamma \cup \{(p', \mathsf{app}(?{,}?))\}, p_1)$
$(\Gamma_2, p'_2) = clone(\Gamma_1, p_2)$
$\Gamma_3 = \Gamma_2 \oplus \{(p', \mathsf{app}(p'_1{,}p'_2))\}$

$clone(\Gamma, p) = (\Gamma_3, p')$, falls $@p = \mathsf{c}(p_1{,}p_2)$
, wobei $p' = new(\Gamma)$
$(\Gamma_1, p'_1) = clone(\Gamma \cup \{(p', \mathsf{c}(?{,}?))\}, p_1)$
$(\Gamma_2, p'_2) = clone(\Gamma_1, p_2)$
$\Gamma_3 = \Gamma_2 \oplus \{(p', \mathsf{c}(p'_1{,}p'_2))\}$

$clone(\Gamma, p) = (\Gamma_n \oplus \Gamma', p')$, falls $@p = \mathsf{t}(p_1, \ldots, p_n)$
, wobei $p' = new(\Gamma)$
$\Gamma_0 = \Gamma \cup \{(p', ?)\}$
$\Gamma' = \{(p', \mathsf{t}([p'_1, \ldots, p'_n]))\}$
$(\Gamma_i, p'_i) = clone(\Gamma_{i-1}, p_i)$, $1 \leq i \leq n$

$clone(\Gamma, p) = (\Gamma_n \oplus \{(p', \mathsf{subst}([(p'_1, q'_1), \ldots, (p'_n, q'_n)])\})\}, p')$

, falls $@p = \mathsf{subst}([(p_1, q_1), \ldots, (p_n, q_n)])$
, wobei $p' = new(\Gamma)$
$\Gamma_0 = \Gamma_0'\Gamma \cup \{(p', ?)\}$
$(\Gamma_i, p_i') = clone(\Gamma_{i-1}', p_i),\ 1 \leq i \leq n$
$(\Gamma_i', q_i') = clone(\Gamma_i, q_i),\ 1 \leq i \leq n$

ishnf

$$ishnf(p) = \begin{cases} wahr \text{ , falls } @p \in & \{\mathsf{true}, \mathsf{false}, \mathsf{nil}, \mathsf{failed}, \mathsf{num}(i), \mathsf{char}(c), \mathsf{c}(p_1,p_2), \\ & \mathsf{t}(ps), \mathsf{subst}(ps), \mathsf{uop}(n), \mathsf{dop}(n), \mathsf{fop}(n,1,cl)\} \\ wahr \text{ , falls } @p = \mathsf{app}(p_1,p_2) \text{ und } ishnf(p_1) \text{ und } @p_1 \neq \mathsf{fop}(n,1,cl) \\ falsch\text{, sonst} \end{cases}$$

show

$show(\Gamma, p) = showstring(\Gamma, pa)$, wobei pa = Zeichenstrom-Repräsentation von $@p$.
$showstring(\Gamma, "c_1, \ldots, c_n") = (\Gamma', p_1)$,
wobei $\Gamma' = \Gamma \cup \ \{(p_i, \mathsf{c}(pc_i,p_{i+1}))|1 \leq i \leq n\} \cup$
$\{(p_{n+1}, \mathsf{nil})\} \cup \{(pc_i, \mathsf{char}(c_i))|1 \leq i \leq n\}$
$\{p_1, \ldots, p_{n+1}, pc_1, \ldots, pc_n\} = new(\Gamma)$

replace

$replace(sub, p, \Gamma) = (\Gamma', p)$, wobei $(\Gamma', ps) = replace'(sub, p, \emptyset, \Gamma)$

$replace'(sub, p, done, \Gamma) = (\Gamma, done)$, falls $@p = \mathsf{uop}(n)$ oder $follow(p) \in done$

$replace'(sub, p, done, \Gamma) = (\Gamma', done \cup \{follow(p)\})$
, falls $@p = \mathsf{v}(n)$ und $follow(p) \notin done$
, wobei $\Gamma' = \begin{cases} \Gamma & \text{, falls } \nexists (pv_j, pt_j) \in sub : @pv_j = \mathsf{v}(n) \\ \Gamma \oplus \{(follow(p), \mathsf{ind}(pt_j))\} & \text{, falls } \exists (pv_j, pt_j) \in sub : @pv_j = \mathsf{v}(n) \end{cases}$

$replace'(sub, p, done, \Gamma) = (\Gamma', done')$, falls $@p = \mathsf{app}(p_1,p_2)$
, wobei $(\Gamma', done') = replace'(sub, p_2, done, \Gamma)$

$replace'(sub, p, done_0, \Gamma_0) = (\Gamma_n, done_n)$, falls $@p = \mathsf{t}([p_1, \ldots, p_n])$
, wobei $(\Gamma_i, done_i) = replace'(sub, p_i, done_{i-1}, \Gamma_{i-1}), 1 \leq i \leq n$

compose

$compose(\Gamma_0, [pt_1, \ldots, pt_n], sub) = (\Gamma_n, [pt_1, \ldots, pt_n])$,
wobei $done_0 = []$
$(\Gamma_i, done_i) = replace'(sub, pt_i, done_{i-1}, \Gamma_{i-1}),\ 1 \leq i \leq n$

5.2.4 Kommentar

progr

Die Semantik eines $\mathcal{QAM}$-Programms bestimmt aus einem Anfangs-Heap einen Ergebnis- und einen Protokollzeichenstrom, sowie einen Fehlerstatus.
Die erste Regel (243) behandelt den Fall einer stop-Anweisung. Sie terminiert ein Programm mit Fehlerstatus 0.
Die nächste Regel behandelt dann den Fall eines Fehlers. Das Kommando error(n) hält die Maschine mit Fehlerstatus n an.
Die letzte Regel (245) definiert die sequentielle Abarbeitung, aufbauend auf der Semantik eines einzelnen Befehls: com.

com

Ein Befehl kann den Heap, den Stack, den Codepointer CP, den Globalpointer GP oder den Localpointer LP verändern, sowie einen Ergebnisstrom und einen Protokollstrom liefern: $(\Gamma, st, CP, GP, LP) \vdash com : (\Gamma', st', CP', GP', LP', res, prot)$
Die Regeln (246) bis (255) beschreiben die Kreierung neuer Objekte im Heap. Nach den einführenden Erläuterungen in Abschnitt 5.1.2 sollten sie verständlich sein.
Regel (256) beschreibt die Auswirkung des destruct-Befehls. Er zerlegt eine Zelle und legt die Anzahl der Argumente der Zelle, sowie die Argumente selbst auf den Stack. Bemerkenswert dabei ist nur der Fall der Zerlegung einer v(n) Zelle. Es wird eine neue Zelle gebaut, die den Index der Variablenzelle in der Form num(n) enthält. Die Adresse dieser Zelle wird als Argument auf den Stack gelegt.
Die folgende Regel definiert das update-Kommando. Das erste Argument p_1 wird auf die Adresse p_2 'umgeleitet', indem die Zelle, auf die p_1 verweist, d.h. einschließlich aller Verweise über ind Zellen, durch die Zelle ind(p_2) überschrieben wird.
Regel (258) behandelt den clone-Befehl. Die Hilsfunktion *clone* generiert aus einem Heap und der Adresse eines Heapobjektes eine Kopie dieses Objektes. Dieser Befehl hat nur dann eine Semantik, wenn das Heapobjekt keine Zyklen enthält, da bei der Kopie das Sharing des ursprünglichen Objektes verlorengeht und ein flaches Objekt erzeugt wird.
Der match(c)-Befehl wird in der folgenden Regel (259) definiert und bietet keine Überraschungen.
Der cmp-Befehl ähnelt dem match-Befehl, nur werden zwei Heapobjekte verglichen.
Regel (261) schließlich behandelt den ishnf-Befehl, der einen Bool'schen Wert auf den Stack legt, der angibt, ob das Heapobjekt, dessen Adresse vorher auf dem Stack lag, in Kopfnormalform ist.
Die Regeln (262) bis (276) definieren die Befehle, mit denen Rechnungen ausgeführt werden können und bedürfen keines weiteren Kommentares.
Die fünf Sprungbefehle werden in Regeln (277) bis (281) behandelt. Die ersten beiden Regeln behandeln die bedingten Sprungbefehle jt, jf. Der dritte den unbedingten Spung mittels jump. Interessant sind nur die Befehle call l und ret. Der erste dient zum Aufruf eines Unterprogrammes und wird häufig Verwendung finden. Er legt die Adresse des

nächsten Befehls ($CP + 1$) auf den Stack und führt das Programm an der Stelle l weiter aus. Der ret-Befehl holt den neuen Wert des CP vom Stack und realisiert damit den Rücksprung aus einem Unterprogramm.
Die folgenden 19 Befehle betreffen die Manipulation des Stacks und den Zugriff auf lokale/globale Daten. Mit Regel (282) wird der pop-Befehl definiert, der eine feste Anzahl von Argumenten vom Stack holt. Der popi-Befehl[6] dient dazu, eine variable Anzahl von Argumenten vom Stack zu holen. Die Anzahl der Argumente, die vom Stack zu entfernen sind, steht als Basiswert num(n) oben auf dem Stack.
Der swap-Befehl vertauscht einfach die beiden obersten Elemente auf dem Stack. Interessanter ist der slide(n)-Befehl, der in Regel (285) definiert wird: Er läßt das oberste Element des Stacks um n Elemente nach unten sacken.
Um Basiswerte auf den Stack zu legen dient der pushb(c)-Befehl, der in Regel (286) beschrieben ist.
Die nächste Regel behandelt den pushi-Befehl, der das Gegenstück zum popi-Befehl darstellt. Er legt ein Element des Stacks oben auf den Stack, dessen Position durch einen oben liegenden Basiswert num(n) gegeben ist. Ist die Position des abzulegenden Elements fest, findet der push(n)-Befehl Verwendung, der in Regel (288) definiert wird.
Die lokalen Daten einer Strategie werden über den LP angesprochen. Diese Daten werden dynamisch verwaltet: Bei der Abarbeitung einer Metaregel werden neue Exemplare dieser Daten angelegt. Nach der Abarbeitung der Metaregel müssen die alten Daten wiederhergestellt werden. Dies wird erledigt, indem der LP auf den Stack gelegt wird um dann später wieder restauriert zu werden. Der pushLP-Befehl legt den LP auf den Stack, wie in Regel 289 beschrieben der popLP-Befehl restauriert den LP wieder (293).
Um einen Basiswert aus einer Heapzelle, deren Adresse auf dem Stack liegt, auf den Stack abzulegen dient der load-Befehl (290).
Der Zugriff auf die lokalen/globalen Daten geschieht mittels der loadLP(n)/loadGP(n)-Befehle, die in den folgenden beiden Regeln definiert werden. Man beachte, daß der Stack zu höheren Adressen hin wächst, der LP/GP zeigen immer auf das Datenelement mit der höchsten Adresse im Stack. Das n-te Element befindet sich also an Adresse $LP/GP-n+1$.
Funktionen, Strategien und Metaregeln erlauben lokale Definitionen. Die in diesen Definitionen eingeführten Variablen werden als Referenzen auf dem Stack angelegt, die Referenz enthält die Adresse eines Heapobjektes. Variablen in Pattern werden ebenfalls als solche Refernzen angelegt. Beim Matching werden dann die Referenzen der Variablen an Subterme gebunden, d.h. in der Referenz auf dem Stack wird die Adresse des Heapobjektes eingetragen, die dem Subterm entspricht. Im Unterschied zu den Patternvariablen müssen für die lokalen Variablen schon vor der Konstruktion der lokalen Definitionen Zellen auf dem Heap alloziiert werden und in die Referenzen eingetragen werden, da die lokalen Definitionen rekursiv sein können. Die alloziierten Zellen enthalten dann Indirektionen der Form ind(?). Der args(i,j)-Befehl (Regel (294)) schafft auf dem Stack Platz für i lokale Variablen und j Patternvariablen. Es werden i neue Heapzellen ind(?) alloziiert.
Um Werte in Stackpositionen zu speichern dienen die store(n) und storei-Befehle (Re-

[6]'pop indirect'

geln (295)f.). Der store(n)-Befehl speichert das oberste Element des Stacks im n-ten Element von oben. Der storei-Befehl nimmt die Position des Zielelements als erstes num(n)-Argument vom Stack.
Die Ablage von lokalen/globalen Daten geschieht mittles der storeLP(n)-, storeGP(n)-Befehle und bietet keine Überraschungen (Regeln (297)f.).
Natürlich muß man auch den Inhalt von LP/GP setzen können. Die Befehle setLP/setGP laden den aktuellen Wert des Stackpointers SP in das entsprechende Register (Regeln (299)f.).
Die Semantik eines unwind-Befehls besteht darin, eine Reduktion in dem Graphen durchzuführen, dessen Adresse auf dem Stack liegt, falls dieser nicht schon in Kopfnormalform ist. Regel (301) beschriebt dies. Der unwind-Befehl kann nur den Stack und den CP verändern: Ggf. wird der Code für eine Funktion ausgeführt und die Rückkehradresse auf den Stack gelegt. Der neue CP ist dann die Adresse des Codes der Funktion. Erst dieser Code baut dann das Ergebnis und ersetzt das Argument dadurch.
Der out(n,*pre*,*post*)-Befehl ist die einzige Quelle von Ausgaben. Er wandelt den Graphen, dessen Adresse auf dem Stack liegt in einen Zeichenstrom um und schickt diesen dann in den angegebenen Ausgabekanal (Regel (302)), wobei der String '*pre*' vorher und der String '*post*' nachher in den Protokollkanal geschickt werden.
Die Umwandlung eines Graphen in eine Zeichenkette, die ebenfalls wieder als Graph vorliegt, wird mit dem show-Befehl vollzogen (Regel (303)). Die Adresse dieses Graphen liegt dann als Ergebnis auf dem Stack. Wie beim out(n,*pre*,*post*)-Befehl muß der Argumentgraph zyklenfrei sein, da die Ausgabe sonst unendlich ist.
Die Operationen auf Substitutionen werden in den Regeln (304) bis (306) behandelt. dom liefert die Domäne einer Substitution als Liste. Die nächste Regel definiert die Anwendung einer Substitution auf einen Graphen. Als Argumente liegen auf dem Stack die Adresse einer Substitutionszelle und die Adresse eines Terms. Die Hilfsfunktion *replace* leistet die Ersetzung, wobei berücksichtigt wird, daß p_t Zyklen beinhalten und ein Graph sein kann.
Die letzte Regel definiert den compose-Befehl zur Komposition zweier Substitutionen. Er benutzt die Hilfsfunktion *compose*, die die zweite Substitution auf die Ergebnisse der ersten anwendet. $[pv'_{i_1}, \ldots, pv'_{i_j}]$ beinhaltet alle Variablen aus der Domäne der zweiten Substitution, die in die Resultatssubstitution übernommen werden, also nicht in der Domäne der ersten Substitution auftauchen.

unwind

Die Semantik des unwind-Befehls wird in den drei Regeln (307ff.) definiert. Die erste Regel behandelt den Fall, daß das Argument bereits in Kopfnormalform ist, dann ist keine Aktion notwendig und der nächste Befehl $CP+1$ kann abgearbeitet werden.
In der zweiten Regel schließlich wird der Fall einer reduzierbaren Funktion betrachtet. Als Ergebnis wird die Adresse des nächsten Befehls auf den Stack gelegt und der CP auf den Anfang des Codes für diese Funktion gesetzt (l).
Die letzte Regel dient zur Traversierung des Argumentes. Wenn es nicht Wurzel eines Redexes ist, dann muß im linken Argument p_1 ein Redex liegen. code_traverse ist dabei die

Adresse folgenden Programmstücks:

```
code_traverse: pop(1)
               ret
```

Durch die Ablage der Adresse des nächsten Befehls auf dem Stack, zusammen mit p wird erreicht, daß nach Abarbeitung der Reduktion von p_1 durch den abschließenden ret-Befehl des Funktionscodes zu code_traverse gesprungen wird. Dieses Programmstück entfernt p_1 vom Stack, daß durch Regel (308) dort abgelegt wurde und springt zu $CP + 1$ zurück.

5.3 Übersetzung $\mathcal{QIR} \rightarrow \mathcal{QAM}$-Code

5.3.1 Einführung

Bei der Übersetzung von $\mathcal{QIR}$-Programmen in $\mathcal{QAM}$-Code muß man einige Abstraktionen vornehmen. Zuerst ist zu beachten, daß die Namen in $\mathcal{QIR}$-Programmen in $\mathcal{QAM}$-Code nicht mehr auftauchen, sondern durch Zahlen ersetzt werden. Wir brauchen also eine Namenszuordnung $\Upsilon : \mathtt{UOp}(Name) \cup \mathtt{V}(Name) \cup \mathtt{DOp}(Name) \cup \mathtt{FOp}(Name) \rightarrow \mathbb{N} \cup \mathbb{N} \times \{0, 1\}$. Die Funktion hat diese Form, da wir für Funktionen auch noch *Stelligkeitsinformation* benötigen. Jede Funktion hat entweder kein Argument[7], oder ein Argument, das dann auch ein Tupel sein kann. Diese Information läßt sich in die Abbildung Υ integrieren, die Funktionsnamen ein Paar $(n, a) \in \mathbb{N} \times \{0, 1\}$ zuordnet. Dieser Anteil von Υ läßt sich aus einem $\mathcal{QIR}$-Programm sofort ablesen: $\Upsilon(name) = (n, 0)$, falls die Definitionen für $name$ alle die Form $(\mathtt{FOp}(name), r, c, ldefs)$ haben, ansonsten $\Upsilon(name) = (n, 1)$. Selbstredend muß Υ auf den Teildomänen $\mathtt{DOp}(Name), \mathtt{FOp}(Name), \mathtt{UOp}(Name), \mathtt{V}(Name)$ injektiv sein.

Ebenfalls brauchen wir eine Zuordnung von Datennamen einer Strategie zu Indices. Dies leistet eine Funktion $\Pi : \mathtt{TV}(Name) \cup Name \rightarrow \mathbb{Z}$, dabei ist $\Pi(\mathtt{TV}(na)) > 0$, falls na ein globaler Name ist und $\Pi(\mathtt{TV}(na)) < 0$, falls na ein lokaler Datenname ist. Bindungen $Name \mapsto \mathbb{N}$ in Π ordnen Namen von Metaregeln einen Index zu.

Bei der Übersetzung muß man lokale Definitionen und Variablen, die über Pattern eingeführt wurden, ebenfalls auf Zahlen abbilden. Eine Funktion $\Sigma : \mathtt{TV}(Name) \rightarrow \mathbb{N}$ leistet dies.

Die Übersetzung von $\mathcal{QIR}$-Programmen zu $\mathcal{QAM}$-Code geschieht wieder mittels einer Natural Semantics und Inferenzregeln der Form $(\Upsilon, \Pi, \Sigma, \ldots) \vdash \mathcal{QIR} \rightarrow \mathcal{QAM}$.

Die genaue Form der einzelnen Funktionen Υ, Π, Σ wird bei den jeweiligen Inferenzregeln definiert.

Die Übersetzung enthält immer einen fixen Codeanteil, welcher eine Reihe von Unterprogrammen enthält, die später benutzt werden. Diesen bezeichnen wir mit Fix$\mathcal{QAM}$. Da er etwas länger ist, wird er später in einem eigenen Abschnitt definiert.

Wir werden oft Codestücke mit Labeln versehen und uns später darauf beziehen. Um die Übersetzung nicht zu verkomplizieren, werden wir die Buchhaltung über diese Label nicht in die Inferenzregeln aufnehmen. Als Konvention gilt, daß Label, die mit l beginnen *lokal*

[7] Z.B. `foo: () = 1;;`

sind, also nur in dem Codestück bekannt, in welchem sie definiert werden. Das erlaubt uns, die Übersetzung übersichtlicher zu machen, im übrigen ist die Buchführung über diese Label trivial und nur lästig. So kann man jeder Regel einen Zähler Z mitgeben und lokale Label dann in der Form lZ: erzeugen, wobei als Ergebnis neben dem erzeugten Code dann noch der um eins erhöhte Stand des Zählers anzugeben wäre.
Wo es sinnvoll erscheint, fügen wir kurze Kommentare in den Code selbst ein. Ein Kommentar wird durch ein Simikolon eingeleitet, etwa: 'loadLP(1) ;goal '

5.3.2 Struktur der Übersetzung

Wir gehen natürlich davon aus, daß das zu übersetzende Programm statisch-semantisch korrekt ist.
Neben dem fixen Code erzeugt die Übersetzung für jede Funktion und jede Strategie ein Codestück.
Der Code für ein Objekt ergibt sich ganz natürlich aus der Zusammensetzung des Codes für Pattern, Terme, etc. Die Inferenzregeln sind entsprechend gegliedert. Jede Gruppe von Regeln beginnt mit einer Erklärung des in der Übersetzung sich ergebenden Stacklayouts, d.h. welches Layout der Code erwartet und welches er hinterläßt, sowie einigen Zwischenstufen.
Die lokalen/globalen Daten werden über LP/GP adressiert. Das Layout ist dabei folgendes:

$GP \rightarrow$	1	varcount	$LP \rightarrow$	1	goal
	2	depth		2	goals
	3	maxdepth		3	LD_1
	4	trs		.	.
	5	attributes		.	.
	6	GD_1		.	.
	.	.		2+$nldv$	LD_{nldv}
	.	.			
	.	.			
	5+$ngdv$	GD_{ngdv}			

Dabei ist $ngdv$ die Anzahl der globalen Daten, die in der Strategie definiert wurden, $nldv$ die der lokalen Daten.
'depth' ist die augenblickliche Tiefe des Suchprozesses, 'maxdepth' die maximal erlaubte (vgl. Regeln für strategy). 'varcount' dient dazu, die 'newvar' Funktion zu implementieren. Sie enthält die höchste bisher verwandte Zahl n in einem v(n) Objekt im Heap. 'trs' enthält die Adresse der Liste aus Namen/Regel-Paaren, wie sie auch von calctrs in Form der Bindung `TV("trs")` geliefert würde. 'attributes' enthält dementsprechend die Adresse der Attributliste. GD_i ist die Adresse des Globalen Datums i. Die Bedeutung von 'goal', 'goals' und LD_i sollten damit ableitbar sein.
Ein Zugriff auf z.B. 'depth' kann jetzt mit loadGP(2) geschehen.

