Kodegenerierung für Programmiersprachen mit Persistenz, Polymorphie und Funktionen höherer Ordnung

Diplomarbeit

von

Bernd J. W. Mathiske

Betreuer:

Prof. Dr. Joachim W. Schmidt

Dr. Martin Lehmann

Fachbereich Informatik

Universität Hamburg

Dezember 1992

Zusammenfassung

Persistenz, Polymorphie und Funktionen höherer Ordnung sind grundlegende Konzepte leistungsfähiger moderner Mehrzweckprogrammiersprachen, die sich besonders zur Entwicklung datenintensiver Systeme eignen. Hinzu kommen die Systemanforderungen Offenheit und Portabilität.

In der vorliegenden Arbeit werden adäquate Methoden zur Kodegenerierung für Sprachen der genannten Kategorie bereitgestellt. Sie werden im Rahmen einer vollständigen exemplarischen Implementation beschrieben.

Als besonderes Ergebnis ist die Unterstützung rekursiver Wertbindungen, die sich nicht nur auf Funktionen, sondern auch auf Aggregate erstrecken, hervorzuheben. Letztere dürfen mit gewissen Einschränkungen sogar komplexe Wertausdrücke enthalten.

Die implementierte Sprache TL (*Tycoon Language*) ist ein erweiterter Lambda-Kalkül zweiter Ordnung. Insbesondere weist sie imperative Konstrukte und Subtypisierungsregeln auf. Als Zwischensprache wird TML (*Tycoon Machine Language*) verwendet. Sie basiert auf einem *untypiserten* Lamda-Kalkül mit zusätzlichen imperativen Konstrukten und Funktionsabschlüssen zur Repräsentation statischer Sichtbarkeitsbereiche. Die Maschinenkodegenerierung erfolgt indirekt, wobei C (wahlweise ANSI-C oder K&R-C) als portable Zielsprache eingesetzt wird. Die Integration weiterer Zielsprachen ist aufgrund einer dedizierten Modularisierung nur wenig aufwendig.

Zur Laufzeit können dynamische Kontextwechsel zwischen interpretiertem TML -Kode, Maschinenkode und externen Routinen erfolgen.

Meinen Eltern,

Eveline Margarete Mathiske, geborene Heim und
Wolfgang Artur Paul Mathiske

in Liebe gewidmet.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Bei der Konzeption von Programmiersprachen treten oft unterschiedliche Schwerpunkte auf, die nur bedingt in Einklang zu bringen sind. Zum einen entstehen Sprachen, die für einen bestimmten Anwendungsbereich prädestiniert sind (z.B. APL, Prolog, SQL), zum anderen wird versucht, möglichst universell einsetzbare sprachliche Werkzeuge, sogenannte Mehrzweckprogrammiersprachen (*general purpose programming languages*) zu schaffen.

Ein wesentliches Merkmal von Mehrzweckprogrammiersprachen ist die Fähigkeit zur Selbstimplementation. Diese Homogenisierung führt zu wesentlichen Vereinfachungen der Wartung und Weiterentwicklung. Zum Zwecke der Erstimplementation eines algorithmischen Kerns ist die Verwendung einer fremden Implementationssprache notwendig. Ihr weiterer Einsatz steht jedoch im Widerspruch zur beabsichtigten Leistungsfähigkeit der implementierten Sprache.

Da andererseits keine Programmiersprache in *jeder* Anwendungssituation allen anderen vorzuziehen ist, werden in großen Systemen häufig mehrere unterschiedliche Programmiersprachen eingesetzt, die sich problemspezifisch ergänzen. Ein anderer wichtiger Grund für die Entstehung heterogener Systeme ist die Wirtschaftlichkeit der Wiederverwendung von Teilsystemen.

Durch die spezifischen Probleme der Heterogenität werden wesentliche Kriterien der Softwarequalität wie Modifizierbarkeit, Wartbarkeit und verschiedene Aspekte der Korrektheit und der Sicherheit stark beeinträchtigt. Diese Erkenntnis motiviert Programmiersprachen, die zumindest befriedigende Mittel zur Bewältigung einiger wichtiger Aufgabenbereiche integrieren. Obiger Ansatz ist insbesondere für datenintensive Anwendungen von Bedeutung, weil diese besonders hohen Korrektheits- und Sicherheitsanforderungen unterliegen.

Im folgenden werden Programmiersprachen betrachtet, die sowohl für die konventionellen Aufgaben von Mehrzweckprogrammiersprachen als auch für die der Datenbankprogrammierung konzipiert sind. Weil die Datenbanksystemprogrammierung selbst eine datenintensive Aufgabe darstellt [Matthes, Schmidt 91b], gilt die Forderung der Selbstimplementation auch für diese Sprachen.

In den Programmiersprachen Pascal/R [Schmidt 77], und DBPL [Matthes, Schmidt 89] werden Relationstypen und mengenorientierte Operationen auf typisierten Relationenvariablen in eine Pascal-ähnliche Programmiersprache integriert. Darüber hinaus existieren Mechanismen zur Definition von Transaktionen und zur Deklaration persistenter Daten.

Während in diesen Sprachen Persistenz nur partiell und als explizit zu deklarierendes Attribut auftritt, ist sie in den sogenannten „persistenten Sprachen“ (PS algol [Atkinson et al. 81],

Napier88 [Dearle et al. 89], Amber [Cardelli 86a], P-Quest [Müller 91; Matthes 91]) ein orthogonales Konzept. *Alle* semantisch relevanten Objekte (Aggregate, abstrakte Datentypen, Funktionen, Module, etc.) werden in einem globalen, programmübergreifenden Adreßraum alloziert und manipuliert, den eine persistente Halde (*persistent heap*) realisiert. Sie sind *ohne* diesbezügliche Deklaration langlebig. Von einem ausgezeichneten Wurzelobjekt (*persistent root*) ausgehend, ist jedes durch statische Sichtbarkeitsregeln oder dynamische Bindungen transitiv erreichbare Objekt persistent.

Das Konzept der orthogonalen Persistenz führt zu erheblichen Vereinfachungen der Systemarchitektur. Die Einführung einer abstrakten Objektspeicherschnittstelle, über die *alle* Speicherzugriffe während der Programmausführung abgewickelt werden [Brown et al. 88; Moss 89] gestattet die völlige Gleichbehandlung von persistenten und nicht-persistenten sowie prozeßlokalen und systemweiten Programmobjekten [Matthes 92]. Durch die weitestgehende Entkopplung der Aufgaben der Datenstrukturierung- und der Datenspeicherung beliebig komplex strukturierter Werte entfällt in vielen Anwendungen der Bedarf an externen Repräsentationen zur langlebigen Speicherung [Atkinson, Morrison 85; Morrison et al. 87; Dearle, Brown 87; Müller 91].

Gegenüber den integrierten Datenbankprogrammiersprachen haben die zur Zeit existierenden persistenten Sprachen einen schwerwiegenden Nachteil: sie weisen keine Zwischenrepräsentationen auf, die statische oder dynamische Programmoptimierungen gestatten (vgl. Anfrageoptimierungen im DBPL-System [Schmidt, Matthes 92]). Für Datenbankprogrammiersprachen sind multiple Programmrepräsentationen auf verschiedenen semantischen Abstraktionsebenen jedoch unerläßlich. Sie dienen außerdem verschiedenen Portierungs- und Transformationsaufgaben sowie dem Versenden kompilierter partiell gebundener Programme in heterogenen Netzwerken [Matthes 92].

Besonders hohen Anforderungen unterliegt die Entwicklung sogenannter Informationssysteme. Dabei handelt es sich um Systeme, die einen flexiblen Zugriff auf große, komplex verknüpfte Datenmengen erlauben. Aufgrund ihrer Komplexität ergibt sich ein großer Bedarf an externen Diensterbringern und eine gesteigerte Bedeutung der Wiederverwendbarkeit von Teilsystemen. Sowohl diesen beiden Aspekten als auch den hohen Qualitätsanforderungen, denen Datenbank-Software bezüglich ihrer Korrektheit unterliegt, tragen moderne Polymorphiekonzepte Rechnung [Milner 78; Cardelli 84; Matthes 92]. Sie erlauben unter der Voraussetzung einer durchgängigen, statisch überprüfbaren Typsicherheit die Formulierung hochgradig generischer Algorithmen, ohne die konventionellen Möglichkeiten der Datenmodellierung einzuschränken.

Einen weiteren wichtigen Beitrag zur Bewältigung obiger Anforderungen leisten Funktionen höherer Ordnung. Allerdings wird vorausgesetzt, daß auch *lexikalisch geschachtelte* Funktionen Datenobjekte „erster Klasse“ sind. Sie kapseln jeweils die dynamischen Instanzen der Daten ihres statischen Sichtbarkeitsbereiches . Durch polymorphe Funktionen höherer Ordnung in einem persistenten System lassen sich zahlreiche Komponenten von Datenbanksystemen mit erheblich geringerem Aufwand als mit konventionellen Methoden (variante Records, Fallunterscheidungen) implementieren [Matthes 92]. Dies gilt vor allem auch für wichtige Standardaufgaben: die Handhabung generischer Datenstrukturen für Massendaten, Mechanismen zur Iterationsabstraktion und die Fehlererholung durch kompensierende Transaktionen [Niederée 92]. Die sprachliche Integration der entsprechenden Konstrukte ist mit den eigenen Mitteln der Programmiersprache möglich (*add-on*-Ansatz [Matthes, Schmidt 91a], z.B. P-

Quest) und *nicht* wie in anderen Systemen nur durch Erweiterung des Sprachkerns. Diesen sogenannten *builtin*-Ansatz vertreten z.B. Napier, Pascal-R und DBPL.
In Zusammenfassung der bisherigen Betrachtungen wird festgestellt:

> Orthogonale Persistenz, Polymorphie und Funktionen höherer Ordnung sind grundlegende Konzepte leistungsfähiger moderner Mehrzweckprogrammiersprachen, die sich besonders für die Entwicklung datenintensiver Systeme eignen sollen.

Im Gegensatz zur *sprachlichen* kann die *systemtechnische* Integration unter Einsatz externer Diensterbringer erfolgen. Ein typischer Mangel der meisten persistenten (und zahlreicher anderer) Systeme ist jedoch das Fehlen leistungsfähiger Programmierschnittstellen. Eine besonders krasses Beispiel ist das Napier-System [Dearle et al. 89], in dem zur Gestaltung graphischer Benutzeroberflächen ein komplettes Fenstersystem neu entwickelt wurde.

Die heutigen persistenten Programmiersprachen sind in der Regel durch Interpreter implementiert. Bei der Einbindung externer Funktionalität treten erhebliche praktische Probleme auf, da meist ihr Interpreterbefehlssatz erweitert werden muß. Eine Erweiterung durch Programmierer, die sich nicht im Besitz der Quelltexte des Systems befinden, ist sogar ausgeschlossen. Zumindest der kommerzielle Einsatz solcher Systeme als Entwicklungsplattform ist damit weitgehend perspektivlos. Die gesamte Problematik läßt sich jedoch dadurch entschärfen, daß sein Befehlssatz um eine *generische* C-Schnittstelle erweitert wird, die *jedem* Programmierer zur Verfügung steht.

Durch eine interpreterbasierte Implementation wird die Maschinenkodegenerierung vermieden. Zum einen ist somit der Implementationsaufwand geringer und zum anderen kann auf einfache Weise ein hoher Portabilitätsgrad erreicht werden, da lediglich der Interpreter portabel sein muß. Der augenscheinlichste Vorteil der Portabilität eines Systems ist die allgemeine Förderung seiner Verbreitung. Für datenintensive Anwendungen tritt ein weiterer Aspekt in den Vordergrund: ihre Langlebigkeit und damit auch die ihrer Daten hängt von der Langlebigkeit ihres Implementationssystems ab. Dessen *eigene* Portabilität ist ebenfalls wichtig, weil der fortschreitenden Entwicklung von Hardware und Betriebssystemen am besten durch Anpassungsfähigkeit begegnet werden kann.

Auf eine Maschinenkodegenerierung sollte trotz der Vorteile eines interpreterbasierten Systems nicht verzichtet werden, da sie in der Regel eine enorme Verbesserung der Performanz bewirkt.

1.1 Zielsetzung

Das Hauptziel der vorliegenden Arbeit ist die Bereitstellung adäquater Methoden der Kodegenerierung für leistungsfähige moderne Mehrzweckprogrammiersprachen mit dem Anwendungsschwerpunkt der Entwicklung datenintensiver Systeme.

Es wird ein durchgehendes Verfahren angestrebt, das alle wichtigen Probleme in kohärenter Weise löst und somit als Grundlage einer Implementation taugt. An diese Forderung schließt das praktische Ziel einer funktionstüchtigen exemplarischen Implementation an, denn Methoden der Kodegenerierung lassen sich erst unter realen Einsatzbedingungen fundiert beurteilen. Dies gilt sowohl für die Qualitäten des generierten Kodes als auch für die der Kodegenerierungs-Software.

Da für die betrachteten Klasse von Programmiersprachen die Verwendung einer portablen, interpretierten Zwischensprache vorausgesetzt wird, sind zwei Aufgaben zu lösen: Zwischenkodegenerierung *und* Maschinenkodegenerierung.

Die Zwischenkodegenerierung portabel zu gestalten, ist naturgemäß einfach. Aber auch die Maschinenkodegenerierung sollte so portabel wie möglich sein. Von hohem Interesse sind alle Möglichkeiten, die es erlauben, effizienten Kode zu erzeugen, ohne Abhängigkeiten von bestimmten Systemarchitekturen zu bedingen. Die dazu benötigten Werkzeuge sollten ebenfalls so portabel wie möglich sein.

Übersetzte Programme sollen wahlweise in Form von Zwischenkode oder Maschinenkode ausgeführt werden können, wobei gegebenenfalls Zwischenkode dynamisch weiterzuübersetzen ist. Dies ist zum Beispiel für große Datenbanksysteme, die Anfrageoptimierungen in heterogenen Netzwerken durchführen, wichtig. Beide Kodearten müssen sich wechselseitig transparent aufrufen können, damit aufwendige Maßnahmen zu ihrer expliziten Koordinierung überflüssig sind. Insbesondere ist zu untersuchen, auf welche Weise Ausnahmen (*exceptions*) über die Grenzen der verschiedenen Implementationsarten hinweg propagiert werden können.

Zur generischen Anbindung externer Diensterbringer ist zumindest eine Programmierschnittstelle zu einer gängigen Mehrzweck- und Systemprogrammiersprache notwendig. Diese muß gleichermaßen vom Zwischenkode und vom Maschinenkode unterstützt werden.

Als absolute Forderung an die zu übersetzende Hochsprache wird die Unterstützung von Persistenz, Polymorphie und Funktionen höherer Ordnung als orthogonale Konzepte vorausgesetzt. Wie in der vorangehenden Einleitung erörtert wird, eröffnen sich dadurch weitreichende neuartige Möglichkeiten der Datenbankprogrammierung.

Für jedes der drei Konzepte existieren bereits diverse erfolgreiche Implementationen. Sie beeinflussen jedoch nicht nur jeweils einzeln wichtige Entwurfsentscheidungen. Aufgrund ihrer Orthogonalität ist ihre *gleichzeitige* Präsenz bei der Ausgestaltung konkreter Datenobjekte und der sie manipulierenden Operationen zu berücksichtigen. Im Extremfall treten persistente, polymorphe Funktionen höherer Ordnung auf. Als Grundlage der Kodegenerierung ist daher ein *allgemeiner* Ansatz erforderlich, der stets die Anforderungen aller drei Konzepte gleichzeitig berücksichtigt. Von diesem Ansatz ausgehend können Analysen vorgenommen werden, die zur Steigerung der Laufzeit- und Speicherplatzeffizienz bestimmte Anforderungen in Spezialfällen ausschließen.

Die konkrete praktische Aufgabe der vorliegenden Arbeit ist die Erstimplementation der Kodegenerierung für die Programmiersprache TL des Tycoon-Systems [Matthes 92]. Als Syste-

mumgebung ist dabei die *Sun Sparc*-Hardware-Architektur und *Sun OS* als Betriebssystem vorgesehen.

Die Sprache TL ist ein besonders ausgeprägtes Beispiel der Klasse leistungsfähiger Mehrzweckprogrammiersprachen mit dem Schwerpunkt Datenbankprogrammierung. Sie weist deren sämtliche Charakteristika auf, insbesondere Persistenz, Polymorphie und Funktionen höherer Ordnung als orthogonale Konstrukte. Ihr leistungsfähiger algorithmischer Kern besteht sowohl aus funktionalen als auch aus imperativen Konstrukten und enthält insbesondere dynamische Ausnahmebehandlungen. TL ist streng typisiert, was aufgrund der hohen Software-Qualitätsanforderungen an die Datenbankprogrammierung eine äußerst wichtige Eigenschaft ist.

Mit Ausnahme zweier Sprachkonstrukte wird eine *vollständige* Implementation von TL angestrebt. Die in der Definition von TL [Matthes 92] vorgesehenen „Records“ und „dynamischen Typen“ sind erst als spätere Erweiterungen vorgesehen. Sie werden im folgenden der Einfachheit halber als nicht im Sprachumfang enthalten betrachtet.

Einige Konstrukte von TL stellen besondere Anforderungen an die Kodegenerierung. Vor allem gilt dies für rekursive Wertbindungen. Rekursive Funktionen sind ein bekannter Spezialfall derselben. Ihre Implementation ist eine Minimalbedingung für den sinnvollen Einsatz von TL. Zusätzlich ist zu untersuchen, inwiefern sich das Konzept, Werte rekursiv anzugeben, in einer allgemeineren Form in die Praxis umsetzen läßt.

Eine zentrales Problem ist die uneingeschränkte Verwendbarkeit veränderlicher Bindungen bei gleichzeitiger Unterstützung eines funktionalen Programmierstils. Es betrifft insbesondere die Repräsentation von Funktionsabschlüssen.

Die Zwischensprache TML ist ein fester Bestandteil des Tycoon-Systems. Sie basiert auf einem untypisierten Lambda-Kalkül und enhält außerdem imperative Konstrukte. Diese vorweggenommene Wahl ist zumindest sehr vielversprechend, da schwach- oder untypisierte Lambda-Kalküle sich schon oft als Zwischensprachen [Cardelli 83; Cardelli 86b; Peyton Jones 87] bewährt haben und auch in kommerziellen Systemen [Muchnick 90; RSRE 91; Schmidt, Matthes 92] als quellsprachenunabhängige Zwischenrepräsentationen eingesetzt werden. Hinzu kommt, daß TML mit den anderen Komponenten des Tycoon-Systems harmonisiert.

Bei der Implementation der Übersetzung ist eine vollkommene Beschränkung auf die Aufgaben der Kodegenerierung möglich, weil die Syntaxanalyse und Typüberprüfung von anderen Systemkomponenten geleistet werden, die bereits voll funktionsfähig sind. Außerdem bestehen in diesem Fall fast ausschließlich streng hierarchische Abhängigkeiten, so daß im Entwicklungsprozeß relativ wenige Rückkopplungen an dieser Schnittstelle zu erwarten sind. Für das Laufzeitsystem einschließlich des Interpreters gelten jedoch andere Bedingungen. Dieses steht als ausführende, bzw. unterstützende Einheit des generierten Kodes in einer engen *Wechselbeziehung* zur Kodegenerierung, so daß von einem hohen Bedarf an beiderseitigen Anpassungen auszugehen ist.

1.2 Übersicht

Der Hauptteil der Arbeit beginnt mit der Beschreibung der Programmierumgebung Tycoon (Kapitel 2). In diesem Zusammenhang werden insbesondere die zu übersetzende Programmiersprache TL und das Objektspeicherprotokoll TSP vorgestellt. Die anschließende ausführliche Beschreibung der Zwischensprache TML (Kapitel 3) umfaßt die formale Definition ihrer dynamischen Semantik sowie technische Aspekte ihrer Implementation und interpretativen Ausführung.

TML tritt als Zielsprache der Zwischenkodegenerierung (Kapitel 4) und als Quellsprache der Maschinenkodegenerierung (Kapitel 5) auf. Die zur Durchführung dieser Transformationen bereitgestellten Methoden sind die Hauptbeiträge der vorliegenden Arbeit.

Die Arbeit endet mit einer Zusammenfassung der wichtigsten Ergebnisse und Erfahrungen. In den Anhängen sind die Zwischenrepräsentationen der beschriebenen Übersetzungsschritte und die Algorithmen einiger besonders wichtiger Teillösungen im Programmquelltext aufgeführt.

Die Sprache TL ist im folgenden nicht nur als Quellsprache des generierten Kodes Gegenstand der Betrachtung, sondern sie wird auch zur Programmdarstellung der Kodegenerierung eingesetzt. Dies ist ein Vorgriff auf die Selbstimplementierung von Tycoon, denn gegenwärtig liegt seine gesamte Implementation noch in Quest vor. In Anbetracht der Tatsache, daß Quest alsbald nicht mehr benötigt wird, hat eine Programmdarstellung in TL länger Bestand.

Kapitel 2

Das Tycoon-System

Das Tycoon[1]-System ist eine moderne Programmierumgebung, die den sprachlichen und architektonischen Rahmen für die flexible Definition und Integration generischer Dienste bereitstellt. Zur Definition neuer und zur Einbindung externer Dienste dient die Programmiersprache TL. Sie bietet den Rahmen zur Benennung, Bindung und Typisierung der relevanten Objekte und Dienste und erfüllt die Aufgaben einer sogenannten Mehrzweckprogrammiersprache (*general purpose programming language*).

Der Hauptanwendungszweck des Tycoon-Systems ist die Realisierung datenintensiver Applikationen. TL bietet die volle Leistungsfähigkeit integrierter Datenbankprogrammiersprachen (wie z.B. DBPL), insbesondere die orthogonale Kombinierbarkeit elementarer Persistenzkonzepte mit typvollständiger Datenstrukturierung und Iterationsabstraktion. Wie auch in anderen modernen persistenten Systemen (z.B. P-Quest, Napier, O2) wird die Persistenz von Daten *und* Programmrepräsentationen durch eine auf niedrigem Abstraktionsniveau angesiedelte Software-Schnittstelle unterstützt.

Da neben der Programmerstellung und -exekution insbesondere in Datenbankanwendungen Optimierungs-, Portierungs- und Transformationsschritte erforderlich sind, existieren im Tycoon-System multiple Programmrepräsentationen auf verschiedenen semantischen Abstraktionsebenen.

Das Tycoon-System ist auf eine möglichst hohe Systemkompatibilität mit existierenden (meist kommerziellen) Diensterbringern ausgerichtet, um deren Funktionalität maximal auszunutzen. Besonders hervorzuheben ist dabei, daß die Realisation und Koordination von Tycoon-Anwendungen in einem unifizierenden sprachlichen Rahmen geschieht. Abbildung 2.1 zeigt die Manipulation heterogener Objekte einer Tycoon-Datenbankapplikation in verschiedenartigen Medien (D: Datenbankobjekte, P: Programmobjekte, B: Bildschirmobjekte).

In diesem Kapitel wird zuerst das Gesamtsystem vorgestellt. Dann wird die Programmiersprache TL beschrieben. Nach einer knappen Darstellung ihrer Grundzüge wird auf Aspekte eingegangen, die für die Kodegenerierung besonders problematisch erscheinen: Variablen, Funktionen höherer Ordnung, Polymorphie und Erweiterungskonzepte (Module, Bibliotheken). Während TL systemtechnisch den oberen Abschluß der Kodegenerierung bildet, stellt das Objektspeicherprotokoll TSP den unteren dar. In seiner Beschreibung wird insbesondere auf Implikationen für die Kodegenerierung eingegangen.

[1] Tycoon: *Typed communicating objects in open environments* [Matthes 92].

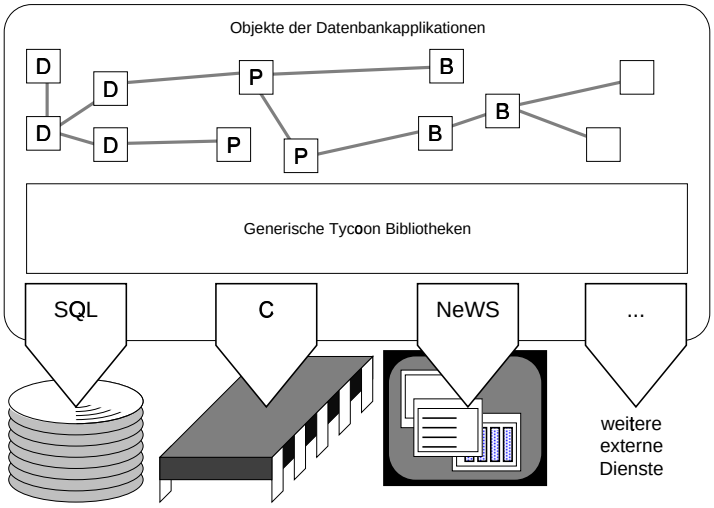

Abbildung 2.1: Integration, Erweiterung und Nutzung generischer Dienste in Tycoon [Matthes 92]

2.1 Die Tycoon-Systemarchitektur

Die Komponenten der Tycoon-Systemarchitektur werden durch drei Schnittstellen in vier Schichten getrennt. Abbildung 2.2 stellt diese Gliederung dar. Die Schnittstellen sind:

- Die Programmiersprache *Tycoon Language* (TL), eine moderne Mehrzweckprogrammiersprache. Sie wird im Tycoon-System nicht nur zur Datenmodellierung und Applikationsprogrammierung eingesetzt, sondern sie bildet auch die Systemprogrammiersprache, in der die frei erweiterbaren Tycoon-Bibliotheken und die Tycoon-Sprachprozessoren implementiert sind. TL wird in Abschnitt 2.3 vorgestellt.
- Die Zwischensprache *Tycoon Machine Language* (TML), die auf einem untypiserten Lambda-Kalkül mit Funktionsabschlüssen zur Unterstützung statischer Sichtbarkeitsregeln basiert. Sie wird in Kapitel 3 eingehend beschrieben. TML ist gewissermaßen der „zentrale“ Gegenstand der vorliegenden Arbeit: es wird sowohl die Übersetzung *nach* TML (Kapitel 4) als auch die *von* TML (Kapitel 5) behandelt. TML-Kode kann sowohl interpretiert als auch in Maschinenkode übersetzt werden.
- Die Objektspeicherschnittstelle *Tycoon Store Protocol* (TSP), die operationale Qualitäten des Objektspeichers (Persistenz, Speicherrückgewinnung etc.) aus Sicht der höheren Schichten verbirgt. Sie wird in Abschnitt 2.4 beschrieben.

Es besteht eine konsequente Trennung zwischen den Aufgaben der Datenmodellierung (TL), Datenmanipulation (TML) und Datenspeicherung (TSP).

Die vorliegende Arbeit handelt im wesentlichen von der Implementation der in Abbildung 2.2 als „Compiler Backend“ bezeichneten Systemkomponente. Im folgenden wird ausschließlich der TML-Kodegenerator „Backend“ genannt; der Maschinenkodegenerator wird als eine eigenständige Komponente angesehen.

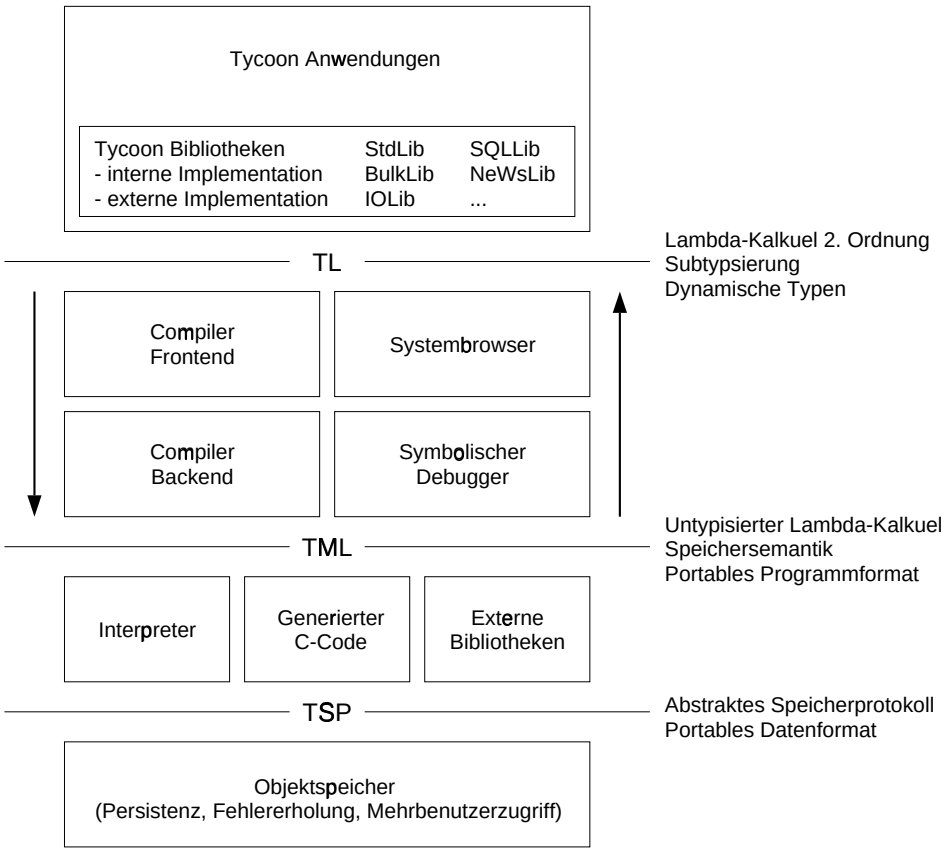

Abbildung 2.2: Die Schichten und Schnittstellen der Tycoon-Systemarchitektur [Matthes 92]

2.2 Inkrementelle Übersetzung

Zahlreiche konventionelle Übersetzersysteme (wie zum Beispiel C-, Modula-2- oder Ada-Compiler) weisen eine strikte Trennung der Übersetzungs- und der Laufzeitumgebungen, insbesondere bezüglich der Sichtbarkeitsbereiche, auf. Dadurch ist jede Übersetzungseinheit in sich abgeschlossen und muß im Falle von Erweiterungen komplett neu übersetzt werden.

Bei der inkrementellen Übersetzung wird das ausführbare Program schrittweise erweitert: Eingabe-, Übersetzungs- und Ausführungsphasen wechseln sich zyklisch ab; ihre relevanten Ergebnisse werden für die nachfolgenden Phasen konserviert. In interaktiven Systemen ist es dabei notwendig, daß der Übersetzer im Laufzeitsystem enthalten ist.

Die interaktiven Programmierumgebungen zahlreicher (meist funktionaler) Systeme und auch die des Tycoon-Systems verhalten sich, obwohl sie übersetzerbasiert sind, der eines Interpreters sehr ähnlich. Solche Systeme werden deshalb auch als Quasi-Interpreter bezeichnet. Typische Vertreter sind zum Beispiel diverse Scheme-Implementationen, aber auch Quest und P-Quest. In Quasi-Interpretern wird jede abgeschlossene Benutzereingabe sofort übersetzt und direkt anschließend ausgeführt (eventuelle Fehlerbehandlung wird hier nicht betrachtet). Daraufhin wird der Ergebniswert angezeigt und die nächste Eingabe erwartet. Wesentlich ist, daß die für die Sprache geltenden Sichtbarkeitsregeln in der interaktiven Programmierumgebung beibehalten werden. Eine notwendige Voraussetzung für dieses Verhalten ist eine Datenstruktur, die alle aus Eingabe, Übersetzung und Ausführung bestehenden Zyklen überdauert.

Das Tycoon-System bietet einen fließenden Übergang zwischen der stapelorientierten Übersetzung, Bindung und Exekution von Modulen und Bibliotheken und der direkten Evaluation interaktiv eingegebener TL- Terme. Dies ermöglicht den Einsatz von TL zur Datenbankprogrammierung, insbesondere auch für *ad-hoc*-Datenbankanfragen. Außerdem ist die Grundlage für nicht-interaktives inkrementelles Übersetzen geschaffen. Eine wichtige Anwendung dieser Möglichkeit ist die zur Laufzeit stattfindende Programmreflektion (*runtime reflection*).

In Zusammenhang mit der interaktiven Benutzerumgebung kommt der von Tycoon unterstützten Persistenz eine besondere Bedeutung zu: sie ermöglicht ein inkrementelles Verhalten über mehrere zeitlich getrennte Benutzersitzungen hinweg.

2.3 Die Programmiersprache TL

In der folgenden Beschreibung der Programmiersprache TL stehen die im Rahmen der vorliegenden Arbeit relevanten Aspekte im Vordergrund. Der nachstehende Abschnitt bietet zunächst eine Übersicht der Programmiersprache TL und umreißt ihre wichtigsten Konstrukte und Eigenschaften. Anschließend führt Abschnitt 2.3.2 in die Syntax ein. Zur ausführlicheren Einführung in die Programmiersprache TL wird auf Kapitel 4 in [Matthes 92] verwiesen.

Die Programmiersprache TL unterstützt alle drei Konzepte, die in der vorliegenden Arbeit besondere Beachtung erfahren: Persistenz, Polymorphie und Funktionen höherer Ordnung. Persistenz bleibt jedoch zunächst außer Betracht, weil sie in TL vollständig transparent ist. Der Ansatz zu ihrer Unterstützung wird in Abschnitt 2.4 im Zusammenhang des Objektspeicherprotokolls beschrieben.

Die Implementation von Funktionen höherer Ordnung ist eines der aufwendigsten Probleme der Kodegenerierung. Es kann nur in enger Beziehung zu den Variablenkonzepten in TL begriffen werden, da auch geschachtelte Funktionen Datenobjekte „erster Klasse“ sind. Abschnitt 2.3.3 stellt zunächst die Variablenkonzepte vor, woran die Motivation von Funktionen höherer Ordnung in Abschnitt 2.3.4 anschließt.

Die für die Kodegenerierung relevanten Aspekte der Polymorphie werden in Abschnitt 2.3.5 betrachtet.

2.3.1 Grundzüge der Programmiersprache TL

TL ist im wesentlichen eine Weiterentwicklung von Quest [Cardelli 89; Cardelli 90], und P-Quest [Matthes 91; Müller 91].[2] Abgesehen von geringen syntaktischen Unterschieden werden alle sprachlichen Konstrukte von Quest in TL voll unterstützt. Diese „Aufwärtskompatibilität“ wird ausgenutzt, indem die Erstimplementation des Tycoon-Systems in Quest erfolgt. Alle Programmteile des Systems können dann durch einfache syntaktische Transformationen nach TL übertragen werden.

Die Sprache TL ähnelt in Bezug auf ihre syntaktische Struktur und ihr Modulkonzept den Sprachen der Modula-Familie (Modula-2 [ModISO 91], Oberon [Wirth 87], Modula-2+ [Rovner et al. 85], Modula-3 [Nelson 91] und Ada [Ichbiah, others 83]). Semantisch ist TL eng mit polymorphen funktionalen Sprachen der ML Sprachfamilie verwandt [Cardelli 89; Cardelli 90; Mauny 91; Field, Harrison 88; Hudak 89]. Der Kern der semantischen Konzepte von TL entspricht dem Modell der Sprache $F_{\leq}$ [Cardelli et al. 91], einer allgemein anerkannten formalen Basis für die Studie neuartiger Typsysteme. Dieses explizit und strikt typisierte Lambda-Kalkül zweiter Stufe mit Subtypisierung ist in eine vollständig entwickelte, modulare Programmiersprache eingebettet.

TL bietet eine Symbiose des funktionalen und des imperativen Programmierstils. Eine Variante des objektorientierten[3] Programmierstils läßt sich aufgrund weitgehender sprachlicher Neutralität ebenfalls anwenden [Matthes 92]. Relationale und logikbasierte Programmierung

[2] P-Quest und Quest sind bis auf eine Ausnahme gleich: das P-Quest-System ist um orthogonale Persistenz erweitert.

[3] In Ermangelung einer allgemein anerkannten Definition dieses Stils existieren ausschließlich „Varianten“.

[Minker 88] wird nicht unmittelbar unterstützt, weil sich unifikationsbasierte Evaluationsmodelle und der deklarative Ansätze zu stark von funktionalen und imperativen Strukturen unterscheiden [Apt 90]. Erstere können jedoch mit einigem Aufwand durch letztere nachgebildet werden [Matthes 92].

Auf imperative Zustandsänderungen (*updates*) kann schon aufgrund ihrer punktuellen Wirkung bei der Manipulationen von Massendaten nicht verzichtet werden. Darüberhinaus bietet TL imperative Kontrollstrukturen (z.B. **if**, **case**, **loop**).[4] TL hat eine strikte Evaluationssemantik, weil diese vorteilhaft mit imperativen Sprachkonzepten interagiert und eine direkte Abschätzung der Speicherkomplexität eines Algorithmus gestattet (vgl. [Hudak 89; Peyton Jones 87]). Die streng deterministische Evaluationsreihenfolge entspricht der Lesart der lexikalischen Notation: „von links nach rechts".

Eine besonders wichtige Kontrollstruktur ist die dynamische Ausnahmebehandlung (*exception handling*). Sie vereinfacht sowohl die Strukturierung als auch die Integration fehlerträchtiger Dienste erheblich [Niederée 92], wovon insbesondere Modulschnittstellen profitieren.

In TL ist weder eine automatische Wertkonvertierung noch ein Überladen von symbolischen oder alphanumerischen Bezeichnern (wie z.B. in Ada) möglich.

Folgende Basistypen sind vordefiniert: *Int*, *Real*, *Longreal*, *Char* und *String*. Die statisch im Backend zu verankernde interne Repräsentation der Basistypen wird durch die Vorgaben des Objektspeicherprotokolls bestimmt.

2.3.2 Elemente der TL-Syntax

Programmkommentare haben die Form (∗ . . . ∗). Die Angabe von Werten der TL-Basistypen basiert auf Symbolproduktionen zur Definition von Programmliteralen. Beispiele:

```
false  true      (* Die beiden Wahrheitwerte des Typs Bool *)
42               (* Eine Ganzzahl vom Typ Int *)
3.14             (* Eine Fließkommazahl vom Typ Real *)
"Hello World!"   (* Eine Zeichenkette vom Typ String *)
```

TL besitzt eine LL(1) Grammatik mit einleitenden Schlüsselworten und schließendem **end**. Die wichtigsten Schlüsselworte sind **let** zur Einleitung von Wertbindungen und **Let** für Typbindungen. Beispiel:

```
let a = 7
```

Die Variable *a* wird durch diese Anweisung deklariert, initialisiert und benannt.

Wert- und Typbezeichner können in TL jederzeit mit (Meta-) Typinformationen annotiert werden. Für Bezeichner, die dynamischen oder rekursiven Bindungen unterliegen (z.B. Formalparameter in Funktionen oder exportierte Bezeichner in Modulschnittstellen) sind diese Annotationen verpflichtend, während sie ansonsten automatisch durch den Übersetzer inferiert werden. Beispiel:

[4] Diese lassen sich in nicht-strikten funktionalen Sprachen (*lazy evaluation*) auf Funktionen zurückführen, was zwar die Grammatik, nicht jedoch die Kodegenerierung vereinfacht ([Abelson et al. 84; Peyton Jones 87]).

```
let a :Int = 7
let f(x :Bool) :Bool = x        (* Funktion *)
```

Da das Programmliteral 7 den Typ *Int* bereits implizit vorgibt, ist seine Angabe redundant. Die explizite Festlegung des Parameters *x* ist jedoch obligatorisch.

TL ist wertorientiert, d.h. jeder Term evaluiert zu einem (evtl. trivialen) Wert. Durch die Schlüsselworte **begin** und **end** werden „Blöcke" eingefaßt, die Anweisungssequenzen enthalten. Die Sichtbarkeit der in einem Block deklarierten Variablen ist durch das Blockende begrenzt. Beispiel:

```
let a = 1
begin
  let a = 2
  let b = a     (* 2 *)
end
a               (* 1 *)
```

Die Variable *b* wird mit dem Wert 2 initialisiert. Der in der letzten Zeile stehende Bezeichner a bezieht sich jedoch auf die zuerst deklarierte Variable dieses Namens. Daher ist der Wert 1 das Endergebnis. Dies gilt im nachfolgenden Beispiel analog, denn Verzweigungen in Kontrollstrukturen gelten ebenfalls als Blöcke.

```
let a = 1
if x > 7 then
  let a = 2
  ...
else
  ...
end
a               (* 1 *)
```

Weitere in den TL-Beispielen der nachfolgenden Kapitel auftretende Konstrukte werden jeweils im betreffenden Zusammenhang erläutert, sofern sie nicht hinreichend intuitiv sind.

2.3.3 Die Variablenarten in TL

In der vorliegenden Arbeit wird die Bezeichnung „Variable" nicht nur für veränderbare sondern auch für unveränderbare Datenobjekte verwendet. Dies widerspricht zwar der etymologisch naheliegenderen Bedeutung, ist aber inzwischen in der Terminologie vieler Programmiersprachen gängig. Die Begriffsverschmelzung beruht darauf, daß die Syntax für die Deklaration und Benutzung von Konstanten und „echten" Variablen weitestgehend vereinheitlicht ist. Außerdem stellen sich bei der Betrachtung der Implementation über alle Systemschichten hinweg weitere Gemeinsamkeiten heraus. Die kennzeichnende Eigenschaft von Variablen, auf die es im folgenden ankommt, ist, daß sie sich im Unterschied zur direkten Angabe von Werten durch Programmliterale indirekt, unter Verwendung eines Bezeichners auf einen Wert beziehen.

Die Veränderbarkeit einer Variablen wird in TL durch das Schlüsselwort **var** ausgedrückt. Beispiele:

```
let var x = 4710    (* x ist veränderbar. *)
let y = 4711        (* y ist nicht veränderbar. *)
```

In TL treten folgende Variablenarten auf:

- Variablen, die direkt in der interaktiven Benutzerumgebung, dem „*Toplevel*“, vereinbart werden und dementsprechend „*Toplevel*-Variablen“ genannt werden. Sie befinden sich *nie* innerhalb eines Blockes oder einer Funktion.
- Lokale Variablen. Sie gleichen syntaktisch *Toplevel*-Variablen, befinden sich jedoch *immer* innerhalb eines Blockes oder einer Funktion.
- Funktionsparameter, die in Funktionssignaturen deklariert werden.
  Beispiel: **let** *f(x :Int y :String)* :**Ok** = ...
- Felder in Aggregaten (Tupel, Arrays, Ausnahmewerte).
  Beispiel: **tuple let** a = *1* **let** *b = 2* **end**

Eine Variable steht zu einer bestimmten Funktion stets in einer der folgenden drei Beziehungen, die sich wechselseitig ausschließen:

- Sie ist ein Parameter der Funktion.
- Sie ist eine lokale Variable im Rumpf der Funktion.
- Sie ist eine globale Variable, d.h. es handelt sich um eine Variable, die in einer umgebenden Funktion oder im *Toplevel* deklariert ist.

Abbildung 2.3 zeigt die möglichen Beziehungen der verschiedenen Variablenarten. Zum Beispiel kann eine lokale Variable tatsächlich als solche in einer Funktion sichtbar sein oder aber als globale Variable. *Toplevel*-Variablen dagegen werden stets als globale Variablen angesprochen.

2.3.4 Funktionen höherer Ordnung

In TL liefert jede Funktionsabstraktion einen gleichberechtigten Wert (*first class value*). Dies gilt insbesondere auch für geschachtelte Funktionen, die auf Datenobjekte des Sichtbarkeitsbereiches, der sie umgibt, zugreifen.

Die folgenden Beispiele entstammen [Abelson et al. 84], wo sie in Scheme angegeben sind. Sie simulieren jeweils den (stark vereinfachten) Vorgang des Abhebens eines Geldbetrages von einem Bankkonto. Die Variable *balance* ist global zur Funktion *withdraw*, d.h. sie befindet sich in deren äußerem Sichtbarkeitsbereich. Im ersten Beispiel ist *withdraw* eine ungeschachtelte Funktion. Diese Anordnung wird auch von Sprachen mit einem primitiven Konzept zur Repräsentation von Funktionsobjekten wie Pascal, Modula-2 oder C unterstützt.

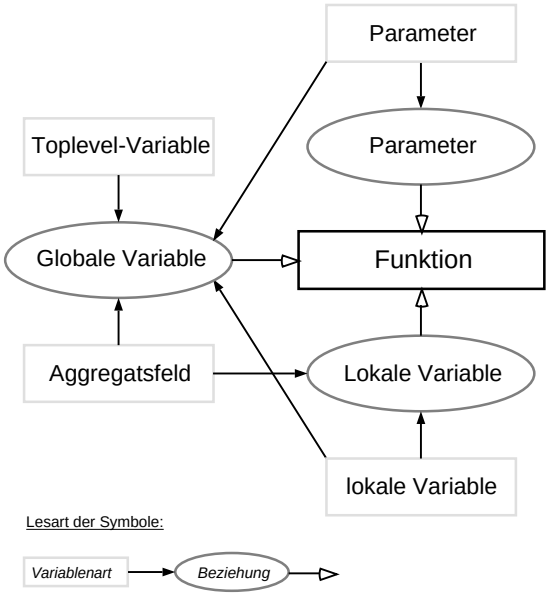

Abbildung 2.3: Die Beziehungen von TL-Variablen zu Funktionen

```
let var balance = 100
let withdraw(amount :Int) :Int =
  if balance >= amount then
    balance := balance - amount
    balance
  else
    raise exception "error" with "Insufficient funds" end
  end

withdraw(25)
⇒ 75
```

Die Notation ⇒ *output* kennzeichnet Ausgaben des Tycoon Systems.

Im nächsten Beispiel werden zwei Instanzen einer unbenannten Funktion erzeugt, die jeweils einzeln die Aufgabe von *withdraw* erfüllen.

```
let makeWithdraw(balance :Int) =
  fun(amount :Int) :Int
    if balance >= amount then
      balance := balance - amount
      balance
    else
      raise exception "error" with "Insufficient funds" end
    end
```

```
let w1 = makeWithdraw(100)
let w2 = makeWithdraw(100)

w1(50)
⇒ 50

w2(70)
⇒ 30

w2(40)
⇒ exception "error" with "Insufficient funds" end

w1(40)
⇒ 10
```

Die Funktionen *w1* und *w2* besitzten jeweils eine eigene, unabhängige Instanz des Parameters *balance*.

In TL können Funktionen höherer Ordnung genau wie in Scheme [Abelson et al. 84; Steele 86] in beliebig tiefer Schachtelung auf globale Variablen zugreifen. Unter diesem wichtigen Aspekt der Ausdruckskraft steht TL also selbst einem modernen Lisp-Dialekt nicht nach.

2.3.5 Polymorphie

Ohne das Thema vertiefen zu wollen, sei eine kurze Definition des Begriffes Subtyppolymorphismus genannt:

> Variablen können dynamisch an Objekte verschiedener Struktur gebunden werden, sofern deren Struktur eine gemeinsame generalisierte Spezifikation erfüllt.

TL definiert eine induktiv definierte Subtypbeziehung auf Typen und hochentwickelte Konzepte ihrer Verwendung zur existentiellen und universellen Typquantifizierung. Es werden die Konzepte des *parametric polymorphism* [Milner 78], des *subtype polymorphism* [Cardelli 84] und des *bounded parametric polymorphism* [Meyer 86] abgedeckt. Darüber hinaus ist die Subtypbeziehung auf Typen für benutzerdefinierbare Typoperatoren höherer Ordnung generalisiert.[5] TL bietet damit alle bekannten Polymorphiekonzepte, die im Rahmen eines typsicheren Systems unterstützt werden können und ist daher ein geeignetes Objekt zur Untersuchung der Möglichkeiten der Kodegenerierung für polymorphe Programmiersprachen.

Für die Kodegenerierung ist folgender Aspekt von entscheidender Bedeutung: alle genannten Polymorphismen basieren auf der Uniformität der Daten, welche durch polymorphe Operationen manipuliert werden. Unnötige Spezialisierungen im Aufbau von Datenobjekten schränken ihre polymorphe Verwendung also ein. Die zulässige Verschiedenartigkeit von TL-Datenobjekten wird anhand der hochsprachliche Typinformationen *statisch* überprüft [Matthes 92]. Letztere können daher im Zuge der Kodegenerierung eliminiert werden, was die Definitionen der Zwischensprache TML und des Objektspeicherprotokolls TSP auch vorsehen.

[5] Zur genaueren Betrachtung wird auf Kapitel 4 in [Matthes 92] verwiesen.

2.3.6 Erweiterungskonzepte in TL

Der hochgradig generische und orthogonale Sprachkern von TL bietet ein hohes Maß an Offenheit und Flexibilität. Nur sehr wenige Primitive sind in der stark reduzierten Kernsprache vordefiniert (Variablen, Funktionen, Typ-variablen, Typoperatoren, einige Kontrollstrukturen). Zur (typsicheren) Unterstützung weiterer semantischer Objekte (Ganzzahlen, Fließkommazahlen, Zeichenketten, Relationen, Sichten, Fenster, etc.) wird der *add-on*-Ansatz verfolgt (vgl. [Matthes, Schmidt 91a]).

Die transparente Unterstützung der Persistenz und die leistungsfähigen Sprachkonstrukte von TL, insbesondere Polymorphie und Funktionen höherer Ordnung, erlauben die Implementation wichtiger Basisfunktionalitäten von Datenbanken in TL selbst. Zum Beispiel wird in [Matthes, Schmidt 91a] beschrieben, wie der *add-on*-Ansatz im weiteren Ausbau des Tycoon-Systems zur Unterstützung hochsprachlicher Abstraktionen für mengenorientierte Datenverarbeitung eingesetzt wird. In Weiterführung dieses Konzeptes wird in [Niederée 92] die Implementation datenintensiver Anwendungen detailliert behandelt. Aus dieser Arbeit geht unter anderem hervor, daß sogar Transaktionen und komplexe Formen der Fehlererholung mit den sprachlichen Mitteln von TL formuliert werden können. Als wesentliche Grundlage ihrer Implementation dient die dynamische Ausnahmebehandlung (*exception handling*).

Zur Organisation auf Wiederverwendbarkeit ausgerichteter Programmteile dienen Module. Im Unterschied zu konventionellen modularisierten Programmiersprachen (z.B. Modula-2) sind als Instanzen einer Modulschnittstelle *mehrere* (implementatorisch unterschiedliche) Module möglich. Sie sind außerdem Datenobjekte „erster Klasse“.

