Universität des Saarlandes
Naturwissenschaftlich-Technische Fakultät
Fachrichtung Informatik
Bachelor-Studiengang

Bachelorarbeit

A Principle Compiler for Extensible Dependency Grammar

vorgelegt von

Jochen Setz

am 10.10.2007

angefertigt unter der Leitung von

Prof. Dr. Gert Smolka

betreut von

Dr. Ralph Debusmann

begutachtet von

Prof. Dr. Gert Smolka
Dr. Ralph Debusmann

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und alle verwendeten Quellen angegeben habe.

Saarbrücken, den 10.10.2007

Einverständniserklärung

Hiermit erkläre ich mich damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek der Fachrichtung Informatik aufgenommen wird.

Saarbrücken, den 10.10.2007

Danksagung

Danken möchte ich Dr. Ralph Debumann und Prof. Gert Smolka, dass sie mir diese Thesis angeboten haben. Vor allem Ralph gebührt großer Dank für die erstklassige Betreuung dieser Arbeit.

Dann möchte ich es nicht verpassen, mich bei meiner Familie und meinen Freunden für ihre Unterstützung zu bedanken.

Abstract

Extensible Dependency Grammar (XDG) ist ein neuer, constraint-basierter Meta-Grammatikformalismus (Debusmann 2006). Die Modelle von XDG werden durch Constraints in Prädikatenlogik erster Stufe charakterisiert, die *Prinzipien* genannt werden. In der Entwicklungsumgebung für XDG, dem XDG Development Kit (XDK), sind die Prinzipien als Constraints auf endlichen Mengen in Mozart/Oz implementiert (Schulte 2002). Allerdings ist die Implementation alles andere als trivial, und bis jetzt konnte nur eine Minderheit der Benutzer des XDK, Experten in Mozart/Oz Constraintprogrammierung, dieses auch um neue Prinzipien erweitern.

In dieser Arbeit entwickle ich ein Programm namens *Principle Writer (PW)*, das Constraints in First-Order Logik automatisch in Constraints in Mozart/Oz umsetzt. Damit schließe ich die bisher bestehende Lücke zwischen der Formalisierung von XDG und der Implementierung. Jetzt können *alle* Benutzer des XDK dieses um neue Prinzipien erweitern. Vom PW profitieren aber auch diejenigen, die die manuelle Umsetzung von Prinzipien in Constraints beherrschen, da die schnellere und sicherere automatische Übersetzung die Grammatikentwicklungszeit deutlich verringert. All das steigert die Attraktivität des Systems als Explorationsplattform für dependenzbasierte Grammatikformalismen erheblich.

Inhaltsverzeichnis

1. Einleitung

Extensible Dependency Grammar (XDG) ist ein constraint-basierter Meta-Grammatikformalismus (Debusmann, Duchier, Koller, Kuhlmann, Smolka & Thater 2004), (Debusmann 2006) für Dependenzgrammatik. Die Modelle von XDG sind *Dependenz-Multigraphen*, das sind Tupel von Dependenzgraphen, die dieselbe Knotenmenge teilen. Die einzelnen Komponenten eines Dependenz-Multigraph heißen *Dimensionen*.

XDG-Grammatiken bestehen aus einem *Multigraph-Typ*, der die Dimensionen, Kantenlabel, Knotenattribute etc. festlegt, einem *Lexikon*, das die Eigenschaften von Wörtern beschreibt, und einer Menge von *Prinzipien*, die die Wohlgeformtheitsbedingungen der Multigraphen charakterisieren. Die Prinzipien werden in der Formalisierung in (Debusmann 2006) in höherstufiger Logik, und in der neueren Formalisierung in (Debusmann 2007*b*) in Prädikatenlogik erster Stufe ausgedrückt.

Der Grundgedanke hinter XDG ist es, als Meta-Grammatikformalismus die Axiomatisierung von dependenzbasierten, konkreten Grammatikformalismen zu ermöglichen. Beispiele sind Lexicalized Context Free Grammar (LCFG), Lexicalized Tree Adjoining Grammar (LTAG) (Joshi 1987) oder Topological Dependency Grammar (TDG) (Duchier & Debusmann 2001), um die Syntax natürlicher Sprachen zu modellieren, und Dominanzconstraints (DC) (Egg, Koller & Niehren 2001) und Steedman's prosodischer Ansatz zur Informationsstruktur (IS) (Steedman 2000), um die Semantik zu modellieren. Einmal in XDG axiomatisiert können die Formalismen dann beliebig erweitert und kombiniert werden, mit dem Ziel, ihre ideale Kombination zu finden. In (Debusmann 2006) war die Eleganz und Ausdrucksmächtigkeit der Syntax-Semantik-Schnittstelle das Hauptkriterium, was in eine Kombination von TDG, DC und IS mündete.

Für XDG gibt es eine Entwicklungsumgebung, das XDG Development Kit (Debusmann, Duchier & Niehren 2004), die einen constraint-basierten Parser, Werkzeuge zur Fehlersuche und eine graphische Benutzeroberfläche beinhaltet. Grammatiken können in einer ausdrucksstarken Beschreibungssprache geschrieben werden, die vor allem das Schreiben von strukturierten XDG-Lexika mit lexikalischen Klassen und Templates unterstützt.

Die Implementierung der Wohlgeformtheitsbedingungen von XDG-Grammatiken, der Prinzipien, war jedoch bisher ein Schwachpunkt. Prinzipien-Implementationen mussten entweder aus der mitgelieferten Prinzipien-Bibliothek entnommen, oder von Hand als Constraint-Programm in Mozart/Oz implementiert werden. Das aber hieß bisher, dass nur Experten in Mozart/Oz-Constraintprogrammierung das XDK auch um neue Prinzipien erweitern konnten, für die Mehrheit der Nutzer blieben nur die vorimplementierten Prinzipien aus der Prinzipien-Bibliothek. Dieser Schwachpunkt schränkte die Attraktivität des XDK doch sehr ein. Gerade wenn es darum ging, neue Grammatikformalismen in XDG zu axiomatisieren, kam man um das Hinzufügen von neuen Prinzipien nicht herum, aber das konnten nur Experten, und auch die nur unter Einsatz von viel Zeit.

In dieser Arbeit geht es darum, diesen Schwachpunkt des XDK zu beheben. Ich entwickle ein Programm namens *PrincipleWriter*, das XDG-Prinzipien in Prädikatenlogik erster Stufe automatisch in Mozart/Oz Constraints umsetzt, ohne dass dazu Kenntnisse in Mozart/Oz

oder in Constraintprogrammierung im Allgemeinen nötig sind. Der Vorteil: mit dem PW kann jeder Nutzer mit Kenntnissen in Prädikatenlogik XDG-Prinzipien implementieren und im XDK ausprobieren. Auch Nutzer, die die Implementation von Prinzipien in Mozart/Oz beherrschen, profitieren vom PW, da sie ihre Ideen sehr viel schneller umsetzen können. Die Kompilation eines normalen Prinzips dauert weniger als eine Sekunde, verglichen mit mehreren Stunden, die das Implementieren von Hand leicht dauern kann.

Natürlich bringt die automatische Kompilation von Prinzipien nicht nur Vorteile. Der Hauptnachteil ist der, dass manche automatisch kompilierte Prinzipien nicht so effizient sind wie die von Hand geschriebenen. Um die Effizienz der automatisch kompilierten Prinzipien zu steigern, habe ich in meiner Arbeit erste Ideen zur automatischen Optimierung entwickelt. In diesem äußerst interessanten Bereich steckt jedoch noch viel mehr Potential, das wir im Laufe dieser Arbeit erst ganz leicht angeschnitten haben.

1.1. Bisherige Arbeiten

XDG ist ein constraint-basierter Grammatikformalismus, und als solcher auch vergleichbar mit unifikationsbasierten Formalismen wie Definite Clause Grammars (DCG) und PATR-II (Shieber 1984). Auch hier werden Wohlgeformtheitsbedingungen durch Constraints beschrieben, allerdings in einer Programmiersprache (Prolog).

Der Grammatikformalismus Head-driven Phrase Structure Grammar (HPSG) ist ebenfalls häufig in Prolog implementiert, z.B. im Babel-System (Müller 1996). Neue Wohlgeformtheitsbedingungen werden hier jedoch, ebenso wie in anderen HPSG-Implementationen wie dem Type Resolution System (Troll) (Gerdemann, Hinrichs, King & Götz 1994), Attribute Logic Engine (ALE) (Carpenter & Penn 1994) und TRALE (Meurers, Penn & Richter 2002), durch Unifikation über Merkmalsstrukturen ausgedrückt.

Die Prädikatenlogik erster Stufe, mit der die Constraints von XDG beschrieben werden, lässt sich auch als Modellierungssprache für Constraintprogrammierung auffassen. Als solche ist sie vergleichbar mit Sprachen wie die Optimization Programming Language (OPL) (van Hentenryck 1999), MiniZinc (Nethercote, Stuckey, Becket, Brand, Duck & Tack 2007), und natürlich auch Mozart/Oz (Smolka 1995), (Schulte 2002).

Das Lösen von First-Order Formeln mithilfe von Constraintprogrammierung ist auch in (Apt & Vermeulen 2002) zu finden. In (Tack, Schulte & Smolka 2006) werden Formeln in Existential Monadic Second Order Logic (EMSO) in neue Propagatoren übersetzt, die auf endlichen Mengen arbeiten. Diese Arbeit kommt meiner noch am nächsten. Allerdings setze ich die XDG-Formeln (in Prädikatenlogik erster Stufe) nicht in neue Propagatoren um, sondern in Code, der existierende Propagatoren von Mozart/Oz einsetzt.

1.2. Gliederung der Arbeit

Die Arbeit ist wie folgt gegliedert. In Kapitel 2 führe ich XDG ein, und in Kapitel 3 das XDG Development Kit (XDK). Diese beiden Kapitel bereiten das Hauptkapitel der Arbeit vor, Kapitel 4, in dem ich den PrincipleWriter vorstelle. Kapitel 5 schließt die Arbeit ab und gibt einen Ausblick.

2. Extensible Dependency Grammar

Extensible Dependency Grammar (XDG) (Debusmann, Duchier, Koller, Kuhlmann, Smolka & Thater 2004), (Debusmann 2006) ist ein Meta-Grammatikformalismus, der auf Dependenzgrammatik (Tesnière 1959) basiert, und diese mit modelltheoretischer Syntax und der parallelen Grammatik-Architektur von Jackendoff (Jackendoff 2002) kombiniert.

Durch das modulare Design von XDG können Grammatiken um jeden beliebigen linguistischen Aspekt erweitert werden. Die Aspekte, wie z.B. grammatische Funktionen, Wortstellung oder Prädikat-Argument-Struktur, werden jeweils auf einer eigenen Dimension behandelt. Die Aufteilung der Aspekte in Dimensionen erleichtert die Modellierung von linguistischen Phänomenen, da jeder Aspekt einzeln behandelt wird, und nicht alle Aspekte gleichzeitig modelliert werden müssen. Vor allem bei Aspekten, die nicht voneinander abhängen, erleichtert das modulare Design das Entwickeln von Grammatiken. Zum Beispiel spielt für die Modellierung der Prädikat-Argument-Struktur die Wortstellung keine Rolle, im Gegensatz zu den grammatischen Funktionen. In früheren Ansätzen musste die Wortstellung jedoch trotzdem bei der Modellierung der Syntax-Semantik-Schnittstelle miteinbezogen werden.

XDG wird durch eine Multigraph-Beschreibungssprache in Prädikatenlogik erster Stufe (First-Order Logik) formalisiert. Eine XDG-Grammatik besteht dabei aus einem Multigraphtyp, einer Menge von Prinzipien, die die Wohlgeformtheitsbedingungen des Multigraphen beschreiben, und einem Lexikon. Nachdem der Multigraphtyp festgelegt ist, werden die Prinzipien in Abhängigkeit des Multigraphtyps in First-Order Logik beschrieben. Das Lexikon drückt Eigenschaften von Wörtern aus.

2.1. Multigraphen

Ein Modell in XDG besteht aus einer Menge von Dependenzgraphen, die alle die gleiche Menge von Knoten teilen. Dadurch können die Dependenzgraphen zusammen als ein Multigraph aufgefasst werden. Ein Modell in XDG ist also ein Dependenz-Multigraph.

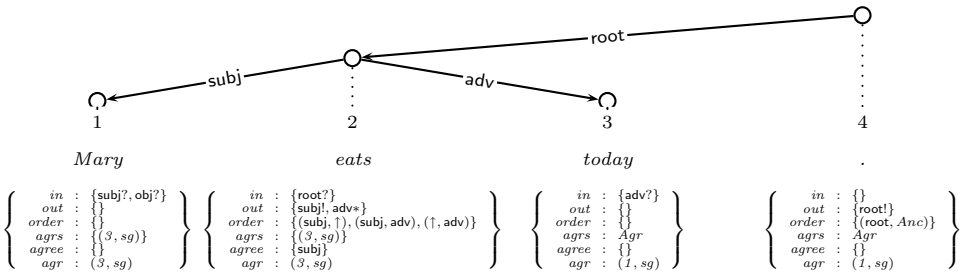


Abbildung 2.1.: Dependenzgraph

Gegeben:

- endliche Menge At von Atomen
- endliche Menge D von Dimensionen
- endliche Menge W von Wörtern
- endliche Menge L von Kantenlabels
- endliche Menge A von Attributen
- endliche Menge T von Typen

wobei gilt: $D, W, L, A \subseteq At$ und $U = \cup\{t|t \in T\}$.
Ein Multigraph ist ein Tupel $(V, E^+, <, nw, na)$ bestehend aus:

1. einem endlichen Intervall V aus $\mathbb{N}$ beginnend bei 1, das die Knoten repäsentiert.
2. einer endlichen Menge $E^+ \subseteq V \times V \times L \times D$ von gelabelten Dominanzen.
3. einer strengen Totalordnung $< \subseteq V \times V$
4. einem Knoten-Wort-Mapping $nw \in V \to W$
5. einem Knoten-Attribut-Mapping $na \in V \to D \to A \to U$.

Abbildung 2.2.: Multigraph formal

2.1.1. Dependenzgraphen

XDG-Modelle enthalten eine spezielle Form von Dependenzgraphen. Die Knoten sind hierbei mit einem numerischen Index, beginnend bei 1, versehen. Dieser Index gibt ihre Position im Eingabesatz wieder. Außerdem wird jeder Knoten mit genau einem Wort des zu analysierenden Satzes assoziiert, das in Abbildung 2.1 unter dem Index zu finden ist. Desweiteren wird jeder Knoten mit Attributen assoziiert, die in Abbildung 2.1 unter dem Wort stehen. Attribute modellieren lexikalische und nichtlexikalische Informationen. Die Knoten werden untereinander durch gerichtete und gelabelte Kanten verbunden. Die Kantenbeschriftungen drücken die Beziehungen der Knoten innerhalb der Dimension aus, z.B. syntaktische Funktionen oder semantische Rollen.

Dependenzgraphen in XDG unterliegen ansonsten keinerlei Restriktionen. Es wird also nicht verlangt, dass sie Baumform haben oder Ähnliches. Die Dependenzgraphen können beliebige Graphen sein, also auch Graphen mit Zyklen oder kreuzenden Kanten. Das ist ähnlich wie bei Word Grammar (WG) (Hudson 1990).

2.1.2. Dependenz-Multigraphen

Die verschiedenen Dependenzgraphen eines Modells können als Multigraph aufgefasst werden, da sie dieselbe Knotenmenge teilen. Jeder Dependenzgraph beschreibt eine andere Dimension.

Formal ist ein Multigraph in XDG definiert wie in Abbildung 2.2. Die Dimensionen D, die Wörter W, die Kantenlabel L und die Attribute A sind endliche Mengen von Atomen At.

Grammatik $G = (MT, lex, P)$ mit

1. MT := Multigraphtyp
2. lex := Lexikon
3. P := Menge von Prinzipien

P und lex sind auf dem gleichen Multigraphtype MT definiert.

Abbildung 2.3.: XDG Grammatik formal

Desweiteren gibt es eine endliche Menge von Typen T und eine Menge von Werten U. Werte sind immer Mengen von Tupeln über endlichen Mengen von Atomen.

V sind die Knoten des Multigraphen, duchnummeriert beginnend mit 1. Der Index gibt die Position des Knotens im Satz an.

Die Menge E^+ beschreibt indirekt die Kanten des Graphen, wobei gilt, dass $(v, v', l, d) \in E^+$ gdw. der Graph in der Dimension d eine Kante von v nach v'' hat, die mit l gelabelt ist, und dann 0 oder mehr Kanten von v'' nach v'. Die Elemente von E^+ heißen *gelabelte Dominanzen.* Gelabelte Dominanzen haben den Zweck, die transitive Hülle der Kantenrelation fix ins Modell zu kodieren, weil diese nicht innerhalb von First-Order Logik ausgedrückt werden kann. Eine andere Option wäre es, in Modellen nur Kanten zu kodieren, und die transitive Hülle über Erweiterungen der Logik (Fixpunkte, Logik zweiter Stufe) zu erreichen. Dafür gibt es allerdings bis dato keine weiteren Gründe.

Das Knoten-Wort-Mapping weist jedem Knoten aus V ein Wort aus W zu.

Das Knoten-Attribut-Mapping weist jedem Knoten für jede Dimension die passende Attribute zu.

Der *Multigraphtyp* legt die möglichen Dimensionen, Wörter, Kantenlabels und Knotenattribute fest. Der Multigraphtyp ist ein Tupel $MT = (D, W, L, dl, A, T, dat)$. Die Dimensionen D, die Wörter W und die Labels L sind endliche Mengen aus Atomen. Das Dimension-Label-Mapping $dl \in D \to 2^L$ legt fest, welche Kantenbeschriftungen aus L für die Dimensionen möglich sind, und das Dimensions-Attribut-Typ-Mapping, welche Attribute mit welchen Werten.

Es gilt, ein Multigraph $M = (V, E^+, <, nw, na)$ der auf den Dimensionen D', den Wörtern W', den Kantenlabels L', den Attributen A' und den Typen T' definiert ist, hat den Multigraphtyp $MT = (D, W, L, dl, A, T, dat)$ gdw. $D' = D$, $W' \subseteq W$, $L' \subseteq L$, $A' = A$, $T' = T$, alle Kanten in Dimension $d \in D'$ haben nur Kantenbeschriftungen in dld, und alle Knotenattribute $a \in A'$ in Dimension $d \in D'$ haben Werte in $datda$, d.h. die, womit sie wohlgetypt sind.

2.2. Grammatiken

Mengen von Modellen in XDG werden durch Grammatiken beschrieben. Eine Grammatik besteht wie in Abbildung 2.3 aus einem Multigraphtyp MT, einem Lexikon lex und einer Menge von Prinzipien P. Für lex und P gilt, dass sie auf MT definiert sind, also auf dem gleichen Multigraphtyp.

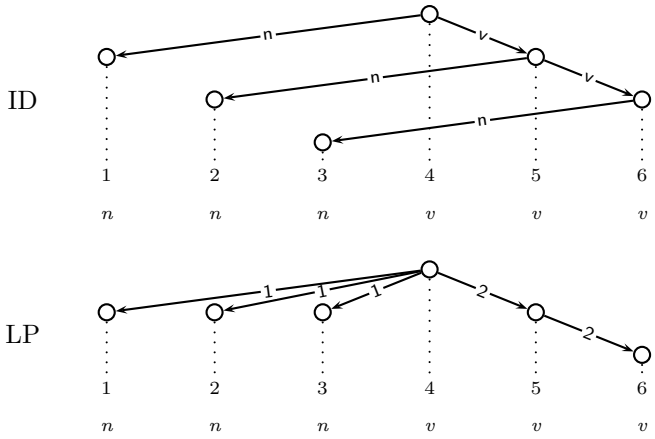


Abbildung 2.4.: Beispielanalyse der CSD-Grammatik für die Eingabe "nnnvvv"

2.2.1. Die Beispielgrammatik CSD

An dieser Stelle wollen wir eine Grammatik vorstellen, auf die wir im Laufe dieser Arbeit immer wieder zurückkommen werden. Die Grammatik CSD (Cross-Serial-Dependencies) beschreibt die Wortketten:

$$\text{CSD} = \{n^1 \dots n^k v^1 \dots v^k \mid k \geq 1\}$$

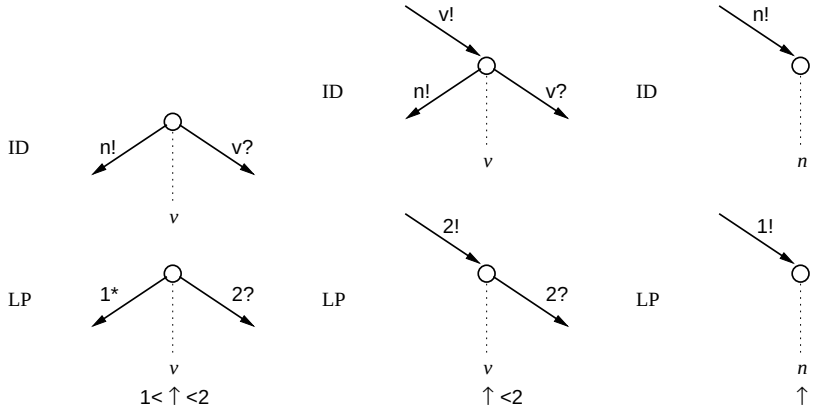
Also Sätze mit k Nomen n, gefolgt von k Verben v. Eine Analyse für den Satz *nnnvvv* zeigt Abbildung 2.4. Jedes Nomen und jedes Verb ist mit einem Index i mit $1 \leq i \leq k$ versehen, und es muss gelten, dass das Nomen n^i ein Argument des Verbes v^i ist. Solche Satzkonstruktionen können nicht mehr mit kontextfreien Grammatiken modelliert werden und kommen z.B. in Nebensätzen des Holländischen vor, z.B.:

(omdat) ik Cecilia Henk de nijlpaarden zag helpen voeren
(dass) ich Cecilia Henk die Nilpferde sah helfen füttern
"(dass) ich Cecilia Henk die Nilpferde füttern sah"

Der Multigraphtyp $MTCSD = (D, W, L, dl, A, T, dat)$ der Grammatik hat die Dimensionen `ID` und `LP`, die Wörter `n` und `v` sowie die Menge $L = \{\texttt{n}, \texttt{v}, \texttt{1}, \texttt{2}\}$. `ID` steht dabei für *Immediate Dominance* und `LP` für *Linear Precedence*, in Analogie zu GPSG (Gazdar, Klein, Pullum & Sag 1985). Das Mapping dl legt fest, dass `n` und `v` für die Dimension `ID` verwendet werden, `1` und `2` für die Dimension `LP`. Die Menge der Attribute A ist $\texttt{in}, \texttt{out}, \texttt{order}$ wobei `in` und `out` den Typ $2^{L \times \{!,?,*,+\}}$ haben, womit beschrieben wird, wieviele eingehende/ausgehende Kanten mit einem bestimmten Label erlaubt sind. `order` hat den Typ $2^{L \times L}$, womit eine Ordnung auf den Kantenlabels beschrieben wird.

