PREPRINT
89.01

Wolfgang Matthes

Datenzugriffsprinzipien in objektorientierten Rechnerarchitekturen

INSTITUT
FÜR INFORMATIK
UND RECHENTECHNIK

Inhalt

1. Einführung

Der Begriff der objektorientierten Architektur steht in direkter Beziehung zum Begriff der Datenabstraktion: eine objektorientierte Architektur stellt die Mittel zur Verfügung, um die Datenabstraktion in einem Rechnersystem bis zur Laufzeit aufrecht zu erhalten.
Datenabstraktion bedeutet, daß es möglich ist, eine Informationsstruktur unabhängig von ihrer rechnerspezifischen Implementierung zu betrachten (s. /3/, /7/, /9/). Jede "höhere" Programmiersprache gewährleistet dies in gewissem Maße. Bei üblichen Rechnerarchitekturen wird jedoch die Abstraktion durch die Compilierung beendet: das lauffähige Programm bezieht sich über rechnerspezifische Adressierungsmodi auf rechnerspezifisch codierte Daten. Dieser Weg von der abstrakten Formulierung zu rechnerspezifischen Befehlsfolgen ist in Bild 1 veranschaulicht.
Bei objektorientierten Architekturen besteht die Abstraktion auch zur Laufzeit. Zugriffe zu den Informationsstrukturen werden nicht unmittelbar mit Speicheradressen im Rahmen rechnerspezifischer Adressierungsverfahren ausgeführt. Stattdessen wird jede Informationsstruktur als ein Objekt aufgefaßt, das über einen Objektidentifier aufgerufen werden kann. (Der Objektidentifier ist faktisch die Ordinalzahl des jeweiligen Objekts in der geordneten Menge aller Objekte.)
Da die Informationsstrukturen natürlich stets in physischen Speichermitteln (RAM, Plattenspeicher usw.) untergebracht sind, werden letztlich immer konkrete Positionsangaben (Adressen, Zeiger auf Spuren eines Plattenspeichers usw.) für die Zugriffe benötigt. Die Abbildung von der abstrakten Informationsstruktur zu konkreten Positionsangaben wird durch Vorkehrungen gewährleistet, die als Objektdeskriptoren, Capabilities usw. bezeichnet werden (s. z. B. /4/, /5/). Das Prinzip ist in Bild 2 veranschaulicht.

Formulierung in höherer PS (Pascal)

```
VAR a,b,c: REAL;
        .
        .
        .
C := a * b;
        .
        .
        .
```

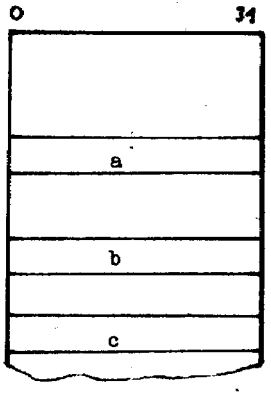
Adressen

X

Y

Z

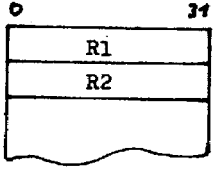
Speicherbelegung



Befehlsfolge

```
LOAD  R1, <X>
LOAD  R2, <Y>
MULT  R1,R2
STORE <Z> ,R1
```

Registerspeicher



Repräsentation einer Variablen

Bild 1 Abstrakte Formulierung und konkrete Speicherbelegung mit Befehlsfolge eines (hypothetischen) Rechners (herkömmliche Architektur)

Ablauf

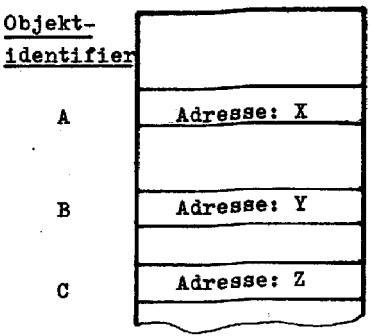
(c := a * b)

INVOKE A
INVOKE B 1)
MULTIPLY
ASSIGN C 2)

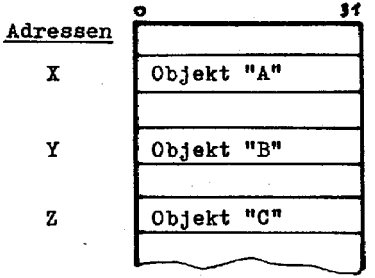
1) Holen der Werte der Argument- Objekte
2) Zuordnen des Ergebnisses zum Resultatobjekt

Tabelle der Objektdeskriptoren

(OBJECT REFERENCE TABLE ORT)



Speicherbelegung



Struktur eines Objektdeskriptors

Bild 2 Ablauf (vgl. Bild 1) und Speicherbelegung bei einer objektorientierten Architektur

