Institut für Informatik
der Ludwig-Maximilians-Universität München
Oettingenstr. 67
D-80538 München

Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
(Prof. Dr. François Bry)

Verarbeitung und Darstellung indefiniter Daten in einer applikativen Sprache

Matthias Nickles

Diplomarbeit

Bearbeitungszeitraum: 03.11.1997 - 02.05.1998

Aufgabensteller: Prof. Dr. François Bry

Betreuer: Dr. rer. nat. Heribert Schütz

Erklärung

Hiermit versichere ich, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

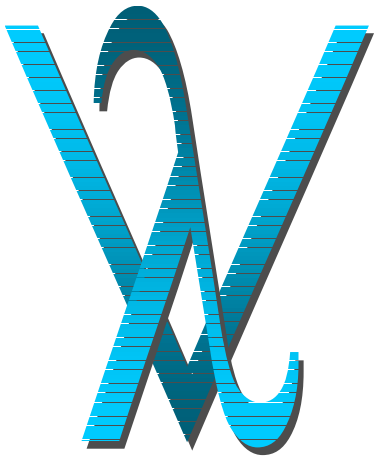
München, den 02.05.98 Matthias L. Nickles

Vorbemerkungen

Zu dieser Diplomarbeit gehört eine Implementation der im Rahmen dieser Arbeit entwickelten Programmiersprache „Prolid“ und der graphischen Darstellungssoftware für Prolid („Interactor“). Da der Quellcode der Software sehr umfangreich ist, ist er nur elektronisch unter folgender URL verfügbar:

`http://www.pms.informatik.uni-muenchen.de/publikationen`
`/diplomarbeiten/Matthias.Nickles`

Anschrift des Autors:
Matthias Nickles, Milbertshofener Str. 71, D-80807 München, Tel. 089/ 40 60 01
EMail: `nicklesm@usa.net`



Inhalt

1 Einleitung……..…….…….3

2 Or-Sets und indefinite Programme…….. 7

2.1 Eine elementare definite Sprache……..7
2.1.1 Abstrakte Syntax……..…7
2.1.2 Deklarative Semantik……. 9

2.2 Allgemeine Or-Sets……...18

2.3 Or-Set-Generatoren……..…….19
2.3.1 Or-Ausdrücke……. 20
2.3.2 Das leere Or-Set……..30
2.3.3 Query-Variablen……. 31

3 Baum-Grammatiken und Zuordnungslisten……..... 33

3.1 Reguläre Baum-Grammatiken……... 33

3.2 Reguläre Funktionen und Zuordnungslisten…….. 35

3.3 Generierung von Zuordnungslisten……...…37

4 Die Programmiersprache Prolid…….. 42

4.1 Layout…….. 44

4.2 Lexikalische Struktur…… 44
4.2.1 Bezeichner und Operatoren…… 45
4.2.2 Zahlen, Zeichen und Strings……...46

4.3 Datentypen…… 47
4.3.1 Typausdrücke……..48
4.3.2 Typ-Deklaration……. 49
4.3.3 Typ-Synonyme……... 50

4.4 Ausdrücke…….. ……...51
4.4.1 Variablen, Konstruktoren und Operatoren…….. ……...52
4.4.2 Applikationen und λ-Abstraktionen…….. 53
4.4.3 Or-Set-Generatoren und Or-Set-Distribution……. 55
4.4.4 Muster, Pattern-Matching und Constraint-Generierung…….60
4.4.5 Bedingte Ausdrücke……65

4.5 Variablen-Deklaration……...71
4.5.1 Top-level-Bindungen…….. 71
4.5.2 Lokale Deklarationen……...73
4.5.3 Typ-Signaturen und Ad-hoc-Polymorphismus……...75

4.6 Programme……77
4.6.1 Module……78
4.6.2 Fixity-Deklarationen……...79

4.7 Auswertung…….. 80
4.7.1 Repräsentationsformen der Kernsprache……80
4.7.2 Operationelle Semantik…….. 83

5 Graphische Darstellung von Resultaten mit dem Interactor……93

5.1 Funktionale Generierung graphischer Komponenten……... 94
5.1.1 Darstellung algebraischer Datentypen……96
5.1.2 Automatisch generierte Darstellungen……... 100

5.2 Darstellung von Or-Sets……... 102

5.3 Darstellung von Zuordnungslisten……... 103

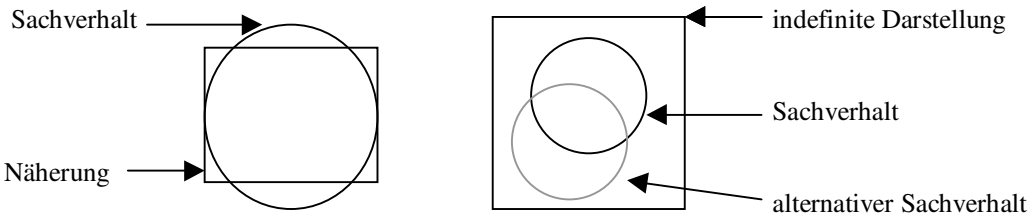
6 Zusammenfassung und Ausblick……... 111

Anhang A: Referenz zum Interpreter…….. 112
Anhang B: Referenz zum Interactor…… 119
Anhang C: Optimierte Rekursionserkennung……...122

Literatur ……... 125

1 Einleitung

Ein Datum, das durch die darüber vorliegenden Informationen nicht vollständig bestimmt wird, heißt *indefinit*. Die Gründe für das Auftreten unvollständiger Informationen bei der maschinellen Datenverarbeitung sind vielfältig: Zum einen lassen sich bestimmte natürliche oder mathematische Objekte prinzipiell nicht exakt beschreiben (z.B. gewisse physikalische Zustände auf Quantenebene). Zum anderen ist man oft nicht in der Lage, über einen realen Zustand alle Informationen in wünschenswerter Vollständigkeit oder Genauigkeit zusammenzutragen (z.B. bei physikalischen Messungen oder bei der Erstellung der Datenbasis für eine Personen-Datenbank). Eine Möglichkeit, solche Objekte oder Zustände zu verarbeiten, besteht in der Verwendung von *Näherungen*. Beispielsweise nähert man die numerische Darstellung reeller Zahlen oft durch rationale Zahlen an. Bezeichnend für Näherungen ist jedoch, daß hier Information über das angenäherte Objekt verfälscht werden oder verlorengehen. Wir sprechen dann von einer *unvollständigen* Repräsentation. Im Interesse der Exaktheit, und weil es oft nicht möglich ist, geeignete Näherungen zu finden[1], bevorzugen wir jedoch *vollständige* Repräsentationen, d.h. Darstellungen, die den wahren Sachverhalt umschreiben, ohne ihn zu begrenzen oder zu verfälschen. Solche *indefiniten* Darstellungen beschreiben das repräsentierte Objekt zwar korrekt, bestimmten es aber nicht genau. Die Zahl π können wir z.B. durch den definiten Wert 3,14 annähern, sie aber auch durch einen indefiniten Wert in Gestalt des Intervalls [3,1..3,2] eingrenzen. Der wahre definite Wert π ist in dem Intervall enthalten, ist durch das Intervall aber nicht eindeutig identifizierbar, da es alternative Zahlen gibt, denen man denselben indefiniten Wert zuordnen kann.



Eine Möglichkeit für eine Darstellung eines indefiniten Datums besteht in der *Abstraktion* von gewissen speziellen Eigenschaften. Beispielsweise können wir einen 1,86m großen Menschen einfach als „groß“ bezeichnen. Es ist jedoch bekannt, daß die Zuordnung von Abstraktionen zu exakten Daten nicht immer zweifelsfrei möglich ist. Mit der mit einer solchen Kategorisierung ggfs. verbundenen *Unschärfe* befaßt sich z.B. die *Fuzzy-Logik*. Wir können auch versuchen, den

[1] Z.B. gibt es keine sinnvolle Näherung für einen Eigennamen in einer Personen-Datenbank

indefiniten Wert durch die Angabe seiner bekannten Eigenschaften zu *umschreiben*. Hierfür eignet sich die Prädikatenlogik. Die Zahl π ließe sich z.B. mittels $\pi \geq 3{,}1 \wedge \pi \leq 3{,}2$ umschreiben, was obiger Intervall-Darstellung entspricht.

Ein anderer Weg ist die Angabe aller alternativen definiten Werte, die den „wahren Wert" des zu beschreibenden Datums ausmachen können. Solche Mengen von exklusiven, definiten *Alternativen* bezeichnen wir allgemein als *Or-Sets*[2], mit denen wir uns in dieser Arbeit hauptsächlich befassen werden. Or-Sets und ähnliche Strukturen sind aus der Stochastik bekannt. Eine theoretisch gut fundierte Theorie, die auf Or-Sets basiert (die dort *frames of discernment* heißen), ist z.B. die *Dempster-Shafer-Theorie* (vgl. [6], S. 268). Diese Theorie reichert die Or-Sets zusätzlich mit Werten an, die den Grad der Wahrscheinlichkeit angeben, daß eine bestimmte Alternative gleich dem wahren Wert des indefiniten Datums ist. Mit solchen Aspekten der *inneren Struktur* von Or-Sets werden wir uns allerdings in dieser Arbeit nicht befassen.

In dieser Arbeit wollen wir eine funktionale Programmiersprache entwerfen, die auf Or-Sets operieren kann und diese Möglichkeit mit den wesentlichen Eigenschaften moderner funktionaler Programmiersprachen wie z.B. *Haskell* nahtlos verknüpft[3]. Dadurch, daß wir eine Programmiersprache um Möglichkeiten zur Verarbeitung indefiniter Daten anreichern, erlauben wir diese Fähigkeiten im Prinzip auch automatisch für alle Applikationen, die in dieser Sprache implementiert werden. Für eine applikative Sprache entscheiden wir uns, weil diese Sprachklasse mathematisch relativ leicht und klar faßbar ist (im Gegensatz zu gewissen prozeduralen Sprachen) und den operationellen Aspekt der Verarbeitung, d.h. den Fortschritt der Auswertung von Ausgangsdaten zu einem Resultat, vergleichsweise explizit realisiert (im Gegensatz z.B. zu *Prolog*).

Or-Sets eignen sich gut zur *Einbindung* in eine Programmiersprache, falls man ein Or-Set, dessen Alternativen dann die herkömmlichen First-class-Objekte der Programmiersprache sind, konzeptionell ähnlich behandeln kann wie einen einzelnen definiten Wert. Wir brauchen dazu im Prinzip lediglich einen universellen Mechanismus, der die internen Auswertungsoperationen der Sprache, die sich bisher auf definite Werte bezogen haben, nun auf die einzelnen Alternativen indefiniter Werte ausdehnt. Da wir nicht wissen, welche der Alternativen der „wahre Wert" des indefiniten Datums ist, muß dieser Mechanismus nach dem Prinzip des *Don't-know-Nichtdeterminismus* auf *alle* Alternativen operieren, um *ein* korrekte Resultat zu erhalten.

Ein anderer, vordergründig einfacher Weg, indefinite Daten in einer funktionalen Sprache zu verarbeiten, bestünde darin, mit den Mitteln der Sprache neue Datentypen zur Repräsentation indefiniter Werte einzuführen (dies geschieht z.B. in *OR-SML* [5], einer Erweiterung von Standard ML [10]) oder vorhandene Datentypen zu „mißbrauchen", d.h. als indefinit anzusehen

---

[2] Später werden wir diesen Begriff genauer fassen

[3] Wie z.B. Laziness, statisches Typechecking, Higher-order-Funktionen und parametrisierte algebraische Datentypen

(z.B. Or-Sets in Form von *Listen* darzustellen). Hier benötigt man neue, spezielle Funktionen, die auf diese Datenstrukturen in der geeigneten Weise operieren. Wir wollen dem Programmierer diese Arbeit abnehmen. Wir erlauben es ihm, jederzeit ohne besondere Vorkehrungen mit indefiniten wie definiten Daten arbeiten zu können und ggfs. indefinite Resultate zu erhalten, ohne daß er sich um anwendungstechnische Unterschiede zwischen „definiten“ und „indefiniten“ Sprachelementen kümmern muß. Dazu soll die Verwendungsmöglichkeit aller bisher in der definiten Sprache enthaltenen Sprachelemente in sinnvoller Weise auf indefinite Daten *ausgedehnt* werden (und für definite Daten unverändert erhalten bleiben). Dabei wäre es unnötig und sogar ungünstig, für indefinite Daten (dies sind für uns Or-Sets) einen oder mehrere *neue* Datentypen einzuführen, d.h. indefinite Werte als First-class-Werte eines speziellen „Or-Set-Typs“ zuzulassen. Zum einen wäre die daraus folgende Unterscheidung von definiten und indefiniten Werte aus Sicht des Programmierers wenig intuitiv, da ein indefiniter Wert schließlich genau einen definiten Wert repräsentiert, zum zweiten bekäme man Schwierigkeiten mit dem Typschema bei der vermischten Verwendung definiter und indefiniter Werte, und man müßte Funktionen für die Verwendung mit Werten dieser neuen Typen überladen. Es ist daher sinnvoller, den Unterschied zwischen definiten und indefiniten Werten aus der Sicht des Programmierers völlig transparent erscheinen zu lassen. Überall dort, wo ein definiter Wert eines Typs T eingesetzt werden darf, soll auch ein indefiniter Wert stehen dürfen, der einen unbekannten Wert vom Typ T beschreibt und der in jeder denkbaren Hinsicht wie ein Wert vom Typ T behandelt wird. Der Nachteil unserer Herangehensweise ist allerdings, daß in diesem Konzept keine „Meta-Funktionen“ auf Or-Sets vorgesehen sind, d.h. Anwender-Funktionen, die auf ein Or-Set als ganzes operieren. Dadurch sind Fragestellungen wie z.B. „ist der Wert x definit?“ nicht möglich.

Um zu zeigen, wie man Or-Sets einfach und nahtlos in die bestehenden Konzepte fortgeschrittener funktionaler Programmiersprachen integrieren kann, werden wir am Anfang von Kapitel 2 eine herkömmliche (d.h. „definite“) funktionale Programmiersprache definieren. Diese Sprache werden wir im weiteren Verlauf von Kapitel 2 um Elemente anreichern, die die Verarbeitung von Or-Sets erlauben. Eines dieser neuen Sprachelemente wird in Darstellung und Gebrauch den existenzquantifizierten *Variablen* funktional-logischer Programmiersprachen wie z.B. *Curry* [9] ähneln. In Kapitel 3 zeigen wir, daß sich bestimmte Or-Sets als formale *Sprachen* auffassen lassen, und zeigen Gemeinsamkeiten zwischen sogenannten *Baum-Grammatiken* und Programmen, die Or-Sets generieren, auf. Wir werden in diesem Zusammenhang einen Algorithmus vorstellen, der die Berechnungen gewisser unendlicher *Baum-Sprachen* terminieren läßt und die Resultate als Grammatik darstellt. Kapitel 4 beschreibt die Programmiersprache *Prolid*, die im Rahmen dieser Diplomarbeit entworfen und implementiert wurde, um die theoretischen Konzepte zu veranschaulichen. Kapitel 5 schließlich zeigt, wie sich indefinite Werte, wie sie von Prolid berechnet werden, auf objektorientierte Weise *graphisch* darstellen

lassen. Auch zu diesem Kapitel gibt es eine Implementation in Gestalt des Java-Programms *Interactor.*

2 Or-Sets und indefinite Programme

Im folgenden werden (funktionale) *indefinite Programme* eingeführt. Dies sind funktionale Programme, die mehr als ein Berechnungsergebnis repräsentieren können. Programme für herkömmliche funktionale Programmiersprachen wie *Haskell* oder *ML* heißen demgegenüber *definit*. Eine Menge von Resultaten stellen wir als Or-Sets dar. Um bei der Programmauswertung mehr als ein Resultat erzeugen zu können, benötigen wir sogenannte *Or-Set-Generatoren*. Or-Set-Generatoren dienen als syntaktische Bestandteile einer funktionalen Sprache zur Repräsentation indefiniter Informationen im Quellcode indefiniter Programme.

2.1 Eine elementare definite Sprache

Um die Semantik von Or-Sets und Or-Set-Generatoren im Rahmen einer funktionalen Programmiersprache beschreiben zu können, definieren wir im folgenden die definiten Elemente einer einfachen funktionalen Sprache (basierend auf dem typisierten λ-Kalkül), die als *Kernsprache* von Prolid fungiert. Danach wird diese Sprache um Elemente erweitert, die die Verarbeitung und Generierung von indefiniten Daten erlauben.
Diese Kernsprache kennt nur wenige Sprachelemente, u.a. aber Higher-order Funktionen, Pattern-Matching und nicht-strikte Auswertung. Sie wurde in Anlehnung an die Kernsprache von Haskell entworfen.
Die folgende Beschreibung geht nur auf die abstrakte Syntax und die dynamische Semantik der Sprache ein. Es wird angenommen, daß alle Programme der Kernsprache „typkorrekt“ sind[4]. Die Regeln für die definiten Sprachelemente stammen, leicht angepaßt, aus [8].

2.1.1 Abstrakte Syntax

Folgende Syntax beschreibt keine konkrete Programmiersprache, sondern Datenstrukturen (*syntaktische Konstrukte*) etwa in der Art, wie sie einem Interpreter für eine funktionale Sprache nach statischer Analyse und Transformationsphase vorliegen (die sogenannte *Kernsprache*). Zur Darstellung der abstrakten Grammatik der Kernsprache wird eine an den Haskell-Report [17]

---

[4] Die Regeln für die Typinferenz für Or-Set-Generatoren (deren sogenannte *statische Semantik*) sind mehr oder weniger trivial und werden in 4.6.3 beschrieben. Die statische Semantik der übrigen Sprach-Bestandteile wird „wie allgemein üblich“ festgelegt. Typinferenz-Regeln für die Haskell-Äquivalente der unter 2.1.1 aufgeführten Sprach-Elemente finden sich z.B. in [14] und [17].

angelehnte BNF-ähnliche Notation verwendet. Die „intuitive Bedeutung“ der einzelnen syntaktischen Konstrukte ist, wenn nötig, bei den zugehörigen Semantik-Regeln in 2.1.2 erklärt.

Programm	→	{ Top-level-Bindung$_1$, …, Top-level-Bindung$_n$ }	(n≥1)
		(darunter genau eine Top-level-Bindung für die Variable „main“)	
Bindung	→	Variable = Rumpf^n	(n≥0)
Rumpf^0	→	Ausdruck (Rumpf bei einer einfachen Variablenbindung)	
Rumpf^n	→	Zuordnung^n	(n≥1)
Ausdruck	→	Konstruktor[5]	
	|	Variable	
	|	Applikation (Ausdruck$_A$, Ausdruck$_B$)	
	|	Verzweigung (Ausdruck$_{Test}$, Muster, Ausdruck$_A$, Ausdruck$_B$)	
	|	⊥	
Muster	→	Variable	
	|	Konstruktor	
	|	Applikation(Applikation(Konstruktor, Variable$_1$), … Variable$_n$)	
		abgekürzt Konstruktor Variable$_1$ … Variable$_n$	(n≥1)
		(Jedes Muster sei *linear*, d.h. eine bestimmte Variable darf nur höchstens einmal vorkommen.)	
λ-Abstraktion	→	Zuordnung^n	(n≥1)
	|	KonstMatch[6]	
	|	FunZuordnung	
Zuordnung^n	→	λ (Variable$_1$,…,Variable$_n$) → Ausdruck	(n≥1)
KonstMatch	→	λ (Muster → Ausdruck ; _ → Ausdruck‘	
		(KonstMatch ist *nicht* Teil der Programm-Syntax, sondern wird als syntaktisches Konstrukt durch die semantische Regel für Verzweigung (2.1.2.7(9)) eingeführt.)	
FunZuordnung	→	λ (Variable$_1$) → … (λ (Variable$_n$) → Ausdruck)	(n≥1)
		(FunZuordnung ist *nicht* Teil der Programm-Syntax, sondern wird als syntaktisches Konstrukt durch die semantische Regel für Top-level-Bindungen (2.1.2.7(11)) eingeführt.)	

[5] Es gebe nur endlich viele Konstruktoren.

[6] Natürlich ist dies nicht „die“ bekannte λ-Abstraktion; der Einfachheit halber verwenden wir den Begriff „λ-Abstraktion „ aber auch für diese, von der Intention her sehr ähnliche Konstruktion (siehe Regeln 2.1.2.7(6)-(8)).

2.1.2 Deklarative Semantik

Folgende Semantik-Regeln definieren den *semantischen Wert* eines Programms. Vorher führen wir noch einige wichtige Begriffe ein, die auch in der übrigen Arbeit noch häufig Verwendung finden werden:

Notation 2.1.2.1: Ausdrücke der Form Applikation(Applikation(e_1, e_2), … e_n) [7] kürzen wir meist als e_1 e_2 … e_n ab, wobei wir ggfs. Klammerungen hinzufügen, um die Eindeutigkeit dieser Notation sicherzustellen. e_1 wird *Kopf* des Ausdrucks genannt, die $e_{2..n}$ die *Argumente*.

Definition 2.1.2.2: Eine *Konstruktor-Applikation* ist ein Ausdruck der Form k e_1 … e_n, wobei k ein Konstruktor ist und die e_i beliebige Ausdrücke sind. Obwohl wir nur endliche viele verschiedene Konstruktoren erlaubte, dürfen Konstruktor-Applikationen unendlich tief sein.

Definition 2.1.2.3: Ein *Term* ist ein Ausdruck, in dem nur Konstruktoren, Variablen, und Unterausdrücke der Form Applikation($term_1$, $term_2$) vorkommen. Ein *Grundterm* ist ein variablenfreier Term[8]. Terme können als *Bäume* betrachtet werden: Jeder Kopf ist ein Knoten und seine Argumente sind die Unterbäume. Grundterme bezeichnen wir als *Werte-Bäume* oder, wenn kein Mißverständnis möglich ist, kurz als „Bäume“. Mengen von Grundtermen Terme heißen entsprechend (*Werte-)Wälder*. Wir notieren Grundterme und Werte-Bäume in der Form K a_1 … a_n (K ist ein Konstruktor der Arität n, die a_i sind wieder Grundterme).

Definition 2.1.2.4: Eine *Substitutionsumgebung* (oder *Variablen-Belegung*) ist eine Funktion, die je einer Variablen je einen Ausdruck zuordnet. Eine einzelne derartige Zuordnung (genannt *Bindung* der jeweiligen Variable) schreiben wir als Paar (Variable, Ausdruck)[9].

Definition 2.1.2.5: Eine *Funktion* (eines Programms der Kernsprache) ist eine Variable, für die im Programm eine Top-level-Bindung existiert. Top-level-Bindungen für parameterlose Funktionen werden auch als *einfache Variablenbindungen* bezeichnet.

---

[7] Dies ist die bekannte *curried*-Notation, benannt nach dem Logiker Haskell B. Curry.

[8] Somit ist jeder Grundterm auch entweder ein Konstruktor oder eine Konstrruktor-Applikation.

[9] Wir verwenden auch den recht allgemeinen Begriff *Umgebung*, wobei damit einfach der Kontext irgendeines Vorgangs gemeint ist. Z.B. wird eine Variablen-Auswertung im Kontext einer bestimmten Substitutions-Umgebung durchgeführt. Wenn klar ist, welche Art von Umgebung gemeint ist, sprechen wir einfach von „Umgebung“.

Notation 2.1.2.6 (für die Semantik-Regeln, vgl. [8], [10])

SemVal	Menge der semantischen Werte. SemVal := GT $\cup$ $\perp \cup$ Cl, wobei GT die Menge aller Grundterme ist[10] und Cl die Menge aller Closures (s.u.).
{ (var_1, $expr_1$), … }, E, E', E_n	Umgebung
E $\oplus$ E'	*Bindungssumme* von E und E', d.h. die Umgebung, die genau alle Bindungen von E und E' enthält. Ausnahme: Bindungen in E' ersetzen Bindungen in E, falls in diesen Bindungen jeweils die selbe Variable gebunden wird.
E(var)	Der Ausdruck, der der Variablen var in der Umgebung E zugeordnet ist.
v, v_n	Semantischer Wert (v, $v_n \in$ SemVal)
$\underline{\lambda}$ …	λ-Abstraktion. Tritt $\underline{\lambda}$ … mehrmals in einer Regel auf, so ist immer dieselbe λ–Abstraktion gemeint.
S $\vdash$ x $\rightarrow$ v	*Herleitung* eines semantischen Wertes v aus der Konstruktion x beim Vorliegen des *Zustandes* S. S ist der für die Herleitung aus x relevante Kontext; er umfaßt in unserer Praxis eine Umgebung sowie ggfs. einen semantischen Wert, der als Argument einer β-Regel fungiert (mehr dazu siehe Regel 2.1.2.7(5))[11]. x ist ein syntaktisches Konstrukt oder eine Closure (s.u.).

[10] Der Einfachheit halber unterscheiden wir in der Notation nicht zwischen solchen „semantischen Grundtermen“ und den in 2.1.1 definierten „syntaktischen Grundtermen“. Ob es sich um das eine oder das andere handelt, ergibt sich an Ort und Stelle aus dem jeweiligen Verwendungskontext.

[11] Bei einer denotationalen Semantik ist es eigentlich üblich, daß sowohl vor $\vdash$ als auch *nach* $\rightarrow$ jeweils ein Zustand angegeben ist; in unseren Regeln treten aber rechts vom Pfeil ausschließlich semantische Werte auf, gewissermaßen als Herleitungs-„Resultat“.

semVal(x)	Semantischer Wert der syntaktischen Konstruktion x. Dies ist der semantische Wert v, falls es eine *gültige Herleitung* … ⊢ x ➔ v gibt. Eine Herleitung ist genau dann gültig, wenn es eine *Regel* gibt, in der die Herleitung gültig ist (siehe unten). Andernfalls hat x den semantischen Wert ⊥.Wir sagen, x wurde mit semVal(x) *bewertet.*
≪ A, E ≫	*Closure* (*Abschluß*), d.h. der semantische Wert des Ausdrucks A in der Umgebung E[12]. Closures zählen sowohl zu den semantischen Werten als auch zu den syntaktischen Konstrukten.
⊥	*Bottom* (syntaktisch und semantisch)
Voraussetzung$_1$ … Voraussetzung$_n$ ——————— … ➔ …	*Regel*. Genau dann wenn alle n Voraussetzungen logisch gelten, ist die Herleitung unter dem Strich gültig. Falls n = 0, ist die Herleitung immer gültig.

[12] Da unsere Kernsprache *lazy*, d.h. nicht-strikt auswerten soll (vgl. 2.1), benötigen wir einen Mechanismus, um den semantischen Wert eines Ausdrucks eindeutig definieren zu können, ohne den Ausdruck vorher zu diesem Wert evaluieren zu müssen. Da der semantische Wert eines Ausdrucks nur von der jeweiligen Umgebung abhängt, in der sie Auswertung stattfindet (genau gesagt der Belegung seiner freien Variablen), können wir dazu eine solche *Closure* verwenden, die den Ausdruck und die Umgebung zusammenfaßt und wie ein semantischer Wert verwendet werden kann. Der konkrete Gebrauch von Closures ist bei Regel 2.1.2.7(4) erläutert.

Semantische Regeln 2.1.2.7 (der definiten Kernsprache)

(1) Variable:

$$\frac{E(\text{Variable}) = v}{E \vdash \text{Variable} \rightarrow v}$$

Erläuterung: Ist eine Variable in einer Umgebung E gebunden, so ist ihr semantischer Wert der des durch die Bindung mit ihr assoziierten Ausdrucks.

(2) Konstruktor:

$$\frac{}{E \vdash \text{Konstruktor} \rightarrow \text{Konstruktor}}$$

Erläuterung: „Konstruktor“ auf der rechten Seite des Herleitungspfeils steht natürlich für einen semantischen Wert; wir unterscheiden der Einfachheit halber bei der Schreibung nicht zwischen „syntaktischen“ und „semantischen“ Grundtermen (vgl. Notation 2.1.2.6).

(3) Konstruktor-Applikation:

$$\frac{E \vdash \text{Ausdruck} \rightarrow \text{Konstruktor } v_1 \ldots v_n \qquad E \vdash \text{Ausdruck'} \rightarrow v}{E \vdash \underline{\text{Applikation}}\ (\text{Ausdruck, Ausdruck'}) \rightarrow \text{Konstruktor } v_1 \ldots v_n\ v}$$

Erläuterung: Es handelt sich hier um ein Auswertung, bei der der Kopf des auszuwertenden Ausdrucks konstant bleibt. Daher nennt man Ausdrücke der Form Konstruktor $e_1 \ldots e_n$ auch *Kopf-Normalformen*. Mehr dazu siehe 3.3.

(4) Funktionale Applikation:

$$\frac{E \vdash \text{Ausdruck} \rightarrow \ll \underline{\lambda} \ldots, E' \gg \quad E \oplus E', \ll \text{Ausdruck}', E \gg \vdash \underline{\lambda} \ldots \rightarrow v}{E \vdash \underline{\text{Applikation}}\ (\text{Ausdruck}, \text{Ausdruck}') \rightarrow v}$$

Erläuterung: Die Closure $\ll \underline{\lambda}\ldots, E' \gg$ ist der semantische Wert von Ausdruck. Sie wird ursprünglich durch eine Top-level-Bindung einer Variablen als Wert zugewiesen (vgl. Regel (11)). Solche Variablen nennt man *Funktionen*. Wir benötigen die Closure, da diesem funktionalen Ausdruck, solange er für sich allein steht (wie in der ersten Zeile der Regel), gewissermaßen Argumente „fehlen“, um ihm einen Grundterm als semantischen Wert zuordnen zu können. In der zweiten Zeile der Regel fungiert Ausdruck' als Argument der λ-Abstraktion $\underline{\lambda}$ …. Hier wird die Closure $\ll$ Ausdruck', E $\gg$ gebraucht, um eine nicht-strikte Semantik der Funktionsapplikation zu realisieren: Wir wollen den semantischen Wert von Ausdruck' nicht schon *vor* dem Funktionsaufruf, d.h. der Durchführung der mit λ … assoziierten β-Regel, bestimmen (wie dies bei *call-by-value* geschieht), sondern erst bei Bedarf. Wann dies genau notwendig ist, und in welchem Ausmaß, bestimmt sich allein durch die für die Semantik von λ … zuständige Regel (vgl. Regel (5)-(8)). Man beachte, daß die Closure auf der linken Seite von ⊢ den semantischen Wert des in ihr enthaltenen Ausdrucks darstellt. Daher repräsentiert in der zweiten Zeile der Regel die Closure $\ll$ Ausdruck', E $\gg$ den *Wert* von Ausdruck' (bewertet in der Umgebung E).

(4') Closure:

$$\frac{E' \vdash \text{Ausdruck} \rightarrow v}{E \vdash \ll \text{Ausdruck}, E' \gg \rightarrow v}$$

Erläuterung: Diese Regel ist für den Fall vorgesehen, daß der Wert eines Ausdrucks, der als semantischen Wert eine Closure besitzt, nicht im Rahmen eines Applikation-Ausdrucks (also als funktionaler Ausdruck mit Argumenten) hergeleitet wird. Dies ist dann der Fall, wenn Ausdruck der Rumpf einer parameterlosen Top-level-Bindung ist (vgl. Regel (10)).

(5) β-Regel (unabweisbares, erfolgreiches Matching):

$$\frac{E \oplus \{ (\text{Variable}, v) \} \vdash \text{Ausdruck} \rightarrow v'}{E, v \vdash \underline{\lambda} (\text{Variable}) \rightarrow \text{Ausdruck} \rightarrow v'}$$

Erläuterung: Die Closure v wird gegen die Variable *gematcht*[13]. Dadurch wird Ausdruck in einer Umgebung bewertet, die die Bindung der Variablen an v enthält. v' ist der semantische Wert von Ausdruck in dieser Umgebung.

(6) β-Regel (erfolgreiches Matching gegen KonstMuster):

$$\frac{\begin{array}{c} v = \text{Konstruktor } v_1 \ldots v_n \\ E \oplus \{ (\text{Variable}_1 , v_1), \ldots, (\text{Variable}_n , v_n) \} \vdash \text{Ausdruck} \rightarrow v' \end{array}}{E, v \vdash \underline{\lambda} (\text{Konstruktor Variable}_1 \ldots \text{Variable}_n) \rightarrow \text{Ausdruck} \underline{;} _ \rightarrow \text{Ausdruck'} \rightarrow v'}$$

Erläuterung: Der semantische Wert v wird erfolgreich gegen das Muster Konstruktor Variable_1 … Variable_n gematcht (d.h. der Kopf von v und der des Musters sind identisch und haben dieselbe Zahl n von Argumenten. Bemerkenswert ist in diesem Zusammenhang, daß daher für das Matching nur der Kopf des semantischen Wertes von $\text{Ausdruck}_{\text{Test}}$ festgestellt werden muß)[14]. Der Wert der λ-Abstraktion im Kontext von E, v ist somit der von $\text{Ausdruck}[v_1/\text{Variable}_1, \ldots, v_n/\text{Variable}_n]$.

---

[13] vgl. Regel (4)

[14] Die Regeln (6) bis (8) sind nur im Zusammenhang mit Regel (9) (Verzweigung) von Bedeutung. Daher wäre es auch möglich gewesen, statt diesen Regeln drei Regeln für die Verzweigung einzuführen. Es erscheint dem Autor jedoch sinnvoller, stattdessen einen möglichst expliziten Bezug zum λ-Kalkül herzustellen, das ja schließlich die theoretische Grundlage der Kernsprache darstellt.

(7) β-Regel (erfolgloses Matching):

$$\frac{v \neq \text{Konstruktor } v_1 \ldots v_n \ \wedge\ v \neq \bot \qquad E \vdash \text{Ausdruck'} \rightarrow v'}{E, v \vdash \underline{\lambda}\ (\text{Konstruktor Variable}_1 \ldots \text{Variable}_n) \rightarrow \text{Ausdruck}\ \underline{;}\ _ \rightarrow \text{Ausdruck'} \rightarrow v'}$$

Erläuterung: Hier paßt das Muster der λ-Abstraktion nicht zum Wert v. Daher ist das Herleitungsresultat der Wert von Ausdruck'.

(8) β-Regel (Matching mit Bottom):

$$\frac{v = \bot}{E, v \vdash \underline{\lambda}\ (\text{Konstruktor Variable}_1 \ldots \text{Variable}_n) \rightarrow \text{Ausdruck}\ \underline{;}\ _ \rightarrow \text{Ausdruck'} \rightarrow \bot}$$

Erläuterung: Matching von ⊥ gegen ein KonstMuster ergibt immer ⊥. Man beachte jedoch, daß es kein Äquivalent dieser Regel für Zuordnungen (vgl. Regel (5)) gibt. Dort nimmt ⊥, anders als bei strikten Programmiersprachen, keine Sonderrolle ein.

(9) Verzweigung[15]:

$$\frac{E \vdash \text{Ausdruck}_{\text{Test}} \rightarrow v \qquad E, v \vdash \underline{\lambda}\ \text{Muster} \rightarrow \text{Ausdruck}_A\ ;\ _ \rightarrow \text{Ausdruck}_B \rightarrow v'}{E \vdash \underline{\text{Verzweigung}}\ (\text{Ausdruck}_{\text{Test}}, \text{Muster}, \text{Ausdruck}_A, \text{Ausdruck}_B) \rightarrow) v'}$$

Erläuterung: Hier hängt der Wert des Gesamtausdrucks davon ab, ob der semantische Wert des sogenannten *Testausdrucks* $\text{Ausdruck}_{\text{Test}}$ erfolgreich gegen das Muster matcht. Wenn ja, so ist der Wert der ganzen Verzweigung der von Ausdruck_A, und andernfalls der von Ausdruck_B, oder ⊥, falls der Wert von $\text{Ausdruck}_{\text{Test}}$ ⊥ ist (vgl. Regel (5) bis (8)).

[15] vgl. die `case`-Ausdrücke von Haskell

(10) Top-level-Bindung (1) (einfache Variablenbindung):

$$\frac{}{E \vdash \text{Variable} \underline{=} \text{Ausdruck} \rightarrow \{ (\text{Variable}, \ll \text{Ausdruck}, E \gg) \}}$$

Erläuterung: Ausdruck wird („verpackt" in einer Closure) der Variablen zugewiesen. Die Closure ist notwendig, weil die Herleitung des semantischen Wertes von Ausdruck in der Umgebung E erfolgen muß (vgl. Regel (4')); E ist hier die Menge aller Top-level-Bindungen.