2.2.2. Lexikon

Das Lexikon bildet Wörter auf Mengen von Lexikoneinträgen ab. Ein Lexikoneintrag legt die Werte der lexikalischen Attribute der jeweiligen Dimensionen fest. In Abbildung 2.5 sieht man die Lexikoneinträge für die Nomen und Verben der CSD Grammatik in schematischer Darstellung, und Abbildung 2.6 zeigt die beiden Lexikoneinträge für das Verb `v` in konkreter Form als Records. Teil (a) von Abbildung 2.5 besagt, dass falls das Verb die Wurzel ist (d.h. keine eingehende Kante hat), muss es auf Dimension `ID` genau eine ausgehende Kante mit



(a) (b) (c)

Abbildung 2.5.: Schematische Darstellung des Lexikons der CSD-Grammatik

$$\text{v} \mapsto \left\{ \begin{array}{l} \left\{ \begin{array}{lcl} ID & : & \left\{ \begin{array}{rcl} in & : & \{\} \\ out & : & \{n!, v?\} \end{array} \right\} \\ LP & : & \left\{ \begin{array}{rcl} in & : & \{\} \\ out & : & \{1*, 2?\} \\ order & : & \{(1, \uparrow, 2)\} \end{array} \right\} \end{array} \right\}, \\ \left\{ \begin{array}{lcl} ID & : & \left\{ \begin{array}{rcl} in & : & \{v!\} \\ out & : & \{n!, v?\} \end{array} \right\} \\ LP & : & \left\{ \begin{array}{rcl} in & : & \{2!\} \\ out & : & \{2?\} \\ order & : & \{(\uparrow, 2)\} \end{array} \right\} \end{array} \right\}, \\ \ldots \end{array} \right\}$$

Abbildung 2.6.: Die Lexikoneinträge für ein Verb der CSD-Grammatik in konkreter Form

$$t ::= c|x$$

$$\phi ::= \neg\phi|\phi_1 \wedge \phi_2|\exists x : \phi|t_1 = t_2|\psi$$

$$\begin{aligned} \psi \quad ::= \quad & v < v' \\ | \quad & v \xrightarrow{l}_d \rightarrow^*_d v' \\ | \quad & w(v) = w \\ | \quad & (t_1 \ldots t_n) \in a_d(v) \end{aligned}$$

Abbildung 2.7.: Terme, Formeln und Prädikate

$$\begin{aligned} tree_d \quad = \quad & \forall v : \neg(v \rightarrow^+_d v) \wedge & (1) \\ & \exists! v : \neg\exists v' : v' \rightarrow_d v \wedge & (2) \\ & \forall v : ((\neg\exists v' : v' \rightarrow_d v) \vee (\exists! v' : v' \rightarrow_d v)) \wedge & (3) \\ & \forall v : \forall v' : \forall l : \forall l' : v \xrightarrow{l}_d \rightarrow^*_d v' \wedge v \xrightarrow{l'}_d \rightarrow^*_d v' \Rightarrow l = l' & (4) \end{aligned}$$

$$\begin{aligned} csd_d \quad = \quad & \forall v, v' : \\ & v \xrightarrow{n}_d v' \Rightarrow \forall v'', v''' : v'' \rightarrow^+_d v \wedge v'' \xrightarrow{n}_d v''' \Rightarrow v''' < v' \end{aligned}$$

Abbildung 2.8.: Das Tree-Prinzip und das CSD-Prinzip

Beschriftung `n`, und auf Dimension `LP` beliebig viele ausgehende Kanten mit Beschriftung `1` haben. Die Dependenten, die durch diese Kanten bezeichnet werden, müssen links vom Verb (markiert mit $\uparrow$) stehen. Rechts vom Verb ist auf `ID` und `LP` höchstens ein Dependent, der mit einer Kante mit Beschriftung `v` bzw. `2` verbunden ist, erlaubt. Teil (b) beschreibt Verben, die als Dependent vorkommen. Für diese wird zusätzlich zu den Kanten aus Teil (a) noch genau eine eingehende Kante von mit Beschriftung `v` bzw. `2` verlangt, sie erlauben jedoch auf `LP` keine ausgehenden Kanten mit Label `1`. Die Ordnung $\uparrow < 2$ fordert, dass 2-Dependenten rechts des Knotens vorkommen. Teil (c) fordert für Nomen, dass sie auf beiden Dimensionen Dependent eines Knotens sein müssen. Die Kante muss mit `n` bzw. `1` beschriftet sein. Nomen können keine ausgehenden Kanten haben.

2.2.3. Prinzipien

Die Prinzipien sind das zentrale Element in XDG. Durch Prinzipien werden die Wohlgefomtheitsbedingungen der XDG-Modelle ausgedrückt. Ein Multigraph ist genau dann wohlgeformt, wenn er alle Prinzipien der Grammatik erfüllt.

Ein Grammatikschreiber kann entweder neue Prinzipien selbst schreiben, oder Prinzipien aus einer vordefinierten Prinzipienmenge auswählen, der *Prinzipienbibliothek*. Typische Prinzipien hieraus sind z.B. das Tree-Principle, das fordert, dass der Dependenzgraph einer bestimmten Dimension Baumform hat. Der Dependenzgraph darf also nur eine Wurzel haben, jeder Knoten hat genau eine Mutter oder keine Mutter, der Dependenzgraph darf keine Zyklen enthalten, und die Teilbäume unterhalb eines Knotens mit verschiedenen Labels sind disjunkt. Ein weiteres wichtiges Prinzip ist das Valency-Principle, das in Abhängigkeit des Lexikons die möglichen ein- und ausgehenden Kanten eines Knotens festlegt.

Formalisiert werden die Prinzipien in First-Order Logik. Dabei gilt, dass die Prinzipien lexikalisiert sein können, also in Abhängigkeit des Lexikons stehen können, wie z.B. das Valency-

Principle, das die möglichen ein- und ausgehenden Kanten jedes Wortes in dessen Lexikoneinträgen definiert. Das Valency-Principle stellt sicher, dass die Kanten des Multigraphen nach den Spezifikationen des Lexikons zulässig sind.

Formal ist ein Prinzip p eine First-Order-Formel aus der Formelmenge ϕ, die über Termen gebildet wird. ϕ ist definiert wie in Abbildung 2.7, wobei die hier fehlenden, üblichen logischen Operatoren bzw. Quantoren $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\forall$, $\exists!$ und $\neq$ mit Hilfe der Formeln aus ϕ hergeleitet werden können. Sie werden daher in XDG als syntaktischer Zucker bereitgestellt.

Ein Term t ist entweder eine Individuenkonstante oder eine Individuenvariable. Die Individuenkonstanten sind in Bezug auf den Multigraphtyp $MT = (D, W, L, dl, A, T, dat)$ Elemente aus der Menge $C = D \cup W \cup L \cup A \cup F \cup \mathbb{N}$, also Dimensionen, Wörter, Kantenbeschriftungen, Attribute, Komponenten von Attributwerten und Knoten. Komponenten von Attributwerten werden durch die Menge $F = \cup\{Fd_1 \cup \ldots \cup Fd_n | 2^{Fd_1 \times \ldots \times Fd_n} \in T\}$ dargestellt. Individuenvariablen können z.B. Knotenvariablen, Wortvariablen usw. sein.

Um über die Multigraph-Modelle sprechen zu können, stellt die Logik in XDG einige vordefinierte Prädikate ψ (Abbildung 2.7) zur Verfügung. Hierbei ist die Interpretation des Prädikats $<$ die strenge totale Ordnung von V definiert aus dem Multigraph. Das Prädikat $v \xrightarrow{l}_d \rightarrow_d^* v'$ steht für eine gelabelte Dominanz, d.h. einen Pfad in Dependenzgraphen der Dimension d vom Knoten v zum Knoten v' mit mindestens einer Kante. Die erste Kante in diesem Pfad muss die Kantenbeschriftung l haben. $w(v) = w$ vergleicht eine Individuenkonstante w aus W mit dem Wort, mit dem der Knoten v über das Node-Word-Mapping assoziiert ist. $(t_1 \ldots t_n) \in a_d(v)$ prüft, ob das Tupel $(t_1 \ldots t_n)$ ein Element des Attributs a auf Dimension d des Knotens v ist.

Drei Shortcuts werden vereinbart, um weitere Kantenbeziehungen zu beschreiben:

$$v \rightarrow_d^+ v' \stackrel{def}{=} \exists l : v \xrightarrow{l}_d \rightarrow_d^* v'$$

definiert einen Pfad von v nach v' der Länge mindestens eins, allerdings mit beliebiger Kantenbeschriftung.

$$v \xrightarrow{l}_d v' \stackrel{def}{=} v \xrightarrow{l}_d \rightarrow_d^* v' \wedge \neg\exists v'' : v \rightarrow_d^+ v'' \wedge v'' \rightarrow_d^+ v'$$

beschreibt einen Pfad der Länge eins mit Kantenbeschriftung l von v nach v' (d.h. eine gelabelte Kante) in Dimension d, indem verlangt wird, dass es keinen Knoten v'' zwischen v und v' geben darf.

$$v \rightarrow v' \stackrel{def}{=} \exists l : v \xrightarrow{l}_d v'$$

verlangt eine Kante von v nach v' in Dimension d mit beliebiger Kantenbeschriftung.

Damit sind die Prinzipien vollständig spezifiziert. Neue Prinzipien können also in First-Order Logik definiert werden, wie in Abbildung 2.8 als Beispiel zu sehen ist. Zu sehen ist das Baumprinzip, das unter anderem von der CSD Grammatik verwendet wird. Die erste Zeile beschreibt die Eigenschaft, dass ein Baum keine Zyklen enthalten darf, indem für alle Knoten gelten muss, dass es keinen Pfad zu sich selbst gibt. Die zweite Zeile besagt, dass es genau einen Knoten gibt, der keinen Vorgänger hat, also die Wurzel ist. Dieser Knoten darf nicht am Ende eines Pfades liegen. Zeile drei verlangt, dass jeder Knoten höchstens einen Mutterknoten hat. Die vierte Zeile stellt sicher, dass zwei Unterbäume eines Knotens mit unterschiedlicher Beschriftung disjunkt sind. Damit stellt man auch sicher, dass es zwischen zwei Knoten nicht mehr als eine Kante gibt.

Die CSD Grammatik verwendet folgende Prinzipien:

- auf `ID`: Tree-Prinzip, Valency-Prinzip und CSD-Prinzip
- auf `LP`: Tree-Prinzip, Valency-Prinzip und Order-Prinzip
- auf `ID` und `LP`: Climbing-Prinzip.

Während das Tree-Prinzip und Valency-Prinzip typische Prinzipien sind, und daher später für die Implementierung einfach aus einer Prinzipienbibliothek ausgewählt werden können, ist das CSD-Prinzip ein speziell für diese Grammatik entwickeltes Prinzip, das sicherstellt, dass alle `n`-Dependenten eines Verbes v nach den `n`-Dependenten der Verben über v (die v dominieren) folgen müssen. In Abbildung 2.8 ist das CSD-Prinzip als Formel in First-Order Logik zu sehen.

2.2.4. Modelle

Modelle einer Grammatik $m\ G = (MT, lex, P)$ sind alle Multigraphen M, für die gilt:

1. M hat Multigraphtyp MT
2. M erfüllt das Lexikon lex
3. M erfüllt alle Prinzipien P

Ein Multigraph $M = (V, E^+, <, nw, na)$ beschreibt einen String $s\ M = nw\ p_1 \ldots nw\ p_n$, wobei für $1 \leq i < j \leq n : (p_i, p_j) \in <$. Die Stringsprache einer Grammatik ist dann $L\ G = \{s\ M | M \in m\ G\}$.

Ein Eingabestring $S = a_1 \ldots a_n$ ist in $L\ G$, wenn es ein Modell mit einer Knotenmenge $V = \{1, \ldots, n\}$ gibt, für die gilt, dass für alle $v \in V : nw\ v \rightarrow a_v$ und $<= 1 < \ldots < n$.

2.3. Eigenschaften

2.3.1. Ausdrucksstärke

Zur Mächtigkeit von XDG kann gesagt werden, dass alle kontextfreien Grammatiken von XDG beschrieben werden können (Debusmann 2006). Das ist aber nicht alles. XDG kann noch weitere Sprachen aus dem Bereich der schwach-kontextsensitiven Sprachen beschreiben, z.B. die Sprache $A^N B^N C^N \ldots$ (mit beliebig vielen Blöcken der Länge N). Die Grammatikformalismen TAG und CCG können nur Sprachen mit vier Blöcken beschreiben. Desweiteren kann XDG sogar Phänomene modellieren, die von keinem schwach-kontextsensitiven Grammatikformalismus abgedeckt werden, z.B. Scrambling (Debusmann 2007*b*). Ein Beweis, dass XDG reguläre Dependenzgrammatiken (Kuhlmann & Möhl 2007), (Kuhlmann 2007) modellieren kann und damit auch Linear Context-Free Rewriting Systems (LCFRS) ist laut Ralph Debusmann zwar noch nicht publiziert, aber fertig.

2.3.2. Abschlusseigenschaften

Die durch XDG beschreibbaren Sprachen sind abgeschlossen unter Vereinigung und Schnitt, nicht jedoch unter Komplement (Debusmann 2007*b*).

2.3.3. Komplexität

Das universelle Erkennungsproblem von XDG ist PSPACE-vollständig, das fixe Erkennungsproblem NP-vollständig (Debusmann 2007*a*).

3. XDG Development Kit

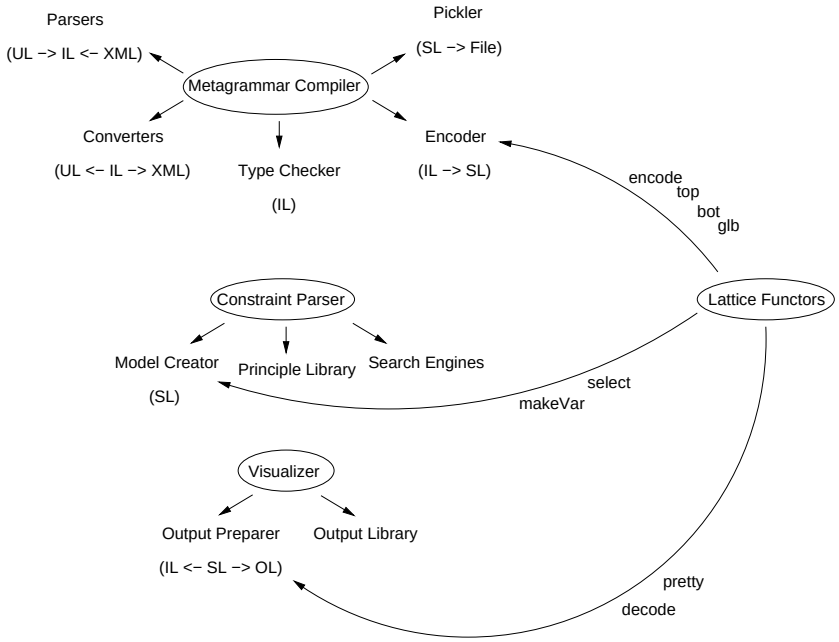


Abbildung 3.1.: Architektur von XDK

Das Extensible Grammar Development Kit (XDK) ist die Implementation des XDG Grammatikformalismus. Mit Hilfe von XDK können XDG Grammatiken aufgeschrieben werden, und Sätze mit der Grammatik geparst werden bzw. es können Lösungen gefunden werden.

3.1. Architektur

Das XDK liefert verschiedene Werkzeuge um Grammatiken aufzuschreiben und um Lösungen für Eingabesätze zu finden. Darunter sind vor allem die XDK Description Language, mit deren Hilfe Grammatiken aufgeschrieben werden können, und ein grafisches Interface um z.B. die Lösungen anzuzeigen. Wie in Abbildung 3.1 zu sehen, besteht XDK aus drei Hauptmodulen:

- dem Metagrammar Compiler, um die aufgeschriebene Grammatik für den Constraint Parser vorzubereiten
- dem Constraint Parser, der die Lösungen sucht
- dem Visualisierer, der die Ergebnisse grafisch aufbereitet und darstellt

Die Lattice-Funktoren stellen Funktionalität für alle drei Module bereit, u.a. Vererbung, die durch die *greatest lower bound*-Verbandsoperation modelliert wird (Metagrammar Compiler), das Auswählen von Lexikoneinträgen mithilfe des Selection Constraint (Duchier 1999), (Duchier 2003) (Constraint Parser), und die Aufbereitung von Lösungen (Visualisierer).

3.1.1. Metagrammar Compiler

Der Metagrammar Compiler transformiert Grammatiken, die mit Hilfe der Description Language geschrieben wurden, in die Solver Language (SL), mit der XDK intern arbeitet. Bei der Übersetzung in die Solver Language werden unter anderem die Klassenhierarchien des Lexikons, die XDG vorsieht und durch die Description Language aufgeschrieben werden können, aufgelöst, und für jedes Wort ein kompletter Eintrag erstellt. Das Lexikon wird also flach gemacht. Die Description Language stellt drei verschiedene Syntaxen zur Verfügung:

1. Die User Language (UL), mit der XDG Grammatiken von Hand aufgeschrieben werden können.

2. Die XML Language (XML) ist eine XML-basierte Sprache zur automatischen Generierung von Grammatiken.

3. Die Intermediate Language (IL) ist eine Sprache, die die Mozart/Oz Syntax verwendet, und die einerseits dazu gebraucht werden kann, Grammatiken automatisch zu generieren. Andererseits wird sie intern im XDK verwendet.

Der Metagrammar Compiler kann auch Grammatiken aus IL in UL oder XML umwandeln, sowie kompilierte SL Grammatiken in Dateien schreiben. XML und IL sind aufgrund ihrer Ausführlichkeit nicht geeignet, um Grammatiken von Hand zu schreiben. Deshalb werden wir in den folgenden Kapiteln nur noch die lesbare User Language betrachten.

Vor dieser Arbeit konnte in der XDK Description Language nur der Multigraphtyp und das Lexikon beschrieben werden, nicht aber die Prinzipien. Die Prinzipien mussten bisher entweder aus der vordefinierten Prinzipienbibliothek ausgewählt werden, ähnlich einem Bausteinsystem, oder mühsam von Hand selbst in Mozart/Oz entwickelt werden:

$$\begin{array}{lccccc} \text{Grammatik G} = (& \text{MT}, & \text{lex}, & \text{P} &) \\ & \uparrow & \uparrow & \uparrow & \\ & \checkmark & \checkmark & \times & \end{array}$$

Wie der Multigraphtyp und das Lexikon mit der XDK Description Language aufgeschrieben werden, und wie die Prinzipien, die in der Description Language fehlen, in XDK implementiert werden, werden wir in den nächsten Kapiteln sehen.

3.1.2. Constraint Parser

Der Constraint Parser baut aus der vom Metagrammar Compiler in die Solver Language kompilierten Grammatik und einem Eingabesatz ein Constraint Satisfaction Problem (CSP) mit Hilfe des Model Creator. Dabei erweitert er das Constraint Satisfaction Problem um die Prinzipien aus der vordefinierten Prinzipienbibliothek. Die Search Engines finden die Lösungen des CSP. Dieser Ablauf ist in Abbildung 3.2 illustriert.

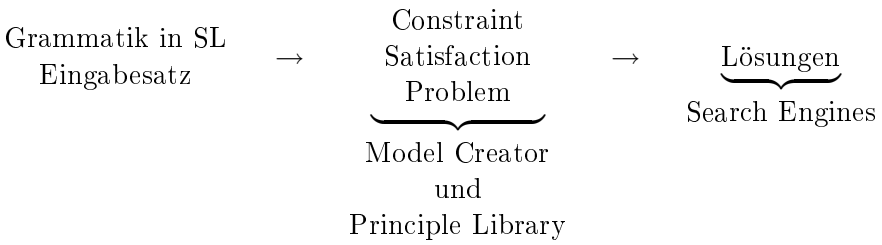


Abbildung 3.2.: Constraint Parser

3.1.3. Visualisierer

Der Visualisierer kann die vom Constraint Parser gefundenen Lösungen entweder grafisch darstellen, oder als LaTeX-Quellcode. Wie die Prinzipien ist auch der Visualisierer als Bibliothek implementiert. Es lassen sich bequem neue Visualisierungsmodule hinzufügen. Der Visualisierer kann dabei kann nicht nur mit vollständigen, sondern auch mit partiellen Lösungen umgehen.

