Diplomarbeit

Generische Daten– und Funktionsvisualisierung in einer persistenten Programmierumgebung

vorgelegt von
Hubertus Koehler
Tarpenbekstraße 62
20251 Hamburg

Betreuer:
Prof. Dr. Joachim W. Schmidt
Prof. Dr. Leonie Dreschler–Fischer

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Datenbanken und Informationssysteme

28. März 1996

Inhaltsverzeichnis

1. Einführung und Überblick **1**
- 1.1 Ausgangspunkt der Arbeit und Problemstellung . . . 2
- 1.2 Lösungsansatz und Gliederung der Arbeit . . . 5

2. Das Tycoon System **7**
- 2.1 Architektur des Tycoon Systems . . . 7
- 2.2 Das semantische Modell von TL . . . 10
  - 2.2.1 Bindungen und Signaturen . . . 11
  - 2.2.2 Benennung und Typisierung vordefinierter Werte und Funktionen . . . 11
  - 2.2.3 Typhierarchien . . . 12
  - 2.2.4 Struktur– und Namensäquivalenz . . . 13

3. Dienste zur Datenvisualisierung **15**
- 3.1 Ziele der Visualisierung . . . 15
  - 3.1.1 Zu visualisierende Informationen . . . 15
  - 3.1.2 Funktionalität . . . 17
- 3.2 Visualisierungsaspekte . . . 19
  - 3.2.1 Grafische Benutzerschnittstelle . . . 19
  - 3.2.2 Visualisierung von Objekten . . . 21
  - 3.2.3 Sichtfokussierte versus objektfokussierte Umgebungen . . . 22
- 3.3 Implementierungsaspekte . . . 23
  - 3.3.1 Bedeutung einer dynamischen Typisierung . . . 23
  - 3.3.2 Ausnutzung von Objektformaten auf Speicherebene . . . 24
  - 3.3.3 Implementierung durch linguistische Reflektion . . . 24
- 3.4 Ansatz zur Datenvisualisierung im Tycoon System . . . 26

4. Grafische Gestaltung der Daten– und Funktionsvisualisierung **28**
- 4.1 Die grafische Klassenbibliothek StarView . . . 28
  - 4.1.1 Die StarView Klassenhierarchie . . . 29
  - 4.1.2 Ereignisbearbeitung . . . 31
  - 4.1.3 Einbindung der Klassenbibliothek in das Tycoon System . . . 31
  - 4.1.4 Erfahrungen mit der Einbindung . . . 33
- 4.2 Bildschirmrepräsentationen . . . 34
  - 4.2.1 Rahmen für visualisierte Objekte . . . 34
  - 4.2.2 Typen . . . 35
  - 4.2.3 Werte . . . 38

4.2.4 Funktionswerte 40

5. Dynamische Typisierung 42
5.1 Laufzeittyprepräsentationen und automorphe Werte 42
5.2 Anwendungen dynamischer Typisierung 43
5.2.1 Meta–Level Operationen 44
5.2.2 Strukturierte Ein– und Ausgabe 44
5.2.3 Generatoren 45
5.2.4 Bindungen an entfernte Prozeduraufrufe 46
5.2.5 Aufgaben dynamischer Typisierung 46
5.3 Konzepte zur Realisierung dynamischer Typisierung 47
5.3.1 Dynamische Typisierung in anderen Sprachen 47
5.3.2 Dynamische Typisierung und Polymorphismus 50
5.3.3 Dynamische Typisierung abstrakter Datentypen 51
5.4 Realisierung dynamischer Typisierung in TL 52
5.4.1 Ursprünglicher Ansatz 52
5.4.2 Spracherweiterungen in TL 54
5.4.3 Programmierschnittstellen 55
5.4.4 Erweiterungen des TL Compilers 57

6. Linguistische Reflektion zur Unterstützung der Visualisierung 64
6.1 Laufzeittyprepräsentationen für die globalen Bindungen einer Funktion 64
6.1.1 Funktionsabschluß 64
6.1.2 Erweiterter Funktionsabschluß 66
6.1.3 Generierter Programmkode 68
6.1.4 Erweiterungen des Typüberprüfers 70
6.1.5 Praktische Erfahrungen mit dem Ansatz 70
6.2 Persistente Threads 71
6.2.1 Interne Darstellung 72
6.2.2 Generierter Programmkode 72

7. Realisierung der Daten– und Funktionsvisualisierung 77
7.1 Die Bibliothek zur Daten– und Funktionsvisualisierung 77
7.2 Strukturanalyse 80
7.3 Abstraktes Datenstrukturmodell 81
7.3.1 Protokoll 81
7.3.2 Zugriffsfunktionen 83
7.4 Unterstützende Schnittstellen 84
7.4.1 Allgemeine unterstützende Schnittstellen 84
7.4.2 Verwaltung der visualisierten Objekte 84
7.4.3 Generierung der Bildschirmrepräsentation 85
7.4.4 Unterstützung des Datenaustauschs 86
7.5 Visualisierungsschnittstelle 87
7.5.1 Typvisualisierung 87
7.5.2 Wertvisualisierung 88
7.5.3 Funktionsvisualisierung 89
7.6 Schnittstelle zum Anwendungsprogramm 91

8. Zusammenfassung und Ausblick 93
8.1 Dynamische Typisierung 93
8.2 Übersetzungszeitreflektive Unterstützung der Visualisierung 94
8.3 Datenvisualisierung und Datenmanipulation 94
8.4 Ausblick 95

A. Schnittstellen des Compilers zur Implementierung dynamischen Typisierung 98
A.1 Schnittstelle 'TLDynamicImpl' 98
A.2 Schnittstelle 'TLDynamic' 100
A.3 Schnittstelle 'TLDynEnv' 100

B. Reflektive Schnittstellen 101
B.1 Schnittstelle 'TypeRep' 101
B.2 Schnittstelle 'Dynamic' 104
B.3 Schnittstelle 'Closure' 107

C. Ausgewählte Schnittstellen der generischen Daten– und Funktionsvisualisierung 108
C.1 Schnittstelle 'BrowserExchange' 108
C.2 Schnittstelle 'Browser' 109

Abbildungsverzeichnis

1.1 Unterschiedliche Zustände eines benutzerdefinierbaren Editors 3

2.1 Interoperabilität im Tycoon System 8
2.2 Ursprüngliche Tycoon Architektur 9
2.3 Erweiterte Tycoon Architektur mit dynamischen Typen 10
2.4 Verschiedene Typhierarchien in TL 12

3.1 Objektfokussierte vs. sichtfokussierte Darstellung 22
3.2 Arten von Reflektion 25

4.1 Ausschnitt aus der StarView Klassenhierarchie 30
4.2 Anbindungsszenario für StarView 32
4.3 Mögliche Komponenten eines Rahmens 34
4.4 Abstrakte Datentypen 36
4.5 Strukturierte Typen 37
4.6 Funktionstypen 37
4.7 Typoperatorapplikationen 37
4.8 Rekursive Typen 38
4.9 Integerwerte 38
4.10 Fließkommazahlen 39
4.11 Zeichenketten 39
4.12 Boolesche Werte 39
4.13 Werte abstrakter Datentypen 40
4.14 Strukturierte Werte 40
4.15 Felder 41
4.16 Programmkode und globale Variablen einer Funktion 41

5.1 Säulen der generischen Programmierung in TL 44
5.2 Kopplung des API für dynamische Typen an den Compiler 56
5.3 Typrepräsentation mit Referenzen 59

6.1 Abschluß einer Funktion 65
6.2 Funktionsabschluß mit Typinformationen für globale Bindungen 67
6.3 TML Repräsentation eines Threads 73
6.4 Funktionsabschluß mit Laufzeittypinformationen für alle Bindungen 73
6.5 Funktionsabschluß nach der Definition von f 75

7.1 Ablauf der Daten- und Funktionsvisualisierung 78

7.2 Struktur der Bibliothek zur Daten– und Funktionsvisualisierung 79
7.3 Mögliche Zustandsübergänge der Browser 83
7.4 Beispiel für die Manipulation varianter Tupel 89

1. Einführung und Überblick

Ausgehend von unterschiedlichen Entwicklungen und Anforderungen stand in den letzten Jahren die Erhöhung der Produktivität im Entwicklungsprozeß im Mittelpunkt der Aktivitäten im Bereich integrierter, datenintensiver Anwendungen. Dabei lassen sich vier Schwerpunkte ausmachen: die Überwindung von Inkompatibilitäten zwischen Datenbankmodellen und Programmiersprachen, der Entwurf geeigneter Architekturen für Datenbankprogrammierumgebungen, die persistente Manipulation ausführbarer Programme sowie die Manipulation und Migration persistenter Kontrollflüsse [Matthes 93; Mathiske et al. 95a].

Zwischen Datenbankmodellen und algorithmisch vollständigen Programmiersprachen lassen sich auf zwei Ebenen Unverträglichkeiten feststellen. Einerseits existiert eine Diskrepanz zwischen mengenorientierter, deklarativer Datenverarbeitung in Datenbanksystemen und dem elementorientierten, prozeduralen Paradigma konventioneller Programmiersprachen, der als *impedence mismatch* bezeichnet wird. Andererseits gibt es Inkompatibilitäten zwischen den Konzepten in Datenbankmodellen und Sprachen zur Anwendungsprogrammierung. Zur Überwindung dieser Schwierigkeiten wurden spezielle Datenbankprogrammiersprachen entwickelt. Sie bieten linguistische Unterstützung auf den Ebenen der Persistenzabstraktion, indem flüchtige und langlebige Daten uniform behandelt werden, der Typvollständigkeit, wobei besonders die Massendatentypen berücksichtigt werden, und der Iterationsabstraktion [Matthes 93].

Getrieben durch technologische und marktwirtschaftliche Entwicklungen haben sich neuartige Informationsstrukturen, wie z.B. multimediale Daten, in grundlegend veränderten, d.h. interaktiven, vernetzten, verteilten und teilautonomen, Informationssystemen entwickelt [Blaser 90; Cattell 91]. Die Unterstützung dieser neuartigen Anforderungen durch unterschiedliche Systeme und Werkzeuge zur Erbringung generischer Dienste in einer integrierten Datenbankprogrammierumgebung erfordert die Bereitstellung geeigneter Architekturen.

Eine bessere Erfassung der Semantik datenintensiver Anwendungen erfordert zusätzlich zu statischen Daten die Beschreibung dynamischer Prozesse [Matthes, Schmidt 94]. Das führte zu der Entwicklung persistenter Programmiersprachen wie **PS–algol** [Atkinson et al. 81], **Napier88** [Dearle et al. 89], **Fibonacci** [Albano et al. 95] und **Tycoon**[1] [Matthes 95]. Basierend auf polymorphen Typsystemen[2] unterstützen sie durch die Einführung des Konzeptes der orthogonalen Persistenz die Langlebigkeit beliebigstrukturierter Datenobjekte. Indem auch Funktionen als Werte erster Klasse behandelt werden, die als Argumente anderen Funktionen übergeben, als Ergebnis einer Funktionsanwendung zurückgegeben und in persistenten Datenstrukturen gespeichert werden können, ist die persistente Manipulation von ausführbaren bzw. bindefähigen Programmen in einer gemeinsamen persistenten Umgebung für alle Werte möglich.

---

[1] Typed communicating objects in open environments

[2] Zur Einfühung in dieses Gebiet s. [Cardelli, Wegner 85]

Die Neustrukturierung betrieblicher Prozesse in Unternehmen schließlich, die zur Automatisierung und effizienteren Gestaltung rechnergestützter Vorgangsbearbeitung durch die Entwicklung von Workflows [Jablonski 94] und Workflowsmanagementsystemen (s. z.B. [Jablonski 95]) führte, erfordert die Behandlung der Unverträglichkeiten zwischen verteilter Programmierung, Transaktionsmanagement und Workflowmanagement [Mathiske et al. 95a]. Durch die Generalisierung des Konzeptes von Threads bilden persistente und migrierende Threads als Werte erster Klasse die Konstruktionsblöcke zur Realisierung langlebiger und verteilter kooperativer Aktivitäten in persistenten Programmierumgebungen.

1.1 Ausgangspunkt der Arbeit und Problemstellung

Die Vereinigung der oben aufgeführten Eigenschaften in einer integrierten persistenten Programmierumgebung wie Tycoon, in der die im Rahmen dieser Arbeit vorgestellten Konzepte implementiert werden, stellt neuartige Anforderungen an die Softwareentwicklung, die der Werkzeugunterstützung bedürfen. In relationalen (z.B. Pascal/R [Schmidt 77], DBPL [Matthes, Schmidt 93a]) und objektorientierten (z.B. O_2C für O_2 [Bancilhon et al. 92], O++ für ODE [Agrawal, Gehani 89]) Datenbankprogrammiersprachen wird das Data–Engineering durch interaktive Browser und andere 4GL–Werkzeuge zur Konstruktion, Visualisierung und Manipulation von Daten in vielfältiger Weise unterstützt. Die Möglichkeit der persistenten Manipulation ausführbarer Programme erlaubt es in persistenten Programmiersprachen, auch Entwurfsanwendungen wie das Software–Engineering als datenintensive Anwendung zu betrachten. Dementsprechend sollten grafische Werkzeuge den Softwareentwicklungsprozeß in allen Phasen unterstützen:

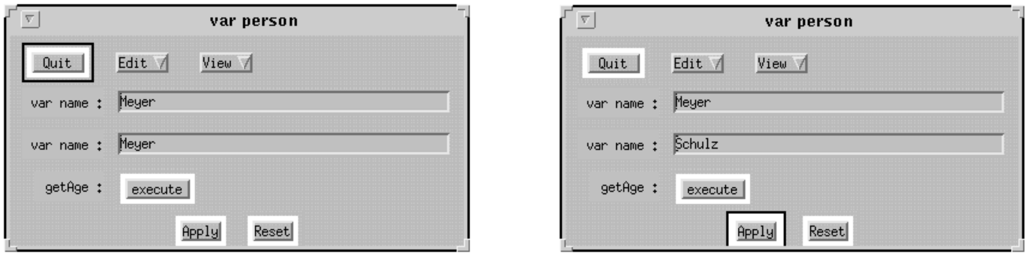
Konstruktion: Da der Programmierer zur Konstruktion von Programmen Kenntnis der im Speicher vorhandenen Werte und Programme, ihrer Zugriffspfade, ihrer Typstruktur und ihrer Beziehungen untereinander benötigt, muß er den Inhalt und die Topologie des Speichers visualisieren und entlang der Bildschirmrepräsentationen durch den Speicher navigieren können.

Manipulation: Der Anwender braucht Unterstützung zur direkten grafischen Manipulation existierender Werte, da die Eingabe textueller Anweisungen kompliziert, umfangreich und fehlerträchtig sein kann.

Programmausführung: Das Verständnis der Funktionalität komplexer Programme erfordert neben ihrer Visualisierung auch eine Analyse ihres Verhaltens. Deshalb muß der Programmierer Funktionen interaktiv mit unterschiedlichen Parametern und veränderten globalen Variablen ausführen können.

Fehlersuche: Zur effizienten Fehlersuche in Programmen benötigt der Programmierer Werkzeugunterstützung durch einen sogenannten *Debugger*, der u.a. die Visualisierung und Manipulation der an einer Berechnung beteiligten Werte unterstützt.

Ergebnisvisualisierung: Um zu überprüfen, ob eine Berechnung korrekt und erfolgreich beendet worden ist, muß der Anwender das Ergebnis einer Programmausführung visualisieren können.


var person
var name : Meyer
var name : Meyer
getAge :

var person
var name : Meyer
var name : Schulz
getAge :

Abbildung 1.1: Unterschiedliche Zustände eines benutzerdefinierbaren Editors

Dienstauswahl: In einem gemeinsamen offenen Dienstmarkt, wie er in [Merz, Lamersdorf 93] beschrieben ist, kann die Anwendungsentwicklung von existierenden Diensten profitieren, die als Konstruktionsblöcke für die Realisierung individueller integrierter Anwendungen dienen. Bei der Auswahl geeigneter Dienste benötigt der Anwendungsprogrammierer Unterstützung, die in einem unifizierenden sprachlichen Rahmen, wie ihn **Tycoon** bietet, durch die Visualisierung der Schnittstellen erfolgen kann.

Zur Realisierung der Aufgaben in den unterschiedlichen Phasen der Sofwareentwicklung ist die grafische Visualisierung der Daten, d.h. statische Daten, Funktionen und persistente Threads, notwendig. Für das **Tycoon** System wurde in [Müßig 94] ein modellneutraler Datenbankbrowser zum Anzeigen und Verändern von sowie zum Navigieren in den komplexen Datenstrukturen des Objektspeichers realisiert. Genaugenommen handelt es sich dabei um eine Bibliothek benutzerdefinierbarer Editoren, mit deren Hilfe der Anwendungsprogrammierer die Visualisierung von Werten der **TL**[3] Wertkonstruktoren sowie von Listen, Tabellen und Iteratoren realisieren kann. Zur Unterstützung aller Phasen des Softwareentwicklungsprozesses ist die Funktionalität dieser Bibliothek jedoch nicht ausreichend. Die wesentlichen Beschränkungen sollen anhand des folgenden **TL** Programmfragments illustriert werden, das die im linken Teil der Abbildung 1.1 dargestellte Bildschirmrepräsentation erzeugt:

```
let var person = record
  let var name = "Meyer"
  let getAge() = 25
end

let var continue = true
let quit() = continue := false

let name1Editor = editor.newString("var name"  let get() = person.name
                                   let set(new :String) = person.name := new)
let name2Editor = editor.newString("var name"  let get() = person.name
                                   let set(new :String) = person.name := new)
let getAgeEditor = editor.newFunction("getAge" "execute" person.getAge)
let personEditor = editor.newRecord("var person"
                                    iter.enum of name1Editor name2Editor getAgeEditor end)
editor.display(personEditor quit editor.somewhere)
```

---

[3] Tycoon Language

Generik: Die Funktionen der Bibliothek erlauben es dem Programmierer, komponentenweise Editoren zu definieren und zu strukturierten Editoren zu aggregieren. Da der Programmierer für die korrekte Aggregation der Editoren entsprechend der Struktur des zu visualisierenden Wertes verantwortlich ist, kann z.B. die Komponente *person.name* in der Bildschirmrepräsentation von *person* zweimal dargestellt werden. In einer persistenten Programmierumgebung kann der Programmierer aber nicht die Struktur aller im Speicher vorhandenen Daten kennen, da der Speicher inkrementell in unabhängigen Sitzungen durch Manipulationen möglicherweise unterschiedlicher Anwender gewachsen ist. Um die Erkundung der Inhalte und der Topologie des Speichers zu ermöglichen, muß die Visualisierung deshalb *automatisch* erfolgen. Das hat zusätzlich den Vorteil, daß sich der zur Visualisierung eines Wertes notwendige Programmkode auf den Aufruf einer generischen, d.h. auf Werte verschiedenen Typs anwendbaren [Cardelli, Wegner 85], Funktion reduziert. Da TL jedoch eine statisch typisierte Programmiersprache ist, in der zur Laufzeit keine Typinformationen mehr vorhanden sind, anhand derer die Struktur eines Wertes bestimmt und die entsprechende Bildschirmrepräsentation generiert werden kann, ist die typsichere automatische Visualisierung inTycoon bisher unmöglich.

Funktionsvisualisierung: Obwohl im Tycoon System Programme bzw. Funktionen Werte erster Klasse und damit persistent sind, wird ihre Visualisierung durch die benutzerdefinierbaren Editoren kaum unterstützt. Lediglich parameterlose Funktionen (s. *person.getValue* im Beispiel) können über ein Interaktionselement dargestellt und aufgerufen werden. Der interaktive Aufruf beliebiger Funktionen ist bisher aufgrund unzureichender reflektiver Fähigkeiten des Tycoon Systems nicht möglich, da dazu zur Laufzeit der Compiler aufgerufen werden muß, um den Funktionsaufruf mit den Aktualparametern zu übersetzen, zu binden und auszuführen. Die Visualisierung der für das Verständnis beliebiger Funktionen wesentlichen Komponenten — die Signatur, der Funktionsrumpf und die globalen Variablen — wird überhaupt nicht unterstützt. Da in Tycoon zur Laufzeit keine Repräsentationen von Typen existieren, ist die Visualisierung von Typen und Funktionssignaturen bisher unmöglich. Die Visualisierung der globalen Variablen einer Funktion hätte analog zu den Komponenten strukturierter Werte durch Aggregation der durch den Programmierer erzeugten Editoren realisiert werden können. Die automatische Visualisierung ist wie für alle anderen Werte auch wegen fehlender Typinformationen zur Laufzeit im Tycoon System bisher nicht realisierbar. Da die Funktionsvisualisierung nicht unterstützt wird, ist auch die Visualisierung der in einem persistenten Thread aktiven Funktionen unmöglich.

Datenmanipulation: Die benutzerdefinierbaren Editoren unterstützen die direkte Datenmanipulation der Bildschirmrepräsentationen variabler Wertbindungen, deren Typ ein Basistyp ist. Dabei wird die Typsicherheit gewährleistet, indem variable Wertbindungen durch Interaktionselemente dargestellt werden, die nur die Visualisierung und Manipulation von Werten eines bestimmten Typs zulassen. Die grafische Manipulation strukturierter Werte, die durch Datenaustausch zwischen zwei Bildschirmrepräsentationen realisiert werden könnte, ist dagegen nicht typsicher möglich, da dazu ein Laufzeittyptest notwendig ist, der aufgrund fehlender Laufzeittypinformationen in TL nicht durchführbar ist. Ein weiteres Problem bei der Datenmanipulation entsteht dadurch, daß aufgrund fehlender Synchronisationsmechanismen Bildschirmrepräsentationen, die den gleichen Wert repräsentieren, zwischenzeitlich unterschiedliche Inhalte haben können (s. z.B. *person.name* im rechten Teil von Abbildung 1.1).

1.2 Lösungsansatz und Gliederung der Arbeit

Ziel dieser Arbeit ist es, Konzepte zur Realisierung einer generischen Daten– und Funktionsvisualisierung am Beispiel der persistenten Programmierumgebung Tycoon vorzustellen, die — integriert in die in [Geisler 95] vorgestellte grafische Entwicklungsumgebung — den gesamten Softwareentwicklungsprozeß unterstützt. Ausgehend von den im letzten Abschnitt beschriebenen Problemen sind die wesentlichen Beiträge zur Umsetzung dieses Zieles

- ▷ die Erweiterung des TL Compilers um ein Konzept zur dynamischen Typisierung, das durch die Bereitstellung von Laufzeitrepräsentationen für Typen die Entwicklung typgesteuerter Anwendungen wie die Objektspeicher– und die Typvisualisierung ermöglicht;
- ▷ die Funktionsvisualisierung, indem neben der Signatur und dem Funktionsrumpf durch systemtechnische Erweiterungen die Darstellung der globalen Variablen unterstützt wird;
- ▷ die grafische Manipulation beliebiger Werte durch direkte Manipulation der und Datenaustausch zwischen den Bildschirmrepräsentationen.

Diese Arbeit gliedert sich in die im folgenden übersichtsartig vorgestellten Kapitel:

Das Tycoon System: Im Anschluß an diese Einführung erfolgt in Kapitel 2 eine Vorstellung des Tycoon Systems. Dem *Ist–Zustand*, wie er sich vor Beginn dieser Arbeit darstellte, wird eine um dynamische Typen, die in Kapitel 5 eingeführt werden, und reflektive Fähigkeiten, die in [Geisler 95] realisiert worden sind, erweiterte Architektur gegenübergestellt, in die die generische Daten– und Funktionsvisualisierung eingeordnet wird. Außerdem werden die Aspekte des semantischen Modells der Programmiersprache TL vorgestellt, die für die Realisierung dynamischer Typisierung und der generischen Visualisierung von Bedeutung sind.

Dienste zur Datenvisualisierung: In Kapitel 3 werden Visualisierungsdienste verschiedener Programmierumgebungen für datenintensive Anwendungen anhand ihrer Ziele, unterschiedlicher Visualisierungsaspekte sowie zweier Implementierungsansätze miteinander verglichen. Da die strikte statische Typisierung von TL die generische Visualisierung erschwert, erfolgt auch eine Diskussion der Vor– und Nachteile gegenüber dynamisch typisierten Programmiersprachen. Als Resultat dieser Analyse wird eine Anforderungsdefinition für die Daten– und Funktionsvisualisierung in Tycoon erstellt.

Grafische Gestaltung der Daten– und Funktionsvisualisierung: Anhand der zu visualisierenden Objekte beschreibt Kapitel 4 die grafische Umsetzung der Anforderungsdefinition, die mit Hilfe der ebenfalls vorgestellten objektorientierten Klassenbibliothek StarView realisiert wird. Die grafische Gestaltung bildet zusammen mit der Anforderungsdefinition die Grundlage der in den Kapiteln 5, 6 und 7 vorgestellten Implementierungskonzepte.

Dynamische Typisierung: Die Voraussetzung zur Realisierung der generischen, d.h. automatischen, Objektspeichervisualisierung in Tycoon bildet das in Kapitel 5 vorgestellte Konzept zur dynamischen Typisierung. Um den Anforderungen unterschiedlicher typgesteuerter Algorithmen an solch ein Konzept gerecht zu werden, und um die Problematik bei der Erweiterung eines statischen Typsystems um dynamische Typisierung zu verdeutlichen, werden verschiedene Anwendungsgebiete dynamischer Typisierung analysiert und unterschiedliche Implementierungsansätze in anderen Programmiersprachen untersucht.

Linguistische Reflektion zur Unterstützung der Visualisierung: Für die Visualisierung der globalen Bindungen einer Funktion sowie des Zustandes persistenter *Threads* ist die dynamische Typisierung nicht ausreichend. In Kapitel 6 werden deshalb zwei Ansätze vorgestellt, die, aufbauend auf der dynamischen Typisierung und unter Ausnutzung der reflektiven Fähigkeiten des Tycoon Systems, deren Visualisierung unterstützen.

Realisierung der Daten– und Funktionsvisualisierung: Unter Verwendung der in den Kapiteln 5 und 6 vorgestellten Systemerweiterungen, wird in Kapitel 7 die Implementierung der generischen Daten– und Funktionsvisualisierung beschrieben. Schwerpunkte bilden dabei ein Protokoll, das einen uniformen Rahmen für die Integration der zu visualisierenden heterogenen Objekte bereitstellt, sowie entsprechend der Zielsetzung dieser Arbeit die Generik, die Funktionsvisualisierung und der Datenaustausch zwischen den Bildschirmrepräsentationen.

Zusammenfassung und Ausblick: Abschließend erfolgen eine Zusammenfassung der wesentlichen Konzepte der dynamischen Typisierung und der generischen Daten– und Funktionsvisualisierung sowie ein Ausblick auf offene Punkte und Anschlußarbeiten.

2. Das Tycoon System

Die integrierte persistente Programmierumgebung Tycoon, die zur Implementierung der Daten– und Funktionsvisualisierung verwendet wird, soll in diesem Kapitel überblicksartig vorgestellt werden. Dazu erfolgt im ersten Abschnitt eine Einführung in die Systemarchitektur, in die die in dieser Arbeit entwickelten dynamischen Typen und der generische Dienst zur Daten– und Funktionsvisualisierung sowie die verwendete externe Klassenbibliothek StarView eingeordnet werden. Im zweiten Abschnitt werden diejenigen Aspekte des semantischen Modells von TL vorgestellt und diskutiert, die eine wichtige Rolle für die Daten– und Funktionsvisualisierung spielen.

Nicht Gegenstand dieses Kapitels ist dagegen eine Einführung in die Syntax der Programmiersprache TL. Zum Verständnis der in dieser Arbeit verwendeten Programmbeispiele sei verwiesen auf [Matthes et al. 94], einer Einführung in TL, und [Mathiske et al. 93], einem Überblick über die praktische Benutzung der interaktiven Systemumgebung und der Bibliotheken.

2.1 Architektur des Tycoon Systems

Die Architektur des Tycoon Systems zeichnet sich durch eine hohe Skalierbarkeit aus, die es erlaubt, in einem einzigen sprachlichen und architektonischen Rahmen Systemimplementierungen zu realisieren, die von einer *standalone* Implementierung im Hauptspeicher auf einem Personal Computer bis zu einer netzwerkfähigen, mehrbenutzerfähigen, optimierenden und persistenten Implementierung reicht. Dabei ist es möglich, Tycoon Anwendungen, die nur eine limitierte Systemfunktionalität benötigen, auf einer effizienten und schlanken Systemversion arbeiten zu lassen, die aber während der Lebenszeit solch einer Anwendung an wachsende operationale Anforderungen angepaßt werden kann [Matthes et al. 95c].

Um die Interoperatibilität zwischen verschiedenen internen und externen Diensten zu ermöglichen, leistet das Tycoon System folgende Beiträge (s. auch Abb. 2.1[1]):

- ▷ Bestehende und neu entwickelte generische Dienste wie z.B. grafische Benutzeroberflächen werden in uniformer Weise als polymorphe Tycoon Bibliotheken integriert, die einen typsicheren Zugriff auf externe Daten und Kode sicherstellen. Innerhalb von TL existieren uniforme Namens–, Typ–, Bindungs– und Lebensdauerregeln z.B. für Bildschirm–, Daten– und Programmobjekte (S,D,P in Abb. 2.1).

- ▷ Integrierte Dienste werden nicht von in Tycoon selbst implementierten Diensten unterschieden, so daß Typsicherheit und Persistenzabstraktion auch für externe Dienste innerhalb des Tycoon Systems systematisch erlangt werden können.

---

[1] entnommen aus: [Matthes et al. 95c]

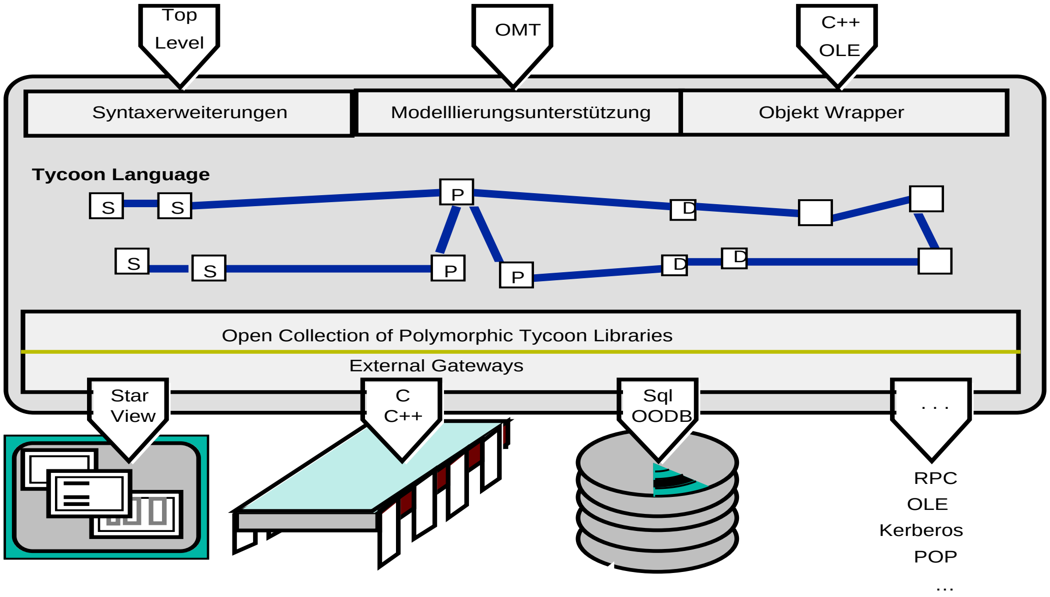

Top Level
OMT
C++ OLE
Syntaxerweiterungen
Modelllierungsunterstützung
Objekt Wrapper
Tycoon Language
S
S
P
D
S
S
P
P
D
D
Open Collection of Polymorphic Tycoon Libraries
External Gateways
Star View
C C++
Sql OODB
. . .
RPC
OLE
Kerberos
POP
...

Abbildung 2.1: Interoperabilität im Tycoon System

- ▹ Das **Tycoon** System kann selbst als Subsystem in größere Systeme integriert werden. So kann beispielsweise **Tycoon** Anwendungskode von Hauptprogrammen aufgerufen werden, die in externen Sprachen wie C geschrieben sind, wie durch die Pfeile am oberen Rand von Abb. 2.1 angedeutet.

In Verbindung mit Abbildung 2.2 erfolgt die Vorstellung der nichtreflektiven Architektur des **Tycoon** Systems anhand der abstrakten Schnittstellenspezifikationen, die strikt die Speicherungs–, Manipulations–, Modellierungs– und Repräsentationsaufgaben voneinander trennt:

TL Schnittstelle: Die Programmiersprache **TL** ist algorithmisch vollständig, strikt typisiert, imperativ und behandelt Funktionen und Typen als Objekte “erster” Klasse. Für alle Typkonstruktoren werden strukturell definierte Subtypisierungsregeln definiert. Da **TL** einen sprachunabhängigen Rahmen für die Übersetzung, Generierung und Bindung von Programmen liefert, dient sie nicht nur der Datenmodellierung und Anwendungsentwicklung, sondern implementiert sich selbst als Systemsprache über Bibliotheken und Sprachprozessoren. Sie ist weitgehend datenmodellneutral, um funktionale, relationale und objektorientierte Modellierungsansätze zu unterstützen. Formal gesehen ist **TL** ein typisierter Lambda Kalkül zweiter Ordnung, der um Zuweisungen, Subtypisierungsregeln sowie im Rahmen dieser Arbeit um dynamische Typen erweitert ist [Cardelli et al. 91].

TML Schnittstelle: Die Zwischenrepräsentation **TML**[2] wird für quellsprachen– und zielmaschinenunabhängige statische (zur Übersetzungszeit) und dynamische (zur Laufzeit)

---

[2] TML: Tycoon Machine Language

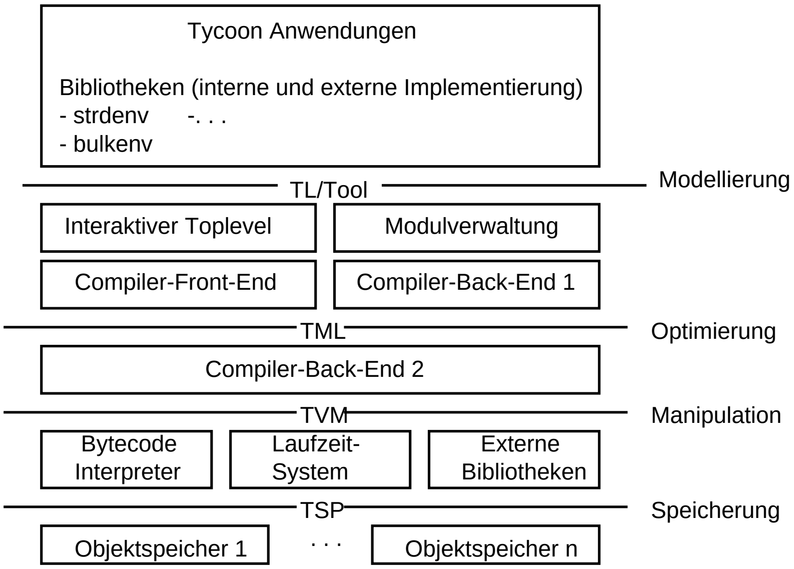

Tycoon Anwendungen
Bibliotheken (interne und externe Implementierung)
- strdenv
-. . .
- bulkenv
TL/Tool
Modellierung
Interaktiver Toplevel
Modulverwaltung
Compiler-Front-End
Compiler-Back-End 1
TML
Optimierung
Compiler-Back-End 2
TVM
Manipulation
Bytecode Interpreter
Laufzeit-System
Externe Bibliotheken
TSP
Speicherung
Objektspeicher 1
. . .
Objektspeicher n

Abbildung 2.2: Ursprüngliche Tycoon Architektur

Optimierungen genutzt [Gawecki, Matthes 95]. TML basiert auf einem untypisierten Lambda Kalkül in *Continuation Passing Style* [Appel 92] und kann für die Implementierung verschiedener Sprachen, wie z.B. TL und TooL [Gawecki, Matthes 96] genutzt werden.

TVM Schnittstelle: TVM[3] ist eine Aufrufschnittstelle, die einen portablen Befehlssatz für Bytekode definiert, der auf einem funktionalen Ausführungsmodell höherer Ordnung basiert. Darüber hinaus bietet TVM einen generischen Mechanismus zur Bindung an übersetzte Funktionsimplementierungen in *externen* Programmiersprachen (z.B. C oder C++), wodurch u.a. externe Dienste wie StarView in Tycoon integriert werden können. Skalierbarkeit wird in dieser Schnittstelle erreicht, indem der TVM Bytekode auf einer virtuellen Maschine interpretiert oder in Zielmaschinenkode übersetzt wird, ein oder mehrere *Threads* gestartet werden können, sowie persistente *Threads* als Werte erster Klasse oder nur eine flüchtige Ausführungsumgebung unterstützt wird.

TSP Schnittstelle: TSP[4] definiert eine datenmodellunabhängige Schnittstelle, die die Manipulation und Visualisierung von Daten von der zuverlässigen Speicherung von Massendaten trennt. Dadurch kann einerseits vollständig von den Prozessen der Zugriffsoptimierung, Speicherrückgewinnung, Persistenz, Nebenläufigkeit, Fehlererholung und Verteilung innerhalb des physischen Speichers abstrahiert werden. Andererseits führt das zu einer erheblichen Reduktion der Komplexität des Objektspeicherprotokolls und leistet einen wichtigen Beitrag zur Portabilität und Skalierbarkeit des Tycoon Systems [Matthes et al. 95a].

---

[3] TVM: Tycoon Virtual Machine

[4] TSP: Tycoon Store Protocol

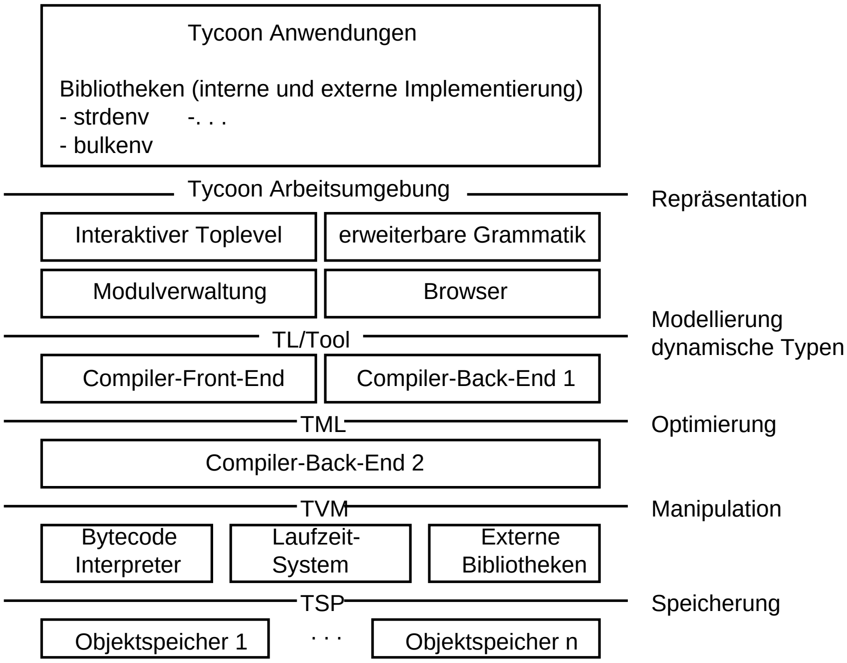

Tycoon Anwendungen
Bibliotheken (interne und externe Implementierung)
- strdenv
-. . .
- bulkenv
Tycoon Arbeitsumgebung
Repräsentation
Interaktiver Toplevel
erweiterbare Grammatik
Modulverwaltung
Browser
Modellierung dynamische Typen
TL/Tool
Compiler-Front-End
Compiler-Back-End 1
TML
Optimierung
Compiler-Back-End 2
TVM
Manipulation
Bytecode Interpreter
Laufzeit-System
Externe Bibliotheken
TSP
Speicherung
Objektspeicher 1
. . .
Objektspeicher n

Abbildung 2.3: Erweiterte Tycoon Architektur mit dynamischen Typen

Alle oberhalb von TVM liegenden Schnittstellen sind in TL selbst implementiert worden, so daß der *bootstrap* des Systems auf unterschiedlichen Hard– und Softwarearchitekturen erleichtert wird. Durch die Erweiterung von Tycoon um dynamische Typen ist die Einführung einer weiteren Schnittstelle möglich, die einerseits die Anwendungsebene von den Aufgaben der Repräsentation trennt. Andererseits können Teile wie z.B. die Modulverwaltung, die vorher unterhalb der TL Schnittstelle angesiedelt und teilweise typunsicher innerhalb des TL Compilers implementiert waren, nach oben verlagert werden (s. Abb. 2.3), und neue Komponenten wie z.B. eine reflektive Schnittstelle zum Übersetzer [Geisler 95] oder eine Schnittstelle zur generischen Datenvisualisierung eingeführt werden.

2.2 Das semantische Modell von TL

Programmierumgebungen für datenintensive Anwendungen, die durch Programmiersprachen der dritten Generation (COBOL, FORTRAN, C, Pascal) implementiert werden, haben das Problem, daß der Datenbankprogrammierer für die Objekte von verschiedenen Servern (z.B. Datenbank–, Programm– und Bildschirmobjekte) verschiedene Benennungs–, Lebensdauer– und Bindungskonzepte erlernen und Bindungen zwischen Objekten auf verschiedenen Medien durch spezielle Namenskonventionen realisieren muß. Deshalb weicht das Tycoon Modell von diesen Umgebungen ab, indem es den gesamten Prozeß der Integration, Erweiterung, Spezialisierung, Nutzung oder auch Neudefinition von generischen Diensten in einem unifizierenden sprachlichen Rahmen abwickelt [Matthes 93].

2.2.1 Bindungen und Signaturen

Das semantische Modell von Tycoon, deren zentrale Entitäten Werte, Typen, Bindungen und Signaturen sind, basiert auf Typtheorien höherer Ordnung [Burstall, Lampson 84; Cardelli 89]. Aufgrund der Bedeutung von Namens–, Sichtbarkeits– und Bindungskonzepten in der Programmierung umfangreicher Applikationen, bilden nicht isoliert Typen und Werte, sondern Signaturen und Bindungen die wesentlichen semantischen Konstruktionsblöcke [Matthes, Schmidt 92].

Bindungen sind geordnete Assoziationen zwischen benutzerdefinierten Namen und semantischen Objekten, die die (wiederholte) Verwendung dieses Namens zur Bezeichnung des gebundenen Objektes in Ausdrücken erlaubt. In TL können Namen an konstante Werte, variable Werte und Typen, die (partielle) Spezifikationen von Werten darstellen, gebunden werden, so daß die Referenzierung eines Wertes oder eines Typs durch verschiedene Namen und die Einführung rekursiver Strukturen auf der Wert– und Typebene möglich sind. Dabei identifizieren Namen in variablen Wertbindungen anonyme Zellen, die wiederum Werte enthalten. Bindungen sind unveränderlich, nur der Inhalt der Zelle einer variablen Wertbindung kann aktualisiert werden [Matthes, Schmidt 93b].

Verschiedene statische und dynamische Bindungsmechanismen erlauben eine Modellierung verschiedenartiger Sichtbarkeitsregeln und Benennungsschemata unabhängig von der Struktur der benannten Objekte. Dadurch, daß Typausdrücke und Typvariablen als gleichberechtigte sprachliche Objekte behandelt werden, erzielt TL eine wesentlich größere Mächtigkeit und Flexibilität als andere Sprachen.

Bindungen sind in die Syntax von Werten eingebettet, so daß diese benannt, als Parameter übergeben usw. werden können. Signaturen dagegen, die als (partielle) Spezifikationen einer Folge von statischen und dynamischen Bindungen agieren, erscheinen in der Syntax von Typen. Es werden konstante Wertsignaturen ($x :T$), variable Wertsignaturen (**var** $x :T$) und Typsignaturen (T'<:T) unterschieden, die den Namen eines konstanten bzw. variablen Wertes mit einem Typ bzw. einen Typ mit einem Supertyp assoziieren. Signaturen erlauben die Überprüfung der Korrektheit von Wert– und Typausdrücken in Abhängigkeit von Namen, ohne Zugriff auf die durch den Namen definierte Bindung zu haben. Signaturen spielen damit eine zentrale Rolle in typsicheren Anwendungsumgebungen wie Tycoon, die eine separate Übersetzung unterstützen. Die Gültigkeit von Signaturen ist auf die Übersetzungszeit, die von Bindungen dagegen auf die Laufzeit beschränkt.

2.2.2 Benennung und Typisierung vordefinierter Werte und Funktionen

Im Gegensatz zu Programmiersprachen, die dem Programmierer eine Vielzahl vordefinierter Datentypen wie ganze Zahlen oder Zeichenketten bereitstellen, welche damit eine Sonderbehandlung gegenüber den benutzerdefinierten Typen erfahren, versucht TL eine Homogenität zwischen diesen Typen zu erreichen. Dies führt zu einer Begrenzung der Sprachkomplexität und einer Erhöhung des Verständnisses von TL Programmen durch die Vermeidung impliziter Konvertierungsoperationen und unübersichtlicher Bindungskonventionen [Matthes 93].

