Brauer, Johannes

Working Paper

Typen, Objekte, Klassen - Teil 2: Sichtweisen auf Typen

Arbeitspapiere der Nordakademie, No. 2010-05

Provided in Cooperation with:
Nordakademie - Hochschule der Wirtschaft, Elmshorn

Suggested Citation: Brauer, Johannes (2010) : Typen, Objekte, Klassen - Teil 2: Sichtweisen auf Typen, Arbeitspapiere der Nordakademie, No. 2010-05, Nordakademie - Hochschule der Wirtschaft, Elmshorn

This Version is available at:
https://hdl.handle.net/10419/38606

ARBEITSPAPIERE DER NORDAKADEMIE
ISSN 1860-0360

Nr. 2010-05

Typen, Objekte, Klassen – Teil 2: Sichtweisen auf Typen

Prof. Dr.-Ing. Johannes Brauer

April 2010

Dieses Arbeitspapier ist als PDF verfügbar: http://www.nordakademie.de/arbeitspapier.html

NORDAKADEMIE
HOCHSCHULE DER WIRTSCHAFT

Köllner Chaussee 11
25337 Elmshorn
http://www.nordakademie.de

Typen, Objekte, Klassen – Teil 2: Sichtweisen auf Typen

Johannes Brauer

24. April 2010

Inhaltsverzeichnis

Zunächst erfolgt die Betrachtung von Datentypen als Repräsentanten von Mengen. Anschließend wird die beim Entwurf von Datenstrukturen für Programme nahe liegende konstruktive Sichtweise auf den Typbegriff behandelt. Schließlich wird die Typdefintion als Abstraktionsmechanismus behandelt. Für alle dargestellten Sichtweisen auf den Typbegriff wird erläutert, wie sich diese in klassischen Typsystemen verschiedener Programmiersprachen wieder finden.

1 Motivation

Nach der im ersten Arbeitspapier[1] der Reihe „Typen, Objekte, Klassen“ erfolgten ersten Annäherung an einige grundlegende Begriffe der Typtheorie werden hier denotationale, abstrakationsorientierte und konstruktive Sichtweisen auf den Typbegriff eingenommen. Diese verschiedenen Sichtweisen haben sich zum Teil „historisch“ entwickelt, d. h. sie sind von der Entwicklung verschiedener Programmiersprachen kaum zu trennen. Wenn auch nicht alle von den Programmiersprachen hervorgebrachten Formen, mit Datentypen umzugehen, in modernen Sprachen noch Verwendung finden, so kann das Studium der Entwicklung auf diesem Gebiet einerseits wertvolle Erkenntnisse über die Vielfalt des Typbegriffs vermitteln

[1] Johannes Brauer. Typen, Objekte, Klassen – Teil 1: Grundlagen. Arbeitspapier 2009-05, NORDAKADEMIE Hochschule der Wirtschaft, Juni 2009

In der englischsprachigen Literatur wird auch von „denotational view“ gesprochen, vgl. Kapitel 7 in

Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers, 2. edition, 2006

In der deutschsprachigen Literatur ist „denotationale Sichtweise“ nicht sehr verbreitet.

andererseits auch den Blick dafür schärfen, manche vermeintliche Neuentwicklung als „alten Hut“ zu erkennen.

Bei dem Versuch eine „Typenlehre“ zu schaffen, die einerseits theoretisch fundiert, andererseits aber für den Unterricht in Bachelor-Studiengängen geeignet ist, darf nach Ansicht des Autors diese Herangehensweise an den Typbegriff nicht fehlen. Damit soll der Versuch unternommen werden, einen weiteren Baustein für die Errichtung eines Begriffsgebäudes von „unten nach oben“ zu erschaffen. Wie die in diesem und den übrigen Arbeitspapieren der Reihe „Typen, Objekte, Klassen“ beschriebenen Bausteine zu einem harmonischen Ganzen zusammenzufügen sind, bleibt aber noch zu klären.

2 *Mengenorientierte Sicht auf Datentypen*

Die Betrachtung von Typen als Wertemengenstellt eine mögliche Sichtweise auf den Typbegriff dar.

Schon in FORTRAN – einer der ersten höheren Programmiersprachen – existierte ein Typbegriff, der zwei Arten von Zahlen unterschied. Ganzzahlige Werte gehörten zum Typ `INTEGER` und Gleitkommazahlen zum Typ `REAL`. Mathematisch betrachtet ist die Menge der zu `INTEGER` gehörenden Zahlen eine endliche Teilmenge der ganzen Zahlen, während der Typ `REAL` eine endliche Teilmenge der rationalen Zahlen darstellt. Ein Wert ist demnach von einem bestimmten Typ, wenn er Element einer Menge ist, die diesen Typ repräsentiert.

Die Analogie zur Mathematik reicht aber nicht weit. So ist die den Typ `INTEGER` repräsentierende Menge keine Teilmenge von `REAL`.

> Die Unterscheidung dieser beiden Zahlenarten in FORTRAN war in erster Linie technisch motiviert: Die Repräsentation von ganzen Zahlen als Dualzahlen (im Einer- oder Zweier-Komplement) erlaubte eine einfache und effiziente elektronische Realisierung von arithmetischen Operationen. Das Rechnen mit aus Mantisse und Exponent bestehenden Gleitkommazahlen ist hingegen vergleichsweise komplex. Für derartige Rechenoperationen stand auf den Rechnern, auf denen die ersten FORTRAN-Compiler liefen, keine Hardware zur Verfügung. Die Unterscheidung der Typen wurde vom Compiler genutzt, um den für die Ausführung der arithmetischen Operationen typgemäßen Maschinen-Code zu erzeugen.

Für die FORTRAN-Typen `INTEGER` und `REAL` gibt es auch in den meisten modernen prozeduralen und objektorientierten Programmiersprachen eine Entsprechung[2]. Dass in diesen auch so genannten „höheren“ Programmiersprachen von den durch die technische Realisierung bestimmten Beschränkungen nicht abstrahiert wird, ist ein seltsamer Anachronismus. Dabei ist weniger die unvermeidlich beschränkte Genauigkeit von Gleitkommazahlen als die Größenbeschränkung von Integer-Zahlen das Problem.

[2] Z. B. `int` und `float` in C oder Java Es handelt sich dabei um so genannte Basistypen (auch „eingebaute“ Typen), die nicht erweitert werden können.

Auch wenn bei einer mengenorientierten Sicht auf Datentypen die Identifikation der zu einem Typ gehörende Wertemenge im Vordergrund steht, ist mit solchen Typen meist auch ein Satz von zulässigen Operationen verknüpft. Das sind bei den genannten

Zahltypen in erster Linie die Grundrechenarten. Die Beschränkung auf Maschinenzahlen führt dann dazu, dass elementare mathematische Gesetze – wie z. B. das Assoziativgesetz – nicht mehr uneingeschränkt gelten. Die Analogie zur Mathematik ist hier nur bedingt gegeben.

Zwar gilt mathematisch (für $x, y, z \in \mathbb{Z}$) die Gleichung:

$$(x + y) - z = x + (y - z)$$

Sind x, y und z hingegen INTEGER-Variablen führt die Berechnung von $x + y$ möglicherweise zu einer Bereichsüberschreitung, die bei der Auswertung der rechten Seite der Gleichung nicht auftritt.

AUFZÄHLUNGSTYPEN stellen ein sehr anschauliches Beispiel für eine mengenorientierte Sicht auf Datentypen dar. Sie sind z. B. aus PASCAL und MODULA-2 sowie C und Java bekannt und dienen in erster Linie der Steigerung der Lesbarkeit von Programmen. Hier wird ein Datentyp dadurch definiert, dass man die Elemente, die Bestandteil der Wertemenge sein sollen, explizit aufzählt. In PASCAL oder MODULA-2 könnte ein Typ für die Wochentage z. B. folgendermaßen konstruiert werden:

```
TYPE Wochentag = (Sonntag, Montag, Dienstag, Mittwoch,
                  Donnerstag, Freitag, Sonnabend)
```

Gemäß C-Syntax lautet die Typdefinition:

```
typedef enum {Sonntag, Montag, Dienstag,
              Mittwoch, Donnerstag,
              Freitag, Sonnabend}
        Wochentag
```

Die Bezeichner `Montag, Dienstag` usw. repräsentieren die symbolischen Konstanten des neuen Typs. Einer Variablen dieses Typs können nur genau diese Konstanten zugewiesen werden. Vom Compiler werden intern diesen Konstanten der Reihe nach die ganzzahligen Werte $0, 1, \ldots$ zugeordnet. Während in PASCAL und MODULA-2 eine strenge Typprüfung durchgeführt wird, d. h. einer Variablen `tag`[3] kann nur eine dieser Konstanten zugewiesen werden, ist in C die Zuweisung jedes Integer-Wertes zulässig.

[3] deklariert durch:
`VAR tag: Wochentag`

Grundsätzlich gilt für die mengenorientierte Sicht auf Datentypen, dass die Semantik eines Ausdrucks ein Wert ist, der in dem seinem Typ zugehörigen Wertebereich enthalten ist. Daraus ergibt sich die Frage: Handelt es sich um eine Typverletzung, wenn z. B. die Multiplikation von zwei `int`-Werten in Java einen Wert außerhalb des Wertebereichs des Basistyps `int` liefert? Diese Frage wird in verschiedenen Typsystemen durchaus unterschiedlich beantwortet, in den meisten Fällen wird eine Bereichsüberschreitung dieser Art aber nicht als Typverletzung betrachtet. Sie könnte ggf. ohnehin erst zur Laufzeit festgestellt werden, sie ist also einer statischen Typprüfung nicht zugänglich. Die Endlichkeit des Wertebereichs eines Typs wird also vom Typsystem ignoriert.

Das ist bei Aufzählungstypen hingegen anders: Werte eines Aufzählungstyps sind geordnet. Für den oben eingeführten Typ `Wochentag` gilt z. B. die Relation *Dienstag* $<$ *Mittwoch*. In PASCAL stehen zwei Standardfunktionen (`succ` und `pred`) zur Verfügung, die es erlauben, den Nachfolger bzw. Vorgänger eines Wertes bezüglich der Ordnung des Datentyps zu bestimmen. Der Ausdruck

```
succ(Sonnabend)
```

wird hier aber (spätestens zur Laufzeit) als Typfehler erkannt.

TEILBEREICHSTYPEN – ebenfalls durch PASCAL eingeführt – beschreiben zusammenhängende Teilmengen eines zugrunde liegenden diskreten[4] Basistyps. Auch hier ist die mengenorientierte

[4] Ein Datentyp heißt diskret, wenn seine Werte aufzählbar sind und es für jeden Wert (außer dem kleinsten und dem größten) einen wohl definierten Vorgänger und Nachfolger gibt.

