Single **A**ssignment **C**

Entwurf und Implementierung einer funktionalen C-Variante mit spezieller Unterstützung shape-invarianter Array-Operationen

Dissertation
zur Erlangung des Doktorgrades
der Technischen Fakultät
der Christian-Albrechts-Universität
zu Kiel

vorgelegt von

Sven-Bodo Scholz

Kiel,
im Oktober1996

Berichterstatter: Prof. Dr. Werner Kluge

.....

Tag der mündlichen Prüfung: .....

Zum Druck genehmigt: Kiel, den .....

.....

(Dekan)

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Funktionale Sprachen basieren auf kontextfreien Ersetzungssystemen wie verschiedenen Varianten des λ-Kalküls [Bar81, Plo74, HS86, CF58], Kombinator-Kalkülen [HS86, CF58, Tur79] oder speziellen Term- bzw. Graph-Ersetzungssystemen [Sta80, Ken84, BvEG$^+$90]. Die Bedeutung von in funktionalen Sprachen spezifizierten Programmen ergibt sich ausschließlich aus einer wiederholten Anwendung der Ersetzungsregeln und abstrahiert somit vollständig von der Architektur der ausführenden Maschine. Daraus ergeben sich mehrere in der Literatur [FH88, BW88, Bac78, GHT84, Hen80, Bur75, Klu92, Rea89, PvE93, Hug89, Hud89] häufig genannte Vorteile:

- Variablen repräsentieren Werte anstelle von zustandsbehafteten Speicherbereichen. Entsprechend konsumieren alle primitiven Operationen ihre Argumente vollständig und erzeugen einen Resultatwert, anstatt Speicherinhalte zu verändern. Dies hat insbesondere bei der Verwendung großer Datenstrukturen wie z.B. Arrays den Vorteil, daß sie vom Programmierer genauso gehandhabt werden können wie skalare Werte, so daß der Umgang mit derartigen Strukturen erleichtert wird.

- Zwischen Funktionen und Daten wird konzeptuell nicht unterschieden. Daraus ergibt sich die Möglichkeit, Funktionen als Argumente zu übergeben (Funktionen höherer Ordnung), Funktionen schrittweise mit Argumenten zu versorgen sowie Funktionen als Ergebnisse von Berechnungen zu generieren.

- Durch die Verwendung ungetypter Kalküle als Basis ist es möglich, benutzerdefinierte Funktionen auf Argumente unabhängig von deren Typ anzuwenden. Dies erlaubt eine Abstraktion gleichartiger Operationen auf verschiedenartigen Argumenten zu Funktionsdefinitionen und trägt somit zu einer erhöhten Wiederverwendbarkeit von Programmspezifikationen bei.

- Die kontextfreien Ersetzungen erlauben einen Umgang mit Programmspezifikationen wie mit mathematischen Gleichungen. Dies erleichtert nicht nur die Ableitung funktionaler Programme aus mathematischen Spezifikationen, sondern auch den formalen Nachweis deren Äquivalenz.
- Die unterliegenden Kalküle weisen die sog. Church-Rosser-Eigenschaft [Bar81, HS86, CF58, Ros84] auf, d.h. das Ergebnis einer Berechnung, falls diese terminiert, ist unabhängig von der Reihenfolge der einzelnen Ersetzungen. Daher sind funktionale Sprachen besonders für die nebenläufige Berechnung unabhängiger Programmteile geeignet.

Trotz dieser Vorteile finden funktionale Sprachen bisher nur wenig Verwendung in sog. Real-World-Anwendungen. Selbst bei numerischen Anwendungen, wo insbesondere die Vorteile im Umgang mit komplexen Datenstrukturen sowie die Vorteile in bezug auf die Identifizierung nebenläufig ausführbarer Programmteile die Verwendung funktionaler Sprachen begünstigen, dominiert der Einsatz imperativer Sprachen wie FORTRAN oder C. Als entscheidender Grund dafür werden zumeist Effizienzprobleme der funktionalen Sprachen angeführt [Can92, MKM84]. Die Ursachen dieser Probleme liegen darin, daß das im Vergleich zu imperativen Sprachen hohe Abstraktionsniveau funktionaler Sprachen komplexe Mechanismen zu deren vollständiger Realisierung erforderlich macht:

Um für alle Programme einer funktionalen Sprache, für die es eine terminierende Berechnungsfolge gibt, eine solche zu finden, ist es bei einigen Funktionsanwendungen erforderlich, daß die Argumente unausgewertet in den Rumpf der Funktion eingesetzt werden. Ihre Auswertung darf erst dann vorgenommen werden, wenn sichergestellt ist, daß sie zum Ergebnis der gesamten Berechnung beitragen (Normal-Order-Reduction).

Eine mehrfache Berechnung von Argumenten läßt sich zwar durch sog. Graph-Reduktionen [Jon87, Joh87, Aug87, JS89, PvE93, FW87, Tur79] vermeiden (Lazy-Evaluation), jedoch erfordert das Einsetzen unausgewerteter Argumente, falls diese im Verlauf weiterer Berechnungen ohnehin ausgewertet werden müssen, einen erheblichen Mehraufwand. Deshalb sind sog. Striktheitsanalysen [CJ85, HY86, Nöc93] entwickelt worden. Sie versuchen, statisch zu inferieren, welche Parameter einer Funktion auf jeden Fall bei der Berechnung einer Funktionsanwendung benötigt werden. In vielen Fällen ist dies jedoch nicht möglich.

Bei der Implementierung funktionaler Sprachen, die auf einem ungetypten Kalkül beruhen, werden durch die Einführung von Operationen auf Konstanten Typüberprüfungen zur Laufzeit erforderlich. Um diesen Mehraufwand zu vermeiden, verwenden die meisten funktionalen Sprachen statische polymorphe Typsysteme [CW85, Myc84, HHJW92, Tur85, PvE95, HAB+95], die auf dem sog. Hindley-Milner Typsystem [Mil87, HS86] basieren. Dies führt jedoch zu einer Beschränkung der legalen Programme auf die Menge der wohlgetypten Programme und damit zu Einschränkungen in bezug auf die Wiederverwendbarkeit von Programmen. So können

Programme, die z.B. Selbstapplikationen oder Listen mit Elementen unterschiedlichen Typs enthalten, nicht berechnet werden, obwohl es dafür sinnvolle Anwendungen gibt. Um diese Einschränkungen möglichst gering zu halten, bieten die meisten modernen Typsysteme sog. Typklassen [WB89, Blo91, HHJW92] zum Überladen primitiver Funktionen. Die Verwendung dieser Konstrukte führt jedoch wieder zu zusätzlichem Laufzeitaufwand für die Verwaltung und Dereferenzierung von Funktionstabellen, den sog. Dictionaries.

Um Funktionen schrittweise auf Argumente anwenden zu können, bedarf es eines Mechanismus zur Verwaltung von sog. Closures [Lan64, Jon87, Joh87]. Es handelt sich dabei um Abschlüsse von Funktionen mit Argumenten, auf die sie bereits angewendet wurden. Eine explizite Berechnung partiell angewandter Funktionen erfordert darüberhinaus Mechanismen zum korrekten Umgang mit relativ freien Variablen [Ber75, Ber76].

Um die Church-Rosser-Eigenschaft für eine nebenläufige Ausführung nutzen zu können, ist in funktionalen Sprachen die Reihenfolge der Berechnung von Programmen nicht vorgegeben. Die Integration von Mechanismen, die auf einer sequentiellen Abfolge von Zustandsmodifikationen beruhen, wie z.B. I/O-Operationen oder Methodenaufrufe im objektorientierten Programmiermodell, erfordern deshalb Hilfsmittel, mit denen statisch partielle Abhängigkeiten bei der Berechnung verschiedener Programmteile garantiert werden können. Einen Überblick über die wichtigsten Ansätze bieten hier [Gor92, HS89, JW93, Nob95, Ach96]. Obwohl es mit diesen Ansätzen möglich ist, auch I/O-intensive Anwendungen wie z.B. Editoren oder Tabellenkalkulationsprogramme funktional zu spezifizieren [dHRvE95], erlauben imperative Sprachen für dieses Anwendungsgebiet intuitivere Spezifikationen, da das den imperativen Sprachen unterliegende Ausführungsmodell auf Zustandstransformationen beruht.

Die Eigenschaft der funktionalen Sprachen, daß jede Funktion alle Argumente vollständig konsumiert und ein Resultat produziert, führt bei der Implementierung primitiver Operationen auf Datenstrukturen, die aus vielen Komponenten bestehen (z.B. Arrays), zu Problemen. Bei Array-Operationen, die nur einen Teil der Elemente eines Arrays modifizieren, muß konzeptuell eine Kopie des gesamten Arrays erstellt werden, die an den entsprechenden Elementpositionen verändert ist. Eine Möglichkeit zur Vermeidung dieses Aufwandes besteht darin, Arrays als zustandsbehaftete Datenstrukturen einzuführen [Lau93, LJ94, vG96]. Modifikationen einzelner Elemente solcher Arrays stellen dann Zustandsänderungen dar und können somit destruktiv implementiert werden. Dies bedeutet jedoch, daß ein Teil des hohen Abstraktionsniveaus funktionaler Sprachen im Umgang mit solchen Strukturen verloren geht. Wie in imperativen Sprachen obliegt es dem Programmierer, explizit darauf zu achten, ob bei der Modifikation eines Arrays eine Kopie erstellt werden muß oder nicht. Eine andere Möglichkeit, den Kopieraufwand bei Array-Operationen zu minimieren, ohne dabei zu Einschränkungen bei der Verwendung von Arrays zu führen, besteht darin, die Anzahl der Referenzen auf ein Array während der Programm-

berechung durch sog. Referenzzähler [Can89, Coh81] nachzuhalten. Modifizierende Array-Operationen können in diesem Fall genau dann destruktiv vorgenommen werden, wenn der Referenzzähler den Wert 1 hat. Die Verwaltung solcher Referenzzähler erfordert jedoch zusätzlichen Aufwand zur Laufzeit.

Aufgrund der vielen Effizienzprobleme kommen die meisten funktionalen Sprachen für eine Verwendung in numerischen Anwendungen nur bedingt in Frage. Daher spielte bisher ihre Eignung für solche Anwendungen beim Design dieser Sprachen nur eine untergeordnete Rolle.

Die funktionale Sprache SISAL [BCOF91] bildet hier eine Ausnahme. Sie verzichtet auf Polymorphismus, Funktionen höherer Ordnung, partielle Anwendungen und Lazy-Evaluation und erreicht Ausführungszeiten vergleichbar mit denen entsprechender FORTRAN-Programme[Can92, OCA86]. Trotz des guten Laufzeitverhaltens weist diese Sprache jedoch noch einige Schwächen in bezug auf numerische Anwendungen auf:

- Durch den Verzicht auf einige der wesentlichen, aber in der Implementierung aufwendigen Merkmale funktionaler Sprachen ist das Abstraktionsniveau von SISAL-Programmen vergleichsweise gering. Dennoch erfordert die Spezifikation von SISAL-Programmen, wie bei allen anderen dem Autor bekannten funktionalen Sprachen, eine ausdrucksorientierte Denkweise. Für einen FORTRAN- oder C-gewohnten Programmierer bedeutet dies eine erhebliche Umstellung, die das Erlernen einer vollkommen neuen Syntax beinhaltet.

- Obwohl das Sprachdesign von SISAL auf numerische Anwendungen ausgerichtet ist, unterstützt SISAL nur sehr elementare Array-Operationen. Komplexe Array-Operationen müssen mittels Schleifenkonstrukten spezifiziert werden, die sämtliche Elemente der Argument-Arrays traversieren. Die Anpassung solcher Array-Operationen an Argument-Arrays anderer Dimensionalität erfordert prinzipiell eine Modifikation ihrer Spezifikation.

- Die Möglichkeiten des I/O in SISAL beschränken sich darauf, daß SISAL-Programmen der Inhalt von Dateien als Parameter übergeben und die Ergebnisse berechneter Programme in (andere) Dateien geschrieben werden können. Ein allgemeiner Mechanismus zur Integration von Zuständen und Zustandsmodifikationen sowie zur Ausführung von interaktiven I/O-Operationen während der Berechnung von Programmen fehlt.

- Die Wiederverwendbarkeit in anderen Sprachen spezifizierter Programme ist nur in eingeschränktem Maße gegeben. Funktionen, die Seiteneffekte verursachen, können in SISAL nicht ohne eine besondere Re-Compilation bzw. Modifikation des Quellprogrammes benutzt werden (vergl. [BCOF91]). Viele Programm-Bibliotheken, insbesondere kommerziell angebotene, liegen jedoch nicht als Quelltext vor, so daß eine Einbindung gar nicht oder nur auf die Gefahr eines nicht-deterministischen Programmverhaltens hin möglich ist. Selbst wenn

der Quelltext verfügbar ist, gestaltet sich bei größeren Bibliotheken eine Re-Compilation meist sehr aufwendig.

Ziel dieser Arbeit ist es zu untersuchen, inwieweit sich die genannten Probleme bei SISAL lösen lassen. Dies erfordert nicht nur den Entwurf neuer Sprachkonstrukte, sondern auch ein Konzept für deren Realisierung. Dabei ist insbesondere im Zusammenhang mit den Array-Operationen darauf zu achten, daß es trotz eines höheren Abstraktionsniveaus gelingt, eine Compilation in effizient ausführbaren Code zu ermöglichen.

Um das Problem der ausdrucksorientierten Schreibweise umgehen zu können, bedarf es einer völlig andersartigen Syntax, so daß anstelle einer SISAL-Erweiterung eine neue funktionale Programmiersprache **S**ingle **A**ssignment **C**, kurz SAC, vorgeschlagen wird. Obwohl ihre Syntax weitestgehend auf der der imperativen Sprache C basiert, gelingt es, diese so einzuschränken, daß eine funktionale Semantik für SAC definiert werden kann. Dadurch kann sich der Programmierer frei zwischen einer ausdrucks- und einer anweisungsorientierten Denkweise entscheiden. Für den C- oder FORTRAN-gewohnten Programmierer bedeutet dies eine erhebliche Erleichterung bei einem Wechsel zu SAC.

Die Array-Darstellung sowie die von SAC unterstützten Array-Operationen ähneln denen in APL [Ive62] und NIAL [JJ93]. Zusammen mit der Möglichkeit, komplexe Array-Operationen dimensionsunabhängig zu spezifizieren, wird dadurch ein hohes Abstraktionsniveau im Umgang mit Arrays erreicht. Sowohl die Array-Darstellung, als auch die Array-Operationen in SAC sind so definiert, daß sie direkt auf den von L.Mullin entwickelten Ψ-Kalkül [Mul88, Mul91, MJ91] abgebildet werden können. Er stellt einen Formalismus für kontextfreie Transformationen kompositionierter Array-Operationen zur Verfügung und eignet sich daher als Grundlage für komplexe Optimierungen. Um trotz des hohen Abstraktionsniveaus eine Compilation in effizient ausführbaren Code zu ermöglichen, wird ein Typsystem entwickelt, das eine Hierarchie von Array-Typen unterstützt; es inferiert für eine entscheidbare Teilmenge aller im Programm vorkommenden Array-Ausdrücke deren Form bzw. Dimensionalität.

Die Ähnlichkeit von SAC zu C legt es nahe, eine Integration von Zuständen bzw. I/O so zu gestalten, daß sie sich notationell möglichst wenig von den Möglichkeiten in C unterscheidet. Basierend auf Uniqueness-Typen[SBvEP93, BS95] werden spezielle Module eingeführt, die mit einer ausgezeichneten, nach außen hin verborgenen Datenstruktur als Zustands-Repräsentant versehen sind. Zur Laufzeit können von diesen speziellen Modulen Instanzen gebildet werden, was der Einführung von entsprechenden Zuständen entspricht. Da dieser Mechanismus dem der Klassen/Objekte objekt-orientierter Sprachen wie C++[Str91] oder SMALLTALK[GK76] entspricht, heißen diese „speziellen Module“ in SAC Klassen und die „Instanzen“ der Klassen Objekte.

Durch die Erweiterung dieses Klassen-Konzeptes um einen sog. CALL-BY-REFERENCE-Mechanismus sowie globale Objekte kann für die Spezifikation von Zuständen

und Zustands-Modifikationen eine Notation verwendet werden, die der imperativer Sprachen fast vollständig entspricht. So können sämtliche Funktionen der Standard-I/O-Bibliothek aus C in SAC mit einer zu C identischen Notation aufgerufen werden, ohne daß es dabei zu Konflikten mit der auf kontextfreien Ersetzungen beruhenden funktionalen Semantik von SAC kommt.

Die Bereitstellung dieser Mechanismen in SAC erlaubt darüberhinaus eine weitreichendere Integration von in anderen Programmiersprachen spezifizierten Programmteilen in SAC, als dies beispielsweise in SISAL möglich ist. So können auch Funktionen, die Seiteneffekte verursachen, in SAC integriert werden, ohne daß eine Anpassung der zu importierenden Funktion erforderlich ist.

Die Arbeit gliedert sich wie folgt: Im nächsten Kapitel wird die Eignung existierender funktionaler Sprachen für die Spezifikation numerischer Algorithmen ausführlich untersucht. Nach einer Identifikation der entscheidenden Anforderungen solcher Algorithmen soll ein Überblick darüber geben werden, inwieweit existierende funktionale Sprachen diesen Anforderungen gerecht werden. Aus den dabei gewonnenen Erkenntnissen ergibt sich das Sprachdesign für SAC.

Eine ausführliche Beschreibung des Sprachdesigns von SAC bietet Kapitel 3. Zentrales Thema der ersten Abschnitte dieses Kapitels ist die Definition einer funktionalen Semantik für den C-Kern von SAC. Dazu wird in Abschnitt 3.1 zunächst eine einfache, ungetypte funktionale Sprache $\mathcal{F}$UN definiert, deren Semantik über eine kontextfreie Substitution erklärt ist. Anschließend wird ein Transformationsschema entwickelt, das die Abbildung von SAC-Programmen in $\mathcal{F}$UN-Programme erlaubt. Zusammen mit dem in Abschnitt 3.2 eingeführten Typsystem kann schließlich die funktionale Semantik des Kernes von SAC vollständig definiert werden.

Gegenstand des Abschnittes 3.3 sind die Erweiterungen des Sprachkerns von SAC um Array-Konstrukte. Dies umfaßt nicht nur eine Beschreibung der Array-Darstellung sowie der auf Arrays verfügbaren Operationen in SAC, sondern auch eine Erweiterung des bis dahin vorgestellten Typsystems um spezielle Array-Typen und Typinferenzregeln. Die Einführung einer Hierarchie von Array-Typen ermöglicht nicht nur die Inferenz des Typs der Elemente von Arrays, sondern auch die Inferenz der Dimensionalität oder sogar der genauen Form von Arrays.

Die letzten drei Abschnitte von Kapitel 3 stellen das Modul- und das Klassensystem von SAC sowie die daraus erwachsenden Möglichkeiten zur Einbindung nicht in SAC geschriebener Programme vor.

In Kapitel 4 wird ein Compiler für die Übersetzung von SAC-Programmen nach C entwickelt. Dabei geht es vor allem um die Fragestellung, welche Übersetzungsmöglichkeiten das Sprachdesign offenläßt und inwieweit eine solche Übersetzung zu laufzeit- bzw. speichereffizientem Code führen kann. Nach den Betrachtungen zur Compilation des Sprachkerns (Abschnitt 4.1) sowie zur Realisierung des Typsystems (Abschnitt 4.2) befaßt sich der Abschnitt 4.3 mit der Umsetzung des Array-Konzeptes. Hier steht die Integration eines Referenzzählermechanismus im Vordergrund, der die

Minimierung der zur Laufzeit durch Arrays belegten Speicherbereiche ermöglicht.

Abschnitt 4.4 geht auf die Probleme bei der Realisierung des Modulsystemes von SAC ein. Dabei wird eine besondere Implementierungstechnik vorgeschlagen, die es ermöglicht, dem Programmierer verborgene Informationen über die Interna eines Modules dem Compiler zugänglich zu machen. Dadurch können Verschlechterungen des Laufzeitverhaltens durch Programm-Modularisierungen vermieden werden.

Die Integration des Klassen-Konzeptes wird in Abschnitt 4.5 vorgestellt. Dabei gelingt es, einen Compilations-Mechanismus zu entwickeln, der eine laufzeiteffiziente Abbildung dieser Konstrukte erlaubt. So können der CALL-BY-REFERENCE-Mechanismus direkt auf den sog. Adressoperator und globale Objekte auf globale Variablen in C abgebildet werden.

Aus der funktionalen Semantik von SAC ergeben sich gegenüber äquivalenten C-Programmen erweiterte Optimierungsmöglichkeiten. Abschnitt 4.6 stellt deshalb zunächst die Erweiterungsmöglichkeiten allgemein bekannter Optimierungen vor und geht dann auf speziell für SAC entwickelte Array-Optimierungen ein. Neben dem Typsystem sind diese Optimierungen maßgeblich dafür verantwortlich, daß aus dimensionsunabhängig spezifizierten Array-Operationen effizient ausführbarer Code erzeugt werden kann.

Basierend auf einer konkreten Implementierung des Compilers, deren Details in [Wol95, Sie95, Gre96] beschrieben sind, untersucht eine Fallstudie in Kapitel 5 die Tragfähigkeit des Sprachdesigns von SAC. Als Beispielproblem dient dabei die in Abschnitt 5.1 vorgestellte Multigrid-Relaxation zur Approximation der Lösung von partiellen Differentialgleichungen. Während in Abschnitt 5.2 die Ausdruckskraft der Sprachelemente von SAC in bezug auf eine dimensionsunabhängige Spezifikation des Multigrid-Relaxations-Algorithmus im Vordergrund steht, diskutiert Abschnitt 5.3 einige Laufzeitmessungen im Vergleich zu äquivalenten SISAL- und FORTRAN-Programmen.

Kapitel 6 bietet schließlich eine Zusammenfassung der Ergebnisse dieser Arbeit und gibt einen Ausblick auf mögliche Ansatzpunkte weiterer Untersuchungen.

Kapitel 2

Funktionale Sprachen und numerische Anwendungen

In diesem Kapitel soll die Tragfähigkeit verschiedener existierender funktionaler Sprachen bzw. Sprachkonzepte für numerische Anwendungen diskutiert werden. Dabei stehen zwei Fragestellungen im Vordergrund:

- Welches sind die wesentlichen Anforderungen/ Wünsche, die ein Anwendungsprogrammierer bei der Spezifikation numerischer Algorithmen an die Programmiersprache stellt?
- Welche dieser Anforderungen werden von existierenden funktionalen Sprachen erfüllt?

Bei der Untersuchung dieser Fragestellungen stehen weniger syntaktische oder semantische Detailfragen im Mittelpunkt als das den einzelnen Sprachen zugrunde liegenden Sprachdesign. Für diese Aufgabe lassen sich drei verschiedene Schwerpunkte identifizieren:

- Die in numerischen Anwendungen meistbenötigten Datenstrukturen sind ein- oder mehrdimensionale Arrays von Gleitkommazahlen. Die Darstellung solcher Strukturen sowie die Sprachkonstrukte zur Beschreibung komplexer Operationen auf ihnen gehören daher zu den wichtigsten Sprachelementen.
- Viele numerische Anwendungen wie z.B. solche aus den Bereichen der sog. Computational Fluid Dynamics [BW91, War93, Bec87], also der Simulation von Strömungsprozessen, der Biochemie [KKS92] oder der Finite Element Methoden [BE76, GZO+78] berechnen nicht einzelne Ergebniswerte, sondern mehrdimensionale Ergebnis-Arrays, die erst durch eine visuelle Darstellung erfaßbar werden. Die Integration eines umfangreichen I/O-Systems ist deshalb für numerische Anwendungen ebenfalls essentiell.

- Gerade im Bereich der numerischen Anwendungen existieren bereits viele handoptimierte Programmbibliotheken. Um ihre Verwendung zu ermöglichen, bedarf es eines Modulsystems, das eine Integration von in anderen Programmiersprachen spezifizierten Funktionen zuläßt, selbst wenn diese via Referenzparameter übergebene Datenstrukturen modifizieren.

Im folgenden soll auf jeden dieser Schwerpunkte gesondert eingegangen werden.

2.1 Darstellung von Arrays und Array-Operationen

Ein Array ist eine Datenstruktur bestehend aus Elementen gleichen Typs, auf die mittels sog. Indizes zugegriffen werden kann. Ein Index kann aus ein oder mehreren Komponenten $i_1, ..., i_n \in \mathbb{N}$ bestehen. Jede dieser Komponenten kann in einem vorgegebenen Bereich $k_j \leq i_j \leq l_j$ mit $k_j, l_j \in \mathbb{N}$ für $j \in \{1, ..., n\}$ variieren. Die Anzahl n der Komponenten heißt Dimension des Arrays. Ein eindimensionales Array heißt Vektor.

Einige funktionale Sprachen wie z.B. MIRANDA [Tur85] oder ERLANG [AWWV96] bieten keinerlei Unterstützung für Arrays. In diesen Sprachen können Arrays nur als geschachtelte Listen oder Tupel dargestellt werden. Die Spezifikation komplexer Operationen auf solchen Strukturen gestaltet sich sehr aufwendig, da sie eine Zerlegung in elementare Listen/Tupel vor und eine Rekombination der elementaren Strukturen nach der eigentlichen Operation erfordert.

In den meisten Sprachen, die Arrays als eigenständige Datenstrukturen zur Verfügung stellen, werden diese als Schachtelungen eindimensionaler Strukturen dargestellt. Diese Art der Array-Darstellung unterstützen unter anderem STANDARD ML [MTH90], CLEAN [PvE95], NESL [Ble94], ID [AGP78] und SISAL [MSA+85][1]. Da die meisten dieser Sprachen ausschließlich eindimensionale Array-Operationen bieten, müssen Operationen auf mehrdimensionalen Arrays durch Schachtelungen der eindimensionalen Operationen dargestellt werden. Neben primitiven Operationen wie Subarray-Selektion, Rotation oder arithmetischen Operationen bieten einige dieser Sprachen auch sog. Array-Comprehension-Konstrukte. Mit Hilfe der Array-Comprehension-Konstrukte ist es möglich, eine für ein Array-Element spezifizierte Berechnungsvorschrift auf eine durch eine Menge von Indizes charakterisierte Menge von Array-Elementen anzuwenden. Die dabei verwendete Notation ist in den einzelnen Sprachen sehr unterschiedlich (FOR-LOOPS in SISAL [MSA+85], APPLY-TO-EACH-CONSTRUCTS in NESL [Ble94] oder ARRAY-COMPREHENSIONS in CLEAN [PvE95]), lehnt sich jedoch an die in der Mathematik übliche Mengennotation an.

Der Umfang der zur Verfügung stehenden Operationen variiert dabei je nach Ausrichtung der Sprache auf ein bestimmtes Anwendungsgebiet sehr stark. Während

[1] In SISAL2.0 [BCOF91] soll es auch „echte“ mehrdimensionale Arrays geben; jedoch in der aktuellen Compiler-Version werden diese noch nicht unterstützt.

z.B. in CLEAN außer den Array-Comprehension-Konstrukten kaum Array-Operationen vorhanden sind, bietet NESL sehr viele verschiedene primitive Operationen auf Arrays. SISAL setzt sich insofern von diesen Sprachen ab, als daß die meisten Operatoren, insbesondere auch die FOR-LOOPS direkt auf n-dimensionale Array-Strukturen angewendet werden können, was zu konziseren Spezifikationen komplexer Operationen führt.

Allen diesen auf der Schachtelung von eindimensionalen Strukturen basierenden Array-Darstellungen ist jedoch gemein, daß Zugriffe auf die Array-Elemente immer über eine feste Anzahl von Indizes erfolgen. Eine Spezifikation von Operationen, die auf Arrays verschiedener Dimensionalität anwendbar sind, ist daher nicht möglich.

Ein anderer Ansatz zur Darstellung von Arrays wird in HASKELL [HAB+95] verfolgt. In HASKELL wird ein Array als Abbildung von Indizes auf Elementwerte verstanden. Dadurch ergibt sich zwar auf direkte Weise, daß Array-Comprehension-Konstrukte auf mehrdimensionale Arrays anwendbar sind, jedoch ist es leider auch in HASKELL nicht möglich, dimensionsunabhängige Array-Modifikationen zu spezifizieren. Dies liegt in der Notwendugkeit begründet, Indizes als Zahlen bzw. Tupel von Zahlen spezifizieren zu müssen, die in HASKELL grundsätzlich eine feste Stelligkeit haben. Ähnlich wie in CLEAN sind auch in HASKELL außer den Array-Comprehension-Konstrukten nur sehr wenige primitive Array-Operationen verfügbar.

Ein dritter Ansatz zur Darstellung von Arrays wird schließlich in Sprachen wie APL [Ive62] oder NIAL [JJ93] verfolgt. In diesen Sprachen werden Arrays durch zwei Vektoren repräsentiert: durch den sog. Datenvektor, der sämtliche Array-Elemente enthält, und durch den sog. Shape-Vektor. Ein Shape-Vektor ist ein Vektor von natürlichen Zahlen größer Null und beschreibt die Menge der zulässigen Indizes eines Arrays. Ein Vektor $[i_1, ..., i_n]$ mit $i_j \in \mathbb{N}$ für alle $j \in \{1, ..., n\}$ ist genau dann ein zulässiger Indexvektor in ein Array A mit dem Shape-Vektor $[s_1, ..., s_n]$, wenn für alle $j \in \{1, ..., n\}$ gilt: $0 \leq i_j < s_j$.

Durch diese Trennung von Element- und Indexmengenspezifikation wird es möglich, Array-Operationen unabhängig von der Dimension des Arrays zu beschreiben. Obwohl APL eine Fülle primitiver Array-Operationen bietet, fehlen Array-Comprehension-Konstrukte, um eine konzise Spezifikation von komplexen Array-Operationen zu erreichen. Im Gegensatz dazu finden sich in NIAL weniger primitive Array-Operationen, dafür jedoch den Array-Comprehension-Konstrukten ähnelnde Konstrukte, die sog. Transformer. Der wichtigste primitive Transformer ist der sog. EACH-TRANSFORMER (siehe [JG89]). Er ermöglicht die Anwendung einer für ein einzelnes Array-Element beschriebenen Operation auf sämtliche Elemente des Arrays. Eine Einschränkung des EACH-TRANSFORMER auf Teilmengen von Indizes ist jedoch nicht möglich.

Fazit Der in APL und NIAL gewählte Ansatz der Trennung von Daten- und Shape-Vektor bietet die größte Flexibilität, jedoch fehlt in beiden Sprachen eine vollständige

Unterstützung von Array-Comprehension-Konstrukten.

Von den Ansätzen mit Arrays als Schachtelung eindimensionaler Strukturen heben sich SISAL durch die Möglichkeiten der direkten Manipulation mehrfach geschachtelter Strukturen und NESL durch die Fülle primitiver Operationen ab. Alle auf der Schachtelung eindimensionaler Strukturen beruhenden Ansätze haben jedoch gemein, daß es nicht möglich ist, Array-Operationen unabhängig von der Dimensionalität der Arrays, auf die sie angewendet werden, zu spezifizieren.

Um konzise Spezifikationen komplexer Array-Operationen unabhängig von der Dimensionalität der Argument-Arrays zu ermöglichen, bedarf es einer Array-Darstellung basierend auf der Trennung von Daten- und Shape-Vektoren kombiniert mit mächtigen Array-Operationen und Array-Comprehension-Konstrukten.

2.2 Integration von I/O-Operationen

Der Bedarf an I/O-Operationen entsteht zum einen bei der Fehlersuche während der Entwurfsphase von Programmen und zum anderen bei der eigentlichen Applikation selbst zur Ausgabe von Ergebnissen. In bezug auf die Fehlersuche ist es vor allem wichtig, ohne viel Spezifikationsaufwand in eine bereits vorhandene Implementierung nachträglich I/O-Operationen einfügen zu können.

Die Anforderungen an die Ausgabemöglichkeiten von Ergebnisdaten hängt von der konkreten Anwendung ab. So ist z.B. bei Problemen der linearen Optimierung [BM87, HH89] eine graphische Darstellung nicht unbedingt erforderlich. Eine numerische Ausgabe der berechneten Werte ist für diese Art von Anwendungen in den meisten Fällen vollkommen ausreichend. Betrachtet man Interpolationsprobleme wie sie im Bereich der Analyse von Meßergebnissen anfallen, so reichen zumeist zweidimensionale, funktionsartige Darstellungen aus. Für den großen Bereich der Simulation von Naturvorgängen wie dem Strömungsverhalten von Flüssigkeiten oder Gasen, der Anordnung der Elemente in großen Molekülen oder dem Torsionsverhalten von Gegenständen sind jedoch aufwendige dreidimensionale Darstellungen erforderlich. Dabei reicht in der Regel eine fest vorgegebene Darstellung nicht aus; es werden vielmehr Möglichkeiten der interaktiven Manipulation der Darstellung wie Drehungen, Vergrößerungen oder Schnittbilder benötigt.

Die Implementierung derartiger Darstellungssysteme erfordert ein vollständiges I/O-System, das neben einfachen File- und Terminal-Operationen auch komplexere Graphik-Operationen unterstützt.

Der Umgang mit Zuständen in funktionalen Sprachen, insbesondere die Integration von I/O-Systemen, ist schon seit geraumer Zeit ein Gebiet aktiver Forschung [HS89, Per91, Hud92, Gor92, JW93, Ach96]. Die Ursache für die Probleme bei der Integration von I/O-Systemen in funktionale Sprachen liegt in der funktionalen Semantik, die auf kontextfreien Substitutionen beruht. Eine der wesentlichen Stärken dieser Semantik liegt darin, daß die Reihenfolge der Ausführung einzelner Substi-

tutionen aufgrund der Church-Rosser-Eigenschaft keinen Einfluß auf das Ergebnis einer Berechnung hat, solange diese terminiert. Die Wahl einer konkreten Reihenfolge von Substitutionen in einem vorgegebenen Programm kann deshalb dynamisch bei der Compilation oder sogar erst bei der eigentlichen Berechnung stattfinden. Da es jedoch bei I/O-Operationen wichtig ist, ihre Ausführungsreihenfolge bei der Programmspezifikation zumindest partiell vorzugeben, müssen für die I/O-Systeme funktionaler Sprachen Mechanismen zur Sequentialisierung bereitgestellt werden.

Aufgrund dieser Probleme bieten manche funktionale Sprachen keinerlei Unterstützung für I/O. Als wichtigstes Beispiel im Kontext numerischer Anwendungen ist hier SISAL zu nennen. Jegliche Visualisierung von Ergebnissen kann daher nicht direkt in SISAL, sondern muß in einer anderen Sprache implementiert werden. Ausgaben von Teilergebnissen zum Zweck der Fehlersuche sind gar nicht möglich.

Für die funktionalen Sprachen mit I/O-Unterstützung sind viele verschiedene Ansätze zur Integration von I/O-Systemen entwickelt worden. Im wesentlichen kann man zwischen drei Grund-Ansätzen unterscheiden: Side-Effecting-I/O, Stream-Based-I/O und Environment-Based-I/O.

Side-Effecting-I/O Die meisten Implementierungen funktionaler Sprachen auf Mono-Prozessor-Systemen legen eindeutig fest, welche Substitution bei einem vorgegebenen Programm als nächstes auszuführen ist. Dies führt insbesondere bei den sog. Eager-Evaluation-Sprachen, die bei allen Funktionsanwendungen eine vollständige Berechnung der Argumente vor der Berechnung des Funktionsrumpfes garantieren, dazu, daß die Reihenfolge der Substitutionen bei der Berechnung von Programmen für den Programmierer leicht vorhersagbar wird. Deshalb bieten einige funktionale Sprachen wie z.B. STANDARD ML, NESL, oder NIAL, I/O-Operationen ohne Mechanismen zur Sequentialisierung bereitzustellen. Eine derargtige, auch als „Side-Effecting-I/O“ [Gor92] bezeichnete Integration von I/O-Operationen führt zwar zu einer nicht-eindeutigen Semantik, diese macht sich in der Praxis erst dann als Nicht-Determinismus bemerkbar, wenn mehrere Argumente, die in einer verteilten Implementierung nebenläufig berechnet werden, I/O-Operationen vornehmen. Da jedoch die verteilten Berechnungen gerade im Kontext numerischer Anwendungen eine zentrale Rolle spielen, ist diese Art der Integration von I/O für eine auf solche Anwendungen ausgerichtete Sprache ungeeignet.

Stream-Based-I/O Zu dieser Art von I/O-Integration lassen sich verschiedene Ansätze rechnen. Sie alle basieren auf der Idee, die FIFO-Eigenschaft von Streams zur Sequentialisierung von I/O-Operationen zu verwenden. Dazu bekommt jedes funktionale Programm einen Stream, der sämtliche Eingaben enthält, als Argument und liefert als Ergebnis einen Stream von Ausgaben.

Eine direkte Umsetzung dieses Prinzips findet sich in den sog. Dialogues, der Basis des I/O in HASKELL-1.2 [HJW+92]. Konzeptuell werden die eigentlichen I/O-

Operationen nicht von dem HASKELL-Programm selbst, sondern von dem das HASKELL-Programm umgebende Betriebssystem ausgeführt; das HASKELL-Programm dient lediglich der Erzeugung von I/O-Anforderungen, den sog. Requests, und der Verarbeitung der zugehörigen, vom Betriebssystem erzeugten Resultate (sog. Responses). Auf diese Weise sind nicht nur die Ausgabeoperationen bzw. die Eingabeoperationen untereinander, sondern sämtliche I/O-Operationen sequentialisiert. Durch die Korrespondenz der Responses zu den Requests wird dieser Ansatz auch als Synchronized-Stream-I/O [Gor92] bezeichnet. Die Hauptprobleme dieses Ansatzes liegen darin, daß zum einen die vom Programmierer zu gewährleistende Synchronisation von Requests und Responses sehr fehleranfällig ist und daß zum anderen die Spezifikationen zur Konstruktion bzw. Konsumierung solcher Streams sehr unübersichtlich sind.

Ausgehend von den Dialogues läßt sich auf einfache Weise das sog. Continuation-Passing-I/O einführen, das ebenfalls in HASKELL-1.2 enthalten ist. Bei diesem Ansatz wird dem Programmierer die Aufgabe abgenommen, auf das synchrone Erzeugen von Requests und das Konsumieren von dazugehörigen Responses zu achten. Dazu werden die I/O-Operationen als spezielle Funktionen definiert, die neben ihren normalen Parametern eine sog. Continuation sowie den Stream von Responses erhalten. Jede dieser primitiven I/O-Funktionen erzeugt genau einen der Requests, konsumiert das erste Element der Responses und wendet anschließend die Continuation auf dieses Element sowie auf die übrigen Responses an. Auf diese Weise lassen sich I/O-Operationen als Schachtelungen von Funktionsanwendungen spezifizieren, wobei durch die Konstruktion der Streams sich eine Ausführungsreihenfolge von außen nach innen ergibt. Durch die Spezifikation der I/O-Operationen mittels Funktionen bleiben dem Programmierer zwar die Synchronisationsprobleme der beiden Streams erspart, jedoch ist die Notation als Schachtelung von Funktionsanwendungen sehr viel umständlicher als vergleichsweise die Notation von I/O in imperativen Sprachen. Darüber hinaus bietet HASKELL1.2 keinerlei Graphikprimitiva, was auch diesen Ansatz für eine Verwendung zur Visualisierung von Ergebnissen numerischer Anwendungen unbrauchbar erscheinen läßt.

Ein anderer, ebenfalls auf Streams basierender Ansatz sind die von Carlsson und Hallgren in den Chalmers-HASKELL-Compiler integrierten sog. Fudgets [CH93]. Der Name steht abkürzend für „functional widgets“. Der Begriff Widget wiederum stammt von der X Windows [Jon85] Graphik-Oberfläche; er bezeichnet die graphischen Grundelemente dieses Systems. Die wesentliche Idee der Fudgets besteht darin, nicht einen Input- und einen Output-Stream für das gesamte Programm zu haben, sondern jedem graphischen Objekt(Widget) ein funktionales Objekt(Fudget) mit eigenem Input- und Output-Stream zuzuordnen. Mit Hilfe von speziellen Funktionen lassen sich dann mehrere Fudgets zu einem komplexen Graphikobjekt kombinieren. Durch die Streams zwischen den einzelnen Fudgets wird die Sequentialität der I/O-Operationen innerhalb der Fudgets sichergestellt.

Dieser Ansatz ist sehr viel flexibler als die bisher geschilderten. Er bietet nicht nur

mächtige Konstrukte zur Erzeugung graphischer Elemente, sondern gestattet auch die nebenläufige Ausführung verschiedener Fudgets. Gerade durch die Mächtigkeit der einzelnen I/O-Konstruke entsteht jedoch das Problem, daß sich das Einfügen von I/O-Operationen in ein existierendes Programm zum Zweck der Fehlersuche sehr aufwendig gestaltet.

Environment-Based-I/O Die Bezeichnung „Environment-Based-I/O“ reflektiert die Idee, eine abstrakte Datenstruktur, die den Zustand der Umgebung des gesamten Programmes repräsentiert, zu benutzen, um eine sequentielle Abfolge der I/O-Operationen sicherzustellen. Jede I/O-Operation verlangt diese im weiteren mit WORLD-Struktur bezeichnete Datenstruktur als zusätzlichen strikten Parameter und liefert sie in - zumindest konzeptuell - modifizierter Form zurück. Unter diesen Voraussetzungen ist eine sequentielle Abfolge der Operationen garantiert, wenn zwischen allen diesen Operationen eine Datenabhängigkeit bezüglich der den Zustand repräsentierenden WORLD-Struktur besteht; eine solche Datenabhängigkeit ist nämlich gleichbedeutend mit einer Schachtelung der Operationen. Da alle Operationen per definitionem strikt bezüglich WORLD-Struktur sind, ist dann eine Auswertung von innen nach außen garantiert. Es gibt verschiedene Ansätze, um diese Datenabhängigkeiten zu gewährleisten.

Ein möglicher Ansatz sind die sog. Monaden [Wad92b, JW93, Wad92a]. Sie sind Bestandteil von HASKELL-1.3 [HAB+95] und in erweiterter Form bereits in verschiedenen HASKELL-Compiler-Versionen enthalten. Der Begriff „Monaden“ entstammt der Kategorientheorie [BW85, LS86]. Da diese aber für ein Verständnis der Funktionsweise dies Ansatzes nicht erforderlich ist, soll auf den Bezug zur Kategorientheorie nicht näher eingegangen werden. Eine detaillierte Beschreibung über diesen Zusammenhang findet sich in [Wad92a].

Die prinzipielle Vorgehensweise im Monaden-Ansatz ist die folgende: Zunächst wird ein abstrakter Datentyp eingeführt, der die WORLD-Struktur enthält. Alle I/O-Operationen wie auch das gesamte Programm liefern keine WORLD-Struktur selbst, sondern den abstrakten Datentyp zurück. Die einzige Funktion, die die WORLD-Struktur aus dem abstrakten Datentyp wieder extrahieren kann, ist eine primitive Funktion BIND. Sie bekommt zwei I/O-Operationen als Argumente, extrahiert aus dem Ergebnis der ersten die modifizierte WORLD-Struktur und wendet die zweite I/O-Operation darauf an. Auf diese Weise lassen sich mit Hilfe von BIND alle I/O-Operationen schachteln. Dadurch daß das gesamte Programm den abstrakten Datentyp liefert, stellt schließlich das Typsystem sicher, daß entweder nur eine I/O-Operation im ganzen Programm enthalten ist oder aber durch Schachtelungen mittels BIND zwischen allen I/O-Operationen Datenabhängigkeiten existieren.

Da sowohl das Extrahieren der WORLD-Struktur als auch die Anwendung der I/O-Operationen auf diese für den Anwendungsprogrammierer transparent bleibt, wird dieser Ansatz in der Literatur auch als Implicit-Environment-Passing [AP95]

bezeichnet.

Die Stärken dies Monaden-Ansatzes liegen vor allem auf notationeller Ebene. I/O-Operationen können wie in imperativen Sprachen spezifiziert werden, d.h. es sind keine zusätzlichen Argumente wie z.B. eine Continuation erforderlich. Außerdem lassen sich mittels einer Infix-Notation von BIND Implementierungen spezifizieren, an denen sich die Ausführungsreihenfolge der einzelnen I/O-Operationen gut erkennen läßt.

Die Verwendung des Monaden-Ansatzes zu Zwecken der Fehlersuche führt jedoch zu Problemen. Sollen in eine bereits existierende Spezifikation einer Funktion, die keine I/O-Operationen enthält, solche eingefügt werden, so ändert sich der Resultattyp, und alle Anwendungen dieser Funktion müssen entsprechend angepaßt werden. Da dies wiederum Änderungen der Signaturen anderer Funktionen nach sich ziehen kann, handelt es sich um einen rekursiven Prozess, der wiederholt werden muß, bis alle potentiellen Aufrufhierarchien angepaßt sind. Da solche speziell für die Fehlersuche eingefügten Ausgaben nach dem Auffinden der Fehler wieder entfernt werden müssen, ist auch der Monaden-Ansatz für diese Zwecke ungeeignet.

Ein anderes Problem dieses Ansatzes entsteht dadurch, daß die WORLD-Struktur implizit durch die Funktion BIND den einzelnen I/O-Operationen zugeführt wird. Sollen mehrere voneinander unabhängige Repräsentationen von (Teil-) Zuständen den einzelnen I/O-Operationen zugeführt werden, so erzeugt der BIND-Mechanismus Datenabhängigkeiten zwischen allen I/O-Operationen, obwohl die zu modifizierenden Teilzustände von einander unabhängig sind. Man spricht deshalb auch von monolithischem I/O [AP95]. Es gibt zwar Ansätze zur Lösung dieses Problems [JD93, KP92, LJ94], jedoch ist nach Wissen des Autors noch keine universelle Lösung bekannt.

Eine Einbindung von Graphik-Operationen wie z.B. den X Windows-Bibliotheksfunktionen ist zwar noch nicht in HASKELL-1.3 enthalten, eine Unterstützung durch den Glasgow-HASKELL-Compiler ist jedoch bereits vorhanden [FJ96].

Ein anderer Ansatz schließlich sind die sog. Uniqueness-Typen in CLEAN [AP93, SBvEP93, AP95, Ach96]. Anstatt einen abstrakten Datentyp mit einer eigenen Zugriffsfunktion BIND bereitzustellen, wird in CLEAN ein Typattribut UNQ eingeführt. Das (die) angewandte(n) Vorkommen eines Funktionsparameters kann (können) genau dann als UNQ attributiert werden, wenn

1. der Funktionsparameter selbst als UNQ annotiert ist und

2. es im Rumpf der Funktion entweder nur genau ein Vorkommen gibt oder aber, im Falle einer Fallunterscheidung, es maximal ein Vorkommen pro Zweig der Fallunterscheidung gibt.

Eine Funktionsanwendung `fun arg_1 ... arg_n` ist nur dann korrekt typbar, wenn für alle UNQ attributierten formalen Parameter `p_j` der Funktion `fun` mit $j \in \{1, ..., n\}$ gilt: `arg_j` ist ebenfalls UNQ attributierbar. Annotiert man bei allen I/O-Operationen

den WORLD-Struktur-Parameter als UNQ, so ist sichergestellt, daß eine WORLD-Struktur niemals Argument von zwei Funktionsanwendungen ist, die beide zum Ergebnis der gesamten Berechnung beitragen. Dies garantiert die Existenz der benötigten Datenabhängigkeiten.

Durch die explizite Übergabe der WORLD-Struktur wird der Umgang mit disjunkten Teilzuständen möglich, ohne die Sequentialisierung aller I/O-Operationen dabei zu erzwingen. CLEAN bietet dafür primitive Funktionen, die die WORLD-Struktur als Argument bekommen und andere abstrakte Datenstrukturen als Resultat liefern. Diese Strukturen repräsentieren jeweils nur einen Teil des Zustandes der Programm-Umgebung wie z.B. das Filesystem, das Terminal oder Events des Graphiksystems. Auch sie sind als UNQ attributiert, um eine sequentielle Abfolge der Operationen auf den einzelnen Teilzuständen sicherzustellen. Komplementär dazu gibt es entsprechende Funktionen, mit denen sich die Datenstrukturen wieder zu einer WORLD-Struktur rekombinieren lassen.

Das I/O-System von CLEAN bietet eine ganze Hierarchie solcher Teilzustände und unterstüzt damit nicht nur ein vollständiges Graphiksystem [AP93, Ach96], sondern auch Möglichkeiten zur programmgesteuerten Prozessverwaltung [AP95, Ach96].

Ein anderer Vorteil des Uniqueness-Typen-Ansatzes liegt in der direkten Nutzbarkeit des UNQ-Attributes für alle Datenstrukturen, für die eine „sequentielle“ Verwendung garantiert werden kann. Insbesondere bei der Anwendung von primitiven Operationen auf große Datenstrukturen wie z.B. Arrays läßt dies eine effizientere Compilation zu, da der von einer als UNQ gekennzeichneten Datenstruktur benötigte Speicherbereich von der Operation direkt verändert werden kann (destructive update). Um solche Optimierungen auch bei Datenstrukturen zu ermöglichen, die nicht vom Programmierer mit dem UNQ-Attribut gekennzeichnet sind, gibt es bereits Ansätze, die Annotation des UNQ-Attributes zu inferieren [BS95].

Das nachträgliche Einfügen von I/O-Operationen zu Fehlersuche-Zwecken stellt sich in CLEAN als genauso aufwendig wie beim Monaden-Ansatz in HASKELL heraus. Sämtliche Funktionsaufrufe aller potentiellen Aufrufhierarchien müssen ggf. um zusätzliche Parameter und Rückgabewerte ergänzt werden.

Außerdem hat die explizite Übergabe der WORLD-Struktur notationelle Nachteile. Zum einen ist bei jedem angewandten Vorkommen einer I/O-Operation sowohl die Übergabe als auch die Rückgabe der WORLD-Struktur von einem pragmatischen Standpunkt aus vollkommen überflüssig; zum anderen ist es durch Definitionen der Form `My_Own_World = IO_Fun arg_1 ... arg_n World` möglich, die WORLD-Struktur umzubenennen, was zu erheblichen Verwirrungen führen kann.

Fazit Aus Sicht des Anwenders stellt Side-Effecting-I/O die einzige Möglichkeit dar, mit einem vertretbaren Spezifikationsaufwand I/O-Operationen zur Fehlersuche einzufügen. Dies führt jedoch evtl. zu Nichtdeterminismen bei einer nicht-

sequentiellen Ausführung solcher Programme.

Vom konzeptuellen Standpunkt her ist der Uniqueness-Typen-Ansatz in CLEAN der universellste; er unterstützt nicht nur nicht-monolithisches I/O, sondern kann auch dazu genutzt werden, Operationen auf großen Datenstrukturen effizienter zu implementieren. Darüber hinaus bietet CLEAN eine vollständige Einbindung der X-Windows-Oberfläche.

Eine Integration von I/O-Operationen sollte daher zwei Ziele verfolgen: Zum einen sollten die Möglichkeiten des Uniqueness-Types-Ansatz in CLEAN in bezug auf nicht-monolithisches-I/O und überschreibende Operationen auf großen Datenstrukturen gegeben sein. Zum anderen sollte es dem Programmierer auch ermöglichen, eine dem Side-Effecting-I/O ähnliche Spezifikation von I/O-Operationen vorzunehmen, ohne zu Nicht-Determinismen bei einer nebenläufigen Berechnung zu führen.

2.3 Wiederverwendbarkeit bereits erstellter Implementierungen

Das Bedürfnis von Anwendungsprogrammierern, Programme und Programmteile für verschiedene Probleme mehrfach nutzen zu können, hat zu der Entwicklung von Modulsystemen geführt. Als Prototyp vieler Modulsysteme, sowohl imperativer als auch funktionaler Sprachen dient MODULA [Wir85]. Es bietet

- sog. Information-Hiding, d.h. die Möglichkeit, Teile der Implementierung eines Moduls vor einem potentiellen Benutzer zu verbergen,
- eine separate Compilation aller Module, wodurch eine wiederholte Compilation bei mehrfacher Verwendung von Modulen vermieden werden kann,
- Mechanismen, um gezielt nur einzelne Symbole eines Moduls zu importieren bzw. zu exportieren sowie
- die Möglichkeit, ganze Modul-Hierarchien zu konstruieren, die gegenseitig voneinander importieren.

Über diese durch Überlegungen des Software-Engineering motivierten Eigenschaften moderner Modulsysteme hinaus spielt bei numerischen Anwendungen ein weiterer Aspekt der Wiederverwendbarkeit von Programmen eine wesentliche Rolle. Auf den meisten Plattformen existieren bereits umfangreiche, oftmals vom Hersteller entwickelte, Bibliotheken für numerische Standardalgorithmen. Sie sind zumeist hardware-nah codiert und enthalten daher sehr effizient ausführbaren Code. Um solche Bibliotheken nutzen zu können, ist es wichtig, auch in anderen, ggf. nicht-funktionalen Sprachen geschriebene Programmteile über das Modulsystem einbinden zu können. Dies erfordert nicht nur die Anwendbarkeit von in anderen Sprachen spezifizierten Funktionen, sondern auch die Möglichkeit, Datenstrukturen zwischen

den verschiedenen Sprachen austauschen zu können. Bei der Verwendung nichtfunktionaler Programmteile muß darüberhinaus sichergestellt werden, daß eventuelle Seiteneffekte korrekt von der Semantik der importierenden funktionalen Sprache erfaßt oder aber ausgeschlossen werden.

Einige funktionale Sprachen wie NESL oder NIAL bieten kein Modulsystem bzw. nur rudimentäre Möglichkeiten zur Modularisierung. Beim Gros der funktionalen Sprachen mit Modulsystem basiert dieses auf den Modulkonzepten von MODULA oder ähnelt ihnen zumindest. Als wichtigste Beispiele sind hier SISAL, CLEAN und HASKELL zu nennen. Während die Modulsysteme von CLEAN und HASKELL sich im wesentlichen auf die Konzepte von MODULA beschränken, bietet SISAL explizite Unterstützung für die Einbindung von bzw. in FORTRAN- bzw. C-Programme(n).

Es soll an dieser Stelle nicht übergangen werden, daß es noch andere Modulkonzepte in funktionalen Sprachen gibt, so z.B. STRUCTURES in STANDARD ML [MTH90] oder FRAMES in KIR [Rei95]. Ein wesentlicher Unterschied dieser Konzepte zu dem von MODULA liegt darin, daß sie es ermöglichen, Module zu parametrisieren und sogar Module zur Laufzeit zu modifizieren. Da nach Meinung des Autors diese Eigenschaften bei numerischen Anwendungen eine eher untergeordnete Rolle spielen, soll hier auf diese Systeme nicht näher eingeangen werden.

Kapitel 3

Das Sprachdesign von SAC

Die Betrachtungen der vorangegangenen Abschnitte zeigen nicht nur auf, daß keine der existierenden funktionalen Sprachen optimal an die Bedürfnisse numerischer Anwendungen angepaßt ist, sondern führen auch zu mehreren Vorgaben für das Sprachdesign von SAC:

- Die Sprache soll so weit wie möglich in Syntax und Semantik einer der bekannteren imperativen Sprachen gleichen, um einen möglichst hohen Grad an Akzeptanz zu erreichen. Die Wahl einer solchen Sprache erleichtert nicht nur dem konventionellen Programmierer das Verständnis, sondern ermöglicht zugleich eine einfachere Compilation. Das Hauptproblem dabei liegt darin, der imperativen Syntax eine funktionale Semantik zu unterlegen, die mit der imperativen Semantik übereinstimmt und trotzdem im Vergleich zur imperativen Sprache möglichst wenige Restriktionen erfordert.

- Einer der Vorteile der funktionalen gegenüber den imperativen Ansätzen ist das ihnen inhärente, höhere Abstraktionsniveau. Im Kontext von Arrays bedeutet dies, vom Modell eines unterliegenden Speichers abstrahieren zu können und damit Speicherverwaltung auf der Ebene der Hochsprache überflüssig zu machen. Darüber hinaus garantiert die Seiteneffektfreiheit funktionaler Sprachen die Lokalität von Array-Operationen. Um dem Programmierer ein noch höheres Abstraktionsniveau im Umgang mit Arrays zu ermöglichen, sollen Arrays in SAC durch zwei Vektoren, nämlich den Datenvektor und den Shape-Vektor ähnlich wie in APL oder NIAL dargestellt werden (vergl. Abschnitt 2.1). Dies erlaubt die Spezifikation benutzerdefinierter Funktionen, die Arrays ohne Vorgaben über deren Dimensionalität als formale Parameter zulassen. Die primitiven Array-Operationen basieren auf einem Array-Kalkül, dem sog. Ψ-Kalkül [Mul88, Mul91, MJ91]. Er bietet neben der Anwendbarkeit aller Operationen auf Arrays jeder Dimensionalität einen Reduktions-Kalkül, der die Vereinfachung geschachtelter Operationen gestattet [Mul88, MT94]. Darüber hinaus

sollen Array-Comprehension-Konstrukte integriert werden, die ebenfalls auf n-dimensionale Arrays angewandt werden können.

- Zustände und Zustandsmanipulationen sollen derart integriert werden, daß die „Eleganz" der imperativen Sprachen im Umgang mit Zuständen sowohl in bezug auf die Spezifikation als auch in bezug auf die effiziente Compilierbarkeit erreicht wird, ohne daß es dabei zu Problemen mit der funktionalen Semantik kommt. Das Hauptproblem dieser Aufgabe liegt darin, daß der Umgang mit Zuständen in imperativen Sprachen auf der Möglichkeit beruht, Seiteneffekte zu erzeugen. Da diese aber gerade in dem SAC unterliegenden funktionalen Modell nicht vorhanden sind, bedarf es hier eines Konzeptes, das beide Modelle verbindet. Als Grundlage dazu dient der auf Uniqueness-Typen basierende Ansatz von CLEAN, da er nicht-monolithisches I/O und destruktives Überschreiben von Datenstrukturen direkt unterstützt.

- Gerade im Bereich der numerischen Anwendungen gibt es viele wiederverwendbare Programmkomponenten wie z.B. Funktionen zum Lösen von Gleichungssystemen, Eigenwertberechnungen, Approxomation der Lösungen partieller Differentialgleichungen etc., die von verschiedenen Programmiersystemen bereits als Bibliotheksfunktionen angeboten werden. Es bedarf daher eines Modulsystems, das neben den in modernen Modulsystemen üblichen Möglichkeiten wie „information hiding" oder „separate compilation" eine universelle Schnittstelle zu anderen Programmiersprachen bereitstellt. Wichtig sind hierbei zum einen eine möglichst große Kompatibilität der verwendeten Datenstrukturen sowie zum anderen die Möglichkeit, Bibliotheksfunktionen, die in anderen Programmiersprachen geschrieben wurden, von SAC aus verwenden zu können, ohne daß der Entwurf von Interface-Funktionen notwendig wird. Da insbesondere auch seiteneffektbehaftete Funktionen auf diese direkte Weise integriert werden sollen, bedarf es eines Mechanismus, der die Berücksichtigung der Effekte solcher Funktionen ermöglicht, ohne die Konfluenzeigenschaft der funktionalen Semantik von SAC zu beeinträchtigen.

Aus mehreren Gründen fiel die Entscheidung für die imperative Sprache C als Grundlage für SAC. Einerseits ist für die meisten Hardware-Plattformen ein C-Compiler vorhanden, andererseits erlaubt C eine sehr maschinennahe Programmierung, was insbesondere bei der Compilation von Vorteil ist. Darüber hinaus sind viele C-Bibliotheken verfügbar, deren Integration durch die Kompatibilität der Datenstrukturen erleichtert wird.

Bevor die Integration von Array-Konstrukten, eines Modulsystems oder von Zuständen entwickelt wird, gilt es zunächst, eine Teilsprache von C zu identifizieren, der sich auf einfache Weise eine funktionale Semantik unterlegen läßt. Gelingt dies, ohne dabei Widersprüche zum Verhalten der entsprechenden C-Programme zu erzeugen, so ergeben sich daraus mehrere Vorteile:

- Dem Programmierer bleibt es überlassen, ob er sich ein funktionales oder ein imperatives Berechnungsmodell vorstellt.
- Die inhärent seiteneffektfreie funktionale Semantik gestattet Compiler-Optimierungen, die über das für imperative Sprachen übliche Maß hinausgehen [SW85, Can93].
- Die funktionale Semantik macht außerdem nebenläufig ausführbare Programmteile explizit.
- Die Ähnlichkeit zu C beschränkt im Fall einer sequentiellen Ausführung die Compilation auf die Übersetzung der neu eingeführten Konstrukte (Arrays, Modulsystem etc.).

Im folgenden soll nun Schritt für Schritt eine solche Teilsprache von C als Kern von SAC identifiziert werden.

3.1 Eine funktionale Semantik für C

Als imperative Programmiersprache bietet C neben mehrfachen Zuweisungen diverse Konstrukte, die Seiteneffekte verursachen können, wie z.B. Schleifen, Sprünge und Zeiger auf Speicherbereiche. Die Vielfalt der Möglichkeiten, Seiteneffekte zu erzeugen, besonders im Zusammenhang mit Zeigern und den sog. Casts, lassen den Versuch aussichtslos erscheinen, eine konzise funktionale Beschreibung der Semantik von C zu finden. Bei genauerer Betrachtung zeigt sich jedoch, daß es möglich ist, mittels einiger weniger Restriktionen eine Teilsprache von C zu identifizieren, deren Semantik sich auf einfache Weise funktional beschreiben läßt.

Zentrale Konstruktionselemente von C sind die sog. Statements. Sie können durch Semikoli verbunden und mittels geschweifter Klammern zu sog. Statement-Blöcken zusammengefaßt werden. Die Semantik dieser Statement-Blöcke wird durch eine axiomatische Semantik beschrieben, die auf einer streng sequentiellen Ausführung der einzelnen Statements beruht. Im wesentlichen bietet C drei verschiedene Arten von Statements: Zuweisungen, Schleifen und IF-THEN-ELSE-Konstrukte. Diese Konstrukte finden sich in ähnlicher Form in funktionalen Sprachen in Form von LET-Blöcken, tail-end-rekursiven Funktionen sowie IF-THEN-ELSE-Ausdrücken wieder. Der entscheidende Unterschied besteht jedoch darin, daß anstelle einer auf dem Kontrollflußmodell basierenden imperativen Semantik eine funktionale Semantik unterlegt wird, die verschiedene Berechnungsfolgen zuläßt.

Im folgenden soll gezeigt werden, daß es möglich ist, diesen Sprachkonstrukten von C eine funktionale Semantik zu unterlegen, bei der die Ausführungsreihenfolge von Berechnungen nur noch durch kausale Abhängigkeiten, d.h. durch die Verfügbarkeit von Operanden bestimmt wird. Dazu wird in zwei Schritten vorgegangen. Zunächst wird eine einfache funktionale Sprache, im weiteren mit $\mathcal{F}$UN

bezeichnet, eingeführt. Anschließend wird der Sprachkern von SAC definiert und schrittweise ein Transformationsschema entwickelt, das SAC-Programme in $\mathcal{F}$UN-Programme abbildet. Dieses Transformationsschema ist so gestaltet, daß die durch die Semantik von $\mathcal{F}$UN definierte funktionale Semantik für SAC-Programme nicht zu Widersprüchen mit der Semantik äquivalenter C-Programme führt; d.h. für jedes SAC-Programm P, das ausschließlich aus C-Sprachkonstrukten besteht, gilt: P hat eine durch $\mathcal{F}$UN definierte Bedeutung Q, gdw. der RETURN-Wert der MAIN-Funktion von P durch die C-Semantik den Wert Q liefert.

3.1.1 Eine einfache funktionale Sprache $\mathcal{F}$UN

$\mathcal{F}$UN ist eine einfache funktionale Sprache, die den Bedürfnissen einer semantikdefinierenden Zwischensprache für SAC angepaßt wurde. Neben Konstanten, Variablen, Abstraktionen (Funktionsdefinitionen) und Applikationen (Funktionsanwendungen) bietet $\mathcal{F}$UN LET- und IF-THEN-ELSE-Ausdrücke sowie einen primitiven Rekursionsoperator (LETREC-Ausdrücke). Die Syntax von $\mathcal{F}$UN ist in Abb. 3.1 dargestellt.

Expr	⇒	*Const*	(Konstanten)
	|	*Id*	(Variablen)
	|	*Prf*	
	|	LETREC *FunDef* [, *FunDef*]$^+$ IN *Expr*	
	|	LET [*Id* = *Expr*]$^+$ IN *Expr*	
	|	IF *Expr* THEN *Expr* ELSE *Expr*	
	|	*Expr* ([*Expr*]*)	(Applikation)
FunDef	⇒	*Id* = λ [*Id*]$^+$. *Expr*	(Abstraktion)
	|	*Id* = *Expr*	
Prf	⇒	+ | - | * | / | == | !=	
	|	< | <= | > | >=	

Abbildung 3.1: Eine einfache funktionale Sprache.

Dabei ist zu beachten, daß die Abstraktionen nur in sehr eingeschränkter Form verwendet werden können. Zum einen sind sie ausschließlich als Werte LETREC-definierter Variablen spezifizierbar, und zum anderen sind sie n-stellig; curryfizierte Darstellungen sind nicht zugelassen. Dies reflektiert die Beschränkung auf vollständige Anwendungen sowie den Ausschluß namenloser Funktionen. Darüber hinaus entspricht die Notation der Applikationen der in C üblichen Notation mit einer Klammerung der Argumente.

Vor der Definition der Semantik von $\mathcal{F}$UN-Ausdrücken wird der Begriff der freien Variablen eingeführt, um die Menge der legalen $\mathcal{F}$UN-Ausdrücke auf sog. wohlgeformte $\mathcal{F}$UN-Ausdrücke einschränken zu können, deren Funktionsdefinitionen (Abstraktionen) keine freien Variablen enthalten.

