Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Typisierung in Programmiersprachen

vorgelegt von

Darwin Willers

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik

Studiengang:	Software-System-Entwicklung
Matrikelnummer:	7199817
Betreuer:	Hermann Lenhart

Hamburg, 15.03.2020

Inhaltsverzeichnis

1 Einleitung

Dieser Bericht soll dem Leser ein grundlegendes Verständnis dafür geben, was Typisierung in Programmiersprachen ist und ihren Nutzen erläutern. Es werden keine Eigenschaften im Detail diskutiert, noch werden Thesen aufgestellt oder bewiesen. Der Bericht gibt lediglich einen Überblick über die gängigen Definitionen und Ansichten, welche in Fachliteratur, aber auch Praxis näheren Texten immer wieder ähnlich oder identisch erläutert werden.

Es wird vom Leser ein Grundverständnis dafür erwartet, wie Computerprogramme entwickelt werden und funktionieren. Fachbegriffe wie Compiler, Übersetzungs- oder Laufzeit werden häufig genutzt und nicht weiter erläutert. Auch die gängigen primitiven Datentypen in Programmiersprachen sollten dem Leser bekannt sein.

Aussagen zu Verhalten von Programmiersprachen, für die keine Quelle angegeben sind, gehen auf eigene Tests zurück und sind mittels des Codes in den Listings leicht selbst zu überprüfen.

2 Definition

Eine oft zitierte, aber komplexe Definition von Prof. Benjamin C. Pierce ist die folgende.

"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."[Pie02]

Professor Pierce definiert Typisierung also als ein System zum Beweisen der Abwesenheit bzw. dem nicht Existieren von Programm-Verhalten. Daraus schließt sich, dass das Ziel oder der Nutzen von Typisierung darin liegt, ungewolltes Verhalten zu verhindern.

Eine weniger tief greifende, aber dafür simplere Definition, welche auch inhaltlich Professor Pierce zitiert, entspringt der englischen Wikipedia Seite für Typisierung im englischen Type Systems"genannt.

"In programming languages, a type system is a logical system comprising a set of rules that assigns a property called a type to the various constructs of a computer program, such as variables, expressions, functions or modules." [Wik]

In diesem Zitat wird sehr schön hervorgehoben, dass nicht nur Variablen Typen haben, sondern auch Ausdrücke und Funktionen und dass diese durch eine Menge an Regeln zugewiesen werden.

Mittels Typisierung müssen Programmierer sich weniger Gedanken machen, wie verscheiden Datentypen im Speicher abgelegt werden und zu interpretieren sind. Es ist nicht nötig, das entsprechende Bitmuster im Code anzugeben, wenn man einer Variable einen Wert zuweisen möchte. Stattdessen werden durch Abstraktionsebenen die uns geläufigen Formen der Darstellung und Nutzung im Code ermöglicht.

Eine komplexe Typisierung erlaubt den Programmierer durch mehr Abstraktion, das gewünschte Programm-Verhalten mit weniger Code zu erzielen. Wenn beispielsweise auch Methoden einen Typ haben, wie es in immer mehr Sprachen der Fall ist, wird es möglich, eben diese in variablen abzulegen oder als Parameter an andere Methoden zu übergeben. Dies macht Methoden wie **foreach** oder **map** möglich.

3 Arten der Typisierung

Es gibt zwei Arten von Typisierung, welche beide sehr weit in Programmiersprachen verbreitet sind, da sie beide gewisse Vor- und Nachteile haben. Einige Sprachen, z. B. Python, welche eine dynamisch typisierte Sprache ist, hat seit Version 3.5 auch eingeschränkte Unterstützung für statische Typisierung. Man spricht dann auch von optionaler Typisierung. [Fou]

Diese ist kein Einzelfall und ist auch in die andere Richtung zu beobachten. Beispielsweise hat C# eine Sprache mit statischer Typisierung seit Version 4.0 das **dynamic** Keyword, welches es erlaubt, den Typ erst zur Laufzeit zu bestimmen und auch zu ändern. [Mic]

Da man Sprachen möglichst flexibel und leicht nutzbar machen möchte und beide Arten der Typisierung klare Vorteile bieten, gehen immer mehr Sprachen dazu über, auch die andere Art in zumindest eingeschränkter Weise zu unterstützen.

