

Objektorientierte Programmierung

unter einer (anderen) konzeptuellen Sicht.

Peter Bachmann

Brandenburgische Technische Universität Cottbus,
Fakultät 1, Mathematik, Naturwissenschaften und Informatik

Postfach 10 13 44, D-03013 Cottbus

e-mail: pb@informatik.tu-cottbus.de

April 1998

Zusammenfassung

Über die kritische Analyse von Begriffsbildungen zur objektorientierten Programmierung, wie sie in einigen Publikationen der letzten Jahre erklärt sind, wird versucht, eine konzeptuelle Sicht zu entwickeln. Das soll bedeuten, daß die analysierten Begriffe Variable, Typ, Konstante, Objekt, Klasse, Vererbung und Polymorphismus möglichst unabhängig von einer speziellen Programmiersprache und damit auch unabhängig von syntaktischen Konstruktionen betrachtet werden. Als Konsequenz entsteht ein Vorschlag für eine Begriffswelt, in dem der statische Modul zur Beschreibung von Sichten sowie flexible Typkonzepte in den Mittelpunkt der Programmbeschreibung gestellt werden.

1 Einführung

Gegenwärtig wird der “Objektorientierung” eine noch euphorischere Erwartungshaltung entgegengebracht, als der strukturierten Programmierung in den sechziger und siebziger Jahren. Auch damals ging es darum, mittels eines gewissen Grundsatzes den Entwurf und die Implementierung von Software überschaubarer und sicherer zu machen. Die Überwindung der “Software-Krise” war das Hauptargument, was die Anwendung der strukturierten Programmierung motivierte. Eine Flut von Einzelarbeiten überschüttete die Fachpresse, Spezial-Konferenzen nahmen sich dieser Thematik an.

Wie bereits damals bezüglich der strukturierten Programmierung, so wird auch heute ständig die peinliche Frage gestellt, ob und mit welcher Konsequenz die objektorientierte Methode Anwendung findet. Das geht soweit, daß die objektorientierte Methode schlechthin als Qualitätsmerkmal der danach entwickelten Software angesehen wird. Das vermittelt den Eindruck, daß **nur** durch objektorientierte Programmierung gute Software entstehen kann.

Es entsteht nun (insbesondere für mich) die skeptische Frage, ob trotz (oder vielleicht auch wegen) dieser eingangs bemerkten Euphorie die induzierten Erwartungen annähernd erfüllt werden können. Um mir diese Frage zu beantworten habe ich versucht, das Wesen der objektorientierten Methode etwas genauer und kritischer zu erfassen. Ich habe mich dabei auf die programmtechnischen Aspekte konzentriert und weniger die Darlegungen zur objektorientierten Modellierung (etwa realer Systeme) allgemein berücksichtigt.

Wenn man in das Wesen einer Sache eindringen will, sollte man sich bemühen, die damit verbundenen Begriffe zu verstehen. Hier entstand für mich die Hauptschwierigkeit: in nur wenigen der von mir gelesenen Arbeiten fand ich eine auch nur annähernd präzise Begriffswelt. Viele der Arbeiten unterscheiden sich in ihren Auffassungen, mitunter wechseln die Auffassungen innerhalb eines Buches und führen mehr zur Verwirrung als zur Klärung.

Nun ist mir natürlich bewußt, daß man im Prozeß der Entwicklung von Vorstellungen, Methoden und Konzepten erst schrittweise zu exakten Begriffen gelangt. Ausgangspunkt ist oft ein grobes und

ungenau, auf Intuition basiertes Gefüge. So war es auch an vielen Stellen in der Mathematik. Ein entscheidender Durchbruch in der Nutzung mathematischer Erkenntnisse erfolgte aber immer dort, wo diese hinreichend präzise abgeklärt waren. Es ist innerhalb der Mathematik nicht nur eine Frage der Ästhetik, sich bei disziplinären Entwicklungen um die Aufdeckung von Analogien zu anderen Zweigen zu bemühen, Dualitäten zu erkennen, daraus Nutzen für den unmittelbaren Gegenstand zu erlangen.

Im mehr praktisch orientierten Teil der Informatik ist diese Ästhetik nur schwer zu erkennen. Betrachtet man nur die Entwicklung bei den Programmiersprachen, so muß man vermuten, daß es scheinbar eine Frage der Ehre ist, sich mit eigenen syntaktischen Konstruktionen und Bezeichnungen von anderen Sprachen zu unterscheiden. Was hier *Variable* heißt, wird dort *entity* genannt. Was in der einen Sprache mit der Zeichenkombination := gemeint ist, wird in der anderen Sprache mit = bezeichnet.

Mancher Leser wird diese Effekte als orthographische Nebensächlichkeiten abtun. Vielleicht ist es sogar vorteilhaft, solche Programmiersprachen bereitzustellen, in denen jeder Nutzer sich seine eigene Orthographie festlegen kann. Erstrebenswert erscheint mir allerdings das Bemühen um eine einheitliche Sicht auf gewisse, insbesondere bewährte, Grundkonzepte. Das sollte dann auch zu einer konzisen und präzisen Begriffswelt führen, die sicher nie endgültig sein kann, einen ständigen Ausbau erfahren wird, aber eben nur dann, wenn es neue Anforderungen tatsächlich erzwingen.

In diesem Artikel möchte ich versuchen, einige wenige mit der objektorientierten Methode verbundene Begriffe wie sie in verschiedenen Arbeiten beschrieben sind, kritisch zu beleuchten. Es geht mir dabei nicht um die Kritik an sich, sondern um den Versuch, eine Bewertung vorzunehmen und daraus - alternative - Konzepte abzuleiten.

Eine Bewertung sollte sich immer an Kriterien orientieren. Die von mir in den Vordergrund gestellten Kriterien beziehen sich auf die Beschreibung von Programmen. Dabei werden syntaktische Fragestellungen völlig ignoriert. Damit entfallen hier Überlegungen, die zu einer guten Lesbarkeit von Programmbeschreibungen führen. Natürlich müssen solche Überlegungen (aber an anderer Stelle) erfolgen. Dafür ist bei den Konzepten darauf zu achten, daß Programmbeschreibungen gut verständlich sind (was mit Lesbarkeit korreliert), daß sie den arbeitsteiligen Entwicklungsprozeß und die Wartungsarbeiten unterstützen. Vier Prinzipien scheinen mir dafür besonders wichtig zu sein:

- **Orthogonalität:** Es sollten wenige Konzepte mit klarer Zweckbestimmung existieren. Es ist zu vermeiden, daß verschiedene Konzepte zur Lösung gleicher Probleme dienen.
- **Reflektion:** Die Programmbeschreibung sollte viel vom Verhalten des beschriebenen Programms reflektieren, ohne daß die Dynamik des Verhaltens nachempfunden werden muß.
- **Lokalität:** Die Bedeutung von Teilen der Programmbeschreibung sollte aus einem engen lokalen Kontext eindeutig hervorgehen. Implizite Annahmen in der Programmbeschreibung sind auf die Fälle zu beschränken, die eindeutig und aus dem lokalen Kontext der Beschreibung einfach bestimmbar sind.
- **Modularität:** Die Programmbeschreibung sollte in verschiedene Teile gliederbar sein, die unterschiedliche Sichten auf das Gesamtprogramm ermöglichen. Es muß klar sein, wie die Verbindung zwischen diesen Teilen ist.

Angesichts der Leistungsfähigkeit moderner Computer und der bereitgestellten Entwicklungssysteme ist die Kürze einer Programmbeschreibung dagegen irrelevant. Auch der Schreibaufwand ist durch die Existenz von leistungsfähigen Editoren vernachlässigbar. Dagegen muß es möglich sein, umfassende Informationen über das Programm zur Verfügung zu stellen.

Bei der Begriffsanalyse und -bestimmung sollen nicht die einzelnen Programmiersprachen separat betrachtet werden, sondern der Versuch zielt auf ein übergreifendes Gerüst. Deshalb werden auch fast ausschließlich Arbeiten und Bücher zitiert, die sich dem Gebiet der Programmiersprachen bzw. der Programmierung im allgemeinen widmen ([1], [4], [8], [11], [15], [20], [23]).

Nach Analyse der Situation sind mir die folgenden Begriffe aufgefallen, die immer wieder im Zusammenhang mit der Objektorientierung genannt werden: *Objekt*, *Klasse*, *Vererbung*, *Polymorphismus*. Da aber zwischen *Objekt* und *Variable* sowie zwischen *Klasse* und *Typ* ein enger Zusammenhang zu existieren scheint, habe ich mit den Betrachtungen zu *Variable* und *Typ* begonnen.

2 Die konzeptuelle Sicht

Damit ist gemeint, daß die objektorientierte Methode ausschließlich in ihrer Verwendung bei der Beschreibung von Software behandelt wird. Es wird nicht berücksichtigt und betrachtet, wie mit den behandelten Begriffen etwa der Briefzustelldienst mit Postamt, Briefkasten, Kunden und Postzusteller modelliert werden kann. Wenn dies nicht auf elektronischem Wege erfolgen soll (dann aber nach hinreichend anderen Prinzipien) ist dies auch nicht mit Software zu realisieren! Das erlaubt, bereits auf eine idealisierte Welt zu reflektieren, in der alle Objekte abstrakt sind.

„Konzeptuelle Sicht“ soll weiterhin heißen, daß sich die Begriffe nicht auf den Computer mit seinen konkreten Bestandteilen und Verhaltensweisen beziehen sollen. Damit wird angestrebt, von speziellen Rechnerarchitekturen, Betriebssystemvarianten und Implementierungen unabhängig zu bleiben. Als Konsequenz werden Begriffe wie Speicher oder Laufzeit nicht benutzt.

Da sich aber die angestrebte Begriffswelt auf Software bezieht, muß das Prinzipielle bei der Entwicklung, Implementierung und Nutzung der Software reflektiert werden. Es ist deshalb auch nicht verboten, wird im Gegenteil zur intuitiven Erfassung sogar empfohlen, sich unter gewissen abstrakten Begriffen konkrete Sachverhalte aus der Realität vorzustellen. Auch wird zur Motivation der Betrachtungsweise auf Implementierungsprinzipien Bezug genommen, nie sollen diese aber den Grundstein bilden.

In den theoretisch orientierten Untersuchungen ist der Abstraktionsgrad oft so groß, daß die dort genutzten Begriffe reale Sachverhalte zu idealisiert widerspiegeln. Der hohe Grad an Abstraktion ist allerdings für gewisse theoretische Betrachtungen unerlässlich. Deshalb kann man theoretische Konzepte nur bedingt zur Erklärung realer Vorgänge nutzen.

Zunächst halte ich es für notwendig, zwischen der **Programmbeschreibung** und dem **Programm** zu unterscheiden. Meist wird der syntaktisch korrekte Text einer Programmiersprache als Programm bezeichnet. Das erscheint mir deshalb als eine zu stark vereinfachte Sicht, weil der Text einer Sprache einer Interpretation bedarf, die von Randbedingungen wie zum Beispiel Systemkonfigurationen abhängen kann. Auch sind gewisse Anforderungen an Programmbeschreibungen verschieden von denen, die man an Programme stellt. So ist die Modularisierung der Programmbeschreibung und das Verbergen von Informationen ein wichtiges Mittel zur Wahrung der Übersicht und zur Gestaltung der arbeitsteiligen Entwicklung. Im Programm ist eine modulare Struktur nicht mehr zwingend, Verbergen von Informationen sogar gegenstandslos.

