Integration eines Modul– und Klassen–Konzeptes in die funktionale Programmiersprache SAC — Single Assignment C

Diplomarbeit
von
Clemens Grelck

am Institut für Informatik und praktische Mathematik
der Christian–Albrechts–Universität zu Kiel

Kiel
30. April 1996

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Ein wichtiges Teilgebiet für den Einsatz von Computern besteht in der Implementierung numerischer Algorithmen. Diese bilden die Grundlage vieler technisch-naturwissenschaftlicher Anwendungsprogramme. Typische Merkmale dafür sind extrem rechen-intensive Operationen sowie die Verarbeitung sehr großer Datenmengen, die in Form von Matritzen oder vergleichbaren Strukturen organisiert sind. Daraus leiten sich besondere Anforderungen an eine geeignete Programmiersprache ab.

Der Programmierer muß bei der Implementierung von Algorithmen auf Matrix-förmigen Datenstrukturen optimal unterstützt werden. Dazu ist ein Array-Konzept erforderlich, welches auf der einen Seite ein hohes Maß an Flexibilität gewährt, auf der anderen Seite jedoch eine einfache und prägnante Formulierung auch komplexer Algorithmen zuläßt. Maßstäbe für Flexibilität sind die Verfügbarkeit von Arrays beliebiger Dimension und Struktur oder die Möglichkeit, Algorithmen unabhängig von der konkreten Dimension und Form eines zugrundeliegenden Arrays spezifizieren zu können. Der Umfang und die Komplexität integrierter primitiver Array-Operationen bestimmen die Ausdrucksfähigkeit, der Grad der Abstraktion von administrativen Problemen, wie z.B. der Speicherverwaltung, die Einfachheit in der Anwendung.

Die Implementierung komplexer Algorithmen erfordert eine Programmiersprache, die auch die Methoden der strukturierten Programmierung unterstützt. Für die Entwicklung umfangreicher Anwendungsprogramme sind zusätzlich Möglichkeiten zur modularen Programm-Entwicklung notwendig. Schwerpunkte liegen dabei in der Wiederverwendbarkeit von Code durch Bildung von Programm-Bibliotheken sowie der separaten Entwicklung und Wartung einzelner Programm-Komponenten.

Eine weitere Anforderung besteht in der Verfügbarkeit von Ein-/Ausgabe-Operationen. Sie bilden die Voraussetzung für die Erstellung von Anwendungsprogrammen, die selbständig Eingabedaten für ein numerisches Verfahren erfassen, Ergebnisse präsentieren oder mit einem Benutzer interagieren. Darüberhinaus stellen Ein-/Ausgabe-Operationen ein wichtiges Hilfsmittel zur Fehlersuche während der Programm-Entwicklung dar.

Neben dem Sprachdesign spielt das Laufzeitverhalten von Programmen eine entscheidende Rolle. Ausführungszeiten und Speicherbedarf sind gerade bei rechen-intensiven Anwendungen auf großen Datenstrukturen besonders kritische Faktoren für die Eignung einer Programmiersprache. Häufig stellt eine effiziente Abbildung auf die zur Verfügung stehende Hardware das entscheidende Kriterium für die Auswahl einer Programmiersprache dar. Dies schließt insbesondere die nebenläufige Ausführung von Programmen auf Multi-Prozessor-Architekturen ein. Daraus resultieren wiederum zusätzliche spezifische Anforderungen an eine Programmiersprache. Diese sollte so weit wie möglich

von der verwendeten Hardware abstrahieren. Dadurch wird die Portabilität von Programmen erhöht, und die Programmierarbeit konzentriert sich verstärkt auf den algorithmischen Teil eines Problems. Darüberhinaus sollte eine Sprache auch bei nebenläufiger Programm-Ausführung deterministische Resultate garantieren.

Zur Erstellung technisch-naturwissenschaftlicher Anwendungen finden zahlreiche unterschiedliche Programmiersprachen Verwendung. Unter besonderer Berücksichtigung von Arrays wurde Anfang der 60er Jahre die Sprache APL [Ive62] entwickelt. Sie stellt dem Programmierer mehrdimensionale Arrays und mächtige Operationen auf diesen zur Verfügung. Mit ihrer Hilfe lassen sich auch komplexe Algorithmen durch verhältnismäßig kurze Programme implementieren. APL ist jedoch primär für interaktive Systeme ausgelegt. Fehlende Möglichkeiten zur Strukturierung und Modularisierung erschweren daher die Entwicklung größerer Anwendungsprogramme. Die interpretative Verarbeitung von APL-Programmen führt darüberhinaus zu hohen Rechenzeiten.

Ein weitaus besseres Laufzeitverhalten erreichen die imperativen Programmiersprachen, bei denen Programme vor ihrer Ausführung in die jeweilige Maschinensprache übersetzt werden. Den imperativen Sprachen liegt ein Berechnungsmodell zu Grunde, das auf der sequentiellen Transformation eines globalen Zustands basiert. Imperative Programme lassen sich daher sehr effizient auf klassische Von-Neumann-Architekturen abbilden, denen das Modell endlicher Automaten zugrundeliegt. Ende der 50er Jahre wurde speziell im Hinblick auf technisch-naturwissenschaftliche Anwendungen die immer noch weit verbreitete Programmiersprache FORTRAN [Weh85] entwickelt. Weitere gebräuchliche Vertreter imperativer Sprachen sind PASCAL [JW74], MODULA-2 [Wir85] oder C [KR88]. Alle genannten Sprachen bieten mehrdimensionale Arrays als strukturierte Datentypen. Primitive Operationen auf Arrays beschränken sich jedoch im wesentlichen auf deren Konstruktion und die Selektion einzelner Elemente. Auch die Speicherverwaltung bleibt bei diesen Maschinen-orientierten Sprachen die Aufgabe des Programmierers. Über ein integriertes Modul-Konzept verfügt lediglich MODULA-2. Dagegen erlauben alle genannten Sprachen eine einfache und intuitive Spezifikation von Ein-/Ausgabe-Operationen.

Die Erzeugung von nebenläufig ausführbarem Code bereitet bei den imperativen Sprachen jedoch i.a. große Schwierigkeiten. Zu diesem Zweck ist es erforderlich, innerhalb des globalen Zustands disjunkte Teilzustände zu identifizieren. Dies wird jedoch durch Sprachkonstrukte wie globale Variablen, Sprünge oder die COMMON-Blöcke in FORTRAN sehr erschwert. So hängt das Gesamtergebnis einer Berechnung grundsätzlich auch von der Berechnungsreihenfolge syntaktisch unabhängiger Teilausdrücke ab. Um dennoch die Entwicklung nebenläufiger Programme zu ermöglichen, sind zusätzliche Sprachkonstrukte zur Deklaration nebenläufig auszuführender bzw. nebenläufig ausführbarer Programmteile und zum Datenaustausch zwischen diesen erforderlich. Diesem Zweck dienen Funktionsbibliotheken, wie z.B. PVM [G+93] für C oder FORTRAN. Sie stellen explizite Operationen zur Gründung und Terminierung von Prozessen sowie zur Kommunikation zwischen Prozessen bereit. Einen deutlich höheren Abstraktionsgrad erreichen spezifische Weiterentwicklungen imperativer Sprachen wie HPF [For94] oder C* [Fra91]. Doch auch sie sind in entscheidender Weise von entsprechenden Angaben des Programmierers abhängig. Dadurch wird dieser zusätzlich belastet, deterministische Resultate können nicht garantiert werden und die Portabilität ist i.a. stark eingeschränkt.

Grundsätzlich besser geeignet zur Entwicklung nebenläufig ausführbarer Programme sind funktionale Programmiersprachen. Sie beruhen auf dem in den 30er Jahren von A. Church entwickelten λ-Kalkül [Chu32, Bar81, Ros84] bzw. auf Kombinatorkalkülen. Dadurch erfüllen sie die Church-Rosser-Eigenschaft, d.h. die Bedeutung eines funktionalen Programms ist grundsätzlich unabhängig von der Reihenfolge, in der seine Teilausdrücke berechnet werden. Daraus resultiert eine Freiheit bei der Wahl der konkreten Berechnungsreihenfolge für syntaktisch unabhängige Teilausdrücke, welche die Möglichkeit der nebenläufigen Berechnung eröffnet.

Einige gebräuchliche funktionale Sprachen, wie z.B. MIRANDA [Tur85] oder ML [QRM+87] verzichten jedoch vollständig auf die Integration von Arrays. Andere, wie KiR [Klu94] oder NIAL [JJ93], kennen Arrays als strukturierte Datentypen und stellen APL-ähnliche Operationen auf diesen zur Verfügung. Kennzeichen dieser Operationen ist die Tatsache, daß alle Elemente eines Arrays auf die gleiche Weise davon betroffen sind. Dies macht die explizite Angabe von Array-Grenzen überflüssig, wodurch die Grundlage für die dimensionsunabhängige Spezifikation von Programmen gelegt wird. Sollen jedoch lediglich Teile eines Arrays von einer Operation betroffen sein, so läßt sich dies ausschließlich durch vorherige Dekomposition und nachfolgende Rekonstruktion des betroffenen Arrays erreichen. Eine differenzierte Behandlung von Arrays ermöglichen dagegen die **Array Comprehensions** von HASKELL [H+95] oder die **For-Loops** in SISAL [Can93, Feo92]. Sie erfordern jedoch umgekehrt die explizite Angabe von Array-Grenzen, was die dimensionsunabhängige Formulierung von Programmen ausschließt. Im Gegensatz zu imperativen Sprachen unterscheiden funktionale Sprachen in der Handhabung nicht zwischen einfachen und strukturierten Datenobjekten. Von der konkreten Darstellung eines Arrays im Speicher eines Rechners wird vollständig abstrahiert, wodurch der Umgang mit Arrays für den Programmierer entscheidend vereinfacht wird.

Das höhere Abstraktionsniveau funktionaler Sprachen wird i.a. mit einem gegenüber imperativen Sprachen sowohl in Fragen der Rechenzeit als auch des Speicherbedarfs nachteiligen Laufzeitverhalten erkauft. Das Beispiel der Sprache SISAL zeigt jedoch, daß sich diese Probleme vermeiden lassen, wenn zugunsten des Laufzeitverhaltens auf eine vollständige Implementierung des funktionalen Paradigmas verzichtet wird. Funktionen höherer Ordnung, partielle Funktionsanwendungen, ein polymorphes Typsystem oder Lazy Evaluation wirken sich i.a. negativ auf das Laufzeitverhalten von Programmen aus. Laufzeitvergleiche in Multiprozessor-Umgebungen, wo die konzeptuellen Vorteile des funktionalen Paradigmas zum Tragen kommen, ergeben für SISAL-Programme Vorteile um den Faktor 2 und mehr gegenüber äquivalenten FORTRAN-Programmen [Can92, OCA86].

Trotz konzeptueller Vorteile kommen funktionale Programmiersprachen bei der Entwicklung realer Anwendungsprogramme auch im technisch-naturwissenschaftlichen Bereich nur selten zur Anwendung. Die Gründe dafür sind vielfältig. Bei sequentieller Ausführung sind funktionale Programme imperativen gewöhnlich deutlich unterlegen. Das liegt u.a. daran, daß die meisten funktionalen Sprachen vollständige Implementierungen eines funktionalen Kalküls darstellen. Die damit verbundene Steigerung der Ausdrucksfähigkeit wird jedoch in vielen Fällen geringer bewertet als die Laufzeitnachteile.

Kommerzielle Unternehmen scheuen die mit einer Umstellung von einer imperativen auf eine funktionale Programmiersprache verbundenen Kosten. Das andere Berechnungsmodell verlangt vom Programmierer ein weitgehendes Umdenken bei der Formulierung von Algorithmen, und auch die Syntax gebräuchlicher funktionaler Sprachen unterscheidet sich deutlich von der imperativer Sprachen. Wird Software-Entwicklung über einen längeren Zeitraum betrieben, entstehen gewöhnlich umfangreiche Funktionsbibliotheken, auf die bei Bedarf zurückgegriffen werden kann. Dies spart Entwicklungszeit und -kosten. Bei einem Wechsel der Programmiersprache werden diese Bibliotheken wertlos und müssen re-implementiert werden.

Daneben stellt die konzeptuell bedingte Schwierigkeit funktionaler Sprachen im Umgang mit Ein-/Ausgabe ein entscheidendes Problem dar. Das funktionale Paradigma kennt lediglich die Berechnung von Werten[1], nicht jedoch die Speicherung von Zuständen. Der geordnete Umgang mit Ein-/Ausgabe-Geräten setzt jedoch eine geeignete Repräsentation von deren jeweiligen Zuständen zwingend voraus. Manche funktionale Programmiersprachen, wie z.B. SISAL oder KiR, verzichten daher vollständig auf explizite Ein-/Ausgabe-Mechanismen. HASKELL und CLEAN [BvEvLP87, PvE95] dagegen bilden mit unterschiedlichen Mitteln einen Zustand auf eine Weise nach, die nicht mit dem

[1] Der Begriff Wert ist in diesem Zusammenhang sehr allgemein zu verstehen. Neben einfachen Datenobjekten wie Zahlen oder Zeichen umfaßt er strukturierte Datenobjekte wie Arrays oder Listen ebenso wie beispielsweise Funktionen.

funktionalen Paradigma konfligiert. Die Flexibilität und der intuitive Charakter der Ein-/Ausgabe-Operationen imperativer Sprachen werden jedoch bei weitem nicht erreicht.

Keine der genannten Programmiersprachen erfüllt in hinreichender Weise die bei der Implementierung technisch-naturwissenschaftlicher Anwendungen an sie gestellten Anforderungen. Aus diesem Grund wurde von S.-B. Scholz 1994 die neue Sprache SAC [Sch94] vorgeschlagen. Der Name SAC steht für **Single Assignment C**. Dadurch soll zum Ausdruck gebracht werden, daß SAC einerseits eine rein funktionale Sprache ist, sich andererseits jedoch syntaktisch eng an die imperative Sprache C anlehnt. Ziel der Entwicklung ist eine Sprache, welche die konzeptuellen Vorteile des funktionalen Paradigmas nutzt, dabei jedoch die Akzeptanzprobleme bestehender funktionaler Sprachen soweit wie möglich vermeidet.

Zu diesem Zweck wird, ausgehend von C, eine Teilmenge an Sprachkonstrukten identifiziert, deren Bedeutung funktional erklärt werden kann. Dazu gehören neben Funktionen auch Mehrfachzuweisungen[2], alle aus C bekannten Schleifen sowie die bedingte Verzweigung. Nicht übernommen werden im wesentlichen globale Variablen, Zeiger und Sprünge. Dieser Kern von SAC erlaubt es, rein funktionale Programme zu schreiben, die sich syntaktisch nicht von äquivalenten C-Programmen unterscheiden.

Ergänzt wird der SAC-Kern um ein auf dem ψ-Kalkül von L. Mullin [Mul88, MT94] basierendes Array-Konzept. Dieses beinhaltet mehrdimensionale Arrays sowie APL-ähnliche primitive Operationen, die auch die dimensionsunabhängige Spezifikation von Algorithmen erlauben. Zusätzlich besitzt SAC zur selektiven Behandlung von Arrays in Form der sog. **With-Loops** ein Konstrukt, das den Array Comprehensions aus HASKELL oder den For-Loops in SISAL ähnlich ist. Wie bei funktionalen Sprachen üblich werden Arrays in SAC als einfache Werte angesehen. Sie dürfen daher sowohl Argumente als auch Rückgabewerte einer Funktion sein. Um Fragen der Speicherverwaltung wie bei C muß sich der SAC-Programmierer nicht kümmern.

Im Rahmen der Diplomarbeiten von H. Wolf [Wol95] und A. Sievers [Sie95] ist ein Compiler entwickelt worden, der SAC-Programme in (sequentielle) C-Programme übersetzt. Besonderes Gewicht wurde dabei auf die Optimierung des Laufzeitverhaltens gelegt. Diesem Zweck dienen umfangreiche Code-Optimierungen, die wesentlich auf der referentiellen Transparenz funktionaler Sprachen aufbauen. Durch sie werden Laufzeiten erreicht, die denen äquivalenter SISAL-Programmen ebenbürtig, häufig sogar überlegen sind.

In der vorliegenden Form erfüllt SAC jedoch nicht alle der eingangs gestellten Anforderungen. Es fehlen u.a. Möglichkeiten zur modularen Programmierung, die Integration von Ein-/Ausgabe-Mechanismen und eine Antwort auf die oben geschilderte Problematik der Umstellung auf eine funktionale Programmiersprache. Diese drei Punkte sind Gegenstand der vorliegenden Arbeit.

Die Sprache SAC wird um ein Modul-Konzept erweitert, das in wesentlichen Punkten denen von MODULA-2, HASKELL oder CLEAN folgt. Dazu gehören getrennte Deklarations- und Implementierungsteile, abstrakte Datentypen und die Möglichkeit der separaten Kompilation einzelner Module. Bei der Implementierung des Modul-Konzeptes wird ein Schwerpunkt darauf gelegt, daß trotz separater Kompilation das Laufzeitverhalten eines modular aufgebauten Programms dem eines äquivalenten nicht-modularen Programms gleichwertig ist.

Die Integration von Ein-/Ausgabe-Mechanismen basiert konzeptuell auf dem für CLEAN entwickelten **Uniqueness-Typing-Konzept** [SBvEP93]. Dieses wird jedoch gegenüber dem Programmierer weitgehend versteckt. Das Uniqueness-Typing-Konzept orientiert sich streng an den spezifischen Anforderungen des funktionalen Paradigmas in Bezug auf Ein-/Ausgabe bzw. Zustände im allgemeinen. Seine korrekte Anwendung setzt daher beim Programmier ein tiefgreifendes Verständnis

[2]Der scheinbare Widerspruch zum Namen der Sprache wird in Abschnitt 2.2 erklärt.

dieser Zusammenhänge voraus. Derartige Kenntnisse sind jedoch bei an imperative Sprachen gewöhnten Programmierern i.a. nicht vorhanden. Dieses Defizit gleicht SAC durch sein Klassen-Konzept aus. Bei einer Klasse handelt es sich um eine besondere Art von Modul, das einen ausgezeichneten abstrakten Datentyp bereitstellt. Ein Ausdruck eines solchen Typs wird als Objekt bezeichnet und repräsentiert per Definition einen Zustand. Solche Objekte können ausschließlich durch entsprechende Funktionen der Klasse erzeugt werden. Darüberhinaus kann die Klasse Funktionen zur Modifikation und zum Löschen eines Objektes enthalten. Eine speziell angepaßte Syntax erlaubt die Spezifikation von Ein-/Ausgabe-Operationen in einer Weise, die der aus C und anderen imperativen Sprachen bekannten Technik sehr ähnlich ist. Bei Benutzung entsprechender Standard-Klassen ist eine Auseinandersetzung mit den Problemen funktionaler Ein-/Ausgabe oder dem Uniqueness-Typing-Konzept nicht erforderlich. Die Abbildung des Klassen-Konzeptes auf das zugrundeliegende Uniqueness-Typing-Konzept erfolgt durch den Compiler. Diese konzeptuelle Grundlage garantiert den Erhalt von Church-Rosser-Eigenschaft und referentieller Transparenz soweit wie vom Compiler zur Durchführung von Optimierungen oder zur Identifikation nebenläufig ausführbarer Programmteile benötigt.

Um die Umstellung von einer imperativen Sprache zu erleichtern, lehnt sich SAC syntaktisch eng an die imperative Sprache C an. Dies kann jedoch nur einen Teil der mit einer Umstellung verbundenen Probleme vermindern. Es ist darüberhinaus zwingend erforderlich, zumindest für eine Übergangszeit bestehende Funktionsbibliotheken weiterverwenden zu können. Um dies zu gewährleisten, stellt SAC eine flexible Schnittstelle zu allen Sprachen bereit, die zum C-Link-Mechanismus kompatibel sind, z.B. C selbst oder FORTRAN. Zu diesem Zweck wird das Modul- und Klassen-Konzept um **externe Module** bzw. **externe Klassen** erweitert. Bei diesen erfolgt die Implementierung in einer von SAC verschiedenen Sprache. Die jeweilige Deklaration dient dem SAC-Compiler als vollständige Beschreibung der Funktionalität. Das hier angewandte Konzept erlaubt sowohl die Übernahme von Datentypen anderer Sprachen in ein SAC-Programm als auch die Verarbeitung von SAC-Datenobjekten durch in einer anderen Sprache implementierte Funktionen. Dabei können auch Seiteneffekt-behaftete Funktionen korrekt in die funktionale Welt von SAC abgebildet werden. In ihrer Anwendung unterscheiden sich externe Module und Klassen in keiner Weise von solchen, die in SAC implementiert sind. Die zur Verfügung gestellte Schnittstelle ermöglicht so die modulweise Migration von einer imperativen Sprache zu SAC.

Die vorliegende Arbeit gliedert sich wie folgt. In Kapitel 2 werden zunächst der Sprachkern von SAC und das Array-Konzept vorgestellt. Kapitel 3 beschreibt das Modul-Konzept. In Kapitel 4 werden die Probleme bei der Integration von Ein-/Ausgabe-Mechanismen in eine funktionale Programmiersprache diskutiert und das Klassen-Konzept von SAC als Lösung vorgestellt. Die Schnittstelle, die SAC gegenüber anderen Sprachen bietet, wird in Kapitel 5 erläutert. Die Kapitel 6 und 7 beschreiben die Kompilation von SAC-Programmen im Detail, wobei der Schwerpunkt auf der Integration des Modul- und Klassen-Konzeptes in den bestehenden Basis-Compiler liegt. Die Arbeit endet mit einer Zusammenfassung in Kapitel 8.

Kapitel 2

Die Programmiersprache SAC

Dieses Kapitel gibt einen Überblick über den Sprachkern und das Array-Konzept von SAC. Zunächst wird die Syntax des Sprachkerns vorgestellt, anschließend die funktionale Interpretation seiner wichtigsten Sprachkonstrukte erläutert. Den Abschluß bildet eine kurze Einführung in das Array-Konzept. Für eine ausführlichere Darlegung des Sprachdesigns und des Array-Konzeptes siehe [Sch94] oder [Wol95].

2.1 Von C nach SAC — Die Syntax

Wie in der Einleitung beschrieben, lehnt sich SAC syntaktisch eng an die imperative Sprache C [KR88] an. Analog zu C besteht ein SAC-Programm aus Typ- und Funktionsdefinitionen sowie einer ausgezeichneten Startfunktion mit dem Namen **main**. Abbildung 2.1 gibt einen ersten Überblick.

Program	⇒	*Definitions* **int main** ( ) *ExprBlock*
Definitions	⇒	$[\, \mathit{TypeDef} \,]^*\, [\, \mathit{FunDef} \,]^*$
TypeDef	⇒	**typedef** *Type Id* ;
Type	⇒	*PrimType* | *Id*
PrimType	⇒	**int** | **float** | **double** | **char** | **bool**
FunDef	⇒	$[$ **inline** $]$ *ReturnList Id ParamList ExprBlock*
ReturnList	⇒	$\mathit{Type}\, [\, ,\, \mathit{Type} \,]^*$
ParamList	⇒	$($ $[\, \mathit{Type\ Id}\, [\, ,\, \mathit{Type\ Id} \,]^*\,]$ $)$

Abbildung 2.1: Aufbau eines SAC-Programms

Funktionsrümpfe bestehen aus Variablen-Deklarationen und Anweisungen. Zu diesen Anweisungen zählen Zuweisungen, Schleifen und bedingte Verzweigungen. Die letzte Anweisung eines Funktionsrumpfes ist immer die aus C bekannte **return**-Anweisung, die den Rückgabewert der Funktion definiert (siehe Abbildung 2.2).

$$
\begin{array}{lcl}
\mathit{ExprBlock} & \Rightarrow & \{\ \left\lceil \mathit{VarDec} \right\rfloor^{*}\ \left\lceil \mathit{Instruction} \right\rfloor^{*}\ \mathit{Return}\ \} \\
\mathit{VarDec} & \Rightarrow & \mathit{Type}\ \mathit{Id}\ ; \\
\mathit{Instruction} & \Rightarrow & \mathit{Assign}\ ; \\
 & | & \mathit{SelAssign}\ ; \\
 & | & \mathit{ForAssign}\ ; \\
\mathit{Assign} & \Rightarrow & \mathit{Id} \left\lceil\ ,\ \mathit{Id} \right\rfloor^{*}\ =\ \mathit{Expr} \\
\mathit{Return} & \Rightarrow & \textbf{return}\ (\ \mathit{Expr} \left\lceil\ ,\ \mathit{Expr} \right\rfloor^{*}\ )\ ; \\
\mathit{SelAssign} & \Rightarrow & \textbf{if}\ (\ \mathit{Expr}\ )\ \mathit{AssignBlock}\ \textbf{else}\ \mathit{AssignBlock} \\
\mathit{ForAssign} & \Rightarrow & \textbf{do}\ \mathit{AssignBlock}\ \textbf{while}\ (\ \mathit{Expr}\ )\ ; \\
 & | & \textbf{while}\ (\ \mathit{Expr}\ )\ \mathit{AssignBlock} \\
 & | & \textbf{for}\ (\ \mathit{Assign}\ ;\ \mathit{Expr}\ ;\ \mathit{Assign}\ )\ \mathit{AssignBlock} \\
\mathit{AssignBlock} & \Rightarrow & \mathit{Instruction} \\
 & | & \{\ \left\lceil \mathit{Instruction} \right\rfloor^{*}\ \}
\end{array}
$$

Abbildung 2.2: Anweisungen in SAC

Die wesentlichen Unterschiede zu C bestehen darin, daß

- lokale Variablen am Beginn eines Funktionsrumpfes (und nur dort) deklariert werden können aber nicht müssen.
- Funktionen mehr als einen Wert direkt zurückliefern können. Aus diesem Grund lassen sich bei der Deklaration einer Funktion mehrere Resultatstypen (*ReturnList*) und in der **return**-Anweisung (*Return*) mehrere Ausdrücke angeben. Die Zuweisung (*Assign*) wird so erweitert, daß mehreren Variablen in einem konzeptuellen Schritt Werte zugewiesen werden können.
- die letzte Anweisung einer Funktion immer eine **return**-Anweisung sein muß und nirgendwo sonst eine **return**-Anweisung stehen darf.
- Funktionen überladen werden können, d.h. ein SAC-Programm darf mehrere Funktionen gleichen Namens enthalten, sofern sich diese in den Typen ihrer formalen Parameter unterscheiden. Diese Form der Überladung stellt nach Strachey/Cardelli einen **ad-hoc Polymorphismus** dar [CW85].
- eine Funktion als **inline** gekennzeichnet werden kann. Diese Compiler-Direktive gibt an, daß bei der Anwendung einer derart ausgezeichneten Funktion ein Aufruf vermieden und stattdessen die Berechnungsvorschrift der Funktion eingesetzt werden soll.

Wie in C bestehen Ausdrücke (*Expr*) aus Anwendungen primitiver und benutzerdefinierter Funktionen, expliziten Typumwandlungen und Konstanten (siehe Abbildung 2.3). Abweichend von C erfolgen explizite Typumwandlungen zwischen den numerischen primitiven Typen **int**, **float** und **double** mit Hilfe der primitiven Funktionen **toi**, **tof** und **tod**. Durch Casts der Form `(:`*Type*`)` *Expr* können dagegen Typumwandlungen zwischen äquivalenten benutzerdefinierten Typen sowie zwischen benutzerdefinierten Typen und ihren jeweiligen primitiven Grundtypen vorgenommen werden.

$$
\begin{array}{lcl}
Expr & \Rightarrow & Id\ (\ \lceil\ Expr \rceil\ ,\ Expr\ \rfloor^{*}\ ) \\
 & | & PrimFunAp \\
 & | & (\ Expr\ ) \\
 & | & (\ :Type\ )\ Expr \\
 & | & Id \\
 & | & Constant \\
 & & \\
PrimFunAp & \Rightarrow & Expr\ BinPrimFun\ Expr \\
 & | & !\ Expr \\
 & | & \textbf{toi}\ (\ Expr\ ) \\
 & | & \textbf{tof}\ (\ Expr\ ) \\
 & | & \textbf{tod}\ (\ Expr\ ) \\
 & & \\
BinPrimFun & \Rightarrow & +\ |\ -\ |\ *\ |\ / \\
 & | & <\ |\ <=\ |\ >\ |\ >=\ |\ ==\ |\ != \\
 & | & \&\&\ |\ ||
\end{array}
$$

Abbildung 2.3: Ausdrücke in SAC

2.2 Die funktionale Interpretation

Im folgenden wird erläutert, wie die Bedeutung der wichtigsten aus C übernommenen Sprachelemente funktional interpretiert wird. Zu diesem Zweck werden beispielhafte SAC-Programmfragmente und dazu semantisch äquivalente Ausdrücke der funktionalen Sprache KiR [Klu94] einander gegenübergestellt.

2.2.1 Die Zuweisung

Auch wenn die C-ähnliche Syntax anderes insinuiert, so handelt es sich bei der Zuweisung in SAC doch ausschließlich um die Substitution einer Variablen durch einen Wert im Sinne einer naiven β-Reduktion des λ-Kalküls: die Zuweisung `a =` *expr*`;`, der eine Sequenz R von weiteren Anweisungen folgt, ist semantisch äquivalent zu dem λ-Kalkül-Ausdruck `(`λ`a.`R *expr*`)`. Eine Sequenz von Zuweisungen wird folglich als Schachtelung von Applikationen und Abstraktionen angesehen. Die mehrfache Zuweisung von Werten an dieselbe Variable innerhalb eines Blockes wird demgemäß als Zuweisungen an verschiedene Variablen gleichen Namens interpretiert. Dabei wird eine Variable durch die nachfolgende Definition einer zweiten Variable gleichen Namens überdeckt. Der Bindungsbereich einer Variablen in SAC erstreckt sich also von ihrer Definition bis zur nächsten Definition einer Variablen gleichen Namens. Abbildung 2.4 veranschaulicht dieses Prinzip anhand eines SAC-Anweisungsblocks und eines semantisch äquivanten KiR-Ausdrucks.

```
(1)  {                                    let A = 5
(2)    A = 5;                             in  let B = 7
(3)    B = 7;                                 in  let A = A+B
(4)    A = A+B;                                   in  A
(5)    return(A);
(6)  }
```

Abbildung 2.4: Die funktionale Interpretation der Zuweisung

In diesem Beispiel entspricht die zweite Zuweisung an die Variable `A` in Zeile 4 der Definition einer neuen Variablen. Diese überdeckt bis zum Ende des Blockes die in Zeile 2 definierte Variable gleichen Namens. Das `A` in der **return**-Anweisung ist folglich an die Variable `A` aus Zeile 4 gebunden und steht somit für den Wert 12. Der Bindungsbereich der in Zeile 2 definierten Variablen `A` erstreckt sich demgemäß von Zeile 3 bis einschließlich der rechten Seite der Zuweisung in Zeile 4.

2.2.2 Die bedingte Verzweigung

In SAC ist die bedingte Verzweigung, im folgenden kurz Verzweigung genannt, Teil einer Sequenz von Anweisungen. Das hat zur Folge, daß nach der Auswertung der Verzweigungsbedingung entweder die Anweisungen der Konsequenz oder die der Alternative und anschließend die der Verzweigung folgenden Anweisungen ausgeführt werden. Dadurch kann es zu einem scheinbaren Konflikt mit dem funktionalen Paradigma kommen, wenn wie in dem Beispiel aus Abbildung 2.5 einer Variablen in Konsequenz oder Alternative einer Verzweigung ein Wert zugewiesen wird. In diesem Fall ist bei einem angewandten Vorkommen dieser Variablen in den auf die Verzweigung folgenden Anweisungen nicht statisch entscheidbar, an welche Definition diese gebunden ist. Bezogen auf das Beispiel lautet die Frage, ob das `A` in Zeile 7 an die Definition in Zeile 2 oder an die in Zeile 4 gebunden ist. Diesem Problem wird durch eine funktionale Interpretation der Verzweigung begegnet, die sowohl Konsequenz als auch Alternative um die der Verzweigung folgenden Anweisungen ergänzt. Wie Abbildung 2.5 zeigt, entsteht dabei ein semantisch äquivalentes Programm, in dem die Bindungsbereiche der Variablen statisch entscheidbar sind.

```
(1)  {                                    let A=5
(2)    A = 5;                             in  if  pred
(3)    if ( pred )                            then let A = 7
(4)      A = 7;                                    in  let B = A+2
(5)    else                                            in  B
(6)      B = 43;                                 else let B = 43
(7)    B = A+2;                                        in  let B = A+2
(8)    return(B);                                          in  B
(9)  }
```

Abbildung 2.5: Die funktionale Interpretation der bedingten Verzweigung

2.2.3 Die Schleifen

Schleifen werden in SAC als abkürzende Schreibweisen für tail-end-rekursive Funktionen interpretiert. Abbildung 2.6 veranschaulicht dieses Prinzip am Beispiel der **while**-Schleife stellvertretend für die anderen in SAC zur Verfügung stehenden Schleifen.

```
(1)  C = 7;
(2)  A = 5;
(3)  I = 0;
(4)  while (I<6)
(5)  {
(6)    C = A+5;
(7)    A = 2*C;
(8)    I = I+1;
(9)  }
(10) ...
```

$\Longleftrightarrow$

```
let C = 7,
    A = 5,
    I = 0
in  def F_while[C,A,I] = if (I<6)
                         then let C = A+5
                              in  let A = 2*C,
                                      I = I+1
                                  in  F_while[C,A,I]
                         else <C,A,I>
    in  letp[1] <C,A,I> = F_while[C,A,I]
        in  ...
```

Abbildung 2.6: Die funktionale Interpretation der **while**-Schleife

2.3 Das Array-Konzept von SAC

Das Array-Konzept von SAC findet seine mathematische Grundlage in dem von L. Mullin entwickelten ψ-Kalkül [Mul88, MT94]. Darin wird ein Array durch zwei Vektoren repräsentiert. Während der **Element-Vektor** die Elemente des Arrays enthält, definiert der **Form-Vektor** dessen strukturelle Eigenschaften. Seine Länge entspricht der Dimensionalität des Arrays. Die einzelnen Elemente geben die jeweilige Anzahl der Array-Elemente pro Dimension wieder.

Die Struktur eines Arrays kann auf verschiedene Arten angegeben werden. Abbildung 2.7 zeigt diese am Beispiel der Definition eines Arrays, das die Zahlen 1 bis 6 in Form einer 2×3-Matrix enthält. Eine Struktur kann entweder implizit durch entsprechende Schachtelung eindimensionaler Arrays (`A`), explizit bei der (optionalen) Variablen-Deklaration (`B`) oder mit Hilfe der primitiven Funktion **reshape** (`C`) definiert werden. Abbildung 2.8 enthält eine formale Beschreibung der syntaktischen Erweiterungen.

```
{
   int[2,3] B;

   A = [[1,2,3], [4,5,6]];
   B = [1,2,3,4,5,6];
   C = reshape([2,3], [1,2,3,4,5,6]);
       ...
}
```

Abbildung 2.7: Verschiedene Formen der Definition von Arrays

Auf diesen Arrays ist eine Reihe von dimensionsunabhängigen primitiven Operationen definiert. Die wichtigsten werden im folgenden kurz erläutert. Dabei bezeichne A ein Array und v einen Form-Vektor.

- `dim`(A)
  liefert die Dimension des Arrays A.

[1] Das `letp`-Konstrukt von KiR stellt eine abkürzende Schreibweise für einen Pattern-Matching-Ausdruck dar. In dem hier vorliegenden Fall wird die Applikation von F_{while} zu einer drei-elementigen Liste reduziert. Die Elemente der Muster-Liste `<C,A,I>` werden in ihrem Bindungsbereich durch die Elemente dieser Liste substituiert.

- `shape(`A`)`
  liefert den zu dem Array A gehörenden Form-Vektor.

- `reshape(`v`, `A`)`
  liefert ein Array, das dieselben Elemente wie das Array A hat, jedoch die durch den Form-Vektor v spezifizierte Struktur. Voraussetzung ist natürlich, daß A und v zueinander kompatibel sind, d.h. die Anzahl der Elemente von A muß die durch v beschriebene Struktur zulassen.

- `psi(`v`, `A`)`
  liefert das durch v spezifizierte Element oder Sub-Array des Arrays A. In diesem Fall wird v auch als Index-Vektor bezeichnet. Voraussetzung ist, daß die Länge des Index-Vektors kleiner oder gleich der Dimension des Arrays ist ($|v| \leq$ `dim(`A`)`) und daß jedes seiner Elemente kleiner oder gleich der Anzahl der Array-Elemente in der betreffenden Dimension ist ($\forall\, i \in \{1, \ldots, |v|\} : v_i \leq$ `shape(`A`)`$_i$). Falls `|`v`| = dim(`A`)`, wird das angegebene Array-Element von A zurückgeliefert, anderenfalls ein entsprechendes Sub-Array. Als abkürzende Schreibweise ist auch `A[v]` zugelassen.

- `modarray(`A`, `v`, `*expr*`)`
  liefert ein Array, das sich von A lediglich in dem durch v spezifizierten Element unterscheidet, welches den Wert *expr* erhält.

Darüber hinaus gibt es primitive Operationen u.a.

- zur elementweisen arithmetischen Verknüpfung von Arrays untereinander sowie mit Skalaren,
- zum Selektieren von Teil-Arrays,
- zum Konkatenieren von Arrays,
- zum Rotieren von Arrays um eine Achse.

Als flexiblen Mechanismus zur Manipulation von Arrays besitzt SAC darüberhinaus die **With-Loop**, deren Syntax[2] Abbildung 2.8 entnommen werden kann. Bei der With-Loop handelt es sich um einen Ausdruck, bestehend aus einem *Generator*, einem optionalen Anweisungsblock und einem *Operator*. Der *Generator* definiert durch Angabe einer unteren und oberen Grenze eine Menge von Index-Vektoren. Für jeden Index-Vektor wird simultan der Anweisungsblock ausgeführt. Der Wert der With-Loop wird durch den *Operator* beschrieben:

- `genarray(`*shape*`, `*expr*`)`
  erzeugt ein Array mit der durch *shape* angegebenen Form. Die durch den *Generator* spezifizierten Elemente werden mit *expr* initialisiert, die anderen mit einem Default-Wert.

- `modarray(`*array*`, `*id*`, `*expr*`)`
  erzeugt ein Array, das dieselbe Form wie *array* hat. Die durch den *Generator* spezifizierten Elemente werden mit *expr* initialisiert, die anderen mit dem jeweils entsprechenden Element von *array*. *id* bestimmt die zu verwendende *Generator*-Variable.

- `fold(`*fun*`, `*array*`, `*neutral*`)`
  verknüpft alle durch den *Generator* spezifizierten Elemente des Arrays *array* mit der Funktion *fun*. Die angegebene Funktion muß zwei-stellig, kommutativ und assoziativ sein. Für benutzerdefinierte Funktionen muß zusätzlich ein neutrales Element *neutral* angegeben werden. Dieses kann für die primitiven Operatoren + und * entfallen.

