F. Steimann
Mitarbeit: D. Keller

Objektorientierte Programmierung

mathematik und informatik

FernUniversität in Hagen

Das Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere das Recht der Vervielfältigung und Verbreitung sowie der Übersetzung und des Nachdrucks, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder ein anderes Verfahren) ohne schriftliche Genehmigung der FernUniversität reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

3 Typen in der objektorientierten Programmierung

> The purpose of a type system is to prevent the occurrence of execution errors during the running of a program. The accuracy of this informal statement depends on the rather subtle issue of what constitutes an execution error. Even when that is settled, the type soundness of a programming language (the absence of certain execution errors in all program runs) is a non-trivial property. A fair amount of careful analysis is required to avoid false and embarrassing claims of type soundness; as a consequence, the classification, description, and study of type systems has emerged as a formal discipline.
>
> Luca Cardelli, in: *CRC Handbook of Computer Science and Engineering* Chapter 103 (1996).

Im Gegensatz zu SMALLTALK sind die meisten objektorientierten Programmiersprachen typisiert, was soviel heißt wie daß Programmelementen bei ihrer Deklaration (s. Abschnitt 3.2) Typen zugeordnet werden. Dabei schränkt ein Typ die Menge der Objekte, für die ein Programmelement stehen kann, und die Menge der Dinge, die damit gemacht werden können, ein. Meistens sind die Regeln zur Verwendung von Typen fester Bestandteil der Sprache — wenn Sie eine solche Sprache neu lernen, dann würden Sie gar nicht auf die Idee kommen, Typsystem und übrige Sprachdefinition voneinander getrennt zu betrachten. Dennoch sind Typen für das Funktionieren eines Programms prinzipiell verzichtbar[55] und es lohnt sich durchaus, das Typsystem einer Sprache von ihrem Rest zu lösen, beispielsweise weil man es austauschen oder verbessern will. Dies um so mehr, als heute gängige Typsysteme entweder ziemlich schwach oder ziemlich kompliziert sind.

So führt diese Kurseinheit Typsysteme am Beispiel von STRONGTALK, einer SMALLTALK-Erweiterung um ein *optionales Typsystem*, ein. Sie geht dabei langsam und inkrementell vor. Wer das zu öde erscheint, die sei gewarnt: Es wird noch kompliziert genug und nicht jede Leserin wird alles, was sie in diesem Kurs über Typsysteme liest, auf Anhieb verstehen. Auch wäre die Alternative, diese Kurseinheit am Beispiel einer bekannteren Sprache mit verpflichtendem Typsystem hochzuziehen, stets mit dem Nachteil belastet, dieses konkrete Typsystem als quasi gottgegeben darstellen zu müssen — wenn Sie dann später eine andere Sprache kennenlernen, hätten Sie höchstwahrscheinlich Schwierigkeiten, das Gelernte abzustreifen und sich mit den neuen Verhältnissen zurechtzufinden. Ziel

---

[55] Wenn man auf Möglichkeiten wie das Überladen von Methoden verzichten kann; Laufzeittypinformation, wie man sie z. B. für das *dynamische Binden* oder für die *Garbage collection* benötigt, kann durch Laufzeitklasseninformation (was nicht dasselbe ist!) ersetzt werden; s. Abschnitt 3.11.3.

dieser Kurseinheit ist aber, daß Sie Typsysteme als das verstehen, was sie sind: eine Möglichkeit zur Spezifikation redundanter Information, die die Qualität von Programmen erhöhen soll.

3.1 Hintergrund

Sie kennen vielleicht aus anderen Programmiersprachen, daß Variablen und anderen Programmelementen bei ihrer Deklaration (Abschnitt 3.2) ein Typ zugeordnet wird. Dieser Typ schränkt die möglichen Werte der *deklarierten Elemente* ein. So lassen sich beispielsweise in einer Variable vom Typ `Boolean` nur Wahrheitswerte, in einer vom Typ `String` nur Zeichenketten speichern.

Typ

Typ ist ein primitiver Begriff, vergleichbar etwa mit dem Begriff der Menge in der Mengentheorie. Ein Typ hat eine *Intension* und eine *Extension*, wobei erstere der Definition des Typs entspricht, letztere seinem Wertebereich, also der Menge der Elemente (Objekte), die zu dem Typ gehören (man sagt auch, „die den Typ haben“ oder „die von dem Typ sind“). Häufig hat ein Typ auch einen Namen, den Typbezeichner. Typen sind die Grundlage von Typsystemen.

Zusammenhang von Typ und Klasse

Ihnen fällt wahrscheinlich sofort die Ähnlichkeit zum Konstrukt der Klasse, wie es in der letzten Kurseinheit eingeführt wurde, auf. Tatsächlich gibt es hier auch einen gewissen Zusammenhang. Um Sie aber nicht gleich in für diese Kurseinheit eher schädliche Denkmuster verfallen zu lassen, soll dieser Zusammenhang zunächst zurückgestellt werden. Eine Aufklärung erfolgt dann in Abschnitt 3.11.

Typsystem

Ein **Typsystem** umfaßt Typausdrücke, Wertausdrücke, Regeln, die Wertausdrücken Typen zuordnen, und Regeln, die von Wertausdrücken einzuhalten sind (zusammen die **Typregeln**). Wertausdrücke (bzw. schlicht Ausdrücke, wenn es nicht um die Abgrenzung von Typausdrücken geht) kennen Sie schon: In SMALLTALK sind es die in Abschnitt 1.4 aufgeführten. Mit den anderen Konzepten werden Sie in den nachfolgenden Abschnitten vertraut gemacht, allerdings in weniger formaler Form, als Sie das nach dieser Definition vielleicht befürchten.

Gründe für die Typisierung

Warum aber typisiert man Variablen und andere Programmelemente? Dafür gibt es mindestens vier gute Gründe:

1. Typisierung regelt das Speicher-Layout.
2. Typisierung erlaubt die effizientere Ausführung eines Programms.
3. Typisierung erhöht die Lesbarkeit eines Programms.
4. Typisierung ermöglicht das automatische Finden von logischen Fehlern in einem Programm.

Speicher-Layout

Ad 1.: Der Compiler kann anhand des Typs einer Variable bestimmen, wie viel Speicherplatz er für die Aufnahme eines Wertes reservieren muß. Dies ist jedoch naturgemäß nur für Variablen mit Wertsemantik relevant und daher für die objektorientierte Programmierung, insbesondere für Sprachen wie SMALLTALK (in denen Referenzsemantik vorherrscht), von untergeordneter Bedeutung.

effizientere Ausführung

Ad 2.: Wenn man weiß, daß die Werte einer Variable immer vom selben Typ sind, also alle demselben Wertebereich entstammen, dann lassen sich bestimmte Optimierungen durchführen. Wenn man z. B. aufgrund der Deklaration einer Variable `x` für gegeben annehmen kann, daß `x` nur ganze Zahlen enthält, dann kann der Compiler für die Übersetzung von `x := x + 1` die Ganzzahladdition, ja sogar die Inkrement-Anweisung des Prozessors verwenden. Kennt der Compiler den Typ von `x` hingegen nicht, dann muß das Programm vor der Ausführung der Addition erst prüfen, von welchem Typ der Wert von `x` ist — handelt es sich um eine Fließkommazahl, so muß es zu der entsprechenden Operation verzweigen, handelt es sich womöglich um gar keine Zahl, dann muß es einen Laufzeitfehler signalisieren oder sich etwas anderes einfallen lassen. Dem kann man entgegenhalten, daß im Falle der objektorientierten Programmierung selbst bei einer Typisierung aller Variablen gelegentlich noch Laufzeittests durchgeführt (oder andernfalls schwere Programmfehler in Kauf genommen) werden müssen, und daß sich die zur Optimierung benötigte Information auch anders als über explizite Typisierung von Variablen (nämlich z. B. über die sog. *Typinferenz*, also die Ausnutzung impliziter Typinformation) gewinnen läßt.

erhöhte Lesbarkeit

Ad 3.: In der Vergangenheit hatten Variablen eher kurze, wenig selbsterklärende Namen. Es ist dann sinnvoll, wenigstens an der Stelle der ersten Erwähnung der Variablen (in der Regel deren *Deklaration*) einen Hinweis darauf zu haben, wofür (für welche Menge von Objekten) die Variable steht. Dies kann über einen Kommentar erfolgen, aber auch durch die Assoziation mit einem Typen, die aussagt, welcher Art die Werte der Variable sein müssen. Doch nicht nur Variablen-, auch Methodennamen können für sich genommen wenig aussagekräftig sein und durch die Verknüpfung mit Typen aussagekräftiger gemacht werden: Eine Deklaration der Methode `next` etwa, die als Typ des Ein- und Ausgabeparameters `ListElement` deklariert, legt nahe, daß sie das in einer Liste auf den Eingabeparameter folgende Element zurückliefert. Ohne die Angabe der Parametertypen müßte man als Nutzer der Funktion, der ihre Implementation nicht kennt, schon über ihren Zweck spekulieren. Dem mag man freilich entgegenhalten, daß man statt dessen ja auch selbsterklärende Namen für Variablen und Methoden vergeben könnte (mehr dazu in Abschnitt 7.1).

automatisches Finden von logischen Fehlern

Es bleibt aber in jedem Fall Punkt 4, das Aufdecken von logischen Fehlern in einem Programm. Ohne externes Wissen, was ein Programm tun soll, verlangt das Finden von Fehlern jedoch ein gewisses Maß an Redundanz, also die mehrfache Lieferung gleicher Information, im Programm, denn nur wenn eine solche Redundanz vorliegt, können Widersprüche entstehen, die auf einen logischen Programmierfehler hinweisen. Die Verknüpfung von deklarierten Elementen mit Typen erlaubt aber genau die Angabe solcher redundanter Information. Die Schaffung dieser Redundanz verlangt jedoch vermehrte Denk- und Schreibarbeit und ist zudem auch noch, im Falle eines fehlerfreien Programms, überflüssig. Dem kann man allerdings entgegnen, daß kaum eine Programmiererin auf Anhieb korrekte Programme schreibt, und wenn eine Typisierung Fehler zu finden in der Lage ist und somit nicht minder aufwendige Tests ersetzt, dann ist das nichts grundsätzlich Schlechtes.

Typen als Schnittstellen

Ein fünfter, oben nicht aufgezählter Grund zur Verwendung eines der heute üblichen Typsysteme ist übrigens die dadurch entstehende *Modularisierung von Programmen*, nämlich wenn ein Typ zugleich eine *Schnittstelle* oder ein *Interface* ausdrückt. Mehr dazu jedoch erst später (in Abschnitt 3.11.2).

Symptom Variablenfehlbelegung

Typfehler

Die der Fehlerentdeckung mittels Typsystemen zugrundeliegende These ist, daß ein guter Teil logischer Programmierfehler bereits frühzeitig daran erkannt werden kann, daß eine Variable einen Wert hat, den sie eigentlich niemals haben dürfte. So zeugt beispielsweise von einem Fehler, wenn einer Variable, die für Zahlen gedacht war, eine Zeichenkette zugewiesen wird. Wenn dann nämlich einem Ausdruck mit einer arithmetischen Operation, die Zahlen als Operanden verlangt, eine solchermaßen fehlbelegte Variable zugeführt wird, kann dieser nicht ausgewertet werden. Ohne Typprüfung würde dieser Fehler erst zur Laufzeit, also wenn der Ausdruck tatsächlich ausgewertet werden soll, in Erscheinung treten und hätte dann in aller Regel einen Programmabbruch zur Folge. Man nennt einen solchen Programmierfehler einen **Typfehler**.

Variablenfehlbelegung ohne Typfehler

Während ein Programmabbruch wenigstens noch eine erkennbare Reaktion auf einen Programmierfehler darstellt, ist es fast noch schlimmer, wenn ein logischer Fehler ohne solche bleibt. So kann es beispielsweise vorkommen, daß man einer Variable, deren Inhalt eine Strecke darstellen soll, eine andere zuweist, deren Inhalt eine Zeit repräsentiert. Mit beiden ließe sich gleich rechnen (dieselben Rechenoperationen durchführen), aber das Ergebnis wäre vermutlich falsch. Merken muß man das allerdings selbst, denn das Programm läuft einfach weiter.

Typinvarianten und Typannotationen

Man kann Variablenfehlbelegungen dieser Art verhindern, indem man Variablen mit expliziten **Invarianten** versieht, die die Menge ihrer zulässigen Werte beschränken, und dann darüber wacht, daß diese Invarianten immer eingehalten werden. Eine besonders einfache Möglichkeit, solche Invarianten zu spezifizieren, erlauben sog. **Typannotationen**, also die Verbindung einer Variable mit einem Typ, wobei der Typ eine Menge von Werten festlegt, die die Variable ausschließlich haben darf. In typisierten Programmiersprachen erfolgt die Typannotation explizit und zwingend bei der *Variablendeklaration*; in nicht oder nur optional typisierten Sprachen kann sie auch (für einzelne oder alle Variablen) hergeleitet (inferiert) werden und ist dann implizit.

Typkorrektheit

semantische Fehler

Ein Programm, in dem alle Variablenbelegungen immer alle Typinvarianten erfüllen, heißt **typkorrekt**. In einer Sprache, die durch ihr Typsystem Typkorrektheit festzustellen erlaubt, nennt man die logischen Fehler, die sich in unzulässigen Wertzuweisungen ausdrücken, auch **semantische Fehler** (und zwar, weil der Inhalt eines Programmelements nicht seiner intendierten Bedeutung entspricht). Dabei ist die Semantik des Programmelements im Programm zweimal, auf redundante, aber unterschiedliche Art, spezifiziert: in Form seines Typs und in Form seiner tatsächlichen Verwendung (festgelegt durch Zuweisungen und Methodenaufrufe). Läßt sich aus beiden ein Widerspruch ableiten, muß eine von beiden falsch gewesen sein.

Invariantenverletzung

Der einzige Weg, eine mit der Typisierung einer Variable ausgedrückte Invariante zu verletzen, also Typ*in*korrektheit herzustellen, ist per Wertzuweisung an die Variable. Ein Typsystem muß also lediglich alle Wertzuweisungen in einem Programm überprüfen, um Freiheit von semantischen Fehlern zu garantieren. Dazu zählen allerdings auch die impliziten Zuweisungen bei Methodenaufrufen (s. Abschnitt 1.6.1), die, auch wegen des dynamischen Bindens, nicht immer alle offensichtlich sind. Im folgenden heißen Zuweisungen und Methodenaufrufe, die nicht zu typinkorrekten Programmen führen können, **zulässig**.

Zulässigkeit von Zuweisungen und Methodenaufrufen

Schwierigkeit der Feststellung

Nun kann man sich vorstellen, daß es für einen Compiler selbst in einfachen Fällen nicht leicht ist, festzustellen, ob eine Wertzuweisung eine Invariante verletzt und somit zu einem typinkorrekten Programm führt. So ist das folgende STRONGTALK-Programmfragment

```
909   | i <Integer> |
910         i := 0.
911         i = 0 ifTrue: [i := 1] ifFalse: [i := 'dumm gelaufen']
```

das zunächst eine temporäre Variable `i` mit dem Typ `Integer` (in STRONGTALK wird die *Typannotation* hinter der Variable in spitzen Klammern angeführt) deklariert und ihr dann, in einer Folge von Anweisungen, zunächst `0` und dann `1` (beides Werte vom Typ `Integer`) zuweist, zwar typkorrekt im Sinne obiger Definition, aber um das zu erschließen, muß man schon wissen, daß die Bedingung in Zeile 911 immer erfüllt ist, der False-Zweig, der zu einer Verletzung der Invariante von `i` (nämlich daß die Werte immer vom Typ `Integer` sein müssen und somit nicht vom Typ `String` sein dürfen) führen würde, also nie ausgeführt wird. Im gegebenen Fall ist das zwar offensichtlich (und bereits von einer recht einfachen Programmanalyse feststellbar), aber es lassen sich auch Fälle konstruieren, in denen eine automatische Programmanalyse streiken muß.[56]

dynamische Typprüfung

Was man jedoch immer tun kann, um Typkorrektheit zu gewährleisten, ist, daß man zur Laufzeit vor einer Variablenzuweisung prüft, ob der zuzuweisende Wert den von der Variable geforderten Typ hat. Diese sog. **dynamische Typprüfung** (engl. dynamic type checking) hat jedoch den entscheidenden Nachteil, daß sie zu spät kommt, nämlich zu einem Zeitpunkt, in dem man bereits nicht mehr viel anderes machen kann als einen Fehler zu signalisieren (der dann günstigenfalls durch eine dafür vorgesehene Fehlerbehandlungsmethode aufgefangen wird, der aber in der Praxis häufig nur zu einem Programmabbruch führt). Man kann jedoch argumentieren, daß auch letzteres immer noch besser ist, als mit falschen Werten weiterzuarbeiten und damit entweder einen Programmabbruch an einer anderen Stelle, die nicht mehr so leicht mit der fehlerhaften Wertzuweisung

---

[56] Aus theoretischer Sicht ist das Problem sogar unentscheidbar, auch wenn solche Aussagen in der Regel auf pathologischen Programmkonstruktionen, die man in der Praxis kaum vorfinden wird, basieren.

in Zusammenhang zu bringen ist[57], in Kauf zu nehmen oder gar einen logischen Fehler, der überhaupt nicht erkannt wird.

