Brauer, Johannes

Working Paper

Typen, Objekte, Klassen - Teil 1: Grundlagen

Arbeitspapiere der Nordakademie, No. 2009-05

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

ARBEITSPAPIERE DER NORDAKADEMIE
ISSN 1860-0360

Nr. 2009-05

Typen, Objekte, Klassen – Teil 1: Grundlagen

Prof. Dr.-Ing. Johannes Brauer

Juni 2009

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

NORDAKADEMIE
HOCHSCHULE DER WIRTSCHAFT
THEORIE PRAXIS
Köllner Chaussee 11
25337 Elmshorn
http://www.nordakademie.de

Typen, Objekte, Klassen – Teil 1: Grundlagen

Johannes Brauer

2. Juli 2009

In diesem ersten Teil einer Reihe von Arbeitspapieren werden Grundbegriffe der Typtheorie (Datentyp, Typkompatibilität, Typsystem) von Programmiersprachen eingeführt. Es wird eine Klassifizierung von Typsystemen vorgenommen. Zum Schluss wird eine kurze Übersicht über die Themen nachfolgender Arbeitspapiere gegeben.

1 Motivation

Die Verwendung der Begriffe *Klasse*, *Typ* und *Objekt* ist in der Software-Technik – insbesondere im Kontext der objektorientierten Programmierung – allgegenwärtig, so dass die Vermutung nahe liegt, es gäbe für sie auch allgemein anerkannte Definitionen. Schaut man in die einschlägige Fachliteratur[1], stellt man sehr schnell fest, dass es zahlreiche Ansätze gibt, diese Begriffe zu definieren, die aber nicht alle zu den gleichen Ergebnissen kommen. So wird z. B. der Typbegriff mal mit dem Mengenbegriff assoziiert (Typen sind Wertemengen), mal werden Typen bzw. Typkompatibilität rein syntaktisch – nur Typbezeichner und Funktionssignaturen sind relevant – mal semantisch betrachtet. Die Definitionen sind dabei häufig von der Sichtweise einer bestimmte Programmiersprache[2] geprägt. Es hat sich daher in der Praxis der Ausbildung von Studierenden der Informatik als schwierig herausgestellt, ein in sich geschlossenes, konsistentes Verständnis dieser Begriffe zu vermitteln. Eine genaue Kenntnis dieser Begriffe ist aber nicht nur wichtig, weil sie zu den grundlegenden Konzepten moderner Programmiersprachen gehören, sondern auch für die adäquate Anwendung entsprechender Sprachkonzepte in der Programmierung erforderlich. Fragen wie

- Was ist der Typ eines Objekts?
- Was ist der Unterschied zwischen einem Typ und einer Klasse? oder
- Was ist ein abstrakter Datentyp?

können auch von Studierenden höherer Semester oft nicht beantwortet werden.

Erschwerend kommt hinzu, dass es weitere Begriffe gibt, die mit den bereits genannten – wiederum häufig abhängig von ihrer Ausprägung in Programmiersprachen – auf vielfältige Art verwoben sind. Zu diesen Begriffen zählen u. a. *Typkompatibilität*, *Unterklasse*, *Untertyp*, *Modul*, *Schnittstelle*, *Vererbung*, *Polymorphie*, *Kovarianz* und *Kontravarianz*.

Es wäre für die Software-Technik im Allgemeinen aber erst recht für die Ausbildung sehr vorteilhaft, wenn es eine Art „Typenlehre“ gäbe, die einerseits theoretisch fundiert, andererseits aber

[1] zum Beispiel:

Friedman, D. P., and Wand, M. *Essentials of Programming Languages*, third ed. The MIT Press, 2008; Turbak, F., and Gifford, D. *Design Concepts in Programming Languages*. The MIT Press, 2008; and Bruce, K. B. *Foundations of Object-Oriented Languages*. The MIT Press, 2002

[2] Es hat auch Anstrengungen gegeben, programmiersprachen-unhängige Taxonomien von Datentypen zu bilden, z. B. in

Meek, B. L. A taxonomy of datatypes. *SIGPLAN Notices 29*, 9 (1994), 159–167

auch in einer Form „lehrbar“ ist, dass sie auch einen praktischen Nutzen erbringt und nicht nur der „Erbauung“ von Vertretern der Theoretischen Informatik dient. Es soll hier keinesfalls bestritten werden, dass es bereits bedeutende Arbeiten – wie z. B. der Aufsatz von Luca Cardelli und Peter Wegner[3] – auf diesem Gebiet gibt, jedoch erscheint fraglich, ob diese Darstellungen für den Unterricht in Bachelor-Studiengängen geeignet sind.