Bei der Konstruktion einer Funktion (genauer eines Terms) oder eines Patternvergleiches wird immer folgendes Stacklayout angenommen:

$$\boxed{\begin{array}{ll} SP \rightarrow & e_1 \\ & \vdots \\ & e_{level} \\ & LV_1 \\ & \vdots \\ & LV_n \\ & PV_1 \\ & \vdots \\ & PV_k \end{array}}$$

Auf dem Stack liegen oben *level* Elemente, danach folgen die Adressen der lokalen Variablen LV_i, danach die Adressen der Patternvariablen PV_j, d.h. der Variablen, die durch ein Pattern eingeführt wurden. Der Parameter *level* wird bei der Übersetzung eines Terms oder eines Patterns jeweils verwaltet.

5.3.3 Regeln

trprog

$$(310)\quad \frac{\begin{array}{c}\left[(\Upsilon, i + nprim) \overset{\mathsf{trfunc}}{\vdash} fud_i \rightarrow cf_i\right]^{1 \leq i \leq nfuds} \\ \left[(\Upsilon, i) \overset{\mathsf{trstrat}}{\vdash} std_i \rightarrow cs_i\right]^{1 \leq i \leq nstds}\end{array}}{\vdash \mathtt{Prog}(sds, cds, funcds, \ldots, dads, fuds, stds) \rightarrow code}$$

$$\begin{array}{lll}
\text{, wobei} & cds = [(cn_i, cd_i)]_{1 \leq i \leq ncds} & \\
& funcds = [(fn_i, fd_i)]_{1 \leq i \leq nfuncds} & \\
& dads = [(dn_i, dd_i)]_{1 \leq i \leq ndads} & \\
& fuds = [(fun_i, fud_i)]_{1 \leq i \leq nfuds} & \\
& stds = [(stn_i, std_i)]_{1 \leq i \leq nstds} & \\
& \Upsilon = \ \{\mathtt{UOp}(cn_i) \mapsto i | 1 \leq i \leq ncds\} \cup & \\
& \qquad \{\mathtt{UOp}(fn_i) \mapsto ncds + i | 1 \leq i \leq nfuncds\} \cup & \\
& \qquad \{\mathtt{DOp}(dn_i) \mapsto i | 1 \leq i \leq ndads\} \cup & \\
& \qquad \{\mathtt{FOp}(npr_i) \mapsto (i, ap_i) | 1 \leq i \leq nprim\} \cup & \\
& \qquad \{\mathtt{FOp}(fun_i) \mapsto (nprim + i, a_i) | 1 \leq i \leq nfuds\} & \\
& [(npr_i, ap_i)]_{1 \leq i \leq nprim} = & [(\texttt{"/"}, 1), (\texttt{"[<-]"}, 1), (\texttt{"+"}, 1), (\texttt{"-"}, 1), \\
& & (\texttt{"*"}, 1), (\texttt{"\textbackslash"}, 1), (\texttt{">"}, 1), (\texttt{"<"}, 1), \\
& & (\texttt{">="}, 1), (\texttt{"<="}, 1), (\texttt{"=="}, 1), (\texttt{"!="}, 1), \\
& & (\texttt{"\~{}"}, 1), (\texttt{"\&"}, 1), (\texttt{"|"}, 1), (\texttt{"++"}, 1), \\
& & (\texttt{"argsnum"}, 1), (\texttt{"opof"}, 1), (\texttt{"attributes"}, 0), \\
& & (\texttt{"newvar"}, 0), (\texttt{"show"}, 1), (\texttt{"dom"}, 1), \\
& & (\texttt{"compose"}, 1), (\texttt{"apply"}, 1)] \\
& code = cf_1 + \!\!+ \cdots\cdots + \!\!+ cf_{nfuds} + \!\!+ cs_1 + \!\!+ \cdots + \!\!+ cs_{nstds} + \!\!+ \mathsf{Fix}\mathcal{QAM} &
\end{array}$$

trfunc

$$(311)\quad \frac{\left[(\Upsilon, \mathsf{function\_}k\mathsf{\_end}) \overset{\mathsf{treq}}{\vdash} eq_i \rightarrow c_i\right]^{1\leq i\leq n}}{(\Upsilon, k) \vdash \mathtt{Function}([eq_i]_{1\leq i\leq n}) \rightarrow cpro +\!\!+ c_1 +\!\!+ \cdots +\!\!+ c_n +\!\!+ cepi},$$

wobei $cpro = \mathsf{function\_}k\mathsf{:}$

$cepi =$ function_k_error: error(1)
function_k_end: push(3)
update
ret

Bei der Abarbeitung des Codes für eine Funktion wird beim Eintritt in die Funktion der Stack in folgender Form erwartet: $ret{\cdot}p{\cdot}st$ Dabei ist ret die Rückkehradresse, p die Adresse des zu reduzierenden Terms. Die Abarbeitung des Codes für eine Funktion endet im Fehlerfall, d.h. wenn keine Gleichung eq_i anwendbar ist, bei function_k_error:. Die Maschine wird dann mit Fehler 1 angehalten[8]. Im Erfolgsfall endet die Ausführung bei function_k_end:, dabei hat der Stack das Aussehen: $p_t{\cdot}ret{\cdot}p{\cdot}st$, p_t ist dabei die Heapadresse des Ergebnisses. Der Code ab function_k_end: überschreibt @p mit dem Ergebnis und kehrt mittels ret zurück.

treq

$$(312)\quad \frac{\left[(\Upsilon, \emptyset, \Sigma, 0) \overset{\mathsf{trterm}}{\vdash} tl_i \rightarrow cl_i\right]^{1\leq i\leq nlv} \quad (\Upsilon, \emptyset, \Sigma, 0) \overset{\mathsf{trterm}}{\vdash} t_c \rightarrow c_c \quad (\Upsilon, \emptyset, \Sigma, 0) \overset{\mathsf{trterm}}{\vdash} t \rightarrow c_t}{(\Upsilon, \Sigma, l_{end}) \vdash (\mathtt{FOp}(na), t, t_c, ldefs) \rightarrow c_{res}}$$

, wobei $ldefs = [(\mathtt{TV}(nl_j), tl_j)]_{1\leq j\leq nlv}$

$\Sigma = \{\mathtt{TV}(nl_j) \mapsto j | 1 \leq j \leq nlv\}$

$c_{res} =$ args(nlv,0)
cl_1
push(2)
update
$\vdots$
cl_{nlv}
push($1 + nlv$)
update
c_c
call evalnf
load
jf lexit
c_t
slide(nlv)
jump l_{end}
lexit: pop(nlv)

[8]'Missing case in the definition of..'

$$
(313)\ \frac{\begin{array}{cc}(\Upsilon,\Sigma,1,\mathsf{lexit},\emptyset)\overset{\mathsf{trpat}}{\vdash}p_1 \rightarrow c_p : d' & \left[(\Upsilon,\emptyset,\Sigma,0)\overset{\mathsf{trterm}}{\vdash}tl_i \rightarrow cl_i\right]^{1\leq i\leq n} \\ (\Upsilon,\emptyset,\Sigma,0)\overset{\mathsf{trterm}}{\vdash}t_c \rightarrow c_c & (\Upsilon,\emptyset,\Sigma,0)\overset{\mathsf{trterm}}{\vdash}t \rightarrow c_t\end{array}}{(\Upsilon,\Sigma,l_{end})\vdash(\mathtt{App(FOp(}na\mathtt{)},p_1),t,t_c,ldefs) \rightarrow c_{res}}
$$

, wobei $ldefs = [(\mathtt{TV(}nl_j\mathtt{)},tl_j)]_{1\leq j\leq nlv}$

$[\mathtt{TV(}np_j\mathtt{)}]_{1\leq j\leq npv} = patvars(p)$

$\Sigma = \{\mathtt{TV(}nl_j\mathtt{)} \mapsto j | 1\leq j\leq nlv\}$
$\qquad \cup\{\mathtt{TV(}np_j\mathtt{)} \mapsto nlv+j | 1\leq j\leq npv\}$

$c_{res} =$ args(nlv,npv)
push($nlv+npv+2$)
destruct
pop(2)
c_p
cl_1
push(2)
update
$\vdots$
cl_{nlv}
push($1+nlv$)
update
c_c
call evalnf
load
jf lexit
c_t
slide($nlv+npv$)
jump l_{end}
lexit: pop($nlv+npv$)

Der Code für eine Gleichung muß prüfen, ob das Pattern der Funktion paßt[9], falls nicht, wird ans Ende des Codes für die Gleichung gesprungen, wo der Code für die nächste Gleichung, oder 'function_k_error' anschließt. Im Fall eines erfolgreichen Matchings muß noch die Bedingung der Gleichung ausgewertet werden und ggf. schließlich das Ergebnis der Gleichung gebaut und zu l_{end} gesprungen werden.

Der Stack sieht dabei folgendermaßen aus: $SP \rightarrow$

$$
\begin{array}{rl}
1 & LV_1 \\
 & \vdots \\
nlv & LV_{nlv} \\
nlv+1 & PV_1 \\
 & \vdots \\
nlv+npv & PV_{npv} \\
nlv+npv+1 & \text{ret} \\
nlv+npv+2 & p_t
\end{array}
$$

[9] Da bei 0-stelligen Funktionen das Pattern immer paßt, braucht man in diesem Fall (Regel (312)) keinen Code für die Überprüfung zu erzeugen.

trstrat

$$
(314)\quad \frac{\begin{array}{cc}\left[(\Upsilon,\Pi,\Sigma,0)\overset{\mathsf{trterm}}{\vdash} tl_i \rightarrow cl_i\right]^{1\leq i\leq nlv} & \left[(\Upsilon,\Pi,\Sigma,0)\overset{\mathsf{trterm}}{\vdash} tgd_i \rightarrow cgd_i\right]^{1\leq i\leq ngd} \\ \left[(\Upsilon,\Pi,\Sigma,0)\overset{\mathsf{trterm}}{\vdash} tld_i \rightarrow cld_i\right]^{1\leq i\leq nld} & (\Upsilon,\Pi,\Sigma,1)\overset{\mathsf{trterm}}{\vdash} sel \rightarrow csel \\ (\Upsilon,\Pi,\Sigma,0,k)\overset{\mathsf{trcntrl}}{\vdash} cntrl \rightarrow ccntrl & (\Upsilon,\Pi,\Sigma,0,k)\overset{\mathsf{trout}}{\vdash} outs \rightarrow cout \\ \multicolumn{2}{c}{\left[(\Upsilon,\Pi,k,i)\overset{\mathsf{trmr}}{\vdash} mrd_i \rightarrow cmr_i\right]^{1\leq i\leq nmr}}\end{array}}{(\Upsilon,k)\vdash \texttt{Strategy}(gd,ld,sel,cntrl,ldefs,outs,mrs) \rightarrow c_{res}}
$$

, wobei $gd = [(\texttt{TV}(ngd_i),tgd_i)]_{1\leq i\leq ngd}$
$ld = [(\texttt{TV}(nld_i),tld_i)]_{1\leq i\leq nld}$
$ldefs = [(\texttt{TV}(nl_i),tl_i)]_{1\leq i\leq nlv}$
$mrs = [(mrn_i,mrd_i)]_{1\leq i\leq nmr}$
$\Pi = \{\texttt{TV}(ngd_i) \mapsto 5+i | 1\leq i\leq ngd\} \cup \{\texttt{TV("trs")} \mapsto 4\} \cup$
$\{\texttt{TV}(nld_i) \mapsto -i-2 | 1\leq i\leq nld\} \cup \{\texttt{TV("goal")} \mapsto -1\} \cup$
$\{\texttt{TV("goals")} \mapsto -2\} \cup \{mrn_i \mapsto i | 1\leq i\leq nmr\}$
$\Sigma = \{\texttt{TV}(nl_i) \mapsto i | 1\leq i\leq nlv\}$
$c_{res} =$

strategy_k: args(0,$ngd+5$)
setGP
args(0,$nld+2$)
setLP
push($nld+2+ngd+5+2$)
storeLP(1) ;goal
push($nld+2+ngd+5+3$)
storeGP(5) ;attributes
push($nld+2+ngd+5+4$)
storeGP(4) ;trs
push($nld+2+ngd+5+5$)
storeGP(3) ;maxdepth
push($nld+2+ngd+5+6$)
inc
storeGP(1) ;varcount
pushb(num(0))
storeGP(2) ;depth
mkconst(nil)
loadLP(1)
mkcons
storeLP(2) ;goals
cgd_1
storeGP(6)
⋮
cgd_{ngd}
storeGP($ngd+5$)
cld_1
storeLP(3)
⋮
cld_{nld}
storeLP($nld+2$)
call strategy_k_CA
pop($nld+2+ngd+5$)
ret
strategy_k_CA: args(nlv,0)
cl_1
push(2)
update
⋮
cl_{nlv}
push($nlv+1$)
update
loadLP(2) ;goal
match(nil)
jf l_1
pushb(num(0))
call output_k
pop($nlv+1$)
ret
l_1: loadGP(2) ;depth
loadGP(3) ;maxdepth
geq
jt l_2
pushb(num(1))
call output_k
pop($nlv+1$)
ret
l_2: loadLP(2) ;goals
$csel$
mkapp
call evalnf
destruct
pop(1)
storeLP(1) ;goal
storeLP(2) ;goals
$ccntrl$
slide(1)
jf l_3
pushb(num(2))
call output_k
pop($nlv+1$)
ret
l_3: pop(nlv)
ret
cmr_1
⋮
cmr_{nmr}
$cout$

$SP \rightarrow$	1	ret
	2	goal
	3	attributes
	4	trs
	5	maxdepth
	6	varnums

Abbildung 5.2: Stacklayout Strategie

Die Abarbeitung des Codes für eine Strategie beginnt bei strategy_k. Das erwartete Stacklayout ist dabei in Abbildung 5.2 gegeben. Dabei ist 'ret' die Rückkehradresse, 'goal' die Heapadresse des Goals, 'attributes' die der Attributliste, 'trs' die Adresse des TRS, 'maxdepth=num(n)' die maximale Tiefe und 'varnum=num(n')' die Anzahl Variablen, die in 'goal' und 'trs' vorkommen. Man beachte, daß das ursprüngliche Goal aus dem $\mathcal{QIR}$-Programm bereits in ein $\mathcal{QAM}$-Objekt umgewandelt und im Heap angelegt ist, gleiches gilt für das TRS und die Attributliste, die ebenfalls schon berechnet[10] und angelegt sein müssen.

trcntrl

Das Stacklayout bei/nach der Abarbeitung dieses Codes ist folgendes:

Vorher		
$SP \rightarrow$	1	e_1
		$\vdots$
	lev	e_{lev}
	$lev+1$	LV_1
		$\vdots$
	$lev+nlv$	LV_{nlv}

Nachher		
$SP \rightarrow$	1	$dead$
	2	$status$
	3	e_1
		$\vdots$
	$lev+2$	e_{lev}
	$lev+3$	LV_1
		$\vdots$
	$lev+2+nlv$	LV_{nlv}

(315) $(\Upsilon, \Pi, \Sigma, lev, k) \vdash$ `Break` $\rightarrow$ pushb(false)
pushb(true)

(316) $(\Upsilon, \Pi, \Sigma, lev, k) \vdash$ `Empty` $\rightarrow$ pushb(true)
pushb(false)

[10] Gemäß den Regeln calctrs

$$(317)\quad \frac{(\Upsilon,\Pi,\Sigma,lev,k)\vdash cn_1 \to c_1 \quad (\Upsilon,\Pi,\Sigma,lev+1,k)\vdash cn_2 \to c_2}{(\Upsilon,\Pi,\Sigma,lev,k)\vdash \texttt{Seq(}cn_1\texttt{,}cn_2\texttt{)} \to c_{res}}$$

, wobei $c_{res} = c_1$
push(2)
jf l_1
slide(1)
c_2
push(3)
and
push(2)
store(4)
slide(1)
l_1:

$$(318)\quad \frac{(\Upsilon,\Pi,\Sigma,lev,k)\vdash cn_1 \to c_1 \quad (\Upsilon,\Pi,\Sigma,lev,k)\vdash cn_2 \to c_2}{(\Upsilon,\Pi,\Sigma,lev,k)\vdash \texttt{Else(}cn_1\texttt{,}cn_2\texttt{)} \to}$$

c_1
push(1)
push(3)
and
jf l_1
pop(2)
c_2
l_1:

$$(319)\quad \frac{(\Upsilon,\Pi,\Sigma,lev)\overset{\text{trterm}}{\vdash} t \to ct \quad (\Upsilon,\Pi,\Sigma,lev,k)\vdash c \to cc}{(\Upsilon,\Pi,\Sigma,lev,k)\vdash \texttt{If(}t\texttt{,}c\texttt{)} \to c_{res}}$$

, wobei $c_{res} =$

ct
call evalnf
load
jt l_1
pushb(true)
pushb(true)
jump l_2
l_1: cc
l_2:

$$(320)\quad \frac{(\Upsilon,\Pi,\Sigma,level,k)\vdash c \to cc \quad \left[(\Upsilon,\Pi,\Sigma,level+2+i-1)\overset{\text{trterm}}{\vdash} t_i \to ct_i\right]^{1\leq i\leq n}}{(\Upsilon,\Pi,\Sigma,level,k)\vdash \texttt{Assign(}c\texttt{,}as\texttt{)} \to c_{res}}$$

, wobei $as = [(\texttt{TV(}n_i\texttt{)},t_i)]_{1\leq i\leq n}$
$v_i = \Pi(\texttt{TV(}n_i\texttt{)})$
$cs_i = \begin{cases} \text{storeLP}(-v_i), \text{ falls } v_i < 0 \\ \text{storeGP}(v_i) \quad , \text{ falls } v_i > 0 \end{cases}$
$c_{res} =$

cc
push(1)
jt l_1
push(2)
jf l_1
ct_1
$\vdots$
ct_n
cs_n
$\vdots$
cs_1
l_1:

$$(321)\quad \frac{(\Upsilon,\Pi,\Sigma,lev)\overset{\mathsf{trterm}}{\vdash} t \to ct}{(\Upsilon,\Pi,\Sigma,lev,k)\vdash \texttt{Call}(n\texttt{,}t) \to \begin{array}[t]{l} ct \\ \mathsf{call\ MR\_}k\mathsf{\_}\Pi(n) \\ \mathsf{push}(1) \\ \mathsf{pushb(true)} \\ \mathsf{store}(3) \end{array}}, \text{ falls } t \neq ()$$

$$(322)\quad (\Upsilon,\Pi,\Sigma,lev,k)\vdash \texttt{Call}(n\texttt{,()}) \to \begin{array}[t]{l} \mathsf{pushb(true)} \\ \mathsf{call\ MR\_}k\mathsf{\_}\Pi(n) \\ \mathsf{push}(1) \\ \mathsf{pushb(true)} \\ \mathsf{store}(3) \end{array}$$

$$(323)\quad \frac{\begin{array}{c}(\Upsilon,\Pi,\Sigma,lev+2)\overset{\mathsf{trterm}}{\vdash} t \to ct \quad (\Upsilon,\Sigma',1,l_1,\emptyset)\overset{\mathsf{trpat}}{\vdash} p \to cp : d' \\ (\Upsilon,\Pi,\Sigma',0,k)\vdash c \to cc\end{array}}{(\Upsilon,\Pi,\Sigma,lev,k)\vdash \texttt{For}(p\texttt{,}t\texttt{,}c) \to c_{res}}$$

, wobei $[\texttt{TV}(np_i)]_{1\leq i\leq npv} = patvars(p)$

$\Sigma' = \Sigma \oplus \{\texttt{TV}(np_i) \mapsto nlv + i | 1 \leq i \leq npv\}$