Typprüfung in SMALLTALK

Man beachte übrigens, daß nach diesem Kriterium SMALLTALK — entgegen häufig zu lesenden Behauptungen — keine dynamische Typprüfung durchführt, da Typfehler erst im letztmöglichen Moment offenbar werden, nämlich wenn auf einer Variable eine Methode aufgerufen werden soll, die für das Objekt, auf das die Variable verweist, gar nicht definiert ist.[58] Um das zu verhindern, findet man in SMALLTALK-Code gelegentlich Figuren wie

```
912  methodDictionaries: anArray
913          "Private - Change the receiver's array of
914           method dictionaries to anArray."
915      (anArray isKindOf: Array)
916          ifFalse: [^ self error: 'must be an Array'].
917      dictionaryArray := anArray
```

(SMALLTALK EXPRESS entnommen). Dies entspricht natürlich genau einer dynamischen Typprüfung, nur daß hier Typ durch Klasse ersetzt wurde und die Prüfung eben nicht automatisch durch ein Laufzeittypsystem erfolgt, sondern ausprogrammiert werden muß.

statische Typprüfung

Sehr viel nützlicher als die dynamische Typprüfung ist die statische Typprüfung, bei der, trotz aller theoretischen Hindernisse, die Typkorrektheit zur Übersetzungszeit gewährleistet werden soll. Die Typprüfung ist damit Aufgabe des Compilers und nicht, wie im Fall der dynamischen Typprüfung, Aufgabe des Laufzeitsystems oder gar der Programmiererin. Wie wir schon gesehen haben, bedeutet dies nicht weniger, als einen Beweis zu führen, daß bei keiner Ausführung eines Programms eine Typinvariante verletzt wird. In der Praxis bedeutet dies aber, daß eine rein statische Typprüfung immer auch Programme zurückweist, die nützlich, sinnvoll und typkorrekt sind (s. obiges Beispiel der Zeilen 909 – 911, das zumindest typkorrekt ist: `i` erhält niemals einen Wert vom Typ `String`). Zwar kann man versuchen, möglichst wenige typkorrekte Programme durch die statische Typprüfung zurückzuweisen, aber wie man sich leicht vorstellen kann, wird mit steigender Genauigkeit das dazu notwendige Typsystem immer aufwendiger und schwieriger zu benutzen, bis es irgendwann so kompliziert ist wie das Programm, dessen Fehler es entdecken soll (so daß man bei auftretenden Typfehlern erst einmal prüfen muß, ob die Ursache tatsächlich in ei-

---

[57] Man denke etwa an die sog. Null pointer exceptions in JAVA, die erst dann auftreten, wenn mit einem Variablenwert `null` tatsächlich etwas gemacht werden soll, was unter Umständen erst am Ende einer langen Zuweisungskette der Fall ist.

[58] SMALLTALK und andere Programmiersprachen werden gelegentlich als dynamisch typisiert (dynamically typed) bezeichnet. Das aber ist Unsinn, denn eine Typisierung findet in SMALLTALK gar nicht, auch nicht zur Laufzeit, statt. Außerdem ist mit dynamischer Typisierung in der Regel dynamische Typprüfung gemeint. Was ein dynamischer Typ sein soll, ist auch gar nicht klar.

nem fehlerhaften Programm oder vielleicht nur in *fehlerhaften Typannotationen* liegt).

statisch und dynamisch in der Praxis

So ist die Suche nach einem guten Typsystem immer die Suche nach einem guten Kompromiß. Die meisten heute in der Praxis verwendeten Typssysteme basieren auf einem solchen: einer statischen Komponente, die möglichst viele Fehler findet, ohne dabei die Programmiererin allzusehr einzuschränken, und einer dynamischen Komponente, die den Rest erledigt. Eine erwähnenswerte Ausnahme davon macht C++: hier wird, zugunsten von Performanz (Speicherplatz und Geschwindigkeit), auf eine dynamische Komponente der Typprüfung vollständig verzichtet. Da die statische Typprüfung von C++ aber nicht alles abdeckt, sind C++-Programme auch nicht automatisch typkorrekt. Mehr dazu in Abschnitt 5.2.5.

3.2 Deklaration, Definition und Verwendung von Programmelementen

Programme bestehen aus Schlüsselwörtern und -zeichen sowie aus Programmelementen, deren Namen, die sogenannten Bezeichner, frei vergeben werden können. Viele Programmiersprachen verlangen, daß man diese Programmelemente vor der ersten Verwendung vereinbart oder deklariert. Durch eine solche **Deklaration** gibt man dem Compiler den Bezeichner bekannt; er kann ihn in der Folge wiedererkennen und mit der Deklaration in Verbindung bringen.

Bei der **Definition** wird dem Bezeichner das zugeordnet, wofür er steht. Im Falle einer Variable ist das eine bestimmte Stelle im Speicher, die genügend Platz bietet, um den Wert der Variable aufzunehmen. Im Falle einer Methode sind es die Anweisungen, die durch die Methode zusammengefaßt werden. Nicht selten (aber immer abhängig von der Programmiersprache) erfolgen Deklaration und Definition in einem Ausdruck. In solchen Fällen spricht man von Deklaration beziehungsweise Definition des Programmelementes in Abhängigkeit davon, was man gerade meint. Bei Variablen ist die Definition in der Regel allerdings uninteressant (der Speicherplatz wird vom Compiler automatisch zugewiesen), so daß man hier häufig Deklaration meint, selbst wenn man Definition sagt. Bei Methoden hingegen ist die korrekte Unterscheidung essentiell: In ihrer Deklaration werden ihr Name (in SMALLTALK der Nachrichtenselektor) bekanntgegeben und die formalen Parameter deklariert (zusammen die *Methodensignatur*), in ihrer Definition wird der Signatur der Methodenrumpf, also die Folge der mit der Methode verbundenen und bei einem Aufruf auszuführenden Anweisungen, zugeordnet. Von der Definition einer Variable zu unterscheiden ist übrigens ihre *Initialisierung*, bei der ihr (der dafür vorgesehenen Speicherstelle) ein Anfangswert zugewiesen wird.

Deklaration und Definition als Basis für die Verwendung

Deklaration und Definition dienen letztlich nur einem: der Verwendung. Die **Verwendung** eines Programmelements äußert sich darin, daß sein Name, der Bezeichner, im Programmtext angeführt oder referenziert wird. An der Stelle der Verwendung steht eine Variable für den Wert, den sie hat (bzw., wenn sie auf der linken Seite einer Zuweisung auftaucht, haben soll). Der Bezeichner einer

Methode steht hingegen meistens für ihren Aufruf (in manchen Sprachen durch ein Schlüsselwort eingeleitet), seltener auch für einen Zeiger auf die Implementierung.

Beispiel einer Variablendeklaration

Variablendeklarationen haben Sie in SMALLTALK bislang an zwei Stellen gesehen: als formale Parameter in Methodendeklarationen und als temporäre, lokale Variablen in *Methodenrümpfen*. Im Beispiel

```
methodeM: a mit: b
        | c |
```

stecken die Deklarationen von `a` und `b` als formaler Parameter und von `c` als temporäre Variable. Weitere Formen der Deklaration werden Sie im Verlauf dieses Kurstextes noch zu Gesicht bekommen. Literale werden übrigens weder deklariert noch definiert; ihre Verwendung ist zugleich ihre Deklaration und ihre Definition. Dasselbe gilt in SMALLTALK auch für Symbole (Symbolliterale).

fehlende und untypisierte Variablendeklarationen

In untypisierten Sprachen werden Variablen ohne Angabe eines Typs (wie z. B. in SMALLTALK) oder gar nicht (etliche Skriptsprachen und z. B. BASIC) deklariert. Letzteres hat den erheblichen Nachteil, daß Variablen durch ihre erste Verwendung quasi implizit deklariert (und damit angelegt) werden, was bei Schreibfehlern dazu führt, daß man plötzlich zwei Variablen anstatt einer hat, wobei die eine mit der anderen nichts zu tun hat. Eine solche Einladung zu Programmierfehlern sollten Sie als diejenige, die die Entscheidung für die Auswahl einer Sprache zu treffen hat, stets ablehnen.

3.3 Typdefinitionen und deren Verwendung

Damit durch ein Typsystem Fehler ausgeschlossen werden können, die auf der Voraussetzungen von Eigenschaften von Objekten beruhen, die diese gar nicht haben (also beispielsweise der Verwendung von Nicht-Zahlen in arithmetischen Ausdrücken), muß bekannt sein, welche Eigenschaften einem Typ und damit seinen Elementen zugeordnet sind. Im Fall von SMALLTALK sind die Eigenschaften, die mit einem Objekt verbunden werden können, schnell gefaßt: Es handelt sich einfach um die Menge der Methoden, die es versteht, also um sein *Protokoll* (s. Abschnitt 1.6.8). Ein solches Protokoll definiert einen Typ: Er umfaßt die Menge der Objekte, die über das Protokoll verfügen.

Wenn man nun eine Variable mit einem solchen Protokoll als Typ typisiert und das Programm typkorrekt ist, dann ist garantiert, daß jede Methode, die im Protokoll enthalten ist und die auf der Variable aufgerufen wird, auch für den Inhalt der Variable, das referenzierte Objekt, definiert ist. Typfehler, also Fehler der Sorte „does not understand“ (s. Abschnitt 1.6.2), treten dann nicht mehr auf.

Typen als Teile von Typdefinitionen

Nun kommen in Protokollen aber selbst Variablen vor, nämlich die formalen Parameter der Methoden, die das Protokoll ausmachen. Außerdem ist eine Methode ein Programmelement, das für ein Objekt steht (mit der Ausführung ein Objekt liefert) und deswegen selbst, genau wie Variablen, typisiert werden sollte. Protokolle definieren also nicht nur Typen, sie verwenden auch selbst welche, nämlich indem sie die Typen der Ein- und Ausgabeobjekte spezifizieren. Ein ein-

faches Beispiel für eine Typdefinition, die selbst Typen verwendet, ist die folgende:

```
920   name ^ <String>
921   name: einName <String> ^ <Self>
922   alter ^ <Integer>
923   alter: einAlter <Integer> ^ <Self>
```

Wie schon bei einer temporären Variable, stehen die *Typannotationen* von formalen Parametern in STRONGTALK in spitzen Klammern dahinter. Diese Schreibweise sollten Sie nicht allzusehr verinnerlichen, da andere Programmiersprachen die spitzen Klammern zur Kennzeichnung von Typvariablen (in Abschnitt 3.12 behandelt) verwenden. Der Rückgabetyp einer Methode wird durch ein vorangestelltes Dach (^) gekennzeichnet und folgt auf den letzten Parameter. Da es in SMALLTALK keine Methoden gibt, die nichts zurückgeben (eine Methode ohne explizite Rückgabeanweisung gibt in SMALLTALK ja immer das Empfängerobjekt zurück), muß auch immer ein Rückgabetyp angegeben werden. Ist dies der Typ selbst, kann der Name `Self` verwendet werden. Es handelt sich dabei gewissermaßen um eine **Pseudo-Typvariable** (entsprechend der Pseudovariable `self`, deren Typ sie darstellt).

Schema für Typdefinitionen

Falls Sie sich wundern, daß obige Zeilen kein Schlüsselwort zur Einleitung der Typdefinition beinhalten: STRONGTALK ist, genau wie SMALLTALK, ein interaktives, browser-gestütztes System, in dem Typen in Formulare eingetragen und nicht in Textdateien spezifiziert werden. Gleichwohl fällt auf, daß innerhalb der Typdefinition in den spitzen Klammern (also da, wo Typen stehen sollen) keine Typdefinition auftauchen, sondern Namen. Und tatsächlich wird in STRONGTALK jeder Typ benannt (in seiner Typdefinition mit einem Namen versehen). Im folgenden werden Typen, ähnlich wie Klassen, in tabellarischer Form notiert. Der Typ Person etwa mit obigem Protokoll liest sich dann wie folgt:

Typ | Person

Protokoll

```
924   name ^ <String>
925   name: einName <String> ^ <Self>
926   alter ^ <Integer>
927   alter: einAlter <Integer> ^ <Self>
```

Selbsttestaufgabe 3.1

Definieren Sie den Typ `Boolean` gemäß obigem Schema!

Typkonstruktoren

In STRONGTALK ist die Protokollbildung der einzige sog. **Typkonstruktor**, d. h., das einzige Sprachkonstrukt, mit dem man neue Typen definieren kann. Andere Programmiersprachen sehen ein reichhaltigeres Angebot vor: In PASCAL beispielsweise gibt es die Typkonstruktoren `record`, `array of`, `set of`, `file of`, Zeiger auf (^) sowie Aufzählungen (enumerations) und Teilbereiche (ranges). In C++ gibt es u. a. `class` und `struct` (entsprechend `record` in PASCAL), JAVA, C# und EIFFEL bieten auch jeweils verschiedene Typkonstruktoren an. Für eine puri-

stische Sprache wie SMALLTALK bzw. STRONGTALK reicht jedoch einer vollkommen aus.

Wie man leicht einsieht, gibt es in STRONGTALK keine primitiven Typen, also keine Typen, deren Definitionen nicht selbst auf einen oder mehrere Typen zurückgeführt werden müßte. Daran rührt auch die Optionalität der Annotierung nichts: Selbst wenn man eine *Typannotation* wegläßt (was immer erlaubt ist), hat die entsprechende Variable bzw. der Rückgabewert der Methode einen Typ, nur wird er an dieser Stelle nicht angegeben. Das wirft natürlich die Frage auf, wie man Typen unter zwangsläufiger Selbstbezüglichkeit überhaupt eine Bedeutung beimessen kann.

3.3.1 Induktiver Aufbau von Typen und Semantik

Um diese Frage zu beantworten, ist es zunächst interessant, festzustellen, daß es Typen gibt, die sich ausschließlich auf sich selbst beziehen, deren Bedeutung also zumindest nicht von der anderer Typen abhängt. Das klassische Beispiel hierfür ist `Boolean`: Alle seine Operationen fordern den Typ `Boolean` als Operanden und haben `Boolean` als Typ zum Ergebnis. Aber woher erhält `Boolean` seine Bedeutung?

denotationale Semantik

Eine eher theoretisch relevante Möglichkeit, solchen nur auf sich selbst beruhenden Typen eine Bedeutung zu geben, ist, sie auf bekannte externe Formalismen abzubilden. Im Beispiel von `Boolean` ist dies natürlich die boolesche Algebra. Jede, die die boolesche Algebra kennt und akzeptiert, wird auch den Typ `Boolean` sofort verstehen und akzeptieren (so er denn den Erwartungen entsprechend definiert ist). Entsprechend läßt sich ein Typ `Fraction` mit den Operationen `+`, `-`, `*` und `/` definieren, der die rationalen Zahlen mit den entsprechenden Operationen repräsentiert. Nimmt man dann noch `Boolean` als mit Bedeutung (Semantik) versehen an, kann man noch Vergleichsoperationen wie `=`, `>`, `<` etc. hinzufügen, ohne in Interpretationsprobleme zu laufen. Andere Typen, für die es eine solche direkte Abbildung nicht gibt, die aber in ihrer Definition rekursiv auf solche Typen zurückgeführt werden können, kann man „induktiv über deren Aufbau“ eine Bedeutung beimessen. Man nennt eine solche Art des Versehens mit Bedeutung eine *denotationale Semantik*.

operationale Semantik

Eine andere, für die praktische Programmierung relevantere Möglichkeit ist, einen Typ und seine Operationen auf Anweisungen einer (gedachten oder realen, Hauptsache wohlspezifizierten) Maschine abzubilden. Die Abbildung für Basistypen wie `Rational` oder `Boolean` ist in der Programmiersprache bzw. deren Compiler gewissermaßen hart verdrahtet. Für von der Programmiererin definierte Typen kann sie dies hingegen nicht sein; deren Bedeutung kann aber vom Compiler, wiederum „induktiv über deren Aufbau“, aus der Bedeutung von Typen, die eine vorgegebene Semantik haben, abgeleitet werden. Man nennt dies dann auch eine *operationale Semantik*.

Man beachte, daß es für beide Arten der Semantik notwendig ist, daß sich alle Typen auf solche zurückführen lassen, deren Bedeutung vorausgesetzt werden kann. Es gibt also kein vollständig in sich selbst definiertes, von Externem unab-

hängiges System. Selbst SMALLTALK bzw. STRONGTALK ist kein solches: Auch wenn die Implementierung von `Boolean` nicht „hart verdrahtet“, sondern auf dynamisches Binden abgewälzt wird, so sind dafür aber mindestens die beiden Wahrheitswerte `true` und `false` dem System bekannt, und `Integer` und `Float` (nicht jedoch `Fraction`!) sind „fest verdrahtet“, inklusive der Vergleichsrelationen (die ja die Wahrheitswerte zum Ergebnis haben).

Zusammenhang mit abstrakten Datentypen

Wenn Sie Kurs 01661 („Datenstrukturen“) bereits belegt haben oder ähnliches Vorwissen besitzen, dann erinnert Sie obiges Schema von Typdefinitionen vielleicht an die Schreibweise abstrakter Datentypen. Auch dort wird ein Typ syntaktisch als eine Menge von Operationen (Funktionen) beschrieben, deren Operanden (Argumente) alle selbst typisiert sind. Es gibt jedoch mindestens zwei wichtige Unterschiede zwischen den Signaturen eines abstrakten Datentyps und dem Protokoll eines STRONGTALK-Typs:

1. Abstrakte Datentypen sind nicht objektorientiert in dem Sinne, daß die Objekte keinen Zustand haben und bei Operationen (Funktionen) die Objekte, auf denen die Operationen ausgeführt werden, nicht ihren Zustand wechseln. Statt dessen geben Operationen neue Objekte zurück. Die Objekte der abstrakten Datentypen sind also gewissermaßen alle unveränderlich (vgl. Abschnitt 1.6.5).
2. Entsprechend haben die den Methoden eines Protokolls entsprechenden Funktionen in den Spezifikationen abstrakter Datentypen immer ein Argument mehr, und zwar vom Typ des Datentypen selbst. Dieses Argument entspricht in der objektorientierten Programmierung dem Nachrichtenempfänger, dem impliziten Parameter `self`.

Der Bezug zu abstrakten Datentypen ist auch eine beliebte Möglichkeit, Typen einer Programmiersprache mit einer Semantik zu versehen.

3.3.2 Verwendung definierter Typen

Definierte Typen können in Programmen verwendet werden, in STRONGTALK bei der Deklaration von (anderen) Typen, von Variablen, von Blöcken und von Methoden. Man spricht dann von einer **Typisierung** der deklarierten Programmelemente. Die Verwendung in Typdefinitionen haben Sie ja oben bereits kennengelernt, die Verwendung in Methoden verläuft analog. Variablen (Instanzvariablen, temporäre Variablen etc.) werden in STRONGTALK genau wie formale Parameter (die ja auch Variablen sind) typannotiert, nämlich durch Hintanstellung eines in spitzen Klammern eingeschlossenen Typnamens. Bei Blöcken taucht der Rückgabetyp im selben Segment wie die formalen Parameter auf, also vor dem Separator `|`. Die vollständig typannotierte Klasse `Stack` aus Abschnitt 2.2.2 sieht in STRONGTALK beispielsweise so aus:

```
Klasse                  Stack
benannte Instanzvariablen   stackcontent <Array>
                        stackpointer <Integer>
indizierte Instanzvariablen   nein
Instanzmethoden
928   push: anElement <Object> ^ <Self>
929           "legt neues Element auf Stapel"
930         stackpointer := stackpointer + 1.
931         stackcontent at: stackpointer put: anElement

932   pop ^ <Self>
933           "entfernt oberstes Element vom Stapel"
934         stackpointer := stackpointer - 1

935   top ^ <Object>
936           "liefert oberstes Element des Stapels"
937         ^ stackcontent at: stackpointer
```

Ein Beispiel für einen typisierten Block finden Sie in Abschnitt 3.12.3, Codezeile 1026.