[2] Die hier angegebene Syntax entspricht der Version 1.0 der SAC-Sprachdefinition und unterscheidet sich insofern von den in [Sch94], [Wol95] und [Sie95] angegebenen Definitionen.

$$
\begin{array}{lcl}
Type' & \Rightarrow & ... \\
 & | & PrimType\ [\ IntConst\ \lceil\ ,\ IntConst\ \rfloor^{*}\ ] \\
 & | & Id\ [\ IntConst\ \lceil\ ,\ IntConst\ \rfloor^{*}\ ] \\
 & | & PrimType\ [\ ] \\
 & | & Id\ [\ ] \\
 & & \\
Expr' & \Rightarrow & ... \\
 & | & [\ Expr\ \lceil\ ,\ Expr\ \rfloor^{*}\ ] \\
 & | & \textbf{with}\ (\ Generator\ )\ \lceil\ AssignBlock\ \rfloor\ Operator \\
 & & \\
Generator & \Rightarrow & Expr\ <=\ Id\ <\ Expr \\
 & & \\
Operator & \Rightarrow & \textbf{genarray}\ (\ [\ IntConst\ \lceil\ ,\ IntConst\ \rfloor^{*}\ ]\ ,\ Expr\ ) \\
 & | & \textbf{modarray}\ (\ Expr\ ,\ Id\ ,\ Expr\ ) \\
 & | & \textbf{fold}\ (\ FoldFun\ ,\ Expr\ \lceil\ ,\ Expr\ \rfloor\ ) \\
 & & \\
FoldFun & \Rightarrow & +\ |\ *\ |\ Id
\end{array}
$$

Abbildung 2.8: Erweiterungen der Syntax im Zusammenhang mit Arrays

Die primitiven Array-Operationen bilden die Grundlage für die dimensionsunabhängige Spezifikation von Algorithmen. Dafür ist es jedoch zusätzlich erforderlich, auch benutzerdefinierte Funktionen dimensionsunabhängig definieren zu können. Diesem Zweck dienen die **variablen Arraytypen**. Ein variabler Arraytyp ist der Supertyp aller Arraytypen des jeweiligen Basistyps mit konkretem Form-Vektor. Der variable Arraytyp `int[]` ist beispielsweise Supertyp aller Integer-Arraytypen, wie `int[2,3]` oder `int[4,5,6,7]`. Diese sind umgekehrt Subtypen des variablen Arraytyps `int[]`. Mit Hilfe variabler Arraytypen lassen sich Funktionen definieren, die auf Arrays beliebiger Form und Dimension angewandt werden können. Lediglich die jeweiligen Basistypen müssen übereinstimmen. Derartige Funktionen werden auch als **parametrisiert polymorph** bezeichnet [CW85].

Kapitel 3

Module

In diesem Kapitel werden zunächst die grundlegenden Anforderungen an ein Modul-Konzept diskutiert. Darauf aufbauend werden beispielhaft die Modul-Konzepte der imperativen Sprache MODULA-2 sowie der funktionalen Sprache HASKELL vorgestellt und schließlich vor diesem Hintergrund das Modul-Konzept von SAC erläutert.

3.1 Aufgaben eines Modul-Konzeptes

Die primäre Aufgabe eines Modul-Konzeptes besteht darin, die Strukturierung großer Programme jenseits der Ebene von Funktionen zu ermöglichen. Zu diesem Zweck werden Funktionen und Typen zu Modulen zusammengefaßt[1]. Ein vollständiges Programm setzt sich wiederum aus mehreren Modulen zusammen, die sich üblicherweise in getrennten Dateien befinden. Von diesen muß genau eines ein ausgezeichnetes Start-Modul sein, das den Beginn der Berechnungsvorschrift enthält.

Ein Modul-Konzept beschränkt sich jedoch nicht auf die textuelle Separierung von Programmteilen. Funktionen und Typen sind außerhalb des Moduls, in dem sie definiert sind, grundsätzlich unbekannt. Erst wenn sie von diesem explizit exportiert werden, sind sie von außen zugänglich. Für eine konkrete Verwendung in einem anderen Modul müssen sie von diesem wiederum importiert werden. Durch die Notwendigkeit expliziten Ex- und Importierens entstehen zwischen den einzelnen Modulen definierte Schnittstellen. Damit wird die systematische Wiederverwendung von Programm-Code durch Anlage von Modul-Bibliotheken erheblich erleichtert. Zudem ist diese weitergehende Möglichkeit der Strukturierung eines Programmes identisch mit einem zusätzlichen Abstraktionsniveau. Von außen betrachtet ist nur die Export-Schnittstelle eines Moduls von Bedeutung, während von seiner tatsächlichen Implementierung vollständig abstrahiert werden kann.

Wenn nicht nur das Start-Modul sondern bliebige Module ihrerseits weitere Module importieren können, entsteht eine Hierarchie von Modulen und damit auch eine Hierarchie von Abstraktionen. Das Start-Modul steht an der Spitze dieser Hierarchie, solche Module, die ihrerseits keine weiteren Module importieren, bilden die Basis. Auf diese Weise wird eine Technik der Software-Entwicklung unterstützt, die aufbauend auf rudimentären Strukturen durch fortgesetzte Abstraktion zu komplexen Programmen führt. Umgekehrt ist jedoch für den Anwender eines Moduls die darunterliegende Hierarchie vollständig transparent, da sie Teil der Implementierung und nicht der Export-Schnittstelle ist.

[1] Je nach zugrundeliegender Sprache können weitere Arten von Symbolen hinzukommen.

Ein entscheidender Schritt besteht darin, auch die separate Kompilation von Modulen zuzulassen. In diesem Fall werden Module nicht (nur) in Form von Quell-Code gespeichert, sondern als fertig übersetzter Objekt-Code. Dadurch wird von der konkreten Implementierung eines Moduls nicht nur abstrahiert, sondern diese sogar vollständig versteckt. Das Binden der einzelnen Objekt-Module zu einem ausführbaren Programm kann dann statisch bei der Kompilation eines Start-Moduls oder dynamisch während der Ladephase erfolgen. Separate Kompilation erfordert die textuelle Trennung der Funktions- und Typdefinitionen eines Moduls einerseits und der Beschreibung seiner Export-Schnittstelle andererseits. Diese wird vom Compiler zur Übersetzung derjenigen Module benötigt, die ihrerseits dieses Modul importieren. Ein Modul-Konzept mit separater Kompilation bietet zahlreiche Vorteile, die es für die Entwicklung größerer Programme unentbehrlich machen:

- Das getrennte Entwickeln, Testen und Warten von Programmteilen durch mehrere Programmierer wird ermöglicht.
- Die Weitergabe von Modulen kann ohne Offenlegung des Quell-Codes erfolgen.
- Die Implementierung eines Moduls kann jederzeit verändert oder ausgetauscht werden. Solange die Schnittstelle davon nicht betroffen ist, hat dies keinen Einfluß auf die anderen Module eines Programms.
- Durch Geheimhaltung des Quell-Codes können Module vor Manipulation geschützt werden. Dadurch kann das korrekte Verhalten z.B. von Standard-Modulen als sehr wahrscheinlich vorausgesetzt werden, was die Fehlersuche eingeschränkt.

3.2 Modul-Konzepte im Vergleich

3.2.1 Das Modul-Konzept von MODULA-2

Die wesentliche Weiterentwicklung der Sprache Modula-2 [Wir85] gegenüber ihrem Vorgänger Pascal [JW74] besteht in der Unterstützung modularer Programmierung. In Modula-2 setzt sich ein Programm i.a. aus mehreren Modulen zusammen, die jeweils einzeln übersetzt werden können. Ein Modul wird durch das Schlüsselwort **MODULE** und den Modulnamen eingeleitet. Danach folgen Import-Anweisungen, die es ermöglichen, andere Module sowohl vollständig als auch teilweise durch die konkrete Angabe von Bezeichnern zu importieren. Im anschließenden Definitionsteil können Typen, Konstanten, (globale) Variablen und Prozeduren definiert werden. Das Modul endet mit einem in die Schlüsselwörter **BEGIN** und **END** eingeschlossenen Anweisungsteil. Importierte Bezeichner werden im Definitions- und Anweisungsteil durch **qualifizierte Namen**, bestehend aus dem Modulnamen und dem eigentlichen Bezeichner getrennt durch einen Punkt, referenziert.

Modula-2 unterscheidet zwischen exportierenden und nicht exportierenden Modulen. Die nicht exportierenden Module entsprechen den Start-Modulen im Sinne von Abschnitt 3.1. Bei ihnen bildet der Anweisungsteil den Beginn der Berechnungsvorschrift des gesamten Programms. Den exportierenden Modulen wird das Schlüsselwort **IMPLEMENTATION** vorangestellt. Bei diesen sog. **Implementierungsmodulen** dient der Anweisungsteil lediglich zur Initialisierung lokaler Datenstrukturen.

Zu jedem Implementierungsmodul gehört ein **Definitionsmodul**, das die Beschreibung der Export-Schnittstelle darstellt. Ein Definitionsmodul beginnt mit dem Schlüsselwort **DEFINITION MODULE**, gefolgt von dem Modulnamen. Mit den anschließenden Import-Anweisungen können solche Bezeichner verfügbar gemacht werden, die zwar im folgenden benötigt werden, aber nicht im

korrespondierenden Implementierungsmodul selbst definiert sind. Dabei kann es sich z.B. um den Typ einer Variablen handeln, der vom Implementierungsmodul seinerseits importiert wird. Danach folgt der Deklarationsteil, bestehend aus Konstanten-, Typen-, Variablen- und Prozedur-Deklarationen. Typen können auf zwei verschiedene Weisen deklariert werden. Bei der **transparent export** genannten Form wird die vollständige Typdefinition, einschließlich z.B. der Feldnamen von Records, angegeben. Dagegen wird bei der **opaque export** genannten Form lediglich der Typbezeichner exportiert. Auf diese Weise wird das Konzept abstrakter Datentypen realisiert. Die Anwendung von opaque export ist allerdings auf Zeigertypen beschränkt.

3.2.2 Das Modul-Konzept von HASKELL

Analog zu MODULA-2 besteht auch ein Programm der funktionalen Sprache HASKELL [H$^+$95] i.a. aus mehreren Modulen. Genau eines davon muß den Namen `Main` tragen und den Wert `main` exportieren. Dieser Wert definiert den Wert des gesamten Programms.

Ein HASKELL-Modul besteht aus einer Implementierung und einer Schnittstelle. Die Modul-Implementierung bildet einen wechselseitig rekursiven Bindungsbereich für HASKELL-Symbole, wie Werte, algebraische Datentypen, Typsynonyme oder Typklassen. Sie beginnt mit dem Schlüsselwort **module**, dem Modulnamen und einer Liste von Bezeichnern für Symbole, die von dem Modul exportiert werden. Im Gegensatz zu MODULA-2, wo aus einem Implementierungsmodul allein nicht hervorgeht, welche Symbole exportiert werden, muß dies bei HASKELL also explizit angegeben werden. Dabei kann ein HASKELL-Modul auch solche Symbole exportieren, die es nicht selbst definiert, sondern lediglich seinerseits aus einem anderen Modul importiert. Im Anschluß an die Exportliste folgen Importe aus anderen Modulen sowie die Definitionen der Symbole.

Da der reine Umfang der Exporte eines Moduls bereits durch die Implementierung festgelegt ist, dient die Modul-Schnittstelle lediglich der Beschreibung der exportierten Symbole für ein importierendes Modul. Ziel ist die statische Typüberprüfung eines Moduls durch ausschließliche Inspektion der eigenen Implementierung sowie der Schnittstellen der importierten Module. Die Modul-Schnittstelle wird durch das Schlüsselwort **interface** und den Modulnamen eingeleitet. Danach folgt eine Import-Liste. Diese hat die Aufgabe, für solche Symbole, die von dem betreffenden Modul zwar exportiert in seiner Implementierung jedoch nicht definiert werden, den Bezug zum definierenden Modul herzustellen. Anschließend kommen die Beschreibungen der exportierten Symbole, z.B. Typsignaturen für Werte. Das Konzept abstrakter Datentypen wird von HASKELL dadurch integriert, daß algebraische Datentypen ohne ihre jeweiligen Konstruktoren exportiert werden können.

Eine besondere Stellung im Modul-Konzept von HASKELL nehmen die **Prelude**- und die **Library**-Module ein, deren Namen mit dem Präfix `Prelude` bzw. `Lib` beginnen. Dabei handelt es sich um vordefinierte Standard-Module, deren exportierte Symbole jedoch selbst Teil der Sprachdefinition von HASKELL sind und einen Großteil der Leistungsfähigkeit dieser Sprache ausmachen. Die Prelude-Module werden standardmäßig sowohl von den Implementierungen als auch den Schnittstellen aller anderen Module importiert.

3.3 Das Modul-Konzept von SAC

3.3.1 Überblick

Das Modul-Konzept von SAC folgt in wesentlichen Punkten denen von MODULA-2 oder HASKELL. Ein Modul besteht in SAC grundsätzlich aus zwei getrennten Teilen, der **Modul-Implementierung** und der **Modul-Deklaration**. Während die Modul-Implementierung Typ- und Funktionsdefinitionen enthält, stellt die Modul-Deklaration eine Beschreibung der Export-Schnittstelle dar. Ein Beispiel für ein SAC-Modul befindet sich in Anhang A.

Modul-Implementierung und Modul-Deklaration bilden neben dem in Abschnitt 2.1 vorgestellten SAC-Programm zwei neue syntaktische Einheiten (vgl. Abbildung 3.1). Anders als z.B in MODULA-2 wird damit der unterschiedliche Charakter von Start-Modulen gegenüber anderen Modulen stärker betont. Während Module i.a. die Aufgabe haben, anderen Modulen eine bestimmte Funktionalität zur Verfügung zu stellen, verknüpft das Start-Modul die Funktionalitäten anderer Module zu einer vollständigen Anwendung. Aus diesem Grund hat ein Start-Modul auch keine Export-Schnittstelle. Da es von anderen Modulen nicht importiert werden kann, benötigt es keinen Namen, sondern muß lediglich als solches erkennbar sein.

3.3.2 Die Modul-Implementierung

Bei der Modul-Implementierung handelt es sich um eine Sammlung von Funktions- und Typdefinitionen. Die genaue Syntax wird in Abbildung 3.1 beschrieben. Eine Modul-Implementierung wird durch das Schlüsselwort **Module** eingeleitet, gefolgt vom Namen des Moduls. Danach folgen Typdefinitionen sowie Funktionsdefinitionen in der in Kapitel 2 angegebenen Form. Im Unterschied zu Programmen besitzen Modul-Implementierungen keine **main**-Funktion.

$$
\begin{array}{lcl}
\text{SAC} - \mathit{Code} & \Rightarrow & \mathit{Program} \mid \mathit{ModuleImp} \mid \mathit{ModuleDec} \\
\mathit{ModuleImp} & \Rightarrow & \textbf{Module}\ \mathit{Id}: \ \mathit{Definitions} \\
\mathit{ModuleDec} & \Rightarrow & \textbf{ModuleDec}\ \mathit{Id}\ :\ \textbf{own}: \mathit{Declarations} \\
\mathit{Declarations} & \Rightarrow & \{\ \lceil\ \mathit{ITypeDec}\ \rceil\ \lceil\ \mathit{ETypeDec}\ \rceil\ \lceil\ \mathit{FunDec}\ \rceil\ \} \\
\mathit{ITypeDec} & \Rightarrow & \textbf{implicit types}\ :\lceil\ \mathit{Id}\ ;\ \rfloor^{*} \\
\mathit{ETypeDec} & \Rightarrow & \textbf{explicit types}\ :\lceil\ \mathit{Id}\ =\ \mathit{Type}\ ;\ \rfloor^{*} \\
\mathit{FunDec} & \Rightarrow & \textbf{functions}\ :\lceil\ \mathit{ReturnList}\ \mathit{Id}\ \mathit{ParamList}\ ;\ \rfloor^{*}
\end{array}
$$

Abbildung 3.1: Syntax von Modulen

3.3.3 Die Modul-Deklaration

Die Modul-Deklaration stellt eine Beschreibung der Export-Schnittstelle eines Moduls dar. Sie enthält die Deklarationen der exportierten Funktionen und Typen. Ein Typ kann von auf zwei verschiedene Weisen exportiert werden, entweder als **expliziter Typ** oder als **impliziter Typ**. Während bei einem expliziten Typ die Typdefinition offen gelegt wird, handelt es sich bei impliziten Typen um abstrakte Datentypen. Ihre tatsächliche Definition ist ausschließlich in der Modul-Implementierung bekannt. Die genaue Syntax von Modul-Deklarationen ist Abbildung 3.1 zu entnehmen.

Eine Modul-Deklaration beginnt mit dem Schlüsselwort **ModuleDec** und dem Modul-Namen. Das Schlüsselwort **own** leitet den eigentlichen Deklarationsteil ein, der aus drei (optionalen) Abschnitten besteht. Der erste Abschnitt enthält die Namen der impliziten Typen, der zweite die Namen und Definitionen der expliziten Typen und der dritte die Prototypen der exportierten Funktionen. Die hier deklarierten Typen und Funktionen müssen mit entsprechenden Definitionen in der Modul-Implementierung koinzidieren.

3.3.4 Das Importieren von Modulen

Voraussetzung für die Verwendung von Typen und Funktionen anderer Module innerhalb eines Programmes oder einer Modul-Implementierung ist deren expliziter Import. Zu diesem Zweck wird die Syntax von Programmen und Modul-Implementierungen um die **import**-Anweisung erweitert. Deren genaue Form kann Abbildung 3.2 entnommen werden.

$$
\begin{array}{lcl}
\mathit{Program}' & \Rightarrow & \mathit{Imports}\ \mathit{Definitions}\ \mathit{Main} \\
\mathit{ModuleImp}' & \Rightarrow & \textbf{Module}\ \mathit{Id}: \mathit{Imports}\ \mathit{Definitions} \\
\mathit{ModuleDec}' & \Rightarrow & \textbf{ModuleDec}\ \mathit{Id}\ :\ \mathit{Imports}\ \textbf{own}: \mathit{Declarations} \\
\mathit{Imports} & \Rightarrow & \left[\ \mathit{ImportBlock}\ \right]^{*} \\
\mathit{ImportBlock} & \Rightarrow & \textbf{import}\ \mathit{Id}\ :\ \textbf{all}\ ; \\
 & | & \textbf{import}\ \mathit{Id}\ :\ \mathit{ImportList} \\
\mathit{ImportList} & \Rightarrow & \{\ \left[\ \mathit{ITypeImp}\ \right]\left[\ \mathit{ETypeImp}\ \right]\left[\ \mathit{FunImp}\ \right]\ \} \\
\mathit{ITypeImp} & \Rightarrow & \textbf{implicit types}\ :\ \mathit{Id}\left[\ ,\ \mathit{Id}\ \right]^{*}\ ; \\
\mathit{ETypeImp} & \Rightarrow & \textbf{explicit types}\ :\ \mathit{Id}\left[\ ,\ \mathit{Id}\ \right]^{*}\ ; \\
\mathit{FunImp} & \Rightarrow & \textbf{functions}\ :\ \mathit{Id}\left[\ ,\ \mathit{Id}\ \right]^{*}\ ;
\end{array}
$$

Abbildung 3.2: Syntax der **import**-Anweisung

Außer in Programmen und Modul-Implementierungen können **import**-Anweisungen auch in Modul-Deklarationen auftreten. An dieser Stelle ermöglichen sie das implizite Importieren von Modulen. Angenommen, in der Deklaration von Modul A wird ein Modul B importiert. Das hat zur Folge, daß immer, wenn Modul A importiert wird, implizit auch Modul B importiert wird. Dadurch werden die von B exportierten Typen und Funktionen faktisch Bestandteil von Modul A.

Die **import**-Anweisung besteht aus dem Schlüsselwort **import** und dem Namen des zu importierenden Moduls. Bei der Import-Beschreibung kann zwischen pauschalem und selektivem Importieren unterschieden werden. Durch das Schlüsselwort **all** werden sämtliche Funktionen und Typen des angegebenen Moduls einschließlich der von diesem implizit importierten Module importiert. Beim selektiven Importieren können dagegen die Namen der zu importierenden Funktionen und Typen unmittelbar angegeben werden. Da in SAC Funktionen grundsätzlich überladen werden können, hat die Beschränkung auf die Angabe von Identifikatoren an dieser Stelle zur Folge, daß alle Funktionen gleichen Namens importiert werden.

In SAC besitzt jedes Modul einen eigenen Namensraum. Das bedeutet, ein Typ oder eine Funktion aus Modul A konfligiert nicht mit einem gleichnamigen Typ oder einer gleichnamigen Funktion aus Modul B, selbst wenn A von B oder beide gemeinsam von einem dritten Modul oder Programm importiert werden. Diese Freiheit erfordert jedoch, daß zwischen gleichnamigen Typen oder Funktionen aus verschiedenen Modulen unterschieden werden kann. Zu diesem Zweck wird die Syntax von SAC dahingehend erweitert, daß überall dort, wo ein Funktions- oder Typname in angewandter (nicht definierender) Form vorkommen darf, diesem ein Modulname, abgetrennt durch einen Doppelpunkt, vorangestellt werden kann. Macht der Programmierer von dieser Möglichkeit keinen Gebrauch, wird zunächst in der eigenen Modul-Implementierung oder dem Programm nach einer Definition des betreffenden Bezeichners gesucht. Führt diese Suche nicht zum Erfolg, so werden zusätzlich alle importierten Funktionen und Typen durchsucht. Werden in diesem Fall mehrere Definitionen gefunden, kann keine eindeutige Zuordnung des Bezeichners zu einem Symbol getroffen werden, was in einer entsprechenden Fehlermeldung des Compilers resultiert. In einem derartigen Fall ist die explizite Angabe eines Modulnamens zwingend erforderlich.

Kapitel 4

Ein-/Ausgabe

In diesem Kapitel werden zunächst die grundsätzlichen Probleme der Integration von Ein-/Ausgabe-Mechanismen in eine funktionale Programmiersprache untersucht. Anschließend werden mögliche Lösungsansätze vorgestellt, wobei der Schwerpunkt einerseits auf dem Monaden-Konzept der Sprache Haskell, andererseits auf dem Uniqueness-Typing-Konzept der Sprache Clean liegt. Zuletzt wird das auf Uniqueness-Typing basierende Klassen-Konzept von Sac schrittweise eingeführt. Dabei wird im Rahmen einer Fallstudie die Anwendbarkeit der vorgestellten Mechanismen für die Realisierung von Ein-/Ausgabe untersucht.

4.1 Ein-/Ausgabe und das funktionale Paradigma

In imperativen Sprachen wird Ein-/Ausgabe gewöhnlich mit Hilfe von Funktionen realisiert, die Datenstrukturen durch Seiteneffekte manipulieren. Eine komplexe Ein-/Ausgabe-Operation wird durch eine Sequenz von atomaren Operationen realisiert. Dabei garantiert der sequentielle Kontrollfluß deterministische Resultate. Diese Technik koinzidiert mit einem intuitiven Verständnis von Ein-/Ausgabe als Folge von Aktionen auf einem Ein-/Augabe-Gerät[1] und stellt daher eine geeignete Methode zu deren Spezifikation dar.

Das funktionale Paradigma kennt dagegen weder Seiteneffekte noch eine festgelegte Ausführungsreihenfolge. Berechnungen erfolgen durch das geordnete Konsumieren und Reproduzieren von Ausdrücken und nicht durch die Modifikation von Datenstrukturen. Auf der damit verbundenen Freiheit von Seiteneffekten beruhen die zentralen Vorteile des funktionalen Paradigmas wie referentielle Transparenz oder die Church-Rosser-Eigenschaft. Die Integration von einfach verwendbaren und intuitiv verständlichen Ein-/Ausgabe-Mechanismen stellt dies jedoch vor erhebliche Probleme.

Bei Ein-/Ausgabe-Operationen ist die Garantie einer bestimmten Ausführungsreihenfolge entscheidend. So muß z.B. eine Datei erst geöffnet werden, bevor aus ihr gelesen oder in sie geschrieben werden kann. Bei Interaktionen mit einem Benutzer soll eine Eingabe-Aufforderung auf dem Bildschirm angezeigt werden, bevor auf eine entsprechende Tastatur-Eingabe gewartet wird. Dazu betrachte man das folgende Sac-Codefragment:

[1] Der Begriff Ein-/Ausgabe-Gerät ist in diesem Zusammenhang sehr allgemein zu verstehen. Neben tatsächlichen Geräten, wie z.B. einem Terminal oder einem Drucker, umfaßt er ebenso Dateisysteme oder einzelne Dateien.

```
z = fprintstring(stdout, "Bitte Zahl eingeben :");
a = fscanint(stdin);
```

Obwohl die C-ähnliche Syntax von SAC einen sequentiellen Kontrollfluß suggeriert, ist die tatsächliche Ausführungsreihenfolge dieser voneinander unabhängigen Funktionsanwendungen vollkommen unbestimmt.

Das Fehlen von Zuständen und ihren Transformationen verhindert die einfache Spezifikation von in imperativen Sprachen üblichen Ein-/Ausgabe-Operationen. Man betrachte beispielsweise folgende Funktion zur Ausgabe eines Zeichens in eine Datei:

```
File putc(File, char)
```

Die Funktion `putc` erwartet als Argumente eine Datei und ein Zeichen. Sie kann das Zeichen jedoch nicht einfach an das Ende der Datei schreiben und diese zurückliefern. Die ursprüngliche Datei könnte zwischen verschiedenen Funktionsanwendungen im Sharing verwendet werden. Die (destruktive) Modifikation des Arguments würde dann die Resultate dieser Funktionsanwendungen beeinflussen. Damit erhielte die Ausführungsreihenfolge von Funktionsanwendungen Einfluß auf das Ergebnis der Berechnung und die Church-Rosser-Eigenschaft wäre nicht mehr erfüllt. Das folgende Beispiel verdeutlicht dieses Problem:

```
File, File fun(File f)
{
  a = putc(f, 'a');
  b = putc(f, 'b');
  return(a,b);
}
```

Falls die Funktion `putc` die als Argument übergebene Datei modifiziert, ist das Resultat einer Anwendung von `fun` abhängig von der Ausführungsreihenfolge der beiden Anwendungen von `putc` in ihrem Rumpf. Das Resultat wäre entweder `(f:"a", f:"ab")` oder `(f:"ba", f:"b")`. Bei einem von Seiteneffekten freien Verhalten von `putc` wäre das Resultat dagegen unabhängig von der Ausführungsreihenfolge `(f:"a", f:"b")`.

Um die Church-Rosser-Eigenschaft zu garantieren, verbietet das funktionale Paradigma daher Seiteneffekte und beschränkt Funktionen dadurch auf das Konsumieren und Reproduzieren von Werten. Für die Funktion `putc` bedeutet dieses Prinzip, daß bei jeder Anwendung eine neue Datei erzeugt werden muß, deren Inhalt dem der übergebenen Datei, ergänzt um das gegebene Zeichen, entspricht. Dies ist jedoch nicht nur sehr ineffizient, sondern entspricht auch nicht dem intendierten Verhalten. Üblicherweise soll eine bestehende Datei tatsächlich modifiziert werden. Besonders deutlich zeigt sich dieses Problem dann, wenn physikalische Geräte, z.B. ein Terminal, betroffen sind, die nicht kopiert werden können.

4.2 Ein-/Ausgabe-Konzepte funktionaler Sprachen

4.2.1 Überblick

Für die Integration von Ein-/Ausgabe-Mechanismen in funktionale Programmiersprachen gibt es zahlreiche unterschiedliche Konzepte. Übersichten finden sich in [HS89, Per91, Gor92]. Die vorgeschlagenen Konzepte lassen sich grob einteilen in **Stream**-basierte und **Environment**-basierte Ansätze.

Die Stream-basierten Ansätze [Dwe89, Tho90, Tur90, CH93] beruhen auf der Abbildung eines Eingabe-Streams auf einen Ausgabe-Stream. Dabei dient der Ausgabe-Stream neben der eigentlichen Ausgabe zusätzlich der Anforderung von Eingabe-Daten. Eine außerhalb des funktionalen Programms befindliche Instanz, gewöhnlich das Betriebssystems, garantiert das ordnungsgemäße Funktionieren der Streams und führt die eigentlichen Ein-/Ausgabe-Operationen aus.

Die Environment-basierten Ansätze [WW88, BWW90, JW93, AP95] versuchen dagegen, die in Abschnitt 4.1 beschriebenen Probleme auf eine direktere Weise zu lösen. Sie machen sich die Tatsache zunutze, daß auch im funktionalen Paradigma unter bestimmten Umständen das destruktive Modifizieren von Datenstrukturen möglich ist. Wenn sichergestellt werden kann, daß das Argument einer Funktionsanwendung nicht von anderen im Sharing verwendet wird, dann wird es nach Auflösung alle Referenzen im Rumpf der betreffenden Funktion nicht länger benötigt. Anstatt das betreffende Datenobjekt zu löschen, kann es zur Konstruktion eines Rückgabewertes der Funktion benutzt werden. Dies entspricht einer destruktiven Modifikation des betreffenden Datenobjektes. Auf diese Weise ließe sich eine Funktion `putc` (vgl. Abschnitt 4.1) konstruieren, die die übergebene Datei tatsächlich ändert. Ihre Verwendung müßte jedoch gewissen Einschränkungen unterliegen, so daß referentielle Transparenz und Church-Rosser-Eigenschaft nicht beeinträchtigt werden. Das Beispiel aus Abschnitt 4.1 muß demgemäß ausgeschlossen sein.

Diese eingeschränkte Form der destruktiven Modifikation findet insbesondere auf einem ausgezeichneten Datenobjekt statt, der **Programm-Umgebung** (**Environment**). Die Programm-Umgebung, im folgenden kurz Umgebung genannt, repräsentiert den Zustand der von dem betreffenden Programm verwendeten Ein-/Ausgabe-Geräte. Ein Programm erhält seine Umgebung als Argument und liefert eine ggf. modifizierte Umgebung als zusätzliches Resultat zurück. Analog benötigt eine Funktion, die eine Ein-/Ausgabe-Operation repräsentiert, die Umgebung als Parameter und liefert eine veränderte Umgebung als Resultat. Durch dieses, **Environment Passing** genannte, Verfahren wird eine eindeutige Reihenfolge der Transformationen der Umgebung erreicht. Das Environment Passing kann entweder explizit durchgeführt werden, wie bei dem in der Sprache CLEAN realisierten **Uniqueness-Typing-Konzept** [SBvEP93, AP92, PvE93, AP95], oder hinter einem zusätzlichen Mechanismus versteckt werden, wie bei dem **Monaden-Konzept** von HASKELL [KP92, Wad92a, Wad92b, JW93].

4.2.2 Monaden — Ein-/Ausgabe in HASKELL

Der Begriff der Monade entstammt ursprünglich der Kategorientheorie. Für die hier betrachtete Anwendung genügt jedoch die Definition einer Monade als ein Tripel bestehend aus einem abstrakten Datentyp und zwei Funktionen **bind** und **unit**.

Der abstrakte Datentyp besteht aus zwei Komponenten. Bei der einen Komponente handelt es sich um einen gewöhnlichen HASKELL-Ausdruck, bei der anderen um die Repräsentation eines Zustands. Dies kann z.B. die Programm-Umgebung als Repräsentation des Zustands der verwendeten Ein-/Ausgabe-Geräte sein. Sämtliche Funktionen, die den Zustand modifizieren oder deren Resultat von dem Zustand abhängt, erhalten seine Repräsentation als zusätzliches Argument. Als Resultat liefern sie einen Wert des abstrakten Datentyps der Monade, bestehend aus dem eigentlichen Funktionswert und der ggf. modifizierten Zustandsrepräsentation.

Die Funktion **bind** hat die Aufgabe, Funktionsanwendungen mit dem zusätzlichen Zustandsargument zu versorgen. Zu diesem Zweck erhält **bind** zwei Funktionen und einen Wert des abstrakten Datentyps als Argumente. Sie trennt den gewöhnlichen Ausdruck von der Zustandsrepräsentation und wendet die erste Funktion auf Ausdruck und Zustand an. Danach wendet sie die zweite Funktion auf das Resultat der ersten Applikation an, also auf den Resultatsausdruck und den modifizierten

Zustand. Dies führt zu einer Sequentialisierung zustandstransformierender Operationen. Um diese Sequentialisierung in jedem Fall zu garantieren, muß **bind** die einzige Funktion sein, die auf die einzelnen Komponenten des abstrakten Datentyps zugreifen kann.

Die Funktion **unit** versteckt das zusätzliche Argument mit der Repräsentation des Zustands vor all jenen Funktionen, die nicht mit diesem in Verbindung stehen. Dazu konsumiert **unit** das zusätzliche Argument vor Ausführung einer derartigen Funktionsanwendung und konstruiert aus deren Resultat und dem konsumierten, nicht modifizierten Zustand wieder einen Wert des abstrakten Datentyps. Auf diese Weise können beliebige Funktionen in das durch Anwendungen der **bind**-Funktion gebildete Environment-Passing-Schema integriert werden.

Charakteristisch für den hier vorgestellten Ansatz ist die vollständige Kapselung der für die Modellierung eines Zustandes erforderlichen Funktionalität in den drei Komponenten der Monade. Das Environment Passing findet ausschließlich implizit statt; die Repräsentation eines Zustands existiert lediglich auf einer konzeptuellen Ebene.

Das konsequente Verstecken des Zustands vor dem Benutzer hat aber auch Nachteile. So sequentialisiert die **bind**-Funktion die Programm-Ausführung über einen globalen Zustand. Auch Operationen, die unabhängig voneinander disjunkte Teilbereiche dieses globalen Zustands betreffen, können nicht nebenläufig ausgeführt werden. Eine zufriedenstellende Lösung für die Realisierung nicht-monolithischer Ein-/Ausgabe im Rahmen des Monaden-Konzeptes fehlt bis jetzt [LJ94].

4.2.3 Uniqueness Typing — Ein-/Ausgabe in CLEAN

Für die funktionale Programmiersprache Clean wurde ein Ein-/Ausgabe-Konzept entwickelt, das sich in der Anwendung stark von dem in Haskell realisierten Monaden-Konzept unterscheidet. In Clean erfolgt die Repräsentation eines Zustands explizit. Eine Funktion, die eine Zustandstransformation durchführen soll, muß diesen Zustand ausdrücklich als Parameter erhalten und den modifizierten Zustand ebenso als Resultatswert zurückliefern.

Was einen Zustand von einem gewöhnlichen funktionalen Ausdruck unterscheidet, ist die Tatsache, daß er grundsätzlich destruktiv modifiziert wird. Die destruktive Modifikation eines Ausdrucks ist dann sicher im funktionalen Sinne, wenn garantiert werden kann, daß dieser niemals im Sharing zwischen verschiedenen Funktionsanwendungen verwendet wird (vgl. Abschnitt 4.1). Diese Eigenschaft wird **Uniqueness** genannt. Es ist jedoch i.a. nicht statisch entscheidbar, ob Argumente oder Resultate einer Funktionsanwendung die Uniqueness-Eigenschaft erfüllen [AP95].

In Clean kommt daher eine abgeschwächte Form der Uniqueness-Eigenschaft zur Anwendung, die auf **Uniqueness-Typen** beruht. Bei der Definition einer Funktion kann dem Typ eines Parameters oder Rückgabewertes durch Voranstellen des Schlüsselwortes **UNQ** das **Uniqueness-Attribut** gegeben werden. Dies hat zur Folge, daß die betreffende Funktion nur auf einen Ausdruck angewendet werden darf, dessen Typ ebenfalls das Uniqueness-Attribut besitzt. Im Rumpf einer Funktion darf ein derartiger Ausdruck höchstens einmal referenziert werden, genauer gesagt höchstens einmal in jedem Zweig eines Konditionals oder Pattern-Matching-Ausdrucks. Uniqueness-Typen sind in diesem Sinne den **Linear Types** [Wad90] ähnlich. Die korrekte Verwendung der Uniqueness-Attribute kann durch das Typsystem von Clean statisch überprüft werden. Damit wird sichergestellt, daß Ausdrücke mit dem Uniqueness-Attribut grundsätzlich destruktiv modifiziert werden dürfen.

Die gewünschte Reihenfolge bei Zustandstransformationen wird in Clean durch explizites Environment Passing angegeben. Um diese Reihenfolge bei der Ausführung eines Programmes zu garantieren, darf eine zustandstransformierende Funktion grundsätzlich erst dann ausgeführt werden, wenn alle vorhergehenden Transformationen des betreffenden Zustands auch tatsächlich stattgefunden haben, d.h. das betreffende Argument in Normalform vorliegt. Die Ausführung eines Programms

erfolgt in CLEAN jedoch nach dem Lazy-Evaluation-Prinzip, das eine derartige Einschränkung nicht kennt. Dem wird dadurch begegnet, daß das Uniqueness-Attribut eines Parameters diese erforderliche, als **Hyper Strictness** bezeichnete Eigenschaft der Funktion in dem betreffenden Parameter impliziert.

Die Programm-Umgebung wird in CLEAN durch ein vordefiniertes Datenobjekt vom Typ `UNQ WORLD` repräsentiert. Interaktive Programme benötigen dieses Datenobjekt als Parameter und liefern es in entsprechend modifizierter Form als zusätzliches Resultat zurück. Durch gewöhnliche Funktionen läßt sich diese anfänglich monolithische Umgebung in voneinander unabhängige Teilumgebungen zerlegen. Dies ermöglicht z.B. die Abspaltung des Dateisystems von der übrigen Umgebung und ebenso die erneute Abspaltung einzelner Dateien eines Dateisystems. Dabei erfolgen alle Interaktionen auf verschiedenen Partitionen der Umgebung vollkommen unabhängig voneinander. Dies bedeutet einen entscheidenden Vorteil gegenüber dem Monaden-Konzept von HASKELL.

Das explizite Environment Passing kann sich jedoch äußerst negativ auf die Lesbarkeit von Programmen auswirken. Funktionen werden durch zahlreiche zusätzliche Parameter und Rückgabewerte überfrachtet, deren einziger Zweck darin besteht, die Umgebung bzw. die jeweils benötigte Teilumgebung zur Verfügung zu stellen.

4.3 Das Klassenkonzept von SAC

4.3.1 Grundlagen

Für die Integration von Ein-/Ausgabe-Mechanismen in die Sprache SAC ergeben sich drei wesentliche Anforderungen:

1. Die funktionalen Eigenschaften von SAC, wie referentielle Transparenz oder die Church-Rosser-Eigenschaft, sollen in vollem Umfang erhalten bleiben.

2. Aus Sicht des Programmierers soll die Spezifikation von Ein-/Ausgabe-Operationen so ähnlich wie möglich zu der in der Sprache C üblichen Form erfolgen.

3. Eine Lösung sollte so allgemein sein, daß sie sich nicht ausschließlich auf Ein-/Ausgabe beschränkt, sondern allgemeine zustandsbehaftete Datenstrukturen und deren destruktive Modifikation zuläßt. Im Anwendungsgebiet von SAC kann dies z.B. im Umgang mit Datenstrukturen sinnvoll sein, die aufgrund ihrer Größe entweder gar nicht oder nur unter Inkaufnahme sehr schlechter Laufzeiten kopiert werden können.

Die dritte Anforderung schließt eine Adaption Stream-basierter Ein-/Ausgabe-Konzepte für SAC aus (vgl. Abschnitt 4.2.1). Ohnehin unterstützt SAC weder Streams noch irgendwelche anderen konzeptuell unendlichen Datenstrukturen. Die Modellierung allgemeiner zustandsbehafteter Datenstrukturen ist dagegen sowohl mit dem Uniqueness-Typing-Konzept als auch mit dem Monaden-Konzept möglich. Letzteres erfordert allerdings Funktionen höherer Ordnung in Verbindung mit einem polymorphen Typsystem (**bind**-Funktion). Auf beides wird bei SAC verzichtet, um ein besseres Laufzeitverhalten von Programmen zu erreichen. Aufgrund dessen Bedeutung in dem für SAC angestrebten Anwendungsgebiet numerischer Algorithmen scheidet daher auch die Übernahme des Monaden-Konzeptes aus. Das Uniqueness-Typing-Konzept dagegen erfordert lediglich die vergleichsweise geringfügige Erweiterung des Typsystems von SAC um das Uniqueness-Attribut. Dies macht es zu dem für SAC am besten geeigneten Ansatz zur Integration von Ein-/Ausgabe und anderen zustandsbehafteten Operationen.

