Fachbereich 14 Informatik
Universität des Saarlandes

Der Oz Inspector – Browsen: Interaktiver, einfacher, effizienter

Diplomarbeit

Angefertigt unter der Leitung von Prof. Dr. Gert Smolka

Thorsten Brunklaus

29.02.2000

Leitung:	Prof. Dr. Gert Smolka
Erstgutachter:	Prof. Dr. Gert Smolka
Zweitgutachter:	Prof. Dr. Reinhard Wilhelm
Betreuung:	Dipl.-Inform. Christian Schulte

Erklärung

Hiermit erkläre ich, daß ich die vorliegende Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Saarbrücken, den 29.02.2000

Thorsten Brunklaus

Zusammenfassung

Diese Arbeit beschreibt Konzept, Entwurf und Implementierung des Inspectors. Der Inspector ist ein interaktives, grafisches Werkzeug zur Darstellung von Oz-Datenstrukturen.

Oz-Datenstrukturen sind komplex und deren Darstellung erfordert ein grafisches Werkzeug. Aus Sicht des Benutzers muß ein solches vor allem sehr effizient, flexibel und interaktiv sein.

In dieser Arbeit wird ein System vorgestellt, daß diese Anforderungen durch einen zweistufigen Ansatz erfüllt. Dieser besteht darin, neben effizienten Basisdiensten einen flexiblen Transformationsmechanismus einzusetzen.

Die vorgestellte Implementierung ist hochmodular und sehr kompakt. Deren Effizienz wird schließlich durch Vergleich mit einem ähnlichen System demonstriert.

Danksagung

Mein Dank gilt Prof. Dr. Gert Smolka, der mir dieses interessante Thema angeboten und mich in vielen Fragestellungen unterstützt hat.

Insbesondere danke ich Christian Schulte für seine konstruktive und effiziente Hilfestellung in allen Belangen der Arbeit.

Nicht zuletzt möchte ich die Mitarbeiter des Lehrstuhls erwähnen, die mir für die Beantwortung vieler Fragen stets bereitwillig zur Verfügung standen.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 1

1.2 Konzepte des Inspectors 1

1.3 Aufbau 2

2 Die Sprache Oz **3**

2.1 Konzepte hinter Oz 3

2.1.1 Logische Variablen 3

2.2 Oz-Werte 4

2.2.1 Einfache Werte 4

2.2.2 Zusammengesetze Werte 4

2.2.3 Chunks 6

2.3 Constraint-Variablen 6

2.4 Eigenschaften für die Programmerstellung 6

3 Anforderungen **9**

3.1 Darstellungskriterien 9

3.1.1 Verfahren 9

3.1.2 Die Kriterien 10

3.2 Informationsextraktion 12

3.2.1 Selektive Darstellung 12

3.3 Transienten 13

3.3.1 Darstellung 13

3.3.2 Aktualisierung 13

3.4 Interaktion 15

3.5 Implementierungsanforderungen 15

4 Entwurf **17**

4.1 Architektur des Inspectors 17

4.2 Architektur der Darstellungseinheit 18

4.2.1 Vom Oz-Wert zur Darstellung 19

4.3 Strukturerzeugung 19

4.3.1 Datenmodell 20

4.3.2 Der Typtransformator 21

4.4 Transientenverwaltung 22

4.4.1 Registrierung 22

4.4.2 Aktualisierung 22

4.5 Strukturerkennung 24

4.5.1 Erzeugung 24

4.5.2 Referenzaktualisierung 25

4.6 Anordnungsverfahren 26

4.6.1 Anordnungsberechnung 26

4.6.2 Ein Problemfall 26

4.7 Der Zeichendienst 27

4.7.1 Markierte Bäume 27

4.7.2 Der Zeichenprozeß 27

4.8 Interaktion 27

4.8.1 Interaktionspunkte 28

4.8.2 Interaktionsmedium 28

5 Implementierung **29**

5.1 Aufbau der Implementierung 29

5.1.1 Module 29

5.1.2 Nachrichtenbehandlung 30

5.1.3 Ausführungsmodell 30

5.2 Oz-Werte 31

5.2.1 Einfache Knoten 31

5.2.2 Containerknoten 32

5.3 Die Darstellungseinheit 34

5.3.1 Basisdienste 35

5.3.2 Transientenverwaltung 35

5.4 Die Grafikeinheit 36

5.4.1 Verwendetes System 36
5.4.2 Architektur 37
5.4.3 Dienste 37

6 Evaluierung **39**
6.1 Durchführung 39
6.2 Geschwindigkeit 40
6.2.1 Tupel 40
6.2.2 Listen 40
6.2.3 Records 41
6.3 Aktualisierungsverhalten 42
6.3.1 Lokale Aktualisierung 42
6.3.2 Exponentielles Wachstum 42
6.3.3 Viele Gleiche Transientendarstellungen 43
6.3.4 Ein schlechter Fall 44
6.4 Zusammenfassung 45

7 Verwandte Arbeiten **47**
7.1 Der Oz-Browser 47
7.1.1 Anordnungsverfahren 47
7.1.2 Filtersysteme 48
7.1.3 Interaktionsverhalten 48
7.1.4 Konfigurierbarkeit 48
7.2 Ein Pretty-Printer-Generator 48
7.3 Abstrakte Graphsysteme 49
7.3.1 Prinzipieller Aufbau 49
7.3.2 Konfigurierbarkeit 49

8 Zusammenfassung **51**
8.1 Zentrale Ideen 51
8.2 Implementierung 52
8.3 Evaluierung 52
8.4 Ausblick 52

Kapitel 1

Einleitung

Diese Arbeit beschreibt Konzept, Entwurf und Implementierung des Inspectors. Der Inspector ist ein interaktives, grafisches Darstellungswerkzeug für die Datenstrukturen von Oz.

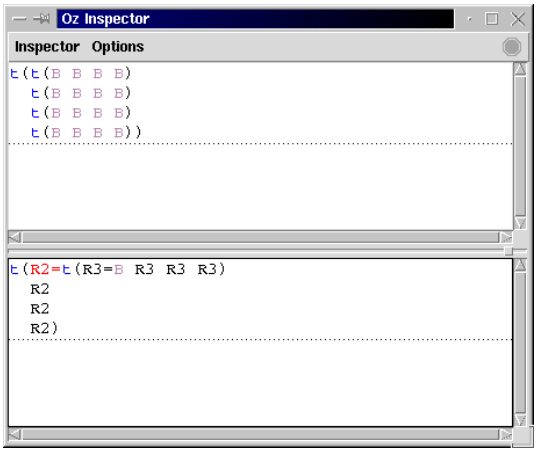
1.1 Motivation

Oz-Datenstrukturen sind in der Regel komplex. Sie können sehr groß sein und Zyklen enthalten. Insbesondere gibt es Werte mit partieller Information. Das heißt, deren Wert ist nur teilweise bekannt und kann später verfeinert werden.

Aus diesem Grund erfordert eine verständliche Darstellung von Oz-Werten ein grafisches Werkzeug wie den Inspector.

Das Hauptinteresse des Benutzers erstreckt sich dabei über die folgenden Punkte:

1. Gute Lesbarkeit der Darstellung.
2. Effizienz.
3. Flexibilität.
4. Interaktion.



1.2 Konzepte des Inspectors

Idee. Der Inspector verwendet zur Erzeugung der Darstellung einen zweistufigen Ansatz. Dieser zweistufige Ansatz besteht darin, wenige, aber dafür sehr effiziente Basisdienste zu verwenden. Darauf aufbauend, wird ein flexibler Transformationsmechanismus eingesetzt.

Darstellung. Die Basisdienste des Inspectors halten wichtige Darstellungskriterien ein. Dadurch wird gute Lesbarkeit der Darstellung garantiert. Entscheidend dabei ist die Teilbaumisomorphie. Das heißt, gleiche Teilbäume werden stets gleich dargestellt. Hinzu kommt das Stabilitätskriterium für die Aktualisierungsmechanismen. Beide zusammen bilden die Grundlage für die Inkrementalität und damit für die Geschwindigkeit des Inspectors.

Flexibilität. Durch den Transformationsmechanismus werden viele Darstellungsentscheidungen von der Implementierung weg auf den Benutzer übertragen. Er legt fest, was wie dargestellt wird. Daraus resultiert eine kompakte und zugleich flexible Implementierung.

Die vorliegende Implementierung realisiert bereits sinnvolle Vorgaben für den Transformationsmechanismus. Dadurch ist dieser für den Benutzer auch ohne spezielle Kenntnisse sofort einsetzbar.

Interaktion. Der Benutzer kann direkt mit der Darstellung des Inspectors interagieren. Die Darstellung läßt sich so schnell und flexibel den aktuellen Wünschen anpassen. Darüber hinaus kann man verschiedene Sichten auf die gleichen Daten erhalten.

Modularität. Die Implementierung des Inspectors ist hochmodular und faktorisiert. Modularität ist eine wichtige Eigenschaft in bezug auf die Erweiterbarkeit beziehungsweise Anpassbarkeit der Implementierung.

1.3 Aufbau

Das nachfolgende Kapitel stellt die für die Arbeit relevanten Aspekte der Programmiersprache Oz vor.

Daran schließt sich der Kern der Arbeit an, der insgesamt vier Kapitel umfaßt:

1. **Anforderungen** an den Inspector. Es wird im Detail diskutiert, welche Eigenschaften der Inspector haben muß.
2. **Entwurf** des Inspectors. In diesem Kapitel wird ein anforderungskonformes System vorgestellt.
3. **Implementierung** des Inspectors. In diesem Kapitel wird der Aufbau der Implementierung sowie interessante Details dazu vorgestellt.
4. **Evaluierung** der Implementierung. Diese erfolgt im Vergleich mit dem Browser [24]. Der Browser ist ebenfalls ein grafisches Darstellungswerkzeug für die Datenstrukturen von Oz.

Neben einer Betrachtung verwandter Arbeiten, folgt abschließend die Diskussion, in der das Erreichte noch einmal zusammengefaßt und bewertet wird.

Kapitel 2

Die Sprache Oz

Oz spielt für diese Arbeit zwei Rollen: Der Inspector stellt die Datenstrukturen von Oz dar und wird selbst in Oz implementiert. Daher stehen in diesem Kapitel vor allem die Datenstrukturen und die programmiersprachlichen Konzepte von Oz im Vordergrund.

2.1 Konzepte hinter Oz

Oz ist eine nebenläufige Programmiersprache, die funktionale, objektorientierte und Constraint-Programmierung unterstützt [30, 31].

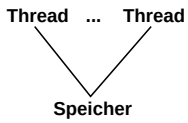


Abbildung 2.1: *Ein Berechnungsraum*

Berechnung ist in Oz nebenläufig in Berechungsräumen organisiert (siehe Abbildung 2.1). Ein Berechnungsraum besteht aus einer Anzahl von Threads, die über einem gemeinsamen Speicher, dem *Constraint*-Speicher, ausgeführt werden.

Der Constraint-Speicher bildet Variablen auf Werte ab und wächst monoton. Das heißt, neue Information muß konsistent zur bereits vorhandenen Information bleiben. Der Versuch, inkonsistente Information zu speichern, ergibt einen Fehler.

2.1.1 Logische Variablen

Für frisch eingeführte Variablen liegen im Constraint-Speicher zunächst keine Informationen vor. Daher erlauben Variablen die schrittweise Konstruktion von Werten. Dies geschieht, indem frische Variablen als Teilwerte verwendet werden. Die vorhandene Information kann nun so lange verfeinert werden, bis der Wert vollständig ist. Zum Beispiel:

```
declare X Y Z
```

```
X = Y|Z %% Listenelement bauen
Y = Z   %% Y und Z sind gleich
Y = 5   %% Y (und implizit Z) werden gebunden

{Show X} --> 5|5
```

Daneben kennt Oz eine Art Nur-Lese-Variable, die *Futures*. Durch eine Future kann man eine logische Variable kapseln. Die gekapselte Variable kann durch die Future dann zwar gelesen, nicht aber gebunden werden.

Logische Variablen und Futures werden auch Transienten genannt.

2.2 Oz-Werte

Oz kennt einfache und zusammengesetzte Werte. Letztere sind rekursiv aus Teilwerten aufgebaut. Eine Aufzählung aller Werte und Typen findet man in [4]. An dieser Stelle soll nur ein Überblick gegeben werden.

2.2.1 Einfache Werte

Zahlen. Oz verfügt über zwei disjunkte Mengen von Zahlen, nämlich ganze Zahlen und Fließkommazahlen. In der aktuellen Implementierung von Oz, Mozart [4], können ganze Zahlen beliebig groß werden, während Fließkommazahlen in ihrer Größe beschränkt sind.

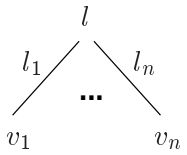
Literale. Die Menge der Literale besteht aus *Atomen* und *Namen*. Atome werden durch Zeichenketten in einfachen Anführungsstrichen denotiert, die auch wegfallen können, sofern der erste Buchstabe ein Kleinbuchstabe ist, z.B. `'Sample'`, `oz_atom`.

Namen sind eindeutige Bezeichner, die durch die Generatorfunktion `NewName` eingeführt werden und keine anderweitige Notation besitzen. Die speziellen Namen `true`, `false` und `unit` sind bereits vordefiniert.

Prozeduren. In Oz sind Prozeduren voll emanzipiert, können also zur Laufzeit erzeugt und an Variablen gebunden werden. Diese Eigenschaft wird sich die Implementierung des Inspectors zunutze machen.

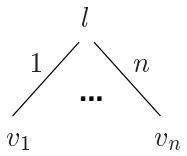
2.2.2 Zusammengesetze Werte

Records. Ein *Record* ist entweder ein Literal oder ein ungeordneter Baum der Form



mit l Literal, $l_1, \ldots, l_n$ paarweise verschiedene Literale oder Zahlen, $v_1, \ldots, v_n$ Werte und $n > 0$. l heißt dann Label, die Werte v_i Felder und n die Weite des Records.

Tupel. Ein Tupel ist ein Spezialfall von Records, nämlich ein geordneter Baum der Form



mit l Literal, $v_1, \ldots, v_n$ Werte und $n > 0$.

Listen. Listen sind spezielle Tupel, die induktiv definiert sind:

- Das Atom `nil` ist eine Liste.
- Ist k ein Wert und s eine Liste, so ist das Tupel `'|'(`k s`)` eine Liste. k heißt dann Kopf und s Schwanz.

Je nachdem, ob der Schwanz s des letzten Elements einer Liste l das Atom `nil` oder eine Variable ist, nennt man l eine *echte* oder *offene* Liste.

Echte Listen werden in einer Kurzform dargestellt: `[1 2 3]` steht also für `'|'(1 '|'(2 '|'(3 nil)))`.

Offene Listen erlauben die Formulierung potentiell unendlicher Datenstrukturen wie etwa Ströme, indem das Ende der Liste stets ein neues offenes Endstück erhält.

Listen, deren Elemente aus Zeichencodes bestehen, sind Strings. Strings besitzen eine alternative Schreibweise, zum Beispiel `"Hallo"` anstelle von `[72 97 108 108 111]`.

Andere Varianten. Ein weiteres spezielles Tupel entsteht durch Verwendung des Labels `#`, das ein Mixfix-Operator ist. Die Formen `1#2#3` beziehungsweise `'#'(1 2 3)` sind demnach gleichwertig. In der Darstellung wird jedoch die Infix-Notation verwendet.

Ähnlich wie bei Listen, lassen sich durch Verwendung des `#`-Operators sogenannte virtuelle Strings bauen. Diese erlauben die flexible Kombination von Strings, ganzen Zahlen, Fließkommazahlen und Atomen. Virtuelle Strings besitzen im Browser oder Inspector auch eine alternative Darstellung: `"Nummer"#5` kann als `"Nummer 5"` dargestellt werden.

2.2.3 Chunks

Chunks erlauben die Konstruktion abstrakter Datentypen, also solchen, die über private Felder verfügen.

In der aktuellen Implementierung sind bereits einige abstrakte Datentypen vordefiniert. Diese verhalten sich nach außen wie ein einfacher Wert, da alle Felder gekapselt sind. Dafür bieten sie aber eine definierte Schnittstelle in Form von Prozeduren. Beispiele sind etwa *Dictionary* und *Array*.