[3] Cardelli, L., and Wegner, P. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv. 17*, 4 (1985), 471–522

Eine Typenlehre ist auch nicht als akademische Übung anzusehen, sondern im Kontext des methodischen Programmierens bedeutsam. Dahinter steht die grundsätzliche Frage: Wie viel Theorie braucht man, um ordentlich programmieren zu können? Dafür ist meines Erachtens unabdingbar, grundlegende Konzepte von Programmiersprachen zu kennen, sonst kann man sie auch nicht adäquat anwenden. Darüber hinaus ist man auch nur so in der Lage, wirkliche Innovationen bei Programmiersprachen von „altem Wein in neuen Schläuchen“ zu unterscheiden.

Der Autor erhebt nicht den Anspruch, mit diesem Arbeitspapier[4] eine solche Typenlehre zu erschaffen. Vielmehr soll der Versuch unternommen werden, ein Begriffsgebäude von „unten nach oben“ zu errichten, in der Hoffnung damit die Grundlage für das Verständnis komplexer Zusammenhänge legen zu können und Studierende in die Lage zu versetzen, die grundlegenden Sprachkonzepte aus konkreten Programmiersprachen herauslösen und von kompliziertem Beiwerk befreien zu können. Damit sind Studierende zukünftig hoffentlich besser in der Lage, neue Entwicklungen bei Programmiersprachen einzuordnen und adäquat zu nutzen.

[4] und den nachfolgenden

In den folgenden Abschnitten werden zunächst grundlegende Gedanken zum Thema *Typen* entwickelt.

2 *Interpretation von Bitmustern*

Ein Arbeitsspeicher eines handelsüblichen von-Neumann-Rechners, angefüllt mit Binärwörtern, repräsentiert für sich genommen noch keinerlei Information. Erst dadurch, dass man den Prozessor in Gang setzt, indem man den Befehlszähler auf eine bestimmte Speicheradresse setzt, werden bestimmte Speicherzelleninhalte als Befehle, andere als Daten interpretiert. Welche dabei wie interpretiert werden und ob das in der gewünschten Weise passiert, hängt einzig und allein davon ab, ob der Programmierer eine Befehlsfolge so aufgeschrieben hat, dass sie seine Absichten korrekt widerspiegelt.

Wir könnten hier drei Arten[5] (Typen) von Binärmustern unterscheiden:

- *Daten* (welcher Art auch immer),
- *Befehle* und
- *Speicheradressen*.

Den Bitmustern selbst kann man aber nicht ansehen, um was es sich jeweils handelt.

[5] Die Unterscheidung ist insofern „künstlich“, als auch Befehle und Speicheradressen vom Programm als Daten verarbeitet werden können. Dies liegt letztlich im grundlegenden Operationsprinzip der von-Neumann-Architektur begründet, Daten und Befehle in einem gemeinsamen Speicher unterzubringen.

Betrachten wir einmal Daten im engeren Sinne. Ein 32 Bit langes

Binärwort

$$01110100011110010111000001100101$$

kann z. B. als Integer-Zahl[6] mit dem Dezimalwert

$$1954115685,$$

als Gleitkommazahl[7]

$$7,90504 \cdot 10^{31}$$

oder als Folge von vier Zeichen[8]

```
'type'
```

interpretiert werden.

An diesem Beispiel wird deutlich, dass auf der Hardware-Ebene für die Ausführung von Programmen Typinformationen nicht erforderlich sind, sofern der Programmierer mit seinem Programm dafür gesorgt hat, dass die Daten repräsentierenden Bitmuster immer bestimmungsgemäß interpretiert werden. Die Hardware ist ohne Weiteres in der Lage, eine Zeichenfolge mit einer Integer-Zahl zu addieren oder zu multiplizieren.

Das wichtigste Ziel, das mit der Typisierung von Daten, d. h. mit der Bildung von *Datentypen*, verbunden ist, beseht nun genau darin, dass auf Bitmuster keine typfremden Operationen ausgeführt werden. Die Prozessor-Hardware ist dazu selbst nicht in der Lage. Das Problem muss also software-technisch gelöst werden.