Sichtweise offenkundig. In PASCAL-Syntax könnte ein Teilbereichstyp[5] für Arbeitstage, der sich auf den oben angegebenen Typ `Wochentag` bezieht, so aussehen:

[5] engl.: subrange type

```
TYPE Arbeitstag = Montag .. Freitag
```

Eine Variable vom Typ `Arbeitstag` kann demnach nur die Werte von `Montag` bis `Freitag` annehmen.

Die interne Repräsentation von Aufzählungstypen besteht – wie bereits erwähnt – in der Regel in einer Abbildung ihrer Konstanten auf kleine ganze Zahlen (meist beginnend bei 0). In einer typstrengen Sprache wie PASCAL wird ein Aufzählungstyp dadurch aber nicht zu einem „Teilbereichstyp“ von `Integer`. Es handelt sich bei den Typen `Wochentag` und `Integer` konzeptionell um disjunkte Mengen. Die Verwendung eines `Integer`- oder eines `Wochentag`-Wertes in einem Kontext, in dem ein Wert des jeweils anderen Typs erwartet wird, wird als Typverletzung behandelt.

Für den Typ `Wochentag` würden die Konstanten `Sonntag` bis `Sonnabend` demnach intern auf die Zahlen 0 bis 6 abgebildet.

In einer Sprache mit einem schwachen Typsystem wie C werden hingegen Werte von Aufzählungstypen und Ganzzahltypen als kompatibel betrachtet.

Die bisherige Betrachtung der mengenorientierten Sicht auf Datentypen konzentrierte sich auf ein paar konkrete Beispiele von Basistypen oder Typkonstruktoren, bei denen die Intuition von Typen als Wertemengen nahe liegt. Im ersten Arbeitspapier[6] der Reihe „Typen, Objekte, Klassen“ wurde die Menge aller möglichen Bitmuster als Beispiel für ein ungetyptes Universum betrachtet.

[6] Johannes Brauer. Typen, Objekte, Klassen – Teil 1: Grundlagen. Arbeitspapier 2009-05, NORDAKADEMIE Hochschule der Wirtschaft, Juni 2009

Um sich von diesen konkreten Beispielen zu lösen, kann man die Menge aller Werte, die eine Programmiersprache zulässt[7] als Universum W aller Werte betrachten. Ein Typ ist dann eine Menge von Werten aus W, wobei nicht jede beliebige Teilmenge aus W ein legaler Typ ist. Legale Typen müssen bestimmte elementare technische Eigenschaften besitzen, deren zugehörige Teilmengen aus W auch als *Ideale*[8] bezeichnet werden. Typen von Programmiersprachen sind immer Ideale.

[7] wie z. B. einfache Werte wie Zahlen und Zeichen aber auch Datenstrukturen wie Felder (arrays), Verbunde (records) aber auch Prozeduren und Funktionen

[8] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, 1985
Mathematisch gesprochen bildet die Menge aller Ideale einen Verband, wenn die Ordnung durch die Mengeninklusion definiert ist.

Die Obermenge aller Typen (Ideale) ist der zur Menge W gehörige Typ.

Die Aussage

„Ein Wert oder Ausdruck besitzt einen Typ (ist von einem Typ).“

bedeutet demnach, dass der Wert oder Ausdruck Element der dem Typ zugeordneten Menge ist. Da die Ideale von W nicht notwendig einen leeren Durchschnitt besitzen, kann ein Wert auch mehr als einen Typ besitzen. Die Konstante `Montag` ist sowohl vom Typ `Wochentag` als auch vom Typ `Arbeitstag` (s. o).

Ein Typsystem, in dem alle Werte immer zu genau einem Typ gehören, bezeichnet man als *monomorph*, Typsysteme, in dem Werte mehr als einem Typ angehören können, heißen *polymorph*.

Das Typsystem einer Programmiersprache wie PASCAL wird aber im allgemeinen nicht als polymorph bezeichnet, weil die Werte in der Regel nur von einem einzigen Typ sind. Konstanten eines Teilbereichstyps stellen eine Ausnahme von dieser Regel dar.

Die Typsysteme von Programmiersprachen sind aber meist gar nicht eindeutig den Kategorien *monomorph* oder *polymorph* zuzuordnen. Sie unterscheiden sich eher im „Grad an Polymorphie“, d. h. darin wie groß (und wie interessant) die Mengen von Werten sind, die mehr als einem Typ angehören können.

Da Typen Mengen sind, entsprechen Subtypen (Untertypen) Teilmengen (Untermengen). Der Typ `Arbeitstag` ist Subtyp von `Wochentag`, weil die Menge der Werte von `Arbeitstag` Teilmenge der Werte von `Wochentag` sind.

Auf die Begriffe *Subtypen*, *Typhierarchie* und *Vererbung* wird in einem weiteren Arbeitspapier eingegangen werden.

Das Typsystem einer Programmiersprache bestimmt, welche Teilmengen von W legale Typen sind. Die Menge der Typen ist in der Regel nur eine kleine Teilmenge der möglichen Teilmengen[9] von W. Die Programmiersprache legt durch ihre Syntax fest, welche Typen beschrieben werden können und wie sie auf Ideale abgebildet werden. Eine Sprache, die nur einen einzigen Typ W kennt, kann auch als typfreie Sprache bezeichnet werden. „Getypte“ Sprachen erlauben Typen auf unterschiedliche Weise zu benennen, durch

- Bezeichner für Basistypen[10],
- Typkonstruktoren für
  - Datenstrukturen[11] oder
  - Funktionen[12]

um nur einige Beispiele zu nennen. Das Ziel der „Typsprache“ einer Programmiersprache besteht darin, dem Programmierer zu ermöglichen, alle „interessanten“ Typen zu benennen, d. h. diejenigen, die in einem bestimmten Kontext von Bedeutung sind. Weil diese Typen aber schwerlich beim Sprachentwurf umfassend festlegbar sind, stellen viele Programmiersprachen eben Typkonstruktoren bereit, die aus vorhandenen Typen neue zu erbauen erlauben.

[9] Ideale

[10] Integer, Character, Boolean etc

[11] z. B.: listOf Integer

[12] z. B.: listOf Integer → Integer

Die mengenorientierte Sicht auf Datentypen hat sicher auch der Programmierer im Blick, wenn er einen Aufzählungstyp definiert. Bei der Definition von komplexeren Typen mittels Typkonstruktoren steht aber wohl eher eine konstruktive Sicht im Vordergrund. Es geht um die Rekonstruktion von Gegenständen eines „Weltausschnitts“ im Rechner. Die zu einem so konstruierten Typ gehörende Teilmenge von W ist dabei eher nicht Gegenstand der Betrachtung.

3 *Konstruktive Sicht auf Datentypen*

Seit langem bieten imperative Programmiersprachen Unterstützung für die Bildung von zusammengesetzten (strukturierten) Datentypen. Hierzu gehören zum einen die sogenannten *Reihungen* (Arrays), die in erster Linie dazu dienen, die mathematischen Strukturen *Vektor* und *Matrix* nachzubilden. Zum anderen ermöglichen die sogenannten *Verbünde* (Records) die Bildung von Strukturen, die

Die Möglichkeit für den Programmierer, Typen zu „bauen“, stammt wesentlich aus den Programmiersprachen Algol W und Algol 68 und wurde später z. B. in Pascal aufgegriffen.

aus meist wenigen Komponenten unterschiedlichen Typs zusammengesetzt sind.

Auch das Klassenprinzip aus objektorientierten Programmiersprachen kann als ein Mittel der Typkonstruktion betrachtet werden (s. Abschnitt 3.4), obwohl es weit darüber hinaus geht.

Wie für jeden Typ müssen auch für strukturierte Typen der Wertebereich und die zulässigen Operationen definiert werden. Bezüglich des Wertebereichs ist festzulegen, wie die strukturierten Typen aus ihren Komponenten, die ebenfalls strukturiert sein können, zusammengesetzt sind. Die „kleinsten" Komponenten sind letztlich immer von einem skalaren Datentyp. Die vordefinierten Operationen auf strukturierten Typen beschränken sich fast ausschließlich darauf, auf die Komponenten einer Struktur zugreifen zu können. Weitergehende Operationen müssen mithilfe der für die skalaren Typen definierten Operationen programmiert werden.

Für die Bildung solcher Strukturen werden dem Programmierer *Typkonstruktoren* zur Verfügung gestellt. Typkonstruktoren erlauben das „Bauen" von neuen Typen. Der Begriff ist nicht mit den aus objektorientierten Programmiersprachen bekannten Konstruktoren (für die Erzeugung von Exemplaren von Klassen) zu verwechseln.

An dieser Stelle werden zunächst nur die *statischen* strukturierten Datentypen behandelt. Bei statischen Strukturen ist der Aufbau aller Komponenten bereits zur Übersetzungszeit festgelegt, weshalb der benötigte Speicherplatz bereits durch den Compiler (d. h. statisch) ermittelt und reserviert werden kann. Ist dies erst zur Laufzeit des Programms möglich, spricht man von *dynamischen* Datenstrukturen bzw. Datentypen. [13]

[13] Dynamische Strukturen werden in Abschnitt 3.5 behandelt.

Historisch betrachtet wurde die konstruktive Sicht auf Datentypen durch Programmiersprachen, die in den sechziger Jahren des vorigen Jahrhunderts entwickelt wurden – wie z. B. Algol W, Algol 68 – eingeführt. Diese Entwicklungen hatten großen Einfluss auf Sprachen der siebziger und achtziger Jahre. Hier seien PASCAL, C und ADA genannt.

3.1 *Reihung (Array)*

Mithilfe einer Reihung wird es möglich, endliche angeordnete Mengen von Komponenten gleichen Typs sozusagen in einem zusammenhängenden „Schrank" mit vielen „Schubladen" unterzubringen. Abbildung 1 zeigt einen Schrank namens `temperaturen` mit zehn Schubladen für die Unterbringung von acht Gleitkommahlen. Die Schubladen sind mit einer *Index* genannten Nummer (hier von 0 bis 7) versehen.

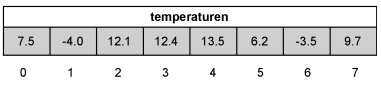

Abbildung 1: Die Reihung (Array) `temperaturen`

Der Zugriff auf die einzelnen Komponenten einer Reihung erfolgt durch die Angabe des Index. In den Sprachen der Pascal-Familie – aber auch in C und Java – geschieht dies, indem der Index eingeschlossen in eckige Klammern hinter dem Bezeichner der Reihung angegeben wird. Auf die Komponente mit dem Index 5 der Reihung `temperaturen` wird mit `temperaturen[5]` zugegriffen. Jede solche Komponente („Schublade") heißt *indizierte Variable*[14]. Die Anzahl der Komponenten heißt *Länge des Feldes*. Der Index kann ein beliebiger Ausdruck sein, der einen zulässigen Indexwert liefert.

[14] Indizierte Variablen sind Variablen. Ihnen können daher Werte zugewiesen werden, und sie können Werte liefern, d. h. in Ausdrücken verwendet werden.

