HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Wissensbasierte Systeme, SS 07

The Mozart Programming System

Tobias Ruf Jan Tammen

24. Juni 2007

Prof. Dr. Hedtstück, HTWG Konstanz

Inhaltsverzeichnis

1. Einleitung

1.1. The Mozart Programming System

In der vorliegenden Ausarbeitung soll das „Mozart Programming System“ [1] vorgestellt werden. Bei *Mozart* handelt es sich um eine Programmierumgebung, die die multiparadigmische Programmiersprache *Oz* implementiert.

Entstanden ist Mozart-Oz ursprünglich als Forschungsprojekt am Deutschen Forschungszentrum für Künstliche Intelligenz (DFKI) sowie der Universität Saarbrücken. Inzwischen wird die Sprache vom Mozart-Konsortium, zu welchem Arbeitsgruppen aus Belgien, Schweden und Deutschland gehören, weiterentwickelt. Die Quellen sind unter einer Open-Source-Lizenz verfügbar; ein kommerzieller Einsatz der Sprache ist ebenfalls möglich.

Ähnlich wie in Java können Oz-Programme in eine Art Byte-Code übersetzt werden und laufen anschließend in einer virtuellen Maschine ab. Auf diese Weise wird eine gewisse Plattformunabhängigkeit erreicht.

1.2. Die Programmiersprache Oz

1.2.1. Features

Als Hauptfeatures und -vorteile von Oz gelten die folgenden Aspekte:

Nebenläufigkeit Arbeit mit leichtgewichtigen Threads, Datenfluss-Synchronisation.

Inferencing Constraintbasierte und logische Programmierung.

Verteilung Transparente Netzwerkunterstützung.

Flexibilität Dynamische Typisierung, inkrementelles Kompilieren.

[1] `http://www.mozart-oz.org`

1.2.2. Datentypen

Die Sprache stellt eine Reihe von Datentypen zur Verfügung, deren Hierarchie in Abbildung 1.1 dargestellt ist.

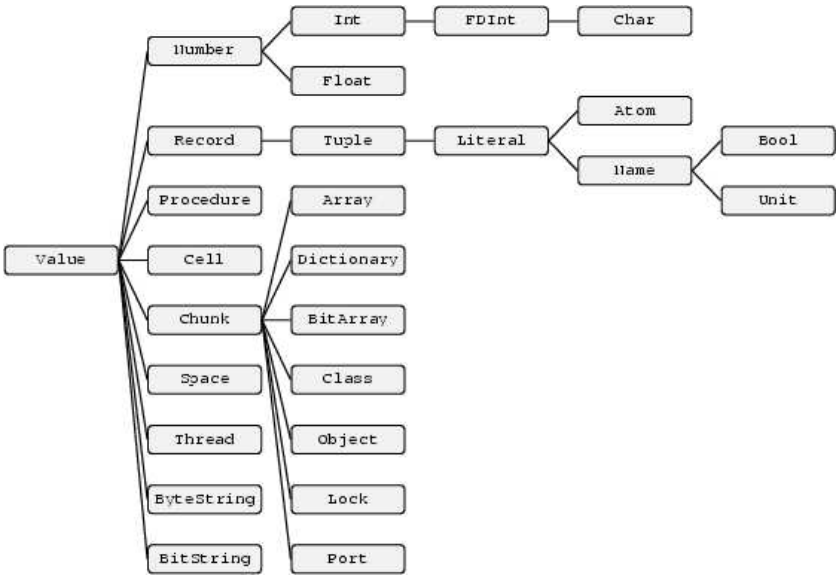


Abbildung 1.1.: Hierarchie der Datentypen in Oz 3. Quelle: [Moz07, Tutorial of Oz, Chapter 3.1]

Einige der wichtigsten Datentypen seien hier kurz aufgeführt (vgl. [Bru00]).

1. Einfache Werte

   Zahlen können in Oz sowohl als ganze Zahlen (`Int, Char`) als auch Fließkommazahlen (`Float`) benutzt werden.

   Literale teilen sich auf in sog. Atome und Namen. Ein Atom wird durch eine alphanumerische Zeichenkette beschrieben, die entweder mit einem Kleinbuchstaben beginnt oder in Hochkomma gefasst ist. Namen sind eindeutige Bezeichner, die über eine spezielle Prozedur namens `NewName` erzeugt werden können.

   Prozeduren können in Oz an Variablen gebunden und auch zur Laufzeit erzeugt werden.

2. Zusammengesetzte Werte

Records bestehen aus einem Bezeichner (Label) sowie einer festen Anzahl von Komponenten oder Argumenten. Argumente bestehen aus dem Tupel (Feature, Feld).

Tupel sind ein Spezialfall von Records, bei denen die Argumente kein explizites Feature besitzen.

Listen sind eine Sonderform der Tupel. Die leere Liste wird mit `nil` denotiert, offene Listen bspw. als `1|2|3|nil` und geschlossene Listen (also Listen mit fester Elementanzahl) als `[1 2 3]`.

3. **Chunks** erlauben es, abstrakte Datentypen zu konstruieren. Oz bringt bereits einige vordefinierte Chunks mit, z.B. `Array` oder `Dictionary`.

1.2.3. Multiparadigmisch?

Wie bereits eingangs erwähnt, unterstützt Oz mehrere Programmierparadigmen:

1. Constraint-Programmierung
2. Funktionale Programmierung
3. Objektorientierte Programmierung
4. Logische Programmierung

Im Gegensatz zu einer Programmiersprache, die nur eines der Paradigmen unterstützt, lassen sich in Oz also die zu lösenden Probleme von mehreren Seiten gleichzeitig mit dem jeweils geeignetsten Paradigma bearbeiten. Erreichen könnte man dies zwar auch durch die Kombination verschiedener Programmiersprachen, dabei bliebe aber der Nachteil, dass man semantische Lücken überwinden und Schnittstellen zwischen den einzelnen Sprachen definieren müsste. Dies würde u.a. zu einer aufwendigeren Fehlersuche führen. Mit Oz lassen sich hingegen die verschiedene Paradigmen problemlos miteinander kombinieren, deren gemeinsame Basis, das Oz Programming Model, im nächsten Abschnitt vorgestellt wird.

1.2.4. Programmiermodell (Oz Programming Model, OPM)

Die Grundlage für Berechnungen in Oz bildet das so genannte *Concurrent Constraint Programming*. Alle weiteren Paradigmen werden durch sog. „syntactic sugar“ (Syntaxerweiterungen) in die Sprache integriert [MS96].

Allgemein verwendet das Modell für Berechnungen die Metapher eines sog. *Berechnungsraums* (Computational Space). In diesem befindet sich zum einen ein *Speicher*, zum anderen eine Anzahl von sog. *Aktoren*. Aktoren führen die eigentlich Berechnung durch, indem sie schrittweise reduziert werden und sich dabei über den gemeinsamen Speicher synchronisieren. Dazu können Aktoren Information in den Speicher schreiben (tell) und auf Information warten und diese anfordern (ask).

Nebenläufigkeit (Concurrency) ist einer der wichtigsten Aspekte des OPM. Dabei bedeutet Nebenläufigkeit, dass verschiedene Berechnungen unabhängig voneinander durchgeführt werden können, nicht, dass diese parallel ablaufen.

2. Paradigmen

2.1. Constraint-Programmierung

2.1.1. Constraints

Ein Constraint (engl. *to constrain* - dt. *einschränken*) ist eine logische Formel, welche die möglichen Werte einer Variablen beschränkt [MS96]. In Oz gibt es mehrere Constrainttypen. Elementare Constraints sind Gleichungen zwischen Variablen bzw. zwischen einer Variablen und einer Struktur. Listing 2.1 zeigt einige Beispiele für elementare Constraints.

```
1 X = 23
2 X = Y
3 X = pair(Y Z)
4 X = student(matrikel:M semester:S name:N)
```

Listing 2.1: Beispiele für elementare Constraints

Neben diesen elementaren Constraints lassen sich in Oz auch sog. *finite domain constraints* verwenden. Mit diesen können Variablen auf endliche Intervalle ganzer Zahlen beschränkt werden. Beispielsweise schränkt `X :: 1#42` die Variable X auf ganze Zahlen zwischen 1 und 42 ein, also $X \in (1, 42)$.

