Basisdienste zur Gestaltung einer reflektiven grafischen Entwicklungsumgebung für eine persistente Programmiersprache

Diplomarbeit

eingereicht bei

Prof. Dr. Joachim W. Schmidt
und
Dr. Martin Lehman

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Datenbanken und Informationssysteme

von
Andreas Geisler
Mendelssohnstr. 4
22761 Hamburg

30. Juni 1995

Inhaltsverzeichnis

1. Einführung

Technologische und administrative Entwicklungen erfordern zunehmend Anwendungen, die komplex strukturierte Massendaten (z.B. Konstruktionsdaten, multimediale Daten) verwalten können. Die Unterstützung solcher datenintensiven Anwendungen, z.B. durch objektorientierte Datenbanktechnologie, ist ein seit einigen Jahren behandeltes Gebiet der Informatik [Heuer 92]. Auch die Entwicklung von großen Softwaresystemen läßt sich als datenintensive Anwendung dieser Kategorie betrachten. Entwurfsanwendungen, wie das Software-Engineering, werden durch relationale Datenbanktechnologie jedoch nur unzureichend unterstützt [Gotthard, Lockemann 85], denn Software-Komponenten sind komplex strukturierte, voneinander abhängige Werte oder Objekte, die sich nicht adäquat auf das Relationenmodell abbilden lassen. Weiterhin handelt es sich bei den betrachteten, persistent zu verwaltenden Entitäten nicht nur um textuelle Repräsentationen von Zwischenergebnissen des Entwicklungsprozesses (z.B. Anforderungsanalysen, Quellcode) oder komplexe grafische Darstellungen (z.B. Objektdiagramme, Prozeßmodelle). Wesentlich ist auch die persistente Manipulation von ausführbaren bzw. bindefähigen Programmen (Funktionen, Prozeduren etc.), d.h. von Daten im weiteren Sinne, im Gegensatz zu oben angeführten nicht ablauffähigen Daten.

Diese Sichtweise, Programme als Daten zu interpretieren und umgekehrt, Daten als Programme, ist der Grundgedanke von Reflektion und ist fast so alt wie die Informatik selbst. Sie geht im Bereich der Rechnerarchitektur auf die Entwicklung der von Neumann-Architektur, in der theoretischen Informatik auf das Konzept der universellen Turing-Maschine und im Bereich der Programmiersprachen auf die Entwicklung von Lisp [McCarthy et al. 62], dem Vorreiter aller funktionalen Programmiersprachen, zurück. Diese Sprachen haben in den letzten zehn Jahren, aufbauend auf Arbeiten im Bereich polymorpher Typsysteme[1] einen starken Entwicklungsschub erhalten. Ihre Erweiterung um das Konzept der orthogonalen Persistenz führte zu Sprachen wie Napier88 [Dearle et al. 89], Galileo [Albano et al. 85], Fibonacci [Albano et al. 91] oder Tycoon [Matthes, Schmidt 92]. Durch orthogonale Persistenz kann der Prozeß des Software-Engineering stärker unterstützt werden, denn Funktionen können, da sie Datenwerte erster Klasse sind, nicht nur als Argumente übergeben oder als Resultat von Funktionen zurückgegeben, sondern auch in persistenten Datenstrukturen gespeichert werden. Somit kann ausführbarer Code in der gleichen persistenten Umgebung manipuliert werden wie alle anderen Datenwerte, im Gegensatz zu Lisp-Dialekten jedoch statisch typisiert.

Ungeachtet der genannten Vorteile polymorph typisierter funktionaler persistenter Sprachen haben herkömmliche, z.B. SQL-basierte relationale Systeme einen Entwicklungsvorsprung, was ihre Benutzerschnittstellen betrifft. Anwender werden bei ihrer Arbeit in vielfältiger Weise durch grafische Werkzeuge unterstützt, die ihnen das Navigieren in der Datenbank

[1] Zur Einführung in dieses Gebiet sei auf [Cardelli, Wegner 85] verwiesen.

und die Anfrageformulierung erleichtern, z.B. durch interaktive Browser und andere 4GL-Werkzeuge. Daneben können Benutzer aus der vorhandenen Datenbasis auch durch algebraische Ausdrücke Werte selektieren, aus diesen neue Werte konstruieren und diese wiederum in der Datenbank in strukturierter Weise entsprechend der Schemadefinition ablegen.

Durch die Betrachtung von Daten im weiteren Sinne liegt es nahe, nicht nur das Data-Engineering durch grafische Werkzeuge zu unterstützen, sondern das Software-Engineering in seiner Gesamtheit, denn die Tätigkeit des Software-Entwicklers enthält einige Analogien zu derjenigen des traditionellen Datenbankbenutzers: Er sucht nach vorhandenen Werten (z.B. Funktionen) und deren Signatur, um sie in neuen Werten zu verwenden, die dann die Basisbausteine für komplexere Werte darstellen usw. Dabei sollen seine neu erzeugten Software-Bausteine Bestandteil der Datenbasis werden, um wiederum anderen Entwicklern zur Verfügung zu stehen. Die Datenbasis der Software-Komponenten wird also ständig aktualisiert und erweitert. Zu den Werkzeugen, die es erlauben, Programme durch direkte grafische Manipulation zu konstruieren [Farkas et al. 92], gehören generische Browser zur Navigation im Objektspeicher, Programmeditoren zur Erzeugung nichtlinearer Programmrepräsentationen (sog. Hyperprogramme [Kirby et al. 92]), sowie Werkzeuge zur Unterstützung der Fehlersuche. In großen, arbeitsteilig zu entwickelnden Systemem kommen zudem noch Werkzeuge zum Versions-, Modul- und Bibliotheks-Management sowie zur Dokumentation hinzu.

Als Implementierungstechnik für oben genannte Werkzeuge ist programmiersprachliche Reflektion, d.h. die Möglichkeit, innerhalb eines Programmes Code zu generieren und diesen in den Programmablauf zu integrieren [Bussche et al. 92], bedeutsam. Reflektion dient jedoch nicht nur indirekt (als Implementierungstechnik für Werkzeuge) der Produktivitätssteigerung, sondern auch direkt, denn sie ermöglicht die typsichere Entwicklung generischer, typgesteuerter Algorithmen.

1.1 Aufgabenstellung

Ziel dieser Arbeit ist es, Dienste und Werkzeuge sowie Konzepte für deren systematische Benutzung zu entwickeln, um das oben beschriebene Szenario einer *reflektiven grafischen Software-Konstruktionsumgebung* für die persistente polymorphe Programmiersprache Tycoon[2] zu ermöglichen. Diese Dienste umfassen

- einen generischen Gateway-Generator zur Integration externer, in C++ implementierter Dienste (insbesondere zur GUI-Programmierung) in die persistente Umgebung;
- die Erweiterung des TL[3]-Compilers in der Weise, daß zur Laufzeit Typrepräsentationen zur Verfügung gestellt werden können, die typsichere reflektive Programmierung ermöglichen;
- eine typsichere Schnittstelle zum Tycoon-Compiler, die von TL-Programmen aus aufrufbar und deshalb zur reflektiven Programmierung geeignet ist; diese wird im folgenden als *aufrufbarer Compiler* bezeichnet;

[2] Tycoon: ***T**yped **c**ommunicating **o**bjects in **o**pen **e**nvironments*
[3] TL: *Tycoon Language*

- *Environments* als Werte erster Klasse, die einerseits dynamisch konstruiert und dem aufrufbaren Compiler als Parameter übergeben werden können und andererseits zur Strukturierung von Benutzersichten auf den persistenten Objektspeicher dienen.

Die Implementierung dieser Dienste erfolgt im Rahmen der persistenten polymorphen Programmierumgebung Tycoon [Matthes 93]. Diese erfüllt die grundlegenden Voraussetzungen dafür, denn Tycoon weist folgende Eigenschaften auf:

Sprachliche Unterstützung funktionaler Modellierung: Basierend auf dem Lambda-Kalkül evaluiert jeder Term zu einem Wert. Funktionen sind Objekte erster Klasse.

Orthogonale Persistenz: Daten sind langlebig, unabhängig von ihrem Typ. Insbesondere sind keine Konvertierungsfunktionen zwischen strukturierten und flachen Datenformaten oder Transportfunktionen vom Hauptspeicher zum Objektspeicher und umgekehrt erforderlich.

Typsichere Integration externer Diensterbringer: Sie ermöglicht die Offenheit des Systems gegenüber externen Diensten, ohne auf die Vorteile strenger und statischer Typisierung verzichten zu müssen.

Selbstimplementierung des Compilers in TL: Diese bildet die Grundlage einer relativ natürlichen Implementierung reflektiver Möglichkeiten, insbesondere die Darstellung von Programm- und Typrepräsentationen in TL selbst.

Zu den weiteren praktischen Arbeiten gehören die Entwicklung eines Prototyps einer grafischen reflektiven Programmierwerkbank zu Demonstrationszwecken sowie die Implementierung von Beispielen zur Nutzung der entwickelten reflektiven Dienste.

1.2 Struktur der Arbeit

Dieser Abschnitt gibt einen Überblick über Aufbau und Inhalt dieser Arbeit. Dazu werden die in den einzelnen Kapiteln behandelten Themen kurz beschrieben.

Die Persistente Programmierumgebung Tycoon: Nach dieser Einführung folgt in Kapitel 2 eine Darstellung des Ist-Zustandes des Tycoon-Systems zu Beginn dieser Arbeit. Diesem wird eine um reflektive Eigenschaften erweiterte Architektur gegenübergestellt. Weiterhin werden die für Kapitel 3 und 4 wesentlichen statischen und dynamischen Typkonzepte von TL dargestellt. Die darauf folgende Diskussion von Programmierstilen in TL schafft die Grundlage für die in Kapitel 3 entwickelte Abbildung des C++-Klassenkonzeptes nach TL.

Ein Generator zur Anbindung externer C++-Bibliotheken: In Kapitel 3 werden Konzepte und Implementierung eines Gateway-Generators zur Einbindung externer, in C++ implementierter Bibliotheken in das Tycoon-System erläutert. Dieser Generator stellt die Grundlage zur Verwirklichung der grafischen Aspekte einer persistenten Entwicklungsumgebung in dieser Arbeit dar.

GUI-Dienste: Nach einer Vorstellung der wesentlichen Konzepte grafischer Benutzerschnittstellen am Beispiel der GUI-Klassenbibliothek StarView wird die Anwendung des Gateway-Generators aus Kapitel 3 zur Einbindung dieser C++-Bibliothek demonstriert. Ferner wird auf Nutzungsmöglichkeiten von *multi threading* zur GUI-Programmierung eingegangen.

Reflektive Dienste: Generik ist eines der Hauptziele des Tycoon-Systems. Mit den Sprachmitteln von TL ist jedoch eine generische Darstellung einer bestimmten Klasse von Algorithmen nicht möglich. Reflektive Möglichkeiten sollen in dieser Situation Abhilfe schaffen. Dazu werden in Kapitel 5 Spracherweiterungen und Schnittstellen vorgestellt und deren Implementierung beschrieben, mit dem Ziel, reflektive Programmierung in Tycoon zu ermöglichen. Dynamische Typisierung, Environments, die Schnittstelle zum TL-Compiler sowie Anwendungsbeispiele sind die wesentlichen Themen.

Die Tycoon-Programmierwerkbank: Konzepte der Benutzung der in den vorigen Kapiteln dargestellten Dienste im Rahmen einer grafischen reflektiven Programmierwerkbank sind das Thema von Kapitel 6. Bezugnehmend auf existierende Systeme (Napier88 [Farkas et al. 92], Tycoon [Mathiske et al. 93]) werden zunächst Möglichkeiten grafischer Benutzerinteraktion in persistenten und reflektiven Umgebungen dargestellt. Aus der uniformen Behandlung von Daten und Programmen sowie von Quellcode und ausführbarem Code ergeben sich zum Teil neuartige Möglichkeiten der interaktiven Programmentwicklung. Mittels dynamischer Typisierung und dem TL-Compiler als Bibliotheksdienst lassen sich Systemkomponenten wie die Modulverwaltung auf die Anwendungsebene verlagern, was zu einer konzeptionellen Vereinfachung des Systems führt.

Zusammenfassung und Ausblick: Zum Schluß werden die wichtigsten Eigenschaften der vorgestellten Dienste und Konzepte zusammengefaßt. Dabei steht der Aspekt der Steigerung der Generik des Programmiersystems im Vordergrund. Ein Ausblick auf zukünftige Forschungsaktivitäten und offene Punkte beschließt diese Arbeit.

2. Die persistente Programmierumgebung Tycoon

Bevor die in der Einführung kurz skizzierten Dienste behandelt werden, soll an dieser Stelle auf die für diese Arbeit wesentlichen Aspekte des Tycoon-Systems eingegangen werden. Zunächst wird die Architektur dieses Systems und die Einordnung der in dieser Arbeit entwickelten Dienste in diese Architektur dargestellt. Grundlage für Kapitel 3 und 5 bilden außerdem die im Anschluß erläuterten statischen und dynamischen Aspekte des Typsystems von TL sowie die Darstellung von Programmierstilen in TL.

Nicht eingegangen werden soll dagegen auf Syntax und elementare Konzepte von TL. Zum Verständnis der in dieser Arbeit dargestellten Programmbeispiele sei auf die Einführung in [Matthes et al. 94] verwiesen. Details zur praktischen Benutzung der interaktiven Systemumgebung und der Bibliotheken finden sich in [Mathiske et al. 93].

2.1 Architektur des Tycoon-Systems

Die Architektur des Tycoon-Systems beruht auf konsequenter Trennung der Aufgaben Datenmodellierung, Datenmanipulation und Datenspeicherung. Diese Trennung wird veranschaulicht in Abbildung 2.1.

In dieser Architektur lassen sich zwei grundlegende Anwendungsebenen identifizieren. Auf unterer Ebene ist das Tycoon-Laufzeitsystem Anwender (Klient) der abstrakten, modellunabhängigen Objektspeicherschnittstelle TSP[1]. Die höhere Anwendungsebene stellt die TL-Sprachebene dar, die durch den TL-Compiler realisiert wird. Auf ihr setzen Anwendungsbibliotheken auf, welche sich in *interne* und *externe* Bibliotheken gliedern. Letztere sind Ausdruck der Offenheit des Systems gegenüber externen Diensterbringern, welche durch den in Kapitel 3 beschriebenen generischen Gateway-Dienst noch erweitert wird.

Zwischen den beiden Anwendungsebenen liegt die Ebene TML[2]. TML stellt eine baumstrukturierte, untypisierte, portable Zwischenrepräsentation dar und wird für diverse quellsprachen- und zielmaschinenunabhängige statische (zur Übersetzungszeit) und dynamische (zur Laufzeit) Optimierungen genutzt [Gawecki, Matthes 94].

Alle oberhalb der TVM[3]-Schicht liegenden Komponenten sind in TL selbst implementiert. Dadurch wird der *bootstrap* des Systems auf unterschiedlichen Hard- und Softwarearchi-

[1] TSP: *Tycoon Store Protocol*, Details zu Eigenschaften der Schnittstelle und zur Datenrepräsentation siehe [Matthes 93]

[2] TML: *Tycoon Machine Language*

[3] TVM: *Tycoon Virtual Machine*

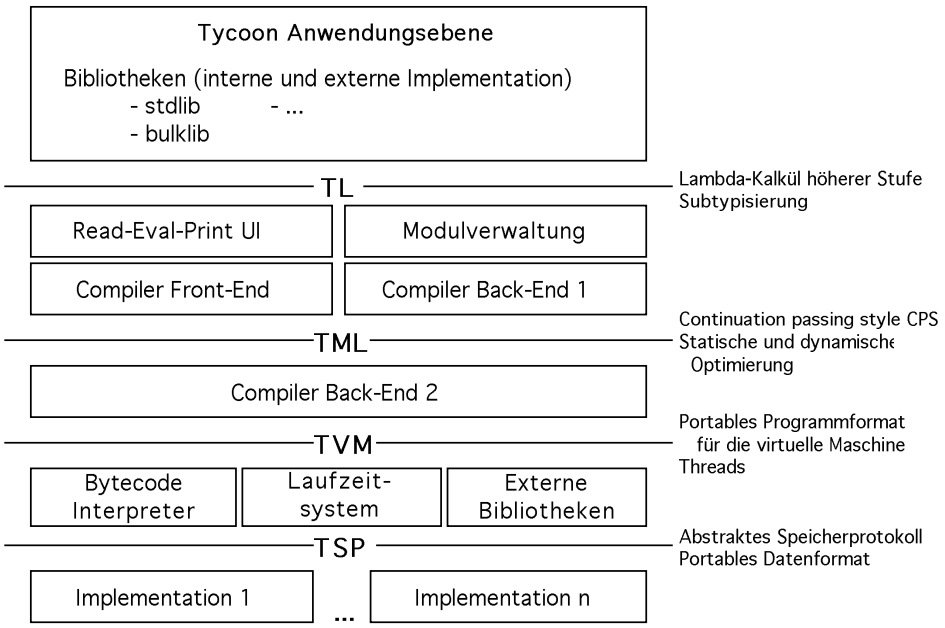

Abbildung 2.1: Tycoon Systemarchitektur

tekturen unterstützt. Durch die Implementierung großer Teile des Systems in TL und die Einführung dynamischer Typen ist es nun möglich, Teile des Compilers, die bisher unterhalb der TL-Ebene lagen wie die Benutzerschnittstelle zum Compiler oder die Modulverwaltung, auf die obere Anwendungsebene zu verlagern, wie in Abbildung 2.2 dargestellt.

Durch diese Öffnung der Architektur soll insbesondere der Compiler dem Benutzer als Bibliotheksdienst zugänglich gemacht werden. Werkzeuge der Programmierumgebung treten dann nicht mehr als ins System integrierte, sozusagen „fest verdrahtete“ Komponenten auf, sondern als Benutzer dieses Bibliotheksdienstes. Dadurch werden die notwendigen Voraussetzungen zur *reflektiven Programmierung* erfüllt, welche Gegenstand von Kapitel 5 ist.

2.2 Erläuterung ausgewählter Typkonzepte von TL

Grundlegend für alle statischen Aussagen über die Semantik von TL-Ausdrücken ist der Begriff der *Signatur*. Eine Signatur ordnet einem Namen statische Information in Form eines *Typausdrucks* zu [Matthes 93]. Solche Typaussagen der Form $x : T$ stellen *partielle* Spezifikationen dar, d.h. der an x gebundene Wert erfüllt mindestens die durch den Typ T gegebene Spezifikation, möglicherweise jedoch auch eine präzisere. Die partielle Ordnung auf Typen und Typoperatoren („ist präziser als“) wird dabei durch die transitive und reflexive *Subtypbeziehung*[4] beschrieben, wobei das *Subsumptionsprinzip* ($a : A \wedge A <: B \Rightarrow a : B$) gilt. Zur Definition von Subtypbeziehungen für die einzelnen Typkonstruktoren ist noch der Begriff der *Subsignatur* wichtig. Signaturen S $(x_1 : T_1, x_2 : T_2, \ldots x_n : T_n)$ heißen Subsignaturen von S', wenn die Anzahl der Signaturkomponenten in S gleich derjenigen in S' ist und $T_i <: T'_i$ sowie $x_i = x'_i$ für alle $i \in \{1..n\}$ gilt, wobei x_i und x'_i beide nicht anonym sein dürfen.

[4] ausgedrückt durch das Infixsymbol $<:$

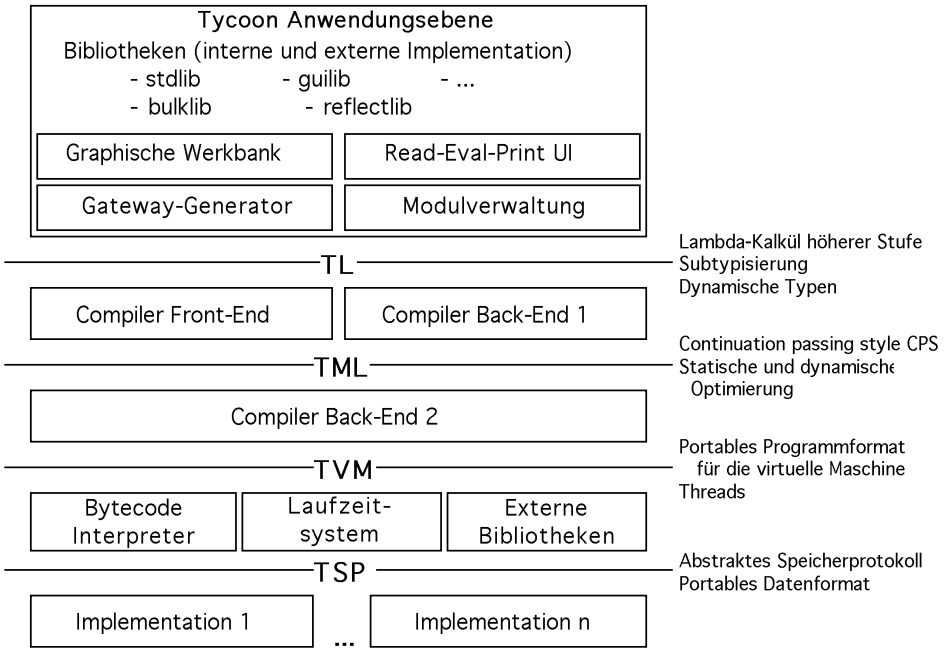

Abbildung 2.2: Reflektive Tycoon Systemarchitektur

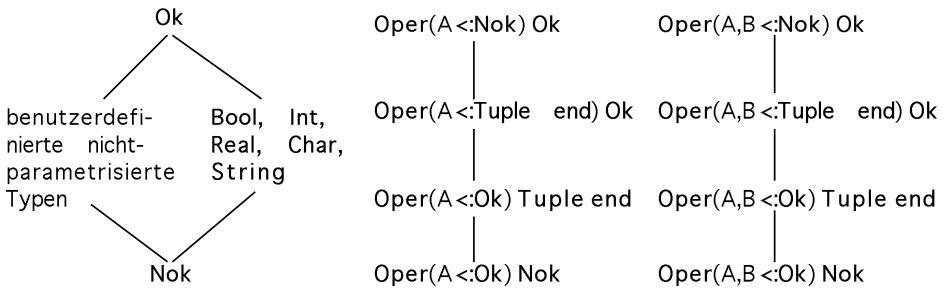

Abbildung 2.3: TL-Typhierarchien

Der Typ **Ok** (triviale Typspezifikation) ist Supertyp aller nichtparametrisierten Typen[5], während der Typ **Nok** (Ergebnistyp aller Ausdrücke, die eine Ausnahme auslösen) Subtyp aller nichtparametrisierten Typen ist. Die Basistypen stehen zueinander nicht in einer Subtypbeziehung. (Abbildung 2.3 links). Parametrisierte Typen (Typoperatoren) erster Stufe lassen sich anhand der Zahl ihrer Argumente in eine Typhierarchie einordnen. In Abbildung 2.3 sind die Hierarchien für Typoperatoren mit einem bzw. zwei Argumenten dargestellt, jeweils mit ihren höchst- und niedrigstliegenden Vertretern und jeweils zwei Beispielen benutzerdefinierter Typoperatoren. Subtypisierung von Typoperatoren erfolgt gemäß der Kontravarianzregel. Typoperatoren höherer Stufe, d.h. Typoperatoren, die Typoperatoren als Argument akzeptieren oder einen Typoperator generieren, bilden wiederum separate Typhierarchien.

[5] d.h. nicht von Typoperatoren

2.2.1 Struktur- und Namensäquivalenz

Subtypisierung beruht in TL auf *struktureller* Kompatibilität. Dies drückt sich in den Subtypsierungsregeln für Tupel-, Record- und Funktionstypen aus.

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

Records: Seien A und B Recordtypen, dann gilt A <: B, wenn die Signaturen von A eine Teilmenge der Signaturen von B enthalten, deren Elemente Subsignaturen der Signaturen von B sind.

Funktionen: Es gilt die Kontravarianzregel.

Für weitere Subtypisierungsregeln sei auf den Sprachreport [Matthes, Schmidt 92] verwiesen. Eine Ausnahme von der Strukturäquivalenz bildet die Kompatibilitätsregel für *abstrakte* Typvariablen, die auf Namensäquivalenz beruht. Beispiele zur Definition abstrakter Datentypen finden sich in Abschnitt 2.3. An dieser Stelle soll das Zusammenspiel von Struktur- und Namensäquivalenz beispielhaft erläutert werden.

Um zufällige, d.h. nicht gewünschte Kompatibilitäten zwischen strukturell äquivalenten Typen zu vermeiden, läßt sich die Möglichkeit, in TL *einschränkende* Wert- und Typbindungen zu definieren, zusammen mit der Namensäquivalenzregel für abstrakte Typvariablen zum Konzept der *semi-abstrakten* Typvariablen kombinieren. Das folgende Beispiel veranschaulicht die Vorteile dieser Kombination aus Struktur- und Namensäquivalenz:

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

let married(p1, p2 :person.T) :Bool = string.equal(p1.name p2.name)
let eiffelTower = building.new(tuple "Tour d´ Eiffel" 95 end)
let meyer = person.new(tuple "Meyer" 50 end)

married(eiffelTower meyer)
⇒ Argument type mismatch: 'person.T' expected, 'building.T' found
    [while checking function argument 'p1']
```

Die Typvariablen *person.T* bzw. *building.T* heißen semi-abstrakt, da sie nicht nur die triviale Spezifikation (<:**Ok**) erfüllen, sondern jeweils auch eine speziellere. Der im Beispiel versuchte Aufruf der Funktion *married* läßt sich nicht korrekt übersetzen, da das erste Argument aufgrund der Namensäquivalenzregel für abstrakte Typvariablen nicht die Bedingung <:*person.T* erfüllt, obgleich die Typstruktur des Wertes *eiffelTower* äquivalent zur Struktur von Werten des Typs *Person* ist. Allerdings lassen sich auf Werte eines semi-abstrakten Typs

alle Operationen anwenden, die auf Werte des als obere Schranke hinter dem <:-Zeichen angegebenen Typs definiert sind. Würde man bei den Bindungen der Bezeichner *person* bzw. *building* den einschränkenden Typ weglassen, so hätten diese Bindungen den vom Compiler inferierten Typ. Die jeweilige Typvariable T wäre dann nicht abstrakt, so daß der Aufruf von *married* wegen der Strukturäquivalenz von *Person* und *Building* fehlerfrei übersetzt werden würde.

2.2.2 Dynamische Typisierung

In datenintensiven, langlebigen und/oder verteilten Anwendungen kann der Programmierer mit Situationen konfrontiert werden, in denen notwendige Typüberprüfungen erst zur Laufzeit möglich sind (näheres dazu in Abschnitt 5.1). Zur Behandlung dieser Situationen werden in [Matthes, Schmidt 92] zwei Sprachkonstrukte vorgeschlagen:

1. Signaturen von Typvariablen können mit dem Schlüsselwort **Dyn** versehen werden.

2. Diese so annotierten Typvariablen können durch einen **typecase**-Ausdruck (ähnlich dem **case**-Konstrukt) zur Laufzeit inspeziert werden.

In jedem Zweig des **typecase**-Ausdrucks wird die dynamische Typvariable auf einen statisch bekannten Typ eingeschränkt, wodurch der Zugriff auf Werte dieses dynamischen Typs statisch überprüft werden kann. Es wird derjenige **when**-Zweig gewählt, dessen Typ zuerst Subtyp des dynamischen Typs ist. Trifft dies auf keinen **when**-Zweig zu, wird der **else**-Zweig ausgeführt. Mit diesen Konstrukten lassen sich ad hoc-polymorphe Funktionen wie im folgenden Beispiel formulieren:

```
let inspect(Dyn T <:Ok) =
  typecase T
  when Int then ...
  when String then ...
  when Tuple name :String end then ...
  ...
  else ...
  end
```

Dieser Ansatz wurde jedoch in der Tycoon-Implementierung nicht weiterverfolgt, da das **typecase**-Konstrukt nur eine Projektion eines dynamischen Typs auf einen festen statischen Typ erlaubt. Da jeder mögliche Zweig statisch antizipiert werden muß, ist es nicht möglich, *typgesteuerte*, algorithmisch vollständige Berechnungen durchzuführen. Insbesondere die oben skizzierte Funktion *inspect* zur Abfrage der Struktur eines dynamischen Typs ist mit diesen Konzepten nicht implementierbar, da die Menge möglicher **when**-Zweige des **typecase**-Ausdrucks unendlich ist. Weiterhin kann sie nicht auf Typoperatoren verallgemeinert werden, da diese separate Typhierarchien bilden, wie schon in Abbildung 2.3 dargestellt. Es wäre dann pro Hierarchie genau eine Funktion zu implementieren, wobei die Zahl möglicher Hierarchien wiederum unendlich ist. In Abschnitt 5.1 wird daher ein alternativer Ansatz vorgestellt und implementiert, der die genannten Einschränkungen überwindet.

2.3 Programmierstile in Tycoon

Die Idee einer interaktiven, grafischen Programmierwerkbank taucht überwiegend im Zusammenhang mit objektorientierten Systemen auf (z.B. [Goldberg, Robson 89]). Dies legt die Vermutung nahe, daß objektorientierte Konzepte als Hilfsmittel zur Verwirklichung dieser Idee besonders geeignet sind. Eine Sprache, die solche Konzepte unterstützt, wäre dann in einem noch höheren Maße im Vorteil gegenüber Sprachen, die diese nicht unterstützen, denn neben den software-technischen Vorteilen (Wiederverwendbarkeit von Code, Klassifizierung, höherer Abstraktionsgrad) kommt dann noch die bessere Unterstützung durch Werkzeuge hinzu. Diese ist durch die zunehmende Komplexität von Software-Systemen unverzichtbar geworden.

Gute Werkzeugunterstützung innerhalb einer Programmierumgebung hängt aber auch stark von der Möglichkeit der Integration einer großen Klasse externer Diensterbringer ab. Die sprachliche Ausgestaltung von TL wurde deshalb möglichst datenmodellneutral gewählt. So werden nicht nur funktionale und imperative, sondern auch objektorientierte Modellierungsansätze unterstützt. Objektorientierte Modellierungsmittel unterstützen somit auch indirekt den Werkbankgedanken wie in Abbildung 2.4 verdeutlicht.

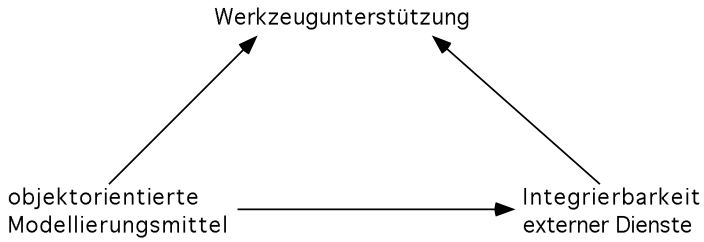

Abbildung 2.4: Determinanten der Werkzeugunterstützung

Nachfolgend wird diskutiert, inwiefern sich bestimmte sprachliche Abstraktionen von TL zur Repräsentation objektorientierter Konzepte eignen. Diese Diskussion bildet die Grundlage für die in Abschnitt 3.2 gewählte Abbildung von C++ nach TL. Es werden Vor- und Nachteile des modularen gegenüber dem objektorientierten Programmierstil behandelt und gezeigt, daß der in TL realisierbare objektorientierte Programmierstil zwar neben rein notationellen auch weitere Modellierungsvorteile bietet, daß diese aber durch einen erheblich höheren Ressourcenbedarf erkauft werden. Um diesen Bedarf zu verringern ist weitergehende Systemunterstützung notwendig, z.B. durch syntaktische Erweiterung verbunden mit einer Änderung der statischen Semantik von TL oder durch die Implementierung eines objektorientierten Typsystems, wie sie z.Zt. mit der Entwicklung der auf TML aufsetzenden Sprache TooL[6] [Gawecki, Matthes 95] verfolgt wird.

Die Hauptkonzepte objektorientierter Programmierung sind Datenkapselung, dynamische Bindung, Subtyppolymorphismus und Vererbung. Nachfolgend wird die Umsetzung dieser Konzepte im modularen (soweit möglich) bzw. im objektorientierten Stil diskutiert.

[6] TooL: Tycoon object oriented Language

2.3.1 Datenkapselung

Der heutzutage sehr hohe Anteil an symbolischer und datenintensiver Verarbeitung in datenverarbeitenden Systemen legt eine *datenorientierte* Modellierung nahe. Sowohl die modulare als auch die objektorientierte Programmierung beinhalten das Konzept des abstrakten Datentyps (ADT). In Tycoon können Modulschnittstellen zur Definition eines ADT verwendet werden. Diese werden durch Tupel- oder Recordtypen repräsentiert:

```
Let Preview = Tuple
  T <:Ok
  new() :T
  changeCurrentPage(self :T currentPage :Int) :Ok
  getCurrentPage(self :T) :Int
  changeZoomFactor(self :T factor :Real) :Real
  ...
end
```

Der Tupelttyp *Preview* wird dabei als abstrakter Datentyp und die Typvariable T in der Signatur von *Preview* als abstrakte Typvariable bezeichnet.

Die Modulimplementation ist dann ein Wert dieses Schnittstellentyps, der entsprechende Wertbindungen für die ADT-Funktionen, die Objektgenerierungsfunktion und eine Typbindung für die abstrakte Typvariable aggregiert.

```
let preview = tuple
  Let T = Word
  let new() :T = ...
  ...
end
```

Der Tupelwert *preview* wird als Implementierung des abstrakten Datentyps und der Typ *Word* als Repräsentationstyp bezeichnet. Die abstrakte Typvariable T ist inkompatibel zu ihrem Repräsentationstyp. Im obigen Beispiel bedeutet dies, daß ein Wert vom Typ *preview.T* nicht an Positionen verwendet werden darf, die einen Wert vom Typ *Word* fordern. Da verschiedene Tupelwerte des abstrakten Datentyps konstruierbar sind, können mehrere Modulimplementationen eines Schnittstellentyps koexistieren.

Der Übergang zur objektorientierten Methodik geschieht nun dadurch, daß in der Signatur des Objekttyps die abstrakte Typvariable sowie die Objektgenerierungsfunktion(en) eliminiert werden. Der implizit gekapselte Objektzustand macht außerdem die Übergabe eines Wertes des abstrakten Typs an die Funktionen des ADT überflüssig [Matthes 93]. Dies führt in diesem Beispiel zu folgendem Objekttyp:

```
Let PreviewOO = Tuple
  changeCurrentPage(currentPage :Int) :Ok
  getCurrentPage() :Int
  changeZoomFactor(factor :Real) :Real
  ...
end
```

Die Objektgenerierungsfunktion liefert nun einen Wert dieses Objekttyps, wobei der Objektzustand im statischen Abschluß (*static closure*) aller ADT-Funktionen enthalten ist.

```
let newPreviewOO() :PreviewOO = begin
  let state = ... (* nicht exportierte Bindung *)
  tuple
    let changeCurrentPage(currentPage :Int) :Ok = ...
      ... (* weitere exportierte Methoden *)
  end
end
```

Die Vorteilhaftigkeit dieser Modellierung wird am folgenden Beispiel deutlich:

```
(* Modulimplementierungen *)
let preview1 :Preview = tuple Let T = ... end
let preview2 :Preview = tuple Let T = ... end

(* Instanzen der ADTs *)
let p1 = preview1.new()
let p2 = preview2.new()

(* Objektgenerierungsfunktionen *)
let newPreview1() :PreviewOO = ...
let newPreview2() :PreviewOO = ...