Die Deklaration der Array-Variablen[15] `temperaturen` sähe in

[15] Man sagt, die Variable ist von einem *Array-Typ*. Der Typ der Komponenten (hier: `REAL` bzw. `float`) heißt auch *Komponententyp*.

OBERON folgendermaßen aus:

```
VAR temperaturen: ARRAY 8 OF REAL;
```

In C-Syntax:
```
float temperaturen[8]
```

In OBERON und in C wird neben dem Komponententyp die Anzahl der Komponenten spezifiziert. Die Komponenten werden implizit mit 0 beginnend nummeriert[16].

[16] Andere Programmiersprachen (Ada, PASCAL, MODULA-2) lassen als Indextyp jeden geordneten, diskreten skalaren Typ zu, also Integer, Aufzählungstypen einschließlich Character und Boolean.

Aus der konstruktiven Sicht auf den Typbegriff sind im Zusammenhang mit Reihungen zwei Aspekte bedeutsam:

1. Die Definition einer Reihung erlaubt neben dem Zugriff auf die einzelnen Komponenten aber auch die Behandlung als ein (zusammengesetztes) Gebilde. So kann eine Reihung durch Angabe ihres Bezeichners z. B. einer Prozedur als Parameter übergeben werden.
2. Die mithilfe von Typkonstruktoren definierten Typen können auch benannt, d. h. mit einem eigenen Bezeichner versehen werden, wie im folgenden Beispiel erläutert wird.

Die Stärke der Reihung besteht in dem effizienten Zugriff auf die Komponenten durch Indizierung, die es dem Laufzeitsystem ermöglicht, die Speicheradresse einer Komponenten auf einfache Weise zu berechnen, da alle Komponenten vom gleichen Typ sind und damit gleich viel Speicherplatz belegen.

Beispiel: (OBERON)

```
TYPE
   TempFeld = ARRAY 8 OF REAL;
   String = ARRAY 256 OF CHAR;
```

Es ist zum einen guter Programmierstil, strukturierte Datentypen mit aussagekräftigen Namen zu versehen, zum anderen bedeutet Typkompatibilität in einigen imperativen Programmiersprachen die Gleichheit der Typnamen. Um z. B. an eine Array-Variable den Wert einer anderen Array-Variablen zuweisen zu können, genügt es nicht, dass die Typen der Variablen strukturgleich sind - also Arrays gleicher Länge mit gleichem Komponententyp - sondern die Typ**namen** müssen übereinstimmen. Dies ist nur durch eine entsprechende Typdeklaration möglich.

Die neu definierten Typbezeichner `TempFeld` und `String` können nun in Variablendeklarationen genauso wie die vordefinierten Typbezeichner der Basistypen benutzt werden. Die Deklaration der Variablen `temperaturen` sieht dann so aus:

```
VAR temperaturen : TempFeld;
```

Typprüfung bei Reihungen: Zu den häufigsten Programmierfehlern überhaupt gehört der Zugriff auf eine Reihung mit einem Index außerhalb des zulässigen Bereichs.

Beispiel (C):

```
int a[10];
for (i = 0; i <= 9; i++)
   a[i + 1] = a[i];
```

Sei unmittelbar hinter dem Array `a` eine weitere Variable deklariert:

```
int a[10];
float b;
for (i = 0; i <= 9; i++)
   a[i + 1] := a[i];
```

Da in der Regel der Compiler die Variable `b` im Speicher unmittelbar hinter dem Array anordnen wird, verändert die Schleife die Variable `b`.

Bei der letzten Iteration wird auf ein Array-Element zugegriffen, das nicht existiert. Derartige Fehler im Programm sind schwer zu finden, da durch derartige Zugriffe Speicherzellen verändert werden, die mit der Reihung nichts zu tun haben und die im Extremfall nicht mal zum Programm gehören.

Grundsätzlich handelt es sich beim Zugriff auf eine Reihung mit einem Index, der nicht zur Reihung gehört, um einen *Typfehler*. Derartige Typfehler können aber auch bei einem statischen Typsystem in der Regel erst zur Laufzeit festgestellt werden. Ein strenges

Typsystem muss jeden zur Laufzeit berechneten Index auf Kompatibilität mit dem Indextyp prüfen.[17] Erst dadurch wird jeder Versuch, auf eine Reihung mit einem unzulässigen Index zuzugreifen, mit einem Laufzeitfehler und damit mit einem Programmabbruch quittiert.

[17] PASCAL war die erste Programmiersprache, in der dies umgesetzt wurde. Falls der Wert eines Index schon zur Übersetzungszeit bekannt ist, kann schon der Compiler die Zulässigkeit überprüfen. Prüfungen zur Laufzeit kosten selbstverständlich Rechenzeit.

ZEICHENKETTEN können im Grunde genommen als Reihungen betrachtet werden, deren Komponenten vom Typ Character sind. Sofern es sich dabei um statische Arrays handelt, muss man sich aber bei der Deklaration einer Variablen auf die Länge der Zeichenketten festlegen.[18] Da man es aber in fast allen Anwendungen mit variabel langen Zeichenketten zu tun hat, wird in vielen Programmiersprachen die Möglichkeit geschaffen, Variablen von einem speziellen Zeichenkettentyp zu deklarieren.

[18] Führte man z. B. in PASCAL den Typ `String = Array 12 of Char` ein, enthielte jede Variable vom Typ `String` eine Zeichenkette aus 12 Zeichen.

Im Folgenden wird exemplarisch die Behandlung von Zeichenketten in Component Pascal[19] dargestellt, die in den Grundzügen der in C ähnelt.

[19] Component Pascal ist eine Weiterentwicklung von OBERON. Zu OBERON vgl. z. B.

Hanspeter Mössenböck. *Objektorientierte Programmierung in Oberon-2*. Springer-Verlag GmbH, 2. edition, 1994

Zu Component Pascal vgl. z. B.

J. Stanley Warford. *Computing Fundamentals*. Vieweg Verlag, 2002

In Component Pascal werden Zeichenketten als *Arrays of characters* dargestellt. Im folgenden Beispiel wird ein Datentyp für Zeichenketten der Länge `StrngLng` definiert.

```
CONST
   StrgLng = 256;
TYPE String = ARRAY StrgLng OF CHAR;
```

Durch die Konstante `StrgLng` wird aber nur die Maximallänge der Zeichenketten festgelegt. Sie werden durch das sogenannte *Null-Zeichen* (null character, hexadezimal: 0X) abgeschlossen. Daher können Variablen des oben definierten Typs `String` maximal 255 Zeichen aufnehmen. Es können aber auch kürzere Zeichenketten zugewiesen werden, ohne dass Typregeln verletzt werden.

Nach der Deklaration

```
VAR s1, s2: String
```

ist die Zuweisung `s1 := s2` zulässig. Die Wirkung besteht darin, dass das vollständige Array `s2` in das Array `s1` kopiert wird. Häufig ist es aber sinnvoll, nur die „gültigen" Zeichen, d. h. bis zum ersten Nullbyte, von `s2` nach `s1` zu kopieren. Dies kann durch den $-Selektor erfolgen. Die Zuweisung lautet dann:

```
s1 := s2$
```

Es führt zu einem Laufzeitfehler, wenn `s2` kein Nullbyte enthält.

DIE EINZIGE OPERATION, die für Reihungen vordefiniert ist, besteht im Zugriff auf ihre Komponenten durch Indizierung. Selbst die Prüfung auf Gleichheit zweier Reihungen ist z. B. in PASCAL auf Zeichenketten[20] beschränkt. Da Reihungen aber als Parameter an Prozeduren übergeben werden können, ist es möglich, komplexere Operationen auf Reihungen als solche zu programmieren. Diese Prozeduren sind aber nicht Bestandteil des Reihungs-Typs.

[20] in ISO 7185 Pascal müssen Strings als `packed array [1 .. n] of char` deklariert werden

3.2 *Verbund (Record)*

Die Komponenten einer Struktur vom Typ Reihung (Array) haben immer alle den gleichen Typ. Für viele Anwendungen ist es aber nützlich, auch Komponenten unterschiedlichen Typs zu einer Struktur zusammenfassen und unter einem Namen gemeinsam ansprechen zu können. Solche Strukturen sind insbesondere in kommerziell-administrativen Anwendungen häufig anzutreffen. Wenn z. B. eine rechnergestützte Kundenverwaltung realisiert werden soll, ist zu überlegen, durch welche Merkmale ein Kunde beschrieben werden soll. Diese Merkmale werden dann in einer Struktur, einem sogenannten *Verbund*, zusammengefasst, wobei die Merkmale unterschiedliche Datentypen erfordern können.

Beispiel: Ein Kundenverwaltungsprogramm soll über die Kunden die Merkmale *Name, Vorname, Adresse, Umsatz* verwalten. Im Folgenden werden geeignete Typdefinitionen (in OBERON-Syntax) angegeben:

```
CONST
   Namlng = 30;
TYPE
   NameT = ARRAY Namlng OF CHAR;
   HausnrT = ARRAY 5 OF CHAR;
   AdressT = RECORD
      Hausnummer : HausnrT;
      Strasse    : NameT;
      Ort        : NameT;
  END;
   KundeT = RECORD
      Name: NameT;
      Vorname: NameT;
      Adresse: AdressT;
      Umsatz : INTEGER
   END;
```

In C-Syntax sähen die Verbunddefinitionen folgendermaßen aus:

```
#define int Namlng = 30;
typedef char NameT[Namlng];
typedef char HausnrT[5];
typedef struct {
      HausnrT Hausnummer;
      NameT   Strasse;
      NameT   Ort;
} AdressT;
typedef struct {
      NameT Name;
      NameT Vorname;
      AdressT Adresse;
      int Umsatz;
 } KundeT;
```

Ein Verbundtyp wird durch das Schlüsselwort `RECORD` eingeleitet und durch `END` abgeschlossen. Dazwischen erfolgt die Deklaration der Komponenten. Jede Komponentendeklaration besteht aus einem Komponentenbezeichner (z.B. `Umsatz`) gefolgt von einem Doppelpunkt gefolgt von dem Komponententyp. Nach einer Variablendeklaration der Form

```
kunde : KundeT
```

kann auf die Komponenten der Verbundvariablen `kunde` mithilfe der sogenannten Punktnotation zugegriffen werden.

So wird der Komponenten `Umsatz` durch

```
kunde.Umsatz := 175000
```

der Wert 175000 zugewiesen.

Man beachte, dass die Komponenten eines Verbunds selbst auch wieder Verbünde sein können. Allgemein gesprochen: Der Typ einer Komponente ist frei wählbar, jeder einfache oder strukturierte

Typ kann benutzt werden. So ist die Komponente `Adresse` des Verbundtyps `KundeT` selbst wieder ein Verbund vom Typ `AdressT`. Dieser wiederum besitzt Komponenten, die von einem Array-Typ sind (`NameT, HausnrT`).

Variablen desselben Verbundtyps[21] können einander zugewiesen werden. Die Prüfung auf Gleichheit ist aber nicht ohne weiteres möglich.