3.4 Zuweisungskompatibilität

Die Typisierung von Variablen (und anderen Programmelementen — wenn im nachfolgenden nur von Variablen die Rede ist, dann sind letztere meistens mit gemeint) soll also bewirken, daß in einem Programm jede Variable nur die Werte haben kann, für die sie (die Variable) vorgesehen ist (die Einhaltung der *Typinvariante*). Voraussetzung dafür ist zum einen, daß jeder Variable ein Typ zugeordnet ist, zum anderen, daß auch jedes Objekt sowie jeder Ausdruck, der für einen Wert oder ein Objekt steht, einen Typ hat. Ersteres geschieht in sogenannten Variablendeklarationen, letzteres ergibt sich aus den zu einem *Typsystem* gehörenden *Regeln zur Zuordnung eines Typs zu Ausdrücken*, nämlich

- bei Literalen aus der Art des Literals, dessen Typ dem Compiler bekannt ist,
- bei der Instanziierung aus dem noch zu klärenden Zusammenhang von der instanziierten Klasse mit den Typen eines Programms sowie
- bei Nachrichtenausdrücken aus der Deklaration der dazugehörigen Methode, die ja (genau wie eine Variablendeklaration) angeben muß, welchen Typs die Objekte sind, die sie liefert.

Feststellung der Zuweisungskompatibilität

Es bleibt die Frage nach den ebenfalls zu einem Typsystem gehörenden *Typregeln, die von Ausdrücken einzuhalten sind*, nämlich wie die Typkorrektheit bzw. andernfalls die Verletzung einer Typinvariante genau festgestellt wird. Es ist ja bereits klar, daß es dazu ausreicht, die Wertzuweisungen in einem Programm zu überprüfen. Diese Überprüfung findet in der Regel in Form der Feststellung der sog. *Zuweisungskompatibilität* statt. Die Sprachregelung ist hier leider nicht ganz einheitlich, aber im folgenden gehen wir davon aus, daß alle typisierten Sprachen den Begriff der Zuweisungskompatibilität kennen und sich lediglich in ihren Definitionen der Regeln, die für das Bestehen einer Zuweisungskompatibilität ein-

gehalten werden müssen, unterscheiden. Vor allem darum wird es in den nächsten Abschnitten gehen.

Angenommen, zwei temporäre Variablen `anzahl` und `erfolgreich` seien wie folgt deklariert:

```
938   | anzahl <Integer> erfolgreich <Boolean> |
```

Dann sind, unter der Annahme, daß `12` vom Typ `Integer` ist und `true` vom Typ `Boolean`, die Zuweisungen

```
939   anzahl := 12
940   erfolgreich := true
```

zulässig (da sie keine Typinvariante verletzen),

```
941   anzahl := false
942   erfolgreich := 12
```

hingegen nicht. Ist eine Zuweisung zulässig, dann spricht man auch von einer **Zuweisungskompatibilität** der beteiligten Typen. Die für das Programmieren relevante Implikation ist allerdings die umgekehrte: Wenn zwei Typen zuweisungskompatibel sind, dann gilt, daß eine entsprechende Zuweisung *zulässig* ist, also zu keiner Verletzung einer Typinvariante führt. Wie Sie noch sehen werden, verlangt Zuweisungskompatibilität keineswegs identische Typen; daraus ergibt sich aber eine sprachliche Uneindeutigkeit, die zunächst behoben werden muß.

Sprachgebrauch

Transitivität

Dem Satz „`a` ist zuweisungskompatibel mit `b`“ kann man nicht eindeutig entnehmen, ob nun `a b` zugewiesen werden kann oder `b a`. Daß beides geht, ist nur dann der Fall, wenn die beteiligten Typen äquivalent in einem noch zu bestimmenden Sinne sind, was aber, wie schon gesagt, nicht unbedingt der Fall sein muß. Im folgenden soll daher die Richtung der erlaubten Zuweisung so gelesen werden, daß beim Satz „`a` ist zuweisungskompatibel mit `b`“ die Zuweisung `b := a` zulässig ist. Die umgekehrte Richtung, `a := b`, kann ebenfalls zulässig sein; dies wird durch den Satz jedoch nicht ausgesagt. Zuweisungskompatibilität ist übrigens (in der Regel) eine transitive Eigenschaft: Wenn `a` zuweisungskompatibel mit `b` ist und `b` zuweisungskompatibel mit `c`, dann ist auch `a` zuweisungskompatibel mit `c`.

Zuweisungskompatibilität bei Methodenaufrufen

Auch bei impliziten Zuweisungen wie der Parameterübergabe von Methodenaufrufen (den dabei stattfindenden Zuweisungen der aktuellen an die formalen Parameter; s. Abschnitt 1.6.1) impliziert Zuweisungskompatibilität Typkorrektheit. Außerdem kann eine Methode, wenn sie Werte zurückgibt, ja selbst in rechten Seiten von Zuweisungen auftreten; der Typ dieser Werte muß dann mit der Variable auf der linken Seite zuweisungskompatibel sein. So sind bei Vorliegen der Deklarationen

```
943   m: p <E> ^ <A>
944   | e <E> a <A> |
```

sowohl die explizite als auch die impliziten Zuweisungen in

```
945   a := self m: e
```

zulässig; den Methodenaufruf kann man im übertragenen Sinne als *zulässig* bezeichnen.

3.5 Typäquivalenz

Es stellt sich nun die Frage, wann ein Typ mit einem anderen zuweisungskompatibel ist. Offensichtlich ist dies der Fall, wenn die Typen dieselben (identisch) sind. Wie bereits oben erwähnt, ist dies aber keine notwendige Voraussetzung für die Zuweisungskompatibilität. Es ist nämlich zumindest auch möglich, daß sich zwei verschiedene Typdefinitionen bis auf ihre Namen gleichen, daß also z. B. in STRONGTALK die die Typdefinitionen ausmachenden Mengen der Methodensignaturen gleich sind. Man spricht in diesen Fällen von einer **Typäquivalenz**.

Unterscheidung von nominaler und struktureller Typäquivalenz

Von der Typäquivalenz gibt es zwei Arten: die **nominale** (sich auf den Namen beziehende) Typäquivalenz, auch **Namensäquivalenz** genannt, und die **strukturelle** Typäquivalenz, auch als **Strukturäquivalenz** bezeichnet. Während die nominale Typäquivalenz verlangt, daß zwei Deklarationen (beispielsweise von Variablen) dieselben Typen anführen, damit Zuweisungskompatibilität vorliegt, kommt es bei der strukturellen Typäquivalenz lediglich darauf an, daß die Typen paarweise gleich definiert sind (also die gleichen Eigenschaften von ihren Werten verlangen), die Typen sich also in ihrer Struktur, aber nicht unbedingt in ihren Namen gleichen.

formale Eigenschaften der Typäquivalenz

Typäquivalenz ist eine symmetrische Eigenschaft: Wenn ein Typ `A` (nominal oder strukturell) äquivalent zu einem Typ `B` ist, dann ist `B` genauso äquivalent zu `A`. Die Reflexivität der Typäquivalenz, also daß jeder Typ äquivalent zu sich selbst ist, ergibt sich von selbst. Außerdem ist Typäquivalenz transitiv: Wenn `A` (nominal oder strukturell) äquivalent zu `B` ist und `B` in der gleichen Art äquivalent zu `C`, dann ist auch `A` äquivalent zu `C` (und, aufgrund der Symmetrie, `C` äquivalent zu `A`).

3.5.1 Strukturäquivalenz

Um strukturelle Typäquivalenz festzustellen, werden die Definitionen der beteiligten Typen *rekursiv expandiert*, was soviel heißt wie daß in einer Typdefinition vorkommende Namen anderer Typen durch ihre Struktur ersetzt werden. Nimmt man beispielsweise die Typdefinitionen

Typ

```
Person
```

Protokoll

```
946   sitz ^ <Wohnung>
947   sitz: einWohnsitz <Wohnung> ^ <Self>
```

Typ `Wohnung`

Protokoll

```
948   straße ^ <String>
949   straße: einStraße <String> ^ <Self>
950   ort ^ <String>
951   ort: einOrt <String> ^ <Self>
```

Typ `Firma`

Protokoll

```
952   sitz ^ <Büro>
953   sitz: einFirmensitz <Büro> ^ <Self>
```

Typ `Büro`

Protokoll

```
954   straße ^ <String>
955   straße: einStraße <String> ^ <Self>
956   ort ^ <String>
957   ort: einOrt <String> ^ <Self>
```

dann sind die Typen `Person` und `Firma` sowie `Wohnung` und `Büro` jeweils strukturäquivalent, aber nicht namensäquivalent. Bei der Strukturäquivalenz haben Namen also lediglich die Funktion der abkürzenden Schreibweise, bei der Namensäquivalenz hingegen auch eine von der Struktur unabhängige Bedeutung. Namensäquivalenz impliziert Strukturäquivalenz, aber nicht umgekehrt; Namensäquivalenz ist somit das stärkere Konzept.

Bedeutung der Strukturäquivalenz

Strukturäquivalenz als Bedingung der Zuweisungskompatibilität reicht aus, um *Typfehler*, also logische und Laufzeitfehler, die auf der Annahme einer nicht vorliegenden Eigenschaft (Methode) bei einem Wert einer Variable basieren, zu verhindern. Sie garantiert, daß die Methoden eines Programms auf den jeweiligen Empfängerobjekten mit den geforderten Parameterobjekten auch durchgeführt werden können. So kann z. B. bei erfolgreicher Typprüfung (und daher vorliegender Typkorrektheit) ohne Kenntnis der konkreten Inhalte der Variablen sichergestellt werden, daß bei Vorliegen der Deklaration `p <Person>` der Ausdruck

```
958   p sitz straße: 'Heimatstraße'
```

keine Typfehler produziert, und gleichzeitig der Ausdruck

```
959   p sitz: 'zuhause'
```

schon zur Übersetzungszeit als fehlerhaft zurückgewiesen wird, da er zu einer Variablenfehlbelegung (die in SMALLTALK noch problemlos möglich gewesen wäre) führt. Man beachte, das letztere sogar zu einer Speicherschutzverletzung führen könnte, wenn die Variable `p` — wie in vielen Sprachen mit Typsystem — Wertsemantik hätte, nämlich dann, wenn der übergebene String größer ist als der zur Aufnahme der Wohnung vorgesehene Speicherplatz.

Type branding

Strukturäquivalenz ist eine rein syntaktische Bedingung. Insbesondere können bei geforderter Strukturäquivalenz Typen zufällig zuweisungskompatibel sein,

die inhaltlich überhaupt nichts miteinander zu tun haben. Dadurch können Objekte, die eigentlich getrennten Typen (disjunkten Wertebereichen) angehören, über Kreuz und über die Typgrenzen hinweg zugewiesen werden. *Semantische Fehler* sind also immer noch möglich. Man trifft daher in Sprachen mit Strukturäquivalenz gelegentlich die Praxis an, jedem Typ eine für ihn charakteristische Methode exklusiv zuzuordnen, so daß er mit keinem anderen mehr strukturäquivalent ist. Diese Technik nennt man **Type branding**.

3.5.2 Namensäquivalenz

inhaltliche Filterfunktion der Typprüfung

Nun können Typen neben ihrer formalen Funktion, Fehler zu vermeiden, noch eine inhaltliche, nämlich eine *Filterfunktion* ausfüllen. Diese setzt allerdings voraus, daß dem Typ auch eine Bedeutung, die über seine bloße Struktur (seine Syntax) hinausgeht, beigemessen werden kann. Dies geschieht heute vor allem durch die Benennung des Typs, die dann, gepaart mit Namensäquivalenz als Bedingung der Zuweisungskompatibilität, verlangt, daß einer Variable nur Werte gleicher Bedeutung zugewiesen werden können. Eine Zuweisung einer Wohnung an ein Büro oder umgekehrt ist dann, trotz im obigen Beispiel strukturell gleich definierter Typen und deswegen ausbleibenden Typfehlern, aufgrund fehlender Namensgleichheit ausgeschlossen, was auch sinnvoll ist, da es sich dabei mit einer gewissen Wahrscheinlichkeit um einen logischen Programmierfehler handelt, der auf mechanische Art sonst kaum zu entdecken wäre. Die Filterfunktion der geforderten Namensäquivalenz drückt also eher eine Absicht der Programmiererin aus denn eine technische Notwendigkeit. Die Bedeutung gerade dieser Funktion sollte man jedoch nicht unterschätzen — nur wenige Möglichkeiten, Fehler in einem Programm aufzudecken bzw. zu vermeiden, sind so einfach zu haben.

Rechnen mit Größen

Ein der Typprüfung per Namensäquivalenz ähnliches Prinzip kommt übrigens in der Physik zur Anwendung: Bei ihren Berechnungen führen Physikerinnen stets eine Art Typprüfung durch, indem sie nicht nur mit den Beträgen der physikalischen Größen, sondern auch mit deren Einheiten rechnen. Wenn Physikerinnen also beispielsweise eine Geschwindigkeit berechnen und bei der Behandlung der Einheiten etwas anderes als m/s herauskommt, dann steckt im Rechenvorgang ein Fehler — das Ergebnis hat nicht den richtigen Typ (die richtige Einheit) und ist deswegen mit hoher Wahrscheinlichkeit falsch.

Nachteil der Namensäquivalenz

Namensäquivalenz hat aber auch einen entscheidenden Nachteil: Sie setzt voraus, daß getrennt voneinander entwickelte Programme zumindest an ihren Schnittstellen (also da, wo Objekte ausgetauscht werden) dieselben Typen verwenden. Dies kann für die Interoperabilität von getrennt voneinander entwickelten Programmen (wie z. B. Web services) ein echtes Hindernis sein.

strukturell vs. nominal

Strukturelle Typäquivalenz bietet mehr Flexibilität als nominale: Sie erlaubt Äquivalenz von Typen, bei deren Definition man vom jeweils anderen nichts wußte. Die erhöhte Flexibilität hat jedoch ihren Preis: Zufällige strukturelle Übereinstimmungen können zu einer Äquivalenz führen, die nicht der intendierten Semantik entspricht. *Type branding* führt in solchen Fällen eine Namensäquivalenz durch die Hintertür ein, mit dem Vorteil, daß diese optional ist.

3.6 Typerweiterung

Wie bereits in Abschnitt 3.4 angedeutet, verlangt die Zuweisungskompatibilität nicht unbedingt Typäquivalenz. Tatsächlich reicht es ja, bei einer rein strukturellen (syntaktischen) Betrachtung, voll aus, daß der Typ der rechten Seite einer Zuweisung das Protokoll (die Menge der Methoden) des Typs der linken Seite enthält, um in der Folge Typfehler zu vermeiden. Anders ausgedrückt: Der Typ auf der rechten Seite einer Zuweisung darf eine Erweiterung dessen auf der linken Seite um zusätzliche Methoden sein.

Vererbung der Methodendeklaration

Die sog. **Typerweiterung** (engl. type extension), wie sie z. B. in den Programmiersprachen MODULA-3 und OBERON (beides Nachfolger von PASCAL) Verwendung findet[59], sieht genau dies vor. Eine Typerweiterung des obigen Typs `Büro` um ein Länderkennzeichen sieht dann beispielsweise wie folgt aus:

```
Typ                 InternationalesBüro
erweiterter Typ     Büro
Protokoll
960   länderkennzeichen ^ <String>
961   länderkennzeichen: einLänderkennzeichen <String> ^ <Self>
```

Der erweiternde Typ, hier `InternationalesBüro`, wird also relativ zu einem bereits bestehenden, dem erweiterten Typ (hier `Büro`), definiert. Die Methodendeklarationen des erweiterten Typs werden dabei gewissermaßen an den erweiternden Typ *vererbt*; dieser braucht sie also nicht noch einmal zu wiederholen.

Zuweisungskompatibilität bei Typerweiterung

Wie man nun leicht einsieht, können Variablen, deren deklarierter Typ `Büro` ist, ohne weiteres auch Objekte vom Typ `InternationalesBüro` enthalten, ohne daß dies zu Typfehlern führt, da alle Methoden, die für `Büro` vorgesehen sind, auch in `InternationalesBüro` vorkommen.[60] Das Umgekehrte ist jedoch nicht der Fall: Wenn man einer Variable vom Typ `InternationalesBüro` ein Objekt vom Typ `Büro` zuweisen könnte, dann hätte man immer dann ein Problem, wenn man über diese Variable auf dessen Methoden zu Länderkennzeichen zugreifen

---

[59] N Wirth „Type extensions“ *ACM Transactions on Programming Languages and Systems* 10:2 (1988) 204–214; „extension“ hier im Sinne von Erweiterung und nicht im Sinne der Extension (Ausdehnung) als Gegenstück zur Intension

[60] Bei Variablen mit Referenzsemantik geht das ohne Einschränkungen, denn die Größe des durch einen Pointer belegten Speicherplatzes ist immer gleich. Bei Variablen mit Wertsemantik hingegen muß der Wert eines erweiterten Typen erst auf einen des Basistypen projiziert werden, d. h., die Inhalte eventueller zusätzlicher Felder müssen unter den Tisch fallen, da für sie im für die Variable reservierten Speicher kein Platz ist. Solange in den Typdefinitionen aber gar keine Felder vorkommen, ist der Typ einer Variable auch nicht für die Berechnung des zur Aufnahme eines Objekts des Typs benötigten Speichers geeignet. Das ist z. B. in STRONGTALK der Fall — und auch gut so, denn Felder zählen nach vorherrschender Meinung zur Implementation und sind, genau wie bei abstrakten Datentypen, nicht Bestandteil einer Typdefinition.

wollte, weil diese schlichtweg für das Objekt nicht definiert sind. Die Zuweisungskompatibilität unter Typerweiterung regelt der Begriff der Typkonformität.