Das Uniqueness-Typing-Konzept erfüllt die erste und die dritte der oben genannten Anforderungen. In diesem Rahmen auch die zweite Anforderung, nämlich die der weitgehenden Ähnlichkeit zu C, zu erfüllen, ist das Ziel der Adaption dieses Konzeptes für SAC. Daraus ergeben sich zwangsläufig deutliche Abweichungen von dem Vorbild CLEAN.

Eine Grundlage des Uniqueness-Typing-Konzeptes besteht in der Unterscheidung zwischen Typen mit und ohne Uniqueness-Attribut. In CLEAN geschieht dies durch das explizite Typ-Attribut **UNQ**. SAC dagegen bedient sich dafür Begriffen und Mechanismen aus der Welt der objekt-orientierten Programmierung. Dies wird durch die Beobachtung motiviert, daß Objekte im Sinne des objekt-orientierten Paradigmas besonders typische Beispiele für die Repräsentation von Zuständen sind. Uniqueness-Typen werden in SAC in Form von **Klassen** eingeführt. Wie in der objekt-orientierten Programmierung besteht eine Klasse in SAC aus einer Typ-Definition und einer Menge von Funktionen. Die konkrete Definition des Typs ist außerhalb der Klasse unbekannt. Dadurch bilden die Funktionen der Klasse die einzige Möglichkeit zum Erzeugen, Modifizieren und Löschen von Instanzen des Klassentyps. Eine solche Instanz heißt Objekt der Klasse. Weitergehende objekt-orientierte Mechanismen, wie z.B. Vererbung, sind für den hier verfolgten Zweck nicht erforderlich.

Die Kapselung von Uniqueness-Typen in Klassen und die damit verbundene Propagierung einer objekt-orientierten Sichtweise soll das Bewußtsein des Programmierers für den unterschiedlichen Charakter von Ausdrücken mit und ohne Uniqueness-Attribut verstärken. Während die einen rein funktional für einen Wert stehen, repräsentieren die anderen einen veränderbaren Zustand.

Unabhängig von der Art und Weise, wie Uniqueness-Typen definiert werden, bleiben die negativen Auswirkungen des expliziten Environment Passing jedoch bestehen (vgl. Abschnitt 4.2.3).

- Die Modifikation eines Zustands `A` durch eine Funktion `modify` läßt sich ausschließlich in der Form `A = modify(A);` beschreiben. Diese Notation hat zwei Nachteile. Zum einen erlaubt sie die implizite Umbenennung des Zustands durch Schreiben von `B = modify(A);`, wodurch die Lesbarkeit des Programm-Codes verringert wird. Zum anderen erzwingt sie die explizite Angabe eines Rückgabewertes, der zumindest von einem pragmatischen Standpunkt aus betrachtet überflüssig ist.

- Alle Zustände, die innerhalb einer bestimmten Hierarchie von Funktionsanwendungen inspiziert oder modifiziert werden, müssen sämtlichen Funktionen dieser Aufruf-Hierarchie als Parameter übergeben und von jeder einzelnen als Resultat zurückgeliefert werden. Davon sind auch solche Funktionen betroffen, die selber keine unmittelbaren Operationen auf den Zuständen ausführen. Dadurch wird z.B. die temporäre Einführung von Ein-/Ausgabe-Operationen zur Fehlersuche während der Programm-Entwicklung außerordentlich erschwert.

Um diesen für den Programmierer unangenehmen Konsequenzen des expliziten Environment Passing zu begegnen und gleichzeitig eine stärker an C orientierte Programm-Notation zu ermöglichen, wird das Uniqueness-Typing-Konzept um zwei neue Mechanismen ergänzt. Ein **Call-by-Reference-Mechanismus** erlaubt die Spezifikation von zustandstransformierenden Funktionen ohne die explizite Rückgabe des modifizierten Zustands. **Globale Objekte** ermöglichen die Deklaration von Zuständen, die im Rumpf jeder Funktion zur Verfügung stehen, ohne dieser explizit als Parameter übergeben worden zu sein.

Auf den ersten Blick scheinen Call-by-Reference-Mechanismus und globale Objekte vollkommen unvereinbar mit dem funktionalen Paradigma zu sein. An dieser Stelle werden sie jedoch als rein syntaktische Kurzschreibweisen für ein entsprechendes explizites Environment Passing der betroffenen Funktionen definiert. So erlauben sie eine kurze und prägnante Spezifikation von Operationen auf Zuständen, ähnlich der imperativer Sprachen, ohne dabei referentielle Transparenz und die Church-Rosser-Eigenschaft in Frage zu stellen.

Die Integration von Mechanismen zur Ein-/Ausgabe in die Sprache SAC ist die wichtigste Anwendung des Klassen-Konzeptes. Sie dient daher als Grundlage einer Fallstudie, mit deren Hilfe die Möglichkeiten des hier vorgestellten Klassen-Konzeptes illustriert werden sollen. Aufbauend auf einer Basisversion, die ausschließlich Klassen und Objekte verwendet (Abbschnitt 4.3.2), werden nacheinander der Call-by-Reference-Mechanismus (Abbschnitt 4.3.3) und globale Objekte (Abbschnitt 4.3.4) hinzugefügt. Dabei sind alle späteren Versionen der Fallstudie semantisch äquivalent zur Basisversion. Diese definiert so deren Bedeutung und damit insbesondere die Bedeutung der neuen Mechanismen.

4.3.2 Klassen und Objekte

In vielen imperativen Sprachen mit objekt-orientierten Erweiterungen, wie z.B. C++ [Str91], werden Klassen durch eine Erweiterung des Typsystems eingeführt. Dabei werden konventionelle Typen und Funktionen in besonderen Strukturen oder Records zusammengefaßt. Dieser Weg wird bei SAC jedoch nicht verfolgt. Einerseits kennt das Typsystem von SAC derzeit keine Strukturen, andererseits bietet die Sprache einen anderen Mechanismus, der nahezu alle der an eine Klasse gestellten Anforderungen bereits erfüllt: das Modul-Konzept. Ein Modul verbindet Typen mit Funktionen zu einer Einheit. Wird ein Typ implizit exportiert, bilden die Funktionen des Moduls die einzige Schnittstelle für seine Verwendung.

Klassen stellen daher in SAC eine besondere Form von Modulen dar. Eine Klasse besteht folglich aus einer Klassen-Implementierung und einer Klassen-Deklaration. Klassen-Deklarationen beginnen mit dem Schlüsselwort **ClassDec** anstelle von **ModuleDec**. Danach folgen **import**-Anweisungen und Deklarationen von impliziten und expliziten Typen sowie Funktionen in der bei Modul-Deklarationen üblichen Form. Der wesentliche Unterschied zu einem Modul besteht darin, daß die Klasse darüberhinaus einen weiteren impliziten Typ zur Verfügung stellt, den **Klassentyp**. Dieser trägt denselben Namen wie die Klasse selbst. Klassentypen sind genau die Uniqueness-Typen in SAC.

Abbildung 4.1 illustriert die Anwendung von Klassen am Beispiel der Basisversion der angesprochenen Fallstudie. Diese demonstriert die Realisierung von Ein-/Ausgabe-Mechanismen mit Hilfe des Klassen-Konzeptes. Sie besteht aus der Deklaration der Klasse `TermFile` und einem einfachen Programm, das die Verwendung von `TermFile` zeigt. Die hier nur auszugsweise wiedergegebene Standard-Klasse `TermFile` ermöglicht SAC-Programmen die Interaktion mit einem Terminal. Wie in C wird dabei zwischen den drei Ein-/Ausgabe-Kanälen `stdin`, `stdout` und `stderr` unterschieden.

Die Klasse `TermFile` importiert ihrerseits die Klassen `World` und `String`. Letztere stellt den Datentyp `string` und darauf operierende Funktionen zur Verfügung, die für die hier betrachtete Fallstudie jedoch nicht von Bedeutung sind. Der Klasse `World` hingegen kommt für die Formulierung interaktiver Programme eine zentrale Bedeutung zu. Ein Objekt der Klasse `World` repräsentiert die Programm-Umgebung. Die von `TermFile` zur Verfügung gestellten Funktionen sind typische Vertreter von drei Arten von Klassen-Funktionen. Eine Konstruktor-Funktion wie `open_stdout` generiert ein Objekt der betreffenden Klasse. In diesem Fall repräsentiert das generierte Objekt den Standard-Ausgabe-Kanal `stdout`. Komplementär dazu konsumiert bzw. löscht die Destruktor-Funktion `close_stdout` das betreffende Objekt. Die Funktion `fprintstring` repräsentiert die dritte Art von Klassen-Funktionen. Konzeptuell konsumiert sie ein Objekt der Klasse `TermFile` und einen Wert des Typs `string` und liefert ein neues Objekt der Klasse `TermFile`. Aufgrund der Uniqueness-Eigenschaft von Objekten kann die betreffende Datenstruktur jedoch in jedem Fall destruktiv modifiziert werden. Daher läßt sich `fprintstring` von einem pragmatischen Standpunkt aus als Modifikator-Funktion bezeichnen. Die Verwendung eines identischen Variablennamens für Argument und dazugehörigen Rückgabewert verdeutlicht diese Vorstellung.

```
ClassDec TermFile :
import World:  all;
import String: all;
own: {
functions: TermFile, World open_stdout(World);
           World           close_stdout(TermFile, World);
           TermFile        fprintstring(TermFile, string);
}


import TermFile: all;

TermFile printline(TermFile stdout, string message)
{
  stdout = fprintstring(stdout, message);
  stdout = fprintstring(stdout, "\n");
  return(stdout);
}

TermFile header(TermFile stdout)
{
  stdout = printline(stdout, "This is SAC !");
  return(stdout);
}

int, World main(World world)
{
  stdout, world = open_stdout(world);
  stdout        = header(stdout);
  stdout        = fprintstring(stdout, "Hello World.\n");
  world         = close_stdout(stdout, world);
  return(0, world);
}
```

Abbildung 4.1: Fallstudie: Ein-/Ausgabe auf der Basis von Klassen und Objekten

Die einzige Möglichkeit, ein Objekt der Klasse `World` zu generieren, besteht in der Spezifikation der Start-Funktion `main`. Während diese bei nicht mit ihrer Umwelt interagierenden Programmen die Signatur `int main()` besitzt, ändert sich diese bei interaktiven Programmen zu `int, World main(World world)`. Das bedeutet, daß ein interaktives Programm seine Umgebung als Parameter erfordert und neben dem eigentlichen Resultatswert die veränderte Umgebung zurückliefert. Auf diese Weise wird u.a. sichergestellt, daß es innerhalb eines Programmes immer höchstens ein Objekt der Klasse `World` gibt.

Die der Start-Funktion als Argument übergebene Umgebung kann in disjunkte Teilumgebungen (Sub-Environments) zerlegt werden. Dies ermöglicht die nebenläufige Ausführung voneinander unabhängiger Ein-/Ausgabe-Operationen (nicht-monolithische Ein-/Ausgabe). Die Anwendung der Funktion `open_stdout` beispielsweise resultiert in einem Objekt der Klasse `TermFile`, das den Standard-Ausgabe-Kanal `stdout` repräsentiert, und einer modifizierten Umgebung, die den Standard-Ausgabe-Kanal nicht länger umfaßt.

Die Generierung eines Objektes der Klasse `TermFile`, das den Standard-Ausgabe-Kanal repräsentiert, ist die Voraussetzung für Ausgabe-Operationen mit Hilfe der Funktion `fprintstring`. Das Objekt der Klasse `World` ist für die eigentliche Ausgabe-Operation dann nicht mehr erforderlich.

Da Modifikationen der Umgebung Teil des Resultats einer Programm-Ausführung sind, wird das als Argument übergebene Objekt der Klasse `World` von der Start-Funktion neben dem eigentlichen Resultatswert zurückgeliefert. Dies ist jedoch nur dann sinnvoll, wenn es an dieser Stelle auch tatsächlich die vollständige Programm-Umgebung repräsentiert. Zu diesem Zweck müssen sämtliche Partitionierungen der Umgebung vorher rückgängig gemacht werden. Im Fall des Standard-Ausgabe-Kanals dient dazu eine Anwendung der Funktion `close_stdout`.

4.3.3 Der Call-by-Reference-Mechanismus

Die einzige Möglichkeit zur Spezifikation einer Objekt-Modifikation besteht in einer Funktion, die konzeptuell ein Objekt konsumiert und ein neues Objekt derselben Klasse produziert. Dies kann z.B. in der Form `A = modify(A);` geschrieben werden. Zumindest bei einer pragmatischen Sichtweise erscheint die explizite Angabe eines Rückgabewertes dabei überflüssig. Daher erlaubt Sac in derartigen Fällen eine andere Notation, die in ihrer Verwendung einem Call-by-Reference-Parameterübergabe-Mechanismus ähnelt, wie er bei imperativen Sprachen üblich ist.

$$
\begin{array}{lcl}
\mathit{ReturnList'} & \Rightarrow & \mathit{Type} \left[\ , \ \mathit{Type} \right]^{*} \\
 & | & \textbf{void} \\
 & & \\
\mathit{ParamList'} & \Rightarrow & (\left[\mathit{Type} \left[\ \& \ \right] \mathit{Id} \left[\ , \ \mathit{Type} \left[\ \& \ \right] \mathit{Id} \right]^{*} \right]) \\
 & & \\
\mathit{ExprBlock'} & \Rightarrow & \{ \ \left[\mathit{VarDec} \right]^{*} \left[\mathit{Instruction} \right]^{*} \left[\mathit{Return} \right] \ \} \\
 & & \\
\mathit{Instruction'} & \Rightarrow & ... \\
 & | & \mathit{FunCall} \ ; \\
 & & \\
\mathit{FunCall} & \Rightarrow & \mathit{Id} \ (\left[\mathit{Expr} \left[\ , \mathit{Expr} \right]^{*} \right])
\end{array}
$$

Abbildung 4.2: Erweiterung der Syntax für den Call-by-Reference-Mechanismus

Abbildung 4.2 gibt einen Überblick über die erforderlichen syntaktischen Erweiterungen. Referenzparameter werden von gewöhnlichen Parametern durch das Symbol „&“ in der Parameterliste einer Funktion unterschieden. Es gibt an, daß der betreffende Parameter von der Funktion implizit zurückgeliefert wird, d.h. ein entsprechender expliziter Rückgabewert sowohl in der Rückgabetypliste als auch der `return`-Anweisung entfällt. Um die Spezifikation von Funktionen zu erlauben, die keinerlei explizite Rückgabewerte aufweisen, wird die Syntax von Sac um **void-Funktionen** in der aus C bekannten Weise erweitert. Dies schließt auch den möglichen Wegfall der `return`-Anweisung ein. Analog dazu werden Funktionsanwendungen erlaubt, bei der keiner Variablen ein Resultatswert zugewiesen wird (*FunCall*).

Durch konsequente Verwendung der als Call-by-Reference-Mechanismus bezeichneten neuen Notation entsteht die in Abbildung 4.3 wiedergegebene Version der Fallstudie. Die Verwendung des Call-by-Reference-Mechanismus läßt sich u.a. am Beispiel der Funktion `printline` beobachten. Ihr erster Parameter `TermFile stdout` wird durch den Referenzoperator „&“ als Referenzparameter

```
ClassDec TermFile :
import World:  all;
import String: all;
own: {
functions: TermFile open_stdout(World &);
           void     close_stdout(TermFile, World &);
           void     fprintstring(TermFile &, string);
}


import TermFile: all;

void printline(TermFile &stdout, string message)
{
  fprintstring(stdout, message);
  fprintstring(stdout, "\n");
}

void header(TermFile &stdout)
{
  printline(stdout, "This is SAC !");
}

int main(World &world)
{
  stdout = open_stdout(world);
  header(stdout);
  fprintstring(stdout, "Hello World.\n");
  close_stdout(stdout, world);
  return(0);
}
```

Abbildung 4.3: Fallstudie: Der Call-by-Reference-Mechanismus

gekennzeichnet. Da er als solcher implizit zurückgeliefert wird, kann und muß seine explizite Rückgabe (vgl. Basisversion der Fallstudie in Abbildung 4.1) entfallen. Weil `stdout` der einzige Rückgabewert in der ursprünglichen Version der Funktion `printline` ist, sind Rückgabetypliste und `return`-Anweisung folglich leer. Dies wird syntaktisch durch den Rückgabetyp `void` und das Entfallen der `return`-Anweisung ausgedrückt. Entsprechend verändern sich auch die Anwendungen von `printline` gegenüber der Basisversion. Dies kann im Rumpf der Funktion `header` beobachtet werden. Da das an `printline` übergebene Objekt `stdout` jetzt implizit zurückgeliefert wird, entfällt die Bindung an eine neue Variable.

Die Fallstudie demonstriert, daß der Call-by-Reference-Mechanismus einen entscheidenden Schritt zur Erreichung des Design-Ziels einer möglichst C-ähnlichen Notation für Ein-/Ausgabe-Operationen darstellt. Besonders auffällig ist dies beispielsweise im Rumpf der Funktion `printline` erkennbar. Die Spezifikation einer komplexen Ausgabe-Operation, die sich ihrerseits aus wiederholten Anwendungen der primitiven Ausgabe-Funktion `fprintstring` zusammensetzt, erfolgt scheinbar als Sequenz von Anweisungen, bei denen keinerlei Resultatswert berechnet wird.

An dieser Stelle wird bereits der ambivalente Charakter des Klassen-Konzeptes von SAC deutlich. Auf der einen Seite kann der Programmierer bei Verwendung einer Klasse Operationen auf Zuständen weitgehend wie in einer imperativen Sprache spezifizieren, nämlich als Sequenz von Seiteneffekte ausführenden Anweisungen. Auf der anderen Seite erlaubt die Definition des Call-by-Reference-Mechanismus als Kurzschreibweise für ein vollständiges, explizites Environment Passing eine eindeutige rein funktionale Interpretation der resultierenden Programme. Das Uniqueness-Typing-Konzept garantiert in Verbindung mit einem Applicative-Order-Ausführungsmechanismus schließlich, daß beide Sichtweisen zu identischen Resultaten führen.

4.3.4 Globale Objekte

Eine Funktion kann grundsätzlich nur auf solchen Objekten Operationen ausführen, die sie entweder als Argument übergeben bekommt oder selber generiert. Es gibt jedoch vielfach Zustände, die einen globalen Charakter haben und potentiell in jeder Funktion zur Verfügung stehen sollen. Die Standard-Ein-/Ausgabe-Kanäle eines Terminals sind Beispiele für derartige Zustände. Auch die sie repräsentierenden Objekte müssen bis jetzt jeder Funktion explizit übergeben werden (vgl. bisherige Versionen der Fallstudie in den Abbildungen 4.1 und 4.3). Es vereinfacht jedoch die Entwicklung von Programmen erheblich, wenn derartige Objekte im Rumpf jeder Funktion verwendet werden können, ohne explizit als Argument übergeben zu werden. Aus diesem Grund verfügt SAC über **globale Objekte**, die sich genau durch diese Eigenschaft auszeichnen. Die dazu erforderlichen syntaktischen Erweiterungen beschreibt Abbildung 4.4.

$$\begin{array}{lll} Definitions' & \Rightarrow & \left[\, TypeDef \,\right]^{*} \left[\, ObjDef \,\right]^{*} \left[\, FunDef \,\right]^{*} \\ ObjDef & \Rightarrow & \textbf{objdef}\ \ Type\ Id\ =\ Expr\ ; \end{array}$$

Abbildung 4.4: Erweiterung der Syntax für globale Objekte

Die Definition globaler Objekte erfolgt in einem zusätzlichen Programm-Abschnitt zwischen den Typ- und den Funktionsdefinitionen. Dazu dient das neue Schlüsselwort **objdef**. Es definiert ein globales Objekt vom Typ *Type* mit dem Namen *Id*, das durch den Ausdruck *Expr* initialisiert bzw. generiert wird. Bei *Type* muß es sich um einen Klassentyp handeln. Der Initialisierungsausdruck besteht i.a. aus dem Aufruf einer Konstruktor-Funktion der jeweiligen Klasse. Ein auf diese Weise definiertes globales Objekt steht nun in den Rümpfen sämtlicher Funktionen zur Verfügung, so als würde es explizit als Parameter übergeben werden. Ein globales Objekt wird zu Beginn der Programm-Ausführung generiert und erst unmittelbar vor der Terminierung wieder gelöscht. Es kann durch entsprechende Funktionen wohl modifiziert, jedoch nicht gelöscht werden.

In Abbildung 4.5 wird die Verwendung globaler Objekte im Rahmen der Fallstudie demonstriert. Die Programm-Umgebung wird jetzt durch das globale Objekt `world` der Klasse `World` repräsentiert. Dadurch erhält die Startfunktion wieder ihre ursprüngliche, einheitliche Signatur `int main()`. Durch die Modellierung als globales Objekt ist nach wie vor gewährleistet, daß ein mit seiner Umgebung interagierendes Programm genau eine Repräsentation dieser Umgebung besitzt. Aufgrund der Anweisung `import World: all;` in der Klassen-Deklaration von `TermFile` steht das globale Objekt `world` in jedem Programm, das die Klasse `TermFile` importiert, zur Verfügung.

Der Standard-Ausgabe-Kanal wird jetzt durch das globale Objekt `stdout` repräsentiert. Es wird mit Hilfe der Funktion `open_stdout` initialisiert. Diese erhält dazu das globale Objekt `world` als Argument. Als globales Objekt steht `stdout` im Rumpf jeder Funktion zur Verfügung, so auch im Rumpf von `printline`. Die explizite Übergabe als Parameter ist nicht mehr notwendig. Die

```
ClassDec TermFile :
import World:  all;
import String: all;
own: {
functions: TermFile open_stdout(World &);
           void     fprintstring(TermFile &, string);
}


import TermFile: all;

objdef TermFile stdout = open_stdout(world);

void printline(string message)
{
  fprintstring(stdout, message);
  fprintstring(stdout, "\n");
}

void header()
{
  printline("This is SAC !");
}

int main()
{
  header();
  fprintstring(stdout, "Hello World.\n");
  return(0);
}
```

Abbildung 4.5: Fallstudie: Globale Objekte

unmittelbare Auswirkung dessen wird an der Funktion `header` deutlich. Aus ihrer Definition ist `stdout` vollständig verschwunden, da es im Rumpf nicht direkt verwendet wird und auch die dort angewandte Funktion `printline` es nun nicht mehr als Argument erfordert.

Analog zum Call-by-Reference-Mechanismus handelt es sich auch bei den globalen Objekten wiederum ausschließlich um eine weitere Kurzschreibweise für explizites Environment Passing. Die Definition eines globalen Objekts ist gleichbedeutend mit einer entsprechenden lokalen Objekt-Definition am Beginn der Startfunktion `main` und der expliziten Übergabe dieses Objekts als Argument an alle Funktionen, die es benötigen. Dabei benötigt eine Funktion alle diejenigen globalen Objekte, die entweder direkt in ihrem Rumpf vorkommen oder rekursiv von einer anderen Funktion benötigt werden, die in ihrem Rumpf angewandt wird.

Die Definition globaler Objekte analog zu Typen und Funktionen legt eine Erweiterung des Modul-Konzeptes nahe. Ein globales Objekt, das in einer Modul- oder Klassen-Implementierung definiert ist, soll von dem betreffenden Modul bzw. der betreffenden Klasse auch exportiert werden können. Abbildung 4.6 zeigt die dafür notwendigen syntaktischen Erweiterungen von Modul- und Klassen-Deklarationen sowie der **import**-Anweisung. In beiden Fällen wird ein zusätzlicher Ab-

$$
\begin{array}{lcl}
\mathit{Declarations'} & \Rightarrow & \{\ [\ \mathit{ITypeDec}\ ]\ [\ \mathit{ETypeDec}\ ]\ [\ \mathit{ObjDec}\ ]\ [\ \mathit{FunDec}\ ]\ \} \\
\mathit{ObjDec} & \Rightarrow & \textbf{global objects}\ :\ [\ \mathit{Type}\ \mathit{Id}\ ;\ ]^{*} \\
\mathit{ImportList'} & \Rightarrow & \{\ [\ \mathit{ITypeImp}\ ]\ [\ \mathit{ETypeImp}\ ]\ [\ \mathit{ObjImp}\ ]\ [\ \mathit{FunImp}\ ]\ \} \\
\mathit{ObjImp} & \Rightarrow & \textbf{global objects}\ :\ \mathit{Id}\ [\ ,\ \mathit{Id}\ ]^{*}\ ;
\end{array}
$$

Abbildung 4.6: Erweiterung der Syntax zum Im- und Exportieren globaler Objekte

schnitt eingeführt, der mit dem Schlüsselwort **global objects** beginnt. In diesem Abschnitt können die zum Export vorgesehenen globalen Objekte unter Angabe ihres Typs deklariert werden. Im Falle selektiven Imports können an dieser Stelle die Namen zu importierender globaler Objekte aufgelistet werden. Auf diese Weise lassen sich globale Objekte genauso Im- und Exportieren wie Funktionen.

```
ClassDec TermFile :
import World:  all;
import String: all;
own: {
global objects: TermFile stdout;
functions: void fprintstring(TermFile &, string);
}
   ...
```

Abbildung 4.7: Fallstudie: Import eines globalen Objektes

Bei Verlagerung der Definition des globalen Objekts `stdout` in die Implementierung der Klasse `TermFile` sind zwei grundsätzliche Varianten zur Weiterentwicklung der Fallstudie denkbar. Abbildung 4.7 zeigt die erste davon. Da das eigentliche Programm keinerlei Veränderung unterworfen ist, enthält die Abbildung lediglich die erweiterte Deklaration der Klasse `TermFile`. Das globale Objekt `stdout` wird von der Klasse `TermFile` exportiert. Durch die Anweisung `import TermFile: all;` wird es von dem hier betrachteten Programm importiert. Anschließend kann es in diesem genauso verwendet werden, als wäre es im Programm selbst definiert. Die Konstruktor-Funktion `open_stdout` wird nun außerhalb der Klasse `TermFile` nicht mehr benötigt und folglich von dieser auch nicht länger exportiert.

Die zweite Variante zeigt Abbildung 4.8. Dabei exportiert die Klasse `TermFile` anstelle des globalen Objekts `stdout` und der Funktion `fprintstring` lediglich eine Funktion `printstring`. Alle Anwendungen von `fprintstring` werden durch entsprechende Anwendungen von `printstring` ersetzt. Diese Funktion verwendet ihrerseits implizit das globale Objekt `stdout`, das dabei vollständig in der Klasse `TermFile` versteckt wird. Bei dieser Variante ist es die Aufgabe des Modul-Systems, die implizite Verwendung globaler Objekte durch Funktionen über die Grenzen einzelner Module hinaus zumindest für den Compiler transparent zu machen.

Auszüge aus den der Integration von Ein-/Ausgabe-Mechanismen dienenden Standard-Klassen und -Modulen befinden sich in Anhang B. Die dort angegebenen Klassen `World` und `TermFile` weichen lediglich geringfügig von den im Rahmen der Fallstudie beschriebenen Versionen ab. Zusätzlich erlaubt die Standard-Klasse `File` Interaktionen mit dem Dateisystem und das Standard-Modul `SysErr` die Handhabung daraus resultierender Laufzeitfehler.

```
ClassDec TermFile :
import World:  all;
import String: all;
own: {
functions: void printstring(string);
}


import TermFile: all;

void printline(string message)
{
  printstring(message);
  printstring("\n");
}

void header()
{
  printline("This is SAC !");
}

int main()
{
  header();
  printstring("Hello World.\n");
  return(0);
}
```

Abbildung 4.8: Fallstudie: Implizite Verwendung eines globalen Objektes

4.3.5 Klassen-Implementierungen in SAC

Bei den der Integration von Ein-/Ausgabe dienenden Klassen handelt es sich um Standard-Klassen, die einen Bestandteil der Sprache SAC ausmachen. Sie stellen jedoch lediglich eine spezielle Anwendung des Klassen-Konzeptes dar. Darüberhinaus lassen sich beliebige weitere zustandsbehaftete Datenstrukturen und deren Modifikation mit Hilfe des Klassen-Konzeptes modellieren. Zu diesem Zweck können auch Klassen-Implementierungen in der Sprache SAC spezifiziert werden.

Abbildung 4.9 zeigt die geringen syntaktischen Unterschiede zwischen Klassen und Modulen. Eine Klassen-Implementierung wird durch das Schlüsselwort **Class** anstelle von **Module** eingeleitet. Sie enthält nach dem Schlüsselwort **classtype** die Definition des Klassentyps. Dieser trägt per Definition denselben Namen wie die Klasse selbst. Der Klassentyp kann jeder beliebige SAC-Datentyp sein. Danach folgen **import**-Anweisungen sowie die Definitionen von Typen, globalen Objekten und Funktionen wie bei einer Modul-Implementierung. Ein Beispiel für eine in SAC implementierte Klasse befindet sich in Anhang A.

Um Funktionen auf dem Klassentyp definieren zu können, ist es erforderlich, Ausdrücke zwischen dem Klassentyp und seinem Basistyp in beiden Richtungen konvertieren zu können. Bei gewöhnlichen benutzerdefinierten Typen dient dazu der Cast-Ausdruck. Dieses Verfahren würde jedoch der

besonderen Bedeutung einer Konvertierung zwischen Ausdrücken mit und ohne Uniqueness-Attribut nicht gerecht werden. Stattdessen stehen in jeder Klassen-Implementierung zwei generische Konvertierungsfunktionen zur Verfügung. Dabei handelt es sich um

$$\begin{array}{rrl} & \mathit{basetype} & \texttt{from\_}\mathit{class}(\mathit{class}) \\ \text{und} & \mathit{class} & \texttt{to\_}\mathit{class}(\mathit{basetype}) \quad . \end{array}$$

Wie bei einem Modul definiert die Klassen-Deklaration die Export-Schnittstelle der Klasse. Alle in der Klassen-Implementierung definierten Typen, globalen Objekte und Funktionen können exportiert werden. Der Klassentyp wird per Definition exportiert und darf daher nicht zusätzlich als expliziter oder impliziter Typ deklariert werden. Die generischen Konvertierungsfunktionen dürfen nicht exportiert werden.

$$\begin{array}{lcl} \mathit{ModuleImp}' & \Rightarrow & \textbf{Module}\ \mathit{Id}: \ \mathit{Imports}\ \mathit{Definitions}' \\ & | & \textbf{Class}\ \mathit{Id}: \ \mathit{ClassTypeDef}\ \mathit{Imports}\ \mathit{Definitions}' \\ & & \\ \mathit{ClassTypeDef} & \Rightarrow & \textbf{classtype}\ \mathit{Type}\ ; \\ & & \\ \mathit{ModuleDec}' & \Rightarrow & \textbf{ModuleDec}\ \mathit{Id}\ :\ \mathit{Imports}\ \textbf{own}: \ \mathit{Declarations}' \\ & | & \textbf{ClassDec}\ \mathit{Id}\ :\ \mathit{Imports}\ \textbf{own}: \ \mathit{Declarations}' \end{array}$$

Abbildung 4.9: Syntax von Klassen-Implementierung und -Deklaration

Kapitel 5

Schnittstellen

Das vorliegende Kapitel beschreibt die von SAC gegenüber anderen Programmiersprachen bereitgestellte Schnittstelle. Nach einer kurzen Einführung in Aufgaben und Ziele von Schnittstellen zwischen Programmiersprachen werden externe Module und Klassen vorgestellt. Anschließend werden die daraus resultierenden Möglichkeiten zur Verwendung von in der Sprache C definierten Symbolen im Rahmen eines SAC-Programms erläutert.

5.1 Aufgaben und Ziele

Die Wiederverwendbarkeit von Code hat entscheidenden Einfluß auf die Effizienz der Software-Entwicklung. Je höher der Anteil bestehenden Codes in einem neuen Anwendungsprogramm ist, desto geringer fallen Entwicklungszeit und -kosten aus. Die Umstellung auf eine neue Programmiersprache ist aus diesem Grund i.a. mit sehr hohen Kosten verbunden, da Programme zumindest anfangs vollständig neu entwickelt werden müssen. Bestehende Funktionsbibliotheken werden wertlos und müssen in der neuen Sprache re-implementiert werden. Diese Kosten führen dazu, daß im Rahmen kommerzieller Software-Entwicklung eher auf das positive Potential einer neuen Programmiersprache verzichtet wird.

Bei der Entwicklung großer Anwendungsprogramme tritt häufig der Fall auf, daß keine Programmiersprache für die Implementierung aller Teilaspekte optimal geeignet ist. Wird trotzdem eine einheitliche Implementierungssprache verwendet, sind negative Auswirkungen unvermeidbar. Dies kann sowohl die Effizienz der Programm-Entwicklung als auch die Qualität der fertigen Anwendung betreffen.

Im Rahmen der Entwicklung einer neuen Programmiersprache stellt sich das Problem, daß sinnvolle Anwendungsprogramme ein Mindestmaß an Funktionalität benötigen, die gewöhnlich nicht Teil des Sprachkerns ist. Dabei kann es sich z.B. um Ein-/Ausgabe-Operationen oder mathematische Funktionen handeln. Dies führt zur Aufspaltung der Entwicklungskapazität zwischen der eigentlichen Weiterentwicklung der Sprache selbst und der Schaffung umfangreicher Standard-Bibliotheken zur Erstellung von Anwendungsprogrammen.

Schnittstellen zwischen Programmiersprachen stellen eine mögliche Lösung für die oben genannten Probleme dar. Sie erlauben die Implementierung unterschiedlicher Teile eines Anwendungsprogramms in verschiedenen Sprachen. Diese als **Mixed Language Programming** bezeichnete Technik bietet eine Reihe von Vorteilen. So kann für jeden Teil eines Anwendungsprogramms die jeweils

am besten geeignete Sprache zur Implementierung verwendet werden. Bei der Entwicklung eines Anwendungsprogramms in einer neuen Sprache stehen individuelle Funktionsbibliotheken ebenso weiter zur Verfügung wie die meist umfangreiche Funktionalität von Standardbibliotheken etablierter Sprachen.

Voraussetzung für die Nutzung dieser Möglichkeiten ist ein hinreichendes Maß an Flexibilität bei der Gestaltung der Schnittstelle. Das vorrangige Ziel muß es daher sein, möglichst viele Elemente einer Sprache auf die jeweils andere abzubilden. Dazu zählt z.B. die Verwendung von Datenstrukturen einer Sprache durch Funktionen einer anderen. Je höher der Anteil dieser Elemente ist, desto flexibler ist die Schnittstelle, und umso besser erfüllt sie die oben genannten Aufgaben. Dieses zu gewährleisten, wird jedoch um so schwerer je stärker sich die betroffenen Programmiersprachen voneinander unterscheiden.

5.2 Externe Module und Klassen

5.2.1 Grundlagen

Auf der Grundlage des Modul- und Klassen-Konzeptes stellt SAC eine Schnittstelle zu anderen Programmiersprachen bereit. Diese ist uni-direktional, d.h. ihre Aufgabe besteht ausschließlich darin, die Funktionalität anderer Sprachen für die Programmierung in SAC zu erschließen[1]. Zu diesem Zweck wird das Modul- und Klassen-Konzept um **externe Module** bzw. **externe Klassen** erweitert. Bei diesen erfolgt die Implementierung in einer von SAC verschiedenen Sprache. Dabei werden alle Sprachen unterstützt, die zu dem in der Sprache C üblichen Link-Mechanismus kompatibel sind. Dies sind neben C selbst z.B. FORTRAN oder auch PASCAL. Zur Vereinfachung wird im folgenden bei einem externen Modul jedoch grundsätzlich eine C-Implementierung unterstellt. Die jeweilige Deklaration enthält die vollständige Beschreibung des Moduls bzw. der Klasse sowohl für den Programmierer als auch den SAC-Compiler. In ihrer Verwendung innerhalb eines SAC-Programms unterscheiden sich externe Module und Klassen nicht von solchen, die in SAC implementiert sind. Letztere werden im folgenden kurz als SAC-Module bezeichnet.

Die Syntax von Modul- und Klassen-Deklarationen wird im Hinblick auf externe Module und Klassen erneut erweitert. Abbildung 5.1 gibt einen vollständigen Überblick darüber. Das Schlüsselwort **external** kennzeichnet ein Modul bzw. eine Klasse als extern. Speziell für diese werden die Deklarationen einzelner Typen, Funktionen und globaler Objekte um **Pragmas** erweitert. Dabei handelt es sich um zusätzliche Beschreibungen des jeweiligen Symbols. Diese erweitern die Menge der von C nach SAC abbildbaren Programme und erhöhen damit die Flexibilität der Schnittstelle. Nicht jedes Pragma findet dabei für jeden Symboltyp Verwendung. Undefinierte Pragmas sind wirkungslos. Dasselbe gilt für Pragmas in der Deklaration eines SAC-Moduls.

5.2.2 SAC-Typen in C-Funktionen

Die hier betrachtete Schnittstelle ist darauf ausgerichtet, die Funktionalität von C für die Programmierung in SAC zu nutzen. Eine Voraussetzung dafür ist jedoch, daß in der Sprache C implementierte Funktionen auf SAC-Datenobjekte angewandt werden können. Zu diesem Zweck ist eine Abbildung von SAC-Typen auf äquivalente C-Typen erforderlich.

[1] Eine Schnittstelle in der umgekehrten Richtung wird indirekt durch die Verwendung von C als Zwischensprache bei der Übersetzung von SAC-Programmen geschaffen.

$$
\begin{array}{lcl}
ModuleDec'' & \Rightarrow & \textbf{ModuleDec} \ [\ \textbf{external}\ ]\ Id\ :\ Imports\ \textbf{own}:\ Declarations \\
 & | & \textbf{ClassDec} \ [\ \textbf{external}\ ]\ Id\ :\ Imports\ \textbf{own}:\ Declarations \\
 & & \\
Declarations' & \Rightarrow & \{\ [\ ITypeDec\ ]\ [\ ETypeDec\ ]\ [\ ObjDec\ ]\ [\ FunDec\ ]\ \} \\
 & & \\
ITypeDec' & \Rightarrow & \textbf{implicit types}\ :[\ Id\ ;\ [\ Pragma\ ]^{*}\ ]^{*} \\
 & & \\
ETypeDec' & \Rightarrow & \textbf{explicit types}\ :[\ Id\ =\ Type\ ;[\ Pragma\ ]^{*}\ ]^{*} \\
 & & \\
ObjDec' & \Rightarrow & \textbf{global objects}\ :[\ Type\ Id\ ;\ [\ Pragma\ ]^{*}\ ]^{*} \\
 & & \\
FunDec' & \Rightarrow & \textbf{functions}\ :[\ Fun\ [\ Pragma\ ]^{*}\ ]^{*} \\
 & & \\
Fun & \Rightarrow & DecReturnList\ \ Id\ \ ParamList\ ; \\
 & | & ReturnList\ \ Id\ \ DecParamList\ ; \\
 & & \\
DecReturnList & \Rightarrow & Type\ [\ ,\ Type\ ]^{*}\ [\ ,\ \ ...\ ] \\
 & | & \textbf{void} \\
 & | & ... \\
 & & \\
DecParamList & \Rightarrow & (\ Type\ [\ \&\ ]\ Id\ \ [\ ,\ Type\ [\ \&\ ]\ Id\ ]^{*}\ \ [\ ,\ \ ...\ ]\ ) \\
 & | & (\ [\ ...\ ]\ ) \\
 & & \\
Pragma & \Rightarrow & \#\textbf{pragma copyfun}\ \ String \\
 & | & \#\textbf{pragma freefun}\ \ String \\
 & | & \#\textbf{pragma initfun}\ \ String \\
 & | & \#\textbf{pragma linkname}\ \ String \\
 & | & \#\textbf{pragma effect}\ \ Id\ [\ ,\ Id\ ]^{*} \\
 & | & \#\textbf{pragma linksign}\ \ [\ Num\ [\ ,\ Num\ ]^{*}\ ] \\
 & | & \#\textbf{pragma refcounting}\ \ [\ Num\ [\ ,\ Num\ ]^{*}\ ]
\end{array}
$$

Abbildung 5.1: Erweiterte Syntax externer Module und Klassen

Die primitiven Typen werden dabei auf den jeweiligen primitiven C-Typ abgebildet (**bool** auf **int**), Arraytypen auf einen Zeiger auf den jeweiligen primitiven Basistyp. Bei Arraytypen mit fester Form ist diese Information ausreichend, da Dimension und Form und damit auch die Größe des Arrays eindeutig bestimmt sind. Im Fall eines variablen Arraytyps sind Dimension, Form und Größe dagegen unbekannt. Dieses Problem läßt sich jedoch auf einfache Weise dadurch umgehen, daß die betreffende Funktion Form und Dimension eines Arrays explizit als zusätzliche Argumente erhält.

