Entwicklung eines korrekten Assemblierers

Emil Sekerinski
Diplomarbeit

Betreuer:
Prof. Dr. rer. nat. G. Goos
Dipl. Inform. F. Weber

Forschungszentrum Informatik an der Universität Karlsruhe
März 1989

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Literaturhilfsmittel benutzt zu haben.

_____

Emil Sekerinski

Entwicklung eines korrekten Assemblierers

Zusammenfassung. In dieser Arbeit wird ein Assemblierer für den MC68020 Prozessor entwickelt und gegenüber einer formalen Spezifikation als korrekt bewiesen. Die Spezifikation ist in Prädikatenlogik gegeben, und die angewandte Beweismethode ist ein auf Prädikatenlogik aufgesetzter Kalkül. Die Implementierung ist in Modula-2 geschrieben und hat, neben ihrer Korrektheit, die Vorzüge, extrem kompakt und effizient zu sein.

Inhaltsverzeichnis

Einleitung

Ziel der vorliegenden Arbeit ist es, ein realistisch großes und komplexes Programm, einen Assemblierer, auf eine Art zu entwickeln, bei der die Korrektheit des Programmes durch mathematische Überlegungen unterstützt wird. Damit soll eine Alternative gezeigt werden zu dem üblichen Verfahren, sich alleine auf das *Testen*, d.h. die Analyse des laufenden Programmes, und die *Codeinspektion*, also die informelle Analyse des Programmtextes, zu stützen.

Daß das Testen als Durchführen von Experimenten am Objekt nicht die gewünschte Aussagekraft hat wie das analoge Vorgehen in der Physik oder im Maschinenbau liegt an der *Unstetigkeit* der beobachteten Größen. Stetigkeit besagt, daß sich bei genügend kleinen Änderungen der Eingabegrößen nur kleine Änderungen der Ausgabegrößen ergeben können ("Die Funktion hat keine Sprünge"), ein Prinzip , das bei *digitalen* Rechnern nicht zutrifft. Damit kann durch eine noch so große Anzahl von Testläufen mit noch so dicht beieinander liegenden Eingabewerten kein Schluß gezogen werden auf das Verhalten bei den Zwischenwerten. Die Essenz dieses Gedankens hat Edsger W. Dijkstra in dem wohlbekannten Satz zusammengefaßt:

> Testing can only show the presence of bugs, but never their absence.

Die Folgerung daraus ist, daß wir uns bei Korrektheitsüberlegungen auf die Analyse des Programmtextes konzentrieren müssen. Trotzdem können wir auf das Testen nicht völlig verzichten. Es ist nachwievor nötig, um sich davon zu überzeugen, daß das Programm mit der ursprünglichen, intuitiven Vorstellung übereinstimmt. Das Finden von Fehlern sollte dabei die Ausnahme sein, und nicht zu einer großangelegten *Fehlersuche* und *-reparatur* (*"debugging"*) als Teil des *Entwicklungszyklus* ausarten, wie üblich.

Die Entscheidung, einen Assemblierer für Korrektheitsüberlegungen zu verwenden wurde getroffen, weil zum einen hier als Teil der Systemsoftware besondere Sorgfalt geboten ist, denn ein Fehler in einem Assemblierer, Übersetzer oder Betriebssystem pflanzt sich automatisch auf die Anwendersoftware fort. Zum anderen können bei einem Assemblierer Erfahrungen gesammelt werden, die bei dem komplexeren Fall eines Übersetzers eingesetzt werden können. Es kann sogar ein Teil des Assemblierers direkt in einem Übersetzer wiederverwendet werden. Dazu ist notwendig, den Assemblierer aufzuspalten in einen Analyseteil, bestehend aus *Symbolentschlüssler* und *Zerteiler*, und einem Syntheseteil, dem *Generator*, mit einer prozeduralen Schnittstelle als Bindeglied. Ein Übersetzer verwendet dann den Generator über genau diese Schnittstelle.

Der Schwerpunkt der Arbeit soll deshalb auf dem Generator liegen, und dort speziell auf genau den Assemblerinstruktionen, die von einem Übersetzer erzeugt werden. Um den Umfang im Rahmen zu halten, wurde auf die Behandlung von externen Referenzen und globalen Definitionen verzichtet. Die Instruktionen für die Fließkommaarithmetik wurden ebenfalls weggelassen; sie ließen sich leicht in die bestehende Struktur einfügen und würden nur zur Länglichkeit beitragen, ohne neue Aspekte zu liefern.

Aus den örtlichen Gegebenheiten heraus entstand die Forderung, den Assemblierer für den Motorola 68020 Prozessor unter dem Unix Betriebssystem und für das *a.out* Objektdateiformat zu konzipieren und in Modula-2 zu implementieren.

In Arbeiten von Polack [1981] und Sørensen & Sufrin [1987] über ähnliche Themen definieren sich die Autoren ihren eigenen Prozessor und ihr eigenes Objektdateiformat. Damit vereinfacht sich die Problematik um einiges, die Arbeiten haben aber nur theoretischen Wert. Zudem führen sie zu Implementierungen, die keine Produktqualität besitzen: sie sind extrem ineffizient und speicherintensiv. Der Grund hierfür ist, daß die Implementierungen die Struktur der Spezifikationen widerspiegeln, diese aber auf Verständlichkeit und nicht auf effiziente Implementierbarkeit ausgelegt sind. (Bei Polack besteht die Implementierung aus einem Symbolentschlüssler, einem Zerteiler, einer semantischen Analyse und einem Codegenerator, wobei die Zwischenrepräsentationen, wie in der Spezifikation angegeben, vollständig aufgebaut werden. Die übliche Implementierungstechnik dahingegen ist es, diese Teile so ineinander zu verzahnen, daß nur dort, wo unbedingt notwendig, Datenstrukturen aufgebaut werden.)

Eines der Resultate dieser Arbeit ist, daß Spezifikationsnotationen und Beweistechniken, richtig eingesetzt, nicht nur das Vertrauen in die Korrektheit des Programmes steigern, sondern auch zu kompakteren und effizienteren Programmen führen, ohne daß die Herleitung unverständlich formal sein muß. Die Größe des Assemblierers beträgt:

Symbolentschlüssler	200 Zeilen
Zerteiler	200 Zeilen
Generator	400 Zeilen

Dies beinhaltet die Übersetzung von 158 Instruktionen und zehn Adressierungsarten. Der Durchsatz beträgt 1600 Zeilen pro Sekunde gemessen auf einer Sun 3/60, mehr als doppelt so viel wie bei dem Unix Assemblierer *as*.

Als Neuerung wurden bei der Entwicklung das Konzept der Implementierungsrestriktionen eingeführt. An manchen Stellen ist es sehr schwierig oder durch die Endlichkeit des Speichers unmöglich, eine Implementierung anzugeben, die die Spezifikation voll erfüllt.

Unser Vorgehen ist es in diesen Fällen eine Implementierung anzugeben, die nur unter der Annahme einer gewissen Bedingung, der Implementierungsrestriktion, die Spezifikation erfüllt. Damit sind Implementierungsrestriktionen eine nachträgliche Verschärfung der Spezifikation. Selbstverständlich dürfen Implementierungsrestriktionen nicht so scharf sein, daß sie das Programm unbrauchbar machen. (Für die Eingaben, die die Implementierungsrestriktion oder den Definitionsbereich der Spezifikation nicht erfüllen, darf eine Implementierung beliebiges Verhalten zeigen. Der Assemblierer bricht in solchen Fällen immer mit einer Fehlermeldung ab.)

Eine weitere Erfahrung ist, daß formale Korrektheitsüberlegungen *nicht* zu Programmen führen, die als unumstößlich korrekt bezeichnet werden dürfen (sofern so etwas überhaupt existiert). Auch die erste Version des Assemblierers hat Fehler enthalten, die erst durch das systematische Testen aufgedeckt wurden. Einige dieser Fehler waren zurückzuführen auf Fehler in der Spezifikation, die aufgrund unpräziser Dokumentation entstanden sind. Weitere Flüchtigkeitsfehler haben sich bei den Beweisen eingeschlichen, vor allem bei trivialen Schritten, die es nicht Wert schienen, detailiert behandelt zu werden. Der größte Teil der Fehler waren Tippfehler, die vom Übersetzer nicht erkannt werden konnten, wie fehlerhafte Stringkonstanten oder vergessene Anweisungen. Korrektheitsbeweise helfen grobe logische Fehler zu vermeiden und das Testen auf ein notwendiges Minimum zu reduzieren, nicht aber auf das Testen zu verzichten.

Die praktische Arbeit hat auch gezeigt wie schwierig es ist, beim Programmieren Korrektheitsbeweise zu führen: man benötigt wesentlich mehr Kreativität und Ideen, und man gerät wesentlich öfter in eine Sackgasse als ohne Beweise. Drei Gründe seien hierfür genannt. Der erste ist, daß das Programm, das man versucht als korrekt zu beweisen, inkorrekt ist (zumindest bei einigen Eingabewerten) oder das Problem suboptimal löst und man dies erst durch die formale Behandlung erkannt hat (dies ist der häufigere der beiden Fälle). Dieser Punkt war aber genau das Ziel und ist als Vorteil und nicht als Nachteil zu werten. Der zweite Grund sind die nicht genügend gut ausgebauten Beweismethoden. Sie bieten für wiederkehrende Situationen keine Standardlösungen und erfordern selbst bei den einfachsten Fällen langatmige Behandlungen (Dieses Problem wurde versucht mit dem später vorgestellen Kalkül anzugehen). Der dritte Grund ist, daß zu Beginn der Arbeit noch nicht genügend Erfahrungen vorlagen. Die Entwicklung eines ähnlichen Assemblierers würde mit dem jetztigen Erfahrungsschatz einen Bruchteil der Zeit erfordern.

Die bestehenden Methoden für Korrektheitsbeweise von Programmen werden bisweilen in *Verifikationskalküle* und *Transformationskalküle* unterteilt. Beide gehen davon aus, daß eine formale Beschreibung des Problems, die *Spezifikation*, gegeben ist. Bei der Verifikation muß der Programmierer eine Idee haben, wie ein Schritt zur Lösung der Spezifikation aussieht. Dieser wird dann mit Hilfe von *Verifikationsregeln* überprüft. Bei der Transformation muß der Programmierer eine Idee haben, welche *Transformationsregel* anzuwenden ist. Falls die Anwendbarkeitsbedingung dieser Regel erfüllt ist, liefert sie automatisch einen Schritt zur Lösung der Spezifikation.

Die Analogie zur Analysis mag hilfreich sein, bei der das Verifizieren mit dem Differenzieren und das Transformieren mit dem Integrieren vergleichen wird. Die Entwicklung eines (ausführbaren) Programmes aus einer (nicht ausführbaren) Spezifikation entspricht dann dem Überführen eines (nicht berechenbaren) Integrals in eine explizite (berechenbare) Form.

Hat man eine Vermutung für die Lösung eines Integrals (einer Spezifikation), läßt

sich diese durch differenzieren (verifizieren) überprüfen. Für jeden Operator sind dazu eine oder einige wenige Differentiationsregeln (Verifikationsregeln) gegeben, derer wir uns dabei bedienen. Die Schwierigkeit hierbei ist, zuerst eine Idee für die Lösung des Integrals (der Spezifikation) zu haben.

Sind Integrationsregeln (Transformationsregeln) gegeben, liefern diese die Lösung direkt. Es muß lediglich die Anwendbarkeitsbedingung überprüft werden. Leider ist es schwierig oder unmöglich, allgemeine Integrationsregeln (Transformationsregeln) anzugeben. Sie existieren immer nur für Spezialfälle, und demnach ist auch ihre Anzahl groß. Die Kunst beim Integrieren (Transformieren) ist es, zu sehen, welche Regel mit welcher Substitution anwendbar ist und das beste Ergebnis liefert.

Bei der Herleitung des Assemblierers soll ein Kalkül eingesetzt werden, der sowohl Verifikations- als auch Transformationstechniken zuläßt. Er wird zu Anfang vorgestellt. Ein Charakteristikum dieses Kalküls ist die Einfachheit, mit der die Semantik von Programmen modelliert wird und mit der Bewiese geführt werden. Der Preis für diese Einfachheit ist der, daß Dijkstras Art von Indeterminismus nicht modellierbar ist. Weil wir ohnehin nur sequentielle, deterministische Sprachelemente benutzen, ist dies für uns ohne Bedeutung.

Das erste Kapitel ist eine Diskussion einiger bekannter und einiger neuerer Verifikationskalküle, die zu den Grundideen eines neuen Kalküls führt. Im zweiten Kapitel werden die Grundbegriffe des Kalküls eingeführt und die Definition der Programmoperatoren, die Verifikationsregeln, gegeben. Nach Bedarf werden dann später komplexere Sätze, die Transformationsregeln, hergeleitet und eingesetzt.

Die folgenden Kapitel beschreiben die Entwicklung des Assemblierers. Die Vorgehensweise ist dabei die, daß nicht die gesamte Spezifikation auf einmal vorgestellt und dann implementiert wird, sondern eine Teilspezifikation *schrittweise verfeinert* – oder besser *erweitert* – und jeder Schritt einzeln implementiert wird. Nur so lassen sich mehrseitige Spezifikationen und Beweise vermeiden. Die Kunst bei dieser Vorgehensweise ist es zu erkennen, welche Aspekte des Problems in welcher Reihenfolge spezifiziert und implementiert werden können.

Das dritte Kapitel macht den ersten Schritt zur Spezifikation des Assemblierers durch die Zerlegung in Zerteiler und Generator. Ohne auf diese Teile einzugehen, wird die Schnittstelle zwischen ihnen beschrieben. Dabei werden die zulässigen Instruktionen mit ihren Adressierungsarten spezifiziert. Außerdem wird eine abstrakte Form der Implementierung angegeben, auf die hingearbeitet werden soll.

Das vierte Kapitel behandelt das Kernproblem des Assemblierers, das Auflösen der distanzabhängigen Sprünge. Die Argumentation ist so gehalten, daß sie sich leicht auf jeden anderen Assemblierer übertragen läßt. Im Gegensatz dazu ist die Bitcodierung der einzelnen Instuktionen ganz prozessorabhängig. Sie ist das Thema des fünften Kapitels.

Die Aufgabe des Assemblierers besteht nicht nur im Übersetzen der einzelnen Instruktionen; sie müssen zusammen mit weiteren, zu generierenden Informationen in der Objektdatei im bestimmten Format, hier im *a.out* Format, abgelegt werden. Dies wird im sechsten Kapitel gezeigt.

Die bisherigen Schritte verdanken ihre Klarheit und Kompaktheit dem Einsatz mächtiger, aber schwer implementierbarer Datenstrukturen. Diese sollen im siebten Kapitel in solche in Modula-2 verfügbaren transformiert werden.

Das letzte Kapitel ergänzt den bisher betrachteten Teil des Assemblierers, den Generator, um den Zerteiler und Symbolentschlüssler, um so zu einer vollständigen, ausführbaren Implementierung zu gelangen.

Im Anhang ist die Syntax der Assemblersprache angegeben. Ihr sind die unterstützten Assemblerinstruktionen und Adressierungsarten sowie ihre Schreibweise zu entnehmen.

Danksagung. An dieser Stelle sei all denjenigen gedankt, die zur Erstellung dieser Arbeit beigetragen haben, Prof. G. Goos dafür, daß es mir dieses Thema anvertraut und die Richtung gelegt hat, Dr. J. Uhl für seine hilfreichen Kommentare und Herrn F. Weber für seine Bereitschaft, sich in zahlreichen Diskussionen intensiv mit der Problematik zu beschäftigen und viele Anregungen zu geben.

1. Literaturstudie Verifikationskalküle

Dieses Kapitel gibt einen Überblick über einige bekannte Verifikationsmethoden, dem Hoare-Kalkül, VDM, Z, und einige neuere Methoden wie dem Prädikatenkalkül von Hehner, dem Relationenkalkül, Backs Methode zur Verfeinerung und dem "Specification Statement" von Morgan. Darauf aufbauend werden die Entwurfsziele eines neuen Kalküls vorgestellt.

Beim Hoare-Kalkül wird ein Programm S spezifiziert durch zwei Bedingungen, einer Nachbedingung Q, die die zulässigen Zustände nach Ausführung von S charakterisiert, und einer Vorbedingung P, die besagt, für welche Anfangszustände das Programm S definiert ist, zusammen geschrieben als:

$$\{P\}\ S\ \{Q\}$$

P und Q werden als prädikatenlogische Formeln geschrieben. Weiterhin ist im Hoare-Kalkül zu jedem Sprachelement eine Regel gegeben, die besagt unter welchen Voraussetzungen welche Vor- und Nachbedingung für dieses Konstrukt gelten. Als Beispiel sei die Regel für die sequentielle Komposition genannt:

$$\frac{\{P\}S_1\{Q\},\ \{Q\}S_2\{T\}}{\{P\}S_1;S_2\{R\}}$$

Sie besagt, daß, falls es ein Prädikat Q gibt, so daß $\{P\}S_1\{Q\}$ und $\{Q\}S_2\{T\}$ gilt, folgt daß auch $\{P\}S_1;S_2\{R\}$ gilt. Hoare & Wirth [1973] haben solche Regeln für alle Elemente von

Pascal angegeben.

Ziel der Programmentwicklung ist es, ein Programm *S* anzugeben, von dem mit Hilfe der Regeln bewiesen werden kann, daß für gegebene *P* und *Q* gilt *{P} S {Q}*. Der Beweis erfolgt entweder *bottom-up*, d.h. beginnend mit den elementaren Anweisungen werden die Vor- und Nachbedingungen immer komplexerer Teile hergeleitet, oder *top-down*, d.h. beginnend mit der Vor- und Nachbedingung des Programmes werden die Vor- und Nachbedingungen der Komponenten entwickelt. Der Beweis sollte aber nicht nachträglich am fertigen Programm vollzogen werden sondern parallel zum Programmentwurf gebracht werden.

Der Hoare-Kalkül beinhaltet keine Vorgehensweise bei der Entwicklung. Er dient, wie jeder Verifikationskalkül, nur zum Überprüfen der Ergebnisse.

Bedeutung hat der Hoare-Kalkül dadurch erlangt, daß er der erste Kalkül war, mit dem die Semantik einer nichttrivialen Programmiersprache komplett beschrieben wurde und die Methode effektiv zum Verifizieren kleinerer Programme eingesetzt wurde. Zudem eignet er sich zur Dokumentation von Programmen, indem die Vor- und Nachbedingungen als Kommentare mit eingefügt werden. Das explizite Formulieren der Vor- und Nachbedingungen, und insbesondere der *Schleifeninvarianten*, führt zu einem wesentlich tieferen Verständnis des Programmes, wodurch der Hoare-Kalkül einen hohen didaktischen Wert besitzt. Er schult zu einer sauberen Denkweise und führt damit auch zu besseren Programmen, wenn die Bedingungen nicht total formal formuliert werden.

Dennoch hat der Hoare-Kalkül außerhalb des akademischen Bereiches keine Bedeutung erlangt. Ein wesentlicher Grund dafür ist, daß er bei der Entwicklung nicht methodisch unterstützt sondern nur Mehrarbeit verursacht. Macht man beispielsweise einen Vergleich mit der Arithmetik, so gibt es in der Arithmetik Gesetze um die Lösung einer Gleichung herzuleiten und nicht nur Gesetze um die Korrektheit einer Lösung zu überprüfen. Hieraus wird die Forderung abgeleitet, daß es neben den reinen Verifikationsregeln in einem Kalkül auch komplexere *Transformationsregeln* geben muß, die die Lösung, zumindest in einfacheren Fällen, sofort liefern.

Ein anderer Grund ist die Technik, mit der das Ein/Ausgabeverhalten beschrieben werden muß. Betrachten wir dazu:

{true} S {z=x*y}

Am Ende der Ausführung von *S* muß *z* das Produkt von *x* und *y* sein. Für *S* läßt sich damit *x:=2; y:=3; z:=6* verwenden. Möchten wir ausdrücken, daß *z* das Produkt von *x* und *y* am Anfang von *S* sein soll, müssen wir dies mit "logischen" Variablen ausdrücken, d.h. mit Variablen, die nur in den Bedingungen und nicht im Programm auftreten. Falls x_0 und y_0 solche Variablen sind, besagt

$\{x_0=x \wedge y_0=y\}\ S\ \{z=x_0*y_0\}$

das Gewünschte. Über *x* und *y* am Ende von *S* wird nichts gesagt. Sollen diese unverändert bleiben, muß die Konjunktion $x=x_0 \wedge y=y_0$ der Nachbedingung hinzugefügt werden. Der Fall, daß logische Variablen zum Beschreiben des Ein/Ausgabeverhaltens benötigt werden ist sehr häufig, denn sonst wäre ja die Ausgabe unabhängig von der Eingabe. Dies führt zu der Konvention, daß mit null indizierte Variablen implizit für die Anfangswerte stehen. (Die folgenden Methoden machen diese Konvention zum Bestandteil des Kalküls)

Als unpraktisch erweist sich, daß zwischen Spezifikationen und Programmen ein fundamentaler Unterschied gemacht wird. Es wäre schöner, könnte man die Zuweisung *z:=x*y* direkt als die Spezifikation (und Implementierung) des obigen Problems auffassen, denn sie drückt das Gewünschte wesentlich kompakter aus. Daraus wird die Forderung abgeleitet, daß Programme und Spezifikationen miteinander kombinierbar sein sollten. Je nach der Komplexität kann dann entweder die mächtigere Spezifikationsnotation oder gleich die Progammiernotation eingesetzt werden.

Bei VDM wird, wie bei den folgenden Methoden, das Ein/Ausgabeverhalten durch ein Prädikat beschrieben, das sowohl die Ein- als auch die Ausgabevariablen direkt enthält, üblicherweise mit der Konvention, daß die Ausgabevariablen zur Unterscheidung einfach gestrichen werden. In VDM besteht eine Spezifikation aus diesem Prädikat, etwas verwirrend Nachbedingung genannt, und einer Vorbedingung, die dieselbe Rolle hat wie beim Hoare-Kalkül, d.h. den Definitionsbereich charakterisiert. Das Multiplikationsbeispiel lautet in VDM:

```
S
ex  wr z
    rd x,y
pre true
post z´=x*y
```

Neu gegenüber dem Hoare-Kalkül ist, daß die Ein- und Ausgabevariablen üblicherweise explizit mit angegeben werden. Spezifikationen dieser Art dürfen mit programmiersprachlichen Operatoren kombiniert werden, um so eine Zerlegung der Spezifikation nach den Verifikationsregeln in noch auszuprogrammierende Teile erhalten zu können. Eine Hierarchie von Spezifikationen kann dadurch gebildet werden, daß Nachbedingungen von Spezifikationen, geschrieben als *post-S*, in anderen Spezifikationen wiederverwendet werden.

Das explizite Formulieren der Vorbedingung führt eine Redundanz mit sich, denn die Vorbedingung muß immer kleiner sein als der implizite Definitionsbereich der Nachbedingung. Da Vorbedingungen sehr komplexe Ausdrücke sein können, werden sie oftmals verstärkt um sie besser schreiben zu können. Dies schränkt die Menge der möglichen Anwendungen unnötig ein. Zudem ist es umständlicher, mit Paaren von Vor- und Nachbedingungen zu rechnen als nur mit Nachbedingungen. Daraus folgern wir, daß es vorteilhaft ist, auf das obligatorische Hinschreiben der Vorbedingung zu verzichten und diese nur zu berechnen, wenn es wünschenswert oder beweistechnisch notwendig ist.

In Z bestehen Spezifikationen nur aus dem Ein/Ausgabeprädikat und den Typen der Ein- und Ausgabevariablen, Signatur genannt. Im Vergleich zu VDM entfällt der Definitionsbereich (Vorbedingung). Prädikat und Signatur zusammen werden als Schema bezeichnet. Beispiel:

```
S = [x,y,z´:INTEGER | z´=x*y]
```

Soll die obige Spezifikation für andere Typen als INTEGER verwendet werden, kann der Typ

importiert werden, d.h. wird Parameter der Spezifikation. Schemata können mit anderen Schemata kombiniert werden, wobei jeweils Signatur und Prädikat gemischt werden. Falls Σ_{out} die Menge der Ausgabevariablen eines Schemas S mit Prädikat P ist, ist die Vorbedingung implizit gegeben durch:

$$\text{pre } S \Leftrightarrow \exists \Sigma_{out} \cdot P$$

Auf Spezifikationen ist eine Verfeinerungrelation definiert, hier geschrieben als $S \sqsubseteq T$, die besagt, daß die konkretere Spezifikation S höchstens das macht, was T zuläßt und mindestens dort definiert ist, wo es T ist. Außerhalb des Definitionsbereiches von T darf S beliebiges Verhalten zeigen.

$$S \sqsubseteq T \quad \Leftrightarrow \quad \text{pre } T \Rightarrow ((S \Rightarrow T) \wedge \text{pre } S)$$

Weiterhin ist zu jedem programmiersprachlichen Operator eine Verifikationsregel gegeben, für die sequentielle Komposition beispielsweise:

$$\frac{\begin{array}{l}\text{pre } T \Rightarrow \text{pre } S_1 \\ (\text{pre } T) \wedge S_1 \Rightarrow (\text{pre } S_2)' \\ (\text{pre } T) \wedge S_1 \wedge S_2' \Rightarrow T[''\backslash'] \end{array}}{S_1 ; S_2 \sqsubseteq T}$$

S' bedeutet, daß alle Variablen aus S einen zusätzlichen Strich erhalten, und $T[''\backslash']$ bedeutet, daß alle einfach gestrichenen Variablen aus T doppelt gestrichen werden.