Deshalb exisistieren in TL in der Standardbibliothek Module, über die die Namen, Konstanten und die Funktionen für die Basistypen explizit bereitgestellt werden. Sie gehorchen dabei den gleichen Syntax–, Typ–, und Evaluationsregeln wie benutzerdefinierte Typen, Werte und Funktionen. Durch einen generischen Mechanismus werden jedoch zusätzlich viele der Funktionen auf den Basistypen an symbolische Bezeichner gebunden, um notationelle Nachteile zu

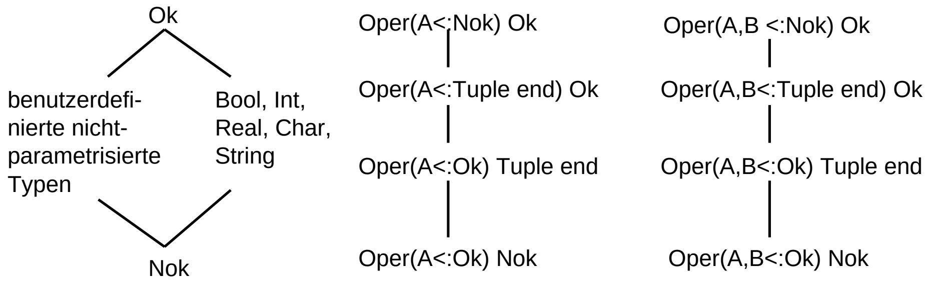


Abbildung 2.4: Verschiedene Typhierarchien in TL

vermeiden:

let + = *int.add*
let <> = *string.concat*
let == = *poly.equal*
...

Im *Frontend* des TL Compilers sind diese Basistypen “fest verdrahtet”. Ebenso ist die interne Wertrepräsentation der Basistypen im *Backend* des TL Compilers statisch verankert und durch Vorgaben des Objektspeicherprotokolls bestimmt.

2.2.3 Typhierarchien

Typaussagen der Form *x :T* stellen partielle Spezifikationen dar, da der durch *x* referenzierte Wert mindestens die durch den Typ *T* gegebene Spezifikation, eventuell aber auch eine präzisere, erfüllt. Die partielle Ordnung auf Typen und Typoperatoren wird dabei durch die reflektive und transitive Subtypbeziehung <: beschrieben, wobei das Subsumptionsprinzip $(a : A \wedge A <: B \Rightarrow a : B)$ gilt. Von Bedeutung für die Definition der Regeln für Subtypbeziehungen in TL ist der Begriff der *Subsignatur*. Signaturen *S* heißen demnach *Subsignaturen* von Signaturen *S’*, wenn die geordneten Folgen *S* und *S’* die gleiche Länge besitzen, und die Typen A_i der Signaturkomponenten in *S* Subtypen der Typen A'_i an gleicher Position in *S’* sind. Sind darüber hinaus die Variablennamen x_i und x'_i beide nicht anonym, so muß $x_i = x'_i$ gelten.

Da es sich bei TL um eine Programmiersprache höherer Ordnung handelt, gibt es eine Vielzahl von Typhierarchien, die durch die Subtypisierungsregeln für die verschiedenen Typkonstruktoren bestimmt sind (s. Abb. 2.4[5]). Von allen nichtparametrisierten Typen ist der Typ **Ok** der Supertyp und der Typ **Nok**, der der Ergebnistyp aller Ausdrücke ist, die ein Ausnahmepaket auslösen, der Subtyp. Während zwischen den Basistypen keine Subtypbeziehungen existieren, ist dies zwischen benutzerdefinierten Typen möglich. Weitere Typhierarchien entstehen abhängig von der Anzahl ihrer Parameter durch Typoperatoren erster Stufe, bei denen die Subtypisierung gemäß der Kontravarianzregel erfolgt. Typoperatoren höherer Ordnung bilden wiederum eigene Typhierarchien. In TL ist es jedoch nicht möglich, einen Zusammenhang zwischen den verschiedenen Typhierarchien mit Hilfe einer *Kind* Spezifikation [Cardelli 89] herzustellen.

---

[5] entnommen aus [Geisler 95]

2.2.4 Struktur– und Namensäquivalenz

In der Regel gilt für die Subtypisierung in *TL* Strukturäquivalenz. So ist beispielsweise die Subtypisierungsregel für Tupeltypen ohne Varianten folgendermaßen definiert:

> Seien A und B Tupeltypen ohne Varianten, dann gilt $A <: B$, wenn A ein Präfix von Signaturen besitzt, die Subsignaturen von B sind.

Für die anderen TL Typkonstruktoren sei auf den Sprachreport [Matthes, Schmidt 92] verwiesen. Eine Ausnahme bilden dagegen die abstrakten Datentypen, deren Kompatibilitätsregel auf der Namensäquivalenz beruht.

Um zufällige, d.h. nicht gewünschte Kompatibilitäten zwischen strukturell äquivalenten Typen zu vermeiden, lassen sich in TL *einschränkende* Wert– und Typbindungen mit der Namensäquivalenzregel für abstrakte Typvariablen zum Konzept der *semiabstrakten* Typvariablen kombinieren. Demnach erfüllt eine abstrakte Typvariable $x.T <: A$ sowohl die Spezifikation $x.T$ (Namenskompatibilität) als auch die Spezifikation A (Strukturkompatibilität). Eine Signatur $a :x.T$ wird nur durch Werte erfüllt, die durch eine Injektionsfunktion *x.new* erzeugt wurden. Die Vorteile dieser Kombination aus Struktur– und Namenskompatibilität sollen anhand des folgenden Beispiels erläutert werden:

```
Let Car = Tuple name :String  age :Int end
let car :Tuple T <:Car  new(:Car) :T end =
  tuple Let T = Car  let new(x :Car) = x end

Let Person = Tuple name :String  age :Int end
let person :Tuple T <:Person  new(:Person) :T end =
  tuple Let T = Person  let new(x :Person) = x end

let audi = car.new(tuple "Audi" 3 end)
let mueller = person.new(tuple "Mueller" 25 end)
let meier = person.new(tuple "Meier" 43 end)

let getAge(item :Tuple name :String  age :Int end) = item.age
let married(p1, p2 :person.T) :Bool = string.equal(p1.name p2.name)

getAge(meier) ⇒ 43
married(meier mueller) ⇒ false
married(meier audi)
⇒ Argument type mismatch: 'person.T' expected, 'car.T' found
   [while checking function argument 'p2']
```

Die Typvariablen *person.T* und *car.T* heißen semiabstrakt, da sie nicht nur die triviale Spezifikation (<:**Ok**), sondern auch noch eine speziellere (*Person* bzw. *Car*) erfüllen. Der erste Aufruf der Funktion *married* ist erfolgreich, da die Werte *meier* und *mueller* mit der gleichen Injektionsfunktion *person.new* erzeugt worden sind und somit aufgrund der Namensäquivalenzregel für abstrakte Typvariablen die Bedingung <:*person.T* erfüllen. Bei dem zweiten Aufruf erfüllt der Wert *audi* diese Bedingung jedoch nicht, obgleich die Typstruktur äquivalent zur Struktur von Werten des Typs *Person* ist.

Auf semiabstrakten Werten sind jedoch alle diejenigen Operationen zulässig, die auf Werten des als obere Schranke angegebenen Typs definiert sind. Da *Person* strukturell äquivalent zu dem Typ des Parameters *item* der Funktion *getAge* ist, ist der Aufruf dieser Funktion mit dem Wert *meier* zulässig. Ebenso wäre der Aufruf der Funktion *married* für den Wert *audi* zulässig, wenn bei den Bezeichnern *person* und *car* nicht explizit der einschränkende Typ angegeben worden wäre, da in diesem Fall beide den vom Compiler inferierten Typ hätten und die beiden Typvariablen $person.T$ und $car.T$ deshalb strukturell äquivalent wären.

3. Dienste zur Datenvisualisierung

Um einen Dienst zur Daten– und Funktionsvisualisierung zu entwickeln, sind zunächst die Anforderungen zu definieren, die dieser erfüllen soll. Dazu werden verschiedene Visualisierungsaspekte untersucht und anhand der typvollständigen Systeme O_2 [Deux 91; Bancilhon et al. 92] und **Napier88** [Morrison et al. 94; Kirby et al. 94; Brown et al. 90] erläutert. Dabei wird auch die Entwicklung in Programmierumgebungen für datenintensive Anwendungen und die sich daraus ergebenden Anforderungen berücksichtigt.

Um unterschiedliche Visualsierungsdienste vergleichen zu können, haben viele Autoren versucht, Taxonomien für die Softwarevisualisierung aufzustellen (z.B. [Myers 90; Stasko, Patterson 92]). Jedoch basieren alle Ansätze auf unterschiedlichen Ideen, ohne alle Aspekte der Visualisierung zu erfassen. Außerdem werden nirgendwo die Ziele der Visualisierung und die Gruppe der Benutzer des interaktiven Systems berücksichtigt [Lavery 95]. Einzig in [Price et al. 93] wird der Versuch gemacht, den Benutzer zu integrieren.

Deshalb gehen zwar einige der in den verschiedenen Taxonomien aufgestellten Kriterien in den Vergleich der untersuchten Systeme ein. Die wesentlichen Fragestellungen orientieren sich jedoch an den Zielen der Visualisierung, der Art der Darstellung auf dem Bildschirm und der Implementierung der Visualisierung, die in den ersten drei Abschnitten diskutiert werden. Ausgehend von den Ergebnissen, die dieser Vergleich erbringt, wird im letzten Abschnitt dieses Kapitels ein Ansatz für die generische Daten– und Funktionsvisualisierung in Tycoon vorgestellt.

3.1 Ziele der Visualisierung

Grundlage für den Entwurf einer Visualisierungskomponente bilden die Ziele, die damit verfolgt werden sollen. Dabei ist zu untersuchen, was zu visualisieren ist und welche Funktionalität auf den visualisierten Objekten bereitgestellt werden soll.

3.1.1 Zu visualisierende Informationen

In herkömmlichen Programmiersprachen hängt die Fragestellung, welche Informationen zu visualisieren sind, vom Paradigma der zugrundeliegenden Programmiersprache ab [Stasko, Patterson 92]. Grundsätzlich können verschiedene Aspekte eines Programmes visualisiert werden, also z.B. der Programmtext, die Datenstrukturen, der Status eines sich in der Ausführung befindlichen Programmes, der Kontrollfluß oder die implementierten Algorithmen, wobei jedoch gilt: je abstrakter der Inhalt der zu visualisierenden Daten ist, desto weniger kann die Generierung der Bildschirmrepräsentation automatisiert werden [Brown 88]. So ist es z.B. möglich, den Programmtext eines Sortieralgorithmus' automatisch zu visualisieren. Um jedoch dessen Arbeitsweise dem Anwender verständlich machen zu können, muß die Visualisierung explizit ausprogrammiert werden, da aus den Strukturen eines Algorithmus' nicht die den Entwurf dokumentierenden semantischen Informationen gewonnen werden können.

In Programmierumgebungen für datenintensive Anwendungen liegt der Fokus auf der Visualisierung persistenter Daten. d.h. auf der Topologie des persistenten Speichers und der Struktur der einzelnen Objekte mit den aktuellen Werten ihrer Komponenten. Durch die konzeptuelle Integration von Datenmodellen und algorithmisch vollständigen Programmiersprachen innerhalb von Datenbankprogrammiersprachen einerseits, und die Entwicklung neuartiger Architekturen für die existierenden Systeme zur Datenbankprogrammierung andererseits, hängen die zu visualisierenden Daten in diesen Systemen jedoch nicht mehr allein von dem Paradigma der zugrundeliegenden Programmiersprache, sondern auch von dem verwendeten Datenmodell und von der Systemarchitektur ab. Anhand zweier grundsätzlich unterschiedlicher Familien von Datenbankprogrammiersprachen — datenmodellabhängige und –unabhängige — werden die zu visualisierenden Daten beschrieben.

Datenmodellabhängige Programmiersprachen

Datenmodellabhängige Programmiersprachen zeichnen sich durch eine strikte Trennung zwischen den für den Anwendungsprogrammierer sichtbaren Abstraktionen und den sie implementierenden Datenstrukturen oder Algorithmen aus, was sich durch die Trennung zwischen Sprachen zur strikt typisierten Applikationsprogrammierung und zur weitgehend typunsicheren Systemimplementierung manifestiert [Matthes 93]. Zwei Ausprägungen dieser Sprachen sind die relationalen (z.B. DBPL [Matthes, Schmidt 93a]) und objektorientierten (z.B. O_2) Datenbankprogrammiersprachen.

Relationale Datenbankprogrammiersprachen wie DBPL bauen auf einem pascalähnlichen Sprachkern auf und sind um Relationentypen, die Elemente gleichen Typs zusammenfassen, sowie mengenorientierte Operationen auf typisierten Relationenvariablen erweitert. Es können Variablen mit relationalem Typ oder auch Werte mit nichtprozeduralem Typ gespeichert werden.

Objektorientierte Systeme dagegen verfügen über typische objektorientierte Merkmale wie Objektidentität, Klassifikation, Mehrfachvererbung. Ebenso bieten sie Mechanismen zur Definition sogenannter komplexer Objekte und Methoden sowie zur späten Bindung und Methodenredefinition in Subklassen an. In O_2 z.B. wird zwischen Werten und Objekten als Instanzen einer Klasse unterschieden, die automatisch persistent sind, solange sie von einem benannten Wurzelobjekt aus einem Schema direkt oder indirekt erreichbar sind. Schemata dienen in datenmodellabhängigen Programmiersprachen der abstrakten Beschreibung der an einer Anwendung beteiligten Relationen bzw. Klassen sowie deren Strukturen und Beziehungen.

In diesen Systemen existieren Komponenten zur Visualisierung von Schemata, den Elementen von Relationen bzw. Objekten einer Klasse sowie von Anfrageergebnissen. O_2 bietet darüberhinaus noch Komponenten zur Visualisierung von Anwendungen, Funktionen, persistenten Typen und persistenten Namen.

Modellunabhängige Datenbankprogrammiersprachen

Persistente Programmiersprachen wie Napier88 und P-Quest [Müller 91; Matthes 91] unterscheiden sich von den oben beschriebenen Datenbankprogrammiersprachen, indem sie einen datenmodellunabhängigen, abstrakten, ausdrucksmächtigen und in sich geschlossenen architektonischen Rahmen für die Entwicklung großer Softwaresysteme zur Manipulation langlebiger Daten anbieten. Dies führt zu einer Unabhängigkeit von einem bestimmten Datenmodell und zu orthogonaler Persistenz. Dabei wird orthogonale Persistenz durch drei Prinzipien erreicht [Atkinson, Morrison 95]:

Unabhängigkeit der Persistenz: Eine abstrakte Objektspeicherschnittstelle wickelt während der Programmausführung alle Speicherzugriffe ab. Sie gestattet die Gleichbehandlung von persistenten und nichtpersistenten sowie prozeßlokalen und systemweiten Datenstrukturen innerhalb von Applikations– und auch Systemprogrammen.

Orthogonalität der Datentypen: Werte beliebiger Datentypen, und damit auch Module und Anwendungsprogramme, können gespeichert werden.

Identifikation von Persistenz: Alle Objekte, die transitiv von einem ausgezeichneten Wurzelobjekt durch statische Sichtbarkeitsregeln oder dynamische Bindungen erreichbar sind, werden langlebig gespeichert und stehen anderen Programmen zur Verfügung.

Die Unabhängigkeit von einem speziellen Datenmodell führt dazu, daß in diesen Systemen nicht die Visualisierung von Schemata und Massendaten, sondern die Visualisierung beliebiger Datenstrukturen des Objektspeichers im Vordergrund steht. Da durch die orthogonale Persistenz Anwendungsprogramme und Module selbst wieder als Daten betrachtet werden können, umfaßt dies auch persistente Programme, wobei die gleichen Aspekte visualisiert werden können wie bei Programmen in nichtpersistenten Programmiersprachen. Damit dient die Datenvisualisierung in persistenten Programmiersprachen nicht nur dem Anwender, sondern sie kann auch den Anwendungsprogrammierer bei der Entwicklung seiner Programme unterstützen.

Während in modellabhängigen Datenbankprogrammiersprachen die Beschreibung der in einer Datenbank gespeicherten Daten durch ein Schema erfolgt, geschieht dies in modellunabhängigen Datenbankprogrammiersprachen durch Typausdrücke. Um das Verständnis der visualisierten Daten zu erhöhen, sollten dementsprechend auch Typen visualisiert werden.

Ein langfristiges Ziel in modellunabhängigen Programmiersprachen ist es, unterschiedliche Datenmodelle uniform durch ein ausreichend ausdrucksfähiges Typsystem auszudrücken [Atkinson, Morrison 86]. Zwei Ansätze, dies zu erreichen, sind erweiterbare Grammatiken [Matthes et al. 95b] und, um die Anwendungssemantik der Datenmodelle zu erfassen, die Erweiterung der persistenten Programmierumgebung unter Ausnutzung des existierenden Typsystems um domänenspezifische konzeptuelle Abstraktionen, durch das System erzwungene Integritätsbedingungen, die über das Typsystem hinausgehen, und erweiterte generische Systemfunktionalität [Wetzel et al. 95]. Anders als in modellabhängigen Datenbankprogrammiersprachen müssen dabei zur Visualisierung der Daten jeweils zusätzliche Dienste bereitgestellt werden, um die Daten entsprechend ihrer Semantik darzustellen.

3.1.2 Funktionalität

Eine zweite Dimension zur Bestimmung der Ziele der Datenvisualisierung ist die auf den dargestellten Daten angebotene Funktionalität. Grundsätzlich hat ein Visualisierungsdienst in persistenten Programmierumgebungen die Aufgabe, die Topologie und den Inhalt des persistenten Speichers darzustellen. Dazu muß der Anwender, neben der Möglichkeit, beliebige langlebige Daten visualisieren zu können, auch in der Lage sein, entlang der visualisierten Daten durch den persistenten Speicher zu navigieren. Weitere Funktionalität ist abhängig von dem Zweck, zu dem die visualisierten Daten verwendet werden sollen, und wird anhand von O_2 und Napier88 diskutiert.

O_2

O_2 verfügt mit O_2Tools über eine komplette grafische Programmierumgebung für den Entwurf und die Entwicklung objektorientierter Datenbankapplikationen. O_2Tools zielt auf zwei unterschiedliche Benutzergruppen ab:

- ▹ Der Softwareentwickler benutzt O_2Tools einerseits, um Daten und Schemata zu visualisieren und zu editieren sowie Anfragen darauf zu formulieren, und andererseits zum Editieren und Testen von Methoden und Programmen sowie zur Fehlersuche.
- ▹ Der Anwender kann O_2Tools benutzen, um Daten der Datenbank und die Schemata zu visualisieren, zu editieren und Anfragen darauf zu formulieren.

Eine Komponente von O_2Tools ist der *Browser*, der die Visualisierung, Modifizierung, d.h. Erzeugen, Verändern und Löschen, von Schemata und Daten der Datenbank sowie Anfragen darauf ermöglicht [O_2Technology 92]. Somit wird die Funktionalität herkömmlicher Visualisierungskomponenten erweitert um Manipulations– und Anfragemöglichkeiten. Um die Manipulation visualisierter Werte realisieren zu können, müssen bei Änderungen in der Datenbank oder im Schema auch die entsprechenden Bildschirmrepräsentation aktualisiert werden und umgekehrt. Außer der direkten Manipulation von Eingabefeldern per Tastatur wird die Manipulation per Datenaustausch zwischen unterschiedlichen Bildschirmrepräsentationen unterstützt. Dies erfordert dynamisch, d.h. zur Laufzeit, eine Typüberprüfung, um sicherzustellen, daß der in eine Bildschirmrepräsentation kopierte Wert den gleichen Typ hat wie der ursprünglich dargestellte Wert. Da der *Browser* in die Entwicklungumgebung integriert ist, können daraus auch andere Dienste wie z.B. ein Quelltexteditor für Methoden gestartet werden.

Napier88

In Napier88 ist der Objektspeicher durch sogenannte Environments strukturiert, die Werte erster Klasse darstellen [Dearle et al. 92]. Jedes Environment enthält eine Liste von Bindungen, die selbst auch wieder Environments referenzieren können. Um auf einen Wert des Objektspeichers zugreifen zu können, muß der Anwendungsprogrammierer den Pfad zu dem Environment spezifizieren, in dem dieser gebunden ist.

Die erste Version des *Browsers* erlaubte nur die Visualisierung von Werten und das Navigieren im Objektspeicher. Der Anwender sollte bei der Programmierung unterstützt werden, indem er ihm beim Auffinden von Werten hilft, die Ergebnisse von Berechnungen in grafischer Form anzeigt und durch Integration in einen *Debugger* die Fehlersuche erleichtert [Dearle, Brown 88].

Die Erweiterung des *Browsers* um einen Mechanismus zur Markierung visualisierter Werte erlaubt eine Unterstützung der Programmkonstruktion durch die Bindung von Werten zu unterschiedlichen Zeitpunkten [Farkas et al. 92]:

Bindung zur Laufzeit: Durch die Markierung visualisierter Werte, die zur Laufzeit gebunden werden sollen, kann automatisch während der Erstellung des Programmtextes eine

Repräsentation des sie lokalisierenden Pfades generiert und in den Quellkode integriert werden.

Bindung zur Übersetzungszeit: Werte und Typen, die zur Übersetzungszeit in einer Übersetzungsumgebung gebunden werden sollen, können mit einem Namen markiert werden, der im Programmtext als Zugriffsspezifikation verwendet werden darf. Als Eingabe erhält der *Compiler* den Quelltext zusammen mit der Abbildungsvorschrift der Markierungen auf Werte und Typen. Indem der gleiche Programmtext mit unterschiedlichen Abbildungsvorschriften verwendet wird, können unterschiedliche Anwendungen erzeugt werden, ohne die ansonsten aufwendige textuelle Spezifikation des Zugriffs auf Werte und Typen ändern zu müssen.

Bindung zur Programmkonstruktionszeit: Durch das direkte Einfügen markierter Werte in den Programmtext, genannt *Hyperprogrammierung* [Kirby et al. 92], erhält man nichtlineare Programmrepräsentationen. Hierbei erfolgt die Abbildung der Markierungen auf die Werte bereits zur Konstruktionszeit des Programmtextes. Diese Art der Programmerzeugung erfordert allerdings zusätzliche grafische Benutzerinteraktion in einem speziellen Programmeditor, der die Repräsentation und Visualisierung markierter Werte innerhalb des Programmtextes ermöglicht.

3.2 Visualisierungsaspekte

In [Larkin, Simon 87] wird behauptet, daß Grafik einen Vorteil gegenüber Text besitzt, da die Repräsentation einfach wahrnehmbar ist und nicht anspruchsvollere logische Schlußfolgerungen für das Verständnis notwendig sind, und da die räumliche Anordnung zur Verringerung der Suchzeit beiträgt. Dies gilt nach [Leung, Apperly 93] aber nur, wenn die Visualisierung sich an der Aufgabe orientiert und den Anforderungen entsprechende Repräsentationen erzeugt werden. In diesem Abschnitt werden verschiedene Visualisierungsaspekte untersucht, die das Verständnis der dargestellten Daten beeinflußen.

3.2.1 Grafische Benutzerschnittstelle

Die Erstellung grafischer Repräsentationen kann sehr zeitintensiv und mit viel Programmtext verbunden sein. Um den Entwickler bei der Konstruktion von Visualisierungsschnittstellen zu unterstützen, gibt es Dienste, die eine Reihe grafischer Grundfunktionalitäten implementieren und bereitstellen. Er muß dann nur noch die für seine Anwendung notwendigen Elemente miteinander verbinden.

Eine ausführliche Vorstellung existierender Dienste sowie deren Vor– und Nachteile findet sich in [Müßig 94]. Dabei wird die Kombination aus systemübergreifender Klassenbibliothek, die die Portabilität von Anwendungen bezüglich unterschiedlicher Fenstersysteme sicherstellt, und Quelltextgenerator für die Integration als generischer Dienst in eine allgemeine Programmiersprache empfohlen. In diesem Abschnitt sollen zusätzlich Aspekte bezüglich der Frage, ob eine existierende externe oder eine in der Programmierumgebung entwickelte Klassenbibliothek verwendet werden soll, und bezüglich der Portabilität diskutiert werden.

Externer versus interner Dienst

Ein in der Programmierumgebung selbst entwickelter Dienst hat den Vorteil, daß der Dienst entsprechend den eigenen Wünschen und Anforderungen entwickelt werden kann. Bei der Verwendung externer Dienste dagegen ist eine Kompromißlösung notwendig, da diese die Anforderungen nicht in idealer Weise erfüllen. Um eine bedingte Anpassung an die eigenen Anforderungen zu ermöglichen, sollte ein externer Dienst aber erweiterbar sein. Diese Forderung wird durch objektorientierte Dienste erfüllt. Der Vorteil externer Dienste ist, daß die Entwicklung einer eigenen Benutzerschnittstelle sehr aufwendig und fehlerträchtig sein kann und einen hohen Wartungsaufwand erfordert.

Sowohl O_2 mit dem Benutzerschnittstellengenerator O_2Look als auch Napier88, verwenden einen internen Dienst zur Entwicklung grafischer Objekte. O_2Look ist voll in die persistente Umgebung integriert, ein Kennzeichen objektorientierter Systeme, die sich durch eine enge konzeptuelle und technologische Verflechtung zwischen grafischen Benutzerschnittstellen, modularer und erweiterbarer Datenabstraktion sowie effizienten Objektspeichern auszeichnen [Matthes 93]. Die wesentlichen Fähigkeiten von O_2Look sind [Deux et. al. 90]:

- Die grafische und interaktive Manipulation beliebig komplexer O_2 Objekte und Werte wird unterstützt.
- Der Programmierer kann grafische Benutzeroberflächen erzeugen.
- Um die Anforderungen spezifischer Anwendungen zu erfüllen, können die grafischen Repräsentationen sowohl statisch definiert als auch dynamisch verändert werden.

Da in O_2Look jedoch die Bildschirmrepräsentationen nur eingeschränkt modifizierbar sind, existiert ein weiterer Dienst ToonMaker, der die Abbildung zwischen Datenbankobjekten und grafischen Strukturen zusätzlich unterstützt, indem er

- die Modifizierung vorgefertigter Präsentationen eines Objektes einer bestimmten Klasse erlaubt und
- die Erzeugung einer Präsentationsschablone unterstützt, die der Programmierer für verschiedene Instanzen einer Klasse wiederverwenden kann [Borras et al. 92].

Diese Eigenschaften erlauben eine interaktive Anpassung der Datenvisualisierung an die Semantik durch den Entwickler. In modellabhängigen Datenbankprogramierumgebungen, in denen sich die Struktur der abgespeicherten Daten kaum verändert, kann dadurch die ansonsten automatische Generierung unterstützt werden. In modellunabhängigen Datenbankprogrammiersprachen dagegen erscheinen diese Eigenschaften nur für sich kaum ändernde Programmstrukturen sinnvoll, wie z.B. für Werte, die einen aus den Modulen der Tycoon Standardbibliothek exportierten Typ besitzen.

In Napier88 ist ein eigenes Fenstersystem WIN mit Funktionen zur Bearbeitung von Bildschirmobjekten entwickelt worden, da es gegenüber externen Diensten den Vorteil hat, daß die Bildschirmobjekte wie alle Daten automatisch persistent sind. Bei der Verwendung externer Dienste ist ein zusätzlicher Mechanismus notwendig, um die auf einem externen *Server* erzeugten und verwalteten Bildschirmobjekte zu speichern. Allerdings weist das System noch einige konzeptuelle Mängel auf, da z.B. zu wenig Interaktionselemente angeboten werden und auch das Modell zur Handhabung von Ereignissen einer Überarbeitung bedarf [Lavery 95].

Portierbarkeit

In verteilten, heterogenen Anwendungen spielt die Portierbarkeit der Anwendungen eine wichtige Rolle. Klassenbibliotheken setzen auf den Fenstersystemen auf, die auf der jeweiligen Hardware installiert sind. Um systemübergreifend zu sein, müssen sie eine identische Programmierschnittstelle und Sprache für die Beschreibung von Benutzerschnittstellen verschiedener Fenstersysteme anbieten, die von der teilweise unterschiedlichen Funktionalität und dem *look&feel* der jeweiligen Fenstersysteme abstrahieren. Dies stellt eine hohe Anforderung an systemübergreifende Klassenbibliotheken dar, wobei es zwei unterschiedliche Ansätze zur Vereinheitlichung unterschiedlicher Funktionalitäten gibt: beim *Schnittmengenansatz* wird nur die allen Fenstersystemen gemeinsame Funktionalität unterstützt, während beim *Obermengenansatz* versucht wird, für jedes Fenstersystem die nicht unterstützte Funktionalität anderer Fenstersysteme nachzubilden [Müßig 94].

3.2.2 Visualisierung von Objekten

Die Art der Visualisierung wird davon beeinflußt, was zu welchem Zweck dargestellt werden soll. Für die Visualisierung von Hierarchien eignet sich z.B. die Darstellung in Form von Graphen. In diesem Abschnitt werden Aspekte bezüglich der Visualisierung von Objekten diskutiert, d.h. Datenbankobjekten in modellabhängigen Programmierumgebungen und Programmobjekten in persistenten Programmiersprachen.

Ein Visualisierungsaspekt betrifft die Bindung eines Objektes an verschiedene Namen. Um diese Semantik dem Anwender auch bei der Visualisierung von Objekten zu vermitteln, sollte ein Objekt nur einmal auf dem Bildschirm dargestellt werden. Falls zu einem Objekt mehrere Bildschirmrepräsentationen existierten, wäre ein erhöhter Verwaltungsaufwand notwendig, um die Konsistenz der visualisierten Daten zu gewährleisten, da bei einer Manipulation eines visualisierten Objektes auch alle anderen Bildschirmrepräsentationen dieses Objektes entsprechend aktualisiert werden müßten.

Da die zu visualisierenden Objekte in der Regel strukturiert sind, hat sich die Strukturdarstellung als eine geeignete Repräsentation von Objekten herausgestellt. Diese Technik wird auch in O_2 und Napier88 angewendet. Unterschiede existieren in der Darstellung der Komponenten. In Napier88 werden grundsätzlich der Name und der Typ der Komponente dargestellt. Um sich den Wert einer Komponente anzusehen, muß der Anwender mit der Maus die Komponente auswählen. Werte von Basistypen werden dann in einem allen Basiswerten gemeinsamen Fenster in textueller Form, strukturierte Werte dagegen in einem eigenen Fenster visualisiert. Diese Darstellung hat den Nachteil, daß sie für den Anwender sehr unübersichtlich werden kann, wenn ein Wert aus vielen Komponenten besteht und der Wert jeder Komponente in einem eigenen Fenster dargestellt wird. Dieses versucht man in O_2 zu verhindern, indem Komponenten in dem Objekt dargestellt werden, zu dem sie gehören. Bei Werten von Basistypen erfolgt dies direkt, bei strukturierten Werten werden zunächst auch nur Name und Typ der Komponente dargestellt. Wird eine strukturierte Komponente mit der Maus angeklickt, so wird der Typ durch eine Repräsentation des strukturierten Wertes ersetzt. Ziel dieser Darstellung ist es, die *Ist–Teil–Von*–Relation sichtbar zu machen [Borras et al. 92]. Für konstante Werte von Basistypen erscheint diese Regelung aus Gründen der Übersicht sinnvoll. Variable Werte von Basistypen sowie strukturierte Komponenten, die wiederum variable Komponenten enthalten können, sollten dagegen weiterhin in eigenen Fenstern visualisiert werden, da ansonsten von einem Wert verschiedene Bildschirmrepräsentationen existieren würden.

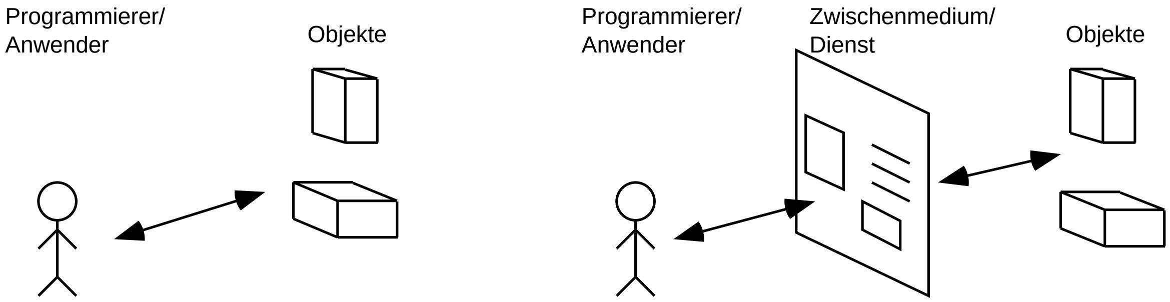


Abbildung 3.1: Objektfokussierte vs. sichtfokussierte Darstellung

Um die *Ist–Teil–Von*–Relation trotzdem sichtbar zu machen, werden z.B. in **Napier88** Referenzen durch Pfeile dargestellt. Bei der Visualisierung großer und komplexer Datenstrukturen erweist sich jedoch die begrenzte Größe des Bildschirms als Problem, da es bei vielen Referenzen für den Anwender unmöglich wird, die Relationen zu identifizieren. Zur Lösung dieses Problems wurde zusätzlich das Konzept des Universums eingeführt, in dem der Anwender mehrere Objekte zu einer Gruppe zusammengefassen kann. Da ein Universum ein in sich abgeschlossenes Bildschirmobjekt darstellt, zeigen Referenzen auf die in einem Universum zusammengefaßten Objekte auf das Universum, nicht aber auf das Objekt selbst. Außerdem können dadurch von einem Objekt mehrere Bildschirmrepräsentationen in unterschiedlichen Universen erzeugt werden. Damit wird durch die Darstellung von Referenzen als Pfeile nicht das Verständnis für die dargestellten Objekte erhöht, sondern im Gegenteil sogar erschwert.

3.2.3 Sichtfokussierte versus objektfokussierte Umgebungen

Eine Frage, die in [Chang et al. 95] diskutiert wird, beschäftigt sich mit dem Problem, ob in Programmierumgebungen die sichtfokussierte oder die objektfokussierte Darstellung gewählt werden soll. Diese Frage betrifft zwar primär Probleme in visuellen Programmiersprachen und –umgebungen, trotzdem sind aber auch einige Aspekte relevant für die Datenvisualisierung in **Tycoon**.

Die meisten Programmierumgebungen wählen die sichtfokussierte Darstellung, in der Dienste als Zwischenmedium für die Untersuchung und Manipulation von Objekten dienen (s. Abb. 3.1[1]). Das drückt sich dadurch aus, daß in einem Dienst nur bestimmte Sichtweisen auf ein Objekt bereitgestellt werden und daß der Anwender über den Dienst mit den Objekten kommuniziert, z.B. indem Menüs nicht den Objekten direkt, sondern dem Dienst zugeteilt sind. Dadurch werden Dienste konkret, Objekte dagegen erscheinen als abstrakt, distanziert und zweitrangig.

In der objektfokussierten Darstellung dagegen soll das Zwischenmedium verschwinden und somit das Objekt konkret, unmittelbar und primär werden (s. Abb. 3.1). Anhand der typsicheren, objektorientierten und nichtpersistenten Programmiersprache **Self** [Ungar, Smith 87] werden einige Prinzipien vorgestellt, um eine objektfokussierte Darstellung zu erreichen:

Unmittelbarkeit: Durch eine unmittelbare Abbildung der Sprachobjekte entsprechend dem zugrundeliegenden Sprachmodell auf ihre korrespondierende Bildschirmrepräsentation sowie eine dreidimensionale Darstellung und die Verwendung von Animation [Chang, Ungar 93] sollen die Bildschirmobjekte als die Objekte selbst und nicht nur als Bildschirmrepräsentation erscheinen.

---

[1] entnommen aus [Chang et al. 95]

Vorrang: Die Funktionalität, d.h. die Eigenheiten und das Verhalten, sollen direkt mit dem Bildschirmobjekt verknüpft sein, so daß der Anwender nicht über die von einem Dienst bereitgestellte Funktionalität, sondern über das Bildschirmobjekt selbst darauf zugreifen kann. Der Zugriff kann z.B. über einen Menüeintrag oder einen Knopf erfolgen.

Verfügbarkeit: Jederzeit soll die gesamte und nicht nur eine durch einen Dienst eingeschränkte Funktionalität für ein Bildschirmobjekt verfügbar sein.

Lebendigkeit: Eine lebendige Schnittstelle erlaubt es einem Objekt, sich unter eigener Kontrolle zu bewegen, zu verändern und zu interagieren.

Für jede Art von Objekten, also z.B. Instanzen, Klassen und Metaklassen in objektorientierten Programmiersprachen, müssen geeignete Repräsentationen gefunden werden, um die objektfokussierte Sicht zu verwirklichen.

3.3 Implementierungsaspekte

Da in persistenten Programmierumgebungen die Verwendung von Bindungen an semantische Objekte nicht auf die Programme beschränkt ist, in denen sie definiert werden, können die zu visualisierenden Daten eine beliebige Struktur haben, die zum Zeitpunkt der Entwicklung des generischen Visualisierungsdienstes jedoch nicht für jeden Wert bekannt sein kann. Der Dienst muß also in der Lage sein, Daten unbekannter Struktur zu visualisieren. Objektorientierte Programmiersprachen wie Smalltalk-80 [Goldberg, Robson 83] versuchen dieses Problem zu umgehen, indem für jede Klasse eine *Print* Methode geschrieben werden muß, die Instanzen dieser Klasse visualisiert. Solch eine Methode ist jedoch nicht sicher, da sie durch eine Methode überschrieben werden kann, die eine völlig andere Funktionalität realisiert. Deshalb sollte der generische Visualisierungsdienst Daten beliebiger Struktur visualisieren können, ohne vom Programmierer abzuhängen.

Eine Möglichkeit, zur Laufzeit Strukturinformationen zu erhalten, ist die Verwendung dynamischer Typisierung, die zunächst kurz vorgestellt wird. Anders als in Kapitel 5 steht dabei ein Vergleich strikt typisierter und dynamisch typisierter Programmiersprachen im Vordergrund. Darauf aufbauend werden zwei Ansätze zur Realisierung der generischen Datenvisualisierung erläutert sowie ihre Vor– und Nachteile diskutiert.

3.3.1 Bedeutung einer dynamischen Typisierung

TL ist eine strikt typisierte Programmiersprache, deren Typüberprüfung statisch zur Übersetzungszeit erfolgt. Die strikte Typisierung stellt sicher, daß alle Programmentitäten vor der Programmausführung auf Konsistenz mit dem Typsystem geprüft werden, so daß die Programmausführung nicht aufgrund von Typfehlern ein falsches Ergebnis liefert oder sogar durch einen Laufzeitfehler abbricht. Dieses ist zur Konsistenzerhaltung der Daten in verteilten datenintensiven Anwendungen [Schmidt, Matthes 93; Schmidt et al. 93], für die Skalierbarkeit eines Systems und für die Unterstützung durch externe Dienste unverzichtbar [Matthes, Schmidt 93b]. Da die Typkorrektheit in der Übersetzungsphase überprüft wird, wird die Programmausführung effizienter. Deshalb ist die statische Typüberprüfung der dynamischen vorzuziehen.

Dynamisch typisierte Sprachen wie CLOS [Kiczales et al. 91], die auch, wie am Beispiel von $P_LO_B!$ [Kirschke 94] gezeigt worden ist, um Persistenz erweitert werden können, haben gegenüber statisch typisierten Sprachen den Vorteil, daß der Programmierer jederzeit auf die Typinformationen der Werte zugreifen kann. Dadurch lassen sich Anwendungen wie die generische Datenvisualisierung (zu weiteren Beispielen s. Abschnitt 5.2) realisieren. In statisch typisierten Programmiersprachen ist bei diesen Anwendungen jedoch zur Übersetzungszeit nicht feststellbar, welchen Typ ein Wert zur Laufzeit haben wird. Zu diesem Zweck müssen sie um das Konzept der dynamischen Typisierung erweitert werden, das in den Situationen, in denen eine statische Typüberprüfung nicht möglich ist, eine Typüberprüfung zu wohldefinierten Zeitpunkten während der Programmlaufzeit erlaubt [Matthews 87].

3.3.2 Ausnutzung von Objektformaten auf Speicherebene

Eine Möglichkeit, Daten beliebigen Typs visualisieren zu können, findet auf einem sehr niedrigen Abstraktionsniveau statt, indem Kenntnisse über die interne Repräsentation von Daten innerhalb des Objektspeichers ausgenutzt werden. Da in der Regel die Abbildung der Daten auf die Objektformate injektiv, aber nicht bijektiv ist, lassen sich die Daten allein aufgrund ihrer Objektformate jedoch semantisch nicht korrekt visualisieren. Deshalb muß entsprechend der zur Laufzeit des Visualisierungsdienstes inspizierten dynamischen Typinformationen aus den Objektformaten eine semantisch korrekte Bildschirmrepräsentation der Daten erzeugt werden. Dazu müssen spezielle Funktionen bereitgestellt werden, mit deren Hilfe die Informationen aus den Objektformaten herausgelesen und innerhalb der Programmiersprache verwendet werden können.

Solch ein Ansatz ist innerhalb von Napier88 realisiert worden [Kirby, Dearle 90]. Der Dienst zur generischen Daten– und Funktionsvisualisierung stellt für jeden Wertkonstruktor eine Funktion bereit, die aus Daten, die mit diesem Konstruktor erzeugt worden sind, eine Bildschirmrepräsentation generiert. Jede dieser Funktionen verwendet weitere Funktionen, die den Zugriff auf die Objektstruktur der Daten realisieren. Um die einem Wert entsprechende Visualisierungsfunktion aufrufen zu können, muß mit Hilfe entsprechender Funktionen die mit ihnen assoziierte Typrepräsentation extrahiert und in ihre Komponenten zerlegt werden.

Dieser Ansatz hat den Nachteil, daß er abhängig ist von der jeweiligen Implementierung des zugrundeliegenden Systems. Ändert sich die Repräsentation der Objektformate, so ist auch der generische Dienst zur Daten– und Funktionsvisualisierung entsprechend anzupassen. Zudem muß der Programmierer dieses Dienstes nicht nur die Programmiersprache, sondern auch die Objektformate kennen. Da diese vor dem Anwendungsprogrammierer verborgen werden sollen, kann dieser Dienst nur von einem Systemprogrammierer entwickelt werden.

3.3.3 Implementierung durch linguistische Reflektion

Ein anderer Ansatz, der unabhängig von den Objektformaten und damit der Implementierung des Systems ist, basiert auf den reflektiven Fähigkeiten von persistenten Programmiersprachen. Dazu erfolgt in diesem Abschnitt zunächst eine Charakterisierung von Reflektion, bevor dieser Ansatz vorgestellt wird.

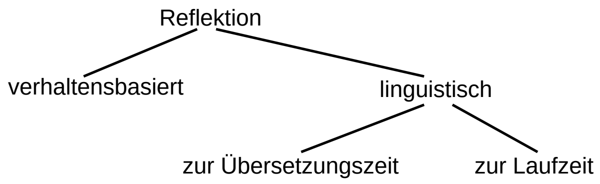


Abbildung 3.2: Arten von Reflektion

Charakterisierung von Reflektion

Programme, die die Fähigkeit besitzen, sich selbst zu inspizieren und zu verändern, werden als reflektiv bezeichnet [Kirby 92]. Dabei wird unterschieden zwischen verhaltensbasierter, d.h. die Veränderung erfolgt durch eine Beeinflussung des Interpretationsvorganges, und linguistischer Reflektion, d.h. ein laufendes Programm generiert neue Programmfragmente, indem Programme als Daten behandelt werden, und integriert diese in seine Ausführung, indem Daten als Programme behandelt werden [Stemple et al. 92b]. Linguistische Reflektion wiederum kann zur Übersetzungszeit oder zur Laufzeit initiiert werden (s. Abb. 3.2).

Bei Übersetzungszeitreflektion definiert der Programmierer Generatorfunktionen, die die Repräsentationen von Programmfragmenten generieren, als Teil des Übersetzungsvorganges ausgeführt werden und das gerade übersetzte Programm ergänzen. Beispielsweise werden in TRPL [Sheard 91] Makros angeboten, die zur Übersetzungszeit evaluiert werden und die es einem übersetzten Programm erlauben, sich an den Kontext anzupassen, in dem die Makros übersetzt werden.

Bei Laufzeitreflektion werden neue Programmteile mit existierenden Programmen einer Umgebung erzeugt und gebunden, indem eine Funktion aufgerufen wird, die eine Zeichenkette als Eingabe erwartet, ausführbaren Programmkode erzeugt und in das ausgeführte Programm einbindet. Solch eine Funktion kann ein dynamisch aufrufbarer *Compiler* sein wie in Napier88 oder wie in O_2 eine Systemfunktion, die einen Interpreter aufruft.