(11) Top-level-Bindung (2):

$$\frac{}{\begin{array}{c} E \vdash \text{Variable} \underline{=} \underline{\lambda} (\text{Variable}_1, \ldots, \text{Variable}_n) \rightarrow \text{Ausdruck} \\ \rightarrow \{ (\text{Variable}, \ll \underline{\lambda} (\text{Variable}_1) \rightarrow \ldots (\underline{\lambda} (\text{Variable}_n) \rightarrow \text{Ausdruck}), E \gg) \} \end{array}}$$

Erläuterung: Hier wird der „Variable" ein Ausdruck zugewiesen, der einen Higher-order-Wert in Form der Closure $\ll \underline{\lambda} (\text{Variable}_1) \rightarrow \ldots (\underline{\lambda} (\text{Variable}_n) \rightarrow \text{Ausdruck}), E \gg$ besitzt. Dadurch wird die Variable zur *Funktion*. Bei ihrer Applikation (vgl. Regel (4)) werden den Variablen $\text{Variable}_{1..n}$ Werte mittels den zugehörigen *unabweisbaren* λ–Abstraktionen zugewiesen. Für die Umgebung E gilt der Regel (10) entsprechendes.

(12) Bindungssumme:

$$\frac{\begin{array}{c} \forall i: 1 \leq i \leq n, E \oplus E' \vdash \text{Top-level-Bindung}_i \rightarrow E_i \\ E' = \oplus(\forall j: 1 \leq j \leq n, E_j) \end{array}}{E \vdash \{ \text{Top-level-Bindung}_1, \ldots, \text{Top-level-Bindung}_n \} \rightarrow E'}$$

Erläuterung: Ein Programm besteht aus einer Menge von Top-level-Bindungen. Diese Bindungen bilden zusammengenommen die Umgebung E' (die sogenannte *Bindungssumme*), in der der semantische Wert der Variablen „main" zu bestimmen ist (vgl. Regel (13*)).

(13*) Definites Programm:

$$\frac{\{\} \vdash \{\ \text{Top-level-Bindung}_1, \ldots, \text{Top-level-Bindung}_n\ \} \rightarrow E \qquad E \vdash \text{Variable „main“} \rightarrow v}{\{\} \vdash \text{Top-level-Bindung}_1, \ldots, \text{Top-level-Bindung}_n \rightarrow v}$$

Erläuterung: Der semantische Wert eines Programms der Kernsprache ist derjenige der Variable „main“ in der Umgebung aller Top-level-Bindungen.

2.2 Allgemeine Or-Sets

Wir definieren nun sogenannte *Allgemeine Or-Sets* (kurz „Or-Sets“) als beliebige Teilmengen *algebraischer Datentypen*. Dazu benötigen wir den Begriff des *algebraischen Datentyps* (kurz „Typ“). Wir nehmen folgendes an: Statisch, d.h. vor der Laufzeit eines Programms unserer Kernsprache werden durch einen Mechanismus namens *Typinferenz* allen Ausdrücken im Programm Typen zugeordnet. Ist diese Zuordnung, die nach den Regeln eines sogenannten *Typschemas* erfolgt, korrekt beendet worden, spielen Typen zur Laufzeit keine Rolle mehr. Sie dienen also nur dazu, eine gewisse elementare Korrektheit eines Programms sicherzustellen. Wir führen Typen nur deswegen ein, um klarzumachen, daß sich ein Or-Set aus Elementen desselben Typs zusammensetzen muß. Später gehen wir stillschweigend davon aus, daß alle Programme, Or-Sets usw. *typkorrekt* im Sinne der Typinferenz sind. Folgende Definition ist übrigens nicht geeignet, als Grundlage einer Typinferenz zu dienen, reicht aber für unsere Zwecke aus[16]:

Definition 2.2.1:
Eine *Familie algebraischer Datentypen* ist eine Menge von mutuell disjunkten, nicht-leeren Mengen von Grundtermen. Da Grundterme nach 2.1.2.3 (Werte-)Bäume sind, können wir algebraische Datentypen als *(Werte-)Wälder* betrachten.

Definitionen 2.2.2:
Es sei T ein algebraischer Datentyp. Dann heißt die leere Menge sowie jede Teilmenge von T (einschließlich T selbst) *Allgemeines Or-Set* des Typs T. Die Elemente eines Or-Sets heißen *Alternativen*. Elemente aus T sowie Or-Sets mit genau einer Alternative heißen *definit*, alle anderen Or-Sets *indefinit*. Allgemeine Or-Sets werden genau wie Mengen, aber mit spitzen anstatt geschweiften Klammern notiert. Die Menge aller Or-Sets eines Typs T definieren wir als $OrSets(T) := Pow(T)$. $T\uparrow := T \cup OrSets(T)$ heißt die sogenannte *erweiterte Domain* des Typs T. Analog zu Typen sind Or-Sets Werte-Wälder.
Sei O ein Or-Set des Typs T. Sei P eine Aussageform $P: T \rightarrow \{wahr, falsch\}$. Wir schreiben $P\uparrow(O)$ für $\forall a: a \in O \Rightarrow P(a)$. Analog sei $f1\uparrow : T\uparrow \rightarrow T'\uparrow$ für eine Funktion $f1 : T \rightarrow T'$ definiert durch $f1\uparrow(O) := < f1(a) : a \in O >$ und $f2\uparrow : T\uparrow \times T\uparrow \rightarrow T'\uparrow$ für $f2: T \times T \rightarrow T'$ sei definiert durch $f2\uparrow (O_A, O_B) := < f2(a,b) : a \in O_A, b \in O_B >$ usw.. Solche Operationen der Bereichsausdehnung einer Aussage oder einer Funktion bezeichnen wir als *Lifting*. Als *Wert eines Or-Sets* O bezeichnen wir eine beliebige Alternative a aus O, wenn im jeweiligen Kontext statt a auch jedes andere Element aus O verwendet werden könnte.

---

[16] Eine engere Definition findet sich in 4.3

Intuitiv repräsentiert ein Or-Set genau eine seiner Alternativen (vgl. Kapitel 1). Weil unbekannt ist, welches Element dies ist, müssen in Form einer Menge alle in Frage kommenden Werte angegeben bzw. bearbeitet werden (*Don't-know-Nichtdeterminismus*)[17].

2.3 Or-Set-Generatoren

Wir wollen nun also die bisher definierte definite Programmiersprache um Elemente erweitern, die es uns erlauben, mit indefiniten Werten ähnlich zu arbeiten wie mit definiten Werten. Diese Erweiterung ist im Prinzip recht einfach: Zuerst benötigen wir eine syntaktische Möglichkeit, Or-Sets generieren zu können[18]. Dies leisten die sogenannten *Or-Set-Generatoren*. Bei der Motivation der Semantik der Or-Set-Generatoren werden wir uns an folgende Leitlinie halten: Da in einer nicht-strikten Programmiersprache Ausdrücke solange unausgewertet bleiben, bis ihr Wert tatsächlich benötigt wird[19], behandeln wir solche Mengen solange wie möglich analog zu einem definiten First-class-Objekt. Da aber Or-Sets keine First-class-Werte sein sollen (vgl. Kapitel 1), operieren alle Funktionen unserer Sprache explizit auf definite Werte[20]. U.a. an dieser Stelle müssen wir also auf die definiten Alternativen innerhalb des Or-Sets Bezug nehmen. Derartige Operationen auf indefinite Daten führen wir dann einfach auf deren einzelne definiten Alternativen aus und fügen die Resultate wieder zu einem indefiniten Datum zusammen (wir nennen dieses Vorgehen *Lifting-Regel,* vgl. Definitionen 2.2.2). Wie wir noch sehen werden, führt die Technik des Liftings allerdings zu Schwierigkeiten, falls die Zahl der definiten Alternativen eines indefiniten Wertes sehr groß oder gar unendlich ist, oder wenn die einzelnen Alternativen nur schwer bestimmbar sind. Außerdem werden wir später auf eine semantische Unklarheit bei der Verknüpfung von als *identisch* intendierten Or-Sets stoßen.

---

[17] Wie wir schon in der Einleitung gesagt haben, werden im Rahmen dieser Arbeit keinerlei Annahmen über die „innere Struktur" eines Or-Sets gemacht, die über die Feststellung des (Nicht-)Vorhandenseins eines Wertes im Or-Set hinausgehen. Alle Elemente eines Or-Sets werden gleich behandelt. Naheliegend wäre es aber z.B., analog zur Dempster-Shafer-Theorie den Elementen jeweils einen Wert zwischen 0 und 1 zuzuordnen, der angibt, wie hoch die Wahrscheinlichkeit ist, daß die jeweilige Alternative gleich dem "wahren Wert" des Or-Sets ist.

[18] Damit ist nicht nur die explizite Aufzählung von semantischen Werten gemeint. Es gibt bekanntlich viele Möglichkeiten, Mengen zu notieren.

[19] D.h. in Haskell-ähnlichen Sprachen praktisch: beim Pattern-Matching und bei der Darstellung von Resultaten als Grundterme

[20] Etwas anders ausgedrückt: wir erlauben keine „Or-Set-Konstruktoren", die in Mustern vorkommen könnten. Wir werden zwar in 4.10 Meta-Operationen auf Or-Sets kennenlernen, diese stellen aber Ausnahmen dar und sind nicht Bestandteil der deklarativen Semantik.

Wir werden im folgenden unsere Kernsprache um drei verschiedene Arten von Or-Set-Generatoren zu einer indefiniten Programmiersprache ergänzen: *Or-Ausdrücke*, *das leere Or-Set* sowie *Query-Variablen*. Alle Or-Set-Generatoren haben als bezeichnende Gemeinsamkeit, daß ihr semantischer Wert ein Or-Set ist. Daher bezeichnen wir Or-Set-Generatoren oft auch einfach als „Or-Sets“, falls dadurch keine Mißverständnisse entstehen[21]. Als weitere Gemeinsamkeit können alle Or-Set-Generatoren syntaktisch überall dort stehen, wo auch sonstige Ausdrücke erlaubt sind. Dadurch ist es für den Programmierer möglich, mit indefiniten Daten prinzipiell genauso zu arbeiten wie mit herkömmlichen Daten.
Um die Semantik der Or-Set-Generatoren festzulegen, ist zum einen für den jeweiligen Or-Set-Generator als Ausdruck eine neue Regel einzuführen. Zum anderen müssen einige schon für die definite Sprache aufgestellte Regeln ergänzt werden. Neben der Erweiterung der Syntax und der Einführung neuer Semantik-Regeln für die Or-Set-Generatoren besteht die Erweiterung der Kernsprache im wesentlichen nur darin, SemVal um Or-Sets zu erweitern. Nur eine einzige bereits eingeführte Semantik-Regel muß angepaßt werden (Regel (13*)).
Anmerkung: Die operationelle Semantik der Or-Set-Generatoren (und damit die Kernsprache, wie sie in diesem Kapitel definiert ist) ist weniger leistungsfähig als die deklarative (vgl. 4.6.3 und 4.6.4). Dieser Unterschied ist aber gerechtfertigt, da es im Rahmen dieser Arbeit nicht darauf ankommt, die deklarative Grammatik möglichst genau auf die technischen Möglichkeiten einer konkreten Implementation auszurichten, sondern vielmehr eine klare, abstrakte und umfassende theoretische Fundierung geschaffen werden sollte.

2.3.1 Or-Ausdrücke

Wir repräsentieren ein indefinites Datum im Quellcode, indem wir mehrerer alternativer Ausdrücke zur Berechnung dieses Datums angeben. Der Wert dieses *indefiniten Gesamtausdrucks* ist ein Or-Set, das genau die Werte aller (möglicherweise wiederum indefiniten) alternativen Ausdrücke umfaßt. Wie leicht einzusehen ist, reicht als neues syntaktisches Element eine Form von Ausdruck aus, die zwei alternative Ausdrücke verknüpft; der semantische Wert des Gesamtausdrucks ist dann die Vereinigung[22] der semantischen Werte der Alternativen (bzw., falls eine der Alternativen definit ist, des entsprechenden einelementigen definiten Or-Sets (vgl. Definitionen 2.2.2)) zu einem neuen Or-Set. Wir notieren solche Ausdrücke in der Form or(a, b) und nennen sie *Or-Ausdrücke*. Hier taucht ein Problem auf:

---

[21] Ähnlich wie man bestimmte Vorkommen von Konstruktoren in einem Quelltext als z.B. Zahlen oder Strings bezeichnet.

[22] Es sei daran erinnert, daß indefinite Werte keinen eigenen Typ haben (vgl. 2.3) und es analog dazu keine „verschachtelten“ Or-Sets gibt (vgl. Definitionen 2.2.2). Daher bleibt nur die Vereinigung.

Kommt an einer bestimmten Stelle in einem Programm solch ein Ausdruck vor, so ist es natürlich möglich, daß dieses *eine* Vorkommen an *mehreren* Stellen im Programm bewertet wird, was, wie wir gleich sehen werden, zu einer semantischen Unklarheit führt. Gegeben sei beispielsweise folgendes Programmfragment:

```
x = or (0, 1)          ( 0 und 1 seien Konstruktoren, sum die Additionsfunktion)
main = sum x x
```

Hier wird der semantische Wert von x an zwei Stellen (im Rumpf von main) verwendet. Bewerten wir die beiden Vorkommen von x, so erhalten wir zweimal das Or-Set < 0, 1 >. Gemäß der oben aufgestellten Lifting-Regel sind von sum die einzelnen Alternativen dieser beiden Or-Sets zu addieren und wir erhalten die Teilresultate 0+0, 0+1, 1+0 und 1+1. Das Gesamtresultat lautet somit < 0, 1, 2 >. Wir könnten uns aber auch auf den Standpunkt stellen, die beiden Vorkommnisse der einen Variablen x im Rumpf von main bezeichnete beide Male *denselben* (unbekannten) definiten Wert, d.h. entweder *beide* Male 0 oder *beide* Male 1. In diesem Fall würde man die Teilresultate 0+0 und 1+1 und mithin das Gesamtresultat < 0, 2 > erwarten. Beide Sichtweisen sind legitim: Das Resultat < 0, 1, 2 > wirkt unter dem Paradigma der funktionalen Programmierung natürlicher: x hat den Wert < 0, 1 > und sollte daher überall durch den gleich bewerteten Ausdruck or(0, 1) ersetzbar sein (gemäß dem für funktionale Sprachen gültigen oder zumindest anzustrebenden Prinzip der *referentiellen Transparenz*). Dagegen entspricht das Ergebnis < 0, 2 > mehr der menschlichen Intuition und verweist (sogar wenn man das Resultat < 0, 1, 2 > für „korrekter“ hält) auf eine allgemeinere Problematik: Die Syntax or(a, b) gibt uns keine Möglichkeit auszudrücken, daß zwei Vorkommen desselben Ausdrucks or(a, b) auch denselben unbekannten Wert x bezeichnen sollen. Die Lösung des Problems ist, einem Or-Sets (als semantischem Wert eines Or-Ausdrucks) optional eine *Identität* zu verleihen, indem man ihm einen *(Or-Set-)Identifikator* zuweist. Werten wir mehrere Or-Ausdrücke zu Or-Sets aus und verleihen diesen denselben Identifikator, so sollen alle diese indefiniten semantischen Werte denselben (unbekannten) definiten Wert beschreiben. Wenn wir dann Resultate erhalten, die aus „Verknüpfungen“ *unterschiedlicher* Alternativen indefiniter semantischer Werte mit *demselben* Identifikator entstanden sind, so sind diese Resultate ungültig. Im obigen Programmfragment werden durch die Lifting-Regel die Teilresultate 0+0, 0+1, 1+0 und 1+1 generiert. Wenn wir annehmen, daß jeder durch den Or-Ausdruck or(0, 1) generierte semantische Wert denselben Identifikator hat, so sind die Teilresultate 0+1 und 1+0 ungültig. Der semantische Wert des Programms ist somit < 0, 2 >.

Bevor wir Semantik-Regeln für indefinite Werte angeben können, müssen wir nun noch den Begriff der „Verknüpfung“ etwas genauer fassen. Viele der bisher aufgestellten Regeln, in denen semantische Werte vorkommen, nehmen keinen Bezug auf die Art dieser Werte. Für die Anwendung der Regel (5) (Matching eines Wertes gegen eine Variable als Muster)

beispielsweise spielt der zu matchende Werte keinerlei Rolle, sogar für ⊥ ist das Matching erfolgreich. Das Prinzip, Werte so spät wie möglich zu dekonstruieren (d.h. Bestandteile des jeweiligen Grundterms zu selektieren), macht die „Laziness“ unserer Programmiersprache aus. Für gewisse Regeln ist die Abstraktion von der genauen Struktur semantischer Werte nicht mehr möglich: es handelt sich dabei um die Regeln für das Pattern-Matching gegen Konstruktor-Applikationen ((6ff)) und um die Regeln für die Applikation ((3ff)). Da Or-Sets, wie wir am Anfang von 2.3.1 gefordert haben, „als ganze“ keine First-class-Werte unserer Sprache sein sollen, müssen wir an diesen Stellen auf deren einzelne Alternativen Bezug nehmen und die genannten Regeln für jede einzelne Alternative gesondert anwenden. Die Resultate der jeweiligen Herleitungen verknüpfen wir dann zu einem neuen Or-Set. Wir *liften* die jeweilige Regel für ihre Verwendung mit einem Or-Set. Bei jeder gesonderten Anwendung der entsprechenden Regel treffen wir eine *Entscheidung* für eine der Alternative des Or-Sets. Jedes einzelne Herleitungsresultat kommt unter der Voraussetzung zustande, daß der „wahre Wert“ des Or-Sets genau diese Alternative ist. Den Vorgang der Aufteilung des Or-Sets in seine Alternativen und deren getrennte Behandlung nennen wir die *Distribution* des Or-Sets. Hat nun das distribuierte[23] Or-Set eine Identität i, so kommt jedes einzelne Herleitungsresultat unter einer bestimmten Annahme über den *definiten* wahren Wert des Or-Sets i zustande. Ist eines der Einzelresultate ein Or-Set, und ist darin als Alternative ein Wert enthalten, der unter einer unterschiedlichen Annahme über i zustande gekommen ist, so ist dieser Wert ungültig und muß aus dem Or-Set entfernt werden. Den Vorgang des Entfernens ungültiger Alternativen nach einer Distribution nennen wir die *Validierung* des Or-Sets. Um die Validierung einfacher gestalten zu können, erlauben wir „verschachtelte Or-Sets“ als temporäre indefinite semantische Werte: Verknüpfen wir mehrere semantische Werte zu einem Or-Set, und ist einer dieser semantischen Werte selbst ein Or-Set (seine Identität sei i), so fügen wir dieses Or-Set als ganzes als Alternative hinzu, und nicht seine einzelnen Elemente. Dadurch behalten wir für diese Objekte die Information, daß es sich um Alternativen des Or-Sets i handelt. Kommt unser Or-Set i im Teilresultat einer Distribution vor, und lautet die Prämisse für das Zustandekommen dieses Teilresultats, daß der wahre Wert des Or-Sets i gleich x sei, so können wir das Or-Set zum Wert x verkürzen.

Wir müssen jetzt genau für diejenigen Regeln „indefinite“ Versionen bereitstellen, bei denen auf die innere Struktur von semantischen Werten Bezug genommen wird, bei denen also ein Wert dekonstruiert wird, indem auf Elemente des Grundterms Bezug genommen wird[24]. Bei genau den Regeln (3), (4), (6) und (7) ist dies erforderlich.

Als Identifikatoren von Or-Sets verwenden wir der Einfachheit halber semantische Werte. Die Zuweisung eines Identifikators erfolgt durch die Funktion verleiheIdentität. Wir nennen

---

[23] (sic!)

[24] Genau gesagt: auf den Kopf von Konstruktor-Applikationen sowie auf Closures

Ausdrücke der Form verleiheIdentität(Or-Set-Generator, Identifikator) *Bestimmte Or-Sets* und Or-Ausdrücke die nicht in dieser Form auftreten *Unbestimmte Or-Sets.* Hat ein Or-Set keinen Identifikator, so dürfen seine Alternativen beliebig mit Alternativen aller Or-Sets verknüpft werden, d.h. es bleibt bei einer Validierung immer unberücksichtigt. Unter „Verknüpfungen" sind genau die Resultate der Herleitungen der neuen Semantik-Regeln (15)/(15') und (16)/(16') zu verstehen, bei denen es sich um die für Or-Sets angepaßten Versionen der bereits bekannten Regeln (3), (4), (6) und (7) handelt (vgl. die Einleitung zu 2.3)[25]. Für die Validierung ist die Funktion validiereSemVal zuständig.

Wir haben bisher noch nichts darüber gesagt, wann genau einem Or-Set mit verleiheIdentität eine Identität verliehen werden soll. Um die Angelegenheit etwas zu vereinfachen, nehmen wir an, daß in einem Programm verleiheIdentität nur in Form von Bestimmten Or-Sets auftritt. Ist ein Or-Ausdruck unter dieser Voraussetzung ein Unbestimmtes Or-Set, so wird bei jeder Bewertung eines solchen Or-Set-Generators ein neues Or-Set generiert, das von anderen auf diese Weise generierten Or-Sets nicht zu unterscheiden ist. Weil zur Programm-Laufzeit bei jeder Evaluierung eines Unbestimmten Or-Sets ein neues Or-Set generiert wird, spricht man auch von einer *dynamischen Generierung von Or-Sets* (vgl. 4.4.3). Unbestimmte Or-Sets werden nützlich sein, wenn wir später sogenannte *Baum-Grammatiken* durch indefinite Ausdrücke darstellen wollen (vgl. 3.2). Weiterhin verwenden wir Unbestimmte Or-Sets, um indefinite Programm-Resultate darzustellen[26]; da Resultate nicht mehr weiter ausgewertet werden, spielt die Identität des Or-Sets dann keine Rolle mehr.

Die naheliegende Möglichkeit, den Identifikator für Bestimmte Or-Sets festzulegen, ist, jedem Vorkommen eines Bestimmten Or-Sets im Programm statisch, d.h. vor der Laufzeit, genau einen Identifikator zu verleihen (lexikalische Zuordnung der Identifikatoren)[27]. Dieses Vorgehen, daß wir bei unserem obigen Beispielprogramm angewendet haben, um das Resultat < 0, 2 > zu erhalten, setzt die natürliche Erwartung um, daß jedes Vorkommen eines Or-Ausdrucks genau einen bestimmten Wert beschreibt. Man kann den Wert eines indefiniten Ausdrucks (bzw. Programms), der als Or-Set-Generatoren ausschließlich Bestimmte Or-Sets mit lexikalischer Zuordnung von Identifikatoren in expliziter Darstellung enthält, auch als Or-Set von Resultaten gewisser definiter Ausdrücke (d.h. Ausdrücken ohne Vorkommen von Or-Set-Generatoren) auffassen. Diese Ausdrücke werden aus dem indefiniten Ausdruck gewonnen, in dem für jedes

---

[25] Die Additionsfunktion in unserem obigen Beispiel ist keine Verknüpfung in diesem Sinne, ist aber zur Veranschaulichung besser geeignet, da sie eingängiger ist als z.B. die β-Regel. Die konkrete Art und Weise der Verknüpfungen und Distribution ist bei den jeweiligen Regeln erklärt (siehe 2.3.1.2.2).

[26] Es ist für funktionale Programmiersprachen allgemein üblich, Berechnungsresultate in der gleichen Notation darzustellen wie Ausdrücke im Quellcode.

[27] Diese Zuordnung wird bei der Prolid-Syntax a `or` b realisiert (siehe 4.4.3)

Bestimmte Or-Set nacheinander genau einer von dessen beiden Teilausdrücke gestrichen wird. Macht man dies mit allen im Ausdruck vorkommenden Bestimmten Or-Sets in allen möglichen Kombinationen, so erhält man bei n Vorkommen von Bestimmten Or-Sets in einem Ausdruck 2^n definite Ausdrücke. Faßt man die definiten semantischen Werte dieser Ausdrücke zu einem Or-Set zusammen, erhält man den Wert des indefiniten Ausdrucks.

Man beachte, daß die lexikalische Zuordnung eines Identifikators operationell mit einer dynamischen Validierung des identifizierten Or-Sets einhergeht: Tritt das Bestimmte Or-Set in einem Rumpf einer rekursiven Funktion auf, und wird der Funktionsrumpf bei der Distribution des Wertes des Bestimmten Or-Sets rekursiv neu bewertet, so wird das Bestimmte Or-Set bei dieser rekursiven Bewertung zu einem definiten Wert evaluiert. Das Bestimmte Or-Set verhält sich bei einer lexikalischen Zuordnung des Identifikators in gewissem Sinne so wie eine globale Variable in einer prozeduralen Programmiersprache: Es existiert zur Laufzeit nur eine einzige, ständig sichtbare Bindung des Or-Sets an die Validierungs-Information (d.h. die Information, ob Alternativen des Or-Sets durch eine Distribution als ungültig anzusehen sind). Es kann aber nützlich sein, das Bestimmte Or-Sets *lokal* zu machen, indem wir zur Laufzeit vor jeder Auswertung des umgebenden Funktionsrumpfes dem generierten Or-Set einen neuen Identifikator zuweisen. Solche „lokalen Bestimmten Or-Sets“ werden wir in 2.3.3 in Gestalt der sogenannten *Query-Variablen* einführen.

2.3.1.1 Syntax

Zu der Grammatik der definiten Sprache (vgl. 2.1.1) werden folgende Produktionen hinzugefügt:

Ausdruck → or (Ausdruck, Ausdruck)

Ausdruck → verleiheIdentität(Ausdruck, Ausdruck)
Ausdruck → entferneIdentität(Ausdruck)

2.3.1.2 Dynamische Semantik

Definition 2.3.1.2.1:

SemVal wird um Or-Sets (mit und ohne Identifikator) ergänzt. $SemVal_{df}$ sei das bisherige SemVal:
$SemVal := SemVal_{df} \cup OrSets(SemVal) \cup \{ (orSet, i) : orSet \in OrSets(SemVal), i \in SemVal \}$

Beachte: Im folgenden schreiben wir indefinite semantische Werte wie Or-Sets in spitzen Klammern. Dennoch handelt es sich genau genommen nicht um Or-Sets. Insbesondere können indefinite semantische Werte verschachtelt werden, weil ihre Elemente (die wir ebenfalls als Alternativen bezeichnen) nicht definit sein müssen. Erst mit Regel (13) generieren wir ein „echtes“ Or-Set.

Semantische Regeln 2.3.1.2.2 (erste Erweiterung der definiten zur indefiniten Kernsprache)

Zu den Regeln der definiten Sprache werden folgende Regeln hinzugefügt:

(14) Or-Set-Generierung durch eine Or-Ausdruck:

$$\frac{E \vdash Ausdruck \Rightarrow v \quad E \vdash Ausdruck' \Rightarrow v'}{E \vdash \underline{or}\ (Ausdruck, Ausdruck') \Rightarrow \langle v, v' \rangle}$$

(15) β-Regel für einen indefiniten Wert ohne Identifikator:

$$v = < v_1, v_2, \ldots >$$
$$E, v_1 \vdash \underline{\lambda}\,(\text{Konstruktor Variable}_1 \ldots \text{Variable}_n) \rightarrow \text{Ausdruck} \;;\; _ \rightarrow \text{Ausdruck'} \Rightarrow v_1'$$
$$E, v_2 \vdash \underline{\lambda}\,(\text{Konstruktor Variable}_1 \ldots \text{Variable}_n) \rightarrow \text{Ausdruck} \;;\; _ \rightarrow \text{Ausdruck'} \Rightarrow v_2'$$
$$\ldots$$

$$E, v \vdash \underline{\lambda}\,(\text{Konstruktor Variable}_1 \ldots \text{Variable}_n) \rightarrow \text{Ausdruck} \;;\; _ \rightarrow \text{Ausdruck'}$$
$$\Rightarrow < v_1', v_2', \ldots >$$

Erläuterung: siehe Regel (15‘)

(15’) β-Regel für einem indefiniten semantischen Wert mit Identifikator:

$$v = (< v_1, v_2, \ldots >, i)$$
$$E, v_1 \vdash \underline{\lambda}\,(\text{Konstruktor Variable}_1 \ldots \text{Variable}_n) \rightarrow \text{Ausdruck} \;;\; _ \rightarrow \text{Ausdruck'} \Rightarrow v_1'$$
$$E, v_2 \vdash \underline{\lambda}\,(\text{Konstruktor Variable}_1 \ldots \text{Variable}_n) \rightarrow \text{Ausdruck} \;;\; _ \rightarrow \text{Ausdruck'} \Rightarrow v_2'$$
$$\ldots$$

$$E, v \vdash \underline{\lambda}\,(\text{Konstruktor Variable}_1 \ldots \text{Variable}_n) \rightarrow \text{Ausdruck} \;;\; _ \rightarrow \text{Ausdruck'}$$
$$\Rightarrow < \text{validiereSemVal}(v_1', \{ (i, v_1) \}), \text{validiereSemVal}(v_2', \{ (i, v_2) \}), \ldots >$$

Erläuterung: Die Regeln (15) und (15’) liften die β-Regel. Da das Muster einer λ-Abstraktion immer ein definiter Ausdruck ist (Grund: siehe 2.1), müssen wir die β-Regel auf die einzelnen Elemente des semantischen Wertes v anwenden. Die Resultate verknüpfen wir wieder zu einem Or-Set. Das Resultat $< v_1', v_2', \ldots >$ stellt für Regel (15’) eine Verknüpfung im Sinne von 2.3.1 dar: Das Or-Set $(< v_1, v_2, \ldots>, i)$ beschreibt einen bestimmten definiten Wert x_i. Werden die einzelnen β-Regeln angewendet (Zeile 2ff der Regel), so wird jedesmal eine andere Alternative in v als x_i angenommen (erst v_1, dann v_2 usw.). Wird nun (durch irgendeine andere Regel) eines der Teilresultate v_1', v_2' usw. verwendet, und ist in einem dieser Teilresultate[28] v_k' wiederum ein indefiniter semantischer Wert mit Identifikator i enthalten, so ist darin (nur) genau die

[28] (falls es indefinit ist)

Alternative v_k gültig (für diese Alternative als „wahren Wert“ x_i hatten wir uns bei der Anwendung der β-Regel auf v_k entschieden und unter dieser Prämisse das Resultat v_k‘ erhalten).

Die Hilfsfunktion validiereSemVal : SemVal × Gültigkeitsinformation → SemVal wird folgendermaßen definiert (mit Gültigkeitsinformation:= (i, v) : i, v ∈ SemVal):

Definition 2.3.1.2.3:

validiereSemVal(< v_1, v_2, … >, gi) → < validiereSemVal(v_1, gi), validiereSemVal(v_2, gi),… >

validiereSemVal ((< v_1, v_2, … >, i), gi) | falls (i, r) = gi für ein r ∈ SemVal → r

validiereSemVal ((< v_1, v_2, … >, i), gi) | sonst →
(< validiereSemVal(v_1, gi), validiereSemVal(v_2, gi),… >, i)

validiereSemVal (v, gi) | v ≠ < … > ∧ v ≠ (…,…) → v

(16) Applikation mit indefinitem semantischen Wert ohne Identifikator:

$$E \vdash \text{Ausdruck} \rightarrow \langle \text{Konstruktor}_1\ v_{11} \ldots v_{1k_1}, \ldots, \ll \underline{\lambda}_1 \ldots, E_1 \gg, \ldots \rangle$$

$$E \vdash \text{Ausdruck'} \rightarrow v'$$

$$E \oplus E_i, \ll \text{Ausdruck'}, E \gg \vdash \underline{\lambda}_i \ldots \rightarrow v_i'$$

$$E \vdash \underline{\text{Applikation}}\ (\text{Ausdruck., Ausdruck'}) \rightarrow \langle \text{Konstruktor}\ v_{11} \ldots v_{1k_1}\ v', \ldots, v_1', \ldots \rangle$$

Erläuterung: siehe Regel (16’)

(16') Applikation mit indefinitem semantischen Wert mit Identifikator:

$$\frac{E \vdash \text{Ausdruck} \rightarrow (< \text{Konstruktor}_1\ v_{11}\ldots v_{1k1}\ , \ldots\ , \ll \underline{\lambda_1}\ \ldots, E_1 \gg, \ldots >, i) \qquad E \vdash \text{Ausdruck'} \rightarrow v' \qquad E \oplus E_i, \ll \text{Ausdruck'}, E \gg \vdash \underline{\lambda_i}\ \ldots \rightarrow v_i'}{E \vdash \underline{\text{Applikation}}\ (\text{Ausdruck., Ausdruck'}) \rightarrow}$$

$$< \text{Konstruktor}\ v_{11}\ \ldots\ v_{1k1}\ v', \ldots, \text{validiereSemVal}(v_1', \{\ (i, \ll \underline{\lambda_1}\ \ldots, E_1 \gg)\ \}), \ldots >$$

Erläuterung: Regel (16) und (16') verfahren mit den einzelnen Alternativen des Wertes von Ausdruck jeweils so wie die „definiten" Regeln (3) und (4). Die jeweiligen Teilresultate werden in der letzten Zeile zu einem indefiniten semantischen Wert zusammengesetzt. Hat der semantische Wert v von Ausdruck einen Identifikator (Regel (16')), so wird analog zur Regel (15') verfahren: Jedes Teilresultat kommt unter der Voraussetzung zustande, daß als „wahrer Wert" von v eine von dessen Alternativen angenommen wird. Daher ist auf das Teilresultat validiereSemVal anzuwenden.

(17) Identifikator-Zuweisung:

$$\frac{E \vdash \text{Ausdruck} \rightarrow v \qquad E \vdash \text{Ausdruck'} \rightarrow v'}{E \vdash \underline{\text{verleiheIdentität}}(\text{Ausdruck, Ausdruck'}) \rightarrow \text{validiereSemVal}(\text{verleiheIdentität}(v, v'), \{\ \})}$$

Die Hilfsfunktion verleiheIdentität : SemVal x SemVal → SemVal sei folgendermaßen definiert[29]:

Definition 2.3.1.2.4:

verleiheIdentität ((v, i), i') → (v, i')
verleiheIdentität (v : v ≠(…,…), i') → (v, i')

[29] Es gibt übrigens keine Regel, die es verbietet, mit verleiheIdentität unterschiedlichen Or-Ausdrücken denselben Identifikator zuzuweisen. Ob es eine solche Regel geben sollte bzw. ob das Fehlen einer solchen Regel irgendeinen Nutzen mit sich bringt, ist unbekannt.

Erläuterung: siehe 2.3.1. Anmerkung: Da in v wiederum Werte mit Identifikator v' enthalten sein können, muß auf das Resultat von verleiheIdentität validiereSemVal angewandt werden. Diese Funktion verkürzt in diesem Anwendungsfall aus folgendem Grund alle indefiniten semantischen Werte in v, deren Identifikatoren gleich v' sind, zu < >: Ist in einem Element v_k von v ein indefiniter semantischer Wert v_k' mit Identifikator v' enthalten sein kann, so sind alle Elemente von v_k' ungültig, die ungleich v_k sind. Diese Bedingung ist immer erfüllt.

(17') Identifikator-Entfernung:

$$\frac{E \vdash \text{Ausdruck} \rightarrow (v, i)}{E \vdash \underline{\text{entferneIdentität}}(\text{Ausdruck}) \rightarrow v}$$

Die Regel (13*) wird durch folgende Regel ersetzt:

(13) Indefinites Programm:

$$\frac{\{\} \vdash \{ \text{Top-level-Bindung}_1, \ldots, \text{Top-level-Bindung}_n \} \rightarrow E \qquad E \vdash \text{Variable „main“} \rightarrow v}{\{\} \vdash \text{Top-level-Bindung}_1, \ldots, \text{Top-level-Bindung}_n \rightarrow \text{verflacheSemVal}(v)}$$

Die Hilfsfunktion verflacheSemVal : SemVal → SemVal dient dazu, einen beliebigen indefiniten semantischen Wert in ein „echtes“ (d.h. nicht ggfs. verschachteltes) Or-Set umzuformen. Dazu sind lediglich Verschachtelungen indefiniter Werte innerhalb von indefiniten Werten aufzulösen und definite Werte in einelementige Or-Sets umzuwandeln:

Definition 2.3.1.2.5:

verflacheSemVal ($< v_1, v_2, \ldots >$) $\rightarrow$ verflacheSemVal (v_1) $\cup$ verflacheSemVal (v_2) $\cup \ldots$
verflacheSemVal (($< v_1, v_2, \ldots >$, i)) $\rightarrow$ verflacheSemVal ($< v_1, v_2, \ldots >$)
(der Identifikator i kann wegfallen, denn verflacheSemVal wird erst in Regel (13) angewendet, d.h. wenn alle indefiniten semantischen Werte schon von den Regeln (15)-(17) validiert wurden.)
verflacheSemVal (v) | $v \neq < \ldots > \wedge v \neq \perp \rightarrow < v >$
verflacheSemVal (v) | sonst → v

2.3.2 Das leere Or-Set

Da man ein Or-Sets dazu verwendet, einen definite Wert durch eine Menge von Alternativen zu beschreiben, von denen der definite Wert eine ist, kann es ein „leeres Or-Set“ eigentlich nicht geben, denn es beschreibt keinen definiten Wert. Gerade diese „Unmöglichkeit“ können wir uns aber zunutze machen, indem wir das leere Or-Set < > als Resultat einer Funktion dazu verwenden, „unmögliche“ Funktions-Ergebnisse zu repräsentieren[30]. Am Ende von 2.3.2 ist ein Beispiel angegeben, ein weiteres Beispiel findet sich in 4.4.5.