Definition 3.1.1 . *Seien Const die Menge der $\mathcal{F}$UN-Konstanten, Var die Menge der $\mathcal{F}$UN-Variablen und Prf die Menge der primitiven Funktionen in $\mathcal{F}$UN. Dann ist die* Menge der freien Variablen *eines $\mathcal{F}$UN-Ausdruckes e definiert als*

$$FV(e) := \begin{cases} \{\} & \text{falls } e \in Const \vee e \in Prf \\ \{e\} & \text{falls } e \in Var \\ FV(e_p) \cup FV(e_t) \cup FV(e_e) & \text{falls } e = \text{IF } e_p \text{ THEN } e_t \text{ ELSE } e_e \\ \bigcup_{i=1}^{n} FV(e_i) \cup (FV(e') \setminus \{v_1, \ldots, v_n\}) & \text{falls } e = \left\{ \begin{array}{l} \text{LET} \\ \quad v_1 = e_1 \\ \quad \vdots \\ \quad v_n = e_n \\ \text{IN } e' \end{array} \right. \\ \left(\bigcup_{i=1}^{n} FV(e_i) \cup FV(e') \right) \setminus \{v_1, \ldots, v_n\} & \text{falls } e = \left\{ \begin{array}{l} \text{LETREC} \\ \quad v_1 = e_1 \\ \quad \vdots \\ \quad v_n = e_n \\ \text{IN } e' \end{array} \right. \\ FV(e') \setminus \{v_1, \ldots, v_n\} & \text{falls } e = \lambda v_1 \ldots v_n . e' \end{cases} .$$

Eine Variable *v heißt* frei *in e, falls $v \in FV(e)$.*

Ein Vorkommen einer Variablen *v in einem $\mathcal{F}$UN-Ausdruck e heißt* frei*, g.d.w. für alle Unterausdrücke e' von e, die dieses Vorkommen von v enthalten, gilt: v ist frei in e'.*

Ein $\mathcal{F}$UN-Ausdruck e heißt wohlgeformt*, falls alle in e vorkommenden Abstraktionen e' keine freien Variablen enthalten, d.h. es gilt $FV(e') = \{\}$.*

3.1.1□

Zur Erläuterung der obigen Definition betrachten wir exemplarisch folgenden $\mathcal{F}$UN-Ausdruck e:

```
LETREC
  f = λv.v
IN f(v)
```

.

Obwohl `v` in e frei ist, ist nur das letzte Vorkommen von `v` in e frei. Die Vorkommen bei der Definition von `f` sind nicht frei, da $\mathtt{v} \notin FV(\lambda \mathtt{v}.\mathtt{v}) = \{\}$. Wegen $FV(\lambda \mathtt{v}.\mathtt{v}) = \{\}$ ergibt sich außerdem, daß e wohlgeformt ist.

Zur Beschreibung der Semantik von $\mathcal{F}$UN wollen wir ein sog. Deduktionssystem verwenden.

Definition 3.1.2 . *Ein* Deduktionssystem $D = (L, R)$ *besteht aus einer Sprache* L *und einer Menge* R *von Regeln der Form*

$$\frac{S_1 \ldots S_n}{S} \quad .$$

Regeln mit $n = 0$ *heißen* Axiome *und haben die Form*

$$\frac{}{S} \quad .$$

Die S_i *und* S *heißen* Sequenzen *und sind Formeln der Sprache* L.

Eine Sequenz $S \in L$ *ist aus einer Menge von Sequenzen* $A = \{S_1, \ldots, S_n \mid S_i \in L\}$ *in einem Deduktionssystem* $D = (L, R)$ ableitbar*:*

$$A \vdash_D S :\Longleftrightarrow \begin{cases} \frac{}{S} \in R \\ \vee\ \exists i : S_i = S \\ \vee\ \exists T_1, \ldots, T_k \in L : A \vdash_D T_i \wedge \frac{T_1 \ldots T_n}{S} \in R \end{cases} .$$

3.1.2□

Um die Semantik von $\mathcal{F}$UN über ein solches Deduktionssystem definieren zu können, betrachten wir Sequenzen der Form $e \to e'$, wobei sowohl e als auch e' $\mathcal{F}$UN-Ausdrücke sind. Anschaulich bedeutet eine solche Sequenz: *e läßt sich zu* e' *transformieren.* Darauf aufbauend soll anschließend die Bedeutung eines $\mathcal{F}$UN-Ausdruckes e dadurch definiert werden, daß die Ableitbarkeit einer Sequenz $e \to e'$ gefordert wird, wobei e' aus der Menge der Konstanten von $\mathcal{F}$UN sein muß. Im folgenden werden also zunächst schrittweise die Regeln $R_{\mathcal{F}\text{UN}}$ eines Deduktionssystemes $D_{\mathcal{F}\text{UN}} = (L_{\mathcal{F}\text{UN}}, R_{\mathcal{F}\text{UN}})$ mit $L_{\mathcal{F}\text{UN}} := \{e_1 \to e_2 \mid e_1, e_2 \in \mathcal{F}\text{UN}\}$ entwickelt.

Vorkommen LETREC-gebundener Variablen im Zielausdruck werden durch ihre Definition ersetzt. Dazu verwenden wir ein Substitutions-Schema $[v \Leftarrow e]b$, das alle freien Vorkommen der Variablen v in b durch den Ausdruck e ersetzt. Damit ergibt sich als LETREC-Regel

$$\text{LETREC} \quad : \quad \frac{\begin{bmatrix} f_1 \Leftarrow \text{LETREC } f_1 = e_1 \ldots f_n = e_n \text{ IN } e_1 \\ \vdots \\ f_n \Leftarrow \text{LETREC } f_1 = e_1 \ldots f_n = e_n \text{ IN } e_n \end{bmatrix} b \to e'}{\begin{array}{l} \text{LETREC} \\ \quad f_1 = e_1 \\ \quad \vdots \qquad \to e' \\ \quad f_n = e_n \\ \text{IN } b \end{array}} \quad .$$

Anschaulich läßt sich diese Regel folgendermaßen formulieren: Ein LETREC-Ausdruck mit Definitionen $f_1 = e_1, \ldots, f_n = e_n$ und Zielausdruck b kann zu e' transformiert werden, falls der Ausdruck b, nachdem alle freien Vorkommen der $f_1, \ldots, f_n$ durch ihre rekursiven Definitionen ersetzt worden sind, sich zu e' transformieren läßt. Operationell kann man daraus ableiten, daß die Berechnung eines LETREC-Ausdruckes zunächst eine Substitution der rekursiven Funktionsdefinitionen in den Rumpf des LETREC-Ausdruckes und dann eine Auswertung des entstehenden Ausdruckes erfordert.
Konstanten sowie Funktionsrümpfe sind Fixpunkte der Transformation. Sie werden durch die Axiome

$$\textsc{Const} \;:\; \frac{}{C \to C} \qquad \text{und}$$

$$\textsc{Fun} \;:\; \frac{}{\lambda v_1 \ldots v_n.e \to \lambda v_1 \ldots v_n.e} \qquad \text{beschrieben.}$$

Die Transformation eines LET-Ausdruckes ergibt sich aus der Transformation des Zielausdruckes, nachdem alle freien Vorkommen der definierten Variablen durch die transformierten Ausdrücke der jeweiligen Definitionen ersetzt worden sind:

$$\textsc{Let} \;:\; \frac{e_1 \to e_1' \;\; \ldots \;\; e_n \to e_n' \quad \left[\begin{array}{c} v_1 \Leftarrow e_1' \\ \vdots \\ v_n \Leftarrow e_n' \end{array}\right] e \to e'}{\text{LET } v_1 = e_1 \ldots v_n = e_n \text{ IN } e \to e'}$$

$$\Longleftrightarrow \forall_{i \in \{1..n\}}\; e_i' \in Const \qquad .$$

Für die Transformation von IF-THEN-ELSE-Ausdrücken ergibt sich

$$\textsc{Cond1} \;:\; \frac{e \to \text{TRUE} \quad e_t \to e_t'}{\text{IF e THEN } e_t \text{ ELSE } e_f \to e_t'} \qquad \textit{und}$$

$$\textsc{Cond2} \;:\; \frac{e \to \text{FALSE} \quad e_f \to e_f'}{\text{IF e THEN } e_t \text{ ELSE } e_f \to e_f'} \qquad .$$

Schließlich folgen die Regeln für Funktionsanwendungen. Für Anwendungen von Abstraktionen gilt

$$\textsc{ApFun} \;:\; \frac{e_f \to \lambda v_1 \ldots v_n.e \;\; e_1 \to e_1' \;\; \ldots \;\; e_n \to e_n' \;\; \begin{bmatrix} v_1 \Leftarrow e_1' \\ \vdots \\ v_n \Leftarrow e_n' \end{bmatrix} e \to e'}{e_f\ (\ e_1, \ldots, e_n) \to e'}$$

$$\Longleftrightarrow \forall_{i \in \{1..n\}}\ e_i' \in Const \qquad .$$

Es ist bei dieser Regel anzumerken, daß $\mathcal{F}\textsc{un}$ durch sie in zweierlei Hinsicht restringiert wird. Zum einen sind nur vollständige Anwendungen möglich, und zum anderen werden durch die Forderung, daß alle e_i' Konstanten sein müssen, Funktionen als Argumente ausgeschlossen. Für die Anwendung primitiver Funktionen ergeben sich diese Restriktionen ohnehin durch die Forderungen, daß deren Stelligkeiten mit der Anzahl der Argumente übereinstimmen und die Argumente im Definitionsbereich der Funktion liegen müssen. Während bei benutzerdefinierten Funktionen das oben eingeführte Substitutionsschema Anwendung findet, ergibt sich die Transformation bei primitiven Funktionen direkt aus dem Namen der Funktion und der Bedeutung der jeweiligen Argumente:

$$\textsc{ApPrf} \;:\; \frac{e_1 \to e_1' \;\; \ldots \;\; e_n \to e_n'}{f(\ e_1, \ldots, e_n) \to VAL(\ f(\ e_1, \ldots, e_n))}$$

$$\Longleftrightarrow (\ f \in Prf \wedge n = ARITY(f) \wedge (e_1, \ldots, e_n) \in DEF(f)) \qquad ,$$

wobei $ARITY$ die Stelligkeit, DEF den Definitionsbereich und VAL das Resultat einer Anwendung einer primitiven Funktion spezifiziert.

Zusammenfassend ergibt sich aus diesen Regeln die vollständige Spezifikation des Deduktionssystemes $D_{\mathcal{F}\textsc{un}} := (L_{\mathcal{F}\textsc{un}}, R_{\mathcal{F}\textsc{un}})$ mit $L_{\mathcal{F}\textsc{un}} := \{e_1 \to e_2 \mid e_1, e_2 \in \mathcal{F}\textsc{un}\}$ sowie $R_{\mathcal{F}\textsc{un}} = \{\textsc{Letrec}, \textsc{Const}, \textsc{Fun}, \textsc{Let}, \textsc{Cond1}, \textsc{Cond2}, \textsc{ApFun}, \textsc{ApPrf}\}$. Damit läßt sich jetzt die Bedeutung eines $\mathcal{F}\textsc{un}$-Ausdruckes definieren durch die

Definition 3.1.3 . *Seien $D_{\mathcal{F}\textsc{un}}$ das oben eingeführte Deduktionssystem und $Const$ die Menge der Konstanten aus $\mathcal{F}\textsc{un}$. Dann wird die Bedeutung eines wohlgeformten $\mathcal{F}\textsc{un}$-Ausdruckes S definiert durch*

$$m_{\mathcal{F}\textsc{un}} : \mathcal{F}\textsc{un} \longrightarrow_{part} Const$$

$$mit\ m_{\mathcal{F}\textsc{un}}(S) = S' :\Longleftrightarrow \{\} \vdash_{D_{\mathcal{F}\textsc{un}}} S \to S' \ \wedge \ S' \in Const \qquad .$$

3.1.3□

Obwohl $D_{\mathcal{F}\textsc{un}}$ nur die Semantik von $\mathcal{F}\textsc{un}$ definiert und keinerlei Ausführungsreihenfolge vorgibt, läßt sich aus $D_{\mathcal{F}\textsc{un}}$ auf direkte Weise ein sequentieller Ausführungsmechanismus ableiten:

- suche eine Regel $\frac{S_1 \ldots S_n}{S}$ aus $R_{\mathcal{F}\text{UN}}$ heraus, für die S auf das zu berechnende Programm vollständig paßt. Außer bei IF-THEN-ELSE-Ausdrücken gibt es maximal eine solche Regel. Im Falle eines IF-THEN-ELSE-Ausdruckes entscheidet die Auswertung des Prädikates;
- berechne sukzessive die Teilausdrücke $S_1 \ldots S_n$;
- fahre solange rekursiv fort, bis keine Unterausdrücke mehr zu berechnen sind.

Auf diese Weise wird ein Applicative-Order-Mechanismus festgelegt, da in den Regeln LET, APFUN und APPRF grundsätzlich nur bereits berechnete Ausdrücke e_i substituiert werden. Gerade diese drei Regeln sind auch die entscheidenden Regeln, um nebenläufig ausführbare Programmteile in $\mathcal{F}$UN-Programmen zu identifizieren, da sie mehrere voneinander unabhängig ausführbare (Argument-) Berechnungen erfordern.

Nach der Einführung der funktionalen Sprache $\mathcal{F}$UN kommen wir jetzt zur Beschreibung des Sprachkernes von SAC sowie zur Entwicklung eines Transformationsschemas von SAC-Programmen in $\mathcal{F}$UN-Ausdrücke.

3.1.2 Der Sprachkern von SAC

Die Syntax des Sprachkernes von SAC ist in Abb. 3.2 dargestellt. Sie umfaßt die zentralen Sprachkonstrukte von C wie Funktionsdefinitionen, Zuweisungen sowie Schleifen- und IF-THEN-ELSE-Konstrukte. Die das Typsystem von SAC betreffenden Ableitungen für $TypeDef$, $Type$ und $VarDec$ sind in Abb. 3.2 nicht angegeben, da sie für diesen Abschnitt nicht weiter von Belang sind. Die entsprechenden Ergänzungen finden sich sowohl im Abschnitt 3.2 über das Typsystem von SAC als auch im Anhang A.

Um die Semantik für diese Kernsyntax zu definieren, wird in diesem und in den nächsten drei Abschnitten schrittweise ein Transformationsschema $\mathcal{TF}_K$entworfen, das jedem SAC-Programm einen $\mathcal{F}$UN-Ausdruck zuordnet. Da bei der Transformation von Zuweisungen in LET-Blöcke eine Variablenmenge als Kontext benötigt wird, wird das Transformationsschema durch eine zweistellige Funktion $\mathcal{TF}_K$: SAC $\times$ $Ids \rightarrow \mathcal{F}$UN definiert, die ein SAC-Programm und eine initial leere Menge von Identifikatoren in einen $\mathcal{F}$UN-Ausdruck abbildet. Mit Hilfe dieser Transformation soll schließlich die Semantik eines SAC-Programmes P_{SAC} als $\text{m}_{\mathcal{F}\text{UN}}(\mathcal{TF}_K(P_{\text{SAC}}, \emptyset))$ definiert werden.

Im Gegensatz zu SAC ist $\mathcal{F}$UN eine ungetypte Sprache, d.h. die Typüberprüfungen erfolgen lediglich bei der Anwendung primitiver Funktionen auf Argumente (PRF-Regel aus Abschnitt 3.1.1) sowie bei der Interpretation von Prädikate von IF-THEN-ELSE-Ausdrücken (COND1- bzw. COND2-Regel aus Abschnitt 3.1.1). Bei der Transformation eines SAC-Programmes in einen $\mathcal{F}$UN-Ausdruck werden daher alle Typinformationen vernachlässigt. Dies reflektieren die Transformationsregeln

$$
\begin{array}{lcl}
Program & \Rightarrow & \left[\, TypeDef \,\right]^{*} \left[\, FunDef \,\right]^{*} \; Type \;\; \text{MAIN} \;\; (\;) \;\; ExprBlock \\
FunDef & \Rightarrow & Type \left[\, , \;\; Type \,\right]^{*} \; FunId \;\; (\; \left[\, ArgDef \,\right] \;) \;\; ExprBlock \\
ArgDef & \Rightarrow & Type \;\; Id \left[\, , \;\; Type \;\; Id \,\right]^{*} \\
ExprBlock & \Rightarrow & \{ \;\; \left[\, Vardec \,\right]^{*} \left[\, Assign \,\right]^{*} \; RetAssign \;\; \} \\
Assign & \Rightarrow & Id \left[\, , \;\; Id \,\right]^{*} \;\; \texttt{=} \;\; Expr \;\; ; \\
 & | & SelAssign \;\; ; \\
 & | & ForAssign \;\; ; \\
RetAssign & \Rightarrow & \text{RETURN} \;\; (\; Expr \left[\, , \;\; Expr \,\right]^{*} \;\;) \;\; ; \\
SelAssign & \Rightarrow & \text{IF} \;\; (\; Expr \;\;) \; AssignBlock \; [\, \text{ELSE} \;\; AssignBlock] \\
ForAssign & \Rightarrow & \text{DO} \;\; AssignBlock \;\; \text{WHILE} \;\; (\; Expr \;\;) \\
 & | & \text{WHILE} \;\; (\; Expr \;\;) \;\; AssignBlock \\
 & | & \text{FOR} \;\; (\; Assign \;\; ; \;\; Expr \;\; ; \;\; Assign \;\;) \;\; AssignBlock \\
AssignBlock & \Rightarrow & Assign \\
 & | & \{ \left[\, Assign \,\right]^{*} \;\; \} \\
Expr & \Rightarrow & Const \\
 & | & Id \\
 & | & Id \;\; (\; Expr \left[\, , \;\; Expr \,\right]^{*} \;\;) \\
 & | & Prf \;\; (\; Expr \left[\, , \;\; Expr \,\right]^{*} \;\;) \\
Prf & \Rightarrow & \texttt{+} \;\; | \;\; \texttt{-} \;\; | \;\; \texttt{*} \;\; | \;\; \texttt{/} \;\; | \;\; \texttt{==} \;\; | \;\; \texttt{!=} \\
 & | & \texttt{<} \;\; | \;\; \texttt{<=} \;\; | \;\; \texttt{>} \;\; | \;\; \texttt{>=}
\end{array}
$$

Abbildung 3.2: Die Kernsyntax von SAC.

$$\mathcal{TF}_K(\; TypeDef_1 \ldots TypeDef_n \; Prg, \emptyset) \mapsto \mathcal{TF}_K(\; Prg, \emptyset) \quad ,$$

$$\mathcal{TF}_K(\; \text{Type MAIN} \;\; () \;\; ExprBlock, \emptyset) \mapsto \mathcal{TF}_K(\; ExprBlock, \emptyset) \quad ,$$

$$\mathcal{TF}_K(\ FunDef_1 \ldots FunDef_n \text{ Type MAIN } () \ \ ExprBlock, \emptyset)$$
$$\mapsto \left\{ \begin{array}{l} \text{LETREC} \\ \quad \mathcal{TF}_K(FunDef_1\text{ , }\emptyset) \\ \quad \vdots \\ \quad \mathcal{TF}_K(FunDef_n\text{ , }\emptyset) \\ \text{IN } \mathcal{TF}_K(\ ExprBlock\text{ , }\emptyset) \end{array} \right. ,$$

$$\mathcal{TF}_K(\ Type_1, \ldots, Type_n\ FunId(\ Ty_1\ Id_1, \ldots, Ty_m\ Id_m)\ ExprBlock, \emptyset)$$
$$\mapsto\ FunId = \lambda Id_1 \ldots Id_m.\ \mathcal{TF}_K(\ ExprBlock, \emptyset)$$

$$\mathcal{TF}_K(\ \{\ VarDec_1 \ldots VarDec_n\ RestExprBlock, \emptyset)$$
$$\mapsto\ \ \mathcal{TF}_K(RestExprBlock, \emptyset) \qquad .$$

Die Vernachlässigung der Typinformationen bei der Transformation von SAC-Programmen in $\mathcal{F}$UN-Ausdrücke hat zur Konsequenz, daß auch nicht-typbare SAC-Programme durch das Schema $\mathcal{TF}_K$ in $\mathcal{F}$UN-Ausdrücke transformiert werden, die gemäß Def. 3.1.3 eine Bedeutung haben. Die Definition der Semantik von SAC kann daher nicht ausschließlich auf der Transformation in $\mathcal{F}$UN-Ausdrücke beruhen, sondern bedarf zuvor einer Beschränkung auf SAC-Programme, die bezüglich des Typsystems von SAC keine Typfehler aufweisen. Eine formale Definition der Semantik von SAC wird deshalb erst in dem Abschnitt 3.2 über das Typsystem von SAC eingeführt. Zunächst wird die schrittweise Vervollständigung des Transformationsschemas $\mathcal{TF}_K$ behandelt.

3.1.3 Mehrfache Zuweisungen

In C werden Variablen als Identifikatoren für Speicherplatzadressen verstanden. Eine Zuweisung an eine Variable ist in diesem Kontext als ein Überschreiben des entsprechenden Speichers zu interpretieren. Mehrfache Zuweisungen an ein und dieselbe Variable sind deshalb möglich, weil ohnehin eine sequentielle Abarbeitungsreihenfolge garantiert wird.

In funktionalen Sprachen steht eine Variable für sich selbst. Eine „Zuweisung“ eines Wertes an eine Variable (LET-Konstrukt) wird als Ersetzung der freien Vorkommen der Variablen im Zielausdruck durch den Wert verstanden. Eine wiederholte Zuweisung an „eine“ Variable kann also nur durch eine Schachtelung von Zuweisungen an konzeptuell verschiedene Variablen gleichen Namens modelliert werden. Als Bindungsbereiche für diese Variablen im funktionalen Sinn ergeben sich damit alle Statements zwischen zwei aufeinander folgenden Zuweisungen an sie, sowie die rechte Seite der zweiten Zuweisung. Eine Transformation von mehrfachen Zuweisungen in geschachtelte LET-Blöcke läßt sich deshalb folgendermaßen algorithmisieren:

Es werden, am Anfang eines Statement-Blockes beginnend, solange Zuweisungen als Definitionen in den zu erzeugenden LET-Block übernommen, bis entweder eine

Zuweisung an eine Variable bereits bekannten Namens erfolgt oder aber eine der definierten Variablen auf der rechten Seite einer Zuweisung benutzt wird. Dies wird rekursiv solange fortgesetzt, bis das Ende des Statement-Blockes erreicht ist.

Um diesen Algorithmus zu formalisieren, nutzen wir den zweiten Parameter des Transformationsschemas $\mathcal{TF}_K$ (im folgenden mit V_{Akt} bezeichnet), um diejenigen Variablennamen zu akkumulieren, für die wir im „aktuellen" LET-Block bereits Zuweisungen erzeugt haben. Damit ergibt sich für das Transformationsschema $\mathcal{TF}_K$ bei Zuweisungen $v = e$; und einem Rest-Programm R:

- Falls dies die erste Zuweisung eines LET-Blockes ist (d.h. $\mathrm{V}_{Akt} = \emptyset$), erzeuge das Schlüsselwort LET vor der Zuweisung, übernimm die definierte Variable v in V_{Akt} und fahre mit dem Rest des Programmes fort.
- Ist bereits ein LET-Block begonnen worden, muß entschieden werden, ob ein neuer LET-Block erforderlich ist oder nicht. Ein LET-Block darf nur dann fortgesetzt werden, wenn v noch nicht im aktuellen LET-Block definiert ist, d.h. $v \notin \mathrm{V}_{Akt}$, und auch keine der in e verwendeten Variablen im aktuellen LET-Block definiert ist ($\mathrm{V}_{Akt} \cap VARS(e) = \emptyset$). Andernfalls ist ein neuer LET-Block zu erzeugen sowie V_{Akt} entsprechend anzupassen.

Als Transformationsregel ergibt sich daraus

$$\mathcal{TF}_K(\ v = e;\ R,\ \mathrm{V}_{Akt}) \mapsto \begin{cases} \texttt{LET} \\ \quad v \texttt{ = } e \\ \quad \mathcal{TF}_K(\ R\texttt{ , }\{v\}) & \text{falls } \mathrm{V}_{Akt} = \emptyset \\ \\ v \texttt{ = } e \\ \mathcal{TF}_K(\ R\texttt{ , }\{v\} \cup\ \mathrm{V}_{Akt}) & \text{falls } \begin{array}{l}(v \notin \mathrm{V}_{Akt}) \\ \wedge(\mathrm{V}_{Akt} \cap VARS(e) = \emptyset)\end{array} \\ \\ \texttt{IN LET} \\ \quad v \texttt{ = } e \\ \quad \mathcal{TF}_K(\ R\texttt{ , }\{v\}) & \textit{sonst} \end{cases},$$

wobei $VARS(e)$ die Menge der in e frei vorkommenden Variablen beschreibt.

Bei der Transformation einer RETURN-Anweisung ist zu beachten, ob diese den Abschluß eines LET-Blockes darstellt (d.h. die Variablenmenge ist nicht leer) oder nicht:

$$\mathcal{TF}_K(\ \text{RETURN}(e);\},\ \mathrm{V}_{Akt}) \mapsto \begin{cases} e & \text{falls } \mathrm{V}_{Akt} = \emptyset \\ \\ \text{IN}\ \ e & \text{sonst} \end{cases} .$$

Um beim „Übergang“ von Zuweisungen zu IF-THEN-ELSE-Konstrukten oder zu Schleifen einen korrekten Abschluß von bereits begonnen LET-Ausdrücken sicherzustellen, bedarf es der Regel

$$\mathcal{TF}_K(\ X, \{v_1, \ldots, v_n\}) \quad \text{mit } n > 0 \\ \mapsto\ \text{IN}\ \ \mathcal{TF}_K(\ X, \emptyset) \quad ,$$

wobei X für SAC-Teilprogramme steht, die weder mit einer Zuweisung noch mit einem RETURN-Ausdruck beginnen.

Beispiel:

```
{ a =3;                                   LET
  b =7;                                     a=3
  c =a+3;                                   b=7
  b =a+19;          wird transformiert in IN LET
  c =42;                                      c=a+3
  ...                                         b=a+19
}                                           IN LET
                                               c=42
                                            IN ...
```

3.1.4 IF-THEN-ELSE-Konstrukte

Bei den aus C bekannten IF-THEN-ELSE-Konstrukten besteht die Schwierigkeit darin, daß die auf das IF-THEN-ELSE-Konstrukt folgenden Anweisungen unabhängig von der Selektion zwischen den beiden Alternativen ausgeführt werden. Dadurch kommt es zu Problemen mit Variablen, denen im IF-THEN-ELSE-Konstrukt Werte zugewiesen werden. Werden diese Variablen in dem Programmtext, der dem IF-THEN-ELSE-Konstrukt folgt, referenziert, so ist nicht mehr eindeutig, welches die zugehörige Definition ist; das Konzept der statischen Bindungen geht also an dieser Stelle verloren.

Um das Verhalten der IF-THEN-ELSE-Konstrukte funktional korrekt zu modellieren, muß also die Eindeutigkeit der Bindungsbereiche von Variablen sichergestellt werden. Dies kann dadurch erreicht werden, daß sämtlicher Programmtext, der dem IF-THEN-ELSE-Konstrukt folgt, in beide Äste des IF-THEN-ELSE-Konstrukte hineinkopiert wird. Durch die dabei entstehenden Kopien der angewandten Variablenvorkommen wird eine eindeutige Zuordnung der Variablen zu den zugehörigen Zuwei-

sungen möglich. Dies führt zu

$$\mathcal{TF}_K(\ \text{IF}\ (e)\ \{A_t;\};\ R,\emptyset) \mapsto \begin{cases} \text{IF}\ (e) \\ \text{THEN}\ \mathcal{TF}_K(\ A_t;\ R,\ \emptyset) \\ \text{ELSE}\ \mathcal{TF}_K(\ R,\ \emptyset) \end{cases} \qquad \textit{sowie}$$

$$\mathcal{TF}_K(\ \text{IF}\ (e)\ \{A_t\}\ \text{ELSE}\ \{A_f\};\ R,\emptyset) \mapsto \begin{cases} \text{IF}\ (e) \\ \text{THEN}\ \mathcal{TF}_K(\ A_t;\ R,\ \emptyset) \\ \text{ELSE}\ \mathcal{TF}_K(\ A_f;\ R,\ \emptyset) \end{cases}$$

als $\mathcal{TF}_K$-Regeln für IF-THEN-ELSE-Konstrukte.

Beispiel:

```
{ a = 3;
  IF (b) {
    a = 42;
  };
  RETURN(a);
}
```

wird transformiert in

```
LET
  a = 3
IN IF (b)
   THEN LET
          a = 42
        IN a
   ELSE a
```

.

3.1.5 Schleifenkonstrukte

Schleifenkonstrukte sind abkürzende Notationen für tail-end-rekursive Funktionen. Das Problem bei der Verwendung von Schleifen, wie C sie gestattet, liegt darin, daß

1. die im Schleifenrumpf „modifizierten“ Variablen nicht explizit initialisiert werden müssen;
2. zwischen der aktuellen und vorherigen Instanz einer Schleifenvariablen nicht syntaktisch unterschieden wird;
3. mehrfache Zuweisungen an ein und dieselbe Variable innerhalb des Schleifenrumpfes erfolgen können;
4. Seiteneffekte auf außerhalb von Schleifen gebundene Variablen durch Zuweisungen innerhalb der Schleifen enstehen können.

Diese Freiheiten bei der Verwendung von Schleifen in C erschweren die Konstruktion einer tail-end-rekursiven Funktion als äquivalentes Substitut für eine Schleife. Betrachten wir z.B. eine WHILE-Schleife schematisch, so ist es das Ziel der Transformation, einen Statement-Block der Form $\{\ A;\ \text{WHILE}(p)\{B\};\ R\}$, wobei A, B und

R für SAC-Statement-Folgen stehen, durch einen äquivalenten Statement-Block der Form $\{\ A;\ y_1, \ldots, y_m = f(x_1, \ldots, x_n);\ R\}$ zu ersetzen. Dabei soll f eine tail-end-rekursive Funktion sein, deren Rumpf die folgende Form hat:

```
{ IF(p) {
    B;
    y1, ..., ym = f(x1, ..., xn);
  }
  RETURN(y1, ..., ym);
}
```

Das Problem dieser Aufgabe liegt darin, die Variablenmengen x_1, ..., x_n sowie y_1, ..., y_m geeignet zu bestimmen. Geeignet bedeutet in diesem Zusammenhang, daß, aus imperativer Sicht gesehen, die Gesamtheit der während der Ausführung der Schleife erfolgenden Variablen-Modifikationen unabhängig von der Anzahl der Schleifendurchläufe durch die Zuweisung $y_1, \ldots, y_m = f(x_1, \ldots, x_n)$; erfaßt wird.

Für die Menge der „Ergebnisvariablen" y_1, ..., y_m bedeutet dies: Sie muß alle im Schleifenrumpf definierten Variablen umfassen, zumindest, soweit diese im Rest R des Statement-Blockes noch benötigt werden. Die Menge der „Parametervariablen" x_1, ..., x_n ergibt sich im wesentlichen aus den im Schleifenrumpf benötigten Variablen. Darüber hinaus muß sie jedoch auch sämtliche „Ergebnisvariablen" enthalten, damit die tail-end-rekursive Funktion f auch dann die gewünschten Resultate liefert, wenn der Schleifenrumpf B kein einziges Mal durchlaufen wird.

Um die Bestimmung dieser Variablenmengen zu formalisieren, werden zunächst zwei Funktionen eingeführt. Eine Funktion *Defs*, mit deren Hilfe sich die Menge der in einem SAC-Programm definierten Variablen charakterisieren läßt, und eine Funktion *Refs*, die eine Bestimmung der in einem SAC-Programm benötigten Variablen ermöglicht.

Die Funktion *Defs* ermittelt für eine Variable a und ein SAC-Programm e, ob a auf der linken Seite einer Zuweisung vorkommt bzw., wenn dies der Fall ist, ob sie, im Sinne von C, auf jeden Fall oder nur unter bestimmten Laufzeitbedingungen definiert wird (d.h. die Zuweisung befindet sich in nur einem Zweig eines IF-THEN-ELSE-Statements). Als Ergebnis liefert *Defs* (a, e) entweder den Wert $false$ (wird nicht definiert), den Wert $true$ (wird auf jeden Fall definiert) oder den Wert *pot* (wird eventuell definiert).

Definition 3.1.4 . *Die Funktion*

$$Defs : Id \times \text{SAC} \longrightarrow \{true, false, pot\}$$

ist definiert durch:

$$Defs\ (a,\ \text{return}(e_1, \ldots, e_n)) \mapsto false \quad ,$$

$$Defs\ (a,\ v_1, \ldots, v_n = e;\ R) \mapsto \begin{cases} true & \text{falls } \exists i \in \{1, \ldots, n\} : v_i = a \\ Defs\ (a,\ R) & \text{sonst} \end{cases} ,$$

$$Defs\ (a, \text{IF}(e)\{A_t\}\ \text{ELSE}\ \{A_e\};\ R) \mapsto \begin{cases} true & \text{falls } \left(\begin{array}{c} Defs\ (a, R) = true \\ \vee\ (Defs\ (a, A_t) = true \wedge\ Defs\ (a, A_e) = true) \end{array} \right) \\ false & \text{falls } \left(\begin{array}{c} Defs\ (a, R) = false \wedge Defs\ (a, A_t) = false \\ \wedge\ Defs\ (a, A_e) = false \end{array} \right) \\ pot & \text{sonst} \end{cases} ,$$

$$Defs\ (a, \text{DO}\{A\}\ \text{WHILE}(e);\ R) \mapsto Defs\ (a,\ A;\ R) \quad ,$$

$$Defs\ (a, \text{WHILE}(e)\{A\};\ R) \mapsto \begin{cases} \text{true} & \text{falls } Defs\ (a, R) = true \\ \text{false} & \text{falls } (Defs\ (a, R) = false \wedge Defs\ (a, A) = false) \\ \text{pot} & \text{sonst} \end{cases} ,$$

$$Defs\ (a, \text{FOR}(A_1;\ e;\ A_2)\{A\};\ R) \mapsto Defs\ (a,\ A_1;\ \text{WHILE}(e)\{A;\ A_2;\}\ R) \quad .$$

3.1.4□

Die Funktion *Refs* bestimmt für eine Variable a und ein SAC-Programm e, wie oft a in e „benötigt" wird. Dazu wird die Anzahl der Verwendungen auf der rechten Seite von Zuweisungen bzw. in Prädikatausdrücken vor einer „erneuten Definition" der Variablen, d.h. einer Verwendung auf der linken Seite einer Zuweisung gezählt. Bei IF-THEN-ELSE-Konstrukten wird das Maximum der Vorkommen in den beiden Zweigen gebildet und bei Schleifen wird anstelle der Schleifenrümpfe bereits der Aufruf der entsprechenden tail-end-rekursiven Funktion zugrunde gelegt. Diese Vorgehensweise führt zwar zu einer rekursiven Definition, ermöglicht jedoch eine Wiederverwendung bei der Compilation von Array-Konstrukten (vergl. Abschnitt 4.3). Formal ergibt sich die

Definition 3.1.5 . *Die Funktion*

$$Refs : Id \times \text{SAC} \longrightarrow \mathbb{N}$$

ist definiert durch

$$Refs\ (a,\ Var) \mapsto \begin{cases} 1 & \text{falls } Var = a \\ 0 & \text{sonst} \end{cases} ,$$

$$Refs\ (a,\ Const) \mapsto 0 \quad ,$$

$$\mathit{Refs}\,(a,\, a(e_1,\ldots,e_n)) \mapsto \sum_{i=1}^{n} \mathit{Refs}\,(a,\, e_i) \quad ,$$

$$\mathit{Refs}\,(a,\, \text{RETURN}(e_1,\ldots,e_n)) \mapsto \sum_{i=1}^{n} \mathit{Refs}\,(a,\, e_i) \quad ,$$

$$\mathit{Refs}\,(a,\, v_1,\ldots,v_n = e;\, R) \mapsto \begin{cases} \mathit{Refs}\,(a,\, e) & \text{falls } a \in \{v_1,\ldots,v_n\} \\ \mathit{Refs}\,(a,\, R) + \mathit{Refs}\,(\mathrm{a},\, e) & \text{sonst} \end{cases} ,$$

$$\begin{aligned} &\mathit{Refs}\,(a,\, \text{IF}(e)\{A_t\}\ \text{ELSE}\ \{A_e\};\, R) \\ &\quad \mapsto \max(\mathit{Refs}\,(a,\, A_t;\, R),\, \mathit{Refs}\,(a,\, A_t;\, R)) + \mathit{Refs}\,(a,\, e) \quad , \end{aligned}$$

$$\begin{aligned} &\mathit{Refs}\,(a,\, \text{IF}(e)\{A_t\};\, R) \\ &\quad \mapsto \max(\mathit{Refs}\,(a,\, A_t;\, R),\, \mathit{Refs}\,(a,\, R)) + \mathit{Refs}\,(a,\, e) \quad , \end{aligned}$$

$$\begin{aligned} &\mathit{Refs}\,(a,\, \text{DO}\ \{A\}\ \text{WHILE}(e);\, R) \\ &\quad \mapsto \mathit{Refs}\,(a,\, y_1,\, \ldots,\, y_m = \mathit{dummy}(\, x_1,\, \ldots,\, x_n);\, R) \quad , \end{aligned}$$

wobei $x_1, \ldots, x_n, y_1, \ldots, y_m$ und dummy gemäß der entsprechenden $\mathcal{TF}_K$-Regel zu bestimmen sind[1].

$$\begin{aligned} &\mathit{Refs}\,(a,\, \text{WHILE}(e)\{A\};\, R) \\ &\quad \mapsto \mathit{Refs}\,(a,\, y_1,\, \ldots,\, y_m = \mathit{dummy}(\, x_1,\, \ldots,\, x_n);\, R) \quad , \end{aligned}$$

wobei $x_1, \ldots, x_n, y_1, \ldots, y_m$ und dummy gemäß der entsprechenden $\mathcal{TF}_K$-Regel zu bestimmen sind[1].

$$\begin{aligned} &\mathit{Refs}\,(a,\, \text{FOR}\ (\ A_1;\, e;\, A_2)\ \{\ A;\};\, R)\text{A} \\ &\quad \mapsto \mathit{Refs}\,(a,\, A_1;\, \text{WHILE}(\ e)\ \{\ A;\, A_2;\};\, R) \quad . \end{aligned}$$

3.1.5□

Mit Hilfe dieser Funktionen lassen sich jetzt aus SAC-Programmen die für eine Transformation von SAC-Schleifen in tail-end-rekursive Funktionen benötigten Variablenmengen ableiten. Die Menge der in einem SAC-(Teil-)Programm A definierten Variablen läßt sich formal beschreiben durch

$$\mathrm{V}_{def}(A) := \{v \mid \mathit{Defs}(v, A) = \mathit{true} \ \vee \ \mathit{Defs}(v, A) = \mathit{pot}\} \quad .$$

Die Menge der potentiell in A definierten Variablen als

$$\mathrm{V}_{pot\_def}(A) := \{v \mid \mathit{Defs}(v, A) = \mathit{pot}\} \quad ,$$

[1] Obwohl es sich hier um eine wechselseitige Definition handelt, ist sie sinnvoll, da stets Ausdrücke geringerer Länge betrachtet werden.

und schließlich die Menge der in A benötigten Variablen

$$\mathrm{V}_{need}(A) := \{v \mid Refs(v, A) > 0\} \quad .$$

Für die Transformation der oben eingeführten schematischen Darstellung von Statement-Blöcken mit WHILE-Schleife $\{\ A;\ \text{WHILE}(\ p)\{\ B\};\ R\}$ in tail-end-rekursive Funktionen ergibt sich damit: Die Menge „Ergebnisvariablen“ wird durch $(\mathrm{V}_{def}(B) \cap \mathrm{V}_{need}(R))$ und die Menge der „Parametervariablen“ durch $\mathrm{V}_{need}(B) \cup (\mathrm{V}_{def}(B) \cap \mathrm{V}_{need}(R))$ charakterisiert. Als Transformationsregeln $\mathcal{TF}_K$ für Schleifen erhalten wir

$$\mathcal{TF}_K(\ \text{WHILE}(\ p)\ \{\ B;\};\ R,\ \emptyset)$$
$$\mapsto \left\{\begin{array}{l} \text{LETREC} \\ \quad dummy = \lambda x_1 \ldots x_n . \\ \qquad\qquad \mathcal{TF}_K\left(\begin{array}{l}\text{IF}(\ p)\ \{ \\ \quad B; \\ \quad y_1, \ldots, y_m\ =\ dummy(\ x_1, \ldots, x_n), \emptyset \\ \}\\ \text{RETURN}(\ y_1, \ldots, y_m);\end{array}\right) \\ \text{IN LET} \\ \quad y_1, \ldots, y_m = dummy(\ x_1, \ldots, x_n) \\ \quad \mathcal{TF}_K(\ R,\ \{y_1, \ldots, y_m\}) \end{array}\right. ,$$

wobei $\{x_i \mid i \in \{1, \ldots, n\}\} = \mathrm{V}_{need}(B) \cup (\mathrm{V}_{def}(B) \cap \mathrm{V}_{need}(R))$,

$\{y_i \mid i \in \{1, \ldots, m\}\} = (\mathrm{V}_{def}(B) \cap \mathrm{V}_{need}(R))$ und

dummy ein neuer, ansonsten im gesamten Programm nicht verwendeter Bezeichner ist;

sowie

$$\mathcal{TF}_K(\ \text{FOR}\ (\ A_1;\ e;\ A_2)\ \{\ A;\};\ R,\ \emptyset)$$
$$\mapsto\ \mathcal{TF}_K(\ A_1;\ \text{WHILE}(\ e)\ \{\ A;\ A_2;\};\ R,\ \emptyset) \quad .$$

Bei den DO-Schleifen kann die Menge der erforderlichen Parameter der tail-end-rekursiven Funktion enger gefaßt werden. Der Grund hierfür liegt darin, daß bei DO-Schleifen sichergestellt ist, daß sie mindestens einmal durchlaufen werden. Auf eine Ergänzung der „Parametervariablen“ um die „Ergebnisvariablen“ kann deshalb bei all den Variablen, die mit Sicherheit im Schleifenrumpf definiert werden, verzichtet werden. Dies führt zu der Transformationsregel

$$\mathcal{TF}_K(\ \text{DO}\ \{\ A;\}\ \text{WHILE}(\ e);\ R,\ \emptyset)$$

$$\mapsto \left\{\begin{array}{l}
\text{LETREC} \\
\quad dummy = \lambda x_1 \ldots x_n . \\
\qquad\qquad \mathcal{TF}_K \left(\begin{array}{l} A; \\ \text{IF}(\ e)\ \{ \\ \quad y_1, \ldots, y_m\ =\ dummy(\ x_1, \ldots, x_n), \emptyset \\ \} \\ \text{RETURN}(\ y_1, \ldots, y_m); \end{array}\right) \\
\text{IN LET} \\
\quad y_1, \ldots, y_m = dummy(\ x_1,\ \ldots,\ x_n) \\
\quad \mathcal{TF}_K(\ R,\ \{y_1, \ldots, y_m\})
\end{array}\right. ,$$

wobei $\{x_i \mid i \in \{1,\ldots,n\}\} = \mathrm{V}_{need}(A) \cup (\mathrm{V}_{pot\_def}(A) \cap \mathrm{V}_{need}(R))$,

$\{y_i \mid i \in \{1,\ldots,m\}\} = (\mathrm{V}_{def}(A) \cap \mathrm{V}_{need}(R))$ sowie

dummy ein neuer, ansonsten im gesamten Programm nicht verwendeter Bezeichner ist.

Beispiel:

```
{ n =6;
  fa=1;
  DO {
    fa=n*fa;
    n =n-1;
  } WHILE( n>0 );
  RETURN( fa );
}
```

$\Longrightarrow$

```
LET
  n =6
  fa=1
IN LETREC
     f=λn fa.LET
                fa=(n*fa)
                n =(n-1)
              IN IF ( n>0)
                 THEN f( n, fa)
                 ELSE fa
   IN LET
        fa=f( n, fa)
      IN fa
```

3.2 Das Typsystem von SAC

In diesem Abschnitt sollen die Grundlagen des Typsystems von SAC dargestellt werden. Dabei beschränken wir uns zunächst auf die atomaren Typen und deren Rolle für die Semantik des bisher dargestellten SAC-Kernes. Die für die Erweiterung von SAC um Arrays benötigten Ergänzungen des Typsystems sind in dem entsprechenden Abschnitt 3.3.2 gesondert beschrieben.

Da SAC weder Funktionen höherer Ordnung noch partielle Anwendungen unterstützt, kann das Typsystem von SAC im wesentlichen auf dem von C aufbauen.

Der wichtigste Unterschied zu C besteht darin, daß anstelle einer Typüberprüfung eine Typinferenz vorgenommen wird. Neben den in Abschnitt 4 genannten Vorteilen für eine Compiler-Implementierung erspart dies dem Programmierer die Notwendigkeit, Typen für alle Variablen deklarieren zu müssen. Um dem C-gewohnten Programmierer optionale Typdeklarationen zu ermöglichen, müssen trotz Typinferenz alle Variablen gleichen Namens innerhalb einer Funktion ausschließlich für Werte desselben Typs stehen.

Die Definition von benutzerdefinierten Typen ist in SAC genauso möglich wie in C. In der Verwendung von Datenstrukturen eines benutzerdefinierten Typs bzw. deren Deklaration unterscheiden sich jedoch die beiden Programmiersprachen.

In C sind der definierende und der definierte Typ vollständig zu einander kompatibel. Das bedeutet, daß eine Datenstruktur eines benutzerdefinierten Typs als Argument für eine Funktion verwendet werden kann, die nicht den benutzerdefinierten Typ selbst, sondern dessen definierenden Typ als Argument erwartet. Entsprechend ist eine Anwendung einer Funktion, die einen benutzerdefinierten Typen als Argument erwartet, auf eine Datenstruktur des definierenden Typs ebenfalls möglich. Durch diese impliziten Typanpassungen sind die benutzerdefinierten Typen in C eher als syntaktische Vereinfachungen denn als eigenständige Typen zu verstehen.

SAC erfordert einen etwas rigideren Umgang mit benutzerdefinierten Typen. Eine Konversion zwischen definierendem und definiertem Typ ist ausschließlich explizit möglich. In Anlehnung an C gibt es dafür in SAC ein sog. Cast-Konstrukt. Es entspricht dem Cast-Konstrukt von C, ist jedoch auf die Verwendung im Zusammenhang mit benutzerdefinierten Typen beschränkt. Eine Konversion zwischen verschiedenen Grund-Typen durch das Cast-Konstrukt, wie es in C möglich ist, erlaubt das Typsystem von SAC nicht.

Diese strikte Unterscheidung zwischen definiertem und definierendem Typ ist insbesondere deshalb von Bedeutung, weil SAC, ähnlich wie C++, eine explizite Überladung von Funktionen zuläßt; eine Funktion ist in SAC also nicht durch ihren Namen allein, sondern durch Namen sowie Anzahl und Typen der Parameter bestimmt.

Die syntaktischen Erweiterungen des in Abb. 3.2 beschriebenen Sprachkernes von SAC um die vom Typsystem benötigten Sprachkonstrukte sind in Abb. 3.3 dargestellt. Da SAC vornehmlich auf numerische Anwendungen ausgerichtet ist, kann auf einen Teil der atomaren Typen von C in SAC verzichtet werden (siehe $PrimType$ in Abb. 3.3). Sowohl Typdefinitionen als auch Variablendeklarationen entsprechen syntaktisch denen in C ($TypeDef$ und $VarDec$ in Abb. 3.3). Lediglich das Cast-Konstrukt wird um einen Doppelpunkt ergänzt, um den Unterschied zum Cast-Konstrukt in C anzudeuten. Die entsprechende Erweiterung der $Expr$-Regel findet sich ebenfalls in Abb. 3.3.

Um den Begriff eines typbaren SAC-Programmes und damit dann auch die Semantik eines SAC-Programmes (vergl. Abschnitt 3.1.2) einführen zu können, müssen wir zunächst das Typinferenzsystem von SAC formalisieren. Die Vewendung eines

$$
\begin{array}{lcl}
Expr & \Rightarrow & \ldots \\
 & | & (\ :\ Type\ )\ Expr \\
 & & \\
TypeDef & \Rightarrow & \text{TYPEDEF}\ \ Type\ Id\ \ ; \\
 & & \\
VarDec & \Rightarrow & Type\ Id\ \ ; \\
 & & \\
Type & \Rightarrow & PrimType \\
 & | & Id \\
 & & \\
PrimType & \Rightarrow & \text{INT} \\
 & | & \text{FLOAT} \mid \text{DOUBLE} \\
 & | & \text{BOOL} \mid \text{CHAR}
\end{array}
$$

Abbildung 3.3: Erweiterung der Syntax von SAC um Typen.

Deduktionssystems erlaubt dabei in ähnlicher Weise wie bei der Definition der Semantik von $\mathcal{F}$UN die Ableitung eines konkreten Typinferenz-Mechanismus.

Zunächst soll die Menge der in SAC spezifizierbaren Typen $\mathcal{T}_{\text{SAC}}$ formatiert werden. Sie besteht aus der Menge aller atomaren Typen $\mathcal{T}_{Simple}$ und der Menge aller benutzerdefinierbaren Typen $\mathcal{T}_{User}$. Um den formalen Umgang mit benutzerdefinierten Typen zu vereinfachen, wollen wir annehmen, daß für die Typinferenz benutzerdefinierte Typen nicht nur als Identifikatoren, sondern als Tupel der Form $< Id, \tau >$ vorliegen, wobei Id der Name und τ der definierende, atomare Typ des benutzerdefinierten Typs ist. Typdefinitionen bedürfen daher keiner weiteren Beachtung. Damit ergibt sich die

Definition 3.2.1 . *Es sei*

$$
\begin{array}{lll}
\mathcal{T}_{Simple} & := \{\text{INT, FLOAT, DOUBLE, BOOL, CHAR}\} & , \\
\mathcal{T}_{User} & := \{<Id,\ \tau> : Id \in Ids,\ \tau \in \mathcal{T}_{Simple}\} & , \\
\mathcal{T}_{\text{SAC}} & := \mathcal{T}_{Simple} \cup \mathcal{T}_{User} & .
\end{array}
$$

3.2.1□

Neben den in SAC spezifizierbaren Typen benötigen wir für die Typinferenz Typkonstruktoren zur Darstellung des Typs von Funktionen. SAC unterstützt nur vollständige Funktionsanwendungen; eine Curryfizierung von mehrstelligen Funktionen ist nicht möglich. Deshalb führen wir neben Funktionstypen $(\tau_1 \rightarrow \tau_2)$ auch Produkt-Typen $(\bigotimes_{i=1}^{n} \tau_i := \tau_1 \times \tau_2 \times \ldots \times \tau_n)$ ein. Für die primitive Ganzzahl-Addition ergibt sich damit zum Beispiel als Typ: INT $\times$ INT $\rightarrow$ INT. Eine formale Definition der ausschließlich für die Typinferenz benötigten Typen $\mathcal{T}_{Infer}$ liefert die

Definition 3.2.2 . *Es sei*

$$\begin{aligned}
\mathcal{T}_{Prod} &:= \{\bigotimes_{i=1}^{n} \tau_i : n \in \mathbb{N},\ \tau_i \in \mathcal{T}_{\text{SAC}}\} \quad , \\
\mathcal{T}_{Fun} &:= \{\tau_1 \rightarrow \tau_2 : \tau_1, \tau_2 \in \mathcal{T}_{\text{SAC}} \cup \mathcal{T}_{Prod}\} \quad , \\
\mathcal{T}_{Infer} &:= \mathcal{T}_{Prod} \cup \mathcal{T}_{Fun} \quad .
\end{aligned}$$

3.2.2□

Um das Typinferenzsystem über ein Deduktionssystem definieren zu können, betrachten wir Sequenzen der Form A$\vdash e:\tau$, wobei A eine Menge von Variable-Typ-Paaren ist, e ein SAC-Programm(-fragment) und τ der für e inferierbare Typ. Unter der Annahme, daß A=$\{(v_1:\tau_1), ..., (v_n:\tau_n)\}$, läßt sich der Ausdruck A$\vdash e:\tau$ dann folgendermaßen interpretieren:

Unter der Voraussetzung, daß für alle $i \in \{1, ..., n\}$ die Variablen v_i den Typ τ_i haben, ist e typbar und hat den Typ τ.

Aufbauend auf diesen Sequenzen werden im folgenden schrittweise die Regeln R_{Type} eines Deduktionssystems $\mathrm{D}_{Type} = (\mathrm{L}_{Type}, \mathrm{R}_{Type})$ entwickelt, mit dessen Hilfe sich dann die Typbarkeit eines SAC-Programmes S als Ableitbarkeit der Sequenz $\{\}\vdash S:\tau$ definieren läßt.

Mit der abkürzenden Notation

$$A\{v\ :\ \tau\} := \begin{cases} A \setminus \{(v,\sigma)\} \cup \{(v,\tau)\} & \text{falls } \exists\sigma : (v,\sigma) \in A \\ A \cup \{(v,\tau)\} & \text{sonst} \end{cases}$$

lassen sich die Regeln R_{Type} folgendermaßen formulieren:

Der Typ eines vollständigen Programmes ergibt sich aus dem Typ der main-Funktion; ist die MAIN-Funktion typbar mit dem Typ τ, so ist auch das gesamte Programm mit dem Typ τ typbar:

$$\textsc{Prg} \quad : \quad \frac{\{\}\vdash \tau\ \textsc{main}()\ Body:\tau}{\{\}\vdash FunDef_1, ..., FunDef_n\ \tau\ \textsc{main}()\ \{Body\}:\tau} \quad .$$

Der Typ einer parameterlosen Funktion entspricht dem seines Rumpfes:

$$\textsc{FunDef1} \quad : \quad \frac{\{\}\vdash Body:\tau}{\{\}\vdash \tau\ F()\ \{Body\}:\tau} \quad .$$

Alle anderen Funktionen bekommen einen Funktionstyp zugeordnet, der den Produkttyp der Argumente in den Typ des Rumpfes abbildet:

$$\textsc{FunDef2} \quad : \quad \frac{\{v_i:\tau_i\}\vdash Body:\tau}{\{\}\vdash \tau\ F(\tau_1\ v_1, ..., \tau_n\ v_n)\ \{Body\}:\bigotimes_{i=1}^{n}\tau_i \rightarrow \tau} \quad .$$

Der Typ von Konstanten kann direkt abgeleitet werden:

$$\textsc{Const} \;:\; \frac{}{\mathrm{A}\vdash Const : TYPE(Const)} \quad ,$$

wobei $TYPE$ jeder Konstanten ihren atomaren Typ zuordnet .

Einer Variablen läßt sich genau dann ein Typ zuordnen, wenn die Menge A bereits ein entsprechendes Variable-Typ-Paar enthält:

$$\textsc{Var} \;:\; \frac{}{\mathrm{A}\vdash v:\tau} \quad \Longleftrightarrow \quad (v:\tau)\in \mathrm{A} \quad .$$

Dadurch ist sichergestellt, daß nur solche Variablen benutzt werden, die entweder kein formaler Parameter sind, oder aber vorher durch eine Zuweisung definiert wurden.

Um die Typbarkeit eines Ausdruckes mit vorangestelltem Cast-Konstrukt formulieren zu können, wird eine Funktion eingeführt, die einen Typ auf seinen sog. Basistyp, d.h. den ihm zu Grunde liegenden atomaren Typ abbildet.

Definition 3.2.3 . *Sei $\tau \in \mathcal{T}_{\mathrm{SAC}}$. Dann ist der* Basistyp *von τ definiert durch $Basetype(\tau)$, wobei*

$$Basetype : \mathcal{T}_{\mathrm{SAC}} \rightarrow \mathcal{T}_{\mathrm{SAC}} \backslash \mathcal{T}_{User}$$

mit

$$Basetype(\tau) \mapsto \begin{cases} \sigma & falls\ \tau = <Id,\ \sigma> \\ \tau & sonst \end{cases} \quad .$$

3.2.3□

Damit läßt sich folgende Regel aufstellen: Ein Ausdruck mit Cast-Konstrukt ist genau dann typbar, wenn der Basistyp des Cast-Konstruktes mit dem Basistyp des nachfolgenden Ausdruckes übereinstimmt:

$$\textsc{Cast} \;:\; \frac{\mathrm{A}\vdash e:\sigma}{\mathrm{A}\vdash (:\tau)e:\tau} \quad \Longleftrightarrow \quad Basetype(\sigma) = Basetype(\tau) \quad .$$

Der Typ eines Funktionsrumpfes mit Variablendeklaration entspricht dem des Funktionsrumpfes ohne Variablendeklaration:

$$\textsc{VarDec} \;:\; \frac{\mathrm{A}\{v:\tau\}\vdash Rest:\sigma \quad \mathrm{A}\vdash Rest:\sigma}{\mathrm{A}\vdash \tau\ v;\ Rest:\sigma}$$

$$\Longleftrightarrow \neg\exists\ (v:\rho) \in \mathrm{A} \text{ mit } \rho \neq \tau \quad .$$

Die Forderung, daß A keine andere Bindung für die Variable enthalten darf, stellt sicher, daß es keine widersprüchlichen Typdeklarationen gibt.

Der Typ der Return-Anweisung liefert schließlich den Typ des gesamten Funktionsrumpfes; er ist der Produkttyp der Return-Werte:

$$\textsc{Return} \;:\; \frac{\mathrm{A}\vdash e_i:\tau_i}{\mathrm{A}\vdash \textsc{return}(e_1,\,...,\,e_n):\bigotimes_{i=1}^{n}\tau_i} \quad .$$

Der Typ eines mit einer Zuweisung v = e; beginnenden Funktionsrumpfes entspricht dem Typ des Restrumpfes nach der Zuweisung, unter der Annahme, daß die Variable v den Typ des Ausdruckes e hat:

$$\textsc{Let} \;:\; \frac{\mathrm{A}\vdash e:\bigotimes_{i=1}^{n}\tau_i \quad \mathrm{A}\{v_i:\tau_i\}\vdash \mathit{Rest}:\tau}{\mathrm{A}\vdash v_1,\,...,\,v_n = e;\,\mathit{Rest}:\tau}$$

$$\Longleftrightarrow \forall i\in\{1,...,n\}:\left(\exists\,(v_i:\sigma_i)\in\mathrm{A}\Rightarrow(\tau_i=\sigma_i)\right) \quad .$$

Die Forderung, daß es kein Variablen-Typ-Paar in A mit einem anderen Typ geben darf, stellt sicher, daß alle gleichnamigen Variablen in einem Funktionsrumpf denselben Typ haben.

Sowohl bei benutzerdefinierten Funktionsanwendungen als auch bei primitiven Funktionsanwendungen muß der Typ der Argumente im Definitionsbereich der jeweiligen Funktion liegen. Die Verwendung der Produkttypnotation garantiert dabei, daß es sich jeweils um vollständige Anwendungen handelt. Der Typ der Anwendung ergibt sich dann aus dem Resultatstyp der jeweiligen Funktion:

$$\textsc{FunAp} \;:\; \frac{\mathrm{A}\vdash e_i:\tau_i \quad \{\}\vdash \tau\ F(\tau_1\ v_1,\,...,\,\tau_n\ v_n)\ \{\mathit{Body}\}:\bigotimes_{i=1}^{n}\tau_i\to\tau}{\mathrm{A}\vdash F(e_1...e_n):\tau} \quad ,$$

$$\textsc{PrfAp} \;:\; \frac{\mathrm{A}\vdash e_i:\tau_i}{\mathrm{A}\vdash F(e_1,\,...,\,e_n):\tau} \quad\Longleftrightarrow\quad \mathit{TYPE}(\,F) = \bigotimes_{i=1}^{n}\tau_i\to\tau \quad ,$$

wobei TYPE jeder primitiven Funktion ihren Typ zuordnet .

Der Typ eines mit einem IF-THEN-ELSE-Konstrukt beginnenden Funktionsrumpfes entspricht dem Typ der fiktiven Funktionsrümpfe, die aus dem *AssignBlock* der jeweiligen Alternative und dem Rest des Rumpfes bestehen:

$$\textsc{Cond1} \;:\; \frac{\mathrm{A}\vdash e:\textsc{bool} \quad \mathrm{A}\vdash \mathit{Ass}_t;\,\mathit{Rest}:\tau \quad \mathrm{A}\vdash \mathit{Rest}:\tau}{\mathrm{A}\vdash \textsc{if}\ (e)\ \{\mathit{Ass}_t;\}\ \mathit{Rest}:\tau} \quad ,$$

$$\textsc{Cond2} \;:\; \frac{\mathrm{A}\vdash e:\textsc{bool} \quad \mathrm{A}\vdash \mathit{Ass}_t;\,\mathit{Rest}:\tau \quad \mathrm{A}\vdash \mathit{Ass}_e;\,\mathit{Rest}:\tau}{\mathrm{A}\vdash \textsc{if}\ (e)\ \{\mathit{Ass}_t;\}\ \textsc{else}\ \{\mathit{Ass}_e;\};\,\mathit{Rest}:\tau} \quad .$$

In ähnlicher Weise ergeben sich die Typen von Funktionsrümpfen, die mit einem Schleifenkonstrukt beginnen:

$$\text{DO} \quad : \quad \frac{\text{A}\vdash e:\text{BOOL} \quad \text{A}\vdash Ass;\, Rest:\tau}{\text{A}\vdash \text{DO}\ \{Ass;\}\ \text{WHILE}\ (e);\, Rest:\tau} \quad ,$$

$$\text{WHILE} \quad : \quad \frac{\text{A}\vdash e:\text{BOOL} \quad \text{A}\vdash Rest:\tau \quad \text{A}\vdash Ass;\, Rest:\tau}{\text{A}\vdash \text{WHILE}\ (e)\ \{Ass;\};\, Rest:\tau} \quad ,$$

$$\text{FOR} \quad : \quad \frac{\text{A}\vdash e:\text{BOOL} \quad \text{A}\vdash Ass_1;\, Rest:\tau \quad \text{A}\vdash Ass_1;\, Ass_3;\, Ass_2;\, Rest:\tau}{\text{A}\vdash \text{FOR}\ (Ass_1;\, e;\, Ass_2)\ \{Ass_3;\};\, Rest:\tau} \quad .$$

Eine Zusammenfassung aller dieser Regeln ergiebt die vollständige Spezifikation des Deduktionssystems $D_{Type} = (L_{Type},\, R_{Type})$ mit

$$\begin{aligned} L_{Type} \quad &= \{\text{A}\vdash e:\tau \mid \\ &\quad \text{A}\subset \{(v:\sigma) \mid v \in Ids,\ \sigma \in \mathcal{T}_{\text{SAC}} \cup \mathcal{T}_{Infer}\},\ e \in \text{SAC},\ \tau \in \mathcal{T}_{\text{SAC}} \cup \mathcal{T}_{Infer}\} \end{aligned}$$

und

$$\begin{aligned} R_{Type} \quad &= \{\text{PRG}, \text{FUNDEF1}, \text{FUNDEF2}, \text{CONST}, \text{VAR}, \text{CAST}, \text{VARDEC}, \text{RETURN}, \\ &\quad \text{LET}, \text{FUNAP}, \text{PRFAP}, \text{COND1}, \text{COND2}, \text{DO}, \text{WHILE}, \text{FOR}\} \quad . \end{aligned}$$

Daraus ergibt sich die

Definition 3.2.4 . *Sei D_{Type} das oben eingeführte Deduktionssystem. Dann heißt ein* SAC*-Programm S* typbar *mit dem Typ τ, gdw. es einen Typ $\tau \in \mathcal{T}_{\text{SAC}}$ gibt, so daß gilt:*

$$\{\} \vdash_{D_{Type}} \{\}\vdash S:\tau \quad .$$

3.2.4□

Mit Hilfe dieses Typbarkeitsbegriffes läßt sich schließlich die Bedeutungsfunktion für SAC-Programme definieren.

Definition 3.2.5 . *Sei Const die Menge der Konstanten in* SAC*. Dann wird die Bedeutung eines Programmes $S \in$ SAC definiert durch*

$$m_{\text{SAC}} : \text{SAC} \longrightarrow_{part} Const$$

$$\begin{aligned} \textit{mit } m_{\text{SAC}}(S) = S' :\Longleftrightarrow\ & S \textit{ ist typbar gem. Def.3.2.4} \\ & \wedge\ m_{\mathcal{FUN}}(\mathcal{TF}_K(S), \emptyset) = S' \quad . \end{aligned}$$

3.2.5□

3.3 Dimensionsunabhängiges Programmieren in SAC

Wie bereits erwähnt, ist das Array-Konzept von SAC so ausgelegt, daß es dem Programmierer ein möglichst hohes Abstraktionsniveau bietet. Zentrale Voraussetzung dafür ist die Möglichkeit, uniform mit Arrays verschiedener Dimensionalität umgehen zu können. Um dies zu erreichen, wird in SAC integriert

- eine Array-Darstellung, bei der n-dimensionale Arrays äquivalent zu einem Vektor von (n-1)-dimensionalen Arrays sind. Dies erlaubt eine konsistente Erweiterung der Anwendbarkeit primitiver Array-Operationen, die Arrays einer festen Dimensionalität als Argument(e) benötigen, auf Arrays höherer Dimensionalität;
- primitive Array-Operationen, die unabhängig von der Dimensionalität elementweise auf dem (den) Argument(en) operiert;
- Array-Comprehension-Konstrukte, die eine elementweise Spezifikation von Arrays unabhängig von der Dimensionalität des zu erzeugenden Arrays zulassen

Als Grundlage für die Array-Darstellung sowie für die primitiven Array-Operationen in SAC dient der von L. Mullin entwickelte Ψ-Kalkül [Mul88, Mul91, MJ91]. Dabei handelt es sich um einen Array-Kalkül, der neben einer Menge von dimensionsunabhängigen Array-Operationen einen Vereinfachungsmechanismus für deren Komposition definiert. Dies garantiert nicht nur eine mathematische Konsistenz der verschiedenen Operationen, sondern bietet auch Ansatzpunkte für Compiler-Optimierungen. Aufbauend auf der Array-Darstellung werden verschiedene Array-Comprehension-Konstrukte entworfen, die ebenfalls eine Anwendung auf Arrays verschiedener Dimensionalität zulassen.

Neben der Einführung verschiedener Array-Sprachkonstrukte wird das in Abschnitt 3.2 vorgestellte Typsystem von SAC um spezielle Array-Typen sowie die dafür benötigten Typinferenzregeln erweitert. Sie erlauben für die meisten Funktionsanwendungen auf n-dimensionale Arrays die Inferenz eines Array-Typs, der die genaue Form des Arrays enthält. Auf diese Weise kann trotz des hohen Abstraktionsgrades eine Compilation in effizient ausführbaren Code erreicht werden.

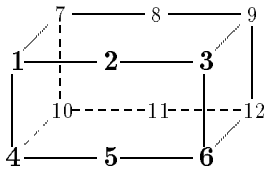
3.3.1 Die Darstellung von Arrays in SAC

Wie in den array-orientierten Sprachen APL und NIAL (vergl. Abschnitt 2.1) oder dem Ψ-Kalkül werden in SAC Arrays durch zwei Vektoren, den Shape-Vektor und den Datenvektor, dargestellt. Dabei gibt der Shape-Vektor die Form des Arrays und der Datenvektor die Elemente des Arrays an. Beispiele dazu sind in Abb. 3.4.

Auf der linken Seite befinden sich graphische Darstellungen je eines ein-, zwei- und drei-dimensionalen Arrays und auf der rechten Seite die jeweiligen Beschreibungen durch Shape- und Datenvektoren. Das zweidimensionale Array A_2 z.B. wird in

$A_1 = \langle\ 1\ 2\ 3\ \rangle$ Shape-Vektor: [3]
Datenvektor: [1, 2, 3]

$A_2 = \begin{pmatrix} 1\ 2\ 3 \\ 4\ 5\ 6 \end{pmatrix}$ Shape-Vektor: [2, 3]
Datenvektor: [1, 2, 3, 4, 5, 6]

$A_3 =$ 

Shape-Vektor: [2, 2, 3]
Datenvektor: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

Abbildung 3.4: Array-Darstellung mittels Shape- und Datenvektor.