[21] die Typnamen müssen übereinstimmen

Selbstverständlich können Verbünde auch als Komponenten von Arrays benutzt werden. Die folgende Variablendeklaration legt z.B. ein Array von 500 Kunden an:

```
kunden : ARRAY 500 OF KundeT
```

Um nun die Adresse des Kunden mit dem Index 27 festzulegen, könnten die nebenstehende Zuweisungen benutzt werden.

```
kunden[27].Adresse.Hausnummer := '7a';
kunden[27].Adresse.Strasse := 'Feldweg';
kunden[27].Adresse.Ort := 'Irgendwo';
```

Aus den letzten Beispielen wird deutlich, dass mit den Typkonstruktoren *Reihung* und *Verbund*[22] beliebig komplexe Datenstrukturen erbaut und mit einem Typbezeichner versehen werden können.

[22] Verbünde sind auch für den Aufbau von dynamischen Datenstrukturen, wie z. B. verkettete Listen oder Bäume, von großer Bedeutung. Vergleiche dazu Abschnitt 3.5.

Für Operationen auf Exemplaren von Verbundtypen gilt im Grunde das Gleiche wie für Reihungen: Vordefiniert ist nur der Komponentenzugriff. Komplexere Operationen können wiederum als Prozeduren definiert werden, den die zu verarbeitenden Verbundexemplare als Parameter übergeben werden. Diese Prozeduren sind aber selbst nicht Bestandteil des Verbundtyps.

Diese Einschränkung wird z. B. in OBERON aufgegeben, indem es möglich ist, Records mit Prozedurfeldern zu definieren.

3.3 Verbünde mit Prozedurfeldern

Schon in PASCAL ist es möglich, einer Prozedur eine andere Prozedur als Parameter zu übergeben, d. h. Prozeduren mit einem formalen Parameter von einem Prozedurtyp zu deklarieren. Der Prozedurtyp ist notwendig, weil in einer statisch getypten Sprache alle Bezeichner[23] unter Angabe eines Typs deklariert werden müssen. Die (willkürliche) Beschränkung von PASCAL, nur formale Parameter mit einem Prozedurtyp versehen zu können, wurde In OBERON aufgegeben. Dadurch wird es nicht nur möglich, Prozedurtypen in eine Typvereinbarung zu übernehmen, sondern auch Prozedurvariablen deklarieren zu können.

[23] und damit auch die formalen Parameter einer Prozedur oder Funktion
Spätestens an dieser Stelle wird die synonyme Verwendung der Begriffe *Typ* und *Datentyp* fragwürdig. Nicht nur Daten besitzen einen Typ sondern auch Programme in Form von Prozeduren, Funktionen oder Methoden. Zur Menge aller möglichen Bitmuster eines ungetypten Universums gehören sie ohnehin.

Beispiel einer Prozedurtypdeklaration:

```
TYPE
   RealFunct = PROCEDURE(x: REAL): REAL;
```

Ein Prozedurtyp entspricht der Signatur einer Prozedur, d. h. er enthält:

- Anzahl und Typen der formalen Parameter
- Im Falle von Funktionen: Ergebnistyp

Der Typ `RealFunct` kann nun z. B. zur Deklaration eines formalen Parameters benutzt werden:

```
PROCEDURE Nullstelle(f: RealFunct; x1, x2: REAL): REAL;
```

Beispiel für die Verwendung eines Prozedurparameters: Traversieren einer Listenstruktur

```
PROCEDURE Trav(first:Node;
               P:PROCEDURE(n:Node));
BEGIN
    WHILE first # NIL
    DO P(first);
       first := first.next
    END
END Trav
```

Die Verwendung des Prozedurparameters `P` ermöglicht es, die Prozedur, die die Wirkung auf die einzelnen Listenknoten definiert, an anderer Stelle zu deklarieren. Dadurch ist die Prozedur `Trav` allgemein verwendbar, der Algorithmus für das Traversieren der Listenstruktur ist unabhängig von der Aktion für jedes einzelne Element der Liste.

Der Bezeichner einer gewöhnlichen Prozedurdeklaration[24] kann als benannte Konstante aufgefasst werden, die für eine Anweisungsfolge (den Prozedurrumpf) steht. Mithilfe von Prozedurtypen können *Prozedurvariablen* wie folgt deklariert werden:

[24] z. B. `Nullstelle`

```
VAR
   f, g, trig: RealFunct;
   combinatorial: PROCEDURE(x:INTEGER):INTEGER;
```

Prozedurvariablen[25] erhalten Werte durch normale Wertzuweisungen. Werte, die an Prozedurvariablen zugewiesen werden können, werden durch normale Prozedurdeklarationen (Prozedurkonstanten) eingeführt.

[25] Prozedurvariablen können auf Gleichheit geprüft werden.

BEISPIELE:

```
PROCEDURE Square(x:REAL):REAL;
BEGIN
    RETURN x*x
END Square;

PROCEDURE Quadratic(a,b,c,x:REAL):REAL;
BEGIN
    RETURN a*x*x+b*x+c
END Quadratic;
```

Mit den oben deklarierten Variablen ergeben sich folgende mögliche Zuweisungen:

```
f := Square;
g := f;
q := Quadratic;
trig := Math.sin;
```

Der Aufruf einer Prozedur mittels einer Prozedurvariablen unterscheidet sich nicht von dem mittels einer Prozedurkonstanten, z. B. `v := f(2.0)`.

Eine weitere Anwendungsmöglichkeit von Prozedurtypen stellt nun die Deklaration von Records mit Prozedurfeldern dar.

Als Beispiel sei die Deklaration eines Record-Typs `Point`

```
TYPE
  Point = RECORD
              x: INTEGER;
              y: INTEGER;
              translate: PROCEDURE (dx, dy: INTEGER);
          END;
```

gegeben, der mit `translate` ein Feld von einem Prozedurtyp enthält. Nach dem Anlegen eines `Point`-Exemplars durch die Deklaration einer Variablen dieses Typs kann nun dem Prozedurfeld eine (anderswo definierte) Prozedurkonstante zugewiesen und diese dann z. B. so

`VAR p: Point`

z. B. `p.translate := TranslatePoint`

```
p.translate(15, 30)
```

aufgerufen werden.

Mit dieser Technik wird es möglich, mit dem Typkonstruktor `RECORD` nicht nur eine Datenstruktur, sondern zugleich auch (als Bestandteil des neuen Typs) Operationssignaturen auf Exemplaren des Typs zu definieren. Durch Records mit Prozedurfeldern sind im Prinzip die Voraussetzungen für die Einführung eines Objektkonzepts geschaffen, wenn man unter einem *Objekt* in erster Linie ein „Ding“ versteht, dass sowohl eine Datenstruktur als auch ein Verhalten besitzt. Man beachte, dass hier verschiedene „Objekte“ des gleichen Typs unterschiedliches, individuelles Verhalten aufweisen können. Diese Betrachtungsweise von Objekten steht nicht im Einklang mit dem Objektbegriff der meisten objektorientierten Programmiersprachen[26]. Dort ist das Verhalten an die Klasse[27] und nicht an das einzelne Exemplar gebunden. Alle Exemplare einer Klasse weisen das gleiche Verhalten auf.

An zwei verschiedene Exemplare des Typs `Point` könnten unterschiedliche `translate`-Prozeduren zugewiesen werden.

[26] z. B. Java, C#, C++ oder Smalltalk

[27] Die Klasse entspräche hier dem Typ.

Die Weiterentwicklung der Sprache OBERON zu OBERON-2[28] führt die Möglichkeit ein, Prozeduren anstatt an Objekte (Records) an Objekttypen (Record-Typen) zu binden. Diese sogenannten *typgebundenen Prozeduren* werden dann auch als *Methoden* und die zugehörigen Record-Typen als *Klassen* bezeichnet, wie es der für objektorientierte Programmiersprachen üblichen Terminologie entspricht.

[28] Hanspeter Mössenböck. *Objektorientierte Programmierung in Oberon-2*. Springer-Verlag GmbH, 2. edition, 1994

3.4 *Klassen*

Der Begriff der *Klasse*[29] wird in der Regel den objektorientierten Programmiersprachen zugeordnet, die eigentlich eine abstraktionsorientierte Sicht auf Datentypen[30] pflegen. Der Klassenbegriff von Java[31] beispielsweise ist vielfältig. Einerseits dienen sie im Sinn der objektorientierten Programmierung zur Definition der Struktur und des Verhaltens ihrer Exemplare, den Objekten. Andererseits können Klassen aber auch als ein Mittel zur Modularisierung genutzt werden, indem sie als „Behälter" für statische Methoden dienen und keine Objektstruktur definieren. In diesem Fall steht die funktionale Abstraktion im Vordergrund. Schließlich ist es auch möglich, mithilfe von Klassen Objektstrukturen ohne Verhalten zu definieren. In diesem Fall würden Klassen als Typkonstruktoren ähnlich wie Verbünde benutzt, was an dieser Stelle kurz erläutert werden soll.

[29] erstmals eingeführt durch die Programmiersprache Simula 67

[30] vgl. Abschnitt 4

[31] Der Klassenbegriff ist in der objektorientierten Programmierung leider nicht einheitlich definiert.

Aus dem Blickwinkel der objektorientierten Programmierung ist dies keine adäquate Nutzung von Klassen, vgl. z.B.
Chenglie Hu. Dataless objects considered harmful. *Commun. ACM*, 48(2):99–101, 2005

Die Betrachtung von Klassen als Typkonstruktoren[32] ist ebenfalls nicht unumstritten, weil es sich dabei um eine Verkürzung des Klassenbegriffs handelt. Aspekte wie Kapselung und Vererbung fallen zunächst unter den Tisch. Allerdings scheint diese Sichtweise neuerdings durch Diskussionen[33] um den richtigen Weg in der Programmiergrundausbildung an Bedeutung zu gewinnen. Dabei geht es um die Kritik an dem in den letzten Jahren vorherrschenden Ansatz, der unter dem Motto „Objektorientierung von Anfang an" stand. Die Kritiker befürworten eine Rückkehr zur prozeduralen Programmierung in der Anfängerausbildung.