2.3.2.1 Syntax

Zu der Grammatik der bisher definierten indefiniten Sprache wird folgende Produktion hinzugefügt:

Ausdruck → impossible

2.3.2.2 Dynamische Semantik

Zu den bisher für die indefinite Sprache vorhandenen Regeln werden folgende Regeln hinzugefügt:

Semantische Regeln 2.3.2.2.1 (zweite Erweiterung der definiten zur indefiniten Kernsprache)

(18) impossible:

$$\frac{}{E \vdash \text{impossible} \rightarrow \langle\,\rangle}$$

[30] Spekulativ: Fügt man zu einem Or-Set eine Alternative hinzu, so wird das Or-Set in gewissem Sinne „indefiniter“. So gesehen ist das Unmögliche von „maximaler Bestimmtheit“.

(19) β-Regel (Matching mit impossible):

$$\frac{v = < >}{E, v \vdash \underline{\lambda} \ldots \rightarrow \text{Ausdruck} \; ; \; _ \rightarrow \text{Ausdruck'} \Rightarrow < >}$$

Erläuterung: (ergibt sich aus (15))

Beispiel (in informeller, Haskell-ähnlicher Notation):

motor = or(Otto, Diesel)
sparsamkeit PKW Diesel = Hoch
sparsamkeit PKW Otto = Niedrig
sparsamkeit Motorrad Otto = Niedrig
sparsamkeit Motorrad Diesel = impossible

Somit evaluiert

sparsamkeit(Motorrad, motor)

zu „Niedrig“, d.h. es gibt keine sparsamen Motorräder.

2.3.3 Query-Variablen

So wie ein leeres Or-Set in gewissem Sinne ein Or-Set mit maximalem Informationsgehalt ist, wollen wir auch unbekannte Werte durch Or-Set-Generatoren beschreiben, über die wir bei der Programmerstellung keinerlei Information besitzen. Der Wert eines solchen Or-Set-Generators muß also in Form eines Or-Sets einen gesamten algebraischen Datentyp umfassen. Wir schreiben solche Or-Set-Generatoren als undefinierte Variable. Die Semantik einer solchen sogenannten *Query-Variablen* ist sehr einfach: Ihr indefiniter Wert umfaßt sämtliche Elemente ihres Typs[31].
Es ist in der deklarativen Semantik wie bei allen Or-Set-Generatoren möglich, Query-Variablen mit verleiheIdentität bzw. entferneIdentität jederzeit einen Identifikator zuzuweisen bzw. zu entfernen. Weil Query-Variablen syntaktisch wie normale Variablen eingesetzt werden, werden

[31] Den Typ ermittelt die statische Typinferenz aus dem Kontext der Variablen im Quellcode.

wir später die operationelle Semantik so definieren, daß 1.) Query-Variablen mit unterschiedlichem Namen auch unterschiedliche Identitäten erhalten, und das 2.) jeder Query-Variablen beim Eintritt der Evaluationsprozedur in den umgebenden Funktionsrumpf ein neuer Identifikator zugewiesen wird. Durch diese Maßnahmen verhalten sich Query-Variablen ähnlich wie lokale Variablen, was ihre intuitive Verwendung erleichtert (vgl. 4.4.3). Es sei an dieser Stelle schon vorweggenommen, daß die operationelle Semantik für Query-Variablen gegenüber der deklarativen Semantik deutlich leistungsschwächer ist, aber auch um ein Vielfaches interessanter. Wir werden nämlich Query-Variablen nicht einfach als syntaktische „Abkürzung“ für entsprechende Or-Ausdrücke verwenden (was nicht nur bei Query-Variablen eines Typs mit einer unendlichen Wertebereich zu Schwierigkeiten führen würde…), sondern wollen sie zur Laufzeit des Programms als Unbekannte betrachten und durch eine Technik ähnlich des aus der funktional-logischen Programmierung bekannten *Narrowing* schrittweise Informationen über ihren definiten Wert sammeln (siehe 4.6.3 und 4.6.4).

2.3.3.1 Syntax

Die bisherige Grammatik wird um folgende Produktion erweitert:

Ausdruck → $\underline{?}\text{Variable}_{\text{Typ}}$

Das Fragezeichen dient der Unterscheidung von normalen (d.h. vor der Verwendung gebundenen) Variablen. Query-Variablen sollen ansonsten in allen syntaktischen Aspekten als Variablen gelten.

2.3.3.2 Dynamische Semantik

Es wird folgende Regel hinzugefügt:

Semantische Regel 2.3.3.2.1 (dritte Erweiterung der definiten zur indefiniten Kernsprache)

(20) Query-Variablen:

$$\frac{}{E \vdash \underline{?}\text{Variable}_{\text{Typ}} \rightarrow \langle\, v : v \in \text{Typ} \,\rangle}$$

3 Baum-Grammatiken und Zuordnungslisten

Wir haben bisher zwei Repräsentationsformen für Or-Sets kennengelernt: Die Angabe eines Or-Sets in Mengenschreibweise und die Angabe einer Funktion unserer indefiniten Kernsprache, deren semantischer Wert ein Or-Set ist. Wir werden nun eine dritte Sichtweise auf Or-Sets einführen, nämlich die als *formale Sprachen*, zusammen mit ihren Definitionsvorschriften *Grammatik* und *Automat*. Es handelt sich hierbei um besondere Sprachen, die sogenannten *Baum-Sprachen*, deren Elemente die Grundterme eines algebraischen Datentyps sind. Der Vorteil dieser neuen Sichtweise liegt darin, daß für Grammatiken und Automaten bereits eine reichhaltige Theorie und vielfältige Anwendungen existieren. Beispielsweise wurde in [18] dargelegt, wie sich indefinite *Datenbankzustände* durch bestimmte Baum-Automaten repräsentieren lassen. Außerdem bieten Baum-Grammatiken eine Möglichkeit zur textuellen Darstellung ggfs. unendlicher Or-Sets als *Berechnungsresultat*. Wir werden zeigen, wie sich aus einem Ausdruck der Kernsprache eine sogenannte *Zuordnungsliste* berechnen läßt und setzen Grammatik in Beziehung zu solchen Zuordnungslisten. Wir definieren die Evaluierung von Programmen zu Zuordnungslisten als die *operationelle Semantik* von Kernsprachenausdrücken.

3.1 Reguläre Baum-Grammatiken

Wie wir in 2.1.2.3 gesehen haben, können Grundterme können als (*Werte-)Bäume* betrachtet werden. Im folgenden werden wir nur endliche Grundterme (d.h. endlich tiefe Werte-Bäume) betrachten.[32]

So wie man Sprachen von Zeichenketten durch kontextfreie Grammatiken definieren kann, so ist dies auch für Wälder möglich:

[32] Genau genommen müßte man noch zwischen syntaktischen und semantischen Konstruktor-Applikationen unterscheiden. Dieser Unterschied ist aber für die folgenden Ausführungen irrelevant, da nur Grundterme betrachtet werden. Außerdem nehmen wir an, daß jedem Konstruktor durch die statische Semantik eine Arität zugewiesen wurdeund jeder Konstruktor im Term mit einer Anzahl von typkorrekten Argumenten gemäß seiner Arität appliziert wird.

Definition 3.1.1

Eine *reguläre Baumgrammatik* ist ein Tupel (N, P, N_S), wobei gilt:

- N ist eine Menge von Nichtterminalen
- P ist eine Menge von k Produktionen der Form $n_i \rightarrow K_i\ a_{i1} \ldots a_{isi}$ oder $n_i \rightarrow n_j$

  wobei $1 \leq i \leq k$. K_i ist ein Konstruktor der Arität s_i ($s_i \geq 0$), die a_{ij} sind rechte Seiten von Produktionen. Die n_i und n_j sind Nichtterminale, wobei jedes Nichtterminal auf einer rechten Seite auch mindestens einmal auf einer linken Seite vorkommen muß. Die Menge aller rechten Seiten einer regulären Baum-Grammatiken G kürzen wir mit RS_G ab.
- N_s ist ein Nichtterminal, genannt *Startsymbol* oder *Axiom*. Es muß mindestens einmal als linke Seite in P vorkommen.

(Als *Signatur* sei zu jeder Baum-Grammatik implizit die Menge aller Konstruktoren und deren Aritäten gegeben.)

(nach [4], S. 67 und [18])

Definitionen 3.1.2

Für eine Grammatik $G = (N, P, N_S)$ seien p und q gegeben mit $p, q \in RS_G$. Wir schreiben $p =>_G q$ ((grammatikalische) *Herleitung* von p aus q), falls ein $a \in N$ innerhalb von p und ein $r \in RS_G$ existieren, so daß $q = p[r/a]$.

Wir schreiben $p =>^*_G q$, falls gilt $p = q$ oder es eine Herleitungskette $p =>_G p_1 =>_G \ldots =>_G p_{n-1} =>_G q$ $(n \geq 1)$ gibt.

Der durch eine reguläre Baumgrammatik $G = (N, P, N_S)$ *generierte* Wald

$L = \{ t \in SemVal : N_S =>^*_G t \}$ heißt *reguläre Baumsprache*.

Zwei reguläre Baum-Grammatiken, die dieselbe reguläre Baumsprache generieren, heißen *äquivalent*.

Eine reguläre Baum-Grammatik stellt eine spezielle Form einer *kontextfreien Grammatik* dar (vgl. [4], S. 69). Viele Sätze über solche Grammatiken gelten daher (ggfs. leicht angepaßt) auch für Baum-Grammatiken (siehe z.B. [4] und [18]).

Weil es in dieser Arbeit nur um reguläre *Baum*-Grammatiken und –Sprachen geht, werden wir diese einfach als reguläre *Grammatiken* bzw. reguläre *Sprachen* bezeichnen.

Für reguläre Grammatiken gibt es eine Normalform, die für uns wegen ihrer Einfachheit nützlich sein wird[33] (nach [4], S. 69ff):

Definition 3.1.3

Eine reguläre Baumgrammatik G = (N, P, N_S) ist in *(Grammatik-)Normalform*, wenn jede Produktion in P eine der folgenden Formen hat:

(i) $n \rightarrow K$

(ii) $n \rightarrow K\ n_1 \ldots n_k$ $(k \geq 1)$

(n und die n_i sind Nichtterminale, K ist ein Konstruktor)

Satz 3.1.4

Jede reguläre Baumgrammatik kann in eine äquivalente reguläre Baumgrammatik in Grammatik-Normalform umgeformt werden[34].

Beweis: siehe [4], S. 70

3.2 Reguläre Funktionen und Zuordnungslisten

Wir wollen darstellen, wie Funktionen der indefiniten Kernsprache verwendet werden können, um reguläre Sprachen in Form von Or-Sets zu generieren. Or-Sets, die mit einer regulären Sprache identisch sind, nennen wir ebenfalls *regulär*.

Definition 3.2.1

Eine Gruppe von n, n≥1 First-order-Bindungen der Form $nt_i = \underline{or}(r_{i1}, \underline{or}(\ldots, r_{ini}))$, wobei die nt_i untereinander verschiedene Variablen sind, und die r_{ij} Terme, in denen als Variablen nur die nt_i vorkommen, nennen wir eine *Zuordnungsliste*. nt_i heißt die *linke Seite* der *Zuordnung* i und $\underline{or}(r_{i1}, \underline{or}(\ldots, r_{ini}))$ entsprechend die *rechte Seite*. Die Variablen nt_i nennen wir Nichtterminale.

---

[33] Ansonsten ist sie zur Darstellung einer Sprache eher ungünstig, da sie sich aus relativ vielen sehr einfachen Produktionen zusammensetzt, was zu einer gewissen Unübersichtlichkeit führen kann.

[34] Man beachte, daß aus diesem Satz keineswegs die Existenz einer *eindeutigen* Normalform der Grammatik folgt.

Definition 3.2.2

Es sei eine beliebige reguläre Baumgrammatik G := ($n_{0..k}$, P, n_0), k≥1 gegeben, wobei P eine Produktionenmenge der folgenden Form ist und alle n_i verschieden sind:

$n_0 \rightarrow r_{01}$

…

$n_0 \rightarrow r_{0m0}$

…

$n_{k-1} \rightarrow r_{(k-1)1}$

…

$n_{k-1} \rightarrow r_{km(k-1)}$

sowie folgende Zuordnungsliste G_F:

$N_0 = \underline{or}(R_{01}, \underline{or}(\ldots, R_{0m0}))$

…

$N_{k-1} = \underline{or}(R_{(k-1)1}, \underline{or}(\ldots, R_{km(k-1)}))$

Für G_F seien die R_{ij} Ausdrücke, die den jeweiligen rechten Seiten r_{ij} der Produktionen in P entsprechen, d.h. bei denen die Vorkommen der Nichtterminale n_i durch entsprechende Variablen N_i ersetzt werden[35].

Dann heißt GF die *korrespondierende Zuordnungsliste* zu G und G die *korrespondierende Grammatik* zu GF. Es ist aus der semantischen Regel (14) leicht zu ersehen, daß semval(n_0) = semval(N_0).

Man beachte, daß alle Or-Ausdrücke Unbestimmte Or-Sets sind, da die Funktion verleiheIdentität nicht vorkommt.[36]

Definition 3.2.3

N_0 heißt *generierende Funktion* der Sprache von G.

Definition 3.2.4

Eine Funktion der Kernsprache, die eine generierende Funktion der Sprache einer regulären Grammatik ist, heißt *reguläre Funktion*.

---

[35] Annahme o.B.d.A.: G sei typkorrekt in dem Sinne, daß G_F gemäß Annahme in 2.1 typkorrekt ist.

[36] Später in Prolid-Programmen müßte man (in Infix-Notation) a `or*` b statt or(a, b) schreiben, vgl. 4.4.3

3.3 Generierung von Zuordnungslisten

Wir haben gesehen, wie man reguläre Grammatiken durch reguläre Funktionen darstellen kann. Nun wollen wir umgekehrt das (möglicherweise unendliche) Berechnungsresultat einer Funktion der Kernsprache als reguläre Grammatik in Gestalt einer sogenannten *Zuordnungsliste* darstellen. Dies ist in erster Linie deshalb interessant, weil man auf diese Weise berechnete unendliche Or-Sets „als ganzes“ darstellen kann.

Zwar ist es dank der Lazy-Auswertung auch mit einer definiten Sprache wie z.B. Haskell möglich, terminierende Programme anzugeben, bei deren Auswertung unendliche Datenstrukturen auftreten, es gibt aber keine Möglichkeit, unendliche Evaluations-Ergebnisse mit Hilfe *endlicher* Ausdrücke (die dann natürlich keine Grundterme mehr sein können) als ganzes darzustellen[37].

Prinzipiell kann solch ein endlicher Ausdruck nur eine Umschreibung des eigentlichen Resultats sein. Eine solche Umschreibung kann auf verschiedene Weise erfolgen; z.B. dadurch, daß man Teile des Resultats unausgewertet läßt, wenn deren Auswertung nicht terminieren würde (sogenannte *partial evaluation,* vgl. z.B. [13]) oder durch Angabe von Eigenschaften, die das Resultat charakterisieren (z.B. durch Logik-Formeln). Die Erkennung der Nicht-Terminierung kann entweder statisch erfolgen (z.B. durch *abstract interpretation,* vgl. z.B. [3]), durch Generierung eines *residual program* mittels Teil-Interpretation für ausgewählte Eingabedaten (vgl. z.B. [13]) oder für beliebige Eingabedaten zur Laufzeit[38] (siehe unten). Eine andere Möglichkeit ist, (potentiell) unendliche Datenstrukturen schon im Quellcode zu identifizieren und auf eine Auswertung solcher Datenstrukturen von vornherein zu verzichten. Dies geschieht in der *symbolischen Programmierung* durch die Verwendung *unausgewerteter Symbole* (Anwendungen gibt es hierfür insbesondere in der Computer-Algebra).

Die im Rahmen dieser Arbeit entwickelte Programmiersprache Prolid verfügt über eine *Rekursionserkennung* zur Laufzeit und verwendet (ggfs. rekursive) *Zuordnungslisten* (eine Form von Deklarationsgruppen) zur Darstellung des Resultats. Im folgenden wollen wir die Grundlagen der Rekursionserkennung für die bereits bekannte indefinite Kernsprache von Prolid entwickeln und Zuordnungslisten in Beziehung zu regulären Baum-Grammatiken setzen.

Zur Erkennung von Nicht-Terminierung durch Rekursionserkennung sind bereits verschiedene Techniken bekannt. Eine Möglichkeit für funktionale Programmiersprachen besteht darin, bei jedem Funktionsaufruf zu prüfen, ob bei einem einschließenden Aufruf der Funktion schon einmal dieselben Argumente verwendet wurden (was übrigens problematisch für Higher-order-Funktionen ist, da hier die Argumente Funktionen sein können). Man prüft also, ob ein bestimmter Auswertungs-*Status* wiederholt eintritt. Diese Technik wurde vielfach verfeinert (vgl.

---

[37] …ausgenommen natürlich durch *Programmierung* dieser Fähigkeit in der jeweiligen Sprache selbst.

[38] Falls möglich (das Halteproblem ist bekanntlich nicht entscheidbar).

z.B. [13]). Ein anderer Ansatz besteht darin, Programme so umzuformen, daß zur Laufzeit eine leistungsfähige Abhängigkeitsanalyse (zur Suche nach Ausdrücken, deren Wert mittels Rekursion von „sich selbst“ abhängt) möglich wird (z.B. durch Finden eines sogenannten *sequential algorithm* zu dem jeweiligen Programm, vgl. [12]). Die meisten Implementationen solcher *loop detectors* werten „unendliche Ausdrücke“ nach erfolgreicher Detektion zu $\perp$ aus.

Was wir benötigen, geht aber über die bloße Rekursions-*Erkennung* mit anschließender Terminierung hinaus; wir wollen für ein nicht-terminierendes Programm zusätzlich ein Auswertungsresultat erhalten, das alle Informationen über das unendliche Resultat enthält und trotzdem keine unausgewerteten Funktionsapplikationen mehr beinhaltet. Eine solches Ergebnis wird bei uns die Form einer sogenannten *Zuordnungsliste* haben und mit einer Baumgrammatik korrespondieren.

Wir definieren nun zuerst eines der wichtigsten Hilfsmittel zur sogenannten *Normalen Auswertung* (*lazy evaluation*, *call-by-name*): die sogenannte *Kopf-Normalform* (vgl. z.B. [15, 16]). Die Kopf-Normalform ist (informell ausgedrückt) ein Ausdruck mit *konstantem Kopf*, d.h. einem Kopf, der bei der Auswertung unverändert bleibt.

Definition 3.3.1

Eine *indefinite Kopf-Normalform* ist eine Menge $\{ (K_1, a_{11}, \ldots, a_{1k1}) , \ldots, (K_n, a_{n1}, \ldots, a_{nkn}) \}$ von Tupeln, wobei die K_i verschiedene Konstruktoren und die a_{ij} beliebige Ausdrücke der Kernsprache sind. Die einzelnen Elemente $hnf_{1..k}$ der indefiniten Kopf-Normalform heißen *Kopf-Normalformen*[39], die a_{ij} heißen die *Argumente* der jeweiligen Kopf-Normalform hnf_i. Der semantische Wert semval[40](hnfs) einer Kopf-Normalform hnfs wird definiert als

semval(or(or(Applikation(Applikation(K_1, a_1),…a_n),…., Applikation(Applikation(K_n, a_{n1}), … a_{nn}))) für $n > 0$ und als $< >$ für $n = 0$. Ein Ausdruck sei in *Normalform*, wenn er in Kopf-Normalform ist und die Argumente der Kopf-Normalform in Normalform sind.

---

[39] Unsere Definition baut auf dem bekannten Begriff der *head normal form* für λ-Abstraktionen auf, wobei wir allerdings durchaus zulassen, daß die Argumente indefinite semantische Werte haben!

[40] Die Funktion semval ist eigentlich für einzelne Ausdrücke definiert, der Einfachheit halber verwenden wir aber hier denselben Bezeichner für indefinite Kopf-Normalformen.

Definition 3.3.2

Eine indefinite Kopf-Normalform HNfs(a) *zu einem Ausdruck* a mit semval(a) = < v_i > ist eine indefinite Kopf-Normalform { (K_1, a_{11}, …,a_{1k1}) , …, (K_n, a_{n1}, …,a_{nkn}) }, für die gilt:

$\forall i, v_i \in \text{semval}(a)\ \exists k, 1 \le k \le n : \text{hd}(v_i) = K_k \wedge (\forall j, 1 \le j \le k_n : \text{arg}_j(v_i) \in \text{semval}(a_{kj}))$.

(hd und arg_i sind die sogenannten *Selektorfunktionen* für (curried-notierte) Applikationen:

$\text{hd}(e_0 \ldots e_{n-1}) := e_0$ und $\text{arg}_i(e_1 \ldots e_n) := e_i$.)

Man beachte, daß aus 3.3.2 weder die Existenz noch die Eindeutigkeit „der“ indefiniten Kopf-Normalform zu einem Ausdruck folgt!

Beispiel: Der Ausdruck impossible hat genau eine indefinite Kopf-Normalform, nämlich { ( ) } (mit dem semantischen Wert < >).

Wir können nun den semantischen Wert eines Ausdrucks dadurch berechnen, daß wir ihn zuerst zu einer indefiniten Kopf-Normalform evaluieren[41], und dann rekursiv die semantischen Werte der Argumente der einzelnen Argumente der einzelnen Kopf-Normalformen evaluieren. Die Resultate für die einzelnen Kopf-Normalformen in der indefiniten Kopf-Normalform fassen wir zu einem Or-Set zusammen.

Die Idee ist nun, jeder indefiniten Kopf-Normalform eine Zuordnung in einer Zuordnungsliste zuzuordnen. Dabei ersetzen wir alle Argumente der einzelnen Kopf-Normalformen durch neue Nichtterminale (d.h. hier Variablen nt_i), für die wir dann wiederum Zuordnungen durch Auswertung zu indefiniten Kopf-Normalformenen berechnen usw.. Auf diese Weise konstruieren wir eine Zuordnungsliste, die mit einer regulären Grammatik in Normalform korrespondiert[42].

Wir geben nun einen einfachen Algorithmus an, der zu einem Ausdruck der Kernsprache eine Zuordnungsliste berechnet.

---

[41] wie das geschieht, spielt hier noch keine Rolle; wir werden in 4.7.2 einen Algorithmus angeben.

[42] Zur Erinnerung: Wir haben unendlich tiefe Grundterme ausgeschlossen

Verfahren 3.3.3 (Berechnung einer Zuordnungsliste p zu einem Ausdruck x)
(imperativ)
Anmerkung: Wir benutzen Ausdrücke zur Benennung von Nichtterminalen: $nt_A \neq nt_B \Leftrightarrow A \neq B$ (mit A, B ∈ Ausdruck). gedächtnis sei eine Menge von Ausdrücken.

1. el := (x), p := { }, gedächtnis := { }
2. e sei das erste Element von el. Entferne e aus el.
3. falls e ∈ gedächtnis: gehe zu 8.
4. { $(K_1, a_{11}, \ldots, a_{1k1})$, …, $(K_n, a_{n1}, \ldots, a_{nkn})$ } := HNfs(e)
   (falls HNfs(e) nicht existiert, terminiere mit ⊥)
5. p := p ∪ { $nt_e \equiv$ generateOrSet({ $K_1\ nt_{a11} \ldots nt_{a1k1}, \ldots, K_n\ nt_{an1} \ldots nt_{ankn}$ }) }
6. gedächntis := gedächtnis ∪{e}
7. el := $(el_1, \ldots, el_{|el|}, \ldots, a_{11}, \ldots, a_{1k1}, \ldots, a_{n1}, \ldots, a_{nkn})$
8. falls el = ( ) : fertig, andernfalls gehe zu 2.

Hilfsfunktion generateOrSet: generateOrSet ($\{alt_1, \ldots, alt_n\}$) → or (alt_1, or(…, alt_n))

Satz 3.3.4
Wenn das Verfahren 3.3.3 für einen Ausdruck x terminiert, und semval(x) nicht < >, ⊥ oder eine Closure ist, so ist p eine Zuordnungsliste korrespondierend mit einer regulären Grammatik in Normalform.

Beweis: (Trivial; lt. Regel 5 hat jede rechte Seite in p die Form der rechten Seite einer Zuordnung gemäß Definition 3.2.1.)

Als Faustregel kann zu einer Funktion zumindest dann eine reguläre Grammatik angegeben werden, wenn Berechnung ihrer Kopf-Normalform und rekursiv die von deren Argumenten usw. terminiert. Dies ist dann der Fall, wenn bei der Auswertung der Funktion keine Linksrekursion auftritt (eine Kopf-Normalform korrespondiert mit einer rechtlinearen Produktion).
In 4.7.2 ist das Verfahren 3.3.3 in einer verbesserten, aber auch komplizierteren Version als *operationelle Semantik* der Sprache Prolid detailliert angegeben und zur implementierten Fassung in Beziehung gesetzt. Insbesondere ist dort die Berechnung von Kopf-Normalformen genau dargestellt. Der wesentliche Unterschied zwischen dem Verfahren 3.3.3 und dem in 4.7.2 geschilderten Algorithmus besteht darin, daß der Algorithmus in 4.7.2 zusätzlich auch für viele linksrekursive Ausdrücke terminiert, und auch für unendliche Grundterme definiert ist (und auch dafür oft terminiert). Das Verfahren 3.3.3 hat dagegen den Vorzug, daß es die Ähnlichkeit von

indefiniten Kopf-Normalformen und rechten Seiten von Zuordnungen bzw. Grammatiken in Normalform direkt algorithmisch umsetzt.

4 Die Programmiersprache Prolid

Prolid („Programming Language for Indefinite Data“) ist eine applikative Programmiersprache zur Verarbeitung indefiniter Daten. Ein Interpreter für Prolid-Programme wurde prototypisch als Haskell – Programm implementiert. Der „konventionelle Teil“ der Sprache entspricht weitgehend Haskell 1.2, wobei allerdings Typ-Klassen und Monaden (als zu umfangreich und unwesentlich für diese Arbeit) nicht in den Sprachumfang von Prolid aufgenommen wurden. Weil aber zur Implementierung der Darstellungsfunktionalität Ad-hoc-Overloading unverzichtbar ist, wurde eine von Haskell abweichende einfache Form der statischen Auflösung von Überladung entwickelt. Einige andere Elemente von Haskell, wie z.B. das Modulsystem, wurden in stark vereinfachter Form übernommen. Salopp ausgedrückt, verhält sich Prolid zu Haskell ungefähr wie die früheste Version von Java zu ANSI-C++.

Prolid ist eine rein funktionale Sprache ohne imperative Elemente. Die Auswertungsstrategie ist call-by-name (die sogenannte „normale“ Auswertungsreihenfolge). Die wichtigsten von Haskell übernommenen Spracheigenschaften des konventionellen („definiten“) Sprachteils sind: durch den Anwender definierbare, parametrisierbare algebraische Datentypen, statische Typinferenz, Higher-order-Funktionen und Pattern-Matching. Die Möglichkeiten zur Verarbeitung indefiniter Daten werden in Gestalt von Or-Set-Generatoren zur Verfügung gestellt, wie sie in 2.3 eingeführt wurden. Zur graphischen Darstellung indefiniter Daten sind keine speziellen Sprachelemente nötig, wenn man einmal von der allgemeinen Fähigkeit zur statischen Auflösung von Funktions-Überladung absieht (siehe oben). Vielmehr werden graphische Darstellungen durch einen besonderen Datentyp repräsentiert, dessen Werte durch ein gesondertes (Java-)Programm, den *Interactor*, objektorientiert dargestellt werden. Diese Fähigkeiten sind im Kapitel 5 beschrieben.

Folgende Sprachbeschreibung deckt alle Sprachelemente ab, konzentriert sich jedoch auf die Or-Set-Generatoren. Für weitergehende Aspekte des konventionellen Sprachanteils wird (soweit es keinen Bezug zur Verarbeitung indefiniter Daten gibt) auf den *Haskell-Report* [17] verwiesen.

Die Sprachbeschreibung basiert teilweise auf dem Haskell-Report. Darauf wird im weiteren Verlauf der Darstellung nicht mehr hingewiesen. Die statische Semantik von Prolid orientiert sich an der in Kapitel 2 beschriebenen Semantik der indefiniten Kernsprache, wobei die implementierte operationelle Semantik Vereinfachung erfahren mußte. Die Unterschiede zwischen deklarativer und operationeller Semantik werden bei der Beschreibung der einzelnen Or-Set-Generatoren unter 4.4.3 erläutert. Die formale operationelle Semantik ist unter 4.7 angegeben. Bei der vorangehenden Einführung der einzelnen Sprachelemente ist aber an wichtigen Stellen bereits eine informelle Beschreibung der Semantik zu finden. Außerdem werden ausgewählte Aspekte zur Implementierung der einzelnen Sprachelemente erläutert[43]. Die

[43] Für Details zur Implementierung muß jedoch auf die Kommentierung im Quellcode verwiesen werden.

Beschreibung der anwendungstechnischen Aspekte des Interpreters (wie z.B. Erstellung, Aufrufparameter, Kommandoschnittstelle o.ä.) findet sich im Anhang A.
Bei der Implementierung wurde auf eine gewisse Skalierbarkeit der Spracheigenschaften geachtet. Z.B. sind die Schnittstellen für ein „echtes“ Modulsystems bereits im Quellcode vorhanden. Dadurch sollte es möglich sein, die Fähigkeiten von Prolid in vielerlei Hinsicht auszuweiten, ohne eine komplett neue Implementierung durchführen zu müssen.

Die Notation für die Darstellung der Syntax[44] entspricht der des Haskell-Reports (vgl. §2.1) und ist eine leicht abgewandelte BNF. Das Verhältnis zur abstrakten Syntax der Kernsprache (vgl. 2.1.1) ist unter 4.7.1 beschrieben.

Notation 4.1:

[muster]	optionales reguläres Muster
{muster}	kein oder mehrfaches Vorkommen
($muster_1$ $muster_2$)	Konkatenation
$muster_1$ | $muster_2$	Alternative
$muster_{1(muster2)}$	Wörter produziert von $muster_1$ ohne die von $muster_2$ produzierten
`terminal`	Terminal
$nonterminal^{(p, a)}$	Nichtterminal mit Assoziativität a und Vorrang p (vgl. 4.6). Enthält p oder a eine Variable, so steht diese Notation für *alle* möglichen Produktionen mit Werten für jede Variable in p von 0 bis 9 bzw. l (links-assoziativ), r (rechts-assoziativ), oder n (nicht assoziativ) für Variablen in a.
$nonterminal^{p}$	Nichtterminal mit Vorrang p (siehe oben).

[44] auch auf lexikalischer Ebene!

4.1 Layout

Prolid-Quellcode wird in dieser Arbeit unter Verwendung von *Layout-Listen* dargestellt. Dies sind Quellcode-Abschnitte, die in geschweiften Klammern gewisse Elemente, unterteilt durch Semikolons, zusammenfassen. Jede Layout-Liste ist mit einem der reservierten Bezeichner `where`, `let` und `of` assoziiert[45]. Alternativ dürfen im Quellcode unter Beachtung eines besonderen *Layouts* (auch *Abseitsregel* genannt) die Klammern und Semikolons weggelassen werden. Das Layout stellt dann die Eindeutigkeit der Syntax sicher.
Die Layout-Grundregel lautet: Fehlt nach einem der genannten Bezeichner die öffnende geschweifte Klammer, so wird die Spalte des folgenden Zeichens gemerkt und (intern) eine neue Layout-Liste erzeugt; steht nun das erste Zeichen (ohne Berücksichtigung von vorangehendem Leerraum) einer Zeile in dieser Spalte, so beginnt ein neues Element der Liste. Diese Zeile sowie folgende Zeilen, deren erstes Zeichen (wieder ohne Berücksichtigung von Leerraum) weiter rechts steht, werden diesem Element zugerechnet. Beginnt eine Zeile links von dieser Spalte, so wird davor die Layout-Liste abgeschlossen. Näheres zum Layout ist unter §1.5 im Haskell-Report zu finden.

Anmerkungen zur Implementierung:
Implementiert wurde die Erzeugung von Layout-Listen aus dem Layout mit der Funktion `Prolid.dolayout`, die vor der lexikalischen Analysephase auf den Quellcode angewandt wird.

4.2 Lexikalische Struktur

Der Quellcode eines Prolid-Programms setzt sich aus ASCII-Zeichen zusammen, Unicode wird nicht unterstützt. Es existieren außerhalb von Strings (vgl. 4.2.2) zwei Kommentar-Formen: Die Zeichenfolge `--` bewirkt, daß alle folgenden Zeichen bis zum Zeilenende ignoriert werden. Außerdem werden alle Zeichen zwischen korrespondierenden Zeichenfolgen `{-` und `-}` ignoriert, wobei `{-` und `-}` auch verschachtelt werden dürfen.

Anmerkungen zur Implementierung:
Für die lexikalische Analyse ist das Modul `ProlidLexer` zuständig, daß vom Lexer-Generator *Alex* aus der Lexer-Spezifikationsdatei `Prolid.x` generiert wurde (siehe Anhang A). Der Tokengenerierung durch `ProlidLexer` geht noch eine Phase voraus, in der sogenannte

[45] Die Bedeutung dieser Schlüsselwörter spielt an dieser Stelle noch keine Rolle

Literate Scripts in Prolid-Quellcode umgewandelt und in den Quellcode Zeilennummern und Dateinamen explizit eingefügt werden (um später bei Syntaxfehlern den Ort des Fehlers anzeigen zu können). Literate Scripts sind Quell-Dateien, in denen die Rollen von Kommentaren und eigentlichem Programmcode vertauscht sind: Kommentare haben kein besonderes syntaktisches Merkmal, sondern kommen als normaler Text vor. Dagegen muß Programmcode durch das Präfix „> “ eingerückt und durch Leerzeilen vom übrigen Text abgesetzt werden. Mehr dazu ist z.B. in [17] zu finden. Literate scripts besitzen die Dateiendung `.lpro`, während für normale Prolid-Quellen `.pro` zu verwenden ist (vgl. 4.5.1). Außerdem besteht die Möglichkeit, Quellcode-Module in Form von Haskell-Datenstrukturen (gewissermaßen „compiliert“) zu speichern, siehe Anhang A. LaTeX-Literate Scripts sind nicht erlaubt.

Anmerkung: Da in 4.2 alle Produktionen sich auf die lexikalische Struktur beziehen, sind Leerräume (whitespace) ausnahmsweise explizit angegeben.