$nlv = |\Sigma|$

$c_{res} =$

pushb(true)	pop(1)
pushb(true)	push(2)
ct	store($nlv + npv + 1 + 3$)
args(nlv,npv)	slide(1)
push($nlv + npv + 3 + lev + 1$)	cp
store(2)	cc
$\vdots$	push(2)
push($nlv + npv + 3 + lev + nlv$)	jt l_4
store($nlv + 1$)	store($nlv + npv + 2 + 2$)
l_1:push($nlv + npv + 1$)	jump l_2
call evalhnf	l_4: push($2 + nlv + npv + 2$)
push(1)	and
match(nil)	store($2 + nlv + npv + 2$)
jt l_2	pop(1)
destruct	jump l_1
	l_2: pop($nlv + npv + 2$)

trout

$$(324)\quad \frac{\begin{array}{c}\left[(\Upsilon,\Pi,\Sigma,lev)\overset{\mathsf{trterm}}{\vdash} ts \rightarrow cs'\right]^{\text{falls } \exists(\texttt{"solved"},ts)\in outs} \\ \left[(\Upsilon,\Pi,\Sigma,lev)\overset{\mathsf{trterm}}{\vdash} tb \rightarrow cb'\right]^{\text{falls } \exists(\texttt{"bound"},tb)\in outs} \\ \left[(\Upsilon,\Pi,\Sigma,lev)\overset{\mathsf{trterm}}{\vdash} td \rightarrow cd'\right]^{\text{falls } \exists(\texttt{"dead"},td)\in outs}\end{array}}{(\Upsilon,\Pi,\Sigma,lev,k)\vdash outs \rightarrow \begin{array}[t]{l}\mathsf{output\_}k\text{: } cs \\ \mathsf{ret} \\ l_1\text{: } cb \\ \mathsf{ret} \\ l_2\text{: } cd \\ l_3\text{: } \mathsf{ret}\end{array}}$$

$$\text{wobei } \begin{array}[t]{l} cs = if(\exists(\texttt{"solved"},t_s)\in outs, cs'', \texttt{""}) \\ cb = if(\exists(\texttt{"bound"},t_b)\in outs, cb'', \texttt{""}) \\ cd = if(\exists(\texttt{"dead"},t_d)\in outs, cd'', \texttt{""}) \end{array}$$

$$cs'' = \begin{array}[t]{l}\mathsf{push}(2) \\ \mathsf{pushb}(\mathsf{num}(0)) \\ \mathsf{eq} \\ \mathsf{jf}\ l_1 \\ cs \\ \mathsf{call\ evalnf} \\ \mathsf{out}(0,\texttt{"Solved\textbackslash","\textbackslash 0"})\end{array}$$

$$cb'' = \begin{array}[t]{l}\mathsf{push}(2) \\ \mathsf{pushb}(\mathsf{num}(1)) \\ \mathsf{eq} \\ \mathsf{jf}\ l_2 \\ cb \\ \mathsf{call\ evalnf} \\ \mathsf{out}(1,\texttt{"Bound:","\textbackslash 0"})\end{array}$$

$$cd'' = \begin{array}[t]{l}\mathsf{push}(2) \\ \mathsf{pushb}(\mathsf{num}(2)) \\ \mathsf{eq} \\ \mathsf{jf}\ l_3 \\ cd \\ \mathsf{call\ evalnf} \\ \mathsf{out}(1,\texttt{"Dead:","\textbackslash 0"})\end{array}$$

trmr

$$
(325)\quad \frac{\begin{array}{cc}(\Upsilon,\Sigma,1,l_1,\emptyset)\overset{\textsf{trpat}}{\vdash} gp \rightarrow cgp : d' & \left[(\Upsilon,\Pi,\Sigma,0)\overset{\textsf{trterm}}{\vdash} t_i \rightarrow ct_i\right]^{1\leq i\leq nlv} \\ (\Upsilon,\Pi,\Sigma,0)\overset{\textsf{trterm}}{\vdash} tc \rightarrow ctc & (\Upsilon,\Pi,\Sigma,0)\overset{\textsf{trterm}}{\vdash} to \rightarrow cto \\ (\Upsilon,\Pi,\Sigma,nld+2)\overset{\textsf{trterm}}{\vdash} tr \rightarrow ctr & \left[(\Upsilon,\Pi,\Sigma,nld+i)\overset{\textsf{trterm}}{\vdash} at_i \rightarrow cat_i\right]^{1\leq i\leq nas}\end{array}}{(\Upsilon,\Pi,k,j)\vdash \texttt{MetaRule((),}tc\texttt{,}gp\texttt{,}tr\texttt{,}as\texttt{,}to\texttt{,}ldefs\texttt{)} \rightarrow c_{res}}
$$

, wobei $as = [(\texttt{TV(}an_i\texttt{)}, at_i)]_{1\leq i\leq nas}$

$ldefs = [(\texttt{TV(}n_i\texttt{)}, t_i)]_{1\leq i\leq nlv}$

$nld = |\{\texttt{TV(}na\texttt{)} \mapsto n \in \Pi | n < 0\}|$

$[\texttt{TV(}np_i\texttt{)}]_{1\leq i\leq npv} = patvars(gp)$

$\Sigma = \{\texttt{TV(}n_i\texttt{)} \mapsto i | 1\leq i\leq nlv\} \cup \{\texttt{TV(}np_i\texttt{)} \mapsto nlv+i | 1\leq i\leq npv\}$

$v_i = \Pi(\texttt{TV(}an_i\texttt{)})$

$cv_i = \begin{cases} \textsf{storeLP}(-v_i), \text{ falls } v_i < 0 \\ \textsf{storeGP}(v_i) \ \ , \text{ falls } v_i > 0 \end{cases}$

$c_{res} =$

MR_k_j:
args(nlv,npv)
loadLP(1) ;goal
cgp
ct_1
push(2)
update
⋮
ct_{nlv}
push($nlv+1$)
update
ctc
call evalnf
load
jf l_1
cto
call evalnf
out(1,"MR:","\0")
pushLP
args(0,nld)

loadLP(1)
store(2)
⋮
loadLP(nld)
store($nld+1$)
setLP
loadLP(2) ;goals
ctr
pushb(num(2))
mktupel
mkfop(16,1,function_16)
mkapp
call evalnf
storeLP(2) ;goals
cat_1
call evalnf
⋮
cat_{nas}
call evalnf

cv_{nas}
⋮
cv_1
loadGP(2) ;depth
inc
storeGP(2) ;depth
call strategy_k_CA
loadGP(2) ;depth
dec
storeGP(2)
pop(nld)
popLP
pop($nlv+npv$)
pushb(false)
store(3)
ret
l_1: pop($nlv+npv$)
pushb(true)
store(3)
ret

$$
(326)\;\frac{\begin{array}{c}(\Upsilon,\Sigma,1,l_1,\emptyset)\overset{\mathsf{trpat}}{\vdash} rp \rightarrow crp : d' \\ (\Upsilon,\Sigma,1,l_1,\emptyset)\overset{\mathsf{trpat}}{\vdash} gp \rightarrow cgp : d'' \qquad \left[(\Upsilon,\Pi,\Sigma,0)\overset{\mathsf{trterm}}{\vdash} t_i \rightarrow ct_i\right]^{1\leq i\leq nlv} \\ (\Upsilon,\Pi,\Sigma,0)\overset{\mathsf{trterm}}{\vdash} tc \rightarrow ctc \qquad (\Upsilon,\Pi,\Sigma,0)\overset{\mathsf{trterm}}{\vdash} to \rightarrow cto \\ (\Upsilon,\Pi,\Sigma,nld+2)\overset{\mathsf{trterm}}{\vdash} tr \rightarrow ctr \quad \left[(\Upsilon,\Pi,\Sigma,nld+i)\overset{\mathsf{trterm}}{\vdash} at_i \rightarrow cat_i\right]^{1\leq i\leq nas}\end{array}}{(\Upsilon,\Pi,k,j)\vdash \mathtt{MetaRule}(rp,tc,gp,tr,as,to,ldefs) \rightarrow c_{res}}
$$

, falls $rp \neq ()$

, wobei $as = [(\mathtt{TV}(an_i), at_i)]_{1\leq i\leq nas}$

$ldefs = [(\mathtt{TV}(n_i), t_i)]_{1\leq i\leq nlv}$

$nld = |\{\mathtt{TV}(na) \mapsto n \in \Pi | n < 0\}|$

$[\mathtt{TV}(np_i)]_{1\leq i\leq npv} = patvars(rp) +\!\!+ patvars(gp)$

$\Sigma = \{\mathtt{TV}(n_i) \mapsto i | 1 \leq i \leq nlv\} \cup \{\mathtt{TV}(np_i) \mapsto nlv + i | 1 \leq i \leq npv\}$

$c_{res} = \mathsf{MR\_}k\mathsf{\_}j$:

$\mathsf{args}(nlv,npv)$
$\mathsf{push}(nlv + npv + 2)$
crp
c_{rest}

c_{rest} = Code der vorigen Regel, ab Zeile $\mathsf{loadLP}(1)$.

trterm

(327) $(\Upsilon,\Pi,\Sigma,lev)\vdash t \rightarrow \mathsf{mkconst}(const2const(t))$

, falls $t \in \{\mathtt{Nil}, \mathtt{Failed}, \mathtt{True}, \mathtt{False}, \mathtt{N}(i), \mathtt{Char}(c)\}$

(328) $(\Upsilon,\Pi,\Sigma,lev)\vdash t \rightarrow ct$

, falls $t \in \{\mathtt{DOp}(na), \mathtt{UOp}(na), \mathtt{FOp}(na)\}$

, wobei $ct = \begin{cases} \mathsf{mkdop}(\Upsilon(t)) & \text{, falls } t = \mathtt{DOp}(na) \\ \mathsf{mkuop}(\Upsilon(t)) & \text{, falls } t = \mathtt{UOp}(na) \\ \mathsf{mkfop}(n,a,\mathsf{function\_}n) & \text{, falls } t = \mathtt{FOp}(na) \text{ und } \Upsilon(t) = (n,a) \end{cases}$

(329) $(\Upsilon,\Pi,\Sigma,lev)\vdash \mathtt{V}(na) \rightarrow \begin{array}{l}\mathsf{pushb}(\mathsf{num}(\Upsilon(na))) \\ \mathsf{mkv}\end{array}$

$$
(330)\;\frac{(\Upsilon,\Pi,\Sigma,lev)\vdash t_2 \rightarrow ct_2 \quad (\Upsilon,\Pi,\Sigma,lev+1)\vdash t_1 \rightarrow ct_1}{(\Upsilon,\Pi,\Sigma,lev)\vdash \mathtt{C}(t_1,t_2) \rightarrow \begin{array}{l} ct_2 \\ ct_1 \\ \mathsf{mkcons}\end{array}}
$$

$$
(331)\;\frac{(\Upsilon,\Pi,\Sigma,lev)\vdash t_2 \rightarrow ct_2 \quad (\Upsilon,\Pi,\Sigma,lev+1)\vdash t_1 \rightarrow ct_1}{(\Upsilon,\Pi,\Sigma,lev)\vdash \mathtt{App}(t_1,t_2) \rightarrow \begin{array}{l} ct_2 \\ ct_1 \\ \mathsf{mkapp}\end{array}}
$$

$$(332)\quad \frac{[(\Upsilon, \Pi, \Sigma, lev + (n-i)) \vdash t_i \to ct_i]^{1 \le i \le n}}{(\Upsilon, \Pi, \Sigma, lev) \vdash \mathtt{T}([t_1, \ldots, t_n]) \to \begin{array}{l} ct_n \\ \vdots \\ ct_1 \\ \mathsf{pushb(num(}n\mathsf{))} \\ \mathsf{mktupel} \end{array}}$$

$$(333)\quad \frac{\begin{array}{c}[(\Upsilon, \Pi, \Sigma, lev + 2(n-i)) \vdash t'_i \to ct'_i]^{1 \le i \le n} \\ [(\Upsilon, \Pi, \Sigma, lev + 2(n-i) + 1) \vdash t_i \to ct_i]^{1 \le i \le n}\end{array}}{(\Upsilon, \Pi, \Sigma, lev) \vdash \mathtt{Subst}([(t_1, t'_1), \ldots, (t_n, t'_n)]) \to \begin{array}{l} ct'_n \\ ct_n \\ \vdots \\ ct'_1 \\ ct_1 \\ \mathsf{pushb(num(}2n\mathsf{))} \\ \mathsf{mksubst} \end{array}}$$

$$(334)\quad (\Upsilon, \Pi, \Sigma, lev) \vdash \mathtt{TV}(na) \to ct$$

$$, \text{wobei } ct = \begin{cases} \mathsf{push(}lev + \Sigma(na)\mathsf{)}, & \text{falls } na \in \mathcal{D}(\Sigma) \\ \mathsf{loadGP(}\Pi(na)\mathsf{)} & , \text{falls } na \notin \mathcal{D}(\Sigma) \text{ und } \Pi(na) > 0 \\ \mathsf{loadLP(}-\Pi(na)\mathsf{)} & , \text{falls } na \notin \mathcal{D}(\Sigma) \text{ und } \Pi(na) < 0 \end{cases}$$

trpat

$$(335)\quad (\Upsilon, \Sigma, lev, lab, done) \vdash t \to ct : done$$

$$, \text{falls } t \in \{\mathtt{Nil}, \mathtt{True}, \mathtt{False}, \mathtt{N}(i), \mathtt{Char}(c), \mathtt{DOp}(na)\}$$

$$, \text{wobei } tc = \begin{cases} const2const(t) & , \text{falls } t \neq \mathtt{DOp}(na) \\ \mathsf{dop(}\Upsilon(\mathtt{DOp}(na))\mathsf{)}, & \text{falls } t = \mathtt{DOp}(na) \end{cases}$$

$$ct = \begin{array}{l} \mathsf{call\ evalhnf} \\ \mathsf{match(}tc\mathsf{)} \\ \mathsf{jt}\ l_1 \\ \mathsf{pop(}lev - 1\mathsf{)} \\ \mathsf{jump}\ lab \\ l_1: \end{array}$$

$$(336)\quad \frac{\begin{array}{c}(\Upsilon, \Sigma, lev + 1, lab, done) \vdash p_1 \to cp_1 : d' \\ (\Upsilon, \Sigma, lev, lab, d') \vdash p_2 \to cp_2 : d''\end{array}}{(\Upsilon, \Sigma, lev, lab, done) \vdash \mathtt{C}(p_1, p_2) \to c_{res} : d''}$$

$$, \text{wobei } c_{res} = \begin{array}{l|l} \mathsf{call\ evalhnf} & \mathsf{jump}\ lab \\ \mathsf{push(1)} & l_1\text{: } \mathsf{destruct} \\ \mathsf{match(c)} & \mathsf{pop(1)} \\ \mathsf{jt}\ l_1 & cp_1 \\ \mathsf{pop(}lev\mathsf{)} & cp_2 \end{array}$$

$$(337)\quad \frac{\begin{array}{c}(\Upsilon,\Sigma,lev+1,lab,done)\vdash p_1 \to cp_1 : d' \\ (\Upsilon,\Sigma,lev,lab,d')\vdash p_2 \to cp_2 : d''\end{array}}{(\Upsilon,\Sigma,lev,lab,done)\vdash \texttt{App}(p_1,p_2) \to c_{res} : d''}$$

, wobei $c_{res} = \begin{array}{l|l} \textsf{call evalhnf} & \textsf{jump } lab \\ \textsf{push}(1) & l_1\textsf{: destruct} \\ \textsf{match(app)} & \textsf{pop}(1) \\ \textsf{jt } l_1 & cp_1 \\ \textsf{pop}(lev) & cp_2 \end{array}$

$$(338)\quad \frac{[(\Upsilon,\Sigma,lev-1+(n-i+1),lab,d_{i-1})\vdash p_i \to cp_i : d_i]^{1\leq i\leq n}}{(\Upsilon,\Sigma,lev,lab,d_0)\vdash \texttt{T}([p_1,\ldots,p_n]) \to c_{res} : d_n}$$

, wobei $c_{res} = \begin{array}{l} \textsf{call evalhnf} \\ \textsf{destruct} \\ \textsf{pop}(1) \\ cp_1 \\ \vdots \\ cp_n \end{array}$

(339) $(\Upsilon,\Sigma,lev,lab,done)\vdash \texttt{TV}(na) \to \textsf{store}(lev+\Sigma(na)) : d'$

, falls $na \notin done$
, wobei $d' = done \cup \{na\}$

(340) $(\Upsilon,\Sigma,lev,lab,done)\vdash \texttt{TV}(na) \to c_{res} : done$
, falls $na \in done$
, wobei $c_{res} =$

$$\begin{array}{l|l} \textsf{call evalnf} & \textsf{call compare} \\ \textsf{push}(lev+\Sigma(na)) & \textsf{jt } l_1 \\ \textsf{call evalnf} & \textsf{pop}(lev-1) \\ \textsf{push}(1) & \textsf{jump } lab \\ \textsf{store}(lev+\Sigma(na)+2) & l_1\textsf{:} \end{array}$$

(341) $(\Upsilon,\Sigma,lev,lab,done)\vdash \texttt{Op}(\texttt{TV}(na)) \to c_{res} : done \cup \{na\}$
, falls $na \notin done$
, wobei $c_{res} = \begin{array}{l|l} \textsf{call evalhnf} & \textsf{jf } l_1 \\ \textsf{push}(1) & \textsf{pop}(lev-1) \\ \textsf{store}(lev+\Sigma(na)+1) & \textsf{jump } lab \\ \textsf{match(v)} & l_1\textsf{:} \end{array}$

(342) $(\Upsilon, \Sigma, lev, lab, done) \vdash$ `Op(TV(`na`))` $\rightarrow c_{res} : done$

, falls $na \in done$

, wobei $c_{res} =$

call evalnf	jump lab	call compare
push(1)	l_1: push($lev + \Sigma(na)$)	jt l_2
match(v)	call evalnf	pop($lev - 1$)
jf l_1	push(1)	jump lab
pop(lev)	store($lev + \Sigma(na) + 2$)	l_2:

(343) $(\Upsilon, \Sigma, lev, lab, done) \vdash$ `Var(TV(`na`))` $\rightarrow c_{res} : done \cup \{na\}$

, falls $na \notin done$

, wobei $c_{res} =$

call evalhnf	jt l_1
push(1)	pop($lev - 1$)
store($lev + \Sigma(na) + 1$)	jump lab
match(v)	l_1:

(344) $(\Upsilon, \Sigma, lev, lab, done) \vdash$ `Var(TV(`na`))` $\rightarrow c_{res} : done$

, falls $na \in done$

, wobei $c_{res} =$

call evalnf	jump lab	call compare
push(1)	l_1: push($lev + \Sigma(na)$)	jt l_2
match(v)	call evalnf	pop($lev - 1$)
jt l_1	push(1)	jump lab
pop(lev)	store($lev + \Sigma(na) + 2$)	l_2:

5.3.4 Fixer Codeanteil: Fix$\mathcal{QAM}$

Code für "/" $\Downarrow$

function_1: push(2)
destruct
pop(2)
call evalnf
destruct
pop(1)
call getsubterm
slide(1)
push(3)
update
ret
getsubterm: push(3)
match(nil)
jf l_1
ret
l_1: push(3)
destruct
pop(1)
load
push(4)
call getarg
slide(1)
call getsubterm
slide(1)
store(3)
ret
getarg: push(3)
pushb(num(0))
lt
jt l_2
error(3)
l_2: push(2)
match(v)
jf l_3
error(3)

```
l3: push(2)
match(app)
jt l4
error(3)
l4: push(2)
destruct
pop(2)
push(1)
match(t)
jt l5
push(4)
pushb(num(1))
eq
jt l6
error(3)
l6:
store(3)
ret
l5: destruct
push(1)
pushb(num(5))
add
pushi
push(1)
push(3)
geq
jt l7
error(3)
l7: pushb(num(2))
add
pushi
push(2)
pushb(num(5))
add
storei
popi
ret
```

Code für "[<-]" ⇓

```
function_2: push(2)
destruct
pop(2)
call evalnf
destruct
pop(1)
clone
call replace
slide(2)
push(3)
update
ret
replace: push(3)
match(nil)
jf l1
push(4)
store(3)
ret
l1: push(3)
destruct
pop(1)
load
push(6)
push(2)
push(6)
call getarg
slide(1)
push(4)
swap
call replace
slide(2)
push(5)
call setarg
slide(3)
store(3)
ret
setarg: push(2)
destruct
pop(1)
swap
push(1)
match(t)
jt l2
push(5)
store(2)
swap
mkapp
```

```
store(3)
ret
l_2: destruct
push(1)
pushb(num(6))
add
pushi
push(2)
pushb(num(8))
add
pushi
pushb(num(3))
add
storei
mktupel
swap
mkapp
store(3)
ret
```

Code für "+","-","*","\" ⇓

```
function_3: push(2)
destruct
pop(2)
call evalnf
destruct
pop(1)
load
swap
load
swap
add
mknum
push(3)
update
ret
```

Code für "-","*","\" ähnlich, nur statt 'add' wird dann 'sub', 'mul' oder 'div' eingesetzt. Dies ergibt Code für function_3 bis function_6