Im Gegensatz zu untypisierten Sprachen wie Lisp [McCarthy et al. 62], in denen linguistische Reflektion eher als gefährlich einzustufen ist, kann sie in typisierten Sprachen typsicher implementiert werden. Das hat zwei Vorteile [Stemple et al. 93]:

- Mehr Informationen in Form von Typen werden bereitgestellt.
- Die Typsicherheit aller neu generierten Programmfragmente wird vor ihrer Ausführung überprüft.

Typsichere linguistische Reflektion stellt einen uniformen Mechanismus zum Entwurf und zur Weiterentwicklung von Programmen bereit, der die Möglichkeiten nichtreflektiver Datenbankprogrammiersprachen übertrifft [Stemple et al. 93]. Sie ist besonders einsetzbar für

- hochgenerische Anwendungen [Stemple et al. 90; Sheard 91],
- die Anpassung an Systemänderungen, die beispielweise bei der generischen Daten– und Funktionsvisualisierung anzutreffen sind [Dearle, Brown 88; Dearle et al. 90],
- die Implementierung von Datenmodellen [Cooper 90; Cooper, Qin 92],

- ▷ die Optimierung von Implementierungen [Cooper et al. 87; Fegaras, Stemple 91] und
- ▷ die Validierung von Spezifikationen [Fegaras et al. 92; Stemple et al. 92a].

Reflektiver Ansatz

Der Einsatz typsicherer linguistischer Reflektion innerhalb des Dienstes zur generischen Daten– und Funktionsvisualisierung dient dazu, diesen zur Laufzeit um Funktionen zu erweitern, die die Visualisierung für Daten eines bestimmten Typs realisieren. Dazu wird anhand der Analyse der Typstruktur der zu visualisierenden Daten Programmkode für diese Funktionen generiert, übersetzt und im persistenten Speicher gebunden.

Dieser Ansatz hat den Vorteil, daß er unabhängig von der Repräsentation von Werten im Objektspeicher ist. Das bedeutet einerseits, daß der Entwickler eines generischen Dienstes zur Daten– und Funktionsvisualisierung die Repräsentation nicht kennen muß. Andererseits muß die Implementierung dieses Dienstes nicht verändert werden, sollte sich diese Repräsentation ändern. Ein Nachteil diese Ansatzes besteht darin, daß bei allen Daten zuerst eine entsprechende Visualisierungsfunktion generiert werden muß, bevor die eigentliche Visualisierung erfolgt. In persistenten Programmierumgebungen läßt sich dieser Nachteil vermindern, indem jede generierte Visualisierungsfunktion zusammen mit dem Typ, für den sie gültig ist, in einer Tabelle gespeichert wird. So muß eine neue Visualisierungsfunktion nur dann generiert werden, wenn für einen Typ noch kein Eintrag in dieser Tabelle existiert.

Solch ein reflektiver Ansatz ist in Napier88 realisiert worden [Kirby, Dearle 90]. Dabei hat sich jedoch herausgestellt, daß die Visualisierung vor allen Dingen bei *Environments* zu langsam ist. Deshalb erfolgt die Visualisierung in Napier88 mit Hilfe des in Abschnitt 3.3.2 vorgestellten Ansatzes, zumal ein weiterer Geschwindigkeitsverlust bei solchen Werten auftritt, für die noch keine Visualisierungsfunktion existiert.

3.4 Ansatz zur Datenvisualisierung im Tycoon System

Ziel der Daten– und Funktionsvisualisierung in Tycoon ist es, sowohl den Programmierer als auch den Anwender beim Inspizieren des Objektspeichers zu unterstützen. Ausgehend von der Diskussion in den vorigen Abschnitten soll der Dienst zur generischen Daten– und Funktionsvisualisierung deshalb folgende Anforderungen erfüllen:

Wert– und Typvisualisierung: Für das Verständnis der Darstellung der Werte ist es für den Anwendungsprogrammierer hilfreich, daß die Visualisierung nicht nur Werte des Objektspeichers, sondern auch deren Typen umfaßt. Da die Visualisierung generisch erfolgen soll, orientiert sich die Repräsentation an den TL Programmkonstruktoren.

Generische Visualisierung mit Hilfe dynamischer Typen: Die Visualisierung soll für beliebige Werte und Typen generisch und unabhängig vom Programmierer erfolgen, so daß die Verwendung dynamischer Typinformationen unverzichtbar ist. Um eine angemessene Geschwindigkeit zu erzielen, erfolgt die Visualisierung nicht reflektiv. Andererseits soll von den Objektformaten abstrahiert werden, so daß die Implementierung unabhängig von einer bestimmten Implementierung der Objektspeicherformate ist.

Unterstützung von Manipulation und Datenaustausch: TL soll zwar nicht zu einer visuellen Programmiersprache erweitert werden, aber es soll die Manipulation von Werten durch direkte Manipulation der Bildschirmrepräsentation sowie durch Datenaustausch zwischen den Bildschirmobjekten unterstützt werden.

Visualisierungsunterstützung: Die Visualisierung erfolgt mit Hilfe einer externen, systemübergreifenden Klassenbibliothek, um den Aufwand der Entwicklung einer eigenen Klassenbibliothek zu vermeiden und um die Portierbarkeit der Daten– und Funktionsvisualisierung zu erhöhen. Dabei soll sichergestellt sein, daß auch die externen Bildschirmobjekte persistent sind.

Erhaltung der Objektidentität: Jedes Objekt soll nur einmal auf dem Bildschirm dargestellt werden, so daß Komponenten strukturierter Werte mit Ausnahme von konstanten Werten der Basistypen in eigenen Fenstern visualisiert werden. Referenzen zwischen den Objekten werden aus Gründen der Übersichtlichkeit nicht dargestellt.

Fokus: Eine rein objektfokussierte Sicht wie in Self wird nicht realisiert, da TL keine visuelle Programmiersprache sein soll. So wird auf die dreidimensionale Visualisierung und auf die Animation verzichtet. Auch wird durch die generierte Bildschirmrepräsentation nur eine durch die Funktionalität des Dienstes bestimmte Sicht auf die Daten bereitgestellt. Allerdings werden alle Operationen auf einem Bildschirmobjekt durch Knöpfe und Menüs direkt über die Bildschirmrepräsentation angeboten.

Realisierung als generische Bibliothek: Um die Datenvisualisierung für verschiedene Zwecke wie z.B. zur Fehlersuche oder zur Darstellung der Ergebnisse einer Berechnung einsetzen zu können, wird sie als generische Bibliothek realisiert.

4. Grafische Gestaltung der Daten– und Funktionsvisualisierung

Grundlage für die Realisierung des in Abschnitt 3.4 beschriebenen Ansatzes zur Daten– und Funktionsvisualisierung bildet die grafische Gestaltung der zu visualisierenden Objekte. Durch die Wahl geeigneter Bildschirmrepräsentationen soll dem Anwender ein intuitives Verständnis der dargestellten Typen und Werte vermittelt werden. Da sich das **Tycoon** System unter anderem durch seine Offenheit auszeichnet, das die uniforme Verwendung externer Diensterbringer in **TL** erlaubt, erfolgt die Realisierung der Daten– und Funktionsvisualisierung mit Hilfe der systemübergreifenden Klassenbibliothek **StarView** [Busch et al. 93]. Sie ist im Rahmen von [Geisler 95] an das **Tycoon System** angebunden worden und bietet eine Reihe von Basisfunktionalität zur Erzeugung und Verwaltung grafischer Objekte an, so daß die Erstellung grafischer Visualisierungsschnittstellen durch den Programmierer vereinfacht wird. Die wesentlichen Konzepte und Elemente von **StarView** werden im ersten Abschnitt dieses Kapitels vorgestellt und beeinflußen die im zweiten Abschnitt betrachtete grafische Gestaltung der zu visualisierenden Werte und Typen.

4.1 Die grafische Klassenbibliothek StarView

StarView ist eine in der Programmiersprache **C++** [Ellis, Stroustrup 90] implementierte Klassenbibliothek, die die Entwicklung grafischer Anwendungen unterstützt. Sie ist auf vielen verschiedenen Fenstersystemen wie beispielsweise **Solaris**, **Microsoft Windows** sowie **Apple Macintosh System 7** unter **MPW 3.3** einsetzbar. Der mit **StarView** erzeugte Programmkode ist portabel, so daß von den Eigenheiten des jeweiligen Fenstersystems abstrahiert werden kann. Trotzdem bleibt aber das jeweilige *look&feel* des Fenstersystems erhalten. Damit unterscheidet sich **StarView** von grafischen Entwicklungswerkzeugen, die nur für ein spezielles Fenstersystem entwickelt worden sind, wie z.B. das **NeWS Toolkit** [SunSoft 92], das für die Darstellung von Editoren in **Tycoon** verwendet wurde [Müßig 94]. Die Portabilität wird erreicht, indem die Eigenheiten des jeweiligen Fenstersystems in den Methoden der entsprechenden Klassen berücksichtigt werden. Allerdings existieren auch Funktionalitäten, die nicht auf allen Fenstersystemen vorhanden sind wie beispielsweise die Ikonifizierung von Fenstern auf dem **Apple Macintosh**. Da diese Funktionalitäten jedoch nicht überall nachgebildet sind, gilt der Grundsatz der Portabilität nur, wenn der Kode ausschließlich Methoden enthält, die auch auf allen Fenstersystemen implementiert sind. Dieser Ansatz erscheint verbesserungswürdig, da der Programmierer wissen muß, welche Methoden diese Bedingung erfüllen.

Anhand der wichtigsten Klassen gibt Abbildung 4.1[1] einen Überblick über Aufbau und

---

[1] entnommen aus [Busch et al. 93]

Umfang der Klassenbibliothek von StarView.

4.1.1 Die StarView Klassenhierarchie

Die zentrale Basisklasse für alle StarView Applikationen ist die Klasse *Application*. Von ihr werden alle Applikationen abgeleitet. Durch das Überladen ihrer virtuellen Methoden kann eine neue Applikation um die gewünschte Funktionalität erweitert werden. Die für die Datenvisualisierung benötigten GUI Elemente sollen in diesem Abschnitt überblicksartig vorgestellt werden.

Fenster: Sie dienen der Präsentation von Applikationsdaten, der Entgegennahme von Eingabedaten über sogenannte Interaktionselemente und deren Verarbeitung. Die Klasse *Window* stellt die Basisfunktionalität zur Verfügung, die allen Fenstern gemein ist. Nur Fenster, die von der Klasse *SystemWindow* abgeleitet sind, können als eigenständige Einheiten auf dem Bildschirm erzeugt und positioniert werden. Von diesen hängen alle anderen Fenster durch eine bei ihrer Erzeugung definierte und dynamisch rekonfigurierbare Eltern–Kind–Relation transitiv ab. Ihre Positionierung erfolgt relativ zu ihrem Elternfenster. In jeder *StarView* Anwendung muß genau ein als Applikationsfenster ausgezeichnetes Fenster der Klasse *WorkWindow* exisistieren. Eine Interaktion ist nur mit Fenstern möglich, die den Fokus besitzen.

Kontrollelemente: Sie sind abgeleitet von der Klasse *Control* und umfassen unter anderem Knöpfe, Listen, Menüs, Rollbalken, Editierfelder und die Darstellung von Zeichenketten. Der Anwendungsprogrammierer hat darüber hinaus die Möglichkeit, eigene von bereits existierenden Kontrollelementen abgeleitete Kontrollfelder zu definieren. Es werden statische und aktive Kontrollelemente unterschieden. Die aktiven Kontrallelemente ermöglichen Dateneingaben über Maus oder Tastatur, während die statischen lediglich der Präsentation von Daten dienen. Formatierte Eingaben sind über speziell definierte Formatierungsklassen, z.B. *NumericFormatter* für Zahlen oder *DateFormatter* für Datumseingaben, möglich. Jedes Interaktionselement kann vom Benutzer mit Funktionen verknüpft werden, die bei bestimmten, von der jeweiligen Klasse festgelegten Benutzeraktionen, z.B. der Auswahl eines Listeneintrags, ausgeführt werden.

Dialoge: Sie erlauben es, Interaktionselemente in einem Fenster zusammenzufassen. Im Gegensatz zu Fenstern der Klasse *WorkWindow* wird der Fokuswechsel zwischen den verschiedenen in ihnen dargestellten Kontrollelementen und Kontrollgruppen gesteuert, so daß deren geordnete Abarbeitung möglich ist. Es wird unterschieden zwischen modalen und nichtmodalen Dialogen. Modale Dialoge müssen beendet sein, bevor innerhalb der Anwendung fortgefahren werden kann. Einige vordefinierte modale Dialoge unterstützen die Anzeige verschiedenartiger Statusinformationen sowie die Kommunikation mit verschiedenen Systemdiensten.

Datenaustausch: Es werden zwei Arten des formatierten Datentransfers zwischen Anwendungen sowie innerhalb einer Anwendung unterstützt — über das Klemmbrett oder per *Drag&Drop*. Das Klemmbrett stellt eine Art Zwischenspeicher dar, in dem Daten abgelegt und als Kopie wieder abgerufen werden können. Es handelt sich dabei um ein standardisiertes Verfahren der Fenstersysteme, das durch die Klasse *Clipboard* implementiert ist. Es kann zu einem Zeitpunkt nur ein Objekt, allerdings in verschiedenen

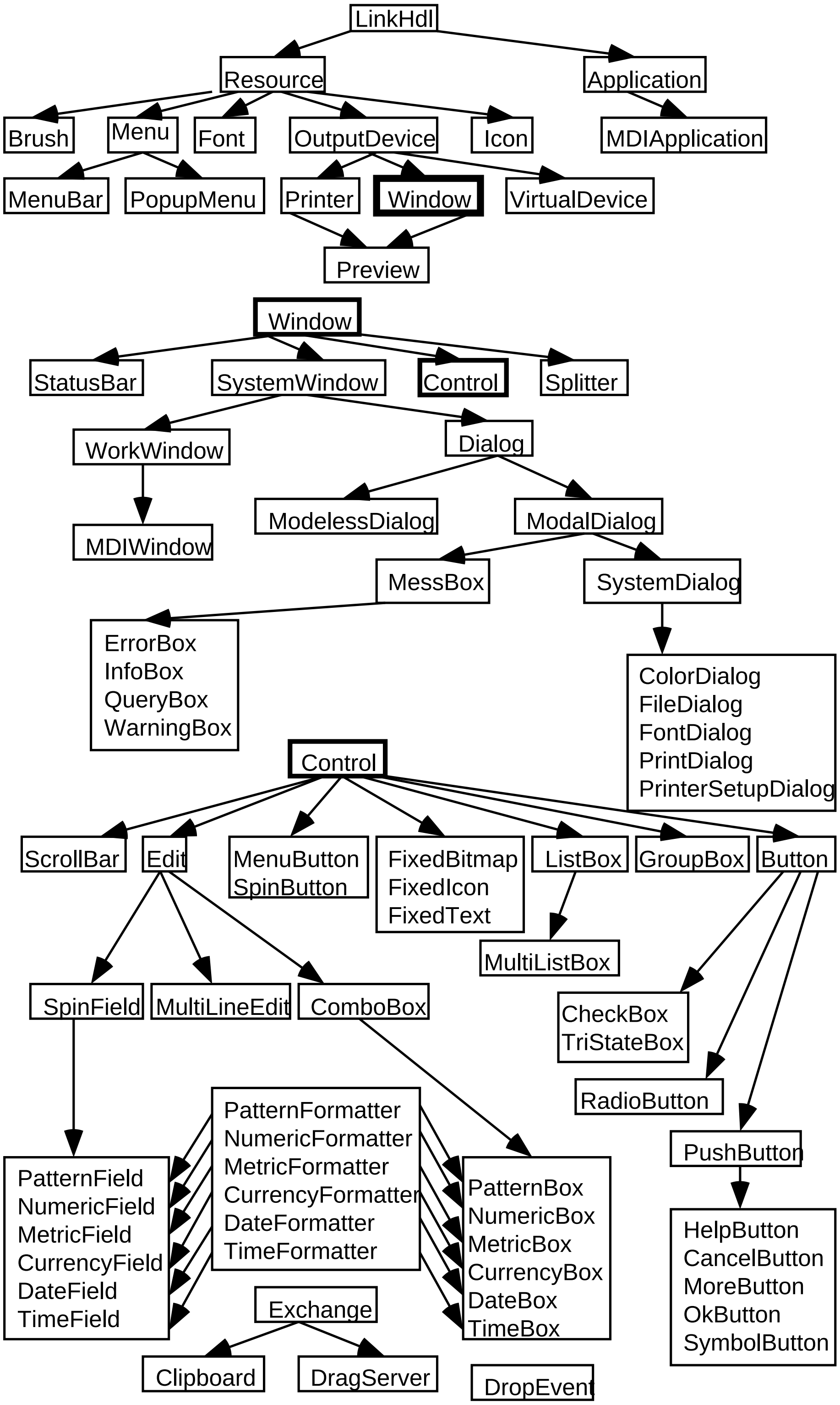

LinkHdl
Resource
Application
Brush
Menu
Font
OutputDevice
Icon
MDIApplication
MenuBar
PopupMenu
Printer
Window
VirtualDevice
Preview
Window
StatusBar
SystemWindow
Control
Splitter
WorkWindow
Dialog
MDIWindow
ModelessDialog
ModalDialog
MessBox
SystemDialog
ErrorBox
InfoBox
QueryBox
WarningBox
ColorDialog
FileDialog
FontDialog
PrintDialog
PrinterSetupDialog
Control
ScrollBar
Edit
MenuButton
SpinButton
FixedBitmap
FixedIcon
FixedText
ListBox
GroupBox
Button
MultiListBox
SpinField
MultiLineEdit
ComboBox
CheckBox
TriStateBox
RadioButton
PushButton
PatternFormatter
NumericFormatter
MetricFormatter
CurrencyFormatter
DateFormatter
TimeFormatter
PatternField
NumericField
MetricField
CurrencyField
DateField
TimeField
PatternBox
NumericBox
MetricBox
CurrencyBox
DateBox
TimeBox
HelpButton
CancelButton
MoreButton
OkButton
SymbolButton
Exchange
Clipboard
DragServer
DropEvent

Abbildung 4.1: Ausschnitt aus der StarView Klassenhierarchie

Formaten, im Klemmbrett abgelegt werden. Beim *Drag&Drop* Verfahren kann der Benutzer ein Objekt mit der Maus anklicken und bei gedrückter Maustaste eine Kopie davon über den Bildschirm ziehen. Wird die Maustaste losgelassen, so wird die Kopie von dem Objekt entgegengenommen, über dem sich der Mauszeiger gerade befindet. Handelt es sich bei dem empfangenden Objekt nicht um ein Interaktionselement, so geht die Kopie verloren. Dieses Verfahren wird mit Hilfe der Klassen *DragServer* und *DropEvent* implementiert.

4.1.2 Ereignisbearbeitung

Bestimmte Ereignisse wie Benutzereingaben oder das Auftreten von Fehlern werden von StarView behandelt, indem benutzerdefinierte Funktionen, sogenannte *Event Handler*, aufgerufen werden. Der Benutzer kann so programmtechnisch auf diese Ereignisse reagieren und entsprechend den Anforderungen des Programmes bestimmte Aktionen auslösen. Dazu stehen zwei Verfahren zur Verfügung:

- ▷ Virtuelle Methoden können in abgeleiteten Klassen überladen werden.
- ▷ Einige der *Event Handler* können mit einer Methode über einen sogenannten *Link* durch eine spezielle Methode zur Installierung des *Event Handlers* verknüpft werden.

Virtuellen Methoden werden bei solchen Klassen verwendet, von denen in der Regel eigene Klassen abgeleitet werden. So kann beispielsweise die Methode zum Zeichnen des Fensterinhaltes in jeder von *Window* abgeleiteten Klasse überladen werden. In TL werden virtuelle Methoden als Attribute einer Klasse aufgefaßt, für die ein Paar von Schreib– und Lesefunktionen erzeugt wird.

Für Ereignisse in Klassen, von denen der Benutzer in der Regel keine eigenen Klassen ableiten muß, wie z.B. bei den aktiven Kontrollfeldern, sind die *Event Handler* vorgesehen. Die Ereignisbearbeitung, die die mit einem bestimmten Ereignis assoziierte Methode aufruft, wird durch das Fenstersystem gesteuert.

4.1.3 Einbindung der Klassenbibliothek in das Tycoon System

In diesem Abschnitt soll die Anbindung der StarView Klassenbibliothek an das Tycoon System überblicksartig erläutert werden (s. Abb. 4.2[2]). Eine detailliertere Beschreibung dazu befindet sich in [Geisler 95].

Das StarView Gateway

Die Einbindung erfolgt mit Hilfe des sogenannten *StarView Gateways*. Dabei handelt es sich um eine Softwarekomponente des Tycoon Systems, die die erforderlichen Bindungen zur auf einem *C++ Server* befindlichen StarView Klassenbibliothek enthält. Es stellt dem TL Programmierer die Methoden der StarView Klassenbibliothek in transparenter Form zur Verfügung. Damit wird von der Existenz eines externen *C++ Servers* abstrahiert und die Bibliothek kann gemäß der TL Syntax und TL Semantik benutzt werden.

Mit Hilfe eines *Gateway Generators*, der die Beschreibung der *Header* Dateien der Klassenbibliothek durch die Schnittstellenbeschreibungssprache GDL erhält, wird das *Gateway* zur

---

[2] entnommen aus [Geisler 95]

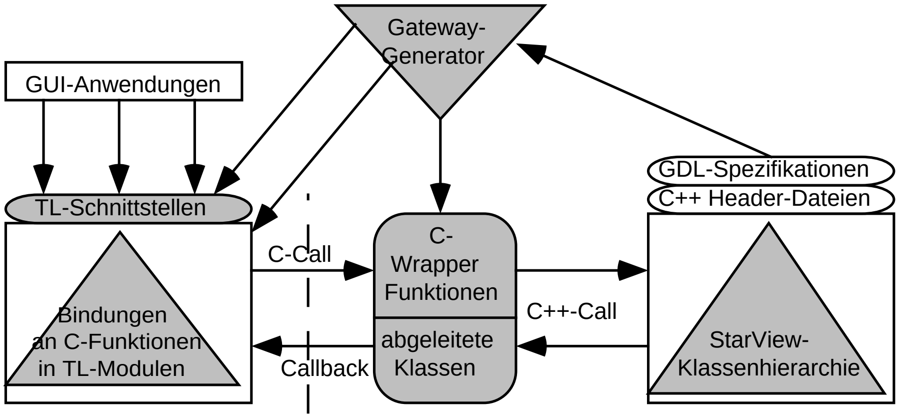

Gateway-
Generator
GUI-Anwendungen
TL-Schnittstellen
Bindungen
an C-Funktionen
in TL-Modulen
C-Call
Callback
C-
Wrapper
Funktionen
abgeleitete
Klassen
C++-Call
GDL-Spezifikationen
C++ Header-Dateien
StarView-
Klassenhierarchie

Abbildung 4.2: Anbindungsszenario für StarView

StarView Klassenbibliothek generiert. Es besteht aus den den C++ Klassen entsprechenden TL Modulen und Schnittstellen sowie den C++ Dateien, in denen jeweils eine Subklasse der zugehörigen C++ Klasse generiert wird und alle nicht-virtuellen Methoden in C Funktionen verpackt und exportiert werden. Bindungen zwischen dem Tycoon System und dem StarView Server werden mittels der bidirektionalen Schnittstelle zwischen TL und C [Mathiske et al. 93] realisiert.

Inkrementelle Entwicklung

Ein StarView Anwendungsprogramm besteht üblicherweise aus drei Teilen:

1. Definition der statischen Struktur der grafischen Interaktionselemente (Fenster, Menüs, Dialoge etc.)
2. Installierung der benutzerdefinierten Aktionen
3. Starten der Ereignisbearbeitung

Mit dem Start der Ereignisbearbeitung, hinter der sich die Ereignisschleife (*Eventloop*) des Basisfenstersystems verbirgt, werden die erzeugten Objekte auf dem Bildschirm sichtbar und können Eingaben und Ereignisse von StarView verarbeitet werden. Erst wenn das Applikationsfenster geschlossen wird, werden die Ereignisschleife wieder verlassen und die Anwendung beendet.

Da die Ereignisschleife Bestandteil eines StarView Programmes ist und durch inkrementelle Übersetzung Änderungen des Programmes nur bei gestarteter Ereignisschleife sichtbar werden, muß die Ereignisschleife erst beendet und wieder neu gestartet werden, bevor die Programmänderungen wirksam werden. Eine typische Anwendung der generischen Daten– und Funktionsvisualisierung ist jedoch die Anbindung an die Tycoon Programmierwerkbank, in der die Ergebnisse von Berechnungen visualisiert werden. Es ist beim Start der Programmierwerkbank nicht bekannt, welche StarView Objekte erzeugt und auf dem Bildschirm angezeigt werden sollen, sondern es werden inkrementell neue grafische Interaktionselemente erzeugt, die auf dem Bildschirm sofort sichtbar sein sollen.

Die Lösung dieses Problems erfolgt in **Tycoon** durch die Möglichkeit, mehrere Kontrollflüsse (*threads of control*) auf demselben Zustand operieren lassen zu können [Matthes, Schmidt 94]. Indem man für die *Eventloop* einer **StarView** Anwendung und den **TL** *Top Level* eigene Kontrollflüsse definiert, kann durch inkrementelle Übersetzungen innerhalb des **TL** *Top Levels* eine **StarView** Anwendung dynamisch geändert werden. Die Auswirkungen dieser Änderungen sind dann sofort sichtbar.

4.1.4 Erfahrungen mit der Einbindung

Ohne größere Probleme ließ sich mit Hilfe des oben beschriebenen Verfahrens die **StarView** Klassenbibliothek unter **Solaris** und **Microsoft WindowsNT** in das **Tycoon** System einbinden. Unter **MPW 3.3** auf dem **Apple Macintosh System 7** traten jedoch mehrere Probleme auf, die sowohl an der Implementierung von **StarView** als auch an den Eigenheiten des Betriebssystems **System 7** liegen.

Auf dem **Apple Macintosh** erfolgt die Interaktion des Benutzers mit dem System innerhalb von Applikationen, von denen zu einem Zeitpunkt jedoch immer nur eine aktiv ist und die Kommunikation zwischen dem Benutzer und dem System übernimmt [Apple Computer 85]. Der Wechsel zwischen zwei Applikationen muß explizit per Mausdruck oder Tastatureingabe erfolgen. Applikationen, die die **StarView** Klassenbibliothek verwenden, müssen immer aus **StarView** heraus gestartet werden. Deshalb ist der **TL** *Top Level*, auf dem die Übersetzung und Ausführung von Programmkode durchgeführt wird, selbst Bestandteil eines **StarView** Programmes und befindet sich innerhalb der *main* Methode der Klasse *Application*. Der *Top Level* ist in der aktuellen Portierung auf den **Apple Macintosh** jedoch mit Hilfe von *TextEdit*, einem Werkzeug der Benutzerschnittstelle zum **Apple Macintosh** Betriebssystem zur Entwicklung von Editoren, implementiert. Die Verwaltung von Ereignissen, die *TextEdit* betreffen, erfolgt durch den *Toolbox Event Manager* des **Apple Macintosh**. Wird vom *TopLevel* jedoch eine **StarView** Anwendung gestartet, so sind zwei Applikationen gleichzeitig aktiv. Der *Toolbox Event Handler* kann jedoch die Ereignisse der **StarView** Applikation nicht verarbeiten, so daß der Rechner abstürzt, sobald die Maus über ein **StarView** Fenster bewegt wird. Im Moment existieren zwei Möglichkeiten, dieses Problem zu lösen:

1. Sobald die Ereignisschleife von **StarView** gestartet wird, wird die Ereignisschleife des *Toolbox Event Managers* eingefroren und es sind keine Eingaben auf dem *Top Level* möglich, bis die **StarView** Anwendung beendet ist.

2. Das **Tycoon** System wird ohne einen eigenen *Top Level* gestartet. Eingaben werden aus einer Datei gelesen und Ausgaben in eine andere Datei geschrieben.

Beide Lösungen sind jedoch wenig befriedigend. Deshalb ist es notwendig, den *Top Level* mit Hilfe der **StarView** Klassenbibliothek anstelle von *TextEdit* zu implementieren.

Ein weiteres Problem ergibt sich unter **TL** bei gleichzeitiger Verwendung der **StarView** Klassenbibliothek und dem **Grand Unified Socket Interface (GUSI)** [Neeracher 95] , einer Bibliothek, die die Kommunikation auf *Socket* Basis auf dem **Apple Macintosh** implementiert. In diesem Fall ist der Empfang von Daten nicht mehr möglich. Es können also nicht beide Bibliotheken gleichzeitig in **TL** verwendet werden. Da die Nutzung von **GUSI** ohne **StarView** zu keinen Problemen führt, verwenden beide Bibliotheken vermutlich gleiche Speicherbereiche, die sie mit eigenen Daten überschreiben, obwohl die andere Bibliothek auf die von ihr erzeugten Daten noch zugreifen möchte.

4.2 Bildschirmrepräsentationen

Die graphische Benutzerschnittstelle der verschiedenen zu visualisierenden Werte und Typen ist in ihrem Aussehen und Verhalten durch die Verwendung der oben beschriebenen StarView Klassenbibliothek und das jeweilige Fenstersystem geprägt. Dabei wird eine für jedes zu visualisierende Objekt O des persistenten Speichers typische Bildschirmrepräsentation $bRep_O$ gewählt. Getrennt nach Typen und Werten werden in den folgenden Abschnitten Aussehen und Verhalten anhand von Abbildungen, die der Darstellung unter *Open Windows* entsprechen, erläutert. Da die Visualisierung von Funktionswerten umfangreicher ist als die anderer Werte, erfolgt deren Beschreibung in einem separaten Abschnitt. Alle visualiserten Objekte werden von einem einheitlichen Rahmen umgeben.

4.2.1 Rahmen für visualisierte Objekte

Ohne einen Rahmen kann keines der zu visualisierenden Objekte dargestellt werden. Er stellt eine den Anforderungen des jeweiligen Objektes entsprechende Grundfunktionalität zur Verfügung. Die Elemente, aus denen ein Rahmen bestehen kann, sind in Abbildung 4.3 dargestellt:

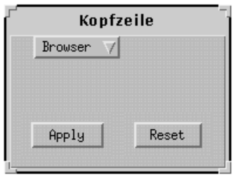


Abbildung 4.3: Mögliche Komponenten eines Rahmens

Grundsätzlich handelt es sich bei dem Rahmen um ein Fenster der Klasse *ModelessDialog*, das von keinem anderen visualisierten Objekt abhängt. Das Schließen eines Rahmens ist nur über den Menüpunkt *Quit* des Menüknopfes möglich.

Jeder Rahmen hat unter der Kopfzeile eine Menüzeile, in der die Menüknöpfe, beginnend auf der linken Seite, angeordnet sind. Am unteren Ende eines Rahmens können zwei weitere Knöpfe erscheinen. Zwischen der Menüzeile und den Knöpfen ist das zu visualisierende Objekt dargestellt.

Kopfzeile

Jeder Rahmen hat eine Kopfzeile, in dem die Art des visualisierten Objektes angezeigt ist. Entsprechend der Namenskonventionen in TL beginnen Typkonstruktoren mit einem Großbuchstaben, also z.B. **Tuple**, und Wertkonstruktoren mit einem Kleinbuchstaben, also z.B. **tuple**. Handelt es sich um einen variablen Wert, so ist zusätzlich das Schlüsselwort **var** vorangestellt.

Menüs

In der aktuellen Version hat der Rahmen nur einen Menüknopf *Browser*, in dem die in Tabelle 4.1 dargestellten Menüeinträge enthalten sein können.

Eintrag	**Subeintrag**	**Funktionalität**
Copy	Copy	In das Klemmbrett kopieren.
	Copy Variant	Variante in das Klemmbrett kopieren.
Paste		Aus dem Klemmbrett kopieren.
Select Element		Anzuzeigendes Feldelement auswählen.
Show	Type	Typ eines Wertes anzeigen.
	Globals	Freie Variablen einer Funktion anzeigen.
	TL	Programmkode einer Funktion anzeigen.
	TML	TML Kode einer Funktion anzeigen.
	Byte Code	Bytekode einer Funktion anzeigen.
Quit		Fenster schließen.

Tabelle 4.1: Funktionalität der Menüeinträge

In Abhängigkeit von den visualisierten Objekten enthält das Menü nur die jeweils benötigten Einträge, z.B. *Copy Variant* nur bei varianten Tupelwerten. Befindet sich kein Objekt im Puffer, so ist der Eintrag *Paste* als inaktiv markiert und nicht auswählbar.

Knöpfe

Nur bei der Visualisierung von veränderlichen Werten der Basistypen wie z.B. Zeichenketten erscheinen die Knöpfe *Apply* und *Reset* im Rahmen. Wird der *Apply* Knopf gedrückt, so wird der Variable der aktuell auf dem Bildschirm sichtbare Wert zugewiesen. Der *Reset* Knopf dagegen liest den Wert aus der Variable und trägt ihn in die Bildschirmrepräsentation ein.

4.2.2 Typen

Die Visualisierung von Typen orientiert sich an den in TL realisierten Typkonzepten. Sie dient dem weitergehenden Verständnis von Werten. Für jeden Typ existieren zwei Arten von Visualisierungen:

1. Die Darstellung der Struktur des Typs.
2. Wenn sie als Typkomponente einer Signatur oder einer Typbindung eingeschachtelt in einem anderen Typ oder Wert dargestellt werden, erfolgt ihre Repräsentation durch einen Knopf. Der Knopf ist mit einer Kurzbeschreibung des Typs beschriftet, die Auskunft darüber gibt, um was für einen Typ es sich handelt. Die Kurzbeschreibung ist entweder ein Typidentifikator oder ein Konstruktor, z.B. *Tuple* für einen Tupeltyp. Durch Drücken dieses Knopfes erscheint die eigentliche Visualisierung des Typs in der anderen Darstellung.

Generell kann man zwischen den Basistypen und den benutzerdefinierten Typen unterscheiden. Für Basistypen und den Supertyp aller nichtparametrisierten Typen **Ok** gibt es keine eigene Repräsentation. Für sie wird nur ein leerer Rahmen mit dem Identifikator des jeweiligen Typs in der Kopfzeile und *Quit* als einzigem Menüeintrag erzeugt. Das Menü aller anderen

Typen enthält zusätzlich den Subeintrag *Copy*, um eine Typbindung in ein visualisiertes *Environment* (s. Abschnitt 7.1) zu kopieren. Im folgenden soll die strukturierte Darstellungsform der verschiedenen zu visualisierenden benutzerdefinierten Typen vorgestellt werden.

Abstrakte Datentypen

Bei abstrakten Datentypen (s. Abb. 4.4) werden der Name und der Supertyp getrennt durch *<:* in einem Knopf dargestellt. Durch Anklicken des Knopfes wird die Struktur des Supertyps in einem neuen Fenster angezeigt. In der Kopfzeile steht **ADT**.

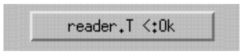


Abbildung 4.4: Abstrakte Datentypen

Strukturierte Typen und Ausnahmetypen

Da die Repräsentation strukturierter Typen der von Ausnahmetypen ähnlich ist, werden beide gemeinsam vorgestellt. Strukturierte Typen werden systematisch durch Aggregation ihrer Komponenten, d.h. Signaturen, erzeugt. Grundsätzlich unterscheidet sich die Visualisierung von varianten Tupeln von der anderer strukturierter Typen.

Tupel–, Rekord- und Ausnahmetypen: Sie werden durch ihre Signaturen dargestellt (s. Abb. 4.5 links). Die Typkomponenten der Signaturen erscheinen links– und deren Identifikatoren rechtsbündig. Getrennt werden sie entweder durch **:** oder **<:**, wenn einer Typvariablen ein Supertyp zugeordnet ist. Können nicht alle Signaturen innerhalb eines Rahmens visualisiert werden, so erscheint rechts neben den Signaturen ein Rollbalken, mit dem innerhalb der Signaturen navigiert werden kann. Nur durch den Eintrag in der Kopfzeile ist feststellbar, ob es sich um einen Tupel–, Rekord– oder Ausnahmetyp handelt.

Variante Tupeltypen: Sie verfügen zusätzlich zu den Signaturen über einen Auswahlknopf, durch den die zu visualisierende Variante ausgewählt werden kann (s. Abb. 4.5 rechts). Er zeigt den Namen der gerade sichtbaren Variante an. Die Größe des Rahmens wird jeweils entsprechend der Anzahl der Signaturen jeder Variante angepaßt. Grundsätzlich wird bei der Visualisierung eines varianten Tupeltyps zunächst die erste Variante dargestellt. Wird der Subeintrag *CopyVariant* des Menüs gedrückt, so wird die aktuell ausgewählte Variante als Tupel in das Klemmbrett kopiert.

Funktionstypen und Typoperatoren

Da Typoperatoren Funktionen sind, die Typen auf Typen abbilden, werden sie ähnlich wie Funktionen dargestellt (s. Abb. 4.6). Sie unterscheiden sich nur durch den Eintrag in der Kopfzeile, durch den ersichtlich ist, ob es sich bei dem visualisierten Objekt um einen Funktionstyp oder um einen Typoperator handelt.

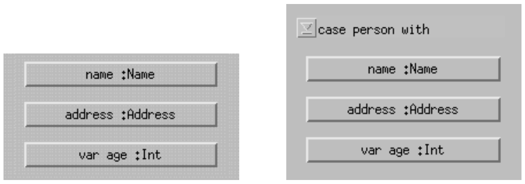

name :Name
address :Address
var age :Int
case person with
name :Name
address :Address
var age :Int

Abbildung 4.5: Strukturierte Typen

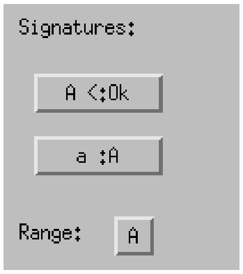

Signatures:
A <:Ok
a :A
Range:
A

Abbildung 4.6: Funktionstypen

Beide bestehen aus einer Liste von Signaturen, deren Beginn durch das Wort *Signatures* gekennzeichnet ist, und einem durch *Range* gekennzeichneten Bereichstyp. Wie bei strukturierten Typen ist es möglich, daß die Signaturen mit einem Rollbalken versehen werden.

Typoperatoranwendungen

Bei der Visualisierung von Typoperatoranwendungen werden — gekennzeichnet mit *Type Operator* — der Typoperator durch einen Knopf und — eingeleitet durch *Type Bindings* — die Liste der Typbindungen ebenfalls durch Knöpfe dargestellt (s. Abb. 4.7).

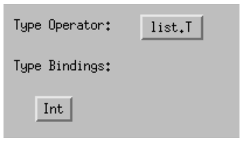

Type Operator:
list.T
Type Bindings:
Int

Abbildung 4.7: Typoperatorapplikationen

Eine Ausnahme bilden Felder, da diese zu den vordefinierten Typen in TL gehören. Bei ihnen wird nur der Elementtyp als Knopf visualisiert.

Rekursive Typen

Bei der Visualisierung rekursiver Typen werden jeweils alle parallel definierten Typbindungen angezeigt (s. Abb. 4.8). Die Typbindung, die den visualisierten Typ repräsentiert, ist durch einen Symbolknopf gekennzeichnet. Jede Typbindung besteht aus einem Typidentifikator, einem Supertyp und der Typdefinition.

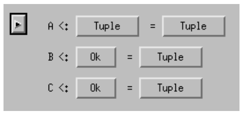


Abbildung 4.8: Rekursive Typen

4.2.3 Werte

Analog zur Visualisierung von Typen gibt es auch bei Werten zwei Darstellungsformen. Für jeden Wert existieren der Subeintrag *Copy* sowie die Einträge *Type* und *Quit* im Menü. Handelt es sich um einen variablen Wert, so erhält das Menü zusätzlich noch den Eintrag *Paste*. Solche Werte können durch Tastatureingaben, durch das Kopieren eines Wertes aus dem Klemmbrett oder durch *Drag&Drop* verändert werden.

Werte von Basistypen

Bei Wertbindungen ist zwischen konstanten und variablen Wertbindungen zu unterscheiden. Für konstante Wertbindungen von Basistypen gibt es nur eine Repräsentation. Sie werden mit Ausnahme von booleschen Werten, bei denen kein Unterschied in der Repräsentation zwischen konstanten und variablen Wertbindungen existiert, textuell dargestellt. Als Komponenten anderer Wertbindungen werden sie direkt innerhalb der Bildschirmrepräsentation dieses Wertes visualisiert. Variable Wertbindungen werden dagegen als Komponenten anderer Werte nur durch einen Knopf repräsentiert, auf dem der Typ des Wertes steht. Durch Drücken des Knopfes erhält man die eigentliche Repräsentation der veränderlichen Wertbindung, die durch die von StarView zur Verfügung gestellten Klassen zur formatierten Anzeige von Werten vorgegeben ist. Diese Unterscheidung ist darin begründet, daß Objekte zwar auf dem Bildschirm nur einmal visualisiert werden sollen, konstante Wertbindungen jedoch nicht veränderlich sind und somit der Übersicht halber als Komponenten anderer Wertbindungen direkt visualisiert werden können. Im folgenden werden die verschiedenen Bildschirmrepräsentationen variabler Wertbindungen näher erläutert.

Ganzzahlen: Werte des Typs *Int* werden durch ein numerisches Eingabefeld visualisiert (s. Abb. 4.9). Variable Werte können durch Tastatureingaben geändert oder durch zwei Knöpfe, die sich rechts neben dem Eingabefeld befinden, in– bzw. dekrementiert werden.

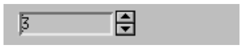


Abbildung 4.9: Integerwerte

Fließkommazahlen: Werte des Datentyps *Real* werden wie ganze Zahlen dargestellt, nur daß es bei variablen Fließkommazahlen keine Knöpfe zum In– bzw. Dekrementieren gibt (s. Abb. 4.10). Die Genauigkeit ist, bedingt durch **StarView**, auf zwei Stellen hinter dem Komma begrenzt.

3,12

Abbildung 4.10: Fließkommazahlen

Zeichenketten: Werte des Datentyps *String* werden durch ein alphanumerisches Eingabefeld repräsentiert (s. Abb. 4.11). Reicht das Eingabefeld nicht zur Darstellung der gesamten Zeichenkette aus, so kann der Inhalt über die Tastatur nach rechts bzw. links gerollt werden.

Zeichenkette

Abbildung 4.11: Zeichenketten

Zeichen: Werte des Datentyps *Char* werden als Zeichenketten der Länge eins visualisiert.

Boolesche Werte: Werte des Datentyps *Bool* werden durch einen Selektionsknopf dargestellt, dessen Wert durch Anklicken mit der Maus verändert werden kann. Der rechte Knopf in Abbildung 4.12 repräsentiert den Wert *true* und der linke den Wert *false*.

Abbildung 4.12: Boolesche Werte

Werte abstrakter Datentypen

Da bei Werten abstrakter Datentypen nur deren Supertyp bekannt ist, ist deren Visualisierung nicht möglich. Sie werden stattdessen durch die Zeichenkette <*hidden*> repräsentiert (s. Abb. 4.13). Variable Wertbindungen können per Tastatur nicht verändert werden, da ihre Manipulationsoperationen nicht bekannt sind. Der Datenaustausch über das Klemmbrett und per *Drag&Drop* ist dagegen erlaubt.

Strukturierte Werte

Werte von Tupeln, varianten Tupeln oder Rekordtypen werden durch die Bindungen ihrer Komponenten beschrieben (s. Abb. 4.14). Der Name jeder Bindung, dem das Schlüsselwort

Abbildung 4.13: Werte abstrakter Datentypen

var bei einer variablen Wertbindung vorangestellt ist, erscheint rechtsbündig als fixierter Text, ihre Typ– bzw. Wertkomponente als Knopf oder in der entsprechenden Repräsentationsform, falls es sich um eine konstante Wertbindung eines Basistyps handelt. Bei varianten Tupeln wird zusätzlich der Name der Variante als fixierter Text angezeigt. Können nicht alle Komponenten in dem Rahmen visualisiert werden, so kann mit einem Rollbalken rechts neben den Komponenten navigiert werden.

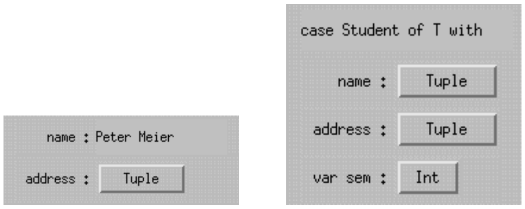

name : Peter Meier

Abbildung 4.14: Strukturierte Werte

Ausnahmewerte

Bei Ausnahmewerten werden analog zu Ausnahmetypen alle Signaturen visualisiert. Der Name des Ausnamewertes erscheint in der Kopfzeile.

Felder

Da die Feldelemente variabel sind, wird für jedes Feldelement ein Knopf erzeugt, der als Eintrag den Index des Elementes und den Elementtyp erhält (s. Abb. 4.15). Lassen sich nicht alle Elemente in einem Rahmen visualisieren, so kann mit Hilfe eines Rollbalkens navigiert werden. Die Visualisierung eines Elementes erfolgt durch Drücken des entsprechenden Knopfes oder durch Auswahl über den Menüeintrag *Select Element*.