2.3 Constraint-Variablen

In Oz lassen sich kombinatorische Probleme mit Hilfe von Constraint-Programmierung effizient lösen. Dazu wird eine gegenüber logischen Variablen erweiterte Variablenform verwendet, die Constraint-Variablen. Diese tragen Information, die angibt, welchen Wert die Variable noch annehmen kann. Für unterschiedliche Probleme gibt es dazu drei Arten: Finite-Domain-, Finite-Set- und Feature-Variablen.

Finite-Domain-Variablen erlauben das Einschränken des möglichen Wertes auf eine Menge ganzer Zahlen. Diese Menge kann durch weitere Berechnung weiter eingeschränkt werden. Angenommen, im Constraint-Speicher befindet sich die Information $X \in \{1, 3, 5, 10\}$. Durch `X \=: 5` wird die Information auf $X \in \{1, 3, 10\}$ präzisiert.

Finite-Set- und Feature-Variablen verhalten sich ähnlich und werden an dieser Stelle nicht näher behandelt. Details dazu finden sich in [30, 31, 4].

2.4 Eigenschaften für die Programmerstellung

Explizite Nebenläufigkeit. In Oz kann man Berechnungen explizit in eigene Threads abspalten.

Implizite Synchronisation. Berechnung in Oz schreitet durch schrittweise Reduktion von Threads voran. Dies geschieht, solange dazu genügend Informationen im Constraint-Speicher vorliegen. Kann ein Ausdruck nicht reduziert werden, wird die Berechnung automatisch so lange gestoppt, bis wieder genügend Information vorliegt.

Klassen und Objekte. Neben der Möglichkeit, Prozeduren zur Abstraktion zu benutzen, kann man in Oz auch objektorientiert programmieren. Dazu stehen Klassen und deren Instanzen, die Objekte, zur Verfügung.

Wichtig ist, daß Objekte Zustand enkapsulieren und Nachrichten an Objekte voll emanzipiert (engl. *first-class*) sind.

Serverkonstruktion. In bezug auf nebenläufige Architektur sind aktive Objekte besonders interessant. Aktive Objekte sind nebenläufige Berechnungsabstraktionen, die über einen Port erhaltene Nachrichten selbsttätig ausführen.

Von diesen wird die Implementierung des Inspectors Gebrauch machen, um möglichst einfach zu bleiben.

Kapitel 3

Anforderungen

Der Inspector soll Oz-Werte effizient und gut lesbar grafisch darstellen. Darüber hinaus soll die Darstellung automatisch aktualisiert werden und flexibel in bezug auf Benutzervorgaben erfolgen.

Dieses Kapitel behandelt die sich daraus ergebenden Anforderungen:

- Es werden Kriterien vorgestellt, die eine gute Darstellung ausmachen.
- Es wird dargestellt, welche Einflußmöglichkeiten der Benutzer auf die Darstellung haben muß.
- Die effiziente Behandlung von Transienten wird diskutiert.

3.1 Darstellungskriterien

Bis auf wenige Ausnahmen, soll die Darstellung von Oz-Werten getreu der Syntax von Oz erzeugt werden. Denn diese ist für den Betrachter besonders einfach lesbar.

3.1.1 Verfahren

Es existieren eine Vielzahl von Algorithmen und Verfahren, Graphen darzustellen [1]. Die meisten Oz-Werte verfügen jedoch über eine textuelle Darstellung, die ein textbasiertes Anordnungsverfahren erlaubt. Die Struktur des Wertes wird dabei durch geeignete Einrückung und visuelle Attribute verdeutlicht. Deswegen spricht man auch von Pretty-Printing.

Die Darstellung der Datenstrukturen wird in einem Fenster auf dem Bildschirm erfolgen, dessen Größe beschränkt ist. In bezug auf Pretty-Printing ergeben sich daher zwei unterschiedliche Ansätze:

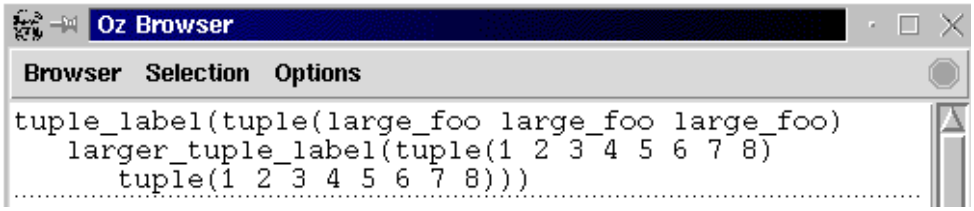
1. Darstellung unter möglichster Einhaltung einer maximalen Textbreite. Das heißt, die Werte werden so angeordnet, daß möglichst viel Text innerhalb der vorgegebenen Breite bleibt. Damit soll zu häufiges Verschieben des sichtbaren

Ausschnittes vermieden werden. Diese Vorgehensweise hat aber den Nachteil, daß die Darstellung von Werten positionsabhängig und deswegen manchmal unstrukturiert wird.

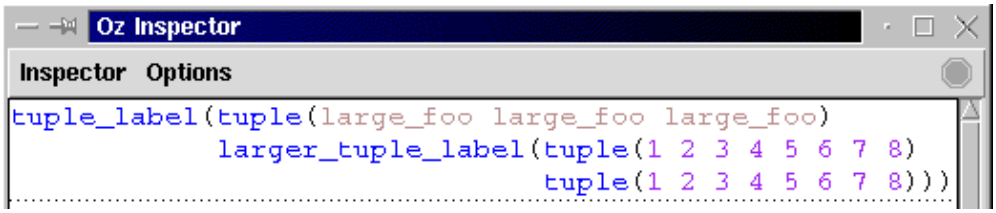
2. Darstellung ohne derartige Einschränkungen. Damit kann die Struktur des Wertes nach festen Kriterien verdeutlicht werden. Das Ergebnis ist zumeist eine bessere Qualität der Darstellung. Das setzt allerdings voraus, daß das Verschieben des Textausschnittes schnell und effizient möglich ist.

3.1.2 Die Kriterien

In den folgenden Paragraphen werden nun die einzelnen Kriterien vorgestellt, die eine gute Darstellung ausmachen. Im Vergleich zum Browser [24] wird deutlich, weswegen der Inspector den zweitgenannten Ansatz verwendet.



(a) Browser: Darstellung ohne Hierarchietransparenz

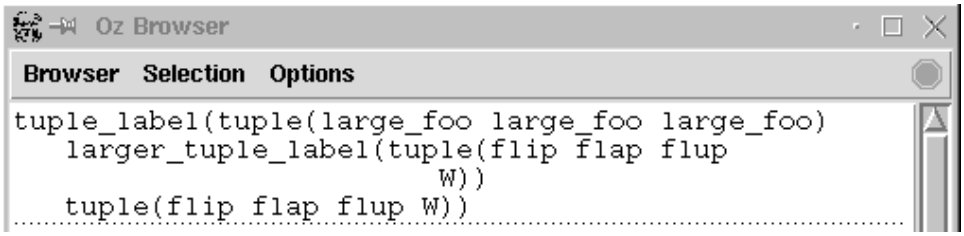


(b) Inspector: Darstellung mit Hierarchietransparenz

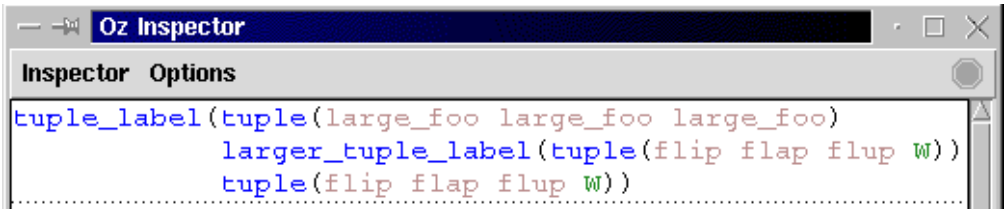
Abbildung 3.1: *Hierarchietransparenz im Vergleich*

Hierarchietransparenz. Oz-Datenstrukturen sind hierarchisch aufgebaut. Diese Hierarchie ist eine wichtige Eigenschaft, die deshalb durch die Darstellung hervorgehoben werden muß. Damit wird erreicht, daß auch große Strukturen leicht erfaßbar werden.

Abbildung 3.1 zeigt dazu ein Beispiel, welches den Browser und den Inspector gegenüberstellt. Die Darstellung des Browsers in Abbildung 3.1(a) ist zwar syntaktisch korrekt, aber schwieriger zu lesen: Die Eltern-Kind-Relationen ist infolge der schlechten Ausrichtung „verwischt“.



(a) Browser: Darstellung ohne Teilbaumisomorphie



(b) Inspector: Darstellung mit Teilbaumisomorphie

Abbildung 3.2: *Teilbaumisomorphie im Vergleich*

Teilbaumisomorphie. Das Anordnungsverfahren darf keine künstlichen Unterschiede in der Darstellung erzeugen: Gleiche Teilbäume müssen unabhängig von ihrer Position stets gleich angeordnet werden. Damit wird zum einen möglich, gleichartige Information in der Darstellung besser zu erkennen. Zum anderen bildet dies die Grundlage für inkrementelle Aktualisierungsverfahren (siehe Abschnitt 3.3.2).

Zur Verdeutlichung zeigt Abbildung 3.2 wieder einen Vergleich zwischen Browser und Inspector. Das Tupel `tuple(flip flap flup W)` ist beim Browser in Abbildung 3.2(a) erst nach genauerem Hinsehen zweimal als solches zu erkennen.

Visuelle Attribute. Eine wichtige Hilfe, die Struktur einer Darstellung zu verdeutlichen, sind typspezifische visuelle Attribute [5]. Für textbasierte Darstellung bietet sich dazu neben der Möglichkeit, verschiedene Zeichensätze zu verwenden, Farbe besonders an. Die Darstellung soll daher so erfolgen, daß jeder einfache Typ eine ihm zugeordnete Farbe hat. Dies gilt ebenfalls für die Hilfstypen, die notwendig sind, zusammengesetzte Typen darzustellen.

Farbwahrnehmung ist Geschmackssache und muß deswegen durch den Benutzer konfigurierbar sein.

Darstellungsform. Oz-Werte sind Graphen, für deren Darstellung sich zwei Möglichkeiten ergeben:

- Bei der Baumdarstellung werden rekursiv alle Teilstrukturen dargestellt, ohne auf Wiederholungen zu achten. Dadurch entsteht eine für den Betrachter

einfach verständliche Struktur, die im Einzelfall jedoch größer als nötig wird. Daneben lassen sich zyklische Strukturen nur unter besonderen Vorkehrungen darstellen (siehe Abschnitt 3.2.1).

- Eine andere Möglichkeit ist es, auf Besonderheiten in der Struktur zu achten. Insbesondere heißt das, die wiederholte Darstellung gleicher Strukturen zu unterbinden. Beim erneuten Antreffen wird dann anstatt des gesamten Wertes eine Referenz auf die Erstdarstellung dargestellt. Damit entsteht eine kompaktere Graphdarstellung, die zwar die Struktur des Wertes besser hervorhebt, aber schwerer zu lesen ist [16].

  Um eine Graphdarstellung zu erzeugen, ist ein Gleichheitstest notwendig. Auf Oz-Datenstrukturen sind zwei Gleichheiten definiert: Strukturelle und Token-Gleichheit. Je nach Situation, erweisen sich beide als sinnvoll für die Darstellung. Deswegen muß der Inspector dem Benutzer die Wahl lassen, welche Gleichheit verwendet wird.

Infolge der leichteren Lesbarkeit wird im Standardfall die Baumdarstellung verwendet und die Graphdarstellung optional verfügbar gemacht.

3.2 Informationsextraktion

Wird dem Werkzeug ein Oz-Wert zur Darstellung übergeben, will der Benutzer möglichst viel über den Wert in Erfahrung bringen. Für viele Datentypen genügt dazu, deren Struktur zu traversieren und die entsprechenden Teilwerte anzuzeigen. Große Datenstrukturen und abstrakte Datentypen verursachen mit dieser Strategie jedoch Probleme.

Bei großen Datenstrukturen kann es vorkommen, daß zuviele Informationen dargestellt werden: Der Benutzer ist häufig nur an einem Teil der Darstellung interessiert, die für ihn relevante Information enthält [16]. Er kann diese aber wegen der Größe der Restdarstellung nur schwer ausfindig machen.

Die normale Darstellung von abstrakten Datentypen ist problematisch. Denn diese sind so angelegt, daß die internen Informationen nur über eine wohl definierte Schnittstelle erreicht werden können. Im allgemeinen ist also nur ein Teil der intern vorhandenen Informationen sichtbar. Zu dieser Klasse von Datentypen gehören alle Chunks, insbesondere Klassen, Objekte, und Prozeduren.

Für viele Zwecke, zum Beispiel bei der schnellen Fehlersuche in Programmen, die ohne Debugger [18] erfolgen soll, ist das nicht ausreichend.

3.2.1 Selektive Darstellung

Um obige Probleme zu lösen, muß die Darstellung selektiv erfolgen. Der Benutzer legt fest, was wie dargestellt wird. Dazu sind zwei Mechanismen vorgesehen:

- Traversierungsfilter. Diese erlauben die Darstellung von Teilbäumen, indem

alle Teilbäume, die außerhalb festlegbarer Breiten- und Tiefenlimits liegen, weggeblendet werden.

Das Vorhandensein derartiger Filter ist die Voraussetzung dafür, zyklische Werte als Baum darstellen zu können.

- Baumtransformation: Die Darstellung eines Bildwertes unter einer Abbildung. Diese ermöglicht die gezielte Extraktion von Information insbesondere aus abstrakten Datentypen, aber auch aus großen Strukturen. Daneben kann der Benutzer flexibel alternative Darstellungen für einzelne Werte festlegen, zum Beispiel eine spezielle Darstellung für Listen, die einen String repräsentieren.

Für den Benutzer erweist sich die Kombination manueller und automatischer Anwendung als sinnvoll:

- Die manuelle Anwendung ermöglicht es, nach erfolgter Darstellung interaktiv eine Veränderung herbeizuführen, um so die gewünschte Sicht auf die Datenstruktur zu erhalten.
- Werden Datenstrukturen sehr groß, kann es sehr aufwendig sein, manuell eine Selektion einzuleiten. Die automatische Selektion wirkt daher bereits bei der Erzeugung der Datenstruktur und macht manuelles Eingreifen unnötig.

In beiden Fällen gilt, daß für den Benutzer eine selektive Darstellung klar erkennbar und der Effekt auf die Darstellung stets interaktiv umkehrbar sein muß.

3.3 Transienten

Oz bietet die Möglichkeit, mit unvollständigen Daten in Form von Transienten zu arbeiten (siehe Abschnitt 2.1.1). Wie mit diesen umzugehen ist, beschreibt dieser Abschnitt.

3.3.1 Darstellung

Die Darstellung von Transienten kann sich ändern und ist daher in der Regel für den Betrachter besonders interessant. Deshalb müssen diese derart dargestellt werden, daß sie eindeutig als partielle Information erkennbar sind.

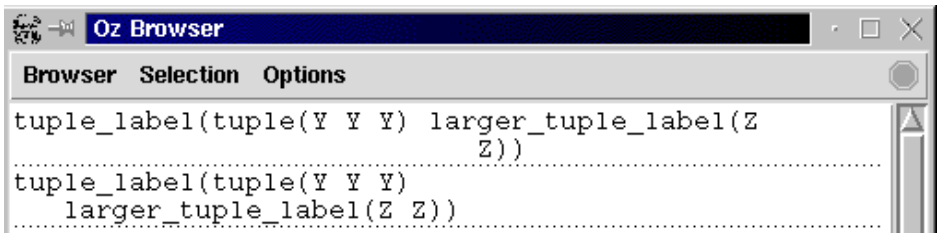
Für Constraint-Variablen kann die durch Reflektion gewonnene Information zur Darstellung verwendet werden. Für Transienten, die selbst keine Information tragen, funktioniert dies nicht so ohne weiteres. Für diese wird deswegen eine generische Darstellung in Form von _ verwendet.