Code für ">", "<", ">=", "<=" ⇓

```
function_7: push(2)
destruct
pop(2)
call evalnf
destruct
pop(1)
load
swap
load
swap
gt
jt l_1
mkconst(false)
jump l_2
l_1: mkconst(true)
l_2: push(3)
update
ret
```

Code für "<", ">=", "<=" ähnlich, durch Ersetzung von 'gt' durch 'lt', 'geq', 'leq'. Ergibt function_7 bis function_10

Code für "==" ⇓

```
function_11: push(2)
destruct
pop(2)
call evalhnf
destruct
pop(1)
call compare
jt l_1
mkconst(false)
jump l_2
l_1: mkconst(true)
l_2: push(3)
update
ret
```

Code für "!=" ⇓

```
function_12: push(2)
call function_11
load
jf l_1
mkconst(false)
jump l_2
l_1: mkconst(true)
l_2: push(3)
```

```
update
ret
```

Code für "~" ⇓

```
function_13: push(2)
destruct
pop(2)
call evalhnf
load
jt l1
mkconst(true)
jump l2
l1: mkconst(false)
l2: push(3)
update
ret
```

Code für "&" ⇓

```
function_14: push(2)
destruct
pop(2)
call evalhnf
destruct
pop(1)
call evalnf
load
jt l1
pop(1)
mkconst(false)
l1: push(3)
update
ret
```

Code für "|" ⇓

```
function_15: push(2)
destruct
pop(2)
call evalhnf
destruct
pop(1)
call evalnf
load
jf l1
pop(1)
mkconst(true)
l1: push(3)
update
ret
```

Code für "++" ⇓

```
function_16: push(2)
destruct
pop(2)
call evalhnf
destruct
pop(1)
call evalhnf
push(1)
match(nil)
jf l1
pop(1)
push(3)
update
ret
l1: destruct
pop(1)
push(3)
push(3)
pushb(num(2))
mktupel
mkfop(16,1,function_16)
mkapp
swap
mkcons
slide(2)
push(3)
update
ret
```

Code für "argsnum" ⇓

```
function_17: push(2)
destruct
pop(2)
call evalnf
push(1)
match(v)
jt l1
push(1)
match(app)
```

```
jt l2
l1: pop(1)
mkconst(num(0))
push(3)
update
ret
l2: destruct
pop(2)
push(1)
match(t)
jt l3
pop(1)
mkconst(num(1))
push(3)
update
ret
l3: destruct
push(1)
mknum
push(2)
pushb(num(5))
add
pushi
update
popi
ret
```

Code für "opof" ⇓

```
function_18: push(2)
destruct
pop(2)
call evalnf
push(1)
match(v)
jf l1
error(4)
l1: push(1)
match(app)
jt l2
push(3)
update
ret
l2: destruct
pop(1)
slide(1)
push(3)
update
ret
```

Code für "attributes" ⇓

```
function_19: loadGP(5)
push(3)
update
ret
```

Code für "newvar" ⇓

```
function_20: loadGP(1)
inc
push(1)
storeGP(1)
mkv
push(3)
update
ret
```

Code für "show" ⇓

```
function_21: push(2)
destruct
pop(2)
call evalnf
show
push(3)
update
ret
```

Code für "dom" ⇓

```
function_22: push(2)
destruct
pop(2)
call evalnf
push(1)
match(failed)
jf l1
error(5)
l1: dom
mkconst(nil)
l3: push(2)
pushb(num(0))
```

```
eq
jf l2
slide(1)
push(3)
update
ret
l2: push(3)
push(3)
dec
store(5)
push(2)
store(4)
slide(1)
mkcons
jump l3
```

Code für "compose" ⇓

```
function_23: push(2)
destruct
pop(2)
call evalnf
destruct
pop(1)
clone
compose
push(3)
update
ret
```

Code für "apply" ⇓

```
function_24: push(2)
destruct
pop(2)
call evalnf
destruct
pop(1)
push(2)
clone
store(3)
repl
push(3)
update
ret
```

Code für compare: Lazy Vergleich ⇓

```
compare: push(3)
call evalhnf
push(3)
call evalhnf
push(2)
push(2)
cmp
jt l1
pop(2)
pushb(false)
store(4)
slide(1)
ret
l1: swap
destruct
push(1)
pushb(num(3))
add
pushi
destruct
push(1)
push(1)
pushb(num(4))
add
pushi
eq
jt l2
popi
popi
pop(1)
pushb(false)
store(4)
slide(1)
ret
l2: push(1)
pushb(num(0))
eq
jf l3
popi
popi
pop(1)
pushb(true)
```

```
store(4)
slide(1)
ret
l3: push(2)
push(2)
dec
store(4)
slide(1)
push(2)
pushb(num(4))
add
pushi
pushb(num(4))
add
pushi
swap
call compare
jt l2
popi
popi
pop(1)
pushb(false)
store(4)
slide(1)
ret
```

Code für evalhnf: Auswertung zu HNF ⇓

```
evalhnf: push(2)
ishnf
jf l1
ret
l1: push(2)
unwind
pop(1)
jump evalhnf
```

Code für evalnf: Auswertung zu Normalform ⇓

```
evalnf: push(2)
call evalhnf
push(1)
match(fop)
jf l1
error(2)
l1: push(1)
destruct
l2: push(1)
pushb(num(0))
eq
jf l3
pop(2)
ret
l3: dec
swap
call evalnf
pop(1)
jump l2
```

5.3.5 Kommentar

trprog

Die Übersetzung eines $\mathcal{QIR}$-Programms beginnt mit der Konstruktion der Namensabbildung Υ. Das Ergebnis ist die Zusammenfügung von Codestücken für jede Funktion und jede Strategie, sowie eines fixen Codestücks Fix$\mathcal{QAM}$.

trfunc

Als Ergebnis werden die einzelnen Codestücke für jede Gleichung, eingerahmt von dem Prolog und dem Epilog, wobei letzterer einmal den Ausstiegscode (function_k_end:) enthält, der den zu reduzierenden Graphen mit dem Ergebnis überschreibt, und daneben den Fehlercode (function_k_error:) beinhaltet, der erreicht wird, wenn keine Gleichung anwendbar war.

treq

Regel (312) behandelt den Fall für eine Funktion ohne Argumente, bei der kein Pattern gematcht werden muß. Σ enthält die Indices der lokalen Variablen der Gleichung.
Der Code beginnt mit dem Anlegen von Platz für die lokalen Variablen (args(nlv,0)), danach werden die rechten Seiten der lokalen Definitionen gebaut: cl_i. Die Referenzen der linken Seiten werden auf die Heapadressen der rechten Seiten umgelenkt. Da die Definitionen rekursiv sind, müssen wir sie eigentlich 'parallel' bauen, d.h. eine Definition kann sich auf eine andere beziehen, die nocht nicht fertiggestellt ist. Wir umgehen diese Problematik dadurch, daß mit dem args(nlv,0)-Befehl schon nlv Heapzellen ind(?) alloziiert und deren Adressen in den Stackpositionen LV_i abgelegt werden. Bei der Konstruktion der Definitionen können wir jetzt diese Heapadressen benutzen. Das Codestück 'push($1+i$) update' sorgt dafür, daß die Zelle, deren Adresse in LV_i liegt, durch die Indirektion ind(p_i) ersetzt wird. Unter derselben Adresse haben wir also nun Zugriff auf das Ergebnis.
Der Code fährt mit der Konstruktion der Bedingung der Gleichung fort (c_c), der sich ergebende Term wird ausgewertet (call evalnf) und der Wert, dessen Adresse auf dem Stack liegt, wird auf 'true' überprüft. Ist dies nicht der Fall, dann wird der Code der Gleichung verlassen (lexit), ansonsten wird das Ergebnis konstruiert (c_t), die zwischenliegenden LV_i entfernt und zum Ende der Funktion bei l_{end} gesprungen.
Falls die Gleichung ein Argument nimmt, muß vorher nur noch Code für das Matchen des Patterns auf den Eingabegraphen gebaut werden. es geschieht in der nächsten Regel. Der zu reduzierende Graph hat die Form 'app(fop(n,1,cl),pt_1)', das Pattern der Gleichung die Form '`App(FOp(na),`P_1`)`', also brauchen wir nur Code für das Matchen von P_1 auf pt_1. Der Code für diesen Fall enthält daher zusätzlich ein kurzes Stück von 'push($nlv+npv+2$)' bis 'c_p', im Vergleich mit der vorigen Regel, wobei man natürlich jetzt noch Platz für die Patternvariablen reservieren muß (args(nlv,npv)). Der push-Befehl holt den zu reduzierenden Graphen nach oben und zerlegt ihn (destruct). Die nun oben liegenden Elemente num(2) und fop(n,1,cl) werden durch pop(2) entfernt, übrig bleibt pt_1. Der Code c_p matcht nun gerade das Pattern auf diesen Graphen. Schlägt der Match fehl, wird zu lexit gesprungen, dort wird dann der Stack aufgeräumt und der Code der nächsten Gleichung (oder function_k_error) ausgeführt, der hier anschließt. Im Erfolgsfall ist der weitere Ablauf identisch mit dem in der vorangegangenen Regel.

trstrat

Diese Übersetzung ist die komplexeste. Zuerst werden die Zuordnungen Π, Σ gebaut. Der Code beginnt mit der Anlage von Platz für die globalen Daten (args(0,$ngd+5$)), die in der Art von Patternvariablen angelegt werden, also keine neuen Zellen alloziieren. Der GP wird durch (setGP) auf das erste Datum gesetzt. In ähnlicher Weise wird Platz für die lokalen Daten reserviert und der LP auf das erste Datum gesetzt. Darauf folgt die Initialisierung der globalen/lokalen Daten aus den übergebenen Werten im Stack (siehe

Abbildung 5.2). Das Datum 'depth' wird zu 0 initialisiert, aus dem übergebenem Goal wird eine Liste 'goal:[]' konstruiert und in 'goals' abgelegt. Nun werden die globalen und lokalen Daten, die von der Strategie definiert werden, initialisiert (cgd_i, cld_i). danach wird die Ablaufsteuerung über call strategy_k_CA aufgerufen. Nach der Rückkehr dieses Aufrufs wird der Stack wieder bereinigt und mit ret zum Aufrufpunkt zurückgekehrt.
Ab strategy_k_CA: steht Code, der die Prüfung auf die Ereignisse "solved", "bound" vornimmt, die Ablaufsteuerung ausführt und danach auf das Ereignis "dead" prüft. Dieser Punkt wird immer angesprungen, wenn der Suchprozess eine Ebene tiefer geht. Es beginnt mit der Konstruktion der lokalen Definitionen. Danach wird mit loadLP(2) die aktuelle Goalliste auf den Stack gelegt und darauf geprüft, ob sie leer ist (match(nil)). Falls dies der Fall ist, wird die Ausgabe dieser Strategie mit dem Wert num(0) auf dem Stack als Argument aufgerufen, was der Anforderung entspricht, die Ausgabe für "solved" vorzunehmen. Danach wird der Stack aufgeräumt und zurückgesprungen. War die Goalliste nicht leer, wird ab l_1 geprüft, ob die Tiefenbegrenzung überschritten ist, wenn dies der Fall ist, wird die Ausgabe mit Argument num(1), was "bound" entspricht, aufgerufen und zurückgekehrt. Andernfalls wird ab l_2 der Selektionsterm `App(`sel`,TV("goals"))` konstruiert und über call evalnf ausgewertet. Auf dem Stack liegt nun die Adresse einer Zelle t([$goal, goals$]), die mit destruct und pop(1) zerlegt wird. Auf dem Stack liegen jetzt das neuen aktuelle Goal und die neue Goalliste, die mit storeLP(1/2) gespeichert werden. Danach folgt der Code für die richtige Ablaufsteuerung $ccntrl$. Nach Durchlauf durch diesen Code, hat der Stack folgendes Aussehen: $dead{\bullet}status{\bullet}old$. Oben liegt ein Bool'scher Wert, der true ist, falls die Ablaufsteuerung 'tot' war, d.h. eine Sackgasse. $status$ ist false, falls ein `Break`-Kommando aktiv wahr.
Wahr die Ablaufsteuerung 'tot', dann wird dir Ausgabe mit Argument num(2), entspricht 'dead', aufgerufen, andernfalls wird einfach zurückgekehrt.

trcntrl

Regel (315) behandelt den Fall eines `Break`-Terms: Als Ergebnis enthält der Stack true•false•st, oder kürzer: $dead =$ true, $status = broken$.
Die nächste Regel für `Empty` setzt $break =$ false, $status = OK$.
Regel (317) betrifft das `Seq(`cn_1`,`cn_2`)`-Konstrukt. Zuerst wird der Code des ersten Arguments abgearbeitet (c_1). Falls $status_1 = OK$, dann wird der Code des zweiten Arguments abgearbeitet, als Ergebnis erhält man dann $dead_3 = dead_1 \wedge dead_2$, $status_3 = status_2$. Andernfalls ist das Ergebnis das des ersten Arguments.
Regel (318) definiert die Übersetzung eines `Else(`cn_1`,`cn_2`)` ähnlich, nur gilt hier, daß der Code des zweiten Arguments nur abgearbeitet wird, wenn das Ergebnis des ersten Arguments $dead_1 =$ true, $status_1 = OK$ war.
Das `If(t,c)`-Konstrukt wird in der folgenden Regel behandelt. Zuerst wird die Bedingung konstruiert (ct). Falls ihre Auswertung false ergibt, wird das Ergebnis $dead =$ true, $status = OK$ abgelegt, andernfalls der Code cc abgearbeitet.
Die Regel (320) beschreibt die Übersetzung des `Assign(c,as)`-Konstruktes. Zuerst wird der Code cc abgearbeitet, falls $dead =$ false und $status = OK$ sind, werden die Ergeb-

$SP \rightarrow$	1	LV_1
		$\vdots$
	nlv	LV_{nlv}
	$nlv+1$	PV_1
		$\vdots$
	$nlv+npv$	PV_{npv}
	$nlv+npv+1$	t
	$nlv+npv+2$	$dead$
	$nlv+npv+3$	$status$
	$nlv+npv+4$	e_1
		$\vdots$
	$nlv+npv+3+lev$	e_{lev}
	$nlv+npv+3+lev+1$	LV_1^{ini}
		$\vdots$
	$nlv+npv+3+lev+nlv$	LV_{nlv}^{ini}

Abbildung 5.3: Stacklayout `For`-Konstrukt

nisse der Zuweisungen gebaut und nachdem alle gebaut wurden, werden die Werte der entsprechende lokalen/globalen Daten ersetzt. Da die Werte erst über GP/LP gespeichert werden, *nachdem* alle Zuweisungen gebaut wurden, benutzt man bei der Konstruktion die alten Werte der Daten, nicht bereits ersetzte.

Die folgenden beiden Regeln (321/322) beschäftigen sich mit dem Aufruf einer Metaregel. Die erste Regel trifft auf den Aufruf einer Metaregel, die Argumente hat. Zuerst wird das zu übergebende Argument konstruiert (ct), danach der Code der Metaregel aufgerufen. Der Argumentterm ist danach durch das Ergebnis ersetzt worden, dem Wert *dead*. Zusätzlich wird noch $status = OK$ abgelegt.

Falls die Metaregel kein Argument hat, wird an seiner Stelle der Wert **true** abgelegt.

Die letzte Regel, (323), beschreibt die Übersetzung des `For(p,t,c)`-Konstrukts und ist sowohl am interessantesten, wie auch am kompliziertesten. Zuerst ist anzumerken, daß diese Regel Σ modifiziert. Patternvariablen aus p ersetzen ggf. lokale Definitionen. Das Stacklayout bei der Abarbeitung ist in Abbildung 5.3 dargestellt.

Der Code kopiert die lokalen Variablen $LV_i = LV_i^{ini}$ und schafft Platz für die Patternvariablen PV_j von p. Als erstes werden Initialwerte für $dead, status$ abgelegt ($dead =$ **true**$, status = OK$). Danach wird der Code für t durchlaufen (ct). Jetzt wird Platz für die Kopien der lokalen Variablen, die neuen Patternvariablen gemacht (**args**(npv,nlv)), worauf LV_j^{ini} nach LV_j kopiert wird.

Der Code läuft jetzt in einer Schleife, bis alle Elemente von t abgearbeitet sind. Die Schleife beginnt bei l_1. Mit **push**($nlv + npv + 1$) wird t hochgeholt und dann zu HNF ausgewertet. Falls t bereits die leere Liste **nil** ist, endet die Schleife durch Sprung zu l_2.

Andernfalls hat t die Form c(t_1,t_2) und wird nun mittels destruct, pop(1) zerlegt, so daß auf dem Stack nun t_1, t_2 liegt. t_2 wird in der Position von t abgespeichert, mit push(2), store($nlv + npv + 1 + 3$). Danach wird das Pattern p durch den Code cp gematcht, wobei im Fehlerfall bei l_1 die Schleife erneut begonnen wird. Ansonsten wird cc, der Code für c abgearbeitet und hinterläßt $dead', status'$ auf dem Stack. Falls $status' = BROKEN =$ false, dann wird die Schleife verlassen, wobei $dead := dead'$ gesetzt wird. Andernfalls wird ab l_4 die Zuweisung $dead := dead \wedge dead'$ durchgeführt und die Schleife erneut durchlaufen.

trout

Diese Regel erzeugt, abhängig davon, ob die Strategie für die Ereignisse 'solved', 'bound' und 'dead' Ausgaben definiert, Code, der das Argument auf den entsprechende Wert (num(0), num(1), num(2)) testet und ggf. den Ausgabeterm baut und über den entsprechenden Kanal ausgibt.

trmr

Die beiden Regeln (325)/(326) beschreiben die Übersetzung einer Metaregel. In der ersten Inferenzregel wird der Fall einer Metaregel ohne Argumente betrachtet. Σ enthält die Namenszuordnungen für die lokalen Definitionen.
Der Code reserviert zuerst Platz für die lokalen und Patternvariablen (args(nlv,npv)), danach wird das aktuelle Goal auf den Stack geholt und durch den Code cgp gegen das Goalpattern gematcht. Bei Fehlschlag wird zu l_1 gesprungen, der Stack aufgeräumt und die Metaregel mit $dead =$ true verlassen.
Ansonsten werden die lokalen Definitionen gebaut ct_i, danach der Code ctc für die Konstruktion der Bedingung der Metaregel ausgeführt und ausgewertet. Falls die Bedingung zu false auswertet, wird die Metaregel über l_1 verlassen.
Andernfalls wird der Code zur Konstruktion des Ausgabeterms cto ausgeführt und über out(1,"MR:","\0") ausgegeben. Danach wird ein neuer Satz lokaler Daten angelegt, indem zuerst der LP auf dem Stack gesichert wird. Danach wird Platz für die lokalen Daten geschaffen und die lokalen Daten in den neuen Satz kopiert. Danach wird der LP auf diesen neuen Datensatz gesetzt (setLP).
Als nächstes wird der Term 'res++goals' konstruiert, ausgewertet und wieder in 'goals' gespeichert.
Als nächstes werden alle Datenzuweisungen konstruiert (cat_i) und erst hiernach gespeichert (cv_i), so daß die Zuweisungen auf die alten Werte zugreifen, nicht auf bereits berechnete.
Als folgendes wird das globale Datum 'depth' erhöht und die Ablaufsteuerung über call strategy_k_CA aufgerufen. Nach der Rückkehr dieses Aufrufs wird die Tiefe wieder erniedrigt, der Stack aufgeräumt, der alte Wert von LP restauriert (popLP). Als Ergebnis wird in diesem Fall $dead =$ false gespeichert.
Die Regel (326) behandelt den Fall einer Metaregel mit Argument. Im Vergleich zum vorigen Fall kommt nur noch der Code für das Matchen des Argumentes gegen das Regelpattern dazu: crp, wobei im Fehlerfall die Regel wieder mit $dead =$ true verlassen wird.