Die Aufgabe der Implementation von Modulen und Modulschnittstellen ist aus Sicht der Kodegenerierung stark vereinfacht. Der einzige zu berücksichtigende Umstand ist die Abgrenzung von Übersetzungseinheiten, denn das Frontend des TL-Übersetzers führt Module und Schnittstellen durch syntaktische Transformationen auf Grundelemente von TL zurück. Modulschnittstellen werden durch Tupel-Typen repräsentiert und Module durch Funktionen, die ein entsprechendes Tupel erzeugen. Die Ausführung einer solchen Erzeugerfunktion entspricht dem dynamischen „Linken“ eines Moduls. Alle zu exportierenden Bestandteile des Moduls werden dabei in dem als Funktionsergebnis zurückzugebenden Tupel zusammengestellt. Die Erzeugerfunktion kapselt die importierten Werte, die in ihrem globalen Sichtbarkeitsbereich liegen. Diese und die nicht exportierten lokalen Werte der Erzeugerfunktion werden durch die exportierten Funktionen wiederum als interne Zustände des fertigen Moduls gekapselt.

Da das Tycoon-System für die „Programmierung im Großen“ konzipiert ist, bietet es eine zusätzliche Strukturierungsebene, in der Module zu Bibliotheken (*libraries*) organisiert werden. Diese Aufgabe hat keine Berührungspunkte mit der Kodegenerierung, die besondere Anforderungen bedingen. Sie wird zum größten Teil rein hochsprachlich bewältigt.

2.4 Die Objektspeicherschnittstelle TSP

Sowohl die vollständige Spezifikation des TSP (*Tycoon Store Protocol*), die eine wesentliche Grundlage der Tycoon-Implementation ist, als auch die Beziehungen zwischen TSP und TML (*Tycoon Machine Language*) sind in [Matthes 92] vorgegeben. TML ist in den Grenzen eingeschränkter Anforderungen unabhängig von einem bestimmten Objektspeicherprotokoll. In

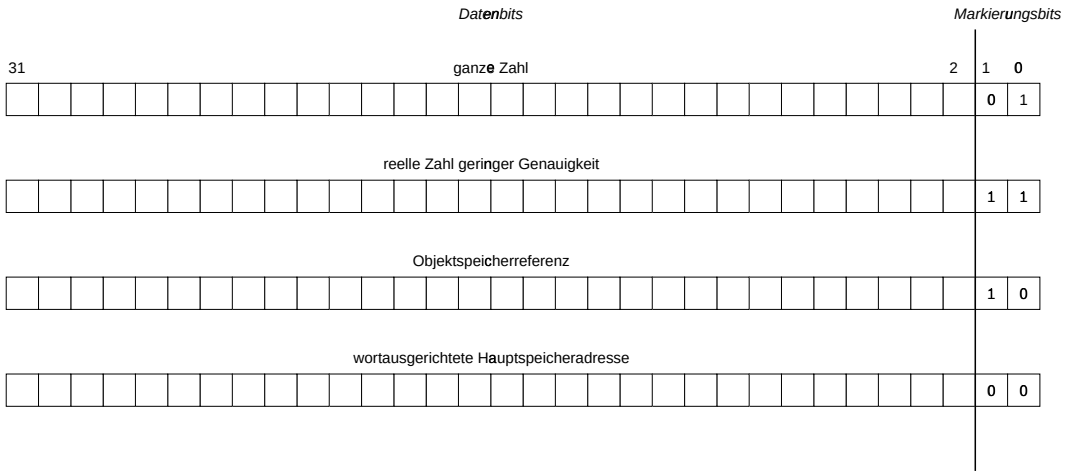

Abbildung 2.4: Die markierte Datenrepräsentation im Tycoon-System [Matthes 92]

Abschnitt 3.5 werden zur formalen Definition der Semantik von TML die relevanten Aspekte des Objektspeicherprotokolls in einer an diese Aufgabe angepaßten Notation dargestellt.

Das TSP bietet die Abstraktion eines homogenen unbegrenzten Objektspeichers, der das Anlegen, Lesen und Ändern von polymorphen, inhomogenen Zustandsvektoren (Speicherobjekten) gestattet. Es ist durch eine reine Softwareschnittstelle realisiert. Speicherzugriffe generierten Kodes können daher nicht direkt durch Maschineninstruktionen implementiert werden.

Es wird vorausgesetzt, daß Datenworte und Adressworte der zugrundeliegenden Systemumgebung die gleiche Größe haben und frei konvertierbar sind.[6] Sie werden im folgenden zusammenfassend „Speicherworte“ genannt. Ein Speicherwort beansprucht 32 Bits, wovon 30 Bits als Nutzdaten zur Verfügung stehen, während zwei Markierungsbits vier Arten von Werten unterscheiden:

- Abstrakte Objektspeicherreferenzen, die Speicherobjekte eindeutig identifizieren.
- Ganzzahlen.
- Fließkommazahlen mit geringer Genauigkeit.
- Wortausgerichtete Hauptspeicheradressen.

Es werden die beiden niederwertigsten (*least significant*) Bits zur Markierung benutzt. Abbildung 2.4 zeigt das Markierungsschema (*tagging*). Wortausgerichtete Hauptspeicheradressen sind dementsprechend stets durch 4 teilbar (4-*byte aligned*, bzw. 2-*bit aligned*).

Die Größe der Speicherobjekte wird beim Anlegen festgelegt und ist nicht dynamisch veränderbar. Es existiert keine Beschränkung der Objektgröße. Die Speicherobjekte sind auf einem

[6] Dies gilt z.B. in einigen Konfigurationen des Betriebssystems MS-DOS und im 24-Bit-Modus des *Finders* für Apple-Macintosh nicht. Neuere Versionen oder Zusätze (*Finder System 7*, *Windows-NT*) ermöglichen jedoch die Kompatibilität von Daten- und Adressworten. Im Betriebssystem *Sun OS*, dem vorrangigen Zielsystem dieser Arbeit, treten keine diesbezüglichen Probleme auf.

niedrigen Abstraktionsniveau selbstbeschreibend. Sie enthalten Formatinformationen, die beim Anlegen festgelegt werden und den späteren Zugriff reglementieren. Außer der als Anzahl der belegten Speicherworte angegebenen Objektgröße wird eine Fallmarke gespeichert, die folgende Fälle unterscheidet:

valueArrayFormat: Ein Vektor von Speicherworten.

immediateArrayFormat: Ein Vektor von Ganzzahlen oder Fließkommazahlen, der keine Referenzen enthält.

byteArrayFormat: Ein Bytevektor.

longRealArrayFormat: Ein Vektor von doppeltgenauen Fließkommazahlen (IEEE-Standard [IEEE 85]). Jede dieser Zahlen belegt zwei Speicherworte.

closureFormat: Dieses Format entspricht *valueArrayFormat*. Zusätzlich wird dem Objektspeicher bekanntgegeben, daß dieser Vektor Referenzen auf ausführbaren TML- oder Maschinenkode enthält.

Die Adressierung der Inhalte von Speicherobjekten findet ausschließlich durch Paare, die aus einer Objektspeicherreferenz und einem relativen Offset innerhalb des referenzierten Objektes bestehen, statt. Sowohl die Schrittweite des Offsets als auch die Größe des adressierten Inhaltes richten sich nach dem Format des referenzierten Objektes.

Zur Effizienzsteigerung bietet das TSP die Möglichkeit, Objekte direkt im Hauptspeicher abzubilden (zu *fixieren*), wo sie direkt durch Maschineninstruktionen manipuliert werden können. Zur Unterstützung der Objektverwaltung muß bereits beim Fixieren der intendierte Zugriffsmodus (Lesen, Lesen und Ändern) angegeben werden. Fixierte Objekte müssen explizit freigegeben werden. Zuvor autorisierte Änderungen werden daraufhin in den Objektspeicher übernommen. Der Fixierungsmechanismus ist keine Besonderheit des Tycoon-Systems, sondern er wird zumindest in sehr ähnlicher Form von fast allen persistenten Objektspeichersystemen (vgl. [Brown, Rosenberg 91; Moss 89; Velez et al. 89]) angeboten. Die Gefahr, daß durch die Nutzung dieser Möglichkeiten der Einsatzbereich der Kodegenerierung von vornherein beschränkt wird, ist daher gering.

Es erfolgt eine automatische Freispeicherverwaltung (*garbage collection*), die ein ausgezeichnetes Wurzelobjekt und die Inhalte der augenblicklich fixierten Objekte als Zustandsvariablen (*state pointer*), d.h. als Ausgangspunkte zur Bestimmung der Erreichbarkeit von Objekten, verwendet. Eine explizite Freigabe von Speicherobjekten ist nicht möglich. Objektspeicherreferenzen können durch die *garbage collection* geändert werden. Alle weiteren nicht im Objektspeicher befindlichen Zustandsvariablen müssen daher ebenfalls der Kontrolle der Freispeicherverwaltung unterliegen. Sie könnten sonst ungewollt invalidiert werden.

Die TML-Evaluatoren (siehe Abschnitt 3.2) stellen die von ihnen benutzten Objektspeicherreferenzen durch eine dem Objektspeicher bekanntgemachte Enumerationsfunktion, die zum Zeitpunkt der *garbage collection* aufgerufen wird, zur Verfügung. Die Implementation der Enumerationsfunktion ist im allgemeinen stark von der Systemumgebung abhängig. Dazu zählen in diesem Fall die Hardware, das Betriebssystem und die Implementationsbasis des Tycoon-Systems (C-Übersetzer, Assembler etc.). Es ist stets das Traversieren der aktuellen Prozedurrahmen, gegebenenfalls sogar der Maschinenregister notwendig.

Alle semantischen Objekte der Sprachen TL und TML werden durch kanonische Abbildungen auf primitive Objektspeicherstrukturen realisiert.[7] Letztere werden ausschließlich unter Benutzung der wohldefinierten Software-Schnittstelle TSP manipuliert.

TL- und TML- Programme abstrahieren vollständig von den operationalen Qualitäten des Objektspeichers: Persistenz, Zugriffsgeschwindigkeit, nebenläufiger Zugriff, Speicherrückgewinnung, Fehlererholung und Verteilung. Damit leistet das Objektspeicherprotokoll einen wichtigen Beitrag zur Portabilität und Skalierbarkeit des Tycoon-Systems: Verschiedene TSP-Implementierungen können das gesamte Spektrum flüchtiger und persistenter Einbenutzer- oder verteilter Mehrbenutzersysteme abdecken. Für die vorliegende Arbeit ist folgender Aspekt wesentlich:

> Die Unterstützung der Persistenz durch die Kodegenerierung kann auf die Einhaltung des Objektspeicherprotokolls TSP reduziert werden.

Gemäß [Matthes 92] ist die im nachfolgenden Kapitel beschriebene Zwischensprache TML geeignet, diese Bedingung zu erfüllen. Der vorliegenden Arbeit obliegt nun der praktische Nachweis.

[7] Die einzige Ausnahme sind flüchtige lokale Variablen, die zu semantikerhaltenden Optimierungen verwendet werden.

Kapitel 3

Die portable untypisierte Zwischenkodesprache TML

Die Zwischensprache TML ist als Tycoon-Systemkomponente vorgegeben. Sie wird in [Matthes 92] definiert und beschrieben.

Ihre erneute formale Definition und ausführliche Darstellung in diesem Kapitel präzisiert die Aufgabenstellung und liefert die nötigen Beschreibungsmittel zur Darstellung der beiden Übersetzungen, an denen TML beteiligt ist: TL nach TML (siehe Kapitel 4) und TML nach C (siehe Kapitel 5).

Dieses Kapitel beginnt mit einer knappen Charakterisierung von TML gefolgt von einer Beschreibung der wesentlichen Konstrukte und Systemkomponenten, die bei der Ausführung von TML relevant sind. Auf der dabei geprägten Zusammenstellung von Begriffen baut die anschließende formale Definition der Eigenschaften von TML auf. Danach wird die Programmrepräsentation von TML vorgestellt, die Gegenstand der in den nachfolgenden Kapiteln beschriebenen Übersetzungen ist. Zu ihrer Illustration in der vorliegenden Arbeit wird dann außerdem noch eine spezielle graphische Darstellungsform eingeführt.

Aus Gründen der Einheitlichkeit stimmen die meisten Bezeichnungen mit denen in [Matthes 92] überein oder sind nur leicht verändert. Die nicht implementationsbezogenen Abschnitte (3.4, 3.5 und 3.6) und die in Abschnitt 3.1 folgende Aufzählung der Einsatzzwecke von TML sind eng an [Matthes 92] angelehnt, um den direkten Bezug herzustellen und die Ergebnisse der vorliegenden Arbeit nahtlos in das gesamte Tycoon-System einzuordnen.

3.1 Charakterisierung der Zwischensprache TML

Der als Resultat des Backends generierte TML-Kode basiert auf einem untypisierten Lambda-Kalkül mit Funktionsabschlüssen zur Unterstützung statischer Sichtbarkeitsregeln. Hinzu kommen imperative Kontrollstrukturen, Ausnahmebehandlungen und Zuweisungen.

Außerdem enthält der Kode einige wenige vordefinierte Funktionen (z.B. das Anlegen und der Identitätsvergleich von Speicherobjekten), die in einem minimalen Laufzeitsystem vorausgesetzt werden. Weitere externe Funktionen (zum Beispiel Arithmetik) können als Funktionsabschluß in den Kode eingebettet werden.

Wesentliche Merkmale sind die strikte Trennung in einen lokalen Evaluationszustand und einen globalen (typischerweise persistenten) Systemzustand sowie die Reduzierung der Typinformation auf ein einzelnes, uniformes Datenformat.

Der Kode wird als Baumstruktur in den Objektspeicher abgelegt.[1] Er kann direkt von einer abstrakten Maschine ausgeführt werden, die als portables Programm realisiert worden ist (siehe Abschnitt 3.3). Daher rührt auch der Name der Kodesprache: „Tycoon Machine Language“, kurz TML. Sie dient außerdem folgenden Zwecken [Matthes 92]:

- TML wird als Ausgangspunkt für die Maschinenkodegenerierung eingesetzt.
- TML soll sich als Zielkode für allgemeine Sprachen höherer Ordnung eignen und ist daher weitgehend unabhängig von speziellen Typaspekten der Sprache TL.
- TML wird als portable Programmrepräsentation zum Versenden von kompilierten und partiell gebundenen Programmen in heterogenen Netzwerken eingesetzt.
- TML muß bekannte statische Programmoptimierungstechniken wie die Elimination linearer Rekursionen, *inlining*-Techniken, die Elimination gemeinsamer Teilausdrücke oder die *strength reduction* [Waite, Goos 85], sowie die Transformation von Funktionen höherer Ordnung gestatten, wie sie in der algebraischen Anfrageoptimierung für relationale und objektorientierte Datenmodelle auftreten [Beeri, Kornatzky 90].
- Der Kode jeder TL-Funktion muß zur Laufzeit für reflektive Programmanalysen und -transformationen, wie sie zur dynamischen Anfrageoptimierung und vorausschauenden Transaktionssynchronisation erforderlich sind, zur Verfügung stehen können.
- TML ist die Grundlage der Definition der dynamischen Semantik von TL (siehe Abschnitt 3.6).

Eine lineare Repräsentation des Kodes wird in keinem der genannten Fälle benötigt.

3.2 Das TML-Maschinenmodell

Als Ausgangspunkt einer TML-Evaluation fungiert in jedem Fall ein Funktionsabschluß. Dadurch werden sowohl die abstrakten Schnittstellen als auch die der Implementation auf allen Systemebenen vereinheitlicht, ohne den Freiheitsgrad der Programmierung in TML einzuschränken. Denn eine beliebige Anweisung kann gegebenenfalls durch eine triviale Erweiterung in eine Funktion eingebettet werden.

Somit steht stets eine Funktion als Ausgangspunkt für die Zuordnung der in TML auftretenden Datenobjekte zur Verfügung. Diese lassen sich grob in die Kategorien „Werte“ und „Variablen“ einteilen.

[1] Genaugenommen kann es sich auch um gerichtete, azyklische Graphen handeln. Diese Verallgemeinerung ist jedoch ausschließlich durch Optimierungen zur Einsparung von Speicherplatz bedingt. Im folgenden ist weiterhin von „Bäumen“ die Rede, weil bei der Interpretation oder Übersetzung von TML-Kode die originären Eigenschaften von echten Bäumen im Vordergrund der Betrachtung stehen. Ob einzelne Knoten oder Subgraphen geteilt werden, ist nicht relevant, sofern keine Zyklen entstehen.

In TML werden zwei Arten von Werten unterschieden:

Einfache Werte (*immediate values*)**:** Eine andere, in der rein abstrakten Betrachtung zutreffende Bezeichnung wäre „atomare Werte“, weil weder eine andere Möglichkeit der expliziten Konstruktion einfacher Werte noch ein destruktiver Zugriff auf sie in TML möglich ist.
Beispiele: Ganzzahlen (Typ *Int*), Wahrheitswerte (Typ *Bool*).

Literale: Die zuletzt genannten Eigenschaften gelten für diese Werte nicht. Sie können theoretisch beliebig viel Speicherplatz einnehmen und auch beliebig komplex aufgebaut sein. Alle Literale einer Funktion werden in einem Vektor zusammengefaßt. Dieser „Literalvektor“ wird bereits zur Übersetzungszeit angelegt, wodurch die Programme zur Interpretation und zur weiteren Übersetzung von TML erheblich vereinfacht werden. Außerdem eröffnen sich Möglichkeiten zur Optimierung, weil Objekte mit potentiell hohem Speicherplatzbedarf gemeinsam erfaßt werden.
Beispiele: Zeichenketten (Typ *String*), geschachtelte Aggregate.

Wie schon in Abschnitt 2.3 werden weiterhin sowohl konstante als auch veränderliche Datenobjekte als Variablen bezeichnet. In TML existieren drei Arten von Variablen. Sie entsprechen den Bereichen, in denen Variablen in TL in Bezug auf eine Funktion deklariert sein können: lokale Variablen, Parameter und globale Variablen (siehe Abbildung 2.3 auf Seite 15).

Der Funktionsabschluß einer Funktion wird durch ein Speicherobjekt repräsentiert, das neben den globalen Variablen der Funktion eine Referenz auf ihren Literalvektor und einen optionalen Funktionszeiger für Maschinenkode enthält. Es besteht eine klare Trennung der statischen (Literalvektor und Kode) und dynamischen (Funktionsabschluß) Daten von Funktionen.

Zu den Literalen zählt insbesondere auch der TML-Kode des Funktionsrumpfes. Literalvektoren und TML-Kode werden bereits durch den TML-Kodegenerator im Objektspeicher angelegt und von allen Funktionsinstanzen (über Objektspeicherreferenzen) gemeinsam genutzt.[2] Die gemeinsame Nutzung trifft auch auf den optionalen Maschinenkode zu. Dabei kann es sich sowohl um durch weitere Übersetzung des TML-Kodes entstandenden Maschinenkode als auch um eine angebundene externe Funktion handeln.

Der zur Laufzeit in einen neu erzeugten Funktionsabschluß einzutragende Literalvektor wird dem Literalvektor der jeweiligen umgebenden Funktion entnommen. Dazu werden in diesen zur Übersetzungszeit bereits die Literalvektoren aller Funktionen der nächsttieferen Schachtelungsebene eingetragen.

Abbildung 3.1 zeigt den schematischen Aufbau eines Funktionsabschlusses und des dazugehörenden Literalvektors. Die Bezeichnungen korrespondieren mit denen der in Anhang A.2 aufgeführten Schnittstelle *TML*, die der Implementierung von TML dient (vgl. Abschnitt 3.7). Jedes Rechteck entspricht einem Maschinenwort.

Die abgebildete Repräsentationsform von Funktionsabschlüssen wird auch als *flat closure* [Appel 92] bezeichnet, weil alle freien Variablen des Funktionsrumpfes in einer flachen Speicherstruktur untergebracht sind. Im Gegensatz dazu bestehen *linked closures* aus verketteten Listen. Sie haben viele Gemeinsamkeiten mit *static links* und *access links* [Aho et al. 86], die z.B. in Pascal- und Algol-Implementation auftreten. Außer diesen beiden Extremen existieren

[2] Dies ist die wichtigste der bei der Beschreibung von Literalen erwähnten Optimierungen.

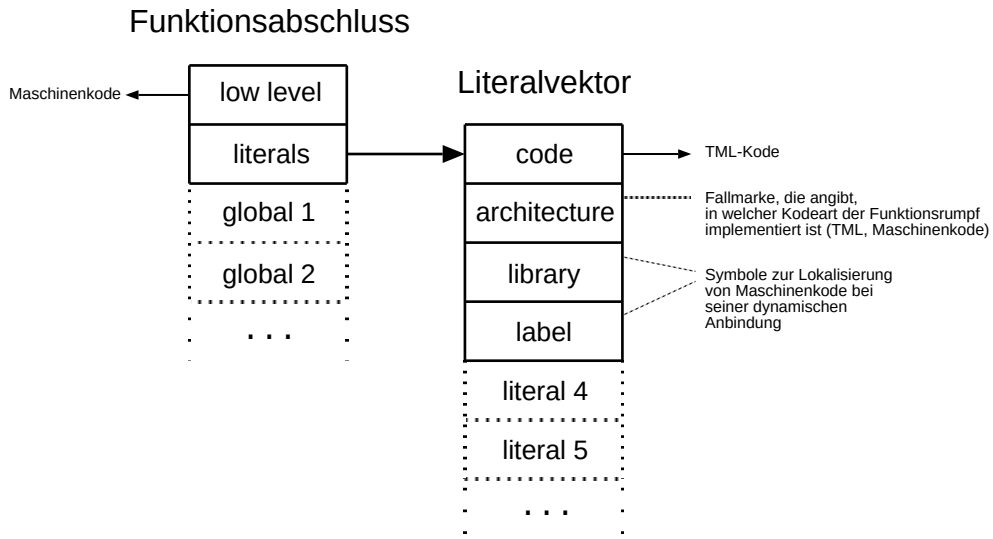

Abbildung 3.1: Der schematische Aufbau eines Funktionsabschlusses und seines Literalvektors

alle erdenklichen Mischformen [Appel, J. 88; Appel 92]. Aufgrund folgender Argumente sind *flat closures* für das Tycoon-System prädestiniert:

- Auf ihren Inhalt kann mit maximaler Effizienz zugegriffen werden. In jeder anderen Repräsentation ist ein mehrschrittiges Verfolgen von Referenzen unvermeidlich. Dies ist in einem persistenten System besonders ineffizient.
- Die Speicherstrukturen aller Funktionsabschlüsse sind paarweise disjunkt. Wenn dagegen Unterstrukturen geteilt werden, dann referenzieren Funktionsabschlüsse nicht selten Daten, die für ihre eigene Exekution irrelevant sind. In Folge dessen entstehen schwerwiegende Probleme:
  - Beim Verwerfen eines Funktionsabschlusses bleiben häufig ungenutzte Teile in von anderen Funktionsabschlüssen referenzierten Unterstrukturen erfaßt [Appel 92]. Dadurch können große Datenmengen ihren Speicherplatz weiterhin okkupieren, ohne von der Freispeicherverwaltung (*garbage collection*) als überflüssig erkannt zu werden. Dieses irreversible Wachstum ist in persistenten Systemen inakzeptabel.
  - Bei der Netzwerkübertragung eines Funktionsabschlusses, der Unterstrukturen mit anderen Funktionsabschlüssen teilt, werden von diesen referenzierte Daten ebenfalls versandt, obwohl sie auf der Empfängerseite nicht benötigt werden.

  Beide Probleme können durch umfassende Analysen behoben werden. Das dazu erforderliche Traversieren der im Speicher verteilten Bestandteile von Funktionsabschlüssen ist jedoch in persistenten Systemen besonders ineffizient.

Der wichtigste Nachteil der *flat closures* besteht darin, daß sie von jedem gekapselten Wert eine Kopie anlegen, was sich in zweierlei Hinsicht auswirkt:

- Sie benötigen relativ viel Speicherplatz durch die wiederholte Aggregation der gleichen Werte. Da die weiteren Indirektionsstufen der Werte davon jedoch nicht betroffen sind, hängt die Speichergröße eines Funktionsabschlusses ausschließlich von der *Anzahl* der globalen Werte ab. Letztere kann nur bei umfangreichen Funktionen hoch sein. Funktionen mit mehreren Instanzen sind jedoch meist klein.
- Ihre Initialisierung ist relativ zeitaufwendig. Dies wird jedoch durch die erhöhte Effizienz ihrer Ausführung in aller Regel mehr als ausgeglichen.

Zusammenfassend wird festgestellt, daß die Nachteile der *flat closures* im vorliegenden Fall wesentlich weniger schwerwiegend sind als die der anderen Arten von Funktionsabschlüssen.

Die Evaluationssemantik für TML-Programme wird durch Zustandsübergänge zwischen abstrakten Konfigurationen beschrieben. Die Bestandteile einer Konfiguration sind:

- ein globaler Objektspeicher,
- ein Funktionsabschluß, der als Vektor globaler Variablen genutzt wird,
- eine Referenz auf einen Literalvektor,
- ein Parametervektor,
- ein Vektor lokaler Variablen,
- eine aktuell auszuführende TML-Instruktion.

Funktionsaufrufe führen zum Anlegen neuer Parametervektoren und lokaler Variablen. Während der Evaluation einer Funktion sind die Parameter und die lokalen Variablen anderer Funktionen nicht zugreifbar. Darin besteht die oben bereits erwähnte strikte Trennung zwischen lokalem Evaluationszustand und globalem Systemzustand.

Die Tycoon-Architektur erlaubt mehrere Aktivierungen (*threads*) von TML-Evaluatoren innerhalb des gleichen Betriebssystemprozesses. Jeder Evaluator besitzt seine eigene Konfiguration. Alle Evaluatoren greifen über eine abstrakt definierte Objektspeicherschnittstelle auf den Objektspeicher zu. Der Objektspeicher ist für die globale Speicherverwaltung (Speicherfreigabe, Parallelitätskontrolle, Fehlererholung) und die Kohärenz zwischen den Objekten verschiedener Evaluatoren innerhalb eines oder mehrerer Prozesse verantwortlich. Zur Effizienzsteigerung kann ein Evaluator explizit veranlassen, daß Speicherobjekte vorübergehend im lokalen Prozeßspeicher fixiert (siehe Abschnitt 3.5) werden. Die Freigabe erfolgt ebenfalls explizit. Der Objektspeicher kann logisch oder physisch in Partitionen (*repositories*) gegliedert sein. Ein Evaluator kann die Unterbringung eines Objektes in einer bestimmten Partition bei der Objektgenerierung über einen (nicht näher spezifizierten) Lokalitätswert angeben.

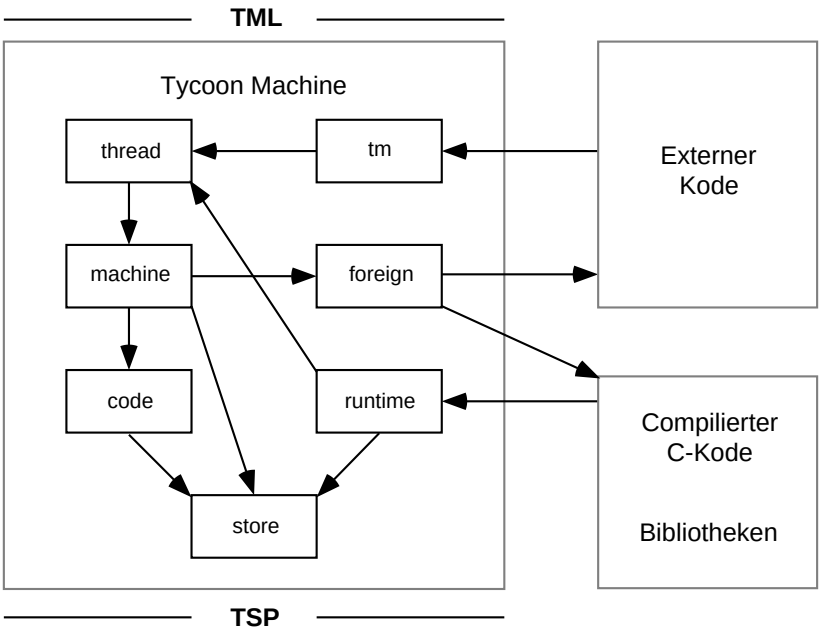

Abbildung 3.2: Die Modulstruktur des TML-Interpreters

3.3 Eine portable abstrakte Maschine zur Interpretation von TML

Das im vorangehenden Abschnitt beschriebene Maschinenmodell ist in einem portablen Interpreterprogramm realisiert worden. Wie aus Abbildung 2.2 auf Seite 9 ersichtlich, ist es ein integraler Bestandteil des Tycoon-Laufzeitsystems.

Abbildung 3.2 zeigt schematisch seine Modulstruktur. Die Module des Interpreters haben folgende Aufgaben:

thread: die Verwaltung dynamischer Kontexte und Ausnahmebehandlungen.

machine: die Interpretation von TML-Instruktionen.

code: die Repräsentation von TML-Instruktionen, Literalvektoren und Funktionsabschlüssen.

store: die Implementation des Objektspeichers gemäß TSP (siehe Abschnitt 2.4).

tm: die Programmierschnittstelle zur Benutzung des Interpreters durch externe Programme.

runtime: die Schnittstelle des Laufzeitsystems zur Unterstützung generierten C-Kodes.

foreign: Bindung und generischer Aufruf externer Funktionen.

Als „extern'" gelten alle nicht in TML implementierten Funktionen, die keinem der TML-Laufzeitsystemkodes zugeordnet sind. Die 14 Laufzeitsystemkodes werden in Anhang A.2

durch den Typ *RuntimeCode* aufgezählt. Sie unterstützen die Ausführung der wenigen vordefinierten TL-Konstrukte. Zum Beispiel dient der Laufzeitsystemkode *valueEqual* dem Vergleich zweier Werte und *exceptionCaseCrashValue* liefert den Ausnahmewert, der im Falle des Scheiterns einer Fallunterscheidung benötigt wird.

Der Kern des Tycoon-Systems ist sehr kompakt. Externe Funktionen müssen durch die Funktion *foreign_bind*, die dem Laufzeitsystemkode *machineBind* entspricht, angebunden werden.

Viele C-Funktionen des Interpreters, insbesondere diejenigen, welche Operationen auf TL-Basistypen wie z.B. *Bool*, *Int* und *String* durchführen, verwenden das Parameterschema generierten C-Kodes (siehe Abschnitt 5.4.1). Dadurch können sie direkt zur Implementation der entsprechenden Tycoon-Standardbibliotheken eingesetzt werden, was den Aufwand für die Erstimplementierung von Tycoon mindert. Jede Verringerung der Mindestgröße des Systems vergrößert zudem seinen potentiellen Einsatzbereich.

3.4 Eine abstrakte Syntax für TML

Die in diesem Abschnitt definierte abstrakte Syntax von TML [Matthes 92] wird in Abschnitt 3.6 zur Beschreibung der operationalen Semantik verwendet.

Ihre syntaktischen Elemente stammen aus folgenden Mengen:

$\mathcal{N}$	die Natürlichen Zahlen, einschließlich der Null.
$Code$	die syntaktisch zulässigen TML-Instruktionen
$BVal = \mathcal{N} \cup \{nil\}$	Basiswerte ($nil \notin \mathcal{N}$)
$BFun$	vordefinierte Funktionen (*builtins*)

Sie werden durch (eventuell indizierte) Variablen bezeichnet, für die folgende Konventionen gelten: $c \in Code$; $i, j, k, m, n \in \mathcal{N}$; $b \in BVal$; $op \in BFun$

Die Menge $Code$ ist durch die Regeln in Tabelle 3.3 induktiv definiert, wobei die Basiswerte und Basisfunktionen als Ausgangspunkt dienen.

Für die Bildung *wohlgeformter* TML-Anweisungen gelten außerdem einige zusätzliche Einschränkungen:

- Die Anzahl der Argumente eines Aufrufes eines Funktionsabschlusses oder einer vordefinierten Funktion muß der Anzahl der Parameter entsprechen.
- Jede **exit** -Instruktion muß sich innerhalb einer **loop** -Anweisung befinden. In der Schachtelung zwischen zwei derartig in Bezug zueinander stehenden Instruktionen darf keine Funktionsabstraktion auftreten.
- Alle Fallanalysen müssen wohlgeformt sein. In einer Anweisung

  $$\textbf{alt}\, c_0\, \textbf{of}\, b_1 \rightarrow i_1 \ldots b_m \rightarrow i_m\, \textbf{do}\, c_1 \ldots c_n\, \textbf{else}\, c_{n+1}$$

  muß für alle $1 \leq j, k \leq m$ gelten:

  $$b_j \neq b_k \quad (j \neq k)$$

  $$1 \leq i_j \leq n$$

c ::=	**nop**	Leere Anweisung
|	**immediate**(b)	Basiswertkonstante
|	**literal**$_i$	Literalkonstante
|	**local**$_i$	Wert einer lokalen Variablen
|	**parameter**$_i$	Parameterwert
|	**global**$_i$	Globaler Wert
|	$c_1[c_2]$	Wert aus dem Objektspeicher
|	**local**$_i \leftarrow c_1$	Zuweisung einer lokalen Variablen
|	**global**$_i\, c_1 \leftarrow c_2$	Zuweisung eines globalen Wertes
|	$c_1[c_2] \leftarrow c_3$	Zuweisung im Objektspeicher
|	$\nabla_n\, c_1\, c_2$	Anlegen eines Funktionsabschlusses
|	$\lambda\, n\, c_1$	Funktionsrumpf
|	$c_0(c_1 \ldots c_n)$	Funktionsapplikation
|	$c_1\,;\, c_2$	Sequentielle Auswertung
|	**loop** c_1	Schleife
|	**exit** c_1	Schleifenabbruch
|	**trap** c_1 **with** l_i **do** c_2	Ausnahmebehandlung
|	**raise** c_1	Erzeugen einer Ausnahme
|	**runtime**(op)($c_1 \ldots c_n$)	Aufruf einer vordefinierten Funktion
|	**alt** c_0 **of** $tags$ **do** $c_1\ \ldots\ c_n$ **else** c_{n+1}	Fallanalyse
;		
$tags$ ::=	$b_1 \rightarrow i_1\ \ldots\ b_n \rightarrow i_n$	Fallmarken
;		

Abbildung 3.3: Abstrakte Syntax für TML-Instruktionen [Matthes 92]

Bei der isolierten Betrachtung von TML steht die Vordefiniertheit der „*builtins*“ im Vordergrund, weshalb diese Bezeichnung in diesem Zusammenhang adäquat ist. In der vorliegenden Arbeit treten jedoch weitere vordefinierte Funktionen auf, die als externe Routinen angebunden sind. Zur Differenzierung wird abweichend von [Matthes 92] die Bezeichnung „*runtime*“ verwendet. Sie bezieht sich darauf, daß die TML-*builtins* generierten Maschinenkode als Laufzeitsystemfunktionen unterstützen. Außerdem spielen sie bei der Interpretation von TML eine analoge Rolle: Sie sind nicht Teil des eigentlichen Kodes.

3.5 Die Semantik der Objektspeicheroperationen

Die Kapselung des Objektspeichers durch seine abstrakte Schnittstelle ermöglicht die Isolierung des globalen Systemzustandes, der durch den Objektspeicher gegeben ist, von lokalen Evaluatorzuständen. Daher kann der Objektspeicher zunächst als unabhängiges System betrachtet werden. Die Beschreibung der Semantik seiner Operationen [Matthes 92] dient im nachfolgenden Abschnitt als Grundlage für die Definition der Semantik von TML-Instruktionen, die den Objektspeicher betreffen.

Zur Beschreibung der Semantik des Objektspeichers werden folgende Notationen verwendet:

$A \xrightarrow{fin} B$	Die Menge der endlichen Abbildungen von A nach B
$\{\}$	Die leere Abbildung
$\{a \mapsto b\}$	Eine Abbildungsdefinition
$f(a)$	Die Applikation von $f \in A \xrightarrow{fin} B$ auf $a \in A$
$Dom(f)$	Der Definitionsbereich von $f \in A \xrightarrow{fin} B$
$Ran(f)$	Der Bildbereich von $f \in A \xrightarrow{fin} B$

Für zwei Abbildungen $f \in A \xrightarrow{fin} B$ und $g \in C \xrightarrow{fin} D$ ist ihre Modifikation f + g definiert durch:

$$(f+g)(a) = \begin{cases} g(a) & \text{falls } a \in Dom(g) = C \\ f(a) & \text{sonst} \end{cases}$$

Dabei gilt:

$$Dom(f+g) = Dom(f) \cup Dom(g) = A \cup C$$

$$Ran(f+g) = Ran(f) \cup Ran(g) = B \cup D$$

Sei ferner Ref die Menge der Objektreferenzen und $SVal = BVal \cup Ref$ die Menge der Zustandswerte (state values), wobei vorausgesetzt wird, daß $BVal$ und Ref disjunkt sind. Als Zustandswerte können auch Werte außerhalb des Objektspeichers angesehen werden. Für die Definition eines Objektspeichers sind allerdings nur Referenzen relevant.

Jeder Referenz wird zusätzlich eine TML-Instruktion zugeordnet, um den Übergang zur Interpretation der Daten als Kode zu modellieren.[3] Außerdem wird die Länge der referenzierten Objektvektoren erfaßt, die den Indexbereich für Zugriffsoperationen explizit festlegt.

[3] Dieser formale Trick dient der Beschreibung des Umstandes, daß die Operation *store.fixExecute* aus einem Funktionsabschluß den Kode des Funktionsrumpfes extrahiert, ohne auf Einzelheiten der Speicherstrukturen eingehen zu müssen. Den Referenzen, die nicht auf einen Funktionsabschluß verweisen, mögen beliebige TML-Instruktionen zugeordnet sein, denn letztere sind von keiner Operation betroffen.

Die Menge $Store$ der möglichen Objektspeicher ist wie folgt definiert:

$$Store = (Ref \times \mathcal{N} \xrightarrow{fin} SVal) \times (Ref \xrightarrow{fin} \mathcal{N}) \times (Ref \xrightarrow{fin} \mathcal{N})$$

Ein leerer initialer Objektspeicher wird durch folgende Operation angelegt:

$$store.init = (\{\}, \{\}, \{\})$$

Für alle weiteren Definitionen der Semantik von Objektspeicheroperationen gelten die Konventionen:

$$\begin{gathered} v, v' \in Ref \times \mathcal{N} \xrightarrow{fin} SVal \\ c, c' \in Ref \xrightarrow{fin} \mathcal{N} \\ s, s' \in Ref \xrightarrow{fin} \mathcal{N} \\ l, r \in Ref \\ x \in SVal \\ g, i, n \in \mathcal{N} \end{gathered}$$

Der Erzeugung eines neuen Objektes in Form eines Zustandswertvektors, der mit *nil* initialisiert wird, entspricht:

$$store.new((v, c, s), n, =)(v', c, s')$$

Dabei wird durch n die Länge des Objektvektors angegeben. Sei r die neue, eindeutige Referenz, dann gilt:

$$\begin{aligned} r &\notin Dom(v) \\ v' &= v + \{(r,0) \mapsto nil\} + \{(r,1) \mapsto nil\} + \ldots + \{(r,n-1) \mapsto nil\} \\ s' &= s + \{r \mapsto n\} \end{aligned}$$

Den i-ten Zustandswert eines Objektes r liefert:

$$store.get((v, c, s), r, i) = v((r, i))$$

Für diesen und den folgenden Fall gilt die Randbedingung $0 \leq i < s(r)$. Die Operation

$$store.set((v, c, s), r, i, x) = (v + \{(r, i) \mapsto x\}, c, s)$$

ersetzt den i-ten Zustandswert eines Objektes r durch x.

Zum Anlegen von Funktionsabschlüssen dient folgende Operation:

$$store.newClosure((v, c, s), g, c, l)((v', c', s'), r)$$

Dabei gibt g die Anzahl der globalen Einträge und l die Referenz des Literalvektors an (vgl. Abschnitt 3.2) und es gilt:[4]

$$\begin{aligned} r &\notin Dom(v) \\ v' &= v + \{(r,0) \mapsto nil\} + \{(r,1) \mapsto l\} + \{(r,2) \mapsto nil\} + \{(r,3) \mapsto nil\} + \ldots + \{(r,g+1) \mapsto nil\} \\ s' &= s + \{r \mapsto g+2\} \end{aligned}$$

[4] Der Wert an der Position 0 ist für die Anbindung externen Kodes an Funktionsabschlüsse reserviert. Weitere Implementationsdetails werden in diesem Zusammenhang vernachlässigt.

Schließlich extrahiert folgende Operation die relevanten Daten für die Ausführung einer durch ein Objekt r gegebenen Funktion:

$$store.fixExecute((v, c, s), r) =$$

$$(c(r), v((r,1)), v((r,2)), v((r,3)), \ldots, v((r, s(r)-1)))$$

Dabei handelt es sich um den Funktionskode, den Literalvektor und die globalen Einträge.

Die Schnittstelle des Objektspeichers enthält noch weitere Operationen, jedoch genügen die hier beschriebenen zur Definition der Semantik von TML.

3.6 Die strukturelle operationale Semantik von TML

Der vorangehende Abschnitt bezieht sich auf den globalen Systemzustand. Zur Modellierung lokaler (nicht persistenter) Evaluationszustände werden weitere semantische Objekte benötigt, die hiermit gleich im Zusammenhang mit den dazugehörigen Mengen vereinbart werden:

$E = [lit\ g_0 \ldots g_n\ p_0 \ldots p_m] \in Env$	Dynamische Kontexte
$L = [l_0 \ldots l_k] \in Loc$	Lokale Variablenvektoren
$v \in Val = SVal \cup \{ok, exception(p), exit(p)\}; p \in SVal$	Auswertungsergebnisse

Ein dynamischer Kontext (E) enthält den Literalvektor, die globalen Werte des Funktionsabschlusses und die Parameterwerte einer auszuführenden Funktion.

Die lokalen Variablen (eines Vektors L) einer Funktion sind nicht Teil des globalen Systemzustandes und nur für eine einzige Funktion sichtbar.

Der ausgezeichnete Wert *ok* beschreibt das Ergebnis von Operationen, die ausschließlich der Erzielung eines Seiteneffektes dienen (z.B. Zuweisungen). $exception(p)$ steht für Ausnahmepakete mit einem zusätzlichen Wert $p \in SVal$. Sie können durch vordefinierte Funktionen (*builtins*) oder durch die Anweisung **raise** p erzeugt werden. Dazu analog erzeugt die **exit** p-Instruktion Exit-Pakete der Form $exit(p)$. Im Unterschied zu TL kann eine TML-Schleife zu einem Wert $p \neq ok$ evaluieren.

Die vorangehenden Vereinbarungen erlauben nun die Definition der TML-Semantik [Matthes 92]. Dabei wird der von [Plotkin 81] als *structural operational semantics* bezeichnete Formalismus verwendet. Zu seiner Namensgebung führte der zugrundeliegende Ansatz:

> Die Semantik einer zusammengesetzten Instruktion wird induktiv durch die Semantik ihrer Konstituenten definiert.

Es werden Axiome und Deduktionsregeln angegeben. Alle Regeln definieren eine strikte, deterministisch sequentielle Evaluation. Die Evaluationsreihenfolge innerhalb einer Regel entspricht der lexikalischen Anordnung. Daraus ergibt sich auch die Evaluationsreihenfolge zusammengesetzter Instruktionen. Sie ist eindeutig, weil für *jede* TML-Instruktion (siehe Tabelle 3.3 auf Seite 28) eine eigene Regel angegeben wird.

Sowohl die Aussagen der Prämissen als auch die der Konklusionen besitzen dabei folgende Form:

$$E, S_1, L_1 \vdash c \Rightarrow \langle S_2, L_2, v \rangle$$

Obige Zeile besagt, daß die Ausführung der (eventuell zusammengesetzten) Instruktion c in dem dynamischen Kontext E, im Objektspeicherzustand S_1 und mit den lokalen Variablenwerten L_1 zu einem Objektspeicherzustand S_2, lokalen Variablenwerten L_2 und einem Auswertungsergebnis v führt. Eine TML-Instruktion kann also ihren dynamischen Kontext E nicht modifizieren.

Zuerst werden die Regeln der Instruktionen aufgeführt, die nur lesend auf den lokalen Evaluationszustand zugreifen:

[Eval nop]

$$\frac{}{E, S, L \vdash \mathbf{nop} \Rightarrow \langle S, L, ok \rangle}$$

[Eval immediate]

$$\frac{}{E, S, L \vdash \mathbf{immediate}(b) \Rightarrow \langle S, L, b \rangle}$$

[Eval literal]

$$\frac{}{[lit\ g_0 \ldots g_n\ p_0 \ldots p_m], S, L \vdash \mathbf{literal}_i \Rightarrow \langle S, L, store.get(S, l, i) \rangle}$$

[Eval get local]

$$\frac{}{E, S, [l_0 \ldots l_k] \vdash \mathbf{local}_i \Rightarrow \langle S, l_0 \ldots l_k, l_i \rangle}$$

[Eval get param]

$$\frac{}{[lit\ g_0 \ldots g_n\ p_0 \ldots p_m], S, L \vdash \mathbf{parameter}_i \Rightarrow \langle S, L, p_i \rangle}$$

[Eval get global]

$$\frac{}{[lit\ g_0 \ldots g_n\ p_0 \ldots p_m], S, L \vdash \mathbf{global}_i \Rightarrow \langle S, L, g_i \rangle}$$

[Eval get indexed]

$$\frac{\begin{array}{c} E, S_1, L_1 \vdash c_1 \Rightarrow \langle S_2, L_2, r \rangle \\ E, S_2, L_2 \vdash c_2 \Rightarrow \langle S_3, L_3, i \rangle \end{array}}{E, S_1, L_1 \vdash c_1[c_2] \Rightarrow \langle S_3, L_3, store.get(S_3, r, i) \rangle}$$

Der Index i und die Referenz r in der Regel [Eval get indexed] sind Beispiele für Instruktionen, die einem beliebig komplexen TML-Ausdruck entsprechen können.

Es folgt die Semantik destruktiver Zuweisungen:

[Eval set local]

$$\frac{E, S_1, [l_0 \ldots l_k] \vdash c \Rightarrow \langle S_2, L_2, l\rangle}{E, S_1, [l_0 \ldots l_k] \vdash \mathbf{local}_i \leftarrow c \Rightarrow \langle S_2, [l_0 \ldots l_{i-1}\, l\, l_{i+1} \ldots l_k], ok\rangle}$$

[Eval set indexed]

$$\frac{\begin{array}{c}E, S_1, L_1 \vdash c_1 \Rightarrow \langle S_2, L_2, r\rangle \\ E, S_2, L_2 \vdash c_2 \Rightarrow \langle S_3, L_3, i\rangle \\ E, S_3, L_3 \vdash c_3 \Rightarrow \langle S_4, L_4, x\rangle\end{array}}{E, S_1, L_1 \vdash c_1[c_2] \leftarrow c_3 \Rightarrow \langle store.set(S_4, r, i, x), L_4, ok\rangle}$$

Funktionsabschlüsse sind in TML komplexe Objekte, da auch geschachtelte Funktionen höherer Ordnung (aus der Sicht einer höheren Sprache wie z.B. TL, die nach TML übersetzt wird) implementiert werden. Ihre Erzeugung [Eval abstract] und Initialisierung [Eval set global] wird jeweils durch eigene Instruktionen geleistet. Durch diese Trennung ist unter anderem für die Übersetzung typsicher gebundener rekursiver Funktionen nach TML notwendig (siehe Abschnitt 4.4.7).

In der folgenden Regel steht n für die Größe des zu erzeugenden Funktionsabschlusses, l für die Referenz des Literalvektors und c_2 für den Kode des Funktionsrumpfes.

[Eval abstract]

$$\frac{\begin{array}{c}E, S_1, L_1 \vdash c_1 \Rightarrow \langle S_2, L_2, l\rangle \\ store.newClosure(S_2, n, c_2, l)(S_3, r)\end{array}}{E, S_1, L_1 \vdash \nabla_n\, c_1\, c_2 \Rightarrow \langle S_3, L_2, r\rangle}$$

Die nachstehende Regel verdeutlicht, daß die Semantik der Operation $\mathbf{global}_i c_1 \leftarrow c_2$ auf die der Instruktion $c_1[\mathbf{immediate}(i+2)] \leftarrow c_2$ zurückgeführt werden kann. Eine eigene Instruktion für den Zweck der Initialisierung von Funktionsabschlüssen ist dennoch angebracht, weil diese im Gegensatz zu anderen Speicherobjekten anschließend nicht mehr verändert werden können. Diese Zusicherung kann von Programmen ausgenutzt werden, die über TML reflektieren und z.B. Synchronisations- oder Fehlererholungsmaßnahmen vorbereiten oder durchführen.

[Eval set global]

$$\frac{E, S_1, L_1 \vdash c_1[\mathbf{immediate}(i+2)] \leftarrow c_2 \Rightarrow \langle S_2, L_2, x\rangle}{E, S_1, L_1 \vdash \mathbf{global}_i c_1 \leftarrow c_2 \Rightarrow \langle S_2, L_2, x\rangle}$$

Die Regel für Funktionsapplikationen liest sich wie folgt: Durch die Instruktion c wird eine Referenz r berechnet, die einen Funktionsabschluß mit Kode c', Literale l und Werte globaler Variablen $g_0 \ldots g_n$ identifiziert. Es folgt die Evaluation der Aktualparameterliste $p_0 \ldots p_m$. Dann wird der Kode c' in einem neuen dynamischen Kontext ($l, g_0 \ldots g_n$ und $p_0 \ldots p_m$) evaluiert. Schließlich wird der dynamische Kontext der aufrufenden Funktion wiederhergestellt. Dieser entspricht der Situation nach der Berechnung des letzten Argumentes (pm).

[Eval apply]

$$\frac{\begin{array}{c}E, S_1, L_1 \vdash c \Rightarrow \langle S_2, L_2, r\rangle \\ store.fixExecute(S_2, r) = (c', l, g_0, \ldots, g_n) \\ E, S_{i+2}, L_{i+2} \vdash c_i \Rightarrow \langle S_{i+3}, L_{i+3}, p_i\rangle \quad i = 0 \ldots m \\ [lit\ g_0 \ldots g_n\ p_0 \ldots p_m], S_{m+3}, L_{m+3} \vdash c' \Rightarrow \langle S, L_{m+2}, x\rangle\end{array}}{E, S_1, L_1 \vdash c(c_0 \ldots c_m) \Rightarrow \langle S, L_{m+2}, x\rangle}$$

Eine aufgerufene Funktion alloziert gegebenenfalls durch die Instruktion $\lambda\, k\, c$ lokale Variablen und initialisiert sie mit *nil*. Sie werden nach der Evaluation des Funktionsrumpfes c wieder verworfen.