[6] vorzeichenbehaftete ganze Zahl

[7] gemäß der Norm IEEE Standard for Binary Floating-Point Arithmetic for microprocessor systems (ANSI/IEEE Std 754-1985).

[8] codiert in ASCII oder ISO 8859-1

In einem Programm einen Ausdruck der Art `1954115685 + 'type'` zu verwenden, ist offensichtlich unsinnig, der Prozessor hätte „kein Problem“ damit.

3 *Was ist ein Datentyp?*

Im vorigen Abschnitt wurde erläutert, dass ein und dasselbe Bitmuster verschiedene Arten von Daten – wir könnten auch sagen: verschiedene *Typen* – repräsentieren kann. Wir sagen, „ein Datum besitzt einen bestimmten Typ“ oder „ist von einem Typ“. Insofern ist die Herkunft des Begriffes *Datentyp* offensichtlich. Andererseits bezeichnet der Begriff Datentyp mehr als eine bestimmte Art von Daten: Er stellt im Kontext der Programmiersprachen ein eigenständiges Konzept dar. Dies wird z. B. daran deutlich, dass auch Dinge, die gar keine Daten sind, einen Datentyp besitzen können, z. B. Variablen in Programmiersprachen wie C, Pascal oder Java. In diesen (und anderen Sprachen) ist bei der Deklaration einer Variablen immer die Angabe eines Datentyps erforderlich.

In der englisch-sprachigen Literatur sind die Begriffe *type*, *data type* oder *datatype* gebräuchlich. Mit der Zusammenschreibung soll betont werden, dass datatype mehr ist als type of data.

Obwohl der Begriff *Datentyp* der spezifischere ist, ist es durchaus üblich, den Begriff *Typ* als Synonym zu verwenden. Es wird noch dargelegt werden, dass auch Funktionen einen Typ besitzen. In diesem Zusammenhang von *Datentyp* zu sprechen, ist eher ungebräuchlich.

Die Begriffe *Typ* und *Datentyp* werden als Synoyme betrachtet.

Eine präzise Definition des Begriffs *Datentyp* ist schwierig und in der Literatur kaum zu finden. Es soll daher eine pragmatische

Annäherung vorgenommen werden, die von der Frage ausgeht: Wozu dienen Datentypen in Programmiersprachen?

In Abschnitt 2 wurden Probleme mit der Interpretation von Bitmustern erläutert. Die Menge der Bitmuster in einem Arbeitsspeicher kann als „ungetyptes Universum“ betrachtet werden. Man könnte auch sagen, es gibt nur einen einzigen Typ. Das führt dazu, dass die Hardware von der Interpretation der Bitmuster her unsinnige Operationen nicht verhindern kann. Was ist das Ergebnis der Addition zweier Speicheradressen? Die Verwendung der Summe für den Zugriff auf den Arbeitsspeicher kann – zu möglicherweise gefährlichen – Fehlfunktionen des Programms führen.

Ein Beispiel für ein ungetyptes Universum aus dem Bereich der Mathematik sind die Mengen der Mengenlehre. Funktionen können z. B. als (möglicherweise unendliche) Mengen von Paaren dargestellt werden. Was ist das Ergebnis der Vereinigung einer Menge, die Nachfolgerfunktion der natürlichen Zahlen repräsentiert, mit einer Menge die die Vorgängerfunktion repräsentiert? Auch dabei handelt es sich um eine offensichtlich unsinnige Operation.

Ein Beispiel für ein ungetyptes Universum aus dem Bereich der Programmiersprachen ist die Menge *symbolischen Ausdrücke*[9], wie sie für Lisp und Scheme definiert sind. Eine Form von S-Ausdrücken sind Listen. Die Liste

```
(f 4 5)
```

kann in einem Kontext einfach eine Liste bestehend aus einem Symbol und zwei Zahlen sein. In einem anderen Kontext bedeutet die Liste die Anwendung einer Funktion `f` auf die Argumente 4 und 5.