Der push($nlv + npv + 2$)-Befehl holt das Argument der Regel auf den Stack

trterm

Die Konstruktion eines Terms[11] für Konstanten ist offensichtlich: Regeln (327)f.
Variablen V(na) werden konstruiert, indem die Codierung der Variablen auf den Stack gelegt wird (pushb(num(Υ(V(na))))) und mittels mkv die Variablenzelle gebaut wird (329).
Die nächsten beiden Regeln (330)f. bauen Listen und Applikationen. Dazu werden zuerst die beiden Argumentterme gebaut und daraus mit mkcons(mkapp) das Ergebnis.
Tupel und Substitutionen werden ähnlich gebaut (332)f. Zuerst werden die Argumentterme gebaut, danach die Anzahl der Argumentterme als num(n) auf den Stack gelegt und mit mktupel(mksubst) das Ergebnis konstruiert.
Variablen TV(na) werden als Zugriffe auf lokale Definitionen, Patternvariablen, lokale oder globale Daten übersetzt, je nachdem, womit na assoziiert ist. Dabei haben lokale Definitionen vorrang vor Daten, da zuerst auf $na \in \mathcal{D}(\Sigma)$ geprüft wird.

trpat

Die Ableitung $(\Upsilon, \Sigma, level, lab, done) \vdash P \rightarrow cp : done'$ steht für die Übersetzung des Patterns P zu Code cp. Dabei enthalten Υ, Σ die Zuordnungen für die Namen der Patternvariablen und Datenconstructoren. $level$ ist die Anzahl von Argumenten, die auf dem Stack liegen, gematcht werden soll das oberste Argument. lab ist ein Codelabel, zu dem im Fall eines fehlgeschlagenen Matches gesprungen wird. In diesem Fall werden $level$ Elemente vom Stack entfernt, bevor der Sprung ausgeführt wird.
$done$ ist eine Menge von Namen von Patternvariablen, die bereits gematcht wurden. Dies braucht man, um zu entscheiden, ob ein Match der erste oder ein weiterer Match einer Patternvariable ist. cp ist der erzeugte Code, $done'$ enthält die Namen der Patternvariablen, die *nach* Abarbeitung von cp bereits gematcht wurden, es gilt $done' = done \cup tp$, tp ist die Menge der Patternvariablen, die von diesem Pattern gematcht werden.
Der Code für das Matchen geht prinzipiell so vor, daß das Argument zuerst zu HNF ausgewertet wird: call evalhnf. Das Argument liegt dabei als oberstes Element auf dem Stack. Danach wird das ausgewertete Argument weiter betrachtet.
Das Matchen einer Konstante (Regel (335)) geschieht einfach durch match(tc). Im Fehlerfall werden alle noch auf dem Stack liegenden zu matchenden Argumente[12] heruntergeholt und zum Endlabel lab gesprungen.
Die beiden Regeln (336)f. beschreiben den Code für das Matchen von Listen und Applikationen. Zuerst wird dabei dann das linke Element gematcht, danach das rechte.
Das Matchen von Tupeln (Regel (338)) geschieht durch Matchen der Argumente von links nach rechts. Dabei wird das Argument durch destruct und pop(1) so zerlegt, daß auf dem Stack n Argumente des Tupels liegen. Die Wurzel des Argumentterms muß nicht auf t gematcht werden, weil durch die Wohlgetyptheit des Programms garantiert ist, daß nur

[11] Ja, es ist eigentlich ein Graph.
[12] Die obersten lev Elemente auf dem Stack, beim Eintritt in den Code.

ein Term eben dieser Form entstehen kann: Kein anderer Term kann den Typ `Tupel(tts)` haben.
Beim Matchen von Patternvariablen `TV(na)` sind zwei Fälle zu unterscheiden. Einmal der erste Match dieser Variablen, bei dem einfach nur der Wert des Terms gespeichert werden muß. Zum anderen ist bei weiteren Matches derselben Variablen die Gleichheit der Normalformen des Arguments und des vorher gematchten Terms zu überprüfen.
Regel (339) behandelt den ersten Fall. Die Heapadresse wird einfach im Platz der Patternvariable gespeichert.
Regel (340) behandelt dann den nächsten Fall. Zuerst wird der Argumentterm zu NF ausgewertet: call evalnf. Danach wird der derzeitige Wert der Variablen auf den Stack gelegt: push($lev + \Sigma(na)$), der dann ebenfalls zu NF ausgewertet und wieder gespeichert wird. Danach werden beide verglichen: call compare. Als Ergebnis bleibt true oder false auf dem Stack. Abhängig von diesem Ergebnis wird dann zum Fehlerausgang lab oder zum Ende gesprungen.
Beim Matchen der Pattern `Op(TV(na))` muß vorher noch der Argumentterm darauf geprüft werden, ob seine Normalform eine Variable ist. Wenn dies zutrifft, dann wird der Fehlerausgang angesprungen (Regeln (341)f.).
Umgekehrt muß beim Matchen eines `Var(TV(na))` geprüft werden, ob die Normalform des Arguments eine Variable ist. Andernfalls wird zum Fehlerausgang gesprungen (Regel (343)f.).

Kapitel 6

Einige Verfahren in der Beschreibungssprache

Für Quetzal wurden folgende Verfahren implementiert:

- Reverse Restructuring ([BT94, BGOT95])
- Semantic Matching ([DMS92])
- LRBasic Narrowing ([Her86])
- Outermost Narrowing ([Ech90])

Im folgenden wird nur kurz die Definition der Verfahren gegeben und dann die Implementierung in $\mathcal{QDL}$ präsentiert.
Dabei werden die Definitionen der meisten Hilfsfunktionen weggelassen, wenn diese keine ungewöhnlichen Operationen vornehmen.

6.1 Reverse Restructuring (R^{R})

R^{R} gehört zu den Dekompositionsverfahren. Seine Definition ist in den folgenden Transformationsregeln gegeben:

Eliminate1: $([x =_? s, L_1, \ldots, L_n], \lambda) \rightleftarrows ([L_1\sigma, \ldots, L_n\sigma], \lambda\sigma)$
if $s \equiv x$ or $x \notin \mathcal{V}(s)$, where $\sigma = \{x \leftarrow s\}$

Eliminate2: $([s =?x, L_1, \ldots, L_n], \lambda) \rightleftarrows ([L_1\sigma, \ldots, L_n\sigma], \lambda\sigma)$
if $x \notin \mathcal{V}(s), s \notin X$, where $\sigma = \{x \leftarrow s\}$

Decompose: $([f(s_1, \ldots, s_m) =_? f(s'_1, \ldots, s'_m), L_1, \ldots, L_n], \sigma) \rightleftarrows$
$([s_1 =_? s'_1, \ldots, s_m =_? s'_m, L_1, \ldots, L_n], \sigma)$, $m \geq 0$

Mutate1: $([s =_? f(s_1, \ldots, s_m), L_1, \ldots, L_n], \sigma) \rightleftarrows$
$([r_1 =_? s_1, \ldots, r_m =_? s_m, s =_? g(l_1, \ldots, l_q), L_1, \ldots, L_n], \sigma)$,
where $g(l_1, \ldots, l_q) \rightarrow f(r_1, \ldots, r_m)$ is a renamed rule from $\mathcal{R}$

	and Eliminate1 does not apply
Mutate2:	$([s =_? s', L_1, \ldots, L_n], \sigma) \rightleftarrows$
	$([s =_? g(l_1, \ldots, l_q)\sigma', L_1, \ldots, L_n], \sigma\sigma')$,where $\sigma' = \{x \leftarrow s'\}$,
	$g(l_1, \ldots, l_q) \to x$ is a renamed rule from $\mathcal{R}$ and neither
	Eliminate1 nor Eliminate2 does apply.

Die Implementierung in $\mathcal{QDL}$:

```
strategy rr
data local sigma={};
           m2done=False;
     global thefuncs=getfuncs(trs);
select defaultselect();

eliminate1();
else
{
eliminate2();
else
{
        if (conscompatibel(goal,thefuncs)) {
decompose();

for (n,r) in trs
{
 mutate1(renamerule(r));
else
         if (~m2done)
  {mutate2(renamerule(r));}
}
        }
}
}

output
dead: goal:goals;
bound: goal:goals;
solved: sigma;

rules
eliminate1(): X=?t --> [], if (~isin(vars(t), X)) | (X==t);
sigma:=compose(sigma, sigma1);
        m2done:=False;
```

```
goals:=applytogoallist(sigma1, goals);
output "el1:"++show(sigma1);
where
sigma1={X<-t};
;

eliminate2(): t=?X --> [], if ~isin(vars(t), X);
sigma:=compose(sigma, sigma1);
        m2done:=False;
goals:=applytogoallist(sigma1, goals);
output "el2:"++show(sigma1);
where
sigma1={X<-t};
;

decompose():
#s=?#t --> res, if (opof(s)==opof(t)) ;
        m2done:=False;
output "decompose";
where
res=makegoals(zip(subs(s), subs(t)));
;

        mutate1(l->#r): #s=?#t --> res, if opof(r)==opof(t);
        m2done:=False;
output "m1:"++show(l->r);
where
res=makegoals(zip(subs(r), subs(t)))++[s=?l];
;

mutate2(l->R): #s=?#t --> res;
sigma:=compose(sigma, sigma1);
        m2done:=True;
output "m2:"++show(sigma1)++":"++show(l->R);
where
res=[s=?apply(sigma1, l)];
sigma1={R <- t};
;
.
```

Die Implementierung beinhaltet auch schon einige der in [BT94] aufgeführten Optimierungen. So werden Decompose und Mutate Regeln nur versucht, wenn das Goal das Prädikat

`conscompatibel` erfüllt. Dies bedeutet nichts anderes, als daß die Wurzeloperatoren der linken und rechten Goalseite keine verschiedenen Constructoren sind, in welchem Falle es keine Lösung für das Goal geben könnte.
Als lokale Daten treten in der Strategie einmal die Ergebnissubstitution `sigma` auf, zum anderen ein Flag `m2done`, das auf `True` gesetzt wird, wenn eine Mutate2 Regel ausgeführt wurde.
Das globale Datum `thefuncs` enthält die Functoren des benutzten TRS. Die Funktion `getfuncs` liefert die Liste dieser Operatorsymbole zurück.
Die Selektionsfunktion `defaultselect` nimmt einfach das erste Element einer Goalliste als aktuelles Goal:

```
functions
   defaultselect: (x:xs) =(x,xs);;
.
```

Die Ablaufsteuerung ist simpel. Zuerst werden die beiden Eliminateregeln versucht, nur beim Fehlschlag beider werden die Decompose und Mutateregeln probiert. Dies aber auch nur, wenn das aktuelle Goal keine verschiedenen Constructoren an den Wurzeln der linken und rechten Seite stehen hat (`conscompatibel(goal,thefuncs)`). Die Funktion `renamerule` benennt eine Regel so um, daß sie nur neue Variablen enthält.
Man beachte, daß *sowohl* die Decompose, *als auch* die Mutateregeln probiert werden, egal ob die Decomposeregel am Anfang fehlschlug.
Die Übertragung der Transformationsregeln in Metaregeln sollte offensichtlich sein. Nur die Bedeutung von `m2done` bleibt noch zu klären. Da es ein lokales Datum ist, wirkt seine Veränderung nur auf die noch nicht durchlaufenen Suchebenen aus. Jede Anwendung, die keine Anwendung einer Mutate2 Regel ist, setzt `m2done` auf `False`. Die Bedingung `if (~m2done)...` ist also nur dann `False`, falls der Schritt in der vorangegangenen Suchebene ebenfalls ein Mutate2 Schritt war. In diesem Fall aber wurde in [BT94] gezeigt, daß keine zusätzliche Lösung gefunden werden kann.
Diese simple Optimierung kann den Suchraum erheblich einschränken.
Die verwendete Hilfsfunktion `applytogoallist` wendet eine Substitution auf eine Liste von Goals an. `subs` liefert die Liste von Subtermen eines Terms, also z.B. `subs(f(X,s(Y)))=[X,s(Y)]`. `zip` sollte hinreichend bekannt sein und macht aus einem Paar von Listen eine Liste von Paaren. `makegoals` wiederum nimmt eine Liste von Paaren und gibt eine Liste von Goals zurück:

```
functions
 makegoals: ([]) = [];
    ((x,y):xs) = (x=?y):makegoals(xs);
 ;
.
```

6.2 Semantic Matching (SM)

SM gehört ebenfalls zu den Dekompositionsverfahren.
Seine Definition aus [DMS92]:

Eliminate: $(x =_? s, L_1, \ldots, L_n, \lambda) \rightsquigarrow (L_1\sigma, \ldots, L_n\sigma, \lambda\sigma)$, where $\sigma = \{x \leftarrow s\}$
Decompose: $(f(s_1, \ldots, s_m) =_? f(s'_1, \ldots, s'_m), L_1, \ldots, L_n, \sigma)$
$\rightsquigarrow (s_1 =_? s'_1, \ldots, s_m =_? s'_m, L_1, \ldots, L_n, \sigma)$
Mutate: $(f(s_1, \ldots, s_m) =_? t, L_1, \ldots, L_n, \sigma)$
$\rightsquigarrow (s_1 =_? l_1, \ldots, s_m =_? l_m, r =_? t, \sigma)$,
where $f(l_1, \ldots, l_m) = r$ is a renamed rule in $\mathcal{R}$

Die Implementierung in $\mathcal{QDL}$:

```
strategy sm
data local sigma={};

select defaultselect();

eliminate1();
else
{
eliminate2();
else
{
decompose();

for (n,r) in trs
{
 mutate(renamerule(r));
}
}
}

output
dead: goal:goals;
bound: goal:goals;
solved: sigma;

rules
eliminate1(): X=?t --> [], if (~isin(vars(t), X)) | (X==t);
sigma:=compose(sigma, sigma1);
goals:=applytogoallist(sigma1, goals);
output "el1:"++show(sigma1);
```

```
where
sigma1={X<-t};
;

eliminate2(): t=?X --> [], if ~isin(vars(t), X);
sigma:=compose(sigma, sigma1);
goals:=applytogoallist(sigma1, goals);
output "el2:"++show(sigma1);
where
sigma1={X<-t};
;

decompose():
#s=?#t --> res, if (opof(s)==opof(t)) ;
output "decompose";
where
res=makegoals(zip(subs(s), subs(t)));
;


        mutate(l->r): #s=?t --> res, if opof(l)==opof(s);
output "m:"++show(l->r);
where
res=makegoals(zip(subs(s), subs(l)))++[r=?t];
;
.
```

Die Implementierung unterscheidet sich von der Definition nur insoweit, daß die Eliminateregel um einen *occur-check* erweitert wurde und eine zusätzliche Eliminate2 Regel aufgenommen wurde, die gebraucht wird, um in links-linearen TRS zu rechnen[1].
Ansonsten ähnelt die Implementierung der von R^{R}, nur daß der Test auf ein Goal, das verschiedene Wurzelconstructoren hat, nicht eingeführt wurde.

6.3 LRBasic Narrowing (LRBN)

Beim Narrowing wird versucht, einen Subterm des Goals mit der linken Seite einer Regel zu unifizieren und im Erfolgsfall diesen Subterm durch die rechte Seite der Regel zu ersetzen und den Unifikator anzuwenden.
Dieser Prozess findet ein Ende, wenn sich die linke und rechte Seite des Goals unifizieren laßen.
Eine Narrowingableitung ist eine Folge solcher Schritte:

[1] In [DMS92] wird die Regel in diesem Fall ad hoc hinzugefügt.

Definition 6.1: (Narrowingschritt)

Gegeben Terme s, t über einer Signatur Σ, sowie ein TRS $\mathcal{R}$ und eine Position $p \in \bar{\mathcal{O}}(s)$. Dann sagt man, daß s an Position p mit einer Regel $l \to r \in \mathcal{R}$ zu t ableitet, gdw.

1. $\sigma = mgu(s/p, l)$ existiert.
2. $t \equiv s[p \leftarrow r]\sigma$

Man schreibt dann auch $s \rightsquigarrow_{p,l \to r,\sigma} t$. Wie gehabt, lassen wir die Indices auch weg, wenn sie unbedeutend sind.

Eine Folge $s_1 \rightsquigarrow \cdots \rightsquigarrow s_n$ von Narrowingschritten bezeichnet man als *Narrowingableitung.*

Für Goals $s =_? t$ erweitert man das oben gesagte dadurch, daß Narrowingschritte in s oder in t betrachtet werden, indem die Menge der Positionen, an denen ein Narrowingschritt stattfinden kann auf $GP(s =_? t) := \{1.p | p \in \bar{\mathcal{O}}(s)\} \cup \{2.p | p \in \bar{\mathcal{O}}(t)\}$ erweitert wird. Wir fassen also $=_?$ eigentlich als einen zusätzlichen Constructor der Signatur auf.

Eine erfolgreiche Narrowingableitung eines Goals $s =_? t$ ist eine Narrowingableitung $s_1 =_? t_1 \rightsquigarrow \cdots \rightsquigarrow s_n =_? t_n$, wobei $\theta = mgu(s_n, t_n)$ existiert.

Die Lösung eines solchen Goals ist dann $\sigma = \sigma_1\sigma_2, \cdots \sigma_n\theta$, wobei σ_i der im i-ten Schritt benutzte mgu ist.

Falls es eine erfolgreiche Narrowingableitung mit Ergebnis θ für ein Goal $s =_? t$ gibt, dann gilt $s\sigma =_{\mathcal{R}} t\sigma$ (Beweis siehe [Hul80]).
Vom normalen Narrowing unterscheidet sich LRBN nur dadurch, daß die Menge der Positionen, an denen ein Narrowingschritt versucht wird, eingeschränkt wird.
Statt in jedem Schritt $GP(s_i =_? t_i)$ als mögliche Positionen zuzulassen, wird die Menge beschränkt auf:

$$GP^i_{lr}(s_i =?t_i) := \begin{cases} GP(s_1 =_? t_1) & \text{, falls } i = 1 \\ GP^{i-1}_{lr} \setminus LSM(GP^{i-1}_{lr}, p_{i-1}) \cup New(p_{i-1}, r_{i-1}) & \text{, sonst} \end{cases}$$

Dabei ist p_j die im j-ten Schritt benutzte Position, $l_j \to r_j$ die dort benutzte Regel und
$New(p, r) := \{p.q | q \in \bar{\mathcal{O}}(r)\}$
$LSM(B, p) := \{q \in B | p \leq q \vee q <_l p\}$.
$<_l$ ist ein Prädikat:
$i.q <_l j.p$, falls $i < j$
$i.q <_l i.p$, falls $q <_l p$
Diese Strategie sortiert also alle Positionen links vom letzten Narrowingschritt aus.
Die Implementierung in $\mathcal{QDL}$:

```
strategy lrbasicn
data
local sigma={};
            gpositions=goalnonvarposis(goal);
```

```
select defaultselect();

 unifyable(); else
    for (n,r) in trs
     {
      for p in gpositions
       {
        onestep((n,renamerule(r)),p);
       }
     }

output
dead: goal:goals;
bound: goal:goals;
solved: sigma;

rules
  unifyable(): s=?t --> [], if sigma1 !=%;
              sigma:=compose(sigma,sigma1);
            where
               sigma1=unify(s,t);
  ;

  onestep((n,l->r),p): g --> res, if sigma1!=%;
              sigma:=compose(sigma,sigma1);
              gpositions:=lrbasicpositions(gpositions,p,r);
            output n++" at "++show(p);
             where
               sigma1=goalunify(g,p,l);
               res=[goalapply(g,p,r,sigma1)];
  ;
.
```

Die Ablaufsteuerung prüft zuerst, ob die beiden Goalseiten unifizierbar sind (`unifyable`), falls nicht wird für jede Regel und jede Position ein Narrowingschritt versucht (`onestep`). Das globale Datum `gpositions` enthält anfangs GP^1_{lr} und wird bei jedem erfolgreichen Narrowingstep in `onestep` entsprechend verändert (die Funktion `lrbasicpositions` leistet dies.). Die Funktion `goalunify` versucht einen Subterm des Goals mit dem gegebenen Term `l` zu unifizieren und gibt `%` zurück, falls die nicht gelang. `goalapply` ersetzt den gegebenen Subterm des Goals durch `r` und wendet die Substitution `sigma1` auf das gesamte Goal an.

6.4 Outermost Narrowing (ON)

Im Vergleich zum normalen Narrowing, werden beim ON die Positionen dahingehend eingeschränkt, daß ein Narrowingstep nur an den äußersten Positionen versucht wird und nur beim Fehlschlag eines solchen Versuches eine innere Position in Betracht gezogen wird. Dabei wird trotzdem an jeder möglichen Position jede mögliche Regel versucht.
Die Implementierung in $\mathcal{QDL}$:

```
strategy outermostn
data
local sigma={};
              gpositions=outermostgoalposis(goal);
              succeeded=False;

select defaultselect();

 unifyable(); else
    for p in gpositions
     {
      for (n,r) in trs
       {
onestep((n,r),p); => {succeeded:=True;}
       }
     if (succeeded)
        {
          break;
        }
     }

output
dead: goal:goals;
bound: goal:goals;
solved: sigma;

rules
  unifyable(): s=?t --> [], if sigma1 !=%;
              sigma:=compose(sigma,sigma1);
            where
               sigma1=unify(s,t);
  ;

  onestep((n,l->r),p): g --> res, if sigma1!=%;
              sigma:=compose(sigma,sigma1);
```

```
              gpositions:=outermostgoalposis(head(res));
              succeeded:=False;
            output n++" at "++show(p);
             where
               sigma1=goalunify(g,p,l);
               res=[goalapply(g,p,r,sigma1)];
  ;
.
```

Diese Implementierung unterscheidet sich nur in der Ablaufsteuerung vom LRBN.
Das Datum `succeeded` steuert den Abbruch weiterer Narrowingversuche, sobals ein Versuch an einer Position erfolgreich war. Die Positionen sind bei dieser Strategie so vorsortiert, daß die äußeren am Anfang stehen (wird von der Funktion `outermostgoalposis` erledigt).
Mit der inneren `for (n,r) in trs` Schleife werden alle Regeln an der Position `p` probiert. Falls ein solcher Schritt erfolgreich war, wird `succeeded` auf `True` gesetzt. Mit `if (succeeded) break;` wird die äußere Schleife verlassen, sobald wenigstens ein erfolgreicher Narrowingschritt stattfand. Innere Positionen werden dann nicht mehr versucht.