Eine Programmbeschreibung kann etwa ein Text sein, der den Regeln einer gewissen Programmiersprache genügt. Es sind natürlich auch andere Beschreibungsformen wie Graphiken (in der visuellen Programmierung zunehmend genutzt) denkbar. Bei SMALLTALK ist die Programmbeschreibung eine strukturierte Menge von Texten, die mit speziellen Hilfsmitteln durchmustert und verändert werden kann. Immer sollen zu Programmbeschreibungen eine Syntax, d.h. die charakteristische Funktion der Menge aller Programmbeschreibungen und eine Semantik, d.h. eine Funktion, die zu jeder Programmbeschreibung eine Bedeutung zuordnet, existieren. In diesem Sinne bildet die Menge aller Programmbeschreibungen eine (oder auch mehrere) Sprache(n).

Eine **Programmbeschreibung** ist ein Element einer Sprache, die Bedeutung einer Programmbeschreibung ist ein **Programm**.

Ein Programm dient zur Steuerung des Ablaufs im Computer. Bei der Programmanwendung wird dieser Ablauf angestoßen. Dazu enthält das Programm Elemente, die diesen Ablauf festlegen, im allgemeinen als Daten bezeichnet. Hier sollen diese Elemente anonym als *Werte* bezeichnet werden.

Ein **Programm** ist eine Menge von Werten.

Die Werte eines Programms sind in jeder Anwendung des Programms gleich und deshalb von der Anwendung unabhängig, sie sind *statisch*. Dagegen ist der Programmablauf *dynamisch*, je nach Anwendung verschieden und so können verschiedene Werte erzeugt und vernichtet werden.

Die Unterscheidung zwischen statischen und dynamischen Elementen ist wichtig, damit der Programmierer bei der Programmbeschreibung die Konsequenzen der eingeführten Konzepte klar erkennen kann.

3 Variable und Konstante

In der Programmierung ist die mit den Variablen verbundene Intention die Speicherung von Werten (Daten). Um sich auf eine Variable beziehen zu können, braucht man dafür eine Identifikation. Als dritter Aspekt tritt der Typ einer Variablen auf. Damit erhält man die drei zur Begriffsbildung oft benutzten Komponenten: *Identifikation, Typ, Wert*.

Dieser Vorstellung kommt die Auffassung vom Tripel (Referenz, Behälter, Wert) [12] sehr nahe. Allerdings ist eine solche Begriffsbestimmung unter folgendem Gesichtspunkt fraglich: Bei einer alleinigen Änderung der dritten Komponente des Tripels, also des Wertes, ändert sich das Tripel, es ist verschieden vom ursprünglichen, wenn man die in der Mathematik übliche Gleichheit zwischen n -Tupeln zugrunde legt. Als Konsequenz erhält man eine andere Variable, was aber offensichtlich nicht beabsichtigt ist.

Das Problem dieses Begriffes liegt nach meiner Auffassung darin, daß unzureichend differenziert wird, wo man Aussagen zu einer Variablen machen möchte. Identifiziert wird eine Variable in der Programmbeschreibung, mit Werten belegt wird sie bei der Anwendung eines Programms. Die Rolle des Typs wird im nächsten Abschnitt betrachtet.

Die Identifikation, das heißt die *Bezeichnung* einer Variablen in der Programmbeschreibung ist aber nicht mit der Variablen selbst zu verwechseln! Es ist ja möglich, daß die gleiche Bezeichnung verschiedene Variable identifiziert, abhängig zum Beispiel von der Stelle im Programm, an der die Bezeichnung auftritt. Die Frage der Identifikation soll weiter unten im Zusammenhang mit den Konstanten behandelt werden.

Ich schlage ein analoges Vorgehen wie in der mathematischen Logik vor, indem über Variable nichts weiter ausgesagt wird, außer daß diese Elemente einer gewissen gegebenen Menge sind und daß Variable mit Werten, das sind ebenfalls Elemente einer gewissen Menge, *belegt* werden können. Da Variable Elemente von Programmen sind, und nach der hier vertretenen Auffassung Programme Mengen von Werten sind, müssen Variable Werte sein!

Eine **Variable** ist ein Element aus einer gegebenen Menge \mathcal{V} (von Variablen). Es gilt $\mathcal{V} \subseteq U$, wobei U die Menge aller Werte sei.

Eine (partielle) Funktion $\sigma : \mathcal{V} \leftrightarrow U$, die den Variablen aus $dom\sigma$ einen Wert aus der Menge U zuordnet, wird **Variablenbelegung** genannt.

Mit $\varphi : A \leftrightarrow B$ wird eine *partielle* Funktion *aus* der Menge A in die Menge B bezeichnet. Es ist $dom\varphi := \{a \mid \exists b : (a, b) \in \varphi\}$ der Definitionsbereich von φ .

Der Wert einer Variablen v ergibt sich nun aus der Anwendung der Variablenbelegung auf die Variable, ist also $\sigma(v)$.

Ist eine Variable ein Element eines Programms, so ist sie statisch, sie ist unabhängig von der Programmabarbeitung. Natürlich gilt das nicht für ihre Belegung, die kann sich während der Programmabarbeitung verändern.

Daneben existieren *dynamische* Variable, die während der Programmanwendung „entstehen“ und „vergehen“. Die Referenz auf dynamische Variable ist ausschließlich über die Variablenbelegung möglich.

Der statische Charakter von *Konstanten* wird in allen Programmiersprachen angenommen. Eine Konstante „hat“ immer einen festen Wert, der sich während der Programmanwendung nicht ändert. Meist werden Konstanten so bezeichnet, daß der Wert aus der Bezeichnung leicht interpretierbar ist. Solche Konstruktionen werden auch Literale genannt. Um bei den mitunter langen Literalen Schreibaufwand zu sparen und um bei mehrfacher Verwendung der gleichen Konstanten im Programm den Wert an einer Stelle festzulegen, existieren Möglichkeiten, frei wählbare Bezeichner für Konstanten einzuführen. Da die Werte von Konstanten de facto irgendwo gespeichert werden müssen, ähneln sie irgendwie den Variablen ([12, 20]).

Aber auch hier möchte ich stärker differenzieren und zwischen der Konstanten und dem durch diese repräsentierten Wert unterscheiden. Konstanten sind Elemente der Programmbeschreibung, mehr noch: alle Elemente der Programmbeschreibung, deren Bedeutung Werte sind, sind Konstanten! Eine Konstante ist ein Element aus einer gewissen Menge (von Konstanten). Der durch eine Konstante repräsentierte Wert wird durch die *Interpretation* der Konstanten festgelegt. Diese Interpretation ist statisch, von der Programmanwendung unabhängig. Verschiedene Konstanten können durchaus gleich interpretiert werden, den gleichen Wert haben.

Eine **Konstante** ist ein Element aus einer gegebenen Menge \mathcal{K} (von Konstanten). Eine Funktion $\kappa : \mathcal{K} \leftrightarrow U$, die gewissen Konstanten einen Wert zuordnet, heißt **Konstanteninterpretation**.

Eine statische Variable v ist ein Wert und Element eines Programms. Sie muß deshalb in der Programmbeschreibung mittels einer Konstanten k beschrieben werden. Die Interpretation κ liefert dann die Variable, d.h. $\kappa(k) = v$. Als Konsequenz der hier aufgebauten Begriffswelt erhält man: statische Variable sind Werte von Konstanten.

4 Typen

Die Benutzung von Typen in den Programmiersprachen ist durch folgende Gesichtspunkte motiviert:

1. Im Rechner müssen alle Daten als Bitfolgen abgespeichert werden, denen man ihre Bedeutung nicht ansieht. Deshalb ist eine gesonderte Interpretationsvorschrift für diese Bitfolgen erforderlich. Diese Interpretationsvorschrift wird aus dem Typ abgeleitet, der einer Variablen (d.h. einem Speicherplatz) zugeordnet wurde.
2. Bei der Implementierung muß jeder Variablen Speicherplatz zugeordnet werden. Der Umfang des benötigten Speicherplatzes hängt von den Werten ab, die einer Variablen durch die Variablenbelegungen zugeordnet werden können. Der Typ einer Variablen macht zur Menge aller möglichen Werte Aussagen, die zur Speicherplanung genutzt werden können.
3. Die Anwendung von Funktionen (bzw. Operatoren und Prozeduren) ist meist auf Werte gewisser Mengen beschränkt. Durch die Typzuordnung eröffnen sich Möglichkeiten zur Verifikation, daß Funktionsanwendungen, unabhängig von der aktuellen Variablenbelegung, immer auf Argumente eines zugelassenen Typs erfolgen, was eine größere Sicherheit für die Korrektheit des beschriebenen Programmes garantiert.

Für meine Betrachtungen sind die ersten beiden Argumente nicht signifikant. Ich will über die Realisierung von Werten mittels Bitfolgen und über Speicherplatz nicht nachdenken!

Es bleibt der dritte Gesichtspunkt: die Sicherheit, daß die Anwendung von Funktionen und gewisser Konstruktionen wie Ergibtanweisungen oder Qualifizierungen beim Zugriff auf Werte korrekt durchgeführt werden. Dies kann während der Programmabarbeitung oder bereits davor in der Programmbeschreibung abgesichert werden.

Im ersten Fall (wie zum Beispiel bei SMALLTALK) ist es erforderlich, daß Typinformationen im Programm enthalten sind (bei SMALLTALK die Information darüber, zu welcher Klasse eine Instanz gehört) und dort auf diese Bezug genommen werden kann. Also müssen Typen Werte sein, die als Konstante in der Programmbeschreibung festgelegt werden! Dann ist es auch konsequent, wenn man erlaubt, daß Variable Typen als Belegung besitzen, daß *Typvariable* existieren.

Will man sich darauf konzentrieren, die Typkorrektheit auf der Ebene der Programmbeschreibung abzusichern, so sind Typen spezielle Konstruktionen in der Programmbeschreibung, die nicht ins Programm eingehen. Typen werden dort gewissen Bestandteilen der Programmbeschreibung (Bezeichner für Variable, Konstanten, Routinen) zugeordnet, um deren Verwendung in einem gewissen Sinne einzuschränken. So geht man in der Typtheorie vor (Siehe zum Beispiel [6, 22]), in der der Aufbau von Inferenzsystemen zur Typableitung und deren Einfluß auf die Ausdruckskraft der Berechnungsmodelle untersucht wird. Damit wird auch das Anliegen des Gesichtspunktes 3 berücksichtigt. Auf der Ebene der Programmbeschreibung soll genauer von *Typ-Ausdrücken* anstatt von Typen gesprochen werden. Mit T_{exp} soll die Menge aller Typ-Ausdrücke bezeichnet werden.