3.1 Statische Typisierung

Bei der statischen Typisierung, wie sie Sprachen wie Java und C# nutzen, steht der Typ zur Übersetzungszeit bereits immer fest und kann nicht mehr geändert werden. Dies wird in der Regel dadurch erreicht, dass der Programmierer bereits zur Entwicklungszeit im Source-Code den Typ für Variablen, Parameter, Rückgabewerte etc. angibt.

```
1 int number1 = 4;
2 var number2 = 12;
3 var number3 = 4.25f;
4
5 int number4 = 3.142;
```

Listing 3.1: Statische Typisierung in C#

In der Regel wird dies explizit getan, aber C#, Java und auch andere Sprachen haben bereits Keywords eingeführt, um den Typ durch den Compiler anhand des Kontexts ermitteln zu lassen. Dies ist in Listing 3.1 zu sehen. Während in Zeile eins noch der Typ der Variable explizit angegeben wird, wird dieser in Zeile zwei und drei später zur Übersetzungszeit durch den Compiler eingefügt, da dieser sich aus den Ausdrücken auf der rechten Seite der Zuweisungen schließt. In Zeile zwei wäre dies erneut **Integer** und in Zeile drei **Float**. Dies ist als Erleichterung für Programmierer gedacht, ändert aber

nichts daran, dass der Typ zur Übersetzungszeit bereits fest steht und unveränderbar ist. Es handelt sich hierbei also nicht um dynamische Typisierung.

Der große Vorteil von statisch typisierten Sprachen ist, dass der falsche Umgang mit Typen zur Laufzeit nicht möglich ist, da der Typ bereits zur Übersetzungszeit bekannt ist wird der Compiler, falls er einen Verstoß findet, diesen melden und kein fertig kompiliertes Programm bereitstellen. Dadurch haben Programme, welche mit statischen Programmiersprachen entwickelt werden, eine höhere Robustheit, da viele mögliche Fehlerquellen bereits durch den Compiler ausgeschlossen werden. Beispielsweise würde Zeile fünf in Listing 3.1 so einen Fehler mit folgender Fehlerbeschreibung produzieren. *error CS0266: Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)* Die höhere Robustheit bedeutet aber auch eine länger Feedback schleife, da Änderung immer erst neu kompiliert werden müssen, bevor diese getestet werden können.

3.2 Dynamische Typisierung

Im Gegensatz zu statischen wird bei der dynamischen Typisierung der Typ erst zur Laufzeit ermittelt und ist auch veränderbar. Das sorgt dafür, dass dynamische Sprachen oft flexibler sind und weniger Code geschrieben werden muss. Zum Beispiel kann in Javascript einfach ein Objekt erstellt werden, das eine dynamische Anzahl von Variablen mit variablen Typen hat wie in Listing 3.2 Zeile fünf.

```
1 function printName(p) {
2   console.log(p.name)
3 }
4
5 printName({name: "Test", age: 21})
```

Listing 3.2: Print Name in C#

In statischen Programmiersprachen müsste man dafür erst eine Klasse oder andere Struktur erstellen, welche genau diese Form hat. Das kann dazu führen, dass in statischen Sprachen viele Klassen erstellt und gepflegt werden müssen, was in dynamischen ganz vermieden werden kann, wenn man dies möchte. Diesen Mehraufwand kann man gut in Listing 3.3 sehen.

```
1 class Person {
2     public string name;
3     public int age;
4
5     public Person(string name, int age) {
6         this.name = name;
```

```
7           this.age = age;
8       }
9   }
10
11  public void printName(Person p) {
12      Console.WriteLine(p.name);
13  }
14
15  printName(new Person("Test", 21));
```

Listing 3.3: Print Name in C#

Außerdem könnte die Javascriptfunktion in Listing 3.2 nicht nur den Namen einer *Person* ausgeben, wie es in Listing 3.3 der Fall ist, sondern den Namen von jedem Objekt. Vorausgesetzt dieses besitzt ein Feld mit dem Namen *name* und und der Typ darf *console.log(...)* Übergeben werden. Diese Voraussetzung sind im Listing 3.3 aufgrund der statischen Typisierung immer erfüllt. Zwar könnte auch diese Funktion mittels Abstraktionstechniken wie Vererbung oder Interfaces dynamischer gemacht werden, allerdings bedeutet dies wieder deutlich mehr Code und eine gute Planung. Dieses Beispiel veranschaulicht schön die Flexibilität einer dynamischen Sprache gegenüber der Robustheit einer statischen Sprache.