[Eval lambda]

$$\frac{E, S_1, [\overbrace{nil \ldots nil}^{k}] \vdash c \Rightarrow \langle S_2, L_2, x\rangle}{E, S_1, L_1 \vdash \lambda\, k\, c \Rightarrow \langle S_2, L_1, x\rangle}$$

Da TML untypisiert ist, ist die Signatur einer Operation $op \in BFun$ durch ihre Arität (Argumentanzahl) $arity(op)$ eindeutig festgelegt.

$$arity : BFun \xrightarrow{fin} \mathcal{N}$$

Die Details der Interaktion zwischen extern definierter Funktionalität und TML-Instruktionen können dann durch eine indizierte Funktion *apply* beschrieben werden:

$$apply_{op} : Store \times SVal^{arity(op)} \to Store \times Val$$

Sie definiert, wie die Operation op anhand ihrer Argumente einen Ergebniswert oder ein Ausnahmepaket berechnet. Dabei kann lesend und schreibend auf den Objektspeicher zugegriffen werden.

In der Konklusion ihrer Regel entspricht die Applikation extern definierter Funktionen jedoch der von TML-Funktionen, so daß beide gleichberechtigt einsetzbar sind.

[Eval apply runtime]

$$\frac{\begin{array}{c}E, S_{i+1}, L_{i+1} \vdash c_i \Rightarrow \langle S_{i+2}, L_{i+2}, p_i\rangle \quad i = 0 \ldots m \\ apply_{op}(S_{m+2}, p_0, \ldots, p_m) = (S, x)\end{array}}{E, S_1, L_1 \vdash \mathbf{runtime}(op)(c_0 \ldots c_m) \Rightarrow \langle S, L_{m+2}, x\rangle}$$

Das Evaluationsergebnis von Sequenzen wird allein durch die zweite Instruktion bestimmt. Die erste Instruktion ist nur durch ihre Seiteneffekte relevant.

[Eval seq]

$$\frac{\begin{array}{c}E, S_1, L_1 \vdash c_1 \Rightarrow \langle S_2, L_2, x\rangle \\ E, S_2, L_2 \vdash c_2 \Rightarrow \langle S_3, L_3, x'\rangle\end{array}}{E, S_1, L_1 \vdash c_1\,;c_2 \Rightarrow \langle S_3, L_3, x'\rangle}$$

Fallanalysen werden durch zwei komplementäre Regeln erfaßt. Die erste beschreibt erfolgreiche und die zweite erfolglose Fallmarkentests.

[Eval alt]

$$\frac{E, S_1, L_1 \vdash c \Rightarrow \langle S_2, L_2, b_j \rangle \quad E, S_2, L_2 \vdash c_{i_j} \Rightarrow \langle S_3, L_3, x \rangle}{E, S_1, L_1 \vdash \textbf{alt}\, c\, \textbf{of}\, b_1 \rightarrow i_1 \ldots b_n \rightarrow i_n\, \textbf{do}\, c_1 \ldots c_m\, \textbf{else}\, c' \Rightarrow \langle S_3, L_3, x \rangle}$$

[Eval alt else]

$$\frac{E, S_1, L_1 \vdash c \Rightarrow \langle S_2, L_2, x \rangle \quad \forall 1 \leq j \leq n : x \neq b_j \quad E, S_2, L_2 \vdash c' \Rightarrow \langle S_3, L_3, x' \rangle}{E, S_1, L_1 \vdash \textbf{alt}\, c\, \textbf{of}\, b_1 \rightarrow i_1 \ldots b_n \rightarrow i_n\, \textbf{do}\, c_1 \ldots c_m\, \textbf{else}\, c' \Rightarrow \langle S_3, L_3, x' \rangle}$$

Eine **exit**-Instruktion erzeugt ein Exit-Paket, in dem das Ergebnis der Evaluation von c enthalten ist.

[Eval exit]

$$\frac{E, S_1, L_1 \vdash c \Rightarrow \langle S_2, L_2, p \rangle}{E, S_1, L_1 \vdash \textbf{exit}\, c \Rightarrow \langle S_2, L_2, exit(p) \rangle}$$

Der Wiederholung einer Schleife und ihrem Verlassen bei gleichzeitiger Evaluation zu einem Exit-Paket entsprechen zwei komplementäre Regeln:

[Eval loop]

$$\frac{E, S_1, L_1 \vdash c \Rightarrow \langle S_2, L_2, x_1 \rangle \quad x_1 \in SVal \cup \{ok\} \quad E, S_2, L_2 \vdash \textbf{loop}\, c \Rightarrow \langle S_3, L_3, x_2 \rangle}{E, S_1, L_1 \vdash \textbf{loop}\, c \Rightarrow \langle S_3, L_3, x_2 \rangle}$$

[Eval loop exit]

$$\frac{E, S_1, L_1 \vdash c \Rightarrow \langle S_2, L_2, exit(p) \rangle}{E, S_1, L_1 \vdash \textbf{loop}\, c \Rightarrow \langle S_2, L_2, p \rangle}$$

Die Regeln für Ausnahmen und ihre Behandlung enthalten viele Analogien zu den drei vorangehenden. Es handelt sich wiederum um die Bildung eines Paketes und zwei komplementäre Evaluationsregeln:

[Eval raise]

$$\frac{E, S_1, L_1 \vdash c \Rightarrow \langle S_2, L_2, p \rangle}{E, S_1, L_1 \vdash \textbf{raise}\, c \Rightarrow \langle S_2, L_2, exception(p) \rangle}$$

[Eval trap]

$$\frac{E, S_1, L_1 \vdash c_1 \Rightarrow \langle S_2, L_2, v\rangle \quad v \neq \mathit{exception}(p)}{E, S_1, L_1 \vdash \mathbf{trap}\, c_1\, \mathbf{with}\, l_i\, \mathbf{do}\, c_2 \Rightarrow \langle S_2, L_2, v\rangle}$$

[Eval trap exception]

$$\frac{E, S_1, L_1 \vdash c_1 \Rightarrow \langle S_2, [l_0 \ldots l_k], \mathit{exception}(p)\rangle \quad E, S_2, [l_0 \ldots l_{i-1}\, p\, l_{i+1} \ldots l_k] \vdash c_2 \Rightarrow \langle S_3, L_3, v_2\rangle}{E, S_1, L_1 \vdash \mathbf{trap}\, c_1\, \mathbf{with}\, l_i\, \mathbf{do}\, c_2 \Rightarrow \langle S_3, L_3, v_2\rangle}$$

Um die Semantik der Propagierung von Ausnahme- und Exit-Paketen in zusammengesetzten Instruktionen kompakt zu definieren, wird folgende Konvention (vgl. [Milner et al. 90]) eingeführt: Für jede Deduktionsregel mit n Prämissen (unter Nichtbeachtung von Seitenbedingungen) der Gestalt

$$\frac{E, S_1, L_1 \vdash c_1 \Rightarrow \langle S'_1, L'_1, x_1\rangle \quad \ldots \quad E, S_n, L_n \vdash c_n \Rightarrow \langle S'_n, L'_n, x_n\rangle}{E, S, L \vdash c \Rightarrow \langle S', L', v'\rangle}$$

und für jedes $k, 1 \leq k \leq n$, für das das Ergebnis x_k kein Exit- oder Ausnahmepaket ist, ist eine zusätzliche Regel der nachstehenden Form einzufügen.

$$\frac{E, S_1, L_1 \vdash c_1 \Rightarrow \langle S'_1, L'_1, x_1\rangle \quad \ldots \quad E, S_k, L_k \vdash c_k \Rightarrow \langle S'_k, L'_k, x_k\rangle \quad x_k = \mathit{exit}(p) \vee x_k = \mathit{exception}(p)}{E, S, L \vdash c \Rightarrow \langle S'_k, L'_k, x_k\rangle}$$

Diese Darstellung modelliert den Abbruch der Evaluation zusammengesetzter Instruktionen, in deren Konstituenten ein Ausnahme- oder Exit-Paket x_k auftritt. Dieses wird gegebenenfalls bis zu den Prämissen der Regeln [Eval loop exit] und [Eval trap exception] propagiert und dort explizit behandelt.

3.7 Die Implementation der Zwischensprache TML

Wie bereits in Abschnitt 3.1 erwähnt, wird außer zu Darstellungszwecken keine lineare Repräsentation von TML benötigt. Das Tycoon-System benutzt TML sowohl bei der Interpretation (siehe Abschnitt 3.3) als auch bei der weiteren Übersetzung (siehe Kapitel 5) nur in Form von direkt in den Objektspeicher abgelegten Baumstrukturen.

Das Modul *tml:TML* implementiert die entsprechenden Datentypen und Operationen. Es kann von beliebigen TL-Programmen zur TML-Generierung und -Analyse genutzt werden. Die wichtigsten Bestandteile der Schnittstelle sind:

- der Typ *Value* zur differenzierten Repräsentation von Werten und entsprechende Konstanten oder Konstruktorfunktionen,
- der Typ T für TML-Instruktionen und entsprechende Konstanten oder Konstruktorfunktionen,

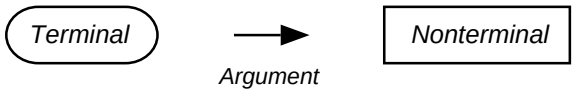

Abbildung 3.4: Die Grundelemente der graphischen TML-Repräsentation

- der Aufzählungstyp *RuntimeCode* und entsprechende Konstanten zur Repräsentation der vordefinierten Funktionen,
- der Typ *Literals* für Literalvektoren,
- der Typ *Closure* zur Repräsentation von Funktionsabschlüssen als Vektoren,
- einige Indizes als Konstanten, die den Aufbau der Literal- und Funktionsabschlußvektoren bestimmen.

Zur genauen Analyse und zum flexiblen Gebrauch sind alle relevanten Datentypen offengelegt. Dennoch sollte die TML-Generierung stets mit Hilfe der Konstruktorfunktionen erfolgen, weil diese eventuelle Änderungen der Datentypen zumindest bedingt abkapseln.

Die Schnittstelle *TML* ist in Anhang A.2 vollständig aufgelistet.

3.8 Eine graphische Darstellung von TML

Der Hauptzweck der in Abschnitt 3.4 beschriebenen Syntax ist die Repräsentation *einzelner* TML-Instruktionen. Zur Darstellung zusammengesetzter TML-Ausdrücke wird eine graphische Darstellung eingeführt. Sie dient der Illustration der Kapitel 4 und 5.

Abbildung 3.4 zeigt die drei graphischen Grundelemente. In Anlehnung an die Terminologie graphischer Grammatiken heißen die runden Knoten, die einzelne TML-Instruktionen explizit angeben, „Terminale“ und die eckigen Knoten, die für nicht näher spezifizierte Ausdrücke stehen, „Nonterminale“.

Argumente, die aus einfachen TML-Werten bestehen, werden ebenfalls direkt in den Terminalen aufgeführt. Die meisten Instruktionen sehen als Argumente wiederum TML-Kode vor. Diese Zuordnung wird durch gerichtete Kanten ausgedrückt.

Ein Beispiel für die so enstehenden Baumstrukturen ist aus Abbildung 3.5 ersichtlich. Die Kennzeichnung der Instruktionen orientiert sich sowohl an den Bezeichnern der Syntax in Abschnitt 3.4 als auch den Bezeichnern der Modulschnittstelle *TML* (siehe Anhang A.2), so daß eine wechselseitige Zuordnung leicht möglich ist. Mangels intuitiver Namen in der Syntax beziehen sich die Bezeichnungen der Argumente auf die Modulschnittstelle. Die Bezeichnung „lambda“ wird durch den gleichnamigen griechischen Buchstaben „λ“ abgekürzt.

Ein als Referenz angegebener TL-Quelltext ist jeweils von einer punktierten Fläche unterlegt. Die Zuordnung von TL- und TML-Elementen wird durch dünne Linien dargestellt.

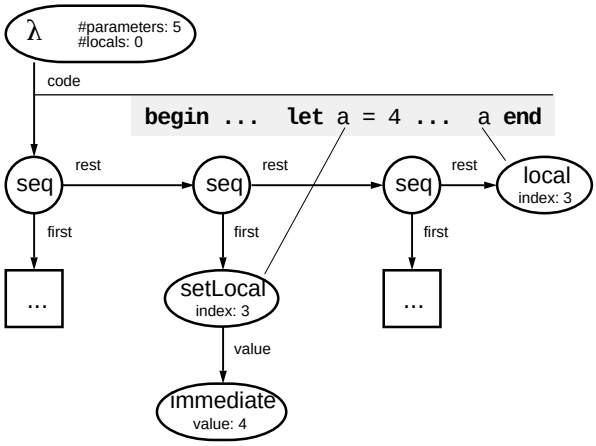

Abbildung 3.5: TML-Beispiel

Kapitel 4

Die Zwischenkodegenerierung

Nach der Darstellung des äußeren Rahmens der Arbeit, der Beschreibung der beteiligten Sprachen und der Bereitstellung der benötigten Beschreibungsmittel wird in diesem Kapitel die Zwischenkodegenerierung behandelt. Sie erfolgt in zwei Phasen, in denen jeweils der gesamte vorliegende Syntaxbaum rekursiv durchlaufen wird. In der ersten Phase werden die Allokationsdaten der in TL deklarierten Datenobjekte ermittelt und in den Syntaxbaum eingetragen. Erst wenn dieser Vorgang abgeschlossen, wird in einem zweiten Durchgang TML-Kode generiert.

Abbildung 4.1 enthält eine schematische Übersicht des Aufbaus und der Funktionsweise des Backends und deutet die Beziehungen zum Frontend an.

Dieses Kapitel beginnt mit der Beschreibung des TL-Syntaxbaumes, der als Eingabedatenstruktur der Zwischenkodegenerierung fungiert. In der anschließenden Darlegung der Programmstrategie werden insbesondere auch der angewandte Programmierstil und die Gliederung des Backends begründet.

Darauf folgt zunächst die ausführliche Beschreibung von der Allokationsphase erzeugten Zwischenergebnisse. Sie gibt Aufschluß über die Zusammenhänge zwischen den Variablenkonzepten und Sichtbarkeitsbereichen von TL, dem Übersetzer-Frontend und der Kodegenerierung. Insbesondere wird deutlich, wie aus den des Syntaxbaum vorhandenen de Bruijn-Indizes eine Repräsentation der TL-Sichtbarkeitsbereiche abzuleiten ist, die im Aufgabenbereich der Allokationsphase adäquat ist.

Dann wird die Allokationsphase beschrieben. In ihrem Aufgabenbereich stellen sich folgende Aufgaben als besonders schwierige Teilprobleme:

- die Berücksichtigung der Polymorphie bei der Allokation von Aggregaten,
- die Allokation von Bindungssequenzen mit abweichenden Sichtbarkeitsregeln (parallele und rekursive Bindungen),
- die Vermeidung unnötiger Zellkonvertierungen für lokale Variablen,
- die Überprüfung der Korrektheit rekursiver Bindungen.

In der anschließend betrachteten Generierungsphase, sind folgende Teilprobleme besonders hervorzuheben:

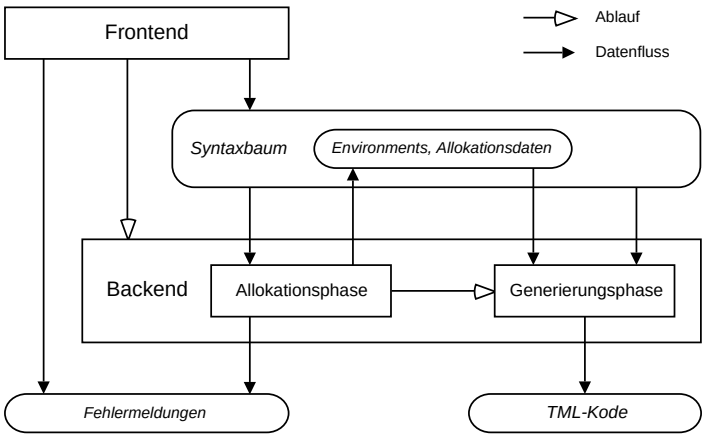

Abbildung 4.1: Schematische Übersicht des Backends

- die Unterstützung der Laufzeitanalyse,
- die Verwaltung von Hilfsvariablen,
- die Berücksichtigung des Objektspeicherprotokolls bei der Implementation veränderlicher Werte,
- das Anlegen von Literalen und die Koordination ihres Zugriffs durch den Kode,
- die Übersetzung rekursiver Bindungen.

Letzteres Teilproblem ist das schwierigste von allen. Es ist isoliert betrachtet bereits recht anspruchsvoll. Darüberhinaus kann es nur im Zusammenhang mit zahlreichen anderen Teilproblemen sinnvoll gelöst werden, da elementare Entwurfsentscheidungen des Übersetzungsalgorithmus betroffen sind.

4.1 Die Repräsentation des Syntaxbaumes

Die Repräsentation attributierter abstrakter Syntaxbäume ist durch das Frontend vorgegeben. Gemäß der TL-Syntax [Matthes 92] ist sie in Werte, Typen, Bindungen und Signaturen gegliedert. Werte und Bindungen werden durch die Modulschnittstelle *TLValue* repräsentiert, Typen und Signaturen durch *TLType*.

Diese Modularisierung ist für die Übersetzung nach TML besonders adäquat, denn die durch das Backend zu eliminierenden hochsprachlichen Typinformationen sind bereits isoliert. Dementsprechend wird fast ausschließlich die Schnittstelle *TLValue* verwendet und *TLType* weitestgehend ignoriert.

Folgende in *TLValue* definierten rekursiven Typen repräsentieren die relevanten Knoten des Syntaxbaumes:

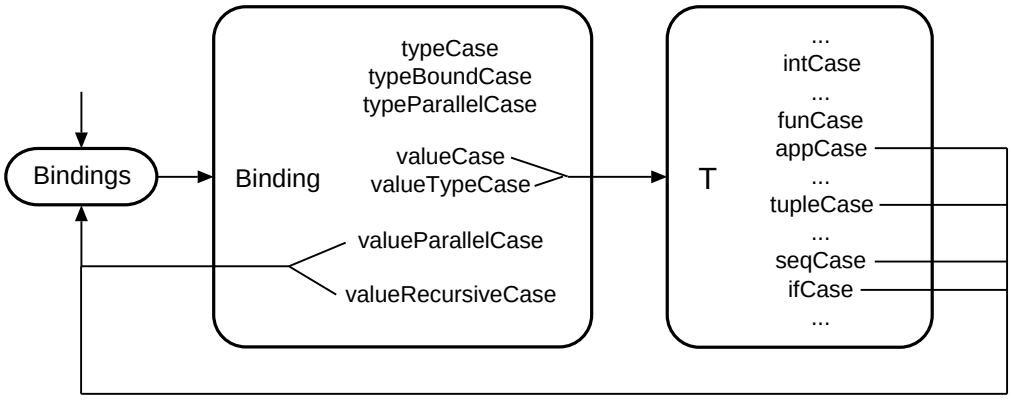

Abbildung 4.2: Die rekursiven Beziehungen der Datentypen des TL-Syntaxbaumes

Bindings : Bindungssequenzen. Diese bestehen jeweils aus einer Liste von Einzelbindungen. Sie sind der Ausgangspunkt der TL-Grammatik, treten aber auch an vielen anderen Stellen auf, z.B. in Kontrollstrukturen oder als Argumentlisten.

Binding : Einzelbindungen. Diese treten in folgenden Varianten auf:

typeCase, typeBoundCase, typeParallelCase : verschiedene Typbindungen, deren innerer Aufbau in *TLType* definiert ist.

valueCase, valueTypeCase : einzelne Wertbindungen. Das Feld *ide* enthält eine Referenz auf den gebundenen Bezeichner. Die Eindeutigkeit dieser Referenz ist später ein geeignetes Mittel zur Identifikation der zugeordneten Variablen (siehe Abschnitt 4.3.2). Der zugewiesene Wert ist im Feld *value* angegeben.

valueParallelCase, valueRecCase : Verzweigungen in Sequenzen paralleler, bzw. rekursiver Bindungen, auf die durch das Feld *bindings* vom Typ *Bindings* verwiesen wird.

T: Werte, bzw. Ausdrücke. Dazu zählen Konstanten, Variablen, Funktionsabstraktionen und -aufrufe, Aggregate und ihre Basisoperationen, Ausnahmewerte und alle Kontrollstrukturen.

Abbildung 4.2 zeigt die rekursiven Beziehungen der Elemente des Syntaxbaumes. Die sehr umfangreiche Schnittstelle *TLValue* ist im Anhang A.1 zur genauen Betrachtung vollständig aufgeführt. Aufgrund der untergeordneten Bedeutung für das Backend wird auf *TLType* nicht näher eingegangen.

4.2 Die Programmstrategie

Die Übersetzung von TL kann nicht durch eine rein lokale, direkte Abbildung geleistet werden. In einigen Fällen hängt die endgültige Ausprägung des für einen Knoten des Syntaxbaumes zu erzeugenden Kodes von Informationen ab, die bei einem einfachen Durchlaufen des Syntaxbaumes erst an späterer Stelle erfaßt werden können. Die wichtigsten Probleme von dieser Art sind folgende:

- Die Allokationsart einer Variablen steht häufig erst nach der Bestimmung aller ihrer Verwendungen fest (siehe z.B. Abschnitt 4.4.5). Der für die Verwendungen zu generierende Kode hängt wiederum von ihrer Allokationsart ab.
- Die TL-Syntax bietet keine Möglichkeit zur mehrmaligen Deklaration von TL-Objekten, die die Übersetzung rekursiver Bindungen unterstützen. Ein bekanntes Beispiel für diese Technik ist die *FORWARD* -Anweisung in Pascal.

Zur Auflösung solcher zyklischer Referenzen können zwei Methoden verwendet werden [Aho et al. 86]:

- Das mehrmalige Durchlaufen bestimmter Teile des Syntaxbaumes.
- Die sogenannte *backpatching*-Technik.
  Sie ermöglicht die Übersetzung einer Eingabedatenstruktur in eine Ausgabedatenstruktur während eines einzigen Durchlaufes durch die Eingabedatenstruktur. Dabei müssen die Teile der Ausgabedatenstruktur, die zum Zeitpunkt ihrer Erzeugung nur unvollständig bestimmt werden können, nachträglich veränderbar und durch Referenzen, die der Eingabedatenstruktur entstammen, zugreifbar sein. Beim Erreichen einer Stelle in der Eingabedatenstruktur, die zusätzliche Informationen für die Übersetzung einer früher besuchten Stelle liefert, wird notwendigerweise stets eine Referenz auf letztere vorgefunden. Diese Referenz wird dann verwendet, um das der zurückliegenden (*back*) Stelle entsprechende Teil der Ausgabedatenstruktur aufzusuchen und zu verändern (*patching*). Von welcher Art die bewußte Referenz ist (z.B. eine Zeichenkette, ein Symbol oder eine Speicheradresse), hängt von der speziellen Aufgabe ab.

Die *backpatching*-Technik ist gewöhnlich mit einem relativ hohen Verwaltungsaufwand und einer geringen lokalen Verständlichkeit verbunden. Sie wird häufig eingesetzt, um die Anzahl der Übersetzerphasen (*passes*) zu reduzieren. Es existieren sogar Übersetzer, die mit nur einer Phase auskommen, wie z.B. Turbo-Pascal. Viele andere Übersetzer übertragen jeweils zwischen zwei Phasen die gesamte Zwischenrepräsentation auf einen Sekundärspeicher und lesen sie anschließend wieder ein. Dieser Vorgang ist extrem zeitraubend.

Durch zwei Entwicklungen wird die Notwendigkeit der Verwendung von Sekundärspeichern jedoch immer mehr eingeschränkt:

- Die Größe der verfügbaren (d.h. erschwinglichen) Hauptspeicher nimmt ständig zu.
- In modernen Programmiersprachen, die die Modularisierung von Programmen gut unterstützen, entstehen typischerweise recht kleine Übersetzungseinheiten. Dies trifft insbesondere auf TL zu.

Aufgrund dieser Voraussetzungen wird die gesamte Übersetzung von TL nach TML im Hauptspeicher geleistet. Da das Frontend bereits mehrere aufwendige Phasen beinhaltet, insbesondere die naturgemäß zeitraubende lexikalische Analyse, ist eine Aufteilung des Backends in zwei Phasen nicht zeitkritisch.

Ein vom Frontend erzeugter Syntaxbaum wird in den folgenden beiden Phasen des Backends jeweils vollständig durchlaufen:

1. Die Allokationsphase

   - ▷ Erfassen der Variablendeklarationen und -benutzungen.
   - ▷ Umsetzen der de Bruijn-Indizes des Syntaxbaumes in direkte Referenzen (siehe Abschnitt 4.3.2).
   - ▷ Erweitern des *Toplevel*-Environments (siehe Abschnitt 4.3.1).
   - ▷ Überprüfung und Zuordnung der Verwendung von *exit*- und *raise*-Anweisungen (siehe Abschnitt 4.4.1).
   - ▷ Überprüfung der Zulässigkeit rekursiver Wertbindungen (siehe Abschnitt 4.4.7).

   Die Ergebnisse der Allokationsphase werden als zusätzliche Attribute in den Syntaxbaum eingetragen.

2. Die Generierungsphase

   - ▷ Generierung der TML-Kodedatenstruktur.
   - ▷ Bestimmung der Indizes lokaler Variablen (siehe Abschnitt 4.5.2).

   Die Generierungsphase wird nur durchgeführt, wenn die Allokationsphase keine Fehlermeldungen ausgegeben hat und verläuft dann stets fehlerfrei.

Diese Aufteilung hat folgende Vorteile:

- ▷ Es ist kein *backpatching* erforderlich. Der TML-Kode kann daher als unveränderliche Datenstruktur angelegt werden und wird so vor späteren Veränderungen geschützt.[1]
- ▷ Die Aufgabe der Variablenallokation wird vollständig von der eigentlichen Kodegenerierung getrennt. Dadurch wird eine hohe Übersichtlichkeit erreicht. Außerdem werden mögliche spätere Anwendungen berücksichtigt, die lediglich einen vollständig attributierten Syntaxbaum mit allen Allokationsdaten benötigen, aber keinen TML-Kode generieren müssen. Dies sind zum Beispiel:
  - Eine alternative Kodegenerierung, die das gleiche Allokationsschema nutzt.
  - Ein Debugger, der im Falle vorhandenen (persistenten) Kodes aber eines fehlenden (nicht persistenten) Syntaxbaumes letzteren unter Verwendung des Frontends und der Allokationsphase erneut erstellt.

[1] Dieser Erfolg wird durch das Fehlen jedweden Auftretens des Schlüsselwortes **var** in der Modulschnittstelle *TML* (siehe Anhang A.2), die die Repräsentation von TML beschreibt, belegt.

- Alle Fehlersituationen, die im Backend auftreten können, werden bereits vor Beginn der Kodeerzeugung überprüft. Die Fallunterscheidung, ob ein Fehler aufgetreten ist, erfolgt nur an einer Stelle. Die Generierungsphase geht stets von einem *vollständig* korrekten Syntaxbaum aus, damit persistent gespeicherte Syntaxbäume bei einer verzögerten Anwendung der Kodegenerierung keine unliebsamen Fehlermeldungen bewirken. Zukünftige Anwendungen für dieses Scenario wären z.B.:
  - Ein Struktur-Editor für TL, der bereits zur Editier-Zeit inkrementell partielle Übersetzungen vornimmt, ohne jedoch Kode zu erzeugen.
  - Ein verkleinertes Laufzeitsystem, das nicht den gesamten Übersetzer enthält.
  - Reflektionsalgorithmen, die vollständig überprüfte Syntaxbäume verwenden.
- Die Umsetzung der im Frontend sehr nützlichen [Matthes 92], im Backend jedoch nutzlosen de Bruijn-Indizes wird in der Allokationsphase gekapselt. Die de Bruijn-Indizes müssen so gut wie möglich vom restlichen Backend isoliert werden, denn sie stellen bezüglich der Korrektheit und der Modifizierbarkeit ein besonders hohes Risiko dar: sie geben *relative* Distanzen in Sichtbarkeitsbereichen an, weshalb ihr Einfluß oftmals durch *transitive* Abhängigkeiten zu weitreichenden Effekten führt.

Weil der abstrakte Syntaxbäume beschreibende Datentyp *TLValue.Bindings* rekursiv definiert ist, wurde als generelle Programmiermethode für beide Phasen der rekursive Abstieg gewählt: Jedem komplexen Knotentyp in der Baumstruktur entspricht eine dedizierte Funktion (*Knotenfunktion*), die Fallunterscheidungen zum rekursiven Aufruf der entsprechenden Funktionen für Folgeknoten enthält.

Von besonderer Bedeutung ist die Problematik der Übersetzung rekursiver Bindungen, die in Abschnitt 4.4.7 und Abschnitt 4.5.7 behandelt wird. Sie hat einen wesentlichen Einfluß auf diverse Entwurfsentscheidungen. Er reicht von den untersten Systemebenen der Kodegenerierung bis hinauf zur Aufgabenverteilung der Übersetzerkomponenten. In Bezug auf letztere gilt die auf den ersten Blick ungewöhnliche Regelung, daß mit der Überprüfung der semantischen Korrektheit rekursiver Bindungen ausschließlich das Backend befaßt ist.

Diese Entscheidung wird mit der theoretischen Unentscheidbarkeit des allgemeinen Problems der Korrektheit rekursiver Bindungen (siehe Abschnitt 4.4.7) begründet. Anders als bei einfachen Bindungen, deren semantische Korrektheit stets entscheidbar und deren Semantik damit wohlbekannt ist, sind bei rekursiven Bindungen im allgemeinen nur Annahmen möglich. Eine Einschränkung auf Seiten des Frontends würde daher möglicherweise der genaueren Beurteilung eines besonders kompetenten Backends zuvorkommen. Zwangsläufig würden ganze Klassen möglicher Backends ausgeschlossen oder daran gehindert, voll zur Geltung zu kommen. Deshalb überprüft das Frontend nur die syntaktische Korrektheit und die Typkorrektheit rekursiver Wertbindungen.

4.3 Die Repräsentation von Variablen und Sichtbarkeitsbereichen

Die Schnittstelle zwischen der Allokations- und der Generierungsphase des Backends (siehe Abbildung 4.1 auf Seite 40) bilden Repräsentationen von Variablen und Sichtbarkeitsbereichen. Diese dienen auch als Grundlage für Browser und Debugger. Ihr Repräsentationsschema wird außerdem im Frontend verwendet, um einen initialen Sichtbarkeitsbereich aufzubauen und um beim Anlegen eines Syntaxbaumes Nullwerte einzutragen, die später von der Allokationsphase ersetzt werden (siehe Abschnitt 4.4).

Eine für die genannten Zwecke adäquate, differenzierte Repräsentation von Variablen und Sichtbarkeitsbereichen leistet das Modul *tlVariable:TLVariable*. Zur Vervollständigung aller in ihm berücksichtigten Bedingungen wird im nachfolgenden Abschnitt zunächst auf die interaktive Benutzerumgebung eingegangen.

4.3.1 Die Unterstützung der interaktiven Benutzerumgebung

Das in Abschnitt 2.2 beschriebene Verhalten des Tycoon-Systems als interaktiver Quasi-Interpreter beruht im wesentlichen auf der Erweiterung eines persistenten Sichtbarkeitsbereiches und einer weiteren persistenten Datenstruktur, welche die gebundenen Werte aufnimmt. Da in TL die lexikalische Reihenfolge für die Sichtbarkeit von Bezeichnern maßgeblich ist, erfolgt die Erweiterung beider Strukturen schrittweise sequentiell.

Die interaktive Benutzerumgebung wird als *top level* bezeichnet, die Repräsentation ihres Sichtbarkeitsbereiches in Anlehnung daran als *Toplevel*-Environment. Letzteres ist mit Hilfe rückwärts verketteter Listen implementiert, welche die statischen Sichtbarkeitsregeln besonders einfach modellieren.

Da TML-Kode portabel sein muß und folglich keine direkten Referenzen auf Datenobjekte enthalten darf, besteht die effizienteste mögliche Zugriffsform auf Laufzeitwerte, die dem *Toplevel*-Environment zugeordnet sind, in der indirekt indizierten Adressierung eines Vektors. Dieser Vektor wird *Toplevel*-Vektor genannt. Die vom Kode zu verwendenden Indizes werden bereits in den Deklarationseinträgen des *Toplevel*-Environments gespeichert.

Die Füllung des *Toplevel*-Vektors kann zur Übersetzungzeit genau festgestellt werden, weil sie analog zur Erweiterung des *Toplevel*-Environments verläuft. Im Falle eines drohenden Überlaufs wird sein Inhalt an den Anfang eines größeren Speicherblocks kopiert, der dann als neuer *Toplevel*-Vektor den vorherigen ablöst. Diese Operation findet stets zwischen einer Übersetzungs- und einer Laufzeitphase statt, unterliegt der alleinigen Kontrolle des Laufzeitsystems und ist daher nicht zeitkritisch.

Die wesentliche Voraussetzung für den dynamischen Wechsel ist, daß keine Referenzen auf den *Toplevel*-Vektor bestehen, die mehr als einen Evaluationszyklus überdauern. Deshalb werden alle variablen Werte mittels Zellkonvertierung ausgelagert, denn sonst müßten sie direkt im *Toplevel*-Vektor referenziert werden. Vor der Laufzeitphase bettet das Laufzeitsystem den Kode in eine neue Funktion (*Toplevel*-Funktion) ein, deren Funktionsabschluß den aktuellen Vektor als globalen Wert erhält. Diese Funktion wird nach der Ausführungsphase verworfen und damit auch die Referenz auf den *Toplevel*-Vektor.

4.3.2 Sichtbarkeitsbereiche

Im Syntaxbaum wird bei der Verwendung eines Bezeichners durch einen de Bruijn-Index auf seine Deklaration verwiesen. Der Index gibt die „lexikalische Entfernung“ an, d.h. wieviele neue Bindungen zwischen dem Bezeichner und seiner eigenen liegen. Dazu ein TL-Beispiel:

```
Let I = Int
let f(x, y :I) :I =  x
let a = 0
begin
  let b = 1
  let c = 2
  f(a b)
end
```

Dem Bezeichner a im Funktionsaufruf ist der de Bruijn-Index 3 zuzuordnen, denn er bezieht sich auf die drittletzte der vorangehenden Bindungen, die durch das Schlüsselwort **let** gekennzeichnet sind. Der de Bruijn-Index von *b* ist jedoch ebenfalls 3. Zwar befindet sich nur die **let**-Bindung von *c* zwischen ihm und seiner eigenen Bindung, doch das Auftreten von *a* und *b* wird vom Frontend ebenfalls als Bindung angesehen. Der Funktionsaufruf hätte nämlich auch wie folgt lauten können:

```
f(let x = a  let y = b)
```

Nun sind die Bindungen klar ersichtlich, im obigen Fall hingegen blieben die Namen der Parameter anonym. Dementsprechend werden Bindungen deren Zielobjekt nicht explizit angegeben ist, „anonyme Bindungen“ genannt. Da Sequenzen in TL syntaktisch betrachtet stets aus Bindungen bestehen, steht auch der Funktionsaufruf von *f* in einer anonymen Bindung. Das Frontend erweitert solche Anweisungen, indem es einen nicht näher spezifizierten Bezeichner „erfindet“:

```
let <anonymous> = f(let <anonymous> = a  let <anonymous> = b)
```

Das Backend berücksichtigt diese speziellen Vorgaben des Frontends in seiner eigenen Repräsentation der lexikalisch geschachtelten Sichtbarkeitsbereiche von TL. Sie werden mit folgender Interpretation durch einfach verkettete Listen analog modelliert: Die leere Liste stellt einen leeren Sichtbarkeitsbereich dar. Bei einer Erweiterung des Sichtbarkeitsbereiches durch eine Deklaration wird einfach ein Element am Anfang der Liste angefügt. Alle Nachfolger eines Listenelements liegen in seinem Sichtbarkeitsbereich. Um das einem de Bruijn-Index entsprechende Element aufzufinden, wird die Liste um die angegebene Zahl von Elementen zurückverfolgt. Eine bereits bestehende Liste kann mehrmals von der gleichen Stelle ausgehend erweitert werden, wodurch auch Verzweigungen in geschachtelten Sichtbarkeitsbereichen auf einfache Weise verwaltet werden.

Die beschriebene Repräsentation eines Sichtbarkeitsbereiches wird „Environment“ genannt (vgl. [Aho et al. 86; Abelson et al. 84]) und entsprechend lautet auch die Deklaration in *TLVariable*:

Let *Environment = list.T(Declaration)*

Zur Erzeugung von Environments werden folgende Konstruktorfunktionen verwendet:

newTypeEnvironment(environment :Environment ide :TLIde.T) :Environment

newAnonymousEnvironment(environment :Environment ide :TLIde.T) :Environment

newTopLevelEnvironment(environment :Environment context :Context
ide :TLIde.T mutable :Bool) :Environment

newParameterEnvironment(function :Function environment :Environment
ide :TLIde.T mutable :Bool) :Environment

newLocalEnvironment(function :Function environment :Environment
context :Context ide :TLIde.T mutable :Bool) :Environment

newIndexedEnvironment(function :Function environment :Environment
context :Context ide :TLIde.T mutable :Bool) :Environment

Die Funktion *newTypeEnvironment* erzeugt eine Typdeklaration. In den anderen Fällen korrespondiert die Benennung mit der Art der erzeugten Variablen (siehe Abschnitt 4.3.3).

Abbildung 4.3 zeigt unterschiedliche Environment-Arten und ihre Schachtelung anhand des bereits bekannten TL-Beispiels. Mit Ausnahme der Parameter-Environments, die aus Signaturen hervorgehen, ist jedes Environment eindeutig einer Bindung zugeordnet. Für die Bezeichner des Funktionsaufrufes *f(a b)* und *x* im Funktionsrumpf wird die Korrespondenz der Sichtbarkeitsbereiche zu den de Bruijn-Indizes verdeutlicht. Aus Gründen der Übersicht werden sie jeweils in einem Kreis wiederholt dargestellt. Die schwarzen Pfeile geben die Verkettung der Environments an und die weißen Pfeile zeigen direkt auf das Deklarations-Environment eines Bezeichners. Um von einem Kreis ausgehend mit schwarzen Pfeilen zum gleichen Environment zu gelangen wie mit einem weißen, ist genau die durch den (de Bruijn-) Index angegebene Anzahl von Pfeilen zu verfolgen.

Folgende Funktion wird beim Auftreten eines Bezeichners im Syntaxbaum aufgerufen und leistet die dargestellte Rückverfolgung eines de Bruijn-Indizes:

```
let rec lookupDeBruijnIndex(environment :Environment index :Int) :Environment =
  if index <= 1 then
    environment
  else
    lookupDeBruijnIndex(list.tail(environment) {index - 1})
  end
```

Als Parameter werden die Referenz auf das aktuelle Environment und der vom Frontend eingetragene de Bruijn-Index des Bezeichners übergeben. Das Ergebnis ist die Referenz auf das Deklarations-Environment des Bezeichners.

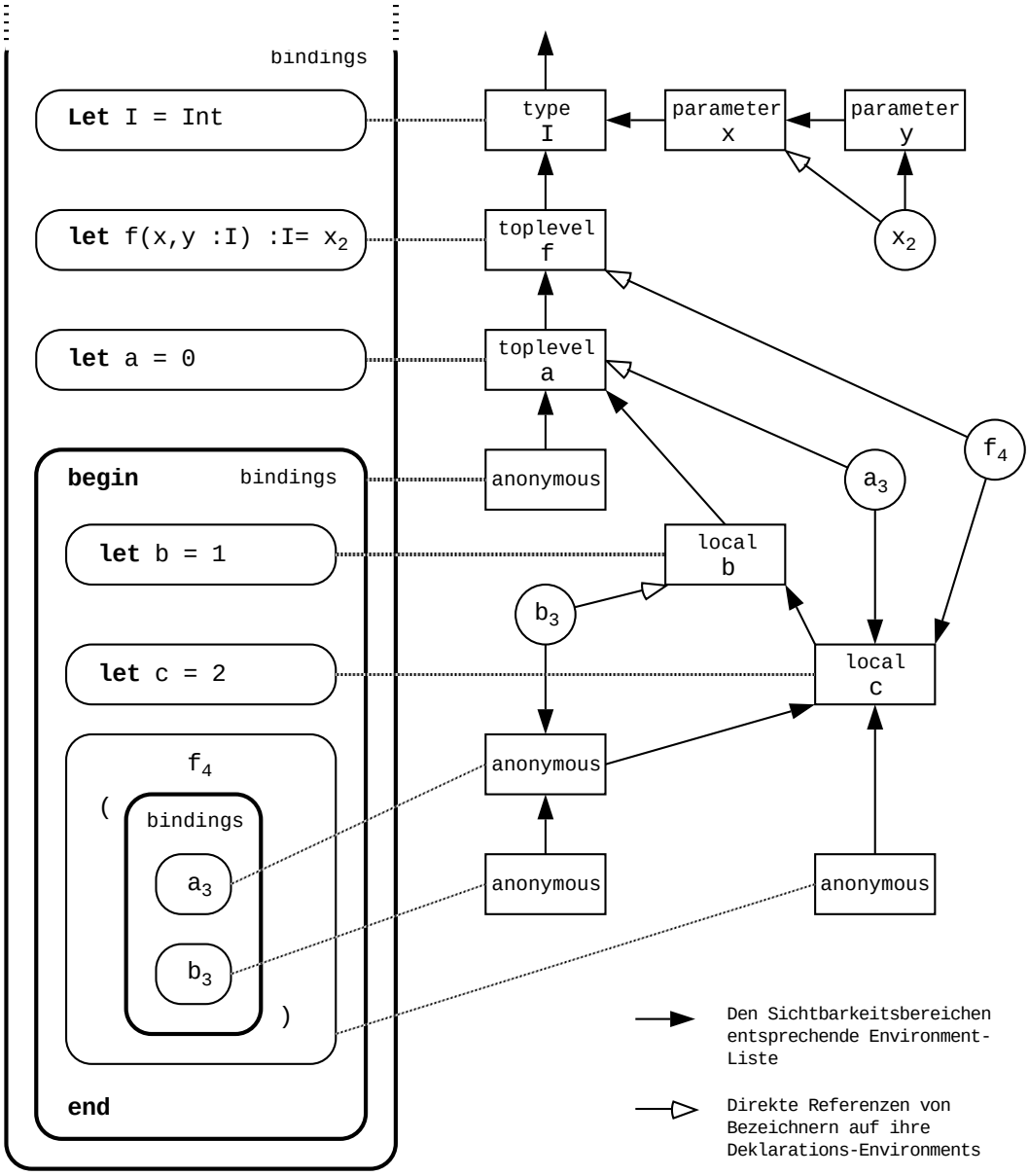

Abbildung 4.3: Zusammenhänge zwischen Bindungen, geschachtelten Sichtbarkeitsbereichen, Environments, Bezeichnern und de Brujin-Indizes

Der Syntaxbaum wird in der Allokationsphase an folgenden Stellen mit einem zusätzlichen Attribut vom Typ *Environment* versehen, wobei das erste Element der Liste dem betrachteten Knoten im Syntaxbaum entspricht:

- in den Varianten *valueCase, valueTypeCase* des Typs *TLValue.Binding*, der sämtliche Bindungskonstrukte in TL beschreibt.
  Es werden sowohl für Typ- als auch für Wertbindungen Environments angelegt, um die vom Frontend vorgegebene Zählweise der de Bruijn-Indizes zu beachten. Da TML untypisiert ist, sind Typinformationen jedoch im weiteren Verlauf nicht relevant. Daher erfolgen die Eintragungen nur in den angegebenen Varianten für Wertbindungen.

- in den Varianten *caseCase, caseCrashCase, caseExhaustiveCase* des Typs *TLValue.T*, die Kontrollstrukturen für alternative Mehrwegverzweigungen repräsentieren (Schlüsselwort **case**).
  Hierbei handelt es sich jeweils um eine lokale Hilfsvariable, die den durch **with**-Klauseln eingeführten Bezeichnern entspricht. Beispiel:

  case a*TupleWithVariants*
  when *variantA* **with** *handleA* **then** ...
  when *variantB* **with** *handleB* **then** ...
  end

  Da ihre Sichtbarkeitsbereiche sich nicht überschneiden, werden die Bezeichner *handleA* und *handleB* zu einer Variablen zusammengefasst.

- in der Variante *forCase* des Typs *TLValue.T*.
  An dieser Stelle wird die Laufvariable der **for**-Anweisung erfaßt.

- in der Variante *tryCase* des Typs *TLValue.T*.
  Hier wird eine spezielle lokale Variable angelegt, die im Falle einer Ausnahmebehandlung den Ausnahmewert zugewiesen erhält, um ihn für anschließende Fallunterscheidungen verfügbar zu halten.

4.3.3 Deklarationen und Variablen

Bindungen erfüllen in TL gleichzeitig mehrere Aufgaben, was bereits an einem einfachen Beipiel ersichtlich ist.

let a = *3 + 4*

In dieser Anweisung treten folgende Aktionen auf:

- ▷ die Berechnung eines Ausdruckes,
- ▷ die Allokation einer Variablen,
- ▷ die Zuweisung des Berechnungsergebnisses an die Variable,
- ▷ die Benennung der Variablen,
- ▷ die Einordnung der Variablen in ihre Umgebung, insbesondere die Erfassung ihres Sichtbarkeitsbereiches.

In der Terminologie vieler typisierter Programmiersprachen (z.B. Modula-2, Pascal) wird für die statische Festlegung der Eigenschaften einer Variablen die Bezeichnung "Deklaration" verwendet. Dieser entsprechen in der Implementation der Übersetzung von TL die beiden letzten Punkte obiger Aufzählung und die Vorbereitung der Allokation. Der Typ in *TLVariable*, der die dafür notwendigen Informationen repräsentiert ist:

```
Let Rec Declaration = Tuple
  ide :TLIde.T
  case type
  case variable with
    variable :Variable
  end
end
```

Das Feld *ide* verweist eindeutig unterscheidbar auf den deklarierten Bezeichner. Die Variante *type* steht für Typdeklarationen, wobei die Erfassung ihrer Anwesenheit genügt. Variablen werden weiter differenziert:

```
and Variable = Tuple
  case anonymous
  case topLevel with
    mutable :Bool
    index :Int
  end
  case nested with
    function :Function
    nested :NestedVariable
  end
```

```
    case global with
      global :GlobalVariable[2]
    end
  end
```

Die Bedeutungen der Varianten sind:

anonymous : eine anonyme Variable.

topLevel : eine Variable im *Toplevel*-Environment.

nested : eine Variable, die in einem geschachtelten (*nested*) Sichtbarkeitsbereich deklariert worden ist, d.h. nicht im *Toplevel*.

global : eine Variable, die global zu einer Funktion deklariert worden ist.

Wenn im Syntaxbaum in der Variante *ideCase* des Typs *TLValue.T* durch einen Bezeichner auf eine Variable verwiesen wird, so erfolgt dort in der Allokationsphase ein Eintrag vom Typ *Variable*.

Die Variablen der Variante *nested* werden weiter unterschieden in Parameter, lokale Variablen und Einträge in Aggregaten (Variante *indexed*) (vgl. Abbildung 2.3 auf Seite 15):

```
and NestedVariable = Tuple
  mutable :Bool
  case parameter with
    index :Int
  end
  case local with
    var index :Int
    allocation :Allocation[3]
  end
  case indexed with
    index :Int
    var objectLocalIndex :Int
  end
end
```

Abbildung 4.4 faßt die Hierarchie der beschriebenen Repräsentationsformen zusammen.

4.3.4 Funktionsparameter und globale Variablen

Aufgrund des Aufbaus der Repräsentation von Funktionsabschlüssen (siehe Abschnitt 3.2) sind nicht nur Parameter sondern auch globale Variablen eng mit Funktionen verbunden. Für die Erfassung der Parameter einer Funktion, sowie zur Sammlung und Aufbereitung ihrer globalen Variablen, enthält *TLVariable* folgenden Typ:

[2] Zu *GlobalVariable* siehe Abschnitt 4.3.4.
[3] Zu *Allocation* siehe Abschnitt 4.4.5.

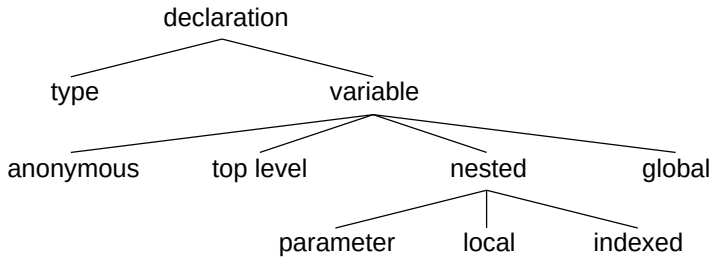

Abbildung 4.4: Die Hierarchie der Repräsentation von Deklarationen

```
and Function = Tuple
  case topLevel
  case nested with
    parent :Function
    globals :assoc.T(TLIde.T  GlobalVariable)
    var nParameters :Int
  end
end
```

Die zu erweiternde Stelle im Syntaxbaum ist die Variante *funCase* des Typs *TLValue.T*.

Die *Toplevel*-Funktion als Wurzel der Sichtbarkeitshierarchie wird mittels der Variante *topLevel* repräsentiert. Alle anderen Funktionen werden als geschachtelt (*nested*) betrachtet und verweisen durch das Feld *parent* auf die sie umgebende Funktion.

Aufgrund der Homogenintät der Datenstrukturen in TML genügt die rein quantitative Erfassung der Parameter. Dazu dient das Feld *nParameters*.

Das Feld *globals* enthält eine Assoziationsliste[4] von Bezeichnern und den Allokationsdaten globaler Variablen, die wie folgt deklariert sind:

```
and GlobalVariable = Tuple
  origin :list.T(Declaration)
  var index :Int
  var mutable :Bool
  case direct
  case indirect with
    parent :GlobalVariable
  end
end
```

Aus den in der Allokationsphase gesammelten Einträgen globaler Variablen wird in der Generierungsphase Initialisierungskode für den Funktionsabschluß erzeugt.

Die ursprüngliche Deklaration einer globalen Variablen wird durch das Feld *origin* angegeben. Sie stammt entweder aus der direkt umgebenden Funktion (Variante *direct*), oder sie läßt

[4] Deren Typ *assoc.T* entstammt einer Tycoon-Standardbibliothek.