[32] Chenglie Hu. Just say 'a class defines a data type'. *Commun. ACM*, 51(3):19–21, 2008

[33] Stuart Reges. Back to basics in cs1 and cs2. *SIGCSE Bull.*, 38(1):293–297, 2006; Stuart Reges and Marty Stepp. *Building Java Programs:A Back to Basics Approach*. Addison-Wesley, Mai 2007; and Robert Sedgewick and Kevin Wayne. *Introduction to Programming in Java: An Interdisciplinary Approach*. Addison-Wesley, 2008

Die folgende Definition der Klasse Point

```
class Point
{    int x;
     int y;}
```

entspricht weitgehend der Record-Definition aus dem vorigen Abschnitt. Der Klassenbezeichner `Point` ist danach als Typbezeichner zur Deklaration von Variablen verwendbar:

```
Point p1, p2;
```

Verglichen mit der Deklaration von Verbundvariablen in PASCAL MODULA-2 oder OBERON gibt es einen wichtigen technischen Unterschied. Während die Deklaration einer Verbundvariablen in diesen Sprachen zur Erzeugung eines Exemplars des Verbundtyps führt, reserviert der Java-Compiler lediglich Speicherplatz für einen Verweis auf eine `Point`-Struktur. Das Erschaffen einer `Point`-Struktur erfordert die Anwendung des `new`-Operators, z. B. in dieser Weise:

```
p1 = new Point();
```

Die Notwendigkeit der Benutzung des `new`-Operator hängt damit zusammen, dass in Java für alle Variablen, die nicht von einem Basistyp sind, Zeigersemantik verwendet wird.

Erst danach ist der Zugriff auf die Komponenten mit der Punktnotation möglich:

```
p1.x = 20; p1.y = 30;
```

Eine äquivalente[34] Definition des `Point`-Typs sähe in OBERON so aus:

[34] im Sinne der Zeigersemantik

```
TYPE Point = POINTER TO
               RECORD
                 x: INTEGER; y: INTEGER
               END;
```

Auch hier müsste nach der Deklaration der Variablen

```
VAR p1, p2 : Point
```

vor dem Zugriff auf die Komponenten der `NEW`-Operator angewendet werden:

```
NEW(p1)
```

Selbstverständlich ist das Hinzufügen von Methoden[35] zu einer Klassendefinition möglich und sinnvoll, wie z. B. der `translate`-Methode:

[35] typgebundene Prozeduren, vgl. Abschnitt 3.3

```
class Point
{   int x;
    int y;
    void translate(int dx, int dy) {
        x += dx;
        y += dy;}
}
```

3.5 *Zeigertypen*

Zur Konstruktion von dynamischen Datenstrukturen[36] sind Verbünde und Reihungen allein nicht geeignet. Sie können in den herkömmlichen imperativen Programmiersprachen in der Regel nur zum Aufbau von statischen Datenstrukturen[37] benutzt werden. Zu den wichtigsten dynamischen Datenstrukturen gehören „verkettete“ Strukturen wie *lineare Listen* oder *Bäume*. Diese werden rekursiv definiert und damit können Listen mit beliebig vielen Elementen oder Bäume mit beliebig vielen Knoten entstehen. Dazu muss man in die Definition eines Verbunds für ein Listenelement lediglich eine Komponente aufnehmen, die auf das nächste Listenelement verweist. Um solche Verweise realisieren zu können, muss in den imperativen Programmiersprachen ein neuer Typ der Art „Verweis auf etwas“ eingeführt werden. Diese Verweise werden auch *Zeiger*[38] genannt. Variablen, die Verweise enthalten, sind von einem *Zeigertyp*[39].

[36] Damit sind hier Datenstrukturen gemeint, die zur Laufzeit wachsen und schrumpfen können.

[37] Deren Größe wird zur Übersetzungszeit ermittelt.

[38] engl. *pointer*

[39] engl. *pointer typ*

Vereinfacht gesprochen ist eine Variable nichts anderes als eine symbolische Speicheradresse, die es dem Programmierer erspart, sich mit den numerischen Adressen des realen Arbeitsspeichers zu befassen. Die Namen der Variablen werden vom Programmierer durch Deklaration festgelegt und vom Compiler beim Übersetzungsvorgang in numerische Adressen umgewandelt. Damit sind

In den funktionalen und den objektorientierten Sprachen sind alle Variablen Zeigervariablen, wodurch der Aufbau von dynamischen Datenstrukturen ohne weiteres – für den Programmierer völlig transparent – möglich ist.

die Namen von Variablen bzw. die Adressen, die sie repräsentieren, für den Programmierer nicht zugänglich. Zum Aufbau sogenannter dynamischer Datenstrukturen ist dies aber notwendig. Um aber zu vermeiden, dass der Programmierer mit numerischen Adressen hantieren muss, wurden die Zeigertypen[40] eingeführt. Ein Wert von einem Zeigertyp ist technisch gesehen eine Speicheradresse. Eine *Zeigervariable* enthält demnach die Adresse einer anderen Variablen.

[40] erstmals in PL/I

In C-Syntax werden durch

```
int i = 15;
int *zeiger = &i;
```

Die C-Syntax ist hier etwas verwirrend. Durch `int *zeiger` wird der Variablen `zeiger` der Typ *pointer to int* zugeordnet. Der Operator & liefert die Adresse seines Operanden.

eine Integer-Variable `i` und eine Zeigervariable `zeiger`, die die Adresse von `i` als Wert enthält, deklariert. Die Variable `i` ist die von der Zeigervariablen `zeiger` *referenzierte Variable*. Der Typ `int` ist der *Bezugstyp* der Zeigervariablen. Abbildung 2 stellt den Zusammenhang zwischen beiden Variablen unter der Annahme dar, dass der Compiler der Variablen `zeiger` die numerische Adresse 584 und der Variablen `i` die Adresse 1023 zugeordnet hat.

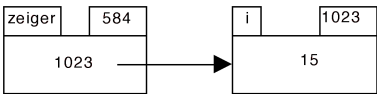

Abbildung 2: Zeigervariable und referenzierte Variable

Auf die Variable `i` kann nun entweder (ganz normal) über ihren Namen zugegriffen werden oder durch *Dereferenzierung* über die Zeigervariable.

In C-Syntax geschieht dies durch den vorangestellten Stern. Durch `*zeiger` wird also auf die Variable `i` zugegriffen.

Anmerkung: Die hier verwendeten Begriffe *Referenz*, *referenzieren* usw. sind eigentlich Resultate einer falschen Übersetzung des englischen *reference*, was eher mit *Verweis* übersetzt werden müsste. In diesem Abschnitt folgen wir „ausnahmsweise“ dem üblichen Sprachgebrauch.

Die Zeigervariable `zeiger` ist ein sogenannter *getypter* Zeiger, da er von der Intention her nur auf Integer-Objekte zeigen kann. Das bedeutet, dass verlangt wird, dass `*zeiger` vom Typ Integer ist und dass die Variable `zeiger` nach Dereferenzierung überall dort, wo eine Integer-Variable erwartet wird, verwendet werden kann. In C gibt es aber auch *ungetypte* Zeiger, die durch

```
void *void_ptr
```

deklariert werden. Ein Zeiger auf `void` kann auf alles zeigen. Jeder Zeiger kann implizit in einen Zeiger auf `void` konvertiert werden und umgekehrt. Dies ist eine Konsequenz der in C nur schwach ausgeprägten Typprüfung. Eine weitere Konsequenz davon ist, dass einer Zeigervariablen ein beliebiger Ausdruck zugewiesen werden kann. Damit liegt die Verantwortung dafür, dass der Wert einer Zeigervariablen eine zum Programm gehörende Adresse repräsentiert, vollständig beim Programmierer.

Die Sprachen der Pascal-Familie erlauben ausschließlich getypte Zeiger und außer der Zuweisung, der Prüfung auf Gleichheit und der Dereferenzierung keine weiteren Operationen. In OBERON sind darüber hinaus als Bezugstypen von Zeigern nur Verbundtypen und Reihungstypen zugelassen.

Die Dereferenzierung von Zeigern, der Zugriff auf Verbundkomponenten und das Indizieren von Reihungen stellen die grundlegenden Operationen für den Zugriff auf Datenstrukturen dar. Die Syntax in den Sprachen der Pascal-Familie ist – verglichen mit C – übersichtlicher, da hier für die drei Operationen eigene Symbole verwendet werden, die jeweils hinter die Variable zu schreiben sind.

Das folgende Beispiel – geschrieben in der Syntax von OBERON – soll dies verdeutlichen:

```
TYPE
   recordT = RECORD
      feld: INTEGER
   END;

   arrayT = ARRAY 100 OF recordT;
   zeigerT = POINTER TO arrayT;

VAR
   zeiger: zeigerT
```

Das Symbol „^ “ wird für die Dereferenzierung benutzt. Durch Hinzufügen weiterer Symbole kann man immer weiter in die Datenstruktur „eintauchen“ :

```
zeiger            (* Pointer auf ein Array von Records mit
                     Integer-Feldern*)
zeiger^           (* Array von Records mit Integer-Feldern*)
zeiger^[25]       (* Record mit Integer-Feld*)
zeiger^[25].feld  (* Integer *)
```

Es ist wichtig, zwischen einem Zeiger und dem Objekt, auf das er zeigt zu unterscheiden. Hierbei ist zu beachten, dass mit den Zeigervariablen bei Zuweisungen die aus objektorientierten Sprachen bekannte Verweissemantik[41] in Erscheinung tritt. Dies sei an einem Beispiel, der Syntax von C folgend, erläutert:

[41] vulgo Referenzsemantik

```
int i1 = 15;
int i2 = 25;
int *zi1 = &i1;
int *zi2 = &i2;
```

Durch die Zuweisung

```
*zi1 = *zi2
```

wird durch Dereferenzierung auf die Integerobjekte `i1` und `i2` zugegriffen und es stellt sich die in Teil (a) von Abbildung 3 dargestellte Situation ein, d. h. die beiden Integervariablen haben den gleichen Wert. Durch die Zuweisung

```
zi1 = zi2
```

bekommt die Zeigervariable `zi1` den gleichen Wert (die gleiche Referenz) wie `zi2`, d. h. beide Zeiger zeigen auf `i2`, wie in Teil (b) von Abbildung 3 gezeigt.

In typstrengen Sprachen sind nur eingeschränkte Operationen mit Zeigern erlaubt:

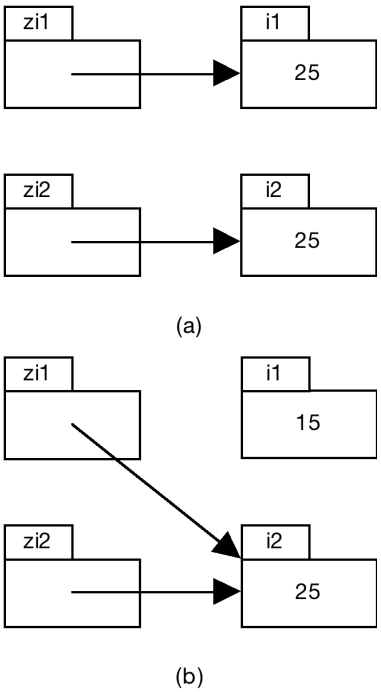

Abbildung 3: Zuweisung von Zeigervariablen

- Zeiger dürfen nur an Zeigervariablen von einem kompatiblen Bezugstyp zugewiesen werden.
- Zeiger können auf Gleichheit oder Ungleichheit geprüft werden. Dies sind die einzigen Ausdrücke in denen Zeiger als Operanden auftreten.