Z wurde weitgehend nur als Spezifikationsnotation konzipiert. Der Vorteil von Z liegt beim Strukturieren großer Spezifikationen durch die vielfältigen Möglichkeiten, Schemata zu kombinieren. In der Tat gibt es zu logischen Operatoren wie Konjunktion und Disjunktionen äquivalente Schemaoperatoren sowie weitere Operatoren wie (sequentielle) Komposition .

An der Art zu Verifizieren hat sich in Z gegenüber dem Hoare-Kalkül wenig verbessert. Die obige Regel läßt sogar schließen, daß das Verifizieren aufwendiger geworden ist. Komplexere Transformationsregeln sind nicht üblich.

Die Definition von Konjunktionen und Disjunktionen auf Schemata läßt Zweifel aufkommen, ob es nicht einfacher wäre, die Signatur des Schemas wegzulassen und die Spezifikationen direkt mit den logischen Operatoren zu verknüpfen. Die Definition der Komposition auf Schemata würde ebenfalls wegfallen wenn man Programme und Spezifikationen gleich behandelt. Dies würde zudem zulassen, Spezifikationen mit allen programmiersprachlichen Operatoren zu verknüpfen.

Hehners Ziel [1984] ist es, freies Mischen von Programmen und Spezifikationen zu ermöglichen, indem Spezifikationen in Prädikatenlogik (erster Stufe) geschrieben werden und alle programmiersprachlichen Operatoren ebenfalls in Logik definiert sind. Eine einfache Spezifikation ist z.B

$$x' > x$$

Sie besagt, daß das ausgegebene x größer sein muß als das anfängliche x. Eine Eigenart von

Hehners Spezifikationen ist, daß sie *total* sein müssen, d.h. für jede Belegung von Eingabevariablen muß (mindestens) eine Ausgabekonfiguration definiert sein. Damit ist

$$x' = 1/x$$

keine Spezifikation, und folglich auch nicht implementierbar, denn für $x=0$ ist kein Verhalten definiert. Eine gültige Spezifikation wäre:

$$x \neq 0 \Rightarrow x' = 1/x$$

Sie läßt für $x=0$ beliebiges Verhalten zu. Jede Spezifikation hat also die Form einer Implikation. Bei Hehner ist für die Anfangszustände, für die beliebiges Verhalten zugelassen ist (und nur für diese) auch Nichtterminierung zugelassen, mit der Begründung, daß, falls es einem Kunden egal ist, was er bekommt, ihm es auch egal sein muß, ob er überhaupt etwas bekommt. Demnach ist die allgemeinste (schwächste) Spezifikation gegeben durch die "Allrelation".

$$\text{abort} \Leftrightarrow \text{true}$$

Sie läßt sowohl Nichtterminierung als auch Termierung mit beliebigem Resultat zu. Mit diesen Konventionen gestalten sich zwei Operatoren besonders einfach: die nichtdeterministische Auswahl und die Verfeinerung, denn sie sind durch die logische Disjunktion bzw. Implikation gegeben.

$$P \vee Q$$

zeigt beliebiges Verhalten (einschließlich Nichtterminierung) genau dann wenn entweder P oder Q beliebiges Verhalten hat und terminiert dort wo sowohl P als auch Q terminieren, mit dem Resultat von entweder P oder Q.

$$P \Rightarrow Q$$

bedeutet, daß P deterministischer ist als Q. Da P und Q total sind, also P auch dort definiert ist wo es Q ist, läßt sich P an Stelle von Q verwenden. Beispiel:

$$x' = x+1 \Rightarrow x' > x$$

Die sequentielle Komposition $P;Q$ entspricht in erster Näherung dem relationalen Produkt $P°Q$, d.h. die Ausgabevariablen von P werden mit den Eingabevariablen von Q identifiziert. Das Problem ist, daß $P;Q$ beliebiges Verhalten zeigen muß wenn entweder P oder Q beliebiges Verhalten hat, was für P in $P°Q$ nicht gilt. Demnach lautet die (etwas umständliche) Definition von $P;Q$:

$$P;Q \Leftrightarrow (\neg \forall x' \cdot P) \Rightarrow (P°Q) \qquad \text{falls } x' \text{ die Menge der Ausgabevariablen von } P \text{ ist}$$

Der dramatische Nachteil dieser Definition der sequentiellen Komposition ist, daß sie nicht assoziativ ist, d.h. es gilt nicht immer $P;(Q;R) \Leftrightarrow (P;Q);R$, und daß sie nicht distributiv bezüglich der Auswahl ist, d.h. $(P \vee Q);R$ ist nicht dasselbe wie $(P;R) \vee (Q;R)$.

Das Anliegen von Hoare & He [1985] ist, zum einen zu einer saubereren Theorie zu gelangen als Hehner, d.h. bei der alle algebraischen Regeln einer indeterministischen, sequentiellen Programmiersprache uneingeschränkt gelten, und zum anderen einen Formalismus vorzustellen, der, in starker Analogie zu Dijkstras *schwächsten Vorbedingungen* [1976], die Programmentwicklung unterstützt.

Hoare & He betrachten dazu wie Hehner Transformationen des Anfangszustandes in den Endzustand, nur mit dem Unterschied, daß der Zustandsraum um ein fiktives Element $\perp$ (bottom) erweitert wurde, das, einfach ausgedrückt, für Nichtterminierung steht. Betrachtet werden wieder nur totale Relationen, jetzt mit der zusätzlichen Einschränkung daß $\perp$ auf alles einschließlich $\perp$ abgebildet werden muß. *Abort* ist wieder definiert durch die Allrelation, d.h. die Relation, die jeden beliebigen Endzustand einschließlich $\perp$ zuläßt. Im Gegensatz zu Hehner wird hier zwischen beliebigem aber terminierendem Verhalten (der Wertebereich ist alles außer $\perp$) und unkontrollierbarem Verhalten (terminierend oder nicht terminierend) unterschieden (der Wertebereich ist alles einschließlich $\perp$). Die Definition der indeterministischen Auswahl und der Implementiert-Relation (Verfeinerungs-Relation) ist dieselbe wie bei Hehner. Die sequentielle Komposition entspricht jetzt dem relationalen Produkt, womit die algebraischen Regeln zum Teil direkt aus dem Relationenkalkül folgen.

Als ein Hilfsmittel zur Programmentwicklung definieren Hoare & He zwei neue Operatoren für die schwächste Vor- und Nachspezifikation (*weakest prespecification* und *weakest postspecification*). Die schwächste Vorspezifikation von R bezüglich P ist das allgemeinste (schwächste) X, so daß $X;R$ die Spezifikation P erfüllt. Analog dazu ist die schwächste Nachspezifikation von Q bezüglich P definiert als das allgemeinste X, so daß $Q;X$ die Spezifikation P erfüllt.

Wird zur Spezifikation eine Nachbedingung p verwendet, gilt für die schwächste Vorbedingung von R bezüglich p laut Dijkstra, daß sie als die (allgemeinste) Nachbedingung von Q aufgefaßt werden kann, so daß $Q;R$ die Nachbedingung p erfüllt. Diese Analogie zur schwächsten Vorspezifikation spiegelt sich in einer Reihe von analogen Gesetzen wieder, nur daß statt der Nachbedingung p die Gesamtspezifikation P genommen wird.

Der Nutzen von Vorbedingungen und Vorspezifikationen ist aber ein unterschiedlicher. Schwächste Vorbedinungen werden von Dijkstra verwendet, um Bedingungen (*guards*) von Alternativen und Schleifen zu berechnen, was mit Vorspezifikationen nicht möglich ist. Ein Einsatzgebiet von schwächsten Vor- und Nachspezifikationen ist bei Datenabstraktionen das Herleiten der konkreten Operation, falls die abstrakte Operation und die Beziehung zwischen konkreten und abstrakten Daten gegeben ist, wie Hoare, He & Sanders [1986] gezeigt haben. Die praktische Arbeit erfordert allerdings eine große Anzahl von Regeln und erweist sich als nicht einfach.

Das Einführen von $\perp$ zieht eine Reihe technischer Konsequenzen mit sich. Ist beispielsweise $\perp$ in der Bildmenge eines Anfangszustandes , muß die Bildmenge dieses Zustandes alle möglichen Endzustände ebenfalls enthalten. Noch fundamentaler ist, daß sich Relationen mit $\perp$ im Zustandraum schlecht schreiben lassen. Die Autoren schlagen deshalb vor, diese als Paar zu schreiben, mit der ersten Komponente einer Bedingung, die die Anfangszustände charakterisiert, die nicht auf $\perp$ abgebildet werden (= Definitionsbereich, Vorbedingung in VDM Terminologie) und der zweiten Komponente einem Prädikat, das das Übergangsverhalten im Defintionsbereich beschreibt (=Nachbedingung in VDM Terminologie). Wie bei VDM hat dies den Nachteil, daß es umständlich ist, mit Paaren zu

rechnen und dies zeigen auch die Beispiele der Autoren.

Hehner (s.o.) war sich dessen bewußt, daß sich durch Rechnen mit Paaren in seinem Kalkül ebenfalls wünschenswerte Eigenschaft wie Assoziativität der sequentiellen Komposition ergeben hätten. Da aber, wie er zeigt, diese Unschönheiten nur in rein theoretischen Fällen auftreten, nimmt er diese lieber in Kauf statt mit Paaren rechnen zu müssen.

Sowohl Hehner als auch Hoare & He fordern von Spezifikationen *Stetigkeit*, was den Umgang mit Rekursionen (und Iterationen als Spezialfall von Rekursionen) vereinfacht. Stetigkeit führt aber zur Beschränktheit des Indeterminismus. Damit ist

$$x' > 0$$

eine unzulässige Spezifikation, denn sie läßt unbeschränkt viele Resultate zu. Diese Einschränkung der Spezifikationen ist sowohl lästig, weil schwierig nachzuweisen, aus auch unnötig.

Back [1988] betrachtet ebenfalls die Entwicklung von Programmen aus Spezifikationen, jedoch mit einer dualen Betrachtungsweise. Für ihn ist eine Spezifikation eine Art (abstrakte) Anweisung neben den üblichen programmiersprachlichen Anweisungen. Spezifikationen sind jetzt Spezialfälle von (abstrakten) Programmen und nicht umgekehrt wie bisher.

Programmentwicklung besteht nun darin, von einer einzigen Anweisung, der Spezifikation des Problems, ausgehend diese zu zerlegen in ausführbare Anweisungen und weiteren nicht ausführbaren Anweisungen (Teilspezifikationen). Die Semantik von Anweisungen beschreibt Back genau wie Dijkstra [1976] durch schwächste Vorbedingungen.

Spezifikationen werden durch *indeterministische Zuweisungen* formuliert, in denen die zugewiesenen Variablen explizit aufgezählt werden und ihre Werte durch ein Prädikat mit gestrichenen und ungestrichenen Variablen beschrieben werden.

Back verwendet schwächste Vorbedingungen nicht nur um die Semantik von Anweisungen anzugeben, sondern auch um eine Verfeinerungsrelation zwischen Anweisungen zu definieren. Eine Anweisung S verfeinert eine Anweisung T (kann statt T verwendet werden) falls, wann immer T eine Nachbedingung q erfüllt, auch S diese Nachbedingung erfüllt.

$$S \sqsubseteq T \quad \Leftrightarrow \quad wp(T,q) \Rightarrow wp(S,q) \qquad \text{für alle } q$$

Weiterhin sind bei Back Spezifikationen mit unbeschränktem Indeterminismus zugelassen. Dies wird erreicht, indem die Bedeutung von Rekursionen und Iterationen durch Disjunktionen über der Menge der transfiniten Ordinalzahlen, und nicht der endlichen, definiert werden.

Bei Backs Kalkül können zwar Spezifikationen in Anweisungen verwendet werden, aber umgekehrt können Anweisungen nicht in Spezifikationen auftreten. Ein freies Mischen ist also nicht möglich. Etwas verwirrend ist auch die Tatsache, daß ein Problem jetzt auf zweierlei Arten beschrieben werden kann: einmal als abstrakte Anweisung und einmal als Nachbedingung, wobei Back in seinen Beispielen nur abstrakte Anweisungen verwendet. Demnach sieht es so aus, daß Nachbedingungen und Vorbedingungen nur als Hilfsmittel zur Semantikdefinition und zum Beweisen benötigt werden. Es erhebt sich die Frage, ob sie somit ein adäquates Mittel für die intendierte Anwendung sind.

Die Arbeit von Morgan [1988] vereinigt in gewisser Hinsicht die Ideen von Back und die des Relationenkalküls. Spezifikationen werden als *Spezifikationsanweisungen* (*specification statements*) geschrieben. Sie haben dieselbe Rolle wie Backs indeterministische Zuweisung, nur mit dem Unterschied, daß außer dem Prädikat, das das Resultat beschreibt, auch der Definitionsbereich explizit geschrieben wird.

[pre, post]

pre ist ein Prädikat über die Eingabevariablen (=Vorbedingung in VDM) und *post* ein Prädikat über Ein- und Ausgabevariablen (=Nachbedingung in VDM). Morgan definiert die Bedeutung dieser Spezifikationsanweisungen, wie die aller anderer Sprachelemente, durch schwächste Vorbedingungen. Für den einfachen Fall, daß *post* ein Prädikat nur auf den gestrichenen Ausgabevariablen ist, lautet die Definition:

$$wp([pre, post], q) \Leftrightarrow pre \wedge \forall \Sigma' \cdot post \Rightarrow q'$$

wobei Σ' die Menge aller gestrichenen Variablen ist und q' für q mit allen freien Variablen einfach gestrichen steht.

Im Relationenkalkül, wie in VDM, muß für solche *pre-post* Paare gelten, daß *pre* schwächer ist als der implizite Definitionsbereich von *post*, d.h. daß *post* (mindestens) ein Resultat zuläßt falls *pre* gilt. Morgan läßt diese Restriktion fallen und gelangt zu Spezifikationsanweisungen, die sich wie *Wunder* verhalten: Es gibt Anfangszustände, für die eine Anweisung kein Verhalten zeigt und trotzdem definiert ist. Das extremste Beispiel ist

[true, false]

denn die Anweisung ist immer definiert und erfüllt jede Spezifikation (*false* impliziert alles, d.h. nach obiger Regel ist die schwächste Vorbedingung dieser Anweisung bezüglich jeder beliebigen Nachbedingung wahr). Als prinzipiellen Grund, weshalb es sinnvoll ist Wunder einzuführen, nennt Morgan Vereinfachungen bei der Herleitung von Programmen, genau so wie negative Zahlen das Lösen von Gleichungen vereinfachen, auch wenn das Resultat positiv ist. Wunder entstehen bei der Programmentwicklung, falls man Fehlentscheidungen getroffen hat, beispielsweise wenn man ein *X* sucht, so daß *Q;X* die Spezifikation *P* erfüllt und durch ungeschickte Wahl von *Q* dies nicht möglich ist. Das Problem bei Wundern ist, daß sie nie in eine ausführbare Anweisung transformiert werden können und deshalb in diesem Fall auch nicht helfen. Morgan gibt auch eine positive Anwendung von Wundern, wenn z.B zwei Anweisungen eine Spezifikation durch (partielle) Wunder erfüllen aber eine bedingte Alternative mit diesen Anweisungen die Spezifikation ohne Wunder erfüllt.

Aus den bisherigen Betrachtungen leiten wir folgende Forderungen an unseren Kalkül ab.

1. Spezifikationen werden durch ein Prädikat mit Anfangs- und Endvariablen geschrieben, und nicht wie in Hoare-Logik durch Vor- und Nachbedingung.

2. Die Typen der Variablen und der Definitionsbereich werden, im Gegensatz zu Z bzw. VDM und Relationenkalkül, nicht explizit hingeschrieben.

3. Alle Programmoperatoren werden in Prädikatenlogik definiert und daraus alle algebraische Gesetze abgeleitet

4. Unbeschränkter Indeterminismus wird zugelassen

2. Der Prädikatenkalkül

In dem hier vorgestellten Prädikatenkalkül wird die Bedeutung aller Programmoperatoren durch logische Formeln definiert. Dadurch können in Logik geschriebene Spezifikationen durch Regeln der Logik in äquivalente Programme transformiert werden. Zusätzlich wird eine Implementierungsordnung definiert, die angibt wann eine Spezifikation durch eine deterministischere ersetzt werden darf.

Notation. Im folgenden werden prädikatenlogische Formeln betrachtet, bestehend aus Negationen, Konjunktionen, Disjunktionen, Implikationen, Äquivalenzen, Allquantoren und Existenzquantoren in der üblichen Schreibweise und mit der üblichen Bedeutung. Jede zusammengesetzte Formel hat damit eine der Formen

$$\neg p \qquad p \wedge q \qquad p \vee q \qquad p \Rightarrow q \qquad p \Leftrightarrow q \qquad \forall x \cdot p \qquad \exists x \cdot p$$

wobei p und q wieder Formeln sind. Falls aus dem Kontext die gebundene Variable x ersichtlich ist, wird abkürzend geschrieben:

$$\forall p(x) \cdot q(x) \quad \Leftrightarrow \quad \forall x \cdot p(x) \Rightarrow q(x)$$
$$\exists p(x) \cdot q(x) \quad \Leftrightarrow \quad \exists x \cdot p(x) \wedge q(x)$$

$p(x)$ bedeutet, daß x in p frei auftritt. Die simultane Substitution der Variablen des Vektors X durch die entsprechenden aus Y in P wird geschrieben:

$$P_X^Y$$

Spezifikationen. Unter Spezifikationen werden Beschreibungen des Ein/Ausgabeverhaltens einer abstrakten Maschine verstanden. Eine Spezifikation besagt, welcher Anfangszustand, gegeben durch die Werte der *Eingabevariablen*, in welchen Endzustand, gegeben durch die

Werte der *Ausgabevariablen*, überführt werden kann. Spezifikationen werden geschrieben durch prädikatenlogische Formeln über *ungestrichenen* Eingabevariablen und *gestrichenen* Ausgabevariablen geschrieben.

Spezifikationen haben im allgemeinen die Eigenschaften, *indeterministisch* und *partiell* zu sein. Indeterministisch bedeutet, daß für einen Anfangszustand mehrere Endzustände zugelassen sind. Ist höchstens ein Endzustand zugelassen, heißt die Spezifikation *deterministisch* (oder funktional). Partiell bedeutet, daß für einen Anfangszustand kein Endzustand definiert ist. Mit Σ' der Menge aller gestrichenen Variablen (Ausgabevariablen) ist der *Definitionsbereich* einer Spezifikation *P* gegeben durch:

$$\Delta P \Leftrightarrow \exists \Sigma' \cdot P$$

Ist *ΔP* äquivalent zu *TRUE*, heißt *P total*. Spezifikationen, die nur ungestrichene Variablen enthalten, werden *Bedingungen* genannt. Ist *b* eine Bedingung, erhält man *b´* durch das einfache Streichen aller in *b* vorkommenden freien Variablen. Gelegentlich schreiben wir *b** und meinen, daß alle freie Variablen in *b* gesternt werden sollen.

Konstantendeklarationen. Die Bedeutung einer Konstantendeklaration ist die, daß überall statt dem Konstantenbezeichner der in der Deklaration angegebene Wert verwendet wird.

$$\text{CONST } x=e;\ P \Leftrightarrow P_x^e \quad \text{und } x' \text{ nicht frei in } P$$

Zu beachten ist, daß *x* nicht zu dem Kontext gehört, der beim Auflösen der Zuweisungen und sequentiellen Kompositionen verwendet wird (siehe Variablendeklarationen).

Typdeklarationen. Unter dem Typ einer Programmvariablen wird eine Eigenschaft verstanden, die während ihrer ganzen Existenz gilt, d.h. nicht nur für ihren Initial- und Finalwert sondern auch für jeden Zwischenwert während der sequentiellen Komposition. Der Typ einer Programmvariablen *x* ist die Menge der Werte, die *x* annehmen kann. Wir schreiben dafür

$$x:T$$

Als Typen werden die einfachen Typen INTEGER, CARDINAL, BOOLEAN und CHAR (mit ihrer üblichen Bedeutung) sowie die zusammengesetzten Typen Variante, Menge, Verbund, Sequenz, Feld, (sequentielle) Datei und Zeiger verwendet. Die Variante (disjunkte Vereinigung) wird geschrieben als

$$T_1 \mid \ldots \mid T_n$$

wobei die T_i verschiedene Typbezeichner sind. Eine Variable vom Typ $T_1 \mid \ldots \mid T_n$ ist von genau einem der Typen $T_1 \ldots T_n$. Dies wird durch

$$x \text{ IS } T_i$$

getestet. Der Ausdruck $x \text{ IS } T_i,\ldots,T_j$ steht abkürzend für:

$$x \text{ IS } T_i,\ldots,T_j \Leftrightarrow x \text{ IS } T_i \vee \ldots \vee x \text{ IS } T_j$$

Die Potenzmenge eines Typs *T* wird bezeichnet durch:

$$\text{SET OF } T$$

Auf Variablen von diesem Typ sind die üblichen Operatoren in der üblichen Notation definiert. Der Verbund (das kartesische Produkt) von *n* Komponenten vom Typ T_i wird geschrieben als

$$(x_1:T_1; \ldots ; x_n:T_n)$$

oder konventioneller als

RECORD x_1:T_1; ... ; x_n:T_n END

mit der Selektion der Komponente x_i aus dem Verbund v durch $v.x_i$. Eine Sequenz mit Elementen vom Typ T wird geschrieben als:

SEQUENCE OF T

Folgende Operationen sind auf Sequenzen s und t definiert:

$\|s\|$	Länge der Sequenz s,
s[i]	i-tes Element der Sequenz s mit $0 \le i < \|s\|$,
$\langle a_0, \ldots, a_{n-1} \rangle$	Konstruktion einer n-elementigen Sequenz,
s°t	Konkatenation von s und t,
s[i..j]	Selektion der Subsequenz von i bis j,
set(s)	Menge der in s enthaltenen Elemente,
flatten(s)	Konkatenation der Elemente aus s, falls diese Sequenzen sind.

Felder werden aufgefaßt als Sequenzen mit fixer Länge, geschrieben als

ARRAY n OF T

mit n der Länge des Feldes, oder allgemeiner als:

ARRAY [m..n] OF T

Es gelten dieselben Operationen wie bei Sequenzen. Sequentielle Dateien werden bezeichnet durch

FILE OF T

mit dem Elementtyp T. Der Ausdruck

POINTER TO T

steht für eine unendliche und ungeordnete Menge von neuen Elementen, die "Zeiger" auf Objekte vom Typ T sind.

Einem Typ U wird durch die Deklaration

TYPE T=U; P

der Name T innerhalb von P gegeben. Zu beachten ist, daß T nicht zu dem Kontext gehört, der beim Auflösen der Zuweisungen und sequentiellen Kompositionen verwendet wird (siehe unten).

Variablendeklarationen. Für einen Typ T gilt:

VAR x: T; P $\quad\Leftrightarrow\quad \exists x' : T \cdot P_x^{\perp x}$

$\perp x$ steht für den initialen Wert von x bei der Ausführung von P. Über dessen Wert kann nichts angenommen werden, außer daß er, hier vereinfachend betrachtet, vom Typ T ist. Der *Kontext* einer Spezifikation P ist die Menge der durch P umgebende Variablen-deklarationen eingeführten Bezeichner.

Elementare Anweisungen. Seien $x_1 \ldots x_n$ die durch den Kontext gegebenen Variablen und e ein Ausdruck über diese Variablen. Dann gilt für die undefinierte Anweisung, die leere Anweisung und die Zuweisung:

ABORT	$\Leftrightarrow$	FALSE
SKIP	$\Leftrightarrow$	$x_1' = x_1 \wedge \ldots \wedge x_n' = x_n$
x:=e	$\Leftrightarrow$	$x_1' = x_1 \wedge \ldots \wedge x' = e \wedge \ldots \wedge x_n' = x_n$

Für die Definitionsbereiche gilt

$$\begin{array}{lcl} \Delta\text{ABORT} & \Leftrightarrow & \text{FALSE} \\ \Delta\text{SKIP} & \Leftrightarrow & \text{TRUE} \\ \Delta x{:=}e & \Leftrightarrow & \Delta e \end{array}$$

wobei Δe der Definitonsbereich des Ausdrucks *e* ist.