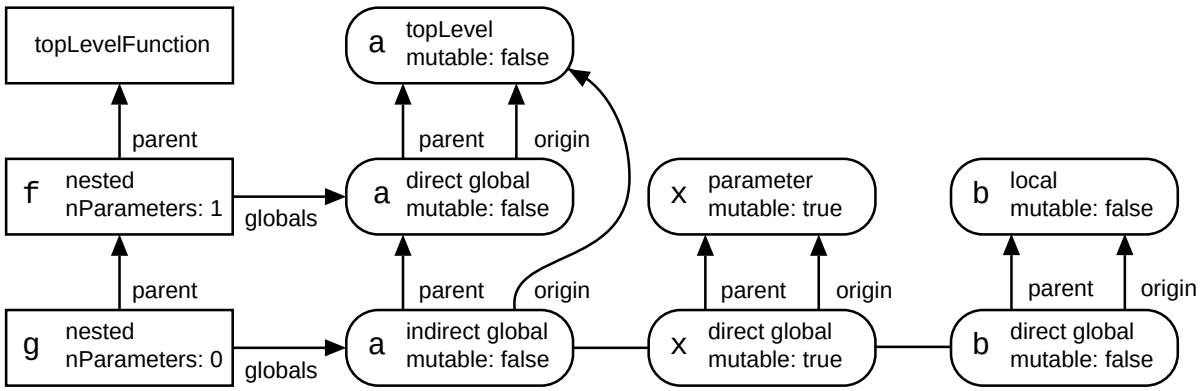

Abbildung 4.5: Verschiedene Ausprägungen globaler Variablen

sich über mehrere Schachtelungsebenen hinweg zurückverfolgen (Variante *indirect*). Letzteres setzt die Existenz einer (gegebenenfalls zu generierenden) globalen Variablen in der direkt umgebenden Funktion voraus, auf die dann das Feld *parent* verweist.

Aus diesen Informationen ergibt sich dann die Variable, die zur Initialisierung vorgesehen wird. Sie ist in der Variante *direct* durch *origin* bestimmt und von der Art *topLevel* oder *nested* (siehe Abschnitt 4.3.3). In der Variante *indirect* wird sie durch *parent* angegeben und ist selbst wiederum global. Das Feld *mutable* gibt an, ob es sich um eine veränderliche Wertbindung handelt.

Abbildung 4.5 zeigt die verschiedenen Ausprägungen globaler Variablen, die in folgendem TL-Beispiel auftreten:

```
let a = ...
let f(var x :Int) :Ok =
  begin
    let b = ...
    let g() :Ok =  x := a + b
    ...
  end
```

4.4 Die Allokationsphase

Die Aufgabe der Allokationsphase besteht darin, den Syntaxbaum rekursiv zu durchlaufen und ihn dabei um zusätzliche Attribute zu erweitern, die zur Laufzeit zu erzeugende Datenobjekte beschreiben. Dabei werden Allokationsinformationen und Sichtbarkeitsbereiche durch die in Abschnitt 4.3 beschriebenen Datenstrukturen repräsentiert.

Außerdem werden alle nach erfolgreicher Beendigung des Frontends noch möglichen Fehlersituationen erkannt. Besonders kompliziert ist die Überprüfung der Korrektheit rekursiver Bindungen (siehe Abschnitt 4.4.7).

4.4.1 Die Programmorganisation der Allokationsphase

Zur Bearbeitung der für die Allokation relevanten Knotentypen ist jeweils eine eigene rekursive Funktion vorgesehen (vgl. Abschnitt 4.2). Die wichtigsten sind:

- *traverseBindings* für Bindungssequenzen mit lexikalischer Anordnung der Sichtbarkeitsbereiche.
- *traverseParallelBindings* für Sequenzen paralleler Bindungen (siehe Abschnitt 4.4.6).
- *traverseRecBindings* für Sequenzen rekursiver Bindungen (siehe Abschnitt 4.4.7).
- *traverseBinding* für den Typ *TLValue.Binding*, der sowohl Einzelbindungen beschreibt, als auch die Verzweigungen in parallele oder rekursive Bindungen.
- *traverseValue* für den Typ *TLValue.Value*.
- *traverseIde* für die Variante *ideCase* des Typs *TLValue.Value*.

Zur Repräsentation von Bindungssequenzen wird im Syntaxbaum einheitlich der Typ *TLValue.Bindings* verwendet. Dennoch ist gewöhnlichen, parallelen und rekursiven Bindungen jeweils eine eigene Funktion zugeordnet, um ihren unterschiedlichen Environment-Strukturen zu entsprechen.

Alle anderen Kontextabhängigkeiten werden durch die folgenden, durchgehend einheitlich benannten Parameter realisiert:

environment :TLVariable.Environment : das aktuelle Environment (siehe Abschnitt 4.4.2).

context :TLVariable.Context : der verallgemeinerte Kontext eines Ausdruckes (siehe unten).

recursion :tlCheckRec.Recursion : eine komplexe Struktur zur Überprüfung der Benutzung von rekursiv gebundenen Bezeichnern (siehe Abschnitt 4.4.7).

state :State : eine Zusammenfassung mehrerer potentiell eigenständiger Parameter, die sich aber nur relativ selten ändern:

log :errorLog.T : der Fehlerausgabekanal.

function :TLVariable.Function : Daten der aktuellen Funktion (siehe Abschnitt 4.3.4).

inLoop :Bool : gibt an, ob der aktuelle Knoten sich innerhalb einer *loop*-Anweisung befindet. Wenn nicht, bedeutet das Auftreten einer *exit*-Anweisung einen Fehler.

inTryHandler :Bool : gibt an, ob der aktuelle Knoten sich innerhalb des *else*- oder eines der *then*-Zweige einer *try*-Anweisung befindet. Wenn nicht, bedeutet das Auftreten einer *reraise*-Anweisung einen Fehler.

Die TL-Grammatik ist sehr einheitlich, denn sie beschränkt sich auf wenige grundlegende Konstrukte, die jeweils für die verschiedensten Zwecke eingesetzt werden. Die Ausdrucksfähigkeit von TL basiert daher zum großen Teil auf Kontextabhängigkeiten. Absolut gleichlautende TL-Anweisungen können an unterschiedlichen Stellen eines Programmes stark voneinander abweichende Bedeutungen haben.

Für die Übersetzung sind allerdings nur wenige prinzipielle Unterschiede relevant. Diese repräsentiert folgender Typ:[5]

```
Let Context = Tuple
  case type, value, function, argument, varArgument, assignment
  case indexed, topLevel with
    var index :Int
  end
end
```

Für jede Variante existiert eine Konstante oder eine Konstruktorfunktion mit dem Suffix „Context“: *typeContext*, *valueContext* usw.

Um die unterschiedlichen Bedeutungen der einzelnen Kontextarten zu veranschaulichen, folgt ein TL-Beispiel, in dem für einige Ausdrücke der jeweilige Kontext als Index angegeben ist:

```
let var a :Int = 0 topLevel
begin
  let assign(A <:Ok  var lValue :A  rValue :A) =  lValue assignment := rValue value
  assign function(:Int type  a varArgument  {3 + 4} argument)
  a value
end
a topLevel
```

Die Kontexte von Bindungen verhalten sich analog zu denen von Werten. Deshalb wird für sie ebenfalls der gleiche Kontexttyp verwendet.

[5] Der hier angegebene Typ *Context* wird ausschließlich in der Allokationsphase verwendet. Er ist jedoch in *TLVariable* definiert, weil einige der dort vorhandenen Konstruktorfunktionen ihn benötigen.

4.4.2 Der Aufbau der Sichtbarkeitsbereiche

Durch die in Abschnitt 4.3.2 beschriebene Verknüpfung der Environments entsteht eine Baumstruktur, die zu einem Teil des Syntaxbaumes kongruent ist. Sie wird deshalb durch simple rekursive Programmierung konstruiert. Ein typisches Beispiel für diese Methode ist die Funktion *traverseBindings*:

```
let rec traverseBindings(state :State  recursion :tlCheckRec.Recursion
                         environment :tlVariable.Environment  context :tlVariable.Context
                         bindings :TLValue.Bindings) :tlVariable.Environment =
  if list.empty(bindings) then
    environment
  else
    let e = traverseBinding(state  recursion  environment  context
                            list.head(bindings))
    traverseBindings(state  recursion  e  context  list.tail(bindings))
  end
```

Dem Parameter *bindings* wird stets eine Liste von Bindungen in lexikalischer Reihenfolge übergeben, so daß die Deklarationen im Endergebnis in umgekehrter Reihenfolge erreichbar sind. In *traverseBinding* wird die Fallunterscheidung der Varianten des Typs *TLValue.Binding* vorgenommen. Alle Varianten führen zu einer Erweiterung des Environments.

Die Parameter haben folgende Bedeutungen:

environment : das zu erweiternde Environment.

ide : der gebundene Bezeichner.

mutable : gibt an, ob es sich um eine variable Bindung (**let var**...) handelt.

function : die Funktion, in der sich die Bindung befindet.
Um bei einer möglichen späteren Verwendung als globale Variable die Funktion zuordnen zu können (siehe Abschnitt 4.3.4), wird diese bereits beim Anlegen der Variable in der Variante *nested* (siehe Abschnitt 4.3.3) eingetragen.

context : der Kontext, in dem die Bindung als Ausdruck betrachtet steht (siehe Typ *Context* auf Seite 55).

Die beschriebene Konstruktion der Environments ist genau an die Verwendung der de Bruijn-Indizes im Syntaxbaum (siehe Abschnitt 4.3.2) angepaßt.

4.4.3 Die Analyse einer Bindung

Um den Gebrauch der in Abschnitt 4.3 beschriebenen differenzierten Datentypen, ihrer Konstruktorfunktionen (siehe Abschnitt 4.3.2) und insbesondere die Fallunterscheidung anhand des Kontextes zu verdeutlichen, folgt nun ein Beispiel von zentraler Bedeutung:

```
let allocateBinding(function :Function  environment :Environment
                    context :Context  binding :TLValue.Binding)
                   :Environment =
  case binding when valueCase, valueTypeCase with b then
    (* Typbindungen und Zuweisungen sind hier ausgeschlossen: *)
    ASSERT(not(context?type orif context?assignment))
    let e =
      case context
      when topLevel then
        tlVariable.newTopLevelEnvironment(environment  context
                                          b.ide  b.mutable)
      when indexed then
        tlVariable.newIndexedEnvironment(function  environment  context
                                         b.ide  b.mutable)
      else
        if tlIde.isAnonymous(b.ide.name) then  (* ist die Bindung anonym? *)
          tlVariable.newAnonymousEnvironment(environment  b.ide)
        else
          tlVariable.newLocalEnvironment(function  environment  context
                                         b.ide  b.mutable)
        end
      end
    b.allocation := optional.new(e)     (* Eintrag in den Syntaxbaum! *)
    e
  end
```

Diese Funktion wird von *traverseBinding* zur Allokation einer einzelnen Wertbindung aufgerufen.

Die genaue Bedeutung einer Wertbindung ist durch die TL-Syntax nicht erfaßt und wird erst durch ihren Kontext festgelegt. In obiger Fallunterscheidung sind dazu alle Möglichkeiten erfaßt.[6] In den Varianten *topLevel* und *indexed* kann syntaktisch auch eine anonyme Bindung auftreten. Dennoch wird stets eine Variable alloziert, weil sie zumindest zur Initialisierung des betroffenen Feldes im Aggregat, bzw. im *Toplevel*-Vektor[7] mit dem Bindungswert benötigt wird.

Am Schluß von *allocateBinding* erfolgt der Eintrag des erweiterten Environments in den Syntaxbaum. Es wird außerdem zur Unterstützung der weiteren rekursiven Schachtelung (siehe Funktion *traverseBindings* auf Seite 56) als Funktionsergebnis zurückgegeben.

4.4.4 Die Speicherstrukturen von Aggregaten

Die im vorangehenden Abschnitt behandelte Kontextanalyse im Rumpf einer Kontenfunktion erfordert die Konstruktion von Kontexttupeln. Ein typisches Beispiel für eine solche Konstruk-

[6] Zu den nicht erfaßten Variablenarten: Parameter werden in Signaturen deklariert, nicht in Bindungen. Globale Variablen treten im Syntaxbaum natürlich nicht explizit auf.

[7] Der *Toplevel*-Vektor wird weitestgehend wie ein Array behandelt (siehe Abschnitt 4.4.4).

tion enthält die Behandlung der Auslösung von Ausnahmen in der Funktion *traverseValue*:[8]

```
...
when raiseCase with v then
  let r = enterExpression(recursion)
  traverseValue(state r environment valueContext v.value)
  traverseBindings(state r environment newIndexedContext(1) v.bindings)
...
```

Das zu erzeugende Ausnahmepaket hat die Struktur eines Aggregates, wobei der Wert *v.value* die Identität der Ausnahme bestimmt und den festen Index 0 einnimmt. Die übrigen Werte sind durch v.bindings angegeben und belegen Indizes ab 1 aufwärts. Das Feld *index* im durch den Aufruf von *newIndexedContext* erzeugten Kontexttupel (siehe Typ *Context* auf Seite 55) wird daher mit 1 initialisiert. Der Aufruf von *traverseBindings* führt über *traverseBinding* und *allocateBinding* schließlich zur Anwendung von *newIndexedEnvironment* für jede einzelne Bindung. In letzterer Funktion wird das Feld *index* des Kontextes als Seiteneffekt jeweils um 1 erhöht, so daß die allozierten Variablen auf einfache Weise den Positionen im Aggregat zugeordnet werden können.

Durch die beschriebene Verwendung des Indexfeldes im Kontextupel wird die Allokation sämtlicher Arten von Aggregaten unterstützt.

Dazu zählt auch der *Toplevel*-Vektor. In seinem Fall wird durch die Funktion *newTopLevelEnvironment* das Feld *index* in der Variante *toplevel* des Typs *Context* inkrementiert wird. Den zu Beginn einer Allokationsphase aktuellen *Toplevel*-Index bestimmt die Funktion *newTopLevelContext*, indem sie im bisherigen *Toplevel*-Environment die jüngste Deklaration einer *Toplevel*-Variablen aufsucht. Diese enthält explizit die Angabe des zuletzt vergebenen Index.

Bei Arrays beginnt die Zuordnung der Indizes mit 0. Obwohl nur in Tupeln mit Varianten ein Diskriminationswert erforderlich ist, erhalten variantenlose Tupel ebenfalls einen zusätzlichen Eintrag, der stets den Wert 1 hat. Sie verhalten sich dadurch genau wie Tupel mit einer Variante und infolgedessen hat das erste Feld in allen Tupeln den Index 1. Diese Vereinheitlichung dient der Implementierung eines Subtyppolymorphismus, der auf folgender Beziehung beruht [Matthes 92]:

> Ein Tupeltyp A *ohne* Varianten mit Signaturen S ist ein Subtyp eines Tupeltyps B *mit* Varianten, falls die Signaturen S Tupelsubsignaturen S' der ersten Variante von B sind.

Durch diese Subtypbeziehung wird das bekannte Datenbankproblem, unvorhergesehenermaßen Erweiterungen und Varianten von Datenstrukturen einzuführen zu müssen, ohne existierende persistente Bindungen invalidieren zu dürfen, typsicher gelöst.

Abbildung 4.6 zeigt für jede in TL auftretende Aggregationsart ein Beispiel und den Aufbau des entsprechenden Speicherobjektes. Der Typ *Punkt* in Abbildung 4.6 ist ein Subtyp von *Ort*. Durch die analoge Anordnung der Felder in den Tupeln *punkt1* und *ort1* gelingt zum Beispiel die korrekte Übersetzung folgender polymorphen Funktion:

[8] Zu *recursion*, bzw. *r* siehe Abschnitt 4.4.7.

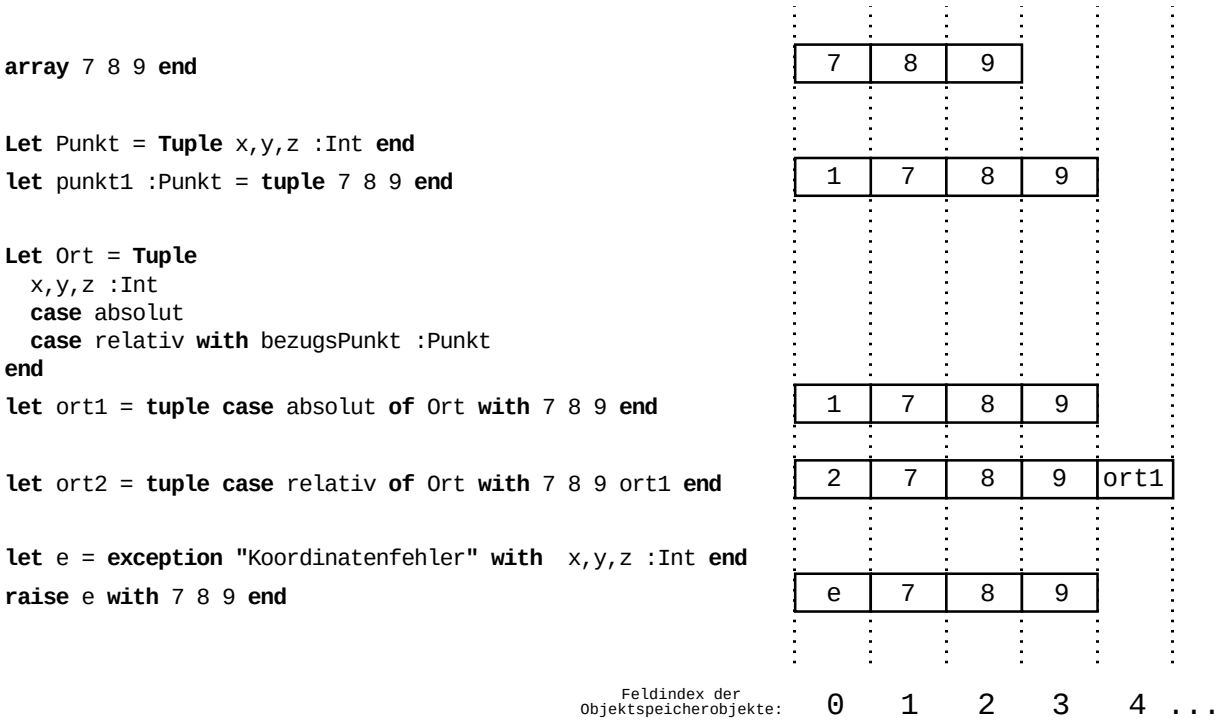

Abbildung 4.6: Verschiedene Formen von Aggregaten und ihre Speicherobjekte

```
let norm(p :Ort) =  p.x * p.x + p.y * p.y + p.z * p.z
```

Aufrufe der Funktion *norm* sind mit allen in Abbildung 4.6 deklarierten „Orten“ möglich:

```
norm(punkt1)
norm(ort1)
norm(ort2)
```

Der Ablauf (und in diesem Fall auch das Ergebnis: 194) ist in jedem der drei Aufrufe genau gleich.

4.4.5 Die Vermeidung der Zellkonvertierung lokaler Variablen

Zur Repräsentation von Variablen, die eine veränderliche Bindung beschreiben, wird in den meisten Fällen eine Zellkonvertierung durchgeführt (vgl. Abschnitt 4.5.4). Dabei ist von entscheidender Bedeutung, ob auf eine lokale Variable potentiell noch zugegriffen werden kann, nachdem die Ausführung der Funktion, in der sie sich befindet, bereits beendet ist. Wenn dies der Fall ist, dann kann ihre Lebensdauer (*extent*) mittels der Auslagerung ihres Wertes in eine Zelle über die Lebensdauer der aktuellen Instanz ihres lokalen Sichtbarkeitsbereiches (*scope*) hinaus verlängert werden. Eine Möglichkeit zur Vermeidung einer Zellkonvertierung bietet das Kopieren des Aktualwertes einer lokalen Variablen in einen anderen Kontext, wenn festgestellt werden kann, daß er bis zu seiner Verwendung unverändert bleibt. Die Zellkonvertierung ist also bei lokalen Variablen häufig vermeidbar, obwohl ihre Deklaration sie (durch das Schlüsselwort **var**) explizit als veränderlich ausweist.

Als Bedingung für die Zellkonvertierung einer Variablen wird die Erfüllung mindestens eines der folgenden Kriterien gefordert:

1. Die Variable ist in der lokalen Funktion das Ziel einer Zuweisung *und* ihr Wert wird in einer tiefer geschachtelten Funktion verwendet.
2. Die Variable ist in einer tiefer geschachtelten Funktion das Ziel einer Zuweisung.
3. Die Variable wird bei einem Funktionsaufruf als veränderliches Argument übergeben.

Dazu jeweils ein (nahezu minimalistisches) Beispiel:

```
begin
  let var a = 1
  a := 2                              (* 1. Bedingung von Kriterium 1 für a. *)
  let addA(x :Int) :Int =  x + a      (* 2. Bedingung von Kriterium 1 für a. *)
  ...
  let var b = 2
  let updateB(y :Int) =  b := y       (* Kriterium 2 für b. *)
  ...
  let var c = 3
  let getA(var z :Int) :Ok =  z := a
  getA(c)                             (* Kriterium 3 für c. *)
end
```

Es verbleiben folgende Situationen, in denen eine Zellkonvertierung vermieden wird:

1. Die Variable ist weder lokal noch als globale Variable das Ziel einer Zuweisung, wobei ihr Wert in einer tiefer geschachtelten Funktion verwendet werden kann. In diesem Fall verhält sich die Variable gänzlich wie eine Konstante. Überflüssige Verwendungen des Schlüsselwortes **var** werden somit unterdrückt.

2. Die Variable findet ausschließlich innerhalb der lokalen Funktion Verwendung. Die Optimierung dieses Falles ist besonders lohnend, da intensiv genutzte Programmschleifen häufig stark lokal begrenzt sind.

Beispiele für diese Kriterien:

```
begin
  (* Kriterium 1 gilt für a: *)
  let var a = 1
  let addA(x :Int) :Int =  x + a
  ...
  (* Iterative Berechnung der n-ten Fibonacci-Zahl, *)
  (* Kriterium 2 gilt für low, high, dummy und i: *)
  let fibonacci(n :Int) :Int =
    begin
      let var low = 1
      let var high = 1
      for i = 1 upto 2 do
        let var dummy = high
        high := low + high
        low := dummy
      end
      high
    end
  end
```

Um die Überprüfung der oben genannten Kriterien zu unterstützen, enthält der Eintrag einer lokalen Variablen im Syntaxbaum ein Feld (namens *allocation*, siehe Abschnitt 4.3.3) folgenden Typs:

```
Def Allocation = Tuple
  case constant
  case mutable with
    var usedAsVarArgument, assignedAsGlobal,
        assignedAsLocal, usedAsGlobalValue :Bool
    var indirect :Bool
  end
end
```

Konstante Deklarationen werden durch die Variante *constant* repräsentiert, veränderlichen entspricht *mutable*. Das Feld *indirect* faßt die anderen vier als Endergebnis zusammen. Ob

eine Zellkonvertierung durchzuführen ist, wird in der nachstehend aufgeführten Prozedur bestimmt, welche bei jedem angewandten Auftreten einer lokalen Variablen aufgerufen wird. Auf diese Weise wird die für ein bestimmtes Teilproblem relevante Information an einer Stelle konzentriert. Zur Laufzeit erfolgt die Zuweisung an das abgeleitete Attribut a.*indirect* zwar bei jedem Aufruf, sie ist aber nicht zeitkritisch.

```
let updateAllocation(allocation :Allocation
                     context :Context  global :Bool) :Ok =
  case allocation
  when constant then
    (* Kontrolle der Einhaltung der Deklaration als Konstante: *)
    ASSERT(not(context?varArgument orif context?assignment))
  when mutable with a then
    case context
    when value, function, argument, indexed then
      if global then a.usedAsGlobalValue := true end
    when varArgument then
      a.usedAsVarArgument := true
    when assignment then
      if global then
        a.assignedAsGlobal := true
      else
        a.assignedAsLocal := true
      end
    end
    a.indirect := a.usedAsVarArgument orif a.assignedAsGlobal orif
                  {a.assignedAsLocal andif a.usedAsGlobalValue}
  end
```

Eine besondere Stärke von TL ist die Ermöglichung einer weitgehenden Zurückhaltung bezüglich Deklarationen veränderlicher Wertbindungen (vgl. Abschnitt 4.4.7). Die bisherigen Erfahrungen mit dem Quest-System zeigen, daß Programmierer tatsächlich recht sparsam mit dem Schlüsselwort **var** umgehen. Die gewählten Kriterien gestatten die Optimierung der am häufigsten auftretenden Fälle. Zur Verbesserung des Verfahrens müßte die Abfolge von Anweisungen in Betracht gezogen werden, was dann in Anbetracht des kombinatorischen Aufwandes wohl eine Überoptimierung wäre.

Wie folgende Überlegung beweist, ist die Existenz eines allgemeinen Verfahrens theoretisch ausgeschlossen. Die Frage, ob bestimmte Variablen in einer bestimmten Reihenfolge benutzt werden, impliziert die Frage, ob sie überhaupt benutzt werden. Diese läßt sich wiederum darauf zurückführen, ob bestimmte Programmverzweigungen benutzt werden. Die Lösung letzteren Problems impliziert jedoch offensichtlich die des bekannten Halteproblems.

4.4.6 Parallele Bindungen

In Sequenzen paralleler Bindungen erweitern die jeweils vorangehenden Bindungen den Sichtbarkeitsbereich der ihnen nachfolgenden nicht. Folgende Bindungssequenz gilt daher als Fehler:

```
let a = 1 and b = a
```

Die Bindung von *b* kann sich nicht auf a beziehen, weil a an dieser Stelle noch nicht sichtbar ist. Wie bei gewöhnlichen Bindungen sind jedoch am Ende einer Sequenz paralleler Bindungen *alle* von ihr gebundenen Bezeichner sichtbar. Deshalb sind folgende Anweisungen korrekt:

```
begin
  let a = 1 and b = 2
  a + b
end
```

Besondere Vorsicht ist geboten, wenn Bezeichner einander überdecken:

```
begin
  let a = 0
  ...
  let a = 1 and b = a
end
```

Die Bindung an *b* ist korrekt. Es ist jedoch zu beachten, daß sie sich auf die obere Variable a bezieht.

Während das akkumulierte Environment in der rekursiven Funktion *traverseBinding* sowohl den Sichtbarkeitsbereich für die Ausdrücke in den Einzelbindungen repräsentiert als auch das Funktionsergebnis liefert, sind diese beiden Aufgaben für parallele Bindungen zu trennen. Die Funktion *traverseParallelBindings* dupliziert daher das anfängliche Environment:

```
and traverseParallelBindings(state :State  recursion :Recursion
                             environment :Environment  context :Context
                             bindings :TLValue.Bindings)) :Environment =
  listParallelBindings(state recursion environment environment context bindings)
```

Die eigentliche Rekursion erfolgt dann in *listParallelbindings*:

```
and listParallelBindings(state :State  recursion :Recursion
                         environment, result :Environment
                         context :Context
                         bindings :TLValue.Bindings) :Environment =
  if list.empty(bindings) then
    result
  else
    let nextEnvironment =
   tlVariable.newAnonymousEnvironment(environment tlVariable.anonymousNoWhereIde)
    let e = traverseBinding(state recursion environment context list.head(bindings))
    let nextResult = list.cons(list.head(e) result)
    listParallelBindings(state recursion nextEnvironment nextResult
                         context list.tail(bindings))
  end
```

Der Parameter *environment* repräsentiert den Sichtbarkeitsbereich für die Einzelbindungen, während *result* das Funktionsergebnis akkumuliert.

Der Aufruf von *newAnonymousEnvironment* ist zwar für das isoliert betrachtete Backend konzeptionell unnötig. Er dient jedoch dazu, der vom Frontend vorgegebenen Zählweise[9] der de Bruijn-Indizes zu entsprechen. Deren Einhaltung ist für das Funktionieren der gesamten Allokationsphase kritisch. Da diesbezügliche Änderungen im Frontend sehr wahrscheinlich sind, ist hier die Modifizierbarkeit besonders wichtig.

Mit Hilfe der Konstruktorfunktionen für Environments (`new...Environment`) kann jede monotone Zählweise auf einfache Weise nachgebildet werden. Da Sichtbarkeitsbereiche stets monoton wachsen, ist die rekursive Programmierung unter Ausnutzung der differenzierten Parametrisierung adäquat.

Sollte zum Beispiel die Zählweise im obigen Fall später dahingehend geändert werden, daß alle Einzelbindungen direkt an der Stelle vor der Sequenz anknüpfen, so ist lediglich die Bindung an *nextEnvironment* zu entfernen und *environment* an Stelle von *nextEnvironment* zu verwenden.

4.4.7 Rekursive Bindungen

Eine grundlegende Forderung in TL ist der Ausschluß von Zugriffen auf uninitialisierte Werte. Vor allem deshalb wird in TL bei jeder Variablendeklaration die Angabe einer expliziten Initialisierung erzwungen. Die einheitliche Gültigkeit dieser Bedingung[10] ist eine notwendige Voraussetzung der Verschmelzung von Deklaration und Initialisierung zur „Bindung“[11].

Gewöhnliche und parallele Bindungen lassen, sofern sie syntaktisch korrekt sind, keinen Zugriff auf uninitialisierte Werte zu. In rekursiven Bindungen, die den Maßstäben des Frontends genügen, ist dies jedoch nicht immer der Fall, obwohl Initialisierungen anzugeben sind.

Zum Beispiel steht folgende einfache Anweisung im Widerspruch zur Semantik von TL:

let rec a :*Int* = a

Der Wert der Variablen a wäre zwangsläufig zum Zeitpunkt der Zuweisung uninitialisiert, weil er durch diese ja erst festgelegt werden sollte. Derartige zyklische Abhängigkeiten treten natürlich auch in beliebig komplexen und unübersichtlichen Konstrukten auf. Sie können sich z.B. über mehrere Variablen erstrecken:

let rec a :*Int* = *b* **and** *b* :*Int* = *c* **and** *c* :*Int* = a + *b*

[9] Auf die Gründe für diese Zählweise wird hier nicht eingegangen, weil sie allein das Fontend betreffen. Eine ausführliche Behandlung dieses Themas enthält [Matthes 92].

[10] Ihr wichtigster Vorzug ist die Unverletzlichkeit der Typsicherheit in Verbindung mit einer weitgehenden Unabhängigkeit von der Implementierung der (Frei-) Speicherverwaltung, bei der es sich typischerweise um eine sogenannte *garbage collection* handelt.

[11] In vielen anderen Zusammenhängen, wie z.B. bei der Betrachtung später Bindungen (*late Bindungs*) von Funktionen höherer Ordnung oder des Datenaustausches in verteilten Systemen sind die Begriffe „Benennung“ und „Bindung“ streng zu unterscheiden [Matthes 92]. In diesem Abschnitt wird auf eine diesbezügliche Differenzierung verzichtet, weil eine Benennung in einer expliziten statischen Bindung stets als Teil der Deklaration in der Bindung aufgeht.

Die Rekursion in einem Aggregat ist hingegen sinnvoll:

Let Rec *A* = **Tuple** *x* :*A* **end**
let rec a :*A* = **tuple** a **end**

Das intendierte Ergebnis entsteht durch folgende Anordnung der erforderlichen Schritte:

1. Es wird ein Tupel erzeugt.
2. Die Variable a wird an das Tupel gebunden.
3. Der Wert der Variablen a wird in das Tupel eingetragen.

Ähnliches gilt für Funktionsabstraktionen:

let rec *f* :**fun**(*n* :*Int*):*Int* = **if** *x* <= *0* **then** *1* **else** *n* * *f*(*n* − *1*) **end**

Wobei zur Laufzeit folgendes geschieht:

1. Es wird ein Speicherobjekt zur Repräsentation eines Funktionsabschlusses erzeugt.
2. Die Variable *f* wird an das Speicherobjekt gebunden.
3. Der Wert der Variablen *f* wird als globale Variable in das Speicherobjekt eingetragen.

Zur näheren Betrachtung werden zwei Klassen von Typen unterschieden: Typen, die durch Aggregation oder Funktionsabstraktion zusammengesetzte Werte beschreiben, heißen „komplex“, alle anderen „einfach“.

Die Klasse der einfachen Typen umfaßt damit die Basistypen und die durch Umbenennung von ihnen abgeleiteten. Werte werden ihrem Typ entsprechend als „einfach“ oder „komplex“ bezeichnet.

Komplexe Werte werden als sogenannte *boxed values* (vgl. [Peyton Jones 87; Cardelli 86b]), d.h. durch Speicherobjekte, implementiert und einfache Werte als *unboxed values*, d.h. durch einzelne Maschinenworte. Bei einfachen Werten ist eine zeitliche Trennung von Erzeugung und Initialisierung nicht möglich, weil ihre Identität durch den Initialisierungswert gegeben ist. Die Identität komplexer Werte hängt dagegen nur von ihrer Erzeugung ab, während die Initialisierungswerte darauf keinen Einfluß haben. Die Referenz eines Speicherobjektes kann direkt nach seiner Erzeugung verwendet und seine Initialisierung verzögert werden, wenn zwischenzeitliche Zugriffe auf seinen Inhalt ausgeschlossen sind.

Der Zugriff auf einen Wert geschieht in TL per Definition immer in einem Ausdruck. Ob in einem Ausdruck auf einen bestimmten Wert zugegriffen wird, läßt sich auf die Frage zurückführen, ob ein bestimmter Programmzweig zur Ausführung gelangt. Dieses Problem ist jedoch äquivalent mit dem semi-entscheidbaren Halteproblem. Da also kein erschöpfendes allgemeines Verfahren existiert, ist eine terminierende Bestimmung fehlerhafter Anweisungen nur anhand stärker einschränkender Kriterien möglich. Zu deren Formulierung dient folgende Definition:

> Eine Anweisung heiße „kritisch“, wenn ein Zugriff auf einen uninitialisierten Wert aufgrund der bewußt zu restriktiv gewählten Kriterien nicht ausgeschlossen werden kann.

Kritische Zugriffe auf den Inhalt von Aggregaten werden vermieden, indem ihre Felder gegebenenfalls nicht verwendet werden. Die Verwendung der in Funktionsabschlüssen aggregierten globalen Variablen hängt nicht nur vom jeweiligen Funktionsaufruf, sondern auch vom Funktionsrumpf ab. Letzterer ist in einem System, das Funktionen höherer Ordung unterstützt, aufgrund der häufigen späten Bindungen (*late bindings*) im allgemeinen nicht statisch analysierbar. Zugriffe auf nichtinitialisierte Funktionsabschlüsse werden daher pauschal unterbunden, indem ihre Aufrufe an allen Stellen verboten werden, die ohne Betrachtung des Funktionsrumpfes kritisch sind.

Die Analyse sollte möglichst wenige Anweisungen als kritisch einstufen. Andererseits müssen die den Kriterien entsprechenden Regeln möglichst einfach sein, damit sie von Programmierern leicht nachvollziehbar sind. Die Betrachtung von Anweisungsreihenfolgen würde dem widersprechen, weil Programmierer bei der Beurteilung der *statischen* Korrektheit eines Programmes zur Betrachtung seines *dynamischen* Verhaltens gezwungen wären.[12]

Das folgende Beispiel verdeutlicht diese Problematik:

```
Let T = Tuple x :Int end
let rec a :T = tuple b.x end
and b :T = tuple 3 end
```

Bei lexikalischer Ausführungsreihenfolge würde auf den uninitialisierten Wert $b.x$ zugegriffen. Die umgekehrte Reihenfolge wäre unkritisch. Das Vertauschen von Anweisungen kann jedoch problematisch sein, wie eine kleine Erweiterung des Beispiels zeigt:

```
Let T = Tuple x :Int end
let var g = 1
let rec a :T =
  begin
    let t = tuple b.x end
    callSideEffect(g)
    t
  end
and b :T = tuple let a = begin g := 2  3 end end
```

Die Funktion *callSideEffect* sei vorher definiert worden und habe einen Seiteneffekt, der von ihrem Argument g abhängt. Beim Vertauschen der Bindungen von a und b muß nun auch die Zuweisung an g berücksichtigt werden, weil *callSideEffect* sonst mit dem Argumentwert 2 statt 1 aufgerufen wird.

[12] Sicherlich kann auch der Standpunkt vertreten werden, daß ein Programmierer nicht in der Lage sein muß zu verstehen, warum sein Programm korrekt übersetzt worden ist. Dies trifft insbesondere auf Programme zu, die von Maschinen generiert worden sind und ausschließlich von solchen weiterverwendet werden. TL wurde jedoch in erster Linie für den Gebrauch durch menschliche Programmierer konzipiert, wobei besonderer Wert auf die Verständlichkeit von Programmen gelegt wird.

Offensichtlich lassen sich leicht noch wesentlich komplexere und unübersichtlichere Fälle konstruieren, in denen nicht ohne weiteres klar ist, welche Reihenfolgen unkritisch sind. Wie im Beispiel sind diese dann womöglich nicht einmal alle konsistent. Außerdem existieren rein syntaktisch korrekte Programme, in denen keine Reihenfolge unkritisch ist, was die ersten beiden Beispielen dieses Abschnittes beweisen. Auch die Verwendung von Funktionsabstraktionen oder Aggregaten schützt davor nicht, wie das folgende Beispiel zeigt:

Let T = **Tuple** x : *Int* **end**
let rec $f()$:*Int* = a.x
and a :T = **tuple let** $x = f()$ **end**

Die Funktion f muß zur Initialisierung des Tupels a aufgerufen werden, greift aber ausgerechnet auf das zu initialisierende Feld x zu.

Die Betrachtung von Reihenfolgen kann also im Einzelfall einfach sein, mündet aber leicht in uferlose Kombinatorik. Wird sie jedoch ausgeschlossen, so können *alle* Anweisungen durch eine terminierende statische Analyse als erlaubt oder kritisch klassifiziert werden. Diese ist in der Praxis mühelos nachvollziehbar. Außerdem ist nun jede Reihenfolge einer erlaubten Sequenz rekursiver Bindungen unkritisch. Beides ist der Einfachheit und der Verständlichkeit von Programmen zuträglich.

Zur Überprüfung einer rekursiven Bindung ist der zugewiesene Ausdruck gemäß seiner Konstruktion von außen nach innen rekursiv zu analysieren. Das Verfahren behandelt in jedem Schritt ein Konstrukt aus einer der folgenden drei Kategorien:

- Aggregationen.
- Funktionsabstraktionen.
- sonstige Konstrukte, die abgekürzt als S-Konstrukte bezeichnet werden. Beispiele: Funktionsaufrufe, Konstrollstrukturen (**if**, **while**...), Sequenzen (**begin** ... **end**), Feldoperationen ('.', '?', '!'), etc.

Einzeln stehende Bezeichner werden der Kategorie S-Konstrukt zugerechnet, wenn sie nicht direkt in ein Aggregat eingetragen werden.

Die von einer rekursiven Bindung gebundenen Bezeichner werden im folgenden kurz „R-Bezeichner“ genannt.

Unter Verwendung obiger Bezeichnungen gilt für die Überprüfung rekursiver Bindungen folgende Regel:

> Die R-Bezeichner einer rekursiven Bindungssequenz dürfen nicht innerhalb eines S-Konstruktes auftreten, das zur Erzeugung eines Bindungswertes der Sequenz ausgewertet wird.

Die Formulierung der Regel unterstreicht die Bedeutung dynamischer Abhängigkeiten. Dennoch ist die entsprechende Analyse rein statisch und ignoriert Auswertungsreihenfolgen in Anweisungssequenzen. Sie beschränkt sich vielmehr auf die Verfolgung der Schachtelungsreihenfolge von Konstrukten der genannten Kategorien.

Eine Sequenz rekursiver Bindungen ist wie folgt zu analysieren:

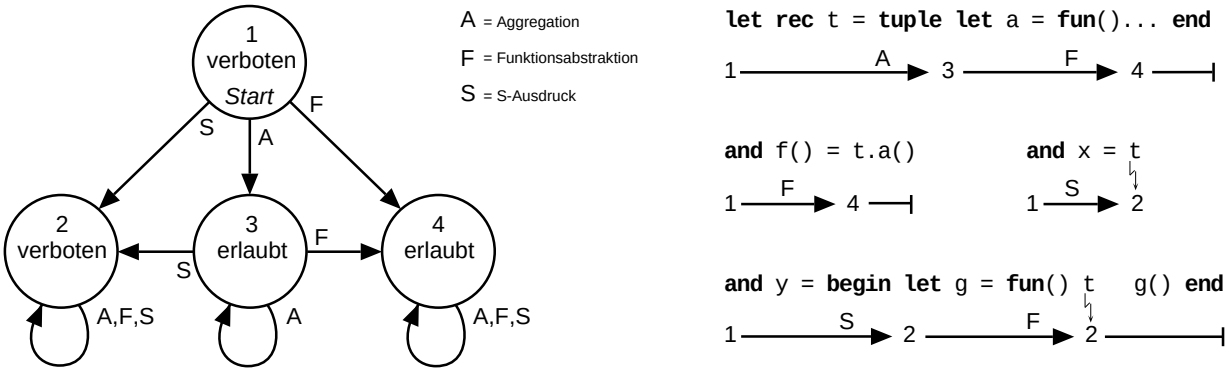

Abbildung 4.7: Die Überprüfung rekursiver Bindungen

- Zunächst werden die Variablen erfaßt, auf die sich die R-Bezeichner eindeutig beziehen. Sie werden als „R-Variablen“ bezeichnet.
- Die Wertausdrücke der Bindungen werden rekursiv von außen nach innen untersucht. In jedem Rekursionsschritt tritt genau ein Konstrukt aus einer der drei Kategorien auf.
- Es gilt stets einer der Modi „erlaubt“ oder „verboten“. Tritt eine R-Variable im Modus „verboten“ auf, so ist die gesamte Bindung kritisch.
- Die Analyse eines Bindungsausdruckes wird im Modus „verboten“ begonnen. Wenn im ersten Rekursionsschritt ein S-Ausdruck auftritt, wird im Modus „verboten“ verblieben, sonst zu „erlaubt“ gewechselt. R-Variablen können nur im Rahmen eines Aggregates oder einer Funktionsabstraktion erlaubt sein.
- Das Auftreten der drei Konstruktkategorien in den weiteren Schritten hat folgenden Einfluß:
  - Ein S-Konstrukt setzt den Modus „verboten“.
  - Tritt eine Funktionsabstraktion im Modus „erlaubt“ auf, so unterbleibt die Untersuchung des Funktionsrumpfes. Er kann einen beliebigen Ausdruck enthalten, weil die Funktion mit Sicherheit erst nach Beendigung der gesamten rekursiven Bindungssequenz aufgerufen werden kann.
  - Aggregate ändern den Modus nicht.
- Innerhalb eines nach dem ersten Rekursionsschritt analysierten Teilausdruckes kann der Modus „verboten“ nicht mehr zurückgesetzt werden, denn alle Teilausdrücke eines kritischen Ausdruckes sind ebenfalls kritisch.
- Nach der Rückkehr aus einem Rekursionsschritt wird der vorherige Modus wieder angenommen.

Abbildung 4.7 zeigt die Modi des Verfahrens als Zustände eines Automaten. Auf der rechten Seite der Abbildung sind in vier TL-Beispielen die jeweiligen Zustandsübergänge angegeben und zu ihren auslösenden syntaktischen Konstrukten in Beziehung gesetzt.

Zu beachten ist, daß für jede Gruppe von R-Variablen, die aus einer gemeinsamen Sequenz rekursiver Bindungen stammt, ein eigener Automat verwaltet wird, so daß beliebig geschachtelte Bindungen systematisch erfaßt werden. Dabei kann zur Vereinfachung ausgenutzt werden, daß ein Auftreten eines S-Konstruktes alle aktuellen Automaten in den Zustand 4 (Modus „verboten“) versetzt.

Die elementaren Operationen zur Überprüfung rekursiver Bindungen sind in einem eigenen Modul (tlCheckRec:TLCheckRec) gekapselt, dessen Schnittstelle und Implementation in Anhang B.1 aufgeführt sind. Der Rahmenalgorithmus, der auf den Operationen in *tlCheckRec* aufbaut, ist mittels eines Parameters (*recursion :Recursion*) der Knotenfunktionen in den allgemeinen rekursiven Ablauf der Allokationsphase eingebettet.

Die Allokation rekursiver Bindungen wird von folgender Funktion geleistet, die gleichzeitig die Überprüfung der R-Variablen veranlaßt:

```
and traverseRecBindings(state :State  recursion :Recursion
                        environment :Environment  context :Context
                        bindings :TLValue.Bindings) :Environment =
  begin
    (* Environment um die R-Variablen erweitern: *)
    let result =
      listRecBindingEnvironments(state  environment  context  bindings)
    (* R-Variablen erfassen: *)
    let variables :Iter.T(Variable)=
      iter.map(list.elements(bindings)  fun(binding :TLValue.Binding)
        case binding when valueCase, valueTypeCase with b then
          let declaration = list.head(optional.value(b.allocation))
          declaration!variable.variable
        end)
    let r = tlCheckRec.newVariables(recursion  variables)
    (* Wertausdrücke mit bereits erweitertem Environment allozieren und
       mit ebenfalls erweitertem Rekursionskontext überprüfen: *)
    listRecBindingValues(state  r  result  context  bindings)
    result
  end
```

Die Funktion *listRecBindingEnvironments* alloziert die Variablen der R-Bezeichner und erweitert das Environment (sie ist einfach rekursiv und verwendet *allocateBinding*, siehe Abschnitt 4.4.3). Die Ausdrücke der Bindungswerte werden erst danach von *listRecBindingValues* durchlaufen. Dadurch sind in *jedem* dieser Ausdrücke *alle* gebundenen Bezeichner sichtbar. In der Mitte wird eine Liste[13] der R-Variablen gebildet und der Funktion *tlCheckRec.newVariables* übergeben, die einen erweiterten Rekursionskontext (*r*) erzeugt. Dieser dient danach in der Funktion *listRecBindingValues*, zur Überprüfung des Auftretens der aktuellen R-Variablen:

[13] Zur Repräsentation dieser Liste wird der Datentyp *Iter.T* verwendet. Er stammt aus einer Tycoon-Standardbibliothek, in der bereits alle benötigten Operationen implementiert sind.

```
and listRecBindingValues(state :State  recursion :Recursion
                         environment :Environment  context :Context
                         bindings :TLValue.Bindings) :Ok =
  if not(list.empty(bindings)) then
    case list.head(bindings) when valueCase, valueTypeCase with b then
      (* Rekursive Bindungen sind nie anonym: *)
      ASSERT(not(tlIde.isAnonymous(b.ide.name)))
      traverseValue(state recursion environment valueContext b.value)
    end
    (* Nur zur Anpassung an die Zählweise des Frontends für de Bruijn-Indizes: *)
    let e = tlVariable.newAnonymousEnvironment(environment tlVariable.anonymousNoWhereIde)

    listRecBindingValues(state recursion e context list.tail(bindings))
  end
```

Der Rekursionskontext ist durch den Parameter *recursion* angegeben. Er wird an die Funktion *traverseValue* weitergereicht, die die Wertausdrücke (*b.value*) der Bindungen (*bindings*) rekursiv durchläuft.

Folgende Funktionen bauen jeweils einen abgeleiteten Rekursionskontext auf, der zwar dieselben Listen von R-Variablen weiterverwendet, aber an das aktuelle Konstrukt angepaßte Modi enthält:

enterAggregate wird beim Auftreten eines Aggregates aufgerufen.

enterFunction wird beim Auftreten einer Funktionsabstraktion aufgerufen.

enterExpression wird beim Auftreten eines S-Konstruktes aufgerufen.

Der eigentliche Test geschieht bei jeder syntaktischen Verwendung eines Bezeichners als Wert, also in der Funktion *traverseIde*. Dazu muß auf die Deklaration der Variable zugegriffen werden, auf die sich der Bezeichner bezieht. Diese muß in *traverseIde* ohnehin bestimmt werden, so daß sich die erwähnte Verknüpfung der Algorithmen anbietet. Der folgende Auszug zeigt die relevanten Anweisungen:

```
traverseIde(state :State  recursion :Recursion
            environment :Environment
            context :Context  value :TLValue.T) :Bool =
  begin
    ...
    case value when ideCase with v then
      let origin = tlVariable.lookupDeBruijnIndex(environment  v.ideRef.index)
      let declaration = list.head(origin)
      if tlCheckRec.variableMisused(recursion  declaration!variable.variable) then
        error(...) (* Es ist eine der hier verbotenen R-Variablen. *)
      end
      (* Es ist entweder keine R-Variable oder sie ist hier erlaubt. *)
      ...
```

4.5 Die Generierungsphase

In der Generierungsphase wird erneut der gesamte Syntaxbaum rekursiv durchlaufen und dabei TML-Kode erzeugt. Eine Überprüfung semantischer Fehler findet nicht mehr statt. Die als Attribute des Syntaxbaumes verfügbaren Ergebnisse der Allokationsphase schreiben die Ausprägung fast aller TML-Instruktionen exakt vor. Lediglich die Indizes der lokalen Variablen sind noch zu bestimmen (siehe Abschnitt 4.5.2).

4.5.1 Die Programmorganisation der Generierungsphase

Wie auch in der Allokationsphase ist die Methode des rekursive Abstiegs für das Programmschema bestimmend (vgl. Abschnitt 4.2). Die rekursiven Funktionen zur Bearbeitung der wichtigsten Knotentypen sind:

- ▷ *translateBindings* für Bindungssequenzen mit lexikalischer Anordnung der Sichtbarkeitsbereiche.
- ▷ *traverseBinding* für den Typ *TLValue.Binding*, der sowohl Einzelbindungen beschreibt, als auch die Verzweigungen in parallele oder rekursive Bindungen.
- ▷ *traverseRecBindings* für Sequenzen rekursiver Bindungen.
- ▷ *traverseRecBinding* für den Typ *TLValue.Binding*, wenn er innerhalb einer Sequenz rekursiver Bindungen auftritt.
- ▷ *translateValue* für den Typ *TLValue.Value*.
- ▷ *translateIde* für die Variante *ideCase* des Typs *TLValue.Value*.

Die einfache Anpaßbarkeit der Programmstruktur an die Struktur der zu bearbeitenden Daten (den Syntaxbaum) kommt auch diesmal voll zur Geltung. Um die Vorteile uneingeschränkt nutzen zu können, müssen allerdings spezielle Techniken zur Konstruktion der Zwischenergebnisse und des Endergebnisses angewandt werden.

Bei der Übersetzung gewöhnlicher Bindungen wäre die Bereitstellung der Teilergebnisse durch Ausgabeparameter oder Rückgabewerte adäquat, weil ihre Verknüpfung synchron zur rekursiven Programmstruktur erfolgen kann. Diese Methode ist jedoch aufgrund der notwendigen strukturellen Analogien der Eingabe- (Syntaxbaum) und Ausgabedaten (Kode) wenig flexibel.

Ein alternativer Ansatz ermöglicht eine unabhängigere Anordnung der Teilergebnisse:

> Die Ergebnisausgabe erfolgt durch Seiteneffekte. Dadurch werden wechselseitige Abhängigkeiten zwischen der Konstruktion des Ergebnisses und dem Durchlaufen der Eingabedaten vermieden.