3.7 Typkonformität

Einen Typ, dessen Definition alle deklarierten Elemente der Definition eines anderen Typen enthält, nennt man mit dem anderen **typkonform**. So ist im obigen Beispiel `InternationalesBüro` mit `Büro` typkonform. Typkonformität ist in vielen Sprachen eine notwendige und hinreichende Voraussetzung für die *Zuweisungskompatibilität*: Es darf dann ein Objekt vom Typ `InternationalesBüro` einer Variable vom Typ `Büro` zugewiesen werden.

formale Eigenschaften der Typkonformität

Typkonformität ist aber reflexiv, d. h., jeder Typ ist konform zu sich selbst. Sie ist weiterhin transitiv: Wenn `A` typkonform zu `B` ist und `B` typkonform zu `C`, dann ist auch `A` typkonform zu `C`. Wie man sich leicht denken kann, ist die Typkonformität jedoch im Gegensatz zur Typäquivalenz nicht symmetrisch: Aus der Tatsache, daß ein Typ `B` typkonform zu einem Typ `A` ist, folgt nicht, daß auch `A` typkonform zu `B` ist. Vielmehr ist dies mit einer kleinen Ausnahme sogar zwingend nicht der Fall: Typkonformität ist meistens antisymmetrisch, was soviel heißt wie daß wenn `B` zu `A` und `A` zu `B` typkonform ist, daß dann `A` und `B` identisch sein müssen.

strukturelle und nominale Typkonformität

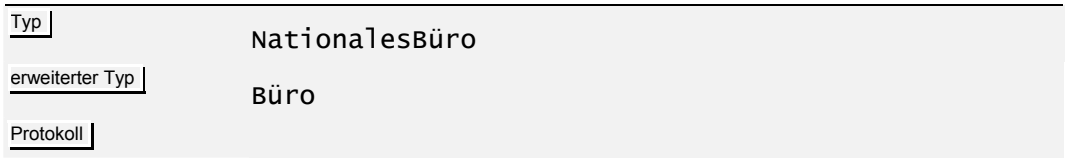
Von der Typkonformität gibt es, genau wie von der Typäquivalenz, zwei Varianten, nämlich eine **strukturelle Typkonformität** und eine namensgebundene (**nominale**) **Typkonformität**. Zur strukturellen Typkonformität reicht es aus, wenn der konforme Typ wie oben alle Elemente des Typs, zu dem er konform sein soll, enthält: Der Typ mit der Definition

```
Typ
                    InternationaleWohnung
Protokoll
962   straße ^ <String>
963   straße: einStraße <String> ^ <Self>
964   ort ^ <String>
965   ort: einOrt <String> ^ <Self>
966   länderkennzeichen ^ <String>
967   länderkennzeichen: einLänderkennzeichen <String> ^ <Self>
```

ist also zum Typ `Büro` strukturell konform. Für die nominale Konformität muß zusätzlich und explizit die Erweiterung eines (oder Ableitung von einem) anderen Typ angegeben werden: die Definition von `InternationalesBüro` aus Abschnitt 3.6 ist also mit `Büro` nicht nur strukturell, sondern auch nominal konform. Da bei der Erweiterung alle Elemente des Typs, der erweitert wird, beim erweiternden erhalten bleiben, folgt die Konformität aus der Erweiterung.

Spezialfall Erweiterung ohne Erweiterung

Nun ist die Teilmengenbeziehung reflexiv, was auf die Typerweiterung übertragen bedeutet, daß ein Typ eine Erweiterung eines anderen sein kann, ohne tatsächlich etwas hinzuzufügen. So ist beispielsweise gemäß folgender Typdefinition



`NationalesBüro` eine Erweiterung von `Büro` und mit den Variablendeklarationen

```
968   | b <Büro> nb <NationalesBüro> |
```

die Zuweisung

```
969   b := nb
```

bei geforderter nominaler und struktureller Typkonformität zulässig. Die umgekehrte Zuweisung ist

```
970   nb := b
```

ist jedoch bei geforderter nominaler Typkonformität nicht zulässig, da `Büro` eben *nicht* nominal konform ist zu `NationalesBüro`; strukturell ist es es hingegen schon.

Typäquivalenz als Spezialfall der Typkonformität

Typäquivalenz impliziert übrigens, jeweils für die nominale und die strukturelle Form getrennt, Typkonformität: Zwei äquivalente Typen sind auch immer konform. Das Umgekehrte ist jedoch meistens nicht der Fall: Zwar ist ein Typ, der angibt, einen anderen zu erweitern, ohne jedoch etwas hinzuzufügen, zu dem anderen strukturell äquivalent, aber nominal schon nicht mehr; sobald etwa hinzugefügt wird, ist es mit der Äquivalenz sowieso vorbei.

Filterfunktion

Genau wie bei der Typäquivalenz hat die nominale Typkonformität zusätzlich zur Gewährleistung der Zuweisungskompatibilität und somit der Abwesenheit von Typfehlern (die ja auch bei einer strukturellen Typkonformität schon gegeben wäre) eine *Filterfunktion*: Es sind nur Objekte von solchen Typen Variablen zuweisbar, für die das die Programmiererin aufgrund semantischer (inhaltlicher) Überlegungen ausdrücklich so vorgesehen hat. Auf diese Filterfunktion werden wir später im Zusammenhang mit sog. *Tagging* oder *Marker interfaces* (in Abschnitt 4.12) noch zurückkommen.

Vorteile der nominalen Typkonformität

Da die Typkonformität bei Nennung des Typen, von dem ein neuer per Erweiterung abgeleitet wird, über den Vorgang der Erweiterung automatisch gegeben ist (und so keine aufwendigen, fallweisen Konformitätstests durchgeführt werden müssen), setzen die meisten gebräuchlichen, typisierten Programmiersprachen auf nominale Typkonformität als Bedingung für die Zuweisungskompatibilität. Interessanterweise wurde STRONGTALK, das ursprünglich ein auf struktureller Konformität beruhendes Typsystem (inkl. *Type branding*) hatte, inzwischen auf nominale Typkonformität umgestellt. Als Begründung wurde angeführt, daß ein strukturelles Typsystem, insbesondere eines, bei dem Typen nicht explizit benannt werden, es der Programmiererin nicht erlaubt, ihre Absicht (intendierte Semantik, die obengenannte Filterfunktion) auszudrücken, was Programme schwerer zu lesen und zu debuggen macht, und daß die Fehlermeldungen, die

eine strukturelle Typprüfung produziert, sich oft nicht auf die eigentliche Fehlerquelle beziehen und sehr schwer zu verstehen sind [STRONGTALK2.0].

Konformität bei Funktionsaufrufen

Fragen der Zuweisungskompatibilität unter Typerweiterung spielen übrigens auch bei Funktionsaufrufen, bei denen ja *implizite Zuweisungen* auftreten (s. Abschnitt 1.6.1), eine wichtige Rolle. So muß bei dem Ausdruck

```
971   a := self m: e
```

der Typ von `e` eine Erweiterung des in `m` für den Parameter geforderten Typ sein und der Rückgabetyp von `m` eine Erweiterung des Typs von `a`.

3.8 Typeinschränkung

Typerweiterung ist nicht die einzige Möglichkeit, auf der Basis eines bereits bestehenden einen neuen, verwandten Typen zu erzeugen; Typeinschränkung ist eine andere.

Typeinschränkung durch Löschen von Methoden

Eine erste, offensichtliche Form der Typeinschränkung liegt dann vor, wenn ein Typ auf Basis eines anderen unter Entfernen von Eigenschaften (Methoden) definiert wird (das Beispiel vom Pinguin als einem Vogel, der nicht fliegen kann, kennen Sie ja bereits aus Abschnitt 2.3.2; das Beispiel vom Quadrat als einem Rechteck, das nur eine Kantenlänge braucht, ist ein anderes). Diese Form der Typeinschränkung stellt zumindest auf Ebene der Typdefinition (der Intensionen) die Umkehrung der Typerweiterung dar. Es liegt auf der Hand, daß diese Form der Typeinschränkung nicht zur Zuweisungskompatibilität führt; dies folgt schon aus der fehlenden Symmetrie der Typkonformität. Sie soll hier deswegen keine weitere Berücksichtigung finden, auch wenn es Sprachen gibt, die sie erlauben (z. B. EIFFEL).

Typeinschränkung durch Spezialisierung

Eine unter dem Gesichtspunkt der Zuweisungskompatibilität interessantere Form der Typeinschränkung besteht darin, die verwendeten Typen einer Typdefinition durch andere, speziellere zu ersetzen (ohne hier schon zu sagen, was „spezieller“ im Zusammenhang mit Typen bedeutet). Diese Form der Typeinschränkung ergibt sich auf natürliche Weise, wenn man sich den Zusammenhang von Extensionen von definierten Typen und solchen, die in Typdefinitionen vorkommen, ansieht.

Das Ganze soll an einem Beispiel verdeutlicht werden. Man denke sich einen Typ `Dokument` wie folgt definiert:

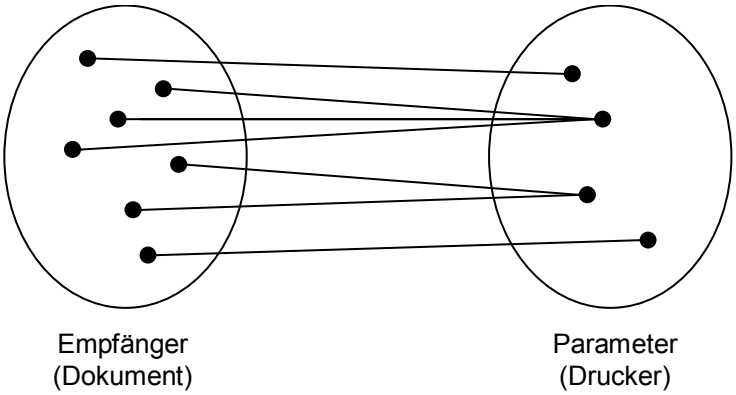
Typ

```
Dokument
```

Protokoll

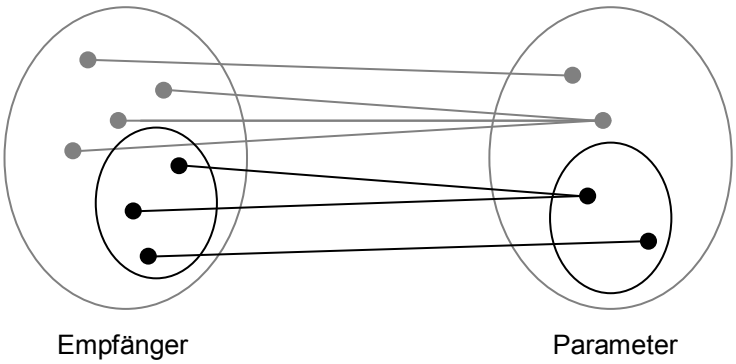
```
972   name ^ <String>
973   name: einString <String> ^ <Self>
974   druckenAuf: einemDrucker <Drucker> ^ <Self>
```

Für die Dauer eines Ausdrucks wird durch die Methode `druckenAuf:` ein konkretes Dokument einem konkreten Drucker zugeordnet. Mengentheoretisch betrachtet ist diese Zuordnung eine Relation zwischen zwei Mengen:



Nun gibt es verschiedene Arten von Dokumenten und Druckern: Man kann z. B. bei Dokumenten zwischen Texten und Diagrammen unterscheiden und bei Druckern zwischen Zeilendruckern und Plottern. Die Extensionen entsprechender Typen sind dann jeweils Teilmengen der Extensionen von `Dokument` und `Drucker`.

Weiterhin ergibt es sich aus der Natur der Sache, daß man Diagramme nur auf Plottern drucken sollte und Texte nur auf Zeilendruckern. Dies geht konform zur obigen Betrachtung eines Methodenaufrufs als Relation: Wenn man die Menge einer Stelle einer Relation wie der obigen auf eine Teilmenge einschränkt, dann schränkt sich dadurch in der Regel auch die Menge der anderen Stelle auf eine Teilmenge ein:



Die andere Menge kann auch gleich bleiben; größer wird sie jedoch nie.

Es ergibt sich daraus die folgende Definition eines Typs `Zeichnung` als Typeinschränkung von `Dokument`:

Typ `Zeichnung`

eingeschränkter Typ `Dokument`

Protokoll

```
975   druckenAuf: einemDrucker <Plotter> ^ <Self>
```

Redefinition

Man beachte, daß die Methode `druckenAuf:` nicht hinzugefügt wurde — sie ersetzt vielmehr die von `Dokument` übernommene. So unterscheidet sich die Methode von der ursprünglichen auch nur in der *Typannotation* des formalen Parameters. Man spricht in diesem Zusammenhang von einer **Redefinition** der Methode (an den ebenfalls dafür verwendeten Begriff des *Überschreibens* sind je nach Programmiersprache andere Bedingungen geknüpft). Die Methoden `name` und `name:` werden übrigens, genau wie bei der Typerweiterung, bei der Typeinschränkung übernommen, solange nichts anderes ausgesagt wird.

Verhältnis von Typeinschränkung und Typerweiterung

Man mag sich fragen, warum bei der Typerweiterung in Abschnitt 3.6 keine zwei Formen analog zur Typeinschränkung eingeführt wurden. Die Typerweiterung würde damit zur vollständigen Umkehrung der Typeinschränkung wie hier beschrieben. Wie Sie noch sehen werden, ist das Ziel nicht die Schaffung zweier Komplementäre, sondern die Vereinigung beider zu *einer* Beziehung zwischen Typen —dazu müssen sie aber in dieselbe und nicht in gegensätzliche Richtungen gehen. Außerdem ist eine Erweiterung des Wertebereichs bei Einschränkung des Definitionsbereichs nicht durch den Begriff der Relation wie oben erklärbar; eine wichtige Analogie zur Realität, die durch Typen zwecks semantischer Prüfung nachgebildet werden soll, ginge damit verloren.

Typeinschränkung und Zuweisungskompatibilität

Nun ergibt sich aber bei der Typeinschränkung auch ohne Löschen das Problem, daß sie die Zuweisungskompatibilität, die ja für die Typerweiterung noch per Typkonformität geregelt werden konnte, aushebelt: Wenn man bei den obigen Typdefinitionen und den Deklarationen

```
976   | d <Dokument> z <Zeichnung> l <Zeilendrucker> |
```

zunächst

```
977   d := z
```

zuweist und dann weiter

```
978   d druckenAuf: l
```

aufruft, dann wäre, Typkonformität von `Zeichnung` und `Dokument` bzw. `Zeilendrucker` und `Drucker` vorausgesetzt, die Typprüfung zwar erfolgreich (denn `druckenAuf:` verlangt für `Dokument` `Drucker` als Argumenttyp), aber zur Laufzeit soll nun eine Zeichnung auf einem Zeilendrucker gedruckt werden, was gemäß der obigen Definition von `Zeichnung` nicht vorgesehen ist. Eine der beiden Zuweisungen, die explizite in Zeile 977 oder die implizite (die Parameterübergabe) in Zeile 978, ist also nicht zulässig. Da gegen die Typkonformität von `Zeilendrucker` und `Drucker` nichts spricht (für beide Typen sind im Beispiel ja gar keine Definitionen angegeben), bleibt nur, daß `Zeichnung` nicht typkonform zu `Dokument` ist, wobei der Grund hierfür in der Einschränkung des Parametertyps von `druckenAuf:` bei der Redefinition zu suchen ist.

Aliasing und Typeinschränkung

Wesentlich für diese Betrachtungsweise, und damit das geschilderte Problem, ist übrigens, daß nach der Zuweisung von Zeile 977 `d` und `z` auf dasselbe Objekt, nämlich eine Zeichnung, verweisen. `d` ist also ein *Alias* für `z` (s. Abschnitt 1.1.6).

Unter *Wertsemantik*, bei der bei der Zuweisung eine Kopie erstellt wird, hätte man hingegen überlegen müssen, wie man ein Objekt vom Typ `Zeichnung` in einer Variable vom Typ `Dokument` speichern kann; je nach interner Repräsentation der Objekte (die ja durch den Typ nicht festgelegt ist), ist dafür nämlich gar nicht genug Platz. Gleichzeitig mit der Kopie könnte dann eine Typkonvertierung erfolgen, bei der aus der Zeichnung ein Dokument gemacht würde (was auch immer das heißen mag). Dieses Dokument müßte dann, per obiger Typdefinition, auch auf einem Zeilendrucker druckbar sein. Es ist allerdings schwer vorstellbar, wie dies umzusetzen ist, wenn das entsprechende Objekt nicht einmal mehr weiß, daß es eine Zeichnung ist, geschweige denn, wie seine interne Repräsentation aussieht. In der Praxis der objektorientierten Programmierung ist daher auch nur die Referenzsemantik in Fragen der Zuweisungskompatibilität interessant.

kein Problem bei Rückgabe

Man beachte übrigens, daß sich bei der Ausgabe aus Methoden (der Rückgabe von Werten) unter Typeinschränkungen kein analoges Problem ergibt: Wenn beispielsweise einem Dokument ein Drucker dauerhaft zugeordnet wird und dieser Drucker mittels einer Methode `drucker` abgefragt werden kann, dann hat die Einschränkung des Rückgabetyps von `drucker` von `Dokument` auf `Plotter` keine negativen Auswirkungen auf die Zuweisungskompatibilität:

```
979   | dr <Drucker> |
980   d := z.
981   dr := d drucker
```

ist völlig in Ordnung, solange nur `Plotter` zuweisungskompatibel mit `Drucker` ist. Die unterschiedliche Zulässigkeit von Typeinschränkungen bei Ein- und Ausgabe wird in Abschnitt 3.9.3 noch genauer beleuchtet.

Löschen als Spezialfall

Was die Freiheit von Typfehlern angeht, kann man das Löschen von Eigenschaften (Methoden) übrigens auch als einen Spezialfall der Typeinschränkung der obigen, zweiten Form auffassen, nämlich einer, in der der Wertebereich auf die leere Menge eingeschränkt wird. So wäre beispielsweise `druckenAuf:` mit einem Parametertyp ohne Elemente gar nicht mehr aufrufbar (da es kein typkorrektes Parameterobjekt gäbe), was einer Löschung gleichkäme.