Sequentielle Komposition. Seien $x_1..x_n$ die durch den Kontext gegebenen Variablen und seien $T_1..T_n$ ihre zugehörigen Typen. Dann gilt für die sequentielle Komposition zweier Spezifikationen *P* und *Q*:

$$P;Q \Leftrightarrow \exists x_1^*{:}T_1..x_n^*{:}T_n \cdot P_{x_1..x_n}{}^{x_1^*..x_n^*} \wedge Q_{x_1'..x_n'}{}^{x_1^*..x_n^*}$$

Für den Definitionsbereich gilt:

$$\Delta(P;Q) \Leftrightarrow P;\Delta Q$$

Alternativen. Für eine Bedingung *b* und Spezifikationen *P* und *Q* gilt:

$$\text{IF b THEN P ELSE Q END} \Leftrightarrow (b \wedge P) \vee (\neg b \wedge Q)$$

$$\text{IF b THEN P END} \Leftrightarrow \text{IF b THEN P ELSE SKIP END}$$

Für die Definitionsbereiche ergibt sich:

$$\Delta\text{IF b THEN P ELSE Q END} \Leftrightarrow (b \wedge \Delta P) \vee (\neg b \wedge \Delta Q)$$

$$\Delta\text{IF b THEN P END} \Leftrightarrow (b \wedge \Delta P) \vee \Delta Q$$

Als Abkürzung wird auch eine ELSIF-Konstruktion zugelassen.

$$\text{IF } b_1 \text{ THEN } P_1 \text{ ELSIF } b_2 \text{ THEN } P_2 \text{ ELSE } P_3 \text{ END}$$

$$\Leftrightarrow \quad \text{IF } b_1 \text{ THEN } P_1 \text{ ELSE IF } b_2 \text{ THEN } P_2 \text{ ELSE } P_3 \text{ END END}$$

Iteration. Die Definition der allgemeinen Iteration erfordert einen zusätzlichen Operator für die Bildung des *kleinsten Fixpunktes*. Stehe

$$\mu X \cdot P(X)$$

für das kleinste *X* (bezüglich der Implikation), so daß $X \Leftrightarrow P(X)$ gilt. Die WHILE-Schleife ist damit für eine Bedingung *b* und eine Spezifikation *P* gegeben durch:

$$\text{WHILE b DO P END} \Leftrightarrow \mu X \cdot \text{IF b THEN P;X END}$$

Eine Lösung von $\mu X \cdot P(X)$ existiert, falls *P* monoton ist, d.h. aus $A \Rightarrow B$ folgt $P(A) \Rightarrow P(B)$ für Spezifikationen *A* und *B*. Dies ist gesichert, wenn *X* in *P(X)* nur in nichtnegierten Formeln auftritt, also insbesondere für die WHILE-Schleife.

Ein Spezialfall der allgemeinen Iteration ist die FOR-Schleife. Für Ausdrücke *e* und *f* gilt:

$$\text{FOR } e \le x < f \text{ DO P(x) END} \Leftrightarrow \text{P(e); P(e+1); ... ; P(f-1)}$$

Für $e > f$ ist die FOR-Schleife äquivalent zu SKIP.

Programme und Programmentwicklung. Programme sind Spezifikationen, die nur aus Konstantendeklarationen, Typdeklarationen, Variablendeklarationen, Zuweisungen, sequentiellen Kompositionen, Fallunterscheidungen und Iterationen bestehen. Bei der

Transformation von Spezifikationen in Programme sind außer den äquivalenzerhaltenden Transformationen auch solche zugelassen, die entweder die Spezifikation deterministischer machen oder ihren Definitionsbereich erweitern. Für eine Spezifikation P bedeutet $P \sqsubseteq Q$, daß P die Spezifikation Q *erfüllt* oder *implementiert*.

$$P \sqsubseteq Q \quad \Leftrightarrow \quad \Delta Q \Rightarrow (\Delta P \wedge P \Rightarrow Q)$$

Die erste Aussage, $\Delta Q \Rightarrow \Delta P$, besagt, daß der Definitionsbereich von P mindestens genau so groß sein muß wie der von Q. Die zweite Aussage, $\Delta Q \Rightarrow (P \Rightarrow Q)$, besagt, daß P deterministischer sein muß als Q innerhalb des Definitonsbereiches von Q, und beliebig sein kann außerhalb. Einige elementare Eigenschaften der *Implementierungsrelation* sind:

1.	$P \sqsubseteq P$	Reflexivität
2.	$P \sqsubseteq \_P \Rightarrow P \Leftrightarrow Q$	Antisymmetrie
3.	$P \sqsubseteq \_R \Rightarrow P \sqsubseteq R$	Transitivität
4.	$P \sqsubseteq ABORT$	Maximum
5.	$P \sqsubseteq (b \wedge P)$	
6.	$b \Rightarrow (P \sqsubseteq Q) \quad \Leftrightarrow \quad P \sqsubseteq (b \wedge Q)$	

Die beiden letzten Aussagen werden durch Fallunterscheidung über b bewiesen. Nach Bedarf werden später noch weitere, komplexere Gesetze folgen.

3. Problematik eines Assemblierers

In diesem Kapitel wird die Aufgabe eines Assemblierers skizziert und die Struktur seiner Spezifikation vorgestellt. Diese Spezifikation soll in den folgenden Kapiteln verfeinert und implementiert werden. Die Modularisierung des resultierenden Programmes wird hier – in einer abstrakten Form – ebenfalls vorgestellt.

Ein Assemblierer hat, analog zu einem allgemeinen Übersetzer, die Aufgabe, ein *Quellprogram* aus einer sequentiellen Textdatei binär zu kodieren zu einem *Objektprogramm* in einer sequentiellen Datei aus Maschinenwörtern. Für die Ein- und Ausgabe eines Assemblierers ergibt sich:

source: FILE OF CHAR
object: FILE OF Word

Die Quellprgramme sind hier MC68020 Assemblerprogramme, auf die Instruktionen und Adressierungsarten beschränkt, wie sie bei der Übersetzung von Pascal-ähnlichen Sprachen benötigt werden. Das genaue Format der Assemblerprogramme ist im Anhang beschrieben.

Unsere Vorstellung von einem Übersetzungsprozeß ist, daß er aus einer Analyse (*Zerteilung*) des Quellprogrammes und einer *Generierung* des Objektprogrammes besteht.

assemble: parse; generate

Parse verarbeitet die Zeichen aus *source* und *generate* erzeugt die Wörter von *object*. Die Schnittstelle zwischen *parse* und *generate* ist eine abstrakte *Zwischenrepräsentation* des

Quellprogrammes; sie enthält alle für die Generierung relevanten Informationen. Im Falle eines Assemblierers ist diese Zwischenrepräsentation eine Sequenz von Instruktionen, die gegebenenfalls mit einer *Marke* gekennzeichnet sein können.

```
prg: SEQUENCE OF LabelledInstruction
```

Prg ist somit die Ausgabe von *parse* und die Eingabe von *generate*. Mit der Notation *[T]* für *T* oder *NIL* wird definiert:

```
LabelledInstruction = (l: [Label]; i: Instruction)
```

Die Struktur der Marken ist an dieser Stelle irrelevant; wir nehmen lediglich an, daß auf Marken die Gleichheit definiert ist, damit diese als Sprungziel von Instruktionen auftreten können. Die Struktur der Instruktionen dagegen ist vollständig bestimmt durch den zugrundeliegenden Prozessor, hier den MC68020:

```
Instruction =
   rtr | rts | trapv |
   asl1 | asr1 | jmp | jsr | moveccrea | moveeaccr | pea |
      clrb | clrw | clrl | negb | negw | negl | notb | notw | notl | tstb | tstw | tstl |
      st | sf | shi | sls | scc | scs | sne | seq | svc | svs | spl | smi | sge | slt | sgt | sle |
      bra | brs | bhi | bls | bcc | bcs | bne | beq | bvc | bvs | bpl | bmi | bge | blt | bgt | ble |
      extw | extl | swap | unlk | rtd | trap | datab | dataw | datal |
   addb | addw | addl | andb | andw | andl | orb | orw | orl | subb | subw | subl |
      eorb | eorw | eorl | bchg | bclr | bset | btst |
      lea | mulsw | muluw |
      adddb | adddw | adddl | cmpb | cmpw | cmpl | subdb | subdw | subdl |
      anddb | anddw | anddl | chkw | chkl | divs | divu | ordb | ordw | ordl |
      cmpaw | cmpal |
      addaw | addal | subaw | subal |
      addib | addiw | addil | subib | subiw | subil |
      andib | andiw | andil | cmpib | cmpiw | cmpil | eorib | eoriw | eoril | orib | oriw | oril |
      bchgi | bclri | bseti | btsti | moveregeaw | moveregeal |
      moveregaw | moveregl |
      aslb | aslw | asll | asrb | asrw | asrl |
      aslib | asliw | aslil | asrib | asriw | asril |
      dbra | link | moveb | movew | movel |
      mulsl | mulul |
   divls | divlu |
   var
```

Für die Anzahl der Operanden gilt:

rtr..trapv	nehmen keine Operanden,
asl1..datal	haben einen Operanden,
addb..mulul	haben zwei Operanden,
divls, divlu	haben drei Operanden,
var	hat zwei Operanden.

Var ist eine sogenannte *Pseudoinstruktion*. Für sie wird kein Code generiert, sondern verändert lediglich den Zustand des Codegenerators und hat damit einen Effekt auf nachfolgend generierte Instruktionen.

Einige Instruktionen stehen für dieselbe arithmetische oder logische Operation und unterscheiden sich nur durch den Suffix *b*, *w* oder *l*, der für die Länge des Operanden Byte, Wort oder Langwort steht. Eine andere Klassifizierung der Instruktionen wäre gewesen, gleiche Operationen zusammenzufassen und die Länge als weiteren Operanden (Parameter) anzugeben. Dies würde einerseits die Aufzählung der Instruktionen verkürzen aber anderseits das Schema der Codegenerierung komplizierter machen. Die Entscheidung für die längere Aufzählung wurde getroffen weil die Instruktionen der Assemblersprache ebenfalls diese Form haben und somit eine Konvertierung entfällt.

Dasselbe Argument trifft auch auf die *set conditionally* Instruktionen *st..sle* und die *branch conditionally* Instruktionen *bhi..ble* zu. Hier wäre es ebenfalls möglich gewesen, die Bedingung als weiteren Operanden zu nehmen.

Der Aufbau der Operanden sei wie folgt:

```
Operand = Dn | An | indirect | postinc | predec | offset | index | direct |
    absolute | immediate
Dn,An,indirect,postinc,predec = (reg: [0..7])
offset = (reg: [0..7]; disp: Long)
index = (reg: [0..7]; disp: Long; indexreg: [0..15] (*0..7: Dn, 8..15: An*); scale: {1,2,4,8})
direct = (l: Label)
absolute, immediate = (n: Long)
```

Long steht für den Bereich der vorzeichenbehafteten 32 Bit Zahlen im Zweierkomplement, und *Word* für dieselbigen mit 16 Bits.

Um die Typen der einzelnen Instruktionen zu definieren, ist es notwendig die zulässigen Operanden jeder Instruktion aufzuzählen. Eine unangenehme Eigenschaft des MC68020 Prozessors ist, daß es sehr viele verschiedenartige Einschränkungen bei der Kombination von Operation und Operanden gibt. Diese Einschränkungen führen zu folgender Klassifizierung der Instruktionen.

```
rtr, rts, trapv = ()

asl1, asr1 = (x: indirect | postinc | predec | offset | index | direct | absolute)
jmp, jsr, pea = (x: indirect | offset | index | direct | absolute)
moveccrea, moveeaccr,
    clrb, clrw, clrl, negb, negw, negl, notb, notw, notl, tstb, tstw, tstl,
    st, sf, shi, sls, scc, scs, sne, seq, svc, svs, spl, smi, sge, slt, sgt, sle =
    (x: Dn | indirect | postinc | predec | offset | index | direct | absolute)
bra, brs, bhi, bls, bcc, bcs, bne, beq, bvc, bvs, bpl, bmi, bge, blt, bgt, ble = (x: direct)
extw, extl, swap = (x: Dn)
unlk = (x: An)
rtd, trap, datab, dataw, datal = (x: immediate)
addb, addw, addl, andb, andw, andl, orb, orw, orl, subb, subw, subl =
    (x: Dn; y: indirect | postinc | predec | offset | index | direct | absolute)
eorb, eorw, eorl, bchg, bclr, bset, btst =
    (x: Dn; y: Dn | indirect | postinc | predec | offset | index | direct | absolute)
lea = (x: indirect | offset | index | direct | absolute; y: An)
mulsw, muluw =
    (x: Dn | indirect | postinc | predec | offset | index | direct | absolute | immediate; y: Dn)
adddb, adddw, adddl, cmpb, cmpw, cmpl, subdb, subdw, subdl =
    (x:Dn | An | indirect | postinc | predec | offset | index | direct | absolute | immediate; y:Dn)
anddb, anddw, anddl, chkw, chkl, divs, divu, ordb, ordw, ordl =
    (x: Dn | indirect | postinc | predec | offset | index | direct | absolute | immediate; y: Dn)
```

cmpaw, cmpal, addaw, addal, subaw, subal =
(x:Dn | An | indirect | postinc | predec | offset | index | direct | absolute | immediate; y:An)
addib, addiw, addil, subib, subiw, subil,
andib, andiw, andil, cmpib, cmpiw, cmpil, eorib, eoriw, eoril, orib, oriw, oril,
bchgi, bclri, bseti, btsti =
(x: immediate; y: Dn | indirect | postinc | predec | offset | index | direct | absolute)
moveregeaw, moveregeal =
(x: immediate; y: indirect | predec | offset | index | direct | absolute)
moveareagw, moveeareagl =
(x: indirect | postinc | offset | index | direct | absolute; y: immediate)
aslb, aslw, asll, asrb, asrw, asrl = (x: Dn; y: Dn)
aslib, asliw, aslil, asrib, asriw, asril = (x: immediate; y: Dn)
dbra = (x: Dn; y: direct)
link = (x: An; y: immediate)
moveb, movew, movel =
(x: Dn | An | indirect | postinc | predec | offset | index | direct | absolute | immediate;
y: Dn | An | indirect | postinc | predec | offset | index | direct | absolute)
mulsl, mulul =
(x: Dn | indirect | postinc | predec | offset | index | direct | absolute | immediate; y: Dn)
divls, divlu = (x: Dn | postinc | predec | indirect | index; y: Dn; z: Dn)
var = (x: direct; y: absolute)

Die obige Liste weicht etwas von der originalen Klassifizierung ab: Überladene Operationen wurden durch Umbenennung vermieden, so daß die Operation ohne Analyse der Operanden erkennbar ist. Beispielsweise wurde

addw ea, Dn
addw Dn, ea

umbenannt in

adddw ea, Dn
addw ea, Dn

respektive, in Analogie zu:

addaw ea, An

Der Programmierer (bzw. der Übersetzer) weiß ohnehin, welche Operation er meint, und so ist es kein Mehraufwand, dies explizit zu machen. Auf der anderen Seite vereinfacht diese Unterscheidung die Spezifikation der Übersetzung (und natürlich die Implementierung).

Die Instruktionen *addqx, subqx* und *moveq* sind nicht explizit aufgezählt. Stattdessen sollen sie vom Generator wann immer möglich statt *addix, subix* und *movex* erzeugt werden. Diese Optimierung sollte in einem Übersetzer ohnehin an einer zentralen Stelle erfolgen, und so scheint es angebracht, dies in den Assemblierer zu verlegen, um so zusätzlich eine dünnere Schnittstelle zu erhalten.

Im folgenden soll unsere Aufmerksamkeit mehr dem Generator gewidmet sein. Von dem Zerteiler wird angenommen, daß er in einem Lauf sequentiell arbeitet, d.h. er liest einige Zeichen aus der Quelldatei und erzeugt daraus die nächste Instruktion. Er wird direkt mit der Technik des *rekursiven Abstiegs* implementiert.

Das Ziel der Entwicklung ist es, eine Implementierung des Generators zu finden, die ebenfalls ein einmaliges, sequentielles Bearbeiten der Instruktionen ermöglicht. Dies ermöglicht, den Generator mit dem Zerteiler so zu verzahnen, daß die Generierung der Instruktionen gleich nach ihrer Analyse erfolgen kann, ohne daß eine Zwischenrepräsentation des gesamten Programmes notwendig ist. In erster Näherung hat die Implementierung des Generators die Form:

generate

≡ InitGenerator; FOR 0≤n<|prg| DO process(prg[n]) END; CloseGenerator

process (l,i): IF l≠NIL THEN SetTarget(l) END; gen(i)

Die verwendeten Prozeduren lassen sich zusammenfassen zu einem Modul mit folgender Schnittstelle:

gen(i: Instruction)
SetTarget (l: Label)
InitGenerator
CloseGenerator

4. Auflösen der distanzabhängigen Sprünge

Es soll im folgenden die erste Form der Spezifikation des Generators, die die zulässigen Übersetzungen von distanzabhängigen Sprüngen definiert, vorgestellt werden. Diese Spezifikation wird in zwei Transformationschritten implementiert. Beim ersten wird die Spezifikation deterministisch gemacht und beim zweiten sequentialisiert.

Die Spezifikation des Generators besagt im wesentlichen, wie die Instruktionen, bestehend aus einer Operation und null bis drei Operanden, bitweise codiert werden sollen. Diese Codierung ist im allgemeinen lediglich von der Operation und den Operanden abhängig und zudem deterministisch, und damit als Funktion der Operation und Operanden ausdrückbar. Eine Ausnahme bilden Instruktionen mit symbolischen Referenzen (Adressen) auf andere Instruktionen des Assemblerprogrammes. Die Codierung einer symbolischen Adresse als relative Adresse (Distanz) zu der referenzierten Stelle ist abhängig von der Länge aller dazwischenliegenden Instruktionen.

Für den häufigen Fall von symbolischen Sprunginstruktionen kommt hinzu, daß diese, abhängig von der Distanz, als kurze oder lange Sprünge codiert werden können. Damit ist nicht nur der Wert der Codierung, sondern auch ihre Länge vom Kontext abhängig, was die Codierung anderer Sprunginstruktionen erschwert.

Es soll deshalb generell zwischen distanzabhängigen Sprüngen und "normalen" Instruktionen unterschieden werden. In diesem Kapitel werden nur die distanzabhängigen Sprünge behandelt.

Ein weiterer wesentlicher Punkt betrifft die Codierung der einzelnen Instruktionen: Sie kann ein oder mehrere Wörter lang sein. Die Codierung aller Instruktionen ist somit die

Konkatenation der Codierungen der einzelnen Instruktionen in ihrer Reihenfolge. Mit der Einschränkung auf distanzabhängige Sprunginstruktionen ergibt sich:

TYPE MachineInstruction = SEQUENCE OF Word
VAR m: ARRAY |prg| OF MachineInstruction

generate branching instructions:
∀0≤n<|prg|· (prg[n].i IS bra..ble) ∧ generate branch(prg[n].i,n)

Bei Sprüngen wird das Sprungziel als relative Adresse codiert, gegeben durch die Differenz zwischen der Zieladresse und der Adresse der Sprunginstruktion plus zwei. Damit haben Vorwärtssprünge eine positive Distanz und Rückwärtssprünge eine negative. Die distanzabhängigen Sprünge können, sofern die Distanz in ein Byte paßt, als kurze Sprünge mit insgesamt einem Wort Länge oder als lange Sprünge mit einem Wort Operationscode und einem Wort Sprungdistanz codiert werden. Zur Definition der Codierungen von Sprüngen sind drei Hilfsfunktionen notwendig.

adr(l) = size(index(l))
size(n) = ∑0≤i<n· |m[i]|
index(l)=n ⇔ prg[n].l=l falls n eindeutig

Adr(l) liefert die Adresse (in Worten) der Marke *l*, und *size(n)* die Adresse der *n*-ten Instruktion, d.h. die Größe des Codes bis (ausschließlich) *n*. *Index(l)* gibt die Nummer der mit *l* markierten Instruktion zurück. Falls *l* nicht deklariert ist, ist *index(l)* undefiniert und damit auch die gesamte Spezifikation. Einschränkend wird gefordert, daß eine Marke nur einmal deklariert werden darf.

generate branch(i,n):
(shortdisp´(i) ∧ m´[n]=shortbranch´(i)) ∨ m´[n]=longbranch´(i)

shortdisp(i): -128<disp(i.x.l)-2<0 ∨ 0<disp(i.x.l)-2<128

shortbranch(i): ‹C[i]+lowbyte(disp(i.x.l)-2)›

longbranch(i): ‹C[i]›°‹disp(i.x.l)-2)›

disp(target): 2*(adr(target)-size(n))

Disp(target) berechnet die Distanz zwischen dem Sprungziel *target* und dem Anfang der *n*-ten Instruktion. Der zweite Faktor von *disp* liefert die Sprungdistanz in Worten, so daß die Multiplikation mit zwei die erforderliche Distanz in Bytes ergibt.

Shortdisp(i) ist wahr falls die Distanz klein genug für einen kurzen Sprung ist. Bei der Codierung von distanzabhängigen Sprüngen bedeutet eine kurze Distanz von null, daß der Sprung lang ist und die eigentliche Distanz im nächsten Wort folgt. Deshalb ist in *shortdisp* null als kurze Distanz ausgeschlossen. Dies hat zur Konsequenz, daß ein Sprung auf die folgenden Instruktion lang sein muß.

Shortbranch(i) liefert die Codierung eines kurzen Sprunges *i* und *longbranch(i)* die Codierung von *i* als langer Sprung. Bei langen Sprüngen ist zu beachten, daß die Sprungdistanz im zweiten Wort steht und die Sprungdifferenz ab dort gezählt wird. Deshalb

muß die Länge des ersten Wortes von *disp* abgezogen werden.

Die gestrichenen Formen *shortdisp´*, *shortbranch´* und *longbranch´* in der Definition von *generate branch* sollen bedeuten, daß *m´* statt *m* zu Berechnung der Distanz verwendet werden soll. Dies bewirkt, daß *m´[n]* links und rechts des Gleichheitszeichens in *generate branch* auftritt. Damit ist *m´[n]* nicht direkt berechenbar und wir benötigen eine Idee zur Implementierung der Spezifikation.

C soll eine Tabelle sein, die für jeden Sprung ihre Codierung als Wort angibt.

C: ARRAY [bra..ble] OF Word

Eine Eigenschaft von der Spezifikation *generate branch* ist, daß sie indeterministisch ist. Falls die Bedingung für einen kurzen Sprung erfüllt ist, kann, muß aber nicht, ein kurzer Sprung generiert werden. Wir sind mit irgendeinem Resultat zufrieden, solange es korrekt ist. Dieser Indeterminismus gibt uns Freiheiten bei der Wahl der Implementierung, die im folgenden ausgenutzt werden sollen.

Bei Rückwärtssprüngen kann, sofern die Länge aller davorliegenden Instruktionen bekannt ist, die Sprungdistanz exakt berechnet werden, die günstigste Entscheidung bezüglich der Sprunglänge getroffen werden und der vollständige Code generiert werden.

Bei Vorwärtssprüngen ist eine einfache Strategie festzulegen, daß der Sprung lang ist, Code für den Sprung ohne die Sprungdistanz zu generieren und sich zu merken, daß an dieser Stelle noch die Distanz einzusetzen ist, sobald das Sprungziel bekannt wird.

Um die Qualität dieses Verfahrens bezüglich der Kompaktheit des Codes zu beurteilen, wäre es notwendig zu wissen, was das Verhältnis von Vorwärts- zu Rückwärtssprüngen und deren Verteilung bezüglich der Distanz ist. Über die Laufzeit läßt sich folgendes sagen. Bekanntlich sind immer nur die innersten Schleifen eines Programmes zeitkritisch. Schleifen bestehen immer aus einem Rückwärtssprung, und dieser wird bei unserem Verfahren optimal übersetzt.

Es ist somit zu zeigen:

(shortdisp´(i) ∧ m´[n]=shortbranch´(i)) ∨ m´[n]=longbranch´(i)
⊒ IF forwardref(i.x.l) THEN m´[n]=longbranch´(i)
ELSE IF shortdisp´(i) THEN m´[n]=shortbranch´(i) ELSE m´[n]=longbranch´(i) END
END

forwardref(l): index(l)>n

Der Beweis erfolgt in drei Schritten. Der erste besagt, daß Vorwärts- und Rückwärtssprünge getrennt betrachtet werden.

(shortdisp´(i) ∧ m´[n]=shortbranch´(i)) ∨ m´[n]=longbranch´(i) (1)
⇔ IF forwardref(i.x.l) THEN
(2)
(shortdisp´(i) ∧ m´[n]=shortbranch´(i)) ∨ m´[n]=longbranch´(i)
ELSE
(shortdisp´(i) ∧ m´[n]=shortbranch´(i)) ∨ m´[n]=longbranch´(i)
END

Die Korrektheit dieser Äquivalenz liefert Satz 1.

Satz 1. P $\Leftrightarrow$ IF b THEN P ELSE P END

Der Beweis dieses Satzes erfolgt durch eine Fallunterscheidung mit b. Der zweite Schritt soll festlegen, daß Vorwärtssprünge immer lang sein sollen. Dazu benötigen wir zwei Hilfssätze.

Satz 2. Q $\sqsubseteq$ P∨Q falls $\Delta P \Rightarrow \Delta Q$

Beweis.

Q $\sqsubseteq$ P∨Q

$\Leftrightarrow$ $\Delta(P \vee Q) \Rightarrow (\Delta Q \wedge Q \Rightarrow P \vee Q)$

$\Leftrightarrow$ $\Delta P \vee \Delta Q \Rightarrow \Delta Q$

$\Leftrightarrow$ TRUE

Satz 3. P $\sqsubseteq$ b∧P

Mit diesen Sätzen soll zu folgender Form übergegangen werden:

IF forwardref(i.x.l) THEN m´[n]=longbranch´(i) (3)
ELSE (shortdisp´(i)∧m´[n]=shortbranch´(i)) ∨ m´[n]=longbranch´(i)
END

Es ist also zu zeigen:

Δ(shortdisp´(i)∧m´[n]=shortbranch´(i)) $\Rightarrow$ Δ(m´[n]=longbranch´(i))

Leider gilt dies *nicht* immer. Es kann nämlich den Fall geben, daß durch die Wahl eines langen Sprunges statt eines kurzen das Program gerade so viel länger wird, daß ein anderer Sprung sogar nicht mehr als langer codiert werden kann. Wir müssen an dieser Stelle eine *Implementierungsrestriktion* einführen und annehmen, daß

Δ(m´[n]=longbranch´(i))

für alle n wahr ist. Es gilt somit

Δ(m´[n]=longbranch´(i)) $\Rightarrow$ (2)$\sqsubseteq$(3)

was gleichbedeutend ist mit:

(2) $\sqsubseteq$ (3)∧Δ(m´[n]=longbranch´(i))

Später werden wir weitere Implementierungsrestriktionen einführen müssen, wenn es darum geht, Sequenzen durch Felder zu implementieren. Es bleibt nur noch festzulegen, daß Rückwärtssprünge optimal generiert werden sollen.