2.1.2. Constraintbasierte Problemlösung

Es existiert eine Reihe kombinatorischer Probleme, die sich mit Variablen ausdrücken lassen, die ganzzahlige, nichtnegative Werte in einem abgeschlossenen Intervall annehmen. Um diese Art von Problemen mithilfe von Oz zu lösen, lassen sich die vorgestellten finite domain constraints verwenden.

Einige Beispiele für solche Probleme sind:

N **Damen.** Auf einem Schachbrett sollen N Damen so platziert werden, dass sie sich gegenseitig nicht schlagen können.

Kartenfärbung. Eine Landkarte (oder allgemein ein Graph) soll so eingefärbt werden, dass benachbarte Länder unterschiedliche Farben haben und die Anzahl der verwendeten Farben minimal ist.

Send More Money. Gegeben sei die Gleichung $SEND+MORE = MONEY$. Nun soll den einzelnen Buchstaben jeweils eine Ziffer zwischen 0 und 9 zugewiesen werden, sodass die Gleichung erfüllt ist. Weiterhin soll $S \neq 0$ sowie $M \neq 0$ gelten.

Konkret werden diese Probleme mithilfe zweier Techniken gelöst, die wir im folgenden vorstellen möchten.

2.1.2.1. Propagierung

Der Constraintspeicher enthält lediglich die zuvor eingeführten „einfachen“ Constraints. Zur Lösung kombinatorischer Probleme werden aber u.U. komplexere Constraints, wie z.B. arithmetische Gleichungen benötigt. Hier kommen die sog. Propagierer ins Spiel. Ein Propagierer hat eine deklarative Semantik; bei dessen Reduktion werden diejenigen „einfachen“ Constraints im Constraintspeicher explizit gemacht, die durch die Semantik des Propagierers impliziert werden. Beispiele für Propagierer sind in Listing 2.2 aufgeführt.

```
1 X < Y
2 X^2 + Y^2 = Z^2
3 5X - 3Y > 2Z
```

Listing 2.2: Beispiele für Propagierer

Beispiel Gegeben sei ein Constraintspeicher mit dem folgenden Inhalt:

$$X \in 0\#9 \wedge Y \in 0\#9$$

sowie zwei Propagatoren:

$$P_1 : X + Y = 9, \qquad P_2 : 2X + 4Y = 24$$

Die Propagierung läuft nun folgendermaßen ab:

1. P_1 liefert keine neuen Erkenntnisse, P_2 hingegen kann aufgrund der im Constraintstore vorhandenen Information die Wertebereiche für X und Y einschränken: $X \in 0\#8$ und $Y \in 2\#6$.

2. P_1 kann nun X einschränken: $X \in 3\#7$; Y bleibt unverändert.

3. P_2 schränkt weiter ein: $X \in 4\#6$ und $Y \in 3\#4$.

4. P_1 wird erneut aktiviert und es ergibt sich: $X \in 5\#6$ und $Y \in 3\#4$.

5. Schließlich folgert P_2: $X = 6$ und $Y = 3$. Damit sind die Werte der Variablen eindeutig bestimmt. [Moz07, Finite Domain Constraint Programming in Oz, Chapter 2.3]

Die Propagierung stellt ein deterministisches Verfahren dar, welches allerdings nicht unbedingt vollständig ist. Das bedeutet, dass u.U. sowohl existierende Lösungen nicht gefunden werden, als auch die Nichtexistenz von Lösungen nicht erkannt wird [MS96].

2.1.2.2. Distribuierung

Um ein vollständiges Lösungsverfahren zu erhalten, muss zusätzlich die sog. Distribuierung eingesetzt werden. Dabei wird das Problem in Unterprobleme aufgeteilt; es entsteht ein Suchbaum, der beispielsweise auch verteilt durchsucht werden kann.

Sobald ein Problem P nicht mehr durch Propagierung gelöst werden kann, wird P so in P_1 und P_2 aufgeteilt, dass gilt: $P = P_1 \wedge P_2$. Diese Aufteilung nennt man Distribuierung. Es wird nun versucht, die beiden Teilprobleme in separaten Berechnungsräumen zu lösen. Dazu kann man oftmals einfach einen neuen Constraint C wählen, dessen Negation $\neg C$ ebenfalls einen Constraint darstellt, und mit diesen neuen Propagatoren in den beiden Zweigen des Baumes die Propagierung fortsetzen. Wann immer die Propagierung nicht weiterhilft, wird der Baum nach diesem Schema weiter aufgeteilt.

Beispiel Gegeben sei ein Constraintspeicher mit dem folgenden Inhalt:

$$X \in 1\#3 \wedge Y \in 2\#3 \wedge Z \in 1\#4$$

sowie zwei Propagatoren:

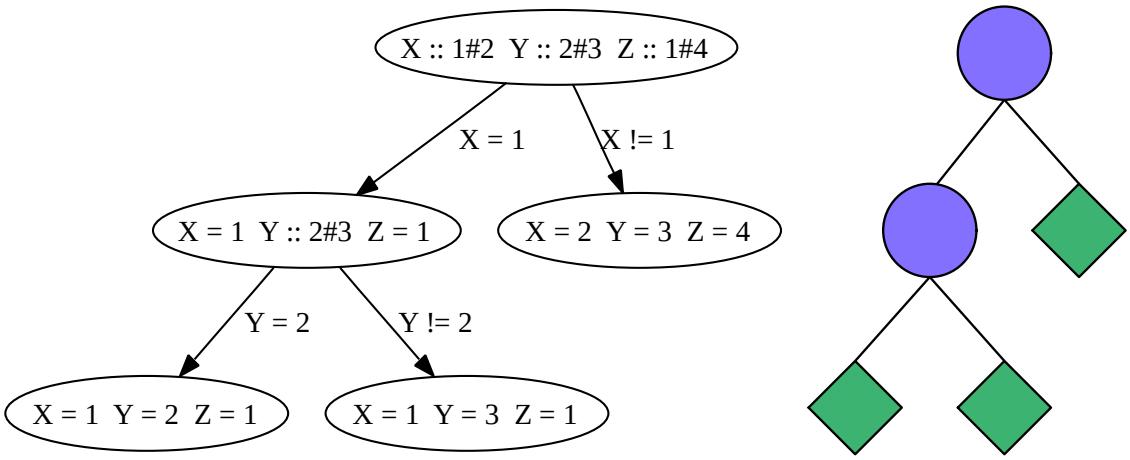
$$P_1 : X < Y, \qquad P_2 : X^2 = Z$$

Nun läuft zunächst die Propagierung ab:

1. P_1 schränkt ein: $X \in 1\#2$.

2. P_2 wird aktiv und schränkt ein: $Z \in 1\#4$.

Zu diesem Zeitpunkt bringt die Propagierung keine weiteren Erkenntnisse, es ist aber auch noch keine Lösung bestimmt worden. Durch Kopieren des Berechnungsraums wird nun der Suchbaum erstellt und im linken Teilbaum mit $C : X = 1$, im rechten mit $\neg C : X \neq 1$ weiter propagiert. Man erhält den in Abbildung 2.1 dargestellten Suchbaum - es ergeben sich schließlich drei Lösungen, markiert durch die grünen Rauten in 2.1b.



(a) Suchbaum mit Werten (b) Suchbaum als Graph

Abbildung 2.1.: Distribuierung

2.1.3. Beispiel „Send More Money“

Nachfolgend möchten wir anhand eines konkreten Beispiels die Wissensverarbeitung mithilfe der Constraint Programmierung in Oz demonstrieren. Entnommen ist diese Lösung des „Send More Money“-Problems der Mozart-Dokumentation [Moz07, Finite Domain Constraint Programming in Oz, Chapter 3.2].

2.1.3.1. Problemdefinition und Modell

Gegeben sei die Gleichung

$$SEND + MORE = MONEY$$

Ziel ist es, den einzelnen Buchstaben paarweise verschiedene Ziffern zwischen 0 und 9 zuzuweisen. Außerdem soll gelten: $S \neq 0$ und $M \neq 0$.