4.2.1 Bezeichner und Operatoren

smallId	→	(smallLetter { smallLetter | largeLetter | digit | ‘ | _ })$_{(reservedId)}$	
queryId	→	? smallId	
largeId	→	largeLetter { smallLetter | largeLetter | digit | ‘ | _ }	
orId	→	orIdWithIndex | orIdWithoutIndex	
orIdWithIndex	→	`or` [int]	(fehlt hier der Index, wird er vom Lexer statisch zugewiesen, siehe 4.4.3)
orIdWithoutIndex	→	`or*`	
reservedId	→	`case` | `data` | `else` | `if` | `import` | `in` | `infix` | `infixl` | `infixr` | `let` | `module` | `of` | `then` | `type` | `where` | `or` | `impossible` | `nt_`{ smallId | digit }	
symbol	→	`!` | `#` | `$` | `%` | `&` | `*` | `+` | `.` | `/` | `<` | `=` | `>` | `@` | `\` | `^` | `|` | `-` | `~`	

Es gibt jeweils einen Namensraum für Variablen (dazu zählen auch die Funktionen), für Konstruktoren, für Typvariablen, für Typkonstruktoren und für Module. Lexikalisch werden diese Namen von bestimmten *Bezeichnern* und *Operatoren* repräsentiert. Bezeichner unterteilen sich in großgeschriebene (largeId; speziell für Modulnamen (moduleId), (Daten.-)Konstruktoren (conId) und Typkonstruktoren (typeConId)), kleingeschriebene (smallId; speziell für Variablen (varId) und Typvariablen (typeVarId)) sowie reservierte Schlüsselwörter (reservedId; z.B. `let`) und solche für Query-Variablen (z.B. `?x`). Operatoren sind entweder gewisse Or-Set-Generatoren (orId) oder bestehen aus nicht-alphanumerischen Zeichenfolgen (symbol-Folgen) und bezeichnen bestimmte reservierte Terminale mit Infix-Syntax (reservedOp und orId, z.B. `->` und `or*`), Konstruktoren (conOp; falls sie mit dem Zeichen : beginnen) oder Variablen (varOp; andernfalls). Sie werden in Infix-Syntax verwendet (ausgenommen ist der Präfix-Operator – (Negation)), wobei für conOp und varOp mittels sogenannter *Sections* auch Präfix-Syntax erlaubt ist (siehe 4.4.1). Die genannten und im folgenden genauer spezifizierten Unterschiede zwischen den Bezeichnern und Operatoren reichen aus, um alle Namensräume vollständig zu trennen.

4.2.2 Zahlen, Zeichen und Strings

Prolid kennt nur dezimal repräsentierte, vorzeichenbehaftete ganze Zahlen mit maschinenabhängiger Breite (aber mindestens 29 Bit) vom Typ `Int` (entsprechend Haskell). Negative Zahlen werden mit Hilfe des Negationsoperators `-` erzeugt. Dies ist der einzige Präfix-Operator in Prolid (vgl. 4.4.1). Er wird intern in die Funktion `negate` umgewandelt, die im Modul `Stdlib` definiert ist[46]. Sein Vorrang ist geringer als der aller anderen Operatoren; er kann nicht mit einer Fixity-Deklaration beeinflußt werden. Man beachte, daß im Unterschied dazu der Vorrang des Negationsoperators unter Haskell immer dem des Subtraktions-Operators entspricht.

int → digit { digit }

Zeichen-Literale (*Characters*) werden in einfache Anführungszeichen (‘) gesetzt. Jedes Zeichen hat einen Code von 0 bis 255. Codes von 0 bis 127 haben eine Repräsentation nach ASCII, die Repräsentation von Zeichen mit Code 128 bis 255 ist systemabhängig (unter Windows™ z.B. nach der ANSI-Norm). Nicht-ASCII Zeichen sowie nicht direkt repräsentierbare Zeichen (wie z.B. der Zeilenvorschub) und das Zeichen ‘ werden durch die üblichen Escape-Sequenzen

[46] `Stdlib.pro` ist die Prelude von Prolid

dargestellt (`\a` : das Zeichen mit ASCII-Code 7 (BEL), `\b` : 8 (BS), `\f` : 12 (FF), `\n` : 10 (LF), `\r` : 13 (CR), `\t` : 9 (TAB), `\v` : 11 (VT)).
Strings sind Zeichen-Sequenzen innerhalb doppelter Anführungszeichen (“). Hier muß statt ‘ das Zeichen “ als Escape-Sequenz geschrieben werden. Sogenannte *string gaps* (Unterbrechungen innerhalb eines Strings durch Leerräume) sind nicht erlaubt. Unicode wird nicht unterstützt.

char	→	‘ graphic$_{(‘ \mid \backslash)}$ | escape) ’
string	→	“ { graphic$_{(“ \mid \backslash)}$ | escape) } “
graphic	→	*Zeichen mit ASCII-Code von 32 bis 126*
escape	→	\ (charEscape | decimal)
charEscape	→	`a | b | f | n | r | t | v | \ | “ | ‘`

decimal ist hier die längste auf \ folgende Ziffernsequenz. Hat sie einen größeren Wert als 255, so wird als Zeichencode dieser Wert modulo 256 angenommen.

Anmerkung zur Implementation:
Zahlen, Zeichen und Strings gelten beim Parsen implizit als Konstruktoren vom Typ `Int`, `Char` bzw. `[Char]`; entsprechende `data`-Deklarationen (vgl. 4.3.1) sind nicht anzugeben.

4.3 Datentypen

Prolid führt eine statische Typisierung aller Funktionsrümpfe gemäß dem polymorphen Hindley-Milner Typsystem durch (vgl. z.B. [14, 19]) und vergleicht ggfs. vom Programmierer vorgegebene Funktionssignaturen mit dem inferierten Funktionstypen. Zusätzlich können während der Typinferenz bestimmte Überladungen von Funktionsnamen aufgelöst werden (sogenannter *Ad-hoc-Polymorphismus*, siehe 4.5.3). Die in Haskell dazu vorgesehenen, komplizierteren und leistungsfähigeren Typ- und Konstruktor-Klassen gibt es nicht. Eine weitere Besonderheit gegenüber dem traditionellen Typ-Schema ergibt sich durch die Möglichkeit, undefinierte Variablen als Query-Variablen zu verwenden (siehe 4.4.3). Die statische Typinferenz hat die Aufgabe, Typfehler vor der Laufzeit zu entdecken und ggfs. Überladungen aufzulösen. Zur Laufzeit spielen Typen in Prolid keine Rolle mehr.
Basis des Typsystems sind sogenannte *algebraische Datentypen*, wie sie in 2.2 eingeführt wurden (kurz „Typen“). Ein Typ wird syntaktisch durch einen *Typausdruck* repräsentiert (die Begriffe „Typ“ und „Typausdruck“ werden oft synonym verwendet). Typausdrücke konstituieren sich aus *Typkonstruktoren* und *Typvariablen*. Eine Typvariable ist implizit allquantifiziert, d.h. sie repräsentiert *jeden* beliebigen Typ. Typvariablen können innerhalb von `data`- und `type`-

Deklarationen (siehe unten) sowie von Funktionssignaturen explizit angegeben werden (vgl. 4.5.3). Während der Typinferenz werden effektiv Typvariablen den Variablen als Typ zugeordnet. Im Kontext des Auftretens der Variablen in einem Ausdruck werden die korrespondierenden Typvariablen ggfs. mit weniger generellen Typausdrücken instanziiert. Konstruktoren können als undefinierte Funktionsnamen angesehen werden, deren Typ explizit bekannt ist. Die Deklaration eines Typkonstruktors dient effektiv dazu, interne Typ-Signaturen für Konstruktoren bereitzustellen.
Die statische Typinferenz ordnet jedem Ausdruck in jedem geladenen Modul den sogenannten *Prinzipal-Typ* zu. Können einem Ausdruck mehrere Typen zugewiesen werden, so ist dies der Typ, aus dem alle diese Typen durch Substitution der freien Typvariablen im zugehörigen Typausdruck hergeleitet werden können. Da die statische Semantik (d.h. die Zuordnung von Typen zu Ausdrücken) von Prolid-Programmen identisch mit denen von Haskell-Programmen ohne Klassen ist, werden die Regeln hierfür hier nicht aufgeführt; sie sind z.B. in [14] aufgeführt.

4.3.1 Typausdrücke

type	→	bType [-> type]	(funktionaler Typ)
bType	→	[bType] aType	(Typ-Applikation)
aType	→	typeConId‘	
	|	typeVarId	
	|	($type_1$, … , $type_k$)	(k≥2)
	|	[type]	
	|	(type)	
typeConId‘	→	typeConId	
	|	()	
	|	[]	
	|	(->)	
	|	(, {,})	
typeVarId	→	varId	
typeConId	→	conId	
simpleType	→	typeConId $typeVarId_1$ … $typeVarId_n$	(n≥0)

Die Syntax von Typausdrücken ist ähnlich zu der von Ausdrücken (aus denen sie ja hergeleitet werden). Zu der Syntax für die einfache Typ-Applikation kommt eine besondere Syntax für die Applikation der vordefinierten Typkonstruktoren für den Funktions-, Tupel- und Listen-Typ: Typausdrücke der Form T_A -> T_B oder (äquivalent dazu) `(->)` T_A T_B heißen *funktionale Typen* oder *Funktionstypen*. Sie werden für Higher-order-Ausdrücke inferiert. Der Typkonstruktor `(->)` wird also analog zu einem Infix-Operator in einem Ausdruck verwendet, was mit der Syntax für λ-Ausdrücke korrespondieren soll. Ein sogenannter *Tupel-Typ* hat die Form (T_1, … ,T_n). Dies ist eine besondere Syntax für die Applikation des Tupel-Typkonstruktors `(,{,}))`. Ein *Listen-Typ* hat die Form `[T]`, was einer Applikation des Listen-Typkonstruktors `[]` entspricht. Daraus folgt, daß alle Elemente einer Liste denselben Typ haben müssen, während es Tupel mit Elementen verschiedenen Typs geben darf.

4.3.2 Typ-Deklaration

topLevelDecl	→	typeDecl	
typeDecl	→	`data` simpleType = constrs	
constrs	→	$constr_1$ | … | $constr_n$	(n≥1)
constr	→	conId $aType_1$ … $aType_k$	(k≥0. k heißt die *Arität* des Konstruktors conId)
	|	$bType_1$ conOp $bType_2$	

Die Deklaration eines algebraischen Datentyps wird vom Schlüsselwort `data` eingeleitet und darf nur auf Modulebene stattfinden (es handelt sich um eine sogenannte Top-level-Deklaration, vgl. 4.6). Mit jeder Deklaration der Form `data` T u_1 … u_k = K_1 t_{11} … t_{1k1} | … | K_n t_{n1} … t_{nkn} wird ein neuer Typ vereinbart, repräsentiert durch den Typausdruck T u_1 … u_k (T ist ein Typkonstruktor, die u_i sind Typvariablen), sowie neue Konstruktoren K_i vom Typ t_{i1} → … → t_{ik} → T u_1 … u_k. Weder der Typkonstruktor noch die Konstruktoren dürfen mehrere Male deklariert werden.

Die Typ-Konstruktoren `(->)`, `(,{,}))`, `[]`, `Char`, `Int` sowie `()` (der Typ `Unit`) müssen in einem Prolid-Modul nicht mit einer `data`-Deklaration vereinbart werden. Ansonsten weisen

diese Typ-Konstruktoren und die damit konstruierten Typen aber keinerlei Besonderheiten gegenüber den vom Anwender vereinbarten Typ-Konstruktoren bzw. Typen auf.

Anmerkungen zur Implementation:
Analog zum Typ `Expression` für Ausdrücke (vgl. 4.4) haben Typ-Ausdrücke den Typ `TypeExpression`. Beim Parsen werden zunächst alle Bezeichner in Typausdrücken vom Typ `TypeExpression` unter dem Konstruktor `TypeRawId` gespeichert. Erst nach Beendigung des Parsens werden die Bezeichner den Klassen `TypeConstructor` und `TypeVariable` zugeordnet. Dies geschieht anhand der `data`-Deklarationen. Daher spielt die Reihenfolge dieser Deklarationen in einem Modul keine Rolle (vgl. 4.6). Als Objekte vom Typ `DataDeclaration :: ModuleEntry` werden Typ-Deklarationen mit den anderen Top-level-Deklarationen in einer Datenstruktur vom Typ `Module` gespeichert. (Nur) bei der Typinferenz wird auf diese Einträge zurückgegriffen, um den Typ der Konstruktoren zu bestimmen.

4.3.3 Typ-Synonyme

topLevelDecl → typeSynDecl

typeSynDecl → `type` simpleType = type

Mit einer Deklaration der Form `type` T u_1 … u_n = t wird der neue Typ T t1 … tk als äquivalent zu $t[t_1/u_1, \ldots, t_k/u_k]$ vereinbart. Der Typkonstruktor T darf an keiner anderen Stelle vereinbart sein, die Typvariablen u_i müssen untereinander verschieden sein.

Nicht alle syntaktisch möglichen Deklarationen von Typ-Synonymen sind erlaubt; mehr dazu ist im Haskell-Report [17] zu finden (§4.2.2).

4.4 Ausdrücke

Übersicht und Vorrangregeln (die Notation mit hochgestellten Indizes ist zu Beginn von Kapitel 4 erklärt):

exp	→	exp^0	
	|	exp^0 or exp	(vgl. 4.4.3)
exp^i	→	exp^{i+1} [$op^{(n, i)}$ exp^{i+1}]	
	|	$lExp^i$	
	|	$rExp^i$	
$lExp^i$	→	($lExp^i$ | exp^{i+1}) $op^{(l, i)}$ exp^{i+1}	
$rExp^i$	→	exp^{i+1} $op^{(r,i)}$ ($rExp^i$ | exp^{i+1})	
exp^{10}	→	- exp	(abweichend von Haskell, vgl. 4.2)
	|	lambdaAbstraction	
	|	letExpression	
	|	ifExpression	
	|	caseExpression	
	|	fExp	
fExp	→	[fExp] aExp	(Applikation)
aExp	→	var	
	|	con	
	|	list | tupel | unit	
	|	(exp)	
	|	(exp^{i+1} $op^{(a,i)}$)	(linke Section)
	|	($op^{(a,i)}$ exp^{i+1})	(rechte Section)
	|	(exp or)	(linke Or-Set-Section)
	|	(or exp)	(rechte Or-Set-Section)
	|	`impossible`	(leeres Or-Set)

Anmerkungen zur Implementation:
Es ist zwischen Prolid-Ausdrücken und Ausdrücken der Kernsprache zu unterscheiden. Die Kernsprache ist im Kapitel 2 definiert und besitzt eine implementatorische Entsprechung in dem Haskell-Typ `Expression` (im Modul `ParserHelper`). Dieser Datentyp besitzt zusätzlich Konstruktoren für Prolid-Ausdrücke, die beim Parsen nicht direkt als Kernsprachen-Konstrukte gelesen werden konnten. Im folgenden wird u.a. beschrieben, wie solche Ausdrücke in die Kernsprache übersetzt werden. Dieser Vorgang heißt *Transformation*. Die in 4.7 angegebene operationelle Semantik muß daher nur die Kernsprache abdecken.
Da jedes Modul die Prelude („Standardbibliothek“) `Stdlib` einbinden muß, gelten im folgenden freie Bezeichner wie z.B. `True` als deklariert.

4.4.1 Variablen, Konstruktoren und Operatoren

aExp	→	var | con | list | tupel | unit | `(` exp `)`	
con	→	`()` | `[]` | `(,{,})` | con‘ | literal (man beachte die Entsprechungen unter 4.3.1)	
literal	→	char | string | int	
con‘	→	conId | `(` conOp `)`	
var	→	varId | `(` varOp `)` | queryVarId	
varId	→	smallId	
queryVarId	→	queryId	
conId	→	largeId	
varOp	→	(symbol { symbol | `:` })$_{(reservedOp)}$ | ‘ varId ‘ | ‘ conId ‘	
conOp	→	(`:` { symbol | `:` })$_{(reservedOp)}$	
reservedOp	→	`::` | `=` | `\` | | | `->` | `@` | `~` | infixOrId	
`op`	→	varOp | conOp	
list	→	`[` exp_1`,` …`,` exp_n `]`	(n≥0, vgl. 4.3.1)
tupel	→	`(` exp_1`,` …`,` exp_n `)`	(n≥2, vgl. 4.3.1)
unit	→	`()`	(vgl. 4.3.1)

Infix-Operatoren dürfen in Präfix-Notation eingesetzt werden, wenn sie in Klammern gesetzt werden. Umgekehrt dürfen Präfix-Funktionen wie Infix-Operatoren verwendet werden, falls sie innerhalb von *backquotes* (`) geschrieben werden. Für solche Operatoren gelten dieselben Regeln wie für sonstige Operatoren; auch Fixity-Deklarationen sind für sie möglich (vgl. 4.7.2).
Die Listen- und Tupel-Konstruktoren sowie `()` (der Unit-Konstruktor) müssen nicht mit einer `data`-Deklaration vereinbart werden. Abgesehen von ihrer z.T. abweichenden Syntax (siehe unten) weisen diese Konstruktoren aber keinerlei Besonderheiten gegenüber den vom Anwender vereinbarten Konstruktoren auf. `[` exp_1`, …,` exp_n `]` ist eine andere Schreibweise für

exp_1 `:` `(`exp_2 `:` `(` … `(`exp_n `:` `[]))).`

Anmerkungen zur Implementation:
Beim Parsen werden zunächst alle Bezeichner in Ausdrücken vom Typ `Expression` unter dem Konstruktor `RawId` gespeichert. Erst nach Beendigung des Parsens werden die Bezeichner den Klassen `Constructor`, `Variable und Function` (für die auf der Top-level-Ebene eines Moduls deklarierten Variablen) zugeordnet. Dies geschieht anhand der `data`-Deklarationen und der Top-level-Funktionsbindungen durch die Funktion `Static.sortIdentifiers`. Daher spielt die Reihenfolge dieser Deklarationen in einem Modul keine Rolle.

4.4.2 Applikationen und λ-Abstraktionen

fExp	→	[fExp] aExp	
exp	→	\ $aPat_1$ … $aPat_n$ -> exp	
	|	exp_1 op exp_2	(Infix-Applikation)
	|	- exp	(Präfix-Applikation)
aExp	→	(exp op)	(linke Section)
	|	(op exp)	(rechte Section)

Applikationen haben die Form e_1 e_2 und sind links-assoziativ. e_1 darf eine Variable oder ein Konstruktor sein. Funktionen mit mehr als einem Argument sind also *curried* zu notieren, partielle Applikationen sind erlaubt (auch falls e_1 ein Konstruktor ist, vgl. aber den Unterschied zu Applikationen in Mustern (vgl. 4.4.4)). Ist e_1 ein Konstruktor, so spricht man auch von *Konstruktor-Applikation* (vgl. 2.1.2), ansonsten von *Funktionsapplikation*.

λ-Abstraktionen haben die Form \ p_1 … p_n -> e, wobei die p_i Muster sind (vgl. 4.6.4). Nicht nur die einzelnen Muster müssen linear sein, sondern p_1 … p_n als ganzes.

Sections dienen zur partiellen Applikation von Operatoren; entweder das linke oder das rechte Argument eines Infix-Operators werden weggelassen. Ausnahme: in einem Ausdruck der Form `(- e)` ist – der Negations- und nicht der Subtraktions-Operator. Man beachte, daß es auch für Or-Set-Generatoren so etwas wie Sections gibt, siehe 4.5.5.

`-` (die numerische Negation) ist der einzige Präfix-Operator in Prolid. Er wird in die in der Standardbibliothek `Stdlib` definierte Funktion `negate` umgesetzt. Man beachte, daß Sections und die Schreibweise `(op)` nur für Infix-Operatoren möglich sind.

Anmerkungen zur Implementation:

Funktionsapplikationen werden im Haskell-Typ `Expression` per `Apply` e_1 e_2 repräsentiert.

Für λ-Abstraktionen werden beim Parsen Ausdrücke des Typs `Expression` mit dem Konstruktor `Lambda` erzeugt, die später durch die Funktion `replaceLambdaAbstractions` im Modul `Static` folgendermaßen in die Kernsprache transformiert werden (vgl. [9] D.4):

Zuerst werden λ-Abstraktionen der Form \ p_1 … p_n -> e, bei denen nicht alle p_i Variablen sind, in Abstraktionen der Form \ x_1 … x_n -> `case (x1, …, xn) of (`p_1`, …, pn) ->` e transformiert. Dann wird für jede λ-Abstraktion der Form \ x_1 … x_n -> r eine neue Top-level-Funktionsbindung der Form fLambda x_1 … x_n fx_1 … fx_k = r generiert (vgl. 4.5.1). Die fx_i sind die freien Variablen von r. Die λ-Abstraktion wird dann durch einen Funktionsaufruf der Form fLambda fx_1 … fx_k ersetzt.

Für Operator-Applikationen gelten folgende Transformationsregeln:

Eine Infix-Applikation der Form e_1 op e_2 wird zu `(op)` e_1 e_2. Die Präfix-Funktionsapplikation - e wird zu `negate` e. Ausdrücke der Form `(`e op`)` werden zu `(op)` e, Ausdrücke der Form `(op` e`)` zu `flip (op)` e. `flip` ist im Modul `Stdlib` definiert. Alle genannten Umsetzungen werden schon beim Parsen vorgenommen (im Modul `ParserHelper.hs`).

4.4.3 Or-Set-Generatoren und Or-Set-Distribution

Im folgenden wird die Syntax der Prolid-Or-Set-Generatoren beschrieben und ein informeller Überblick über ihre operationelle Semantik gegeben.
Zwischen der in 2.3 dargestellten deklarativen Semantik der drei Arten von Or-Set-Generatoren und der implementierten operationellen Semantik von Prolid gibt es einige Unterschiede praktischer Natur. *Or-Ausdrücke* gibt es wie unter 2.3.1 beschrieben auch in Prolid in zwei Varianten als Bestimmte und Unbestimmte Or-Sets. Allerdings gibt es für die Zuweisung eines Identifikators an ein Bestimmtes Or-Set in Prolid einige Regeln, die sowohl die Implementierung als auch den praktischen Einsatz von Or-Ausdrücken erleichtern. Das *leere Or-Set* hat in Prolid eine zur unter 2.3.2 beschriebenen Semantik äquivalente Bedeutung, *Query-Variablen* wurden dagegen in einer gegenüber 2.3.3 eingeschränkten Fassung implementiert. Bevor wir die einzelnen Or-Set-Generatoren und die jeweiligen Unterschiede zur deklarativen Semantik beschreiben, übertragen wir das Konzept in 2.3 eingeführte Konzept der *Or-Set-Distribution* auf die operationelle Semantik.

4.4.3.1 Or-Set-Distribution

In den Semantik-Regeln unter 2.3 haben wir nach jeder Verknüpfung von Alternativen mehrerer Or-Sets das resultierende Or-Set validiert. Dabei wurden miteinander verknüpfte unterschiedliche Alternativen von Or-Sets mit derselben Identität als ungültig angesehen und aus dem Resultat entfernt. Im Prinzip könnte man eine solche Validierung auch zur Laufzeit eines Prolid-Programms nach allen Operationen vornehmen, bei denen Or-Sets involviert sind. Dies wäre jedoch unter Performance-Gesichtspunkten ungünstig, weil man überflüssige Zwischenresultate erhielte, die bei der Validierung kurz darauf wieder wegfielen. Die Implementation verwendet daher eine etwas andere Technik, bei der die Validierung in gewissem Sinne „on the fly“ durchgeführt wird: Jede Auswertung findet in einer sogenannte *Or-Set-Umgebung* statt. Diese legt für jedes Bestimmte Or-Set und jede Query-Variable fest, welche der Alternativen von dessen semantischem Wert gültig sind. Die Or-Set-Umgebung wird im Gegensatz zur Substitutionsumgebung (die die Variablenbindungen enthält) beim Eintritt in Funktionsrümpfe ggfs. dynamisch erweitert. Für Or-Ausdrücke ist ein Element der Or-Set-Umgebung ein Paar der Form (i, a), wobei i der Identifikator eines Bestimmten Or-Sets ist und a angibt, ob der Wert des linken oder des rechten Teilausdrucks (bei Infix-Notation des Or-Ausdrucks) gerade gültig ist (wir bezeichnen diese Werte mit L und R). Gibt es zu einem Identifikator eines Or-Sets keinen Eintrag in der Or-Set-Umgebung oder zu einem Or-Set keinen Identifikator (für Werte Unbestimmter Or-Sets), so sind alle Elemente des Or-Sets gültig. Handelt es sich um eine Query-Variable, so bezeichnet man die zu ihrem Identifikator in der Or-Set-Umgebung eingetragenen Elemente als *Constraints*. Constraints haben die Form einfacher boolscher Ausdrücke, die

angeben, welche Alternativen des Wertes der Query-Variablen (d.h. des algebraischen Datentyps der Variablen als Or-Set) gerade gültig sind. Das aus der Evaluierung des Or-Set-Generators sich ergebende Or-Set enthält die semantischen Werte genau der gültigen Alternativen. Die Or-Set-Umgebung ist anfangs leer und wird während der Evaluierung der Funktion `main` dynamisch erweitert. Eine Erweiterung findet statt, wenn ein Bestimmtes Or-Set oder eine Query-Variable *distribuiert* wird. Unter einer (Or-Set-)Distribution versteht man operationell die „Entscheidung" für eine der Alternativen des Or-Sets (oder, bei Query-Variablen, auch das Ausscheiden einer Alternative, vgl. 4.4.4): Wird ein Ausdruck ausgewertet, der einen Or-Set-Generator als Unterausdruck enthält, so ist das Resultat ein Or-Set, daß genau die semantischen Werte enthält, die sich ergeben, wenn man den Ausdruck für jede gemäß der aktuellen Or-Set-Umgebung gültigen Alternative a des Or-Set-Generators gesondert auswertet. Ist der Or-Set-Generator ein Bestimmtes Or-Set mit Identifikator i, so erfolgt jede dieser Teilauswertungen in der um (i, a) erweiterten Or-Set-Umgebung. Dieser Vorgang heißt (operationelle) *Distribution*. Der Unterschied zur Distribution, wie sie in 2.3 eingeführt wurde, ist also, daß wir dank der Or-Set-Umgebung schon *vor* der Auswertung des entsprechenden Teilausdrucks wissen, welche der Alternativen des distribuierten Or-Set-Generators dabei gültig sind. Der Ablauf von Distributionen ist im Detail unter 4.4.4 und 4.4.5 geschildert und wird unter 4.7.2 formal festgelegt.

4.4.3.2 Bestimmte Or-Sets

or	→	orWithIndex | orWithoutIndex	(„Operatoren" für Or-Ausdrücke)
orWithIndex	→	OrIdWithIndex	(„Operatoren" für Bestimmte Or-Sets)
exp	→	exp^0 orWithIndex exp	
aExp	→	(or exp)	(rechte Or-Set-Section)
	|	(exp or)	(linke Or-Set-Section)

Bestimmte Or-Sets sind unter dem syntaktischen Aspekt gesehen Operatoren im Sinne von 4.3.1. Sie werden ähnlich wie Variablen- oder Konstruktor-Operatoren verwendet. Sogar Sections sind möglich. Verwendet man keine Section, ist die Syntax für globale Or-Ausdrücke a `or` b. Im Quellcode kann direkt (ohne Leerraum) hinter das Schlüsselwort `or` auch eine natürliche Zahl

(der sogenannte *Index*) als Or-Set-Identifikator geschrieben werden (vgl. 2.3). Geschieht dies nicht (und folgt direkt auf `or` kein Stern und kein Plus-Zeichen), so wird diesem Vorkommen von "`or`" im Quellcode statisch ein exklusiver Index als Identifikator seines semantischen Wertes zugewiesen. Diese automatische Indexgenerierung erfolgt während der Tokengenerierung, also lexikalisch (vgl. 2.3).
`or` in all seinen Schreibweisen ist als „Operator" rechts-assoziativ und hat einen niedrigeren Vorrang als alle Variablen- und Konstruktor-Operatoren; Fixity-Deklarationen sind für Or-Set-Generatoren nicht möglich.
Zur Implementation der Syntax und zur statischen Semantik (Typinferenz) siehe 4.4.3.3.

4.4.3.3 Unbestimmte Or-Sets

or	→	orWithIndex | orWithoutIndex	(„Operatoren" für Or-Ausdrücke)
orWithoutIndex	→	OrIdWithoutIndex	(„Operator" für Unbestimmte Or-Sets)
exp	→	exp^0 orWithoutIndex exp	

Unbestimmte Or-Sets werden wie Bestimmte Or-Sets behandelt, mit dem einzigen Unterschied, daß die Or-Set-Umgebung bei der Auswertung keine Rolle spielt und bei einer Distribution auch nicht erweitert wird; jede Alternative ist jederzeit gültig.

Statische Semantik und Typinferenz für das Nichtterminal or (Bestimmtes und Unbestimmtes Or-Set):
Für einen Ausdruck der Form e_1 or e_2 mit Typ t, wobei t_1 der Typ von e_1 und t_2 der Typ von e_2 ist, gilt $t = t_1 = t_2$. Die Typinferenz ist folgendermaßen implementiert: ist pt_1 der Prinzipal-Typ von e_1 und pt_2 der Prinzipal-Typ von e_2, so ist pt_2 der Typ von e_1 or e_2, falls pt_2 genereller als pt_1 ist, und pt_1 andernfalls.

Anmerkungen zur Implementation des Nichtterminals or (Bestimmte und Unbestimmte Or-Sets):

Bestimmte Or-Sets und Unbestimmte Or-Sets werden durch den Konstruktor

`Or` a_1 a_2 index `:: Expression` repräsentiert. index ist `NoIndex` (für Unbestimmte Or-Sets) oder eine natürliche Zahl für Bestimmte Or-Sets (ggfs. automatisch zugewiesen, siehe 4.4.3.2). a_1 und a_2 sind beliebige Ausdrücke.

Or-Set-Sections und die Präfix-Syntax (or) aExp werden folgendermaßen transformiert:

(or e) ➔[47] (\ x -> x or e)
(e or) ➔ (\ x -> e or x)
(or) e_1 e_2 ➔ e_1 or e_2

(x ist eine neue Variable. Die λ-Abstraktionen werden gemäß 4.4.2 transformiert)

4.4.3.4 Das leere Or-Set (`impossible`)

aExp → `impossible`

Es gibt keinen Unterschied zur unter 2.3.2 beschriebenen Semantik.

Anmerkungen zur Implementation:
impossible wurde als besonderer Konstruktor implementiert (`Constructor "impossible" :: Expression`), der (wie ⊥) als Typausdruck eine allquantifizierte Typvariable hat. Auch das Verhalten von `impossible` bei der Auswertung (insbesondere beim Pattern-Matching) ähnelt ⊥; siehe dazu 4.7.2 (operationelle Semantik) und 4.4.4 (Pattern-Matching).

---

[47] a ➔ b steht für „transformiere Ausdrucksform a nach b“

4.4.3.5 Query-Variablen

queryVarId	→	queryId
var	→	queryVarId
aExp	→	var

Query-Variablen sind, syntaktisch gesehen, nicht-deklarierte, nicht-gebundene Variablen, denen man im Quellcode zur Unterscheidung von „normalen Variablen“ ein Fragezeichen voranstellt. Davon abgesehen gibt es aber keinen syntaktischen Unterschied zu sonstigen Variablen.

Anmerkungen zur Implementation:
Bei der operationellen Semantik der Query-Variablen gibt es notwendigerweise Unterschiede zur deklarativen Semantik. Folgte man naiv der deklarativen Semantik, so müßte jeder solche Or-Set-Generator die gesamte Domain eines Datentyps analog zu einem Or-Ausdruck repräsentieren. Sogar wenn man dies mit Hilfe von (Lazy-)Aufzählungsfunktionen bewerkstelligen könnte, wären Berechnungsresultate auf diese Weise im allgemeinen nicht darstellbar. Es ist daher sinnvoller, in Anlehnung an Logik-Programmiersprachen wie *Prolog* und funktional-logische Sprachen wie *Curry* [9] den semantischen Wert dieser Or-Set-Generatoren als Symbole darzustellen, ggfs. mit Zusatzinformationen. Der Variablenname entspricht in etwa dem Index eines Bestimmten Or-Sets, zur Laufzeit jeder Query-Variablen bei jedem Eintritt der Evaluation in den Funktionsrumpf, in dem die Query-Variable lexikalisch vorkommt, ein neuer Name zugewiesen wird (vgl. 2.3). Dadurch verhalten sich Query-Variablen ähnlich wie lokale Variablen. Constraints, enthalten in der Or-Set-Umgebung, bestimmen, welche der Alternativen des Or-Set-Generators gerade gültig oder ungültig sind. Diese Constraints entsprechen von der Bedeutung her gesehen den Paaren (identifikator, L | R), die für Bestimmte Or-Sets in der Or-Set-Umgebung enthalten sind, haben aber die Form von Ausdrücken des Typs `Bool` (vgl. 4.4.4 (Pattern-Matching und Constraint-Generierung)). Eine Distribution (ggfs. einschließlich der Generierung von neuen Constraints) findet für eine Query-Variable *nur* dann statt, wenn der Testausdruck eines `case`-Ausdrucks zu einer Query-Variable evaluiert. In allen anderen Fällen bleibt eine Query-Variable bei der Auswertung unverändert. Die operationelle Semantik von Query-Variablen ergibt sich daher ausschließlich aus der Semantik des Pattern-Matchings im Zusammenhang mit `case`-Ausdrücken und ist im Detail unter 4.4.4 bzw. 4.4.5 geschildert.
Einer Query-Variablen wird bei der Ausgabe die generierte Menge von Constraints beigestellt (eine Teilmenge der Or-Set-Umgebung). Da der Prolid-Interpreter bei der Ausgabe außerdem den Typ des Resultats anzeigt, sind damit alle möglichen Informationen über den indefiniten semantischen Wert der Variablen gegeben.

4.4.4 Muster, Pattern-Matching und Constraint-Generierung

pat	→	pat^0	
pat^i	→	pat^{i+1} [$conOp^{(n,i)}$ pat^{i+1}]	
	|	$lPat^i$	
	|	$rPat^i$	
$lPat^i$	→	($lPat^i$ | pat^{i+1}) $conOp^{(l,i)}$ pat^{i+1}	
$rPat^i$	→	pat^{i+1} $conOp^{(r,i)}$ ($rpat^i$ | pat^{i+1})	
pat^{10}	→	aPat	
	|	con $aPat_1$ … $aPat_k$	(k≥1, k = Arität von con)
aPat	→	var [@ aPat]	(As-Patterns)
	|	con	(Arität von con = 0)
	|	literal	
	|	- int	
	|	(pat)	
	|	(pat_1, …, pat_n)	
	|	[pat_1, …, pat_n]	
	|	~ aPat	(Irrefutable-Patterns)

Muster (*Patterns*) tauchen in `case`-Ausdrücken, Lambda-Abstraktionen und Bindungen auf. Die Semantik des *Pattern-Matchings*, also des Vergleichs des Wertes eines Ausdrucks (*Testausdruck*) mit einem Muster, ergibt sich aus der Semantik von `case`-Ausdrücken (bzw. Verzweigung in der abstrakten Syntax) der Kernsprache (siehe 2.1.2 und 4.7) in Verbindung mit den Transformationsregeln für die genannten Ausdrücke und Bindungen. Diese Transformationen führen das Pattern-Matching in allen Fällen auf bestimmte einfache `case`-Ausdrücke zurück, ausgenommen bei einem aus einer Top-level-Funktionsbindung entstandenen sogenannten *einfachen Pattern-Binding*, falls dort nur Variablen als Muster vorkommen (siehe 4.5.1). Die vereinfachte Form von `case`-Ausdrücken ist in Form der Verzweigung-Syntax ein Teil der Kernsprache.

Die Syntax von Mustern entspricht weitgehend der von einfachen Ausdrücken. Or-Set-Generatoren und Funktionen sind in Mustern allerdings nicht erlaubt (vgl. Einleitung und 2.3). Zu beachten ist weiterhin, daß Muster linear sein müssen, d.h. jede Variable darf nur höchstens einmal vorkommen. Partielle Konstruktor-Applikationen sind in Mustern verboten, d.h. die Arität von Konstruktoren in Mustern muß der Arität des jeweiligen Konstruktors bei seiner Deklaration mittels `data` entsprechen (vgl. 4.3.2).

Prolid kennt die von Haskell bekannten *As-Patterns* und alle sonstigen Varianten von *Irrefutable-Patterns* mit Ausnahme von Irrefutable-Patterns der Form N p, wobei N ein mit `newtype` definierter Konstruktor ist. Ein As-Pattern hat die Form variable@submuster und erlaubt es, variable als Name für den mit submuster matchenden Wert zu gebrauchen. Irrefutable-Patterns sind Muster, für die das Pattern-Matching immer erfolgreich ist. Es handelt sich dabei um Variablen, Wildcards (_), As-Patterns (variable@submuster) und Muster der Form ~submuster. Freien Variablen im submuster werden allerdings an ⊥ gebunden, falls das Pattern-Matching mit submuster fehlschlagen würde (näheres siehe unter „Anmerkungen zur Implementation“). Das Irrefutable-Pattern Wildcard (_) kann als freie, für jedes Vorkommen von _ neue Variable betrachtet werden, für die beim Matching keine Bindung erzeugt wird.