Ein spezieller Zeigerwert wird für Zeiger benutzt, die auf kein Objekt zeigen. Dieser Wert heißt z. B. in OBERON `NIL`[42]. Dieser Wert kann jeder Zeigervariablen zugewiesen werden.

[42] in C: `NULL`

Durch die Deklaration einer Zeigervariablen wird nur der für die Speicherung der Adresse erforderliche Speicherplatz durch den Compiler reserviert, aber nicht der Speicherplatz für die Datenstruktur, auf die der Zeiger zeigt. Betrachten wir dazu folgendes Beispiel (OBERON-Syntax):

```
CONST
   Namlng = 30;
TYPE
   NameT = ARRAY Namlng OF CHAR;
   KundeT = RECORD
      Name: NameT;
      Umsatz : INTEGER
   END;
VAR
   kundeP = POINTER TO KundeT
```

Die Deklaration der Zeigervariablen `kundeP` bewirkt lediglich, dass im Speicher die Zeigervariable angelegt wird, wobei der Inhalt undefiniert ist (vgl. Teil (a) von Abbildung 4). Der Speicherplatz für die dynamische Variable des Bezugstyps (hier: `KundeT`) muss zur Laufzeit durch den Aufruf einer Standardprozedur, die in OBERON `NEW` heißt, angefordert werden. Nach einem Aufruf der Form

```
NEW(kundeP)
```

ergibt sich im Speicher die in Teil (b) von Abbildung 4 dargestellte Situation, d. h. Speicherplatz für die dynamische Variable vom Typ `KundeT` wird reserviert und ihre Adresse in der Zeigervariablen `kundeP` abgelegt.

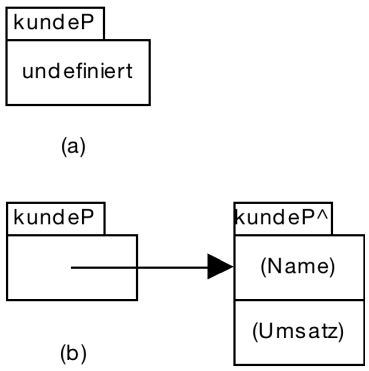

Abbildung 4: Erzeugung einer dynamischen Variablen

Für die Speicherung aller Variablen werden üblicherweise (stark vereinfacht) zwei Speicherstrukturen verwendet. Im sogenannten *Stack* werden alle statischen Größen abgelegt. Hierzu gehören auch die Zeigervariablen. Speicherplatz für die dynamischen Variablen, die mit Hilfe der Prozedur `NEW` angelegt werden, wird hingegen auf dem sogenannten *Heap* reserviert. Trifft man keinerlei Vorkehrungen, wächst der Heap beständig mit der Laufzeit des Programms, wenn man davon ausgeht, dass ein Programm immer wieder Bedarf für neue dynamische Variablen hat. Allerdings treten gewöhnlich auch Situationen ein, wo bestimmte dynamische Variablen

nicht mehr benötigt werden und deren Speicherplatz wieder für andere zur Verfügung gestellt werden könnte. Computersysteme bzw. Laufzeitsysteme für verschiedene Programmiersprachen unterscheiden sich in der Behandlung dieses Problems. Entweder wird dem Programmierer die Verantwortung auferlegt, nicht mehr benötigten Platz auf dem Heap wieder freizugeben.[43] Oder das Laufzeitsystem stellt einen automatischen, *Garbage Collection* genannten, Mechanismus bereit, der dafür sorgt, dass der Heap automatisch bereinigt wird und der Speicherplatz für nicht mehr benötigte Heap-Objekte wieder freigegeben wird.

[43] Hierfür stehen dann wiederum Standardprozeduren wie `FREE`, `DEALLOCATE` oder `DISPOSE` zur Verfügung, die dann mit der entsprechenden Zeigervariablen als Parameter aufgerufen werden müssten.

Die erste Variante[44] erlaubt zwar eine einfachere Implementierung der Sprache, hat jedoch, abgesehen von der Last für den Programmierer, zwei gravierende Nachteile:

1. Der Programmierer könnte versehentlich den Speicherplatz für ein Objekt auf dem Heap vorzeitig freigeben, obwohl es noch von anderen Zeigervariablen referenziert wird. Es kommt zu so genannten „baumelnden Zeigern“[45].
2. Der Programmierer könnte vergessen, den Speicherplatz für ein nicht mehr benötigtes (und nicht mehr zugängliches) Objekt auf dem Heap freizugeben. Es kommt zu so genannten „Speicherlecks“[46].

[44] anzutreffen in den meisten imperativen Programmiersprachen (so z. B. in C oder PASCAL, Ausnahme: OBERON) aber auch in C++

[45] engl.: *dangling pointers*

[46] engl.: *memory leaks*

Automatische Garbage Collection galt lange Zeit als zu aufwändig und auch zu komplex. Inzwischen hat sich diese Einstellung grundlegend gewandelt, so dass dieses Konzept nicht nur in den funktionalen Programmiersprachen, wo dies schon immer üblich war, sondern auch in modernen hybriden Sprachen wie OBERON und Java und in den objektorientierten Sprachen wie Eiffel und Smalltalk realisiert ist.

Die Schwierigkeit besteht darin, einen effizienten Algorithmus zu finden, der in der Lage ist, „Speichermüll“ und gültige Objekte zu unterscheiden.

Die Verwendung von Zeigertypen zur Konstruktion dynamischer Datenstrukturen soll nun am Beispiel einfach verketteter, linearer Listen erläutert werden. Das Ziel einer entsprechenden Typdefinition besteht darin, dass Exemplare dieses Typs ihre Größe dynamisch an die zur Laufzeit eines Programms sich verändernden Anforderungen anpassen können.

Der Speicherbedarf für eine Kundenliste (z. B.) kann wachsen und schrumpfen.

Die grundlegende Technik, um dies zu ermöglichen, besteht darin, Verbünde zu deklarieren, die neben den eigentlichen Datenfeldern (den „Nutzdaten“) Felder mit Zeigern auf weitere Komponenten der Datenstruktur enthalten. Man spricht dann von *rekursiven* Typdefinitionen, da ein Verbund eines Typs einen Zeiger auf eben diesen Typ als Feld enthält, wie das folgende Beispiel[47] zeigt:

[47] die Darstellung erfolgt hier ausschließlich in OBERON-Syntax

```
TYPE
   DataT = ....
   ListP = POINTER TO ListT;
   ListT = RECORD
      data: DataT;
      next: ListP
   END;
VAR
   first: ListP;
```

Das Feld `data` dient hier nur als „Platzhalter“ für die Nutzinformation, die eine reale Anwendung in einer Liste ablegt.

Die Deklaration des Zeigertyps `ListP` allein erzeugt noch keine Liste. Diese muss dynamisch während des Programmablaufes erzeugt werden. Die Zuweisung von `NIL` an die Variable `first` definiert die leere Liste. Eine „längere“ Liste wird durch wiederholte Ausführung der Prozedur NEW erzeugt. Durch die Anweisungsfolge:

```
NEW(first);
NEW(first^.next);
NEW(first^.next^.next);
NEW(first^.next^.next^.next);
```

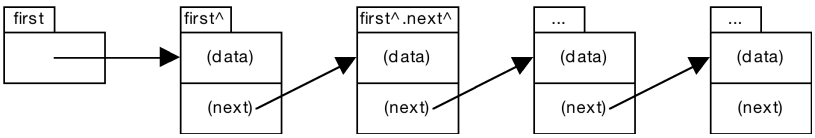

Abbildung 5: Verkettete lineare Liste

entsteht die in Abbildung 5 veranschaulichte Liste. Das Ende der Liste sollte dann wieder durch Zuweisung von `NIL` an den letzten `next`-Zeiger definiert werden:

```
first^.next^.next^.next^.next := NIL
```

Einfacher ist es, neue Listenelemente immer am Anfang der Liste einzufügen. Ausgehend von der leeren Liste wird durch die Anweisungen

```
NEW(first);
first^.data := d1;
first^.next := NIL;
```

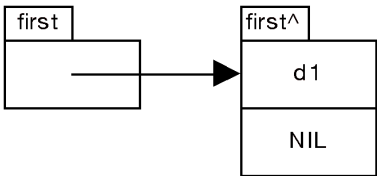

Abbildung 6: Liste mit einem Element

die in Abbildung 6 dargestellte Liste mit einem Element erzeugt.

Sei `new` eine Variable die auf ein neu erzeugtes Listenelement zeigt, das an den Anfang der durch die Variable `first` definierten Liste gesetzt werden soll, dann kann genau dies durch die beiden folgenden Anweisungen geschehen:

```
new.next := first;
first := new;
```

Der Vorgang ist in Abbildung 7 veranschaulicht.

3.6 Zusammenfassung

In der konstruktiven Sicht stellen Datentypen zusammengesetzte Strukturen aus einfacheren Komponenten dar. Der Nutzen für den Programmierer besteht zum einen darin, diese Strukturen mit Typbezeichnern versehen und diese dann zur Deklaration von Variablen nutzen zu können. Zum anderen kann eine Struktur in bestimmten Situationen[48] als Ganzes behandelt werden. Dies

[48] z. B. bei der Übergabe als Parameter

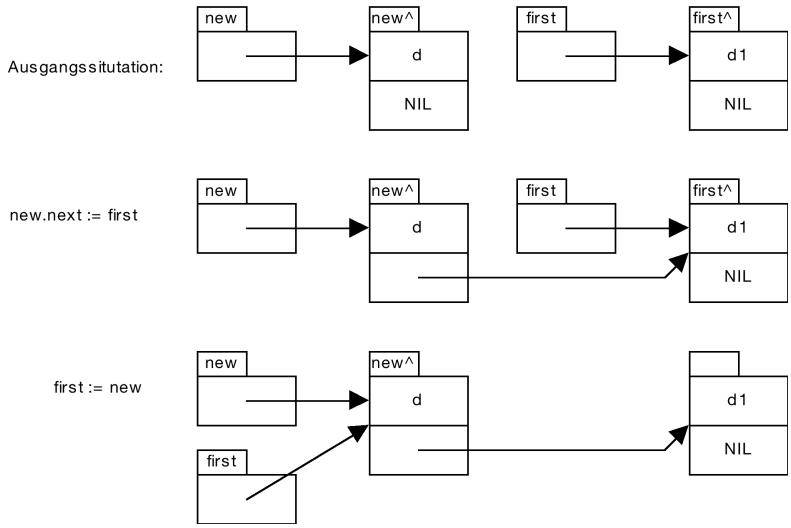

Abbildung 7: Einfügen eines neuen Elementes

erspart dann die möglicherweise mühselige Einzelbehandlung der Komponenten.

Selbstverständlich kann man die Exemplare des Typs `POINTER TO ListT` als die Menge aller Zeiger, die auf ein Exemplar des Bezugstyps `ListT` verweisen, betrachten. Eine solche mengenorientierte Sichtweise ist in diesem Fall ungebräuchlich und auch nicht nützlich.