5.2.3 C-Typen in SAC-Programmen

Eine Aufgabe der Schnittstelle besteht darin, einen möglichst großen Anteil der in der Sprache C zur Verfügung stehenden Typen in die Sprache SAC abzubilden. Dabei muß im wesentlichen zwischen solchen Typen unterschieden werden, die sich explizit abbilden lassen, und solchen, die eine implizite Abbildung erfordern.

Zur ersten Gruppe gehören die primitiven Typen sowie dynamisch allozierte Arrays eines primitiven Basistyps. Sie finden äquivalente Entsprechungen in den jeweiligen SAC-Typen. Daher können sie unmittelbar zur Deklaration einer C-Funktion verwendet werden. Entsprechende benutzerdefinierte Typen können von einem externen Modul oder einer externen Klasse als explizite Typen exportiert werden. Aufzählungstypen zählen ebenfalls zur ersten Gruppe, da sie verhältnismäßig einfach durch den primitiven Typ **int** ersetzt werden können.

Die zweite Gruppe bilden im wesentlichen die Strukturen (**struct**) und Varianten (**union**) sowie beliebig verschachtelte Konstruktionen aus Strukturen, Varianten und Arrays. Zu ihr gehören jedoch auch die statischen Arrays. Für die Typen dieser Gruppe existieren keine äquivalenten SAC-Typen. Sie können jedoch von einem externen Modul bzw. einer externen Klasse als implizite Typen exportiert werden[2]. In diesem Fall erfolgen sämtliche Operationen auf einem entsprechenden Datenobjekt mit der Hilfe von Funktionen, die von dem externen Modul bzw. der externen Klasse zusätzlich zur Verfügung gestellt werden müssen. Auf diese Weise kann auch für derartige Typen mittelbar eine Abbildung auf SAC-Typen realisiert werden.

Bei externen impliziten Typen handelt es sich grundsätzlich um strukturierte Datentypen, die eine Speicherverwaltung erforderlich machen. Analog zu Arrays soll der SAC-Programmierer damit jedoch wie bei funktionalen Sprachen üblich nicht belastet werden. Stattdessen übernimmt der SAC-Compiler diese Aufgabe. Im Rahmen dieser Speicherverwaltung ist es erforderlich, Datenobjekte kopieren und löschen zu können. Da dem SAC-Compiler die Implementierung eines externen, impliziten Typs nicht bekannt ist, muß er sich zu diesem Zweck entsprechender Funktionen des jeweiligen externen Moduls bedienen. Zu einem externen, impliziten Typ *impltype* gehören daher zusätzlich zwei Funktionen

$$\begin{array}{rrl} & \textit{impltype} & \texttt{copy\_}\textit{impltype}\texttt{(}\textit{impltype}\ \textit{id}\texttt{)} \\ \text{und} & \texttt{void} & \texttt{free\_}\textit{impltype}\texttt{(}\textit{impltype}\ \textit{id}\texttt{)}\ . \end{array}$$

Mit Hilfe der Pragmas **copyfun** und **freefun** besteht die Möglichkeit, von diesen festen Namen abzuweichen. Dies ist z.B. dann von Vorteil, wenn ein bestehendes C-Modul mit Hilfe einer externen Modul-Deklaration in ein SAC-Programm integriert werden soll. Zu diesem Zweck kann die Deklaration eines impliziten Typs um eines oder beide der genannten Pragmas ergänzt werden.

5.2.4 Globale C-Variablen in SAC-Programmen

Globale Variablen dienen in der Sprache C der Modellierung eines globalen Zustands. Mit Hilfe des in Kapitel 4 beschriebenen Klassen-Konzeptes lassen sich globale Variablen daher auf einfache Weise auf globale Objekte abbilden. Als Voraussetzung dafür muß zunächst der Typ dieser Variable durch eine externe Klasse nach SAC abgebildet werden. Von dieser externen Klasse kann dann ein entsprechendes globales Objekt exportiert werden. Die korrekte Handhabung des globalen Zustands innerhalb der funktionalen Welt von SAC gewährleistet dann das Klassen-Konzept.

Das Klassen-Konzept gewährleistet auch die Initialisierung eines globalen Zustands. Zu diesem Zweck muß eine externe Klasse *class* für jedes von ihr exportierte globale Objekt *obj* eine Funktion

$$\textit{class}\quad \texttt{create\_}\textit{obj}\ \texttt{( )}$$

besitzen. Analog zu den impliziten Typen besteht auch an dieser Stelle die Möglichkeit, mit Hilfe des Pragmas **initfun** von dem vorgegebenen Funktionsnamen abzuweichen. Die Initialisierung eines globalen Objektes kann ihrerseits von anderen globalen Objekten abhängen. In diesem Fall verwendet die entsprechende C-Funktion weitere globale Variablen. Derartige Abhängigkeiten von anderen

[2] Auch dynamisch allozierte Arrays können bei Bedarf als implizite Typen exportiert werden.

globalen Objekten müssen mit Hilfe des Pragmas **effect** angegeben werden. Darüberhinaus kann es sinnvoll sein, dem globalen Objekt einen anderen Namen als den für die globale Variable in C verwendeten zu geben. Mit Hilfe des Pragmas **linkname** läßt sich in diesen Fällen der notwendige Bezug herstellen.

5.2.5 C-Funktionen in SAC-Programmen

Oberfächlich betrachtet sind Funktionen in C und SAC einander sehr ähnlich. Trotzdem läßt sich nicht jede in C spezifizierbare Funktion in die funktionale Welt von SAC übertragen. Voraussetzung dafür ist ein „funktionales Verhalten“. Dies bedeutet konkret, daß

- Parameter ausschließlich lesend verwendet werden,
- der Resultatswert neu erzeugt wird,
- der Resultatswert ausschließlich von den Argumenten abhängt,
- die Anwendung frei von Seiteneffekten ist.

C-Funktionen, die diese Bedingungen erfüllen, können in SAC-Programmen genauso verwendet werden wie in SAC implementierte. Darüberhinaus bietet die Schnittstelle jedoch weitere Möglichkeiten, sowohl SAC-spezifische Merkmale auch in externen Modulen zu nutzen, als auch die Menge der nach SAC abbildbaren C-Funktionen zu erweitern.

Überladene Funktionen

Im Gegensatz zu C können Funktionen in der Sprache SAC überladen werden. Um von dieser Möglichkeit auch im Rahmen externer Module Gebrauch machen zu können, darf der bei der Implementierung in C verwendete Funktionsname von dem in der Modul-Deklaration angegebenen abweichen. In einem derartigen Fall wird der Implementierungsname durch das Pragma **linkname** angegeben. Auf diese Weise können beliebige primitive und benutzerdefinierte SAC-Funktionen ebenso wie andere externe Funktionen durch eine in C implementierte Funktion überladen werden.

Funktionen mit mehreren Resultatswerten

Obwohl Funktionen in der Sprache C höchstens einen Wert direkt zurückliefern können, lassen sich dennoch Funktionen mit mehren Resultatswerten spezifizieren. Dabei wird ein zusätzlicher Resultatswert durch einen weiteren Parameter modelliert, der eine Speicheradresse angibt, an der die Funktion den eigentliche Resultatswert ablegt.

Derartige Funktionen verhalten sich nicht funktional im obigen Sinne, da die speziellen Rückgabe-Parameter schreibend verwendet werden. Um diese trotzdem nach SAC abbilden zu können, muß der besondere Charakter der Rückgabe-Parameter explizit gemacht werden. Zu diesem Zweck wird eine solche Funktion mit der entsprechenden Zahl von Resultatswerten in der in SAC üblichen Form deklariert. Dies stellt die Schnittstelle jedoch vor das Problem, eine eindeutige Zuordnung zwischen den Parametern der Deklaration und denen der Implementierung zu treffen.

Standardmäßig wird davon ausgegangen, daß der erste deklarierte Resultatswert auf den echten Resultatswert der C-Funktion und die weiteren Resultatswerte und Parameter in der Reihenfolge ihrer Deklaration auf deren Parameter abgebildet werden. Von diesem Standard kann jedoch wenn nötig abgewichen werden. Zu diesem Zweck wird durch Angabe des Pragmas **linksign** eine beliebige Permutation zwischen den Resultatswerten und Parametern der Deklaration und denen der Implementierung definiert.

Funktionen mit variabler Parameterliste

Im Gegensatz zu SAC lassen sich in der Sprache C Funktionen mit einer variablen Anzahl von Parametern spezifizieren. Davon wird z.B. bei den Standardfunktionen der **printf**-Familie zur formatierten Ausgabe oder denen der **scanf**-Familie zur Analyse von Eingabedaten Gebrauch gemacht. Um auch derartige Funktionen in SAC-Programmen verwenden zu können, wird die Syntax von Modul-Deklarationen um Parameter- und Rückgabelisten variabler Länge erweitert (vgl. Abb. 5.1). Variable Rückgabelisten beziehen sich dabei auf das im vorhergehenden Abschnitt beschriebene Verfahren zur in C üblichen Spezifikation von Funktionen mit mehreren Resultatswerten.

Funktionen, die eine globale Variable verwenden

C-Funktionen, die eine globale Variable verwenden, verhalten sich ebenfalls nicht funktional im obigen Sinne. Der Resultatswert kann neben den Argumenten von dieser Variable abhängen und/oder die Anwendung führt zu einem Seiteneffekt. Trotzdem läßt sich eine solche Funktion nach SAC abbilden. Zu diesem Zweck muß die globale Variable in der in Abschnitt 5.2.4 beschriebenen Form durch ein globales Objekt modelliert werden. Dann kann mit Hilfe des Pragmas **effect** die Abhängigkeit der Funktion von diesem globalen Objekt deklariert werden.

Funktionen, die einen Parameter modifizieren

C-Funktionen, welche ein als Argument übergebenes Datenobjekt modifizieren, können auf zwei unterschiedliche Arten nach SAC abgebildet werden. Wird der betreffende Typ in SAC als Klassentyp zur Verfügung gestellt, so kann ein solcher Parameter einfach als Referenz-Parameter deklariert werden.

Handelt es sich dagegen um einen Typ ohne Uniqueness-Attribut, so kann eine Erweiterung des oben beschriebenen Pragmas **linksign** verwendet werden. Dabei wird der besondere Charakter eines derartigen Parameters, zusätzlich einen weiteren Resultatswert darzustellen, wiederum explizit gemacht. Die Funktionsdeklaration enthält für ihn sowohl einen Resultatswert als auch einen Parameter. Die Verbindung beider zu einem einzigen Parameter in der Implementierung der Funktion wird durch Angabe des Pragmas **linksign** hergestellt. Dieses erlaubt in derartigen Fällen auch nicht-injektive Abbildungen.

Funktionen mit eigener Speicherverwaltung

Die Speicherverwaltung für Datenobjekte strukturierter Typen wird in SAC vollständig durch den Compiler und damit unsichtbar für den Programmierer durchgeführt. Um externe Module und SAC-Module in ihrer Verwendung transparent zu halten, werden auch Anwendungen externer Funktionen in die Speicherverwaltung einbezogen. Zu diesem Zweck müssen jedoch gewisse Annahmen über das Verhalten einer externen Funktion getroffen werden, die dem oben beschriebenen „funktionalen Verhalten“ mit den genannten Erweiterungen der Schnittstelle entsprechen. Insofern kann bei der Implementierung einer externen Funktion vollständig auf eine Speicherverwaltung verzichtet werden.

Vielfach kann jedoch eine Funktion effizienter implementiert werden, wenn sie die notwendige Speicherverwaltung für alle oder auch nur für ausgewählte Parameter selbständig durchführt. Dies kann mit Hilfe des Pragmas **refcounting** bei der Deklaration einer Funktion angegeben werden. Voraussetzung für die Implementierung einer derartigen Funktion ist jedoch die detaillierte Kenntnis der SAC-Speicherverwaltung.

Kapitel 6

Grundkonzeption des kompilierenden Systems

Dieses Kapitel beschreibt die Grundkonzeption der Übersetzung von SAC-Programmen in ausführbaren Maschinen-Code. Zu diesem Zweck wird zunächst das von H. Wolf entwickelte Basissystem [Wol95] vorgestellt. Die Integration des Modul- und Klassen-Konzeptes stellt jedoch insbesondere vor dem Hintergrund separater Kompilation eine Reihe zusätzlicher Anforderungen an ein kompilierendes System. Diese werden identifiziert, Lösungsansätze diskutiert und schließlich die in SAC verwirklichte Variante der separaten Kompilation von Modulen und Klassen erläutert.

6.1 Das Basissystem

Die Kompilation eines SAC-Programms in ein ausführbares Maschinen-Programm erfolgt in drei grundlegenden Schritten. Zunächst wird ein SAC-Programm in ein semantisch äquivalentes Programm der Sprache C übersetzt. Daraus wird mit Hilfe eines C-Compilers ein Objekt-Modul generiert. Dieses Objekt-Modul schließlich wird von einem Linker mit den Objekt-Modulen des C-Laufzeitsystems zu einem ausführbaren Maschinen-Programm gebunden. Abbildung 6.1 zeigt diese Vorgehensweise in schematischer Darstellung.

Durch die Verwendung von C als universeller Zwischensprache kann bei der Entwicklung des eigentlichen SAC-Compilers von einer konkreten Hardware-Architektur abstrahiert werden. Bei der Erzeugung von ausführbarem Code für ein bestimmtes Zielsystem werden dagegen die Vorteile einer hochentwickelten, bestehenden Compiler-Technologie ausgenutzt. C-Compiler sind für sehr viele unterschiedliche Hardware-Architekturen erhältlich. Durch ihre Verwendung erreichen SAC-Programme daher einen hohen Grad an Portabilität. Gebräuchliche C-Compiler berücksichtigen in besonderem Maße die spezifischen Belange der jeweiligen Ziel-Architektur, wie z.B. Registeranzahl, Cache-Größen oder Pipeline-Verarbeitung. Dadurch kann besonders effizienter Maschinen-Code generiert werden.

Nachteilig wirkt sich die Verwendung einer Zwischensprache jedoch auf die Kompilationszeiten von Programmen aus. Dies liegt darin begründet, daß nacheinander zwei verschiedene Compiler geladen werden müssen, die jeweils eine lexikalische und syntaktische Überprüfung ihrer Eingabedaten durchführen. In Anbetracht der spezifischen Anforderungen im Bereich numerischer Algorithmen wird der Ausführungsgeschwindigkeit des generierten Codes sowie der Portabilität von Programmen jedoch allgemein eine höhere Priorität eingeräumt.

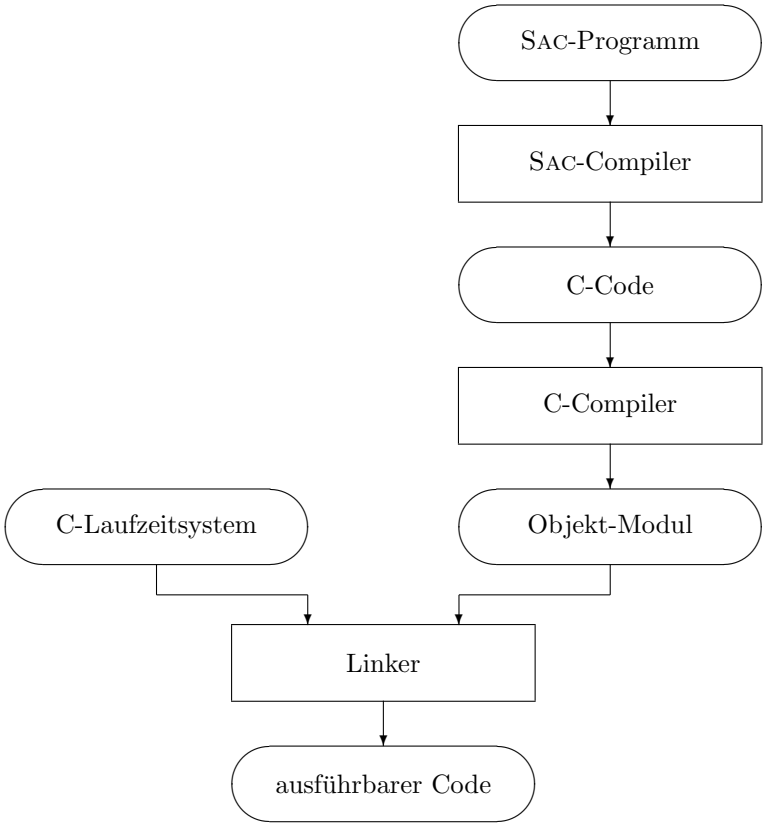

Abbildung 6.1: Übersetzung von SAC in ausführbaren Maschinen-Code

6.2 Anforderungen des Modul- und Klassen-Konzeptes

Das Modul- und Klassen-Konzept stellt eine Reihe spezifischer Anforderungen an ein kompilierendes System. Grundsätzlich sollen einzelne Module[1] separat kompiliert werden können. Zu diesem Zweck muß sich jedes Modul, insbesondere jede Modul-Deklaration und jede Modul-Implementierung in einer eigenen Datei befinden. Dies macht es für ein kompilierendes System erforderlich, zwischen verschiedenen Arten von Dateien zu unterscheiden.

Das in Abschnitt 6.1 beschriebene Kompilationssystem verarbeitet lediglich eine einzige Art von Datei. Dabei handelt es sich um die Quelldatei, die ein SAC-Programm enthält. Dieses wird in ausführbaren Maschinen-Code übersetzt, ohne daß weitere Dateien darin involviert sind[2]. Der Name der Quell-Datei muß beim Start der Kompilation angegeben werden. Der Name des ausführbaren Programms kann ebenfalls angegeben werden, anderenfalls erhält es den Standard-Namen `a.out`. Da nicht zwischen unterschiedlichen Dateien unterschieden werden muß, stellt das Kompilationssystem keine besonderen Anforderungen an die Namen von Dateien. Lediglich per Konvention besitzen die Namen von Quell-Dateien das Suffix `.sac` .

[1] Da es sich bei Klassen um spezielle Module handelt, werden sie im folgenden unter diesem Begriff subsumiert, solange ihre besonderen Eigenschaften nicht von Bedeutung sind.

[2] An dieser Stelle wird von temporären Dateien, die bei der Erzeugung von C-Code entstehen, oder den Dateien, die das C-Laufzeitsystem enthalten, abgesehen.

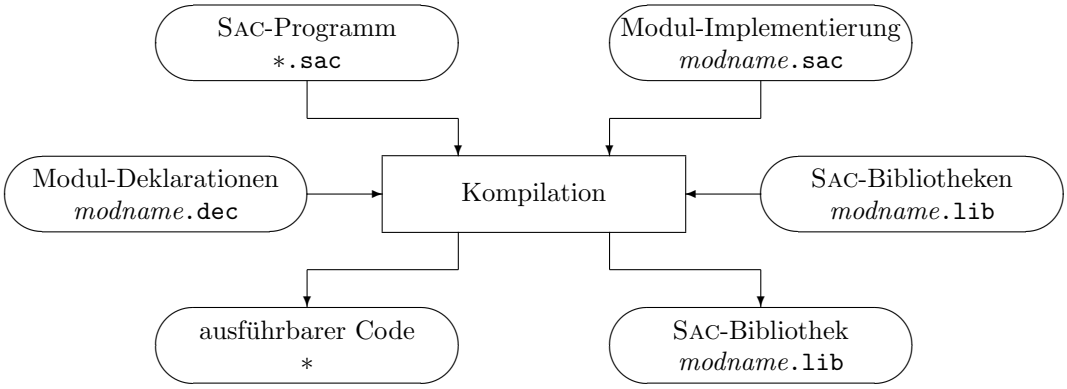

Abbildung 6.2: Anforderungen des Modul-Konzeptes an das kompilierende System

Bei Einführung des in Kapitel 3 beschriebenen Modul-Konzeptes erhöht sich die Komplexität des Kompilationssystems erheblich. Abbildung 6.2 stellt dies schematisch dar. Zunächst muß zwischen zwei verschiedenen Arten von Quell-Dateien unterschieden werden. Dabei handelt es sich zum einen um herkömmliche SAC-Programme, zum anderen um Modul-Implementierungen. Sollen letztere separat kompiliert werden, kommt als Ziel-Datei jedoch kein ausführbares Programm in Frage. Stattdessen ist ein neuer Dateityp erforderlich, der ein Modul in übersetzter und für die spätere Generierung eines ausführbaren Programms geeigneter Form enthält. Diese Art von Datei wird im folgenden als SAC-Bibliothek bezeichnet.

Wird ein SAC-Programm (oder eine Modul-Implementierung) kompiliert, das seinerseits Module importiert, so sind neben der Quell- und der Ziel-Datei weitere Dateien von der Kompilation betroffen. Dabei handelt es sich einerseits um die Deklarationen der importierten Module, andererseits um die entsprechenden SAC-Bibliotheken als Resultate vorangegangener Kompilationen. Dies macht es erforderlich, daß das Kompilationssystem vom Namen eines Moduls auf die betreffenden Dateinamen schließen kann. Zu diesem Zweck muß eine Datei, die eine Modul-Deklaration enthält, den Namen des Moduls, ergänzt um das Suffix `.dec`, tragen. Einer SAC-Bibliothek wird vom Kompilationssystem zwangsweise der Name des Moduls, ergänzt um das Suffix `.lib`, gegeben.

6.3 Möglichkeiten der Modularisierung in C

Wie in Abschnitt 6.1 beschrieben, erfolgt die Kompilation von SAC-Programmen in ausführbaren Maschinen-Code über die Zwischensprache C. Daher bilden die Modularisierungsmöglichkeiten von C letztendlich den Rahmen für die Implementierung des SAC-Modul-Konzeptes. Die Sprache C [KR88] verfügt über kein mit SAC vergleichbares Modul-Konzept. Es stehen lediglich rudimentäre Werkzeuge zur Modularisierung von Programmen zur Verfügung. Grundsätzlich kann ein Programm auf mehrere Dateien aufgeteilt werden. Jede einzelne Datei enthält Definitionen von Typen, globalen Variablen und Funktionen.

Der Bindungsbereich einer Typdefinition beschränkt sich auf die jeweilige Datei. Soll derselbe Typ in mehreren Dateien Verwendung finden, sind jeweils identische Typdefinitionen erforderlich. Trotzdem läßt sich auch in C das Konzept der Datenabstraktion anwenden. Dazu dient der primitive Typ **void***, der einen Zeiger auf einen beliebigen Datentyp definiert. Bei Definition eines Zeigertyps in einer Datei kann in anderen Dateien durch eine gleichnamige Definition des Typs **void*** von dem eigentlich zugrundeliegenden Typ abstrahiert werden.

Globale Variablen und Funktionen haben im Gegensatz zu Typen einen globalen Namensraum, d.h. ein Symbol darf in allen Dateien eines Programms höchstens einmal definiert sein. Abweichend davon kann der Bindungsbereich einer globalen Variablen oder einer Funktion auf die jeweilige Datei beschränkt werden, indem der Definition das Schlüsselwort **static** vorangestellt wird. Ist dies nicht der Fall, so können sie grundsätzlich in allen Dateien eines Programms verwendet werden. Voraussetzung dafür ist lediglich eine entsprechende Deklaration vor dem ersten angewandten Vorkommen. Dazu dient das Schlüsselwort **extern** gefolgt von Typ und Namen einer globalen Variablen bzw. einem Funktionsprototyp.

Ein C-Compiler übersetzt jede Datei eines Programms separat in Maschinen-Code, ein sog. Objekt-Modul. Angewandte Vorkommen von in anderen Dateien definierten globalen Variablen und Funktionen resultieren in symbolischen Referenzen. Erst durch den Linker werden alle Objekt-Module zu einem ausführbaren Programm gebunden und eine entsprechende Datei erzeugt.

Die Zwischensprache C erfüllt damit die notwendigen Voraussetzungen für eine Implementierung des SAC-Modul-Konzeptes und die separate Kompilation einzelner Module. Abbildung 6.3 zeigt in Anlehnung an Abbildung 6.1 die prinzipielle Vorgehensweise bei der Kompilation eines SAC-Programms oder einer Modul-Implementierung.

Der eigentliche SAC-Compiler übersetzt ein Programm oder eine Modul-Implementierung in ein C-Programm. Dabei werden Informationen aus den Deklarationen der importierten Module extrahiert. Diese resultieren in zusätzlichen Typdefinitionen und Funktionsdeklarationen. Aus dem C-Programm wird mit Hilfe eines C-Compilers ein Objekt-Modul generiert. Im Falle einer Modul-Implementierung ist die Kompilation damit beendet. Das Objekt-Modul bildet eine SAC-Bibliothek im Sinne von Abschnitt 6.2. Ein aus einem SAC-Programm entstandes Objekt-Modul dagegen wird mit den Objekt-Modulen aller importierten Module sowie denen des C-Laufzeitsystems von einem Linker zu einem ausführbaren Programm gebunden.

6.4 Probleme und Grenzen separater Kompilation

Mit der Möglichkeit, einzelne Module separat zu übersetzen, werden im wesentlichen zwei Ziele verfolgt. Erstens soll die Kompilationszeit großer Programme vermindert werden. Bei Änderungen an einem Programm müssen jeweils nur die betroffenen Module rekompiliert werden. Je größer der Modul-lokal durchführbare Anteil des gesamten Kompilationsprozesses ist, desto besser wird dieses Ziel erreicht. Zweitens sollen die Implementierungen von Funktionen und abstrakten Datentypen vor dem Anwender eines Moduls wirksam versteckt werden.

Je nach Art der zugrundeliegenden Sprache kann sich die separate Kompilation einzelner Module jedoch negativ auf die Qualität des Kompilats auswirken. Dies liegt darin begründet, daß während der Kompilation eines einzelnen Moduls weniger Information zur Verfügung steht, als dies bei gemeinsamer Kompilation aller Module eines Programms oder der Zusammenfassung des vollständigen Programms in einem einzigen Modul der Fall wäre. Der Umfang dieser negativen Auswirkungen entscheidet letztendlich über die praktische Verwendbarkeit eines Modul-Systems.

Im Anwendungsgebiet numerischer Algorithmen ist die Ausführungsgeschwindigkeit von Programmen ein entscheidendes Kriterium für die Bewertung einer Programmiersprache. Daher ist eine wichtige Anforderung an die Implementierung des Modul-Konzeptes von SAC die, daß die Modularisierung eines Programmes keine negativen Konsequenzen für dessen Laufzeitverhalten hat. Von der Erfüllung dieser Anforderung hängt in erheblichem Maße die Akzeptanz des Modul-Konzeptes und der Sprache selbst ab. Dies ist bei einer separaten Kompilation gemäß der schematischen Darstellung in Abbildung 6.3 jedoch nicht erfüllbar. Einige Gründe dafür werden im folgenden erläutert.

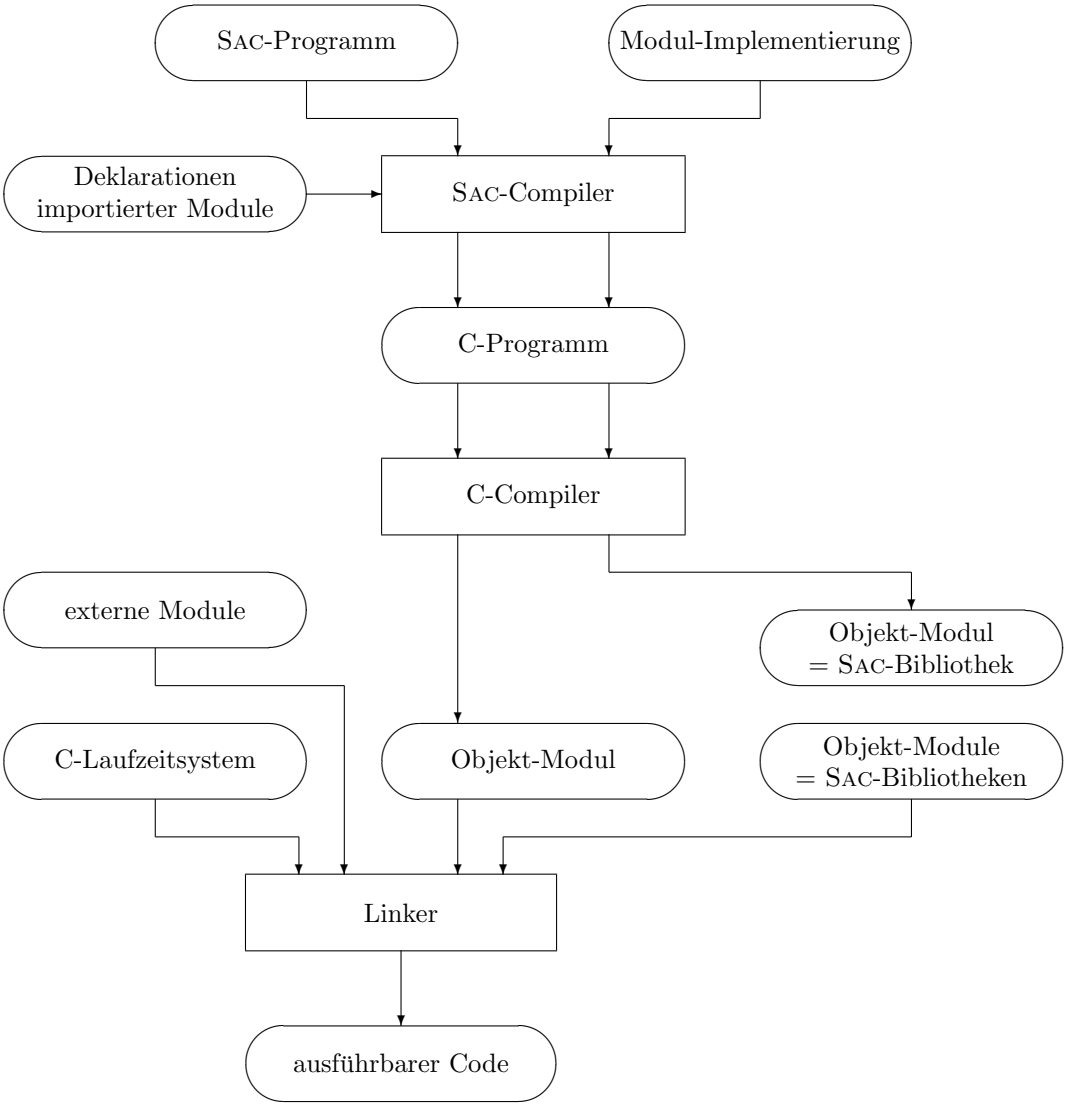

Abbildung 6.3: Erweiterung des kompilierenden Systems

6.4.1 Implizite Typen

Implizite Typen können in der Zielsprache C ausschließlich mit Hilfe des allgemeinen Zeigertyps **void*** modelliert werden. Dies hat jedoch zur Folge, daß grundsätzlich jeder Typ, der von einem Modul als impliziter Typ exportiert wird, durch den SAC-Compiler auf einen C-Zeigertyp abgebildet werden muß. Falls es sich bei dem zugrundeliegenden Typ jedoch um einen der in SAC und C identischen primitiven Typen handelt, wirkt sich dies negativ auf das Laufzeitverhalten aus. Anstatt auch in C den jeweiligen primitiven Typ zu verwenden, muß ein entsprechendes Datenobjekt im Heap erzeugt werden, auf das dann mit Hilfe eines Zeigers zugegriffen werden kann. Neben dem erhöhten Speicherbedarf für Datenobjekt und Zeiger verringert sich die Ausführungsgeschwindigkeit durch dynamische Allokation und De-Allokation von Heap-Speicher sowie die erforderliche Dereferenzierung des Zeigers bei jedem Zugriff.

6.4.2 Function Inlining

Die Sprache SAC bietet dem Programmierer die Möglichkeit, eine Funktion mit dem Schlüsselwort **inline** zu kennzeichnen. Dadurch wird der Compiler veranlaßt, eine Anwendungen dieser Funktion nach Möglichkeit durch Einsetzen ihres Rumpfes zu ersetzen. Diese als **Function Inlining** bezeichnete Optimierung verringert die Laufzeit eines Programmes häufig erheblich [Sie95]. Die separate Kompilation von Modulen beschränkt die Anwendbarkeit dieser Optimierung zwangsläufig auf diejenigen Fälle, bei denen sich Anwendung und Definition einer Funktion innerhalb desselben Moduls befinden.

6.4.3 Dimensionsunabhängige Funktionen

Das Array-Konzept von SAC erlaubt es, Funktionen auf Arrays unabhängig von deren konkreter Dimension oder Form zu spezifizieren (vgl. Abschnitt 2.3). Dabei wird die statische Typüberprüfung zwangsläufig auf einen Vergleich der Basistypen beschränkt, was zusätzliche Überprüfungen zur Laufzeit erforderlich macht. Die Folge ist, daß formunabhängig spezifizierte Programme i.a. weniger effizient ausgeführt werden als äquivalente formabhängige Programme. In Anbetracht der Bedeutung, die der Ausführungsgeschwindigkeit zukommt, ist dies jedoch vollkommen inakzeptabel und würde zwangsläufig die praktische Verwendbarkeit der dimensions- und formunabhängigen Spezifikation von Funktionen stark einschränken.

Eine Lösung dieses Problems besteht in der **Spezialisierung** formunabhängig spezifizierter Funktionen. Zu diesem Zweck wird versucht, bei einer Funktionsanwendung die genauen Formen der entsprechenden Argument-Arrays statisch zu inferieren. Auf der Basis dieser zusätzlichen Information kann der Compiler dann aus der Definition der formunabhängigen Funktion eine zu dieser äquivalente formabhängige Funktion generieren.

Die Spezialisierung einer formunabhängigen Funktion kann der Compiler jedoch nur dann durchführen, wenn er bei der Kompilation einer Funktionsanwendung den Rumpf der betreffenden Funktion kennt. Unter den Bedingungen eines Modul-Konzeptes mit separater Kompilation beschränkt sich die Spezialisierung daher zwangsläufig auf diejenigen Fälle, bei denen sich Funktionsdefinition und -anwendung in demselben Modul befinden. Dies führt unmittelbar zu dem Dilemma, daß ein Programmierer entweder auf die Modularisierung seines Programmes oder auf dessen formunabhängige Spezifikation verzichten muß, will er Laufzeitnachteile vermeiden.

6.5 Separate Kompilation in SAC

Die Implementierung des SAC-Modul-Konzeptes erfolgt unter der Prämisse, daß die Modularisierung eines Programmes keinen negativen Einfluß auf das Laufzeitverhalten des Kompilats haben darf. Unter dieser Bedingung wird versucht, die zu Beginn von Abschnitt 6.4 erläuterten Ziele einer separaten Kompilation von Modulen soweit wie möglich zu erreichen.

Diese entgegengesetzten Anforderungen werden durch eine bedingt separate Kompilation erfüllt. Im Mittelpunkt dieses Verfahrens steht die SAC-Bibliothek. Im Gegensatz zu dem in Abbildung 6.3 dargestellten Verfahren besteht eine SAC-Bibliothek aus zwei Komponenten, die in einer Datei zusammengefaßt werden. Bei der einen Komponente handelt es sich um ein Objekt-Modul, das das vollständige Kompilat einer Modul-Implementierung enthält. Dieser Teil entspricht also der SAC-Bibliothek gemäß Abbildung 6.3. Die andere Komponente einer SAC-Bibliothek bildet der **SAC-Informationsblock (SIB)**. Dabei handelt es sich um eine erweiterte textuelle Beschreibung der exportierten Symbole. Der SAC-Informationsblock erschließt dem SAC-Compiler im Falle eines angewandten Vorkommens des Moduls die zur Übersetzung in effizienten Code notwendigen Informationen. Dazu gehört z.B. der Rumpf einer **inline**-deklarierten Funktion. Im Gegensatz zur Modul-Deklaration ist der SAC-Informationsblock ausschließlich zur Auswertung durch den SAC-Compiler bestimmt und wird vor dem Benutzer versteckt.

Abbildung 6.5 enthält eine schematische Darstellung der bedingt separaten Kompilation. Der SAC-Compiler wertet nicht nur die Deklarationen der importierten Module aus, sondern zusätzlich deren SIBs. Zu diesem Zweck werden diese aus den SAC-Bibliotheken extrahiert. Bei der Kompilation einer Modul-Implementierung erzeugt der SAC-Compiler neben einem C-Programm einen neuen SAC-Informationsblock für dieses Modul. Zusammen mit dem durch den C-Compiler nachfolgend generierten Objekt-Modul bildet er eine neue SAC-Bibliothek als Resultat der Kompilation der Modul-Implementierung. Bei der Kompilation eines SAC-Programms extrahiert der Linker die erforderlichen Objekt-Module aus den jeweiligen SAC-Bibliotheken, bevor sie mit dem Kompilat des Programms, den externen Modulen und dem C-Laufzeitsystem zu einem ausführbaren Maschinen-Programm gebunden werden.