2.1.3.2. Programm

Listing 2.3 zeigt den Quellcode des fertigen Oz-Scripts.

```
1 declare Money
2 local
3    proc {Money Root}
```

```
4       S E N D M O R Y % Variablen deklarieren
5    in
6       Root = sol(s:S e:E n:N d:D m:M o:O r:R y:Y) % Datenstruktur
7       Root ::: 0#9                                % Domainconstraint
8       {FD.distinct Root}                          % paarweise versch.
9       S \=: 0                                     % S und M != 0
10      M \=: 0
11      1000*S + 100*E + 10*N + D +                 % Lsg. der Gleichung
12      1000*M + 100*O + 10*R + E =:
13      10000*M + 1000*O + 100*N + 10*E + Y
14      {FD.distribute ff Root}                     % Distribuierung
15   end
16 in
17   {Browse {SearchAll Money}}                     % Loesung berechnen
18   {ExploreOne Money}                             % Suchbaum anzeigen
19 end
```

Listing 2.3: Das „Send More Money“-Problem in Oz

Es werden zunächst für alle Buchstaben Variablen deklariert und anschließend ein `Record` angelegt, der für jeden Buchstaben einen Eintrag enthält (Zeile 6). In den nachfolgenden Zeilen werden die einzelnen Constraints deklariert:

- alle Einträge in `Root` sind beschränkt auf $(0, 9)$,
- alle Einträge in `Root` sind paarweise verschieden (distinct),
- S und M sind ungleich 0,
- die Werte der einzelnen Buchstaben erfüllen die Gleichung.

In Zeile 14 wird die Distribuierung mit der sog. „first-fail“-Strategie angestoßen. Bei dieser Strategie wird zur Erzeugung des zur Distribuierung benötigten Constraints die höchstwertige Variable verwendet, bei der die Anzahl möglicher Werte minimal ist.

Schließlich wird in Zeile 17 die Suche nach der Lösung begonnen (s. Kapitel 3.2.1) und in der darauf folgenden Zeile der Suchbaum ausgegeben.

2.1.4. Beispiel Sudoku

Anhand eines weiteren Beispiels möchten wir die Lösung von Problemen und die Wissensverarbeitung in Oz zeigen. Sudoku ist ein Logikrätsel, das sich in den letzten Jahren

großer Beliebtheit erfreut hat. In der Regel besteht ein Sudoku-Rätsel aus einem 9×9-Gitter, das mit einigen Zahlen vorbelegt ist. Ziel ist es, die freien Felder des Gitters so zu füllen, dass in der jeder *Reihen*, *Spalten* und in jedem der neun 3×3-*Unterquadrate* die Ziffern zwischen 1 und 9 genau einmal vorkommen.

Abbildung 2.2 zeigt ein Beispiel für ein Sudoku-Rätsel im Startzustand.

	1	2			9	6		
		7	3	5			9	
8				4			3	7
4		3	2				1	
	8		7	6		9		
9					3	8		4
	6				5	7		1
7		9	6		1			
	5			2			6	9

Abbildung 2.2.: Ein 9×9-Sudoku

2.1.4.1. Problemdefinition und Modell

Gegeben sei ein Sudoku Quadrat der Ordnung n, das ein $n^2 \times n^2$-Gitter mit n^4 Felder mit den Werten von 0 bis n^2 bildet [Sim05]. Jeder Wert, 0 bis n^2, darf in jeder Reihe, jeder Spalte und in jedem der n^2 $n \times n$-Unterquadrate genau einmal vorkommen.

Üblicherweise werden Sudokus der Ordnung 3 (9×9-Gitter) verwendet. Laut [Sim05] gibt es 6.670.903.752.021.072.936.960 komplett ausgefüllte und gültige Sudoku-Quadrate der Ordnung 3.

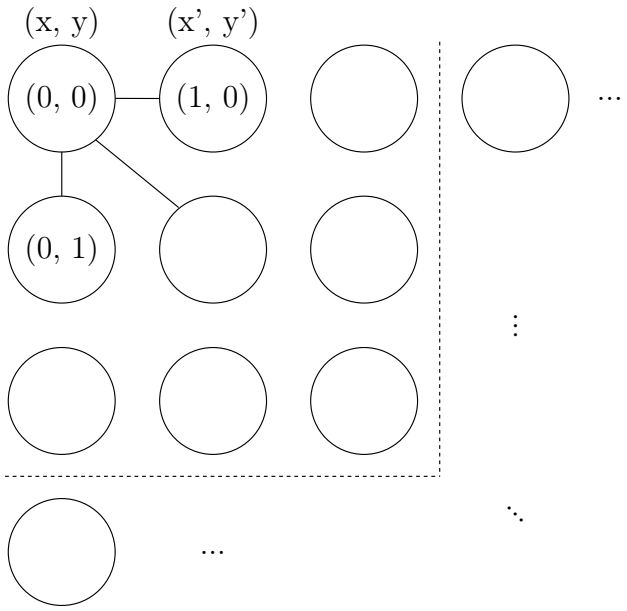


Abbildung 2.3.: Linker oberer Block eines Sudoku-Rätsels als Graph

Darstellen lässt sich das Sudoku-Problem (hier: 9×9-Sudoku) auch als Graphfärbeproblem, wie Abbildung 2.3 zeigt.

Aus diesem Graphen lässt sich direkt ableiten, wann ein Knoten mit einem anderen Knoten durch eine Kante verbunden ist. Dies ist genau dann der Fall, wenn

- $x = x'$, (Knoten liegen in derselben *Spalte*) oder
- $y = y'$, (Knoten liegen in derselben *Zeile*) oder
- $\lceil x/3 \rceil = \lceil x'/3 \rceil \wedge \lceil y/3 \rceil = \lceil y'/3 \rceil$ (Knoten liegen im selben 3×3-Block).

Das Problem besteht nun darin, dass benachbarte Knoten nicht den selben Ziffen (analog: Farben) zugeordnet werden dürfen. Es lassen sich also folgenden $3 \cdot 9 = 27$ Constraints ableiten:

- *distinct((0,0), (1,0), ..., (8,0))* (Zeile 1)
- *distinct((0,1), (1,1), ..., (8,1))* (Zeile 2)
- ...

- *distinct((0,0), (0,1), ..., (0,8))* (Spalte 1)
- *distinct((1,0), (1,1), ..., (1,8))* (Spalte 2)
- ...
- *distinct((0,0), (1,0), ..., (2,2))* (Block 1)
- *distinct((3,0), (4,0), ..., (2,5))* (Block 2)
- ...

Es ergibt sich also für jede Zeile, jede Spalte und jede Block-Zelle ein Constraint.

2.1.4.2. Programm

Listing 2.4 zeigt unser erstelltes Programm zur Lösung eines 3×3-Sudokus.