Auf der Ebene eines Programmes sind, wenn vorhanden, wie bei [23], Typen als Werte spezielle Mengen von Werten. Wie auch in der Mathematik kann man eine Menge mit einer Eigenschaft (Prädikat) identifizieren, nämlich daß ein Element zu dieser Menge gehört. Zu jedem Element t und jedem Typ T gilt entweder t ist vom Typ T ($t \in T$) oder t ist nicht vom Typ T ($t \notin T$). Zusätzlich wird davon ausgegangen, daß diese Entscheidung konstruktiv ist (was bei Mengen i.a. nicht notwendig ist). Ein Typ T als Menge von Werten ist die Bedeutung eines Typ-Ausdruckes τ , kurz: $\llbracket \tau \rrbracket = T$. Die Auffassung, daß Typen spezielle Mengen sind, ist auch eine Grundlage der Theorie der abstrakten Datentypen, wo man Typen sogar als Algebra betrachtet. Das heißt, die über den Mengen ausführbaren Operationen gehören immer mit zum Typ. Damit eröffnet man zugleich die Möglichkeit, Typen mittels algebraischer Induktion zu definieren.

In funktionalen Sprachen wird dieses Prinzip direkt übernommen und konsequent eingesetzt. In imperativen objektorientierten Sprachen wird ein Kniefall an dieses Prinzip gemacht, indem in einer Klasse die Werte gemeinsam mit den darüber ausführbaren Operationen gekapselt werden. Das hat aber nichts

mit der induktiven Definition von Wertemengen zu tun und sollte deshalb auch nicht mit dem Begriff der abstrakten Datentypen vermischt werden. Es werden nämlich immer konkrete Datentypen beschrieben, die natürlich zur Implementierung abstrakter Typen einsetzbar sind. Die über den Typen ausführbaren Operationen sind oft implizit gegeben: Festpunktoperationen bei Festpunktzahlen, Gleitpunktoperationen bei Gleitpunktzahlen, Verschiebeoperationen bei Bitfolgen, Anwendung der Wertebelegung auf Variable, Änderung der Wertebelegung von Variablen, Anwendung von Routinen auf Werte, usw.. Dagegen ist auch nichts einzuwenden. Typen als Mengen sind leicht verständlich, wenn man von einem vorhandenen Grundverständnis zu einigen Basistypen und den zugehörigen Operationen ausgeht.

Ein **Typ** ist eine Menge. Mit \mathcal{T} soll die **Menge aller Typen** bezeichnet werden. Die Vereinigung der Typmenge heißt **Universum** U ($U = \bigcup \mathcal{T}$), das ist die Menge aller Werte. Es existiert eine Funktion, $typ : \mathcal{V} \leftrightarrow \mathcal{T}$, **Typzuordnung** genannt, die jeder Variablen einen Typ zuordnet. Für jede Variablenbelegung $\sigma : \mathcal{V} \leftrightarrow U$ wird gefordert, daß $\forall v \in dom \sigma : \sigma(v) \in typ(v)$.

Wegen der Existenz der Typzuordnung typ kann man die Variablenmenge \mathcal{V} in ein System disjunkter Untermengen $\{\mathcal{V}_t \mid t \in \mathcal{T} \wedge \forall v \in \mathcal{V}_t : typ(v) = t\}$ zerlegen. Wir nehmen an, daß jede der Mengen \mathcal{V}_t (potentiell) unendlich ist, also beliebig viele Variable zur Verfügung stehen, und daß $\mathcal{V} = \bigcup \{\mathcal{V}_t \mid t \in \mathcal{T}\}$ gilt.

Es hängt nun von der betrachteten Programmiersprache ab, welche Typen insgesamt verfügbar (genauer: beschreibbar) sind. Schrittweise werden aber einige grundsätzliche Anforderungen an Typen formuliert und beispielhaft gezeigt, wie ein schlüssiges Typkonzept entwickelt werden kann. Dazu werden syntaktische Festlegungen zu Typ-Ausdrücken gemacht, die aber nur exemplarisch für die Erläuterungen in diesem Artikel zu verstehen sind.

Zunächst soll gefordert werden, daß eine gewisse Menge von Basistypen existiert, zum Beispiel Mengen von Festpunktzahlen und Gleitpunktzahlen unterschiedlicher Genauigkeit, Zeichenmengen usw.. Zu jedem Basistyp T gibt es auf der Ebene der Programmbeschreibung einen Typ-Ausdruck τ mit $\llbracket \tau \rrbracket = T$. Die Basistypen sollen paarweise disjunkt sein, d.h., für je zwei Basistypen T, T' gilt: $T \cap T' = \emptyset$.

Variablen wurden als Werte betrachtet. Deshalb ist es notwendig, Mengen von Variablen als Typen zuzulassen.

Ein **variable**-Typ beschrieben durch einen Typ-Ausdruck der Form **var** τ , wobei τ ein beliebiger Typ-Ausdruck ist, hat die Bedeutung:

$$\llbracket \text{var } \tau \rrbracket := \mathcal{V}_{\llbracket \tau \rrbracket}$$

Da Typen als Mengen einführt sind, existiert eine Untertypen-Beziehung, die sich einfach aus der Mengeninklusion ergibt. Das ist eine (partielle) Ordnung über den Typen. Klar ist damit, daß ein Wert nun Element aus mehreren Typen sein kann. Nicht jede Untermenge muß aber ein (zugelassener) Typ sein. Die Mengeninklusion über den Typen induziert eine Quasiordnung $\lesssim \subseteq T_exp \times T_exp$:

$$\tau \lesssim \tau' :\Leftrightarrow \llbracket \tau \rrbracket \subseteq \llbracket \tau' \rrbracket.$$

In der Typtheorie wird die Relation \lesssim meist induktiv über den Aufbau der Typ-Ausdrücke definiert. Ziel ist es dabei, die Äquivalenz $\tau \lesssim \tau' \Leftrightarrow \llbracket \tau \rrbracket \subseteq \llbracket \tau' \rrbracket$ zu sichern. Dies ist nicht immer möglich, mindestens sollte aber die Implikation $\tau \lesssim \tau' \Rightarrow \llbracket \tau \rrbracket \subseteq \llbracket \tau' \rrbracket$ gelten.

\lesssim beeinflußt auch Inferenzregeln zur Ableitung der Typen von Ausdrücken, mittels derer die Überprüfung der Typsicherheit einer Programmbeschreibung ermöglicht wird. Ein solches Vorgehen erfordert die vollständige Angabe der Programmbeschreibungssprache. Da diese hier nur in den Grundkonzepten vorgestellt werden soll, wurde die Relation \lesssim über die Bedeutung der Typ-Ausdrücke eingeführt. Es handelt sich im allgemeinen tatsächlich um eine Quasiordnung, da für verschiedene τ und τ' durchaus $\llbracket \tau \rrbracket = \llbracket \tau' \rrbracket$ gelten kann! Insbesondere geht aus den Festlegungen zu **variable**-Typen hervor, daß aus $\text{var } \tau \lesssim \text{var } \tau'$ immer folgt, daß $\llbracket \tau \rrbracket = \llbracket \tau' \rrbracket$ und damit auch $\llbracket \text{var } \tau' \rrbracket = \llbracket \text{var } \tau \rrbracket$.

Eine wesentliche Konsequenz hat die Entscheidung, ob Typen als Werte zugelassen werden. Falls ja, muß Sorgfalt geübt werden, um Anomalien auszuschließen. Man könnte etwa (wie in gewissen Mengenlehren) gestuft vorgehen. Zum Beispiel könnten wir die Menge \mathcal{TT} aller Typen erster Stufe einführen und fordern, daß $\mathcal{T} = \mathcal{TT} \cup \{\mathcal{TT}\}$. Falls eine Variable v mit einem Typ als Wert belegt ist, also den Typ $typ(v) = \mathcal{TT}$ hat, so ist v eine Typvariable. Damit wird *Polymorphismus* (Siehe 8) in natürlicher Weise möglich. Allerdings wird die im Gesichtspunkt 3 beschriebene Verifikation erschwert, wenn nicht gar unmöglich gemacht, wenn der Typ einer Variablen v' über die Belegung von v fixiert ist, also

$typ(v') = \sigma(v)$ gilt.

Es soll hier das einfachere und wahrscheinlich für die Praxis überschaubarere Konzept verfolgt werden, wonach Typen keine Werte sind. Das erfordert, nur auf der Grundlage der Programmbeschreibung Aussagen zum Programmverhalten, nämlich den erlaubten Werten von Variablen, der korrekten Anwendung von Funktionen und den Typen deren Resultate zu machen. Dazu müssen, analog zu den funktionalen Sprachen, Inferenzregeln entwickelt werden. Allerdings ist wegen den möglichen Seiteneffekten bei imperativen Sprachen die Entwicklung vollständiger Inferenzregeln komplizierter.

5 Objekt und Klasse

Mit diesen Begriffen stoßen wir erstmalig in die Gefilden der „objektorientierten Programmierung“ vor, obwohl Objekte, anstatt von Variablen oder Konstanten, oft auch in der „konventionellen“ Programmierung betrachtet werden (Siehe [4]). Ich schließe mich dieser Auffassung an und identifiziere die Begriffe **Objekt** und Variable. Das heißt, daß ich auf den Begriff Objekt verzichten kann.

Der in [11] angedeutete „innere Zustand“ ist nun einfach der durch die Variablenbelegung beschriebene Wert der Variablen.

Im Paradigma der objektorientierten Programmierung jedoch verfolgt man die Vorstellung, daß ein Objekt solche Eigenschaften hat, daß es als Modell für Dinge der Realität dienen kann. Bei vielen Erläuterungen wird auch deutlich, daß damit gleichzeitig das Prinzip der Datenkapselung, also des Verbergens von gewissen Informationen, berücksichtigt werden soll.

Viel allgemeiner, d.h. nicht nur auf Programmiersprachen bezogen sondern als Komponenten von beliebigen Systemen, werden Objekte in [12] angesehen. Es ist aber fraglich, ob durch solche unscharfe Formulierungen die Begriffsbildung vorangebracht werden kann. Was heißt zum Beispiel, daß eine Eigenschaft durch Werte „erfaßbar“ ist? Inwiefern können Objekte „Tätigkeiten ausführen“? Die hier angebotene Begriffsbildung erscheint zwar viel ärmer, aber dafür präziser. Sie ist kompatibel zu der in [2] vertretenen Auffassung.

Dort wird, wie auch ansonsten einheitlich vertreten, die **Klasse** als ein spezieller Typ betrachtet, deren Elemente Variable sind. Zusätzlich wird gefordert, daß alle zu einer Klasse gehörenden Variablen nach einem einheitlichen „Baumuster“ ([12]) aufgebaut sind. Dieser Forderung kann man auch dadurch gerecht werden, daß eine Klasse als eine Menge von Variablen mit einem gemeinsamen Typ betrachtet wird, also als einen Variablentyp. Auch dadurch würde der Begriff der Klasse überflüssig. Allerdings sind bei Klassen zusätzliche Eigenschaften vorhanden, insbesondere die *Vererbung*. Für diese, durchaus vorteilhafte Technik wird ein analoges Konzept vorgeschlagen, was dann die Einführung eines *Klassentyps* sinnvoll macht (Siehe 7).

Nun kann man unbedenklich solche Beschreibungen wie: *zu einer Klasse gehören* oder *Instanz einer Klasse sein* mit *Element einer Klasse sein* assoziieren.

Als ein weiteres Charakteristikum von Klassen wird die Möglichkeit einer einheitlichen internen Strukturierung der „inneren Zustände“ ihrer Objekte, wie sie von **records** her bekannt ist, und die Existenz von Operationen, mittels derer die Werte von Objekten manipuliert werden können, im allgemeinen **Methoden** genannt, angeführt.