[9] auch S-Ausdrücke (bzw. engl. s-expressions) genannt. Vgl. z. B. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, part 1. *Commun. ACM 3*, 4 (1960), 184–195

Bei der Typisierung geht es also darum, Ordnung in das „Chaos“ der ungetypten Universen zu bringen mit dem Ziel, die Anwendung unsinniger, fehlerhafter Operationen zu vermeiden und damit die Qualität von Software zu verbessern, d. h. insbesondere den Nachweis der Korrektheit zu ermöglichen.

Typkompatibilität ist dabei der wichtigste Aspekt. Ein Operator kann auf seine Operanden nur dann fehlerfrei angewendet werden, wenn diese den erwarteten Typ haben. Die Anwendung einer Funktion ist nur dann sinnvoll möglich, wenn ihre Argumente typkompatibel zu den in der Funktionsdefinition spezifizierten Parametern sind. Für den Wert einer Funktionsanwendung muss wiederum Typkompatibiliät im Kontext seiner Verwendung gelten.

Auch die Verwendbarkeit von größeren Software-Bausteinen[10] kann auf die Typkompatibilität der an der Schnittstelle dieser Bausteine angebotenen Dienste zurückgeführt werden. Der Nutzer (Klient) einer Komponente muss wissen, wie die Dienste in Anspruch genommen werden können und welches Ergebnis ein Dienst liefert.

[10] Objekte, Komponenten, Web-Services …

Die erste Frage bezieht sich auf die *syntaktische* Kompatibilität, d. h. welche Typbezeichner und Funktionssignaturen stehen zur

Verfügung. Die zweite Frage betrifft die *semantische* Kompatibilität d. h. liefern die Dienste auch die erwarteten Resultate, verhalten sich die Funktionen gemäß ihrer Spezifikation. In vielen Texten, die sich mit der Typkompatibilität befassen, wird nur der syntaktische Aspekt beleuchtet. Für die Gewährleistung der Korrektheit von Software ist die Herstellung semantischer Kompatibilität unerlässlich.

A. Simons beschreibt, wie syntaktische Inkompatibilität Ursache des Verlusts des Mars Climate Orbiter war, während die Ariane-5-Katastrophe auf die Nutzung einer alten, semantisch inkompatiblen Software-Komponente zurückzuführen war; in

Simons, A. J. H. The theory of classification, part 1: Perspectives on type compatibility. *Journal of Object Technology 1*, 1 (May-June 2002), 55–61

Neben der Prüfung der Typkompatibilität erfüllen Datentypen noch weitere Aufgaben. Programmiersprachen stellen dem Programmierer Typkonstruktoren zur Verfügung, mit der die Menge der Datentypen erweitert werden kann. Dieses Mittel dient in erster Linie der *Datenabstraktion*. Damit ist die Zusammensetzung von elementaren Datentypen zu komplexeren Strukturen gemeint, die dann mit einem neuen Typbezeichner versehen werden können. Darauf werden wir in einem weiteren Arbeitspapier zurückkommen (vgl. Abschnitt 6).

Speicherplatzreservierung durch den Compiler ist ein möglicher Nutzen von statischen Typangaben bei der Deklaration von Variablen. Dieser Aspekt des Themas Datentypen wird in dieser Abhandlung nur am Rande eine Rolle spielen.

Im nächsten Abschnitt wird der Frage nachgegangen, wie die

4 *Prüfung der Typkompatibilität*

ermöglicht werden kann. Da die „Bewohner“ eines ungetypten Universums die dafür nötigen Informationen nicht bereitstellen können, muss einem Programm, dessen Bestandteile auf typgemäße Verwendung hin überprüft werden sollen, prüfbare Redundanz hinzugefügt werden. Dies kann durch den Programmierer explizit erfolgen: Bei der Deklaration von Variablen ist eine Typangabe erforderlich[11], z. B. für die Deklaration von Integer-Variablen:

`int i, j`	in Java, C
`var i, j : integer`	in Pascal, Modula, Oberon

[11] Wir sprechen von einem *expliziten* Typsystem.

Eine andere Möglichkeit besteht darin, dass der Typ von Programmobjekten[12] aus ihrer Notation oder ihrer Verwendung abgeleitet wird. Der Scheme-Ausdruck

```
(define pi 3.14159)
```

[12] Unter dem *Programmobjekt* sollen alle Bestandteile eines Programms verstanden werden, die einen Typ haben können, also z. B. Konstanten, Variable., Funktionen, Prozeduren.

erlaubt es dem Scheme-Interpreter aus der Schreibweise der Zahl zu ermitteln, dass sie vom Typ `Rational` ist und dem Variablenbezeichner `pi` ebenfalls diesen Typ zuzuordnen.[13] Fortan kann geprüft werden, ob eine Verwendung von `pi` typgemäß erfolgt.