Es kommen zwei unterschiedliche Programmierstile zum Einsatz, die sich unter den gegebenen Bedingungen orthogonal verhalten: funktionale und imperative Programmierung.[14]

[14] Ein rein im funktionalen oder im imperativen Stil gestaltetes Programm wäre erheblich komplizierter als das in dieser Arbeit vorgestellte. Jeder Programmierstil ist zur Lösung bestimmter Probleme besonders adäquat. In TL sind mehrere Stile integriert, so daß die *gleichzeitige* Nutzung ihrer Vorteile innerhalb *eines* Programmes möglich ist. Hybride Systeme, deren Konstituenten jeweils nur einen Stil unterstützen, werden dieser Anforderung meist nicht gerecht.

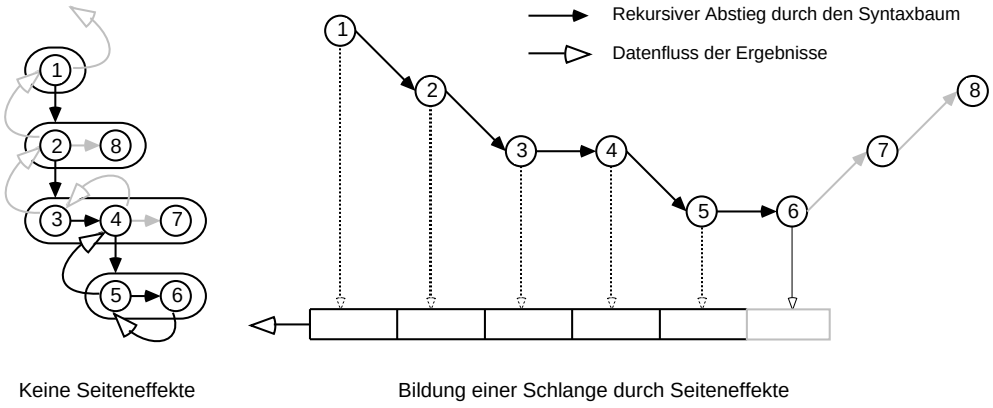

Abbildung 4.8: Alternative Methoden der Bereitstellung von Ergebnissen des rekursiven Abstiegs

In der Allokationsphase ist dieser Ansatz auf besonders einfache Weise verwirklicht worden: ihre wesentlichen Ergebnisse werden per Zuweisung als Attribute in den bereits vorhandenen Syntaxbaum eingetragen.[15]

Der als Ergebnis der Generierungsphase erzeugte Kode ist jedoch vollkommen eigenständig und muß daher selbsttragend konstruiert werden. Dabei treten häufig sequentielle Strukturen auf, zu deren Akkumulation der abstrakte Datentyp „Schlange“ (*queue*)[16] verwendet wird. Die zu erweiternden Schlangen werden beim rekursiven *Abstieg* jeweils als Eingabeparameter (*call-by-value*-Semantik) der Knotenfunktionen weitergereicht. Dabei entsteht kein explizit aufwärts orientierter Datenfluß, der die Organisation wesentlich beeinflussen würde.

Abbildung 4.8 zeigt zwei schematische Darstellungen einer typischen rekursiven Datenstruktur des Syntaxbaumes. Sie besteht aus 8 Knoten, die im rekursiven Abstieg in der durch die Numerierung angegebenen Reihenfolge besucht werden. Beide Darstellungen zeigen die Situation nach Erreichen des sechsten Knotens.

Auf der linken Seite wird die notwendige Rückpropagierung der Ergebnisse bei einem Verzicht auf Seiteneffekte dargestellt. Diese Methode ist adäquat, wenn die Teilergebnisse zu einer rekursiven Struktur verknüpfen werden, die der des Syntaxbaumes ähnelt. Sie wird deshalb in folgenden wichtigen Aufgabenbereichen des Backends angewandt:

- Bei der Konstruktion der Environments in der Allokationsphase (siehe Abschnitt 4.4.2).
- Bei der TML-Generierung für Wertausdrücke (siehe Abschnitt 4.5.5.2).

Wie zuvor erwähnt, kann eine flache sequentielle Struktur jedoch wesentlich einfacher durch Seiteneffekte erstellt werden. Auf der rechten Seite sind die gleichen 8 Knoten des Syntaxbaumes in aufgelockerter Anordnung abgebildet, um den Aspekt der sequentiellen Abarbeitung

[15] Dies gilt für die Allokation von Werten uneingeschränkt. Environments werden außerdem durch seiteneffektfreie funktionale Programmierung verknüpft (siehe Abschnitt 4.4.2).

[16] Dieser Datentyp wird meist „Warteschlange“ genannt, eine Bezeichnung, die z.B. im Bereich der Betriebsmittelverwaltung zutreffend ist. Er ist jedoch auch in vielen Situationen nützlich, in denen der Aspekt des Wartens nicht im Vordergrund steht.

hervorzuheben. Der ebenfalls sequentielle Charakter der unten abgebildeten Schlange stimmt auf natürliche Weise mit dieser Vorgehensweise überein. Die Schachtelung wird jedoch aufgehoben, so daß eine flache Struktur entsteht. Dieser Effekt ist eine der wichtigsten strukturellen Veränderungen im Übergang von TL zu TML. Er wird in folgenden Bereichen genutzt:

- Bei der Übersetzung von Bindungen (siehe Abschnitt 4.5.7). Diese treten im allgemeinen als geschachtelte Sequenzen auf. Aufgrund der dominierenden Rolle von Bindungen in TL hat die Lösung dieser Aufgabe Einfluß auf die gesamte Übersetzung.
  Die Umsetzung in flache Listen ist eine Voraussetzung für die Zuordnung von Argumenten in Funktionsaufrufen. Sie vereinfacht außerdem den Kode, was eine erhöhte Effizienz der Interpretation, geringeren Speicherplatzbedarf, und geringeren Aufwand bei der Reflektion zur Folge hat.
- Bei der Erstellung der Literalvektoren (siehe Abschnitt 4.5.3).
- Bei der Anordnung der Fallmarken und Verzweigungen in Fallunterscheidungen (siehe Abschnitt 4.5.8).

4.5.2 Die Verwaltung der lokalen Variablen

Die Indizes der in der Allokationsphase angelegten lokalen Variablen sind noch nicht festgelegt. Dadurch kann die Generierungsphase, die zusätzliche lokale Variablen als Hilfsvariablen einführt, die Indizierung aller lokalen Variablen optimierend durchführen.

Eine Hilfsvariable wird zum Beispiel bei der Erzeugung eines unbenannten komplexen Objektes verwendet, um zum Zwecke der Initialisierung auf das Objekt zugreifen zu können. Abbildung 4.9 zeigt zur Veranschaulichung den TML-Kode folgender Anweisungen:

```
let f(t :Tuple :Int end) = ...
f(tuple 7 end)
```

Das unbenannte Tupel wird durch ein Speicherobjekt mit zwei Feldern repräsentiert, wobei das erste Feld die Fallmarke 1 (siehe Abschnitt 4.4.4) und das zweite Feld die 7 zugewiesen bekommt. Die **local**-Instruktion wird von den beiden Zuweisungen geteilt.

Die Funktion *newLocalIndex* liefert den aktuellen Index für eine lokale Variable. Sie wird an zahlreichen Stellen beim Anlegen von Hilfsvariablen eingesetzt. Außerdem wird sie von der Funktion *allocationNewLocalIndex* aufgerufen, die den Index für bereits in der Allokationsphase angelegte lokale Variablen in das entsprechende Attribut des Syntaxbaumes einträgt.

Zur optimierenden Verwaltung des aktuellen Indizes dient der Parameter *state* der betroffenen Knotenfunktionen. Er hat folgenden Typ:[17]

```
Let State = Tuple
  literals :queue.T(TML.Value)
  var nLocals :Int
  var localIndex :Int
end
```

[17] Zu Feld *literals* siehe Abschnitt 4.5.3.

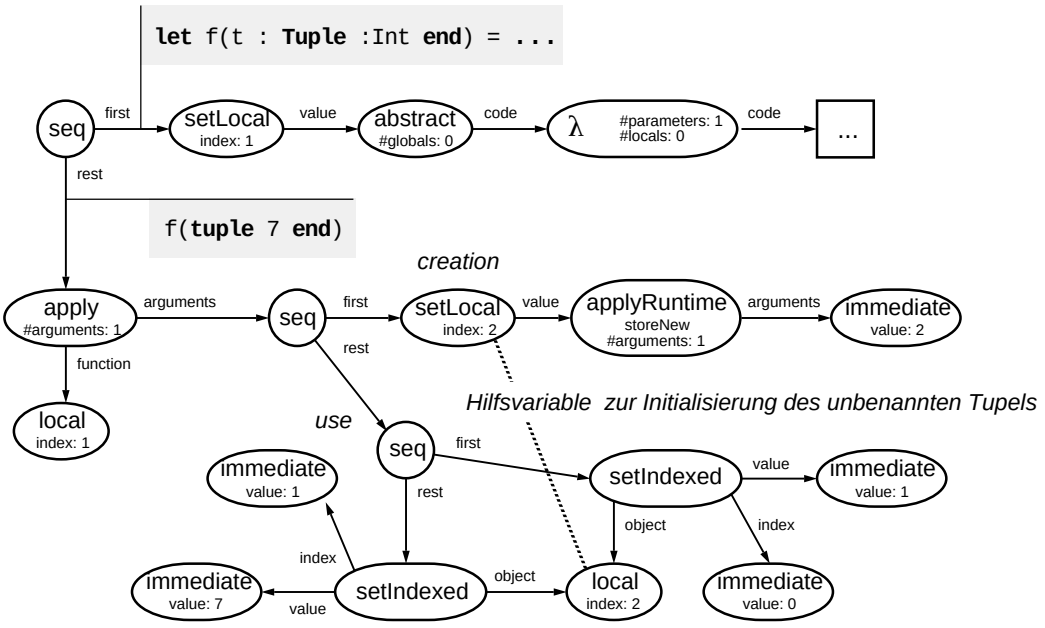

Abbildung 4.9: Ein Beispiel für den Einsatz einer Hilfsvariablen

Für jede übersetzte Funktion wird ein neuer *State* erzeugt. Das Feld *localIndex* wird durch jeden Aufruf von *newLocalIndex* inkrementiert. Die Wiederverwendung gleicher Indizes in getrennten Sichtbarkeitsbereichen (z.B. in verschiedenen Zweigen einer Fallunterscheidung) wird durch folgende Funktionen unterstützt:

let *markLocals(state :State) :Int = state.localIndex*

let *releaseLocals(state :State localIndex :Int)* :**Ok** =
state.localIndex := localIndex

TL ist bezüglich der Semantik lokaler Variablen nahe mit anderen blockorientierten Sprachen wie z.B. Pascal, C, Modula-2 oder Ada verwandt. In deren Übersetzern wird häufig die hier beschriebene Methode angewandt. In der TL-Syntax treten Blöcke als Bindungssequenzen auf. Um alle lokalen Variablen zu erfassen, ist zusätzlich die Betrachtung bestimmter Ausdrücke, wie z.B. Fallunterscheidungen, in denen Hilfsvariablen eingeführt werden, notwendig. Es ist jedoch am einfachsten und nicht zuletzt am sichersten, alle Ausdrücke pauschal zu erfassen. In jedem Fall ist der Sichtbarkeitsbereich einer lokaler Variablen entweder durch die Grenzen eines Ausdruckes oder das Ende einer Bindungssequenz begrenzt. Deshalb werden obige Funktionen an folgenden zwei Stellen eingesetzt:

```
let rec translateBindings(... ) :TML.T =
  begin
    let m = markLocals(state)
    ... (* hier steht die eigentliche Übersetzung einer Bindungssequenz,
           insbesondere das Durchlaufen der Bindungen *)
    releaseLocals(state m)
    ... (* hier steht die Aufbereitung des Ergebnisses *)
  end

and translateValue(... ) :Value =
  begin
    let m = markLocals(state)
    let result = ... (* hier steht die Übersetzung eines Wertes,
                        insbesondere das Durchlaufen des Ausdrucks *)
    releaseLocals(state m)
    result
  end
```

Durch die Verwendung der lokalen Variablen m wird der Stapelspeicher des Übersetzerprogrammes als *working stack* [Jähnichen et al. 78] ausgenutzt.

Der maximale im Feld *localIndex* erreichte Index wird beim Aufruf von *newLocalIndex* im Feld *nLocals* erfaßt, die am Ende der Übersetzung einer Funktion die zu allozierende Gesamtanzahl lokaler Variablen anzeigt.

Abbildung 4.10 zeigt anhand eines TL-Beispiels das Prinzip der durch den Stapelmechanismus ermöglichten Wiederverwendung von Indizes. Als Anzahl der anfangs belegten Indizes kann eine beliebige Zahl angenommen werden. Zu beachten ist, daß die *mark* und *release*-Aktionen immer paarweise auftreten.

4.5.3 Die Akkumulation der Literale

Aufgrund ihrer Zuordnung zu Funktionen werden Literale (siehe Abschnitt 3.2) mit Hilfe des in Abschnitt 4.5.2 auf Seite 73 beschriebenen Parameters *state* verwaltet. Im Feld *literals* werden alle Literale einer Funktion durch eine Schlange sequentiell akkumuliert. Ihre Reihenfolge ist prinzipiell beliebig. Sie muß lediglich mit den bei der Kodeerzeugung vergebenen Indizes übereinstimmen. Da diese synchron mit der Erfassung der Literale verläuft, ist dies sehr einfach einzuhalten: als Index wird die aktuelle Länge der Schlange gewählt.

Zur Veranschaulichung dient die rechte Seite von Abbildung 4.8. Ein Wert *value* des Typs *TML.Value* wird durch folgende Anweisung in die aktuelle Literalschlange eingefügt:

```
queue.push(state.literals value)
```

Die Umwandlung einer fertigen Literalschlange in einen Literalvektor leistet eine einfache Funktion:

```
let queueLiterals(literals :queue.T(TML.Value)) :TML.Literals =
  arrayOp.create(:TML.Value queue.elements(literals))
```

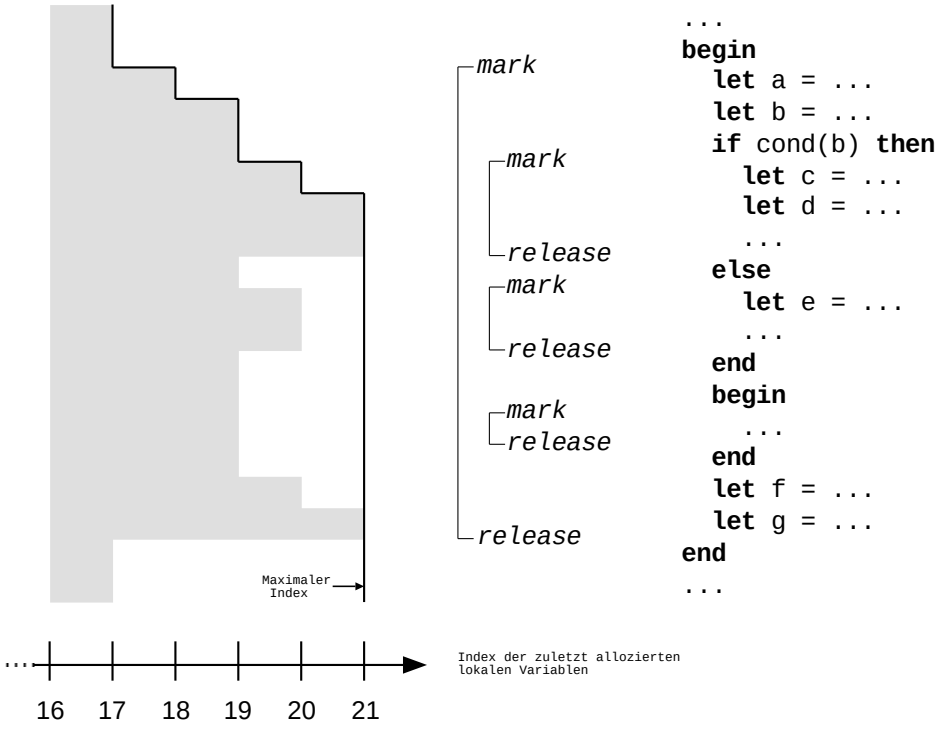

Abbildung 4.10: Die Wiederverwendung lokaler Variablen

Die Konvertierung zwischen zwei verschiedenen Massendatentypen kann in Tycoon auf einfache Weise unter impliziter Verwendung des Moduls *iter:Iter* geschehen. Jede Standardbibliothek eines Massendatentyps[18] enthält zwingend zwei Funktionen, die den Typ in Beziehung zu Iterationen setzen. Sei zum Beispiel M ein Massendatentyp, dann lauten die entsprechenden Signaturen:

- *create(A::TYPE from :Iter.T(A)) :M(A)*
  Diese Funktion konstruiert eine Datenstruktur vom Typ M mit den Elementen der Iteration *from*. Die einzelnen Elemente vom Typ A bleiben unverändert. Die Reihenfolge wird beibehalten, sofern dies sinnvoll ist (bei Mengen zum Beispiel nicht). Die Zeitkomplexität der Operation ist linear.

- *elements(A ::TYPE m :M(A)) :Iter.T(A)*
  Diese Funktion ist quasi die Umkehrung der zuerst genannten. Sie erzeugt eine Iteration, die auf der Massendatenstruktur m basiert. Die einzelnen Elemente vom Typ A bleiben wiederum unverändert. Eine durch den Typ M bedingte Reihenfolge wird gegebenenfalls beibehalten. Die Zeitkomplexität der Operation ist konstant.

Aus der Zusammensetzung der Funktion *queueLiterals* folgt, daß ihre Zeitkomplexität linear ist. Ihre Anwendungen in der Generierungsphase erhöhen den gesamten Aufwand zum Erfassen der Literale nur um einen kleinen Faktor, der in der Größenordnung von zwei liegt, denn jedes Literal wird genau zweimal bearbeitet: Bei der Akkumulation durch *queue.push* und bei der Konvertierung durch *queueLiterals*.

Diese Erhöhung des Laufzeitaufwandes gegenüber dem theoretisch minimal erforderlichen ist unkritisch und durch den Vorteil einer übersichtlichen und flexiblen Programmierung gerechtfertigt.

Die Akkumulation eines Literals und die Erzeugung seines Verwendungskodes werden durch folgende Funktion zusammengefaßt:

```
let pushLiteral(state :State  value :TML.Value) :TML.T =
  begin
    let index = queue.size(state.literals)
    queue.push(state.literals  value)
    tml.newLiteralCode(index)
  end
```

Dadurch ist bereits eine transparente Schnittstelle für spätere Optimierungen vorgegeben. Die Funktion *pushLiteral* könnte z.B. Literale gleichen Inhalts in einer einzigen Kopie zusammenfassen und dementsprechend den gleichen Index wiederverwenden.

[18] Genaugenommen sind es ausnahmslos Typoperatoren, die ganze Klassen von Typen, sogenannte „Kinds“ [Cardelli 89], beschreiben. Bei einem Massendatentyp handelt es sich in der Regel um einen lediglich durch den Elementtyp parametrisierten abstrakten Datentyp, wie z.B. Liste, Kellerspeicher, Schlange oder Menge. Durch Abstraktion vom Elementtyp und Konzentration auf die Eigenschaften Konstruktortyps kommt es zu der vereinfachenden Bezeichnung *Typ*.

4.5.4 Die Kodegenerierung für veränderliche Datenobjekte

TL hat ein konventionelles Variablenkonzept: Eine Variable beansprucht einen Speicherplatz in einem (logischen) Speichermedium. An diesem Platz ist ihr aktueller Wert gespeichert.

Die Angabe des Speicherplatzes kann wiederum als Wert betrachtet werden. Deshalb wird sie als *location value*, abgekürzt *l-value*, bezeichnet. Der Wert, der als Inhalt der Variablen angesehen wird, heißt *read value*, bzw. *r-value* [Bause, Tölle 90]. Eine weniger sachliche, aber einprägsamere Herleitung [Aho et al. 86] lautet:

> Auf der linken Seite einer Zuweisung wird stets ein *l-value* verwendet, *r-values* können nur auf der rechten Seite auftreten.

Im folgenden werden die deutschen Äquivalente „L-Wert“ und „R-Wert“ verwendet.

In TL treten L-Werte in folgenden Situationen auf:

- ▷ in Bindungen veränderlicher Variablen.
  let var a = *1*
- ▷ in Zuweisungen.
  a := *2*
- ▷ bei der Verwendung veränderlicher Parameter, bzw. Argumente.
  let *f*(**var** *x* :*Int*):*Int* = *x* := *3*
  f(a)

In Systemen, die alle Variablen im Hauptspeicher abbilden (z.B. Modula-2, C), werden L-Werte durch Hauptspeicheradressen repräsentiert. Im Tycoon-System werden Variablen jedoch auch im Objektspeicher angelegt. Der naheliegende Ansatz, Objektreferenzen als Ersatz für Adressen einzusetzen, ist nicht allgemein genug, weil auch Felder in Aggregaten veränderlich sein können:

let *t* = **tuple let var** a = *1* **end**
t.a := *2*

Die Objektspeicherschnittstelle TSP bietet als einzige Möglichkeit für die Manipulation der Elemente eines Speicherobjektes den indirekt indizierten Zugriff. Der L-Wert eines Feldes in einem Aggregat muß daher aus zwei Komponenten bestehen: der Referenz des Speicherobjektes und dem Index des Feldes.

Der Einsatz von Funktionen höherer Ordnung mit veränderlichen Parametern erzwingt die Verwendung einer einheitlichen Repräsentation von L-Werten. Denn unter dieser Voraussetzung kann im allgemeinen weder die Struktur der Argumente noch die Wirkungsweise einer aufgerufenen Funktion statisch bestimmt werden.

Deshalb wird auch für einfache Variablen, die nicht Teil eines Aggregates sind, ein Speicherobjekt (*cell*, vgl. [Kranz et al. 86; Peyton Jones 87; Cardelli 86b]) angelegt, wenn sie potentiell als veränderliches Argument übergeben werden. Diese Methode wird „Zellkonvertierung“ (*cell conversion*) genannt. Der L-Wert dieser Variablen besteht aus ihrer Objektreferenz und der Konstanten 0, die den Index des „einzigen Feldes des Speicherobjektes“ darstellt.

Die doppelwertige Repräsentation von L-Werten bedingt keinen Bruch mit der Semantik von TL, denn im Gegensatz zu vielen anderen Sprachen können L-Werte in TL nicht explizit als R-Werte verwendet werden. In den oben erwähnten Sprachen Modula-2 und C geschieht dies durch die Bestimmung der Hauptspeicheradresse einer Variablen (mittels **ADR**, bzw. &) und deren Weiterverwendung, etwa als Zeiger (*pointer*).

Der Verlust an Ausdruckskraft durch den Verzicht auf Zeiger wird in TL durch das leistungsfähige Typsystem weitgehend kompensiert. Zeiger dienen meistens der Bildung rekursiver Datenstrukturen. Diese können in TL durch rekursive Typen modelliert und mittels rekursiver Bindungen (vgl. Abschnitt 4.4.7), deren statisch zugesicherte Typsicherheit unverletzlich ist, konstruiert werden. Außerdem werden fehlerhafte Referenzen (sogenannte *dangling pointers*) vollständig vermieden.

Das beschriebene einheitliche Schema zur Repräsentation von L-Werten kann für alle in TL auftretenden veränderlichen Variablen eingesetzt werden. Es gibt jedoch Situationen, in denen es ohne Fehlerrisiko durch eine effizientere Implementation ersetzt werden kann, die anstelle der Objektspeicherschnittstelle reine Hauptspeicheroperationen verwendet. Eine entsprechende statische Analyse zur Vermeidung der Zellkonvertierung lokaler veränderlicher Variablen wird in Abschnitt 4.4.5 erläutert.

L-Werte können nur implizit durch folgende syntaktischen Einheiten angegeben werden:

- ▷ Bezeichner.
- ▷ indizierte Zugriffe auf Arrays.
- ▷ Feldzugriffe auf Tupel.

Die Funktion, die eine Repräsentation eines L-Wertes erzeugt, bildet diesen Zusammenhang direkt ab:

```
and translateAddress(state :State  value :TLValue.T) :Address =
  case value                                  (* Syntaktisches Element:... *)
  when ideCase with v then                    (* ... Bezeichner => Variable *)
    translateVariableAddress(optional.value(v.allocation))
  when indexCase with v then                  (* ... indizierter Array-Zugriff *)
    tuple case indexed of Address with
      let object = translateAllValue(state  v.value)
      let index = translateAllValue(state  v.index)
    end
  when fieldCase with v then                  (* ... Tupelfeldzugriff *)
    tuple case indexed of Address with
      let object = translateAllValue(state  v.value)
      let index = newIntCode(v.ideRef.index)
    end
  end
```

Die Funktion *translateAllValue* erzeugt Kode für einen R-Wert, *newIntCode* für eine Ganzzahl. Die Übersetzung des L-Wertes einer beliebigen Variablen leistet die Funktion *translateVariableAddress*, in der alle notwendigen Fallunterscheidungen vorgenommen werden.

Der Typ *Address* enthält zwei Varianten:

```
Let Address = Tuple
  case local with index :Int end
  case indexed with object, index :TML.T end
end
```

Die Variante *local* repräsentiert L-Werte lokaler Variablen ohne Zellkonvertierung durch die Angabe ihrer Indizes; die Variante *indexed* realisiert das einheitliche Repräsentationsschema, das für alle anderen Fälle gilt: Das Feld *object* enthält die Referenz des Speicherobjektes und *index* gibt den Index innerhalb des Speicherobjektes an.

Eine Fallunterscheidung anhand der Varianten des Typs *Address* erfolgt z.B. bei der Übersetzung von Zuweisungen:

```
and translateAssignment(state :State  bindings :TLValue.Bindings) :TML.T =
  begin
    let addressBinding = ...  (* Der linken Seite der Zuweisung Entsprechendes *)
    let value = ...           (* Der rechten Seite der Zuweisung Entsprechendes *)
    case addressBinding when valueCase, valueTypeCase with b then
      case translateAddress(state b.value)
      when local with a then
        tml.newSetLocalCode(a.index  value)            (* Direkte Addressierung *)
      when indexed with a then
        tml.newSetIndexedCode(a.object  a.index  value) (* Indirekte Addressierung *)
      end
    end
  end
```

Durch Erweiterung des Typs *Address* könnten auf einfache Weise weitere Schemata zur Repräsentation von L-Werten realisiert werden. Diese Anpaßbarkeit ist z.B. im Hinblick auf spätere zusätzliche Optimierungen, die L-Werte involvieren (wie z.B. lokale Aggregate), nützlich.

Die Repräsentation veränderlicher globaler Variablen orientiert sich am allgemeinen Schema für L-Werte. Durch folgende Konvention ist in allen Situationen eine eindeutige Zuordnung möglich:

> Die zwei Werte einer veränderlichen globalen Variablen haben innerhalb eines Funktionsabschlusses stets direkt aufeinander folgende Positionen, wobei die Objektreferenz dem Objektindex voransteht.

Eine analoge Anordnung gilt auch für veränderliche Parameter. Die Anzahl der in der Signatur einer Funktion angegebenen Parameter erhöht sich pro Angabe des Schlüsselwortes **var** jeweils um Eins.

Argumente von Funktionsaufrufen sind im Syntaxbaum als Bindungssequenz repräsentiert. In den Einzelbindungen befinden sich Einträge (*isVarArgument*), die angeben, ob es sich um eine Übergabe an einen veränderlichen Parameter handelt. Wenn dies der Fall ist, wird

die Funktion *translateVariableAddress* verwendet, um die Objektreferenz und den Index der übergebenen Variablen zu bestimmen. Daraufhin wird Kode für die Übergabe dieser *zwei* Werte erzeugt.

Abbildung 4.11 zeigt einige L-Werte in verschiedenen Einsatzbereichen. Zum Beispiel wird für $t.a$ und g in dem Funktionsaufruf

$$f(t.a \;\; g \;\; t.b)$$

jeweils der L-Wert übergeben und für $t.b$ der R-Wert. Die Zellkonvertierung der lokalen Variablen g wird im oberen Bereich gezeigt.

4.5.5 Die Kodegenerierung für Wertausdrücke

Da die Teilsyntax für Wertausdrücke sehr reichhaltig ist, sind die diesem Bereich zugeordneten Routinen entsprechend umfangreich. Sie werden durch eine spezielle Schnittstelle gegen Änderungen der Syntax oder der Semantik von Bindungen abgekapselt.

4.5.5.1 Die interne Zwischenrepräsentation des Kodes für Wertausdrücke

Die in Abschnitt 4.4.7 beschriebene Trennung der Erzeugung und Initialisierung von R-Werten in rekursiven Bindungen bedingt, daß der Kode für R-Werte zumindest in bestimmten Fällen nicht als atomare Einheit betrachtet werden kann. Seine Bestandteile werden in einer speziellen Zwischenrepräsentation zusammengefaßt, um dennoch alle R-Werte durch ein einheitliches Programmschema erfassen zu können.

Die einzige Alternative dazu wäre das sogenannte „Ausprogrammieren“: Für jeden betroffenen Knotentyp wären mehrere Funktionen erforderlich, die zwar in der Ergebniskonstruktion variieren würden, aber die gleichen Fallunterscheidungen beim Durchlaufen des Syntaxbaumes durchzuführen hätten. Dadurch würden zahlreiche Teilprobleme vervielfältigt und verteilt. Dieser Umstand ist für die spätere Modifizierbarkeit und Wartung des Programmes kritisch.[19]

Wie in Abschnitt 4.4.7 beschrieben, sind zwei Arten von R-Werten (in diesem Abschnitt einfach „Werte“ genannt) zu unterscheiden:

[19] Die Nachteile des puren Ausprogrammierens zeigten sich exemplarisch anläßlich der im Rahmen dieser Arbeit vorgenommenen Modifikationen des Quest-Übersetzers und der *Abstrakten Quest-Maschine*, in deren Quelltexten diese Methode dominiert. Darüberhinaus besteht dort eine generelle Tendenz zum Verzicht auf Parametrisierung. Diese radikale Auffassung des Programmierkriteriums *Einfachheit* wird in der vorliegenden Arbeit nicht geteilt.

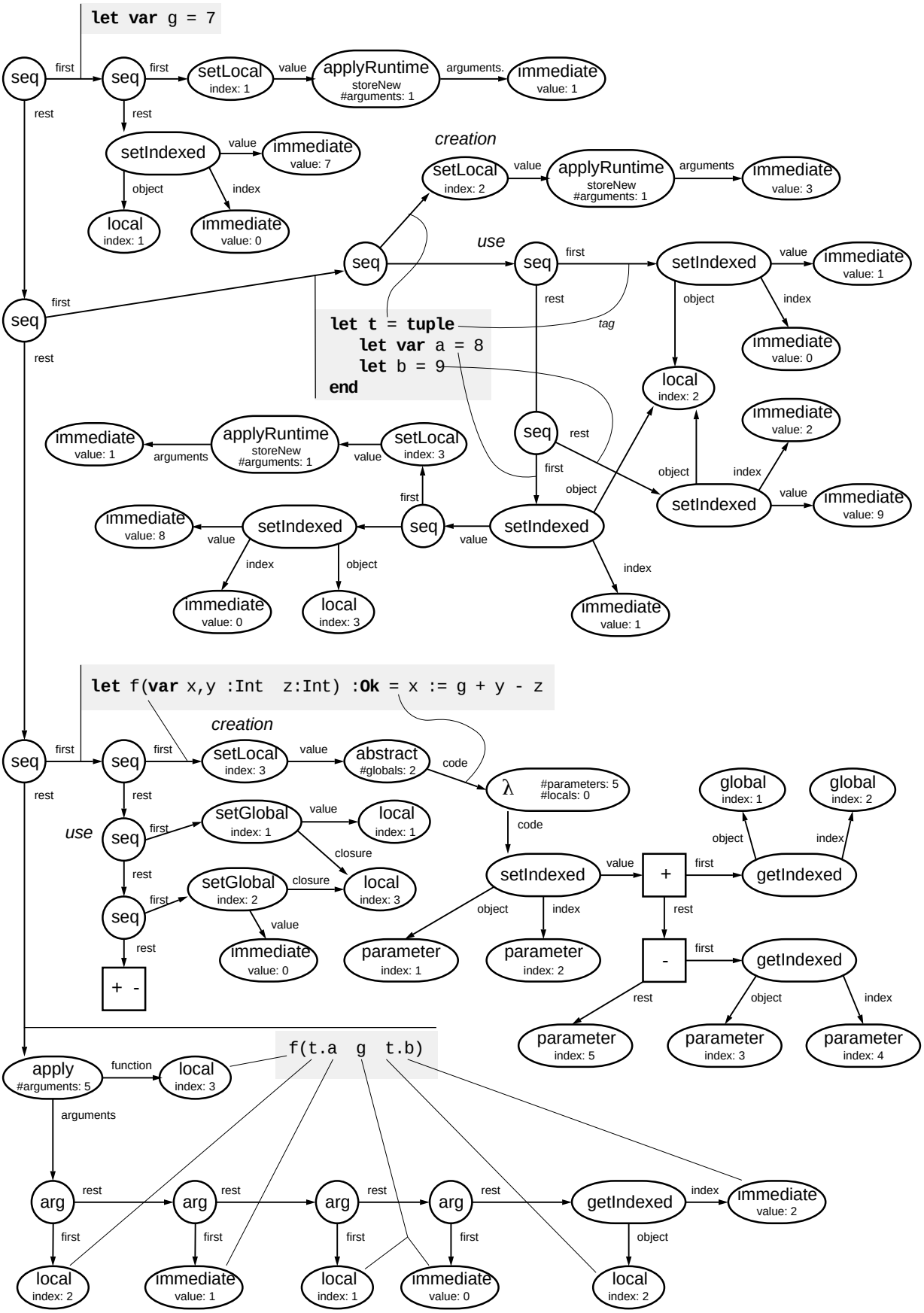

Abbildung 4.11: TML-Kodebeispiel mit L-Werten

einfache Werte: Erzeugung und Initialisierung sind untrennbar.

komplexe Werte: Erzeugung und Initialisierung sind zunächst getrennte Operationen.

Folgender Typ modelliert diesen Sachverhalt:

```
Let Value = Tuple
  creation :TML.T
  case simple with end
  case complex with
    use :TML.T
    local :TML.T
  end
end
```

Das Feld *creation* enthält in jedem Fall den Erzeugungskode, der in der Variante *simple* zugleich den Initialisierungskode darstellt. In der Variante *complex* ist letzterer durch das Feld *use* angegeben. Komplexe Werte werden von Objektspeicherobjekten repräsentiert, deren Objektreferenz durch den im Feld *local* eingetragenen Kode als Wert bereitgestellt wird. Dabei handelt es sich stets um eine **local**-Instruktion einer eigens allozierten lokalen Variablen, da dieser Kode am effizientesten ist. Er wird außerdem nur einmal für das gesamte Objekt erzeugt und bei Bedarf geteilt.

Diese kleine Optimierung ist in den Abbildungen 4.9 und 4.11 ersichtlich, die beide TML-Kodebeispiele für komplexe Werte enthalten. In ersterer Abbildung sind an jeweils einer und in letzterer an jeweils zwei Stellen die kursiv gedruckten Hinweise *creation* und *use* plaziert, um auf die entsprechende Untergliederung des Kodes hinzuweisen. Die lokale Variable, auf die sich das Feld *local* im obigen Typ *Value* bezieht, ist in Abbildung 4.9 eine Hilfsvariable im Sinne von Abschnitt 4.5.2 und entsprechend gekennzeichnet. In Abbildung 4.11 wird dagegen keine zusätzliche Variable benötigt, weil die komplexen Werte ohnehin an eine lokale Variable gebunden werden. Als weitere Optimierung übernimmt letztere zusätzlich die Rolle einer Hilfsvariablen.

Bei der Übersetzung rekursiver Bindungen wird zunächst der Identitätskode benötigt (siehe Abschnitt 4.4.7), während der Initialiserungskode komplexer Werte erst später verarbeitet wird. Der Erzeugungskode kann dem Identitätskode in der Variante *complex* direkt vorangestellt werden, weil der Wert einer TML-Sequenz allein durch ihre letzte Komponente bestimmt ist. Den entsprechend kombinierten Erzeugungs- und Identitätskode eines Wertes liefert folgende Funktion:

```
let valueCreationCode(value :Value) :TML.T =
  case value
  when simple with v then
    v.creation
  when complex with v then
    tml.newSeqCode(v.creation  v.local)
  end
```

Wird der fertige Wert in Form einer Sequenz aller seiner Kodekomponenten benötigt, dann wird diese auf folgende Weise zusammengesetzt:

```
let valueAllCode(value :Value) :TML.T =
  case value
  when simple with v then
    v.creation
  when complex with v then
    tml.newSeqCode(v.creation  tml.newSeqCode(v.use  v.local))
  end
```

Dies geschieht häufig direkt nach der Generierung der einzelnen Kodekomponenten und ist daher in einer kurzen Funktion zusammengefaßt:

```
and translateAllValue(state :State  value :TLValue.T) :TML.T =
  valueAllCode(translateValue(state  valueContext  noLocalIndex  value))
```

4.5.5.2 Die Kodegenerierung für beliebige Wertausdrücke

Die Funktion *translateValue* kapselt die im vorangehenden Teilabschnitt genannten, durch die Verwendung eines Wertes zu treffenden Unterscheidungen von der Übersetzung seines Syntaxbaumknotens ab, indem sie einen Wert des Typs *Value* erzeugt. Sie realisiert damit die postulierte Schnittstelle zur Übersetzung von Werten:[20]

```
and translateValue(state :State  context :Context
                   localIndex :optional.T(Int)
                   value :TLValue.T) :Value =
  begin
    let m = markLocals(state)
    let result =
      case value
      when funCase with v then
        translateFunction(state  localIndex  optional.value(v.allocation)  v.value)
      when arrCase with v then
        translateIndexed(state  localIndex  noIndexedTag  v.bindings)
      when tupleCase with v then
        translateIndexed(state  localIndex  tupleTag  v.bindings)
      when tupleCaseCase with v then
        let tag = optional.new(newIntCode(v.ideRef.index))
        translateIndexed(state  localIndex  tag  v.bindings)
      when seqCase with v then
        tuple case simple of Value with
          translateBindings(state  context  v.bindings)
        end
      else
        tuple case simple of Value with
          translateSimpleValue(state  value)
        end
      end
    releaseLocals(state  m)
    result
  end
```

Die weitaus meisten Varianten des Typs *TLValue.T* (siehe Abschnitt 4.1 und Anhang A.1) betreffen einfache Werte. Deshalb ist diesen eine eigene Funktion (*translateSimpleValue*) zugeordnet, die nur noch die absolut notwendigen Parameter übernimmt. Die Variante *seqCase* ist jedoch ausgegliedert, weil sie die einzige Variante einfacher Werte ist, deren Übersetzung kontextabhängig ist (siehe Abschnitt 4.5.7.1). Die anderen vier in *translateValue* aufgeführten Varianten entsprechen den verschiedenen Ausprägungen komplexer Werte: Funktionen, Arrays, Tupel und variante Tupel. In dieser Aufzählung fehlen die Ausnahmepakete, weil sie niemals direkt, sondern stets als Teil einer *raise*-Anweisung, die den einfachen Werten zugerechnet wird, angegeben werden.

[20] Die Funktionen *markLocals* und *releaseLocals* werden in Abschnitt 4.5.2 beschrieben.

4.5.5.3 Die Kodegenerierung für einfache Werte

Die Funktion *translateSimpleValue* enthält entsprechend der Anzahl der Varianten des Typs *TLValue.T* den Großteil der Fallunterscheidungen in der gesamten Kodegenerierung. Die Übersetzung ist in den meisten Fällen einfach und direkt:

```
and translateSimpleValue(state :State  value :TLValue.T) :TML.T =
  case value
    ...
  when intCase with v then     (* Ganzzahl *)
    tml.newImmediateCode(tml.newIntValue(v.val))
    ...
  when stringCase with v then  (* Zeichenkette *)
    pushLiteral(state  tml.newStringValue(v.val))
    ...
  when indexCase with v then   (* Indizierter Array-Zugriff *)
    let object = translateAllValue(state  v.value)
    let index = translateAllValue(state  v.index)
    tml.newGetIndexedCode(object  index  true)
    ...
  when ideCase with v then     (* Bezeichner => Variable *)
    translateVariableValue(optional.value(v.allocation))
    ...
  when whileCase with v then   (* Schleife mit vorangestellter Abbruchbedingung *)
    let value = translateAllValue(state  v.condition)
    let elseBranch = translateBindings(state  voidContext  v.bindings)
  tml.newLoopCode(tml.newAltCode(value  falseTags  exitBranches  elseBranch  false))
    ...
  end
```

Wie das letzte Beispiel andeutet, sind Kontrollstrukturen syntaktisch betrachtet Werte. Sie werden in Abschnitt 4.5.8 gesondert beschrieben.

Die Variante *ideCase* tritt besonders häufig auf, weil sie Variablen entspricht. Die Vielzahl unterschiedlicher Ausprägungen von Variablen setzt die Funktion *translateVariableValue* in den jeweils zutreffenden Zugriffskode für ihren R-Wert um, indem sie Fallunterscheidungen durchführt, die den in Abschnitt 4.3.3 beschriebenen Repräsentationsformen entsprechen. Sie ist damit das Pendant der Funktion *translateVariableAddress*, die den L-Wert liefert.

4.5.5.4 Die Kodegenerierung für komplexe Werte

Wie bei der Beschreibung des Typs *Value* auf Seite 83 angedeutet und auch in Abschnitt 4.5.2 bereits erwähnt, wird zur Übersetzung eines komplexen Wertes eine lokale Variable alloziert, die zur Laufzeit die Objektreferenz enthält. Ein Beispiel ist in Abbildung 4.9 auf Seite 74 angegeben.

Der Parameter *localIndex* der Funktionen *translateValue*, bzw. *translateFunction* und *translateIndexed* gibt dafür optional[21] einen Index vor (vgl Seite 4.5.5.2). Die zu verwendende Variable muß dann bereits vorher alloziert worden sein. Dies ist z.B. in Bindungen lokaler Variablen stets der Fall.

In Bindungen anderer Variablenarten ist die Allokation einer zusätzlichen lokalen Variablen angebracht. Ihre Zuweisung fällt zur Laufzeit gegenüber der Erzeugung des Speicherobjektes nicht ins Gewicht. Selbst wenn sie nicht weiter verwendet wird, entstehen keine relevanten Verluste. Außer bei leeren Arrays und Funktionen ohne globale Variablen wird die Objektreferenz jedoch (zur Initialisierung) mehrfach benötigt. Die Repräsentation durch eine lokale Variable ist dann optimal, denn sie gestattet den effizientesten Zugriff.

4.5.6 Die Kodegenerierung für Funktionsabschlüsse

Wie bereits in Abschnitt 3.2 beschrieben, werden Funktionsabschlüsse jeweils durch ein Speicherobjekt realisiert. Zur Erzeugung des Kodes, der die Felder eines Funktionsabschlusses initialisiert, wird die Abschnitt 4.3.4 erörterte Repräsentation globaler Variablen herangezogen. Diese erfaßt sowohl R-Werte als auch L-Werte. Für letztere gilt das in Abschnitt 4.5.4 beschriebene Schema, das zwei Werte vorsieht: eine Objektreferenz und einen Index.

In Abbildung 4.11 ist dafür ein Beispiel angegeben. Die Funktion f greift auf den L-Wert der globalen Variablen g zu. Ihr Funktionsabschluß hat daher zwei globale Werte, die jeweils durch eine **setGlobal**-Instruktion initialisiert werden. Dabei wird zur späteren Adressierung der Zelle der Variablen g der Wert der lokalen Variablen mit dem Index 1 und der Speicherobjekt-Offset 0 eingetragen. Außerdem werden die Funktionen + und − in f als globale R-Werte benutzt. Ihre Initialisierung ist durch ein graphisches Nonterminalsymbol angedeutet.

Der Kode für Funktionsabschlüsse wird durch die Funktion *translateFunction* generiert. Da er komplexe Werte im Sinne von Abschnitt 4.4.7 betrifft, wird die Variante *complex* der in Abschnitt 4.5.5.1 beschriebenen Zwischenrepräsentation für R-Werte erzeugt. Folgender Ausschnitt zeigt die wesentlichen Anweisungen:

[21] Optionale Werte sind im Bereich der Datenbankprogrammierung unter der Bezeichnung „Nullwerte“ bekannt. Die Quest-Vorversion der Tycoon-Standardbibliothek *optional:Optional* implementiert sie durch einen Typoperator, der TL-Tupel mit zwei Varianten definiert. Eine der Varianten ist feldlos und die andere hat ein Feld, dessen Typ der Parameter des Typoperators ist.

```
let translateFunction(...) :Value =
  begin
    let translateBody() :TML.T = ...
    (* ... ruft 'translateAllValue' auf, um den Rumpf zu übersetzen. *)
    ...
    (* Eine Hilfsvariable im Sinne von Abschnitt 4.5.2: *)
    let closure = tml.newGetLocalCode(closureLocalIndex  false)

    (* Erzeugung des Initialisierungskodes und *)
    (* die Bestimmung der Anzahl der globalen Werte: *)
    let var nGlobals = 0
    let globals = translateGlobals(closure  assoc.allSnd(function!nested.globals)
                                   (* out: *) nGlobals)

    (* Das Ergebnis sind Kodefragmente eines komplexer Wertes: *)
    tuple case complex of Value with
      let creation = tml.newSetLocalCode(closureLocalIndex
                                         tml.newAbstractCode(nGlobals  translateBody()))
      let use = globals
      let local = closure
    end
  end
```

Die Variable *closureLocalIndex* enthält den Index der Hilfsvariablen. Die Funktion *translateGlobals* stellt für jede gobale Variable anhand der in Abschnitt 4.3.4 beschriebenen Datenstrukturen fest, von welchen anderen Variablen sie abstammt und verknüpft die entsprechenden *setGlobal*-Instruktionen mittels *seq*-Instruktionen zu einer fertigen Initialisierungsequenz.

4.5.7 Die Kodegenerierung für Bindungen

Die Syntax von Bindungen ist im Vergleich zu der von Werten nur wenig reichhaltig, doch ihre Übersetzung ist ungleich schwieriger. Gleichlautende Bindungskonstrukte können in verschiedenen Kontexten ganz unterschiedliche Bedeutungen haben. Weitere Probleme sind die abweichende Anordnung des Kodes für rekursive Bindungen und die von der Veränderlichkeit eines Parameters abhängige Anzahl von Argumenten, die aus einer einzelnen Bindung hervorgeht.

4.5.7.1 Die Kontextabhängigkeiten von Bindungen

Die Kodegenerierung wird durch Kontextabhängigkeiten beeinflußt, die sich im wesentlichen zu den bereits in der Allokationsphase aufgetretenen analog verhalten. Aufgrund des höheren Informationsgehaltes des Syntaxbaumes wird allerdings nur noch eine schwächere Differenzierung benötigt. Anstelle von acht in der Allokationsphase (siehe Abschnitt 4.4.1) genügt es nun, vier Fälle zu unterscheiden, die sich zudem ausschließlich auf Bindungen beziehen. Letzterer Umstand trägt wesentlich zu den in Abschnitt 4.5.5 beschriebenen Abkapselungen der Übersetzung von Wertausdrücken bei.

Der Kontextparameter *context* der in die Übersetzung von Bindungen involvierten Knotenfunktionen hat folgenden Typ:

```
Let Context = Tuple
  construct :Construct
  case indexed with
    objectLocalIndex :Int
    object :TML.T
    var size :Int
  end
  case argument, value, void
end
```

Wie schon in der Allokationsphase wird die Ambivalenz varianter Tupel ausgenutzt: Der Typ *Context* fungiert nicht nur als Aufzählungstyp für Fallunterscheidungen, sondern er definiert auch Aggregate, die kontextbezogene Daten transportieren.

Das Feld *construct* verweist auf die Funktion, die im jeweiligen Kontext zur Bildung von Kodesequenzen zu verwenden ist. Beim Anlegen eines Kontextes der Variante *argument* wird die Funktion *tml.newArgumentCode* eingetragen, in allen anderen Fällen *tml.newSeqCode*. Die Verwendung einer Funktion höherer Ordnung erspart entsprechende Fallunterscheidungen in der Konstruktorfunktion für Kodesequenzen (*constructPieces*, siehe Abschnitt 4.5.7.2).

Die Variante *indexed* tritt bei der Übersetzung der Initialisierungssequenzen von Aggregaten auf (vgl. Abschnitt 4.5.2 und 4.5.5). Das Feld *objectLocalIndex* enthält den Index der lokalen Variablen, die zur Aufnahme der Objektreferenz vorgesehen ist. Um nur eine Kopie ihres meist mehrmals benötigten Zugriffskodes (**local**) anlegen zu müssen, wird dieser im Feld *object* gespeichert. Im Feld *size* wird beim Durchlaufen der Initialisierungssequenz die Gesamtgröße des Speicherobjektes mitgerechnet.

Aus der Kontextvariante kann abgeleitet werden, ob Kode zur Erzeugung eines Endwertes verlangt wird, wenn eine TL-Sequenz als Wertausdruck auftritt (*indexed*, *argument*, *value*) oder ob nur die Seiteneffekte relevant sind (*void*). Beispiele:

```
x := begin let a = 1 and b = 2 end
(* '2' wird als Endwert der Bindungssequenz weiterverarbeitet. *)

for i = 1 to 10 do
  let a = 1 and b = 2 end
  (* Ein Endwert der Bindungssequenz wird nicht benötigt. *)
  ...
end
```

Für die Bindungssequenz gilt im oberen Beispiel der Kontext *value*, im unteren *void*.

4.5.7.2 Die Anordnung des Kodes von Bindungssequenzen

Die Übersetzung von Bindungen ist das Haupteinsatzgebiet der in Abschnitt 4.5.1 postulierten Technik der Ergebniskonstruktion mit Hilfe von Schlangen. Mit ihrer Hilfe werden die

von der Zwischenrepräsentation für Werte (siehe Abschnitt 4.5.5) unterschiedenen Kodefragmente in spezieller Weise angeordnet. Folgender Typ repräsentiert die Klassifizierung, die dazu bezüglich eines Kodefragmentes getroffen wird:

```
Let Piece = Tuple
  code :TML.T
  case creation, use
end
```

Das Feld *code* verweist auf das bewußte Kodefragment. Die Variante *creation* gibt an, daß der Kode vornehmlich der Erzeugung eines Datenobjektes dient; *use* steht für anderweitige Nutzung. Zu letzterer Einstufung zählt insbesondere auch die Initialisierung. Die in dieser Weise verpackten Kodefragmente werden im folgenden „Stücke“ (*pieces*) genannt.