Anmerkungen zur Implementation / Informelle Übersicht über die operationelle Semantik des Pattern-Matchings:
(Die formale Semantik des Pattern-Matchings ergibt sich aus Kapitel 2 (deklarative Semantik), 4.4.5 (Transformation von `case`-Ausdrücken) und 4.7 (operationelle Semantik); folgende Ausführungen sind im Zusammenhang mit der operationellen Semantik von `case`-Ausdrücken und der deklarativen Semantik von Or-Set-Generatoren zu sehen.)

Ein sogenanntes *definites Pattern-Matching* (d.h. ein Pattern-Matching wie bei Haskell, siehe unten) kann drei Resultate haben: „Fehlschlag“, „Erfolg“ (zusammen mit einer Liste von Variablenbindungen) und „Divergenz“. Im Erfolgsfall wird für jede Variable im Muster eine (Lazy-)Bindung an den Wert des korrespondierenden Teilausdrucks des Testausdrucks erzeugt und für den weiteren Verlauf der Evaluierung zur aktuellen Substitutionsumgebung hinzugefügt.
Im folgenden sei e_{Test} ein Ausdruck der gegen ein Muster m matcht. Eine *erweiterte Kopf-Normalform* sei eine Kopf-Normalform oder eine Query-Variable.

Bevor das eigentliche Pattern-Matching von Prolid geschildert wird, definieren wir das sogenannte „definite Pattern-Matching“, wie in ähnlicher Form es von Haskell bekannt ist. Der semantische Wert von e_{Test} sei hier kein Or-Set oder aber das leere Or-Set (`impossible`):

Pattern-Matching gegen ein Irrefutable-Pattern m ist immer erfolgreich, auch wenn ⊥ oder `impossible` gegen das Muster matcht. Ist das Irrefutable-Pattern eine Variable, so wird die Variable an den *Wert* von e_{Test} gebunden (lazy, d.h. daß e_{Test} dazu an dieser Stelle nicht ausgewertet wird). Hat m die Form var@submuster oder ~submuster, so werden dieselben Bindungen erzeugt, wie bei einem Matching gegen submuster, oder Bindungen aller Variablen in submuster an ⊥, falls das Matching gegen submuster fehlschlagen würde.

Matcht ⊥ gegen ein Muster, daß nicht „irrefutable“ ist, so ist das Resultat „Divergenz“. In diesem Fall liefert der entsprechende `case`-Ausdruck ⊥ als Resultat zurück. Analog ergibt sich `impossible`, wenn `impossible` gegen ein „refutable“ Muster matcht.
Ergibt sich für den matchenden Wert nicht ⊥, `impossible` oder ein Or-Set, und ist das Muster m „refutable“, und sind der Kopf von m[48] und der erweiterten Kopf-Normalform von e_{Test} derselbe Konstruktor, so ist das Resultat des Matchings das der Pattern-Matchings der Argumente des Kopfes gegen deren korrespondierende Submuster von m. Sind die Konstruktoren verschieden, schlägt das Matching fehl. Tritt bei einem Matching der Argumente der Konstruktor-Applikation „Divergenz“ auf bzw. schlägt es fehl, so ist das Gesamtresultat „Divergenz“ bzw. „Fehlschlag“. In jedem Fall muß, falls m nicht „irrefutable“ ist, e_{Test} also nur zu seiner Kopf-Normalform ausgewertet werden. Das Pattern-Matching ist sozusagen nur „so strikt wie nötig“.

Nun darauf aufbauend das *indefinite Pattern-Matching*, implementiert für die Kernsprache durch die Funktion `match` im Modul `Prolid`. Der semantische Wert von e_{Test} sei nun irgendein Or-Set:

Besteht m nur aus einer Variable, so verhält sich das Pattern-Matching genau wie im definiten Fall. Andernfalls ist das Resultat des Pattern-Matching eine *Menge* von Resultaten, wie sie beim definiten Pattern-Matching auftreten. e_{Test} wird zu einer Menge von erweiterten Kopf-Normalformen ausgewertet, die alle einzeln gegen m gematcht werden. Die Einzelresultate werden dann zum Gesamtresultat als Menge zusammengefaßt.
Jedes Element des semantischen Wertes von e_{Test} *kann* dadurch zustande kommen, daß im Verlauf der Auswertung von e_{Test} mindestens ein Or-Set-Generator ausgewertet wird. Somit stellt die Reduktion von e_{Test} eventuell eine *Distribution* im Sinne von 4.4.3 dar. Dies muß bei der Auswertung einer Alternative des `case`-Ausdrucks, in dem m enthalten ist (bei einem erfolgreichen Pattern-Matching einer erweiterten Kopf-Normalform gegen m) berücksichtigt werden. Gemäß den Regeln einer Distribution muß diese Auswertung in einer erweiterten Or-Set-Umgebung erfolgen: Jede einzelne erweiterte Kopf-Normalform wurde in einer bestimmten Or-Set-Umgebung ausgewertet. Diese Or-Set-Umgebung enthält für alle bei dieser Auswertung involvierten Bestimmten Or-Sets und Query-Variablen Angaben über die Alternativen dieser Or-Set-Generatoren, die bei den Distributionen während dieser Auswertung als gültig (oder im Falle von Query-Variablen auch: ungültig) festgelegt wurden. Die Vereinigung dieser Or-Set-Umgebung und der aktuellen Or-Set-Umgebung (in der der `case`-Ausdruck als ganzes

[48] m ist eine Konstruktor-Applikation

ausgewertet wird) bildet die genannte „erweiterte Or-Set-Umgebung“ für die Auswertung der Alternativen des `case`-Ausdrucks. Sie wird als Teil des Resultats des Pattern-Matchings betrachtet und bei der weiteren Auswertung des `case`-Ausdrucks berücksichtigt (vgl. 4.4.5.1).

Ist eine der erweiterten Kopf-Normalformen eine *Query-Variable* q, so stellt das Pattern-Matching dieses Or-Set-Generators gegen m selbst eine Distribution dar: Es sei ou_q die Or-Set-Umgebung, in der q erzeugt wurde (wir nennen sie „erweiterte Or-Set-Umgebung“). q repräsentiert alle Elemente (Grundterme) seines Typs, die gemäß der q betreffenden Constraints in ou_q gültig sind. Für jede dieser Alternativen ist ein separates Pattern-Matching durchzuführen. Somit ergibt sich für jede Query-Variable in der Menge der aus e_{Test} resultierenden erweiterten Kopf-Normalformen eine *Menge* r_q von Pattern-Matching-Resultaten. Das individuelle Matching der Alternativen stellt die Distribution von q dar. Zu jedem einzelnen „Erfolgs“-Resultat gehört ein Constraint, der die jeweiligen erfolgreich gematchen Alternativen von q bezeichnet. Dieser Constraint wird zur erweiterten Or-Set-Umgebung hinzugefügt, in der der mit dem erfolgreichen Matching korrespondierende Zweig des `case`-Ausdrucks ausgewertet wird (vgl. 4.4.5.1). Es können mehrere (ggfs. unendlich viele) Alternativen mittels eines einzigen Negativ-Constraint bezeichnet werden. Diese Constraint-Generierung bei der Auswertung eines vereinfachten `case`-Ausdrucks geht folgendermaßen vor sich:

Vereinfachter `case`-Ausdruck (äquivalent zur verzweigung der Kernsprache):

`case` e_{Test} `of`

K $v_1 \ldots v_n$ `->` e_1

_ `->` e_2

(K ist ein Konstruktor, die v_i sind Variablen)

Die gemäß ou_q gültigen Alternativen von q werden also gegen K und gegen _ gematcht. Es sei r_1 die Resultatemenge für das Pattern-Matching gegen K und r_2 die Resultatemenge für das Pattern-Matching gegen _. An jedes „Erfolgs“-Element einer der beiden Resultatemengen sei eine erweiterte Or-Set-Umgebung gebunden, in der e_1 bzw. e_2 ausgewertet werden. Genau dann wenn ou_q keinen Constraint q /= K und keinen Constraint q == L, L≠K enthält, ist für das Pattern-Matching von q gegen K in r_1 das Resultat „Erfolg“ zusammen mit der Or-Set-Umgebung ouq ∪ { Constraint q == K } enthalten (bei der Auswertung von e_1 ist q also gewissermaßen „definit“, so als gelte q = K). Analog ist genau dann, wenn ou_q keinen Constraint

q == K enthält, in r_2 das Resultat „Erfolg“ enthalten, zusammen mit der Or-Set-Umgebung ouq ∪ { Constraint q /= K }[49].

Es bestehen folgende *Einschränkungen* für die Implementierung dieses *Constraint-Checking*-Verfahrens: Gemäß der formalen operationellen Semantik muß n = 0 gelten, falls q Element der Menge der erweiterten Kopf-Normalformen von e_{Test} ist! Diese Einschränkung besteht, weil der implementierte Constraint-Tester nur Constraints der Form Variable == Konstruktor und Variable /= Konstruktor versteht (== bzw. /= sei die semantische Gleichheit bzw. Ungleichheit). Für spätere Versionen von Prolid wäre die Einbindung leistungsfähigerer externer Constraint-Solver wünschenswert. Eine weitere Einschränkung ist, daß beim Hinzufügen eines Constraints q /= k zu einer Constraint-Menge, die q als ungleich allen anderen Konstruktoren (des jeweiligen Typs) definiert, sich das Resultat „erfüllbar“ ergibt[50].

Die vorgestellte Behandlung von Query-Variablen ähnelt der Technik des sogenannten *Narrowing* [5, 9], die bei einigen funktional-logischen Programmiersprachen wie z.B. *Curry* [9] oder *ALF* als Teil des Evaluationsmechanismus verwendet wird. Narrowing dient essentiell dazu, *Term-Gleichungen* zu lösen. Aus der Sicht des Narrowing besteht ein funktionales Programm aus einer Menge von Term-Gleichungen in Form der Top-level-Funktionsdeklarationen (den sogenannten *Regeln*), wobei man speziell auf Funktionen mit einem Resultat vom Typ Bool abzielt (solche Funktionen entsprechen den *Prädikaten* logischer Programmiersprachen wie z.B. *Prolog*). Die freien Variablen in Termen nehmen die Rolle von Unbekannten ein, für die es Lösungen zu finden gilt, z.B. in einer Gleichung der Form X > 5 = True. Eine solche Gleichung wird gelöst, indem die linke Seite mit allen passenden linken Seite der Regeln unifiziert wird. Dabei entstehen Bindungen für die Unbekannten. Die Vorkommnisse der Variablen auf der rechten Seite der jeweiligen Gleichung werden entsprechend instanziiert. Für eine funktionale Sprache gilt: Beschränkt man sich auf First-Order-Objekte als Werte der Variablen, so wird die Funktion für eine Unbekannte als Argument effektiv für alle möglichen Argumente ausgewertet, wobei die rechte Seite der Funktionsbindung (also der Verknüpfung der Deklarationen einer Funktion) nichtdeterministisch für *alle* Belegungen der Unbekannten auszuwerten ist. Dies entspricht der Instanziierung. In Prolid tritt an die Stelle der Unifikation die Generierung eines Constraints. Ein positiver Constraint Variable == Konstruktor als Gültigkeitsinformation in der

---

[49] In der Implementation findet der Einfachheit halber kein eigenes Pattern-Matching gegen das Muster _ statt. Vielmehr wird e_2 ausgewertet, wenn in r_1 das Teilresultat „Fehlschlag“ enthalten ist. Dies ist für das Matching von q gegen K genau dann der Fall, wenn bei einem Matching gegen _ in r_2 „Erfolg“ enthalten wäre. Es ist also durchaus möglich, daß r_1 „Erfolg“ und „Fehlschlag“ zugleich enthält.

[50] Im allgemeinen ist die Erfüllbarkeit einer solchen Formelmenge nicht entscheidbar.

Or-Set-Umgebung, in der die rechte Seite der Funktion auszuwerten ist, nimmt die Stelle einer Variablen-Bindung ein. Weil Prolid als „Funktionsparameter“ (bzw. nach der Transformation in die Kernsprache: als Muster in einem Zweig eines `case`-Ausdrucks) auch das Wildcard _ kennt, gibt es auch Negativ-Constraints (Variable /= Konstruktor). Ein solcher Constraint steht für *alle anderen* Bindungen als die der Variablen an den Konstruktor. Die nichtdeterministische Auswertung des Funktionsrumpfes beim Narrowing[51] schließlich entspricht der Distribution des Or-Set-Generators in Gestalt der Query-Variablen.

Zur Implementation der Syntax: Muster haben wie Ausdrücke den Typ `Expression`. Verwendet werden (nur) die Konstruktoren `Variable` (auch für Wildcards, intern sind Muster also nicht unbedingt linear), `Constructor`, `Apply` (nur für Konstruktor-Applikationen), `Irrefutable` (für Muster der Form ~submuster) und `As`. Letztere beiden `Expression`-Formen sind nach der Transformation von `case`-Ausdrücken in die Kernsprache nicht mehr vorhanden Die Implementation des Pattern-Matching folgt genau der unter 4.7.2 beschriebenen operationellen Semantik der Kernsprache.

4.4.5 Bedingte Ausdrücke

4.4.5.1 `case`-Ausdrücke

exp	→	`case` exp `of` { alts }	
alts	→	alt1 `;` … `;` altn	(n≥1)
alt	→	pat `->` exp [`where` lDecls]	
	|	pat gdPat [`where` lDecls]	
gdPat	→	guard `->` exp [gdPat]	
guard	→	`|` exp^0	

`case`-Ausdrücke sind in Prolid eines der grundlegenden Konstrukte. Eine Variante, die `case`-Ausdrücke mit zwei Alternativen sind Teil der Kernsprache (verzweigung, vgl. 2.1). Andere `case`-Ausdrücke und das Pattern-Matching in Verbindung mit Bindungen und λ-Abstraktionen werden per Transformation mittels dieser Konstruktion implementiert. Auch die operationelle Semantik bezieht sich auf diese Form von `case`-Ausdrücken.

---

[51] oder jedenfalls bei vielen Narrowing-Strategien, insbesondere dem *Lazy-Narrowing* [2] und dem *Needed-Narrowing* [22].

Die allgemeine Form des `case`-Ausdruck ist

`case` e_{Test} `of { `p_1 $match_1$ `; … ; `p_n $match_n$ `}` (n ≥1), wobei jedes $match_i$ folgende Form hat:

`|` $guard_{i1}$ `->` e_{i1}

…

`|` $guard_{im}$ `->` e_{im}

`where` $lDecls_i$ (m≥1)

Die p_i sind Muster, die $match_i$ bestehen aus *Guards* $guard_{ij}$ und Ausdrücken e_{ij} sowie einer optionalen Liste lokaler Bindungen (`where` $lDecls_i$), die sich auf die ganze Alternative p_i $match_i$ bezieht. Die konkrete Syntax erlaubt das Weglassen von Guards: p `->` e `where` lDecls ist eine abkürzende Schreibweise für p `|` `True` `->` e `where` lDecls.

Anmerkungen zur Implementation / Informelle operationelle Semantik von `case`-Ausdrücken[52]: Die Auswertung eines `case`-Ausdrucks erfolgt durch Pattern-Matching von e_{Test} (dem sogenannten *Test-Ausdruck* oder *Selektor*) gegen die alternativen Muster p_i unter Berücksichtigung der Guards. Die um die Berücksichtigung der Guards erweiterten Pattern-Matchings erfolgen sequentiell (also deterministisch) von oben nach unten im Sinne des obigen Schemas. Evaluiert e_{Test} zu einem Or-Set, so kann jedes einzelne dieser Pattern-Matchings eine Menge von Resultaten haben: e_{Test} wird in der aktuellen Or-Set-Umgebung zu einer indefiniten Kopf-Normalform, d.h. zu einer Menge von Kopf-Normalformen evaluiert (im folgenden die „Alternativen“ von e_{Test} genannt). Für jedes Element dieser Menge wird ein eigenes Pattern-Matchings gegen das Muster durchgeführt (näheres siehe 4.4.4). Ist in der Resultatemenge für ein Pattern-Matching gegen p_i für eine bestimmte Alternative alt von e_{Test} das Teilresultat „Erfolg“ (zusammen mit einer Bindungsliste für die im Muster enthaltenen Variablen sowie der sogenannten „erweiterten Or-Set-Umgebung“, vgl. 4.4.4) enthalten, und evaluiert mindestens ein $guard_{ij}$ zu `True`, so ist der Wert von e_{ij} (ausgewertet in der um die Bindungsliste erweiterten Substitutionsumgebung und der „erweiterten Or-Set-Umgebung“ für das kleinste i und das kleinste j, für daß dies zutrifft, im für den `case`-Ausdruck sich als Auswertungsresultat ergebenden Or-Set enthalten. Ist dagegen für ein einzelnes Pattern-Matching gegen p_i in dieser

[52] Die formale Semantik ergibt sich aus Kapitel 2 (deklarative Semantik von verzweigung-Ausdrücken), den Transformationsregeln für `case`-Ausdrücke und 4.7.2 (operationelle Semantik).

Menge für eine Alternative von e_{Test} das Teilresultat „Fehlschlag“ enthalten, oder evaluieren andernfalls alle $guard_{ij}$ zu `False`, so ist e_{ij} nicht im Resultat des `case`-Ausdrucks enthalten. Analog ist ⊥ bzw. `impossible` genau dann im Resultat enthalten, wenn sich für mindestens ein Pattern-Matching eines (Or-Set-)Elements von e_{Test} ⊥ bzw. `impossible` ergibt.

Zur Implementation der Syntax:
Alle `case`-Ausdrücke werden durch Transformation auf folgende elementare Form zurückgeführt (Funktion `Static.transformation`):

```
case eTest of
 p  -> e1
 _  -> e2
```

p ist dabei ein Muster der Form var oder con var_1 … var_n (n≥0) (andere Muster werden als *Deep Patterns* bezeichnet).
Dies entspricht der abstrakten Kernsprachen-Syntax (vgl. Kap. 2)
verzweigung (Ausdruck$_{Test}$, Muster, Ausdruck$_A$, Ausdruck$_B$).

Die Transformation in die Kernsprache erfolgt nach folgenden Regeln, die in der Implementation angewendet werden, bis keine Änderung mehr möglich ist (vgl. [17], §3.17.2):

(1) `case t of {` p_1 $match_1$ `;` … `;` p_n $match_n$ `}` ➔ (vgl. (a))

```
case t of {
 p1 match1 ;
 _  -> …
       case t of {
            pn matchn ;
            _  -> ⊥
       } … }
```

(jedes $match_i$ hat die Form `|` $g_{i,1}$ `->` $e_{i,1}$ `;` … `;` `|` $g_{i,mi}$ `->` $e_{i,mi}$ `where {` $lDecls_i$ `}`

(2) `case t of { p | g1 -> e1 ; … | gn -> en where { lDecls } -> e‘ }` ➔

```
case e‘ of {
  y -> case t of {
         p -> let { lDecls } in
                   (case g1 of True -> e1 ; _ ->
                   … case gn of True -> en ; _ -> y );
         _ -> y } }
```

(y sei dabei eine neue Variable. `let`-Ausdrücke werden weiter transformiert, siehe)

(3) `case t of { ~p -> e; _ -> e‘ }` ➔

`(\x1‘ … xn‘ -> e1) (case t of { p -> x1 }) … (case t of { p -> xn })`

`where` e_1 = $e[x_1‘ / x_1, …, x_n‘ / x_n]$

({ x_i } = variablen(p), die $x_i‘$ seien neue Variablen).

(4) `case t of { v@ p -> e; _ -> e‘ }` ➔

```
case t of { p -> ( \x -> e ) t ; _ -> e‘ }
```

Die λ-Abstraktion in (3) und (4) wird gemäß 4.6.3 in die Kernsprache transformiert.

`case t of { K p1 … pn -> e ; _ -> e‘ }` ➔

```
case t of {
 K x1 … xn -> case x1 of {
                p1 -> … case xn of { pn -> e ; _ -> e‘ } …
                _ -> e‘ }
```

falls nicht alle p_i Variablen sind; K ist ein Konstruktor, die x_i sind neue Variablen

Im Zusammenhang mit der Transformation komplexer `case`-Ausdrücke mit indefiniten Testausdrücken in diese einfache Form taucht folgendes Problem auf:

Wenn der Ursprungsausdruck (im einfachsten Fall) folgende Form hat (A, B, X, Y seien verschiedene Konstruktoren, i ein bestimmter Index; der Übersichtlichkeit wegen wird die Layout-Regel verwendet):

(1)
```
case A ori B of
 A -> X
 B -> Y
```

so ergibt sich nach der Transformation daraus:

(2)
```
case A ori B of
 A -> X
 _ -> case A ori B of
       B -> Y
       _ -> ⊥
```

Statt dem für (1) intuitiv richtigen Ergebnis < X, Y > erhält man für (2) und damit auch für (1) das Resultat < X, Y, ⊥ >. Der Grund dafür ist, daß beim Matching des Wertes von A `ori` B gegen den ersten _-Zweig von (2) keine Gültigkeitsinformation für das Or-Set i in die Or-Set-Umgebung aufgenommen wird, in der der zweite Testausdruck A `ori` B von (2) ausgewertet wird. Dadurch matcht der Wert von A `ori` B auch gegen den zweiten _-Zweig von (2). Eine Abhilfe für dieses Problem für Typen mit einer endlichen Domain ist 4.4.5.2 für `if`-Ausdrück geschildert.
Es ist übrigens schon allein für die Transformation von case-Ausdrücken in die Kernsprache wichtig, über Bestimmte Or-Sets zu verfügen. Wie man an den Transformationsregeln sehen kann, werden Or-Ausdrücke z.B. in Form von Testausdrücken für verschiedene Transformationen mehrfach bewertet (vgl. 2.3, z.B. durch Zuweisung an eine Variable, die an mehreren Stellen auftritt).

4.4.5.2 `if`-Ausdrücke

exp → `if` exp_1 `then` exp_2 `else` exp_3

Anmerkungen zur Implementation:
Ausdrücke der Form `if` e_1 `then` e_2 `else` e_3 werden in folgendes Kernsprachenkonstrukt transformiert:

```
case e1 of {
  True -> e2
  ; _ -> case e1 of {
            False -> e3;
            _ -> impossible } }
```

Man beachte die Bemerkungen in 4.4.5.1 zur Problematik der Transformation von indefiniten `case`-Ausdrücken in die Kernsprache. Das dort erwähnte Problem tritt bei obiger Konstruktion nicht auf, weil hier allen möglichen Konstruktoren des Typs `Bool` genau eine Alternative in einem der beiden `case`-Ausdrücke zugeordnet ist. Ist nun e_1 ein Ausdruck mit dem Wert < `True`, `False` >, so führt das Matching von `True` im zweiten `case`-Ausdruck gegen _ zum Teilresultat < >, mithin also zum Gesamtresultat < e2, e3 >, was der intuitiven Erwartung entspricht.

4.5 Variablen-Deklaration

Im folgenden werden Top-level-Deklarationen und lokale Deklarationen mittels `let` und `where` beschrieben. Top-level-Deklarationen bleiben bei der Übersetzung in die Kernsprache erhalten. Lokale Deklarationen werden in `case`-Ausdrücke und λ-Abstraktionen übersetzt, wobei letztere wiederum in Top-level-Bindungen transformiert werden (sogenanntes *Lifting* der λ-Abstraktion). Die einzige Top-level-Bindung für Variablen ist in Prolid die *Funktionsbindung* (zu der auf Modulebene auch einfache Variablenbindungen zählen, also solche, bei der die Arität der Funktion gleich null ist. Der Begriff „Funktionsbindung“ wird auch verwendet, wenn die rechte Seite der Deklaration keinen funktionalen Typ hat!). Die *Muster-Bindung* (*pattern binding*) ist als lokale Bindung mittels `let` oder `where` erlaubt, anders als bei Haskell aber nicht auf dem Top-level eines Moduls. Higher-order-Bindungen (z.B. `(+) = add`) und die Deklaration von Funktionalen (z.B. `map fn … = …`) sind auf jeder Ebene erlaubt.

4.5.1 Top-level-Bindungen

funDecl	→	funLhs = exp [`where` lDecls]	
	|	funLhs gdRhs [`where` lDecls]	
funLhs	→	var aPat { aPat }	
	|	pat^{i+1} $varOp^{(a,i)}$ pat^{i+1}	
	|	$lPat^{i}$ $varOp^{(l,i)}$ pat^{i+1}	
	|	pat^{i+1} $varOp^{(r,i)}$ $rPat^{i}$	
gdRhs	→	guard = exp [gdRhs]	
	|	| exp^{0}	
lDecls	→	$lDecl_1$; … ; $lDecl_n$	(n≥0)
lDecl	→	funDecl	
	|	patDecl	
patDecl	→	pat^{0} = exp [`where` lDecls]	
	|	pat^{0} gdRhs [`where` lDecls]	

Die Funktionsbindung für eine Variable x bestehen aus einer Menge von Deklarationen der folgenden Form:

x p_{11} … p_{1k} $match_1$

…

x p_{n1} … p_{nk} $match_n$

wobei die p_{ij} Muster sind und jedes $match_i$ die Form

= e_i `where` { $lDecls_i$ }

oder

| g_{i1} = e_{i1}

…

| g_{imi} = e_{imi}

`where` { $lDecls_i$ }

hat ($n \geq 1$, $1 \leq i \leq n$, $m_i \geq 1$).

= e_i `where` { $lDecls_i$ } ist eine Abkürzung für | `True` = e_i `where` { $lDecls_i$ }. Hat die linke Seite einer Deklaration die Form a x b, und ist x ein Variablen-Operator, so steht diese Schreibweise für (x) a b.

Anmerkungen zur Implementation:
Deklarationen obiger Form für eine Variable x werden in folgende Form übersetzt:

x x_1 … x_k =

`case` (x_1, …, x_k) `of`

(p_{11}, …, p_{ik}) $match_1$

…

(p_{m1}, …, p_{mk}) $match_m$

(die x_i sind neue Variablen)

Diese Zusammenfassung von Deklarationen zu einer einzigen Bindung findet allerdings nur statt, wenn die Deklarationen für x nicht durch Signaturen für x getrennt sind. Ist diese Trennung der Fall, so gilt x als *überladen* und es werden so viele eigene Bindungen erzeugt wie Signaturen vorhanden sind (vgl. 4.5.3). Dies ist eine Erweiterung gegenüber Haskell.

Die rechte Seite dieser Form wird wiederum mit den Transformationen für komplexe `case`-Ausdrücke in die Kernsprache transformiert (vgl. 4.4.5.1). Die Bindung wird dann als Wert vom

Typ `FunctionBinding :: ModuleEntry` in der Datenstruktur vom Typ `Modul` (vgl. 4.6) gespeichert (zusammen mit anderen Top-level-Deklarationen wie z.B. `data`-Deklarationen). In der Substitutionsumgebung, in der Ausdrücke ausgewertet werden, und in der alle durch Pattern-Matching erzeugten Bindungen für Variablen enthalten sind, kommen Top-level-Funktionsbindungen nicht vor.

4.5.2 Lokale Deklarationen

exp → `let` lDecls `in` exp

Die lokale Deklarationen dienen der Variablenbindung. Ein Ausdruck der Form `let {` d_1 `;` … `;` d_n `}` `in` e vereinbart eine Liste von ggfs. wechselseitig-rekursiven Variablendeklarationen. Es darf sich dabei sowohl um Funktions- als auch um Musterbindungen handeln. Der Geltungsbereich der Deklarationen ist (lexikalisch) 1.) die rechte Seite jeder Deklaration d_i und 2.) e. Die sehr ähnliche Syntax für lokale Deklarationen mit `where` wird effektiv in `let`-Ausdrücke transformiert. Funktions- und Musterbindungen haben die unter 4.5.1 beschriebenen Formen funDecl und patDecl. Musterbindungen werden lazy gematcht (so als wäre dem jeweiligen Muster ein ~ vorangestellt, um es „irrefutable“ (unabweisbar) zu machen).

Anmerkungen zur Implementation:

Ein `let`-Ausdruck der Form `let {` d_1 `;` … `;` d_n `}` `in` e wird folgendermaßen in die Kernsprache übersetzt. (Da Musterbindungen, auch im Zusammenhang mit `where`, nur über `let`-Ausdrücke verfügbar sind, wird hiermit auch deren Semantik beschrieben.):

Zuerst wird jede Deklaration d_i in eine Deklaration der Form $p_i = e_i$ übersetzt:

Ist d_i eine Muster-Bindung, hat also die Form

p `|` g_1 = e_1

… `|` g_m = e_m `where {` decls `}`

(p ist ein Muster, die g_i sind Guards), so wird d_i in die Form

p = `let` decls `in`

`if` g_1 `then` e_1 `else` …

`if` g_m `then` e_m `else` ⊥

transformiert. Die rechte Seite dieser Deklaration wird mit den Transformationsregeln für `if`-Ausdrücke (vgl. 4.4.5.2) in einen `case`-Ausdruck e_i übersetzt.

Ist d_i eine Funktionsbindung, so wird die Transformation nach $p_i = e_i$ gemäß 4.5.1 vollzogen.

Nun werden diese Deklarationen gemäß folgender Regeln transformiert (bis keine Regel mehr angewendet werden kann):

- transformiere immer `let { `p_1` = `e_1` ; … ; `p_n` = `e_n` } in `e_0 ➔

  `let(~`p_1`, …,~`p_n`) = (`e_1`, …, `e_n`) in `e_0
- falls fvar(p) ∪ fvar(e_1) = {}: `let p = `e_1` in `e_0 ➔

  `case `e_1` of ~p -> `e_0
- andernfalls („rekursives `let`“): `let p = `e_1` in `e_0 ➔

  `let p = Y(\ ~p -> `e_1`) in `e_0

Bis zu diesem Punkt werden die Transformationen im Zuge des Parsens der `let`-Syntax im Modul `ParserHelper` durchgeführt. Zuletzt werden noch die Transformationsregeln für `case`-Ausdrücke und λ-Abstraktionen angewendet, um den resultierenden case-Ausdruck in die Kernsprache zu übersetzten.

4.5.3 Typ-Signaturen und Ad-hoc-Polymorphismus

topLevelDecls → signDecl

signDecls → vars :: type

(Typ-)Signaturen sind nur für Top-level-Funktionsdeklarationen erlaubt. Eine Signatur für die Variable x bezieht sich auf die nächste folgende Funktionsbindung für x. Eine Funktionsbindung für x umfaßt alle Deklarationen für x, die nicht durch eine Signatur für x getrennt sind und sich im selben Modul befinden (vgl. 4.6). Mehrere nicht durch Deklarationen für x getrennt Signaturen für x sind nicht erlaubt.
Gibt es für x in einem Programm mehrere Bindungen, so ist x eine *überladene Funktion* (siehe unten). Durch jede dieser Bindungen wird eine andere Funktion mit dem Namen x beschrieben (wir nennen sie die sogenannte *Instanz von x*). Die verschiedenen Bindungen für x müssen sich nur in Bezug auf die Arität der jeweiligen Instanz von x gleichen; die Typen der formalen Parameter und des Funktionsresultats dürfen sich unterscheiden. Für jedes Vorkommnis von x im Programm wird statisch entschieden, auf welche der Bindungen für x es sich bezieht.
Eine Signatur der Form x :: t legt den Typ vom nachfolgend gebunden x als t fest. t muß (abgesehen von Typ-Variablen-Umbenennungen) identisch mit dem inferierten Typ für x sein, oder weniger generell. In letzterem Fall gilt t während der Typinferenz für alle Vorkommnisse von x als Typ von x, und nicht der inferierte Typ wie es sonst der Fall wäre. Im Unterschied zu Haskell ist es nicht erlaubt, mit Hilfe einer Signatur den Typ einer Funktion zu generalisieren (sogenannte *polymorphe Rekursion*, vgl. [17], §4.4.1).

Somit können Typ-Signaturen in Prolid drei Aufgaben erfüllen:

- Kommentierung des Programms und Überprüfung auf Übereinstimmung der Signatur und des inferierten Typs einer Funktion
- Zuweisung eines weniger generellen Typs an eine Funktion als durch die automatische Typinferenz ohne Berücksichtigung der Signatur ermittelt würde
- Trennung von Deklarationsgruppen zwecks Überladung einer Funktion (*Ad-hoc-Polymorphismus*)

Anmerkungen zur Implementation:
Die in Prolid enthaltene Form des Ad-hoc-Polymorphismus wurde eigens für diese Diplomarbeit entwickelt, weil für die graphische Darstellung von Grundtermen und Zuordnungslisten eine Form der Ad-hoc-Überladung von Funktionen unumgänglich ist (siehe unten), die zeitraubende Implementierung der Haskell- oder Gofer-eigenen *Typ-Klassen* aber vermieden werden sollte.

Gibt es im Quellcode zu einer Funktion x mehrere Bindungen d_i, so wird statisch jede dieser Bindungen in eine Bindung d'_i für eine neue Funktion x#i transformiert. Von der Umbenennung der Funktion abgesehen, sind die neuen Bindungen mit den alten identisch. Für jede Bindung wird die gewöhnliche Typinferenz durchgeführt und die Korrektheit der Signatur geprüft. Danach wird für jedes Vorkommnis von x im Programm versucht, dieses Vorkommnis x_v durch den korrekten Bezeichner x#i zu ersetzen. Auch dies geschieht vor der Laufzeit während der Typinferenz; daher spricht man von *statischer Auflösung* der Überladung von x. Es ist klar, daß dies nur funktionieren kann, wenn der inferierte Typ von x_v nicht zu generell gegenüber den Signaturen der Bindungsgruppen von x ist (in gewissen Grenzen kann diee Inferenz dieses Typs durch die „freiwillige" Hinzufügung einer Signatur für die Funktion, in deren Rumpf x_v vorkommt, vom Programmierer beeinflußt werden, siehe oben). Genau bedeutet dies folgendes: Im Zuge der normalen statischen Typinferenz (Funktion `typeof` im Modul `Static`) für den Ausdruck, in dem x_v auftritt, werden die Typen a_i der syntaktischen Argumente von x_v bestimmt[53]. Dies bedingt, daß die Funktionsapplikation von x_v dieselbe Arität wie x hat. Dann wird jede einzelne Signatur der Bindungen d'_i daraufhin überprüft, ob die Signatur mit dem Typausdruck a_1 `->` … `->` a_n `->` r unifiziert (r sei eine neue Typvariable) und, wenn ja, ob die Signatur genereller ist als dieser Typausdruck (Funktion `resolveOverloadings` im Modul `Static`). Sind beide Bedingungen für eine Signaturen zu einer Bindung d'_i erfüllt, so wird x_v durch den Variablenamen x#i ersetzt. Sind die Bedingungen für mehrere Signaturen erfüllt, so wird die im Quelltext am frühesten vorkommende Instanz verwendet. Das genannte Verfahren wird solange durchgeführt, bis alle überladenen Aufrufe von x aufgelöst sind oder keine Änderung mehr eintritt. Natürlich kann der Prinzipal-Typ des Ausdrucks, in dem Aufrufe von x vorkommen, i.a. erst dann bestimmt werden, wenn die Überladung dieser Vorkommnisse aufgelöst sind. Sind nach Beendigung des geschilderten Verfahrens noch unaufgelöste Vorkommen von x vorhanden, so werden diese in Dummy-Funktionen umgewandelt, deren Aufruf zu $\perp$ evaluiert (dem Autor erschien dieses Verhalten einer Lazy-Programmiersprache angemessener zu sein als das Auslösen eines statischen Fehlers).

Das genannte Verfahren ist weit weniger leistungsfähig als die Klassen von Haskell, erfüllt aber den ihm im Rahmen dieser Arbeit zugedachten Zweck: Es soll möglich sein, für mehrere algebraische Datentypen eine einzige überladene Funktion zur graphischen Darstellung verwenden zu können. Dies ist nicht nur bequem und eleganter gegenüber der Verwendung einer

---

[53] Weil diese statische Typ-Inferenz generelle Typen ergeben kann, als die Typen der konkreten Argumente, mit denen x_v zur Laufzeit tatsächlich aufgerufen wird, ist die statische Auflösung von Überladung prinzipiell leistungsschwächer als die zur Laufzeit, so wie es z.B. mittels virtueller Methodentabellen bei C++ oder mittels *Dictionaries* bei *Gofer* und *Haskell* geschieht.

eigenen Funktionen pro Typ, sondern auch notwendig, falls ein Typ parametrisiert ist und seine Darstellungsfunktion alle Untertypen darstellen können soll (siehe Kapitel 5).