[13] Wir sprechen von einem *impliziten* Typsystem.

Die Beschaffenheit des Typsystems einer Programmiersprache bestimmt nun nicht nur, wie die für die Prüfung der Typkompatibilität erforderliche Information „beschafft“ wird[14], sondern auch in welchem Ausmaß und zu welchem Zeitpunkt Typprüfungen tatsächlich stattfinden. Eine diesbezügliche Charakterisierung von Typsystemen erfolgt im nächsten Abschnitt.

[14] implizit oder explizit

Hinsichtlich des Zeitpunkts der Typprüfung unterscheidet man zwischen statischen und dynamischen Typsystemen. In einem *statischen Typsystem* erfolgt – vereinfacht gesagt[15] – die Typprüfung zur Übersetzungszeit, d. h. in Regel durch den Compiler oder Interpreter. Die Typprüfung erfolgt durch eine statische Analyse des Programms[16]. Es werden alle Programme zurückgewiesen, die einem bestimmten Satz von Typregeln nicht genügen.

Ein *dynamisches Typsystem* prüft zur Laufzeit unmittelbar vor Anwendung die Zulässigkeit einer Operation.

Ein weiteres

Unterscheidungsmerkmal für Typssysteme ist das Ausmaß, in dem Typregeln die Anwendbarkeit von Operationen tatsächlich einschränken. Man unterscheidet:

- Ein *strenges Typsystem* stellt jederzeit sicher, dass auf eine Variable bzw. Objekt nur der typgemäße Satz von Operationen angewendet werden kann.
- Ein *schwaches Typsystem* lässt auch „typfremde“ Operationen gelegentlich zu. [17]

Die Unterscheidung ist für konkrete Programmiersprachen nicht immer eindeutig. *Streng* und *schwach* sind keine diskreten Werte auf einer Koordinatenachse sondern markieren eher die Endpunkte einer Skala, zwischen denen ein „Kontinuum“ existiert. Das gilt im Grunde auch für die Unterscheidung zwischen statischen und dynamischen Typsystemen. Auch in den Laufzeitsystemen von statisch getypten Programmiersprachen werden häufig noch Typprüfungen vorgenommen.[18]

Es ist wichtig, Programmiersprachen mit einem strengen, aber dynamischen Typsystem nicht mit schwach getypten Sprachen zu verwechseln. Erstere verursachen Ausnahmen[19] bei fehlerhaften Operationen, letztere lassen unsinnige Operationen mit nicht vorhersagbaren Folgewirkungen zu.

Typsysteme von Programmiersprachen können also durch ein dreidimensionales, orthogonales Koordinatensystem mit den Skalen

- schwach – streng
- statisch – dynamisch
- explizit – implizit

klassifiziert werden. Einige bekannte Programmiersprachen lassen sich wie folgt einordnen[20]:

- C hat ein schwaches, statisches und explizites Typsystem.
- Java besitzt ein strenges, statisches und explizites Typsystem.
- Haskell und ML besitzen ein strenges, statisches und implizites Typsystem.
- Lisp, Scheme and Smalltalk haben ein strenges, dynamisches und implizites Typsystem.

[15] genauer gesagt: Typprüfungen erfolgen, *bevor* das Programm ausgeführt wird. In Java werden Typprüfungen auch durch den Class-Loader vorgenommen, also an bereits übersetzten Programmkomponenten.

[16] d. h. das Programm wird hierzu nicht ausgeführt

In der Industrie werden „statische“ Sprachen (wie Java, C++ und C#) viel häufiger benutzt als dynamische (wie CLOS, Python, Self, Perl, PHP oder Smalltalk).

Andererseits haben gerade Java und C# gewisse typische „dynamische“ Techniken adaptiert, wie Garbage-collection, oder Reflection.

[17] Typisches Beispiel: „Pointerarithmetik“ in C

[18] Z. B. kann die Einhaltung der Indexgrenzen eines statisch definierten Arrays nur zur Laufzeit eines Programms geprüft werden.