3.3 Optionale Typisierung

Bei dynamisch typisierten Sprachen, welche eine statische Annotation des Typs unterstützen, spricht man von optionaler Typisierung. Die statische Typisierung hat keinerlei Auswirkung auf die Software, da es keine Typüberprüfung zur Übersetzungszeit gibt. Oft gibt es nicht einmal eine Übersetzungszeit, sondern der Code wird direkt von einem Interpreter interpretiert. Die statische Annotation im Code dient als Hilfe für den Programmierer oder kann mittels Tools genutzt werden, um mögliche Typfehler zu finden oder Fehlermeldungen sprechender zu machen. Es ist der Versuch, die Robustheit einer statisch typisierten Sprache für dynamisch typisierte Sprachen nutzbar zu machen. Oft ist dies besonders hilfreich, wenn Softwareprojekte sehr groß und/oder komplex werden. [JL11]

4 Starke und schwache Typisierung

Leider ist starke und schwache Typisierung nicht einheitlich definiert. In den meisten Fällen bezieht man sich aber auf Typsicherheit oder Speichersicherheit. Einige Leute verstehen starke und schwache Typisierung leider als Synonyme für statische und dynamische Typisierung. Weswegen es ratsam ist, sich vorher auf eine gemeinsame Definition zu verständigen, um Missverständnisse zu vermeiden. Im Folgenden werden die häufig in diesem Zusammenhang genutzten Konzepte der Typsicherheit und der Speichersicherheit erläutert.

4.1 Typsicherheit

Oft wird Typsicherheit den statisch typisierten Programmiersprachen nachgesagt und den dynamisch typisierten abgesprochen. In der Regel argumentieren diese Leute damit, dass zur Laufzeit Typen veränderbar oder nicht veränderbar sind. Allerdings hat dies gar nichts mit Typsicherheit zu tun. Typsicherheit kann sowohl in statischen als auch in dynamisch typisierten Sprachen existieren.

Vielmehr meint Typsicherheit, dass Variablen oder Ausdrücke nur in ihrem aktuellen Typ interpretiert werden. Sollen diese anderes interpretiert werden, muss der Programmierer die explizit angeben. Dies nennt sich dann auch explizite Typumwandlung. Dabei widerspricht eine Typumwandlung nicht der statischen Typisierung. Es geht hier nicht darum, den Typ einer Variable zu ändern, sondern deren Inhalt in einer anderen Form zu interpretieren.

```
1 x = 1 + "2"
```

Listing 4.1: Implizite Typumwandlung

Das in Listing 4.1 zu sehende Beispiel wird häufig genutzt um Typsicherheit zu erläutern. Während erfahrene Programmierer diese Verhalten vielleicht bereits kennen, würden alle anderen verständlicher weise erwarten das **x = 3** gilt. Da aber jeder Integer als String dargestellt werden kann, aber nur eine kleine Menge an möglichen Strings auch als Integer dargestellt werden können, verhalten sich Sprachen, welche diesen Ausdruck zulassen, so das **x = "12"** gelten würde. Was hier passiert, ist das der Integer **1** implizit in den String **"1"** gewandelt wird. Beispielsweise wäre dies so für C#, Java oder auch Javascript. Wobei es sich bei den beiden erst genannten um statisch typisierte Sprachen handelt.

Nur weil etwas möglich ist, ist dies noch lange nicht im Sinne des Programmierers. Um ungewollten Interpretation wie diese zu vermeiden, gibt es Typsicherheit. In typsicheren Sprachen ist das Umwandeln von Typen nur explizit erlaubt. Das bedeutet, dass eine typsichere Sprache an dieser Stelle einen Fehler werfen würde. Abhängig davon, welche Art von Typisierung genutzt wird, würde dies bereits zur Übersetzungszeit oder erst zur Laufzeit passieren. Fehler zu werfen, ist entgegen der allgemeinen Definition des Wortes Fehler eine durchaus erwünschte Eigenschaft von Programmiersprachen. Denn 'Exceptions' vermeiden ungewolltes oder unvorhersehbares Verhalten. Das der Nutzer davon möglichst wenig merkt und das Programm nicht abstürzt, ist die Aufgabe des Programmierers. Python wäre beispielsweise eine dynamisch typisierte Sprache, welche für den Code in Listing 4.1 einen Fehler werfen würde.

```
1 x = str(1) + "2"
2 y = 1 + int("2")
3 z = 1 + int("abc")
```