Teil II

Implementierung

'Ein Plan muß lange gereift und in allen Phasen durchdacht sein, bevor er zur Anwendung kommt'

Indiana Jones

'Hat hier mal jemand einen Büchsenöffner?'

Pandora

Kapitel 7

Vorbemerkungen

Die in dem vorhergehenden Teil definierten Sematiken für das System, die abstrakte Maschine und die Übersetzung von der Zwischendarstellung zu $\mathcal{QAM}$-Code sind 'idealisiert'. Es wurde keine Rücksicht auf reale Computer genommen, auf denen sie letztlich implementiert werden müssen, und die einige Einschränkungen erzwingen.
Die Beachtung dieser Einschränkungen in den Semantiken selbst, etwa die Beschränkung von Zahlen auf den Bereich $\{0, .., 2^{32} - 1\}$, würde ein Vielzahl von Zusatzbedingungen und Spezialfällen erzeugen, die die Sicht auf die grundlegenden Sachverhalte nur verwischt.
Das Verhalten einer Implementation bei Verletzung dieser zusätzlichen Bedingungen und Erweiterungen der Implementation in Bezug auf die Spezifikation unterliegen verschiedenen Überlegungen des Programmierers.
Einerseits muß die Effizienz der Implementierung ein Ziel sein. So wäre etwa die Entscheidung, Zahlen als 'bignums'[1] darzustellen, abzuwägen gegen den Nutzen dieser Darstellung im Gegensatz zur Darstellung als Wert der Wortlänge des Rechners. Ist die Wahrscheinlichkeit, daß solche bignums oft Verwendung finden eher gering, ist sicher die Darstellung als Zahl fester Länge aus Effizienzgründen vorzuziehen.
Andererseits ist auch die Portabilität der Implementierung ein wichtiger Faktor. Dies verhindert aber oft die Benutzung effizienter Konstrukte eines Rechners. So kann ein SPARC Rechner auf Wörter nur an Adressen zugreifen, die glatt durch die Wortlänge teilbar sind, während ein M68k Prozessor auch an ungeraden Adressen auf Wörter zugreifen kann[2]. Dies ist selbstverständlich schon bei der Auslegung der Datenstrukturen zu berücksichtigen. Ein Ausweg ist die Bereitstellung spezialisierter Versionen für Operationen, wie das Auslesen eines Datenworts aus dem Heap, die die Eigenschaften der verschiedenen Rechner berücksichtigen und schlimmstenfalls eine allgemeine, ineffiziente Routine zur Verfügung stellen, um die Portierung auf beliebige Rechner zu sichern.
Die Implementierungsentscheidungen bei der Implementierung von Quetzal wurden unter diesen Gesichtspunkten getroffen. Da die Rechner, auf denen Quetzal implementiert wurde, beide eine Wortlänge von 32bit haben, wurden Zahlen auf diese Länge beschränkt. Da aus

[1] Worunter hier Zahlen verstanden werden, die 'wachsen' können, also nicht von Anfang an auf eine bestimmte Länge, etwas 32bit, festgelegt sind.

[2] Jedenfalls, wenn es wenigstens ein M68020 ist

vorangegangenen Erfahrungen mit dem **PaNaMa** System die Signifikanz der Speichereffizienz der Implementierung bekannt war, wurden die kritischen Datenstrukturen, insbesondere der Aufbau der Heapzellen darauf ausgelegt, möglichst kompakt zu sein, auch wenn dadurch der Zugriff auf diese Strukturen verlangsamt wurde.
Im Gegensatz dazu wurde bei unkritischen Strukturen, etwa der Zwischendarstellung, der Einfachheit und dem effizienten Zugriff der Vorzug gegeben.
Da in Quetzal Zeichenketten nur als Listen von Zeichen definiert waren, obwohl sie sehr häufig in Programmen auftreten dürften, wurde der Implementierung ein neuer Datentyp 'String' hinzugefügt. Dies erfordert Ergänzungen bei der Übersetzung in die $\mathcal{QIR}$, bei der Implementierung der $\mathcal{QAM}$ und natürlich bei der Übersetzung $\mathcal{QIR} \rightarrow \mathcal{QAM} - Code$.
Aus Speichereffizienzgründen wurde die Anzahl aller definierten Objekte zusammen auf 2^{16} beschränkt. Der Grund hierfür ist, daß jede Heapzelle, die einen Datenconstructor, Funktionssymbol oder Useroperator repräsentiert, ebenfalls den Index dieses Symbols enthält. Die Anzahl scheint aber für alle jetzt vorstellbaren Anwendungen ausreichend, da sie sich ja zur Laufzeit eines Programmes nicht mehr ändern kann. Im Vergleich dazu, wird der Index einer Variablen durch eine 32bit Zahl repräsentiert, da zur Laufzeit ja immer neue Variablen erzeugt werden können.
Der Rest dieses Teils gliedert sich, wie folgt: Als nächstes werden die Komponenten der Implementierung vorgestellt, sowie ihre Semantik und das Zusammenspiel mit anderen Teilen des Systems geschildert, sofern dies nicht schon in anderen Abschnitten der Arbeit geschehen war.
Im Anschluß daran werden dann einige interessante Details der Implementierung geschildert, die über den Rahmen einer offensichtlichen Implementierung hinausgehen.
Zum Schluß wird noch auf die Erfahrungen bei dem gewählten Weg der Implementierung auf drei verschiedenen Betriebssystemen eingegangen.

Kapitel 8

Systemkomponenten

Wie schon in der Einleitung erwähnt, besteht Quetzal aus zwei Hauptkomponenten: Einmal dem Frontend, das die Kommunikation mit dem Benutzer übernimmt und zum zweiten das Backend, welches die eigentliche Implementierung des Quetzal-Systems ist. Diese beiden Komponenten sind nicht nur konzeptionell, sondern auch räumlich getrennt, sie bestehen jeweils aus einem eigenen Programm und das Zusammenspiel zwischen ihnen beschränkt sich auf eine genau festgelegte Kommunikation über einen gemeinsamen Kanal.
Dieser Ansatz hat den Vorteil, daß die Frontends leicht ausgetauscht werden können, ohne Rückwirkungen auf den eigentlich wichtigen Teil des Systems. Implementiert wurden zwei Frontends, einmal ein Terminal FE, welches die Kommunikation mit dem Benutzer über ein normales ASCII Terminal erlaubt. Zum anderen wurde ein Batch Frontend implementiert, das es erlaubt, Quetzal als Compiler für ein Skript einzusetzen.
Ein drittes Window Frontend ist denkbar, welches die Kommunikation mit dem User über XWindow Komponenten, wie Fenster und Knöpfe erlaubt. Aus Zeitgründen wurde dieses Frontend nicht mehr implementiert, allerdings wird später beschrieben werden, wie es eingebunden werden könnte[1].
Als nächstes wird zuerst der Aufbau des FE, sowie dessen Benutzung vorgestellt. Die Kommunikation des FE mit dem BE wird ebenfalls hier behandelt. Der daran anschließende Abschnitt stellt das Backend vor, seine Komponenten und deren Zusammenspiel untereinander und mit dem FE.

8.1 Frontend

Der Aufbau des Frontends ist in Abbildung 8.1 dargestellt. Dabei ist der *Driver* das zentrale Element. Er nimmt die aufbereiteten Benutzereingaben von dem *IOInterface* entgegen, verarbeitet sie und zerlegt sie in elementarere Aktionen.
Das IOInterface ist das Bindeglied zum Benutzer und der einzige Teil, der in den einzelnen Frontends verschieden ist. Es sorgt dafür, daß die Benutzereingaben in genormte Aktionen

[1] In der Tat müssen dazu nur einige *Routinen* ausgetauscht werden, da das Frontend ebenfalls modular aufgebaut ist.

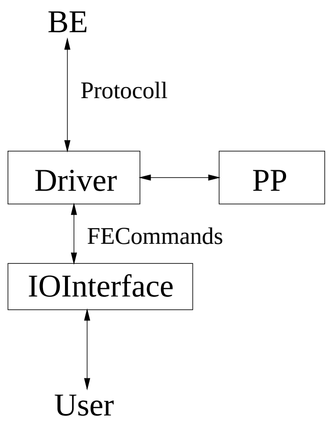


Abbildung 8.1: Aufbau des Frontend

für den Driver zerlegt werden. Das Interface ist auch für die Ausgabe zuständig. Der Driver ruft für jede Ausgabe eine Routine im IOInterface auf, die dann die Ausgabe vornimmt.
Der Präprozessor wird vom Driver aufgerufen. Er bearbeitet ein Eingabefile und nimmt einfache Ersetzungen vor und entfernt Kommentare. Seine Funktionsweise ist dabei an den C-Präprozessor (cpp) angelehnt, allerdings sind nur einfache Macroersetzungen zulässig. Die Einbindung von verschiedenen Files wird ebenfalls vom Präprozessor übernommen, mittels der `#include` *filename* Direktive.
Die Komponenten arbeiten, soweit wie möglich, asynchron. So veranlaßt eine `compile` Aktion[2] nur, daß der Präprozessor das Eingabefile bearbeitet, das IOInterface kann während dessen weiter Benutzereingaben bearbeiten. Allerdings ist auch hier garantiert, daß sich keine Kommandos überschneiden können. Wenn der Präprozessor das File bearbeitet hat, dann meldet er dies dem Driver zurück, der es daraufhin an das Backend zum Parsen schickt.
Diese Asynchronität ist notwendig, um z.B. das Window Frontend zu realisieren, da dort dauernd Benutzereingaben bearbeitet werden müssen, um Beeinträchtigungen beim Bildschirmaufbau und bei der Reaktivität der Applikation zu vermeiden[3].
Das Frontend nimmt eine Reihe von *Aktionen* vom Benutzer entgegen und wertet sie aus, schickt sie ggf. an das Backend.
Die einzelnen Aktionen sind

- `compile` *filename* Das gegebene File wird durch den Präprozessor bearbeitet und anschließend an das Backend geschickt, das den Inhalt dann parsed und zur $\mathcal{QIR}$ hinzufügt.

- `quit` Beendet das Programm.

- `run` *filename* Das gegebene File wird als neue Quelle von Aktionen geöffnet.

[2] s.u.
[3] Was daran liegt, daß XWindow den Bildschirmaufbau den einzelnen Applikationen überläßt.

- `settrs` *`name`* Setzt den Namen des aktuellen TRS auf *name*.
- `settactic` *`name`* Setzt den Namen der aktuellen Taktik auf *name*.
- `setstrat` *`name`* Setzt den Namen der aktuellen Strategie auf *name*.
- `setreport` *`report-level`* Hiermit wird festgelegt, welche Ergebnisse/Protokolle man vom Lösungsprozeß eines Goals sehen will. *report-level* ist eine Folge von Angaben aus:
  - `all` Ist die Abkürzung für die restlichen Angaben zusammen.
  - `result` Resultatsausgaben sollen angezeigt werden.
  - `mr` Aufrufe einer Metaregel sollen angezeigt werden.
  - `bound` Tiefenüberschreitungen sollen protokolliert werden.
  - `dead` Sackgassen sollen ausgegeben werden.
  - `stats` Am Ende soll eine Gesamtstatistik ausgegeben werden.

  Z.B. werden nach `setreport stats result` nur Ergebnisse und die Statistik angezeigt.
- `setverbosity` *`output-level`* Dies legt die Ausführlichkeit der restlichen Ausgaben fest. *output-level* ist eine Folge von:
  - `all` Ist die Abkürzung für die restlichen Angaben zusammen.
  - `verbose` Alle Warnungen, Bemerkungen werden angezeigt.
  - `terse` Nur Fehlermeldungen werden angezeigt.
- `status` Zeigt den aktuellen Status des Systems an, z.B.

```
Quetzal>status
TRS: unset
Strategy: rr
Tactic: unset
Depth: infinite
AM Code size: 1930
# Objects defined: 63
AM Heap size: 400000
AM stack size: 10000
OutputLevel:  terse verbose
ReportLevel:  result mr bound dead stats
```

  Die Ausgabe sollte sich selbst erklären.

- `solve` *goalname* Startet das Lösen eines Goals *goalname*. Als TRS, Taktik und Strategie werden die aktuell gesetzten verwandt. Die statische Semantik der definierten Objekte wird geprüft, danach werden die Definitionen compiliert[4]. Danach wird das TRS und die Attributliste erzeugt und der Code für die Strategie aufgerufen. Z.B.:

```
Quetzal> show g1
g1: =?(s(z),s(z))
Quetzal>status
TRS: t1
Strategy: rr
Tactic: ta1
Depth: 2
AM Code size: 16260
# Objects defined: 51
AM Heap size: 400000
AM stack size: 10000
OutputLevel:  terse verbose
ReportLevel:  result stats
Quetzal>solve g1
Solved:{}
Completed.
Found 1 solutions
ran into 0 dead ends
hit bound 4 times
23902 cells on heap used
564 stack cells used
0 GCs occured
11 nodes computed in total
634746 AM commands executed
time needed was 1.190
```

  Wobei hier vorher `setreport result stats` gesetzt war und das TRS die Regeln für die Addition auf natürlichen Zahlen enthielt.

- `kill` *objs* Löscht die Objekte *objs*. Dabei ist *objs* eine Folge von Spezifikationen der Form *name* : *type*. So löscht `kill g1:goals` das Goal `g1`. Man kann eine der Angaben weglassen, es wird dann ein Wildcard angenommen.

- `show` *objs* Zeigt die angegebenen Objekte an, die Spezifizierung erfolgt, wie bei der `kill` Aktion.

[4]Natürlich nur, wenn dies noch nicht vorher geschehen ist, oder sich inzwischen Änderungen ergeben haben.

- `eval` *expr* Dies wertet den Ausdruck *expr* zu Normalform aus und zeigt in an.
- `term` *name*, `confluent` *name* sind in der aktuellen Version nicht implementiert.

8.1.1 Eine Beispielsitzung

Zum Schnuppern folgt hier eine kleine Vorführung des Terminal Frontends.

```
isthesin@moos qterm
Quetzal version 1.0 on Jul 11 1996
(c) 1996 by Stephan Thesing
isthesin@techfak.uni-bielefeld.de

Quetzal>compile ../Implementation/prolog
Parsed
Quetzal>compile ../Implementation/ex
Parsed
Quetzal>show :rules t1 ta1 g1
r1: ->{add{z,X},X}
r2: ->{add{s(X),Y},s(add{X,Y})}

t1: (TRS) r1 r2

ta1: (TACTIC) r1>r2  r1,r2

g1: =?{s(z),s(z)}
Quetzal>settrs t1
Quetzal>settactic ta1
Quetzal>setstrat rr
Quetzal>setreport stats
Quetzal>setdepth 4
Quetzal>solve g1
static semantics...syntax checked
static semantics...typing checked
Completed.
Found 1 solutions
ran into 0 dead ends
hit bound 6 times
47294 cells on heap used
782 stack cells used
1 GCs occured
22 nodes computed in total
1260845 AM commands executed
```

```
time needed was 2.500
Quetzal>quit
isthesin@moos
```

8.1.2 Optionen beim Aufruf der Programme

Das Programm besitzt relativ wenige Optionen:

```
isthesin@moos qterm --help
Usage: qterm [options] [filename]
where options comes from:
-h   : print this help
--help: synonym for -h
-v   : print version and exit
--version: synonym for -v
--stack N : set stack to N cells
-s N      : synonym for --stack
--heap N  : set heap to N cells
-p N      : synonym for --heap
--name-space N       : set name space to N kB
-n N                 : synonym for --name-space
--filename-space N   : set filename space to N kB
-f N                 : synonym for --filename-space
--definition-space N : set definition space to N kN
-d N                 : synonym for --definition-space
```

Die Optionen `--name-space`, `--filename-space` und `--definition-space` setzen die Größen der jeweiligen Bereiche für den Präprozessor, der diese Bereiche statisch anlegt. Da der Initialwert hinreichend groß ist, sollte man nie in die Verlegenhait kommen, sie zu benutzen. Die restlichen Optionen sollten klar sein.
Dem Batch Frontend muß als zwingender Parameter der Name eines Skripts gegeben werden. Die Ausgabe erfolgt so, daß Resultate auf `stdout` und Protokoll auf `stderr` ausgegeben wird.

8.1.3 Das Protokoll FE $\Longleftrightarrow$ BE

Da das Frontend und das Backend auch räumlich getrennt sind, können sie nur über einen gemeinsamen Kanal kommunizieren. In der vorliegenden Implementierung wurde als Kommunikationskanal eine Unix Pipe[5] benutzt.
Die Benutzung anderer Mechanismen, etwa *shared memory* oder die *messages* von SYSV wären auch möglich gewesen, sind allerdings nicht allzu portabel und weisen einige Einschränkungen auf, etwa in der Länge der übermittelbaren Daten. Allerdings ist es einfach

[5]Bzw. Messages in der Implementierung unter AmigaDOS.

möglich, die Kommunikation auf solche Weisen umzustellen, da die Kommunikationsroutinen in einem Modul eingekapselt sind und Kommunikationsanforderungen nur über höhere Routinen und einen eigenen Datentyp `BECommunication` geschehen, der völlig unabhängig ist von der Implementierung der Kommunikation selbst.
In der Praxis sieht etwa die Anforderung des Frontends an das Backend, ein File zu parsen, so aus:

```
 BESendCommunication(PrCompile, filename, NULL);
```

Antworten aus der Kommunikation werden meistens asynchron behandelt, d.h. bis zum Eintreffen einer Antwort wird mit anderen Dingen fortgefahren. Das Abholen der Antwort geschieht dann einfach mir

```
BECommunication *com;

com=GetBECommunication();

if (com->Type==PrOk) ...
```

Das Format der Kommunikation selbst, also der Daten, die über die Pipe gehen, kann sich dabei je nach verwandter Kommunikationsmethode ändern.
Die Typen der Kommunikation entsprechen in etwa denen der Aktionen, mit zusätzlichen Typen für eine positive oder negative Antwort und Typen für die Ausgaben eines Goallösungsprozesses.
Man beachte, daß das so gewählte Konzept sich auch dahingehend erweitern läßt, daß die beiden Prozesses auf *verschiedenen* Rechnern laufen können. Wenn auch nichts anderes, so erspart es doch das Einloggen auf einem schnelleren Rechner und das starten beider Prozesse dort.

8.1.4 Implementierungsumfang

Die Implementierungszeit für das Frontend belief sich auf ca. eine Woche, wobei hier noch die meiste Zeit für die Spezifikation des Protokolls und der internen Asynchronität verwandt wurde.
Die Größe der Implementierung ist bescheiden:

FE	source size	LOC	executable size
Terminal	60000	3500	29304
Batch	60000	3500	29600

Dabei bedeutet LOC 'Lines Of Code' und die Größe des ausführbaren Programms bezieht sich auf die SPARC10 Implementierung.

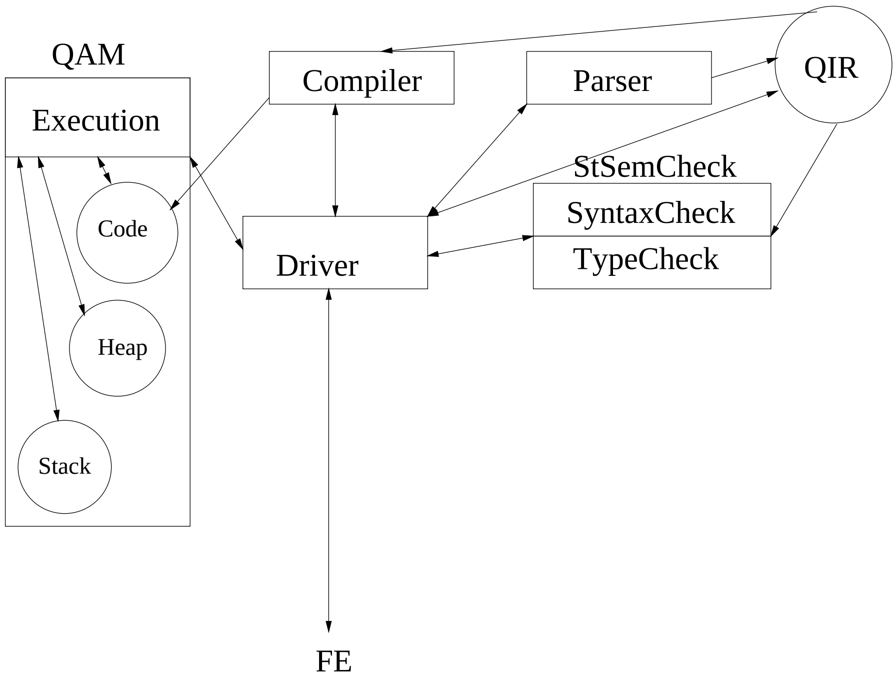


Abbildung 8.2: Aufbau des Backends

8.2 Backend

Der Aufbau des Backends ist wesentlich umfangreicher, als der des Frontends. Er ist in Abbildung 8.2 dargestellt.