(* Objekttyp-Instanzen *)
let p3 = newPreview1()
let p4 = newPreview2()
```

In TL gilt, daß verschiedene Implementierungen abstrakter Datentypen untereinander nicht kompatibel sind, es gilt also Namensäquivalenz. In diesem Beispiel sind die Typen von p1 und p2 nicht kompatibel, da weder ihre Namen äquivalent sind (preview1.T $\not\equiv$ preview2.T), noch eine Subtypbeziehung zwischen ihnen besteht. P3 und p4 hingegen sind vom gleichen Typ (PreviewOO), obwohl sie unterschiedlich implementiert wurden.

Das Speichern der Methodenzeiger beim Objekt stellt sicher, daß die zum Objektzustand passende Methode gewählt wird (dynamische Bindung) und bietet außerdem einen notationellen Vorteil beim Methodenaufruf (*objekt.methode()* statt *modul.methode(exemplarvariable)*).

Diese beiden Vorteile werden jedoch gegen einen erheblich größeren Speicherplatzbedarf für Objekte eingetauscht, denn die Methoden werden *pro Instanz eines Objekttyps* gespeichert. Insbesondere bei der Modellierung von komplexen Vererbungssituationen wird dieser Umstand untragbar (siehe nächster Abschnitt).

2.3.2 Vererbung und Subtyppolymorphismus

Vererbung unterstützt die inkrementelle Software-Entwicklung. In einigen objektorientierten Sprachen (z.B. C++, Eiffel) dient das entsprechende Sprachkonzept nicht nur zur Wiederverwendung von Code, sondern führt auch zu einer expliziten Subtypisierung und damit zum

Subtyppolymorphismus. In TL hingegen werden Subtypbeziehungen durch den Typüberprüfer inferiert und nicht explizit durch den Programmierer festgelegt. Sprachkonstrukte zur Modellierung von Vererbungssituationen existieren *unabhängig* von Subtypisierung und Subtyppolymorphismus und bedingen diese nicht.

Angenommen, der Objekttyp PreviewOO im vorigen Beispiel soll Methoden des Objekttyps Window *und* Printer „erben", so läßt sich dies in TL mit Hilfe der *Repeat*-Anweisung realisieren:

```
Let WindowOO = Tuple
  show() :Ok
  ...
end

Let PreviewOO = Tuple
  Repeat WindowOO
  changeCurrentPage(currentPage :Int) :Ok
  getCurrentPage() :Int
  changeZoomFactor(factor :Real) :Real
  ...
end
```

Hierdurch werden im Objekttyp *PreviewOO* alle Signaturen (d.h. die exportierten Methoden) des Objekttyps *WindowOO* wiederholt und die gewünschte Subtypbeziehung (*PreviewOO* <: *WindowOO*) ist erreicht. In der Objektgenerierungsfunktion werden durch die **open**-Anweisung den Signaturen der Methoden entsprechende Wertbindungen wiederholt:

```
let newWindowOO = begin
  let show() :Ok = ...
  ...
end

let newPreviewOO = begin
  let window = newWindowOO()
  tuple
    open window
    let changeCurrentPage(currentPage :Int) :Ok = ...
    ...
  end
end
```

Im modularen Stil läßt sich Subtyppolymorphismus im Beispiel wie folgt erreichen:

```
Let Preview = Tuple
  T <:window.T     (* Export einer einschränkenden Typbindung *)
  new() :T
  ...
end
```

Die *Repeat*-Anweisung läßt sich auch in der Modulschnittstelle verwenden:

```
Let Preview = Tuple
  Repeat Window
  T <:window.T
  new() :T
  ...
end
```

Will man jetzt für einen Wert vom Typ $preview.T$ eine ADT-Funktion eines Supertyps aufrufen, so braucht man nicht mehr zu wissen, welches Modul die Signatur dieser Funktion exportiert, da sie durch die *Repeat*-Anweisung auch in der Modulschnittstelle *Preview* sichtbar ist. Will man allerdings als menschlicher Leser vom angewandten Auftreten einer Funktion zu ihrer Definitionsstelle gelangen, muß man u.U. die ganze ADT-Hierarchie durchsuchen, um ihre Signatur zu erfahren. Dies ist auch ein häufig genannter Kritikpunkt am objektorientierten Programmierstil. Dieser Nachteil wird häufig durch Werkzeugeinsatz (z.B. Klassen-Browser) zu kompensieren versucht.

Zur Modellierung von Mehrfachvererbung ist der Record-Typkonstruktor geeignet, da aufgrund der Subtypisierungsregel für Recordtypen eine aus Recordtypen bestehende Subtyphierarchie einen gerichteten azyklischen Graphen (und nicht nur eine Baumstruktur) bilden kann.

```
Let PreviewOO = Record
  Repeat WindowOO
  Repeat PrinterOO
  ...
end
```

Falls *WindowOO* und *PrinterOO* Recordtypen sind, gilt:

$(PreviewOO <: WindowOO) \wedge (PreviewOO <: PrinterOO)$

Im modularen Stil läßt sich ein solcher Sachverhalt nicht ausdrücken, da für abstrakte Typvariablen höchstens *ein* einschränkender Typ angegeben werden kann (z.B. $T <: window.T$, nicht aber $T <: window.T, printer.T$).

3. Ein Generator zur Anbindung externer C++-Bibliotheken

Die objektorientierte Programmiersprache C++ findet in der industriellen Praxis zunehmende Verbreitung. Gegenüber anderen objektorientierten Sprachen erhofft man sich durch ihren Einsatz einen sanfteren, allmählicheren Übergang in die Benutzung objektorientierter Technologien. Dies liegt unter anderem darin begründet, daß es sich um eine hybride Sprache handelt, die sowohl einen objektorientierten als auch einen imperativen Programmierstil erlaubt. Letzterer wird dadurch ermöglicht, daß die inzwischen sehr weit verbreitete Sprache C eine echte Teilsprache von C++ ist.

Die Verbreitung von C++ wird insbesondere auch durch die zunehmende Entwicklung allgemein verwendbarer Klassenbibliotheken vorangetrieben. Dabei wird insbesondere die Wiederverwendbarkeit von Code betont, der für viele wichtigste Aspekt der Objektorientierung. Die Benutzung externer (d.h. nicht selbst entwickelter) Bibliotheken wird besonders attraktiv durch die Möglichkeit, Klassen nach eigenen Anforderungen zu spezialisieren und Methoden zu überschreiben (engl. *overriding*), was ein Höchstmaß an software-technischer Flexibilität verspricht. Hinzu kommt, daß diese Produkte meist von vielen Benutzern mit unterschiedlichsten Anforderungen erprobt und getestet sind. Diese Aspekte tragen entscheidend zur Effizienz der Software-Entwicklung bei. Da externe Bibliotheken zur Grundkonzeption des Tycoon-Systems gehören [Matthes 93] und das Angebot an C++-Klassenbibliotheken ständig wächst, liegt es nahe, sich auch solche Dienste innerhalb des sprachlichen Rahmens von TL nutzbar zu machen.

Die zentralen Begriffe dieses Kapitels hängen wie folgt zusammen: Ein *Gateway* ist die Software-Komponente eines Systems, die die erforderlichen Bindungen gehörend zu *einer* externen Bibliothek (nachfolgend *C++-Server* genannt) enthält. Es besteht im allgemeinen aus einer Menge modular zusammenhängender Teil-Gateways. Ein *generisches Gateway* ist durch *Schnittstellenbeschreibungen* parametrisierbar und generiert als Instanzen spezialisierte (Teil-)Gateways.

In diesem Kapitel wird beschrieben, welche Aufgaben und Probleme zu lösen sind, die sprachlichen Konzepte von C++ und TL aufeinander abzubilden. Dazu wird eine im Rahmen dieser Arbeit entwickelte Schnittstellenbeschreibungssprache (*gateway description language* GDL) benutzt. Die Implementierung dieser Schnittstellenbeschreibungssprache führt zu einem generischen Gateway oder einem Gateway-Generator, wie er in Kapitel 3.3 beschrieben ist. Den Zusammenhang zwischen den einzelnen Abschnitten dieses und des letzten Kapitels veranschaulicht Abbildung 3.1.

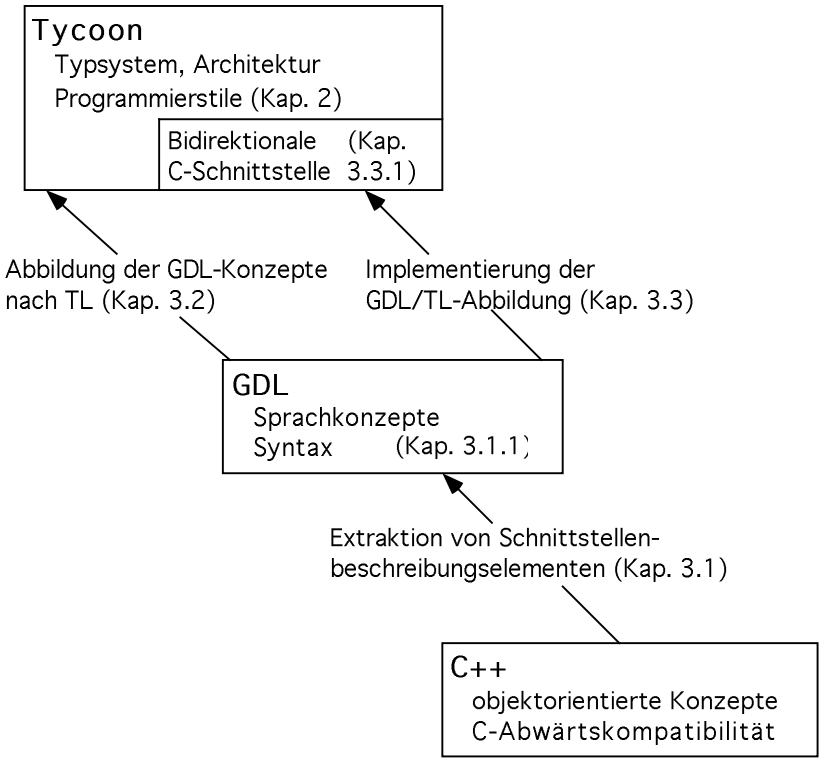

Abbildung 3.1: Struktur von Kapitel 3

Anforderungen an generische Gateway-Implementationen

Ziel ist es, die in C++-Klassenschnittstellen definierten Methoden dem TL-Programmierer in transparenter Form zur Verfügung zu stellen, d.h. er soll von der Existenz eines externen C++-Servers abstrahieren und die externe Bibliothek gemäß der TL-Syntax und -Semantik benutzen können. Insbesondere sollen dabei die Vorteile von TL wie statische Typisierung, Funktionen höherer Ordnung und Polymorphismus in Bezug auf den externen Diensterbringer erhalten bleiben.

Allgemein: Um eine Sprache L_{intern} durch ein generisches Gateway für eine Sprache L_{extern} zu erweitern, müssen folgende Anforderungen erfüllt werden [Schmidt, Matthes 93]:

- Das Typsystem von L_{intern} muß mächtig genug sein, Typen von Werten aus L_{extern} auszudrücken. Dies betrifft sowohl die Basistypen als auch insbesondere die Typkonstruktoren (3.2).
- Es muß ein Werkzeug geben, daß die Abbildung von Signaturen aus L_{intern} auf Signaturen aus L_{extern} unterstützt (3.3).
- Es muß einen Mechanismus geben, um zur Laufzeit Bindungen an externe Objekte, insbesondere Funktionen, zu etablieren (3.3.1).

Nachfolgend wird erläutert, auf welche Weise die Sprache TL und das im Rahmen dieser Arbeit entwickelte Gateway-Generierungswerkzeug zur Erfüllung dieser Bedingungen beitragen.

3.1 Schnittstellenbeschreibungen

Als Grundlage zur Anbindung eines Dienstes benötigt man eine Beschreibung seiner Schnittstellen. Leider erlaubt C++ nicht die konsequente Trennung von Schnittstelle und Implementation durch sprachliche Mittel. Da in den sogenannten C++-Header-Dateien, die per Konvention zur Beschreibung von Klassenschnittstellen benutzt werden, auch Implementationscode vorkommen kann, eignet sich deren Inhalt nicht als Eingangsinformation, auf dessen Grundlage man eine Abbildung der Sprachkonzepte nach TL definieren und anwenden könnte. Ein präprozessorbasierter Ansatz zur Trennung von Schnittstelle und Implementation in C++ wird in [Martin 91] vorgeschlagen, bei dem die notwendigen Programmierkonventionen für den Programmierer explizit sichtbar gemacht werden. Entsprechend wird auch für diese Arbeit ein Schnittstellenbeschreibungsmittel benötigt, das die Möglichkeit bietet, von Implementierungsaspekten vollständig zu abstrahieren sowie für die Tycoon-Umgebung spezifische Ergänzungen zuzulassen. Nachfolgend wird deshalb die C++-Schnittstellenbeschreibungssprache **GDL** (*gateway definition language*) eingeführt. Diese hat u.a. folgende wichtige Eigenschaften:

Deklarativität: GDL ist eine reine Spezifikationssprache ohne algorithmische Bestandteile.

Modularität: Schnittstellenbeschreibungen können andere importieren.

Pragmatik: Um die Erstellung von GDL-Dateien so einfach wie möglich zu gestalten, wird eine C++-nahe Syntax gewählt, so daß C++-*Headerfiles* leicht als Grundlage verwendet werden können.

Außerdem ist vorgesehen, daß spätere Versionen der GDL (bzw. des Gateway-Generators) die Anbindung weiterer Sprachen unterstützen.

Die Benutzung von Schnittstellenbeschreibungssprachen ist weit verbreitet im Kontext verteilter, offener, heterogener Umgebungen. Dienste werden dort üblicherweise in einer IDL (engl. *interface definition language*) beschrieben, zu der sprachspezifische Abbildungsvorschriften zur *Stub*-Generierung existieren ([OMG 91], [DCE 93], [Jansen et al. 94], [Sun 90]). Einige dieser IDLs haben den Anspruch, verschiedene Sprachabbildungen unterstützen zu können (z.B. für C und C++), da in heterogenen Umgebungen typischerweise verschiedene Sprachen zur Anwendungsprogrammierung eingesetzt werden. Innerhalb der persistenten integrierten Programmierumgebung Tycoon ist allerdings nur die Abbildung von GDL nach TL von Interesse, da TL nicht nur für alle Datenmodellierungs- und Applikationsprogrammieraufgaben, sondern auch für den überwiegenden Anteil der Systemprogrammierung benutzt wird.

3.1.1 GDL-Sprachkonzepte

GDL ist die Sprache, die vom C++/TL-Gateway-Generator akzeptiert wird. Sie ist vom Typ LL(1) und unterstützt eine Untermenge der C++-Syntax bezüglich der Deklaration von Konstanten, Typen und Klassen mit ihren Methoden, sowie die C++-Basistypen. Eine Beschreibung ihrer syntaktischen und lexikalischen Regeln findet sich in Anhang A.

GDL-Spezifikation

Eine GDL-Schnittstellenspezifikation besteht aus dem Kopfteil sowie beliebig vielen Typ- und Konstantendeklarationen. Der Kopfteil enthält neben der Liste der importierten Schnittstel-

lenbeschreibungen die Namen der zu generierenden TL-Modulimplementation und der TL-Schnittstelle. Dies ist notwendig, um Namenskonflikte mit bereits existierenden TL-Modulen vermeidbar zu machen. Diese können entstehen, weil TL-Bibliotheken keine gekapselten Namensräume darstellen und somit keine Überdeckungen von Modulbezeichnern möglich sind. Ein Beispiel:

gateway *"C++" window :Window*
import *outputDevice resource ...*

Der Import von Schnittstellenbeschreibungen ist unqualifiziert und entspricht einer **open**-Anweisung in TL auf Modulebene. Qualifizierter Import ist derzeit nicht notwendig, da C++ in der Version 2.3 noch keine Namensräume oberhalb von Klassen[1] unterstützt.

Typ- und Konstantendeklarationen

Konstanten werden durch die *define*-Anweisung sowie durch den Typkonstruktor *enum* definiert. Ein Beispiel:

define *BLACK 0*
enum *ClosedDay* {*MONDAY, SATURDAY = 5, SUNDAY = 6*}

Die *define*-Anweisung ist allerdings nicht wie in C Bestandteil des Präprozessors, sondern in die Sprache integriert.

Basistypen: Es stehen grundsätzlich alle C-Basistypen zur Verfügung (*char, int, float* ...), allerdings kommt bei der Abbildung dieser Typen zum Tragen, daß TL nicht so ein reichhaltiges Repertoire an numerischen Typen unterschiedlicher Größe wie C++ hat. Details der Abbildung der Basistypen werden in Abschnitt 3.2 beschrieben.

An **Typkonstruktoren** stehen die Anweisungen *typedef* und *class* zur Verfügung. Erstere dient der Vergabe von Bezeichnern für Typen (wie in C++).

Klassen aggregieren Operationen und Attribute (*members* in C++Terminologie) zu einer *öffentlichen* Schnittstelle, d.h. es wird im Sinne einer strikten Trennung von Schnittstelle und Implementation nicht wie in C++ zwischen privaten, geschützten und öffentlichen Bestandteilen einer Klasse unterschieden. Die aus Kompatibilitätsgründen zu C auch in C++ eingeführten *structs* sind in GDL nicht notwendig, da der einzige Unterschied zwischen Strukturen und Klassen darin besteht, daß der Sichtbarkeitsbereich ihrer Bestandteile bei Strukturen auf „öffentlich", bei Klassen auf „privat" voreingestellt ist. Über die reine Spezifikation hinausgehende Implementationsdetails wie z.B. Funktionsrümpfe (*inlining*) sind ausdrücklich nicht zugelassen. GDL-Klassen bilden keinen Namensraum, d.h. daß insbesondere Operationen in verschiedenen Klassen innerhalb einer GDL-Spezifikation eindeutig sein müssen. Diese Forderung ergibt sich aus der Art der gewählten Abbildung von Klassen auf von Modulschnittstellen exportierte ADTs (siehe Abschnitt 3.2). Der Regelfall ist, eine Klasse pro GDL-Spezifikation zu definieren und nur wechselseitig rekursive Klassendefinitionen in einer Spezifikation zusammenzufassen. Die Angabe einer oder mehrerer Oberklassen hat Auswirkungen auf die resultierende TL-Typhierarchie (siehe Abschnitt 3.2.1). Ein Beispiel für eine Klassendefinition:

[1] sog. *namespaces*, derzeit noch in der Standardisierungsphase

```
class Window=>T "A window is a rectangular outputdevice" : Outputdevice
{
 ...
 virtual void Paint "Redraw method" (const &Rectangle rRect);
 ...
}
```

In Methodendeklarationen werden die in C++ üblichen Parameterübergabemechanismen unterstützt (*by value, pointer, reference*). Ferner bedeutet das Schlüsselwort *virtual*, daß Vorkehrungen getroffen werden, die betreffende Methode mit einer TL-Funktion durch den Programmierer überschreibbar zu machen. Ein *Callback* durch den externen Server resultiert dann in einem Aufruf der entsprechenden TL-Funktion anstelle der Standardimplementierung dieser Methode, wobei für den Server der Unterschied zwischen einer in C++ und einer in TL implementierten virtuellen Methode nicht erkennbar ist. Nähere Erläuterungen zu diesem Mechanismus enthält Abschnitt 3.3.1.

Durch das Schlüsselwort *static* werden Methoden einer Klasse gekennzeichnet, die ohne Nennung eines Objektes dieser Klasse aufgerufen werden können. In C++ dient diese Kennzeichnung der Vermeidung globaler Funktionen.

Bezeichnervergabe und Dokumentation

Bei der Deklaration können GDL-Bezeichner optional eine Umbenennungskomponente (ausgedrückt durch => *identifier*) enthalten, um sie bei der *Stub*-Generierung an gängige Programmierkonventionen anpassen zu können, sofern eine automatische Bezeichnervergabe nicht erwünscht ist. Optional kann ein GDL-Bezeichner durch eine Zeichenkette zu Dokumentationszwecken ergänzt werden. Diese wird dann als Kommentar an passender Stelle in der generierten Schnittstelle eingefügt.

Einbeziehung von TL-Standardbibliotheken

Mitunter enthalten C++-Bibliotheken Definitionen von Standardklassen, für die ähnliche Abstraktionen in den TL-Standardbibliotheken bereits existieren (z.B. für Standarddatentypen). Damit nun die gewohnten TL-ADTs auch in Signaturen von Operationen eines externen Dienstes benutzt werden können, wird dies dem Generator durch die *use*-Klausel mitgeteilt, z.B.:

```
use T from string as String using stringConvert
(* Verwende den TL-Typ T aus dem Modul string anstelle von String
   unter Benutzung von Umwandlungsfunktionen, die im Modul
   stringConvert implementiert sind *)

class Window=>T : Outputdevice
{
   ...
   String getText();
   ...
}
```

Zur Generierung der Implementation von *Window::getText()* benötigt der Generator Umwandlungsfunktionen, die C++-Objekte der Klasse *String* in TL-Werte des Typs *string.T* transformieren können (bzw. umgekehrt). Dem Generator wird mitgeteilt, daß diese durch das Modul *stringConvert* implementiert sind. Alle Konvertierungsschnittstellen sind konzeptionell Instanzen des Typoperators *Convert*:

```
Let Convert(A <:Ok) =
  Tuple
    T <:Ok
    new(:A) :T
    get(:T) :A
  end,
```

z.B. *StringConvert = Convert(string.T)*.

In der entsprechenden Funktionssignatur der generierten Schnittstelle für die Klasse *Window* kann dann der durch das TL-Modul *string* exportierte Typ *T* anstelle der externen Klasse *String* verwendet werden. Die Implementierung der Umwandlungsfunktionen *new* und *get* im Modul *stringConvert* mit der Schnittstelle *StringConvert* ist allerdings manuell zu leisten, da Wissen über die Semantik der einander entsprechenden Typen erforderlich ist.

3.1.2 Schnittstellenbeschreibungssprachen im Vergleich

In verteilten Systemen tritt häufig die Anforderung auf, Dienste unabhängig von bestimmten Programmiersprachen auf einem abstrakteren Niveau spezifizieren zu können. In Tabelle 3.1 werden die Konzepte von Schnittstellenbeschreibungssprachen verschiedener, zum Teil kommerzieller verteilter Systemumgebungen einander gegenübergestellt.

Auffällig ist bei den meisten dieser Sprachen eine konzeptionelle Nähe zu einer bestimmten imperativen oder objektorientierten Programmiersprache wie z.B. C (DCE-IDL [DCE 93]), C++ (Corba-IDL [OMG 91]) oder Modula-3 (ILU [Jansen et al. 94]). Dies erleichtert sicher die Abbildung auf eine favorisierte (oder für wichtig befundene) Implementierungssprache, zeigt aber andererseits, daß die genannten Sprachen zur Beschreibung der öffentlichen Schnittstelle verteilter Objekte bzw. Dienste nicht ausreichen.

Bei der Benutzung der Schnittstellenbeschreibungen dieser Systeme muß der Programmierer die jeweiligen Sprachabbildungsvorschriften kennen und anwenden. Die GDL des Tycoon-Systems dient hingegen nicht als Schnittstellenbeschreibung für den Anwendungsprogrammierer, sondern als reines Hilfsmittel zur Generierung von TL-Schnittstellen. Für den Anwendungsprogrammierer hat dies den Vorteil, daß er von der Existenz dieser Beschreibungssprache und der Abbildungsvorschriften nach TL abstrahieren und nur mit Hilfe der generierten TL-Schnittstellen programmieren kann.

Aufbauend auf den Erfahrungen mit der Abbildung von GDL-Strukturen auf TL ist es Gegenstand weiterer Arbeiten im Tycoon-Projekt, Vorschriften für eine Abbildung einer der standardisierten IDLs in TL-Typstrukturen zu spezifizieren. Im Rahmen einer vom Autor durchgeführten Studie ist dies für die IDL der OMG vollständig gelungen. Dadurch ist es möglich, mit Hilfe des TL-Typüberprüfers die Schnittstellen unabhängig voneinander entwickelter externer Dienste, die also nicht in einer expliziten Vererbungsbeziehung zueinander

Konzept	SUN-RPC Language (C-Stil)	DCE-IDL (C-Stil)	OMG-IDL (C++-Stil)	ILU-ISL (Modula-3-Stil)	GDL (TL- und C++-Stil)
Schnittstelle	`program` (Name und Version)	`interface` (Name und Attribute)	`interface` (mehrere pro Übersetzungseinheit)	`INTERFACE` (Name und Version)	`gateway` (Interface- und Modulename)
Basistypen	wie in C, zusätzlich `string`, `bool`, `opaque`	wie in C, zusätzlich `handle_t`, `error_status_t`, `byte`	wie in C++, zusätzlich `boolean`, `octet`, `any`	wie MODULA-3, zusätzlich BYTE	C-Typen
Typkonstruktoren	wie in C, discriminated unions	wie in C, discriminated unions, pipe	wie in C, discriminated unions, sequence	`ARRAY`, `SEQUENCE`, `RECORD`, `UNION`, `OPTIONAL`, `ENUMERATION`, `OBJECT (CLASS)`	`enum`, `class`, `struct`
Konstanten	wie in C	wie in C	wie in C++	numerische, Strings	numerische, Strings
Exceptions	—	—	√	√	—
Operationen	C-Prototypen	C-Prototypen mit Funktions- und Parameter-Attributen	C-Prototypen, optionales Attribut "oneway", `raises`-Klausel, Kontext-Ausdruck, Parameter-Attribute (in, out, inout)	Methoden von Objekttypen (Modula-3-Prozeduren), `RAISES`-Klausel, Attribute (`FUNCTIONAL`, `ASYNCHRONOUS`)	C++-Prototypen mit Einschränkungen
Modularität	höchstens über Präprozessor, unüblich	textueller Import	#include-Anweisung (Präprozessor)	qualifizierter Import	unqualifizierter Import (open)
Besonderheiten	—	context-handles, pipes	Kontext-Ausdrücke	siblings, Unterstützung für Garbage Collection	use-Klausel zur Benutzung TL-implementierter Typen

Tabelle 3.1: Schnittstellen-Beschreibungssprachen

stehen, auf Kompatibilität hin zu überprüfen. In einem offenen Markt von Diensten könnte man dann z.B. anhand der Abbildungsregeln überprüfen, ob ein bestimmter angebotener Dienst einer in TL beschriebenen Referenz-Schnittstelle genügt. Diese Typkomponente eines solchen *Traders* [Müller-Jones et al. 95] wäre auch ein Anwendungsfeld für *dynamische Typen* (siehe Abschnitt 5.1), da Typrepräsentationen aus verschiedenen Kontexten miteinander verglichen werden müssen.

3.2 Spezifikation der GDL/TL-Sprachabbildung

In diesem Kapitel wird die Abbildung der GDL-Konzepte auf TL definiert, d.h. es wird beschrieben, wie in TL die GDL-Basistypen, konstruierte Typen und Konstanten sowie Klassenoperationen so ausgedrückt werden, daß die Vorzüge des C++-Typsystems bewahrt bleiben.

Eine Kompilationseinheit der GDL (genannt *gateway*) wird in genau ein TL-Schnittstellenmodul und ein TL-Implementationsmodul mit den in der Kopfzeile des *gateways* angegebenen Namen übersetzt, wobei in der TL-Schnittstelle die Signaturen aller im *gateway* definierten Typen und Konstanten exportiert werden.

In Tabelle 3.2 werden die einander entsprechenden Sprachkonzepte von GDL und TL im Überblick dargestellt und in den folgenden Abschnitten auf Einzelheiten der gewählten Abbildung eingegangen. Vorweg sei gesagt, daß die in Abschnitt 2.3 beschriebene exemplarorientier-

GDL	**TL**
Gateway	Modul
Konstante (*define*)	Wertbindung
Konstante (*enum*)	Export von Werten eines semi-abstrakten Typs (<:Int)
class	ADT mit Konstruktor- und Destruktorfunktion, Lese- und Schreibfunktionen für Attribute
typedef	Typbindung
Klassenmethoden	Funktionen eines ADT
Subtyppolymorphismus durch Vererbung	Subtyppolymorphismus durch einschränkende Typbindungen

Tabelle 3.2: Gegenüberstellung der Sprachkonzepte

te Modellierung von Objekttypen bei tiefgeschachtelten Vererbungshierarchien zu hohem Speicheraufwand führt. Die versuchsweise Realisierung der Anbindung der GUI-Klassenbibliothek StarView^{TM} [2] in diesem Stil erforderte außerdem einen sehr hohen Typüberprüfungsaufwand bei der Übersetzung der generierten Module, bedingt durch die große Zahl erforderlicher *Repeat*-Anweisungen und dadurch häufiger Subtyptests. Zur Anbindung klassenorientierter Bibliotheken ist die exemplarorientierte Modellierung von Objekttypen demzufolge nicht geeignet, da Sprachmittel zur Klassifikation von Objekten fehlen. Daher stellt sich die Abbildung objektorientierter Konzepte von C++ auf den modularen Stil unter Verwendung semiabstrakter Datentypen als die bessere weil effizientere Alternative dar. Wie sich im folgenden herausstellt, wird dabei die ursprüngliche Ausdrucksmächtigkeit von C++ nicht wesentlich eingeschränkt.

[2] siehe Kapitel 4

3.2.1 Abbildung der Typsysteme

In diesem Abschnitt wird die Abbildung des Typsystems von GDL, d.h. C++, auf TL bezüglich Basistypen, Konstanten und Klassen erläutert.

Basistypen

Die Abbildung der GDL-Basistypen auf TL orientiert sich an den Erfordernissen der bidirektionalen C-Schnittstelle [Mathiske et al. 93]. Problematisch ist jedoch die Anbindung von Funktionen, in deren Signatur Zeiger vom Typ *void* vorkommen, das heißt Zeiger auf beliebige Objekte. Diese Objekte können dann zwar mangels Typinformation von C aus nicht manipuliert, wohl aber z.B. in Datenstrukturen verwaltet oder einfach weitergereicht werden. Die korrekte Behandlung externer Referenzen auf TL-Datenstrukturen wird in Abschnitt 3.3.1 beschrieben.

Konstanten

Define-Anweisungen werden in entsprechende Bindungen übersetzt:

define *BLACK 0* $\longrightarrow$ ***let*** *black = 0*

wobei der tatsächliche Wert der Konstante im Implementationsmodul verborgen ist und in der Schnittstelle die Signatur

*black :**Int***

exportiert wird. Die *enum*-Anweisung im Beispiel auf Seite 18 erzeugt entsprechende Bindungen, wobei folgende Signaturen exportiert werden:

*ClosedDay <:**Int***
monday, saturday, sunday :ClosedDay

Ein Vorteil dieser Abbildung ist, daß Werte vom Typ *ClosedDay* an allen Stellen verwendbar sind, an denen Werte vom Typ *Int* verlangt werden, jedoch umgekehrt Funktionen, in deren Signaturen der Typ *ClosedDay* vorkommt, gegen falsche Verwendung geschützt sind. Letzteres ist in C nicht der Fall, denn man könnte statt eines gültigen Wochentages auch beliebige andere ganzzahlige Werte übergeben.

Klassen

Die in Abschnitt 2.3 geführte Diskussion ergab, daß der objektbasierte, exemplarorientierte Programmierstil weniger geeignet zur Modellierung großer klassenbasierter Programmsysteme ist. Daher wird eine GDL-Klasse mit ihren Methoden auf eine Menge von Funktionen auf einem abstrakten Datentyp, dem Typ der Objekte dieser Klasse, abgebildet. Diese Funktionen werden zusammen mit dem abstrakten Typ im Rahmen einer Modulschnittstelle exportiert. Aus einer GDL-Schnittstellenbeschreibung mit mehreren Klassen entsteht eine TL-Schnittstelle mit entsprechend vielen abstrakten Datentypen. Für Klassenkomponenten, die

keine Funktionen sind (also öffentlich sichtbare Attribute), wird je eine Lese- und Schreibfunktion erzeugt. Dadurch werden GDL-Klassen, die einen semi-abstrakten Datentyp repräsentieren, in einen vollständig abstrakten Datentyp überführt. Man erhält so auch immer den aktuellen Wert des Attributes, was bei einer statischen Bindung an einen Wert nicht der Fall ist.

Die Klassenhierarchie der externen Bibliothek wird in TL dadurch nachgebildet, daß die aus den GDL-Klassenspezifikationen resultierenden ADTs in eine Subtypbeziehung zueinander gebracht werden, die der ursprünglichen Vererbungshierarchie entspricht. Da in TL das Subsumptionsprinzip gilt (siehe Abschnitt 2.2), können Funktionen eines aus einer Klassenspezifikation resultierenden ADTs auch auf Instanzen eines Subtyps dieses Typs angewandt werden (*Subtyppolymorphismus*). Bei Mehrfachvererbung sind allerdings Umwandlungsfunktionen zu generieren, da im Typsystem von TL Typbindungen nur *einer* Einschränkung gehorchen können (siehe Beispiel auf Seite 14 unten). Die Typumwandlung muß dabei vom C++-Compiler vorgenommen werden, da nur er das tatsächliche Speicher-Layout von Objekten kennt.

Als Beispiel sei die Klasse *Preview*, Subklasse von *Window* und *Printer*, aufgeführt:

```
gateway "C++" preview :Preview
import window printer fraction
export
  class Preview : Window, Printer {
    constructor Preview=>new(Window* parent)
    Fraction changeZoomFactor(const Fraction& factor)
    ...
  }
end
```

Die resultierende TL-Schnittstelle sieht wie folgt aus:

```
interface Preview
import window printer fraction optional ...
export
 T <:window.T                          (* T ist Subtyp der erstgenannten Superklasse *)
 new(parent :optional.T(window.T)) :T  (* Konstruktor *)
 changeZoomFactor(self :T  factor :fraction.T) :fraction.T
 ...                                   (* weitere Methoden der Klasse Preview *)
 asPrinter(:T) :printer.T              (* in C++ implementierte Typkonvertierung *)
end
```

3.2.2 Abbildung weiterer Sprachkonzepte

Methoden und Overriding

Parameterübergaben an TL-Funktionen haben grundsätzlich Referenzsemantik, d.h. es werden keine Kopien der Aktualparameter erzeugt. In den Signaturen von TL-Funktionen ist

daher nicht mehr ersichtlich, ob an den C++-Server Objekte oder Objektreferenzen übergeben werden. Die jeweils korrekte Übergabeform wird in der Implementation der Abbildung von GDL nach C gewählt (siehe Beispiele für *Wrapper*-Funktionen auf Seite 30).

Zeiger werden auf den durch das TL Standardmodul *optional* exportierten Typoperator *T* abgebildet (siehe obiges Beispiel). Dieser Datentyp zur Modellierung optionaler Werte hat folgende Definition:

```
Oper(A <:Ok) <:Ok
  Tuple
    case nil
    case notNil with value :A
  end
```

Er wird in dem Sinne verwendet, daß ein nullwertiger Zeiger in C der Option *nil* entspricht. Der Vorteil dieser Modellierung ist, daß der in C-Programmen häufig auftauchende Fehler der Dereferenzierung eines nullwertigen Zeigers in TL nicht auftreten kann.

Funktionen eines ADT erhalten als ersten Parameter immer eine Variable dieses Typs, da sie ja nicht (wie beim objektorientierten Ansatz) auf einen gekapselten Objektzustand zugreifen können (vgl. Abschnitt 2.3.1). Eine Ausnahme hiervon bilden Methoden, die in der GDL-Spezifikation mit dem Schlüsselwort *static* gekennzeichnet sind. Da diese Methoden unabhängig von einem bestimmten Objekt aufgerufen werden können, braucht hier keine Variable des ADTs als erster Parameter übergeben zu werden. Die Signaturen der Konstruktoren eines ADTs entsprechen den Signaturen der Konstruktoren der zugrundeliegenden Klasse.

Subklassenbildung in C++ ist häufig mit der (signaturerhaltenden) Redefinition von virtuellen Methoden verbunden, dem sogenannten *overriding*. Wird ein aus einer GDL-Klassenspezifikation resultierender ADT vom TL-Programmierer in der Absicht benutzt, ihn zu spezialisieren, so muß dem externen C++-Server die Möglichkeit gegeben werden, TL-Funktionen aufzurufen. Dazu wird jede virtuelle Methode als Attribut eines ADTs betrachtet, für das ein Paar von Schreib-/Lesefunktionen generiert wird. Beispiel:

```
interface Window
import rectangle ...
export
  ...
  setPaint(:T :Fun(:rectangle.T) :Ok) :Ok
  getPaint(:T) :Fun(:rectangle.T) :Ok
end
```

Overriding bedeutet dann, dieses Attribut destruktiv zu ändern (im Beispiel durch einen Aufruf der Funktion *setPaint*). Im Gegensatz zu C++ ist dann ein „Overriding“ auf Objektebene möglich statt nur auf Klassenebene, so daß sich verschiedene Instanzen eines ADTs dem Server gegenüber unterschiedlich verhalten können, wenn ihre funktionswertigen Attribute unterschiedliche Werte aufweisen. Allerdings entsteht durch dieses Vorgehen auch ein erhöhter Platzbedarf (siehe 3.3.3).

Templates

Neuere C++-Versionen ermöglichen die Definition parametrisierter Typen (Typoperatoren) durch *templates*. C++-Templates sind Typoperatoren erster Stufe, wobei als Typparameter nur Klassen erlaubt sind. Dieses wenig orthogonale Konstrukt läßt sich leicht auf einfache TL-Typoperatoren abbilden, d.h. auf nichtrekursive Typoperatoren erster Stufe. Beispiel:

C++	***TL***
template<***class*** *T*> ***class*** *List* {	***Tuple***
public:	*List* <:***Oper***(*T* <:***Ok***) ***Ok***
List();	*new*(*T* <:***Ok***) :*T*(*E*)
List& cons(T elem);	*cons*(*T* <:***Ok*** *elem* :*T* *l* :*List*(*T*)) :*List*(*T*)
T& head();	*head*(*T* <:***Ok*** *l* :*List*(*T*)) :*T*
List& tail();	*tail*(*T* <:***Ok*** *l* :*List*(*T*)) :*List*(*T*)
}	***end***

Hauptanwendungsfall für den Template-Mechanismus ist die Definition generischer Container-Klassen. Da dessen Definition durch die Mächtigkeit der TL-Typoperatoren eine der Hauptstärken von Tycoon ist, erscheint es nicht als lohnenswert, ähnliche Dienste externer Anbieter anzubinden. Außerdem werden Templates noch nicht von allen C++-Compilern unterstützt. Aus den genannten Gründen werden Templates vorerst nicht in die Definition der GDL aufgenommen. Sollte in Zukunft doch verstärkt der Wunsch nach Einbindung von C++-Templates in TL aufkommen, aus welchem Grund auch immer, ist eine entsprechende Erweiterung der GDL jederzeit möglich, da die Abbildung bereits definiert ist.

In TL nicht direkt abbildbare C++-Konzepte

Einige C++-Sprachkonzepte gestatten eine komfortablere Notation gewisser Programmiersituationen. Zu diesen Konzepten gehören:

- variabel lange Parameterlisten,
- vorbelegte Argumente,
- überladen von Funktionen und Operatoren, d.h. die Möglichkeit, mehrere Funktionen und Operatoren gleichen Namens aber unterschiedlicher Signatur zu definieren.

Zur Begrenzung der Sprachkomplexität von TL wurde auf solche Konzepte verzichtet [Matthes 93]. Da TL-Funktionen nur *eine* eindeutige Signatur besitzen können, ist eine überladene Funktion durch eine Menge von TL-Funktionen abzubilden. Die Größe dieser Menge entspricht der Anzahl der möglichen Signaturen für diese überladene Funktion. Operatoren sind entsprechend auf TL-Funktionen abzubilden.

3.2.3 Benennungsschema

Die Abbildung von GDL-Bezeichnern auf TL-Bezeichner gehorcht folgenden Regeln:

- Falls vorhanden, wird der nach dem Umbenennungsoperator (=>) angegebene Bezeichner verwendet.
- TL-Konventionen für Groß-/Kleinschreibung von Bezeichnern werden vom Generator automatisch berücksichtigt.
- GDL-Bezeichner, deren Abbildung zu einem TL-Schlüsselwort führen würde, sind umzubenennen.
- Parameterbezeichner einer Funktion dürfen nachfolgende Modulbezeichner in der Signatur dieser Funktion nicht überdecken wie in folgendem Fall:

  import *color*
  changeColor(color :color.T) :color.T

  Der Parameter *color* muß umbenannt werden.

3.3 Implementierung der GDL/TL-Sprachabbildung

Die in den vorigen Abschnitten behandelten Abbildungsregeln erlauben eine fast vollständig automatisierbare Erzeugung von TL-Schnittstellen und -Modulen sowie des erforderlichen C++-Codes (bestehend aus C-*Wrapper*-Funktionen und abgeleiteten Klassen) aus Spezifikationen in GDL. Die Größe mancher Bibliotheken macht ein solches Vorgehen auch zwingend erforderlich, da eine manuelle Codierung durch ihren schematischen, repetitiven Charakter ziemlich fehlerträchtig wäre.

An die Implementation eines generischen Gateways werden folgende Anforderungen gestellt:

- Die Größe des generierten Gateway-Codes sollte sich in einem akzeptablen Rahmen bewegen.
- Es sollte keine signifikante Erhöhung der Laufzeit entstehen.
- Generierter Gateway-Code sollte modular aufgebaut sein.

3.3.1 Basismechanismen

In TL stehen die Modularisierungsmechanismen Modul, Modulschnittstelle und Bibliothek zur Verfügung. Diese unterstützen die Entwicklung und Wartung großer Softwaresysteme und erleichtern die Interaktion mit externen Diensterbringern [Matthes 93]. Das Modulkonzept spielt eine wichtige Rolle als Abstraktionsmechanismus zur Schnittstellenbeschreibung von abstrakten Datentypen, auf den das C++-Klassenkonzept abgebildet wird. Im folgenden werden die Basismechanismen zur Bindung an externe Funktionen und zur Behandlung von externen sowie schwachen Referenzen erläutert.

Bindung an externe Funktionen

Die Bindung an externe Funktionen (in Maschinencode) wird im Tycoon-System durch die bidirektionale Schnittstelle zwischen TL und C verwirklicht [Mathiske et al. 93]. Zu dieser Schnittstelle gehören die in der initialen Umgebung vordefinierte Funktion *bind* sowie das Bibliotheksmodul *cCallback*. Der Aspekt der Bidirektionalität ist besonders im Kontext der Anbindung externer GUI-Dienste wichtig, da bei diesen sämtliche Reaktionen des Systems auf Benutzerinteraktionen durch *Callbacks* verwirklicht werden.

Zur Integration externer Funktionen in TL dient die vordefinierte Funktion *bind* mit folgender Signatur:

*bind(Function <:***Ok** *library, label, format :String) :Function*

Der erste Parameter beschreibt den Typ der aus dem Aufruf resultierenden TL-Funktion. Die beiden folgenden String-Literale definieren eine eindeutige symbolische Adresse, unter der der Maschinencode der externen Funktion zu finden ist. Als letztes folgt eine Codierung der formalen Parameter und des Rückgabewertes, wobei nur die unstrukturierten TL-Basistypen erlaubt sind. Komplexere Bindungsanforderung wie z.B. die Übergabe strukturierter Werte müssen auf eine mehrfache Anwendung dieser Basisprimitive abgebildet werden. Details zur bidirektionalen C-Schnittstelle finden sich in [Mathiske et al. 93].

Eine ähnliche bidirektionale Schnittstelle könnte man sich auch zwischen TL und C++ vorstellen, allerdings liegt durch die Kompatibilität von C++ zu C ein Ansatz nahe, der auf einer Abbildung von C++-Sprachkonzepten auf C beruht. Dies hat folgende Vorteile:

1. Portabilität: C++-Compiler codieren Typinformation über Parameter und Klassenzugehörigkeit in die Namen von Methoden hinein (*function name encoding* oder auch *name mangling*), um typsicheres Binden und *Overloading* von Funktionsnamen zu implementieren. Ein Codierungsschema hierfür wird in [Ellis, Stroustrup 90] beschrieben, es wird allerdings ausdrücklich auf Implementationsabhängigkeiten hingewiesen. Ein TL/C++-Gateway, das Annahmen über ein bestimmtes Codierungsschema macht, ist deshalb im allgemeinen nicht portabel.

2. Einfachheit: Diese wird durch Nutzung vorhander Bindungsmechanismen erreicht.

3. Wartbarkeit: Da C++ noch im Prozeß der Standardisierung begriffen ist, ist ein auf standardisierten Konzepten (ANSI-C) basierendes Gateway leichter zu warten.

Callbacks

Da, wie oben erwähnt, GUI-Applikationen in hohem Maße auf *Callbacks* basieren, ist ein Mechanismus notwendig, der TL-Funktionen von externen Diensterbringern aus aufrufbar macht. Dieser Mechanismus wird durch das Bibliotheksmodul *cCallback* implementiert. Durch den Konstruktor *cCallback.new* werden TL-Funktionen derart verpackt, daß sie für den externen Diensterbringer als C-Funktionszeiger erscheinen und damit wie normale C-Funktionen aufgerufen werden können.

Abbildung 3.2 verdeutlicht den *Callback*-Mechanismus anhand eines typischen Aufrufszenarios einer GUI-Anwendung. Aus der Funktion *clickHandler(b :button.T)* **:Ok** wird mittels der

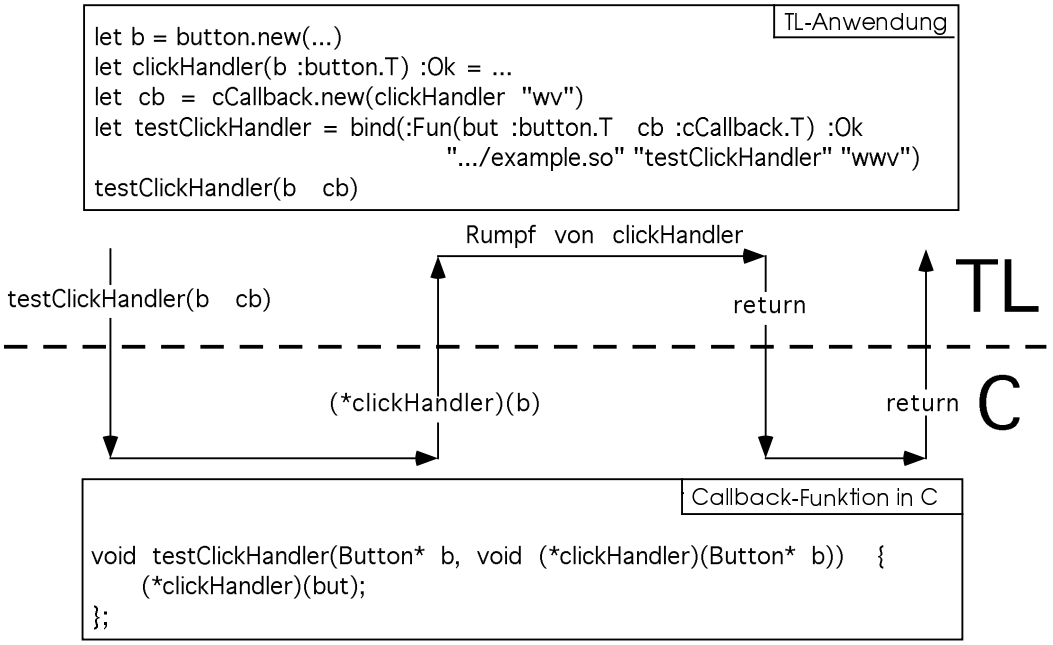

Abbildung 3.2: Callbacks in TL

Callback-Generierungsfunktion *cCalback.new* ein Callback-Objekt erzeugt. Die C-Funktion *testClickHdl*, die über einen Aufruf von *bind* auch als TL-Funktion zur Verfügung steht, bekommt dieses Callback-Objekt als Argument übergeben. In dieser C-Funktion erscheint das Callback-Objekt als C-Funktionszeiger und kann als solcher aufgerufen werden. Der Ablauf eines Aufrufs des TL-Pendant von *testClickHdl* ist in Abbildung 3.2 grafisch dargestellt. Dabei wird insbesondere der Wechsel zwischen den beteiligten Sprachebenen deutlich.

Externe Referenzen

Grundsätzlich ermöglicht die C-Schnittstelle die Übergabe von Referenzen auf TL-Objekte. Allerdings ist nicht garantiert, daß diese ihre Gültigkeit behalten, da zu jedem beliebigen Zeitpunkt die automatische Freispeicherverwaltung in Aktion treten kann (*garbage collection*), wodurch sich, implementationsbedingt, die Referenzen im gesamten Tycoon-Objektgraphen ändern können. Durch eine zusätzliche Indirektion kann man jedoch verhindern, daß dadurch externe Referenzen auf TL-Datenstrukturen ungültig werden. Dazu werden alle TL-Objekte, von denen eine Objektspeicherreferenz an C übergeben werden soll, in einer Tabelle verwaltet, deren Lebensdauer mindestens so groß ist wie die des Anwendungsprogramms. Übergibt man dann Referenzen auf Einträge in diese Tabelle an C, so ist man sicher, daß diese zur Laufzeit der Anwendung ihre Gültigkeit behalten.

Schwache Referenzen

Die meisten TL-Programme überlassen der automatischen Freispeicherverwaltung die Rückgewinnung des von ihnen nicht mehr referenzierbaren Speichers. Liegen diese Speicherbereiche allerdings in anderen Adreßräumen als dem TL-Objektspeicher, z.B. auf der C++-Halde, so

ist ein zusätzlicher Laufzeitmechanismus erforderlich, der für die Freigabe solcher Speicherbereiche verantwortlich ist. Dieser Mechanismus könnte durch *schwache Referenzen* implementiert werden, wie sie z.B. für die Laufzeitbibliothek von Modula-3 in [Horning et al. 93] vorgeschlagen werden (Schnittstelle *WeakRef*).

3.3.2 Wrapper-Funktionen

C++-Compiler erlauben es, Funktionen im C-Stil zu binden, also ohne die Codierung von Typinformation in Funktionsnamen. Dadurch lassen sich C++-Methoden von C-Programmen aus aufrufen. Dies bietet die Möglichkeit, die Implementation einer GDL-Spezifikation in folgender Weise zu realisieren:

- Methodenaufrufe werden in C-Funktionen eingehüllt.
- Die übersetzten C-Funktionen werden dynamisch zum Tycoon-Laufzeitsystem hinzugebunden.
- In den generierten TL-Modulimplementationen werden C-Funktionen mittels der *bind*-Primitive an TL-Funktionen gebunden.

Beispiele für *Wrapper*-Funktionen:

```
/* in GDL:  void Window::SetText(const String& rStr) */
extern "C" void window_SetText(Window* self, String* str)
    self->SetText(*str)

/* in GDL: static Size& System::GetScreenSizePixel() */
extern "C" Size* system_GetScreenSizePixel()
    return &System::GetScreenSizePixel()
```

Bei der Generierung der *Wrapper*-Funktionen sind die verschiedenen Parameterübergabe- und Wertrückgabemechanismen von C++ zu berücksichtigen. In C existiert die Möglichkeit der Übergabe oder Rückgabe von Referenzen nicht, stattdessen müssen Zeiger als Argumente übergeben oder als Wert zurückgegeben werden. Verlangt die eingehüllte C++-Methode eine Referenz als Argument, so ist der betreffende Zeiger zunächst zu dereferenzieren, bevor er der Methode übergeben wird (im Beispiel betrifft dies das Argument *str*). Gibt sie eine Referenz zurück, so ist die Adresse des betreffenden Objektes zurückzugeben.

3.3.3 Emulation von Overriding

Wie in Abschnitt 3.2.2 behandelt, werden virtuelle Methoden als Attribut einer Klasse aufgefaßt, für das ein Paar von Schreib-/Lesefunktionen erzeugt wird. Die Implementation dieser Funktionen bedient sich einer Tabelle, deren Einträge jeweils ein Objekt mit einem Methodennamen und einem Callback assoziieren. Von jeder Klasse, die virtuelle Methoden exportiert, wird eine Subklasse generiert, wobei jede dieser Methoden auf folgende Weise überschrieben wird:

- Aus der Callback-Tabelle wird über den Objektzeiger und den Methodennamen auf den vom Benutzer der Bibliothek installierten Callback (C-Funktionszeiger) zugegriffen und dieser, sofern vorhanden, ausgeführt.
- Ist kein Callback installiert, so wird die entsprechende Methode der Superklasse aufgerufen.

Vom externen Server getätigte Aufrufe virtueller Methoden lösen dadurch indirekt Aufrufe von TL-Funktionen aus. Abbildung 3.3 veranschaulicht diesen Mechanismus.

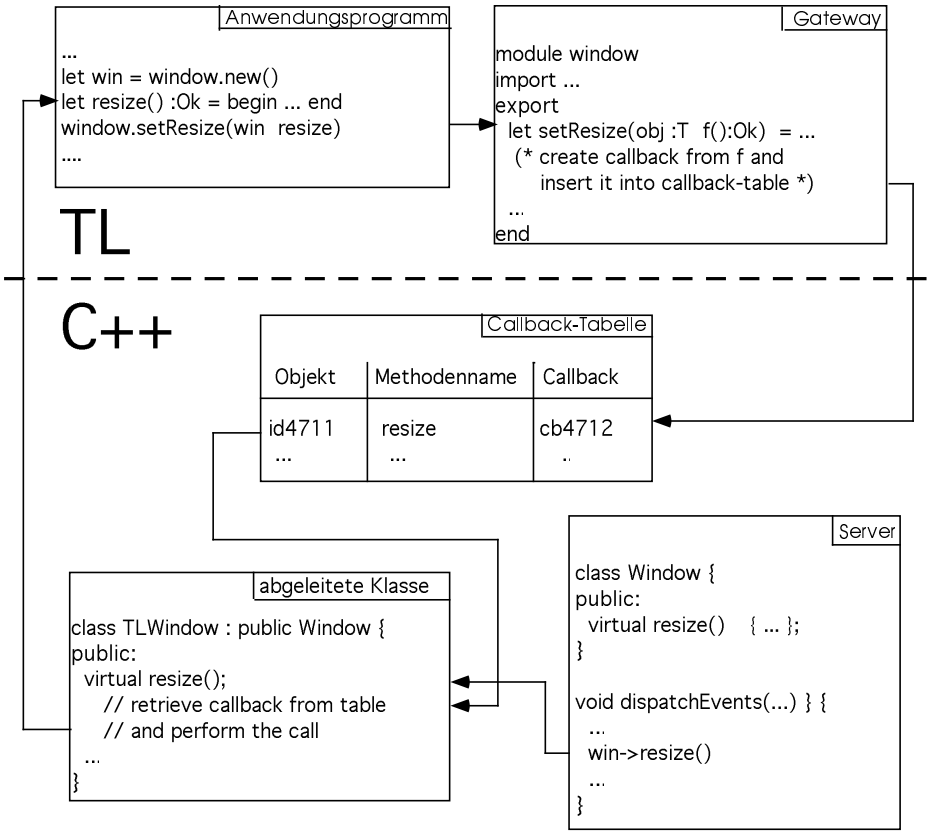

Abbildung 3.3: Emulation von Overriding

Probleme durch Mehrfachvererbung

Besitzt eine mit virtuellen Methoden ausgestattete Klasse weitere Subklassen, so kann man in C++ auch in abgeleiteten Klassen dieser Subklassen Methoden der Oberklasse überschreiben. Praktisch bedeutet dies, daß nicht nur von jeder Klasse, die virtuelle Methoden besitzt, eine von ihr abgeleitete Klasse generiert werden muß, sondern auch von allen ihren Subklassen, da man ja auch in diesen Methoden überschreiben können möchte (Abbildung 3.4). Diese Vorgehensweise führt allerdings bei tiefen Klassenhierarchien, bei denen schon auf oberster Ebene viele virtuelle Methoden vorkommen, die potentiell in vielen anderen Subklassen überschrieben werden können, zu umfangreichem Objekt-Code.

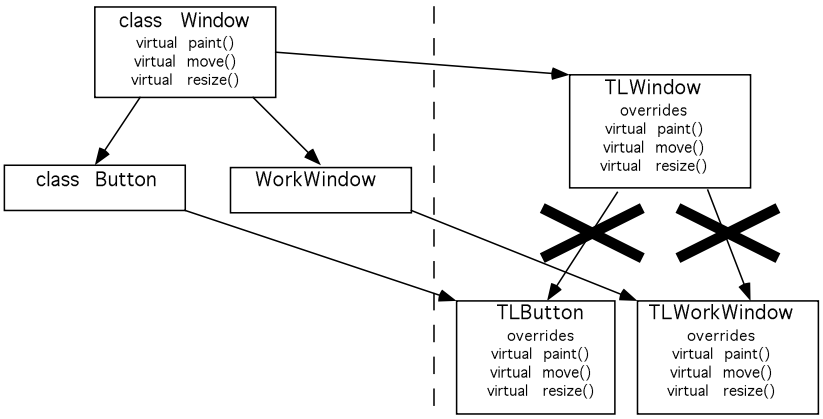

Abbildung 3.4: Overriding ohne Mehrfachvererbung

Wünschenswert wäre, wenn der Generator den Code für virtuelle Methoden nicht in jeder Subklasse neu erzeugen müßte, sondern stattdessen von der Möglichkeit der Mehrfachvererbung Gebrauch machen könnte, wie in Abbildung 3.5 dargestellt. Dies ist allerdings nicht

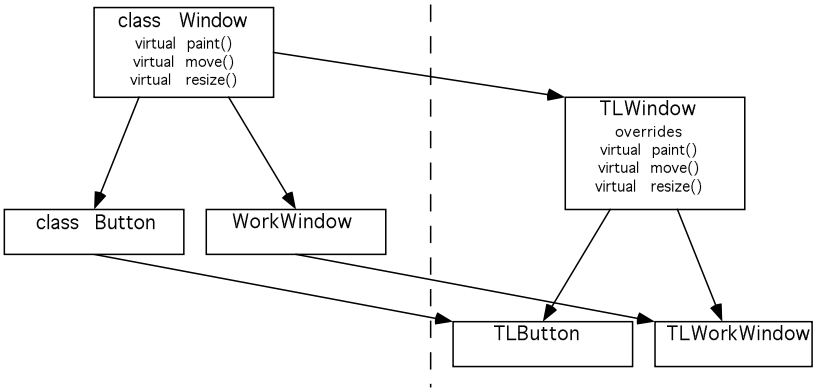

Abbildung 3.5: Overriding mit Mehrfachvererbung

möglich, da in C++ im Falle von Mehrfachvererbung der Wert eines Objektzeigers durch implizite oder explizite Typkonvertierung (*casting*) verändert wird[3]. Diese Umrechnung kann nur vom C++-Compiler selbst vorgenommen werden, da nur er das konkrete Speicher-Layout von Objekten kennt.

Da allerdings der Aufruf von C++-Methoden von TL aus über C-*Wrapper*-Funktionen geschieht, ist aus Sicht des C++-Compilers keine Notwendigkeit zur Umrechnung gegeben. Es kann also passieren, daß eine C++-Methode ein anderes Speicher-Layout eines Objektes erwartet als tatsächlich vorhanden ist, d.h. in C++ gilt in diesem Fall das Subsumptionsprinzip nicht. Diese Situation ist beispielhaft in Abbildung 3.6 dargestellt.

Eine Methode, implementiert in Klasse B, kann, angewandt auf ein Objekt der Klasse C, zu einem falschen Zugriff auf Instanzvariablen des Objektes führen, wenn nicht vorher der Betrag $delta(B)$ zum Objektzeiger addiert wird.

[3] "*With multiple inheritance, casting may change the value of a pointer.*" [Ellis, Stroustrup 90], S. 221

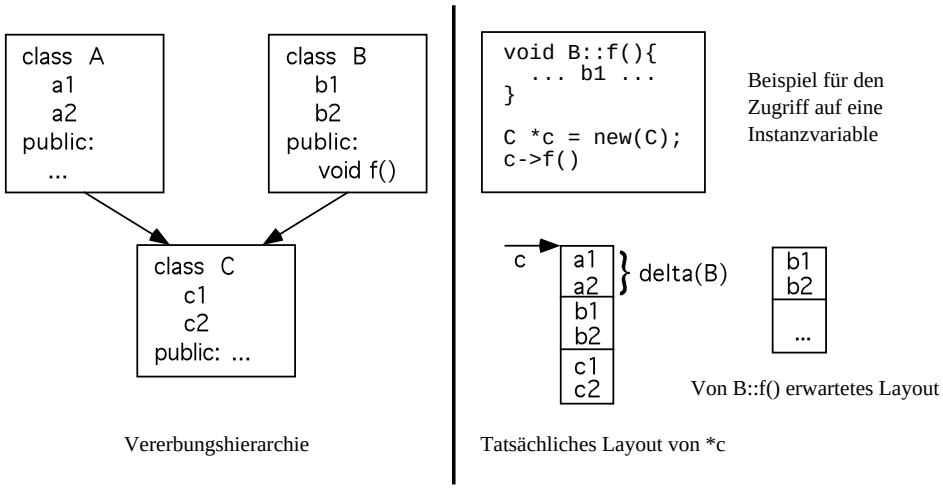

Abbildung 3.6: Falscher Zugriff auf Instanzvariablen bei Mehrfachvererbung

3.3.4 Architektur und Implementierung

Der Gateway-Generator ist in TL unter Nutzung folgender generischer Tycoon-Dienste implementiert worden:

- Tycoon-Compiler-Toolkit [Schröder, Matthes 92]: Parser-Generator auf Basis von LL(1)-Grammatiken,
- Generische *Unparser*-Schnittstelle zur Generierung formatierter Texte,
- Bulktyp- und Iterationsabstraktionen.

Der Parsing-Phase folgt eine Überprüfung aller Deklarationen, um sicherzustellen, daß nur korrekt übersetzbare TL-Module generiert werden. Treten hierbei keine Fehler auf, so werden die vorgefundenen Typdeklarationen als *Dictionary* von Typbeschreibungen in einer Datei gespeichert, um von anderen Spezifikationen importiert werden zu können. Die Typbeschreibungen haben folgende Struktur:

```
Let GDLType =
  Tuple
    moduleName :String              (* Name des TL-Modules, in dem der Typ zu
                                       definieren ist *)
    name, tlName :String            (* Namen des Typs in C++ bzw. TL *)
    isAbstract :Bool                (* abstrakter Typ? *)
    passingMode :String             (* Parameterübergabemodus für  bind *)
   case classCase with
    virtuals :list.T(GDLsyntaxTree.Header)      (* Liste aller virtuellen
                                       Methodendeklarationen, incl. geerbter *)
    isForward :Bool                 (* Forward-Deklaration? *)
   case baseCase, enumCase, typedefCase, defineCase
  end
```

Sie enthält Information, die sowohl zur Typüberprüfung als auch zur Generierung benötigt wird.

Zur Überprüfung der Deklarationen wird zunächst anhand der Import-Liste der zu übersetzenden Spezifikation eine Übersetzungsumgebung aufgebaut, die alle in den importierten Spezifikationen definierten Typbeschreibungen enthält. Es werden folgende Überprüfungen vorgenommen:

- Modulglobale Bezeichner (Typnamen, Methodennamen, Namen von Enumeratorelementen) dürfen nicht mit TL-Schlüsselwörtern oder dem jeweiligen Modul-/Schnittstellenbezeichner kollidieren. Ferner dürfen sie nicht redeklariert werden.
- Klassendeklarationen:
  - Alle genannten Basisklassen müssen deklariert sein.
  - Alle in Methoden verwendeten Bezeichner müssen sichtbar sein.
  - Eine Überdeckung von Modulbezeichnern durch Parameterbezeichner muß ausgeschlossen sein[4].
- **typedef**: Der angegebene Typbezeichner muß deklariert sein.
- Zu jeder Forward-Deklaration muß eine Definition existieren.

Abbildung 3.7 gibt einen Überblick über die Struktur des Generators. Kästchen symbolisieren Module mit deren Schnittstellen, wobei Verbindungslinien Import-Beziehungen zwischen Modulen darstellen (höheres Modul importiert tieferliegendes). Ellipsen repräsentieren reine Datenobjekte und Pfeile entsprechend Datenflüsse.

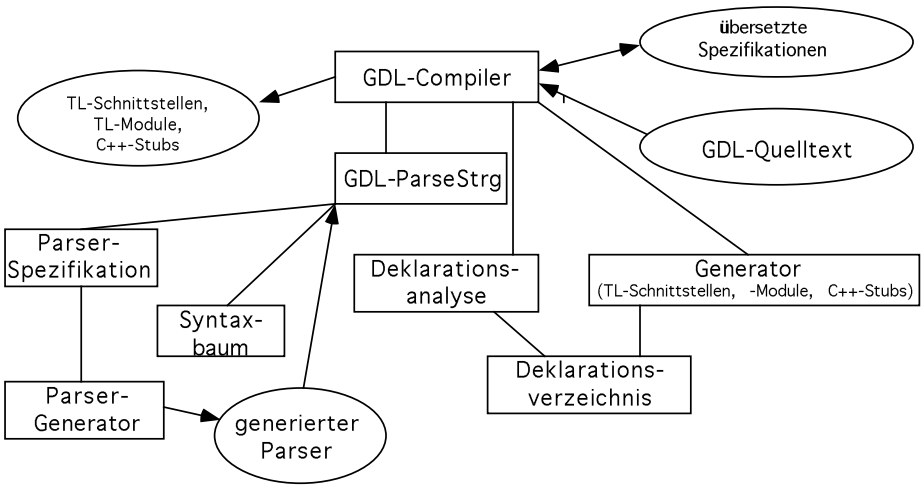

Abbildung 3.7: Architektur des Gateway-Generators

[4] siehe Beispiel auf Seite 27

4. GUI-Dienste

Durch die immer größere Leistungsfähigkeit der Hardware, insbesondere was die Verfügbarkeit von Arbeitsspeicher und Prozessorleistung angeht, sind die Möglichkeiten zur Implementierung grafischer Benutzerschnittstellen (*graphical user interface - GUI*) in den letzten Jahren stark gestiegen [Mandelkern 93]. Spezialisierte Bibliotheken machen die grafischen Leistungen der in modernen Arbeitsplatzrechnern benutzten Fensterbetriebssysteme (z.B. Motif, Windows, Presentation Manager) auf einer höheren Abstraktionsebene für den Anwendungsprogrammierer zugänglich. Dadurch wird die Komplexität der GUI-Programmierung und somit der Gesamtaufwand für die Anwendungsentwicklung reduziert.

Problematisch ist allerdings, daß die Schnittstellen dieser Bibliotheken nicht immer zur Anwendungsprogrammiersprache passen [Dearle 88], da sie meist nur für eine bestimmte Sprache angeboten werden. Der Programmierer muß dann im Extremfall nicht nur eine zusätzliche Sprache beherrschen, sondern sich auch um deren Einbindung in die Anwendungsprogrammiersprache kümmern, was die Komplexität der Aufgabe abermals steigert. Durch das im letzten Kapitel vorgestellte generische Gateway wird dem Anwendungsprogrammierer jedoch dieser zusätzliche Aufwand abgenommen. Er bekommt darüberhinaus die Möglichkeit, die Vorzüge einer interaktiven persistenten Umgebung für die GUI-Programmierung zu nutzen. Auf diese Vorzüge müßte er eventuell verzichten, wenn seine Entscheidung für eine Applikationsprogrammiersprache von der Verfügbarkeit von GUI-Bibliotheken abhinge.

In diesem Kapitel wird am Beispiel der Einbindung der GUI-Klassenbibliothek StarView^{TM} [Busch et al. 93] eine Anwendung des C++-Gateway-Generators beschrieben. Dabei zeigt sich, daß die Nutzung dieser Bibliothek von Tycoon aus eine Qualitätsverbesserung aus zwei unterschiedlichen Sichten mit sich bringt: einerseits aus der Sicht des Klienten (in diesem Fall der Tycoon-Umgebung), dadurch, daß damit eine plattformübergreifende einheitliche GUI-Programmierschnittstelle zur Verfügung steht. Andererseits aus der Sicht der GUI-Programmierung, denn die Nutzung dieser Bibliothek von Tycoon aus ist durch interaktive Übersetzung und Auswertung, Persistenz und *multi threading* flexibler als deren direkte Verwendung in C++-Anwendungen.

4.1 Konzepte von StarView

StarView ist eine C++-Klassenbibliothek, die es erlaubt, bei der Anwendungsentwicklung von der Gestaltung der grafischen Benutzeroberfläche des jeweiligen Fenstersystems zu abstrahieren, um dadurch die Portabilität von GUI-Anwendungen zu erhöhen. Kennzeichnend für grafische Benutzeroberflächen ist einerseits das Erscheinungsbild und die Funktionalität der Fenster und Interaktionselemente (*look*) und andererseits das Verhalten des Systems (*feel*) auf

Interaktionen des Anwenders. Die Klassenhierarchie dieser Bibliothek ist so gestaltet, daß das *look and feel* des jeweilig zugrundeliegenden Fenstersystems weitgehend beibehalten wird und nur bei den Eigenschaften und Funktionen Anpassungen vorgenommen werden, bei denen sich die Fenstersysteme graduell unterscheiden. Es wird also eine Mischung aus dem *Schnittmengenansatz* und dem *Obermengenansatz* verfolgt [Müßig 94], da nicht alle nichtvorhandene Funktionalität eines Fenstersystems durch StarView nachgebildet wird[1] Diese Mischform wirft allerdings die Frage nach dem Grad der Portabilität von StarView-Anwendungen auf, da GUI-Anwendungen nur dann portabel sind, wenn der Programmierer darauf geachtet hat, nur Funktionalität zu benutzen, die auf allen unterstützten Systemen verfügbar ist.

Einen Eindruck von Aufbau und Umfang der Bibliothek soll Abbildung 4.1 verschaffen.

4.1.1 Typische GUI-Elemente

Aus der Vielzahl der angebotenen Klassen sollen in diesem Abschnitt exemplarisch die Eigenschaften der wichtigsten GUI-Elemente beschrieben werden.

Fenster: Sie dienen der Präsentation der Anwendungsdaten und der Entgegennahme der Eingaben des Benutzers[2] durch die im Ausgabebereich des jeweiligen Fensters dargestellten Interaktionselemente. Nur Fenster, die den *focus* besitzen, können Eingaben entgegennehmen. Sie stehen über eine Vater-Sohn-Beziehung (festgelegt bei der Erzeugung, dynamisch rekonfigurierbar) zueinander in einem Abhängigkeitsverhältnis. Systemfenster (*class SystemWindow*) sind unabhängig von ihrem Vater-Fenster verschiebbar, während alle anderen Fenster nur relativ zum Vater-Fenster positioniert werden können. Arbeitsfenster können vom Benutzer geschlossen und dadurch als Piktogramm (*icon*) auf dem Bildschirmhintergrund dargestellt werden.

Interaktionselemente (controls): Dies sind die vom Anwender durch Maus und Tastatur manipulierbaren Schnittstellenelemente. Sie umfassen Knöpfe, Auswahllisten, Eingabefelder, Menüs etc. Mit diesen Elementen können Funktionen verknüpft werden, die bei bestimmten Benutzerinteraktionen (z.B. Maus-Klick auf einen Knopf) aufgerufen werden.

Dialoge: Durch Dialoge werden Interaktionselemente zusammengefaßt, um umfangreiche Benutzereingaben zu unterstützen. Sie steuern den Focus-Wechsel zwischen Interaktionselementen und ermöglichen dadurch ihre geordnete Bearbeitung. Es wird zwischen *modalen* und *nichtmodalen* Dialogen unterschieden, wobei modale Dialoge im Gegensatz zu den nichtmodalen beendet werden müssen, bevor in der Anwendung fortgeschritten werden kann.

Datenaustausch: Der Transfer von Daten zwischen Anwendungen, bzw. fensterübergreifend innerhalb einer Anwendung wird durch standardisierte Verfahren unterstützt. Diese umfassen das Klemmbrett (*clipboard*) sowie das *drag and drop protocol*. Das Klemmbrett dient als Zwischenspeicher für Daten, von denen Kopien (evtl. für andere Anwendungen) angefordert werden können. *Drag and drop* ist ein verwandtes Verfahren, wobei

[1] Ein Beispiel hierfür ist die Möglichkeit der Ikonifizierung von Fenstern in manchen Systemen, die auf dem Apple Macintosh nicht existiert und in der StarView-Implementation für dieses System auch nicht nachgebildet wird.

[2] Außerhalb von Fenstern sind keine Eingaben möglich.

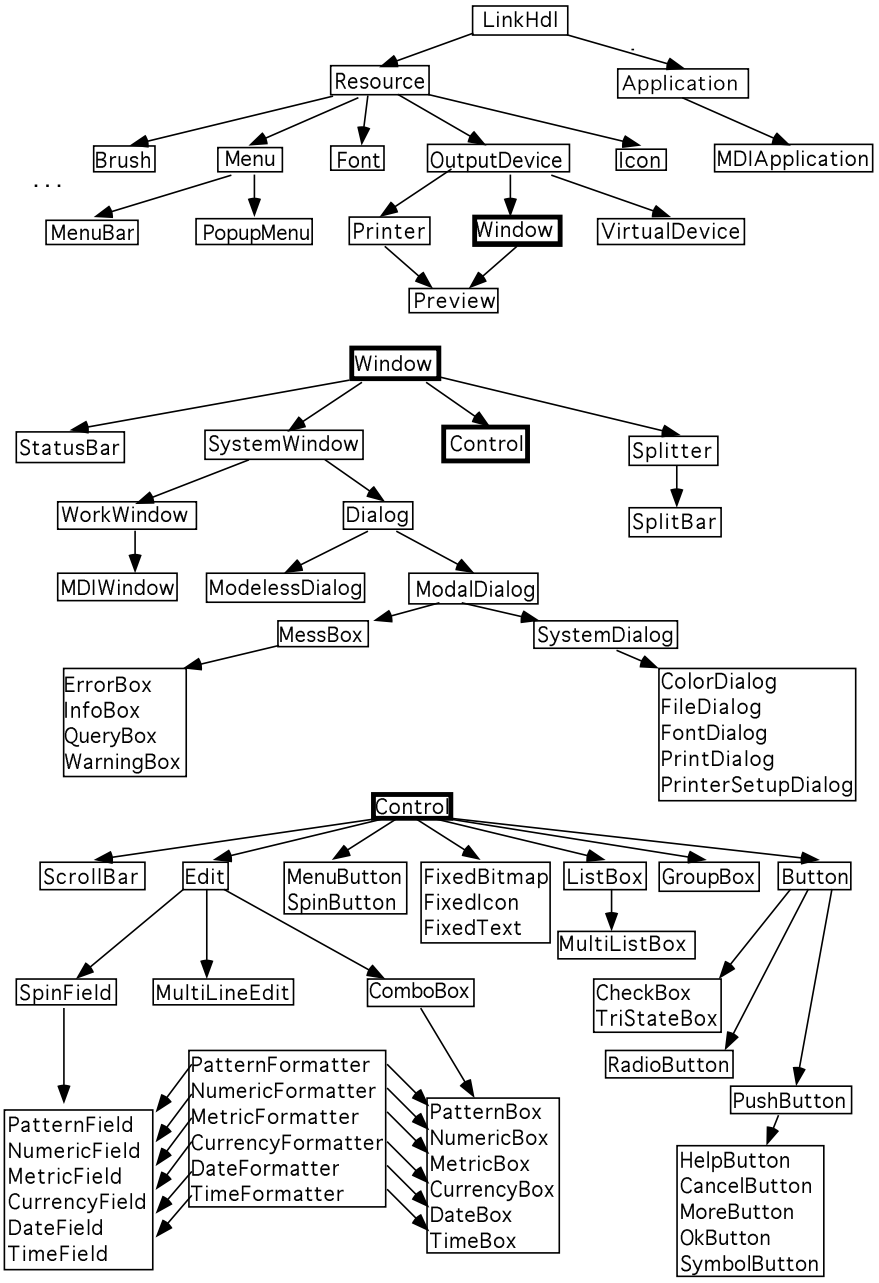

Abbildung 4.1: Klassenhierarchie von StarView

der Benutzer Datenkopien durch gleichzeitiges Drücken einer Maustaste und Bewegung des Mauszeigers auf ein Ziel anfordern kann. Loslassen der Maustaste beendet diesen Vorgang.

4.1.2 Ereignisbearbeitung

Bei Auftreten bestimmter Ereignisse werden von StarView benutzerdefinierte Funktionen (*event handler*) aufgerufen. Damit erhält der Benutzer die Möglichkeit, programmtechnisch

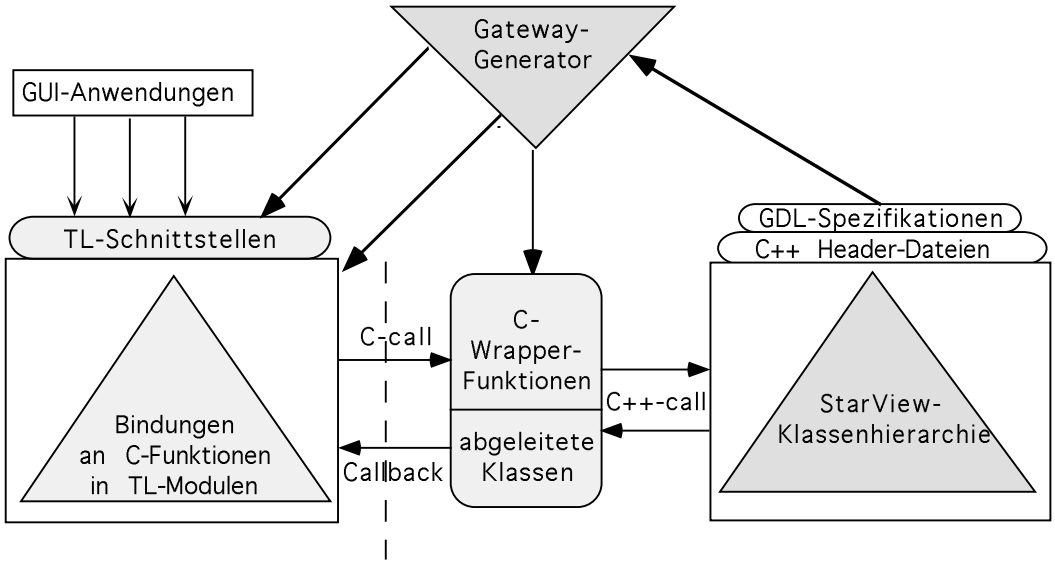

Abbildung 4.2: Anbindungsszenario für StarView

auf Ereignisse zu reagieren und die Anwendung entsprechend zu steuern. Die Assoziation eines *event handler* mit einem Ereignis erfolgt in C++ entweder durch überschreiben virtueller Funktionen oder durch einen speziellen, von StarView zur Verfügung gestellten Mechanismus. Die Ereignisverarbeitung mit Aufruf der jeweilig mit einem bestimmten Ereignis assoziierten Funktion/Methode erfolgt durch das Fenstersystem selbst. In Abbildung 3.2 auf Seite 29 wurde der Vorgang der Ereignisbearbeitung in Zusammenhang mit dem Tycoon *Callback*-Mechanismus veranschaulicht.

4.2 Durchführung der Anbindung

In diesem Abschnitt wird der Einsatz des Gateway-Generators zur Anbindung der StarView-Bibliothek beispielhaft demonstriert. Konkretisiert auf diesen Anwendungsfall wird das Anbindungsszenario anhand von Abbildung 4.2 veranschaulicht.

Die Durchführung der Anbindung verläuft dabei in drei Phasen:

1. Erstellung der GDL-Schnittstellenbeschreibungen

Die Klassendeklarationen von StarView liegen in Form mehrerer *Header*-Dateien vor. Diese bilden die Grundlage zur Erstellung der erforderlichen GDL-Spezifikationen zur Schnittstellengenerierung, wie in Abschnitt 3.1.1 auf Seite 17ff beschrieben. Die Aufteilung der Klassen auf GDL-Dateien ist prinzipiell frei wählbar. Zu beachten ist jedoch, daß wechselseitig rekursiv definierte Klassen in derselben GDL-Spezifikation definiert werden, da Module in TL nicht rekursiv definiert sein können. Ferner ist die Möglichkeit von Namenskonflikten je größer, desto mehr Klassen in derselben Datei spezifiziert werden, da Klassen in GDL keinen Namensraum darstellen. Ein Beispiel für eine typische GDL-Spezifikation:

```
gateway "C++" scrollBar :ScrollBar
import window control resId range
export
  enum ScrollType{SCROLL_DONTKNOW, SCROLL_LINEUP,
          SCROLL_LINEDOWN, SCROLL_PAGEUP, SCROLL_PAGEDOWN}

 class ScrollBar=>T : public Control
 {
    constructor ScrollBar=>new(Window* pParent)

    virtual void    Scroll()
    ...
    Range           ChangeRange(const Range& rRange=>r)
    Range           GetRange() const
    ...
    ScrollType      GetType() const
 }
end
```

2. Generierung

Die Generierung der TL-Module und -Schnittstellen sowie der C++-Module erfolgt aufgrund des großen Umfanges der Bibliothek werkzeugunterstützt: Eingabedateien für das Unix-Werkzeug *make* werden aus den Import-Listen der GDL-Spezifikationen automatisch erzeugt. Eine GDL-Spezifikation kann erst dann übersetzt werden, wenn alle von ihr importierten Spezifikationen bereits übersetzt sind.

Vor der Übersetzung der generierten TL-Module ist eine statische Beschreibung der Bibliothek (TL-Konstrukt *library*) zu erstellen, die Sichtbarkeitsbeziehungen der beteiligten Module und Schnittstellen untereinander beschreibt. Auch diese kann aus den Import-Listen der GDL-Spezifikationen automatisch erzeugt werden.

Aus obiger GDL-Spezifikation wird folgende TL-Schnittstelle generiert:

```
interface ScrollBar
import optional :Window link range resId window control
export
 ScrollType <:Int
 SCROLL_DONTKNOW, SCROLL_LINEUP, ... (* etc. *) SCROLL_DRAG :ScrollType
 T <:control.T
 new(parent :optional.T(window.T)) :T
 scroll(self :T) :Ok
 setScroll(self :T  scroll() :T) :Ok
 getScroll(self :T) :Fun() :T
 ...
 changeRange(self :T r :range.T) :range.T
 getRange(self :T) :range.T
 ...
```

```
  getType(self :T) :ScrollType
  deleteT(self :T) :Ok
end
```

Zu dieser Schnittstelle gehört folgende Implementation:

```
module scrollBar
import optional :Window link range resId window control word cCallback
  gdlObject
export
  let lib = "svenv/svenv" (* wird vom Laufzeitsystem benutzt *)

  Let ScrollType = Int
  let SCROLL_DONTKNOW = 0  let SCROLL_LINEUP = 1 ... (* etc. *)

  Let T = control.T
  let new = bind(:Fun(parent :word.T) :T lib "scrollBar_new" "ww")
  let new(parent :optional.T(window.T)) :T =
    if optional.null(parent) then new(word.nil)
    else new(optional.value(parent))
    end
  let scroll = (* Bindung an die Default-Implementation *)
    bind(:Fun(self :T) :Ok lib "scrollBar_Scroll" "wv")
  let setScroll(self :T scroll() :Ok) :Ok =
    gdlObject.setHandler(self "scroll" cCallback.new(scroll "v"))
    (* Trägt den Callback in die globale Callback-Tabelle ein. *)
  let getScroll(self :T) :Fun() :Ok =
    cCallback.get(:Fun() :Ok gdlObject.getHandler(self "scroll"))
    (* Extrahiert die Funktion, die in der Callback-Tabelle unter dem Namen
       "scroll" und dem Objekt 'self' gefunden wurde. *)
  ...
  let changeRange =
    bind(:Fun(self :T r :range.T) :range.T lib "scrollBar_ChangeRange" "www")
  let getRange = bind(:Fun(self :T) :range.T lib "scrollBar_GetRange" "ww")
  ...
  let getType = bind(:Fun(self :T) :ScrollType lib "scrollBar_GetType" "wi")
  let deleteT = bind(:Fun(self :T) :Ok lib "scrollBar_deleteT" "wv")

  (* benutzte Parameterübergabemodi:
      Format   C-Typ    TL-Typ
      ------------------------
      w        void*    32-Bit-Wort
      i        long     Int
      v        void     Ok (nur als Rückgabewert)
   *)
end
```

Die in der Implementation benutzten C-Funktionen werden folgendermaßen generiert:

```
/* scrollBar.cxx */

class TyScrollBar : public ScrollBar
{
public:
  TyScrollBar(Window* pParent) : ScrollBar(pParent) {};

  typedef void (*scrollFunc)();
  ... /* folgen Typdefinitionen für Funktionspointer der geerbten virtuellen
         Funktionen */

  virtual void Scroll()
  {
    scrollFunc scroll;
    scroll = (scrollFunc) gdlobj_getHandler(this, "scroll");
    /* Aufsuchen der Callback-Funktion aus der globalen Callback-Tabelle */
    if (scroll) return (*scroll)();
    else return ScrollBar::Scroll();
  };
  ... /* folgen Redefinitionen für alle geerbten virtuellen Funktionen */
};
extern "C" TyScrollBar* scrollBar_new(Window* pParent) {
  return new TyScrollBar(pParent);
}
extern "C" void scrollBar_Scroll(ScrollBar* self) {
  self->ScrollBar::Scroll();
}
...
extern "C" Range* scrollBar_ChangeRange(ScrollBar* self, Range* rRange) {
  return new Range(self->ChangeRange(*rRange));
}
extern "C" Range* scrollBar_GetRange(ScrollBar* self) {
  return new Range(self->GetRange());
}
...
extern "C" ScrollType scrollBar_GetType(ScrollBar* self) {
  return self->GetType();
}
extern "C" void scrollBar_deleteT(ScrollBar* self) {
  delete self;
}
```

3. Übersetzung

Die übersetzten C++-Module werden in einer *shared dynamic link library* (DLL) zusammengefaßt. Aus den der *bind*-Primitive übergebenen Parametern (Funktionsname, Bibliotheksname, Parameterformate) kann das Laufzeitsystem entsprechende Aufrufe an die DLL erzeugen,

ohne daß es selbst geändert werden muß. Schließlich werden die TL-Module anhand der Bibliotheksspezifikation (*library*-Konstrukt) übersetzt.

Änderungen an GDL-Spezifikationen erfordern einen erneuten Durchlauf dieser Phasen. Eine erneute Übersetzung der abhängigen TL-Schnittstellen und -Module ist nur dann notwendig, wenn sich die Klassenhierarchie ändert. Alle anderen Änderungen, z.B. die Erweiterung von Signaturen, machen, im Gegensatz zu „normalen" TL-Modulen, keine Neuübersetzung der abhängigen Module notwendig, da exportierte Funktionen lediglich ihre Parameter an den externen Diensterbringer weiterleiten und niemals TL-Funktionen aufrufen. Die Modulverwaltung unterscheidet jedoch nicht zwischen Modulen, die externe Funktionalität benutzen und solchen, die dies nicht tun, so daß eine Neuübersetzung gemäß den normalen Abhängigkeitsregeln trotzdem vorgenommen wird.

Multi-Threading

GUI-Anwendungen verlaufen üblicherweise nach folgendem Schema:

- Festlegung der statischen Struktur der grafischen Interaktionselemente (Menüs, Dialog, Arbeitsfenster etc.),
- Definition der *event handler*,
- Start der Ereignisverarbeitung.

Nach dem Starten der Ereignisverarbeitung sind, trotz der Möglichkeit zur inkrementellen Übersetzung, Programmänderungen nicht durchführbar, ohne die Anwendung zu verlassen, da sich Änderungen nur bei gestarteter Ereignisverarbeitung auswirken.

Durch die Möglichkeit in Tycoon, mehrere Kontrollflüsse (*threads of control*) auf demselben Zustand operieren zu lassen [Matthes, Schmidt 94], läßt sich dieser Nachteil ausgleichen. Die Ereignisverwaltung (*event loop*) wird dazu in einem eigenständigen *thread* gestartet, dem *event loop thread*. In einem parallel zum *event loop thread* laufenden *thread* (z.B. dem TL *top level*) kann dann durch inkrementelle Übersetzung die Applikation dynamisch modifiziert werden, wobei die Auswirkungen dieser Änderungen sofort sichtbar und testbar sind.

Diese Funktionalität macht die interaktive Erzeugung von Benutzerschnittstellen potentiell komfortabler als bei herkömmlichen Werkzeugen (*interface builder*, z.B. SUN's *Developer's Graphical User Interface Design Editor* [Sun 92] oder dem *Design Editor* von Star Division [Busch et al. 93]), da auch die Funktionalität der *event handler* iterativ entwickelt und getestet werden kann. Diese Eigenschaft ist deshalb so wertvoll, weil der weitaus größte Teil des Programmcodes von Anwendungen mit grafischer Benutzerschnittstelle eine Reaktion auf Ereignisse darstellt. Zu einer kompletten *visuellen Programmierumgebung* wie z.B. VisualBASIC [Maslo, Dittrich 93] fehlt jedoch noch grafische Unterstützung zur direkten Manipulation von GUI-Elementen, z.B. zur Erzeugung und Positionierung von Knöpfen, Menüs und Dialogen.

Gegenstand zukünftiger Projekte könnte der Aufbau einer visuellen Entwicklungsumgebung für Prototypen datenintensiver Anwendungen sein, aufbauend auf vorhandenen Diensten (StarView-Gateway, *persistent threads*). Dabei wäre zu erforschen, inwieweit sich diese Form der (semi-)visuellen Programmierung für das *prototyping* von Anwendungen eignet und wie sie in das bisherige Spektrum der Werkzeuge für Sprachen der vierten Generation (z.b. INGRES 4GL [Ingres 90]) einzuordnen ist.

5. Reflektive Dienste

Das Tycoon-System zeichnet sich besonders durch die vorhandenen generischen Dienste in seinen Bibliotheken aus, die ein hohes Maß an Wiederverwendung erlauben. Beispielhaft sei hier das Tycoon-*Compiler-Toolkit* [Schröder, Matthes 92] erwähnt, welches zur Implementierung des TL-Compilers verwendet wurde. In diesem Kapitel wird behandelt, auf welche Weise Dienste des Compilers selbst (z.B. parsing, statische und dynamische Typüberprüfung, Codegenerierung) in einer Bibliothek zur Verfügung gestellt werden müssen, um streng typisierte reflektive Programmierung in TL zu ermöglichen.

Nach einer allgemeinen Charakterisierung von Reflektion in Programmiersprachen und der Behandlung typischer Beispiele und Anwendungen wird auf eine der Grundvoraussetzungen zur reflektiven Programmierung in statisch typisierten Sprachen eingegangen, nämlich die *dynamische Typisierung*. Diese Spracheigenschaft ist Voraussetzung für die im folgenden vorgestellte typsichere Schnittstelle des aufrufbaren Compilers und wurde in Kooperation mit Hubertus Koehler[1] als Erweiterung des bisherigen TL-Compilers implementiert. Die Schnittstelle des aufrufbaren Compilers wiederum ist nicht nur Voraussetzung zur reflektiven Programmierung, sondern bildet darüberhinaus auch die Grundlage für Programmierwerkzeuge der Tycoon-Entwicklungsumgebung (siehe Kapitel 6). Das Kapitel schließt mit einer Betrachtung von Schnittstellen zu Compilern anderer Sprachen und einer Bewertung ihrer Eignung zur reflektiven Programmierung.

Charakterisierung von Reflektion

Programme mit der Fähigkeit, sich selbst zu inspizieren und zu verändern, werden als *reflektiv* bezeichnet [Kirby 92a]. Dies kann entweder durch Beeinflussung ihres Interpretationsvorganges (verhaltensbasierte Reflektion) oder durch Änderung der eigenen Repräsentation (linguistische Reflektion) erfolgen. Die Änderung der eigenen Repräsentation beruht dabei auf der Fähigkeit von Programmen, Programmrepräsentationen zu generieren (d.h. Programme als Daten zu behandeln), sie in ausführbaren Code zu transformieren und diesen in den eigenen Programmablauf zu integrieren (d.h. Daten als Programme zu behandeln).

Reflektive Fähigkeiten sind im Zusammenhang mit Assemblersprachen oder untypisierten höheren Sprachen eher als gefährlich einzustufen, da es keine wirksamen Mechanismen zur Einschränkung der Verhaltensänderung auf ein „sinnvolles“ Maß gibt. Typisierte Sprachen erlauben es jedoch, alle Modifikationen, die ein Programm sich selbst zufügt, auf Typkorrektheit zu überprüfen. Man spricht in diesem Fall auch von *typsicherer linguistischer Reflektion*. Diese kann sowohl zur Übersetzungszeit als auch zur Laufzeit initiiert werden. Einen Überblick über die Arten von Reflektion gibt Abbildung 5.1.

[1] Arbeitstitel seiner Diplomarbeit: Generische Daten- und Funktionsvisualisierung

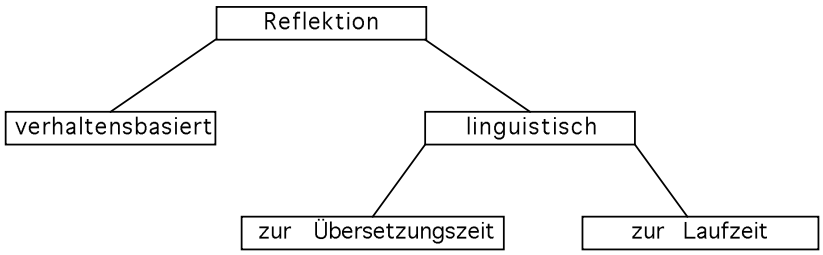

Abbildung 5.1: Ausformungen von Reflektion

Eine Programmiersprache um reflektive Fähigkeiten zu erweitern bedeutet nicht, ihre Ausdrucksmächtigkeit zu erhöhen, wenn diese Sprache ohnehin schon algorithmisch vollständig ist. Sie erlaubt jedoch eine generische und damit kürzere und besser wiederverwendbare Darstellung einiger im Bereich datenintensiver Anwendungen häufig gebrauchter Algorithmen, auf die im Verlauf dieses Kapitels noch näher eingegangen wird. Ein Beispiel für die Erhöhung von Ausdrucksmächtigkeit durch Reflektion ist die in [Bussche et al. 92] beschriebene Erweiterung der Relationenalgebra um einen Evaluationsoperator, mit dem in Relationen enthaltene Programmrepräsentationen ausgeführt werden können.

In dieser Arbeit besteht ein zyklischer Zusammenhang zwischen Reflektion zur Übersetzungszeit und Reflektion zur Laufzeit: zur Übersetzungszeit ausgeführte Funktionen werden zur Implementierung dynamischer Typen benutzt. Diese ermöglichen eine streng typisierte Schnittstelle eines aufrufbaren Compilers. Diese Schnittstelle wiederum kann in der Implementation des Compilers (in einer weiteren Bootstrap-Iteration) verwendet werden, um zur Übersetzungszeit durch einen rekursiven Aufruf „an sich selbst“ Werte zu berechnen. Dadurch wird eine allgemeinere Form von Reflektion zur Übersetzungszeit möglich, als diejenige, die zur Implementierung dynamischer Typen in „hart verdrahteter“ Weise eingesetzt wird.

5.1 Dynamische Typen zur reflektiven Programmierung

Ziel der Entwicklung höherer Programmiersprachen ist die Bereitstellung immer mächtigerer Abstraktionsmechanismen, um eine im hohen Maße *generische* Programmierung zu ermöglichen. Im einzelnen bedeutet dies, daß der Programmierer durch Wiederverwendung existierender Codefragmente und durch Automatisierung gewisser Teile der Codierungsarbeit immer weniger Code selbst schreiben muß. Die verschiedenen Arten des *Polymorphismus* stellen dabei Schlüsselkonzepte bzw. Säulen der generischen Programmierung dar (siehe Abb. 5.2).

Parametrischer Polymorphismus dient zur Formulierung von Funktionen, die *typunabhängige* Berechnungen durchführen.

Subtyppolymorphismus unterstützt den Aufbau von *Typhierarchien.*

Reflektiver Polymorphismus (Ad-hoc-Polymorphismus) erlaubt die Formulierung von Funktionen, die *typgesteuerte* Berechnungen durchführen.

Die Benutzung der ersten beiden Säulen ist in [Matthes et al. 94] beschrieben, während Benutzung und Implementierung der dritten Säule Gegenstand dieses Kapitels sind.

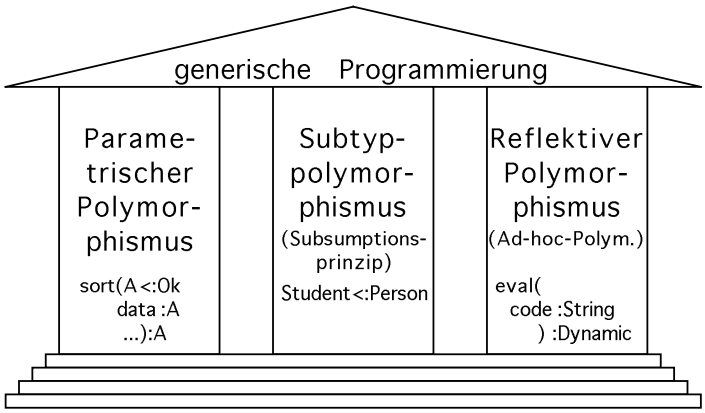

Abbildung 5.2: Säulen der generischen Programmierung

Subtyppolymorphismus (Inklusionspolymorphismus) und parametrischer Polymorphismus werden in [Cardelli, Wegner 85] unter dem Begriff *universeller Polymorphismus* zusammengefaßt. Durch ihn wird im Kontext statisch typisierter Sprachen eine ähnliche Flexibilität erreicht, wie sie vorher nur in dynamisch typisierten Sprachen möglich war. Reflektiver Polymorphismus als Form des Ad-hoc-Polymorphismus jedoch erlaubt es dem Programmierer, Funktionen mit einem (noch) höheren Grad an Generik zu schreiben als demjenigen, der durch universellen Polymorphismus erreichbar ist. Dadurch werden dem Programmierer größere Möglichkeiten zur Wiederverwendung von Code ermöglicht. Gerade im Bereich datenintensiver Anwendungen, die durch einen hohen Anteil schematischer und repetitiver Programmierung charakterisiert sind, ist diese höhere Generik für eine Reihe häufig verwendeter Funktionen erwünscht [Stemple et al. 90]. Aus Implementierungssicht ist der Hauptunterschied zwischen universellen und ad hoc-polymorphen Funktionen derjenige, das erstere für jeden Argumtenttyp jeweils denselben Code, letztere hingegen unterschiedlichen Code ausführen.

5.1.1 Anwendungen dynamischer Typisierung

Manche Programmiersituationen machen dynamische Typisierung unumgänglich. Diese sind dadurch charakterisiert, daß der Kontext, in dem ein Wert erzeugt wurde, und der Kontext, in dem er benutzt wird, über keine gemeinsame statisch überprüfbare Spezifikation verfügen. Das für diese Arbeit wichtigste Beispiel zur Verdeutlichung dieses Sachverhaltes ist die Funktion *eval*, die in ihrer einfachsten Form als Parameter eine Zeichenkette übernimmt, diesen als Ausdruck der Sprache auswertet und das Ergebnis zurückliefert.

eval(sourceText :String) :???

Wie durch die Fragezeichen angedeutet, läßt sich das Ergebnis dieser Funktion innerhalb des Aufrufkontextes nicht statisch typisieren. Der Typ des Rückgabewertes kann erst zur Ausführungszeit festgestellt werden, da er vom dynamisch gebundenen Quelltext abhängt.

Weitere Anwendungen dynamischer Typen umfassen:

- Datenaustausch mittels (standardisierter) linearer Datenrepräsentationen,
- Datenaustausch mit strukturierten Datenspeichern (*repositories*),

- Generierung, Konvertierung und Vermittlung (*trading)* von Schnittstellenbeschreibungen (siehe Kap. 3.1.2),
- Bindung an externen Code: RPC *Stubs* [da Silva 95], Dynamic Link Libraries,
- Typgesteuerte, generische Algorithmen: Da die Menge der Typkonstruktoren in jeder Programmiersprache begrenzt ist, im Gegensatz zur Menge der definierbaren Typen, lassen sich mit Hilfe von Laufzeit-Typinformation für folgende Anwendungen generische rekursive Bibliotheksfunktionen zur Verfügung stellen, die ansonsten vom Anwendungsprogrammierer implementiert werden müßten:
  - Generische Datenvisualisierung [Müßig 94]: Hierbei wird die in der objektorientierten Programmierung übliche Implementierung jeweils einer Visualisierungsmethode pro Klasse durch die Implementierung einer einzigen generischen Funktion ersetzt, welche anhand der übergebenen Laufzeit-Typrepräsentation entscheidet, wie die Ausgabe gestaltet werden muß.
  - Erzeugung von *Default*-Werten für beliebige Typen: Da in TL keine uninitialisierten Wertbindungen möglich sind, ist in einigen Programmiersituationen ein generischer Konstruktor hilfreich. Dies trifft vor allem bei Benutzung variabler Wertbindung im imperativen Programmierstil zu, insbesondere, wenn komplexe Strukturen initialisiert werden müssen.
  - Test auf strukturelle Gleichheit beliebig strukturierter Werte: Anders als die Identitätsfunktion (TL-Basisprimitive) kann der meist sehr aufwendige tiefe Vergleich von Baumstrukturen durch einen typgesteuerten Algorithmus als Bibliotheksdienst angeboten werden.

5.1.2 Laufzeit-Typrepräsentationen und automorphe Werte

Typrepräsentationen, die der Compiler lediglich als Hilfsstrukturen zur statischen Typüberprüfung benutzt, werden zur Laufzeit nicht mehr benötigt, da nach erfolgreicher Typüberprüfung das Programm als statisch korrekt angesehen werden kann [Matthews 87]. Zur Durchführung typgesteuerter Berechnungen ist es jedoch notwendig, Typen zur Laufzeit als *Wert* der Sprache zu repräsentieren. Dieser Wert wird in der Literatur [Matthews 87; Abadi et al. 92; Leroy, Mauny 91] meistens als *dynamischer Typ* bezeichnet, obwohl diese Bezeichnung eher irreführend ist, da es sich ja tatsächlich um einen *Wert* handelt, der einen statischen Typ repräsentiert.

Sei T ein Typ in einem statischen Kontext S, so soll mit $t_{S,T}$ dessen Repräsentation zur Programmlaufzeit bezeichnet werden.

Aussagen über statische Typen sind, bedingt durch die statischen Sichtbarkeitsregeln in TL, immer kontextabhängig. Charakteristisch für dynamische Typen ist hingegen, daß sie ihren Kontext gewissermaßen „in sich“ tragen, d.h. es gibt keinen allen dynamischen Typen gemeinsamen Kontext außer dem initialen Kontext, in dem z.B. die Basistypen definiert sind.

Der in der Praxis häufigste Fall ist die Benutzung von Laufzeit-Typrepräsentationen im Zusammenhang mit einem Wert, der durch sie beschrieben wird. Solch eine Aggregation, bestehen aus einem beliebigen Wert und dessen Typrepräsentation, wird als *automorpher* (selbstbeschreibender) Wert [Cardelli 89] oder auch als *dynamic* bezeichnet [Abadi et al. 89; Cardelli 86; Leroy, Mauny 91].

Ein *automorpher Wert* ist ein Paar $(x, t_{S,T})$, so daß x vom Typ T im statischen Kontext S ist.

5.1.3 Spracherweiterungen von TL

Das bisherige Konzept für dynamische Typen wurde in Abschnitt 2.2.2 erläutert. Abweichend davon wird in dieser Arbeit ein zur Übersetzungszeit reflektiver Ansatz gewählt, um Laufzeit-Typrepräsentationen für algorithmisch vollständige Berechnungen zugänglich zu machen und dabei dem Umstand Rechnung zu tragen, daß das Typsystem von TL höherer Ordnung ist.

Folgende Spracherweiterungen sind zur Implementierung dieses Ansatzes notwendig:

1. Neue Basistypen *typeRep_T* und *dynamic_T* für Laufzeit-Typrepräsentationen bzw. automorphe Werte.

2. Zu diesen Basistypen gehörige Konstruktoren (*typeRep_new* und *dynamic_new*).

3. Eine inverse Operation zu *dynamic_new*, *dynamic_be* genannt, die zur Laufzeit aus einem automorphen Wert die Wertkomponente extrahiert, falls dessen Typkomponente ein Subtyp des als Argument übergebenen Typs ist.

Diese Anweisungen lassen sich aus zwei Gründen nicht als Bibliotheksfunktionen implementieren[2]: Erstens benötigen sie den Zugriff auf interne Datenstrukturen des Compilers und zweitens sollen sie auf *beliebige* Typen anwendbar sein. Somit sind sie so generisch, daß sich ihre Signaturen nicht im TL-Typsystem formulieren lassen. In einer Pseudo-Notation hingegen ließen sie sich folgendermaßen ausdrücken:

*typeRep_new(T ::**ANYTYPE**) :typeRep_T*
*dynamic_new(T <:**Ok** v :T) :dynamic_T*
*dynamic_be(d :dynamic_T T <:**Ok**) :T*
(löst Ausnahme aus, falls der dynamische Subtyptest fehlschlägt *)*

Hierbei stellt **ANYTYPE** den allgemeinsten *Kind*[3] dar, also insbesondere auch den *Kind* aller Typoperatoren. Die Übergabe von Typparametern wird bei den Anweisungen *typeRep_new* und *dynamic_be* durch die Syntax erzwungen. Bei der Anweisung *dynamic_new* wird immer der inferierte Typ zur Konstruktion des automorphen Wertes benutzt, es wird also kein Typparameter übergeben.

Diese Anweisungen sind keine Funktionen. Sie lassen sich vielmehr als Makros auffassen, die während der Typüberprüfungsphase expandiert werden, wobei neuer Code erzeugt wird, der an die Stelle der ursprünglichen Anweisung tritt. Dieser Vorgang wird in Abschnitt 5.1.5 dargestellt.

[2] Die verwendete Notation mit Unterstrich soll allerdings andeuten, daß Operationen auf Werten dieser Typen in entsprechend benannten Modulen zur Verfügung stehen.

[3] *Kinds* strukturieren den Typraum in ähnlicher Weise, wie Typen den Werteraum strukturieren [Cardelli 89].

Behandlung von Polymorphismus und abstrakten Typen

Die Einführung dynamischer Typen darf keine Möglichkeit eröffnen, das Typsystem zu unterlaufen. Dies ist insbesondere dann ein Problem, wenn es Polymorphismus und abstrakte Typen enthält [Matthews 87]. Beispielsweise darf es nicht erlaubt sein, mit Hilfe der polymorphen Funktion

$$\textbf{let}\ cast(A <: \textbf{Ok}\ \ d : dynamic\_T) : A = dynamic\_be(d : A)$$

das Typsystem zu unterlaufen, indem man z.B. aus einem Integer-Wert einen automorphen Wert konstruiert, ihn dieser Funktion übergibt und das Resultat z.B. als String weiterverwendet. In TL wird deswegen der Aufruf von *dynamic_be* statisch verboten, wenn der übergebene Typ eine polymorphe Typvariable ist.

Die Erzeugung von Repräsentationen abstrakter Typen kann nur zum Teil zur Übersetzungszeit erfolgen. Dies liegt daran, daß erst zur Laufzeit bekannt ist, auf welchem Pfad (d.h. von welchem Tupel- oder Modul-Objekt aus) ein abstrakter Typ referenziert wird. Diese Information muß jedoch unbedingt in dessen Repräsentation enthalten sein, damit nicht fälschlicherweise Typrepräsentationen abstrakter Typen zueinander kompatibel sind, deren Implementation unterschiedlich ist, wie folgendes Beispiel verdeutlicht:

> Es gelte $S \vdash d : stack.T$. Durch die Bindung
>
> $$\textbf{let}\ d = dynamic\_new(s)$$
>
> wird ein automorpher Wert aus diesem Kellerspeicher-Objekt erzeugt. In einem anderen Kontext wird es z.B. durch
>
> $$\textbf{let}\ d : dynamic\_T = dynamic.intern(fileHandle)$$
> $$\textbf{let}\ s = dynamic\_be(d : stack.T)$$
>
> als statisch typisierter Wert verfügbar gemacht. Der dynamische Typtest muß jedoch fehlschlagen, falls das Tupel (im allgemeinen ein gebundenes Modul) *stack* nicht dieselbe Identität hat wie das, was im Kontext zur Zeit der Erzeugung des automorphen Wertes gebunden war. Pfade von abstrakten Typvariablen sind also bei der Erzeugung von Typrepräsentationen dynamisch zu binden.

Die Inspektion von Laufzeit-Typrepräsentationen darf es dem Anwender nicht erlauben, die konkrete Struktur abstrakter Typen zu erfahren und dieses Wissen als Grundlage weiterer Berechnungen zu verwenden, denn dies stünde im Widerspruch zum Grundgedanken abstrakter Datentypen. Die Repräsentation eines abstrakten Typs enthält darum lediglich dessen Namen und Information über den Pfad, auf dem er referenziert wurde. Diese Pfadinformation sollte idealerweise systemübergreifend eindeutig sein, in der derzeitigen Implementation ist sie jedoch lediglich innerhalb eines Objektspeichers eindeutig.

Typregeln

Die statische TL-Semantik wird in [Matthes 93] durch Typregeln formal beschrieben. Diese Regeln werden in folgender Weise erweitert bzw. ergänzt:

[Type Builtin]

$$\frac{\vdash S\ sig \qquad A \in \{\mathbf{Ok}, \mathbf{Nok}, \mathbf{Bool}, \mathbf{Int}, \mathbf{String}, \mathbf{typeRep\_T}, \mathbf{dynamic\_T}\}}{S \vdash A\ type}$$

Diese Regel drückt aus, daß die Menge der Basistypen um die Typen *typeRep_T* und *dynamic_T* erweitert wird.

[Value TypeRep]

$$\frac{S \vdash A\ type}{S \vdash \mathbf{typeRep\_new}(A) : \mathbf{typeRep\_T}}$$

Die Anwendung der Anweisung **typeRep_new** auf einen wohlgeformten Typen resultiert in einem Wert vom Typ *typeRep_T*.

[Value Dynamic]

$$\frac{S \vdash x : A \qquad S \vdash A <: \mathbf{Ok}}{S \vdash \mathbf{dynamic\_new}(x) : \mathbf{dynamic\_T}}$$

Die Anwendung der Anweisung **dynamic_new** auf einen Wert x vom Typ $A <:$**Ok** resultiert in einem Wert vom Typ *dynamic_T*.

[Value Be]

$$\frac{S \vdash x : \mathbf{dynamic\_T} \qquad S \vdash A <: \mathbf{Ok}}{S \vdash \mathbf{dynamic\_be}(x, A) : A}$$

Die Anwendung der Anweisung **dynamic_be** auf einen automorphen Wert und einen wohlgeformten Typ $<:$**Ok** resultiert in einem Wert dieses Typs.

Zur Realisierung dieser Regeln ist, im Sinne des Orthogonalitätsprinzips von TL, eine Erweiterung der Syntax polymorpher Funktionen wie in folgendem Beispiel notwendig:

let *f(**Dyn** T <:**Ok** x :T) :... =*
*... **dynamic_new**(x) ...*

Durch das Schlüsselwort **Dyn** wird dem Compiler hierbei mitgeteilt, daß er Vorkehrungen treffen muß, um der Funktion *f* zur Laufzeit eine Typrepräsentation des übergebenen (oder inferierten) Typs *T* zur Verfügung zu stellen. Ohne diese Erweiterung muß *f* folgendermaßen formuliert werden:

let *f(x :dynamic_T) :... = ... x ...*

Die Erzeugung eines automorphen Wertes erfolgt *vor* dem Aufruf, da im Rumpf von *f* statisch nur die Zusicherung $T <: Ok$ bekannt ist. Es ist dann keine Vermischung von parametrischem und reflektivem Polymorphismus möglich.

5.1.4 Programmierschnittstellen für dynamische Typen und automorphe Werte

Zur Entwicklung datenstrukturgetriebener Algorithmen benötigt der Programmierer neben den reflektiven Basiskonstrukten eine Programmierschnittstelle, die ihm Funktionen sowohl zur Inspektion als auch zur Konstruktion von Typrepräsentationen und automorphen Werten zur Verfügung stellt. Diese sind in den Modulen *typeRep* und *dynamic* implementiert, welche internes Wissen um die Darstellung von Typrepräsentationen durch den Compiler besitzen, ohne jedoch Teil des Compilers zu sein, wie Abb. 5.3 verdeutlicht.

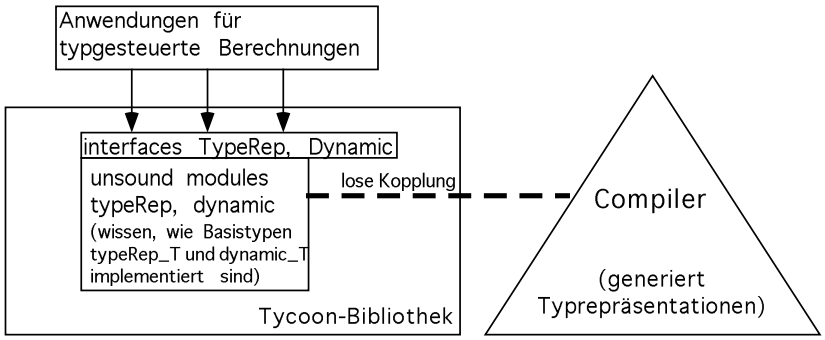

Abbildung 5.3: Kopplung des API für dynamische Typen an den Compiler

Schnittstelle 'TypeRep'

Diese Schnittstelle stellt Funktionen zur strukturierten Analyse und Konstruktion von Typrepräsentationen und Typbindungen zur Verfügung (Schnittstelle siehe Anhang B.1). Wichtigste Operation auf Typrepräsentationen ist die Funktion *inspect*, die eine detailliertere Beschreibung der ihr übergebenen Typrepräsentation liefert. Ferner gibt es für jeden strukturierten TL-Typ Konstruktorfunktionen, um aus Zwischenergebnissen von *inspect*, z.B. Listen von Signaturen, wiederum Typrepräsentationen aufbauen zu können. Die Anweisung *typeRep_new* reicht dazu nicht aus, da man mit ihr nur Typrepräsentationen von statisch bekannten Typen erzeugen kann.

Die Funktion *isSubType* ist die öffentlich zugängliche Version des Subtyptests des TL-Compilers. Hervorzuheben ist noch die Funktion *fmt*, die eine Zeichenkette aus einer Typrepräsentation erzeugt. Sie wird vor allem in Generatoren bei der laufzeit-reflektiven Programmierung verwendet, um Quelltext für berechnete Typen zu erzeugen.

Schnittstelle 'Dynamic'

Durch die Anweisung *dynamic_be* läßt sich die Wertkomponente eines automorphen Wertes wieder auf die statisch typisierte Ebene zurückholen. Oftmals gibt es jedoch Programmiersituationen, bei denen man statisch keinerlei Information über die Typkomponente eines automorphen Wertes und deshalb keinen Anhaltspunkt zur Parametrisierung der *dynamic_be*-Anweisung hat. In diesem Fall ist eine sukzessive Inspektion des automorphen Wertes nötig, um an dessen elementare Wertkomponenten zu gelangen, wie folgendes Beispiel zeigt:

Sei d in einem anderen Kontext gebunden worden durch