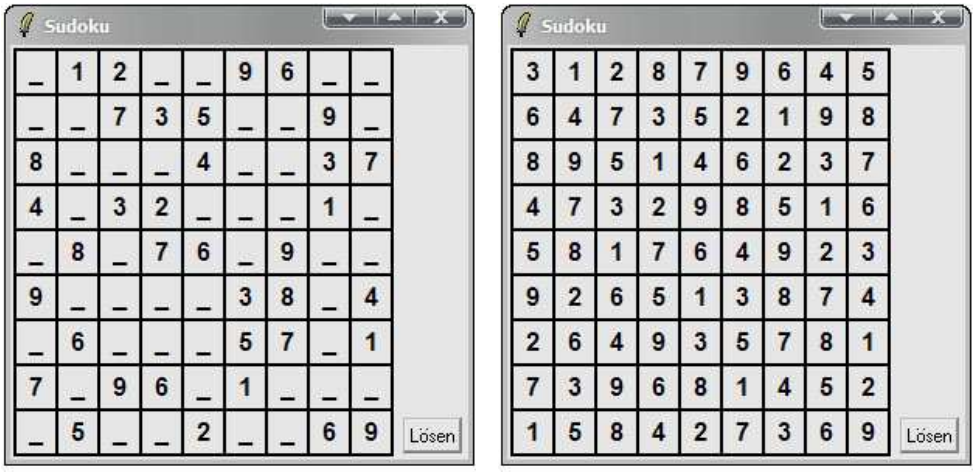
```
1  proc {ErstelleFeld Feld}
2     % Fuer jede Zelle eine Variable
3     Feld = feld(a0:A0 a1:A1 a2:A2 a3:A3 a4:A4 a5:A5 a6:A6 a7:A7 a8:A8
4           b0:B0 b1:B1 b2:B2 b3:B3 b4:B4 b5:B5 b6:B6 b7:B7 b8:B8
5           c0:C0 c1:C1 c2:C2 c3:C3 c4:C4 c5:C5 c6:C6 c7:C7 c8:C8
6           d0:D0 d1:D1 d2:D2 d3:D3 d4:D4 d5:D5 d6:D6 d7:D7 d8:D8
7           e0:E0 e1:E1 e2:E2 e3:E3 e4:E4 e5:E5 e6:E6 e7:E7 e8:E8
8           f0:F0 f1:F1 f2:F2 f3:F3 f4:F4 f5:F5 f6:F6 f7:F7 f8:F8
9           g0:G0 g1:G1 g2:G2 g3:G3 g4:G4 g5:G5 g6:G6 g7:G7 g8:G8
10          h0:H0 h1:H1 h2:H2 h3:H3 h4:H4 h5:H5 h6:H6 h7:H7 h8:H8
11          i0:I0 i1:I1 i2:I2 i3:I3 i4:I4 i5:I5 i6:I6 i7:I7 i8:I8)
12    Feld ::: 1#9                % Basis-Constraint: Werte zwischen 1 und 9
13    A1 =: 1 A2 =: 2 A5 =: 9 A6 =: 6 % Vorbelegung Zeile 1
14    B2 =: 7 B3 =: 3 B4 =: 5 B7 =: 9 % Vorbelegung Zeile 2
15    C0 =: 8 C4 =: 4 C7 =: 3 C8 =: 7 % ...
16    D0 =: 4 D2 =: 3 D3 =: 2 D7 =: 1
17    E1 =: 8 E3 =: 7 E4 =: 6 E6 =: 9
18    F0 =: 9 F5 =: 3 F6 =: 8 F8 =: 4
19    G1 =: 6 G5 =: 5 G6 =: 7 G8 =: 1
20    H0 =: 7 H2 =: 9 H3 =: 6 H5 =: 1
21    I1 =: 5 I4 =: 2 I7 =: 6 I8 =: 9
22 end
23
24 declare Feld Sudoku
```

```
25 proc {Sudoku Feld Loesung}
26    Loesung = Feld
27    {FD.distinct [A0 A1 A2 A3 A4 A5 A6 A7 A8]} % Constraints Zeile 1
28    {FD.distinct [A0 B0 C0 D0 E0 F0 G0 H0 I0]} % Spalte 1
29    {FD.distinct [A0 A1 A2 B0 B1 B2 C0 C1 C2]} % Block 1
30    {FD.distinct [B0 B1 B2 B3 B4 B5 B6 B7 B8]} % Zeile 2
31    {FD.distinct [A1 B1 C1 D1 E1 F1 G1 H1 I1]} % Spalte 2
32    {FD.distinct [A3 A4 A5 B3 B4 B5 C3 C4 C5]} % Block 2
33    {FD.distinct [C0 C1 C2 C3 C4 C5 C6 C7 C8]} % Zeile 3
34    {FD.distinct [A2 B2 C2 D2 E2 F2 G2 H2 I2]} % Spalte 3
35    {FD.distinct [A6 A7 A8 B6 B7 B8 C6 C7 C8]} % Block 3
36    {FD.distinct [D0 D1 D2 D3 D4 D5 D6 D7 D8]} % Zeile 4
37    {FD.distinct [A3 B3 C3 D3 E3 F3 G3 H3 I3]} % Spalte 4
38    {FD.distinct [D0 D1 D2 E0 E1 E2 F0 F1 F2]} % Block 4
39    {FD.distinct [E0 E1 E2 E3 E4 E5 E6 E7 E8]} % Zeile 5
40    {FD.distinct [A4 B4 C4 D4 E4 F4 G4 H4 I4]} % Spalte 5
41    {FD.distinct [D3 D4 D5 E3 E4 E5 F3 F4 F5]} % Block 5
42    {FD.distinct [F0 F1 F2 F3 F4 F5 F6 F7 F8]} % Zeile 6
43    {FD.distinct [A5 B5 C5 D5 E5 F5 G5 H5 I5]} % Spalte 6
44    {FD.distinct [D6 D7 D8 E6 E7 E8 F6 F7 F8]} % Block 6
45    {FD.distinct [G0 G1 G2 G3 G4 G5 G6 G7 G8]} % Zeile 7
46    {FD.distinct [A6 B6 C6 D6 E6 F6 G6 H6 I6]} % Spalte 7
47    {FD.distinct [G0 G1 G2 H0 H1 H2 I0 I1 I2]} % Block 7
48    {FD.distinct [H0 H1 H2 H3 H4 H5 H6 H7 H8]} % Zeile 8
49    {FD.distinct [A7 B7 C7 D7 E7 F7 G7 H7 I7]} % Spalte 8
50    {FD.distinct [G3 G4 G5 H3 H4 H5 I3 I4 I5]} % Block 8
51    {FD.distinct [I0 I1 I2 I3 I4 I5 I6 I7 I8]} % Zeile 9
52    {FD.distinct [A8 B8 C8 D8 E8 F8 G8 H8 I8]} % Spalte 9
53    {FD.distinct [G6 G7 G8 H6 H7 H8 I6 I7 I8]} % Block 9
54 end
```

Listing 2.4: Lösen eines Sudoku-Rätsels mit Mozart

Wir bilden zunächst in der Prozedur `ErstelleFeld` mithilfe eines Records die 81 Zellen in einer Datenstruktur ab, um diesen Variablen später die Constraints zuweisen zu können. Außerdem wird dort die Vorbelegung des Feldes getätigt, wir verwenden hier das Beispiel aus Abbildung 2.2.

Der Prozedur `Sudoku` wird ein Feld übergeben und es wird anschließend durch Erstellen der erwähnten Constraints die Lösung des Rätsels ermittelt. Dieses wird in der Variablen `Loesung` abgelegt. Wie man sieht, wird für die Lösung dieses Sudokus keine Distribuierung benötigt, da sich die Lösung allein durch die Auswertung der Constraints



(a) Ungelöstes Sudoku (b) Gelöst

Abbildung 2.4.: Oberfläche des Sudoku-Lösers

ergibt! Schließlich können wir die Lösung graphisch ausgeben. Aus Platzgründen ist dieser Teil des Quellcodes hier nicht aufgeführt, er findet sich im Anhang in Listing A.1. Abbildung 2.4 zeigt die Oberfläche des Sudoku-Lösers, mit dem in Abbildung 2.2 vorgestellten Beispiel für ein Sudoku. Das Sudoku kann über einen Klick auf den Button „Lösen“ gelöst werden. Das korrekt ausgefüllte Sudoku ist in Abbildung 2.4b zu sehen.

2.2. Funktionale Programmierung

Die funktionale Programmierung fußt auf der Auswertung (mathematischer) Ausdrücke. Funktionen verhalten sich hierbei - im Gegensatz zu anderen, nicht-funktionalen Programmiersprachen - wie mathematische Funktionen. Dies hat u.a. den Vorteil, dass Programme sich nun mit mathematischen Beweisverfahren besser validieren lassen. Ein wichtiges Merkmal funktionaler Programme ist, dass anstatt Schleifen und Zuweisungen mit Rekursion gearbeitet wird.

Oz unterstützt sowohl die sog. *eager evaluation* als auch *lazy evaluation*. Eager evaluation (strikte Auswertung) bedeutet, dass Argumente einer Funktion vor Ausführung der eigentlichen Funktion ausgewertet werden. Im Gegensatz dazu werden Argumente bei der lazy evaluation erst bei Bedarf ausgewertet, also dann, wenn deren Wert in einer Rechenoperation benötigt wird.