3.3.2 Aktualisierung

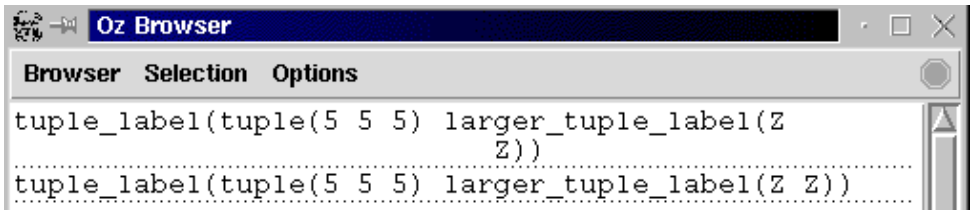
Das System muß in der Lage sein, neue Informationen für einen dargestellten Transienten zu erkennen. Die dazu notwendige Überwachung muß mit minimalen Aufwand erfolgen und insbesondere ereignisbasiert funktionieren.

Liegt für einen Transienten neue Information vor, muß die Darstellung aktualisiert werden. Die gesamte Darstellung des betroffenen Wertes neu zu erstellen, ist naiv. Statt dessen erfolgt die Aktualisierung inkrementell: Es werden nur die betroffenen Teilbäume ersetzt und die restliche Darstellung mit möglichst geringem Aufwand auf den neuesten Stand gebracht.

Die Grundlage dazu ist mit Einhaltung der in Abschnitt 3.1.2 vorgestellten Teilbaumisomorphie für die Darstellung bereits gegeben. Darüber hinaus darf sich das Erscheinungsbild des betroffenen Wertes nur sehr wenig und vorhersagbar ändern. Bleibt die Änderung zusätzlich fast immer auf die betroffenen Teilbäume beschränkt, ergibt sich daraus das sehr wichtige Stabilitätskriterium [10, 3, 16]: Die Orientierung des Betrachters bleibt damit auch nach der Änderung erhalten und die Effizienz wird maximiert, denn der Radius notwendiger Arbeiten über die betroffenen Teilbäume hinaus bleibt sehr klein.



(a) Darstellung vor Aktualisierung von `Y`

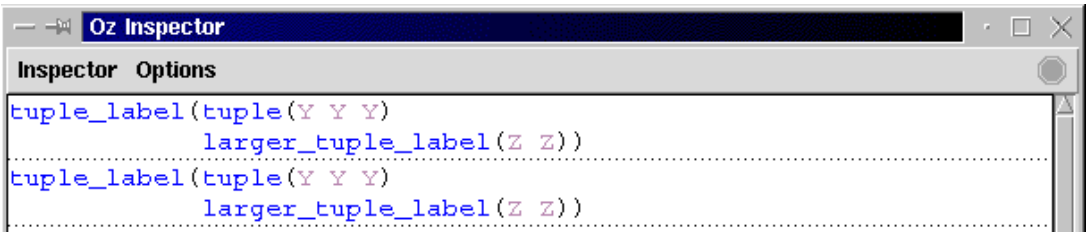


(b) Darstellung nach `Y = 5`

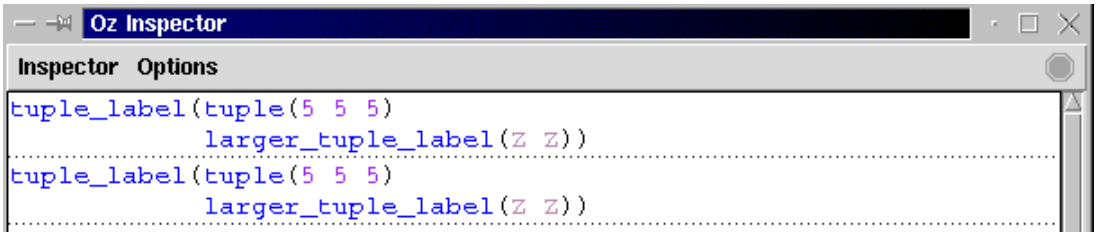
Abbildung 3.3: *Browser: Aktualisierung ohne Stabilitätskriterium*

Um zu verdeutlichen, warum das Stabilitätskriterium wichtig für große Strukturen ist, zeigt Abbildung 3.3 ein zwar kleines, aber aussagekräftiges Beispiel am Browser. Neben der Tatsache, daß in Abbildung 3.3(a) ein und der selbe Wert verschieden dargestellt wird, gilt dies auch für die durch `Y = 5` verursachte Aktualisierung. Wie Abbildung 3.3(b) zeigt, wird die erste Darstellung inkrementell aktualisiert, die zweite jedoch unerwartet auf eine Zeile ausgerichtet – und sieht danach aber immer noch anders aus als der gleiche Wert darüber. Erfolgt derartiges rekursiv in einer großen Struktur, wird das zum echten Problem für die Übersichtlichkeit.

Abbildung 3.4 zeigt den selben Vorgang mit Stabilitätskriterium.



(a) Darstellung vor Aktualisierung von `Y`



(b) Darstellung nach `Y = 5`

Abbildung 3.4: *Inspector: Aktualisierung mit Stabilitätskriterium*

3.4 Interaktion

Das Werkzeug soll eine interaktive Umgebung bieten, die dem Benutzer die leichte Erforschung der dargestellten Strukturen nach klassischer Versuch-und-Irrtum-Strategie erlaubt. Dies erfordert, daß keine allzu große Barriere zwischen der Darstellung und der Datenstruktur selbst besteht [8, 16].

Das Problem wird gelöst, indem der Benutzer direkt mit der Darstellung interagiert und diese dadurch seinen Wünschen anpassen kann. Die direkte Interaktion hat den Zweck, viele die Darstellung betreffende Aktionen lokal auslösen und wieder rückgängig machen zu können.

3.5 Implementierungsanforderungen

Der Inspector soll in Oz selbst programmiert werden. Dabei stehen neben der Ausführungseffizienz die folgenden Aspekte im Vordergrund:

Modularität Die Implementierung soll möglichst klar in funktionale Einheiten aufgeteilt werden. Dazu sind die gegebenen Abstraktionsmöglichkeiten von Oz, etwa Klassen und vor allem Module, zu verwenden. Module erlauben die nachträgliche Änderung einzelner Komponenten, ohne das ganze System neu übersetzen zu müssen. Damit kann der Inspector leicht an Änderungen in der Implementierung von Oz, Mozart [4], angepaßt werden.

Minimalität Die Implementierung soll in bezug auf den Codeumfang möglichst

kompakt sein. Dazu sind zwei Dinge notwendig: Die Verwendung generischer Konzepte und Faktorisierung, wo immer möglich.

Nebenläufigkeit Die Implementierung erfolgt nebenläufig. Das bedeutet, daß mehrere Threads unabhängig voneinander auf Dienste des Inspectors zugreifen können. Die Folge wäre, daß sich sich möglicherweise Schreibzugriffe auf die internen Datenstrukturen überlagern und so zu inkonsistenten Systemzuständen führen könnten. Dies muß durch Gegenseitigen Ausschluß verhindert werden: Kritische Bereiche des Systems werden synchronisiert, so daß jeweils nur ein Thread Zugriff auf die kritischen Bereiche des Inspectors erhält.

Wird feinkörnig synchronisiert, das heißt, gibt es mehrere kritische Bereiche, die unabhängig voneinander sychnronisiert sind, entsteht die Gefahr der Verklemmung. Zwei Threads versuchen dabei vergeblich, eine Freigabe für den jeweils anderen kritischen Bereich zu erhalten. Dies funktioniert jedoch nicht, da jeder Thread bereits den kritischen Bereich blockiert, den der jeweils andere Thread anfordert. Feinkörnige Synchronisation ist daher problematisch und sollte nur verwendet werden, wenn dies sicher gewährleistet werden kann.

Die Architektur des Inspectors muß in bezug auf Nebenläufigkeit möglichst einfach sein und Gegenseitigen Ausschluß ohne Verklemmung gewährleisten.

Kapitel 4

Entwurf

Dieses Kapitel beschreibt den Entwurf des Inspectors. Der Schwerpunkt der Betrachtungen liegt dabei auf dem Kern des Systems, der sogenannten Darstellungseinheit. Die Darstellungseinheit erzeugt die eigentliche Wertdarstellung und ist vollständig unabhängig vom Inspector, der diese lediglich als Modul einbindet.

4.1 Architektur des Inspectors

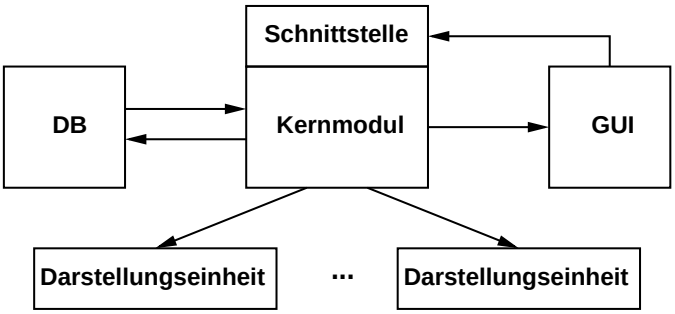


Abbildung 4.1: *Komponenten und Datenfluß des Inspectors*

Der Inspector (siehe Abbildung 4.1) ist so angelegt, daß er mehrere Darstellungseinheiten unabhängig voneinander verwalten kann. Dadurch wird es möglich, Datenstrukturen gleichzeitig aus verschiedenen Blickwinkeln zu betrachten und so maximale Übersicht zu erhalten.

Daneben sind weitere Module definiert, deren Details erst in Kapitel 5 beschrieben werden:

Schnittstelle Die Implementierung des Inspectors erfolgt nebenläufig. Um auf einfache Weise gegenseitigen Ausschluß ohne Verklemmung zuzusichern, wird global synchronisiert. Dies geschieht mittels einer Schnittstelle, über die alle Nachrichten von außen eintreffen. Unterhalb dieser ist keine weitere Sicherung notwendig.

Optionsmodul Das Optionsmodul (DB) enthält sinnvolle Parametervorgaben für den Inspector selbst, sowie für die Darstellungseinheiten.

Kernmodul Das Kernmodul verwaltet die verschiedenen Darstellungseinheiten und bearbeitet alle Nachrichten, die über die Schnittstelle eintreffen.

GUI Das GUI-Modul enthält die Funktionalität, die erforderlich ist, Optionen interaktiv per grafischem Dialog einstellen zu können.

4.2 Architektur der Darstellungseinheit

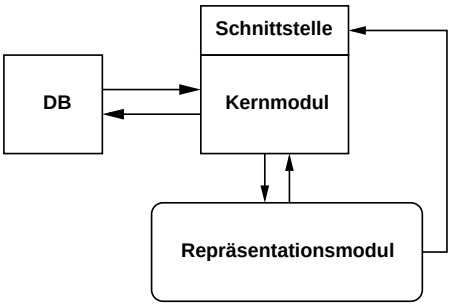


Abbildung 4.2: *Architektur der Darstellungseinheit*

Abbildung 4.2 zeigt die Komponenten der Darstellungseinheit, sowie den Datenfluß zwischen diesen. Im einzelnen lassen sich folgende Komponenten ausmachen:

Schnittstelle Die Darstellungseinheit ist vollkommen unabhängig vom Inspector und wird ebenfalls nebenläufig implementiert. Da auch hier gegenseitiger Ausschluß ohne Verklemmung zugesichert werden muß, wird ebenfalls global synchronisiert. Alle Nachrichten von außen treffen über die Schnittstelle ein und werden an das Kernmodul zur Bearbeitung weitergeleitet.

Die Anzahl derartiger Nachrichten ist gegenüber dem Aufwand ihrer Bearbeitung vergleichsweise gering. Deswegen können die Kosten für die Sicherung etwas teurer sein, denn unterhalb der Barriere ist keine weitere Sicherung notwendig.

Optionsmodul Das Optionsmodul (DB) implementiert die Parametervorgaben für das System. Alle Systemkomponenten fragen die für sie relevanten Optionen über das Kernmodul ab.

Kernmodul Das Kernmodul ist die „Kommandozentrale“, von der aus die einzelnen Aktionen gesteuert werden. Alle Nachrichten, die über die Schnittstelle eintreffen, werden durch dieses Modul bearbeitet.

Repräsentationsmodul Das Repräsentationsmodul ist aus mehreren Untermodulen zusammengesetzt, die die Darstellungserzeugung und -verwaltung realisieren.

4.2.1 Vom Oz-Wert zur Darstellung

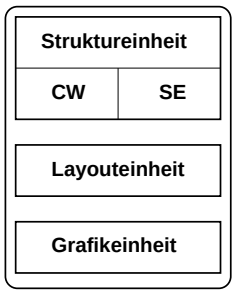


Abbildung 4.3: *Aufbau des Repräsentationsmoduls*

Der Aufbau des Repäsentationsmoduls (siehe Abbildung 4.3) verdeutlicht die notwendigen Dienste, um einen Oz-Wert darzustellen:

1. **Strukturaufbau:** Für einen Oz-Wert muß zunächst eine datenstrukturelle Repräsentation erzeugt werden, die im folgenden Knoten genannt wird. Knoten werden durch die Struktureinheit mit Hilfe zweier Untermodule, dem Constraint-Wächter (CW) und dem Strukturerkennungsmodul (SE), erzeugt. Ersteres erledigt zentral die Verwaltung und Überwachung von Transienten und löst im Bedarfsfall eine Aktualisierungsnachricht aus. Das Strukturerkennungsmodul wird für die Graphdarstellung benötigt. Es erkennt bereits dargestellte Strukturen und wandelt dementsprechend den Wert in einen Referenzknoten.

2. **Geometrische Anordnung** Die geometrische Anordnung (engl. *Layout*) der Datenstrukturen muß berechnet werden. Das dazu verwendete Verfahren muß die in Abschnitt 3.1.2 vorgestellten Kriterien einhalten und wird durch die Layouteinheit realisiert.

3. **Zeichnen** Die Struktur muß gemäß der berechneten Anordnung auf dem Bildschirm gezeichnet werden. Dies wird von der Grafikeinheit erledigt, die darüber hinaus GUI-Ereignisse über die Schnittstelle an das Kernmodul weiterreicht.

4.3 Strukturerzeugung

In Abschnitt 3.2.1 wird festgelegt, daß die Darstellungserzeugung selektiv und konfigurierbar erfolgen muß. Dies erfordert einen mehrstufigen Erzeugungsprozeß für Knoten, der in Abbildung 4.4 dargestellt ist.

Der Ablauf gestaltet sich wie folgt: Zunächst wird der Tiefenfilter durchlaufen, um sicherzustellen, daß eine angegebene maximale Tiefe nicht überschritten wird. Die Überschreitung der Grenztiefe führt zur Erzeugung eines speziellen Limitknotens. Dieser zeigt die künstliche Tiefenbegrenzung an und ist kein Element des Oz-Universums.

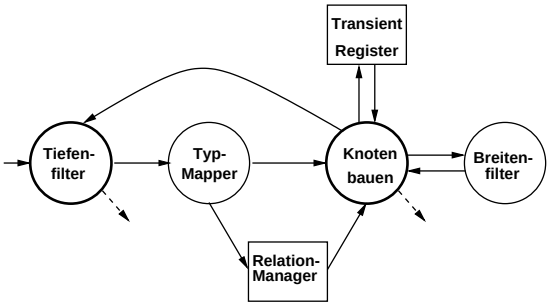


Abbildung 4.4: *Die Knotenerzeugung*

Anderenfalls wird der Oz-Wert unverändert an den Typtransformator (engl. *Mapper*) weitergereicht. Dieser prüft nun, ob der vorhandene Wert automatisch transformiert werden soll. Falls ja, wird der Ergebniswert der Transformation, ansonsten der ursprüngliche Wert weitergereicht.

Bevor nun der Knoten endlich erzeugt werden kann, erfolgt im Relationsmodus noch der Durchlauf durch den Relationsmanager. Dieser prüft unter Rückgriff auf eine benutzerdefinierbare Äquivalenzrelation, ob der Wert bereits bekannt ist. Falls ja, wird der vorhandene Wert durch einen Referenzwert ersetzt, dessen Knoten dann erstellt wird.