Der Ausgangspunkt von Bindungssequenzen ist im Syntaxbaum stets durch eine gewöhnliche Bindungssequenz vom Typ *TLValue.Bindings* angegeben (siehe Abschnitt 4.1). Für ihre Übersetzung wird jeweils eine eigene Schlange angelegt, die alle Stücke der Sequenz aufnimmt. Anschließend werden die Kodes der in einer *spezifischen* Reihenfolge eingetragenen Stücke durch eine *kanonische* Transformation zu einer fertigen TML-Sequenz angeordnet. Die gesamte Übersetzung von Bindungen ist auf diese Konstruktionsmethode ausgerichtet.

Der (nicht seltene) Spezialfall, der von der Fallunterscheidung des Typs *Piece* profitiert, ist die Übersetzung rekursiver Bindungen. Alle anderen Bindungsarten nutzen im wesentlichen nur die Eigenschaften der Schlange. Die Vereinheitlichung aller auftretenden Bindungsarten erlaubt die Wiederverwendung diverser Funktionen, die Teilaufgaben erledigen und vor allem die problemlose Verflachung ihrer Schachtelung. Dies ist wichtig, wenn die Bindungssequenz eine Argumentliste angibt, weil sonst keine Zuordnung der Einzelbindungen zu den Parametern der aufzurufenden Funktion möglich ist.

Gewöhnliche Einzelbindungen tragen jeweils ihren gesamten Kode in einem *use*-Stück in diese Schlange ein. Eine Fallunterscheidung wird in diesem Fall nicht benötigt, weil die Anordnung des Kodes der lexikalischen Reihenfolge der TL-Anweisungen, aus denen er hervorgeht, genau entspricht.

Dies ist bei rekursiven Bindungen anders. Deren Erzeugungskodes werden von den Initialisierungskodes getrennt verarbeitet, wobei erstere in *create*-Stücken und letztere in *use*-Stücken erfaßt werden. Zuerst müssen alle *create*-Stücke der gesamten Sequenz in die Schlange eingefügt werden, dann erst folgen die *use*-Stücke in gewohnter Manier. Dazu wird temporär für jede Sequenz rekursiver Bindungen eine weitere Schlange angelegt. Die rekursiven Einzelbindungen fügen dann ihre *create*-Stücke in die Hauptschlange ein, die *use*-Stücke jedoch in die „Extra-Schlange“. Am Schluß wird letztere einfach an die Hauptschlange angehängt und die gewünschte Anordnung ist erreicht.

Abbildung 4.12 stellt die Stücke des folgenden TL-Beispiels und ihre Anordnung in den beiden Schlangen, die bei der Übersetzung entstehen, dar.

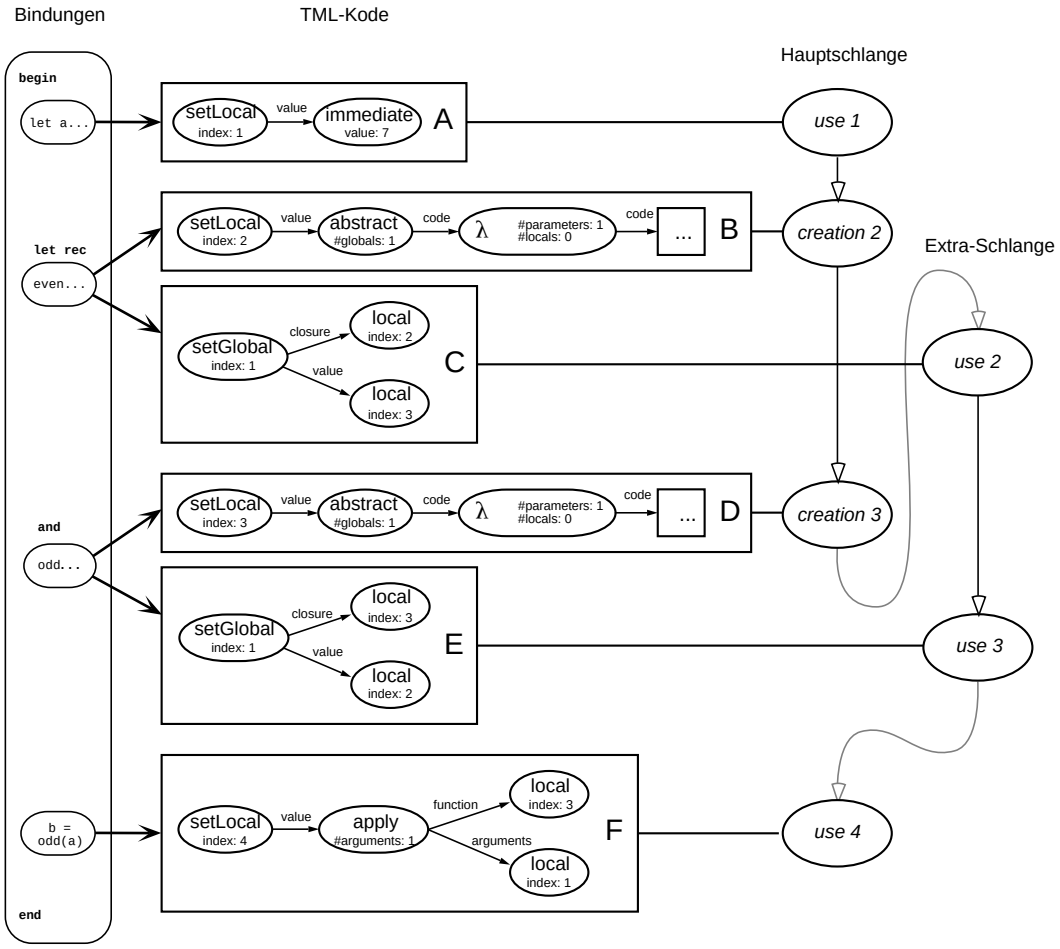

Abbildung 4.12: Die Gliederung des TML-Kodes für TL-Bindungen und seine Akkumulation

```
begin
  let a = 7

  let rec even(x :Int) =  x == 0 orif odd(x − 1)
  and odd(x :Int) =  x ==  1 orif even(x − 1)

  let b = odd(a)
end
```

Beide Funktionen sind in der jeweils anderen als globale Variable sichtbar. Auf diesen Umstand sind die **setGlobal**-Instruktionen zurückzuführen.

Die Funktion *constructPieces*[22] leistet die kanonische Verarbeitung der Stücke einer Bindungssequenz und liefert den fertigen Sequenzkode. Dazu wird ihr die Konstruktorfunktion

[22] Diese Funktion ist ziemlich kurz (25 Zeilen), aber dennoch ein wenig anspruchsvoll. Zum Vergleich mit der Beschreibung ihres Effektes ist sie in Anhang B.2 aufgeführt.

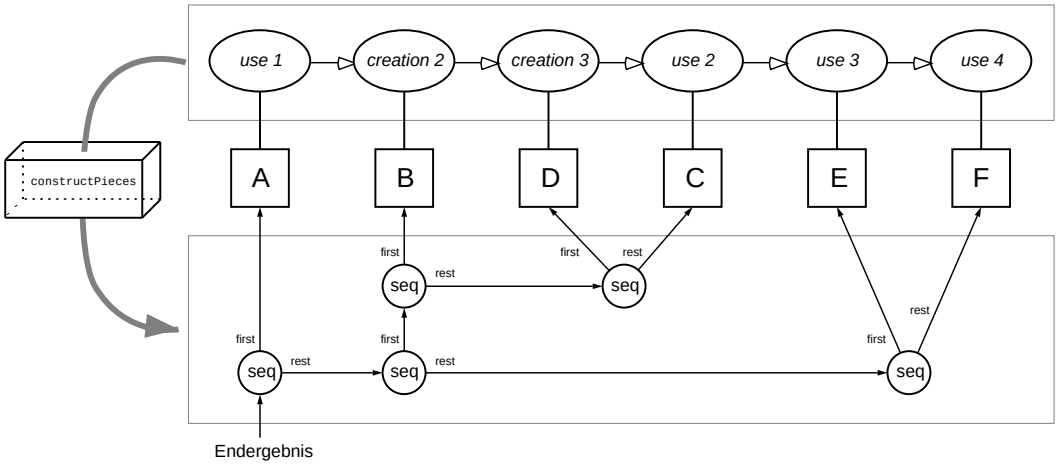

Abbildung 4.13: Die Anordnung der TML-Kodefragmente beim Abschluß der Übersetzung einer Bindungssequenz

des aktuellen Kontextes als Argument übergeben, so daß wahlweise Anweisungssequenzen (**seq**) oder Argumentlisten (**arg**) zusammengesetzt werden. Ihr Effekt ist folgender:

- Alle Kodefragmente behalten untereinander ihre Reihenfolge bei. Lediglich ihre Gruppierung durch **seq**-, bzw. **arg**-Instruktionen wird beeinflußt.
- Die Haupteinträge in die Ergebnissequenz liefern die *use*-Stücke. Wenn ihnen kein *create*-Stück vorangeht, wird ihr Kode direkt als Element der Ergebnissequenz erfaßt.
- Treten *create*-Stücke vor einem *use*-Stück auf, dann werden sie mit diesem durch **seq**-Instruktionen zu einer Untersequenz zusammengefaßt, die dann ein Element der Hauptsequenz bildet.

Abbildung 4.13 zeigt die Verarbeitung der in Abbildung 4.12 dargestellten Stücke. Die Zuordnung der Kodefragmente basiert auf der Kennzeichnung der entsprechenden TML-Nonterminale mit den Buchstaben A bis F.

Argumentlisten haben nicht nur die gleiche syntaktische Form wie Bindungssequenzen, sie lassen sich auch bis auf zwei Unterschiede vollkommen analog übersetzen. Das Problem, daß für Argumente eine spezielle Sequenzinstruktion (**arg**) vorgesehen ist, wird durch die oben erwähnte Parametrisierung von *constructPieces* gelöst. Wesentlich schwieriger erscheint auf den ersten Blick die in Abschnitt 4.5.4 beschriebene Regelung, daß für L-Wert-Bindungen *zwei* Einträge in der TML-Ergebnissequenz erforderlich sind. R-Wert-Bindungen verhalten sich dagegen wie gewöhnlich. Im unteren Bereich von Abbildung 4.11 befinden sich Beispiele für beide Fälle. Durch die Verwendung von Schlangen ist der Aufwand zur Bewältigung des Problems verschiedener Anzahlen von Kodestücken pro Einzelbindung verschwindend gering. Es genügt, einfach die erforderliche Anzahl von Stücken in die aktuelle Schlange einzufügen.

Die Zeitkomplexität der Kodegenerierung für Bindungssequenzen läßt sich analog zur Argumentation in Abschnitt 4.5.3 abschätzen. Die Funktion *constructPieces* verhält sich ebenso

if	**alt**
andif	**alt, immediate**
orif	**alt, immediate**
case	**alt, runtime, raise, ...**
exit	**exit, immediate**
loop	**loop**
while	**loop, alt**
for	**loop, alt, runtime, ...**
try	**trap, runtime, ...**
raise	**raise**
reraise	**raise, runtime**
assert	**alt, raise, runtime, literal, nop**

Abbildung 4.14: Zur Übersetzung von TL-Kontrollstrukturen erforderliche TML-Instruktionen

wie *queueLiterals* linear. Das Verbinden von Schlangen fällt nicht zusätzlich ins Gewicht, denn es ist eine extrem effiziente Operation (hauptsächlich besteht sie aus einer Zuweisung). Als obere Grenze kann daher insgesamt wiederum ein konstanter Faktor (ca. 2 bis 3) des theoretischen Minimalaufwandes angenommen werden, denn die Knotenfunktionen haben ebenfalls eine lineare Komplexität.

4.5.8 Der TML-Kode von Kontrollstrukturen

Tabelle 4.14 zeigt in einer Übersicht alle TL-Kontrollstrukturen und ordnet ihnen die TML-Instruktionen ihrer Implementation zu.

Syntaktisch betrachtet sind alle Kontrollstrukturen Ausdrücke, die einfache Werte beschreiben. Deshalb wird ihre Übersetzung von der in Abschnitt 4.5.5.2 auf Seite 86 dargestellten Funktion *tanslateSimpleValue* geleistet. Besonders einfach ist die Kodegenerierung für **loop**- und **exit**-Anweisungen, wie aus folgendem Auszug hervorgeht:

```
...
when loopCase with v then
  tml.newLoopCode(translateBindings(state  voidContext  v.bindings))
when exitCase then
  tml.newExitCode(okCode)  (* 'okCode': Kode für den ausgezeichneten Wert 'ok' *)
...
```

Auch für alle anderen Kontrollstrukturen werden keine Techniken verwendet, die über das bereits beschriebene Repertoire hinausgehenden. Am kompliziertesten ist die **case**-Anweisung, deren Sprungmarken und Verzweigungen in Vektoren anzuordnen sind. Für diese Aufgabe wird das Prinzip der Methode zur Erfassung der Literalvektoren (siehe Abschnitt 4.5.3) wiederverwendet. Die Sprungmarken und Verzweigungen werden beim Durchlaufen des Syntaxbaumes in einer Schlange akkumuliert. Am Schluß wird diese durch den Aufruf einer Funktion aus einer Tycoon-StandardBiblithek (*arrayOp.create*) in einen Vektor transformiert.

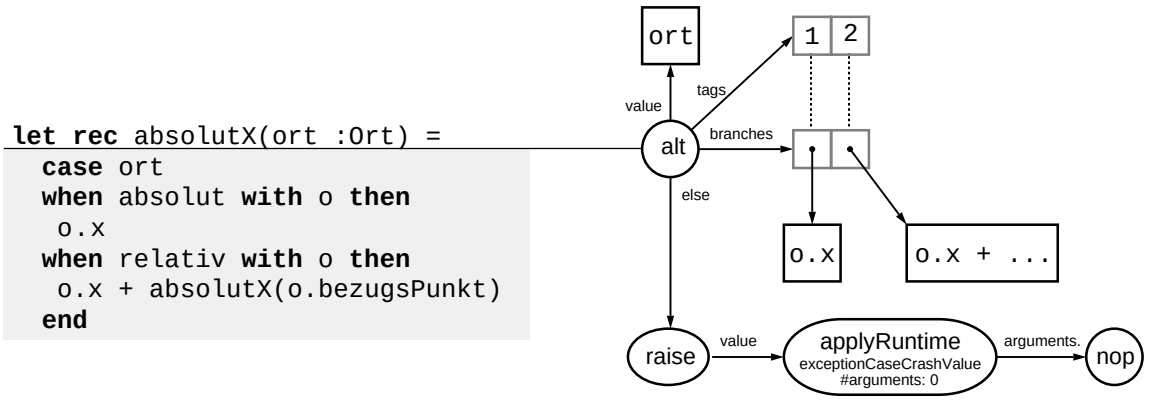

Abbildung 4.15: Der TML-Kode einer Fallunterscheidung

Abbildung 4.15 zeigt eine **case**-Anweisung und ihren TML-Kode. Sie enthält als zusätzliches graphisches Element Vektoren von TML-Werten, die Fallmarken und Verzweigungsreferenzen repräsentieren. Die Zuordnung zwischen diesen wird in der Implementation durch Indizes bestimmt, deren explizite Angabe in der Graphik allerdings wenig anschaulich wäre. Stattdessen geben feine gestrichelte Linien die Zuordnung an. Die TL-Anweisung bezieht sich auf den in Abbildung 4.6 auf Seite 59 gezeigten Tupeltyp *Ort*. Da die Definition von Subtypen mit zusätzlichen Varianten möglich ist, kann die Fallunterscheidung scheitern. In diesem Fall wird eine vordefinierte Ausnahme ausgelöst, deren Wert eine Laufzeitsystemfunktion liefert. Alternativ können für diese Verzweigung mit Hilfe des TL-Schlüsselwortes **else** benutzerdefinierte Anweisungen angegeben sein.

Ein Beispiel für eine Ausnahmebehandlung ist in Abschnitt 5.4.2 auf Seite 106 in Kombination mit dem dazugehörenden C-Kode angegeben.

4.6 Die Unterstützung der Laufzeitanalyse

Eine Laufzeitanalyse benötigt eine Symboltabelle, Allokationsdaten und Quelltextverweise. In konventionellen Systemen müssen die Symboltabelle und die Allokationsdaten in einer zusätzlichen Datei gespeichert werden, für die eine eigene Repräsentation zu erstellen ist. In persistenten Systemen besteht jedoch die Möglichkeit, den Syntaxbaum direkt als Symboltabelle zu verwenden, da seine Langlebigkeit auch ohne externe Dateien erreicht wird. Durch diese Maßnahme werden dann gleich zwei Transformationen überflüssig (externe Ein- und Ausgabe). Die Allokationsdaten sind im Tycoon-Systems nach erfolgter Übersetzung bereits im Syntaxbaum enthalten, so daß für ihre Speicherung ebenfalls kein zusätzlicher Aufwand erforderlich ist.

Der Syntaxbaum enthält zu jedem operational relevanten syntaktischen Konstrukt einen Quelltextverweis. Bei einer schrittweisen Ausführung des TML-Kodes kann deshalb durch Vergleich mit dem Syntaxbaum jederzeit der Bezug zum Quelltext zurückverfolgt werden. Sobald der Kode jedoch einmal ohne Kontrolle der Laufzeitanalyse ausgeführt wird, geht die

Zuordnung verloren. Es ist deshalb erforderlich, an jeder Instruktion, die potentiell einen Eintritt in die Laufzeitanalyse auslöst, Quelltextverweise anzubringen. Diese lassen sich aufgrund ihrer Eindeutigkeit leicht im Syntaxbaum auffinden, sodaß der vollständige Analysekontext wiederhergestellt werden kann.

Die TML-Instruktionen, welche die Laufzeitanalyse aufrufen können, sind:

raise: das Auslösen einer Ausnahme.

reraise: das wiederholte Auslösen einer Ausnahme.

runtime: die Ausführung einer Laufzeitsystemfunktion (vgl. Abschnitt 3.3).
Der Laufzeitsystemkode **tupleProject** löst selbständig eine Ausnahme aus, wenn ein Zugriff auf ein Tupelfeld scheitert.

Im weiteren Ausbau des Tycoon-Systems sind außerdem folgende Instruktionen vorgesehen:

debug: ein expliziter Aufruf der Laufzeitanalyse, der direkt aus einer entsprechenden Anweisung im TL-Quelltext hervorgeht.

suspend: eine spezielle Instruktion, die den Zweck hat, einen laufenden Evaluationskontext einzufrieren. Die Kontrolle über diesen Evaluationskontext kann einem anderen Evaluationskontext oder aber der Benutzerumgebung obliegen, wobei letztere die Laufzeitanalyse aufrufen könnte. Außerdem sind in diesem Zusammenhang diverse Fehlersituationen möglich, die ebenfalls die Laufzeitanalyse auslösen sollten.

In einem speziellen Modus kann der Interpreter jede Auslösung einer Ausnahme abfangen und sofort die Laufzeitanalyse einschalten. Zu deren Orientierung ist allen obigen Instruktionen jeweils ein Literal als zusätzliches Argument beigefügt, das die Quelltextposition in Form von Dateiname, Zeilennummer und Buchstabenspalte angibt. Diese Information kann während der Generierung einfach aus dem Syntaxbaum übertragen werden. Als Beispiel folgt der Auszug der Funktion *translateSimpleValue* (vgl. Abschnitt 4.5.5.2), der die Übersetzung von Tupelfeldzugriffen vornimmt:

```
...
when projectCase with v then
  let object = translateAllValue(state  v.value)
  let index = newIntCode(v.ideRef.index)
  let arguments = tml.newArgumentCode(object
                  tml.newArgumentCode(index
                    pushLiteral(state  tml.newSourcePositionValue(v.pos))))
...
```

Die Funktion *tml.newSourcePositionValue* erzeugt zur Übersetzungszeit den Literalwert der Quelltextposition. Durch *pushLiteral* wird ein entsprechender Literalkode erzeugt (siehe Abschnitt 4.5.3), der dann zur Laufzeit die Adressierung des Literals leistet.

Kapitel 5

Die Maschinenkodegenerierung

Zum Zwecke der Maschinenkodegenerierung wird C als portable Zielsprache verwendet. Die Vorteile dieser Entwurfsentscheidung werden im nachfolgenden Abschnitt erörtert. Daraufhin wird die Strategie zur Übersetzung von TML nach C beschrieben. Eine wichtige Anforderung an den C-Kode ist die Einhaltung der in TML festgelegten Auswertungsreihenfolge. Es wird eine portabel verwendbare Methode angegeben, die dies sicherstellt. Besonders aufwendige Maßnahmen erfordert die Unterstützung von Funktionsaufrufen und Ausnahmebehandlungen. Im Zusammenhang dieser beiden Konstrukte werden die elementaren Bestandteile des C-Kodes beschrieben. Die anschließend beschriebene Methode der Maschinenkodegenerierung zeichnet sich insbesondere durch das Delegieren von Optimierungen an externe Dienste und durch die Auswechselbarkeit der Zielsprache aus. Am Ende des Kapitels werden die notwendigen Maßnahmen zur Verwendung von Modula-2 und Pascal als exemplarische zusätzliche Zielsprachen erläutert.

5.1 C als portable Zielsprache

Die in [Matthes 92] geforderte Portabilität von Anwendungen bedingt, isoliert betrachtet, nicht zwingend die Portabilität des Tycoon-Systems selbst: TL und TML-Kode sind portabel und setzten lediglich *irgendeine* Tycoon-Implementation auf der jeweiligen Zielmaschine voraus. Es ist daher prinzipiell möglich, die Maschinenkodegenerierung auf völlig verschiedene Weisen zu gestalten.

Wenn jedoch weitere Anforderungen der Anwendungen mit in Betracht gezogen werden, dann folgt auch die Forderung der *eigenen* Portabilität des Tycoon-Systems. Es ist insbesondere zur Erstellung großer Informationssysteme konzipiert. Folgende Anforderungen derselben sind im gegebenen Zusammenhang entscheidend: Langlebigkeit und Heterogenität (der Einsatz in unterschiedlichen Hardware- und Software-Umgebungen).

Naturgemäß ist Portabilität förderlich für die Gewährleistung von Heterogenität. Durch das zunehmende Aufkommen heterogener Netzwerke nimmt die Bedeutung der Heterogenität ständig zu. Die Zahl der jeweils potentiell von Konfigurationsentscheidungen betroffenen Systeme wächst im Zuge der Ausdehnung der Netze.

Auch zur Sicherstellung der *Langlebigkeit* der Anwendungen ist die Protabilität des Implementationssystems wichtig. Selbst wenn erstere physisch isoliert als eigenständige Programme

vorliegen, ist eine vollkommene Unabhängigkeit illusorisch, da die Entwicklung großer Systeme solange diese genutzt werden in praktisch keinem Fall jemals beendet ist. Die Langlebigkeit der Anwendungen hängt daher stets auch von der Langlebigkeit des Implementationssystems ab. Weil ein Überdauern der Verfügbarkeit bestimmter Umgebungen jedoch nur durch Umgebungswechsel möglich ist, impliziert die Forderung nach Langlebigkeit die nach Heterogenität. Letztere wiederum führt zur Forderung nach Portabilität und damit ist die Langlebigkeit der Anwendungen sogar ein *hinreichendes* Kriterium der Portabilität des Implementationssystems.

Zusammenfassend wird festgestellt:

> Portabilität ist eine vorrangige Anforderung des Tycoon-Systems, die es auch bei der Maschinenkodegenerierung zu beachten gilt.

Daher scheidet die direkte Erzeugung von Maschinenkode als Möglichkeit aus. Auch die allseits vorhandenen Assembler sind jeweils zu umgebungsspezifisch.

Geeigneter erscheinen dagegen architekturunabhängige Zwischensprachen, denn sie sind konzeptionell portabel. Doch diese Eigenschaft allein genügt nicht, denn sie trifft auch auf TML zu. Weitere Zwischensprachen kommen nur in Betracht, wenn sie sich durch besonders effizienten Kode auszeichnen.

Für funktionale Sprachen ist der in [Peyton Jones 87] vorgestellte G-Kode sehr geeignet. Unter anderem wird die mit TL semantisch verwandte Sprache ML in G-Kode übersetzt. Der wichtigste Vorteil des G-Kodes gegenüber den meisten anderen Zwischensprachen, die effiziente Unterstützung verzögerter Auswertungen (*lazy evaluation*), kann jedoch nicht ausgenutzt werden, da sowohl TL als auch TML *strikt* evaluativ ist. Außerdem ist die Unterstützung von Persistenz im G-Kode nicht vorgesehen.

Eine besonders für Optimierungszwecke geeignete Zwischenrepräsentation ist CPS (*continuation passing style*), eine aus der denotationellen Semantik stammende Variante des Lambda-Kalküls. CPS wird unter anderem ebenfalls zur Übersetzung von ML eingesetzt [Tarditi et al. 91; Appel 92]. Viele in anderen Repräsentationen nur mit viel Aufwand realisierbare Optimierungen können mit Hilfe von CPS durch einfache Substitutionsregeln geleistet werden [Gawecki 91; Appel 92]. Außerdem kann der jeweils vorhandene Satz elementarer Funktionen problemlos erweitert werden [Gawecki 91]. Auf diese Weise ließen sich die Tycoon-Laufzeitsystemfunktionen, insbesondere die des TSP in CPS-Kode einbetten. Die elementaren Funktionen in CPS sind jedoch nicht nur flexibel definierbar, sondern auch strukturell für die Übersetzung in Maschinensprache bestimmend. Die heutigen CPS-Implementationen unterscheiden sich daher in der Regel sehr stark voneinander.

Wenn eine Zwischensprache nicht hinreichend standardisiert oder wenig verbreitet ist, dann wird durch ihren Einsatz das grundsätzliche Problem der *portablen* Maschinenkodegenerierung lediglich um einen Schritt verlagert. Auf diesem Gebiet werden jedoch große Anstrengungen unternommen [RSRE 91; TDF 91], so daß mit einer eventuellen zukünftigen Änderung dieser Situation zu rechnen ist.

Als weitere Alternative bietet sich die Wahl einer Hochsprache als portable Zwischenrepräsentation an. Grundsätzlich sind wie bereits erwähnt jedoch nur weit verbreitete, etablierte Programmiersprachen geeignet, deren fortwährende Unterstützung ihrer Implementation in allen

relevanten Umgebungen sehr wahrscheinlich ist. Außerdem wird die faktische Durchsetzung einer hinreichenden Standardisierung gefordert.

Sprachen wie Common-Lisp [Steel 84] oder Ada [Ichbiah, others 83] erfüllen zwar beide Bedingungen, scheiden aber aufgrund allzu aufwendiger Laufzeitsysteme aus.

Modula-2 ist weit verbreitet und rein formell inzwischen auch standardisiert [King 89; ModISO 91; Schröder 91]. Die Unterschiede der existierenden Implementationen sind jedoch gravierend. Eine hinreichende Verbreitung der Übersetzer, die den neuesten Standard unterstützen, wird eventuell erst zu einem Zeitpunkt erreicht werden, an dem die Sprache Modula-2 inzwischen wieder verdrängt wird.

Die Sprachen, deren Standards sich am besten durchgesetzt haben, sind FORTRAN und COBOL. Sie sind sehr leistungsfähig in ihren begrenzten Anwendungsgebieten, im allgemeinen jedoch wenig ausdrucksstark und äußerst unpraktisch. Die neueste FORTRAN-Version, Fortran-90, in der einigen Mängeln durch Erweiterungen begegnet wurde, ist noch nicht sehr verbreitet.

C [Kernighan, Ritchie 77; Kernighan, Ritchie 88] hat sich bereits in zahlreichen Systemen als Zwischensprache für die Kodegenerierung bewährt (z.B. Modula-2 [Schäfer, Bomarius 87], Modula-3 [Nelson 91], Standard-ML [Tarditi et al. 91]). Der wichtigste Vorteil von C ist die hervorragende Unterstützung der Heterogenität:

- Durch seine Standardisierung erreicht C einen ausgezeichneten Portabilitätsgrad.
- C ist nahezu einzigartig in seiner Verfügbarkeit. Es existieren für alle bekannteren Hardware-Architekturen und Betriebssysteme C-Übersetzer. Außerdem gehören sie gewöhnlich zu den ersten Produkten, die in einer neuen Umgebung entstehen.
- Sollte einmal kein C-Übersetzer in einer bestimmten Umgebung vorhanden sein, dann kann das GNU-System dazu verwendet werden, mit relativ geringem Aufwand einen C-Übersetzer (*gcc*) zu erzeugen. Das gesamte GNU-System ist als sogenanntes *public domain*-Produkt im Quelltext verfügbar.

Hinzu kommt die ausgeprägte Kompatibilität mit anderen Systemen:

- Alle Betriebssysteme unter denen C-Übersetzer verfügbar sind, haben auch C-Programmierschnittstellen, nicht zuletzt weil viele von ihnen weitgehend in C (z.B. UNIX) oder in einer verwandten Sprache (z.B. Apple-Finder in Pascal) implementiert worden sind.
- Andere Programmiersprachen haben sehr häufig C-Programmierschnittstellen und werden dadurch ebenfalls zugänglich. Die Implementation noch nicht vorhandener Schnittstellen ist wenig aufwendig, da durch maschinennahe Anweisungen in C explizit Speicherbereiche manipuliert werden können.
- Anwendungen, die Programmierschnittstellen haben, bieten diese meist in C oder in mindestens einer hinreichend verwandten Sprache wie Pascal, Modula-2 oder FORTRAN an.
- Auch in der umgekehrten Schnittstellenbeziehung gehört C de facto zum Standard: Systeme, die sogenannte externe Routinen aufrufen können, weisen meistens eine C-Schnittstelle auf.

Folgende Eigenschaften zeichnen C als besonders geeignet für die Kodegenerierung aus:

- C Konstrukte enthält, die eine sehr direkte Implementation der meisten TML-Instruktionen erlauben (vgl. z.B. Abschnitt 5.5.1).
- C ist sehr „maschinennah", d.h. C-Anweisungen abstrahieren nur wenig von Konstrukten gebräuchlicher Hardware-Architekturen, die nach dem von-Neumann-Prinzip arbeiten: Register, Stapelspeicher, indirekte Adressierung, bedingte und unbedingte Sprünge. Dadurch bietet C im Verhältnis zu den meisten anderen Programmiersprachen eine hohe Performanz.
- Da C für die Systemprogrammierung konzipiert ist, verfügen die meisten C-Übersetzer über sehr gute Optimierer.

Besonders im Falle der *Sun Sparc*-Architektur, dem Zielsystem der Erstimplementation, ist die Laufzeiteffizienz hoch. Der *Sun* C-Übersetzer verfügt über einen leistungsfähigen Optimierer (*iropt*, (vgl. Abschnitt 5.6).

Es ist zwar nicht notwendig, aber auch nicht auszuschließen, daß später einzelne Tycoon-Implementationen in speziellen Systemen entstehen werden, die eine andere Zielsprache als C verwenden. Auch in einem solchen Fall ist eine leicht zu erstellende Version auf C-Basis äußerst nützlich, da sie die Durchführung des *bootstrap* erheblich vereinfacht.

Im Anschluß an die Aufzählung der Vorteile von C soll auch auf einige Kritikpunkte an dieser Programmiersprache eingegangen werden:

1. Nicht der gesamte Sprachumfang von C ist wirklich portabel. Am gefährlichsten sind die subtilen Lücken in der Festlegung von Auswertungsreihenfolgen.
2. Die statische Überprüfung der Korrektheit von C-Anweisungen ist im Vergleich zu Programmiersprachen wie Modula-2 oder Ada wenig umfangreich. Deshalb häufen sich bei mangelnder Programmierdisziplin Laufzeitfehler, die in restriktiveren Sprachen vermieden werden. Vor allem ist C relativ „schwach typisiert", d.h. an vielen Stellen erfolgen implizite Typanpassungen.

Wie dem erstgenannten Problem wirksam begegnet werden kann, wird in Abschnitt 5.3 beschrieben. Das zweite Problem ist im vorliegenden Zusammenhang irrelevant, denn zur sinnvollen Nutzung der C-Generierung muß ihre Korrektheit ohnehin vorausgesetzt werden.

5.2 Die Übersetzungsstrategie

Zur Bestimmung einer Übersetzungsstrategie kann aus Gründen der Portabilität kein spezieller C-Übersetzer betrachtet werden. Es wird jedoch davon ausgegangen, daß *jeder* C-Übersetzer zumindest die Aufgabe, für die er konzipiert ist, effektiv löst: die Übersetzung typischer „handgeschriebener", d.h. durch menschliche Programmierer erstellter, C-Anweisungen. Wenn also generierte C-Anweisungen große strukturelle Ähnlichkeit mit typischen handgeschriebenen Anweisungen für den gleichen Zweck haben, dann kann erwartet werden, daß ihre Laufzeiteffizienz zumindest nicht ungewöhnlich schlecht und wahrscheinlich sogar relativ

hoch ist. Außerdem sind nicht nur alle C-Übersetzer auf die Verarbeitung handgeschriebener Anweisungen vorbereitet, sondern in der Regel auch die von ihnen eingesetzten Optimierer.

Jedem TML-Funktionsrumpf entspricht eine C-Funktion, wobei TML-Parameter direkt auf C-Parameter abgebildet werden. Zusätzlih erhalten alle C-Funktionen jeweils einen weiteren Parameter, der aktivierten Funktionsinstanzen den indizierten Zugriff auf ihren eigenen Funktionsabschluß erlaubt. Sofern Abweichungen von der Orientierung an handgeschriebenem C auftreten, mindern sie die Performanz nachweislich nicht wesentlich oder sie beruhen auf Bedingungen, die auch jeden anderen Ansatz in ähnlicher Weise betreffen würden. Zum Beispiel existiert in C kein Sprachkonstrukt, das den dynamischen Ausnahmebehandlungen in TML direkt entspricht. Sie werden mit Hilfe der C-Standardfunktionen *setjmp* und *longjmp* implementiert. Diese Technik hat sich bereits im SRC Modula-3-System [Nelson 91] bewährt, welches ebenfalls C zur Maschinenkodegenerierung benutzt. Für die übrigen TML-Kontrollstrukturen sind in C korrespondierende Konstrukte vorhanden.

Da alle komplexen TML-Werte durch Objektspeicherstrukturen implementiert werden müssen, werden die Möglichkeiten der Datenstrukturierung in C weitgehend außer acht gelassen. Zur Manipulation der Datenobjekte des Objektspeichers ist die Verwendung von Laufzeitsystemfunktionen des TSP obligatorisch. Die übrigen Laufzeitsystemfunktionen führen zum Teil sehr primitive Operationen, wie z.B. die Inkrementierung einer Ganzzahl, durch. Diese Funktionsaufrufe könnten ohne weiteres durch direkte Anweisungen ersetzt werden. Es bietet sich jedoch an, diese Aufgabe im Rahmen der Implementation einer allgemeineren *inlining*-Technik zu erledigen (siehe 6.3).

Eine sehr augenscheinliche, die Laufzeiteffizienz jedoch nur geringfügig beeinträchtigende, Abweichung von handgeschriebenem C ergibt sich aus den unterschiedlichen Grundstrukturen in TML und C. TML ist entsprechend seiner Abstammung vom Lambda-Kalkül vollständig wertorientiert, d.h. jede TML-Anweisung ist auch ein Wertausdruck. In der C-Syntax sind Wertausdrücke dagegen stets Teil einer C-Anweisung, die ihrerseits keinen Wert liefert. Um dennoch eine kontinuierliche Weitergabe von Werten zu erreichen, ist daher eine Zwischenspeicherung von Werten durch Seiteneffekte unumgänglich. Dazu werden die effizientesten in C verfügbaren Zwischenspeicher verwendet: lokale Variablen. Mit Hilfe lokaler Variablen wird auch die Einhaltung der in TML festgelegten Auswertungsreihenfolge auf portable Weise sichergestellt (siehe Abschnitt 5.3).

Während der C-Generierung eingeführte lokale Variablen werden im folgenden „Temporärvariablen“ genannt. Diese Bezeichnung betont den Aspekt ihres jeweils eng begrenzten Einsatzes zur Laufzeit. Sie dient vor allem der Unterscheidung von den durch die TML-Kodegenerierung allozierten lokalen Variablen.

Es wird grundsätzlich jeder TML-Teilausdruck in einer Temporärvariablen zwischengespeichert. Da TML streng wertorientiert ist, führt diese Maßnahme zu drastischen Vereinfachungen der Übersetzung (vgl. Abschnitt 5.5.3). Auf dieser Grundlage wird das Übersetzungsschema schließlich so stark vereinheitlicht, daß die Zielsprache mit geringem Aufwand ausgetauscht werden kann, ohne den Kernalgorithmus zu verändern. Das zugrundeliegende Prinzip und der Gebrauch dieser Anpassungsfähigkeit werden in Abschnitt 5.7 beschrieben.

Im folgenden werden die Auswirkungen der einheitlichen Verwendung von Temporärvariablen auf das Laufzeitverhalten des generierten Kodes erörtert.
Sowohl die Zuweisung an eine Temporärvariable als auch der anschließende Zugriff auf ihren Wert zählen in jeder konventionellen Hardware-Umgebung (sowohl CISC als auch RISC)

zu den schnellsten C-Operationen. Sie erfolgen stets anläßlich einer weiteren, typischerweise mindestens ebenso teuren Operation, nämlich der Erzeugung des zwischenzuspeichernden Wertes. Daher existitiert stets eine obere Schranke des Laufzeitzuwachses in der Nähe des Faktors 3. Dies ist jedoch eine sehr großzügige, rein theoretische Abgrenzung. Sie beweist, daß ausufernde Laufzeiten prinzipiell ausgeschlossen sind. In der Praxis ist der tatsächliche allgemeine Effizienzverlust vernachlässigbar gering, was auf folgende Ursachen zurückzuführen ist:

- Zur Verlangsamung tragen ausschließlich die vermeidbaren Temporärvariablen bei.
- Eine Registerallokation von Temporärvariablen ist in nicht allzu komplizierten Ausdrücken sehr wahrscheinlich. Durch die in Abschnitt 5.5.3 angegebene Wiederverwendungsstrategie (vgl. auch Abbildung 4.10 auf Seite 76), wird die benötigte Anzahl von Temporärvariablen optimiert.
- Einige der häufigsten Grundoperationen sind sehr laufzeitintensiv. Dabei handelt es sich vor allem um die Aufrufe der Laufzeitsystemfunktionen des TSP. Aber auch alle anderen Funktionsaufrufe sind relativ „teuer".
- Die Elimination überflüssiger Temporärvariablen gehört im allgemeinen zu den am erfolgreichsten gelösten Aufgaben nachgeschalteter Optimierer. Deren Wirksamkeit wird außerdem dadurch begünstigt, daß Temporärvariablen überwiegend in eng begrenztem Kontext verwendet werden.

5.3 Eine portabel verwendbare Methode zur Festlegung der Auswertungsreihenfolge

Die Auswertungsreihenfolge ist in TML streng deterministisch festgelegt. Zwar erzeugt jeder einzelne C-Übersetzer ebenfalls deterministischen Kode, die Auswertungsreihenfolge kann jedoch von Übersetzer zu Übersetzer variieren. Deshalb ist eine naive Übersetzung von TML nach C nicht portabel. Die wichtigsten zu beachtenden Lücken in der Festlegung von Auswertungsreihenfolgen in C werden im folgenden erörtert.

In C existieren zwei Arten von Sequenzen:

Ausdruckssequenzen: Einzelne Ausdrücke werden durch Kommata *getrennt* und durch runde Klammern zusammengefaßt.
Beispiel: *(i = 2, f(&i, j * 3), i + 1)*
Der Wert einer Ausdruckssequenz ist durch den letzten Einzelausdruck bestimmt.

Anweisungssequenzen: Einzelanweisungen, die jeweils durch ein Semikolon *beendet* werden und „Blöcke". Ein Block ist eine durch geschweifte Klammern begrenzte Anweisungssequenz.
Beispiel: *i = 2; while (i < 4) {f(&i, j * 2); g(i + j);} x = h(j + 3);}*
Weder Einzelanweisungen noch Blöcke liefern einen Wert.

In beiden Fällen gilt innerhalb einer Sequenz die Auswertungsreihenfolge „von links nach rechts“. Alle weiteren Festlegungen ergeben sich aus folgender Regel [Bause, Tölle 90]:

> Die Auswertungsreihenfolge zwischen sogenannten *sequence points* ist i.a. unbestimmt. Es ist nur garantiert, daß beim Erreichen eines solchen *sequence point* alle auszuführenden Anweisungen abgeschlossen sind bzw. werden. *Sequence points* sind z.B.
>
> | | |
> |---|---|
> | ; | (Semikolon) |
> | , | (Kommaoperator) |
> | \|\| | (logisches Oder) |
> | && | (logisches Und) |

Beliebig angeordnete und geschachtelte Anweisungssequenzen sind stets streng deterministisch. Sequenzen von Ausdrücken können jedoch durch Operatoren verbunden sein. Die Zusicherung der Reihenfolge in Ausdruckssequenzen bezieht sich dann lediglich auf die Ausdrücke einer Sequenz untereinander. Es ist daher durchaus möglich, daß zwischen den Auswertungen durch Kommata getrennter Ausdrücke auch Auswertungen von Ausdrücken, die nicht der selben Sequenz angehören, durchgeführt werden. Ein einfaches Beispiel verdeutlicht, daß diese mangelnde Festlegung leicht zu unterschiedlichen Ergebnissen führt:

```
int i, x;

x = (i = 1, ++i) + (i = 2, i);
```

Bei einer Auswertung der Teilausdrücke von links nach rechts wird 4 an x zugewiesen. Wird jedoch $i = 2$ nach $i = 1$, aber vor $++i$ ausgewertet, dann ist das Ergebnis 6. Wenn außerdem das ganz rechtsstehende i vor $++i$ ausgewertet wird, ergibt sich 5 als Endsumme.

Ausdruckssequenzen wären durch ihre Wertorientierung eigentlich für die Übersetzung von TML-Sequenzen prädestiniert. Ihre mangelnde Portabilität legt jedoch nahe, Anweisungssequenzen zu bevorzugen. Diese können Ausdruckssequenzen unter Verwendung von Temporärvariablen, die Werte über Anweisungsgrenzen hinweg transportieren, leicht nachbilden.

Argumentlisten in Funktionsaufrufen gleichen syntaktisch Ausdruckssequenzen, doch die Auswertungsreihenfolge der Argumentausdrücke ist durch den Programmiersprachenstandard *nicht* festgelegt. In der Praxis verwenden alle C-Implementationen entweder die Reihenfolge „von links nach rechts“ oder „von rechts nach links“ als festes Übergabeschema. Daher könnte die Kodegenerierung durch einen simplen Umschalter an den jeweils aktuellen C-Übersetzer angepaßt werden. Diese Methode ist jedoch nur zur Übersetzung in einer bestimmten Systemumgebung geeignet. Sie ist als Optimierung für statisch gebundene C-Programme vorgesehen.

Eine bestimmte Reihenfolge kann auf *portable* Weise erzwungen werden, indem für jedes Argument, das nicht als unmittelbarer Wert angegeben ist, eine Temporärvariable zur Zwischenspeicherung verwendet wird. Zum Beispiel wird der Aufruf *f(g(7), 8, h(9))* durch folgende Umformung deterministisch:

```
Value t1, t2;
```

```
t1 = g(7);
t2 = h(9);
f(t1, 8, t2);
```

Der so entstehende C-Programmtext ist auch als portable Repräsentation zur Programmübertragung in heterogenen Netzwerken geeignet. Es bietet sich daher an, TML-Kode zu diesem Zweck nicht nur lediglich zu lineariseren, sondern gleich nach C zu übersetzen. Dies gilt vor allem für Hochgeschwindigkeitsnetze, in denen es weniger auf die versendeten Datenmengen als vielmehr auf die Einsparung von Rechenzeiten ankommt.

C-Programmtext ist faktisch sowohl für Reflektionszwecke als auch für die Rücktransformation nach TML denkbar ungeeignet. Daher kann für reflektive Programme auf eine zusätzliche lineare Repräsentation von TML nicht verzichtet werden.

5.4 Grundstrukturen des generierten C-Kodes

In diesem Abschnitt werden die wichtigsten elementaren Bestandteile des C-Kodes vorgestellt. Dies geschieht im Rahmen der eingehenden Betrachtung des C-Kodes von Funktionsaufrufen und Ausnahmebehandlungen. Durch diese beiden Konstrukte werden die anspruchsvollsten der in der Gestaltung des C-Kodes auftretenden technischen Problemstellungen bedingt. Sie zeichnen sich insbesondere durch die Eigenschaft, Systemgrenzen durchdringen zu können, aus.

5.4.1 Funktionsaufrufe

Die eigentlichen Funktionsaufrufe werden stets durch einen Aufruf der Laufzeitsystemfunktion *thread_enterFrame* eingeleitet und durch *thread_leaveFrame* abgeschlossen. Der Prolog *thread_enterFrame* erfüllt folgende Aufgaben:

- Der Funktionsabschluß wird fixiert (vgl. Abschnitt 2.4), d.h. es wird sichergestellt, daß eine Kopie im Hauptspeicher vorliegt. Deren Hauptspeicheradresse wird nach Abschluß aller weiteren Aufgaben als Ergebnis zurückgegeben und dann an eine Temporärvariable zugewiesen.
- Es wird ermittelt, welche Kodeart der Funktionsabschluß enthält. Wenn es mehrere sein sollten (zunächst TML und Maschinenkode), wird aufgrund einer in der Benutzerumgebung einstellbaren Präferenz entschieden, wie weiter zu verfahren ist. Gegebenenfalls wird TML-Kode dynamisch in Maschinenkode übersetzt und als solcher ausgeführt. Es kann jedoch auch umgekehrt aus dem Maschinenkode heraus der Interpreter aufgerufen werden.
- Externe Funktionen werden erkannt und ihren Anforderungen entsprechend aufgerufen. Aus der Anwendungssicht besteht keinerlei Unterschied zwischen einem Aufruf einer eigenen und einer externen Funktion, weil letztere stets durch einen dem Tycoon-Standard entsprechenden Funktionsabschluß angebunden werden. Der so gekapselte Funktionsrumpf kann jedoch eine beliebige Gestalt annehmen. Es muß lediglich deren Grundprinzip bekannt und in *thread_enterFrame* „eingebaut“ sein.

- ▷ Noch nicht im Hauptspeicher präsenter Maschinenkode wird bei Bedarf geladen und dynamisch gebunden.
- ▷ Die Objektspeicherreferenz des Funktionsabschlusses wird auf einem globalen Stapel abgelegt, damit sie im Falle einer Ausnahme zur Rücknahme der Fixierung des Funktionsabschlusses verfügbar ist.

Die Aufgaben des Epilogs *thread_leaveFrame* sind:

- ▷ Die Rücknahme der Fixierung des Funktionsabschlusses.
- ▷ Das Entfernen der Referenz des Funktionsabschlusses von dem bewußten Stapel.

Der größte Nachteil dieses Verfahrens ist die relativ geringe Effizienz. Alle Funktionsaufrufe werden mindesten um Faktor 3 verlangsamt. Sehr viel größer ist der durchschnittliche Verlust allerdings nicht, weil der Objektspeicher bei der Rücknahme einer Fixierung das betreffende Hauptspeicherobjekt nicht unbedingt entfernt, sondern so lange wie möglich weiter zur Verfügung hält (*caching*).

Durch das beschriebene Verfahren werden alle bedeutsamen systemtechnischen Probleme gekapselt:

> Bei *jedem* Funktionsaufruf wird von jeglicher Speicherungsart, Lokalisation, Kodeart und dynamischen Transformation des Funktionskodes abstrahiert.

Die weiteren Anwendungsmöglichkeiten können überhaupt nicht unterschätzt werden. Zum Beispiel ist die Grundlage dafür geschaffen, daß Datenbankanfragen aufgrund einer reflektiven Untersuchung des TML-Kodes *ad hoc* auf folgende verschiedenen Weisen bearbeitet werden können: durch direkte Interpretation, durch dynamische Übersetzung in Maschinenkode oder durch einen entfernten Aufruf im Netzwerk, wobei wahlweise TML-Kode oder C-Programmtext übertragen werden kann.

Zur Veranschaulichung werden folgende TL-Anweisungen betrachtet:

```
let id(x :Int) = x;
id(2);
```

Der entsprechede C-Kode:

```
Value function_1_1(Value *g, Value p1)
{
  return thread_leaveFrame(p1);
}

Value function_1_0(Value *g)
{
  Value t_1, t_2, t_3;

  t_1 = runtime_newClosure(store_get(g[1], 0x11L), 0x1L);
```

```
  store_set(g[2], 0xb5L, t_1);
  t_3 = store_get(g[2], 0xb5L);
  t_2 = thread_enterFrame(t_3);
  t_1 = (*((Value (**)()) t_2)[0])(t_2, 0x9L);
  store_set(g[2], 0xb6L, t_1);
  t_1 = store_get(g[2], 0xb6L);
  return thread_leaveFrame(t_1);
}
```

Der Typ *Value* repräsentiert Maschinenworte (*typedef unsigned long Value;*). Die Funktion *function_1_1* entspricht dem Funktionsrumpf von *id*. Der Parameter *x* wird durch *p1* repräsentiert. Die Epilog-Funktion *thread_leaveFrame* steht am Ende einer jeden Benutzerfunktion und läßt das Funktionsergebnis passieren. Dadurch wird Kode eingespart, weil Funktionsaufrufe häufiger sind als Funktionsdeklarationen. Der zusätzliche Parameter *g* enthält die Hauptspeicheradresse des gefixten Funktionsabschlusses und ermöglicht so den Zugriff auf globale Werte.

Die Funktion *function_1_0* implementiert die aktuelle *Toplevel*-Funktion. Sie legt mit Hilfe der Laufzeitsystemfunktion *runtime_newClosure* einen neuen Funktionsabschluss an und weist diesen an die *Toplevel*-Variable mit dem (exemplarischen) Index 45 im *Toplevel*-Vektor zu. Dieser Zahl entspricht mit Markierungsbits (siehe Abschnitt 2.4) 181, was in hexadezimaler C-Notation $0xb5L$ ergibt. Der *Toplevel*-Vektor steht der *Toplevel*-Funktion stets als erste (und einzige) globale Variable zur Verfügung. Der entsprechende Index im Funktionsabschluß ist 2, so daß der Zugriff die Form *g[2]* hat. Der Aufruf *store_get(g[1], 0x11L)* ist ein Zugriff auf den Literalvektor der *Toplevel*-Funktion (vgl. Abbildung 3.1 auf Seite 24). Er extrahiert aus diesem den TML-Kode von *id*.