Grundsätzlich ist es gleichgültig, ob diese Vorkehrungen von der Hardware des Rechners unterstützt werden oder ob sie durch Software (das "Laufzeitsystem") emuliert werden. Hardwaremittel zur Unterstützung von Zugriffsprinzipien ähnlich Bild 2 wurden verschiedentlich bereits seit Anfang der 60er Jahre realisiert (s. z. B. /10/), allerdings nicht mit dem Begriff der objetorientierten Architektur bezeichnet. Dieser ist namentlich durch ein Mikroprozessorsystem bekannt geworden, das in der Literatur viel diskutiert wurde (z. B. in /5/, /12/, /13/, /16/), aber kommerziell nicht sonderlich erfolgreich war. Die Problematik ist bereits aus einem Vergleich der Bilder 1 und 2 ersichtlich: Jeder Zugriff zu einem Objekt erfordert zunächst wenigstens einen Zugriff zur Objekttabelle; Datenabstraktion zur Laufzeit kostet also Organisationsaufwand. Sind die einzelnen Objekte relativ elementar (z. B. Fest- oder Gleitkommazahlen), so erhöht sich die Zugriffszeit beträchtlich: Ein Objektdeskriptor hat wenigstens die gleiche Anzahl an Bits wie das von ihm beschriebene Objekt. Zusätzlich schränken technologische Gegebenheiten (Anzahl der Kontakte und Busleitungen, Chipfläche usw.) die Möglichkeit ein, eine Vielzahl von Objektdeskriptoren in Registern oder anderen Speichermitteln mit besonders kurzer Zugriffszeit zu halten.

Im umgekehrten Fall, d. h. wenn überwiegend Objekte von großem Umfang zu verarbeiten sind, entstehen andere Probleme. Für Zugriffe zu derartigen Informationsstrukturen ist ein Schema ähnlich Bild 2 stets erforderlich (nötigenfalls muß es direkt von Anwendungsprogrammen implementiert werden, etwa durch das Verwalten von Zeigern). Somit ist die Unterstützung auf der zugrunde liegenden Architekturebene (Laufzeitsystem...Hardware) auf jeden Fall wünschenswert. Besondere Gesichtspunkte werden im folgenden näher erläutert. Dabei sind lediglich Zugriffe zu Datenstrukturen von Interesse; Organisation und interne Repräsentation von Programmen werden nicht betrachtet.

2. Beispiel für eine komplexe Datenstruktur: die Ternärvektorliste (TVL)

TVL wurden vorzugsweise zur rechentechnischen Behandlung BOOLEscher Probleme eingeführt (/1/, /11/). Es handelt sich um Listen, die aus Vektoren ternärer Variablen bestehen. Eine solche Variable kann einen von drei möglichen Werten annehmen (0, 1, "-" bzw. - nicht ganz exakt - "don't care"). Mit derartigen Listen können BOOLEsche Gleichungen sowie BOOLEsche Differentialgleichungen dargestellt werden; sie sind aber auch einer Vielzahl weiterer Interpretationen zugänglich (/1/, /2/). Typische Eigenschaften von TVL sind:

- TVL repräsentieren Datenstrukturen im Rahmen von Problemstellungen, die im Sinne der Komplexitätstheorie zu den besonders komplizierten gehören (NP- vollständige oder exponentielle Komplexität; s. z. B. /1/, Kap. 9).
- Sie sind in sich nochmals strukturiert. Zu jeder TVL gehört neben der eigentlichen Liste der Ternärvektoren eine Informationsstruktur, die die Beziehungen zwischen den Spalten der TVL und den zugeordneten Variablen beschreibt (Variablenliste) sowie weitere deskriptive Information (Zahl der Variablen, Zeilenzahl usw.). Dies ist in Bild 3 dargestellt.
- TVL können sehr groß werden (z. B. bei 16 Variablen bis zu 2^{15} Zeilen).
- Die Algorithmen, in denen die TVL verwendet werden, sind ihrerseits recht kompliziert (vgl. /1/, /11/).

Deshalb müssen zwei Erfordernisse erfüllt werden:

1. Jeweils effektivste Speicherorganisation und Programmstruktur für die leistungsentscheidenden Algorithmen (Basis- und Mengenalgorithmen nach /11/) in Abhängigkeit von der konkreten Rechnerstruktur bis hin zur Nutzung spezieller Hardware (/8/, /18/, /19/).

BOOLEsche Gleichung

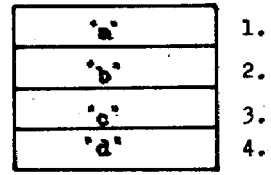
$ab\bar{c} \vee \bar{b}d \vee \bar{a}c = 1$

zugehörige TVL

a	b	c	d
1	1	0	-
-	0	-	1
0	-	1	-

Variablenliste

korrespondierende Spalten



(Es sind die Namen der Variablen in den korrespondierenden Spalten gespeichert.)