```
let d = dynamic_new(tuple let name = "Hans" let age = 43 end),
```

so kann auf die Komponente *name* folgendermaßen zugegriffen werden:

```
case dynamic.inspect(d)
when tupleCase with d then
  let nameDyn = list.head(d.bindings)!valueCase.value()
  dynamic.inspect(nameDyn)!stringCase.val
    (* ! = Variantenprojektion *)
else ...
end.
```

Automorphe Werte von Arrays erfordern eine besondere Betrachtung. Sinn der Inspektionsfunktion ist es ja, eine detailliertere Beschreibung des automorphen Wertes zu liefern[4]. Inspeziert man einen automorphen Wert eines Arrays, so müßte ein Array zurückgegeben werden, bei dem jedes einzelne Element mit seiner Typrepräsentation aggregiert ist. Dies würde insbesondere bei großen Arrays zu erheblicher redundanter Typinformation und damit Speicherverschwendung führen. Für diesen Fall stehen daher die Funktionen

```
getIndexed(:dynamic_T) :dynamic_T
```

und

```
setIndexed(a :dynamic_T  value :dynamic_T) :Ok
```

zum indizierten Zugriff auf die Komponenten eines Arrays innerhalb eines automorphen Wertes zur Verfügung.

Als wichtige Operationen auf automorphen Werten seien noch die Funktionen *extern* und *intern* hervorgehoben (siehe dazu auch [Cardelli 89]), die eine lineare Repräsentation eines automorphen Wertes z.B. auf eine Datei schreiben bzw. von ihr lesen. Diese Operationen dienen z.B. zum typisierten Datenaustausch zwischen unabhängigen Objektspeichern oder verteilten autonomen Anwendungen. Diese Funktionen werden implementiert durch Aufrufe von Funktionen des *Tycoon Store Protocol*, welche die Linearisierung von Objektgraphen realisieren.

5.1.5 Implementierung durch Reflektion zur Übersetzungszeit

Ein großer Vorteil persistenter Umgebungen ist es, daß Programme in derselben Umgebung (d.h. konkret im selben Objektspeicher) ausgeführt werden können, in der sie übersetzt wurden. Wenn dann auch noch der Compiler in derselben Sprache geschrieben ist wie seine Quellsprache, so ist es relativ einfach möglich, seine internen Datenstrukturen zur Laufzeit verfügbar zu machen. Diese Eigenschaft wird zur Implementierung der Anweisung *typeRep_new(:T)* ausgenutzt, indem sie zur Übersetzungszeit wie folgt ausgewertet wird.

1. T wird innerhalb des statischen Kontexts S auf Wohlgeformtheit geprüft.

[4] siehe Typ *Expansion* in Schnittstelle *Dynamic* (Anhang B.2)

2. Aus T wird eine kontextunabhängige Repräsentation $t_{S,T}$ generiert (siehe dazu Abschnitt 5.1.5).

3. Um die Pfade abstrakter Typen (siehe Abschnitt 5.1.3) zur Laufzeit in die Typrepräsentation integrieren zu können, wird ein Funktionsaufruf generiert, der die Pfade (d.h. die Wertkomponenten) aller benutzten abstrakten Typvariablen zusammen mit $t_{S,T}$ als Aktualparameter übergeben bekommt. Diese Funktion fügt die übergebenen Pfade dann zur Laufzeit an die entsprechenden Stellen in $t_{S,T}$ ein.

4. Der generierte Funktionsaufruf wird in den ursprünglichen Syntaxbaum implantiert.

Zur Konstruktion von automorphen Werten wird Code generiert, der ein Tupel bestehend aus dem übergebenen Wert und seiner Typrepräsentation aufbaut. Die Typrepräsentation wird dabei aus dem inferierten statischen Typ wie oben angegeben generiert. Dieses Tupel wird wiederum in den ursprünglichen Syntaxbaum eingesetzt. Der beispielsweise für den Aufruf *dynamic_new(3)* generierte Code sieht dann aus, als hätte der Programmierer folgendes geschrieben:

```
tuple
  let value = 3
  let type = typeRep_new(:Int)
end
```

Durch den Compiler wird allerdings sichergestellt, daß die Typrepräsentation auch wirklich zum Wert paßt, da dessen Typ inferiert und nicht, wie im Beispiel, explizit angegeben wird.

Die Umsetzung der Anweisung *dynamic_be* in Code, der einen Laufzeit-Subtyptest veranlaßt, verläuft ähnlich wie bei der Generierung von Typrepräsentationen, denn auch *Funktionen* des Compilers können als Objektliteral in den Syntaxbaum eingefügt werden. Es wird in diesem Fall eine Funktionsapplikation generiert, die als Wert eine Compiler-Funktion enthält, die die gewünschte Funktionalität (Subtyptest) realisiert. Probleme treten bei diesem Ansatz (Bindung zur Übersetzungszeit) jedoch durch die hybride Natur der jetzigen Modulverwaltung auf, die lineare Repräsentationen übersetzter Module sowohl im Objektspeicher hält (genauer gesagt in einem Modul-*Cache*) als auch im Dateisystem ablegt. Die lineare Repräsentation eines übersetzten Moduls, die Code enthält, der einen Subtyptest basierend auf Strukturäquivalenz ausführt, verbraucht jedoch unnötig viel Platz, denn zum Bindungszeitpunkt dieses Moduls existiert dieser Code ohnehin bereits im Objektspeicher. Für diesen Fall der Benutzung von *dynamic_be* innerhalb eines Moduls sollte die Bindung dieses Codes zweckmäßigerweise zu einem späteren Zeitpunkt erfolgen, am besten in der separaten Modulbindungsphase.

Aufbau von Typrepräsentationen

Die Typrepräsentationen, die in der Typüberprüfungskomponente des TL-Compilers eingesetzt werden, basieren auf der sogenannten *de Bruijn Notation* [de Bruijn 72; Abadi et al. 90] zur Formalisierung der Sichtbarkeitsregeln von Variablen. Dies bedingt die Abhängigkeit *aller* dieser Typrepräsentationen von *einem* gemeinsamen statischen Kontext, sofern sie Indices auf Signaturen in diesem Kontext enthalten, was in der Regel der Fall ist.

Da dynamische Typen vor allem dazu gebraucht werden, Werte zusammen mit ihrer Typinformation zwischen verschiedenen Kontexten zu transferieren, ist die Abhängigkeit der Typinformation von einem gemeinsamen Kontext, der über einen initialen Kontext hinausgeht, unerwünscht. Eine Lösung für dieses Problem wäre, jeweils eine Typrepräsentation zusammen mit ihrem Kontext zu aggregieren. Dieses Vorgehen hat jedoch zwei gravierende Nachteile:

1. Der Kontext enthält im allgemeinen Signaturen, die von der Typrepräsentation nicht (per de Bruijn Index) referenziert werden. Dieser u.U. sehr große Anteil an überflüssiger Information würde sich insbesondere beim Transfer von Typrepräsentationen zwischen autonomen Objektspeichern negativ auswirken. Ein Entfernen nicht referenzierter Signaturen würde eine aufwendige Anpassung sämtlicher Indices im Kontext nach sich ziehen.

2. Beim dynamischen Subtyptest (initiiert durch die Anweisung *dynamic_be*) müßte der Kontext, in der die Typkomponente des automorphen Wertes gilt, mit dem aktuellen Kontext vereinigt werden, da Subtypbeziehungen in der derzeitigen Implementation des TL-Typüberprüfers nur in einem gemeinsamen Kontext gelten können. Auch dies wäre eine teure Laufzeitoperation.

Aus diesen Gründen wird eine zweite Darstellung von Typrepräsentationen eingeführt, die Typvariablen nicht durch de Bruijn Indices, sondern durch direkte Referenzen auf ihr definierendes Auftreten repräsentiert. Diese Darstellung entspricht folgendem rekursiven Typ T:

```
Let Rec T <:Ok =
 Tuple
  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
           (* Pfad (als String), auf dem der ADT referenziert wurde. Wird
              zur Laufzeit durch die OID des betreffenden Moduls ersetzt. *)
         end
  case funCase with                 (* Funktionstyp *)
   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                  (* Typoperator *)
      signatures :list.T(Signature)
      range :T
    case arrayCase with
      elementType :T
    case appCase with                   (* Typoperatorapplikation *)
      oper :T
      arguments :list.T(Signature)      (* rekursiver Typ *)
    case recCase with
      bindingIndex :Int
      typeBindings :list.T(Signature)   (* Typ einer veränderlichen Bindung *)
    case varCase with
      type :T
  end

and Signature <:Ok = Tuple              (* Enthält auch Typbindungen *)
  ide :Tuple name :String pos :SourcePosition end
  var type :T
  case typeCase, typeEqualCase with     (* T <:A bzw. Let T = A *)
    typeID :time.T                      (* Zeitstempel *)
  case typeBoundCase with               (* Let [Rec] T <:B = A *)
    typeID :time.T                      (* Zeitstempel *)
    var bound :T    case valueCase
  case locationCase
end

and Case <:Ok = Tuple
  label :String
  signatures :list.T(Signature)         (* zusätzliche Felder dieser Variante *)
end
```

Das nachfolgende Beispiel verdeutlicht zusammen mit Abbildung 5.4 den Unterschied zwischen beiden Typdarstellungen.

> Angenommen, der initiale Kontext enthalte die Deklaration des Typs *Int* an Position 50 und die Deklaration des Typs *String* an Position 55. Es gelten folgende Typbindungen:
>
> ```
> Let Age = Int@50
> Let Address = Tuple street :String@56 city :String@57  zip :Int@ end
> Let Person = Tuple name :String@58  age :Age@3  address :Address@3 end
> Let Student = Tuple
>                   name :String@59
>                   age :Age@4
>                   address :Address@4
> ```

semester :Int@57
end

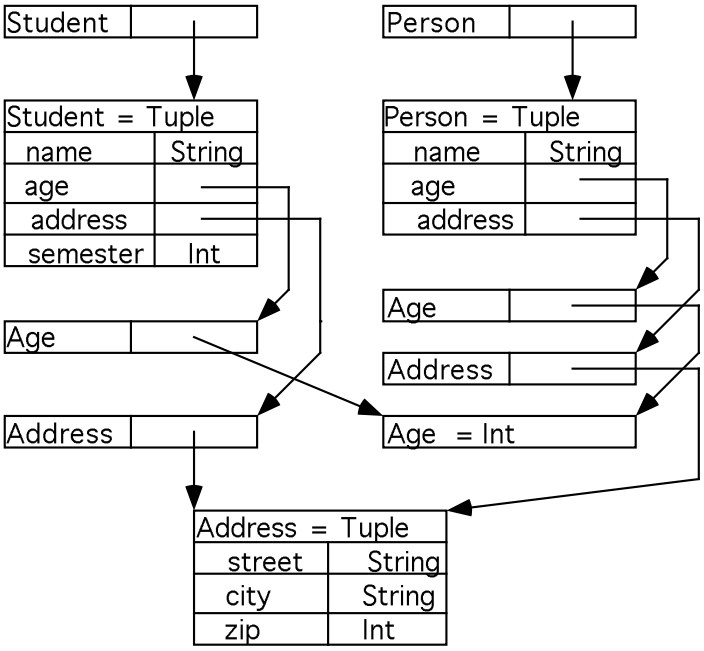

Abbildung 5.4: Typrepräsentationen mit Objektreferenzen

Die Darstellung mit direkten Referenzen entspricht einem gerichteten Graph, der jedoch durch die Möglichkeit rekursiver Typdefinitionen auch Zyklen enthalten kann. Zu beachten ist dabei, daß Typrepräsentationen gemeinsame Komponenten teilen (da sie Typvariablen enthalten können), aber trotzdem kontextunabhängig sind. Wird z.B. ein automorpher Wert in eine Datei geschrieben, so werden alle von seiner Typkomponente aus transitiv erreichbaren Signaturen erfaßt. Man erhält dadurch automatisch den gewünschten minimalen Kontext[6], da dieser durch die Darstellung als Objektgraph in jeder Typrepräsentation implizit vorhanden ist.

Generierte Typrepräsentationen für Typbindungen werden in einer globalen Tabelle im Objektspeicher verwaltet. Durch die Verwendung der Objektidentität der Typbezeichner von Signaturen als Schlüssel dieser Tabelle wird gewährleistet, daß Typrepräsentationen nur dann eine Typbindung teilen, wenn sie dieselbe Quelltextdefinition dieser Typbindung referenzieren.

5.2 Der aufrufbare Compiler

Mit der Anwendungsschnittstelle des aufrufbaren TL-Compilers werden zwei Ziele verfolgt:

1. Unterstützung typsicherer laufzeitreflektiver Programmierung.
2. Realisierung eines flexiblen Mechanismus zur *inkrementellen Übersetzung und Ausführung* von TL-Ausdrücken als Bibliotheksdienst.

[6] d.h. der Kontext enthält keine nichtreferenzierten Typbindungen

Inkrementelle Übersetzung wird dem Benutzer bisher durch den interaktiven *top Level* (textuelle Schnittstelle des Tycoon-Systems) angeboten. Diesem mangelt es aber insofern an Flexibilität, als daß er nur eine flache, monolithische Sicht auf Bindungen im Objektspeicher bietet. Da er nur monoton wachsen kann, geht schnell die Übersicht über vorhandene Bindungen verloren. Sein Einsatzbereich beschränkt sich im wesentlichen auf die Auswertung kurzer Ausdrücke (Anfragen) und die Parametrisierung des Compilers durch Auswertung spezieller Kommandos. In großen persistenten Systemen ist jedoch eine kontrollierte Evolution von Daten und Funktionen im Objektspeicher nur dann möglich, wenn vorhandene Bindungen geeignet strukturiert werden können. Nachfolgend werden daher *Environments* eingeführt, welche den bisherigen *top Level* ersetzen. Sie ermöglichen es dem Anwendungsprogrammierer, den Objektspeicher als eine Art *repository* für programmiersprachliche Objekte zu verwenden. Die Strukturierung des Namensraumes aller Objekte im Objektspeicher durch Environments [Dearle et al. 92], die Möglichkeit der inkrementellen Übersetzung sowie nicht zuletzt die Persistenz (durch Erreichbarkeit definiert) bewirken durch ihr Zusammenspiel einen Synergie-Effekt, der sich besonders bei der GUI-gestützten Programmierung, beispielsweise für Ad-hoc-Anfragen, positiv auswirkt (siehe Abschnitt 6.2).

5.2.1 Environments

In [Morrison 94; Dearle 89] sind Environments definiert als die unendliche Vereinigung aller benannten Kreuzprodukte. Konkret enthalten Environments eine nicht notwendigerweise geordneten Sequenz von Werten des nachfolgend definierten Typs *Binding*. Sie werden dem aufrufbaren Compiler als Parameter übergeben und erlauben es damit, Ausdrücke gegen bereits existierende Werte und Typdefinitionen auszuwerten. Anders als z.B. in Napier88 gibt es keine Sprachkonstrukte zur Manipulation von Environments. Stattdessen wird ein *add-on* Ansatz bevorzugt und ein Bibliotheksmodul, basierend auf dynamischen Typen und automorphen Werten, implementiert. Diese Vorgehensweise harmonisiert mit der Auffassung, Massendatentypen besser nicht in die Sprache zu integrieren, sondern sie in Bibliotheken als generische Dienste anzubieten und damit den Sprachkern so einfach wie möglich zu halten [Matthes, Schmidt 91]. Im Tycoon-System ist es jedoch auch möglich, die Sprache durch *syntaktische Erweiterung* [Cardelli et al. 94] mittels erweiterbarer Grammatiken mit einer benutzerfreundlicheren Syntax für die Manipulation von Environments (z.B. in der Art von Napier88) auszustatten.

Das Bibliotheksmodul *environment* (siehe Anhang C.1) enthält einen abstrakten Datentyp (*environment.T*), auf dem folgende Grundfunktionen definiert sind (vergleiche [Dearle 89]):

1. Erzeugen eines initialen Environment,
2. Erweiterung um Wert- oder Typbindungen,
3. Aufsuchen von Wert- oder Typbindungen anhand ihres Namens,
4. Entfernen von Bindungen.

Bindungen

Nach [Morrison et al. 90] sind Bindungen charakterisiert durch ein 4-Tupel bestehend aus Name, Wert, Typ und einen boolschen Wert, der angibt, ob die Bindung veränderlich oder

konstant ist. Mit Hilfe dynamischer Typen und automorpher Werte lassen sich Bindungen durch folgenden Optionstyp darstellen:

```
Let Binding = Tuple
  name :String
  case value with
    value :dynamic_T
  case location with
    value() :dynamic_T
    setValue(:dynamic_T) :Ok
  case type with
    type :typeRep_T
end
```

Anders als bei obigem 4-Tupel werden hier auch Typbindungen berücksichtigt.

Veränderliche Bindungen müssen durch den Compiler erzeugt werden, da nur er die Zelle kennt, in der der betreffende Wert gespeichert wird. Er kann dem Benutzer jedoch Zugriffsfunktionen auf diese Zelle zur Verfügung stellen. Eine veränderliche Bindung kann somit auf zwei Arten modifiziert werden: Einerseits von der Benutzerebene direkt durch Aufruf der Funktion *setValue()*, andererseits durch die reflektive Auswertung eines Ausdrucks (Zuweisung) gegen ein Environment, das diese Bindung enthält.

Verwendungsmöglichkeiten

Da Environments zu jeder Bindung sowohl Typinformation als auch den Wert selbst enthalten, ermöglichen sie sowohl die Übersetzung als auch die Ausführung von Ausdrücken. Sie subsumieren damit die Funktionalität des bisherigen TL *top level*.

Durch Schachtelung, ausgehend von einer persistenten Wurzel vom Typ *environment.T*, bieten sie eine flexible Möglichkeit zur Strukturierung und Typisierung des Objektspeichers (siehe Abbildung 5.5).

Weitere Eigenschaften sind

- Erweiterbarkeit: Environments können sowohl wachsen (durch Auswertung von Ausdrücken) als auch schrumpfen (durch Löschen von Bindungen).
- Migrationsmöglichkeit von Bindungen: Durch die Darstellung der (dynamischen) Typinformation durch direkte Referenzen sind Bindungen zwischen Environments leicht kopierbar, ohne die referentielle Integrität von Typen zu verletzen. Dadurch ist eine Restrukturierung des Objektspeichers durch den Anwender möglich[7].
- Unterstützung von Reflektion: Der Compiler kann sich selbst, d.h. die Bindungen aus denen er besteht, typisiert im Objektspeicher in einem Environment darstellen und damit seinen Zustand, z.B. durch den Anwender oder durch Werkzeuge einer Programmierwerkbank zugreifbar machen.

[7] Eine Restrukturierung des Objektspeichers konnte bisher nur von der Freispeicherverwaltung des Laufzeitsystems bei der *garbage collection* vorgenommen werden.

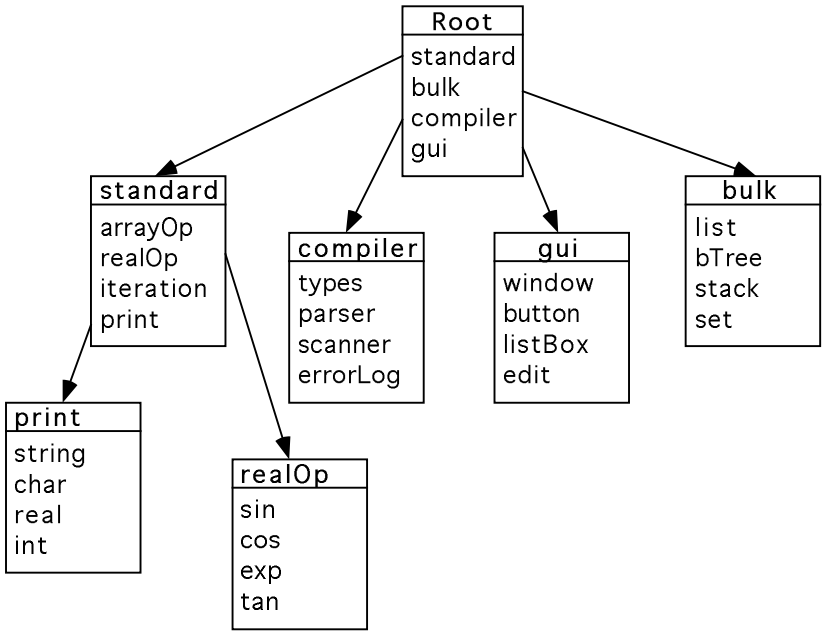

Abbildung 5.5: Objektspeicher als Baum von Environments

Der Graph, der durch die Schachtelung von Environments entsteht, entspricht dem hierarchischen Aufbau traditioneller Dateisysteme [Dearle 89]. Anders als in diesen lassen sich jedoch Daten beliebiger Struktur typisiert speichern, während Dateisysteme nur wenige unterschiedliche, flach strukturierter Dateiformate (z.B. ASCII-Format, Binärformat) zulassen.

Das Environment der obersten Ebene (Wurzel der Persistenz) läßt sich als Einstiegspunkt eines *repository* für typisierte Funktionen und Daten auffassen. Das Auffinden von Wert- und Typbindungen in diesem *repository* kann durch ein GUI-gesteuertes Navigationswerkzeug oder durch programmiersprachlich spezifizierte Anfragen unterstützt werden.

5.2.2 Architektur der Compiler-Schnittstelle

Die bisherige Architektur des TL-Compilers bietet dem Benutzer lediglich einen schmalen Einstiegspunkt, nämlich eine kommandoorientierte textuelle Schnittstelle nach dem Muster *read – eval – print* zur Übersetzung und Ausführung von TL-Ausdrücken (siehe Kapitel 2). Ziel einer revidierten Architektur ist die „Verbreiterung“ dieser Schnittstelle, einerseits zur Benutzung aus laufzeitreflektiven Anwendungsprogrammen heraus, andererseits als Basis für Werkzeuge einer integrierten Programmierwerkbank, wie in Abbildung 5.6 veranschaulicht. Sie ermöglicht es, TL als eingebettete Sprache in Anwendungsprogrammen zu verwenden, wobei dann auch die Programmierwerkbank als Anwendungs- und nicht als Systemprogramm zählt.

Bisher existierten gebundene Anwendungsprogramme getrennt vom Compiler im Objektspeicher. Obwohl die Laufzeitumgebung (d.h. der Objektspeicher) den Compiler enthält, sind seine Komponenten von Anwendungsprogrammen aus nicht zugänglich, da keine Typinformation zu diesen Komponenten in einer gemeinsamen Umgebung existiert.

Dynamische Typinformation macht es jedoch möglich, Laufzeitkomponenten des Compilers

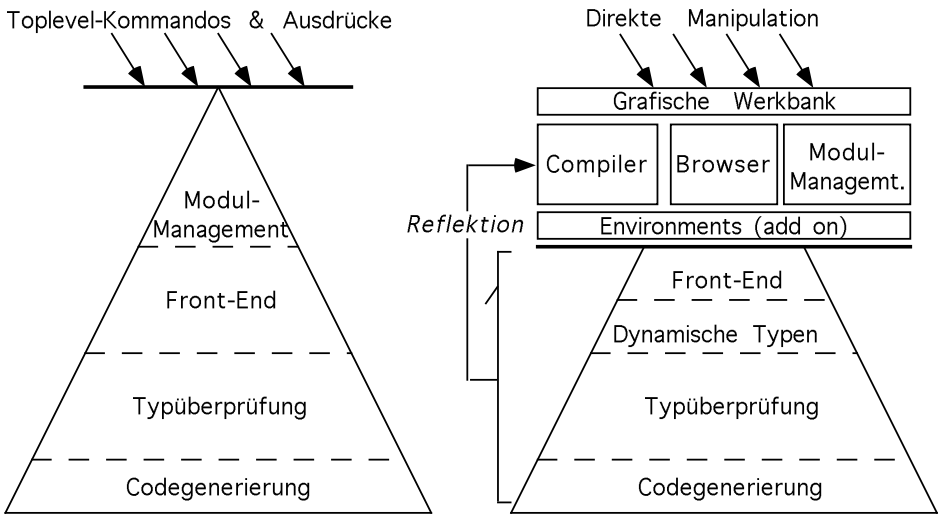

Abbildung 5.6: Schmaler Einstiegspunkt der bisherigen Architektur gegenüber einer reflektiven Architektur

in einem Environment zu binden und dieses z.B. innerhalb einer Programmierwerkbank dem Benutzer als Auswertungsumgebung zur Verfügung zu stellen. Weiterhin können ihm, abgesehen von Komponenten des Compilers selbst, auf diese Weise auch alle TL-Standardmodule zur Verfügung gestellt werden. Dadurch erhält der Benutzer eine Bibliothek vordefinierter Bindungen im Objektspeicher schon zum Systemgenerierungszeitpunkt.

Da Komponenten des Compilers als vordefinierte Bindungen existieren, enfällt damit die Notwendigkeit einer Kommandosprache zur Steuerung des Übersetzungsvorganges (setzen von Schaltern, Zustandsvariablen etc., Details zur Kommandosprache siehe [Mathiske et al. 93]), da hierfür TL-Anweisungen eingesetzt werden können, z.B.

compiler.traceSubtypeTest(true)

statt

do set traceSubtypeTest true.

Eine auf dieser Compilerschnittstelle aufsetzende textuelle Benutzerschnittstelle ähnelt in ihrem Verhalten sehr den aus verschiedenen Betriebssystemen (z.B. UNIX-Derivate) bekannten *Shells*, mit dem Unterschied, daß statt einer Kommandosprache mit mehr oder weniger hoch entwickelten Kontrollstrukturen eine algorithmisch vollständige Programmiersprache zur Verfügung steht [Dearle 88]. Analog zu *Shell*-Kommandointerpretern, die Dateien in hierarchischen Verzeichnisstrukturen manipulieren, bearbeitet das Tycoon-Laufzeitsystem persistente Wert- und Typbindungen in hierarchischen Environments.

Zur Durchführung der Übersetzung baut der Compiler intern verschiedene Datenstrukturen auf, die Zwischenergebnisse des Übersetzungsvorganges repräsentieren, wie z.B. Syntaxbäume und Zwischencode. Durch die reflektive Zugangsmöglichkeit zu Komponenten des Compilers können dem Benutzer diese Strukturen für eigene Auswertungen nach jedem Übersetzungsvorgang zur Verfügung gestellt werden.

Schnittstelle 'Compiler'

Aufbauend auf existierenden Komponenten (Parser, Typüberprüfer, Codegenerierung) wird in dieser Arbeit eine von TL-Programmen aus aufrufbare Schnittstelle zum TL-Compiler entwickelt. Wesentliches Merkmal dieser Schnittstelle ist die Parametrisierung des Übersetzungsvorgangs durch ein Environment (siehe Abschnitt 5.2.1). Dadurch können Programme gegen Werte der aktuellen Laufzeitumgebung übersetzt und ausgeführt werden, z.B. durch eine Funktion folgender Signatur:

*compileString(code :String env :environment.T) :**Fun**() :Binding*

Diese Funktion übersetzt eine Zeichenkette gegen das übergebene Environment und gibt eine Funktion zurück, die bei ihrer Ausführung den übersetzten Ausdruck auswertet. Die letzte durch Auswertung des Ausdrucks entstandene Bindung wird zurückgegeben und als Seiteneffekt das übergebene Environment um die neu etablierten Bindungen erweitert. Bei Fehlern während der Übersetzung wird ein Ausnahmepaket mit entsprechenden Fehlermeldungen erzeugt.

Mit Hilfe von Funktionen höherer Ordnung kann der Übersetzungs- und Auswertungsvorgang in seine einzelnen Phasen zerlegt und am jeweiligen Phasenende quasi angehalten und wiederaufgenommen werden.

```
parse(code :String)                 (* Parsing *)
  :Fun(env :environment.T)          (* Typüberprüfung *)
    :Fun()                          (* Codegenerierung *)
      :Fun() :Binding               (* Ausführung, Environment-Manipulation *)
```

Die Funktionen *compileString* und *eval* können dann durch die Bindungen

***let** compileString(env :environment.T code :String) :**Fun**() :Binding =*
* parse(code)(env)()*

***let** eval(code :String env :environment.T) :Binding =*
* compileString(env code)()*

definiert werden. Das der Funktion *eval* übergebene Environment wird typischerweise zuvor mit Werten und Typen aus dem äußeren Kontext gefüllt, wobei diese zunächst in automorphe Werte bzw. Typrepräsentationen konvertiert werden müssen. Umgekehrt können automorphe Werte und Typrepräsentationen aus dem durch die Auswertung von Ausdrücken erweiterten Environment herausgeholt und im äußeren Kontext benutzt werden. Dies folgt direkt aus der oben beschriebenen Kontextunabhängigkeit von Laufzeit-Typrepräsentationen.

Falls der Benutzer die Typen der durch den Compiler erzeugten Bindungen an automorphe Werte statisch kennt, so kann er durch Anwendung der Anweisung *dynamic_be* (dynamischer Subtyptest) die entsprechenden automorphen Werte wieder als statisch typisierte Werte in den äußeren Kontext einbinden. Hat er dieses Wissen nicht, so kann er immerhin noch durch Inspektion der automorphen Werte typgesteuerte Berechnungen im äußeren Kontext durchführen.

Im folgenden Beispiel soll eine Operation auf zwei automorphen Werten durchgeführt werden, deren Typ statisch nicht bekannt ist. Unter der Annahme, daß beide Werte eine Komponente mit der Signatur *salary :Int* enthalten, wird Code generiert, der die beiden Felder addiert und das Ergebnis zurückliefert. Falls diese Annahme nicht zutrifft, wird eine Ausnahme ausgelöst mit den Compiler-Fehlermeldungen als Ausnahmepaket.

```
let sumOfSalaries(employee1, employee2 :dynamic_T) :Int =
  begin
    let env = environment.initial()
    environment.addValueBinding(env "employee1" employee1)
    environment.addValueBinding(env "employee2" employee2)
    let code = "employee1.salary + employee2.salary"
    let resultBinding :Binding = compiler.eval(env code)
    let result :dynamic_T = resultBinding!valueCase.value
    dynamic_be(result :Int)
  end
```

Dieselbe Funktionalität könnte auch mit Hilfe dynamischer Typinspektion implementiert werden. In Abschnitt 5.3.3 werden die beiden typischen Implementierungsalternativen für typgesteuerte Algorithmen, Reflektion und Interpretation kurz diskutiert.

Die Bindung von freien Bezeichnern im Quelltext an aktuelle Werte erfolgt unmittelbar vor Ausführung der von *compileString* zurückgegebenen Funktion. Im zu übersetzenden Code referenzierte (Wert)-Bindungen im übergebenen Environment können also bei Bedarf nach der Übersetzung noch typsicher modifiziert werden[8], da der übersetzte Code nur Positionsinformation über Werte im Environment enthält und nicht die Werte selbst. Wird jedoch vor der Ausführung keine Änderung mehr am Environment vorgenommen, so ist der resultierende Effekt derselbe, als hätte die Bindung direkt zur Übersetzungszeit stattgefunden.

Integration der reflektiven Compilerschnittstelle

Die Hauptproblematik der Integration reflektiver Fähigkeiten in Tycoon liegt in den unterschiedlichen Repräsentationsformen für Typen zur Laufzeit und zur Übersetzungszeit (siehe dazu Abschnitt 5.1.5). Diese Tatsache macht häufige Konvertierungen von Typen in die jeweils andere Darstellung erforderlich: Der Typüberprüfer verlangt Typen in der Darstellung mit de-Bruijn-Indices, Laufzeitmanipulationen mit Typrepräsentationen erfordern die Darstellung mit direkten Referenzen. Diese Konvertierungsoperationen verhindern eine zufriedenstellende Performanz.

Ziel weiterer Aktivitäten im Tycoon-Projekt ist die Vereinheitlichung der Darstellung von Typrepräsentationen sowohl zur Übersetzungszeit als auch zur Laufzeit, um auch den Anforderungen dynamischer Typisierung gerecht zu werden. Dies macht eine Re-Implementierung des Typüberprüfers erforderlich, welche im Rahmen einer parallel zu dieser Arbeit laufenden Diplomarbeit stattfindet.

[8] Dies wird ermöglicht durch die Funktion *environment.changeValue(env :environment.T name :String value :dynamic_T)*.

5.3 Anwendungen von Laufzeit-Reflektion

Laufzeit-Reflektion ist immer dann sinnvoll anzuwenden, wenn zur Generierung von Quellcode auf Information zugegriffen wird, die erst zur Laufzeit vorhanden ist. Diese Information kann entweder vom Benutzer eingegeben werden oder aus dynamischer Typinformation bestehen. Typsichere Laufzeit-Reflektion ist eng gekoppelt mit dynamischer Typinformation: Einerseits ist sie selbst eine Anwendung, die auf dynamischer Typisierung beruht, andererseits verhilft sie einigen generischen, typgesteuerten Anwendungen zu einer potentiell effizienteren Implementierung, indem der Anteil interpretativer Berechnungen durch spezialisierte generierte Codefragmente ersetzt wird. Dies verdeutlichen die nachfolgenden Beispiele.

5.3.1 Dynamische Codegenerierung

Der in Kapitel 3 vorgestellte Gateway-Generator ist als typgesteuerte Anwendung einzustufen, denn GDL-Schnittstellenbeschreibungen können als Typen aufgefaßt werden, da sie auf TL-Schnittstellen (d.h. TL-Tupeltypen) abbildbar sind. Das Prinzip des Generators basiert somit auf der Generierung von Gateway-Implementationen durch Interpretation von Typrepräsentationen.

Die Generierungsphase ist dabei völlig getrennt von der Benutzungsphase, da der generierte Quelltext zunächst im Dateisystem abgelegt und anschließend übersetzt und in den Objektspeicher importiert (gebunden) wird. Bei Änderungen an Schnittstellenbeschreibungen kann eine Neugenerierung und Einbindung des zugehörigen *Stub*-Codes somit nicht im laufenden Betrieb einer Anwendung erfolgen. Insbesondere kann die Bibliotheksbeschreibung (*library*) nur statisch geändert werden.

In den meisten Fällen ist eine dynamische Änderung der Gateway-Implementation nicht erforderlich, da Änderungen an Schnittstellenbeschreibungen eher selten sind, und Anwendungen sich nicht notwendigerweise *dynamisch* an diese Änderungen anpassen müssen. Bei benutzerseitig konfigurierbaren Anwendungen ist jedoch eine dynamische Anpassung des laufenden Programmes an Benutzereingaben nötig.

Als Beispiel hierfür sei die in [Cooper, Kirby 94] diskutierte dynamische Anpassung der grafischen Benutzerschnittstelle einer Anwendung angeführt. Hierbei kann der Benutzer zwischen mehreren UI-Komponenten auswählen, wobei u.U. eine hohe Vernetzung zwischen diesen Komponenten besteht. Ohne Reflektion ist der Zugriff der funktionalen Komponenten einer Anwendung auf Komponenten der Benutzerschnittstelle nur *indirekt* möglich: Zunächst muß eine abstrakte UI-Komponente aufgerufen werden, die anhand der zuvor eingegebenen Benutzerpräferenzen entscheidet, welches konkrete UI-Element aufzurufen ist. Mit Reflektion läßt sich in die laufende Anwendung Code integrieren, der den Aufruf der konkreten UI-Komponente *direkt* ausführt, wodurch ein Effizienzgewinn erzielt wird. Ist die Menge der UI-Komponenten durch den Benutzer erweiterbar (z.B. durch Konstruktion von Dialogen aus primitiveren UI-Komponenten), ist eine Lösung ohne Reflektion schon nicht mehr möglich, da zur Konstruktionszeit der Anwendung nicht bekannt ist, welche UI-Komponenten es zur Laufzeit geben wird.

In Abschnitt 5.2.2 wird beschrieben, wie interne Zustände der Programmierumgebung durch den Benutzer mit Hilfe von TL-Ausdrücken manipuliert werden können. Allgemein kann jede TL-Anwendung die reflektive Compiler-Schnittstelle dem Benutzer zur Verfügung stellen, indem es ein Environment mit Werten aus dem eigenen Kontext füllt und vom Benutzer

eingegebene Zeichenketten als Ausdrücke gegen dieses Environment auswertet. Dadurch, daß der Anwendungsprogrammierer selbst bestimmt, welche Werte der Anwendung er dem Benutzer sichtbar und damit manipulierbar macht, ist eine kontrollierte Zustandsveränderung gewährleistet. Diese Möglichkeit ist natürlich nicht geeignet für Endbenutzer-Anwendungen, da Endbenutzer sich normalerweise nicht mit den Details der Implementierungssprache auseinandersetzen wollen und können.

Natürlicher Verbund beliebiger Relationen

Ein weiteres Beispiel für dynamische Codegenerierung zur Erreichung höherer Generik ist der natürliche Verbund beliebig typisierter Relationen [Kirby 92b]. Hierbei wird aus den Typrepräsentationen der Eingaberelationen eine spezialisierte Repräsentation einer Funktion generiert, die den Verbund durchführt. Dabei ist auch die Typrepräsentation der Resultatrelation zu ermitteln. Im Anhang D findet sich die vollständige Definition des Generators in TL. Dessen Benutzung im Zusammenhang mit dem aufrufbaren Compiler vollzieht sich etwa folgendermaßen[9]:

```
import environment joinGenerator set compiler

Let R1 = Record a,b,c :String end
Let R2 = Record b,c,d :String end
Let Result = Record a,b,c,d :String end

let code :String = generateJoin(typeRep_new(:R1) typeRep_new(:R2))
let env :environment.T = ... (* obtain environment from generator *)
let joinFunction :dynamic_T = compiler.eval(s code)!valueCase.value
let joinFunction =
  dynamic_be(joinFunction :Fun(:set.T(R1) :set.T(R2)) :set.T(Result))
... (* define rel1 :set.T(R1) and rel2 :set.T(R2) *) ...
let resultRel = join(rel1 rel2)
```

Eine nichtreflektive Lösung erfordert mehr Berechnung zur Laufzeit, da die Eingaberelationen mittels Ad-hoc-Typinformation[10] zunächst auf Kompatibilität getestet werden müssen, und der Algorithmus zur Produktion der Resultattupel, der u.a. einen Test auf Gleichheit zweier Attributbezeichner enthält, diese Typinformation interpretieren muß.

Problematisch an der reflektiven Lösung ist die Tatsache, daß der Benutzer der generierten Funktion deren Signatur statisch kennen muß, also insbesondere auch den Resultattyp, obwohl der Generator ihm die Aufgabe der Berechnung des Resultattyps eigentlich abgenommen hat. Die Angabe der vollständigen Signatur der generierten Funktion kann umgangen werden, indem man davon ausgeht, daß grundsätzlich eine Funktion mit dem Rückgabetyp *dynamic_T* generiert wird. Nach Durchführung der Verbundoperation steht die Resultatrelation dann als automorpher Wert zur Verfügung. Dieser kann zwar durch ein generisches Datenvisualisierungswerkzeug dargestellt werden, doch ohne weiteres statisches Wissen um den Aufbau seiner Typkomponente lassen sich keine Mengenoperationen auf die Wertkomponente anwenden, was seinen Anwendungsbereich unakzeptabel drastisch einschränkt.

[9] Relationen seien hier durch Mengen dargestellt.

[10] z.B. eine Listendarstellung der Attributnamen der Elementtupel

5.3.2 Implementierung von Übersetzungszeit-Reflektion

Nutzt man das Modul *compiler* in einer zusätzlichen *bootstrap*–Iteration zur Implementierung des Compilers selbst, ist man in der Lage, TL-Funktionen schon zur Übersetzungszeit auszuführen. Dadurch wird die Übersetzung selbst zu einem laufzeit-reflektiven Vorgang.

Eine einfache Anwendung für diese Methodik ist z.B. das Auswerten von Ausdrücken zur Übersetzungszeit, sofern diese nicht auf Laufzeitwerte zugreifen. Ein Beispiel (in einer testweise implementierten Syntax):

let** zehnMega = **reflect** 10 * 1024 * 1000 **end

Hierbei wird der in **reflect ... end** geklammerte Ausdruck zur Übersetzungszeit ausgewertet und das Ergebnis als Objektliteral in den Syntaxbaum implantiert. Die Umgebung, in der dieser Ausdruck ausgewertet wird, kann wiederum mit Werten des Compilers vordefiniert werden.

Entscheidend für einen sinnvollen Einsatz von Reflekion zur Übersetzungszeit ist jedoch die Möglichkeit, auf die aktuelle Übersetzungszeitumgebung zugreifen zu können, also insbesondere auf die Typinformation des aktuellen Kontexts. Dies ist in der reflektiven Programmiersprache TRPL [Sheard 90] verwirklicht. TRPL erlaubt die Definition von Macros, die durch Argumente aus der abstrakten Syntax der Sprache parametrisiert werden. Im Rumpf des Macros kann mit Hilfe vordefinierter Abstraktionen auf Typ- und Coderepräsentationen der aktuellen Übersetzungsumgebung zugegriffen werden. Die durch den Macroaufruf generierte abstrakte Syntax wird in den aktuellen Syntaxbaum eingegliedert und in der Übersetzung fortgefahren.

Reflektion zur Laufzeit ist mächtiger als Reflektion zur Übersetzungszeit, da Generatoren auf Laufzeitwerte (z.B. Benutzereingaben, zur Laufzeit modifizierte Werte im Objektspeicher) zugreifen können. Weiterhin ist in persistenten Systemen die Abgrenzung zwischen Übersetzungszeit und Laufzeit weniger scharf als in nichtpersistenten Systemen, da durch die Möglichkeit der inkrementellen Übersetzung und Ausführung praktisch die gesammte Lebensdauer des Objektspeichers als „Laufzeit“ betrachtet werden kann, in der persistente Seiteneffekte auftreten. Bei reflektiven Anwendungen gehören zu diesen Seiteneffekten insbesondere die persistente Speicherung der Generatoren sowie des generierten Codes. Durch das Angebot beider Formen von Reflektion wird die Ausdrucksmächtigkeit des Systems somit nicht erhöht.

5.3.3 Reflektion zur Laufzeit versus Typinspektion

Zur Implementierung generischer typgesteuerter Algorithmen ist oft sowohl eine reflektive als auch eine interpretative Lösung möglich. Die reflektive Lösung besteht in der Generierung und Ausführung typspezifisch spezialisierter Codefragmente, die in einer durch Typen indizierten persistenten Tabelle gespeichert werden. Die interpretative Lösung beruht auf der Inspektion dynamischer Typinformation (s. Abschnitt 5.1.4).

Der reflektive Aufruf eines Compilers zur Laufzeit ist eine vergleichsweise teure Operation, deren Nutzung sich nur dann rechtfertigen läßt, wenn ein Effizienzgewinn gegenüber einer interpretativen Lösung zu erwarten ist. Dies ist nur dann der Fall, wenn eine generierte Funktion so häufig aufgerufen wird, daß sich die relativ hohen Übersetzungkosten gegenüber den Interpretationskosten amortisieren (Abbildung 5.7).

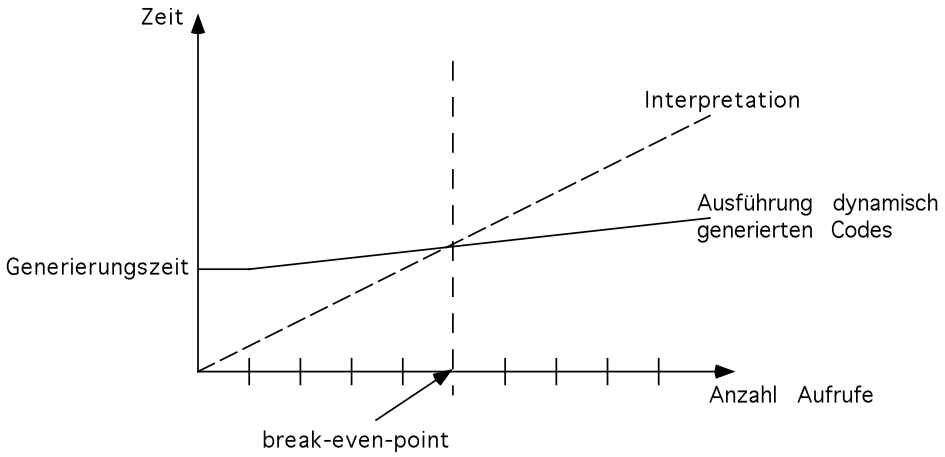

Abbildung 5.7: Break-Even-Analyse als Entscheidungshilfe für reflektive oder interpretative Implementierung typgesteuerter Algorithmen

Persistente Systeme sind in dieser Beziehung im Vorteil, da sich die Übersetzungkosten über mehrere Programmausführungen hinweg amortisieren können. Es ist dadurch auch ein weiter vom Ursprung entfernter *break-even-point* tolerierbar. Der Effizienzgewinn hängt dabei maßgeblich von der Effizienz der Verwaltung der generierten Funktionen ab. Wichtige Parameter sind dabei die Größe der Tabelle, die Trefferrate und die Kosten für das Auffinden einer Funktion in der persistenten Tabelle [Kirby 92b]. In relationalen Datenbanksystemen, z.B. System R, hat sich die Übersetzung mengenorientierter Anfragen als durchweg überlegen gegenüber ihrer Interpretation herausgestellt [Härder 87]. Dies gilt auch für Ad-hoc-Anfragen, sogar dann, wenn der übersetzte Code nicht für mehrmalige Benutzung aufgehoben wird.

Die Generierung von Funktionen anhand dynamischer Typinformation und deren Einbindung in den Programmablauf kann auch als eine eingeschränkte Form des *Lernens* aufgefaßt werden, wobei mit jeder generierten Funktion die Wissensbasis des reflektiven Programmes erweitert wird. Die maximale Größe dieser Wissensbasis ist dabei nur durch die maximale Größe des Objektspeichers beschränkt. Wird diese überschritten, so können z.B. nach der LRU-Strategie (*least recently used*) Funktionen gegen neu generierte ausgetauscht werden. Diese Form des *Vergessens* hat lediglich Auswirkungen auf die Performanz zukünftiger Aufrufe, wenn diese eine Neugenerierung von Funktionen veranlassen.

5.4 Aufrufbare Compiler für andere Programmiersprachen

In diesem Abschnitt folgt eine Zusammenstellung weiterer Ansätze, Schnittstellen zum Compiler in der eigenen Sprache als Dienst im Gesamtsystem zu begreifen. Auffällig ist, daß die genannten Beispiele alle aus dem Bereich der funktionalen Sprachen entstammen. Ein besonderer Zusammenhang zwischen funktionaler und reflektiver Programmierung läßt sich jedoch nicht belegen.

5.4.1 Lisp

In Lisp und seinen verschiedenen Dialekten wird der Interpreter/Übersetzer durch die Basisprimitive *eval* dem Benutzer zur Verfügung gestellt. Dadurch können beliebige S-Ausdrücke zur Laufzeit konstruiert und ausgewertet werden. Diese S-Ausdrücke können dabei Bezeichner aus der implizit bekannten aktuellen Auswertungsumgebung referenzieren. Die in dieser Arbeit vorgestellte Schnittstelle zum TL-Compiler ist dagegen explizit mit einer Auswertungsumgebung zu parametrisieren.

Die *eval*-Primitive ist ein sehr flexibles Mittel zur laufzeitreflektiven Programmierung, jedoch ist das Schreiben von Generatoren ohne Typüberprüfung als noch schwieriger und fehlerträchtiger anzusehen als in einem typsicheren Kontext. Im Gegensatz dazu ermöglichen die nachfolgend behandelten Sprachen typsichere linguistische Reflektion.

5.4.2 PS-algol und Napier88

PS-algol und dessen Nachfolger Napier88 bieten den Compiler als Funktion im Objektspeicher an. Dadurch wird er für reflektive Programme zur Laufzeit zugreifbar.

In PS-algol [Glasgow 88] wird diese Funktion, abgesehen vom Quelltext, mit einer Struktur parametrisiert, die zunächst als Platzhalter fungiert und bei erfolgreicher Übersetzung das Ergebnis aufnimmt. Als Ergebnisse sind nur Funktionen zugelassen, deren Signaturen statisch bekannt sein müssen. Der Compiler überprüft zur Laufzeit, ob der aus dem Übersetzungsvorgang inferierte Typ mit dem Typ des vom Benutzer übergebenen Platzhalters übereinstimmt. Bei erfolgreicher Übersetzung wird ein Zeiger auf den Platzhalter zurückgegeben, ansonsten ein Nullwert. Zur reflektiven Programmierung eignet sich diese Schnittstelle nur bedingt, da sie nicht mit Werten aus dem aufrufenden Kontext parametrisiert wird. Allerdings kann das generierte Programm zur Laufzeit auf Werte im Objektspeicher über vordefinierte Funktionen zugreifen.

Der aufrufbare Napier88-Compiler stellt eine flexiblere Schnittstelle zur Verfügung. Beliebige Napier88-Ausdrücke können gegen sogenannte *declaration sets* [Kirby et al. 94a] übersetzt werden. Diese stellen einen zusätzlichen äußeren Sichtbarkeitsbereich dar und entsprechen in etwa den oben vorgestellten Environments. Ausgewertete Ausdrücke liefern einen Wert des in Napier88 vordefinierten Typs *any*[11]. Durch Variantenprojektion, die einen dynamischen Typtest impliziert, läßt sich der in diesen Wert injizierte Wert statisch typisiert in den äußeren Kontext integrieren. Diese Funktionalität entspricht in etwa der der oben beschriebenen Funktion *dynamic_be*. Falls keine Annahmen über den Typ des injizierten Wertes gemacht werden können, gelten auch hier die am Schluß von Abschnitt 5.3.1 aus Seite 63f gemachten Einschränkungen über die Verwendbarkeit automorpher Werte.

Eine weitere wichtige Eigenschaft des Napier88-Compilers ist die Unterstützung der in Kapitel 4 beschriebenen Bindungsstile, insbesondere Bindung zur Programmkonstruktionszeit und zur Übersetzungszeit sowie von getrennter Übersetzung und Bindung [Cutts 92].

5.4.3 ML und CAML

In Standard ML of New Jersey [Appel et al. 93] existieren Schnittstellen zum Compiler als ML-Module. Ähnlich wie in der oben beschriebenen Schnittstelle zum Tycoon-Compiler läßt

[11] Der Typ *any* stellt konzeptionell einen Verbund mit einer nicht statisch fixierten Menge an Varianten dar. Er entspricht dem im vorigen beschriebenen Typ des automorphen Wertes (*dynamic_T*).

sich der Übersetzungs- bzw. Interpretationsvorgang durch ein Environment parametrisieren. Dieses gliedert sich in einen statischen Anteil (Namens- und Typinformation) und einen dynamischen (aktuelle Werte). Fehlermeldungen werden durch Auslösen einer Ausnahme an die aufrufende Umgebung propagiert.

Die Compilerschnittstelle von Standard ML eignet sich nicht zur typsicheren reflektiven Programmierung. Es ist nicht möglich, Environments mit Werten der aktuellen Laufzeitumgebung zu füllen und Resultate eines Auswertungsvorgangs in den den Compiler aufrufenden Kontext zu integrieren, was direkt auf das Fehlen dynamischer Typinformation zurückzuführen ist. Somit können Environments nur durch Auswertung von Ausdrücken erweitert werden. Im Modul *Environment* von Standard ML existieren jedoch Funktionen zum Filtern von Bindungen und zum Konkatenieren von Environments.

Ein Vorschlag zur Erweiterung von ML um automorphe Werte ist in [Leroy, Mauny 91] beschrieben und im System CAML implementiert [Weis 90]. Auch hier wird dynamische Typinformation dazu verwendet, dem Benutzer Systemfunktionalität typsicher zur Verfügung zu stellen, insbesondere auch den Compiler selbst durch eine Funktion *eval_syntax :ML*[12] $\rightarrow$ *dyn*[13]. Mit Hilfe dieser Funktion kann CAML leicht als eingebettete Sprache in Programmen verwendet werden. Ein Beispiel hierfür ist ihre Benutzung im Rahmen einer Beweiser-Konstruktionsumgebung zur interaktiven Definition von Beweistaktiken in CAML selbst. Außerdem bildet sie die Grundlage für die Makroübersetzung von CAML. Gegenüber der in dieser Arbeit entwickelten Schnittstelle fehlt jedoch die Möglichkeit der Übersetzung gegen eine existierende Laufzeitumgebung. Allerdings können auch hier Auswertungsresultate in den äußeren Kontext durch das Vorhandensein dynamischer Typinformation integriert werden.

[12] abstrakte ML Syntax
[13] Typ automorpher Werte

6. Die Tycoon-Programmierwerkbank

Werkzeuge zur Programmierunterstützung, seien es Bibliotheksdienste oder Anwendungsprogramme, sind dann besonders sinnvoll einsetzbar, wenn sie durch den Programmierer in beliebiger Kombination verwendet werden können. Eine Steigerung der Produktivität des Entwicklers läßt sich jedoch nur dann erzielen, wenn ihm gleichzeitig Regeln und Benutzungsschemata für diese Werkzeuge zur Verfügung stehen. Unter der Metapher *Programmierwerkbank* soll eine Umgebung verstanden werden, in der Werkzeuge nicht isoliert benutzt, sondern in einem gemeinsamen Rahmen integriert werden. Dabei spielt die Persistenz von Werten und Typrepräsentationen eine wichtige Rolle in allen Phasen des Programmkonstruktionsprozesses.

Nach einer Darstellung der Möglichkeiten für Anwendungen grafischer Benutzerinteraktion in persistenten und reflektiven Umgebungen wird in Abschnitt 6.2 eine grafische Schnittstelle zur Visualisierung und Manipulation des Objektspeichers vorgestellt. Abschnitt 6.3 stellt ein Konzept zur typsicheren Implementierung einer Modulverwaltungskomponente auf Anwendungsebene vor. Dies basiert auf dem TL-Sprachkern mit dynamischer Typisierung sowie der reflektiven Compiler-Schnittstelle. Den Abschluß dieses Kapitels bildet die Vorstellung einer neuartigen Programmiertechnik unter dem Namen *Hyperprogrammierung* mit einem Ausblick auf Implementierungsmöglichkeiten dieser Technik in Tycoon.

6.1 Anwendungen für grafische Benutzerinteraktion in persistenten und reflektiven Umgebungen

Für den Zugriff auf persistente Werte in Programmen werden in [Farkas et al. 92; Kirby et al. 92; Kirby 92a] verschiedene Varianten beschrieben, die sich insbesondere durch den Bindungszeitpunkt der persistenten Werte in das zu konstruierende Programm unterscheiden. Diese Varianten stellen unterschiedliche Anforderungen an die Art der Unterstützung des Programmkonstruktionsprozesses durch grafische Werkzeuge.

Bindung zur Laufzeit: Programme enthalten textuelle Spezifikationen zum Zugriff auf nichtlokale persistente Daten (Werte und Typen). Diese brauchen erst zur Laufzeit vorhanden sein, wodurch diese Variante die größte Flexibilität ermöglicht.

In Napier88 wird der Benutzer durch GUI-basierte Navigation in den Strukturen des Objektspeichers bei der Konstruktion dieser Zugriffspezifikationen unterstützt. Das Navigationswerkzeug (*store browser*) kann dabei den vom Benutzer durch direkte Manipulation verfolgten Pfad auch automatisch in eine textuelle Spezifikation überführen und ihm zur weiteren Verwendung im Programmtext zur Verfügung stellen.

Separate Bindungsphase: Diese Variante wird im Rahmen von Modulkonzepten vor allem in dateibasierten, imperativen Programmiersprachen (Modula 2, C, ...) verwendet. In persistenten Systemen kann diese Phase mit Funktionen höherer Ordnung simuliert werden [Kirby et al. 92] (siehe dazu auch Abschnitt 6.3).

In Tycoon stellt die *import*-Anweisung innerhalb eines Moduls die textuelle statische Zugriffsspezifikation auf persistente Daten dar. Die Bindung des (Haupt-)Programms und die Initialisierung aller seiner importierten Komponenten erfolgt durch eine *import*-Anweisung auf dem *top level*, dessen Bindungen persistent sind. Textuelle Beschreibungen der vorhandenen Bibliotheken (*libraries*) sind außerhalb des Objektspeichers im Dateisystem abgelegt. Ihnen kann der Programmierer benötigte Information über die Komponentenstruktur einer Anwendung oder Bibliothek und über die Sichtbarkeitsverhältnisse der in der *library* enthaltenen Module und Schnittstellen entnehmen. GUI-basierte Werkzeugunterstützung zur Navigation in Bibliotheken und zur Unterstützung des Programmierers bei der Suche nach Wert- und Typsignaturen existiert derzeit nicht, könnte aber z.B. auf der Basis von Hypertext in Form eines Quelltext-Navigators implementiert werden.

Bindung zur Übersetzungszeit: Programme referenzieren existierende Werte und Typen in einer Übersetzungsumgebung. Diese ermöglicht dem Compiler eine Abbildung von Namen auf Werte und Typen gemäß statischer Sichtbarkeitsregeln. Referenzierte Werte werden direkt in den ausführbaren Code gebunden.

In Napier88 wird dieser Bindungsmechanismus durch den *store browser* dahingehend unterstützt, daß der Benutzer beim Navigieren gefundene Werte und Typen mit einem Namen (*tag*) versehen und diesen im Programmtext direkt benutzen kann, ohne Zugriffspezifikationen codieren zu müssen. Die Abbildungsvorschrift von *tags* auf Werte und Typen dient zusammen mit dem Quelltext als Eingabe für den Compiler. Die *tags* können als freie Variablen betrachtet werden, so daß derselbe Quelltext seine Bedeutung je nach der zur Übersetzungszeit gültigen Umgebung erhält.

In Tycoon wird dieser Bindungsstil durch den interaktiven *top level* simuliert. GUI-Unterstützung zur Navigation im Objektspeicher und zur interaktiven Programmkonstruktion ist Gegenstand von Abschnitt 6.2.

Bindung zur Programmkonstruktionszeit: Erlaubt man das direkte Einfügen eines Wertes in den Quelltext anstelle eines Bezeichners für diesen Wert, so erhält man eine nichtlineare Programmrepräsentation. Diese wird in [Kirby et al. 92] in Anlehnung an Hypertext als Hyperprogramm bezeichnet. Hyperprogramme enthalten keine freien Variablenbezeichner und können somit zur Visualisierung von Funktionsabschlüssen verwendet werden [Connor et al. 94].

Diese Art der Programmkonstruktion ist ohne grafische Benutzerinteraktion in speziellen Programmeditoren nicht möglich, da herkömmliche Texteditoren nur die Erstellung flacher textueller Programmrepräsentationen erlauben. Da Hyperprogramme direkte Bindungen an Werte enthalten, müssen auch erstere im Objektspeicher gehalten werden. Dies setzt allerdings das Vorhandensein von Fehlererholungsmechanismen voraus, da ansonsten im Falle eines Systemabsturzes gerade in Bearbeitung befindliche Hyperprogramme unwiederbringlich verlorengehen. Eigenschaften, Voraussetzungen und Möglichkeiten dieser neuartigen Programmiermethodik werden in Abschnitt 6.4 behandelt.

GUI-Unterstützung	Bindungszeitpunkt			
	Komposition	Übersetzung	Bindungsphase	Laufzeit
Navigation im Objektspeicher	√	√		√
Direkte Wertmanipulation	√	√		
Hyperprogramm-Editierung	√			
Hypertext-Navigation			√	

Tabelle 6.1: GUI-Unterstützung für verschiedene Bindungsvarianten

Tabelle 6.1 gibt einen Überblick über mögliche GUI-Unterstützung für die oben behandelten Bindungsvarianten. Unter dem Begriff *direkte Wertmanipulation* sollen dabei die verschiedenen Möglichkeiten verstanden sein, Funktionen durch grafische Benutzerinteraktion aufzurufen, die aktuell selektierte Werte manipulieren. Ein Beispiel hierfür ist das oben beschriebene *tagging* in Napier88 oder auch das Sammeln von Werten und Typen durch Mausaktionen in eine Übersetzungsumgebung, wie es in Kapitel 6 beschrieben ist.

Reflektive Programmierung (siehe Kapitel 5) ist ein weiteres Beispiel für die Nützlichkeit grafischer Benutzerinteraktion zur Programmkonstruktion. Das Schreiben von Programmgeneratoren wird allgemein als schwerer angesehen als die Entwicklung konventioneller Programme [Kirby et al. 94b; Kirby 92b]. Der Grund dafür liegt unter anderem darin, daß der Programmierer auf zwei verschiedenen Ebenen für die Korrektheit eines Generators sorgen muß: Erstens darf der Generator nur korrekte, d.h. übersetzbare Programme erzeugen und zweitens sollen die generierten Programme ein korrektes Verhalten im Sinne der Anwendung zeigen. Dieser Sachverhalt wird durch die Erfahrungen mit der Entwicklung des Gateway-Generators im Rahmen dieser Arbeit bestätigt. Grafische Werkzeugunterstützung kann dem Programmierer eine Verstehenshilfe bei der Entwicklung von Generatoren bieten, z.B. durch die Eliminierung von „syntaktischem Rauschen“ oder die unterschiedliche Darstellung konstanter und variabler Teile des Generierungsresultates [Kirby 92a; Kirby et al. 94b].

6.2 Grafische Objektspeichermanipulation

Store Browser sind in persistenten Systemen erprobte Werkzeuge zur Visualisierung von Daten und Funktionen. Die Implementation eines solchen grafischen Werkzeuges wird durch die in den vorigen Kapiteln beschriebenen Dienste (GUI-Anbindung, dynamische Typen, Environments, aufrufbarer Compiler) direkt unterstützt.

Im vorigen Kapitel wurde schon auf die Bedeutung von Environments als Hauptstrukturierungsmittel für Werte und Typrepräsentationen im Objektspeicher hingewiesen. Durch die grafische Visualisierung des Objektspeicherinhalts anhand seiner Environmentstruktur soll der Benutzer nun in die Lage versetzt werden, diesen durch direkte Interaktion manipulieren zu können. Zu diesen Manipulationen gehören:

- Visualisierung der Struktur von Wert- und Typbindungen, wobei ein rekursiver Abstieg in der Environmentstruktur möglich ist,
- Kopieren bzw. Verschieben von Bindungen zwischen Environments,

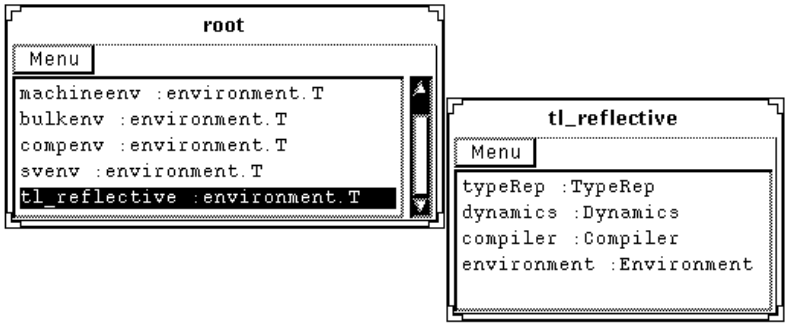

Abbildung 6.1: Visualisierung eines Environments

- Erzeugung neuer Bindungen durch Auswertung von Ausdrücken in einem existierenden Environment,
- Änderung der Wertkomponente vorhandener Bindungen,
- Löschen vorhandener Bindungen.

Die in Abschnitt 5.2.1 genannten Grundoperationen werden somit auch an der grafischen Benutzerschnittstelle unterstützt.

Die Navigation im Objektspeicher erfolgt ausgehend von einem Wurzel-Environment, das initial aus Komponenten des Compilers selbst besteht. Dabei sind nur die Komponenten in Environments gebunden, die der Benutzer zugreifen darf, d.h. die öffentliche Schnittstelle zum Compiler, wie sie in Abschnitt 5.2.2 beschrieben wird. Außerdem existieren vordefiniert eine Reihe von Standardmodulen, bei denen sichergestellt ist, daß ihre gemeinsame Benutzung durch Compiler und Anwendungsprogramme keine ungewollten Seiteneffekte, z.B. Veränderung von Zustandsvariablen in Modulen, produziert.

Environments werden als (evtl. mit Rollbalken versehene) Tabellen dargestellt (Abbildung 6.1). Einträge in diesen Tabellen symbolisieren Bindungen und enthalten deren Signatur in textueller Form, wobei Typen abgekürzt dargestellt sind. Durch Doppelklick auf eine Bindung wird die Struktur der zugehörigen Wert- und Typinformation textuell[1] angezeigt, evtl. in einem neuen Fenster.

In den meisten Fällen hat der Programmierer Ausdrücke auszuwerten, die freie Variablen referenzieren. Zu diesem Zweck kann er durch Navigation in der Environmentstruktur aufgefundene Bindungen interaktiv über das *drag and drop protocol* in ein Ziel-Environment kopieren oder verschieben. Dies ist möglich, da Bindungen sowohl auf Wert- als auch auf Typebene durch die Wahl einer kontextunabhängigen Typrepräsentation (Kap. 5.1.5) voneinander völlig unabhängig sind.

Der Auswertungsvorgang wird ausgelöst, indem Programmtext in das Klemmbrett (*clipboard*) kopiert und anschließend in das gewünschte Environment eingefügt wird. Bei fehlerfreier Übersetzung werden die dadurch neu entstandenen Bindungen im betreffenden Environment angezeigt, ansonsten Fehlermeldungen in einer Tabelle ausgeben. Falls der Programm-

[1] Die Verbesserung der Datenvisualisierung und der Ausbau dieses Werkzeuges zu einem vollständigen grafischen Browser ist Gegenstand eine parallel laufenden Diplomarbeit.

text aus dem Eingabefenster der Werkbank stammt[2], kann durch Anklicken einer Fehlermeldung die Stelle im Quelltext angesprungen werden, an der der Fehler auftrat.

Der Wert einer existierenden Bindung (d.h. auch eine Typrepräsentation bei Typbindungen) kann gegen einen anderen Wert ausgetauscht, d.h. aktualisiert werden, indem vor dem Einfügen des Quelltextes eine Bindung im betreffenden Environment ausgewählt wird. Dieser Austausch erfolgt jedoch nur dann, wenn der Typ des ausgewerteten Ausdrucks Subtyp des Typs der vorher ausgewählten Bindung ist. Durch das Konzept der statischen Bindung in TL werden andere Bindungen durch solch eine Aktualisierungsoperation nicht beeinflußt, auch wenn es sich um eine variable Wertbindung handelt. Dies steht im Gegensatz zu der in [Dearle et al. 92] vorgeschlagenen Architektur eines zweiphasigen Bindungsvorgangs.

Bindungen werden gelöscht durch Selektieren und anschließende Auswahl des entsprechenden Punktes im Menü des jeweiligen Environment-Fensters.

6.3 Modulverwaltung

Durch die Modularisierungsmechanismen von TL werden keine neuen Benennungs-, Bindungs- und Typisierungskonzepte definiert, sondern nur die Orthogonalität existierender Konzepte eingeschränkt [Matthes 93]. Dies legt die Vermutung nahe, daß die Konstrukte **module** und **interface** auf Konstrukte des Sprachkerns, inklusive der im vorigen Kapitel beschriebenen Erweiterungen, abgebildet werden können. Gelingt diese Abbildung, so läßt sich die Modulverwaltungskomponente des Tycoon-Systems allein unter Verwendung des Compiler-API (Schnittstellen *TypeRep*, *Dynamic*, *Environment* und *Compiler*, siehe Anhang B und C) implementieren, d.h. auf der Anwendungsebene (siehe Abbildung 2.2). Dies würde zu einer größeren Transparenz der Implementierung führen, verglichen mit der bisherigen, weitgehend typunsicheren Implementierung unter Verwendung compiler-interner Repräsentationen.

In den folgenden Abschnitten werden verschiedene Abbildungsvarianten für die syntaktischen Konstrukte **module** und **interface** auf elementare Sprachkonstrukte von TL diskutiert. Die werkzeuggestützte Steuerung des Übersetzungsvorganges (*make*), Versionskontrolle und Bibliotheksmanagement, aufbauend auf den nachfolgenden Abbildungsvorschlägen, sind Gegenstand weiterer Arbeiten im Tycoon-Umfeld.

Abweichend von [Matthes 93] wird von einem vereinfachten Modulkonzept ohne das *library*-Konstrukt ausgegangen, bei dem in der Import-Liste jeweils sowohl Modul als auch Modulschnittstelle angegeben werden müssen. Zur weiteren Diskussion sei die in Abbildung 6.2 dargestellte Modulkonfiguration verwendet, mit dessen Hilfe die durch abstrakte Typvariablen und Rautenimport auftretenden Probleme illustriert werden können. Die Übersetzung und Bindung von Modulen und Modulschnittstellen unterscheidet sich in einigen Punkten wesentlich von der Auswertung von Ausdrücken des Sprachkerns:

- Module können in jeder beliebigen Reihenfolge übersetzt werden.
- Die Übersetzungsumgebung eines Moduls soll nur Typbindungen für benutzte Schnittstellen enthalten, inklusive der eigenen.
- Anstelle einer inkrementellen Auswertung erfolgt die Bindung aller zu einem System gehörenden Module in einer separaten Phase. Dabei muß sichergestellt werden, daß ein

[2] Der Text könnte auch aus einer Anwendung stammen, die nicht unter der Kontrolle der Werkbank steht.

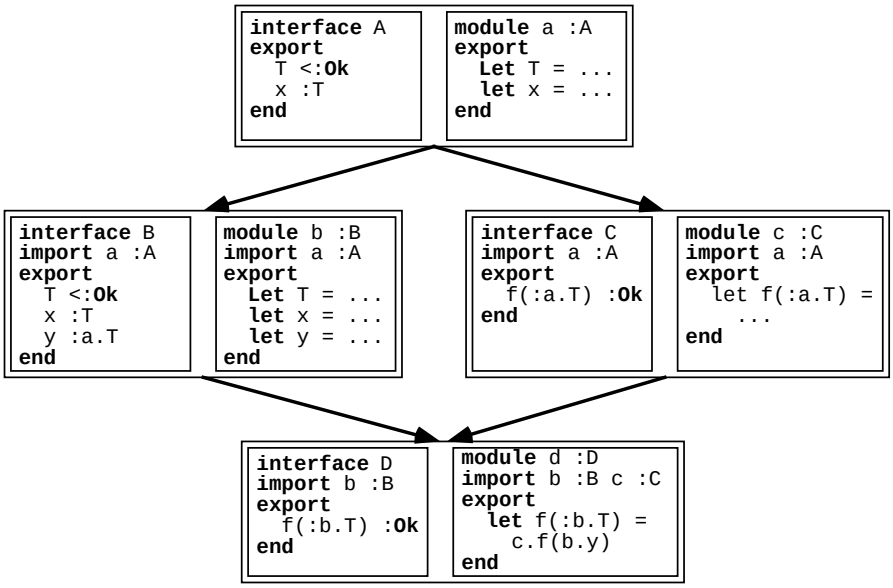

Abbildung 6.2: Beispiel zur Modulverwaltung

Modul nur mit Modulen gebunden werden kann, deren Schnittstellen in einer Subtypbeziehung zu denjenigen stehen, gegen die es übersetzt worden ist.

Um diese Eigenschaften zu berücksichtigen, wird ein Modul vor der Übersetzung in einen Ausdruck umgeformt, der von der folgenden Schnittstelle *ModuleSystem* Gebrauch macht.

Let *ModuleSystem* = ***Tuple***
T <:***Ok***
(In einem Modulsystem werden übersetzte und gebundene Module verwaltet. *)*
new(linkEnv :environment.T) :T
(Erzeugt ein neues System, dessen Module in 'linkEnv' gebunden werden.*)*
addLinkedModule(s :T name :String mod :dynamic_T) :dynamic_T
(Fügt die Bindung < name, mod > in 's' ein. Gibt 'mod' zurück. *)*
addCompiledModule(s :T name :String import(:T) :dynamic_T) :***Ok***
(Fügt das übersetzte Modul 'import' als Funktion unter dem*
*Namen 'name' dem System 's' hinzu. *)*
getCompiledModule(s :T name :String) :***Fun****(:T)* :***Ok***
(Gibt das übersetzte Modul 'name' aus dem System 's' zurück. *)*
import(s :T name :String) :dynamic_T
(Falls Modul 'name' schon in 's' gebunden, wird es als automorpher*
Wert zurückgegeben, ansonsten die mit 'name' assoziierte Import-Funktion
*ausgeführt. *)*
end
let *system :ModuleSystem* = ***tuple*** *...* ***end***

6.3.1 Statische Bindung importierter Module in Modulschnittstellen

Eine Modulschnittstelle entspricht einem benannten Tupeltyp. Um importierte Module in seinen Sichtbarkeitsbereich zu bringen und gleichzeitig separate Übersetzung zu gewährleisten, wird dieser Tupeltyp wiederum in einen Tupeltyp gehüllt, der zusätzliche Signaturen für jedes importierte Modul enthält[3].