Für den Fall, daß es sich um einen zusammengesetzen Typ handelt, wird ein Containerknoten gebaut. Dieser erledigt eigenständig die Konstruktion der Kindknoten unter Beachtung des Breitenfilters, in dessen Limitfall erneut ein Limitknoten erzeugt wird.

4.3.1 Datenmodell

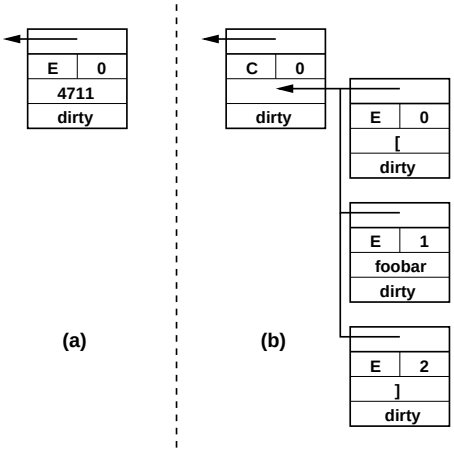


Abbildung 4.5: *Einfache (a) und zusammengesetzte (b) Knoten*

Die Werte von Oz sind induktiv definiert und lassen sich in einfache und zusammengesetzte Werte unterteilen (siehe Abschnitt 2.2.2). Dementsprechend werden einfache

Knoten und Containerknoten definiert. Letztere können eine beliebige Anzahl von Kindknoten verwalten, wie auf Abbildung 4.5 dargestellt.

Um eine Darstellung inkrementell aktualisieren zu können, müssen die Knoten zwei Eigenschaften erfüllen:

1. Wahlfreien Zugriff auf Kindknoten in auf- und absteigender Traversierung.

2. Markierung veränderter und unveränderter (Kind-)Knoten.

Deshalb verfügen alle Knoten über die folgenden Datenfelder:

Elternfeld Für aufsteigende Traversierung wird der jeweilige Elternknoten benötigt. Das Kernmodul ist dabei der Elternknoten für alle Knoten ohne direkten Vorgänger, die im folgenden Wurzelknoten genannt werden.

Indexfeld Das Indexfeld enthält die Position des Knoten in der Knotenliste des Elternknotens. Zusammen mit dem Elternknoten werden dadurch effiziente Teilbaumersetzungen möglich.

Typfeld Das Typfeld gibt den Typ des Knoten an; hier vereinfacht entweder `E` für einfacher Knoten oder `C` für Containerknoten. Dieser wird für Operationen benötigt, bei der zur Laufzeit der Typ eines Knotens ermittelt werden muß.

Repräsentationsfeld Das Repräsentationsfeld enthält bei einfachen Knoten die Stringrepräsentation des Wertes beziehungsweise die Liste der Kindknoten bei Containerknoten.

Traversierungsindikator Der Traversierungsindikator zeigt an, ob der Knoten oder einer seiner Kindknoten verändert (dirty) wurde oder nicht (clean). Durch diesen wird die in Abschnitt 3.3.2 geforderte Inkrementalität realisiert.

4.3.2 Der Typtransformator

Die mächtigste Variante, eine selektive Darstellung zu erzeugen, ist der Typtransformator. Aufgrund der Tatsache, daß das Werkzeug selbst in Oz geschrieben werden soll, kann man diesen durch typspezifische Prozeduren realisieren: Der darzustellende Oz-Wert wird einfach derart in einen anderen Oz-Wert umgewandelt, daß der gewünschte Effekt entsteht.

Neben dem Effekt, beliebige Teile des Baumes ausblenden zu können, lassen sich vor allem die Interna von abstrakten Datentypen sichtbar machen.

Eine transformierte Darstellung muß für den Benutzer als solche erkennbar sein. Zu diesem Zweck wird die Darstellung mit einer anderen Hintergrundfarbe unterlegt.

Ein Spezialfall der Typtransformation ist die Möglichkeit, nicht die Darstellung zu verändern, sondern eine Aktion auszulösen. Diese wird in Form einer Oz-Prozedur spezifiziert, deren Argument der der Darstellung zugrunde liegende Oz-Wert ist.

Aktionen ermöglichen das einfache Hinzuziehen externer Darstellungssysteme wie etwa des Investigators [23] zusammen mit dem Oz Explorer [27] und Da Vinci [6]. Dadurch lassen sich zum Beispiel weitere Informationen über Finite-Domain-Variablen, die Teil eines Constraint-Problems sind, ermitteln.

4.4 Transientenverwaltung

Ein Kernpunkt des Entwurfs ist die zentrale Verwaltung von Transienten und deren Darstellungen in einem Dictionary. Transienten- und Finite-Constraint-Knoten melden sich deswegen bei der Erzeugung beim sogenannten Constraint-Wächter an.

4.4.1 Registrierung

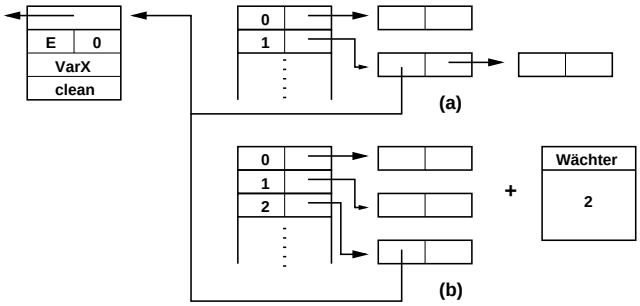


Abbildung 4.6: *Anmelden von Transienten*

Abbildung 4.6 verdeutlicht die Vorgehensweise bei der Registrierung von Transienten: Ist der Transient bereits bekannt, existiert dazu ein Eintrag im Dictionary (Fall a). Dann wird der zu registrierende Knoten der bestehenden Warteliste hinzugefügt und kein neuer Wächter erzeugt. Wird hingegen ein neuer Transient registriert (Fall b), erhält das Dictionary einen weitereren Eintrag, dessen Warteliste mit dem Knoten initialisiert ist. Danach wird ein neuer Wächter gestartet.

Diese Verfahrensweise ist dem Vorgehen des Emulators bei Transientensuspension nachempfunden [26].

4.4.2 Aktualisierung

Hat der Constraintspeicher neue Informationen über einen Transienten, für den ein Wächter existiert und triggert, muß die Darstellung aktualisiert werden. Dies geschieht in vier Schritten:

1. Löschen der alten Transientenknoten.
2. Einklinken der neuen Wertrepräsentationen.
3. Neuberechnung des Layouts und Neudarstellung für die betroffenen Wurzelknoten.

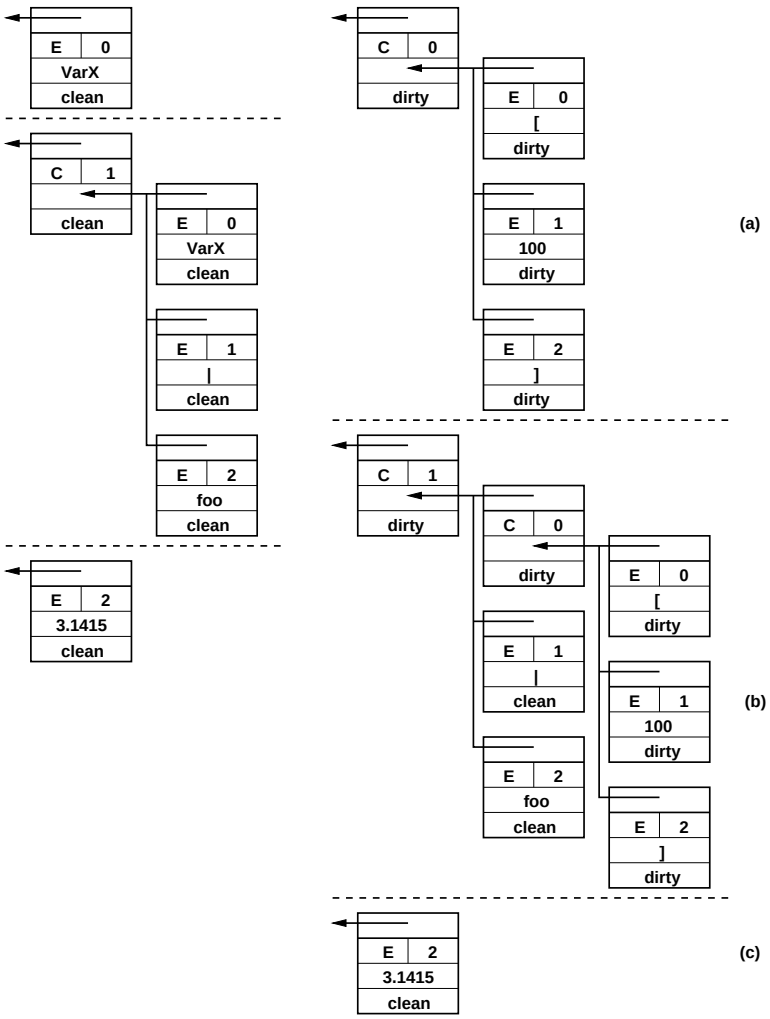


Abbildung 4.7: *Der Aktualisierungsfall am Beispiel von drei dargestellten Werten*

4. Verschieben der unveränderten Wurzelknoten, die unterhalb der betroffenen Wurzelknoten liegen.

Wächterverhalten. Im Aktivierungsfall löst der Wächter eine Nachricht an das Strukturmodul aus, deren Argument die Warteliste enthält. Danach wird entschieden, was mit dem Wächter geschieht: Ist der Transient des Wächters nun transparent, terminiert der Wächter. Ansonsten wird weiter überwacht.

Löschen und Einklinken. Zunächst werden die alten Transientendarstellungen gelöscht und die Knotenstrukturen intern aktualisiert. Dies erfolgt in aufsteigender Traversierung unter Verwendung der Eltern- und Indexfelder, deren Effekt Abbildung 4.7 darstellt. Dabei treten zwei Fälle auf:

(a) Die vollständige Ersetzung eines Baumes, die einfach zu handhaben ist.

(b) Die Teilbaumersetzung. Für diesen Fall sind die Traversierungsflags gedacht. Während der Ersetzung wird in aufsteigender Traversierung der Verweis auf den Wurzelknoten ermittelt und dabei das Traversierungsflag für jeden durchlaufenen Containerknoten aktiviert. Dadurch ist sichergestellt, daß alle von der Änderung betroffenen Teilbäume aktualisiert werden.

Der Ersetzungsvorgang umfaßt alle Darstellungen des betroffenen Transienten auf einmal. Diese können sich jedoch in unterschiedlichen Wurzelknoten befinden, weswegen die Verweise aller betroffenen, paarweise verschiedenen Wurzelknoten in einer Liste gesammelt werden.

Anordnen und Darstellen. Sind die Datenstrukturen intern aktualisiert, kann mit Hilfe der erstellten Liste das Layout der betroffenen Wurzelknoten jeweils in einem Durchgang neu berechnet werden. Dies geschieht in absteigender Traversierung. Weist dabei ein Teilbaum ein gelöschtes Traversierungsflag auf, werden dessen vorhandene Daten wiederverwendet und ein weiteres Absteigen verhindert. Dieses Vorgehen wird erst durch das Voraussetzen eines Anordnungsverfahrens mit Teilbaumisomorphie ermöglicht, wie in Abschnitt 3.1.2 verlangt.

Die Darstellung erfolgt dann beginnend mit dem Wurzelknoten in der Liste, der den kleinsten Index aufweist. Alle Wurzelknoten, deren Index größer oder gleich dem ermittelten kleinsten Index ist, erhalten den Zeichenbefehl. Das Zeichnen erfolgt ebenfalls in absteigender Traversierung. Ist das Traversierungsflag eines (Teil-)Baumes gelöscht, ist dessen Darstellung unverändert korrekt (Fall c in Abbildung 4.7). Der betroffene Baum wird deswegen in einer Operation nur verschoben, nicht jedoch gezeichnet.

4.5 Strukturerkennung

Im Relationsmodus soll die wiederholte Darstellung gleicher Information unterdrückt und durch Referenzen auf die definierende Darstellung ersetzt werden. Diese Arbeit leistet der Relationsmanager. Ähnlich wie bei der Transientenverwaltung (siehe Abschnitt 4.4), werden dazu bekannte Werte in einem Dictionary gespeichert.

Der Skopus für die Graphdarstellung ist auf jeweils alle Kindknoten eines Wurzelknotens beschränkt. In Relationsmodus verwaltet die Darstellungseinheit daher neben einem Dictionary aller Wurzelknoten auch ein Dictionary der zugehörigen Relationsmanager.

4.5.1 Erzeugung

Für den Relationsmodus muß das bisher betrachtete Datenmodell erweitert werden: Jeder Containerknoten erhält nun ein weiteres Feld, dessen Inhalt auf einen speziellen Strukturknoten zeigt, der eine Referenzliste verwaltet. Transientenknoten werden dabei auch wie Containerknoten behandelt, da diese zu Containerknoten werden können.

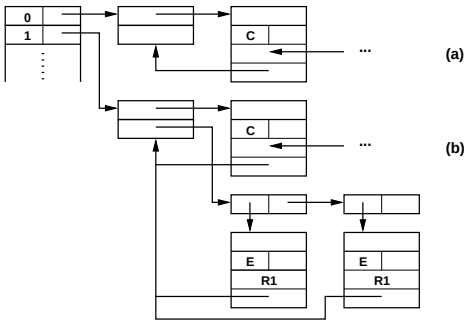


Abbildung 4.8: *Strukturerkennung durch den Relationsmanager*

Abbildung 4.8 verdeutlicht die Vorgehensweise der Strukturerkennung, die in Pre-Order-Traversierung arbeitet: Wird ein Teilbaum das erste Mal angemeldet, wird dies gespeichert und die Darstellung unverändert erzeugt (Fall a). Ist der angemeldete Wert bereits bekannt, wird eine Referenz erzeugt und in die Referenzliste eingetragen (Fall b). Referenzknoten verfügen ebenfalls über einen Verweis auf den Strukturknoten, damit sie später einfach aus der Liste gelöscht werden können.

Bei der Darstellung prüft nun jeder Containerknoten, ob die ihm zugehörige Referenzliste leer ist. Falls nicht, dann wird die Marke des Knotens mitangezeigt, ansonsten erfolgt eine normale Darstellung.

4.5.2 Referenzaktualisierung

```
declare X Y Z              (a)
{Inspect X}
Z = t(foo X) X = l(Z Y Y) --> R0=l(t(foo R0) R2=Y R2)
Y = Z                      --> R0=l(R1=t(foo R0) R1 R1)
declare X Y                (b)
{Inspect X}
X = tuple(Y Y Y Y)         --> tuple(R1=Y R1 R1 R1)
Y = 5                      --> tuple(5 5 5 5)
```

Abbildung 4.9: *Referenzaktualisierung*

Abbildung 4.9 zeigt die zwei verschiedenen Klassen von Aktualisierung, die in bezug auf Strukturerkennung unterschieden werden können: Das Auftreten neuer beziehungsweise das Verschwinden vorhandener Referenzen und die Zuweisung atomarer Werte an Transienten.

Im ersten Fall erfordert der Aktualisierungsvorgang von dem betroffenen Containerknoten das Überprüfen der Referenzliste. Ergibt sich daraufhin eine Änderung des Zustandes, muß dies aktualisiert werden, auch wenn kein Unterbaum des Referenzknoten selbst verändert wurde (Fall a).

Handelt es sich um einen Transientenknoten, der atomar oder zu einer anderen Re-

ferenz wird, löst dieser mit Hilfe der Referenzliste manuell eine Aktualisierung der einzelnen Referenzknoten aus (Fall b). Dies ist deswegen erforderlich, weil Referenzknoten, die auf einen Transientenknoten verweisen, nicht an den Constraint-Wächter übergeben werden.

4.6 Anordnungsverfahren

In diesem Abschnitt wird ein Anordnungsverfahren vorgestellt, das die in den Abschnitten 3.1.2 und 3.3.2 genannten Anforderungen erfüllt. Um die Darstellung zu vereinfachen, wird im folgenden der Begriff *Umriß* verwendet. Dieser bezeichnet das kleinste Rechteck, das eine Darstellung umhüllt.