4.6 Programme

Prolid besitzt noch kein echtes Modulsystem im Sinne von Haskell. „Module“ sind in Prolid z.Zt. Quelldateien (mit der Dateiendung `.pro`, `.ipro` bzw. `.lpro`, vgl. Anhang A), die Code-Einheiten darstellen, jeweils bestehend aus einer Menge von *Top-level-Deklarationen* (Funktionsbindungen und den zugehörigen Spezifikationen, ggfs. Deklarationen für algebraische Datentypen und Typ-Synonyme sowie ggfs. `import`-Direktiven und *Fixity*-Deklarationen). Abweichend von Haskell ist jeder Datei genau ein Modul zugeordnet. Der Prolid-Interpreter lädt eine beliebige Anzahl von Modulen, wobei alle Top-level-Funktionsbindungen, `type`- und `data`.-Deklarationen eines jeden Moduls in allen anderen Modulen sichtbar sind. Wird eine Funktion in mehr als einem Modul unterschiedlich definiert, so ist undefiniert, welche dieser Bindungen gültig ist; ein Fehler tritt nicht auf. *Qualifizierte Namen* wie bei Haskell sind nicht erlaubt. Mehrfache identische Definition sind somit harmlos. Es muß in genau einem der Module eine 0-stellige Funktion `main` definiert sein. Dieses Modul wird als Programm bezeichnet. Sein semantischer Wert ist durch die operationelle Semantik definiert. Module müssen geschlossen sein, d.h. jeder Bezeichner in einem Modul muß entweder in dem Modul selbst oder in einem importierten Modul definiert sein. Ausgenommen sind Query-Variablen.

Das Laden mehrerer Module wird entweder durch das Einfügen von `import`-Direktiven in ein Modul erreicht (siehe unten), oder durch entsprechende interaktive Interpreter-Kommandos (siehe Anhang A). Jedes Modul muß das Prelude-Modul `Stdlib` importieren. Werden Module mittels `import` geladen, so entspricht die Reihenfolge beim Laden der Reihenfolge der `import`-Direktiven im Quellcode. Mutuell rekursive Module sind nicht erlaubt. Die Reihenfolge von Top-level-Deklarationen spielt innerhalb eines Moduls keine Rolle, ausgenommen im Zusammenhang mit Spezifikationen bei Ad-hoc-Überladung (vgl. 4.5.3). Ist ein Bezeichner in einem anderen Modul definiert, so muß dieses Modul vorher (mittels `import`) geladen worden sein. Die Reihenfolge beim Laden von Modulen ist auch beim Vorhandensein von *Fixity*-Deklarationen wichtig; hier wirkt eine solche Deklaration auch auf alle nachfolgend geladenen Module, das „Überschreiben“ durch eine weitere, später geladene Fixity-Deklaration für denselben Operator ist nicht erlaubt.

4.6.1 Module

Module besitzen folgende Syntax:

module	→	`module` moduleId `where` body	
moduleId	→	conId | " fileId "	
body	→	[importDecls] [fixityDecls] topLevelDecls	
importDecls	→	$impDecl_n$ `;` … $impDecl_n$ `;`	(n≥1)
importDecl	→	`import` moduleId	
fixityDecls	→	$fixityDecl_n$ `;` … $fixityDecl_n$ `;`	(n≥1)
topLevelDecls	→	$topLevelDecl_n$ `;` … $topLevelDecl_n$ `;`	(n≥1, abweichend von Haskell)
topLevelDecl	→	typeDecl	(vgl. 4.4)
	|	typeSynDecl	(vgl. 4.4)
	|	signDecl	(vgl. 4.6.2)
	|	funDecl	(vgl. 4.6.2)

(fileId ist ein Dateiname (ggfs. mit Pfad) in systemabhängiger Form. Fehlt die Dateiendung, so wird `.pro` angenommen)

Anmerkungen zur Implementation:
`import`-Direktiven werden von der Funktion `mergeModules` im Modul `ParserHelper` verarbeitet. Die Verarbeitung erfolgt also nicht durch einen eigenen Präprozessor (wie es z.B. bei den `#include`-Direktiven von C/C++ der Fall ist), sondern „vor Ort" während der Syntaxanalyse. Für eine Weiterentwicklung von Prolid wird empfohlen, `mergeModules` so auszubauen, daß Prolid ein Haskell–äquivalentes Modulsystem erhält. Im wesentlichen müssen dazu Import- und Export-Listen von Bezeichnern erlaubt werden).
Alle gemeinsam geladenen Module (bzw. deren Top-level-Deklarationen) befinden sich schließlich in einer einzigen Datenstruktur vom Typ `Module`.

4.6.2 Fixity-Deklarationen

Eine *Fixity-Deklaration* legt am Anfang eines Moduls die Assoziativität und den Vorrang von Infix-Operatoren (sowohl für Variablen- als auch Konstruktor-Operatoren) fest. Ein Operator kann links-assoziativ (infixl), rechts-assoziativ (infixr) oder nicht-assoziativ (infix) sein. Sein Vorrang wird in Stufen von 0 (niedrigster Vorrang) bis 9 festgelegt. Fehlt der Vorrang in der Fixity-Deklaration, so wird 9 angenommen, fehlt für einen Operator die Fixity-Deklaration ganz, so wird links-Assoziativität mit Vorrang 9 angenommen.

fixityDecl	→	`infixl` [digit] ops	
	→	`infix` [digit] ops	
	→	`infixr` [digit] ops	
ops	→	op_1 `,` … `,` op_n	($n \geq 1$)
op	→	varOp | conOp	

4.7 Auswertung

Im folgenden wird die operationelle Semantik von Prolid anhand der Auswertungsfunktion „evaluation“ dargestellt. Sie ist (auch in der Implementation als Funktion `prolid.evaluation`) bewußt einfach gehalten.
Der Übersichtlichkeit halber wird eine gegenüber der tatsächlichen Implementation abstrahierte Fassung der Auswertungsfunktion beschrieben[54].

4.7.1 Repräsentationsformen der Kernsprache

In dieser Arbeit wird die indefinite Kernsprache von Prolid in drei verschiedenen Darstellungen verwendet. Die ungefähren Entsprechungen sind aus folgender Tabelle ersichtlich. Es handelt sich natürlich nicht um exakte Äquivalenzen im mathematischen Sinn, sondern lediglich um eine

---

[54] Die wesentlichen Unterschiede zum Interpreter-Code (Funktion `evaluation` plus Hilfsfunktionen im Modul `prolid.hs`) sind:

- Die Implementation speichert alle Funktionsbindungen an einer festen Stelle und benutzt dann bei Funktionsaufrufen Indizes in diesen Speicherbereich statt der Funktionsrümpfe selbst. In beiden Fällen werden Variablen-Substitutionen von Funktionsbindungen getrennt verwaltet.
- Die tatsächliche Funktion `evaluation` kennt zwei Modi: Einen Modus, um die Normalform in Gestalt der Zuordnungsliste zu errechnen (Nf), und einen zur Berechnung der erweiterten Kopf-Normalform (abgekürzt HNf. Dies ist eine Kopf-Normalformen oder eine Query-Variable oder eine Query-Variable). In der Beschreibung wird diese Funktionalität der Übersichtlichkeit wegen auf die Funktionen evaluation, evaluationNf und evaluationHNf aufgeteilt.
- Die Aufgaben der Funktionen (constrained)(Mis)match werden in der Implementation von einer einzigen Funktion `match` im Modul `Prolid` übernommen.
- Die implementierte Fassung von evaluation(H)Nf besitzt einen Modus, um statt Zuordnungen lediglich einen Grundterm (bzw. eine Closure) zu erzeugen (wie bei definiten funktionalen Programmiersprachen). Dieser Modus wird nicht gesondert beschrieben. Er kann als Sonderfall einer einelementigen Zuordnungsliste aufgefaßt werden, bei der die einzige rechte Seite ein Grundterm ist.
- Die Notation ist, soweit möglich, dieselbe, die für die deklarative Semantik verwendet wurde. Der Zusammenhang zu den entsprechenden algebraischen Datentypen und Funktionsbindungen, die bei der Implementation verwendet wurden, ist trivial. Die genauen Entsprechungen sind unter 4.6.1.1 beschrieben.
- Das Syntax-Konstrukt Index-Zuordnung(…) ist in der Implementation eine strikte interne Funktion. Deren Semantik ist aber wegen der Striktheit dieselbe wie die von Index-Zuordnung.

Hilfestellung zum leichteren Vergleich der deklarativen und operationellen Semantik sowie der prototypischen Haskell-Implementation:

Darstellung für: Semantik (abstrakte Syntax, Kap. 2)	Quellcode (konkrete Syntax, Kap. 4)	Implementation (Haskell-Typ, Kap. 4)
Programm	(Modul mit Bindung für `main`)	`Module`
Top-level-Bindung	funDecl	`ModuleEntry :: FunctionBinding`
$Rumpf^0$	exp	`Expression`
$Rumpf^n$	`\` var_1 … var_n `->` e	(keine direkte Entsprechung; Felder in `FunctionBinding`)
Ausdruck	exp	`Expression`
Konstruktor	cons	`Constructor :: Expression`
Variable	var	`Variable :: Expression`
Applikation(e_1, e_2)	e_1 e_2	`Apply` e_1 e_2 `:: Expression`
Verzweigung(e_{Test}, p, e_A, e_B)	`case` e_{Test} `of` p `->` e_A `_ ->` e_B	`Case` e_{Test} p e_A e_B `:: Expression`
⊥	(keine direkte Entsprechung)	`Bottom :: Expression`
verleiheIdentität(or (e_1, e_2),i)	e_1 orWithIndex e_2 `(or)`	`Or` e_1 e_2 `i ::`
or (e_1, e_2)	e_1 orWithoutIndex e_2 `(or*)`	`Or` e_1 e_2 `NoIndex :: Expression`
?$Variable_{Typ}$	queryVar	`Variable :: Expression`

impossible	cons	`Constructor`
(semantischer Wert < >)		“`impossible`“

4.7.2 Operationelle Semantik

Wir geben nun die „offizielle“ Semantik von Prolid an. Es handelt sich hierbei um eine Formalisierung der bei der Beschreibung der einzelnen Sprachelemente angegebenen informellen operationellen Semantik (vgl. 4.4 und 4.5). Zuerst benötigen wir noch weitere Objekte:

Definitionen 4.7.2.1

(wir setzen die Definitionen und die Grammatik der Kernsprache in 2.1 und der Zuordnungslisten in 3.3 voraus. Die Nichtterminale der Grammatik der indefiniten Kernsprache (2.1.1) werden als *Mengen* der jeweils produzierten Ausdrücken betrachtet):

Jede Auswertung findet in einer bestimmten Umgebung statt:

Umgebung := { (su, ou) : su ∈ Substitutionsumgebung, ou ∈ Or-Set-Umgebung }

Anmerkung: Umgebung ist eine *Menge* von Umgebungen! Wir wollen unsere Notation (auch im weiteren Verlauf) ein wenig an den in der Implementation verwendeten Datentypen ausrichten.

Abschlüsse definieren wir analog zu den von der deklarativen Semantik her bekannten Closures (vgl. 2.1.2):

Abschluß := { (a, u) : a ∈ Ausdruck, u ∈ Umgebung }

Folgende Funktion brauchen wir, um die Umgebung festzustellen, in der der Wert eines Ausdrucks zu berechnen ist:

umgebung : Abschluß → Umgebung

umgebung ((a, u)) → u

Die *Substitutionsumgebung* enthält die aktuellen Variablenbindungen. Weil wir es für die operationelle Semantik vermeiden wollen, analog zur deklarativen Semantik Closures bzw. Abschlüsse als Ausdrücke zuzulassen, müssen wir von jetzt an statt Ausdrücken explizit Abschlüsse an die Variablen in der Umgebung binden:

Substitutionsumgebung := { (v, c : v ∈ Variable, c ∈ Abschluß }

Die Or-Set-Umgebung enthält die Gültigkeits-Informationen für Bestimmte Or-Sets und Query-Variablen (vgl. 4.4.3):

Or-Set-Umgebung := { (index, alt) : index ∈ N, alt ∈ Alternativen-Nummer }
∪ Constraint

Alternativen-Nummer := { L, R }

Constraint := { var { $\underline{==}$, $\underline{/=}$ } kA
: var ∈ Variable, kA ∈ Konstruktor-Applikation ∪ Variable }

Gemäß 3.3 sind Nichtterminale bestimmte Variablen. Wir spezifizieren nun eine für die Auswertung günstige Darstellung:

Nichtterminal := { Nichtterminal c : c ∈ Abschluß }

Nichtterminal ⊂ Variable

Hier nimmt der Abschluß die Rolle des Variablenbezeichners ein. Die Idee dabei ist, daß unser folgendes Rekursionserkennungsverfahren bei jeder Berechnung einer Normalform jedem als Abschluß vorliegenden Redex genau ein Nichtterminal zuordnet (vgl. Verfahren 3.3.3).

Bei der Berechnung der indefiniten Kopf-Normalform (vgl. 3.3) werden die Argumente dieser Ausdrucksform nicht ausgewertet, sondern bleiben in Form von Abschlüssen als Redexe erhalten.

Argument := Abschluß

Argumente := { $(a_1, \ldots, a_n)$: $a_i \in$ Argument, $n \in N$}

In 3.3 haben wir Kopf-Normalformen eingeführt. Wir ändern ihre Definition so, daß der Kopf an eine Or-Set-Umgebung gebunden ist und aus einer Query-Variablen bestehen darf[55] (sogenannte „erweiterte Kopf-Normalform“, vgl. 4.4.4 und 4.4.5). Wir brauchen die Or-Set-Umgebung, weil gemäß der Semantik von verzweigung (d.h. den `case`-Ausdrücken der Kernsprache) bei einem Matching einer Kopf-Normalform gegen ein Muster der zum Muster gehörige Zweig der Verzweigung in der Or-Set-Umgebung auszuwerten ist, in der die Kopf-Normalform entstanden ist (vgl. 4.4.4 und 4.4.5).

HNf := { $((k, ou), arg_1, \ldots, arg_n)$: k ∈ Konstruktor
∪ { ⊥ } ∪ Variable, $arg_i \in$ Argument, ou ∈ Or-Set-Umgebung, n ∈ N }

(k, ou) nennen wir *Kopf*, die arg_i *Argumente* der Kopf-Normalform (vgl. Notation 2.1.2.1).

Die indefinite Kopf-Normalform ist eine Menge von Kopf-Normalformen[56]:

HnfSet := { $\{hnf_1, \ldots, hnf_n\}$: $hnf_i \in$ HNf, n ∈ N }[57]

(der Wert von `impossible` ist z.B. als indefinite Kopf-Normalform notiert ())

---

[55] Das gibt uns die Möglichkeit, Query-Variablen als Resultat einer Auswertung zurückzuliefern

[56] Gewissermaßen ein Or-Set, allerdings nicht genau gemäß der Definition in 2.2

[57] auch hier gilt: HnfSet ist eine *Menge* von indefiniten Kopf-Normaformen!

Wir definieren nun einige Hilfsfunktionen für Pattern-Matching und Constraint-Tests.
Alle Verfahren sind unter 4.4.4 (Pattern-Matching) bzw. 4.4.5 (`case`-Ausdrücke) näher erklärt.

Die Funktion match prüft, ob die Elemente einer indefiniten Kopf-Normalform erfolgreich gegen ein Muster matchen. Für alle erfolgreich matchenden Kopf-Normalformen wird eine Umgebung zurückgeliefert wird, die die beim Matching entstandenen neuen Variablenbindungen enthält. Die Or-Set-Umgebungen innerhalb der Umgebungen wird aus dem Kopf der einzelnen Kopf-Normalformen entnommen. In diesen Or-Set-Umgebungen ist der zu dem jeweiligen erfolgreichen Matching passenden Zweig der Verzweigung (d.h. des Kernsprachen-`case`-Ausdrucks) auszuwerten:

match : Muster × HNfSet → { { $u_1,\ldots,u_n$ } : $u_i \in$ Umgebung, $n \in N$ }

({ $(v_1, a_1), \ldots, (v_n, a_n)$ }, ou) ∈ match(k $v_1 \ldots v_n$, hnfset)

:⇔ ((k, ou), $a_1, \ldots, a_n$) ∈ hnfset für Argumente a_i, wobei k ∈ Konstruktor

mismatch ist die „Inverse“ zur Funktion match. Für jedes nicht matchende Element der indefiniten Kopf-Normalform wird die Or-Set-Umgebung zurückgeliefert, in der der „Wildcard“-Zweig der Verzweigung (d.h. des „_“-Zweigs des Kernsprachen-`case`-Ausdrucks) ausgewertet wird:

mismatch : Muster × HNfSet → { { $ou_1,\ldots,ou_n$ } : $ou_i \in$ Or-Set-Umgebung, $n \in N$ }

ou ∈ mismatch(k $v_1 \ldots v_n$, hnfset) :⇔ ((k, ou), $a_1, \ldots, a_n$) ∉ hnfset (mit $a_1,\ldots,a_k \in$ Argument)

constrainedMatch generiert einen „positiven“ Constraint (d.h. einen der Form Variable == Konstruktor). Dies geschieht, wenn bei einem Matching einer Query-Variablen gegen ein Muster die um den positiven Constraint erweiterte Or-Set-Umgebung (genau gesagt, die darin ggfs. enthaltenen Constraints über die Query-Variable) „erfüllbar“ ist, d.h. die unten stehende Funktion constraintCheck das Resultat erfüllbar liefert.

constrainedMatch :

Muster × Or-Set-Umgebung × HNfSet → {{ $ou_1,\ldots,ou_n$ } : $ou_i \in$ Or-Set-Umgebung, $n \in N$ }

{ var == k $v_1 \ldots v_n$ } ∪ ou_{var} ∈ constrainedMatch(k $v_1 \ldots v_n$, ou, hnfset)[58]

:⇔ ((var, ou_{var})) ∈ hnfset ∧ var ∈ Query-Variable

∧ constraintCheck (constraints(ou_{var} ∪ ou ∪ { var == k $v_1 \ldots v_n$ })) = erfüllbar ∧ n = 0[59]

---

[58] Man beachte, daß sogenannte *Deep-Patterns* wie z.B. $K_H\ K_1$ … nicht Bestandteil der Kernsprache sind, aber in Prolid mittels Transformation als Muster und damit auch für das Narrowing erlaubt sind

[59] Die Beschränkung auf 0-stellige Konstruktor-Applikationen (also einzelne Konstruktoren) als Muster auf der rechten Seite von == bzw. /= ist notwendig, weil die Funktion constraintCheck sonst nicht funktioniert,

Die Funktion constraintMismatch generiert analog zu constraintMatch einen „negativen“ Constraint (Variable $\underline{/=}$ Konstruktor), falls constraintCheck unerfüllbar ergibt:

constrainedMismatch :

$\text{Muster} \times \text{Or-Set-Umgebung} \times \text{HNfSet} \rightarrow \{\{ ou_i,\ldots,ou_n \} : ou_i \in \text{Or-Set-Umgebung}, n \in N \}$

$\{ \text{var} \underline{/=} k\ v_1 \ldots v_n \} \cup ou_{var} \in \text{constrainedMismatch}(k\ v_1 \ldots v_n, ou, hnfset)$

$:\Leftrightarrow ((var, ou_{var})) \in hnfset \wedge var \in \text{Variable} \wedge$

$\text{constraintCheck}(\text{constraints}(ou_{var} \cup ou \cup \{ \text{var} \underline{==} k\ v_1 \ldots v_n \})) = \underline{\text{unerfüllbar}} \wedge n = 0$

Der Constraint-Tester constraintCheck prüft, ob eine Menge von Constraints *erfüllbar* ist, d.h. ob es eine Belegung für die enthaltenen Query-Variablen gibt, so daß die Konjunktion aller Constraints logisch wahr wird:

$\text{constraintCheck} : \{\{ c_1,\ldots,c_n \} : c_i \in \text{Constraint}, n \in N \} \rightarrow \{\underline{\text{erfüllbar}}, \underline{\text{unerfüllbar}}\}$

$\text{constraintCheck} \{ c_1, \ldots, c_k \} \rightarrow \exists v_1,\ldots,v_n \in \text{SemVal} : (c_1 \wedge \ldots \wedge c_k)[var_1 \leftarrow v_1,\ldots, var_n \leftarrow v_n]$,

wobei $\{ var_1, \ldots, var_n \} := \cup \text{variablen}(c_j)$. $\underline{==}$ bzw. $\underline{/=}$ werden als die semantische Gleichheit bzw. Ungleichheit interpretiert (vgl. 2.1.2ff). In 4.4.4 ist eine Einschränkung den tatsächlich implementierten Constraint-Tester betreffend formuliert.

generateOrSet ist nützlich, um ein Or-Set in Mengennotation in einen Ausdruck umzuwandeln:

$\text{generateOrSet} : \{ a : a \in \text{Ausdruck} \} \rightarrow \text{Ausdruck}$

$\text{generateOrSet} (\{alt_1, \ldots, alt_n\}) \rightarrow or(alt_1, \ldots, alt_n)$

---

siehe 4.4.4. Für eine Weiterentwicklung von Prolid sollten aber auch für Muster mit lokalen Variablen erlaubt werden.

Verfahren 4.7.2.2: (Auswertungsfunktionen zur Berechnung der Zuordnungsliste)

Die Funktion evaluation berechnet angelehnt an das Verfahren 3.3.3 zu einem Ausdruck eine Gruppe von Zuordnungen[60]. Sie greift dabei auf die Hilfsfunktionen evaluationNf und evaluationHNf zurück, die einen Ausdruck zur Normalform bzw. indefiniten Kopf-Normalform reduziert.
evaluation und evaluationNf werden der größeren Übersichtlichkeit wegen prozedural in einer leicht verständlichen *lazy* Pseudo-Programmiersprache dargestellt[61]. Die von evaluation durchgeführte Rekursionserkennung ist in der Implementierung aufgrund etlicher Tricks und der Lazy-Auswertung von Haskell übrigens viel effizienter als es die vorliegende Beschreibung erscheinen läßt. Insbesondere werden alle Ausdrücke eines Prolid-Programms vor der Auswertung an einer festen Stelle (in einem Array) gespeichert. Daher kommt man an allen Stellen, an denen Ausdrücke auszuwerten sind, mit Referenzen in Gestalt von Array-Indizes auf diese Ausdrücke aus. Eine weitere Beschleunigung könnte erzielt werden, falls man statt Substitutionsumgebungen sogenannte *Substitutionsbäume* verwenden würden (vgl. [20]). Eine Umstellung der Verfahren von (Werte-)Baum- auf *Graph-Reduzierung* (vgl. [15]) sollte ebenfalls in Erwägung gezogen werden, allerdings ist nicht bekannt, ob das Rekursionserkennungsverfahren dafür ohne weiteres angepaßt werden könnte.

Das *Gedächtnis* merkt sich Redexe (in Gestalt von Abschlüssen) und ihre Redukte[62]. Jedes Redex steht gleichzeitig für ein Nichtterminal:
gedächtnis := { (a_1, a_1'), (a_2, a_2'), … : $a_i \in$ Abschluß, $a_i' \in$ Ausdruck }

Die Funktion zuordnungsliste *expandiert* ein Nichtterminal zu einer Zuordnungsliste. Zu jedem Nichtterminal steht im Gedächtnis ein Redukt (s.o.). Dieses bildet die rechte Seite des Nichtterminals und mit diesem zusammen die erste Zuordnung des Resultats. Weitere Zuordnungen ergeben sich durch den rekursiven Aufruf der Funktion für die Nichtterminal-Vorkommen in der genannten rechten Seite.

---

[60] Bei der Ausgabe wird die Zuordnungsliste pro forma als let-Ausdruck dargestellt, um ihm einen „Sinn“ in der Prolid-Syntax zu geben.

[61] Die Verwaltung der quasi „globalen“ Variablen gedächtnis (s.u.) ist rein funktional nämlich recht schlecht darstellbar, vgl. die Implementation.

[62] Wir verwenden die Begriffe „Redex“ und „Redukt“ informell im Sinne von „unausgewerteter“ bzw. „ausgewerteter Ausdruck“.

zuordnungsliste : Variable → Zuordnungsliste
zuordnungsliste (Nichtterminal (a, u)) begin
return (
{ Nichtterminal(a, u) = a' } ∪ ⋃(zuordnungsliste(nt) : nt ∈ variablen(a'))
wobei a' :⇔ ((a, u), a') ∈ gedächtnis)
end zuordnungsliste

Das Resultat des Aufrufs der Funktion evaluation für den Rumpf der Variablen „main" ist die Semantik eines Prolid-Programms.
evaluation wertet das Argument zu einer Zuordnung des Nichtterminals „n_0" (dem Axiom der Zuordnungsliste) und seiner rechten Seite sr aus. Durch die (ggfs. sich rekursiv fortsetzende) Expansion aller Nichtterminale in sr durch die Hilfsfunktion zuordnungsliste erhält man alle Zuordnungen von Nichtterminalen, die vom Axiom aus erreichbar sind:

evaluation : Ausdruck → Zuordnungsliste

evaluation (a) begin

gedächtnis := { }
sr := evaluationNf (a, ({ }, { }), ( ))
return ({ „n_0" = sr } ∪ ⋃(zuordnungsliste(nt) : nt ∈ variablen(sr)))

end evaluation

evaluationNf[63] ist die rekursive Version des Verfahrens 3.3.3.
evaluationNf : Ausdruck × Umgebung × Argumente → Ausdruck

evaluationNf (a, (su, ou), (arg_1, … ,arg_n)) begin

es wird geprüft, ob das Redex schon im Gedächtnis enthalten ist („Rekursionserkennung"):
if ∃a'∈ Ausdruck : ((a, (su, ou)), a') ∈ gedächtnis then begin

[63] Das Resultat ist keine Normalform im Sinne eines Grundterms, weil das Resultat aber nicht mehr weiter reduziert wird, verwenden wir diesen Begriff bzw. die Abkürzung Nf.

wenn ja, liefern wir ein Nichtterminal zurück, daß zu genau dem Redex gehörig ist (das Redex (a, (su, ou)) ist sein Variablenname):

return (Nichtterminal (a, (su, ou)))

end

wir berechnen die indefinite Kopf-Normalform (HNfSet) des Redex:

hnfs := evaluationHNf (a, (su, ou), (arg_1, …,arg_n)))

wir benötigen die einzelnen Bestandteile von hnfs:

{ ((k_i, ou_i), arg_{i1}, …, $arg_{in(i)}$)$_i$: $1 \le i \le |hnfs|$} := hnfs

{(a_{i1}, (su_{i1}, ou_{i1})), …,($a_{in(i)}$, ($su_{in(i)}$, $ou_{in(i)}$)) : $1 \le i \le |hnfs|$} := {arg_{i1}, …, $arg_{in(i)}$: $1 \le i \le |hnfs|$}

if hnfs = { } then begin *(wir haben das leere Or-Set erhalten)*

return (impossible)

end

wir merken uns das Redex und (eine Referenz auf) sein Redukt a' (das wir erst im nächsten Schritt tatsächlich berechnen, was in unserer Lazy-Sprache aber kein Problem ist):

gedächtnis := { ((a, (su, ou)), a') } ∪ gedächtnis

…und fahren mit der Reduktion der Argumente der einzelnen Kopf-Normalformen fort:

a' := generateOrSet ({

k_i evaluationNf(arg_{i1}, (su_{i1}, ou ∪ ou_i ∪ ou_{i1}), ( ))

… evaluationNf($arg_{in(i)}$, ($su_{in(i)}$, ou ∪ ou_i ∪ $ou_{in(i)}$), ( )) : $1 \le i \le |hnfs|$ })

return a'

end evaluationNf

evaluationNf hat für die tatsächliche Implementierung eine Verbesserung erfahren. Diese verbesserte Version von evaluationNf geben wir nicht an, da sie deutlich komplexer als das dargestellte Verfahren ist. Sie befindet sich im Anhang C.

Verfahren 4.7.2.3 (Berechnung der indefiniten Kopf-Normalform eines Ausdrucks):

Die Grundidee dieser Funktion ist, jeden Higher-order-Ausdruck mit einer Liste seiner Argumente aufzurufen: Für jede Applikation evaluationHNf rekursiv aufzurufen und dabei das Argument der Applikation an die bisherige Argumente-Liste anzuhängen und an die neue Inkarnation von evaluationHNf zu übergeben (das sogenannte „unwinding the spline“). Auf diese Weise erhält man alle Argumente eines curried-notierten Funktionsaufrufs. Um „lazy“ auszuwerten, d.h. die Argumente nicht vor dem rekursiven Aufruf von evaluationHNf evaluieren zu müssen, handelt es sich bei den Argumenten um Abschlüsse.

evaluationHNf : Ausdruck × Umgebung × Argumente → HNfSet

Haben wir einen Konstruktor als Kopf eines Ausdrucks erreicht, so sind wir fertig:
evaluationHNf (k : k ∈ Konstruktor, (su, ou), (arg_1, …,arg_n))
→ { ((k, ou), arg_1, …, arg_n) }

evaluationHNf (Applikation (x, y), u, (arg_1, …,arg_n)) *(Siehe oben)*
→ evaluationHNf (x, u, ((y, u), arg_1, …,arg_n))

Bei der Auswertung eines Bestimmten Or-Sets wird zuerst überprüft, ob nur eine seiner beiden Alternativen gültig ist. Dies ist dann der Fall, wenn der Index des Or-Ausdrucks in der aktuellen Or-Set-Umgebung enthalten ist. Andernfalls wird der Or-Ausdruck distribuiert, d.h. seine linke Alternative wird in der erweiterten Or-Set-Umgebung ausgewertet, bei der genau dieses Alternative als gültig ausgewiesen wird (analog für die rechte Alternative):
evaluationHNf (or_i (alt_A, alt_B), (su, ou), (arg_1, …,arg_n))
| falls (i, L) ∈ ou → evaluationHNf (alt_A, (su, ou), (arg_1, …,arg_n))
| falls (i, R) ∈ ou → evaluationHNf (alt_B, (su, ou), (arg_1, …,arg_n))
| sonst →
evaluationHNf (alt_A, u_A, (arg_1, …,arg_n)) ∪ evaluationHNf (alt_B, u_B, (arg_1, …,arg_n)) :
u_A := (su, { (i, L) } ∪ ou), u_B := (su, { (i, R) } ∪ ou)

Für ein Unbestimmtes Or-Set ist dieser Aufwand nicht notwendig:
evaluationHNf (or* (alt_A, alt_B), u, (arg_1, …,arg_n)) →
evaluationHNf (alt_A, u, (arg_1, …,arg_n)) ∪ evaluationHNf (alt_B, u, (arg_1, …,arg_n))

Ist eine Variable in der aktuellen Substitutionsumgebung enthalten, wird ihr Inhalt ausgewertet. Andernfalls handelt es sich um eine Query-Variable. Wir geben hier die Variable unausgewertet

zurück, was die Generierung von Constraints auslöst, falls evaluationHNf für den testausdruck einer verzweigung aufgerufen wurde:

evaluationHNf (var: var ∈ Variable, (su, ou), (arg_1, …,arg_n))

| (∃a, u_v: (a, u_v) ∈ Abschluß, (var, (a, u_v)) ∈ su) → evaluationHNf (a, u_v, (arg_1, …,arg_n))

| n = 0 ∧ (var = k, k ∈ Konstruktor) ∈ ou → { ((k, ou)) } *(hier ist die Query-Variable sicher definit und kann zu k verkürzt werden)*

| n = 0 → { ((var, ou)) } (*initiiert das Narrowing. evaluationHNf(var) ist allerdings undefiniert für n > 0, falls es keine Substitution für var gibt, vgl. die Definition der Funktion constrainedMatch)*

es wird der Rumpf einer Funktion aufgerufen. Dabei werden den Parameter-Variablen x_1 ,…,$x_{arität}$ die angesammelten Argumente zugewiesen.

evaluationHNf (λ (x_1 ,…,$x_{arität}$) → r, (su, ou), (arg_1, …,arg_n))

| falls n = arität → evaluationHNf (r, ({ (x_i, arg_i) : 1≤i≤n }, ou), ( ))

Falls zu wenige Argumente zur Verfügung stehen, geben wir einen Abschluß zurück:

| falls n < arität →

(λ (x_1 ,…,$x_{arität}$) → r) evaluationHNf (arg_1, umgebung(arg_1), ( ))

… evaluationHNf (arg_n, umgebung(arg_n), ( ))

Bei einer Higher-order-Bindung für die Funktion werden nicht alle aktuellen Argumente „aufgebraucht“:

| falls n > arität →

evaluationHNf (r, ({ (x_i, arg_i) : 1≤i≤arität }, ou), ($arg_{arität+1}$, …, x_n))

Für eine Verzweigung wertet man den testausdruck t zur indefiniten Kopf-Normalform aus und match die einzelnen Köpfe gegen das Muster. Hinterher werden für die jeweiligen Resultate die Zweige a (bei erfolgreichem Matching) und/oder b (sonst) ausgewertet:

evaluationHNf (Verzweigung (t, m, a, b), (su, ou), (arg_1, …,arg_n))

| falls m ∈ Variable → evaluationHNf (a, ({ (m, t) } ∪ su, ou), (arg_1, …,arg_n))

| falls m = ⊥ → { ((⊥, {})) }

| falls m = impossible → { ( ) }

| sonst → (mit hnfset := evaluationHNf (t, (su, ou), ( ))

{ evaluationHNf (a, (su ∪ su_A, ou ∪ ou_A), (arg_1, …,arg_n))

: (su_A, ou_A) ∈ match(m, hnfset) }

∪ { evaluationHNf (b, (su, ou_B), (arg_1, …,arg_n))$_i$

: ou_B ∈ mismatch(m, hnfset) }

$\cup$ { evaluationHNf (a, (su, ou $\cup$ ou_A), (arg_1, …,arg_n))

: $ou_A \in$ constrainedMatch(m, ou, hnfset) }

$\cup$ { evaluationHNf (a, (su, ou $\cup$ ou_A), (arg_1, …,arg_n))

: $ou_A \in$ constrainedMismatch(m, ou, hnfset) }

[$\cup$ { ((⊥, {})) } falls ((⊥, ..), ..) $\in$ hnfset]

evaluationHNf (impossible, (su, ou), (arg_1, …,arg_n)) $\rightarrow$ { ( ) }

evaluationHNf (⊥, (su, ou), (arg_1, …,arg_n)) $\rightarrow$ { ((⊥, { })) }

5 Graphische Darstellung von Resultaten mit dem Interactor

Die graphische Darstellung der Zuordnungslisten, die wir als Resultate der Auswertung von Prolid-Programmen erhalten, stellt hohe Anforderungen an eine Graphik-Software. Zuordnungslisten können z.B. Zyklen enthalten. In der Haskell-Welt ist *daVinc*i für die Darstellung von Graphen die wohl meistgebrauchte Software. DaVinci läuft als eigenständiger Prozeß parallel zu dem Haskell-Programm, daß die Knoten des Graphen generiert. DaVinci ist mit diesem Programm über Pipes verbunden und nimmt über diese Verbindung Darstellungskommandos entgegen. Während die Idee, die Darstellungs-Software als eigenständiges Programm auszuführen, so interessant ist, daß sie für diese Arbeit übernommen wurde, erschien die imperative Natur der daVinci-Schnittstelle als zu unflexibel. Graphen liegen als Haskell-Strukturen vor, werden dann in Befehlssequenzen „zerlegt“ und von daVinci wieder zusammengesetzt und dargestellt. Da Graphen in Gestalt von Zuordnungslisten generische Resultate von Prolid-Programmen sind, erschien es sinnvoller, Graphen „als ganzes“ analog zu sonstigen Berechnungsresultaten (endlichen Grundtermen und Or-Sets) an die Darstellungssoftware zu übermitteln. Es soll bei der Darstellung keinen prinzipiellen Unterschied zwischen algebraischen Datentypen, Or-Sets und Graphen geben. Es soll vielmehr möglich sein, im Prinzip *jedes* Berechnungsresultat weitgehend automatisch darzustellen. Dieser gewissermaßen „ganzheitliche“ Ansatz führt direkt zur objektorientierten Denkweise: Um von der konkreten Natur der Berechnungsresultate zu abstrahieren, werden sie aus Sicht der Darstellungssoftware abstrahierend als *darstellbare Objekte* betrachtet. Alle Objekte müssen über gemeinsame Eigenschaften verfügen, die ihre Darstellbarkeit ausmachen. Um dies zu gewährleisten, werden Berechnungsresultate in eine solche Objektform „transformiert“. Da es sich um Prolid-Objekte handelt, die von einem externen Programm dargstellt werden sollen, nennen wir sie *externe Objekte.* Dazu gibt es einen Prolid-Datentyp `ExternalObject`, in den die Resultate der Auswertung, also Grundterme, Or-Sets oder Zuordnungslisten, durch Prolid-Funktionen überführt werden. Jedes Objekt dieses Typs repräsentiert ein graphisches Java-Objekt. Die Darstellungssoftware muß sich auf diese Weise nicht mehr um die ursprüngliche Natur des Objekts kümmern. Der große praktische Nutzen dieser „funktional-objektorientierten“ Programmierung liegt darin, daß auf der „funktionalen Seite“ keinerlei neue Sprachelemente wie z.B. Monaden notwendig sind, um externe graphische Darstellungen objektorientiert generieren zu können. `ExternalObject` ist ein normaler algebraischer Datentyp. Trotzdem repräsentiert jeder Wert dieses Typs genau den darzustellenden Wert (z.B. eines algebraischen Datentyps) *und gleichzeitig* genau ein graphisches Java-Objekt.