SAC durch den Shape-Vektor [2, 3] und den Datenvektor [1, 2, 3, 4, 5, 6] beschrieben. Aus dem Shape-Vektor lassen sich wie in Abschnitt 2.1 beschrieben direkt die zulässigen Zugriffs-Indizes i mit $[0, 0] \leq i \leq [1, 2]$ ableiten. Dabei referenziert [0, 0] den Wert 1, [0, 1] den Wert 2, [1, 0] den Wert 4, usw.

In gleicher Weise lassen sich Arrays höherer Dimensionalität spezifizieren. Betrachten wir z.B. das dreidimensionale Array A_3 aus Abb. 3.4, so läßt sich dies durch den Shape-Vektor [2, 2, 3] und den Datenvektor [1, 2, ..., 11, 12] definieren; der Index [0, 1, 2] selektiert hier den Wert 6. Die Darstellung von Arrays läßt sich formalisieren durch die

Definition 3.3.1 . *Ein* n-dimensionales Array *A wird definiert durch einen* Shape-Vektor *$[s_0, ..., s_{(n-1)}]$ und einen* Datenvektor *$[d_0, ..., d_{(m-1)}]$, wobei*

$$m = \prod_{i=0}^{(n-1)} s_i \quad \textit{gilt.} \tag{3.1}$$

Die Menge der legalen Indizes *in A ist definiert durch:*

$$\mathcal{I}(A) := \{[i_0, ..., i_{(n-1)}] \mid \forall j \in \{0, ..., n-1\} : 0 \leq i_j < s_j\} \quad . \tag{3.2}$$

Ein Index $[i_0, ..., i_{(n-1)}]$ korrespondiert *mit dem Element d_k, gdw. $[i_0, ..., i_{(n-1)}]$ ein legaler Index in A mit*

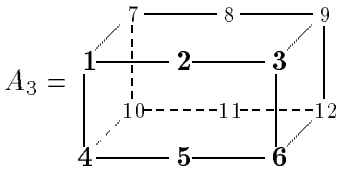
$$k = \sum_{i=0}^{(n-1)} \left(i_i * \prod_{j=i+1}^{(n-1)} s_j \right) \qquad \textit{ist.} \tag{3.3}$$

3.3.1□

Diese Darstellung von Arrays gestattet auf natürliche Weise eine Äquivalenz zwischen Schachtelungen von (mehrdimensionalen) Arrays und mehrdimensionalen Arrays selbst. Betrachten wir A_2 aus Abb. 3.4, so kann folgende Äquivalenz formuliert werden:

$$A_2 = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} :\Longleftrightarrow \left\langle \left\langle 1\ 2\ 3 \right\rangle \left\langle 4\ 5\ 6 \right\rangle \right\rangle \quad .$$

In gleicher Weise gilt für A_3 aus Abb. 3.4:

$A_3 =$ 

$$:\Longleftrightarrow \left\langle \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} \right\rangle$$

$$:\Longleftrightarrow \left\langle \left\langle \left\langle 1\ 2\ 3 \right\rangle \left\langle 4\ 5\ 6 \right\rangle \right\rangle \left\langle \left\langle 7\ 8\ 9 \right\rangle \left\langle 10\ 11\ 12 \right\rangle \right\rangle \right\rangle$$

$$:\Longleftrightarrow \begin{pmatrix} \left\langle 1\ 2\ 3 \right\rangle & \left\langle 4\ 5\ 6 \right\rangle \\ \left\langle 7\ 8\ 9 \right\rangle & \left\langle 10\ 11\ 12 \right\rangle \end{pmatrix} \quad .$$

Diese Äquivalenzen lassen sich formalisieren durch die

Definition 3.3.2 . *Sei A ein n-dimensionales Array mit einem Shape-Vektor $[s_0, ..., s_{(n-1)}]$ und Datenvektor $[A_0, ..., A_{(m-1)}]$. Weiterhin sei für alle $i \in \{0, ..., m-1\}$ A_i ein n'-dimensionales Array mit einem Shape-Vektor $[s'_0, ..., s'_{(n'-1)}]$ und Datenvektor $[d_{i1}, ..., d_{i(m'-1)}]$. Dann ist A äquivalent zu einem $(n+n')$-dimensionalem Array mit dem Shape-Vektor $[s_0, ..., s_{(n-1)}, s'_0, ..., s'_{(n'-1)}]$ und dem Datenvektor $[d_{00}, ..., d_{0(m'-1)}, ..., d_{(m-1)0}, ..., d_{(m-1)(m'-1)}]$.*

3.3.2□

Aus dieser Äquivalenz wird deutlich, daß es selbst für die Darstellung geschachtelter Arrays vollkommen ausreicht, einen Datenvektor und einen Shape-Vektor zu spezifizieren. Syntaktisch bietet SAC dafür zum einen die Möglichkeit, Vektoren durch eine Liste von Elementen in eckigen Klammern darzustellen, und zum anderen die zweistellige primitive Funktion RESHAPE; sie ordnet einem Datenvektor (zweites Argument) einen Shape-Vektor (erstes Argument) zu, falls Gleichung (3.1) von Def. 3.3.1 erfüllt ist. Damit läßt sich A_3 aus Abb. 3.4 schreiben als

```
{ ...
  A = RESHAPE([2, 2, 3], [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]);
  ...
}
```

Da Vektoren nichts anderes als eindimensionale Arrays sind, hat jeder in SAC spezifizierte Vektor einen impliziten Shape-Vektor der Form $[n]$, wobei n der Anzahl der Elemente entspricht. RESHAPE dient daher nicht nur dazu, einem mehrdimensionalen Array initial einen Shape-Vektor zuzuweisen, sondern kann auch dazu genutzt werden, die Form eines Arrays zu verändern, solange Gleichung (3.1) aus Def. 3.3.1 gilt.

Eine andere Möglichkeit, mehrdimensionale Arrays in SAC zu spezifizieren, ergibt sich aus dem in Def. 3.3.2 festgelegten Zusammenhang zwischen geschachtelten und mehrdimensionalen Arrays. A_3 aus Abb. 3.4 kann somit auch erzeugt werden durch

```
{ ...
  A = [[[1, 2, 3],[4, 5, 6]],[[7, 8, 9],[10, 11, 12]]];
  ...
}                                                        .
```

Aufgrund der Unübersichtlichkeit dieser Schreibweise, ist ihre Verwendung jedoch fragwürdig.

Eine dritte Möglichkeit zur direkten Spezifikation von mehrdimensionalen Arrays in SAC bietet das Typsystem. Es unterstützt einen eigenen Typ für jede Kombination von atomarem Typ und Shape-Vektor. Durch vorherige Deklaration als ein Array mit dem gewünschten Shape-Vektor kann eine explizite Zuordnung eines Shape-Vektors entfallen. Für das Beispiel-Array A_3 ergibt sich

```
{ INT[2, 2, 3] A;


  ...
  A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12];
  ...
}                                                .
```

Die für die Integration von Arrays benötigten syntaktischen Erweiterungen des in den vorherigen Abschnitten beschriebenen SAC-Kernes sind in Abb. 3.5 dargestellt. Dabei fällt auf, daß neben den bereits beschriebenen Typen mit festem Shape-Vektor auch Typen der Art $PrimType[\overbrace{\bullet, ..., \bullet}^{m}, s_1, ..., s_n]$ mit $m > 0$ und $n \geq 0$ sowie $PrimType[\,]$ zulässig sind. Es handelt sich hierbei um Supertypen der bisher vorgestellten Array-Typen, die benötigt werden, um Argumenttypen unabhängig vom Shape-Vektor beziehungsweise unabhängig von der Dimensionalität des Argument-Arrays spezifizieren zu können. INT[•, •, •] bezeichnet z.B. ein dreidimensionales Array von Integer-Zahlen, während INT[ ] ein Array unbekannter Dimensionalität von Integer-Zahlen sein kann. Daraus ergibt sich die in Abb. 3.6 angedeutete Hierarchie von Array-Typen. Der Typ τ steht dabei exemplarisch für einen atomaren Typ in SAC; die Verbindung zweier Array-Typen durch eine Linie ist als transitive

$$
\begin{array}{lcl}
Expr & \Rightarrow & \ldots \\
 & | & [\ Expr\ \lceil\ ,\ Expr\ \rfloor^{*}\ ] \\
 & | & \text{RESHAPE}\ (\ Expr\ ,\ Expr\ ) \\
 & & \\
Type & \Rightarrow & \ldots \\
 & | & PrimType\ [\ Int\ \lceil\ ,\ Int\ \rfloor^{*}\ ] \\
 & | & PrimType\ [\ \bullet\ \lceil\ ,\ \bullet\ \rfloor^{*}\ \lceil\ ,\ Int\ \rfloor^{*}\ ] \\
 & | & PrimType\ [\ ] \\
 & | & Id\ [\ Int\ \lceil\ ,\ Int\ \rfloor^{*}\ ] \\
 & | & Id\ [\ \bullet\ \lceil\ ,\ \bullet\ \rfloor^{*}\ \lceil\ ,\ Int\ \rfloor^{*}\ ] \\
 & | & Id\ [\ ]
\end{array}
$$

Abbildung 3.5: Array-Syntax in SAC.

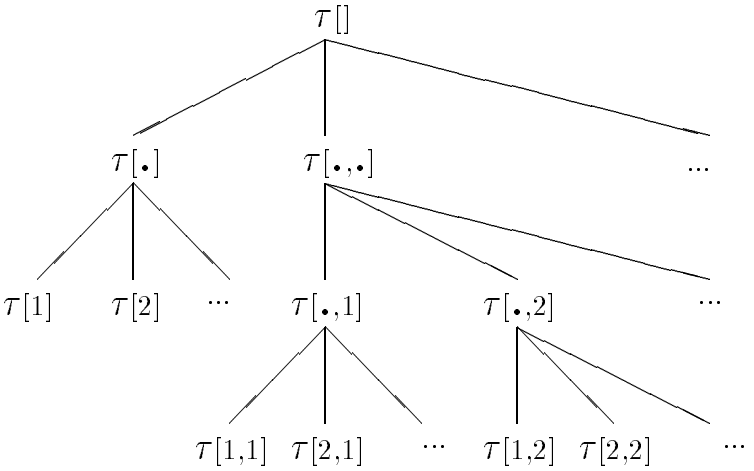


Abbildung 3.6: Hierarchie von Array-Typen in SAC.

Subtyprelation zu verstehen. Der jeweils weiter unten abgebildete Typ ist der Subtyp des höher gelegenen. Insgesamt stellt die Hierarchie einen unendlichen Baum dar, dessen Wurzel der alle Arrays des Basistypes τ umfassende Typ $\tau[\,]$ ist, und dessen Blätter die einzelnen Array-Typen mit verschiedenem festem Shape-Vektor sind.

3.3.2 Typinferenz auf n-dimensionalen Arrays

Aufgrund der im letzten Abschnitt beschriebenen Array-Typen bedarf die der Menge $\mathcal{T}_{\text{SAC}}$ aller in SAC zulässigen Typen aus Def. 3.2.1 einer Ergänzung. Während die

atomaren Typen $\mathcal{T}_{Simple}$ unverändert bleiben, werden die benutzerdefinierten Typen $\mathcal{T}_{User}$ um Array-Typen mit festem Shape-Vektor als definierendem Typ erweitert. Die Menge der Array-Typen $\mathcal{T}_{Array}$ ergibt sich zunächst aus der Menge der Array-Typen unbekannter Shape-Vektoren bzw. unbekannter Dimensionalität $\mathcal{T}_{Dim}$ und der Array-Typen mit festem Shape-Vektor. Diese wiederum werden unterteilt in solche, die auf atomaren Typen basieren $\mathcal{T}_{SimpleArray}$ und solche, die auf benutzerdefinierten Typen basieren $\mathcal{T}_{UserArray}$.

Definition 3.3.3 . *Es sei*

$$\begin{aligned}\mathcal{T}_{Simple} &:= \{\text{INT, FLOAT, DOUBLE, BOOL, CHAR}\},\\ \mathcal{T}_{User} &:= \{<Id,\ \tau> : \tau \in \mathcal{T}_{Simple} \cup \mathcal{T}_{SimpleArray}\},\\ \mathcal{T}_{Basic} &:= \mathcal{T}_{Simple} \cup \mathcal{T}_{User}\ .\end{aligned}$$

Weiterhin sei

$$\begin{aligned}\mathcal{T}_{SimpleArray} &:= \{\tau[s_1,...,s_n] : n \in \mathbb{N}, s_i \in \mathbb{N}, \tau \in \mathcal{T}_{Simple}\},\\ \mathcal{T}_{UserArray} &:= \{\tau[s_1,...,s_n] : n \in \mathbb{N}, s_i \in \mathbb{N}, \tau \in \mathcal{T}_{User}\},\\ \mathcal{T}_{Dim} &:= \{\tau[\overbrace{\bullet,...,\bullet}^{m}, s_1,...,s_n] : m \in \mathbb{N}, n \in \mathbb{N}_0, s_i \in \mathbb{N}, \tau \in \mathcal{T}_{Basic}\}\\ &\quad \cup \{\ \tau[\,] : \tau \in \mathcal{T}_{Basic}\},\\ \mathcal{T}_{Array} &:= \mathcal{T}_{SimpleArray} \cup \mathcal{T}_{UserArray} \cup \mathcal{T}_{Dim}\end{aligned}$$

und schließlich

$$\mathcal{T}_{\text{SAC}} := \mathcal{T}_{Basic} \cup \mathcal{T}_{Array}\quad .$$

3.3.3□

Die Erweiterungen in bezug auf die benutzerdefinierten Typen aus $\mathcal{T}_{User}$ bzw. $\mathcal{T}_{UserArray}$ erfordern darüber hinaus eine Erweiterung der Funktion *Basetype*, um eine Typinferenz für Cast-Ausdrücke mit allen benutzerdefinierten Typen zu ermöglichen (vergl. CAST-Regel aus Abschnit 3.2).

Definition 3.3.4 . *Sei $\tau \in \mathcal{T}_{\text{SAC}}$. Dann ist der* Basistyp *von τ definiert durch Basetype(τ), wobei*

$$Basetype : \mathcal{T}_{\text{SAC}} \to \mathcal{T}_{\text{SAC}} \backslash (\mathcal{T}_{User} \cup \mathcal{T}_{UserArray})$$

mit

$$Basetype(\tau) \mapsto \begin{cases} \sigma & falls\ \tau = <Id,\ \sigma> \\ \sigma[r_1,...,r_m] & falls\ \begin{array}{l}\tau = <Id,\ \sigma>[r_1,...,r_m]\\ \wedge\ \sigma \in \mathcal{T}_{Simple}\ \wedge\ r_i \in \mathbb{N}_{\bullet}\end{array} \\ \sigma[r_1,...,r_m,s_1,...,s_n] & falls\ \begin{array}{l}\tau = <Id,\ \sigma[s_1,...,s_n]>[r_1,...,r_m]\\ \wedge\ s_i \in \mathbb{N}\ \wedge\ r_i \in \mathbb{N}_{\bullet}\end{array} \\ \sigma[\,] & falls\ \tau = <Id,\ \sigma>[\,] \wedge \sigma \in \mathcal{T}_{Simple} \\ \sigma[\,] & falls\ \tau = <Id,\ \sigma[s_1,...,s_n]>[\,] \\ \tau & sonst \end{cases}$$

.

3.3.4□

Die in Abb. 3.6 dargestellte Hierarchie von Array-Typen kann formal durch eine Subtyprelation $\preceq$ beschrieben werden.

Definition 3.3.5 . *Sei* $\mathbb{N}_\bullet := (\mathbb{N} \cup \{\bullet\})$. *Um die natürlichen Zahlen mit dem* $\bullet$*-Symbol vergleichen zu können, wird eine Relation* $\leq_t$ *zwischen zwei Elementen aus* $\mathbb{N}_\bullet$ *eingeführt:*

$$\leq_t \subset \mathbb{N}_\bullet \times \mathbb{N}_\bullet$$

mit

$$x \leq_t y := \{(x,y) \mid x \in \mathbb{N}_\bullet, y = \bullet\} \quad .$$

Damit kann die Subtyprelation für Typen $\tau \in \mathcal{T}_{\text{SAC}}$ *definiert werden als*

$$\preceq \ \subset \mathcal{T}_{\text{SAC}} \times \mathcal{T}_{\text{SAC}}$$

mit

$$\begin{aligned}\tau_1 \preceq \tau_2 := \ & \{(\tau_1,\tau_2) \mid \tau_1 = \tau[s_1,...,s_n], n \in \mathbb{N}, s_i \in \mathbb{N}_\bullet, \tau_2 = \tau[\,]\} \\ & \cup\{(\tau_1,\tau_2) \mid \tau_1 = \tau_2\} \\ & \cup\{(\tau_1,\tau_2) \mid \tau_1 = \tau[r_1,...,r_n], \tau_2 = \tau[s_1,...,s_n], \\ & \qquad n \in \mathbb{N}, r_i \in \mathbb{N}_\bullet, s_i \in \mathbb{N}_\bullet, \forall i \in \{1,...,n\} : r_i \leq_t s_i\} \quad .\end{aligned}$$

Zwei Typen τ_1 *und* τ_2 *heißen* kompatibel *(*$\tau_1 \sim \tau_2$*), gdw.* $\tau_1 \preceq \tau_2$ *oder* $\tau_2 \preceq \tau_1$. *Von zwei kompatiblen Typen* τ_1 *und* τ_2 *heißt* τ_1 spezieller *(*$min(\tau_1, \tau_2) = \tau_1$*), gdw.* $\tau_1 \preceq \tau_2$.

3.3.5□

Durch die Einführung von Subtypen muß das in Abschnitt 3.2 definierte Basis-Regelwerk des Typsystems von SAC an all den Stellen erweitert werden, wo anstelle von Typgleichheit nur noch Typkompatibilität gemäß obiger Definition gefordert werden muß. Dies sind im wesentlichen die Regeln für Funktionsdeklarationen (PRG, FUNDEF1, FUNDEF2) und Variablendeklarationen (VARDEC). Zusätzlich ist auch die Regel LET davon betroffen, da sie die Typ-Gleichheit gleichnamiger Variablen sicherstellt. Diese durch die Existenz von Variablendeklarationen motivierte Restriktion wird im Kontext von Arrays dahingehend verschärft, daß alle gleichnamigen Variablen eines Array-Typs innerhalb eines Funktionsrumpfes denselben Shape-Vektor haben müssen. Insgesamt ergeben sich die folgenden neuen Regeln als Substitute für die gleichnamigen Regeln aus Abschnitt 3.2. Eine vollständige Darstellung aller Typinferenzregeln findet sich im Anhang B.

Bei der Inferenz des Resultattyps benutzerdefinierter Funktionen muß der inferierte Typ nicht dem angegebenen Typ entsprechen, sondern darf auch spezieller sein:

$$\text{PRG} \;:\; \frac{\{\}\vdash \tau \text{ MAIN}()\ Body:\sigma}{\{\}\vdash FunDef_1,\ ...,\ FunDef_n\ \tau \text{ MAIN}()\ \{Body\}:\sigma}$$
$$\Longleftrightarrow \sigma \preceq \tau \quad ,$$

$$\text{FUNDEF1} \;:\; \frac{\{\}\vdash Body:\sigma}{\{\}\vdash \tau\ F()\ \{Body\}:\sigma}$$
$$\Longleftrightarrow \sigma \preceq \tau \quad ,$$

$$\text{FUNDEF2} \;:\; \frac{\{v_i:\tau_i\}\vdash \text{Body}:\sigma}{\{\}\vdash \tau\ F(\tau_1\ v_1,\ ...,\ \tau_n\ v_n)\ \{\text{Body}\}:\bigotimes_{i=1}^{n}\tau_i \to \sigma}$$
$$\Longleftrightarrow \sigma \preceq \tau \quad .$$

In gleicher Weise dürfen auch die Argumente einer Funktionsanwendung spezieller als der Parametertyp der Funktion sein:

$$\text{FUNAP} \;:\; \frac{\text{A}\vdash e_i:\sigma_i \quad \{\}\vdash \tau F(\tau_1\ v_1,\ ...,\ \tau_n\ v_n)\ \{\text{Body}\}:\bigotimes_{i=1}^{n}\tau_i \to \sigma}{\text{A}\vdash F(e_1...e_n):\sigma}$$
$$\Longleftrightarrow (\forall i \in \{1,...,n\} : \sigma_i \preceq \tau_i) \wedge (\sigma \preceq \tau) \quad .$$

Die LET-Regel entspricht bis auf eine Umformulierung der Restriktion für namensgleiche Variablen der aus Abschnitt 3.2:

$$\text{LET} \;:\; \frac{\text{A}\vdash e:\bigotimes_{i=1}^{n}\sigma_i \quad \text{A}\{v_i:\tau_i\}\vdash R:\tau}{\text{A}\vdash v_1,\ ...,\ v_n = e;\ R:\tau}$$
$$\Longleftrightarrow \forall i \in \{1,...,n\} :$$
$$(\exists\ (\text{v}_i:\rho_i) \in \text{A} \Rightarrow \rho_i \sim \sigma_i \wedge \tau_i = \min(\ \rho_i,\ \sigma_i)$$
$$(\neg\exists\ (\text{v}_i:\rho_i) \in \text{A} \Rightarrow \tau_i = \sigma_i) \quad .$$

Besonders zu beachten sind bei dieser Regel die Fälle, in denen es in A Variable-Typ-Paare $(\text{v}_i:\rho_i)$ gibt, die einen spezielleren Typ ρ_i haben, als die für den Ausdruck e inferierten (Teil-)Typen σ_i; d.h. $\exists\ (\text{v}_i:\rho_i) \in \text{A}$ mit $\rho_i \preceq \sigma_i$. Da die oben angesprochene Restriktion gleichnamiger Variablen für die σ_i Gleichheit mit den ρ_i fordert, diese jedoch im Falle $\rho_i \neq \sigma_i$ nicht durch das Typsystem sichergestellt werden kann, müssen diese Fälle bei einer Compiler-Implementierung besonders beachtet werden und ggf. zu Laufzeitfehlern führen.

Alternativ diese Fälle durch das Typsystem auszuschließen, ist zwar vielleicht vom konzeptuellen Standpunkt aus zu bevorzugen, für praktische Belange jedoch unakzeptabel, da die Menge der typbaren Programme dadurch zu sehr eingeschränkt

wird.

Auch die Regel für Variablendeklarationen entspricht im wesentlichen der aus Abschnitt 3.2. Der einzige Unterschied besteht darin, daß eine Inferenz ohne Variablendeklaration nicht der Inferenz mit Variablendeklaration entsprechen, sondern nur zu ihr kompatibel sein muß:

$$\text{VARDEC} \;:\; \frac{\mathrm{A}\{v:\tau\}\vdash \mathit{Rest}:\sigma \quad \mathrm{A}\vdash \mathit{Rest}:\sigma'}{\mathrm{A}\vdash \tau\ v;\ \mathit{Rest}:\sigma''}$$

$$\Longleftrightarrow \neg\exists\ (v:\rho)\in \mathrm{A} \text{ mit } \rho\neq\tau \\ \wedge\ (\sigma\sim\sigma')\wedge(\sigma''=\min(\ \sigma,\ \sigma'))\qquad .$$

Im Falle der Ungleichheit der dabei inferierten Typen σ und σ' kann der speziellere Typ als Gesamttyp verwendet werden. Der Grund dafür liegt ebenfalls in der Typ-Restriktion gleichnamiger Variablen.

In gleicher Weise können die Regeln für IF-THEN-ELSE-Konstrukte und Schleifen erweitert werden:

$$\text{COND1} \;:\; \frac{\mathrm{A}\vdash e:\text{BOOL} \quad \mathrm{A}\vdash \mathit{Ass}_t;\ \mathit{Rest}:\tau \quad \mathrm{A}\vdash \mathit{Rest}:\tau'}{\mathrm{A}\vdash \text{IF } (e)\ \{\mathit{Ass}_t;\}\ \mathit{Rest}:\tau''}$$

$$\Longleftrightarrow (\tau\sim\tau')\wedge(\tau''=\min(\ \tau,\ \tau'))\qquad ,$$

$$\text{COND2} \;:\; \frac{\mathrm{A}\vdash e:\text{BOOL} \quad \mathrm{A}\vdash \mathit{Ass}_t;\ \mathit{Rest}:\tau \quad \mathrm{A}\vdash \mathit{Ass}_e;\ \mathit{Rest}:\tau'}{\mathrm{A}\vdash \text{IF } (e)\ \{\mathit{Ass}_t;\}\ \text{ELSE } \{\mathit{Ass}_e;\};\ \mathit{Rest}:\tau''}$$

$$\Longleftrightarrow (\tau\sim\tau')\wedge(\tau''=\min(\ \tau,\ \tau'))\qquad ,$$

$$\text{WHILE} \;:\; \frac{\mathrm{A}\vdash e:\text{BOOL} \quad \mathrm{A}\vdash \mathit{Rest}:\tau \quad \mathrm{A}\vdash \mathit{Ass};\ \mathit{Rest}:\tau'}{\mathrm{A}\vdash \text{WHILE } (e)\ \{\mathit{Ass};\};\ \mathit{Rest}:\tau''}$$

$$\Longleftrightarrow (\tau\sim\tau')\wedge(\tau''=\min(\ \tau,\ \tau'))\qquad ,$$

$$\text{FOR} \;:\; \frac{\mathrm{A}\vdash e:\text{BOOL} \quad \mathrm{A}\vdash \mathit{Ass}_1;\ \mathit{Rest}:\tau \quad \mathrm{A}\vdash \mathit{Ass}_1;\ \mathit{Ass}_3;\ \mathit{Ass}_2;\ \mathit{Rest}:\tau'}{\mathrm{A}\vdash \text{FOR } (\mathit{Ass}_1;\ e;\ \mathit{Ass}_2)\ \{\mathit{Ass}_3;\};\ \mathit{Rest}:\tau''}$$

$$\Longleftrightarrow (\tau\sim\tau')\wedge(\tau''=\min(\ \tau,\ \tau'))\qquad .$$

Die Regel für die DO-Schleife bleibt als einzige unverändert, da es bei DO-Schleifen nicht erforderlich ist, zwei Typen für alternative Rumpfausdrücke zu inferieren.

Wie aus obigen Regeln ersichtlich, hat die Entscheidung, bei gleichnamigen Array-Variablen den gleichen Shape-Vektor zu fordern, weitreichende Auswirkungen. Sie sorgt bei vielen Regeln dafür, daß der speziellere von zwei alternativen Typen als Resultattyp inferiert werden kann. Dies kann zwar eventuell zu Laufzeitfehlern führen, in der Praxis ist dies jedoch äußerst selten der Fall.

Das Bestreben, einen möglichst speziellen Typ inferieren zu können, ist primär dadurch motiviert, möglichst effizient ausführbaren Code erzeugen zu wollen. Je spezieller ein inferierter Typ ist, um so laufzeiteffizienter kann der aus einem Programm compilierte Code sein. Betrachten wir unter diesem Gesichtspunkt noch einmal die Regel für Funktionsanwendungen (FUNAP): Falls der Typ des aktuellen Argumentes spezieller als der des für die Funktion deklarierten Parametertypes ist, wird diese Information einfach vernachlässigt. Der für die Funktion inferierte Resultattyp basiert ausschließlich auf den für die Funktion deklarierten, weniger speziellen Parametertypen. Bei Anwendungen, in denen Funktionen nur auf Argumente eines oder weniger verschiedener Typen angewendet werden, bedeutet dies, daß das Deklarieren speziellerer Funktionen evtl. erhebliche Laufzeitvorteile des compilierten Codes nach sich ziehen kann. Dies widerspricht jedoch deutlich einem der erklärten Ziele beim Design von SAC, nämlich dem Ziel, dimensionsunabhängiges Programmieren zu unterstützen. Deshalb soll die Regel für Funktionsdeklarationen mit Parametern (FUNDEF2) nochmals erweitert werden. Anstatt nur eine Inferenz mit den deklarierten Parametertypen zu ermöglichen, wird auch eine Inferenz mit spezielleren Typen für jeden Parameter zugelassen:

$$\text{FUNDEF2} \quad : \quad \frac{\{v_i : \tau_i\} \vdash B : \tau}{\text{A} \vdash \sigma F(\sigma_1\ v_1, ..., \sigma_n\ v_n)\ B : \bigotimes_{i=1}^{n} \tau_i \rightarrow \tau} \\ \iff \tau_i \preceq \sigma_i \wedge \tau \preceq \sigma \qquad .$$

Diese Regel nimmt eine ganz zentrale Rolle bei der Konzeption des Typsystems ein. Sie gestattet eine Spezialisierung von Funktionen durch das Typsystem. Allerdings bergen die dadurch gewonnenen Möglichkeiten einer spezielleren Typisierung auch Probleme. In diesem Zusammenhang muß die Regel FUNAP betrachtet werden, nachdem sie an die neue Regel FUNDEF2 angepaßt wurde.

$$\text{FUNAP} \quad : \quad \frac{\text{A} \vdash e_i : \sigma_i \quad \{\} \vdash \rho F(\rho_1\ v_1, ..., \rho_n\ v_n)\ B : \bigotimes_{i=1}^{n} \tau_i \rightarrow \tau}{\text{A} \vdash F(e_1 ... e_n) : \tau} \\ \iff \forall i \in \{1, ..., n\} : \sigma_i \preceq \tau_i \preceq \rho_i \wedge \tau \preceq \rho \qquad .$$

Der problematische Teil dieser Regel ist die Forderung $\sigma_i \preceq \tau_i \preceq \rho_i$ für alle Argumenttypen. Sie erlaubt es, verschiedene, wenn auch zueinander kompatible Resultattypen τ zu inferieren. Die Eindeutigkeit des Typsystems beschränkt sich dadurch auf Klassen kompatibler Typen. Eindeutigkeit entsteht erst durch eine Realisierung des Typsystems in einem Compiler. Im Bestreben um möglichst spezielle Typen liegt es zwar nahe, für jeden auftretenden Argument-Typ eine entsprechende Spezialisierung der Funktion vorzunehmen ($\tau_i = \sigma_i$), jedoch kann dies nicht in allen Fällen getan werden, da es bei rekursiven Funktionen zu unendlich vielen Spezialisierungen kom-

men kann. Eine konkrete Realisierung sollte daher entsprechende Abbruchkriterien bieten, um ein Terminieren der Typinferenz zu garantieren.

3.3.3 Primitive Operationen auf Arrays

Als Basis für die primitiven Operationen dient der bereits erwähnte Ψ-Kalkül. Er beschreibt einen Kalkül zur Vereinfachung von Kompositionen primitiver Array-Operationen. Über die Möglichkeiten der Vereinfachung hinaus sind die primitiven Operationen des Ψ-Kalkül so angelegt, daß die in Def. 3.3.2 beschriebene Äquivalenz von geschachtelten und mehrdimensionalen Arrays auch im Zusammenhang mit den primitiven Operationen aufrecht erhalten werden kann. Anschaulich gesehen bedeutet dies, daß eine k-fache Schachtelung von m-dimensionalen Operationen den gleichen Effekt hat, wie eine l-fache Schachtelung von entsprechenden n-dimensionalen Operationen, falls $k * m = l * n$.

Die hier vorgestellten primitiven Array-Operationen stimmen weitestgehend mit denen des Ψ-Kalkül s überein. Im wesentlichen kann zwischen fünf verschiedenen Arten von Operationen auf Arrays unterschieden werden:

1. Funktionen zur Erzeugung von Arrays,
2. Funktionen, die die Dimensionalität bzw. den Shape-Vektor eines Arrays bestimmen,
3. Funktionen zur Selektion von Elementen bzw. Teil-Arrays,
4. Funktionen, zur Veränderung der Form von Arrays, sowie
5. Funktionen, die die Elemente eines Arrays im Wert verändern.

Sowohl die Funktionalität all dieser Funktionen als auch ihre Einbindung in das Typinferenzsystem sollen im folgenden definiert werden.

Array-Erzeugung

Für die Erzeugung großer Arrays mit identischen Werten bietet SAC die primitive Funktion GENARRAY(*Shape*, *Const*). Sie bekommt einen Shape-Vektor und eine Konstante als Argumente und erzeugt daraus ein Array mit dem vorgegebenen Shape-Vektor und einem Datenvektor, der ausschließlich aus Elementen dieses einen Wertes besteht. Dabei darf die Konstante auch ein Array sein, was zu einer entsprechenden Erweiterung des Shape-Vektors gem. Def. 3.3.2 (Schachtelung von Arrays) führt. So erzeugt z.B. GENARRAY(`[3,5], 7`) das Array

$$\begin{pmatrix} 7 & 7 & 7 & 7 & 7 \\ 7 & 7 & 7 & 7 & 7 \\ 7 & 7 & 7 & 7 & 7 \end{pmatrix} \quad .$$

Das gleiche Array kann durch GENARRAY(`[3]`, GENARRAY(`[5]`, `7`)) ebenfalls spezifiziert werden.
Eine formale Darstellung der Funktionalität von GENARRAY bietet die

Definition 3.3.6 . *Sei S ein Vektor von natürlichen Zahlen mit Shape-Vektor $[n]$ und Datenvektor $[d_0, ..., d_{(n-1)}]$. Sei V eine in* SAC *zulässige Datenstruktur. Dann läßt sich* GENARRAY *definieren durch*

$$\text{GENARRAY} : \text{INT}[\bullet] \times \mathcal{T}_{\text{SAC}} \rightarrow \mathcal{T}_{Array}$$

mit

$$\text{GENARRAY}(S, V) \mapsto A' \qquad ,$$

wobei A' je nach Beschaffenheit von V unterschiedlich definiert ist:

- *falls V ein skalarer Wert ist, d.h. $V \in \mathcal{T}_{Basic}$,*
  ist A' ein Array mit Shape-Vektor $[d_0, ..., d_{(n-1)}]$ und Datenvektor $[\underbrace{V, ..., V}_{\prod_0^{(n-1)} d_i}]$;

- *falls V selbst ein Array mit*
  Shape-Vektor $[s_0, ..., s_{(k-1)}]$ und Datenvektor $[v_0, ..., v_{(m-1)}]$ ist,
  ist A' ein Array mit
  Shape-Vektor $[d_0, ..., d_{(n-1)}, s_0, ..., s_{(k-1)}]$ und
  Datenvektor $[\underbrace{v_0, ..., v_{(m-1)}, ..., v_0, ..., v_{(m-1)}}_{\prod_0^{(n-1)} d_i}]$.

3.3.6□

Die in dieser Definition angegebene Signatur der Funktion GENARRAY ist zwar mathematisch korrekt, für eine Typinferenz mit Hilfe der PRFAP-Regel aus Abschnitt 3.2 jedoch zu ungenau. So läßt sich z.B. aus ihr fast keinerlei Information über den Resultattyp ableiten. Da die Situation bei fast allen primitiven Array-Operationen ähnlich ist, werden Anwendungen dieser Funktionen nicht mittels des Schemas PRFAP, sondern mit speziellen Schemata für die jeweilige Funktion getypt, um einen möglichst speziellen Resultattyp inferieren zu können. Für GENARRAY ergibt sich:

Die exakte Form des erzeugten Arrays ist in keinem Falle inferierbar, da sie vom Wert des ersten Argumentes abhängt. Jedoch läßt sich immer dann wenigstens die Dimensionalität des Resultats herleiten, wenn die genaue Form des ersten Argumentes und ggf. die Dimensionalität des zweiten Argumentes bekannt sind:

$$\text{GEN1} \;:\; \frac{\text{A}\vdash s:\text{INT}[n] \quad \text{A}\vdash a:\sigma}{\text{A}\vdash \text{GENARRAY}(\,s,\,a):\sigma[\overbrace{\bullet,\ldots,\bullet}^{n}]} \quad ,$$

$$\text{GEN2} \;:\; \frac{\text{A}\vdash s:\text{INT}[n] \quad \text{A}\vdash a:\sigma[s_1,\ldots,s_m]}{\text{A}\vdash \text{GENARRAY}(\,s,\,a):\sigma[\overbrace{\bullet,\ldots,\bullet}^{n},s_1,\ldots,s_m]} \quad \Longleftrightarrow \quad s_i \in \mathbb{N}_\bullet \quad .$$

Fehlt die Information über die Form des ersten oder die Dimensionalität des zweiten Argumentes, so kann lediglich ein allgemeiner Array-Typ abgeleitet werden:

$$\text{GEN3} \;:\; \frac{\text{A}\vdash s:\rho \quad \text{A}\vdash a:\tau}{\text{A}\vdash \text{GENARRAY}(\,s,\,a):\sigma[\,]}$$

$$\Longleftrightarrow (\text{INT}[\bullet] \preceq \rho \wedge \tau \preceq \sigma[?]) \vee (\text{INT}[n] \preceq \rho \wedge \tau = \sigma[\,]) \quad .$$

Form-inspizierende Operationen

SAC bietet zwei Operationen, um die Form eines Arrays festzustellen: Eine Funktion DIM(*Array*), die die Dimensionalität eines Arrays bestimmt und eine Funktion SHAPE(*Array*), die den Shape-Vektor eines Arrays liefert. Hierzu soll exemplarisch folgendes Array A betrachtet werden, das bei allen weiteren Beispielen zu den primitiven Array-Operationen in SAC wieder aufgegriffen werden wird. Sei

$$\mathtt{A} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix} = \text{RESHAPE}([3,4],[1,2,3,4,5,6,7,8,9,10,11,12]) \quad .$$

Dann gilt:

$$\begin{array}{ll} \text{DIM}(\mathtt{A}) & = 2, \\ \text{SHAPE}(\mathtt{A}) & = [3,4], \text{ sowie} \\ \text{SHAPE}(\text{SHAPE}(\mathtt{A})) & = [2] \end{array} \quad .$$

Formal können wir DIM und SHAPE folgendermaßen definieren:

Definition 3.3.7 . *Sei A ein Array mit Shape-Vektor $[s_0,\ldots,s_{(n-1)}]$ und Datenvektor $[d_0,\ldots,d_{(m-1)}]$. Dann wird* DIM *definiert durch*

$$\text{DIM} : \mathcal{T}_{Array} \rightarrow \text{INT}$$

mit

$$\text{DIM}(A) \mapsto n \quad .$$

SHAPE *wird definiert als:*

$$\text{SHAPE} : \mathcal{T}_{Array} \rightarrow \mathcal{T}_{Array}$$

mit

$$\text{SHAPE}(A) = A' \quad ,$$

wobei A' *ein Array mit Shape-Vektor* $[n]$ *und Datenvektor* $[s_0, ..., s_{(n-1)}]$ *ist.*

3.3.7□

Für die Typinferenz von Anwendungen von DIM und SHAPE ergeben sich folgende Regeln. Auf ein Array angewandt liefert die Funktion DIM immer einen Wert vom Typ INT:

$$\text{DIM} \quad : \quad \frac{\text{A}\vdash a:\tau}{\text{A}\vdash \text{DIM}(a):\text{INT}} \qquad \Longleftrightarrow \qquad \tau \in \mathcal{T}_{Array} \quad .$$

Für das Ergebnis einer Anwendung von SHAPE ist nur dann der Shape-Vektor des Resultattyps bekannt, wenn der Typ des Argumentes wenigstens die Dimensionalität des Argumentes enthält:

$$\text{SHAPE1} \quad : \quad \frac{\text{A}\vdash a:\tau}{\text{A}\vdash \text{SHAPE}(a):\text{INT}[n]} \qquad \Longleftrightarrow \qquad \tau \preceq \sigma[\overbrace{\bullet, ..., \bullet}^{n}] \quad ;$$

ansonsten kann nur inferiert werden, daß es sich um einen Vektor mit Elementen des Typs INT handelt:

$$\text{SHAPE2} \quad : \quad \frac{\text{A}\vdash a:\tau[\,]}{\text{A}\vdash \text{SHAPE}(a):\text{INT}[\bullet]} \quad .$$

Selektion

Für die Selektion von Teil-Arrays sowie einzelnen Elementen eines Arrays bietet SAC eine Funktion PSI(Idx, $Array$) $\equiv Array[Idx]$. Sie bekommt einen Indexvektor und ein Array als Argumente und selektiert das durch den Indexvektor bezeichnete Element bzw. Teil-Array des Arrays. Dabei beginnt die Numerierung in jeder Achse bei Null (vergl. Def. 3.3.1(3.2)). Entspricht die Länge des Indexvektors der Dimensionalität des Arrays, so wird das korrespondierende Element selektiert; ist sie kleiner, so wird aus dem Array ein Teil-Array selektiert. Die Elemente des Teil-Arrays werden durch alle legalen Indizes in das Array charakterisiert, für die der Indexvektor ein Prefix ist. Unter Verwendung des bei den form-inspizierenden Operationen eingeführten Beispiel-Arrays A gilt

$$\begin{array}{llll}
\text{PSI}([1,0],\text{A}) & = \text{A}[[1,0]] & = 5 & , \\
\text{PSI}([1],\text{A}) & = \text{A}[[1]] & = [5,6,7,8] & , \\
\text{DIM}(\text{PSI}([1],\text{A})) & = \text{DIM}(\text{A}[[1]]) & = 1 & , \\
\text{SHAPE}(\text{PSI}([1],\text{A})) & = \text{SHAPE}(\text{A}[[1]]) & = [4] & .
\end{array}$$

PSI ist definiert durch

Definition 3.3.8 . *Sei A ein Array mit Shape-Vektor $[s_0, ..., s_{(n-1)}]$ und Datenvektor $[d_0, ..., d_{(m-1)}]$ und S ein Vektor von ganzen Zahlen mit Shape-Vektor $[k]$ und Datenvektor $[d'_0, ..., d'_{(k-1)}]$. Dann wird* PSI *definiert durch*

$$\text{PSI} : \text{INT}[\bullet] \times \mathcal{T}_{Array} \rightarrow_{part} \mathcal{T}_{\text{SAC}}$$

mit

$$\text{PSI}(S, A) \mapsto \begin{cases} d_l & \textit{falls } S \textit{ mit } d_l \textit{ gem. Def. 3.3.1(3.3) korrespondiert} \\ A' & \textit{falls } k < n \ \wedge \ \forall i \in \{1, ..., k\} : 0 \leq d'_i \leq s_i \end{cases} ,$$

wobei A' ein Array mit

Shape-Vektor $[s_{(k+1)}, ..., s_n]$ und Datenvektor $[d_{start}, ..., d_{(start+size-1)}]$
mit $start = \sum_{i=0}^{(k-1)} \left(d'_i * \prod_{j=i+1}^{(n-1)} s_j \right)$ und $size = \prod_{i=k}^{(n-1)} s_i$ ist.

3.3.8□

Beim Entwickeln der Typinferenzregeln für PSI tritt dadurch ein Problem auf, daß je nach Argumentkonstellation entweder ein Basistyp oder aber ein Array als Ergebnis möglich ist. Deshalb wird ein Supertyp $\tau[?]$ eingeführt, der sowohl alle Skalare des atomaren Typs τ, als auch alle Arrays des Typs $\tau[\,]$ umfaßt. Eine entsprechende Erweiterung der in Abb. 3.6 beschriebenen Hierarchie von Array-Typen ist in Abb. 3.7 dargestellt.

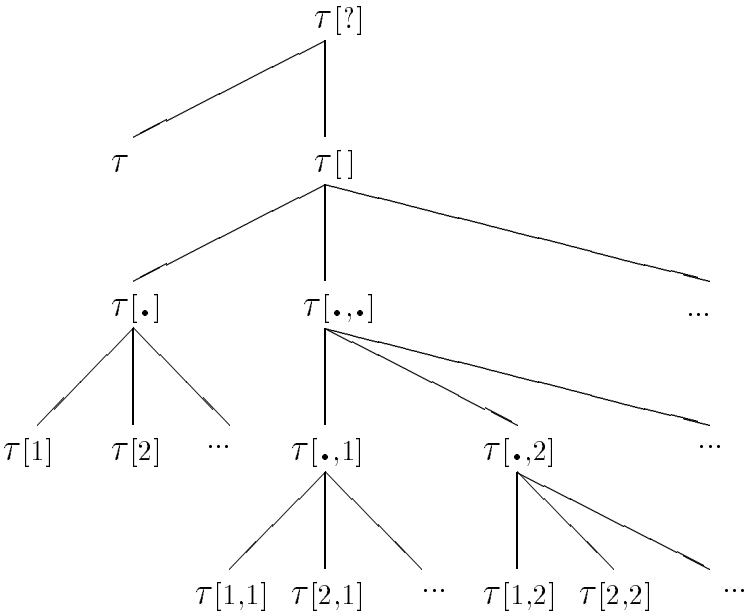


Abbildung 3.7: Erweiterte Array-Typen-Hierarchie in SAC.

Da die Typen der Form $\tau[?]$ jedoch eine sehr große Menge von Datenobjekten umfassen, sind sie nicht Bestandteil der Sprache SAC, sondern werden ausschließlich für die Typinferenz genutzt. Formal führt dies zu den folgenden Erweiterungen des Typinferenzsystems.

Definition 3.3.9 . *Seien $\mathcal{T}_{Prod}$ und $\mathcal{T}_{Fun}$ wie in Def.3.2.2 gegeben. Dann wird $\mathcal{T}_{Infer}$ umdefiniert als:*

$$\mathcal{T}_{Infer} = \mathcal{T}_{Prod} \cup \mathcal{T}_{Fun} \cup \{\tau[?] : \tau \in \mathcal{T}_{Basic}\} \quad .$$

Entsprechend wird die Subtyprelation $\preceq$ aus Def.3.3.5 umdefiniert durch

$$\preceq \ \subset \mathcal{T}_{\mathrm{SAC}} \times \mathcal{T}_{\mathrm{SAC}} \cup \mathcal{T}_{Infer}$$

mit

$$\begin{aligned}\tau_1 \preceq \tau_2 := \ & \tau_1 \preceq \tau_2 \ \textit{gem. Def. 3.3.5} \\ & \cup \{(\tau_1, \tau_2) \mid \tau_1 \in \mathcal{T}_{Basic} \ \wedge \ \tau_2 = \tau_1[?]\} \\ & \cup \{(\tau_1, \tau_2) \mid \tau_1 = \sigma[?] = \tau 2\} \\ & \cup \{(\tau_1, \tau_2) \mid \tau_1 = \sigma[s_1, ..., s_n], \tau 2 = \sigma[?], n \in \mathbb{N}, s_i \in \mathbb{N}_+\} \\ & \cup \{(\tau_1, \tau_2) \mid \tau_1 = \sigma[\,], \tau 2 = \sigma[?]\} \quad .\end{aligned}$$

3.3.9□

Mit Hilfe dieser zusätzlichen Typen lassen sich folgende Inferenzregeln für die Funktion PSI entwickeln. Sind der Shape-Vektor des ersten Argumentes und die Dimension des zweiten Argumentes bekannt, so kann entschieden werden, ob der Resultatstyp ein skalarer Typ oder ein Array ist:

$$\text{PSI1} \ : \ \frac{\mathrm{A}\vdash s : \mathrm{INT}[n] \quad \mathrm{A}\vdash a : \tau}{\mathrm{A}\vdash \mathrm{PSI}(\,s,\,a) : \sigma} \qquad \Longleftrightarrow \qquad \tau \preceq \sigma[\overbrace{\bullet, ..., \bullet}^{n}] \quad ,$$

$$\text{PSI2} \ : \ \frac{\mathrm{A}\vdash s : \mathrm{INT}[n] \quad \mathrm{A}\vdash a : \tau}{\mathrm{A}\vdash \mathrm{PSI}(\,s,\,a) : \sigma[r_{(n+1)}, ..., r_m]}$$

$$\Longleftrightarrow \tau = \sigma[r_1, ..., r_m] \text{ mit } n < m \text{ und } r_i \in \mathbb{N}_+ \quad .$$

In allen anderen Fällen kann nicht entschieden werden, ob es sich beim Resultat um einen skalaren Wert oder ein Array handelt. Deshalb muß hier der Typ $\sigma[?]$ inferiert werden.

$$\text{PSI3} \ : \ \frac{\mathrm{A}\vdash s : \rho \quad \mathrm{A}\vdash a : \tau}{\mathrm{A}\vdash \mathrm{PSI}(\,s,\,a) : \sigma[?]}$$

$$\Longleftrightarrow (\rho = \mathrm{INT}[n] \wedge \tau = \sigma[\,]) \vee (\mathrm{INT}[\bullet] \preceq \rho \wedge \tau \preceq \sigma[\,]) \quad .$$

Formverändernde Operationen

Es gibt vier formverändernde Array-Funktionen in SAC: RESHAPE, TAKE, DROP und CAT.

Die Funktion RESHAPE(*Shape*, *Array*) ist bereits aus dem Abschnitt 3.3.1 bekannt. Sie ordnet einem Array einen neuen Shape-Vektor zu, falls die Anzahl der Elemente des Arrays mit der für die neue Form benötigten Anzahl an Elementen übereinstimmt. So gilt für das oben eingeführte Beispiel-Array A:

$$\text{RESHAPE}([2,6],\mathtt{A}) = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \end{pmatrix} \quad .$$

Formal läßt sich RESHAPE definieren durch die

Definition 3.3.10 . *Seien A ein Array mit Shape-Vektor $[s_0, ..., s_{(n-1)}]$ und Datenvektor $[d_0, ..., d_{(m-1)}]$ und S ein Vektor von natürlichen Zahlen mit Shape-Vektor $[k]$ und Datenvektor $[d'_0, ..., d'_{(k-1)}]$. Dann wird* RESHAPE *definiert durch*

$$\text{RESHAPE} : \text{INT}[\bullet] \times \mathcal{T}_{Array} \rightarrow_{part} \mathcal{T}_{Array}$$

mit

$$\text{RESHAPE}(S, A) \mapsto A' \iff \prod_{i=0}^{(n-1)} s_i = m = \prod_{i=0}^{(k-1)} d'_i \qquad ,$$

wobei A' ein Array mit Shape-Vektor $[d'_0, ..., d'_{(k-1)}]$ und Datenvektor $[d_0, ..., d_{(m-1)}]$ ist.

3.3.10□

Falls die Größe des Shape-Vektors bekannt ist, läßt sich die Dimensionalität des Resultates inferieren mittels

$$\text{RESHAPE1} \quad : \quad \frac{\mathrm{A}\vdash s : \text{INT}[n] \quad \mathrm{A}\vdash a : \tau}{\mathrm{A}\vdash \text{RESHAPE}(\ s, a) : \sigma[\overbrace{\bullet, ..., \bullet}^{n}]} \qquad \Longleftrightarrow \qquad \tau \preceq \sigma[\,] \qquad ;$$

ansonsten muß ein allgemeiner Resultatstyp angenommen werden:

$$\text{RESHAPE2} \quad : \quad \frac{\mathrm{A}\vdash s : \rho \quad \mathrm{A}\vdash a : \tau}{\mathrm{A}\vdash \text{RESHAPE}(\ s, a) : \sigma[\,]} \qquad \Longleftrightarrow \qquad \tau \preceq \sigma[\,] \wedge \text{INT}[\bullet] \preceq \rho \qquad .$$

Die Funktion TAKE(*Shape*, *Array*) selektiert ein Teil-Array des zweiten Argumentes, das bezüglich jeder Dimension soviele Elemente hat, wie das erste Argument vorgibt. Dabei werden jeweils die ersten Elemente in jeder Dimension selektiert. Wenn das erste Argument weniger Komponenten hat als die Dimensionalität des

zweiten Argumentes, so werden von den fehlenden Achsen jeweils alle Elemente selektiert. Unter Verwendung des obiges Beispiel-Arrays A gilt:

$$\begin{array}{lll}
\text{TAKE}([2,2],\texttt{A}) & = \begin{pmatrix} 1 & 2 \\ 5 & 6 \end{pmatrix} & , \\
\text{TAKE}([2],\texttt{A}) & = \text{TAKE}([2,4],\texttt{A}) = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix} & , \\
\text{DIM}(\text{TAKE}([2],\texttt{A})) & = 2 & \text{sowie} \\
\text{SHAPE}(\text{TAKE}([2],\texttt{A})) & = [2,4] & .
\end{array}$$

Die Formalisierung der Funktionalität von TAKE liefert die

Definition 3.3.11 . *Seien A ein Array mit Shape-Vektor $[s_0, ..., s_{(n-1)}]$ und Datenvektor $[d_0, ..., d_{(m-1)}]$ und S ein Vektor von natürlichen Zahlen mit Shape-Vektor $[k]$ und Datenvektor $[d'_0, ..., d'_{(k-1)}]$. Dann ist* TAKE *definiert durch*

$$\text{TAKE} : \text{INT}[\bullet] \times \mathcal{T}_{Array} \rightarrow_{part} \mathcal{T}_{Array}$$

mit

$$\text{TAKE}(S, A) \mapsto A' \iff k \leq n \ \wedge \ \forall i \in \{0, ..., (k-1)\} : d'_i \leq s_i \quad ,$$

wobei $\text{SHAPE}(A') = [d'_0, ..., d'_{(k-1)}, s_k, ..., s_{(n-1)}]$ *und für alle legalen Indizes i in A' gilt:* $\text{PSI}(i, A') = \text{PSI}(i, A)$.

3.3.11□

Für das Typinferenzsystem ergibt sich, daß der genaue Shape-Vektor des Ergebnistyps von dem konkreten Wert des ersten Argumentes abhängt und daher nicht inferierbar ist. Lediglich unter der Voraussetzung, daß die Shape-Vektoren beider Argumente bekannt sind, und daß der Zugriffs-Vektor kürzer als die Dimensionalität des Arrays ist, kann ein speziellerer als ein allgemeiner Array-Typ inferiert werden

$$\text{TAKE1} \ : \ \frac{\text{A}\vdash s : \text{INT}[m] \quad \text{A}\vdash a : \sigma[s_1, ..., s_n]}{\text{A}\vdash \text{TAKE}(s, a) : \sigma[\overbrace{\bullet, ..., \bullet}^{m}, s_{(m+1)}, ..., s_n]} \quad \iff \quad s_i \in \mathbb{N}_\bullet \quad .$$

In allen anderen Fällen muß ein allgemeiner Resultatstyp inferiert werden:

$$\text{TAKE2} \ : \ \frac{\text{A}\vdash s : \rho \quad \text{A}\vdash a : \sigma[\,]}{\text{A}\vdash \text{TAKE}(s, a) : \sigma[\,]} \quad \iff \quad \text{INT}[n] \preceq \rho \quad .$$

Die Funktion DROP(*Shape*, *Array*) ist komplementär zu TAKE. Anstatt die durch das erste Argument spezifizierte Anzahl von Elementen zu selektieren, entfernt DROP entsprechend viele Elemente. Fehlende Komponenten des ersten Argumentes werden

hier als Null angenommen. So gilt für das Beispiel-Array A

$$\begin{array}{ll} \text{DROP}([2,2],\text{A}) & = \begin{pmatrix} 11 & 12 \end{pmatrix} \quad , \\ \text{DROP}([2],\text{A}) & = \text{DROP}([2,0],\text{A}) = \begin{pmatrix} 9 & 10 & 11 & 12 \end{pmatrix} \quad , \\ \text{DIM}(\text{DROP}([2],\text{A})) & = 2 \quad \text{sowie} \\ \text{SHAPE}(\text{DROP}([2],\text{A})) & = [1,4] \quad . \end{array}$$

Eine Formalisierung von DROP liefert die

Definition 3.3.12 . *Seien A ein Array mit Shape-Vektor $[s_0,...,s_{(n-1)}]$ und Datenvektor $[d_0,...,d_{(m-1)}]$ und S ein Vektor von natürlichen Zahlen mit Shape-Vektor $[k]$ und Datenvektor $[d'_0,...,d'_{(k-1)}]$. Dann ist* DROP *definiert als*

$$\text{DROP} : \text{INT}[\bullet] \times \mathcal{T}_{Array} \rightarrow_{part} \mathcal{T}_{Array}$$

mit

$$\text{DROP}(S,A) \mapsto A' \Longleftrightarrow k \leq n \ \wedge \ \forall i \in \{1,...,k\} : d'_i \leq s_i \quad ,$$

wobei $\text{SHAPE}(A') = [(s_0 - d'_0),...,(s_{(k-1)} - d'_{(k-1)}), s_k, ..., s_n]$ *und für alle legalen Indizes i in A' gilt:* $\text{PSI}(i,A') = \text{PSI}(i + [s_0,...,s_{(k-1)}, \underbrace{0,...,0}_{(n-k)}], A)$.

3.3.12□

Die Typinferenzregeln für DROP entsprechen denen von TAKE. Nur in den wenigsten Fällen läßt sich etwas über den Shape-Vektor des Resultates aussagen:

$$\text{DROP1} \ : \ \frac{\text{A}\vdash s : \text{INT}[m] \quad \text{A}\vdash a : \sigma[s_1,...,s_n]}{\text{A}\vdash \text{DROP}(s,a) : \sigma[\overbrace{\bullet,...,\bullet}^{m}, s_{(m+1)},...,s_n]} \quad \Longleftrightarrow \quad s_i \in \mathbb{N}_\bullet \quad .$$

In allen anderen Fällen muß auch hier ein allgemeiner Resultattyp inferiert werden:

$$\text{DROP2} \ : \ \frac{\text{A}\vdash s : \rho \quad \text{A}\vdash a : \sigma[\,]}{\text{A}\vdash \text{DROP}(s,a) : \sigma[\,]} \quad \Longleftrightarrow \quad \text{INT}[n] \preceq \rho \quad .$$

Die letzte formverändernde Array-Operation ist CAT(*Dim*, *Array*, *Array*). Sie konkateniert zwei Arrays bezüglich einer über das erste Argument vorgebbaren Achse, falls die beiden Arrays in allen übrigen Achsen die gleiche Form haben. Dabei werden die Achsen eines n-dimensionales Arrays von 0 bis $n-1$ gezählt. Eine Konkatenation mit dem Array A ist z.B. folgendermaßen möglich:

$$\text{CAT}(1,\text{A},\text{RESHAPE}([3,1],[13,14,15])) = \begin{pmatrix} 1 & 2 & 3 & 4 & 13 \\ 5 & 6 & 7 & 8 & 14 \\ 9 & 10 & 11 & 12 & 15 \end{pmatrix} \quad .$$

Formal ergibt sich

Definition 3.3.13 . *Sei A ein Array mit Shape-Vektor $[s_0, ..., s_{(n-1)}]$ und Datenvektor $[d_0, ..., d_{(m-1)}]$ sowie B ein Array mit Shape-Vektor $[r_0, ..., r_{(n'-1)}]$ und Datenvektor $[c_0, ..., c_{(m'-1)}]$. Dann ist* CAT *definiert als*

$$\text{CAT} : \text{INT} \times \mathcal{T}_{Array} \times \mathcal{T}_{Array} \rightarrow_{part} \mathcal{T}_{Array}$$

mit

$$\text{CAT}(d, A, B) \mapsto A' \iff \begin{array}{l} 0 \leq d < n = n' \\ \wedge\ \forall i \in \{0, ..., n-1\} : (i \neq d) \Longrightarrow r_i = s_i \end{array} \quad ,$$

wobei $\text{SHAPE}(A') = [s_0, ..., s_{(d-1)}, s_d + r_d, s_{(d+1)}, ..., s_{(n-1)}]$ *und für alle legalen Indizes i in A' gilt:*

$$\text{PSI}(i, A') = \begin{cases} \text{PSI}(i, A) & \textit{falls } \text{PSI}([d], i) < s_d \\ \text{PSI}(i - [\underbrace{0, ..., 0}_{(d)}, s_d, \underbrace{0, ..., 0}_{(n-d-1)}], B) & \textit{sonst} \end{cases} .$$

3.3.13□

Da sich aus dem Typ des ersten Argumentes (INT) nicht ableiten läßt, bezüglich welcher Achse konkateniert werden soll, läßt sich auch nicht die exakte Form, sondern höchstens die Dimensionalität des Ergebnisses inferieren. Selbst dies ist nur dann möglich, wenn sichergestellt ist, daß beide die gleiche Dimensionalität haben können und von mindestens einem der beiden Arrays die Dimensionalität bekannt ist:

$$\text{CAT1} \quad : \quad \frac{\text{A}\vdash d : \text{INT} \quad \text{A}\vdash a : \tau \quad \text{A}\vdash b : \rho}{\begin{array}{l} \text{A}\vdash \text{CAT}(d, a, b) : \sigma[\overbrace{\bullet, ..., \bullet}^{n}] \\ \Longleftrightarrow (\exists r_i, s_i \in \mathbb{N}_\bullet : \sigma[s_1, ..., s_n] \preceq \tau \wedge \sigma[r_1, ..., r_n] \preceq \rho) \\ \quad \wedge \neg(\tau = \rho = \sigma[\,]) \end{array}} .$$

Ist jedoch von beiden Arrays die Dimensionalität unbekannt, so kann lediglich der Elementtyp des Resultat-Arrays inferiert werden:

$$\text{CAT2} \quad : \quad \frac{\text{A}\vdash d : \text{INT} \quad \text{A}\vdash a : \sigma[\,] \quad \text{A}\vdash b : \sigma[\,]}{\text{A}\vdash \text{CAT}(d, a, b) : \sigma[\,]} \quad .$$

Wertverändernde Operationen

Die Operationen zur Veränderung der Elemente eines Arrays bestehen aus den elementweisen Erweiterungen der primitiven arithmetischen Operationen $+, -, *, /$ sowie den Funktionen ROTATE und MODARRAY.