Listing 2.5 zeigt ein einfaches Beispiel für funktionale Programmierung in Oz.

```
declare
fun {Map List Function}
   case List of nil then nil
   [] Head|Tail then {Function Head}|{Map Tail Function}
   end
end

{Browse {Map [1 2 3 4] fun {$ X} (X+X)*2 end}}
```

Listing 2.5: Funktionales Programm mit eager evaluation

Der Funktion `Map` wird eine Liste sowie eine Funktion übergeben; die Funktion wird dann rekursiv auf alle Listenelemente angewandt. In Zeile 9 wird `Map` beispielhaft mit der Liste `[1 2 3 4]` und einer anonym definierten Funktion aufgerufen.

2.3. Objektorientierte Programmierung

Mit Einführung der Programmiersprache Smalltalk hat die objektorientierte Programmierung weite Verbreitung gefunden und ist heutzutage das in der Wirtschaft am häufigst eingesetzten Programmierparadigma. Objekte kapseln (veränderlichen) *Zustand* und *Verhalten*. Besonders in Anwendungen, die mit externen Systemen oder Anwendern interagieren eignen sich Objekte gut, um die Problemstellung imperativ zu formulieren und zu lösen [MS96].

In Oz lassen sich Klassen mit Attributen und Methoden deklarieren und anschließend Objekte dieser Klassen erzeugen. Weiterhin kann der Anwender seine Klassenhierarchie mit Mehrfachvererbung aufbauen. Listing 2.6 zeigt die Deklaration einer Klasse `Counter` sowie einige Methodenaufrufe.

```
declare Counter
class Counter
   attr val
   meth browse
      {Browse @val}
   end
   meth inc(Value)
      val := @val + Value
   end
   meth init(Value)
      val := Value
   end
```

```
13 end
14
15 declare C in
16 C = {New Counter init(23)}
17 {C inc(1)}
18 {C browse}
```

Listing 2.6: Deklaration einer Klasse und Erzeugen eines Objekts

In Zeile 16 wird ein Objekt der Klasse `Counter` erzeugt und der Variablen `C` zugewiesen. Die Zeilen 17 und 18 zeigen Aufrufe der Methoden `inc()` bzw. `browse()`.

2.4. Logische Programmierung

Mithilfe von logischen Programmiersprachen wie z.B. Prolog lassen sich Probleme lösen, für die kein (effizienter) Algorithmus bekannt ist, man also nach der Lösung *suchen* muss. Diese Art von Problem taucht bspw. in der Künstlichen Intelligenz, der Sprachverarbeitung oder der Unternehmensforschung (Operation Research) auf [Moz07, Tutorial of Oz, Chapter 12].

Oz erweitert die Ideen von Prolog und setzt dabei nicht auf die von Prolog bekannten Horn-Klauseln, sondern verwendet eine einfache Kompositions-Syntax höherer Ordnung [VBD+03].

Mithilfe der logischen Programmierung lassen sich aber nicht nur die genannten Suchprobleme lösen, sondern auch herkömmliche algorithmisch lösbare Probleme. Listing 2.7 zeigt zunächst ein *deterministisches* Programm, d.h. es ist keine Suche nötig.

```
1 declare
2 fun {Append Xs Ys}
3    case Xs of nil then Ys
4    [] X|Xr then X|{Append Xr Ys} end
5 end
6
7 declare X Y in
8 {Browse {Append [1 23 abc] X}}
```

Listing 2.7: Deterministisches logisches Programm

Die Funktion `Append` soll den übergebenen Parameter `Ys` an `Xs` anhängen und das Ergebnis zurückliefern. In Zeile 8 wird in dem beispielhaften Aufruf `X` an die Liste `[1 23 abc]` angehängt.

Im Gegensatz dazu wird bei der *nicht-deterministischen* Programmierung davon ausgegangen, dass man nur durch Suche die bzw. alle Lösungen finden kann. In Oz lässt sich so etwas mithilfe des `choice`-Statements realisieren, wie Listing 2.8 zeigt.

```
1  declare
2  proc {Append Xs Ys Zs}
3     choice
4        Xs = nil
5        Zs = Ys
6     [] X Xr Zr in
7        Xs = X|Xr
8        Zs = X|Zr
9        {Append Xr Ys Zr}
10    end
11 end
12
13 declare Anfrage Suche in
14 proc {Anfrage Loesung} X Y
15 in
16    {Append X Y [1 2 3 4 5]}
17    Loesung = sol(X Y)
18 end
19
20 Suche = {New Search.object script(Anfrage)}
21 {Browse {Suche next($)}}
```

Listing 2.8: Nicht-deterministisches logisches Programm. Quelle: [VBD+03]

Hier wird zunächst wieder die Prozedur `Append` definiert, dieses mal jedoch wird ein sog. *choice-point* festgelegt. Anschließend wird eine weitere Prozedur `Anfrage` deklariert, welche die Suchanfrage beinhaltet und das Ergebnis zurückliefert. Die eigentliche Suche wird über die Klasse `Search` (s. auch Kapitel 3.2.1) verwaltet. Über die Methode `next()` (Zeile 21) lässt sich die jeweils nächste Lösung ermitteln - auf diese Weise können sukzessive alle Lösungen bestimmt werden.

3. System Module

Neben der Implementierung der Basisumgebung für Oz bietet das Mozart Programming System auch so genannte *System Module*, die ein effektiveres und effizienteres Entwickeln von Anwendungen ermöglichen. Die Module lassen sich in mehrere Einsatzgebiete gliedern, die im Folgenden erläutert werden.

3.1. Application Programming

Für das *Application Programming* finden sich zwei Module, die das Arbeiten mit Applikationen und das Verwenden der vorhandenen Module erleichtern.

`Application` Das Modul `Application` ermöglicht den Zugriff auf die Argumente einer Anwendung und bietet Prozeduren um Anwendungen zu beenden. Der Zugriff auf die Argumente ist vergleichbar mit `Strings[] args` in Java. Es stehen 3 verschiedene Argumentarten für Web-, Kommandozeilen- und GUI-Anwendungen zur Verfügung. Außerdem können mit Hilfe dieses Moduls die Argumente auf unterschiedliche Arten geparst und verarbeitet werden.

`Module` Für den Zugriff, das Linken und Laden von Modulen ist das Modul `Module` zuständig. Hier kann auf vorhandene Module, wie z.B. `Application`, zugriffen werden um Prozeduren zu verwenden. Außerdem können eigene Oz-Anwendungen als Module geladen werden und später dann gelinkt werden.

3.2. Constraint Programming

Die Module, die in den Bereich des *Constraint Programming* fallen, stellen einen der wichtigsten Teile für die Erweiterung von Oz dar. Hier finden sich Module, die die Semantik und Syntax für das Constraint Programming erweitern. Beispielsweise wird die Semantik durch das Modul `FD` um *Finite Set Constraints* erweitert, eine Erweiterung auf Mengen von Constraints.

Ein kurzer Überblick aller Module für das Constraint Programming,

`Search` Dieses Modul bietet Unterstützung für die *Suche* von Lösungen, näheres siehe Punkt 3.2.1.

`FD` Für den leichteren Umgang mit *Finite Domain Constraints* finden sich in diesem Module beispielsweise vordefinierte Propagatoren, siehe Punkt 3.2.2.

`Schedule` Das Modul `Schedule` unterstützt die Entwicklung von Scheduling-Anwendungen, in denen Tasks auf gegebene Ressourcen verteilt werden, so dass keine zeitlichen Überlappungen auftreten. In [Moz07, System Modules, Part II] wird diese Vermeidung von zeitlichen Überlappungen als *Serialization for Unary Resources* bezeichnet. Außerdem bietet das Modul Propagatoren und Verteiler um mehrere Tasks, die verschiedene Ressourcen benötigen, so zu verteilen, damit jede Ressource *serialized* ist. Zusätzlich wird auch noch das so genannte kumulative Scheduling unterstützt, hier dürfen die Kapazitäten der Ressourcen nicht überschritten werden.