4.2.4 Funktionswerte

Funktionen sind in TL Werte erster Klasse, die im Rumpf sowohl ihre formalen Parameter und lokal definierten Bindingen als auch lexikalisch außerhalb von ihnen definierte Bindungen referenzieren dürfen. Da global in Funktionen referenzierte Bindungen aber den Wert überleben können, in dem sie definiert werden, werden sie mit der Funktion zusammen in einem Funktionsabschluß abgespeichert. Damit ist es für das Verständnis einer Funktion in TL notwendig, ihre Signatur, ihren Programmkode und ihre globalen Bindungen zu visualisieren. Da im Abschluß einer Funktion jedoch auch eine persistente Repräsentation des TML Kodes und

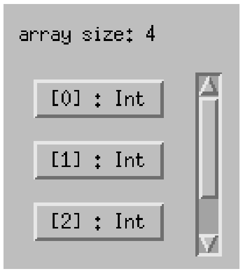
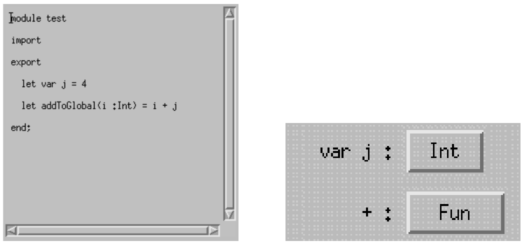


Abbildung 4.15: Felder



```
module test
import
export
  let var j = 4
  let addToGlobal(i :Int) = i + j
end;
```

Abbildung 4.16: Programmkode und globale Variablen einer Funktion

der TVM Kode gespeichert sind, sollen diese zu Testzwecken auch im Rahmen dieser Arbeit visualisiert werden.

Zunächst wird nur die Signatur einer Funktion angezeigt. Da sie dem Typ einer Funktion gleicht, wird sie auch entsprechend visualisiert. In getrennten Fenstern können Bildschirmrepräsentationen der anderen Komponenten durch Auswahl des entsprechenden Submenüeintrags erzeugt werden. Die Visualisierung der globalen Variablen erfolgt analog zu Bindungen in strukturierten Werten (s. Abb. 4.16 links). Die verschiedenen Koderepräsentationen werden durch ein mehrzeiliges manipulierbares Editierfenster dargestellt. Mit Hilfe von Rollbalken kann sowohl in vertikaler als auch in horizontaler Richtung navigiert werden. Um das Verständnis des Programmkodes zu erhöhen, wird das gesamte Modul angezeigt, in dem die Funktion definiert worden ist (s. Abb. 4.16 rechts).

Um die Funktionsvisualisierung zu vervollständigen, soll der Anwender auch in der Lage sein, eine visualisierte Funktion interaktiv auszuführen. Deshalb wird unterhalb der Funktionssignatur ein weiterer Knopf mit der Aufschrift *execute* generiert. Wird dieser Knopf gedrückt, so wird die Funktion ausgeführt. Zur Zeit ist diese Funktionalität aber nur für parameterlose Funktionen realisiert.

5. Dynamische Typisierung

Die typsichere generische Datenvisualisierung erfordert die Inspektion von Typinformationen zur Laufzeit, um beliebige Daten entsprechend ihrer Struktur korrekt visualisieren zu können. Deshalb müssen strikt typisierte Programmiersprachen wie TL um ein Konzept zur dynamischen Typisierung erweitern werden (s. Abschnitt 3.3.1). Auch der ursprüngliche Sprachreport von TL [Matthes, Schmidt 92] enthält Sprachkonstrukte, die die dynamische Typisierung ermöglichen. Allerdings wurde dieses Konzept nie vollständig realisiert, da es bei der Implementierung des Kodes zur Laufzeitunterstützung verschiedene Unzulänglichkeiten offenbarte (s. Abschnitt 5.4.1).

Aus diesem Grund erfolgt im Rahmen dieser Arbeit in Kooperation mit [Geisler 95] die Realisierung eines verbesserten, neuen Ansatzes. Nach einer Begriffseinführung zur dynamischen Typisierung werden im zweiten Abschnitt verschiedene, über die generische Datenvisualierung hinaus für Tycoon relevante Anwendungsgebiete dynamischer Typisierung vorgestellt. Anschließend erfolgt eine Analyse bekannter Ansätze zur Realisierung dynamischer Typisierung innerhalb statischer Programmiersprachen, unter besonderer Berücksichtigung der Probleme, die sich in Verbindung mit abstrakten Datentypen und Polymorphismus ergeben. Ausgehend von diesen Untersuchungen wird schließlich der Ansatz erläutert, der in der aktuellen Version des Tycoon Systems implementiert worden ist.

5.1 Laufzeittyprepräsentationen und automorphe Werte

Im Gegensatz zu den Programmteilen, die bereits statisch typgeprüft worden sind, müssen Typinformationen von Werten, deren Typkorrektheit erst zur Laufzeit festgestellt werden kann, als Laufzeitwerte repräsentiert werden. Solche Werte werden in der Literatur in der Regel als *dynamische Typen* bezeichnet [Matthews 87; Abadi et al. 92; Leroy, Mauny 91]. Diese Bezeichnung ist jedoch irreführend, da es sich um Werte handelt, die statische Typen repräsentieren, so daß hier ein anderer Begriff verwendet wird.

In TL ist die Gültigkeit eines Typs T immer auf einen statischen Kontext S beschränkt. Mit $t_{S,T}$ sei die kontextunabhängige Typrepräsentation des Typs T in dem statischen Kontext S zur Laufzeit bezeichnet. Die häufigste Verwendung von Laufzeittyprepräsentationen erfolgt in Verbinding mit solchen Werten, deren Typ durch die Laufzeittyprepräsentation beschrieben wird. Solch ein Aggregat aus Wert und zugehöriger Laufzeittyprepräsentation wird in der Literatur als *automorpher* (selbstbeschreibender) Wert [Cardelli 89] oder als *dynamic* [Abadi et al. 92] bezeichnet. Ein *automorpher* Wert ist also ein Paar (x, $t_{S,T}$), so daß $t_{S,T}$ eine Laufzeitrepräsentation des Typs des Wertes x darstellt.

5.2 Anwendungen dynamischer Typisierung

Es gibt eine Reihe von Anwendungen, in denen die Typinformation erst zur Laufzeit bekannt ist, so daß sie mit Hilfe statischer Typsysteme nicht zu realisieren sind. Diese werden innerhalb dieses Abschnittes charakterisiert und anhand von Beispielen, die für das Tycoon System relevant sind, untersucht. Diese Charakterisierung stellt ein wichtiges Kriterium für die Entwicklung eines Konzeptes zur Integration dynamischer Typisierung in TL dar.

Als ein typischer Vertreter von Anwendungen, die auf dynamische Typisierung angewiesen sind, galten ursprünglich generische Funktionen, d.h. Funktionen die auf Werte verschiedener Typen angewendet werden können, aber ein Ergebnis liefern, das unabhängig von deren Typstruktur berechnet wird [Leroy, Mauny 91]. Ein Beispiel dafür ist die Funktion, die die Anzahl der Elemente einer Liste eines beliebigen Typs ermittelt. Die Integration verschiedenener Arten von Polymorphismus in das statische Typsystem ermöglichte jedoch schließlich die statische Typüberprüfung solch generischer Anwendungen.

Die Frage, ob dynamische Typisierung in statischen Typsystemen generell überflüssig ist, läßt sich mittels der in [Cardelli, Wegner 85] erfolgten Charakterisierung der unterschiedlichen Arten des Polymorphismus beantworten. Generell wird zwischen zwei Arten von Polymorphismus unterschieden:

Universeller Polymorphismus wird bei Funktionen verwendet, die auf einer unendlichen Menge verschiedener Typen arbeiten, die alle über eine gemeinsame Struktur verfügen.

Ad–hoc–Polymorphismus erlaubt die Formulierung typgesteuerter Funktionen, die auf einer endlichen Menge verschiedener Typen arbeiten, die keine gemeinsame Struktur besitzen. Die Ausführung solch einer Funktion hängt vom Typ des der Funktion übergebenen Parameters ab.

Beim *universellen* Polymorphismus unterscheidet man wiederum

Parametrischen Polymorphismus, der die oben vorgestellten generischen Funktionen realisiert, die einen impliziten oder expliziten Typparameter erhalten, der den Typ jeder Anwendung der Funktion spezifiziert, und

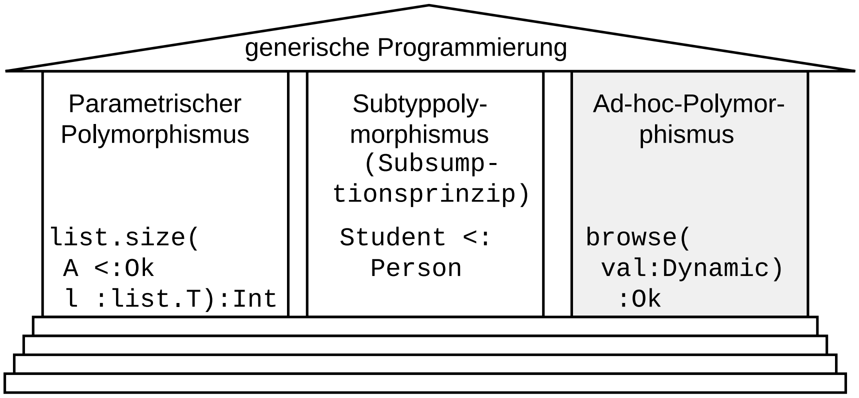
Inklusionspolymorphismus, der es ermöglicht, ein Objekt als Mitglied verschiedener, nicht unbedingt disjunkter Klassen zu betrachten. Eine Instanz des *Inklusions*polymorphismus ist der *Subtyp*polymorphismus [Cardelli 84], in dem ein Typ Subtyp eines anderen Typs ist (z.B. *Student <:Person*). In diesem Fall kann überall dort, wo ein Wert vom Typ *Person* erwartet wird, auch ein Wert des Typs *Student* verwendet werden.

Sowohl der parametrische als auch der Subtyp Polymorphismus sind in TL in das statische Typsystem integriert. Da der Kontext einer *ad–hoc*–polymorphen Funktion, in dem ein Wert erzeugt wird, und der Kontext, in dem der Wert benutzt wird, keine gemeinsame statisch überprüfbare Spezifikation besitzen, benötigen solche Anwendungen dynamische Typisierung. Abbildung 5.1[1] illustriert die Arten des Polymorphismus, die in TL die Säulen der generischen Programmierung darstellen.

In den folgenden Abschnitten werden einige Beispiele für generische, datenstrukturabhängige Funktionen vorgestellt, die sich grob in folgende Anwendungsklassen unterteilen lassen:

---

[1] leicht modifiziert aus [Geisler 95] übernommen


generische Programmierung
Parametrischer Polymorphismus
Subtyppoly-morphismus (Subsump-tionsprinzip)
Ad-hoc-Polymor-phismus

```
list.size(
 A <:Ok
 l :list.T):Int
```

```
Student <:
   Person
```

```
browse(
 val:Dynamic)
  :Ok
```

Abbildung 5.1: Säulen der generischen Programmierung in TL

- Meta–Level Operationen, die reflektives Wissen des Compilers benötigen
- strukturierte Ein– und Ausgabefunktionen
- Generatoren, die unterschiedlichen Kode für unterschiedliche Eingabetypen generieren
- Bindungen an entfernte Prozeduraufrufe

5.2.1 Meta–Level Operationen

Als *Meta–Level Operationen* werden hier solche Operationen bezeichnet, die reflektives Wissen des Compilers benötigen. Ein typischer Vertreter solcher Operationen ist die Funktion

eval(sourceText :String) :???

die ein Stück Programmkode erhält und unter Ausnutzung von Laufzeitreflektion (s. Abschnitt 3.3.3) übersetzt und ausführt. In untypisierten Sprachen wie Lisp oder SNOBOL4 [Griswold et al. 71] läßt sich die *eval* Funktion ohne dynamische Typisierung realisieren. In typsicheren Sprachen wie TL dagegen muß die Bindung eines *eval* Aufrufs an einen Bezeichner typgeprüft werden. Wie durch die Fragezeichen angedeutet ist, kann der Ergebnistyp dieses *eval* Aufrufes jedoch nicht statisch ermittelt werden, da er vom übergebenen Programmkode abhängt. Deshalb muß als Ergebnis dieser Funktion ein automorpher Wert erzeugt werden, der eine Typüberprüfung zur Laufzeit ermöglicht.

5.2.2 Strukturierte Ein– und Ausgabe

Im Tycoon–System wird nicht nur der Quelltext, sondern auch der zugehörige, ausführbare Kode von Modulen und ihren Schnittstellen als separate Dateien abgespeichert. Dies erfordert zwei Funktionen:

*extern(object :??? writeHdl :FileHandle) :**Ok***

schreibt ein im persistenten Speicher erzeugtes Objekt beliebigen Typs linearisiert in eine Datei und *intern(readHandle :FileHandle) :???*

liest ein Objekt beliebigen Typs aus einer Datei in den persistenten Speicher.

Das in eine Datei geschriebene Objekt muß die Eigenschaft besitzen, daß es selbstbeschreibend ist und somit keine Referenzen auf andere Objekte außerhalb der Datei enthält [Cardelli 89]. Da automorphe Werte diese Eigenschaft erfüllen, sind beide Funktionen typische Anwendungen dynamischer Typisierung. Um ein Objekt mittels der *extern* Funktion in eine Datei schreiben zu können, muß ein Konstruktor existieren, der aus einem beliebigen Wert einen automorphen Wert erzeugt. Um andererseits auf einem Objekt, das aus einer Datei in den persistenten Speicher gelesen worden ist, Operationen ausführen zu können, muß eine weitere Operation existieren, die einen automorphen Wert $(x, t_{S,T})$ auf einen Wert x mit einem statischen Typ T' in einem Kontext S' projiziert, wobei $T' <: T$ in S' gilt. Diese Operation führt also zur Laufzeit einen Typtest durch und propagiert ein Ausnahmepaket, falls dieser Typtest fehlschlägt.

Eine andere Form der strukturierten Ausgabe, die Gegenstand dieser Arbeit ist, ist die generische Visualisierung solcher Daten und Funktionen auf dem Bildschirm, die sich innerhalb des persistenten Speichers befinden. Um für ein zu visualisierendes Objekt die entsprechende Darstellungsform generieren zu können, ist eine Typanalyse des Objektes notwendig. Da es jedoch potentiell unendlich viele verschiedene Typen innerhalb einer solch komplexen Sprache wie TL gibt, läßt sich statisch nicht feststellen, welche Typen die zu visualisierenden Objekte eventuell haben können. Deshalb muß die generische Datenvisualisierung mittels automorpher Werte erfolgen, die eine Typanalyse zur Laufzeit ermöglichen. Das gleiche Problem tritt bei der strukturierten Eingabe mittels generischer Editoren auf [Müßig 94]. In beiden Fällen muß es einen Mechanismus geben, mittels dessen man die Struktur von Objekten inspizieren kann, um anhand ihrer Typinformationen entscheiden zu können, welche Operationen durchzuführen sind.

5.2.3 Generatoren

Generatoren bilden ein weiteres Anwendungsgebiet für dynamische Typsierung. Ein Operator in relationalen Datenbanken ist der sogenannte Verbindungsoperator *join*, der zwei Relationen kombiniert und dann denjenigen Teil der Ergebnisrelation auswählt, der mindestens je ein Attribut gleichen Werts aus jeder Relation enthält. Eine Verallgemeinerung davon ist der natürliche Verbindungsoperator *natural join*, bei dem die Ergebnisrelation nur eines der überlappenden Attribute enthält [Schmidt 87]. Ein Generator, der für zwei beliebige Relationen den Programmkode generiert, der die natürliche Verbindungsoperation auf diesen beiden Relationen durchführt, läßt sich aus folgenden Gründen nicht als generische Funktion realisieren [Stemple et al. 90]:

- Die Konkatenierungsfunktion, die die Ergebnisrelation erzeugt, muß den Typ der Ergebnisrelation in Abhängigkeit von den Typen der zu verbindenden Relationen ermitteln.

- Die Funktion, die zwei Tupelkomponenten auf Gleichheit überprüft muß sowohl den Namen der jeweiligen Komponenten als auch deren Typ auf Gleichheit testen. Die Namen und Typinformationen der Attribute sind jedoch erst zur Laufzeit bekannt, so daß der Test dynamisch erfolgen muß.

- ▷ Die Funktion, die Tupel beider Relationen aggregiert und dabei nur ein Attribut der überlappenden Komponenten erzeugt, muß ebenfalls in der Lage sein, die Namen der Attribute und deren Typ auf Gleichheit zu testen.

Damit sind die Typen der Elemente beider Relationen statisch nicht festgelegt:

joinGenerator(typ1, typ2 :???) :String

Dieses Beispiel macht deutlich, daß ein Konzept zur dynamischen Typisierung neben automorphen Werten auch Laufzeittyprepräsentationen umfassen muß.

5.2.4 Bindungen an entfernte Prozeduraufrufe

Eines der Ziele im Tycoon Projekt ist es, das System so skalieren zu können, daß die Sprache TL als Beschreibungssprache für verteilte Anwendungen genutzt werden kann [Mathiske et al. 95b]. Dabei bilden synchrone entfernte Prozeduraufrufe (*RPC*) die bevorzugte Technik für die Entwicklung von *Client–Server* Anwendungen [Corbin 91], da sie eine hohe Ebene der Kommunikation zur Verfügung stellen, auf der Klientenprogramme Prozeduren aus anderen, entfernten Anbieterprogrammen aufrufen können [Birrel, Nelson 84]. Die Auswahl des Anbieters sowie die Bindung an eine entfernte Prozedur erfolgen dynamisch zur Laufzeit, da die Programme auf dem Anbieter und dem Klienten unabhängig voneinander entwickelt, übersetzt und gebunden werden. Das kann dazu führen, daß die Signatur des Prozeduraufrufs durch den Klienten nicht mit der Signatur der Prozedur des Anbieters übereinstimmt, so daß die Typkorrektheit des entfernten Prozeduraufrufs zur Laufzeit überprüft werden muß, um die Typsicherheit gewährleisten zu können. Damit bilden auch *RPCs* eine typische Anwendung dynamischer Typisierung.

5.2.5 Aufgaben dynamischer Typisierung

Aus den in den vorhergehenden Abschnitten vorgestellten Beispielen lassen sich zwei wesentliche Arten von Anwendungen definieren, die nur mittels dynamischen Typisierung realisierbar sind:

1. Die *Manipulation von Werten mit dynamischer Typinformation*, die nicht statisch zur Übersetzungszeit bestimmt werden kann, ist immer dann notwendig, wenn Werte, die außerhalb der statischen Umgebung des Typüberprüfers generiert werden, manipuliert werden müssen. Zu dieser Art von Anwendungen gehören die *eval–, intern–* und *extern–* Funktionen sowie Bindungen an enfernte Prozeduraufrufe.

2. Das *Inspizieren von Typinformationen* zur Laufzeit ist notwendig, um hochgenerische, datenstrukturabhängige Algorithmen zu entwickeln [Stemple et al. 90; Stemple et al. 92a; Kirby 92]. Beispiele dafür sind die generische Datenvisualisierung, die generische Datenmanipulation und die Generatoren zum Erzeugen von Programmkode.

Um diesen Aufgaben gerecht zu werden, muß ein Konzept zur Realisierung dynamischer Typisierung folgende Komponenten umfassen:

- ▷ *Datentypen* für Laufzeittyprepräsentationen und automorphe Werte, sowie Konstruktoren zum Erzeugen von Werten dieser Datentypen,

- einen *Projektionsoperator*, der einen automorphen Wert auf seine Wertkomponente projiziert und dabei zur Laufzeit überprüft, ob der Typ, den der Wert erhalten soll, ein Supertyp der Typkomponente des automorphen Wertes ist,
- Operationen zum *Inspizieren* von Laufzeittyprepräsentationen und automorphen Werten,
- eine Operation, die einen *Subtyptest* auf Laufzeittyprepräsentationen durchführt, und
- *Konstruktoren*, die aus Laufzeittyprepräsentationen und automorphen Werte strukturierte Laufzeittyprepräsentationen und automorphe Werte erzeugen

5.3 Konzepte zur Realisierung dynamischer Typisierung

In einer Reihe statisch typisierter Programmiersprachen sind bereits Konzepte für die Integration dynamischer Typisierung realisiert worden. Darüber hinaus gibt es theoretische Ansätze, in denen die Interaktion zwischen statisch und dynamisch überprüften Teilen einer Programmiersprache untersucht werden. Diese Konzepte und Ansätze werden in diesem Abschnitt vorgestellt und anhand bestimmter Kriterien analysiert. Dazu werden im folgenden Abschnitt zunächst Ansätze in monomorphen Sprachen vorgestellt. Besondere Probleme ergeben sich bei der Behandlung abstrakter Datentypen und des Polymorphismus in Verbindung mit dynamischer Typisierung, so daß diese Aspekte jeweils in einem eigenen Abschnitt behandelt werden.

5.3.1 Dynamische Typisierung in anderen Sprachen

Die Analyse der betrachteten Sprachen orientiert sich im wesentlichen an den folgenden, für die Entwicklung eines Konzepts zur dynamischen Typisierung in **Tycoon** wichtigen Fragestellungen:

- Welche Typen bietet die Sprache für die dynamische Typisierung an?
- Welche Sprachkonstrukte werden für Operationen auf diesen Typen angeboten und welche Funktionalität wird durch diese bereitgestellt?
- Welche zusätzlichen Operationen existieren auf diesen Typen?

Ziel bei der Auswahl der Sprachen ist es, möglichst verschiedene Konzepte und Lösungsansätze vorzustellen. Andere Sprachen, die ebenfalls über Konzepte zur dynamischen Typisisierung verfügen, die hier aber nicht vorgestellt werden, sind **CLU** [Liskov et al. 81] und **Cedar/Mesa** [Lampson 83], deren Hauptziel die Unterstützung von Programmidionomen aus Lisp ist, sowie **Modula-2+** [Rovner 86] zur Integration persistenter Daten, und **Modula-3** [Cardelli et al. 88; Cardelli et al. 89], in denen die Ansätze aus **Cedar/Mesa** direkt übernommen worden sind.

Amber

Amber ist eine funktionale, polymorphe und modulare Sprache, bei deren Entwicklung eines der Hauptziele die Integration dynamischer Typisierung in eine statisch typisierte Sprache war. Sie kann auch als Systemprogrammiersprache verwendet werden [Cardelli 86].

Dynamic ist der Typ automorpher Werte in **Amber**. Auf Werten dieses Typs existieren zwei Sprachkonstrukte:

dynamic *e*
coerce *e* ***to*** *t*

Mit *dynamic* wird aus einem Wert e, der einen beliebigen Typ haben kann, ein automorpher Wert erzeugt. Falls ein automorpher Wert *e* den Typ *t* hat, so kann man diesen Wert mittels *coerce* auf einen Wert des Typs *t* projizieren. Dazu erfolgt zur Laufzeit eine Typüberprüfung zwischen der Typkomponente des automorphen Wertes *e* und dem Typ *t*, die im Fehlerfall einen Laufzeitfehler erzeugt.

Die Hauptanwendung automorpher Werte in Amber ist die Handhabung persistenter Daten, die mittels *extern* als automorpher Wert im Dateisystem gespeichert und mittels *intern* geladen werden [Abadi et al. 89].

Die Verwendung von *coerce* ist nur dann sinnvoll, wenn der Typ des automorphen Wertes bereits bekannt ist. Ist dies jedoch nicht der Fall, so bietet Amber noch eine Reihe von Primitiven zum Inspizieren und Manipulieren automorpher Werte mit unbekanntem Typ. Das Primitiv *typeOf* liefert eine Kodierung des Typs eines dynamisch typisierten Wertes *val*, indem der Typ von *val* um einen Schritt expandiert wird. Handelt es sich dabei nicht um einen Basistyp, so kann man die Struktur des Typs noch weiter analysieren. Weitere Primitive erlauben das schrittweise Expandieren eines automorphen Wertes, dessen Typ kein Basistyp ist. Beispielsweise gibt das Primitiv *exposeRecord*, falls es sich bei dem übergebenen automorphen Wert um einen *Record* handelt, eine Liste zurück, deren Elemente jeweils aus dem Namen und dem automorphen Wert einer Komponente des *Records* bestehen. Analog existieren Primitive für Varianten, Felder, Werte mit rekursivem Typ und Funktionen mit einem Übergabeparameter.

CAML

CAML [Weis 90], eine Implementierung von ML [Milner et al. 90], ist eine funktionale, polymorphe Programmiersprache, die gegenüber ML um ein Konzept zur dynamischen Typisierung erweitert worden ist [Leroy, Mauny 91]. ML ist eine implizit typisierte Programmiersprache. Das bedeutet, daß bei der Typüberprüfung jeweils der generellste Typ für einen Wert inferiert wird. Darüber hinaus existiert das Konzept der Musterüberprüfung, das den Vergleich des Formats eines Wertes mit dem Format einer Schablone erlaubt.

In CAML gibt es einen vordefinierten Datentyp *dyn*, der diejenigen Werte repräsentiert, die bezüglich ihres Typs selbstbeschreibend sind. Er entspricht damit dem Typ *Dynamic* in Amber. Auf Werten des Typs *dyn* gibt es das Konstrukt *dynamic e*, das den Ausdruck *e* evaluiert und aus dem Evaluationsergebnis einen automorphen Wert erzeugt.

Da das Binden von Werten an Bezeichner und die Fehlerbehandlung bereits durch die Musterüberprüfung abgedeckt sind, wird in CAML kein dem *coerce* in Amber entsprechendes Konstrukt benötigt. Stattdessen existiert eine zusätzliche Art von Mustern, nämlich die dynamischen Mustern:

dynamic$(p : \tau)$

Solch ein Muster selektiert alle automorphen Werte, deren Wertkomponente dem Muster p entspricht und deren Typkomponente genereller als der Typausdruck τ ist.

Mit Hilfe dieses Konzeptes zur dynamischen Typisierung sind in CAML unter anderem die Funktionen *extern*, *intern* und *print* zum Ausdrucken eines automorphen Wertes auf dem Bildschirm in ML Syntax realisiert.

Napier88

Napier88 [Morrison et al. 94] ist wie **Tycoon** ein persistentes, polymorphes Programmiersystem. Es existiert ein vordefinierter Datentyp *any*, der der Typ der Vereinigung aller Werte ist. Auf Werten dieses Datentyps gibt es zwei Operationen:

any*(Ausdruck)*
project *Ausdruck* ***as*** *Bezeichner* ***onto*** *Projektionsliste*
  default *:Ausdruck*

Mittels der Operation **any** können Ausdrücke explizit in den Typ *any* eingefügt werden. Die Operation **project** erlaubt eine Projektion eines Wertes des Typs *any* auf einen Wert, der an den nach dem Schlüsselwort *as* angegebenen Bezeichner *Id* gebunden wird. In der Projektionsliste, jeweils bestehend aus einem Typbezeichner und einem Ausdruck, können mehrere Alternativen angegeben werden. Der Gültigkeitsbereich des Bezeichners *Id* ist auf den Ausdruck der jeweiligen Alternative beschränkt. Dieser Mechanismus verhindert Seiteneffekte auf dem projizierten Wert während der Evaluation dieses Ausdrucks und ermöglicht eine statische Typüberprüfung innerhalb dieses Ausdrucks. Zur Laufzeit erfolgt eine Typüberprüfung des Typs des zu projizierenden Ausdrucks mit den Typbezeichnern der verschiedenen Alternativen. Bei dem ersten erfolgreichen Typtest wird der zugehörige Ausdruck ausgeführt. Ist für keine Alternative der Projektionsliste der Typtest erfolgreich, so wird der Ausdruck nach dem Schlüsselwort *default* ausgeführt. Die *project* Operation entspricht damit den dynamischen Mustern in **CAML**. Darüber hinaus existiert noch eine weitere Funktion:

getTypeRep : ***proc****(any → TypeRep)*

Sie erhält als Parameter einen Wert des Typs *any* und liefert eine Typrepräsentation des Wertes zurück, durch deren Manipulation es möglich ist, die Typbeschreibungsinformation eines *any* zu untersuchen [Stemple et al. 93]. Mittels weiterer Hilfsfunktionen, die auf der Kenntnis der Objektformate von Werten beruhen (s. Abschnitt 3.3.2), kann man anhand dieser Strukturinformationen die Wertkomponente eines automorphen Wertes inspizieren [Kirby, Dearle 90]. Allerdings sind diese Hilfsfunktionen in der **Napier88** Standardbibliothek nicht enthalten [Kirby et al. 94]. Somit stehen dem Anwendungsprogrammierer keine Operationen zum Inspizieren von Werten des Typs *any* zur Verfügung. **Napier88** stellt statt dessen als Alternative die Nutzung linguistischer Laufzeitreflektion zur Verfügung, mit deren Hilfe sich u. a. generische, typgesteuerte Anwendungen realisieren lassen (s. Abschnitt 3.3.3).

Ansatz von Abadi, Cardelli, Pierce und Plotkin

In diesem Ansatz [Abadi et al. 89] werden ein neuer Basistyp *Dynamic* für automorphe Werte sowie die beiden Operationen

dynamic*(a :T)*
typecase *a* ***of*** *Projektionsliste* ***else*** *c*

in die statisch typisierte Sprache eingeführt. Die Operation **dynamic** konstruiert einen automorphen Wert aus einem Wert a mit Typ *T*. Die Operation **typecase** dient wie die *project* Operation in **Napier88** zum Inspizieren von dynamisch typisierten Werten. Der Unterschied zu der *project* Operation liegt aber in der Möglichkeit, vor einem Projektionszweig Mustervariablen erster Ordnung V_i definieren zu können, die zu jedem Subausdruck innerhalb des

Typausdrucks des zugehörigen Zweiges passen. Ihr Gültigkeitsbereich erstreckt sich auf den gesamten Zweig. Dadurch wird die Ausdrucksmächtigkeit von *typecase* erhöht. So läßt sich im Gegensatz zu *project* in einer Funktion, die aus Werten beliebigen Typs eine Zeichenkettenrepräsentation erzeugt, innerhalb eines einzigen Zweiges von *typecase* eine Vorschrift für Tupel mit zwei Komponenten definieren, deren beide Komponenten einen beliebigen Typ haben können:[2]

```
let rec toString(val :Dynamic) :String =
  typecase val of
    ...
    (X,Y) (v :Tuple x :X  y :Y end)
      “<” <> toString(dynamic (v.x :X))
      <> toString(dynamic (v.y :Y)) <> “>”
    ...
  end
```

Um die Funktion *toString* rekursiv innerhalb des Evaluationsausdrucks des Projektionszweiges aufrufen zu können, müssen aus den Komponenten des Tupels wieder automorphe Werte erzeugt werden. Dies ist möglich, da die Mustervariablen X und Y erst zur Laufzeit an die entsprechenden Typen gebunden werden. Weitere Konstrukte zum Inspizieren von automorphen Werten werden in diesem Ansatz nicht vorgestellt, da das theoretische Interesse dieses Ansatzes in der Interaktion zwischen den durch *typecase* erlaubten statisch und dynamisch typüberprüften Teilen der Sprache liegt.

Zusammenfassung

Allen hier vorgestellten Ansätzen ist gemein, daß sie ein Konzept zur Manipulation automorpher Werte anbieten. Die Entwicklung generischer, datenstrukturabhängiger Algorithmen ist jedoch nur mit dem Ansatz von Amber sowie mittels sprachlicher Laufzeitreflektion in Napier88 möglich. Weitere Unterschiede ergeben sich durch die Ausdrucksmächtigkeit des jeweiligen Projektionsoperators. So ist es in dem Ansatz von [Abadi et al. 89] durch die Einführung von Mustervariablen erster Klasse in eleganter Weise möglich, viele ähnliche Zweige von **typecase** in einem einzigen Zweig zusammenzufassen. Dies erlaubt es einerseits, die Anzahl der Zweige in **typecase** zu reduzieren, andererseits wird dessen Wiederverwendbarkeit erhöht.

5.3.2 Dynamische Typisierung und Polymorphismus

In Programmiersprachen, in denen das Konzept des parametrischen Polymorphismus realisiert ist, und in denen Funktionen Werte erste Klasse darstellen, ist es ohne Einschränkungen möglich, automorphe Werte von polymorphen Funktionen zu erzeugen. Problematisch ist dagegen jedoch die Frage, ob in automorphen Werten die Typvariablen mit Ausnahme von Mustervariablen geschlossen sein müssen, oder ob es erlaubt ist, automorphe Werte von solchen Werten zu erzeugen, bei deren Typ es sich um eine universell oder existentiell quantifizierte Typvariable aus dem umgebenden Kontext handelt [Abadi et al. 92]:

```
let f(A <:Ok a :A) = dynamic(a)
```

---

[2] In diesem Beispiel wird eine TL Syntax verwendet, die um die eben vorgestellten Typen und Konstrukte erweitert ist.

Die in diesem Beispiel definierte polymorphe Funktion *f* soll aus dem Parameter a, deren Typ die universell quantifizierte Typvariable *A* ist, einen automorphen Wert erzeugen und diesen zurückgeben. Da die polymorphe Typvariable *A* aber erst bei der Applikation von *f* instantiiert wird, ist statisch nur bekannt, daß a den Typ **Ok** besitzt. Diese Funktion würde also fälschlicherweise als Ergebnis immer einen automorphen Wert liefern, deren Typkomponente aus einer Repräsentation des Typs **Ok** besteht. Deshalb könnten in automorphen Werten nur geschlossene Typvariablen erlaubt werden. Dies würde jedoch zu einer erheblichen Einschränkung der Mächtigkeit automorpher Werte in polymorphen Programmiersprachen führen. Eine andere Lösungsmöglichkeit besteht darin, die polymorphe Funktion um einen zusätzlichen Parameter zu erweitern, der eine Laufzeitrepräsentation der jeweils übergebenen Typvariablen darstellt, die dann in die Typkomponente der automorphen Werte eingesetzt wird. So würde bei der Applikation von *f* auf den Wert 3 zur Laufzeit eine Typrepräsentation des Typs von 3, also von *Int*, erzeugt werden, die als Typkomponente in den automorphen Wert eingesetzt wird.

Ein weiteres Problem, das sich in Programmiersprachen mit explizitem Polymorphismus im Zusammenhang mit automorphen Werten ergibt, entsteht durch deren zusätzliche sprachliche Ausdrucksmächtigkeit. Gegenüber monomorphen erhöht sich in polymorphen Programmiersprachen die Anzahl der definierbaren Typen um ein Vielfaches. Das bedeutet, daß sich auch die Anzahl der Typen erhöht, auf die ein automorpher Wert projiziert werden kann. Die bisher vorgestellten Konzepte reichen jedoch nicht aus, um innerhalb von Projektionslisten in geeigneter Weise gegen polymorphe Typvariablen zu testen. In [Abadi et al. 92] wird ein Konzept vorgestellt, um diesem Problem zu begegnen. Es erweitert die in [Abadi et al. 89] vorgestellten Mustervariablen erster Ordnung zum Vergleichen von Typen um Mustervariablen höherer Ordnung. Diese decken den Kontext von Mustern ab, d.h. es handelt sich dabei um Muster, die über eine bestimmte Menge von Typvariablen abstrahieren.

```
let dynApply(f :Dynamic  x :Dynamic) =
  typecase f of
  (F,G) g :Fun(A <:Ok  a :F(A)) :G(A)
    typecase x of
    (W) a :F(W)  dynamic(g(:W a) :G(W))
    else raise exception “falsche Parameter” end
  else raise exception “falsche Parameter” end

dynApply(dynamic(fun(A <:Ok  x :A) x) dynamic(3)))
dynApply(dynamic(fun(A <:Ok  x : Tuple y :A  z :A end) tuple x.z x.y end)
    dynamic(tuple 1 2 end))
```

Im obigen, in *TL* ähnlicher Syntax beschriebenen Beispiel, stellen *F* und *G* Mustervariablen höherer Ordnung und *W* eine Mustervariable erster Ordnung dar. Sie ermöglichen die beiden Anwendungen von *dynApply*.

5.3.3 Dynamische Typisierung abstrakter Datentypen

Abstrakte Datentypen dienen dazu, Informationen zu verstecken und modulare Programmierung zu ermöglichen. Formal kann man abstrakte Datentypen als existentielle Typen betrachten [Mitchell, Plotkin 88], wodurch sie als Objekte erster Klasse behandelt werden können, die dem Programmierer in einer persistenten Programmierumgebung orthogonale Persistenz sichern [Ohori et al. 90]. Um diese Betachtungsweise zu formalisieren, sind zwei Typregeln

Pack und *Open* notwendig [Abadi et al. 92]. Mittels der Typregel *Pack* wird die tatsächliche Struktur eines abstrakten Datentyps in einem Paket versteckt. Nur innerhalb des speziellen Ausdrucks **open** ist diese sichtbar und kann mittels einer festen, explizit spezifizierten Menge von Schnittstellenfunktionen auf sie zugegriffen werden. Da die Typüberprüfung abstrakter Datentypen nicht strukturell erfolgt, gilt in folgendem Beispiel

```
Let ADT = Tuple T <:Ok .... end
let adt :ADT = Tuple Let T = Tuple end ... end
let adt1 = adt
```

adt.T ≮: *adt1.T*. Im Zusammenhang mit automorphen Werten ergibt sich jedoch die Fragestellung, ob bei abstrakten Datentypen die Typüberprüfung abstrakt oder strukturell erfolgen soll [Abadi et al. 92]. Für beide Entscheidungen gibt es gute Gründe:

1. Die abstrakte Typüberprüfung versteckt weiterhin die Typrepräsentation des abstrakten Datentyps und verhindert so eine zufällige Typgleichheit in den Fällen, in denen verschiedene abstrakte Datentypen die gleiche Repräsentation besitzen.

2. Die strukturelle Typüberprüfung erlaubt es, automorphe Werte eines abstrakten Datentyps als Werte verschiedener Versionen des gleichen abstrakten Datentyps zu behandeln. Dies ist z.B. in *Client–Server* Umgebungen unverzichtbar, auf denen automorphe Werte des gleichen abstrakten Datentyps auf verschiedenen Rechnern erzeugt und ausgetauscht werden.

In [Abadi et al. 92] wird eine Lösung dieses Problems vorgestellt, die beide Arten der Typüberprüfung unterstützt, indem eine erweiterte Typregel *Open* zum Entpacken eines existentiellen Typs eingeführt wird. Sie erlaubt es innerhalb von **typecase**, alternativ die Struktur des abstrakten Datentyps zu inspizieren oder die Typabstraktion weiterhin zu gewährleisten.

5.4 Realisierung dynamischer Typisierung in TL

In diesem Abschnitt wird der in TL realisierte Ansatz beschrieben, der die in Abschnitt 5.2.5 definierten Aufgaben dynamischer Typisierung unter Berücksichtigung der in Abschnitt 5.3.1 vorgestellten Ansätze und Probleme umsetzt. Dazu werden zunächst der ursprünglich in TL entwickelte Ansatz [Matthes, Schmidt 92] sowie dessen bei der Implementierung festgestellten Nachteile diskutiert. Die bei der Realisierung dieses Ansatzes gesammelten Erfahrungen trugen zu der in dieser Arbeit realisierten Lösung bei.

5.4.1 Ursprünglicher Ansatz

In diesem Ansatz werden zwei Sprachkonstrukte definiert, um die Aufgaben dynamischer Typisierung (s. Abschnitt 5.2.5) zu unterstützen:

- Die Signatur einer Typvariablen T kann mit dem Schlüsselwort **Dyn** versehen werden. Solch eine Typvariable kann wie eine Wertvariable zur Laufzeit inspiziert werden, um Informationen über die Struktur des tatsächlich an T gebundenen Typs zu erhalten.

- Typvariablen können mittels des Konstrukts **typecase** inspiziert werden. Innerhalb der Zweige des Konstrukts **typecase** wird die Typvariable auf einen statisch bekannten Typ eingeschränkt, so daß dort die Typüberprüfung statisch erfolgen kann.

Automorphe Werte lassen sich erzeugen, indem Wertbindungen mit ihren zugehörigen Typinformationen aggregiert werden [Matthes 93]:

```
Let Auto = Tuple Dyn T <:Ok  x :T end
let a1 = tuple Let Dyn T = Int let x = 3 end
let a2 = tuple Let Dyn T = String let x = "string" end

asString(a :Auto) :String =
  typecase a.T
  when Int then fmt.int(a.x)
  when String then a.x
  else "???"
  end

asString(a1)  asString(a2)
```

Im Unterschied zu den in Abschnitt 5.3.1 vorgestellten Ansätzen muß ein individueller Wert nicht explizit in einen infiniten Vereinigungstyp injiziert und aus diesem wieder auf seine Wertkomponente projiziert werden, um eine Typanalyse durchzuführen. Insbesondere ist es damit auch möglich, eine reflektive Typanalyse durchzuführen, wenn kein automorpher Wert, sondern nur eine Laufzeittyprepräsentation, vorliegt.

Dieser Ansatz bringt jedoch drei entscheidende Einschränkungen mit sich, die dazu führten, daß er nicht weiterverfolgt wurde:

1. Die Eigenschaften des Konstrukts **typecase** sind wie in den meisten anderen vorgestellten Ansätzen nicht ausreichend, um algorithmisch vollständigen Kode für generische, datenstrukturabhängige Anwendungen zu schreiben. Das Konstrukt **typecase** ist beschränkt auf Programmsituationen, in denen der Subtyptest einer Laufzeittyprepräsentation auf einer endlichen Menge statischer Typen durchgeführt wird.

2. Da es in TL eine Vielzahl von Typhierarchien gibt (s. Abschnitt 2.2.3), über die aber nicht mit Hilfe einer *Kind* Spezifikation quantifiziert werden kann, müßte man für jede Typhierarchie eine eigene Funktion zum Inspizieren von Typen bereitstellen:

   ```
   let inspectType(Dyn T <:Ok) = ...
   let inspectTypeOper1(Dyn T <:Oper(X <:Nok) Ok) = ...
   let inspectTypeOper2(Dyn T <:Oper(X <:Nok  Y <:Nok) Ok) = ...
   ```

   Stattdessen möchte man eine generische Funktion mit einer passenden Signatur schreiben, z.B.

   ```
   let inspectAnyType(Dyn T <:???) = ...
   ```

3. Die Kodegenerierung für dynamische Typvariablen führt einige versteckte Komplexität in den TL Sprachprozessor ein.

Aufgrund dieser Nachteile wurde im Rahmen dieser Arbeit ein neuer Ansatz entwickelt, der dem minimalistischen Ansatz von TL folgt, Namen, Konstanten und Funktionen vordefinierter Datentypen aus Modulen der Tycoon Standardbibliothek zu importieren, indem die Verwaltung von Laufzeittyprepräsentationen explizit gemacht und in TL Bibliotheken herausfaktorisiert wird.

5.4.2 Spracherweiterungen in TL

Der hier vorgestellte, zur Übersetzungszeit reflektive Ansatz (siehe Abschnitt 3.3.3) macht Laufzeittyprepräsentationen für algorithmisch vollständige Berechnungen unter Berücksichtigung der Tatsache zugänglich, daß TL eine Programmiersprache höherer Ordnung ist. Dazu wird TL um folgende Typen und Anweisungen erweitert:

1. Neue Basistypen *typeRep_T* und *dynamic_T* für Laufzeittyprepräsentationen bzw. automorphe Werte.
2. Konstruktoren *typeRep_new* und *dynamic_new* zum Erzeugen von Werten dieser Typen.
3. Eine zu *dynamic_new* inverse Operation *dynamic_be*, die zur Laufzeit aus einem automorphen Wert die Wertkomponente extrahiert.
4. Polymorphe Typvariablen können innerhalb einer Funktionssignatur mit dem Schlüsselwort **Dyn** versehen werden.

Laufzeittyprepräsentationen sind Werte erster Klasse, die einen statischen Typ repräsentieren. Die verwendete Notation bei den Typen und Anweisungen soll andeuten, daß es in entsprechenden Modulen weitere Operationen auf diesen Typen gibt. Sie lassen sich jedoch nicht innerhalb von Bibliotheksmodulen realisieren, da TL nicht über eine *Kind* Spezifikation verfügt (s. Abschnitt 5.4.1), und da die Anweisungen auf interne Datenstrukturen des Compilers zugreifen. Die Anweisungen lassen sich aber in folgender Pseudonotation spezifizieren:

*typeRep_new(T ::**ANYTYPE**) :typeRep_T*
*dynamic_new(T <:**Ok** v :T) :dynamic_T*
*dynamic_be(d :dynamic_T T <:**Ok**) :T*

Hierbei soll durch **ANYTYPE** ausgedrückt werden, daß es sich bei T um einen beliebigen Typ des Typsystems handeln kann. Durch die Syntax wird bei *typeRep_new* und *dynamic_be* die Angabe des Typparameters erzwungen. Bei *dynamic_new* dagegen wird die Typkomponente des automorphen Wertes aus dem übergebenen Wert inferiert und braucht deshalb nicht mit übergeben zu werden.