Beschreibende Information des Objekttyps "TVL"

Variablenzahl
Zeilenzahl
Adresse Var.liste
Adresse TVL- Bereich

Bild 3 Darstellung BOOLEscher Gleichungen mit Ternärvektorlisten (Beispiel)

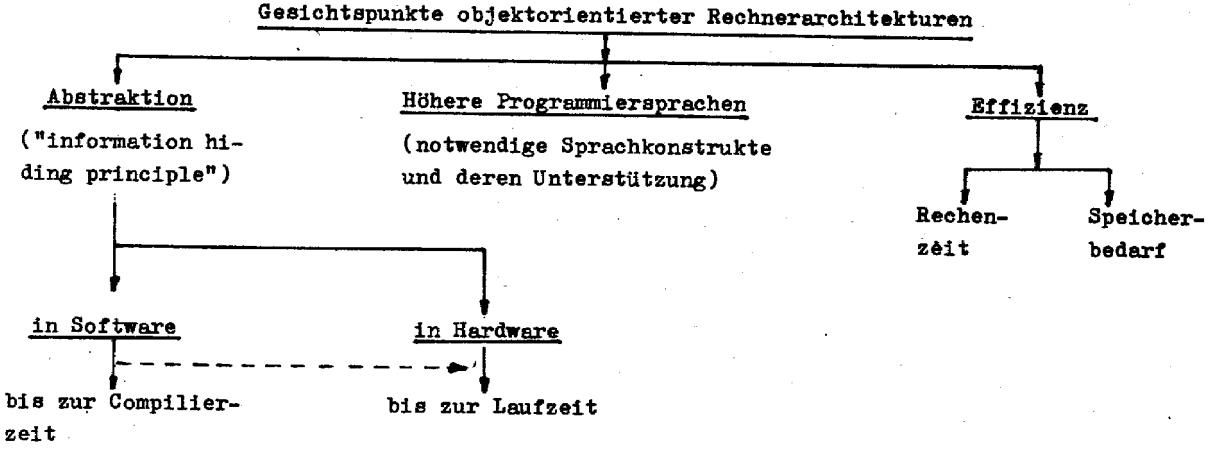
2. Abstrakte Formulierung der Anwendungs- Algorithmen unabhängig von der konkreten Implementierung (Datenabstraktion).

Die folgenden Betrachtungen gelten somit allgemein für Informationsstrukturen, die
- sehr umfangreich sein können
- eine hohe Effektivität der Speicherung und Verarbeitung erfordern (Laufzeiteffizienz)
- in komplexen Algorithmensystemen angewandt werden.

Bild 4 gibt einen Überblick, unter welchen wesentlichen Gesichtspunkten objektorientierte Rechnerarchitekturen zu untersuchen sind.

3. Datenabstraktion und Zugriffsorganisation

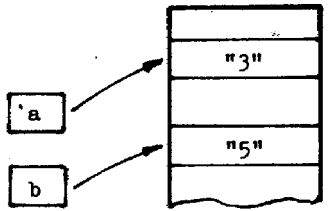
In der herkömmlichen Programmierweise hat eine Variable zwei Funktionen (/9/): sie bezeichnet die Informationsstruktur (im Text des Quellprogramms), und sie repräsentiert zur Laufzeit einen Behälter für die Information. (durch die Adressenangaben in den Befehlen; vgl. Bild 1). Bei einem konsequenten objektorientierten Ansatz sind diese beiden Funktionen voneinander getrennt (auf diesem Prinzip beruht z. B. die Programmiersprache CLU; s. /7/, /9/). Variable repräsentieren die Namen von Objekten (Bezeichnerfunktion). Die Objekte selbst sind die Behälter der Information. Eine Variable kann zur Laufzeit auf ein Objekt zugreifen (vgl. Bild 2). Theoretisch ist die Lebensdauer der Objekte unbeschränkt, und es ist möglich, daß verschiedene Variable gleichzeitig auf ein Objekt zugreifen. Bild 5 veranschaulicht die Unterschiede anhand einer einfachen Zuweisung (der Variablen a ist anfänglich der Wert 3 zugeordnet, der Variablen b der Wert 5). Im herkömmlichen Fall sind den Variablen direkt Speicherplätze zugewiesen, und eine Operation a := b wird so realisiert, daß der Inhalt des b zugewiesenen Speicherplatzes in den



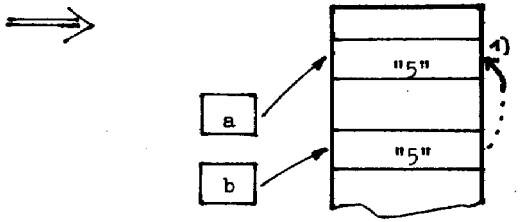
Die Abstraktion muß bis zur Laufzeit wirksam bleiben, wenn die Größe der Objekte nicht vorhersagbar ist !