6.6 Der SAC-Informationsblock (SIB)

6.6.1 Grundaufbau

Der SAC-Informationsblock enthält Angaben über Funktionen, Typen und globale Objekte in textueller Form. Diese beschränken sich nicht auf die von einem Modul exportierten Symbole, sondern beinhalten ebenso nicht exportierte Symbole und aus anderen Modulen importierte Symbole. Abbildung 6.4 zeigt den grundlegenden Aufbau eines SAC-Informationsblocks. Er beginnt mit dem Modul-Namen eingeschlossen in `<` und `>` und endet mit der Zeichenfolge `<###>`. Dazwischen befinden sich drei jeweils optionale Abschnitte für Typen, globale Objekte und Funktionen.

$$SIB \quad \Rightarrow \quad < Id > \quad SIB - Items \quad < \#\#\# >$$

$$SIB - Items \quad \Rightarrow \quad \left[TypeInfo \right]^{*} \left[ObjInfo \right]^{*} \left[FunInfo \right]^{*}$$

Abbildung 6.4: Der Grundaufbau des SAC-Informationsblocks

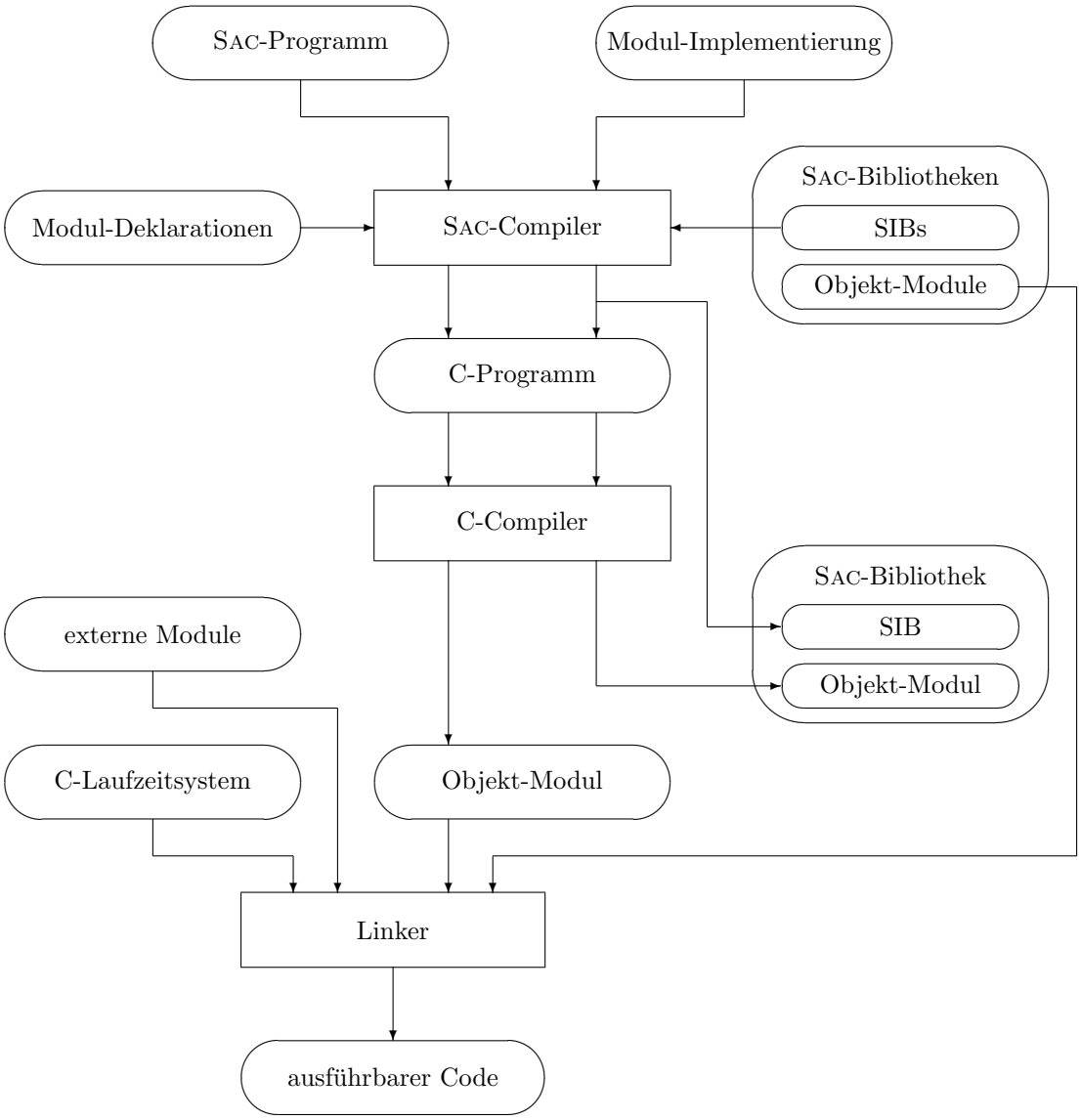

Abbildung 6.5: Die bedingt separate Kompilation

6.6.2 Funktionen im SIB

Grundsätzlich läßt sich in einem SIB zwischen primären und sekundären Funktionen unterscheiden. Als Unterscheidungsmerkmal dient dabei der Grund, weshalb der jeweilige Eintrag im SIB erforderlich ist. Es gibt drei primäre Gründe. Voraussetzung ist jeweils zusätzlich, daß die betreffende Funktion exportiert wird.

- Die Funktion verwendet implizit ein globales Objekt. Dies ist aus dem Funktionsprototyp in der Modul-Deklaration nicht ersichtlich. Der SAC-Compiler benötigt diese Information jedoch, um globale Objekte durch explizite Parameter und Rückgabewerte zu ersetzen.
- Die Funktion ist mit dem Schlüsselwort **inline** gekennzeichnet. In diesem Fall ist ihr Rumpf für die Erzeugung effizienten Codes erforderlich.
- Die Funktion hat einen oder mehrere Parameter eines variablen Arraytyps, d.h. sie ist formunabhängig spezifiziert. Auch in diesem Fall ist der Rumpf für die Code-Generierung notwendig.

Beim Importieren einer **inline**-deklarierten oder einer formunabhängig spezifizierten Funktion kann jedoch deren Rumpf nicht ohne weiteres in den neuen Kontext übertragen werden. Es muß zusätzlich sichergestellt sein, daß alle dort verwendeten Symbole auch in dem neuen Kontext zur Verfügung stehen. Dies kann aus zwei Gründen nicht der Fall sein. Entweder wird das betreffende Symbol im Rahmen eines selektiven Imports nicht berücksichtigt, oder es wird von seinem Modul überhaupt nicht exportiert. Um dies ggf. auszugleichen, sind entsprechende weitere Angaben im SAC-Informationsblock notwendig. Aus diesem Grund resultiert ein primärer Funktionseintrag i.a. in weiteren sekundären Einträgen für benötigte Funktionen, Typen oder globale Objekte.

$$
\begin{array}{lcl}
FunInfo & \Rightarrow & [\ \textbf{inline}\ ]\ FunHeader\ Body\ [\ SIBFunPragma\ ]^{*} \\
FunHeader & \Rightarrow & DecReturnList\ ModName\ DecParamList \\
Body & \Rightarrow & ; \\
 & | & ExprBlock \\
SIBFunPragma & \Rightarrow & Pragma \\
 & | & \#\textbf{pragma types}\ \ ModName\ [\ ,\ ModName\ ]^{*} \\
 & | & \#\textbf{pragma functions}\ \ FunList \\
ModName & \Rightarrow & [\ Id:\ ]\ Id \\
FunList & \Rightarrow & ModName\ DecParamList\ [\ ,\ ModName\ DecParamList\ ]^{*}
\end{array}
$$

Abbildung 6.6: Funktionen im SIB

Abbildung 6.6 zeigt den Aufbau eines Funktionseintrags. Dieser entspricht weitgehend der gewöhnlichen Grammatik einer Funktionsdefinition bzw. -deklaration. Der Funktionsprototyp dient der Identifikation der Funktion. Im Falle einer **inline**-deklarierten oder formunabhängig spezifizierten Funktion folgt der Funktionsrumpf, sonst lediglich ein **;**. Bei einem sekundären Funktionseintrag kann es sich grundsätzlich auch um eine externe Funktion handeln. Aus diesem Grund finden einerseits die erweiterten Formen von Rückgabe- und Parameterliste Verwendung, andererseits können

Pragmas Bestandteil des Eintrags sein (vgl. Abschnitt 5.2.5). Dieser Notation wird sich auch bei SAC-Funktionen zur Angabe weiterer Informationen bedient. Implizit verwendete globale Objekte werden mit Hilfe des Pragmas **effect** angegeben. Bei Eintrag eines Funktionsrumpfes werden die dort benutzten Typen und Funktionen mit Hilfe der nur im SAC-Informationsblock erlaubten Pragmas **types** und **functions** angegeben. Die dabei zu einem benötigten Symbol gemachten Angaben dienen ausschließlich der eindeutigen Identifikation des eigenen Eintrags des jeweiligen Symbols im SAC-Informationsblock.

6.6.3 Typen im SIB

Auch bei Typen läßt sich zwischen primären und sekundären Einträgen im SAC-Informationsblock unterscheiden. Jeder implizite Typ eines Moduls resultiert in einem primären Eintrag, damit seine Implementierung für die Generierung effizienten Codes zur Verfügung steht (vgl. Abschnitt 6.4.1). Sekundäre Einträge entstehen bei Benutzung eines Typs in einem im SIB eingetragenen Funktionsrumpf.

Abbildung 6.7 zeigt den Aufbau eines Typeintrags im SIB. Das Schlüsselwort **classtype** kennzeichnet Typen mit Uniqueness-Attribut, anderenfalls wird der Typeintrag durch das Schlüsselwort **typedef** eingeleitet. Anschließend folgen Implementierung und Name des Typs. Handelt es sich bei einem sekundären Typeintrag um einen externen impliziten Typ, so wird die in diesem Fall unbekannte Typ-Implementierung durch das Schlüsselwort **implicit** ersetzt. Zusätzlich können die entsprechenden Pragmas folgen (vgl. Abschnitt 5.2.3).

6.6.4 Globale Objekte im SIB

Bei globalen Objekten existiert ausschließlich die Form des sekundären Eintrags aufgrund der impliziten Verwendung durch eine Funktion. Abbildung 6.7 zeigt den Aufbau eines Objekteintrags. Er besteht lediglich aus dem Schlüsselwort **objdef** gefolgt von Typ und Namen. Bei externen globalen Objekten können zusätzlich die entsprechenden Pragmas folgen (vgl. Abschnitt 5.2.4).

$$
\begin{array}{lcl}
TypeInfo & \Rightarrow & \textbf{typedef}\ \ Type\ ModName\ ; \\
 & | & \textbf{classtype}\ \ Type\ ModName\ ; \\
 & | & \textbf{typedef}\ \ \textbf{implicit}\ \ ModName\ ;\ \ \left[\ Pragma\ \right]^{*} \\
 & | & \textbf{classtype}\ \ \textbf{implicit}\ \ ModName\ ;\ \ \left[\ Pragma\ \right]^{*} \\
 & & \\
ObjInfo & \Rightarrow & \textbf{objdef}\ \ Type\ ModName\ ;\ \ \left[\ Pragma\ \right]^{*}
\end{array}
$$

Abbildung 6.7: Typen und globale Objekte im SIB

Kapitel 7

Von SAC nach C — Der Compiler

Dieses Kapitel beschreibt die Übersetzung von Programmen der Sprache SAC in die Zwischensprache C durch den SAC-Compiler. Den Schwerpunkt bildet dabei die Integration des Modul- und Klassen-Konzeptes in den von H. Wolf entwickelten Basis-Compiler [Wol95].

7.1 Der Basis-Compiler

Der SAC-Compiler ist als Mehr-Phasen-Compiler implementiert. Dies gibt ihm einen modularen Aufbau, wodurch Erweiterung und Wartung vereinfacht werden. Abbildung 7.1 zeigt den grundlegenden Aufbau des Basis-Compilers.

Die Übersetzung von SAC nach C beginnt mit der lexikalischen und syntaktischen Analyse des Quell-Programms. Programme, die der Grammatik von SAC nicht genügen, werden mit einer entsprechenden Fehlermeldung zurückgewiesen. Syntaktisch korrekte Programme dagegen werden in eine interne, hierarchische Repräsentation, den **Syntax-Baum**, überführt. Diese interne Repräsentation bildet die Grundlage für alle weiteren Kompilationsphasen.

Die Phase der Baum-Vereinfachung dient lediglich dazu, die Implementierung der nachfolgenden Phasen zu erleichtern, indem das zu übersetzende SAC-Programm syntaktisch vereinfacht wird. Auf der Basis der daraus hervorgehenden Zwischensprache SAC_{flat} wird durch die Typ-Inferenz jedem Ausdruck ein Typ zugeordnet. Inferierte Typen werden mit den vom Programmierer angegebenen Variablen-Deklarationen verglichen. Fehlende Variablen-Deklarationen werden entsprechend ergänzt. Der auf diese Weise entstehende vollständig getypte Code der Zwischensprache SAC_{typed} wird anschließend optimiert. Mit der Referenzzählung beginnt bereits die Vorbereitung auf die Code-Erzeugung. Die dabei gewonnenen Daten dienen als Grundlage für die Erzeugung von C-Code, der den von Arrays belegten Speicher effizient verwaltet. Die Übersetzung endet mit der Generierung von C-Code auf Basis des getypten und optimierten Zwischen-Codes.

7.2 Die Integration des Modul- und Klassen-Konzeptes

Abbildung 7.2 gibt einen Überblick über die für die Integration des Modul- und Klassen-Konzeptes notwendigen Erweiterungen des Basis-Compilers. Diese bestehen sowohl in der Erweiterung vorhandener als auch in der Entwicklung zusätzlicher Kompilationsphasen. Nach der lexi-

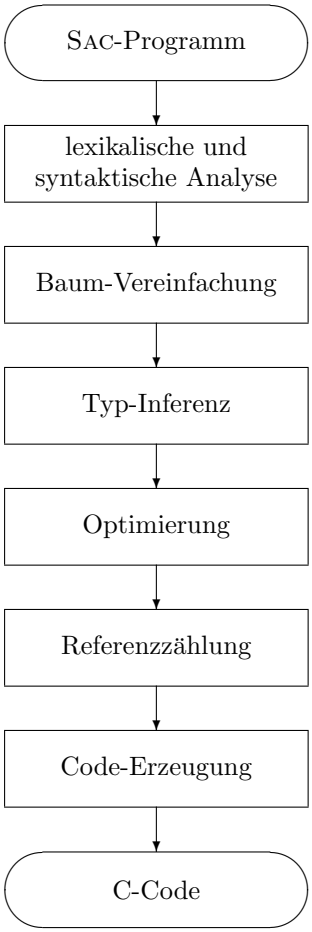

Abbildung 7.1: Der Aufbau des Basis-Compilers

kalischen und syntaktischen Analyse eines SAC-Programms bzw. einer Modul-Implementierung folgen zwei neue Kompilationsphasen, die sich mit dem Importieren von Modulen beschäftigen. Beim Modul-Import werden alle **import**-Anweisungen aufgelöst und durch entsprechende Deklarationen von Typen, Funktionen und globalen Objekten ersetzt. Die dadurch entstehende Zwischensprache wird als SAC_{import} bezeichnet. In der darauffolgenden Phase der SIB-Auswertung werden zusätzliche Informationen über importierte implizite Typen und Funktionen aus den SAC-Informationsblöcken der betreffenden Module gewonnen.

Baum-Vereinfachung und Typ-Inferenz müssen für die Belange des Modul- und Klassen-Konzeptes erweitert werden. Bei der Kompilation einer Modul-Implementierung findet im Anschluß an die Typ-Inferenz eine Konsistenzprüfung zwischen der kompilierten Modul-Implementierung und der dazugehörigen Modul-Deklaration statt. Es folgt eine Analyse aller Funktionsrümpfe. Dabei werden für jede Funktionsdefinition drei Listen erstellt, die die von dieser Funktion benötigten Symbole, d.h. benutzerdefinierte Typen, Funktionen und globale Objekte, enthalten. Bei der Kompilation einer Modul-Implementierung wird auf der Basis dieser Daten sowie der eigenen Modul-Deklaration ein neuer SAC-Informationsblock erzeugt.

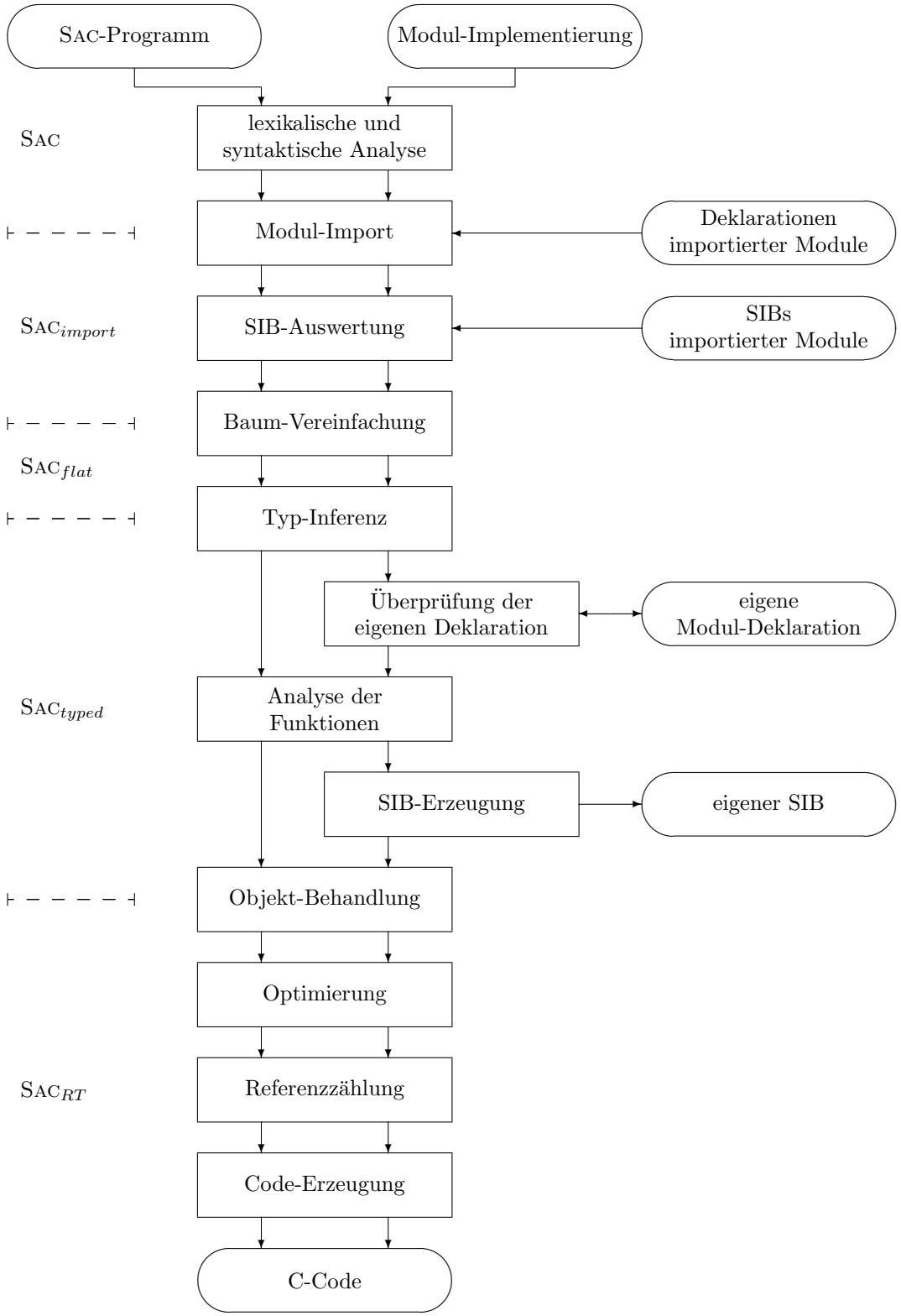

Abbildung 7.2: Integration des Modul- und Klassen-Konzeptes

Die bei der Analyse der Funktionen gewonnenen Daten über die Verwendung globaler Objekte dienen zusätzlich der nun folgenden Objekt-Behandlung. Diese besteht in der Eliminierung der in den Abschnitten 4.3.3 und 4.3.4 eingeführten Kurzschreibweisen des Call-by-Reference-Mechanismus und der globalen Objekte. Die interne Repräsentation des SAC-Codes wird dahingehend transformiert, daß globale Objekte durch explizite Parameterübergabe und der Call-by-Reference-Mechanismus durch Hinzufügen expliziter Rückgabewerte ersetzt werden. Auf dem derartig modifizierten Code wird die Einhaltung der mit dem Uniqueness-Attribut verbundenen Restriktionen überprüft. Die als Resultat der Objekt-Behandlung entstehende Zwischensprache SAC_{RT} besitzt volle referentielle Transparenz. Auf dieser Basis werden Optimierungen, Referenzzählung und Code-Erzeugung durchgeführt.

7.3 Der Modul-Import

7.3.1 Zielsetzung

Die Aufgabe dieser Kompilationsphase besteht in der Auswertung der **import**-Anweisungen. Sie werden durch Deklarationen der importierten Symbole ersetzt. Der Modul-Import läßt sich seinerseits in zwei Abschnitte gliedern:

1. Einlesen der Deklarationen importierter Module.
2. Übernahme der Deklarationen importierter Symbole in den Syntax-Baum.
   Dies kann auf zwei unterschiedliche Weisen geschehen:
   (a) durch pauschale Übernahme aller Symbole eines Moduls oder
   (b) selektive Übernahme einzelner Symbole.

7.3.2 Laden von Modul-Deklarationen

Der Modul-Import beginnt mit der lexikalischen und syntaktischen Analyse der Modul-Deklarationen, die unmittelbar in den **import**-Anweisungen des kompilierten Programms referenziert werden. Analog zu SAC-Programmen bzw. Modul-Implementierungen wird dabei für jede analysierte Modul-Deklaration eine interne, hierarchische Repräsentation erzeugt. Diese enthält neben dem Modul-Namen weitere **import**-Anweisungen sowie Deklarationen impliziter und expliziter Typen, globaler Objekte und Funktionen.

Nachfolgend werden die **import**-Anweisungen der analysierten Modul-Deklarationen untersucht und ggf. weitere Modul-Deklarationen geladen. Dieses Verfahren wird solange fortgesetzt, bis sämtliche benötigten Modul-Deklarationen in eine interne, hierarchische Repräsentation überführt worden sind. Jede Modul-Deklaration wird höchstens einmal eingelesen. Dadurch werden wechselseitige Importe oder der mehrfache Import eines Moduls auf verschiedenen Wegen aufgelöst.

7.3.3 Pauschaler Import

Durch die Anweisung `import` *modname* `all;` werden alle Symbole des Moduls *modname* importiert. Dies sind sowohl die Symbole, die von dem Modul *modname* selbst definiert und exportiert werden, als auch solche Symbole, die mittelbar über **import**-Anweisungen in der Modul-Deklaration von *modname* referenziert werden.

Zu diesem Zweck werden zunächst alle Symbol-Deklarationen aus der internen Repräsentation der Modul-Deklaration von *modname* in den Syntax-Baum des kompilierten Programmes übertragen. Um das mehrfache Importieren desselben Symbols zu verhindern, werden sie in der Modul-Deklaration von *modname* als bereits importiert gekennzeichnet. Anschließend werden die **import**-Anweisungen in der Modul-Deklaration von *modname* analog behandelt.

7.3.4 Selektiver Import

Bei der selektiven Form der **import**-Anweisung werden einzelne Symbole spezifiziert, die aus einem bestimmten Modul importiert werden sollen. Die Aufgabe besteht nun darin, einem derartigen Symbol zunächst eindeutig eine Deklaration zuzuordnen und diese dann in den Syntax-Baum des kompilierten Programmes zu übernehmen. Diese Deklaration muß sich jedoch nicht in dem in der **import**-Anweisung genannten Modul selbst befinden. Sie kann ebenso in irgendeinem anderen Modul erfolgen, das (rekursiv) von der Deklaration des ursprünglich genannten Moduls importiert wird.

Die Zuordnung einer Deklaration zu einem zu importierenden Symbol wird durch die Funktion $\mathcal{F}indDecl$ beschrieben. Ihre Definition erfolgt mit Hilfe der folgenden primitiven Operationen:

- $\mathcal{I}sDefinedIn$(symbol, module)
  gibt an, ob das Symbol *symbol* in dem Modul *module* definiert ist oder nicht.
- $\mathcal{I}mports$(module)
  liefert die Menge der **import**-Anweisungen in der Modul-Deklaration von *module*.
- $\mathcal{I}sImportAll$(imp)
  gibt an, ob die **import**-Anweisung *imp* einen pauschalen oder selektiven Import beinhaltet.
- $\mathcal{I}mportList$(imp)
  liefert die Menge der bei der selektiven **import**-Anweisung *imp* angegebenen Symbole.
- $\mathcal{M}odName$(imp)
  liefert das in der **import**-Anweisung *imp* angegebene Modul.

Die Funktion $\mathcal{F}indDecl$ erhält als Argumente das zu importierende Symbol und sowie das angegebene Modul. Als Resultat liefert sie eine Menge von Modulen, die eine Deklaration des betreffenden Symbols enthalten. Ist diese Menge leer, so ist das Symbol in dem in der **import**-Anweisung angegebenen Modul sowie allen von diesem (rekursiv) implizit importierten Modulen unbekannt. Hat sie dagegen mehr als ein Element, so kann keine eindeutige Zuordnung einer Deklaration erfolgen. In beiden Fällen kann die **import**-Anweisung nicht ausgeführt werden, was zu einer entsprechenden Fehlermeldung durch den SAC-Compiler führt.

$\mathcal{F}indDecl$(symbol, module)
= **if** $\mathcal{I}sDefinedIn$(symbol, module)
then {module}
else $\mathcal{F}indAll$(symbol, $\mathcal{I}mpMods$(symbol, $\mathcal{I}mports$(module)))

Wird das Symbol in dem genannten Modul selber definiert, so ist die Zuordnung in jedem Fall eindeutig. Anderenfalls wird mit Hilfe der Funktion $\mathcal{F}indAll$ rekursiv in den in der Deklaration des ursprünglich angegebenen Modul importierten Modulen gesucht. Zu diesem Zweck ermittelt die

Funktion $\mathcal{I}mp\mathcal{M}ods$ die Menge der in Frage kommenden Module. Dabei handelt es sich um alle diejenigen Module, die entweder pauschal importiert werden oder aus denen selektiv das gesuchte Symbol importiert wird.

$\mathcal{I}mp\mathcal{M}ods$(symbol, imports)
= **if** imports = $\emptyset$
then $\emptyset$
else **let** i $\in$ imports
in **if** $\mathcal{I}s\mathcal{I}mport\mathcal{A}ll$(i) $\vee$ symbol $\in$ $\mathcal{I}mport\mathcal{L}ist$(i)
then { $\mathcal{M}od\mathcal{N}ame$(i) } $\cup$ $\mathcal{I}mp\mathcal{M}ods$(symbol, imports $\setminus$ { i })
else $\mathcal{I}mp\mathcal{M}ods$(symbol, imports $\setminus$ { i })

Die Funktion $\mathcal{F}ind\mathcal{A}ll$ sucht in einer Menge von Modulen nach denjenigen, die eine Definition des gegebenen Symbols enthalten. Die Suche erstreckt sich zusätzlich auf alle von den gegebenen Modulen ihrerseits importierten Module. Als Resultat liefert sie eine Menge von Modulen, die eine entsprechende Definition enthalten.

$\mathcal{F}ind\mathcal{A}ll$(symbol, modules)
= **if** modules = $\emptyset$
then $\emptyset$
else **let** m $\in$ modules
in **if** $\mathcal{I}s\mathcal{D}efined\mathcal{I}n$(symbol, m)
then {m}
$\cup$ $\mathcal{F}ind\mathcal{A}ll$(symbol, $\mathcal{I}mp\mathcal{M}ods$(symbol, $\mathcal{I}mports$(m)))
$\cup$ $\mathcal{F}ind\mathcal{A}ll$(symbol, modules $\setminus$ {m})
else $\mathcal{F}ind\mathcal{A}ll$(symbol, $\mathcal{I}mp\mathcal{M}ods$(symbol, $\mathcal{I}mports$(m)))
$\cup$ $\mathcal{F}ind\mathcal{A}ll$(symbol, modules $\setminus$ {m})

7.3.5 Besondere Behandlung von Klassen

Das bisher Gesagte gilt gleichermaßen für den Import von Modulen wie von Klassen. Darüberhinaus muß beim Importieren einer Klasse grundsätzlich ein neues Symbol für den Klassentyp erzeugt werden. Dabei ist es unerheblich, ob die Klasse pauschal importiert wird oder nur einige ihrer Symbole. Da es sich bei dem Klassentyp per Definition um einen impliziten Typ handelt, der denselben Namen wie die Klasse selbst trägt, können alle dafür relevanten Informationen aus der Klassen-Deklaration entnommen werden. Dieser Typ muß zusätzlich mit dem Uniqueness-Attribut versehen werden.

7.4 Die Auswertung von SAC-Informationsblöcken

Der SAC-Informationsblock enthält Informationen zu den Symbolen eines SAC-Moduls, die die Angaben in der entsprechenden Modul-Deklaration ergänzen. Aufgabe der vorliegenden Kompilationsphase ist daher die Analyse der SIBs importierter Module und die Integration der dabei gewonnenen Informationen in den Syntax-Baum des kompilierten Programmes.

In Analogie zu seinem Aufbau (vgl. Abschnitt 6.6) wird auch bei der Auswertung eines SIBs zwischen primären und sekundären Informationen differenziert. Bei den primären Informationen handelt es sich einerseits um die zwingend erforderliche Angabe der Implementierungen impliziter Typen, andererseits um optionale ergänzende Angaben zu Funktionen. Diese können ihrerseits weitere sekundäre Informationen über benötigte Typen, Funktionen oder globale Objekte erforderlich machen.

Für jeden importierten impliziten Typ und für jede importierte Funktion wird nach einem Eintrag in dem betreffenden SAC-Informationsblock gesucht. Zu diesem Zweck wird der SIB, sofern dies noch nicht geschehen ist, aus der jeweiligen SAC-Bibliothek extrahiert, lexikalisch und syntaktisch analysiert und zuletzt in eine geeignete interne Repräsentation transformiert.

Für einen impliziten Typ muß ein Eintrag vorhanden sein. Die darin angegebene Typ-Implementierung wird ausgewertet und die Definition des impliziten Typs entsprechend ergänzt. Sofern im Falle einer Funktion ein Eintrag gefunden wird, kann dieser zwei grundlegende Arten von zusätzlichen Informationen enthalten. Zum einen kann es sich dabei um einen Funktionsrumpf handeln, zum anderen um Angaben über von dieser Funktion benötigte weitere Symbole. Zunächst wird die betreffende Funktionsdeklaration um diese Informationen ergänzt. Desweiteren muß sichergestellt werden, daß alle benötigten Symbole in dem kompilierten Programm zur Verfügung stehen. Ist dies bei einem Symbol nicht der Fall, wird ein implizites Importieren dieses Symbols erforderlich. Implizit bedeutet dabei, daß dieser Import unsichtbar für den Programmierer erfolgt. Die dafür notwendigen Daten können entsprechenden sekundären Einträgen im SIB entnommen werden. Falls eine Funktion implizit importiert wird, muß obiges Verfahren rekursiv angewandt werden, da auch sie ihrerseits möglicherweise weitere Symbole benötigt.

7.5 Die Baum-Vereinfachung

Die Baum-Vereinfachung hat lediglich die Aufgabe, der internen Repräsentation eines Programmes eine möglichst einfache Struktur zu geben. Dadurch wird die Implementierung nachfolgender Kompilationsphasen erheblich erleichtert. Die dabei entstehende vereinfachte Zwischensprache wird als SAC_{flat} bezeichnet.

Zu diesem Zweck werden geschachtelte Ausdrücke durch mehrere nicht geschachtelte Ausdrücke ersetzt. Dabei wird sukzessive jeder Teil-Ausdruck eines geschachtelten Ausdrucks durch eine neue, künstliche Variable ersetzt. Aus dieser Variablen und dem ersetzten Teil-Ausdruck wird eine neue Zuweisung generiert, die vor dem behandelten Ausdruck in die Folge von Anweisungen eingefügt wird. Dieses Verfahren führt u.a. dazu, daß in Argumentposition von Funktionsanwendungen, den Prädikaten von bedingten Verzweigungen und Schleifen oder bei der Definition konstanter Arrays ausschließlich Konstanten und Variablen vorkommen. Auf dieselbe Weise werden Ausdrücke in der **return**-Anweisung grundsätzlich durch Variablen ersetzt. Eine weitere Baum-Vereinfachung besteht darin, **for**-Schleifen durch äquivalente **while**-Schleifen zu ersetzen und im Rumpf einer With-Loop definierte Variablen so umzubenennen, daß sie sich von außerhalb definierten unterscheiden.

Durch Erweiterung der Syntax von SAC um die **objdef**-Anweisung zur Definition eines globalen Objektes (vgl. Abschnitt 4.3.4) können geschachtelte Ausdrücke jedoch auch außerhalb von Funktionsrümpfen auftreten. In diesem Fall läßt sich das oben beschriebene Verfahren nicht anwenden. Stattdessen wird der gesamte Initialisierungsausdruck einer **objdef**-Anweisung durch eine Funktionsanwendung ersetzt. Die dabei angewandte Funktion wird neu generiert. Die Funktion $\mathcal{F}latten$ beschreibt dieses Vorgehen.

$\mathcal{F}latten[\![$ `objdef` *type name* `=` *expr* `;` R $]\!]$
$\Rightarrow$ `objdef` *type name* `=` `CREATE__`*name*`();` $\mathcal{F}latten[\![$ R $]\!]$
$\mathcal{F}latten[\![$ *type* `CREATE__`*name*`()`
`{ return(`*expr*`); }` $]\!]$

Die künstliche Funktion ist parameterlos. Ihr Name wird aus dem Namen des globalen Objekts in einer Weise abgeleitet, die Konflikte mit anderen künstlichen oder benutzerdefinierten Funktionen ausschließt. Auf die so generierte Funktion kann anschließend die gewöhnliche Baum-Vereinfachung angewandt werden.

7.6 Die Typ-Inferenz

Die als Typ-Inferenz bezeichnete Kompilationsphase stellt einen wichtigen Teil der semantischen Analyse eines SAC-Programms dar. Durch sie wird jedem Ausdruck ein eindeutiger Typ zugeordnet. Da jeder Ausdruck als Konsequenz der Baum-Vereinfachung unmittelbar einer lokalen Variablen zugewiesen wird, genügt es, als Typ-Annotation eine entsprechende Variablen-Deklaration zu erzeugen. Die exakten Typ-Inferenz-Regeln werden in [Wol95] beschrieben. Eine bereits bestehende Variablen-Deklaration wird mit dem inferierten Typ des zugewiesenen Ausdrucks verglichen. Inkompatible Typen führen dabei zu einem Fehler. Solche Fehler können zwei unterschiedliche Ursachen haben. Einerseits kann eine durch den Programmierer gegebene Deklaration im Widerspruch zur tatsächlichen Verwendung der Variable stehen. Andererseits können der Variablen innerhalb desselben Rumpfes Ausdrücke verschiedenen Typs zugewiesen werden. Dies stellt insofern eine Einschränkung der in Abschnitt 2.2 erläuterten funktionalen Interpretation der Mehrfachzuweisung dar. Das Typ-System von SAC verlangt von aus funktionaler Sichtweise verschiedenen Variablen gleichen Namens innerhalb eines Funktionsrumpfes, daß sie denselben Typ haben. Dies ist erforderlich, um die notwendige Analogie zu C herzustellen, insbesondere vor dem Hintergrund, daß der Programmierer selber Variablen-Deklarationen angeben kann.

Die Typ-Informationen werden einerseits für die Generierung von C-Code benötigt, andererseits zur Auflösung von Funktionsüberladungen. Einer Funktionsanwendung kann in SAC nicht allein aufgrund des Funktionsnamens eine Definition eindeutig zugeordnet werden. Wegen der Möglichkeit, Funktionen zu überladen (vgl. Abschnitt 2.1), müssen die Typen der Argumente zu diesem Zweck bekannt sein. Da die Zielsprache C keine entsprechende Überladung von Funktionen gestattet, weist die Typ-Inferenz jeder Funktion zusätzlich einen eindeutigen Namen zu, der neben dem ursprünglichen Namen auf den Typen ihrer formalen Parameter beruht.

Besondere Beachtung erfordern die parametrisiert polymorphen Funktionen. Dies sind Funktionen, die einen oder mehrere Parameter eines variablen Arraytyps, z.B `int[]`, besitzen (vgl. Abschnitt 2.3). Beschränkt man sich bei ihnen auf einen Vergleich des Basistyps von formalem und aktuellem Parameter, so sind i.a. zusätzliche Überprüfungen zur Laufzeit erforderlich. Dies liegt darin begründet, daß die Spezifikation derartiger Funktionen gewöhnlich auf den primitiven Array-Funktionen basiert. Diese sind zwar prinzipiell dimensions- und formunabhängig definiert. Das bedeutet jedoch nicht, daß sie keinerlei Anforderungen an ihre Argumente stellen. So ist z.B. die Funktion + nur auf Arrays identischer Form definiert. Dies kann jedoch bei der Verwendung variabler Arraytypen durch das Typsystem nicht garantiert werden.

Um ein Höchstmaß an Effizienz bei der Ausführung von Programmen zu erreichen, sind derartige Überprüfungen zur Laufzeit unbedingt zu vermeiden. Daher werden Arraytypen grundsätzlich in Verbindung mit einem Form-Vektor inferiert. Bei der Anwendung einer parametrisiert polymorphen

Funktion sind daher die vollständigen Arraytypen ihrer Argumente bekannt. Diese werden dazu verwendet, aus der parametrisiert polymorphen Funktion eine neue, zusätzliche monomorphe Funktion zu generieren. Sie wird anstelle der ursprünglichen Funktion an der betreffenden Stelle angewendet. Durch diese Technik der Spezialisierung können die parametrisiert polymorphen Funktionen vollständig eliminiert werden. Auf diese Weise generiert die Typ-Inferenz ein zum ursprünglichen semantisch äquivalentes Programm, das vollständig dimensions- und formabhängig spezifiziert ist.

Die Phase der Typ-Inferenz muß für die Belange des Modul- und Klassen-Konzeptes an verschiedenen Stellen erweitert werden. Beispielsweise dürfen Referenzparameter und globale Objekte ausschließlich im Zusammenhang mit Klassen-Typen verwendet werden. Um dies sicherzustellen, ist eine entsprechende Überprüfung aller Definitionen von Funktionen und globalen Objekten erforderlich.

Auch die eigentliche Typ-Inferenz erfährt durch die globalen Objekte eine Erweiterung. Bisher konnte ein angewandtes Vorkommen einer Variablen im Rumpf einer Funktion ausschließlich an einen formalen Parameter oder eine lokale Variable gebunden sein. Durch die Einführung der globalen Objekte entsteht eine dritte Bindungsmöglichkeit. Dem muß Rechnung getragen werden, wenn der Typ eines Ausdrucks inferiert wird. Grundsätzlich überdecken dabei formale Parameter oder lokale Variablen ein gleichnamiges globales Objekt.

Eine wichtige Aufgabe der Typ-Inferenz besteht in der eindeutigen Zuordnung eines definierenden zu jedem angewandten Vorkommen eines Symbols. Diese Zuordnung wird durch das Modul-System mit separaten Namesräumen erschwert, da verschiedene Symbole gleichen Namens aus unterschiedlichen Modulen importiert werden können. Die konkrete Zuordnung erfolgt durch die in Abschnitt 7.3 definierte Funktion $\mathcal{F}indDecl$. Wird ein Symbol durch einen qualifizierten Namen (vgl. Abschnitt 3.3.4) referenziert, so beginnt die Suche in dem angegebenen Modul. Anderenfalls beginnt die Suche innerhalb des kompilierten Programmes bzw. der kompilierten Modul-Implementierung. Eine Ausnahme hiervon bilden lediglich Typ-Symbole, die als Bestandteil der Deklaration einer importierten Funktion oder eines importierten globalen Objekts auftreten. In diesen Fällen beginnt die Suche in dem Modul, das auch die Definition der Funktion bzw. des globalen Objekts enthält.

Externe Module dürfen Funktionen enthalten, die entweder eine variable Anzahl von Parametern oder eine variable Anzahl von Rückgabewerten haben (vgl. Abschnitt 5.2.5). Derartiger Funktionen stellen besondere Anforderungen an die Typ-Inferenz. Bei Anwendung einer Funktion mit einer variablen Rückgabewertliste können Typen nur für die fixen Rückgabewerte inferiert werden. Es ist daher für den Programmierer zwingend erforderlich, die Typen aller weiteren Rückgabewerte durch entsprechende Variablen-Deklarationen anzugeben. Funktionen mit variabler Parameterliste erschweren die Zuordnung einer Funktionsdefinition zu einer Funktionsanwendung. Diese erfolgt wegen der Möglichkeit, Funktionen zu überladen, grundsätzlich auf der Basis des Funktionsnamens und der Typen der Argumente der Anwendung. Dabei wird zunächst nach einer Funktion mit fixer Parameterliste gesucht. Ist diese Suche nicht erfolgreich, werden Funktionen mit variabler Parameterliste einbezogen. Bei diesen können lediglich die Typen der fixen Parameter mit den Typen der korrespondierenden Argumente verglichen werden. Von allen in dieser Hinsicht kompatiblen Funktionsdefinitionen wird diejenige mit der maximalen Anzahl fixer Parameter ausgewählt.