Bei der Überladung der arithmetischen Operationen wird nicht nur die elementweise Verknüpfung zweier Arrays, sondern auch die elementweise Verknüpfung eines Arrays mit einem skalaren Wert eingeführt. So gilt für das Beispiel-Array A

$$\texttt{3} * \texttt{A} = \begin{pmatrix} 3 & 6 & 9 & 12 \\ 15 & 18 & 21 & 24 \\ 27 & 30 & 33 & 36 \end{pmatrix} \quad \text{sowie} \quad \texttt{A} + \texttt{A} = \begin{pmatrix} 2 & 4 & 6 & 8 \\ 10 & 12 & 14 & 16 \\ 18 & 20 & 22 & 24 \end{pmatrix} \quad .$$

Diese Überladungen lassen sich formulieren als

Definition 3.3.14 . *Sei AriOp eine arithmetische Funktion aus* $\{+, -, *, /\}$. *Außerdem bezeichne VAL(AriOp(x,y)) das Ergebnis der Anwendung von AriOp auf zwei skalare Werte x und y. Seien weiterhin A ein Array mit Shape-Vektor* $[s_0, ..., s_{(n-1)}]$ *und Datenvektor* $[d_0, ..., d_{(m-1)}]$ *und B ein Array mit Shape-Vektor* $[r_0, ..., r_{(n'-1)}]$ *und Datenvektor* $[c_0, ..., c_{(m'-1)}]$ *sowie V ein skalarer Wert. Dann lassen sich folgende Funktionen* $AriOp_{AS}$, $AriOp_{SA}$ *und* $AriOp_{AA}$ *definieren.*

$$AriOp_{AS} : \mathcal{T}_{SimpleArray} \times \mathcal{T}_{Simple} \rightarrow_{part} \mathcal{T}_{SimpleArray}$$

mit

$$AriOp(A, V) \mapsto A' \quad ,$$

wobei A' *ein Array mit*
Shape-Vektor $[s_0, ..., s_{(n-1)}]$ *und*
Datenvektor $[VAL(AriOp(d_0, V)),\ ...,\ VAL(AriOp(d_{(m-1)}, V))]$ *ist.*

$$AriOp_{SA} : \mathcal{T}_{Simple} \times \mathcal{T}_{SimpleArray} \rightarrow_{part} \mathcal{T}_{SimpleArray}$$

mit

$$AriOp(V, A) \mapsto A' \quad ,$$

wobei A' *ein Array mit*
Shape-Vektor $[s_0, ..., s_{(n-1)}]$ *und*
Datenvektor $[VAL(AriOp(V, d_0)),\ ...,\ VAL(AriOp(V, d_{(m-1)}))]$ *ist.*

$$AriOp_{AA} : \mathcal{T}_{SimpleArray} \times \mathcal{T}_{SimpleArray} \rightarrow_{part} \mathcal{T}_{SimpleArray}$$

mit

$$AriOp(A, B) \mapsto A' \Longleftrightarrow n = n' \wedge \forall i \in \{0, ..., n-1\} : r_i = s_i \quad ,$$

wobei A' *ein Array mit*
Shape-Vektor $[s_0, ..., s_{(n-1)}]$ *und*
Datenvektor $[VAL(AriOp(d_0, c_0)),\ ...,\ VAL(AriOp(d_{(m-1)}, c_{(m-1)}))]$ *ist.*

Mit diesen Funktionen lassen sich schließlich die arithmetischen Funktionen folgendermaßen erweitern. Es sei

$$AriOp : \mathcal{T}_{\text{SAC}} \times \mathcal{T}_{\text{SAC}} \rightarrow_{part} \mathcal{T}_{\text{SAC}}$$

mit

$$AriOp(x,y) = \begin{cases} VAL(AriOp(x,y)) & falls\ x,y \in \mathcal{T}_{Simple} \\ AriOp_{SA}(x,y) & falls\ x \in \mathcal{T}_{Simple} \wedge y \in \mathcal{T}_{SimpleArray} \\ AriOp_{AS}(x,y) & falls\ x \in \mathcal{T}_{SimpleArray} \wedge y \in \mathcal{T}_{Simple} \\ AriOp_{AA}(x,y) & falls\ x,y \in \mathcal{T}_{SimpleArray} \end{cases} .$$

3.3.14□

Die zugehörigen Typinferenzregeln reflektieren obige Fallunterscheidung. Im Falle der Anwendung auf einen skalaren Wert und ein Array resultiert die Anwendung in einem Array, das den gleichen Typ wie das Array-Argument hat

$$\text{ARIAS} : \frac{\text{A}\vdash a:\tau \quad \text{A}\vdash s:\sigma}{\text{A}\vdash \text{ARIOP}(a,\,s):\tau} \iff \tau \preceq \sigma[\,] \quad ,$$

$$\text{ARISA} : \frac{\text{A}\vdash s:\sigma \quad \text{A}\vdash a:\tau}{\text{A}\vdash \text{ARIOP}(s,\,a):\tau} \iff \tau \preceq \sigma[\,] \quad .$$

Bei der Anwendung auf zwei Arrays entspricht der Resultattyp dem spezielleren der beiden Argument-Typen, da die Funktion ARIOP ohnehin Gleichheit der Shape-Vektoren der beiden Argumente erfordert:

$$\text{ARIAA} : \frac{\text{A}\vdash a:\tau \quad \text{A}\vdash b:\sigma}{\text{A}\vdash \text{ARIOP}(a,\,b):\rho}$$
$$\iff (\rho = \tau \preceq \sigma) \ \vee \ (\rho = \sigma \preceq \tau) \quad .$$

Die Funktion ROTATE(Dim, Num, $Array$) rotiert die Elemente eines Arrays bezüglich einer durch das erste Argument vorgebbaren Achse um eine durch das zweite Argument vorgebbare Anzahl von Elementen. Dabei stehen positive Anzahlen für ein Rotieren in Richtung aufsteigender Indizes und respektive negative für ein Rotieren in Richtung fallender Indizes. Für das Beispiel-Array gilt:

$$\text{ROTATE}(1,1,\mathtt{A}) = \begin{pmatrix} 4 & 1 & 2 & 3 \\ 8 & 5 & 6 & 7 \\ 12 & 9 & 10 & 11 \end{pmatrix} \quad .$$

Formal ergibt sich die

Definition 3.3.15 . *Sei A ein n-dimensionales Array mit Shape-Vektor $[s_0, ..., s_{(n-1)}]$ und Datenvektor $[d_0, ..., d_{(m-1)}]$. Dann ist* ROTATE *definiert als*

$$\text{ROTATE} : \text{INT} \times \text{INT} \times \mathcal{T}_{SimpleArray} \rightarrow_{part} \mathcal{T}_{SimpleArray}$$

mit

$$\text{ROTATE}(d, n, A) = A' \Longleftrightarrow 0 \leq d < n \quad ,$$

wobei $\text{SHAPE}(A') = [s_0, ..., s_{(n-1)}]$ *und für alle legalen Indizes* $i = [i_0, ..., i_{(n-1)}]$ *gilt:*

$$\text{PSI}(i, A') = \text{PSI}([i_0, ..., i_{(d-1)}, (i_d - n) \bmod s_d, i_{(d+1)}, ..., i_{(n-1)}] \quad .$$

3.3.15□

Als Typinferenzregel ergibt sich

$$\text{ROT} \quad : \quad \frac{\text{A}\vdash d:\text{INT} \quad \text{A}\vdash n:\text{INT} \quad \text{A}\vdash a:\tau}{\text{A}\vdash \text{ROTATE}(\ d,\ n,\ a):\tau} \quad .$$

Mit Hilfe der Funktion $\text{MODARRAY}(Shape, Val, Array)$ ist es dem Programmierer schließlich möglich, gezielt Array-Elemente oder Teil-Arrays zu modifizieren. Der als erstes Argument übergebene Vektor natürlicher Zahlen selektiert dabei genauso wie bei der Funktion PSI ein Element bzw. Teil-Array, welches durch den Wert des zweiten Argumentes ersetzt wird. Unter nochmaliger Verwendung des Beispiel-Arrays gilt:

$$\text{MODARRAY}([1,2], 42, \texttt{A}) = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 42 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix} \qquad \text{sowie}$$

$$\text{MODARRAY}([1], [20,21,22,23], \texttt{A}) = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 20 & 21 & 22 & 23 \\ 9 & 10 & 11 & 12 \end{pmatrix} \quad .$$

Formal erhalten wir die

Definition 3.3.16 . *Seien A ein Array mit Shape-Vektor $[s_0, ..., s_{(n-1)}]$ und Datenvektor $[d_0, ..., d_{(m-1)}]$ und S ein Vektor von natürlichen Zahlen mit Shape-Vektor $[k]$ und Datenvektor $[d'_0, ..., d'_{(k-1)}]$. Dann ist* MODARRAY *definiert als:*

$$\text{MODARRAY} : \text{INT}[\bullet] \times \mathcal{T}_{\text{SAC}} \times \mathcal{T}_{SimpleArray} \rightarrow_{part} \mathcal{T}_{SimpleArray}$$

mit

$$\begin{aligned} &\text{MODARRAY}(S, V, A) = A' \\ \Longleftrightarrow &\begin{pmatrix} (k = n \ \wedge \ \forall i \in \{0, ..., n-1\} : 0 \leq d'_i < s_i \ \wedge \ V \in \mathcal{T}_{Simple}) \\ \vee \begin{pmatrix} k < n \ \wedge \ \forall i \in \{0, ..., k-1\} : 0 \leq d'_i < s_i \\ \wedge \ V \textit{ ist ein Array mit Shape-Vektor} [s_k, ..., s_{(n-1)}] \end{pmatrix} \end{pmatrix} \quad , \end{aligned}$$

wobei A' ein Array ist mit SHAPE*(A')=$[s_0, ..., s_{(n-1)}]$ und für alle legalen Indizes $i = [i_0, ..., i_{(n-1)}]$ gilt:*

$$\text{PSI}(i, A') = \begin{cases} \text{PSI}(i, A) & \text{falls } \exists j \in \{0, ..., k-1\} : i_j \neq d'_j \\ V & \text{falls } k = n \ \wedge \ \forall j \in \{0, ..., k-1\} i_j = d'_j \\ \text{PSI}([i_k, ..., i_{(n-1)}], V) & \text{falls } k < n \ \wedge \ \forall j \in \{0, ..., k-1\} i_j = d'_j \end{cases} .$$

3.3.16□

Die Restriktionen in der Anwendbarkeit von MODARRAY finden sich entsprechend in den Typinferenzregeln wieder. Für die Modifikation einzelner Array-Elemente gilt

$$\text{MOD1} \ : \ \frac{\text{A}\vdash s:\rho \ \ \text{A}\vdash v:\sigma \ \ \text{A}\vdash a:\tau}{\text{A}\vdash \text{MODARRAY}(\ s,\ v,\ a):\tau}$$

$$\Longleftrightarrow \exists s_i \in \mathbb{N}_{\bullet} : \sigma[s_1, ..., s_n] \preceq \tau \wedge \text{INT}[n] \preceq \rho \quad .$$

Bei Teil-Arrays ergibt sich

$$\text{MOD2} \ : \ \frac{\text{A}\vdash s:\rho \ \ \text{A}\vdash v:\sigma \ \ \text{A}\vdash a:\tau}{\text{A}\vdash \text{MODARRAY}(\ s,\ v,\ a):\tau}$$

$$\Longleftrightarrow \exists r_i, s_i \in \mathbb{N}_{\bullet} : \ \sigma'[s_1, ..., s_m] \preceq \sigma \wedge \text{INT}[n] \preceq \rho \quad .$$
$$\wedge \ \sigma'[r_1, ..., r_n, s_1, ..., s_m] \preceq \tau$$

3.3.4 Die WITH-Konstrukte – Array Comprehensions in SAC

Wie bereits im Abschnitt 2.1 erläutert, bieten Array-Comprehension-Konstrukte die Möglichkeit, Operationen auf elementweiser Basis für Teilmengen der zulässigen Indexmange eines Arrays zu spezifizieren. Da die Verwendung solcher Konstrukte in vielen numerischen Anwendungen eine zentrale Rolle spielt, muß bei deren Design für SAC sichergestellt sein, daß sie auf Arrays unabhängig von deren Dimensionalität angewendet werden können, um weiterhin eine dimensionsunabhängige Spezifikation von Algorithmen zu ermöglichen. Diese Konstrukte, in SAC als WITH-Konstrukte bezeichnet, bestehen aus drei Teilen (vergl. Abb. 3.8): Einem Generatorteil, einem Filterteil und dem eigentlichen Operationsteil. Generator- und Filterteil zusammen beschreiben eine Indexmenge für welche die im Operationsteil spezifizierte Operation ausgeführt werden soll.

Der entscheidende Unterschied zwischen den Array-Comprehension-Konstrukten in SAC und denen in anderen Sprachen wie z.B. SISAL oder NESL, deren Array-Darstellungen auf einer Schachtelung von eindimensionalen Arrays beruhen, liegt im Generatorteil. Um ein Array-Comprehension-Konstrukt für ein n-dimensionales Array zu spezifizieren, müssen die Indexgrenzen für jede einzelne Achse spezifiziert

$$
\begin{array}{lcl}
\mathit{WithExpr} & \Rightarrow & \text{WITH} \ \texttt{(} \ \mathit{Generator} \ [\ \texttt{,} \ \mathit{Filter} \ ]^{*} \ \texttt{)} \ \mathit{Operation} \\
\mathit{Generator} & \Rightarrow & \mathit{Expr} \ \texttt{<=} \ \mathit{Id} \ \texttt{<=} \ \mathit{Expr} \\
\mathit{Filter} & \Rightarrow & \mathit{Expr} \\
\mathit{Operation} & \Rightarrow & [\ \mathit{AssignBlock} \ ] \ \mathit{ConExpr} \\
\mathit{ConExpr} & \Rightarrow & \text{GENARRAY} \ \texttt{(} \ \mathit{ConstVec} \ \texttt{,} \ \mathit{Expr} \ \texttt{)} \\
 & | & \text{MODARRAY} \ \texttt{(} \ \mathit{Id} \ \texttt{,} \ \mathit{Expr} \ \texttt{,} \ \mathit{Expr} \ \texttt{)} \\
 & | & \text{FOLD} \ \texttt{(} \ \mathit{FoldFun} \ \texttt{,} \ \mathit{Expr} \ \texttt{,} \ \mathit{Expr} \ \texttt{)} \\
\mathit{FoldFun} & \Rightarrow & \mathit{FoldOp} \\
 & | & \mathit{Id} \\
\mathit{FoldOp} & \Rightarrow & \texttt{+} \ | \ \texttt{-} \ | \ \texttt{*} \ | \ \texttt{/}
\end{array}
$$

Abbildung 3.8: Die Syntax der WITH-Konstrukte in SAC.

werden. Dies erfolgt in Sprachen wie NESL durch eine n-fache Schachtelung von Array-Comprehension-Konstrukten und in Sprachen wie SISAL durch die explizite Angabe von n Indexvariablen. Keine dieser Lösungen erlaubt Anwendungen auf Arrays verschiedener Dimensionalität. In SAC jedoch ist dies mit Hilfe der Darstellung von Arrays durch Shape- und Datenvektor möglich. Die entscheidende Idee liegt darin, auch für n-dimensionale Arrays nur eine einzige Indexvariable zu verwenden, die als Indexvektor beim Traversieren von Arrays verwendet wird. Die Indexgrenzen werden dann in Form von Minimal- bzw. Maximal-Vektoren angegeben (siehe *Generator*-Ableitung in Abb. 3.8).

Der Filterteil gestattet, wie auch bei den Array-Comprehension-Konstrukten anderer Sprachen, die Einschränkung der Menge der Indizes durch die Formulierung von Prädikaten auf der Indexvariablen.

Für den Operationsteil bietet SAC drei verschiedene Grundoperationen (siehe *ConExpr* in Abb. 3.8).

GENARRAY($Shape$, Val) ermöglicht die Erzeugung großer Arrays, deren Werte von der jeweiligen Indexposition abhängen. Nicht vom Generator- und Filterteil

erfaßte Indizes werden dabei mit 0 bzw. FALSE initialisiert. So gilt beispielsweise

$$\begin{array}{l}\text{WITH}([1,1] <= \texttt{idx} <= [2,2])\\ \text{GENARRAY}([4,4],3)\end{array} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 3 & 3 & 0 \\ 0 & 3 & 3 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} ,$$

$$\begin{array}{l}\text{WITH}([1,1] <= \texttt{idx} <= [2,2])\\ \text{GENARRAY}([4,4,2],\texttt{idx})\end{array} = \begin{pmatrix} <0\,0> & <0\,0> & <0\,0> & <0\,0> \\ <0\,0> & <1\,1> & <1\,2> & <0\,0> \\ <0\,0> & <2\,1> & <2\,2> & <0\,0> \\ <0\,0> & <0\,0> & <0\,0> & <0\,0> \end{pmatrix} .$$

MODARRAY(Id, Val, $Array$) dient der Modifikation eines bereits existierenden Arrays. Die Funktionalität des Operators entspricht einer Projektion der primitiven MODARRAY-Operation auf sämtliche Indizes aus dem Generator- / Filterteil . Die exemplarische Betrachtung eines Arrays

$$\texttt{B} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} .$$

ergibt

$$\begin{array}{l}\text{WITH}([0,1] <= \texttt{idx} <= [2,2])\\ \text{MODARRAY}(\texttt{idx},42,\texttt{B})\end{array} = \begin{pmatrix} 1 & 42 & 42 & 4 \\ 5 & 42 & 42 & 8 \\ 9 & 42 & 42 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} ,$$

$$\begin{array}{l}\text{WITH}([1,0] <= \texttt{idx} <= [2,2])\\ \text{MODARRAY}(\texttt{idx},\text{PSI}(\texttt{idx},\texttt{B})+20,\texttt{B})\end{array} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 25 & 26 & 27 & 8 \\ 29 & 30 & 31 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} ,$$

$$\begin{array}{l}\text{WITH}([1,1] <= \texttt{idx} <= [2,2])\\ \text{MODARRAY}(\texttt{idx},\text{PSI}(\texttt{idx}-[1,0],\texttt{B}),\texttt{B})\end{array} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 2 & 3 & 8 \\ 9 & 6 & 7 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} .$$

FOLD($FoldOp$, $InitVal$, Val) erlaubt das „Zusammenfalten“ der Elemente eines Arrays auf einen Wert mittels einer binären Operation. Dabei kann es sich sowohl um eine der primitiven arithmetischen Funktionen $\{+,-,*,/\}$ als auch um eine benutzerdefinierte Funktion handeln. Der zweite Parameter dieser Operation gibt einen initialen Wert vor, in der Regel das neutrale Element bezüglich der Funktion. Auch hierzu einige Beispiele mit B wie oben bereits definiert und folgender Funktion min:

```
INT min( INT a, INT b)
{ IF( a<=b)
    min=a;
  ELSE
    min=b;
  RETURN(min);
}                          .
```

Dann gilt

$$\begin{matrix}\text{WITH}([0,0] <= \texttt{idx} <= [2,2]) \\ \text{FOLD}(+, 0, \text{PSI}(\texttt{idx}, \texttt{B}))\end{matrix} = 1+2+3+5+6+7+9+10+11 = 54 \quad ,$$

$$\begin{matrix}\text{WITH}([1,1] <= \texttt{idx} <= [2,2]) \\ \text{FOLD}(+, 0, 1)\end{matrix} = 1+1+1+1 = 4 \quad ,$$

$$\begin{matrix}\text{WITH}([1,1] <= \texttt{idx} <= [2,2]) \\ \text{FOLD}(\texttt{min}, 1000, \text{PSI}(\texttt{idx}, \texttt{B}))\end{matrix} = 6 \quad ,$$

$$\begin{matrix}\text{WITH}([1,1] <= \texttt{idx} <= [2,2]) \\ \text{FOLD}(\texttt{min}, 0, \text{PSI}(\texttt{idx}, \texttt{B}))\end{matrix} = 0 \quad .$$

Bei allen Varianten der WITH-Konstrukte kann vor dem Operationsteil ein optionaler Assignment-Block eingefügt werden. Er dient ausschließlich einer übersichtlicheren Notation für den Fall einer komplexen Berechnung der einzelnen Werte Val.

Um die Bedeutung der WITH-Konstrukte in SAC formal im Sinne der Def. 3.2.5 fassen zu können, wird zunächst ein Transformationsschema

$$\mathcal{TF}_W : \text{SAC} \rightarrow_{part} \mathcal{F}\text{UN}$$

definiert, das WITH-Konstrukte in äquivalente $\mathcal{F}$UN-Programme transformiert. Die einzelnen Transformationsregeln des Schemas werden im folgenden schrittweise vorgestellt. Anschließend werden die für eine Typinferenz auf WITH-Konstrukten benötigten Ableitungsregeln entwickelt.

Ein WITH-Konstrukt mit einer GENARRAY-Operation kann auf einfache Weise auf ein WITH-Konstrukt mit MODARRAY-Operation abgebildet werden. Dies erfolgt durch die Regel

$$\mathcal{TF}_W\begin{pmatrix}\text{WITH}(e_1\ \texttt{<=}\ Idx\ \texttt{<=}\ e_2\ , f_1, ..., f_n)\\ \{Ass;\}\\ \text{GENARRAY}(e_{shape}, e_{val})\end{pmatrix}$$

$$\mapsto \begin{array}{l}\text{LET}\\ \quad New\_A = \text{GENARRAY}(\ e_{shape}\texttt{,}\ \texttt{0})\\ \text{IN}\ \mathcal{TF}_W\begin{pmatrix}\text{WITH}(\ e_1\ \texttt{<=}\ Idx\ \texttt{<=}\ e_2\ ,\ f_1,\ ...,\ f_n)\\ \quad \{Ass;\}\\ \text{MODARRAY}(\ Idx,\ e_{val},\ New\_A)\end{pmatrix}\end{array},$$

wobei New_A für einen ansonsten im Programm unbenutzten Variablennamen steht.

Ein WITH-Konstrukt mit einer MODARRAY-Operation entspricht einer tail-endrekursiven Funktion, die sukzessive das im Operationsteil spezifizierte Array für alle durch den Generator- bzw. Filterteil charakterisierten Indizes modifiziert:

$$\mathcal{TF}_W\begin{pmatrix}\text{WITH}(e_1\ \texttt{<=}\ Idx\ \texttt{<=}\ e_2\ , f_1, ..., f_n)\\ \{Ass;\}\\ \text{MODARRAY}(\ Idx,\ e_{val},\ e_{array})\end{pmatrix}$$

$$\mapsto \begin{array}{l}\text{LETREC}\\ \quad loop = \lambda\ Array\ Idx\,.\\ \qquad \text{IF}\ (Idx\ >\ e_2)\\ \qquad \text{THEN}\ Array\\ \qquad \text{ELSE IF}\ (f_1\ \text{AND}\ \ldots\ \text{AND}\ f_n)\\ \qquad\quad \text{THEN LET}\\ \qquad\qquad \texttt{val} = \mathcal{TF}_K(\ \{Ass;\ \text{RETURN}(e_{val});\},\ \emptyset)\\ \qquad\qquad \text{IN}\ loop(\ \text{MODARRAY}(\ Idx,\ \texttt{val},\ Array),\\ \qquad\qquad\qquad next\_Idx(\ Idx))\\ \qquad\quad \text{ELSE}\ loop(\ Array,\ next\_Idx(\ Idx))\\ \text{IN}\ loop(\ e_{array}\texttt{,}\ e_1)\end{array},$$

wobei *loop* ein ansonsten unbenutzter Funktionsname sowie $Array$ ein unbenutzter Variablenname sind.

WITH-Konstrukte mit FOLD-Operationen können auf ähnliche Weise transformiert werden. Der wesentliche Unterschied besteht darin, daß anstelle einer sukzessiven Array-Modifikation bei FOLD-Operationen ein Wert akkumuliert wird:

$$\mathcal{TF}_W \begin{pmatrix} \text{WITH}(e_1 \texttt{ <= } Idx \texttt{ <= } e_2 \texttt{ , } f_1, ..., f_n) \\ \{Ass;\} \\ \text{FOLD}(\ Fun,\ e_{neutral},\ e_{val}) \end{pmatrix}$$

$$\mapsto \quad \begin{array}{l} \text{LETREC} \\ \quad loop \texttt{ = } \lambda\ Acc\ Idx\ . \\ \qquad \text{IF}\ (Idx\ >\ e_2) \\ \qquad \text{THEN}\ Acc \\ \qquad \text{ELSE IF}\ (f_1\ \text{AND}\ \ldots\ \text{AND}\ f_n) \\ \qquad\quad \text{THEN LET} \\ \qquad\qquad \texttt{val = } \mathcal{TF}_K(\ \{Ass;\ \text{RETURN}(e_{val});\},\ \emptyset) \\ \qquad\qquad \text{IN}\ loop(\ Fun(\ Acc,\ \texttt{val}),\ next\_Idx(\ Idx)) \\ \qquad\quad \text{ELSE}\ loop(\ Acc,\ next\_Idx(\ Idx)) \\ \text{IN}\ loop(\ e_{neutral},\ e_1) \end{array} ,$$

wobei *loop* ebenso für einen ansonsten unbenutzten Funktionsnamen und *Acc* für einen neuen Variablennamen steht.

Die Typinferenzregeln für WITH-Konstrukte lassen sich auf die Typinferenzregeln der entsprechenden (primitiven) Funktionen zurückführen. Zunächst sollen die WITH-Konstruke mit GENARRAY-Operation betrachtet werden:

$$\text{WITH1}\ :\ \frac{\begin{array}{c} A\vdash e_1:\tau_1 \quad A\vdash e_2:\tau_2 \quad A\vdash e_{shape}:\tau_3 \quad A\{v:\tau\}\vdash f_i:\text{BOOL} \\ A\{v:\tau\}\vdash \{Ass;\ \text{RETURN}(e_{val});\}:\rho \\ A\{dummy\_var:\rho\}\vdash \text{GENARRAY}(e_{shape},\ dummy\_var):\sigma \end{array}}{A\vdash \text{WITH}(e_1 \le v \le e_2, f_1, \ldots, f_n)\{Ass;\}\ \text{GENARRAY}(e_{shape},\ e_{val}):\sigma}$$

$$\Longleftrightarrow \exists \tau \in \mathcal{T}_{\text{SAC}} : \tau = \min(\ \tau_1, \min(\ \tau_2, \tau_3)) = \min(\ \min(\ \tau_1, \tau_2), \tau_3)$$

Diese Regel weist zwei Besonderheiten auf. Zum einen ergibt sich der Typ der Generatorvariablen v nicht ausschließlich aus dem Maximimum- bzw. Minimum-Ausdruck (e_1 bzw. e_2), sondern ebenfalls aus dem den Shape-Vektor spezifizierenden Ausdruck (e_{shape}). Die Forderung nach der Existenz eines solchen Typs τ stellt nicht nur eine Kompatibilität der Typen dieser drei Ausdrücke sicher, sondern inferiert zugleich einen möglichst speziellen Typ für die Generatorvariable. Zum anderen kann die Typinferenz durch das Einführen einer künstlichen Variablen *dummy_var* auf die Typinferenz eines Funktionsrumpfes sowie einer Anwendung der primitiven Funktion GENARRAY zurückgeführt werden.

Die Typinferenz eines WITH-Konstruktes mit MODARRAY-Operation gleicht im wesentlichen dem obigen Schema. Der einzige Unterschied besteht darin, daß sich der Typ der Generatorvariablen ausschließlich aus dem Maximimum- und dem Minimum-Ausdruck ergibt:

$$\textsc{With2} \;:\; \frac{\begin{array}{c}\mathrm{A}\vdash e_1:\tau_1 \;\; \mathrm{A}\vdash e_2:\tau_2 \;\; \mathrm{A}\{v:\tau\}\vdash f_i:\textsc{bool} \\ \mathrm{A}\{v:\tau\}\vdash \{Ass;\ \textsc{return}(e_{val});\}:\rho \\ \mathrm{A}\{v:\tau,\ dummy\_var:\rho\}\vdash \textsc{modarray}(\ v, dummy\_var, e_{array}):\sigma\end{array}}{\mathrm{A}\vdash \textsc{with}(e_1\le v\le e_2, f_1,\ldots,f_n)\{Ass;\}\ \textsc{modarray}(\ v, e_{val}, e_{array}):\sigma}$$

$$\Longleftrightarrow \exists \tau\in\mathcal{T}_{\mathrm{SAC}} : \tau = \min(\ \tau_1, \tau_2)$$

Die Regel für WITH-Konstrukte mit FOLD-Operation schließlich basiert auf dem für die Faltungs-Funktion inferierbaren Typ, entspricht ansonsten jedoch den obigen Regeln:

$$\textsc{With3} \;:\; \frac{\begin{array}{c}\mathrm{A}\vdash e_1:\tau_1 \;\; \mathrm{A}\vdash e_2:\tau_2 \;\; \mathrm{A}\{v:\tau\}\vdash f_i:\textsc{bool} \\ \mathrm{A}\{v:\tau\}\vdash \{Ass;\ \textsc{return}(e_{val});\}:\sigma \\ \mathrm{A}\vdash e_{identity}:\sigma \;\; \mathrm{A}\vdash fun:\sigma\times\sigma\to\sigma\end{array}}{\mathrm{A}\vdash \textsc{with}(e_1\le v\le e_2, f_1,\ldots,f_n)\{Ass;\}\ \textsc{fold}(\ fun, e_{identity}, e_{val}):\sigma}$$

$$\Longleftrightarrow \exists \tau\in\mathcal{T}_{\mathrm{SAC}} : \tau = \min(\ \tau_1, \tau_2)$$

3.4 Das Modulsystem von SAC

Vor dem Hintergrund einer funktionalen Programmiersprache läßt sich ein Modul als eine Zusammenfassung von Funktions- und Typdefinitionen zu einer syntaktischen Einheit verstehen. Durch geeigenete Export- bzw. Import-Mechanismen können aus solchen Modulen dann neue Programme bzw. Module definiert werden. Das erlaubt nicht nur einen modularen Programmaufbau, sondern erhöht auch die Wiederverwendbarkeit einzelner Programmkomponenten. Deshalb ist für den Entwurf und die Realisierung großer Programmsysteme ein komfortables Modulsystem unerläßlich.

Das Modulsystem von SAC umfaßt alle wesentlichen Merkmale moderner Modulsysteme wie sie in MODULA, SISAL, HASKELL oder CLEAN zu finden sind:

- jedes Modul läßt sich separat kompilieren;
- es können auch Module benutzt werden, deren Implementierung für den Benutzer unzugänglich ist („information hiding“). Im Gegensatz zu den meisten anderen Systemen können in SAC auch für ein „Inlining“ vorgesehene Funktionen für den Benutzer transparent bleiben und müssen nicht als eine Art „Makros“ nach außen hin bekannt gemacht werden;
- es gibt Typen, deren genaue Deklaration nur innerhalb der Modulimplementierung bekannt sind („opaque types“);
- der Compiler identifiziert selbsttätig die für den Binde-Vorgang benötigten Objektdateien;

- es lassen sich sowohl gezielt einzelne Symbole importieren („explicit import"), als auch ganze Import-Hierarchien („implicit import");
- wechselseitige Importe sind zulässig; sie werden mittels Fixpunktbildung durch den Compiler korrekt gehandhabt;
- Namenskonflikte zwischen Funktionen gleichen Namens in verschiedenen Modulen werden durch streng modulspezifische Namensbereiche aufgelöst.

Darüber hinaus dient das Modulsystem in SAC als Grundlage für das im nächsten Abschnitt vorgestellte Klassenkonzept in SAC. Es ermöglicht eine Integration von zustandsbehafteten Objekten sowie zugehörigen Modifikationsfunktionen, ohne Konflikte mit der funktionalen Semantik von SAC zu erzeugen.

Schließlich ermöglichen die Module in SAC in Kombination mit den Klassen eine Schnittstelle zu nicht in SAC geschriebenen Programmteilen (siehe Abschnitt 3.6).

Ein SAC-Modul besteht aus zwei Teilen: der Moduldeklaration und der Modulimplementierung. Die Modulimplementierung enthält die vollständige Definition der vom Modul exportierten Symbole (Typen / Funktionen), sowie eventuell noch weitere, lokal benötigte Symbole. Abb. 3.9 stellt sowohl die syntaktischen Ergänzungen der in Abb. 3.2 eingeführten Syntax von SAC-Programmen als auch die Syntax von Modulimplementierungen dar.

$$
\begin{array}{lcl}
\mathit{Program} & \Rightarrow & \left[\, \mathit{ImportBlock} \,\right]^{*}\ \mathit{Definitions}\ \mathit{Main} \\
\mathit{ModuleImp} & \Rightarrow & \textsc{module}\ \mathit{Id}\ :\ \left[\, \mathit{ImportBlock} \,\right]^{*}\ \mathit{Definitions} \\
\mathit{ImportBlock} & \Rightarrow & \textsc{import}\ \mathit{Id}\ :\ \textsc{all}\ ; \\
 & | & \textsc{import}\ \mathit{Id}\ :\ \mathit{ImportList} \\
\mathit{ImportList} & \Rightarrow & \{\ \left[\, \mathit{ITypeImp} \,\right] \left[\, \mathit{ETypeImp} \,\right] \left[\, \mathit{FunImp} \,\right]\ \} \\
\mathit{ITypeImp} & \Rightarrow & \textsc{implicit types}\ :\ \mathit{Id} \left[\, ,\, \mathit{Id} \,\right]^{*}\ ; \\
\mathit{ETypeImp} & \Rightarrow & \textsc{explicit types}\ :\ \mathit{Id} \left[\, ,\, \mathit{Id} \,\right]^{*}\ ; \\
\mathit{FunImp} & \Rightarrow & \textsc{functions}\ :\ \mathit{Id} \left[\, ,\, \mathit{Id} \,\right]^{*}\ ; \\
\mathit{Definitions} & \Rightarrow & \left[\, \mathit{TypeDef} \,\right]^{*} \left[\, \mathit{FunDef} \,\right]^{*} \\
\mathit{Main} & \Rightarrow & \textsc{int main}\ (\ )\ \mathit{ExprBlock}
\end{array}
$$

Abbildung 3.9: Die Syntax von Modulimplementierungen in SAC.

Der Unterschied zu einem SAC-Programm besteht im wesentlichen darin, daß Modulimplementierungen keine MAIN-Funktion enthalten (vergl. Abb. 3.9). Sowohl SAC-Programme als auch Modulimplementierungen können wiederum andere Module importieren. Solche import-Anweisungen können entweder implizit sämtliche Symbole eines anderen Moduls durch Verwendung des Schlüsselwortes ALL importieren oder aber explizit einige zu importierende Symbole selektieren. Dabei wird zwischen drei verschiedenen Arten von Symbolen unterschieden: Funktionen, expliziten Typen und impliziten Typen. Während bei expliziten Typen die vollständige Typdeklaration importiert wird und somit auch Daten-Objekte eines solchen Typs konkret erzeugt werden können, ist bei impliziten Typen nur der Name des Typs bekannt. Um Objekte eines impliziten Typs zu erzeugen, bedarf es daher immer einer entsprechenden Generatorfunktion innerhalb des jeweils importierten Moduls.

Moduldeklarationen stellen die Schnittstelle der Module für den Gebrauch durch andere SAC-Module/Programme dar; sie spezifizieren die von einem Modul exportierten Symbole sowie ggf. deren Beschaffenheit. Auch hier wird grundsätzlich zwischen Funktionen, expliziten Typen und impliziten Typen unterschieden. Um ohne Wissen über die Modulimplementierung eindeutig entscheiden zu können, in welchem Modul welches Symbol definiert ist, werden die zu exportierenden Symbole nach Herkunft getrennt exportiert (vergl. Abb. 3.10). Symbole, die in der zu-

$$
\begin{array}{lcl}
\mathit{ModuleDec} & \Rightarrow & \mathit{ModuleHeader}\ \left[\ \mathit{ImportBlock}\ \right]^{*}\ \text{OWN}:\ \mathit{Declarations} \\
\mathit{ModuleHeader} & \Rightarrow & \text{MODULEDEC}\ \left[\ \text{EXTERNAL}\ \right]\ \mathit{Id}\ : \\
\mathit{Declarations} & \Rightarrow & \{\ \left[\ \mathit{ITypeDec}\ \right]\left[\ \mathit{ETypeDec}\ \right]\left[\ \mathit{FunDec}\ \right]\ \} \\
\mathit{ITypeDec} & \Rightarrow & \text{IMPLICIT TYPES}\ :\left[\ \mathit{Id}\ ;\ \right]^{*} \\
\mathit{ETypeDec} & \Rightarrow & \text{EXPLICIT TYPES}\ :\left[\ \mathit{Id}\ =\ \mathit{Type}\ ;\ \right]^{*} \\
\mathit{FunDec} & \Rightarrow & \text{FUNCTIONS}\ :\left[\ \mathit{OneFunDec}\ \right]^{*}
\end{array}
$$

Abbildung 3.10: Die Syntax von Moduldeklarationen in SAC.

gehörigen Modulimplementierung definiert werden, sind als OWN zu kennzeichnen, während von anderen Modulen importierte Symbole durch entsprechende Import-Anweisungen als solche kenntlich gemacht werden.

Dabei ist es nicht erforderlich, als Herkunftsmodul dasjenige Modul zu benennen, das dieses Symbol definiert, sondern es genügt, ein Modul zu benennen, welches dieses Symbol exportiert. Da der Bereich der eigenen Symbole auch leer sein darf, ist es in SAC möglich, mehrere Module einfach mittels einer Moduldeklaration zu-

sammenzufassen, ohne eine Modulimplementierung zu erstellen.

3.5 Zustände und Zustandsmodifikationen in SAC

In C lassen sich aufgrund der imperativen Semantik auf direkte Weise Zustände über (globale) Variablen und Zustandsmodifikationen über Seiteneffekte auf diesen Variablen modellieren. Eine solche Modellierung von Zuständen bzw. Zustandsmodifikationen bietet zweierlei Vorteile: Zum einen erlaubt sie eine kurze, prägnante Spezifikation von Zustandstransformationsfolgen auf globalen Objekten, wie sie zum Beispiel für Input/Output-Zwecke sehr vorteilhaft ist; zum anderen gestatten sie eine laufzeiteffiziente Abbildung auf die ausführende Maschine. Eine direkte Übernahme dieser Modellierung nach SAC ist aufgrund der funktionalen Semantik nicht möglich. Statt dessen bedarf es der Integration einer der in Abschnitt 2.2 beschriebenen Sequentialisierungsmechanismen.

Ausgehend von einem auf Uniqueness-Typen basierendem Ansatz ist es dennoch möglich, die Vorteile von C bezüglich der Spezifikation von Zuständen und Zustandsmodifikationen vollständig nach SAC zu übernehmen. Dazu werden Uniqueness-Typen in Form sog. Klassen eingeführt. Eine Datenstruktur eines solchen Typs bezeichnen wir als Instanz der Klasse bzw. als Objekt. Durch die über die Uniqueness-Typen erzwungene Uniqueness-Eigenschaft solcher Objekte ist sichergestellt, daß alle Funktionen, die auf ein Objekt angewendet werden (können), dieses nicht kopieren, sondern ausschließlich konsumieren oder modifizieren (propagieren). Striktheit aller Funktionen in bezug auf Parameter mit Uniqueness-Typ vorausgesetzt, stellt dies eine sequentielle Abfolge solcher Funktionen in bezug auf ein Objekt sicher. Deshalb können Objekte als zustandsbehaftete Datenstrukturen betrachtet werden.

Um Zustandsmodifikationen genauso konzise wie in C spezifizieren zu können, werden außerdem globale Objekte und ein sog. CALL-BY-REFERENCE-Mechanismus zur Argumentübergabe eingeführt. Sie erlauben es dem Programmierer, konzeptuell benötigte Parameter in der Programmspezifikation wegzulassen, da es mittels Inferenz-Algorithmen möglich ist, diese durch den Compiler einzufügen. Sie können also als syntaktische Vereinfachungen angesehen werden und führen deshalb nicht zu Problemen mit der funktionalen Semantik.

3.5.1 Klassen und Objekte in SAC

Wie bereits erwähnt, basieren die zustandsbehafteten Datenstrukturen in SAC auf Uniqueness-Typen wie sie für CLEAN entwickelt wurden. In CLEAN können Uniqueness-Typen vom Programmierer überall dort benutzt werden, wo Typannotationen erlaubt sind[2]. Dies gestattet ein großes Maß an Flexibilität, da die Uniqueness-Eigenschaft nur an den Stellen gefordert werden muß, wo Zustände modifiziert werden

[2] In CLEAN1.0 werden Uniqueness-Typen sogar inferiert [BS95]

sollen. Soll lesend zugegriffen werden, so kann auf die Uniqueness-Eigenschaft verzichtet werden. Da jedoch Uniqueness-Typen nur als Mittel zur Integration von Zuständen in die funktionale Welt dienen sollten, bietet SAC auf der Seite der Hochsprache keine Uniqueness-Typen. Statt dessen wird in SAC zwischen „normalen“ und zustandsbehafteten Datenstrukturen unterschieden.

Um eine Separierung zustandsbehafteter Datenstrukturen sowie der zugehörigen Modifikations-Funktionen von dem Rest eines Programmes zu erzwingen, erfolgt ihre Definition in besonderen Modulen, den sog. Klassen. Sie enthalten einen ausgezeichneten impliziten Typ, der per definitionem dem Namen der Klasse entspricht und ein Uniqueness-Typ ist. Die Ergänzungen der Syntax von SAC um Klassen-Deklarationen bzw. -Implementierungen sind in Abb. 3.11 dargestellt. Außer daß

$$
\begin{array}{lcl}
\mathit{ClassDec} & \Rightarrow & \mathit{ClassHeader}\ \left[\ \mathit{ImportBlock}\ \right]^{*}\ \text{OWN}:\ \mathit{Declarations} \\
\mathit{ClassHeader} & \Rightarrow & \text{CLASSDEC}\ \left[\ \text{EXTERNAL}\ \right]\ \mathit{Id}\ : \\
\mathit{ClassImp} & \Rightarrow & \text{CLASS}\ \mathit{Id}:\ \left[\ \mathit{ImportBlock}\ \right]^{*}\ \text{CLASSTYPE}\ \mathit{Type}\ ;\ \mathit{Definitions}
\end{array}
$$

Abbildung 3.11: Die Syntax von Klassen in SAC.

Klassen durch spezielle Schlüsselwörter (CLASSDEC bzw. CLASSIMP) eingeleitet werden, unterscheiden sie sich von Modulen nur dadurch, daß sie immer einen impliziten Typ mit einem der Klasse identischen Namen exportieren. Er wird durch eine spezielle Klassentypdeklaration in der Klassenimplementierung definiert (siehe Abb. 3.11).

Um eine Klassenimplementierung in SAC spezifizieren zu können, bedarf es der Möglichkeit, eine „normale“ Datenstruktur in eine zustandsbehaftete und zurück transformieren zu können. Dies entspricht einer Konversionsmöglichkeit zwischen Uniqueness-Typen und anderen Typen. Dazu werden für jede Klasse *class*, die über eine Deklaration CLASSTYPE *type*; definiert ist, die generischen Konstruktor-/Destruktor-Funktionen

$$
\begin{array}{lll}
\mathit{class} & \text{TO\_}\mathit{class}(\ \mathit{type}\ \text{a}) & \text{und} \\
\mathit{type} & \text{FROM\_}\mathit{class}(\ \mathit{class}\ \text{a}) &
\end{array}
$$

eingeführt.

Durch den Verzicht auf eine Einführung expliziter Uniqueness-Typen kann auch auf deren formale Einführung in das Typsystem von SAC verzichtet werden. Statt dessen wird die Verwendung von Objekten dahingehend restringiert, daß für jedes Teilprogramm R und jedes Objekt a $\mathit{Refs}\,(a,\ R) \leq 1$ mit *Refs* aus Def. 3.1.5 gelten muß. Die einzige Erweiterung des Typsystems ist damit die Ergänzung der Typinfe-

renzregeln um Regeln für Anwendungen der generischen Funktionen TO_*class* und FROM_*class*:

$$\text{ToClass} \quad : \quad \frac{\text{A} \vdash e : \sigma}{\text{A} \vdash \text{TO\_}class(e) : <class, \tau>} \quad \Longleftrightarrow \quad Basetype(\sigma) = \tau \quad ,$$

$$\text{FromClass} \quad : \quad \frac{\text{A} \vdash e : <class, \tau>}{\text{A} \vdash \text{FROM\_}class(e) : \sigma} \quad \Longleftrightarrow \quad Basetype(\sigma) = \tau \quad .$$

3.5.2 Die Modifikation von Objekten

Mit den bisher eingeführten syntaktischen Konstrukten lassen sich Objektmodifikationen wie auch in CLEAN nur durch explizites Zurückgeben des modifizierten Zustandes realisieren. Sollen Folgen von Modifikationen auf ein und dem selben Objekt ausgeführt werden, so führt dies zu Programmausschnitten folgender Art:

```
state, erg1= modify_fun1( state, ...);
state, erg2= modify_fun2( state, ...);
⋮                                                    (3.4)
state, ergn= modify_funn( state, ...);
```

In vielen Anwendungsfällen ist der Programmierer jedoch gar nicht an allen Zuständen interessiert, sondern nur an einigen Zwischenzuständen oder sogar nur an der Abfolge der Modifikationen, wie es z.B. bei I/O-Operationen in der Regel der Fall ist. Für diesen Fall bietet C die Möglichkeit, Seiteneffekte zu erzeugen. Damit ließe sich obige Folge von Modifikationen erheblich übersichtlicher formulieren:

```
erg1= modify_fun1( state, ...);
erg2= modify_fun2( state, ...);
⋮                                                    (3.5)
ergn= modify_funn( state, ...);
```

Neben den notationellen Vorteilen hat dies auch Vorteile bei der Compilation, da bei allen Funktionsaufrufen auf die explizite Rückgabe des modifizierten Zustandes verzichtet werden kann.

Um eine solche Schreibweise auch in SAC nutzen zu können, bedarf es eines Mechanismus, mit dem solche Seiteneffekte gezielt eliminiert werden, um die unterliegende funktionale Semantik aufrecht erhalten zu können. Dies erfolgt in SAC mittels sog. CALL-BY-REFERENCE-Parameter. Wird bei einer Funktionsdeklaration

in SAC ein Parameter mit einem '&'-Zeichen als CALL-BY-REFERENCE-Parameter gekennzeichnet, so bedeutet dies, daß alle Funktionsanwendungen der Form (3.5) als notationelle Abkürzungen für ein Programmsegment der Form (3.4) stehen.

1. Für jede Funktionsanwendung der Form $r_1, ..., r_m = fun(e_1, ..., e_n);$ mit $e_i = fun2(a_1, ..., a_k)$ und für alle $1 \leq j < i$ enthält e_j keine Funktionsanwendung:

   - Generiere eine bisher unbenutzte Variable tmp_x mit dem selben Typ wie e_i.
   - Ersetze $r_1, ..., r_m = fun(e_1, ..., e_n);$ durch
     $tmp_x = fun2(a_1, ..., a_k);$
     $r_1, ..., r_m = fun(e_1, ..., e_{i-1}, tmp_x, e_{i+1}, ..., e_n);$
   - Wende dieses Verfahren solange rekursiv an, bis keine geschachtelten Funktionsanwendungen mehr vorkommen.

2. Für jede Funktionsanwendung der Form $r_1, ..., r_m = fun(e_1, ..., e_n);$ für die es eine Funktionsdefinition $rty_1, ..., rty_m\ fun(arg_1, ..., arg_n)\{...\}$ mit mindestens einem call-by-reference Parameter arg_i gibt:

   - Für alle $1 \leq i \leq n$ mit $arg_i = ty_i\ \&id_i$:
     - Ersetze $r_1, ..., r_m = fun(e_1, ..., e_n);$ durch
       $r_1, ..., r_m, e_i = fun(e_1, ..., e_n);$, falls e_i eine Variable ist;
       ansonsten generiere eine bisher unbenutzte Variable tmp_x mit demselben Typ wie e_i und ersetze $r_1, ..., r_m = fun(e_1, ..., e_n);$ durch
       $tmp_x = e_i;$
       $r_1, ..., r_m, tmp_x = fun(e_1, ..., e_{i-1}, tmp_x, e_{i+1}, ..., e_n);$.

3. Für jede Funktionsdefinition der Form
   $rty_1, ..., rty_n\ fun(arg_1, ..., arg_m)\{...$ RETURN$(e_1, ..., e_n);\}$
   mit $arg_i = ty_i\ \&id_i$ und für alle $1 \leq j < i : a_j = ty_j\ id_j$:

   - Ersetze $rty_1, ..., rty_n\ fun(arg_1, ..., arg_{i-1}, ty_i\ id_i\&, arg_{i+1}, ..., arg_m)$
     durch $rty_1, ..., rty_n, ty_i\ fun(arg_1, ..., arg_{i-1}, ty_i\ id_i, arg_{i+1}, ..., arg_m)$.
   - Ersetze RETURN$(e_1, ..., e_n);$ durch RETURN$(e_1, ..., e_n, a_i);$.
   - Wende dieses Verfahren solange rekursiv an, bis alle call-by-reference Parameter in $a_1, ..., a_m$ eliminiert sind.

Abbildung 3.12: Das Transformationsschema für CALL-BY-REFERENCE-Parameter.

Abb. 3.12 beschreibt ein Transformationsschema, das SAC-Programme, die CALL-BY-REFERENCE-Parameter benutzen, in bedeutungsgleiche SAC-Programme abbildet, die keine CALL-BY-REFERENCE-Parameter mehr enthalten. Dadurch wird auch für CALL-BY-REFERENCE-Parameter enthaltende SAC-Programme eine seiteneffektfreie funktionale Semantik definiert, obwohl aus Sicht der äquivalenten Semantik von C an dieser Stelle Seiteneffekte erzeugt werden.

Die Transformation besteht aus drei Phasen. Da sie ggf. eine ERweiterung um zusätzliche Rückgabewerte erfordert und Funktionen mit mehreren Rückgabewerten in SAC nicht in Argumentposition auftreten dürfen, werden geschachtelte Funktionsanwendungen in der ersten Phase eliminiert. Diese auf den ersten Blick rein technische Maßnahme hat jedoch auch eine Auswirkung auf die Semantik. Während in C die Auswertungsreihenfolge der Argumente einer Funktion und damit die Abfolge darin eventuell enthaltener I/O-Operationen in der Sprachdefinition [KR90] nicht festgelegt und damit implementierungsabhängig ist, stellt diese Transformation in SAC eine innermost-leftmost Sequentialisierung solcher Operationen sicher.

In der zweiten Phase werden alle Funktionsanwendungen auf Argumente, die als CALL-BY-REFERENCE-Parameter übergeben werden sollen, um die zusätzlich benötigten Rückgabewerte ergänzt. Sie werden Variablen zugewiesen, die denselben Namen wie die Variablen in der jeweiligen Argumentposition haben. Müssen bei einer Funktionsanwendung mehrere zusätzliche Rückgabewerte ergänzt werden, so erfolgt dies von links nach rechts.

Die dritte Phase eliminiert schließlich die CALL-BY-REFERENCE-Parameter aus den Funktionsdefinitionen. Dazu werden sämtliche CALL-BY-REFERENCE-Parameter von links nach rechts in die jeweilige RETURN-Anweisung übernommen und die korrespondierenden Resultattypen in der Funktionsdeklaration ergänzt.

3.5.3 Globale Objekte

Die Verwendung von Objekten in SAC ist nur dann sinnvoll, wenn es sich entweder um große Datenstrukturen handelt, bei denen der Programmierer aus Effizienzgründen ein Kopieren verhindern will, oder aber wenn diese Datenstrukturen den Zustand von System-Komponenten (z.B. des Terminals oder des kompletten Filesystems) repräsentieren, bei denen das Erzeugen einer Kopie nicht realisierbar ist. Für die meisten dieser Anwendungen werden die Objekte über die gesamte oder zumindest fast die gesamte Laufzeit des Programmes benötigt. Wie bei allen Objekten in SAC wird auch die Erzeugung bzw. Zerstörung solch „globaler Objekte“ aufgrund der Uniqueness-Eigenschaft vom Programmierer direkt kontrolliert. Dies erfolgt mittels spezieller Funktionen der entsprechenden Klasse, da die Definition der Objektstruktur nur innerhalb der Klassenimplementierung verfügbar ist. Als Konsequenz müssen die globalen Objekte am Anfang des Programmes erzeugt und dann überall dort im Programm, wo sie benötigt werden, mittels Parameterübergabe verfügbar gemacht werden. Da aber der Programmierer ohnehin die Existenz der

globalen Objekte selbst kontrolliert, führen diese „zusätzlichen“ Parameter gerade in den Fällen, wo der Programmierer weniger am eigentlichen Zustand als an den Zustandsmodifikationen interessiert ist, nicht nur zu einen erhöhten Laufzeitaufwand, sondern erschweren auch die Lesbarkeit von Programmen.

Auch für dieses Problem bieten imperative Sprachen wie C eine elegante Lösung. Es können globale Variablen vereinbart werden, auf die von jeder Stelle des Programmes direkt zugegriffen werden kann. Ähnlich wie mit den CALL-BY-REFERENCE-Parametern lassen sich nun auch solche globalen Objekte als sytaktische Vereinfachungen in SAC integrieren.

Globale Objekte können in SAC mittels des Schlüsselwortes OBJDEF defininiert werden(siehe Abb. 3.13). Um auch für globale Objekte eine modul-/klassen-über-

$$
\begin{array}{lcl}
ObjDef & \Rightarrow & \text{OBJDEF } Type\ Id\ =\ Expr\ ; \\
ImportList & \Rightarrow & \{\ [\ ITypeImp\ ]\ [\ ETypeImp\ ]\ [\ ObjImp\ ]\ [\ FunImp\ ]\ \} \\
ObjImp & \Rightarrow & \text{GLOBAL OBJECTS : } Id\ [\ ,\ Id\ ]^{*}\ ; \\
Declarations & \Rightarrow & \{\ [\ ITypeDec\ ]\ [\ ETypeDec\ ]\ [\ ObjDec\ ]\ [\ FunDec\ ]\ \} \\
ObjDec & \Rightarrow & \text{GLOBAL OBJECTS : } [\ Type\ Id\ ;\ ]^{*}
\end{array}
$$

Abbildung 3.13: Die Syntax von globalen Objekten in SAC.

greifende Verwendung zu ermöglichen, können diese genauso wie Funktionen oder Typen importiert (*ObjImp*) und exportiert (*ObjDec*) werden. Die entsprechenden syntaktischen Erweiterungen der *ImportList* bzw. der *Declarations* sind ebenfalls Abb. 3.13 zu entnehmen.

Ähnlich wie für die CALL-BY-REFERENCE-Parameter wird die Semantik der globalen Objekte in SAC mit Hilfe eines einfachen Transformationsschemas (siehe Abb. 3.14) definiert. Dieses Schema ersetzt die Objekt-Definitionen durch Zuweisungen im Hauptprogramm und inferiert die erforderlichen Ergänzungen der Parameterlisten von Funktionen, die globale Objekte entweder selbst modifizieren oder aber Funktionen aufrufen, die solches veranlassen.

Die Transformation erfolgt in drei Phasen. Zunächst werden für jede Funktion alle globalen Objekte bestimmt, die benötigt werden. Dabei werden nicht nur diejenigen globalen Objekte berücksichtigt, die in der jeweiligen Funktion direkt referenziert sind, sondern rekursiv auch alle die Objekte, die für den Aufruf anderer Funktionen innerhalb des Funktionsrumpfes benötigt werden. Mit Hilfe dieser Variablenmengen kann dann überprüft werden, ob alle benötigten globalen Objekte bereits übergeben werden. Fehlende Objekte werden als CALL-BY-REFERENCE-Parameter hinzugefügt.

1. Für jede Funktionsdefinition außer für MAIN:
   - Bestimme die Menge aller verwendeten globalen Objekte.
   - Vereinige diese Menge mit den Mengen der im Funktionsrumpf aufgerufenen Funktionen.
   - Wiederhole diesen Vorgang solange, bis ein Fixpunkt erreicht wird.
   - Füge alle benötigten globalen Objekte, die noch nicht in der Liste der Parameter der Funktion auftauchen, als CALL-BY-REFERENCE-Parameter hinzu.
2. Für jede Anwendung einer Funktion, die in der ersten Phase modifiziert wurde:
   - Füge die benötigten Objekte in der durch die erste Phase vorgegebenen Reihenfolge als zusätzliche Argumente ein.
3. Für jede Definition eines globalen Objektes der Form OBJDEF v=e;:
   - Füge v=e; zu Beginn der MAIN-Funktion ein.
   - Eliminiere die eigentliche Objekt-Definition.

Abbildung 3.14: Das Transformationsschema für globale Objekte.

Auf der Basis dieser hinzugefügten Parameter sorgt die zweite Phase für eine entsprechende Anpassung aller Funktionsanwendungen der so modifizierten Funktionen.

Da nach diesen beiden ersten Phasen sichergestellt ist, daß alle benötigten Objekte auch an die jeweilige Funktion übergeben werden, können schließlich in der dritten Phase alle Objekt-Definitionen entfernt werden. Stattdessen wird die Erzeugung der benötigten Objekte am Beginn des Hauptprogrammes eingefügt.

3.5.4 I/O in SAC

Für die Integration von I/O-Operationen in ein SAC-Programm ist eine Interaktion mit dem Betriebssystem notwendig. Da dies inhärent zustandsbehaftet ist, kann in SAC durch ein oder mehrere globale Objekte modelliert werden, die vor der eigentlichen Programmausführung bereits existieren und diese dann auch überleben. Deshalb gibt es in SAC sog. generische Objekte von einem ausgezeichneten Typ namens WORLD, die als CALL-BY-REFERENCE-Parameter an die MAIN-Funktion übergeben werden. Sie sind in einer Klasse WORLD definiert und werden bei der Anwendung des in Abschnitt 3.5.3 vorgestellten Transformationsschemas gesondert behandelt. Da sie als CALL-BY-REFERENCE-Parameter an das gesamte Programm übergeben

werden, kann die Objektdefinition eliminiert werden, ohne daß eine Konstruktorfunktion zu Beginn der MAIN-Funktion eingefügt werden muß. Bei der Modellierung der Umgebung durch solche generischen Objekte muß darauf geachtet werden, daß die einzelnen Objekte vollständig disjunkte Teile der Umgebung repräsentieren, da es sonst zu Seiteneffekten und damit zu Nichtdeterminismen bei einer nebenläufigen Ausführung kommen kann.

Die eigentlichen I/O Operationen erfolgen über Kanäle, die vom Betriebssystem (Pipes, Files, etc.) oder darauf aufbauenden Programmsystemen wie z.B. einem Window-Manager (Event-Queues, etc.) verwaltet werden. Um nebenläufige Operationen auf voneinander unabhängigen Kanälen zu ermöglichen, können diese wiederum in SAC als Objekte modelliert werden. Für Kanäle, die permanent vorhanden sind, wie z.B. `stdin` oder `stdout`, kann dies durch globale Objekte erfolgen, während temporäre Objekte wie z.B. geöffnete Files oder Fenster durch die Anwendung entsprechender Konstruktor-/Destruktor-Funktionen erst erzeugt und später wieder gelöscht werden müssen. Da das Einrichten bzw. Löschen solcher Kanäle eine Interaktion mit der Umgebung darstellt, modifizieren die zugehörigen Konstruktor- bzw. Destruktor-Funktionen wiederum generische Objekte, die sie deshalb, zumindest in der rein funktionalen Darstellung, als CALL-BY-REFERENCE-Parameter erhalten müssen.

Da es in SAC möglich ist, auf eine explizite Übergabe von globalen Objekten zu verzichten, ist es nicht erforderlich, die generischen WORLD-Zustände bei der Erzeugung von Kanälen auf Hochsprachenebene explizit zu übergeben. Dies ermöglicht syntaktisch eine direkte Verwendung der für C zur Verfügung gestellten Betriebssystemsfunktionen wie `printf` oder `fopen`. Ihre Einbindung beschränkt sich dadurch im wesentlichen auf das Spezifizieren adäquater Klassendeklarationen. Da in SAC über das Modul- bzw. Klassen-System auch eine Schnittstelle zu anderen Programmiersprachen hergestellt wird (näheres zu diesem Thema siehe Abschnitt 3.6), ist es schließlich sogar möglich, die I/O-Operationen direkt auf die entsprechenden C-Routinen abzubilden. Eine detaillierte Beschreibung der Integration des I/O-Systems von C in SAC findet sich in [Gre96]. An dieser Stelle soll lediglich anhand einiger kleiner Beispieldeklarationen (siehe Abb. 3.15) gezeigt werden, wie sich die Standard-I/O-Bibliothek von C in SAC integrieren läßt.

Diese Bibliothek stellt die Verbindung zwischen einem Programm und dem UNIX-Filesystem dar. Um das Filesystem als Ganzes zu modellieren, enthält die Klasse WORLD also ein Objekt `FileSys`. Aus diesem Filesystem lassen sich mittels der Funktionen `fopen` bzw. `fclose` einzelne Files separieren bzw. reintegrieren. Die jeweiligen EFFECT-Pragmas (siehe Abb. 3.15) zeigen dem Compiler an, daß diese Funktionen das Filesystem modifizieren. Sie sind erforderlich, da die Klasse `File` nicht in SAC implementiert ist und dieser Effekt für den Compiler sonst nicht inferierbar wäre. Dadurch, daß die eigentlichen Schreib-/Lese-Funktionen `fprintf`/`fscanf` diesen Effekt nicht annotiert bekommen, besteht keinerlei Datenabhängigkeit zwischen den Schreib-/Leseoperationen auf verschiedenen Files. Innerhalb eines Files wird diese

```
CLASSDEC WORLD:

OWN: {
  GLOBAL OBJECTS:
    WORLD FileSys;
}
```

```
CLASSDEC EXTERNAL File:

IMPORT WORLD:ALL;

OWN: {
  FUNCTIONS:
    File fopen( CHAR[] filename, CHAR[] type);   #PRAGMA EFFECT FileSys
    VOID fclose( File stream);                   #PRAGMA EFFECT FileSys
    VOID fprintf( File & stream, CHAR[] format, ...);
    ...  fscanf( File & stream, CHAR[] format);
}
```

```
CLASSDEC EXTERNAL TermFile:

IMPORT WORLD:ALL;

OWN: {
  GLOBAL OBJECTS:
    TermFile stdin;                              #PRAGMA EFFECT FileSys
    TermFile stdout;                             #PRAGMA EFFECT FileSys
    TermFile stderr;                             #PRAGMA EFFECT FileSys
  FUNCTIONS:
    VOID printf( CHAR[] format, ...);            #PRAGMA EFFECT FileSys
    VOID fprintf( TermFile & stream, CHAR[] format, ...);
                                                 #PRAGMA EFFECT FileSys
    ...  scanf( CHAR[] format);                  #PRAGMA EFFECT FileSys
    ...  fscanf( TermFile & stream, CHAR[] format);
                                                 #PRAGMA EFFECT FileSys
}
```

```
MODULDEC Stdio:

IMPORT File:ALL;
IMPORT TermFile:ALL;

OWN: { }
```

Abbildung 3.15: Auszüge der Integration von *stdio* in SAC.

jedoch durch die jeweils durch `fopen` erzeugten File-Objekte sichergestellt.

Mit diesem Ansatz läßt sich die Standard-I/O-Bibliothek von C fast vollständig integrieren. Hier fehlt einzig noch die Verfügbarkeit der drei standardmäßig geöffneten Files `stdin`, `stdout` und `stderr`, die direkt mit den Eingabe/Ausgabe-Kanälen

des Terminals korrespondieren. Die einfachste Möglichkeit, sie in SAC zu integrieren, wäre eine Abbildung auf globale Objekte vom Typ `File`. Das Problem bei dieser Lösung besteht darin, daß somit zwar die einzelnen Schreib- bzw. Leseoperationen untereinander sequentialisiert werden, nicht jedoch die Abfolge aller dieser Operationen. Um nicht gleich alle File-Zugriffe zu sequentialisieren, gibt es eine Klasse `TermFile`, die nur diese drei globalen Objekte enthält und durch Einführen der entsprechenden EFFECT-Pragmas für die File-modifizierenden Funktionen eine Synchronisation sicherstellt (siehe Abb. 3.15).

Um `stdio` als eine einzelne Struktur modellieren zu können, gibt es schließlich noch ein Modul `stdio`, das die beiden Klassen `TermFile` und `File` zu einer Struktur zusammenfaßt.

3.5.5 Ein Beispiel

In diesem Abschnitt soll anhand eines kleinen Beispiels gezeigt werden, wie ein SAC-Programm, das CALL-BY-REFERENCE-Parameter und globale Objekte für I/O benutzt, schrittweise von einer C-ähnlichen Darstellung in eine Darstellung transformiert wird, aus der sich die funktionale Bedeutung direkt ablesen läßt. Dies erfolgt mit Hilfe der beiden in den Abschnitten 3.5.2/3.5.3 spezifizierten Transformationsschemata. Als Grundlage für die zur Verfügung stehenden I/O-Primitiva dienen die in Abb. 3.15 dargestellten Klassendefinitionen.

Als Beispiel dient hier ein einfaches Hello-World-Programm. Es gibt mittels der Funktion `PrintHeader` zunächst "This is SAC !" und "Version 0.5" aus, um dann direkt durch einen `printf`-Aufruf im Hauptprogramm "Hello World!" auszugeben (siehe Abb. 3.16). Dabei ist zu bemerken, daß die Syntax fast vollkommen der Syntax

```
IMPORT Stdio:ALL;

VOID PrintHeader( TermFile & out)
{ printf( "This is SAC !\n");
  fprintf( out, "Version 0.5\n");
}

INT MAIN()
{ PrintHeader(stdout);
  printf("Hello World!\n\);
  ...
  RETURN(res);
}
```

Abbildung 3.16: Erweitertes Hello-World Programm in SAC.

von C entspricht; lediglich der Typ `TermFile` ist SAC-spezifisch.

Bevor das Transformationsschema $\mathcal{TF}_O$ zur Eliminierung der globalen Objekte angewandt werden kann, werden die benötigten Deklarationen der importier-

ten Klassen (`FileSys`, `stdout`, `printf` und `fprintf`) eingefügt[3]. Da `FileSys` vom Typ WORLD ist, erhält die MAIN-Funktion dieses Objekt als CALL-BY-REFERENCE-Parameter (vergl. Abschnitt 3.5.4), bevor die eigentliche Transformation $\mathcal{TF}_O$ gemäß Abb. 3.14 durchgeführt wird.

```
VOID printf( WORLD & FileSys, CHAR[] format, ...);

VOID fprintf( WORLD & FileSys, TermFile & stream, CHAR[] format, ...);

VOID PrintHeader( WORLD & FileSys, TermFile & out)
{ printf( FileSys, "This is SAC !\n");
  fprintf( FileSys, out, "Version 0.5\n");
}

INT MAIN( WORLD & FileSys)
{ stdout = CREATE_stdout( FileSys);
  PrintHeader( FileSys, stdout);
  printf( FileSys, "Hello World!\n");
  ...
  RETURN(res);
}
```

Abbildung 3.17: Hello-World Programm nach der Anwendung von $\mathcal{TF}_O$.

Das Ergebnis der Anwendung von $\mathcal{TF}_O$ auf das Beispiel ist in Abb. 3.17 dargestellt. Gemäß Regel 1 von $\mathcal{TF}_O$ ergibt sich für `printf`, `fprintf` und `PrintHeader` `FileSys` als zusätzlicher CALL-BY-REFERENCE-Parameter. Während dies bei `PrintHeader` direkt aus dem Programmtext zu entnehmen ist, sorgen bei den importierten Funktionen die entsprechenden EFFECT-Pragmas dafür. Die Anpassung der zugehörigen Funktionsanwendungen erfolgt durch Regel 2 von $\mathcal{TF}_O$. Schließlich wird die aus `TermFile` importierte Objekt-Deklaration für `stdout` gemäß Regel 3 von $\mathcal{TF}_O$ am Anfang der MAIN-Funktion eingefügt. Da bei externen Objekten die eigentliche Objektdefinition dem Compiler nicht zugänglich ist, wird eine Erzeugungsfunktion mit dem generischen Namen CREATE_`stdout` verwendet, die die Implementierung der Klasse `TermFile` bereitstellen muß. Außerdem erhält diese Funktion ebenfalls `FileSys` als CALL-BY-REFERENCE-Parameter, da in der Klassendeklaration mittels EFFECT-Pragma diese Abhängigkeit spezifiziert ist.

Nach der Eliminierung der globalen Objekte durch $\mathcal{TF}_O$ können nun durch Anwendung des Transformationsschemas $\mathcal{TF}_R$ sämtliche CALL-BY-REFERENCE-Parameter entfernt werden. Das Ergebnis dieser Transformation ist in Abb. 3.18 dargestellt.

[3]Es handelt sich hierbei im wesentlichen um ein literales Einfügen. Details können [Gre96] entnommen werden.

```
WORLD printf( WORLD FileSys, CHAR[] format, ...);

WORLD, TermFile fprintf( WORLD FileSys, TermFile stream,
                         CHAR[] format, ...);

WORLD, TermFile PrintHeader( WORLD FileSys, TermFile out)
{ FileSys = printf( FileSys, "This is SAC !\n");
  FileSys, out = fprintf( FileSys, out, "Version 0.5\n");
  RETURN( FileSys, out);
}

WORLD, INT MAIN( WORLD FileSys)
{ stdout = CREATE_stdout( FileSys);
  FileSys, stdout = PrintHeader( FileSys, stdout);
  FileSys = printf( FileSys, "Hello World!\n");
  ...
  RETURN( FileSys, res);
}
```

Abbildung 3.18: Hello-World Programm nach der Anwendung von $\mathcal{TF}_R$.

3.6 Die Schnittstelle zu anderen Programmiersprachen

Ähnlich wie in SISAL das Modulsystem dazu genutzt wird, eine Schnittstelle zu anderen Hochsprachen zu herzustellen, können in SAC sowohl Module als auch Klassen diese Aufgabe übernehmen. Solche Module/Klassen werden in SAC durch das Schlüsselwort EXTERNAL gekennzeichnet.

Im Gegensatz zu SISAL (vergl. Abschnitt 2.3) soll in SAC jedoch keine Unterscheidung der externen Module/Klassen nach der Hochsprache, in der sie implementiert sind, verbunden mit etwaigen Restriktionen in der Benutzung dieser Strukturen, vorgenommen werden. Stattdessen wird eine „konsistente" Einbindung in SAC angestrebt, d.h. die vom importierten Code zur Verfügung gestellten Datenstrukturen bzw. Funktionen sollen in SAC genauso benutzt werden können, als seien sie in SAC spezifiziert. Dabei ergeben sich folgende Probleme:

- Die Abbildung von Datenstrukturen bzw. Funktionen in das C-Objekt-Format des Binde-Systems unterscheidet sich bei den einzelnen Hochsprachen.

- Komplexe Datenstrukturen werden in imperativen Sprachen grundsätzlich anders gehandhabt als in funktionalen.

  Die Möglichkeiten zum Umgang mit komplexen Datenstrukturen sind in imperativen Sprachen so restringiert, daß der Code zur Allozierung bzw. Freigabe des für sie benötigten Speichers entweder aus dem Gültigkeitsbereich der Variablen oder aber aus vom Programmierer explizit spezifizierten Allozierungs- bzw. Deallozierungs-Statements abgeleitet werden kann.

In funktionalen Sprachen ist dies nicht der Fall. Deshalb bedarf es spezieller Speicherverwaltungsmechanismen, die erst zur Laufzeit entscheiden, wann Speicherbereiche (de-)alloziert werden müssen (können).

- Aufgrund der funktionalen Semantik dürfen SAC-Funktionen ihre Argumente nicht modifizieren, und müssen ihre Resultatwerte, die sich ausschliesslich aus den Argumenten berechnen dürfen, prinzipiell neu erzeugen.
- Schließlich bietet SAC ein Klassenkonzept, das es in anderen Programmiersprachen nicht gibt.