Bild 4

a) herkömmlich

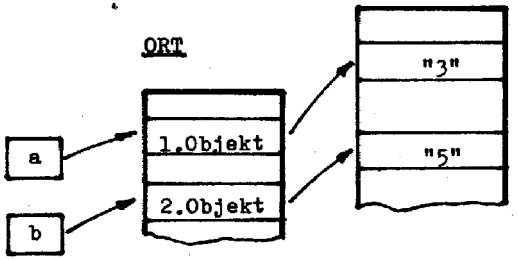


a := b

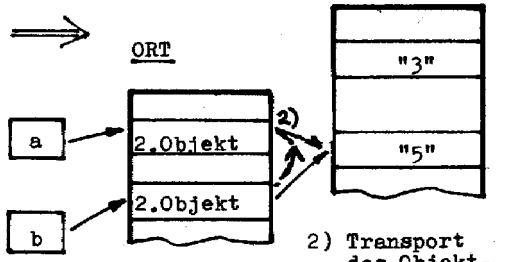


1) Transport des Wertes

b) objektorientiert



a := b



2) Transport des Objekt-deskriptors

Bild 5 Ablauf einer Zuweisung

Speicherplatz der Variablen a transportiert wird.
Im objektorientierten Fall werden keine Werte transportiert, sondern es wird der Objektdeskriptor, der das zweite Objekt (Wert = 5) adressiert, in die Position der Objekttabelle gebracht, die der Variablen a zugeordnet ist. Damit greifen beide Variablen a, b mit identischen Objektdeskriptoren auf ein einzigen Objekt zu. Das andere Objekt (Wert = 3) existiert weiterhin. Wenn kein weiterer Objektdeskriptor darauf zugreift, kann im praktischen Rechenbetrieb der Speicherplatz zur freien Verfügung gestellt werden.
Tafel 1 zeigt die Unterschiede im Programmablauf, wenn mehrere Operationen mit den Variablen ausgeführt werden. Es ist ersichtlich, daß nach einer Zuweisung im herkömmlichen Fall eine Änderung der zugewiesenen Variablen (b) keine Auswirkung auf die andere Variable (a) hat. Bei objektorientierter Zugriffsorganisation (wenn nicht Werte, sondern Deskriptoren kopiert werden) wirkt sich jede Änderung des Wertes eines Objekts auf alle Variablen aus, die auf dieses Objekt zugreifen, d. h. diese Änderung ist für alle betreffenden Variablen sichtbar ("visibility effect"; vgl. bes. /7/, /9/).

4. Strukturierung von Objekten

Objektorientierte Architekturen sind besonders dann von Interesse, wenn überwiegend umfangreiche und kompliziert aufgebaute Informationsstrukturen zu verarbeiten sind. Im Sinne der Abstraktion ist jede derartige Struktur als ein einziges Objekt aufzufassen; es ist aber auch notwendig, zu dessen Teilen zugreifen zu können. Dies führt zum Begriff des Verbundobjekts: das ist eine Informationsstruktur, die ihrerseits aus einzelnen Objekten besteht. Diese Objekte sind in einer Objektliste aufgeführt. Die Objektliste selbst ist das eigentliche Verbundobjekt. Bild 6 zeigt dies am Beispiel einer Ternärvektorliste, die gemäß Bild 3 aus den Objekten "Variablenliste" und "TVL- Bereich" besteht (die be-

Ablauf	herkömmlich	objektorientiert
a := 3 b := 5 . . . a := b [1] . . . b := 9 x := OP (a)	Operation mit a = 5	Operation mit a = 9 [2]

1) a, b wurden zwischenzeitlich nicht verändert.

2) Durch die Zuweisung (a := b) wird die Änderung von b (:= 9) für a sichtbar (a und b zeigen beide auf das gleiche Objekt).