7.7 Die Überprüfung der eigenen Deklaration

7.7.1 Aufgaben

Die in einer Modul-Deklaration gemachten Angaben über Typen, Funktionen und globale Objekte müssen mit entsprechenden Definitionen in der korrespondierenden Modul-Implementierung über-

einstimmen. Zu diesem Zweck wird im Rahmen der Kompilation einer Modul-Implementierung eine Überprüfung der dazugehörigen Modul-Deklaration durchgeführt. Je nachdem, ob diese Deklaration bereits existiert, wird entweder eine Konsistenzprüfung vorgenommen oder eine standardisierte Modul-Deklaration aus den Daten der kompilierten Modul-Implementierung erzeugt.

7.7.2 Konsistenzprüfung

Die Konsistenzprüfung der Modul-Deklaration umfaßt die folgenden Punkte:

- Zu einer Klassen-Implementierung gehört eine Klassen-Deklaration, zu einer Modul-Implementierung eine Modul-Deklaration.
- Der im Kopf der Modul-Implementierung angegebene Modulname muß mit dem im Kopf der Modul-Deklaration angegebenen übereinstimmen.
- Alle **import**-Anweisungen müssen korrekt aufgelöst werden können.
- Jedem impliziten Typ muß eine Typdefinition gleichen Namens zugeordnet werden können.
- Jedem expliziten Typ muß eine Typdefinition gleichen Namens und identischer Typ-Implementierung zugeordnet werden können.
- Jeder Deklaration eines globalen Objekts muß eine entsprechende Definition gleichen Namens und Typs zugeordnet werden können.
- Jeder Funktionsdeklaration muß eine Funktionsdefinition gleichen Namens und identischer Signatur zugeordnet werden können.
- Alle in Funktions- und Objekt-Deklarationen angewandt vorkommenden Typ-Symbole müssen innerhalb der Modul-Deklaration bekannt sein.

7.7.3 Erzeugung einer Deklaration

Zu einer Klassen-Implementierung wird eine Klassen-Deklaration, zu einer Modul-Implementierung eine Modul-Deklaration gleichen Namens erzeugt. Diese enthält als standardisierte Angaben:

- alle ursprünglichen **import**-Anweisungen der Implementierung,
- für jede Typ-Definition einen impliziten Typ,
- für jede Objekt-Definition eine entsprechende Objekt-Deklaration,
- für jede Funktionsdefinition eine entsprechende Funktionsdeklaration.

Auf diese Weise ist sichergestellt, daß die erzeugte Deklaration die obigen Konsistenzbedingungen erfüllt. Sie kann vom Programmierer als Grundlage für die Erstellung einer individuellen Deklaration herangezogen werden.

7.8 Die Analyse der Funktionen

7.8.1 Zielsetzung

Die Aufgabe der Funktionsanalyse besteht darin, die von einer Funktion verwendeten Typen, Funktionen und globalen Objekte zu ermitteln und nachfolgenden Kompilationsphasen zur Verfügung zu stellen. Diese Daten sind aus zwei Gründen erforderlich.

Bei der Kompilation einer Modul-Implementierung muß für dieses Modul ein SAC-Informationsblock generiert werden. Dieser enthält Angaben über implizit von einer Funktion verwendete globale Objekte. Bei formunabhängig spezifizierten oder **inline**-gekennzeichneten Funktionen sind darüberhinaus Angaben über im Rumpf verwendete Typen sowie angewandte Funktionen notwendig (vgl. Abschnitt 6.6).

Sowohl bei Modul-Implementierungen als auch bei der Kompilation eines SAC-Programmes werden im Rahmen der Objekt-Behandlung alle angewandten Vorkommen globaler Objekte eliminiert. Dies geschieht durch Einführung entsprechender zusätzlicher formaler Parameter. Voraussetzung dafür ist jedoch wiederum die Kenntnis der von einer Funktion benötigten globalen Objekte.

Zu diesem Zweck werden bei der Analyse der Funktionen jeder Funktionsdefinition drei verschiedene Mengen zugeordnet:

- $\mathcal{T}ypes$(fundef) — die Menge der benötigten Typen,
- $\mathcal{F}uns$(fundef) — die Menge der benötigten Funktionen,
- $\mathcal{O}bjs$(fundef) — die Menge der benötigten globalen Objekte.

7.8.2 Analyse benötigter Typen

Die Analyse benötigter Typen ist lediglich bei formunabhängig spezifizierten oder mit dem Schlüsselwort **inline** gekennzeichneten Funktionen in Modul-Implementierungen erforderlich. Sie dient als Grundlage für einen entsprechenden SIB-Eintrag. Ziel dieses Eintrags ist es zu garantieren, daß alle im Rumpf einer Funktion verwendeten Typen auch dann bekannt sind, wenn dieser Rumpf in einen anderen Kontext, z.B. ein importierendes Programm, gestellt wird. Deshalb können solche Typen ausgeklammert werden, die ohnehin in jedem importierenden Programm bekannt sind. Dabei handelt es sich einerseits um die primitiven Typen einschließlich der primitiven Arraytypen, andererseits um alle Typen, die in der Rückgabe- oder der Parameterliste der Funktion auftreten.

$$\mathcal{T}ypes[\![\ \mathit{retlist}\ \mathit{name}(\ \mathit{paramlist}\ )\ \{\ \mathit{vardecs}\ \mathit{instructions}\ \mathit{return}\ \}\ ]\!]$$
$$= \mathcal{T}ypes[\![\ \mathit{vardecs}\ ]\!]\ \setminus\ (\ \mathcal{T}ypes[\![\ \mathit{retlist}\ ]\!]\ \cup\ \mathcal{T}ypes[\![\ \mathit{paramlist}\ ]\!]\ \cup\ \mathcal{P}rim\mathcal{T}ypes\ )$$

$$\mathcal{T}ypes[\![\ \mathit{rettype},\ \mathrm{R}\ ]\!] = \mathit{rettype} \cup \mathcal{T}ypes[\![\ \mathrm{R}\ ]\!]$$

$$\mathcal{T}ypes[\![\ \mathit{type}\ \mathit{param},\ \mathrm{R}\ ]\!] = \mathit{type} \cup \mathcal{T}ypes[\![\ \mathrm{R}\ ]\!]$$

$$\mathcal{T}ypes[\![\ \mathit{type}\ \&\mathit{param},\ \mathrm{R}\ ]\!] = \mathit{type} \cup \mathcal{T}ypes[\![\ \mathrm{R}\ ]\!]$$

$$\mathcal{T}ypes[\![\ \mathit{type}\ \mathit{var};\ \mathrm{R}\ ]\!] = \mathit{type} \cup \mathcal{T}ypes[\![\ \mathrm{R}\ ]\!]$$

7.8.3 Analyse benötigter Funktionen

Mit der Analyse der benötigten Funktionen werden zwei Ziele verfolgt. Zum einen wird diese Information wie bei den Typen im vorhergehenden Abschnitt zur Erstellung eines SIB-Eintrags gebraucht. Zum anderen dient sie als Grundlage für die im folgenden Abschnitt erläuterte Analyse benötigter globaler Objekte.

$$\mathcal{F}uns[\![\ \textit{retlist}\ \textit{name}\texttt{(}\ \textit{paramlist}\ \texttt{)}\ \{\ \textit{vardecs}\ \textit{instructions}\ \textit{return}\ \}\ ]\!]$$
$$= \mathcal{F}uns[\![\ \textit{instructions}\ ]\!] \setminus \{\textit{name}\}$$

$$\mathcal{F}uns[\![\ \textit{vars}\ \texttt{=}\ \textit{fun}\texttt{(}\ \textit{arguments}\ \texttt{);}\ \mathrm{R}\ ]\!] = \{\textit{fun}\} \cup \mathcal{F}uns[\![\ \mathrm{R}\ ]\!]$$

$$\mathcal{F}uns[\![\ \textit{vars}\ \texttt{=}\ \textit{expr}\ \texttt{;}\ \mathrm{R}\ ]\!] = \mathcal{F}uns[\![\ \mathrm{R}\ ]\!]$$

$$\mathcal{F}uns[\![\ \textit{fun}\texttt{(}\ \textit{arguments}\ \texttt{);}\ \mathrm{R}\ ]\!] = \{\textit{fun}\} \cup \mathcal{F}uns[\![\ \mathrm{R}\ ]\!]$$

$$\mathcal{F}uns[\![\ \texttt{if (}\ \textit{predicate}\ \texttt{)}\ \textit{consequence}\ \texttt{else}\ \textit{alternative}\ \texttt{;}\ \mathrm{R}\ ]\!]$$
$$= \mathcal{F}uns[\![\ \textit{consequence}\ ]\!] \cup \mathcal{F}uns[\![\ \textit{alternative}\ ]\!] \cup \mathcal{F}uns[\![\ \mathrm{R}\ ]\!]$$

$$\mathcal{F}uns[\![\ \texttt{while (}\ \textit{predicate}\ \texttt{)}\ \textit{body}\ \texttt{;}\ \mathrm{R}\ ]\!]$$
$$= \mathcal{F}uns[\![\ \textit{body}\ ]\!] \cup \mathcal{F}uns[\![\ \mathrm{R}\ ]\!]$$

$$\mathcal{F}uns[\![\ \texttt{do}\ \textit{body}\ \texttt{while (}\ \textit{predicate}\ \texttt{);}\ \mathrm{R}\ ]\!]$$
$$= \mathcal{F}uns[\![\ \textit{body}\ ]\!] \cup \mathcal{F}uns[\![\ \mathrm{R}\ ]\!]$$

7.8.4 Analyse benötigter globaler Objekte

Die Analyse benötigter globaler Objekte unterscheidet sich insofern von den Analysen benötigter Typen und Funktionen, als daß sie nicht isoliert für eine einzelne Funktion durchgeführt werden kann. Eine Funktion benötigt nicht nur solche globalen Objekte, die explizit in ihrem Rumpf verwendet werden, sondern zusätzlich sämtliche, die rekursiv von einer in ihrem Rumpf angewandten Funktion benötigt werden.

$$\mathcal{O}bjs[\![\ \text{fundef}\ ]\!]$$
$$= \mathcal{O}wn\mathcal{O}bjs[\![\ \text{fundef}\ ]\!] \cup \mathcal{O}thers\mathcal{O}bjs[\![\ \mathcal{F}uns[\![\ \text{fundef}\ ]\!]\ ,\ \{\text{fundef}\}\ ]\!]$$

$$\begin{array}{l}
\mathcal{O}thers\mathcal{O}bjs[\![\ \text{funs, done}\ ]\!] \\
\quad = \textbf{let}\ \text{todo} = \text{funs} \setminus \text{done} \\
\quad\ \ \textbf{in}\ \ \textbf{if}\ \ \text{todo} = \emptyset \\
\qquad\quad \textbf{then}\ \emptyset \\
\qquad\quad \textbf{else}\ \ \textbf{let}\ \text{f} \in \text{todo} \\
\qquad\qquad\quad \textbf{in}\ \ \mathcal{O}wn\mathcal{O}bjs[\![\ \text{f}\ ]\!] \\
\qquad\qquad\qquad\quad \cup\ \mathcal{O}thers\mathcal{O}bjs[\![\ \mathcal{F}uns[\![\ \text{f}\ ]\!]\ ,\ \text{done} \cup \{\text{f}\}\ ]\!] \\
\qquad\qquad\qquad\quad \cup\ \mathcal{O}thers\mathcal{O}bjs[\![\ \text{funs} \setminus \{\text{f}\}\ ,\ \text{done}\ ]\!]
\end{array}$$

Die obige Definition von $\mathcal{O}bjs$ läßt die Vereinigung direkt und indirekt benötigter globaler Objekte deutlich werden. Dabei bezeichnet $\mathcal{O}wn\mathcal{O}bjs[\![$ fundef $]\!]$ die Menge der direkt im Rumpf der Funktion *fundef* auftretenden globalen Objekte. Diese Menge läßt sich durch einfaches Traversieren des Funktionsrumpfes analog zur Analyse benötigter Funktionen identifizieren. Eine mögliche Zuordnung angewandter Vorkommen von Variablen zu globalen Objekten liegt als Bestandteil der Typ-Inferenz bereits vor. Daher wird auf eine exakte Definition von $\mathcal{O}wn\mathcal{O}bjs$ an dieser Stelle verzichtet.

Analog dazu liefert die Funktion $\mathcal{O}thers\mathcal{O}bjs$ die Menge der indirekt benötigten globalen Objekte. Dabei wird auf die Analyse benötigter Funktionen zurückgegeriffen. Neben diesen erhält $\mathcal{O}thers\mathcal{O}bjs$ eine Liste der bereits untersuchten Funktionen als Argument. Diese gewährleistet die Terminierung der Analyse auch für den Fall wechselseitig rekursiver Funktionen.

7.9 Die Erzeugung des SAC-Informationsblocks

Im Anschluß an die Funktionsanalyse wird bei der Kompilation einer Modul-Implementierung der SAC-Informationsblock dieses Moduls generiert. Sein Aufbau wird in Abschnitt 6.6 beschrieben und soll daher an dieser Stelle nicht wiederholt werden. Alle erforderlichen Informationen für die Erstellung eines Eintrags für eine Funktion, einen Typ oder ein globales Objekt liegen jeweils vor.

Offen ist dagegen noch die Frage, für welche Symbole im einzelnen SIB-Einträge erzeugt werden müssen. Dies wird ausgehend von der korrespondierenden Modul-Deklaration durch die Funktion $\mathcal{SIB}items$ beschrieben. Deren Definition stützt sich auf die Funktion $\mathcal{DEF}$, die jeder Typ-, Objekt- und Funktionsdeklaration eine entsprechende Definition in der Modul-Implementierung zuordnet. Diese Verbindung wird im Rahmen der in Abschnitt 7.7 beschriebenen Konsistenzprüfung hergestellt.

$$\mathcal{SIB}items[\![\ imptypes\ exptypes\ objdecs\ fundecs\ ]\!] = \mathcal{SIB}items[\![\ imptypes\ ]\!] \cup \mathcal{SIB}items[\![\ fundecs\ ]\!]$$

$$\mathcal{SIB}items[\![\ imptype\ ;\ \mathrm{R}\ ]\!] = \{\mathcal{DEF}[\![\ imptype\ ]\!]\} \cup \mathcal{SIB}items[\![\ \mathrm{R}\ ]\!]$$

$$\mathcal{SIB}items[\![\ fundec\ ;\ \mathrm{R}\ ]\!] = \mathcal{SIB}items[\![\ \mathcal{DEF}[\![\ fundec\ ]\!]\ ]\!] \cup \mathcal{SIB}items[\![\ \mathrm{R}\ ]\!]$$

$$\begin{array}{l}\mathcal{SIB}items[\![\ fundef\ ]\!] \\ \quad = \textbf{if} \quad \mathcal{IsInline}[\![\ fundef\ ]\!] \vee \mathcal{IsShapeIndependent}[\![\ fundef\ ]\!] \\ \quad\ \ \textbf{then}\ \{\ fundef\ \} \\ \qquad\qquad \cup\ \mathcal{T}ypes[\![\ fundef\ ]\!] \\ \qquad\qquad \cup\ \mathcal{O}bjs[\![\ fundef\ ]\!] \\ \qquad\qquad \cup\ \mathcal{F}uns[\![\ fundef\ ]\!] \\ \qquad\qquad \cup\ \mathcal{SIB}items[\![\ \mathcal{F}uns[\![\ fundef\ ]\!]\ ]\!] \\ \quad\ \ \textbf{else}\ \ \textbf{if} \quad \mathcal{O}bjs[\![\ fundef\ ]\!] \neq \emptyset \\ \qquad\qquad \textbf{then}\ \{\ fundef\ \} \cup \mathcal{O}bjs[\![\ fundef\ ]\!] \\ \qquad\qquad \textbf{else}\ \emptyset\end{array}$$

7.10 Die Behandlung von Objekten

7.10.1 Zielsetzung

Aufgabe der Objekt-Behandlung ist die Abbildung des Klassen-Konzeptes von SAC auf das reine Uniqueness-Typing–Konzept. Diese Abbildung erfolgt in drei Stufen:

1. Eliminierung von globalen Objekten,
2. Eliminierung von Referenz-Parametern,
3. Kontrolle der Uniqueness-Eigenschaft.

Globale Objekte und der Call-by-Reference-Mechanismus werden in Kapitel 4 als Kurzschreibweisen eingeführt. Sie stellen demnach lediglich syntaktische Vereinfachungen dar. Durch sie wird jedoch die referentielle Transparenz von SAC-Programmen eingeschränkt. Daher ist es erforderlich, diese Kurzschreibweisen aufzulösen, bevor Kompilationsphasen durchgeführt werden, die auf vollständige referentielle Transparenz angewiesen sind. Im Rahmen des hier betrachteten Compilers betrifft dies lediglich die Optimierungsphase. Vollständige referentielle Transparenz wäre jedoch auch zur Identifikation nebenläufig ausführbarer Programmteile im Rahmen der Erzeugung nicht-sequentiellen Ziel-Codes erforderlich.

Voraussetzung für die korrekte Abbildung von Zuständen in das funktionale Paradigma mit Hilfe des Uniqueness-Typing-Konzeptes ist die restringierte Verwendung von Ausdrücken, deren Typen das Uniqueness-Attribut besitzen. Die Überprüfung der Uniqueness-Eigenschaft dieser Ausdrücke bildet den dritten Teil der Objekt-Behandlung.

7.10.2 Globale Objekte

Die Eliminierung der globalen Objekte erfolgt auf der Basis der bei der Analyse der Funktionen gewonnenen Daten (vgl. Abschnitt 7.8). Als Resultat dieser Analyse wird jeder Funktionsdefinition *fundef* die Menge $\mathcal{O}bjs[\![\ \mathit{fundef}\ ]\!]$ der von dieser Funktion benötigten globalen Objekte zugeordnet. Für jedes dieser globalen Objekte wird die Parameterliste der Funktion um einen gleichnamigen Referenz-Parameter erweitert. Das hat zur Folge, daß anschließend alle Vorkommen ursprünglich globaler Objekte im Rumpf der Funktion an diese zusätzlichen Parameter gebunden sind. Dadurch werden sie zu gewöhnlichen bzw. lokalen Objekten. Diese Änderung einer Funktionsdefinition muß entsprechende Anpassungen aller Anwendungen der betroffenen Funktion nach sich ziehen. Zu diesem Zweck werden die vorhandenen Argumente einer Funktionsanwendung explizit um die von der angewandten Funktion benötigten globalen Objekte erweitert.

Eine exakte Beschreibung der Eliminierung globaler Objekte wird im folgenden durch das Transformationsschema $\mathcal{GO}$ gegeben.

$$\mathcal{GO}\ [\![\ \mathrm{code}_{\mathrm{SAC}_{typed}}\ ]\!] \quad \longrightarrow \quad \mathrm{code}_{\mathrm{SAC}_{no\_go}}$$

$\mathcal{GO}$ transformiert Code der Zwischensprache SAC_{typed} in semantisch äquivalenten Code der Zwischensprache SAC_{no_go}, die sich von SAC_{typed} durch das vollständige Fehlen globaler Objekte unterscheidet. Ein Teil der zur Definition von $\mathcal{GO}$ verwendeten Subschemata besitzt einen zweiten Parameter. Dieser bildet den jeweiligen Kontext der Transformation und besteht ebenfalls aus einem Code-Fragment der Zwischensprache SAC_{typed}. Das leere Code-Fragment wird durch das Symbol $\sqcup$ bezeichnet.

$\mathcal{GO}$⟦ *fundef* R ⟧
⇒ $\mathcal{GO}fundef$⟦ *fundef* ⟧⟦ $\mathcal{M}ake\mathcal{L}ist$⟦ $\mathcal{O}bjs$⟦ *fundef* ⟧ ⟧ ⟧ $\mathcal{GO}$⟦ R ⟧

$\mathcal{GO}fundef$⟦ *retlist fun*(*paramlist*) { *body* } ⟧⟦ *objlist* ⟧
⇒ *retlist fun*($\mathcal{GO}parlist$⟦ *paramlist* ⟧⟦ *objlist* ⟧) { $\mathcal{GO}inst$⟦ *body* ⟧ }

Das Transformationsschema $\mathcal{GO}$ traversiert die Parameterlisten und Rümpfe aller Funktionen. $\mathcal{O}bjs$⟦ *fundef* ⟧ bezeichnet die Menge der von *fundef* benötigten globalen Objekte. Durch Anwendung von $\mathcal{M}ake\mathcal{L}ist$ wird auf dieser Menge eine Ordnung definiert.

$\mathcal{GO}parlist$⟦ *paramlist* ⟧⟦ ␣ ⟧ ⇒ *paramlist*

$\mathcal{GO}parlist$⟦ ␣ ⟧⟦ `objdef` *type name* = *init*; R ⟧
⇒ $\mathcal{GO}parlist$⟦ *type* `&`*name* ⟧⟦ R ⟧

$\mathcal{GO}parlist$⟦ *paramlist* ⟧⟦ `objdef` *type name* = *init*; R ⟧
⇒ $\mathcal{GO}parlist$⟦ *type* `&`*name*`,` *paramlist* ⟧⟦ R ⟧

Für jedes benötigte globale Objekt wird die Parameterliste der betroffenen Funktion um einen zusätzlichen Referenzparameter erweitert. Name und Typ sind identisch mit denen des globalen Objektes.

$\mathcal{GO}inst$⟦ *vars* = *funap* ; ⟧
⇒ *vars* = $\mathcal{GO}ap$⟦ *funap* ⟧⟦ $\mathcal{M}ake\mathcal{L}ist$⟦ $\mathcal{O}bjs$⟦ $\mathcal{DEF}$⟦ *funap* ⟧ ⟧ ⟧ ⟧;

$\mathcal{GO}inst$⟦ *funap* ; ⟧
⇒ $\mathcal{GO}ap$⟦ *funap* ⟧⟦ $\mathcal{M}ake\mathcal{L}ist$⟦ $\mathcal{O}bjs$⟦ $\mathcal{DEF}$⟦ *funap* ⟧ ⟧ ⟧ ⟧;

$\mathcal{GO}ap$⟦ *fun*(*argumentlist*) ⟧⟦ *objlist* ⟧
⇒ *fun*($\mathcal{GO}arglist$⟦ *argumentlist* ⟧⟦ *objlist* ⟧)

Der Rumpf einer Funktion wird von $\mathcal{GO}$ ausschließlich traversiert, um Funktionsanwendungen ggf. an die geänderte Signatur ihrer jeweiligen Definition anzupassen. Daher wird auf eine vollständige Darstellung der Traversierung an dieser Stelle verzichtet. Die durch $\mathcal{DEF}$ zum Ausdruck gebrachte Zuordnung einer Definition zu jeder Anwendung einer Funktion erfolgt durch die Typ-Inferenz (vgl. Abschnitt 7.6). Die Verwendung von $\mathcal{M}ake\mathcal{L}ist$ stellt sicher, daß die Argumentliste einer Funktionsanwendung in exakt derselben Reihenfolge um globale Objekte erweitert wird wie die Parameterliste der angewandten Funktion.

$\mathcal{GO}arglist$⟦ *argumentlist* ⟧⟦ ␣ ⟧ ⇒ *argumentlist*

$\mathcal{GO}arglist$⟦ ␣ ⟧⟦ `objdef` *type name* = *init*; R ⟧
⇒ $\mathcal{GO}arglist$⟦ *name* ⟧⟦ R ⟧

$\mathcal{GO}arglist[\![$ *argumentlist* $]\!][\![$ `objdef` *type name* `=` *init*; R $]\!]$
$\Rightarrow$ $\mathcal{GO}arglist[\![$ *name*`,` *argumentlist* $]\!][\![$ R $]\!]$

Die Argumentliste wird um die von der angewandten Funktion benötigten globalen Objekte erweitert. Aufgrund der Definiton der Menge $\mathcal{O}bjs$ gilt dabei $\mathcal{O}bjs[\![\ \mathcal{DEF}[\![$ *funap* $]\!]\ ]\!] \subset \mathcal{O}bjs[\![$ *fundef* $]\!]$ für jede Funktionsanwendung *funap* im Rumpf einer Funktion *fundef*.

$\mathcal{GO}fundef[\![$ `int main() {` *vardecs instructions return* `}` $]\!][\![$ *objlist* $]\!]$
$\Rightarrow$ `int main(` $\mathcal{GO}mainparlist[\![\ \sqcup\ ]\!][\![$ *objlist* $]\!]$ `)`
`{`
$\mathcal{GO}mainvardec[\![$ *vardecs* $]\!][\![$ *objlist* $]\!]$
$\mathcal{GO}maininst[\![$ *instructions* $]\!][\![$ *objlist* $]\!]$
return
`}`

Eine von der anderer Funktionen abweichende Handhabung erfordert die **main**-Funktion. Bei den von ihr benötigten globalen Objekten wird in der Behandlung grundsätzlich zwischen den globalen Objekten der Klasse `World` und denen anderer Klassen unterschieden.

$\mathcal{GO}mainparlist[\![$ *parlist* $]\!][\![\ \sqcup\ ]\!] \Rightarrow$ *parlist*

$\mathcal{GO}mainparlist[\![\ \sqcup\ ]\!][\![$ `objdef World` *name* `=` *init*; R $]\!]$
$\Rightarrow$ $\mathcal{GO}mainparlist[\![$`World &`*name* $]\!][\![$ R $]\!]$

$\mathcal{GO}mainparlist[\![$ *parlist* $]\!][\![$ `objdef World` *name* `=` *init*; R $]\!]$
$\Rightarrow$ $\mathcal{GO}mainparlist[\![$`World &`*name*`,` *parlist* $]\!][\![$ R $]\!]$

$\mathcal{GO}mainparlist[\![$ *parlist* $]\!][\![$ `objdef` *type name* `=` *init*; R $]\!]$
$\Rightarrow$ $\mathcal{GO}mainparlist[\![$ *parlist* $]\!][\![$ R $]\!]$

Ein Objekt der Klasse `World` repräsentiert einen Teil der Umgebung des Programms. Verwendet ein Programm ein solches `World`-Objekt, so beeinflußt die Umgebung das Ergebnis der Berechnung. Daher werden `World`-Objekte zu Parametern der **main**-Funktion und damit des gesamten Programmes. Der Tatsache, daß auch umgekehrt das Programm seine Umgebung beeinflussen kann, wird durch die Anwendung des Call-by-Reference-Mechanismus Rechnung getragen.

$\mathcal{GO}mainvardec[\![$ *vardecs* $]\!][\![\ \sqcup\ ]\!] \Rightarrow$ *vardecs*

$\mathcal{GO}mainvardec[\![$ *vardecs* $]\!][\![$ `objdef World` *name* `=` *init*; R $]\!]$
$\Rightarrow$ $\mathcal{GO}mainvardec[\![$ *vardecs* $]\!][\![$ R $]\!]$

$\mathcal{GO}mainvardec[\![$ *vardecs* $]\!][\![$ `objdef` *type name* `=` *init*; R $]\!]$
$\Rightarrow$ $\mathcal{GO}mainvardec[\![$ *type name*; *vardecs* $]\!][\![$ R $]\!]$

$\mathcal{GO}maininst[\![\ instructions\ ]\!][\![\ \sqcup\ ]\!] \Rightarrow \mathcal{GO}inst[\![\ instructions\ ]\!]$

$\mathcal{GO}maininst[\![\ instructions\ ]\!][\![\ \texttt{objdef World}\ name\ \texttt{=}\ init\texttt{;}\ \mathrm{R}\ ]\!]$
$\quad \Rightarrow \mathcal{GO}maininst[\![\ instructions\ ]\!][\![\ \mathrm{R}\ ]\!]$

$\mathcal{GO}maininst[\![\ instructions\ ]\!][\![\ \texttt{objdef}\ type\ name\ \texttt{=}\ init\texttt{;}\ \mathrm{R}\ ]\!]$
$\quad \Rightarrow \mathcal{GO}maininst[\![\ name\ \texttt{=}\ init\texttt{;}\ instructions\ ]\!][\![\ \mathrm{R}\ ]\!]$

Globale Objekte, die nicht der Klasse `World` angehören, werden dagegen zu lokalen Objekten der **main**-Funktion. Deren Rumpf wird daher um entsprechende Variablen-Deklarationen sowie Zuweisungen der jeweiligen Initialisierungsausdrücke erweitert.

7.10.3 Call-by-Reference-Mechanismus

Der Call-by-Reference-Mechanismus wird in Abschnitt 4.3.3 als Kurzschreibweise für Objekt-modifizierende Funktionen eingeführt. Referenz-Parameter einer Funktion werden von dieser implizit zurückgegeben. Eine Teilaufgabe der Objekt-Behandlung besteht darin, diese impliziten Rückgabewerte explizit zu machen und dadurch Anwendungen der Kurzschreibweise Call-by-Reference-Mechanismus zu eliminieren. Zu diesem Zweck wird für jeden Referenz-Parameter in der Parameterliste einer Funktion ein zusätzlicher Rückgabetyp in der Rückgabetypliste und ein zusätzlicher Rückgabewert in der **return**-Anweisung eingetragen. Der Referenz-Parameter wird daraufhin zu einem gewöhnlichen Parameter. Sämtliche Anwendungen einer derartig modifizierten Funktion müssen entsprechend angepaßt werden.

Eine exakte Beschreibung der Eliminierung von Referenz-Parametern wird im folgenden durch das Transformationsschema $\mathcal{CBR}$ gegeben.

$$\mathcal{CBR}\ [\![\ \mathrm{code}_{\mathrm{SAC}_{no\_go}}\ ]\!] \quad \longrightarrow \quad \mathrm{code}_{\mathrm{SAC}_{RT}}$$

$\mathcal{CBR}$ transformiert Code der Zwischensprache SAC_{no_go} in semantisch äquivalenten Code der Zwischensprache SAC_{RT}, die sich von SAC_{no_go} durch das Fehlen von Referenz-Parametern unterscheidet. Ein Teil der zur Definition von $\mathcal{CBR}$ verwendeten Subschemata besitzt einen zweiten oder dritten Parameter. Diese bilden wie bei $\mathcal{GO}$ im vorangegangenen Abschnitt den jeweiligen Kontext der Transformation und repräsentieren hier jeweils ein Code-Fragment der Zwischensprache SAC_{no_go}.

$\mathcal{CBR}[\![\ retlist\ fun\texttt{(}\ paramlist\ \texttt{)}\ \{\ vardecs\ instructions\ return\ \}\ \mathrm{R}\ ]\!]$
$\quad \Rightarrow \mathcal{CBR}retlist[\![\ retlist\ ]\!][\![\ paramlist\ ]\!]\quad fun\texttt{(}\ \mathcal{CBR}parlist[\![\ paramlist\ ]\!]\ \texttt{)}$
$\quad\ \{$
$\quad\ vardecs$
$\quad\ \mathcal{CBR}inst[\![\ instructions\ ]\!]$
$\quad\ \mathcal{CBR}ret[\![\ return\ ]\!][\![\ paramlist\ ]\!]$
$\quad\ \}$
$\quad\ \mathcal{CBR}[\![\ \mathrm{R}\ ]\!]$

Bei der Transformation einer Funktionsdefinition wird die Parameterliste traversiert, um Referenz-Parameter durch gewöhnliche Parameter zu ersetzen, Rückgabetypliste und **return**-Anweisung, um

in Abhängigkeit von der Parameterliste zusätzliche Rückgabewerte einzutragen, und der Rumpf, um Funktionsanwendungen entsprechend anzupassen. Anschließend wird die Transformation mit der nächsten Funktionsdefinition fortgesetzt.

$\mathcal{CBR}parlist[\![\ \sqcup\ ]\!] \Rightarrow \sqcup$

$\mathcal{CBR}parlist[\![\ type\ name,\ \mathrm{R}\ ]\!] \Rightarrow type\ name,\ \mathcal{CBR}parlist[\![\ \mathrm{R}\ ]\!]$

$\mathcal{CBR}parlist[\![\ type\ \texttt{\&}name,\ \mathrm{R}\ ]\!] \Rightarrow type\ name,\ \mathcal{CBR}parlist[\![\ \mathrm{R}\ ]\!]$

Bei der Transformation einer Parameterliste werden lediglich die Referenz-Operatoren „&“ eliminiert.

$\mathcal{CBR}retlist[\![\ retlist\ ]\!][\![\ \sqcup\ ]\!] \Rightarrow retlist$

$\mathcal{CBR}retlist[\![\ \texttt{void}\ ]\!][\![\ type\ name,\ \mathrm{R}\ ]\!] \Rightarrow \mathcal{CBR}retlist[\![\ \texttt{void}\ ]\!][\![\ \mathrm{R}\ ]\!]$

$\mathcal{CBR}retlist[\![\ \texttt{void}\ ]\!][\![\ type\ \texttt{\&}name,\ \mathrm{R}\ ]\!] \Rightarrow \mathcal{CBR}retlist[\![\ type\ ]\!][\![\ \mathrm{R}\ ]\!]$

$\mathcal{CBR}retlist[\![\ retlist\ ]\!][\![\ type\ name,\ \mathrm{R}\ ]\!] \Rightarrow \mathcal{CBR}retlist[\![\ retlist\ ]\!][\![\ \mathrm{R}\ ]\!]$

$\mathcal{CBR}retlist[\![\ retlist\ ]\!][\![\ type\ \texttt{\&}name,\ \mathrm{R}\ ]\!] \Rightarrow \mathcal{CBR}retlist[\![\ type,\ retlist\ ]\!][\![\ \mathrm{R}\ ]\!]$

Die Rückgabetypliste einer Funktion wird für jeden Referenz-Parameter in deren Parameterliste um den Typ des Referenz-Parameters erweitert. Der spezielle Rückgabetyp **void** repräsentiert die leere Rückgabetypliste und wird daher beim ersten Auftreten eines Refernz-Parameters eliminiert.

$\mathcal{CBR}ret[\![\ return\ ]\!][\![\ \sqcup\ ]\!] \Rightarrow return$

$\mathcal{CBR}ret[\![\ \sqcup\ ]\!][\![\ type\ name,\ \mathrm{R}\ ]\!] \Rightarrow \mathcal{CBR}ret[\![\ \sqcup\ ]\!][\![\ \mathrm{R}\ ]\!]$

$\mathcal{CBR}ret[\![\ \sqcup\ ]\!][\![\ type\ \texttt{\&}name,\ \mathrm{R}\ ]\!] \Rightarrow \mathcal{CBR}ret[\![\ \texttt{return(}\ name\ \texttt{);}\ ]\!][\![\ \mathrm{R}\ ]\!]$

$\mathcal{CBR}ret[\![\ \texttt{return(}\ retvars\ \texttt{);}\ ]\!][\![\ type\ name,\ \mathrm{R}\ ]\!]$
$\quad \Rightarrow \mathcal{CBR}ret[\![\ \texttt{return(}\ retvars\ \texttt{);}\ ]\!][\![\ \mathrm{R}\ ]\!]$

$\mathcal{CBR}ret[\![\ \texttt{return(}\ retvars\ \texttt{);}\ ]\!][\![\ type\ \texttt{\&}name,\ \mathrm{R}\ ]\!]$
$\quad \Rightarrow \mathcal{CBR}ret[\![\ \texttt{return(}\ name,\ retvars\texttt{);}\ ]\!][\![\ \mathrm{R}\ ]\!]$

Die **return**-Anweisung einer Funktion wird für jeden Referenz-Parameter in deren Parameterliste um den Namen des Referenz-Parameters erweitert. Im Falle einer Funktion, die ursprünglich keinen Rückgabewert besitzt (**void**-Funktion), wird beim ersten Auftreten eines Referenz-Parameters die fehlende **return**-Anweisung hinzugefügt.

$\mathcal{CBR}inst[\![\ funap;\ ]\!]$
$\quad \Rightarrow\ \mathcal{CBR}ap[\![\ \sqcup\ ]\!][\![\ funap\ ]\!][\![\ \mathcal{DEF}[\![\ funap\ ]\!]\ ]\!]\ ;$

$\mathcal{CBR}inst[\![\ vars = funap;\ ]\!]$
$\quad \Rightarrow\ \mathcal{CBR}ap[\![\ vars =]\!][\![\ funap\ ]\!][\![\ \mathcal{DEF}[\![\ funap\ ]\!]\ ]\!]\ ;$

$\mathcal{CBR}ap[\![\ vars\ ]\!][\![\ fun(\ arguments\ )\ ]\!][\![\ retlist\ fun(\ paramlist\ )\ body\ ]\!]$
$\quad \Rightarrow\ \mathcal{CBR}vars[\![\ vars\ ]\!][\![\ arguments\ ]\!][\![\ paramlist\ ]\!]\ \ fun(\ arguments\ )$

Der Rumpf einer Funktion wird vollständig traversiert. Wie schon beim Transformationsschema $\mathcal{GO}$ wird auch an dieser Stelle die Darstellung auf den allein interessanten Fall einer Funktionsanwendung beschränkt. Dabei müssen ggf. zusätzliche explizite Rückgabewerte der angewandten Funktion Variablen zugewiesen werden.

$\mathcal{CBR}vars[\![\ vars\ ]\!][\![\ \sqcup\ ]\!][\![\ \sqcup\ ]\!]\ \Rightarrow\ vars$

$\mathcal{CBR}vars[\![\ \sqcup\ ]\!][\![\ arg,\ \mathrm{R}\ ]\!][\![\ type\ name,\ \mathrm{RR}\ ]\!]\ \Rightarrow\ \mathcal{CBR}vars[\![\ \sqcup\ ]\!][\![\ \mathrm{R}\ ]\!][\![\ \mathrm{RR}\ ]\!]$

$\mathcal{CBR}vars[\![\ \sqcup\ ]\!][\![\ arg,\ \mathrm{R}\ ]\!][\![\ type\ \&name,\ \mathrm{RR}\ ]\!]\ \Rightarrow\ \mathcal{CBR}vars[\![\ arg =]\!][\![\ \mathrm{R}\ ]\!][\![\ \mathrm{RR}\ ]\!]$

$\mathcal{CBR}vars[\![\ vars\ ]\!][\![\ arg,\ \mathrm{R}\ ]\!][\![\ type\ name,\ \mathrm{RR}\ ]\!]\ \Rightarrow\ \mathcal{CBR}vars[\![\ vars\ ]\!][\![\ \mathrm{R}\ ]\!][\![\ \mathrm{RR}\ ]\!]$

$\mathcal{CBR}vars[\![\ vars\ ]\!][\![\ arg,\ \mathrm{R}\ ]\!][\![\ type\ \&name,\ \mathrm{RR}\ ]\!]\ \Rightarrow\ \mathcal{CBR}vars[\![\ arg,\ vars\ ]\!][\![\ \mathrm{R}\ ]\!][\![\ \mathrm{RR}\ ]\!]$

Für jeden Referenz-Parameter in der Parameter-Liste der angewandten Funktion wird die linke Seite der betroffenen Zuweisung um eine neue Variable erweitert, die mit dem entsprechenden Argument der Anwendung identisch ist. Da der Call-by-Reference-Mechanismus ausschließlich auf Klassen-Typen anwendbar ist und diese grundsätzlich abstrakte Datentypen sind, handelt es sich in der Zwischensprache SAC_{no_go} bei dem betreffenden Argument zwangsläufig um eine Variable. Falls es sich bei der angewandten Funktion ursprünglich um eine **void**-Funktion handelt, muß entsprechend beim ersten Auftreten eines Referenz-Parameters eine neue Zuweisung erzeugt werden.