3.2. XDK Description Language

Wir werden die XDK Description Language nicht komplett definieren, sondern sie anhand eines Beispiels erklären. Ebenso werden wir im Beispiel sehen, wie der Multigraphtyp und das Lexikon beschrieben werden. Dafür werden wir die CSD-Grammatik aus Kapitel 2 verwenden, und zeigen, wie diese in XDK implementiert wird.

3.2.1. Multigraphtyp-Definition

```
defdim id {
  deftype "id.label" {n v}
  deflabeltype "id.label"
  defentrytype {in: tuple("id.label" {"!","?","+","*"})
                out: tuple("id.label" {"!","?","+","*"}}

  [ ... ]
}
```

Abbildung 3.3.: Multigraphtyp-Definition Beispiel: die Dimension ID der CSD-Grammatik

Zuerst definieren wir den Multigraphtyp unserer Grammatik. In der XDK Description Language können hierzu Typen `T` an Typnamen `a` gebunden werden mit dem Eintrag `deftype a T`. Im Beispiel in Abbildung 3.3 wird die Menge $\{n\ v\}$ an den Typnamen `id.label` gebunden. In der nächsten Zeile wird die Menge `id.label` als Menge der möglichen Kantenbeschriftungen für die ID-Dimension definiert (`deflabeltype`). Die beiden nächsten Zeilen legen die lexikalischen Attribute der Knoten fest mit dem Befehl `defentrytype`. Das `in`-Attribut hat z.B. als Typ eine Menge aus Paaren, wobei jedes Paar zuerst einen Wert aus `id.label` hat, gefolgt von einem Atom aus der Menge {"!" "?" "+" "*"}.

3.2.2. Lexikon-Definition

Für die Beschreibung des Lexikons können lexikalische Klassen verwendet werden, die eine Repräsentation von lexikalischen Einträgen sind, die über beliebig viele Variablen abstrahieren können. Sie sind damit ähnlich zu Templates in anderen Grammatikformalismen wie PATR-II (Shieber 1984). Außerdem kann mit den lexikalischen Klassen eine Vererbungshierarchie aufgebaut werden. Als Beispiel entwickeln wir ein Lexikon für die CSD-Grammatik. Wir definieren zwei lexikalische Klassen: eine für Verben und eine für Nomen (vgl. Abbildung 2.5).

Die Klasse für Verben sieht wie folgt aus. Sie abstrahiert über die Variable `Word`, die das Wort des lexikalischen Eintrags repräsentiert. Wörter werden den Lexikoneinträgen über das Attribut `word` auf der XDK-eigenen `lex`-Dimension zugeordnet:

```
defclass "verb" Word {
  dim id {out: {n! v?}}
  dim lp {out: {"1"* "2"?}
          order: <"1" "^" "2">}
  dim lex {word: Word}}
```

Die Klasse für die Nomen sieht so aus:

```
defclass "noun" Word {
  dim id {in: {n!}}
  dim lp {in: {"1"!}}
  dim lex {word: Word}}
```

Im folgenden Schritt benutzen wir die lexikalischen Klassen. Wir definieren mit dem Schlüsselwort `defentry` zuerst einen Eintrag für Verben, die als Wurzelknoten vorkommen:

```
defentry {
  "verb" {Word: "v"}
  dim lp {out: {"1"*}}}
```

Der Eintrag erbt von der lexikalischen Klasse `verb`, übernimmt damit alle Attribute der Klasse `verb`, und erlaubt zusätzlich auf der Dimension `lp` noch beliebig viele ausgehende Kanten mit dem Label `1`.

Ein Verb, das kein Wurzelknoten ist, würde die Klasse `verb` noch um zwei eingehende Kante ergänzen, so dass das Verb auf beiden Dimensionen genau ein Dependent eines Verbes ist:

```
defentry {
  "verb" {Word: "v"}
  dim id {in: {v!}}
  dim lp {in: {"2"!}}}
```

Ein konkretes Nomen muss die Klasse `noun` gar nicht erweitern, sondern nur davon erben, dementsprechend kurz ist der Eintrag für das Nomen n:

```
defentry {
  "noun" {Word: "n"}}
```

Der Metagrammar Compiler erzeugt aus der Klassenhierarchie und den Eintragsdefinitionen Lexikoneinträge, so, als hätten wir das Lexikon nicht mit Hilfe von Klassen implementiert, sondern für jedes Wort alle Attribute angegeben. Er macht das Lexikon also flach:

```
defentry {
  dim id {in: {}
          out: {n! v?}}
  dim lp {in: {}
          out: {"1"* "2"?}
          order: <"1" "^" "2">}
  dim lex {word: "v"}}

defentry {
  dim id {in: {v!}
          out: {n! v?}}
  dim lp {in: {"2"!}
          out: {"2"?}
          order: <"^" "2"}
  dim lex {word: "v"}}

defentry {
  dim id {in: {n!}}
  dim lp {in: {"1"!}}
  dim lex {word: "n"}}
```

3.2.3. Prinzipien-Instanziierung

```
defdim id {

  [ ... ]

  useprinciple "principle.graph" {
    dims {D: id}}
  useprinciple "principle.tree" {
    dims {D: id}}
  useprinciple "principle.valency" {
    dims {D: id}}
  useprinciple "principle.csd" {
    dims {D: id}}
}
```

Abbildung 3.4.: Prinzipien-Instanziierung Beispiel: Die Dimension ID der CSD-Grammatik

Bei der Prinzipien-Instanziierung wird eine Teilmenge der Prinzipien der Prinzipienbibliothek ausgewählt, um die Wohlgeformtheitsbedingungen der Grammatik festzulegen. Um wiederverwendbar zu sein, abstrahieren viele der Prinzipien aus der Prinzipienbibliothek über die Dimensionen, die sie einschränken. Diese *Dimensionsvariablen* werden bei der Instanziierung an konkrete Dimensionen gebunden. In Abbildung 3.4 werden so mit dem Schlüsselwort `useprinciple` die Prinzipien `principle.graph`, `principle.tree`, `principle.valency` und `principle.csd` instanziiert, und jeweils die Dimensionsvariable `D` an die konkrete Dimension `id` gebunden.

3.3. Constraint Parser

Das Herzstück vom XDK, der Constraint Parser, muss zum Finden von Lösungen Multigraphen modellieren und die Prinzipienbibliothek implementieren. Wie das funktioniert, wird in den nächsten Kapiteln gezeigt.

3.3.1. Modellierung von Multigraphen

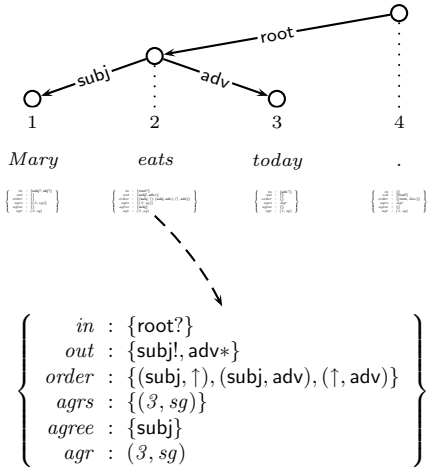


Abbildung 3.5.: Dependenzgraph für Peter eats today

Der Constraint Parser modelliert Multigraphen mithilfe von endlichen Mengen von natürlichen Zahlen. Der Dependenzgraph aus Abbildung 3.5 z.B. hat vier Knoten. Der Knoten für das Wort eats hat zwei Töchter. Bei der Modellierung des Dependenzgraphen wird für jeden Knoten ein *Knotenrecord* erzeugt, der die ausgehenden Kanten als Töchtermenge repräsentiert. Für das Wort eats sieht der Knotenrecord so aus:

$$\left\{ \begin{array}{rl} index & = 2 \\ word & = eats \\ nodeSet & = \{1,2,3,4\} \\ model & = \left\{ daughtersL = \left\{ \begin{array}{l} \mathsf{adv} = \{3\} \\ \mathsf{root} = \{\} \\ \mathsf{subj} = \{1\} \end{array} \right\} \right\} \end{array} \right\} \tag{3.1}$$

Das Attribut index bezeichnet den Index des Knotens, der den Knoten eindeutig benennt. Das Attribut nodeSet enthält alle Knoten, bzw. deren Indizes, die der Dependenzgraph enthält. Der Unterrecord daughtersL enthält für jedes Kantenlabel die Knoten, die über dieses Label direkt über eine Kante erreicht werden können.

In Mozart/Oz Syntax wird der Record so dargestellt:

```
o(index: 2
  word: eats
  nodeSet: {1 2 3 4}#4
  model: o(daughtersL: o(adv: {3}#1                    (3.2)
                         root: {}#0
                         subj: {1}#1)))
```

Da in Mozart/Oz jeder Record einen Namen haben muss, werden die `o` Zeichen als Dummynamen verwendet. Mengen werden in geschweiften Klammern geschrieben, gefolgt von Ihrer Kardinalität.

Der Constraint Parser erzeugt für jeden Knoten noch weitere Mengen neben daughtersL. Diese sind in Anhang, Abschnitt A.2 aufgelistet. Die Mengen werden vor allem für die bequemere und effizientere Implementation der Prinzipien verwendet. Der komplette Record des Knotens für das Wort eats mit allen Mengen sieht dann so aus:

```
o(index: 2
  word: eats
  nodeSet: {1 2 3 4}#4
  model: o(mothers: {4}#1
           daughters: {1 3}#2
           up: {4}#1
           down: {1 3}#2
           index: 2
           eq: {2}#1
           equp: {2 4}#2
           eqdown: {1 2 3}#3
           labels: {5}#1                               (3.3)
           mothersL: o(adv: {}#0
                       root: {4}#1
                       subj: {}#0)
           daughtersL: o(adv: {3}#1
                         root: {}#0
                         subj: {1}#1)
           upL: o(adv: {}#0
                  root: {4}#1
                  subj: {}#0)
           downL: o(adv: {3}#1
                    root: {}#0
                    subj: {1}#1)))
```

Die Attribute werden in den Unterrecords attrs (nichtlexikalische Attribute) und entry (lexikalische Attribute) modelliert. Der Knoten für das Wort eats mit den Attributen sieht dann so aus:

```
o(index: 2
  word: eats
  nodeSet: {1 2 3 4}#4
  entryIndex: 1
  model: o(daughtersL: o(adv: {3}#1
                         root: {}#0
                         subj: {1}#1))
  attrs: o(agr: 6)
  entry: o('in': o(adv: {0}#1
                   root: {0 1}#2
                   subj: {0}#1)
           out: o(adv: {0 1 2 3}#4
                  root: {0}#1
                  subj: {1}#1)
           order: {2 36 37}#3
           agrs: {6}#1
           agree: {6}#1))
```
(3.4)

Um einen kompletten Multigraph zu modellieren bekommt der Record für jede Dimension ein Attribut mit dem Namen der Dimension. Dieses Attribut hat als Wert den `model`-Unterrecord des Dependenzgraphen der Dimension.

3.3.2. Modellierung von Prinzipien

Ein Prinzip besteht aus zwei Teilen, der Prinzipiendefinition und einer Menge von Node-Constraint-Funktoren.

Bei der Prinzipiendefinition wird der Name des Prinzips definiert, eine Menge von Dimensionsvariablen, über die das Prinzip abstrahiert, und eine Menge von Node-Constraint-Funktoren und deren Priorität, die das Prinzip implementieren. Z.B. sieht die Prinzipiendefinition des CSD-Prinzipes so aus:

```
defprinciple "principle.csd" {
  dims {D}
  constraints {"CSD": 110}}
```

Der Name des Prinzipes ist “principle.csd”. Es abstrahiert über die Dimensionsvariable D und benutzt den Node-Constraint-Funktor CSD mit Priorität 110. Die Prioritäten legen die zeitliche Reihenfolge der Constraint-Anwendung fest—je höher die Priorität, desto eher wird der Constraint angewandt. Damit lassen sich in manchen Fällen Effizienzgewinne erzielen.

Die Node-Constraint-Funktoren werden direkt als Oz-Prozeduren implementiert, die den Namen `Constraint` tragen. Sie haben keinen Rückgabewert. Jeder Funktor hat vier Argumente:

1. Nodes: Die Liste der Knotenrecords, die den Multigraphen repräsentieren
2. G: Die Grammatik
3. GetDim: Eine Funktion, die Dimensionsvariablen konkrete Dimensionen zuweist

```
(  1) proc {Constraint Nodes G GetDim}
(  2)    DIDA = {GetDim 'D'}
(  3)    PosMs = {Map Nodes
(  4)             fun {$ Node} Node.pos end}
(  5) in
(  6)    for Node in Nodes do
(  7)       NDaughtersMs =
(  8)       {Map Nodes
(  9)        fun {$ Node} Node.DIDA.model.daughtersL.n end}
( 10)       NDaughtersUpM = {Select.union NDaughtersMs Node.DIDA.model.up}
( 11)       PosNDaughtersUpM = {Select.union PosMs NDaughtersUpM}
( 12)       NDaughtersM = Node.DIDA.model.daughtersL.n
( 13)       PosNDaughtersM = {Select.union PosMs NDaughtersM}
( 14)    in
( 15)       {FS.int.seq [PosNDaughtersUpM PosNDaughtersM]}
( 16)    end
( 17) end
```

Abbildung 3.6.: "CSD" Node-Constraint-Funktor

Beispiel: CSD-Prinzip

In Abbildung 3.6 ist als Beispiel der Node-Constraint-Funktor "CSD" des CSD-Prinzips zu sehen. In Zeile 2 holt sich der Funktor die konkrete Dimension, die durch die Dimensionsvariable `D` repräsentiert wird. In der dritten und vierten Zeile wird eine Liste mit den Positionsmengen der Knoten erstellt.

In der `for`-Schleife über die Knoten werden in jedem Durchgang die n-Dependenten aller Knoten bestimmt (Zeile 7–9). In Zeile 10 werden die n-Dependenten der Vorgänger des aktuellen Knotens `Node` bestimmt, und in Zeile 11 deren Positionen ermittelt. In Zeile 12 werden die n-Dependenten des aktuellen Knotens bestimmt, und in Zeile 13 deren Position. In Zeile 15 schließlich wird mit Hilfe der Funktion `FS.int.seq` ausgedrückt, dass die Positionen der n-Dependenten der Vorgänger des aktuellen Knotens immer vor den Positionen der n-Dependenten des aktuellen Knotens stehen müssen. Die Funktion `FS.int.seq` bekommt zwei disjunkte Mengen aus den natürlichen Zahlen, und prüft, ob alle Elemente der ersten Menge kleiner sind als die Elementen der zweiten Menge.

In dem Beispiel ist gut zu sehen, wie die verschiedenen Mengen aus den Modellen verwendet werden. Allerdings sieht man anhand dieses Beispiels auch, dass die Implementierung eines Prinzips nicht trivial ist und u.U. kaum mehr Ähnlichkeiten mit dessen Formalisierung hat (Vgl.: 2.8). Damit kommen wir zur Problemstellung.

3.4. Problemstellung

Wir haben gesehen, wie der Multigraphtyp und das Lexikon und einer XDG Grammatik mit Hilfe der XDK Description Language aufgeschrieben werden können. Leider kann der letzte Teil einer XDG-Grammatik, die Prinzipien, nicht so schön entwickelt werden. Die XDK Description Language erlaubt nur, Prinzipien aus einer Prinzipienbibliothek auszuwählen, nicht aber neue zu entwickeln. Um neue Prinzipien hinzufügen zu können, muss der Grammatikschreiber sehr vertraut sein mit Mozart/Oz-Constraintprogrammierung, was man bei typischen

Grammatikschreibern wie z.B. Linguisten nicht voraussetzen kann.

Hier besteht also eine große Lücke zwichen der Formalisierung (XDG) und der Implementierung (XDK) des Grammatikformalismus. Diese Lücke wird in den nächsten Kapiteln geschlossen, wo die Entwicklung eines Werkzeugs namens PrincipleWriter (PW) beschrieben wird. Der PW erlaubt es, Prinzipien als Formeln in First-Order Logik aufzuschreiben, und diese dann automatisch in Mozart/Oz Constraints umzusetzen.

4. PrincipleWriter

4.1. Einführung

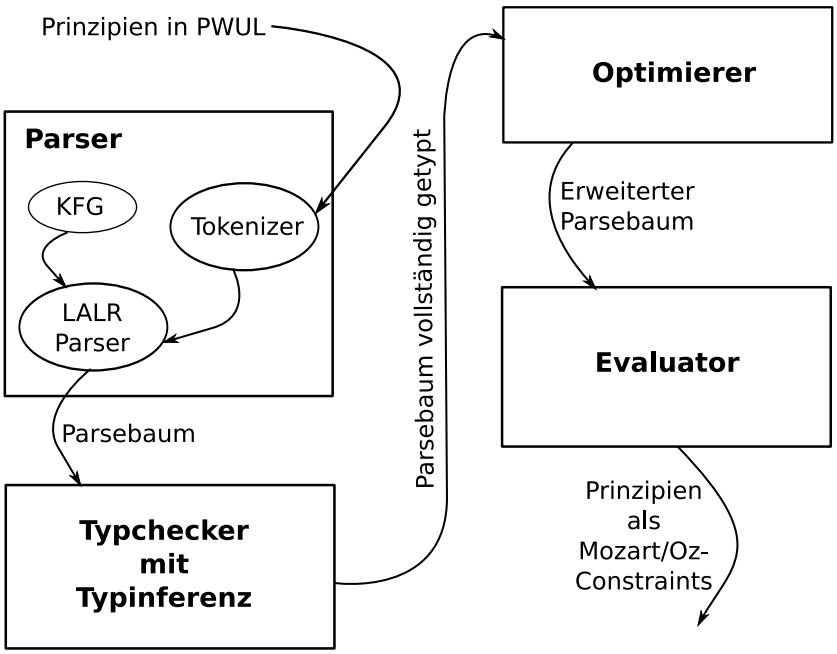


Abbildung 4.1.: Architektur des PrincipleWriters

In den beiden vorherigen Kapiteln haben wir gesehen, wie Grammatiken in XDG formalisiert werden, und mit Hilfe des XDK implementiert werden können. Während der Multigraphtyp und das Lexikon analog zur XDG Formalisierung mit Hilfe der XDK Description Language aufgeschrieben werden konnten, konnten Prinzipien nur aus einer Bibliothek von vordefinierten Prinzipien ausgewählt werden, oder mussten mühsam von Hand implementiert werden. Dass das bloße Auswählen von Prinzipien aus der vordefinierten Prinzipienbibliothek nicht immer reicht, haben wir anhand der CSD-Beispielgrammatik aus den beiden letzen Kapiteln gesehen, die neben drei vordefinierten Prinzipien auch ein Prinzip braucht, das neu implementiert werden musste.

In XDG werden Prinzipien in einer speziellen First-Order Logik formalisiert, die einige spezielle Prädikate bereitstellt, um Constraints über Multigraphen ausdrücken zu können. Die Prinzipien sind also logische Formeln. Zur Implementierung müssen die Prinzipien in Mozart/Oz-Mengenconstraints umgesetzt werden. Ein direkter Bezug zu den logischen Formeln aus XDG ist nicht offensichtlich. Von einem Prinzipienschreiber müssen also Programmierkenntnisse in Mozart/Oz Constraintprogrammierung verlangt werden, und um die Constraints zu optimie-

ren, müssen sogar Expertenkenntnisse verlangt werden. Die Optimierung von Prinzipien ist notwendig, da sie sonst wegen ihrer hohen Laufzeit nicht brauchbar sind.

Um diese Lücke zwischen der Formalisierung und der Implementation zu schließen wird in den nächsten Kapiteln die Entwicklung eines Tools beschrieben, dem PrincipleWriter, das es Grammatikschreibern erlaubt, Prinzipien als logische Formeln aufzuschreiben. Dazu wurde die PrincipleWriter User Language (PWUL) entwickelt, aus der der PrincipleWriter dann die Prinzipien in Mozart/Oz Constraints umsetzt. Mit Hilfe der XDK Description Language und des PrincipleWriters können Grammatiken vollkommen analog zu ihrer Formalisierung aus XDG in XDK implementiert werden.

4.1.1. Architektur des PrincipleWriters

Der PrincipleWriter besteht wie in Abbildung 4.1 zu sehen ist aus vier Komponenten:

1. Dem Parser, der die logische Formel zur weiteren Verarbeitung in die abstrakte Syntax, die eine Baumstruktur ist, bringt.
2. Einem Modul, das die fehlenden Typen inferiert, und einen Typchecker beinhaltet.
3. Dem Optimizer.
4. Dem Evaluator, der aus der Baumstruktur Mozart/Oz-Constraints erzeugt.

Der PrincipleWriter bekommt als Eingabe eine Datei, die ein Prinzip als logische Formel in der PWUL Syntax enthält. Der Parser enthält einen Tokenizer aus der Mozart/Oz Modulbibliothek, der die Eingabe zerlegt. Dann bringt der Parser mit Hilfe des LALR-Parsers aus der Mozart/Oz Modulbibliothek die zerlegte Eingabe in Baumform.

Das Modul, das das Typchecking und die Typinferenz übernimmt, ergänzt dann in der Baumstruktur die fehlenden Typen und gibt Fehlermeldungen bei falscher Typisierung, bzw. bei nicht zu inferierenden Typen aus.

Der vollständig getypte Baum wird dann vom Optimizer modifiziert. In der derzeitigen Version des Optimizers werden spezielle Teilformeln erkannt, für die optimierte Constraints eingesetzt werden können.

Der Evaluator wandelt den optimierten Baum in Mozart/Oz-Constraints um, die dann in eine Datei geschrieben werden, die direkt in das XDK eingefügt werden kann, um die Prinzipienbibliothek zu erweitern.

Da der PrinzipleWriter ein eigenständiges Werkzeug ist, muss die XDK Description Language nicht modifiziert werden, bestehende Grammatiken können weiterhin verwendet werden, und innerhalb der Grammatiken können handgeschriebene und automatisch kompilierte Prinzipien beliebig gemischt werden. Außerdem erlaubt dieses Vorgehen das manuelle Nachbearbeiten der automatisch erzeugten Mozart/Oz Constraints, um eventuelle weitere Optimierungen durch Experten zu erzielen. Die PrincipleWriter User Language (PWUL) und die einzelnen Komponenten des PrincipleWriters werden in den nächsten Kapiteln genau erläutert.