Bezüglich der Strukturierung der Werte von Objekten ist die Situation einfach zu erklären: jeder Wert ist Element eines **record**-Typs, besteht damit aus einer festen Zahl von Komponenten eines gewissen Typs. Jede Komponente ist mittels **Indizes** selektierbar. Die Indizes sind Elemente einer, im allgemeinen endlichen, **Indexmenge**. Von der Indexmenge wird nur gefordert, daß ihre endlichen Teilmengen Definitionsbereiche von Werten eines **record**-Typs sein können. Sie werden damit Bestandteile von Werten, bilden selbst aber keine Werte, insbesondere keine Variable. Würde man Indizes als Werte zulassen, so ergeben sich Schwierigkeiten bei der Typinferenz und die Typkorrektheit wäre auf der Ebene der Programmbeschreibung nicht abzusichern.

Die Beschreibung eines **record**-Typs beinhaltet also unter anderem eine Indexmenge I und zu jedem Index $i \in I$ einen Typ T_i , d.h. eine Menge. In der Mathematik ist diese Konstruktion als indizierte Mengenfamilie bekannt. Ein Wert w eines Objektes ist damit eine Funktion $w : I \rightarrow U$, die jedem Index i einen Wert $w(i) \in T_i$ zuordnet, also ein Element aus dem kartesischen Produkt über der indizierten Mengenfamilie:

$$\times \{(i, T_i) \mid i \in I\} = \{w \mid w : I \rightarrow U \wedge \forall i \in I : w(i) \in T_i\}.$$

Mit U wurde das Universum aller Werte bezeichnet.

Es ist davon auszugehen, daß entsprechende **record**-Typen existieren.

Ein **record**-Typ beschrieben durch einen Typ-Ausdruck der Form $\{i_1 : \tau_1, \dots, i_n : \tau_n\}$, wobei die i_1, \dots, i_n Indizes und die τ_1, \dots, τ_n Typ-Ausdrücke sind, hat die Bedeutung:

$$\llbracket \{i_1 : \tau_1, \dots, i_n : \tau_n\} \rrbracket = \times \{(i_j, [\tau_j]) \mid j \in \{1, \dots, n\}\}$$

Bedeutungen von **record**-Typen sind also Mengen von Funktionen. Jede dieser Funktion ist ein Wert, der zu Indizes gewisse Werte zuordnen. Somit gehen Indizes in Werte ein, sind aber selbst keine Werte.

Daraus folgt auch die wichtige Einschränkung, daß Indizes keine Variablen sein dürfen. Wäre dies nicht so, würde, da alle Objekte vom gleichen Typ die gleichen Indizes „haben“, eine Variablenbelegung über den Indizes den Wert aller Objekte bestimmen. Als Konsequenz hätten alle Objekte aus einer Klasse immer den gleichen Wert. In objektorientierten Programmiersprachen wie SMALLTALK, C++, Java ... setzt man sich über diese Problematik galant hinweg. Durch den Speicherklassen-Modifizierer **static** (C++,Java) wird dieser Effekt veranlaßt, es werden implizit globale Variable deklariert, aber als Variablen der Instanzen verschleiert. Das ist eine inkonsistente Begriffsbildung.

Natürlich soll erlaubt sein, daß der zum Index i zugeordnete Typ T_i eine Untermenge der Variablenmenge ist. Falls w der Wert eines Objekts o ist, so kann damit für einen gewissen Index i durchaus $w(i) = o$ gelten. Das entspricht der Referenz auf das eigene Objekt.

Für zwei **record**-Typen $\tau = \{i_1 : \tau_1, \dots, i_n : \tau_n\}$ und $\tau' = \{i'_1 : \tau'_1, \dots, i'_{n'} : \tau'_{n'}\}$ gilt $\tau' \lesssim \tau$ genau dann, wenn $n = n'$ und eine Bijektion $\pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ existiert mit $i_j = i'_{\pi(j)}$ und $\tau'_{\pi(j)} \lesssim \tau_j$.

In Programmiersprachen wie EIFFEL ([16]) und Veröffentlichungen wie [9] wird davon ausgegangen, daß zusätzliche Forderungen an die Instanzen einer Klasse gestellt werden können: sie haben gewisse *Invarianten*, das sind Eigenschaften, zu erfüllen. Invarianten können insbesondere Beziehungen zwischen den Werten verschiedener Indizes festlegen. Nach dem hier vertretenen Typkonzept bedeutet dies, daß nicht alle Werte aus dem kartesischem Produkt als Werte der Objekte zugelassen sind. Man geht zu Untermengen über.

6 Methoden und Routinen

Bezüglich der Methoden ist die Situation schwieriger. In den Zitaten aus [2, 23] kommt zum Ausdruck, daß zur Beschreibung einer Klasse die Operationen gehören, die auf die Objekte der Klasse angewendet werden können.

Im Konzept der abstrakten Datentypen besteht der Hauptzweck der einem Typ zugeordneten Operationen darin, daß darüber implizit die Menge aller Werte des Typs definiert wird. Diese Operationen wirken als *Konstruktor* von Werten, der mathematische Hintergrund ist eine frei erzeugte Algebra. Da aber diese Erzeugung meist modulo gewisser Axiome erfolgt, sind noch andere Operationen zur Formulierung dieser Axiome erforderlich. Oft dienen diese Operationen zur Selektion der Werte von Grundtypen, die Argumente von Konstruktoren sind und werden deshalb *Selektoren* genannt. Da bei abstrakten Datentypen keine Variablen in unserem Sinne verwendet werden, ist man bei Nutzung eines Typs immer auf die Konstruktoren und Selektoren angewiesen.

Dieses Konzept wird in der objektorientierten Programmierung dadurch simuliert, daß mittels vom Nutzer formulierter Methoden der Zugriff auf Komponenten eines Objekts ermöglicht wird, deren explizite Nutzung ansonsten verboten ist. Diese Komponenten werden als *privat* bezeichnet. Allerdings ist dies oft mit einer weitreichenden Konsequenz verbunden: jegliche Operationen sind nur als Methoden von Klassen beschreibbar! Damit wird die Ausdruckskraft eines Programmes ausschließlich durch die bestehenden Klassen und die darin enthaltenen Methoden bestimmt.

Zudem wird konzeptuell die Vorstellung stimuliert, daß die Objekte einer Klasse deren Methoden „ausführen“ können, falls an die Objekte eine entsprechende Aufforderung in Form einer Nachricht gesendet wird. Das Objekt selbst wird als aktives Element angesehen, das über Methoden „verfügt“. Die Methoden werden als Teile von Objekten zu dynamischen Elementen.

Das entspricht aber keinesfalls den Implementierungsprinzipien. Tatsächlich ist es so, daß einer Methode m einer Klasse ein Objekt o dieser Klasse als ein (impliziter) Parameter zugeordnet wird. Alle in der Beschreibung der Methode formulierten Bezugnahmen auf Indizes der Klasse werden dadurch auf die entsprechenden Komponenten des Objekts o „umgelenkt“. Die Methode m wird auf das Objekt o angewendet. Also sind die Methoden Prozeduren bzw. Funktionen (hier zusammenfassend als Routinen bezeichnet) im klassischen Sinne, ausgerüstet mit einem zusätzlichen Parameter. In Modula-3 wird bei der Deklaration einer Methode sogar gefordert, daß dieser dann eine Routine als Wert zugeordnet wird,

deren erster Parameter sich immer auf das Objekt bezieht. Das ist ein Kniefall an die objektorientierte Programmierung, um die dort übliche Notation zu nutzen.

Ich halte die Trennung zwischen Methode und Objekt für die angemessene Betrachtungsweise, insbesondere bei abstrakten Modellen, wie sie durch Software gegeben sind. Der Computer wird nach wie vor durch die Software zu einer informationsverarbeitenden „Maschine“, auch wenn das Speichern der Information dabei ein unverzichtbarer Bestandteil ist. Den operationalen Bestandteilen der Software, also den Routinen, kommt damit eine besondere Bedeutung zu. Die Vorstellung, daß etwa eine natürliche Zahl die Fähigkeit besitzt, sich mit einer anderen natürlichen Zahl im Sinne einer Addition zu „paaren“, ist doch seltsam. Außerdem würde mir bei einer angestrebten Addition $3 + 4$ die Entscheidung schwer fallen, ob ich 3 mit 4 paaren lasse oder der 4 die Ehre gebe, dies mit der 3 zu tun. Auch ist die Einzahlung auf mein Gehaltskonto nicht eine Fähigkeit meines Kontos, sondern eine Operation meiner Bank, die diese auf Veranlassung durch die Landesstelle für Finanzen vornimmt. Die Bank ist statisch, mein Konto dynamisch (manchmal stärker, als mir lieb ist).

Selbst bei den sogenannten *Konstruktoren* und *Destruktoren*, das sind Methoden, die in einigen Programmiersprachen automatisch beim Erzeugen und Vernichten des entsprechenden Objektes aufgerufen werden, ist eine Bindung an das Objekt nicht zwingend. Es ist die Fähigkeit der Bank, ein Konto zu eröffnen und zu schließen, kaum die Fähigkeit des Kontos selbst. Die automatische Aktivierung des Konstruktors oder Destruktors beim Erzeugen bzw. Vernichten von Objekten ist eine der problematischen impliziten Annahmen bei der objektorientierten Programmierung. Dazu kommt, daß über Vererbung die Wirkung von Konstruktoren bzw. Destruktoren unübersichtlich werden kann. Statt implizit beim Erzeugen einen Konstruktor zu aktivieren sollte explizit ein Konstruktor als Routine definiert werden, die dann das Erzeugen veranlassen muß und die sonstigen gewünschten Aktivitäten wie Initialisierungen übernehmen kann. Das Erzeugen eines Objektes kann dann sowohl über den Konstruktor als auch direkt erfolgen. Immer aber muß explizit klar sein, welche Effekte entstehen.

Die dynamische Variante der Zuordnung von Methoden an die Objekte ist damit auch möglich (bei Oberon favorisiert), indem man Komponenten von Klassen einführt, deren Typen Variablen von Mengen von Routinen sind. Das hat die Konsequenz, daß die Methoden tatsächlich zum Objekt gehören! Damit ist der Vorteil verbunden, daß verschiedene Instanzen verschiedene Methoden in der gleichen Komponente haben *können* (nicht müssen!). Diese Zuordnung kann durch entsprechende explizite Konstruktoren erfolgen. Das entspricht dem dynamischen Binden. Es erfordert aber auch, daß Routinen als Werte aufgefaßt werden und Typen existieren, die gewisse Mengen von Routinen beschreiben. Das ist ein in vielen Programmiersprachen akzeptiertes Konzept. Der Typ von Routinen wird, analog zur funktionalen Programmierung, als die Menge der Parameter mit ihrem Typ angegeben.

Während allerdings in der funktionalen Programmierung nur Funktionen betrachtet werden, die keinerlei Nebeneffekte haben, ist in der imperativen Programmierung die Existenz von Nebeneffekten normal, in der objektorientierten Programmierung sogar essentiell, da ja eine Methode den „inneren“ Zustand eines Objektes beeinflussen soll.