Die Funktion *dynamic_be* erhält den automorphen Wert d und die Typvariable T, die ein geschlossener Typ sein muß, d.h. nicht quantifiziert sein darf. Die Ausnahme *dynamic.error* wird zur Laufzeit ausgelöst, wenn der Typ der Wertkomponente von v kein Subtyp von T ist. Diese Anweisung hat nicht die Ausdrucksmächtigkeit wie **typecase** in dem Ansatz von [Abadi et al. 92], da die Einführung von Mustervariablen erster und höherer Ordnung die Komplexität des TL–Sprachprozessors erheblich erhöht hätte und da diese Lösung für die in Abschnitt 5.2 vorgestellten Anwendungen dynamischer Typisierung ausreichend ist.

Polymorphe Typvariablen, die mit dem Schlüsselwort **Dyn** gekennzeichnet sind, dürfen innerhalb von Laufzeittyprepräsentationen verwendet werden:

let *makeDyn(**Dyn** T <:**Ok** x :T) = dynamic_new(x)*
makeDyn(3) makeDyn(list.singleton(3))

Das Erzeugen von Laufzeittyprepräsentationen aus polymorphen Typvariablen, die nicht durch das Schlüsselwort **Dyn** gekennzeichnet sind, ist jedoch verboten und führt zur Übersetzungszeit zu einem Typfehler.

Eine weitere Einschränkung bei der Erzeugung von automorphen Werten bilden Unifikationsvariablen. Diese werden innerhalb des TL Typüberprüfers zum Instantiieren von Typbindungen in Typoperatorapplikationen eingeführt.

```
let x = list.new()        (* x :list.T(E’) *)
let y = list.cons(13 x)   (* x :list.T(Int) *)
let z = list.cons(‘A’ x)  (* Typfehler!! *)
```

Damit ist es wie in der Bindung von *x* möglich, Werte zu erzeugen, deren Typ eine Typoperatorapplikation mit einer nichtinstantiierten Unifikationsvariable ist. Später kann dann die Unifikationsvariable mit einer Typbindung instantiiert werden, wie dies in der Bindung von *y* implizit mit dem Typ *Int* erfolgt. Danach ist es nicht mehr erlaubt, die Unifikationsvariable mit einer anderen Typbindung zu instantiieren, so daß die Bindung von *z* zu einem Typfehler führt. Unifikationsvariablen innerhalb von Laufzeittyprepräsentationen führen zu Abhängigkeiten zwischen statischen Typen und Laufzeittyprepräsentationen, wie das folgende Beispiel illustriert:

```
let x = list.new()
let d = dynamic_new(x)                 (* d :list.T(E’) *)
let s = dynamic_be(d :list.T(Int))     (* d :list.T(Int) *)
let y = list.cons(‘A’ x)               (* Typfehler !! *)
```

Da Laufzeittyprepräsentationen jedoch unabhängig von einem statischen Kontext sein sollen, sind Unifikationsvariablen innerhalb von Laufzeittyprepräsentationen verboten.

5.4.3 Programmierschnittstellen

Um generische, datenstrukturabhängige Programme entwickeln zu können, werden aufbauend auf diesen Spracherweiterungen die beiden Module *typeRep* und *dynamic* in der Tycoon Standardbibliothek *tl_reflective* zur Verfügung gestellt. Diese implementieren Funktionalität zum Inspizieren und Konstruieren von Laufzeittyprepräsentationen und automorphen Werten sowie einige Hilfsfunktionen. Dazu bedienen sie sich internen Wissens über die Darstellung von Laufzeittyprepräsentationen durch den Compiler. Sie selbst sind aber nicht Bestandteil des Compilers (s. Abb. 5.2).

Schnittstelle ‘TypeRep’

Diese Schnittstelle (vgl. Anhang B.1) bietet Operationen auf Laufzeittyprepräsentationen an. Sie unterstützt die Inspektion von Typinformationen zur Laufzeit, eine der beiden Anforderungen an dynamische Typen. Dazu werden die folgenden Typen und Funktionen exportiert:

- Der Typ *T* ist der Typ einer Laufzeittyprepräsentation für einen beliebigen Typ der verschiedenen TL Typhierarchien (vgl. **ANYTYPE** auf Seite 5.4.2). Er entspricht dem Basistyp *typeRep_T* und sollte an dessen Stelle innerhalb von TL Programmen verwendet werden.

- Der Typ *Expansion* ist eine detaillierte Beschreibung einer Laufzeittyprepräsentation.

- Die Funktion *inspect* ermöglicht eine strukturierte Analyse einer Laufzeittyprepräsentation, indem diese um einen Schritt expandiert wird.

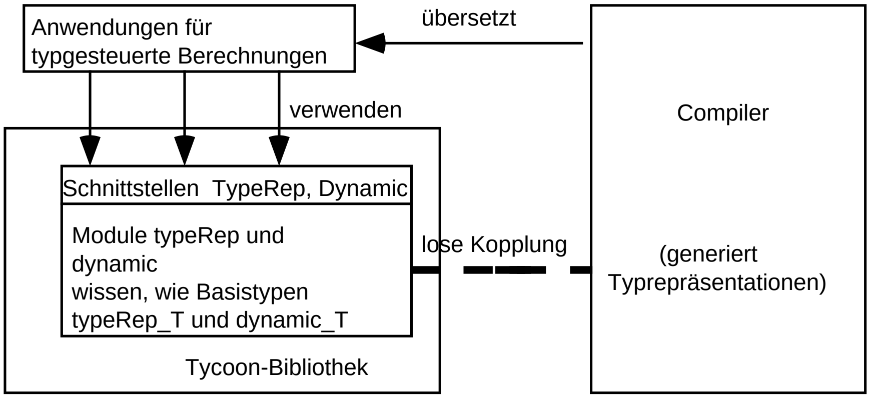

Anwendungen für typgesteuerte Berechnungen
übersetzt
verwenden
Compiler
Schnittstellen TypeRep, Dynamic
Module typeRep und dynamic wissen, wie Basistypen typeRep_T und dynamic_T
lose Kopplung
(generiert Typrepräsentationen)
Tycoon-Bibliothek

Abbildung 5.2: Kopplung des API für dynamische Typen an den Compiler

- ▷ Zu jedem TL Basistyp existiert eine Laufzeittyprepräsentation.
- ▷ Die *isSubType* Funktion macht die Subtyptestfunktion des Compilers für Laufzeittyprepräsentationen für Anwendungsprogrammierer zugänglich.
- ▷ Mit Hilfe einer Reihe von Konstruktoren ist es möglich, für jeden strukturierten TL Typ eine Laufzeittyprepräsentation aus Komponenten existierender Laufzeittyprepräsentationen zu erzeugen. Der Unterschied zu *typeRep_new* liegt darin, daß diese Konstruktoren keinen statischen Typ erwarten. Damit ist es möglich, Laufzeittyprepräsentationen von Typen zu erzeugen, von denen kein statischer Typ existiert.
- ▷ Einige zusätzliche Hilfsfunktionen erleichtern dem Programmierer die Arbeit mit Laufzeittyprepräsentationen. So expandiert z.B. die Funktion *exposed* eine Laufzeittyprepräsentation so lange, bis es sich nicht mehr um einen Typbezeichner handelt.

Schnittstelle 'Dynamic'

Die Lösung der anderen Anforderung an dynamische Typen — die Manipulation automorpher Werte — wird von der Schnittstelle *Dynamic* (s. Anhang B.2) unterstützt. Sie exportiert folgende Typen und Funktionen:

- ▷ Der Typ *T* ist der Typ automorpher Werte in TL. Er entspricht dem TL Basistyp *dynamic_T* und sollte an dessen Stelle in TL Programmen benutzt werden. Durch diese Schnittstelle wird sichergestellt, daß jeder Zugriff auf die Wertkomponente des automorphen Wertes zu dem Typ paßt, der durch seine Typkomponente repräsentiert wird.
- ▷ Der Typ *Expansion* liefert eine detaillierte Beschreibung eines automorphen Wertes.
- ▷ Die Funktion *inspect* führt eine Ein–Schritt–Expansion auf einem automorphen Wert durch. Dabei ist zu beachten, daß die Objektidentität der Wertkomponente eines automorphen Wertes erhalten bleibt. Im Gegensatz zu der Anweisung *dynamic_be* wird diese Funktion in Programmsituationen verwendet, in denen der Programmierer keine

Kenntnis des statischen Typs eines automorphen Wertes besitzt. Eine sukzessive Inspektion eines automorphen Wertes liefert dessen elementare Wertkomponenten. Diese Funktion stellt die Kernfunktionalität für datenstrukturabhängige Programme wie die generische Daten– und Funktionsvisualisierung bereit.

- ▹ Einen Sonderfall bei der Ein–Schritt–Expansion bilden Felder. Normalerweise müßte deren Inspektion ein Feld von automorphen Werten liefern. Da jedoch alle Feldelemente gleichen Typs sind, würde dies zu redundanter Speicherung von Typinformationen führen. Aus diesem Grund werden die beiden Funktionen *getIndexed* und *setIndexed* für den indizierten Zugriff auf Felder exportiert.
- ▹ Die Funktion *typeOf* liefert die Typkomponente eines automorphen Wertes.
- ▹ Eine Reihe von Konstruktoren ermöglicht die Konstruktion von strukturierten automorphen Werten aus Komponenten anderer automorpher Werte, die man mittels *inspect* erhält. Das hat den Vorteil, daß ein automorpher Wert nicht zuerst auf seine Wertkomponente projiziert werden muß, die der Konstruktion eines neuen Wertes dient, aus dem dann schließlich mittels *dynamic_new* der gewünschte automorphe Wert erzeugt wird.
- ▹ Die Funktionen *intern* und *extern* lesen eine linearisierte Repräsentation eines automorphen Wertes aus einer Datei in den Objektspeicher bzw. schreiben diese aus dem Objektspeicher in eine Datei (s. Abschnitt 5.2.2).
- ▹ Einige zusätzliche Hilfsfunktionen bieten weitere Funktionalität auf automorphen Werten an. Die Funktion *newDefault* z.B. erzeugt einen vordefinierten automorphen Wert zu dem als Paramter übergebenen Typ.

5.4.4 Erweiterungen des TL Compilers

Die Implementierung der in Abschnitt 5.4.2 beschriebenen TL Spracherweiterungen erfordert Erweiterungen des TL Compilers. Dabei vereinfachen zwei Eigenschaften der Tycoon Programmierumgebung den Zugriff auf Funktionen des Compilers und somit die Implementierung mittels Reflektion zur Übersetzungszeit: Da Tycoon eine persistente Programmierumgebung ist, können Programme im gleichen Objektspeicher übersetzt und ausgeführt werden. Außerdem ist der TL Compiler wie alle Anwendungsprogramme auch in TL geschrieben.

Der abstrakte Wertsyntaxbaum des TL Compilers wird um je eine Variante für die Anweisungen *typeRep_new*, *dynamic_new* und *dynamic_be* erweitert. Allen drei Anweisungen ist gemein, daß zur Übersetzungszeit während der Typüberprüfungsphase des Compilers der jeweilige Ausdruck analysiert und durch generierten Kode ersetzt wird, der zur Laufzeit ausgeführt wird. Die Kodegenerierung erfolgt dabei mittels der zum Compiler gehörenden Schnittstellen *TLDynamicImpl*, *TLDynamic* und *TLDynEnv* (s. Anhang A).

Generierung von Laufzeittyprepräsentationen

Um Laufzeitrepräsentationen von Typen zu generieren, muß zunächst eine Darstellungsform gefunden werden, die in der Lage ist, Typen adäquat zu repräsentieren. Es bietet sich an, die Repräsentation zu wählen, die schon während der Übersetzungszeit innerhalb des TL Compilers verwendet wird, aber bei der Kodegenerierung verloren geht. In der aktuellen Version des TL Compilers ist dies jedoch nicht angebracht. Dort basieren die Typrepräsentationen auf der sogenannten *de Bruijn Notation* [de Bruijn 72; Abadi et al. 90] zur Implementierung der Sichtbarkeitsregeln von Variablen. Dabei verweisen Variablen mittels Indices auf Signaturen innerhalb eines statischen Kontextes. Dies führt zu einer Abhängigkeit *aller* Typrepräsentationen von *einem* gemeinsamen Kontext. Laufzeittyprepräsentationen sollen jedoch unabhängig von einem bestimmten Kontext sein, damit sie in jedem Kontext, der über einen initialen Kontext hinausgeht, gültig sind. Aus diesem Grund wäre es notwendig, eine Typrepräsentation jeweils mit ihrem Kontext zu aggregieren. Diese Lösung hat jedoch zwei entscheidende Nachteile:

1. In einem gemeinsamen Kontext für alle Typrepräsentationen werden in der Regel nur ein geringer Teil der Signaturen dieses Kontextes von einer Typrepräsentation durch einen *deBruijn* Index referenziert. Es wäre aber zu aufwendig, bei der Übertragung einer Typrepräsentation in einen anderen Kontext den gesamten Kontext mitzuführen. Deshalb sollte der jeweils mitgeführte Kontext minimal sein, das heißt nur die Signaturen enthalten, die von einer Typkomponente aus referenziert werden. Aber auch die Eliminierung aller nicht von einer Typrepräsentation referenzierten Signaturen zur Laufzeit ist ein teurer Vorgang, da alle *deBruijn* Indices innerhalb der verbleibendenden Signaturen entsprechend angepaßt werden müssen.

2. Der Subtyptest zwischen zwei Typrepräsentationen, der beispielsweise bei der Anweisung *dynamic_be* durchgeführt wird, muß innerhalb eines gemeinsamen Kontextes erfolgen. Deshalb müssen die beiden Kontexte, in denen die Typrepräsentationen jeweils gültig sind, zunächst miteinander vereinigt werden. Dies ist ebenfalls mit hohen Laufzeitkosten verbunden.

Das führt zu der Wahl einer alternativen Laufzeitrepräsentation von Typen in TL, in der Typvariablen eine direkte Referenz auf ihr definierendes Auftreten enthalten. Diese Art der Darstellung entspricht einem zyklischen, gerichteten Graphen. Die Zyklen entstehen dann, wenn in einer Typrepräsentation rekursive Typbindungen definiert sind. Enthalten verschiedene Typrepräsentationen gemeinsame Typvariablen, so referenzieren beide Repräsentationen die gleiche Typvariable. Im Unterschied zur *deBruijn Notation* sind die Typrepräsentationen aber kontextunabhängig, da eine Typrepräsentation nur alle von einer Typkomponente aus transitiv erreichbaren Signaturen enthält, wenn sie aus einem Kontext in einen anderen übertragen wird. Damit ist die Forderung, daß eine Typrepräsentation nur die referenzierten Typbindungen enthalten soll, erfüllt.

Der Unterschied zwischen den beiden Darstellungsformen soll anhand eines Beispiels in Verbindung mit Abb. 5.3 illustriert werden:

> Angenommen, im initialen Kontext sei der Typ *Int* an der Position 54 und der Typ *String* an Position 52 definiert. Es werden folgende Typbindungen definiert:
>
> ***Let*** *Name = String@52*
> ***Let*** *Address =* ***Tuple*** *street :String@53 city :String@54 zip :Int@57* ***end***
> ***Let*** *Person =* ***Tuple*** *name :Name@2 address :Address@2* ***end***
> ***Let*** *Student =* ***Tuple*** *name :Name@3 address :Address@3 semester :Int@58* ***end***

Bei der Repräsentation durch direkte Referenzen auf das definierende Auftreten eines Typs tauchen bei benutzerdefinierten Typen wie z.B. *Person* Typbezeichner zweimal auf — einmal

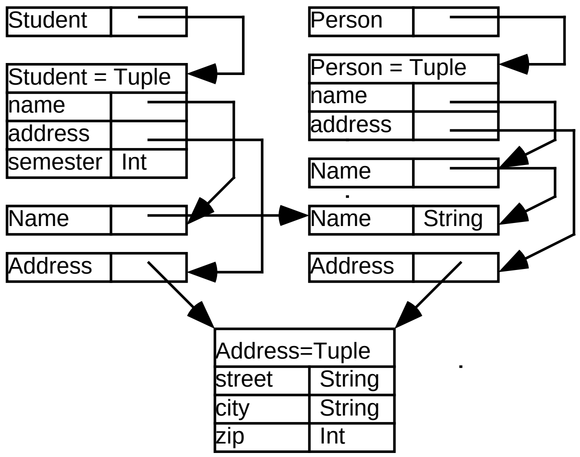


Abbildung 5.3: Typrepräsentation mit Referenzen

als Bezeichner, an den der Typ gebunden wird, und einmal als Identifikator bei der Definition der Typkomponente.

Die Verwendung unterschiedlicher Darstellungsformen für Typen innerhalb des TL Compilers und für Laufzeittyprepräsentationen hat jedoch den Nachteil, daß eine hochgradig rekursive Transformation zwischen den beiden Repräsentationen erforderlich ist, sowohl bei der Erzeugung von Laufzeittyprepräsentationen, als auch bei einem Subtyptest auf Laufzeittyprepräsentationen, da dazu der im TL Compiler implementierte Subtyptest verwendet wird, der auf Typen in der *deBruijn Notation* arbeitet. Das führt zu erheblichen Kosten zur Laufzeit. Um den Aufwand bei der Generierung von Laufzeittyprepräsentationen zu verringern, werden alle generierten Laufzeittyprepräsentationen für Typbindungen in einer globalen Tabelle gespeichert, die durch die Schnittstelle *TLDynEnv* verwaltet wird. Da in Tycoon jedes erzeugte Objekt eine eigene Identität erhält, kann anhand der Objektidentität des Typbezeichners einer Signatur festgestellt werden, ob für eine Typbindung bereits eine Laufzeittyprepräsentation erzeugt worden ist. Somit wird eine Laufzeittyprepräsentation für jede Typbindung nur einmal erzeugt. Andererseits gewährleistet dieses Verfahren auch, daß Laufzeittyprepräsentationen nur dann eine Typbindung teilen, wenn sie dieselbe Definition dieser Typbindung referenzieren.

Die Transformation von Laufzeittyprepräsentationen in statische Typrepräsentationen des Compilers erfolgt in zwei Schritten. Im ersten muß für jede in einer Laufzeittyprepräsentation referenzierte Typvariable überprüft werden, ob sie schon in dem Kontext definiert ist, in dem sie gültig sein soll. Ist dies nicht der Fall, so muß sie zunächst in den Kontext eingefügt werden. Um zufällige Namensäquivalenzen zwischen strukturell unterschiedlichen Typen zu vermeiden, werden die Namen von Typidentifikatoren um die Position, an der sie im Quelltext definiert worden sind, sowie um einen Zeitstempel, der bei der Generierung der Laufzeittyprepräsentation erzeugt worden ist, erweitert. Im zweiten Schritt erfolgt die eigentliche Transformation, bei der die *deBruijn* Indices für Typvariablen gesetzt werden müssen.

Trotz der Transformationskosten ist im Rahmen dieser Arbeit die zweite Darstellungsform gewählt worden, zumal in der zukünftigen Version des TL Compilers Typen ebenfalls durch direkte Referenzen auf ihr definierendes Auftreten repräsentiert werden [3], so daß die Transformationskosten entfallen. Der rekursive Typ T der Schnittstelle *TLDynamicImpl*, die die Transformation von Laufzeittyprepräsentionen in statische Typrepräsentationen realisiert, exportiert die Repräsentation von Laufzeittypen innerhalb des TL Compilers.

Ein Problem, das bei der Generierung von Laufzeittyprepräsentationen zu beachten ist, ist die Behandlung abstrakter Datentypen (s. Abschnitt 5.3.3). Da in TL für abstrakte Typvariablen Namensäquivalenzregeln gelten (s. Abschnitt 2.2.4), müssen diese für Laufzeittyprepräsentationen weiterhin erfüllt sein. Zusätzlich sollen aber abstrakte Typvariablen $x.T$ und $x1.T$ äquivalent sein, wenn *x* und *x1* innerhalb eines Objektspeichers das gleiche Objekt referenzieren, oder wenn sie in zwei verschiedenen Objektspeichern durch die gleiche Injektionsfunktion erzeugt worden sind. Damit ist die Generierung von Laufzeittyprepräsentationen also auch abhängig von den Werten *x* und *x1*.

Eine Lösung dieses Problems, die es vermeidet, die tatsächliche Struktur der abstrakten Typvariablen offenlegen zu müssen, ist die Ersetzung der Pfade, d.h. der Wertkomponente, abstrakter Typvariablen durch (weltweit eindeutige) universelle Bezeichner (*UID*). Solch eine *UID* besteht aus einer Plattformidentifikation, einer Maschinenidentifikation sowie einem Zeitstempel und wird durch eine Funktion zur Generierung von Symbolen der Tycoon Bibliothek erzeugt. Wird diese Funktion mit einem Objekt aufgerufen, von dem noch keine *UID* erzeugt worden ist, so wird ein neues Symbol generiert. Ansonsten wird die bereits erzeugte *UID* zurückgegeben. Ursprünglich ist dieser Mechanismus entwickelt worden, um beim Austausch von Daten innerhalb eines Netzwerkes die zu übertragende Datenmenge zu reduzieren. Werden zwei Objekte in verschiedenen Objektspeichern als ubiquitär [Mathiske et al. 95b] registriert, so können *UID* Werte anstelle der Objekte ohne Typinformationen innerhalb des Netzwerkes verschickt werden. Über die *UID* kann überprüft werden, ob zwei solche Werte in verschiedenen Objektspeichern "semantisch äquivalent" sind.

Ausgehend von diesen Überlegungen, erfolgt die Generierung einer Laufzeittyprepräsentation des Typs T in vier Schritten:

1. T wird innerhalb des statischen Kontextes S auf Wohlgeformtheit überprüft.
2. Aus T wird eine kontextunabhängige Repräsentation $\mathrm{t}_{S,T}$ generiert.
3. Es wird ein Funktionsaufruf generiert, der die Pfade aller abstrakten Typvariablen zusammen mit $\mathrm{t}_{S,T}$ als Parameter erhält. Diese Funktion fügt zur Laufzeit die *UID* Werte für die Pfade in die entsprechenden Stellen in $\mathrm{t}_{S,T}$ ein.
4. Der generierte Funktionsaufruf wird in den Syntaxbaum eingefügt.

Generierung automorpher Werte

Bei der Generierung automorpher Werte wird zunächst der Typ aus dem Wert inferiert. Aus diesem wird wie im vorherigen Abschnitt beschrieben eine Laufzeittyprepräsentation generiert, die mit dem Wert zu einem Tupel aggregiert und in den Syntaxbaum eingesetzt wird. Beispielsweise hätte der Programmierer statt des Aufrufs *dynamic_new(2)* auch schreiben können

---

[3] Eine Implementierung des Typüberprüfers mit direkten Objektreferenzen wird zur Zeit im Rahmen einer anderen Diplomarbeit realisiert und in [Bremer 96] beschrieben werden

```
tuple
  let value = 2
  let type = typeRep_new(:Int)
end
```

mit dem Unterschied, daß der Compiler zusichert, daß *type* auch den Typ von *value* repräsentiert, da dieser aus dem Wert inferiert und nicht explizit vom Programmierer angegeben wird.

Generierung der Projektion

Da auch Funktionen des Compilers als Literale in den Syntaxbaum gehängt werden können, wird der Subtyptest, der zur Laufzeit von *dynamic_be* ausgeführt wird, realisiert, indem ein Aufruf der Funktion generiert wird, die den Subtyptest im TL Compiler durchführt. Ein Nachteil dieser Lösung entsteht daraus, daß im Tycoon System übersetzte Module sowohl linearisiert im Dateisystem gespeichert sind als auch nach dem Binden im Objektspeicher. Obwohl die Funktion, die den Subtyptest durchführt, Bestandteil des Compilers ist und somit bereits im Objektspeicher vorhanden ist, wird sie mitsamt des größten Teils des Compilers und allen im aktuellen Kontext definierten Bindungen in das Dateisystem hinausgeschrieben, falls der Aufruf von *dynamic_be* innerhalb eines Moduls erfolgt. Dies führt zu zusätzlichem Zeitaufwand bei den Funktionen *intern* und *extern* sowie zu erheblich größeren Dateien. Eine Lösung diese Problems bestünde darin, daß man ähnlich wie beim Verschicken von Objekten zwischen zwei Objektspeichern solche Funktionen, die wie die des Compilers in jedem Objektspeicher vorhanden sind, registriert. Bei *extern* wird das Objekt durch seine UID ersetzt und umgekehrt bei *intern* die UID durch den Objektidentifikator im Objektspeicher.

Da beim Transformieren einer Laufzeittyprepräsentation in eine Typrepräsentation des Compilers die Namen von Typidentifikatoren verändert werden, muß auch vom Typparameter der Anweisung *dynamic_be* zunächst eine Laufzeittyprepräsentation erzeugt werden, die dann wieder zurückverwandelt wird in eine Typrepräsentation des Compilers. Als initialer Kontext, in dem die beiden Typrepräsentationen gültig sein sollen, wird dabei der Kontext gewählt, der auch bei der Initialisierung des Tycoon Systems erzeugt wird. Er enthält eine im TL Compiler minimal benötigte Anzahl von Bindungen wie beispielsweise die Definitionen der Basistypen.

Die Umsetzung von *dynamic_be* umfaßt im Typüberprüfer folgende Schritte:

1. Der Typparameter T wird innerhalb des statischen Kontextes S auf Wohlgeformtheit überprüft.
2. Aus T wird eine Laufzeittyprepräsentation $t_{S,T}$ erzeugt.
3. Falls es sich bei dem Typparameter um eine universell quantifizierte Typvariable handelt, die nicht durch das Schlüsselwort **Dyn** gekennzeichnet ist, oder um eine uninstantiierte Unifikationsvariable, wird eine Typfehlermeldung erzeugt.
4. Es wird ein Funktionsaufruf generiert, der bei einem erfolgreichen Subtyptest zwischen $t_{S,T}$ und der Typkomponente des automorphen Wertes in dem gemeinsamen statischen Kontext S' dessen Wertkomponente zurückliefert.
5. Der generierte Funktionsaufruf wird in den Syntaxbaum eingefügt.

Um mit dem Schlüsselwort **Dyn** gekennzeichnete polymorphe Typvariablen innerhalb von Laufzeittyprepräsentationen verwenden zu können, muß bei jedem Aufruf einer polymorphen Funktion für den aktuellen Typparameter eine Laufzeittyprepräsentation erzeugt und als Parameter mit übergeben werden (s. Abschnitt 5.3.2). Dazu wird während der Übersetzungsphase für den Anwendungsprogrammierer nicht sichtbar die Funktionssignatur für jede mit **Dyn** gekennzeichnete polymorphe Typvariable um einen zusätzlichen Parameter für die Laufzeittyprepräsentation des aktuellen Typparameters erweitert.

```
let f(Dyn A <:Ok) = typeRep_new(A)
    ==>
let f(Dyn A <:Ok  a_Rep :typeRep_T) = typeRep_new(A)

f(:Int)  ==>  f(:Int typeRep_new(:Int))
```

Wird wie in obigem Beispiel aus der polymorphen Typvariable *A* im Rumpf von *f* eine Laufzeittyprepräsentation erzeugt, so wird diese nicht aus *A* generiert, sondern es wird die bei der Funktionsapplikation erzeugte Laufzeittyprepräsentation *a_Rep* eingesetzt. Jeder andere Zugriff auf die Typvariable *A* erfolgt so, als wäre sie nicht mit **Dyn** gekennzeichnet. Die Funktion *f* wird jedoch weiterhin mit der ursprünglich definierten Anzahl von Parametern aufgerufen. Der Compiler generiert dann aus allen mit **Dyn** gekennzeichneten polymorphen Typvariablen Laufzeittyprepräsentationen.

Eine für den Programmierer transparentere Lösung wäre die Einführung eines weiteren Typs und Konstruktors gewesen:

- *typeRep_Closed(X <:***Ok***) <:typeRep_T* ist der Typ von Laufzeittyprepräsentationen, die einen geschlossenen Typ darstellen.

- *typeRep_newClosed(X <:***Ok***) :typeRep_Closed(X)* erzeugt aus einem geschlossenen Typ eine Laufzeittyprepräsentation.

Soll eine Laufzeittyprepräsentation von einer polymorphen Typvariablen erzeugt werden, so ist dieser Funktion als weiterer Parameter explizit eine Laufzeittypprepäsentation des aktuellen Typparameters, die den Typ *typeRep_Closed* hat, zu übergeben. Da *typeRep_Closed* ein Typoperator ist, ist sichergestellt, daß die Laufzeittyprepräsentation die polymorphe Typvariable repräsentiert. Dieser Ansatz hat den Vorteil, daß er keine versteckten komplexen Erweiterungen des TL Sprachprozessors erfordert.

Allerdings gibt es auch zwei entscheidende Nachteile, die zu der in TL realisierten Lösung geführt haben: Zum einen ist die Erzeugung von Laufzeittyprepräsentationen, in denen eine polymorphe Typvariable innerhalb eines strukturierten Typs verwendet wird, komplizierter, wie folgender Vergleich illustriert:

```
let f(A <:Ok  a_Rep :typeRep_Closed(A)) =
  begin
    Let T = Tuple a :A  b :Int end
    let aSig =
      tuple_case valueCase of typeRep.Signature with
        let ide = tuple
          let name = “a”
          let pos = source.noWhere
        end
        let type = a_Rep
      end
    let bTuple = typeRep_new(:Tuple b :Int end)
    let sigs = list.cons(aSig typeRep.signatures(bTuple))
    let tRep = typeRep.newTuple(sigs
                    typeRep.newEmptyCases())
    ...
  end
```

```
let f(Dyn A <:Ok) =
  begin
    Let T = Tuple
      a :A
      b :Int
    end
    let tRep = typeRep_new(:T)
    ...
  end
```

Zum anderen erlaubt diese Lösung keine uniforme Verwendung polymorpher Funktionen. Anders als beim Schlüsselwort **Dyn** ist die Anzahl der beim Funktionsaufruf zu übergebenden Parameter abhängig davon, ob von einer polymorphen Typvariablen im Funktionsrumpf eine Laufzeittyprepräsentation erzeugt werden soll oder nicht. Um die in Abschnitt 6.2.2 beschriebene Lösung zur Generierung von Laufzeittypinformationen der Typen globaler Bindungen zu verwenden, müßten also nicht nur die Funktionssignaturen um die Parameter für die Laufzeittyprepräsentationen polymorpher Typvariablen erweitert werden, sondern auch alle Funktionsaufrufe polymorpher Funktionen. Da der in 6.2.2 realisierte Ansatz übersetzungszeitreflektiv ist, müste der Compiler bei jedem Funktionsaufruf testen, ob die Funktion polymorphe Typvariablen erwartet und ob dazu die Laufzeittyprepräsentation mit generiert und als Parameter übergeben werden muß. Damit führt dieser Ansatz, anders als bei der Verwendung des Schlüsselwortes **Dyn** zu Geschwindigkeitsverlusten beim Übersetzen und zu einer höheren Komplexität des Sprachprozessors.

6. Linguistische Reflektion zur Unterstützung der Visualisierung

Linguistische Reflektion, wie sie in Abschnitt 3.3.3 vorgestellt worden ist, stellt einen uniformen Mechanismus zur Softwareentwicklung zur Verfügung, der unter anderem die Anpassung an einige der ständig auftretenden Änderungen innerhalb datenintensiver Anwendungen durch reflektiven Zugriff auf die Typen des sich ändernden Systems ermöglicht [Atkinson, Morrison 95]. Um systematische Informationen über die in einer Berechnung anzutreffenden Strukturen zur Verfügung zu haben, und um die Typsicherheit aller neu generierten Programmfragmente vor ihrer Ausführung gewährleisten zu können, muß linguistische Reflektion typsicher implementiert sein [Stemple et al. 93]. Dies ist mit Hilfe dynamischer Typsisierung möglich.

In diesem Kapitel soll gezeigt werden, wie linguistische Reflektion die Daten– und Funktionsvisualisierung unterstützen kann. Dazu wird im ersten Abschnitt ein übersetzungszeit–reflektiver Ansatz vorgestellt, der einen typisierten Zugriff auf die globalen Bindungen einer Funktion ermöglicht. Im zweiten Abschnitt wird diskutiert, wie dieser Ansatz modifiziert werden kann, um die Kontrollflüsse persistenter *Threads* typisiert verfolgen zu können.

6.1 Laufzeittyprepräsentationen für die globalen Bindungen einer Funktion

Um die Funktionalität einer Funktion zu verstehen, muß der Anwender Zugriff auf ihre Signatur, ihren Rumpf und alle globalen Bindungen haben. Deshalb muß die Daten– und Funktionsvisualisierung auch auf die globalen Bindungen einer Funktion als automorphe Werte zugreifen, um sie generisch visualisieren zu können. Bei der Generierung automorpher Werte werden jedoch für Funktionen nur Laufzeittyprepräsentationen ihrer Signatur erzeugt, nicht aber von ihren globalen Bindungen, so daß es notwendig ist, die globalen Bindungen auf andere Art mit ihren Laufzeittyprepräsentationen zu assoziieren.

6.1.1 Funktionsabschluß

Anhand der folgenden Funktion *addToGlobal* werden die Problematik der globalen Bindungen und die Laufzeitrepräsentation von Funktionen in Tycoon erläutert:

```
let var j = 3
let addToGlobal(i :Int) =
  begin
    let result = i + j
    let printResult() :Ok = print.int(result)
```

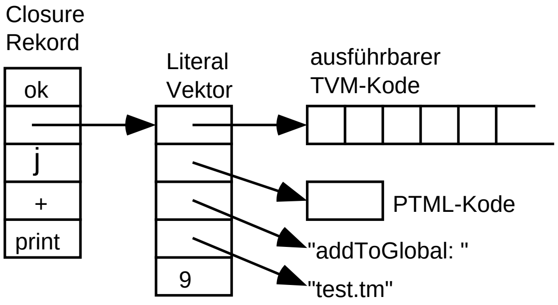


Abbildung 6.1: Abschluß einer Funktion

```
    tuple "addToGlobal: " printResult end
  end
```

In *addToGlobal* ist *result* eine *lokale* Bindung, in *printResult* aber eine *freie* bzw. *globale* Bindung. Da die Funktion *printResult* als Komponente des von *addToGlobal* zurückgegebenen Tupels gespeichert ist und später ausgeführt werden kann, überlebt *result* die Lebensdauer von *addToGlobal*. Um globale Bindungen auch außerhalb des Kontextes referenzieren zu können, in dem sie definiert worden sind — in diesem Beispiel *result* in *printResult* nach der Ausführung von *addToGlobal* — werden Funktionen zur Laufzeit durch einen *Closure Record* repräsentiert, in dem alle globalen Bindungen einer Funktion gespeichert sind [Gawecki, Matthes 94].

Abbildung 6.1 zeigt die Laufzeitrepräsentation der Funktion *addToGlobal*, die in dem Modul *test.tm* in Zeile 9 definiert sei. Sie besteht aus dem *Closure Record* und einem Literalvektor. Der erste Eintrag des *Closure Records* hat bisher keine Funktion, so daß er leer ist, und der zweite enthält eine Referenz auf den Literalvektor. In den anderen Einträgen des *Closure Records* sind die globalen Bindungen einer Funktion verzeichnet. Im Literalvektor sind alle diejenigen Objekte gespeichert, die nicht direkt innerhalb der Byte Sequenz des ausführbaren Kodes repräsentiert werden können, sondern wie z.B. Zeichenketten zur Laufzeit vom *Heap* geholt werden müssen. In weiteren Feldern des Literalvektors existieren Referenzen auf das Modul sowie die Zeile, in dem die Funktion definiert ist, eine persistente Repräsentation des TML Kodes (*PTML*) [Kiradjiev 94] und den ausführbaren TVM Kode.

Die Eliminierung aller lokalen Bindungen aus dem Funktionsabschluß erfolgt in einer eigenen Übersetzungsphase nach der TML und vor der TVM Kodegenerierung in der sogenannten Closure Konvertierung. Zu diesem Zeitpunkt existieren jedoch keine Typinformationen mehr, da der TML Kode eine untypisierte Zwischenrepräsentation ist. Um die globalen Bindungen mit Laufzeittyprepräsentationen versehen zu können, müßten in der Phase der Typanalyse die globalen Bindungen ermittelt, Laufzeittyprepräsentationen von ihren Typen erzeugt und in der TML Zwischenrepräsentation mitgeführt werden sowie in der TVM Kodegenerierung in den *Closure Record* eingetragen werden. Dazu müßten neben der Typüberprüfungsphase auch die TML und TVM Kodegenerierung verändert werden. Außerdem muß bei Laufzeittyprepräsentationen zur Laufzeit die Funktion ausgeführt werden, die die Pfade der abstrakten Datentypen durch ihre *UID* ersetzt. Der Programmkode für Funktionen müßte also zusätzlich so verändert werden, daß für alle globalen Bindungen diese Funktionsaufrufe erfolgen.

Um diesen Aufwand zu vermeiden, wird im Rahmen dieser Arbeit ein übersetzungszeitreflektiver Ansatz vorgestellt, der nur Veränderungen in der Typüberprüfungsphase notwendig macht. Die Idee dieses Ansatzes besteht darin, daß der Typüberprüfer explizite Funktionsaufrufe in den zu generierenden Kode einfügt, die beim Binden der Funktion ausgeführt werden und Laufzeittyprepräsentationen mit allen globalen Bindungen assoziiert sowie in den Funktionsabschluß einträgt. Über eine Schnittstelle, die Kenntnis von der internen Repräsentation einer Funktion hat, kann der Anwendungsprogrammierer später auf die globalen Bindungen als automorphe Werte zugreifen.

6.1.2 Erweiterter Funktionsabschluß

Um die globalen Bindungen einer Funktion mit ihren Laufzeittyprepräsentationen zu assoziieren, ist eine Erweiterung des *Closure Records* notwendig. Diese Verknüpfung kann jedoch nicht direkt im *Closure Record* erfolgen, da

- der ausführbare TVM Kode davon ausgeht, daß im *Closure Rekord* nur die Werte der globalen Bindungen stehen, nicht aber deren Laufzeittyprepräsentationen,
- zur Typüberprüfungszeit noch nicht bekannt ist, an welcher Position im *Closure Record* welche globale Bindung referenziert wird und
- bei variablen globalen Bindungen im *Closure Record* nur die Basisadresse einer Zelle steht und die relative Adresse nur innerhalb des TVM Kodes bekannt ist.

Deshalb werden alle globalen Bindungen zusätzlich mit der Laufzeitrepräsentation ihres Typs in einer eigenen Datenstruktur abgespeichert und von dem ersten, zur Zeit leeren Eintrag des *Closure Rekords* aus referenziert. Diese Datenstruktur und die Funktionen, über die diese Informationen in den Funktionsabschluß eingetragen werden, werden für den Typüberprüfer durch das Modul *tlClosure* exportiert. Die Datenstruktur wird repräsentiert durch den folgenden rekursiven Listentyp:

```
Let Rec TypeList <:Ok = Tuple
  case rValue with
    name :String
    value :dynamic_T
    next :TypeList
  case lValue with
    name :String
    value() :dynamic_T
    setValue(:dynamic_T) :Ok
    getId() :Location
    next :TypeList
  case nil with
end
```

Dabei wird unterschieden zwischen konstanten (*rValue*) und variablen (*lValue*) globalen Bindungen. Beide haben einen Namen, an den sie gebunden sind, und eine Referenz auf die nächste globale Bindung. Entsprechend den Bindungen automorpher Werte erfolgt der Zugriff auf den Wert konstanter globaler Bindungen direkt über die Komponente *value*, und auf

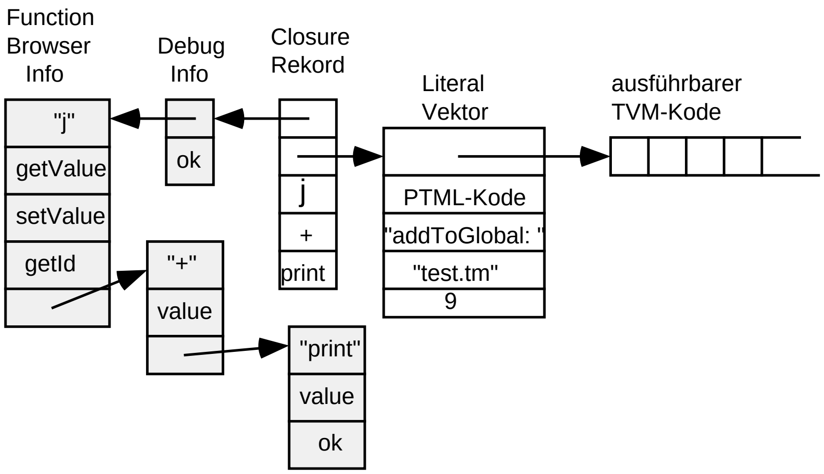


Abbildung 6.2: Funktionsabschluß mit Typinformationen für globale Bindungen

den variabler globaler Bindungen über die Funktionen *value* und *setValue* sowie auf dessen Zelle über *getId*.

Da es vorstellbar ist, daß für andere Dienste wie beispielsweise zur Fehlersuche noch weitere Informationen im Funktionsabschluß gespeichert werden müssen und dafür nicht jeweils ein eigener Eintrag im *Closure Rekord* erzeugt werden soll, wird diese Datenstruktur in eine zweite Liste des Typs

```
Let Rec DebugInfo <:Ok = Tuple
  case functionBrowserInfo with
    info :TypeList
    next :DebugInfo
  case nil with
end
```

eingetragen, die vom ersten Eintrag des *Closure Records* referenziert wird. In dieser Liste können in Abhängigkeit von der Variante des Tupels verschiedene Informationen gespeichert werden. Zur Zeit existiert nur die Variante *functionBrowserInfo* für die Liste der globalen Bindungen mit ihren Laufzeittyprepräsentationen. Abbildung 6.2 zeigt die um diese Datenstrukturen erweiterte Repräsentation der Funktion *addToGlobal*.

Um die globalen Bindungen zusammen mit ihren Laufzeittyprepräsentationen typsicher in die Liste vom Typ *TypeList* eintragen zu können, werden die beiden Funktionen

```
valueInClosure(name :String  value :dynamic_T  closure :Closure) :Closure
locationInClosure(name :String  value() :dynamic_T  setValue(:dynamic_T) :Ok
    getId() :Location  closure :Closure) :Closure
```

für konstante bzw. variable globale Bindungen exportiert, die die um den entsprechenden Eintrag erweiterte Funktion zurückgeben. Dabei müssen Funktionen als Werte des abstrakten Datentyps *Closure* als Parameter übergeben werden, wobei sichergestellt sein muß, daß der erste Eintrag des *Closure Records* für Werte des Datentyps *DebugInfo* reserviert ist.

Das Modul *tlClosure* wird nur innerhalb des *Compilers* benutzt. Der Anwendungsprogrammierer dagegen kann nur über die Schnittstelle *Closure* (s. Anhang B.3) typsicher auf die mit den Laufzeittyprepräsentationen versehenen globalen Bindungen zugreifen. Sie exportiert

- ▹ einen abstrakten Datentyp *T*, der intern als *TLClosure.Closure* definiert ist,
- ▹ eine Funktion *coerce*, die aus einem Funktionswert einen Wert vom Typ *T* erzeugt,
- ▹ zu *valueInClosure* und *locationInClosure* äquivalente Funktionen und
- ▹ Funktionen, mit denen man auf die Einträge der Listen vom Typ *DebugInfo* und *TypeList* zugreifen kann.

Eigentlich sollten die Typen und Funktionen des Moduls *tlClosure* auch in *Closure* definiert sein. Da sie jedoch von Funktionen des Typüberprüfers verwendet werden, müssen sie auch im Kontext der Bibliothek, die die Module des Typüberprüfers enthält, sichtbar sein. Die Funktion *coerce* aus *Closure* wiederum läßt sich nur mit Hilfe der Module *typeRep* und *dynamic* implementieren, in deren Kontext die Module des Typüberprüfers sichtbar sein müssen. Da Zyklen in der Modulverwaltung über Bibliotheksgrenzen hinweg jedoch nicht erlaubt sind, erfolgt eine Trennung der entsprechenden Funktionalitäten.