4.2. PWUL und die dazugehörige Grammatik

Um Prinzipien als logische Formeln aufschreiben zu können wurde eine spezielle Syntax entwickelt, die *PrincipleWriter User Language (PWUL)*. Diese soll in diesem Kapitel zuerst vor-

gestellt werden. Danach werden wir uns ansehen, wie die zugehörige Grammatik aussieht, und wie diese im *PrinciplеWriter* verarbeitet wird.

4.2.1. PrincipleWriter User Language

Um die *PrinciplеWriter User Language* zu erklären, wollen wir das CSD-Prinzip aus der CSD Grammatik aufschreiben. Zur Erinnerung, das Prinzip war definiert als:

$$csd_d = \forall v, v' : v \xrightarrow{n}_d v' \Rightarrow \forall v'', v''' : v'' \rightarrow_d^+ v \wedge v'' \xrightarrow{n}_d v''' \Rightarrow v''' < v'$$

Bevor wir die logische Formel selbst notieren, müssen wir den Namen und die Dimensionen, über die abstrahiert werden soll, angeben. In unserem Fall wird nur über eine Dimension mit der Dimensionsvariablen `D` abstrahiert:

```
defprinciple "principle.csdPW" {
  dims {D}
  constraints {

    [ ... ]

  }
}
```

Zu sehen ist, dass in PWUL ein Prinzip analog zur Formalisierung in XDG aus einem Namen, einer Menge von Dimensionsvariablen und Constraints besteht. Vor der Prinzipiendefinition (`defprinciple`) selbst könnte man noch Namen an Typen binden mit Ausdrücken wie `deftype a T`, der den Typ `T` an den Typnamen `a` bindet. Der Constraint des CSD-Prinzips kann nun als logische Formel eingefügt werden. Das komplette Prinzip sieht dann so aus:

```
defprinciple "principle.csdPW" {
  dims {D}
  constraints {

    forall V::node:
      forall V1::node:
        edge(V V1 n D) =>
          forall V2::node:
            forall V3::node:
              dom(V2 V D) & edge(V2 V3 n D) => V3<V1

  }
}
```

Die komplette Liste von Operatoren und Prädikaten der PWUL ist in Figure 4.2 zu sehen. Die Operatoren der linken Spalte sind absteigend ihrer Bindungsstärke sortiert, wobei die Quantoren gleich stark binden. Die Operatoren der rechten Spalte binden alle gleich stark, aber schwächer als die Operatoren der linken Spalte.

Figure 4.3 zeigt die Syntax für Ausdrücke von Mengen und Tupel, die auch analog zur XDG Formalisierung gehalten sind. Der Zugang zu Attributen unterscheidet sich von der XDG Formalisierung, da man in PWUL auf Attribute zugreifen kann, ohne abfragen zu müssen, ob ein bestimmtes Tupel in der entsprechenden Menge enthalten ist. In PWUL liefert der Term `V.D.entry.a` das lexikalische Attribut `a` von Knoten `V` in der Dimension `D`. In XDG kann

XDG	PWUL	XDG	PWUL
$\neg$	`~`	$=$	`=`
$\wedge$	`&`	$\neq$	`~=`
$\vee$	`|`	$\in$	`in`
$\Rightarrow$	`=>`	$\notin$	`notin`
$\Leftrightarrow$	`<=>`	$\subseteq$	`subseteq`
$\forall$	`forall`	$\parallel$	`disjoint`
$\exists$	`exists`	$\cap$	`intersect`
$\exists!$	`existsone`	$\cup$	`union`
		$\setminus$	`minus`

XDG	PWUL
$V1 \rightarrow_D V2$	`edge(V1 V2 D)`
$V1 \stackrel{L}{\rightarrow}_D V2$	`edge(V1 V2 L D)`
$V1 \rightarrow^*_D V2$	`domeq(V1 V2 D)`
$V1 \rightarrow^+_D V2$	`dom(V1 V2 D)`
$V1 \stackrel{L}{\rightarrow}^+_D V2$	`dom(V1 V2 L D)`
$<$	`<`
$w(v) = w$	`v.word = w`

Abbildung 4.2.: Operatoren und Prädikate der PWUL

XDG	PWUL	
$\{a_1, \ldots, a_n\}$	$\{\mathtt{a}_1 \ldots \mathtt{a}_n\}$	(Mengen)
$(a_1, \ldots, a_n)$	$[\mathtt{a}_1 \ldots \mathtt{a}_n]$	(Tupel)
$\{(a_1^1 \ldots, a_n^1) \ldots (a_1^k \ldots, a_m^k)\}$	$\{[\mathtt{a}_1^1 \ldots, \mathtt{a}_n^1] \ldots [\mathtt{a}_1^k \ldots, \mathtt{a}_m^k]\}$	(Mengen von Tupeln)
	`V.D.entry.a`	(Zugang zu lex. Attributen)
	`V.D.attrs.a`	(Zugang zu nicht-lex. Attributen)
	`V::node`	(Typannotation)

Abbildung 4.3.: Ausdrücke in PWUL

man nur $(t_1 \ldots t_n) \in a_d(v)$ schreiben. Desweiteren kann man in PWUL auch Tupel-wertige Attribute haben und nicht nur Mengen von Tupeln wie in XDG. Typannotationen gibt es in XDG gar nicht, sind für die Übersetzung der Formeln in Constraints aber essentiell.

Die erlaubten Typen sind in Figure 4.4 aufgelistet. Sie folgen auch der XDG Formalisierung. Für die Typen der rechten Spalte gilt, dass `Dom` Typen aus der linken Spalte sein müssen. Desweiteren gilt, dass in der PrincipleWriter User Language alle Variablen mit einem Großbuchstaben beginnen müssen. Konstanten werden klein geschrieben.

4.2.2. Das Parsermodul des PrincipleWriters

Das Parsermodul des PrincipleWriters besteht aus drei Teilen. Einem Tokenizer, einer kontextfreien Grammatik und einem Parser.

Sowohl der Tokenizer als auch der Parser stammen aus der Funktionsbibiliothek von Mozart/Oz. Als Parser kommt ein LALR-Parser zum Einsatz. Die zugehörige Grammatik wird als Produktionsregeln in Mozart/Oz Syntax aufgeschrieben. Diese Regeln können in gängiger

XDG	PWUL	XDG	PWUL
$\{a_1, \ldots, a_n\}$	{a$_1$ ... a$_n$}	2^{Dom}	set(Dom)
$dl\ D$	label(D)	$Dom_1 \times \ldots \times Dom_n$	tuple(Dom$_1$ ... Dom$_n$)
V	node	$2^{Dom_1 \times \ldots \times Dom_n}$	set(tuple(Dom$_1$ ... Dom$_n$))
D	dim		
A	attr		
W	word		

Abbildung 4.4.: Typen der PWUL

```
conj(value(coord:7#7
           sem:dom(value(coord:7
                         sem:constant(token(coord:7
                                            sem:'V2')))
                   value(coord:7
                         sem:constant(token(coord:7
                                            sem:'V')))
                   value(coord:7
                         sem:constant(token(coord:7
                                            sem:'D')))))
     value(coord:7#7
           sem:ledge(value(coord:7
                           sem:constant(token(coord:7
                                              sem:'V2')))
                     value(coord:7
                           sem:constant(token(coord:7
                                              sem:'V3')))
                     value(coord:7
                           sem:constant(token(coord:7
                                              sem:n)))
                     value(coord:7
                           sem:constant(token(coord:7
                                              sem:'D'))))))
```

Abbildung 4.5.: Beispieloutput des Parsers für: $V2 \rightarrow_D V \wedge V2 \overset{n}{\rightarrow}_D V3$

Notation in Anhang B nachgelesen werden.

Das Parsermodul nimmt ein Prinzip, das in der PrincipleWriter User Language aufgeschrieben ist als Eingabe. Mit Hilfe des Tokenizers wird die Eingabe zerlegt. Der Parser baut aus den Tokens und der Grammatik einen Ableitungsbaum. Der Ableitungsbaum wird als Record realisiert. Eine vereinfachte Darstellung des Records ist in Figure 4.5 zu sehen. Im Feld `sem` beginnt ein neuer Unterbaum, bzw. ein Blatt. Das Feld `coord` bezeichnet die Zeilennummer des Tokens im Eingabestring, die später für Fehlermeldungen beim Typchecker verwendet wird.

4.3. Typchecker mit Typinferenz

Im letzten Kapitel haben wir gesehen, wie der Benutzer ein Prinzip mit Hilfe der PrincipleWriter User Language aufschreiben kann, und wie die Mozart/Oz-Repräsentation der abstrakten Syntax, die das Parsermodul aus der PWUL generiert, aussieht.

Da Typfehler vom Parser nicht erkannt werden, wird das Typcheckermodul gebraucht. Aber das Typcheckermodul kann noch mehr, es kann fast alle Typen inferieren, so dass der Benutzer praktisch gar keine Typen mehr annotieren muss, was das Schreiben von Prinzipien noch weiter vereinfacht. Warum Typannotationen überhaupt notwendig sind, werden wir später bei der Übersetzung der Formeln in Constraints in Mozart/Oz sehen.

Der Typchecker arbeitet direkt mit der Mozart/Oz-Repräsentation der abstrakten Syntax, die wie gesehen ein Record ist, der den Parsebaum repräsentiert. Der Parsebaum wird vom Typchecker um Typannotationen erweitert. Um die Arbeitsweise zu veranschaulichen verwenden wir Inferenzregeln. Dabei wurde kein vollständiges Typsystem entwickelt, die Regeln dienen nur zum einfachen Aufschreiben der verschiedenen Fälle. Als Beispiel die Regel für eine Variable mit Typannotation: [1]

$$\frac{\Gamma \vdash \texttt{x :: type}}{\Gamma \cup \{x \mapsto \mathit{type}\} \vdash \texttt{x :: type}} x \mapsto T \in \Gamma \Rightarrow T = \mathit{type}$$

Sie besagt, dass wenn wir einen Kontext Γ haben, und eine Variable `x` mit Typannotation `type` finden, wir $x \mapsto \mathit{type}$ in den Kontext aufnehmen, sofern die Variable `x` nicht schon im Kontext mit einem anderen Typ ist. Die Formel wird nicht verändert. Der Kontext mappt Variablen auf Typen. Die Seitenbedingungen der Regeln werden nicht nur für die Typinferenz, sondern vor allem auch für das Typchecking gebraucht. Wertet die Formel in der Seitenbendingung zu `false` aus, wird eine Fehlermeldung ausgegeben.

4.3.1. Arbeitsweise des Typcheckers

Um die Typen überprüfen bzw. inferieren zu können, steigt der Typchecker von der Wurzel des Parsebaums abwärts zu den Blättern und merkt sich auf dem Weg die Variablen, die noch zu inferieren sind, und die Variablen-Typ Paare, die er schon erkannt hat, bzw. die annotiert sind. An einem Blatt angekommen, bewegt er sich wieder zurück in Richtung der Wurzel, und vereinigt an jedem Knoten die Kontexte, die an diesem Knoten zusammenkommen auf dem Weg zur Wurzel. Lassen sich die Kontexte nicht vereinigen, z.B. wegen eines Typeclashes, wird mit einer Fehlermeldung abgebrochen.

[1] Auch wenn der Typchecker nicht mit der konkreten Syntax arbeitet, sondern mit der abstrakten, verwenden wir wegen der besseren Lesbarkeit die PrincipleWriter User Language Syntax in den Regeln.

Bevor wir uns das Typchecking mit Typinferenz für einige ausgewählte Regeln und deren Implementierungen anschauen werden ein paar Hilfsfunktionen beschrieben.

Die Funktion `Type`

Das Typchecking wird angestoßen durch die Funktion `Type`, die als Argument den Record, den der Parser gebaut hat, nimmt, und einen vollständig getypten Record zurückliefert. Betrachten wir uns den Anfang des Parsebaums, der mit den folgenden drei Regeln gebildet wird:

$$\begin{aligned} \text{S} &::= \text{DefType} * \text{DefPrinciple}+ \\ \text{DefType} &::= \texttt{deftype}\ \text{Constant Type} \\ \text{DefPrinciple} &::= \texttt{defprinciple}\ \text{Constant}\ \{\ \text{Dims Constraints}\ \} \end{aligned} \tag{4.1}$$

`Type` erstellt eine Liste mit den Constraints aus dem Parsebaum. Das sind die Formeln, die gecheckt werden sollen. Für jede Formel ruft `Type` die Hauptfunktion `DecideStrategy` auf, die das Typchecking übernimmt. Da durch das Abschneiden des oberen Teils des Baumes Informationen über vom Prinzipienschreiber definierte Typbenennungen verloren gehen, erstellt `Type` vorher einen Record mit dieser Information. Dafür wird die Funktion `CollectDefTypes` verwendet. Dieser Record wird an `DecideStrategy` übergeben, ebenso wie ein leerer Kontext.

Die Funktion `CollectDefTypes`

Die mit `deftype` vom Grammatikschreiber definierten Typbenennungen werden von der Funktion `CollectDefTypes` aus dem ihr übergebenen kompletten Parsebaum extrahiert, und in einem Record gespeichert, das Typnamen auf Typen abbildet.

Die Funktion `UnionContext`

Der Kontext des Typcheckers besteht aus einem Record, das Variable-Typ Paare enthält. Die Funktion `UnionContext` versucht zwei Kontexte zu vereinigen. Dabei wird überprüft, dass es keine zwei Variablen mit unterschiedlichen Typen gibt, was einen Typeclash auslösen würde.

4.3.2. Die Implementation des eigentlichen Typcheckings

Nachdem wir die Hilfsfunktionen vorgestellt haben, kommen wir jetzt zum Kern des Typinferenzmoduls, der von den Funktionen `DecideStrategy` und `TypeNonQuantifier` gebildet wird. `DecideStrategy` übernimmt das Typchecking und die Typinferenz für die Quantoren. Sie stellt auch den Startpunkt dar, da sie von `Type` aufgerufen wird. Ist der oberste Knoten im Baum kein Quantor, übergibt sie den Baum an `TypeNonQuantifier`, ansonsten versucht sie den Quantor zu typen, indem sie sich mit der Formel hinter dem Quantor rekursiv aufruft. `TypeNonQuantifier` ruft nicht sich selbst rekursiv wieder auf, sondern ruft `DecideStrategy` auf. Die Teilung in die zwei Funktionen hat den Grund die Lesbarkeit des Quellcodes zu steigern.

Beide Funktionen bekommen als Argumente den Record `ConstantATypeTreeRec` der die Abbildung von Typnamen auf Typen enthält (Siehe Funktion CollectDefTypes, Seite 34), die mit `deftype` vom Grammatikschreiber definiert wurden. Als zweites Argument einen Kontext und als drittes eine Formel, die getypt, bzw. überprüft werden soll.

Beide Funktionen haben als Rückgabewert einen Record der Form:

∃, getypt:

$$\frac{\Gamma \vdash \texttt{exists X :: type : Form} \quad \Gamma \cup \{X \mapsto \mathit{type}\} \vdash \texttt{Form}}{\Gamma \vdash \texttt{exists X :: type : Form}} (\mathit{type} \in \mathit{Dom})$$

∃, ungetypt, inferrierbar:

$$\frac{\Gamma \vdash \texttt{exists X : Form} \quad \Gamma \cup \{X \mapsto \mathit{type}\} \vdash \texttt{Form}}{\Gamma \vdash \texttt{exists X :: type : Form}} (\mathit{type} \in \mathit{Dom})$$

∃, ungetypt, nicht inferrierbar:

$$\frac{\Gamma \vdash \texttt{exists X : Form} \quad \Gamma \vdash \texttt{Form}}{\texttt{Error : CouldnotinferTypeofX}} \begin{array}{l}(\mathit{con} \mapsto T \notin \Gamma) \vee \\ (T = \_)\end{array}$$

Abbildung 4.6.: Die Regeln für das Typchecking des ∃ Quantors

```
o(operation:TypedForm
  con:NewContext)
```

wobei `TypedForm` die Eingabeformel ist, die um Typannotationen ergänzt wurde, und deren Typen überprüft wurden. `NewContext` ist der Kontext, nachdem die Eingabeformel bearbeitet wurde.

Die gesamte Implementierung der beiden Funktionen wollen wir uns nicht anschauen, sondern lediglich ein paar ausgesuchte Beispiele.

Die Funktion `DecideStrategy`

Bevor wir die Implementierung von `DecideStrategy` am Beispiel des ∃-Quantors erklären, werden wir uns die Regeln für den Quantor anschauen. Die Regeln und die Implementierung der anderen Quantoren funktioniert analog.

Für das Typchecking und die Typinferenz von Quantoren werden drei Fälle unterschieden. In Abbildung 4.6 sind die drei Regeln zu sehen. Die erste Regel behandelt den Fall, dass schon eine Typannotation an der Variablen `X`, über die quantifiziert wird, existiert. In diesem Fall wird versucht, die Typen der Formel `Form` zu checken, bzw. zu inferieren, wobei $X \mapsto \mathit{type}$ in den Context übernommen wird. Können die Typen von `Form` inferiert und überprüft werden, wird $X \mapsto \mathit{type}$ wieder aus dem Kontext entfernt, da diese Variable im Baum nur innerhalb von `Form` vorkommen darf. Sollte die Variable an einer anderen Stelle vorkommen, muss sie an einen anderen Quantor gebunden sein.

Die zweite Regel steht für den Fall, dass keine Typannotation an der Variablen `X` vorliegt, der Typ aber aus der Formel `Form` inferiert werden kann. Wenn dies der Fall ist enthält der Kontext, nachdem `Form` bearbeitet wurde, einen Eintrag für die Variable `X`. Dieser Eintrag wird aus dem Kontext entfernt, nachdem der Typ dann annotiert wurde.

Die dritte Regel beschreibt, dass keine Typannotation vorliegt, und der Typ von `X` auch nicht aus der Formel `Form` inferiert werden konnte. Dann bricht der Typchecker mit einer Fehlermeldung ab.

```
(  1) fun {DecideStrategy ConstantATypeTreeRec Context Form}
(  2)    FirstOp = Form.sem
(  3) in
(  4)    case FirstOp
(  5)    of exists(ConstantTree DomTree FormTree) then
(  6)       VarName = {Helpers.constantTree2A ConstantTree}
(  7)       Dom = {Helpers.tree2T DomTree ConstantATypeTreeRec}
(  8)       NewContext = {UnionContext Context o(VarName:Dom)}
(  9)       NewValues = {DecideStrategy
( 10)                    ConstantATypeTreeRec NewContext FormTree}
( 11)    in
( 12)       o(operation:exists(ConstantTree DomTree NewValues.operation)
( 13)         con:NewValues.con)
( 14)    [] exists(ConstantTree FormTree) then
( 15)       VarName = {Helpers.constantTree2A ConstantTree}
( 16)    in
( 17)       if {HasFeature Context VarName} then
( 18)          raise 'ERROR: Variable already defined'#VarName end
( 19)       else
( 20)          NewContext = {UnionContext Context o(VarName:undef)}
( 21)          NewValues = {DecideStrategy
( 22)                       ConstantATypeTreeRec NewContext FormTree}
( 23)       in
( 24)          case NewValues.con.VarName
( 25)          of undef then
( 26)             raise 'ERROR: Variable has no type'#VarName end
( 27)          else
( 28)             o(operation:exists(ConstantTree
( 29)                                NewValues.con.VarName
( 30)                                NewValues.operation)
( 31)               con:NewValues.con)
( 32)          end
( 33)       end
( 34)    else
( 35)       {TypeNonQuantifier ConstantATypeTreeRec Context Form}
( 36)    end
( 37) end
```

Abbildung 4.7.: Quellcode der Funktion `DecideStrategy`

$v_1 \rightarrow_d v_2$ v_1, v_2 sind Konstanten:

$$\frac{\Gamma \vdash \texttt{edge(v1 v2 d)}}{\Gamma \vdash \texttt{edge(v1 v2 d)}}$$

$v_1 \rightarrow_d v_2$ v_1, v_2, d sind Variablen:

$$\frac{\Gamma \vdash \texttt{edge(v1 v2 d)}}{\begin{array}{l}\Gamma \cup \{v1 \mapsto node, v2 \mapsto node, d \mapsto dim\} \\ \vdash \texttt{edge(v1 v2 d)}\end{array}} \quad \begin{array}{l}(v1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = node) \wedge \\ (v2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = node) \wedge \\ (d \mapsto T_3 \in \Gamma \Rightarrow T_3 = dim)\end{array}$$

Abbildung 4.8.: Regel für das Typchecking einer ungelabelten Kante

Schauen wir uns die Implementierung von `DecideStrategy` an, die in Abbildung 4.7 zu sehen ist. In Zeile 2 wird die zu untersuchende Formel aus der abstrakten Syntax extrahiert. Danach folgt eine Fallunterscheidung, wo durch Pattern-Matching entweder der $\exists$-Quantor mit Typannotation gefunden wird (Zeilen 5–13), oder ohne Typannotation (Zeilen 14–33) Die Fälle für die anderen Quantoren sind nicht aufgeführt. Wird kein Quantor gefunden, wird der Baum und der Kontext an `TypeNonQuantifier` weitergegeben (Zeile 35).

Im ersten Fall wird in Zeile 6 der Name der Variable aus dem Parsebaum extrahiert. Zeile 7 extrahiert den Typ aus dem Unterbaum `DomTree`. Danach wird das Paar aus dem Variablennamen und dem Typ in den Kontext aufgenommen (Zeile 8). In Zeile 9 folgt der rekursive Aufruf für die Formel nach dem Quantor `FormTree` mit dem neuen Kontext.