IF forwardref(i.x.l) THEN m´[n]=longbranch´(i)
ELSE (shortdisp´(i)∧m´[n]=shortbranch´(i)) ∨ m´[n]=longbranch´(i)
END

$\sqsupseteq$ IF forwardref(i.x.l) THEN m´[n]=longbranch´(i)
ELSE IF shortdisp´(i) THEN m´[n]=shortbranch´(i) ELSE m´[n]=longbranch´(i) END
END

Die Begründung dieses Schrittes basiert auf Satz 4.

Satz 4. IF b THEN P ELSE Q END $\sqsubseteq$ (b∧P) ∨ Q falls (b∧ΔQ) $\Rightarrow$ ΔP

Beweis. Es gelte b. Dann ist unter der Bedingung $\Delta Q \Rightarrow \Delta P$ zu zeigen

P $\sqsubseteq$ P∨Q

was nach Satz 2 wahr ist. Es gelte $\neg b$. Dann lautet die (trivialerweise richtige) Aussage:

$Q \sqsubseteq Q$

Somit ist nachzuweisen:

$(\text{shortdisp}'(i) \wedge \Delta(m'[n]=\text{longbranch}'(i)) \Rightarrow \Delta(m'[n]=\text{shortbranch}'(i))$

Dies gilt wegen:

$\text{shortdisp}'(i) \quad \Rightarrow \quad \Delta(m'[n]=\text{shortbranch}'(i))$

Die bisherigen Transformationen haben die Spezifikation lediglich deterministisch gemacht und damit die Lösungsidee festgelegt, aber noch nicht realisiert. Diese besteht darin, beim sequentiellen Bearbeiten der Instruktionen die Rückwärtssprünge optimal zu übersetzen und die Vorwärtssprünge symbolisch stehen zu lassen. Sobald die Definition einer Marke erreicht wird, werden alle symbolischen Referenzen auf diese Marke durch die entsprechenden relativen Adressen ersetzt.

Das Resultat der bisherigen Transformationen werde *deterministic generate branching instructions* genannt, und die neue, sequentielle Form *sequential generate branching instructions*. Dann ist zu zeigen

deterministic generate branching instructions
⇔ sequential generate branching instructions

Zur Definition von *sequential generate branching instructions* wird *m* erweitert zu:

```
TYPE UnresolvedInstruction = SEQUENCE OF (Word | Label);
VAR m: ARRAY |prg| OF UnresolvedInstruction
```

Die Funktion *size(n)* bleibt definiert; sie macht keinen Unterschied von welchem Typ die Elemente sind.

sequential generate branching instructions:

```
FOR 0≤n<|prg| DO
   IF prg[n].l≠NIL THEN resolve(n) END;
   prg[n].i IS bra..ble ∧ generate symbolic branch(prg[n].i,n))
END
```

generate symbolic branch(i,n):

```
IF forwardref(i.x.l) THEN m[n]:=symbolicbranch(i)
ELSE IF shortdisp(i) THEN m[n]:=shortbranch(i) ELSE m[n]:=longbranch(i) END
END
```

symbolicbranch(i): ‹C[i]›°‹i.x.l›

resolve(N):

```
FOR 0≤n<N, 0≤p<|m[n]| DO
   IF m[n,p] IS Label ∧ index(m[n,p])=N THEN m[n,p]:=disp(m[n,p])-2*p END
END
```

Die Schleifenform von *sequential generate branching instructions* und *resolve* erfordert eine Invariante als Beweishilfsmittel.

Satz 5. Falls es eine Invariante *I(N)* gibt, so daß
(1) I(0) ⇔ SKIP
(2) I(N); P(N) ⇔ I(N+1) für alle N≥0
folgt
I(N) ⇔ FOR 0≤n<N DO P(n) END für alle N≥0

Der Beweis von Satz 5 erfolgt durch Induktion über *N*. Für die Äquivalenz von *deterministic generate branching instructions* und *sequential generate branching instructions* werden zwei Lemmata benötigt.

Lemma 1.

resolve(N) ⇔
(∀0≤n<N· ∀0≤p<|m[n]|·
IF m[n,p] IS Label ∧ index(m[n,p])=N THEN m´[n,p]=disp(m[n,p])-2*p
ELSE m´[n,p]=m[n,p] END) ∧
(∀N≤n<|prg|· m´[n]=m[n])

Zur Begründung wird Satz 5 genommen mit der Invariante *resolve(N)*. Beim Auflösen der Zuweisungsoperatoren in *resolve(N)* ist zu beachten, daß bei der Zuweisung an eine Feldkomponente die restlichen implizit gleich bleiben. Mit Lemma 1 gilt:

Lemma 2.

resolve(N) ⇒ |m[n]|=|m´[n]| für 0≤n<|prg|
⇒ size(n) = size´(n)
⇒ adr(l) = adr´(l)
⇒ disp(l) = disp´(l)
⇒ shortbranch(c,l) = shortbranch´(c,l) ∧
longbranch(c,l) = longbranch´(c,l) ∧
shortdisp(l) ⇔ shortdisp´(l)

Der Beweis der ersten Implikation erfolgt durch Fallunterscheidung mit der Bedingung *0≤n<N* bzw. *N≤n<|prg|* und, für den ersten dieser beiden Fälle, durch eine weitere Fallunterscheidung mit *m[n,p] IS Label ∧ index(m[n,p])=N*. Die Korrektheit der drei folgenden Implikationen folgt direkt aus den Definitionen von *size, adr* und *disp*.

Für den Beweis der Äquivalenz von *deterministic generate branching instructions* und *sequential generate branching instructions* wird wieder Satz 5 angewandt, mit der Invariante:

I(N):
(∀0≤n<N· prg[n].i IS bra..ble ∧ partial generate branch(prg[n].i,n)) ∧
(∀N≤n<|prg|· m´[n]=m[n])

partial generate branch(i,n):
IF forwardref(i.x.l) THEN
IF index(i.x.l)<N THEN m´[n]=longbranch´(i)
ELSE m´[n]=symbolicbranch´(i) END
ELSE IF shortdisp´(i) THEN m´[n]=shortbranch´(i)

```
                                    ELSE m´[n]=longbranch´(i) END
    END
```

Die erste Bedingung von Satz 5, die Induktionsbasis, ist trivialerweise richtig. Mit

```
P(N):
    IF prg[N].l≠NIL THEN resolve(N) END;
    generate symbolic branch(prg[N].i,N)
```

ist für die zweite Bedingung, dem Induktionsschritt, zu zeigen:

```
    I(N); P(N)                                                                    (4)
⇔   (∀0≤n<N· prg[n].i IS bra..ble ∧ partial generate branch(prg[n].i,n)) ∧
    (∀N≤n<|prg|· m´[n]=m[n]));
    IF prg[N].l≠NIL THEN resolve(N) END;
    prg[N].i IS bra..ble ∧ generate symbolic branch(prg[N].i, N)
⇔   I(N+1)
```

Das Zusammenfassen der ersten beiden Komponenten der sequentiellen Komposition, unter der Annahme *prg[n].l≠NIL* und unter Zuhilfenahme von Lemma 1, ergibt:

```
(∀0≤n<N·
    prg[n].i IS bra..ble ∧
    IF forwardref(prg[n].i.x.l) THEN
        IF index(prg[n].i.x.l)<N THEN m´[n]=longbranch´(prg[n].i)
                                 ELSE m´[n]=symbolicbranch´(prg[n].i) END
    ELSE IF shortdisp´(prg[n].i.x.l) THEN m´[n]=shortbranch´(prg[n].i)
                                     ELSE m´[n]=longbranch´(prg[n].i) END
    END) ∧
(∀N≤n<|prg|· m´[n]=m[n])
;
(∀0≤n<N· ∀0≤p<|m[n]|·
    IF m[n,p] IS Label ∧ index(m[n,p])=N THEN m´[n,p]=disp(m´[n,p])-2*p
                                        ELSE m´[n,p]=m[n,p] END) ∧
(∀N≤n<|prg|· m´[n]=m[n])
```

Die sequentielle Kompositon wird laut Definition aufgelöst durch das Umbenennen der Ausgabevariablen der ersten Komponente und der Eingabevariablen der zweiten Komponente in frische Variablen, hier gesternt geschrieben. Die nun durch Konjunktionen verbundenen Allquantoren können zusammengefaßt werden. Dies führt zu:

```
∃m*·
(∀0≤n<N·
    prg[n].i IS bra..ble ∧
    IF forwardref(prg[n].i.x.l) THEN
        IF index(prg[n].i.x.l)<N THEN m*[n]=longbranch*(prg[n].i)
                                 ELSE m*[n]=symbolicbranch*(prg[n].i) END
    ELSE IF shortdisp*(prg[n].i.x.l) THEN m*[n]=shortbranch*(prg[n].i)
```

ELSE m*[n]=longbranch*(prg[n].i) END
END ∧
(∀0≤p<|m*[n]|·
IF m*[n,p] IS Label ∧ index(m[n,p])=N THEN m´[n,p]=disp*(m*[n,p])-2*p
ELSE m´[n,p]=m*[n,p] END) ∧
(∀N≤n<|prg|· m*[n]=m[n] ∧ m´[n]=m*[n])

Nach Lemma 2 dürfen in der obigen Formel *shortdisp*, shortbranch*, longbranch** und *disp** in die entsprechende gestrichene Form umbenannt werden. Die Zuweisungen an *m*[n]* werden mit Hilfe der *Ein-Punkt-Regel* der Logik aufgelöst:

∃x· (P(x) ∧ x=e) ⇔ P(e)

Dazu muß der über *p* laufende Allquantor nach außen gezogen und eine Fallunterscheidung mit *m*[n,p] IS Label ∧ index(m[n,p])=N* gemacht werden. Dabei werden genau die symbolischen Sprünge in lange umgewandelt, die auf die *N*-te Instruktion gehen. Das Resultat lautet dann:

(∀0≤n<N· (5)
prg[n].i IS bra..ble ∧
IF forwardref(prg[n].i.x.l) THEN
IF index(prg[n].i.x.l) ≤ N THEN m´[n]=longbranch´(prg[n].i)
ELSE m´[n]=symbolicbranch´(prg[n].i) END
ELSE IF shortdisp´(prg[n].i.x.l) THEN m´[n]=shortbranch´(prg[n].i)
ELSE m´[n]=longbranch´(prg[n].i) END
END) ∧
(∀N≤n<|prg|· m´[n]=m[n])

Für den Fall *prg[n].l=NIL* erhalten wir obiges Resultat sofort aus (4) wegen

prg[n].l=NIL⇒ index(l)<N ⇔ index(l)≤N für alle l≠NIL

Es bleibt nach (4) zu zeigen, daß die Komposition des obigen Zwischenresultates mit *generate symbolic branch(prg[N].i, N)* gerade *I(N+1)* ergibt. Wegen

n=N ∧ forwardref(l) ⇒ index(l)>N

gilt aber

generate symbolic branch(prg[N].i,N) ⇔ partial generate branch(prg[N].i,N)

woraus durch das Auflösen der sequentiellen Kompositon folgt:

(5); generate symbolic branch(prg[N].i,N) ⇔ I(N+1)

Damit ist die Äquivalenz von *deterministic generate branching instructions* und *sequential generate branching instructions* gezeigt.

5. Erzeugung von Maschinencode

Die bisherige Spezifikation der Codegenerierung ist auf Sprunginstruktionen eingeschränkt, und die bisherige Implementierung behandelt ebenfalls nur die Sprunginstruktionen. In diesem Kapitel sollen Spezifikation und Implementierung um alle restlichen Instruktionen erweitert werden.

Die Spezifikation für die Übersetzung aller Instruktionen hat dieselbe Form wie die auf Sprunginstruktionen eingeschränke Spezifikation:

generate machine instructions:
$\forall 0 \le n < |prg| \cdot$ generate machine instruction(prg[n].i,n)

Die Codierung einer Nicht-Sprung-Instruktion ist weitgehend gegeben durch eine Funktion der Operation und der Operanden. Bei den zugelassenen Adressierungsarten kann das Resultat der Codierung einer Instruktion ein bis sieben Wörter lang sein. Die Operationen *addaw..subal, addib...aubil* und *moveb...movel* können unter gewissen Bedingungen als "schnelle" Operationen mit einem Wort Länge codiert werden. Die Spezifikation soll deshalb folgende Optimierungen berücksichtigen:

addaw #data,An	→	addqw #data,An	falls 1≤data≤8
...		...	-"-
subal #data,An	→	subql #data,An	-"-
addib #data,ea	→	addqb #data,ea	falls 1≤data≤8

...		...	-"-
subil #data,ea	→	subql #data,ea	-"-
moveb #data,ea	→	moveq #data,ea	falls -128≤data<128
movew #data,ea	→	-"-	-"-
movel #data,ea	→	-"-	-"-

Im folgenden werde *C* erweitert zu einer Tabelle, die die "normale" Codierung jeder Operation als ein Wort liefert, und *Q* sei eine Tabelle mit der Codierung der obigen Operationen als *Quick*-Operationen.

```
C: ARRAY [rtr..divlu] OF Word
Q: ARRAY [addaw..subil] OF Word


generate machine instruction(i,n):
   ( i IS rtr,rts,trapv ∧ m´[n]=C[i]
   ∨ i IS asl1,asr1,jmp,jsr,moveccrea,moveeaccr,pea,
         clrb,clrw,clrl,negb,negw,negl,notb,notw,notl,tstb,tstw,tstl,
         st,sf,shi,sls,scc,scs,sne,seq,svc,svs,spl,smi,sge,slt,sgt,sle ∧
      m´[n]=‹C[i]+mode(i.x)›°para(i.x)
   ∨ i IS bra..ble ∧ (shortdisp´(i) ∧ m´[n]=shortbranch´(i)) ∨ m´[n]=longbranch´(i)
   ∨ i IS extw,extl,swap,unlk ∧ m´[n]=‹C[i]+i.x.reg›
   ∨ i IS rtd ∧ m´[n]=‹C[i]›°‹i.x.n›
   ∨ i IS trap ∧ m´[n]=‹C[i]+i.x.n›
   ∨ i IS datab,dataw, datal ∧ m´[n]=im(i.x.n)
   ∨ i IS addb,addw,addl,andb,andw,andl,orb,orw,orl,subb,subw,subl,
         eorb,eorw,eorl,bchg,bclr,bset,btst ∧
      m´[n]=‹C[i] + i.x.reg SHL 9 + mode(i.y)›°para(i.y)
   ∨ i IS lea,mulsw,muluw,
         adddb,adddw,adddl,cmpb,cmpw,cmpl,subdb,subdw,subdl,
         anddb,anddw,anddl,chkw,chkl,divs,divu,ordb,ordw,ordl,
         cmpaw,cmpal ∧
      m´[n]=‹C[i] + i.y.reg SHL 9 + mode(i.x)›°para(i.x)
   ∨ i IS addaw,addal,subaw,subal ∧
      IF i.x IS immediate ∧ 1≤i.x.n≤8 THEN (*addqx, subqx #data,An *)
         m´[n]=‹Q[i]+(i.x.n MOD 8) SHL 9+i.y.reg›
      ELSE m´[n]=‹C[i]+i.y.reg SHL 9+mode(i.x)›° para(i.x)
      END
   ∨ i IS addib,addiw,addil,subib,subiw,subil ∧
      IF 1≤i.x.n≤8 THEN (*addqx, subqx #data,ea *)
         m´[n]=‹Q[i]+(i.x.n MOD 8) SHL 9+mode(i.y)›° para(i.y)
      ELSE m´[n]=‹C[i]+mode(i.y)›° im(i.x.n) °para(i.y)
      END
   ∨ i IS andib,andiw,andil,cmpib,cmpiw,cmpil,eorib,eoriw,eoril,orib,oriw,oril ∧
      m´[n]=‹C[i]+mode(i.y)›° im(i.x.n) °para(i.y)
   ∨ i IS bchgi,bclri,bseti,btsti,moveregeaw,moveregeal ∧
      m´[n]=‹C[i]+mode(i.y)›° word(i.x.n) °para(i.y)
```

```
∨ i IS moveareagw,moveeareagl ∧
   m´[n]=‹C[i]+mode(i.x)›° word(i.y.n) °para(i.x)
∨ i IS aslb,aslw,asll,asrb,asrw,asrl ∧ m´[n]=‹C[i]+i.x.reg SHL 9+i.y.reg›
∨ i IS aslib,asliw,aslil,asrib,asriw,asril ∧
   m´[n]=‹C[i]+(i.x.n MOD 8) SHL 9+i.y.reg›
∨ i IS dbra ∧ m´[n]=‹C[dbra]+i.x.reg›°‹disp´(i.y.l)-2›
∨ i IS link ∧
   IF i.y.n ∈ Word THEN m´[n] = ‹C[link]+i.x.reg›°‹i.y.n›
   ELSE m´[n] = ‹4808H+i.x.reg›°long(i.y.n)
   END
∨ i IS moveb,movew,movel ∧
   IF i.x IS immediate ∧ -80H≤i.x.n<80H ∧ i.y IS Dn THEN (*moveq #data,Dn *)
      m´[n]=‹7000H+i.y.reg SHL 9+lowbyte(i.x.n)›
   ELSE m´[n]=‹(mode(i.y) DIV 8) SHL 6+(mode(i.y) MOD 8)SHL 9+mode(i.x)›
      ° para(i.x) ° para(i.y)
   END
∨ i IS mulsl,mulul ∧ m´[n]=‹4C00H+mode(i.x)›°‹C[i]+i.y.reg SHL 12›°para(i.x)
∨ i IS divls,divlu ∧
   m´[n]=‹4C40H+mode(i.x)›°‹C[i]+i.z.reg SHL 12+i.y.reg›°para(i.x)
∨ i IS var ∧ m´[n]=‹› )
```

Die Instruktion *decrement and branch* wird wie alle anderen nicht distanzabhängigen Instruktionen codiert und deshalb hier statt bei den Sprunginstruktionen behandelt.

Der Operator SHL steht für das arithmetische Schieben nach links um *n* Bits.

$x \text{ SHL } n = x*2^n$

Long(l) ist eine Funktion, die ein Langwort in zwei Wörter umwandelt.

```
long(l):        ‹highword(l)›°‹lowword(l)›
```

Unmittelbare Operanden werden durch *im(n)* codiert. Die Länge des codierten Operanden ist abhängig von der Operation, bei Byte- und Wortoperationen ein Wort und bei Langwortoperationen zwei Wörter.

```
im(n):
   IF i IS datal,lea,adddl,cmpl,subdl,anddl,chkl,ordl,cmpal,addal,subal,addil,
      subil,andil,cmpil,eoril,oril,movel,mulsl,mulul,divls,divlu
   THEN im=long(n)
   ELSE im=‹n›
   END
```

Die Codierung einer *effektiven Adresse* erfolgt durch die Codierung der Adressierungsart in den unteren sechs Bits des Operationscodes und durch die Codierung des Operanden in den nachfolgenden Wörtern. Als einzige Instruktionen haben *moveb, movew* und *movel* zwei effektive Adressen; die Adressierungsart wird dann in den Bits 0 bis 5 und 6 bis 11 codiert.

```
mode(op):
   ( op IS Dn ∧ mode=op.n
   ∨ op IS An ∧ mode=8H+op
```

```
    ∨ op IS postinc ∧ mode = 18H+op.reg
    ∨ op IS predec ∧ mode= 20H+op.reg
    ∨ op IS indirect ∧ mode=10H+op.reg
    ∨ op IS offset ∧
       IF op.disp∈Word THEN mode=28H+op.reg
       ELSE mode=30H+op.reg
       END
    ∨ op IS index ∧  mode = 30H+op.reg)
    ∨ op IS direct  ∧
       IF varlab(op.l) THEN mode=39H (*absolute long*)
       ELSE mode=3AH   (*relative*)
       END
    ∨ op IS absolute  ∧ mode=39H   (*absolute long*)
    ∨ op IS immediate  ∧  mode=3CH)

para(op):
    ( op IS Dn,An,postinc,predec,indirect ∧ para=‹›
    ∨ op IS offset ∧
       IF op.disp∈Word THEN (*EA = (An)+d16 : no format extension*)
          para=‹op.disp›
       ELSE (*EA = (An)+bd : full format extension*) para=‹170H›°long(op.disp)
       END
    ∨ op IS index ∧
       IF -80H≤op.disp<80H THEN (*EA = (An)+(Xn)*scale+d8 : brief  extension*)
          para=‹op.indexreg SHL 12 + LOG(op.scale) SHL 9 + op.disp›
       ELSIF x.disp ∈ Word THEN (*EA = (An)+(Xn)*scale+bd16 : full extension *)
          para=‹op.indexreg SHL 12 + LOG(op.scale) SHL 9 + 120H›°‹op.disp›
       ELSE (* EA = (An)+(Xn)*scale+bd32 : full format extension *)
          para=‹op.indexreg SHL 12 + LOG(op.scale) SHL 9 + 130H›°long(op.disp)
       END
    ∨ op IS direct  ∧
       ( varlab(op.l)  ∧ varindex(op.l)<n  ∧ para=‹varadr(op.l)›
       ∨ codelab(op.l)  ∧ para=‹disp´(op.l)-2*pos(para)›)
    ∨ op IS absolute  ∧ para=long(op.n)
    ∨ op IS immediate ∧ para=im(op.n) )
```

Bei der direkten Adressierungsart wird unterschieden zwischen Referenzen auf Variablen und Referenzen auf Codemarken. Variablenreferenzen werden durch die Pseudoinstruktion *var* definert, beispielweise durch

```
var max: 4
```

Dabei ist *max* der Name der Variablen und 4 die Länge in Bytes. Alle Variablen werden mit 20000H beginnend hintereinander abgelegt. Die Adresse einer Variablen ergibt sich somit durch Summation der Längen aller vorhergehender Variablen plus 20000H:

```
varadr(l) = varsize(varindex(l))
varsize(n) = 20000H+∑0≤k<n ∧ prg[k].i IS var· prg[k].i.y
```

varindex(l)=n ⇔ prg[n].i IS var ∧ prg[n].i.x=l falls n eindeutig

Varlab(l) ist eine Bedingung, die nur erfüllt ist wenn *l* eine Variable referenziert. Einschränkend werden nur Rückwärtsreferenzen zugelassen. Analog dazu ist *codelab(l)* eine Bedingung, die nur erfüllt ist wenn *l* eine Variable referenziert.

varlab(l) ⇔ prg[k].i IS var ∧ prg[k].i.x=l für ein k
codelab(l) ⇔ prg[k].l=l für ein k

An dieser Stelle sei angemerkt, daß Variablenreferenzen zusätzlich Relokationsinformation benötigen. Dies ist das Thema des nächsten Kapitels.

Bei Referenzen auf Codemarken tritt die gleiche Problematik auf wie bei langen Sprunginstruktionen: Die Codierung hängt zusätzlich von der Position der Instruktion und gegebenenfalls von der Codierung aller nachfolgender oder vorhergehender Instruktionen ab. Zum Unterschied von Sprunginstruktionen kann die Referenz abhängig von der Instruktion an irgendeiner Postion innerhalb der Maschineninstruktion stehen. Deshalb wird der Operator *pos(s)* verwendet um die aktuelle Position innerhalb der Sequenz herauszufinden, in der *s* auftritt.