```
Let _C = Tuple
  Let A = _A.Interface
  a :A
  Let Interface = Tuple f(:a.T) :Ok end
end
```

Ein Modul wird durch eine Funktion dargestellt, bei deren Ausführung Bindungen an importierte Module etabliert werden, wobei das Wissen, mit welchen Komponenten die Bindung erfolgen muß, zur Übersetzungszeit bekannt ist. Modul *c* aus obigem Beispiel wird unter Benutzung der oben eingeführten Bindung für *system* in folgende Funktion transformiert:

```
fun(s :system.T)
  begin
    Let A = _A.Interface
    let a = dynamic_be(system.import(s "a") :A)
    Let C = Tuple f(:a.T) :Ok end
    let f(:a.T) = ...
    let result :C = tuple
      let f = f
    end
    system.addLinkedModule(s "c" dynamic_new(result))
  end
```

Die Anweisung **dynamic_be** stellt sicher, daß Modul *c* nur mit einem Modul a gebunden werden kann, das zur Bindezeit eine mindestens so starke Schnittstellenspezifikation erfüllt wie diejenige, die zur Übersetzungszeit von *c* gegolten hat. Es findet also eine modulübergreifende Typüberprüfung zum Bindezeitpunkt mit Hilfe elementarer Sprachkonstrukte statt. In anderen Programmiersprachen, die separate Übersetzung unterstützen, wird hierfür auf Ad-hoc-Mechanismen zurückgegriffen, wie z.B. die bereits in Kapitel 3 angesprochene Codierung von Signaturen in Funktionsnamen (*name mangling*) in C++, Zeitstempel oder Quelltext-CRC[4]-Prüfsummen. Der Hauptvorteil der Typüberprüfung zur Bindezeit ist, daß dadurch die Bedingungen für die Notwendigkeit von Neuübersetzungen weniger restriktiv sind als mit oben genannten Ad-hoc-Mechanismen. Ist z.B. vor der Bindung von *c* das Modul a mit einer im Sinne der Subtyprelation verfeinerten Schnittstelle übersetzt worden, so ist eine Neuübersetzung von *c* mit dieser verfeinerten Schnittstelle für a nicht notwendig.

Aufgrund der einschränkenden Wertbindung *let result :C = ...* wird geprüft, ob die Bindungen des Moduls seiner Schnittstelle entsprechen. Nachdem das Resultattupel (d.h. das gebundene Modul) gebildet wurde, sorgt das Modul selbst dafür, daß es in gebundener Form in das Environment des Modulsystems eingefügt wird.

[3] Unterstriche vor Bezeichnern sollen diese als im urspünglichen Quelltext nicht sichtbar kennzeichnen.
[4] CRC: Cyclic Redundancy Check

Die Hauptschwäche dieses Ansatzes ist jedoch, daß bei der Übersetzung eines Moduls die eigene Schnittstelle erneut übersetzt werden muß (im Beispiel *Let C = Tuple ...*), falls diese Module importiert. Dies liegt an der Namensäquivalenzregel für abstrakte Typvariablen. Führte man nämlich statt einer Neuübersetzung die Typbindung *Let C = _C.Interface* ein, so wäre die einschränkende Bindung für *result* fehlerhaft, da $C.a.T \neq a.T$, was während der Prüfung der Signatur von *f* festgestellt würde.

Verschärft wird die Situation noch dadurch, daß auch die importierten Modulschnittstellen erneut übersetzt werden müssen, falls diese Module importieren, was bei der Übersetzung von Modul *d* auftritt.

```
fun(s :system.T) :dynamic_T
 begin
  Let _A = _A.Interface
  let _a = dynamic_be(system.import(s "a") :_A)
  Let B = Tuple T <:Ok  x :T  y: _a.T end
  let b = dynamic_be(system.import(s "b") :B)
  Let C = Tuple f(:_a.T) :Ok end
  let c = dynamic_be(system.import(s "c") :C)
  Let D = Tuple ... end
  let f(:b.T) = c.f(b.y)
  let result :D = ...
  system.addLinkedModule(s "d" dynamic_new(result))
 end
```

Wird diese Neuübersetzung durch Bindungen *Let B = _B.Interface* und *Let C = _C.Interface* unterlassen, so schlägt zur Bindezeit von *d* die Anweisung *dynamic_be(system.import("c") :C)* fehl, da in der Typrepräsentation des durch die Funktion *system.import* bekommenen automorphen Wertes ein anderer Pfad für die abstrakte Typvariable *T* enthalten ist als in der Typrepräsentation, die für *C* zum Aufrufzeitpunkt des dynamischen Typtests vorliegt[5]. Um diesen Fehler zu umgehen, könnte man bei der Transformation des Moduls *b* bzw. *c* vor der Injektion des Resultates in einen automorphen Wert durch *type casting* dafür sorgen, daß es den Typ *B.Interface* bzw. *C.Interface* bekommt. Durch den Rautenimport von a schlägt dann allerdings schon statisch die Überprüfung des Aufrufs *c.f(b.y)* fehl, da in diesem Fall $B.a.T \neq C.a.T$ gilt.

6.3.2 Variable statische Bindung importierter Module

Anstelle den Modulimport als statische unveränderliche Bindung zu formulieren, wäre stattdessen auch eine Zuweisung an eine veränderliche Bindung denkbar. Ein Problem stellt jedoch die Initialisierung dieser variablen Bindung dar, da zum Übersetzungszeitpunkt einer Modulschnittstelle der Wert des importierten Moduls nicht notwendigerweise existiert. Zur Initialisierung einer variablen Bindung mit einem Nullwert könnte die Schnittstelle *Environment* um eine zusätzliche Funktion *addNilLocationBinding(name :String)* ergänzt werden. Die Modulschnittstelle *C* des Beispiels hätte dann folgendes Aussehen:

[5] siehe dazu Abschnitt 5.1.5

```
let var a :A = nil (* fiktive Syntax *)
Let C = Tuple
  f(:a.T) :Ok
end
```

Im Modul *c* würde dann der Variablen a der tatsächliche Modulwert zugewiesen:

```
fun(s :system.T) :dynamic_T
  begin
    a:= dynamic_be(system.import(s "a") :A)
    let f(:a.T) = ...
    let result :C = tuple ... end
    system.addLinkedModule(s "c" dynamic_new(result))
  end
```

Die Übersetzung dieses Codes wird allerdings vom Typüberprüfer zurückgewiesen, da in TL nicht auf abstrakte Typkomponenten veränderlicher Variablen zugegriffen werden darf. Durch Zuweisungen könnte sonst die tatsächliche Repräsentation eines ADT zur Laufzeit geändert werden, was Funktionen auf diesem ADT unweigerlich zum Absturz bringen würde. Die Korrektheit der Verwendung von a wird im Beispiel dadurch sichergestellt, daß die Zuweisung nur einmalig zur „Initialisierung" des Moduls erfolgt und eine Verwendung von a vor dieser Zuweisung ausgeschlossen ist. Um *c* übersetzen zu können, müßte dem Typüberprüfer jedoch dieser Sachverhalt vorher mitgeteilt werden, d.h. Module sind dann mit abgeschwächten Regeln zu übersetzen. Dies widerspräche allerdings der ursprünglichen Zielsetzung dieses Abschnitts, die Modulverwaltung rein auf Anwendungsebene typsicher implementierbar zu machen. Außerdem kann man bei dieser Variante nicht mehr von getrennter Übersetzung sprechen, da die Identitäten der variablen Modulbindungen über alle Übersetzungsvorgänge hinweg erhalten bleiben müssen.

Es zeigt sich, daß die statische Bindung importierter Module in Modulschnittstellen nicht zum gewünschten Ergebnis führt. Was eigentlich benötigt wird, ist eine Möglichkeit, importierte Modulbezeichner in Schnittstellen *dynamisch* zu binden, um unerwünschte Neuübersetzungen von Modulschnittstellen bzw. eine Veränderung der TL-Semantik zu vermeiden.

6.3.3 Dynamische Bindung importierter Module in Modulschnittstellen

Die TL-Syntax erlaubt es, Typoperatoren auch mit Werten zu parametrisieren, um bei der Definition eines Typoperators auf Typkomponenten von Tupelwerten zugreifen zu können. Eine Modulschnittstelle, die transitiv Module importiert, wie z.B. *D*, hätte dann folgendes Aussehen:

```
Let _D(_a :A() b :_B(_a)) = Tuple
  Let B = B(_a)
  f(:b.T) :Ok
end
```

Das zugehörige Modul ließe sich dann in folgende Darstellung transformieren:

```
fun(s :system.T) :dynamic_T
  begin
    let _a = dynamic_be(system.import(s "a") :A)
    Let B = _B(_a)
    let b = dynamic_be(system.import(s "b") :B)
    Let C = _C(_a)
    let c = dynamic_be(system.import(s "c") :C)
    Let D = _D(_a b)
    let f(:b.T) = c.f(b.y)
    let result :D = ...
    system.addLinkedModule(s "d" dynamic_new(result))
  end
```

Auf diese Weise werden Modulschnittstellen jeweils eindeutig mit denselben Modulwerten instanziiert, bevor die Bindungen eines Moduls übersetzt werden, so daß die Identität von Pfaden abstrakter Typvariablen gewahrt bleibt. Jedoch gehen auch bei dieser Lösung transitiv importierte Module in den transformierten Quellcode ein, so daß seitens der Modulverwaltung zusätzlicher Aufwand betrieben werden muß, will man dem Benutzer nicht zumuten, auch transitiv von importierten Modulschnittstellen importierte Module aufzuführen (im Beispiel hieße dies, Modul a in der Import-Liste von *D* zu erwähnen).

Unglücklicherweise läßt die TL-Syntax die Typoperatorapplikation mit Wertvariablen als Parameter nicht zu, da diese syntaktisch nicht von Typvariablen unterschieden werden. Ein anderer Weg, dynamische Bindung von Modulen in Modulschnittstellen zu erreichen, wäre die Umformung von Modulschnittstellen in Funktionen, die ein Tupel mit einer Typkomponente zurückliefern.

```
let _D(a :A  x :_B(a).Interface) = tuple
  Let Interface = Tuple ... end
end
```

Der Ausdruck *_B(a).Interface* ist allerdings statisch nicht zulässig, da allgemeine Typausdrücke nicht von Werten abhängen dürfen. Die Orthogonalisierung der TL-Syntax zur Ermöglichung der Typoperatorapplikation mit Wertparametern ist daher unumgänglich.

6.3.4 Steuerung des Übersetzungs- und Bindevorgangs

Die Anweisung des Benutzers, das Beispielmodulsystem zu übersetzen und zu binden, könnte von einem Werkzeug unter Verwendung der Schnittstellen *ModuleSystem, Environment und Compiler* in folgender Weise abgearbeitet werden:

```
let linkEnv = environment.new()
let sys = system.new(linkEnv)
let compileEnv = environment.new()
environment.addValueBinding(compileEnv "system" dynamic_new(system))

compiler.eval(compileEnv getCode("A"))
let a :dynamic_T = compiler.eval(compileEnv getCode("a"))!valueCase.value
```

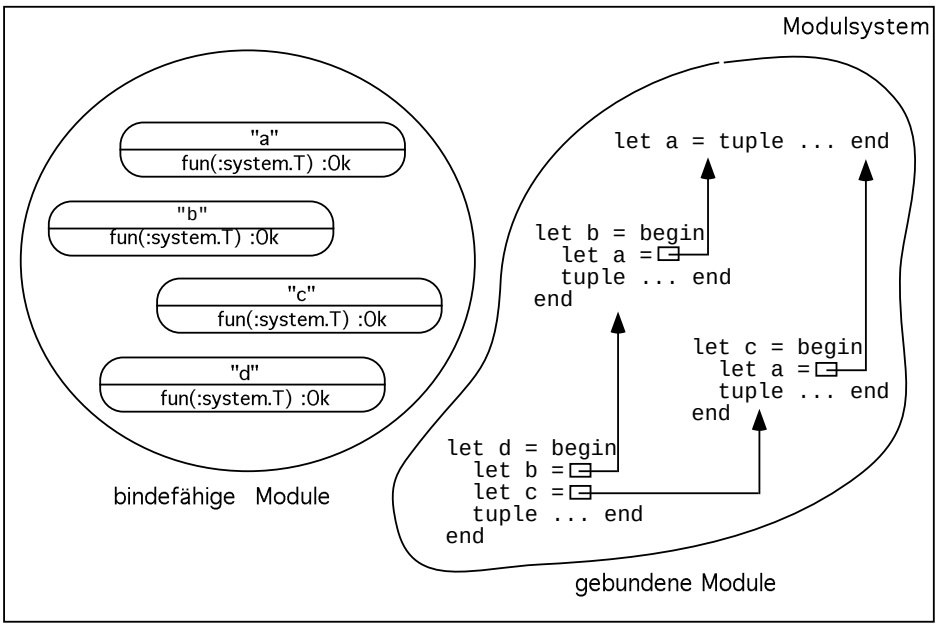

Abbildung 6.3: Beispielmodulsystem im Objektspeicher

system.addCompiledModule(sys "a" ***dynamic_be****(a :**Fun**(:system.T) :**Ok**))*
... (die letzten drei Zeilen für Module b, c und d wiederholen *)*
system.import("d") (Bindung von d mit allen transitiv importierten Module *)*

Die Funktion *getCode* dient hierbei als Abstraktion für das Auffinden des Quelltextes in Modulsyntax und dessen Transformation in die im letzten Abschnitt vorgestellte Form, wobei auch die Ergänzung der Modul-Importlisten durch von Schnittstellen transitiv importierte Module vorgenommen wird. Abbildung 6.3 veranschaulicht den Zustand des Beispielmodulsystems nach Beendigung des Bindevorgangs. Jedes übersetzte Modul stellt eine geschlossene Funktion dar, referenziert also keine freien Variablen. Da es Typrepräsentationen seiner importierten Module enthält, trägt es alle notwendige Information in sich, um in einem beliebigen Modulsystem typsicher gebunden (d.h. ausgeführt) werden zu können. Insbesondere kann es dadurch auch zwischen verschiedenen Modulsystemen transferiert oder zwischen verschiedenen Objektspeichern ausgetauscht werden.

6.4 Hyperprogrammierung

Der Begriff der Hyperprogrammierung (engl. *hyper programming*) wurde durch Forschungsarbeiten an der Universität von St. Andrews (Schottland) geprägt. Dabei entstand im Rahmen einer Dissertation [Kirby 92a; Kirby et al. 93] die erste bekannte Implementierung zur Unterstützung dieser Programmiermethodik im Rahmen der persistenten Programmiersprache Napier88.

Hyperprogrammierung basiert auf der nur in persistenten Systemen gegebenen Möglichkeit, die Laufzeitumgebung (d.h. den Objektspeicher) in den Programmkonstruktionsprozeß

einzubeziehen. Diese Möglichkeit wird schon in Abschnitt 6.2 genutzt. Der entscheidende Unterschied liegt jedoch im Bindungszeitpunkt der Daten im Objektspeicher an das zu konstruierende Programm: Daten bzw. ausführbare Codefragmente, die schon zum Programmkonstruktionszeitpunkt im Objektspeicher vorliegen, werden nämlich schon zu diesem frühestmöglichen Zeitpunkt in den Programmtext gebunden.

Nachfolgend werden die aus dieser Bindungsvariante resultierenden Möglichkeiten und Konsequenzen erläutert. Ein Ausblick auf Implementierungsmöglichkeiten dieses Programmierstils in Tycoon beschließt diesen Abschnitt.

6.4.1 Charakterisierung und Voraussetzungen

In Analogie zu Hypertext als nichtlineare Repräsentationsform für Text handelt es sich bei einem Hyperprogramm um eine nichtlineare Repräsentationsform für Programme. Die Nichtlinearität bezieht sich dabei auf das Vorhandensein von direkten Referenzen auf existierende Werte im Programmtext. Ein Hyperprogramm ist demnach eine Mischform aus Text und ausführbarem Code. Im Gegensatz zu flachen textuellen Programmrepräsentationen können Hyperprogramme nicht im Dateisystem des zugrundeliegenden Betriebssystems gespeichert werden, sondern sind in dieser Form nur im Objektspeicher verfügbar. Eine zweite unmittelbare Folge der Nichtlinearität ist, daß alternative Benutzerinteraktionsformen außer der Text-Editierung zur Programmerstellung benötigt werden, um direkte Referenzen auf Werte denotieren zu können. Grundlegende Vorausetzung zur Unterstützung von Hyperprogrammierung sind demnach:

- Referentielle Integrität des Objektspeichers, da Hyperprogramme invalidiert sind, wenn sie Referenzen auf nicht existente Daten enthalten. Referentielle Integrität ist durch das *Tycoon Store Protocol* gesichert, denn Objekte können vom Benutzer nicht explizit gelöscht werden.

- Hyperprogrammrepräsentationen müssen als Werte der Sprache darstellbar sein. Dies erfordert wiederum reflektive Fähigkeiten zur Transformation dieser Repräsentationen in ausführbaren Code. Die Grundlagen hierfür wurden im vorigen Kapitel dargestellt. Zur Unterstützung von Hyperprogrammierung müßte die dort beschriebene Schnittstelle zum Compiler dahingehend erweitert werden, daß sie auch die Übersetzung nichtlinearer Programmrepräsentationsformen ermöglicht.

- Werkzeugunterstützung zur Programmkonstruktion: Der Benutzer muß in der Lage sein, im Objektspeicher zu navigieren und gefundene Werte durch direkte Interaktion in das zu bearbeitende Programm einzufügen.

Einerseits stellt Hyperprogrammierung wegen ihrer hohen Anforderungen an effektive Benutzerinteraktionsmöglichkeiten eine Herausforderung zur Entwicklung von Programmierwerkzeugen (Browser, Editor) dar, andererseits bietet dieser Programmkompositionsstil einige Möglichkeiten, die Vorteile persistenter Systeme gegenüber dateibasierten Systemen auszunutzen. Persistente Bindung und referentielle Integrität sind dabei Eckpfeiler dieser Technologie [Morrison et al. 95].

6.4.2 Vorteile

Vorteile der Hyperprogrammierung, wie sie durch deren Entwickler genannt werden [Kirby et al. 92; Connor et al. 94; Farkas et al. 92], fallen grundsätzlich in zwei Kategorien:

1. Vorteile, die sich nur aus dem Vergleich mit bisherigen Programmierstilen im Kontext von PS-algol und Napier88 unter der Annahme einer *geschlossenen* integrierten persistenten Umgebung ergeben,
2. Vorteile, die sich durch die Benutzung direkter statt symbolischer Referenzen ergeben und somit auch in anderen persistenten Systemen realisiert werden können.

Unter die erste Kategorie fallen Vorteile, die im Zusammenhang mit Napier's Environment-Konzept stehen. Durch die Strukturierung des Objektspeichers in geschachtelte Environments stellt sich für den Benutzer ständig das Problem, den genauen Zugriffspfad auf persistente Werte zu spezifizieren. Durch Bindung von Werten zur Programmkompositionszeit

- erfolgen Typüberprüfung und Zugriff auf persistente Werte schon vor der Programmlaufzeit, wodurch eine mögliche Laufzeitfehlerquelle vermieden wird. Wegen der referentiellen Integrität ist zusätzlich gewährleistet, daß diese Zugriffe auch zu einem späteren Zeitpunkt nicht fehlschlagen werden.
- ergibt sich eine erhebliche Reduzierung der Codegröße im Vergleich zu linearen Programmrepräsentationen, die einen relativ hohen Anteil an Zugriffscode auf persistente Daten (20%, [Sjøberg et al. 94]) enthalten.

In Tycoon existiert das Problem weitschweifiger Zugriffspfadspezifikationen nicht, da das Persistenzmodell modulbezogen und der Modulnamensraum flach strukturiert ist. Der Zugriff auf persistente Daten eines Moduls zur Laufzeit kann nicht fehlschlagen, wenn das Modul in der separaten Bindungsphase verfügbar gewesen ist.

Zur zweiten Kategorie der Vorteile von Hyperprogrammen gehören die Tatsachen, daß

- sie sich zur Visualisierung von Funktionsabschlüssen eignen,
- sie eine gemeinsame Sicht auf Quellcode und ausführbarem Code erlauben und der Programmierer somit diese beiden Konzepte nicht mehr getrennt handhaben muß,
- sie nichtkorrumpierbare Verbindungen zwischen Quellcode und ausführbaren Code ermöglichen,
- sie neuartige Möglichkeiten zur Versionskontrolle, zum Konfigurationsmanagement und zur Dokumentation eröffnen.

Visualisierung von Funktionsabschlüssen

Trifft der Benutzer beim Navigieren in der persistenten Umgebung auf eine Funktion oder Prozedur, so ist er im allgemeinen zumindest an folgender Information interessiert:

- Welche Signatur hat die Funktion/Prozedur?

- Welche freien Variablen referenziert sie?
- Welche Seiteneffekte können dadurch bei ihrem Aufruf auftreten?

Die Antwort auf die erste Frage kann z.B. durch das in Abschnitt 6.2 beschriebene Werkzeug beantwortet werden, da mit jeder Bindung in einem Environment auch Typinformation gespeichert wird, die dem Benutzer präsentiert werden kann. Ein Nachteil bisheriger Browser-Technologie [Kirby, Dearle 90; Dearle et al. 90] war, daß es noch keine adäquaten Konzepte zur Visualisierung von Funktionsabschlüssen gab, d.h. von Prozeduren oder Funktionen, die freie Variablen referenzieren. Selbst wenn neben dem ausführbaren Code noch Quelltext-repräsentationen (oder Verweise auf Orte, wo diese zu finden sind) zur Visualisierung zur Verfügung stehen, reichen diese allein nicht aus, um die unterschiedliche Semantik verschiedener Funktionsabschlüsse zu erfassen, da sie aus demselben Quelltext hervorgegangen sein können. Die Frage nach eventuellen Seiteneffekten einer Prozedur, die gerade in Datenbanksystemen eine wichtige Rolle spielt, ist somit auch nicht zu beantworten. Benötigt wird daher ein 1:1-Abbildung zwischen Quelltext und ausführbarem Code, wie durch folgendes Beispiel (aus [Connor et al. 94] entnommenen und modifiziert) deutlich wird:

```
let counterGenerator() :Fun() :Int =
  begin
    let var x = 0
    fun() :Int
      begin
        x:= x+1
        x
      end
  end