6.1.3 Generierter Programmkode

Um die globalen Bindungen einer Funktion zusammen mit den Laufzeittyprepräsentationen in den Abschluß einer Funktion einzutragen, ohne die TML oder TVM Kodegenerierung verändern zu müssen, werden während der Typüberprüfungsphase für alle globalen Bindungen Funktionsaufrufe der Funktionen *valueInClosure* bzw. *locationInClosure* des Moduls *tlClosure* generiert, die zur Laufzeit ausgeführt werden. Da jedoch die Semantik eines Programmes dadurch nicht verändert werden darf, müssen alle für eine Funktion *f* generierten Funktionsaufrufe ineinander verschachtelt werden. Die Funktion *f* selbst wird im innersten Funktionsaufruf definiert und als Parameter übergeben. Jeder dieser Funktionsaufrufe fügt zur Laufzeit eine globale Bindung mit ihrer Laufzeittyprepräsentation in *f* ein und gibt *f* als Ergebnis an den ihn umgebenden Funktionsaufruf als Parameter zurück. So wird *f* durch die ineinander verschachtelten Funktionsaufrufe nach oben durchgereicht, ihr Funktionsabschluß dabei um die globalen Bindungen mit ihren Laufzeittyprepräsentationen erweitert und an den im ursprünglichen Programmkode definierten Bezeichner gebunden. Beispielsweise wird für die oben definierte Funktion *addToGlobal* während der Typüberprüfungsphase folgender Programmkode generiert:

```
let var j = 3
let addToGlobal =
  valueInClosure("print" dynamic_new(print)
    locationInClosure("j"  let getValue = ...   let setValue = ...   let getId =...
      valueInClosure("+" dyanamic_new(+)
        fun(i :Int)
          begin
            let result = i + j
            let printResult =
              valueInClosure("print" dynamic_new(print)
                valueInClosure("result" dynamic_new(result)  fun() :Ok print.int(result)))
    tuple "addToGlobal: " printResult end
  end)))
```

Diese Funktionsaufrufe werden nur einmal beim Binden an *addToGlobal* ausgeführt, so daß nur die ursprüngliche Funktionsdefinition gebunden wird.

Da eine Funktion *f* lexikalisch innerhalb einer anderen Funktion *g* definiert werden kann, ist es möglich, daß in *f* globale Bindungen referenziert werden, deren Typ polymorph ist:

```
let g(T <:Ok  t :T) = begin let f() = t  f end
```

Von polymorphen Typvariablen dürfen Laufzeittyprepräsentationen aber nur erzeugt werden, wenn sie durch das Schlüsselwort **Dyn** gekennzeichnet sind, so daß sie bei der Kodegenerierung um **Dyn** erweitert werden müssen. Da es jedoch zu aufwendig ist, festzustellen, von welchen polymorphen Typvariablen Laufzeittyprepräsentationen erzeugt werden müssen, und da diese Überprüfung die Komplexität des Typüberprüfers weiter erhöhen würde, werden alle polymorphen Typvariablen mit **Dyn** gekennzeichnet. Das hat allerdings auch den Nachteil, daß sich das Laufzeitverhalten verschlechtert, weil bei jedem Funktionsaufruf von allen polymorphen Typvariablen Laufzeittyprepräsentationen erzeugt werden müssen.

Für rekursive Wertbindungen ist der für *addToGlobal* vorgestellte Ansatz zu modifizieren, da deren Wertkomponenten keine Funktionsapplikationen sein dürfen. Deshalb werden für die funktionalen Komponenten der parallelen rekursiven Bindungen die Funktionsaufrufe, die die globalen Bindungen mitsamt den Laufzeittyprepräsentationen in den Funktionsabschluß eintragen, erst nach der Definition aller rekursiven Bindungen generiert. Da das Ergebnis paralleler Bindungen immer der Wert der zuletzt definierten Bindung ist, wird abschließend noch eine Referenz auf diese Bindung erzeugt:

aus

```
let rec j <:Ok = 3
and a() <:Ok =  tuple b() j end
and b() <:Ok = j
```

wird generiert

```
let rec j <:Ok = 3
and a() <:Ok = tuple b() j end
and b() <:Ok = j
valueInClosure("j" dynamic_new(j) valueInClosure("b" dynamic_new(b) a))
valueInClosure("j" dynamic_new(j) b)
b
```

Durch diesen Ansatz wird die Anzahl der definierten Bindungen erweitert, so daß sich die Semantik des Programmkodes ändert, wenn die rekursiven Bindungen als Bindungen von Rekords, Tupeln, Funktionsapplikationen oder Feldern definiert werden, da in diesen Fällen die Anzahl der definierten Bindungen der Anzahl der parallen rekursiven Bindungen entsprechen muß. Diese Programmsituationen treten jedoch nur äußerst selten auf und können in der Regel vermieden werden, indem die rekursiven Bindungen außerhalb dieser Konstrukte definiert und innerhalb referenziert werden. Da es zusätzlich die Komplexität des Typüberprüfers entscheidend erhöhen sowie zu erheblichen Übersetzungs- und Laufzeiteinbußen führen würde, wenn der Ansatz darauf erweitert wird, erzeugt der Typüberprüfer in diesen Fällen eine Fehlermeldung.

6.1.4 Erweiterungen des Typüberprüfers

Mit Hilfe der Funktionen des Moduls *tlClosure* ist es möglich, durch Änderungen im Typüberprüfer den in den letzten beiden Abschnitten beschriebenen Ansatz zu implementieren. Dazu müssen im Typüberprüfer

- ▹ die globalen Bindungen einer Funktion gesammelt,
- ▹ von ihren Typen Laufzeittyprepräsentationen erzeugt und
- ▹ die Funktionsaufrufe generiert werden.

Im Gegensatz zur aktuellen Strategie ist dies jedoch nicht in einem Durchlauf des abstrakten Syntaxbaumes möglich. Stattdessen müssen alle funktionalen Komponenten des Syntaxbaumes zweimal durchlaufen werden — einmal in der *Erweiterungsphase* und ein zweites Mal in der *Typüberprüfungsphase.*

In der Erweiterungsphase werden die Funktionen typüberprüft, die polymorphen Typvariablen mit dem Schlüsselwort **Dyn** versehen und festgestellt, welche Bindungen global in einer Funktion sind. Für jede als global identifizierte Bindung werden eine Laufzeittyprepräsentation ihres Typs sowie die Werte, die den Funktionen *valueInClosure* bzw. *locationInClosure* als Parameter übergeben werden sollen, generiert und in die Liste der globalen Bindungen der Funktion eingetragen. Dabei ist darauf zu achten, daß diese Liste von jeder globalen Bindung nur einen Eintrag enthält, auch wenn sie mehrfach referenziert wird. Da Funktionen ineinander verschachtelt sein können, müssen nach der Erweiterungsphase einer lokal definierten Funktion — im oben vorgestellten Beispiel die Funktion *printResult* — alle Bindungen dieser Liste, die auch in der umgebenden Funktion — im Beispiel *print* in *addToGlobal* — global sind, ebenfalls in deren Liste eingetragen werden.

Nachdem die Erweiterungsphase abgeschlossen ist, müssen für jedes Element der Liste der globalen Bindungen die Funktionsaufrufe für *valueInClosure* bzw. *locationInClosure* generiert werden. Analog zu der Funktion, die den Subtyptest zur Laufzeit bei der Projektion automorpher Werte auf ihre Wertkomponente durchführt, werden *valueInClosure* und *locationInClosure* dabei als Literale in den Syntaxbaum gehängt. Dies ist die einzige Typunsicherheit dieses Ansatzes, da der Typ, den solch ein Literal haben soll, explizit definiert werden muß. Der Typüberprüfer kann nur feststellen, ob der Typ des jeweiligen Funktionsaufrufs mit dem definierten Typ, nicht jedoch, ob dieser mit dem tatsächlichen Typ der entsprechenden in *tlClosure* definierten Funktion übereinstimmt. Ändert sich jedoch die Signatur der Funktionen, so muß dementsprechend ein anderer Typ definiert werden, da es ansonsten zu einem Laufzeitfehler kommt.

In der Typüberprüfungsphase schließlich müssen die generierten Funktionsaufrufe auf Typkorrektheit geprüft werden. Da Funktionen ineinander geschachtelt sein können, wird diese Phase erst durchlaufen, wenn der Kode für die äußerste Funktion — im Beispiel *addToGlobal* — generiert worden ist. So wird sichergestellt, daß der generierte Kode für die geschachtelten Funktionen — im Beispiel *printResult* — nur einmal typgeprüft wird.

6.1.5 Praktische Erfahrungen mit dem Ansatz

Mit dem in diesem Abschnitt vorgestellten Ansatz sind zu Testzwecken die Standardbibliotheken übersetzt und ein *Bootstrap* des TL Compilers durchgeführt worden. Gleichzeitig wurden die Übersetzungszeiten und die Größe des generierten TVM Kodes ermittelt und den Werten

Laufzeittyprepräsentationen	**variable globale Bindungen**	
	nein	**ja**
normal	Faktor 3 - 12	Faktor 16 - 21
ubiquitär	Faktor 2,5	Faktor 10

Tabelle 6.1: Änderung der Dateigrößen

des ursprünglichen TL Compilers gegenübergestellt. Dabei hat sich die Übersetzungszeit des Typüberprüfers in der Regel um den Faktor zwei bis drei, in Ausnahmefällen sogar auf das vier– bis fünffache erhöht.

Die Größe des erzeugten TVM Kodes (s. Tabelle 6.1) läßt sich nur anhand der Dateien ermitteln, in die dieser mittels des *extern* Algorithmus geschrieben wird. Dabei hat sich ein sehr uneinheitliches Bild ergeben, da sich die Unterschiede zum ursprünglichen Typüberprüfer breit gestreut im Bereich des drei– bis zwölffachen und in Modulen, in denen Funktionen variable globale Bindungen referenzieren, sogar im Bereich des sechzehn– bis zwanzigfachen bewegen. Diese Heterogenität ist dadurch zu erklären, daß vom *extern* Algorithmus auch alle generierten Laufzeittyprepräsentationen der globalen Bindungen in die Dateien geschrieben werden. Um ein genaueres Bild über die Größe des tatsächlich erzeugten TVM Kodes zu erhalten, wurden deshalb zu Testzwecken die Laufzeittyprepräsentationen, die in die globale Tabelle eingetragen werden, in der alle erzeugten Laufzeittyprepräsentationen verwaltet werden, als ubiquitär gekennzeichnet, so daß sie nicht mit in die Dateien geschrieben wurden, sondern nur deren *UID*. Dabei stellte sich heraus, daß die Dateien relativ einheitlich nur noch um den Faktor 2,5 größer geworden sind. Lediglich bei Modulen, in denen Funktionen variable globale Bindungen referenzieren, lag der Faktor etwa bei zehn. Der Unterschied läßt sich dadurch erklären, daß bei variablen globalen Bindungen in der *setValue* Funktion die Funktion aufgerufen wird, die den Subtyptest innerhalb des Typüberprüfers durchführt.

An dieser Stelle sei darauf hingewiesen, daß die auf diesem Wege erzeugten Dateien sich jedoch nur in dem persistenten Speicher *binden* lassen, in dem der TVM Kode erzeugt worden ist, da nur dort die als ubiquitär gekennzeichneten Laufzeittyprepräsentationen definiert sind. Dies schränkt natürlich die Wiederverwendbarkeit dieser Dateien erheblich ein.

Anhand der untersuchten Leistungsgrößen kann man feststellen, daß die Verwendung dieses Typüberprüfers einen erheblich höheren Zeit– und Plattenbedarf gegenüber dem ursprünglichen Typüberprüfer benötigt. Auch die hier nicht untersuchten Ausführungszeiten der Programme werden sich vergrößern, so daß in den Typüberprüfer ein Schalter eingebaut worden ist, mit dem zwischen den Versionen des Typüberprüfers gewechselt werden kann. Nur derjenige Anwender, der tatsächlich Laufzeittypinformationen über die globalen Bindungen einer Funktion haben möchte, sollte diesen Schalter entsprechend setzen und die Module und Funktionen übersetzen, von denen er die Informationen benötigt.

6.2 Persistente Threads

Ein *Thread* ist eine Repräsentation von Kode, der sich in der Ausführung befindet. Er beschreibt einen einzelnen sequentiellen Kontrollfluß. Innerhalb eines, möglicherweise persistenten, Adressraumes können mehrere *Threads* ausgeführt werden, die auf lokale, aber auch auf gemeinsame Variablen zugreifen. Erzeugt wird ein *Thread*, indem ein nichtparametrisiertes Kodefragment und persistente Daten an einen Evaluator übergeben werden.

Innerhalb des Tycoon Systems sind *Threads* Werte erster Klasse, die wie Daten und Funktionen persistent sind und entweder im Kontext eines individuellen Programmes oder eines persistenten Moduls gebunden werden können. Damit können sie einerseits als Aktivitäten betrachtet werden, die man unter anderem erzeugen, ausführen, synchronisieren, suspendieren sowie terminieren kann, und andererseits als passive Daten, die persistent gespeichert, mit Attributen versehen, mit anderen persistenten Datenstrukturen assoziiert, zwischen den Knoten eines Netzwerkes verschickt und durch Berechnungen manipuliert werden können. Dadurch eignen sie sich besonders für langlebige, verteilte und kooperative Anwendungen [Matthes, Schmidt 94].

Die Möglichkeit, *Threads* persistent zu machen, führt zu dem Wunsch, sie auch wie andere Daten und Funktionen des Objektspeichers zu visualisieren. Problematisch dabei ist, daß persistente *Threads* abstrakte Datentypen sind, und ihre interne Repräsentation somit nicht zugänglich ist. In diesem Abschnitt wird deshalb ein Ansatz vorgestellt, der die Visualisierung suspendierter *Threads* unterstützt. Es handelt sich dabei um eine Erweiterung des übersetzungszeitreflektiven Ansatzes für Funktionen, wie er im letzten Abschnitt vorgestellt worden ist. Allerdings wird im Rahmen dieser Arbeit nur die Idee des Ansatzes erläutert, eine Implementierung hat nicht stattgefunden.

6.2.1 Interne Darstellung

Ein *Thread*, der eine Funktion f ausführt (s. Abbildung 6.3[1]), wird in der TML Zwischenrepräsentation als ein Quadrupel (E,S,L,c) dargestellt, wobei E die Referenzen auf den unveränderlichen Literalvektor, die unveränderlichen globalen Bindungen von f und die von anderen *Threads* nicht zugänglichen, unveränderlichen Aktualparameter von f aggregiert; S beschreibt den aktuellen Zustand des Objektspeichers; L faßt die lokalen Bindungen von f zusammen, die von anderen *Threads* nicht zugänglich sind; c schließlich enthält die Instruktion von f, die gerade ausgeführt wird. Die Visualisierung eines persistenten *Threads* umfaßt damit die Funktion f, ihre Aktualparameter und die zu f lokalen Bindungen, wobei die Laufzeittyprepräsentationen für globale Bindungen mit Hilfe des im letzten Abschnitt beschriebenen Ansatzes gewonnen werden können. Es müssen also zusätzlich noch Laufzeittyprepräsentationen für die Aktualparameter und die lokalen Bindungen generiert werden.

Diese werden wie die mit ihren Laufzeittypinformationen assoziierten globalen Variablen in den *Closure* Rekord eingetragen. Dazu wird der oben definierte Typ *DebugList* um eine Variante *ThreadBrowserInfo* ergänzt. Diese Variante wiederum besteht aus einer Liste des Typs *TypeList*, in die die Aktualparameter und lokalen Bindungen eingetragen werden. Für den Zugriff auf die Elemente dieser Liste müssen Funktionen analog zu denen für die globalen Variablen exportiert werden. Ist z.B. die oben definierte Funktion *addToGlobal* mit dem Wert 3 in einem *Thread* aufgerufen worden, und ist dieser *Thread* nach der Definition von *result* suspendiert worden, so muß der in Abbildung 6.4 dargestellte Zustand visualisiert werden.

6.2.2 Generierter Programmkode

Um die Aktualparameter und lokalen Bindungen mit ihren Laufzeittyprepräsentationen zu assoziieren und in den aktuellen Kontext einer Funktion einzutragen, muß der Programmkode,

---

[1] entnommen aus [Matthes, Schmidt 94]

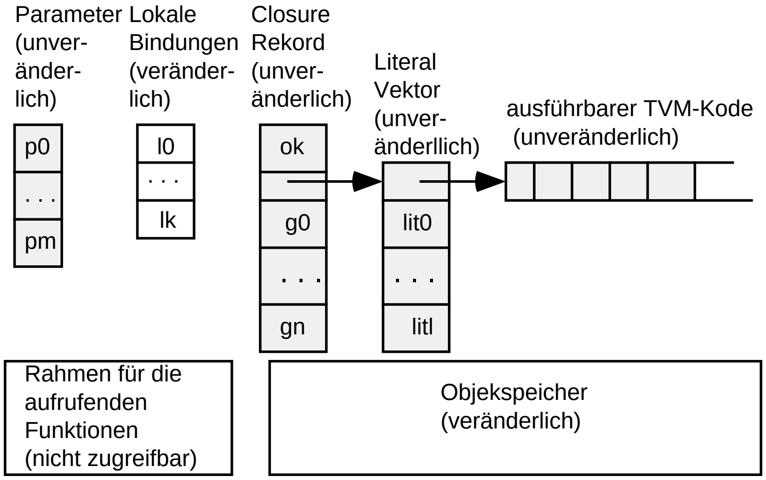

Parameter (unver-änder-lich)
Lokale Bindungen (veränder-lich)
Closure Rekord (unver-änderlich)
Literal Vektor (unver-änderllich)
ausführbarer TVM-Kode (unveränderlich)
p0
. . .
pm
l0
. . .
lk
ok
g0
. . .
gn
lit0
. . .
litl
Rahmen für die aufrufenden Funktionen (nicht zugreifbar)
Objekspeicher (veränderlich)

Abbildung 6.3: TML Repräsentation eines Threads

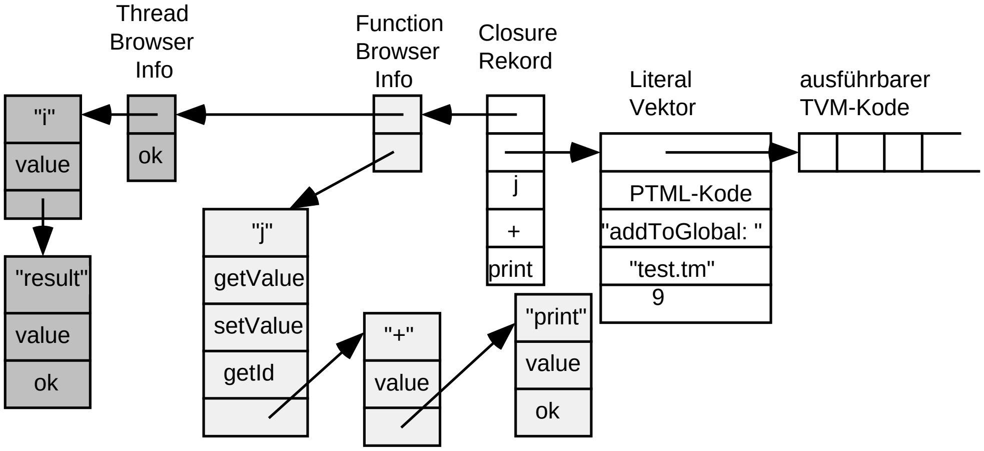

Thread Browser Info
Function Browser Info
Closure Rekord
Literal Vektor
ausführbarer TVM-Kode
"i"
value
ok
"result"
value
ok
"j"
getValue
setValue
getId
"+"
value
j
+
print
"print"
value
ok
PTML-Kode
"addToGlobal: "
"test.tm"
9

Abbildung 6.4: Funktionsabschluß mit Laufzeittypinformationen für alle Bindungen

der die Funktion definiert, umgeschrieben werden. Die Idee des Ansatzes, die hier nur in ihrer Grundkonzeption vorgestellt werden soll, ist es, den Rumpf einer Funktion so umzuschreiben, daß die Aktualparameter und jede nichtanonyme Bindung nach ihrer Definition in einen für die Funktion lokalen Kontext eingetragen werden, der vor der Funktionsdefinition global definiert und initialisiert worden ist. Nach der Funktionsdefinition wird ein Eintrag in der Liste der *Debug* Informationen erzeugt, in dem der lokale Kontext gespeichert wird.

Der zu generierende Programmkode soll anhand des folgenden kleinen Beispiels erläutert werden:

aus

```
let f(x :Int) = begin
  let t = tuple let var a = x + 3  let b = x + 5 end
  t
end
```

wird generiert

```
let f = begin
  let context = tuple
    let var val = tuple_case nil of TypeList with end
  end
  let fIntern(x :Int) =
    valueInContext(context  "x"  dynamic_new(x)
      fun() begin
        let t = begin
          let var a = x + 3
          locationInContext("a"  let getValue = ...  let setValue = ...  let getId = ...
            fun() begin
              let b = x + 5
              valueInContext("b"  dynamic_new(b)
                fun() tuple let var a = a let b = b end)
            end)
        end
        valueInContext("t"  dynamic_new(t) fun() t)
      end)
  contextInClosure(context fIntern)
end
```

Zunächst wird der lokale Kontext einer Funktion als variable Bindung innerhalb eines Tupels definiert. Dadurch kann der jeweils aktuelle Kontext *context.val* von verschiedenen Bindungen über die konstante Bindung *context* referenziert werden. Danach erfolgt die eigentliche Funktionsdefinition, in der der Programmkode des Funktionsrumpfes in eine dem in TML verwendeten *Continuation Passing Style* ähnliche Repräsentation umgeschrieben wird. Analog zu den globalen Variablen werden mit Hilfe der Funktionen

```
valueInContext(A <:Ok  context :Context  name :String  value :dynamic_T  cont() :A) :A
locationInContext(A <:Ok  context :Context  name :String  getValue() :dyanamic_T
  setValue(:dynamic_T) :Ok  getLocation() :LocationId  cont() :A) :A
```

alle konstanten bzw. variablen Wertbindungen nach ihrer Definition in den lokalen Kontext zusammen mit ihren Laufzeittyprepräsentationen eingetragen. Beide Funktionen speichern

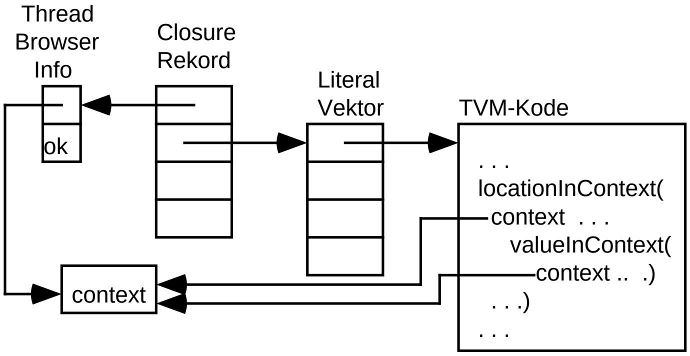


Abbildung 6.5: Funktionsabschluß nach der Definition von f

zuerst den aktuellen Zustand des Kontextes, tragen dann die jeweiligen Komponenten in den Kontext ein, führen die als Parameter übergebene Funktion *cont* aus, stellen den ursprünglichen Kontext wieder her und geben das Ergebnis von *cont* zurück. In der Funktion *cont* werden jeweils die restlichen Bindungen des Programmkonstruktes erzeugt, das gerade definiert wird. Auf diese Weise enthält der Kontext immer nur die aktuell sichtbaren Bindungen. Als Ergebnis wird von der innersten *cont* Funktion das ursprünglich im Programmkode definierte Konstrukt zurückgegeben.

Für alle TL Wertkonstruktoren müssen Regeln definiert werden, um den entsprechenden Programmkode zu generieren. Hier soll beispielhaft die Regel für Tupelwerte erläutert werden. Jede Tupelkomonente (im Beispiel a und b) ist zunächst zu definieren. Danach wird sie mittels *valueInContext* oder *locationInContext* in den lokalen Kontext eingetragen. Der Parameter *cont* ist eine Funktion, die die restlichen Komponenten in analoger Weise definiert und registriert. Als Ausnahme wird bei der letzten Komponente eine Funktion übergeben, die das ursprünglich zu definierende Tupel zurückgibt, dessen Komponenten aus den entsprechend zuvor definierten Bindungen bestehen. Dieses Tupel wird wie im ursprünglichen Programmkode gebunden (im Beispiel an t) und danach ebenfalls mit *valueInContext* oder *locationInContext* registriert. Würde die Ausführung von *fIntern* beispielsweise unterbrochen werden, nachdem die Tupelkomponente b in den lokalen Kontext eingetragen wurde, bevor aber die als Parameter übergebene Funktion *cont* ausgeführt worden ist, so enthält der lokale Kontext von *fIntern* die Einträge für x, a und b. Da die Funktionen *locationInContext* und *valueInContext* den ursprünglichen Kontext wiederherstellen, nachdem die Funktion *cont* ausgeführt worden ist, enthält der lokale Kontext nach der Definition von t nur noch den Eintrag von x. Somit enthält bei der Ausführung von *fIntern* der lokale Kontext immer nur die aktuellen lokalen Bindungen.

Damit der Anwendungsprogrammierer mit Hilfe entsprechender durch das Modul *closure* exportierter Funktionen zur Laufzeit auf den jeweils aktuellen lokalen Kontext einer Funktion zugreifen kann, wird dieser durch die Funktion

contextInClosure(context :Context closure :Closure) :Closure

aus dem Modul *tlClosure* in den *Closure Record* von *fIntern* eingetragen.

Indem der generierte Programmkode in einer *begin end* Sequenz definiert ist, wird dieser bei der Definition von *f* ausgeführt und nur der von *fIntern* referenzierte Funktionswert an *f* gebunden. Da der lokale Kontext in den *Closure Record* dieses Funktionswertes eingetragen ist und den gleichen Wert referenziert wie die Funktionen *locationInClosure* und *valueInClosure* innerhalb dieses Funktionswertes, kann der Anwendungsprogrammierer während der Ausführung von *f* jederzeit auf den aktuellen lokalen Kontext zugreifen (s. Abb. 6.5).

Auch dieser Ansatz, der übrigens ebenfalls zu Zwecken der Fehlersuche eingesetzt werden kann, zeigt, wie man bestimmte Probleme elegant mit Übersetzungszeitreflektion lösen kann. Da er jedoch zu erheblich höheren Übersetzungs– und Ausführungszeiten führt, sollte er nur in Ausnahmefällen wie z.B. zur Fehlersuche eingesetzt werden.

7. Realisierung der Daten– und Funktionsvisualisierung

In den vorherigen Kapiteln sind die Voraussetzungen zur Realisierung des generischen Dienstes zur Daten– und Funktionsvisualisierung beschrieben worden, also die Definition der Anforderungen, die Gestaltung der grafischen Benutzerschnittstelle, die Integration dynamischer Typisierung in TL und systemunterstützende Funktionalität zur Visualisierung der globalen Bindungen einer Funktion. In diesem Kapitel schließlich wird die in der Bibliothek *browseenv* realisierte Implementierung beschrieben. Dazu werden im ersten Abschnitt der Aufbau der Bibliothek und der Ablauf der Visualisierung überblicksartig vorgestellt. In den folgenden Abschnitten werden die wesentlichen Konzepte der Komponenten der Bibliothek erläutert, die bei der Realisierung der generischen Daten– und Funktionsvisualisierung eine Rolle spielen.

7.1 Die Bibliothek zur Daten– und Funktionsvisualisierung

Das Hauptstrukturierungsmittel für Werte und Typrepräsentationen im Objektspeicher sind sogenannte *Environments* [Geisler 95]. Sie sind in [Morrison et al. 94; Dearle 89] definiert als die unendliche Vereinigung aller benannten Kreuzprodukte. Das bedeutet konkret, daß Environments eine nicht notwendigerweise geordnete Sequenz von Werten des weiter unten definierten Datentyps *Binding* enthalten. Solche Bindungen, die benutzerdefinierte Namen an semantische Objekte binden, spielen damit eine zentrale Rolle bei der Visualisierung von Objekten und dem Datenaustausch zwischen ihren Bildschirmrepräsentationen in TL. Anders als bei [Morison et al. 90] umfassen sie nicht nur konstante und variable Wertbindungen sondern auch Typbindungen. Das führt neben einer Vereinheitlichung der Benennungsmechanismen zu einer Vermeidung redundanter Deklarationen für identische semantische Objekte in verschiedenen Werkzeugen [Matthes 93]. Dadurch ist es möglich, einen universellen Rahmen zur Daten– und Funktionsvisualisierung bereitzustellen, in dem Objekte nur einmal auf dem Bildschirm dargestellt werden, auch wenn sie auf unterschiedlichen Zugriffspfaden erreicht werden, und der den uniformen Datenaustausch zwischen den Bildschirmobjekten erlaubt.

Bindungen sind durch den varianten Tupeltyp

```
Let Binding = Tuple
  name :String
  case typeCase with
    type :typeRep_T
  case valueCase with
    value :dynamic_T
  case locationCase with
```

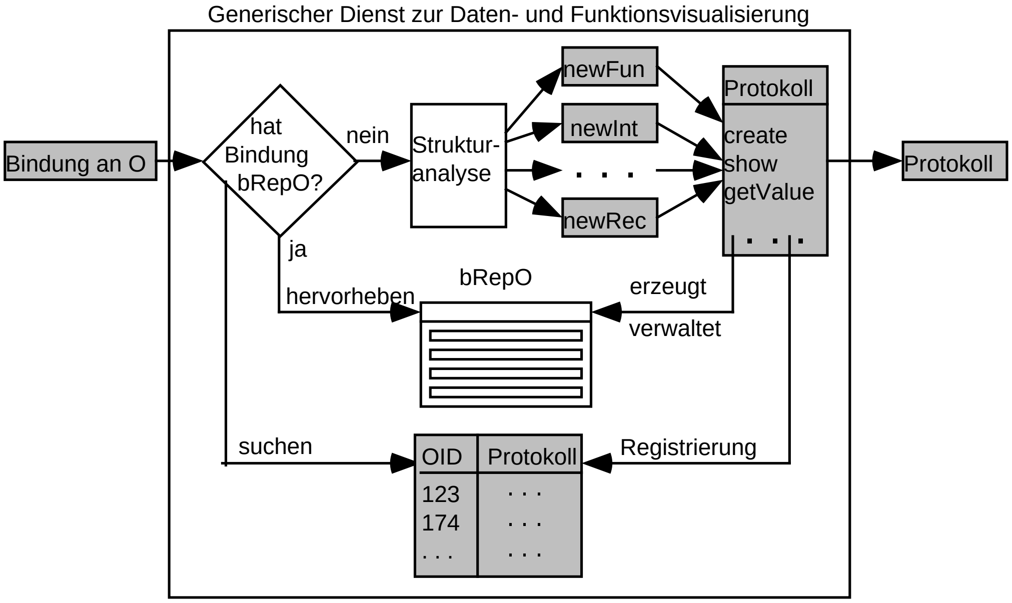

Generischer Dienst zur Daten- und Funktionsvisualisierung
Bindung an O
hat Bindung bRepO?
nein
ja
Struktur-analyse
newFun
newInt
newRec
Protokoll
create
show
getValue
Protokoll
hervorheben
bRepO
erzeugt
verwaltet
suchen
Registrierung
OID
Protokoll
123
174

Abbildung 7.1: Ablauf der Daten– und Funktionsvisualisierung

```
    value() :dyanamic_T
    setValue(:dynamic_T) :Ok
    getLocation() :Location
end
```

des Moduls *Environment* repräsentiert. Variable Wertbindungen sind im Objektspeicher durch Zellen, bestehend aus einer Basisadresse und einer dazu relativen Adresse, dargestellt und eindeutig identifizierbar [Gawecki, Matthes 94]. Da eigentlich nur der Compiler diese Zelle kennt, werden der lesende und verändernde Zugriff auf die Wertkomponente durch die beiden Funktionen *value* und *setValue* realisiert. Bei der Daten– und Funktionsvisualisierung sollen Objekte nur einmal auf dem Bildschirm dargestellt werden, um das Verständnis für die Zusammenhänge zwischen den Objekten zu verdeutlichen und um den Datenaustausch zwischen Bildschirmrepräsentationen realisieren zu können. Da die Wertkomponente variabler Bindungen durch ihre Zelle eindeutig identifizierbar ist, wird noch die Funktion *getLocation* benötigt, die die Zelle einer variablen Wertbindung liefert.

Die Visualisierung einer Bindung besteht aus vier Schritten (s. Abb. 7.1):

1. Es wird überprüft, ob für das durch die Bindung repräsentierte Objekt O des persistenten Speichers schon eine Bildschirmrepräsentation $bRep_O$ existiert. Ist dies der Fall, so wird sie nur optisch hervorgehoben.
2. Andernfalls wird die Struktur von O analysiert.
3. Entsprechend der Strukturanalyse wird ein uniformes Protokoll generiert, das $bRep_O$ kapselt und über Protokollfunktionen zugänglich macht.
4. Schließlich wird $bRep_O$ erzeugt und angezeigt.

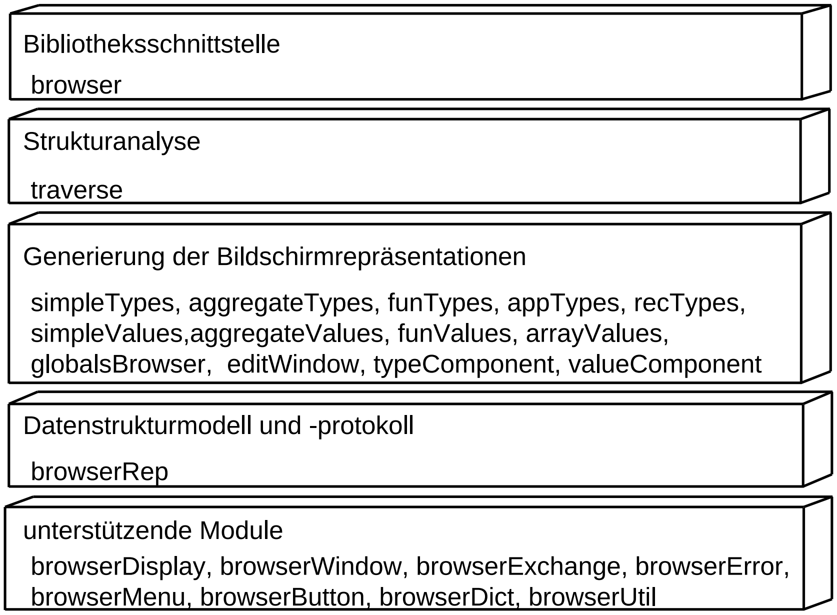

Bibliotheksschnittstelle
browser
Strukturanalyse
traverse
Generierung der Bildschirmrepräsentationen
simpleTypes, aggregateTypes, funTypes, appTypes, recTypes,
simpleValues,aggregateValues, funValues, arrayValues,
globalsBrowser, editWindow, typeComponent, valueComponent
Datenstrukturmodell und -protokoll
browserRep
unterstützende Module
browserDisplay, browserWindow, browserExchange, browserError,
browserMenu, browserButton, browserDict, browserUtil

Abbildung 7.2: Struktur der Bibliothek zur Daten– und Funktionsvisualisierung

Diese Schritte sollen vor dem Anwendungsprogrammierer verborgen werden und der Zugriff auf den Dienst zur generischen Daten– und Funktionsvisualisierung über eine nach außen sichtbare Schnittstelle erfolgen. Aus diesen Überlegungen heraus ist die Bibliothek in die folgenden in Abb. 7.2 dargestellten Blöcke unterteilt:

- Der unterste Block stellt einige Module zur Unterstützung der Visualisierung zur Verfügung.
- Im zweiten Block werden das abstrakte Datenstrukturmodell, das einen universellen Rahmen zur Repräsentation von Objekten bietet, und das abstrakte Protokoll definiert, über das die Objekte angesprochen werden können.
- Der dritte Block dient der Generierung der Bildschirmrepräsentationen für die verschiedenen Objekte.
- Der vierte Block implementiert die Strukturanalyse, von der aus die Generierung der Bildschirmrepräsentationen aufgerufen wird.
- Das Modul *browser* schließlich stellt die für den Anwendungsprogrammierer nach außen sichtbare Schnittstelle des Dienstes dar.

In den folgenden Abschnitten werden die einzelnen Blöcke genauer vorgestellt. Um die darin vorgestellten Konzepte besser verstehen zu können, orientiert sich die Reihenfolge der Abschnitte jedoch nicht an der in Abbildung 7.2 dargestellten Struktur der Bibliothek.

7.2 Strukturanalyse

Im Gegensatz zu den benutzerdefinierbaren Editoren aus [Müßig 94], ist bei der generischen Daten– und Funktionsvisualisierung nicht bekannt, welche Struktur das zu visualierende Objekt hat. Die Struktur muß deshalb getrennt für die Wert– und Typkomponenten einer Bindung analysiert werden. Ziel ist es dabei nicht, die ganze Struktur des Objektes zu erfassen, da diese sehr komplex und deren Analyse entsprechend laufzeitintensiv sein kann. Außerdem möchte der Anwender eventuell gar nicht alle Komponenten detailliert visualisieren. Deshalb werden die Wert– bzw. Typkomponente einer Bindung lediglich mit Hilfe von *inspect* aus den Modulen *dynamic* bzw. *typeRep* um einen Schritt expandiert und mit den daraus gewonnenen wert– bzw. typspezifischen Informationen die entsprechende Funktion zum Generieren des Protokolls und der Bildschirmrepräsentation aufgerufen. Nur Typidentifikatoren werden bis zu ihrer Definition expandiert, da es für das Verständnis von Typen nicht notwendig ist, Bindungen von Typidentifikatoren an andere Typidentifikatoren zu visualisieren. Wertbindungen, bei deren Typ es sich um eine nichtabstrakte Typoperatoranwendung handelt, werden ebenfalls um einen weiteren Schritt expandiert, um dessen Wertkomponente zu visualisieren. Die Struktur des Typoperators und der Typbindungen können mit Hilfe des Typs der Wertbindung visualisiert werden.

Da die Struktur von Komponenten eines Wertes bzw. Typs erst analysiert wird, wenn der Anwender diese visualisieren möchte, entsteht eine komplexe rekursive Wechselbeziehung zwischen dem Modul, das die Strukturanalyse durchführt, und den Modulen, die die wert– bzw. typspezifische Visualisierung realisieren. Da in TL wechselseitig rekursive Funktionsaufrufe zwischen Modulen verboten sind, muß die Funktion, die die Strukturanalyse durchführt, als Parameter mit übergeben werden:

binding(parent :window.T bndg :Environment.Binding actPos :BrowserUtil.Location
keepPermanent, isSub :Bool) :BrowserRep.T

Sie erwartet als Parameter das Elternfenster, die zu visualisierende Bindung, die Position, an der die Bildschirmrepräsentation erscheinen soll, sowie zwei weitere Parameter, die der Strukturanalyse mitteilen, ob die zu visualisierende Bindung permanent mit ihrer Bildschirmrepräsentation gespeichert werden soll (s. Abschnitt 7.6) und ob die Bindung innerhalb der Bildschirmrepräsentation eines strukturierten Wertes visualisiert werden soll.

Der Datenaustausch zwischen Bildschirmrepräsentationen wird durch den Austausch von Bindungen realisiert. Dazu muß es möglich sein, aus einer visualisierten Bindung den aktuellen Inhalt der Wert– bzw. Typkomponente zu erhalten sowie diesen bei variablen Wertbindungen zusätzlich verändern zu können. Deshalb werden in der Strukturanalyse folgende Funktionen generiert und als weitere Parameter an die Visualisierungskomponenten übergeben (s. Abschnitt 7.5):

getValue() :Environment.Bindung
*setValue(:Environment.Binding) :**Ok***

Eine andere Operation, die bei Wertbindungen im Gegensatz zu Typbindungen notwendig ist, ist die Visualisierung des Typs der Wertkomponente. Da jedoch Wertbindungen, deren Typ eine nichtabstrakte Typoperatoranwendung ist, um zwei Schritte expandiert werden, läßt sich anhand des visualisierten Wertes nicht mehr feststellen, daß es sich bei dem Typ um eine Typoperatoranwendung handelt. Deshalb wird der Funktion, die die Bildschirmrepräsentation generiert, folgende Funktion zur Visualisierung des gekapselten Typs übergeben:

```
browseType(parent :window.T  actPos :Location) :T
```

7.3 Abstraktes Datenstrukturmodell

Bei der generischen Daten– und Funktionsvisualisierung entstehen viele unterschiedliche Bildschirmobjekte. Um diese heterogenen Objekte in eine gemeinsame Umgebung zu integrieren und in objektorientierter Form ansprechen zu können, ist es notwendig, ein gemeinsames Protokoll für alle Bildschirmobjekte zu definieren [Kilberth et al. 93]. Das für diesen Dienst definierte Protokoll, das im ersten Teil dieses Abschnittes beschrieben wird, setzt auf dem für Editoren auf [Müßig 94], ist jedoch an die zusätzlichen Anforderungen der generischen Daten– und Funktionsvisualisierung und an die Verwendung der StarView Klassenbibliothek angepaßt. Für den einheitlichen Zugriff auf die Komponenten werden Funktionen bereitgestellt, die im zweiten Teil erläutert werden.

7.3.1 Protokoll

In dem Protokoll werden die allen Objekten gemeinsamen Eigenschaften definiert und zur Verfügung gestellt. Funktionale Komponenten ermöglichen den Zugriff auf die gekapselten objektspezifischen Eigenschaften. In diesem Abschnitt werden die verschiedenen Komponenten des Protokolls vorgestellt.

Interne Daten, die den zugehörigen Rahmen und den Status einer Bildschirmrepräsentation betreffen, werden durch den Typoperator *Hidden* beschrieben:

```
Let Hidden(BrowserT <:Ok) <:Ok = Tuple
  var windowId :browserWindow.T
  var menu :list.T(MenuT)
  var buttons :list.T(pushButton.T)
  var parent :optional.T(BrowserT)
  var keepPermanent :Bool
  state :set.T(State)
  size :Location
end
```

Diese Informationen sind variabel, da sie sich während der Lebensdauer eines Ojektes ändern können. Es ist möglich, visualisierte Objekte innerhalb eines Rahmens beliebig zu kombinieren und so eine hierarchische Struktur aufzubauen. Nur solche Objekte, die nicht innerhalb eines anderen Objektes dargestellt werden, besitzen ein eigenes Fenster (*windowId*) und eine Liste von Menüknöpfen mit zugehörigen Menüeinträgen (*menu*). Diese Objekte sind im Gegensatz zu den in [Müßig 94] vorgestellten Editoren völlig unabhängig von anderen visualisierten Objekten. Ansonsten verweist *parent* auf das Objekt, in dem das Objekt dargestellt wird. Repräsentiert das visualisierte Objekt eine veränderliche Wertbindung eines Basistyps, so enthält *buttons* eine Liste von Knöpfen, mit denen die Wertkomponente manipuliert werden kann (s. Abschnitt 4.2.1). Bei der Generierung des Protokolls kann der Anwender festlegen, ob das Protokoll des zu visualisierenden Objektes permanent in der entsprechenden Tabelle (s. Abschnitt 7.4.2) gespeichert werden soll. Entsprechend wird die Komponente *keepPermanent* gesetzt. Die permanente Speicherung der Protokolle eignet sich besonders, wenn von Bindungen häufiger Bildschirmrepräsentationen erzeugt werden müssen. Wird der Dienst zur Daten– und Funktionsvisulisierung z.B. zur Darstellung der Berechnungsergebnisse eingesetzt, so kann ein Wert an verschiedene Namen gebunden werden. Jedesmal, wenn dieser Wert erneut visualisiert werden soll und keine Bildschirmrepräsentation mehr existiert, so muß nur die Bildschirmrepräsentation aus dem bestehenden Protokoll generiert werden, nicht jedoch das Protokoll selbst. In *size* wird die Größe des visualisierten Objektes, bestehend aus der Breite und der Höhe, die beide variabel sind, festgehalten.

Ein visualisiertes Objekt kann sich gleichzeitig in mehreren gültigen Zuständen befinden, die in *state* gespeichert sind. Abhängig davon, welche Zustände die Menge aktuell enthält, ist genau definiert, welche Operationen auf einem Objekt erlaubt sind. Die zulässigen Zustände werden in Tabelle 7.1 erklärt.

Zustand	**Bedeutung**
new	Ist während der Initialisierung gesetzt.
exists	Das Objekt ist mit *create* erzeugt worden.
visible	Das Objekt ist auf dem Bildschirm sichtbar.
windowed	Das Objekt befindet sich in einem Rahmen.
showThis	Ein Objekt, das einen Rahmen besitzt, soll auf dem Bildschirm angezeigt werden.

Tabelle 7.1: Zulässige Objektzustände

Zwar wird der Typoperator *Hidden* innerhalb des Dienstes zur Daten– und Funktionsvisualisierung sichtbar exportiert, Manipulationen sollten aber nur in den dafür definierten Hilfsmodulen vorgenommen werden und nicht in den Funktionen, die die Bildschirmrepräsentation generieren, um eine einheitliche Funktionalität dieser Komponenten zu gewährleisten.

Das Protokoll wird durch den folgenden rekursiven Datentyp exportiert:

```
Let Rec T <:Ok = Tuple
  hidden :Hidden(T)
  label :String
  var menuItems :list.T(MenuItem)
  createButtons :Bool
  allowDragAndDrop :Bool
  create(parent :window.T  actPos :Location) : Ok
  apply() :Bool
  refresh() : Ok
  show() : Ok
  unshow() : Ok
  getValue() :Environment.Binding
  setValue(binding :Environment.Binding) : Ok
end
```

In *label* wird, sofern das Objekt über einen eigenen Rahmen verfügt, die Kopfzeile gespeichert. Die Komponente *menuItems* enthält die für ein Objekt gültigen Menüeinträge mitsamt der zugehörigen Aktionen. Bei variablen Wertbindungen von Basistypen hat *createButtons* den Wert *true*, so daß für dieses Objekt Manipulationsknöpfe erzeugt werden. Nur wenn *allowDragAndDrop* den Wert *true* hat, darf die visualisierte Typbindung per *Drag&Drop* verändert bzw. in ein anderes Objekt kopiert werden.