Die lokale Variable *t_2* enthält die Hauptspeicheradresse des gefixten Funktionsabschlusses von *id*. Ohne Typisierung würde ein Ausdruck der Form $t\_2[0]$ den Maschinenkode adressieren. Der dazugehörige *type cast* $(Value(**)())$ veranlaßt den C-Übersetzer, den Wert von *t_2* als Zeiger auf einen Vektor von Funktionszeigern zu interpretieren. Der Dereferenzierung des Funktionszeigers folgt dann die Parameterliste. Als erstes Argument wird *t_2* an den kanonischen Zusatzparameter *g* der aufgerufenen Funktion übergeben. Dann folgt der Wert 2 in Form von $0x9L$.

Am Schluß wird das Endergebnis in das *Toplevel*-Vektorelement mit dem Index 46, bzw. $0xb6L$ eingetragen, dort gleich anschließend wieder ausgelesen und an die Benutzerumgebung zurückgegeben.

5.4.2 Ausnahmebehandlungen

Die wohl interessanteste Kontrollstruktur ist die Ausnahmebehandlung. Abbildung 5.1 zeigt ihre Grundstruktur anhand eines einfachen, sowohl in Form von TL als auch TML und C angegebenen Beispiels. Es wird dabei angenommen, daß *error* eine beliebige zuvor zugewiesene lokale Variable sei, die auf einen Ausnahmewert verweist. Zur Implementation geschachtelter Ausnahmebehandlungen wird ein Stapelspeicher verwaltet, auf dem sogenannte „Programmkontextblöcke“ abgelegt werden. Ein Programmkontextblock enthält sämtliche Registerinhalte der CPU (*central processing unit*), insbesondere den Instruktionszeiger und den Pro-

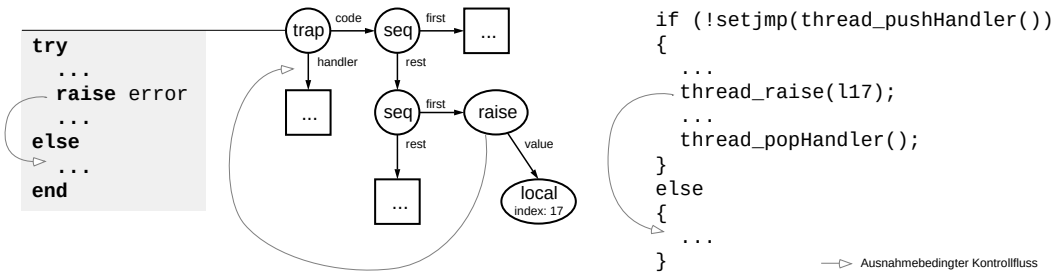

Abbildung 5.1: Ausnahmebehandlungen in TL, TML und C

grammstapelzeiger. Die Inhalte der FPU (*floating point unit*) werden unter *Sun OS* nicht gespeichert.

Jeder C-Programmblock, der eine Ausnahmebehandlung vorsieht, erzeugt mit Hilfe der Funktion *thread_pushHandler* ein neues Stapelelement, das dann durch einen *setjmp*-Aufruf mit einem frisch gewonnenen Programmkontextblock belegt wird. Die C-Standardfunktion *setjmp* liefert zur Steuerung des Kontrollflusses zunächst den Wert 0, im Falle einer Ausnahme jedoch stets einen anderen Wert.

Benutzerdefinierte Ausnahmen werden durch die Laufzeitsystemfunktion *thread_raise* ausgelöst, welche das oberste Element vom Stapel holt und in den entsprechenden Programmkontext zurückspringt. Sie verwendet dazu intern den C-Standardaufruf *longjmp(1)*. Als erstes macht sie jedoch die Fixierung aller durch diesen Rücksprung übergangenen Funktionen rückgängig (vgl. Abschnitt 5.4.1).

Die Funktion *thread_popHandler* wird an allen Stellen aufgerufen, an denen ein Block, der eine Ausnahmebehandlung vorsieht, verlassen wird, ohne daß eine Ausnahme auftritt. Sie entfernt die ungenutzten Programmkontextblöcke vom Stapel, die zwischenzeitlich durch den Kontrollfluß überholt sind.

Das beschriebene Schema ist mit dem des Interpreters verträglich. Zu Beginn seiner Aktivierung ruft dieser jeweils einmal *setjmp* auf und gewinnt dadurch die Kontrolle über Ausnahmen aus nicht interpretierten Programmteilen. Dabei kann es sich sogar um externe Routinen handeln, die nicht von Tycoon generiert worden sind, sofern sie *thread_raise* korrekt verwenden.

Vom Interpreter ausgelöste Ausnahmen werden intern durch andere Mechanismen gehandhabt. Sie werden jedoch in dem Moment, da sie auf eine Evaluationsgrenze zum Maschinenkode treffen, mittels *thread_raise* propagiert, so daß sich ein reibungsloses Miteinander von TML- und Maschinenkode ergibt.

5.5 Die Implementation der Maschinenkodegenerierung

Aus Sicht des übrigen Tycoon-Systems bildet die Maschinenkodegenerierung eine geschlossene Einheit. Das Modul *tmCompile* liefert die Schnittstelle zum Auslösen der Maschinenkodegenerierung und steuert den groben Ablauf, der in die folgende zeitlich getrennte Phasen gegliedert ist:

1. Die Übersetzung von TML in C-Fragmente (Funktion *assembleUnit* in Modul *tmAssemble*).

   Die Repräsentation der Fragmente ist durch die Modulschnittstelle *TMCode* spezifiziert. Zu ihrer Konstruktion dienen die Funktionen des Moduls *tmCode*. Sie bestehen jeweils aus einer oder mehreren fast vollständigen Anweisungen in Form von Zeichenketten und einer abstrakten Beschreibung ihrer Verknüpfung durch Kontrollstrukturen. Die Erzeugung und Kombination von Zeichenketten, die Einzelanweisungen entsprechen, sind im Modul *tmFormatC* zusammengefaßt. Unter seiner Verwendung ist der im Modul *tmAssemble* angesiedelte Grundalgorithmus vollkommen von jeglichen lexikalischen Details isoliert.

2. Das Zusammenstellen der C-Fragmente zu C-Programmtext(Funktion *compileUnit* in Modul *tmTargetC*).

   Die vorbereiteten Anweisungen der Fragmente werden nun sequentiell als Programmzeilen ausgegeben. Dabei werden verknüpfende syntaktischen Elemente eingefügt, (z.B. Semikola am Ende von Anweisungen und Schlüsselworte von Kontrollstrukturen). Außerdem werden die Funktions- und Variablendeklarationen generiert. Dazu erforderliche Bezeichner und diverse andere Elemente werden wiederum mit Hilfe von *tmFormatC* erzeugt. Der kanonische Ausgabealgorithmus für C-Fragmente ist im Modul *tmTargetC* implementiert. Zur formatierten Textausgabe in eine Datei wird das Modul *tmWriter* verwendet.

3. Die weitere Übersetzung des C-Programmtextes bis hin zum Maschinenkode (Funktion *writeUnit* in Modul *tmTargetC*).

   Diese Phase ist in Abhängigkeit von der Systemumgebung weiter untergliedert. Im Falle der *Sun Sparc*-Architektur und des *Sun OS*-Betriebssystems liegt der Programmtext in einer Datei mit dem Namenszusatz (*extension*) '.c' vor. Er wird dann mit Hilfe des C-Übersetzers *cc* in eine Objektdatei (Zusatz '.o') übersetzt. Anhand dieser erstellt schließlich der Linker *ld* ein sogenanntes *shared object* (Zusatz '.so'), auch *shared library* genannt. Dabei handelt es sich um eine dynamisch ladbare Bibliothek, die den ausführbaren Maschinenkode enthält.

Abbildung 5.2 zeigt die Zuständigkeiten der Module der Maschinenkodegenerierung.

5.5.1 Die Isolation der lexikalischen Details

Während der Übersetzung von TML in C-Fragmente wird zur jede Erzeugung oder Konkatenation von Zeichenketten stets eine Konstruktorfunktion des Moduls *tmFormat* verwendet. In diesem sind somit alle lexikalischen Details gekapselt. Deren Konzentration an einer Stelle ist für die Programmwartung und -modifikation ideal. Außerdem ist dieser Ansatz die wichtigste Voraussetzung der Unabhängigkeit des abstrakten Übersetzungsalgorithmus von einer bestimmten Zielsprache (vgl. Abschnitt 5.7).

Durch die Konstruktorfunktionen werden folgende Elemente erzeugt:

- ▷ der Datentyp zur Repräsentation eines Maschinenwortes (*unsigned long int*),

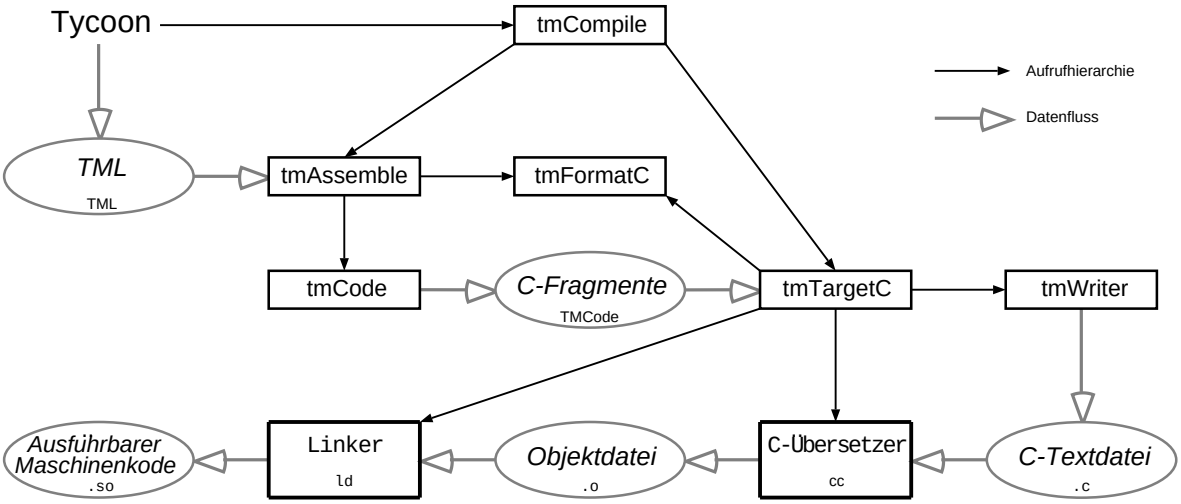

Abbildung 5.2: Schematischer Aufbau und Ablauf der Maschinenkodegenerierung

- Programmliterale dieses Typs,
- Variablennamen,
- Funktionsnamen,
- Anweisungen zum Verlassen von Funktionen und zur gleichzeitigen Wertrückgabe (*return*),
- Argumenttrennzeichen (Komma),
- Funktionsaufrufe,
- Zuweisungen,
- Namen von Sprungmarken (die zum Verlassen von Schleifen benötigt werden),
- Anweisungen zum Verlassen von Schleifen (*goto*, gefolgt von einer Sprungmarke),
- diverse Aufrufe von Laufzeitsystemfunktionen.

Um die Entstehung des C-Kodes zu veranschaulichen, werden im folgenden einige Konstruktorfunktionen angegeben. Manche bilden TML-Instruktionen direkt ab. Dies ist zum Beispiel beim lesenden Zugriff auf lokale Variablen, Parameter, globale Variablen und Felder von Speicherobjekten der Fall:

let *local(index :Int) :String = "l" <> fmt.int(index)*
let *parameter(index :Int) :String = "p" <> fmt.int(index)*
let *global(index :Int) :String = "g" <> "[" <> fmt.int(index) <> "]"*
let *get(object, index :String) :String =*
"store_get(" <> object <> ", " <> index <> ")"

Eine Zuweisung wird durch die textuelle Konkatenation ihrer linken Seite, des Zuweisungszeichens und ihrer rechten Seite gebildet:

```
let assign(variable, value :String) :String =  variable <> " = " <> value
```

Zum gezielten Verlassen von Schleifen ist in C aufgrund der Überladung des Schlüsselwortes *break* im allgemeinen die Verwendung von Sprungmarken und *goto*-Anweisungen notwendig. Das Modul *tmAssemble* verwaltet dazu Indizes, die eine überschneidungsfreie Benennung gewährleisten. Die Funktion *loopLabel* kapselt die indexabhängige Erzeugung der Bezeichner, die dann an *exitLoop* zur Implementation der TML-Instruktion **exit** übergeben werden.

```
let loopLabel(label :Int) :String =  "END_LOOP_" ¡¿ fmt.int(label)
let exitLoop(loopLabel :String) :String =  "goto " ¡¿ loopLabel
```

Die **runtime**-Instruktionen zum Aufruf von Laufzeitsystemfunktionen werden mit bereits als Zeichenketten vorliegenden Argumenten in komplette C-Funktionsaufrufe umgesetzt:

```
let applyRuntime(runtimeCode :TML.RuntimeCode  arguments :String) :String =
  case runtimeCode
  when machineBind then
    "tm_machineBind"
  when storeNew then
    ßtore_new"
  ...
  end
  <> "(" <> arguments <> ")"
```

Die von den Konstruktorfunktionen gelieferten Zeichenketten enthalten stets nur kleine syntaktische Einheiten, die nicht über Einzelanweisungen hinausgehen. Sie werden durch die Funktionen des Moduls *tmAssemble* (siehe 5.5.3) zu größeren Einheiten kombiniert, deren Repräsentation der nachfolgenden Abschnitt beschreibt.

5.5.2 Die Repräsentation der C-Fragmente

Vollwertige C-Anweisungen und alle übergeordneten syntaktischen Konstrukte werden mit Hilfe der Typen der in Anhang A.3 repräsentierten Modulschnittstelle *TMCode* repräsentiert. Anweisungen werden dabei ihrer Stellung in Kontrollstrukturen entsprechend verknüpft. Jede Kontrollstruktur stellt selbst wieder eine Anweisung dar. Daher werden alle zu unterscheidenden Arten von Anweisungen durch einen *rekursiven* varianten Tupeltyp (*Statement*) repräsentiert. Es werden folgende Varianten unterschieden:

nopStatement: die leere Anweisung.

verbatimStatement: Einzelanweisungen.

blockStatement: Anweisungssequenzen, sogenannte „Blöcke“.

altStatement: Fallunterscheidungen.

loopStatement: Schleifen.

trapStatement: Anweisungen mit Ausnahmebehandlungen.

Aufgrund der in Abschnitt 3.2 beschriebenen Konvention, daß der Ausgangspunkt einer jeden Evaluation ein Funktionsabschluß ist, sind alle Anweisungshierarchien einer Funktion zugeordnet. Diese repräsentiert folgender Typ:

```
and Function =Tuple
  name :String
  nParameters, nLocals, nTemps :Int
  body :Statement
end
```

Das Feld *body* verweist auf die Anweisungen des Funktionsrumpfes. Funktionen werden außerdem mit einem Namen versehen und ihre Variablen werden quantitativ erfaßt. Diese Angaben genügen später zur Generierung aller erforderlichen Deklarationen.

Die nächste Hierarchiestufe bilden Ausführungseinheiten:

```
and Entry = Tuple
  main :Function
  functions :queue.T(Function)
end
```

Dabei kann es sich entweder um ein Modul oder um eine *Toplevel*-Phrase handeln. In beiden Fällen handelt es sich um eine Gruppe von Funktionen, von denen jeweils eine besonders ausgezeichnet ist. Dies ist dann die Funktion zur Modulinitialisierung oder die *Toplevel*-Funktion.

Es können mehrere Ausführungseinheiten gemeinsam übersetzt werden, z.B. in Folge einer zusammenhängenden Eingabe mehrerer *Toplevel*-Phrasen. Damit ist die höchste Stufe erreicht - die Übersetzungseinheit:

```
Let Rec Unit = Tuple entries :queue.T(Entry) end
```

Wie schon bei der Implementation von TML (vgl. Abschnitt 3.7) sind Konstruktorfunktionen zur Erzeugung von Werten der Repräsentationstypen vorgesehen.

5.5.3 Die Übersetzung von TML in C-Fragmente

TML-Kode wird durch rekursive Typen repräsentiert (vgl. Abschnitt 3.7 und Anhang A.2). Dies trifft zwar auf die in Abschnitt 5.5.2 beschriebenen C-Fragmente ebenfalls zu, sie weisen dabei jedoch häufig sequentielle Unterstrukturen auf. Diese werden durch Schlangen repräsentatiert, was die erneute Anwendung der bereits in der Zwischenkodegenerierung erfolgreichen Symbiose von funktionalem und imperativem Programmierstil erlaubt (vgl. Abschnitt 4.5.1 und 4.5.7). Im vorliegenden Fall bedeutet dies, daß die TML-Strukturen durch rekursiven Abstieg durchlaufen werden und daß die Konstruktion der Ergebnisse zum großen Teil durch

Seiteneffekte, d.h. durch Akkumulation in Schlangen, erfolgt. Die jeweils aktuellen Schlangen werden wiederum durch Parameter im rekursiven Abstieg weitergereicht.

Für die Ausführungseinheiten einer Übersetzungseinheit und für die Funktionen einer Ausführungseinheit wird jeweils eine eigene Schlange angelegt (vgl. die Typdeklarationen in Abschnitt 5.5.2). In den beiden Fällen erfolgt eine einfache sequentielle Anordnung.

Einfache TML-Instruktionen wie z.B. **immediate** oder **local** werden direkt in C übersetzt. Die meisten TML-Instruktionen sind jedoch komplex, d.h. sie verweisen auf weiteren TML-Kode. In zahlreichen konventionellen Übersetzersystemen wird der Kode rekursiv durchlaufener komplexe Ausdrücke durch eine Stapelverwaltung angeordnet [Jähnichen et al. 78]. Jeder komplexe Ausdruck wird dabei in seine Teilausdrücke zerlegt, die jeweils an Temporärvariablen zugewiesen und *vorangestellt* ausgegeben werden. Die letzte Anweisung, welche die am weitesten außenliegende Verknüpfung enthält, wird so lange auf dem Stapel zurückgehalten, bis alle Anweisungen für ihre Teilausdrücke ausgegeben worden sind. Dieses Verfahren erscheint auf den ersten Blick sehr geeignet, denn es sieht genau in der beabsichtigten Weise den Gebrauch von Temporärvariablen (vgl. Abschnitt 5.2) vor.

Die übergeordneten Programmeinheiten werden jedoch nicht berücksichtigt:

- Die C-Deklarationen von Temporärvariablen sind den weiteren Anweisungen des betreffenden C-Funktionsrumpfes voranzustellen, ihre Anzahl steht jedoch erst nach einem vollständigen Durchlaufen des TML-Funktionsrumpfes fest. Daher wäre bei einem Stapelverfahren stets die gesamte Funktion zwischenzuspeichern.

- TML-Funktionen können geschachtelt sein, C-Funktionen jedoch nicht. Deshalb muß der C-Kode von TML-Funktionen grundsätzlich isoliert werden. Da alle Funktionen außer der *Toplevel*-Funktion geschachtelt sind, wäre bei einem Stapelverfahren häufig fast die gesamte Übersetzungseinheit zwischenzuspeichern.

Als Alternativen zur Stapelverwaltung kommen *backpatching* (siehe Abschnitt 4.2) oder eine zweiphasige Übersetzung in Frage. Zur Lösung des zweiten Problems eignet sich auch eine simple Abarbeitungsliste, die TML-Funktionsrümpfe referenziert. Deren Übersetzung kann suspendiert werden, weil eine C-Funktion jeweils nur einem TML-Funktionsrumpf entspricht und *nicht* einer Funktionsabstraktion.

Die Reihenfolge der C-Funktionen ist nur dann beliebig, wenn sie in der späteren Objektdatei als externe Symbole exportiert werden, was nicht wünschenwert ist, da so unnötige Namenskonflikte provoziert werden. Unter der Verwendung des C-Schlüsselwortes **static** bleibt ihr Sichtbarkeitsbereich auf die jeweilige Übersetzungseinheit beschränkt. Sie müssen dann allerdings jeweils vor all ihren Verwendungen deklariert werden.[1]. Dieser Zwang kann durch wohlplazierte Deklarationen der Funktionsköpfe umgangen werden, was jedoch einigen Verwaltungsaufwand bedingt.

Alle erwähnten Teilprobleme werden durch die Verwendung der in Abschnitt 5.5.2 beschriebenen Zwischenrepräsentation und zwar insbesondere durch den Einsatz von Schlangen gelöst. Die wichtigsten Vorteile sind dabei:

[1] In diesem Fall wird im C-Jargon genaugenommen von einer „Definition“ als Spezialfall der Deklaration gesprochen, weil der Funktionsrumpf mit angegeben ist Eine pure Deklaration besteht dagegen nur aus dem Funktionskopf.

- ▷ Alle Schlangen werden in der Halde alloziert, so daß die Gefahr eines Speicherüberlaufes sehr gering ist.
- ▷ Bei der Erfassung der Funktionen entsteht automatisch die erforderliche Reihenfolge.
- ▷ Die Festlegung der Anzahl der zu deklarierenden Temporärvariablen ist nicht mehr an eine bestimmte Vorgehensweise gebunden.
- ▷ Die C-Fragmente können bei gleichzeitiger Verwendung mehrerer Schlangen mit geringem Aufwand sehr flexibel kombiniert werden.

Ein weiterer wichtiger Aspekt ist die Unabhängigkeit des Übersetzungsalgorithmus von der Zielsprache. Wie in Abschnitt 5.5.1 beschrieben, können die lexikalischen Details von Einzelanweisungen leicht isoliert werden. Für eine einzelne Zielsprache wäre dies auch bezüglich der Kontrollstrukturen möglich. Die Syntax von Programmiersprachen ist jedoch gerade in Bezug auf Kontrollstrukturen typischerweise *strukturell* und nicht nur lexikalisch sehr verschieden. Deshalb ist eine abstrakte Zwischenrepräsentation der Kontrollstrukturen zur Abkapselung ihrer textuellen Umsetzung vorzuziehen.

Das gewählte zweiphasige Verfahren ist erheblich flexibler in der Anordnung des C-Kodes als alle anderen betrachteten Verfahren. Es ist aber dennoch eng mit dem Stapelverfahren verwandt. Letzteres wird mit Hilfe von C-Fragmenten des Typs *Statement*, die in der Variante *blockStatement* auftreten, gewisserweise simuliert. Sie enthalten ein Feld folgenden Typs:

```
and Block = Tuple
  level :Int
  var nTemps :Int
  statements :queue.T(Statement)
end
```

Die „Ausgabe“ von Anweisungen erfolgt durch Einfügen in die Schlange des Feldes *statements*. Die vorteilhaften algorithmischen Aspekte des Stapelverfahrens können so voll genutzt werden, ohne die Nachteile in Kauf nehmen zu müssen.

Im Feld *nTemps* wird die Anzahl der in einem Block benötigten Temporärvariablen mitgerechnet, und das Feld *level* enthält die Blockschachtelungstiefe. Letztere zwei Angaben erlauben die Ausnutzung einer Spezialität von C: lokale Blöcke. Wahlweise kann für jeden C-Fragment-Block ein tatsächlicher C-Block generiert werden, in dem seine eigenen Temporärvariablen lokal deklariert sind. Zur Angabe eines Beispiels wird folgende TL-Anweisung betrachtet (vgl. Abschnitt 5.4.1):

```
begin
  let x = id(4)
end
```

Die Funktion *f* sei beliebig. *Mit* lokalen Blöcken ensteht folgender Programmauszug:

```
...
Value l1;

{
  Value t_1_1;

  {
    Value t_2_1, t_2_2;

    t_2_2 = store_get(g[2], 0xb5L);
    t_2_1 = thread_enterFrame(t_2_2);
    t_1_1 = (*((Value (**)()) t_2_1)[0])(t_2_1, 0x11L);
  }
  l1 = t_1_1;
}
...
```

Die lokale C-Variable *l1* geht aus der TL-Variablen x hervor; *thread_enterFrame* bereitet den Funktionsaufruf vor, der in der darauffolgenden Zeile steht. Dies wird im einzelnen in Abschnitt 5.4.1 erklärt. Im vorliegenden Zusammenhang interessieren nur die Tenporärvariablen. Sie haben zwei jeweils „Indizes“, die durch '_' getrennt werden. Der erste Index steht für die Blockschachtelungstiefe und unterscheidet so Sichtbarkeitsgrenzen, Der zweite Index unterscheidet die Variablen eines Blockes voneinander. Es folgt der gleiche C-Auszug in der Version *ohne* lokale Blöcke:

```
...
Value l1;
Value t_1, t_2, t_3;

t_3 = store_get(g[2], 0xb5L);
t_2 = thread_enterFrame(t_3);
t_1 = (*((Value (**)()) t_2)[0])(t_2, 0x11L);
l1 = t_1;
...
```

Die Anzahl und Verwendung der Temporärvariablen verhält sich vollkommen analog zur ersten Version. Als Benennungsschema wird jedoch eine einfache Zählweise verwendet, weil dann wiederum die Anzahl der Variablen zur vollständigen Spezifikation ihrer Deklarationen genügt.

Zur Verwaltung der Temporärvariablen wird das gleiche Verfahren wie in der Zwischenkodegenerierung angewandt. Die Funktionen *markTemps* und *releaseTemps* nutzen dabei abermals implizit den Programmstapel. Folgendes Beispiel deutet das Prinzip an:

```
...
m = markTemps(state)
(* Alle hier in der Mitte belegten Temporärvariablen... *)
...
releaseTemps(state m)
(* ... stehen hier unten erneut zur freien Verfügung.*)
...
```

Diese Methode wird in Abschnitt 4.5.2 ausführlich beschrieben Zur schnellen Übersicht eignet sich die Betrachtung von Abbildung 4.10 auf Seite 76.

5.5.4 Die Transformation der C-Fragmente zu C-Programmtext

Die Transformation der als Zwischenrepräsentation vorliegenden Fragmente in fertigen Programmtext ist zielsprachenspezifisch. Im folgenden wird der Algorithmus für C beschrieben.

Zur Textausgabe wird das Modul *tmWriter:TMWriter* verwendet, das eine primitive Formatierung erlaubt. Seine wichtigste Funktion ist *line*. Sie gibt eine Zeichenkette als Zeile aus. Die Funktion *newLine* erzeugt eine Leerzeile, *indent* und *outdent* regeln die Einrückung, *file* öffnet eine Datei und *close* schließt sie wieder. Diese einfachen Mittel genügen vollständig. Sämtliche in der vorliegenden Arbeit als Beispiele aufgeführten C-Texte sind mit ihrer Hilfe entstanden.

Die Funktion *writeUnit* des Moduls *tmTargetC:TMTarget* leistet die Ausgabe von Übersetzungseinheiten. Sie erzeugt stets als erstes einige Standarddeklarationen, wie z.B. die des Maschinenworttyps *Value* und der Laufzeitsystemfunktionen:

```
typedef unsigned long Value;
extern Value store_new();
...
```

Dann folgen die Ausführungseinheiten, eine nach der anderen. Diese gehen wiederum aus der einfachen Aneinanderreihung ihrer Funktionen hervor. Eine einzelne Funktion wird wie folgt ausgegeben:

```
let writeFunction(s :State  f :Function  targetLanguage :TargetLanguage) :Ok =
 begin
   writeFunctionHead(s  f  targetLanguage)
   writeFunctionParameterDeclarations(s  f  targetLanguage)
   openBlock(s)
     writeFunctionLocalDeclarations(s  f)
     writeFunctionTempDeclarations(s  f)
     writeStatement(s  f.body  false)
   closeBlock(s)
   newLine(s)
 end
```

Der Parameter *s* enthält die Referenz der Ausgabedatei und diverse Zustandsvariablen. In zahlreichen anderen Systemen werden globale Variablen für diese Zwecke verwendet. Dies geschieht in der vorliegenden Arbeit *grundsätzlich* nicht, denn globale Variablen verhindern in aller Regel sowohl die Verwendung eines Programmes als Subsystem als auch seine (sinnvolle) Ausführung durch parallele Prozesse mit gemeinsamem Adreßraum (sogenannte *threads* oder auch *lightweight processes*). Diese unnötigen Einschränkungen können stets durch einen Parameter für „globale“ Daten vermieden werden.

Die Funktionen *openBlock* und *closeBlock* erzeugen einen C-Block, d.h. die entsprechenden geschweiften Klammern und benutzen *indent* und *outdent* zur Einrückung. Eigentlich ist letztere überflüssig, aber in der Entwicklungsphase der Maschinenkodegenerierung ist die so verbesserte Übersichtlichkeit sehr hilfreich.

Der typische Aufbau einer C-Funktion ist folgender:

Funktionskopf
'{',
  Variablendeklarationen

  Anweisungen des Funktionsrumpfes
'}'.

Diesen Elementen können die entsprechenden Funktionsaufrufe in *writeFunction* durch Namensvergleich zugeordnet werden. Dabei wird dann allerdings die Funktion *writeFunctionParameterDeclarations* noch nicht erfaßt. Sie tritt nur bei der Erzeugung von sogenanntem K&R-C [Kernighan, Ritchie 77], der älteren Version dieser Sprache, wirksam in Aktion. Im Falle von ANSI-C [Kernighan, Ritchie 88], der neueren, standardisierten Version von C bleibt sie ohne Effekt. Der Parameter *targetLanguage* unterscheidet die beiden C-Dialekte. Er wird auch an die Funktion *writeFunctionHead* weitergereicht, weil nur in ANSI-C vollständige Parameterdeklaration im Funktionskopf erfolgen. Zur Veranschaulichung dient folgende TL-Funktion, wobei nur die in diesem Zusammenhang relevanten Teile betrachtet werden:

fun(*x, y :Int, z :Bool) :Int* = **begin** ... **end**

Die Übersetzung nach ANSI-C ergibt:

```
static Value function_1_1(Value *g, Value p1, Value p2, Value p3)
{
  ...
}
```

In K&R-C werden zunächst lediglich die Bezeichner der Variablen aufgezählt; ihre Deklarationen folgen dann außerhalb der Parameterliste:

```
static Value function_1_1(g, p1, p2, p3)
  Value *g, Value p1, Value p2, Value p3;
{
  ...
}
```

Jeder ANSI-C-übersetzer akzeptiert auch K&R-C, aber nicht umgekehrt. Der Maschinenkode verschiedener C-Übersetzer eines Betriebssystems ist jedoch sehr häufig aufrufkompatibel, so daß zur Laufzeit bedenkenlos hin und her gewechselt werden kann. Zum Beispiel sind die unter *Sun OS* vorhandenen Übersetzer *gcc* für ANSI-C und *cc* für K&R-C diesbezüglich verträglich.

Die Anweisungen von Funktionsrümpfen werden durch die rekursive Funktion *writeStatement* ausgegeben, die damit den Hauptteil der gesamten Ausgabe leistet. Sie nimmt eine Fallunterscheidung anhand der Varianten des in Abschnitt 5.5.2 beschriebenen Typs *Statement* (siehe auch Anhang A.3) vor und reichert die auszugebenden Anweisungen entsprechend um restliche syntaktische Einheiten an. Folgende Auszüge mögen dies veranschaulichen:

```
...
when verbatimStatement with v then
  line(s  v.verbatim <> ";")
...
when trapStatement with t then
  line(s  "if (!setjmp(thread_pushHandler()))")
  openBlock(s)
    writeStatement(s  t.code  false)
    line(s  "thread_popHandler();")
  closeBlock(s)
  line(s  "else")
  openBlock(s)
    writeStatement(s  t.handler  false)
  closeBlock(s)
...
```

Die Semikola werden erst in dieser späten Phase angebracht, wodurch einige Flüchtigkeitsfehler vermieden werden. Ein exemplarisches Endergebnis der Variante *trapStatement* ist aus Abbildung 5.1 auf Seite 106 ersichtlich.

5.5.5 Die Übersetzung von C in Maschinenkode

Eine besondere Anforderung an den erzeugten Maschinenkode bedingt zahlreiche Abhängigkeiten vom Betriebssystem: er muß *dynamisch* in das bereits laufende Tycoon-System eingebunden werden können. Dies ist zum einen notwendig, weil er typischerweise erst zur Laufzeit generiert wird und zum andern weil Endbenutzern nicht die Möglichkeit verschafft wird, das Laufzeitsystem statisch zu verändern.

Auf welche Weise die genannte Abforderung erfüllt werden kann, läßt sich nur im Einzelfall genau klären, da die heutigen Betriebssysteme diesbezüglich sehr unterschiedliche Voraussetzungen bieten. Bei der Erstimplementation von Tycoon unter *Sun OS* werden dynamisch ladbare Bibliotheken (*dynamic libraries*) erzeugt. Sie werden auch *shared libraries* oder *shared objects* genannt, denn ihr ursprünglicher Anwendungszweck ist die gemeinsame Nutzung durch mehrere Programmen in nur einer Laufzeitinstanz. Da dies vor allem im Mehrbenutzerbetrieb zu drastischen Speicherplatzeinsparungen führt, sind *shared libraries* zumindest für Mehrbenutzerbetriebssysteme nicht ungewöhnlich. Vergleichbare Möglichkeiten existieren zum Beispiel unter *IBM AIX* und *VAX VMS*.

Bei der Benutzung und Erzeugung von *shared objects* sind folgende Bedingungen zu beachten:

- ▷ Der Maschinenkode muß relokatibel, d.h. frei im Hauptspeicher verschiebbar sein.
- ▷ Es ist stets nur eine Instanz ladbar. Ihre Freigabe wird per *reference counting* verwaltet.
- ▷ Programmliterale wie Zeichenketten können nur dann verwendet werden, wenn die benutzten *shared objects* bereits zur *statischen* Linkzeit angegeben werden.[2] Im Hinblick auf die meisten anderen Systeme sollte auf Programmliterale ohnehin verzichtet werden.
- ▷ Globale Variablen sind zulässig, sollten aber ebenfalls im Hinblick auf weniger leistungsfähige Systeme vermieden werden.

Die Erzeugung von *shared objects* aus C-Quelltext geschieht unter *Sun OS* in zwei Schritten. Zuerst wird durch einen C-Übersetzer (z.B. *cc*, *gcc*) eine Objektdatei generiert. Diese wird dann durch den Linker *ld* zu einem fertigen *shared object* aufbereitet. Zur Laufzeit können *shared objects* mit Hilfe der Betriebssystemfunktion *dlopen*, die den dynamischen Linker aufruft, geladen werden. Die Funktion *dlsym* liefert die Speicheradressen der Daten, die exportierten Symbolen entsprechen.

Jeder Tycoon-Übersetzungseinheit entspricht jeweils ein *shared object*. Zwecks Ausführung werden mit Hilfe von *dlsym* die Funktionszeiger der Hauptfunktionen der Ausführungseinheiten ermittelt. Daraufhin können diese Zeiger dereferenziert und als Funktionen aufgerufen werden. Alles weitere läuft dann besondere Einschränkungen durch das Betriebssystem ab.

5.6 Optimierungen

Eine wirklich effektive Optimierung ist nur unter Berücksichtigung der spezifischen Bedingungen einer bestimmten Hardware-Architektur möglich. Da die Notwendigkeit umgebungsabhängiger Wartung des Tycoon-Systems jedoch so weit wie möglich einzuschränken ist, wird die Optimierung vollständig als externer Dienst angesehen.

Daß der delegierte Aufwand erheblich ist, belegt z.B. die Betrachtung des optimierenden C-Übersetzers von *Sun* für die *Sun Sparc*-Hardware-Architektur. Die ihm zugrundeliegenden langjährigen Erfahrungen mehrerer professioneller Optimierungsspezialisten sind durch seine Verwendung direkt für Tycoon nutzbar. Er verwendet die Repräsentationssprache *Sun IR* (*intermediate representation*) [Hall, Barry 90; Muchnick 90], für die sein *globaler* Optimierer *iropt* folgende Maßnahmen durchführt: Registerzuweisung, *loop-invariant code motion*, *induction-variable strength reduction*, *common subexpression elimination*, *copy propagation*, *dead code elimination*, *loop unrolling* und *tail recursion elimination*.

[2] Ein zwingender Grund für diese Einschränkung ist bisher auch unter Berücksichtigung aller bekannten Details nicht auszumachen. Interessanterweise ist die Verwendung von Zeichenkettenliteralen in einem rein dynamischen *shared object* nämlich doch möglich, wenn in ihm an irgendeiner Stelle die Funktion *printf* aufgerufen wird. Damit soll angedeutet sein, wie fintenreich der Umgang mit nicht endbenutzernahen Diensten von Betriebssystemen häufig ist. Auf derartige Tricks sollte man sich jedoch zumindest bei der Implementation von Datenbanksystemen keinesfalls verlassen.

Außerdem wird auf den daraufhin erzeugten Maschinenkode ein *peephole optimizer* zur Erledigung folgender Aufgaben angewandt: Registerverbindungen, Anweisungs-Scheduling, Schleifenumkehrung, Quersprünge, Beseitigung unnötiger Sprünge, Beseitigung unnötiger Speicher- und Ladebefehle, *constant propagation* und erneute *dead code elimination*.

Recht hinderlich für diese Optimierungen sind allerdings die Aufrufe der Laufzeitsystemfunktionen, weil ihre Bedeutung von einem externen Optimierer nicht erfaßt werden kann. Dies gilt insbesondere für die Funktionsaufrufe des TSP zum Zugriff auf persistente Datenobjekte. Da diese absolut unvermeidlich sind, müßte ihre Optimierung bereits *vor* der C-Generierung stattfinden. Eine sinnvolle Erweiterung des Tycoon-Systems wäre demnach ein zwischen der TML- und der C-Generierung plazierter TML-Optimierer. Aufgrund der Maschinenunabhängigkeit von TML wäre er absolut portabel einsetzbar und der Aufwand seiner Erstellung somit von dauerhaftem Nutzen.

5.7 Die Umstellung auf andere Zielsprachen

TML besteht wie fast alle Zwischensprachen aus wenigen primitiven Konstrukten. Zur Kodegenerierung wird daher nur ein geringer Teil der in C verfügbaren Konstrukte benötigt. Die Beobachtung, daß diese fast alle zum Standardrepertoire imperativer Mehrzweckprogrammiersprachen gehören, ist der Ausgangspunkt eines allgemeineren Ansatzes zur Maschinenkodegenerierung.

Aus den in Abschnitt 5.1 genannten Gründen wird C als Zielsprache favorisiert. Die tatsächliche Verfügbarkeit einer bestimmten Programmiersprache hängt jedoch von diversen Umständen ab. Nicht zuletzt die ökonomischen Rahmenbedingungen sind meist von entscheidender Bedeutung. Durch die Eigenschaft des Tycoon-Systems, mit geringem Aufwand auf verschiedene Zielsprachen umgestellt werden zu können, wird die Anzahl der in Betracht kommenden Übersetzersysteme drastisch erhöht.

In einem Tycoon-System können sogar *mehrere* Zielsprachen installiert sein, zwischen denen *zur Laufzeit* vor jeder Übersetzungsphase gewechselt werden kann. Auf diese Weise ist z.B. die *ad hoc*-Auswahl des maximal effizienten Systems für eine Spezialaufgabe möglich. Am wichtigsten ist jedoch die Förderung der Offenheit des Tycoon-Systems:

> Zusätzliche Zielsprachen vergrößern die Auswahl an Programmierschnittstellen und erweitern dadurch die Möglichkeiten zur Anbindung externer Dienste.

Die Technik des Austauschens der Zielsprache ist sehr einfach und elegant: es wird lediglich eine Modulreferenz gewechselt. Diese Lösung demonstriert eindrucksvoll die Ausdrucksmächtigkeit der Kombination von Polymorphie und Funktionen höherer Ordnung. Denn auf diesen Grundkonzepten basiert die Eigenschaft der Module, Werte „erster Klasse“ zu sein (vgl. Abschnitt 2.3.6).

Zur Übersetzung von TML werden gemäß Abschnitt 5.5.1 und 5.5.2 folgende Konstrukte benötigt:

- die Funktionsdeklaration,
- Funktionsaufruf,
- Funktionszeiger,
- die Argumentübergabe mit einfacher Kopiersemantik (*call-by-value*),
- die Funktionswertrückgabe,
- funktionslokale Variablen,
- die Zuweisung,
- die wiederholte Ausführung (Schleife),
- das gezielte Verlassen einer bestimmten Schleife,
- die Fallunterscheidung,
- die Typanpassung (*type cast*),
- ein Basistyp, der Maschinenworte beschreibt,
- Programmliterale für den gesamten Wertebereich von Maschinenworten,
- die indiziert indirekte Adressierung,
- Maßnahmen zur Ausnahmebehandlung.

Alle aufgezählten Konstrukte sind z.B. in C, Modula-2, Pascal und PL/1 direkt oder zumindest mittelbar[3] verfügbar. Sie unterscheiden sich dabei lediglich in ihrer jeweiligen lexikalischen Ausgeprägung, sowie in einigen syntaktischen Eigenheiten. Diese Spezifika werden von den Schnittstellen *TMFormat* und *TMTarget* isoliert (siehe Abschnitt 5.5.1, bzw 5.5.2). Letztere sind für jede Zielsprache erneut formal und sinngemäß korrekt durch entsprechende Module zu implementieren. Der im Modul *tmAssemble* implementierte abstrakte Übersetzungsalgorithmus bleibt davon jedoch stets vollkommen unberührt.

Um die Auswechselbarkeit der Zielsprache zu belegen, werden nachfolgend einige spezielle Konstrukte als Beispiele angeführt. Dabei wird jeweils der C-Implementation eine entsprechende Lösung für Modula-2 gegenübergestellt. Zunächst wird das Modul *tmFormatC:TMFormat*, in dem lexikalische Besonderheiten gekapselt sind, betrachtet.

Eine Zuweisung wird durch die textuelle Konkatenation ihrer linken Seite, des Zuweisungszeichens und ihrer rechten Seite gebildet:

[3] „Mittelbar“ bezieht sich im wesentlichen auf die Ausnahmebehandlungen. Sie können auf zwei Weisen realisiert werden: mit Hilfe einer C-Schnittstelle unter Verwendung der in C vorhandenen Konstrukte (siehe Abschnitt 5.4.2) oder durch Generierung von Fallunterscheidungen, mit denen jeder Funktionsaufruf umgeben wird.

```
let assign(variable, value :String) :String =  variable <> " = " <> value
```

Das Zuweisungszeichen '=' ist in Modula-2 durch ':=' zu ersetzen.

Zum gezielten Verlassen von Schleifen ist in C aufgrund der Überladung des Schlüsselwortes *break* die Verwendung von Sprungmarken und *goto*-Anweisungen notwendig. Das Modul *tmAssemble* verwaltet dazu Indizes, die eine überschneidungsfreie Benennung gewährleisten. Die Funktion *loopLabel* kapselt die indexabhängige Benennung, die dann an *exitLoop* zur Implementation der TML-Instruktion **exit** übergeben wird.

```
let loopLabel(label :Int) :String =  "END_LOOP_" <> fmt.int(label)
let exitLoop(loopLabel :String) :String =  "goto " <> loopLabel
```

In Modula-2 kann die TML-Instruktion **exit** direkt auf das nicht überladene Schlüsselwort **EXIT** abgebildet werden:

```
let loopLabel(label :Int) :String =  ""
let exitLoop(loopLabel :String) :String =  "EXIT"
```

Alle gebräuchlichen imperativen Mehrzweckprogrammiersprachen sind mehr oder weniger streng typisiert. Diese Eigenschaft ist jedoch zum Zwecke der Kodegenerierung nicht erforderlich. Sie ist sogar sehr hinderlich, da vorwiegend uninterpretierte Daten flexibel zu manipulieren sind. Besonders betroffen sind Funktionsaufrufe, da diese stets mit Hilfe von Funktionszeigern erfolgen. Ein typischer Funktionsaufruf des generierten C-Kodes hat z.B. folgende Form (vgl. Abschnitt 5.4.1):

```
t_1 = (*((Value (**)()) t_2)[0])(t_2, l_1, l_2);
```

In Modula-2 ist für solche Aufrufe zunächst die Deklaration eines Typs und einer Hilfsvariablen notwendig:

```
TYPE
  Value = LONGINT;
  Cheat = ^RECORD
            CASE :INTEGER OF
            | 1: PROCEDURE(Value):Value;
            | 2: PROCEDURE(Value,Value):Value;
            | 3: PROCEDURE(Value,Value,Value):Value;
              ...
            END;
          END;
...
VAR
  c :Cheat;
...
  c := P(t2);
  t1 := c^.p3(t2, l1, l2);
```

Offensichtlich genügt es, für jede unterschiedliche Parameteranzahl, die im Kode auftritt, jeweils eine Variante im Typ *Cheat* vorzusehen.

In Pascal dagegen können Funktionszeiger nicht explizit manipuliert werden. Aber es existiert immerhin die Möglichkeit, Funktionen als Argumente zu übergeben. Wenn das betreffende Pascal-System mindestens eine externe Programmierschnittstelle hat, was fast immer der Fall ist, dann kann das Typsystem wie folgt überlistet werden:

```
type
  Value        = longint;
  ValuePointer = ^Value;
...
function call3(f, g, p1, p2 : Value) : Value; external;
...
t1 := call3(ValuePointer(t2)^, t2, l1, l2);
```

Die Funktion *call3* ruft eine externe Funktion auf, die den von *t2* referenzierten Wert (Offset 0 im Funktionsabschluß) als korrekt typisierten Funktionszeiger auffaßt und aufruft. Diese externe Funktion kann naheliegender Weise in Pascal implementiert sein, es kommt jedoch auch jede andere Sprache in Frage, sofern die Parameterkonventionen des Pascal-Systems eingehalten werden. In Pascal bietet sich folgende Implementation an:

```
unit callN;  interface

type
  Value = longint;
...
function call3(function f(g, p1, p2 : Value) : Value; g, p1, p2 : Value) : Value;
...
implementation
...
function call3(function f(g, p1, p2 : Value) : Value; g, p1, p2 : Value) : Value;
begin
  call3 := f(g, p1, p2);
end;
...
end.
```

Eine solche „unit“ ist eine unselbständige Übersetzungseinheit mit Bibliothekscharakter. Es ist zu beachten, daß in obigem Programm *keine* **use**-Anweisung stehen darf, die sich auf diese *unit* bezieht, weil sonst ein Typkonflikt auftritt. In diesem Zusammenhang sei zugestanden, daß einige Annahmen bezüglich der jeweiligen Pascal-Implementation vorausgesetzt werden, insbesondere die Repräsentation ungeschachtelter Funktionen durch einfache Funktionszeiger. Die beschriebene Methode ist jedoch vertretbar, weil sie stets entweder durchgängig fehlerlos funktioniert oder aber überhaupt nicht.

Die Laufzeit pro Funktionsaufruf wird in Pascal gegenüber C scheinbar verdoppelt. In dieser Rechnung sind jedoch auch die Pascal-Äquivalente des Prologes *thread_enterFrame* und des

Epiloges *thread_leaveFrame* (vgl. Abschnitt 5.4.1) zu berücksichtigen. Da es sich bei *call3* im Gegensatz zu diesen lediglich um einen reines Durchreichen typischerweise wenig zahlreicher Argumente handelt, dürfte der tatsächliche Effizienzverlust sehr viel geringer als Faktor 2 sein.

Kapitel 6

Zusammenfassung

Eine eingehende Beurteilung des generierten Kodes im Rahmen größerer Anwendungen ist erst nach erfolgter Selbstimplementierung des Tycoon-Systems möglich. Erste Laufzeitmessungen lassen jedoch bereits einige grundsätzliche Schlüsse zu. Diese werden im nachstehenden Abschnitt erörtert. Anschließend werden die Erfahrungen mit dem Quest-System geschildert, das als Entwicklungsumgebung für die Erstimplementation des Tycoon-Systems dient. Diese sind insbesondere aufgrund der Ähnlichkeit von Quest und TL von dauerhafter Bedeutung. An die Beschreibung des derzeitigen Standes der Implementation schließt die Aufzählung der vorgesehenen Erweiterungen an. Am Schluß werden die wesentlichen Ergebnisse der Arbeit zusammengefaßt und durch einen Ausblick auf weitere Entwicklungen ergänzt.

6.1 Das Laufzeitverhalten des Kodes

Zur Abschätzung der Performanz sind einige einfache Tests durchgeführt worden, wie z.B. die Berechnung von Fibonacci-Zahlen oder das Traversieren von Listen. Eine genaue Quantifizierung erfordert zwar weitaus umfangreichere Untersuchungen, die Erwartungen bezüglich qualitativer Unterschiede haben sich jedoch bereits unzweifelhaft bestätigt.

Die Performanz des TML-Interpreters ist etwas schlechter als die des Quest-Interpreters, was nicht überrascht, da letzterer schließlich keine Objektspeicherschnittstelle zu unterstützen hat. Deutlich überlegen ist die Ausführungsgeschwindigkeit des Maschinenkodes. Sie wird außerdem erwartungsgemäß durch Einschalten der Optimierungsoption des C-Übersetzers *cc* noch gesteigert. Dadurch wird belegt, daß Optimierungen durch externe Dienste wirkungsvoll geleistet werden können. Abbildung 6.1 zeigt die Laufzeiten der genannten Varianten beim Durchlaufen von 400000 Listenelementen. Der Maschinenkode wird in der optimierten Version zwischen 17 und 52 mal schneller ausgeführt als TML und in der nicht optimierten zwischen 14 und 37 mal.