Die konkrete Lösung dieser Probleme hängt insbesondere von der Art der Abbildung der SAC-Konstrukte auf das C-Objekt-Format, also der Implementierung eines Compilers von SAC nach C ab. Das Sprachdesign kann an dieser Stelle lediglich ein Konzept zur Lösung der Probleme vorgeben. Die Kernidee besteht darin, mittels gezielter Compiler-Anweisungen, den sog. Pragmas, dem Compiler bei einer externen Modul- bzw. Klassen-Deklaration Hinweise darüber zur Verfügung zu stellen, wie sich die in dem Modul/der Klasse definierten Datenstrukturen und Funktionen verhalten bzw. wie aus den SAC-konformen Deklarationen auf die Struktur des C-Objekt-Formates zu schließen ist. Die syntaktischen Ergänzungen um die Verschiedenen Pragmas in SAC sind in Abb. 3.19 dargestellt.

$$
\begin{array}{lcl}
ITypeDec & \Rightarrow & \text{IMPLICIT TYPES} : [\, Id\ ;\ [\, ITypePragma\,]^{*}\,]^{*} \\
FunDec & \Rightarrow & \text{FUNCTIONS} : [\, OneFunDec\, [\, FunPragma\,]^{*}\,]^{*} \\
ObjDec & \Rightarrow & \text{GLOBAL OBJECTS} : [\, Type\ Id\ ;\ [\, ObjPragma\,]^{*}\,]^{*} \\
ITypePragma & \Rightarrow & \#\text{PRAGMA COPYFUN}\ String \\
 & | & \#\text{PRAGMA FREEFUN}\ String \\
FunPragma & \Rightarrow & \#\text{PRAGMA LINKNAME}\ String \\
 & | & \#\text{PRAGMA LINKSIGN}\ [\, Num\, [\, ,\, Num\,]^{*}\,] \\
 & | & \#\text{PRAGMA EFFECT}\ [\, Id :]\, Id\ \ [\, ,\ \ [\, Id :]\, Id\,]^{*} \\
ObjPragma & \Rightarrow & \#\text{PRAGMA INITFUN}\ String
\end{array}
$$

Abbildung 3.19: Ergänzungen der SAC-Syntax um Pragmas.

Von SAC nicht unterstützte (komplexe) Datenstrukturen wie z.B. records oder unions lassen sich als implizite Typen einführen. Um ein Kopieren sowie Freigeben solcher Strukturen durch den aus SAC-Programmen erzeugten Code zu ermöglichen,

muß jedes externe Modul zu jedem impliziten Typ *impty* zwei entsprechende generische Funktionen zur Verfügung stellen.

	impty COPY_*impty*(*impty* a)	kopiert eine Datenstruktur vom Typ *impty*
und	VOID FREE_*impty*(*impty* a)	gibt den zugehörigen Speicherbereich frei.

Da solche Funktionen in vielen Bibliotheken bereits existieren, jedoch einen anderen Namen haben, erlauben die bei der Deklaration von impliziten Typen eingeführten optionalen Pragmas COPYFUN und FREEFUN entsprechende Umbenennungen.

In ähnlicher Weise können auch globale Objekte genutzt werden, die nicht in SAC implementiert sind. Für sie entfällt zwar die Notwendigkeit einer generischen Kopierfunktion, stattdessen wird jedoch für jedes globale Objekt *globobj* einer externen Klasse *class* eine generische Konstruktorfunktion

class CREATE_*globobj*() die *globobj* erzeugt und initialisiert

erforderlich. Bei Bedarf kann auch diese Funktion mittels des Pragmas INITFUN umbenannt werden.

Die Pragmas LINKNAME und LINKSIGN dienen der Steuerung der Abbildung von SAC-Funktions-Deklarationen bzw.-Aufrufen auf die im Objekt-File erwarteten C-Deklarationen/-Aufrufe und können bei den Funktionsdeklarationen der jeweiligen Modul- bzw. Klassen-Deklaration annotiert werden. Während LINKNAME den Namen der deklarierten Funktion im Objekt-File angibt, erlaubt LINKSIGN eine Permutation der Parameter sowie Rückgabewerte einer Funktion. Da C im Gegensatz zu SAC nur genau einen Rückgabewert zuläßt, läßt sich eine Abbildung der Rückgabewerte und Parameter einer SAC-Funktion auf den Rückgabewert und die Parameter einer C-Funktion durch eine Liste von natürlichen Zahlen wie folgt definieren: Für jeden Rückgabewert/Parameter einer Funktionsdeklaration von links nach rechts gesehen spezifiziert LINKSIGN durch je eine Zahl die Position des Rückgabewertes/Parameters in der Implementierung. Dabei steht 0 für den Rückgabewert der C-Funktion. Auf diese Weise ist es nicht nur möglich, Rückgabewerte zu Parametern zu machen, sondern es können auch je ein Rückgabewert und ein Parameter auf die gleiche Parameterposition projiziert werden.

Dies erweist sich insbesondere dann als hilfreich, wenn Funktionen eingebunden werden sollen, die einen Seiteneffekt auf einem ihrer Argumente verursachen. So kann z.B. folgende C-Funktion zur Multiplikation zweier komplexer Zahlen

```
VOID mult_cplx( DOUBLE *a, DOUBLE *b)
{ DOUBLE tmp;
  tmp = a[0] * b[0] - a[1] * b[1];
  a[1]= a[0] * b[1] + a[1] * b[0];
  a[0]= tmp;
}
```

in SAC integriert werden über eine Moduldeklaration

```
MODULEDEC complex:
OWN:{
  EXPLICIT TYPES:
    cplx = DOUBLE[2];

  FUNS:
    cplx *( cplx a, cplx b);
      #PRAGMA LINKNAME mult_cplx
      #PRAGMA LINKSIGN [1,1,2]
    ⋮
}
```

Das beim Aufruf von `mult_cplx` ggf. erforderliche erstellen einer Kopie des ersten Argumentes erfolgt dabei transparent für den SAC-Programmierer durch den Compiler (siehe [Gre96]).

Ein drittes Pragma, das bei Funktionsdeklarationen annotiert werden kann, ist das Pragma EFFECT. Es wird immer dann benötigt, wenn eine Funktion integriert werden soll, die Seiteneffekte auf globalen Objekten erzeugt oder von solchen abhängt (vergl. Abschnitt 3.5.4).

Kapitel 4

Compilation von SAC nach C

Nachdem im letzten Kapitel das Sprachdesign von SAC vorgestellt worden ist, befaßt sich dieses Kapitel mit dem Entwurf eines Compilers von SAC nach C. Dazu werden schrittweise Compilations-Schemata entwickelt, die SAC-Programme in eine Zwischensprache übersetzen. Diese Zwischensprache läßt einerseits eine einfache Abbildung in C-Programme durch macro-artige Expansionen zu und ermöglicht andererseits Flexibilität bei der Wahl der C-Darstellung. Die Details einer konkreten C-Implementierung des Compilers sind [Gre96, Sie95, Wol95] zu entnehmen.

4.1 Der Sprachkern von SAC

Wie bereits erläutert, hat die Intention, nach C zu compilieren, schon beim Sprachdesign eine wichtige Rolle gespielt. Der entscheidende Faktor dabei ist die Äquivalenz der über die funktionale Sprache $\mathcal{F}$UN definierten Semantik mit der C-Semantik entsprechender Sprachkonstrukte. So können die Sprachkonstrukte des Sprachkerns von SAC fast vollständig nach C übernommen werden. Die einzige Ausnahme bildet die Existenz mehrfacher Rückgabewerte in SAC-Funktionen. Sie stellt eine echte Erweiterung gegenüber C dar.

Prinzipiell bietet C zwei verschiedene Möglichkeiten, mehrfache Rückgabewerte zu realisieren. In diesem Zusammenhang soll folgendes SAC-Beispielprogramm betrachtet werden:

```
INT, INT, INT f( INT p)
{
  ⋮
  RETURN(r, s, t);
}
```

```
INT MAIN( )
{
  ⋮
  u, v, w = f(x);
  ⋮
}
```

.

Die erste Möglichkeit einer Realisierung in C basiert darauf, die Rückgabewerte der Funktion in einer Datenstruktur zusammenzufassen. Diese Datenstruktur dient dann als Rückgabewert der aufgerufenen Funktion, so daß die aufrufende Funktion anschließend die benötigten Komponenten aus ihr selektieren kann. Für obiges Beispiel könnte ein entsprechendes C-Programm folgendermaßen aussehen:

```
TYPEDEF STRUCT f_res {
    INT res1;
    INT res2;
    INT res3;
  } f_struct;

f_struct *f( INT p)
{ STATIC f_struct RESULT;
  ⋮
  RESULT.res1 = r;
  RESULT.res2 = s;
  RESULT.res3 = t;
  RETURN(RESULT);
}
```

```
INT MAIN( )
{
  ⋮
  { f_struct *RESULT;
    RESULT = f(x);
    u = RESULT->res1;
    v = RESULT->res2;
    w = RESULT->res3;
  }
  ⋮
}
```

Die andere Möglichkeit der Realisierung multipler Rückgabewerte in C nutzt den sog. Adress-Operator & von C. Dabei übergibt die aufrufende Funktion die Adresse der Ergebnisvariablen als Argumente an die aufgerufene Funktion. Mit Hilfe dieser Adressen kann dann die aufgerufene Funktion den Variablen der aufrufenden Funktion die entsprechenden Resultate zuweisen. Das Compilat obigen Beispieles könnte also auch wie folgt aussehen:

```
INTf( INT p, INT *s_ptr, INT *t_ptr )
{
  ⋮
  *s_ptr = s;
  *t_ptr = t;
  RETURN(r);
}
```

```
INT MAIN( )
{
  ⋮
  u = f(x, &v, &w);
  ⋮
}
```

Um die Wahl zwischen diesen beiden Alternativen flexibel zu gestalten, bietet es sich an, mehrfache Rückgabewerte nicht direkt nach C zu compilieren, sondern in eine Zwischensprache, die zwar schon der C-Darstellung nahekommt, aber trotzdem die Entscheidung über die Implementierung offen läßt. Da es sich bei dieser Zwischensprache nicht um eine vollständige Sprache, sondern eher um Abstraktionen von C-Segmenten handelt, soll im weiteren von sog. ICM-Befehlen gesprochen werden,

wobei ICM für „Intermediate Code Macros“ steht. Um die gewünschte Abstraktion zu erreichen, müssen bei allen Funktionen mit mehrfachen Rückgabewerten nicht nur die eigentlichen RETURN-Anweisungen, sondern auch Funktionsdeklarationen und Funktionsanwendungen über ICM-Befehle erzeugt werden. Dazu werden folgende ICM-Befehle entworfen:

FUNDEC($name$, tag_1, $type_1$, arg_1, ..., tag_m, $type_m$, arg_m) erzeugt eine Deklaration für eine Funktion mit dem Namen $name$. Jede Gruppe tag_i, $type_i$, arg_i repräsentiert ein Argument bzw. Resultat der Funktion. Die Marke tag_i zeigt an, ob es sich um ein Argument (tag_i = IN), oder ein Resultat (tag_i = OUT) der Funktion handelt. Im Falle eines Argumentes bezeichnen $type_i$ und arg_i Typ und Namen des Argumentes; bei Resultaten stehen $type_i$ und arg_i für Typ und Namen der Variablen im RETURN-Ausdruck.

FUNRET(arg_1, ..., arg_n) erzeugt eine RETURN-Anweisung sowie ggf. zusätzlich benötigte Zuweisungen, um der aufrufenden Funktion mehrere Resultate zugänglich zu machen. Dabei bezeichnen die einzelnen arg_i Namen der Variablen im ursprünglichen RETURN-Ausdruck.

FUNAP($name$, tag_1, arg_1, ..., tag_m, arg_m) erzeugt einen Funktionsaufruf der Funktion $name$ sowie ggf. zusätzlichen C-Programmtext, um die Resultate den entsprechenden Variablen zuzuweisen. Ähnlich wie bei dem ICM-Befehl FUNDEC steht jeweils ein Paar bestehend aus einer Marke tag_i und einem Namen arg_i für je ein Argument bzw. eine Resultatvariable.

Damit läßt sich das Beispiel formulieren als:

```
FUNDEC( f, IN, INT, p, OUT, INT, r,
        OUT, INT, s, OUT, INT, t )
{
  ⋮
  FUNRET( r, s, t );
}
```

```
INT MAIN( )
{
  ⋮
  FUNAP( f, IN, x, OUT, u,
         OUT, v, OUT, w );
  ⋮
}
```

Aus dieser Darstellung lassen sich beide der oben vorgestellten Varianten durch eine jeweils kontextfreie Expansion der einzelnen ICM-Befehle herleiten. Um alle Funktionen mit mehreren Rückgabewerten in eine solche Darstellung compilieren zu können, müssen zwei Voraussetzungen gegeben sein:

1. In einer RETURN-Anweisung dürfen keine beliebigen Ausdrücke sondern nur Variablen stehen.

2. Funktionsanwendungen dürfen nicht geschachtelt sein, sondern müssen direkt auf der rechten Seite einer Zuweisung stehen.

Da diese Voraussetzungen für alle SAC-Programme durch das Einfügen temporärer Variablen geschaffen werden können, werden sie im weiteren ohne Beschränkung der Allgemeinheit als gegeben angenommen.

Mit C_{ICM} als Menge aller C-Programme, die ICM-Befehle enthalten, läßt sich dann die Compilation des Sprachkernes von SAC durch ein Compilations-Schema

$$C[\![\text{SAC-Programm}]\!] \mapsto C_{ICM}\text{-Programm}$$

beschreiben. Die einzelnen Compilations-Regeln ergeben sich wie im folgenden dargestellt. Alle Typ- und Funktionsdefinitionen können separat compiliert werden:

$$C[\![Tydef_1 ... Tydef_m \ Fundef_1 ... Fundef_n \ Main]\!] \mapsto \left\{ \begin{array}{l} C[\![Tydef_1]\!] \ldots C[\![Tydef_m]\!] \\ C[\![Fundef_1]\!] \ldots C[\![Fundef_n]\!] \ C[\![Main]\!] \end{array} \right. .$$

Typedefinitionen können literal übernommen werden:

$$C[\![\texttt{TYPEDEF} \ \ \tau \ TypeId;]\!] \mapsto \texttt{TYPEDEF} \ \ \tau \ TypeId; \quad .$$

Bei Funktionsdefinitionen wird die Deklaration generell durch den FUNDEC-Befehl ersetzt. Dazu müssen die Variablen der zugehörigen RETURN-Anweisung ermittelt werden:

$$C\left[\!\!\left[\begin{array}{l} Tr_1, \ \ldots, \ Tr_n \ FunId(\ Ta_1 \ a_1, \ \ldots, \ Ta_m \ a_m \ ) \\ \{ \ Vardec_1, \ldots, Vardec_k \\ \quad Body; \\ \quad \texttt{RETURN}(\ r_1, \ \ldots, \ r_n \ ); \\ \} \end{array} \right]\!\!\right] \mapsto \left\{ \begin{array}{l} \texttt{FUNDEC}(\ FunId, \ \texttt{IN}, \ Ta_1, \ a_1, \ \ldots, \ \texttt{IN}, \ Ta_m, \ a_m, \\ \qquad\qquad \texttt{OUT}, \ Tr_1, \ r_1, \ \ldots, \ \texttt{OUT}, \ Tr_n, \ r_n \ ) \\ \{ \ C[\![Vardec_1]\!], \ \ldots, \ C[\![Vardec_k]\!] \\ \quad C[\![Body; \ \texttt{RETURN}(\ r_1, \ \ldots, \ r_n \ );]\!] \\ \} \end{array} \right. .$$

Variablendeklarationen können genauso wie die Typedefinitionen literal übernommen werden:

$$C[\![\tau \ \ v; \ ]\!] \mapsto \tau \ v; \quad .$$

Funktionsanwendungen werden generell durch FUNAP-Befehle ersetzt; sie enthalten neben den Argumenten ebenfalls die Variablennamen der linken Seite der Zuweisung:

$$\mathrm{C}[\![v_1, \ldots, v_n = FunId(e_1, \ldots, e_m); Rest]\!] \mapsto \begin{cases} \text{FUNAP}(FunId, \text{IN}, e_1, \ldots, \text{IN}, e_m, \\ \qquad\qquad \text{OUT}, v_1, \ldots, \text{OUT}, v_n); \\ \mathrm{C}[\![Rest]\!] \end{cases} .$$

Bei allen anderen Konstrukten des Sprachkernes wie zB. Schleifen und IF-THEN-ELSE-Konstrukten wird das Compilations-Schema lediglich rekursiv auf die Komponenten der jeweiligen Konstrukte angewandt. Stellvertretend ist hier lediglich die Regel für DO-Schleifen dargestellt:

$$\mathrm{C}\left[\!\!\left[\begin{array}{l} \text{DO } \{ Ass; \} \text{ WHILE } (e); \\ Rest \end{array} \right]\!\!\right] \mapsto \begin{cases} \text{DO } \{ \mathrm{C}[\![Ass]\!]; \} \text{ WHILE } (e); \\ \mathrm{C}[\![Rest]\!]; \end{cases} .$$

Die RETURN-Anweisung wird schließlich direkt in einen FUNRET-Befehl übersetzt:

$$\mathrm{C}[\![\text{RETURN}(r_1, \ldots, r_n);]\!] \mapsto \text{FUNRET}(r_1, \ldots, r_n); .$$

4.2 Das Typsystem von SAC

Das Typsystem von SAC unterscheidet sich von dem in C in mehrfacher Hinsicht:

1. Variablen müssen nicht deklariert werden, sondern können auch vom Typsystem inferiert werden.
2. Das Typsystem kann Spezialisierungen von Funktionen erzeugen.
3. Alle Funktionen können durch eine getrennte Spezifikation verschiedener Funktionsrümpfe für verschiedene Argumenttypen überladen werden; dies umfaßt auch die primitiven Funktionen.
4. Die Array-Typen in SAC sind in C nicht vorhanden.
5. Benutzerdefinierte Typen sind eigenständige Typen und können nicht synonym mit Objekten des Typs, durch den sie definiert sind, verwendet werden.
6. Casts sind ausschließlich zwischen benutzerdefinierten Typen und denjenigen Typen möglich, durch die sie definiert sind.

Aufgrund der fundamentalen Unterschiede läßt sich das Typsystem von SAC nicht mittels einer einfachen Programmtransformation auf das von C abbilden, sondern

muß als eigenständiger Bestandteil des Compilers von SAC nach C vollständig implementiert werden. Darüber hinaus müssen die vorhandenen Typangaben eines SAC-Programmes derartig ergänzt bzw. verändert werden, daß am Ende der Compilation ein typkorrektes C-Programm vorliegt. Daraus ergeben sich zwei verschiedene Phasen der Compilation von SAC nach C; zum einen eine Typinferenzphase, in der sämtliche fehlenden Variablendeklarationen ergänzt sowie Funktionsspezialisierungen vorgenommen werden; zum anderen die eigentliche Compilation der SAC-Typdefinitionen und Variablendeklarationen in entsprechende C-Deklarationen.

Die Implementierung des Typinferenzsystems läßt sich unmittelbar aus dem in Abschnitt 3.2 erläuterten Deduktionssystem (vergl. auch Anhang B) ableiten. Ausgehend von der PRG-Regel für SAC-Programme ergeben sich für die einzelnen Unterstrukturen jeweils eindeutig die anzuwendende „Regel“, da es für alle Sprachkonstrukte in SAC genau eine Deduktions-Regel gibt. Probleme entstehen lediglich bei den Regeln für Funktionsdeklarationen und Funktionsanwendungen durch die Spezialisierung von Funktionen. Wie bereits in Abschnitt 3.3.2 ausführlich diskutiert, kann weder auf eine Spezialisierung verzichtet, noch generell eine solche vorgenommen werden. Für die meisten praktischen Belange beschränkt sich die gewünschte Spezialisierung jedoch auf wenige Instanzen. Daher scheint eine Lösung durch einen Compiler-Parameter sinnvoll, der lediglich eine bestimmte, fest vorgegebene Zahl von Spezialisierungen einer Funktion zuläßt. Sie garantiert ein Terminieren des Typsystems einerseits und ermöglicht andererseits auch für dimensionsunabhängig spezifizierte SAC-Programme die Erzeugung effizient ausführbaren Codes. Darüber hinaus ist es in den meisten Fällen sogar möglich, für alle in einem Programm vorkommenden Arrays den vollständigen Shape-Vektor zu inferieren. Daher liegt es nahe, den Compiler von SAC nach C zunächst auf solche Programme zu beschränken. Dies vereinfacht nicht nur die Compilation, sondern garantiert auch eine effiziente Ausführbarkeit.

Die Compilation der Typdefinitionen und Variablendeklarationen beschränkt sich auf die Array-Typen in SAC, da die auf den primitiven Typen beruhenden Deklarationen direkt nach C übernommen werden können (vergl. Compilations-Schema aus Abschnitt 4.1). Die in C benötigten Deklarationen für SAC-Arrays hängen direkt von der Darstellung der Arrays in C ab. Durch die Verwendung weiterer ICM-Befehle läßt sich die konkrete Darstellung von SAC-Arrays in C genauso flexibel wie die Darstellung der multiplen Rückgabewerte von Funktionen gestalten. Um den Compilations-Aufwand möglichst gering zu halten, bietet es sich an, alle Array-Typen, die auf benutzerdefinierten Typen basieren, auf solche Array-Typen zurückzuführen, die ausschließlich auf den primitven Typen $\mathcal{T}_{Simple}$ beruhen. Durch diese Zuordnung kann auf sämtliche Typdefinitionen verzichtet werden, bei denen der definierende Typ ein Array-Typ ist. Deshalb bedarf es lediglich eines neuen ICM-Befehls zur Erzeugung von Array-Variablen:

DECLARRAY($name$, τ, s_1, ..., s_n) erzeugt die Variablendeklaration(en), die benötigt werden, um in C eine SAC-Array-Variable $name$ mit dem Typ $\tau[s_1, ..., s_n]$ mit Referenzzähler darzustellen.

Entsprechend ergeben sich die Erweiterungen der bisher vorgestellten Compilations-Regeln für Typdefinitionen und Variablendeklarationen

$$\mathrm{C}[\![\,\mathrm{TYPEDEF}\quad \tau\ \mathit{TypeId};\,]\!] \mapsto \begin{cases} \mathrm{TYPEDEF}\quad \tau\ \mathit{TypeId}; & \text{falls } \mathit{Basetype}(\tau) \in \mathcal{T}_{\mathit{Simple}} \\ \texttt{/* deleted typedef */} & \text{sonst} \end{cases} \quad \text{und}$$

$$\mathrm{C}[\![\,\tau\ v;\,]\!] \mapsto \begin{cases} \tau\ v; & \text{falls } \mathit{Basetype}(\tau) \in \mathcal{T}_{\mathit{Simple}} \\ \mathrm{DECLARRAY}(v, \sigma, s_1, \ldots, s_n); & \text{falls } \mathit{Basetype}(\tau) = \sigma[s_1, ..., s_n] \end{cases} .$$

4.3 Arrays und Array-Operationen

Konzeptuell sind zwei, nämlich der Shape- und der Datenvektor, notwendig, um SAC-Arrays darzustellen. Durch die Einschränkung auf SAC-Programme, für die die Shape-Vektoren sämtlicher Arrays durch das Typsystem inferierbar sind, kann auf eine Verwaltung von Shape-Vektoren zur Laufzeit verzichtet werden. Das Problem der Compilation reduziert sich damit im wesentlichen auf eine Realisierung der impliziten Speicherverwaltung. Sie wird dadurch erforderlich, daß in SAC, wie in allen funktionalen Sprachen, Arrays konzeptuell genauso gehandhabt werden wie alle anderen Datenobjekte auch. Das bedeutet, daß sie wie skalare Werte (Integer-Zahlen, Gleitkomma-Zahlen, etc.) als Argumente an Funktionen übergeben und durch mehrfaches Vorkommen der entsprechenden formalen Parameter im Rumpf der Funktion dupliziert werden können. Im Gegensatz zu skalaren Werten führt eine direkte Umsetzung dieses Prinzips bei Datenstrukturen wie Arrays zu einem erheblichen Zeitaufwand sowie Speicherbedarf. Um überflüssige Kopien von Arrays zu vermeiden sowie den Speicherbedarf so gering wie möglich zu halten, wird die Methode der sog. Referenzzählung[SS88, Coh81, Can89] verwendet.

Die zugrunde liegende Idee ist hier, anstelle expliziter Kopien zunächst nur Zeiger auf das Array zu übergeben und erst dann, wenn eine dieser „virtuellen Kopien" modifiziert werden soll, eine reale Kopie zu erstellen. Um die Anzahl der virtuellen Kopien nachhalten zu können, wird zu jedem Array ein Referenzzähler assoziiert. Der Referenzzähler eines aktuellen Parameters wird beim Eintritt in den Funktionsrumpf um die Anzahl seiner angewandten Vorkommen im Rumpf erhöht und anschließend um 1 vermindert. Dies modelliert ein Erzeugen von Kopien für alle angewandten Vorkommen im Rumpf sowie das Konsumieren des Parameters durch die

Funktion selbst. Sobald der Referenzzähler den Wert 0 erreicht, wird der zugehörige Speicherbereich freigegeben.

Wie die meisten imperativen Programmiersprachen unterstützt C einen solchen Umgang mit Datenstrukturen nicht, sondern bietet dem Programmierer stattdessen die Möglichkeit, die Verwaltung des Speichers explizit zu steuern. Dazu stehen Zeiger, Stern- und Adress-Operatoren sowie drei verschiedene Möglichkeiten der Speicher-(De)-Allozierung zur Verfügung:

- Zum einen kann eine Datenstruktur statisch alloziert werden. Dies wird initiiert durch die Deklaration einer globalen Variablen oder sog. static-Deklarationen in Funktionsrümpfen. Alle statischen Datenstrukturen werden zu Beginn der Programm-Ausführung alloziert und am Ende der Programm-Ausführung erst wieder freigegeben.

- Eine lokale Deklaration einer Datenstruktur in einem Funktionsrumpf bewirkt eine Speicherallozierung zur Laufzeit beim Eintritt in den Funktionsrumpf; die Freigabe des Speichers erfolgt beim Verlassen des Funktionsrumpfes.

- Die dritte Möglichkeit bieten Systemroutinen zur expliziten Allozierung bzw. Freigabe von Speicherbereichen (`malloc`, `free`).

Da es für eine Implementierung des Referenzzählens notwendig ist, zur Laufzeit entscheiden zu können, ob alloziert, dealloziert oder modifiziert werden muß, bietet sich eine Verwaltung des Speichers durch die Systemroutinen `malloc` und `free` an. Virtuelle Kopien von Arrays können durch das Kopieren von Zeigern auf die jeweilige Array-Darstellung im Speicher sowie entsprechende Erhöhung des zugehörigen Referenzzählers angelegt werden. Lesende Zugriffe auf Arrays können ohne Kopieraufwand auch auf den virtuellen Kopien mittels Zeigerderefenzierung (Stern-Operator) erfolgen; lediglich bei Array-Modifikationen muß neuer Speicher alloziert und eine reale (modifizierte) Kopie erzeugt werden.

Für eine konkrete Array-Darstellung bestehend aus Datenvektor und zugehörigem Refenzzähler gibt es in C verschiedene Möglichkeiten:

- Eine Abbildung auf zwei getrennte Speicherbereiche erfordert die Verwaltung von jeweils zwei Zeigern pro Array. Das bedeutet insbesondere bei intensiver Nutzung rekursiver Funktionen einen erhöhten Speicherbedarf, erlaubt jedoch einen schnellen Zugriff auf die Daten, da nur genau eine Indirektion notwendig ist.

- Eine Zusammenfassung der beiden Datenfelder zu einer Datenstruktur ist zwar die kompakteste Darstellung, erfordert jedoch bei jedem Zugriff auf die eigentlichen Array-Elemente die Berücksichtigung eines offsets.

- Schließlich ist eine Verwendung von Deskriptoren, die den Referenzzähler sowie einen Zeiger auf die eigentlichen Array-Elemente enthalten, möglich. Sie erlaubt es, in späteren Compiler-Versionen zusätzliche Attribute von Arrays wie beispielsweise Shape-Vektoren in die Deskriptoren einzufügen, ohne irgendwelche Veränderungen bei Array-Zugriffen vornehmen zu müssen. Der Nachteil dieser Variante liegt in der erforderlichen zweiten Dereferenzierung beim Zugriff auf Array-Elemente.

Um sich bei der Compilation nicht von vornherein auf eine dieser Varianten festlegen zu müssen, sollen auch hier ICM-Befehle eingeführt werden, die sich später kontextfrei in die gewünschten C-Fragmente übersetzen lassen. Für die Allozierung, Initialisierung sowie Zuweisungen von Arrays stehen folgende ICM-Befehle zur Verfügung:

ALLOCARRAY(τ, *name*, n) erzeugt Befehle zum Allozieren von Speicher für das Array *name* und initialisiert den Referenzzähler mit n. τ gibt den Typ der Elemente an.

CREATECONSTARRAYS(*name*, d_1, ..., d_n) erzeugt Befehle zum Initialisieren des Arrays *name* mit den Werten d_1, ..., d_n. Der Referenzzähler des Arrays bleibt dabei unberührt.

CREATECONSTARRAYA(*name*, d_1, ..., d_n) entspricht CREATECONSTARRAYS; anstelle skalarer Daten handelt es sich bei den d_i um Array-Variablen.

ASSIGNARRAY(*name*$_1$, *name*$_2$) erzeugt eine Zuweisung *name*$_1$ = *name*$_2$ von Array-Variablen. Der Referenzzähler, auf den via *name*$_1$ bzw. *name*$_2$ zugegriffen werden kann, wird dabei nicht verändert.

Für die Handhabung der Referenzzähler von SAC-Arrays sowie für die Freigabe von Array gibt es zwei weitere ICM-Befehle:

INCRC(*name*, n) erhöht den Referenzzähler des Arrays *name* um n.

DECRCFREEARRAY(*name*, n) erniedrigt den Referenzzähler des Arrays *name* um n. Wird der Referenzzähler dabei 0, so wird der zugehörige Speicherbereich freigegeben.

Mit Hilfe dieser ICM-Befehle kann in den folgenden Abschnitten schrittweise die Compilation von SAC-Programmen, die Arrays bzw. Array-Operationen enthalten, vorgestellt werden.

4.3.1 Compilation von Zuweisungen

Vor einer formalen Erweiterung des in Abschnitt 4.1 vorgestellten Compilations-Schemas soll zunächst die Compilation eines parameterlosen Funktionsrumpfes an einem einfachen Beispiel betrachtet werden:

```
{ INT[4] a;
  INT[4] b;

  a = [1, 2, 3, 4];     wird zu
  b = a;
  RETURN(a);
}
```

```
{ DECLARRAY(INT, a, 4);
  DECLARRAY(INT, b, 4);

  ALLOCARRAY(INT, a, 2);
  CREATECONSTARRAYS(a, 1, 2, 3, 4);
  ASSIGNARRAY(b, a);
  DECRCFREEARRAY(b, 1);
  C⟦RETURN(a);⟧
}
```

Nach den ICM-Befehlen zur Erzeugung der notwendigen Deklarationen für die beiden Arrays `a` und `b` wird zunächst der für das Array `a` benötigte Speicher alloziert und anschließend mit dem CREATECONSTARRAYS-Befehl initialisiert. Bereits bei der Allozierung des Speichers wird der Referenzzähler des Arrays mit 2 initialisiert. Er ergibt sich direkt aus der Anzahl der freien Vorkommen von `a` im Rest des Funktionsrumpfes und kann mittels der in Def.3.1.5 beschriebenen *Refs*-Funktion bestimmt werden. Anschließend wird dem Array `b` das Array `a` zugewiesen und die Referenzzähler von `a` und `b` angepaßt. Die Anpassung entspricht einem Erhöhen des Referenzzählers von `b` um die Anzahl der freien Vorkommen von `b` im Rest-Programm (*Refs* `(b,` RETURN`(a);)=0`) und anschließendem Dekrementieren des Referenzzählers von `a`, um das Konsumieren von `a` nachzubilden. Da es sich bei `b` ohnehin nur um eine virtuelle Kopie von `a` handelt, beide Array-Variablen also auf den gleichen Speicherbereich verweisen, können diese Operationen zusammengefaßt werden. Deshalb erfolgt ein Dekrementieren des Referenzzählers von `b` um den Wert 1. Schließlich folgt das Compilat der RETURN-Anweisung.

Bei der formalen Spezifikation des Compilations-Schemas ergibt sich ein Problem bei der Bestimmung der freien Vorkommen der Array-Variablen im Rest eines Funktionsrumpfes, da bei der Compilation von einigen Konstrukten, wie z.B. IF-THEN-ELSE-Konstrukten, nur Teilprogramme für die Compilation betrachtet werden und somit der Rest des Funktionsrumpfes nicht vorliegt. Hierzu eine modifizierte Version obigen Beispieles:

```
{
  IF (TRUE)
    a = [1, 2, 3, 4];
  ELSE
    a = [5, 6, 7, 8];
  b = a;
  RETURN(a);
}
```

Bei diesem Beispiel liegt während der Compilation des THEN-Zweiges ausschließlich die Zuweisung `a = [1, 2, 3, 4];` vor. Die notwendige Initialisierung des Referenzzählers von `a` mit 2 kann jedoch nur aus dem Kontext des gesamten Funktionsrumpfes abgeleitet werden. Deshalb wird ein zweites Compilations-Schema CR definiert, das neben dem eigentlich zu compilierenden Teilprogramm auch noch den Kontext für die Referenzzählung bereitstellt:

$$\mathrm{CR}[\![\,\text{SAC-}\textit{Prg-Segment}\,,\ \text{SAC-}\textit{Prg-Kontext}\,]\!] \mapsto \mathrm{C}_{\mathrm{ICM}}\text{-Prg} \quad .$$

Die einzelnen Compilations-Regeln dieses Schemas sowie dessen Einbettung in das im Abschnitt 4.1 vorgestellt C-Schema soll im Rest dieses sowie den folgenden Abschnitten schrittweise entwickelt werden. Zur Beschreibung der verschiedenen Regeln wird *AdjustRC*(*var*, *n*) als Kurzschreibweise für ggf. erforderliche Referenzzähler-Anpassungen der Variablen *var* um das ganzzahlige offset *n* verwendet. Formal ist

$$\textit{AdjustRC}(var, n) := \begin{cases} \text{DECRCFREEARRAY}(\ var,\ n); & \text{falls } n < 0 \\ \texttt{/* no RC adjusments */} & \text{falls } n = 0 \\ \text{INCRC}(\ var,\ n); & \text{falls } n \geq 0 \end{cases} \quad .$$

Damit kann sich wieder dem Entwurf der Compilations-Regeln für Zuweisungen an Array-Variablen zugewendet werden. Die Compilation der Zuweisung eines konstanten Arrays resultiert in einer Speicherallozierung mit anschließender Initialisierung:

$$\mathrm{CR}[\![v \texttt{ = [} d_1\texttt{, } \ldots\texttt{, } d_n\texttt{]; } \mathit{Rest}\ ,\ \mathcal{F}]\!] \mapsto \begin{cases} \textsc{AllocArray}(\ \tau\texttt{, } v\texttt{, } \mathit{Refs}\ (v\texttt{, } \mathit{Rest; } \mathcal{F}))\texttt{;} \\ \textsc{CreateConstArrayS}(\ v\texttt{, } d_1\texttt{, } \ldots\texttt{, } d_n)\texttt{;} \\ \mathrm{CR}[\![\mathit{Rest}\ ,\ \mathcal{F}]\!] \\ \qquad\qquad \text{falls } \tau = \mathit{Basetype}(\mathit{TYPE}(d_i)) \ \wedge\ \tau \in \mathcal{T}_{\mathit{Simple}} \\ \\ \textsc{AllocArray}(\ \tau\texttt{, } v\texttt{, } \mathit{Refs}\ (v\texttt{, } \mathit{Rest; } \mathcal{F}))\texttt{;} \\ \textsc{CreateConstArrayA}(\ v\texttt{, } d_1\texttt{, } \ldots\texttt{, } d_n)\texttt{;} \\ \mathrm{CR}[\![\mathit{Rest}\ ,\ \mathcal{F}]\!] \\ \qquad\qquad \text{falls } \tau[s_1, ..., s_m] = \mathit{Basetype}(\mathit{TYPE}(d_i)) \end{cases}.$$

Der bei der Allozierung benötigte Wert für die Initialisierung des Referenzzählers kann mittels der *Refs*-Funktion und des Programmkontexts $\mathcal{F}$ ermittelt werden.
Die Zuweisung eines Arrays an ein anderes wird in einen ASSIGNARRAY-Befehl mit anschließender Anpassung des Referenzzählers übersetzt. Auch in diesem Fall findet der Programmkontext $\mathcal{F}$ bei der Bestimmung der freien Vorkommen der Variablen Verwendung:

$$\mathrm{CR}[\![v \texttt{ = } w\texttt{; } \mathit{Rest}\ ,\ \mathcal{F}]\!] \mapsto \begin{cases} \textsc{AssignArray}(\ v\texttt{, } w)\texttt{;} \\ \mathit{AdjustRC}(v\texttt{, } \mathit{Refs}\ (v\texttt{, } \mathit{Rest; } \mathcal{F})\texttt{ - 1)}\texttt{;} & \text{falls } \mathit{Basetype}(\mathit{TYPE}(\mathrm{v})) \in \mathcal{T}_{\mathit{Array}} \\ \mathrm{CR}[\![\mathit{Rest}\ ,\ \mathcal{F}]\!] & \\ v = w;\ \mathrm{CR}[\![\mathit{Rest}\ ,\ \mathcal{F}]\!] & \text{sonst} \end{cases}.$$

Mit diesen Compilations-Regeln ergibt sich für die Compilation des THEN-Zweiges des modifizierten Beispiels:

$$\mathrm{CR}[\![\texttt{a = [1,2,3,4]; , } \begin{array}{l}\texttt{b = a;}\\ \textsc{return}\texttt{(a);}\end{array}]\!] \Rightarrow \begin{array}{l}\textsc{AllocArray}(\textsc{int}\texttt{, a, 2);}\\ \textsc{CreateConstArrayS}\texttt{(a, 1, 2, 3, 4);}\end{array},$$

da *Refs* (a, b=a; RETURN(a);)=2.

4.3.2 Compilation von Funktionsdefinitionen

Abhängig von der Implementierung der Referenzzähler von Arrays erfordert die Übergabe eines Arrays als Argument in SAC zusätzliche formale Paramter bzw. zusätzliche C-Anweisungen im Funktionsrumpf. Deshalb muß der in Abschnitt 4.1 vorgestellte Mechanismus zur Compilation von Funktionsdeklarationen sowie Funktionsaufrufen speziell für Datenstrukturen mit Referenzzählern erweitert werden.

Auf der einen Seite bedeutet dies eine Ergänzung der dafür zuständigen ICM-Befehle, auf der anderen Seite muß das Compilations-Schema ebenfalls angepaßt werden. Die Ergänzung der ICM-Befehle beschränkt sich darauf, die ohnehin vorhandenen Marken zur Kennzeichnung der formalen Parameter bzw. der Rückgabewerte um zwei weitere Alternativen IN_RC und OUT_RC für Arrays zu erweitern sowie entsprechende Marken bei dem FUNRET-Befehl einzufügen. Dadurch ergeben sich für diese ICM-Befehle folgende erweiterte Definitionen:

FUNDEC($name$, tag_1, $type_1$, arg_1, ..., tag_m, $type_m$, arg_m) erzeugt eine Deklaration für eine Funktion mit dem Namen $name$. Jede Gruppe tag_i, $type_i$, arg_i repräsentiert ein Argument bzw. Resultat der Funktion. Die Marke tag_i zeigt an, ob es sich um ein Argument (tag_i = IN) oder ein Resultat (tag_i = OUT) handelt und ob dieses einen Referenzzähler hat (tag_i = IN_RC bzw. tag_i = OUT_RC) oder nicht (tag_i = IN bzw. tag_i = OUT). Im Falle eines Argumentes bezeichnen $type_i$ und arg_i Typ und Namen des Argumentes; bei Resultaten stehen $type_i$ und arg_i für Typ und Namen der Variablen im RETURN-Ausdruck.

FUNRET(tag_1, arg_1, ..., tag_n, arg_n) erzeugt eine RETURN-Anweisung sowie ggf. zusätzlich benötigte Anweisungen, um der aufrufenden Funktion mehrere Resultate zugänglich zu machen. Dabei bezeichnen die einzelnen arg_i Namen der Variablen im ursprünglichen RETURN-Ausdruck. Die Marken tag_i zeigen an, ob die Resultate einen Referenzzähler haben (tag_i = OUT_RC) oder nicht (tag_i = OUT).

FUNAP($name$, tag_1, arg_1, ..., tag_m, arg_m) erzeugt einen Funktionsaufruf der Funktion $name$ sowie ggf. zusätzlichen C-Programmtext, um die Resultate den entsprechenden Variablen zuzuweisen. Ähnlich wie bei dem ICM-Befehl FUNDEC steht jeweils ein Paar bestehend aus einer Marke tag_i und einem Namen arg_i für je ein Argument bzw. eine Resultatvariable.

Die Anpassung der Compilations-Schemata beschränkt sich nicht nur auf die Integration der erweiterten Marken und den Übergang vom C- auf das CR-Schema für den Rumpf der Funktion, sondern erfordert auch die Anpassungen der Refenzzähler für Array-Parameter. Zunächst die Regel für Funktionsdeklarationen:

$$
\mathrm{C}\left[\!\!\left[\begin{array}{l} Tr_1,\ \ldots,\ Tr_n\ \ FunId(\ Ta_1\ a_1,\ \ldots,\ Ta_m\ a_m\ ) \\ \{\ Vardec_1,\ \ldots,\ Vardec_k \\ \quad Body; \\ \quad \text{RETURN}(\ r_1,\ \ldots,\ r_n\ ); \\ \} \end{array}\right]\!\!\right]
$$

$$
\mapsto \left\{\begin{array}{l} \text{FUNDEC}(\ FunId,\ intag_1,\ Ta_1,\ a_1,\ \ldots,\ intag_m,\ Ta_m,\ a_m, \\ \qquad\qquad\qquad outtag_1,\ Tr_1,\ r_1,\ \ldots,\ outtag_n,\ Tr_n,\ r_n\ ) \\ \{\ \mathrm{C}[\![\ Vardec_1\ ]\!],\ \ldots,\ \mathrm{C}[\![\ Vardec_k\ ]\!] \\ \quad AdjustRC(\ b_i,\ Refs\ (b_i,\ Body)\texttt{-1}); \\ \quad \mathrm{CR}[\![\ Body;\ \text{RETURN}(\ r_1,\ \ldots,\ r_n\ );\ ,\ \varepsilon\ ]\!] \\ \} \end{array}\right. ,
$$

wobei $intag_i = \begin{cases} \text{IN} & \text{falls } Basetype(Ta_i) \in \mathcal{T}_{Simple} \\ \text{IN\_RC} & \text{sonst} \end{cases}$,

$outtag_i = \begin{cases} \text{OUT} & \text{falls } Basetype(Tr_i) \in \mathcal{T}_{Simple} \\ \text{OUT\_RC} & \text{sonst} \end{cases}$

und $b_i \in \{a_i \mid Basetype(Ta_i) \in \mathcal{T}_{Array}\}$.

Sie erzeugt für Array-Parameter sowie Array-Rückgabewerte IN_RC bzw. OUT_RC-Marken und sorgt für eine Anpassung der Referenzzähler von Array-Parametern. Dazu wird der jeweilige Referenzzähler um die um 1 verringerte Anzahl der freien Vorkommen im Rumpf erhöht bzw. erniedrigt. Die Compilation des eigentlichen Rumpfes erfolgt mit dem CR-Schema, das mit einem leeren Kontext (ε) aufgerufen wird.
Die Compilation von Funktionsanwendungen muß lediglich um eine Anpassung der Referenzzähler der Resultate erweitert werden. Das Dekrementieren der Arrays erfolgt durch das Compilat für die Funktion selbst. So ergibt sich

$$\mathrm{CR}[\![v_1, \ldots, v_n = FunId(e_1, \ldots, e_m); Rest, \mathcal{F}]\!]$$
$$\mapsto \begin{cases} \text{FUNAP}(FunId, intag_1, e_1, \ldots, intag_m, e_m, \\ \qquad\qquad outtag_1, v_1, \ldots, outtag_n, v_n); \\ AdjustRC(v_i, Refs(v_i, Rest; \mathcal{F})\text{-}1); \\ \mathrm{CR}[\![Rest, \mathcal{F}]\!] \end{cases},$$

wobei $intag_i = \begin{cases} \text{IN} & \text{falls } Basetype(TYPE(e_i)) \in \mathcal{T}_{Simple} \\ \text{IN\_RC} & \text{sonst} \end{cases}$

und $outtag_i = \begin{cases} \text{OUT} & \text{falls } Basetype(TYPE(v_i)) \in \mathcal{T}_{Simple} \\ \text{OUT\_RC} & \text{sonst} \end{cases}$.

Die Compilation der RETURN-Anweisung wird um die Marken zur Kennzeichnung der Array-Rückgabewerte ergänzt:

$$\mathrm{CR}[\![\text{RETURN}(r_1, \ldots, r_n); , \mathcal{F}]\!] \mapsto \text{FUNRET}(tag_1, r_1, ..., tag_n, r_n);,$$

wobei $tag_i = \begin{cases} \text{OUT} & \text{falls } Basetype(TYPE(r_i)) \in \mathcal{T}_{Simple} \\ \text{OUT\_RC} & \text{sonst} \end{cases}$.

4.3.3 Compilation der IF-THEN-ELSE-Konstrukte

Auch nach der Einführung der Referenzzählung kann die Compilation von IF-THEN-ELSE-Konstrukten auf die Compilation der einzelnen Komponenten (THEN-Zweig, ELSE-Zweig und Restrumpf) zurückgeführt werden. Bei der Compilation der Zweige ist zu beachten, daß zum einen die Kontexte für die Bestimmung der Anzahl der freien Vorkommen von Array-Variablen (zweite Komponente des CR-Schemas) ergänzt und zum anderen die Referenzzähler für den jeweiligen Zweig individuell angepaßt werden. Die notwendige Anpassung der Referenzzähler von Array-Variablen ergibt sich direkt aus der Definition der *Refs*-Funktion für IF-THEN-ELSE-Konstrukte. Sie liefert das Maximum der für die Konkatenation von THEN-Zweig bzw. ELSE-Zweig mit dem Restrumpf der Funktion ermittelten Werte. Eine korrekte Anpassung des Referenzzählers einer Array-Variablen für einen Zweig ergibt sich also durch Dekrementieren des Referenzzählers um die Differenz der beiden Werte, falls die Array-Variable im anderen Zweig häufiger referenziert wird. Formal läßt sich die Compilation von IF-THEN-ELSE-Konstrukten fassen durch:

$$\mathrm{CR}\left[\!\!\left[\begin{array}{l}\texttt{IF (}e\texttt{) \{}\\ \quad Ass_t\texttt{;}\\ \texttt{\}}\\ \texttt{ELSE \{}\\ \quad Ass_e\texttt{;}\\ \texttt{\};}\\ Rest\end{array}\right]\!\!\right] \mapsto \left\{\begin{array}{l}\texttt{IF (}e\texttt{) \{}\\ \quad \textsc{DecRCFreeArray}\texttt{(}\ a_i\texttt{,}\ m_i\texttt{);}\\ \quad \mathrm{CR}\left[\!\!\left[Ass_t\texttt{;}\ ,\ Rest\,\mathcal{F}\right]\!\!\right]\\ \texttt{\}}\\ \texttt{ELSE \{}\\ \quad \textsc{DecRCFreeArray}\texttt{(}\ b_i\texttt{,}\ n_i\texttt{);}\\ \quad \mathrm{CR}\left[\!\!\left[Ass_e\texttt{;}\ ,\ Rest\,\mathcal{F}\right]\!\!\right]\\ \texttt{\}}\\ \mathrm{CR}\left[\!\!\left[Rest\ ,\ \mathcal{F}\right]\!\!\right]\end{array}\right. ,$$

wobei $a_i \in \{v \mid \mathit{Basetype}(\mathit{TYPE}(v)) \in \mathcal{T}_{\mathit{Array}}$
$\wedge\ \mathit{Refs}\,(v, \mathit{Ass}_e; \mathit{Rest}; \mathcal{F}) > \mathit{Refs}\,(v, \mathit{Ass}_t; \mathit{Rest}; \mathcal{F})\}$
und $m_i = \mathit{Refs}\,(a_i, \mathit{Ass}_e; \mathit{Rest}; \mathcal{F}) - \mathit{Refs}\,(a_i, \mathit{Ass}_t; \mathit{Rest}; \mathcal{F})$,
sowie $b_i \in \{v \mid \mathit{Basetype}(\mathit{TYPE}(v)) \in \mathcal{T}_{\mathit{Array}}$
$\wedge\ \mathit{Refs}\,(v, \mathit{Ass}_t; \mathit{Rest}; \mathcal{F}) > \mathit{Refs}\,(v, \mathit{Ass}_e; \mathit{Rest}; \mathcal{F})\}$
und $n_i = \mathit{Refs}\,(b_i, \mathit{Ass}_t; \mathit{Rest}; \mathcal{F}) - \mathit{Refs}\,(b_i, \mathit{Ass}_e; \mathit{Rest}; \mathcal{F})$.

Zur Verdeutlichung obiger Regel sowie der Interaktion dieser Regel mit den Regeln für Zuweisungen aus Abschnitt 4.3.1 soll nochmals eine modifizierte Form unseres Beispieles betrachtet werden. In Abb. 4.1 ist schrittweise der gesamte Compilations-Vorgang eines Funktionsrumpfes dargestellt. Den Ausgangspunkt der Compilation bildet der in Abb. 4.1(a) dargestellte Funktionsrumpf. Von der letzten Version des Beispieles unterscheidet er sich lediglich dadurch, daß eine Array-Variable `d` eingeführt wird, die nur in einem der beiden Zweige des IF-THEN-ELSE-Konstruktes referenziert ist. Da `d` ansonsten nicht referenziert wird, resultiert die Compilation der Zuweisung an `d` in einer Allozierung mit Referenzzähler 1 sowie anschließender Initialisierung durch durch den CREATECONSTARRAYS-Befehl (Abb. 4.1(b)). Bei der Compilation des IF-THEN-ELSE-Konstruktes erfolgt zum einen die Anpassung des Referenzzählers im THEN-Zweig, da weder hier noch in dem Programmteil hinter dem IF-THEN-ELSE-Konstrukt die Variable `d` referenziert wird und zum anderen die Erweiterung des bisher leeren Kontextes um den Rest des Rumpfes bei der Compilation der einzelnen Zweige. Auf diese Weise gelangt man zu Abb. 4.1(c). Anschließend werden noch die drei verbleibenden Array-Zuweisungen sowie die RETURN-Anweisung compiliert. Bis auf die Compilation des ELSE-Zweiges sind diese Transformationen bereits aus den vorherigen Versionen des Beispiels bekannt. Die Übersetzung des ELSE-Zweiges resultiert in dem ASSIGNARRAY-Befehl sowie anschließender Erhöhung des Referenzzählers von `a` um `1`, da *Refs* (`a, b=a; RETURN(a);`)-1 = 2-1 = 1. Der vollständig compilierte Funktionsrumpf ist in Abb. 4.1(d) abgebildet.

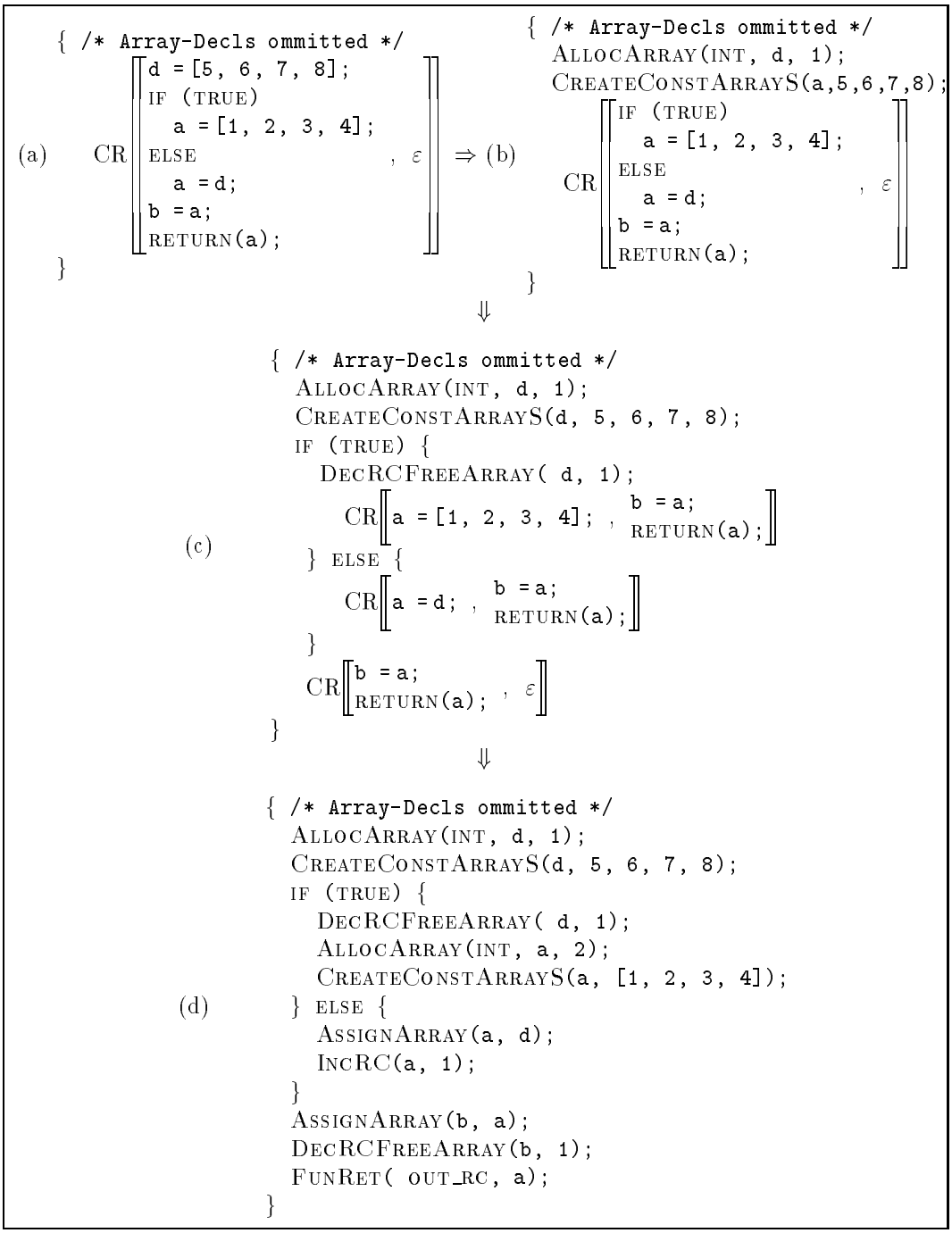


Abbildung 4.1: Beispiel-Compilation eines Funktionsrumpfes

4.3.4 Compilation der Schleifenkonstrukte

Ähnlich wie bei den IF-THEN-ELSE-Konstrukten kann auch die Compilation der Schleifenkonstrukte auf die Compilation der Komponenten (Schleifenrumpf und Restrumpf der Funktion) zurückgeführt werden. Das Problem liegt hier in der Anpassung der Referenzzähler der im Schleifenrumpf vorkommenden Array-Variablen. Zum einen müssen unabhängig von der Anzahl der Schleifendurchläufe nach der Schleife definierte Referenzzählerstände vorliegen, und zum anderen muß zwischen den verschiedenen Arten der Verwendung von Array-Variablen im Schleifenrumpf unterschieden werden (ausschließlich angewandte Vorkommen, angewandte vor definierenden Vorkommen definierende Vorkommen vor angewandten Vorkommen, etc.).

Eine korrekte Anpassung der Referenzzähler läßt sich aus der in Abschnitt 3.1.5 entwickelten Transformation $\mathcal{TF}_K$ von SAC-Schleifen in tail-end-rekursive Funktionen ableiten. Dazu wird ein Schleifenkonstrukt zunächst in eine Dummy-Funktionsdefinition sowie in eine Anwendung dieser Funktion transformiert. Für diese Ausdrücke ist die Compilation bereits bekannt, so daß alle benötigten Referenzzähler-Anpassungen inferiert werden können. Anschließend transformieren wir die Funktionsdefinition nebst Anwendung wieder in die entsprechende Schleife. Auf diese Weise erhalten wir Compilations-Regeln, die direkt auf die Schleifenkonstrukte anwendbar sind, ohne daß bei der Compilation eine explizite Transformation in die funktionale Darstellung und zurück erforderlich wird.

Exemplarisch wollen wir an dieser Stelle die Compilations-Regel für WHILE-Schleifen ableiten. Betrachten wir dazu folgende schematische Darstellung einer WHILE-Schleife in einem Funktionsrumpf:

```
{ ...
  WHILE (Expr) {
    Assigns;
  };
  Rest; }
```

Die Transformation in eine tail-end-rekursive Funktion gemäß $\mathcal{TF}_K$ ergibt

```
{ ...
  y1, ..., ym = dummy( x1, ..., xn);
  Rest;
}
```

wobei `dummy` definiert ist als

```
τ1, ..., τm dummy( σ1 x1, ..., σn xn)
{
  IF (Expr) {
    Assigns;
    y1, ..., ym = dummy( x1, ..., xn);
  };
  RETURN(y1, ..., ym);
}
```

mit $\{x_1, ..., x_n\} = \mathrm{V}_{need}(Assigns) \cup (\mathrm{V}_{pot\_def}(Assigns) \cap \mathrm{V}_{need}(Rest))$
und $\{y_1, ..., y_m\} = (\mathrm{V}_{def}(Assigns) \cap \mathrm{V}_{need}(Rest))$.

Unter Anwendung der bisher definierten Compilations-Regeln ergibt sich für obige Funktionsanwendung

$$\mathrm{CR}\left[\!\!\left[\begin{array}{l} \{\ \ldots \\ \quad \mathtt{y_1, \ldots, y_m = dummy(x_1, \ldots, x_n);} \\ \quad \mathit{Rest}; \\ \} \end{array} \;,\; \mathcal{F} \right]\!\!\right]$$

$$\Rightarrow \quad \begin{array}{l} \{\ \ldots \\ \quad \textsc{FunAp}\mathtt{(dummy, \ldots);} \\ \quad \mathit{AdjustRC}\mathtt{(y_i,}\ \mathit{Refs}\ \mathtt{(y_i,}\ \mathit{Rest};\ \mathcal{F}\mathtt{)-1);} \\ \quad \mathrm{CR}[\![\mathit{Rest};\ ,\ \mathcal{F}]\!] \\ \} \end{array}$$

und für die Definition der Funktion `dummy`

$$\mathrm{C}\left[\!\!\left[\begin{array}{l} \tau_1, \ldots, \tau_m\ \mathtt{dummy(}\ \sigma_1\ \mathtt{x_1, \ldots,}\ \sigma_n\ \mathtt{x_n)} \\ \{ \\ \quad \textsc{if}\ \mathtt{(}\mathit{Expr}\mathtt{)}\ \{ \\ \quad\quad \mathit{Assigns}; \\ \quad\quad \mathtt{y_1, \ldots, y_m = dummy(x_1, \ldots, x_n);} \\ \quad \}; \\ \quad \textsc{return}\mathtt{(y_1, \ldots, y_m);} \\ \} \end{array} \right]\!\!\right]$$

```
FUNDEC( dummy, ...)
{ /* Array-Decls ommited */

  AdjustRC( x_i, Refs (x_i, Body_of_dummy)-1);
  IF (Expr) {
      DECRCFREEARRAY( a_i, m_i);
      CR[[ Assigns; , y_1, ..., y_m = dummy( x_1, ..., x_n);
                      RETURN(y_1, ..., y_m); ]]
⇒     FUNAP( dummy, ...);
  } ELSE {
      DECRCFREEARRAY( b_i, n_i);
  }
  FUNRET(tag_1, y_1, ..., tag_m, y_m);
}
```

mit a_i, m_i, b_i und n_i gemäß der Compilations-Regel für IF-THEN-ELSE-Konstrukte aus Abschnitt 4.3.3 .

Um den compilierten Funktionsrumpf und die Funktionsanwendung wieder in eine WHILE-Schleife transformieren zu können, werden zunächst die initialen Referenzzähleranpassungen *AdjustRC*(x_i, *Refs* (x_i, *Body_of_dummy*)-1) in beide Zweige des IF-THEN-ELSE-Konstruktes verlagert. Dadurch erhalten wir in beiden Zweigen des IF-THEN-ELSE-Konstruktes aufeinanderfolgende Referenzzählermodifikationen, von denen sich die meisten paarweise auf ein und dasselbe Array beziehen. Die Ursache hierfür liegt darin, daß zunächst die Referenzzähler aller Argumente auf das Maximum der freien Vorkommen der beiden Zweige angepaßt und anschließend sofort wieder auf die Anzahl der freien Vorkommen im jeweiligen Zweig erniedrigt werden. Deshalb können diese Operationen durch eine ausschließliche Anpassung der Referenzzähler an den jeweiligen Zweig zusammengefaßt werden. Für den THEN-Zweig ergibt sich also für die formalen Parameter mit Array-Typ folgende Anpassung:

$$AdjustRC(\ x_i,\ Refs\left(x_i,\ \begin{array}{l} Assigns; \\ y_1, \ldots, y_m = \texttt{dummy}(\ x_1, \ldots, x_n); \\ \text{RETURN}(y_1, \ldots, y_m); \end{array}\right)\text{-}1)$$

und für den ELSE-Zweig:

AdjustRC(x_i, *Refs* (x_i, RETURN(y_1, ..., y_m);)-1)
= DECRCFREEARRAY(w_i, 1) mit $w_i \in \{x_i\} \backslash \{y_i\}$.

Insgesamt ergibt sich für die Funktion `dummy` also als Compilat

```
FunDec( dummy, ...)
{ /* Array-Decls ommited */

  IF (Expr) {
      AdjustRC( x_i, Refs (x_i, Assigns; Body_ext)-1);
      CR[[ Assigns; , y_1, ..., y_m = dummy( x_1, ..., x_n);
                      RETURN(y_1, ..., y_m); ]] ,
      FunAp( dummy, ...);
   } ELSE {
      DecRCFreeArray( w_i, 1);
   }
   FunRet(tag_1, y_1, ..., tag_m, y_m);
}
```

wobei $w_i \in \{ y_i \} \setminus \{ x_i \}$

und $Body\_ext = \begin{cases} y_1, \ldots, y_m = dummy(x_1, \ldots, x_n); \\ RETURN(y_1, \ldots, y_m); \end{cases}$.

Durch Rücktransformation in eine WHILE-Schleife erhält man schließlich

```
{ ...
  WHILE (Expr) {
      AdjustRC( x_i, Refs (x_i, Assigns; Body_ext)-1);
      CR[[ Assigns , y_1, ..., y_m = dummy( x_1, ..., x_n);
                     RETURN(y_1, ..., y_m); ]]
   };                                                      ,
   DecRCFreeArray( w_i, 1);
   AdjustRC( y_i, Refs (y_i, Rest; F)-1);
   CR[[ Rest; , F ]]
}
```

wobei $w_i \in \{ y_i \} \setminus \{ x_i \}$

und $Body\_ext = \begin{cases} y_1, \ldots, y_m = dummy(x_1, \ldots, x_n); \\ RETURN(y_1, \ldots, y_m); \end{cases}$.

Allgemein kann so die Compilation von WHILE-Schleifen formalisiert werden durch

$$\mathrm{CR}\left[\!\!\left[\begin{array}{l}\texttt{WHILE (}e\texttt{)\{}\\ \quad Ass;\\ \texttt{\};}\\ Rest\end{array}\;,\;\mathcal{F}\right]\!\!\right] \mapsto \left\{\begin{array}{l}\texttt{WHILE (}e\texttt{) \{}\\ \quad AdjustRC\texttt{(}\ x_i\texttt{,}\ Refs\ \texttt{(}x_i\texttt{,}\ Ass; Body\_ext\texttt{)-1);}\\ \quad \mathrm{CR}\left[\!\!\left[Ass;\ ,\ Body\_ext\right]\!\!\right]\\ \texttt{\};}\\ \textsc{DecRCFreeArray}\texttt{(}\ w_i\texttt{,}\ \texttt{1);}\\ AdjustRC\texttt{(}\ y_i\texttt{,}\ Refs\ \texttt{(}y_i\texttt{,}\ Rest\ \mathcal{F}\texttt{)-1);}\\ \mathrm{CR}\left[\!\!\left[Rest\ ,\ \mathcal{F}\right]\!\!\right]\texttt{;}\end{array}\right.$$

mit

$x_i \in \{x_1, ..., x_n\} = \mathrm{V}_{arg} = \mathrm{V}_{need}(Ass) \cup (\mathrm{V}_{def}(Ass) \cap \mathrm{V}_{need}(Rest))$,

$y_i \in \{y_1, ..., y_m\} = \mathrm{V}_{res} = (\mathrm{V}_{def}(Ass) \cap \mathrm{V}_{need}(Rest))$,

$w_i \in \mathrm{V}_{arg} \setminus \mathrm{V}_{res}$

und $Body\_ext = \begin{cases} y_1\texttt{, ..., }y_m\texttt{ = }dummy\texttt{(}\ x_1\texttt{, ...,}\ x_n\texttt{);} \\ \textsc{return}\texttt{(}y_1\texttt{, ..., }y_m\texttt{);}\end{cases}$.

In ähnlicher Weise lassen sich die Compilations-Regeln für die anderen Schleifenkonstrukte ableiten. Auf eine detailierte Herleitung soll hier verzichtet werden, eine formale Spezifikation der entsprechenden Compilations-Regeln findet sich im Anhang D.

4.3.5 Compilation der primitiven Array-Operationen

Die Compilation von primitiven Array-Operationen läßt sich in zwei Schritte zerlegen: die Erzeugung von C-Programmfragmenten zur Implementierung der Funktionalität der Operationen und das Einfügen der benötigten Anpassungen von Referenzzählern. Um ein einheitliches Schema für alle primitiven Funktionen - also auch für solche, die nicht auf Arrays angewendet werden - zu erhalten, verwenden wir für die Compilation der Funktionalität der Array-Operationen das entsprechend zu erweiternde C-Schema. Das Einfügen der ggf. benötigten Anpassungen von Referenzzählern bei Array-Argumenten bzw. Array-Resultaten sowie die Anwendung des C-Schemas können dann durch eine einzige Compilations-Regel des CR-Schemas für alle primitiven Funktionen erreicht werden:

$$\mathrm{CR}[\![v_1, \ldots, v_n = Prf(a_1, \ldots, a_m); \; Rest \, , \, \mathcal{F}]\!] \mapsto \begin{cases} \mathrm{C}[\![v_1, \ldots, v_n = Prf(a_1, \ldots, a_m);]\!] \\ AdjustRC(w_i, Refs(w_i, Rest; \mathcal{F})\texttt{-1}); \\ \mathrm{DECRCFREEARRAY}(b_i, 1); \\ \mathrm{CR}[\![Rest \, , \, \mathcal{F}]\!] \end{cases} ,$$

wobei $w_i \in \{ v_i \mid Basetype(TYPE(v_i)) \in \mathcal{T}_{Array} \}$
und $b_i \in \{ a_i \mid Basetype(TYPE(a_i)) \in \mathcal{T}_{Array} \}$.