Wie aus dem „Missbrauch“ von Klassen als Typkonstruktoren im vorigen Abschnitt schon erkennbar wurde, ist der Übergang zu einer abstraktionsorientierten Sicht auf Datentypen fließend und nahe liegend.

4 *Abstraktionsorientierte Sicht auf Datentypen*

4.1 Datenabstraktion

Abstraktion ist das wichtigste Mittel zur Beherrschung von Komplexität. In der Programmiertechnik kann man prozedurale Abstraktion und Datenabstraktion unterscheiden, obwohl zwischen beiden eine Wechselbeziehung besteht. Die Komplexität der Funktionalität eines Programms lässt sich dadurch beherrschen, dass man es hierarchisch in Form von Prozeduren gliedert. Prozeduren stellen aber auch schon eine Form von Datenabstraktion bereit: Der Zugriff auf die lokalen Variablen einer Prozedur ist von außen nicht möglich, sie können nur innerhalb der Prozedur verwendet werden. Der Datenaustausch zwischen einem Klienten und einer Prozedur ist nur über die Prozedurparameter möglich. Allerdings ist die Lebensdauer von lokalen Variablen an die Lebensdauer einer Prozedur-Inkarnation gebunden, was für viele Anwendungsfälle ungeeignet ist. Deswegen hat schon in den sechziger Jahren die Entwicklung leistungsfähiger, flexibler Datenabstraktionskonzepte für Programmiersprachen begonnen; Vorreiter war hier sicherlich Simula[49].

Die von der Programmiersprache möglicherweise angebotene Möglichkeit, globale Variablen zu benutzen, bleibt dabei unberücksichtigt.

[49] Ole-Johan Dahl and Kristen Nygaard. Simula: an algol-based simulation language. *Commun. ACM*, 9(9):671–678, 1966

Der Begriff Datenabstraktion besitzt zwei Aspekte:

1. Die Konstruktion eines Datentyps z. B. in Form eines Verbundtyps ermöglicht die Abstraktion von seinen Komponenten.[50]
2. Die Einschränkung der Sichtbarkeit von Namen der Bestandteile eines Datentyps kann ein weiteres Ziel der Abstraktion sein. Dahinter steckt das Geheimnisprinzip[51], das erstmals von Parnas[52] formuliert wurde.

Im Titel des genannten Aufsatzes von Parnas taucht der Begriff des Moduls auf. Die Gliederung komplexer Systeme in Moduln[53] hat folgende software-technische Zielsetzungen bzw. Vorteile:

1. Die „kognitive Last“, die einem Programmierer zu einem bestimmten Zeitpunkt auferlegt wird, wird dadurch verringert, dass er die Details eines „fremden“ Moduls nicht kennen muss, um das Gesamtsystem oder einen Ausschnitt davon zu verstehen.
2. Das Geheimnisprinzip verhindert, das ein Programmierer in fehlerhafter Weise von der Kenntnis der Implementierungsdetails eines „fremden“ Moduls Gebrauch macht. Das vereinfacht die Fehlersuche.
3. Programmbausteine (die Moduln) können unabhängig von einander durch verschiedene Programmierer entwickelt werden.

Wie diese Ziele durch die Aufnahme des Modul-Konzepts in Programmiersprachen umgesetzt werden, zeigt der folgende Abschnitt. Der Modulbegriff hat auf den ersten Blick wenig mit dem Begriff des Datentyps zu tun, aber in einer besonderen Ausprägung können Moduln so konstruiert werden, dass sie einen abstrakten Typ exportieren. Auch das wird am Ende des nächsten Abschnitts deutlich werden.

[50] vgl. Abschnitt 3.2

[51] engl. information hiding

[52] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972

[53] Dieser Vorgang wird auch als *Modularisierung* bezeichnet.

Zuvor soll noch anhand eines kleinen Beispiels gezeigt werden, warum es besonderer Sprachkonstrukte bedarf, um die o. g. Ziele und Vorteile der Datenabstraktion zu erreichen.

Betrachten wir dazu eine Abstraktion für Punkte in der Ebene, wie sie z. B. in Scheme realisiert werden könnte:

- `(make-point x y)` erzeuge ein Punkt mit den ganzzahligen Koordinaten x und y.
- `(point-x p)` liefere die x-Koordinate des Punktes `p`.
- `(point-y p)` liefere die y-Koordinate des Punktes `p`.

Eine Implementierung dieser drei Prozeduren muss die folgenden Axiome erfüllen:

$$\bigwedge_{n,m\in\mathbb{Z}} (\texttt{point-x}\ (\texttt{make-point}\ n\ m)) = n \tag{1}$$

$$\bigwedge_{n,m\in\mathbb{Z}} (\texttt{point-y}\ (\texttt{make-point}\ n\ m)) = m \tag{2}$$

Damit sind alle Informationen verfügbar, um diese drei Prozeduren verwenden zu können. Eine Kenntnis ihrer Implementierung ist

nicht erforderlich.

Eine einfach Implementierung könnte wie folgt aussehen

```
(define make-point
  (lambda (x y) (cons x y)))
(define point-x
     (lambda (point) (car point)))
(define point-y
     (lambda (point) (cdr point)))
```

Die Prozeduren könnten z. B. so verwendet werden:

```
(point-y (make-point 3 4))
```

liefert 4.

Allerdings kann die Abstraktion leicht durchbrochen werden, indem man von der Kenntnis der Implementierung Gebrauch macht und z. B. statt der Prozedur `point-y` die Scheme-Standardprozedur `cdr` verwendet.

Der Ausdruck

```
(cdr (make-point 3 4))
```

liefert auch 4.

Das ist solange unproblematisch, wie die Implementierung nicht verändert wird. Speicherte man z. B. in den `cons`-Paaren gar nicht die kartesischen Koordinaten sondern Polarkoordinaten, lieferte die Anwendung der Prozedur `cdr` nicht mehr die y-Koordinate, wohingegen die Prozedure `point-y`[54] weiterhin die korrekte y-Koordinate berechnete.

[54] nach einer entsprechenden Anpassung

Die Kenntnis der Implementierung einer Abstraktion zu nutzen, kann also leicht zu Fehler führen. In Scheme kann die oben eingeführte Abstraktion für Punkt auch so umgesetzt werden:

```
(define-struct point (x y))
```

Damit stehen die drei Prozeduren `make-point`, `point-x` und `point-y` automatisch zur Verfügung und erfüllen auch die Axiome. Der Versuch, z. B. die Prozedur `car` auf eine mit `make-point` erzeugte Struktur anzuwenden, wird jetzt mit einer Fehlermeldung quittiert:

```
> (car (make-point 3 4))
Error: car: expects argument of type <pair>; given #<point>
```

Eine Abstraktionsverletzung ist bei Verwendung von Scheme-Structures[55] so nicht mehr möglich.

[55] zu erzeugen mit `define-struct`

4.2 Modularten

Verschiedene Programmiersprachen[56], die in den siebziger Jahren entwickelt wurden, haben erstmals ein Modulkonzept – wie im vorangegangenen Abschnitt erläutert – systematisch eingeführt.

[56] darunter *Clu*, *Ada* und *MODULA-2*

Module können in drei Arten unterschieden werden:

1. Funktionsmoduln

   Module ohne Gedächtnis

   Das Modul enthält eine Sammlung von Funktionen, deren Ergebnisse (Funktionswerte) nur von den Eingabeparametern (Argumenten) abhängen. Funktionsmodule enthalten keine Daten, sondern nur Funktionen und Prozeduren. Sie werden im weiteren Verlauf nicht näher betrachtet.

Module mit Gedächtnis

2. Datenmoduln oder abstrakte Datenstrukturen
   Das Modul enthält (lokale) Daten(-strukturen) und verbirgt die Einzelheiten der Datenrepräsentation dadurch, dass es *nur* die Zugriffsoperationen exportiert. Diese Art von Moduln wird als Einführung in die modulbasierte Datenabstraktion in Abschnitt 4.3 näher erläutert. Sie haben durchaus ihre eigene Existenzberechtigung. Im Kontext der abstraktionsorientierten Sicht auf Datentypen können sie aber auch als „Vorstufe“ zu den abstrakten Datentypen angesehen werden.
3. Abstrakte Datentypen
   Das Modul exportiert einen Datentyp und die zugehörigen Operationen auf Objekten dieses Typs. Details der Struktur des Datentyps und der Zugriffsoperationen werden verborgen. Der wesentliche Unterschied zu den Datenmoduln besteht darin, dass sie zusätzlich einen Datentyp exportieren, so dass sich mehrere Exemplare einer Datenstruktur erzeugen lassen. Modulbasierte abstrakte Datentypen werden in Abschnitt 4.4 erläutert.

STELLVERTRETEND sollen die modulbasierte Datenabstraktion bzw. die modulbasierten abstrakten Datentypen am Beispiel von MODULA-2[57] erläutert werden.

[57] Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, 4. edition, 1989

4.3 Modulbasierte Datenabstraktion

Das MODULA-2 zugrunde liegende Modulkonzept kann wie folgt allgemein charaterisiert werden:

- Ein Modul ist ein klar umrissener, in sich abgeschlossener Programmteil, der eine funktional spezifizierte Teilaufgabe realisiert und dabei die Details der Realisierung verbirgt.
- Die Schnittstelle bildet Verbindung des Moduls mit der Umgebung, über die das Modul seine Dienste anbietet.
- Entwurfsentscheidungen werden innerhalb eines Moduls realisiert und vor dem Rest des Systems verborgen. Die Schnittstelle ist unabhängig von der Realisierung. Der Zugriff auf die im Modul gekapselten Datenstrukturen erfolgt über Zugriffsfunktionen.

information hiding, Geheimnisprinzip

prozedurale Schnittstellen

Module in MODULA-2 enthalten Vereinbarungen von Prozeduren, Konstanten, Variablen und Typen. Objekte aus anderen Modulen können importiert werden. Die Trennung zwischen der Schnittstelle eines Moduls und seiner Realisierung kann[58] durch Anlegen eines *definition module* und eines *implememtation module* erfolgen.

[58] Es gibt auch andere Möglichkeiten, Modulschnittstellen in MODULA-2 zu definieren. Die werden hier aber nicht betrachtet.

DAS FOLGENDE BEISPIEL zeigt die Schnittstelle eines Moduls, der eine Puffer nach dem FIFO-Prinzip realisieren soll.

```
definition module buffer;
    import elem_type, buffersize;
```

```
    procedure is_empty();
    procedure is_full(): boolean;
    procedure put(x: elem_type) : boolean;
    procedure get(var x: elem_type);
end buffer.
```

Bemerkungen:

1. `elem_type` sei ein Typ und `buffersize` eine Integer-Konstante. Beide werden importiert und müssen daher in einem umschließenden Gültigkeitsbereich deklariert sein.
2. Das Modul exportiert die Prozedurbezeichner `is_empty`, `is_full`, `put` und `get`. Nur diese Bezeichner des Moduls `buffer` können von anderen Moduln per Import verwendet werden.
3. Man beachte: Ein `definition module` spezifiziert exakt nur die Aufrufschnittstellen der Operationen (Syntax). Ihre Wirkung (Semantik) kann hier nicht formal spezifiziert werden.
4. Ein `definition module` kann getrennt übersetzt werden.