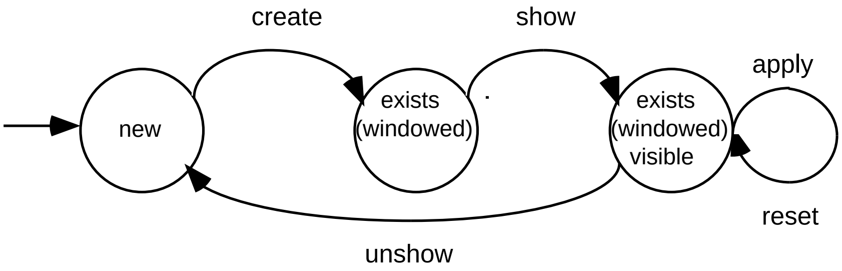


Abbildung 7.3: Mögliche Zustandsübergänge der Browser

Die funktionalen Komponenten realisieren den Zugriff auf die gekapselten, objektspezifischen Eigenschaften. Sie enthalten nur die für ein spezielles Objekt gültigen Anweisungen. Diejenigen Anweisungen, die bei allen Objekten gleich sind, werden durch die im nächsten Abschnitt beschriebenen Funktionen realisiert. Diese überprüfen auch Integritätsbedingungen (s. Abschnitt 7.3.2), so daß die funktionalen Komponenten niemals direkt aufgerufen werden dürfen.

Die Funktionen *create, apply, refresh, show* und *unshow* dienen der Manipulation der Bildschirmrepräsentation, die erst durch *create* erzeugt werden muß, bevor die anderen Funktionen ausgeführt werden dürfen. Da Werte von Basistypen per Tastatur– oder Mauseingaben manipulierbar sind, können in den Bildschirmrepräsentationen und der Wertkomponente unterschiedliche Werte existieren. Durch *apply* wird die Wertkomponente mit den Inhalten der Bildschirmrepräsentationen aktualisiert, während *reset* die umgekehrte Richtung realisiert. *Reset* muß bei Wertbindungen immer aufgerufen werden, wenn ein Objekt mit *show* auf dem Bildschirm dargestellt wird, um die Bildschirmobjekte mit den richtigen Werten zu füllen. Die Funktion *unshow* schließlich entfernt ein Objekt vom Bildschirm.

Die Funktionen *getValue* und *setValue* erlauben den lesenden und verändernden Zugriff auf ein Objekt und werden zum Datenaustausch zwischen Objekten benötigt. Der Aufruf von *setValue* ist allerdings nur bei variablen Wertbindungen erlaubt und führt ansonsten zu einer Fehlermeldung.

7.3.2 Zugriffsfunktionen

Für den Zugriff auf die funktionalen Komponenten des Protokolls gelten Integritätsbedingungen, die durch die Zugriffsfunktionen realisiert werden. Für die Komponenten, die die Manipulation von Bildschirmobjekten betreffen, erfolgt deren Überprüfung anhand der oben definierten Zustände. Dabei sind die in Abb. 7.3[1] dargestellten Zustandskombinationen und –übergänge möglich, wobei die in Klammern stehenden Zustände nur bei solchen Objekten auftreten können, die einen eigenen Rahmen besitzen. Es ist zu beachten, daß das Protokoll nach dem Schließen der Bildschirmrepräsentation mittels *unshow* wieder in den Zustand *new* übergeht und nicht in *exists* (zur Begründung s. Abschnitt 7.4.2).

Die Funktionen, die die Zustandsübergänge realisieren, überprüfen zunächst, ob das Objekt eine gültige Zustandskombination aufweist. Ist dies der Fall, so werden die entsprechende funktionale Komponente des Protokolls ausgeführt und der Nachfolgezustand gesetzt. Die

---

[1] entnommen aus [Müßig 94]

Funktionen *show* und *unshow* haben zusätzlich die Aufgabe, das *Drag&Drop* Verfahren mit Hilfe des Moduls *browserExchange* zu initialisieren, falls das Attribut *allowDragAndDrop* des Protokolls des zu visualisierenden Objektes den Wert *true* besitzt. Für alle Objekte ist dabei ein *Drag* erlaubt, jedoch nur für variable Wertbindungen ein *Drop*. Außerdem muß das Protokoll in diesen Funktionen in die Tabelle, in der die Objekte mit ihren Bildschirmrepräsentationen gespeichert werden (s. Abschnitt 7.4.2) eingetragen bzw. aus ihr entfernt werden, sofern das Attribut *keepPermanent* den Wert *false* hat. Andernfalls ist der Eintrag schon nach der Generierung des Protokolls erfolgt.

Der Datenaustausch ist nur zwischen auf dem Bildschirm sichtbaren Objekten erlaubt. Die Typsicherheit wird dabei durch *getValue* und *setValue* gewährleistet. Die Funktion *setValue* unterbindet den Austausch von Typbindungen und überprüft, ob der Typ des neuen Wertes in Subtypbeziehung zum Typ des alten Wertes steht. Nur wenn beide Bedingungen erfüllt sind, wird der alte Wert durch den neuen ersetzt. Andernfalls erscheint eine Fehlermeldung auf dem Bildschirm.

7.4 Unterstützende Schnittstellen

Die in diesem Abschnitt vorgestellten Schnittstellen realisieren allen Objekten gemeinsame unterstützende Funktionalitäten. Zur Realisierung zweier Hauptziele der Daten– und Funktionsvisualisierung — alle Objekte nur einmal auf dem Bildschirm darzustellen und den Datenaustausch zwischen Bildschirmrepräsentationen zu ermöglichen — existieren die beiden Module *browserDict* und *browserExchange*. Wie das Modul *browserDisplay* auch, werden sie jeweils in einem eigenen Unterabschnitt betrachtet.

7.4.1 Allgemeine unterstützende Schnittstellen

Die meisten Hilfsschnittstellen bedürfen keiner umfassenden Erläuterung, so daß sie nur überblicksartig vorgestellt werden.

browserError: Alle innerhalb dieses Dienst generierten Ausnahmepakete werden durch diese Schnittstelle auf dem Bildschirm visualisiert.

browserUtil: In diesem Modul sind alle diejenigen Unterstützungsfunktionen zusammengefaßt, die keiner anderen Schnittstelle zugeordnet werden können. Das betrifft die Erzeugung einiger StarView Objekte und die Bestimmung der Größe solcher Objekte.

browserWindow: Funktionen zur Erzeugung und Manipulation der Rahmenfenster, in denen die visualisierten Objekte eingebettet sind, werden durch diese Schnittstelle exportiert.

browserButton: Falls im Protokoll die Komponente *createButtons* den Wert *true* hat, so werden die Manipulationsknöpfe durch dieses Modul erzeugt und verwaltet.

browserMenu: Entsprechend der Einträge in *menuItems* wird in dieser Schnittstelle das zugehörige Menü erzeugt.

7.4.2 Verwaltung der visualisierten Objekte

Um überprüfen zu können, ob für ein Objekt schon eine Bildschirmrepräsentation existiert, wird es zusammen mit dem generierten Protokoll in eine Tabelle eingetragen. Durch einen Parameter kann der Anwender bei der Generierung des Protokolls angeben, ob die Komponente *keepPermanent* des Protokolls den Wert *true* haben soll und damit dieser Eintrag permanent in dieser Tabelle gespeichert, oder nach dem Schließen der Bildschirmrepräsentation wieder entfernt wird.

Die Verwaltung dieser Tabelle erfolgt durch das Hilfsmodul *browserDict*. Lesende, einfügende und löschende Zugriffe erfolgen einheitlich für Bindungen über exportierte Funktionen, wobei die Tabelle durch das Modul gekapselt ist. Tatsächlich existieren für Typen, konstante und variable Wertbindungen jeweils eigene Tabellen. Die Verwendung der richtigen Tabelle erfolgt intern in Abhängigkeit von der Art der Bindung. Bei Typ– und konstanten Wertbindungen ist die Objektidentität des Objektbezeichners der Schlüssel der Tabellenelemente, bei variablen Wertbindungen die durch *getLocation* erhaltene Zelle.

Bildschirmobjekte sind in **Tycoon** normalerweise nicht persistent, da sie auf dem **StarView Server** erzeugt und verwaltet werden. Es gibt aber einen Mechanismus, flüchtige Daten trotzdem persistent zu machen, indem sie in eine Datenstruktur eingetragen werden, die individuell für jedes Objekt dieser Datenstruktur zu genau festgelegten Zeitpunkten Operationen zum Erzeugen, Speichern oder Vernichten aufruft. Diese Operationen müssen für jedes Objekt beim Eintrag in die Datenstruktur mit übergeben werden. So wird z.B. jedes Objekt dieser Datenstruktur beim erneuten Start einer **Tycoon** Sitzung durch Aufruf der entsprechenden Funktion neu erzeugt. Um nicht jedes Bildschirmobjekt verwalten zu müssen, wird immer das komplette Protokoll einer zu visualisierenden Bindung beim Eintrag in die oben beschriebene Tabelle in die Datenstruktur eingefügt und beim Entfernen aus der Tabelle wieder aus ihr gelöscht.

Für jedes Protokoll wird eine eigene Funktion *create* übergeben. Falls sich das Protokoll im Zustand *new* befindet, führt die Funktion keine Operationen aus. Hat es dagegen den Zustand *visible*, so müssen wegen der Integritätsbedingungen der Zustand auf *new* und die Funktion *display* (s. Abschnitt 7.4.3) aufgerufen werden, damit die Bildschirmrepräsentation neu erzeugt und auf dem Bildschirm angezeigt werden kann. Andere mögliche Zustände gibt es nicht. Würde ein Protokoll allerdings durch *unshow* in den Zustand *exists* übergehen, so müßten auch für diese Protokolle neue Bildschirmrepräsentationen erzeugt werden, ohne diese zu visualisieren. Da in einer persistenten Umgebung jedoch über viele Sitzungen hinweg eine Vielzahl von Objekten erzeugt und somit auch visualisiert werden, kann die Tabelle eine große Anzahl von Objekten enthalten. Da wahrscheinlich aber nur die wenigsten dieser Objekte in jeder Sitzung visualisiert werden sollen, wäre es zu aufwendig, jedesmal für alle Objekte der Tabelle eine neue Bildschirmrepräsentation erzeugen zu müssen, so daß das Protokol bei *unshow* in den Zustand *new* übergeht.

7.4.3 Generierung der Bildschirmrepräsentation

Ist für ein Objekt das Protokoll definiert worden, so kann es mittels der durch das Modul *browserDisplay* exportierten Funktion *display* auf dem Bildschirm angezeigt werden. Dabei wird zwischen zwei Fällen unterschieden — das Protokoll des Objektes befindet sich in dem Zustand *visible*, so daß die Bildschirmrepräsentation nur noch einmal hervorgehoben werden muß, oder es muß erst eine Bildschirmrepräsentation entsprechend dem Protokoll erzeugt und auf dem Bildschirm angezeigt werden. In beiden Fällen werden die oben beschriebenen Zugriffsfunktionen verwendet. Muß eine Bildschirmrepräsentation neu erzeugt werden, so müssen ebenfalls der Rahmen, das Menü und gegebenenfalls die Manipulationsknöpfe der Bildschirmrepräsentation generiert werden.

7.4.4 Unterstützung des Datenaustauschs

Der Dienst zur Daten– und Funktionsvisualisierung soll den Datenaustausch über das Klemmbrett und durch das *Drag&Drop* Verfahren unterstützen. Die dazu benötigte **StarView** Funktionalität wird durch das Modul *browserExchange* (s. Anhang C.1) bereitgestellt und soll im folgenden erläutert werden.

Datenaustausch über das Klemmbrett

Für alle Bildschirmobjekte, die mittels dieses oder eines anderen Dienstes erzeugt worden sind, stellt *browserExchange* ein Klemmbrett zum Datenaustausch zur Verfügung. Daten sollten nur durch die Menüoperationen *Cut*, *Copy* und *Paste* in das Klemmbrett gestellt bzw. aus ihm herauskopiert werden.

Um Daten im Klemmbrett abzulegen, müssen sie mit einem Format versehen werden, wobei ein Objekt in verschiedenen Formaten gleichzeitig abgespeichert werden kann. Über die den Formaten entsprechenden Zugriffsfunktionen können Objekte ins Klemmbrett kopiert bzw. herauskopiert werden. Da Bindungen jedoch keinem der in **StarView** vordefinierten Formate entsprechen, wird für die Daten– und Funktionsvisualisierung ein eigenes Format *Bindings* definiert. Jedes Klemmbrett wird mit den Bindungen des aktuell im Klemmbrett abgelegten Objektes aggregiert. Der Zugriff auf die Bindungen wird über die Funktionen *copyHdl* und *pasteHdl* (s. Anhang C.1) gesteuert. Die Funktion *copyHdl* liefert eine Funktion zurück, die die mittels der als Parameter übergebenen Funktion *selectedBindings* erhaltenen Bindungen in das Klemmbrett kopiert. Die Funktion *pasteHdl* liefert eine Funktion, die überprüft, ob das Objekt im Klemmbrett das Format *Bindings* hat. Ist dies der Fall, so wird die als Parameter übergebene Funktion *insertBindings* mit den im Klemmbrett gespeicherten Bindungen ausgeführt. Andernfalls wird die ebenfalls als Parameter übergebene Funktion *default* ausgeführt. Die so erhaltenen Funktionen müssen bei jedem Objekt mit den Menüeinträgen *Copy* bzw. *Paste* verknüpft werden, um den Datenaustausch zwischen Objekten über das Klemmbrett realisieren zu können.

Zwei weitere Funktionen *insertEnablePaste* und *deleteEnablePaste* werden für den Menüeintrag *Paste* benötigt, um diesen zu steuern. Immer, wenn Bindungen in das Klemmbrett kopiert werden, muß für alle Bildschirmobjekte der Menüeintrag *Paste*, sofern er vorhanden ist, aktiviert werden. Entsprechend muß dieser Eintrag deaktiviert werden, wenn Bindungen aus dem Klemmbrett gelöscht werden. Dazu werden alle Menüs, die diesen Eintrag enthalten, in einer Tabelle verwaltet. Wird ein Objekt auf dem Bildschirm angezeigt, so muß der Eintrag *Paste* mit *insertEnablePaste* und der Funktion, die diesen Eintrag aktiviert bzw. deaktiviert, in die Tabelle eingetragen und wenn es vom Bildschirm gelöscht wird mit *deleteEnablePaste* wieder aus der Tabelle entfernt werden.

Datenaustausch per Drag&Drop

Der Datenaustausch per *Drag&Drop* wird in **StarView** über die Klassen *DragServer*, die das Ziehen (*Drag*) eines Objektes steuert, und *DropEvent*, die das Fallenlassen eines Objektes (*Drop*) steuert, realisiert. Die Schnittstelle *BrowserExchange* stellt einen eigenen *DragServer* für Objekte dieses Dienstes bereit.

Um dieses Verfahren für ein Bildschirmobjekt anwenden zu können, müssen die Funktionen *initDrag* und *initDrop* (s. Anhang C.1) aufgerufen werden. Bei einem durch *initDrag*

Modul	**Funktionsname**	**visualisierte Typen**
simpleTypes	newInfo	Basistypen, **Ok**
	newAdt	abstrakte Datentypen
aggregateTypes	newRecord	Tupel, Rekords, Ausnahmetypen
	newVariant	variante Tupel
funTypes	new	Funktionen, Typoperatoren
appTypes	newArray	Felder
	newApp	Typoperatoranwendungen
recTypes	new	rekursive Typen

Tabelle 7.2: Funktionen zur Visualisierung von Typen

initialisierten Objekt können durch Drücken der linken Maustaste die durch die als Parameter übergebene Funktion *getBindings* erhaltenen Bindungen erfaßt werden. Solange die linke Maustaste gedrückt ist, können diese Bindungen über den Bildschirm gezogen werden. Über einem mit *initDrop* initialisierten Objekt können die gezogenen Bindungen fallengelassen und mit der als Parameter übergebenen Funktion *setBindings* in die Bildschirmrepräsentation eingefügt werden. Ein mit *initDrop* initialisiertes Objekt sollte, wenn es vom Bildschirm gelöscht wird, mit *uninitDrop* wieder deaktiviert werden.

7.5 Visualisierungsschnittstelle

Die Visualisierungsschnittstelle hat die Aufgabe, aus den in der Strukturanalyse erhaltenen typ– bzw. wertspezifischen Informationen eines Objektes O mit Hilfe der im letzten Abschnitt beschriebenen unterstützenden Schnittstellen ein Protokoll zu generieren, das eine entsprechende Bildschirmrepräsentation $bRep_O$ realisiert. Dazu existieren zu allen in Abschnitt 4.2 vorgestellten Bildschirmrepräsentationen Generierungsfunktionen, die zur besseren Strukturierung dieser Schnittstelle auf verschiedene Module verteilt werden. Allen diesen Funktionen ist gemein, daß sie eine Funktion *getValue* als Parameter erhalten, die als Ergebnis den aktuellen Wert einer Bindung zurückliefert. Diese Funktion wird jeweils der Protokollfunktion *getValue* zugewiesen. Ansonsten kann grundsätzlich zwischen der Visualisierung von Typ– und Wertbindungen unterschieden werden, so daß deren Generierungsfunktionen in getrennten Abschnitten vorgestellt werden. Da die Visualisierung von Funktionen komplexer ist als die anderer Werte, wird sie in einem eigenen Abschnitt behandelt.

7.5.1 Typvisualisierung

Gemeinsamkeiten bei der Visualisierung von Typen ergeben sich dadurch, daß Typbindungen im Gegensatz zu Wertbindungen nicht veränderlich sein können. Aus diesem Grund ist es nicht erlaubt, eine Bindung aus dem Klemmbrett in die Bildschirmrepräsentation zu kopieren, so daß der Menüeintrag *Paste* nicht generiert werden muß. Das Fallenlassen einer anderen Bindung über der Bildschirmrepräsentation ist ebenfalls verboten, während das Kopieren von Typbindungen in ein Ziel–*Environment* erlaubt ist. Um ein destruktives Verändern durch den Programmierer zu verhindern, löst die Funktion *setValue* des abstrakten Datenstrukturmodells jeweils ein Ausnahme aus. Schließlich gibt die Funktion *apply* ausschließlich den Wert *true* zurück, da sich die Werte in Bildschirmrepräsentationen nicht von denen der Laufzeittyprepräsentation unterscheiden können.

Die Generierung der anderen Komponenten des Protokolls ist dagegen typspezifisch und erfolgt mit Hilfe der StarView Klassenbibliothek. Die verschiedenen Module und Funktionen, die die Protokolle von Typen generieren, sind in Tabelle 7.2 dargestellt.

Unterstützt werden diese Funktionen durch ein weiteres Modul *typeComponent*, das für Signaturen Operationen zum Erzeugen und Manipulieren von Bildschirmrepräsentationen bereitstellt.

7.5.2 Wertvisualisierung

Die Module und Funktionen, die aus den wertspezifischen Informationen die Protokolle mit entsprechenden Bildschirmrepräsentationen zur Visualisierung von Werten generieren, sind in Tabelle 7.3 dargestellt.

Modul	**Funktionsname**	**Typen der visualisierte Werte**
simpleValues	newOk	**Ok**, abstrakte Datentypen
	newBase	Basistypen
aggregateValues	newRecord	Tupel, Rekords
	newTupleCase	variante Tupel
	newException	Ausnahmetypen
arrayValues	new	Felder

Tabelle 7.3: Funktionen zur Visualisierung von Werten

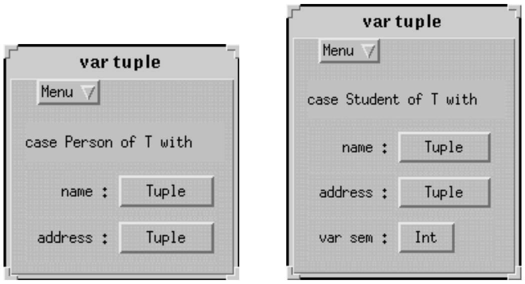
Analog zur Visualisierung von Signaturen gibt es ein Modul *valueComponent*, das Operationen zum Erzeugen und Manipulieren von Bindungen bereitstellt. Bei der Wertvisualisierung werden zwei Aspekte genauer betrachtet und erläutert — die Darstellung bestimmter Komponenten direkt in anderen Bildschirmrepräsentationen und die Manipulation variabler Wertbindungen durch Datenaustausch zwischen Bildschirmrepräsentationen.

Eingebettete Bildschirmrepräsentationen

Konstante Wertbindungen von Basistypen werden als Komponenten anderer Wertbindungen aus Gründen der Übersicht nicht als eigenständige Objekte auf dem Bildschirm visualisiert, sondern in deren Bildschirmrepräsentation integriert (s. Abschnitt 4.2.3). Damit können Komponenten außer durch Knöpfe auch in Form eines Protokolles repräsentiert werden, das deren Bildschirmrepräsentation realisiert. Das ist die einzige Form, in der es eine Abhängigkeit zwischen Bildschirmrepräsentationen gibt. Entsprechend müssen die Protokollfunktionen des Elternobjektes auch jedesmal die entsprechenden Protokollfunktionen der Kindobjekte aufrufen.

Manipulation von Werten durch Datenaustausch

Um den Wert variabler Wertbindungen verändern zu können, wird jeder Generierungsfunktion eine Funktion *setValue* als Parameter übergeben, die jedoch nicht den Inhalt der Bildschirmrepräsentation entsprechend anpaßt. Diese muß also bei einem Datenaustausch per Klemmbrett



case Person of T with

name : Tuple

address : Tuple

case Student of T with

name : Tuple

address : Tuple

var sem : Int

Abbildung 7.4: Beispiel für die Manipulation varianter Tupel

oder *Drag&Drop* innerhalb der Protokollfunktion *setValue* aktualisiert werden. Bei variablen Wertbindungen der Basistypen erfordert dies nur einen Aufruf der Funktion *refresh*. Bei strukturierten Werten ist zu unterscheiden, ob deren Komponenten als Knopf dargestellt sind oder aus einer Bildschirmrepräsentation bestehen. Im ersten Fall ist die Aktion, die beim Drücken des Knopfes ausgelöst wird, dahingehend zu ändern, daß sie den neuen Wert visualisiert. Durch Drücken des Knopfes wird dann die Bildschirmrepräsentation des neuen Wertes entweder erzeugt, falls diese noch nicht auf dem Bildschirm sichtbar ist, oder hervorgehoben. Im zweiten Fall ist, falls sich der Wert geändert hat, die alte Bildschirmrepräsentation zu löschen sowie eine neue zu generieren und im Elternobjekt anzuzeigen.

Bei varianten Tupeln, bei denen sich durch die Manipulation die dargestellte Variante ändert (s. Abb. 7.4), und bei Feldern ist zusätzlich zu beachten, daß sich die Anzahl der Elemente ändern kann. In beiden Fällen ist eine komplett neue Bildschirmrepräsentation zu generieren. Um den Aufwand bei varianten Tupeln zu verringern, wird für eine Variante die einmal erzeugte Bildschirmrepräsentation gespeichert und wiederverwendet, falls der Wert später wieder diese Variante annimmt.

7.5.3 Funktionsvisualisierung

Die bei der Visualisierung von Funktionen generierten Protokollfunktionen verwalten die Bildschirmrepräsentation der Funktionssignatur, die sich nicht von der von Funktionstypen unterscheidet, sowie den Knopf, mit dem eine Funktion interaktiv ausgeführt werden kann. Im ersten Unterabschnitt wird erläutert, wie die interaktive Funktionsapplikation realisiert werden kann.

Wie in Abschnitt 4.2.4 bereits erläutert, sind für das Verständnis einer Funktion auch der Programmkode und die im Funktionsrumpf referenzierten globalen Bindungen wichtig. Diese Informationen, wie auch der TML Kode und der TVM Kode, erhält man nicht durch die Strukturanalyse, sondern aus den Komponenten des Funktionsabschlusses (s. Abb. 6.2). Zur Visualisierung dieser Komponenten, die in den anderen beiden Unterabschnitten beschrieben werden, werden Funktionen generiert, die mit den entsprechenden Subeinträgen des Menüs der Funktionssignatur verknüpft und bei deren Auswahl ausgeführt werden.

Interaktive Funktionsapplikation

Die aktuelle Implementierung der Funktionsvisualisierung erlaubt nur die Ausführung parameterloser Funktionen. Dafür wird zu der Funktionssignatur noch ein Knopf mit der Aufschrift *execute* generiert und angezeigt. Drückt der Anwender diesen Knopf, so wird die Funktionsdefinition in ein allen Funktionen gemeinsames *Environment* eingetragen, da aus der Funktionsdefinition nicht ersichtlich ist, in welchem *Environment* sie gebunden ist. Mit Hilfe der Funktion *eval* der reflektiven Compilerschnittstelle des Tycoon Systems [Geisler 95] kann die Funktionsapplikation in diesem *Environment* übersetzt und ausgeführt werden. Das Ergebnis wird als anonyme Bindung visualisiert.

Soll diese Funktionalität auch für nichtparameterlose Funktionen angeboten werden, so müssen vor dem Aufruf von *eval* die Parameter bestimmt werden. Da Parameter aber eine polymorphe Typvariable als Typ haben oder Funktionen sein können, ist deren Eingabe über eine direkte Manipulation von mit Nullwerten gefüllten Bildschirmrepräsentationen nicht möglich. Stattdessen könnte die Funktionsdefinition in das gemeinsame *Environment* eingetragen werden, in das der Anwender ebenfalls die Bindungen kopiert, die der Funktion als Parameter übergeben werden sollen. In einer weiteren Bildschirmrepräsentation müßte für jeden Parameter ein Eingabefeld für Zeichenketten generiert werden, in die der Anwender die Namen der Bindungen den Funktionsparametern zuweisen kann. Nachdem die Eingabe abgeschlossen ist, können diese Namen aus der Bildschirmrepräsentation gelesen und eine Zeichenkette generiert werden, die den Funktionsaufruf repräsentiert. Diese wird zusammen mit dem gemeinsamen *Environment* der Funktion *eval* übergeben, die den Funktionsaufruf übersetzt und ausführt.

Visualisierung globaler Bindungen

Die Visualisierung globaler Bindungen ist nur möglich, wenn von deren Typen Laufzeittyprepräsentationen wie in Abschnitt 6.1 beschrieben generiert und in den Funktionsabschluß eingetragen worden sind. Ansonsten wird eine entsprechende Fehlermeldung erzeugt und auf dem Bildschirm angezeigt.

Mit Hilfe der entsprechenden Funktionen der Schnittstelle *Closure* werden die globalen Bindungen mitsamt ihren Laufzeittyprepräsentationen aus dem Funktionsabschluß gelesen und in eine Liste von Bindungen verwandelt. Zur Verwaltung dieser Liste wird ein Protokoll generiert, das analog zu Bindungen strukturierter Werte die Wertkomponente konstanter Wertbindungen von Basistypen direkt und die aller anderen Bindungen über einen Knopf darstellt. Die Bildschirmrepräsentation der Liste der globalen Bindungen wird in einen eigenen Rahmen eingebettet mit einem Menü, das nur den Eintrag *Quit* erhält. Da es sich bei der Visualisierung globaler Bindungen nicht um die Darstellung eines Wertes handelt, ist es nicht erlaubt, andere Objekte in die Bildschirmrepräsentation zu kopieren. Das generierte Protokoll wird mit dem Protokoll der Funktion verknüpft, damit eine Bildschirmrepräsentation der Liste der globalen Bindungen nicht unabhängig von der Bildschirmrepräsentation der Funktion existieren kann.

Visualisierung der verschiedenen Koderepräsentationen

Die verschiedenen Koderepräsentationen werden in mehrzeiligen, nicht manipulierbaren Editierfeldern visualisiert. Diese Editierfelder werden allerdings nicht von dem in Abschnitt 7.3.1 beschriebenen Protokoll verwaltet, da sich ihre Funktionalität zu sehr von den anderen Bildschirmrepräsentationen unterscheidet und da sie mit diesen nicht kombinierbar sein müssen.

Von Funktionen, die auf dem *Toplevel* definiert werden, wird keine persistente Repräsentation des Programmkodes erzeugt, so daß dessen Visualisierung nicht möglich ist. Wird eine

Funktion dagegen in einem Modul definiert, so kann man mit Hilfe des Dateinamens, der in ihrem Literalvektor gespeichert ist, den Programmkode aus der entsprechenden Datei lesen, sofern diese noch existiert. Da jedoch der Programmkode in einer Datei unabhängig von seiner ausführbaren TVM Repräsentation im persistenten Speicher existiert, ist nicht sichergestellt, daß der Programmkode in der Datei noch mit dem übereinstimmt, den die Funktion ausführt.

Um den Kontext, in dem eine Funktion definiert ist, verständlich zu machen, wird jeweils der Programmkode des gesamten Moduls und nicht nur der Funktion visualisiert. Da jedoch verschiedene Funktionen im gleichen Modul definiert sein können, ist deren Bildschirmrepräsentation im Gegensatz zu den Bildschirmrepräsentationen der globalen Bindungen und der anderen Koderepräsentationen nicht von der Existenz einer Bildschirmrepräsentation einer bestimmten Funktion abhängig. Um den Programmkode eines Moduls nicht mehrmals auf dem Bildschirm zu visualisieren, müssen alle Module analog zu den visualisierten Objekten in einer durch die Schnittstelle *BrowserDict* verwalteten Tabelle gespeichert werden, deren Schlüssel der Dateiname ist. Nur vom Programmkode noch nicht visualisierter Module wird eine Bildschirmrepräsentation erzeugt.

Bei der Übersetzung von Funktionen kann optional vom TML Kode eine persistente Repräsentation (PTML) erzeugt und in den Literalvektor eingetragen werden. Nur in diesem Fall ist es möglich, den TML Kode einer Funktion darzustellen. Dazu muß der PTML Kode zunächst in die TML Repräsentation und danach in eine formatierte Zeichenkette umgewandelt und auf dem Bildschirm visualisiert werden.

Von jeder übersetzten Funktion wird der TVM Kode vom Literalvektor der Funktion referenziert. Bei dessen Visualisierung wird er disassembliert, in eine formatierte Zeichenkette transformiert und auf dem Bildschirm dargestellt.

7.6 Schnittstelle zum Anwendungsprogramm

Die Details der Implementierung und das Protokoll werden vor dem Anwendungsprogrammierer versteckt. Nur über die Schnittstelle *Browser* (s. Anhang C.2) wird ihm der Dienst zur Daten– und Funktionsvisualisierung zugänglich gemacht. Sie ermöglicht die Visualisierung von Bindungen und exportiert die Protokollfunktionen, die den Zugriff auf die Bildschirmrepräsentationen ermöglichen.

Die Visualisierung von Bindungen erfolgt durch die Funktion *binding*, die außer der darzustellenden Bindung noch ein Fenster, das als Elternfenster der zu generierenden Bildschirmrepräsentation verwendet wird und keinen Nullwert haben darf, und einen boole'schen Wert *keepPermanent*, mit dem die entsprechende Komponente des Protokolls gesetzt wird, als Parameter erwartet. Für die Bindung werden ein Protokoll, das als Ergebnis zurückgegeben wird, und die Bildschirmrepräsentation erzeugt.

Da die Daten– und Funktionsvisualisierung als generischer Dienst von anderen Diensten genutzt werden soll, wird davon ausgegangen, daß es bereits eine Hauptanwendung und damit insbesondere ein Anwendungsfenster gibt. Nur wenn dieses Anwendungsfenster auf dem Bildschirm sichtbar ist, wird auch die Bildschirmrepräsentation der entsprechenden Bindung durch StarView angezeigt.

Als Ergebnis der Visualisierung wird das Protokoll zurückgegeben, auf das der Anwendungsprogrammierer über die ebenfalls exportierten Protokollfunktionen zugreifen kann. Allerdings ist der Zugriff auf die Protokollfunktionen *create* und *show* nicht möglich, da eine Bildschirmrepräsentation mit Hilfe der Funktion *display* erzeugt und dargestellt werden soll, die zusätzlich die Generierung eines Rahmens, eines Menüs und gegebenenfalls der Manipulationsknöpfe realisiert.

Wie in Abschnitt 7.1 erwähnt, wird der Objektspeicher durch *Environments*, die in Tycoon Werte erster Klasse sind, strukturiert. Diese sollen jedoch nicht anhand ihrer Datenstruktur, sondern entsprechend ihrer Semantik visualisiert werden. Deshalb ist in [Geisler 95] eine Schnittstelle zur Visualisierung von *Environments* — ein *Environment Browser* — entwickelt worden. Im Rahmen dieser Arbeit wurde diese Schnittstelle in die Daten– und Funktionsvisualisierung integriert. Dadurch ist eine Anbindung an die Tycoon Programmierwerkbank möglich, die Werkzeuge zur Programmierunterstützung in einem gemeinsamen Rahmen integriert.

Bei der Integration der beiden Dienste ist eine Anpassung der Implementierung des *Environment Browsers* an die Strukturen des Dienstes zur Daten– und Funktionsvisualisierung notwendig. Dazu werden, wie bei allen anderen Objekten auch, die objektspezifischen Eigenschaften durch das allgemeingültige Protokoll gekapselt, so daß der Zugriff auf einen *Environment Browser* nur noch über die Protokollzugriffsfunktionen möglich ist. Außerdem erfolgt dadurch eine Vereinheitlichung des Datenaustauschs über ein gemeinsames Klemmbrett und einen gemeinsamen *Drag Server* für Bindungen und *Environments*. Um die Manipulation des Objektspeicherinhalts anhand seiner *Environmentstruktur* durch direkte Interaktion mit dem *Environment Browser* zu integrieren, wird ein zusätzlicher Menüknopf *Bindings* eingeführt, der Menüeinträge *Cut*, *Copy* und *Paste* enthält. Außerdem wird die textuelle Darstellung von Bindungen des *Environments*, die keine direkte Manipulation der Bindungen eines *Environments* erlaubte, durch die Visualisierung mit Hilfe des Dienstes zur Daten– und Funktionsvisualisierung ersetzt.

8. Zusammenfassung und Ausblick

Die Motivation für die Themenstellung dieser Arbeit bestand darin, eine generische grafische Daten– und Funktionsvisualisierung in einer persistenten Programmierumgebung zu entwickeln, die — integriert in die in [Geisler 95] vorgestellte grafische reflektive Programmierumgebung — den gesamten Prozeß der Softwareentwicklung unterstützt. Die wesentlichen Beiträge zur Realisierung dieses Zieles sind:

1. Die Steigerung der Generik des Tycoon Systems durch die Einführung von Laufzeittyprepräsentationen und automorphen Werten.
2. Die Unterstützung der Visualisierung von Funktionen durch Übersetzungszeitreflektion.
3. Die grafische Visualisierung beliebiger Daten und die Manipulation variabler Wertbindungen anhand ihrer Bildschirmrepräsentationen.

8.1 Dynamische Typisierung

Die Einführung von **automorphen Werten** und **Laufzeittyprepräsentationen** erweitert die generischen Fähigkeiten des Tycoon Systems um die Klasse der *typgesteuerten* Berechnungen, die bisher nur typunsicher innerhalb des TL Compilers realisiert werden konnten. Im Gegensatz zu Anwendungen des parametrischen Polymorphismus und des Subtyppolymorphismus, in denen von den Typen abstrahiert werden kann, handelt es sich dabei um typabhängige Berechnungen. Da bei diesen Anwendungen des ad–hoc Polymorphismus der Typ der zu bearbeitenden Werte statisch nicht bekannt ist, müssen die Typinformationen dieser Werte zur Laufzeit verfügbar sein.

Die beiden wesentlichen Eigenschaften des im Rahmen dieser Arbeit vorgestellten Konzeptes zur dynamischen Typisierung sind die Einführung von automorphen Werten und Laufzeittyprepräsentationen als **Werte erster Klasse** in TL sowie die Möglichkeit der **Manipulation** und der **Inspektion** von Werten mit dynamischer Typinformation. Dadurch können einserseits Anwendungen realisiert werden, die nur Laufzeittyprepräsentationen, nicht aber automorphe Werte benötigen, wie dies bei der generischen Typvisualisierung der Fall ist. Andererseits werden beide Anwendungsklassen dynamischer Typisierung unterstützt:

- ▹ Anwendungen wie der entfernte Prozeduraufruf in verteilten Umgebungen und der reflektive Aufruf des Übersetzers erfordern die Manipulation von Werten mit dynamischer Typinformation, da sie Werte miteinander verbinden, die in unterschiedlichen Kontexten erzeugt worden sind.

- Die Fähigkeit zur Inspektion von automorphen Werten und Laufzeittyprepräsentationen ermöglicht die Entwicklung hochgenerischer, datenstrukturabhängiger Algorithmen wie der generischen Datenvisualisierung, die abhängig von der Typstruktur eines zu visualisierenden Wertes eine geeignete Bildschirmrepräsentation generiert.

8.2 Übersetzungszeitreflektive Unterstützung der Visualisierung

Um **globale Bindungen von Funktionen**, die in TL Werte erste Klasse sind, zu visualisieren, ist zusätzliche Systemunterstützung notwendig, da bei der Erzeugung automorpher Werte von Funktionswerten keine Laufzeittyprepräsentationen der globalen Bindungen erzeugt werden. Aufbauend auf den Spracherweiterungen zur dynamischen Typisierung wird ein übersetzungszeitreflektiver Ansatz zur Unterstützung der Visualisierung globaler Bindungen mit folgenden Eigenschaften entwickelt:

- Unter Kenntnis der Repräsentation von Funktionen im Objektspeicher wird für den Anwendungsprogrammierer die Schnittstelle *Closure* bereitgestellt, die für globale Bindungen assoziiert mit den Laufzeittyprepräsentationen des Typs ihrer Wertkomponente die typsichere Registrierung in und das typsichere Lesen aus dem erweiterten Funktionsabschluß ermöglicht.
- Der TL Typüberprüfer wird erweitert, so daß, ohne die Semantik des Programmes zu verändern, zur Übersetzungszeit Aufrufe der Registrierungsfunktionen der Schnittstelle *Closure* generiert werden, die zur Laufzeit alle globalen Bindungen mit den vom Typüberprüfer erzeugten Laufzeittyprepräsentationen in den Funktionsabschluß eintragen.

Dieser Ansatz nutzt die reflektiven Fähigkeiten persistenter Programmierumgebungen aus, in denen die Programmkonstruktionsumgebung der Laufzeitumgebung entspricht. Dadurch werden grundlegende Änderungen am *Backend* des Compilers vermieden. Da die generierten Funktionsaufrufe, die die globalen Bindungen mit den Laufzeittyprepräsentationen in den Funktionsabschluß einer Funktion f eintragen, nur einmal während der Bindung von f ausgeführt werden, erhöht sich auch die Laufzeit der veränderten Programme kaum. Nur die Übersetzungszeit von Funktionen steigt linear etwa um den Faktor 2,5, wobei der Programmierer durch die Einstellung eines Schalters bestimmen kann, ob für eine Funktion die globalen Bindungen mit den Laufzeittyprepräsentationen registriert werden sollen.

Zur Visualisierung des Zustandes **persistenter Threads**, die in TL ebenfalls Werte erster Klasse sind, wird eine Erweiterung des übersetzungszeitreflektiven Ansatzes vorgestellt. Dazu wird ebenfalls der jeweils aktuelle **lokale Kontext** einer sich in der Ausführung befindlichen Funktion innerhalb eines persistenten Threads in den erweiterten Funktionsabschluß eingetragen, auf den der Anwender zur Laufzeit zugreifen kann. Dadurch wird nicht nur die Visualisierung persistenter Threads ermöglicht, sondern auch die Entwicklung eines Werkzeuges zur Fehlersuche (*Debugger*) unterstützt.

8.3 Datenvisualisierung und Datenmanipulation

Um den Softwareentwicklungsprozeß grafisch zu unterstützen, wird ein Konzept zur **Objektspeichervisualisierung und –manipulation** vorgestellt. Unter Verwendung dynamischer

Typisierung werden für beliebige Werte des Objektspeichers entsprechend ihrer Struktur automatisch Bildschirmrepräsentationen generiert. Indem die Fensteridentität der visualisierten Objekte äquivalent zu ihrer Objektidentität ist, d.h. indem mit Ausnahme konstanter Wertbindungen in strukturierten Werten zu jedem Objekt des Objektspeichers nur eine Bildschirmrepräsentation erzeugt wird, werden die Strukturen der visualisierten Objekte und die Beziehungen zwischen den Objekten für den Programmierer anschaulich dargestellt. In Verbindung mit der in [Geisler 95] vorgestellten Visualisierung von *Environments*, kann der Programmierer so die Topologie des Objektspeichers erkunden, indem er ausgehend von einem persistenten Wurzelobjekt anhand der dargestellten Strukturen durch den Objektspeicher navigiert. Die Visualisierung von Typen erhöht das Verständnis der dargestellten Werte zusätzlich.

Die direkte **Manipulation** von Werten über ihre Bildschirmrepräsentation wird, entsprechend der Funktionalität grafischer Benutzerschnittstellen, durch Tastatureingaben sowie durch den Datenaustausch zwischen zwei Bildschirmrepräsentationen über eine Zwischenablage und das *Drag&Drop* Verfahren unterstützt. Dadurch werden dem Programmierer die Veränderung variabler Wertbindungen und die Umstrukturierung des Objektspeichers durch Kopieren von Bindungen zwischen Environments erleichtert, die ansonsten die textuelle Eingabe teilweise komplizierter und fehlerträchtiger TL Ausdrücke erfordert.

Ist der Funktionsabschluß um die mit ihren Laufzeittyprepräsentationen assoziierten globalen Bindungen erweitert, so können alle für das Verständnis einer **Funktion** notwendigen Informationen, d.h. die Signatur, der Funktionsrumpf und die globalen Bindungen, **visualisiert** werden. Die — aufbauend auf dynamischer Typisierung — in [Geisler 95] vorgestellte laufzeitreflektive Schnittstelle zum Compiler wird verwendet, um für parameterlose Funktionen zusätzlich den **interaktiven Funktionsaufruf** zu unterstützen. Dadurch erhält der Programmierer die Möglichkeit, die Funktionsweise parameterloser Funktionen besser zu verstehen, indem er durch Manipulation der globalen Variablen das Ergebnis eines Funktionsaufrufes beeinflußen kann.

8.4 Ausblick

Abschließend werden in diesem Abschnitt die bereits in den einzelnen Kapiteln diskutierten offenen Punkte zusammengefaßt und ein Ausblick auf zukünftige Arbeiten bezüglich der generischen Datenvisualisierung in Tycoon gegeben.

Bei Laufzeittyprepräsentationen hat sich die Repräsentation durch direkte Objektreferenzen gegenüber der im Typüberprüfer gewählten *de Bruijn* Notation als vorteilhaft erwiesen. Um die laufzeitintensive Transformation zwischen statischen und dynamischen Typrepräsentationen zu vermeiden, ist ein Typüberprüfer zu entwickeln, der Typen ebenfalls durch direkte Objektreferenzen repräsentiert. Aufbauend auf solch einer Implementierung, die im Rahmen von [Bremer 96] realisiert wird, ist die Generierung von Laufzeittyprepräsentationen entsprechend anzupassen.

Ein zweites Problem bezüglich dynamischer Typisierung entsteht durch die zusätzliche Speicherung übersetzter Module im Dateisystem, wodurch die arbeitsteilige Anwendungsentwicklung mit mehreren unabhängigen Objektspeichern gewährleistet wird. Da im Objektspeicher alle gebundenen Objekte transitiv von der Subtyptestfunktion des Compilers abhängen, wird für jedes Modul, in dem die Operation *dynamic_be* verwendet wird, der gesamte Objektspeicher mit in das Dateisystem herausgeschrieben. Zur Lösung dieses Problems wäre zu untersuchen, inwieweit sich die in [Mathiske et al. 95b] beschriebene Methode eignet, um die Funktionen des Compilers, die in der Regel in jedem Objektspeicher vorhanden sind, als ubiquitär zu kennzeichnen. Dadurch wird auch die Subtyptestfunktion zusammen mit allen transitiv referenzierten Objekten beim Herausschreiben auf das Dateisystem abgeschnitten. Beim Hereinlesen werden die Referenzen auf die entsprechende, im Objektspeicher vorhandene Subtyptestfunktion wiederhergestellt.

Bei der **Funktionsvisualisierung** existieren noch zwei offene Fragen bezüglich des interaktiven Aufrufes von Funktionen, die die Übergabe von Parametern erwarten, und der Visualisierung des Programmtextes. Beim interaktiven Aufruf parametrisierter Funktionen ist das Problem der Parameterübergabe zu lösen. In Abschnitt 7.5.3 ist dazu ein Ansatz vorgestellt worden, indem die Parameter in das *Environment* kopiert werden, in dem der Funktionsaufruf übersetzt, gebunden und ausgeführt werden soll, und in einer Editiermaske deren Namen den Funktionsparametern zugewiesen werden.