7.10.4 Überprüfung der Uniqueness-Eigenschaft

Uniqueness wird in Abschnitt 4.2.3 als eine Eigenschaft definiert, die ein Ausdruck erfüllt, wenn er höchstens einmal referenziert wird. Diese Uniqueness-Eigenschaft wird von SAC für alle Objekte, d.h. Ausdrücke eines Klassentyps gefordert. Als letzter Teil der Objekt-Behandlung wird diese restringierte Verwendung überprüft. Diesem Zweck dient die Funktion $\mathcal{UNQ}$, die damit die Uniqueness-Eigenschaft für die Sprache SAC auch formal definiert.

$$\begin{array}{lcllcl}
\mathcal{UNQ} & : & Var_{unq} \times \mathrm{instr}^{+}_{\mathrm{SAC}_{RT}} & \longrightarrow & \mathbb{B} \\
\mathcal{UNQ}used & : & Var_{unq} \times \mathrm{expr}_{\mathrm{SAC}_{RT}} & \longrightarrow & \mathbb{B} \\
\mathcal{NOT}used & : & Var_{unq} \times \mathrm{instr}^{+}_{\mathrm{SAC}_{RT}} \cup \mathrm{expr}_{\mathrm{SAC}_{RT}} & \longrightarrow & \mathbb{B}
\end{array}$$

Die Funktion $\mathcal{UNQ}$ gibt an, ob die gegebene Variable eines Klassentyps in der gegebenen Sequenz von Anweisungen konform mit der geforderten Uniqueness-Eigenschaft verwendet wird. Zusätzlich werden zwei Hilfsfunktionen benötigt. $\mathcal{UNQ}used$ prüft, ob eine Variable in einem Ausdruck genau einmal vorkommt. $\mathcal{NOT}used$ ermittelt dagegen, ob eine Variable in dem gegebenen Ausdruck oder in ihrem Bindungsbereich innerhalb einer Sequenz von Anweisungen überhaupt verwendet wird.

$$\mathcal{UNQ}_{var}[\![\ v_1,\ \ldots\ ,v_n = expr;\ \mathrm{R}\ ]\!]$$
$$\iff \begin{cases} (\ \mathcal{UNQ}used_{var}[\![\ expr\ ]\!] \vee \mathcal{NOT}used_{var}[\![\ expr\ ]\!]\ ) & : \quad var \in \{v_1,\ \ldots\ ,v_n\} \\ \wedge\ \mathcal{UNQ}_{var}[\![\ \mathrm{R}\ ]\!] & \\ (\ \mathcal{UNQ}used_{var}[\![\ expr\ ]\!] \wedge \mathcal{NOT}used_{var}[\![\ \mathrm{R}\ ]\!]\ ) & : \quad var \notin \{v_1,\ \ldots\ ,v_n\} \\ \vee\ \ (\ \mathcal{NOT}used_{var}[\![\ expr\ ]\!] \wedge \mathcal{UNQ}_{var}[\![\ \mathrm{R}\ ]\!]\ ) & \end{cases}$$

$$\mathcal{UNQ}_{var}[\![\ \texttt{return}(v_1,\ \ldots\ ,v_n);\ ]\!] \iff |\ \{i \in \{1,\ldots,n\} \mid v_i = var\}\ | \ \leq\ 1$$

$$\mathcal{NOT}used_{var}[\![\ v_1,\ \ldots\ ,v_n = expr;\ \mathrm{R}\ ]\!]$$
$$\iff \begin{cases} \mathcal{NOT}used_{var}[\![\ expr\ ]\!] & : \quad var \in \{v_1,\ \ldots\ ,v_n\} \\ \mathcal{NOT}used_{var}[\![\ expr\ ]\!] \wedge \mathcal{NOT}used_{var}[\![\ \mathrm{R}\ ]\!] & : \quad var \notin \{v_1,\ \ldots\ ,v_n\} \end{cases}$$

$$\mathcal{NOT}used_{var}[\![\ \texttt{return}(v_1,\ \ldots\ ,v_n);\ ]\!] \iff \{i \in \{1,\ldots,n\} \mid v_i = var\}\ =\ \emptyset$$

Bei der Betrachtung einer Zuweisung muß grundsätzlich zwischen zwei Fällen unterschieden werden. Wird der betrachteten Variable ein neuer Wert zugewiesen, so endet der ursprüngliche Bindungsbereich mit dem auf der rechten Seite stehenden Ausdruck. In diesem Fall ist die Uniqueness-Eigenschaft dann erfüllt, wenn die Variable in dem Ausdruck höchstens einmal referenziert wird und die Eigenschaft auch in dem neuen Bindungsbereich gilt. Im anderen Fall darf die Variable maximal entweder einmal im Ausdruck oder einmal in den nachfolgenden Anweisungen referenziert werden.

$$\begin{array}{lcl}
\mathcal{UNQ}used_{var}[\![\ const\ ]\!] & \iff & \text{false} \\
\mathcal{UNQ}used_{var}[\![\ PrimFunAp\ ]\!] & \iff & \text{false} \\
\mathcal{UNQ}used_{var}[\![\ v\ ]\!] & \iff & var\ =\ v \\
\mathcal{UNQ}used_{var}[\![\ f(a_1,\ \ldots\ ,a_n)\ ]\!] & \iff & |\ \{i \in \{1,\ldots,n\} \mid a_i = var\}\ |\ =\ 1 \\
\mathcal{UNQ}used_{var}[\![\ [a_1,\ \ldots\ ,a_n]\ ]\!] & \iff & |\ \{i \in \{1,\ldots,n\} \mid a_i = var\}\ |\ =\ 1
\end{array}$$

$$\begin{array}{lcl}
\mathcal{NOT}used_{var}[\![\ const\ ]\!] & \iff & \text{true} \\
\mathcal{NOT}used_{var}[\![\ PrimFunAp\ ]\!] & \iff & \text{true} \\
\mathcal{NOT}used_{var}[\![\ v\ ]\!] & \iff & var\ \neq\ v \\
\mathcal{NOT}used_{var}[\![\ f(a_1,\ \ldots\ ,a_n)\ ]\!] & \iff & \{i \in \{1,\ldots,n\} \mid a_i = var\}\ =\ \emptyset \\
\mathcal{NOT}used_{var}[\![\ [a_1,\ \ldots\ ,a_n]\ ]\!] & \iff & \{i \in \{1,\ldots,n\} \mid a_i = var\}\ =\ \emptyset
\end{array}$$

Als Resultat der Baum-Vereinfachung (vgl. Abschnitt 7.5) können an Argumentpositionen einer Funktionsanwendung bzw. in einer Array-Definition ausschließlich Variablen und Konstanten auftreten. Da es sich bei einem Klassentyp um einen abstrakten Datentyp handelt, ist das Auftreten einer der hier betrachteten Variablen bei der Anwendung einer primitiven Funktion ausgeschlossen.

$$\mathcal{UNQ}_{var}[\![\ \texttt{if}\ (\ \mathit{predicate}\ )\ \mathit{consequence}\ \texttt{else}\ \mathit{alternative};\ \mathrm{R}\ ]\!]$$
$$\Longleftrightarrow\ \mathcal{UNQ}_{var}[\![\ \mathit{consequence};\ \mathrm{R}\ ]\!]\ \ \wedge\ \ \mathcal{UNQ}_{var}[\![\ \mathit{alternative};\ \mathrm{R}\ ]\!]$$

$$\mathcal{NOT}used_{var}[\![\ \texttt{if}\ (\ \mathit{predicate}\ )\ \mathit{consequence}\ \texttt{else}\ \mathit{alternative};\ \mathrm{R}\ ]\!]$$
$$\Longleftrightarrow\ \mathcal{NOT}used_{var}[\![\ \mathit{consequence};\ \mathrm{R}\ ]\!]\ \ \wedge\ \ \mathcal{NOT}used_{var}[\![\ \mathit{alternative};\ \mathrm{R}\ ]\!]$$

Konsequenz und Alternative einer bedingten Verzweigung werden jeweils separat auf die Einhaltung der Uniqueness-Eigenschaft untersucht. Gemäß der funktionalen Interpretation (vgl. Abschnitt 2.2) der bedingten Verzweigung werden sie jeweils um die nachfolgenden Anweisungen erweitert. Das Prädikat braucht nicht betrachtet zu werden, da es sich dabei als Ergebnis der Baum-Vereinfachung ausschließlich um eine Variable oder Konstante des primitiven Typs **bool** handeln kann.

$$\mathcal{UNQ}_{var}[\![\ \texttt{while}\ (\ \mathit{predicate}\ )\ \mathit{body};\ \mathrm{R}\ ]\!]$$
$$\Longleftrightarrow\ \mathcal{UNQ}_{var}[\![\ \mathit{body};\ \mathit{body};\ \mathrm{R}\ ]\!]$$

$$\mathcal{NOT}used_{var}[\![\ \texttt{while}\ (\ \mathit{predicate}\ )\ \mathit{body};\ \mathrm{R}\ ]\!]$$
$$\Longleftrightarrow\ \mathcal{NOT}used_{var}[\![\ \mathit{body};\ ]\!]\ \ \wedge\ \ \mathcal{NOT}used_{var}[\![\ \mathrm{R}\ ]\!]$$

Der Uniqueness-Überprüfung einer **while**-Schleife liegt die Vorstellung zu Grunde, daß eine Schleife durch die n-fache Konkatenation des Schleifenrumpfes ersetzt werden kann. Dabei gibt $n \in \mathbb{N}$ die Anzahl der Schleifendurchläufe an. Dieses n ist jedoch i.a. nicht statisch inferierbar und kann zudem für ein und dieselbe Schleife verschiedene Werte annehmen. Daher wird die Bedeutung der **while**-Schleife in Abschnitt 2.2 mit Hilfe einer tailend-rekursiven Funktion erklärt. Für die reine Überprüfung der Uniqueness-Eigenschaft ist die genaue Kenntnis des Wertes von n jedoch nicht erforderlich, weshalb auf die vereinfachende Vorstellung von der Konkatenation von Schleifenrümpfen zurückgegriffen werden kann. Mit der zweifachen Konkatenation wird das minimale n gewählt, bei dem alle potentiellen Uniqueness-Verletzungen auftreten. Analog zur bedingten Verzweigung braucht das Prädikat der Schleife nicht betrachtet zu werden.

$$\mathcal{UNQ}_{var}[\![\ \texttt{do}\ \mathit{body}\ \texttt{while}\ (\ \mathit{predicate}\ );\ \mathrm{R}\ ]\!]$$
$$\Longleftrightarrow\ \mathcal{UNQ}_{var}[\![\ \mathit{body};\ \ \texttt{while}\ (\ \ \mathit{predicate}\ )\ \mathit{body};\ \mathrm{R}\ ]\!]$$

$$\mathcal{NOT}used_{var}[\![\ \texttt{do}\ \mathit{body}\ \texttt{while}\ (\ \mathit{predicate}\ );\ \mathrm{R}\ ]\!]$$
$$\Longleftrightarrow\ \mathcal{NOT}used_{var}[\![\ \mathit{body};\ \ \texttt{while}\ (\ \ \mathit{predicate}\ )\ \mathit{body};\ \mathrm{R}\ ]\!]$$

Die **do**-Schleife kann auf einfache Weise auf die **while**-Schleife zurückgeführt werden.

$$\mathcal{UNQ}_{var}[\![\ v = \texttt{with}\ (\ \mathit{generator}\ )\ \mathit{body}\ \mathit{operator};\ \mathrm{R}\ ]\!]$$
$$\Longleftrightarrow\ \mathcal{UNQ}_{var}[\![\ \mathrm{R}\ ]\!]\ \wedge\ \ \mathcal{NOT}used_{var}[\![\ \mathit{body}]\!]\ \wedge\ \ \mathcal{UNQ}_{var}[\![\ \mathit{body}]\!]$$

$$\mathcal{NOT}used_{var}[\![\ v = \texttt{with}\ (\ \mathit{generator}\ )\ \mathit{body}\ \mathit{operator};\ \mathrm{R}\ ]\!]$$
$$\Longleftrightarrow\ \mathcal{NOT}used_{var}[\![\ \mathit{body}\ ]\!]\ \ \wedge\ \ \mathcal{NOT}used_{var}[\![\ \mathrm{R}]\!]$$

Eine gesonderte Betrachtung erfordert die With-Loop. Ausdrücke eines abstrakten Datentyps können ausschließlich in ihrem Rumpf auftreten. Dessen Anweisungen werden für jedes Element der durch den *Generator* definierten Index-Menge ausgewertet (vgl. Abschnitt 2.3). Dabei wird bewußt keine Ordnung auf der Index-Menge definiert. Eine Sequentialisierung durch explizites Environment Passing ist daher nicht sinnvoll. Folglich repräsentiert eine Variable im Rumpf einer With-Loop ggf. n Referenzen auf ein entsprechendes definierendes Vorkommen außerhalb der With-Loop, wobei n die Mächtigkeit der Index-Menge beschreibt. Die Uniqueness-Eigenschaft wird daher bei Verwendung einer Variablen im Rumpf einer With-Loop grundsätzlich nicht erfüllt[1]. Dies bedeutet jedoch nicht, daß überhaupt keine Variablen mit Uniqueness-Attribut im Rumpf einer With-Loop auftreten dürfen. Variablen-Definitionen und daran gebundene angewandte Vorkommen innerhalb des Rumpfes stellen keinen grundsätzlichen Konflikt mit der Uniqueness-Eigenschaft dar. Für derartige Fälle muß diese daher zusatzlich überprüft werden. Dieser Vorgang wird dadurch erleichtert, daß als Konsequenz der Baum-Vereinfachung (vgl. Abschnitt 7.5) alle im Rumpf einer With-Loop definierten Variablen so umbenannt sind, daß Namensgleichheit mit außerhalb definierten Variablen ausgeschlossen ist.

7.11 Die Optimierungen

Auf der als Resultat der vorangegangenen Kompilationsphasen entstandenen Zwischensprache SAC$_{RT}$ werden zahlreiche Optimierungen durchgeführt. Diese nutzen in vollem Umfang die referentielle Transparenz von SAC$_{RT}$ aus. Die wichtigsten Optimierungen werden im folgenden genannt, eine detaillierte Beschreibung findet sich in [Sie95].

- **Function Inlining** ersetzt einen Funktionsaufruf durch den Rumpf der betreffenden Funktion.
- **Constant Folding** ersetzt die Anwendung einer primitiven Funktionen auf Konstanten durch das Resultat der Operation.
- **Common Subexpression Elimination** sorgt dafür, daß identische Ausdrücke nur einmal berechnet werden.
- **Loop Unrolling** ersetzt eine Schleife durch das mehrfache Einfügen ihres Rumpfes.
- **Dead Code Removal** eliminiert solche Programmteile, die nicht zur Berechnung des Gesamtergebnisses beitragen.
- **Array Elimination** ersetzt kleine Arrays durch die direkte Verarbeitung ihrer jeweiligen Elemente.
- **Index Vector Elimination** vermeidet die explizite Erzeugung von Index-Vektoren für die Selektion von Array-Elementen oder Teil-Arrays durch die primitiven Funktionen.

7.12 Die Referenzzählung

In einer funktionalen Sprache wie SAC werden konzeptuell alle Argumente einer Funktionsanwendung konsumiert und die Resultatswerte neu generiert. Wird ein Datenobjekt mehrfach referenziert, so muß für jede Referenz eine Kopie erzeugt werden. Im Falle einfacher Datentypen, wie z.B. **int** oder **float**, geschieht genau dasselbe auch in imperativen Sprachen, wie der Zielsprache C.

[1] Der Sonderfall einer 1-elementigen Index-Menge wird an dieser Stelle nicht separat betrachtet.

Bei Ausführung einer Funktionsanwendung erzeugt das Laufzeitsystem auf dem Laufzeitstack ein **Activation Record**, das u.a. Kopien der Argumente enthält.

Für strukturierte Datentypen, wie z.B. Arrays, läßt sich dieses Verfahren jedoch nicht anwenden. Sämtliche Elemente eines Argument-Arrays bei jeder Funktionsanwendung auf dem Laufzeitstack zu kopieren, würde sowohl die Ausführungszeit eines Programmes als auch den Speicherbedarf für den Laufzeitstack erheblich erhöhen. Daher wird für strukturierte Datentypen in imperativen Sprachen ein anderes Verfahren gewählt. Bei diesem wird die eigentliche Datenstruktur in einem anderen Speicherbereich, dem **Heap**, abgelegt und auf dem Laufzeitstack lediglich Referenzen auf diesen Speicherbereich kopiert. Diese unterschiedliche Repräsentation einfacher und strukturierter Datenobjekte wird gegenüber dem Programmierer jedoch nicht verborgen, sondern muß bei der Programmierung entsprechend Berücksichtigung finden. Dies macht sich z.B. darin bemerkbar, daß lediglich Referenzen auf strukturierte Datenobjekte als Argument an eine Funktion übergeben werden können, nicht jedoch das Datenobjekt selbst. Die Verwaltung eines von einem strukturierten Datenobjekt belegten Speicherbereichs wird dem Programmierer überlassen. Sie kann explizit durch Funktionen zur Allokation und De-Allokation von Speicherbereichen oder implizit durch entsprechende Variablen-Deklarationen erfolgen.

Im Gegensatz zu C und anderen imperativen Sprachen wird in SAC grundsätzlich nicht zwischen der Behandlung strukturierter und einfacher Datenobjekte unterschieden. Das bedeutet für die Übersetzung von SAC nach C, daß der Compiler entsprechende Anweisungen zur Speicherverwaltung von Arrays in das Kompilat integrieren muß. Um ein effizientes Laufzeitverhalten zu erreichen, sollen dabei Speicherbereiche so selten wie möglich kopiert und so früh wie möglich freigegeben werden. Zu diesem Zweck wird jedem zur Ablage eines Arrays im Heap allozierten Speicherbereich ein **Referenzzähler** zugewiesen. Dessen Stand gibt zur Laufzeit die Anzahl der auf diesen Speicherbereich existierenden Referenzen an und spiegelt damit die Anzahl der konzeptuell existierenden Kopien des betreffenden Arrays wider. Bei der Ausführung einer Funktionsanwendung wird der Stand des Referenzzählers eines als Argument übergebenen Arrays um die Anzahl der angewandten Vorkommen des entsprechenden formalen Parameters im Rumpf der Funktion erhöht. Anschließend wird er um eins erniedrigt, was der konzeptuellen Konsumierung des Arrays durch die Funktionsanwendung entspricht. Hat ein Referenzzähler den Stand null, so kann der betreffende Speicherbereich freigegeben oder zur Konstruktion eines neuen Arrays wiederverwendet werden.

Voraussetzung für die Generierung von C-Code, der eine Speicherverwaltung für Arrays in der oben beschriebenen Form durchführt, ist eine weitere semantische Analyse des SAC-Programms. Diese betrifft lokale Variablen und formale Parameter einer Funktion, deren Typ ein Arraytyp ist, im folgenden als Arrayvariablen bezeichnet. Zu jeder Definition einer Arrayvariablen wird die Anzahl der im jeweiligen Bindungsbereich liegenden Referenzen ermittelt. Diese statische Referenzzählung wird durch die Funktion

$$\mathcal{R}ef\colon\ Var_{array}\ \times\ expr_{\mathrm{SAC}_{RT}}\ \cup\ instr^{+}_{\mathrm{SAC}_{RT}}\ \longrightarrow\ I\!N$$

beschrieben. $\mathcal{R}ef$ erwartet als Argumente eine Arrayvariable und einen SAC-Ausdruck oder eine Sequenz von SAC-Anweisungen. Das Resultat gibt die Anzahl der Referenzen auf diese Arrayvariable in dem gegebenen Ausdruck bzw. der gegebenen Sequenz von Anweisungen wider. Eine genaue Definition der Funktion $\mathcal{R}ef$, insbesondere unter Berücksichtigung von bedingten Verzweigungen, Schleifen und der With-Loop, wird in [Wol95] angegeben.

Die Integration des Modul- und Klassen-Konzeptes erfordert lediglich eine Erweiterung des Umfangs der von der Referenzzählung betroffenen Variablen. Neben den Arrayvariablen müssen zusätzlich alle Variablen und formalen Parameter eines externen impliziten Typs in die Referenzzählung einbezogen werden. Dies führt zu der erweiterten Funktion

$$\mathcal{R}ef'\colon\ Var_{array}\ \cup\ Var_{hidden}\ \times\ expr_{\mathrm{SAC}_{RT}}\ \cup\ instr^{+}_{\mathrm{SAC}_{RT}}\ \longrightarrow\ I\!N\ .$$

Da die Definition von $\mathcal{R}ef'$ mit der von $\mathcal{R}ef$ identisch ist, wird sie im folgenden kurz als $\mathcal{R}ef$ bezeichnet.

7.13 Die Code-Erzeugung

7.13.1 Aufgaben

In dieser letzten Phase der Kompilation wird aus der internen Repräsentation eines SAC$_{RT}$-Programmes ein C-Programm in textueller Form erzeugt. Dies ist für den in Abschnitt 2.1 beschriebenen Sprachkern von SAC unproblematisch, da dessen Konstrukte weitgehend äquivalente Entsprechungen in der Sprache C finden. Fehlende Variablen-Deklarationen werden durch die Typ-Inferenz ergänzt, Überladungen benutzerdefinierter und primitiver Funktionen durch Umbenennung aufgelöst. Lediglich die Tatsache, daß Funktionen in SAC mehrere Rückgabewerte haben können, erfordert besondere Beachtung bei der Code-Erzeugung.

Wesentlich komplexer ist dagegen die Abbildung des Array-Konzeptes von SAC auf die Sprache C. Für Arrays muß C-Code erzeugt werden, der die Verwaltung des zur Repräsentation eines Arrays benötigten Speichers vollständig beinhaltet. Die Grundlage der Speicherverwaltung bildet die im vorhergehenden Abschnitt beschriebene statische Referenzzählung. Zudem müssen die primitiven Array-Operationen und die With-Loop in effizienten C-Code übersetzt werden.

7.13.2 Intermediate Code Macros (ICM)

Um die semantische Lücke zwischen SAC und C zu verringern, wird mit Hilfe der **Intermediate Code Macros (ICM)** eine zusätzliche Zwischenstufe definiert. Die Code-Generierung erfolgt dadurch in zwei Schritten. Im ersten Schritt wird C-Code erzeugt, der Intermediate Code Macros enthält. Der zweite Schritt erfolgt implizit durch Makro-Expansion der ICMs während der Übersetzung durch den C-Compiler[2]. Durch die Intermediate Code Macros wird ein zusätzliches Abstraktionsniveau geschaffen, das die Änderung der konkreten Implementierung von z.B. Array-Operationen ohne Eingriff in die eigentliche Code-Generierung ermöglicht.

Die Intermediate Code Macros des Basis-Compilers lassen sich in drei Gruppen einteilen. Die der ersten Gruppe implementieren einen Funktionswertrückgabe-Mechanismus, der es SAC-Funktionen erlaubt, mehr als einen Resultatswert zu haben. Die zweite Gruppe dient der Abstraktion von einer konkreten Repräsentation der Arrays und der damit verbundenen Frage der Speicherverwaltung. Die dritte Gruppe schließlich implementiert die primitiven Array-Operationen sowie die With-Loop. Im Rahmen der vorliegenden Arbeit sind jedoch nur die ICMs der ersten und zweiten Gruppe von Interesse. Die wichtigsten davon werden im folgenden kurz erläutert:

FUN_DEC$(f, tag_1, \alpha_1, a_1, \ldots, tag_n, \alpha_n, a_n\ )$ generiert eine Funktionsdeklaration mit dem Namen f. Jeder Funktionsparameter wird durch drei ICM-Parameter beschrieben. Die Marke tag_i klassifiziert den i-ten Parameter als formalen Parameter (**IN-Parameter**) oder Rückgabewert (**OUT-Parameter**) der Funktion mit (`in_rc`/`out_rc`) oder ohne (`in`/`out`) dynamische Referenzzählung. α_i gibt den Typ an, a_i den Namen. Dabei handelt es sich bei IN-Parametern

[2]Ein Teil der ICMs besitzt eine variable Anzahl von Parametern. Da der C-Präprozessor derartige Makros nicht verarbeitet, werden diese durch spezielle Ausgabe-Funktionen realisiert.

(`in`/`in_rc`) um den Namen des formalen Parameters, bei OUT-Parametern (`out`/`out_rc`) um den Namen der Variablen in der **return**-Anweisung.

FUN_RET$(tag_1, r_1, \ldots, tag_n, r_n)$ erzeugt C-Code für die Rückgabe von Funktionswerten. Die r_i sind die Namen der Variablen aus der **return**-Anweisung. Die Marken tag_i geben an, ob für den jeweiligen Resultatswert eine Referenzzählung durchgeführt wird (`out_rc`) oder nicht (`out`).

FUN_AP$(f, tag_1, a_1, \ldots, tag_n, a_n)$ realisiert die Anwendung der benutzerdefinierten Funktion f und zugleich die Zuweisung der Resultatswerte an Variablen. Analog zu dem ICM **FUN_DEC** klassifizieren Marken die einzelnen Parameter als IN-Parameter (`in`/`in_rc`) oder OUT-Parameter (`out`/`out_rc`) mit oder ohne Referenzzählung. Die a_i bezeichnen bei IN-Parametern das Argument der Funktionsanwendung. Dies kann eine Variable oder eine Konstante eines primitiven Typs sein. Bei OUT-Parametern bezeichnet a_i dagegen die Variable, der der betreffende Rückgabewert zugewiesen wird.

DECL_ARRAY$(A, \alpha, s_1, \ldots, s_n)$ beschreibt die Deklaration einer Arrayvariablen A mit dem Basistyp α. Die s_i geben die Form des entsprechenden Arrays an. Durch dieses ICM werden aufgrund der Werte s_i die Anzahl der Elemente, die Dimension und die Form eines Arrays in den C-Code integriert, so daß auf diese Informationen von anderen ICMs zugegriffen werden kann.

CREATE_CONST_ARRAY$(A, a_1, \ldots, a_n)$ alloziert Speicher für ein Array A und initialisiert dieses mit den a_i.

ASSIGN_RC(B, A) realisiert eine Zuweisung der Form `A=B;` für Arrays.

SET_RC(A, n) setzt den Referenzzähler des Arrays A auf den Wert n.

INC_RC(A, n) erhöht den Referenzzähler des Arrays A um den Wert n.

DEC_RC(A, n) erniedrigt den Referenzzähler des Arrays A um den Wert n.

DEC_RC_FREE_ARRAY(A, n) erniedrigt den Referenzzähler des Arrays A um den Wert n. Falls er danach den Wert null besitzt, wird der von A belegte Speicherbereich freigegeben.

7.13.3 Code-Erzeugung im Basis-Compiler

Die Code-Erzeugung für ein SAC$_{RT}$-Programm wird durch das Kompilationsschema $\mathcal{CC}$ beschrieben.

$$\mathcal{CC} \;:\; \text{code}_{\text{SAC}_{RT}} \longrightarrow \text{code}_{\text{ICM}} \cup \text{code}_{C}$$

Hierdurch wird Code der Zwischensprache SAC$_{RT}$ in eine Darstellung transformiert, die aus Konstrukten der Sprache C ergänzt um Intermediate Code Macros (ICM) besteht. Eine ausführliche Beschreibung des Kompilationsschemas $\mathcal{CC}$ für die Konstrukte des SAC-Sprachkerns sowie des Array-Konzeptes erfolgt in [Wol95].

7.13.4 Objekte

Die besonderen Eigenschaften von Objekten erlauben die Erzeugung von optimiertem C-Code. Die Uniqueness-Eigenschaft vereinfacht die Speicherverwaltung bei strukturierten Typen. So ist eine

dynamische Referenzzählung nicht erforderlich, da mehrfache Referenzen auf dasselbe Objekt ausgeschlossen sind. Als Konsequenz der Objekt-Behandlung (vgl. Abschnitt 7.10) beinhaltet die Zwischensprache SAC_{RT} keinerlei Referenz-Parameter und globale Objekte. Die dadurch erreichte referentielle Transparenz ist jedoch im erzeugten, sequentiellen C-Code weder notwendig noch hilfreich, da der C-Compiler diese Eigenschaft nicht nutzt. Der sequentielle Kontrollfluß macht zusätzlich das explizite Environment Passing überflüssig. Auf der Grundlage des Wissens um die ursprünglichen Referenz-Parameter und globalen Objekte läßt sich daher ein optimierter C-Code erzeugen. Zu diesem Zweck werden zunächst sämtliche durch die Transformationsschemata $\mathcal{CBR}$ und $\mathcal{GO}$ zusätzlich eingefügten Parameter und Rückgabewerte von Funktionen wieder eliminiert. Stattdessen werden ursprünglich globale Objekte auf globale C-Variablen abgebildet. Ursprüngliche Referenz-Parameter werden auf die in C übliche Form der Call-by-Reference-Parameterübergabe durch Übergabe der Adresse des betreffenden Daten-Objekts abgebildet.

Für die Kompilation von Objekten werden die folgenden Intermediate Code Macros erweitert bzw. zusätzlich eingeführt:

FUN_DEC$(f, tag_1, \alpha_1, a_1, \ldots, tag_n, \alpha_n, a_n\ )$,

FUN_RET$(tag_1, r_1, \ldots, tag_n, r_n)$ und

FUN_AP$(f, tag_1, r_1, \ldots, tag_n, r_n)$ werden um die zusätzliche Marke `inout` zur Kennzeichnung von Referenz-Parametern erweitert.

DECL_UNQ_ARRAY$(A, \alpha, s_1, \ldots, s_n)$ deklariert analog zu **DECL_ARRAY** ein Array ohne dynamische Referenzzählung.

FREE_ARRAY(A) löscht ein Array ohne Bezugnahme auf einen Referenzzähler.

ALLOC_RC(A) konvertiert ein Array ohne in ein Array mit dynamischer Referenzzählung.

MAKE_UNQ_ARRAY$(A,\ B)$ konvertiert ein Array A mit dynamischer Referenzzählung in ein Array B mit Uniqueness-Eigenschaft und ohne dynamische Referenzzählung. Zu diesem Zweck muß das Array A je nach Stand des Referenzzählers kopiert werden.

Die wichtigsten Erweiterungen des Kompilationsschemas $\mathcal{CC}$ gegenüber der in [Wol95] beschriebenen Form werden im folgenden erläutert. Zu diesem Zweck werden die folgenden Bezeichnungen eingeführt.

Var_{array} bezeichnet die Menge der Variablen und formalen Parameter einer Funktion, deren Typ ein Arraytyp ist. Die Elemente werden im folgenden auch Array-Variablen bzw. Array-Parameter genannt.

Var_{unq} bezeichnet die Menge der Variablen und formalen Parameter einer Funktion, deren Typ ein Uniqueness-Typ ist. Ihre Elemente werden Objekte genannt, die Elemente aus $Var_{array} \cap Var_{unq}$ auch Array-Objekte.

$Was\mathcal{CBR}(var)$ bezeichnet einen formalen Parameter einer Funktion, der ursprünglich ein Referenz-Parameter war, oder eine Variable, die zur Eliminierung eines Referenz-Parameters zusätzlich eingefügt worden ist.

$Was\mathcal{GO}(var)$ bezeichnet eine Variable oder einen formalen Parameter einer Funktion, die bzw. der zur Eliminierung eines globalen Objektes zusätzlich eingefügt worden ist.

$$\mathcal{CC}[\![\ \texttt{objdef}\ \textit{type}\ \textit{name}\ \texttt{=}\ \textit{init};\ ]\!]$$
$$\Longrightarrow \begin{cases} \texttt{DECL\_UNQ\_ARRAY(}\ \textit{name, base, } s_0, \ldots, s_n\ \texttt{);} & : \quad \textit{type} \equiv \textit{base}[s_0, \ldots, s_n] \\ \textit{type name;} & : \quad \text{sonst} \end{cases}$$

Definitionen globaler Objekte werden zu Deklarationen externer (globaler) Variablen kompiliert. Der Initialisierungsausdruck hat dabei keine Bedeutung, die Initialisierung erfolgt als Resultat der Objekt-Behandlung im Rumpf der **main**-Funktion.

$$\mathcal{CC}[\![\ \textit{retlist fun(}\ \textit{paramlist}\ \texttt{)}\ \{\ \ \textit{body;}\ \texttt{return(}\ \textit{retvars}\ \texttt{);}\ \}\ ]\!]$$
$$\Longrightarrow \texttt{FUN\_DEC(}\ \textit{fun},\ \mathcal{CC}\textit{out}[\![\ \textit{retlist}\ ]\!][\![\ \textit{retvars}\ ]\!]\ ,\ \mathcal{CC}\textit{in}[\![\ \textit{paramlist}\ ]\!])$$
$$\{$$
$$\mathcal{CC}\textit{param}[\![\ \textit{paramlist}\ ]\!][\![\ \textit{body;}\ \texttt{return(}\ \textit{retvars}\ \texttt{);}\ ]\!]$$
$$\mathcal{CC}[\![\ \textit{body;}\ \texttt{return(}\ \textit{retvars}\ \texttt{);}\ ]\!]$$
$$\}$$

$$\mathcal{CC}\textit{out}[\![\ \textit{type}, \text{R}\ ]\!][\![\ \textit{name}, \text{RR}\ ]\!]$$
$$\Longrightarrow \begin{cases} \mathcal{CC}\textit{out}[\![\text{R}]\!][\![\text{RR}]\!] & : \quad \mathcal{WASCBR}(\textit{name}) \vee \mathcal{WASGO}(\textit{name}) \\ \texttt{out\_rc},\ \textit{type, name, } \mathcal{CC}\textit{out}[\![\text{R}]\!][\![\text{RR}]\!] & : \quad \textit{name} \in \mathit{Var}_{\mathit{array}} \setminus \mathit{Var}_{\mathit{unq}} \\ \texttt{out},\ \textit{type, name, } \mathcal{CC}\textit{out}[\![\text{R}]\!][\![\text{RR}]\!] & : \quad \text{sonst} \end{cases}$$

$$\mathcal{CC}\textit{in}[\![\ \textit{type, name}, \text{R}\ ]\!]$$
$$\Longrightarrow \begin{cases} \mathcal{CC}\textit{in}[\![\text{R}]\!] & : \quad \mathcal{WASGO}(\textit{name}) \\ \texttt{inout},\ \textit{type, name, } \mathcal{CC}\textit{in}[\![\text{R}]\!] & : \quad \mathcal{WASCBR}(\textit{name}) \\ \texttt{in\_rc},\ \textit{type, name, } \mathcal{CC}\textit{in}[\![\text{R}]\!] & : \quad \textit{name} \in \mathit{Var}_{\mathit{array}} \setminus \mathit{Var}_{\mathit{unq}} \\ \texttt{in},\ \textit{type, name, } \mathcal{CC}\textit{in}[\![\text{R}]\!] & : \quad \text{sonst} \end{cases}$$

Bei der Kompilation einer Funktionsdefinition zu dem ICM **FUN_DEC** werden alle durch die Objekt-Behandlung (vgl. Abschnitt 7.10) zugefügten Rückgabetypen und formalen Parameter ignoriert. Die ursprünglichen Referenz-Parameter erhalten die Marke `inout`.

$$\mathcal{CC}\textit{param}[\![\ \textit{type name}, \text{R}\ ]\!][\![\ \textit{instrs}\ ]\!]$$
$$\Longrightarrow \mathcal{CC}\textit{rc}[\![\ \textit{name}\ ]\!][\![\ \textit{instrs}\ ]\!] \quad \mathcal{CC}\textit{param}[\![\ R\ ]\!][\![\ \textit{instrs}\ ]\!]$$

$$\mathcal{CC}\textit{rc}[\![\ \textit{name}\ ]\!][\![\ \textit{instrs}\ ]\!]$$
$$\Longrightarrow \begin{cases} \texttt{FREE\_ARRAY(}\ \textit{name}\ \texttt{);} & : \quad \textit{name} \in \mathit{Var}_{\mathit{array}} \cap \mathit{Var}_{\mathit{unq}} \\ & \quad \wedge\ \ \mathcal{R}\mathit{ef}_{\mathit{name}}[\![\ \textit{instrs}\ ]\!] = 0 \\ \texttt{DEC\_RC\_FREE\_ARRAY(}\ \textit{name}\texttt{, 1);} & : \quad \textit{name} \in \mathit{Var}_{\mathit{array}} \setminus \mathit{Var}_{\mathit{unq}} \\ & \quad \wedge\ \ \mathcal{R}\mathit{ef}_{\mathit{name}}[\![\ \textit{instrs}\ ]\!] = 0 \\ \texttt{INC\_RC(}\ \textit{name, X-1}\ \texttt{);} & : \quad \textit{name} \in \mathit{Var}_{\mathit{array}} \setminus \mathit{Var}_{\mathit{unq}} \\ & \quad \wedge\ \ X = \mathcal{R}\mathit{ef}_{\mathit{name}}[\![\ \textit{instrs}\ ]\!] \geq 2 \\ \sqcup & : \quad \text{sonst} \end{cases}$$

Am Beginn eines Funktionsrumpfes müssen für Array-Parameter ggf. ICMs zur Speicherverwaltung eingefügt werden. Wird ein Array-Objekt im Rumpf der Funktion nicht referenziert, kann es gelöscht werden. Für Array-Parameter ohne Uniqueness-Eigenschaft wird dagegen der Referenzzähler zur Laufzeit um die Anzahl der Referenzen im Rumpf der Funktion erhöht. Zusätzlich wird er um eins erniedrigt, da eine Referenz durch die Funktionsanwendung aufgelöst worden ist.

$$
\begin{array}{l}
\mathcal{CC}[\![\ \textit{type name};\ \mathrm{R}\ ]\!] \\
\quad \Longrightarrow \left\{ \begin{array}{lcl}
\texttt{DECL\_ARRAY(}\ \textit{name, base, } s_0, \ldots, s_n\ \texttt{);} & : & \textit{name} \in Var_{array} \setminus Var_{unq} \\
\mathcal{CC}[\![\ \mathrm{R}\ ]\!] & & \wedge\ \textit{type} \equiv \textit{base}[s_0, \ldots, s_n] \\
\\
\texttt{DECL\_UNQ\_ARRAY(}\ \textit{name, base, } s_0, \ldots, s_n\ \texttt{);} & : & \textit{name} \in Var_{array} \cap Var_{unq} \\
\mathcal{CC}[\![\ \mathrm{R}\ ]\!] & & \wedge\ \textit{type} \equiv \textit{base}[s_0, \ldots, s_n] \\
\\
\textit{type name};\ \ \mathcal{CC}[\![\ \mathrm{R}\ ]\!] & : & \text{sonst}
\end{array} \right.
\end{array}
$$

Bei der Kompilation von Variablen-Deklarationen muß an Stelle von **DECL_ARRAY** ggf. der ICM **DECL_UNQ_ARRAY** verwendet werden.

$$
\mathcal{CC}[\![\ \texttt{return(}\ \textit{retvars}\ \texttt{);}\ ]\!] \Longrightarrow \texttt{FUN\_RET(}\ \mathcal{CC}ret[\![\ \textit{retvars}\ ]\!]\ \texttt{);}
$$

$$
\begin{array}{l}
\mathcal{CC}ret[\![\ \textit{name},\ \mathrm{R}\ ]\!] \\
\quad \Longrightarrow \left\{ \begin{array}{lcl}
\mathcal{CC}ret[\![\mathrm{R}]\!] & : & \mathcal{W}as\mathcal{GO}(\textit{name}) \\
\texttt{inout},\ \textit{name},\ \mathcal{CC}ret[\![\mathrm{R}]\!] & : & \mathcal{W}as\mathcal{CBR}(\textit{name}) \\
\texttt{out\_rc},\ \textit{name},\ \mathcal{CC}ret[\![\mathrm{R}]\!] & : & \textit{name} \in Var_{array} \setminus Var_{unq} \\
\texttt{out},\ \textit{name},\ \mathcal{CC}ret[\![\mathrm{R}]\!] & : & \text{sonst}
\end{array} \right.
\end{array}
$$