Um die konkrete Implementierung der Array-Operationen flexibel zu gestalten, sollen auch hier ICM-Befehle als Zwischensprache verwendet werden. Sie sind im wesentlichen so konzipiert, daß jede Operation in genau einen ICM-Befehl übersetzt wird. Lediglich für stark überladene Operationen wie zB. die arithmetischen Operationen oder die PSI-Funktion, deren Implementierung je nach Argument-Typ stark variiert, existieren verschiedene ICM-Befehle, die jeweils nur einen Teil der möglichen Überladungen abdecken.

Zur Verdeutlichung der generellen Vorgehensweise wird an dieser Stelle nur die Compilation der PSI-Funktion dargestellt. Eine vollständige Darstellung der ICM-Befehle zur Realisierung der Array-Operationen sowie der zugehörigen Compilations-Regeln des C-Schemas finden sich im Anhang C bzw. im Anhang D.

Die PSI-Funktion liefert je nach Beschaffenheit der Argumente entweder einen skalaren Wert oder aber ein Teil-Array als Ergebnis. Eine Realisierung dieser Operation erfordert daher entweder einen einfachen Zugriff auf das Array oder aber das Allozieren von Speicher für einen Resultat-Array sowie das Kopieren eines ganzen Speicherbereichs. Da aufgrund des Typsystems statisch inferierbar ist, welcher dieser beiden Fälle jeweils vorliegt, entwerfen wir zwei verschiedene ICM-Befehle für die Realisierung der PSI-Funktion:

PSIVXA_S(*array*, *result*, *len_idx*, *idx_vect*) erzeugt eine Zuweisung des durch den Indexvektor *idx_vect* in *array* selektierten, skalaren Array-Elementes an die Variable *result*. *len_idx* spezifiziert die Dimensionalität von *array* bzw. die Länge des Zugriffsvektors *idx_vect*. Die Referenzzähler von *array* und *idx_vect* bleiben unverändert.

PSIVXA_A(*dim_array*, *array*, *result*, *len_idx*, *idx_vect*) erzeugt eine Zuweisung des durch *idx_vect* in *array* selektierten Teil-Arrays von *array* an die Variable *result*. Da es sich beim Resultat um ein Array handelt, werden zunächst jedoch Befehle zur Speicherallozierung für das Resultat sowie die Initialisierung des Referenzzähler mit dem Wert 1 erzeugt. *dim_array* spezifiziert die Dimensionalität von *array*, *len_idx* die Länge des Zugriffsvektors *idx_vect*. Die Referenzzähler von *array* und *idx_vect* bleiben unverändert.

Für die Compilation von Anwendungen der Funktion PSI ergibt sich als Erweiterung des C-Schemas

$$\mathrm{C}[\![\texttt{v = PSI(idx, array);}]\!] \mapsto \begin{cases} \text{PsiVxA\_S}(array, v, n, idx); & \text{falls } \begin{cases} n = \text{DIM}(array) \\ \wedge\ [n] = \text{SHAPE}(idx) \end{cases} \\ \text{PsiVxA\_A}(m, array, v, n, idx); & \text{falls } \begin{cases} m = \text{DIM(array)} \\ \wedge\ [n] = \text{SHAPE}(idx) \\ \wedge\ m \neq n \end{cases} \end{cases}.$$

4.3.6 Compilation der WITH-Konstrukte

die WITH-Konstrukte lassen sich durch Schleifen-Konstrukte in C realisieren. Als Beispiel dazu zunächst folgender SAC-Programmausschnitt:

```
...
v =0;
a =GENARRAY( [100,], 3);
b =WITH( [7] <= idx <= [42] ) {
       IF( PSI( [0], idx) < 17)
          v = v + 1;
       v = v * PSI( idx, a);
    }MODARRAY( idx, v, a);
...
```

Ein erster, naiver Ansatz zur Compilation dieses Ausschnittes ersetzt das WITH-Konstrukt durch den Aufruf einer neuen Funktion `dummy` mit den freien Variablen des WITH-Konstrukt-Rumpfes als Argumenten. Der Rumpf dieser Funktion besteht aus einer Zuweisung von `a` an `b` sowie nachfolgender sukzessiver Modifikation von `b` innerhalb einer WHILE-Schleife

```
...
v = 0;
a = GENARRAY( [100,], 3);
b = dummy( v, a);
...
```

```
INT[100] dummy( INT v, INT[100] a)
{ b = a;
  idx= [7];
  WHILE( idx <= [42]) {
    IF( PSI( [0], idx) < 17)
      v = v+1;
    v = v * PSI( idx, a);
    b = MODARRAY( idx, v, b);
    idx ++;
  }
  RETURN(b);
}
```

.

Das Problem dieses Ansatzes liegt bei den Variablen, die im Rumpf des WITH-Konstruktes zunächst frei vorkommen und denen später dann ein anderer Wert zugewiesen wird (`v` in unserem Beispiel). Durch die Abbildung in eine Schleife sorgt eine solche Zuweisung für einen unerwünschten Seiteneffekt. Um dies zu vermeiden, bedarf es der Umbenennung dieser Variablen bei allen Zuweisungen im Schleifenrumpf. Auf diese Weise wird in dem Beispiel `v = v+1` zu `new_v = v+1`. Eine solche Umbenennung von Variablen führt jedoch zu Problemen bei der Compilation von IF-THEN-ELSE-Konstrukten, falls sich nur in einem Zweig eine solche Zuweisung befindet und die Variable im nachfolgenden Programm referenziert wird. In obigem Beispiel tritt dieses Problem bei der Zuweisung `v = v * PSI( idx, a)` auf. Ist das Prädikat des IF-THEN-ELSE-Konstruktes wahr, so muß das `v` auf der rechten Seite umbenannt werden, ansonsten nicht. Einen Ausweg ohne Programmteile duplizieren zu müssen, bietet hier eine Umbenennung der Variablen vor dem IF-THEN-ELSE-Konstrukt bzw. allgemein am Anfang der Schleife:

```
...
v = 0;
a = GENARRAY( [100,], 3);
b = dummy( v, a);
...
```

```
INT[100] dummy( INT v, INT[100] a)
{ b = a;
  idx = [7];
  WHILE( idx <= [42]) {
    new_v = v;
    IF( PSI( [0], idx) < 17)
      new_v = new_v+1;
    new_v =new_v * PSI( idx, a);
    b = MODARRAY( idx, new_v, b);
    idx ++;
  }
  RETURN(b);
}
```

.

Diese Lösung ist zwar korrekt im Sinne der in Abschnitt 3.3.4 definierten Semantik,

aus Laufzeiteffizienz-Überlegungen heraus soll jedoch ein expliziter Funktionsaufruf vermieden werden. Deshalb wird ein sog. Inlining der Funktion dummy vorgenommen und der Funktionsaufruf auf der linken Seite durch den Rumpf der Funktion auf der rechten Seite ersetzt. Dabei ist zu beachten, daß lokale Variablen der Funktion dummy umbenannt werden müssen, falls sie im das WITH-Konstrukt umgebenden Programm vorkommen. Für das Beispiel ergibt sich damit:

```
...
v = 0;
a = GENARRAY( [100,], 3);
b = a;
new_idx = [7];
WHILE( new_idx <= [42]) {
  new_v = v;
  IF( PSI( [0], new_idx) < 17)
    new_v = new_v+1;
  new_v = new_v * PSI( new_idx, a);
  b = MODARRAY( new_idx, new_v, b);
  new_idx ++;
}
...
```

Da das entstandene Programmfragment wiederum syntaktisch korrektes SAC ist, ist eine Realisierung der Compilation durch eine Hochsprachen-Transformation denkbar. Im Gegensatz zu einer expliziten Hochsprachen-Transformation bietet jedoch eine Realisierung der Schleife durch ICM-Befehle die Möglichkeit, weitere spezielle Optimierungen bei der Übersetzung nach C vorzunehmen. Ansatzpunkte dafür sind beispielsweise die Array-Zuweisung b=a sowie die Anwendung der MODARRAY-Operationen. Deshalb werden folgende ICM-Befehle zur Realisierung der WITH-Konstrukte mit MODARRAY-Operation entworfen:

BEGINMODARRAY(*res*, *dim_res*, *src*, *start*, *stop*, *idx*, *idx_len*) erzeugt eine Schachtelung von WHILE-Schleifen, um ein WITH-Konstrukt mit MODARRAY-Operation zu realisieren. Dabei variiert die Generatorvariable *idx* mit *idx_len* Komponenten von *start* bis *stop* ohne Berücksichtigung der Referenzzähler dieser drei Vektoren. Außerdem wird eine Initialisierung des Speichers des *dim_res*-dimensionalen Resultat-Arrays *res* für alle Indexpositionen kleiner *start* mit den entsprechenden Array-Elementen von *src* vorgenommen. Sowohl der Referenzzähler von *src* als auch der Zähler von *res* bleiben dabei unverändert.

ENDMODARRAYS(*res*, *dim_res*, *src*, *val*) stellt eines der möglichen Gegenstücke zu BEGINMODARRAY dar. Dazu wird zunächst der im Rumpf des WITH-Konstruktes erzeugte skalare Wert *val* an die aktuelle Indexposition des Arrays *res*

mit der Dimensionalität *dim_res* kopiert und anschließend werden die durch BEGINMODARRAY erzeugten Schleifenkonstrukte beendet. Schließlich werden noch die fehlenden Initialisierungen für die Indexpositionen größer *stop* (aus dem zug. BEGINMODARRAY-Konstrukt) des Resultat-Arrays *res* durch Kopieren der entsprechenden Elemente von *src* vorgenommen. Die Referenzzähler aller involvierten Arrays bleiben dabei unverändert.

ENDMODARRAYA(*res*, *dim_res*, *src*, *val*, *idx_len*) entspricht im wesentlichen dem EndModarrayS-Befehl. Der Unterschied besteht ausschließlich darin, daß es sich bei *val* um ein Array der Dimension (*dim_res* − *idx_len*) handelt, dessen Referenzzähler nach dem Kopieren der Werte in das Array *res* um 1 erniedrigt wird, was ggf. zur Freigabe des Speichers von *val* führt.

Die entsprechenden ICM-Befehle für die anderen Varianten des WITH-Konstruktes finden sich in Anhang C. Um unser Beispiel mit Hilfe obiger ICM-Befehle übersetzen zu können, bedarf es zweier Anpassungen: Zum einen müssen neue Variablen für die Schranken des Generatorteils des WITH-Konstruktes eingeführt werden, da BEGINMODARRAY Variablen erwartet. Zum anderen müssen sowohl für b als auch für new_idx Speicherbereiche alloziert und am Ende des Compilates die Referenzzähler angepaßt werden:

```
...
CR[[ v = 0;                                                          ]]
  [[ a = GENARRAY( [100,], 3);   , b = dummy(v,a,new_start,new_stop); ]]
  [[ new_start= [7];             , ...                               ]]
  [[ new_stop= [42];                                                  ]]
ALLOCARRAY(int, b, 1);
ALLOCARRAY(int, new_idx, 1);
BEGINMODARRAY( b, 1, a, new_start, new_stop, new_idx, 1);
    CR[[ new_v = v;                                                  ]]
      [[ IF( PSI( [0], new_idx) < 17)                                ]]
      [[    new_v = new_v+1;            , RETURN(new_v);             ]]
      [[ new_v= new_v * PSI( new_idx, a);                            ]]
ENDMODARRAYS( b, 1, a, new_v);
DECRCFREEARRAY( idx, 1);
DECRCFREEARRAY(a, 1);
DECRCFREEARRAY( new_start, 1);
DECRCFREEARRAY( new_stop, 1);
...
```

Dies läßt sich für den allgemeinen Fall formalisieren durch:

$$
\mathrm{CR}\left[\!\!\left[\begin{array}{l} res = \text{WITH}(\ e_1 \texttt{ <= } idx \texttt{ <= } e_2\ ) \\ \qquad\quad \{\ Assigns;\ \} \\ \quad \text{MODARRAY}(\ idx,\ val,\ array);\\ Rest; \end{array} \right.\right. \quad , \mathcal{F} \left.\!\!\right]\!\!\right]
$$

$$
\mapsto \left\{ \begin{array}{l}
\mathrm{CR}\left[\!\!\left[\begin{array}{ll} v_1 = e_1; & array_2 = dummy(\ w_1,\ ..,\ w_n);\\ v_2 = e_2; & Rest;\\ & \mathcal{F} \end{array}\right.\!\!\right] \\
\text{ALLOCARRAY}(\ idx,\ 1);\\
\text{BEGINMODARRAY}(\ res,\ dim\_res,\ array,\ v_1,\ v_2,\ idx,\ len\_idx\ );\\
\quad \text{INCRC}(\ idx,\ Refs\ (idx,\ Assigns;\ \text{RETURN}(val);));\\
\quad \text{INCRC}(\ w_i,\ Refs\ (w_i,\ Assigns;\ \text{RETURN}(val);));\\
\quad \mathrm{CR}[\![Assigns;\ ,\ \text{RETURN}(val);]\!]\\
\text{ENDMODARRAYS}(\ res,\ dim\_res,\ array,\ val\ )^1;\\
\text{DECRCFREEARRAY}(\ idx,\ 1);\\
\text{DECRCFREEARRAY}(\ w_i,\ 1);\\
\mathrm{CR}[\![Rest\ ,\ \mathcal{F}]\!]
\end{array} \right. ,
$$

wobei $\{w_1, ..., w_n\} = \mathrm{V}_{need}(Assigns; \text{RETURN}(val);) \cup \{v_1, v_2, a\} \setminus \{idx\}$.

4.4 Das Modulsystem

Anstelle eines vollständigen Modulsystems bietet C lediglich rudimentäre Möglichkeiten zur Modularisierung. Die für ein C-Programm benötigten Typ- und Funktionsdefinitionen können sich über mehrere Dateien erstrecken. Alle so definierten Symbole befinden sich jedoch konzeptuell in einem Namensraum. Wird ein Symbol in mehreren Dateien definiert, so steht nur eine der Definitionen zur Verfügung, selbst wenn sich die Definitionen unterscheiden. Um Funktionen benutzen zu können, die sich in anderen Dateien befinden, bedarf es sog. EXTERN-Deklarationen. Sie werden durch das Schlüsselwort EXTERN gekennzeichnet und bestehen aus dem Namen der Funktion sowie der Spezifikation von Argument- und Resultattypen. Für sämtliche in einer Datei verwendeten, nicht-elementaren Typen müssen die Typdefinitionen vorliegen. Die einzige Möglichkeit, Typdefinitionen zu verbergen stellen die sog. VOID-Zeiger dar. Ein Zugriff auf die hinter einem VOID-Zeiger liegende Datenstruktur ist nur mittels einer expliziten Typannotation, dem sog. cast, möglich. Auf dieser Basis lassen sich die einzelnen Dateien eines C-Programmes separat compilieren und schließlich mittels eines speziellen Bindeprogrammes, dem sog. linker, zu einem ausführbarem Maschinenprogramm zusammenfassen.

Die Abbildung des Modulsystems von SAC auf die Modularisierungsmöglichkei-

[1] ENDMODARRAYA(*res*, *dim_res*, *array*, *val*, *len_idx*) falls *val* ein Array ist.

ten von C ist damit vorgezeichnet. Zunächst muß aus den IMPORT-Anweisungen des Programmes sowie den IMPORT-Anweisungen der Moduldeklarationen ermittelt werden, welche Symbole genau benötigt werden und in welchem Modul sie definiert sind. Dann müssen für alle importierten Funktionen EXTERN-Deklarationen erzeugt werden. Um die in SAC existierende Trennung der Namensräume bei Symbolen aus verschiedenen Modulen in C zu modellieren, erhalten alle Funktionsnamen den Namen des Moduls, in dem sie definiert sind, sowie ein nicht eingebbares Zeichen als Prefix. Schließlich werden für alle importierten Typen Typdefinitionen erzeugt. Wie bei Funktionen werden auch die Typsymbole umbenannt, um gleichnamige Symbole in verschiedenen Modulen zu ermöglichen. Da für implizite Typen keine Typdefintion vorliegt, werden diese in VOID-Zeiger abgebildet.

Das Hauptproblem der Abbildung des SAC-Modulkonzeptes nach C liegt in der Bestimmung der Menge der zu importierenden Symbole sowie der Menge der zugehörigen Module, in denen sie definiert sind. Die Ursachen für die Probleme liegen in der Flexibilität der Spezifizierbarkeit von Importen, nämlich:

1. der Möglichkeit, importierte Symbole exportieren zu können. Dadurch werden Symbole oft aus Modulen importiert, in denen sie gar nicht definiert sind;

2. der Möglichkeit, sämtliche von einem Modul exportierten Symbole mittels der ALL-Anweisung zu importieren. Da dies im allgemeinen wiederum importierte Symbole umfaßt, zieht ein ALL-Import in der Regel weitere Importe aus anderen Modulen nach sich;

3. der Möglichkeit, wechselseitig rekursive Importe zu spezifizieren, was zu Terminierungsproblemen bei der Bestimmung der benötigten Importe führt.

Ein Algorithmus zur Inferenz der importierten Symbole mitsamt den zugehörigen Modulen, in denen sie definiert sind, ist in Abb. 4.2 dargestellt. Er beschreibt, wie aus einer Menge von SAC-Import-Anweisungen (IMPORTSET) eine Menge von Symbol-Modulname-Paaren (SYMBS) gewonnen werden kann, die alle zu importierenden Symbole nebst zugehörigem Definitions-Modul umfaßt. Um eine Terminierung des Algorithmus gewährleisten zu können, bedarf es außerdem einer Menge von Modulnamen (ALLSET), in der alle mit der ALL-Anweisung importierten Module vermerkt werden. Nach der Initialisierung dieser Mengen (Abb. 4.2 1.) werden sukzessive alle Importe aus IMPORTSET aufgelöst.

Importe mit ALL-Anweisung werden entweder ignoriert, falls der entsprechende Modulname bereits in ALLSET enthalten ist, oder durch die Informationen der zugehörigen Moduldeklaration ersetzt: moduleigene Symbole werden als Symbol-Modulname-Paare in SYMBS ergänzt, importierte Symbole als zusätzlich aufzulösende Importe in IMPORTSET übernommen. Zur Beschreibung dieses Vorganges werden zwei Funktionen verwendet (vergl. Abb. 4.2 2.):

OWNSYMBS(*Modulname*) selektiert die als OWN deklarierten Symbole aus der Moduldeklaration des Moduls *Modulname*.

1. Initialisiere SYMBS mit $\emptyset$, ALLSET mit $\emptyset$ und IMPORTSET mit den Import-Anweisungen des zu compilierenden Programmes.

2. Für alle Import-Anweisungen der Form IMPORT x:ALL; aus IMPORTSET:

   - Entferne IMPORT x:ALL; aus der Menge IMPORTSET.
   - Falls $x \notin$ ALLSET:
     - Ergänze ALLSET um x.
     - Vereinige SYMBS mit $\bigcup_{S \in \text{OWNSYMBS}(x)} \{< x : S >\}$.
     - Vereinige IMPORTSET mit IMPORTS(x).

3. Für alle Import-Anweisungen der Form IMPORT x:y; aus IMPORTSET mit y $\neq$ ALL und für alle Symbole S $\in$ DEFSYMBS(y):

   - Entferne S aus der Import-Liste y.
   - Berechne die Menge der in Frage kommenden Module mit Definitionen für S $Defs$:= FINDDEFMODS(S, x, $\emptyset$).
   - Falls $Defs = \{x'\}$:
     - Vereinige SYMBS mit $\{< x' : S >\}$.

     sonst:
     - Abbruch und Fehlermeldung

4. Wiederhole 2. und 3., bis IMPORTSET = $\emptyset$.

Abbildung 4.2: Inferenz der Menge der zu importierenden Symbole

IMPORTS(*Modulname*) selektiert die in der Moduldeklaration des Moduls *Modulname* deklarierten Importe.

Selektive Importe werden Symbol für Symbol aus IMPORTSET entfernt und das zugehörige Symbol-Modulname-Paar in SYMBS ergänzt. Um die Menge der in der Import-Liste eines selektiven Imports angegebenen Symbole zu bestimmen, bedient man sich einer weiteren Funktion DEFSYMBS. Die Bestimmung des Moduls, in dem ein aus Modul x importiertes Symbol S letztendlich definiert ist, bedarf einer vierten Funktion FINDDEFMODS:

```
FINDDEFMODS( Symb, Mod, TestedMods)
= IF Mod ∈ TestedMods
  THEN ∅
  ELSE IF Symb ∈ OWNSYMBS(Mod)
       THEN { Mod }
       ELSE LET
              PossMods = {x | IMPORT x:y; ∈ IMPORTS(Mod)
                              ∧ ( y = ALL ∨ Symb ∈ DEFSYMBS(y) )}
            IN  ⋃_{x∈PossMods}  FINDDEFMODS( Symb, x, TestedMods ∪ {Mod})
```

Ist das Symbol Symb in dem Modul Mod selbst definiert, so wird die Suche beendet. Ansonsten wird die Suche auf alle in Frage kommenden Importe ausgedehnt und die Ergebnisse in einer Menge zusammengefaßt. Um Terminierungsprobleme durch rekursive Importe zu vermeiden, verwenden wir auch hier eine Menge, die alle bereits untersuchten Module enthält (TESTEDMODS).

Die nach der Bestimmung der benötigten Symbole sowie deren Herkunft erfolgende Umsetzung von Moduldeklarationen in EXTERN-Deklarationen sowie Typdefinitionen läßt sich folgendermaßen algorithmisieren:

Sei $< Mod : Symb >$ ein durch den Algorithmus aus Abb. 4.2 gewonnenes Modulname-Symbol-Paar sowie *SymbDecl* die zugehörige Symboldeklaration aus der Moduldeklaration des Moduls *Mod*. Dann wird in das Programm eingefügt:

- EXTERN C$[\![\tau_1 \ \ldots \ \tau_m \ Mod_Symb(\ \sigma_1 \ a_1, \ \ldots, \ \sigma_n \ a_n)]\!]$
  falls SymbDecl eine Funktionsdeklaration mit $SymbDecl \equiv \tau_1$ ... τ_m Symb($\sigma_1 \ a_1$, ..., $\sigma_n \ a_n$); ist.

- TYPEDEF τ *Symb*;
  falls *SymbDecl* ein expliziter Typ mit $SymbDecl \equiv Symb = \tau$; ist.

- TYPEDEF VOID* *Symb*;
  falls *SymbDecl* ein impliziter Typ ist.

Durch diese Art der direkten Umsetzung des Modulsystems von SAC nach C sind vordergründig alle in Abschnitt 3.4 für das Modulsystem entworfenen Designziele wie „separate compilation“ oder „information hiding“ erreicht. Eine nähere Betrachtung zeigt jedoch, daß diese Art der Umsetzung von Modulen bei vielen Beispielen dazu führt, daß ein in Module zerlegtes Programm gegenüber einem nicht-modularisierten Programm zu Code mit erheblich schlechterem Laufzeitverhalten compiliert wird. Für dieses Problem lassen sich drei Ursachen identifizieren:

- Die Definition von impliziten Typen führt durch die Behandlung als VOID-Zeiger zu unnötigen Indirektionen. Dies macht sich insbesondere bei den primitiven Typen INT, FLOAT, ... bemerkbar.

- Function-Inlining bringt über Modulgrenzen hinaus keine Laufzeitvorteile mehr, da der Rumpf der Funktion dem importierenden Programm nicht zur Verfügung steht.

- Der Import von dimensionsunabhängig spezifizierten Funktionen beschränkt die Möglichkeiten des Typsystems erheblich, da aufgrund des nicht bekannten Funktionsrumpfes keine Spezialisierungen mehr inferiert werden können.

Durch die bisher vorgestellte Compilation von Modulen entsteht somit nicht nur ein Konflikt zwischen den Designzielen Modularisierung und Laufzeiteffizienz, sondern auch zwischen der dimeniionsunabhängigen Spezifikation und der Modularisierung. Diese Konflikte können nur dadurch aufgelöst werden, daß dem importierenden Programm die benötigten Informationen über Funktionsrümpfe bzw. Typdefinitionen zur Verfügung gestellt werden. Um trotzdem das Designziel „information hiding" zu erreichen, werden diese Informationen bei der Compilation von Modulen in sog. SAC-Information-Blöcke, kurz SIB genannt, abgelegt. Dabei reicht es im allgemeinen nicht aus, nur die direkt benötigten Funktionsrümpfe und Typdefinitionen über den SIB zur Verfügung zu stellen, da diese Ausdrücke wiederum importierte Symbole benutzen können. Die dafür benötigten Deklarationen sind ebenfalls Bestandteil des SIB.

Die Einführung des SIB erfordert eine entsprechende Erweiterung der Umsetzung von Moduldeklarationen in Funktionsdeklarationen und Typdefinitionen:

- Für Funktionssymbole gilt: Existiert die Funktion im zugehörigen SIB, so wird sie literal übernommen. Andernfalls wird eine EXTERN-Deklaration erzeugt. Falls eine dem SIB entnommene Funktion weitere Importe benötigt, werden auch diese direkt aus dem SIB übernommen.

- Die Handhabung expliziter Typen bleibt unverändert.

- Für implizite Typen gilt: Sämtliche impliziten Typen werden dem SIB entnommen; es werden keine Typdefinitionen auf VOID-Zeiger erzeugt. Genauso wie bei den Funktionen kann es auch hier zu der Übernahme weiterer Deklarationen aus dem SIB kommen.

4.5 Klassen und Objekte

Die Compilation von Klassen sowie von Programmen, die Klassen importieren, kann im wesentlichen auf die Compilation von Modulen bzw. Modul-Importen zurückgeführt werden. Lediglich die Compilation von Klassen-Typen sowie von Objekten solchen Typs muß ergänzt werden.

Der Unterschied zwischen einem Typ τ als Klassen-Typ und einem benutzerdefinierten Typ τ besteht ausschließlich darin, daß Variablen vom Klassen-Typ nur in

restringierter Form verwendet werden dürfen. Ein formaler Parameter/eine Variable a eines Klassen-Typs τ darf maximal einmal in seinem/ihrem Bindungsbereich bzw. maximal einmal pro Zweig evtl. vorhandener IF-THEN-ELSE-Konstrukte referenziert werden; d.h. es gilt: $Refs\ (a,\ Fun\_Body) \leq 1$ bzw. $Refs\ (a,\ Rest\_of\_Fun\_Body) \leq 1$ für jede Zuweisung an a innerhalb des zugehörigen Funktionsrumpfes *Fun_Body*. Dies birgt zwar die Möglickeit, sämtliche Modifikationen einer a zugewiesenen Datenstruktur destruktiv vorzunehmen, erlaubt ansonsten jedoch eine Compilation der Klassentypen wie implizite Typen einer Uniqueness-Eigenschaft. Für Objekte mit atomarem Typ (INT, BOOL, ..) ist dies sicherlich auch sinnvoll, bei SAC-Arrays führt es jedoch zu einer uneffizienten Implementierung. Da aufgrund der Uniqueness-Eigenschaft der Referenzzähler von Objekten ohnehin nur die Werte 0 oder 1 annehmen kann, was einer Allozierung bzw. Freigabe dieser Strukturen gleichkommt, kann auf seine Implementierung verzichtet werden. Die Aufgabe der Integration von Klassen und Objekten zerfällt damit in folgende Teilaufgaben:

- Behandlung der Klassen-Typen als implizite Typen.
- Überprüfen der Uniqueness-Eigenschaft.
- Einführung einer weiteren Darstellung für SAC-Arrays als Datenstruktur ohne Referenzzähler.
- Realisierung der Konvertierungsfunktionen zwischen den beiden verschiedenen Array-Darstellungen.

Da die Implementierung auf der der Module aufbaut, sind die Definitionen aller impliziten Typen einer Klassendefinition über den SIB verfügbar. Die Erstellung des SIB bei der Compilation von Klassen muß also um die Definition der Klassentypen ergänzt werden. Importiert ein Programm eine Klasse *class*, so wird dem entsprechenden SIB die Definition des Klassentyps CLASSTYPE τ; entnommen und dem importierenden Programm als Typdefinition der Form TYPEDEF τ *class_class*; zur Verfügung gestellt.

Um bei der Überprüfung der Uniqueness-Eigenschaft sowie der späteren Compilation wieder zwischen Objekten und anderen Datenstrukturen unterscheiden zu können, muß compiler-intern weiterhin zwischen Typen mit und ohne Uniqueness-Eigenschaft unterschieden werden. Für die Typinferenzphase ist dies jedoch bis auf die Ergänzung der generischen Konversionsfunktionen FROM_*class* und TO_*class* ohne Belang.

Die Sicherstellung der Uniqueness-Eigenschaft kann mit dem gleichen Mechanismus erfolgen, der für die Inferenz der benötigten Anpassungen der Referenzzähler von SAC-Arrays benötigt wird, da sie beide auf der *Refs*-Funktion aus Def. 3.1.5 beruhen. Er muß lediglich dahingehend erweitert werden, daß er auch für Objekte von skalarem Typ anwendbar ist.

Für die Darstellung von Arrays ohne Referenzzählung sowie für die Konversion von Arrays mit Referenzzählung in solche ohne Referenzzählung und zurück führen wir folgende neue ICM-Befehle ein:

DECLUNQARRAY(*name*, τ, s_1, ..., s_n) erzeugt die Variablendeklarationen, die benötigt werden, um in C eine SAC-Array-Variable *name* mit dem Typ $\tau[s_1, ..., s_n]$ ohne Referenzzähler darzustellen.

MAKEUNQARRAY(*res*, *src*) erzeugt C-Befehle, die aus dem SAC-Array mit Referenzzählung *src* ein SAC-Array ohne Referenzzählung machen und dieses dann der Variablen *res* zuweisen. Falls der Referenzzähler von *src* 1 ist, kann einfach der Zeiger auf das Array zugewiesen und der Referenzzähler von *src* eliminiert werden. Ansonsten muß Speicher für ein neues Array ohne Referenzzähler alloziert und die einzelnen Array-Elemente kopiert werden.

FREEUNQARRAY(*name*) erzeugt Befehle zum Freigeben eines Arrays ohne Referenzzähler *name*.

ALLOCRC(*name*, *n*) erzeugt Befehle zum Allozieren des Speichers für einen Referenzzähler, assoziiert diesen mit dem Array *name* und initialisiert ihn mit dem Wert *n*.

Damit können jetzt die Compilations-Schemata entsprechend erweitert werden. Variablendeklarationen werden für Arrays ohne Referenzzählung in den DECLUNQ-ARRAY-ICM übersetzt:

$$\mathrm{C}[\![\tau\ v;\ ]\!] \mapsto \begin{cases} \tau\ v; & \text{falls } \mathit{Basetype}(\tau) \in \mathcal{T}_{\mathit{Simple}} \\ \textsc{DeclArray}(v, \sigma, s_1, \ldots, s_n); & \text{falls } \mathit{Basetype}(\tau) = \sigma[s_1, ..., s_n] \wedge \neg\, \mathit{UNQ}(\tau) \\ \textsc{DeclUnqArray}(v, \sigma, s_1, \ldots, s_n); & \text{falls } \mathit{Basetype}(\tau) = \sigma[s_1, ..., s_n] \wedge\ \mathit{UNQ}(\tau) \end{cases}.$$

Das Prädikat *UNQ* zeigt dabei an, ob es sich um einen Klassentyp handelt oder nicht. Die Erzeugung von Arrays ohne Referenzzählung erfolgt ausschließlich über die generischen Konversionsfunktionen TO_*class*. Sie wird daher im Falle eines Array-Types als Basistyp in den MAKEUNQARRAY-ICM und ansonsten direkt in eine Zuweisung compiliert:

$$\mathrm{CR}[\![v \texttt{ = } \textsc{to}\_class(w)\texttt{;}\ Rest\ ,\ \mathcal{F}]\!]$$
$$\mapsto \begin{cases} \textsc{MakeUnqArray}(\ w,\ v\ )\texttt{;} \\ \mathrm{CR}[\![Rest\ ,\ \mathcal{F}]\!] & \text{falls } Basetype(TYPE(w)) \in \mathcal{T}_{Array} \\ & \\ v \texttt{ = } w\texttt{;} \\ \mathrm{CR}[\![Rest\ ,\ \mathcal{F}]\!] & \text{sonst} \end{cases} .$$

In gleicher Weise wird auch bei den korrespondierenden Konversionsfunktionen FROM-*_class* verfahren. Bei Arrays muß in diesem Fall ein Referenzzähler erzeugt und entsprechend initialisiert werden:

$$\mathrm{CR}[\![v \texttt{ = } \textsc{from}\_class(w)\texttt{;}\ Rest\ ,\ \mathcal{F}]\!]$$
$$\mapsto \begin{cases} v \texttt{ = } w\texttt{;} \\ \textsc{AllocRC}(\ v,\ Refs\ (v,\ Rest;\ \mathcal{F}))\texttt{;} & \text{falls } \begin{array}{c} Basetype(TYPE(w)) \\ \in \mathcal{T}_{Array} \end{array} \\ \mathrm{CR}[\![Rest\ ,\ \mathcal{F}]\!] & \\ & \\ v \texttt{ = } w\texttt{;} \\ \mathrm{CR}[\![Rest\ ,\ \mathcal{F}]\!] & \text{sonst} \end{cases} .$$

Die Anpassung von Referenzzählern, wie sie nach Zuweisungen, am Beginn von Funktionsrümpfen und von Schleifen durch *AdjustRC* aus Abschnitt 4.3.1 erfolgt, beschränkt sich bei Arrays mit Uniqueness-Eigenschaft auf Werte von 0 bzw. -1 für n. Da im letzteren Fall das Array freigegeben werden muß, ergibt sich als Erweiterung zu der Definition in Abschnitt 4.3.1

$$AdjustRC(var, n)$$
$$:= \begin{cases} \textsc{DecRCFreeArray}(\ var,\ n)\texttt{;} & \text{falls } n < 0 \wedge \neg UNQ(TYPE(var)) \\ \textsc{FreeUnqArray}(\ var)\texttt{;} & \text{falls } n < 0 \wedge UNQ(TYPE(var)) \\ \texttt{/* no RC adjusments */} & \text{falls } n = 0 \\ \textsc{IncRC}(\ var,\ n)\texttt{;} & \text{falls } n \geq 0 \end{cases} .$$

Schließlich muß die Paramterübergabe beim Aufruf benutzerdefinierter Funktionen für Arrays ohne Referenzzählung angepaßt werden. Dazu reicht es aus, bei Array-Parametern ohne Referenzzählung statt der IN_RC-Marke die IN-Marke zu verwenden. Die entsprechenden formalen Ergänzungen der Compilations-Regeln sind dem Anhang D zu entnehmen.

4.5.1 CALL-BY-REFERENCE-Parameter

Konzeptuell sind CALL-BY-REFERENCE-Parameter in SAC lediglich notationelle Abkürzungen. Sämtliche CALL-BY-REFERENCE-Parameter eines Programmes lassen sich mit

Hilfe des Transformations-Schemas $\mathcal{TF}_R$ durch Hinzufügen zusätzlicher Rückgabewerte eliminieren (vergl. Abschnitt 3.5). Eine Implementierung dieser Transformation als Bestandteil des Compilers ermöglicht dadurch auch die Compilation von SAC-Programmen, die CALL-BY-REFERENCE-Parameter enthalten. Die auf diese Weise entstehenden C-Programme entsprechen jedoch nicht denen, die man bei einer direkten Compilation erwarten würde. Hierzu soll ein Beispiel betrachtet werden.

(a)

```
BOOL IncCtr( Ctr & cnt)
{
  f_cnt = FROM_Ctr(cnt);
  IF (f_cnt == MAX_INT)
    res = FALSE;
  ELSE {
    f_cnt++;
    res = TRUE;
  }
  cnt = TO_Ctr(f_cnt);

  RETURN( res );
}

b = IncCtr( mycnt);
```

$\Downarrow \mathcal{TF}_R$

(b)

```
BOOL, Ctr IncCtr( Ctr cnt)
{
  f_cnt = FROM_Ctr(cnt);
  IF (f_cnt == MAX_INT)
    res = FALSE;
  ELSE {
    f_cnt++;
    res = TRUE;
  }
  cnt = TO_Ctr(f_cnt);

  RETURN( res, cnt );
}

b, mycnt = IncCtr( mycnt);
```

$\Rightarrow$ C[[ ]]

(c)

```
BOOL IncCtr( Ctr cnt, Ctr * cnt_ptr)
{
  f_cnt = cnt;
  IF (f_cnt == MAX_INT)
    res = FALSE;
  ELSE {
    f_cnt++;
    res = TRUE;
  }
  cnt = f_cnt;
  *cnt_ptr = cnt;
  RETURN( res );
}

b = IncCtr( mycnt, &mycnt);
```

$\Uparrow$Merge Parameters

(d)

```
BOOL IncCtr( Ctr * cnt_ptr)
{ cnt = *cnt_ptr;
  f_cnt = cnt;
  IF (f_cnt == MAX_INT)
    res = FALSE;
  ELSE {
    f_cnt++;
    res = TRUE;
  }
  cnt = f_cnt;
  *cnt_ptr = cnt;
  RETURN( res );
}

b = IncCtr( &mycnt);
```

Abbildung 4.3: Beispiel-Compilation mit CALL-BY-REFERENCE-Parameter

In Abb. 4.3(a) sehen wir die Definition einer Funktion `IncCtr`, die z.B. Bestandteil einer Klasse für einen primitiven Zähler sein könnte. Der Klassentyp `Ctr` sei INT.

Die Funktion `IncCtr` erhöht nun den als Referenzparamteter übergebenen Parameter `cnt`, falls dieser nicht schon maximal ist. Als Rückgabewert liefert `IncCtr` einen Wahrheitswert, der anzeigt, ob die Operation ausgeführt werden konnte oder nicht. Die Konversionen von `cnt` in *f_cnt* und zurück sind erforderlich, da sämtliche primitiven Funktionen in SAC ausschließlich auf Typen ohne Uniqueness-Eigenschaft definiert sind. Direkt unter der Funktionsdefinition befindet sich eine Beispielanwendung: Es soll ein Zähler `mycnt` erhöht und der Wahrheitswert, welcher anzeigt, ob die Operation ausgeführt werden konnte, einer Variablen `b` zugewiesen werden.

Die Anwendung des Transformations-Schemas $\mathcal{TF}_R$ zur Eliminierung der CALL-BY-REFERENCE-Parameter liefert die in Abb. 4.3(b) dargestellte Funktionsdefinition bzw. -anwendung. Sie unterscheiden sich von den ursprünglichen Ausdrücken lediglich durch einen zusätzlichen Rückgabewert vom Typ `Ctr`. Durch Compilation dieser Ausdrücke mittels der Compilations-Schemata sowie Expansion der entstehenden ICM-Befehle erhält man das C-Programm aus Abb. 4.3(c). Dadurch, daß `cnt` sowohl formaler Parameter als auch Rückgabe-Wert ist, entsteht im Compilat ein zweiter formaler Parameter. Er enthält die Adresse des Zählers, um die Rückgabe des veränderten Wertes zu implementieren. Aufgrund ihres Auftretens sowohl in der Definition als auch in der Anwendung können die beiden Parameter zu einem zusammengefaßt werden, was zu der Darstellung in Abb. 4.3(d) führt. Dadurch wird anstelle eines zusätzlichen Parameters lediglich eine Zuweisung `cnt = *cnt_ptr;` am Beginn des Funktionsrumpfes erforderlich, was insbesondere bei rekursiven Funktionen zu Laufzeitgewinnen führt. Eine Übertragung dieser Optimierung auf den allgemeinen Fall, d.h. ohne Wissen darüber, ob diese Parameter aus einem CALL-BY-REFERENCE-Parameter entstanden sind oder nicht, bereitet jedoch Probleme. Dies ist dadurch begründet, daß sichergestellt sein muß, daß alle Funktionsaufrufe von der Form `IncCtr( x, &x)` sind. Dies kann aber nur dann auch über Modulgrenzen hinaus garantiert werden, wenn die beiden Parameter aus einem CALL-BY-REFERENCE-Parameter entstanden sind. Deshalb sollte bei der Implementierung der Transformation $\mathcal{TF}_R$ die ursprüngliche Form nachgehalten werden, um schließlich - zumindest konzeptuell - direkt aus der Version mit CALL-BY-REFERENCE-Parametern Abb. 4.3(a) das Compilat mit zusammengefaßten formalen Parametern Abb. 4.3(d) erzeugen zu können.

Dazu werden nochmals die ICM-Befehle FUNDEC sowie FUNAP um einen neuen Typ von Marken erweitert: die INOUT-Marke. Sie zeigt an, daß eine entsprechende Zuweisung am Beginn des Funktionsrumpfes erzeugt werden muß bzw. daß bei der Funktionsanwendung der Adress-Operator & zu verwenden ist. Die vollständige Definition dieser ICM-Befehle findet sich in Anhang C.

Die zugehörigen formalen Ergänzungen der Compilations-Regeln für Funktionsdefinitionen und -anwendungen um CALL-BY-REFERENCE-Parameter ergeben sich wie folgt:

Bei Funktionsdefinitionen wird für CALL-BY-REFERENCE-Parameter grundsätzlich die Marke INOUT eingeführt:

$$
\mathrm{C}\left[\!\!\left[\begin{array}{l} Tr_1,\ \ldots,\ Tr_n\ FunId(\ Ta_1\ c_1\ a_1,\ \ldots,\ Ta_m\ c_m\ a_m\ ) \\ \{\ Vardec_1,\ \ldots,\ Vardec_k \\ \quad Body; \\ \quad \text{RETURN}(\ r_1,\ \ldots,\ r_n\ ); \\ \} \end{array}\right]\!\!\right]
$$

$$
\mapsto \left\{\begin{array}{l} \text{FUNDEC}(\ FunId,\ intag_1,\ Ta_1,\ a_1,\ \ldots,\ intag_m,\ Ta_m,\ a_m, \\ \qquad\qquad\qquad outtag_1,\ Tr_1,\ r_1,\ \ldots,\ outtag_n,\ Tr_n,\ r_n\ ) \\ \{\ \ \mathrm{C}[\![Vardec_1]\!],\ \ldots,\ \mathrm{C}[\![Vardec_k]\!] \\ \quad AdjustRC(\ b_i,\ Refs\ (b_i,\ Body)\texttt{-1}); \\ \quad \mathrm{CR}[\![Body;\ \text{RETURN}(\ r_1,\ \ldots,\ r_n\ );\ ,\ \varepsilon]\!] \\ \} \end{array}\right. ,
$$

$$
\text{wobei } intag_i = \begin{cases} \text{INOUT} & \text{falls } c_i = \& \\ \text{IN} & \text{falls } c_i \neq\ \& \ \wedge\ Basetype(Ta_i) \in \mathcal{T}_{Simple} \\ \text{IN\_RC} & \text{falls } c_i \neq\ \& \ \wedge\ Basetype(Ta_i) \in \mathcal{T}_{Array} \end{cases} ,
$$

$$
outtag_i = \begin{cases} \text{OUT} & \text{falls } Basetype(Tr_i) \in \mathcal{T}_{Simple} \\ \text{OUT\_RC} & \text{sonst} \end{cases}
$$

$$
\text{und } b_i \in \{a_i \mid Basetype(Ta_i) \in\ \mathcal{T}_{Array}\} \quad .
$$

In gleicher Weise signalisieren INOUT-Marken als CALL-BY-REFERENCE-Parameter übergebene aktuelle Parameter bei Funktionsanwendungen:

$$
\mathrm{CR}[\![v_1,\ \ldots,\ v_n\ \texttt{=}\ FunId(\ e_1,\ \ldots,\ e_m\ );\ Rest\ ,\ \mathcal{F}]\!]
$$

$$
\mapsto \left\{\begin{array}{l} \text{FUNAP}(\ FunId,\ intag_1,\ e_1,\ \ldots,\ intag_m,\ e_m, \\ \qquad\qquad\qquad outtag_1,\ v_1,\ \ldots,\ outtag_n,\ v_n\ ); \\ AdjustRC(\ v_i,\ Refs\ (v_i,\ Rest;\ \mathcal{F})\texttt{-1}); \\ \mathrm{CR}[\![Rest\ ,\ \mathcal{F}]\!] \end{array}\right. ,
$$

$$
\text{wobei } intag_i = \begin{cases} \text{INOUT} & \text{falls } e_i \text{ CALL-BY-REFERENCE-Parameter} \\ \text{IN} & \text{falls } Basetype(\,TYPE(e_i)) \in \mathcal{T}_{Simple} \\ \text{IN\_RC} & \text{sonst} \end{cases}
$$

$$
\text{und } outtag_i = \begin{cases} \text{OUT} & \text{falls } Basetype(\,TYPE(v_i)) \in \mathcal{T}_{Simple} \\ \text{OUT\_RC} & \text{sonst} \end{cases} .
$$

4.5.2 Globale Objekte

Wie bei den CALL-BY-REFERENCE-Parametern ermöglichen die globalen Objekte lediglich eine kompaktere Notation. Sie gestatten es dem Programmierer, konzeptuell erforderliche Parameter in der Programmspezifikation wegzulassen. Mittels des Transformationsschemas $\mathcal{TF}_O$ können diese Parameter durch den Compiler inferiert und in das Programm eingefügt werden, so daß ein äquivalentes SAC-Programm gänzlich ohne globale Objekte entsteht. Eine Compilation des dabei entstehenden SAC-Programmes nach C ist jedoch ähnlich wie bei der Compilation der CALL-BY-REFERENCE-Parameter aus Effizienzüberlegungen abzulehnen. Für eine Compilation in ein sequentiell auszuführendes C-Programm sind diese Parameter nämlich weder notwendig noch nützlich, da durch die imperative Semantik ohnehin die Ausführungsreihenfolge festgelegt ist. Statt dessen bietet es sich an, auf die zusätzlichen Parameter zu verzichten und die globalen Objekte direkt in globale Variablen abzubilden.

Da eine Realisierung der Transformation $\mathcal{TF}_O$ trotzdem erforderlich ist, um die Uniqueness-Eigenschaft dieser Objekte zu verifizieren, sollte auch bei dieser Transformation die ursprüngliche Form des Programmes nachgehalten werden, um eine Compilation der globalen Objekte in effizient ausführbaren Code zu ermöglichen. Sie kann durch folgende Ergänzungen des C-Schemas realisiert werden. Die Definitionen von globalen Objekten werden in Deklarationen von globalen Variablen übersetzt:

$$\mathrm{C}[\![\,\texttt{OBJDEF}\ \tau\ Id\ \texttt{=}\ e\texttt{;}\,]\!] \mapsto \begin{cases} \tau\ Id\texttt{;} & \text{falls } \mathit{Basetype}(\tau) \in \mathcal{T}_{\mathit{Simple}} \\ \textsc{DeclUnqArray}(v, \sigma, s_1, \ldots, s_n)\texttt{;} & \text{falls } \mathit{Basetype}(\tau) = \sigma[s_1, ..., s_n] \end{cases}.$$

Die Initialisierung der globalen Objekte erfolgt am Beginn des MAIN-Programmes durch

$$\mathrm{C}[\![\,\mathit{Tydef}_1 ... \mathit{Tydef}_m\ \ \mathit{Objdef}_1 ... \mathit{Objdef}_r\ \ \mathit{Fundef}_1 ... \mathit{Fundef}_n\ \ \mathit{Main}\,]\!] \mapsto \begin{cases} \mathrm{C}[\![\,\mathit{Tydef}_1\,]\!] \ldots \mathrm{C}[\![\,\mathit{Tydef}_m\,]\!] \\ \mathrm{C}[\![\,\mathit{Objdef}_1\,]\!] \ldots \mathrm{C}[\![\,\mathit{Objdef}_r\,]\!] \\ \mathrm{C}[\![\,\mathit{Fundef}_1\,]\!] \ldots \mathrm{C}[\![\,\mathit{Fundef}_n\,]\!]\ \ \mathrm{C}[\![\,\mathit{Main}'\,]\!] \end{cases},$$

wobei Main' aus Main durch Einfügen von Zuweisungen der Form $id_i = e_i$; am Beginn des Rumpfes von Main für alle $\mathit{Objdef}_i \equiv$ OBJDEF $\tau_i\ id_i = e_i$; entsteht .

4.6 Maschinenunabhängige Optimierungen

Aufgrund des hohen Abstraktionsgrades der Sprachkonstrukte von SAC gegenüber der unterliegenden Rechnerarchitektur ist der Einsatz von Optimierungsverfahren unabdingbar, um effizient ausführbaren Code erzeugen zu können. Einen Großteil der Optimierungen, insbesondere alle maschinennahen Optimierungen des sog. Compiler-Backend, können durch eine Compilation nach C vollständig dem C-Compiler überlassen werden. Bei einer genaueren Betrachtung der Optimierungsmöglichkeiten eines C-Compilers stellt man jedoch fest, daß diese in bezug auf SAC-Programme zwei wesentliche Schwächen aufweisen (vergl. [Sie95]).

Zum einen wird die Anwendbarkeit von Optimierungen durch die in C möglichen Seiteneffekte eingeschränkt; Optimierungen, die Programmodifikationen über Anwendungen benutzerdefinierter Funktionen hinaus erfordern, werden in der Regel nicht vorgenommen. Aufgrund der Seiteneffektfreiheit von SAC wären diese jedoch für SAC-Programme möglich.

Zum anderen ist eine Optimierung von Array-Operationen durch einen C-Compiler kaum möglich. Das liegt daran, daß nach einer Compilation der SAC-Array-Operationen nur noch Folgen von indizierten Zeigerzugriffen vorliegen, die wegen der komplexen Möglichkeiten beim Umgang mit Zeigern in C nur geringfügig optimiert werden können. Optimierungen auf der Ebene von SAC-Programmen führen hier zu wesentlich besseren Ergebnissen.

Um diese Schwächen zu vermeiden, müssen sowohl die allgemein aus der Literatur (Zusammenfassungen siehe [ASU86, BGS94]) bekannten als auch spezielle, auf das Array-Konzept von SAC abgestimmte Optimierungen bei der Compilation von SAC nach C vorgenommen werden. Durch eine Optimierung auf Hochsprachenebene sind nicht nur weiterreichende Optimierungen in bezug auf die Array-Operationen möglich, sondern es wird auch die Unabhängigkeit von der Zielarchitektur gewahrt, da für die eigentliche Umsetzung in Maschinen-Code nach wie vor ein C-Compiler als Backend dient.

Im folgenden Abschnitt sollen kurz die wichtigsten aus der Literatur bekannten Optimierungen vorgestellt werden. Für jede dieser Optimierungen soll dabei untersucht werden, ob sie einem nachfolgenden C-Compiler überlassen werden kann, bzw. inwiefern eine direkte Anwendung auf SAC-Programme einer indirekten Anwendung durch den C-Compiler überlegen ist. Eine detaillierte Beschreibung dieser Algorithmen sowie eine Diskussion der damit verbundenen Laufzeitverbesserungen finden sich in [Sie95].

Anschließend wird ein kurzer Überblick über die Ansätze zur Optimierung von Array-Ausdrücken im allgemeinen gegeben. Zudem werden die speziell für die Array-Konstrukte in SAC entwickelten Optimierungen vorgestellt.

4.6.1 Allgemeine Optimierungen

Function Inlining eliminiert Funktionsaufrufe durch literales Einsetzen des Funktionsrumpfes. Da diese Optimierung ggf. zu einer erheblichen Vergrößerung des Programmtextes führen kann, müssen alle Funktionen, deren Anwendungen auf diese Weise eliminiert werden sollen, in SAC mit dem Schlüsselwort INLINE gekennzeichnet werden. So wird z.B.

```
b = sqr(7);    mit    INLINE INT sqr( INT a)
                      { RETURN(a*a);          zu   b = (7*7);    .
                      }
```

Die explizite Integration dieser Optimierung in den SAC-Compiler ist erforderlich, da FUNCTION-INLINING von den meisten C-Compilern nicht unterstützt wird.

Common Subexpression Elimination vermeidet eine mehrfache Auswertung identischer Ausdrücke, soweit diese statisch inferierbar sind. Dazu werden die identischen Ausdrücke durch eine temporäre Variable ersetzt, die mit dem benötigten Ausdruck initialisiert wird. Ein Programmausschnitt

```
v = 7;                          v = 7;
a = fac(v);      wird transformiert in   t_i = fac(v);
c = fac(v);                     a = t_i;          .
                                c = t_i;
```

Diese Optimierung wird zwar von den meisten C-Compilern unterstützt, jedoch bleiben Ausdrücke, die Aufrufe von benutzerdefinierten Funktionen enthalten, unberücksichtigt, da im allgemeinen für C-Funktionen nicht inferiert werden kann, ob diese Seiteneffekte verursachen oder nicht. Da in SAC jedoch durch Anwendung der Transformationsschemata $\mathcal{TF}_R$ und $\mathcal{TF}_O$ eine seiteneffektfreie Darstellung der benutzerdefinierten Funktionen erreicht wird, kann diese Optimierung für eine größere Menge von Ausdrücken durchgeführt werden. So kann in SAC im Gegensatz zu C auch ein Ausschnitt

```
v = 7;                          v = 7;
a = fac(v);                     t_i = fac(v);
a += c_killer(v);  optimiert werden zu   a = t_i;          .
c = fac(v);                     a += c_killer(v);
                                c = t_i;
```

Constant Folding ersetzt Anwendungen primitiver Funktionen auf Konstanten durch die entsprechenden Resultate. Um möglichst viele solcher statischen Berechnungen durchführen zu können, werden außerdem Konstanten propagiert. Dies bedeutet, Vorkommen von Variablen, die durch Konstanten initialisiert sind, werden durch die Konstanten ersetzt. Die Folge von Zuweisungen

```
a = 7;                                  a = 7;
b = (a+3);          wird dabei zu       b = 10;          .
c = (a+b);                              c = 17;
```

Obwohl die meisten C-Compiler diese Optimierung unterstützen, bedarf es einer Realisierung dieser Optimierung bei der Compilation von SAC nach C. Der wesentliche Grund dafür sind die in C nicht vorhandenen primitiven Array-Operationen von SAC. Darüber hinaus ist jedoch auch eine Realisierung des Constant Folding für die mit C identischen primitiven Operationen sinnvoll, da die Seiteneffektfreiheit benutzerdefinierter Funktionen in SAC im Gegensatz zu C ein Propagieren von Konstanten über Anwendungen solcher Funktionen hinaus zuläßt.

Loop Invariant Removal verlagert Ausdrücke, deren Berechnung nicht von den im Rumpf einer Schleife definierten Variablen abhängen, aus der Schleife heraus. Auch diese Optimierung kann durch das Fehlen von Seiteneffekten in SAC weitreichender erfolgen als durch einen C-Compiler. So kann z.B.

```
FOR(i=0; i<n; i++) {                    t_k = (x+y);
  a = (x+y+i);                          FOR(i=0; i<n; i++) {
  z = (a+i);          transformiert werden in     a = (t_k+i);
  b += c_killer(a);                       b += c_killer(a);   .
}                                       }
                                        z = (a+i);
```

Loop Unrolling ersetzt eine Schleife durch wiederholtes Einfügen des Schleifenrumpfes. Beispiel:

```
FOR(i=2; i<5; i++) {                    n *= PSI(a, [2]);
  n *= PSI(a, [i]);         wird zu     n *= PSI(a, [3]);
}                                       n *= PSI(a, [4]);   .
```

Diese Optimierung könnte prinzipiell vollständig einem C-Compiler überlassen werden. Das Problem liegt hier einzig darin, daß durch Loop Unrolling neue Anwendungsmöglichkeiten der oben genannten Optimierungen entstehen, so daß diese Optimierung vor den obigen erfolgen muß.

Loop Unswitching entfernt IF-THEN-ELSE-Konstrukte aus Schleifen. Entscheidend für das Entfernen von IF-THEN-ELSE-Konstrukten sind deren Prädikate. Ist so ein Prädikat schleifeninvariant, so wird das IF-THEN-ELSE-Konstrukt im Rahmen des Loop Invariariant Removal ohnehin aus der Schleife entfernt und der Schleifenrumpf in beiden Alternativen reproduziert. Unter den übrigen Prädikaten eignen sich nur noch solche für eine Optimierung, die direkt von der Induktionsvariablen der Schleife abhängen. Sie erlauben in vielen Fällen ein Entfernen des IF-THEN-ELSE-Konstruktes durch eine Aufspaltung der Schleife

in ein oder mehrere Schleifen mit neuen Grenzen für die Induktionsvariable. Zum Beispiel kann

```
FOR(i=0; i<m; i++) {
  IF(i<p)
    n *= i;
  ELSE
    n += i;
}
```

zerlegt werden in

```
FOR(i=0; i<p; i++) {
  n *= i;
}
FOR(; i<m; i++) {
  n += i;
}
```

.

Da auch für diese Optimierung das Wissen um die Seiteneffektfreiheit benutzerdefinierter Funktionen von entscheidender Bedeutung für die Anwendbarkeit der Optimierung ist, sollte sie ebenfalls Bestandteil des SAC-Compilers sein.

Dead Code Removal entfernt sämtliche Variablenzuweisungen, die nicht zum Ergebnis des gesamten Programmes beitragen. Ein Funktionsrumpf

```
{
  v = 42;
  w = c_killer(v);
  FOR(i=0; i<v; i++) {
    w += c_killer2( v, w);
  }
  w = v;
  RETURN(w); }
```

läßt sich reduzieren zu

```
{
  v = 42;
  w = v;
  RETURN(w); }
```

.

Diese Optimierung wird erst durch die Seiteneffektfreiheit von SAC möglich; erlaubt sie doch eine genaue Bestimmung der benötigten Ausdrücke. Konventionelle C-Compiler wie z.B. der GCC[Sta94] können daher diese Optimierung nur in wenigen Spezialfällen anwenden.

4.6.2 Optimierung von Array-Ausdrücken

Die Optimierung von Array-Ausdrücken hat eine besondere Bedeutung, da jedes Array einen erheblichen Aufwand zur Verwaltung des zugehörigen Speichers erfordert. Deshalb unterstützen insbesondere die Compiler von Sprachen, die auf numerische Anwendungen ausgerichtet sind, etliche Optimierungen zur Vermeidung dieses Aufwandes. Prinzipiell kann zwischen zwei verschiedenen Arten der Optimierung unterschieden werden: Hochsprachlichen Transformationen mit dem Ziel, die Erzeugung von Arrays zu vermeiden und Optimierungen, die darauf ausgerichtet sind, den für die Verwaltung des Speichers benötigten Code zu minimieren.

Für die Eliminierung von Arrays durch Hochsprachentransformationen bieten sich unterschiedliche Möglichkeiten, Arrays durch andere Sprachkonstrukte zu ersetzen bzw. ganze Programmfragmente so zu modifizieren, daß auf die Verwendung von Arrays verzichtet werden kann.

Einen möglichen Ansatz dazu stellt der Versuch dar, temporäre Arrays bei der Komposition von Array-Operationen zu vermeiden. So lassen sich insbesondere bei Sprachen mit APL-artigen Array-Operationen, wie SAC eine darstellt, Schachtelungen primitiver Array-Operationen oft durch einzelne Array-Operationen ersetzen: `TAKE([1], TAKE([2], a))` kann z.B. durch `TAKE([2,1], a)` ausgedrückt werden. Die Grundlage für derartige Optimierungen bieten die Reduktionsregeln des Ψ-Kalküls [Mul88, MT94], da die primitiven Operationen in SAC weitestgehend denen des Ψ-Kalkül s entsprechen. Einen anderen Ansatz zur Vermeidung temporärer Arrays bei der Schachtelung von Array-Operationen verfolgen die für HASKELL entwickelten sog. Deforestation-Verfahren [Wad90, GLJ93, Gil96]. Sie identifizieren Schachtelungen von Funktionen, die Listen erzeugen, mit solchen, die Listen konsumieren, und können bei geeigneter Konstellation dieser Funktionen das Programm derart transformieren, daß eine Erzeugung der temporären Struktur nicht mehr erforderlich wird.

Arrays, auf die nur mit konstanten Indizes zugegriffen wird,können durch skalare Werte ersetzt werden. Dabei können nicht nur die Array-Definitionen eliminiert, sondern auch die entsprechenden Selektionsoperationen durch Variablenzugriffe ersetzt werden. Eine solche an die sog. Array-Fission in SISAL[Can93] angelehnte Optimierung stellt die Array-Elimination in SAC dar.

Speziell aus dem Array-Konzept von SAC ergibt sich eine weitere Ansatzmöglichkeit für die Eliminierung von Arrays, die sich für ein kompetitives Laufzeitverhalten als essentiell herausgestellt hat. Durch die Verwendung von Indexvektoren bei selektiven Zugriffen wird zwar auf der einen Seite eine dimensionsunabhängige Programmierung möglich, auf der anderen Seite werden dadurch jedoch Arrays (Vektoren) eingeführt, die in vergleichbaren Programmen anderer Programmiersprachen durch skalare Werte dargestellt werden. Mit Hilfe einer speziellen Optimierung, der sog. Index-Vector-Elimination lassen sich solche, ausschließlich als Indizes verwendeten Arrays identifizieren und durch entsprechende skalare Werte ersetzen.

Neben den Optimierungen, die eine Vermeidung von Arrays in der Quellsprache zum Ziel haben, müssen jedoch auch die Optimierungen zur Verringerung des Verwaltungsaufwandes betrachtet werden. Im Vordergrund dieser Optimierungen stehen Versuche, die Anzahl der für die Referenzzählung benötigten Operationen sowie den Kopieraufwand bei Modifikationsoperationen durch sog. Update-In-Place Analysen [Can89, Can92, SS88, SCA93] zu minimieren.

Andere Ansätze wie das sog. Framework Preconstruction in SISAL[CE95] basieren auf der Idee, nicht mehr benötigten Speicherplatz in einigen Situationen nicht sofort freizugeben, mit dem Ziel, ihn wiederverwenden zu können. Dies macht sich insbesondere in solchen Schleifen positiv bemerkbar, in denen bei jeder Inkarnation temporäre Arrays gleicher Größe benötigt werden.

Darüber hinaus ergeben sich diverse compilations-spezifische Optimierungen, wie z.B. die Verwendung einer eigenen Speicherverwaltung, alternative Array-Darstellungsformen etc. In diesem Zusammenhang ist in bezug auf SAC vor allem das sog. With-Loop-Unrolling zu nennen. Es handelt sich hierbei um die Übertragung des

Loop-Unrolling auf die WITH-Konstrukte von SAC.

Im folgenden werden die Array-Eliminierung und die Index-Vector-Eliminierung kurz vorgestellt, da diese beiden Optimierungen speziell für SAC entwickelt wurden.

Array-Eliminierung

Eine Ersetzung von Arrays durch eine entsprechende Anzahl von skalaren Werten ist weder in allen Fällen möglich noch sinnvoll. So kann in den Fällen, in denen mit einem variablen Index auf ein Array zugegriffen wird, eine Array-Eliminierung nicht durchgeführt werden, da statisch nicht inferiert werden kann, auf welches Array-Element zugegriffen werden soll. Die Anwendung der Array-Eliminierung beschränkt sich damit auf Arrays, deren Elemente ausschließlich durch Zugriffe mit konstanten Indizes selektiert werden. In der Praxis kommen solche Arrays immer dann vor, wenn Arrays wie Records verwendet werden, wie es z.B. für die Darstellung komplexer Zahlen durch ein Array von Gleitkommazahlen der Fall ist. So wird ein Programmfragment

```
{                                          {
  c = [r,i];                                 c_0 = r;
  ⋮                                          c_1 = i;
  ... PSI(1,c) ... PSI(0,c) ... ;     zu     ⋮                          .
  ⋮                                          ... c_0 ... c_1 ... ;
}                                            ⋮
                                           }
```

Selbst für den Fall, daß eine solche Ersetzung durch skalare Elementwerte möglich ist, wirkt sich diese Modifikation jedoch nicht immer positiv aus. Ursache hierfür sind Anwendungen benutzerdefinierter Funktionen auf solche Arrays. Eine vollständige Ersetzung der Arrays durch skalare Werte bedeutet für solche Funktionen eine je nach Größe des Arrays ggf. nicht unerhebliche Anzahl zusätzlicher Parameter. Darüber hinaus sind solche Modifikationen nur bei Funktionen, deren Definition explizit vorliegt, möglich. Für Funktionen aus importierten Modulen bedeutet dies, daß eine solche Modifikation nur erfolgen kann, wenn sich deren Definitionen im SIB (vergl. Abschnitt 4.4) befinden.

Eine Restriktion auf Arrays einer vorgebbaren Maximalgröße sowie ein Verzicht auf die Modifikation von Aufrufen benutzerdefinierter Funktionen erlaubt einerseits eine im Vergleich zu sonstigen Optimierungen einfache Implementierung, und bietet andererseits in den meisten Fällen die gewünschten Laufzeitverbesserungen (vergl. [Sie95]).

Der Algorithmus zur Array-Elimination ist in Abb. 4.4 skizziert. Er basiert auf der Idee, zunächst beide Darstellungen zu erzeugen und anschließend mittels des Dead-Code-Removal die überflüssige Version zu entfernen. Dazu werden zunächst bei allen elementweise spezifizierten Arrays Variablen für jede Komponente des Ar-

1. Für alle Zuweisungen von Arrays mit weniger als *minarray* Elementen:
   - Bei Zuweisungen der Form v = [e_1, ..., e_n]; oder der Form v = RESHAPE(*shp*, [e_1, ..., e_n]); füge neue Element-Variablen v_idx_i für alle legalen Indizes idx_i in v sowie Zuweisungen $v\_idx_1 = e_1$; ...; $v\_idx_n = e_n$; ein.
   - Bei allen anderen Zuweisungen $v = expr$; füge neue Element-Variablen v_idx_i für alle legalen Indizes idx_i in v sowie Zuweisungen v_idx_1 = PSI(idx_1, v); ...; v_idx_n = PSI(idx_n, v); ein.
2. Für alle Selektionen der Form PSI(*const*, *a*), bei denen *const* eine Konstante ist, und a eine Array-Variable mit weniger als *minarray* Elementen:
   - ersetze PSI(*const*, v) durch v_*const*.
3. Führe Dead Code Removal durch.

Abbildung 4.4: Algorithmus zur Array-Elimination

rays eingefügt (Abb.4.4 1.). Anschließend werden alle Selektionen mit konstanten Indizes durch die entsprechenden Variablen ersetzt (Abb.4.4 2.). In der dritten und abschließenden Phase wird das Dead-Code-Removal durchgeführt. Konnten in der zweiten Phase alle Array-Zugriffe durch Variablen ersetzt werden, so wird hier das Array eliminiert. Sind noch Array-Zugriffe verblieben, so werden überflüssig eingeführte Variablen für Array-Komponenten wieder entfernt.