4.6.1 Anordnungsberechnung

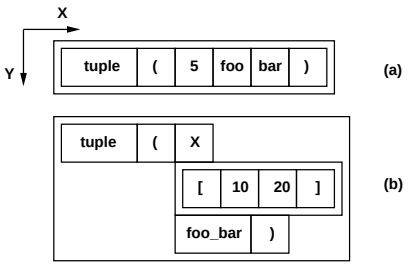


Abbildung 4.10: *Geometrische Konstruktion*

Abbildung 4.10 zeigt den Effekt der Anordnungsberechnung, die rekursiv erfolgt:

- Für Basisknoten wird die Größe des Umriß in Form von X- und Y-Dimensionen berechnet.
- Für einen Containerknoten werden zunächst die Umrisse der einzelnen Kindknoten berechnet. Diese werden mit Hilfe zweier Regeln so angeordnet, daß der Umriß des Containerknotens bezüglich dieser Regeln minimal wird:
  1. Ist keiner der Kindknoten selbst ein Containerknoten, werden diese horizontal angeordnet (Fall a).
  2. Ansonsten werden die Kindknoten vertikal, das heißt, untereinander angeordnet (Fall b).

Dieses Verfahren wird auch Inklusionsverfahren [14] genannt.

4.6.2 Ein Problemfall

Wird ein Transient in einem bisher horizontal ausgerichteten Containerknoten selbst zu einem Containerknoten, verletzt die dann erfolgende Aktualisierung auf den ersten Blick das Stabilitätskriterium: Die vertikale Ausrichtung des Baumes muß erst

hergestellt werden. Dies geschieht jedoch nur ein einziges Mal und stets vorhersagbar, so daß dies noch vertretbar ist. Die Kosten für die Neuausrichtung sind auch nicht teurer, da alle anderen Teilbäume zu diesem Zeitpunkt garantiert atomar sind. Nach der Neuanordnung kann der Platzbedarf auf der Y-Achse jedoch unnötig groß werden, wenn die Basistypen sehr kompakt sind. Dieses Problem tritt aber nur selten auf.

4.7 Der Zeichendienst

Der Zeichendienst muß ebenfalls inkrementell arbeiten. Er muß damit auskommen, Strukturen nur einmal zu zeichnen. Wenn sich die relative Position in der Darstellung ändert, dürfen einmal dargestellte Teile nur verschoben werden.

4.7.1 Markierte Bäume

Für den Zeichendienst muß das bisher betrachtete Datenmodell noch einmal erweitert werden: Jeder Knoten erhält nun eine eindeutige Marke. Mit Hilfe dieser Marken werden einzelne Knoten zu einer Knotenmenge zusammengefaßt. Dazu wird beim Zeichnen jedem Knoten eine Menge von Marken zugeordnet. In bezug auf eine Marke definiert sich die Knotenmenge nun durch alle Knoten, deren Markierung diese Marke enthält. Über diese Marke kann die Knotenmenge dann als abstrakte Einheit angesprochen werden. Das heißt, eine angeforderte Operation erfolgt zugleich auf allen der Menge zugehörigen Knoten.

Diese Marken entsprechen denen, die das von Mozart verwendete Grafiksystem Tcl/Tk zur Verfügung stellt.

4.7.2 Der Zeichenprozeß

Abbildung 4.11 zeigt eine intern aktualisierte Datenstruktur, die nun gezeichnet werden soll. Das Zeichnen erfolgt stets in absteigender Traversierung der Struktur. Die Idee ist, alle Kindknoten eines Containerknotens zu einer Knotenmenge zusammenzufassen. Jedem Knoten werden dazu die Marken aller Elternknoten entlang des Pfades bis zum Wurzelknoten und seine eigene Marke zugeordnet. Für den Knoten `100` in Abbildung 4.11 ist dies zum Beispiel `[t0 t4 t8]`.

Wird während der Traversierung ein Knoten mit gelöschten Traversierungsflag angetroffen, ist dessen Darstellung unverändert korrekt. Die Marke des Knotens wird nun dazu verwendet, den Knoten beziehungsweise den gesamten Baum an die neue Position zu schieben. Dies gelingt deswegen, weil jedem Kindknoten ebenfalls diese Marke zugeordnet ist. Für den Knoten `200` in Abbildung 4.11 geschieht dies mit `t7`.

4.8 Interaktion

In diesem Abschnitt wird dargelegt, wie die direkte Interaktion realisiert wird.

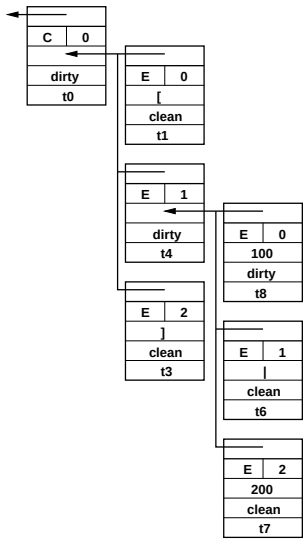


Abbildung 4.11: *Funktionsweise der Markierungen*

4.8.1 Interaktionspunkte

Um direkt interagieren zu können, bedarf es der Festlegung von Interaktionspunkten, über die man lokal Operationen auslösen kann. Zwei Klassen von Darstellungselementen werden dazu verwendet:

Limitindikatoren Traversierungsfilter plazieren, wenn die Darstellung unterdrückt wird, einen Limitindikator an der betreffenden Stelle. Diese sind also bestens geeignet, über diese die Einstellungen des jeweiligen Filters zu verändern.

Trennsymbole Label und Trennsymbole, also etwa runde und eckige Klammern beziehungsweise `|` und `#`, erlauben den schnellen und eindeutigen Zugriff auf Teile der betreffenden Datenstruktur. Über diese kann man dann alle für diesen Typ verfügbaren Operationen auslösen, also beispielsweise einen definierten Typtransformator aufrufen.

4.8.2 Interaktionsmedium

Wenn der Benutzer auf einen Interaktionspunkt klickt, will er Einstellungen in bezug auf die Darstellung vornehmen. Um das zu ermöglichen, werden Kontextmenüs eingesetzt. Diese bieten das für den entsprechenden Typ verfügbare „Sortiment“ von Einstellungen an. Das verfügbare Sortiment kann durch den Benutzer festgelegt werden.

Um die Orientierung für den Benutzer zu vereinfachen, sind alle Menüs nach gleichem Muster aufgebaut. Wenn eine spezifische Klasse von Operationen für einen Typ nicht verfügbar ist, sind die entsprechenden Einträge deaktiviert.

Kapitel 5

Implementierung

In diesem Kapitel werden grundlegende Aspekte der Implementierung vorgestellt, wobei insbesondere die Strukturverwaltung im Vordergrund steht.

5.1 Aufbau der Implementierung

Die grundlegende Methodik für die Implementierung besteht darin, die logischen Einheiten des Entwurfs (siehe Abschnitte 4.2 und 4.1) in Form von Klassen abzubilden. Diese binden durch Mehrfachvererbung einen Teil ihrer Subkomponenten ein und sind in Module gekapselt. Die einzelnen Dienste werden demzufolge innerhalb der Klassen durch Methoden realisiert.

5.1.1 Module

`Inspector`	`TreeWidget`	`TreeNodes`
`InspectorMain`	`StoreListener`	`CreateObjects`
`InspectorOptions`	`RelationManager`	`LayoutObjects`
`TreeWidget`	`GraphicSupport`	`DrawObjects`
	`Aux`	
	`TreeNodes`	

Tabelle 5.1: *Module der Implementierung*

Oz bietet die Möglichkeit, Module in Form von Funktoren zu beschreiben. Tabelle 5.1 zeigt die Module, die für die Implementierung definiert wurden. Es lassen sich drei primäre Module erkennen, die auf Untermodule zurückgreifen:

Inspector Der `Inspector` besteht aus dem Kernmodul `InspectorMain`, welches die Kernlogik und Benutzeroberfläche beinhaltet, sowie dem Konfigurationsmodul `InspectorOptions`. Darin sind Optionen sowohl für den Inspector als auch für Darstellungseinheiten realisiert.

TreeWidget Das `TreeWidget` realisiert die Darstellungseinheit, und bindet den Constraint-Wächter `StoreListener`, die Strukturerkennung `RelationManager` und den Grafikdienst `GraphicSupport` als separate Module ein. Daneben sind die Knotenrepräsentationen in zwei Modulen realisiert, und zwar dem Hilfsmodul `Aux` und den eigentlichen `TreeNodes`.

TreeNodes Die Knotenrepräsentation erfolgt als zusammengeführte Klasse, welche die drei Dienstgruppen Strukturverwaltung, Anordnung und Zeichnen in separaten Unterklassen realisiert. Jene befinden sich jeweils in den Modulen `CreateObjects`, `LayoutObjects` und `DrawObjects`.

In bezug auf den Anordnungsdienst weicht die Implementierung insofern vom idealisierten Entwurf ab, als dieser direkt mit der Knotenrepräsentation verschmolzen wurde, anstatt ihn im TreeWidget selbst zu realisieren.

5.1.2 Nachrichtenbehandlung

Der Inspector und die jeweiligen TreeWidgets müssen über sichere Schnittstellen kommunizieren. Infolge der Tatsache, daß Nachrichten an Objekte in Oz voll emanzipiert sind, kann dazu ein Port herangezogen werden. Ein Port ist ein synchronisierter Verweis (für alle Producer) auf einen Strom [22], von dem das betreffende Objekt direkt die Nachrichten liest (Consumer).

5.1.3 Ausführungsmodell

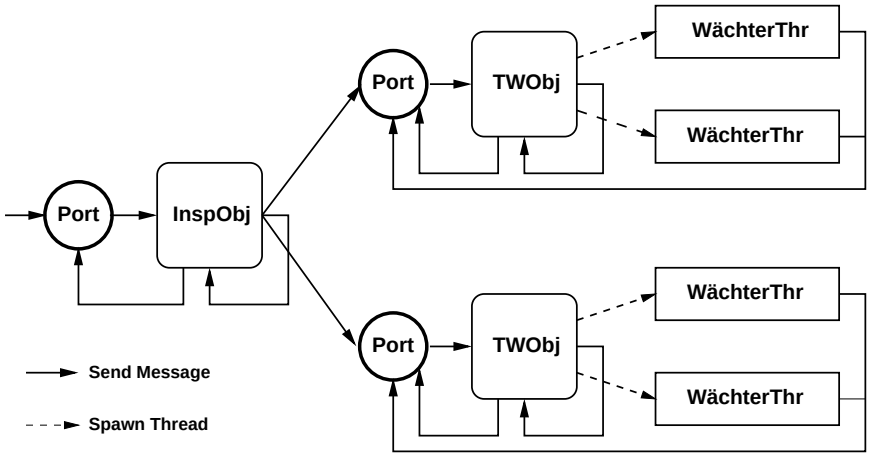


Abbildung 5.1: *Nachrichtenfluß und Ausführung mit zwei Darstellungseinheiten*

Abbildung 5.1 zeigt das Ausführungsmodell für den Inspector, das heißt, wie die durch den jeweiligen Port anfallenden Nachrichten abgearbeitet werden. Dies geschieht für das Inspectorobjekt (InspObj) und alle TreeWidgetobjekte (TWObj) jeweils in einen eigenem Thread, der ausschließlich entweder die vom Port oder die lokal generierten Methodenaufrufe abarbeitet. Eine derartige Kombination nennt man auch *aktives Objekt* [9].

Abbildung 5.1 zeigt ebenfalls, daß die Transientenwächter (siehe Abschnitt 5.3) in Form von Threads (WächterThr) realisiert sind, deren Aktualisierungsnachricht über den Port an das zugehörige TreeWidget geschickt wird.

5.2 Oz-Werte

In der Implementierung werden Oz-Werte als Objekte dargestellt. Bis auf wenige Ausnahmen, ist dazu für jeden Oz-Typ eine eigene Knotenklasse definiert. Diese entsteht durch Mehrfachvererbung aus drei Unterklassen, welche die Dienstgruppen Strukturverwaltung, Anordnung und Zeichnen für jeden Knotentyp festlegen.

5.2.1 Einfache Knoten

Create	`value`	Referenz auf den Oz-Wert
	`type`	Typtag des Knotens
	`parent`	Verweis auf den Elternknoten
	`index`	Position im Elternknoten
	`visual`	Verweis auf die Darstellungseinheit
	`tag`	Eindeutige ID für den Knoten
Layout	`xDim`	Relative Breite des Knotens
	`string`	Stringrepräsentation des Wertes
Draw	`dirty`	zweiwertiges Traversierungsflag

Tabelle 5.2: *Attribute von Einfachen Knoten, nach Dienstgruppen getrennt*

Einfache Knoten arbeiten mit den in Tabelle 5.2 dargestellten Attributen und besitzen folgende Methoden:

create(Value Parent Index Visual Depth) leistet den Strukturaufbau, indem

1. Die übergebenen Werte in den entsprechenden Attributen gespeichert werden.
2. Eine für jeden Knoten eindeutige ID mittels `newTag`-Methode der zugehörigen Darstellungseinheit `Visual` erstellt und in `tag` gespeichert wird.

3a. Im Fall eines Containerknotens, die Methode `createContainer` aufgerufen wird, die den Aufbau der Substruktur übernimmt.

3b. Im Fall eines Transientenknotens, dieser mittels `logVar`-Methode der zugehörigen Darstellungseinheit `Visual` beim Contraint-Wächter angemeldet wird.

layout berechnet für einfache Knoten die Stringrepräsentation `string` des Wertes, deren Länge in Textzeichen in `xDim` gespeichert wird. Im Fall eines Containerknotens, wird der in Abschnitt 4.6 definierte Umriß des Knotens ausgerechnet.

`xDim` dient als Traversierungsflag für die Layout-Phase. Ist `xDim` frei, wird die Anordnung neu berechnet, ansonsten geschieht nichts.

draw(X Y)/undraw/makeDirty realisieren die Zeichendienste. Das Verschieben und die Löschung eines Teilbaums erfolgen seitens des Grafiksystems mit jeweils nur einer Instruktion. Da die Struktur keine absoluten Koordinaten enthält, ist beim Verschieben keine weitere Traversierung auf Oz-Seite erforderlich.

Leider funktioniert dies nicht beim Löschen: Dieser Zustand muß allen Unterbäumen bekannt gemacht werden, um unnötige Transientenaktualisierung zu vermeiden. Die `undraw`-Methode ruft also immer die `makeDirty`-Funktion auf, die rekursiv für jeden Knoten des betroffenen Teilbaumes das `dirty`-Flag setzt.

Ob der durch den Knoten definierte Baum gezeichnet oder nur verschoben werden muß, wird durch das gesetzte beziehungsweise gelöschte `dirty`-Flag entschieden. Die eigentliche Grafikoperation wird dabei immer durch Aufruf der entsprechenden Lowlevel-Methoden der zugehörigen Darstellungseinheit `visual` ausgelöst (siehe Abschnitt 5.4).

Dazu gesellen sich in jeder Unterklasse eine Reihe von Hilfsmethoden, die zur Realisierung des entsprechenden Dienstes notwendig sind.

5.2.2 Containerknoten

Create	`items`	Dictionary für die Unterknoten
	`width`	Dargestellte Breite
	`maxWidth`	Maximale Anzahl Unterknoten (Speicherinfo)
	`depth`	Tiefeninformation
Layout	`lastXDim`	X-Dimension des letzten Unterknotens
	`yDim`	Y-Dimension des Umriß
	`horzMode`	Anordnungsmodusflag
Draw	`dirty`	dreiwertiges Traversierungsflag

Tabelle 5.3: *Weitere Attribute für Containerknoten*

Containerknoten müssen eine Substruktur verwalten und benötigen daher zusätzliche Attribute, die in Tabelle 5.3 dargestellt werden. Ihnen fehlt allerdings das `string`-Attribut, denn deren Repräsentation entsteht nur durch Kombination der Kindknotenrepräsentationen.