3.9 Subtyping und Inklusionspolymorphie

Die Einführung von Typäquivalenz und Typkonformität bezog sich bislang lediglich auf das Verhältnis der Typdefinitionen, also der Intensionen der Typen. Die Frage des Zusammenhangs der Wertebereiche der Typen, also der Extensionen, ist dabei unberücksichtigt geblieben. Wenn aber die obige Definition von Typkorrektheit weiter Bestand haben soll, dann müssen die Werte zuweisungskompatibler Typen zum Wertebereich des Typen, an den zugewiesen werden soll, gehören.

Zur Erinnerung: *Typannotationen* stellen *Invarianten* dar, die die möglichen Werte einer Variable beschränken. Diese Invarianten dürfen durch Zuweisungen nicht verletzt werden. Wenn man aber nun Zuweisungen von einem anderen Typen zuläßt, dann wird die Typkorrektheit nur dann nicht verletzt, wenn der Wertebe-

reich des anderen Typen (seine Extension) in dem dessen, dem zugewiesen wird, enthalten (inkludiert) ist. Mit anderen Worten: Damit eine Zuweisung `a := b`, bei der sich die Typen von `a` und `b` unterscheiden, zulässig ist, muß die Extension des Typs von `b` eine Teilmenge der Extension des Typs von `a` sein.

mangelnde Teilmengenbeziehung bei Typerweiterung

Im Fall der Typerweiterung ist dies nicht automatisch der Fall. So handelt es beispielsweise bei der Extension des Typs

```
Typ
                DreiDPunkt
erweiterter Typ
                ZweiDPunkt
Protokoll
982   z ^ <Float>
983   z: zKoordinate <Float> ^ <Self>
```

als Erweiterung von

```
Typ
                ZweiDPunkt
Protokoll
984   x ^ <Float>
985   x: xKoordinate <Float>
986   y ^ <Float>
987   y: yKoordinate <Float>
988   + einZweiDPunkt <Self> ^ <Self>
```

nicht unbedingt um eine Teilmenge der Extension von `ZweiDPunkt`, denn es ist z. B. nicht klar, was das Ergebnis der Addition eines dreidimensionalen zu einem zweidimensionalen Punkt sein könnte — geometrisch ist die Addition zweier Punkte unterschiedlicher Dimensionen jedenfalls nicht definiert.

Selbsttestaufgabe 3.2

Versuchen Sie, das Beispiel mit `ZweiDPunkt` und `DreiDPunkt` so zu retten, daß sowohl Typerweiterung als auch Inklusion von Extensionen darin vorkommt. Evtl. finden Sie in Abschnitt 2.3 nützliche Hinweise.

Herstellung der Teilmengenbeziehung bei Typerweiterung

Das Phänomen der mangelnden Extensionsinklusion bei Typerweiterung läßt sich darauf zurückführen, daß dem erweiterten Typ (im Beispiel `Dokument`) eigene, d. h. nicht einer seiner Erweiterungen entstammende Werte (Objekte) zugestanden werden. Wäre die Extension eines erweiterten Typs als die Vereinigung der Extensionen seiner Subtypen (hier `Text` und `Zeichnung`) definiert, gäbe es dieses Problem nicht. Dies ist ein sehr guter Grund dafür, daß Supertypen — genau wie Generalisierungen (Abschnitt 2.3) — keine eigenen Objekte haben sollten (vgl. a. Abschnitte 3.9.1 und 7.8).

Teilmengenbeziehung bei Typeinschränkung

Auch nicht selbstverständlich ist die Teilmengenbeziehung bei der Typeinschränkung: Durch das Weglassen von Eigenschaften (Methoden) wird die Extension, also die Menge der Werte (Objekte), die darunter fallen, eher größer denn kleiner — je weniger spezifisch die Menge der geforderten Eigenschaften

ist, desto mehr Objekte fallen darunter. Die sich daraus ergebende Teilmengenbeziehung wäre also eher die umgekehrte (die Extension des einschränkenden Typen enthält die des eingeschränkten). Etwas anders sieht es aus, wenn durch Typeinschränkung (*Redefinition*) die Ein- oder Rückgabetypen von Methoden beschränkt werden: Die Menge der Zeichnungen ist eine Teilmenge der Menge der Dokumente, auch weil sich Zeichnungen eben nur auf Plottern ausgeben lassen. Die Zuweisungskompatibilität von `Zeichnung` mit `Dokument` wäre also, was die Inklusion der Extensionen angeht, kein Problem.

Erweiterung und Einschränkung als gleichgerichtete Maßnahmen

Man könnte nun die Typerweiterung unter oben gemachter Einschränkung und die zweite Form der Typeinschränkung als in dieselbe Richtung zielende Maßnahmen ansehen: Beide schränken Extensionen ein. Das läßt sich wie folgt erklären: Wenn man einer Menge von Objekten, die durch eine Anzahl Attribute alle gleichermaßen charakterisiert werden, weitere Attribute beimißt, dann schränkt man diese Menge ein, wenn die hinzugefügten Attribute nicht alle Objekte der Menge charakterisieren. Wenn man beispielsweise wie oben geschehen die Attributmenge des Typs `Dokument` um die Methode `zeilen ^ <Collection>` erweitert, dann fallen die Zeichnungen aus der durch `Dokument` beschriebenen Menge von Objekten heraus, weil sie keine Zeilen haben. Alternativ könnte man auch sagen, daß Dokumente grundsätzlich über Zeilen verfügen können, diese aber bei Zeichnungen immer in der Anzahl 0 vorliegen (also die entsprechende Collection immer leer ist; eine Typeinschränkung!), nur erscheint das weniger natürlich.[61] Man beachte die Parallelität zum Begriff der *Spezialisierung* (Abschnitt 2.3.2): Der durch Typerweiterung oder -einschränkung aus `Dokument` hervorgegangene Typ `Zeichnung` ist spezieller als seine Vorlage.

Problemquelle implizite Allquantifizierung

Nun ergibt sich aber gemäß obigem Beispiel (Zeilen 976–978) ein Sachverhalt, der trotz aller Harmonie von Typerweiterung und -einschränkung nicht weniger als den Verlust der Zuweisungskompatibilität bedeutet. Dieser resultiert jedoch bei genauerer Betrachtung nicht daraus, daß Zeichnungen keine Dokumente wären, sondern aus der mit der Typkorrektheit verbundenen, impliziten Allquantifiziertheit von Typinvarianten: Eine Methodendeklaration

```
989   druckenAuf: einDrucker <Drucker> ^ <Self>
```

im Protokoll eines Typs `Dokument` wird nämlich interpretiert als „`druckenAuf:` ist definiert für alle Empfängerobjekte vom Typ `Dokument` und Parameterobjekte vom Typ `Drucker`“, was aber in dieser Allgemeinheit sachlich falsch ist.

Dependent types

Typsysteme mit Typinvarianten der hier vorgestellten Art sind nicht in der Lage, andere als implizit allquantifizierte Aussagen über Wertebereiche zu treffen. Dies

---

61 Statt dessen würde man eher vermuten, daß es sich um einen Programmierfehler handelt, wenn jemand bei einer Zeichnung auf ihre Zeilen zugreifen will. Außerdem müßte bei erster Annahme der allgemeinste Typ, von dem alle anderen abgeleitet sind (`Object` in STRONGTALK), immer alle Attribute deklarieren, die einem jemals in den Sinn kämen, und das wäre nun wirklich unpraktisch.

ist gewissermaßen der Preis der Einfachheit. Abhilfe schaffen neuere Typsysteme wie die Idee von den *Dependent types*[62] (die aber hier nicht weiter verfolgt werden soll, zumal sie noch in keiner mir bekannten objektorientierten Programmiersprache verwendet wird). Zu den Dependent types sei nur so viel gesagt, daß man sich Parametertypen von Methoden als Funktionen des Typs, zu dem die Methode gehört, vorstellen kann. Der Parametertyp von `druckenAuf:` aus obigem Beispiel wäre dann, in Abhängigkeit davon, ob die Methode auf einem Objekt vom Typ `Dokument` oder `Zeichnung` aufgerufen wird, `Drucker` oder `Plotter`. Wie man sich leicht vorstellen kann, ist die statische Prüfung solcher Bedingungen (Invarianten) aber nicht so einfach.

Die Vereinigung von Typerweiterung und Typeinschränkung mit Zuweisungskompatibilität und der daraus folgenden Typkorrektheit bietet der Begriff des Subtyps.

3.9.1 Der Begriff des Subtyps

Ein **Subtyp** ist als ein Typ definiert, dessen Werte oder Objekte überall da auftauchen dürfen, wo ein Wert des Typs, von dem er ein Subtyp ist, verlangt wird. Subtyp steht dabei nicht für eine besondere Art von Typ, sondern vielmehr für eine Rolle in einer Beziehung zwischen zwei Typen, nämlich der **Subtypenbeziehung**. Die Gegenrolle heißt **Supertyp**.

Zuweisungskompatibilität von Subtypen

Man beachte, daß diese Definition von Subtypen Zuweisungskompatibilität impliziert: Wenn die Objekte eines Subtypen überall da auftauchen dürfen, wo Objekte seines Supertypen erwartet werden, dann dürfen sie auch Werte von Variablen sein, die mit dem Supertypen annotiert (auf Werte des Supertypen beschränkt) sind. Ein Subtyp ist also mit seinem Supertyp per Definition zuweisungskompatibel. Es steckt in dieser Definition aber eine gewisse Zirkularität (Subtyp als Voraussetzung und Ergebnis der Zuweisungskompatibilität), die eine Einfachheit der Zusammenhänge vortäuscht, die es in Wirklichkeit nicht gibt; die eigentliche Frage, was nämlich erfüllt sein muß, damit ein Objekt eines Typen tatsächlich da erscheinen darf, wo ein Objekt eines anderen Typen erwartet wird, bleibt unberücksichtigt. Eine Befassung mit dieser Frage erfolgt hier aber nur insoweit, wie dies heutige Typsysteme auch tatsächlich tun; eine genauere Betrachtung erfolgt dann erst in Abschnitt 6.1.

Subtyphierarchie

Ein Subtyp kann selbst wieder Subtypen haben usw.; man spricht dann auch von einer **Subtypen**- oder einfach nur von einer **Typhierarchie**. In einer solchen Hierarchie kann man **direkte** von **indirekten Subtypen** unterscheiden: Zwischen einem Typ und seinem direkten Subtyp liegt kein weiterer Typ in der Typhierarchie, bei einem indirekten Subtyp hingegen schon. Die Subtypenbeziehung ist transitiv und reflexiv; insbesondere ist also jeder Typ ein Subtyp von sich selbst (das folgt schon aus obiger Definition des Begriffs Subtyp). Die Frage der Sym-

---

[62] DL Shang „Covariant deep subtyping reconsidered“ *SIGPLAN Notices* 30:5 (1995) 21–28.

metrieeigenschaft muß noch bis zum nächsten Unterabschnitt zurückgestellt werden.

mehrere direkte Supertypen

Je nach verwendetem Typsystem kann ein Typ auch mehrere direkte Supertypen haben. Die sich daraus ergebende Struktur ist dann aber keine Hierarchie mehr (im strengen Sinne; man spricht aber dennoch häufig von einer solchen, manchmal auch von einer **Mehrfachhierarchie**), sondern nur noch ein gerichteter azyklischer Graph (engl. directed acyclic graph, kurz DAG). Alle obengenannten Eigenschaften der Subtypenbeziehung bestehen jedoch weiter fort.

Begriff des Subtyping

Wenn Subtypen, ähnlich wie bei der Typerweiterung oder -einschränkung, auf Basis von bereits bestehenden definiert werden, spricht man auch vom (nominalen) **Subtyping** (s. u.). Eine solche Subtypendefinition erfolgt dann immer unter Angabe des oder der direkten Supertypen, und relativ dazu. Dabei verlangt die obige Definition von einem Subtypen einen bestimmten Zusammenhang zwischen den Definitionen (Intensionen) von Sub- und Supertyp: Die Ergänzungen oder Änderungen, die eine Subtypendefinition relativ zu der ihres oder ihrer Supertypen vornimmt, müssen gewährleisten, daß die Werte (Objekte) des Subtyps überall da auftauchen dürfen, wo ein Wert des Supertyps verlangt wird. Dies läßt sich durch folgende einfache Regel ausdrücken:

> Wenn ein Typ Y ein Subtyp eines Typs X ist, dann müssen alle Bedingungen, die für Objekte des Typs X erfüllt sind, auch für Objekte des Typs Y erfüllt sein.

Es darf also insbesondere keine Bedingung, die ein Supertyp an seine Objekte stellt, durch einen Subtypen aufgehoben oder relativiert werden. Logisch gesprochen heißt das, daß die Bedingungen (die Intension) des Subtyps die des Supertyps impliziert. Daraus folgt, daß die Typerweiterung als Basis einer Subtypendefinition infrage kommt (da die Intension des Supertypen unverändert übernommen und lediglich ergänzt wird), die Typeinschränkung hingegen zunächst einmal nicht. Dennoch wäre die Typeinschränkung vom Subtyping auszuschließen eine unnötige Einschränkung, wie Sie gleich noch sehen werden.

Subtypen- als Teilmengenbeziehung

Wenn man die eingangs dieses Abschnitts gemachten Bemerkungen zur Typkorrektheit auf das Subtyping und die damit implizierte Zuweisungskompatibilität überträgt, dann ergibt sich für die Extensionen von Supertypen und Subtypen, daß die Subtypenbeziehung als eine Teilmengenbeziehung gedeutet werden muß: Die Extension eines Subtyps ist in den Extensionen all seiner (direkten und indirekten) Supertypen enthalten. Umgekehrt umfaßt die Extension eines Supertyps die Extensionen all seiner Subtypen. Es ergibt sich, daß nur wenn die Extensionen aller direkten Subtypen eines Typs paarweise disjunkt sind, man es mit einer echten Typhierarchie zu tun hat, in der jeder Typ nur genau einen direkten Supertyp hat. Ist die Extension eines Supertyps genau gleich der Vereinigung der Extensionen seiner Subtypen, hat der Supertyp keine eigenen Werte, also keine Werte, die nicht zugleich Wert eines seiner Subtypen sind. Diese Bedingung ent-

spricht der Idee von der *Generalisierung* aus Abschnitt 2.3.1 und im übrigen gute objektorientierter Praxis (s. Abschnitt 7.8).

3.9.2 Strukturelles und nominales Subtyping

Beim Subtyping unterscheidet man wie bei der Typäquivalenz und -konformität zwischen nominalem und strukturellem Subtyping. Nominales Subtyping liegt vor, wenn ein Subtyp aus einem namentlich erwähnten Supertyp abgeleitet sein muß, um als sein Subtyp zu gelten. Strukturelles Subtyping liegt vor, wenn ein Typ lediglich die obige Definition von Subtyp erfüllen muß, um als solcher zu gelten. Nominales Subtyping impliziert strukturelles; analog zur Typkonformität macht das nominale Subtyping die Subtypenbeziehung antisymmetrisch, das strukturelle hingegen nicht.

3.9.3 Kovarianz und Kontravarianz bei Methodenaufrufen

Daß Typerweiterung als Basis des Subtyping keine technischen Probleme bereitet, sollte hinreichend klar geworden sein: Typfehler sind damit ausgeschossen und es bleibt lediglich das semantische Problem, daß Werte eines Subtyps inhaltlich keine Werte des Supertyps sind (wie im Beispiel von zwei- und dreidimensionalen Punkten). Es bleibt noch die Frage, ob und falls ja in welchem Umfang Typeinschränkung im Rahmen des Subtyping erlaubt ist. Diese Frage soll an einem Beispiel beantwortet werden.

Angenommen, es ist ein Typ `A` wie folgt definiert:

```
Typ                A
Protokoll
990   m: y <Y> ^ <Y>
```

Zuweisungen der Art

```
991   x := a m: z
```

wobei `a` eine Variable vom Typ `A` sei (für ein Objekt vom Typ `A` steht), sind nach den Regeln des Subtyping dann zulässig, wenn die Variable `x` mit einem Supertyp und `z` mit einem Subtyp von `Y` (jeweils einschließlich des Typs `Y` selbst) deklariert ist.

Nun sei weiterhin der Typ `B` wie folgt als Subtyp von `A` abgeleitet:

```
Typ                B
Supertyp           A
Protokoll
992   m: y <Z> ^ <X>
```

Die Methode `m:` wird also in `B` *redefiniert*. Die Frage ist nun, in welchem Verhältnis die dabei verwendeten Typen `X` und `Z` zu `Y` stehen müssen, damit die Zuweisung aus Zeile 991 weiterhin zulässig ist, selbst wenn die Variable `a` auf ein Objekt vom Typ `B` verweist. Mit anderen Worten: Welche Bedingungen sind an die

Parametertypen bei der Redefinition zu stellen, damit eine Zuweisung eines Objekts vom Typ `B` an eine Variable vom Typ `A` in der Folge zu keiner Verletzung einer (anderen) Typinvariante führt? Solche Folgefehler waren ja bereits in Abschnitt 3.8 thematisiert worden.

analytische Betrachtung

Die Antwort läßt sich systematisch herleiten, indem man sich die zu Zeile 991 gehörenden impliziten Zuweisungen genau anschaut. Zuerst wird ja `z` dem formalen Parameter von `m:`, `y`, zugewiesen. Wenn `y` nun in `B` einen anderen Typ als `Y` bekommen soll, dann darf es sich dabei nur um einen handeln, der mehr Werte zuläßt — würde er weniger Werte zulassen, könnte es sein, daß er die Zuweisung von `z` ausschließt, wodurch Zeile 991 zu einem Typfehler (einem typinkorrekten Programm) führen würde. Der Typ des formalen Parameters `y` von `m:` in `B`, `Z`, muß also ein Supertyp dessen in `A`, `Y`, sein. Zuletzt, d. h., nach erfolgter Auswertung des Methodenaufrufs, wird dann das Ergebnis `x` zugewiesen. Wenn nun der Rückgabewert von `m:` in `B`, `X`, einen anderen Typ als in `A`, `Y`, bekommen soll, dann kann dies aufgrund der geforderten Zuweisungskompatibilität nur um einen Typen handeln, der weniger Werte zuläßt, da ja sonst die Zuweisung an `x` zu einem Typfehler führen könnte. Es kommt also als Rückgabetyp für `m:` in `B` nur ein Subtyp von dem in `A`, `Y`, infrage. Es ergibt sich also, daß sich bei einer Redefinition einer Methode die Eingabeparametertypen einer Funktion nur „nach oben“ (also zu einem Supertypen hin), die Ausgabeparameter hingegen nur „nach unten“ (hin zu einem Subtypen) verändern dürfen, wenn die Typkorrektheit eines Programms nicht verletzt werden soll.