Tafel 1 Veranschaulichung der unterschiedlichen Auswirkungen einer Zuweisung im herkömmlichen und objektorientierten Fall

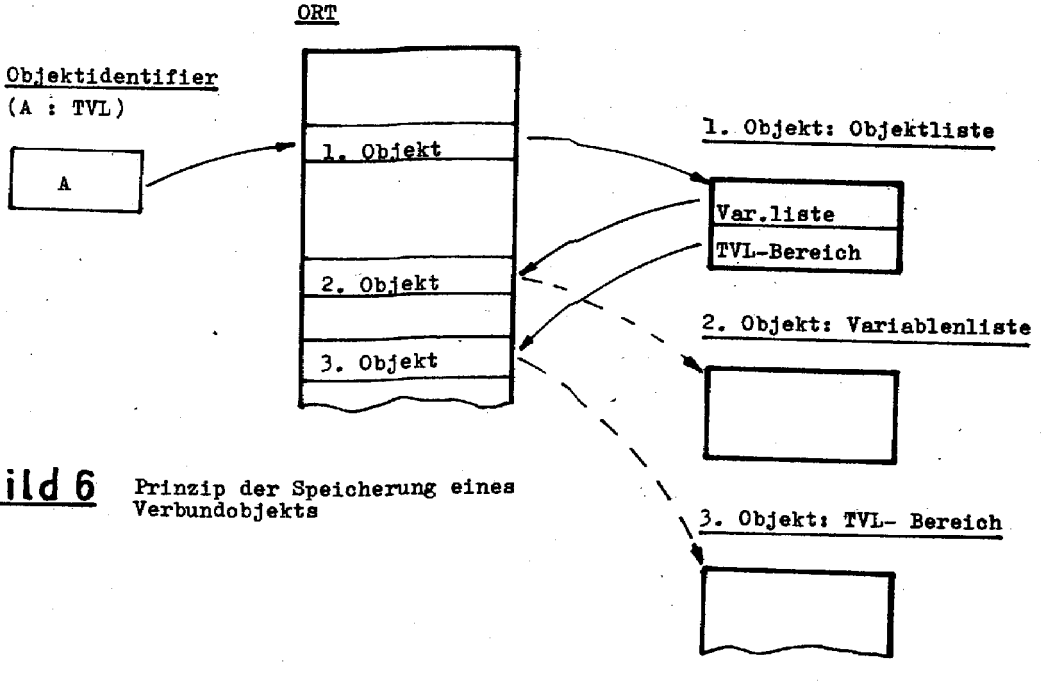


Bild 6 Prinzip der Speicherung eines Verbundobjekts

schreibende Information, z. B. die Zeilenzahl, ist in den Objektdeskriptoren enthalten).
Zugriffe zu diesen Objekten sind mit Selektionsoperationen (vgl. /7/) möglich. Diese greifen zunächst zum Gesamt- Objekt zu (d. h. zur Objektliste) und entnehmen daraus den Objektidentifier des gewünschten Teil- Objekts. Damit kann auf bereits beschriebene Weise (Bild 2) zu diesem Objekt zugegriffen werden.

5. Effizienzprobleme und Sprachkonstrukte

Der objektorientierte Ansatz kann bei konsequenter Implementierung zu extremer Ineffizienz führen. So läßt sich beispielsweise die Struktur "TVL" gem. Bild 3 in einer Programmiersprache, die über ein Typenkonzept die Datenabstraktion unterstützt (z. B. Ada; /6/), auf "naive" Weise wie folgt schrittweise durch Aggregieren ihrer Bestandteile deklarieren:

```
type TVAR is (H, L, X);
-- ternäre Variable: H: 1, L: 0, X: -
type TVEC is array (1..VAR_COUNT) of TVAR;
-- ternärer Vektor
type TVL_AREA is array (1..ROW_COUNT) of TVEC;
-- TVL- Bereich
type VAR_LIST is array (1..VAR_COUNT) of VAR_NAME;
-- Variablenliste; VAR_NAME ist anderweitig definiert
-- ROW_COUNT und VAR_COUNT repräsentieren die Zeilen-
-- bzw. Variablen- (Spalten-)- Anzahl durch ganze Bi-
-- närzahlen (INTEGER).
-- Die gesamte TVL wird so deklariert:
type TVL is record
        VL: VAR_LIST;
        TVL_PROPER: TVL_AREA;
end record;
```

Bei einer strikt objektorientierten Implementierung (im Sinne von /7/) wird jede Deklaration als die eines Objekts aufgefaßt. Die Ternärvariablen sind die elementaren Objekte, daraus werden Ternärvektoren gebildet und aus diesen wiederum Ternärvektorlisten. Nach dem Schema von Bild 6 wird somit für jede Ternärvariable ein Objektidentifier und ein Objektdeskriptor gespeichert (etwa 64...128 bit gegenüber 2 bit für die übliche rechnerinterne Darstellung !).
Es ist ersichtlich, daß im Interesse der Effizienz die Implementierung derartiger Datenstrukturen beeinflußbar sein muß, d. h. es muß möglich sein, bei der Deklaration die Art und Weise der Speicherung zu bestimmen.
Offensichtlich ist in manchen Fällen das Prinzip nach Bild 6 nützlich. In anderen Fällen muß hingegen eine dichtgepackte maschineninterne Darstellung erzwungen werden, selbst wenn dadurch die Eleganz des objektorientierten Ansatzes beeinträchtigt wird.
Beispielsweise ist es von Vorteil, die TVL als Verbundobjekt zu deklarieren:

- Bei mehreren TVL mit identischer Variablenbelegung wird die Variablenliste nur einmal benötigt.
- Gibt es mehrere an sich identische BOOLEsche Gleichungen mit unterschiedlicher Variablenbelegung (z. B. enthalten größere digitale Systeme oft eine Vielzahl an sich identischer Schaltnetzwerke), so können diese mit einem einzigen TVL- Bereich (TVL_AREA) dargestellt werden.
- Um festzustellen, ob zwei TVL eine identische Variablenzuordnung haben, ist es lediglich erforderlich, die Objektidentifier ihrer Variablenlisten miteinander zu vergleichen.

Andererseits ist es indiskutabel, Variablenlisten und (erst recht) TVL- Bereiche ihrerseits gemäß Bild 6 aus Objekten aufzubauen.
Es erweist sich somit als zweckmäßig, zusätzlich zu den Verbundobjekten zusammengesetzte Objekte einzuführen. Bild 7 zeigt die grundsätzliche Einteilung der Objekte.

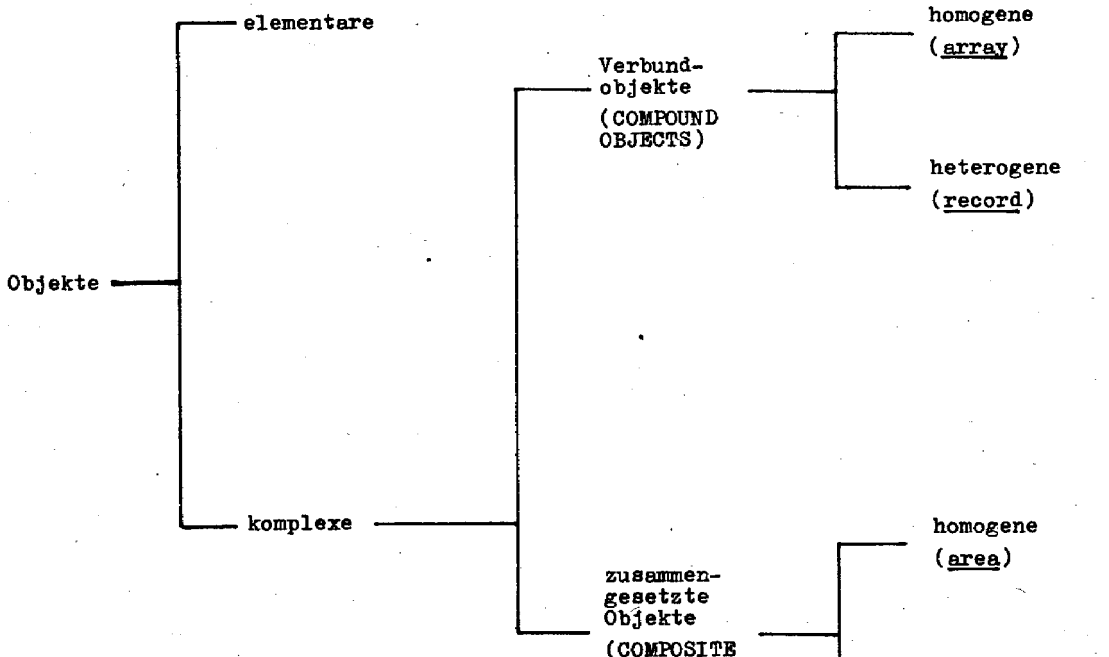


Bild 7 Übersicht über die Einteilung der Objekte

Im folgenden werden Verbundobjekte durch übliche Sprachkonstrukte (record, array usw.) bezeichnet. Für zusammengesetzte Objekte werden neue Sprachkonstrukte eingeführt (in Anlehnung an /9/): area weist den Compiler an, daß die Struktur in der jeweils maschinenspezifisch effizientesten Weise zu speichern ist. area an sich steht für homogene zusammengesetzte Objekte, record area für solche, die aus verschiedenartigen Teilstrukturen zusammengesetzt sind. Damit ist die Struktur "TVL" wie folgt zu deklarieren:

```
type TVAR is area (L, H, X); -- Codierung mit 2 bit
type TVEC is area (1..VAR_COUNT) of TVAR;
-- Es handelt sich um ein direktes Aneinanderreihen der
-- 2 bit- Ternärvariablen.
type TVL_AREA is area (1..ROW_COUNT) of TVEC;
-- Die Ternärvektoren werden direkt aneinandergefügt.
-- Bei der Aggregierung von area- Strukturen wird von
-- maschinenspezifischen Details abstrahiert; es wird an-
-- genommen, daß Wort- und Segmentgrenzen sowie andere
-- Einzelheiten der jeweiligen Speicherorganisation auto-
-- matisch berücksichtigt werden.
-- Im folgenden wird ohne nähere Erläuterung der Variablen-
-- name deklariert:
type VAR_NAME is record area
     TAG_FIELD: INTEGER range 0..15 := 0;
     DERIVATION_CODE: INTEGER range 0..15 := 0;
     BINARY_NAME: INTEGER range 0..(2 exp 24)-1;
end record area;
-- Ein Variablenname besteht demgemäß aus einem 32 bit-
-- Wort, das ein 4 bit- Kennungsfeld, einen 4 bit- Ablei-
-- tungscode sowie einen binär codierten Variablennamen
-- von 24 bit Länge enthält. Damit wird die Variablenliste
-- deklariert:
type VAR_LIST is area (1..VAR_COUNT) of VAR_NAME;
```