Zudem ergibt sich eine Änderung des `dirty`-Flags, das nun dreiwertig wird: fresh, true und false. Der hinzugekommene fresh-Wert wird zur Identifikation niemals dargestellter Unterbäume benötigt. Diese werden dann herausgefiltert, wenn die Darstellung beziehungsweise deren Aktualisierung so schnell wie möglich beendet werden soll (Stopbefehl).

Erweiterung der Basisdienste

Infolge der Substruktur benötigen die `create`-, `layout`- und `draw`-Methoden der spezialisierten Containertypen standardisierte Hilfsmethoden, um ihre Arbeit zu ver-

richten. Der Basiscontainertyp definiert dazu folgende Standardmethoden:

checkHorzMode(I)/horizontalLayout(I XD)/verticalLayout(I XD YD) werden von der `layout`-Methode eines Containerknotens verwendet, wenn neu berechnet werden muß, also `xDim` frei ist.

Zunächst wird mittels `checkHorzMode` überprüft, ob die Ausrichtung horizontal oder vertikal erfolgen soll. Das Kriterium dafür ist das Auffinden eines Containerknotens in den direkten Kindknoten.

Abhängig davon, wird die entsprechende Anordnungsmethode aufgerufen, die rekursiv die Kindknoten anordnet.

Die ermittelten Geometriewerte werden in den Attributen den Knotens gespeichert und anschließend wird das `dirty`-Flag gesetzt, um eine Darstellungsaktualisierung möglich zu machen.

performDraw(X Y)/horizontalDraw(I X Y)/verticalDraw(I X Y) Für alle Containertypen gibt es nur eine `draw`-Methode, die mittels `dirty`-Flag testet, ob gezeichnet oder verschoben werden soll.

Ist `dirty` gesetzt, geschehen drei Dinge:

1. Die ID `tag` des Containerknotens wird mittels `tagTreeDown`-Methode der zugehörigen Darstellungseinheit `visual` an die Tk-Engine übertragen und dort global gespeichert.
2. `performDraw(X Y)` wird aufgerufen. Diese Methode kann spezialisiert sein und erledigt die Darstellung der Substruktur unter Benutzung von `horizontalDraw` oder `verticalDraw`.
3. Die ID des Containerknotens wird mittels `tagTreeUp`-Methode der zugehörigen Darstellungseinheit `visual` wieder entfernt, da alle Kindknoten dargestellt beziehungsweise verschoben wurden.

Die Kombinatorfunktionen testen bei jeder Iteration, ob der Benutzer Stop befohlen hat oder nicht.

Ersetzungsdienste

Eine wesentliche Operation des Systems ist die Veränderung von Teilbäumen infolge von neuen Speicherinformationen oder Interaktion. Die dazu notwendige inkrementelle Aktualisierung geschieht stets in zwei Schritten:

1. Einklinken neuer Teilbäume und Änderungspropagierung in aufsteigender Traversierung.
2. Anordnen und Darstellen in jeweils absteigender Traversierung.

Der zweite Schritt wird im wesentlichen durch die Darstellungseinheit gesteuert, die dazu auf die `layout`- und `draw`-Methoden der Knoten zurückgreift (siehe Abschnitt 5.3.1).

Aus Sicht der Implementierung, ergeben sich für ersten Schritt zwei Klassen: Operationen, die aufgrund bereits vorhandener Daten jederzeit umkehrbar sind, und solche, für die es spezieller Vorsorge bedarf, um die Umkehrbarkeit zu erhalten.

Folgende Methoden dienen dem Einklinken neuer Teilbäume:

tell/replace(I Value) Liegt für einen Transientenknoten neue Information vor, kann der alte Wert vergessen werden. Deswegen ist eine `tell`-Operation nichts weiter, als ein `replace`-Aufruf an den Elternknoten, um den neuen Wert einzuklinken. Für Knoten, die Feature-Constraints repräsentieren, erfolgt die `replace`-Operation im Knoten selbst.

modifyWidth(I N)/modifyDepth(I N) Werden die Traversierungsfilter eingesetzt, kann durch das `value`-Attribut jeden Knotens jederzeit die ursprüngliche Information dargestellt werden, so daß ebenfalls ein `replace`-Aufruf ausreicht. Dies ist allerdings nur für die Tiefenänderung (`modifyDepth`) relevant, da eine Breitenänderung (`modifyWidth`) im Containerknoten direkt abgewickelt wird.

map(I F)/link(I Value)/unlink(I) Wird ein Typtransformator mittels (`map`) eingesetzt, muß die Umkehrbarkeit durch zusätzliche Maßnahmen gesichert werden.

Dazu werden spezielle Adapter-Objekte verwendet. Diese führen den alten Wert mit, verhalten sich ansonsten aber wie der aktuell dargestellte Typ. Um eine beliebige Schachtelung von Typtransformator und Filtern zu erlauben, muß verhindert werden, daß ein Adapter-Objekt durch `replace`, `link` und `unlink` irrtümlich überschrieben wird. Dies wird mit einem Test `mustChange` realisiert: Evaluiert dieser zu true, wird der neue Knoten nicht direkt im Dictionary plaziert, sondern in das vorhandene Adapter-Objekt geklinkt.

Zwei Methoden dienen der Änderungspropagierung:

notify setzt das Layout-Traversierungsflag (xDim wird frei) entlang des Pfades vom betroffenen Knoten bis hinauf zum Wurzelknoten.

getRootIndex ist eine Optimierung für die `tell`-Aufrufe. Die Ermittlung des Wurzelindex, der in die Aktualisierungsliste aufgenommen werden muß, erfordert bereits eine aufsteigende Traversierung. Um eine erneute Traversierung entlang des selben Pfades durch `notify` zu verhindern, leistet `getRootIndex` dessen Anteil, das Layout-Traversierungsflag zu löschen, gleich mit. Dies funktioniert deswegen, weil zu diesem Zeitpunkt sichergestellt ist, daß sich der Baum tatsächlich ändern wird – sonst hätte der zuständige Wächter nicht getriggert.

5.3 Die Darstellungseinheit

Die Darstellungseinheit selbst ist als Klasse `TreeWidget` realisiert, die durch Mehrfachvererbung die Klassen `StoreListener` und `GrapicSupport` aus den gleichnamigen Modulen an sich bindet.

5.3.1 Basisdienste

Die Darstellungseinheit verwaltet im wesentlichen alle dargestellten Strukturen und leistet die Aktualisierung, sofern das erforderlich ist.

Die Implementierung sieht dazu folgende Methoden vor:

display(Value) Diese Methode leistet die Erstdarstellung eines neuen Wertes `Value` unter Verwendung von `treeCreate`.

treeCreate(Val Parent Index Depth) Diese Methode leistet den eigentlichen Aufbauprozeß des Knotens, wie er in Abschnitt 4.3 definiert ist. Nach Durchlauf der verschiedenen Filter, wird das für den entstandenen Wert passende Knotenobjekt generiert. Die Positionierung in der Struktur wird dabei über `Parent` und `Index` erreicht, die an den Konstruktor des zu erzeugenden Knotens übergeben werden. Wird ein Containerknoten konstruiert, greift dieser zur Konstruktion seiner Kindknoten rekursiv auf diese Methode zurück.

update(Is M)/performUpdate(I X Y MaxX) Diese Methoden leisten die Layout- und Zeichenphase bei der Aktualisierung. Alle Wurzelknoten in der Liste `Is` erhalten den `layout`-Befehl und während dessen wird das Minimum `M` der Liste ermittelt. Dieses Minimum `M` gibt den Startindex `I` für die `performUpdate`-Methode an, die rekursiv für alle darunter liegenden Wurzelknoten den `draw`-Befehl absetzt. Dabei wird die maximale X-Dimension `MaxX` ermittelt, die für die Berechnung des sichtbaren Ausschnittes erforderlich ist.

call(Obj Mesg) Die Methode dient der Realisierung knotenbasierter Interaktion. Wird auf einen Wert geklickt, wird zunächst das dazugehörige Knotenobjekt `Obj` gesucht und ein dem Typ entsprechenden Menu geöffnet (sofern vorhanden).

Ist nun eine Aktion aus dem Menu gefunden, die als `Mesg` codiert ist, wird diese mit Hilfe der `call`-Funktion sicher an das Objekt `Obj` geschickt.

5.3.2 Transientenverwaltung

Die `StoreListener`-Klasse realisiert den Constraint-Wächter und muß im wesentlichen zwei Dinge leisten: Das Verwalten übergebener Transientenknoten und die Aktualisierungsnachricht bewirken, wenn neue Information vorliegt.

Dazu wird auf einer doppelt verketteten Transientenliste gearbeitet, deren Einträge aus Objekten bestehen und jeweils einen bekannten Transienten repräsentieren. Innerhalb dieser wird die Liste seiner Darstellungen in Form von Knotenreferenzen verwaltet.

Die Operationen der Transientenverwaltung sind in Form von Methoden realisiert:

logVar(Value Node) Diese Methode wird von den Transientenknoten aufgerufen, um sich zu registrieren. Dazu wird zunächst überprüft, ob der Wert `Value` bereits in der Transientenliste verzeichnet ist:

- Ist der Wert bekannt, wird der Knoten `Node` in die Referenzliste des ermittelten Transienteneintrags `EntryObj` eingetragen.
- Ist der Wert unbekannt, wird ein neuer Transienteneintrag `EntryObj` erzeugt, dessen Referenzliste mit einem Knoten besetzt ist. Danach wird ein Thread gestartet, der die Überwachung des Wertes leistet und dazu die `listen`-Methode mit dem `EntryObj` als Argument aufruft.

listen(EntryObj) Die `listen`-Methode synchronisiert mittels zweier Funktionen, `GetsBoundB` für logische Variablen und `WaitQuiet` für Futures, zunächst darauf, daß der überwachte Wert eine Änderung erfahren hat. Falls ja, wird die `notifyNodes(EntryObj)`-Nachricht auf den Port der zugehörigen Darstellungseinheit geschickt.

notifyNodes(EntryObj) realisiert die Aktualisierung der Darstellung in zwei Phasen:

1. Diese Methode ruft für jeden Transientenknoten in der Referenzliste des `EntryObj` die `tell`-Methode auf, deren Rückgabewert jeweils der Wurzelindex ist. Diese werden in einer Liste `Is`, deren Elemente paarweise disjunkt sind, gesammelt.
2. Mit der so erstellen Liste wird `update(Is 0)` aufgerufen, welche den Top-Down-Anteil der Aktualisierung realisiert (siehe Abschnitt 5.3.1).

5.4 Die Grafikeinheit

Die Implementierung der Darstellungseinheit ist in bezug auf die Grafikeinheit vollständig gekapselt: Wird die Grafikeinheit ausgetauscht, sollte das System sofort mit dem neuen Grafiksystem zusammenarbeiten.

5.4.1 Verwendetes System

Das Tcl/Tk-Toolkit bietet zwei grundlegende Komponenten, sogenannte *Widgets*, an, welche die erforderlichen Darstellungsleistungen erbringen können:

Canvas-Widget Das Canvas-Widget erlaubt die flexible Verwaltung grafischer Primitive, darunter Text und Bitmaps, auf einem sehr grundlegenden Level. Zur Erzeugung von Darstellungen steht ein Satz Steuerfunktionen zur Verfügung, die allesamt manuell ausgelöst werden müssen. Zur Verwaltung größerer Einheiten kann das bereits erwähnte Markierungsprinzip verwendet werden.

Textwidget Diese Einheit ist besonders für textbasierte Darstellungen ausgelegt und wird auch vom Browser verwendet. Es bietet einige, sehr einfache Merkmale einer Textverarbeitung, etwa Zeilenumbruch und Texthervorhebung, ist aber vergleichsweise langsam. Auch bei Benutzung des Textwidgets kann das Markierungssystem verwendet werden.

Die Implementierung benutzt das Canvas-Widget, da dies eindeutig schneller und flexibler zu handhaben ist.

5.4.2 Architektur

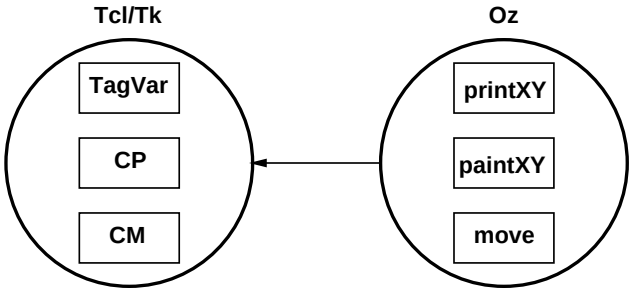


Abbildung 5.2: *Client-Server Architektur des Grafiksystems*

Abbildung 5.2 zeigt die vereinfachte Client-Server-Architektur des Grafiksystems der Darstellungseinheit.

Um möglichst effizient zu sein, wird möglichst viel Arbeit in bezug auf die Darstellung auf die Tcl/Tk-Shell ausgelagert, indem:

1. Serverseitig Skripte verwendet werden, die eine wiederholte Übertragung identischer Information zu verhindern.
2. Alle Baummarkierungen (Tags) auf dem Server erzeugt werden, und zwar derart, daß es ausreicht, pro Traversierung jede Knoten ID nur einmal zu übertragen.

5.4.3 Dienste

Auf der Oz-Seite stehen dazu mehrere Methoden zur Verfügung, die einen Aufruf der entsprechenden Tcl/Tk-Skripte bewirken. Es sind dies:

printXY(X Y String Tag ColorKey)/paintXY `printXY` wird zur Darstellung atomarer Knoten benötigt und bewirkt den Aufruf des `CP`-Skriptes, welches die Darstellung ausführt. Dazu wird unter Benutzung der `TagVar`-Variable und `Tag` zunächst die vollständige Markierung des Knotens erstellt, um damit ein entsprechendes Canvas-Textitem zu bauen.

`paintXY` erledigt entsprechendes für die Darstellung von Traversierungslimits.

place(X Y Tag)/move(X Y Tag FTag) Diese beiden Methoden realisieren das Verschieben. Einfache Knoten werden mittels `place` direkt an die Koordinaten gesetzt, während Containerknoten verschoben werden. Dazu wird das `move`-Skript aufgerufen. Mit Hilfe der Position des ersten Elements, dargestellt durch `FTag`, wird die die Differenz zur Sollposition (`X`, `Y`) ausgerechnet. Um diesen Betrag wird der durch `Tag` definierte Baum dann verschoben.

Die Trennung zwischen erstem Element- und Baumtag ist deswegen erforderlich, weil sich das erste Element durch Wegfall/Hinzufügen von Klammern ändern kann und daher die Position des Baumtags alleine nicht mehr stimmt.

delete(Tag) ruft das Skript auf, welches den durch `Tag` beschriebenen Teilbaum in einer Operation löscht.

tagTreeDown(Tag)/tagTreeUp Diese beiden Methoden dienen dazu, die Variable `TagVar` serverseitig auf die richtige Tiefe zu setzen. `tagTreeDown` fügt dazu das Tag `Tag` an die durch `TagVar` beschriebene Liste als erstes Element an, `tagTreeUp` entfernt dementsprechend das aktuelle erste Element von `TagVar` wieder. `TagVar` enthält damit immer alle Tags der Containerknoten entlang des Pfades vom Wurzelknoten bis zur aktuellen Tiefe.

Daneben gibt es noch einige, kleinere Hilfsfunktionen, die an dieser Stelle nicht weiter erläutert werden.

Kapitel 6

Evaluierung

Die Evaluierung hat das Ziel, zu überprüfen, inwieweit die diskutierten Konzepte tatsächlich funktionieren. Dazu werden zwei charakteristische Fallgruppen betrachtet, die zusammen Aufschluß über die Qualität des Inspectors geben:

1. Die Geschwindigkeit der Darstellungerzeugung ohne Transienten.
2. Die Effizienz des Aktualisierungsverfahrens.

Ergänzend dazu, wird noch der Speicherbedarf für die Strukturen herangezogen.

6.1 Durchführung

Architektur	Intel 80x86
Prozessor	Pentium MMX
Taktfrequenz	200 Mhz
Hauptspeicher	128 MB
Betriebssystem	Linux 2.2.10 (Suse 6.2)
Mozart Version	1.1.0

Tabelle 6.1: *Die Testplattform*

Im Rahmen der Evaluierung wird als Vergleichssystem der Browser eingesetzt, dessen Werte sich direkt vergleichen lassen.