Listing 4.2: Explizite Typumwandlung

Im Listing 4.2 sieht man eine mögliche Lösung für die typsichere Sprache Python. In Zeile eins würde das bereits bekannte Ergebnis von **x = "12"** stehen. In Zeile zwei wäre die wohl logischere Lösung mit **y = 3**. Zeile drei wiederum würde erneut einen Fehler werfen, da der String **'abc'** nicht in einen Integer umgewandelt werden kann.

4.2 Speichersicherheit

[Hic14]

Speichersicherheit ist die Abwesenheit von Möglichkeiten in einer Sprache, den Speicher falsch zu nutzen. Beispiele dafür wären das Zugreifen auf nicht mehr reservierten oder bereits wieder freigegebenen Speicher und Buffer-Overflows. Viele dieser Probleme können in Programmiersprachen schon durch Typisierung und die Abwesenheit von Zeigern umgangen werden. Beispielsweise wird dort der Speicher durch die Abstraktionsebene reserviert und wieder frei gegeben. NullPointerExceptions werden abgefangen und es werden Fehler geworfen. Ist eine Sprache 'memorysafe' also speichersicher, so ist die Integrität des Speichers sicher gestellt. Viele Schwachstellen in Software gehen darauf zurück, dass diese Sprachen nicht 'memorysafe' waren und so Zustände des Programms auf ungewollte Weise erreicht werden konnten. Oft waren dies sogar Zustände, welche man nur so erreichen konnte. Eine Sprache, welche keine dieser Sicherheiten bietet, ist **C**. Neuere Sprachen sind in der Regel 'memorysafe', um viele Exploitmöglichkeiten zu vermeiden.

5 Zusammenfassung

Typisierung ist eine Möglichkeit, um mittels einer oder mehrerer Abstraktionsebenen das Schreiben von Software zu vereinfachen. Jede höhere Programmiersprache nutzt die Typisierung mehr oder weniger stark. Wobei moderner Sprachen dazu neigen, eine starke Abstraktion zu nutzen. Es gibt dabei zwei primäre Arten der Umsetzung.

Die statische Typisierung erlaubt es, einen Programmierfehler frühzeitig zu erkenne und erhöht so die Robustheit des Codes. Allerdings würde der Quellcode auch oft größer und man ist weniger flexibel für Änderungen.

Die dynamische Typisierung besitzt eine sehr kurze Feedbackschleife und es muss weniger Code geschrieben werden. Allerdings werden viele Fehler erst zur Laufzeit entdeckt, weshalb mehr getestet werden muss.

Die optionale Typisierung ist der Versuch, die Vorteile beider Arten zu verbinden. Hierbei handelt es sich um dynamisch typisierte Sprachen, welche eine optionale statische Annotation erlauben.

Des Weiteren sind auch Typsicherheit und Speichersicherheit wichtige Aspekte eine Programmiersprache und hängen eng mit der Typisierung zusammen. Sie sorgen zusätzlich für Sicherheit und Robustheit von Software.

Literaturverzeichnis

[Fou] Python Software Foundation. typing — support for type hints. https://docs.python.org/3/library/typing.html. Zugriff am 13.03.2021.

[Hic14] Michael Hicks. What is memory safety? http://www.pl-enthusiast.net/2014/07/21/memory-safety/, 2014. Accessed: 05.12.2020.

[JL11] David J. Greaves Jukka Lehtosalo. Language with a pluggable type system and optional runtime monitoring of type errors. *University of Cambridge Computer Laboratory*, 2011.

[Mic] Microsoft. Verwenden des typs „dynamic“. https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/types/using-type-dynamic. Zugriff am 13.03.2021.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Wik] Wikipedia. Type systems. https://en.wikipedia.org/wiki/Type_system. Zugriff am 12.03.2021.