[19] engl. exceptions

[20] Dabei ist zu berücksichtigen, dass die genannten Kategorien „Kontinua“ darstellen. So werden z. B. auch in Sprachen mit dynamischer Typprüfung statische Prüfungen vorgenommen und umgekehrt.

5.1 Sind strenge Typsysteme besser als schwache?

Die Attribute *streng* und *schwach* stellen keine Qualitätsmerkmale eines Typsystems dar. Ob ein strenges oder schwaches Typsystem zu bevorzugen ist, hängt vielmehr davon ab, welchen Verwendungszwecken eine Programmiersprache dient. Die Programmiersprache C, die ein schwaches Typsystem besitzt, ist in den frühen 1970er Jahren für die Entwicklung des Betriebssystem Unix[21] geschaffen worden. Der originäre Verwendungszweck von C war also die Systemprogrammierung. Für einen Systemprogrammierer ist es nun völlig „normal“ und auch notwendig, arithmetische Operationen auf Speicheradressen[22] auszuführen. Die Berechnung von Speicheradressen ist z. B. bei der Implementierung von Array-Zugriffen notwendig. Dass das Typsystem von C derartige Operationen toleriert, ist also kein Manko.

Es ist indes durchaus fragwürdig, die Programmiersprache C für die gewöhnliche Anwendungsentwicklung einzusetzen und damit die Vorteile eines strengen Typsystems aufzugeben.

[21] Ritchie, D. M. The development of the c language. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages* (New York, NY, USA, 1993), ACM, pp. 201–208

[22] vorzeichenlose ganze Zahlen
In Pascal hingegen darf man auf Zeigervariablen (engl. pointers) keine arithmetischen Operationen ausführen, obwohl diese auch nur Speicheradressen in Form von ganzen Zahlen enthalten.

5.2 Sind statische Typsysteme besser als dynamische?

Statische Typsysteme haben den Vorteil, dass bestimmte Fehler bereits vor Ablauf des Programms gefunden werden. Allerdings weist jedes bekannte statische, explizite Typsystem auch viele eigentlich korrekte Programme zurück. Dies sei an einem einfachen Beispiel erläutert:

Nehmen wir an, die Werte zweier Integer-Variablen sollen vertauscht werden. Das Programmfragment[23]

[23] in einer an Pascal, Modula oder Oberon angelehnten Syntax

```
...
var i, j, h: integer;
      x, y: real;
...
h := i; i := j; j := h
```

bewerkstelligt das unter Verwendung einer „Hilfsvariablen“ `h`. Wollte man außerdem die Werte zweier Real-Variablen vertauschen, kann `h` nicht wieder verwendet werden, da die Zuweisung einer Real-Variablen an eine Integer-Variable

```
...
h := x; ...
```

eine Typverletzung darstellt und daher vom Compiler zurückgewiesen wird. Nähme der Compiler die Typverletzung hin, würde die Anweisungsfolge

```
h := x; x := y; y := h
```

hingegen korrekt ausgeführt[24].

[24] vorausgesetzt Integer- und Real-Variablen belegen gleich viel Speicherplatz

Es ist in einer „klassischen“, statisch getypten Programmiersprache auch nicht möglich, eine Prozedur `swap` zu schreiben, die die

Werte zweier Variablen egal welchen Typs vertauscht[25], da in der Prozedurdeklaration der Typ der Parameter angegeben werden muss. Man wäre also gezwungen für jeden Typ zu vertauschender Variablen eine eigene „swap"-Prozedur zu schreiben. Ein ähnliches Problem tritt auf, wenn man z. B. die Anzahl der Elemente einer Liste bestimmen will. Der Algorithmus dafür ist unabhängig vom Typ der Listenelemente, gleichwohl ist es nicht möglich eine Prozedur für Listen beliebigen Elementtyps zu schreiben.

[25] z. B. durch den Aufruf `swap(x, y)`

Das oben verwendete Attribut „klassisch" deutet darauf hin, dass es auch statisch getypte Programmiersprachen gibt, die derlei Beschränkungen der Ausdrucksmächtigkeit nicht aufweisen. Sprachen, wie z. B. Pascal, bezeichnet man als *monomorph*, womit ausgedrückt werden soll, dass jedes Programmobjekt immer nur genau einen Typ besitzen kann.