Die gesamte TVL ist das vorstehend beschriebene Verbundobjekt (record). Dessen Teile belegen jeweils zusammenhängende Speicherbereiche in der maschinenspezifisch zweck-

mäßigsten Codierung.
Das in Bild 5 b) illustrierte Prinzip der Zuweisung ist bei komplexen Informationsstrukturen oft nützlich: es ist einfacher, einen Deskriptor zu transportieren als das vollständige Objekt (z. B. 8 Bytes im Gegensatz zu einigen tausend Bytes). Andererseits gibt es Fälle, in denen ein Ablauf gemäß Bild 5 a) gewünscht wird. Somit ist es zweckmäßig, Sprachkonstrukte einzuführen, die es gestatten, das jeweils erforderliche Prinzip der Zuweisung auszuwählen:

1. Zuweisung im Sinne von Bild 5 b): a := b.
   Die Variable a erhält den selben Objektidentifier wie die Variable b. Wird eine Variable als Resultat einer Operation zugewiesen (z. B. a := b OP c), so wird für das aus der Verknüpfung resultierende neue Objekt ein neuer Objektdeskriptor erzeugt.

2. Modifikation eines existierenden Objekts: new a := b.
   Das Objekt, das durch den Objektdeskriptor der Variablen a beschrieben wird, erhält den Wert des Objekts, das der Variablen b zugeordnet ist. Dabei wird das gesamte Objekt kopiert.

3. Lambda- Ausdruck: a := lambda OP (x, y, ...).
   Es wird ein Objekt erzeugt, das aus der Befehlsfolge (Funktion, Prozedur) OP zusammen mit den aktuellen Parametern x, y, ... besteht (es handelt sich um ein heterogenes Verbundobjekt aus den Befehlen und den aktuellen Parameterangaben). Der Variablen a wird der betreffende Objektdeskriptor zugeordnet.

4. Berechnen eines Lambda- Ausdrucks: yield a.
   Der Lambda- Ausdruck wird durch seinen Wert ersetzt (die Berechnung, z. B. von OP (x, y, ...), wird ausgeführt; das resultierende Objekt wird der jeweiligen Variablen zugeordnet). Soll der Lambda- Ausdruck selbst erhalten bleiben, so ist eine Zuweisung gem. Punkt 1 anzugeben (etwa b := yield a), sonst gemäß Punkt 2 (new b := yield a).

Der Vorteil der Lambda- Ausdrücke besteht in der Einsparung von Speicherplatz und Rechenzeit, wenn Resultate nicht unmittelbar benötigt werden, etwa im Rahmen von Abläufen der Verwaltung und Problemanalyse. Beispiel:
Der Ablauf habe die Form

1. a := OP (b, c, d);
2. weitere Schritte, in denen a nicht benötigt wird
3. x := OP1 (a, e, f);

Das der Variablen a zugeordnete Objekt ist nach Schritt 1 mit dem berechneten Wert belegt. Dieser Speicherbereich steht für Schritt 2 nicht zur Verfügung.
Alternativ kann der Ablauf so formuliert werden:

1. a := lambda OP(b, c, d);
2. wie oben
3. x := OP1 (yield a, e, f);

In Schritt 1 wird nun nicht mehr der Wert des Resultats, sondern der (zumeist wesentlich kürzere) Lambda- Ausdruck gespeichert, so daß für die folgenden Schritte mehr Speicherplatz verfügbar ist. Der Wert wird erst in Schritt 3 berechnet, und dessen Speicherplatz wird sofort nach Bildung des Resultates x wieder freigegeben.