In den Zeilen 12 und 13 wird dann der Rückgaberecord gebaut, wobei der Unterbaum `FormTree` durch den bearbeiteten Baum aus dem rekursiven Aufruf ersetzt wird. Als Kontext wird der Kontext aus dem rekursiven Aufruf zurückgegeben, aus dem das in Zeile 8 hinzugefügte Paar wieder entfernt wurde.

Beim zweiten Fall wird auch erst der Variablenname bestimmt (Zeile 15). Danach wird überprüft ob die Variable schon im Kontext ist, und falls ja eine Fehlermeldung ausgegeben. Ist die Variable noch nicht im Kontext, wird sie diesem mit unbestimmtem Typ (`undef`) hinzugefügt (Zeile 20). In Zeile 21 folgt dann die Rekursion mit neuem Kontext für die Formel `FormTree`. In den Zeilen 24 und 25 wird dann geprüft, ob der rekursive Aufruf den Typ der Variablen inferieren konnte. Falls nicht, wird entsprechend der Regel 3 eine Fehlermeldung ausgegeben. Konnte der Typ ermittelt werden, wird der Ausgaberecord gebaut, indem dem Baum die Typannotation hinzugefügt wird (Zeile 29), und `FormTree` durch den bearbeiteten Baum aus dem rekursiven Aufruf ersetzt wird. Dies entspricht Regel 2.

Die Funktion `TypeNonQuantifier`

Um die Funktion `TypeNonQuantifier` kennen zu lernen, schauen wir uns zuerst an, wie die Kanten von ihr behandelt werden. Abbildung 4.8 zeigt zwei Regeln für ungelabelte Kanten. Die Typen für `v1`, `v2` und `d` müssen nicht ermittelt werden, da Kanten nur zwischen Knoten vorkommen dürfen. `d` ist immer vom Typ `dim`. Es muss nur noch überprüft werden, ob `v1`,`v2` und `d` Konstanten oder Variablen sind. Ist `v1`,`v2` oder `d` eine Variable, wird sie in den Kontext mit Typ `node`, bzw. `dim` übernommen, falls das zu keinem Typeclash führt. Der Parsebaum wird nicht verändert, da auch für die Evaluierung hier keine Typannotationen nötig sind.

Abbildung 4.9 zeigt den Teil der Funktion `TypeNonQuantifier`, der ungelabelte Kanten

```
(  1) [] edge(TermTree1 TermTree2 TermTree3) then
(  2)    ConstantAI1 = {Helpers.termTree2AI TermTree1}
(  3)    ConstantS1 = {V2S ConstantAI1}
(  4)    NewContext1 =
(  5)    if {Char.isAlpha ConstantS1.1} andthen
(  6)       {Char.isUpper ConstantS1.1} andthen
(  7)       {All ConstantS1
(  8)        fun {$ Ch}
(  9)           {Char.isAlpha Ch} orelse {Char.isDigit Ch}
( 10)        end}
( 11)    then
( 12)       {UnionContext Context o(ConstantAI1:node)}
( 13)    else
( 14)       Context
( 15)    end
( 16)    ConstantAI2 = {Helpers.termTree2AI TermTree2}
( 17)    ConstantS2 = {V2S ConstantAI2}
( 18)    NewContext2 =
( 19)    if {Char.isAlpha ConstantS2.1} andthen
( 20)       {Char.isUpper ConstantS2.1} andthen
( 21)       {All ConstantS2
( 22)        fun {$ Ch}
( 23)           {Char.isAlpha Ch} orelse {Char.isDigit Ch}
( 24)        end}
( 25)    then
( 26)       {UnionContext NewContext1 o(ConstantAI2:node)}
( 27)    else
( 28)       NewContext1
( 29)    end
( 30)    ConstantAI3 = {Helpers.termTree2AI TermTree3}
( 31)    ConstantS3 = {V2S ConstantAI3}
( 32)    NewContext3 =
( 33)    if {Char.isAlpha ConstantS3.1} andthen
( 34)       {Char.isUpper ConstantS3.1} andthen
( 35)       {All Constant S3
( 36)        fun {$ Ch}
( 37)           {Char.isAlpha Ch} orelse {Char.isDigit Ch}
( 38)        end}
( 39)    then
( 40)       {UnionContext NewContext2 o(ConstantAI3:dim)}
( 41)    else
( 42)       NewContext2
( 43)    end
( 44) in
( 45)    o(operation:edge(TermTree1 TermTree2 TermTree3)
( 46)      con:NewContext3)
```

Abbildung 4.9.: Quellcode des Typcheckings für eine ungelabelte Kante aus der Funktion `TypeNonQuantifier`

bearbeitet. Aus den drei Unterbäumen, die die Knoten und die Dimension repräsentieren, werden jeweils die Namen extrahiert, und dann geprüft ob es Variablen oder Konstanten sind. Falls es Konstanten sind werden sie in den Kontext aufgenommen. Der neue Kontext wird dann zusammen mit dem unveränderten Parsebaum zurückgegeben. Da in dem rekursiven Ablauf bei jeder Regel die veränderten Kontexte nach oben gereicht werden, können die Variablen der Quantoren annotiert werden, falls der Typ inferiert werden konnte.

4.3.3. Beispiel CSD-Prinzip

Zur Veranschaulichung werfen wir einen kurzen Blick auf das CSD-Prinzip. Ohne die Typinferenz müsste das Prinzip so geschrieben werden:

```
defprinciple "principle.csdPW" {
  dims {D}
  constraints {

    forall V::node: forall V1::node:
      edge(V V1 n D) =>
        forall V2::node:
          forall V3::node: dom(V2 V D) & edge(V2 V3 n D) => V3<V1
  }
}
```

Mit der Typinferenz müssen gar keine Typen mehr annotiert werden, was das Schreiben von Prinzipien stark erleichtert:

```
defprinciple "principle.csdPW" {
  dims {D}
  constraints {

    forall V: forall V1:
      edge(V V1 n D) =>
        forall V2:
          forall V3: dom(V2 V D) & edge(V2 V3 n D) => V3<V1
  }
}
```

4.4. Semantik

Die semantische Auswertung der abstrakten Syntax wird im PrincipleWriter vom Evaluatormodul übernommen. Auch wenn im Programmablauf des PrincipleWriters der Optimizer vor dem Evaluator kommt, werden wir uns zuerst den Evaluator anschauen, da die Funktionalität des Optimizer so einfacher zu verstehen ist.

Die Aufgabe des Evaluators ist es, aus der abstrakten Syntax mit Hilfe von semantischen Überführungsregeln den Mozart/Oz-Code der Prinzipien zu generieren. Der generierte Mozart-Code greift auf Funktionen aus dem Modul `PW.oz` zurück, um die logischen Operationen (Quantoren, Konnektive etc.) zu implementieren. Der Evaluator übergibt den Record, der die abstrakte Syntax enthält, an die Evaluierungsfunktion, die eine Liste von Tupeln der Form

```
Name#DV#Constraints
```

zurückgibt. Hierbei ist `Name` der Prinzipienname, `DV` sind die Dimensionsvariablen, über die das Prinzip abstrahiert, und `Constraints` der Mozart/Oz-Code für die logische Formel. Aus jedem dieser Tupel wird dann eine Prinzipiendefinition und ein Node-Constraint-Funktor generiert und als zwei Dateien gespeichert, die direkt in die XDK-Prinzipienbibliothek eingegliedert werden können.

4.4.1. Semantik der PWUL

Die Semantik der PrincipleWriter User Language wird durch eine Interpretationsfunktion definiert, die aus den Regeln aus Anhang D besteht, die die Formel rekusiv auswertet innerhalb der Umgebung (V, M, T). Hierbei ist:

- V ein Record mit drei Feldern:

$$V = \left\{ \begin{array}{lcl} a & : & \left\{ \begin{array}{c} \text{VarName:Oz-Atom-Name} \\ \vdots \end{array} \right\} \\ i & : & \left\{ \begin{array}{c} \text{VarName:Oz-Integer-Name} \\ \vdots \end{array} \right\} \\ n & : & \left\{ \begin{array}{c} \text{VarName:Oz-Node-Name} \\ \vdots \end{array} \right\} \end{array} \right\}$$

  eines für atomare Variablen, eines für numerische Variablen und eines für Knotenvariablen. Jedes Feld enthält einen Record aus Paaren, deren erster Wert der Variablenname ist und der zweite die Entsprechung in den zu generierenden Mozart/Oz-Constraints.

- M ein Modus. Bei der Interpretation muss eine Variable je nach Umgebung atomar, numerisch oder als Knoten interpretiert werden. Der Modus $M = a, i, n$ bestimmt, wie die Variable interpetiert wird.

- T eine Typannahme.

Die Funktionsweise der Regeln wollen wir anhand einiger Beispiele erklären.

Quantoren

Die Interpretation von Quantoren wollen wir am Beispiel des $\exists$-Quantors erklären. Die Interpretationsregel sieht so aus:

$[\![exists\ Constant :: Dom : Form]\!]^{V,\ M,\ T} =$

```
if Dom == node then
  NodeRecV = Constant#'NodeRec'
in
  '{PW.existsNodes NodeRecs
    fun {$ '#NodeRecV#'}'#
```

$[\![Form]\!]^{(\ V.n \cup \{Constant : NodeRecV\}\ ),M,T}$

```
#
    end}'
else
  AV = Constant#'A'
  IV = Constant#'I'
in
  '{PW.existsDom {T2Lat '#Dom#'}
    fun {$ '#AV#IV#'}'#
```

$[\![Form]\!]^{\left(\begin{array}{l} V.a \cup \{Constant : AV\} \\ V.i \cup \{Constant : IV\} \end{array}\right),M,T}$

```
#
    end}'
end
```

Es wird unterschieden, ob über einen Knoten oder eine andere Domäne quantifiziert wird. Da die Knotenmenge in jedem Node Constraint Funktor `NodeRecs` heißt, kann der Mozart/Oz Code direkt generiert werden. Die Funktionsbibliothek `PW` stellt einen ∃-Quantor über die Knoten zur Verfügung, der die Knotenmenge als Argument hat (immer `NodeRecs` in Node-Constraint-Funktoren), und die Funktion, die für einen Knoten wahr sein muss. Der Rumpf dieser Funktion ergibt sich aus der Interpretation der Formel `Form` innerhalb der bisherigen Umgebung, der die Variable `Constant` und ihre Entsprechung im generierten Mozart-Code (Constant#`'NodeRec'`) hinzugefügt wird.

Wird über eine andere Domäne als über Knoten quantifiziert, muss zuerst ein Name für die Domäne generiert werden. Der ∃-Quantor über nicht-Knoten-Mengen, der von der PW-Bibliothek bereitgestellt wird, bekommt den generierten Namen und eine Funktion als Argumente. Die Variable `Constant` hat in diesem Fall zwei Entsprechungen im generierten Mozart-Code: Constant#`'A'` (quantifiziertes Element als Oz-Atom) und Constant#`'I'` (als Oz-Integer). Beide werden u.U. von der `PW`-Funktionsbibliothek gebraucht.

Konstanten und Variablen

Bei der Interpretation von Konstanten und Variablen muss erst festgestellt werden, ob `Constant` eine Konstante oder eine Variable ist, da der kontextfreie Parser das nicht unterscheidet. Variablen beginnen immer mit einem Großbuchstaben und dürfen nur Buchstaben und Zahlen enthalten. Alles andere sind Konstanten.

Bei der Interpretation einer Variablen wird anhand der Umgebung festgestellt, ob es sich um eine atomare, eine numerische oder eine Knotenvariable handelt. Die Interpretation gibt dann die Mozart/Oz-Entsprechung des Variablennamens aus dem Record V zurück.

$[\![Constant]\!]^{V,\ M,\ T}$ =

```
if {Char.isAlpha {Atom.toString Constant}.1} andthen
   {Char.isUpper {Atom.toString Constant}.1} andthen
   {List.all {Atom.toString Constant}
      fun {$ Ch} {Char.isAlpha Ch} orelse {Char.isDigit Ch} end}} then
   if M == i then
      if T == node then
         V.n.Constant.index
      else
         V.i.Constant
      end
   elseif M == a then
      V.a.Constant
   else
      V.n.Constant
   end
else
   if M == i then
      '{{T2Lat '#T#'}.a2I '#Constant#'}'
   elseif M == a then
      Constant
   end
end
```

Abbildung 4.10.: Interpretation von Konstanten und Variablen

$[\![Expr_1\ union\ Expr_2\ =\ Expr_3]\!]^{V,\ M,\ T}$ =

```
'{PW.union '#
```
$[\![Expr_1]\!]^{V,i,T}$`#' '#`$[\![Expr_2]\!]^{V,i,T}$`#' '#`$[\![Expr_3]\!]^{V,i,T}$`#}'`

Abbildung 4.11.: Interpretation für die Vereinigung von Mengen

```
defprinciple "principle.disjPW" {
  dims {D}
  constraints {
    forall V::node: forall V1::node:
       forall L::label(D): forall L1::label(D):
          dom(V V1 L D) & dom(V V1 L1 D) => L=L1
  }
}
```

Abbildung 4.12.: Prinzip Disjunktheit von Teilbäumen mit unterschiedlichem Label in PWUL

Logische Operatoren und Mengenoperatoren

Bei logischen Operatoren und Mengenoperatoren werden zuerst die Unterbäume interpretiert, und diese Interpretationen dann als Argumente für Funktionsapplikationen aus der `PW`-Bibliothek eingefügt. Die `PW`-Bibliothek stellt für jeden logischen Operator und jede Mengenoperation eine Funktion zur Verfügung. Da Mengen in XDK immer Mengen von natürlichen Zahlen sind, muss die Auswertung der Unterbäume eines Mengenoperators im Modus $M = i$ interpretiert werden. Abbildung 4.11 zeigt die Regel für die Vereinigung von Mengen.

4.4.2. Beispiel: Disjunktheit von Unterbäumen mit unterschiedlichem Label

Als Beispiel wollen wir uns die Evaluierung eines Teils des Baumprinzips anschauen. In Abbildung 4.12 ist der Constraint zu sehen, der fordert, dass zwei Unterbäume eines Knotens mit gleichem Label disjunkt sein müssen.

Die Interpretationsfunktion extrahiert zuerst den Namen des Prinzips. Danach baut sie einen Record, der Dimensionsvariablen auf konkrete Dimensionen abbildet, auf die zur Laufzeit mit der Funktion `Principle.dVA2DIDA` zugegriffen werden kann. Im generierten Mozart/Oz-Code wird im Beispiel die Dimensionsvariable `D` immer durch `{Principle.dVA2DIDA \'D\'}` repräsentiert.

Die beiden ersten Quantoren, die über Knoten quantifizieren, werten wie vorher beschrieben aus:

```
{PW.forallNodes NodeRecs
 fun {$ VNodeRec}
    {PW.forallNodes NodeRecs
     fun {$ V1NodeRec}
```

$[\![$`forall L::label(D): forall L1::label(D):`
`dom(V V1 L D) & dom(V V1 L1 D) => L=L1`$]\!]^{V1,M,T}$

```
     end}
 end}
```

Der nächste Quantor quantifiziert nicht über einen Knoten, sondern über die Label der Dimension `D`. Die konkreten Labels der Dimension können erst zur Laufzeit des Prinzips ermittelt werden. Dazu dient die Funktion `PW.t2Lat`, die mit dem Typ `label('D')` als Argument aufgerufen wird. Hier wird deutlich, wieso die Typannotationen notwendig sind. Ohne diese wäre es z.B. nicht klar, ob über Knoten oder einer anderen endlichen Domäne quantifiziert wird.

Die Evaluierung nach allen Quantoren sieht dann so aus:

```
{PW.forallNodes NodeRecs
 fun {$ VNodeRec}
    {PW.forallNodes NodeRecs
     fun {$ V1NodeRec}
        {PW.forallDom {PW.t2Lat label('D')}
         fun {$ LA LI}
            {PW.forallDom {PW.t2Lat label('D')}
             fun {$ L1A L1I}
```

$[\![\texttt{dom(V V1 L D) \& dom(V V1 L1 D) => L=L1}]\!]^{V1,M,T}$

```
             end}
         end}
     end}
 end}
```

Bei der Evaluierung der Implikation werden erst beide Unterbäume ausgewertet. Die Umgebung wird nicht verändert. Bei der Evaluierung der Dominanzen werden die zwei ersten Unterbäume mit dem Modus $M = n$ evaluiert. Der dritte Unterbaum mit $M = a$ und Typ $T = \mathit{label}(\mathit{TermV4})$, wobei $\mathit{TermV4}$ die Evaluierung des vierten Baums mit Modus $M = a$ und Typ $T = \mathit{dim}$ ist. Die Konjunktion funktioniert wie die Implikation.

Bei der Evaluation von `V` und `V1` wird zuerst geprüft, ob es sich um Variablen oder Konstanten handelt. Sie beginnen mit einem Großbuchstaben, also sind es Variablen. Da sie im Modus $M = n$ evaluiert werden, können die Mozart/Oz-Entsprechungen der Variablen aus dem Unterrecord n des Records V ausgelesen werden. In unserem Fall sind das `VNodeRec` und `V1NodeRec`. Die Variablen `L` und `L1` evaluieren zu `LA` und `L1A`, da sie für die Funktion `PW.ldom` im Modus $M = a$ ausgewertet werden müssen. Die Variable `D` wird im Modus $M = a$ ausgewertet. Ihre Mozart/Oz-Entsprechung kann aus dem Unterrecord a von V ausgelesen werden, die in unserem Fall `{Principle.dVA2DIDA \'D\'}` ist.

Da die Gleichheit von Termen, ausgedrückt durch die Funktion `PW.eq`, Gleichheit von natürlichen Zahlen testet, müssen `L` und `L1` im Modus $M = i$ evaluiert werden, und bekommen die Mozart/Oz-Variablennamen `LI` und `L1I`, die bei der Interpretation der beiden letzten Quantoren der Umgebung zugefügt wurden.

Der komplette Constraint sieht dann so aus:

```
{PW.forallNodes NodeRecs
 fun {$ VNodeRec}
    {PW.forallNodes NodeRecs
     fun {$ V1NodeRec}
        {PW.forallDom {PW.t2Lat label('D')}
         fun {$ LA LI}
            {PW.forallDom {PW.t2Lat label('D')}
             fun {$ L1A L1I}
                {PW.impl
                 {PW.conj
                  {PW.ldom
                   VNodeRec V1NodeRec LA {Principle.dVA2DIDA 'D'}}
```

```
                fun {$}
                   {PW.ldom
                    VNoderec V1NodeRec L1A {Principle.dVA2DIDA 'D'}}
                end}
               fun {$}
                  {PW.eq
                   LI L1I
                  }
               end}
            end}
         end}
      end}
   end}
```

Die Evaluation gibt dann den Namen des Prinzips, die Dimensionsvariablen und den generierten Mozart/Oz-Constraint zurück, und baut aus diesen Informationen die Prinzipiendefinition und den Node-Constraint-Funktor. Die Prinzipiendefinition sieht so aus:

```
defprinciple "principle.disjPW" {
  dims {D}
  constraints {"DisjPW": 140}}
```

Und der Node-Constraint-Funktor sieht so aus:

```
functor
import
   PW at 'PW.ozf'
export
   Constraint
define
   proc {Constraint NodeRecs G Principle FD FS Select}
      [Constraint]
   end
end
```

4.4.3. Die Funktionsbibliothek PW

Die meisten Funktionen der `PW`-Funktionsbibliothek geben die Argumente direkt an die Funktionen der `FD`- und `FS`-Mozart/Oz Bibliotheken weiter, die Constraints über endliche Mengen und endliche Domänen bereitstellen. Beispielhaft wollen wir die Implementierung der Quantoren und der ungelabelten Kanten und Pfade vorstellen.

Die Quantoren der PW-Bibliothek

Die Quantoren, die über Knoten quantifizieren, deklarieren eine unspezifizierte Mengenvariable M. Dann wird über die Knoten, über die quantifiziert werden soll, iteriert. Bei jedem Schleifendurchgang wird der aktuelle Knoten dann zur Menge M hinzugefügt, wenn die Formel im Rumpf des Quantors für den aktuellen Knoten zu 1 auswertet. Damit die $\forall$-Quantifizierung wahr wird, muss nach der Iteration die Menge M gleichmächtig der Knotenmenge sein, über die quantifiziert wurde. Die $\exists$-Quantifizierung wird wahr, wenn M mindestens ein Element, und die $\exists!$-Quantifizierug, wenn M genau ein Element enthält.

Die Quantoren über die anderen Domänen funktionieren ähnlich wie die Knotenquantoren, mit dem Unterschied dass über die Konstanten der Domäne, über die quantifiziert wird, iteriert wird.

Ungelabelte Kanten und Pfade der PW-Bibliothek

Um zu prüfen, ob es eine Kante von Knoten v zu Knoten v_1 gibt, wird getestet, ob v_1 in der Tochtermenge `daughters` von v ist. Ein Pfad von v nach v_1 mit mindestens einer Kante existiert, wenn v_1 in der `down`-Menge von v ist, ein Pfad mit beliebiger Menge wird mit der `eqdown` überprüft etc.

4.5. Optimierung

Nachdem wir nun wissen wie bei der Evaluierung aus der Mozart/Oz-Darstellung der abstrakten Syntax die Mozart/Oz Constraints generiert werden, widmen wir uns jetzt der Optimierung, die noch vor der Evaluierung stattfindet.

Wir wissen jetzt, dass jeder Quantor in der PrincipleWriter User Language in eine `for`-Schleife umgesetzt wird, die über die Elemente des Typs der Variable iteriert. Die Ausführungzeit steigt damit linear zur Mächtigkeit dieses Typs. Mit jeder Schachtelung von Quantoren steigt die Anzahl von letztendlich gestarteten Propagierern, was sich leicht negativ auf die Laufzeit auswirken kann.