Um aussagefähige Zahlen zu erhalten, wurden wiederholt mehrere Benchmarks auf der in Tabelle 6.1 dargestellten Plattform durchgeführt, und zwar jeweils mit beiden Systemen. Für jeden Benchmark wurden dabei der Zeitbedarf in Millisekunden (t_I für den Inspector beziehungsweise t_B für den Browser) und der Speicherverbrauch in Kilobytes (m_I beziehungsweise m_B) festgehalten.

Die so erhaltenen Zwischenergebnisse werden dann gemittelt zur Berechnung des Endergebnisses, dem Speedup $\Delta_T = \frac{t_B}{t_I}$ und der Speichereinsparung $\Delta_M = \frac{m_B}{m_I}$, herangezogen.

6.2 Geschwindigkeit

Zu Anfang steht die Frage, wie sich die Kerndatenstrukturen Tupel, Listen und Records gegenüber dem Browser verhalten – genauer, wie schnell sie sind und wie gut sie skalieren.

Um dies zu beantworten, wird in fünf Schritten eine jeweils um Faktor Acht vergrößerte Struktur aufgebaut. Zuletzt sind so über 32000 Knoten am Stück darzustellen – hier am Beispiel von Records gezeigt:

```
T = r(a:A b:A c:A d:A e:A f:A g:A h:A)
A = r(a:B b:B c:B d:B e:B f:B g:B h:B) % 5 (Schritt 1)
B = r(a:C b:C c:C d:C e:C f:C g:C h:C) % 5 (Schritt 2)
C = r(a:D b:D c:D d:D e:D f:D g:D h:D) % 5 (Schritt 3)
D = r(a:E b:E c:E d:E e:E f:E g:E h:E) % 5 (Schritt 4)
E = 5                                 %   (Schritt 5)
```

Die so erzeugte Struktur in `T` enthält in den Blättern keine Transienten, sondern nur einfache Zahlen (jedesmal `5`). Damit wird der Einfluß der Transientenverwaltung beider Systeme ausgeblendet.

6.2.1 Tupel

Operation	t_I	m_I	t_B	m_B	Δ_T	Δ_M
8 Blätter	50	1	100	5	2	5
64 Blätter	80	22	400	40	5	1.8
512 Blätter	550	135	2790	282	5	2
4096 Blätter	3480	611	20020	1867	5.8	3
32768 Blätter	25030	4420	163330	13787	6.5	3

Tabelle 6.2: *Geschwindigkeit für Tupel*

Tabelle 6.2 zeigt die Ergebnisse für normale Tupel. Im Vergleich mit dem Browser, ist der Inspector für größere Strukturen circa sechsmal schneller und benötigt lediglich ein Drittel des Speichers.

6.2.2 Listen

Die Darstellung von echten Listen erfolgt normalisiert, das heißt, nur mit Klammern versehen. Festzustellen, ob es sich um eine echte Liste handelt, erfordert zusätzlichen Aufwand während der Strukturerzeugung. Um herauszufinden, ob dieser Aufwand signifikant ist, wurden auch Listen getestet.

Die in Tabelle 6.3 dargestellten Ergebnisse machen deutlich, daß sich echte Listen in etwa wie die in Abschnitt 6.2.1 betrachteten Tupel verhalten. Der zusätzliche Aufwand ist also vernachlässigbar.

Operation	t_I	m_I	t_B	m_B	Δ_T	Δ_M
8 Blätter	50	1	90	2	1.8	2
64 Blätter	140	5	620	26	4.4	5
512 Blätter	640	66	2950	210	4.6	3
4096 Blätter	3140	549	19930	1869	6.3	3
32768 Blätter	24610	4420	166400	13999	6.7	3

Tabelle 6.3: *Geschwindigkeit für Listen*

6.2.3 Records

Operation	t_I	m_I	t_B	m_B	Δ_T	Δ_M
8 Blätter	80	1	100	7	1.2	7
64 Blätter	340	30	760	42	2.2	1.4
512 Blätter	1180	185	3810	298	3.2	1.6
4096 Blätter	7140	1013	30470	1918	4.2	1.9
32768 Blätter	60260	7655	241430	14955	4.0	1.9

Tabelle 6.4: *Geschwindigkeit für Records*

Da die Features von Records mitangezeigt werden müssen, ist deren Darstellung prinzipiell bei beiden Systemen etwas teurer als die von Tupeln, wie Tabelle 6.4 zeigt.

Die zeitliche Zunahme ist beim Inspector aber mit über 110 Prozent deutlich ausgeprägter als beim Browser mit circa 50 Prozent gegenüber den Tupeln. Das liegt daran, daß der Inspector die Trenner zwischen Feature und Teilbaum separat dargestellt. Er ermöglicht damit unterschiedliche Farben für Feature und Trennelement.

Das Beispiel ist extra so gewählt, daß sich dieser Overhead besonders stark bemerkbar macht, indem Feature und Trenner gleich breit sind (1 Textzeichen). Werden die Feature breiter, das heißt, mehrere Textzeichen lang, schrumpft dieser Overhead wieder zusammen (siehe Abschnitt 6.3.1).

Insgesamt ist der Inspector immer noch deutlich schneller als der Browser. Für das Beispiel ergibt sich jedoch ein Rückgang des Geschwindigkeitszuwachses von Faktor Sechs auf Faktor Vier.

In bezug auf den Speicherverbrauch ergeben sich zwei Schlußfolgerungen:

1. Der Speicherbedarf von Tupeln gegenüber Records ist beim Browser deutlich zu hoch – der Zuwachs für Records beträgt lediglich circa 9 Prozent.

2. Der Speicherbedarf von Records gegenüber Tupeln ist beim Inspector erheblich: circa 73 Prozent mehr – das ist aber immer noch nur halb so viel wie beim Browser.

6.3 Aktualisierungsverhalten

Die Ergebnisse des vorigen Abschnittes geben bereits darüber Auskunft, welche Erwartungen man an die Effizienz Aktualisierungsverhalten stellen kann: Der Browser kann nur im Einzelfall schneller als der Inspector werden, und das nur dann, wenn die Aktualisierungsmechanismen des Inspectors im schlechten Fall arbeiten.

6.3.1 Lokale Aktualisierung

Um zu prüfen, wie effektiv die Aktualisierungstrategien beider Systeme arbeiten können, wird eine große Struktur nach erfolgter Darstellung nur einmal lokal verändert.

```
X = {MakeTuple huge 100}
{For 1 49 1 proc {$ I} X.I = FD end}
{For 51 100 1 proc {$ I} X.I = FD end}
%% {Inspect X} bzw. {Browse X}
```

Mit obigem Beispiel ensteht ein Tupel, das 99-mal das exportierte Interface des ziemlich umfangreichen Finite-Domain-Moduls und an der 50. Position eine freie Variable enthält. Dieses wird nun dem Inspector beziehungsweise Browser übergeben, deren Traversierungslimits beide gleich konfiguriert sind.

Operation	t_I	m_I	t_B	m_B	Δ_T	Δ_M
`X` darstellen	5010	978	50220	2450	10	2.5
`X.50 = FD`	310	995	690	2473	2	2.5

Tabelle 6.5: *Lokale Aktualisierung in Großer Struktur*

Tabelle 6.5 zeigt nun die Ergebnisse in Hinblick auf die Erstdarstellung des Wertes und die Aktualisierung. Zwei Dinge sind ersichtlich: Erstens fällt der Browser bei der Erstdarstellung um Faktor Zehn gegenüber dem Inspector zurück. Zweitens erfolgt die Aktualisierung bei beiden Systemen sehr effizient. Besonders interessant ist, daß der Inspector zwar absolut um den Faktor Zwei schneller ist, die Aktualisierung relativ aber mit 6 Prozent Anteil an der Erstdarstellung gegenüber 1 Prozent des Browsers etwas gewichtiger ist. Es wird also der Eindruck erweckt, daß der Mechanismus des Browsers besonders effizient ist.

6.3.2 Exponentielles Wachstum

Das Ersetzen eines einzelnen Transienten an sich ist noch nicht allzu aussagekräftig. Interessant wird es aber, wenn ein und der selbe Transient mehrfach auftritt und die Datenstruktur infolge der Aktualisierung exponentiell wächst:

```
%% {Inspect T} bzw. {Browse T}
T = t(A A A A)
A = t(B B B B)
```

```
B = t(C C C C)
C = t(D D D D)
D = t(E E E E)
E = 5
```

An Hand der nun ermitteln Werte läßt sich dann beurteilen, welchen Einfluß die Transientenverwaltung und -aktualisierungstrategie des jeweiligen Systems auf die Gesamtgeschwindigkeit der Darstellung hat.

Operation	t_I	m_I	t_B	m_B	Δ_T	Δ_M
`T` darstellen	20	1	30	1	1	1
`T = t(A A A A)`	30	5	30	3	1	0.6
`A = t(B B B B)`	40	11	70	13	2	1
`B = t(C C C C)`	90	17	490	55	5	3
`C = t(D D D D)`	290	55	2150	212	7	4
`D = t(E E E E)`	1140	201	9180	845	8	4
`E = 5`	2240	200	18200	569	8	3

Tabelle 6.6: *Exponentielles Wachstum*

Tabelle 6.6 zeigt die Ergebnisse für beide Systeme: Auffallend ist, daß der Browser infolge der Transienten noch weiter hinter den Inspector zurückfällt, und zwar von Faktor Sechs (siehe Abschnitt 6.2.1) auf Faktor Acht. Entgegen dem ersten Eindruck aus Abschnitt 6.3.1, ist die Verwaltung der Transienten beim Browser also relativ teuer gelöst. Zugleich entsteht der Eindruck, daß dessen Strategie, den Aktualisierungsaufwand zu minimieren, bei diesem Beispiel nicht richtig funktioniert. Zudem benötigt der Browser noch mehr Speicher als der Inspector, und zwar viermal mehr.

6.3.3 Viele Gleiche Transientendarstellungen

Die in Abschnitt 6.3.2 ermittelten Ergebnisse geben bereits Aufschluß darüber, wie die Systeme Änderungen in einem Transienten handhaben, der zahlreich dargestellt wird. Dabei kam der Verdacht auf, daß die Aktualisierungsstrategie des Browsers nicht richtig funktioniert. Dieses und das nachfolgende Beispiel sind so angelegt, darüber Klarheit zu bringen.

```
X = {MakeTuple big_tuple 30}
Y = {MakeTuple lines 30}
{For 1 30 1 proc {$ I} X.I = Y end}
{For 1 30 1 proc {$ I} Y.I = V end}
%% {Inspect X} bzw. {Browse X}
V = 5
```

In diesem Fall werden 900 Darstellungen des gleichen Transienten `V` durch `V = 5` transparent, also in einem Rutsch ersetzt. Die Vermutung ist, daß dies sehr effizient geschehen wird, wenn die Aktualisierungsstrategien funktionieren.

Die Zahlen in Tabelle 6.7 zeigen Interessantes:

Operation	t_I	m_I	t_B	m_B	Δ_T	Δ_M
`X` darstellen	490	150	4350	608	9	4
`V = 5`	860	143	9180	366	10	2.5

Tabelle 6.7: *Viele Gleiche Transientendarstellungen*

1. Sowohl für die Erstdarstellung als auch für die Aktualisierung benötigt der Browser sehr lange. Er fällt auf Faktor Neun bis Faktor Zehn hinter den Inspector zurück.

2. Die Aktualisierungsstrategie des Inspectors funktioniert in diesem Fall sehr effektiv.

3. Der Speicherbedarf für die Transienten ist beim Browser sehr hoch und macht einen Unterschied von circa 40 gegenüber circa 5 Prozent beim Inspector aus (nach Aktualisierung).

6.3.4 Ein schlechter Fall

Die in Abschnitt 6.3.3 diskutierte Konstellation wird nun derart abgewandelt, daß viele verschiedene Transienten spaltenweise fast gleichzeitig ersetzt werden:

```
X = {MakeTuple big_tuple 30}
Y = {MakeTuple lines 30}
{For 1 30 1 proc {$ I} X.I = Y end}
%% {Inspect X} bzw. {Browse X}
{For 1 30 1 proc {$ I} Y.I = 5 end}
```

Diese Konstellation sieht zunächst harmlos aus, hat es aber in bezug auf den Inspector in sich: Durch das spaltenweise Ersetzen einelner Transienten muß jede Struktur quasi komplett traversiert werden, um die Darstellung zu aktualisieren. Natürlich werden die verbliebenen Transienten jeweils nicht neu gezeichnet, jedoch fällt der Traversierungsoverhead durch die nur atomaren Werte sehr stark ins Gewicht. Die 30 verschiedenen Transientenwächter „feuern“ zudem in kurzer Folge, so daß die Struktur 30-fach angeordnet und gezeichnet werden muß, bevor die engültige Darstellung fertig ist, die mit der von Abschnitt 6.3.3 übereinstimmt. Da die Änderungen stets sehr lokal sind, erscheint das Werzeug dadurch besonders langsam.

Operation	t_I	m_I	t_B	m_B	Δ_T	Δ_M
`X` darstellen	740	159	4590	609	6	4
∀`I: Y.I = 5`	14200	143	9370	366	0.6	2.5

Tabelle 6.8: *Verschiedene Transienten in ganzer Struktur*

Die Zahlen in Tabelle 6.8 ergeben folgendes:

1. Der Inspector ist erstmals circa 40 Prozent langsamer als der Browser – dieses Szenario ist ein schlechter Fall für die Aktualisierungsstrategie des Inspectors.

Allerdings tritt dieser Unterschied nur bei diesem speziellen, sehr schlechten Fall zu Tage. Ändert man das Beispiel leicht ab, indem die Ersetzungen größer werden beziehungsweise dies in Records geschieht, schrumpft der Abstand gewaltig zusammen. Der Inspector überholt den Browser schließlich wieder.

2. Die Aktualisierungsstrategie des Browsers funktioniert definitiv nicht. Denn die Aktualisierungszeiten stimmen mit dem Beispiel aus Abschnitt 6.3.3 fast überein.

3. In bezug auf den Speicherverbrauch ändert sich gegenüber dem Beispiel aus Abschnitt 6.3.3 nichts wesentliches.

6.4 Zusammenfassung

Zwei wesentliche Dinge lassen sich in bezug auf die Evaluierungsergebnisse festhalten:

- Der Inspector ist im Regelfall zwischen Faktor Drei bis Faktor Zehn schneller als der Browser.
- Der Inspector benötigt nur die Hälfte bis ein Drittel soviel Speicher wie der Browser.

Kapitel 7

Verwandte Arbeiten

In diesem Kapitel werden Darstellungssysteme vorgestellt, die mit dem in dieser Arbeit diskutierten System zusammenhängen.

7.1 Der Oz-Browser

Der Oz-Browser ist dieser Arbeit sehr ähnlich, da der Inspector als Nachfolger des Browsers konzipiert ist. Der Browser kann also ebenfalls Oz-Werte und deren Evolution darstellen. Er unterscheidet sich aber in einigen, wesentlichen Punkten vom Inspector, die jetzt in Kürze dargestellt werden.

7.1.1 Anordnungsverfahren

Das Anordnungsverfahren fällt ebenfalls unter die Kategorie Pretty-Printing, allerdings kommt ein einrückungsbasiertes Zeilenumbruchverfahren zum Einsatz. Ein solches versucht, eine vorgegebene maximale Textbreite nicht zu überschreiten. Die damit erzeugten Darstellungen werden zwar in der Regel sehr kompakt, jedoch ergeben sich zwei Nachteile:

1. Die Struktur des Wertes wirkt bisweilen ziemlich unübersichtlich. Denn gleiche Teilbäume müssen je nach Position in der Darstellung unterschiedlich dargestellt werden, um in die vorgegebene Darstellungsbreite zu passen (siehe Abschnitt 3.1.2).

2. Eine Aktualisierung der Darstellung kann teurer sein als nötig: Eventuell müssen auch unveränderte Teilbäume bearbeitet werden, um die Darstellungsbreite nicht zu überschreiten.