Eine mögliche Imlementierung von `buffer` könnte so aussehen.

```
implementation module buffer;
   var  in, out: [0..buffersize-1];
        n: [0..buffersize];
        buf: array[0..buffersize] of elem_type;
        full, empty: boolean;

    procedure put(x: elem_type);
    begin
        if n < buffersize
        then buf[in] := x;
            in := (in+1) mod buffersize;
            n := n+1;
            full := n = buffersize;
            empty := false;
        end;
    end put;

    procedure get(var x: elem_type);
    begin
        if n > 0
        then x := buf[out];
            out := (out+1) mod buffersize;
            n := n-1;
            empty := n = 0;
            full := false;
        end;
    end get;
```

```
    procedure is_empty(): boolean;
    begin
        return empty;
    end is_empty;

    procedure is_full(): boolean;
    begin
        return full;
    end is_full;

begin (* Initialisierungen *)
    n := 0; in := 0; out := 0;
    empty := true; full := false;
end buffer.
```

Die Verwendung des Moduls ist nur über die exportierten Prozeduren möglich. Für die Übersetzung eines `buffer` importierenden Moduls ist nur `das definition module` erforderlich. Die Implementierung ist vollständig gekapselt und kann auch, solange die Schnittstelle unverändert bleibt, beliebig geändert werden.

Damit ist das Ziel der Abstraktion erreicht, Verletzungen der Abstraktion sind nicht möglich, da sie vom Compiler unterbunden werden.

Der Nachteil von abstrakten Datenstrukturen dieser Art besteht darin, dass auf diese Weise nur ein Exemplar der Datenstruktur in einem Programm eingerichtet werden kann. Wollte man mehrere Exemplare anlegen können, ist eine Erweiterung des Konzepts erforderlich: Aus der abstrakten Datenstruktur muss ein abstrakter Datentyp werden, von dem dann beliebig viele Exemplar angelegt werden könnten. Das Konzept modulbasierter abstrakter Datentypen wird im folgenden Abschnitt beschrieben.

4.4 Abstrakte Datentypen

Durch Import des Moduls `buffer`[59] lässt sich in einem Programm **ein** Exemplar eines solchen Puffers erzeugen. Wenn hingegen das Modul `buffer` einen Datentyp – nennen wir ihn `FIFObuf` – exportierte, wäre in importierenden Moduln folgende Deklaration möglich:

[59] vgl. Abschnitt 4.3

Die Möglichkeit, das Modul unter einem anderen Namen zu kopieren, wird nicht betrachtet.

```
var puf1, puf2: FIFObuf;
```

Alle Zugriffsprozeduren müssen nun einen weiteren Parameter haben, über den die `FIFObuf`-Exemplare angesprochen werden können; z. B. :

```
put(puf1, e1); get(puf2, e2);
```

MODULA-2 bietet die Möglichkeit des Exports *privater Typen*[60]. Von privaten Typen ist in importierenden Moduln nur der Typname bekannt.

[60] engl.; *opaque type*

Die Schnittstelle von `buffer` könnte demnach so aussehen:

```
definition module buffer;
    import elem_type, buffersize;
    type FIFObuf;
    procedure is_empty(puf: FIFObuf; );
    procedure is_full(puf: FIFObuf; ): boolean;
    procedure put(puf: FIFObuf; x: elem_type) : boolean;
    procedure get(puf: FIFObuf; var x: elem_type);
    procedure init(var puf: fifobuf);
end buffer.
```

hier wird `FIFObuf` als privater Typ exportiert.

Bei Übersetzung dieses *definition module* und von importierenden Moduln sind die Typstruktur und sein Speicherplatzbedarf unbekannt. Um dieses Problem zu umgehen, sind in MODULA-2 private Typen immer Zeigertypen. Das wiederum bedeutet, dass ein Exemplar eines privaten Typs zur Laufzeit erst erzeugt werden muss. Deshalb sieht die Schnittstelle jetzt eine zusätzlich Prozedur `init` vor. Die Implementierung von `buffer` könnte ausschnittsweise wie folgt aussehen:

```
implementation module buffer;
   type FIFOBuf = pointer to buffer;
        index = [0 .. buffersize - 1];
        buffer = RECORD
                    container : array [index] of elem_type;
                    in, out: index;
                    full, empty := boolean;
                 END;

    procedure put(puf: FIFOBuf; x: elem_type);
    begin
    if not puf@.full
    then puf@.container[in] := x;
         puf@.in := (puf@.in+1) mod buffersize;
         puf@full := puf@.in = puf@.out;
         puf@.empty := false;
    else error
    end
    end put;

    procedure get(puf: FIFOBuf; var x: elem_type);
    begin
    ...
    end get;

    procedure is_empty(puf: FIFOBuf): boolean;
    begin
        return put@.empty;
    end is_empty;

    procedure is_full(puf: FIFOBuf): boolean;
```

```
    begin
        return puf@.full;
    end is_full;

    procedure init (var put : FIFOBuf);
       (* erzeugt leeren Puffer *)
    begin
    new (puf);
    puf@.in := 0;
    puf@.out := 0;
    puf@.full := FALSE;
    puf@.empty := TRUE
    end init;

begin (* Initialisierungen *)
end buffer.
```

Die Benutzung von `buffer` könnte nun andeutungsweise so erfolgen:

```
module m;
    :
    type elem_type = record data: integer end;
    from buffer import fifobuf, put, get, init, is_full, is_empty;
    var puf1, puf2: fifobuf;
        e: elem_type;
    :
begin
    :
    init(puf1);
    e.data :=0;
    repeat
        put(puf1, e);
        inc(e.data);
    until is_full(puf1);
    :
end m.
```

MODULE, DIE PRIVATE TYPEN EXPORTIEREN, sind also von der dritten der in Abschnitt 4.3 genannten Modularten: abstrakte Datentypen. Die Abstraktion wird dabei im wesentlichen durch folgende Konzepte gewährleistet:

- Von privaten Typen ist nach außen nur der Typname bekannt.
- Zuweisungen an Variablen von einem privaten Typ sind nicht möglich.
- Nur die Prüfung auf Gleichheit und die Übergabe als Parameter. ist erlaubt.

DER BEGRIFF *abstrakter Datentyp* kann zusammenfassend so charakterisiert werden:

- Er wird durch den Programmierer neu definiert.
- Verwendungs- und Implementierungsteil sind strikt getrennt, d. h. die Repräsentation der Datenstruktur ist nach außen unsichtbar.
- Der Verwendungsteil (*Definitionsteil*, *Schnittstelle*, *interface*) beschreibt die Eigenschaften eines Datentyps aus Anwendersicht durch Festlegung,
  - welche Operationen es gibt,
  - wie sie benutzt werden und
  - was sie bewirken.

Auch vordefinierte Datentypen wie INTEGER oder REAL sind abstrakt. Von ihnen sind nur der Typname und die Signaturen der erlaubten Operationen bekannt, sie können ohne Kenntnis ihrer Implementierung verwendet werden. Die maschineninterne Repräsentation der Werte ist (innerhalb gewisser Grenzen) unerheblich. Jedenfalls ist sie unsichtbar. Meistens wird aber die Verwendung des Begriffs *abstrakter Datentyp* auf vom Programmierer zu definierende Typen begrenzt.

Man beachte, dass die Wirkung der Operationen eines abstrakten Datentyps mit den hier vorgestellten Sprachmitteln durch ein `definition module` bestenfalls informell[61] spezifiziert werden können. Mit der Frage, wie auch die Semantik eines abstrakten Datentyps formal und implementierungsunabhängig spezifiziert werden kann, wird sich ein weiteres Arbeitspapier der Reihe „Typen, Objekte, Klassen“ beschäftigen,

[61] z. B. durch geeignete Wahl der Bezeichner oder Kommentare in der Schnittstelle

5 *Fazit*

Die Darstellung verschiedener Sichtweisen auf den Typbegriff soll lediglich dabei helfen, seine Komplexität durch Systematisierung der Betrachtung besser zu erfassen. In der praktischen Arbeit verschwimmen die Sichtweisen sowohl für Programmierer als auch für Sprachentwickler. Es ist offensichtlich, dass es gerade zwischen der konstruktiven und der abstraktionsorientierten Sicht Wechselbeziehungen gibt. Die Implementierung eines abstrakten Datentyps hat natürlich auch etwas „Konstruktives“.

Der Schritt von den abstrakten Datentypen zu dem Begriffspaar *Objekt* und *Klasse* im Sinne der objektorientierten Programmierung ist nicht sehr groß. Ohne das an dieser Stelle ausführlich darlegen zu wollen, sei folgende „syntaktische“ Betrachtung angestellt: Die Operationen des abstrakten Datentyps `FIFObuf` aus Abschnitt 4.4 besitzen alle als ersten Parameter den zu verarbeitenden Puffer. Dieser Parameter ist also in gewisser Weise ausgezeichnet. Objektorientierte Programmiersprachen „tragen dem Rechnung“, indem sie ihn „nach vorne ziehen“. Die Behandlung der Erweiterung des Typbegriffs durch die objektorientierten Programmiersprachen ist aber Thema eines weiteren Arbeitspapiers.

Statt `put(puf1, e)` schreibt man in Smalltalk `puf1 put: e` oder in Java `puf1.put(e)`.

Literatur

[1] Johannes Brauer. Typen, Objekte, Klassen – Teil 1: Grundlagen. Arbeitspapier 2009-05, NORDAKADEMIE Hochschule der Wirtschaft, Juni 2009.

[2] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, 1985.

[3] Ole-Johan Dahl and Kristen Nygaard. Simula: an algol-based simulation language. *Commun. ACM*, 9(9):671–678, 1966.

[4] Chenglie Hu. Dataless objects considered harmful. *Commun. ACM*, 48(2):99–101, 2005.

[5] Chenglie Hu. Just say 'a class defines a data type'. *Commun. ACM*, 51(3):19–21, 2008.

[6] Hanspeter Mössenböck. *Objektorientierte Programmierung in Oberon-2*. Springer-Verlag GmbH, 2. edition, 1994.

[7] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[8] Stuart Reges. Back to basics in cs1 and cs2. *SIGCSE Bull.*, 38(1):293–297, 2006.

[9] Stuart Reges and Marty Stepp. *Building Java Programs:A Back to Basics Approach*. Addison-Wesley, Mai 2007.

[10] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers, 2. edition, 2006.

[11] Robert Sedgewick and Kevin Wayne. *Introduction to Programming in Java: An Interdisciplinary Approach*. Addison-Wesley, 2008.

[12] J. Stanley Warford. *Computing Fundamentals*. Vieweg Verlag, 2002.

[13] Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, 4. edition, 1989.