Experten im Schreiben von Prinzipien als Mozart/Oz-Constraints können die verschiedenen Knotenmengen ausnutzen, die bei der Modellierung des Multigraphen gebildet werden. Sie können damit die Anzahl von Quantoren, und deren Schachtelung reduzieren. Aus einigen dieser Techniken konnte ich Muster erkennen, die regelmäßig vorkommen.

Der Optimierer automatisiert einige dieser Techniken. Er durchwandert den durch die Typinferenz vollständig getypten Parsebaum und versucht durch Pattern-Matching Muster zu erkennen, für die Optimierungen bekannt sind. Wird ein Muster gefunden, ersetzt der Optimierer den Teilbaum, der das Muster im Parsebaum repräsentiert, durch einen neuen Teilbaum, der die Optimierung repräsentiert. In der Evaluation wird dann die entsprechende optimierte Funktion aus der PW-Bibliothek eingesetzt, die das Problem so löst, wie Experten es tun würden.

Der Optimierer findet derzeit nicht alle diese Muster, die er optimieren kann, da manche Muster verschiedene äquivalente Darstellungen in den logischen Formeln haben, z.B. gilt $\forall\, x : F \;\equiv\; \neg\, \exists\, x :\; \neg\, F$. Um dieses Problem zu beheben, müssten die Formeln vor dem Patternmatching in eine Art Normalform überführt werden, was bisher noch nicht gemacht wird, aber einen interessanten Ansatz für weitere Forschungen bietet, die generierten Prinzipien noch besser zu optimieren.

Wie die Verwendung der Knotenmengen die Anzahl der Quantifizierungen und die verschachtelungstiefe der Quantifizierungen verringern kann, schauen wir uns an Beispielen an.

4.5.1. ZeroOrOneMother

Die *ZeorOrOneMother*-Optimierung kann z.B. einen Constraint des Baumprinzips erheblich verbessern. Zur Erinnerung, der Constraint wird in First-Order Logik so beschrieben:

$$\forall v : (\neg \exists v' : v' \rightarrow_d v) \vee (\exists^1 v' \rightarrow_d v)$$

Bei der unoptimierten Evaluation würde ein Mozart/Oz-Constraint generiert, der verschachtelte Quantoren enthält, die Verschachtelungstiefe wäre 1.

Die Idee der Optimierung folgt direkt aus den beiden Komponenten der Disjunktion. Die linke Seite der Disjunktion prüft, ob der Knoten v aus der äußeren Quantifizierung keine Mutter hat. Da bei der Modellierung des Multigraphen für jeden Knoten eine Menge `mothers` gebildet wird, die alle Mütter des Knotens enthält, muss nicht für jeden Knoten geprüft werden, ob er eine Mutter von v ist. Es reicht zu überprüfen, ob die Muttermenge von v leer ist. Die rechte Seite prüft, ob der Knoten v genau eine Mutter hat. Anstatt dafür über alle Knoten zu iterieren, reicht es zu prüfen, ob die Muttermenge von v genau ein Element enhält.

Ob die Muttermenge leer ist oder genau ein Element enthält kann über die Kardinalität der Menge festgestellt werden. Der erste Optimierungsschritt sieht dann so aus:

$$\forall v : \underbrace{(\neg \exists v' : v' \rightarrow_d v)}_{|mothers(v)|=0} \vee \underbrace{(\exists^1 v' \rightarrow_d v)}_{|mothers(v)|=1}$$

Der nächste Optimierungsschritt eliminiert die Disjunktion, indem nur noch geprüft wird, ob die Kardinalität der Muttermenge ≤ 1 ist. Die vollständig optimierte Formel sieht jetzt so aus:

$$\forall v : |mothers(v)| \leq 1$$

Sie enthält nur noch eine Quantifizierung und es müssen weniger Propagierer gestartet werden als im ersten Optimierungsschritt.

Der Optimierer arbeitet durch Patternmatching auf der Mozart/Oz Darstellung der abstrakten Syntax. Für die Beispiel-Optimierung sucht er ein Pattern der Form:

```
disj(neg(exists(X _ edge(Y Z D)))
     existsone(X _ edge(Y Z D)))
```

mit $Z \neq Y$. Ein gefundener Unterbaum dieser Form wird durch einen neuen Baum der Form:

```
zeroOrOneDaughters(Z D)
```

ersetzt, und damit auch die abstrakte Syntax erweitert. Bei der Evaluierung wird die entsprechende optimierte Funktion aus der `PW`-Bibliothek benutzt. Für die Beispieloptimierung sieht sie so aus:

```
fun {ZeroOrOneMothers NodeRec DIDA}
   {FD.reified.lesseq
    {FS.card NodeRec.DIDA.model.mothers} 1}
end
```

4.5.2. Disjunktheit von Unterbäumen mit unterschiedlichem Label

Den bei der Evaluierung vorgestellten Constraint wollen wir hier nochmal aufgreifen. Der Constraint verlangt, dass die Unterbäume eines Knotens mit unterschiedlichem Label disjunkt sein müssen. Aus der Formel des Constraints:

$$\forall v : \forall v' : \forall l : \forall l' : v \xrightarrow{l}_d \rightarrow_d^* v' \wedge v \xrightarrow{l'}_d \rightarrow_d^* v' \Rightarrow l = l'$$

ist zu erkennen, dass zweimal verschachtelt quantifiziert wird. Einmal über die Knoten und einmal über die Kantenbeschriftungen.

Wenn wir uns überlegen, was die geforderte Disjunktheit bedeutet, wird die Optimierung leicht klar. Der Constraint verlangt, dass es in jedem Knoten keine zwei ausgehenden Kanten mit der gleichen Beschriftung geben darf. Das bedeutet, dass es keine verschiedenen Pfade von einem Knoten v zu einem Knoten v' aus der Menge *down* der Knoten unterhalb von v geben darf, deren erste Kante das gleiche Label hat.

Die Menge *downL* enthält Paare aus einem Label und einer Untermenge der Knoten aus *down*. Für jede ausgehende Kante l des Knotens v wird solch ein Paar gebildet. Die Untermengen enthalten die Knoten, die von v über die Kante l erreicht werden können. Wenn die Unterbäume mit unterschiedlichem Label disjunkt sind, bilden die Untermengen aus *downL* eine Partition von *down*. Die Optimierung sieht jetzt so aus, dass nur noch für jeden Knoten geprüft wird, ob *downL* eine Partition von *down* bildet:

$$\forall v : downL(v) \text{ ist Partition von } down(v)$$

Es wird also nur noch einmal über die Knoten quantifiziert. Die Überprüfung, ob $downL(v)$ eine Partition von $donw(v)$ ist, geschieht dann mit Hilfe eines Propagierers aus der Mozart/Oz-Bibliothek für endliche Mengen.

Der Optimierer ersetzt also ein Pattern der Form:

```
forall(V1
       D
       forall(L1
              _
              forall(L2
                     _
                     impl(conj(ldom(V V1 L1 D))
                               ldom(V V1 L2 D))
                     eq(L1 L2))))
```

durch einen neuen Baum der Form:

```
disjointSubtreesL(V D)
```

Die Interpretation dieser Optimierung sieht dann so aus:

```
fun {DisjointSubtreesL NodeRec DIDA}
   {ReifiedPartition
    NodeRec.DIDA.model.downL
    NodeRec.DIDA.model.down}
end
```

wobei `ReifiedPartition` eine reifizierte Version des Constraints `FS.partition` aus der Oz-Finite-Set-Constraints-Bibliothek ist.

4.5.3. Informelle Analyse der Optimierung

Der Optimierer kann noch deutlich mehr Pattern erkennen als wir hier vorgestellt haben. Die Optimierungen machen exzessiv Gebrauch der Mengen aus den Modellen. Dabei eliminieren sie Quantifizierungen und verringern die Anzahl der gestarteten Propagierern.

Erste Vergleiche zwischen optimierten Prinzipien, die vom PrincipleWriter generiert wurden, und unoptimierten generierten Prinzipien, sowie schon vorhandenen Prinzipien, die von

Experten von Hand optimiert wurden, haben gezeigt, dass die automatisch optimierten Prinzipien deutlich effizienter sind als die unoptimierten. Die automatisch optimierten Prinzipien sind zwar nicht ganz so schnell wie die von Hand optimierten, sie sind allerdings jetzt schon effizient genug für den praktischen Einsatz, gerade zum Experimentieren mit neuen Prinzipien.

Verglichen wurden zwei Grammatiken aus der XDK-Distribution. Die Nutshell-Einführungs-Grammatik (nut1.ul) aus (Debusmann 2006), die eine vereinfachte englische Grammatik ist mit einer Syntax-Semantik-Schnittstelle. Sie hat drei Dimensionen, 12 Prinzipien und ein kleines Lexikon. Die zweite Grammatik war die große Grammatik aus der Dissertation von Ralph Debusmann, die ebenfalls eine Englische Grammatik ist, mit 12 Dimensionen, 53 Prinzipien und einem großen Lexikon. Für beide Grammatiken wurden zwei repräsentative Sätze getestet. Einmal mit den vom PrincipleWriter optimierten Prinzipien, und einmal mit den von Hand optimierten. Die Ergebnisse waren folgende:

	Nut1.ul	Diss.ul
Handoptimiert	94	1.46
	109	0.655
PWUL-nicht-Optimiert	375	17
	468	5.65
PWUL-Optimiert	172	10.21
	234	2.57

Diese ersten Tests haben einerseits gezeigt, dass die Prinzipien, die durch den PrincipleWriter generiert wurden, alle Prinzipien abdecken, die die Diss-Grammatik beinhaltet. Das ist ein guter Test, da die Diss-Grammatik fast eine Obermenge aller bisher benutzten Prinzipien benutzt. Dass die Optimierungen greifen kann man anhand der Parsezeiten sehen. Bei beiden Grammatiken sorgen sie für eine Effizienzsteigerung von mehr als 100%. und kommen der Effizienz der von Hand optimierten Prinzipien recht nahe. Bei der grossen, aufwändigen diss-Grammatik sind die optimierten Prinzipien immer noch ca. doppelt so schnell wie die nichtoptimierten, aber ca. 5mal langsamer als die von Hand optimierten.

Es bleibt also Spielraum für weitere Forschungen, wie die Prinzipien noch besser automatisch optimiert werden können. In der jetzigen Version sind die Optimierungen aber ausreichend, um viele neue Prinzipien aufzuschreiben, ohne sich um die Optimierung Gedanken machen zu müssen, und damit neue Ideen für dependenzbasierte Grammatikformalismen zu explorieren.

5. Zusammenfassung und Zukünftige Arbeiten

5.1. Zusammenfassung

In dieser Arbeit habe ich in den ersten Kapiteln den Extensible Dependency Grammar Formalismus (Debusmann, Duchier, Koller, Kuhlmann, Smolka & Thater 2004) erklärt, und wie Grammatiken damit formal entwickelt werden können. Grammatiken bestehen in XDG aus drei Teilen: Einem Multigraphtyp, einem Lexikon und einer Menge von Prinzipien, die die Wohlgeformtheitsbedingungen der Multigraph-Modelle der Grammatik regeln. In der neuesten Formalisierung von XDG (Debusmann 2007*b*) werden die Prinzipien in Prädikatenlogik erster Stufe ausgedrückt. Als Beispiel habe ich die CSD-Grammatik vorgestellt, die Gebrauch von einem speziell für diese Grammatik entwickelten Prinzip macht (CSD-Prinzip). Als nächstes habe ich das XDG Development Kit (XDK) (Debusmann, Duchier & Niehren 2004) vorgestellt und beschrieben, wie damit XDG-Grammatiken implementiert werden können. Mit Hilfe der XDK Description Language, die Bestandteil des XDK ist, können der Multigraphtyp und das Lexikon einer Grammatik analog zur Formalisierung in XDG aufgeschrieben werden. Die dann noch fehlende Komponenten einer Grammatik, die Prinzipien, können nicht mit der XDK Description Language beschrieben werden, sondern nur aus einer Bibliothek von vordefinierten Prinzipien ausgewählt, oder mühsam von Hand implementiert werden.

Ich habe gezeigt, wie die im ersten Kapitel vorgestellte CSD-Grammatik in XDK implementiert wird. Dabei wurde deutlich, wie schwierig es ist, das XDK um weitere Prinzipien zu erweitern, da diese als Mozart/Oz Constraints implementiert werden mussten, was nur von Experten der Mozart/Oz Constraintprogrammierung gemacht werden kann, da auch auf die Effizienz der Constraints zu achten ist. Das bedeutet, die Constraints mussten von Hand optimiert werden. Die große Lücke zwischen der Formalisierung und der Implementierung wurde deutlich. Die Lücke war für typische Grammatikschreiber, wie z.B. Linguisten, die keine Experten in der Mozart/Oz Constraint Programmierung sind, praktisch unüberbrückbar groß, so dass die Erweiterbarkeit des XDK in Punkto Prinzipien praktisch nicht gegeben war.

Um diese Lücke zu schliesen habe ich den PrincipleWriter entwickelt. Der PrincipleWriter ist ein Werkzeug mit dem Prinzipien analog zur XDG-Formalisierung als prädikatenlogische Formeln erster Stufe aufgeschrieben werden können. Die Mozart/Oz-Constraints werden dann vom PrincipleWriter erzeugt, so dass neue Prinzipien leicht geschrieben werden können. Die generierten Mozart/Oz-Constraints können dann einfach ins XDK eingegliedert werden. Damit ist die Erweiterbarkeit von XDG jetzt auch in der Implementierung, dem XDK, gegeben.

Um die Prinzipien als logische Formeln aufschreiben zu können, habe ich eine Syntax entwickelt, die PrincipleWriter User Language (PWUL), mit der Prinzipien analog zur Formalisierung aufgeschrieben werden können. Um das Aufschreiben weiter zu vereinfachen hat der PrincipleWriter einen Typchecker mit Typinferenz, so dass Typfehler frühzeitig aufgedeckt werden können und Typannotationen fast nicht mehr nötig sind, was die Prinzipien in PWUL noch dichter an die XDG Formalisierung bringt.

Wie die Semantik der PrincipleWriter User Language definiert ist, und wie mit Hilfe der Semantik die in Logik geschriebenen Prinzipien in Mozart/Oz-Constraints übersetzt werden, habe ich dann gezeigt. Da die eigentliche Evaluierung die Operatoren der Logik fast direkt in Mozart/Oz-Constraints übersetzt, wurde deutlich, dass die generierten Constraints noch optimiert werden müssen, da z.B. durch verschachtelte Quantoren die generierten Prinzipien kaum effizient genug sind für den praktischen Einsatz.

Ob es möglich ist, die Prinzipien automatisch so zu optimieren, dass sie effizient genug für den Praxiseinsatz sind, war die Herausforderung. In dem Kapitel über die Optimierung habe ich gezeigt, wie diese funktioniert, und dass es möglich ist, die Prinzipien ausreichend gut zu optimieren. Die Optimierung funktioniert durch Pattern-Matching auf der Mozart/Oz-Darstellung der abstrakten Syntax und geschieht damit noch vor der Evaluierung. Die Pattern, auf die gematcht wird, haben versierte Grammatikschreiber bisher auch schon genutzt, um die Prinzipien zu optimieren. Von deren Erfahrungen konnte ich profitieren.

Durch den PrincipleWriter habe ich die Lücke zwischen der XDG Formalisierung und der Implementierung (XDK) geschlossen, so dass jetzt jeder einfach neue Prinzipien entwickeln kann und damit neue Grammatken schreiben kann, ohne Kenntnisse der Mozart/Oz Constraintprogrammierung zu haben, und ohne sich Gedanken um die Optimierung zu machen. Aber auch die Experten profitieren vom PrincipleWriter, da für sie das Implementieren von Prinzipien eine sehr zeitaufwändige Arbeit war.

5.2. Zukünftige Arbeiten

Auch wenn die bisher in Praxistests generierten Prinzipien effizient genug sind für den Einsatz in der Praxis, bleibt die Herausforderung, die automatische Optimierung so zu verbessern, dass die generierten Prinzipien so gut wie die Prinzipien der Mozart/Oz Experten werden, oder vielleicht sogar noch besser. Dieses Feld bietet viel Raum für weitere Forschungen. Auch die Frage ob es eine optimale Optimierung gibt, und falls ja, ob diese dann automatisiert werden kann ist eine interessante Frage.

Bisher ist die Optimierung noch sehr rudimentär durch einfaches Pattern-Matching realisiert. Die Muster entspringen dem Erfahrungsschatz von Experten. Um die Optimierung weiter zu verbessern, könnten weitere Muster gesucht werden und damit das Pattern-Matching erweitert werden.

Ein Problem beim Optimieren durch reines Pattern-Matching habe ich im Optimierungskapitel bereits erwähnt. Da verschiedene Formeln äquivalent sein können, werden manche Optimierungen nicht vorgenommen, da kein Muster passt. Beim Optimieren von Hand kann man solche Äquivalenzen erkennen und die Optimierung vornehmen. Um dieses Problem beim automatisierten Optimieren in den Griff zu bekommen, müsste untersucht werden, ob eine Art Normalform für die logischen Formeln gefunden werden kann, in die die Formeln automatisch überführt werden können und die Äquivalenzen damit aufgelöst werden würden.

Eine andere Idee die Optimierung zu verbessern wird motiviert durch die Feststellung, dass die ursprüngliche Implementierung des Barierenprinzips exzessiven Gebrauch von Selection-Union-Constraints macht. Damit ist das Prinzip deutlich schneller, als die automatisch generierte Variante des PrincipleWriters.

Die eigentliche Idee für die weitere Optimierung ist folgende: Praktisch jedes Prinzip verwendet Pfade in irgendeiner Weise. Alle Pfade starten mit einem Knoten, den wir den Ankerknoten des Prinzips nennen, und gehen von diesem Knoten weiter. Dadurch sollte es nicht nötig sein

über mehr Knoten als über den Ankerknoten zu quantifizieren. Dies würde bedeuten, dass alle Prinzipien so implementiert werden könnten, dass ihre Laufzeit linear zur Knotenmenge wäre.

Ein kleines Beispiel soll die Idee verdeutlichen. Beim Climbing-Prinzip:

```
forall V::node: forall V1::node:
   dom(V V1 D1) => dom(V V1 D2)
```

ist der Ankerknoten `V`. Das Prinzip besagt, dass wenn `V1` von `V` dominiert wird auf der Dimension `D1`, dann muss `V1` auch auf der Dimension `D2` von `V` dominiert werden. Das bedeutet aber nichts anders, als dass die Knoten, die in der Dimension `D1` von `V` dominiert werden, eine Teilmenge der Knoten sein müssen, die in der Dimension `D2` von `V` dominiert werden. Damit kann die zweite Quantifizierung entfernt werden und Gebrauch des Teilmengen-Constraints gemacht werden:

```
forall V::node:
   down_D1(V) subseteq down_D2(V)
```

Damit hat man eine lineare Implementierung des Prinzips, da nur noch über einen Knoten quantifiziert wird, während die ursprüngliche Implementierung quadratisch war.

Alle Prinzipien der Prinzipienbibliothek könnten so umgeschrieben werden, dass sie nur noch über einen Knoten quantifizieren, und somit linear zur Knotenmenge sind. Ob diese Optimierung automatisiert werden kann, ist eine interessante Frage und Herausforderung.

A. XDK Description Language

A.1. Interpretation der Typen

Gegeben eine Menge `A` aus Atomen und eine Menge `D` von Dimensionen, Typen werden wie folgt interpretiert:

- $\{\mathtt{a_1} \ldots \mathtt{a_n}\}$ als die Menge $\{\mathtt{a_1}, \ldots, \mathtt{a_n}\} \uplus \{\top, \bot\}$, wobei $\top$ und $\bot$ hinzugefügt werden in der Funktion als top und bottom des lattice des entsprechenden Types.
- `string` als die Menge aller Atome plus $\top$ und $\bot$: $\mathtt{A} \uplus \{\top, \bot\}$, z.B., die Interpretation von Strings kann unendlich sein (wenn `A` unendlich ist), im Gegensatz zu der interpretation einer endlichen Domäne.
- `int` als die Menge aller natürlichen Zahlen plus $\top$ und $\bot$
- `list(T)` für alle $n > 0$ als die Menge von allen n-Tupel deren Projektionen Elemente der Interpretation von `T`, plus $\top$ und $\bot$ sind.
- `set(T)` und `iset(T)` als die Potenzmenge der Interpretation von `T`
- `card` als die Potenzmenge der Menge der natürlichen Zahlen.
- $\mathtt{tuple}(\mathtt{T_1} \ldots \mathtt{T_n})$ als die Menge von allen `n`-Tupeln deren `i`te Projektion ein Element der Interpretation von $\mathtt{T_i}$ (für $\mathtt{1} \leq \mathtt{i} \leq \mathtt{n}$)
- $\{\mathtt{a_1} : \mathtt{T_1} \ldots \mathtt{a_n} : \mathtt{T_n}\}$ als die Menge aller Funktionen f mit:
  1. $Dom\ f = \{\mathtt{a_1}, \ldots, \mathtt{a_n}\}$
  2. für alle $1 \leq \mathtt{i} \leq \mathtt{n}$, $f\ \mathtt{a_i}$ ist ein element der Interpretation von $\mathtt{T_i}$
- Gegeben eine Dimensionsvarable `D` an Dimension d, `label(D)` wird interpretiert als der Typ von Kantenbeschriftungen von d.
- Gegeben eine Typvariable `X` vom Typ `T`,`tv(X)` wird interpretiert als `T`.

wobei die Typen `label(D)` und `tv(X)` nur bei Prinzipientypdefinitionen und nicht bei Metagrammartypdefinitionen verwendet werden dürfen.

A.2. Constraint Parser

- `mothers`: die Menge der Mutterknoten von v.
- `daughters`: die Menge der Töchter von v.
- `up`: die Menge der Knoten über v.