   Insbesondere bei größeren Datenstrukturen, die eine für die Darstellungbreite „ungeeignete“ Struktur aufweisen, kann sich dieser Aspekt verheerend auswirken (siehe voriges Kapitel).

7.1.2 Filtersysteme

Da der Browser im Standardfall eine Baumdarstellung von Werten erzeugt, muß eine endliche Zyklenauffaltung sichergestellt sein. Dazu verfügt er wie der Inspector über Traversierungsfilter, deren Limits sich ebenfalls konfigurieren lassen.

Der Browser verfügt über keinen Typtransformator im Sinne des Inspectors.

Anstelle eines allgemeinen Relationsmodus, verfügt der Browser über die Möglichkeit, den Baum als (a) Pseudo-Graph beziehungsweise (b) minimalen Pseudo-Graph darzustellen. Dies entspricht in etwa dem Einsetzen (a) struktureller und (b) knotenbasierter Gleichheit beim Inspector. In bezug auf Listen ist die Darstellung nicht immer minimal, wenn eine normalisierte Liste vorliegt.

7.1.3 Interaktionsverhalten

Der Browser unterstützt eine „Vorform“ von knotenbasierter Interaktion. Diese erlaubt zwar, Teile der Darstellung zu verändern. Dies geschieht jedoch ziemlich umständlich unter Benutzung des globalen Konfigurationsdialoges.

7.1.4 Konfigurierbarkeit

Die Konfigurierbarkeit des Browsers in bezug auf die Darstellung beschränkt sich auf einige, statisch vorgegebene Optionen. Neben den Limits von Traversierungsfiltern, können festgelegte alternative Darstellungen einiger Typen, etwa virtuellen Strings und Chunks, ausgewählt werden.

7.2 Ein Pretty-Printer-Generator

Hughes [13] und Wadler [32] stellen jeweils eine Bibliothek vor, die die schnelle Konstruktion von Pretty-Printern erlaubt.

Beide Systeme basieren auf algebraischen Operatoren, die die Anordnungsmöglichkeiten definieren und benutzerseitig konfiguriert werden können.

Um nun eine Darstellung zu erzeugen, muß ein Text durch Operatoren „annotiert“ werden. Daraus liefert der Pretty-Printer dann das eigentliche Ergebnis. Wie beim Browser, wird die Darstellung ebenso in eine maximale Textbreite gepreßt. Damit ergeben sich die gleichen Probleme in bezug auf die Lesbarkeit der Darstellung.

Zudem ist der so generierte Pretty-Printer noch nicht darauf vorbereitet, Teile der Darstellung inkrementell zu aktualisieren. Denn die Texte werden immer am Stück erzeugt.

Die Idee, die Anordnung der Werte nicht direkt zu erzeugen, wurde auch von Bernhard Latz in seinem Browser [17] aufgegriffen. Zunächst wird ein Strom von Operatoren erzeugt, die dann interpretiert werden, um das Ergebnis zu produzieren.

7.3 Abstrakte Graphsysteme

Einige Eigenschaften, die der Inspector aufweist, sind von Graphdarstellungssystemen wie dem Prototyp von Henry [8], Da Vinci [6] oder Latour [11] inspiriert worden.

7.3.1 Prinzipieller Aufbau

Die meisten Graphsysteme funktionieren nach dem gleichen Prinzip, das auf zwei wesentlichen Komponenten beruht:

1. Einem Fundus verschiedener Anordnungsverfahren für Graphen [1], mit denen eine Darstellung erzeugt werden kann.
2. Erforschung der Darstellung durch interaktive Manipulation beziehungsweise Selektion.

Die verschiedenen Systeme unterscheiden sich nun darin, welche Anordnungsverfahren dem Fundus angehören und in welcher Form die selektive Darstellung erfolgen kann.

Letztere erfolgt zumeist nach dem Muster, daß entweder Teilbäume oder beliebige Teile der Darstellung ausgewählt werden, auf denen dann eine Operation erfolgt. Dies könnte die Darstellung in einem anderen Fenster mit anderem Anordnungsverfahren sein, oder aber das Wegblenden zur besseren Übersicht.

Anstatt trickreiche Anordnungsverfahren wie etwa Fischaugen-Ansichten [25] zu verwenden, setzt der Inspector den Typtransformator ein. Dieser erreicht ähnliche Flexibilität und baut auf der Tatsache auf, daß der Inspector den darzustellenden Oz-Wert einfach und reversibel modifizieren kann.

7.3.2 Konfigurierbarkeit

Sowohl der Prototyp von Henry als auch Da Vinci besitzen eine Schnittstelle zur Programmiersprache C. Über diese werden die Graph-Beschreibungen geliefert. Darüber hinaus erlaubt die Schnittstelle auch, an interaktive Ereignisse eigene Behandler zu klinken. Damit kann der Programmierer weitgehend auf die Erscheinungsform des Systems Einfluß nehmen. Er entscheidet also, welche Möglichkeiten dem Benutzer interaktiv zur Verfügung stehen.

Ein gutes Beispiel für eine Anwendung der Konfigurierbarkeit ist der Oz Investigator [23]. Dessen grafische Darstellung wird zum großen Teil von Da Vinci erzeugt. Da Vinci wurde dazu applikationsspezifisch angepaßt.

Die Darstellungseinheit des Inspectors läßt sich für andere Oz Applikationen in ähnlicher Weise verwenden.

Kapitel 8

Zusammenfassung

Im Rahmen dieser Arbeit wurden Konzept, Entwurf und Implementierung des Inspectors vorgestellt und dessen Effizienz im Vergleich zum Browser evaluiert.

8.1 Zentrale Ideen

Idee. Der Inspector verwendet zur Erzeugung der Darstellung einen zweistufigen Ansatz. Dieser zweistufige Ansatz besteht darin, wenige, aber dafür sehr effiziente Basisdienste zu verwenden. Darauf aufbauend, wird ein flexibler Transformationsmechanismus eingesetzt.

Darstellung. Die Basisdienste des Inspectors halten wichtige Darstellungskriterien ein. Dadurch wird gute Lesbarkeit der Darstellung garantiert. Entscheidend dabei ist die Teilbaumisomorphie. Das heißt, gleiche Teilbäume werden stets gleich dargestellt. Hinzu kommt das Stabilitätskriterium für die Aktualisierungsmechanismen. Beide zusammen bilden die Grundlage für die Inkrementalität und damit für die Geschwindigkeit des Inspectors.

Flexibilität. Durch den Transformationsmechanismus werden viele Darstellungsentscheidungen von der Implementierung weg auf den Benutzer übertragen. Er legt fest, was wie dargestellt wird. Daraus resultiert eine kompakte und zugleich flexible Implementierung.

Die vorliegende Implementierung realisiert bereits sinnvolle Vorgaben für den Transformationsmechanismus. Dadurch ist dieser für den Benutzer auch ohne spezielle Kenntnisse sofort einsetzbar.

Interaktion. Der Benutzer kann direkt mit der Darstellung des Inspectors interagieren. Die Darstellung läßt sich so schnell und flexibel den aktuellen Wünschen anpassen. Darüber hinaus kann man verschiedene Sichten auf die gleichen Daten erhalten.

8.2 Implementierung

Die vorliegende Implementierung des Inspectors ist hochmodular und kompakt. Sie besteht aus zwei Teilen, der Benutzeroberfläche des Inspectors und der Darstellungseinheit. Letztere leistet die eigentliche Wertdarstellung und kann völlig unabhängig vom Inspector in andere Applikationen eingebunden werden. Der Codeumfang ist mit insgesamt 8000 Zeilen, verteilt auf 3000 beziehungsweise 5000 Zeilen, gegenüber dem Browser mit 15000 Zeilen nur etwa halb so groß.

Das Modulsystem von Oz erwies sich während der Entwicklung des Inspectors als sehr nützlich, da es getrennte Kompilierung ermöglicht. Mit etwa elf Modulen wird davon auch reichlich Gebrauch gemacht. Interessant dabei war, daß sich die Modularisierung nicht als Hindernis für die erreichbare Ausführungsgeschwindigkeit entpuppte.

Neben der Verwendung von Modulen, werden konsequent Klassen mit Mehrfachvererbung zur Kapselung logischer Einheiten eingesetzt. Die Struktur der Implementierung wird dadurch extrem übersichtlich und auch durch Dritte wartbar.

Darüber hinaus macht die Verwendung von aktiven Objekten die Architektur in bezug auf Nebenläufigkeit einfach und sicher.

Die aktuelle Implementierung von Oz, Mozart [4], kann um eigene Datentypen erweitert werden. Die vorliegende Implementierung der Darstellungseinheit kann diese neuen Datentypen ohne Eingriff in den Kern „richtig“ darstellen. Dazu muß lediglich benutzerseitig eine entsprechende Transformationsfunktion definiert werden.

8.3 Evaluierung

Die Geschwindigkeit der vorliegenden Implementierung ist sehr gut: Der Inspector zwischen dreimal und zehnmal schneller als der Browser. Dieses Ergebnis macht deutlich, daß neben der Verfügbarkeit effizienter Basisdienste, die Inkrementalität der Aktualisierungsmechanismen unabdingbar für die Gesamteffizienz des Darstellungswerkzeuges ist. Darüber hinaus benötigt der Inspector je nach Situation nur die Hälfte bis ein Drittel des Speichers, den der Browser beansprucht.

Der Inspector ist damit in der Lage, auch größere Datenstrukturen ohne Probleme darzustellen.

8.4 Ausblick

Der Inspector wurde mit dem Ziel konstruiert, schnell und flexibel zu sein. Leider ist das derzeit von Oz verwendete Grafiksystem Tcl/Tk nicht besonders schnell, vergleicht man es mit modernen Alternativen. In absehbarer Zeit steht jedoch eine neue Grafikanbindung zur Verfügung, die auf GTK+ zurückgreift und wesentlich schneller sein soll [29].

Damit ergibt sich die Gelegenheit, die Faktorisierung der Implementierung in be-

zug auf die Darstellungseinheit zu überprüfen, indem dieses ein neues Grafikmodul erhält. Hält die Faktorisierung, kommt der Inspector so möglicherweise in den Genuß einer weiteren Geschwindigkeitssteigerung, ohne daß weitere Änderungen im Kern notwendig wären.

Literaturverzeichnis

[1] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: an annotated bibliography. Technical report, Department of Computer Science, Brown University, Juni 1994.

[2] Anthony Bloesch. Aesthetic Layout of Generalized Trees. *Software - Practice and Experience*, 23(8):817–827, August 1993.

[3] R. F. Cohen, G. Battista, R. Tamassia, and I. G. Tollis. A Framework for Dynamic Graph Drawing. Technical Report CS-92-34, Department of Computer Science, Brown University, Box 1910, Providence, RI 02912, August 1992.

[4] Mozart Consortium. The Mozart programming system, 1999. Verfügbar unter `http://www.mozart-oz.org/index.html`.

[5] Cheng Ding and Prabaker Mateti. A Framework for the Automated Drawing of Data Structure Diagrams. *IEEE Transactions on Software Engineering*, 16(5), Mai 1990.

[6] M. Fröhlich and M. Werner. The Graph Visualization System *da vinci* — A User Interface for Applications. Technical Report 5/94, Department of Computer Science, University of Bremen, Germany, September 1994.

[7] Jeremy Gibbons. Deriving Tidy Drawings of Trees. Report CDMTCS-003, Centre for Discrete Mathematics and Theoretical Computer Science, University of Auckland, Auckland, New Zealand, Juni 1995.

[8] Tyson Rombauer Henry. *Interactive Graph Layout: The Exploration of Large Graphs*. PhD thesis, Department of Computer Science, University of Arizona, Tucson, USA, Juni 1992.

[9] Martin Henz. *Objects in Oz*. Dissertation, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, Juni 1997.

[10] Ivan Herman, Maylis Delest, and Guy Melancon. Tree Visualisation and Navigation Clues for Information Visualisation. Report INS-R9806, Information Systems (INS), Centrum voor Wiskunde en Informatica (CWI), Amsterdam, Netherlands, März 1998.

[11] Ivan Herman, Guy Melancon, M. M. de Ruiter, and Maylis Delest. Latour - a tree visualisation system. Report INS-R9904, Information Systems (INS),

Centrum voor Wiskunde en Informatica (CWI), Amsterdam, Netherlands, April 1999.

[12] Andy Hoyler. Methods For Evaluating User Interfaces. Research Report 301, School of Cognitive and Computing Science, University of Sussex, Brighton, UK, November 1993.

[13] John Hughes. The Design of a Pretty-printing Library. In J. Leuring and E. Meijer, editors, *Advanced Functional Programming*, Lecture Notes in Computer Science, vol. 925. Springer-Verlag, Berlin, 1995.

[14] Ari Juutistenaho. Linear Time Algorithms for Layout of Generalized Trees. Technical Report A-1994-6, Department of Computer Science, University of Tampere, Tampere, Finland, März 1994. revised version will appear in Fundamenta Informaticae.

[15] Andrew J. Kennedy. Drawing Trees. *Journal of Functional Programming*, 6(3):527–534, Mai 1996.

[16] Harsha P. Kumar, Catherine Plaisant, and Ben Shneiderman. Browsing Hierarchical Data with Multi-Level Dynamic Queries and Pruning. *International Journal of Human-Computer Studies*, 46(1):103–124, Januar 1997.

[17] Bernhard Latz. Eine Benutzerschnittstelle für Oz. Diplomarbeit, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, 1994.

[18] Benjamin Lorenz. Ein Debugger für Oz. Diplomarbeit, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, April 1999.

[19] Michael Mehl. *The Oz Virtual Machine - Records, Transients, and Deep Guards.* Dissertation, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, Mai 1999.

[20] Michael Mehl, Ralf Scheidhauer, and Christian Schulte. An Abstract Machine for Oz. In Manuel Hermenegildo and S. Doaitse Swierstra, editors, *Programming Languages, Implementations, Logics and Programs, Seventh International Symposium, PLILP'95*, volume 982 of *Lecture Notes in Computer Science*, pages 151–168, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[21] Michael Mehl, Christian Schulte, and Gert Smolka. Futures and By-need Synchronisation for Oz, Mai 1998. Verfügbar unter `http://www.ps.uni-sb.de/~smolka/drafts/oz-futures.ps`.

[22] Johan Montelius. *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages.* PhD thesis, SICS Swedish Institute of Computer Science, SICS Box 1263, S-164 28 Kista, Sweden, April 1997. SICS Dissertation Series 25.

[23] Tobias Müller. Practical Investigation of Constraints with Graph Views. In Konstantinos Sagonas and Paul Tarau, editors, *Proceedings of the International Workshop on Implementation of Declarative Languages (IDL'99)*, September 1999.

[24] Konstantin Popov. An Exercise in Concurrent Object-Oriented Programming: The Oz Browser. In *Proceedings of WOz'95, International Workshop on Oz Programming*, pages 101–108, Institut Dalle Molle d'Intelligence Artificielle Perceptive, Martigny, Switzerland, November 1995.

[25] Manojit Sarkar and Marc H. Brown. Graphical Fisheye Views. Technical Report CS-93-40, Department of Computer Science, Brown University, Box 1910, Providence, RI 02912, März 1993.

[26] Ralf Scheidhauer. *Design, Implementierung und Evaluierung einer virtuellen Maschine für Oz.* Dissertation, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, Dezember 1998.

[27] Christian Schulte. Oz Explorer: A Visual Constraint Programming Tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, 1997. The MIT Press.

[28] Aaron Scott. A Survey of Graph Drawing Systems. Technical Report 95-1, Department of Computer Science, University of Newcastle, Callaghan 2308, Australia, Dezember 1994.

[29] Andreas Simon. Gtk+ – eine effiziente Grafikanbindung für Oz. Diplomarbeit, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, 2000. Vorläufiger Titel, in Vorbereitung.

[30] Gert Smolka. The Definition of Kernel Oz. In Andreas Podelski, editor, *Constraint Programming: Basic and Trends*, Lecture Notes in Computer Science, vol. 910, pages 251–292. Springer-Verlag, Berlin, Germany, 1995.

[31] Gert Smolka. The Oz Programming Model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, Germany, 1995.

[32] Philip Wadler. A prettier printer. *Journal of Functional Programming*, 1999. to appear.