Index-Vector-Elimination

Ziel dieser Optimierung ist es,die explizite Erzeugung von Arrays (Vektoren) zu vermeiden, die ausschließlich für Selektionszwecke in Arrays genutzt werden. Um dies als Hochsprachentransformation durchführen zu können, müssen Selektionsfunktionen eingeführt werden, die direkt mit offsets in die Elementvektoren operieren: Dazu werden zwei Funktionen

$$\text{IDX\_PSI} : \text{INT} \times \mathcal{T}_{Array} \rightarrow_{part} \mathcal{T}_{\text{SAC}}$$

und

$$\text{IDX\_MODARRAY} : \text{INT} \times \mathcal{T}_{\text{SAC}} \times \mathcal{T}_{SimpleArray} \rightarrow_{part} \mathcal{T}_{SimpleArray}$$

eingeführt. Sie entsprechen den Funktionen PSI bzw. MODARRAY insofern, als daß gilt:

$$\begin{aligned} \text{IDX\_PSI}(idx, a) &= \text{PSI}([i_0, ..., i_{n-1}], a) && \text{, bzw.} \\ \text{IDX\_MODARRAY}(idx, val, a) &= \text{MODARRAY}([i_0, ..., i_{n-1}], val, a) && , \end{aligned}$$

g.d.w. *idx* das offset des durch $[i_0, ..., i_{n-1}]$ gekennzeichneten Elementes (Teil-Arrays) im Element-Vektor von a ist, d.h.

$$idx = \sum_{j=0}^{n-1} i_j * \prod_{k=j+1}^{\text{DIM}(a)-1} \text{PSI}([k], \text{SHAPE}(a)) \quad .$$

Diese Funktionen sind jedoch ausschließlich als intern verfügbar zu verstehen. Ein Ausdruck PSI([1,2], a) wird damit zu IDX_PSI((1*8+2), a), falls SHAPE(a) = [10,8].

Durch die Indizierung der Arrays mittels Vektoren werden Index-Berechnungen durch Anwendungen arithmetischer Operationen auf Vektoren statt Skalaren spezifiziert. Um auch in diesen Situationen auf eine Erzeugung von Vektoren zur Laufzeit verzichten zu können, muß die Analyse der Art der angewandten Vorkommen eines Arrays solche Operationen einbeziehen. Dadurch kann z.B. unter der Annahme, daß SHAPE(a) = [10, 8] gilt, eine Transformation des Programmfragmentes

```
v = [1,2];                 v__10_8= (1*8+2);
b = PSI(v+1, a);      in   b      = IDX_PSI(v__10_8+(1*8+1), a);   erfolgen.
c = PSI(v+[2,3], a);       c      = IDX_PSI(v__10_8+(2*8+3), a);
d = PSI(4*v, a);           d      = IDX_PSI(4*v__10_8, a);
```

Der Algorithmus zur Index-Vector-Elimination ist in Abb. 4.5 beschrieben. Zunächst muß für jede Definition einer Array-Variablen v bestimmt werden, wie sie nachfolgend verwendet wird (Abb. 4.5 1.). Dazu führen wir Attribute ein, die die Verwendung eines Arrays kennzeichnen. Wir unterscheiden zwischen der Verwendung als „normalem" Vektor: VECT und der Verwendung als Indexvektor in ein Array a: IDX(shp), wobei shp = SHAPE(a).

Die Bestimmung dieser Attribute erfolgt durch eine Funktion $Uses(v)$, die jeder Array-Variablen für eindimensionale Arrays mit Elementen des Typs INT eine Menge solcher VECT bzw. IDX(shp) Attribute zuordnet. Bei der Bestimmung der IDX(shp)-Attribute ist zu beachten, daß diese nicht nur bei Anwendungen der Form PSI(v, a), sondern auch bei reinen Indexberechnungen inferiert werden. Dazu folgendes Programmfragment als Beispiel:

```
v = [2,3,4];
w = v-[1,1,1];
x = v+[1,1,1];
e = PSI(v,a);
e += PSI(3*v,b);
e += PSI(w,c);
e += PSI(x,d);
f = [x,x];
RETURN(e,f);
```

für gilt:

$$\begin{aligned} Uses(\text{v}) &= \{\ \text{IDX}(\text{SHAPE}(\text{a})),\ \text{IDX}(\text{SHAPE}(\text{b})),\\ &\qquad \text{IDX}(\text{SHAPE}(\text{c})),\ \text{VECT}\}\\ Uses(\text{w}) &= \{\ \text{IDX}(\text{SHAPE}(\text{c}))\}\\ Uses(\text{x}) &= \{\ \text{IDX}(\text{SHAPE}(\text{d})),\ \text{VECT}\} \end{aligned} \quad .$$

1. Bestimme für jede Array-Variable v die Menge ihrer Anwendungsarten $Uses(v)$.
2. Ersetze jede Array-Zuweisung $v = Expr$; durch:

$$\begin{cases} v\_shp_1 \texttt{ = } \mathcal{O}_{\text{IVE}}[\![Expr \texttt{ , } shp_1]\!]\texttt{;} \\ \texttt{...} \\ v\_shp_n \texttt{ = } \mathcal{O}_{\text{IVE}}[\![Expr \texttt{ , } shp_n]\!]\texttt{;} \\ \\ v \texttt{ = } Expr\texttt{;} \\ v\_shp_1 \texttt{ = VECT2OFFSET(v, } shp_1\texttt{);} \\ \texttt{...} \\ v\_shp_n \texttt{ = VECT2OFFSET(v, } shp_n\texttt{]);} \\ \\ v = Expr; \end{cases} \quad \begin{array}{ll} & \\ & \\ \text{falls } Uses(v) = & \{\text{IDX}(shp_1), \\ & \ldots, \text{IDX}(shp_n)\} \\ & \\ & \\ \text{falls } Uses(v) = & \{\text{VECT}, \text{IDX}(shp_1), \\ & \ldots, \text{IDX}(shp_n)\} \\ & \\ & \\ \text{sonst} & \end{array}$$

wobei

$$\mathcal{O}_{\text{IVE}}[\![w \texttt{ , [} s_1 \texttt{, ..., } s_n \texttt{]}]\!] \mapsto w\_s_1\_\ldots\_s_n$$

$$\mathcal{O}_{\text{IVE}}[\![\texttt{[} a_1 \texttt{, ..., } a_m \texttt{] , [} s_1 \texttt{, ..., } s_n \texttt{]}]\!] \mapsto (\ldots((a_1 * s_2 + a_2) * s_3 \ldots + a_m) * s_{m+1} * \ldots * s_n)$$

$$\mathcal{O}_{\text{IVE}}[\![e_1 \ \pm e_2 \ , \ shp]\!] \mapsto \begin{cases} \mathcal{O}_{\text{IVE}}[\![\overbrace{\texttt{[}e_1\texttt{, ..., }e_1\texttt{]}}^{m} \ , \ shp]\!] \pm \mathcal{O}_{\text{IVE}}[\![e_2 \ , \ shp]\!] & \text{falls } \begin{array}{l} Basetype(TYPE(e_1)) = \text{INT} \\ \wedge\ m = \text{DIM}(e_2) \end{array} \\ \mathcal{O}_{\text{IVE}}[\![e_1 \ , \ shp]\!] \pm \mathcal{O}_{\text{IVE}}[\![\overbrace{\texttt{[}e_2\texttt{, ..., }e_2\texttt{]}}^{m} \ , \ shp]\!] & \text{falls } \begin{array}{l} Basetype(TYPE(e_2)) = \text{INT} \\ \wedge\ m = \text{DIM}(e_1) \end{array} \\ \mathcal{O}_{\text{IVE}}[\![e_1 \ , \ shp]\!] \pm \mathcal{O}_{\text{IVE}}[\![e_2 \ , \ shp]\!] & \text{sonst} \end{cases}$$

$$\mathcal{O}_{\text{IVE}}[\![e_1 \ *\!/ \ e_2 \ , \ shp]\!] \mapsto \begin{cases} e_1 *\!/ \mathcal{O}_{\text{IVE}}[\![e_2 \ , \ shp]\!] & \text{falls } Basetype(TYPE(e_1)) = \text{INT} \\ \mathcal{O}_{\text{IVE}}[\![e_1 \ , \ shp]\!] *\!/ e_2 & \text{falls } Basetype(TYPE(e_2)) = \text{INT} \end{cases}$$

3. Ersetze alle Selektionen PSI(idx, a) durch IDX_PSI($idx\_s_1\_\ldots\_s_n, a$) sowie MODARRAY(idx, val, a) durch IDX_MODARRAY($idx\_s_1\_\ldots\_s_n, val, a$) falls SHAPE($a$) = $[s_1, \ldots, s_n]$.
4. Für alle Array-Variablen v mit $Basetype(TYPE(v)) = \text{INT}[m]$:
   für alle IDX($[s_1, \ldots, s_n]$) aus $Uses(v)$:
   füge Variablendeklarationen INT $v\_s_1\_\ldots\_s_n$; ein.

Abbildung 4.5: Algorithmus zur Index-Vector-Elimination

Die Zuordnungen der Attribute IDX(SHAPE(a)) und IDX(SHAPE(b)) zu `v` ergeben sich direkt aus den Selektionen auf die Arrays `a` und `b`. IDX(SHAPE(c)) läßt sich über $Uses$(`w`) und die Zuweisung `w = v-[1,1,1];` ableiten. Eine Zuordnung von IDX(SHAPE(d)) zu `v` aufgrund der Zuweisung `x = v+[1,1,1];` ist nicht sinnvoll, da `x` nicht ausschließlich als Index in `d`, sondern auch als Vektor benötigt wird. Deshalb

führt `x = v+[1,1,1];` zu der Zuordnung von VECT zu `v`.

Nach der Zuordnung der Verwendungsattribute können alle nötigen Programmtransformationen vorgenommen werden. Prinzipiell wird für jede Nutzung eines Vektors `v` als Index in ein Array mit Shape-Vektor [s_1, ..., s_n] eine neue Indexvariable vom Typ INT v_s_1_..._s_n eingeführt. Indexberechnungen ergeben sich dann als skalare Operationen auf diesen Variablen. Wird eine Variable ausschließlich als Index benutzt, so kann die eigentliche Zuweisung `v` = *Array_Expr*; entfallen und durch Zuweisungen an die neuen Indexvariablen `v`_shp_i für alle vorkommenden Shapes shp_1, ..., shp_n ersetzt werden. Das Schema $\mathcal{O}_{\text{IVE}}\left[\!\left[Expr\ ,\ Shp \right]\!\right]$ aus Abb. 4.5 2 beschreibt dabei die Umsetzung der Indexberechnungen. Wird `v` jedoch auch als Vektor benötigt, verbleibt die Instanzierung `v` = *Array_Expr*; im Programm und die neuen Variablen `v`_shp_i werden mittels einer weiteren internen primitiven Funktion VECT2OFFSET aus dem Vektor `v` gewonnen (vergl. Abb.4.5 2). Dabei ist

$$\text{VECT2OFFSET} : \text{INT}[.] \times \text{INT}[.] \rightarrow_{part} \text{INT}$$

definiert als

$$\text{VECT2OFFSET}([v_1, ..., v_n], [s_1, ..., s_m]) = \sum_{i=1}^{n} v_i * \prod_{j=i+1}^{m} s_j \qquad \text{falls } m \geq n \qquad .$$

Neben der Transformation der Zuweisungen an Indexvektorvariablen werden alle PSI-Zugriffe sowie MODARRAY-Anwendungen durch die internen Funktionen IDX_PSI bzw. IDX_MODARRAY ersetzt (Abb. 4.5 3). Außerdem bedarf es schließlich der Einfügung der zugehörigen Variablendeklarationen (Abb. 4.5 4).

Kapitel 5

Eine Fallstudie: Multigrid-Relaxation

Nachdem das Sprachdesign von SAC sowie die Compilation von SAC nach C vorgestellt worden sind, soll in diesem Kapitel die Verwendbarkeit von SAC anhand einer Fallstudie untersucht werden. Als Beispielproblem dient die numerische Approximation spezieller partieller Differentialgleichungen, den sog. Poisson-Gleichungen, durch Multigrid-Relaxation [Hac85, HT82, Bra84]. Multigrid-Relaxation ist nicht nur zentraler Bestandteil vieler „Real-World-Applications", sondern stellt auch verschiedenartige Anforderungen an ihre Implementierung: Zum einen werden iterative Modifikationen mehrdimensionaler Arrays mit mehr als 10^6 Elementen benötigt und zum anderen müssen Arrays verschiedener Größe aufeinander abgebildet werden. Außerdem wird Multigrid-Relaxation für Probleme unterschiedlicher Dimensionalität eingesetzt; eine dimensionsunabhängige Spezifikation ist daher von besonderem Interesse.

Nach einer kurzen Erläuterung des Multigrid-Verfahrens soll auf zwei verschiedene Fragestellungen eingegangen werden.

- Inwieweit ist das Sprachdesign von SAC dafür geeignet, solche Algorithmen elegant dimensionsunabhängig zu spezifizieren?
- Welches Laufzeitverhalten zeigt das Compilat solcher SAC-Programme im Vergleich zu äquivalenten dimensionsspezifischen FORTRAN- und SISAL-Implementierungen?

5.1 Multigrid-Relaxation

Poisson-Gleichungen sind partielle Differentialgleichungen der allgemeinen Form

$$\Delta\, u\,(x_0,\, \ldots\, , x_{p-1}) \;=\; f\,(x_0,\, \ldots\, , x_{p-1}) \;\;\mid (x_0,\, \ldots\, , x_{p-1}) \;\in\; \Omega \;\;,$$

wobei Δ den Laplace-Operator und Ω den Definitionsbereich darstellen. Zu gegebenem f ist u derartig zu approximieren, daß eine vorgebbare Randbedingung erfüllt wird.

Die bekannten Gauss-Seidel- oder Jacobi-Relaxationsmethoden basieren auf einer Diskretisierung dieser Gleichung auf p-dimensionalen Gittern mit einem festen Gitterabstand h. Sie modifizieren iterativ die inneren Elemente eines Arrays a mit Shape-Vektor $[s_1, \ldots, s_p]$, indem gewichtete Summen der jeweiligen Elemente selbst mit ihren Nachbarelementen gebildet werden. Durch die Verwendung p-dimensionaler Koeffizienten-Arrays c mit Shape-Vektor $[\underbrace{3, \ldots, 3}_{p}]$ läßt sich die Modifikation der inneren Elemente von a spezifizieren als:

$$\forall i_0 \in \{1, ..., s_0 - 2\} ... \forall i_{p-1} \in \{1, ..., s_{p-1} - 2\} :$$
$$a'[i_0, ..., i_{p-1}] = \sum_{j_0=0}^{2} \cdots \sum_{j_{p-1}=0}^{2} c[j_0, ..., j_{p-1}] * a[(i_0 + j_0 - 1), ..., (i_{p-1} + j_{p-1} - 1)] \quad (5.1)$$

Dieser Vorgang wird solange wiederholt, bis ein Fixpunkt im Bezug auf eine vorgebbare Toleranz erreicht ist.

Obwohl diese Relaxationsverfahren dafür bekannt sind, hochfrequente Fehler schnell zu beheben, ist das Konvergenzverhalten bei niederfrequenten Fehlern eher schlecht. Dies ist zurückzuführen auf die langsame elementweise Propagation von korrigierten Werten durch das gesamte Gitter, das in realen Anwendungen mehr als 10^3 Elemente pro Dimension umfassen kann. Abhilfe für dieses Problem bieten die sog. Multigrid-Verfahren [Hac85, HT82, Bra84], die die eigentliche Relaxation in sog. Vergröberungs- und Verfeinerungschritte einbinden.

Der zu implementierende Algorithmus läßt sich folgendermaßen skizzieren: Die Relaxationsschritte werden rekursiv auf Gittern mit Abständen $h_1, ..., h_k, h_{(k+1)}, ..., h_m$ mit $h_{(k+1)} = 2 * h_k$ angewandt. Ausgehend vom feinsten Gitter werden rekursiv Vergröberungen des Gitters vorgenommen bis das gröbste Gitter erreicht ist. Anschließend erfolgen abwechselnd Relaxationsschritte und Verfeinerungen, so daß Fehler schneller propagieren können. Sowohl die Vergröberungen als auch die Verfeinerungen erfolgen durch eine Berechnung gewichteter Summen von Nachbarelementen. Abb. 5.1 veranschaulicht dies für den zweidimensionalen Fall. Die schwarzen Punkte stellen Gitterpunkte dar, die zu einem groben Gitter mit Gitterabstand $2*h$ gehören. Die schwarzen und die unausgefüllten Punkte zusammen bilden die Elemente eines feineren Gitter mit Gitterabstand h. Die Pfeile in der Mitte des linken Gitters deuten an, wie die Elemente des gröberen Gitters aus denen des feineren Gitters bei einer Vergröberung entstehen. Sie werden als gewichtete Summe aller „Nachbarelemente" des feineresn Gitters berechnet. Entsprechend zeigt das rechte Gitter, aus welchen Elementen des gröberen Gitters die Elemente des feineren Gittes im Verlauf einer Verfeinerung berechnet wird.

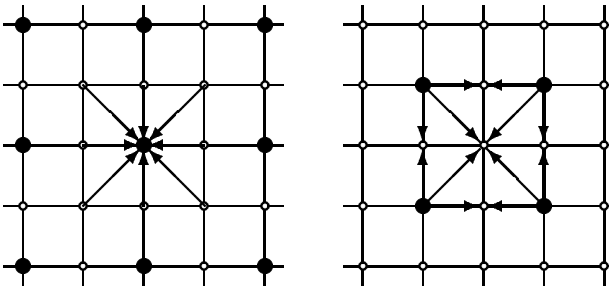


Abbildung 5.1: Vergröberung und Verfeinerung des Gitters

5.2 Spezifikation von Multigrid-Algorithmen

Aus den Darstellungen des vorherigen Abschnittes ergeben sich für die Implementierung der Multigrid-Relaxation drei Teilalgorithmen: Relaxation, Vergröberung des Gitters und Verfeinerung des Gitters. Die Probleme und Möglichkeiten einer dimensionsunabhängigen SAC-Implementierung für jeden dieser Teilalgorithmen soll im folgenden diskutiert werden.

5.2.1 Relaxation

Um die entscheidenden Mechanismen bei einer dimensionsunabhängigen Spezifikation identifizieren zu können, soll zunächst der zweidimensionale Fall untersucht werden. Für den Relaxationsschritt ergibt sich damit Gleichung (5.1) zu

$$\begin{aligned} &\forall i_0 \in \{1, ..., s_0 - 2\} : \forall i_1 \in \{1, ..., s_1 - 2\} : \\ &a'[i_0, i_1] = \sum_{j_0=0}^{2} \sum_{j_1=0}^{2} c[j_0, j_1] a[(i_0 + j_0 - 1), (i_1 + j_1 - 1)] \end{aligned} \quad . \tag{5.2}$$

Eine direkte Umsetzung dieser Beschreibung in geschachtelte Schleifenkonstrukte wie z.B. mit den DO-Schleifen in FORTRAN oder den FOR-INITIAL-Schleifen in SISAL ist in SAC ebenfalls möglich. Bezeichne `a` das Argument-Array mit Shape-Vektor `[m,n]` und `c` das Array der Gewichte mit Shape-Vektor `[3,3]`, so ergibt sich:

```
new_a = a;
FOR( ix = 1; ix <= m-2; ix++) {
  FOR( iy = 1; iy <= n-2; iy++) {
    val = 0;
    FOR( jx = 0; jx <= 2; jx++) {
      FOR( jy = 0; jy <= 2; jy++) {
        val += (c[[ jx, jy]] * a[[ ix+jx-1,] iy+jy-1]]);
    }
    new_a = MODARRAY( [ ix, iy], val, new_a);
  }
}
```

Der Nachteil dieser Spezifikation liegt darin, daß eine Modifikation des Programmes zur Anwendung auf Arrays höherer Dimensionalität das Einfügen zusätzlicher FOR-Schleifen erfordert. Um dies zu vermeiden, bedarf es der Verwendung der WITH-Konstrukte. Sie führt zu folgender, sehr viel übersichtlicheren Spezifikation:

```
new_a = WITH( [1,1] <= i <= [m-2, n-2] ) {
          val= WITH( [0,0] <= j <= [2,2] )
               FOLD( +, 0, c[j] * a[ i+j-1);
        }
        MODARRAY( i, val, a);
```

Eine Anpassung dieser Variante an Argument-Arrays `a` anderer Dimensionalität erfordert ausschließlich die Erweiterung der unteren bzw. der oberen Schranken für die Indexvektoren in den Generatorteilen der WITH-Konstrukte. Dies ist die entscheidende Voraussetzung für eine dimensionsunabhängige Spezifikation. Um sie zu erreichen, bedarf es lediglich einer Spezifikation dieser Schranken in Abhängigkeit des Shape-Vektors von `a` bzw. `c`. Aus Gleichung (5.1) läß sich `SHAPE(a)*0+1 <= i <= SHAPE(a)-2` für das äußere WITH-Konstrukt und `SHAPE(c)*0 <= i <= SHAPE(c)-1` für das innere ableiten.

Die auf den ersten Blick etwas merkwürdige Formulierung der unteren Schranken ist dadurch bedingt, daß dies die kompakteste Notation ist, Vektoren mit konstanten Elementwerten zu erzeugen, die entsprechend der Dimensionalität eines gegebenen Arrays viele Elemente enthalten.

Als dimensionsunabhängige Spezifikation erhalten wir somit:

```
new_a = WITH( SHAPE(a)*0 + 1 <= i <= SHAPE(a)-2 ) {
          val= WITH( SHAPE(c)*0 <= j <= SHAPE(c)-1 )
               FOLD( +, 0, c[j] * a[ i+j-1]);
        }
        MODARRAY( i, val, new_a);
```

Obwohl SISAL mit den FOR-Schleifen ein dem WITH-Konstrukt ähnliches Sprachkon-

strukt bietet, ist eine dimensionsunabhängige Spezifikation in SISAL nicht möglich. Der Grund dafür liegt in der Indizierung von Arrays durch Skalare statt durch Vektoren. Sie erfordert das Einfügen zusätzlicher Indexvariablen und verhindert somit eine dimensionsunabhängige Spezifikation.

5.2.2 Vergröberung des Gitters

Die Implementierung von Gittervergröberungen ähnelt der der Relaxationsschritte. Ein Element des gröberen Gitters berechnet sich durch eine gewichtete Summe aller Nachbarelemente des entsprechenden Elementes im feinen Gitter. Bezeichnet `R` das Array der Koeffizienten für diese Summenbildung und `a_f` das Array des feinen Gitters, dann kann die Berechnung des Arrays für das grobe Gitter `a_c` dimensionsunabhängig spezifiziert werden als

```
a_c = WITH( 0*SHAPE(a_f)+1 <= i <= SHAPE(a_f)/2-1 ) {
        val = WITH( 0*SHAPE(R) <= j <= SHAPE(R)-1 )
              FOLD( +, 0, R[j] * a_f[ 2*i+j-1]);
      }GENARRAY( SHAPE(a_f)/2+1, val);
```

5.2.3 Verfeinerung des Gitters

Die Implementierung von Gitterverfeinerungen läßt sich nicht ganz so einfach spezifizieren. Das Problem besteht darin, daß sich die Elemente des feineren Gitters in Abhängigkeit von ihrer Position relativ zu den Elementen des gröberen Gitters unterschiedlich berechnen. Eine dimensionsspezifische Lösung für p-dimensionale Gitter läßt sich durch die Verwendung p geschachtelter FOR-Schleifen über alle Elemente des gröberen Gitters formulieren. Eine solche Spezifikation für den zweidimensionalen Fall ist beispielsweise

```
a_f = WITH( 0*SHAPE(a_c) <= i <= SHAPE(a_c)*2-3 )
      GENARRAY( SHAPE(a_c)*2-2, 0);

FOR( i = 1; i <= SHAPE(a_c)[0]-2; i++) {
  FOR( j = 1; j <= SHAPE(a_c)[1]-2; j++) {
    a_f = MODARRAY( [2*i  , 2*j  ], a_c[[i,j]], a_f);
    a_f = MODARRAY( [2*i-1, 2*j  ], 0.5 * (a_c[[i,j]]+a_c[[i-1,j]]), a_f);
    a_f = MODARRAY( [2*i  , 2*j-1], 0.5 * (a_c[[i,j]]+a_c[[i,j-1]]), a_f);
    a_f = MODARRAY( [2*i-1, 2*j-1], 0.25* (a_c[[i,j]]+a_c[[i-1,j-1]]
                                          +a_c[[i-1,j]]+a_c[[i,j-1]]), a_f);
  }
}
```

wobei `a_f` das Array für das feinere Gitter und `a_c` das Array für das gröbere Gitter bezeichnet. Sie ist nahezu identisch zu FORTRAN-, C- oder SISAL-Implementierungen. Durch die Verwendung von separaten Indexvariablen für jede Achse der Arrays wird

der Vorteil einer Indizierung durch Vektoren neutralisiert. Stattdessen wird - wie in herkömmlichen Sprachen - ein fehlerträchtiges direktes Indizieren der Nachbarelemente erforderlich.

Um die Gitterverfeinerung dimensionsunabhängig spezifizieren zu können, bedarf es der Verwendung eines etwas modifizierten Algorithmus. Die Grundidee liegt darin, vor der eigentlichen Berechnung der Gitterelemente das feinere Gitter so zu initialisieren, daß alle Elemente gleichartig berechnet werden können. Dabei wird das feine Gitter abwechselnd mit den Werten des groben Gitters und dem Wert 0 vorbelegt. Dadurch haben alle Elemente, die ein aus dem groben Gitter übernommenes Element umgeben, unterschiedliche Muster von Nachbarelementen mit dem Wert 0. Sie erlauben es, alle Elemente des feinen Gitters auf die gleiche Weise als gewichtete Summe der Nachbarelemente zu berechnen wie bei der Relaxation und der Gittervergröberung auch. Um dies zu veranschaulichen, soll das Koeffizienten-Array P für den zweidimensionalen Fall als Beispiel dienen:

$$P = \begin{pmatrix} 1/4 & 1/2 & 1/4 \\ 1/2 & 1 & 1/2 \\ 1/4 & 1/2 & 1/4 \end{pmatrix} \quad .$$

Bei der „Berechnung“ eines Elementes, das aus dem groben Gitter übernommen wurde, wird der Wert dieses Elementes reproduziert, da es mit 1 multipliziert wird und alle umgebenden Punkte nur den Wert 0 beisteuern. Wird ein Element berechnet, das zwischen zwei Elementen liegt, die aus dem groben Gitter übernommen wurden, so tragen lediglich diese beiden Nachbarelemente zum neuen Wert des Elementes bei, da alle anderen Nachbarelemente den Wert 0 haben. Sie werden jeweils mit dem Faktor 1/2 gewichtet. Auf gleiche Weise ergibt sich ein Element, das in der Mitte von vier aus dem groben Gitter übernommenen Elementen liegt, aus der Summe dieser Werte, gewichtet mit dem Faktor 1/4.

Dieser Algorithmus ist für eine dimensionsunabhängige Spezifikation weit besser geeignet. Mit `P` als Koeffizienten-Array und `a_c` als Array des gröberen Gitters ergibt sich für die Berechnung des Arrays `a_f` des feineren Gitters folgendes Programmsegment:

```
a_f = WITH( 0*SHAPE(a_c) <= i <= SHAPE(a_c)*2-3 )
      GENARRAY( SHAPE(a_c)*2-2, 0);

a_f = WITH( 0*SHAPE(a_c) <= i <= SHAPE(a_c)-1 )
      MODARRAY( 2*i, a_c[i], a_f);

a_c = WITH( 0*SHAPE(a_f)+1 <= i <= SHAPE(a_f)/2-1 ) {
         val = WITH( 0*SHAPE(P) <= j <= SHAPE(P)-1 )
               FOLD( +, 0, P[j] * a_f[ i+j-1]);
      }MODARRAY( i, val, a_f);
```

Diese Spezifikation der Gitterverfeinerung ist kompakter als die oben dargestellte, dimensionsabhängige Implementierung. Sie vermeidet ein fehleranfälliges explizites Spezifizieren der einzelnen Nachbarindizes im Rumpf der inneren FOR-Schleife und ist darüberhinaus auf Arrays verschiedener Dimensionalität anwendbar.

5.3 Laufzeitverhalten des Multigrid-Algorithmus

Basierend auf den dimensionsunabhängigen Spezifikationen aus dem vorangegangenen Abschnitt soll hier das Laufzeitverhalten einer solchen SAC-Multigrid-Spezifikation untersucht und gegen das äquivalenter FORTRAN- und SISAL-Implementierungen verglichen werden.

Dazu wird eine C-Implementierung des im Kapitel 4 vorgestellten Compilers namens SAC2C verwendet. Sie entstand in Zusammenarbeit mit C. Grelck, A. Sievers und H. Wolf. SAC2C unterstützt den vollen Sprachumfang von SAC [Wol95, Gre96] sowie sämtliche, in Abschnitt 4.6 vorgestellten Optimierungen [Sie95]. Darüber hinaus ist eine Klassenbibliothek vergleichbar mit der `stdio`-Bibliothek von C verfügbar.

SAC2C erzeugt aus SAC-Spezifikationen C-Programme, die anschließend mittels des GNU C-Compilers GCC (Version 2.6.3) in Maschinen-Code übersetzt werden. Für die FORTRAN- und SISAL-Programme werden der SUN FORTRAN-Compiler F77 (Version sc3.0.9) und der SISAL-Compiler OSC (Version 13.0.2) eingesetzt. Während der FORTRAN-Compiler direkt Maschinen-Code erzeugt, generiert der SISAL-Compiler ebenso wie der SAC-Compiler zunächst C-Code und verwendet den GCC (Version 2.6.3) zur erzeugung von Maschinen-Code.

Anstelle eines artifiziellen Multigrid-Beispiels wird der Multigrid-Kern MG der NAS-Benchmarks [BBB+94] als Basis für den Leistungsvergleich gewählt. Er realisiert eine vorspezifizierte Anzahl kompletter Multigrid-Zyklen auf einem dreidimensionalen Array mit $2^n, n \in \{3, 4, ...\}$ Elementen pro Achse als feinstem Gitter. Jeder Multigrid-Zyklus besteht aus einer schrittweisen Vergröberung hin zum gröbsten Gitter (4x4x4 Elemente), gefolgt von einer alternierenden Sequenz von Relaxationsschritten und Verfeinerungen.

Das verwendete FORTRAN-Programm entstammt direkt den Benchmarks[1] Das SISAL-Programm wurde aus dem FORTRAN-Programm abgeleitet, während das SAC-Programm die dimensionsunabhängigen Spezifizierungen aus Abschnitt 5.2 verwendet.

Drei verschiedene Problemgrößen mit bis zu 128 Elementen pro Achse wurden auf einen SUN ULTRASPARC-170 mit 192MB Speicher untersucht. Für jede Problemgröße wächst der Zeitverbrauch linear mit der Anzahl der durchgeführten Multigrid-Zyklen; der Speicherverbrauch dagegen bleibt konstant. Zeit- und Speicherverbrauch eines einzigen Multigrid-Zyklus sind daher charakteristisch für jede Problemgröße.

[1] Lediglich die initiale Array-Erzeugung wurde vereinfacht und die Problemgröße modifiziert.

Die Laufzeiten und der Speicherverbrauch wurden mittels des Timer- bzw. des Prozess-Status-Befehls des Betriebssystems gemessen. Aufgrund der Genauigkeit des Timer-Befehls von nur 1/50 Sekunde wurden die Laufzeiten für einzelne Multigrid-Zyklen von denen für mehrfache Multigrid-Zyklen abgeleitet. Bei den SAC- und FORTRAN-Implementierungen kann der Speicherverbrauch direkt durch den Prozess-Status-Befehl ermittelt werden. In SISAL ist dies aufgrund der eigenen Heap-Verwaltung durch das Laufzeitsystem von SISAL nicht so einfach möglich. Deshalb wurde zunächst eine minimale Heap-Vorgabe experimentell ermittelt, bevor der Speicherverbrach der SISAL-Implementierung gemessen wurde.

Abb. 5.2 zeigt den Vergleich von Zeit- und Speicherverbrauch für drei verschiedene Problemgrößen: 32, 64 und 128 Elemente pro Achse. Die Höhe der Balken im linken Diagramm stellt den Zeitverbrauch relativ zum Zeitverbrauch des SAC-Programms dar; die absoluten Laufzeiten für einen Multigrid-Zyklus sind in den Balken angemerkt.

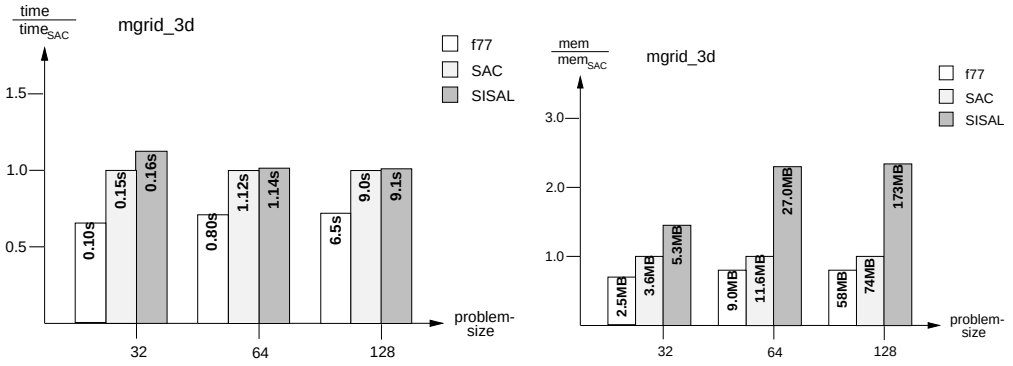


Abbildung 5.2: Laufzeit- und Speicherbedarf einer 3D Multigrid-Relaxation

Für alle drei untersuchten Problemgrößen kann beobachtet werden, daß die Laufzeit des SAC-Programms geringfügig unterhalb des Verbrauchs der SISAL-Implementierung liegt. Das FORTRAN-Programm dagegen benötigt durchschnittlich nur 70% der Laufzeit des SAC-Programs. Für den Unterschied zwischen der Laufzeit des FORTRAN- und des SAC- Programmes lassen sich mehrere Begründungen finden:

Erstens verursacht der Verfeinerungsalgorithmus in SAC aufgrund seiner dimensionsunabhängigen Spezifikation einen Mehraufwand, der zu einer Verlangsamung von ca. 10% führt.

Ein weiterer möglicher Grund liegt in der Tatsache, daß der benutzte SAC-Compiler in bezug auf die Speicherverwaltung sehr einfachen Code erzeugt. Er verfügt weder über ein eigenes Speicherverwaltungssystem noch über Optimierungen zur Minimierung der erforderlichen Speicherverwaltungsoperationen. Per Hand durchgeführte Modifikationen des C-Codes betreffend der Optimierung der Speicher-Operationen für einen einzelnen Relaxationsschritt deuten auf potentielle Leistungs-

steigerungen von ca. 9% für das gesamte Multigrid-Programm hin.

Darüber hinaus bietet die gegenwärtige Implementierung der WITH-Konstrukte Verbesserungsmöglichkeiten. Messungen handoptimierten Codes für einzelne Relaxationsschritte ergeben eine Beschleunigung um weitere 7%. Diese Experimente weisen darauf hin, daß durch die Integration weiterer Optimierungen in den SAC-Compiler FORTRAN-ähnliche Laufzeiten erreicht werden können.

Obwohl der SISAL-Compiler bereits sehr ausgefeilte Optimierungstechniken verwendet [SW85, Can89, CE95, FO95], ist das Laufzeitverhalten des von ihm erzeugten Codes nicht besser, als das der SAC-Implementierung. Der Grund dafür liegt in der Art und Weise, mit der in SISAL multidimensionale Arrays im Speicher verwaltet werden. Anstatt kontinuierliche Speicherblöcke für multidimensionale Arrays zu nutzen, implementiert der gegenwärtig verfügbare SISAL-Compiler Arrays als Vektoren von Vektoren, was zu beachtlichem Mehraufwand führt, sobald auf zueinander benachbarte Elemente zugegriffen werden muß[2].

Auf die gleiche Art wie das linke Diagramm relative Laufzeiten darstellt, bildet die rechte Graphik von Abb 5.2 den relativen Speicherverbrauch der drei Implementierungen ab. Es wird deutlich, daß die FORTRAN-Implementierung die effizienteste Speichernutzung ermöglicht. Durchschnittlich braucht sie nur etwa 80% des von der SAC-Implementierung benötigten Speicherplatzes. Diese Beziehung läßt sich an der größten Problemgröße erläutern. In diesem Fall beginnt der Algorithmus mit einem Array von 128x128x128 Gleitkommazahlen vom Typ DOUBLE, das dementsprechend $128 * 128 * 128 * 8 = 16,8$MB Speicherkapazität erfordert. Da der Multigrid-Algorithmus so wie er in [BBB+94] beschrieben ist mindestens drei Arrays dieser Größe benötigt, wird jede Implementierung mindestens $3 * 16,8 = 50,4$MB Speicher beanspruchen. Die bei den NAS-Benchmarks genutzte FORTRAN-Version benötigt 58MB Speicherplatz. Der Unterschied von 8MB zwischen dem Minimalwert und dem tatsächlichen Wert ergibt sich aus den Speicherbedarf des Programmes selbst, sowie dem Speicherbedarf für lokale Variablen und Arrays zur Repräsentation von gröberen Gittern. Die SAC-Implementierung benötigt 74MB Speicherkapazität, so daß sich ein Unterschied von 16MB zu der FORTRAN-Implementierung ergibt. Dies entspricht in etwa dem Speicherbedarf für ein Array des feinsten Gitters. Der Grund für diesen Mehrbedarf an Speicher resultiert aus den Unterschieden der Algorithmen, die die für die Verfeinerungsschritte verwendet werden. Während in der FORTRAN-Implementierung nur ein Array von der Größe des feineren Gitters benötigt wird, erfordert die dimensionsunabhängige SAC-Implementierung zwei: eins für die anfängliche Array-Erzeugung und ein zweites für das WITH-Konstrukt, das alle Elemente des feineren Arrays neu berechnet. Dieser zusätzliche Speicherbedarf könnte vermieden werden, wenn der Compiler in der Lage wäre, aufeinander folgende WITH-Konstrukte zusammenzufassen, sofern dies möglich ist. Für derartige

[2] Für die nächste SISAL-Version (Version 2.0) [BCOF91] ist eine Implementierung mehrdimensionaler Arrays durch zusammenhängende Speicherbereiche vorgesehen [Old92, FO95].

Optimierungen liefert der Ψ-Kalkül eine formale Grundlage. Eine Übersetzung aller WITH-Konstrukte in Ausdrücke des Ψ-Kalküls, gefolgt von einer systematischen Vereinfachung dieser Ausdrücke würde einen neuen Ausdruck des Ψ-Kalkül s ergeben, für den nur ein Array erforderlich wäre. Ein allgemeines Schema für solche Vereinfachungen ist jedoch noch nicht bekannt.

Vergleicht man den Speicherbedarf mit dem der SISAL-Implementierung, so zeigt sich, daß die SISAL-Implementierung erheblich mehr Speicher beansprucht als die beiden anderen Implementierungen. Im Fall des 128x128x128-Problems benötigt die SISAL-Implementierung 173MB Speicher - mehr als doppelt soviel wie die SAC-Implementierung. Für dieses Laufzeitverhalten konnte bisher keine befriedigende Erklärung gefunden werden. Wie bereits erwähnt, ist die Art und Weise, mit der der gegenwärtige SISAL-Compiler Arrays implementiert, für die Multigrid-Relaxation weniger geeignet als eine Implementierung mittels kontinuierlicher Speicherbereiche wie sie der SAC-Compiler verwendet. Um diese These zu untermauern und um den Einfluß des Mehraufwandes zu untersuchen, der durch die ungünstige Array-Repräsentation entsteht, wurde der SAC-Multigrid-Algorithmus auf zwei- und vierdimensionale Arrays angewendet und das SISAL-Programm entsprechend umgeschrieben. Das gemessene Laufzeitverhalten dieser Programmversionen ist in Abb. 5.3 dargestellt.

Wie in Abb 5.2 zeigen die Diagramme den relativen Zeit- und Speicherverbrauch für den zwei- und den vierdimensionalen Fall. Durch Anpassung der Problemgrößen beanspruchen die maximalen Arrays jeweils ca. 8.4MB Speicherkapazität.

Die Zahlen demonstrieren anschaulich den wachsenden Mehraufwand der Array-Repräsentation in SISAL bei ansteigenden Dimensionalitäten. Während der zweidimensionale Multigrid-Algorithmus nur etwa 70% der Zeit der SAC-Implementierung benötigt, ist das Verhältnis im vierdimensionalen Fall genau umgekehrt. Diese Beobachtung macht deutlich, daß die Integration von aufwendigen Optimierungstechniken, wie sie bereits im SISAL-Compiler eingesetzt werden, auch für die SAC-Implementierung substantielle Leistungsverbesserungen ermöglichen können.

Die Meßergebnisse zum Speicherverbrach zeigen, daß der relative Speicherverbrauch im Vergleich zur SAC-Implementierung bei wachsender Dimensionalität ansteigt. Während der Speicherverbrauch des SISAL-Programmes im zweidimensionalen Fall ungefähr das Doppelte des Verbrauchs der SAC-Implementierung ausmacht, wächst der Mehrverbrauch im vierdimensionalen Fall etwa auf das Dreifache des SAC-Wertes.

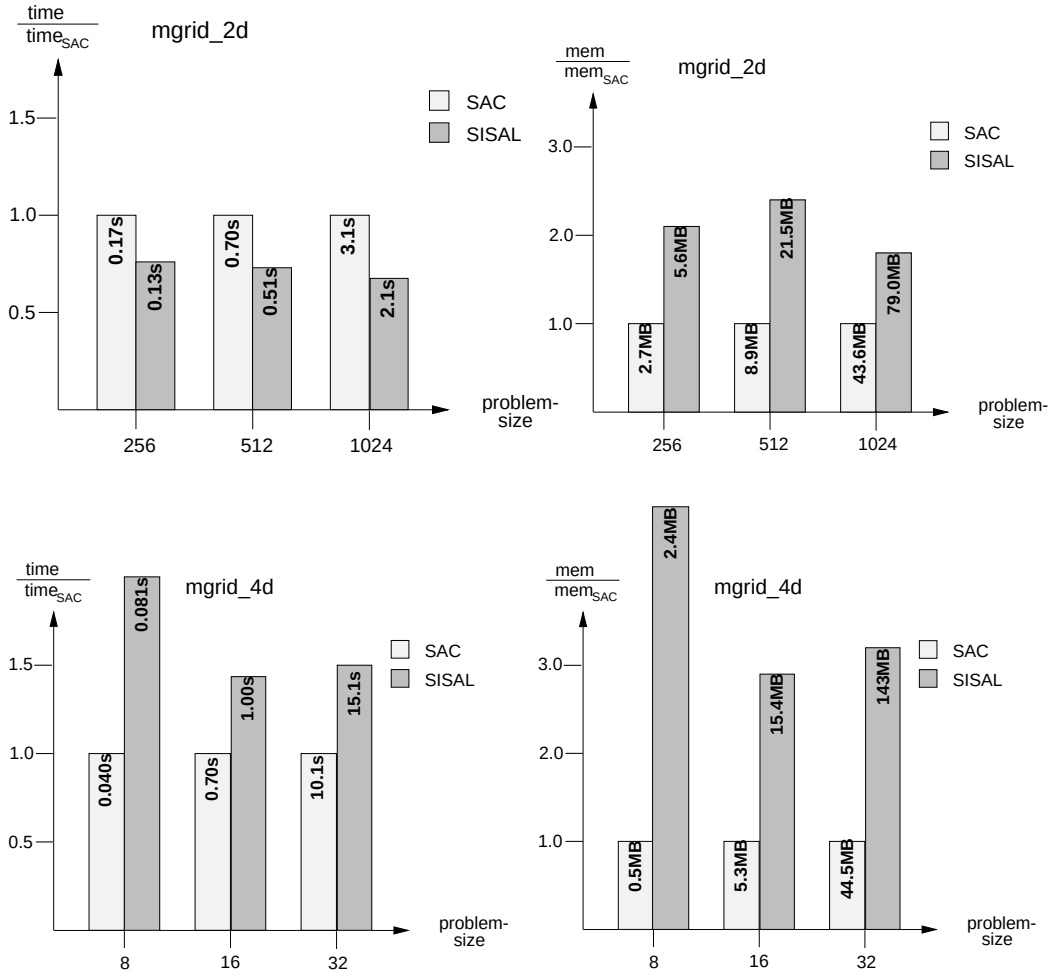


Abbildung 5.3: Laufzeit- und Speicherbedarf einer 2D/4D Multigrid-Relaxation

Kapitel 6

Zusammenfassung

In der vorliegenden Arbeit wird eine neue funktionale Programmiersprache SAC vorgestellt, deren Sprachdesign sich primär an den Bedürfnissen numerischer Anwendungen orientiert. Ihre Entwicklung war durch die Beobachtung motiviert, daß funktionale Sprachen trotz ihrer konzeptuellen Vorteile insbesondere in bezug auf die Verwendung komplexer Datenstrukturen sowie der Identifizierung nebenläufig auswertbarer Programmteile im Bereich der numerischen Anwendungen wenig praktische Verwendung finden. Aus Betrachtungen über die Stärken und Schwächen existierender funktionaler Sprachen ergaben sich folgende Designziele für SAC:

- eine Syntax, die den Umstieg von imperativen Sprachen erleichtert;
- ein hohes Abstraktionsniveau im Umgang mit Arrays;
- eine Integration von I/O-Operationen, die eine einfache Notation bei der Verwendung dieser Konstrukte zuläßt;
- eine Möglichkeit, in imperativen Sprachen spezifizierte Funktionen verwenden zu können, ohne diese anpassen zu müssen;
- eine Compilierbarkeit in effizient ausführbaren Code.

Die Syntax von SAC basiert auf einer Teilmenge der Syntax von C; mit Ausnahme von Zeigern und globalen Variablen sind fast alle Sprachkonstrukte von C Bestandteil von SAC. Dies wird durch die Definition einer funktionalen Semantik für SAC-Programme möglich. Die Ähnlichkeit von SAC- zu C-Programmen bietet mehrere Vorteile: Zum einen erleichtert sie C-gewohnten Programmierern den Umstieg, und zum anderen vereinfacht sie die Compilation nach C, von wo aus es Compiler für die meisten Hardware-Plattformen gibt.

Diese funktionale Variante von C wird um ein Array-System erweitert, das dimensionsunabhängige Spezifikationen komplexer Array-Operationen ermöglicht. Die entscheidende Voraussetzung für derartige Spezifikationen ist die Darstellung von

Arrays durch Shape- und Datenvektoren. Neben komplexen Array-Operationen, wie sie auch in APL verfügbar sind, werden in SAC spezielle Sprachkonstrukte zur elementweisen Spezifikation von Array-Operationen eingeführt: die sog. WITH-Konstrukte. Sie sind mit den Array-Comprehension-Konstrukten anderer funktionaler Sprachen vergleichbar, lassen sich im Gegensatz zu diesen jedoch bezüglich der Dimensionalität der zu erzeugenden bzw. modifizierenden Arrays parametrisieren. Dadurch können auch diese Konstrukte dazu verwendet werden, Array-Operationen dimensionsunabhängig zu spezifizieren.

Solche Spezifikationen bieten verschiedene Vorteile: Ein explizites Traversieren der Elemente eines Arrays mittels einer Schachtelung von FOR-Schleifen entfällt oder kann häufig ganz durch ein einziges WITH-Konstrukt ersetzt werden. Viele explizite und fehleranfällige Spezifikationen von Indizes entfallen ebenfalls. Für Indexberechnungen sind keine elementweisen Spezifikationen erforderlich, sondern es können Vektor-Operationen verwendet werden. Dadurch werden Programme konziser und somit ihre Wartung erleichtert. Darüberhinaus erhöht die Dimensionsunabhängigkeit der Spezifikation die Wiederverwendbarkeit von Programmteilen.

Außer dem Array-System bietet SAC ein Modulsystem, das dem von Sprachen wie MODULA, CLEAN oder HASKELL ähnelt. Zusammen mit den aus CLEAN bekannten Uniqueness-Typen stellt er die Basis für einen neuartigen Mechanismus zur Integration von Zuständen und Zustandsmodifikationen und damit zur Integration von I/O-Operationen dar. Dieser Mechanismus beruht auf der Idee, Uniqueness-Typen nicht explizit als Typattribute zur Verfügung zu stellen, sondern ihre Einführung ausschließlich über ausgezeichnete Typdefinitionen in speziellen Modulen, den sog. Klassen, zu ermöglichen. Datenstrukturen eines solchen Klassentyps (sog. Objekte) dürfen per definitionem nur so verwendet werden, daß jedes angewandte Vorkommen dieser Strukturen die Uniqueness-Eigenschaft erfüllt. Sie können deshalb als zustandsbehaftete Datenstrukturen angesehen werden.

Basierend auf den Klassentypen lassen sich zwei syntaktische Vereinfachungen einführen: sog. CALL-BY-REFERENCE-Parameter und „globale Objekte“. Die Verwendung von CALL-BY-REFERENCE-Parametern erlaubt es dem Programmierer, bei Funktionen, die den Zustand eines Objektes verändern, durch eine Annotation bei der Funktionsdeklaration auf eine explizite Rückgabe des modifizierten Objektes zu verzichten. Globale Objekte sind Objekte, die zu Beginn der Laufzeit eines Programmes erzeugt und bei dessen Terminierung wieder freigegeben werden. Sie können im gesamten Programm referenziert bzw. modifiziert werden, ohne daß sie explizit als Parameter übergeben werden müssen. Diese syntaktischen Vereinfachungen sind nur deshalb möglich, weil die konzeptuell benötigten Parameter und Rückgabewerte statisch inferiert werden können. Die Verwendung derartiger Konstrukte führt zu Notationen im Umgang mit Zuständen, die denen imperativer Sprachen gleichen. Dies hat verschiedene Vorteile: I/O-Operationen zum Zwecke der Fehlersuche während der Entwicklungs-Phase von Programmen lassen sich mit nur wenigen Programmänderungen einfügen. Außerdem lassen sich in imperativen Sprachen ge-

schriebene Programme, die Seiteneffekte verursachen, wie z.B. die `stdio`-Bibliothek von C, in SAC direkt integrieren. Schließlich können im Falle einer sequentiellen Berechnung auf einfache Weise die durch den Compiler inferierten zusätzlichen Parameter und Rückgabewerte wieder entfernt werden, was zu einer effizienten Implementierung führt.

Auf der Basis der in dieser Arbeit vorgestellten Compilations-Schemata ist die Implementierung eines Compilers von SAC nach C entstanden. Erste Laufzeitmessungen des erzeugten Codes zeigen, daß es trotz des hohen Abstraktionsniveaus gelingt, dimensionsunabhängig spezifizierte SAC-Programme in effizient ausführbaren Code zu compilieren. So werden bei Anwendungen wie z.B. einem Multigrid-Verfahren auf einem dreidimensionalen Array mit mehr als $2*10^6$ Elementen ähnliche Laufzeiten wie bei vergleichbaren, aber für eine feste Dimensionalität spezifizierten, SISAL-Programmen erreicht. Verantwortlich dafür ist vor allem die Einführung eines Typinferenzsystems, das eine Hierarchie von Array-Typen unterstützt. Ausgehend von der Form bzw. der Dimensionalität von Arrays in Argumentposition ermöglicht es dem Compiler, Funktionsspezialisierungen vorzunehmen, so daß aus dimensionsunabhängigen Spezifikationen dimensionsspezifischer Code erzeugt werden kann. Ein weiterer Grund für das gute Laufzeitverhalten liegt in den vorgestellten Optimierungen, insbesondere der sog. Index-Vector-Elimination. Durch sie wird die Erzeugung von Arrays zur Laufzeit vermieden, die entweder nur als Index in Arrays oder aber für die Berechnung solcher Indizes benötigt werden.

Die bisherigen Erfahrungen zeigen, daß alle Designziele erreicht werden konnten. Damit wird es dem Programmierer numerischer Anwendungen möglich, unter Weiterverwendung vorhandener Programmbibliotheken ohne größeren Lernaufwand von C auf SAC umzustellen. Die sich daraus ergebenden Vorteile, wie z.B. die Verfügbarkeit eines ausdrucksstarken Array-Systems müssen nicht mit Effizienzverlusten bezahlt werden.

Die im Zusammenhang mit der Entwicklung des Compilers gewonnenen Erfahrungen bieten außerdem verschiedene Ansatzpunkte für weitere Untersuchungen.

Das Array-System von SAC hat im Ψ-Kalkül eine formale Grundlage, die eine systematische Transformation von Array-Operationen ermöglicht. Daraus lassen sich verschiedene Optimierungen ableiten, die zur Vermeidung temporärer Arrays bei bestimmten Kompositionen von Array-Operationen dienen. Untersuchungen darüber, ob es ein minimierendes Transformationsschema für allgemeine Kompositionen von Array-Operationen gibt, sind daher für eine Weiterentwicklung des Compilers von größtem Interesse.

Aufgrund des guten Laufzeitverhaltens des erzeugten Codes ist mit dem existierenden Compiler die Grundlage dafür geschaffen, eine Code-Erzeugung für nebenläufige Programmausführungen zu integrieren. Von besonderem Interesse ist dabei die Fragestellung, inwieweit die vom Typsystem inferierten Informationen über die Form von Array-Ausdrücken sich als hilfreich für eine gleichmäßige Lastverteilung zur Laufzeit erweisen.

Anhang A

Die vollständige SAC-Syntax

$$
\begin{array}{lcl}
SACfile & \Rightarrow & Program \\
 & | & ModuleImp \\
 & | & ClassImp \\
 & | & ModuleDec \\
 & | & ClassDec
\end{array}
$$

A.1 Modul-/Klassen-Deklarationen

ModuleDec ⇒ *ModuleHeader* [*ImportBlock*]* OWN : *Declarations*

ModuleHeader ⇒ MODULE|DEC [EXTERNAL] *Id* :

ClassDec ⇒ *ClassHeader* [*ImportBlock*]* OWN : *Declarations*

ClassHeader ⇒ CLASS|DEC [EXTERNAL] *Id* :

ImportBlock ⇒ IMPORT *Id* : ALL ;
| IMPORT *Id* : *ImportList*

Declarations ⇒ { [*ITypeDec*] [*ETypeDec*] [*ObjDec*] [*FunDec*] }

ITypeDec ⇒ IMPLICIT TYPES : [*Id* ; [*ITypePragma*]*]*

ITypePragma ⇒ #PRAGMA COPYFUN *String*
| #PRAGMA FREEFUN *String*

ETypeDec ⇒ EXPLICIT TYPES : [*Id* = *Type* ;]*

ObjDec ⇒ GLOBAL OBJECTS : [*Type Id* ; [*ObjPragma*]*]*

ObjPragma ⇒ #PRAGMA INITFUN *String*

FunDec ⇒ FUNCTIONS : [*OneFunDec* [*FunPragma*]*]*

OneFunDec ⇒ *DecReturnList Id DecParamList* ;

FunPragma ⇒ #PRAGMA LINKNAME *String*
| #PRAGMA LINKSIGN [*Num* [, *Num*]*]
| #PRAGMA READONLY [*Num* [, *Num*]*]
| #PRAGMA REFCOUNTING [[*Num* [, *Num*]*]]
| #PRAGMA EFFECT [*Id* :] *Id* [, [*Id* :] *Id*]*
| #PRAGMA TOUCH [*Id* :] *Id* [, [*Id* :] *Id*]*

ImportList ⇒ { [*ITypeImp*] [*ETypeImp*] [*ObjImp*] [*FunImp*] }

ITypeImp ⇒ IMPLICIT TYPES : *Id* [, *Id*]* ;

ETypeImp ⇒ EXPLICIT TYPES : *Id* [, *Id*]* ;

ObjImp ⇒ GLOBAL OBJECTS : *Id* [, *Id*]* ;

FunImp ⇒ FUNCTIONS : *Id* [, *Id*]* ;

A.2 Typen

$$
\begin{array}{lcl}
\mathit{DecParamList} & \Rightarrow & \texttt{(\ )} \\
& | & \texttt{(\ ...\ )} \\
& | & \texttt{(}\ \mathit{Type}\ \texttt{[\ \&\ ][}\ \mathit{Id}\ \texttt{][\ ,\ }\mathit{Type}\ \texttt{[\ \&\ ][}\ \mathit{Id}\ \texttt{]]}^{*}\texttt{[\ ,\ ...\ ])} \\
\\
\mathit{DecReturnList} & \Rightarrow & \textsc{void} \\
& | & \texttt{...} \\
& | & \mathit{Type}\ \texttt{[\ ,\ }\mathit{Type}\ \texttt{]}^{*}\texttt{[\ ,\ ...\ ]} \\
\\
\mathit{ParamList} & \Rightarrow & \texttt{(\ )} \\
& | & \texttt{(}\ \mathit{Type}\ \texttt{[\ \&\ ]}\ \mathit{Id}\ \texttt{[\ ,\ }\mathit{Type}\ \texttt{[\ \&\ ]}\ \mathit{Id}\ \texttt{]}^{*}\ \texttt{)} \\
\\
\mathit{ReturnList} & \Rightarrow & \textsc{void} \\
& | & \mathit{Type}\ \texttt{[\ ,\ }\mathit{Type}\ \texttt{]}^{*} \\
\\
\mathit{Type} & \Rightarrow & \mathit{PrimType} \\
& | & \mathit{PrimType}\ \texttt{[\ [[}\ \mathit{Int}\ \texttt{,\ ]}^{*}\ \mathit{Int}\ \texttt{]]} \\
& | & \texttt{[}\ \mathit{Id}\ \texttt{:\ ]}\ \mathit{Id} \\
& | & \texttt{[}\ \mathit{Id}\ \texttt{:\ ]}\ \mathit{Id}\ \texttt{[\ [[}\ \mathit{Int}\ \texttt{,\ ]}^{*}\ \mathit{Int}\ \texttt{]]} \\
\\
\mathit{PrimType} & \Rightarrow & \textsc{int} \\
& | & \textsc{float} \mid \textsc{double} \\
& | & \textsc{bool} \mid \textsc{char}
\end{array}
$$

A.3 Programme

$$
\begin{array}{lcl}
Program & \Rightarrow & [\ ImportBlock\ ]^{*}\ Definitions\ Main \\
ModuleImp & \Rightarrow & \textsc{module}\ Id:\ [\ ImportBlock\ ]^{*}\ Definitions \\
ClassImp & \Rightarrow & \textsc{class}\ Id:\ [\ ImportBlock\ ]^{*}\ Definitions \\
Definitions & \Rightarrow & [\ TypeDef\ ]^{*} [\ ObjDef\ ]^{*} [\ FunDef\ ]^{*} \\
Main & \Rightarrow & \textsc{int main}\ (\ )\ ExprBlock \\
TypeDef & \Rightarrow & \textsc{typedef}\ Type\ Id\ ; \\
ObjDef & \Rightarrow & \textsc{objdef}\ Type\ Id\ =\ Expr\ ; \\
FunDef & \Rightarrow & [\ \textsc{inline}\ ]\ ReturnList\ Id\ ParamList\ ExprBlock
\end{array}
$$

A.3.1 Zuweisungen

$$
\begin{array}{lcl}
\mathit{ExprBlock} & \Rightarrow & \{\ [\ \mathit{VarDec}\ ]^{*}\ [\ \mathit{Instruction}\ ]^{*}\ [\ \mathit{Return}\ ]\ \} \\
\\
\mathit{VarDec} & \Rightarrow & \mathit{Type}\ \mathit{Id}\ [\ =\ \mathit{Expr}\ ]\ ; \\
\\
\mathit{Instruction} & \Rightarrow & \mathit{Assign}\ ; \\
 & | & \mathit{FunCall}\ ; \\
 & | & \mathit{SelAssign}\ ; \\
 & | & \mathit{ForAssign}\ ; \\
\\
\mathit{Assign} & \Rightarrow & \mathit{Id}\ [\ ,\ \mathit{Id}\ ]^{*}\ =\ \mathit{Expr} \\
 & | & \mathit{Id}\ [\ \mathit{Expr}\ ]\ =\ \mathit{Expr} \\
 & | & \mathit{Id}\ \mathit{AssignOperator}\ \mathit{Expr} \\
 & | & \mathit{IncDec} \\
\\
\mathit{AssignOperator} & \Rightarrow & +=\ |\ -=\ |\ *=\ |\ /= \\
\\
\mathit{Return} & \Rightarrow & \text{RETURN}\ (\ \mathit{Expr}\ [\ ,\ \mathit{Expr}\ ]^{*}\ )\ ; \\
\\
\mathit{IncDec} & \Rightarrow & \mathit{Id}\ ++\ |\ \mathit{Id}\ -- \\
 & | & ++\ \mathit{Id}\ |\ --\ \mathit{Id} \\
\\
\mathit{SelAssign} & \Rightarrow & \text{IF}\ (\ \mathit{Expr}\ )\ \mathit{AssignBlock}\ \text{ELSE}\ \mathit{AssignBlock} \\
\\
\mathit{ForAssign} & \Rightarrow & \text{DO}\ \mathit{AssignBlock}\ \text{WHILE}\ (\ \mathit{Expr}\ )\ ; \\
 & | & \text{WHILE}\ (\ \mathit{Expr}\ )\ \mathit{AssignBlock} \\
 & | & \text{FOR}\ (\ \mathit{Assign}\ ;\ \mathit{Expr}\ ;\ \mathit{Assign}\ )\ \mathit{AssignBlock} \\
\\
\mathit{AssignBlock} & \Rightarrow & \mathit{Instruction} \\
 & | & \{\ [\ \mathit{Instruction}\ ]^{+}\ \}
\end{array}
$$

A.3.2 Ausdrücke

$$
\begin{array}{lcl}
Expr & \Rightarrow & [\, Id : \,]\ Id \\
 & | & Id\ [\ Expr\ ] \\
 & | & WithExpr \\
 & | & FunCall \\
 & | & IncDec \\
 & | & PrimFunAp \\
 & | & (\ \ Expr\ \ ) \\
 & | & [\ Expr\, [\ ,\ \ Expr\,]^{*}\ \ ] \\
 & | & Id\ \ [\ \ Expr\ \ ] \\
 & | & (\ \ : PrimType\ ) \ Expr \\
 & | & (\ \ : Id\ ) \ Expr \\
 & | & StringConstant \\
 & | & IntegerConstant \\
 & | & FloatConstant\, [\, Specifier\,] \\
 & | & \texttt{TRUE} \mid \texttt{FALSE}
\end{array}
$$

$$
\begin{array}{lcl}
PrimFunAp & \Rightarrow & ArrayFunAp \\
 & | & ConvertFunAp \\
 & | & Expr\ BinPrimFun\ Expr
\end{array}
$$

$$
\begin{array}{lcl}
ConvertFunAp & \Rightarrow & \texttt{TOI}\ \ (\ Expr\ ) \\
 & | & \texttt{TOF}\ \ (\ Expr\ ) \\
 & | & \texttt{TOD}\ \ (\ Expr\ )
\end{array}
$$

$$
\begin{array}{lcl}
BinPrimFun & \Rightarrow & + \mid - \mid * \mid / \\
 & | & < \mid <= \mid > \mid >= \mid == \mid != \\
 & | & \&\& \mid ||
\end{array}
$$

$$
Specifier \Rightarrow \texttt{f} \mid \texttt{F} \mid \texttt{d} \mid \texttt{D}
$$

$$
FunCall \Rightarrow [\, Id : \,]\ Id\ (\ [\, Expr [\ ,\ Expr\,]^{*}\ ]\ )
$$

A.3.3 Array-Operationen

$$
\begin{array}{lcl}
\mathit{WithExpr} & \Rightarrow & \texttt{WITH} \ \texttt{(} \ \mathit{Generator} \ \texttt{)} \ \texttt{[} \ \mathit{AssignBlock} \ \texttt{]} \ \mathit{ConExpr} \\
\\
\mathit{Generator} & \Rightarrow & \mathit{Expr} \ \texttt{<=} \ \mathit{Id} \ \texttt{<=} \ \mathit{Expr} \\
\\
\mathit{ConExpr} & \Rightarrow & \texttt{GENARRAY} \ \texttt{(} \ \mathit{ConstVec} \ \texttt{,} \ \mathit{Expr} \ \texttt{)} \\
 & | & \texttt{MODARRAY} \ \texttt{(} \ \mathit{Expr} \ \texttt{,} \ \mathit{Id} \ \texttt{,} \ \mathit{Expr} \ \texttt{)} \\
 & | & \texttt{FOLD} \ \texttt{(} \ \mathit{FoldFun} \ \texttt{,} \ \mathit{Expr} \ \texttt{,} \ \mathit{Expr} \ \texttt{)} \\
\\
\mathit{FoldFun} & \Rightarrow & \mathit{FoldOp} \\
 & | & \texttt{[} \ \mathit{Id} \ \texttt{:} \ \texttt{]} \ \mathit{Id} \\
\\
\mathit{FoldOp} & \Rightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
\\
\mathit{ArrayFunAp} & \Rightarrow & \texttt{DIM} \ \texttt{(} \ \mathit{Expr} \ \texttt{)} \\
 & | & \texttt{SHAPE} \ \texttt{(} \ \mathit{Expr} \ \texttt{)} \\
 & | & \texttt{PSI} \ \texttt{(} \ \mathit{Expr} \ \texttt{,} \ \mathit{Expr} \ \texttt{)} \\
 & | & \texttt{TAKE} \ \texttt{(} \ \mathit{ConstVec} \ \texttt{,} \ \mathit{Expr} \ \texttt{)} \\
 & | & \texttt{DROP} \ \texttt{(} \ \mathit{ConstVec} \ \texttt{,} \ \mathit{Expr} \ \texttt{)} \\
 & | & \texttt{RESHAPE} \ \texttt{(} \ \mathit{ConstVec} \ \texttt{,} \ \mathit{Expr} \ \texttt{)} \\
 & | & \texttt{CAT} \ \texttt{(} \ \mathit{ConstVec} \ \texttt{,} \ \mathit{Expr} \ \texttt{,} \ \mathit{Expr} \ \texttt{)} \\
 & | & \texttt{ROTATE} \ \texttt{(} \ \mathit{ConstVec} \ \texttt{,} \ \mathit{ConstVec} \ \texttt{,} \ \mathit{Expr} \ \texttt{)} \\
 & | & \texttt{GENARRAY} \ \texttt{(} \ \mathit{ConstVec} \ \texttt{,} \ \mathit{Expr} \ \texttt{)} \\
 & | & \texttt{MODARRAY} \ \texttt{(} \ \mathit{Id} \ \texttt{,} \ \mathit{Expr} \ \texttt{,} \ \mathit{Expr} \ \texttt{)}
\end{array}
$$