Die Entscheidung, statt proprietären Darstellungskomponenten Java bzw. Java-Klassenbibliotheken zu verwenden, hat außerdem die Vorteile, daß die „darstellerischen Fähigkeiten“ von Prolid unbegrenzt erweitert werden können und unter fast jedem

Computersystem zur Verfügung stehen. Das konkrete Aussehen der graphischen Darstellungen wird allein auf der Java-Seite durch die vorhandenen Klassenbibliotheken festgelegt und kann jederzeit geändert werden, ohne auf der Prolid-Seite Änderungen vornehmen zu müssen. Umgekehrt kümmert sich die Java-Seite lediglich um die Darstellung; um die funktionale Natur der darzustellenden Objekte braucht sie sich nicht zu kümmern. Der Nachteil speziell gegenüber daVinci ist allerdings, daß daVinci hochspezialisiert und weit entwickelt ist.

5.1 Funktionale Generierung graphischer Komponenten

Der Typ `ExternalObject` ist im Modul `Stdlib` folgendermaßen deklariert. Jeder Wert vom Typ `ExternalObject` heißt *externes Objekt.*

```
type Class [Char]
type Method [Char]

data ExternalObject
  = ExternalMethod ExternalObject Method [ExternalObject]
  | ExternalGeneric gt
  | ExternalStatic Class Method [ExternalObject]
```

Jeder Wert dieses Typs repräsentiert ein Java-Objekt[64]. Das Java-Objekt wird entweder durch die Java-Methode beschrieben, die es berechnet, oder es handelt sich um ein sogenanntes *primitives Objekt*. Primitive Objekte sind Objekte von Klassen wie z.B. `Integer` oder `Character`, die in Java als Äquivalent zu einem primitiven Typ wie z.B. `int` oder `char` fungieren. Obwohl man mit dem Typ `ExternalObject` im Prinzip jedes Java-Objekt darstellen könnte, wollen wir uns in dieser Arbeit auf primitive Objekte und graphische Komponenten der Klasse `java.awt.Component` beschränken.

Bei Methodenaufrufen wird zwischen Aufrufen statischer Methoden (z.B. von Java-Konstruktoren) und sonstigen Methoden unterschieden:

Resultate statischer Methodenaufrufe werden durch `ExternalStatic k m` a konstruiert. Ist m der leere String, so wird der Konstruktor der Klasse aufgerufen. k muß die vollständig

---

[64] Prolid ist im Prinzip nicht auf Java festgelegt; die beschriebenen Konzepte funktionieren im Prinzip immer, falls sich die Objekte der externen Programmiersprache durch algebraische Datentypen sinnvoll repräsentieren lassen. Weil aber Java eine (fast) ideale objektorientierte Sprache ist und der Interactor ein Java-Programm, verwendet diese Arbeit Java als Referenzsprache.

qualifizierte Bezeichnung der Klasse sein (z.B. "`java.awt.Label`" statt nur "`Label`")[65]. a ist eine Liste von Werten des Typs `ExternalObject`. Die Elemente der Liste werden beim Methodenaufruf als Argumente übergeben.

Soll das Java-Objekt durch eine nicht-statische Methode m eines Java-Objekts o generiert werden, so ist das externe Objekt durch `ExternalMethod` o m a zu konstruieren. a ist wieder eine Liste mit den Argumenten des Methodenaufrufs.

Für primitive Objekte ist das externe Objekt mit `ExternalGeneric p` zu konstruieren. Bei p darf es sich um Werte der Prolid-Typen `Int`, `Char` und `String` handeln. Vom Interactor werden aus diesen Werten Java-Objekte der Klassen `Integer`, `Character` und `String` erzeugt.

Externe Objekte sollen nun dazu dienen, die Resultate funktionaler Berechnungen darzustellen. Wie genau die externen Objekte konstruiert werden, hängt natürlich von der Art des darzustellenden Resultats ab (siehe 5.2 und 5.3). In jedem Fall liegt jedoch nach der Konstruktion genau ein Wert vom Typ `ExternalObject` vor (ggfs. mit Unterobjekten), der genau eine Java-Objekt der (Ober-)Klasse `java.awt.Component` repräsentiert, oder ein Or-Set solcher Objekte.

Um externe Objekte darstellen zu können, wird das Java-Programm *Interactor* aufgerufen (`Interactor.java`). Das Programm setzt Java® 1.1.3 oder höher und die Java Foundation Classes von Sun® und Netscape® („JFC", ab Version 0.6) voraus. Von den JFC werden die Teil-APIs „Swing" (graphische Komponenten in Form von Beans, in der Klassenhierarchie basierend auf dem Java-AWT) und „Reflect" (siehe unten) verwendet[66]. Der Interactor startet wiederum den Prolid-Interpreter in einem eigenen Kind-Prozeß. Interactor und Interpreter kommunizieren über die Standard-Ein- und Ausgabe: Die Resultate des Interpreters in Form von `ExternalObject`-Grundtermen werden als Strings im Quellcode-Format an den Interactor gesandt. Der Interactor wiederum schickt Befehle an die Kommandoschnittstelle des Interpreters (etwa „Lade Modul x !"), was in diesem Zusammenhang aber keine Rolle spielen soll[67]. Der Interactor kann mit beliebig viele Interpreter-Prozessen zugleich verwalten. Für jeden Prozeß

---

[65] Der einfache Grund dafür ist, daß es keine `package`- und `import`-Statements gibt.

[66] Die Übersetzung des Interactor-Quellcodes, die Installation, der Aufruf und die Bedienung sind in Anhang B beschrieben.

[67] Die Übermittlung von externen Objekten als String-Darstellung von Termen ist natürlich nicht ideal; für eine Weiterentwicklung von Prolid sollte eine binäres Tokenformat entwickelt werden. Eine weitere Effizienzsteigerung könnte erzielt werden, wenn Interpreter und Interactor ihre Daten nicht über die Standard-Ein-/Ausgabe des Betriebssystems austauschen müßten, sondern dies z.B. über einen gemeinsamen, vom Betriebssystem zugewiesenen Speicherbereich tun könnten.

wird ein eigenes Client-Fenster geöffnet, das die Konsole des Interpreters enthält, und eine Fläche zur Anzeige der externen Objekte (Klassen `InteractionFrame` und `InteractionPanel`). Die Konsolen nehmen asynchron Tastatureingaben für den Interpreter entgegen und stellen die Prolid-Resultate textuell dar (Klasse `InterpreterInputFrame`).
Der vom Interactor jeweils übermittelte Term wird nun geparst. Dabei werden die vom Wert des Typs `ExternalObject` repräsentierten Methodenaufrufe ausgeführt und so das bzw. die graphische(-n) Java-Objekt(-e) erstellt. Der Parser wurde vom Java-Parsergenerator *JavaCC* aus der Spezifikationsdatei `ProlidExternalParser.jj` generiert[68]. Die Methodenaufrufe werden mit Hilfe der seit Java 1.1 vorhandenen Reflect-API ausgeführt. Diese Bibliothek erlaubt es, Methoden von Klassen und Objekten aufzurufen, die zur Laufzeit ermittelt wurden (siehe dazu insbesondere die Funktion `findMethod` in `ProlidExternalParser.jj`). Eine Schwierigkeit hierbei besteht darin, daß die Methoden anhand der exakten Typen ihrer formalen Parameter bestimmt werden. Ist ein Parameter mit als der Klasse T zugehörig deklariert worden, so findet die gegenwärtige (Beta-)Version der Reflect-API nicht die richtige Methode, falls eine Unterklasse von T angegeben wird. Daher prüft `findMethod` in einer Schleife ggfs. auch alle Oberklassen der im externen Objekt spezifizierten Klassen.
Wird ein Or-Set geparst, wo wird daraus ein Objekt der vordefinierten Klasse `OrSetView` erzeugt (siehe 5.4). Für Or-Sets müssen also keine externen Objekte generiert werden (dies wäre auch gar nicht möglich, da Or-Sets in Prolid ja keine First-class-Objekte sind)[69]. Der Aufruf der paint-Methode der letztendlich generierten Komponente finden in der Funktion `view` der Klasse `InteractionFrame` statt.

5.1.1 Darstellung algebraischer Datentypen

Zwischen algebraischen Datentypen und der Java-Klasse `Component` (die Wurzel der Hierarchie der graphischen Klassen des *Abstract Window Toolkit* (AWT)) gibt es eine interessante Gemeinsamkeit: Grundterme[70] eines algebraischen Datentyps können ebenso wie Objekte der Klasse `Component` (genannt *Komponenten*) als *Bäume* angesehen werden. Nach 2.1.2.3 gilt: Für Typen sind die Knoten mit den Konstruktor-Vorkommen des Terms markiert.

---

[68] Prolid-Resultate werden mittels eines Pretty-Printers erzeugt (Funktion `prettyExpression` im Modul `Pretty`); der Pretty-Printer und die Parser-Spezifikation müssen aneinander angepaßt sein.

[69] Die Klasse `OrSetView` kann aber selbstverständlich an spezielle Bedürfnisse angepaßt werden, vgl. 5.3.1.

[70] Partielle Konstruktor-Applikationen und andere ggfs. definite Evaluations-Resultate als Grundterme (z.B. sonstige Closures oder Bottom) werden nicht betrachtet.

Die Argumente eines Konstruktor-Vorkommens bilden als Teilbäume die Söhne seines Knotens, die nullstelligen Konstruktoren sind die Blätter des Baums. Für die Klasse Component sind die Knoten die Objekte dieser Klasse, die in einer gemeinsamen *Containerhierarchie* enthalten sind. Unter einem *Container* versteht man in der objektorientierten Graphikprogrammierung eine Klasse graphischer Objekte, die andere graphische Objekte visuell enthalten können (die Komponenten des Containers). Beispiele für Container sind *Fenster* oder *Icon-Leisten*. Die in einem Container enthaltenen Komponenten (die ihrerseits Komponenten enthalten können) bilden besagte Containerhierarchie. Atomare Komponenten (z.B. *Labels* oder *Buttons*) oder leere Container sind die Blätter des Baums. Wir nennen derartige Bäume *und* ihre Repräsentation als externe Objekte vom Typ `ExternalObject` *Komponentenbäume*. Die Grundterme heißen *Werte-Bäume*. Ausdrücklich erlaubt sind unendliche Bäume (auch auf Java-Seite!, dazu später mehr). Die Idee für die Darstellung algebraischer Datentypen ist es nun, eine rekursive Funktion zu definieren, die eine Zuordnung genau eines Grundterms zu genau einer geeigneten graphischen Komponente vornimmt, also einen Werte-Baum auf einen Komponentenbaum abbildet. Jedem Konstruktor wird ein Objekt einer dem Konstruktor-Typ entsprechenden Java-Klasse zugeordnet. Hat ein Konstruktor Parameter, so ist die Darstellungsklasse i.a. ein Container. Es wird für jedes Argument des Konstruktors die Darstellungsfunktion rekursiv aufgerufen, und die dabei erzeugten Objekte werden als Komponenten in den Container des Konstruktors aufgenommen[71].

In Prolid gibt es zwei solche Abbildungsfunktionen, die beide die generierten graphischen Komponenten mit der unter 5.1 beschriebenen Technik durch Objekte vom Typ `ExternalObject` repräsentieren: Die überladenen Funktionen `view` und `defaultView`[72]. `defaultView` ist eine automatisch generierte Funktion, siehe 5.2.1[73]. Eine Instanz von `view` wird vom Programmierer für jeden Typ t bereitgestellt, dessen Werte dargestellt werden sollen. Der Typ einer Instanz von `view` ist somit `t -> ExternalObject`. Auf der Java-Seite müssen natürlich Klassen für die externen Objekte definiert werden. Die „Swing“-API bietet dafür eine gute Grundlage. Im Rahmen dieser Arbeit wurden zur Demonstration verschiedene

---

[71] Auch der umgekehrte Weg, d.h. die graphische *Eingabe* von Werten algebraischer Typen, wäre mithilfe dieses Prinzips denkbar: Man konstruiert (z.B. mit einem *GUI-Builder*) den Komponenten-Baum und ordnet dann, rekursiv absteigend, jedem Container bzw. den Blatt-Komponenten einen Konstruktor entsprechend der jeweiligen Komponenten-Klasse zu. Man muß lediglich darauf auchten, daß diese Zuordnung eindeutig ist. Dies erreicht man durch eine zur Menge der algebraischen Typen isomorphe Klassenhierarchie.

[72] Zur Technik der Ad-hoc-Überladung siehe 4.

[73] Die Bezeichner `view` und `defaultView` werden vom Interactor verwendet, um sie als „Darstellungskommandos“ an den Interpreter zu schicken. Ansonsten sind diese Namen willkürlich und nicht etwa reservierte Bezeichner. Es handelt sich um normale Prolid-Funktionen, die keinerlei besondere Sprachelemente verwenden, sondern genauso auch z.B. in Haskell erstellt werden könnten.

`view`-Instanzen und die zugehörigen Java-Klassen erstellt. Diese Beispiele befinden sich im Modul `Main.pro` und sind dort auch kommentiert. Die Swing-Dokumentation ist unter [23] zu finden. Da es an dieser Stelle unmöglich ist, eine Einführung in die Swing-Architektur zu geben, sollte diese konsultiert werden, um ein genaues Verständnis für den Code der im Rahmen dieser Arbeit erstellten Java-Klassen zu bekommen.

Übersicht über die neu entwickelten Java-Klassen zur Darstellung von Prolid-Resultaten:

Klasse	*Zweck*	*unmittelbare Oberklasse bzw. Interface*
`BasicListView`	Listendarstellung von `ToggleComponent`-Elementen	[…].`swing.JList`
`BoolView`	An-/Aus-Schalter zur Darstellung eines Wahrheitswertes. Es gibt allerdings z.Zt. keine Möglichkeit, den Schalter sinnvoll zu bedienen…	[…].`swing.JCheckBox`
`ConsView`	Darstellung einer Liste durch eine Komponente für den Listenkopf und eine für den Rumpf (wieder ein `ConsView`). Dies entspricht der dekonstruierten Form einer Prolid-Liste.	[…].`swing.JPanel`
`GenericList`	Modell-Liste (Daten) beliebiger Objekte (Hilfsklasse)	[…].`util.Vector`
`GenericNode`	Modell eines `TreeViews`	[…].`swing.tree.TreeNode`
`GenericPair`	Modell-Paar zweier beliebiger Objekte (Hilfsklasse)	[…].`Object`
`GraphView`	Darstellung von Zuordnungslisten durch zyklische Graphen (siehe 4.7.3)	[…].`swing.JPanel`

`OrSetView`	Listendarstellung eines endlichen Or-Sets	`BasicListView`
`ReferenceNode`	Modell eines `ReferenceView`s (Wrapper-Klasse, um `ReferenceView`s als Knoten in `TreeView`s aufnehmen zu können und so den `TreeView` in einen zyklischen Graphen der Klasse `GraphView` integrieren zu können)	`GenericNode`
`ReferenceView`	Visuelle Referenz (vulgo „Pfeil“) auf eine Komponente innerhalb eines `GraphView`	`[…].swing.JLabel` (die `paint`-Methode wird überschrieben und erweitert)
`TextListView`	konventionelle Listendarstellung als Tabelle von Textreihen (eine Alternative zu `ConsView`, allerdings für „funktionale“ Listen weniger gut geeignet, da man kein Pattern mit Variablen für alle Elemente einer beliebig langen Liste erzeugen kann)	`[…].awt.List`
`ToggleComponent`	Stellt eine beliebige graphische Komponente dar mit der Möglichkeit, sie auf Mausklick hin zu verbergen	`[…].swing.JPanel`
`TreeView`	Baumdarstellung. Zugehörige Modell-Klasse ist `GenericNode` (und mittelbar ein algebraischer Datentyp, vgl. 4.7.1)	`[…].swing.JTree`

Ein Problem tritt bei der Erstellung von `view`-Instanzen für einen parametrisierten Typ t auf. Im Rumpf der entsprechenden Bindung v_t muß i.a. die `view`-Instanz v_p für den Parameter-Typ t_p aufgerufen werden. Ist v ein Aufruf von `view` im Rumpf von v_t, so wird für das Argument von v eine allquantifizierte Typvariable inferiert. Somit kann v nicht statisch durch die korrekte Instanz

v_p ersetzt werden. Eine Umgehung dieses Problems besteht darin, mittels einer Typ-Signatur für v_t den Typ des Arguments von v zu verengen.

Beispiel: Folgende `view`-Instanz stellt eine einzelne Zeichenkette dar:

```
view :: String -> ExternalObject

view string =
  ExternalStatic "com.sun.java.swing.JLabel" ""
  [ExternalGeneric string]
```

(1) Folgende Bindung soll Listen von beliebigen Objekten darstellen, funktioniert aber nicht wie erwartet, denn die Überladung des Aufrufs `view hd` kann nicht statisch aufgelöst werden:

```
view :: [t] -> ExternalObject

view []      = ExternalStatic "BasicListView" "" []
view (hd:tl) = ExternalMethod (view tl) "addElem" [view hd]
```

(2) ersetzt (1) durch eine Lösung, die zumindest für Listen von Zeichenketten funktioniert:

```
view :: [String] -> ExternalObject

view []      = ExternalStatic "BasicListView" "" []
view (hd:tl) = ExternalMethod (view tl) "addElem" [view hd]
```

5.1.2 Automatisch generierte Darstellungen

Stellt man an die Darstellung keine besonderen Ansprüche, so können Instanzen der Darstellungsfunktion `defaultView` auch automatisch aus den `data`-Deklarationen von nicht-parametrisierten Typen generiert werden. Die Generierung erfolgt statisch direkt nach dem Laden eines Moduls in den Interpreter. Der Interpreter muß dazu ohne den Schalter `-n` aufgerufen werden (vgl. Anhang). Die Beschränkung auf nicht-parametrisierte Typen ist deshalb notwendig, weil i.a. sonst die entstehenden Überladungen der Darstellungsfunktion nicht statisch aufgelöst werden können (vgl. 5.2). Man darf für parametrisierte Typen aber die Menge der automatisch generierten Instanzen von `defaultView` manuell um weitere Instanzen ergänzen, siehe unten.

Aus einer `data`-Deklaration der Form `data` t = K_1 t_{11} … t_{1k1} | … | K_n t_{n1} … t_{nkn} wird durch die Haskell-Funktion `Static.generateDefaultViews` eine Deklarationsgruppe von `defaultView` der folgenden Form generiert :

```
defaultView :: t -> ExternalObject

defaultView K1 x11 … x1k1 =
  ExternalStatic "GenericNode"  ""
  [ExternalGeneric "K1",
  vector( [defaultView x11, …,defaultView x1k1] )]
…
defaultView Kn xn1 … xnkn =
  ExternalStatic "GenericNode"  ""
  [ExternalGeneric "Kn",
  vector( [defaultView xn1, …,defaultView xnkn] )]
```

Jeder Aufruf von `defaultView` erzeugt einen Komponentenbaum wie in 5.2 beschrieben (bzw. ein Or-Set solcher Bäume). Jeder Knoten besteht aus einem Container der Klasse `GenericNode`. Die Argumente-Liste, die bei jedem Aufruf des Konstruktors dieser Java-Klasse angegeben ist, enthält als erstes Element den Namen des jeweiligen Konstruktors (als Markierung des Knotens). Das zweite Argument ist ein Java-Vektor, bestehend aus Unterbäumen, generiert durch Instanzen von `defaultView` für die Parameter-Typen t_{i1} … t_{iki}[74].

`GenericNode` ist von der Swing-Klasse `TreeNode` abgeleitet[75].

---

[74] vector ([…]) steht für ein `ExternalObject`, daß ein Java-Objekt der Klasse `java.util.Vector` repräsentiert. Man könnte die `defaultView`s der Argumente des Konstruktors K_i auch als Prolid-Liste übergeben, dann müßte man aber für jede Listen-Länge einen eigenen Objekt-Konstruktor der Klasse `GenericNode` bereitstellen.

[75] Obwohl GenericNode eigentlich eine Modell-Klasse ist, also nur „Daten“ und nicht deren Darstellung repräsentiert, kommt `defaultView` mit der Erzeugung von Objekten dieser Klasse aus. Um die Verknüpfung der `GenericNode`-Objekte zu einem Baum zu erleichtern, wird nämlich der Wurzelknoten dieses Modell-Baums erst nach seiner Fertigstellung in das zugehörige Darstellungsobjekt der Klasse `TreeView` umgewandelt.

Beispiel für die semi-automatische Darstellung von parametrisierten („polymorphen“) Typen durch manuelle Ergänzung der Darstellungsfunktion (*Und-Oder-Bäume* sind n-äre Bäume, deren innere Knoten entweder mit „And“ oder mit „Or“ markiert sind, und deren Blätter mit Zeichenketten) :

```
defaultView :: String -> ExternalObject
defaultView string =
  ExternalStatic "GenericNode" "" [ExternalGeneric string,
  ExternalStatic "GenericList" "" []]

data AndOrTree = Leaf String | And [AndOrTree] | Or [AndOrTree]

defaultView :: [AndOrTree] -> ExternalObject
defaultView nodes =
  ExternalStatic "GenericNode" "" [ExternalGeneric  "Kinder",
                                   andOrNodeList nodes]

andOrNodeList :: [AndOrTree] -> ExternalObject
andOrNodeList [] = ExternalStatic "GenericList" "" []
andOrNodeList (h:t) =
  ExternalMethod (andOrNodeList t) "addElem" [defaultView h]
```

Hier kann für allgemeine Listen die `defaultView`-Instanz nicht automatisch erstellt werden. Der Ausweg ist, für spezielle Listen (hier `String` und `[AndOrTree]`) Instanzen vorzugeben. Da dann der Typ der Listenelemente bekannt ist, kann sogar der überladene Aufruf `defaultView h` im Rumpf der Hilfsfunktion `andOrNodeList` aufgelöst werden.

5.2 Darstellung von Or-Sets

Eine rechte Seite einer Zuordnung der Form a_1 `or*` ... `or*` a_n wird als Liste der Alternativen a_i dargestellt. Die einzelnen Alternativen werden mittels der vorangehend beschriebenen Technik angezeigt. Eine leeres Or-Set wird durch den Text „impossible“ angezeigt. Die Java-Klasse zur Darstellung mehrelementiger Or-Sets heißt `OrSetView` und ist direkt von der Klasse `BasicListView` abgeleitet (siehe 4.7).

5.3 Darstellung von Zuordnungslisten

Der Interpreter evaluiert einen Ausdruck a zu einer Zuordnungsliste. Ist der semantische Wert von a ein endliches Or-Set, und sind alle Alternativen des Or-Sets endliche Grundterme, so kann die Zuordnungsliste durch das externe Objekt `view` a (bzw. `defaultView` a) graphisch dargestellt werden, falls sich der Wert von a durch den Interactor graphisch darstellen läßt[76], denn der semantische Wert des `let`-Ausdrucks, als der die Zuordnungsliste dargestellt wird, ist immer der von a. In diesem Fall würde es sich bei dem externen Objekt um einen Komponentenbaum handeln (bzw. mehrere Komponentenbäume, falls der Wert von a ein Or-Set mit mehr als einem Element ist).
Besser geeignet zur Darstellung von Zuordnungslisten sind Darstellungen von Graphen, da sich so auch unendliche Or-Sets und unendliche Grundterme darstellen lassen. Da der durch `:fi` a produzierte `let`-Ausdruck mutuell rekursiv ist, und die rechte Seite rs_i jeder der n Zuordnungen $nt_i \equiv rs_i$ ein Grundterm ist, wenn man die Vorkommnisse von Nichtterminalen in rs_i als Konstruktoren auffaßt, stellt die Zuordnungsliste unter dieser Voraussetzung einen zusammenhängenden, gerichteten Graphen dar[77]: Die rechte Seite rs_i einer Zuordnung $nt_i \equiv rs_i$ ist ein Werte-Baum rsb_i, falls man das Vorkommnis eines Nichtterminals in rs_i als Konstruktor-Blatt ansieht. Man vereinigt nun die rsb_i und fügt für jedes Vorkommnis r_{ij} eines Nichterminals nt_i in jeder rechten Seite rs_i eine Kante (r_{ij}, rsb_i) zum Graphen hinzu (diese Kanten nennen wir *Referenzen*, die r_{ij} *Referenzknoten*). Derart konstruierte Graphen nennen wir *Produktionengraphen*[78] (einer Zuordnungsliste bzw. Der korrespondierenden Grammatik). Repräsentieren zwei Produktionengraphen dasselbe Or-Set, so heißen sie *äquivalent*.

In dieser Arbeit wurden drei Arten der Darstellung von Produktionengraphen entwickelt:

(1) Darstellung eines *äquivalenten* Produktionengraphen als ggfs. zyklischer Graph, wobei das externe Objekt allein mit den Spracheelementen von Prolid generiert wird
(2) Optimale, direkte Darstellung als ggfs. zyklischer Graph durch einen Eingriff in den Auswertungsmechanismus für die `view-` und die `defaultView`-Funktion
(3) Wie (1) bzw. (2), aber Darstellung des externen Objekts durch Java-Klassen, die unendliche Grundterme als (ggfs. unendlich tiefen) *Baum* darstellen.

---

[76] Es gelten die unter 4.7 genannten Einschränkungen.

[77] wie unter 3.2 geschildert, werden Zuordnungslisten mit Grammatiken und Variablen in einer Zuordnungsliste mit Nichtterminalen assoziiert.

[78] Übergangsdiagramme interessieren uns nicht, da wir ja eine Sprache selbst darstellen wollen, und nicht Automaten, die die Sprache erkennen.

Zu (1):

Am einfachsten lassen sich Produktionengraphen von Zuordnungslisten darstellen, bei denen jede rechte Seite vom Typ `ExternalObject` ist, oder ein Or-Set solcher Werte. In diesem Fall hat jede Variable den Typ `ExternalObject`. Jeder Teilgraph rsb_i eines Produktionengraphen g_E ist dann ein Komponentenbaum oder eine Or-Set-Darstellung von solchen, vorausgesetzt man ersetzt alle Referenzknoten durch Grundterme vom Typ `ExternalObject`. Man verwendet anfangs für diesen Zweck unsichtbare graphische Komponenten. Stellt man nun alle diese Komponentenbäume dar, indem man sie zu einem externen Objekt der Java-Klasse `GraphView` als Wald zusammenfaßt, und ersetzt danach jede Komponente eines Referenzknoten r_{ij} durch ein externes Objekt, das einen Pfeil zu den Koordinaten der Wurzel des Komponentenbaums bzw. der Or-Set-Darstellung rsb_i repräsentiert, so erhält man eine Darstellung für g_E. Praktisch ersetzt man die Referenzknoten natürlich nur durch Komponenten einer einzigen Java-Klasse `ReferenceView` (bzw. der Modell-Klasse `ReferenceNode`) und ändert ggfs. ihre Visualisierung, nachdem die Koordinaten der einzelnen Komponentenbäume feststehen (wozu man diese natürlich zuerst dargestellt haben muß!).

Hat man nun stattdessen eine beliebige Zuordnungsliste (bzw. dessen Produktionengraphen) als Resultat für einen Ausdruck a erhalten, so kann man sehr einfach einen äquivalenten Produktionengraphen der beschriebenen Form erhalten, in dem man sich die Zuordnungsliste für den Ausdruck `view` a berechnen läßt (bzw. `defaultView` a).

Zu (2):

Die Lösung (1) ist zwar sehr einfach und elegant, hat aber den Nachteil, daß die Berechnung der Zuordnungsliste für `view` a bzw. `defaultView` a aufwendig sein kann im Vergleich zur Durchführung der Berechnung der Zuordnungsliste nur für a. Außerdem sind i.a. die für `view` a bzw. `defaultView` a resultierenden Grammatiken schlechter zu vereinfachen, weil die rechten Seiten der Produktionen i.a. komplizierter ausfallen. Da der Produktionengraph direkt aus der Grammatik konstruiert wird, wirkt sich das auch auf dessen Größe aus. Im folgenden wird daher eine Alternative beschrieben, die allerdings eine unschöne Ausnahmeregel bei der Evaluierung von Darstellungsfunktionen voraussetzt:

Ein Weg, aus einer Zuordnungsliste, deren Zuordnungen bzw. Produktionen *nicht* vom Typ `ExternalObject` sind, ein darstellbares externes Objekt zu konstruieren, besteht darin, auf jede rechte Seite der Zuordnungen die Funktion `view` bzw. `defaultView` anzuwenden. Da für jedes Vorkommnis eines Nichtterminals eine Referenz gezeichnet werden soll, muß nun die `view` bzw. `defaultView`-Funktion für Anwendungen auf Nichtterminale überladen werden.

Die nötige Überladung muß intern geschehen[79]. Überladen werden müssen alle Funktionen der Form `view` und `view#n` bzw. `defaultView` und `defaultView#n` (vgl. 4.5.3), und zwar derart, daß die Anwendung dieser Funktionen auf ein Nichtterminal ein externes Objekt der Klasse `ReferenceNode` (die Modell-Klasse der Darstellungsklasse `ReferenceView`) ergibt (siehe (1)). Sind alle rechten Seiten durch `view` bzw. `defaultView` in Grundterme vom Typ `ExternalObject` umgewandelt worden, kann man wie unter (1) beschrieben fortfahren, um die Graphdarstellung zu erzeugen.

Zu (3):
Hat man mit einer der Methoden (1) oder (2) einen Produktionengraph konstruiert, so kann man Teilgraphen, die einen unendlichen Grundterm repräsentieren, auch als unendlich tiefen *Baum* darstellen statt als (ggfs. zyklischen) Graphen. Man muß lediglich nach dem Parsen des externen Objekts der Klasse `GraphView` auf Java-Seite die externen Objekte der Modell-Klasse `ReferenceNode` durch die referenzierten Objekte (dies sind die Komponentenbäume bzw. die Or-Set-Darstellungen rsb_i) selbst ersetzen, falls die Referenz-Knoten nicht eine fremde Alternative eines Or-Sets referenzieren (solche Pfeile werden als *Bögen* dargestellt, während Pfeile zu fremden Objekten als *Vektoren* angezeigt werden). Dabei macht man sich zunutze, daß Java-Objekte intern durch Kopieren des *Zeigers* auf das Objekt mehrfach referenziert werden können[80]. Dadurch sind diese Objekte auf dem Java-Heap physikalisch nur ein einziges Mal vorhanden. Man muß nur verhindern, daß die einzelnen, jetzt ggfs. unendlichen Komponentenbäume „auf einmal“ dargestellt werden. Dies ist dadurch sicherzustellen, daß alle Knoten des Baums bei ihrer ersten Darstellung „kollabiert“ sind, d.h. erst durch eine Anwenderaktion („Mausklick“) ihre Söhne darstellen (ausgenommen ist der Wurzelknoten).
Methode (3) wurde nur für automatisch generierte Ansichtsfunktionen (d.h. Instanzen von `defaultView`) implementiert, weil nur dann sichergestellt ist, daß der Baum kollabiert dargestellt wird. Das Prinzip ist aber natürlich nicht auf die View-Klasse `TreeView` und die Modell-Klassen `TreeNode` beschränkt.

---

[79] Man bedenke, daß i.a. der Typ der Nichtterminal-Vorkommen nicht bekannt ist und seine Inferenz u.U. relativ zeitaufwendig sein könnte, falls die Zuordnungsliste mutuell rekursiv ist. Daher beläßt die Implementation während der Typinferenz alle Nichtterminal als Variablen und ersetzt sie während der Auswertung durch Dummy-Konstruktoren, für die dann die Darstellungsfunktionen `view` und `defaultView` „überladen“ sind.

[80] Echte Kopien erhält man dagegen durch Verwendung der `clone()`-Methode, die allerdings für viele Klassen nicht verfügbar ist.

Insgesamt stehen für Zuordnungslisten somit sechs verschiedene Darstellungsmöglichkeiten mit ihren spezifischen Vor- und Nachteilen zur Verfügung:

Darstellungsfunktion	*Graph-Generierung*	*Java-Klasse*	*Interpreter-Kommando zur Generierung des externen Objekts*
`view`	(1)	`GraphView`	`:fv` mit `cyclic = true` (siehe `GraphView.java`; die Umschaltung von `cyclic` erfolgt im Interactor-Menü „View“)
`defaultView`	(1)	`GraphView`	`:fd` mit `cyclic = true`
`view`	(2)	`GraphView`	`:vf` mit `cyclic = true`
`defaultView`	(2)	`GraphView`	`:df` mit `cyclic = true`
`defaultView`	(1) mit (3)	`TreeView`[81]	`:fd` mit `cyclic = false`
`defaultView`	(2) mit (3)	`TreeView`[82]	`:df` mit `cyclic = false`

Anmerkung: Es ist mit dem Interactor nicht möglich, Query-Variablen graphisch darzustellen.

---

[81] Das `TreeView`-Objekt wird wie bei (1) und (2) in ein `GraphView`-Objekt integriert. Der Objekt-Konstruktor des `GraphView` ersetzt im Modell des `TreeView`-Objekts die Referenzen der Klasse `ReferenceView` durch ihre Ziele (die Teilbäume rsb_i), statt die Referenzen wie bei (1) oder (2) als Pfeile darzustellen. Sichtbar ist schließlich nur noch das `TreeView`-Objekt.

[82] Das `TreeView`-Objekt wird wie bei (1) und (2) in ein `GraphView`-Objekt integriert. Der Objekt-Konstruktor des `GraphView` ersetzt im Modell des `TreeView`-Objekts die Referenzen der Klasse `ReferenceView` durch ihre Ziele (die Teilbäume rsb_i), statt die Referenzen wie bei (1) oder (2) als Pfeile darzustellen. Sichtbar ist schließlich nur noch das `TreeView`-Objekt.

Beispiele

1) Darstellung eines unendlichen Or-Sets als Graph mittels explizit angegebener Darstellungsfunktion `view`

Dargestellt ist der Term `infTF`, wobei die die Bindungen

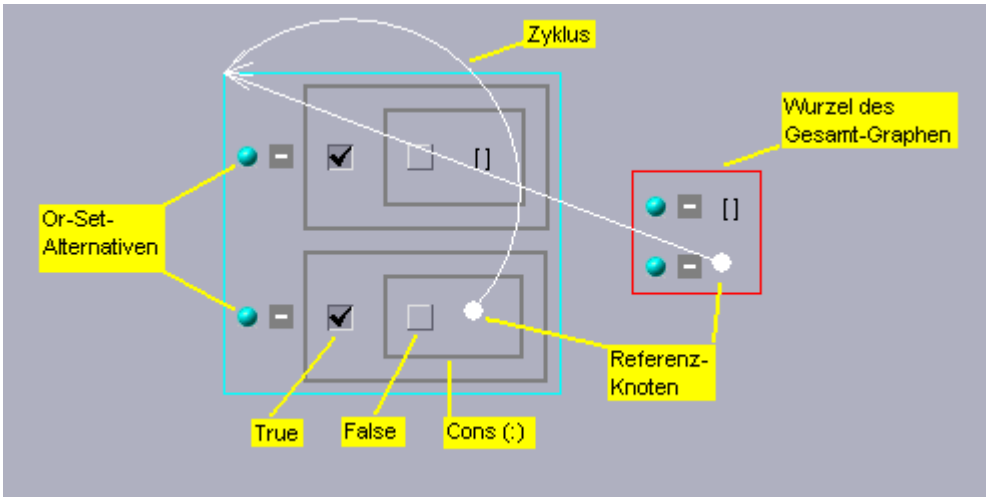
```
infTF = True : infFT or* []
infFT = False : infTF
```

vorausgesetzt werden. Die Darstellungsfunktion wird folgendermaßen definiert:

```
view :: Bool -> ExternalObject
view b =
 ExternalStatic "BoolView" "" [ExternalGeneric (if b then 1 else
0)]

view :: [Bool] -> ExternalObject
view [] = ExternalStatic "ConsView" "" []
view (hd:tl) = ExternalStatic "ConsView" "" [view hd, view tl]
```

Die „Cons“-Komponenten in der Darstellung werden durch die zweite Instanz von `view` generiert (`:: [Bool] -> ExternalObject`). Die linke Sub-Komponente von Cons ist der Listenkopf, die rechte der Rumpf (`[]` ist auch in der Darstellung die leere Liste). Die Listen-Atome `True` und `False` werden als An/Aus-Schalter durch die erste `view`-Instanz (`:: Bool -> ExternalObject`) dargestellt.