Die Implementierung der Nicht-Sprung-Instruktionen verläuft anlog zu der Implementierung der Sprunginstruktionen, nur daß die Spezifikation der Nicht-Sprung-Instruktionen nicht deterministisch gemacht werden muß sondern gleich sequentialisert werden kann. Es sei deshalb hier gleich das Resultat angegeben:

```
TYPE UnresolvedInstruction = SEQUENCE OF (Word | Label);
VAR m: ARRAY |prg| OF UnresolvedInstruction

sequential generate machine instructions:
   FOR 0≤n<|prg| DO
      IF prg[n].l≠NIL THEN resolve(n) END;
      generate symbolic normal instruction(prg[n].i, n))
   END

generate symbolic machine instruction(i,n):
   ( i IS rtr,rts,trapv ∧ m[n]:=C[i]
   ∨ i IS asl1,asr1,jmp,jsr,moveccrea,moveeaccr,pea,
         clrb,clrw,clrl,negb,negw,negl,notb,notw,notl,tstb,tstw,tstl,
         st,sf,shi,sls,scc,scs,sne,seq,svc,svs,spl,smi,sge,slt,sgt,sle ∧
      m[n]:=‹C[i]+mode(i.x)›°para(i.x)
   ∨ i IS bra..ble ∧
      IF forwardref(i.x.l) THEN m[n]:=symbolicbranch(i)
      ELSE
         IF shortdisp(i) THEN m[n]:=shortbranch(i) ELSE m[n]:=longbranch(i) END
      END
   ∨ i IS extw,extl,swap,unlk ∧ m[n]:=‹C[i]+i.x.reg›
   ∨ i IS rtd ∧ m[n]:=‹C[i]›°‹i.x.n›
   ∨ i IS trap ∧ m[n]:=‹C[i]+i.x.n›
   ∨ i IS datab,dataw, datal ∧ m[n]:=im(i.x.n)
   ∨ i IS addb,addw,addl,andb,andw,andl,orb,orw,orl,subb,subw,subl,
         eorb,eorw,eorl,bchg,bclr,bset,btst ∧
```

```
      m[n]:=‹C[i] + i.x.reg SHL 9 + mode(i.y)›°para(i.y)
   ∨ i IS lea,mulsw,muluw,
        adddb,adddw,adddl,cmpb,cmpw,cmpl,subdb,subdw,subdl,
        anddb,anddw,anddl,chkw,chkl,divs,divu,ordb,ordw,ordl,
        cmpaw,cmpal ∧
      m[n]:=‹C[i] + i.y.reg SHL 9 + mode(i.x)›°para(i.x)
   ∨ i IS addaw,addal,subaw,subal ∧
      IF i.x IS immediate ∧ 1≤i.x.n≤8 THEN (*addqx, subqx #data,An *)
        m[n]:=‹Q[i]+(i.x.n MOD 8) SHL 9+i.y.reg›
      ELSE m[n]:=‹C[i]+i.y.reg SHL 9+mode(i.x)›° para(i.x)
      END
   ∨ i IS addib,addiw,addil,subib,subiw,subil ∧
      IF 1≤i.x.n≤8 THEN (*addqx, subqx #data,ea *)
        m[n]:=‹Q[i]+(i.x.n MOD 8) SHL 9+mode(i.y)›° para(i.y)
      ELSE m[n]:=‹C[i]+mode(i.y)›° im(i.x.n) °para(i.y)
      END
   ∨ i IS andib,andiw,andil,cmpib,cmpiw,cmpil,eorib,eoriw,eoril,orib,oriw,oril ∧
      m[n]:=‹C[i]+mode(i.y)›° im(i.x.n) °para(i.y)
   ∨ i IS bchgi,bclri,bseti,btsti,moveregeaw,moveregeal ∧
      m[n]:=‹C[i]+mode(i.y)›° word(i.x.n) °para(i.y)
   ∨ i IS moveregw,moveeregl ∧
      m[n]:=‹C[i]+mode(i.x)›° word(i.y.n) °para(i.x)
   ∨ i IS aslb,aslw,asll,asrb,asrw,asrl ∧ m[n]:=‹C[i]+i.x.reg SHL 9+i.y.reg›
   ∨ i IS aslib,asliw,aslil,asrib,asriw,asril ∧
      m[n]:=‹C[i]+(i.x.n MOD 8) SHL 9+i.y.reg›
   ∨ i IS dbra ∧
      IF forwardref(i.y.l) THEN m[n]:=‹C[dbra]+i.x.reg›°‹i.y.l›
      ELSE m[n]:=‹C[dbra]+i.x.reg›°‹disp(i.y.l)-2›
      END)
   ∨ i IS link ∧
      IF i.y.n ∈ Word THEN m[n]: = ‹C[link]+i.x.reg›°‹i.y.n›
      ELSE m[n]: = ‹4808H+i.x.reg›°long(i.y.n)
      END
   ∨ i IS moveb,movew,movel ∧
      IF i.x IS immediate ∧ -80H≤i.x.n<80H ∧ i.y IS Dn THEN (*moveq #data,Dn *)
        m[n]:=‹7000H+i.y.reg SHL 9+lowbyte(i.x.n)›
      ELSE m[n]:=‹(mode(i.y) DIV 8) SHL 6+(mode(i.y) MOD 8)SHL 9+mode(i.x)›
         ° para(i.x) ° para(i.y)
      END
   ∨ i IS mulsl,mulul ∧ m[n]:=‹4C00H+mode(i.x)›°‹C[i]+i.y.reg SHL 12›°para(i.x)
   ∨ i IS divls,divlu ∧
      m[n]:=‹4C40H+mode(i.x)›°‹C[i]+i.z.reg SHL 12+i.y.reg›°para(i.x)
   ∨ i IS var ∧ m´[n]=‹› )

para(op):
   ( op IS Dn,An,postinc,predec,indirect ∧ para=‹›
   ∨ op IS offset ∧
      IF op.disp∈Word THEN (*EA = (An)+d16 : no format extension*)
```

```
        para=‹op.disp›
      ELSE (*EA = (An)+bd : full format extension*) para=‹170H›°long(op.disp)
      END
    ∨ op IS index ∧
      IF -80H≤op.disp<80H THEN (*EA = (An)+(Xn)*scale+d8 : brief  extension*)
        para=‹op.indexreg SHL 12 + LOG(op.scale) SHL 9 + op.disp›
      ELSIF x.disp ∈ Word THEN (*EA = (An)+(Xn)*scale+bd16 : full extension *)
        para=‹op.indexreg SHL 12 + LOG(op.scale) SHL 9 + 120H›°‹op.disp›
      ELSE (* EA = (An)+(Xn)*scale+bd32 : full format extension *)
        para=‹op.indexreg SHL 12 + LOG(op.scale) SHL 9 + 130H›°long(op.disp)
      END
    ∨ op IS direct  ∧
      ( varlab(op.l)  ∧ varindex(op.l)<n  ∧ para=‹varadr(op.l)›
      ∨ codelab(op.l)  ∧
        IF forwardref(op.l) THEN para=‹op.l›
        ELSE para=‹disp(op.l)-2*pos(para)›
        END)
    ∨ op IS absolute  ∧ para=long(op.n)
    ∨ op IS immediate ∧ para=im(op.n) )

  resolve(N):
    FOR 0≤n<N, 0≤p<|m[n]| DO
      IF m[n,p] IS Label ∧ index(m[n,p])=N THEN m[n,p]:=disp(m[n,p])-2*p END
    END
```

Gegenüber *generate machine instruction* hat sich geändert, daß *m´[n]=e* zu *m[n]:=e* wurde, daß die Generierung von *bra..ble* deterministisch gemancht wurde und daß bei *bra..ble, dbra* und bei *para* für den Fall von Codereferenzen jetzt zwischen Vorwärts- und Rückwärtsreferenzen unterschieden wird.

6. Generierung im Objektdateiformat

Die vollständige Übersetzung eines Assemblerprogrammes erfordert neben der Übersetzung der einzelnen Instruktionen auch die Generierung von zusätzlichen Informationen, wie einem Kopfteil und der Relokationsinformation. In diesem Kapitel soll diese letzte Erweiterung der Spezifikation vorgestellt und implementiert werden. Dabei erweist es sich nützlich, vorher eine weitere, vereinfachende Transformation des bisherigen Resultates durchzuführen.

Das *a.out* Objektdateiformat unterteilt eine Objektdatei in fünf Sektionen: den Kopfteil, den Programmcode, die Relokationsinfomation, die Symboltabelle und die Stringtabelle.

Der *Kopfteil* hat eine feste Länge und muß immer vorhanden sein. Er besteht aus folgenden Eintragungen:

```
VAR header: (machtype, magic: Word; text,data,bss,syms,entry,trsize,drsize: Long)
```

Die Felder sind in unserem Fall definiert durch:

```
header´.machtype = 2              (* machine type = MC68020 *)
header´.magic = 108H              (* magic number = nmagic *)
header´.text = 2* | flatten(m´) | (* size of text segment *)
header´.data = 0                  (* size of initialized data *)
header´.bss =  varsize( | prg | ) (* size of uninitialized data *)
header´.syms = 0;                 (* size of symbol table *)
header´.entry = 0                 (* entry point *)
```

```
header´.trsize = 8*|relocinfo´|        (* size of text relocation *)
header´.drsize = 0                     (* size of data relocation *)
```

Der *Programmcode*, in Unix Terminologie *Text* genannt, besteht aus der Sequenz der übersetzten Instruktionen:

```
flatten(m´)
```

Die *Relokationsinformation* ist notwendig für Referenzen auf Objekte, deren Adresse während der Übersetzung nicht bekannt ist. In unserem Fall ist dies erforderlich für Variablenreferenzen, denn Variablen werden absolut adressiert, und nicht relativ wie Sprungziele. Bei der Übersetzung wird eine relative Adresse eingesetzt, die dann beim Laden (evtl. mit anderen Moduln zusammen) in eine absolute umgewandelt wird.

Die Relokationsinformation für eine Variable besteht aus einem Zeiger in den Programmcode auf die Referenz und einer Typangabe, *Symbol* genannt, die in unserem Fall besagt, daß es sich um eine lange (vier Byte) Referenz auf eine lokale, nicht initialisierte Variable handelt.

```
VAR relocinfo: SET OF (adr,symbol: Long)
```

Die Reihenfolge der Relokationsinformation ist irrelevant, und so kann eine beliebige gewählt werden, die als *r* bezeichnet werden soll:

```
VAR r: SEQUENCE OF (adr,symbol: Long)
set(r´) = relocinfo´ ∧ |r´|=#relocinfo´
```

Die *Symboltabelle* und die *Stringtabelle* sind nur notwendig bei der Definition von globalen Bezeichnern und bei externen Referenzen. Beide sind deshalb hier leer. Die Struktur der Stringtabelle erfordert es aber, daß ihre Länge immer angegeben wird. Da das erste Langwort für die Länge selbst zur Stringtabelle gehört, hat diese mindestens die Länge vier.

```
VAR stringlength: Long
stringlength´=4
```

Zusammenfassend ist die Generierung der Objektdatei gegeben durch:

```
generate:
   object´=header´°flatten(m´)°r´°stringlength´
```

Dabei beinhaltet die obige Zuweisung eine implizite Typkonvertierung der Komponenten in Sequenzen aus Wörtern.

Der Rest dieses Kapitel ist der Generierung der Relokationsinformation gewidmet. Bevor diese spezifiziert wird, werden die Spezifikationen von *long, im* und *para* so geändert und eine neue Spezifikation *word* so definiert, daß folgende Transformationen der Zuweisungen an *m* in *sequential generate instruction* möglich sind (mit den alten Formen von *long, im* und *para* auf der linken Seite und den neuen auf der rechten):

```
m[n]:=s°‹w›        ⇔    m[n]:=s; word(w)
m[n]:=s°long(l)    ⇔    m[n]:=s; long(l)
```

m[n]:=s°im(k) ⇔ m[n]:=s; im(k)
m[n]:=s°para(op) ⇔ m[n]:=s; para(op)

Ein einfaches Auflösen der sequentiellen Komposition auf der rechten Seite ergibt, daß folgende, offensichtliche Definitionen diesen Forderungen genügen.

word(w): m[n]:=m[n]°‹w›

long(l): m[n]:=m[n]°‹highword(l)›°‹lowword(l)›

im(n):
```
IF i IS datal,lea,adddl,cmpl,subdl,anddl,chkl,ordl,cmpal,addal,subal,addil,
   subil,andil,cmpil,eoril,oril,movel,mulsl,mulul,divls,divlu
THEN long(n)
ELSE word(n)
END
```

para(op):
```
( op IS Dn,An,postinc,predec,indirect ∧ SKIP
∨ op IS offset ∧
   IF op.disp∈Word THEN (*EA = (An)+d16 : no format extension*)
      word(op.disp)
   ELSE (*EA = (An)+bd : full format extension*) word(170H); long(op.disp)
   END
∨ op IS index ∧
   IF -80H≤op.disp<80H THEN (*EA = (An)+(Xn)*scale+d8 : brief  extension*)
      word(op.indexreg SHL 12 + LOG(op.scale) SHL 9 + op.disp)
   ELSIF x.disp ∈ Word THEN (*EA = (An)+(Xn)*scale+bd16 : full extension *)
      word(op.indexreg SHL 12 + LOG(op.scale) SHL 9 + 120H); word(op.disp)
   ELSE (* EA = (An)+(Xn)*scale+bd32 : full format extension *)
      word(op.indexreg SHL 12 + LOG(op.scale) SHL 9 + 130H); long(op.disp)
   END
∨ op IS direct  ∧
   ( varlab(op.l)  ∧ varindex(op.l)<n  ∧ long(varadr(op.l))
   ∨ codelab(op.l)  ∧
      IF forwardref(op.l) THEN word(op.l)
      ELSE word(disp(op.l)-2*pos(para))
      END)
∨ op IS absolute  ∧ long(op.n)
∨ op IS immediate ∧ im(op.n) )
```

Zu beachten ist, daß *pos(para)* durch *|m[n]|* ersetzt wurde. Die oben angegebenen Transformationen werden so oft angewandt, bis jede Zuweisung an *m[n]* übergeführt wird in *m[n]:=‹›* gefolgt von Aufrufen der neu definierten Spezifikationen *word, long, im* und *para*. Die Zuweisung der leeren Sequenz an *m* kann mit der Regel

(P;Q) ∨ (P;R) ⇔ P;(Q∨R) Distributivität von ; bzgl. ∨

vor alle Alternativen gezogen werden. Das Resultat der Transformation von *sequential generate machine instructions* lautet dann:

```
m[n]:=‹›;
(  i IS rtr,rts,trapv ∧ word(C[i])
∨ i IS asl1,asr1,jmp,jsr,moveccrea,moveeaccr,pea,
       clrb,clrw,clrl,negb,negw,negl,notb,notw,notl,tstb,tstw,tstl,
       st,sf,shi,sls,scc,scs,sne,seq,svc,svs,spl,smi,sge,slt,sgt,sle ∧
    word(C[i]+mode(i.x)); para(i.x)
∨ i IS bra..ble ∧
    IF forwardref(i.x.l) THEN word(C[i]); word(i.x.l)
    ELSE IF -128<disp(i.x.l)-2 THEN word(C[i]+lowbyte(disp(i.x.l)-2)
                                  ELSE word(C[i]); word(disp(i.x.l)-2) END
    END
∨ i IS extw,extl,swap,unlk ∧ word(C[i]+i.x.reg)
∨ i IS rtd ∧ word(C[i]); word(i.x.n)
∨ i IS trap ∧ word(C[i]+i.x.n)
∨ i IS datab,dataw, datal ∧ im(i.x.n)
∨ i IS addb,addw,addl,andb,andw,andl,orb,orw,orl,subb,subw,subl,
       eorb,eorw,eorl,bchg,bclr,bset,btst ∧
    word(C[i] + i.x.reg SHL 9 + mode(i.y)); para(i.y)
∨ i IS lea,mulsw,muluw,
       adddb,adddw,adddl,cmpb,cmpw,cmpl,subdb,subdw,subdl,
       anddb,anddw,anddl,chkw,chkl,divs,divu,ordb,ordw,ordl,
       cmpaw,cmpal ∧
    word(C[i] + i.y.reg SHL 9 + mode(i.x)); para(i.x)
∨ i IS addaw,addal,subaw,subal ∧
    IF i.x IS immediate ∧ 1≤i.x.n≤8 THEN (*addqx, subqx #data,An *)
       word(Q[i]+(i.x.n MOD 8) SHL 9+i.y.reg)
    ELSE word(C[i]+i.y.reg SHL 9+mode(i.x)); para(i.x)
    END
∨ i IS addib,addiw,addil,subib,subiw,subil ∧
    IF 1≤i.x.n≤8 THEN  (*addqx, subqx #data,ea *)
       word(Q[i]+(i.x.n MOD 8) SHL 9+mode(i.y)); para(i.y)
    ELSE word(C[i]+mode(i.y)); im(i.x.n); para(i.y)
    END
∨ i IS andib,andiw,andil,cmpib,cmpiw,cmpil,eorib,eoriw,eoril,orib,oriw,oril ∧
    word(C[i]+mode(i.y)); im(i.x.n); para(i.y)
∨ i IS bchgi,bclri,bseti,btsti,moveregeaw,moveregeal ∧
    word(C[i]+mode(i.y)); word(i.x.n); para(i.y)
∨ i IS moveregw,moveearegl ∧
    word(C[i]+mode(i.x)); word(i.y.n); para(i.x)
∨ i IS aslb,aslw,asll,asrb,asrw,asrl ∧ word(C[i]+i.x.reg SHL 9+i.y.reg)
∨ i IS aslib,asliw,aslil,asrib,asriw,asril ∧
    word(C[i]+(i.x.n MOD 8) SHL 9+i.y.reg)
∨ i IS dbra ∧
    IF forwardref(i.x.l) THEN word(C[dbra]+i.x.reg); word(i.y.l)
    ELSE word(C[dbra]+i.x.reg); word(disp(i.y.l)-2)
    END
```

```
∨ i IS link ∧
   IF i.y.n ∈ Word THEN word(C[link]+i.x.reg); word(i.y.n)
   ELSE word(4808H+i.x.reg); long(i.y.n)
   END
∨ i IS moveb,movew,movel ∧
   IF i.x IS immediate ∧ -80H≤i.x.n<80H ∧ i.y IS Dn THEN (*moveq #data,Dn *)
      word(7000H+i.y.reg SHL 9+lowbyte(i.x.n))
   ELSE word((mode(i.y) DIV 8) SHL 6+(mode(i.y) MOD 8)SHL 9+mode(i.x));
      para(i.x); para(i.y)
   END
∨ i IS mulsl,mulul ∧
   word(4C00H+mode(i.x)); word(C[i]+i.y.reg SHL 12); para(i.x)
∨ i IS divls,divlu ∧
   word(4C40H+mode(i.x)); word(C[i]+i.z.reg SHL 12+i.y.reg); para(i.x)
∨ i IS var ∧ SKIP )
```

In der obigen Form sind zusätzlich *shortbranch, longbranch* und *symbolicbranch* durch Einsetzen der Definition eliminiert worden. Für die Vereinfachung von *shortdisp* wird verwendet, daß Rückwärtssprünge, und nur solche werden getestet ob sie kurz sind, eine negative Distanz haben:

¬forwardref(l) ⇒ disp(l)<0

Damit gilt:

¬forwardref(l) ⇒ (shortdisp(l) ⇔ -128<disp(l))

Nun zurück zur Generierung von Relokationsinformation. Diese wird erzeugt, indem *relocinfo* zunächst leer initialisiert wird und dann beim sequentiellen Bearbeiten der Instruktionen für jede Variablenreferenz eine Eintragung vorgenommen wird, d.h. *relocinfo* erhält für jede Variablenreferenz genau eine Eintragung.

```
sequential generate machine instructions and relocation information:
   relocinfo:={};
   FOR 0≤n<|prg| DO
      IF prg[n].l≠NIL THEN resolve(n) END;
      generate symbolic instruction and relocation(prg[n].i,n)
   END

generate symbolic instruction and relocation(i,n):
   ...

para(op):
   ( ...
   ∨ op IS direct ∧
      ( varlab(op.l) ∧ varindex(op.l)<n ∧
         relocinfo:=relocinfo∪{(size(n)+|m[n]|, 840H)} (*long, local, bss*);
         long(varadr(op.l))
      ∨ codelab(op.l) ∧ ...)
   ∨ ... )
```

Punkte symbolisieren gleichgebliebene Teile. Die obigen Transformationen von *para* usw. waren notwendig, um beim Erzeugen der Relokationsinformation Seiteneffekte der Funktion *para* in der alten Form zu vermeiden, denn *para* hätte die Codierung des Parameters als Sequenz von Wörtern und die Relokationsinformation zu liefern.

7. Transformation der Datenstrukturen

Ziel dieses Kapitels ist es, schwer implementierbare Datenstrukturen wie Sequenzen durch einfach implementierbare wie Felder oder Listen zu ersetzen, und dabei durch das Einführen weiterer, redundanter Datenstrukturen die Effizienz zu steigern. Die Transformationen betreffen die Menge *relocinfo* und die Sequenz *m*.

Die Technik, mit der Datentransformationen durchgeführt werden, ist zunächst durch Angabe einer Beziehung zwischen den bisherigen Datenstrukturen, auch *abstrakte* Daten genannt, und den einzuführenden, auch *konkrete* Daten genannt. Weiterhin ist es notwendig, zu jeder Operation über den abstrakten Daten eine entsprechende über den konkreten anzugeben. Folgender Satz besagt, daß das Ersetzen aller abstrakten Operationen in einer Spezifikation durch die entsprechenden konkreteren in eine äquivalente Spezifikation resultiert. Dieser Satz wurde ursprünglich von Hoare, He & Sanders [1986] formuliert.

Satz über Datentransformationen. Sei A ein abstrakter Datentyp bestehend aus einer Initialisierungsoperation A_0 und n weiteren Operationen $A_1...A_n$ über den Daten a, und sei C analog dazu ein abstrakter Datentyp bestehend aus einer Initialisierungsoperation C_0 und n weiteren Operationen $C_1...C_n$ über den Daten c. Sei $P(A_0,...,A_n)$ eine deterministische Spezifikation über den Operationen $A_0...A_n$. Es gilt

$$P(C_0,...,C_n) \sqsubseteq P(A_0,...,A_n)$$

falls es eine Relation I über c und a' gibt, so daß

$C_0 \sqsubseteq A_0; I$ und

$I; C_i \sqsubseteq A_i; I$ für alle $1 \le i < n$

Zu beachten ist, daß *a* nicht zum Kontext von C_i und *c* nicht zum Kontext von A_i gehören.

Transformation von *relocinfo*. Die Menge *relocinfo* soll implementiert werden durch ein Feld *reloctable* der Größe *maxReloc* und durch einen Index *rc* für die Anzahl der Eintragungen in *reloctable*.

```
VAR reloctable: ARRAY[0..macReloc-1] OF (adr,symbol: Long);
   rc: CARDINAL
```

Die Beziehung zwischen *relocinfo, reloctable´* und *rc´* lautet:

I : set(reloctable´[0..rc´-1])=relocinfo

Diese Beziehung gilt nur, wenn die Anzahl der Relokationseinträge nicht größer ist als *maxReloc*, d.h. wir haben es wieder mit einer Implementierungsrestriktion zu tun:

R: rc´ < maxReloc

Die beiden abstrakten Operationen sind:

relocinfo:={} und
relocinfo:=relocinfo∪{x}

Als die zugehörigen konkreten Operationen sollen genommen werden:

rc:=0 und
reloctable[rc]:=x; rc:=rc+1

Nach dem Transformationssatz ist zu zeigen:

$R \Rightarrow$ rc:=0 $\Leftrightarrow$ relocinfo:={}; I und

$R \Rightarrow$ I; reloctable[rc]:=x; rc:=rc+1 $\Leftrightarrow$ relocinfo:=relocinfo∪{x}; I

Für die rechte Seite der ersten Äquivalenz gilt:

relocinfo:={}; I
$\Leftrightarrow$ relocinfo:={}; set(reloctable´[0..rc´-1])=relocinfo
$\Leftrightarrow$ set(reloctable´[0..rc´-1])=relocinfo={}
$\sqsupseteq$ rc:=0

Für die rechte Seite der zweiten Äquivalenz wird umgeformt:

relocinfo:=relocinfo∪{x}; I
⇔ relocinfo:=relocinfo∪{x}; set(reloctable´[0..rc´-1])=relocinfo
⇔ set(reloctable´[0..rc´-1])=relocinfo∪{x}

Der Ausdruck *reloctable´[0..rc´-1]* ist dabei nur definiert, falls die Implementierungsrestriktion *rc´<maxReloc* erfüllt ist. Für die rechte Seite ergibt sich:

I; reloctable[rc]:=x; rc:=rc+1
⇔ set(reloctable´[0..rc´-1])=relocinfo; reloctable[rc]:=x; rc:=rc+1
⇔ set(reloctable´[0..rc´-1])=relocinfo ∧ reloctable´[rc]=x; rc:=rc+1
⊑ set(reloctable´[0..rc´])=relocinfo∪{x}; rc:=rc+1
⇔ set(reloctable´[0..rc´-1])=relocinfo∪{x}

In der Objektdatei wird aber nicht die Menge *relocinfo´* abgelegt, sondern eine beliebige Sequenz *r´* ihrer Elemente.

set(r´)=relocinfo

Es liegt nahe, für *r* gerade die ersten *rc´* Elemente aus *relocinfo* zu nehmen:

I ⇔ set(reloctable´[0..rc´-1])=relocinfo

Transformation von *m*. Das Ziel dieser Transformation ist es, zum einen die zweidimensionale Sequenz *m* flachzudrücken in eine eindimensional Sequenz und diese mit Hilfe einer Implementierungsrestriktion in der Länge zu begrenzen. Zum anderen soll für jede definierte Marke *l* ein Objekt *l^* eingeführt werden, in dem die Adresse der Marke und ihr Typ eingetragen werden, um so ohne Suchen zugreifen zu können. Der Typ *Label*, dessen Struktur bisher irrelevant war, wird deshalb festgelegt zu:

```
TYPE Label = POINTER TO LabelDef;
   LabelDef = (def,bss: BOOLEAN; adr: Long)
```

Zur Repräsentation von *m* wird verwendet:

```
VAR c: ARRAY [0..maxCode-1] OF (Word | Label);
   cs: CARDINAL
```

Außerdem soll ein Zähler *bs* für die Endadresse (Größe) aller bisher deklarierten Variablen eingeführt werden.

```
VAR bs: CARDINAL
```

Die Implementierungsrestriktion und die Beziehung zwischen *m, n, c´, cs´, bs´* und *l^´* (für alle Marken *l*) lauten:

R: |flatten(m´)| < maxCode
I: cs´=|flatten(m[0..n])| ∧ c´=flatten(m[0..n]) ∧
bs´=varsize(n) ∧
(∀codelab(l)· index(l)≤n ∧ l^´.def ∧ ¬l´.bss ∧ l^´.adr=adr(l)
∨ index(l)>n ∧ ¬l^´.def) ∧
(∀varlab(l)· varindex(l)≤n ∧ l^´.def ∧ l´.bss ∧ l^´.adr=varadr(l)

∨ varindex(l)>n ∧ ¬l^´.def)

Die Invariante *I* ist abhängig von dem Schleifenzähler *n*, der bei der FOR-Schleife implizit mitgezählt wird. Um Operaionen auf *n* zu transformieren, ist es notwendig, dies explizit zu machen.

```
        FOR 0≤n<|prg| DO
           IF prg[n].l≠NIL THEN resolve(n) END;
           m[n]:=‹›;
           (prg[n].i IS rtr,rts,trapv ∧ word(C[i])
           ∨ ...
           ∨prg[n].i IS var ∧ SKIP)
        END
⇔       n:=-1;
        WHILE n<|prg|-1 DO
           n:=n+1; m[n]:=‹›;
           IF prg[n].l≠NIL THEN resolve(n) END;
           (prg[n].i IS rtr,rts,trapv ∧ word(C[prg[n].i])
           ∨ ...
           ∨prg[n].i IS var ∧ SKIP)
        END
```

Neben dem Umschreiben der FOR-Schleife in eine WHILE-Schleife wurde auch die Zuweisung *m[n]:=‹›* vor *resolve(n)* gezogen. Dies ist erlaubt, weil *resolve(n)* unabhängig von *m[n]* ist und die Bedingung *prg[n].l≠NIL* ebenfalls. Es sind nun folgenden Transformationen durchzuführen:

```
n:=-1                  →   cs:=0; bs:=20000H; FOR lab(l) DO l^.def:=FALSE END
m[n]:=m[n]°‹w›         →   c[cs]:=w; cs:=cs+1
n:=n+1; m[n]:=‹›       →   IF prg[n].l≠NIL THEN prg[n].l^.def:=TRUE;
                             prg[n].l^.bss:=FALSE; prg[n].l^.adr:=cs
                           END;
                           IF prg[n].i IS var THEN prg[n].i.x^.def:=TRUE;
                              prg[n].i.x^.bss:=TRUE; prg[n].l^.adr:=bs;
                              bs:=bs+prg[n].i.y
                           END;

lab(l)  ⇔  codelab(l) ∨ varlab(l)
```

Für *n:=-1* wird mit Hilfe der Regel *x:=e;P(x)⇔P(e)* gerechnet:

```
        n:=-1; I
⇔       n:=-1;
        cs´=|flatten(m[0..n])| ∧ c´=flatten(m[0..n]) ∧
        bs´=varsize(n) ∧
        (∀codelab(l)· index(l)≤n∧l^´.def∧¬l´.bss ∧ ^´.adr=adr(l) ∨ index(l)>n∧¬l^´.def) ∧
        (∀varlab(l)·varindex(l)≤n∧l^´.def∧l´.bss∧l^´.adr=varadr(l) ∨ varindex(l)>n∧¬l^´.def)
⇔       cs´=0 ∧ bs:=20000H ∧ (∀codelab(l)· ¬l^´.def) ∧ (∀varlab(l)·¬l^´.def)
```

⇉ cs:=0; bs:=20000H; FOR lab(l) DO l^.def:=FALSE END

Für die zweite Regel ist zu zeigen *m[n]:=m[n]°‹w›; I ⇔ I; c[cs]:=x; cs:=cs+1* :

```
        m[n]:=m[n]°‹w›; I
⇔       m[n]:=m[n]°‹w›;
        cs´=|flatten(m[0..n])| ∧ c´=flatten(m[0..n]) ∧
        bs´=varsize(n) ∧ (∀codelab(l)· ... ) ∧ (∀varlab(l)· ... )
⇔       cs´=|flatten(m[0..n])°‹w›| ∧ c´=flatten(m[0..n])°‹w› ∧
        bs´=varsize(n) ∧ (∀codelab(l)· ... ) ∧ (∀varlab(l)· ... )
⇔       cs´=|flatten(m[0..n])| ∧ c´=flatten(m[0..n]) ∧
        bs´=varsize(n) ∧ (∀codelab(l)· ... ) ∧ (∀varlab(l)· ... ) ;
        c[cs]:=w; cs:=cs+1
⇔       I; c[cs]:=w; cs:=cs+1
```

Nachdem das Schema solcher Beweise klar ist, erlauben wir uns, bei der dritten Regel etwas informeller zu argumentieren. Wenn an *m[n]* eine neue, leere Sequenz durch *n:=n+1; m[n]:=‹›* angehängt wird, bleibt *flatten(m[0..n])* unverändert und so muß weder *cs* noch *c* verändert modifiziert werden. Für *bs´=varsize(n)* kann sich durch das Erhöhen von *n* eine Änderung ergeben, und zwar (laut Definition von *varsize*) für den Fall *prg]n].i IS var*. Der neue Wert von *bs* ergibt sich durch die Addition der Größe der durch *prg[n].i* deklarierten Variablen:

```
IF prg[n].i IS var THEN bs:=bs+prg[n].i.y END
```

Für das vierte Konjunkt der Invariante, den Allquantor über *codelab(l)*, tritt beim Erhöhen von *n* ein Änderung ein, falls es ein *l* mit *index(l)=n* gibt, d.h. falls *prg[n[.l≠NIL*. Für diesen Fall ist *l^´.def, ¬l´.bss* und *l^´.adr=adr(l)* zu erreichen:

```
IF prg[n].l≠NIL THEN
   prg[n].l^.def:=TRUE; prg[n].l^.bss:=FALSE; prg[n].l^.adr:=cs
END
```

Aus der Invariante *I* folgt für diesen Fall *cs´=size(n)*, womit die Zuweisung von *cs* statt *size(n)* gerechtfertigt wird. Die Argumentation für den Allquantor über *varlab(l)* verläuft parallel und führt zu:

```
IF prg[n].i IS var THEN
   prg[n].i.x.l^.def:=TRUE; prg[n].i.x.l^.bss:=TRUE; prg[n].i.x.l^.adr:=bs
END
```

Die obige Liste der Transformationen enthält nicht alle Operationen über *n*. Deshalb kann in der transformierten Form *n* nicht eliminiert werden und alle Operationen, die *n* verändern, müssen beibehalten werden. Das Resultat, wieder umgeschrieben auf die FOR-Schleife und ohne die Erzeugung von Relokationsinformation, lautet:

```
sequential generate:
   cs:=0; bs:=20000H; FOR lab(l) DO l^.def:=FALSE END
   FOR 0≤n<|prg| DO
```

```
      IF prg[n].l≠NIL THEN
        prg[n].l^.def:=TRUE; prg[n].l^.bss:=FALSE; prg[n].l^.adr:=cs
      END;
      IF prg[n].i IS var THEN
        prg[n].i.x.l^.def:=TRUE; prg[n].i.x.l^.bss:=TRUE; prg[n].i.x.l^.adr:=bs
      END;
      IF prg[n].l≠NIL THEN resolve(n) END;
      generate symbolic(prg[n].i,n)
    END

  generate symbolic(i,n):
    (i IS rtr,rts,trapv ∧ word(C[i])
    ∨ ...
    ∨ i IS bra..ble ∧
      IF ¬i.x.l^.def THEN word(C[i]); word(i.x.l)
      ELSE IF -128<disp(i.x.l)-2 THEN word(C[i]+lowbyte(disp(i.x.l)-2)
                                  ELSE word(C[i]); word(disp(i.x.l)) END
      END
    ∨ ...
    ∨ i IS dbra ∧
      IF ¬i.x.l^.def THEN word(C[dbra]+i.x.reg); word(i.y.l)
      ELSE word(C[dbra]+i.x.reg); word(disp(i.y.l))
      END
    ∨ ...
    ∨i IS var ∧ SKIP)

  para(op):
    ( ...
    ∨ op IS direct ∧
      IF op.l^.def ∧ op.l^.bss THEN long(op.l^.adr)
      ELSE
        IF ¬op.l^.def THEN word(op.l) ELSE word(2*(op.l^.adr-cs)) END
      END
    ∨ ... )

  resolve(n):
    FOR 0≤k<cs DO
      IF c[k]=prg[n].l THEN c[k]:=2*(cs-k)
    END

  disp(l):    2*(l^.adr-cs)

  word(w):    c[cs]:=w; cs:=cs+1
```

Neben den drei oben angegebenen Transformationen von Operatoren wurden eine Reihe weiterer Transformationen von Ausdrücken durchgeführt. Für die Fälle *bra..ble, dbra* und *direct* in *para* wurde *forwardref(l)* ersetzt durch ¬*l^.def*:

I ⇒ (forwardref(l) ⇔ ¬l^´.def)

Die Begründung folgt direkt aus der Definition von *forwardref(l)* als *index(l)>n* und der Invariante. Die alte Definition von *disp(l)* durch *2*(adr(l)-size(n))* wurde ebenfalls ersetzt:

I∧¬forwardref(l)∧|m[n]|=0 ⇒ 2*(adr(l)-size(n)) = 2*(l^´.adr-cs´)

I∧¬forwardref(l) ⇒ 2*(adr(l)-size(n)-|m[n]|) = 2*(l^´.adr-cs´)

Die erste Implikation wurde bei *bra..ble* und *dbra* angewandt und die zweite in *para*. Weil Variablen zuerst deklariert werden müssen bevor sie verwendet werden können, vereinfacht sich *varlab(l)* für den Fall *direct* in *para* zu:

I ⇒ (varlab(l) ⇔ l^´.def∧l^´.bss)

Es bietet sich an, einige Vereinfachungen des Resultates durchzuführen. Beispielsweise wird die Abfrage *prg[n].i IS var* zweimal kurz hintereinander ausgeführt und die Fallunterscheidungen für *bra..ble, dbra* und *direct* in *para* lassen sich vereinfachen. Bevor dies durchgeführt wird, soll noch die letzte Datentransformation vorgenommen werden.

Transformation von *c*. Die Idee bei dieser Transformation ist es, *c* aufzuspalten in ein Feld *code*, das nur die Maschinenwörter enthält und sich die symbolischen Vorwärtsreferenzen in einer linear verketteten Liste zu merken. Pro Marke wird eine solche Liste angelegt. Dazu werden die Markendefinition erweitert.

```
TYPE Label=POINTER TO LabelDef;
   ForwPtr=POINTER TO ForwRef;
   LabelDef=(def,bss: BOOLEAN; adr: Long; forw: ForwPtr);
   ForwRef=(loc: Long; next: ForwPtr);
VAR code: ARRAY [0..maxCode-1] OF Word;
```

Die Funktion *next(f)* gibt die Menge aller in der Liste mit Anker *f* eingetragenen Adressen von Vorwärtsreferenzen.

```
next(l) =   {}   falls f=NIL
            f^.loc ∪ next(f^.next)   falls f≠NIL
```

Die Invariante für *c, code´* und *l^´.forw* (für alle Marken *l*) ist gegeben durch:

I :(∀0≤n<cs· c[n] IS Word ⇒ code´[n]=c[n]) ∧
(∀codelab(l)· ¬l^.def ⇒ next(l^´.forw)={0≤n<cs | c[n]=l) ∧

Daraus ergeben sich folgende Transformationen (mit den alten Operationen auf der linken Seite und den neuen auf der rechten):

```
cs:=0                  →   FOR codelab(l) DO l^.forw:=NIL END
w∈Word∧c[cs]:=w        →   code[cs]:=w
l∈Label∧c[cs]:=l       →   enterforwardref(l)
resolve(n)             →   FOR k∈next(prg[n].l^.forw) DO c[k]:=2*(cs-k) END
```

enterforwardref(l):
NEW(f); f^.loc:=cs; f^.next:=l^.forw; l^.forw:=f; cs:=cs+1

Der Nachweis der Korrektheit dieser Transformationen erfordert die Definition der Semantik

von Zeiger-Operationen, die hier nicht gegeben wurde. Die Definitionen von Luckham & Suzuki [1979] oder London et al. [1979] ließen sich hier anwenden.

Das Resultat dieser Datentransformation wird wieder *sequential generate* genannt. Die vollständige Form inklusive der Relokationsinformation lautet:

```
TYPE Label=POINTER TO LabelDef;
   ForwPtr=POINTER TO ForwRef;
   LabelDef=(def,bss: BOOLEAN; adr: Long; forw: ForwPtr);
   ForwRef=(loc: Long; next: ForwPtr);

VAR code: ARRAY [0..maxCode-1] OF Word;
   reloctable: ARRAY[0..maxReloc-1] OF (adr,symbol: Long);
   cs,bs: Long;
   rc: CARDINAL;

sequential generate:
   cs:=0; bs:=20000H; rc:=0;
   FOR 0≤n<|prg| DO
      IF prg[n].l≠NIL THEN SetTarget(prg[n].l) END;
      IF prg[n].i IS var THEN SetSize(prg[n].i.x,prg[n].i.y) END;
      generate symbolic(prg[n].i)
   END

generate symbolic(i):
   ( i IS rtr,rts,trapv ∧ word(C[i])
   ∨ i IS asl1,asr1,jmp,jsr,moveccrea,moveeaccr,pea,
         clrb,clrw,clrl,negb,negw,negl,notb,notw,notl,tstb,tstw,tstl,
         st,sf,shi,sls,scc,scs,sne,seq,svc,svs,spl,smi,sge,slt,sgt,sle ∧
      word(C[i]+mode(i.x)); para(i.x)
   ∨ i IS bra..ble ∧
      IF i.x.l^.def THEN
         IF -128<disp(i.x.l)-2 THEN word(C[i]+lowbyte(disp(i.x.l)-2)
                               ELSE word(C[i]); word(disp(i.x.l)) END
      ELSE  word(C[i]); enterforwardref(i.x.l)
      END
   ∨ i IS extw,extl,swap,unlk ∧ word(C[i]+i.x.reg)
   ∨ i IS rtd ∧ word(C[i]); word(i.x.n)
   ∨ i IS trap ∧ word(C[i]+i.x.n)
   ∨ i IS datab,dataw, datal ∧ im(i.x.n)
   ∨ i IS addb,addw,addl,andb,andw,andl,orb,orw,orl,subb,subw,subl,
         eorb,eorw,eorl,bchg,bclr,bset,btst ∧
      word(C[i] + i.x.reg SHL 9 + mode(i.y)); para(i.y)
   ∨ i IS lea,mulsw,muluw,
         adddb,adddw,adddl,cmpb,cmpw,cmpl,subdb,subdw,subdl,
         anddb,anddw,anddl,chkw,chkl,divs,divu,ordb,ordw,ordl,
         cmpaw,cmpal ∧
      word(C[i] + i.y.reg SHL 9 + mode(i.x)); para(i.x)
   ∨ i IS addaw,addal,subaw,subal ∧
```

```
      IF i.x IS immediate ∧ 1≤i.x.n≤8 THEN (*addqx, subqx #data,An *)
        word(Q[i]+(i.x.n MOD 8) SHL 9+i.y.reg)
      ELSE word(C[i]+i.y.reg SHL 9+mode(i.x)); para(i.x)
      END
   ∨ i IS addib,addiw,addil,subib,subiw,subil ∧
      IF 1≤i.x.n≤8 THEN (*addqx, subqx #data,ea *)
        word(Q[i]+(i.x.n MOD 8) SHL 9+mode(i.y)); para(i.y)
      ELSE word(C[i]+mode(i.y)); im(i.x.n); para(i.y)
      END
   ∨ i IS andib,andiw,andil,cmpib,cmpiw,cmpil,eorib,eoriw,eoril,orib,oriw,oril ∧
      word(C[i]+mode(i.y)); im(i.x.n); para(i.y)
   ∨ i IS bchgi,bclri,bseti,btsti,moveregeaw,moveregeal ∧
      word(C[i]+mode(i.y)); word(i.x.n); para(i.y)
   ∨ i IS moverearegw,moveearegl ∧
      word(C[i]+mode(i.x)); word(i.y.n); para(i.x)
   ∨ i IS aslb,aslw,asll,asrb,asrw,asrl ∧ word(C[i]+i.x.reg SHL 9+i.y.reg)
   ∨ i IS aslib,asliw,aslil,asrib,asriw,asril ∧
      word(C[i]+(i.x.n MOD 8) SHL 9+i.y.reg)
   ∨ i IS dbra ∧
      IF i.x.l^.def THEN word(C[dbra]+i.x.reg); word(disp(i.y.l))
      ELSE word(C[dbra]+i.x.reg); enterforwardref(i.y.l)
      END
   ∨ i IS link ∧
      IF i.y.n ∈ Word THEN word(C[link]+i.x.reg); word(i.y.n)
      ELSE word(4808H+i.x.reg); long(i.y.n)
      END
   ∨ i IS moveb,movew,movel ∧
      IF i.x IS immediate ∧ -80H≤i.x.n<80H ∧ i.y IS Dn THEN (*moveq #data,Dn *)
        word(7000H+i.y.reg SHL 9+lowbyte(i.x.n))
      ELSE word((mode(i.y) DIV 8) SHL 6+(mode(i.y) MOD 8)SHL 9+mode(i.x));
        para(i.x); para(i.y)
      END
   ∨ i IS mulsl,mulul ∧
      word(4C00H+mode(i.x)); word(C[i]+i.y.reg SHL 12); para(i.x)
   ∨ i IS divls,divlu ∧
      word(4C40H+mode(i.x)); word(C[i]+i.z.reg SHL 12+i.y.reg); para(i.x)
   ∨ i IS var ∧ SKIP )
```

mode(op):

```
   ( op IS Dn ∧ mode=op.n
   ∨ op IS An ∧ mode=8H+op
   ∨ op IS postinc ∧ mode = 18H+op.reg
   ∨ op IS predec ∧ mode= 20H+op.reg
   ∨ op IS indirect ∧ mode=10H+op.reg
   ∨ op IS offset ∧
      IF op.disp∈Word THEN mode=28H+op.reg
      ELSE mode=30H+op.reg
      END
   ∨ op IS index ∧  mode = 30H+op.reg)
```

```
    ∨ op IS direct ∧
       IF op.l^.def ∧ op.l^.bss THEN mode=39H (*absolute long*)
       ELSE mode=3AH   (*relative*)
       END
    ∨ op IS absolute  ∧ mode=39H   (*absolute long*)
    ∨ op IS immediate  ∧  mode=3CH)


para(op):
    ( op IS Dn,An,postinc,predec,indirect ∧ SKIP
    ∨ op IS offset ∧
       IF op.disp∈Word THEN (*EA = (An)+d16 : no format extension*)
          word(op.disp)
       ELSE (*EA = (An)+bd : full format extension*) word(170H); long(op.disp)
       END
    ∨ op IS index ∧
       IF -80H≤op.disp<80H THEN (*EA = (An)+(Xn)*scale+d8 : brief  extension*)
          word(op.indexreg SHL 12 + LOG(op.scale) SHL 9 + op.disp)
       ELSIF x.disp ∈ Word THEN (*EA = (An)+(Xn)*scale+bd16 : full extension *)
          word(op.indexreg SHL 12 + LOG(op.scale) SHL 9 + 120H); word(op.disp)
       ELSE (* EA = (An)+(Xn)*scale+bd32 : full format extension *)
          word(op.indexreg SHL 12 + LOG(op.scale) SHL 9 + 130H); long(op.disp)
       END
    ∨ op IS direct  ∧
       IF op.l^.def THEN
          IF op.l^.bss THEN
             reloctable[rc].adr:=2*cs; reloctable[rc].sym:=840H; rc:=rc+1
             long(op.l^.adr)
          ELSE word(2*(op.l^.adr-cs))
          END
       ELSE enterforwardref(op.l)
       END
    ∨ op IS absolute  ∧ long(op.n)
    ∨ op IS immediate ∧ im(op.n) )

SetTarget(l):
    l^.def:=TRUE; l^.bss:=FALSE; l^.adr:=cs;
    FOR loc∈next(l^.forw) DO code[loc]:=2*(cs-loc) END

SetSize(l,size):
    l^.def:=TRUE; l^.bss:=TRUE; l^.adr:=bs; bs:=bs+size

disp(l):      2*(l^.adr-cs)

enterforwardref(l):
    VAR f: ForwPtr;
    NEW(f); f^.loc:=cs; f^.next:=l^.forw; l^.forw:=f; cs:=cs+1
```

word(n):
    code[cs]:=lowword(n); cs:=cs+1

long(n):
    code[cs]:=highword(n); cs:=cs+1; code[cs]:=lowword(n); cs:=cs+1

8. Implementierung in Modula

In diesem Kapitel wird die letzte Transformation des Generators, die alle nicht in Modula vorhandenen Elemente eliminiert, vollzogen. Dies betrifft unter anderem die bisher verwendeten Varianten und die Fallunterscheidungen durch Disjunktionen. Zusätzlich wird in den undefinierten Fällen eine Fehlerbehandlung spezifiziert. Außerdem wird der Assemblierer vervollständigt durch den Symbolentschlüssler und den Zerteiler.

Die bisher eingesetzte Notation für Varianten bei den Operanden soll durch den in Modula üblichen Verbund mit Variantenteil ersetzt werden. Dies führt zur Definition der Operanden als:

```
TYPE AddrMode=(Dn,An,indirect,postinc,predec,offset,index,direct,absolute,
       immediate);
   Operand=RECORD
         CASE m:AddrMode OF
            Dn,An,indirect,postinc,precdec,offset,index:
               reg: [0..7];
               (*offset and index mode only:*)
               disp: INTEGER;
               (*index mode only:*)
               indexreg: [0..15]; (*0..7: Dn, 8..15: An*)
               scale: [1..8]; (*1,2,4,8*) |
            direct: l: Label |
            absolute,immediate: n: INTEGER
```

```
    END
  END
```

und zu folgenden Transformationen der Typtests:

op IS An	→	op.m=An
...		...
op IS immediate	→	op.m=immediate

Der Zugriff durch die Selektion mit einem Punkt bleibt unverändert. Weiterhin muß der Typ *Instruction* transformiert werden. Statt einer Variantenkonstruktion wie bei den Operanden wird ein Quadrupel bestehend aus

```
o: Operation
xop,yop,zop: Operand
```

verwendet, wobei die Operation als Aufzählungtyp definiert wird:

```
TYPE Operation=(rtr, ... , divlu)
```

Die sich ergebenden Transformationen lauten

i IS rtr	→	o=rtr
...		...
i IS divlu	→	o=divlu
i.x	→	xop
i.y	→	yop
i.z	→	zop

Dies bedeutet, daß die Transformation von *generate instruction* vier Parameter hat, eine für die Operation und drei für die Operanden. Um aber die Übergabe von eventuell bedeutungslosen Operanden zu vermeiden, wird lediglich die Operation übergeben und die Operanden als globale Variablen gehalten. Die so erhaltene Prozedur soll kurz *gen* genannt werden. Die Schnittstelle des Generators läßt sich dann definieren zu:

```
DEFINITION MODULE Generator;

 TYPE Keyword =
  (*registers*)
   (d0,d1,d2,d3,d4,d5,d6,d7,a0,a1,a2,a3,a4,a5,a6,a7,
  (*no operand instructions*)
   rtr,rts,trapv,
  (*one operand instructions*)
   asl1,asr1,jmp,jsr,moveccrea,moveeaccr,pea,
   clrb,clrw,clrl,negb,negw,negl,notb,notw,notl,tstb,tstw,tstl,
   st,sf,shi,sls,scc,scs,sne,seq,svc,svs,spl,smi,sge,slt,sgt,sle,
   bra,brs,bhi,bls,bcc,bcs,bne,beq,bvc,bvs,bpl,bmi,bge,blt,bgt,ble,
   extw,extl,swap,unlk,rtd,trap,datab,dataw,datal,
  (*two operand instructions*)
   addb,addw,addl,andb,andw,andl,orb,orw,orl,subb,subw,subl,
    eorb,eorw,eorl,bchg,bclr,bset,btst,
   lea,mulsw,muluw,
    adddb,adddw,adddl,cmpb,cmpw,cmpl,subdb,subdw,subdl,
    anddb,anddw,anddl,chkw,chkl,divs,divu,ordb,ordw,ordl,
    cmpaw,cmpal,
```

```
    addaw,addal,subaw,subal,
    addib,addiw,addil,subib,subiw,subil,
    andib,andiw,andil,cmpib,cmpiw,cmpil,eorib,eoriw,eoril,orib,oriw,oril,
    bchgi,bclri,bseti,btsti,moveregeaw,moveregeal,
    moveearegw,moveearegl,
    aslb,aslw,asll,asrb,asrw,asrl,
    aslib,asliw,aslil,asrib,asriw,asril,
    dbra,link,moveb,movew,movel,
    mulsl,mulul,
  (*three operand instructions*)
    divls,divlu,
  (*pseudoinstruction*)
    var);
  Operation = [rtr..divlu];

  AddrMode = (Dn,An,indirect,postinc,predec,offset,index,direct,absolute,
    immediate);
  Label;
  Operand = RECORD
      CASE m:AddrMode OF
        Dn,An,indirect,postinc,predec,offset,index:
          reg: [0..7];
          (*offset and index mode only:*)
          disp: INTEGER;
          (*index mode only:*)
          indexreg: [0..15]; (*0..7:Dn, 8..15:An*)
          scale: [1..8]; (*1,2,4,8*) |
        direct: l: Label |
        absolute,immediate: n: INTEGER
      END
    END;

  VAR xop,yop,zop: Operand;

  PROCEDURE gen (o: Operation);

  PROCEDURE NewLabel (): Label;
  PROCEDURE SetTarget (codelab: Label);
  PROCEDURE SetSize (varlab: Label; size: CARDINAL);
  PROCEDURE defined (l: Label): BOOLEAN;
  PROCEDURE referred (l: Label): BOOLEAN;

  PROCEDURE InitGenerator;
  PROCEDURE CloseGenerator;
END Generator.
```

Der Typ *Operation* wurde als Unterbereichstyp von *Keyword* definiert, um die Implementierung des Symbolentschlüsslers zu vereinfachen. Die Funktionen *defined* und *referred* werden vom Symbolentschlüssler und Zerteiler zur Fehlerbehandlung verwendet.
Zur Implementierung von *gen* ist im Rumpf von *generate instruction* die Fallunterscheidung durch Disjunktion zu transformieren. Es bietet sich mit der Definition der Operation als Aufzählungstyp an, eine CASE-Anweisung zu verwenden. Sie ist formal definiert durch

$$\text{CASE e OF } c_1\text{: } P_1 \mid \ldots \mid c_n\text{: } P_n \text{ ELSE } P_0 \text{ END}$$
$$\Leftrightarrow \quad (e{=}c_1 \wedge P_1) \vee \ldots \vee (e{=}c_n \wedge P_n) \vee (e{\neq}c_1 \wedge \ldots \wedge e{\neq}c_n \wedge P_0)$$

falls alle c_i verschieden sind. Der ELSE-Teil darf weggelassen werden. Als Abkürzung wird zugelassen

$$\ldots \mid k_1..k_n\text{:}P \mid \ldots$$

mit der Bedeutung

... ∨ ((e=k_1 ∨ ... ∨ e=k_n) ∧ P) ∨ ...