6. Zusammenfassung

Die vorstehenden Überlegungen repräsentieren einen Ausschnitt aus Untersuchungen, die das Ziel verfolgen, eine Architekturdefinition für die Implementierung hochkomplexer Algorithmen anzugeben. Dabei ist sowohl den Gesichtspunkten der Effizienz gerecht zu werden (wegen des Rechenzeitbedarfs und des Umfangs der Daten) als auch jenen der Abstraktion (wegen der Kompliziertheit vieler Algorithmen und der erforderlichen Portabilität zwischen den verschiedenen rechentechnischen Mitteln).
Für die Diskussion von Sprachkonstrukten ist in diesem Zusammenhang deren "Tiefenstruktur" wesentlich (diese ist durch die jeweils bewirkten Effekte gekennzeichnet) und nicht die "Oberflächenstruktur", die durch die Syntax einer konkreten Programmiersprache gegeben ist.
So könnten anstelle zusätzlicher Schlüsselworte Compileranweisungen vorgesehen werden, um die Implementierung von Informationsstrukturen steuern zu können (im Falle der Sprache Ada bieten sich dafür pragma- Anweisungen an). Dies korrespondiert mit der Erkenntnis (vgl. /15/), daß auch sehr universell ausgelegte Programmiersprachen für manche Problemkreise nicht ohne weiteres geeignet sind, sondern der Ergänzung durch Spezialsprachen bedürfen; diese werden dann zweckmäßigerweise dem allgemeinen Erscheinungsbild der zugrunde liegenden Universalsprache angepaßt.

7. Literaturverzeichnis

/1/ Bochmann, D.; Zakrevskij, A. D.; Posthoff, Ch. (Herausg.): Boolesche Gleichungen. Berlin: Verlag Technik 1984.

/2/ Fehmel, J.; Posthoff. Ch.; Steinbach, B.: Binäre Systeme- Rechnergestützter Schaltungsentwurf. Wissenschaftliche Schriftenreihe der Technischen Hochschule Karl- Marx- Stadt, 1982, H. 7.

/3/ Hext, J. B.: Data Abstraction Facilities. The University of Sydney: Basser Department of Computer Science Technical Report No. 120; March 1977.

/4/ Jones, A. K.; Gehringer, E. F. (ed.s): The CM^x Multiprocessor Project: A Research Review. Carnegie- Mellon University Pittsburgh/PA. Computer Science Department CMU- CS- 80- 131, July 1980.

/5/ Kahn, K. C.: Object- oriented languages tackle massive programming headaches. Electronics, Nov. 17, 1982, S. 141- 145.

/6/ Ledgard, H.: ADA- An Introduction/ Ada Reference Manual (July 1980). New York- Heidelberg- Berlin: Springer- Verlag 1981.

/7/ Liskov, B. et al.: CLU Reference Manual. Cambridge/Mass.: Massachusetts Institute of Technology MIT/LCS/TR-225 1979.

/8/ Matthes, W.: Spezielle Hardware zur Verarbeitung von Ternärvektorlisten. Dissertation (A). Technische Universität Karl- Marx- Stadt 1987.

/9/ Moss, J. E. B.: Abstract Data Types in Stack Based Languages. Cambridge/Mass.: Massachusetts Institute of Technology MIT/LCS/TR-190 1978.

/10/ Organick, E. I.: Computer Systems Organization- The B 5700/B 6700 Series. New York- London: Academic Press 1973.

/11/ Posthoff, Ch.; Steinbach, B.: Binäre Gleichungen- Algorithmen und Programme. Wissenschaftliche Schriftenreihe der Technischen Hochschule Karl- Marx- Stadt, 1979, H. 1.

/12/ Rattner, J.; Lattin, W. W.: Ada determines architecture of 32 bit- microprocessor. Electronics, February 24, 1981, Vol. 4 No. 54, S. 119- 126.

/13/ Schlechter, J.: Objektorientierte Architektur- Basis für ein neues Mikroprozessorkonzept. Nachrichtentechnik Elektronik, Berlin 34 (1984), H. 12, S. 465- 468.

/14/ Schwanke, R. W.: Execution Environments in Programming Languages and Operating Systems. Carnegie- Mellon University Pittsburgh/PA. Computer Science Department CMU- CS- 82- 124, 1982.

/15/ Schwartz, R. L.; Melliar- Smith, P. M.: The Suitability of Ada for Artifical Intelligence Applications. SRI International, Menlo Park, California, 1980.

/16/ Zeigler, S. et al.: Ada for the Intel 432 Microcomputer. Computer, June 1981, S. 47- 56.

/17/ Zemanek, H.: Abstrakte Objekte. Elektronische Rechenanlagen 10 (1967), H. 5, S. 208-211.

/18/ WP 235 744. Anordnung zur Verarbeitung von Ternärvektorlisten.

/19/ WP 242 299. Spezialprozessoranordnung zur Verarbeitung von Ternärvektorlisten.