Die Nebeneffekte von Routinen wirken sich auf eine gewisse *Umgebung* aus, die bei der Anwendung der Routine (dynamisch) verändert wird. Praktisch geht in die Umgebung die gesamte aktuelle Konstellation des Computers ein. Auswirken darf sich davon allerdings nur ein gewisser Teil davon, der von den Routinen des Programms angesprochen wird. Dazu gehören genutzte Dateien und als Stapel bzw. als Halde organisierte Speicherbereiche, die insbesondere zur Realisierung der Variablenbelegung $\sigma : \mathcal{V} \leftrightarrow U$ erforderlich sind.

Der Definitionsbereich $dom\sigma$ kann sich dabei genauso verändern, wie die Werte $\sigma(v)$ für gewisse Variable $v \in dom\sigma$. Das ist möglich, da dynamisch neue Variable geschaffen (mittels Routinen wie `new` bzw. die Aktivierung von lokalen Vereinbarungen) und auch vernichtet werden. Mit \mathcal{E} wird die Menge aller möglichen Umgebungen bezeichnet. Ist T der Typ des Argumentes einer Routine ρ , T' der Typ des expliziten Resultates, so ist ρ mathematisch beschrieben durch $\rho : \mathcal{E} \times T \leftrightarrow \mathcal{E} \times T'$. Um Mengen von Routinen zu beschreiben, kann man auf die Angabe des immer implizit existierenden Parameters \mathcal{E} verzichten. Zu beachten ist aber, daß im Prinzip hier nur ein expliziter Parameter zugelassen ist. Will man mehrere Parameter betrachten, so hat man den Typ T zu strukturieren (etwa als `record`). Zur Beschreibung derartiger Routinen werden Typ-Ausdrücke der Form $\tau \rightarrow \tau'$ benutzt.

Die Routine ρ kann auf jeden Wert $w \in T$ „angewendet“ werden. Also muß der Typ T_a des aktuellen Parameters eines korrekten Aufrufes von ρ eine Untermenge von T sein, auf die Typ-Ausdrücke bezogen heißt dies, daß $\tau_a \lesssim \tau$ sein muß, wenn τ_a den Typ des aktuellen Parameters beschreibt.

Nun sollen, wie allgemein üblich, Routinen höherer Ordnung, die Routinen als Parameter besitzen

dürfen, zugelassen werden. Damit hat die obige Betrachtung Auswirkungen auf die Festlegung der Relation \lesssim zwischen Typ-Ausdrücken der Form $\tau \rightarrow \tau'$, die Mengen von Routinen als Bedeutung besitzen. Um diese Bedeutungen zu beschreiben, soll etwas näher auf die Situation bei partiellen Funktionen $f : A \Leftrightarrow B$ eingegangen werden. Da in der Informatik die Berechnung das Anliegen ist, sollen hier partielle Funktionen immer berechenbar sein. Der Definitionsbereich $domf$ einer berechenbaren Funktion ist im allgemeinen unentscheidbar. Auch das Komplement des Definitionsbereiches ist unentscheidbar (sonst wäre $domf$ entscheidbar). Die Bedeutung der Beschreibung einer Routine ist eine berechenbare Funktion $\rho : \mathcal{E} \times T \Leftrightarrow \mathcal{E} \times T'$ mit $dom\rho \subseteq \mathcal{E} \times T$. Um bei dieser Beschreibung den Definitionsbereich möglichst genau anzugeben, ist T klein (im Sinne der Mengeninklusion) zu wählen. Es sind alle die Werte w nicht in T aufzunehmen, von denen bekannt ist, daß für jedes $e \in \mathcal{E} : (e, w) \notin domf$.

Es sei deshalb eine *entscheidbare* Beziehung zwischen Routinen und Typen, genannt **anwendbar**, gegeben. Diese Beziehung soll die folgende Implikation erfüllen:

Wenn ρ **nicht** auf w **anwendbar** ist, so gilt $(e, w) \notin dom\rho$ für alle $e \in \mathcal{E}$.

Mit $\rho : \mathcal{E} \times T \Leftrightarrow \mathcal{E} \times T'$ wird nun eine Routine $\rho : \mathcal{E} \times T \Leftrightarrow \mathcal{E} \times T'$ bezeichnet, für die **zusätzlich** gilt: $\forall w \in T : \rho$ ist **anwendbar** auf w .

Wenn eine Routine ρ_f als formaler Parameter innerhalb einer Routine ρ benutzt wird, so darf offenbar ρ_f durch den aktuellen Parameter ρ_a ersetzt werden, wenn ρ_a immer dann **anwendbar** ist, wo auch ρ_f **anwendbar** ist. Jeder Argumentwert von ρ_f muß auch als Argumentwert von ρ_a zugelassen sein. Mit anderen Worten: Der Argumenttyp von ρ_f muß Teilmenge des Argumenttyps von ρ_a sein. Die Umkehrung gilt für die Resultattypen. Ein solches Verhalten wird als *Contravarianz* (siehe [6]) bezeichnet. Es wird festgelegt:

Ein **routinen**-Typ beschrieben durch einen Typ-Ausdruck der Form $\tau \rightarrow \tau'$, wobei τ und τ' Typ-Ausdrücke sind, hat die Bedeutung:

$$\llbracket \tau \rightarrow \tau' \rrbracket = \{ \rho : \mathcal{E} \times T \Leftrightarrow \mathcal{E} \times T' \mid \llbracket \tau \rrbracket \subseteq T \wedge T' \subseteq \llbracket \tau' \rrbracket \}.$$

Als Konsequenz erhält man: $\hat{\tau} \rightarrow \hat{\tau}' \lesssim \tau \rightarrow \tau'$ genau dann, wenn $\tau \lesssim \hat{\tau}$ und $\hat{\tau}' \lesssim \tau'$.

7 Sichten und Vererbung

Der neuralgische Punkt ist allerdings die Frage, welche Änderungen der Umgebung, insbesondere der Variablenbelegungen, durch Routinen herbeigeführt werden können. Das geht aus dem Typ der Routine nicht hervor.

Die Nutzung der Methoden ist in der objektorientierten Programmierung das wesentliche Mittel, um den inneren Zustand eines Objektes zu verändern. Nebeneffekte sind damit essentiell. Lokal sind die Nebeneffekte immer auf die Komponenten des der Methode zugehörigen Objektes beschränkt. Global können öffentliche Komponenten anderer Objekte beeinflußt werden. Es gibt keinerlei Einschränkungen darüber, welche der öffentlichen Komponenten wo genutzt werden dürfen. Was öffentlich ist, darf beliebig genutzt werden, sobald ein entsprechendes Objekt mit einer solchen Komponente verfügbar ist. Es existiert zwar eine über die Vererbung (siehe unten) definierte Klassenhierarchie, aber alle Klassen in der Hierarchie sind frei nutzbar. Auch die Bildung von Cluster in Eiffel stellt nur eine Gruppierung dar, die keine Nutzungsrechte zwischen den Clustern fixiert.

Das Modulkonzept, wie etwa in MODULA II benutzt, läßt dagegen eine bedeutend differenziertere Beschreibung von *Sichten* zu. Ein *Modul* ist ein Teil einer Programmbeschreibung, in dem alle die Informationen zusammengefaßt werden, die unter einer gewissen Sicht, die natürlich vom Entwerfer eines Programms geprägt ist, enge Gemeinsamkeiten aufweisen. Ein Modul sollte das einzige Mittel sein, um Sichten zu formulieren, um Informationen zu verbergen, bzw. positiv ausgedrückt, Informationen verfügbar zu machen (zu exportieren). Die vom Modul bereitgestellten Informationen bzw. die von ihm in Anspruch genommenen Informationen (Importe) werden in seiner *Schnittstelle* angegeben. Es ist kein zwingender Grund erkennbar, daß (wie in MODULA-II) Moduln geschachtelt werden müssen. Alle Verbindungen zwischen Moduln erfolgen über die Schnittstelle. Über Exporte und Importe wird eine Verflechtung zwischen den Moduln beschrieben, die zu einer Hierarchie von Moduln führen kann, aber nicht muß. Die Beschreibung der Exporte und Importe soll detailliert sein.

Da die Modularisierung der Programmbeschreibung zur Strukturierung von Sichten dient, muß sie sich

auf der Ebene des Programms nicht reflektieren. Ein Programm als eine Menge von Werten braucht keine andere Strukturierung als sie durch die enthaltenen Werte gegeben ist. In der Programmbeschreibung muß man sich auf Informationen beziehen können, diese *identifizieren*. Damit ist es dort erforderlich, *Identifikatoren* zu *definieren*, d.h. an Informationen zu binden. Durch die Verfügbarkeit einer Menge von Identifikatoren einschließlich der angebotenen Information ist eine gewisse Sicht gegeben.

Nach dem hier verfolgtem Konzept sind die in der Programmbeschreibung enthaltenen Informationen Konstanten, deren Interpretation als Werte ins Programm eingehen, und Typen, die nur auf der Ebene der Programmbeschreibung existieren. Diese Informationen sind im allgemeinen strukturiert und können über Identifikatoren aufeinander Bezug nehmen.

Nach dieser Vorbereitung kann der Begriff des Moduls präzisiert werden:

Ein **Modul** besteht aus einer Schnittstelle und einer Menge von Definitionen. Eine **Definition** bindet einen Bezeichner an einen Typ oder eine Konstante. Die **Schnittstelle** eines Moduls besteht aus einem Export und einem Import. Der **Export** ist eine Menge von Identifikatoren, die im Modul sichtbar sind. Der **Import** ist eine Menge von Identifikatoren. Ein importierter Identifikator muß von einem Modul exportiert werden. Ein Identifikator ist in einem Modul **sichtbar**, falls er importiert oder im Modul definiert ist.

Indizes von **records** werden in der Programmbeschreibung über Identifikatoren definiert. Will man gewisse Indizes außerhalb des Moduls nutzbar machen, so muß man die entsprechenden Identifikatoren exportieren. Damit wird die Angabe von privaten und öffentlichen Indizes in der Beschreibung des **record**-Typs durch die Festlegungen im Export eines Moduls ersetzt. Das ist die Konsequenz aus der Auffassung, daß Sichten ausschließlich über den Modul beschrieben werden.

Nun kann das Modell meiner Bank in einem Modul spezifiziert werden. Der Landesstelle für Finanzen wird die Einzahlungsroutine exportiert, mir selbst dann zusätzlich die Auszahlungsroutine und freie Sicht auf einen Index, unter dem Bemerkungen eingetragen oder abgefragt werden können.

Mit Modula-3 ([14]) wurde eine Modifikation von Modula-II vorgenommen, um Konzepte der objektorientierten Programmierung aufzunehmen. Das Modulkonzept wurde zwar prinzipiell beibehalten, leider aber über ein zusätzliches Klassenkonzept aufgeweicht.

Für das Verwischen von Sichten ist meiner Meinung nach im großen Maße das *Vererbungskonzept* verantwortlich. Aus den im Anhang angegebenen Zitaten zur Vererbung (und den hier aus Platzgründen nicht aufgeführten zusätzlichen Erläuterungen) geht hervor, daß Vererbung ein in verschiedenen Programmiersprachen verfügbarer Mechanismus ist, um die Beschreibung von Klassen oder deren Teile nachzunutzen. Wie die angedeuteten Mechanismen im einzelnen wirken, hängt stark von der entsprechenden Programmiersprache ab.