- `down`: die Menge der Knoten unter v.
- `eq`: die Menge die nur den eigenen Knoten v enthält.
- `equp`: die Menge der Knoten über oder gleich v.
- `eqdown`: die Menge der Knoten unterhalb oder gleich v.
- `labels`: die Menge der Kantenlabel der eingehenden Kanten von v.
- `mothersL`: die Menge der Mütter von v nach Kantenlabel sortiert.
- `daughtersL`: die Menge der Töchter von v nach Kantenlabel sortiert.
- `upL`: die Menge der Knoten über v sortiert nach den labeln der eingehenden Kanten.
- `downL` die Menge der Knoten unter v sortiert nach den Kantenlabeln der ausgehenden Kanten.

B. Kontextfreie Grammatik der PWUL

$$\text{S} \quad ::= \quad \text{DefType} * \text{DefPrinciple+} \tag{B.1}$$

$$\text{DefType} \quad ::= \quad \texttt{deftype}\ \text{Constant Type} \quad \text{(type definition)} \tag{B.2}$$

$$\text{DefPrinciple} \quad ::= \quad \texttt{defprinciple}\ \text{Constant}\ \{\ \text{Dims Constraints}\ \} \quad \text{(principle definition)} \tag{B.3}$$

$$\text{Dims} \quad ::= \quad \texttt{dims}\ \{\ \text{Constant} +\ \} \quad \text{(dimensions)} \tag{B.4}$$

$$\text{Constraints} \quad ::= \quad \texttt{constraints}\ \{\ \text{Form} +\ \} \quad \text{(constraints)} \tag{B.5}$$

$$\begin{array}{rcll} \text{Dom} & ::= & \{\text{Constant} +\} & \text{(domain)} \\ & | & \texttt{label}(\text{Constant}) & \text{(edge labels)} \\ & | & \texttt{node} & \text{(nodes)} \\ & | & \texttt{dim} & \text{(dimensions)} \\ & | & \texttt{attr} & \text{(attributes)} \\ & | & \texttt{word} & \text{(words)} \\ & | & \text{Constant} & \text{(type reference)} \end{array} \tag{B.6}$$

$$\begin{array}{rcll} \text{Type} & ::= & \text{Dom} & \text{(domain)} \\ & | & \texttt{set}(\text{Dom}) & \text{(set)} \\ & | & \texttt{tuple}(\text{Dom} +\ ) & \text{(tuple)} \\ & | & \texttt{set}(\texttt{tuple}(\text{Dom} +\ )) & \text{(set of tuples)} \\ & | & (\text{Type}) & \text{(bracketing)} \end{array} \tag{B.7}$$

$$\begin{array}{rcll} \text{Term} & ::= & \text{Constant} & \text{(constant)} \\ & | & \text{Integer} & \text{(integer)} \end{array} \tag{B.8}$$

$$\text{Tuple} \quad ::= \quad [\text{Term} +\ ] \quad \text{(tuple)} \tag{B.9}$$

$$\begin{array}{rcll} \text{Expr} & ::= & \text{Term} & \text{(term)} \\ & | & \{\text{Term} +\} & \text{(set)} \\ & | & \text{Tuple} & \text{(tuple)} \\ & | & \{\text{Tuple} +\} & \text{(set of tuples)} \\ & | & \text{Expr} :: \text{Type} & \text{(type annotation)} \\ & | & \text{Term} \ .\ \text{Constant}_1\ .\ \texttt{entry}\ .\ \text{Constant}_2 & \text{(lexical attributes access)} \\ & | & \text{Term} \ .\ \text{Constant}_1\ .\ \texttt{attrs}\ .\ \text{Constant}_2 & \text{(non-lexical attributes access)} \\ & | & (\text{Expr}) & \text{(bracketing)} \end{array} \tag{B.10}$$

Form	::=	~ Form	(negation)
	|	Form_1 & Form_2	(conjunction)
	|	Form_1 | Form_2	(disjunction)
	|	Form_1 => Form_2	(implication)
	|	Form_1 <=> Form_2	(equivalence)
	|	`exists` Constant :: Dom : Form	(existential quantification)
	|	`existsone` Constant :: Dom : Form	(one-existential quantification)
	|	`forall` Constant :: Dom : Form	(universal quantification)
	|	Expr_1 = Expr_2	(equality)
	|	Expr_1 `in` Expr_2	(element)
	|	Expr_1 `notin` Expr_2	(not element)
	|	Expr_1 `subseteq` Expr_2	(subset)
	|	Expr_1 `disjoint` Expr_2	(disjointness)
	|	Expr_1 `intersect` Expr_2 = Expr_3	(intersection)
	|	Expr_1 `union` Expr_2 = Expr_3	(union)
	|	Expr_1 `minus` Expr_2 = Expr_3	(minus)
	|	`edge`(Term_1 Term_2 Term_3 Term_4)	(labeled edge (v v l d))
	|	`edge`(Term_1 Term_2 Term_3)	(edge (v v d)))
	|	`dom`(Term_1 Term_2 Term_3 Term_4)	(labeled strict dominance (v v l d))
	|	`dom`(Term_1 Term_2 Term_3)	(strict dominance (v v d))
	|	`domeq`(Term_1 Term_2 Term_3)	(dominance (v v d))
	|	Term_1 < Term_2	(precedence)
	|	Term_1 . `word` = Term_2	(word)
	|	(Form)	(bracketing)

(B.11)

C. Inferenzregeln für die Typinferenz

Getypte Variable:

$$\frac{\Gamma \vdash \texttt{X} :: \texttt{type}}{\Gamma \cup \{X \mapsto type\} \vdash \texttt{X} :: \texttt{type}} X \mapsto T \in \Gamma \Rightarrow T = type$$

Ungetypte Variable, inferrierbar:

$$\frac{\Gamma \cup \{X \mapsto type\} \vdash \texttt{X}}{\Gamma \cup \{X \mapsto type\} \vdash \texttt{X} :: \texttt{type}}$$

Ungetypte Variable, noch nicht inferrierbar:

$$\frac{\Gamma \vdash \texttt{X}}{\Gamma \cup \{X \mapsto \texttt{\_}\} \vdash \texttt{X}} X \mapsto T \notin \Gamma$$

$\exists$, getypt:

$$\frac{\begin{array}{l}\Gamma \vdash \texttt{exists X :: type : Form} \\ \Gamma \cup \{X \mapsto type\} \vdash \texttt{Form}\end{array}}{\Gamma \vdash \texttt{exists X :: type : Form}} (type \in Dom)$$

$\exists$, ungetypt, inferrierbar:

$$\frac{\begin{array}{l}\Gamma \vdash \texttt{exists X : Form} \\ \Gamma_2 \cup \{X \mapsto type\} \vdash \texttt{Form}\end{array}}{\Gamma \vdash \texttt{exists X :: type : Form}} (type \in Dom)$$

$\exists$, ungetypt, nicht inferrierbar:

$$\frac{\begin{array}{l}\Gamma \vdash \texttt{exists X : Form} \\ \Gamma \vdash \texttt{Form}\end{array}}{\texttt{ErrorCoudnotinferTypeofX}} \begin{array}{l}(con \mapsto T \notin \Gamma) \vee \\ (T = \texttt{\_})\end{array}$$

$\exists!$, getypt:

$$\frac{\begin{array}{l}\Gamma \vdash \texttt{existsone X :: type : Form} \\ \Gamma \cup \{X \mapsto type\} \vdash \texttt{Form}\end{array}}{\Gamma \vdash \texttt{existsone X :: type : Form}} (type \in Dom)$$

$\exists!$, ungetypt, inferrierbar:

$$\frac{\begin{array}{l}\Gamma \vdash \texttt{existsone X : Form} \\ \Gamma_2 \cup \{X \mapsto type\} \vdash \texttt{Form}\end{array}}{\Gamma \vdash \texttt{existsone X :: type : Form}} (type \in Dom)$$

$\exists!$, ungetypt, nicht inferrierbar:

$$\frac{\begin{array}{l}\Gamma \vdash \texttt{existsone X : Form} \\ \Gamma \vdash \texttt{Form}\end{array}}{\texttt{ErrorCoudnotinferTypeofX}} \begin{array}{l}(con \mapsto T \notin \Gamma) \vee \\ (T = \texttt{\_})\end{array}$$

$\forall$, getypt:

$$\frac{\Gamma \vdash \texttt{forall X :: type : Form} \quad \Gamma \cup \{X \mapsto type\} \vdash \texttt{Form}}{\Gamma \vdash \texttt{forall X :: type : Form}} (type \in Dom)$$

$\forall$, ungetypt, inferrierbar:

$$\frac{\Gamma \vdash \texttt{forall X : Form} \quad \Gamma_2 \cup \{X \mapsto type\} \vdash \texttt{Form}}{\Gamma \vdash \texttt{forall X :: type : Form}} (type \in Dom)$$

$\forall$, ungetypt, nicht inferrierbar:

$$\frac{\Gamma \vdash \texttt{forall X : Form} \quad \Gamma \vdash \texttt{Form}}{\texttt{ErrorCoudnotinferTypeofX}} \begin{array}{l}(con \mapsto T \notin \Gamma) \vee \\ (T = \_)\end{array}$$

$v_1 \rightarrow_d v_2$ v_1, v_2 *Konstanten*:

$$\frac{\Gamma \vdash \texttt{edge(v1 v2 d)}}{\Gamma \vdash \texttt{edge(v1 v2 d)}}$$

$v_1 \rightarrow_d v_2$ v_1, v_2, d *Variablen*:

$$\frac{\Gamma \vdash \texttt{edge(v1 v2 d)}}{\begin{array}{l}\Gamma \cup \{v1 \mapsto node, v2 \mapsto node, d \mapsto dim\} \\ \vdash \texttt{edge(v1 v2 d)}\end{array}} \begin{array}{l}(v1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = node) \wedge \\ (v2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = node) \wedge \\ (d \mapsto T_3 \in \Gamma \Rightarrow T_3 = dim)\end{array}$$

$v_1 \xrightarrow{l}_d v_2$ v_1, v_2, l, d *Variablen*:

$$\frac{\Gamma \vdash \texttt{edge(v1 v2 l d)}}{\begin{array}{l}\Gamma \cup \{v1 \mapsto node, v2 \mapsto node, l \mapsto label(d), d \mapsto dim\} \\ \vdash \texttt{edge(v1 :: node v2 :: node l :: label(d)d :: dim)}\end{array}} \begin{array}{l}(v1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = node) \wedge \\ (v2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = node) \wedge \\ (l \mapsto T_3 \in \Gamma \Rightarrow T_3 = label(d)) \\ (d \mapsto T_4 \in \Gamma \Rightarrow T_4 = dim)\end{array}$$

$v_1 \xrightarrow{l}_d v_2$ v_1, v_2, l, d *Konstanten*:

$$\frac{\Gamma \vdash \texttt{edge(v1 v2 l d)}}{\Gamma \vdash \texttt{edge(v1 :: node v2 :: node l :: label(d)d :: dim)}}$$

$v_1 \xrightarrow{l}{}^*_d v_2$ v_1, v_2, d *Variablen*:

$$\frac{\Gamma \vdash \texttt{domeq(v1 v2 d)}}{\begin{array}{l}\Gamma \cup \{v1 \mapsto node, v2 \mapsto node, d \mapsto dim\} \\ \vdash \texttt{domeq(v1 :: node v2 :: node d :: dim)}\end{array}} \begin{array}{l}(v1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = node) \wedge \\ (v2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = node) \wedge \\ (d \mapsto T_3 \in \Gamma \Rightarrow T_3 = dim)\end{array}$$

$v_1 \xrightarrow{l}{}^*_d v_2$ v_1, v_2, d *Konstanten*:

$$\frac{\Gamma \vdash \texttt{domeq(v1 v2 d)}}{\Gamma \vdash \texttt{domeq(v1 :: node v2 :: node d :: dim)}}$$

$v_1 \overset{l}{\rightarrow}_d^+ v_2$ v_1, v_2, l, d *Variablen*:

$$\frac{\Gamma \vdash \texttt{dom(v1 v2 l d)}}{\begin{array}{l}\Gamma \cup \{v1 \mapsto node, v2 \mapsto node, l \mapsto label(d), d \mapsto dim\} \\ \vdash \texttt{dom(v1 :: node v2 :: node l :: label(d)d :: dim)}\end{array}} \quad \begin{array}{l}(v1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = node) \wedge \\ (v2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = node) \wedge \\ (l \mapsto T_3 \in \Gamma \Rightarrow T_3 = label(d)) \\ (d \mapsto T_4 \in \Gamma \Rightarrow T_4 = dim)\end{array}$$

$v_1 \overset{l}{\rightarrow}_d^+ v_2$ v_1, v_2, l, d *Konstanten*:

$$\frac{\Gamma \vdash \texttt{dom(v1 v2 l d)}}{\Gamma \vdash \texttt{dom(v1 :: node v2 :: node l :: label(d)d :: dim)}}$$

$v_1 \rightarrow_d^+ v_2$ v_1, v_2, d *Variablen*:

$$\frac{\Gamma \vdash \texttt{dom(v1 v2 d)}}{\begin{array}{l}\Gamma \cup \{v1 \mapsto node, v2 \mapsto node, d \mapsto dim\} \\ \vdash \texttt{dom(v1 :: node v2 :: node d :: dim)}\end{array}} \quad \begin{array}{l}(v1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = node) \wedge \\ (v2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = node) \wedge \\ (d \mapsto T_3 \in \Gamma \Rightarrow T_3 = dim)\end{array}$$

$v_1 \rightarrow_d^+ v_2$ v_1, v_2, d *Konstanten*:

$$\frac{\Gamma \vdash \texttt{dom(v1 v2 d)}}{\Gamma \vdash \texttt{dom(v1 :: node v2 :: node d :: dim)}}$$

Präzedenz, v1 und v2 Variablen:

$$\frac{\Gamma \vdash \texttt{v1 < v2}}{\begin{array}{l}\Gamma \cup \{v1 \mapsto node, v2 \mapsto node\} \\ \vdash \texttt{v1 :: node < v2 :: node}\end{array}} \quad \begin{array}{l}v1, v2 \mapsto T \notin \Gamma \vee \\ (v1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = (node \wedge \_) \vee \\ v2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (node \wedge \_))\end{array}$$

Präzedenz, v1 und v2 Konstanten:

$$\frac{\Gamma \vdash \texttt{v1 < v2}}{\Gamma \vdash \texttt{v1 :: node < v2 :: node}}$$

Word, v1 und v2 Variablen:

$$\frac{\Gamma \vdash \texttt{v1.word} = \texttt{v2}}{\begin{array}{l}\Gamma \cup \{v1 \mapsto node, v2 \mapsto word\} \\ \vdash \texttt{v1 :: node.word} = \texttt{v2 :: word}\end{array}} \quad \begin{array}{l}v1, v2 \mapsto T \notin \Gamma \vee \\ (v1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = (node \wedge \_) \vee \\ v2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (word \wedge \_))\end{array}$$

Word, v1 und v2 Konstanten:

$$\frac{\Gamma \vdash \texttt{v1.word} = \texttt{v2}}{\Gamma \vdash \texttt{v1 :: node.word} = \texttt{v2 :: word}}$$

Element, teilweise annotiert, inferrierbar:

$$\frac{\Gamma \vdash \texttt{X1 in X2 :: set(label(D))}}{\begin{array}{l}\Gamma \cup \{X1 \mapsto label(D), X2 \mapsto set(label(D))\} \\ \vdash \texttt{X}_1\texttt{:: label(D) in X}_2\texttt{:: set(label(D))}\end{array}} \quad (*)$$

* $X_{1-2} \mapsto T \notin \Gamma \vee$
$(Expr_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = (label(D) \wedge \_) \vee$
$Expr_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_))$

Element, teilweise annotiert, inferrierbar:

$$\frac{\Gamma \vdash \texttt{X1} :: \texttt{label(D) in X2}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto label(D), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{label(D) in X2} :: \texttt{set(label(D))}\end{array}} \quad (*)$$

* $X_{1-2} \mapsto T \notin \Gamma \vee$
$(X_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = (label(D) \wedge \_) \vee$
$X_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_))$

Element, inferrierbar:

$$\frac{\Gamma \cup \{X1 \mapsto label(D)\} \vdash \texttt{X1 in X2}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto label(D), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{label(D) in X2} :: \texttt{set(label(D))}\end{array}} \quad (*)$$

* $X_2 \mapsto T \notin \Gamma \vee$
$X_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_))$

Element, inferrierbar:

$$\frac{\Gamma \cup \{X_2 \mapsto set(label(D))\} \vdash \texttt{X1 in X}_\texttt{2}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto label(D), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{label(D) in X2} :: \texttt{set(label(D))}\end{array}} \quad (*)$$

* $X_1 \mapsto T \notin \Gamma \vee$
$X_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = (label(D) \wedge \_))$

Nicht Element, teilweise annotiert, inferrierbar:

$$\frac{\Gamma \vdash \texttt{X1 notin X2} :: \texttt{set(label(D))}}{\begin{array}{l}\Gamma \cup \{X1 \mapsto label(D), X2 \mapsto set(label(D))\} \\ \vdash \texttt{X}_\texttt{1}\texttt{:: label(D) notin X}_\texttt{2}\texttt{:: set(label(D))}\end{array}} \quad (*)$$

* $X_{1-2} \mapsto T \notin \Gamma \vee$
$(Expr_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = (label(D) \wedge \_) \vee$
$Expr_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_))$

Nicht Element, teilweise annotiert, inferrierbar:

$$\frac{\Gamma \vdash \texttt{X1} :: \texttt{label(D) notin X2}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto label(D), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{label(D) notin X2} :: \texttt{set(label(D))}\end{array}} \quad (*)$$

* $X_{1-2} \mapsto T \notin \Gamma \vee$
$(X_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = (label(D) \wedge \_) \vee$
$X_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_))$

Nicht Element, inferrierbar:

$$\frac{\Gamma \cup \{X1 \mapsto label(D)\} \vdash \texttt{X1 notin X2}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto label(D), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{label(D) notin X2} :: \texttt{set(label(D))}\end{array}} \quad (*)$$

$$\begin{array}{ll} * & X_2 \mapsto T \notin \Gamma \vee \\ & X_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_)) \end{array}$$

Nicht Element, inferrierbar:

$$\frac{\Gamma \cup \{X_2 \mapsto set(label(D))\} \vdash \texttt{X1 notin X}_2}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto label(D), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{label(D) notin X2} :: \texttt{set(label(D))}\end{array}} \quad (*)$$

$$\begin{array}{ll} * & X_1 \mapsto T \notin \Gamma \vee \\ & X_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = (label(D) \wedge \_)) \end{array}$$

Teilmenge, teilweise annotiert, inferrierbar:

$$\frac{\Gamma \vdash \texttt{X1 subseteq X2} :: \texttt{set(label(D))}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto set(label(D)), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{set(label(D)) subseteq X2} :: \texttt{set(label(D))}\end{array}} \quad (*)$$

$$\begin{array}{ll} * & X_{1-2} \mapsto T \notin \Gamma \vee \\ & (X_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = (set(label(D)) \wedge \_) \vee \\ & X_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_)) \end{array}$$

Teilmenge, teilweise annotiert, inferrierbar:

$$\frac{\Gamma \vdash \texttt{X1} :: \texttt{label(D) subseteq X2}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto set(label(D)), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{set(label(D)) subseteq X2} :: \texttt{set(label(D))}\end{array}} \quad (*)$$

$$\begin{array}{ll} * & X_{1-2} \mapsto T \notin \Gamma \vee \\ & (X_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = (set(label(D)) \wedge \_) \vee \\ & X_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_)) \end{array}$$

Teilmenge, inferrierbar:

$$\frac{\Gamma \cup \{X_1 \mapsto set(label(D))\} \vdash \texttt{X1 subseteq X2}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto set(label(D)), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{set(label(D)) subseteq X2} :: \texttt{set(label(D))}\end{array}} \quad (*)$$

$$\begin{array}{ll} * & X_2 \mapsto T \notin \Gamma \vee \\ & X_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_)) \end{array}$$

Teilmenge, inferrierbar:

$$\frac{\Gamma \cup \{X_2 \mapsto set(label(D))\} \vdash \texttt{X1 subseteq X2}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto set(label(D)), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{set(label(D)) subseteq X2} :: \texttt{set(label(D))}\end{array}} \quad (*)$$

* $X_1 \mapsto T \notin \Gamma \vee$
  $X_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = set((label(D)) \wedge \_))$

Disjunktheit, teilweise annotiert, inferrierbar:

$$\frac{\Gamma \vdash \texttt{X1 disjoint X2} :: \texttt{set(label(D))}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto set(label(D)), X_2 \mapsto set(label(D)) \\ \vdash \texttt{X1} :: \texttt{set(label(D))}\ \texttt{disjoint X2} :: \texttt{set(label(D))}\end{array}} \quad (*)$$

* $X_{1-2} \mapsto T \notin \Gamma \vee$
  $(X_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = (set(label(D)) \wedge \_) \vee$
  $X_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_))$

Disjunktheit, teilweise annotiert, inferrierbar:

$$\frac{\Gamma \vdash \texttt{X1} :: \texttt{label(D) disjoint X2}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto set(label(D)), X_2 \mapsto set(label(D)) \\ \vdash \texttt{X1} :: \texttt{set(label(D))}\ \texttt{disjoint X2} :: \texttt{set(label(D))}\end{array}} \quad (*)$$

* $X_{1-2} \mapsto T \notin \Gamma \vee$
  $(X_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = (set(label(D)) \wedge \_) \vee$
  $X_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_))$

Disjunktheit, inferrierbar:

$$\frac{\Gamma \cup \{X_1 \mapsto set(label(D))\} \vdash \texttt{X1 disjoint X2}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto set(label(D)), X_2 \mapsto set(label(D)) \\ \vdash \texttt{X1} :: \texttt{set(label(D))}\ \texttt{disjoint X2} :: \texttt{set(label(D))}\end{array}} \quad (*)$$