Änderung im Verhältnis zur Änderung des Empfängertyps: Kovarianz und Kontravarianz

Nun ändern sich aber bei der Redefinition nicht nur die Parametertypen (Ein- und Aus- bzw. Rückgabe), sondern auch der Typ des Empfängers. Dieser ändert sich bei der Redefinition aber immer nach unten (da der redefinierende Typ ja als Subtyp vom redefinierten abgeleitet wird). Es folgt also, daß die Eingabeparameter zum Empfängertyp gegenläufig variieren müssen, der Ausgabeparameter hingegen gleichgerichtet. Man spricht im ersten Fall daher von einer **Kontravarianz**, im zweiten von einer **Kovarianz**, und sagt:

> Wenn die Eingabeparameter einer redefinierten Methode kontravariant und die Ausgabeparameter kovariant redefiniert werden, dann bleibt Zuweisungskompatibilität des redefinierenden Typen mit dem redefinierten erhalten.

Man spricht im Kontext von Subtyping auch von *Typkonformität des Subtypen mit dem Supertypen*.

Problem bei Identität von Ein- und Ausgabeparametern

Nun enthält die Idee von der Gegenläufigkeit der Veränderung von Parameter- und Ergebnistypen beim Redefinieren einen kleinen Schönheitsfehler: Wenn es sich nämlich bei der Eingabe in eine Funktion und bei ihrer Ausgabe um dasselbe Objekt handelt, kann diesem nicht einmal (bei der Eingabe) ein Supertyp und einmal (bei der Ausgabe) ein Subtyp zugedacht werden, denn der Subtyp verlangt ja mehr Eigenschaften, als der Supertyp garantiert. Wenn also z. B. ein Typ `A` wie

Typ

A

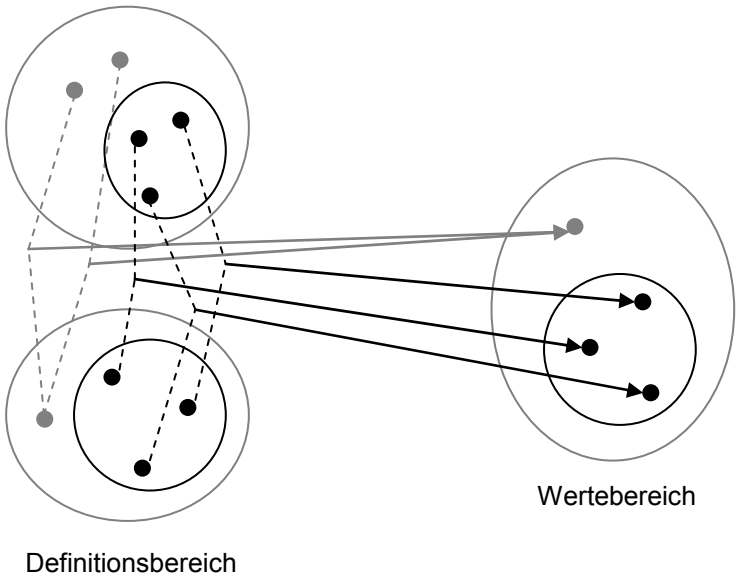
Protokoll

```
993    x ^ <X>
994    x: <X> ^ <Self>
```

definiert ist und `x` das Objekt zurückgeben soll, das mit `x:` geliefert wurde, dann kann ein Subtyp `B` die beiden Methoden schlecht so redefinieren, daß `x:` Objekte eines Supertypen von `X` entgegennimmt, während gleichzeitig `x` Objekte eines Subtypen zurückliefert. Da das Umgekehrte freilich auch nicht geht, bleibt nichts anderes, als die Unveränderlichkeit der Parametertypen, auch als **Invarianz** oder besser (da das dazu passende Adjektiv „invariant" in seiner Bedeutung schon belegt ist) als **Novarianz** bezeichnet, zu verlangen.

Problem des mangelnden Realismus

Obige analytischen Überlegungen führen also, mit der eben gemachten Einschränkung, zur Regel von den kontravarianten Parameter- und den kovarianten Rückgabetypredefinitionen. Es gibt aber noch einen zweiten Ansatz zur Klärung der Frage nach der richtigen Varianz der Parametertypen redefinierter Funktionen, die diese Betrachtung deutlich in Frage stellen. Dazu soll (wie schon in Abschnitt 3.8) die Interpretation von Methoden als Relationen bzw., da hier der Rückgabetyp mit berücksichtigt wird, als Funktionen oder Abbildungen herhalten. Der hier zweistellige Definitionsbereichs (Empfängertyp plus Parametertyp) der Funktion steht dabei stellvertretend für beliebige Stellenzahl, also für Methoden mit beliebig vielen Parametern. Der Wertebereich ist hingegen immer einstellig, da eine Methode stets nur einen Wert zurückgibt.



Wenn man nun die Anzahl der Empfängerobjekte einschränkt (was ja beim Übergang zu einem Subtypen geschieht), dann schrumpft damit nicht nur der Wertebereich der Funktion (wie in Abschnitt 3.8 schon illustriert), sondern auch die Menge der möglichen Eingabewerte (der zweite und alle weiteren Definitionsbereiche), die mit der bereits eingeschränkten Menge der Empfänger gemeinsam auftreten können. Es verhalten sich also nicht nur die Ergebnis-, sondern auch die Parametertypen kovariant.

Dieses Ergebnis ist gewissermaßen frustrierend, da es die soeben hergeleitete Kontravarianzregel für Parametertypen infrage stellt: Was programmiertechnisch möglich und sinnvoll erscheint, hat in der Realität (der Interpretation oder Semantik) keine Bedeutung. Auf der anderen Seite erklärt es aber, warum kontravariante Parameterredefinitionen in der Programmierpraxis nicht benötigt werden.[63] Kovarianz für Parametertypen zuzulassen, so sinnvoll es auch zu sein scheint, erlaubt jedoch typinkorrekte Programme; Sie werden im Kontext der Programmiersprache EIFFEL (Abschnitt 5.3.5) noch ausführlicher auf das Problem und eine mögliche Lösung hingewiesen.

3.9.4 Inklusionspolymorphie

Ein von Christopher Strachey, einem der Urväter der Programmierung als wissenschaftliche Disziplin, eingeführter Begriff ist der der **Polymorphie**. Polymorphie bedeutet allgemein Vielgestaltigkeit und wird vor allem in der Biologie verwendet. In der Programmierung steht er für verschiedene Dinge, die jedoch alle mit Typen zu tun haben.

Unter **Inklusionspolymorphie** (engl. inclusion polymorphism, ein in [Cardelli & Wegner 1985] eingeführter Spezialfall der Polymorphie, der neuerdings auch Subtyppolymorphie oder engl. subtype polymorphism genannt wird) versteht man im wesentlichen dasselbe wie unter Subtyping: Wo Objekte eines Typs erwartet werden, können Objekte anderer Typen erscheinen, weil der erste Typ die anderen subsumiert (inkludiert). Der Begriff ist vor allem in Abgrenzung zum *parametrischen Polymorphismus* (engl. parametric polymorphism, s. Abschnitt 3.12) gebräuchlich; sonst redet man eher von Subtyping.

Attraktivität der Inklusionspolymorphie

Das Interessante an der Inklusionspolymorphie ist, daß sich der Wertebereich von Typen dadurch auf unvorhergesehene Umfänge aufweiten läßt. Dies ist insbesondere für die Weiterentwicklung und Wiederverwendung von Programmen interessant, bei der einfach neue Typen hinzugefügt, die anstelle bereits existierender eingesetzt werden können, ohne daß dazu am Programm sonst etwas geändert werden müßte. Die Regeln einer strengen Typprüfung werden durch Inklusionspolymorphie aufgelockert, ohne an Typsicherheit zu verlieren.

Unzulänglichkeit der einfachen Definition

Insgesamt krankt die Definition des Subtyping und der Inklusionspolymorphie in der objektorientierten Programmierung jedoch daran, daß nicht klar definiert ist, was alles zu verlangen ist, damit ein Objekt eines Typs tatsächlich auch da auftauchen kann, wo ein Objekt eines anderen Typen erwartet wird. Zwar gibt die Regel von Ko- und Kontravarianz eine klare Bedingung vor, aber wie Sie schon gesehen haben, ist diese Bedingung aus praktischen Gründen nicht unumstritten. Dazu kommt, daß die Regel einerseits gar nicht ausreicht, um Ersetzbarkeit zu garantieren, und andererseits zu streng ist (s. Abschnitt 6.1). Da Ersetzbarkeit aber der Definition des Subtypenbegriffs zugrundeliegt, bleibt das ganze

---

63 Wer ein Beispiel weiß oder hat, möge es mir bitte schicken!

schwammig. In diesem Kapitel habe ich mich, den meisten gängigen objektorientierten Programmiersprachen folgend, darauf zurückgezogen, zu garantieren, daß keine Typfehler, also Fehler der Art, daß eine bestimmte, geforderte Eigenschaft (Methode) bei einem Objekt nicht vorhanden ist, auftreten können; alles weitere wird dann in Abschnitt 6.1 behandelt.

3.10 Typumwandlungen

Zuweisungskompatibilität unter Subtyping erlaubt also die Zuweisung von Objekten eines Subtyps an Variablen eines Supertyps. Für die statische Typprüfung ergibt sich daraus kein Problem, weil sichergestellt ist, daß die Subtypen alle Eigenschaften ihrer Supertypen erhalten, so daß keine Typfehler auftreten können. Für die Programmiererin ergibt sich aber manchmal das Problem, daß sie ein Objekt, auf das eine Variable eines Supertyps verweist, wie ein Objekt seines tatsächlichen Typs verwenden möchte, in der Regel, weil sie eine Methode darauf aufrufen möchte, die der Supertyp nicht hat. Genau diesen Methodenaufruf würde die Typprüfung aber zurückweisen.

Für diesen Zweck gibt es die Möglichkeit der **Typumwandlung** (engl. type cast). Eine Typumwandlung ist ein Verfahren, bei dem der vorgefundene Typ eines Ausdrucks (einer Variable oder eines Methodenaufrufs) in einen vorgegebenen konvertiert wird. Mit dem Objekt, für das der Ausdruck steht, passiert dabei gar nichts — es wird lediglich der Compiler (bzw. der Type checker) davon überzeugt, daß der Ausdruck den bei der Umwandlung angegebenen Typ hat. Sollte sich zur Laufzeit herausstellen, daß das nicht der Fall ist, kann ein Laufzeittypsystem — soweit vorhanden — dies bei seiner *dynamischen Typprüfung* bemerken und ggf. einen entsprechenden Fehler melden (vgl. die Anmerkungen dazu in Abschnitt 3.1).

Arten von Typumwandlungen

Typumwandlungen können grundsätzlich in verschiedene Richtungen erfolgen: zu Supertypen, zu Subtypen oder zu solchen, die weder Super- noch Subtyp des Ausgangstyps sind. Man spricht entsprechend von **Up cast**, **Down cast** oder **Cross cast**. Up casts sind immer typsicher, Down casts und Cross casts nicht. Down casts sind relativ häufig; sie kommen vor allem dort vor, wo kein parametrischer Polymorphismus (Abschnitt 3.12) zur Verfügung steht. Cross casts sind eher selten; in der *interfacebasierten Programmierung* (s. u.) stehen sie für einen Rollenwechsel eines Objekts.

Verwendungsempfehlungen

Nun sind Typumwandlungen entweder überflüssig oder unsicher. Man sollte daher versuchen, auf sie zu verzichten. Wo unverzichtbar, sollten Typumwandlungen mit einem Typtest abgesichert werden. Dabei wird zur Laufzeit geprüft, ob das Objekt, für das der typgewandelte Ausdruck steht, auch den gewünschten Typ hat. Ist das nicht der Fall, sollten die Teile des Programms, die den bei der Typumwandlung genannten Typ voraussetzen, nicht ausgeführt werden. Sie werden in späteren Abschnitten zu den einzelnen Programmiersprachen noch Beispiele für diese Praxis zu sehen bekommen.

3.11 Der Zusammenhang von Typen und Klassen

Wenn in diesem Abschnitt bislang ausschließlich von Typen die Rede war und Klassen dabei ignoriert wurden, so hat das gute Gründe: Während eine Klasse die Implementierung ihrer Objekte festlegt, ist eine Typdefinition vollkommen frei von Implementierungsaspekten. Zwar können auch *abstrakte Klassen* (Abschnitt 2.4.3) ausschließlich aus Methodendeklarationen bestehen, also ohne jeden Implementierungsanteil daherkommen, aber auch ihr Zweck ist in der Regel, zumindest eine partielle Implementierung vorzugeben, die anderen Klassen, ihren Subklassen, gemeinsam ist, so daß sie diese erben können: Schließlich drückt die Klassenhierarchie ja eine „genetische" Verwandtschaft aus. Eine Typprüfung soll aber ohne Ansehen der Implementierung stattfinden; sie baut daher auf abstrakte Spezifikationen, eben auf Typen.

Es sind also Typen abstrakte Spezifikationen, die zum einen den Wertebereich von Variablen einschränken und zum anderen das *Protokoll* (den Funktionsumfang) von Objekten angeben. Im Gegensatz dazu sind Klassen Konstrukte, die Objekte als Instanzen zu bilden erlauben und mit Implementierung versehen. Da Objekte aber auch den Wertebereich von Typen ausmachen, stellt sich natürlich die Frage, welcher Art der Zusammenhang zwischen Typen und Klassen ist.

Diese Frage soll anhand der schematischen Klassendefinitionen aus Abschnitt 2.1.2 beantwortet werden. In SMALLTALK ist diese ja stets von der Form[64]

```
Klasse                  <Klasse 1>
Superklasse             <Klasse 2>
benannte Instanzvariablen  <Instanzvariable 1>, …
Instanzmethoden

995   <Instanzmethode 1>: <formaler Parameter 1> …
996   …
```

Vergleich von Klassen- und Typdefinition

Es fällt zunächst auf, daß bestimmte Elemente einer Klassendefinition auch in einer Typdefinition auftauchen. Im einzelnen sind dies

- ein (eindeutiger) Name,
- ein zweiter Name, von dessen dazugehöriger Definition abgeleitet wird sowie
- eine Menge von Methodennamen, jeweils mit einer Anzahl formaler Parameter.

Nun werden in SMALLTALKs Klassendefinitionen anders als bei den Typdefinitionen STRONGTALKs keine Typen verwendet — wie auch, denn in SMALLTALK gibt es ja schließlich keine Typen. Statt dessen findet man aber in SMALLTALK-

64 Klassenvariablen und -methoden können hier unter den Tisch fallen, da diese ja nicht Objekte, sondern Klassen (als Instanzen ihrer Metaklassen) charakterisieren.

Programmen manchmal Namen wie „aString“, „anInteger“ etc. für formale Parameter, die nahelegen, daß der Wert einer Variable Instanz einer bestimmten Klasse sein soll. Überprüft wird das jedoch nicht. In STRONGTALK hingegen ist die Ähnlichkeit von Klassendefinitionen mit Typdefinitionen noch größer: Hier sind auch die formalen Parameter der Methoden in den Klassendefinitionen typisiert (s. Abschnitt 3.3). Man beachte, daß in STRONGTALK, anders als z. B. in JAVA oder C++, Instanzvariablen kein Bestandteil einer Typdefinition sein können (vgl. dazu Fußnote 60).

Nun dient ja ein Typsystem in der objektorientierten Programmierung vor allem der Sicherstellung des Umstands, daß alle von einem Objekt aufgrund des deklarierten Typs der Variable, die es benennt, erwarteten Eigenschaften (Methoden) bei diesem Objekt auch vorhanden sind. Dies ist aber immer dann der Fall, wenn sich die Elemente der Typdefinition in der Klassendefinition des Objekts wiederfinden, die Klassendefinition also mit der Typdefinition gewissermaßen strukturell konform ist, so daß die Zuweisung einer Instanz der Klasse an eine Variable des Typs die Anforderungen der Zuweisungskompatibilität erfüllt. Um einen Compiler diese Zuweisungskompatibilität auf einfachere Weise als die Prüfung der Strukturkonformität, die ja eine *rekursive Expansion* der Typdefinitionen erfordert, feststellen zu lassen, gibt es zwei Möglichkeiten (bei beiden handelt es sich gewissermaßen um Varianten einer Namenskonformität):

1. jede Klasse sagt explizit, mit welchen Typen sie konform ist, oder
2. jede Klasse spezifiziert implizit selbst einen Typ.

Im ersten Fall müßte der Compiler noch prüfen, ob eine Klasse tatsächlich auch über alle Eigenschaften der von ihr genannten Typen verfügt; im zweiten Fall ist das automatisch der Fall, da der Typ ja gewissermaßen aus der Klasse erzeugt wird. Diese zweite Art wird von den allermeisten typisierten, objektorientierten Programmiersprachen bevorzugt, doch auch die erste kommt in populären Sprachen vor: So kann beispielsweise in JAVA und C# jede Klasse angeben, mit Variablen welcher Interface-Typen ihre Instanzen zuweisungskompatibel sein sollen (s. Abschnitte 4.7 und 5.1.4.2). Auch STRONGTALK stellt beide Möglichkeiten zur Verfügung.

3.11.1 Subklassen und Subtypen

Man könnte nun versucht sein, den Zusammenhang von Klassen und Typen auch unter Vererbung bzw. Subtyping beizubehalten und damit zu erwarten, daß eine Instanz einer Subklasse einer Klasse dem Wertebereich des zur Superklasse gehörenden Typs angehört. Das ist jedoch dann nicht der Fall, wenn in der Subklasse Änderungen vorgenommen werden, die eine Typkonformität vom zur Subklasse gehörendem zum zur Superklasse gehörenden Typ aufheben, also z. B. Methoden gelöscht oder inkompatibel redefiniert werden. Die meisten objektorientierten Programmiersprachen verbieten das jedoch, so daß sich die Subklassenbeziehung tatsächlich auf eine parallele Subtypenbeziehung übertragen läßt.