Die Darstellung wurde mit dem Befehl `:fv` erzeugt und stellt somit den mit der Methode 5.2.3-(1) erzeugten Graphen zu folgender Zuordnungsliste dar:

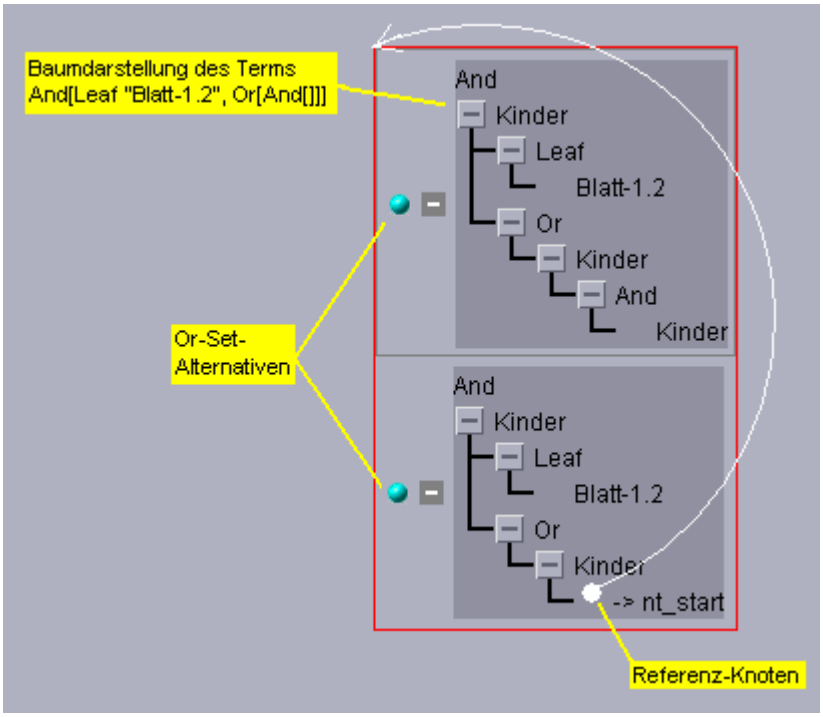
```
let {
  nt_start = True:False:nt_start or* []
} in nt_start
```

(diese Zuordnungsliste erhält man durch Verwendung des Kommandos `:fi` statt `:fv`)
Der Unterschied zur „Ausgangs-Grammatik“ `infTF` kommt dadurch zustande, daß der Prolid-Interpreter Grammatiken bzw. deren korrespondierende Zuordnungslisten recht effektiv vereinfachen kann.

2) Automatisch generierte Darstellung eines unendlichen Or-Sets

Dargestellt ist der Term `infTreeSet`, wobei die Deklarationen zu Und-Oder-Bäumen in 5.2.1 vorausgesetzt werden, sowie die Bindung

```
infTreeSet = And [Or [infTreeSet or* And []], Leaf “Blatt-1.2“]
```



Die Darstellung wurde mit dem Befehl `:df` erzeugt und stellt somit den mit der Methode 5.2.3-(2) erzeugten Graphen zu folgender Zuordnungsliste dar:

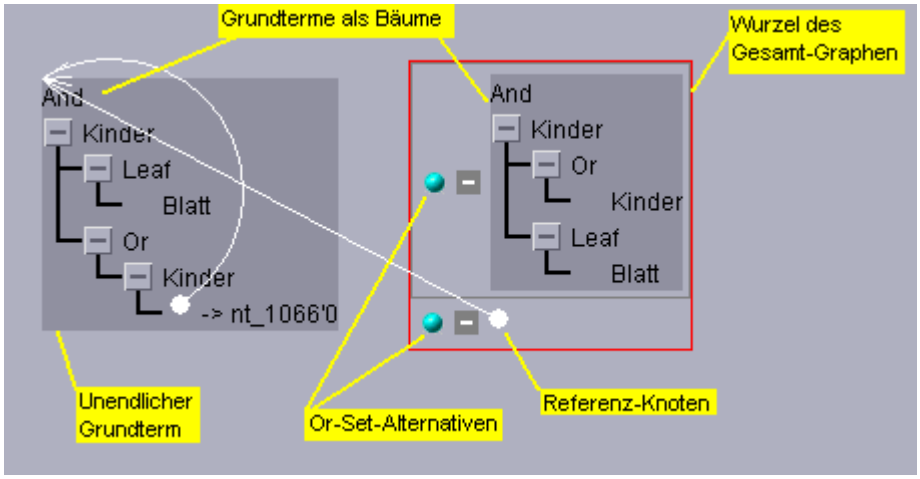
```
let {
  nt_start = And [Or [And []], Leaf "Blatt-1.2"]
             or*
             And [Or [nt_start], Leaf "Blatt-1.2"]
} in nt_start
```

(diese Zuordnungsliste erhält man durch Verwendung des Kommandos `:fi` statt `:df`)

3) Automatisch generierte Darstellung eines endlichen Or-Sets mit einem unendlichen Grundterm darin

Dargestellt ist der Term `And [Or [], Leaf "Blatt"] or infDeepTree`,
wobei die Deklarationen aus dem Beispiel in 5.2.1 (Und-Oder-Bäume) verwendet werden und `infDeepTree` folgendermaßen definiert ist:

```
infDeepTree = And [Leaf "Blatt", Or [infDeepTree]]
```



Die Darstellung wurde mit dem Befehl `:df` erzeugt und stellt somit den mit der Methode 5.2.3-(2) erzeugten Graphen zu folgender Zuordnungsliste dar:

```
let {
  nt_start = And [Or [], Leaf "Blatt"] or* nt_1066'0
 ;nt_1066'0 = And [Leaf "Blatt", Or [nt_1066'0]]
} in nt_start
```

(diese Zuordnungsliste erhält man durch Verwendung des Kommandos `:fi` statt `:df`)

Beispiel 1) befindet sich im Modul `View`, die Beispiele 2) und 3) sind im Modul `AndOrTree` definiert.

6 Zusammenfassung und Ausblick

Diese Arbeit hat gezeigt, daß sich Verarbeitungsmöglichkeiten für indefinite Daten in Gestalt von *Or-Sets* nahtlos in die Konzepte moderner *funktionaler Programmiersprachen* einfügen lassen. Es wurden in Gestalt von *Or-Set-Generatoren* verschiedene Spracheelemente zur Darstellung indefiniter Daten eingeführt und diese mit Elementen der Sprache *Haskell* zu der neuen *indefiniten funktionalen Programmiersprache* Prolid integriert. Dabei wurde aufgezeigt, daß sich die Methode des *Narrowing*, die von funktional-logischen Programmiersprachen wie z.B. *Curry* verwendet wird, mit dem Konzept der Or-Sets verbinden läßt. Weitere Anstrengungen wären nötig, um die Gründe und Konsequenzen dieser Gemeinsamkeit genauer zu erforschen. Für eine Weiterentwicklung von Prolid wäre es in diesem Zusammenhang außerdem unbedingt erforderlich, den implementierten rudimentären *Constraint-Tester* zu verbessern. Weiterhin wurden *reguläre Baum-Grammatiken* als Repräsentanten algebraischer Datentypen und bestimmte indefinite Programme als äquivalent erkannt und ein Algorithmus vorgestellt, um indefinite Programmen zur Berechnung regulärer unendlicher Baum-Sprachen zu einer Grammatik auszuwerten (*Rekursionserkennung*). Es sollte angestrebt werden, diesen Algorithmus weiterzuentwickeln, z.B. für die Nutzung mit *Graph-Reduktion* als Auswertungsmethode. Zum Schluß haben wir Möglichkeiten entwickelt, mit unserer rein funktionalen Sprache indefinite Daten (insbesondere Baum-Grammatiken) *graphisch* darzustellen und dabei dank der möglichen *Isomorphie* von algebraischen Datentypen und Objektklassen *objektorientierte* Mechanismen zu nutzen, wie sie z.B. von der Programmiersprache *Java* bereitgestellt werden. Die dargestellten Konzepte könnten vermutlich ohne größere Probleme genutzt werden, um auch die graphische *Eingabe* von indefiniten Daten zu realisieren.

Anhang A: Referenz zum Interpreter

A.1 Installation

- Das Archiv `prolid.zip` einfach in einem neuen Verzeichnis entpacken (Vorschlag: `/usr/Prolid-0.3.0` unter UNIX und `\Program Files\Prolid-0.3.0` unter der englischen Version von Windows 95/NT),
  ggfs. Pfad und Dateiattribute für das Executable `prolid` (UNIX) bzw. `prolid.exe` (Windows) setzen.

A.2 Bedienung

(Konsolen-Kommandos beim Start von der Kommandozeile der Shell aus oder innerhalb der Konsole eines Interactor-Interaktionsfensters)

- `:q` beendet den Interpreter. Wird der Interpreter im Interactor verwendet, reicht das Schließen des Fensters

- ohne vorangehendes `:`-Kommando wird ein eingegebener Ausdruck *ohne* Rekursionserkennung ausgewertet. Ist der Interactor aktiv, und ergibt sich als Resultat kein `ExternalObject`, so wird vom Interactor nachfolgend der Befehl `defaultView` <Resultat> initiiert.

- `:fi` <Ausdruck> wertet <Ausdruck> mit eingeschalteter Rekursionserkennung aus, d.h. man erhält eine *Zuordnungsliste* als Resultat. Dies ist die „offizielle Semantik“ des Interpreters (vgl. Kapitel 4).Achtung: Ausdrücke mit Query-Variablen sollten nicht mit :fi evaluiert werden, weil dann die Contraints nicht dargstellt werden.

- `:fv` <Ausdruck> berechnet mittels einer Zuordnungsliste für `view` <Ausdruck> die graphische Darstellung für <Ausdruck> (vgl. 5.3). Nur im Interactor sinnvoll. Natürlich müssen passende Instanzen der Darstellungsfunktion `view` geladen worden sein. Tip: Mit `:vf` statt `:fv` erzielt man *manchmal* bessere Resultate (vgl. 5.3).

- `:fd` <Ausdruck> berechnet mittels einer Zuordnungsliste für `defaultView` <Ausdruck> die graphische Darstellung für <Ausdruck> (vgl. 5.3). Nur im

Interactor sinnvoll. Natürlich müssen passende Instanzen der Darstellungsfunktion `defaultView` vorhanden worden sein. Dazu darf der Interpreter nicht mit den Schalter `-n` gestartet worden sein (unterdrückt die Generierung von `defaultView` - Instanzen). Tip: Mit `:df` statt `:fd` erzielt man *manchmal* bessere Resultate (vgl. 5.3).

- `:vf` <Ausdruck> berechnet mittels einer direkten Darstellung per `view` einer Zuordnungsliste für <Ausdruck> die graphische Darstellung für <Ausdruck> (vgl. 5.3). Nur im Interactor sinnvoll. Natürlich müssen passende Instanzen der Darstellungsfunktion `view` geladen worden sein.

- `:fd` <Ausdruck> berechnet mittels einer direkten Darstellung per `defaultView` der Zuordnungsliste für <Ausdruck> die graphische Darstellung für <Ausdruck> (vgl. 5.3). Nur im Interactor sinnvoll. Natürlich müssen passende Instanzen der Darstellungsfunktion `defaultView` vorhanden worden sein. Dazu darf der Interpreter nicht mit den Schalter `-n` gestartet worden sein (unterdrückt die Generierung von `defaultView` - Instanzen).

- `:t` <Ausdruck> zeigt den Typ von <Ausdruck> an, ohne ihn auszuwerten

- `:l` <$Modul_1$> … <$Modul_n$> entfernt alle z.Zt. geladenen Module und lädt die Module 1 bis n.

- `:l` ohne nachfolgende Modulliste lädt die Standardbibliothek `Stdlib.pro`.

- `:a` <$Modul_1$> … <$Modul_n$> (wie `:l` …, entfernt aber keine bereits geladenen Module)

- `:r` lädt alle z.Zt. geladenen Module neu. Nützlich zum Debuggen.

- :s <Image> schreibt die Deklarationen aller z.Zt. geladenen Module in „compilierter“ Form (d.h. als Haskell-Datenstrukturen) in ein einziges *Image-Modul* namens <Image>, das die Dateiendung `.ipro` haben sollte. Image-Module werden deutlich schneller als Module im Quellcode (Endung `.pro`) geladen. <Image> kann bei jedem Interpreter-Kommando und in der Kommandozeile beim Interpreter-Start als Dateiname eines Moduls angegeben werden.
  Beispiel: `:s Main.ipro`

A.3 Übersetzung:

- Die notwendigen Tools finden sich unter folgenden URLs:
  - Alex (Lexer-Generator):
    `http://odyssey.ucc.ie/www/user-dirs/dornan/alex.html`
  - Happy (Parser-Generator):
    `http://www.dcs.gla.ac.uk/fp/software/happy/`
  - Glasgow Haskell Compiler:
    `http://www.dcs.gla.ac.uk/fp/software/ghc/`
- Checkliste vor der Übersetzung:
  - Ziel „Glasgow Haskell Compiler“ (GHC) (ab Version 2.04) oder runhugs (1.4):
    - `ProlidParser.hs` und `ModuleParser.hs` mit „Happy“ unter Verwendung des Switches `-g` erzeugen
    - in `prolid.hs` und `Static.hs` "`import Trace`" deaktivieren, (nur) in `Static.hs` "`trace … =` ..." aktivieren
    - im Modul `Settings.hs` Einstellungen kontrollieren
    - nur fuer GHC: in `Input.hs` "`hReady =` ..." und "`hWaitForInput =` ..." auf `False` setzen
  - WinHugs/Hugs (1.4) (nur zu Debuggingzwecken empfehlenswert):
    - `ProlidParser.hs` und `ModuleParser.hs` mit „Happy“ *ohne* Verwendung des Switches `-g` erzeugen
    - in `prolid.hs` und `Static.hs` "`import Trace`" aktivieren, in `Static.hs` "`trace … =` ..." deaktivieren (klappt nicht fuer runhugs)
    - Ggfs. in `Settings.hs` "`toggleCheckTypes = False`" setzen
    - in `Input.hs` "`hReady =` ..." und "`hWaitForInput =` ..." aktivieren

- Compilieren mit dem „Glasgow Haskell Compiler“ (GHC) (ab Version 2.06)

  - Die Spezifikationsdateien mit der Endung `.y` und `.x`-Dateien sind in Haskell-Module übersetzen.

    - Für die .x-Dateien ist dazu der Lexer-Generator „Alex“ zu verwenden:

      ```
      alex prolid.x ProlidLexer.hs +RTS –H132m –K6m
      alex ModuleScan.x ModuleScan.hs
      alex Preproc.x Preproc.hs
      ```

  - Für die .y-Dateien ist dazu der Parser-Generator „Happy“ (ab Version 1.2) zu verwenden:

    ```
    happy prolid.y –info HappyInfo.txt –o ProlidParser.hs
    ```

    (`HappyInfo.txt` enthält anschließend nützliche Informationen über den erstellten Parser.) Wichtig: in `ProlidParser.hs` muß von Hand die Zeile

    ```
    happyNewToken action sts stk ((TokenLine _ _):tks) =
      happyNewToken action sts stk tks
    ```

    direkt vor der zweiten Deklaration für `happyNewToken` eingefügt werden! (Dient dazu, Zeilennummer-Tokens abzufangen. Für diese Tokens sind keine grammatikalischen Produktionen definiert.)

    ```
    happy ModuleParser.y –o ModuleParser.hs
    ```

    (vgl. auch die Skripten `Make…Lexer/Preproc/Parser….bat`)

  - Übersicht über alle Module (weiter unten stehende Module importieren evtl. vorangehende)

`Alex`.	Laufzeitumgebung für Lexer
`Settings`	einige Schalter (zu Debuggingzwecke)
`LexerHelper`	diverse Hilfsfunktionen
`Preproc`	fügt vor der lexikalischen Phase Zeilennummern und Modulnamen in den Prolid-Quellcode ein (vgl. `Preproc.x`)

`ProlidLexer`	Lexer (generiert aus `prolid.x`)
`ParserHelper`	diverse Hilfsfunktionen
`ModuleScan`	Ersatz für das langsame Haskell-`read` zum Einlesen von Image-Modulen (lexikalisch, vgl. `ModuleScan.x`)
`ModuleParser`	Ersatz für das langsame Haskell-`read` zum Einlesen von Image-Modulen (kontextfrei, vgl. `ModuleParser.y`)
`ProlidParser`	Parser (generiert aus `prolid.y`)
`Pretty`	Pretty-Printer für Prolid-Resultate
`Input`	zeilenweise Konsolen-Eingabe
`Static`	statische Phase (Typ-Checking, Aufkösung von Überladung u.ä.)
`prolid`	Hauptmodul, Laufzeitverhalten von Prolid-Programmen

- Compilieren und Linken der Haskell-Module (ruft u.a. auch Perl und den GNU-C++-Compiler gcc auf):

```
rm *.hi          (um die „Versionskontrolle“ des GHC auszuschalten)
rm *.o

ghc -H132m -K2m -c -O2 -fvia-C -O2-for-C Alex.hs
ghc -H132m -K2m -c -O2 -fvia-C -O2-for-C Settings.hs
ghc -H132m -K2m -c -O2 -fvia-C -O2-for-C LexerHelper.hs
ghc -H132m -K2m -c -O2 -fvia-C -O2-for-C Preproc.hs
ghc -H132m -K2m -c -O2 -fvia-C -O2-for-C ProlidLexer.hs
ghc -H132m -K2m -c -O2 -fvia-C -O2-for-C ParserHelper.hs
ghc -H132m -K2m -c -O2 -fvia-C -O2-for-C ModuleScan.hs
ghc -H132m -K2m -fglasgow-exts -c -O2 -fvia-C -O2-for-C
    ModuleParser.hs
ghc -H132m -K2m -fglasgow-exts -c -O2 -fvia-C -O2-for-C
    ProlidParser.hs
ghc -H132m -K2m -c -O2 -fvia-C -O2-for-C Pretty.hs
ghc -H132m -K2m -c -O2 -fvia-C -O2-for-C Input.hs
ghc -H132m -K2m -c -O2 -fvia-C -O2-for-C Static.hs
ghc -H132m -K2m -c -O2 -fvia-C -O2-for-C prolid.hs

ghc -o prolid *.o
```

(vgl. auch Skript `makeInterpreter.sh`)

Vorsicht: zumindest mit dem GHC unter Cygwin (das Gnu-Unix für Windows95/NT) sollten nie mehrere `.hs`-Dateien durch denselben Compileraufruf übersetzt werden, weil sonst der Compiler abstürzen kann!

A.4 Aufruf

- Kommandozeilen-Parameter der ausführbaren Datei `prolid`:

  - Auswertung von main in einem Modul `MyModule` (Datei `MyModule.pro`) erfolgt durch `./prolid MyModule +RTS -K2m -H50m` aus dem Prolid-Verzeichnis heraus.

    (Modulnamen von "literate" Skripten sind beim Aufruf mit der Endung `.lpro` zu versehen, *Image-Module* (angelegt mit `:s` <Datei>) werden bei der Dateiendung `.ipro` als solche erkannt und deutlich schneller als Quellcode-Module (Endung `.pro`) geladen.)

  - Konsolen-Modus: Schalter `-c` nach dem Modulnamen.

  - Interactor-Modus (vgl. Anhang B): Schalter `-i` oder `-f` (bei Verwendung von `runhugs`)

  - Keine automatische Erzeugung von `defaultView`-Funktionen: `-n` (empfehlenswert, falls keine Graphik gebraucht wird. Bringt natürlich Probleme, wenn `defaultView`-Darstellungen erzeugt bzw. verwendet werden sollen)

Beispiel-Aufrufe:

```
./prolid Main -c –n +RTS -H50m -K2m
```

(ruft den Interpreter mit dem Quellcode des Moduls `Main` auf (vgl. Kommando `:s`), unterdrückt die Erstellung von `defaultView`s. Heap: 50 MB, Stack: 2 MB)

```
./prolid Main.ipro -c +RTS -H50m -K2m
```

(ruft den Interpreter mit dem Image des Moduls `Main` auf. Heap: 50 MB, Stack: 2 MB)

Achtung: *zu viel* Heap (+RTS –H…) verlangsamt die Ausführung, aber 49 MB sind i.a. zu wenig! Hier hilft leider nur experimentieren.

Tip: Nach jedem Ändern und (Neu-)Laden eines `.pro`-Moduls sollte mit `:s` ein Image-Modul (Endung `.ipro`) dazu erstellt werden. Image-Module werden deutlich schneller als `.pro`-Module geladen.

- Aufruf des Konsolenmodus für das Prolid-Modul `Main.pro` unter WinHugs:
  - Hugs starten mit `d:\hugs\winhugs.exe +t +k –h7m` (reserviert 7 MB Heap)
  - in das Verzeichnis mit den Quelldateien wechseln
  - `:l prolid`
  - `run "Main -c"`
- Aufruf des Konsolenmodus für das Prolid-Modul `Main.pro` mit runhugs (ohne Interactor):
  - in das Verzeichnis mit den Quelldateien wechseln
  - `runhugs -h2m prolid Main -c`

Anmerkung: Die Einbindung des Interpreters in den *Interactor* ist im Anhang B geschildert.

Anhang B: Referenz zum Interactor

B.1 Installation

(üblicherweise zusammen mit dem Interpreter, siehe A.1)

B.2 Bedienung

Der Interactor führt eine theoretisch beliebige Zahl von sogenannten *Interaktionen* gleichzeitig aus. Jede Interaktion stellt in einem eigenen inneren Fenster eine Interpreter-Konsole und darüber einen Bereich für die graphische Darstellung der Interpreter-Resultate dar. Eine Interaktion wird über die Kommandozeile des Interactors initiiert (siehe B.4). Weitere Interaktionen können im „Interaction"-Menü des Hauptfenster mit dem Befehl „New…" gestartet werden[83]. Jede Interaktion ist mit genau einem Interpreter-Prozeß assoziiert (vgl. Kapitel 5).

Die Bedienung der Interpreter-Konsole erfolgt wie im Anhang A.2 geschildert, mit dem Zusatz, daß man bereits ausgeführte Kommandos wiederholen kann, indem man den Konsolen-Cursor irgendwo in die Zeile des Kommandos stellt und die <Enter>-Taste drückt, nachdem man das Kommando vorher editiert hat.

Stellt der Interactor nach der Berechnung des Resultats einer Auswertung fest, daß das Resultat vom Typ `ExternalObject` ist, so stellt er es über der Konsole graphisch dar. Andernfalls ruft er die `defaultView`-Funktion für das Resultat auf, um so eine graphische Darstellung zu ermitteln. Den Fortschritt dieser Vorgänge sollte man in der Java-Konsole beobachten.

Wichtig: Es treten manchmal die Java-typischen Refresh-Probleme bei der Darstellung auf. Wenn Graphen nur unvollständig dargestellt werden (z.B. mit Pfeilen, die ins „Nichts" führen), erlangt man eine korrekte Darstellung durch mehrfache Benutzeraktionen auf die betroffenen Komponenten (z.B. Mausklick auf `+/-`-Schalter zum expandieren/implodieren von Knoten).

Die Interpreter-Befehle `:a`, `:r` und `:l` sind auch über das Menü „Module" im jeweiligen Interaktions-Fenster erreichbar („Add…" = `:a` mit interaktiver Auswahl der Modul-Datei, „Reload" = `:r` und „Clear" = `:l`). Wird das Fenster geschlossen, wird der zugehörige Interpreter-Prozeß automatisch beendet, d.h. der Interpreter-Befehl `:q` wird nicht benötigt.

---

[83] (wobei einfache Rechner üblicherweise bereits mit einer einzigen Interaktion ausgelastet sind…)

Im Interaktions-Menü „View“ können gewisse Zyklen im angezeigten Graphen ausgeschaltet werden (Befehl „Cyclic graph off/on“, siehe 5.3) und die Darstellung gedruckt werden („Print“). Die Druckfunktion funktioniert leider nicht auf allen Systemen.

Das Interaktions-Menü „Console“ enthält die üblichen Zwischenablage-Kommandos („Cut“, „Copy“, „Paste“), die sich alle auf den markierten Bereich im Konsolen-Bereich des Interaktions-Fensters beziehen. Auch diese Kommandos arbeiten nicht unter allen Betriebssystemen[84]. „Print“ druckt den gesamten Verlauf einer Konsolen-Sitzung.

B.3 Übersetzung

- Die notwendigen Tools JDK, JDK und JavaCC finden sich unter
  `http://www.javasoft.com/index.html`

- Ggfs. die `.jj`-Datei (Spezifikation des Parser für *externe Objekte*, siehe 5.1) per „JavaCC“-Aufruf in `.java`-Module übersetzen (falls JavaCC im Verzeichnis \JavaCC installiert ist):

```
java /cp:p "\JavaCC\JavaCC.zip" COM.sun.labs.javacc.Main
     ProlidExternalDataParser.jj
```

- Aus dem Verzeichnis mit den Quell-Dateien heraus:

```
del *.class
javac –classpath ".; %SWING_HOME%;%SWING_HOME%\swing.jar;
%SWING_HOME%\windows.jar; %SWING_HOME%\motif.jar;
%SWING_HOME%\jlf.jar; %SWING_HOME%\mac.jar;
%SWING_HOME%\1.1_libs\swing.jar;
%SWING_HOME%\1.1_libs\windows.jar;
%SWING_HOME%\1.1_libs\motif.jar;
%SWING_HOME%\1.1_libs\jlf.jar;
%SWING_HOME%\1.1_libs\mac.jar; %CLASSPATH%"
Interactor.java
```

B.4 Aufruf

---

[84] z.B. verfügen viele Unix-Desktops nicht über eine Prozeß-übergreifende Zwischenablage.

- In das Verzeichnis mit den `.class`-Dateien wechseln

- ```
  java -classpath ".;%SWING_HOME%; %SWING_HOME%\swing.jar;
  %SWING_HOME%\windows.jar; %SWING_HOME%\motif.jar;
  %SWING_HOME%\jlf.jar; %SWING_HOME%\mac.jar;
  %SWING_HOME%\1.1_libs\swing.jar;
  %SWING_HOME%\1.1_libs\windows.jar;
  %SWING_HOME%\1.1_libs\motif.jar;
  %SWING_HOME%\1.1_libs\jlf.jar;
  %SWING_HOME%\1.1_libs\mac.jar; %CLASSPATH%" Interactor
  ```

- Mögliches Argument nach `java ... Interactor` ist (in Anführungszeichen!) der Aufrufbefehl des Prolid-Interpreters, z.B.

  ```
  "./prolid Main.ipro -i +RTS -H50m -K2m"
  ```
  oder
  ```
  "runhugs –h7m prolid Main -f"
  ```

  Dabei ist üblicherweise `-i` als Switch zu verwenden, wenn das Executable `prolid` verwendet wird, und `-f` zusammen mit `runhugs` (vgl. A.4). Hugs sollte aus Performance-Gründen nur zu Debugging-Zwecken verwendet werden.

  Anmerkung: Im Prinzip können auch andere Programme als der Prolid-Interpreter zusammen mit dem Interactor verwendet werden, solange sie die Standard-Ein-/Ausgabe zur Bedienung verwenden und der Interactor ihre Ausgaben „versteht“. Anpassungen des Interactors wären ggfs. in `ProlidExternalDataParser.jj` (Parser für externe Objekte) und bei der Klasse `InterpreterInputInterface` (Thread-Manager für die Interaktions-Fenster, erkennt Anfang und Ende relevanter Interpreter-Ausgaben) in der Datei `Interactor.java`) durchzuführen (vgl. Kommentare im Quellcode).
  ```

Anhang C: Optimierte Rekursionserkennung

Hier hat die neue Funktion evaluationNf‘ denselben Aufbau wie evaluationHNf. Dadurch werden Einträge in die globale Variable gedächtnis (*Memoing*) häufiger vorgenommen Das Memoing unterbleibt zwar auch hier bei der Berechnung der indefiniten Kopf-Normalform, dafür geht die Berechnung der Normalform per evaluationNf‘ weiter in die Tiefe. Dadurch wird die Terminierung von Programmen möglich, für die das in 4.7.2 angegebene Verfahren scheitert.

(gedächtnis, zuordnungsliste, evaluation wie 4.7.2.2)

evaluationNf (a, (su, ou), (arg_1 , … ,arg_n)) begin

if ∃a’∈ Ausdruck : ((a, (su, ou)), a’) ∈ gedächtnis then begin
  return (Nichtterminal (a, (su, ou)))
end

gedächtnis := { ((a, (su, ou)), a’) } ∪ gedächtnis

a’ := evaluationNf’ (a, (su, ou), (arg_1, …,arg_n)))

man beachte die mutuelle Rekursion der Definitionen von gedächtnis und a’

return a’
end evaluationNf

Folgende Funktion hat denselben Aufbau wie evaluationHNf. Die dort angegebenen Erläuterungen gelten daher auch hier:

evaluationNf’ : Ausdruck × Umgebung × Argumente → Ausdruck

evaluationNf’ (k : k ∈ Konstruktor, (su, ou), (arg_1, …,arg_n)) →
  k evaluationNf (arg_1, umgebung(arg_1), ( )) evaluationNf (arg_n, umgebung(arg_n), ( ))

evaluationNf’ (Applikation (x, y), u, (arg_1, …,arg_n)) →
  evaluationNf (x, u, ((y, u), arg_1, …,arg_n))

evaluationNf' (or* (alt_A, alt_B), u, (arg_1, …,arg_n)) →

or* (evaluationNf (alt_A, u, (arg_1, …,arg_n)), evaluationNf (alt_B, u, (arg_1, …,arg_n)))

evaluationNf' (or_i (alt_A, alt_B), (su, ou), (arg_1, …,arg_n))

| falls (i, L) ∈ ou

→ evaluationNf (alt_A, u, (arg_1, …,arg_n))

| falls (i, R) ∈ ou

→ evaluationNf (alt_B, u, (arg_1, …,arg_n))

| sonst

→ or* (evaluationNf (alt_A, u_A, (arg_1, …,arg_n)),

evaluationNf (alt_B, u_B, (arg_1, …,arg_n)))

wobei u_A := (su, { (i, L) } ∪ ou), u_B := (su, { (i, R) } ∪ ou)

evaluationNf' (var: var ∈ Variable, (su, ou), (arg_1, …,arg_n))

| (∃a, u_v: (a, u_v) ∈ Abschluß, (var, (a, u_v)) ∈ su)

→ evaluationNf (a, u_v, (arg_1, …,arg_n))

| n = 0 ∧ (var ≡ k, k ∈ Konstruktor) ∈ ou

→ k

| n = 0

→ var

evaluationNf' (λ (x_1 ,…,$x_{arität}$) → r, (su, ou), (arg_1, …,arg_n))

| falls n = arität

→ evaluationNf (r, ({ (x_i, arg_i) : 1≤i≤n }, ou), ( ))

| falls n < arität

→ (λ (x_1 ,…,$x_{arität}$) → r) evaluationNf (arg_1, umgebung(arg_1), ( ))

… evaluationNf (arg_n, umgebung(arg_n), ( ))

| falls n > arität

→ evaluationNf (r, ({ (x_i, arg_i) : 1≤i≤arität }, ou), ($arg_{arität+1}$, …, x_n))

evaluationNf' (Verzweigung (t, m, a, b), (su, ou), (arg_1, …,arg_n))

| falls m ∈ Variable

→ evaluationNf (a, ({ (m, t) } ∪ su, ou), (arg_1, …,arg_n))

| falls m = ⊥

→ ⊥

| falls m = <u>impossible</u>

→ <u>impossible</u>

| sonst, *d.h. ($m = k\ v_1 \ldots v_n : k \in$ Konstruktor, $v_i \in$ Variable)*

→ (wobei hnfset := evaluationHNf (t, (su, ou), ( )))

{evaluationNf (a, (su ∪ su_A, ou ∪ ou_A), (arg_1, …,arg_n))

: (su_A, ou_A) ∈ match(m, hnfset)}

∪ { evaluationNf (b, (su, ou_B), (arg_1, …,arg_n))$_i$

: ou_B ∈ mismatch(m, hnfset) }

∪ { evaluationNf (a, (su, ou ∪ ou_A), (arg_1, …,arg_n))

: ou_A ∈ constrainedMatch(m, ou, hnfset) }

∪ { evaluationNf (a, (su, ou ∪ ou_A), (arg_1, …,arg_n))

: ou_A ∈ constrainedMismatch(m, ou, hnfset) }

[∪ { ⊥ } falls ((⊥, ..), ..)∈ hnfset]

evaluationNf' (<u>impossible</u>, (su, ou), (arg_1, …,arg_n))

→ <u>impossible</u>

evaluationNf' (⊥, (su, ou), (arg_1, …,arg_n))

→ ⊥

Literatur

1. A.V. Aho, R. Sethi, J.D. Ullmann. *Compilerbau*. Addison-Wesley, 1990.

2. M. Hanus. A Unified Computation Model for Functional and Logic Programming. Forschungsbericht, Symposium on Principles of Programming Languages, 1997.

3. P. Cousot, R. Cousot. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. ACM Principles of Programming Languages, 1977.

4. F. Gécseg, M. Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.

5. E. Gunter, L. Libkin. *OR-SML: A Functional Database Programming Language for Disjunctive Information*. Forschungsbericht, 1994.

6. J. Giarratano, G. Riley. *Expert Systems – Principles and Programming*. PWS Publishing Company, 1994.

7. D. Flanagan. *Java in a Nutshell*. O'Reilly & Associates, 1996.

8. K. Hammond, C. Hall. *A Dynamic Semantics for Haskell (Draft)*. Forschungsbericht, Universität von Glasgow, 14. April 1992.

9. M. Hanus (Hrsg.). *An Integrated Functional Logic Language (Draft of June 6, 1997)*. Technischer Bericht, RWTH Aachen, Juni 1997.

10. R. Harper, R. Milner, M. Tofte. *The Definition of Standard ML*. MIT Press, 1991.

11. P. Hudak, J. Fasel, J. Peterson. *A gentle introduction to Haskell*. Technischer Bericht YALEU/DCS/RR-901, Yale Universität, Mai 1996.

12. J. Hughes, A. Ferguson. *A Loop-detecting Interpreter for Lazy, Higher-order Programs*. Forschungsbericht, Chalmers Technische Hochschule und die Universität von Glasgow, 199?

13. N.D. Jones, P. Sestoft, H. Søndergård. *Mix. a self-applicable partial evaluator for experiments in compiler generation*. Lisp and Symbolic Computation, 2(1):9-50, 1989.

14. S.P. Jones, P.L. Wadler. *The Static Semantics of Haskell*. Unveröffentlichter technischer Bericht, Universität von Glasgow, 1992.

15. S.P. Jones. *Implementing Functional Languages*. Prentice Hall, 1992.

16. L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

17. J. Peterson, K. Hammond (Hrsg.). *Report on the Programming Language Haskell*. Technischer Bericht, Universitäten von Yale und Glasgow, April 1997.

18. H. Schütz. *Indefinite Information with a Data Model Based on Algebraic Datatypes*. Forschungsbericht PMS-FB-1997-19, Institut für Informatik der Universität München, 1997.

19. M.I. Schwartzbach. *Polymorphic Type Inference*. Vorlesung, Universität von Aarhus, März 1995.

20. P. Graf. *Term Indexing*. Dissertation, Universität des Saarlandes, 1995

21. J. Young, P. Hudak. *Finding fixpoints on function spaces*. Forschungsbericht YALEU/DCS/RR-505, Institut für Informatik, Yale Universität, Dezember 1986.

22. S. Antony, R. Echahed, M. Hanus. *A Needed Narrowing Strategy*. In Proc. 21st ACM Symposium on Principles of Programming Languages, S. 268-279, Portland, 1994.