Die Visualisierung des Programmtextes ist für auf dem interaktiven *TopLevel* definierte Funktionen unmöglich, da der Programmtext nicht im Funktionsabschluß gespeichert ist. Nur für Funktionen, die in einem Modul definiert sind, kann der Programmtext anhand des im Dateisystem gespeicherten Moduls visualisiert werden. Dabei kann allerdings nicht überprüft werden, ob der Programmtext im Modul noch mit dem von der Funktion tatsächlich ausgeführten übereinstimmt. Zur Lösung beider Probleme ist die Möglichkeit der Speicherung des Programmtextes im Funktionsabschluß zu untersuchen. Zu klären ist dabei die Handhabung des Programmtextes von in Modulen definierten Funktionen. In Verbindung mit dem in [Kornacker 95] eingeführten Konzept persistenter Sicherungspunkte kann dadurch ein weiterer Beitrag zur Realisierung einer voll integrierten persistenten Programmierumgebung geleistet werden, indem der Programmierer direkt den visualisierten Funktionstext editieren, übersetzen, binden und testen kann. Stellt er aber fest, daß er die Änderungen wieder rückgängig machen möchte, so kann er durch ein Zurücksetzen des Systemzustandes (*Rollback*) den ursprünglichen Zustand der Funktion wiederherstellen.

Da die Generierung der Bildschirmrepräsentationen automatisch entsprechend der Struktur der Werte erfolgt, kann die **Semantik** der zu visualisierenden Werte nicht berücksichtigt werden. Es gibt jedoch Situationen, wie z.B. bei den Massendatentypen, die in Tycoon durch externe Bibliotheken und nicht innerhalb des Typsystems von TL realisiert sind [Matthes, Schmidt 91], in denen die Berücksichtigung semantischer Informationen das Verständnis der visualisierten Daten erhöht. Eine Lösung dieses Problems besteht darin, für bestimmte Datentypen der Standardbibliothek, wie dies z.B. bei *Environments* erfolgt ist, eine die Semantik berücksichtigende Visualisierung zu realisieren und in die Bibliothek zur Datenvisualisierung zu integrieren. Alternativ ist die Entwicklung eines Werkzeuges wie ToonMaker für O_2 [Borras et al. 92] zu untersuchen, das es dem Anwender erlaubt, für beliebige Typen Präsentationsschablonen zu erzeugen, die zur Visualisierung von Werten dieses Typs wiederverwendet werden können.

Die Visualisierung **persistenter Threads**, die einen weiteren wichtigen Beitrag zum Verständnis der Inhalte und der Topologie des Objektspeichers sowie der im Informationssystem ablaufenden Prozesse darstellt, ist im Rahmen dieser Arbeit nicht realisiert worden. Da persistente Threads in TL als Werte erster Klasse in Form abstrakter Datentypen repräsentiert sind, erfordert ihre Visualisierung ebenfalls die Berücksichtigung semantischer Informationen, um die in einem persistenten Thread aktiven Funktionen, ihre Signatur, ihren Rumpf, ihre globalen Bindungen und ihren aktuell gültigen lokalen Kontext zu visualisieren. In Abschnitt 6.2 wird ein übersetzungszeitreflektiver Ansatz zur Visualisierung des lokalen Kontextes einer Funktion vorgestellt, der aber bisher nicht implementiert worden ist. Zur Realisierung dieses

Ansatzes müssen für jeden Wertkonstruktor in TL Regeln zur Generierung des entsprechenden Programmkodes definiert werden. Dabei ist zu untersuchen, welchen Einfluß die Erweiterung des Programmkodes auf das Übersetzungs– und das Laufzeitverhalten der Funktionen sowie auf die Komplexität des Typüberprüfers hat.

A. Schnittstellen des Compilers zur Implementierung dynamischen Typisierung

A.1 Schnittstelle 'TLDynamicImpl'

```
Interface TLDynamicImpl
import :TLType :TLIde tlEnv list time tm :Source
export
  Let Path = Tuple var oid :String end
  Let Rec T <:Ok = Tuple                  (* Laufzeit–Typrepräsentation *)
    case okCase, nokCase with
    case baseCase with                    (* Basistyp *)
      name :String
    case ideCase with                     (* Typbezeichner *)
      definition :Signature               (* direkte Referenz zur Definition *)
    case adtCase with                     (* abstrakter Typbezeichner *)
      definition :Signature               (* direkte Referenz zur Definition *)
      path :Tuple var oid :String end
    case funCase with                     (* Funktionstyp *)
      signatures :list.T(Signature)
      range :T
    case tupleCase with                   (* Tupeltyp *)
      signatures :list.T(Signature)
      cases :list.T(Case)
    case recordCase with                  (* Recordtyp *)
      signatures :list.T(Signature)
    case exceptionCase with               (* Ausnahmetyp *)
      signatures :list.T(Signature)
    case operCase with                    (* Typoperator *)
      signatures :list.T(Signature)
      range :T
    case arrayCase with                   (* Arraytyp *)
      elementType :T
    case appCase with                     (* Typoperatorapplikation *)
      oper :T
      arguments :list.T(Signature)
    case recCase with                     (* rekursiver Typ *)
      bindingIndex :Int
      typeBindings :list.T(Signature)
    case varCase with                    (* Typ einer veränderlichen Bindung *)
      type :T
  end

  and Signature <:Ok = Tuple
    ide :Tuple name :String  pos :Source.Position  end
    var type :T
    case typeCase, typeEqualCase with  (* T <:A  bzw. Let T = A *)
      typeId :time.T                   (* Zeitstempel *)
    case valueCase
    case locationCase
  end

  and Case <:Ok = Tuple
    label :String
    signatures :list.T(Signature)        (* zusätzliche Felder dieser Variante *)
  end
  Let Value = Tuple              (* automorphe Werte *)
    value :Ok
    type :T
  end
  makeType(t :T var outEnv :tlEnv.T) :TLType.T
  (* transformiert t in einen  statischen Typ *)

  isSubType(small, big :T  env :tlEnv.T) :Bool

  coerce(A <:Ok  d :Value  t :T  exc :Exception  pos :Souerce.Position) :A
  (* Liefert d.value wenn isSubType(d.type t), ansonsten coerceError *)

  isADTRef(ref :String) :Bool
  (* Liefert true wenn ref einen durch dieses Modul erzeugten ADT repräsentiert *)

  getPath(ref :String) :String
  (* Liefert die Pfadkomponente des von diesem Modul erzeugten ref *)

end
```

A.2 Schnittstelle 'TLDynamic'

```
Interface TLDynamic
import :TLValue :TLType tlEnv :TLDynamicImpl dictionary
export
  Let Entry = Tuple
    path :TLDynamicImpl.Path
    value :TLValue.T
  end
  makeTypeRep(type :TLType.T  env :tlEnv.T
  pathes :dictionary.T(Entry String)) :TLDynamicImpl.T
  (* Zurückgegebene Pfade müssen noch an patchTypeRep übergeben werden! *)
  patchTypeRep(A <:Ok  paths :Array(TLDynamicImpl.Path)
       modules :Array(Ok) t :A) :A
  (* ersetzt Platzhalter in paths durch die aktuellen Werte aus modules *)
  makeDynType(type :TLType.T env :tlEnv.T) :TLValue.T
  (* implementiert typeRep_new *)
  makeDynValue(value :TLValue.T  type :TLType.T  env :TLEnv.T) :TLValue.T
  (* implementiert dynamic_new *)
  makeCoerce(dyn :TLValue.T type :TLType.T env :tlEnv.T) :TLValue.T
  (* implementiert dynamic_be *)
end
```

A.3 Schnittstelle 'TLDynEnv'

```
Interface TLDynEnv
import :TLIde tlEnv :TLDynamicImpl
export
  error :Exception
  T <:Ok
  (* Eine dynamische Umgebung ist eine geordnete Folge von Typbindungen *)
  dynEnv() :T
  (* Liefert die globale dynamische Umgebung. Es wird beim Erzeugen eines neuen
     Objektspeichers im Root Vektor abgelegt. Laufzeit–Typrepräsentation enthalten Referenzen
     auf die Bindungen dieser Umgebung *)
  init(env :tlEnv.T) :Ok
  (* Initialisiert 'theEnv' mit Builtin Typen. Wird nach der Systemerzeugung aufgerufen *)
  initReflectively() :Ok
  (* Initialisiert 'theEnv' mit der dynamischen Umgebung aus dem Root Vektor.
     Liefert Fehlermeldung wenn es nicht gefunden wird. *)
  insert(sig :TLDynaicImpl.Signature) :Ok
  (* Fügt sig in 'theEnv' ein. Liefert absoluten Index in 'theEnv'.
     Erzeugt Fehlermeldung, wenn sig schon in 'theEnv' vorhanden ist. *)
  get(ide :TLIde.T) :TLDynamicImpl.Signature
  (* Liefert Signatur mit Schlüssel 'ide'. Erzeugt Fehlermeldung, wenn nicht gefunden. *)
end
```

B. Reflektive Schnittstellen

B.1 Schnittstelle 'TypeRep'

```
interface TypeRep
import list time  :Iter  :Source
export
  error :Exception
  Let T = dynamic_T
  Let Ide = Tuple name :String pos :Source.Position end

  Let Signature = Tuple
    ide :Ide
    type :T
    case typeCase, typeEqualCase with
       typeId :time.T
    case valueCase
    case locationCase
  end

  Let Case = Tuple      (* zusätzliche für diesen case gültige Felder*)
    label :String
    signatures = list.T(Signature)
  end

  Let Expansion = Tuple
    case okCase, nokCase with
    case baseCase with
      ide :Ide
    case ideCase with
      ref :String
      definition :Signature
    case adtCase with
      ref :String
      definition :Signature
      path :Tuple oid :String end
    case funCase with
      signatures :list.T(Signature)
      range :T
    case tupleCase with
      signatures :list.T(Signature)
      cases :list.T(Case)
    case recordCase with
    signatures :list.T(Signature)
  case exceptionCase with
    signatures :list.T(Signature)
  case operCase with
    signatures :list.T(Signature)
    range :T
  case arrayCase with
    elementType :T
  case appCase with
    oper :T
    arguments:list.T(Signature)
  case recCase with
    bindingIndex :Int
    typeBindings :list.T(Signature)
    type :T
    bound :T
end (* beachte: TLDynamicImpl.T <:Expansion *)

okType, int, real, bool, char, string :T

isSubType(small, big :T) :Bool
(* Liefert true wenn die Laufzeit–Typrepräsentationen in der Subtypbeziehung
   small <:big stehen. *)

inspect(type :T):Expansion
(* Liefert detailiertere Beschreibung von 'type' *)

exposed(type :T) :T
(* Entfernt definition(s) von 'type' *)

equal(t1, t2 :T) :Bool
(* Liefert true wenn t1 und t2 strukturell äquivalent sind *)

signatures(t :T) :Iter.T(Signature)
(* Wenn t ein tuple, record, exception, oper oder function repräsentiert,
   liefere eine Iteration des Signaturen; ansonsten Fehlermeldung *)

fmt(type :T) :String
(* Liefert eine String–Repräsentation von 'type' *)

fmtShort(type :T) :String
(* Liefert eine kurze String–Repräsentatoion von 'type', e.g. "Tuple" "Int" etc *)

isEmptyCases(cases :list.T(Case)) :Bool
(* Liefert true wenn 'cases' leer sind *)

newEmptyCases() :list.T(Case)
(* erzeugt leere cases *)

newArray(type :T) :T
newFun(signatures :list.T(Signature) range :T) :T
newTuple(signatures :list.T(Signature) cases :list.T(Case)) :T
newRecord(signatures :list.T(Signature)) :T
```

```
  newException(signatures :list.T(Signature)) :T
  newOper(signatures :list.T(Signature)  range :T) :T

end
```

B.2 Schnittstelle 'Dynamic'

```
interface dynamic
import list typeRep :TypeRep iter :Iter tm reader writer
export
  error :Exception
  Let Type = dynamic_T
  Let T = DynValue
  (* Ein automorpher Wert value :T ist ein tuple val :T  type :Type end,
     wobei 'type' eine Laufzeit–Typrepräsentation von val's Typ ist *)

  Let LocationId Tuple base :tm.OID  offset :Int end
  (* Eine locationId ist die Zelle einer Variablen, bestehend aus einer Basis
     und einem Offset *)

  Let Binding = Tuple
    name :String
    case typeCase with
      type :Type
    case valueCase with
      value :T
    case locationCase with
      value() :T
      setValue(:T) :Ok
      idgetId() :LocationId
  end

  Let Signature = TypeRep.Signature
  Let Case = TypeRep.Case

  Let Expansion = Tuple
    case okCase
    case boolCase with
      idval :Bool
    case charCase with
      idval :Char
    case intCase with
      idval :Int
    case realCase with
      idval :Real
    case stringCase with
      idval :String
    case adtCase with
    case funCase with
      signatures :list.T(Signature)
      range :T
    case tupleCase with
      tag :String
      fields :list.T(Binding)
    case recordCase with
      fields :list.T(Binding)
    case exception with
      val :String
    signatures :list.T(Signature)
  case arrayCase with
    type :Type   (* type is the element type *)
    size :Int
  case appCase with
    oper :Type
    arguments :list.T(Signature)
    value :T
end

typeOf(value :T) :Type
(* Liefert Typeattribut von 'value' *)

valueOf(value :T) :Ok
(* Liefert Wertattribut von 'value' mit der geringst möglichen Typinformation *)

inspect(value :T) :Expansion
(* Liefert detailiertere Beschreibung von 'value' *)

signatures(value :T) :iter.T(Signature)
(* Wenn 'value' der Wert einer Funktion oder Ausnahme ist, liefere eine Iteration ihrer
   Signaturen; ansonsten Fehlermeldung *)

bindings(value :T) :iter.T(Binding)
(* Wenn 'value' der Wert eines Tupels oder Rekords ist, liefere seine Bindungen, ansonsten
   Fehlermeldung *)

functions(value :T) :iter.T(T)
(* Liefert eine Iteration aller funktionalen Komponenten eines Wertes. Besonders sinnvoll, wenn
   'value' ein Modul ist *)

getIndexed(value :T  i :Int) :T
(* Indizierter Zugriff auf Array *)

setIndexed(value :T  i :Int  newValue :T) :Ok
(* Indizierter verändernder Zugriff auf Array *)

getIndexedId(value :T  i :Int) :LocationId
(* Liefer LocationId des i-ten Array-Elements *)

intern(r :reader.T) :T
(* Liefert einen automorphen Werten von einem reader *)

extern(w :writer.T  value :T) :Ok
(* Schreibt einen automorphen Wert in einen Writer *)

newDefault(type :Type) :T
(* Erzeugt einen default Wert mit Typ 'type' *)

fmtShort(value :T) :String
(* Liefert eine kurze String-Beschreibung eines Wertes, e.g. "tuple" "12" etc. *)

newArray(A <:Ok arr :Array(A)  type :Type) :T
```

```
  newTuple(fields :list.T(Binding) :T
  newTupleCase(tag :String  type :Type  fields :list.T(Binding)) :T
  (* Raise error if tag is not a tag of type 'type' or if 'fields' do not match 'type' *)

  newRecord(fields :list.T(Binding)) :T
  newException(value :String  signatures :list.T(Signature)) :T

  areEqualLocationIds(id1, id2 :LocationId) :Bool
  (* Test, ob id1 und id2 die gleiche Location haben *)

  isWrongTag(tag :String) :Bool
  (* Test, ob 'tag' ein leerer Tuple Tag ist *)

end
```

B.3 Schnittstelle 'Closure'

```
interface Closure
import :TLClosure list
export
T <:Ok
(* Typ der internen Repräsentation einer Funktion *)

Let LocationId = TLClosure.LocationId
Let TypeList = TLClosure.TypeList
(* Liste der globalen Variablen mit ihren Laufzeit–Typinformationen *)

Let DebugInfo = TLClosure.DebugInfo
(* Liste von Debug Informationen einer Funktion *)

error :Exception reason :String end
coerce(f :dynamic_T) :T
(* falls f eine Funktion ist, so wird diese als Wert des Typs T zurückgegeben;
   ansonsten wird ein Ausnahmepaket generiert *)

valueInClosure(name :String  value :dynamic_T  closure :T) :T
(* fügt konstante globale Variable in den Funktionsabschluß von 'closure' ein
   und gibt diese Funktion zurück *)

locationInClosure(name :String  value() :dynamic_T setValue(:dynamic_T) :Ok
      getId() :LocationID  closure :T) :T
(* fügt veränderliche globale Variable in den Funktionsabschluß von 'closure' ein und gibt
   'closure' zurück *)

getTypeList(debugInfo :DebugInfo) :TypeList
(* liefert aus einer Liste von Debug Informationen die Liste der globalen Variablen mit ihren
   Laufzeit–Typinformationen *)

collectGlobalVariables(typeList :TypeList) :list.T(Dynamics.Binding)
(* erzeugt aus 'typeList' eine Liste von Bindungen *)

end
```

C. Ausgewählte Schnittstellen der generischen Daten– und Funktionsvisualisierung

C.1 Schnittstelle 'BrowserExchange'

```
interface BrowserExchange
import list  :Environment window clipboard dragServer menu  :Link
export
 Let EnableT = Fun (enable :Bool) <:Ok
 (* Typ, der die Funtion repräsentiert, mit dem ein Menüeintrag aktiviert werden kann *)

 ClipboardT <:Tuple cb :clipboard.T end
 DragServerT <:Tuple ds :dragServer.T end

 theClipboard :ClipboardT
 (* das Klemmbrett, das von allen Browsern benutzt werden sollte *)

 theDragServer :DragServerT
 (* der DragServer, der von allen Browsern benutzt werden sollte *)

 newClipboard() :ClipboardT
 (* Erzeugung eines neuen Klemmbretts *)

 insertEnablePaste(cb :ClipboardT  m :menu.Popup  enablePaste :EnableT) :Ok
 (* enablePaste aktiviert bzw. deaktiviert Menüeintrag für Paste, falls Klemmbrett
    gefüllt oder geleert wird *)

 deleteEnablePaste(cb :ClipboardT  m :menu.Popup) :Ok
 (* entfernt enablePaste Funktion für m *)

 pasteHdl(cb :ClipboardT  insertBindings(bnds :list.T(Environment.Binding)) :Ok
   default(data :String) :Ok) :Link.Handler
   (* Handle, um Bindungen via insertBindings aus cb in das Objekt zu kopieren *)

 copyHdl(cb :Clipboard.T  selectedBindings : Fun() :list.T(Environment.Binding)) :Link.Handler
 (* Handle, um via selectedBindings erhaltene Bindungen in cb zu kopieren *)

 newDragServer() :DragServerT
 (* erzeugt neuen DragServer *)
```

```
  initDrag(ds :DragServerT  win :window.T
    getBindings() :list.T(Environment.Binding)) :Ok
  (* initialisiere Drag für win auf ds mit getBindings *)

  initDrop(ds :DragServerT  win :window.T
    setBindings(bnds :list.T(Environment.Binding))) :Ok
  (* initialisiere Drop für win auf ds mit setBindings *)

  uninitDrop(win :window.T) :Ok
  (* uninitialisiere Drop für win *)

end
```

C.2 Schnittstelle 'Browser'

```
interface Browser
import environment :Environment  window listBox  :BrowserUtil
export
  T <:Ok
  (* Typ eines Bindungsbrowsers *)

  binding(parent :window.T  bnd :Environment.Binding  keepPermanent :Bool) :T
  (* erzeugt von einer Bindung einen Browser und gibt ihn zurück; erzeugtes
     Protokoll wird permanent in Tabelle gespeichert, falls keepPermanent 'true'  *)

  env(parent :window.T  name :String  env :environment.T
    errorWin :listBox.T  keepPermanent :Bool) :T
  (* erzeugt einen Browser von einem Environment *)

  display(parent :window.T  browser :T  pos :BrowserUtil.Location) :Ok
  (* zeigt einen Browser auf dem Bildschirm an *)

  apply(br :T) :Bool
  (* setzt die visualisierte Bindung mit den Werten der Bildschirmrepräsentation *)

  refresh(br :T) :Ok
  (* füllt die Bildschirminhalte mit dem Wert einer Bindung *)

  unshow(br :T) :Ok
  (* entfernt eine Bildschirmrepräsentation vom Bildschirm *)

  getValue(br :T) :Environment.Binding
  (* liefert die Bindung, die durch br visualisiert wird *)

  setValue(br :T  bnd :Environment.Binding) :Ok
  (* ersetzt die durch br visualisierte Bindung durch bnd *)

end
```

Literaturverzeichnis

Abadi et al. 89: M. Abadi, L. Cardelli, B. C. Pierce und G.D. Plotkin. *Dynamic Typing in a Statically Typed Language.* In: *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages.* ACM, Januar 1989, S. 213–227.

Abadi et al. 90: M. Abadi, L. Cardelli, P.-L. Curien und J.-J. Lévy. *Explicit Substitutions.* Technical Report 54, Digital Equipment Corporation, Systems Research Center, Palo-Alto, California, Februar 1990.

Abadi et al. 92: M. Abadi, L. Cardelli, B. Pierce und D. Rémy. *Dynamic Typing in Polymorphic Languages.* In: *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, Juni 1992.

Agrawal, Gehani 89: R. Agrawal und N.H. Gehani. *Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++.* In: *Proceedings of the second International Workshop on Database Programming Languages*, Salishan, Oregon, 1989.

Albano et al. 95: A. Albano, G. Ghelli und R. Orsini. *An Introduction to Fibonacci: A Programming Language for Object Databases.* Technischer Report FIDE/95/120, FIDE Technical Report Series, Pisa, 1995.

Appel 92: A.W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

Apple Computer 85: Inc. Apple Computer. *Inside Macintosh*, Bd. I, II, and III. Addison–Wesley, Reading, Massachusetts, 1985.

Atkinson et al. 81: M.P. Atkinson, K.J. Chisholm und W.P. Cockshott. *PS–algol: An Algol with a Persistent Heap.* ACM SIGPLAN Notices, Jg. 17, 1981, Nr. 7.

Atkinson, Morrison 86: M.P. Atkinson und R. Morrison. *Integrated Persistent Programming Systems.* In: *Proceedings of the 19th International Conference on Systems Sciences*, Hawaii, 1986, S. 842–854.

Atkinson, Morrison 95: M. Atkinson und R. Morrison. *Orthogonally Persistent Object Systems.* Very Large Databases Journal, Jg. 4, 1995, Nr. 3.

Bancilhon et al. 92: F. Bancilhon, C. Delobel und P. Kanellakis. *Building an Object–Oriented Database System: The Story of O_2.* Morgan Kaufmann Publishers, 1992.

Birrel, Nelson 84: A.D. Birrel und B.J. Nelson. *Implementing remote procedure calls.* ACM Transactions on Computing Systems, Jg. 2, Februar 1984, Nr. 1, S. 39–59.

Blaser 90: A. Blaser (Hrsg.). *Database Systems of the 90s*, Lecture Notes in Computer Science, Bd. 466. Springer–Verlag, Berlin u.a., 1990.

Borras et al. 92: P. Borras, J.C. Mamou, D. Plateau, B. Poyet und D. Tallot. *Building user interfaces for database applications: The O_2 experience.* ACM SIGMOD Record, Jg. 21, 1992, Nr. 1, S. 32–38.

Bremer 96: G. Bremer. *Typüberprüfung in Polymorphen Programmiersprachen: Aufgaben und Lösungsansätze.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, 1996. (inVorbereitung).

Brown et al. 90: A.L. Brown, A. Dearle, R. Morrison, D.S. Munro und J. Rosenberg. *A Layered Persistent Architecture for Napier88.* In: *Proceedings International Workshop on Security and Persistence of Information, Bremen, Germany*, Workshops in Computing, Berlin, 1990, S. 155–172. Springer–Verlag.

Brown 88: M.H. Brown. *Perspectives on Algorithm Animation.* In: *Proceedings of the ACM CHI'88 Conference on Human Factors in Computing Systems*, 1988, S. 33–38.

Burstall, Lampson 84: R. Burstall und B. Lampson. *A kernel language for abstract data types and modules.* In: *Semantics of Data Types*, Lecture Notes in Computer Science, Bd. 173. Springer–Verlag, 1984.

Busch et al. 93: A. Busch, Th. Kuehnel und A. Jahnke. *StarView 2.0 Die portable C++ Klassenbibliothek.* STAR DIVISION, Hamburg, 1993.

Cardelli et al. 88: L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow und G. Nelson. *Modula-3 Report.* Research Report 31, DEC Systems Research Center, 1988.

Cardelli et al. 89: L. Cardelli, J. Donahue, M. Jordan, B. Kalsow und G. Nelson. *The Modula-3 type system.* In: *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Januar 1989, S. 202–212.

Cardelli et al. 91: L. Cardelli, S. Martini, J.C. Mitchell und A. Scedrov. *An Extension of System F with Subtyping.* In: T. Ito und A.R. Meyer (Hrsg.). *Theoretical Acpects of Computer Software*, Lecture Notes in Computer Science. Springer–Verlag, 1991, S. 750–770.

Cardelli, Wegner 85: L. Cardelli und P. Wegner. *On Understanding Types, Data Abstraction, and Polymorphism.* ACM Computing Surveys, Jg. 17, Dezember 1985, Nr. 4, S. 471–522.

Cardelli 84: L. Cardelli. *A Semantics of Multiple Inheritance.* In: G. Kahn, D.B. MacQueen und G. Plotkin (Hrsg.). *Semantics of Data Types*, Lecture Notes in Computer Science, Bd. 173. Springer–Verlag, 1984, S. 51–67.

Cardelli 86: L. Cardelli. *Amber.* In: G. Goos et al. (Hrsg.). *Combinators and Functional Programming Languages*, Lecture Notes in Computer Science, Bd. 242, Berlin u.a., 1986, S. 21–47. Springer–Verlag.

Cardelli 89: L. Cardelli. *Typeful Programming.* Technischer Report 45, Digital Equitpment Corp., Systems Research Center, Palo–Alto, 1989.

Cattell 91: R.G.G. Cattell. *Next–Generation Database Systems.* Communications of the ACM, Jg. 34, 1991, Nr. 10, S. 31–33.

Chang et al. 95: B.-W. Chang, D. Unger und R.B. Smith. *Getting Close to Objects: Object–Focused Programming Environments.* In: M. Burnett, A. Goldberg und T. Lewis (Hrsg.). *Visual Object Oriented Programming.* Prentice Hall, Englewood Cliffs, New Jersey, 1995, S. 185–198.

Chang, Ungar 93: B.-W. Chang und D. Ungar. *Animation: From Cartoons to the User Interface.* In: *UIST'93 Conference Proceedings*, Atlanta, Ga, November 1993, S. 45–55.

Cooper et al. 87: R.L. Cooper, M.P. Atkinson, A. Dearle und D. Abderrahmane. *Constructing Database Systems in a Persistent Environment.* In: *Proceedings of the 13th International Conference on Very Large Databases*, 1987, S. 117–125.

Cooper, Qin 92: R.L. Cooper und Z. Qin. *A Graphical Data Modelling Program With Constraint Specification and Management.* In: *Proceedings of the 10th British National Conference on Databases*, Aberdeen, 1992.

Cooper 90: R.L. Cooper. *Configurable Data Modelling Systems.* In: *Proceedings of the 9th International Conference on the Entity Relationship Approach*, Lausanne, 1990, S. 35–52.

Corbin 91: J.R. Corbin. *The Art of Distributed Applications.* Sun Technical Reference Library. Springer–Verlag, 1991.

de Bruijn 72: N.G. de Bruijn. *Lambda–calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church–Rosser theorem.* Indag. Math., Jg. 34, 1972, Nr. 5, S. 381–392.

Dearle et al. 89: A. Dearle, R. Connor, F. Brown und R. Morrison. *Napier88 – A Database Programming Language?* In: R. et al. Hull (Hrsg.). *Proceedings of the Second International Workshop on Database Programming Languages*, San Mateo u.a., 1989, S. 179–195. Morgan Kaufmann Publishers.

Dearle et al. 90: A. Dearle, Q.I. Cutts und G.N.C. Kirby. *Browsing, Grazing and Nibbling Persistent Data Structures.* In: J. Rosenberg und D.M. Koch (Hrsg.). *Persistent Object Systems*, Newcastle, 1990, S. 56–69. Proceedings of the 3rd International Workshop on Persistent Object Systems, Springer Verlag.

Dearle et al. 92: A. Dearle, Q.I. Cutts und R.C.H. Connor. *An Application Architecture using Type–Safe Incremental Linking.* Technischer Report FIDE/92/56, University of St Andrews, Schottland, 1992.

Dearle, Brown 88: A. Dearle und A.L. Brown. *Safe Browsing in a Strongly Typed Persistent Environment.* Computer Journal, Jg. 31, 1988, Nr. 6, S. 540–544.

Dearle 89: A. Dearle. *Environments: a flexible binding mechanism to support system evolution.* In: *Proc. HICSS–22, Hawaii*, Bd. II, Januar 1989, S. 46–55.

Deux et. al. 90: O. Deux et. al. *The Story of* O_2. IEEE Transactions on Knowledge and Data Engineering, Jg. 2, 1990, Nr. 1, S. 91–108.

Deux 91: O. Deux. *The O_2 System.* Communications of the ACM, Jg. 34, 1991, Nr. 10, S. 34–48.

Ellis, Stroustrup 90: M.A. Ellis und B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, 1990.

Farkas et al. 92: A. Farkas, A. Dearle, G. N. C. Kirby, Q. I. Cutts, R. Morrison und R. C. H. Connor. *Persistent Program Construction through Browsing and User Gesture with some Typing.* Technischer Report CS/92/52, University of St Andrews, 1992.

Fegaras et al. 92: L. Fegaras, T. Sheard und D. Stemple. *Uniform Traversal Combinators: Definition, Use and Properties.* In: *Proceedings of the 11th International Conference on Automated Deduction*, Saratoga Springs, 1992.

Fegaras, Stemple 91: L. Fegaras und D. Stemple. *Using Type Transformation in Database System Implementation.* In: *Proceedings of the 3rd International Conference on Database Programming Languages*, Nafplion, 1991, S. 289–305.

Gawecki, Matthes 94: A. Gawecki und F. Matthes. *The Tycoon Machine Language TML: An Optimizable Persistent Program Representation.* FIDE Technical Report Series FIDE/94/100, Fachbereich Informatik, Universität Hamburg, Deutschland, August 1994.

Gawecki, Matthes 95: A. Gawecki und F. Matthes. *Integrating Query and Program Optimization Using Persistent CPS Representation.* In: M.P. Atkinson (Hrsg.). *Fully Integrated Data Environments.* Springer–Verlag, 1995.

Gawecki, Matthes 96: A. Gawecki und F. Matthes. *Exploiting Persistent Intermediate Code Representations in Open Database Environments.* In: *Proceedings of the 5th Conference on Advances in Database Technology, EDBT'96*, Avignon, Frankreich, März 1996. (erscheint noch).

Geisler 95: A. Geisler. *Basisdienste zur Gestaltung einer reflektiven grafischen Entwicklungsumgebung für eine persistente Programmiersprache.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, 1995.

Goldberg, Robson 83: A. Goldberg und D. Robson. *Smalltalk–80: The Language and its Implementation.* Addison-Wesley Publishing Company, 1983.

Griswold et al. 71: R.E Griswold, J.F. Poage und I.P. Polonsky. *The SNOBOL4 Programming Language.* Prentice Hall, Englewood Cliffs, New Jersey, 1971.

Jablonski 94: S. Jablonski. *MOBILE: A Modular Workflow Model and Architeture.* In: *Proceedings of the 4th International Working Conference on Dynamic Modelling and Information Systems*, Noordwijkerhout, Niederlande, November 1994.

Jablonski 95: S. Jablonski. *Workflow–Management–Systeme: Motivation, Modellierung, Architektur.* Informatik Spektrum, Jg. 18, 1995, Nr. 1, S. 13–24.

Kiczales et al. 91: G. Kiczales, J. des Rivieres und D. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, Ma., 1991.

Kilberth et al. 93: K. Kilberth, G. Gryczan und H. Züllighoven. *Objektorientierte Anwendungsentwicklung, Konzepte, Strategien, Erfahrungen.* Vieweg, Braunschweig, 1993.

Kiradjiev 94: P. Kiradjiev. *Dynamische Optimierung in CPS–orientierten Sprachen.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 1994.

Kirby et al. 92: G.N.C. Kirby, R.C.H. Connor, Q.I. Cutts, R. Morrison, A. Dearle und A.M. Farkas. *Persistent Hyper–Programs.* In: A. Albano und R. Morrison (Hrsg.). *Proc. 5th International Workshop on Persistent Object Systems, San Milano, Italien.* Springer-Verlag, 1992, S. 86–106.

Kirby et al. 94: G. Kirby, F. Brown, R. Connor, Q. Cutts, A. Dearle, V. Moore, R. Morrison und D. Munro. *The Napier88 Standard Library Reference Manual Version 2.2.* Technischer Report CS/94/7, University of St Andrews, 1994.

Kirby, Dearle 90: G.N.C. Kirby und A. Dearle. *An Adaptive Graphical Browser for Napier88.* Technischer Report CS/90/16, University of St.Andrews, 1990.

Kirby 92: G. N. C. Kirby. *Reflection and Hyper–Programming in Persistent Programming Systems.* Dissertation, University of St Andrews, 1992.

Kirschke 94: H. Kirschke. *Persistenz in objekt–orientierten Programmiersprachen am Beispiel von CLOS.* Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Hamburg, 1994.

Kornacker 95: M. Kornacker. *Persistente Sicherungspunkte für langlebige Aktivitäten in offenen Umgebungen.* Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Hamburg, 1995.

Lampson 83: B. Lampson. *A description of the Cedar Language.* Technischer Report CSL-83-15, Xerox Palo Alto Research Center, 1983.

Larkin, Simon 87: J.H. Larkin und H.A. Simon. *Why a Diagram is (Sometimes) Worth Ten Thousand Words.* Cognitive Science, 1987, Nr. 11, S. 65–99.

Lavery 95: D. Lavery. *The Design of Effective Software Visualizations for Persistent Programming Environments.* FIDE Technical Report Series FIDE/95/116, Department of Computing Science, University of Glasgow, Schottland, 1995.

Leroy, Mauny 91: X. Leroy und M. Mauny. *Dynamics in ML.* Rapport de Recherche 1491, INRIA, Domaine de Voluceau, Rocquencourt 78153 Le Chesnay Cedex, France, Juli 1991.

Leung, Apperly 93: Y.K. Leung und M.D. Apperly. *E cubed: Towards the Metrication of Graphical Presentation Techniques for Large Data Sets.* In: *East–West International Conference on Human–Computer Interface: Proceedings of the EWHCI'93*, 1993, S. 9–26.

Liskov et al. 81: B. Liskov, T. Bloom, E. Moss, J.C. Schaffert, R. Scheifler und A. Snyders. *CLU Reference Manual.* Springer–Verlag, 1981.

Mathiske et al. 93: B. Mathiske, F. Matthes und S. Müßig. *The Tycoon System and Library Manual.* DBIS Tycoon Report 212–93, Fachbereich Informatik, Universität Hamburg, Germany, Dezember 1993.

Mathiske et al. 95a: B. Mathiske, F. Matthes und J.W. Schmidt. *On Migrating Threads.* In: *Proceedings on the second International Workshop on Next Generation Information Technologies and Systems*, Naharia, Israel, Juni 1995.

Mathiske et al. 95b: B. Mathiske, F. Matthes und J.W. Schmidt. *Scaling Database Languages to Higher-Order Distributed Programming.* In: *Proceedings of the Fifth International Workshop on Database Programming Languages*, 1995.

Matthes et al. 94: F. Matthes, S. Müßig und J.W Schmidt. *Persistent Polymorphic Programming in Tycoon: An Introduction.* FIDE Technical Report Series FIDE/94/106, Fachbereich Informatik, Universität Hamburg, Germany, August 1994.

Matthes et al. 95a: F. Matthes, R. Müller und J.W. Schmidt. *Towards a Unified Model of Untyped Object Stores: Experience with the Tycoon Store Protocol.* In: M.P. Atkinson (Hrsg.). *Fully Integrated Data Environments.* Springer–Verlag, 1995.

Matthes et al. 95b: F. Matthes, J.W. Schmidt und J. Wahlen. *Using extensible Grammars to Support Data Modeling.* In: M.P. Atkinson (Hrsg.). *Fully Integrated Data Environments.* Springer–Verlag, 1995.

Matthes et al. 95c: F. Matthes, G. Schröder und J.W. Schmidt. *Tycoon: A Scalable and Interoperable Persistent System Environment.* In: M.P. Atkinson (Hrsg.). *Fully Integrated Data Environments.* Springer–Verlag, 1995.

Matthes, Schmidt 91: F. Matthes und J.W. Schmidt. *Bulk Types: Built–In or Add–On?* In: *Database Programming Languages: Bulk Types and Persistent Data.* Morgan Kaufmann Publishers, September 1991.

Matthes, Schmidt 92: F. Matthes und J.W. Schmidt. *Definition of the Tycoon Language – A Preliminary Report.* DBIS Tycoon Report 062–92, Fachbereich Informatik, Universität Hamburg, Germany, 1992.

Matthes, Schmidt 93a: F. Matthes und J.W. Schmidt. *DBPL: The System and its Environment.* In: M. Jarke (Hrsg.). *Database Application Enineering with DAIDA*, Bd. 1 of Research Reports Esprit. Springer–Verlag, Berlin, 1993, S. 319–348.

Matthes, Schmidt 93b: F. Matthes und J.W. Schmidt. *System Construction in the Tycoon Environment: Architectures, Interfaces and Gateways.* In: *Proceedings of Euro–Arch'93 Congress*, Berlin u.a., 1993. Springer–Verlag.

Matthes, Schmidt 94: F. Matthes und J.W. Schmidt. *Persistent Threads.* In: *Proceedings of the Twentieth International Conference on Very Large Data Bases, VLDB*, Santiago, Chile, September 1994, S. 403–414.

Matthes 91: F. Matthes. *P–Quest: Installation and User Manual.* DBIS Tycoon Report 101-91, Fachbereich Informatik, Universität Hamburg, 1991.

Matthes 93: F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer–Verlag, Berlin u.a., 1993.

Matthes 95: F. Matthes. *Higher–Order Persistent Polymorphic Programming in Tycoon.* In: M.P. Atkinson (Hrsg.). *Fully Integrated Data Environments.* Springer–Verlag, Berlin, 1995.

Matthews 87: D. Matthews. *Static and Dynamic Type Checking.* In: *Proceedings of the First International Workshop on Database Programming Languages, Roscoff, Finistere, France,* September 1987, S. 43–52.

McCarthy et al. 62: J. McCarthy, J. Abrahams, D.J. Edwards, T.P. Hart und M.I. Levin. *The Lisp Programmers' Manual.* M.I.T. Press, Cambridge, Massachusetts, 1962.

Merz, Lamersdorf 93: M. Merz und W. Lamersdorf. *Cooperation Support for an Open Service Market.* In: *Proceedings of the IFIP TC6/WG6.1 International Conference on Open Distributed Processing,* Holland, 1993. Elsevier Science Publishers B.V.

Milner et al. 90: R. Milner, M. Tofte und R.W. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

Mitchell, Plotkin 88: J. Mitchell und G. Plotkin. *Abstract Types have existential type.* ACM Transactions on Programming Languages and Systems, Jg. 10, July 1988, Nr. 3.

Morison et al. 90: R. Morison, M.P. Atkinson, A.L. Brown und A. Dearle. *On the Classification of Binding Mechanisms.* Infomation Processing Letters, Jg. 34, 1990, Nr. 2, S. 51–55.

Morrison et al. 94: R. Morrison, A.L. Brown, R.C.H. Connor, Q.I. Cutts, A. Dearle, G.N.C. Kirby und D.S. Munro. *The Napier88 Reference Manual (Release 2.0).* FIDE Technical Report Series FIDE/94/104, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1994.

Müller 91: R. Müller. *Sprachprozessoren und Objektspeicher: Schnittstellenentwurf und –implementierung.* Diplomarbeit, Fachbereich Informatik, Johann Wolfgang Goethe–Universität, Frankfurt, Deutschland, November 1991.

Müßig 94: S. Müßig. *Beiträge zur typsicheren generischen Datenvisualisierung.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Juli 1994.

Myers 90: B.A. Myers. *Taxonomies of Visual Programming and Program Visualization.* Journal of Visual Languages and Computing, Jg. 1, 1990, Nr. 1, S. 97–123.

Neeracher 95: Matthias Neeracher. *Grand Unified Socket Interface.* ETH Zürich, Online Manual: http://err.ethz.ch/members/neeri/macintosh.html, 1995.

O_2 Technology 92: O_2Technology. *O_2 Tools User Manual,* 1992. Draft Version.

Ohori et al. 90: A. Ohori, I. Tabkha, R. Connor und P. Philbrow. *Persistence and Type Abstraction Revisited.* In: A. Dearle, G.M. Shaw und S.B. Zdonik (Hrsg.). *Implementing Persistent Object Bases, Principles and Practice,* 1990, S. 141–153.

Price et al. 93: B.A. Price, I.S. Small und R.M. Baecker. *A Principled Taxonomy of Software Visualization.* Journal of Visual Languages and Computing, Jg. 4, 1993, Nr. 3.

Rovner 86: P. Rovner. *On extending Modula-2 to build large, integrated Systems.* IEEE Software, Jg. 3, November 1986, Nr. 6, S. 46–57.

Schmidt et al. 93: J.W. Schmidt, F. Matthes und P. Valduries. *Building Persistent Apllication Systems in Fully Integrated Data Environments: Modularization, Abstraction and Interoperability.* In: *Proceedings of Euro–Arch'93 Congress*, Berlin, Oktober 1993. Springer–Verlag.

Schmidt, Matthes 93: J.W. Schmidt und F. Matthes. *Lean Languages and Models: Towards an Interoperable Kernel for Persistent Object Systems.* In: *Proceedings of the IEEE International Workshop on Research Issues in Data Engineering, Interoperability in Multidatabase Systems*, Vienna, Austria, April 1993, S. 2–16.

Schmidt 77: J.W. Schmidt. *Some High Level Language Constructs for Data of Type Relation.* ACM Transactions on Database Systems, Jg. 2, 1977, Nr. 3, S. 247–261.

Schmidt 87: J.W. Schmidt. *Datenbankmodelle.* In: P.C. Lockemann und J.W. Schmidt (Hrsg.). *Datenbankhandbuch.* Springer–Verlag, Berlin, 1987, Ch. 1, S. 1–83.

Sheard 91: T. Sheard. *Automatic Generation and Use of Abstract Structure Operators.* ACM ToPLaS, Jg. 19, 1991, Nr. 4, S. 531–557.

Stasko, Patterson 92: J.T. Stasko und C. Patterson. *Understanding and Characterizing Software Visualization Systems.* In: *Proceedings of the 1992 Workshop on Visual Languages*, Seattle, 1992, S. 3–10.

Stemple et al. 90: D. Stemple, L. Fegaras, T. Sheard und A. Socorro. *Exceeding the Limits of Polymorphism in Database Programming Languages.* In: *Advances in Database Technology, EDBT'90*, Lecture Notes in Computer Science, Bd. 416. Springer-Verlag, 1990, S. 269–285.

Stemple et al. 92a: D. Stemple, T. Sheard und L. Fegaras. *Linguistic Reflection: A Bridge from Programming to Database Languages.* In: *Proceedings 25th Annual Hawaii International Conference on System Sciences*, 1992, S. 46–55.

Stemple et al. 92b: D. Stemple, R.B. Stanton, T. Sheard, P. Philbrow, R. Morrison, G.N.C. Kirby, L. Fegaras, R.L. Cooper, R.C.H. Connor, M.P. Atkinson und S. Alagic. *Type–Safe Linguistic Reflection: A Generator Technology.* Research Report CS/92/6, University of St. Andrews, Department of Computing Science, Juli 1992.

Stemple et al. 93: D. Stemple, R. Morrison, G.N.C. Kirby und R.C.H. Connor. *Integrating Reflection, Strong Typing and Static Checking.* In: *Proceedings of the 16th Australian Computer Science Conference, Brisbane, Australia*, 1993, S. 83–92.

SunSoft 92: Inc. SunSoft. *The NeWS Toolkit 3.1 Reference Manual.* SunSoft, Inc., MountainView, 1992.

Ungar, Smith 87: D. Ungar und R.B. Smith. *SELF The Power of Simplicity.* SIGPLAN Notices, Jg. 22, 1987, Nr. 12, S. 227–241.

Weis 90: P. Weis. *The CAML Reference Manual, Version 2.6.1.* Technischer Report 121, INRIA, 1990.

Wetzel et al. 95: I. Wetzel, F. Matthes und J.W. Schmidt. *The STYLE Data Modeling Workbench: Systematics of Typed Language Environments.* In: M.P. Atkinson (Hrsg.). *Fully Integrated Data Environments.* Springer–Verlag, Berlin, 1995.