Der *Driver* ist für die Kommunikation mit dem FE zuständig und steuert die restlichen Komponenten.

Der *Parser* generiert aus einem File in der $\mathcal{QDL}$ dessen Darstellung in der $\mathcal{QIR}$ und fügt sie zu der bereits vorhandenen Darstellung hinzu. Dabei werden kaum Konsistenzchecks durchgeführt, nur solche, die sicherstellen, daß die erzeugten Terme auch wirklich $\mathcal{QIR}$-Terme sind, wie in Abschnitt 4.3.1 definiert.

Der *StSemCheck* nimmt die Prüfung der statischen Semantik vor. Er ist unterteilt in die Prüfung der syntaktischen Zusatzbedingungen und in die Typprüfung. Dabei nimmt die Typprüfung die Überprüfungen der Nebenbedingungen vor, die in Abschnitt 4.5 durch die Inferenzregeln futyp, futyp', pattype, termtype, sttype definiert werden. Die syntaktische Prüfung kümmert sich um die restlichen Bedingungen und wird vor der Typprüfung vorgenommen.

Der *Compiler* erzeugt aus der $\mathcal{QIR}$-Darstellung, welche den statischen Semantik-Check überlebt hat, den $\mathcal{QAM}$-Code, gemäß den Übersetzungsregeln aus Abschnitt 5.3.3.

Die $\mathcal{QAM}$ wurde direkt anhand der gegebenen Semantik aus Abschnitt 5.2 implementiert. Auf die Details der Implementierung wird weiter unten eingegangen.

8.2.1 Arbeitsweise des Backends

Die Arbeitsweise des Backends läßt sich am leichtesten anhand eines kleinen Beispiels erklären.
Wenn der Driver vom Frontend eine *Solve* Kommunikation erhält, prüft er zuerst, ob die darin enthaltenden Objekte, TRS, Taktik, Strategie, Goal, überhaupt definiert sind. Ist dies der Fall, wird nachgesehen, ob das augenblickliche Programm schon übersetzt wurde und bereits in $\mathcal{QAM}$-Code vorliegt. Falls dies noch nicht geschehen ist, wird der Stsem-Check durchgeführt und danach die Compilierung des Programms. Sollte jetzt immer noch kein Fehler aufgetreten sein, wird die Konstruktion des TRS und der Attributliste, sowie des Goals, im Heap durchgeführt und ein Stackframe erzeugt, daß dem entspricht, welches beim Eintritt in den Code für eine Strategie erwartet wird[6]. Jetzt wird der Code für die Strategie aufgerufen und nach dessen Rückkehr wird dann dem Frontend die Beendigung der *Solve*-Kommunikation mitgeteilt.

8.2.2 Änderungen an den Semantiken

Die Implementierung der Semantiken macht einige Änderungen nötig. So ist, wie bereits erwähnt, der Wertebereich von Zahlen auf 32 bit festgelegt. Bei Überschreitung dieses Wertebereichs beendet die $\mathcal{QAM}$ die Ausführung des Programms mit einem Fehler.
Ebenfalls zu einem Fehlerabbruch kommt es, wenn eine bestimmte Anweisung keine Semantik besitzt, etwa ein div Befehl, der versucht, durch 0 zu dividieren.
Da in der Definition der Semantik der $\mathcal{QAM}$ Stack, Code und Heap beliebig groß werden können, dies aber in der Realität natürlich nicht machbar ist, mußte die Semantik bei der Überschreitung der vorgegebenen Grenzen für die Größe dieser Komponenten ergänzt werden. Es wird immer versucht, den Stack und den Codebereich durch Vergrößerung des Speicherbereichs zu erweitern[7]. Sollte des virtuelle Speicher des Prozesses erschöpft sein, wird mit einem Fehler abgebrochen. Im Gegensatz hierzu kann der Heap nicht mehr wachsen, sondern bleibt in seiner Größe bestehen. Statt dessen wird eine Garbage Collection durchgeführt, die alle unbenutzten Zellen des Heaps entfernt, indem sie die benutzten Zellen zusammenpackt. Hierbei findet ein einfacher Copy Garbage Collector Anwendung[8]. Der *root set*, d.h. die Ausgangszeiger in den Heap sind dabei all die Heapadressen, die zum Zeitpunkt der GC auf dem Stack liegen.
Die Einführung von Strings als zusätzlichem Datentyp machten einige Änderungen notwendig. Neben den trivialen Änderungen in der $\mathcal{QIR}$ wurde auch ein neuer primitiver Heapzellentyp string in die $\mathcal{QAM}$ eingeführt. Die sich ergebenden Änderungen in der Compilierung sind natürlich offensichtlich. Etwas komplizierter waren die resultierenden Änderungen an der Semantik einiger $\mathcal{QAM}$-Befehle, nämlich destruct, ishnf, match, cmp. Diese wurden daraufhin erweitert, daß der Fall eines Strings als Argument zu einem dieser Befehle einmal als äquivalent zum Auftreten einer leeren Liste nil angesehen wird, dann,

[6] Siehe Abbildung 5.2.
[7] Mittels der Unix `realloc()` Routine
[8] Siehe [Wil92].

wenn der String leer ist. Das Auftreten eines nicht leeren Strings wird nun analog zum Auftreten einer Char-Liste `c` behandelt.

8.2.3 Implementierungsumfang

Der Umfang des Backends ist wesentlich größer, als der des Frontends. Trotz des komplexeren Aufbaus und größeren Schwierigkeit der Implementierung, gelang die Fertigstellung in der relativ kurzen Zeit von nur vier Wochen. Einige Zahlen, um einen Eindruck von dem Unfang der Arbeit zu bekommen:

source size	LOC	executable size
340774	15343	185716

Trotz des Umfangs traten während der Implementierung kaum Schwierigkeiten auf. Abgesehen von offensichtlichen Bugs wurden keine grundlegenden Designfehler entdeckt.

Kapitel 9

Details der Implementierung

9.1 Heap

In der Implementierung wurden Heapzellen variabler Größe benutzt, da subst, t Zellen immer eine dynamische Länge haben, die von der Anzahl der Elemente in dem Tupel, der Substitution abhängt.
Außerdem ist nicht einzusehen, warum man überhaupt im Heap mehr Platz als unbedingt nötig verschwenden soll, so daß jede Heapzelle nur die Information trägt, die sie wirklich braucht. Als Mindestgröße einer Zelle ergeben sich daraus 6 Bytes, 2 Bytes für den Identifizierung der Zelle und 4 Bytes für eine Indirektionsadresse, die bei der Garbage Collection benötigt wird. Bemerkenswert ist die Implementierung der Stringzellen. Einerseits muß man den eigentlichen String ebenfalls irgendwo speichern, was aber wegen der möglichen Länge eines Strings nicht im Heap geschieht, sondern in einer seperaten Stringtabelle. Im Heap steht dann nur noch der Index in diese Tabelle und zusätzlich ein Index in den dort abgelegten String. Bei der Kreiierung der Stringzelle ist dieser zweite Index 0, zeigt also auf den Anfang des Strings. Wenn jetzt dieser String mit einem destruct Befehl zerlegt wird, müßte ein neuer String angelegt werden, der sich vom alten nur dadurch unterscheidet, daß er ein Zeichen später beginnt. Ein flotter Ausweg ist es, einfach den alten String weiter zu benutzen und nur den Index in den String um eins zu erhöhen. Das erlaubt eine schnellere Verwaltung, ohne irgendwelche Probleme bei der Implementierung hervorzurufen.
subst, t Zellen im Heap tragen neben der Zellenkennung die Anzahl der folgenden Elemente und daran anschließend die Addressen dieser Elemente. Damit ergibt sich die Größe dieser Zellen zu $2 + 4 + n * 4$ Bytes, wobei n die Anzahl der Elemente ist.
Heapadressen sind als Offset der Zelle vom Anfang des Heaps implementiert.

9.2 GC

Der Garbage Collector in Quetzal ist simpel, insgesamt umfaßt er nur 173 Zeilen (großzügig formatierten) C-Code. Der Algorithmus ist einfach: Allen Heapadressen auf dem Stack

wird gefolgt[1], die dabei angetroffenen Zellen werden in einen zweiten Heap kopiert, die Zelle wird mit einer Indirektion auf die neue Adresse im neuen Heap überschrieben, die neue Adresse wird zurückgegeben. Allen Adressen in der Heapzelle wird daraufhin ebenfalls gefolgt.
Der einzig beachtenswerte Punkt ist die Behandlung von `string` Zellen. Wird eine solche Zelle angetroffen, dann wird in der Stringtabelle dieser String als benutzt gekennzeichnet. Am Schluß werden dann alle unbenutzten Strings freigegeben.
Die Performanz des Copy Collectors im Vergleich mit einem Mark/Compact Collector gibt im den Vorzug, trotz des höheren Speicherbedarfs. Andere Techniken, wie Reference Counting sind wegen der möglichen Zyklen im Heap nicht anwendbar.

9.3 Stack

Im Gegensatz zu Heapzellen und $\mathcal{QAM}$-Instruktionen sind Stackzellen von fester Länge. Der Grund dafür ist einfach, daß der Vorteil der Speicherersparnis hier zurück tritt hinter den Vorteil der einfacheren Adressierung. So greifen viele $\mathcal{QAM}$-Befehle auf das n-te Element des Stacks von oben zu. Bei variabel langen Stackzellen müste man die Länge der oben liegenden n Zellen addieren, um die Adresse zu bekommen. Im Gegensatz dazu erfolgt bei fester Länge der Zugriff auf die n-te Zelle in konstanter Zeit. Im übrigen läßt sich mit variabel langen Zellen kaum Platz sparen, da die meisten Zellen sowieso Heapadressen oder Codeadressen sind, die beide 6 Bytes benötigen.
Eine Stackzelle besteht zum einen aus einem Stacktag, der den Inhalt der Zelle beschreibt: Stackadresse, Heapadresse, Codelabel, Basiswert oder ein Black Hole. Zum anderen trägt jede Zelle noch ein Byte, das bei Basiszellen zur Identifizierung des Basiswertes dient. Die restlichen 4 Bytes sind dann für Heap-, Stack-, Codeadressen oder den eigentlichen Basiswert reserviert.
Der Stackpointer ist nun einfach als Index in ein Array von solchen Stackzellen realisiert.

9.4 $\mathcal{QAM}$-Code

$\mathcal{QAM}$-Instruktionen sind ebenfalls variabel lang. Das ist einleuchtend, da die meisten Befehle ohne Parameter auskommen und so in einem Byte kodiert werden können. Außerdem benötigt man nie relative Adressen[2] zu einem Codestück, sondern immer nur absolute, die einfach als Offset vom Codeanfang implementiert werden können, ohne daß Schwierigkeiten auftauchen.
Die Länge eines Befehls reicht von einem Byte, bis zur 10 Bytes für den `out` Befehl. Die Einsparung an Codeplatz ist dabei offensichtlich.
Das Handling von Labeln im Code ist denkbar einfach gelöst. Beim Addieren von Code

[1] Über Indirektionen `ind(p)` hinweg.
[2] D.h. n Befehle weiter/zurück

wird für jedes Label, das angesprungen werden soll[3] eine Referenz auf dieses Label hinzugefügt. Ebenso können Label definiert werden. Zum Schluß werden dann die Refernzen durch die Werte der Label ersetzt (*Backpatching*), eine alte Technik aus dem Assemblerbau. Für die Bereitstellung lokaler Label gibt es eine eigene Funktion, die anhand eines Zählers bei jedem Aufruf ein neues, unbenutztes, Label liefert.
Als Beispiel der Implementierung der Semantik der $\mathcal{QAM}$-Befehle hier die Realisierung des jt l und des storeGP(n) Befehls (siehe Regeln (277, 298)):

```
  case INSjt:
istrue=(QAMStack[QAMSP--].BTag==BAStrue);

        if ( istrue)
  QAMCP=QAMGETLONG(QAMCode+QAMCP);
        else
  QAMCP+=4;
      return 0;
      break;

....
  case INSstoreGP:
      num=QAMGETLONG(QAMCode+QAMCP);
      QAMCP+=4;

      cell=QAMStack[QAMSP--];

      QAMStack[QAMGP-num+1]=cell;
      return 0;
      break;
```

Dabei dienen die `QAMGETLONG()` Routinen zum Auslesen der Argumente aus dem Code. Der einzige Unterschied besteht darin, daß in der Semantikdefinition der Code Pointer als Index in die Befehlsliste gesehen wird, statt als Adresse von variabel langen Befehlen.

9.5 Compilierung

Die Compilierung ist ziemlich unaufregend. In der Tat ist die Implementierung der Übersetzungsregeln direkt und ohne Umschweife möglich[4].
Eine eigene Routine `AddIns()` dient zum Hinzufügen eines Befehls. Die Indexumgebungen Σ, Υ, Π wurden als Symboltabelle implementiert, in der mittels `FindSE()` Einträge gesucht werden können.
Als Beispiel die Übersetzung von Variablen aus trterm, Regel (329):

[3] Etwa in call evalnf

[4] Was man auch dahingehend interpretiern kann, daß die Semantik ziemlich 'low-level' ist.

```
case QIRV:
      se=FindSE(QIRV, term->Args.V, SYMBOL_ALL);
      AddIns(INSpushb, BASnum, se->Index);
      AddIns(INSmkv);
```

Direkter läßt sich die Semantik kaum übertragen.

9.6 StSemCheck

Der Syntaxcheck ist im großen und ganzen uninteressant, da meistens damit beschäftigt, das Vorhandensein von Definitionen für auftretende Bezeichner zu überprüfen.
Einzig die Implementierung der Regeln sort-lt, sort-of zur Prüfung der Wohlgetyptheit von Goals und Regeln ist bemerkenswert.
Der verfolgte Ansatz ist der, daß bei der Typprüfung (sort-of) jedem Term alle seine möglichen Sorten[5] zugewiesen werden. Variablen werden bei ihrem ersten Auftreten *alle* Sorten als mögliche Typen zugewiesen. Beim weiteren Prüfen der Terme werden dann die Möglichkeiten, welcher Sorte diese Terme haben können, durch ihre Stellungen in anderen Termen eingeschränkt. Sieht man z.B. einen Term der Form `s(X)`, so schränkt dies die Typen der Variable `X` auf `nat` ein[6].
Der Witz der Implementierung liegt nun darin, daß die verschiedenen Sorten als Bitfelder repräsentiert werden. Jeder Sorte wird ein eineindeutiger Index zugewiesen. Die möglichen Sorten eines Terms sind nun als Bitfeld repräsentiert, in dem all die Bits gesetzt sind, denen der Index einer möglichen Sorte entspricht. Die Operationen der Einschränkung und Inklusion von möglichen Typen laßen sich jetzt einfach und effizient als Durchschnitt und Untermenge auf Bitfeldern realisieren.
Der Typcheck nun ist wieder interessanter. Es wird der schon in [Mil78] vorgestellte Algorithmus verwandt, der darauf beruht, die Typen der Terme anfangs durch Typvariablen zu bezeichnen, die dann später durch Typterme ersetzt werden, wenn die Typen dieser Terme mit vorgegebenen Typen unifiziert werden. Als Spezialfall muß man dabei Regel (104) berücksichtigen. Sie behandelt den Fall, daß das Argument ein Userterm, oder ein Tupel von solchen Usertermen ist und darauf ein Operator des Typs `arg --> term` angewandt wird.
Eine korrekte Implementierung müßte für jede Applikation `App(a,b)` die beiden Möglichkeiten, daß sich a mit `term` oder `Tupel([term,...,term])` einerseits oder einem anderen Term andererseits unifizieren läßt, berücksichtigen.
Obwohl möglich, ist es ineffizient, dies zu prüfen. Quetzal macht den Ansatz, daß der Typ von a bereits `arg -->term` sein muß, wenn die Situation aus Regel (104) vorliegt.
Man beachte, daß nur die primitive Funktion `opof` anfangs den fraglichen Typ haben kann. Funktionen können diesen Typ nur haben, wenn in ihren Definitionen `opof` auftaucht.

[5] Da wir Sorteninklusion haben, können es mehrere sein.
[6] Wobei wir hier von einer Signatur ausgehen, die die natürlichen Zahlen als `s,z` Terme der Sorte `nat` repräsentiert.

Kapitel 10

Erfahrungen bei der Implementierung

Quetzal wurde auf drei Betriebssystemen und zwei verschiendenen Rechner gleichzeitig entwickelt:

OS	Architektur
Solaris 2.4	SPARC 10
NetBSD 1.1	MC68k
AmigaDOS 3.0	MC68k

Die beiden ersten Systeme sind Unix Derivate, wobei Solaris an SYSV angelehnt ist und NetBSD ein BSD4.4 Derivat ist. Da dritte System ist ein Multitasking OS, das auf einer shared memory Architektur basiert und im Gegensatz zu Unix keine Memory/Ressourcen Protection bietet.
Der Grund für dieses Vorgehen war die Überlegung, daß Fehler in der Implementierung leichter zu finden wären, wenn man auf unterschiedlichen Plattformen implementiert. Außerdem ergibt sich so die Portabilität des Systems quasi von alleine.
Im Zuge der Implementierung hat sich dieser Ansatz gerechtfertigt. Vor allem die Realisierung auf einem System ohne Memory Protection offenbarte einige Fehler, die ansonsten von dem Betriebssystem geschluckt worden wären.
Als Nachteil kann man ansehen, daß sich die Rechner, obwohl angeblich alle ANSI/POSIX.1 kompatibel, doch in Einzelheiten unterscheiden. Hier wären Namen, Inhalte von (Standard-) Headerdateien zu nennen, ebenso wie unterschiedliche Auffassungen über Rückgabewerte einiger Funktionen, etwa dem Verhalten beim Lesen von einer gebrochenen Pipe.
Dieser Nachteil konnte teilweise durch die Benutzung des GNU **autoconf** Packetes behoben werden. Dieses dient dazu, ein Softwarepacket bei der Installation mit den Informationen zu versorgen, die sich auf verschiedenen Maschinen unterscheiden, etwa der Lage bestimmter Headerdateien (siehe [Mac95]).
Eine besondere Hilfe bei der Programmierung war [Her96], alle Einzelheiten und subtilen Differenzen in dem Verhalten der beiden UNIX Systeme fanden hier ihre Erklärung.

Alles in allem kann man sagen, daß die Realisierung auf drei verschiedenen Systemen keineswegs drei mal soviel Arbeit nach sich zog, sondern im Gegenteil den Gesamtprozeß vielleicht noch etwas beschleunigt hat.

Teil III

Ergebnisse

Hiermit schließt denn das Leben in Quiché. Es gibt nichts mehr zu sehen. Die alte Weisheit der Könige ist dahin. So ist nun alles zu Ende in Quiché, Santa Cruz genannt.

Popol Vuh, das Buch des Rates der Quiché Maya

Kapitel 11

Anwendbarkeit des Verfahrens

Bei der Bewertung muß man einige Punkte berücksichtigen:

- Effizienz des Verfahrens
- Implementierung von Strategien
- Komplexität

11.1 Effizienz

Die Verfahrensweise der Übersetzung zu einer abstrakten Maschine hat sich als durchaus praktikabel erwiesen. Obwohl noch keine Optimierungen an dem erzeugten $\mathcal{QAM}$-Code vorgenommen wurden, ist die resultierende Geschwindigkeit beim Lösen eines Goals durchaus akzeptabel und eröffnet gute Perspektiven für zukünftige Entwicklungen.
Hält man sich vor Augen, daß es eine ganze Reihe von Optimierungstechniken, sowohl was den $\mathcal{QAM}$-Code angeht, als auch, was die Optimierung auf der $\mathcal{QIR}$ betrifft, noch anwendbar sind, dann kann man diese Implementierung durchaus als gelungen bezeichnen. Dabei ist noch nicht berücksichtigt, daß die Implementierung selbst sicher noch stark optimiert werden könnte.
Auch der Vergleich mit Ergebnissen aus dem **PaNaMa** System ist durchaus positiv für Quetzal.

11.2 Implementierung von Strategien

Die in das System gesetzten Hoffnungen wurden voll und ganz erfüllt. Die gesamten in Abschnitt 6 dargestellten Strategien wurden innerhalb eines einzigen Tages realisiert. Im Vergleich dazu dauerte die Implementierung dieser Strategien in **PaNaMa** annähernd zwei Monate.

Die Lesbarkeit der Startegiedefinitionen ist ebenfalls gut und nah an den ursprünglichen Definitionen der Strategien. Mit Quetzal steht nun erstmals ein Werkzeug zur Verfügung, mit dem neue Strategien schnell und einfach getestet werden können.
Inwieweit allerdings der Gesichtspunkt der Universalität mit Zustimmung betrachtet werden kann, muß sich erst noch in der weiteren Verwendung des Systems zeigen. Naturgemäß kann dies mit der erfolgreichen Demonstration einiger weniger Strategien nicht beantwortet werden.