`FS` Hinter diesem Modul verbergen sich *Finite Set Constraints*. Es handelt sich um eine neue Constraint-Art für Oz, die eine n-elementige Menge mit n finite Domain Variablen assoziiert. Mengen von Constraint Variablen erweisen sich, laut [Moz07, System Modules, Part II], bei kombinatorischer Suche und Natural Language Processing als sehr nützlich. Im Umfang des Moduls befinden sich Prozeduren, Propagatoren etc. für den einfachen Umgang mit Finite Set Constraints.

`RecordC` Um auch mit anderen Datentypen als Domain Variablen, Intervall von Integern, auf den Constraint Store zuzugreifen, bietet das Modul `RecordC` die Möglichkeit, Constraints vom Datentyp `Record` zu verwenden. Es werden Prozeduren angeboten um das *Telling* auf den Store für Records zu erweitern.

`Combinator` Damit Constraints miteinander kombiniert werden können, bietet dieses Modul Prozeduren zur Kombination der Constraints. Bei den Kombinatoren handelt es sich hauptsächlich um konditionale oder logische, wie z.B. `Combinator.'or'` oder `Combinator.'not'`. Hier handelt sich um eine direkte Erweiterung der Oz-Syntax, die ohne weiteren Aufwand verwendet werden kann, daher werden die Operatoren auch als String aufgerufen.

`Space` Mit Hilfe dieses Moduls können eigene *Inferenzmaschinen* erstellt werden. Es werden Prozeduren bereitgestellt, mit denen Spaces verwaltet werden können. Durch die eigene Verwaltung, wie das Erzeugen neues Spaces `Spaces.create` oder das Zusammenfügen von Spaces `Spaces.merge`, können für die Inferenzmaschine neue Schlussfolgerungen erreicht werden.

Im Folgenden wird näher auf die Module `Search` und `FD` eingegangen, da diese den meisten Einsatz beim Constraint Programming finden.

3.2.1. Modul `Search`

Für die Suche bietet das Modul 3 Möglichkeiten, die *Basis-Suchmaschinen*, die *Universellen Suchmaschinen* und die *Parallele Suche*. Allgemein ist zur Suche in Mozart zu sagen, dass nach einer bzw. der ersten möglichen Lösung, allen Lösungen und der besten Lösung gesucht werden kann. Für die Suche nach der besten Lösung muss noch eine Ordnung übergeben werden, die definiert, was eine gute oder schlechte Lösung ist.

3.2.1.1. Basis-Suchmaschinen

Um die Suche mit den Basis-Suchmaschinen zu ermöglichen, müssen in den Prozeduren, mit denen eine Lösung gefunden werden soll, Entscheidungspunkte gesetzt werden. Dieses Setzten der Punkte erfolgt über das Schlüsselwort `choice`, das nebenbei bemerkt aus dem Modul `Combinator` stammt.

Ein kleines Beispiel nach [VBD+03] zur Verdeutlichung. Hier sollen nach den Kinder eines gegebenen Vaters gesucht werden.

```
1  declare
2  proc {Father F C}
3     choice F=terach C=abraham
4     [] F=terach C=nachor
5     [] F=terach C=haran
6     [] F=abraham C=isaac
7     [] F=haran C=lot
8     [] F=haran C=milcah
9     [] F=haran C=yiscah
10    end
11 end
12
13 proc {ChildrenFun X Kids}
14    {Search.base.all proc {$ K} {Father X K} end Kids}
15 end
16
17 {Browse {ChildrenFun terach}}
```

Listing 3.1: Suche nach allen Lösungen

Der letzte Aufruf in Listing 3.1 stößt die Suche mit dem Vater „terach“ an. Durch den Aufruf `Search.base.all` wird nach allen möglichen Lösungen gesucht und als Ergebnis wird die Liste `[abraham nachor haran]` zurückgegeben.

3.2.1.2. Universelle Suchmaschinen

Mit den Universellen Suchmaschinen lässt sich eine parametrisierte Suche durchführen. Hierzu bietet das Modul die Möglichkeit des so genannten *Recomputation* [Moz07, System Modules, Part II, Chapter 4.2], die Suche zu stoppen und verschiedene Ausgabemodi zu verwenden.

Mit Hilfe des Recomputation können auch für sehr große Suchbäume, für die in der Regel nicht genügend Speicherplatz zur Verfügung steht, Lösungen gefunden werden. Bei der normalen Suche in Oz wird in jedem Distributionschritt eine neue Kopie des Spaces erstellt. Dadurch kann der Speicherbedarf bei sehr großen Suchbäumen schnell sehr groß werden. Um diesem Problem Abhilfe zu schaffen, kann mittels der Recomputation angegeben werden, dass nur alle n Distributionsschritte eine Kopie erstellt werden soll. Die Verringerung des Speicherbedarfs geht allerdings auf die Kosten der Zeit. Schlägt eine Suche im $n-1$-ten Schritt nach der letzten Kopie fehl, muss zur letzten Kopie zurück gesprungen werden und einige Suchpfadteile könnten mehrfach durchsucht werden.

3.2.1.3. Parallele Suche

Mit Hilfe der parallelen Suche soll die Exploration von Suchbäumen beschleunigt werden, siehe [Moz07, System Modules, Part II, Chapter 4.4]. Um die Suche zu parallelisieren, wird der Suchbaum in mehrer Untersuchbäume aufgeteilt und auf mehrere Oz-Prozesse verteilt. Die Verteilung auf mehrere Oz-Prozesse kann lokal auf einen PC laufen oder auch auf mehrere PCs im Netzwerk verteilt werden. Damit die Suche parallel laufen kann, stellt das Modul die Klasse `Search.parallel` bereit, die alle Aufgaben übernimmt.

3.2.2. Finite Domain Constraints: `FD`

Ein weiterer wichtiger Teil für das Constraint Programming stellt das Modul `FD` für *Finite Domain Constraints* dar. Das Modul definiert Prozeduren, Propagatoren und Distributions-Strategien für die Suche.

Die meisten hier definierten Prozeduren und Propagatoren können implizit in Oz verwendet werden. Beispielsweise bedarf es bei Prozeduren für das Telling auf den Constraintstore keinen expliziten Import des Moduls. Die definierten Propagatoren gliedern sich in generische, symbolische, und binäre (0/1 Propagatoren), vgl. [Moz07, System

Modules, Part II, Chapter 5]. Die generischen Propagatoren lassen Definitionen für den Wertebereich von Domain Variablen zu, z.B. kleiner gleich „=<:“, die auch implizit in Oz verwendet werden können.

Zusätzlich bietet das Modul *Reflection* und das so genannte *Watching*, nach [Moz07, System Modules, Part II, Chapter 5.5], für Domains (Domain Variablen) an. Bei Reflection kann z.B. die aktuelle untere oder obere Grenze der Domain Variable zurückgegeben werden, `reflect.min` bzw. `reflect. max`. Mittels des Watching können Domains beobachtet werden. Beispielsweise wird `true` bei `{FD.watch.size *Domain1 +Domain2 ?B}` zurück gegeben, wenn die Intervallgröße von `Domain1` kleiner wird als die Größe von `Domain2`.

3.3. Verteilte Programmierung

Für die Entwicklung von *verteilten Anwendungen* bietet das Mozart Programming System einen Satz von Modulen, die die Entwicklung erleichtern. Neben Modulen für den Verbindungsaufbau oder die Remote-Steuerung von Oz-Prozessen, finden sich auch Module für die Manipulation für URLs (Uniform Resource Locator) oder für statistische Analysen der Verteilung.

Ein Überblick aller verfügbaren Module für die verteilte Programmierung,

`Connection` Um in Oz eine Verbindung zwischen Oz-Prozessen herzustellen, werden so genannte *Tickets* [Moz07, System Modules, Part III] verwendet. Ein Ticket stellt einen String dar, der für den Verbindungsaufbau verschickt wird. Um eine Verbindung herzustellen, erzeugt ein Prozess, der so genannte *Server*, ein Ticket, das vom *Client* auf der anderen Seite geholt bzw. empfangen wird. Das Modul `Connection` unterscheidet hier zwei Verbindungstypen, nach [Moz07, System Modules, Part III]:

One-to-one Erlaubt nur Einzelverbindungen zwischen 2 Prozessen, mit einem einmal verwendbaren Ticket.