Monomorphie

In *polymorphen* Programmiersprachen kann ein Programmobjekt mehr als einen Typ besitzen. Eine Prozedur z. B. ist polymorph, wenn sie mit Argumenten unterschiedlichen Typs aufgerufen werden kann. Häufig wird dabei gefordert, dass diese Typen „etwas gemeinsam haben". Diese Gemeinsamkeit könnte z. B. darin bestehen, dass sie *Untertyp* eines gemeinsamen *Obertyps* sind. Ohne auf diese Begriffe an dieser Stelle[26] genau einzugehen, soll ihr Nutzen am Beispiel der oben erwähnten Prozedur `swap` erläutert werden: Nähmen wir an, die Typen `integer` und `real` wären Untertyp eines Typs `number`, könnte der Kopf einer Prozedur zum Vertauschen von Variablen wie folgt angegeben werden:

Polymorphie

Turbak und Gifford verwenden anstelle der Typsystem-Kategorien *streng* und *schwach* die Kategorien *simple* und *expressive*: . Sprachen wie Pascal oder C fallen danach in die erste, Sprachen mit Polymorphie in die zweite Kategorie.

Turbak, F., and Gifford, D. *Design Concepts in Programming Languages*. The MIT Press, 2008

[26] Das Thema *Polymorphie* wird in einem weiteren Arbeitspapier behandelt (vgl. Abschnitt 6).

```
procedure swapNumbers(var v1, v2: number)
```

Diese Prozedur könnte dann sowohl mit zwei Integer- als auch mit zwei Real-Variablen aufgerufen werden. Die Typkompatibilität ist gegeben, da an der Stelle, wo eine `number`-Variable erwartet wird, auch ein Exemplar eines Untertyps von `number` eingesetzt werden darf.

Ein Mindestmaß an Polymorphie gibt es auch in Programmiersprachen mit einem einfachen, statischen Typsystem. Andernfalls wäre Ausdrücke der Art `2.0 * 4` oder `5 + 3.1` nicht zulässig, da der Multiplikations- bzw. Additionsoperator entweder für Integer- oder Real-Operanden definiert wäre.

Kaum eine andere Frage rund um die Konzepte von Programmiersprachen wird seit Jahrzehnten so heftig diskutiert, wie die nach den Vor- und Nachteilen von statischen und dynamischen Typsystemen. Und es ist auch nicht zu erwarten, dass diese Frage in absehbarer Zeit abschließend beantwortet werden wird. Auch an dieser Stelle soll kein Versuch in dieser Richtung unternommen werden. Zum Schluss dieses Abschnitts sollen nur noch ein paar markante Unterschiede erläutert und einige häufig anzutreffende Argumente wieder gegeben werden.

Aus den voran gegangenen Absätzen kann man heraus lesen:

- In einem statischen Typsystem besitzen die Variablen einen Typ.
- In einem dynamischen Typsystem besitzen die Werte einen Typ.

Diese Aussagen sind unstrittig.

Die Verfechter statischer Typsysteme führen für diese u. a. folgende Argumente ins Feld:

- Sie helfen, Entwicklungskosten zu reduzieren.
- Sie beschleunigen das Codieren, da die im Programmtext vorhandenen Typinformationen den Entwicklungswerkzeugen umfassendere Unterstützung des Programmierers ermöglichen.
- Da viele potentielle Fehler bereits durch den Compiler entdeckt werden, muss man weniger testen.
- Es treten weniger Laufzeitfehler auf.
- Die übersetzten Programme sind schneller.

Befürworter dynamischer Typsysteme halten dem entgegen:

- Der für Programmierer möglicherweise einfachere Einstieg in statische Programmiersprachen wird mit einem Mangel an Ausdrucksmächtigkeit erkauft. Dynamische Sprachen eröffnen dem Programmierer mehr Möglichkeiten und machen ihn dadurch produktiver.
- Entwicklungssysteme für dynamische Programmiersprachen ermöglichen es, in hohem Maße inkrementell zu arbeiten, d. h. kleine Programmänderungen können, ohne das Gesamtsystem neu zu übersetzen, sofort überprüft werden. Der Unterschied zwischen Übersetzungs- und Laufzeit ist praktisch aufgehoben.
- Da Compiler nur statische Typfehler finden können, die gravierendsten Fehler aber semantischer Natur sind, ist ausgiebiges, systematisches Testen auch bei der Verwendung von statischen Programmiersprachen unbedingt erforderlich. In einer dynamischen Umgebung werden die statischen Typfehler durch diese Tests mit entdeckt.
- Durch das Fehlen praktikabler Methoden, korrekte Programme zu entwickeln, kommt es auch bei Verwendung statischer Programmiersprachen zu Laufzeitfehlern. Möglicherweise treten sie seltener auf, aber dafür gibt es kaum Belege. Die erweiterten Ausdrucksmöglichkeiten in dynamischen Sprachen erlegen dem Programmierer aber auch eine größere Verantwortung auf.
- Grundsätzlich gilt: Typprüfungen, die zur Laufzeit ausgeführt werden müssen, kosten Rechenzeit, die bei statischer Typprüfung nicht anfällt. Allerdings fällt dies bei modernen optimierenden Just-in-Time-Compilern kaum mehr ins Gewicht. Außerdem hängt die Geschwindigkeit der meisten Anwendungen eher von Netzwerk und Datenbank ab.

5.3 Sind implizite Typsysteme besser als explizite?

Implizite Typsysteme befreien den Programmierer davon, redundante Typinformationen dem Programm hinzuzufügen. Implizite Typsysteme sind meist auch dynamisch, wie z. B. in Scheme oder Lisp. Es gibt aber auch Programmiersprachen, die ein statisches, implizites Typsystem besitzen und damit zusätzlich die Vorteile statischer Typprüfungen aufweisen.

Als Argument für explizite Typsysteme wird häufig angeführt, dass die Typangaben im Programmtext diesen auch gleichzeitig dokumentieren, was der Lesbarkeit der Programme zugute kommt. Kritiker dieser These wenden ein, dass Typangaben in Programmen diese leichter durch den Compiler verarbeitbar machen, die bessere Lesbarkeit durch den Menschen hingegen zweifelhaft ist. Gelegentlich muss der Programmierer für das Schreiben expliziter Typinformationen mehr Zeit aufwenden als für die Programmlogik.

Allerdings lassen sich nicht alle Typinformationen aus dem Programmtext ableiten, so dass Sprachen mit impliziten, statischen Typsystemen, wie z. B. Haskell oder ML, einen Kompromiss eingehen müssen. Kann der Typ eines Programmobjekts statisch nicht ermittelt werden, muss der Programmierer die fehlende Typinformation hinzufügen.

6 Wie geht es weiter?

Nach der in diesem Arbeitspapier erfolgten ersten Annäherung an einige grundlegende Begriffe der Typtheorie sollen in weiteren Arbeitspapieren folgende Themen behandelt werden:

a) *Sichtweisen auf Typen* – hier werden denotationale, abstraktionsorientierte und konstruktive Sichtweisen auf Typen vorgestellt und es wird gezeigt, wie sich diese in klassischen Typsystemen verschiedener Programmiersprachen wieder finden.
b) *Datentypen als Abstraktionsmittel* werden ausführlicher betrachtet unter besonderer Berücksichtigung algebraischer Spezifikationstechniken.
c) *Erweiterung des Typbegriffs durch die Objektorientierten Programmiersprachen* – hierbei geht es u. a. um
   a Objekte, Klassen, Objekttypen
   b Subklassen, Subtypen
   c Kovarianz, Kontravarianz
   d Polymorphie

Literatur

[1] Bruce, K. B. *Foundations of Object-Oriented Languages*. The MIT Press, 2002.

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

[3] Friedman, D. P., and Wand, M. *Essentials of Programming Languages*, third ed. The MIT Press, 2008.

[4] McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, part 1. *Commun. ACM 3*, 4 (1960), 184–195.

[5] Meek, B. L. A taxonomy of datatypes. *SIGPLAN Notices 29*, 9 (1994), 159–167.

[6] Ritchie, D. M. The development of the c language. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages* (New York, NY, USA, 1993), ACM, pp. 201–208.

[7] Simons, A. J. H. The theory of classification, part 1: Perspectives on type compatibility. *Journal of Object Technology 1*, 1 (May-June 2002), 55–61.

[8] Turbak, F., and Gifford, D. *Design Concepts in Programming Languages*. The MIT Press, 2008.