Bei der Kompilation einer **return**-Anweisung werden alle Variablen, die mittelbar durch die Eliminierung globaler Objekte hinzugefügt worden sind, ignoriert. Solche, die auf die Eliminierung ursprünglicher Referenz-Parameter zurückgehen, erhalten die Marke `inout`.

$$
\begin{array}{l}
\mathcal{CC}[\![\ a\ \texttt{=}\ b\texttt{;}\ \mathrm{R}\ ]\!] \\
\quad \Longrightarrow \left\{ \begin{array}{lcl}
\texttt{ASSIGN\_RC(}\ b,\ a\texttt{);} \\
\mathcal{CC}rc[\![\ a\ ]\!][\![\ \mathrm{R}\ ]\!]\ \ \mathcal{CC}[\![\ \mathrm{R}\ ]\!] & : & a,b \in Var_{array} \setminus Var_{unq} \\
\\
a\ \texttt{=}\ b\texttt{;}\quad \mathcal{CC}rc[\![\ a\ ]\!][\![\ \mathrm{R}\ ]\!]\ \ \mathcal{CC}[\![\ \mathrm{R}\ ]\!] & : & \text{sonst}
\end{array} \right.
\end{array}
$$

Bei der Kompilation einer aus zwei Variablen bestehenden Zuweisung muß an Stelle einer entspechenden Zuweisung in C bei Arrays mit dynamischer Referenzzählung der ICM **ASSIGN_RC** verwendet werden. Zusätzlich muß ggf. der Referenzzähler der zugewiesenen Variable verändert oder ein Array-Objekt gelöscht werden.

$$
\begin{array}{l}
\mathcal{CC}[\![\ \textit{varlist}\ \texttt{=}\ \textit{fun}\texttt{(}\ \textit{arglist}\ \texttt{);}\ \mathrm{R}\ ]\!] \\
\quad \Longrightarrow\ \texttt{FUN\_AP(}\ \textit{fun},\ \mathcal{CC}apout[\![\ \textit{varlist}\ ]\!],\ \mathcal{CC}apin[\![\ \textit{arglist}\ ]\!]\ \texttt{);} \\
\qquad \mathcal{CC}vars[\![\ \textit{varlist}\ ]\!][\![\ \mathrm{R}\ ]\!]\ \ \mathcal{CC}[\![\ \mathrm{R}\ ]\!]
\end{array}
$$

$$\mathcal{CC}vars[\![\ var,\ \mathrm{R}\ ]\!][\![\ instrs\ ]\!]$$
$$\Longrightarrow\ \mathcal{CC}rc[\![\ var\ ]\!][\![\ instrs\ ]\!]\quad \mathcal{CC}vars[\![\ R\ ]\!][\![\ instrs\ ]\!]$$

$$\mathcal{CC}apout[\![\ var,\ \mathrm{R}\ ]\!]$$
$$\Longrightarrow \begin{cases} \mathcal{CC}apout[\![\ \mathrm{R}\ ]\!] & : \quad \mathcal{W}as\mathcal{CBR}(var) \vee \mathcal{W}as\mathcal{GO}(var) \\ \texttt{out\_rc},\ var,\ \mathcal{CC}apout[\![\ \mathrm{R}\ ]\!] & : \quad var \in Var_{array} \setminus Var_{unq} \\ \texttt{out},\ var,\ \mathcal{CC}apout[\![\ \mathrm{R}\ ]\!] & : \quad \text{sonst} \end{cases}$$

$$\mathcal{CC}apin[\![\ arg,\ \mathrm{R}\ ]\!]$$
$$\Longrightarrow \begin{cases} \mathcal{CC}apin[\![\ \mathrm{R}\ ]\!] & : \quad \mathcal{W}as\mathcal{GO}(arg) \\ \texttt{inout},\ arg,\ \mathcal{CC}apin[\![\ \mathrm{R}\ ]\!] & : \quad \mathcal{W}as\mathcal{CBR}(arg) \\ \texttt{in\_rc},\ arg,\ \mathcal{CC}apin[\![\ \mathrm{R}\ ]\!] & : \quad arg \in Var_{array} \setminus Var_{unq} \\ \texttt{in},\ arg,\ \mathcal{CC}apin[\![\ \mathrm{R}\ ]\!] & : \quad \text{sonst} \end{cases}$$

Bei der Kompilation einer Zuweisung, deren rechte Seite eine Funktionsanwendung enthält, werden ebenfalls sämtliche durch die Objekt-Behandlung hinzugefügten Variablen und Argumente ignoriert. Solche Argumente, die mit ursprünglichen Referenz-Parametern der angewandten Funktion korrespondieren, erhalten die Marke `inout`. Zusätzlich müssen die Referenzzähler der zugewiesenen Variablen verändert bzw. Array-Objekte gelöscht werden.

$$\mathcal{CC}[\![\ a\ \texttt{=}\ \texttt{from\_}class\texttt{(}b\texttt{);}\ \mathrm{R}\ ]\!]$$
$$\Longrightarrow \begin{cases} a\ \texttt{=}\ b\texttt{;} \\ \texttt{ALLOC\_RC(}a\texttt{);} \\ \texttt{SET\_RC(}a,\ X\texttt{);} \\ \mathcal{CC}[\![\ \mathrm{R}\ ]\!] \end{cases} : \quad a,b \in Var_{array} \wedge\ X = \mathcal{R}ef_a[\![\ \mathrm{R}\ ]\!]$$
$$a\ \texttt{=}\ b\texttt{;}\quad \mathcal{CC}[\![\ \mathrm{R}\ ]\!] \quad : \quad \text{sonst}$$

$$\mathcal{CC}[\![\ a\ \texttt{=}\ \texttt{to\_}class\texttt{(}b\texttt{);}\ \mathrm{R}\ ]\!]$$
$$\Longrightarrow \begin{cases} \texttt{MAKE\_UNQ\_ARRAY(}b,\ a\texttt{);}\quad \mathcal{CC}[\![\ \mathrm{R}\ ]\!] & : \quad a,b \in Var_{array} \\ a\ \texttt{=}\ b\texttt{;}\quad \mathcal{CC}[\![\ \mathrm{R}\ ]\!] & : \quad \text{sonst} \end{cases}$$

Anwendungen der generischen Klassen-Konvertierungsfunktionen (vgl. Abschnitt 4.3.5) müssen gesondert betrachtet werden. Liegt dem betreffenden Klassentyp kein Arraytyp zugrunde, so können sie durch einfache Zuweisungen ersetzt werden. Bei der Konvertierung eines Array-Objektes muß dagegen zusätzlich ein Referenzzähler erzeugt und initialisiert werden. In der umgekehrten Richtung muß die Uniqueness-Eigenschaft sichergestellt werden, d.h. der betreffende Speicherbereich darf nicht mehrfach referenziert werden. Zu diesem Zweck muß zur Laufzeit der Referenzzähler überprüft werden. Hat er den Wert eins, so bestehen keine weiteren Referenzen. Die Funktionsanwendung resultiert in einer einfachen Zuweisung und der Referenzzähler kann gelöscht werden. Anderenfalls muß das Array kopiert werden.

7.13.5 Importierte Symbole

Im Rahmen der Abbildung des SAC-Modulsystems auf die Sprache C müssen für importierte Funktionen und globale Objekte Deklarationen erzeugt werden. Diese werden in der Sprache C durch

das Schlüsselwort **extern** eingeleitet. Das Kompilationsschema $\mathcal{CC}$ beschreibt die Abbildung eines importierten globalen Objektes nach C. Für globale Objekte, deren Klassentyp ein Arraytyp zugrundeliegt, wird zu diesem Zweck der neue ICM **DECL_EXTERN_ARRAY(** *name, base, dim* **)** eingeführt.

$$\mathcal{CC}[\![\ \texttt{objdef}\ \textit{type}\ \textit{name}\ \texttt{;}\ ]\!] \Longrightarrow \begin{cases} \texttt{DECL\_EXTERN\_ARRAY(}\ \textit{name, base, dim}\ \texttt{);} & : \quad \textit{type} \equiv \textit{base}[s_0, \ldots, s_n] \\ \textit{extern type name;} & : \quad \text{sonst} \end{cases}$$

Importierte Funktionen werden unmittelbar auf entsprechende Extern-Deklarationen in C abgebildet. Dazu wird analog zu Funktionsdefinitionen der ICM **FUN_DEC** verwendet. Lediglich bei der Kompilation der Resultatstypliste muß der Tatsache Rechnung getragen werden, daß keine Variablen aus einer **return**-Anweisung zur Verfügung stehen. Die betreffenden Argumente des ICM erhalten $\sqcup$ als Wert.

$$\mathcal{CC}[\![\ \textit{retlist}\ \textit{fun}\texttt{(}\ \textit{paramlist}\ \texttt{);}\ ]\!] \Longrightarrow \texttt{extern FUN\_DEC(}\ \textit{fun}, \mathcal{CC}\textit{out}_{ext}[\![\ \textit{retlist}\ ]\!], \mathcal{CC}\textit{in}[\![\ \textit{paramlist}\ ]\!]\texttt{);}$$

7.13.6 Die Schnittstelle zu anderen Sprachen

Die Verwendung externer Module im Rahmen der in Kapitel 5 vorgestellten Schnittstelle zu anderen Sprachen muß bei der Code-Erzeugung besonders berücksichtigt werden. Im wesentlichen sind die folgenden Erweiterungen notwendig:

- Die für Arrays durchgeführte Speicherverwaltung muß auf alle externen impliziten Typen ausgedehnt werden. Dies schließt eine dynamische Referenzzählung für Typen ohne Uniqueness-Attribut ein.
- Bei der Kompilation von Funktionsdeklarationen müssen die erweiterten Möglichkeiten, die aus der Angabe von Pragmas oder der Zulässigkeit variabler Parameter- und Rückgabewertlisten resultieren, Berücksichtigung finden.
- Anwendungen externer Funktionen müssen analog zu den jeweiligen Deklarationen kompiliert werden. Im Gegensatz zu SAC-Funktionen führen sie standardmäßig keine Speicherverwaltung durch. Diese muß vollständig in die Funktionsanwendung integriert werden.

Die Speicherverwaltung für externe implizite Typen mit oder ohne Uniqueness-Attribut wird exakt in derselben Weise durchgeführt wie bei Arrays. Lediglich an den Stellen, an denen spezifisches Wissen über die Repräsentation einer Datenstruktur erforderlich ist, sind zusätzliche Intermediate Code Macros erforderlich. Diese treten im Kompilat an die Stelle des jeweiligen Array-ICMs. Im einzelnen handelt es sich dabei um:

DECL_HIDDEN(*H*) an Stelle von DECL_ARRAY,

DECL_UNQ_HIDDEN(*H*) an Stelle von DECL_UNQ_ARRAY,

DEC_RC_FREE_HIDDEN(*H, n, freefun*) an Stelle von DEC_RC_FREE_ARRAY,

FREE_HIDDEN(*H, freefun*) an Stelle von FREE_ARRAY und

MAKE_UNQ_HIDDEN(*H, I, copyfun*) an Stelle von MAKE_UNQ_ARRAY.

Da die Kompilation einer externen Funktionsdeklaration aufgrund der weitreichenden Möglichkeiten der Schnittstelle sehr komplex ist, wird an dieser Stelle kein allgemeines Kompilationsschema $\mathcal{CC}$ angegeben, sondern die Kompilation an einem Beispiel illustriert. Gegeben sei die folgende Deklaration:

```
   ...
int[10], int ExtFun(hidden &A, int[10] B, int[10] C);
  #pragma linkname "MyFun"
  #pragma linksign [1, 0, 2, 1, 3]

float[5], float ExtFun(float[5] A, float B);
  #pragma linkname "YourFun"
  #pragma effect object
  #pragma linksign [1, 3, 2, 3]
  #pragma refcounting [2]
   ...
```

Das Pragma **effect** wird durch die Objekt-Behandlung ausgewertet. Sie sorgt für eine korrekte Abbildung des dadurch angegebenen Seiteneffektes auf das globale Objekt `object`. Da sämtliche im Rahmen der Objekt-Behandlung zusätzlich eingeführten Parameter und Resultatswerte bei der Code-Generierung ignoriert werden, kann an dieser Stelle auf eine Darstellung der Auswirkungen der Objekt-Behandlung verzichtet werden. Bei der Erzeugung von ICM-Code werden die beiden Funktionsdeklarationen wie folgt kompiliert:

```
   ...
extern  CFUN_DEC( MyFun, int, upd, int[], B, upd, hidden, A, in, int[], C);
extern  CFUN_DEC( YourFun, , out, float[], , in_rc, float[], A, upd, float, B);
   ...
```

Der ICM **CFUN_DEC**($f, \alpha, tag_1, \alpha_1, a_1, \ldots, tag_n, \alpha_n, a_n$) entspricht dem ICM **FUN_DEC**, trägt jedoch den spezifischen Anforderungen der Schnittstelle Rechnung. Der echte Resultatstyp der angewandten C-Funktion wird explizit angegeben (α). Die weiteren Argumente definieren die Parameter der C-Funktion in genau dieser Reihenfolge. Die zusätzliche Marke `upd` bezeichnet Parameter, bei denen das übergebene Argument modifiziert wird.

Die Überladung der Funktion `ExtFun` wird durch Umbenennung aufgelöst (Pragma **linkname**). Das Pragma **linksign** bestimmt den zweiten Resultatstyp von `ExtFun (MyFun)` nämlich `int` als Resultatstyp der C-Funktion `MyFun`. Der erste Resultatswert und der zweite Parameter werden auf einen gemeinsamen `upd`-Parameter abgebildet. Referenz-Parameter werden ebenfalls auf `upd`-Parameter des ICMs abgebildet. Das Pragma **refcounting** gibt an, daß die C-Funktion `YourFun` die Speicherverwaltung für ihren ersten Parameter (`float[5]`) selbst durchführt. Dem wird durch die Marke `in_rc` Rechnung getragen.

```
   ...
extern  int MyFun( int *, void *A, int *C);
extern  void YourFun( float **, float *A, int *A_rc, float *);
   ...
```

Die Auflösung der Intermediate Code Macros (ICM) führt zu den obigen beiden Funktionsdeklarationen. Externe implizite Typen (`hidden`) werden auf den C-Typ `void*` abgebildet, Arrays auf Zeiger auf ihren jeweiligen Basistyp. Parameter, durch die zusätzliche Resultatswerte modelliert werden, führen zu Zeigern auf den jeweiligen Typ. Parameter, für die eine Funktion die notwendige Speicherverwaltung selbst durchführt, resultieren in jeweils zwei Parametern der C-Funktion. Der erste steht für das eigentliche Datenobjekt, der zweite für den Referenzzähler. Die folgende Tabelle gibt einen vollständigen Überblick über die Zuordnung von C-Typen zu ICM-Parametern. Dabei steht α für einen primitiven Typ.

	α	α`[]`	`hidden`
`in`	α	α`*`	`void*`
`in_rc`		α`*, int*`	`void*, int*`
`out`	α`*`	α`**`	`void**`
`out_rc`		α`**, int**`	`void**, int**`
`upd`	α`*`	α`*`	`void*`

Um auch die Kompilation der Anwendung einer externen Funktion zu veranschaulichen, wird das obige Beispiel entsprechend erweitert.

```
{ ...
  Z, i = ExtFun(A, B, C);
  Y, k = ExtFun(D, k);
  ... }
```

Es sei dabei vorausgesetzt, daß die Argumente der beiden Anwendungen jeweils von korrektem Typ sind und ihnen vorher Werte zugewiesen werden. Ferner werden die Resulte der Anwendungen im nachfolgenden Teil des Blockes verwendet.

```
{ ...
  MAKE_UNQ_ARRAY(B, Z);
  CFUN_AP( MyFun, i, upd, Z, upd, A, in, C);
  ALLOC_RC(Z);
  SET_RC(Z, Ref_Z[[...]]);
  DEC_RC_FREE_ARRAY(C, 1);

  CFUN_AP( YourFun, , out, Y, in_rc, D, upd, k);
  ALLOC_RC(Y);
  SET_RC(Y, Ref_Y[[...]]);
  ... }
```

Der neue ICM **CFUN_AP**$(f, r, tag_1, r_1, \ldots, tag_n, r_n)$ tritt analog zu **CFUN_DEC** an die Stelle des ICMs **FUN_AP**. Da `MyFun` das Array `B` modifiziert, muß vorher durch den ICM **MAKE_UNQ_ARRAY** sichergestellt werden, daß es zur Laufzeit keine weiteren Referenzen auf

den betreffenden Speicherbereich gibt. Nach der Anwendung von `MyFun` muß ein neuer Referenzzähler für `Z` angelegt und mit der Anzahl der Referenzn im verbleibenden Block initialisiert werden. Durch die Funktionsanwendung wird eine Referenz auf das Array `C` aufgelöst. Da `MyFun` keine Speicherverwaltung für `C` durchführt, muß im Anschluß an die Anwendung von `MyFun` der Referenzzähler dekrementiert und der Speicherbereich ggf. freigegeben werden. Da die Funktion `YourFun` für das Array `D` die Speicherverwaltung selbst durchführt (`in_rc`), entfällt eine entsprechende Behandlung nach deren Anwendung. Dies gilt jedoch nicht für das als Resultat zurückgelieferte Array `Y`, weshalb für `Y` ein Referenzzähler erzeugt und initialisiert werden muß.

```
{ ...
  i = MyFun(Z, A, C);
  ...
  YourFun(&Y, D, RC(D), &k);
  ... }
```

Der ICM **CFUN_AP** wird letztlich durch eine C-Funktionsanwendung ersetzt. Bei OUT-Parametern (`out/out_rc`) erhält die Funktion eine Adresse als Argument. Das gleiche gilt für primitive Datenobjekte, die modifiziert werden sollen (`upd`). Für einen Parameter mit Referenzzählung (`in_rc`/`out_rc`) erhält die Funktion neben dem eigentlichen Argument zusätzlich dessen Referenzzähler.

Kapitel 8

Zusammenfassung

Die vorliegende Arbeit befaßt sich mit der Integration eines Modul- und Klassen-Konzeptes in die funktionale Programmiersprache SAC. Die 1994 erstmals vorgeschlagene Sprache berücksichtigt in besonderem Maße die Anforderungen der Entwicklung technisch-naturwissenschaftlicher Anwendungen auf der Basis numerischer Verfahren. Dies findet seinen Ausdruck u.a. in einem umfangreichen Array-Konzept, das auch die dimensionsunabhängige Spezifikation von Algorithmen zuläßt. Syntaktisch orientiert sich SAC stark an der imperativen Programmiersprache C. Auf diese Weise sollen die mit einer Umstellung vom imperativen auf das funktionale Paradigma verbundenen Schwierigkeiten minimiert und so die Akzeptanz der Sprache SAC erhöht werden.

Das Modul-Konzept von SAC bildet die Grundlage für die strukturierte Entwicklung umfangreicher Anwendungsprogramme. Neben expliziten Mechanismen zum Im- und Exportieren von Symbolen umfaßt es getrennte Namensräume, abstrakte Datentypen sowie die Möglichkeit zur separaten Kompilation einzelner Module. Durch das spezielle Format der SAC-Bibliothek stehen dem Compiler Informationen über importierte Symbole zur Verfügung, welche über die in der betreffenden Modul-Deklaration gemachten Angaben hinausgehen. Auf diese Weise können negative Einflüsse der Modularisierung auf das Laufzeitverhalten eines Programmes weitgehend ausgeschlossen werden.

Aufbauend auf dem Modul-Konzept erlaubt das Klassen-Konzept von SAC einen einfachen und intuitiv verständlichen Umgang mit Zuständen. Die Integration von Zuständen in das funktionale Paradigma erfolgt dabei auf der konzeptuellen Grundlage des Uniqueness-Typing. Gegenüber dem Programmierer wird dieses jedoch weitgehend versteckt. Stattdessen wird auf der Sprachebene zwischen funktionalen Ausdrücken und Zuständen (Objekten) unterschieden. Bei Objekten besitzt der Programmierer die vollständige Kontrolle über das Erzeugen, Löschen oder Kopieren von Datenstrukturen. Syntaktische Erweiterungen wie die Kurzschreibweisen des Call-by-Reference-Mechanismus und der globalen Objekte erlauben die Spezifikation von Operationen auf Zuständen in einer Form, die der imperativer Sprachen sehr ähnlich ist. Durch Auflösung dieser Kurzschreibweisen während der Kompilation können referentielle Transparenz und Church-Rosser-Eigenschaft trotzdem garantiert werden. Diese stehen für Optimierungen und die Identifikation nebenläufig ausführbarer Programmteile uneingeschränkt zur Verfügung. Bei der Erzeugung sequentiellen C-Codes wird zur Steigerung der Effizienz auf ursprüngliche Anwendungen von Call-by-Reference-Mechanismus und globalen Objekten zurückgegriffen und diese unmittelbar auf die entsprechenden Elemente der Sprache C abgebildet.

Die wichtigste Anwendung des Klassen-Konzeptes von SAC besteht in der Bereitstellung von Ein-/Ausgabe-Mechanismen. Standard-Klassen ermöglichen die Spezifikation von SAC-Programmen, die mit einem Dateisystem oder einem Terminal interagieren. Der Grundkonzeption von SAC folgend

orientieren sich diese Standard-Klassen stark an der C-Standard-Bibliothek `stdio`. Die Tatsache, daß nicht nur deren Funktionalität SAC-Programmen nahezu uneingeschränkt zur Verfügung gestellt werden kann, sondern auch die Form der Ein-/Ausgabe-Programmierung sich praktisch nicht von dem in C Üblichen unterscheidet, unterstreicht die Leistungsfähigkeit des Klassen-Konzeptes. Es gelingt dabei, die grundlegenden Vorteile des funktionalen Paradigmas, wie referentielle Transparenz oder Church-Rosser-Eigenschaft, mit der in imperativen Sprachen üblichen, einfach anwendbaren und intuitiv verständlichen Form der Spezifikation von Ein-/Ausgabe-Operationen zu verknüpfen.

Auf der Basis des Modul- und Klassen-Konzeptes besitzt SAC eine flexible Schnittstelle zu anderen Programmiersprachen, insbesondere zu C. Externe Modulen bzw. Klassen können in jeder Programmiersprache implementiert werden, die zum C-Link-Mechanismus kompatibel ist. Die jeweilige Deklaration dient dem SAC-Compiler als vollständige und ausschließliche Beschreibung der Funktionalität. Die Schnittstelle erlaubt die Anwendung externer Funktionen auf SAC-Datenobjekte, wie z.B. Arrays, ebenso wie die Verwendung beliebiger externer Datenstrukturen in SAC-Programmen. Die Flexibilität der Schnittstelle wird durch die Erweiterung der Modul- und Klassen-Deklarationen um Pragmas zusätzlich erhöht. Mit ihrer Hilfe lassen sich selbst verschiedene Formen von Seiteneffekte ausführenden Funktionen korrekt in die funktionale Programmiersprache SAC abbilden. Ihre Leistungsfähigkeit stellt die Schnittstelle im Rahmen der C-Implementierung der Standard-Klassen zur Ein-/Ausgabe unter Beweis.

Weiterentwicklungen des in der vorliegenden Arbeit beschriebenen Modul- und Klassen-Konzeptes sind in verschiedene Richtungen denkbar. Im Rahmen des Modul-Konzeptes lassen sich die eher rudimentär ausgeprägten Möglichkeiten zum selektiven Importieren von Symbolen erweitern. Gerade bei umfangreichen Modulen ist das Verfahren der expliziten Angabe sämtlicher zu importierender Symbole ungeeignet, wenn lediglich wenige Symbole nicht importiert werden sollen.

Das Klassen-Konzept von SAC sequentialisiert sämtliche Zugriffe auf ein Objekt. Dabei wird nicht zwischen lesenden und schreibenden Zugriffen differenziert. Tatsächlich ist diese strenge Sequentialisierung jedoch nicht erforderlich. Zwischen jeweils zwei schreibenden Zugriffen können lesende Zugriffe in beliebiger Reihenfolge erfolgen. Daraus ergibt sich ein bisher nicht genutztes Potential zur nebenläufigen Ausführung von Programmteilen. Eine andere Weiterentwicklung des Klassen-Konzeptes ist in Richtung einer erweiterten Adaption objekt-orienierter Mechanismen, wie z.B. Vererbung, denkbar.

Neben der Weiterentwicklung des Klassen-Konzeptes selbst sind weitere Anwendungen der damit zur Verfügung gestellten Funktionalität möglich. Ein-/Ausgabe beschränkt sich nicht auf die für SAC zunächst realisierte Form der Datei-orientierten Ein-/Ausgabe. Die Integration Stream-basierter Ein-/Ausgabe-Konzepte stellt ebenso eine interessante Anwendung für das Klassen-Konzept dar wie der Komplex Ereignis-gesteuerter Ein-/Ausgabe im Zusammenhang mit der Programmierung graphischer Benutzeroberflächen.

Anhang A

Beispiele für Module und Klassen

Das Modul „complex“

```
ModuleDec complex:

own:
{
explicit types:
  cplx=float[2];

functions:
  bool          ==(cplx a, cplx b);
  cplx          +(cplx a, cplx b);
  cplx          -(cplx a, cplx b);
  cplx          *(cplx a, cplx b);
  cplx          mkcplx(float a, float b);
  float         Re(cplx a);
  float         Im(cplx a);
  float, float  ReIm(cplx a);
}

Module complex:

typedef float[2] cplx;

inline bool ==(cplx a, cplx b)
{
  return((Re(a)==Re(b)) && (Im(a)==Im(b)));
}

inline cplx +(cplx a, cplx b)
{
  return((:cplx)(:float[2])a+(:float[2])b);
}
```

```
inline cplx -(cplx a, cplx b)
{
  return((:cplx)(:float[2])a-(:float[2])b);
}

inline cplx *(cplx a, cplx b)
{
  return((:cplx)[Re(a)*Re(b)-Im(a)*Im(b), Re(a)*Im(b)+Im(a)*Re(b)]);
}

inline cplx mkcplx( float r, float i)
{
  return((:cplx)[r, i]);
}

inline float Re(cplx a)
{
  return(((:float[2])a)[0]);
}

inline float Im(cplx a)
{
  return(((:float[2])a)[1]);
}

inline float, float ReIm(cplx a)
{
  return(((:float[2])a)[0], ((:float[2])a)[1]);
}
```

Die Klasse „IntStack“

```
ClassDec IntStack:

own:
{
global objects:
  IntStack Stack;

functions:
  IntStack createIntStack();
  void     push(IntStack &stack, int a);
  void     push(int a);
  int      pop(IntStack &stack);
  int      pop();
  void     deleteIntStack(IntStack stack);
}
```

```
Class IntStack:

classtype int[100];

objdef IntStack Stack=createIntStack();

void push(IntStack &stack, int a)
{
  p = from_IntStack(stack);
  h = p[0];
  p = modarray(p, [h], a);
  p = modarray(p, [0], p[0]+1);
  stack = to_IntStack(p);
}

void push(int a)
{
  push(Stack, a);
}

int pop(IntStack &stack)
{
  p = from_IntStack(stack);
  h = p[0];
  z = p[[h-1]];
  p = modarray(p, [0], h-1);
  stack = to_IntStack(p);
  return(z);
}

int pop()
{
  return(pop(Stack));
}

IntStack createIntStack()
{
  A = with ([0] <= i < [100]) genarray([100], 0);
  A = modarray(A, [0], 1);
  return(to_IntStack(A));
}

void deleteIntStack(IntStack stack)
{
  p = from_IntStack(stack);
}
```

Anhang B

Standard-Klassen für Ein-/Ausgabe

Die Standard-Klasse „World“

```
ClassDec external World:

own:
{
global objects: World FileSys;
                World Terminal;
}
```

Die Standard-Klasse „Termfile“

```
ClassDec external TermFile:

import World:  {global objects: Terminal;}
import String: {implicit types: string;}

own:
{
global objects: TermFile stdin;
                TermFile stdout;
                TermFile stderr;

functions:

  void fprintf(TermFile &stream, string format, ... );

  void fprint(TermFile &stream, int n);       #pragma linkname "fprintint"
  void fprint(TermFile &stream, float n);     #pragma linkname "fprintfloat"
  void fprint(TermFile &stream, double n);    #pragma linkname "fprintdouble"
  void fprint(TermFile &stream, bool b);      #pragma linkname "fprintbool"
  void fprint(TermFile &stream, char c);      #pragma linkname "fprintchar"
  void fprint(TermFile &stream, string s);    #pragma linkname "fprintstring"
```

```
  void printf(string format, ... );     #pragma effect stdout

  void print(int n);      #pragma effect stdout    #pragma linkname "printint"
  void print(float n);    #pragma effect stdout    #pragma linkname "printfloat"
    ...

  int, ...     fscanf(TermFile &stream, string format);

  bool, int    fscanint(TermFile &stream);
  bool, float  fscanfloat(TermFile &stream);
  bool, double fscandouble(TermFile &stream);
  bool, bool   fscanbool(TermFile &stream);
  bool, char   fscanchar(TermFile &stream);
  bool, string fscanstring(TermFile &stream);

  int, ...     scanf(string format);    #pragma effect stdin

  bool, int    scanint();               #pragma effect stdin
  bool, float  scanfloat();             #pragma effect stdin
    ...

  char fgetc(TermFile &stream);
  void fputc(char c, TermFile &stream);
  void ungetc(char c, TermFile &stream);

  char getchar();          #pragma effect stdin
  void putchar(char c);    #pragma effect stdout

  void fflush(TermFile &stream);
  void fpurge(TermFile &stream);
    ...
}
```

Die Standard-Klasse „File“

```
Classdec external File:

import World:    {global objects: FileSys;}
import String:   {implicit types: string;}
import SysErr:   {explicit types: syserr;}

own:
{
functions:

  syserr, File fopen(string name, string mode);   #pragma effect FileSys
  void         fclose(File stream);               #pragma effect FileSys

  void fprintf(File &stream, string format, ... );
  void fprint(File &stream, int n);       #pragma linkname "fprintint"
  void fprint(File &stream, float n);     #pragma linkname "fprintfloat"
    ...

  int, ...    fscanf(File &stream, string format);
  bool, int   fscanint(File &stream);
  bool, float fscanfloat(File &stream);
    ...
```

```
  char fgetc(File &stream);
  void fputc(char c, File &stream);
  void ungetc(char c, File &stream);

  bool feof(File &stream);
  void fflush(File &stream);
  void fpurge(File &stream);
  void fseek(File &stream, int offset, string mode);
  int  ftell(File &stream);
  void rewind(File &stream);

  void         remove(string pathname);              #pragma effect FileSys
  string       tmpnam();                             #pragma effect FileSys
  string       tempnam(string dir, string prefix);   #pragma effect FileSys
  syserr, File tmpfile();                            #pragma effect FileSys
  string       mktemp(string template);              #pragma effect FileSys
                                                     #pragma linksign [1,1]
    ...
}
```

Das Standard-Modul „SysErr“

```
ModuleDec external SysErr:

import String:   {implicit types: string;}
import TermFile: {global objects: stderr;}

own:
{
explicit types:  syserr = int;

functions:
  string strerror(syserr error);
  void   perror(string s, syserr error);    #pragma effect stderr
  void   exit(int exitcode);                #pragma effect stderr
  bool   fail(syserr error);
  bool   succ(syserr error);
}
```

Literaturverzeichnis

[AP92] P. Achten and R. Plasmeijer: *The Beauty and the Beast.* University of Nijmegen, 1992.

[AP95] P. Achten and R. Plasmeijer: *The ins and outs of Clean I/O.* Journal of Functional Programming, Vol. 5(1), 1995, pp. 81–110.

[Bar81] H. P. Barendregt: *The Lambda Calculus, Its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics, Vol. 103. North-Holland, 1981.

[BvEvLP87] T.H. Brus, M.C. van Eekelen, M.O. van Leer, and M.J. Plasmeijer: *CLEAN: A Language for Functional Graph Rewriting.* In G. Kahn (Ed.): FPCA'87, LNCS, Vol. 274. Springer, 1987.

[BWW90] J. Backus, J. Williams, and E. Wimmers: *An Introduction to the Programming Language FL.* In D.A. Turner (Ed.): Research Topics in Functional Programming. Addison-Wesley, 1990, pp. 219–247.

[Can92] D.C. Cann: *Retire Fortran? A Debate Rekindled.* Communications of the ACM, Vol. 35(8), 1992, pp. 81–89.

[Can93] D.C. Cann: *The Optimizing SISAL Compiler: Version 12.0.* Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 1993. part of the SISAL distribution.

[CH93] M. Carlsson and T. Hallgren: *FUDGETS - A Graphical User Interface in a Lazy Functional Language.* In FPCA '93, Copenhagen. ACM Press, 1993, pp. 321–330.

[Chu32] A. Church: *A Set of Postulates for the Foundation of Logic.* ANM, Vol. 33, 1932, pp. 346–366.

[CW85] L. Cardelli and P. Wegner: *On Understanding Types, Data Abstraction, and Polymorphism.* Computing Surveys, Vol. 17(4), 1985, pp. 471–522.

[Dwe89] A. Dwelly: *Functions and Dynamic User Interfaces.* In FPCA '89, London, 1989, pp. 371–381.

[Feo92] J.T. Feo: *SISAL.* Technical Report UCRL-JC-110915, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 1992.

[For94] High Performance Fortran Forum: *High Performance Fortran language specification V1.1*, 1994.

[Fra91] J. Frankel: C^* *language reference manual.* Thinking Machines Corp., Cambridge, MA, 1991.

[G+93] A. Geist et al.: *PVM3: User's Guide and Reference Manual.* Oak Ridge National Laboratory, Oak Ridge, TN, 1993.

[Gor92] A.D. Gordon: *Functional Programming and Input/Output.* PhD thesis, University of Cambridge, England, 1992. Available as Technical Report No.285.

[GS95] C. Grelck and S.B. Scholz: *Classes and Objects as Basis for I/O in SAC.* In T. Johnsson (Ed.): Proc. 7th International Workshop on the Implementation of Functional Languages '95, Båstad, Sweden. Chalmers University, 1995, pp. 30–44.

[H+95] K. Hammond et al.: *Report on the Programming Language Haskell.* University of Glasgow, 1995. Version 1.3.

[HS89] P. Hudak and R.S. Sundaresh: *On the Expressiveness of Purely Functional I/O Systems.* Technical report, Yale University, 1989.

[Ive62] K.E. Iverson: *A Programming Language.* Wiley, New York, 1962.

[JJ93] M.A. Jenkins and W.H. Jenkins: *The Q'Nial Language and Reference Manuals.* Nial Systems Ltd., Ottawa, Canada, 1993.

[JW74] K. Jensen and N. Wirth: *Pascal User Manual and Report.* Springer, 1974.

[JW93] S.L. Peyton Jones and P. Wadler: *Imperative functional programming.* In POPL '93, New Orleans, LA. ACM Press, 1993.

[Klu94] W. Kluge: *A User's Guide for the Reduction System π-RED.* Internal Report 9419, University of Kiel, 1994.

[KP92] D.J. King and P.Wadler: *Combining Monads.* In Proceedings of the Fifth Annual Glasgow Workshop on Functional Programming. Springer Verlag, 1992.

[KR88] B.W. Kernighan and D.M. Ritchie: *The C Programming Language.* Prentice-Hall, 1988. 2nd edition.

[LJ94] J. Launchbury and S. Peyton Jones: *Lazy Functional State Threads.* In Programming Languages Design and Implementation. ACM Press, 1994.

[MT94] L. Mullin and S. Thibault: *A Reduction Semantics for Array Expressions: The PSI Compiler.* Technical Report CSC-94-05, University of Missouri, Rolla, 1994.

[Mul88] L.M. Restifo Mullin: *A Mathematics of Arrays.* PhD thesis, Syracuse University, 1988.

[OCA86] R.R. Oldehoeft, D.C. Cann, and S.J. Allan: *SISAL: Initial MIMD Performance Results.* In W. Händler et al. (Eds.): CONPAR '86, LNCS, Vol. 237. Springer, 1986, pp. 120–127.

[Per91] N. Perry: *The Implementation of Practical Functional Programming Languages.* PhD thesis, Imperial College, London, 1991.

[PvE93] R. Plasmeijer and M. van Eekelen: *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, 1993. ISBN 0-201-41663-8.

[PvE95] M.J. Plasmeijer and M. van Eekelen: *Concurrent Clean 1.0 Language Report.* University of Nijmegen, 1995.

[QRM+87] D. Mac Queen, R.Harper, R. Milner, et al.: *Functional Programming in ML.* Lfcs education, University of Edinburgh, 1987.

[Ros84] J.B. Rosser: *Highlights of the History of the Lambda Calculus.* Annals of the History of Computing, Vol. 6(4), 1984, pp. 337–349.

[SBvEP93] S. Smetsers, E. Barendsen, M. van Eekelen, and R. Plasmeijer: *Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs.* Technical report, University of Nijmegen, 1993.

[Sch94] S.-B. Scholz: *Single Assignment C – Functional Programming Using Imperative Style.* In John Glauert (Ed.): Proc. 6th International Workshop on the Implementation of Functional Languages '94, Norwich, UK. University of East Anglia, 1994.

[Sie95] A. Sievers: *Maschinenunabhängige Optimierungen eines Compilers für die funktionale Programmiersprache SAC.* Diplomarbeit, Institut für Informatik und praktische Mathematik, Universität Kiel, 1995.

[Str91] B. Stroustrup: *The C++ Programming Language.* Addison-Wesley, 1991. 2nd edition.

[Tho90] S. Thompson: *Interactive Functional Programs. A Method and a Formal Semantics.* In D.A. Turner (Ed.): Research Topics in Functional Programming. Addison-Wesley, 1990, pp. 249–285.

[Tur85] D.A. Turner: *Miranda: a Non-Strict Functional Language with Polymorphic Types.* In IFIP '85, Nancy, France, LNCS, Vol. 201. Springer, 1985.

[Tur90] D.A. Turner: *An Approach to Functional Operating Systems.* In D.A. Turner (Ed.): Research Topics in Functional Programming. Addison-Wesley, 1990, pp. 199–217.

[Wad90] P. Wadler: *Linear types can change the world!* In M. Broy and C.B. Jones (Eds.): Programming Concepts and Methods. North Holland, 1990.

[Wad92a] P. Wadler: *Comprehending Monads.* Mathematical Structures in Computer Science, Vol. 2(4), 1992.

[Wad92b] P. Wadler: *The essence of functional programming.* In POPL '92, Albuquerque, NM. ACM Press, 1992.

[Weh85] H. Wehnes: *FORTRAN-77: Strukturierte Programmierung mit FORTRAN-77.* Carl Hanser Verlag, 1985. 4.Auflage.

[Wir85] N. Wirth: *Programming in MODULA-2.* Springer, 1985.

[Wol95] H. Wolf: *SAC→C : Ein Basiscompiler für die funktionale Programmiersprache SAC.* Diplomarbeit, Institut für Informatik und praktische Mathematik, Universität Kiel, 1995.

[WW88] J. Williams and E. Wimmers: *Sacrificing Simplicity for Convenience: Where Do You Draw the Line ?* In POPL '88, San Diego, CA, 1988, pp. 169–179.

Hiermit versichere ich, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, den 30. April 1996