Many-to-one Erlaubt Verbindung von mehreren Clients zu einem Server mit dem gleichen Ticket.

`Remote` Mittels dieses Moduls können über das Netzwerk andere Oz-Prozesse gestartet werden. Diese neu erzeugten Prozesse können *remote* gesteuert werden. Von der Klasse `Remote.manager` wird auf einem Computer im Netzwerk eine Instanz erzeugt, der so genannte *Remote Modul Manager*, der es erlaubt remote den Prozess zu steuern und auf die Ressourcen des entfernten Computers zuzugreifen.

`URL` Das Modul `URL` dient zur Erzeugung und Manipulation von URLs, für das WWW und das Dateisystem. Durch die Prozedur `URL.make` wird eine URL erzeugt, die sich als Datenstruktur verwalten lässt, beispielsweise `{URL.make ‘‘http://www.mozartoz.org/home-1.1.0/share/FD.ozf‘‘}`. Das Modul kümmert sich um die systemspezifischen Seperatoren für Verzeichnisse, somit spielt es keine Rolle, ob Verzeichnisse durch \ oder / getrennt werden.

`Resolve` Damit schnell Dateien oder Unterverzeichnisse in einer URL gefunden und verarbeitet werden können, bietet das Modul `Resolve` einige Prozeduren. Wie der Name schon sagt, soll die übergebene URL nach verschiedenen Arten aufgelöst werden, z.B. kann mit `root` das Rootverzeichnis für alle übergebenen URLs gesetzt werden.

`Fault` Mit Hilfe dieses Moduls können Fehler bei Verteilung erkannt und behandelt werden. Anzumerken ist an dieser Stelle, dass dieses Modul laut der Dokumentation [Moz07, System Modules, Part III] im aktuellen Release (Version 1.3.2) noch nicht komplett fertig gestellt ist.

`Discovery` Bei der Lokation von Diensten (Oz-Servern) im Netzwerk hilft das Modul `Discovery`. Ein Oz-Server wird im einen Wert initialisiert, in der Regel mit einen Ticket, und wartet bis ein Client eine Anfrage startet. Sucht ein Client einen Dienst im Netzwerk sendet er einen Broadcast über das Netzwerk und wartet auf die Antwort des zugehörigen Servers. In der aktuellen Version 1.3.2 ist die Funktionalität nur für Linux- und Solaris-Netzwerke verfügbar, siehe [Moz07, System Modules, Part III].

`DPInint` Die so genannte Verteilungsschicht, nach [Moz07, System Modules, Part III], wird bei Mozart nur bei Bedarf dynamisch nachgeladen. Über das Modul `DPInit` können einige Parameter zur Ladezeit der Verteilungsschicht initialisiert werden. `IpPort` legt beispielsweise fest über welchen Port kommuniziert werden soll.

`DPStatistics` Das Modul `DPStatistics` unterstützt die statistische Analyse von verteilten Anwendungen, um möglicherweise einige Aspekte verbessern zu können. Mit Hilfe des Moduls können Logfiles in verschiedenen Ausgabemodi erstellt werden.

3.4. Open Programming

In der Mozart Dokumentation [Moz07, System Modules, Part IV] werden diese Module als Verbindung zum Rest der *Computational World* beschrieben. Mit Hilfe dieser Module

erhält man Zugriff auf Dateien und Zugang zum Betriebssystem.

`Open` Mit diesem Modul ermöglicht Mozart den Zugriff auf Dateien (`Open.file`), Sockets (`Open.socket`) und Pipes (`Open.pipe`). Wie üblich können Dateien geöffnet, eingelesen, geschrieben und wieder geschlossen werden. Mit Sockets können Verbindungen zum Netzwerk über TCP oder UDP geöffnet und verarbeitet werden. Über die Klasse `Open.pipe` können Prozesse über die übergebene PID (Process ID) *geforkt* werden, um Daten mit diesem Prozess auszutauschen. Zusätzlich beinhaltet das Modul die Klasse `Open.text`, die in Verbindung mit den anderen Klassen verwendet werden kann um Text buchstaben- oder zeilenweise einzulesen. Außerdem werden Exceptions von diesem Modul ausgelöst, falls Probleme beim Zugriff auftreten, z.B. *already closed*.

`OS` Die Prozeduren des Moduls `OS` sind laut [Moz07, System Modules, Part IV] zu POSIX (Portable Operating System Interface) kompatibel. Hauptsächlich handelt es sich um Prozeduren für die Interaktion mit dem Betriebssystem, z.B. das Erzeugen von Zufallszahlen (`OS.rand`) oder Zugriff auf das Dateisystem (`OS.tmpnam`, absoluter Pfad zu einer neu erzeugten Datei). Außerdem erhält man auch Zugriff auf Verzeichnisse, beispielsweise liefert `OS.getDir` alle Pfade zu Dateien im gegebenen Verzeichnis. Zusätzlich bietet das Modul Zugriff auf Sockets, die Systemzeit und Umgebungsvariablen. Bei den so genannten *Low Level Procedures* können Angaben über die Lese- oder Schreibberechtigung gemacht werden, z.B. read only usw. Zu diesen Prozeduren zählen auch Prozeduren, wie `OS.kill`, für die Prozess-Steuerung, näheres siehe [Moz07, System Modules, Part IV].

3.5. System Programmierung

In das Gebiet der System Programmierung fallen Module, die die *Funktionalität* das Mozart Programming System betreffen. Im Folgendem einen Überblick dieser Module,

`Pickle` Das Modul `Pickle` dient dazu, zustandslose Werte zu speichern, also um Peristenz herzustellen. Die Werte können unter einem Dateinamen gespeichert und über diesen wieder geladen werden, z.B. komprimierte Speicherung `Pickle.saveCompressed`.

`Property` Um die Parameter der *Mozart Engine* (Virtual Maschine) abzufragen und zu aktualisieren, dient dieses Modul. Hierzu wird eine Schnittstelle mit 3 Prozeduren bereitgestellt. Zum Setzten von *Properties* die Prozedur `Property.put` und

für den Zugriff auf die Parameter `Property.get` (nur Lesen des Parameters) und `Property.condGet`. Die Einstellungen können beispielsweise die Distribuierung, die Spaces oder den Speichernutzung betreffen, näheres siehe [Moz07, System Modules, Part V].

`Error` Damit Fehlermeldungen, in Form von Exceptions, leserlich gestaltet werden können, bietet das Modul `Error` die Möglichkeit die Fehler zu formatieren. So genannte *Error Formatters* können erstellt werden, die Aussage über einen speziellen Fehler geben können. Außerdem bietet das Modul Prozeduren zur Fehlerbehandlung in einen Fehlerfall, z.B. `Error.printException`.

`ErrorFormatters` Mozart stellt im Modul `ErrorFormatters` schon einige vordefinierte Fehlermeldungen bereit, die von `Error`-Modul verwendet werden und auch selbst verwendet werden können. Beispielsweise formatiert `os` Exceptions und Fehler, die von der Schnittstelle zum Betriebssystem hoch kommen.

`Finalize` Mit Hilfe dieses Moduls können in Functors oder Klassen gekapselte Daten (Ressourcen) automatisch wieder freigegeben werden, falls der Functor oder die Klasse nicht mehr existiert. Hierfür wird in [Moz07, System Modules, Part V] ein Beispiel für eine Datenbankverbindung, die in einer Klasse gekapselt ist, vorgestellt. Falls die Klasse nicht mehr benötigt wird, wird die Instanz durch die Garbage Collection zerstört. Bevor allerdings die Instanz gelöscht wird, sollte die Datenbankverbindung geschlossen werden, diese Aufgabe übernimmt das Modul `Finalize`.

`System` Im Modul `System` werden Prozeduren zusammengefasst, die in Verbindung mit der Mozart Engine stehen. Die meisten Prozeduren beziehen sich auf die formatierte Ausgabe von beispielsweise Fehler oder einfachen Strings.