11.3 Komplexität

Die Komplexität des Systems wurde durch die vorangehende, genaue Definition der Semantik entschärft. Die pedantische Vorgehensweise erlaubt auch prinzipiell Beweise für Eigenschaften des Systems, was in dieser Form nur selten anzutreffen ist.
Ein weiterer Vorteil ist, daß diese genaue Semantik die Implementation 'geführt' hat. Diese ist dadurch schnell und relativ problemlos gelungen. Abgesehen von den üblichen Fehlern, die durch die Implementierung in **C** hervorgerufen wurden, gelang die Realisierung des Projektes ohne Hindernisse. Der Lohn der Implementierung in **C** ist ein effizientes System, was im nachhinein die Entscheidung für diese Sprache rechtfertigt.
Alles in allem hat sich gezeigt, daß die sorgfältige Definition den Programmierzyklus extrem verkürzt hat, wie man im Hinblick auf Erfahrungen in der Realisierung des **PaNaMa** Systems sagen kann. Als Beispiel sei die Übertragung des fixen Codeanteils Fix$\mathcal{QAM}$ für die Compilierung genannt. Da dieser bereits als LaTeX-Source vorlag, da vorher definiert, konnte der **C**-Code einfach durch die Umdefinition einiger LaTeX-Makros und Bearbeitung des Abschnittes durch LaTeX erzeugt werden.

11.4 Vergleich mit PaNaMa

Um eine ungefähre Vorstellung davon zu bekommen, wie effizient die Implementierung ist, wurden die Ergebnisse von Probeläufen auf einigen wenigen Beispielen mit Ergebnissen des **PaNaMa** Systems verglichen.
Dabei wurden folgende Definitionen benutzt:

```
sorts
nat;
.

cons
z: nat;
s: nat -> nat;
.

funcs
```

```
add: nat nat -> nat;
.

rules
r1: add(z,X) -> X;
r2: add(s(X), Y) -> s(add(X,Y));
.

trs
t1: r1 r2;
.

tactics
ta1: r1>r2;
.

goals
g1: s(z)=?s(z);
g2: add(X,Y)=?s(s(z));
.

sorts
foo;
.

cons
a:foo;
b: foo -> foo;
c: foo->foo;

.

funcs
f: foo -> foo;
g: foo foo -> foo;
.

rules
r10: f(a) -> a;
r11: f(b(X)) -> b(f(X));
r12: f(c(X)) -> a;
r13: g(a,X) -> b(a);
r14: g(b(X),a) -> a;
```

```
r15: g(b(X),b(Y)) -> c(a);
r16: g(b(X),c(Y)) -> b(a);
r17: g(c(X),Y) -> b(a);
.

trs
t2: r10 r11 r12 r13 r14 r15 r16 r17;
.

tactics
ta2: r10>r11>r12>r13>r14>r15>r16>r17;
.

goals
g10: g(X,f(X))=?c(a);
.
```

Das TRS `t1` enthält dabei die Regeln für die Addition auf natürlichen Zahlen. Die Regeln `t2` sind [AEH94] entnommen.

Getestet wurde mit einer Suchtiefe von 10 Stufen, wobei dazu zu bemerken ist, daß **PaNaMa** und Quetzal den Begriff der Suchtiefe etwas unterschiedlich auslegen. Quetzal erhöht die Suchtiefe mit jeder erfolgreich ausgeführten Metaregel, während in **PaNaMa** eine neue Suchebene nur dann begonnen wird, wenn ein Narrowingschritt erfolgreich war, oder ein Mutateschritt gemacht werden konnte.

Trotzdem ist bei der gewählten kleinen Suchtiefe der Unterschied nicht allzu groß.

Das Goal `g1` ist einfach eine simple Unifikation. `g2` steht für die Frage $X + Y =_? 2$ und hat 3 Lösungen.

Das Goal `g10` wurde in [AEH94] verwandt, um die Vorteile der Needed-Narrowing Strategie gegenüber anderen Strategien zu unterstreichen und ist dem zufolge etwas künstlich.

Für Quetzal wurde ein Heap von 400000 Bytes verwandt, die Resultate beziehen sich auf die Implementierung auf einer SPARC10 Station.

Die angegebenen Zeiten sollten nur als Richtwerte benutzt werden, da naturgemäß die von einem Prozeß benötigte Zeit auf einem Unix System großen dynamischen Schwankungen unterliegt.

11.4.1 Resultate

Zuerst die Resultate von Quetzal:

Bsp.	Strategie	Lsg.	Dead	Bound	Heap	Stack	GCs	Kntn.	AM	Zeit
Han	RR	1	10	0	52926	434	1	26	1317483	2.66
	lrbasic	25	4	6	1771056	735	41	90	56881779	106.59
	SM	1	10	0	112437	343	2	31	3069951	5.84
	outermost	0	2	0	8424	125	0	2	267604	0.51
g_1	RR	1	3	0	20252	490	0	10	54216	1.01
	lrbasic	1	0	0	436	85	0	1	10848	0.02
	SM	1	0	0	3615	226	0	2	94762	0.18
	outermost	1	0	0	436	85	0	1	10848	0.03
g_2	RR	3	0	4	41883	690	0	34	1150013	2.25
	lrbasic	3	7	2	347321	2184	8	25	10796001	20.67
	SM	2	0	2	16320	471	0	22	444160	0.9
	outermost	2	0	0	8227	277	0	5	203895	0.41

PaNaMa:

Bsp.	Strategie	Lsg.	Dead	Bound	Kntn.	Zeit
Han	RR	1	3	0	15	0.02
	lrbasic	25	4	6	59	1.48
	SM	1	6	0	8	0.02
	outermost	0	2	0	2	0.01
g_1	RR	1	0	0	2	0.01
	lrbasic	1	0	0	1	0.02
	SM	1	0	0	1	0.02
	outermost	1	0	0	1	0.02
g_2	RR	3	0	0	26	0.02
	lrbasic	3	1	0	9	0.03
	SM	3	1	0	6	0.02
	outermost	3	1	0	9	0.03

Hierbei bedeutet 'Lsg.' die Anzahl gefundener Lösungen, 'Dead' die Anzahl von Sackgassen, 'Bound' die von Tiefenüberschreitungen, 'Kntn.' die Anzahl gerechneter Knoten im Suchbaum, 'Heap' die Anzahl alloziierter Zellen auf dem Heap, 'Stack' die maximale Anzahl benutzter Stackzellen, 'GCs' die Anzahl aufgetretender Garbage Collections und 'AM' die Anzahl ausgeführter $\mathcal{QAM}$-Befehle.

11.4.2 Diskussion

Zuerst fällt auf, daß die Anzahl von Sackgaßen und Tiefenüberschreitungen sich zwischen beiden Systemen unterscheidet. Dies liegt daran, daß in **PaNaMa** einige Optimierungen eingebaut sind, die Sackgaßen vorher erkennen können und auch die Suchtiefe anderes interpretiert wird (s.o.). Dies erklärt auch die unterschiedlich Anzahl gerechneter Knoten. I.A. ist das **PaNaMa** System um einen Faktor 100 schneller, was das Auffinden von Lösungen angeht. Allerdings implementiert es auch eine ganze Reihe von Optimierungen, die in Quetzal mit etwas Mühe ebenfalls noch eingebaut werden könnten.

Trotzdem zeigen diese Ergebnisse, daß Quetzal noch einigermaßen effizient ist, insbesondere da hier noch einige Ebenen der Optimierung offenstehen: Optimierung auf der Ebene der Strategiedefinitionen, auf der Ebene der $\mathcal{QAM}$ und der Übersetzung und natürlich auf der Ebene der Implementierung selber.

11.5 Deffizienzen

Als Nachteile bleiben anzumerken, daß einerseits aus Zeitmangel einige Beweise und Implementierungen verschiedener Teile nicht mehr erfolgen konnte. Hier wurde der Realisierung der wichtigen Teile des Systems der Vorzug gegeben.
Besonders wären hier der Terminierungs- und der Confluencecheck zu nennen, die bereits einmal in **PaNaMa** implementiert wurden und deshalb zu Gunsten der neuen Teile des Systems übergangen wurden.
Im übrigen kann man sich manche Verbesserungen in der Benutzerfreundlichkeit des Systems vorstellen, etwa ein Window Frontend, bessere und genauere Fehlermeldungen, sowie die Verwendung von effizienteren Algorithmen für einige Teile, etwa den Typcheck.
Man beachte aber, daß diese Verbesserungen lediglich simple Programmierarbeit darstellen und keine Auswirkungen auf grundlegende Konzepte und Entwürfe des Systems haben. Die Kernfunktionalität des Systems ist komplett implementiert und stabil.

Kapitel 12

Zusammenfassung und Ausblick

Es wurde ein Beschreibungssystem für semantische Unifikationsverfahren implementiert. Das System wurde genau definiert und diese Definition auch realisiert.
Ausgehend von einer genauen Definition der Semantik des Systems in Form mehrerer Natural Semantics wurde einer abstrakte Maschine entworfen, ihre Semantik definiert und die Übersetzung von Programmen zu Code für diese Maschine angegeben.
Anhand dieser Definitionen wurde das System auf mehreren Architekturen implementiert und die Realisierbarkeit des Entwurfs gezeigt.
Das Ergebnis entspricht den Anforderungen und zeigt die Vorteile einer solchen Vorgehensweise.
Das System bietet trotzdem noch Potential für Erweiterungen und Verbesserungen. Neben den aus Zeitgründen weggelassenen Komponenten kann man auch noch Erweiterungen, z.B. des Funktionskonzeptes vornehmen, um etwa curryfizierte Funktionen definieren zu können. Dies macht nur wenige Änderungen an der $\mathcal{QAM}$ nötig, erfordert jedoch einige Änderungen am Parser, StSemCheck und dem Compiler.
Die Einführung neuer primitiver Befehle in die $\mathcal{QAM}$, etwa zur Unifikation von Termen, wäre ein möglicher nächster Schritt.
Etwas unbefriedigend ist immer noch ein Vorhandensein gewisser Maschinenabhängigkeiten in der $\mathcal{QAM}$, die man ebenfalls entfernen könnte. Als wichtigster Punkt bleibt natürlich die Anwendung vielfälltigster Optimierungen auf die Compilierung und die $\mathcal{QAM}$ an sich. Interessant wäre der Gedanke, eine parallele Implementierung vorzunehmen, die auf Architekturen, die dies unterstützen, eine weitere Geschwindigkeitssteigerung erlauben.
Eine Erweiterung in Richtung auf eine breadth first Suche, statt der in der Semantik festgelegten depth first Suche würde auch Änderungen an der Semantik des Systems an sich verlangen, die allein zu beschreiben schon interessant wären.
Eine Erweiterung des grundlegenden Konzeptes bezüglich der Kombination verschiedener Strategien ist es ebenfalls wert, bedacht zu werden. Auch im Hinblick auf die wirkliche Anwendung einer logisch-funktionalen Sprache wird immer klarer, daß *eine* Strategie *allein* kaum als Grundlage dieser Sprache dienen kann, sondern eine Kombination von Strategien für verschiedene Goaltypen notwendig ist, wenn man eine effiziente, universelle Sprache haben will.

Literaturverzeichnis

[AA94] Zena Ariola und Arvind. Properties of first-order functional language with sharing. In: R. Sleep, R. Plasmeijer, und M. van Eekelen, Hrsg., *Term Graph Rewriting*. John Wiley Publishers, June 1994
`ftp://csg-ftp.lcs.mit.edu/pub/papers/csgmemo/memo-347-1.ps.gz`.

[AEH94] S. Antoy, R. Echahed, und M. Hanus. A Needed Narrowing Strategy. In: *Proceedings of the 21st ACM Symposium on Principles of Programming Languages*, Seiten 268–278. ACM Press, 1994
`http://www-i2.informatik.rwth-aachen.de/~hanus/publications/papers/POPL94.dvi.Z`.

[Ave95] Jürgen Avenhaus. *Reduktionssysteme*. Springer, 1995.

[BBS92] Steffen van Bakel, Simon Brock, und Sjaak Smetsers. Partial Type Assignment in Left-Linear Applicative Term Rewriting Systems. In: Raoult, Hrsg., *Proceedings of the 17th Colloquium on Trees in Algebra and Programming (CAAP'92)*, Seiten 300–321. Springer LNCS 581, 1992
`ftp://ftp.cs.kun.nl/pub/CSI/SoftwEng.FunctLang/papers/baks92-typingtrs.ps.gz`.

[BEG+87] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, und M.R. Sleep. Term Graph Rewriting. In: Bakker, Nijman, und Treleaven, Hrsg., *Proceedings of Parallel Architectures and Languages Europe (PARLE '87)*, Seiten 141–158. Springer LNCS 259, 1987
`ftp://ftp.cs.kun.nl/pub/CSI/SoftwEng.FunctLang/papers/barh87-TGR.ps.gz`.

[BGM88] P.G. Bosco, E. Giovannetti, und C. Moiso. Narrowing vs. SLD-Resolution. *Theoretical Computer Science*, **59**:Seiten 3–23, 1988.

[BGOT95] B. Bütow, R. Giegerich, E. Ohlebusch, und S. Thesing. A New Strategy for Semantic Matching. In: *Proceedings of the 7th International Symposium on Programmimg Languages, Implementations, Logics and Programs*, LNCS. Springer Verlag, 1995
`ftp://ftp.uni-bielefeld.de/pub/papers/techfak/pi/PLILP95_paper.ps.gz`.

[BKW92] A. Bockmayr, S. Krischer, und A. Werner. An Optimal Narrowing Strategy for General Canonical Systems. In: *proc. of the 3rd Intern. Workshop on Conditional Term Rewriting Systems*, Seiten 483–497. Springer LNCS 656, 1992.

[BS93] Erik Barendsen und Sjaak Smetsers. Conventional and Uniqueness Typing in Graph Rewrite Systems. Technischer Report CSI-R9328, Computing Science Institute, University of Nijmegen, December 1993 `ftp://ftp.cs.kun.nl/pub/CSI/SoftwEng.FunctLang/papers/bare93-typinggrs.ps.gz`.

[BT94] B. Bütow und S. Thesing. Reverse Restrucuring: Another Method of Solving Algebraic Equations. Forschungsberichte der Technischen Fakultät, Abteilung Informationstechnik, Report 94-07, Universität Bielefeld, 1994 `ftp://ftp.uni-bielefeld.de/pub/papers/techfak/pi/Report94-07.ps.gz`.

[BT95] B. Bütow und S. Thesing. PaNaMa user's manual. Forschungsberichte der Technischen Fakultät, Abteilung Informationstechnik, Report 95-01, Universität Bielefeld, 1995.

[D9̈5] Heiko Dörr. *Efficient Graph Rewritung and Its Implementation.* Springer, LNCS 922, 1995.

[Dau92] M. Dauchet. Simulation of Turing Machines by a Regular Rewrite Rule. *Theoretical Computer Science*, **103**(2):Seiten 409–420, 1992.

[DG89] J. Darlington und Y. Guo. Narrowing and Unification in Functional Programming - an Evaluation Mechanism for Absolute Set Abstraction. In: *Proc. of the Conference on Rewriting Techniques and Applications*, Seiten 92–108. Springer LNCS 355, 1989.

[DJ90] N. Dershowitz und J.-P. Jouannaud. Rewrite Systems. In: J. van Leuwen, Hrsg., *Handbook of Theoretical Computer Science*, Band B, Seiten 243–320. Elsevier Science Publisher B.V., 1990.

[DMR76] Martin Davis, Yuri Matijasevič, und Julia Robinson. Hilbert's tenth problem. Diophantine equations: positive aspects of a negative solution. In: Felix E. Browder, Hrsg., *Proceedings of the Symposium in Pure Mathematics*, Seiten 323–378. American Mathematical Society, May 1976.

[DMS92] N. Dershowitz, S. Mitra, und G. Sivakumar. Decidable Matching for Convergent Systems. In: *Proceedings of the 11th Conference on automatic deduction (CADE)*, Seiten 133–146, 1992.

[Ech90] R. Echahed. On Completeness of Narrowing Strategies. *Theoretical Computer Science*, **72**:Seiten 133–146, 1990.

[Fay79] M.J. Fay. First-Order Unification in an Equational Theory. In: *Proc. 4th Workshop on Automated Deduction*, Seiten 161–167. Academic Press, 1979.

[Fri85] L. Fribourg. SLOG: A Logic Programming Language, Interpreter Based on Clausal Superposition and Rewriting. In: *Proc. of the 1985 Symposium on Logic Programming*, Seiten 172–184. IEEE Computer Soc. Press, 1985.

[Han94] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, **19&20**:Seiten 583–628, 1994
`http://www-i2.informatik.rwth-aachen.de/~hanus/publications/papers/JLP94.dvi.Z`.

[Her86] A. Herold. Narrowing Techniques Applied to Idempotent Unification. Technischer report, Univ. Kaiserslautern, 1986.

[Her96] Helmut Herold. *UNIX-Systemprogrammierung*. Addison-Wesley, 1996.

[HI73] Ken Hirose und Shigeaki Iida. A proof of negative answer to Hilbert's 10th Problem. *Proceedings of the Japan Academy*, **49**(1):Seiten 10–12, 1973.

[Hil00] David Hilbert. Mathematische Probleme. Vortrag, gehalten auf dem internationalen Mathematiker-Kongreß zu Paris 1900. *Nachrichten von der Königlichen Gesellschaft der Wissenschaften zu Göttingen*, Seiten 253–297, 1900.

[Hil02] David Hilbert. Mathematical problems. Lecture delivered before the International Congress of Mathematicians at Paris in 1900. *Bulletin of the American Mathematical Society*, **8**:Seiten 437–479, 1902.

[Höl89] S. Hölldobler. *Foundations of Equational Logic Programming*. Springer, LNCS 353, 1989.

[Hul80] J.-M. Hullot. Canonical Forms and Unification. In: *Proc. 5th Conference on Automated Deduction*, Seiten 318–334. Springer, LNCS 87, 1980.

[JM91] J. P. Jones und Y. V. Matijasevič. Proof of recursive unsolvability of Hilbert's Tenth Problem. *The American Mathematical Monthly*, **98**(8):Seiten 689–709, Oktober 1991.

[Kah87] G. Kahn. Natural semantics. In: *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, Seiten 22–39. Springer, LNCS 247, 1987.

[KKSV94] J. R. Kennaway, J. W. Klop, M. R. Sleep, und F. J. de Vries. On the adequacy of graph rewriting for simulating term rewriting. *ACM TOPLAS*, **16**(3):Seiten 493–523, May 1994
`ftp://ftp.sys.uea.ac.uk/pub/kennaway/Adequacy.ps.Z`.

[Klo92] J.W. Klop. Term Rewriting Systems. In: S. Abramsky, D. Gabbay, und T. Maibaum, Hrsg., *Handbook of Logic in Computer Science*, Band 2, Seiten 1–116. Oxford University Press, 1992.

[Mac95] David MacKenzie. *Autoconf. Creating Automatic Configuration Scripts.* FSF- Free Software Foundation, Inc., 2.7 Auflage, November 1995.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, **17**:Seiten 348–375, 1978.

[MM82] A. Martelli und U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **4**(2):Seiten 258–282, 1982.

[MNKLRA90] J.J. Moreno-Navarro, H. Kuchen, R. Loogen, und M. Rodríguez-Artalejo. Lazy Narrowing in a Graph Machine. In: *Proc. Second International Conference on Algebraic and Logic Programming*, Seiten 298–317. Springer LNCS 463, 1990 `file://ftp.informatik.rwth-aachen.de/pub/reports/1990/90-11.ps.gz`.

[MT90] Robin Milner und Mads Tofte. *Commentary on Standard ML.* MIT Press, 1990.

[MTH90] Robin Milner, Mads Tofte, und Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[NRS89] W. Nutt, P. Réty, und G. Smolka. Basic Narrowing Revisited. *Journal of Symbolic Computation*, **7**:Seiten 295–317, 1989.

[Ohl94] Enno Ohlebusch. *Modular Properties of Composable Term Rewriting Systems.* Dissertation, University of Bielefeld, 1994. Forschungsberichte der Technischen Fakultät, Abteilung Informationstechnik, Report 94-01.

[PJ86] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* C.A.R. Hoare editor. Prentice-Hall, 1986.

[PJL92] Simon L. Peyton Jones und David Lester. *Implementing Functional Languages. A Tutorial.* C.A.R. Hoare editor. Prentice-Hall, 1992.

[Sch94] David A. Schmidt. *The Structure of Typed Programming Languages.* Foundations of Computing Series. MIT Press, 1994.

[Wil92] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In: *Proceedings of the 1992 International Workshop on Memory Management*, St. Malo, France, September 1992. Springer.

[WM92] Reinhard Wilhelm und Dieter Maurer. *Übersetzerbau. Theorie, Konstruktion, Generierung.* Springer Verlag, 1992.

[WZ95] H.R. Walters und H. Zantema. Rewrite Systems for Integer Arithmetic. In: Jie Hsiang, Hrsg., *Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, Seiten 324–338. Springer, LNCS 914, 1995.

[You89] Jia-Huai You. Enumerating Outer Narrowing Derivations for Constructor–Based Term Rewriting Systems. *Journal of Symbolic Computation*, **7**:Seiten 319–341, 1989.