Um den Effizienzverlust durch Prozeduraufrufe abzuschätzen, sind die gleichen Testaufgaben jeweils einmal durch rekursive und iterative Programmierung gelöst worden. Die durch häufige Funktionsaufrufe bedingte relative Verlangsamung der rekursiven Version ist in TML fast unbedeutend. Dies ist vor allem auf die relative Ineffizienz der übrigen Operationen bei der Interpretation zurückzuführen. Außerdem wirkt sich die höhere Instruktionenanzahl der iterativen Lösung negativ aus.

	rekursiv	iterativ
TML	100.0	95.2
Quest-Bytekode	55.3	35.7
Maschinenkode ohne Opt.	6.9	2.5
Maschinenkode mit Opt.	5.8	1.8

Abbildung 6.1: Ein Laufzeitenvergleich zwischen interpretiertem Kode und Maschinenkode (Dauer des Traversierens von 400000 Listenelementen auf einer *Sun* 4 in Sekunden)

6.2 Die Bewertung des Quest-Systems als Implementationsbasis

Die in den vorangehenden Kapiteln beschriebenen Systemkomponenten sind zunächst in der Sprache Quest implementiert worden. TL ist eine Weiterentwicklung von Quest, die alle relevanten sprachlichen Konstrukte weiterhin unterstützt. Der Umfang der syntaktischen Anpassungen, die zur Übertragung von Quest-Programmen nach TL erforderlich sind, ist gering. Zudem kann die Umstellung mit Hilfe eines in Quest implementierten Parser-Generators[1] automatisiert werden. Diese „Aufwärtskompatibilität" von Quest reduziert die erforderlichen Zwischenschritte der Selbstimplementation des Tycoon-Systems auf ein Minimum. Insbesondere erlaubt sie von Beginn an die Verwendung von Vorversionen der Tycoon-Standardbibliotheken, die somit frühzeitig im Rahmen einer komplexen Aufgabenstellung getestet, verbessert und erweitert werden können. Unter den zahlreichen Modulen der Tycoon-Standardbibliotheken, die in der Implementation des Tycoon-Systems verwendet werden, befinden sich zum Beispiel:

fmt: Konvertierung von Werten der Basistypen (Ganzzahlen, Fließkommazahlen, Wahrheitswerte usw.) in formatierte Zeichenketten.

print: Bildschirmausgabe von Zeichenketten.

optional: konstante Optionalwerte. Sie werden unter anderem zur Realisierung oder besser „Simulation" optionaler Funktionsparameter verwendet, da Funktionen, die variable Argumentlisten akzeptieren, weder in Quest noch in TL unterstützt werden.

iter: Iterationsabstraktion. Generische Iteratoren vereinfachen vor allem die bei der Übersetzung von TL besonders häufige Abarbeitung sequentieller Datenstrukturen. Die vordefinierten Selektions- und Anfragemechanismen reduzieren den programmtechnischen Aufwand zur Lösung einiger Teilprobleme erheblich. Um einen Eindruck von der Effektivität dieser Technik zu vermitteln, sei als Beispiel der in Abschnitt 4.4.7 beschriebene Algorithmus zur Überprüfung rekursiver Bindungen genannt. Seine in Anhang B.1 aufgeführte Implementation führt die meisten ihrer relevanten Operationen mit Hilfe des Moduls *iter:Iter* durch. Die Nutzung einer weiteren starken Vereinfachung durch generische Iteratoren ist in Abschnitt 4.5.3 im Zusammenhang der Akkumulation von Funktionsliteralen angegeben: generische Iteratoren leisten auf kanonische Weise die Konvertierung zwischen beliebigen Tycoon-Massendatentypen (*bulk types*, z.B. List, Set, Queue).

[1] Der TL-Parser ist ein Produkt dieses Generators.

> Der Aufwand der Transformation ist dabei stets nur um einen (kleinen) konstanten Faktor höher als der theoretisch minimale.

Polymorphie wird im Rahmen der vorliegenden Arbeit vorwiegend unter Verwendung der beiden letztgenannten Module ausgenutzt. Auch in fast allen anderen Fällen beschränkt sich ihr Einsatz auf Tycoon-Standardbibliotheken, was nicht zuletzt darauf zurückzuführen ist, daß diese entsprechend mitgestaltet wurden. Damit ist gelungen, bereits in einer frühen Entwicklungsphase hochgradig wiederverwendbare Bestandteile des Tycoon-Systems zu schaffen.

Außer auf Polymorphie basieren die generischen Iteratoren auf Funktionen höherer Ordnung, die im übrigen als mittelbare Alternative zu Programmverzweigungen angewandt werden. Dies führt in der Regel zu substantiellen Programmvereinfachungen.

Die Kapselung lokaler Zustände durch geschachtelte Funktionen (vgl. Abschnitt 2.3.4) wird nur in eingeschränkter Form ausgenutzt. Dabei verlassen die Funktionsabschlüsse den Sichtbarkeitsbereich der jeweils umgebenden Funktion nicht. Sie dienen lediglich lokalen Vereinfachungen (siehe z.B. Anhang B.2). Geschachtelte Funktionen als Rückgabewerte von Funktionen kommen jedoch bei der Erstellung des Frontends voll zum Einsatz. Sie sind eine wesentliche Grundlage des ebenfalls in Quest implementierten Parsergenerators.

Module sind in Quest ebenfalls Werte „erster Klasse“. Auf diesem Umstand basiert die Realisierung der Zielsprachenunabhängigkeit der Maschinenkodegenerierung (vgl. Abschnitt 5.7). Die Kodegenerierung verschiedener Zielsprachen kann in Modulen gekapselt werden, die *zur Laufzeit* des Tycoon-Systems auswechselbar sind.

Quest bietet im Rahmen der gestellten Aufgaben adäquate Datenstrukturen. Besonders hervorzuheben sind die orthogonale Kombinierbarkeit der Konstrukte und die statische Typsicherheit. Für zahlreiche sehr spezielle Teilaufgaben der Zwischen- und der Maschinenkodegenerierung hat sich der in TL naheliegende Ansatz, Fallunterscheidungen durch variante Tupel auszudrücken, bewährt. Er wird zum Beispiel in den Repräsentationen der komplexen Zwischenrepräsentationen des Übersetzers verfolgt.

Die Symbiose von funktionalen und imperativen Konstrukten als algorithmischer Kern ist besonders ausdrucksstark und anpassungsfähig. Sie ist sowohl in der Zwischenkodegenerierung als auch in der Maschinenkodegenerierung von strategischer Bedeutung, da jeweils eine Hauptaufgabe darin besteht, rekursive Strukturen in sequentielle zu transformieren und flexibel anzuordnen. Das entsprechende Übersetzungsprinzip wird in Abschnitt 4.5.1 beschrieben und durch Abbildung bGenAltErg verdeutlicht.

Die vorangehende Betrachtung der sprachlichen Eigenschaften von Quest kann wie folgt zusammengefaßt werden:

> Die Programmiersprache Quest wird den Anforderungen der gestellten Aufgaben im hohen Maße gerecht.

Allerdings sind in der weiteren Entwicklung des Tycoon-Systems noch einige Verbesserungen möglich, wenn TL voll zum Einsatz kommt. Zum Beispiel hätte sich als zusätzliche Vereinfachung durch Polymorphie der gleichartige rekursive Abstieg durch Typ- und Wertbindungssequenzen in der Allokationsphase des Backends (siehe Abschnitt 4.4) angeboten. Die Voraussetzung dafür wäre jedoch die Verallgemeinerung der Subtypbeziehungen zwischen Typen zu

einer Subtypbeziehung zwischen Typoperatoren. Nach dem *bootstrap* kann eine diesbezügliche Verschmelzung der entsprechenden Funktionen erfolgen.

Im Rahmen der folgenden Betrachtung der systemtechnischen Bedingungen beim Einsatz des Quest-Systems wird deutlich, warum ihm gegenüber dem P-Quest-System der Vorzug zu geben ist, obwohl dieses bis auf die *zusätzliche* Eigenschaft der Persistenz identisch ist.

Die Entwicklung des Tycoon-Systems ist nur bezüglich der Speicherung seiner Komponenten auf Langlebigkeit angewiesen. Unter diesem Aspekt stehen die komplexen Aufgaben der Organisation der Quelltexte und Bibliotheken, sowie der Versionskontrolle und automatischen Rekonfiguration des Tycoon-Systems. Da das Quest-System diese nicht selbst unterstützt, werden einige externe Dienste als zusätzliche Entwicklungswerkzeuge (Dateisystem, *make*, Projektmanager) benötigt. Letztere liegen in fast allen Betriebssystemumgebungen (insbesondere auch unter *Sun OS*) in Form von eigenständigen Programmen vor und weisen keine zu ihrer direkten Einbindung geeigneten Schnittstellen auf. Quest unterstützt diese Dienste durch die Möglichkeit, beliebige Daten, insbesondere übersetzte Module, Schnittstellen und komplette Programme in einem speziellen Format in Dateien abzulegen und geschwind wieder einzulesen und zu binden.

Eine persistente Speicherung von Modulen, die an der Implementation des Tycoon-Systems beteiligt sind, läßt die externe Versionskontrolle (*make*) außer acht. Ihr Nutzen ist daher nur kurzfristig. Aufgrund der Größe des Tycoon-Systems führt jede Strategie der manuellen Versionskontrolle zu einer schwerwiegenden Verlangsamung der Entwicklungszyklen oder zu einer untragbaren Häufung von Fehlern.

Zusammenfassend wird festgestellt:

> Die Persistenz des P-Quest-Systems ist unter den gegebenen Bedingungen weder notwendig noch hilfreich. Als Voraussetzung für ihre effektive Nutzung müßten leistungsfähige Entwicklungswerkzeuge in das Quest-System integriert sein.

Als Konsequenz wird in der Entwicklung des Tycoon-Systems die nicht-persistente Quest-Version eingesetzt, deren Effizienz nicht durch Objektspeicherzugriffe gemindert wird. Dies ist auch unbedingt notwendig, denn der Byte-Kode-Interpreter beider Systeme ist langsam. Letzterer Umstand wird dadurch gemildert, daß die Module des Tycoon-Systems eine hinreichend geringe Größe erreichen, um einigermaßen akzeptable Entwicklungszyklen zu gewährleisten. Deren Anzahl wird jedoch durch wenig aussagekräftige und häufig irreführende Fehlermeldungen drastisch erhöht. Übersetzungsgeschwindigkeit und Fehlermeldungen stehen aus Entwicklersicht in einer Wechselbeziehung: Eine Verbesserung in einem der beiden Bereiche kompensiert auch Mängel des jeweils anderen.

Ein adäquates System zur Untersuchung des Laufzeitverhaltens von Quest existiert nicht. Ohne die hochentwickelte statische Programmüberprüfung wäre der Einsatz von Quest daher gänzlich ausgeschlossen. Der *post mortem*-Debugger des Quest-Systems bietet nur minimale Unterstützung. Zumindest werden jedoch die häufigsten Fehler, scheiternde Fallmarken und unbehandelte Ausnahmen nachvollziehbar präsentiert.

Insgesamt wird das Quest-System als die richtige Wahl zur Implementation von Tycoon und insbesondere der Kodegenerierung angesehen, da die Vorteile der ausgezeichneten sprachlichen Ausdrucksmittel die Nachteile der ausreichenden Entwicklungsumgebung bei weitem

überwiegen. Dies wird nicht zuletzt durch die praktische Durchführung der gestellten Aufgaben belegt. Vor allem sind die im Einsatz des Quest-Systems gewonnenen Erfahrungen aufgrund der sprachlichen und systemtechnischen Ähnlichkeiten im hohen Maße für die Entwicklung des Tycoon-Systems relevant.

6.3 Der Stand der Implementierung

Die in Kapitel 4 beschriebene Zwischenkodegenerierung ist vollständig implementiert und soweit bekannt ohne jede Einschränkung funktionsfähig. Mit Ausnahme der Unterstützung der Laufzeitanalyse und der Freispeicherverwaltung gilt dies auch die Maschinenkodegenerierung für die (Kapitel 5).

Beide Aufgaben basieren auf der Implementation der in Abschnitt 2.4 beschriebenen Enumerationsfunktion, die alle Zustandsvariablen eines Evaluationskontextes ermittelt. Um eine hardware-abhängige Implementation dieser Funktion zu unterstützen würde genügen, alle Temporärvariablen jeweils gleich nach ihrer Deklaration initialisieren. Es wird jedoch vorgezogen, im Anschluß an die vorliegende Arbeit zunächst alle sich bietenden Alternativen genau zu untersuchen, bevor eine diesbezügliche Festlegung des Kodes erfolgt.

Im diesem Zusammenhang steht auch die weitere Verbesserung der Portabilität des C-Kodes. Besonders attraktiv erscheint folgende Methode, bei der weder TML-Variablen noch Temporärvariablen direkt auf C-Variablen abgebildet werden. Statt dessen werden sie auf einem durch explizite C-Anweisungen manipulierten Programmstapel alloziert und indirekt adressiert. Dieser Stapel ist dann hardware-unabhängig zugreifbar, so daß eine *portable* Enumerationsfunktion zur Unterstützung der Laufzeitanalyse und der Freispeicherverwaltung realisierbar ist. Hervorzuheben ist, daß das bisherige abstrakte Übersetzungsschema (siehe Abschnitt 5.5.3) gegebenenfalls ohne Änderung weiterverwendet werden kann, weil die indirekte Adressierung aus seiner Sicht nur eine lexikalische Anpassung bedeutet. Der offensichtliche Nachteil der genannten Alternative ist ihre geringere Laufzeiteffizienz. Diese kann jedoch noch nicht genau quantifiziert werden und ist zudem stark hardware-abhängig.

Der Wechsel zwischen TML- und Maschinenkode ist bisher nur in einer Richtung möglich. Aufrufe von Maschinenkode durch den Interpreter funktionieren problemlos. Der Aufruf des Interpreters von Seiten des Maschinenkodes ist jedoch noch nicht fehlerfrei.

Das Tycoon-System wird gegenwärtig noch als „Satellit“ des Quest-Systems betrieben, d.h. die Tycoon-Benutzerumgebung und die TL-Übersetzung werden von letzterem ausgeführt. Es besitzt jedoch bereits einen eigenen Objektspeicher, in den alle Tycoon-Daten übertragen werden und auf dem der generierte Kode operiert. Die Schnittstelle zwischen beiden Systemen bildet das Modul *TMRTS*. Ihre wichtigsten Funktionen sind das Laden von TML-Kode in den Tycoon-Objektspeicher, das Auslösen von TML- und Maschinenkodeevaluationen und das Inspizieren von Tycoon-Speicherobjekten.

Der Implementationsstand der Kodegenerierung ist für die Selbstimplementierung (*bootstrap*) des Tycoon-Systems mehr als ausreichend. Dies gilt jedoch noch nicht für das Laufzeitssystem. Insbesondere ist die Verwaltung des generierten Maschinenkodes verbesserungswürdig. Für jede Übersetzungseinheit wird eine eigene *shared library* (siehe Abschnitt 5.5.5 erzeugt. Sowohl für die entsprechenden Dateien als auch für den im Einsatz beanspruchten Hauptspeicherplatz existiert noch keine systematische Freispeicherverwaltung.

In der nächsten Projektphase wird zunächst das Laufzeitsystems verbessert. Dann werden die wichtigsten Tycoon-Standardbibliotheken nach TL portiert und und der *bootstrap* des Tycoon-Systems in Angriff genommen. Nach dessen erfolgreicher Durchführung wird die Portierung auf weitere Hardware-Plattformen angestrebt. Für den Anfang sind IBM RS6000 (AIX), Digital Equipment VAX (VMS) und Apple Macintosh (Finder) vorgesehen.

Chronologisch unabhängig von obiger Vorgehensweise werden folgende Ziele angestrebt:

- Die Unterstützung der Laufzeitanalyse und der Freispeicherverwaltung durch den Maschinenkode.
- Die Erweiterung der generischen C-Programmierschnittstelle um die Möglichkeit, komplexe C-Datenstrukturen (Zeichenketten, *structs*, *unions*) zu transferieren.
- Zusätzliche externe Schnittstellen.
- Der Einsatz weiterer Zielsprachen.
- *inlining*-Techniken für TL-Funktionen, Laufzeitsystemfunktionen, externe Aufrufe und C-Anweisungen.
- Die wahlweise Verwendung schneller Hauptspeicheroperationen anstelle des TSP, d.h. die Möglichkeit zum gezielten Verzicht auf Persistenz zugunsten der Effizienz.
- Die wahlweise Erzeugung statisch gebundener und damit selbständiger Programme.

6.4 Resümee und Ausblick

Als wichtigste Grundlage der vorliegenden Arbeit hat sich die in [Matthes 92] postulierte konsequente Trennung zwischen den Aufgaben der Datenmodellierung (TL), Datenmanipulation (TML) und Datenspeicherung (TSP) herausgestellt. Besonders hervorzuheben ist in diesem Zusammenhang die durch Software-Schnittstellen erreichte Skalierbarkeit des Systems. Dies betrifft vor allem die Objektspeicherschnittstelle (Transparenz von Persistenz, Zugriffsgeschwindigkeit, nebenläufigem Zugriff, Speicherrückgewinnung, Fehlererholung und Verteilung). Darüber hinaus abstrahieren Funktionsaufrufe von jeglicher Speicherungsart, Lokalisation, Kodeart und dynamischen Transformation des Funktionskodes.

Die multiplen Programmrepräsentationen des Tycoon-Systems ergänzen sich aufgrund ihrer Ansiedelung auf verschiedenen Abstraktionsebenen besonders wirkungsvoll. So steht zum Beispiel der Effizienz des Maschinenkodes die Flexibilität und Portabilität von TML gegenüber. In Form von C-Quelltext steht eine zusätzliche Programmrepräsentation zur Verfügung, die besonders für den Einsatz in heterogenen Hochgeschwindigkeitsnetzwerken geeignet ist.

Portabilität ist eine vorrangige Anforderung bei der Gestaltung der Maschinenkodegenerierung, da sie die Einsatzfähigkeit in heterogenen Umgebungen entscheidend fördert. Dadurch trägt sie außerdem zur kontinuierlichen Unterstützung und zur Langlebigkeit des Tycoon-Systems bei. Die bereitgestellte Maschinenkodegenerierung ist aus folgenden Gründen besonders portabel:

- Die verwendete Zielsprache C ist standardisiert und ausgezeichnet verfügbar.

- ▷ Weitere Zielsprachen können mit geringem Aufwand hinzugefügt werden.
- ▷ Fast alle Systemabhängigkeiten werden durch Laufzeitsystemfunktionen gekapselt.

Die Einsatz einer portablen Zielsprache steht durchaus im Einklang mit den Anforderungen der Kodeoptimierung. Die weitaus meisten diesbezüglichen technischen Maßnahmen können an externe Dienste delegiert werden.

Die wichtigsten Beiträge der Zwischenkodegenerierung sind:

- ▷ Die Unterstützung des Subtyppolymorphismus von Tupeln.
- ▷ Ein einheitliches Adressierungsschema für veränderliche Werte.
- ▷ Die Übersetzung besonders komplexer rekursiver Wertbindungen.

Letztere erfassen in den meisten anderen Systemen lediglich Funktionen. Dabei können prinzipiell auch Aggregate und mit bestimmten Einschränkungen sogar Wertausdrücke in rekursiven Bindungen auftreten, ohne einen Zugriff auf einen uninitialisierten Wert zu bedingen. In der vorliegenden Arbeit wird sowohl ein Überprüfungsverfahren, das die häufigsten dieser Fälle als zulässig erkennt, als auch die entsprechende Methode zur Kodegenerierung bereitgestellt.

Als besonders adäquat haben sich folgende methodische Ansätze erwiesen:

- ▷ Die mehrphasige Übersetzung komplexer rekursiver Strukturen. Dadurch wird eine große Übersichtlichkeit der einzelnen Transformationsalgorithmen erreicht, nicht zuletzt, weil kein *backpatching* erforderlich ist.
- ▷ Die Symbiose von funktionalem und imperativem Programmierstil bei der Übersetzung rekursiver Strukturen, aus denen sequentielle Anordnungen als Ergebnis hervorgehen. Im speziellen bedeutet dies meist die Akkumulation von Teilergebnissen durch Schlangen bei rekursivem Abstieg.
- ▷ Die Verwendung von Modulen als Datenobjekte „erster Klasse“. Auf dieser Technik basiert die dynamische Auswechselbarkeit der Zielsprache der Maschinenkodegenerierung.

Der Einsatz von Netzwerken, insbesondere Hochgeschwindigkeitsnetzwerken ermöglicht prinzipiell eine sehr flexible Verteilung und Nutzung von Programmobjekten. Die sich bietenden Chancen werden jedoch bisher meist nur in Spezialanwendungen („*number crunching*“ o.ä.) ausgenutzt.

Ein besonders schwerwiegendes Hindernis für die flexible Gestaltung von Programmen ist das Binden von Programmobjekten, d.h. der Eintrag fester oder relativer Referenzen zu anderen Programmobjekten. Binden bedingt stets eine ungleiche Behandlung von Programmen und Daten. Außerdem ist der betroffene Maschinenkode nicht in anderen Adreßräumen verwendbar. In zahlreichen Systemen kommt hinzu, daß er auch in ein und demselben Adreßraum nicht frei verschiebbar ist. Eine völlige Überwindung dieser Beschränkungen ist nur möglich, wenn jedwedes Binden unterbleibt.

Im Tycoon-System müßten *alle* Funktionen über Funktionszeiger aufgerufen werden. Dies trifft bereits lediglich auf die Laufzeitsystemfunktionen nicht zu. Durch einen zusätzlichen kanonischen Parameter einer jeden Benutzerfunktion, der auf einen Vektor von Funktionszeigern des Laufzeitsystems verweist, könnten daher folgende Ziele erreicht werden:

- Innerhalb homogener Netzwerke könnte Maschinenkode als portable Programmrepräsentation verwendet werden.
- Der generierte Maschinenkode könnte in Speicherobjekten des Objektspeichers gehalten werden. Er unterläge damit insbesondere auch dessen Freispeicherverwaltung.
- Das Laufzeitsystems wäre dynamisch konfigurierbar.

Diese Erweiterungen lassen die Implementation von Programmen realistisch erscheinen, die aus zahlreichen verteilten Komponenten kleiner Granularität bestehen, welche jede bestehende Homogenität von Netzwerken zur Effizienzsteigerung durch direktes Versenden von Maschinenkode ausnutzen. Dies wäre dann die Eroberung einer weiteren Domäne rein interpreterbasierter Systeme.

Anhang A

Die Modulschnittstellen der Zwischenrepräsentationen

A.1 Der Syntaxbaum: TLValue

```
interface TLValue
(* System : Tycoon Compiler
   File   : TLValue.spec
   Author : Florian Matthes
            (Translated from Quest to TL by Bernd Mathiske)
   Date   : 02-AUG-1992
   Purpose: Abstract syntax trees for TL values
*)
import
  list
  optional
  :Source
  :TLIde
  :TLType
  :TLVariable
export

  (* Nested bindings, types and signatures are destructively normalized
     by tlCheckValue.value *)

  Let Rec T = Tuple
    pos :Source.Position
    case wrongCase    (* error already flagged, suppress further messages *)
    case okCase
    case intCase with (* literal values: *)
      val :Int
    end
    case realCase with
      val :Real
    end
```

```
case longrealCase with
  val :String
end
case charCase with
  val :Char
end
case stringCase with
  val :String
end
case ideCase with
  ideRef :TLIde.Ref
  var allocation :optional.T(TLVariable.Variable)
end
case funCase with
  var signatures :TLType.Signatures
  value :T
  var rangeType :optional.T(TLType.T)
  location :optional.T(T)
  var allocation :optional.T(TLVariable.Function)
end
case appCase with
  value :T
  var bindings :list.T(Binding)
end
case arrCase with
  var bindings :list.T(Binding)
  location :optional.T(T)
end
case indexCase with
  value :T
  index :T
end
case tupleCase with
  var bindings :list.T(Binding)
  location :optional.T(T)
end
case tupleCaseCase with
  var bindings :list.T(Binding)
  ideRef :TLIde.Ref
  (* .index = index in case label list, set during type checking *)
  var type :TLType.T
  location :optional.T(T)
end
case recordCase with
  var bindings :list.T(Binding)
  location :optional.T(T)
end
```

```
case fieldCase, projectCase, testCase with  (* v.l — v!l — v?l *)
  value :T
  ideRef :TLIde.Ref
  (* .index = index in field label list [ignoring non-dynamic type
     components],  set during type checking,
     tlIde.wrongIndex if record field access. *)
end
case extendCase with
  value :T
  var bindings :list.T(Binding)
end
case exceptionCase with
  value :T
  var signatures :TLType.Signatures
end
case seqCase with
  var bindings :list.T(Binding)
  location :optional.T(T)
end
case ifCase with
  condition :T
  var thenBindings :list.T(Binding)
  var elseBindings :list.T(Binding)
end
case caseCase, caseCrashCase, caseExhaustiveCase with
  value :T
  branches :list.T(CaseBranch)
  var elseBindings :list.T(Binding)
    (* empty for caseExhaustiveCase, caseCrashCase *)
  var allocation :optional.T(TLVariable.Environment)
end
case typecaseCase, typecaseCrashCase with
  var type :TLType.T
  branches :list.T(TypecaseBranch)
  var elseBindings :list.T(Binding) (* empty for typecaseCrashCase *)
end
case loopCase with
  var bindings :list.T(Binding)
end
case exitCase
case whileCase with
  condition :T
  var bindings :list.T(Binding)
end
case forCase with
  ide :TLIde.T
  value1, value2 :T
  var bindings :list.T(Binding)
  isUpTo :Bool
  var allocation :optional.T(TLVariable.Environment)
end
```

```
    case tryCase with
      var bindings :list.T(Binding)
      branches :list.T(TryBranch)
      var elseBindings :list.T(Binding)
      var allocation :optional.T(TLVariable.Environment)
    end
    case raiseCase with
      value :T
      var bindings :list.T(Binding)
    end
    case reraiseCase
    case assertCase with
      value :T
    end
    case andifCase, orifCase with
      value1, value2 :T
    end
  end

  and CaseBranch = Tuple
    pos :Source.Position
    labels :TLIde.RefList
    ide :TLIde.T          (* TLIde.wrong if no ide in branch *)
    var bindings :list.T(Binding)
  end

  and TypecaseBranch = Tuple
    pos :Source.Position
    var type :TLType.T
    var bindings :list.T(Binding)
  end

  and TryBranch = Tuple
    pos :Source.Position
    value :T
    ide :TLIde.T          (* TLIde.wrong if no ide in branch *)
    var bindings :list.T(Binding)
  end

(* – Type and Value Bindings: *)

(* Note that TLType.TypeBinding <:TLValue.Binding and
          TLType.TypeBindings <:TLValue.Bindings *)
```

```
  and Binding = Tuple
    (* – Type bindings: *)
    case typeCase with
      ide :TLIde.T
      type :TLType.T
    end
    case typeBoundCase with
      ide :TLIde.T
      type :TLType.T
      bound :TLType.T
    end
    case typeParallelCase with
      bindings :list.T(TLType.TypeBinding)
        (* typeCase — typeBoundCase *)
    end
    (* – Value bindings: *)
    case valueCase, valueTypeCase with
      ide :TLIde.T
      value :T
      mutable :Bool
        (* irrelevant after tlCheckValue.bindings since mutables
           are then flagged by a Var(type). *)
      type :TLType.T
        (* tlType.wrong if valueCase. Set to type inferred in
             tlCheckValue.binding. *)
      var isVarArgument :Bool
        (* set in tlCheckValue.value *)
      var allocation :optional.T(TLVariable.Environment)
        (* set during variable allocation *)
    end
    case valueParallelCase, valueRecCase with
      bindings :list.T(Binding)
        (* valueCase — valueTypeCase *)
    end
    case openCase with
      ideRef :TLIde.Ref
      bound :TLType.T
        (* wrongCase if not specified. *)
    end
  end

  Let CaseBranches = list.T(CaseBranch)

  Let TypecaseBranches = list.T(TypecaseBranch)

  Let TryBranches = list.T(TryBranch)

  Let Bindings = list.T(Binding)

  Let Location = optional.T(T)

end;
```

A.2 Der TML-Kode: TML

```
interface TML
(* System:  Tycoon machine
   File:    TML.spec
   Author:  Bernd Mathiske
   Date:    28-JUL-1992
   Purpose: Abstract Syntax of Tycoon Machine Code
*)
import
  :Source
export

   Let RuntimeCode = Tuple case                  (* Codes for the following RTS functions: *)
     machineBind,                                (* binding of an external function *)
     storeNew,                                   (* creation of a new store object *)
     exceptionRaisedValue,                       (* value of the last exception raised *)
     exceptionAssertValue,                       (* exception value for assertion failure *)
     exceptionCaseCrashValue,                    (* exception value for case-failure *)
     exceptiontypecaseCrashValue,                (* exception value for typecase-failure *)
     tupleProject,                               (* tuple projection *)
     tupleTest,                                  (* tuple variant test *)
     valueEqual,                                 (* identity comparison of 2 arbitrary values *)
     valueNotEqual,                              (* negated identity comparison of 2 arbitrary values *)
     intLess,                                    (* comparison of 2 integer numbers *)
     intGreater,                                 (* comparison of 2 integer numbers *)
     intSuccessor,                               (* incrementation of an integer number *)
     intPredecessor                              (* decrementation of an integer number *)
   end

                                                 (* TL features supported by the runtime codes: *)
   machineBindRuntimeCode,                       (* builtin.bind(Dyn F <:Ok  library, label :String):F *)
   storeNewRuntimeCode,                          (* var, tuple, array, ... *)
   exceptionRaisedValueRuntimeCode,              (* v in: try... else when... with v then... *)
   exceptionAssertValueRuntimeCode,              (* raise exception when ASSERT(...) fails *)
   exceptionCaseCrashValueRuntimeCode,           (* case without else: raise exception if no tag matches *)
   exceptiontypecaseCrashValueRuntimeCode,       (* typecase ohne else: raise exception if no tag matches *)
   tupleProjectRuntimeCode,                      (* t!a  *)
   tupleTestRuntimeCode,                         (* t?a  *)
   valueEqualRuntimeCode,                        (* x = y *)
   valueNotEqualRuntimeCode,                     (* x =/= y *)
   intLessRuntimeCode,                           (* exit conditions in for-loops *)
   intGreaterRuntimeCode,                        (* exit conditions in for-loops *)
   intSuccessorRuntimeCode,                      (* counters in for-loops *)
   intPredecessorRuntimeCode :RuntimeCode        (* counters in for-loops *)

   Let Rec Value = Tuple                         (* TML runtime values *)
     case nil
     case literals with value :Literals end
     case code with value :T end
     case polymorph with value :Int end
     case real with value :Real end
     case longReal with value :String end
```

```
  case string with value :String end
  case sourcePosition with pos :Source.Position end
end

and T = Tuple case                                   (* TML code *)
  case nopCode
  case immediateCode with
    value :Value
  end
  case literalCode with
    index :Int
  end
  case getLocalCode with
    index :Int                                       (* 1 <= index < nLocals *)
    mutable :Bool
  end
  case getParameterCode with
    index :Int                                       (* 1 <= index < nParameters *)
  end
  case getGlobalCode with
    index :Int                                       (* 1 <= index < nGlobals *)
  end
  case getIndexedCode with
    object :T
    index :T                                         (* 0 <= index < size(object) *)
    mutable :Bool
  end
  case setLocalCode with
    index :Int                                       (* 1 <= index < nLocals *)
    value :T
  end
  case setGlobalCode with
    closure :T
    index :Int                                       (* 1 <= index < nGlobals of closure *)
    value :T
  end
  case setIndexedCode with
    object :T
    index :T                                         (* 0<= index < size(object) *)
    value :T
  end
  case abstractCode with
    locality :T
    nGlobals :Int
    code :T
  end
  case lambdaExternalCode with
    nParameters :Int
    library :String
    label :String
  end
```

```
  case lambdaCode with
    nParameters :Int
    nLocals :Int
    code :T
  end
  case applyRuntimeCode with
    nArguments :Int
    arguments :T
    runtimeCode :RuntimeCode
  end
  case applyCode with
    nArguments :Int
    arguments :T
    function :T
  end
  case altCode with
    value :T
    tags :Tags
    branches :Branches
    elseBranch :T
    tagsExhaustive :Bool    (* value is guaranteed to be member of tags *)
  end
  case seqCode, argumentCode with
    first :T
    rest :T
  end
  case loopCode with
    code :T
  end
  case exitCode with
    value :T
  end
  case trapCode with
    code :T
    handler :T
  end
  case raiseCode with
    value :T
  end
end

and Tags = Array(Value)
and Branches = Array(T)

and Literals = Array(Value)

externalArchitecture :Value      (* Machine code tag *)
portableArchitecture :Value      (* TML code tag *)

Let Closure = Array(Value)
```

```
    closureLowLevelIndex :Int          (* 0 *)
    closureLiteralsIndex :Int          (* 1 *)
    closureGlobalsOffset :Int          (* 2 *)

    literalsCodeIndex :Int             (* 0 *)
    literalsArchitectureIndex :Int     (* 1 *)
    literalsLibraryIndex :Int          (* 2 *)
    literalsLabelIndex :Int            (* 3 *)
    literalsLiteralsOffset :Int        (* 4 *)

    bytesPerValue :Int                 (* 4 *)

    (* Value creation: *)
    newLiteralsValue(l :Literals) :Value
    newCodeValue(c :T) :Value
    nilValue, falseValue, trueValue :Value
    newIntValue(i :Int) :Value
    newRealValue(r :Real) :Value
    newLongRealValue(l :String) :Value
    newCharValue(c :Char) :Value
    newStringValue(s :String) :Value
    newSourcePositionValue(pos :Source.Position) :Value

    (* Code creation: *)
    nopCode :T
    newImmediateCode(value :Value) :T
    newLiteralCode(index :Int) :T
    newGetLocalCode(index :Int  mutable :Bool) :T
    newGetParameterCode(index :Int) :T
    newGetGlobalCode(index :Int) :T
    newGetIndexedCode(object, index :T  mutable :Bool) :T
    newArgumentCode(first, rest :T) :T
    newSetLocalCode(index :Int  value :T) :T
    newSetGlobalCode(closure :T  index :Int  value :T) :T
    newSetIndexedCode(object :T  index :T  value :T) :T
    newAbstractCode(locality :T  nGlobals :Int  code :T) :T
    newLambdaExternalLiterals(nParameters :Int  library, label :String) :Literals
    newLambdaCode(nParameters, nLocals :Int  code :T) :T
    newApplyRuntimeCode(runtimeCode :RuntimeCode  arguments :T) :T
    newApplyCode(function :T  arguments :T) :T
    newAltCode(value :T  tags :Tags  branches :Branches  elseBranch :T  exhaustive :Bool) :T
    newSeqCode(first, rest :T) :T
    newLoopCode(code :T) :T
    newExitCode(value :T) :T
    newTrapCode(code, handler :T) :T
    newRaiseCode(value :T) :T

end;
```

A.3 Die Repräsentation von Zielsprachenfragmenten: TM-Code

```
interface TMCode
(* System:  Tycoon machine
   File:    TMCode.spec
   Author:  Bernd Mathiske
   Date:    24-FEB-1992
   Purpose: Representation of target source code fragments
*)
import
  queue
  :TML
export

  Let Rec Unit = Tuple                (* Compilation unit *)
    entries :queue.T(Entry)
  end

  and Entry = Tuple                   (* Module or toplevel statement *)
    main :Function
    functions :queue.T(Function)
  end

  and Function = Tuple
    name :String
    nParameters, nLocals, nTemps :Int
    body :Statement
  end

  and Statement = Tuple
    case nopStatement
    case verbatimStatement with
      verbatim :String
    end
    case blockStatement with
      block :Block
    end
    case altStatement with
      value :String
      branches :Array(AltBranch)
      elseBranch :Statement
    end
    case loopStatement with
      label :Int
      statement :Statement
    end
    case trapStatement with
      code :Statement
      handler :Statement
    end
  end
```

```
  and Block = Tuple
    level :Int
    var nTemps :Int
    statements :queue.T(Statement)
  end

  and AltBranch = Tuple
    tags :queue.T(TML.Value)
    statement :Statement
  end


  newUnit() :Unit
  (* Create a new compilation unit with an initially empty queue of entries. *)

  newBlock(level :Int) :Block
  (* Create a new block at the specified nesting level,
     initialized with an empty statement queue and no local variables. *)

  newBlockTemp(b :Block) :Int
  (* Increment the number of local variables of the specified block and
     return the current index. *)


  (* The following functions just return tuples of their arguments. *)

  newEntry(main :Function  functions :queue.T(Function)) :Entry
  newFunction(name :String  nParameters, nLocals, nTemps :Int
              body :Statement) :Function

  nopStatement :Statement
  newVerbatimStatement(verbatim :String) :Statement
  newAltStatement(value :String  branches :Array(AltBranch)
                  elseBranch :Statement) :Statement
  newLoopStatement(loopLabel :Int  statement :Statement) :Statement
  newTrapStatement(code, handler :Statement) :Statement

end;
```

Anhang B

Spezielle Algorithmen

B.1 Die Überprüfung rekursiver Bindungen

```
interface TLCheckRec
(* System : Tycoon Compiler
   File   : TLCheckType.spec
   Author : Bernd Mathiske
   Purpose: Check whether recursively bound variables are correctly used *)
import
  :Iter
  :TLVariable
export

  Recursion <:Ok            (* Recursion contexts *)

  noRecursion :Recursion    (* The empty recursion context *)

  newVariables(recursion : Recursion
               variables :Iter.T(TLVariable.Variable)) :Recursion
  (* Create a new recursion context with an additional list
     of recursively bound 'variables'. *)

  enterFunction(recursion : Recursion) :Recursion
  (* Return a modified recursion context which takes into account that
     a function body has been entered. *)

  enterAggregate(recursion : Recursion) :Recursion
  (* Return a modified recursion context which takes into account that
     an aggregate has been entered. *)

  enterExpression(recursion : Recursion) :Recursion
  (* Return a modified recursion context which takes into account that
     an S-Expression has been entered. *)

  variableMisused(recursion :Recursion  variable :TLVariable.Variable) :Bool
  (* Check whether the occurrence of 'variable' is critical in this 'recursion' context. *)
end;
```

```
module tlCheckRec :TLCheckRec
(* System : Tycoon Compiler
   File   : tlCheckRec.impl
   Author : Bernd Mathiske
   Date   : 01-AUG-1992
   Purpose: Check whether recursively bound variables are correctly used
*)
import
  optional
  iter :Iter
  :TLVariable
export

  Let Variable = TLVariable.Variable

  (* A layer of variables that refers to a sequence of recursive bindings: *)
  Let Scope = Tuple
    variables :Iter.T(Variable)
    case inBinding, inAggregate, inExpression  (* context dependent modes *)
  end

  Let Recursion = Iter.T(Scope)  (* exported *)

  let noRecursion = iter.new(:Scope)

  let newVariables(recursion :Recursion
                   variables :Iter.T(Variable)) :Recursion =
    begin
      let scope = tuple case inBinding of Scope with variables end
      iter.cons(scope  recursion)
    end

  let rec enterFunction(recursion : Recursion) :Recursion =
    if iter.empty(recursion) then
      recursion
    else
      let head :Scope = iter.head(recursion)
      if head?inExpression then
        recursion
      else
        enterFunction(iter.rest(recursion))
      end
    end
```

```
  let enterAggregate(recursion : Recursion) :Recursion =
    if iter.empty(recursion) then
      recursion
    else
      let head :Scope = iter.head(recursion)
      if head?inBinding then
        let scope =
          tuple case inAggregate of Scope with
            head.variables
          end
        iter.cons(scope  iter.rest(recursion))
      else
        recursion
      end
    end

  let enterExpression(recursion : Recursion) :Recursion =
    if iter.empty(recursion) then
      recursion
    else
      let head :Scope = iter.head(recursion)
      if head?inExpression then
        recursion
      else
        let scope =
          tuple case inExpression of Scope with
            head.variables
          end
        iter.cons(scope  iter.rest(recursion))
      end
    end

  let rec variableMisused(recursion :Recursion  variable :Variable) :Bool =
    if iter.empty(recursion) then
      false
    else
      let head :Scope = iter.head(recursion)
      case head
        when inBinding then
          if iter.member(variable  head.variables) then
            true
          else
            variableMisused(iter.rest(recursion)  variable)
          end
        when inAggregate then
          variableMisused(iter.rest(recursion)  variable)
        when inExpression then
          iter.some(recursion  fun(scope :Scope) :Bool
            iter.member(variable  scope.variables))
      end
    end
end;
```

B.2 Die Konstruktion von TML-Sequenzen

```
Let Construct = Fun(first, rest :TML.T) :TML.T

Let Piece = Tuple
  code :TML.T
  case creation, use
end

let rec constructPieces(construct :Construct  pieces :queue.T(Piece)) :TML.T =
  if queue.empty(pieces) then
    tml.nopCode
  else
    let rec nextPiece() :TML.T =
      begin
        let piece :Piece = queue.pop(pieces)
        if piece?use orif queue.empty(pieces) then
          piece.code
        else
          tml.newSeqCode(piece.code  nextPiece())
        end
      end
    let rec getPieces() :TML.T =
      begin
        let piece = nextPiece()
        if queue.empty(pieces) then
          piece
        else
          construct(piece  getPieces())
        end
      end
    getPieces()
  end
```

Literaturverzeichnis

Abelson et al. 84: Abelson, H., Sussman, G.J., and Sussman, J. *Structure and Interpretation of Computer Programs.* The MIT Press, 1984.

Aho et al. 86: Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

Appel, J. 88: Appel, A. W. and J., Trevor. „Optimizing closure environment representations". Technical Report 168, Department of Computer Science, Princeton, 1988.

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

Apt 90: Apt, K.R. „Logic Programming". In: van Leeuwen, J., Hrsg., *Handbook of Theoretical Computer Science*, Band B, Seite 493–574. Elsevier Science Publishers, 1990.

Atkinson et al. 81: Atkinson, M.P., Chisholm, K.J., and Cockshott, W.P. „PS-algol: An Algol with a Persistent Heap". *ACM SIGPLAN Notices*, 17(7), Juli 1981.

Atkinson, Morrison 85: Atkinson, M.P. and Morrison, R. „First class persistent procedures". *ACM Transactions on Programming Languages and Systems*, 7(4), Oktober 1985.

Bause, Tölle 90: Bause, F. and Tölle, W., Hrsg. *C++ für Programmierer.* Vieweg, 1990.

Beeri, Kornatzky 90: Beeri, Catriel and Kornatzky, Yoram. „Algebraic Optimizations of Object-Oriented Query Languages". In: Abiteboul, S. and Kanellakis, P.C., Hrsg., *Third International Conference on Database Theory*, Band 470, *Lecture Notes in Computer Science*, Seite 72–88. Springer-Verlag, 1990.

Brown et al. 88: Brown, A.L., Connor, R.C.H., Carrick, R., Dearle, A., and Morrison, R. „The Persistent Abstract Machine". PPRR 59-88, Universities of Glasgow and St Andrews, März 1988.

Brown, Rosenberg 91: Brown, A.L. and Rosenberg, J. „Persistent Object Stores: An Implementation Technique". In: *Proceedings of the Fourth International Workshop on Persistent Object Systems, Martha's Vineyard, Massachusetts.* Morgan Kaufmann Publishers, Januar 1991.

Cardelli et al. 91: Cardelli, L., Martini, S., Mitchell, J.C., and Scedrov, A. „An Extension of System F with Subtyping". In: Ito, T. and Meyer, A.R., Hrsg., *Theoretical Aspects of Computer Software, TACS'91*, Lecture Notes in Computer Science, Seite 750–770. Springer-Verlag, 1991.

Cardelli 83: Cardelli, L. „The functional abstract machine". *Polymorphism*, 1(1), 1983.

Cardelli 84: Cardelli, L. „A Semantics of Multiple Inheritance". In: Kahn, G., MacQueen, D.B., and Plotkin, G., Hrsg., *Semantics of Data Types*, Band 173, *Lecture Notes in Computer Science*, Seite 51–67. Springer-Verlag, 1984.

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

Cardelli 86b: Cardelli, L. „The Amber Machine". In: *Combinators and Functional Programming Languages*, Band 242, *Lecture Notes in Computer Science*. Springer-Verlag, 1986.

Cardelli 89: Cardelli, L. „Typeful Programming". Digital Systems Research Center Reports 45, DEC SRC Palo Alto, May 1989.

Cardelli 90: Cardelli, L. „The Quest Language and System (Tracking Draft)". Digital Systems Research Center, DEC SRC Palo Alto, 1990. (shipped as part of the Quest V.12 system distribution).

Dearle et al. 89: Dearle, A., Connor, R., Brown, F., and Morrison, R. „Napier88 – A Database Programming Language?". In: *Proceedings of the Second International Workshop on Database Programming Languages, Salishan, Oregon*, Juni 1989.

Dearle, Brown 87: Dearle, A. and Brown, A.L. „Safe Browsing in a Strongly Typed Persistent Environment". PPRR 33-87, Universities of Glasgow and St Andrews, April 1987.

Field, Harrison 88: Field, A.J. and Harrison, P.G. *Functional Programming*. Addison-Wesley, Workingham, England, 1988.

Gawecki 91: Gawecki, A. „Ein optimierender Übersetzer für Smalltalk". Technical Report 152, hbg-info, September 1991.

Hall, Barry 90: Hall, M. and Barry, J., Hrsg. *The Sun Technology Papers*. Springer-Verlag, 1990.

Hudak 89: Hudak, P. „Conception, Evolution, and Application of Functional Programming Languages". *ACM Computing Surveys*, 21(3):359–411, September 1989.

Ichbiah, others 83: Ichbiah et al. „The Programming Language Ada: Reference Manual". Technical Report MIL-STD-1815A-1983, ANSI, 1983.

IEEE 85: IEEE Computer Society, Los Alamitos, CA. *ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*, 1985.

Jähnichen et al. 78: Jähnichen, S., C., Oeters, and B., Willis. *Übersetzerbau*. Vieweg, 1978.

Kernighan, Ritchie 77: Kernighan, B.W. and Ritchie, D.M. *The C Programming Language*. Prentice-Hall, 1977.

Kernighan, Ritchie 88: Kernighan, B.W. and Ritchie, D.M. *The C Programming Language, 2nd Edition*. Prentice-Hall, 1988.

King 89: King, K.N. „The International Standardization of Modula-2". In: *Computing Trends in the 90's*. ACM Press, Februar 1989.

Kranz et al. 86: Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. „ORBIT: An Optimizing Compiler for Scheme". *ACM SIGPLAN Notices*, 21(7):219–233, Juli 1986.

Matthes, Schmidt 89: Matthes, F. and Schmidt, J.W. „The Type System of DBPL". In: *Proceedings of the Second International Workshop on Database Programming Languages, Salishan, Oregon*, Seite 255–260, Juni 1989.

Matthes, Schmidt 91a: Matthes, F. and Schmidt, J.W. „Bulk Types: Built-In or Add-On?". In: *Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Greece*. Morgan Kaufmann Publishers, September 1991. (also appeared as TR FIDE/91/27).

Matthes, Schmidt 91b: Matthes, F. and Schmidt, J.W. „Towards Database Application Systems: Types, Kinds and Other Open Invitations". In: *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, Band 504, *Lecture Notes in Computer Science*, April 1991.

Matthes 91: Matthes, F. „P-Quest: Installation and User Manual". DBIS Tycoon Report 101-91, Fachbereich Informatik, Universität Hamburg, Germany, Oktober 1991.

Matthes 92: Matthes, F. *Generische Datenbankprogrammierung: Sprachliche und architektonische Grundlagen.* PhD thesis, Fachbereich Informatik, Universität Hamburg, Germany, September 1992.

Mauny 91: Mauny, M. „Functional Programming using CAML". Technical report, INRIA, Domaine de Voluceau Rocquencourt 78153 Le Chesnay Cedex - France, September 1991.

Meyer 86: Meyer, B. „Genericity versus Inheritance". In: *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, Portland, Oregon*, Seite 391–405, Oktober 1986.

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

Milner 78: Milner, R. „A Theory of Type Polymorphism in Programming". *Journal of Computer and System Sciences*, 17:348–375, 1978.

Minker 88: Minker, J. *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann Publishers, 1988.

ModISO 91: ISO/IEC JTC1/SC22/WG13. *Interim Version of the 4th Working Draft Modula-2 Standard*, 1991.

Morrison et al. 87: Morrison, R., Brown, A. L., Carrick, R., Connor, R.C.H., Dearle, A., and Atkinson, M.P. „Polymorphism, Persistence and Software Reuse in a Strongly Typed Object-Oriented Environment". *Software Engineering Journal*, Seite 199–204, Dezember 1987.

Moss 89: Moss, J.E.B. „Addressing Large Distributed Collections of Persistent Objects: The Mneme Project's Approach". In: *Proc. of the 2nd Workshop on Database Programming Languages, Portland, Oregon*, Seite 358–374, Juni 1989.

Muchnick 90: Muchnick, S.S. „Optimizing Compilers for the SPARC Architecture". In: Hall, M. and Barry, J., Hrsg., *The Sun Technology Papers*. Springer-Verlag, 1990.

Müller 91: Müller, Rainer. „Sprachprozessoren und Objektspeicher: Schnittstellenentwurf und -implementierung". Master's thesis, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, Germany, November 1991.

Nelson 91: Nelson, G., Hrsg. *Systems programming with Modula-3.* Prentice Hall series in innovative technology, 1991.

Niederée 92: Niederée, C. „Generische Dienste für datenintensive Anwendungen: Iterationsabstraktion, Integritätsüberwachung, Fehlererholung". Master's thesis, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

Peyton Jones 87: Peyton Jones, S. L. *The Implementation of Functional Programming Languages.* Prentice-Hall International, Englewood Cliffs, N.J., 1987.

Plotkin 81: Plotkin, G.D. „A structural appraoch to operational semantics". DIAMI FN 19, Computer Science Department, Aarhus University, 1981.

Rovner et al. 85: Rovner, P., Levin, R., and Wick, J. „On Extending Modula-2 for Building Large, Integrated Systems". Digital Systems Research Center Reports 3, DEC SRC Palo Alto, Januar 1985.

RSRE 91: RSRE. „TDF Specification". Technical report, Defense Research Agency, RSRE, St. Andrews Road, Malvern, Worcestershire WR 14 3PS, UK, Oktober 1991. (2 parts).

Schäfer, Bomarius 87: Schäfer, R. and Bomarius, F. „Modula2C: Ein Übersetzer von Modula-2 nach C". Master's thesis, Kaiserslautern, Germany, Dezember 1987.

Schmidt, Matthes 92: Schmidt, J.W. and Matthes, F. „The Database Programming Language DBPL: User and System Manual". FIDE Technical Report FIDE/92/47, Fachbereich Informatik, Universität Hamburg, Germany, Juli 1992.

Schmidt 77: Schmidt, J.W. „Some High Level Language Constructs for Data of Type Relation". In: *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Toronto, Canada*, August 1977.

Schröder 91: Schröder, Gerald. „Studienarbeit: Die Standardisierung von Modula-2". Master's thesis, Fachbereich Informatik, Universität Hamburg, Germany, November 1991.

Steel 84: Steel, G. L. Jr. *Common Lisp: The Language.* Digital Press, 1984.

Steele 86: Steele, G.L. Jr. „The Revised^3 Report on the Algorithmic Language Scheme". *ACM SIGPLAN Notices*, 21(12):37–79, Dezember 1986.

Tarditi et al. 91: Tarditi, D., Acharya, A., and Lee, P. „No Assembly Required: Compiling Standard ML to C". Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, März 1991.

TDF 91: „Some Examples of Mapping ANSI C to TDF“. Technical report, Defense Research Agency, RSRE, St. Andrews Road, Malvern, Worcestershire WR 14 3PS, UK, Oktober 1991.

Velez et al. 89: Velez, F., Bernard, G., and Darnis, V. „The O_2 Object Manager: an Overview“. Rapport Technique 27-89, GIP Altair, Domaine de Voluceau Rocquencourt 78153 Le Chesnay Cedex - France, Februar 1989.

Waite, Goos 85: Waite, W.M. and Goos, G. *Compiler Construction.* Texts and monographs in computer science. Springer-Verlag, 1985.

Wirth 87: Wirth, N. „The Programming Language Oberon“. Technical report, Department Informatik, ETH Zürich, Switzerland, 1987.