Aus rein technischer Sicht ist Vererbung ein Mechanismus zur Reduktion von Schreibaufwand. Dies wird zum Beispiel in der Sprache Sather-K ([13]) dadurch deutlich, daß die Semantik der Vererbung ausschließlich durch textuelle Ersetzung beschrieben ist. Der Einfluß auf die systematische Programmgestaltung kommt dadurch zustande, daß zentrale Konzepte tatsächlich auch an zentralen Stellen beschrieben werden. Änderungen an diesen Stellen haben damit automatisch Auswirkungen auf alle Erben. Das ist allerdings zugleich eine große Gefahr, wenn nämlich unzureichende Übersicht über die Erben bzw. die Art und Weise, in der geerbt wird, vorhanden ist.

Die textuelle Ersetzung ist (soweit benutzt) die technische Seite der Vererbung. Inhaltlich wird durch Vererbung eine Beziehung zwischen Typen festgelegt. Nach meinem Verständnis ist das aber nicht unbedingt eine Unter- bzw. Obertyp Beziehung, wie in [23] behauptet. Es ist angebrachter, eine Klasse K' eine abgeleitete Klasse von K zu nennen, falls K' von K erbt. Ein Untertyp ist eine Untermenge, das muß bei einer abgeleiteten Klasse, zumindestens im hier vertretenen Konzept, nicht so sein. Wenn zum Beispiel ein **record**-Typ τ einen **record**-Typ τ' erbt, indem τ neben allen Komponenten von τ' noch zusätzliche Komponenten besitzt, so gilt **nicht**: $[[\tau]] \subseteq [[\tau']]$, also auch nicht $\tau \lesssim \tau'$! Auch sehe ich nicht die Notwendigkeit, Methoden **automatisch** in die Vererbung einzubeziehen. Routinen gehören bei mir primär zu Sichten (Moduln), sie können deshalb exportiert und importiert werden.

Wichtig erscheint mir die Klärung der Frage, welche Vererbungsmechanismen welche Typbeziehung herstellen und wie diese Typbeziehungen für die systematische Entwicklung korrekter Software genutzt werden können. Man findet viele Artikel und Bücher zu allgemeinen methodologischen Untersuchungen, insbesondere zum objektorientierten Entwurf, die aber wenig konkrete Schlußfolgerungen auf die Vererbungsmechanismen selbst zulassen. Einige bemerkenswerte Ansätze zur Einordnung von Vererbungsmechanismen findet man in [9, 10, 12]. Auf der Grundlage der Hoare-Logik zur Programmverifikation wird eine Klassifikation der Vererbung nach der Richtung der Inklusion, in der Vor- und Nachbedin-

gungen von Methoden aus Ober- und Unterklassen zueinander in Beziehung stehen, angegeben. Nun ist Programmverifikation eine nicht ganz einfache Angelegenheit. Das beginnt damit, daß die Spezifikation oft schon nicht der Intuition entspricht. Zum Beispiel wurde in der Arbeit [10], die sich immerhin speziell mit der Konstruktion korrekter Software beschäftigt, für die Methode *are_connected* (Seite 45) eine zu scharfe Nachbedingung angegeben, die nicht notwendigerweise erfüllt sein muß. Läßt man zur Formulierung von Vor- und Nachbedingungen den vollen Prädikatenkalkül der ersten Stufe zu, so sind Beweise unter Umständen sehr aufwendig und die Korrektheit der Beweise wieder ein eigenständiges Problem. Ich vertrete die Auffassung, solche Aussagen über Programmeigenschaften in verbindlichen Formulierungen zuzulassen, eventuell auch zu fordern, die mit vertretbarem Aufwand und möglichst automatisch überprüfbar sind. In einem ersten Schritt sollte das Typkonzept konsequent ausgebaut und die Typkorrektheit detailliert verifiziert werden.

Verständlich ist das Anliegen, daß eine Routine außer auf Argumente eines gewissen Typs τ auch auf alle Argumente der davon abgeleiteten Typen τ' angewendet werden kann. In der objektorientierten Programmierung ist das die Standardannahme. Nach der hier vorgestellten Konzeption ist das nicht zwingend, da nicht immer $\tau' \lesssim \tau$ gilt. Ich halte es aber auch für besser, eine solche Annahme nicht zwingend zu machen. Tatsächlich werden in der Praxis „vererbte“ Methoden oft neu definiert. Ist ein derartiges Verhalten gewollt, so sollte dies explizit beschrieben werden. Analog zu ADA95 (`class`-Typ) kann man etwa alle aus τ abgeleiteten Typen τ' in einen Typ zusammenfassen.

Hier soll Vererbung ausschließlich auf die Erweiterung der Komponenten von zusammengesetzten Typen (etwa `record`-Typen) beschränkt werden. Der Wert eines `record`-Typs ist eine Funktion, die jedem Index einen Wert zuordnet. Falls nun der Typ τ' von τ abgeleitet ist, so kann man jeden Wert $f \in \llbracket \tau \rrbracket$ zu einem Wert $f' \in \llbracket \tau' \rrbracket$ machen, indem f um die neuen Indizes erweitert wird. Es gilt dann $f \subseteq f'$. Umgekehrt kann f' durch Einschränkung auf Indizes in τ zu einem Wert $f \in \llbracket \tau \rrbracket$ machen. Aus diesem Grund soll die Vererbung mit dem Symbol \sqsubseteq beschrieben werden.

Vererbung ist eine partielle Ordnung \sqsubseteq über Typ-Ausdrücken. Ein Typ τ' erbt einen Typ τ , falls $\tau \sqsubseteq \tau'$.

Die Vereinigung aller der Typen, die einen Typ τ erben, wird nun als Klasse bezeichnet und mittels Klassentyp beschrieben. Damit ist ein Klassentyp eine Verallgemeinerung der Vereinigung von Typen, wie sie in manchen Sprachen existieren (z.B. `unions` in C):

Ein **Klassentyp**, beschrieben durch einen Typ-Ausdruck der Form `class τ` , wobei τ ein Typ-Ausdruck ist, hat die Bedeutung:

$$\llbracket \text{class } \tau \rrbracket := \bigcup \{ \llbracket \tau' \rrbracket \mid \tau \sqsubseteq \tau' \}.$$

Als Konsequenz erhalten wir: wenn $\tau \sqsubseteq \tau'$, so `class τ'` \lesssim `class τ` .

8 Polymorphismus

Im Zusammenhang mit Vererbung und der Oberklassen - Unterklassen Beziehung wird bei der objektorientierten Programmierung oft der Polymorphismus erwähnt. Aus [2] geht lediglich hervor, daß die Argumente einer polymorphen Routine von verschiedenem Typ sein können. Um trotzdem eine Übersicht über die möglichen Typen der Argumente zu haben, werden *polymorphe Typen* eingeführt ([8]).

In [11] werden die zwei Hauptkategorien, *universal* und *ad hoc* Polymorphismus, unterschieden. Beim *universal*-Polymorphismus hat eine Routine für verschiedene Typen der Parameter gleiches Verhalten. So wird Polymorphismus auch in [23] beschrieben. Ziel ist es, die gleiche Beschreibung einer Routine für Parameter aus unterschiedlichen Typen nutzen können. Insbesondere bei der Vererbung kann eine Routine auf alle Argumente mit Typen, die einen gewissen Typ erben, angewendet werden. Genau dies wird durch den im vorangegangenen Abschnitt behandelten Klassentyp ermöglicht. Allerdings ist dies kein echter Polymorphismus, sondern dieses Verhalten ist durch einen Typ beschrieben, der sich aus der Vereinigung einer gewissen Menge von Typen ergibt.

Darüber hinaus sind aber auch kompliziertere Sachverhalte, durch den sogenannten *parametric*-Polymorphismus [11] beschrieben, erwünscht. Als Beispiel kann man etwa eine Routine nehmen, die für Listen als Argument deren Länge (die Anzahl der Elemente) bestimmt. Die Beschreibung einer solchen Routine ist unabhängig vom Typ der Elemente. Eine solche Routine „arbeitet“ gleichermaßen für Listen mit `integer`-Elementen, `string`-Elementen, usw. Das heißt, die Argumente können sowohl aus der Menge der `integer`-Listen, der `string`-Listen, usw. sein. Also ist ein beliebiges Element aus der

Vereinigung aller dieser Typen zugelassen.

Die Vereinigung von Typen als Typkonstruktion ist zu aber zu ausdrucksarm, um obiges Ziel zu erreichen. Man will zum Beispiel auch die Menge aller Listen beschreiben können, deren Elemente aus Paaren von Elementen besteht, deren beiden Komponenten vom gleichen, aber beliebigen Typ sind. In den funktionalen Sprachen (ML, Miranda,...) beschreibt man solche Typen dadurch, daß sogenannte *Typvariable* benutzt werden, die an die Stelle von Typen treten können. Eine Typvariable darf durch einen beliebigen Typ ersetzt werden. Ein polymorpher Typ ist ein Typ, der Typvariable enthält (Siehe auch [23]). Zu beachten ist, daß in den funktionalen Sprachen der Begriff der Variablen in einem anderen Sinne gebraucht wird als in den imperativen Sprachen. Bei den funktionalen Sprachen hat eine Variable den Charakter eines Parameters, der durch einen beliebigen Wert ersetzt werden kann. Deshalb soll hier der Begriff *Typ-Parameter* benutzt werden.

Dieses Konzept soll dadurch übernommen werden, daß in Typ-Ausdrücken auch spezielle *Typ-Parameter* erlaubt sind. Mit T_par soll die Menge aller Typ-Parameter und mit T_p_exp sollen alle solche Typ-Ausdrücke, die Typ-Parameter enthalten können, bezeichnet werden. T_exp bezeichnet weiterhin die Menge aller Typ-Ausdrücke ohne Typ-Parameter. Eine Funktion $\varphi : T_par \rightarrow T_exp$, die jedem Typ-Parameter einen Typ-Ausdruck zuweist, heißt eine *Substitution*. Jede Substitution φ kann eindeutig zu $\varphi^* : T_p_exp \rightarrow T_exp$ erweitert werden. Für einen beliebigen Typ-Ausdruck τ ist $\varphi^*(\tau)$ der Ausdruck, der entsteht, wenn jeder Typ-Parameter x in τ durch $\varphi(x)$ ersetzt wird.