3.6. Window Programming

Wie bei jeder höheren Programmiersprache stellt Mozart Oz Module für die Oberflächenprogrammierung bereit. Zur Programmierung nutzt Mozart eine objektorientierte Schnittstelle zu *Tk* (Toolkit). Ursprünglich entstand Tk mit der Skriptsprache Tcl [1] (Tool Command Language) und stellt ein plattformunabhängiges *Graphical User Interface Toolkit* dar, siehe [Tcl07].

[1] `http://www.tcl.tk/`

`Tk` Dieses Modul stellt die Schnittstelle zu Tk dar. Hier können Fenster (`Tk.frame`), so genannte *Widgets* oder Buttons (`Tk.button`) zu einer Benutzeroberfläche zusammengestellt werden. Außerdem können Bilder zu Darstellung geladen werden. Zusätzlich bietet das Modul noch so genannte *Listeners*, die den Listener des Java-Oberflächenkonzepts ähneln, um die Oberfläche über bestimmte Ereignisse zu benachrichtigen, siehe [Moz07, System Modules, Part VI].

`TkTools` Um die Erstellung von Oberflächen zu erleichtern, wird das Modul `TkTools` von Mozart bereitgestellt. In diesem Modul finden sich vordefinierte Oberflächenkomponenten, wie Dialoge (`TkTools.dialog`) oder Menüleisten (`TkTools.menubar`), die einfach in der GUI verwendet werden können.

4. Projekte und aktuelle Entwicklungen

Schließlich noch 2 kurze Beispiele was in Form von Projekten rund um Mozart und Oz passiert. Zunächst wird ein sehr interessantes Projekt, *Strasheela*, vorgestellt, das sich mit dem regelbasierten Komponieren von Musik beschäftigt. Außerdem wird kurz auf einen Versuch eingegangen, bei dem Mozart und Oz in die Eclipse IDE integriert werden soll.

4.1. Strasheela

Das von Torsten Anders veröffentliche *Strasheela*[1] stellt ein in Mozart/Oz implementiertes System dar, mit dem Constraintbasiert Musik erzeugt werden kann. Der Benutzer gibt deklarativ, in Form von Oz-Code, eine Musiktheorie an. Oz versucht durch Suche die definierten Constraints zu terminieren. Um diese Lösungen auch als Musik zu hören, bietet Strasheela die Möglichkeit die Lösung als MIDI-Datei zu exportieren.

Ein Beispiel, das Strasheela mitliefert, ist sind die so genannten *All-Interval Series*. Diese Musiktheorie stammt aus dem Feld der seriellen Musik (serial Musik), siehe [Fri07]. Eine Serie von Tönen besteht in diesem Beispiel aus 12 Tönen einer Tonleiter, die für jede Serie nur einmal verwendet werden dürfen. Außerdem müssen die Abstände zwischen den einzelnen Tönen eindeutig sein, d.h. pro Serie darf jeder Abstand (Intervall) nur einmal vorkommen. Laut [And07] gibt es für dieses Problem 3856 Lösungen, die von Mozart alle gefunden werden, siehe Abbildung 4.1.

4.2. MozEclipse

Im Februar diesen Jahres startete Craig Ugoretz das Projekt *MozEclipse*[2]. Ziel dieses Projekts ist es, Mozart und Oz in die Eclipse IDE zu integrieren. Durch die Integration

[1] `http://strasheela.sourceforge.net/strasheela/doc/`
[2] `http://gforge.info.ucl.ac.be/projects/mozeclipse/`

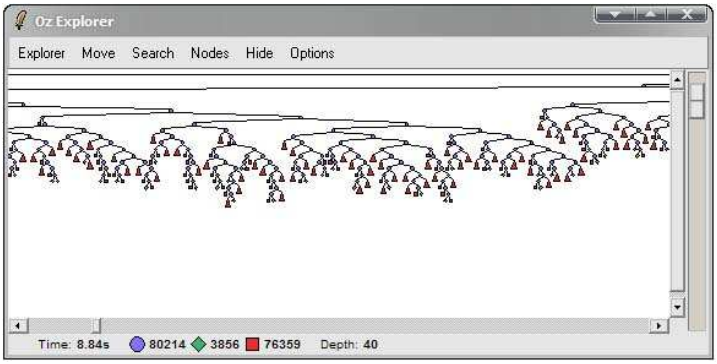


Abbildung 4.1.: Suche nach allen Lösungen von *All-Interval Series*, grüne Raute

soll der Bekanntheitsgrad von Mozart/Oz steigen. Momentan ist Mozart in den Emacs integriert, der vor allem Anfängern einige Schwierigkeiten bereitet. Durch die Integration in Eclipse sollen die Hemmungen beseitigt werden, da viele Anwender gut mit Eclipse vertraut sind. Leider ist die Zukunft dieses Projekts noch sehr ungewiss, da es seit dem Anlegen der Projekt-Homepage keine weiteren Aktivitäten gab.

A. Quellcode Sudoku-Löser (GUI)

```
local
   N = 3
   Window = {New Tk.toplevel tkInit(title:'Sudoku' width:200 height:200)}
   Mainframe = {New Tk.frame tkInit(parent:Window
           relief:solid
           borderwidth:1)}
   Startbutton = {New Tk.button tkInit(parent:Window
              text:'START'
              action: proc {$}
             Feld = {Sudoku Feld}
                {List.forAllInd {Record.toList Feld}
                 proc {$ Index Wert}
              Row = (Index-1) div (N*N)
              Col = (Index-1) mod (N*N)
                 in
              {Tk.batch [grid(row:Row
                  column:Col
                  {New Tk.label tkInit(parent:Mainframe
                          text:Wert
                          relief:solid
                          borderwidth:1
                          padx:8
                          pady:4
                          font:{New Tk.font tkInit(family:helvetica
                                 weight:bold
                                 size:12)}
                         )
                  }
                 )
                  ]}
                 end}
                end
```

```
33              )}
34      Feld = {ErstelleFeld}
35  in
36      {Tk.send pack(Mainframe side:left padx:2 pady:2)}
37      {Tk.send pack(Startbutton side:bottom padx:1#m pady:1#m)}
38      {List.forAllInd {Record.toList Feld}
39       proc {$ Index Wert}
40          Row = (Index-1) div (N*N)
41          Col = (Index-1) mod (N*N)
42          DisplayString
43       in
44          if ({FD.reflect.size Wert} == 1)
45          then DisplayString = {Int.toString Wert}
46          else DisplayString = '_' end
47          {Tk.batch [grid(row:Row
48              column:Col
49              {New Tk.label tkInit(parent:Mainframe
50                 text:DisplayString
51                 relief:solid
52                 borderwidth:1
53                 padx:8
54                 pady:4
55                 font:{New Tk.font tkInit(family:helvetica
56                       weight:bold
57                       size:12)}
58               )
59              }
60             )
61       ]}
62       end}
63  end
```

Listing A.1: Graphische Ausgabe der Sudoku-Lösung

Quellen

[And07] Anders, Torsten: *Strasheela*. `http://strasheela.sourceforge.net/strasheela/doc/`. Version: 2007

[Bru00] Brunklaus, Thorsten: *Der Oz Inspector - Browsen: Interaktiver, einfacher, effizienter*, Universität des Saarlandes, Diplomarbeit, Februar 2000

[Fri07] Frisius, Rudolf: *Serielle Musik*. `http://www.frisius.de/rudolf/texte/tx317.htm`. Version: 2007

[Moz07] Mozart Consortium: *Mozart Documentation*. `http://www.mozart-oz.org/documentation/`. Version: 2007

[MS96] Müller, Martin ; Smolka, Gert: Oz: Nebenläufige Programmierung mit Constraints. In: *KI - Künstliche Intelligenz* (1996), September, S. 55–61. – Themenheft: Logische Programmierung

[Sim05] Simonis, Helmut: Sudoku as a Constraint Problem / Imperial College London. 2005. – Forschungsbericht

[Tcl07] Tcl Core Team: *Tcl Developer Site*. `http://www.tcl.tk/`. Version: 2007

[VBD+03] Van Roy, Peter ; Brand, Per ; Duchier, Denys ; Haridi, Seif ; Henz, Martin ; Schulte, Christian: Logic programming in the context of multiparadigm programming: the Oz experience. In: *Theory and Practice of Logic Programming* (2003). – To appear.