let counter1 = counterGenerator()
let counter2 = counterGenerator()
```

Hierbei modifizieren *counter1* und *counter2* jeweils unterschiedliche Variablen, obwohl sie *dieselbe* Quelltextrepräsentation teilen, denn sie enthalten jeweils verschiedene Instanzen von x in ihrem Funktionsabschluß.

Hyperprogramme erlauben es, die gewünschte Korrespondenz zwischen Quellcode und ausführbarem Code herzustellen. Dadurch gelingt es, die bisher vom Benutzer getrennt wahrgenommenen Konzepte *Quellcode* und *Funktionsabschluß* zu unifizieren [Connor et al. 94]. Freie Variablen werden dann nicht mehr symbolisch, sondern durch direkte Referenzen dargestellt. Die Semantik einer Funktion/Prozedur ist somit unabhängig von einer Auswertungsumgebung und wird direkt durch ihre Hyperprogrammrepräsentation erfaßt. Abbildung 6.4 zeigt eine solche Repräsentation für *counter1*.

Referentiell integre Beziehung zwischen ausführbarem Code und Quellcode

Im Programmentwicklungsprozeß entstehen eine Reihe von Programmformen (Quellcode, übersetzter Code, ausführbarer Code, Laufzeitdatenobjekte), zwischen denen unterschiedliche Beziehungsformen möglich sind. Diese Beziehungsformen werden in [Kirby et al. 92] als *ursächlicher Zusammenhang, Assoziation* und *direkte Bindung* identifiziert.

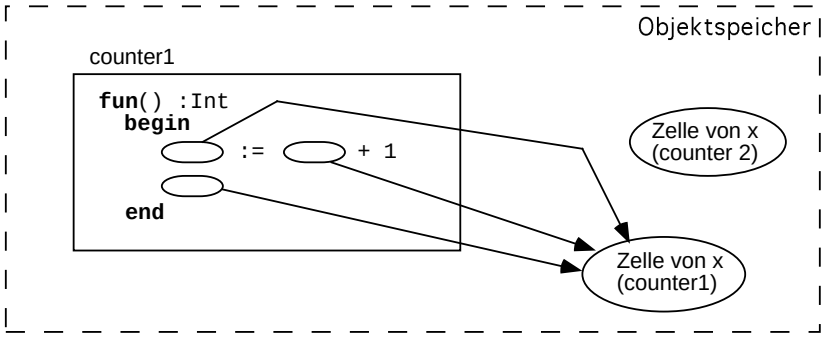

Abbildung 6.4: Hyperprogramm zum Beispiel

Ein *ursächlicher Zusammenhang* zwischen zwei Entitäten A und B besteht dann, wenn eine Änderung von A, hervorgerufen durch einen bestimmten Prozeß, in einer Änderung von B resultiert. Ein Beispiel hierfür ist der Zusammenhang zwischen Quellcode und übersetztem Code, hervorgerufen durch einen Compiler.

Eine *Assoziation* ist eine allgemeine Beziehung, die durch einen externen Mechanismus verwaltet wird, z.B. eine Assoziation zwischen einem ausführbaren Programm und zugehörigem Quellcode, verwaltet durch einen Debugger. Diese Beziehung kann leicht von außen korrumpiert werden, falls gegen bestimmte, durch den externen Mechanismus definierte Konventionen verstoßen wird, z.B. durch Löschen von Quelltexten, ohne den Debugger zu informieren.

Direkte Bindungen sind nur in einer Laufzeitumgebung möglich. Unterstützt diese referentielle Integrität, wie dies z.B. bei allen Tycoon-Objektspeicherimplementationen der Fall ist, so ist die Gültigkeit direkter Bindungen auch bei Änderungen garantiert, da das explizite Löschen von Objekten nicht möglich ist und somit keine ungültigen Referenzen entstehen können.

In dateibasierten Systemen existieren direkte Bindungen nur zwischen den flüchtigen Laufzeitdatenobjekten. Alle anderen Beziehungen zwischen den beteiligten Entitäten entstehen durch externe Mechanismen (statische Binder, dynamische Binder). In persistenten Systemen, in denen ausführbare Programme als Werte der Sprache repräsentiert werden können (z.B. als Prozedur), sind die Assoziationen zwischen Bibliothekscode und ausführbarem Code durch direkte Bindungen ersetzt. Wird weiterhin der Quellcode als Wert der Sprache dargestellt, so kann auf externe Speichermechanismen wie Dateisysteme ganz verzichtet werden. Als einzige nichtrobuste Beziehungen zwischen Software-Entitäten bleiben jedoch die Assoziationen zwischen ausführbaren Programmen und Quellcode.

Durch die Benutzung von Hyperprogrammen als Quellcode- Repräsentation können Assoziationen zwischen ausführbarem Code und Quellcode durch robuste direkte Bindungen ersetzt werden. Dies folgt direkt aus der Tatsache, daß Hyperprogramme, genauso wie ausführbare Programme, direkte Bindungen an Datenobjekte enthalten können. Jeder direkten Bindung eines ausführbaren Programms an ein Daten- oder Prozedurobjekt steht dann eine korrespondierende Bindung vom zugehörigen Hyperprogramm an dieses Datenobjekt gegenüber, wie Abbildung 6.5 veranschaulicht.

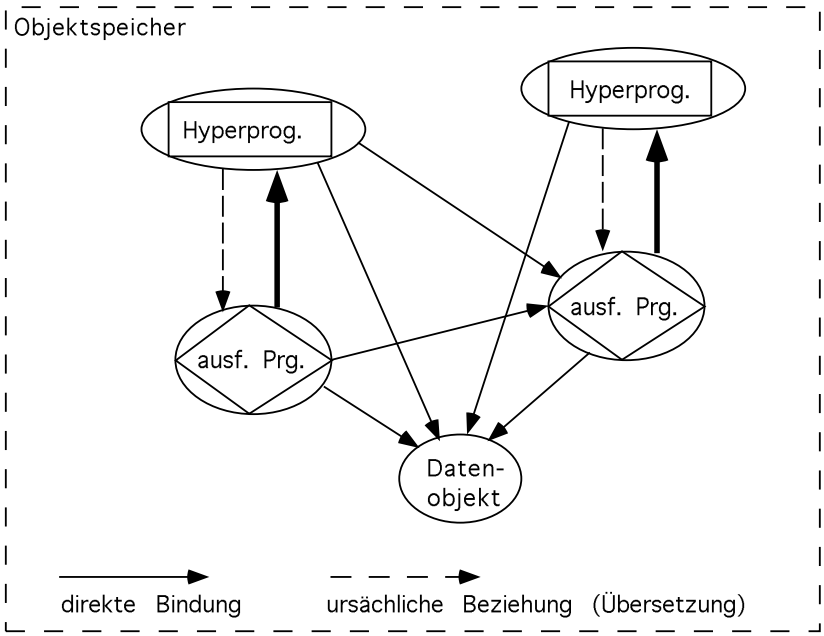

Abbildung 6.5: Direkte Bindung zwischen ausführbarem Code und Quellcode

Versionskontrolle, Konfigurationsmanagement und Dokumentation

Analog zu einem Debugging-System, daß sich auf die referentielle Integrität von Bindungen zwischen ausführbarem Code und Quelltext verlassen kann, profitiert auch ein System zur Versionskontrolle von dieser Eigenschaft. Hypercode kann direkte Referenzen auf sog. *version controller* [Morrison et al. 95; Morrison et al. 94] enthalten, welche für die Organisation von Versionen von Objekten verantwortlich sind.

Software-Konfigurationsmanagement kann von der Tatsache profitieren, daß durch direkte Referenzen die Berechnung der aktuellen Konfiguration eines Programms durch Inspektion der transitiven Hülle seiner Hyperprogrammrepräsentation möglich ist. Dies kann z.B. für den Vergleich einer aktuellen Konfiguration mit einer Zielkonfiguration (Design) ausgenutzt werden.

Durch Ersetzung der losen Kopplung zwischen Anwendung und Dokumentation durch direkte Referenzen läßt sich auch eine referentiell integre Bindung zwischen Programmkomponenten und dessen Dokumentation erreichen. Dies sichert natürlich nicht die Korrektheit der Dokumentation, fördert jedoch ihre Konsistenz mit dem Quellcode in ähnlicher Weise wie beim *literate programming* [Knuth 92].

6.4.3 Implementierungsmöglichkeiten in Tycoon

Mit herkömmlicher, batch-orientierter Compiler-Technologie (Scanner, Parser, ...) lassen sich nichtlineare Programmrepräsentationen nicht verarbeiten. Deshalb gibt es zwei prinzipiell unterschiedliche Wege, Hyperprogrammierung zu ermöglichen:

1. *Strukturgesteuertes Editieren:* Hierbei entfallen die ersten beiden Phasen des Übersetzungsvorganges (Scanning, Parsing) zugunsten eines Grafischen Werkzeuges, das es erlaubt, den sonst vom Parser erzeugten Syntaxbaum direkt durch Benutzerinteraktion zu erstellen.

2. *Abbildung nichtlinearer auf lineare Programmrepräsentationen:* Der Benutzer erstellt mit einem Spezialeditor eine nichtlineare Programmrepräsentation, die vor dem Übersetzen in eine lineare Form überführt wird.

zu 1.:

Das Werkzeug zum strukturgesteuerten Editieren unterscheidet sich signifikant von sonstigen sprachsensitiven Editoren, aus deren Klasse beispielhaft der in der VAX/VMS-Umgebung verfügbare LSE (*Language Sensitive Editor*, [LSE 87]) erwähnt sei. Dieses Werkzeug ermöglicht dem Benutzer ein syntaxgesteuertes top-down Vorgehen durch textuelle Expansion von Backus-Nauer-Produktionen. Programme werden jedoch als linearer Text repräsentiert, der durch konventionelle Compiler verarbeitet werden kann. Ein Syntaxbaumeditor kommt dagegen ohne lineare textuelle Repräsentationen aus, so daß existierende Werte direkt durch Benutzerinteraktion mit dem Objektspeicher in die grafische Repräsentation des Syntaxbaums eingefügt werden können.

Die in Abschnitt 5.1.5 beschriebene Implementation der Basiskonstrukte zur Erzeugung von Typrepräsentationen und automorphen Werten beruht auf der Erweiterung des TL-Syntaxbaumes für Werte um eine zusätzliche Variante, die einen beliebigen TL-Laufzeitwert aufnehmen kann, dem OID[6]-Knoten. Vom Back-End werden solche Knoten bei der Übersetzung genauso behandelt wie simple Basiswerte, die im Quelltext als Literal auftauchen. Dadurch ist es möglich, zur Übersetzungszeit Laufzeitwerte durch das Front-End in den Syntaxbaum zu binden. Diese Bindung kann nun vom Benutzer mit Hilfe des Syntaxbaumeditors durchgeführt werden, der das bisherige Front-End ersetzt.

Folgendes Programmierszenario ist denkbar: Durch den Syntaxbaumeditor werden dem Benutzer die verschiedenen syntaktischen Kategorien von TL veranschaulicht und zur Auswahl und weiteren Bearbeitung zur Verfügung gestellt. Textuelle Eingabe ist nur noch zur Benennung von Bindungen an definierender oder angewandter Position sowie zur Darstellung von Literalkonstanten notwendig. Zur Benutzung existierender Werte und Typen dient das in Abschnitt 6.2 dargestellte Objektspeichernavigationswerkzeug, mit dem Bindungen per *drag and drop* an die gewünschten Stellen im Syntaxbaum eingefügt werden können.

zu 2.:

Der zweite Weg wurde bei der Implementation des Hyperprogrammiersystems von Napier88 beschritten [Kirby et al. 93]. Die durch einen Spezialeditor erstellte Hyperprogrammrepräsentation wird dabei in eine lineare Form überführt, bei der alle direkten Referenzen durch eindeutige Bezeichner ersetzt werden, die nicht mit lokalen Bezeichnern kollidieren. Die im Quelltext zur Kompositionszeit „gebundenen“ Werte werden der in Abschnitt 5.4.2 angesprochenen flexiblen Compiler-Schnittstelle in Form einer Symboltabelle (*declaration set*) übergeben.

[6] OID: *Object Identifier*

Einträge in dieser Tabelle bestehen aus dem eindeutigen Bezeichner, einer Typrepräsentation sowie dem Wert selbst.

Die in Abschnitt 5.2.2 vorgestellte Schnittstelle zum TL-Compiler läßt sich in ähnlicher Weise zur „Simulation“ der Übersetzung eines nichtlinearen Programmtextes verwenden, was aus der Analogie zur Schnittstelle des Napier-Compilers (Abschnitt 5.4.2) ersichtlich ist.

Für beide Varianten gilt, daß, um eine direkte Bindung zwischen einem Funktionsabschluß und seiner Hyperprogrammrepräsentation herstellen zu können, eine Referenz zu letzterer im Code der übersetzten Funktion (d.h. im Funktionsabschluß) gespeichert werden muß. Dadurch wird sichergestellt, daß, solange eine Funktion im Objektspeicher existiert auch ihre eindeutige Hyperprogrammrepräsentation zur Verfügung steht. Zur Repräsentation von Hyperprogrammen werden im Napier-System drei verschiedene Formen eingesetzt, die jeweils optimiert sind auf ihre Eignung zur Bearbeitung durch Editier-/Browserwerkzeuge, zur Speicherung in Funktionsabschlüssen (kompaktes Format) und zur Verarbeitung durch den Compiler. Verfolgt man in Tycoon die erste Variante, so bietet sich der Syntaxbaum als Hyperprogrammrepräsentation an, da dieser OID-Knoten, d.h. direkte Referenzen, enthält. Der Syntaxbaum kann durch den Compiler direkt verarbeitet werden, verbraucht aber im Vergleich zu einer mehr textorientierten Darstellung wesentlich mehr Speicherplatz. Für die zweite Variante ist es sicher vorteilhaft, auf die in Napier gemachten Erfahrungen mit unterschiedlichen Repräsentationsformen zurückzugreifen.

6.4.4 Bewertung

Nach [Kirby 92a] unterstützt Hyperprogrammierung drei Ziele:

- Die Reduzierung des Quelltextumfangs dadurch, daß weniger Code (insbesondere für Zugriffsspezifikationen) geschrieben werden muß.
- Größere Zuverlässigkeit des Codes durch einen geringeren Umfang an dynamischer Typprüfung.
- Besseres Verständnis der persistenten Umgebung durch Repräsentationen für Funktionsobjekte mit freien Variablen.

Auf die ersten beiden Punkte und ihre Relevanz im Tycoon-System wurde bereits in Abschnitt 6.4.2 eingegangen. Zu einem besseren Verständnis der persistenten Umgebung reichen Hyperprogrammrepräsentationen für Funktionsobjekte allein jedoch nicht aus. Hyperprogrammierung muß vielmehr darauf abzielen, den gesamten Softwareentwicklungsprozeß in der persistenten Umgebung ablaufen zu lassen, wobei die referentielle Integrität von Bindungen im Objektspeicher in vielfältiger Hinsicht genutzt werden kann. Die mit dem Tycoon-System verfolgte Zielsetzung einer offenen Systemumgebung mit Integrationsmöglichkeit möglichst vieler externer Diensterbringer unter einem gemeinsamen programmiersprachlichen Rahmen steht in einigen Punkten dazu im Widerspruch. Bindungen an externe Objekte (z.B. C-Funktionen) können auch durch Bindung zur Kompositionszeit nicht sicher gemacht werden, da sie nicht der Kontrolle des Objektspeichers unterliegen. Es müssen Werkzeuge zur Programmerstellung innerhalb der persistenten Umgebung bereitgestellt werden, deren Funktionalität prinzipiell nicht durch externe Dienste abgedeckt werden kann. Auch bei anderen Werkzeugen, z.B. zur Versionskontrolle, gilt, daß der kleinste gemeinsame Nenner zur Programmrepräsentation

zwischen verschiedenen Systemen die reine Textform ist, so daß zur Unterstützung von Hyperprogrammierung Neuentwicklungen solcher Werkzeuge innerhalb der persistenten Umgebung notwendig sind. Ansätze dazu sind in [Morrison et al. 95] zu finden.

7. Zusammenfassung und Ausblick

Die Motivation für die Themenstellung dieser Arbeit bestand darin, mit Hilfe der in Kapitel 3 und 5 beschriebenen Dienste die Grundlagen für eine offene, d.h. zur reflektiven Programmierung geeigneten Compilerarchitektur zu schaffen, um darauf aufbauend die Implementierung einer grafischen reflektiven Programmierumgebung zu ermöglichen. Ziele einer solchen Umgebung sind:

1. Erhöhung der Produktivität des Programmierers: Diese hängt hauptsächlich von der Generik des Systems ab, d.h. eine Erweiterung der generischen Dienste bedeutet auch eine Steigerung der Produktivität.

2. Erleichterung des Verständnisses der persistenten Umgebung: Dies kann durch den Einsatz grafischer Werkzeuge zur Visualisierung und Konstruktion von Programmen und Daten erreicht werden, da visuelle Aspekte eine große Rolle beim menschlichen Verstehen spielen.

7.1 Dienste zur Steigerung der Generik

Konkret wird die Generik des Tycoon-Systems durch folgende Maßnahmen gesteigertn:

- Entwicklung eines C++-Gateway-Generators,
- Einführung von Laufzeit-Typrepräsentationen und automorphen Werten in die Sprache TL,
- Repräsentation des TL-Compilers als Bibliotheksdienst zur laufzeitreflektiven Programmierung.

Durch den **C++-Gateway-Generator** wird die generische Integration einer weiteren Klasse externer Diensterbringer in das Tycoon-System ermöglicht. Aufgrund des polymorphen Typsystems von TL konnte eine vollständige Abbildung der für die Schnittstellenbeschreibung wesentlichen Konzepte beider Sprachen aufeinander erreicht werden, trotz der grundsätzlichen Differenzen in den direkt unterstützten Programmierparadigmen (funktional und imperativ in TL vs. objektorientiert und imperativ in C++). Durch den Generatoransatz in Verbindung mit einer Schnittstellenbeschreibungssprache (GDL) wird dem Programmierer ein großer Anteil gleichförmiger, repetitiver und damit fehlerträchtiger Codierungsarbeit abgenommen. Der Zusatzaufwand an Speicher und CPU-Zeit zur Nutzung der durch generierte *Gateways* angesprochenen externen Dienste erscheint akzeptabel, gemessen an den Vorteilen,

die sich durch ihre Benutzung innerhalb einer persistenten, polymorphen, mit vielfältigen generischen Diensten ausgerüsteten Umgebung, wie Tycoon sie bietet, ergeben. Weiterhin wird der *Gateway*-Generator zur Einbindung einer portablen GUI-Klassenbibliothek eingesetzt, wodurch die Grundlage zur Realisierung von grafischen Visualisierungs- und Programmkonstruktionswerkzeugen gelegt wird.

Durch die Einführung von **Laufzeit-Typrepräsentationen und automorphen Werten** in TL wird die Entwicklung einer weiteren Klasse polymorpher Algorithmen ermöglicht. Diese umfaßt Algorithmen, die *typgesteuerte* Berechnungen durchführen. Anwendungen für diese Art des Ad-hoc-Polymorphismus ergeben sich immer dort, wo Werte in unterschiedlichen Kontexten typsicher verwendet werden sollen, d.h. Typen dieser Werte statisch nicht bekannt sind, z.B. in unabhängigen Objektspeichern oder in unabhängigen Evaluationsumgebungen wie bei der laufzeitreflektiven Programmierung. Diese Anwendungen sind deutlich gegenüber solchen abzugrenzen, bei denen über Typen abstrahiert werden kann, was durch parametrischen Polymorphismus und Subtyppolymorphismus unterstützt wird.

Laufzeitreflektive Programmierung wird durch die Bereitstellung des Compilers als Bibliotheksdienst ermöglicht. Dynamische Generierung und Ausführung von Codefragmenten stellt oftmals eine effizientere Alternative gegenüber der Interpretation von Laufzeit-Typinformation dar, insbesondere wenn generierter Code über die Laufzeit einer Anwendung hinaus in der persistenten Umgebung aufbewahrt werden kann. Die Implementation des Standardbeispiels für diese Art der generischen Programmierung, der natürliche Verbund, wird unter Nutzung dynamischer Typinformation und des aufrufbaren Compilers durchgeführt. Weiterhin werden Compilerschnittstellen anderer Programmiersprachen auf ihre Eignung zur reflektiven Programmierung hin untersucht.

7.2 Add-On-Werkzeuge zur Programmentwicklung

Um das Verständnis des Programmierers für die persistente Umgebung zu erhöhen, sind grafische Werkzeuge ein geeignetes Mittel, da hierarchische und netzwerkartige Strukturen nur schwer textuell darzustellen sind. Das in Kapitel 6.2 beschriebene **Werkzeug zur grafischen Objektspeichervisualisierung und -manipulation** stellt die Environment-Struktur des Objektspeichers grafisch dar. Es erlaubt dem Benutzer, TL-Ausdrücke gegen Environments auszuwerten, wobei die daraus resultierenden Seiteneffekte wie z.B. die Etablierung neuer Bindungen im Objektspeicher sofort für den Benutzer sichtbar werden. Weiterhin hat er die Möglichkeit, durch direkte Manipulation Restrukturierungen im Objektspeicher vorzunehmen, indem er Bindungen per Mausaktion zwischen Environments verschiebt. Zur Nutzung reflektiver Möglichkeiten kann das System die Bindungen, aus denen es besteht, dem Benutzer in Environments zur Verfügung stellen. Dies betrifft insbesondere den aufrufbaren Compiler und die Schnittstellen zur Manipulation von Typrepräsentationen und automorphen Werten.

Aufbauend auf den reflektiven Diensten und Spracherweiterungen zur dynamischen Typisierung wird ein Konzept für die **typsichere Implementierung der Modulverwaltungskomponente** des Tycoon-Systems auf Anwendungsebene entwickelt. Mit Hilfe dieses Konzeptes braucht man zur Implementierung dieser Komponente nicht mehr auf compiler-interne Datenstrukturen und Funktionalität zurückgreifen. Typsichere Bindung von Modulen wird dabei durch in den „Objektcode“ integrierte dynamische Typtests erreicht, wobei der Code für die Typtests im gebundenen Programm nicht mehr existiert.

Unter dem Begriff **Hyperprogrammierung** wird eine Technik vorgestellt, die darauf abzielt, die persistente Umgebung möglichst weitgehend in den Softwarekonstruktionsprozeß einzubeziehen. Hauptvorteile dieser Technik sind ihre Eignung zur Visualisierung von Funktionsabschlüssen, die Integration der Konzepte „Quellcode“ und „ausführbarer Code“ zu einer gemeinsamen Sicht sowie die Robustheit von Bindungen zwischen ausführbarem Code und Quellcode in einem Objektspeicher, der die referentielle Integrität aller Bindungen sicherstellt. Tycoon bietet mit Reflektion, Persistenz und GUI-Funktionalität alle Voraussetzungen zur Implementierung eines Hyperprogrammiersystems.

7.3 Ausblick und offene Fragen

In diesem Abschnitt werden abschließend die teilweise in den einzelnen Kapiteln schon angesprochenen offenen Punkte zusammengefaßt und ein Ausblick auf zukünftige Aktivitäten zur Realisierung einer reflektiven grafischen Programmierumgebung für Tycoon gegeben.

Probleme bei der **Integration von C++-Diensten** ergeben sich durch die unterschiedliche Lebensdauer von Tycoon-Bindungen und C++-Objekten, die in zweifacher Hinsicht zu einer Invalidierung von Bindungen im Objektspeicher führen kann:

1. Bei einem Wiederanlauf des Tycoon-Systems existieren evtl. Bindungen an nicht mehr existierende C++-Objekte.

2. Durch Aufruf seines Destruktors kann ein C++-Objekt jederzeit zerstört werden. Die zugehörige TL-Bindung darf dann nicht mehr verwendet werden.

Eine Lösung des ersten Problems ist die explizite Registrierung solcher Objekte als flüchtige Ressourcen mit dem Ziel, diesen durch Neuerzeugung bei Wiederanlauf des Systems eine Art Pseudo-Persistenz zu verschaffen [Mathiske et al. 95a]. Durch die Einführung schwacher Referenzen (*Weak References* [Horning et al. 93]) in Tycoon kann der explizite Aufruf von Destruktoren vermieden und der automatischen Freispeicherverwaltung überlassen werden.

Zur Verwirklichung der in Abbildung 2.2 dargestellten **reflektiven Systemarchitektur** bedarf es noch weiterer konkreter Integrationsarbeiten:

- Die Vereinheitlichung der statischen und dynamischen Typrepräsentation zur Vermeidung laufzeitintensiver Typkonvertierungen (siehe Abschnitt 5.2.2) sowie die

- Erweiterung der TL-Syntax und -Semantik um wertparametrisierte Typoperatoren (siehe Abschnitt 6.3).

Bei der Reimplementation der Modulverwaltungskomponente auf Anwendungsebene ist zu entscheiden, ob übersetzte Module und Modulschnittstellen weiterhin zusätzlich im Dateisystem gehalten werden sollen, um die arbeitsteilige Anwendungsentwicklung mit mehreren unabhängigen Objektspeichern zu unterstützen. Dies würde wegen der Nichtlinearität von Typrepräsentationen zu einem hohen Speicherplatzverbrauch für Typrepräsentationen von Modulschnittstellen im Dateisystem führen, der proportional zur Anzahl (transitiv) importierter Schnittstellen wächst. Um dies zu verhindern, wäre ein Mechanismus erforderlich, der es erlaubt, geschlossene nichtlineare Typrepräsentationen quasi „aufzubrechen“ (zur separaten Speicherung von Schnittstellen in Dateien) und diese auch wieder zusammenzusetzen (zum

Aufbau einer Übersetzungsumgebung für Module). Es ist zu untersuchen, ob das in [Mathiske et al. 95b] beschriebene Verfahren zum dynamischen Binden an ubiquitäre Ressourcen für diese Zwecke einsetzbar ist.

Bei der Nutzung **grafischer Möglichkeiten** zur Programmvisualisierung und -konstruktion steht zunächst die Weiterentwicklung der Objektspeichervisualisierung im Vordergrund. Bevor weitere Aktivitäten zum Forschungsthema Hyperprogrammierung im Tycoon-Umfeld unternommen werden, ist eine grundsätzliche Entscheidung darüber notwendig, welche Bedeutung der persistenten Umgebung für den Softwareentwicklungsprozeß zugemessen wird und ob der Entwicklungsaufwand für spezialisierte Werkzeuge zur Unterstützung dieses Programmierstils im Verhältnis zu den dadurch realisierbaren Vorteilen angemessen ist. Ferner ist diese Methodik auf ihre Eignung zum „Programmieren im Großen“ hin zu untersuchen und mit dem Modulkonzept zu vergleichen.

Eine besondere Bedeutung in einer **voll integrierten persistenten Umgebung** kommt der Objektspeicherimplementation zu, die ein hohes Maß an Sicherheit und Funktionalität (insbesondere für den Mehrbenutzerbetrieb) gewährleisten muß, um das Dateisystem des Betriebssystems vollwertig ersetzen zu können. Hauptvorteil einer solchen Umgebung wäre dann, daß sie die Funktionalität heutiger Datenbanken, Betriebssysteme, Programmiersprachen und deren Benutzerschnittstellen unter einheitlichen Bindungs-, Benennungs- und Typisierungskonzepten subsumieren würde. Angesichts der zunehmenden Heterogenität von Systemen wäre dies ein entscheidender Beitrag der Informatik zur konzeptionellen Vereinfachung und Orthogonalisierung der Benutzung von Rechensystemen.

A. Sprachbeschreibung GDL

Syntaktische Konventionen

Folgende EBNF-ähnliche Notation wird zur Definition syntaktischer Elemente verwendet, wobei A und B syntaktische Ausdrücke repräsentieren und Terminalsymbole durch Fettdruck dargestellt werden.

Symbol	Bedeutung
Id ::== A	Nicht-Terminal Id wird durch A definiert
'x'	das Zeichen x
identifier	ein gültiger TL-Bezeichner
"string"	String-Literal
A B	A gefolgt von B
$A \mid B$	A oder B
$[A]$	null- oder einmaliges Vorkommen von A
$\{A\}$	null- oder mehrmaliges Vorkommen von A

Reservierte Schlüsselwörter

Folgende Bezeichner sind reservierte Schlüsselworte und dürfen nicht als Bezeichner in GDL-Quelltexten verwendet werden:

as class const constructor define end enum export from include import gateway static typedef use using virtual

Symbole

Der Aufbau von Symbolen entspricht dem von TL [Matthes, Schmidt 92], wobei (in Anlehnung an C) Kommentare auch alternativ durch /* */ geklammert sein können. '#' and ';' werden als Formatierungszeichen behandelt, also überlesen.

Produktionen

Übersetzungseinheit

description ::= *header*
[**include** *"string"*{*"string"*}]
export {*typeDecl*}
end

header ::= **qateway** "C++" *module-name* : *interface-name*
[**import** *gateway-name* {*gateway-name*}]

Typen und Konstanten

typeDecl ::= *typedef* | *define* | *use* | *enum* | *class*

typedef ::= **typedef** *identifier [ref] declarator*

define ::= **define** *identifier int*

use ::= **use** *declarator* **from** *identifier* **as** *identifier* **using** *identifier*

enum ::= **enum** *declarator* '{' *enumList* '}'

enumList ::= *enumerator* { ',' *enumerator*}

enumerator ::= *declarator* ['=' *integer*]

ref ::= * | &

declarator ::= *identifier* ['=¿' TL-*identifier*]
["documentation-string"] (* Umbenennung und Dokumentation *)

Klassen

class ::= *classHead* ['{' [*headerList*] '}']

classHead ::= **class** *declarator* [*baseSpec*]

baseSpec ::= ':' *baseList*

baseList ::= *identifier* { ',' *identifier*}

Methoden

headerList ::= *methodHeader* {*methodHeader*}

methodHeader ::= *normalHeader* | *constructorHeader* | *virtualHeader*

normalHeader ::= [**static**] *identifier* [*ref*] *nameAndSigs*

constrHeader ::= **constructor** *nameAndSigs*

virtualHeader ::= **virtual** *identifier* [*ref*] *nameAndSigs*

nameAndSigs ::= *declarator* '(' [*sigList*] ')' [**const**]

sigList ::= *signature* { ',' *signature*}

signature ::= [**const**] *identifier* [*ref*] *identifier*

B. Schnittstellen für dynamische Typen

B.1 Schnittstelle 'TypeRep'

```
interface TypeRep
import list time :Iter :Source
export
  Let T = typeRep_T
  (* T is a type representation for an arbitrary parameterized or closed type. *)
  Let Ide = Tuple name :String  pos :Source.Position end
  Let Signature = Tuple
    ide :Ide
    type :T
    case typeCase, typeEqualCase with (* T<:A rsp. T = A *)
      typeID :time.T
    case typeBoundCase with
      typeID :time.T
      bound :T
    case valueCase, locationCase
  end
  Let Case = Tuple
    label :String
    signatures :list.T(Signature) (* additional fields for this variant *)
  end
  Let Expansion = Tuple
    case okCase, nokCase
    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 (* type operator application *)
     oper :T
     arguments :list.T(Signature)
  case recCase with
    bindingIndex :Int  (* No. of binding in typeBindings *)
    typeBindings :list.T(Signature)
end

isSubType(small, big :T) :Bool
(* Return true if the type representations are in the subtype relationship
   small<:big. *)
inspect(type :T) :Expansion
(* Return a more detailed description of the type 'type'. *)
exposed(type :T) :T
(* Strip definition(s) of 'type'. *)
signatures(t :T) :Iter.T(Signature)
(* If t is a tuple, record, exception, oper or function representation,
   return an iteration of signatures, else raise error *)
fmt(type :T) :String
(* Return a string containing a TL type expression corresponding to type *)
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
(* Constructors for type representations *)
end
```

B.2 Schnittstelle 'Dynamic'

```
interface Dynamic
import list typeRep :TypeRep :Iter reader writer
export
  error :Exception
  Let T = dynamic_T
  (* A dynamic value :T is a  tuple val :T  type :Type end
     where type is the type representation of val. *)
  Let Binding = Tuple
    name :String
    case typeCase with type :typeRep.T
    case valueCase with value :T
    case locationCase with
      value() :T
      setValue(:T) :Ok
  end
  Let Expansion = Tuple
    case okCase
    case boolCase with val :Bool
    case charCase with val :Char
    case intCase with val :Int
    case realCase with val :Real
    case stringCase with val :String
    case adtCase
    case funCase with
      signatures :list.T(TypeRep.Signature)
      range :typeRep.T
    case tupleCase with
      tag: String
      fields :list.T(Binding)
    case recordCase with fields :list.T(Binding)
    case exceptionCase with
      val :String
      signatures :list.T(TypeRep.Signature)
    case arrayCase with
      type :typeRep.T   (* type is the element type *)
      size :Int
    case appCase with
      oper :typeRep.T
      arguments :list.T(TypeRep.Signature)
      value :T
  end
  typeOf(value :T) :typeRep.T
  (* Return the type attribute of value. *)
  valueOf(value :T) :Ok
  (* Return the value attribute of value with least possible static type information *)
```

```
  inspect(value :T) :Expansion
  (* Return a more detailed description of the value 'value'. *)
  signatures(value :T) :Iter.T(TypeRep.Signature)
  (* If value is a value of a function or exception, return an iteration of
     signatures, else raise error *)
  bindings(value :T) :Iter.T(Binding)
  (* If value is a dynamic of a tuple or record, return an iteration of bindings,
  else raise error *)
  functions(value :T) :Iter.T(T)
  (* Return an iteration of all functional components of a value. Especially
     useful when value is a dynamic of a linked module. *)
  getIndexed(value :T  i :Int) :T
  (* Return the i'th element of value if it is a dynamic of an array else
     raise error *)
  setIndexed(value :T  i :Int newValue :T) :Ok
  (* Set the i'th element of value to newValue if it is a dynamic of an array
     else raise error *)
  intern(r :reader.T) :T
  (* Obtain a dynamic from a reader. *)
  extern(w :writer.T  value :T) :Ok
  (* Write a dynamic to a writer. *)
  newDefault(type :typeRep.T) :T
  (* Create a default value with type 'type'. *)
  newArray(A <:Ok  arr :Array(A)  type :typeRep.T) :T
  newTuple(fields :list.T(Binding)) :T
  newTupleCase(tag :String  type :typeRep.T fields :list.T(Binding)) :T
  (* Raise error if tag is not a tag of type or if fields do not match type. *)
  newRecord(fields :list.T(Binding)) :T
  newException(value :String  signatures :list.T(TypeRep.Signature)) :T
end
```

C. Schnittstellen des aufrufbaren Compilers

C.1 Schnittstelle 'Environment'

```
interface Environment
import :Iter :Dynamic
export
 lookupError :Exception with name :String end
 T <:Ok
 (* A collection of Bindings *)
 new() :T
 (* Return an Environment containing the initial bindings *)
 clear(:T) :Ok
 (* Clear all Bindings except the initial. *)
 size(:T) :Int
 (* Return the size of an Environment. *)
 getLast(:T) :Dynamic.Binding
 (* Return last binding of an Environment *)
 concat(e1, e2 :T) :T
(* Return concatenated environment. Bindings in e1 will possibly hide bindings in e2 *)
 lookup(:T name :String) :Dynamic.Binding
 (* Return last binding named 'name'. Raise lookupError if not found *)

 dropBinding(:T name :String) :Ok
 (* Removes binding for 'name' from environment. *)
 dropLastBinding(:T) :Ok
 (* Remove last binding of an environment. *)
 addBinding(:T binding :Dynamic.Binding) :Ok
 (* Extend environment by adding bnd. *)
 addValueBinding(:T :String :dynamic_T) :Ok
 (* Short cut for addBinding(...). *)
 addTypeBinding(:T :String :typeRep_T) :Ok
 (* Short cut for addBinding(...). *)
 addLocationBinding(:T :String :dynamic_T) :Ok
 (* Short cut for addBinding(...). *)
 changeValue(:T :String :dynamic_T) :Ok
```

```
  (* Change value of a Binding *)
  ...
  symbols(:T) :Iter.T(String)
  (* Return iteration over symbols bound in environment *)
  create(from :Iter.T(Dynamic.Binding)) :T
  (* Create an environment from an iteration of bindings *)
  elements(:T) :Iter.T(Dynamic.Binding)
  (* Return iteration over bindings in environment *)
end
```

C.2 Schnittstelle 'Compiler'

```
interface Compiler
import environment source errorLog :Scanner :Symbol :TLValue :TML tm
export
  error :Exception with log :errorLog.T end

  parse(:source.T) :TLValue.Bindings
  (* Return abstract Syntax for source. *)
  check(:TLValue.Bindings :environment.T)()() :Dynamic.Binding
  (* Checks bindings and returns a function which translates bindings
     and returns a function which executes them.
     Updates environment on execution. *)
  compileSyntaxTree(:environment.T :TLValue.Bindings)() :Dynamic.Binding
  (* Compiles bindings and returns a function which executes them.
     Updates environment on execution. *)
  compileSource(:environment.T :source.T)() :Dynamic.Binding
  (* Compiles src and returns a function which executes them.
     Updates environment on execution. *)
  compileString(:environment.T :String)() :Dynamic.Binding
  (* Compiles src and returns a function which executes them.
     Updates environment on execution. *)
  eval(:environment.T :String) :Dynamic.Binding
  (* Executes a string, updates environment and returns the resulting Binding. *)

  (* Retrieval of intermediate results. *)
  getScan() :Scanner.T(Symbol.T)
  getParseTree() :TLValue.Bindings
  getAttributedParseTree() :TLValue.Bindings
  getTMLTree() :TML.T
  getTVMCode() :tm.OID

  (* compiler switches: *)
  optimize(:Bool) :Ok
  ...
  getOptimize() :Bool
  ...

  (* switches for system debugging (output to stdout): *)
  printTVM(:Bool) :Ok
  traceSubType(:Bool) :Ok
  ...
  getPrintTVM() :Bool
  getTraceSubType() :Bool
  ...
end
```

D. Beispiel für eine typgesteuerte Funktion: natürlicher Verbund

Die Funktion *generateJoin* erfüllt folgende Teilaufgaben:

1. Generierung einer Typrepräsentation der Ausgaberelation, ausgehend von den Typrepräsentationen der Eingaberelation,

2. Generierung einer Funktion (*defineMatch*), die feststellt, ob zwei Join-Felder gleich sind. Der Test auf paarweiser Gleichheit der Join-Attribute wird dabei auf den Test auf (tiefe) Gleichheit zweier automorpher Werte zurückgeführt. Dadurch arbeitet die generierte Funktion auch auf nicht flach strukturierten Tupeln. Die Funktion *andCompose* erzeugt aus einer Menge von Strings einen String, dessen Worte durch "andif" verknüpft sind.

3. Generierung einer Funktion (*defineConcat*), die aus passenden Attributen der Eingabetupel ein Ausgabetupel konstruiert. Die dazu benutzte Funktion höherer Ordnung *reduce* hat folgende Signatur:

   ```
   Fun(A, B, C <:Ok
         i :Iter.T(A)
         apply(:A) :B
         accumulate(:B :C) :C
         base :C) :C
   ```

   Sie wendet *apply* auf jedes Element von *i* an und akkumuliert das Resultat mittels *accumulate*, beginnend mit *base*.

4. Generierung einer rekursiven Funktion, die die Join-Operation ausführt.

   ```
   let signatureEqual(s1, s2 :typeRep.Signature) :Bool =
     string.equal(s1.ide.name s2.ide.name)
     andif typeRep.equal(s1.type s2.type)

   let getStructureFields(t :typeRep.T) :relation.T(typeRep.Signature) =
     set.create(
       list.elements(
         case typeRep.inspect(t)
         when recordCase with t1 then t1.signatures
         else raise exception "Not a record type"
   ```

```
      end)
    signatureEqual)

let generateJoin(type1, type2 :typeRep.T) :String =
  begin
    let type1Fields = getStructureFields(type1)
    let type2Fields = getStructureFields(type2)
    let resultFields = set.union(type1Fields type2Fields)
    let resultType :typeRep.T =
      typeRep.newRecord(list.create(set.elements(resultFields)))
    let defineMatch() :String =
      begin
        let overlap = set.intersection(type1Fields type2Fields)
        let testSet = set.new(:String string.equal)
        iter.forEach( (* fill testSet with compare expressions *)
          set.elements(overlap)
          fun(sig :typeRep.Signature) :Ok
            begin
              let expr =
                "dynamic.equal(dynamic_new(arg1." <> [1] sig.ide.name ")" <>
                        "dynamic_new(arg2." <> sig.ide.name <> "))"
              try set.insert(testSet expr) else ok end
            end)
        "fun(arg1 :Rec1 arg2 :Rec2) :Bool " <> andCompose(testSet)
      end

    let defineConcat() :String =
      begin
        let fields  =
          reduce(
            set.elements(resultFields)
            fun(sig :typeRep.Signature) :String
              begin
                let takeFrom =
                  if set.member(sig type1Fields) then "arg1"
                  else "arg2"
                  end
                "let " <> sig.ide.name <> " = " <> takeFrom <> "." <>
                 sig.ide.name <> " "
              end
            fun(binding :String fields :String)
              string.cat(fields binding)
            "")
        "fun(arg1 :Rec1 arg2 :Rec2) :Result record " <> fields <> "end"
      end
```

[1] Operator zur Verkettung von Zeichenketten

```
let defineOneJoin() :String =
  "    let rec onejoin(element :Rec1 rel :Rel2 :ResultRel = \n" <>
  "      if relation.empty(rel) then \n" <>
  "        relation.new(:Result fun(x,y :Result) x == y) \n" <>
  "      else \n" <>
  "        let choose = relation.choose(rel) \n" <>
  "        let rest = onejoin(element relation.rest(rel)) \n" <>
  "        let match = " <> defineMatch() <> "\n" <>
  "        let concat = " <> defineConcat() <> "\n" <>
  "        if match(element choose) then \n" <>
  "          let newRecord = concat(element choose) \n" <>
  "          relation.insert(rest newRecord) \n" <>
  "        end \n" <>
  "        rest \n" <>
  "      end \n"

  "Let Rec1 = " <>  typeRep.fmt(type1) <> "\n" <>
  "Let Rec2 = " <>  typeRep.fmt(type2) <> "\n" <>
  "Let Result = " <>  typeRep.fmt(resultType) <> "\n" <>
  "\n" <>
  "Let Rel1 = relation.T(Rec1)\n" <>
  "Let Rel2 = relation.T(Rec2)\n" <>
  "Let ResultRel = relation.T(Result)\n" <>
  "let rec join(rel1 :Rel1 rel2 :Rel2) :ResultRel = \n" <>
  "  if relation.empty(rel1) then \n" <>
  "    relation.new(:Result fun(x,y :Result) x == y) \n" <>
  "  else \n" <>
  "    " <> defineOneJoin() <> "\n" <>
  "    let joinOne = onejoin(relation.choose(rel1) rel2) \n" <>
  "    let joinOthers = join(relation.rest(rel1) rel2) \n" <>
  "    relation.union(joinOne joinOthers) \n" <>
  "  end \n"
end
```

Abbildungsverzeichnis

Literaturverzeichnis

Abadi et al. 89: M. Abadi, L. Cardelli, B. C. Pierce und G.D. Plotkin. *Dynamic Typing in a Statically Typed Language.* Technical Report 47, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, Juni 1989.

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.

Albano et al. 85: A. Albano, L. Cardelli und Orsini R. *Galileo: A Strongly-Typed, Interactive Conceptual Language.* ACM Transactions on Database Systems, Jg. 10, 1985, Nr. 2, S. 230–260.

Albano et al. 91: A. Albano, G. Ghelli und R. Orsini. *A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language.* In: *Proceedings of the Seventeenth International Conference on Very Large Databases*, 1991, S. 565–575.

Appel et al. 93: Andrew Appel, Lal George, David MacQueen und John Reppy. *Standard ML of New Jersey.* AT&T Bell Laboratories, New Jersey, 1993.

Busch et al. 93: A. Busch, T Kuehnel und A Jahnke. *Starview 2.0.* STAR DIVISION, Hamburg, 1993.

Bussche et al. 92: J. Van den Bussche, D. Van Gucht und G. Vossen. *Reflective Programming in the Relational Algebra.* Technical Report 9210, Justus-Liebig-Universität Giessen, 1992.

Cardelli et al. 94: L. Cardelli, F. Matthes und M. Abadi. *Extensible Syntax with Lexical Scoping.* Technical Report 121, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, Februar 1994.

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 86: L. Cardelli. *Amber.* In: *Combinators and Functional Programming Languages*, Lecture Notes in Computer Science, Bd. 242. Springer-Verlag, 1986.

Cardelli 89: L. Cardelli. *Typeful Programming.* Technical Report 45, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, Mai 1989.

Connor et al. 94: R. Connor, Q. Cutts, G.N.C. Kirby, V.S. Moore und R. Morrison. *Unifying Interaction with Persistent Data and Program.* In: *2nd International Workshop on User Interfaces to Databases*, Ambleside, Cumbria, 1994, S. 185–200.

Cooper, Kirby 94: Richard Cooper und G.N.C. Kirby. *Type-Safe Linguistic Run-time Reflection: A Practical Perspective.* FIDE Technical Report Series FIDE/94/108, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1994.

Cutts 92: Q.I. Cutts. *Delivering the Benefits of Persistence to System Construction and Execution.* Dissertation, University of St Andrews, 1992.

da Silva 95: Miguel Mira da Silva. *Automating Type-safe RPC.* FIDE Technical Report Series FIDE/95/114, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1995.

DCE 93: OSF DCE Application Development Guide. Prentice Hall, 1993.

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: *Proceedings of the Second International Workshop on Database Programming Languages, Portland, Oregon*, Juni 1989.

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.* Springer-Verlag, 1990, S. 56–69.

Dearle et al. 92: A. Dearle, Q.I. Cutts und R.C.H. Connor. *An Application Architecture using Type-Safe Incremental Linking.* FIDE Technical Report Series FIDE/92/56, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1992.

Dearle 88: A. Dearle. *On the Construction of Persistent Programming Environments.* Dissertation, University of St Andrews, Maerz 1988.

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.

Ellis, Stroustrup 90: M.A. Ellis und B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 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.* Technical Report CS/92/52, University of St Andrews, 1992.

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

Gawecki, Matthes 95: A. Gawecki und F. Matthes. *TooL: A Persistent Language Integrating Subtyping, Matching and Type Quantification.* (Zur Veröffentlichung freigegeben.), Mai 1995.

Glasgow 88: PS-algol Reference Manual, 4th edition, nr. PPRR-12-88. Persistent Programming Research Group, 1988.

Goldberg, Robson 89: Adele Goldberg und David Robson. *Smalltalk-80: The Language.* Addison-Wesley Publishing Company, 1989.

Gotthard, Lockemann 85: W. Gotthard und C. Lockemann. *Datenbanksysteme für Software-Produktionsumgebungen – Anforderungen und Konzepte.* In: *Methoden und Werkzeuge zur Entwicklung von Programmsystemen*, 1985, S. 185–210.

Härder 87: Theo Härder. *Behandlung von ad hoc-Anfragen.* In: P.C. Lockemann und J.W. Schmidt (Hrsg.). *Datenbankhandbuch.* Springer, Berlin, 1987, Ch. 3.6.8.2, S. 330–335.

Heuer 92: Andreas Heuer. *Objektorientierte Datenbanken.* Addison–Wesley (Deutschland) GmbH, Bonn, 1992.

Horning et al. 93: J. Horning, B. Kalsow, P. McJones und G. Nelson. *Some useful Modula-3 Interfaces.* Technical Report 113, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, 1993.

Ingres 90: Ingres Corp. *Application Editor User's Guide for INGRES/Windows 4GL for the UNIX and VMS Operation Systems*, Alameda, 1990. Ingres Corp.

Jansen et al. 94: B. Jansen, D. Severson und M. Spreitzer. „ILU 1.6.4. Reference Manual". ftp, May 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. 93: G.N.C. Kirby, Q.I. Cutts, R.C.H. Connor und R. Morrison. *The Implementation of a Hyper-Programming System.* Technical Report CS/93/5, University of St Andrews, 1993.

Kirby et al. 94a: Kirby, Brown, Connnor, Cutts, Dearle, Moore, Morrison und Munro. *The Napier88 Standard Library Reference Manual (Version 2.2).* FIDE Technical Report Series FIDE/94/105, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1994.

Kirby et al. 94b: G.N.C. Kirby, R.C.H. Connor und R. Morrison. *START: A Linguistic Reflection Tool Using Hyper-Program Technology.* FIDE Technical Report Series FIDE/94/96, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1994.

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

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

Kirby 92b: G.N.C. Kirby. *Persistent Programming with Strongly Typed Linguistic Reflection.* FIDE Technical Report Series FIDE/92/40, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1992.

Knuth 92: Donald E. Knuth. *Literate Programming.* Center for the Study of Language and Information, Leland Stanford Junior University, 1992.

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.

LSE 87: Guide to VAX Language-Sensitive Editor, München, 1987.

Mandelkern 93: D. Mandelkern. *Graphical User Interfaces: The Next Generation.* Communications of the ACM, Jg. 36, 1993, Nr. 4, S. 36–39.

Martin 91: Bruce Martin. *The Separation of Interface and Implementation in C++.* In: *Usenix Association C++ Conference*, 1991, S. 51–63.

Maslo, Dittrich 93: P. Maslo und St. Dittrich. *Das große Buch zu VisualBASIC 3.0 für Windows.* Data Becker, Düsseldorf, 1993.

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, Deutschland, Dezember 1993.

Mathiske et al. 95a: B. Mathiske, F. Matthes und J.W. Schmidt. *On Migrating Threads.* (Accepted for publication.), Juni 1995.

Mathiske et al. 95b: B. Mathiske, F. Matthes und J.W. Schmidt. *Scaling Database Languages to Higher-Order Distributed Programming.* (Submitted for publication.), Maerz 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, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1994.

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, Deutschland, Oktober 1992.

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 93: F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, 1993.

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, P.W. Abrahams, D.J. Edwards, T.P. Hart und M.I. Levin. *The Lisp Programmars' Manual.* MIT Press, Cambridge, Massachusetts, 1962.

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

Morrison et al. 94: R. Morrison, R.C.H. Connor, Q.I. Cutt und G.N.C. Kirby. *Persistent Possibilities for Software Environments.* FIDE Technical Report Series FIDE/94/92, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1994.

Morrison et al. 95: R. Morrison, R.C.H. Connor, Q.I. Cutts, V.S. Dunstan und G. Kirby. *Exploiting Persistent Linkage in Software Engineering Environments.* FIDE Technical Report Series FIDE/95/115, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1995.

Morrison 94: R. et al Morrison. *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-Jones et al. 95: K. Müller-Jones, M. Merz und W. Lamersdorf. *The TRADEr: Integrating trading into DCE.* In: *Proceedings of the Third International Conference on Open Distributed Processing (ICODP '95), Brisbane, Australia*, Februar 1995.

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

OMG 91: The Common Object Request Broker: Architecture and Specification. Object Management Group, Dezember 1991. OMG Document Number 91.12.1, Revision 1.1.

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.

Schröder, Matthes 92: G. Schröder und F. Matthes. *Using the Tycoon Compiler Toolkit.* DBIS Tycoon Report 061-92, Fachbereich Informatik, Universität Hamburg, Deutschland, Mai 1992.

Sheard 90: T. Sheard. *A user's guide to TRPL: A compile-time reflective programming language.* University of Massachusetts, 1990.

Sjøberg et al. 94: D.I.K. Sjøberg, Q.I. Cutts, R. Welland und M.P. Atkinson. *Analysing Persistent Language Applications.* Technical Report CS/94/109, University of St Andrews, Oslo and Glasgow, 1994.

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.

Sun 90: Network Programming Guide, Mountain View, March 1990. Sun Microsystems Inc.

Sun 92: Sun Microsystems, Inc. *OpenWindows Developers's Guide 2.0 User's Guide*, Mountain View, 1992.

Weis 90: Pierre Weis. *Dynamics in ML.* Technical Report 121, INRIA, Domaine de Voluceau, Rocquencourt 78153 Le Chesnay Cedex, France, 1990.