Die Bedeutung eines Typ-Parameters ist nun die Vereinigung aller der Typen, die durch eine beliebige Ersetzung der Typ-Parameter mit Typen erhalten werden. Bei dieser Ersetzung muß darauf geachtet werden, daß gleiche Typ-Parameter gleich ersetzt werden. Wichtig ist weiterhin, daß nach der Ersetzung keine Typ-Parameter mehr vorhanden sind.

$$\llbracket x \rrbracket := \bigcup \{ \llbracket \varphi(x) \rrbracket \mid \varphi : T_par \rightarrow T_exp \}.$$

Diese Definition ist analog für **record**-Typen τ anwendbar:

$$\llbracket \tau \rrbracket := \bigcup \{ \llbracket \varphi^*(\tau) \rrbracket \mid \varphi : T_par \rightarrow T_exp \}.$$

Als Konsequenz erhält man, daß $\varphi(\tau) \lesssim \tau$ falls τ ein beliebiger Typ-Parameter oder ein **record**-Typ ist. Für andere Typen ist das nicht notwendig so. Zum Beispiel bei den polymorphen **routinen**-Typen, deren Bedeutung jetzt definiert ist durch:

$$\llbracket \tau \rightarrow \tau' \rrbracket := \{ \rho : \mathcal{E} \times T \leftrightarrow \mathcal{E} \times T' \mid \llbracket \tau \rrbracket \subseteq T \wedge T' \subseteq \llbracket \tau' \rrbracket \wedge (\forall \varphi, \sigma, w : w \in \llbracket \varphi^*(\tau) \rrbracket \wedge \rho(\sigma, w) = (\sigma', w') \Rightarrow w' \in \llbracket \varphi^*(\tau') \rrbracket) \}.$$

Auf diese Art kann man zum Beispiel auch ausdrücken, daß eine Funktion (etwa **head**) Listen mit Elementen eines beliebigen Typs als Argument akzeptiert und als Resultat einen Wert aus dem Typ der Elemente liefert. Funktionen, deren Typ polymorph ist, werden auch polymorphe Funktionen genannt (Siehe [8]). Bezüglich der Untertyp-Relation \lesssim erhält man:

$$\hat{\tau} \rightarrow \hat{\tau}' \lesssim \tau \rightarrow \tau' \text{ genau dann, wenn für alle Substitutionen } \varphi : \varphi^*(\tau) \lesssim \varphi^*(\hat{\tau}) \text{ und } \varphi^*(\hat{\tau}') \lesssim \varphi^*(\tau').$$

Zur Definition der Bedeutung der Typ-Ausdrücke ist es ausreichend, wenn man sich auf solche Substitutionen beschränkt, die alle Typ-Parameter eliminieren, wie dies oben praktiziert wurde. Für die Typinferenz zur Überprüfung der Typsicherheit und zur induktiven Definition der Relation \lesssim benötigt man jedoch allgemeinere Substitutionen der Form $\varphi : T_par \rightarrow T_p_exp$, die dann eindeutig zu $\varphi^* : T_p_exp \rightarrow T_p_exp$ erweitert werden können. Auf diese Problematik soll hier aber nicht weiter eingegangen werden.

Aus [1, 12] kann man entnehmen, daß Methoden (bei mir Routinen genannt) auf unterschiedliche Argumente unterschiedlich reagieren können (*ad hoc* Polymorphismus). Das ist keine Besonderheit, denn z.B. liefert die Addition für die Argumente 3 und 4 ein anderes Resultat als für die Argumente 3 und 5. In [2, 11] kommt zum Ausdruck, daß die unterschiedliche Verhaltensweise insbesondere durch die verschiedenen möglichen Typen der Argumente bestimmt wird. So kann man sowohl Festpunktzahlen als auch Gleitpunktzahlen addieren. In ersten Fall entsteht eine Festpunktzahl als Resultat, im zweiten Fall eine Gleitpunktzahl. Diesen Sachverhalt könnte man durch einen Typ-Ausdruck der Form $x \rightarrow x$ beschreiben, für den x ein Typ-Parameter ist und als Substitutionen nur $\varphi(x) \in \{\mathbf{fixed}, \mathbf{float}\}$ zugelassen sind. Noch allgemeiner sind **overflow**-Typen:

Ein **overload**-Typ, beschrieben durch einen Typ-Ausdruck der Form $\tau_1 \rightarrow \tau'_1 \mid \dots \mid \tau_n \rightarrow \tau'_n$, wobei $\tau_1, \dots, \tau_n, \tau'_1, \dots, \tau'_n$ beliebige Typen sind, hat die Bedeutung:

$$\llbracket \tau_1 \rightarrow \tau'_1 \mid \dots \mid \tau_n \rightarrow \tau'_n \rrbracket := \{ \rho : \mathcal{E} \times T \leftrightarrow \mathcal{E} \times T' \mid \forall i, \sigma, w : \llbracket \tau_i \rrbracket \subseteq T \wedge T' \subseteq \llbracket \tau'_i \rrbracket \wedge w \in \llbracket \tau_i \rrbracket \wedge \rho(\sigma, w) = (\sigma', w') \Rightarrow w' \in \llbracket \tau'_i \rrbracket \}.$$

Aus einem solchen **overload**-Typ geht nicht hervor, daß für verschiedene Eingaben verschiedene Varianten von Routinen genutzt werden sollen. Diese Aussage ist im Rahmen des Typkonzeptes auch nicht notwendig.

Wichtig ist dann, daß zur Realisierung von **overload**-Routinen Möglichkeiten zur Abfrage bereitstehen, ob ein Wert Element eines gewissen Typs ist. Mit Hilfe einer solchen Abfrage kann dann auf den unterschiedlichen Typ eines Argumentes unterschiedlich reagiert werden. Wenn diese Information auf der Grundlage eines strengen Typkonzeptes bereits bei der Beschreibung des Aufrufes der Routine zur Verfügung steht (oder abgeleitet werden kann), so ist eine Abfrage bei der Anwendung des Programms hinfällig und kann aus Effizienzgründen entfallen. Ist dies aber nicht möglich, muß der Programmierer dynamische Vorsorgen treffen. Das Typkonzept sollte es erlauben, aus der Programmbeschreibung heraus erkennen zu lassen, welche Informationen bereitstehen und wie diese genutzt werden.

Allerdings wird **overloading** von Routinen oft nur genutzt, um verschiedene Routinen, die ähnliche Effekte haben, gleich bezeichnen zu können, zum Beispiel die Addition von Festpunktzahlen und die Addition von Gleitpunktzahlen. Mathematisch gesehen, sind das verschiedene Funktionen. Sieht man von Problemen ab, die sich aus der Verwendung von Bezeichnern ergeben, so kann auf **overloading** verzichtet werden. In diesem Sinne sind auch Routinen, die auf unterschiedliche Typen der Argumente unterschiedlich reagieren, überflüssig. Man kann sie durch eine entsprechende Menge von verschiedenen Routinen ersetzen.

Wie weit Typkonzepte ausgebaut werden können ist unter anderem Gegenstand der Typtheorie. Das betrifft auch den Polymorphismus. In [6, 22] wird die Semantik solcher Konzepte durch Ableitungsmechanismen beschrieben. Das ist zwar sehr formal und für den „normal gebildeten“ Programmierer wahrscheinlich zu schwer verständlich, bildet aber eine saubere Basis für die Auswahl der Konzepte.

9 Schlußfolgerungen

In diesem Artikel wurden einige Begriffe und die damit in Zusammenhang stehenden Konzepte kritisch betrachtet sowie eigene Vorschläge vorgestellt. Die Hauptmotive, die mich zu den gemachten Vorschlägen veranlassten, wurden in der Einleitung genannt. Es zeigte sich, daß zwei Konzepte auf der Beschreibungsebene eine zentrale Rolle spielen:

- das **Modulkonzept** zur Beschreibung von Sichten und
- das **Typkonzept** zur Beschreibung von verifizierbaren Aussagen über das Programmverhalten.

Beide Konzepte sind nicht neu. Sie sind in verschiedenen Programmiersprachen mit unterschiedlicher Konsequenz und Zielrichtung eingesetzt. Hier wurde nur angedeutet, daß die Nachnutzung und sinnvolle Kombination bewährter Konzepte sinnvoller sein kann, als die Einführung neuer Techniken.

Es ist angedacht, die hier entwickelten Ansätze in einem konkreten Programmiersystem zu implementieren. Ein solches Programmiersystem soll hauptsächlich die Informationen der Programmbeschreibung verwalten, also als *Informationssystem* ausgelegt sein. Es erlaubt, beim Aufbau der Programmbeschreibung schrittweise Informationen aufzubauen und zu modifizieren. Neben den Möglichkeiten der ausführlichen Information über Eigenschaften des Teilproduktes muß in jedem Entwicklungsschritt dessen Konsistenz gesichert werden. Das entspricht der Prüfung von Integritätsbedingungen in Informationssystemen. Die textuelle Repräsentation der Programmbeschreibung wird dabei nur fragmentarisch existieren. Vielmehr soll die Entwicklung von Programmen durch die Interaktion mit dem Programmiersystem erfolgen. Das entlastet von vielen Problemen, die in Programmiersprachen bzw. auf Programmiersprachen basierenden Entwicklungssystemen durch die textuelle Repräsentation entstehen. Dazu gehören zum Beispiel

- komplizierte syntaktische Konstruktionen mit technischen Zeichen wie Trennern (Semikolon, Kolon, Komma, Klammerzeichen) und deren Beherrschung durch den Entwickler,
- Navigation in Texten, um zum Beispiel den Scope von Bezeichnern zu erkennen,
- umfassende Qualifikation von Bezeichnern, um die Eindeutigkeit bei Doppelverwendung zu sichern,

- Einschränkung der Länge gewisser Programmteile, um die Lesbarkeit zu erleichtern,
- künstliche Anordnungen, die durch die lineare Ordnung eines Textes gegeben sind, aber inhaltlich nicht relevant sind (z.B. Reihenfolge von Deklarationen),
- und so weiter.

Natürlich sind auch die gegenwärtig verfügbaren Entwicklungssysteme mit Komponenten ausgerüstet, die zur Lösung der genannten Probleme beitragen. So sind ausgefeilte Editoren und darauf basierende Navigationstools (Browser) verfügbar. Visuelle Komponenten erleichtern die Entwicklung spezieller Software, z.B. graphischer Oberflächen. Immer aber basieren diese Systeme auf gewissen Programmiersprachen, erfordern damit Compiler, die Texte analysieren und strukturieren. Erst nachdem ein Compilerlauf erfolgt ist, kann Information über das Produkt zur Verfügung gestellt werden.

Im hier angedeuteten Ansatz wird der Compiler mit den Teilen Scanner und Parser überflüssig. Auch ein Texteditor entfällt. Das Informationssystem arbeitet wie ein inhaltlich orientierter Editor für die Verwaltung der entsprechenden Informationen.

Folgende Präzisierungen sind für das Vorhaben erforderlich:

- Ausbau des Typkonzeptes,
- Bestimmung der Inferenzregeln zur impliziten Definition der Relation \lesssim , so daß aus $\tau \lesssim \tau'$ immer $[\tau] \subseteq [\tau']$ folgt,
- explizite Angabe der Vererbungsrelation \sqsubseteq zwischen Typ-Beschreibungen, so daß aus $\tau \sqsubseteq \tau'$ auch $\text{class } \tau' \lesssim \text{class } \tau$ folgt,
- Struktur und Bedeutung von Konstanten entsprechend den verfügbaren Typen und deren Beschreibungstechnik,
- erforderliche Integritätsbedingungen und Techniken ihrer Absicherung,
- Art und Weise der Interaktion des Informationssystems mit dem Nutzer,
- interne Repräsentation der durch das Informationssystem verwalteten Programmbeschreibung,
- Transformation der Programmbeschreibung in ein Programm,
- Interpretation von Programmen bzw. Transformation von Programmen in herkömmliche Programmiersprachen oder Maschinencode,
- Form der externen Dokumentation der Programmbeschreibung, um eine druckbare Variante zu erhalten.