3.11.2 Typen als Schnittstellenspezifikationen von Klassen

Eine Klasse liefert eine Implementierung. Nach gängigen Prinzipien nicht nur der objektorientierten Programmierung sind Implementierungen aber hinter *Schnittstellen* (oder *Interfaces*) zu verbergen: Nur die Elemente einer Klassendefinition, die für Benutzerinnen einer Klasse zu Verwendung gedacht sind, sollen durch die Schnittstelle nach außen getragen werden — der Rest soll verborgen bleiben (das sog. *Geheimnisprinzip*).

Zugriffsmodifikatoren

absolute und relative Schnittstelle

In Programmiersprachen wie JAVA, C++ etc. gibt es spezielle Schlüsselwörter, die einem Element einer Klassendefinition (beispielsweise einer Methode) vorangestellt seine Sichtbarkeit festlegen. Diese sog. *Zugriffsmodifikatoren* (engl. access modifier) legen gemeinsam mit der Klassendefinition, die ihre vollständige Implementierung beinhaltet, auch die Schnittstelle der Klasse fest. Je nach Sprache ist diese Schnittstelle für alle Benutzerinnen der Klasse gleich oder unterscheidet sich nach Lokalität oder anderen Eigenschaften von benutzender und benutzter Klasse. Im ersten Fall könnte man von einer absoluten Schnittstelle sprechen; um sie zu spezifizieren, reicht es, zwischen sichtbar und unsichtbar zu unterscheiden. Im zweiten Fall ist die Schnittstelle relativ.

absolute Schnittstellen als Typen

Eine absolut spezifizierte Schnittstelle einer Klasse kommt, wenn sie wirklich keinerlei Implementierungsgeheimnisse verrät, einem Typ gleich. Sie besteht nämlich nur aus Deklarationen von Methoden. Die gemachte Einschränkung ist notwendig, weil manche Sprachen, so z. B. JAVA und C++, die Instanzvariablen ihrer Objekte in die Schnittstelle der Klassen aufzunehmen erlauben. Mit den Instanzvariablen wird aber die Repräsentation der Objekte nach außen sichtbar, was dem Gedanken des Geheimnisprinzips widerspricht.

Wahrung des Implementationsgeheimnisses

Wenn man nun eine Variable mit einem solchen die Schnittstelle repräsentierenden Typ deklariert und eine Typprüfung erfolgreich durchgeführt hat, dann ist sichergestellt, daß über diese Variable nur auf die Elemente einer Klasse zugegriffen wird, die auch Bestandteil des Interfaces der Klasse sind. Wenn jede Instanz dieser Klasse ausschließlich über typisierte Variablen ansprechbar ist, ist damit die Wahrung des Geheimnisprinzips garantiert. Typen dienen damit einem weiteren Zweck, den man zunächst einmal nicht mit ihnen assoziieren würde, nämlich der Wahrung des Implementationsgeheimnisses/Einhaltung der Schnittstellen durch den Compiler.

Dieser überaus nützliche Zusammenhang zwischen Klassen, ihren Schnittstellen und Typen wurde erst relativ spät, nämlich mit der Programmiersprache JAVA und ihrem Interface-als-Typ-Konzept, so weiterentwickelt, daß eine Klasse verschiedene Schnittstellen anbieten kann, die alle zugleich Typen der Klasse (genauer: Supertypen des der Klassen entsprechenden Typs) sind. Die damit ermöglichte *interfacebasierte Programmierung*, die in Kurs 01853 ausführlich behandelt wird, betrachte ich persönlich als den wichtigsten Beitrag JAVAs zur Disziplin der objektorientierten Programmierung (s. a. Abschnitt 4.12).

3.11.3 Gründe für die Trennung von Typen und Klassen

Nun mögen Sie sich vielleicht fragen, warum Typen und Klassen über so viele Seiten als getrennte Begriffe dargestellt wurden, nur um am Ende zum Schluß zu kommen, daß eine Klassendefinition in der Regel auch als Typdefinition herhält. Nun, erstens ist das nicht in allen Sprachen der Fall und zweitens ist es selbst in den Sprachen, in denen es der Fall zu sein scheint, nicht immer so (s. Fußnote 65). So handelt es sich eher um die Symbiose zweier verschiedener Konzepte, die unterschiedlichen Zwecken dienen, deren strukturelle Ähnlichkeit sich aber durch eine syntaktische Zusammenlegung ausnutzen läßt:

- Klassen dienen der Angabe von Implementierungen und damit als Container von ausführbarem Code;
- Typen dienen der Formulierung von Invarianten, die für Variablenbelegungen gelten müssen und deren Verletzung auf einen (logischen oder semantischen) Programmierfehler hinweist.
- Da beide im wesentlichen über die gleichen Elemente verfügen, läßt sich die Definition beider in einem Sprachkonstrukt zusammenfassen.

Unterschiede zur Laufzeit

Der Unterschied der beiden Konzepte Klasse und Typ manifestiert sich auch darin, welche Rolle sie zur Laufzeit eines Programms spielen: Typinformation beeinflußt die Ausführung eines laufenden Programms insofern, als sie ein Programm bei Verletzung einer Invariante abbrechen läßt (durch einen dynamischen Typtest) und damit einem anderen, schwieriger zuordenbaren Fehler zuvorkommt. Klasseninformation beeinflußt die Ausführung des laufenden Programms insofern, als sie Grundlage des dynamischen Bindens ist und in einem Programm als Eigenschaft von Objekten abgefragt werden kann. In Sprachen, in denen jede Klasse einen Typ definiert, ist diese Unterscheidung jedoch nicht immer klar getroffen und wird deswegen von Programmiererinnen auch nicht unbedingt wahrgenommen.

3.12 Generische Typen oder parametrischer Polymorphismus

Typen beschränken die Wertebereiche von Variablen und Methoden. Inklusionspolymorphie lockert diese Beschränkung insofern, als dadurch Wertebereiche von Typen um die von Subtypen erweitert werden können, selbst wenn diese Subtypen zum Zeitpunkt der Typdefinition noch gar nicht bekannt waren (Abschnitt 3.9.4). Nun ist Inklusionspolymorphie nicht die einzige Möglichkeit, den Wertebereich eines Typs variabel zu halten, ohne die statische Typprüfung aufgeben zu müssen. Eine andere ist, einen Typ mit einem oder mehreren anderen zu parametrisieren.

parametrische Typdefinition

Instanziierung

Eine **parametrische Typdefinition** unterscheidet sich von einer normalen dadurch, daß in der Typdefinition verwendete, andere Typen nicht genannt (referenziert) werden müssen, sondern durch Platzhalter, die Typparameter, vertreten werden können. Diese Platzhalter sind Variablen, deren Wert implizit (also ohne entsprechende Deklaration) auf Typen beschränkt ist; man nennt sie auch **Typvariablen**. Diese Typvariablen werden erst bei der Verwendung eines parametrisierten Typs in der Deklaration eines anderen Programmelements mit ei-

nem Wert, also einem Typ, belegt. Man spricht bei dieser Wertzuweisung an eine Typvariable von einer **Instanziierung des parametrischen Typs**; erst bei ihr entsteht ein konkreter Wertebereich, der dann dem deklarierten Programmelement zugeordnet wird. Insbesondere hat ein parametrischer Typ, bei dem Typvariablen nicht belegt sind, keinen konkreten Wertebereich. Dieser Umstand ist bei der Betrachtung von *Zuweisungskompatibilität unter parametrischem Polymorphismus* wichtig.

generische Typen

Die Idee des **parametrischen Polymorphismus** ist, aus einer Typdefinition durch Parametrisierung viele zu machen. Eine parametrische Typdefinition steht also nicht für einen Typ, sondern für (theoretisch) beliebig viele — sie erlaubt es gewissermaßen, Typen nach Bedarf zu generieren.[65] Wohl deswegen bezeichnet man parametrische Typen (Typdefinitionen) auch als **generische Typen** (Typdefinitionen) oder kurz als **Generics**. Wie eben schon erwähnt, wird der Wertebereich bei einer solchen Typgeneration jeweils mitgeneriert.

formale und aktuelle Typparameter

Es erfolgt also die Zuweisung eines Typs zu einer Typvariable bei der Verwendung eines parametrisch definierten Typs in einer Deklaration, beispielsweise der Deklaration einer Variable oder des Rückgabewerts einer Methode. Oberflächlich betrachtet entspricht diese Verwendung in etwa dem Aufruf einer (ja auch an einer anderen Stelle definierten) Methode oder besser (und schon aufgrund der Verwendung des Begriffs Instanziierung) eines Konstruktors; deswegen nennt man die Typvariablen, die in parametrischen Typdefinitionen vorkommen, auch **formale Typparameter** und die konkreten Typen, die bei der Verwendung des Typen in Deklarationen in die formalen Parameter eingesetzt werden, auch **aktuelle Typparameter**. Trotz dieser Analogie zu Methoden- bzw. Konstruktoraufrufen muß man sich immer vor Augen halten, daß die Verwendung eines parametrisch definierten Typs bereits zur Übersetzungszeit zu einer Zuweisung an die Typvariablen führt, man es also keineswegs mit etwas Dynamischem zu tun hat. Insbesondere müssen Typen keine Objekte sein, um Typvariablen zugewiesen werden zu können.

3.12.1 Einfacher parametrischer Polymorphismus

Ein einfaches Beispiel für eine generische Typdefinition in STRONGTALK ist das folgende:

```
Typ                 A

Typvariablen        T

Protokoll

997   x ^ <T>
998   x: einT <T> ^ <Self>
```

[65] Entsprechend gehören, einen Zusammenhang von Klassen und Typen wie in Abschnitt 3.11 beschrieben vorausgesetzt, zu einer parametrischen Klassendefinition beliebig viele Typen.

`T` ist dabei eine Typvariable. Beim Vorkommen von `T` im Abschnitt „Typvariablen“ handelt es sich um ihre Deklaration (Vereinbarung); beim Vorkommen im Abschnitt „Protokoll“ um ihre Verwendung.

Das für den Tatbestand der Parametrisierung wichtige an dieser Typdefinition ist, daß `x:` anstelle des Parameter- und `x` anstelle des Rückgabetyps `T` nennt, wobei `T` eben kein Typ, sondern eine Typvariable ist. Für Typvariablen verwendet man traditionell einzelne Großbuchstaben; dies hat den nützlichen Nebeneffekt, daß man durch eine Typvariable keinen tatsächlichen Typen verdeckt, wie es sonst versehentlich passieren könnte: Man könnte die Typvariable nämlich auch beispielsweise „Integer“ nennen, aber sie wäre deswegen immer noch eine Variable und der Typ `Integer` wäre innerhalb der Typdefinition nicht mehr sichtbar.

beispielhafte Instanziierung

Wenn man nun den Typ `A` verwenden, also z. B. eine temporäre Variable vom Typ `A` deklarieren möchte, muß man sich festlegen, welchen Wert die Typvariable `T` in der Typdefinition und damit welchen Typ die Rückgabe von `x` und die Eingabe von `x:` haben sollen. Soll `T` beispielsweise den Wert `Integer` bekommen, dann schreibt man

```
999   | a <A[Integer]> |
```

und *instanziiert* dabei den parametrischen Typen. `Integer` ist dabei der aktuelle Typparameter (eine Typkonstante, wenn man so will), der in STRONGTALK in ekkige Klammern gesetzt wird. Er wird für diese Verwendung des parametrischen Typs (und nur für diese) in den formalen Typparameter (die Typvariable) eingesetzt. Der Typ von `a`, `A[Integer]`, wird damit zu

```
Typ
                A[Integer]
Protokoll
1000  x ^ <Integer>
1001  x: einInteger <Integer> ^ <Self>
```

definiert, wobei hier `A[Integer]` der (generische) Name des Typen ist. Diese Typdefinition wird jedoch nirgends hingeschrieben — sie ergibt sich immer neu aus der Instanziierung der parametrischen Typdefinition mit einem konkreten Typen. Es sind dann bei obiger Deklaration von `a` die Methodenaufrufe

```
1002  a x: 12
1003  a x + a x
```

zulässig,

```
1004  a x: 'mal sehen, was passiert'
1005  a x, a x
```

hingegen nicht. Für letztere wäre eine Typdeklaration

```
1006  | a <A[String]> |
```

notwendig gewesen, die natürlich auch möglich ist.

Ein und dieselbe parametrische Typdefinition kann in einem Programm beliebig oft verwendet werden, selbst in derselben Deklaration:

mangelnde Zuweisungskompatibilität verschiedener Typinstanzen

```
1007  | a <A[Integer]> b <A[String]> c <A[Boolean]> |
```

gibt `a`, `b` und `c` jeweils verschiedene Typen, die jedoch alle Instanzen der parametrischen Definition von `A` sind. Dennoch sind `a`, `b` und `c` wechselseitig nicht zuweisungskompatibel; sie haben tatsächlich verschiedene Typen. `a` ist jedoch mit `d` wie in

```
1008  | d <A[Integer]> |
```

deklariert zuweisungskompatibel und umgekehrt, da beide denselben Typen haben.

3.12.2 Collections als Standardanwendungsfall für parametrischen Polymorphismus

Eine wichtige Gruppe von Klassen, die Sie in den letzten beiden Kapiteln kennengelernt haben, sind die sog. Collection-Klassen. Auch diese bilden jeweils einen Typ, so daß Variablen, die auf eine Collection verweisen, mit diesem Typ deklariert werden können.

Problemstellung

Nun dienen Collections ja u. a. dem Zweck, :*n*-Beziehungen zwischen einem Objekt und mehreren anderen zu ermöglichen, indem sie dafür Zwischenobjekte zur Verfügung stellen (s. Abschnitt 2.7). Und so bilden die mit den Collection-Klassen assoziierten Typen auch nur die Typen für die Zwischenobjekte. Was man jedoch eigentlich bei der Deklaration von *n*-wertigen Attributen angeben (deklarieren) möchte, ist der Typ der in Beziehung stehenden Objekte.

Angabe des Elementtypen einer Collection

Wenn das Attribut beispielsweise `kinder` heißt und man damit eine Person mit einer Menge anderer Objekte vom Typ `Person`, den Kindern, in Beziehung setzen möchte, dann nutzt es nichts, wenn man `kinder` vom Typ `Person` deklariert — es könnte dann höchstens eine Person enthalten und nicht mehrere. Was man vielmehr gern hätte, wäre etwas, das dem Array-Typkonstruktor `array [<Bereich>] of <Elementtyp>` (spitze Klammern hier wieder als Begrenzer von metasyntaktischen Variablen) von PASCAL gleicht: Im gegebenen Beispiel würde man gern deklarieren, daß `kinder` den Typ `Collection of Person` haben soll. Genau das tut

```
Klasse
                    Person
benannte Instanzvariablen
                    kinder <Collection[Person]>
Instanzmethoden
1009  ...
```

Deklaration einer Collection mit variablen Elementtypen

Passend dazu ist es möglich, `Collection` in STRONGTALK als parametrischen Typ wie folgt zu definieren:

```
Typ
                Collection
Typvariablen
                E
Protokoll
1010  at: einIndex <Integer> ^ <E>
1011  at: einIndex <Integer> put: einElement <E> ^ <Self>
```

Das Programmfragment

```
1012  | p <Person> |
1013  p := Person new.
1014  p kinder at: 1 put: Person new.
1015  p kinder at: 2 put: Person new.
1016  (p kinder at: 1) kinder at: 1 put: Person new
```

ist demnach typkorrekt (und weist `p` zwei Kinder und ein Enkelkind zu).

Ein anderes Beispiel für eine parametrische Definition einer Collection ist `Dictionary`: Hier sollte nicht nur der Element-, sondern auch der Schlüsseltyp variabel gehalten werden. Eine entsprechende parametrische Typdefinition, diesmal mit zwei Typparametern, kann wie folgt aussehen:

```
Typ
                Dictionary
Typvariablen
                S E
Supertyp
                Collection[E]
Protokoll
1017  at: einSchlüssel <S> ^ <E>
1018  at: einSchlüssel <S> put: einElement <E> ^ <Self>
```

Dabei ist der parametrische Typ `Dictionary` ein Subtyp des ebenfalls parametrischen Typs `Collection`. Man beachte, daß der Typparameter `E` hier bereits in der Supertypdeklaration verwendet wird. Ein Dictionary, in dem Integer auf beliebige Objekte abgebildet werden, erhält man dann durch die Instanziierung `Dictionary[Integer, Object]`. Es ist mit einer Variable vom Typ `Collection[Object]` *zuweisungskompatibel*. Auf die Einzelheiten des Subtypings bei parametrischen Typen wird in Abschnitt 3.12.3 eingegangen.

3.12.3 Parametrischer Polymorphismus und Inklusionspolymorphie

Nun war die Speicherung von Personen in Collections, wie sie oben benötigt wurde, auch schon ohne den parametrischen Polymorphismus möglich, nämlich per Inklusionspolymorphie (Subtyping). So würde es zunächst ausreichen, wenn `Collection` wie folgt definiert wäre:

```
Typ
                Collection
Protokoll
1019  at: einIndex <Integer> ^ <Object>
1020  at: einIndex <Integer> put: einObjekt <Object> ^ <Self>
```

Notwendigkeit von Down casts

An die Stelle der Typvariable `E` tritt also der (konkrete) Typ `Object`. Da in STRONGTALK alle Typen Subtypen von `Object` sind, kann man jedes beliebige

Objekt in einer solchen Collection speichern. In der Klasse `Person`, die `Collection` verwendet, würde dann `kinder` schlicht als vom Typ `Collection` (ohne Typparameter) deklariert. Das obige Programmfragment (Zeilen 1012–1016) könnte dann auch beinahe so bleiben, bis auf eine kleine Ausnahme: Zeile 1016 enthält jetzt einen Typfehler, da das Ergebnis von `p kinder at: 1` vom Typ `Object` ist und das Protokoll von `Object` keine Methode `kinder` unterstützt. Es wäre also erst noch eine Typumwandlung von `Object` nach `Person`, ein *Down cast* (s. Abschnitt 3.10), vonnöten. Deren Zulässigkeit ist aber davon abhängig, was wirklich in der Collection drinsteckt, und das kann der Compiler nicht (oder nur sehr aufwendig) feststellen. Die Lösung, die Inklusionspolymorphie bietet, beinhaltet also eine Sicherheitslücke in der statischen Typprüfung, die der parametrische Polymorphismus behebt.