Weiterhin müssen die Operatoren für das arithmetische Schieben und das Logarithmieren (in *para*) ersetzt werden. Wegen $x\ SHL\ n = x*2^n$ führt dies zu

```
CONST SH6=64; SH8=256; SH9=512; SH12=4096
x SHL 6  =  x*SH6
...
x SHL 12  =  x*SH12
```

und mit $LOG(2^n) = n$ gilt für *n=1,2,4,8*:

```
    x:=LOG(n) SHL 9
⇔   CASE n OF
      1: x:=0 |
      2: x:=200H |
      3: x:=400H |
      8: x:=600H
    END
```

Weitere Details sind folgender Implementierung zu entnehmen. Zu beachten ist, daß sich für undefinierte Eingaben, d.h. solche, die die Spezifikation oder die Implementierungsrestriktionen nicht erfüllen, die Implementierung beliebig verhalten darf; hier wurde gewählt, in solchen Fällen mit einer Fehlermeldung abzubrechen.

```
IMPLEMENTATION MODULE Generator; (*Emil Sekerinski, 24.3.89*)

FROM Storage IMPORT ALLOCATE;
FROM StdIO IMPORT WriteBlock;
FROM Scanner IMPORT Error;

 CONST maxCode=8000H;
  maxReloc=3000;
  SH6=64;
  SH8=256;
  SH9=512;
  SH12=4096;

 TYPE Word=SHORTCARD; (*16 bits*)
  Long=INTEGER; (*32 bits*)

  Label=POINTER TO LabelDef;
  ForwPtr=POINTER TO ForwRef;
  LabelDef=RECORD
     def: BOOLEAN;
     bss: BOOLEAN; (*bss or code label*)
     adr: Long;
     forw: ForwPtr
   END;
  ForwRef=RECORD
     loc: Long;
     next: ForwPtr
   END;

 VAR code: ARRAY [0..maxCode-1] OF Word;
  reloctable: ARRAY [0..maxReloc-1] OF
   RECORD adr,symbol: Long END;
  cs,bs: Long; (*code size, bss size*)
  rc: CARDINAL; (*relocation count*)
```

```
  C: ARRAY Operation OF Word;
  Q: ARRAY [addaw..subil] OF Word;

PROCEDURE InitCodes;
  VAR n: CARDINAL;
BEGIN
(*no operand instructions*)
  C[rtr]:=4E77H; C[rts]:=4E75H; C[trapv]:=4E76H;
(*one operand instructions*)
  C[asl1]:=0E1C0H; C[asr1]:=0E0C0H; C[jmp]:=4EC0H; C[jsr]:=4E80H;
  C[moveccrea]:=42C0H; C[moveeaccr]:=44C0H; C[pea]:=4840H;
  C[clrb]:=4200H; C[clrw]:=4240H; C[clrl]:=4280H;
  C[negb]:=4400H; C[negw]:=4440H; C[negl]:=4480H;
  C[notb]:=4600H; C[notw]:=4640H; C[notl]:=4680H;
  C[tstb]:=4A00H; C[tstw]:=4A40H; C[tstl]:=4A80H;
  FOR n:=0 TO 15 DO
    C[VAL(Keyword,n+ORD(st))] := 50C0H+n*SH8;
    C[VAL(Keyword,n+ORD(bra))] := 6000H+n*SH8
  END;
  C[extw]:=4880H; C[extl]:=48C0H; C[swap]:=4840H;
  C[unlk]:=4E58H; C[rtd]:=4E74H;  C[trap]:=4E40H;
(*two operand instructions*)
  C[addb]:=0D100H; C[addw]:=0D140H; C[addl]:=0D180H;
  C[andb]:=0C100H; C[andw]:=0C140H; C[andl]:=0C180H;
  C[eorb]:=0B100H; C[eorw]:=0B140H; C[eorl]:=0B180H;
  C[orb] := 8100H; C[orw] := 8140H; C[orl] := 8180H;
  C[subb]:= 9100H; C[subw]:= 9140H; C[subl]:= 9180H;
  C[bchg]:=140H; C[bclr]:=180H; C[bset]:=1C0H; C[btst]:=100H;
  C[adddb]:=0D000H; C[adddw]:=0D040H; C[adddl]:=0D080H;
  C[anddb]:=0C000H; C[anddw]:=0C040H; C[anddl]:=0C080H;
  C[cmpb] :=0B000H; C[cmpw] :=0B040H; C[cmpl] :=0B080H;
  C[cmpaw]:=0B0C0H; C[cmpal]:=0B1C0H;
  C[chkw] := 4180H; C[chkl] := 4100H;
  C[divs] := 81C0H; C[divu] := 80C0H; C[lea]  := 41C0H;
  C[mulsw]:=0C1C0H; C[muluw]:=0C0C0H;
  C[ordb] := 8000H; C[ordw] := 8040H; C[ordl] := 8080H;
  C[subdb]:= 9000H; C[subdw]:= 9040H; C[subdl]:= 9080H;
  C[addaw]:=0D0C0H; C[addal]:=0D1C0H;
  C[subaw]:= 90C0H; C[subal]:= 91C0H;
  C[addib]:= 600H; C[addiw]:= 640H; C[addil]:= 680H;
  C[subib]:= 400H; C[subiw]:= 440H; C[subil]:= 480H;
  C[andib]:= 200H; C[andiw]:= 240H; C[andil]:= 280H;
  C[cmpib]:=0C00H; C[cmpiw]:=0C40H; C[cmpil]:=0C80H;
  C[eorib]:=0A00H; C[eoriw]:=0A40H; C[eoril]:=0A80H;
  C[orib] :=   0H; C[oriw] :=  40H; C[oril] :=  80H;
  C[aslb] :=0E120H; C[aslw] :=0E160H; C[asll] :=0E1A0H;
  C[asrb] :=0E020H; C[asrw] :=0E060H; C[asrl] :=0E0A0H;
  C[aslib]:=0E100H; C[asliw]:=0E140H; C[aslil]:=0E180H;
  C[asrib]:=0E000H; C[asriw]:=0E040H; C[asril]:=0E080H;
  C[bchgi]:=840H; C[bclri]:=880H; C[bseti]:=8C0H; C[btsti]:=800H;
  C[moveregeaw]:=4880H; C[moveregeal]:=48C0H;
  C[moveearegw]:=4C80H; C[moveearegl]:=4CC0H;
  C[dbra]:=56C8H; C[link]:=4E50H;
  C[moveb]:=1000H; C[movew]:=3000H; C[movel]:=2000H;
  C[mulsl]:=800H; C[mulul]:=0H;
(*three operand instructions*)
  C[divls]:=800H; C[divlu]:=0H;
(*quick instructions*)
  Q[addaw]:=5048H; Q[addal]:=5088H;
  Q[subaw]:=5148H; Q[subal]:=5188H;
  Q[addib]:=5000H; Q[addiw]:=5040H; Q[addil]:=5080H;
  Q[subib]:=5100H; Q[subiw]:=5140H; Q[subil]:=5180H
END InitCodes;

PROCEDURE mode (op: Operand): Word;
BEGIN
  CASE op.m OF
    Dn: RETURN op.reg |
```

```
    An: RETURN 8H+op.reg |
    postinc: RETURN 18H+op.reg |
    predec: RETURN 20H+op.reg |
    indirect: RETURN 10H+op.reg |
    offset:
      IF (-8000H<=op.disp) & (op.disp<8000H) THEN RETURN 28H+op.reg
      ELSE RETURN 30H+op.reg
      END |
    index: RETURN 30H+op.reg |
    direct:
      IF op.l^.def & op.l^.bss THEN RETURN 39H (*absolute long*)
      ELSE RETURN 3AH (*relative*)
      END |
    absolute: RETURN 39H (*absolute long*) |
    immediate: RETURN 3CH
  END
END mode;

PROCEDURE lowbyte (n:Word): Word;
  TYPE W= RECORD hi,lo: CHAR END;
  VAR w: W;
BEGIN w:=W(n);
  RETURN ORD(w.lo)
END lowbyte;

PROCEDURE word (n: Long);
BEGIN code[cs]:=n; INC(cs)
END word;

PROCEDURE long (n: Long);
  TYPE L= RECORD hi,lo: Word END;
  VAR l: L;
BEGIN l:=L(n);
  code[cs]:=l.hi; INC(cs);
  code[cs]:=l.lo; INC(cs)
END long;


PROCEDURE enterforwardref (l: Label);
  VAR f: ForwPtr;
BEGIN NEW(f);
  WITH f^ DO loc:=cs; next:=l^.forw END;
  l^.forw:=f; INC(cs)
END enterforwardref;

PROCEDURE gen (o: Operation);

  PROCEDURE im (n: Long);
  BEGIN
    CASE o OF
      datal,lea,adddl,cmpl,subdl,anddl,chkl,ordl,cmpal,addal,subal,addil,
      subil,andil,cmpil,eoril,oril,movel,mulsl,mulul,divls,divlu: long(n)
    ELSE word(n)
    END
  END im;

  PROCEDURE para (op: Operand);
    VAR e: Word; f: ForwPtr;
  BEGIN
    CASE op.m OF
      Dn,An,postinc,predec,indirect: |
      offset:
        IF (-8000H<=op.disp) & (op.disp<8000H) THEN word(op.disp)
        ELSE word(170H); long(op.disp)
        END |
      index:
        CASE op.scale OF
          1: e:= 800H |
```

```
        2: e:= 0A00H |
        4: e:= 0C00H |
        8: e:= 0E00H
       END;
       IF (-80H<=op.disp) & (op.disp<80H) THEN
         word(op.indexreg*SH12 + e + lowbyte(op.disp))
       ELSIF (-8000H<=op.disp) & (op.disp<8000H) THEN
         word(op.indexreg*SH12 + e + 120H); word(op.disp)
       ELSE word(op.indexreg*SH12 + e + 130H); long(op.disp)
       END |
     direct:
       IF op.l^.def THEN
         IF op.l^.bss THEN (*generate relocation info*)
           IF rc=maxReloc THEN Error("relocation table overflow") END;
           reloctable[rc].adr:=2*cs; reloctable[rc].symbol:=840H; INC(rc);
           long(op.l^.adr)
         ELSE word(2*(op.l^.adr-cs))
         END
       ELSE enterforwardref(op.l)
       END |
     absolute: long(op.n) |
     immediate: im(op.n)
   END
 END para;

 VAR disp: INTEGER;
   m: Word;
BEGIN (*gen*)
 IF cs>maxCode-7 THEN Error("code table overflow") END;
 CASE o OF
 (*no operand instructions*)
   rtr..trapv: word(C[o]) |
 (*one operand instructions*)
   asl1..sle: word(C[o]+mode(xop)); para(xop) |
   bra..ble:
     IF xop.l^.def THEN disp:=2*(xop.l^.adr-cs-1);
       IF -128<disp THEN word(C[o]+lowbyte(disp))
       ELSE word(C[o]); word(disp)
       END
     ELSE word(C[o]); enterforwardref(xop.l)
     END |
   extw..unlk: word(C[o]+xop.reg) |
   rtd: word(C[rtd]); word(xop.n) |
   trap: word(C[trap]+VAL(Word,xop.n)) |
   datab..datal: im(xop.n)|
 (*two operand instructions*)
   addb..btst: word(C[o] + xop.reg*SH9 + mode(yop)); para(yop) |
   lea..cmpal: word(C[o] + yop.reg*SH9 + mode(xop)); para(xop) |
   addaw..subal:
     IF (xop.m=immediate) & (1<=xop.n) & (xop.n<=8) THEN
       (*addq,subq #data,An*)
       word(Q[o] + (VAL(Word,xop.n) MOD 8)*SH9 + yop.reg)
     ELSE word(C[o] + yop.reg*SH9 +  mode(xop)); para(xop)
     END |
   addib..subil:
     IF (1<=xop.n) & (xop.n<=8) THEN (*addq,subq #data,ea*)
       word(Q[o] + (VAL(Word,xop.n) MOD 8)*SH9 + mode(yop)); para(yop)
     ELSE word(C[o]+mode(yop)); im(xop.n); para(yop)
     END |
   andib..oril: word(C[o]+mode(yop)); im(xop.n); para(yop) |
   bchgi..moveregeal: word(C[o]+mode(yop)); word(xop.n); para(yop) |
   moveearegw,moveearegl: word(C[o]+mode(xop)); word(yop.n); para(xop) |
   aslb..asrl: word(C[o] + xop.reg*SH9 + yop.reg) |
   aslib..asril: word(C[o] + (xop.reg MOD 8)*SH9 + yop.reg) |
   dbra:
     IF yop.l^.def THEN
       word(C[dbra]+xop.reg); word(2*(yop.l^.adr-cs))
     ELSE word(C[dbra]+xop.reg); enterforwardref(yop.l)
```

```
    END |
   link:
    IF (-8000H<=yop.n) & (yop.n<8000H) THEN
      word(C[link]+xop.reg); word(yop.n)
    ELSE word(4808H+xop.reg); long(yop.n)
    END |
   moveb..movel:
    IF (xop.m=immediate) & (-80H<=xop.n) & (xop.n<80H) & (yop.m=Dn) THEN
      word(7000H + yop.reg*SH9 + lowbyte(xop.n)) (*moveq #data,Dn*)
    ELSE m:=mode(yop); word(C[o]+(m DIV 8)*SH6+(m MOD 8)*SH9+mode(xop));
      para(xop); para(yop)
    END |
   mulsl,mulul:
    word(4C00H+mode(xop)); word(C[o] + yop.reg*SH12); para(xop) |
  (*three operand instructions*)
   divls,divlu:
    word(4C40H+mode(xop)); word(C[o] + zop.reg*SH12 + yop.reg);
    para(xop)
 END
END gen;

PROCEDURE NewLabel (): Label;
 VAR l: Label;
BEGIN NEW(l); l^.def:=FALSE; l^.forw:=NIL; RETURN l
END NewLabel;

PROCEDURE SetTarget (codelab: Label);
 VAR f: ForwPtr;
BEGIN
 WITH codelab^ DO def:=TRUE; bss:=FALSE; adr:=cs; f:=forw;
  WHILE f#NIL DO code[f^.loc]:=2*(cs-f^.loc); f:=f^.next END
 END
END SetTarget;

PROCEDURE SetSize (varlab: Label; size: CARDINAL);
BEGIN
 WITH varlab^ DO def:=TRUE; bss:=TRUE; adr:=bs END;
 INC(bs,size)
END SetSize;

PROCEDURE defined (l: Label): BOOLEAN;
BEGIN RETURN l^.def
END defined;

PROCEDURE referred (l: Label): BOOLEAN;
BEGIN RETURN l^.forw#NIL
END referred;

PROCEDURE InitGenerator;
BEGIN InitCodes; cs:=0; bs:=20000H; rc:=0
END InitGenerator;

PROCEDURE CloseGenerator;
 VAR header: RECORD
      machtype,magic: Word;
      text,data,bss,syms,entry,trsize,drsize: Long
    END;
  stringlength: Long;
BEGIN
 header.machtype:=2;    (*machine type = MC68020*)
 header.magic:=108H;    (*magic number = nmagic*)
 header.text:=2*cs;     (*size of text segment in bytes*)
 header.data:=0;        (*size of initialized data*)
 header.bss:=bs;        (*size of uninitialized data*)
 header.syms:=0;        (*size of symbol table*)
 header.entry:=0;       (*entry point*)
 header.trsize:=8*rc;   (*size of text relocation*)
 header.drsize:=0;      (*size of data relocation*)
```

```
  WriteBlock(header,SIZE(header));
  WriteBlock(code,2*cs);
  WriteBlock(reloctable,8*rc);
  stringlength:=4;
  WriteBlock(stringlength,4)
 END CloseGenerator;

END Generator.
```

Bei der Entwicklung des Symbolentschlüsslers und Zerteilers wurden Standardtechniken des Übersetzerbaus eingesetzt, auf die hier nicht näher eingegangen werden soll. Die Implementierung des Symbolentschlüsslers ist angelehnt an Wirth [1983].

```
DEFINITION MODULE Scanner;

FROM Generator IMPORT Keyword,Label;

 TYPE Symbol=(eof,eol,immed,lbr,rbr,times,plus,comma,minus,number,
    colon,lbl,keywrd);

 VAR sym: Symbol;     (*last symbol read*)
  num: CARDINAL;   (*number if sym=number*)
  lab: Label;      (*label if sym=lbl*)
  key: Keyword;    (*keyword if sym=keywrd*)

 PROCEDURE GetSym; (*get next symbol; results: sym,num,lab,key*)
 PROCEDURE Error(s: ARRAY OF CHAR);
  (*write s on error output and terminate*)
 PROCEDURE InitScanner;
 PROCEDURE CloseScanner;
END Scanner.


IMPLEMENTATION MODULE Scanner; (*Emil Sekerinski, 16.2.89*)

FROM StdIO IMPORT Read,WriteBlock,WriteError;
FROM Generator IMPORT Keyword,Label,NewLabel,defined;

 CONST EOL=12C;
  P=997; (*prime, hash table size*)
  bufLen=10000;
  free=bufLen;

 VAR
  ch: CHAR; (*last character read*)
  last: CARDINAL; (*index of end of last identifier read*)
  buf: ARRAY [0..bufLen] OF CHAR;
  hash: ARRAY [0..P-1] OF RECORD
     b: CARDINAL; (*string index*)
     s: Symbol; (*lbl or keywrd*)
     l: Label;
     k: Keyword
    END;

 PROCEDURE EnterKW (s: ARRAY OF CHAR; key: Keyword);
  VAR i,p,h,d: CARDINAL;
 BEGIN i:=0; p:=last; h:=0;
  (*enter s in buf and compute hash index h*)
  WHILE i<=HIGH(s) DO buf[last]:=s[i];
   h:=(256*h+ORD(s[i])) MOD P; INC(last); INC(i)
  END;
  buf[last]:=0C; INC(last); d:=1;
  WHILE hash[h].b#free DO (*collision*) h:=h+d; d:=d+2;
   IF h>=P THEN h:=h-P END;
  END;
  WITH hash[h] DO b:=p; s:=keywrd; k:=key END
```

```
END EnterKW;

PROCEDURE InitKeys;
BEGIN
  EnterKW("d0",d0); EnterKW("d1",d1); EnterKW("d2",d2);
  EnterKW("d3",d3); EnterKW("d4",d4); EnterKW("d5",d5);
  EnterKW("d6",d6); EnterKW("d7",d7);
  EnterKW("a0",a0); EnterKW("a1",a1); EnterKW("a2",a2);
  EnterKW("a3",a3); EnterKW("a4",a4); EnterKW("a5",a5);
  EnterKW("a6",a6); EnterKW("a7",a7);
  EnterKW("rtr",rtr); EnterKW("rts",rts); EnterKW("trapv",trapv);
  EnterKW("asl1",asl1); EnterKW("asr1",asr1);
  EnterKW("jmp",jmp);   EnterKW("jsr",jsr);
  EnterKW("moveccrea",moveccrea); EnterKW("moveeaccr",moveeaccr);
  EnterKW("pea",pea);
  EnterKW("clrb",clrb); EnterKW("clrw",clrw); EnterKW("clrl",clrl);
  EnterKW("negb",negb); EnterKW("negw",negw); EnterKW("negl",negl);
  EnterKW("notb",notb); EnterKW("notw",notw); EnterKW("notl",notl);
  EnterKW("tstb",tstb); EnterKW("tstw",tstw); EnterKW("tstl",tstl);
  EnterKW("st",st);   EnterKW("sf",sf);   EnterKW("shi",shi);
  EnterKW("sls",sls); EnterKW("scc",scc); EnterKW("scs",scs);
  EnterKW("sne",sne); EnterKW("seq",seq); EnterKW("svc",svc);
  EnterKW("svs",svs); EnterKW("spl",spl); EnterKW("smi",smi);
  EnterKW("sge",sge); EnterKW("slt",slt); EnterKW("sgt",sgt);
  EnterKW("sle",sle);
  EnterKW("bra",bra); EnterKW("brs",brs); EnterKW("bhi",bhi);
  EnterKW("bls",bls); EnterKW("bcc",bcc); EnterKW("bcs",bcs);
  EnterKW("bne",bne); EnterKW("beq",beq); EnterKW("bvc",bvc);
  EnterKW("bvs",bvs); EnterKW("bpl",bpl); EnterKW("bmi",bmi);
  EnterKW("bge",bge); EnterKW("blt",blt); EnterKW("bgt",bgt);
  EnterKW("ble",ble);
  EnterKW("extw",extw); EnterKW("extl",extl);
  EnterKW("swap",swap); EnterKW("unlk",unlk);
  EnterKW("rtd",rtd);   EnterKW("trap",trap);
  EnterKW("datab",datab); EnterKW("dataw",dataw); EnterKW("datal",datal);
  EnterKW("addb",addb); EnterKW("addw",addw); EnterKW("addl",addl);
  EnterKW("andb",andb); EnterKW("andw",andw); EnterKW("andl",andl);
  EnterKW("orb",orb);   EnterKW("orw",orw); EnterKW("orl",orl);
  EnterKW("subb",subb); EnterKW("subw",subw); EnterKW("subl",subl);
  EnterKW("eorb",eorb); EnterKW("eorw",eorw); EnterKW("eorl",eorl);
  EnterKW("bchg",bchg); EnterKW("bclr",bclr);
  EnterKW("bset",bset); EnterKW("btst",btst);
  EnterKW("lea",lea);     EnterKW("mulsw",mulsw); EnterKW("muluw",muluw);
  EnterKW("adddb",adddb); EnterKW("adddw",adddw); EnterKW("adddl",adddl);
  EnterKW("cmpb",cmpb);   EnterKW("cmpw",cmpw);   EnterKW("cmpl",cmpl);
  EnterKW("subdb",subdb); EnterKW("subdw",subdw); EnterKW("subdl",subdl);
  EnterKW("anddb",anddb); EnterKW("anddw",anddw); EnterKW("anddl",anddl);
  EnterKW("chkw",chkw);   EnterKW("chkl",chkl);
  EnterKW("divs",divs);   EnterKW("divu",divu);
  EnterKW("ordb",ordb);   EnterKW("ordw",ordw);   EnterKW("ordl",ordl);
  EnterKW("cmpaw",cmpaw); EnterKW("cmpal",cmpal);
  EnterKW("addaw",addaw); EnterKW("addal",addal);
  EnterKW("subaw",subaw); EnterKW("subal",subal);
  EnterKW("addib",addib); EnterKW("addiw",addiw); EnterKW("addil",addil);
  EnterKW("subib",subib); EnterKW("subiw",subiw); EnterKW("subil",subil);
  EnterKW("andib",andib); EnterKW("andiw",andiw); EnterKW("andil",andil);
  EnterKW("cmpib",cmpib); EnterKW("cmpiw",cmpiw); EnterKW("cmpil",cmpil);
  EnterKW("eorib",eorib); EnterKW("eoriw",eoriw); EnterKW("eoril",eoril);
  EnterKW("orib",orib);   EnterKW("oriw",oriw);   EnterKW("oril",oril);
  EnterKW("bchgi",bchgi); EnterKW("bclri",bclri);
  EnterKW("bseti",bseti); EnterKW("btsti",btsti);
  EnterKW("moveregeaw",moveregeaw); EnterKW("moveregeal",moveregeal);
  EnterKW("moveearegw",moveearegw); EnterKW("moveearegl",moveearegl);
  EnterKW("aslb",aslb); EnterKW("aslw",aslw); EnterKW("asll",asll);
  EnterKW("asrb",asrb); EnterKW("asrw",asrw); EnterKW("asrl",asrl);
  EnterKW("aslib",aslib); EnterKW("asliw",asliw); EnterKW("aslil",aslil);
  EnterKW("asrib",asrib); EnterKW("asriw",asriw); EnterKW("asril",asril);
  EnterKW("dbra",dbra);   EnterKW("link",link);
```

```
  EnterKW("moveb",moveb); EnterKW("movew",movew); EnterKW("movel",movel);
  EnterKW("mulsl",mulsl); EnterKW("mulul",mulul);
  EnterKW("divls",divls); EnterKW("divlu",divlu);
  EnterKW("var",var)
END InitKeys;

PROCEDURE Number;
  VAR d: CARDINAL;
BEGIN sym:=number; num:=0;
  REPEAT d:=ORD(ch)-60B;
    IF (MAX(CARDINAL)-d) DIV 10 < num THEN Error("constant too large") END;
    num:=10*num+d; Read(ch)
  UNTIL (ch<"0") OR (ch>"9")
END Number;

PROCEDURE Identifier;
  VAR id,i,j,h,d: CARDINAL;
BEGIN id:=last; h:=0;
  (*enter identifier in buf and compute hash index h*)
  REPEAT
    IF id>= bufLen THEN Error("string buffer overflow") END;
    buf[id]:=ch; h:=(256*h+ORD(ch)) MOD P;
    INC(id); Read(ch)
  UNTIL (ch<"0") OR ("9"<ch) & (CAP(ch)<"A") OR ("Z"<CAP(ch));
  buf[id]:=0C; INC(id); d:=1;
  LOOP (*search identifier in hash table*)
    IF hash[h].b=free THEN (*new entry*)
      WITH hash[h] DO b:=last; s:=lbl; l:=NewLabel(); lab:=l END;
      sym:=lbl; (*keep identifier*) last:=id; RETURN
    END;
    (*compare hash[h].key with last*)
    i:=hash[h].b; j:=last;
    WHILE (buf[i]=buf[j]) & (buf[i]#0C) DO INC(i); INC(j) END;
    IF buf[i]=buf[j] THEN (*found*)
      WITH hash[h] DO sym:=s; lab:=l; key:=k END;
      RETURN
    ELSE (*collision*) h:=h+d; d:=d+2;
      IF h>=P THEN h:=h-P END;
      IF d=P THEN Error ("hash table overflow") END
    END
  END
END Identifier;

PROCEDURE GetSym;
BEGIN (*ignore all control characters except EOL*)
  WHILE (ch<=' ') & (ch#EOL) DO Read(ch) END;
  CASE ch OF
    177C: sym:=eof |
    EOL: sym:=eol; Read(ch) |
    "#": sym:=immed; Read(ch) |
    "(": sym:=lbr; Read(ch) |
    ")": sym:=rbr; Read(ch) |
    "*": sym:=times; Read(ch) |
    "+": sym:=plus; Read(ch) |
    ",": sym:=comma; Read(ch) |
    "-": sym:=minus; Read(ch) |
    "0".."9": Number |
    ":": sym:=colon; Read(ch) |
    ";": REPEAT Read(ch) UNTIL (ch=EOL) OR (ch=177C);
        IF ch=EOL THEN sym:=eol; Read(ch) ELSE sym:=eof END |
    "a".."z","A".."Z": Identifier
  ELSE Error("illegal character")
  END
END GetSym;

PROCEDURE Error(s: ARRAY OF CHAR);
BEGIN WriteError(s); HALT
END Error;
```

```
 PROCEDURE InitScanner;
  VAR n: CARDINAL;
 BEGIN ch:=" "; last:=0;
  FOR n:=0 TO P-1 DO hash[n].b:=free END;
  InitKeys
 END InitScanner;

 PROCEDURE CloseScanner;
  VAR n: CARDINAL;
 BEGIN
  FOR n:=0 TO P-1 DO
   IF (hash[n].s=lbl) & ~defined(hash[n].l) THEN
    Error("undefined symbol")
   END
  END
 END CloseScanner;

END Scanner.
```