Anhang B

Die Typinferenzregeln

B.1 Allgemeine Regeln

B.1.1 Ausdruckbasierte Regeln

$$\textsc{Const} \quad : \quad \frac{}{\mathrm{A}\vdash Const : TYPE(Const)} \quad ,$$

wobei $TYPE$ jeder Konstanten ihren atomaren Typ zuordnet

$$\textsc{Var} \quad : \quad \frac{}{\mathrm{A}\vdash v:\tau} \quad \Longleftrightarrow \quad (v:\tau)\in \mathrm{A}$$

$$\textsc{Cast} \quad : \quad \frac{\mathrm{A}\vdash e:\sigma}{\mathrm{A}\vdash (:\tau)e:\tau} \quad \Longleftrightarrow \quad Basetype(\sigma) = Basetype(\tau)$$

$$\textsc{Return} \quad : \quad \frac{\mathrm{A}\vdash e_i:\tau_i}{\mathrm{A}\vdash \textsc{return}(e_1,\ ...,\ e_n):\bigotimes_{i=1}^{n}\tau_i}$$

$$\textsc{FunAp} \quad : \quad \frac{\mathrm{A}\vdash e_i:\sigma_i \quad \{\}\vdash \rho F(\rho_1\ v_1,\ ...,\ \rho_n\ v_n)\ B:\bigotimes_{i=1}^{n}\tau_i \to \tau}{\mathrm{A}\vdash F(e_1 ... e_n):\tau}$$

$$\Longleftrightarrow \forall i \in \{1,...,n\} : \sigma_i \preceq \tau_i \preceq \rho_i \wedge \tau \preceq \rho$$

$$\textsc{PrfAp} \quad : \quad \frac{\mathrm{A}\vdash e_i:\tau_i}{\mathrm{A}\vdash F(e_1,\ ...,\ e_n):\tau} \quad \Longleftrightarrow \quad TYPE(\ F) = \bigotimes_{i=1}^{n}\tau_i \to \tau \quad ,$$

wobei $TYPE$ jeder primitiven Funktion ihren Typ zuordnet

B.1.2 Programmsegmentbasierte Regeln

$$\textsc{Prg} \quad : \quad \frac{\{\} \vdash \tau \ \textsc{main}() \ Body : \sigma}{\{\} \vdash \ FunDef_1, \ ..., FunDef_n \ \tau \ \textsc{main}() \ \{Body\} : \sigma}$$

$$\Longleftrightarrow \sigma \preceq \tau$$

$$\textsc{FunDef1} \quad : \quad \frac{\{\} \vdash Body : \sigma}{\{\} \vdash \tau \ F() \ \{Body\} : \sigma}$$

$$\Longleftrightarrow \sigma \preceq \tau$$

$$\textsc{FunDef2} \quad : \quad \frac{\{v_i : \tau_i\} \vdash B : \tau}{\mathrm{A} \vdash \sigma F(\sigma_1 \ v_1, \ ..., \ \sigma_n \ v_n) \ B : \bigotimes_{i=1}^{n} \tau_i \rightarrow \tau}$$

$$\Longleftrightarrow \tau_i \preceq \sigma_i \wedge \tau \preceq \sigma$$

$$\textsc{VarDec} \quad : \quad \frac{\mathrm{A}\{v : \tau\} \vdash Rest : \sigma \quad \mathrm{A} \vdash Rest : \sigma'}{\mathrm{A} \vdash \tau \ v; \ Rest : \sigma''}$$

$$\Longleftrightarrow \neg\exists \ (v : \rho) \in \mathrm{A} \text{ mit } \rho \neq \tau \\ \wedge \ (\sigma \sim \sigma') \wedge (\sigma'' = \min(\ \sigma, \ \sigma'))$$

$$\textsc{Let} \quad : \quad \frac{\mathrm{A} \vdash e : \bigotimes_{i=1}^{n} \sigma_i \quad \mathrm{A}\{v_i : \tau_i\} \vdash R : \tau}{\mathrm{A} \vdash v_1, \ ..., \ v_n = e; \ R : \tau}$$

$$\Longleftrightarrow \forall i \in \{1, ..., n\} : \\ (\exists \ (\mathrm{v}_i : \rho_i) \in \mathrm{A} \Rightarrow \rho_i \sim \sigma_i \wedge \tau_i = \min(\ \rho_i, \ \sigma_i) \\ (\neg\exists \ (\mathrm{v}_i : \rho_i) \in \mathrm{A} \Rightarrow \tau_i = \sigma_i)$$

B.1.3 IF-THEN-ELSE-Konstrukte

$$\textsc{Cond1} \quad : \quad \frac{\mathrm{A} \vdash e : \textsc{bool} \quad \mathrm{A} \vdash Ass_t; \ Rest : \tau \quad \mathrm{A} \vdash Rest : \tau'}{\mathrm{A} \vdash \textsc{if} \ (e) \ \{Ass_t;\} \ Rest : \tau''}$$

$$\Longleftrightarrow (\tau \sim \tau') \wedge (\tau'' = \min(\ \tau, \ \tau'))$$

$$\textsc{Cond2} \quad : \quad \frac{\mathrm{A} \vdash e : \textsc{bool} \quad \mathrm{A} \vdash Ass_t; \ Rest : \tau \quad \mathrm{A} \vdash Ass_e; \ Rest : \tau'}{\mathrm{A} \vdash \textsc{if} \ (e) \ \{Ass_t;\} \ \textsc{else} \ \{Ass_e;\}; \ Rest : \tau''}$$

$$\Longleftrightarrow (\tau \sim \tau') \wedge (\tau'' = \min(\ \tau, \ \tau'))$$

B.1.4 Schleifenkonstrukte

$$\textsc{Do} \quad : \quad \frac{\mathrm{A}\vdash e:\textsc{bool} \quad \mathrm{A}\vdash Ass;\, Rest:\tau}{\mathrm{A}\vdash \textsc{do}\ \{Ass;\}\ \textsc{while}\ (e);\, Rest:\tau}$$

$$\textsc{While} \quad : \quad \frac{\mathrm{A}\vdash e:\textsc{bool} \quad \mathrm{A}\vdash Rest:\tau \quad \mathrm{A}\vdash Ass;\, Rest:\tau'}{\mathrm{A}\vdash \textsc{while}\ (e)\ \{Ass;\};\, Rest:\tau''}$$

$$\Longleftrightarrow (\tau \sim \tau') \wedge (\tau'' = \min(\ \tau,\ \tau'))$$

$$\textsc{For} \quad : \quad \frac{\mathrm{A}\vdash e:\textsc{bool} \quad \mathrm{A}\vdash Ass_1;\, Rest:\tau \quad \mathrm{A}\vdash Ass_1;\, Ass_3;\, Ass_2;\, Rest:\tau'}{\mathrm{A}\vdash \textsc{for}\ (Ass_1;\, e;\, Ass_2)\ \{Ass_3;\};\, Rest:\tau''}$$

$$\Longleftrightarrow (\tau \sim \tau') \wedge (\tau'' = \min(\ \tau,\ \tau'))$$

B.2 Array-Operationen

B.2.1 Forminspizierende Operationen

$$\textsc{Dim} \quad : \quad \frac{\mathrm{A}\vdash a:\tau}{\mathrm{A}\vdash \textsc{dim}(a):\textsc{int}} \quad \Longleftrightarrow \quad \tau \in \mathcal{T}_{Array}$$

$$\textsc{Shape1} \quad : \quad \frac{\mathrm{A}\vdash a:\tau}{\mathrm{A}\vdash \textsc{shape}(a):\textsc{int}[n]} \quad \Longleftrightarrow \quad \tau \preceq \sigma[\overbrace{\bullet, ..., \bullet}^{n}]$$

$$\textsc{Shape2} \quad : \quad \frac{\mathrm{A}\vdash a:\tau[\,]}{\mathrm{A}\vdash \textsc{shape}(a):\textsc{int}[\bullet]}$$

B.2.2 Selektion

$$\textsc{Psi1} \quad : \quad \frac{\mathrm{A}\vdash s:\textsc{int}[n] \quad \mathrm{A}\vdash a:\tau}{\mathrm{A}\vdash \textsc{psi}(\ s,\ a):\sigma} \quad \Longleftrightarrow \quad \tau \preceq \sigma[\overbrace{\bullet, ..., \bullet}^{n}]$$

$$\textsc{Psi2} \quad : \quad \frac{\mathrm{A}\vdash s:\textsc{int}[n] \quad \mathrm{A}\vdash a:\tau}{\mathrm{A}\vdash \textsc{psi}(\ s,\ a):\sigma[r_{(n+1)}, ..., r_m]}$$

$$\Longleftrightarrow \tau = \sigma[r_1, ..., r_m] \text{ mit } n < m \text{ und } r_i \in \mathbb{N}_\bullet$$

$$\textsc{Psi3} \quad : \quad \frac{\mathrm{A}\vdash s:\rho \quad \mathrm{A}\vdash a:\tau}{\mathrm{A}\vdash \textsc{psi}(\ s,\ a):\sigma[?]}$$

$$\Longleftrightarrow (\rho{=}\ \textsc{int}[n] \wedge \tau{=}\sigma[\,]) \vee (\textsc{int}[\bullet] \preceq \rho \wedge \tau \preceq \sigma[\,])$$

B.2.3 Formverändernde Operationen

$$\textsc{Reshape1} : \frac{\mathrm{A}\vdash s:\textsc{int}[n] \quad \mathrm{A}\vdash a:\tau}{\mathrm{A}\vdash \textsc{reshape}(s,a):\sigma[\overbrace{\bullet,\ldots,\bullet}^{n}]} \iff \tau \preceq \sigma[\,]$$

$$\textsc{Reshape2} : \frac{\mathrm{A}\vdash s:\rho \quad \mathrm{A}\vdash a:\tau}{\mathrm{A}\vdash \textsc{reshape}(s,a):\sigma[\,]} \iff \tau \preceq \sigma[\,] \wedge \textsc{int}[\bullet] \preceq \rho$$

$$\textsc{Take1} : \frac{\mathrm{A}\vdash s:\textsc{int}[m] \quad \mathrm{A}\vdash a:\sigma[s_1,\ldots,s_n]}{\mathrm{A}\vdash \textsc{take}(s,a):\sigma[\overbrace{\bullet,\ldots,\bullet}^{m},s_{(m+1)},\ldots,s_n]} \iff s_i \in \mathbb{N}_\bullet$$

$$\textsc{Take2} : \frac{\mathrm{A}\vdash s:\rho \quad \mathrm{A}\vdash a:\sigma[\,]}{\mathrm{A}\vdash \textsc{take}(s,a):\sigma[\,]} \iff \textsc{int}[n] \preceq \rho$$

$$\textsc{Drop1} : \frac{\mathrm{A}\vdash s:\textsc{int}[m] \quad \mathrm{A}\vdash a:\sigma[s_1,\ldots,s_n]}{\mathrm{A}\vdash \textsc{drop}(s,a):\sigma[\overbrace{\bullet,\ldots,\bullet}^{m},s_{(m+1)},\ldots,s_n]} \iff s_i \in \mathbb{N}_\bullet$$

$$\textsc{Drop2} : \frac{\mathrm{A}\vdash s:\rho \quad \mathrm{A}\vdash a:\sigma[\,]}{\mathrm{A}\vdash \textsc{drop}(s,a):\sigma[\,]} \iff \textsc{int}[n] \preceq \rho$$

$$\textsc{Cat1} : \frac{\mathrm{A}\vdash d:\textsc{int} \quad \mathrm{A}\vdash a:\tau \quad \mathrm{A}\vdash b:\rho}{\mathrm{A}\vdash \textsc{cat}(d,a,b):\sigma[\overbrace{\bullet,\ldots,\bullet}^{n}]}$$
$$\iff (\exists r_i, s_i \in \mathbb{N}_\bullet : \sigma[s_1,\ldots,s_n] \preceq \tau \wedge \sigma[r_1,\ldots,r_n] \preceq \rho) \wedge \neg(\tau = \rho = \sigma[\,])$$

$$\textsc{Cat2} : \frac{\mathrm{A}\vdash d:\textsc{int} \quad \mathrm{A}\vdash a:\sigma[\,] \quad \mathrm{A}\vdash b:\sigma[\,]}{\mathrm{A}\vdash \textsc{cat}(d,a,b):\sigma[\,]}$$

B.2.4 Wertverändernde Operationen

$$\textsc{AriAS} \;:\; \frac{\mathrm{A}\vdash a:\tau \quad \mathrm{A}\vdash s:\sigma}{\mathrm{A}\vdash \textsc{AriOp}(a,\,s):\tau} \qquad \Longleftrightarrow \qquad \tau \preceq \sigma[\,]$$

$$\textsc{AriSA} \;:\; \frac{\mathrm{A}\vdash s:\sigma \quad \mathrm{A}\vdash a:\tau}{\mathrm{A}\vdash \textsc{AriOp}(s,\,a):\tau} \qquad \Longleftrightarrow \qquad \tau \preceq \sigma[\,]$$

$$\textsc{AriAA} \;:\; \frac{\mathrm{A}\vdash a:\tau \quad \mathrm{A}\vdash b:\sigma}{\mathrm{A}\vdash \textsc{AriOp}(a,\,b):\rho}$$

$$\Longleftrightarrow (\rho = \tau \preceq \sigma) \;\vee\; (\rho = \sigma \preceq \tau)$$

$$\textsc{Rot} \;:\; \frac{\mathrm{A}\vdash d:\textsc{int} \quad \mathrm{A}\vdash n:\textsc{int} \quad \mathrm{A}\vdash a:\tau}{\mathrm{A}\vdash \textsc{rotate}(\,d,\,n,\,a):\tau}$$

$$\textsc{Mod1} \;:\; \frac{\mathrm{A}\vdash s:\rho \quad \mathrm{A}\vdash v:\sigma \quad \mathrm{A}\vdash a:\tau}{\mathrm{A}\vdash \textsc{modarray}(\,s,\,v,\,a):\tau}$$

$$\Longleftrightarrow \exists s_i \in \mathbb{N}_\bullet : \sigma[s_1,...,s_n] \preceq \tau \wedge \textsc{int}[n] \preceq \rho$$

$$\textsc{Mod2} \;:\; \frac{\mathrm{A}\vdash s:\rho \quad \mathrm{A}\vdash v:\sigma \quad \mathrm{A}\vdash a:\tau}{\mathrm{A}\vdash \textsc{modarray}(\,s,\,v,\,a):\tau}$$

$$\Longleftrightarrow \exists r_i, s_i \in \mathbb{N}_\bullet : \begin{array}{l} \sigma'[s_1,...,s_m] \preceq \sigma \wedge \textsc{int}[n] \preceq \rho \\ \quad \wedge\; \sigma'[r_1,...,r_n,s_1,...,s_m] \preceq \tau \end{array}$$

$$\textsc{Gen1} \;:\; \frac{\mathrm{A}\vdash s:\textsc{int}[n] \quad \mathrm{A}\vdash a:\sigma}{\mathrm{A}\vdash \textsc{genarray}(\,s,\,a):\sigma[\overbrace{\bullet,...,\bullet}^{n}]}$$

$$\textsc{Gen2} \;:\; \frac{\mathrm{A}\vdash s:\textsc{int}[n] \quad \mathrm{A}\vdash a:\sigma[s_1,...,s_m]}{\mathrm{A}\vdash \textsc{genarray}(\,s,\,a):\sigma[\overbrace{\bullet,...,\bullet}^{n},s_1,...,s_m]} \qquad \Longleftrightarrow \qquad s_i \in \mathbb{N}_\bullet$$

$$\textsc{Gen3} \;:\; \frac{\mathrm{A}\vdash s:\rho \quad \mathrm{A}\vdash a:\tau}{\mathrm{A}\vdash \textsc{genarray}(\,s,\,a):\sigma[\,]}$$

$$\Longleftrightarrow (\textsc{int}[\bullet] \preceq \rho \wedge \tau \preceq \sigma[?]) \vee (\textsc{int}[n] \preceq \rho \wedge \tau = \sigma[\,])$$

B.2.5 WITH-Konstrukte

$$\textsc{With}1 \;:\; \frac{\begin{array}{c} A\vdash e_1:\tau_1 \quad A\vdash e_2:\tau_2 \quad A\vdash e_{shape}:\tau_3 \quad A\{v:\tau\}\vdash f_i:\textsc{bool} \\ A\{v:\tau\}\vdash \{Ass;\ \textsc{return}(e_{val});\}:\rho \\ A\{dummy\_var:\rho\}\vdash \textsc{genarray}(e_{shape},\ dummy\_var):\sigma \end{array}}{A\vdash \textsc{with}(e_1\le v\le e_2, f_1,\ldots,f_n)\{Ass;\}\ \textsc{genarray}(e_{shape},\ e_{val}):\sigma}$$

$$\Longleftrightarrow \exists \tau\in\mathcal{T}_{\mathrm{SAC}} : \tau = \min(\ \tau_1,\ \min(\ \tau_2,\ \tau_3)) = \min(\ \min(\ \tau_1,\ \tau_2),\ \tau_3)$$

$$\textsc{With}2 \;:\; \frac{\begin{array}{c} A\vdash e_1:\tau_1 \quad A\vdash e_2:\tau_2 \quad A\{v:\tau\}\vdash f_i:\textsc{bool} \\ A\{v:\tau\}\vdash \{Ass;\ \textsc{return}(e_{val});\}:\rho \\ A\{v:\tau,\ dummy\_var:\rho\}\vdash \textsc{modarray}(\ v,\ dummy\_var,\ e_{array}):\sigma \end{array}}{A\vdash \textsc{with}(e_1\le v\le e_2, f_1,\ldots,f_n)\{Ass;\}\ \textsc{modarray}(\ v,\ e_{val},\ e_{array}):\sigma}$$

$$\Longleftrightarrow \exists \tau\in\mathcal{T}_{\mathrm{SAC}} : \tau = \min(\ \tau_1,\ \tau_2)$$

$$\textsc{With}3 \;:\; \frac{\begin{array}{c} A\vdash e_1:\tau_1 \quad A\vdash e_2:\tau_2 \quad A\{v:\tau\}\vdash f_i:\textsc{bool} \\ A\{v:\tau\}\vdash \{Ass;\ \textsc{return}(e_{val});\}:\sigma \\ A\vdash e_{identity}:\sigma \quad A\vdash fun:\sigma\times\sigma\to\sigma \end{array}}{A\vdash \textsc{with}(e_1\le v\le e_2, f_1,\ldots,f_n)\{Ass;\}\ \textsc{fold}(\ fun,\ e_{identity},\ e_{val}):\sigma}$$

$$\Longleftrightarrow \exists \tau\in\mathcal{T}_{\mathrm{SAC}} : \tau = \min(\ \tau_1,\ \tau_2)$$

Anhang C

Die Intermediate-Code-Macros

C.1 ICM-Befehle für Funktionsaufrufe

FUNDEC(*name*, tag_1, $type_1$, arg_1, ..., tag_m, $type_m$, arg_m) erzeugt eine Deklaration für eine Funktion mit dem Namen *name*. Jede Gruppe tag_i, $type_i$, arg_i repräsentiert ein Argument bzw. Resultat der Funktion. Die Marke tag_i zeigt an, ob es sich um ein Argument (tag_i = IN) oder ein Resultat (tag_i = OUT) handelt und ob dieses einen Referenzzähler hat (tag_i = IN_RC bzw. tag_i = OUT_RC) oder nicht (tag_i = IN bzw. tag_i = OUT). Im Falle eines Argumentes bezeichnen $type_i$ und arg_i Typ und Namen des Argumentes; bei Resultaten stehen $type_i$ und arg_i für Typ und Namen der Variablen im RETURN-Ausdruck.

FUNRET(tag_1, arg_1, ..., tag_n, arg_n) erzeugt eine RETURN-Anweisung sowie ggf. zusätzlich benötigte Anweisungen, um der aufrufenden Funktion mehrere Resultate zugänglich zu machen. Dabei bezeichnen die einzelnen arg_i Namen der Variablen im ursprünglichen RETURN-Ausdruck. Die Marken tag_i zeigen an, ob die Resultate einen Referenzzähler haben (tag_i = OUT_RC) oder nicht (tag_i = OUT).

FUNAP(*name*, tag_1, arg_1, ..., tag_m, arg_m) erzeugt einen Funktionsaufruf der Funktion *name* sowie ggf. zusätzlichen C-Programmtext, um die Resultate den entsprechenden Variablen zuzuweisen. Ähnlich wie bei dem ICM-Befehl FUNDEC steht jeweils ein Paar bestehend aus einer Marke tag_i und einem Namen arg_i für je ein Argument bzw. eine Resultatvariable.

C.2 ICM-Befehle zur Integration von Arrays

DECLARRAY(*name*, τ, s_1, ..., s_n) erzeugt die Variablendeklaration(en), die benötigt werden, um in C eine SAC-Array-Variable *name* mit dem Typ $\tau[s_1, ..., s_n]$ mit Referenzzähler darzustellen.

ALLOCARRAY(τ, *name*, n) erzeugt Befehle zum Allozieren von Speicher für das Array *name* und initialisiert den Referenzzähler mit n. τ gibt den Typ der Elemente an.

CREATECONSTARRAYS(*name*, d_1, ..., d_n) erzeugt Befehle zum Initialisieren des Arrays *name* mit den Werten d_1, ..., d_n. Der Referenzzähler des Arrays bleibt dabei unberührt.

CREATECONSTARRAYA(*name*, d_1, ..., d_n) entspricht CREATECONSTARRAYS; anstelle skalarer Daten handelt es sich bei den d_i um Array-Variablen.

ASSIGNARRAY(*name*$_1$, *name*$_2$) erzeugt eine Zuweisung *name*$_1$ = *name*$_2$ von Array-Variablen. Der Referenzzähler, auf den via *name*$_1$ bzw. *name*$_2$ zugegriffen werden kann, wird dabei nicht verändert.

INCRC(*name*, n) erhöht den Referenzzähler des Arrays *name* um n.

DECRCFREEARRAY(*name*, n) erniedrigt den Referenzzähler des Arrays *name* um n. Wird der Referenzzähler dabei 0, so wird der zugehörige Speicherbereich freigegeben.

C.3 ICM-Befehle für Array-Operationen

PSIVXA_S(*array*, *result*, *len_idx*, *idx_vect*) erzeugt eine Zuweisung des durch den Indexvektor *idx_vect* in *array* selektierten, skalaren Array-Elementes an die Variable *result*. *len_idx* spezifiziert die Dimensionalität von *array* bzw. die Länge des Zugriffsvektors *idx_vect*. Die Referenzzähler von *array* und *idx_vect* bleiben unverändert.

PSIVXA_A(*dim_array*, *array*, *result*, *len_idx*, *idx_vect*) erzeugt eine Zuweisung des durch *idx_vect* in *array* selektierten Teil-Arrays von *array* an die Variable *result*. Da es sich beim Resultat um ein Array handelt, werden zunächst jedoch Befehle zur Speicherallozierung für das Resultat sowie die Initialisierung des Referenzzähler mit dem Wert 1 erzeugt. *dim_array* spezifiziert die Dimensionalität von *array*, *len_idx* die Länge des Zugriffsvektors *idx_vect*. Die Referenzzähler von *array* und *idx_vect* bleiben unverändert.

C.4 ICM-Befehle für WITH-Konstrukte

BEGINMODARRAY(*res*, *dim_res*, *src*, *start*, *stop*, *idx*, *idx_len*)
erzeugt eine Schachtelung von WHILE-Schleifen, um ein WITH-Konstrukt mit MODARRAY-Operation zu realisieren. Dabei variiert die Generatorvariable *idx*

mit *idx_len* Komponenten von *start* bis *stop* ohne Berücksichtigung der Referenzzähler dieser drei Vektoren. Außerdem wird eine Initialisierung des Speichers des *dim_res*-dimensionalen Resultat-Arrays *res* für alle Indexpositionen kleiner *start* mit den entsprechenden Array-Elementen von *src* vorgenommen. Sowohl der Referenzzähler von *src* als auch der Zähler von *res* bleiben dabei unverändert.

EndModarrayS(*res*, *dim_res*, *src*, *val*) stellt eines der möglichen Gegenstücke zu BeginModarray dar. Dazu wird zunächst der im Rumpf des WITH-Konstruktes erzeugte skalare Wert *val* an die aktuelle Indexposition des Arrays *res* mit der Dimensionalität *dim_res* kopiert und anschließend werden die durch BeginModarray erzeugten Schleifenkonstrukte beendet. Schließlich werden noch die fehlenden Initialisierungen für die Indexpositionen größer *stop* (aus dem zug. BeginModarray-Konstrukt) des Resultat-Arrays *res* durch Kopieren der entsprechenden Elemente von *src* vorgenommen. Die Referenzzähler aller involvierten Arrays bleiben dabei unverändert.

EndModarrayA(*res*, *dim_res*, *src*, *val*, *idx_len*) entspricht im wesentlichen dem EndModarrayS-Befehl. Der Unterschied besteht ausschließlich darin, daß es sich bei *val* um ein Array der Dimension ($dim\_res - idx\_len$) handelt, dessen Referenzzähler nach dem Kopieren der Werte in das Array *res* um 1 erniedrigt wird, was ggf. zur Freigabe des Speichers von *val* führt.

C.5 ICM-Befehle für Arrays ohne Referenzzähler

DeclUnqArray(*name*, τ, s_1, ..., s_n) erzeugt die Variablendeklarationen, die benötigt werden, um in C eine SAC-Array-Variable *name* mit dem Typ $\tau[s_1, ..., s_n]$ ohne Referenzzähler darzustellen.

MakeUnqArray(*res*, *src*) erzeugt C-Befehle, die aus dem SAC-Array mit Referenzzählung *src* ein SAC-Array ohne Referenzzählung machen und dieses dann der Variablen *res* zuweisen. Falls der Referenzzähler von *src* 1 ist, kann einfach der Zeiger auf das Array zugewiesen und der Referenzzähler von *src* eliminiert werden. Ansonsten muß Speicher für ein neues Array ohne Referenzzähler alloziert und die einzelnen Array-Elemente kopiert werden.

FreeUnqArray(*name*) erzeugt Befehle zum Freigeben eines Arrays ohne Referenzzähler *name*.

AllocRC(*name*, *n*) erzeugt Befehle zum Allozieren des Speichers für einen Referenzzähler, assoziiert diesen mit dem Array *name* und initialisiert ihn mit dem Wert *n*.

Anhang D

Die Compilations-Regeln

D.1 Compilation von Programmen

$$\mathrm{C}[\![\, Tydef_1 \ldots Tydef_m \;\; Objdef_1 \ldots Objdef_r \;\; Fundef_1 \ldots Fundef_n \;\; Main \,]\!]$$
$$\mapsto \left\{ \begin{array}{l} \mathrm{C}[\![\, Tydef_1 \,]\!] \ldots \mathrm{C}[\![\, Tydef_m \,]\!] \\ \mathrm{C}[\![\, Objdef_1 \,]\!] \ldots \mathrm{C}[\![\, Objdef_r \,]\!] \\ \mathrm{C}[\![\, Fundef_1 \,]\!] \ldots \mathrm{C}[\![\, Fundef_n \,]\!] \;\; \mathrm{C}[\![\, Main' \,]\!] \end{array} \right. ,$$

wobei $Main'$ aus $Main$ durch Einfügen von Zuweisungen der Form $id_i = e_i$; am Beginn des Rumpfes von $Main$ für alle $Objdef_i \equiv$ OBJDEF $\tau_i \; id_i = e_i$; entsteht

$$\mathrm{C}[\![\, \textsc{objdef} \;\; \tau \;\; Id \;\texttt{=}\; e\texttt{;} \,]\!]$$
$$\mapsto \left\{ \begin{array}{ll} \tau \;\; Id\texttt{;} & \text{falls } Basetype(\tau) \in \mathcal{T}_{Simple} \\ \textsc{DeclUnqArray}(v\texttt{,}\sigma\texttt{,}s_1\texttt{,}\ldots\texttt{,}s_n)\texttt{;} & \text{falls } Basetype(\tau) = \sigma[s_1, ..., s_n] \end{array} \right.$$

$$\mathrm{C}[\![\, \textsc{typedef} \;\; \tau \;\; TypeId\texttt{;} \,]\!]$$
$$\mapsto \left\{ \begin{array}{ll} \textsc{typedef} \;\; \tau \;\; TypeId\texttt{;} & \text{falls } Basetype(\tau) \in \mathcal{T}_{Simple} \\ \texttt{/* deleted typedef */} & \text{sonst} \end{array} \right.$$

D.2 Compilation von Funktionen

$$
\mathrm{C}\left[\!\!\left[\begin{array}{l} Tr_1,\ \ldots,\ Tr_n\ FunId(\ Ta_1\ c_1\ a_1,\ \ldots,\ Ta_m\ c_m\ a_m\ ) \\ \{\ Vardec_1,\ \ldots,\ Vardec_k \\ \quad Body; \\ \quad \textsc{return}(\ r_1,\ \ldots,\ r_n\ ); \\ \} \end{array}\right]\!\!\right]
$$

$$
\mapsto \left\{\begin{array}{l} \textsc{FunDec}(\ FunId,\ intag_1,\ Ta_1,\ a_1,\ \ldots,\ intag_m,\ Ta_m,\ a_m, \\ \qquad\qquad outtag_1,\ Tr_1,\ r_1,\ \ldots,\ outtag_n,\ Tr_n,\ r_n\ ) \\ \{\ \ \mathrm{C}[\![Vardec_1]\!],\ \ldots,\ \mathrm{C}[\![Vardec_k]\!] \\ \quad AdjustRC(\ b_i,\ Refs\ (b_i,\ Body)\texttt{-1}); \\ \quad \mathrm{CR}[\![Body;\ \textsc{return}(\ r_1,\ \ldots,\ r_n\ );\ ,\ \varepsilon]\!] \\ \} \end{array}\right. ,
$$

$$
\text{wobei } intag_i = \begin{cases} \textsc{inout} & \text{falls } c_i = \& \\ \textsc{in} & \text{falls } c_i \neq\ \& \ \wedge\ Basetype(Ta_i) \in \mathcal{T}_{Simple} \\ \textsc{in\_rc} & \text{falls } c_i \neq\ \& \ \wedge\ Basetype(Ta_i) \in \mathcal{T}_{Array} \end{cases} ,
$$

$$
outtag_i = \begin{cases} \textsc{out} & \text{falls } Basetype(Tr_i) \in \mathcal{T}_{Simple} \\ \textsc{out\_rc} & \text{sonst} \end{cases}
$$

$$
\text{und } b_i \in \{a_i \mid Basetype(Ta_i) \in\ \mathcal{T}_{Array}\}
$$

$$
\mathrm{C}[\![\tau\ \ v;]\!]
$$

$$
\mapsto \left\{\begin{array}{ll} \tau\ \ v; & \text{falls } Basetype(\tau) \in \mathcal{T}_{Simple} \\ \textsc{DeclArray}(v,\sigma,s_1,\ldots,s_n); & \text{falls } \begin{array}{l} Basetype(\tau) = \sigma[s_1,\ldots,s_n] \\ \wedge\ \neg\, UNQ(\tau) \end{array} \\ \textsc{DeclUnqArray}(v,\sigma,s_1,\ldots,s_n); & \text{falls } \begin{array}{l} Basetype(\tau) = \sigma[s_1,\ldots,s_n] \\ \wedge\ UNQ(\tau) \end{array} \end{array}\right.
$$

$$
\mathrm{CR}[\![\textsc{return}(\ r_1,\ \ldots,\ r_n\ );\ ,\ \mathcal{F}]\!] \mapsto \textsc{FunRet}(\ tag_1, r_1, \ldots, tag_n, r_n\ ); \quad ,
$$

$$
\text{wobei } tag_i = \begin{cases} \textsc{out} & \text{falls } Basetype(TYPE(r_i)) \in \mathcal{T}_{Simple} \\ \textsc{out\_rc} & \text{sonst} \end{cases}
$$

D.3 Compilation von Zuweisungen

$$\mathrm{CR}[\![v \texttt{ = } w\texttt{;}\ \mathit{Rest}\ ,\ \mathcal{F}]\!] \mapsto \begin{cases} \textsc{AssignArray}\texttt{(}\ v\texttt{,}\ w\texttt{);} \\ \mathit{AdjustRC}\texttt{(}v\texttt{,}\ \mathit{Refs}\ \texttt{(}v\texttt{,}\ \mathit{Rest;}\ \mathcal{F}\texttt{) - 1);} & \text{falls } \mathit{Basetype}(\mathit{TYPE}(\mathrm{v})) \in \mathcal{T}_{\mathit{Array}} \\ \mathrm{CR}[\![\mathit{Rest}\ ,\ \mathcal{F}]\!] \\ v = w;\ \mathrm{CR}[\![\mathit{Rest}\ ,\ \mathcal{F}]\!] & \text{sonst} \end{cases}$$

$$\mathrm{CR}[\![v \texttt{ = [} d_1\texttt{,}\ \ldots\texttt{,}\ d_n\texttt{];}\ \mathit{Rest}\ ,\ \mathcal{F}]\!] \mapsto \begin{cases} \textsc{AllocArray}\texttt{(}\ \tau\texttt{,}\ v\texttt{,}\ \mathit{Refs}\ \texttt{(}v\texttt{,}\ \mathit{Rest;}\ \mathcal{F}\texttt{));} \\ \textsc{CreateConstArrayS}\texttt{(}\ v\texttt{,}\ d_1\texttt{,}\ \ldots\texttt{,}\ d_n\texttt{);} \\ \mathrm{CR}[\![\mathit{Rest}\ ,\ \mathcal{F}]\!] \\ \qquad\qquad \text{falls } \tau = \mathit{Basetype}(\mathit{TYPE}(d_i)) \ \wedge\ \tau \in \mathcal{T}_{\mathit{Simple}} \\ \textsc{AllocArray}\texttt{(}\ \tau\texttt{,}\ v\texttt{,}\ \mathit{Refs}\ \texttt{(}v\texttt{,}\ \mathit{Rest;}\ \mathcal{F}\texttt{));} \\ \textsc{CreateConstArrayA}\texttt{(}\ v\texttt{,}\ d_1\texttt{,}\ \ldots\texttt{,}\ d_n\texttt{);} \\ \mathrm{CR}[\![\mathit{Rest}\ ,\ \mathcal{F}]\!] \\ \qquad\qquad \text{falls } \tau[s_1, ..., s_m] = \mathit{Basetype}(\mathit{TYPE}(d_i)) \end{cases}$$

$$\mathrm{CR}[\![v_1\texttt{,}\ \ldots\texttt{,}\ v_n \texttt{ = } \mathit{FunId}\texttt{(}\ e_1\texttt{,}\ \ldots\texttt{,}\ e_m\ \texttt{);}\ \mathit{Rest}\ ,\ \mathcal{F}]\!] \mapsto \begin{cases} \textsc{FunAp}\texttt{(}\ \mathit{FunId}\texttt{,}\ \mathit{intag}_1\texttt{,}\ e_1\texttt{,}\ \ldots\texttt{,}\ \mathit{intag}_m\texttt{,}\ e_m\texttt{,} \\ \qquad\qquad \mathit{outtag}_1\texttt{,}\ v_1\texttt{,}\ \ldots\texttt{,}\ \mathit{outtag}_n\texttt{,}\ v_n\ \texttt{);} \\ \mathit{AdjustRC}\texttt{(}\ v_i\texttt{,}\ \mathit{Refs}\ \texttt{(}v_i\texttt{,}\ \mathit{Rest;}\ \mathcal{F}\texttt{)-1);} \\ \mathrm{CR}[\![\mathit{Rest}\ ,\ \mathcal{F}]\!] \end{cases},$$

$$\text{wobei } \mathit{intag}_i = \begin{cases} \textsc{inout} & \text{falls } e_i \text{ CALL-BY-REFERENCE-Parameter} \\ \textsc{in} & \text{falls } \mathit{Basetype}(\mathit{TYPE}(e_i)) \in \mathcal{T}_{\mathit{Simple}} \\ \textsc{in\_rc} & \text{sonst} \end{cases}$$

$$\text{und } \mathit{outtag}_i = \begin{cases} \textsc{out} & \text{falls } \mathit{Basetype}(\mathit{TYPE}(v_i)) \in \mathcal{T}_{\mathit{Simple}} \\ \textsc{out\_rc} & \text{sonst} \end{cases}$$

$$\mathrm{CR}[\![v_1\,,\ \ldots,\ v_n\ \texttt{=}\ Prf(\ a_1\,,\ \ldots,\ a_m\ );\ Rest\,,\ \mathcal{F}]\!] \mapsto \left\{\begin{array}{l} \mathrm{C}[\![v_1\,,\ \ldots,\ v_n\ \texttt{=}\ Prf(\ a_1\,,\ \ldots,\ a_m\ );]\!] \\ AdjustRC(\ w_i\,,\ Refs\ (w_i\,,\ Rest;\ \mathcal{F})\texttt{-1});\\ \textsc{DecRCFreeArray}(\ b_i\,,\ \texttt{1});\\ \mathrm{CR}[\![Rest\ ,\ \mathcal{F}]\!] \end{array}\right. ,$$

wobei $w_i \in \{v_i \mid Basetype(TYPE(v_i)) \in \mathcal{T}_{Array}\}$
und $b_i \in \{a_i \mid Basetype(TYPE(a_i)) \in \mathcal{T}_{Array}\}$

D.4 IF-THEN-ELSE-Konstrukte

$$\mathrm{CR}\left[\!\!\left[\begin{array}{l}\textsc{if}\ \texttt{(e)}\ \{\\ \quad Ass_t;\\ \}\\ \textsc{else}\ \{\\ \quad Ass_e;\\ \};\\ Rest\end{array}\right]\!\!\right], \mathcal{F}] \mapsto \left\{\begin{array}{l}\textsc{if}\ \texttt{(e)}\ \{\\ \quad \textsc{DecRCFreeArray}(\ \mathtt{a}_i\,,\ \mathtt{m}_i)^1;\\ \quad \mathrm{CR}[\![Ass_t;\ ,\ Rest\,\mathcal{F}]\!]\\ \}\\ \textsc{else}\ \{\\ \quad \textsc{DecRCFreeArray}(\ \mathtt{b}_i\,,\ \mathtt{n}_i)^2;\\ \quad \mathrm{CR}[\![Ass_e;\ ,\ Rest\,\mathcal{F}]\!]\\ \}\\ \mathrm{CR}[\![Rest\ ,\ \mathcal{F}]\!]\end{array}\right. ,$$

wobei $\mathrm{a}_i \in \{\mathrm{v} : Basetype(TYPE(\mathrm{v})) \in \mathcal{T}_{Array}$
$\wedge\ Refs\ (\mathrm{v}, Ass_e; Rest; \mathcal{F}) > Refs\ (\mathrm{v}, Ass_t; Rest; \mathcal{F})\}$
und $\mathrm{m}_i = Refs\ (\mathrm{a}_i, Ass_e; Rest; \mathcal{F}) - Refs\ (\mathrm{a}_i, Ass_t; Rest; \mathcal{F})$,
sowie $\mathrm{b}_i \in \{\mathrm{v} : Basetype(TYPE(\mathrm{v})) \in \mathcal{T}_{Array}$
$\wedge\ Refs\ (\mathrm{v}, Ass_t; Rest; \mathcal{F}) > Refs\ (\mathrm{v}, Ass_e; Rest; \mathcal{F})\}$
und $\mathrm{n}_i = Refs\ (\mathrm{b}_i, Ass_t; Rest; \mathcal{F}) - Refs\ (\mathrm{b}_i, Ass_e; Rest; \mathcal{F})$

[2] FREE_ARRAY(a_i) falls $UNQ(TYPE(\mathrm{a}_i))$.
[2] FREE_ARRAY(b_i) falls $UNQ(TYPE(\mathrm{b}_i))$.

D.5 Schleifenkonstrukte

$$\mathrm{CR}\left[\!\!\left[\begin{array}{l}\texttt{DO \{}\\ \quad \mathit{Ass};\\ \texttt{\} WHILE (e);}\\ \mathit{Rest}\end{array}\;,\;\mathcal{F}\right]\!\!\right] \mapsto \left\{\begin{array}{l}\texttt{GOTO label}_k\texttt{;}\\ \texttt{DO \{}\\ \quad \textsc{DecRCFreeArray}\texttt{(v}_i\texttt{, 1);}\\ \texttt{LABEL}_k\texttt{:}\\ \quad \mathit{AdjustRC}\texttt{(x}_i\texttt{, }\mathit{Refs}\texttt{ (x}_i\texttt{, }\mathit{Ass};\mathit{Body\_ext}\texttt{)-1);}\\ \quad \mathrm{CR}\left[\!\!\left[\mathit{Ass;}\;,\;\mathit{Body\_ext}\right]\!\!\right]\\ \texttt{\} WHILE (e);}\\ \textsc{DecRCFreeArray}\texttt{(w}_i\texttt{, 1);}\\ \mathit{AdjustRC}\texttt{(y}_i\texttt{, }\mathit{Refs}\texttt{ (y}_i\texttt{, }\mathit{Rest}\ \mathcal{F}\texttt{)-1);}\\ \mathrm{CR}\left[\!\!\left[\mathit{Rest}\;,\;\mathcal{F}\right]\!\!\right]\texttt{;}\end{array}\right.$$

mit
$\mathrm{x}_i \in \{\mathrm{x}_1, ...,\mathrm{x}_n\} = \mathrm{V}_{arg} = \mathrm{V}_{need}(Ass) \cup (\mathrm{V}_{pot\_def}(Ass) \cap \mathrm{V}_{need}(Rest))$,
$\mathrm{y}_i \in \{\mathrm{y}_1, ...,\mathrm{y}_m\} = \mathrm{V}_{res} = (\mathrm{V}_{def}(Ass) \cap \mathrm{V}_{need}(Rest))$,
$\mathrm{v}_i \in \mathrm{V}_{res} \setminus \mathrm{V}_{arg}$,
$\mathrm{w}_i \in \mathrm{V}_{arg} \setminus \mathrm{V}_{res}$
und $\mathit{Body\_ext} = \left\{\begin{array}{l}\texttt{IF (e)}\\ \quad \texttt{y}_1\texttt{, ..., y}_m\texttt{ = dummy(x}_1\texttt{, ..., x}_n\texttt{);}\\ \texttt{RETURN(y}_1\texttt{, ..., y}_m\texttt{);}\end{array}\right.$

$$\mathrm{CR}\left[\!\!\left[\begin{array}{l}\texttt{WHILE (}e\texttt{)\{}\\ \quad \mathit{Ass};\\ \texttt{\};}\\ \mathit{Rest}\end{array}\;,\;\mathcal{F}\right]\!\!\right] \mapsto \left\{\begin{array}{l}\texttt{WHILE (}e\texttt{) \{}\\ \quad \mathit{AdjustRC}\texttt{(}x_i\texttt{, }\mathit{Refs}\texttt{ (}x_i\texttt{, }\mathit{Ass};\mathit{Body\_ext}\texttt{)-1);}\\ \quad \mathrm{CR}\left[\!\!\left[\mathit{Ass;}\;,\;\mathit{Body\_ext}\right]\!\!\right]\\ \texttt{\};}\\ \textsc{DecRCFreeArray}\texttt{(}w_i\texttt{, 1);}\\ \mathit{AdjustRC}\texttt{(}y_i\texttt{, }\mathit{Refs}\texttt{ (}y_i\texttt{, }\mathit{Rest}\ \mathcal{F}\texttt{)-1);}\\ \mathrm{CR}\left[\!\!\left[\mathit{Rest}\;,\;\mathcal{F}\right]\!\!\right]\texttt{;}\end{array}\right.$$

mit
$x_i \in \{x_1, ..., x_n\} = \mathrm{V}_{arg} = \mathrm{V}_{need}(Ass) \cup (\mathrm{V}_{def}(Ass) \cap \mathrm{V}_{need}(Rest))$,
$y_i \in \{y_1, ..., y_m\} = \mathrm{V}_{res} = (\mathrm{V}_{def}(Ass) \cap \mathrm{V}_{need}(Rest))$,
$w_i \in \mathrm{V}_{arg} \setminus \mathrm{V}_{res}$
und $\mathit{Body\_ext} = \left\{\begin{array}{l}y_1\texttt{, ..., }y_m\texttt{ = }\mathit{dummy}\texttt{(}x_1\texttt{, ..., }x_n\texttt{);}\\ \texttt{RETURN(}y_1\texttt{, ..., }y_m\texttt{);}\end{array}\right.$

$$\mathrm{CR}\left[\!\!\left[\begin{array}{l}\texttt{FOR (}Ass_1\texttt{; e; }Ass_2\texttt{)\{}\\ \quad Ass;\\ \texttt{\};}\\ Rest\end{array}\;,\;\mathcal{F}\right]\!\!\right] \mapsto \mathrm{CR}\left[\!\!\left[\begin{array}{l}Ass_1\texttt{;}\\ \texttt{WHILE (e)\{}\\ \quad Ass;\\ \quad Ass_2\texttt{;}\\ \texttt{\};}\\ Rest\end{array}\;,\;\mathcal{F}\right]\!\!\right]$$

D.6 Compilation von Array-Operationen

$$\mathrm{C}[\![\texttt{v = DIM(array);}]\!] \mapsto \left\{\begin{array}{ll}\texttt{v = n;} & \text{falls } \mathit{Basetype}(\mathit{TYPE}(\text{array})) = \tau[s_1, ..., s_n]\end{array}\right.$$

$$\mathrm{C}[\![\texttt{v = SHAPE(array);}]\!] \mapsto \left\{\begin{array}{ll}\textsc{AllocArray}\texttt{(}\tau\texttt{, v, 1);} & \\ \textsc{CreateConstArrayS}\texttt{(v, }s_1\texttt{, ..., }s_n\texttt{);} & \text{falls } \begin{array}{l}\mathit{Basetype}(\mathit{TYPE}(\text{array}))\\ = \tau[s_1, ..., s_n]\end{array}\end{array}\right.$$

$$\mathrm{C}[\![v\texttt{ = PSI(}idx\texttt{, }array\texttt{);}]\!] \mapsto \left\{\begin{array}{ll}\textsc{PsiVxA\_S}(array, v, n, idx); & \text{falls } \left\{\begin{array}{l}n = \textsc{dim}(array)\\ \wedge\ [n] = \textsc{shape}(idx)\end{array}\right.\\ \textsc{PsiVxA\_A}(m, array, v, n, idx); & \text{falls } \left\{\begin{array}{l}m = \textsc{dim}(\text{array})\\ \wedge\ [n] = \textsc{shape}(idx)\\ \wedge\ m \neq n\end{array}\right.\end{array}\right.$$

$$\mathrm{C}[\![\texttt{v = RESHAPE(shape, array);}]\!] \mapsto \left\{\begin{array}{l}\textsc{AssignArray}\texttt{(v, array);}\\ \textsc{IncRC}\texttt{(v, 1);}\end{array}\right.$$

D.7 WITH-Konstrukte

$$
\mathrm{CR}\left[\!\!\left[\begin{array}{l} res \texttt{ = } \textsc{with}\texttt{(}\ e_1 \texttt{ <= } idx \texttt{ <= } e_2\ \texttt{)} \\ \qquad \texttt{\{}\ Assigns;\ \texttt{\}} \\ \quad \textsc{modarray}\texttt{(}\ idx\texttt{,}\ val\texttt{,}\ array\texttt{);} \\ Rest; \end{array} \;,\; \mathcal{F} \right]\!\!\right]
$$

$$
\mapsto \left\{ \begin{array}{l}
\mathrm{CR}\left[\!\!\left[\begin{array}{ll} v_1 \texttt{=} e_1\texttt{;} & array_2 \texttt{ = } dummy\texttt{(}\ w_1\texttt{,}\ \ldots\texttt{,}\ w_n\texttt{);} \\ v_2 \texttt{=} e_2\texttt{;} \quad , & Rest; \\ & \mathcal{F} \end{array}\right]\!\!\right] \\
\textsc{AllocArray}\texttt{(}\ idx\texttt{,}\ 1\texttt{);} \\
\textsc{BeginModarray}\texttt{(}\ res\texttt{,}\ dim\_res\texttt{,}\ array\texttt{,}\ v_1\texttt{,}\ v_2\texttt{,}\ idx\texttt{,}\ len\_idx\ \texttt{);} \\
\quad \textsc{IncRC}\texttt{(}\ idx\texttt{,}\ Refs\,(idx,\ Assigns;\ \textsc{return}(val);));} \\
\quad \textsc{IncRC}\texttt{(}\ w_i\texttt{,}\ Refs\,(w_i,\ Assigns;\ \textsc{return}(val);));} \\
\quad \mathrm{CR}\left[\!\!\left[Assigns;\ ,\ \textsc{return}(val);\right]\!\!\right] \\
\textsc{EndModarrayS}\texttt{(}\ res\texttt{,}\ dim\_res\texttt{,}\ array\texttt{,}\ val\ \texttt{)}^3\texttt{;} \\
\textsc{DecRCFreeArray}\texttt{(}\ idx\texttt{,}\ 1\texttt{);} \\
\textsc{DecRCFreeArray}\texttt{(}\ w_i\texttt{,}\ 1\texttt{);} \\
\mathrm{CR}\left[\!\!\left[Rest\ ,\ \mathcal{F}\right]\!\!\right]
\end{array} \right. ,
$$

wobei $\{w_1, ..., w_n\} = \mathrm{V}_{need}(Assigns; \textsc{return}(val);) \cup \{v_1, v_2, a\} \setminus \{idx\}$

[3] ENDMODARRAYA(*res*, *dim_res*, *array*, *val*, *len_idx*) falls *val* ein Array ist.

D.8 Objekt-Konvertierung

$$\mathrm{CR}[\![v \texttt{ = } \textsc{to}\_class(w)\texttt{;}\ Rest\ ,\ \mathcal{F}]\!]$$
$$\mapsto \begin{cases} \textsc{MakeUnqArray}(\ w\texttt{,}\ v\texttt{);} \\ \mathrm{CR}[\![Rest\ ,\ \mathcal{F}]\!] & \text{falls } Basetype(TYPE(w)) \in \mathcal{T}_{Array} \\ \\ v \texttt{ = } w\texttt{;} \\ \mathrm{CR}[\![Rest\ ,\ \mathcal{F}]\!] & \text{sonst} \end{cases}$$

$$\mathrm{CR}[\![v \texttt{ = } \textsc{from}\_class(w)\texttt{;}\ Rest\ ,\ \mathcal{F}]\!]$$
$$\mapsto \begin{cases} v \texttt{ = } w\texttt{;} \\ \textsc{AllocRC}(\ v\texttt{,}\ Refs\ (v\texttt{,}\ Rest\texttt{;}\ \mathcal{F}))\texttt{;} & \text{falls } \begin{array}{c} Basetype(TYPE(w)) \\ \in \mathcal{T}_{Array} \end{array} \\ \mathrm{CR}[\![Rest\ ,\ \mathcal{F}]\!] \\ \\ v \texttt{ = } w\texttt{;} \\ \mathrm{CR}[\![Rest\ ,\ \mathcal{F}]\!] & \text{sonst} \end{cases}$$

Literaturverzeichnis

[Ach96] P. Achten: *Interactive Functional Programs: Models, Methods, and Implementation.* PhD thesis, University of Nijmegen, 1996.

[AGP78] Arvind, K.P. Gostelow, and W. Plouffe: *The ID-Report: An asynchronous Programming Language and Computing Machine.* Technical Report 114, University of California at Irvine, 1978.

[AP93] P. Achten and R. Plasmeijer: *The Beauty and the Beast.* Technical Report 93-03, University of Nijmegen, 1993.

[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.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman: *Compilers - Principles, Techniquies, and Tools.* Addison-Wesley, 1986. ISBN 0-201-10194-7.

[Aug87] L. Augustsson: *Compiling Lazy Functional Languages Part II.* PhD thesis, Chalmers University of Technologie, Göteborg, 1987.

[AWWV96] J. Armstrong, M. Williams, C. Wikstrom, and R. Virding: *Concurrent Programming in Erlang.* Prentice Hall, 1996. ISBN 0-13-285792-8.

[Bac78] J. Backus: *Can Programming be Liberated from the von Neumann Style?* Communications of the ACM, Vol. 21(8), 1978, pp. 613–641.

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

[BBB+94] D. Bailey, E. Barszcz, J. Barton, et al.: *The NAS Parallel Benchmarks.* RNR 94-007, NASA Ames Research Center, 1994.

[BCOF91] A.P.W. Böhm, D.C. Cann, R.R. Oldehoeft, and J.T. Feo: *SISAL Reference Manual Language Version 2.0.* CS 91-118, Colorado State University, Fort Collins, Colorado, 1991.

[BE76] K.J. Bathe and E.L.Wilson: *Numerical Methods in Finite Element Analysis*. Prentice-Hall, 1976.

[Bec87] A. Beckmann: *Die Modellierung mesoskaliger quasigeostrophischer Instabilität*. Technical Report 167, Institut für Meereskunde, Universität Kiel, 1987.

[Ber75] K. Berkling: *Reduction Languages for Reduction Machines*. In Proceedings of the 2nd Annual International Symposium on Computer Architecture. ACM/IEEE, 1975, pp. 133–140.

[Ber76] K. Berkling: *A Symmetric Complement to the Lambda Calculus*. ISF 76-7, GMD, Bonn, 1976.

[BGS94] D.F. Bacon, S.L. Graham, and O.J. Sharp: *Compiler Transformations for High-Performance Computing*. ACM Computing Surveys, Vol. 26(4), 1994, pp. 345–420.

[Ble94] G.E. Blelloch: *NESL: A Nested Data-Parallel Language (Version 3.0)*. Carnegie Mellon University, 1994.

[Blo91] S. Blott: *Type Classes*. PhD thesis, Glasgow University, 1991.

[BM87] E.P. Beisel and M. Mendel: *Optimierungsmethoden des Operations Research*. Vieweg, 1987.

[Bra84] A. Brandt: *Multigrid Methods: 1984 Guide*. Dept of applied mathematics, The Weizmann Institute of Science, Rehovot/Israel, 1984.

[BS95] E. Barendsen and S. Smetsers: *Uniqueness Type Inference*. In M. Hermenegildo and S.D. Swierstra (Eds.): PLILP'95, LNCS, Vol. 982. Springer, 1995, pp. 189–206.

[Bur75] W.H. Burge: *Recursive Programming Techniques*. Addison-Wesley, 1975.

[BvEG+90] H.P. Barendregt, M. van Eekelen, J. Glauert, et al.: *Term Graph Reduction*. In PARLE '90, Eindhoven, LNCS, Vol. 259. Springer, 1990, pp. 141–158.

[BW85] M. Barr and C. Wells: *Toposes, Triplets, and Theories*. Springer Verlag, 1985.

[BW88] R.S. Bird and P.L. Wadler: *Functional Programming*. Prentice Hall, 1988.

[BW91] R.L. Bowers and J.R. Wilson: *Numerical Modeling in Applied Physics and Astrophysics.* Jones and Bartlett Publishers, 1991. ISBN 0-86720-123-1.

[Can89] D.C. Cann: *Compilation Techniques for High Performance Applicative Computation.* Technical Report CS-89-108, Lawrence Livermore National Laboratory, LLNL, Livermore California, 1989.

[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 California, 1993. part of the SISAL distribution.

[CE95] D.C. Cann and P. Evripidou: *Advanced Array Optimizations for High Performance Functional Languages.* IEEE Transactions on Parallel and Distributed Systems, Vol. 6(3), 1995, pp. 229–239.

[CF58] H.B. Curry and R. Feys: *Combinatory Logic Vol. I.* Studies in Logics and the Foundations of Mathmatics. North-Holland, 1958.

[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.

[CJ85] C. Clack and S. Peyton Jones: *Strictness Analysis - a Practical Approach.* In FPCA '85, Nancy, LNCS, Vol. 201. Springer, 1985.

[Coh81] J. Cohen: *Garbage Collection of Linked Data Structures.* ACM Computing Surveys, Vol. 13(3), 1981, pp. 341–367.

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

[dHRvE95] W. de Hoon, L. Rutten, and M. van Eekelen: *Implementing a Functional Spreadsheat in Clean.* Journal of Functional Programming, Vol. 5(3), 1995, pp. 383–414.

[FH88] A.J. Field and P.G. Harrison: *Functional Programming.* International Computer Science Series. Addison-Wesley, 1988. ISBN 0-201-19249-7.

[FJ96] S. Finne and S. Peyton Jones: *Composing Haggis.* In Proceeding of the fifth Eurographics Workshop on Programming Paradigms for Computer Graphics, Sept.95. Springer, 1996. (To be published).

[FO95] S.M. Fitzgerald and R.R. Oldehoeft: *Update-in-place Analysis for True Multidimensional Arrays.* In A.P.W. Böhm and J.T. Feo (Eds.): High Performance Functional Computing, 1995, pp. 105–118.

[FW87] J. Fairbairn and S.C. Wray: *TIM: A Simple Abstract Machine to Execute Supercombinators.* In G. Kahn (Ed.): FPCA'87, Portland, Oregon, LNCS, Vol. 274. Springer, 1987.

[GHT84] H. Glaser, C. Hankin, and D. Till: *Principles of Functional Programming.* Prentice Hall, 1984.

[Gil96] A. Gill: *Cheap Deforestation for Non-strict Functional Languages.* PhD thesis, Glasgow University, 1996.

[GK76] A. Goldberg and A. Kay: *Smalltalk-80, Instruction Manual.* Xerox, Palo Alto Research Center (PARC), Palo Alto, CA, 1976.

[GLJ93] A. Gill, J. Launchbury, and S.L. Peyton Jones: *A Short Cut to Deforestation.* In FPCA '93, Copenhagen. ACM Press, 1993, pp. 223–232.

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

[Gre96] C. Grelck: *Integration eines Modul- und Klassen-Konzeptes in die funktionale Programmiersprache SAC – Single Assignment C.* Master's thesis, Institut für Informatik und praktische Mathematik, Universität Kiel, 1996.

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

[GZO+78] R.H. Gallagher, O.C. Zienkiewicz, J.T. Oden, et al.: *Finite Elements in Fluids.* Wiley Series in Numerical Methods in Engineering, Vol. 3. Wiley, 1978.

[HAB+95] K. Hammond, L. Augustsson, B. Boutel, et al.: *Report on the Programming Language Haskell: A Non-strict, Purely Functional Language.* University of Glasgow, 1995. Version 1.3.

[Hac85] W. Hackbusch: *Multi-grid Methods and Applications.* Springer, 1985.

[Hen80] P. Henderson: *Functional Programming.* Prentice Hall International, 1980. ISBN 0-13-331579-7.

[HH89] G. Hämmerlein and K.H. Hoffmann: *Numerische Mathematik*. Grundwissen Mathematik, Vol. 7. Springer, 1989.

[HHJW92] C. Hall, K. Hammond, S. Peyton Jones, and P. Wadler: *Type Classes in Haskell*. Technical report, University of Glasgow, 1992.

[HJW+92] P. Hudak, S. Peyton Jones, P. Wadler, et al.: *Report on the Programming Language Haskell*. Yale University, 1992. Version 1.2.

[HS86] J.R. Hindley and J.P. Seldin: *Introduction to Combinators and Lambda Calculus*. London Mathematical Society Student Texts, Vol. 1. Cambridge University Press, 1986.

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

[HT82] W. Hackbusch and U. Trottenberg: *Multigrid Methods*. LNM, Vol. 960. Springer, 1982.

[Hud89] P. Hudak: *Conception, Evolution, and Application of Functional Programming Languages*. ACM Computing Surveys, Vol. 21(3), 1989, pp. 359–411.

[Hud92] P. Hudak: *Mutable Abstract Datatypes – or – How to Have Your State and Munge It Too*. DCS RR-914, Yale University, 1992.

[Hug89] J. Hughes: *Why Functional Programming Matters*. The Computer Journal, Vol. 32(2), 1989, pp. 98–107.

[HY86] P. Hudak and M.F. Young: *Higher Order Strictness Analysis in Untyped Lambda Calculus*. In POPL '86, St. Petersburg, Florida. ACM, 1986.

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

[JD93] M.P. Jones and L. Duponcheel: *Composing monads*. YALEU/DCS/RR 1004, Yale University, New Haven, CT, USA, 1993.

[JG89] M.A. Jenkins and J.I. Glagow: *A Logical Basis for Nested Array Data Structures*. Computer Languages Journal, Vol. 14(1), 1989, pp. 35–51.

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

[Joh87] T. Johnsson: *Compiling Lazy Functional Languages*. PhD thesis, Chalmers University of Technologie, Göteborg, 1987.

[Jon85] O. Jones: *Introduction to the X Window System*. Prentice Hall, 1985.

[Jon87] S.L. Peyton Jones: *The Implementation of Functional Programming Languages.* Series in Computer Science. Prentice-Hall International, 1987. ISBN 0-13-453325-9.

[JS89] S. Peyton Jones and J. Salkild: *The Spineless Tagless G-Machine.* In FPCA '89, London, 1989.

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

[Ken84] J.R. Kennaway: *An Outline of Some Results of Staples on Optimal Reduction Orders in Replacement Systems.* CSA 19, University of East Anglia, Norwich, 1984.

[KKS92] F. Kaden, I. Koch, and J. Selbig: *Heuristische Analyse und Vorhersage von Proteinstrukturen.* GMD Arbeitspapiere 656, GMD, Sankt Augustin, 1992.

[Klu92] W.E. Kluge: *The Organization of Reduction, Data Flow and Control Flow Systems.* MIT Press, 1992. ISBN 0-262-61081-7.

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

[KR90] B.W. Kernighan and D.M. Ritchie: *Programmieren in C.* PC professionell. Hanser, 1990. ISBN 3-446-15497-3.

[Lan64] P.J. Landin: *The Mechanical Evaluation of Expressions.* Computer Journal, Vol. 6(4), 1964.

[Lau93] J. Launchbury: *Lazy Imperative Programming.* In ACM Sigplan Workshop on State in Programming Languages. ACM Press, 1993, pp. 46–56.

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

[LS86] J. Lambek and P. Scott: *Introduction to Higher Order Categorial Logic.* Cambridge University Press, 1986.

[Mil87] R. Milner: *A Theory of Type Polymorphism in Programming.* Journal of Computer and System Sciences, Vol. 17, 1987, pp. 348–375.

[MJ91] L.M. Restifo Mullin and M. Jenkins: *A Comparison of Array Theory and a Mathematics of Arrays.* In Arrays, Functional Languages and Parallel Systems. Kluwer Academic Publishers, 1991, pp. 237–269.

[MKM84] J.R. McGraw, D.J. Kuck, and M.Wolfe: *A Debate: Retire Fortran?* Physics Today, Vol. 37(5), 1984, pp. 66–75.

[MKS96] L. Mullin, W. Kluge, and S.-B. Scholz: *On Programming Scientific Applications in Sac.* In W. Kluge (Ed.): Proceedings of the 8th International Workshop on Implementation of Functional Languages. Christian-Albrechts-Universität, Kiel, 1996, pp. 321–339.

[MSA+85] J.R. McGraw, S.K. Skedzielewski, S.J. Allan, R.R. Oldehoeft, et al.: SISAL*: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2.* M 146, Lawrence Livermore National Laboratory, LLNL, Livermore California, 1985.

[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.

[MTH90] R. Milner, M. Tofte, and R. Harper: *The Definition of Standard ML.* MIT Press, 1990. ISBN 0-262-63132-6.

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

[Mul91] L.M. Restifo Mullin: *The Ψ-Function: A Basis for FFP with Arrays.* In L.M. Restifo Mullin (Ed.): Arrays, Functional Languages and Parallel Systems. Kluwer Academic Publishers, 1991, pp. 185–201.

[Myc84] A. Mycroft: *Polymorphic Type Schemes and Recursive Definitions.* In Symposium on Programming, LNCS, Vol. 167. Springer, 1984, pp. 217–239.

[Nob95] R. Noble: *Lazy Functional Components for Graphical User Interfaces.* PhD thesis, University of York, 1995.

[Nöc93] E. Nöcker: *Strictness Analysis Using Abstract Reduction.* In FPCA '93, Copenhagen. ACM Press, 1993, pp. 255–265.

[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.

[Old92] R.R. Oldehoeft: *Implementing Arrays in SISAL 2.0.* In Proceedings of the Second SISAL Users' Conference, 1992, pp. 209–222.

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

[Plo74] G. Plotkin: *Call by Name, Call by Value, and the Lambda Calculus.* Theoretical Computer Science, Vol. 1, 1974.

[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 Eckelen: *Concurrent Clean 1.0 Language Report.* University of Nijmegen, 1995.

[Rea89] C. Reade: *Elements of Functional Programming.* International Computer Science Series. Addison-Wesley, 1989. ISBN 0-201-12915-9.

[Rei95] C. Reinke: *Functions, Frames, and Interactions.* In T. Johnsson (Ed.): Proceedings of the Workshop on the Implementation of Functional Languages'95. Chalmers University, 1995, pp. 157–172.

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

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

[SCA93] A.V.S. Sastry, W. Clinger, and Z. Ariola: *Order-of-evaluation Analysis for Destructive Updates in Strict Functional Languages with Flat Aggregates.* In FPCA '93, Copenhagen. ACM Press, 1993, pp. 266–275.

[Sch94] S.-B. Scholz: **S***ingle* **A***ssignment* **C** *– Functional Programming Using Imperative Style.* In John Glauert (Ed.): Proceedings of the 6th International Workshop on the Implementation of Functional Languages. University of East Anglia, 1994.

[Sie95] A. Sievers: *Maschinenunabhängige Optimierungen eines Compilers für die funktionale Programmiersprache Single Assignment C.* Master's thesis, Institut für Informatik und praktische Mathematik, Universität Kiel, 1995.

[SS88] S. Skedzielewski and R.J. Simpson: *A Simple Method to Remove Reference Counting in Applicative Languages.* Technical Report UCRL-100156, Lawrence Livermore National Laboratory, LLNL, Livermore California, 1988.

[Sta80] J. Staples: *Computation on Graph-Like Expressions.* Theoretical Computer Science, Vol. 10, 1980, pp. 171–185.

[Sta94] R.M. Stallman: *Using and Porting GNU CC.* Free Software Foundation, Cambridge, USA, 1994.

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

[SW85] S. Skedzielewski and M.L. Welcome: *Data Flow Graph Optimization in IF1.* In FPCA '85, Nancy, LNCS, Vol. 201. Springer, 1985, pp. 17–34.

[Tur79] D.A. Turner: *A New Implementation Technique for Applicative Languages.* Software-Practice and Experience, Vol. 9, 1979, pp. 31–49.

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

[vG96] J. van Groningen: *The Implementation and Efficiency of Arrays in Clean 1.1.* In W. Kluge (Ed.): Proceedings of the 8th International Workshop on Implementation of Functional Languages. Christian-Albrechts-Universität, Kiel, 1996, pp. 131–154.

[Wad90] P.L. Wadler: *Deforestation: transforming programs to eliminate trees.* Theoretical Computer Science, Vol. 73(2), 1990, pp. 231–248.

[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, Albequerque. ACM Press, 1992.

[War93] Z.U.A. Warsi: *Fluid Dynamics: Theoretical and Computational Approaches.* CRC Press, 1993.

[WB89] P. Wadler and S. Blott: *How to Make ad-hoc Polymorphism Less ad hoc.* In POPL '89. ACM Press, 1989, pp. 60–76.

[Wir85] N. Wirth: *Programming in Modula-2.* Texts and Monographs in Computer Science. Springer, 1985. ISBN 0-387-15078-1.

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