* $X_2 \mapsto T \notin \Gamma \vee$
  $X_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_))$

Disjunktheit, inferrierbar:

$$\frac{\Gamma \cup \{X_2 \mapsto set(label(D))\} \vdash \texttt{X1 disjoint X2}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto set(label(D)), X_2 \mapsto set(label(D)) \\ \vdash \texttt{X1} :: \texttt{set(label(D))}\ \texttt{disjoint X2} :: \texttt{set(label(D))}\end{array}} \quad (*)$$

* $X_1 \mapsto T \notin \Gamma \vee$
  $X_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = set((label(D)) \wedge \_))$

Schnitt, teilweise annotiert, inferrierbar:

$$\frac{\Gamma \vdash \texttt{X1 intersect X2} :: \texttt{set(label(D))}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto set(label(D)), X_2 \mapsto set(label(D)) \\ \vdash \texttt{X1} :: \texttt{set(label(D))}\ \texttt{intersect X2} :: \texttt{set(label(D))}\end{array}} \quad (*)$$

* $X_{1-2} \mapsto T \notin \Gamma \vee$
  $(X_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = (set(label(D)) \wedge \_) \vee$
  $X_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_))$

Schnitt, teilweise annotiert, inferrierbar:

$$\frac{\Gamma \vdash \texttt{X1} :: \texttt{label(D) intersect X2}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto set(label(D)), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{set(label(D))} \ \texttt{intersect X2} :: \texttt{set(label(D))}\end{array}} \ (*)$$

$$* \quad \begin{array}{l} X_{1-2} \mapsto T \notin \Gamma \vee \\ (X_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = (set(label(D)) \wedge \_) \vee \\ X_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_)) \end{array}$$

Schnitt, inferrierbar:

$$\frac{\Gamma \cup \{X_1 \mapsto set(label(D))\} \vdash \texttt{X1 intersect X2}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto set(label(D)), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{set(label(D))} \ \texttt{intersect X2} :: \texttt{set(label(D))}\end{array}} \ (*)$$

$$* \quad \begin{array}{l} X_2 \mapsto T \notin \Gamma \vee \\ X_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_)) \end{array}$$

Schnitt, inferrierbar:

$$\frac{\Gamma \cup \{X_2 \mapsto set(label(D))\} \vdash \texttt{X1 intersect X2}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto set(label(D)), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{set(label(D))} \ \texttt{intersect X2} :: \texttt{set(label(D))}\end{array}} \ (*)$$

$$* \quad \begin{array}{l} X_1 \mapsto T \notin \Gamma \vee \\ X_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = set((label(D)) \wedge \_)) \end{array}$$

Vereinigung, teilweise annotiert, inferrierbar:

$$\frac{\Gamma \vdash \texttt{X1 union X2} :: \texttt{set(label(D))}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto set(label(D)), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{set(label(D))} \ \texttt{union X2} :: \texttt{set(label(D))}\end{array}} \ (*)$$

$$* \quad \begin{array}{l} X_{1-2} \mapsto T \notin \Gamma \vee \\ (X_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = (set(label(D)) \wedge \_) \vee \\ X_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_)) \end{array}$$

Vereinigung, teilweise annotiert, inferrierbar:

$$\frac{\Gamma \vdash \texttt{X1} :: \texttt{label(D) union X2}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto set(label(D)), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{set(label(D))} \ \texttt{union X2} :: \texttt{set(label(D))}\end{array}} \ (*)$$

$$* \quad \begin{array}{l} X_{1-2} \mapsto T \notin \Gamma \vee \\ (X_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = (set(label(D)) \wedge \_) \vee \\ X_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_)) \end{array}$$

Vereinigung, inferrierbar:

$$\frac{\Gamma \cup \{X_1 \mapsto set(label(D))\} \vdash \texttt{X1 union X2}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto set(label(D)), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{set(label(D))} \ \texttt{union X2} :: \texttt{set(label(D))}\end{array}} \ (*)$$

* $X_2 \mapsto T \notin \Gamma \vee$
$X_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_))$

Vereinigung, inferrierbar:

$$\frac{\Gamma \cup \{X_2 \mapsto set(label(D))\} \vdash \texttt{X1 union X2}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto set(label(D)), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{set(label(D))}\ \texttt{union X2} :: \texttt{set(label(D))}\end{array}} \quad (*)$$

* $X_1 \mapsto T \notin \Gamma \vee$
$X_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = set((label(D)) \wedge \_))$

Minus, teilweise annotiert, inferrierbar:

$$\frac{\Gamma \vdash \texttt{X1 minus X2} :: \texttt{set(label(D))}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto set(label(D)), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{set(label(D))}\ \texttt{minus X2} :: \texttt{set(label(D))}\end{array}} \quad (*)$$

* $X_{1-2} \mapsto T \notin \Gamma \vee$
$(X_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = (set(label(D)) \wedge \_) \vee$
$X_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_))$

Minus, teilweise annotiert, inferrierbar:

$$\frac{\Gamma \vdash \texttt{X1} :: \texttt{label(D) minus X2}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto set(label(D)), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{set(label(D))}\ \texttt{minus X2} :: \texttt{set(label(D))}\end{array}} \quad (*)$$

* $X_{1-2} \mapsto T \notin \Gamma \vee$
$(X_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = (set(label(D)) \wedge \_) \vee$
$X_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_))$

Minus, inferrierbar:

$$\frac{\Gamma \cup \{X_1 \mapsto set(label(D))\} \vdash \texttt{X1 minus X2}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto set(label(D)), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{set(label(D))}\ \texttt{minus X2} :: \texttt{set(label(D))}\end{array}} \quad (*)$$

* $X_2 \mapsto T \notin \Gamma \vee$
$X_2 \mapsto T_2 \in \Gamma \Rightarrow T_2 = (set(label(D)) \wedge \_))$

Minus, inferrierbar:

$$\frac{\Gamma \cup \{X_2 \mapsto set(label(D))\} \vdash \texttt{X1 minus X2}}{\begin{array}{l}\Gamma \cup \{X_1 \mapsto set(label(D)), X_2 \mapsto set(label(D))\} \\ \vdash \texttt{X1} :: \texttt{set(label(D))}\ \texttt{minus X2} :: \texttt{set(label(D))}\end{array}} \quad (*)$$

* $X_1 \mapsto T \notin \Gamma \vee$
$X_1 \mapsto T_1 \in \Gamma \Rightarrow T_1 = set((label(D)) \wedge \_))$

D. Regeln der Interpretationsfunktion

$[\![Constant]\!]^{V,\ M,\ T}$ =

```
if {Char.isAlpha {Atom.toString Constant}.1} andthen
   {Char.isUpper {Atom.toString Constant}.1} andthen
   {List.all {Atom.toString Constant}
      fun {$ Ch} {Char.isAlpha Ch} orelse {Char.isDigit Ch} end}} then
   if M == i then
      if T == node then
         V.n.Constant.index
      else
         V.i.Constant
      end
   elseif M == a then
      V.a.Constant
   else
      V.n.Constant
   end
else
   if M == i then
      '{{T2Lat '#T#'}.a2I '#Constant#'}'
   elseif M == a then
      Constant
   end
end
```

$[\![Integer]\!]^{V,\ M,\ T}$ =

```
if T == node then
   if M == i then
      Integer
   else
      '{Nth NodesRecs '#Integer#'}'
   end
end
```

$[\![\{Term+\}]\!]^{V,M,set(Dom)}$ =

```
local
   Vs = {Map Term+ fun {$ Term} [[Term]]^{V,i,Dom} end}
   V = '['#{FoldL Vs fun {$ AccV V} AccV#' '#V end}#']'
in
   '{FS.value.make '#V#'}'
end
```

$[\![[Term+]]\!]^{V,M,tuple(Dom+)} =$

```
    local
      Vs = {List.mapInd Term+
        fun {$ I Term}
          Dom = {Nth Dom+ I}
        in
          [[Term]]^{V,a,Dom}
        end}
      V = '['#{FoldL Vs fun {$ AccV V} AccV#' '#V end}#']'
    in
      '{{T2Lat '#tuple(Dom+)#'}.as2I '#V#'}'
    end
```

$[\![[Term+]+]\!]^{V,M,set(tuple(Dom+))} =$

```
    if M == i then
      Vs = {Map [Term+]+
        fun {$ Term+}
          [[Term+]]^{V,M,tuple(Dom+)}
        end}
      V = '['#{FoldL Vs fun {$ AccV V} AccV#' '#V end}#']'
    in
      '{FS.value.make '#V#'}'
    end
```

$[\![Expr :: Type]\!]^{V,\ M,\ T} = [\![Expr]\!]^{V,M,Type}$

$[\![Term_1.Term_2.entry.Constant]\!]^{V,\ M,\ T} =$

```
    [[Term1]]^{V,n,node} #'.'#
    [[Term2]]^{V,a,dim} #'.entry.'#
    [[Constant]]^{V,a,attr}
```

$[\![Term_1.Term_2.attrs.Constant]\!]^{V,\ M,\ T} =$

```
    [[Term1]]^{V,n,node} #'.'#
    [[Term2]]^{V,a,dim} #'.attrs.'#
    [[Constant]]^{V,a,attr}
```

$[\![(Expr)]\!]^{V,\ M,\ T} = [\![Expr]\!]^{V,\ M,\ T}$

$[\![\sim\ Form]\!]^{V,\ M,\ T} =$ `'{PW.nega '#` $[\![Form]\!]^{V,\ M,\ T}$`#'}'`

$[\![Form_1\ \&\ Form_2]\!]^{V,\ M,\ T} =$

```
    '{PW.conj '#
      [[Form1]]^{V, M, T}#' '#
      [[Form2]]^{V, M, T}#
    '}'
```

$[\![Form_1 \mid Form_2]\!]^{V,\ M,\ T} =$

```
'{PW.disj '#
```
$[\![Form_1]\!]^{V,\ M,\ T}$`#' '#`
$[\![Form_2]\!]^{V,\ M,\ T}$`#`
```
'}'
```

$[\![Form_1 \ => \ Form_2]\!]^{V,\ M,\ T} =$

```
'{PW.impl '#
```
$[\![Form_1]\!]^{V,\ M,\ T}$`#' '#`
$[\![Form_2]\!]^{V,\ M,\ T}$`#`
```
'}'
```

$[\![Form_1 \ <=> \ Form_2]\!]^{V,\ M,\ T} =$

```
'{PW.equi '#
```
$[\![Form_1]\!]^{V,\ M,\ T}$`#' '#`
$[\![Form_2]\!]^{V,\ M,\ T}$`#`
```
'}'
```

$[\![exists\ Constant :: Dom : Form]\!]^{V,\ M,\ T} =$

if Dom == node then
  NodeRecV = Constant#`'NodeRec'`
in
  `'{PW.existsNodes NodeRecs`
    `fun {$ '#`NodeRecV`#'}'#`
      $[\![Form]\!]\big(\ V.n \cup \{Constant : NodeRecV\}\ \big),M,T$`#`
    `end}'`
else
  AV = Constant#`'A'`
  IV = Constant#`'I'`
in
  `'{PW.existsDom {T2Lat '#`Dom`#'}`
    $[\![Form]\!]\begin{pmatrix} V.a \cup \{Constant : AV\} \\ V.i \cup \{Constant : IV\} \end{pmatrix}_{M,T}$ `#`
    `end}'`
end

$[\![existsone\ Constant :: Dom : Form]\!]^{V,\ M,\ T} =$

if Dom == node then
  NodeRecV = Constant#'NodeRec'
in
  '{PW.existsOneNodes NodeRecs
    fun {$ '#NodeRecV#'}'#
      $[\![Form]\!]^{(\ V.n \cup \{Constant : NodeRecV\}\ ),M,T}$#
    end}'
else
  AV = Constant#'A'
  IV = Constant#'I'
in
  '{PW.existsOneDom {T2Lat '#Dom#'}
    $[\![Form]\!]^{\left(\begin{array}{l} V.a \cup \{Constant : AV\} \\ V.i \cup \{Constant : IV\} \end{array}\right),M,T}$#
    end}'
end

$[\![forall\ Constant :: Dom : Form]\!]^{V,\ M,\ T} =$

if Dom == node then
  NodeRecV = Constant#'NodeRec'
in
  '{PW.forAllNodes NodeRecs
    fun {$ '#NodeRecV#'}'#
      $[\![Form]\!]^{(\ V.n \cup \{Constant : NodeRecV\}\ ),M,T}$#
    end}'
else
  AV = Constant#'A'
  IV = Constant#'I'
in
  '{PW.forAllDom {T2Lat '#Dom#'}
    $[\![Form]\!]^{\left(\begin{array}{l} V.a \cup \{Constant : AV\} \\ V.i \cup \{Constant : IV\} \end{array}\right),M,T}$#
    end}'
end

$[\![Expr_1\ =\ Expr_2]\!]^{V,\ M,\ T} =$

'{PW.equals '#$[\![Expr_1]\!]^{V,i,T}$#' '#$[\![Expr_2]\!]^{V,i,T}$#}'

$[\![Expr_1\ in\ Expr_2]\!]^{V,\ M,\ T} =$

'{PW.'in' '#$[\![Expr_1]\!]^{V,i,T}$#' '#$[\![Expr_2]\!]^{V,i,T}$#}'

$[\![Expr_1\ notin\ Expr_2]\!]^{V,\ M,\ T} =$

'{PW.notin '#$[\![Expr_1]\!]^{V,i,T}$#' '#$[\![Expr_2]\!]^{V,i,T}$#}'

$[\![Expr_1\ subseteq\ Expr_2]\!]^{V,\ M,\ T} =$

'{PW.subseteq '#$[\![Expr_1]\!]^{V,i,T}$#' '#$[\![Expr_2]\!]^{V,i,T}$#}'

$[\![Expr_1\ disjoint\ Expr_2]\!]^{V,\ M,\ T} =$
'{PW.disjoint '#$[\![Expr_1]\!]^{V,i,T}$#' '#$[\![Expr_2]\!]^{V,i,T}$#}'

$[\![Expr_1\ intersect\ Expr_2\ =\ Expr_3]\!]^{V,\ M,\ T} =$
'{PW.intersect '#
$[\![Expr_1]\!]^{V,i,T}$#' '#$[\![Expr_2]\!]^{V,i,T}$#' '#$[\![Expr_3]\!]^{V,i,T}$#}'

$[\![Expr_1\ union\ Expr_2\ =\ Expr_3]\!]^{V,\ M,\ T} =$
'{PW.union '#
$[\![Expr_1]\!]^{V,i,T}$#' '#$[\![Expr_2]\!]^{V,i,T}$#' '#$[\![Expr_3]\!]^{V,i,T}$#}'

$[\![Expr_1\ minus\ Expr_2\ =\ Expr_3]\!]^{V,\ M,\ T} =$
'{PW.minus '#
$[\![Expr_1]\!]^{V,i,T}$#' '#$[\![Expr_2]\!]^{V,i,T}$#' '#$[\![Expr_3]\!]^{V,i,T}$#}'

$[\![edge(Expr_1\ Expr_2\ Expr_3\ Expr_4]\!]^{V,\ M,\ T} =$
'{PW.labeledEdge '#
$[\![Expr_1]\!]^{V,n,node}$#' '#
$[\![Expr_2]\!]^{V,n,node}$#' '#
$[\![Expr_3]\!]^{V,a,label(Expr_4)}$#' '#
$[\![Expr_4]\!]^{V,a,dim}$#'}'

$[\![edge(Expr_1\ Expr_2\ Expr_3]\!]^{V,\ M,\ T} =$
'{PW.edge '#
$[\![Expr_1]\!]^{V,n,node}$#' '#
$[\![Expr_2]\!]^{V,n,node}$#' '#
$[\![Expr_3]\!]^{V,a,dim}$#'}'

$[\![dom(Expr_1\ Expr_2\ Expr_3\ Expr_4]\!]^{V,\ M,\ T} =$
'{PW.labeledStrictDominance '#
$[\![Expr_1]\!]^{V,n,node}$#' '#
$[\![Expr_2]\!]^{V,n,node}$#' '#
$[\![Expr_3]\!]^{V,a,label(Expr_4)}$#' '#
$[\![Expr_4]\!]^{V,a,dim}$#'}'

$[\![dom(Expr_1\ Expr_2\ Expr_3]\!]^{V,\ M,\ T} =$
'{PW.strictDominance '#
$[\![Expr_1]\!]^{V,n,node}$#' '#
$[\![Expr_2]\!]^{V,n,node}$#' '#
$[\![Expr_3]\!]^{V,a,dim}$#'}'

$[\![domeq(Expr_1\ Expr_2\ Expr_3]\!]^{V,\ M,\ T} =$
'{PW.dominance '#
$[\![Expr_1]\!]^{V,n,node}$#' '#
$[\![Expr_2]\!]^{V,n,node}$#' '#
$[\![Expr_3]\!]^{V,a,dim}$#'}'

$[\![Expr_1\ <\ Expr_2]\!]^{V,\ M,\ T} =$
'{PW.prec #'$[\![Expr_1]\!]^{V,n,node}$#' '#$[\![Expr_2]\!]^{V,n,node}$#'}'

$[\![Expr_1.word \ = \ Expr_2]\!]^{V,\ M,\ T} =$
'{PW.word #'$[\![Expr_1]\!]^{V,n,node}$#' '#$[\![Expr_2]\!]^{V,a,word}$#'}'

$[\![(Form)]\!]^{V,\ M,\ T} = [\![Form]\!]^{V,\ M,\ T}$

Literaturverzeichnis

Apt, K. R. & Vermeulen, C. F. M. (2002), First-order logic as a constraint programming language, *in* 'LPAR 02: Proceedings of the 9th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning', Springer, pp. 19–35.

Carpenter, B. & Penn, G. (1994), ALE the Attribute Logic Engine. user's guide, Technical report, Carnegie Mellon University, Pittsburgh/US.

Debusmann, R. (2006), Extensible Dependency Grammar: A Modular Grammar Formalism Based On Multigraph Description, PhD thesis, Universität des Saarlandes.

Debusmann, R. (2007*a*), The complexity of First-Order Extensible Dependency Grammar, Technical report, Saarland University.

Debusmann, R. (2007*b*), Scrambling as the intersection of relaxed context-free grammars in a model-theoretic grammar formalism, *in* 'ESSLLI 2007 Workshop Model Theoretic Syntax at 10', Dublin/IE.

Debusmann, R., Duchier, D., Koller, A., Kuhlmann, M., Smolka, G. & Thater, S. (2004), A relational syntax-semantics interface based on dependency grammar, *in* 'Proceedings of COLING 2004', Geneva/CH.

Debusmann, R., Duchier, D. & Niehren, J. (2004), The XDG grammar development kit, *in* 'Proceedings of the MOZ04 Conference', Vol. 3389 of *Lecture Notes in Computer Science*, Springer, Charleroi/BE, pp. 190–201.

Duchier, D. (1999), Axiomatizing dependency parsing using set constraints, *in* 'Proceedings of MOL 6', Orlando/US.

Duchier, D. (2003), 'Configuration of labeled trees under lexicalized constraints and principles', *Research on Language and Computation* **1**(3–4), 307–336.

Duchier, D. & Debusmann, R. (2001), Topological dependency trees: A constraint-based account of linear precedence, *in* 'Proceedings of ACL 2001', Toulouse/FR.

Egg, M., Koller, A. & Niehren, J. (2001), 'The Constraint Language for Lambda Structures', *Journal of Logic, Language, and Information* .

Gazdar, G., Klein, E., Pullum, G. & Sag, I. (1985), *Generalized Phrase Structure Grammar*, B. Blackwell, Oxford/UK.

Gerdemann, D., Hinrichs, E., King, P. J. & Götz, T. (1994), 'Troll—type resolution system. user's guide'.

Hudson, R. A. (1990), *English Word Grammar*, B. Blackwell, Oxford/UK.

Jackendoff, R. (2002), *Foundations of Language*, Oxford University Press.

Joshi, A. K. (1987), An introduction to tree-adjoining grammars, *in* A. Manaster-Ramer, ed., 'Mathematics of Language', John Benjamins, Amsterdam/NL, pp. 87–115.

Kuhlmann, M. (2007), Dependency Structures and Lexicalized Grammars, Doctoral dissertation, Saarland University, Saarbrücken/DE.

Kuhlmann, M. & Möhl, M. (2007), Mildly context-sensitive dependency languages, *in* 'Proceedings of ACL 2007', Prague/CZ.

Meurers, D., Penn, G. & Richter, F. (2002), 'Trale—troll and ale'.

Müller, S. (1996), The Babel-System–an HPSG Prolog implementation, *in* 'Proceedings of the Fourth International Conference on the Practical Application of Prolog', London/UK, pp. 263–277.

Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J. & Tack, G. (2007), Minizinc: Towards a standard cp modelling language, *in* '13th International Conference on Principles and Practice of Constraint Programming', LNCS, Springer.

Schulte, C. (2002), *Programming Constraint Services*, Vol. 2302 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag.

Shieber, S. M. (1984), The design of a computer language for linguistic information, *in* 'Proceedings of COLING 1984', pp. 362–366.

Smolka, G. (1995), The Oz programming model, *in* J. van Leeuwen, ed., 'Computer Science Today', Lecture Notes in Computer Science, vol. 1000, Springer-Verlag, Berlin/DE, pp. 324–343.

Steedman, M. (2000), *The Syntactic Process*, MIT Press, Cambridge/US.

Tack, G., Schulte, C. & Smolka, G. (2006), Generating propagators for finite set constraints, *in* F. Benhamou, ed., '12th International Conference on Principles and Practice of Constraint Programming', Vol. 4204 of *Lecture Notes in Computer Science*, Springer, pp. 575–589.

Tesnière, L. (1959), *Eléments de Syntaxe Structurale*, Klincksiek, Paris/FR.

van Hentenryck, P. (1999), *The OPL Optimization Programming Language*, MIT Press, Cambridge/US.