Der Symbolentschlüssler macht seine Ein- und Ausgabe durch das Modul *StdIO*, das die angegebenen Prozeduren durch Unix-Aufrufe realisiert und zusätzlich eine Pufferung der Eingabe vornimmt.

```
DEFINITION MODULE StdIO;

FROM SYSTEM IMPORT BYTE;

 PROCEDURE Read (VAR ch: CHAR);
  (*read ch from standard input; set ch=177C at end of file*)

 PROCEDURE WriteBlock (VAR b: ARRAY OF BYTE; n: CARDINAL);
  (*write n bytes of b to standard output*)

 PROCEDURE WriteError (s: ARRAY OF CHAR);
  (*write s to standard error output*)

END StdIO.
```

Die Struktur des Zerteilers ist nach den Regeln des rekursiven Abstiegs bestimmt durch die Syntax der Assemblersprache, wie im Anhang beschrieben. Da der Zerteiler von nichts Darüberliegendem aufgerufen wird, ist er als Programmodul *Assembler* realisiert.

```
MODULE Assembler; (*Emil Sekerinski, 23.3.89*)

FROM Scanner IMPORT Symbol,sym,num,lab,key,
 GetSym,Error,InitScanner,CloseScanner;
FROM Generator IMPORT Keyword,AddrMode,Label,Operand,xop,yop,zop,
 gen,NewLabel,SetTarget,SetSize,defined,referred,
 InitGenerator,CloseGenerator;


 TYPE Modes = SET OF AddrMode;

 VAR
  mx: ARRAY [asl1..mulul] OF Modes; (*mode of first operand*)
  my: ARRAY [addb..mulul] OF Modes; (*mode of second operand*)

 PROCEDURE InitModes;
  VAR i: Keyword;
 BEGIN
  mx[asl1]:=Modes{indirect,postinc,predec,offset,index,direct,absolute};
  mx[asr1]:=Modes{indirect,postinc,predec,offset,index,direct,absolute};
```

```
mx[jmp]:=Modes{indirect,offset,index,direct,absolute};
mx[jsr]:=Modes{indirect,offset,index,direct,absolute};
mx[moveccrea]:=Modes{Dn,indirect,postinc,predec,offset,index,direct,absolute};
mx[moveeaccr]:=Modes{Dn,indirect,postinc,predec,offset,index,direct,absolute,immediate};
mx[pea]:=Modes{indirect,offset,index,direct,absolute};
FOR i:=clrb TO sle DO
  mx[i]:=Modes{Dn,indirect,postinc,predec,offset,index,direct,absolute}
END;
FOR i:=bra TO ble DO mx[i]:=Modes{direct} END;
mx[extw]:=Modes{Dn};
mx[extl]:=Modes{Dn};
mx[swap]:=Modes{Dn};
mx[unlk]:=Modes{An};
mx[rtd]:=Modes{immediate};
mx[trap]:=Modes{immediate};
mx[datab]:=Modes{immediate};
mx[dataw]:=Modes{immediate};
mx[datal]:=Modes{immediate};
FOR i:=addb TO subl DO
  mx[i]:=Modes{Dn};
  my[i]:=Modes{indirect,postinc,predec,offset,index,direct,absolute}
END;
FOR i:=eorb TO btst DO
  mx[i]:=Modes{Dn};
  my[i]:=Modes{Dn,indirect,postinc,predec,offset,index,direct,absolute}
END;
mx[lea]:=Modes{indirect,offset,index,direct,absolute};
my[lea]:=Modes{An};
mx[mulsw]:=Modes{Dn,indirect,postinc,predec,offset,index,direct,absolute,immediate};
my[mulsw]:=Modes{Dn};
mx[muluw]:=Modes{Dn,indirect,postinc,predec,offset,index,direct,absolute,immediate};
my[muluw]:=Modes{Dn};
FOR i:=adddb TO subdl DO
  mx[i]:=Modes{Dn,An,indirect,postinc,predec,offset,index,direct,absolute,immediate};
  my[i]:=Modes{Dn}
END;
FOR i:=anddb TO ordl DO
  mx[i]:=Modes{Dn,indirect,postinc,predec,offset,index,direct,absolute,immediate};
  my[i]:=Modes{Dn}
END;
FOR i:=cmpaw TO subal DO
  mx[i]:=Modes{Dn,An,indirect,postinc,predec,offset,index,direct,absolute,immediate};
  my[i]:=Modes{An}
END;
FOR i:=addib TO btsti DO
  mx[i]:=Modes{immediate};
  my[i]:=Modes{Dn,indirect,postinc,predec,offset,index,direct,absolute}
END;
mx[moveregeaw]:=Modes{immediate};
my[moveregeaw]:=Modes{indirect,predec,offset,index,direct,absolute};
mx[moveregeal]:=Modes{immediate};
my[moveregeal]:=Modes{indirect,predec,offset,index,direct,absolute};
mx[moveearegw]:=Modes{indirect,postinc,offset,index,direct,absolute};
my[moveearegw]:=Modes{immediate};
mx[moveearegl]:=Modes{indirect,postinc,offset,index,direct,absolute};
my[moveearegl]:=Modes{immediate};
FOR i:=aslb TO asrl DO
  mx[i]:=Modes{Dn};
  my[i]:=Modes{Dn}
END;
FOR i:=aslib TO asril DO
  mx[i]:=Modes{immediate};
  my[i]:=Modes{Dn}
END;
mx[dbra]:=Modes{Dn};
my[dbra]:=Modes{direct};
mx[link]:=Modes{An};
my[link]:=Modes{immediate};
```

```
  FOR i:=moveb TO movel DO
    mx[i]:=Modes{Dn,An,indirect,postinc,predec,offset,index,direct,absolute,immediate};
    my[i]:=Modes{Dn,An,indirect,postinc,predec,offset,index,direct,absolute}
  END;
  mx[mulsl]:=Modes{Dn,indirect,postinc,predec,offset,index,direct,absolute,immediate};
  my[mulsl]:=Modes{Dn};
  mx[mulul]:=Modes{Dn,indirect,postinc,predec,offset,index,direct,absolute,immediate};
  my[mulul]:=Modes{Dn}
END InitModes;

PROCEDURE expression (VAR n: INTEGER);
BEGIN
  IF sym=minus THEN GetSym;
    IF sym#number THEN Error("illegal expression") END;
    IF num>ORD(MAX(INTEGER)) THEN Error("negative number too small") END;
    n:=-INTEGER(num)
  ELSE n:=num
  END;
  GetSym
END expression;

PROCEDURE operand (VAR op: Operand; md: Modes);
BEGIN
  CASE sym OF
    keywrd:
      IF key<=d7 THEN op.m:=Dn; op.reg:=ORD(key); GetSym
      ELSIF key<=a7 THEN op.m:=An; op.reg:=ORD(key)-8; GetSym
      ELSE Error("operand expected")
      END |
    lbr:
      GetSym;
      IF (sym#keywrd) OR (key<a0) OR (key>a7) THEN
        Error("address register expected")
      END;
      op.reg:=ORD(key)-8; GetSym;
      IF sym=rbr THEN GetSym;
        IF sym=plus THEN op.m:=postinc; GetSym ELSE op.m:=indirect END
      ELSIF sym=minus THEN GetSym;
        IF sym#rbr THEN Error ("')' missing") END;
        op.m:=predec; GetSym
      ELSIF sym=comma THEN GetSym;
        IF (sym#number) & (sym#minus) THEN Error("offset expected") END;
        expression(op.disp);
        IF sym=comma THEN GetSym;
          IF (sym#keywrd) OR (key>a7) THEN
            Error("illegal addressing mode")
          END;
          op.indexreg:=ORD(key); GetSym;
          IF sym#times THEN Error("scale specification expected") END;
          GetSym;
          IF sym#number THEN Error("scale expected") END;
          IF (num#1) & (num#2) & (num#4) & (num#8) THEN
            Error ("scale must be 1, 2, 4, or 8")
          END;
          op.scale:=num; op.m:=index; GetSym
        ELSE op.m:=offset
        END;
        IF sym#rbr THEN Error ("')' missing") END;
        GetSym
      ELSE Error("illegal addressing mode")
      END |
    lbl: op.m:=direct; op.l:=lab; GetSym |
    number,minus: op.m:=absolute; expression(op.n) |
    immed: op.m:=immediate; GetSym; expression(op.n)
  ELSE Error("illegal operand")
  END;
  IF ~(op.m IN md) THEN Error("addressing mode not allowed") END
```

```
 END operand;

 PROCEDURE statement;
  VAR i: [rtr..var];
 BEGIN i:=key; GetSym;
  CASE i OF
    rtr..trapv: gen(i) |
    asl1..datal: operand(xop,mx[i]); gen(i) |
    addb..mulul:
      operand(xop,mx[i]);
      IF sym#comma THEN Error("comma expected") END;
      GetSym; operand(yop,my[i]);
      gen(i) |
    divls,divlu:
      operand(xop, Modes{Dn,indirect,postinc,predec,offset,index,direct,
        absolute,immediate});
      IF sym#comma THEN Error("comma expected") END;
      GetSym; operand(yop,Modes{Dn});
      IF sym#colon THEN Error("colon expected") END;
      GetSym; operand(zop,Modes{Dn});
      gen(i) |
    var:
      operand(xop,Modes{direct});
      IF defined(xop.l) THEN Error("label already defined") END;
      IF referred(xop.l) THEN Error("label already referred") END;
      IF sym#colon THEN Error("colon expected") END;
      GetSym; operand(yop,Modes{absolute});
      SetSize(xop.l,yop.n)
  END;
 END statement;

 PROCEDURE LabelledStatement;
 BEGIN
  IF sym=lbl THEN
    IF defined(lab) THEN Error("label already defined") END;
    SetTarget(lab); GetSym;
    IF sym#colon THEN Error("label must be followed by :") END;
    GetSym
  END;
  IF (sym=keywrd) & (key>a7) THEN statement END
 END LabelledStatement;

BEGIN (*Assembler*)
 InitGenerator; InitScanner; InitModes;
 GetSym; LabelledStatement;
 WHILE sym=eol DO GetSym; LabelledStatement END;
 IF sym#eof THEN Error("illegal syntax") END;
 CloseScanner; CloseGenerator
END Assembler.
```

Anhang A: Die Assemblersprache

Ein Assemblerprogramm besteht aus einer Folge von *Symbolen* eines *Vokabulars*, die entsprechend den Regeln der *Syntax* geformt sein müssen. An Symbolen werden *Bezeichner*, *Schlüsselwörter*, *Zahlen*, *Operatoren* und *Trennsymbole* unterschieden. Diese werden aus *Zeichen* eines *Zeichensatzes* zusammengesetzt. Dabei wird der ASCII Zeichensatz unterstellt. Er besteht aus Groß- und Kleinbuchstaben, Ziffern, grafischen Zeichen, Kontrollzeichen und dem Leerzeichen. Zur Beschreibung der Syntax wird ein erweiterter Backus-Naur Formalismus benutzt.

A.1 Vokabular

1. Bezeichner sind Folgen von Buchstaben und Ziffern, wobei das erste Zeichen ein Buchstabe sein muß.

    identifier = letter {letter | digit}.
    letter="a" | ... | "z" | "A" | ... | "Z".
    digit="0" | .. | "9".

2. Schlüsselwörter sind folgende Zeichenreihen:

    d0, d1, d2, d3, d4, d5, d6, d7, a0, a1, a2, a3, a4, a5, a6, a7, a8,
    rtr, rts, trapv,
    asl1, asr1, jmp, jsr, moveccrea, moveeaccr, pea,
    clrb, clrw, clrl, negb, negw, negl, notb, notw, notl, tstb, tstw, tstl,
    st, sf, shi, sls, scc, scs, sne, seq, svc, svs, spl, smi, sge, slt, sgt, sle,
    bra, brs, bhi, bls, bcc, bcs, bne, beq, bvc, bvs, bpl, bmi, bge, blt, bgt, ble,
    extw, extl, swap, unlk, rtd, trap, datab, dataw, datal,
    addb, addw, addl, andb, andw, andl, orb,orw, orl, subb, subw, subl,
    eorb, eorw, eorl, bchg, bclr, bset, btst,

lea, mulsw, muluw,
adddb, adddw, adddl, cmpb, cmpw, cmpl, subdb, subdw, subdl,
anddb, anddw, anddl, chkw, chkl, divs, divu, ordb, ordw, ordl,
cmpaw, cmpal,
addaw, addal, subaw, subal,
addib, addiw, addil, subib, subiw, subil,
andib, andiw, andil, cmpib, cmpiw, cmpil, eorib, eoriw, eoril, orib, oriw, oril,
bchgi, bclri, bseti, btsti, moveregeaw, moveregeal,
moveareg w, moveareg l,
aslb, aslw, asll, asrb, asrw, asrl,
aslib, asliw, aslil, asrib, asriw, asril,
dbra, link, moveb, movew, movel,
mulsl, mulul
divls, divlu,
var

3. Zahlen sind Folgen von Ziffern. Sie haben die übliche mathematische Bedeutung.

number = digit {digit}.

4. Operatoren bestehen aus einem der folgenden Zeichen

( ) * + , - : ;

5. Trennsymbole sind das Leerzeichen und die Kontrollzeichen. Zu den Kontrollzeichen gehört das Zeilenendezeichen EOL, das für Ende einer Zeile steht, und das Dateiendezeichen EOF, dem nichts mehr folgt.

6. Kommentare werden durch einen Strichpunkt eröffnet und gehen bis zum Ende der Zeile. Sie bestehen aus einer beliebigen Folge von Zeichen. Kommentare verändern nicht die Bedeutung eines Programmes.

Bezeichner, Schlüsselwörter und Zahlen müssen voneinander durch Operatoren oder Trennsymbole getrennt werden. Ansonsten haben Trennsymbole keine Auswirkung.

A.2 Programmaufbau

Programme, die von dem Assemblierer akzeptiert werden, bestehen aus einer durch Zeilenendezeichen getrennten Folge von Befehlen, die mit einem Bezeichner markiert sein können. Befehle sind entweder echte Instruktionen oder die Pseudoinstruktion *var*. Echte Instruktionen werden entsprechend der Anzahl der Parameter in vier Klassen unterteilt.

```
program = LabelledInstruction {EOL LabelledInstruction} EOF.
LabelledInstruction = [identifier ":"] [instruction].
instruction =
   operation0 |
   operation1 operand |
   operation2  operand "," operand |
   operation3 operand "," operand ":" operand |
   pseudoinstruction.
operation0 = rtr | rts | trapv.
operation1 =
   asl1 | asr1 | jmp | jsr | moveccrea | moveeaccr | pea |
   clrb | clrw | clrl | negb | negw | negl | notb | notw | notl | tstb | tstw | tstl |
   st | sf | shi | sls | scc | scs | sne | seq | svc | svs | spl | smi | sge | slt | sgt | sle |
```

```
    bra | brs | bhi | bls | bcc | bcs | bne | beq | bvc | bvs | bpl | bmi | bge | blt | bgt | ble |
    extw | extl | swap | unlk | rtd | trap | datab | dataw | datal.
operation2 =
    addb | addw | addl | andb | andw | andl | orb | orw | orl | subb | subw | subl |
    eorb | eorw | eorl | bchg | bclr | bset | btst |
    lea | mulsw | muluw |
    adddb | adddw | adddl | cmpb | cmpw | cmpl | subdb | subdw | subdl |
    anddb | anddw | anddl | chkw | chkl | divs | divu | ordb | ordw | ordl |
    cmpaw | cmpal |
    addaw | addal | subaw | subal |
    addib | addiw | addil | subib | subiw | subil |
    andib | andiw | andil | cmpib | cmpiw | cmpil | eorib | eoriw | eoril | orib | oriw | oril |
    bchgi | bclri | bseti | btsti | moveregeaw | moveregeal |
    moverearegw | moveearegl |
    aslb | aslw | asll | asrb | asrw | asrl |
    aslib | asliw | aslil | asrib | asriw | asril |
    dbra | link | moveb | movew | movel |
    mulsl | mulul.
operation3 = divls | divlu.
pseudoinstruction = var operand ":" operand.
operand = Dn | An | "("An ( ")" ["+"] | "-" ")" | "," offset["," Rn "*" scale] ")") |
    identifier | address | "#" expression.
offset = expression.
expression = ["-"] number.
scale = number.
address = number.
Rn = Dn | An.
Dn = d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7.
An = a0 | a1 | a2 | a3 | a4 | a5 | a6 | a7.
```

Operationen benötigen zu ihrer Ausführung keinen, einen, zwei oder drei Operanden. Die Art, wie auf die Operanden zugegriffen wird, heißt Adressierungsart. Es werden zehn Adressierungarten unterschieden.

1. Datenregister direkt: *Dn*
   Der Operand ist im Datenregister *Dn* enthalten.

2. Adreßregister direkt: *An*
   Der Operand ist im Adreßregister *An* enthalten.

3. Register indirekt: *(An)*
   Die Adresse des Operanden steht in Register *An*.

4. Register indirekt postinkrement: *(An)+*
   Die Adresse des Operanden steht in Register *An*. Nach dem Zugriff wird als Seiteneffekt das Register *An* um eins erhöht.

5. Register indirekt prädekrement: *(An-)*
   Zuerst wird das Register *An* um eins erniedrigt. Dies ergibt die Adresse des Operanden.

6. Indirekt mit Verschiebung: *(An, offset)*
   Die Adresse des Operanden ergibt sich aus dem Inhalt des Registers *An* plus dem Wert von *offset*.

7. Indiziert: *(An, offset, Rn*scale)*
   Die Adresse des Operanden ergibt sich aus dem Inhalt des Registers *An* plus dem Wert von *offset* plus dem Produkt des Inhalts des Registers *Rn*mit *scale*. Der Faktor *scale* darf dabei 1, 2, 4 oder 8 sein.

8. Direkt: *identifier*

Der Operand steht in der durch *identifier* bezeichneten Adresse.

9. Absolut: *address*
   Der Operand steht in der durch *address* gegebenen Adresse.

10. Unmittelbar: *#expression*
    Als Operand wird der Wert von expression verwendet. Dieser wird bei der Übersetzung ermittelt.

Anhang B: Literaturverzeichnis

Aho, A. V., R. Sethi, and J. D. Ullman. *Compilers - Principles, Techniques, and Tools*. Addison-Wesley, Reading, 1986.

Apt, K. R. Ten Years of Hoare´s Logic: A Survey - Part 1. *ACM Transactions on Programming Languages and Systems* 3, 4 (1981).

Apt, K. R. and G. D. Plotkin. Countable Indeterminism and Random Assignment. *Journal of the ACM* 33, 6 (1986).

Back, R. J. R. A Calculus of Refinement for Program Derivations. *Acta Informatica* 23 (1988).

deBakker, J. W. *Mathematical Theory of Program Correctness*. Prentice-Hall, Englewood Cliffs, 1980.

Dijkstra, E. W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, 1976.

Dijkstra, E. W. and A. J. M. van Gasteren. A Simple Fixpoint Argument Without the Restriction to Continuity. *Acta Informatica* 23 (1986).

Hehner, E. C. R. Predicative Programming - Parts I and II. *Communications of th ACM* 27, 2 (1984).

Hehner, E. C. R. *The Logic of Programming*. Prentice-Hall, Englewood Cliffs, 1984.

Hehner, E. C. R., L. E. Gupta, and A. J. Malton. Predicative Methodology.

Acta Infomatica 23 (1986).

He Jifeng, C. A. R. Hoare, and J. W. Sanders. Data Refinement Refined.
In Robinet, B. and R. Wilhelm. *ESOP 86 European Symposium on Programming*. (Lecture Notes in Computer Science 213) Springer-Verlag, Heidelberg, 1986.

Hoare, C. A. R. The Mathematics of Programming.
In *Foundations of Software Technology and Theoretical Computer Science 85*. (Lecture Notes in Computer Science 206) Springer-Verlag, Heidelberg, 1986.

Hoare, C. A. R., I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey, and B. A. Sufrin. Laws of Programming.
Communications of the ACM 30, 8 (1987).

Hoare, C. A. R. and He Jifeng. The weakest prespecification.
Fundamenta Informaticae 9 (1986).

Hoare, C. A. R. and N. Wirth. An Axiomatic Definition of the Programming Language PASCAL.
Acta Informatica 2 (1973).

Jones, C. B. *Systematic Software Development using VDM.*
Prentice-Hall, Engelwood Cliffs, 1986.

London, R. L., J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, G. J. Popek. Proof Rules for the Programming Language Euclid.
In Bauer, F. L. and M. Broy (Eds.). *Program Construction*. (Lecture Notes in Computer Science 69) Springer-Verlag, Heidelberg, 1979.

Luckham, D. C. and N. Suzuki. Verification of Array, Record and Pointer Operations in Pascal.
ACM Transactions on Programming Languages and Systems 1, 2 (1979).

Morgan, C. The Specification Statement.
ACM Transactions on Programming Languages and Systems 10, 3 (1988).

Motorola Inc. *MC68020 32 Bit Microprocessor User´s Manual, Second Edition.*
Prentice-Hall, Englewood Cliffs, 1985.

Polack, W. *Compiler Specification and Verification*.
(Lecture Notes in Computer Science 124) Springer-Verlag, Heidelberg, 1981.

Roscoe, A. W. and C. A. R. Hoare. Laws of Occam Programming.
Theoretical Computer Science 60, 2 (1988).

Sørensen, I. H. and B. Sufrin. Formal Specification and Design of a Simple Assembler.
In Hayes, I. (Ed.) *Specification Case Studies*. Prentice-Hall, Englewood Cliffs, 1987.

Waite, W. M. and G. Goos. *Compiler Construction*.
Springer-Verlag, New York, 1984.

Wirth, N. *Compilerbau*.
Teubner-Verlag, Stuttgart, 1983.

Woodcock, J. C. P. *Using Z*.
Preliminary Draft. 1988.