A Zitate

Variable:

[2, S. 35]: „...,the combination of name, type and value ist called a **variable**.“

[8, S. 137]: „A *variable* is a storage object in which a pure value may be stored.“

[12, S. 3]: „Im zustandsorientierten Programmieren ist eine **Variable** ein Tripel (Referenz, Behälter, Wert).“

[15, S. 158]: „Eine **Variable** ist ein Objekt, dessen gespeicherter Wert sich während der Ausführung ändern kann.“

[23, S. 39]: „Beim Programmieren ist eine **Variable** ein Objekt, das einen Wert enthält. Dieser Wert kann beliebig oft inspiziert und überschrieben werden.“

Typ:

[4, S. 34]: „Every data item in a computer has a representation and a number of properties collectively known as its 'type'“.

[12, S. 146]: „In Programmiersprachen bezeichnen wir Merkmale und Verhalten von Konstanten, Variablen und sonstigen Objekten als ihren Typ.“

[15, S. 193]: „Ein **Datentyp** ist eine Menge von Werten.“

[23, S. 10]: „Was genau ist ein Typ? Die naheliegendste Antwort ist vielleicht, daß ein Typ eine Menge von Werten ist. Wenn wir sagen, daß v den Typ T hat, so meinen wir lediglich, daß gilt: $v \in T$.“

Konstante:

[12, S. 3]: „Eine Variable heißt eine **Konstante**, wenn ihr Wert unveränderlich ist“.

[15, S. 161]: „Eine Konstante ist ein Sprachelement, das für die Dauer des Programms einen festen Wert hat.“

[20, S. 163]: „A **named constant** is a variable that is bound to a value only at the time it is bound to storage; its value cannot be changed by assignment or by an input statement.“

Objekt:

[2, S. 117]: „Such a variable of an abstract type (class type) is called an **object**.“

[4, S. 59]: „Eine Nachricht N zusammen mit der ihr zugeordneten Information J soll hinfort **Objekt** genannt werden.“

[11, S. 195]: „Ein *Objekt* ist als Abstraktion eines „realen Dinges“ zu verstehen, das einen inneren Zustand besitzt, der nur mittels Methoden (i.e. Operationen) des Objektes manipuliert werden kann.“

[12, S. 146]: „Ein **Objekt** ist ein elementares Teilsystem. Es repräsentiert einen beliebigen Gegenstand (Person, Ding, Thema, Sachverhalt) und besitzt meßbare, durch Werte erfassbare Eigenschaften. Wir nennen sie **Attribute** des Objekts. Ferner kann ein Objekt Tätigkeiten ausführen: Personen handeln und reagieren auf Ereignisse in der Umwelt, Sachverhalte können sich ändern.“

[15, S. 151]: „Ein zugeteilter Speicherbereich wird hier **Objekt** genannt.“

[23, S. 110]: „Ein Objekt ist eine verborgene Variable, zusammen mit einer Menge exportierter Operationen (Funktionen, Prozeduren usw.)“.

Klasse:

[2, S. 116]: „A **class** is a description of a kind (category) of object. A class contains:

- A description of the **internal variables** of objects of the class.
- the **operations** that can be applied to objects of the class.“

[11, S. 196]: „Mit dem Konzept des Objekts eng verbunden ist der Begriff der Klasse. Sie beschreibt eine Menge von Objekten hinsichtlich Struktur, Hierarchie in Bezug auf Vererbung und Verhalten. ...Die Klasse stellt also eine Schablone dar, nach der Objekte (dieser Klasse) erzeugt werden können.“

[12, S. 146]: „Jedes Objekt gehört zu einer Menge von Objekten mit gleichen Merkmalen und gleichem Verhalten. Wir sagen, die Objekte gehören zur gleichen **Klasse** und bezeichnen sie als **Instanzen** oder **Ausprägungen** der Klasse. Die Klasse repräsentiert das Konzept oder den abstrakten Begriff, den wir uns von ihren Objekten machen. Technisch gibt eine Klasse das Baumuster ihrer Objekte wieder. In Programmiersprachen bezeichnen wir Merkmale und Verhalten von Konstanten, Variablen und sonstigen Objekten als ihren Typ. Die Wörter *Klasse* und *Typ* betonen unterschiedliche Aspekte des gleichen Begriffs.“

[23, S. 232]: „Mit einer *Klasse* können gleichartige Objekte mit denselben Operationen definiert werden und in beliebiger Zahl erzeugt werden.“

Vererbung:

[4, S. 118]: „**Inheritance** allows us to define new classes by extending existing classes.“

[8, S. 50]: „Inheritance means that any function defined for a superclass also applies to all subclasses“.

[11, S. 195]: „...ist eine Sprache dann objektorientiert, wenn sie folgende Sprachmittel zur Verfügung stellt: Objekte, Klassen und Vererbung“.

[11, S. 196]: „Unter Vererbung ist jener Mechanismus zu verstehen, der es ermöglicht, Attribute und Methoden von Klassen an andere Klassen weiterzugeben.“

[15, S. 376]: „Vererbung ist ein wichtiger Mechanismus in objektorientierten Sprachen. Er ermöglicht die gemeinsame Nutzung von Daten und Operationen zwischen Klassen.“

[23, S. 133]: „Bei *Vererbung* geht es um Typsysteme mit Unter- und Obertypen, wobei Untertypen insbesondere Operationen von ihren Obertypen erben können.“

Polymorphismus:

[1, S. 65]: „*Polymorphism*, meaning “many forms,” refers to the ability of different objects to respond to the same message differently, e.g. ...“.

[4, S. 150]: „A polymorphic function accepts actual parameters of different types for a single formal parameter.“

[8, S. 520,521]: „A *polymorphic type* is a single type definition that includes two or more alternative type declarations. ...“

[11, S. 196]: „Unter Polymorphismus ist die Fähigkeit einer Größe zu verstehen, zur Laufzeit unterschiedliche Ausprägungen annehmen zu können. Während in monomorphen Programmiersprachen Funktionen bzw. Prozeduren und ihre Parameter, Operatoren und ihre Operanden, etc. einen eindeutigen Typ besitzen, erlauben polymorphe Sprachen dafür mehr als einen Typ.“

[12, S. 149, 174]: „Die Anforderung eines Dienstes ist ein **polymorpher Methodenaufruf**: Der Aufrufer sagt, was er von einem Objekt *a* will. Das Objekt *a* bestimmt, wie es den geforderten Dienst erbringt.“

„Methoden in Oberklassen können bei Verhaltensgleichheit **polymorph** aufgerufen werden; das Objekt interpretiert den Aufruf entsprechend der Unterklasse, zu der es gehört“

[20, S. 543]: „Polymorphism denotes the particular kind of dynamic binding that occurs in a language that incorporates inheritance. Specifically, **polymorphism** is a typing concept in which a specific message may be sent to different instances of different classes at different times.“

A *polymorphic function* is a function that accepts arguments of a polymorphic type.“

[23, S. 133]: „Bei *Polymorphie* geht es um Abstraktionen, die auf Werten verschiedener Typen einheitlich arbeiten.“

[23, S. 143]: „Ein **Polytyp** ist ein Typ, der eine oder mehrere Typvariablen enthält.“

Literatur

- [1] D. Appleby, *Programming Languages - Paradigm and Practice*, McGraw-Hill International Editions, 1991.
- [2] H.E. Bal, D. Grune, *Programming Language Essentials*, Addison Wesley, 1994.
- [3] J. de Bakker, E. de Vink, *Control Flow Semantics*, The MIT Press; Cambridge, Massachusetts; London, England, 1996.
- [4] F.L. Bauer, G. Goos, *Informatik, Eine einführende Übersicht, Erster Teil*, Springer Verlag, 1973.
- [5] B. Carpenter, *The Logic of Typed Feature Structures*, Cambridge University Press, 1992.
- [6] G. Castagna, *Object-Oriented Programming, A Unified Foundation*, Birkhäuser, Boston 1997.
- [7] O.L. Dahl, B. Myhrhaug, K. Nygaard, *SIMULA 67 - Common Base Language*, Norwegian Computing Center Publication No. 2, Oslo 1968.
- [8] A.E. Fischer, F.S. Grodzinsky, *The Anatomy of Programming Languages*, Prentice-Hall, 1993.
- [9] A. Frick, R. Neumann, W. Zimmermann, *Eine Methode zur Konstruktion robuster Klassenhierarchien*, Softwaretechnik-Trends, Mitteilungen der Fachgruppen, Band 16, Heft 3, Sptember 1996, S. 16-23.
- [10] A. Frick, W. Zimmer, W. Zimmermann, *Über die Konstruktion robuster objektorientierter Klassenbibliotheken*, Softwaretechnik-Trends, Mitteilungen der Fachgruppen, Band 15, Heft 3, Oktober 1995, S. 16-23.
- [11] H. Gall, M. Hauswirth, R. Klösch, *Objektorientierte Konzepte in Smalltalk, C++, Objective-C, Eiffel und Modula-3*, Informatik-Spektrum 18/4, 1995, S. 195-202.
- [12] G. Goos, *Vorlesungen über Informatik, Band 2: Objektorientiertes Programmieren und Algorithmen*, Springer, 1996.
- [13] G. Goos, H. Schmidt, *Sather-K, The Language*, Universität Karlsruhe, Fakultät für Informatik, Monash University, Department Software Development, 4/96.
- [14] S. P. Harbison, *MODULA-3*, Prentice Hall, 1992.
- [15] K.L. Loudon, *Programmiersprachen - Grundlagen, Konzepte, Entwurf*, International Thomson Publishing GmbH, 1994.
- [16] B. Meyer, *EIFFEL, The Language*, Prentice Hall, 1992.
- [17] H.D. Mills, *Mathematical Foundation for Structured Programming*, Federal Systems Division, IBM Corporation, Gaithersburg, Maryland 20760.
- [18] M. Reiser, N. Wirth, *Programming in OBERON, Steps beyond Pascal and Modula*, Addison-Wesley, 1992.2
- [19] G.Riedewald, J.Maluszynski, P.Dembinski, *Formale Beschreibung von Programmiersprachen*, Akademie-Verlag Berlin, 1983.
- [20] R.W. Sebesta, *Concepts of Programming Languages*, The Benjamin/Cummings Publishing Company, Inc., 1993.
- [21] K.Slonneger, B.L. Kurtz, *Formal Syntax and Semantics of Programming Languages*, Addison-Wesley Publishing Company, 1995.
- [22] S. Thompson, *Type Theory and Functional Programming*, Addison-Wesley Publishing Company, 1991.
- [23] D.A. Watt, *Programmiersprachen - Konzepte und Paradigmen*, Carl Hanser Verlag, 1990.

- [24] J. F. H. Winkler, *Type Compatability for Extensible Module Types, Their Reference Parameters, and Their Pointer Types*, Friedrich-Schiller-Universität Jena, FORSCHUNGSERGEBNISSE Der Fakultät für Mathematik und Informatik, Nr. Math/Inf/96/41.
- [25] G. Winskel, *The Formal Semantics of Programming Languages, An Introduction*, The MIT Press; Cambridge, Massachusetts; London, England, 1993.
- [26] D. R. Musser, A. Saini, *STL Tutorial and Reference Guide*, Addison-Wesley Publishing Company, 1996.