Unzulänglichkeit des einfachen parametrischen Polymorphismus

Nun ist aber auch der parametrische Polymorphismus nicht ohne Makel. Zum einen wäre es ohne Inklusionspolymorphie nicht möglich, in einer Collection mit Elementtyp `XYZ` auch Objekte eines Subtyps von `XYZ` zu speichern. Solche *heterogenen Collections* kommen aber in der Praxis immer wieder vor, so daß man selbst bei Verwendung einer parametrischen Definition von Collections nicht auf Inklusionspolymorphie verzichten wird. Zum anderen wird die erhöhte Typsicherheit bei der Verwendung von parametrisch definierten Typen (wo man ja zumindest bei homogener, also ohne Ausnutzung der Inklusionspolymorphie, Belegung der mit einem Typparameter typisierten Variablen ohne Typumwandlungen auskommt) mit einer geringeren Typsicherheit innerhalb der Typdefinition (bzw. Klassendefinition) selbst erkauft. Dies verlangt nach Erklärung.

Stellen Sie sich einen Collection-Typ `MyCollection` vor, dessen Werte solche Collections sein sollen, deren Elemente sortiert und summiert werden können. Dieser Typ sei ein Subtyp von `Collection` und verfüge weiterhin über entsprechende Methoden `sortieren` und `summieren`:

Typ	`MyCollection`
Typvariablen	`E`
Supertyp	`Collection[E]`
Protokoll	

```
1021  sortieren ^ <Self>
1022  summieren ^ <Number>
```

Intuitiv verlangt die Sortierbarkeit der Objekte vom Typ `MyCollection`, daß auf den Elementen eine Vergleichsfunktion definiert ist. Dies ist aber nicht für alle Typen und somit auch nicht für alle möglichen Belegungen der Typvariable `E` der Fall. Auch verlangt die Methode `summieren`, daß sich aus den Elementen einer solchen Collection ein Wert aggregieren läßt, der vom Typ `Number` oder einem Subtypen davon ist. Man kann daraus schließen, daß die Elemente ebenfalls vom Typ `Number` sein oder zumindest Methoden besitzen müssen, die einen solchen Wert zurückliefern. Und so würde auch eine Implementierung der Methode `summieren` in etwa wie folgt aussehen:

```
1023  summieren ^ <Number>
```

```
1024        ^ elements
1025            inject: 0
1026            into: [ :summe <Number> :element <Number> | summe +
   element].
```

Das aber verlangt, daß der Elementtyp von `MyCollection` `Number` oder ein Subtyp davon sein muß, da sonst die Zuweisung an den formalen Blockparameter `element` nicht zulässig wäre. Insbesondere würde das Codefragment

```
1027  | liste <MyCollection[String]> |
1028  …
1029  liste summiere
```

zu einem Typfehler führen, weil in Zeile 1026 einer Variable vom Typ `Number` ein Objekt vom Typ `String` zugewiesen wird. Nun kann aber die Definition des parametrischen Typs `MyCollection` nicht wissen, wie sie hinterher verwendet wird, und wenn eine Addition durchgeführt werden soll, ist sie darauf angewiesen, daß sie nur mit Typen von addierbaren Objekten instanziiert wird. Es wird also die erhöhte Typsicherheit außerhalb der Typdefinition, nämlich bei ihrer Verwendung, durch eine verminderte Typsicherheit innerhalb erkauft.

mehr Ausdrucksstärke benötigt

Was man gerne hätte, um diesen Mangel zu beheben, wäre die Sicherheit, daß alle Typen, die für `E` eingesetzt werden können, bestimmte Eigenschaften haben, im gegebenen Beispiel, daß sie sortierbar und addierbar sind. Entsprechend sollte ein Typfehler nicht erst in Zeile 1026 moniert werden, sondern bereits an der Stelle, an der die unzulässige Wertzuweisung an die Typvariable stattfindet, nämlich bei der Verwendung (der *Instanziierung*) der parametrischen Typdefinition in der Deklaration von Zeile 1027. Genau das erlaubt der beschränkte parametrische Polymorphismus, der im nächsten Unterabschnitt behandelt wird. Zunächst jedoch noch zu einem anderen wichtigen Aspekt von parametrischem Polymorphismus und Subtyping.

Subtypenbeziehung von aktuellen Typparametern überträgt sich nicht auf Instanzen parametrischer Typen

Unter den Typdefinitionen

Typ	A

Typ	B
Supertyp	A

sowie

Typ	G
Typvariablen	T
Protokoll	

```
1030  x: <T> ^ <Self>
1031  x ^ <T>
```

und den Variablendeklarationen

```
1032  | a <A> b <B> ga <G[A]> gb <G[B]> |
```

ist die Zuweisung

```
1033  a := b
```

sicher zulässig. Nun könnte man annehmen, dasselbe sei auch für

```
1034  ga := gb
```

der Fall. Dahinter verbirgt sich aber die Frage, ob `G[B]` ein Subtyp von `G[A]` ist, ob sich also die Subtypenbeziehung von `B` zu `A` auf entsprechende Typinstanzen vom selben parametrischen Typ überträgt. Intuitiv scheint dies der Fall, zumal beispielsweise

```
1035  ga x: b
```

oder

```
1036  ga x: gb x
```

aufgrund der Subtypenbeziehung von `B` zu `A` kein Problem darstellen sollte (aus Sicht der Typprüfung entspricht dies ja den Verhältnissen der Zuweisung aus Zeile 1033). Nun ist aber nach den Regeln der statischen Typprüfung auch

```
1037  ga x: a
```

erlaubt. Da `ga` aber nach der Zuweisung aus Zeile 1034 lediglich ein Alias für das von `gb` bezeichnete Objekt ist, `ga` also auf ein Objekt vom Typ `G[B]` verweist und dieser als Parametertyp von `x:` nur `B` zuläßt, handelt es sich bei obigem Methodenaufruf um eine Typverletzung. Der Fehler liegt jedoch nicht im Methodenaufruf, der in der Tat typkorrekt ist, sondern vielmehr in der Zuweisung aus Zeile 1034: `G[B]` ist eben kein Subtyp von `G[A]`, nur weil `B` ein Subtyp von `A` ist (man beachte die Parallelität zu dem in Abschnitt 3.8 beschriebenen Problem). Dieser Trugschluß ist einer der häufigsten Anfängerfehler.

Selbsttestaufgabe 3.3

Prüfen Sie nach demselben Schema wie oben, ob `G[A]` ein Subtyp von `G[B]` sein darf.

Merken Sie sich also unbedingt, daß parametrischer Polymorphismus die Subtypenbeziehung seiner aktuellen Typparameter nicht auf die durch Instanziierung erzeugten Typen überträgt. Dies wird auch in Abschnitt 3.12.5 noch eine besondere Rolle spielen. Vor diesem Hintergrund beinahe paradox erscheint, daß sich Subtyping jedoch dazu einsetzen läßt, das oben beschriebene Problem mit der „inneren Typsicherheit“ von parametrisch definierten Typen zu lösen.

3.12.4 Beschränkter[66] parametrischer Polymorphismus

Obiges Beispiel hat gezeigt, daß die einfache Form des parametrischen Polymorphismus für Typsicherheit in der objektorientierten Programmierung nur teilweise nützlich ist: Da die Typvariablen selbst nicht typisiert sind, kann man innerhalb der Typdefinition (und der den Typ implementierenden Klassen) keine Aussagen über den Typ machen. Außerhalb, bei der Verwendung (Instanziierung) der Typdefinition, geht das schon, da hier die Typvariable durch einen Typ ersetzt ist.

wertbeschränkte Typvariablen

Was man also gern hätte, ist, daß die Typvariable innerhalb der mit ihr parametrisierten Typdefinition selbst wertbeschränkt ist, und zwar derart, daß man bei den als Werte zulässigen Typen ein bestimmtes, benötigtes Protokoll voraussetzen kann. Die aktuellen Typparameter sind dann nicht mehr beliebig zu wählen, sondern nur noch aus solchen Typen, die die Einschränkungen erfüllen. Eine Möglichkeit, das zu erzielen, wäre, Metatypen einzuführen, deren Wertebereiche Typen mit durch die Metatypen vorgegebenen Eigenschaften sind. Diese Möglichkeit wird jedoch in der Praxis nicht genutzt.

Supertypen als Schranken

Statt dessen verwendet man eine Art der Beschränkung des Wertebereichs von Typvariablen, die auf Subtyping beruht. Wenn man nämlich erzwingen kann, daß ein aktueller Typparameter (also der Wert der Typvariable) Subtyp eines bestimmten Typs ist, der die benötigten Eigenschaften (Methoden) umfaßt, dann ist damit alles erreicht, was man benötigt: Aufgrund der Regeln des Subtyping hat jeder solche Typ die Eigenschaften des Supertyps (s. Abschnitt 3.9).

Beispiel einer beschränkten parametrischen Typdefinition

Ein solchermaßen durch einen Supertyp beschränkte parametrische Typdefinition ist die folgende:

Der Rest der Definition geht wie oben. Der Ausdruck `E < Number` im Abschnitt

```
Typ           MyCollection
Typvariablen  E < Number
```

„Typvariablen“ ist Deklaration und Beschränkung zugleich; die Beschränkung ist aber wie gesagt keine Typisierung wie in normalen Variablendeklarationen. Sie drückt vielmehr aus, daß die Typen, die als Werte für `E` eingesetzt werden dürfen, Subtypen von `Number` sein müssen. Die Deklaration aus Zeile 1027 wird da-

---

66 Anders als manch Kollege übersetze ich das englische „bounded polymorphism”, manchmal auch „constrained polymorphism” genannt, nicht mit „gebundenem Polymorphismus“ (was dann ja auch „bound polymorphism” heißen müßte). Tatsächlich bezieht sich „ungebundener generischer Typ“ (engl. unbound generic type) manchmal auf die generische Typdefinition, und „Bindung“ auf die „Instanziierung“ der Typparameter mit aktuellen Typen, also auf die Verwendung der Typdefinition. Gemeint ist aber wohl immer dasselbe und die landläufigen Bedeutungen von „beschränkt“ und „gebunden“ unterscheiden sich ja auch nicht großartig.

mit unzulässig und führt zu einem entsprechenden Typfehler während der statischen Typprüfung; die Deklaration

```
1038  | liste <MyCollection[Integer]> |
```

ist hingegen OK.

3.12.5 Rekursiv beschränkter parametrischer Polymorphismus

Rekursive Typen sind Typen, die sich in ihrer Definition selbst referenzieren. Ein Beispiel für einen rekursiven Typ hatten Sie oben schon kennengelernt: Der (zur Klasse `Person` gehörende) Typ `Person` hat Methoden, die `Person` als Parameter- bzw. Rückgabetypen haben. Rekursive Typen sind ein wichtiges Instrument der Programmierung: Ohne sie wären dynamische Strukturen wie beispielsweise verzeigerte Listen oder Bäume kaum möglich. Rekursive Typen machen aber auch bestimmte Probleme — so ist beispielsweise die strukturelle Äquivalenz zweier rekursiver Typen nicht so leicht festzustellen, da die dazu notwendige *Expansion rekursiver Typen* (also das Einsetzen der Struktur für jeden darin vorkommenden Typnamen; vgl. Abschnitt 3.5.1) unendlich große Definitionen ergibt.

parametrische polymorphe Definition binärer Methoden

Es ergibt sich nun ein weiteres Problem, wenn man in einer parametrischen Typdefinition den Typ eines Methodenarguments (eines formalen Parameters einer Methode) variabel halten möchte, dieser Typ aber ausgerechnet der definierte ist (eine *binäre Methode*; vgl. Fußnote 24 in Abschnitt 1.6). So möchte man beispielsweise den Test auf Gleichheit so definieren, daß das Objekt, das gleich sein soll, vom selben Typ sein muß wie das, mit dem man Gleichheit feststellen möchte. Für den Typ `Object` schreibt man dazu einfach

```
Typ
                    Object
Protokoll
1039  = einObjekt <Object> ^ <Boolean>
1040  …
```

für den Typ `Number`

```
Typ
                    Number
Protokoll
1041  = eineZahl <Number> ^ <Boolean>
1042  …
```

usw. Nun ist aber `Number` ein Subtyp von `Object`, so daß man die Deklaration von = eigentlich aus `Object` übernehmen könnte — wenn der Typ des Parameter automatisch so angepaßt würde, daß er dem definierten Typ entspricht. In einem ersten Ansatz wäre man vielleicht versucht, den Gleichheitstest in `Object` einfach als `= einObject <Self> ^ <Boolean>` zu deklarieren, aber das würde,

wenn die *Pseudo-Typvariable* `Self` beim Subtyping jeweils den Subtyp annehmen soll, zu einer kovarianten Redefinition mit den bereits bekannten Problemen führen.[67] Auch hier bietet parametrischer Polymorphismus eine Alternative, wenn auch nicht ganz so, wie vielleicht erwartet.

Lösung durch Rekursion

Man ersetzt dazu zunächst den Typ des Parameters durch eine Typvariable `T`. Nun kann man schlecht

```
Typ                 T
Typvariablen        T
Protokoll
1043  = einT <T> ^ <Boolean>
```

schreiben, da der Typ dann keinen Namen hätte und somit auch nicht verwendbar (referenzierbar) wäre. Was man aber sehr wohl machen kann, ist, einen allgemeinen parametrischen Typ zu definieren, der nur dem Zweck des Gleichheitstests dient und der den Parametertyp des Tests variabel hält, wie in

```
Typ                 Equatable
Typvariablen        T
Protokoll
1044  = einT <T> ^ <Boolean>
```

Man kann dann die gewünschte Rekursion indirekt, nämlich per Definition eines nicht parametrischen Typs als Subtyp des parametrisierten Typs `Equatable` herstellen, wobei man den zu definierenden Typ gleichzeitig als aktuellen Typparameter einsetzt. So liefert z. B.

```
Typ                 Integer
Supertyp            Equatable[Integer]
```

eine Methode mit der Signatur `= einT <Integer> ^ <Boolean>` im Protokoll von `Integer`. Allerdings kann man so nicht erzwingen, daß bei der Definition des Typs `Integer` oben genau `Integer` als aktueller Typparameter eingesetzt wird; es hätte auch jeder andere Typ, z. B. `String`, sein können — der Gleichheitstest wäre dann mit `= einT <String> ^ <Boolean>` falsch deklariert.

Genau diese Beschränkung des aktuellen Typparameters kann man nun mit einer stilistischen Figur erreichen, die vermutlich manch einer von Ihnen erhebliche Kopfschmerzen bereiten wird (zumindest macht sie das mir immer wieder aufs neue): Man beschränkt den formalen Typparameter `T` von `Equatable` auf einen

---

[67] Genau das macht übrigens auch EIFFELS `like Current` (s. Abschnitt 5.3.5.2).

Subtyp von `Equatable[T]`, wobei das Vorkommen von `T` in `Equatable[T]` eine Verwendung der gerade erst eingeführten Typvariable `T` darstellt.

```
Typ            Equatable
Typvariablen   T < Equatable[T]
Protokoll
1045  = einT <T> ^ <Boolean>
```

verlangt also im obigen Beispiel der Typdefinition von `Integer` als Subtyp einer Instanziierung der parametrischen Definition von `Equatable`, daß der aktuelle Typparameter `Integer` ein Subtyp von `Equatable[Integer]` sein muß. Genau das sagt aber die obige Typdefinition von `Integer` aus! Stünde dort `Equatable[String]` oder irgend etwas anderes als Typschranke, wäre dies nicht mehr der Fall (s. Abschnitt 3.12.3) und die Definition von `Integer` verursachte einen statisch feststellbaren Typfehler.

ein paar Worte zum Trost

Wenn Sie hier ein Verständnisproblem haben, trösten Sie sich — es dauert eine Weile, bis man es verstanden hat, und noch länger, bis solche Figuren zum aktiven Repertoire gehören. Gleichwohl sollten Sie sich damit befassen: Das JAVA-Collections-Framework in der Version von JAVA 5 ist voll solcher Typdefinitionen, nicht weil sie schön sind, sondern weil man sie braucht, um das Framework typsicher zu machen, ohne seine Flexibilität zu opfern. Auch Sie werden, wenn Sie objektorientiert programmieren, über kurz oder lang solche Konstrukte von sich geben müssen.

3.13 Parametrischer Polymorphismus und das Kovarianzproblem

In gewisser Weise hat man es beim rekursiv beschränkten parametrischen Polymorphismus wie oben vorgestellt mit einem Fall von kovarianter Redefinition zu tun: Der Parametertyp der Methode = ändert sich mit dem Empfängertyp. Allerdings ergibt sich daraus, anders als bei der Verwendung von `Self` als Typvariable, kein Widerspruch zur Kontravarianzregel des Subtyping, denn `Integer` wird dadurch unmittelbar ja lediglich zu einem Subtyp von `Equatable[Integer]` und nicht etwa von `Equatable[Object]`. Tatsächlich sind `Equatable[Integer]` und `Equatable[Object]` ja zwei vollkommen verschiedene Typen (mit disjunkten Wertebereichen) und `Equatable[T]` ist gar kein Typ (so daß man auch keine Variable mit ihm deklarieren kann), so daß keinerlei Zuweisungskompatibilität und damit auch kein Problem mit Typkorrektheit besteht.

Trotzdem stellt sich die Frage, ob sich das in Abschnitt 3.9.3 angesprochene allgemeine Problem der wünschenswerten kovarianten Redefinition von Eingabeparametern in Methoden mittels parametrischen Polymorphismus nicht irgendwie lösen läßt. Die Antwort ist unbefriedigend: nur zum Teil.

So kann man, um das Beispiel von Dokumenten und Druckern aus Abschnitt 3.9.3 wieder aufzugreifen, einen parametrischen Typ `Dokument` wie folgt definieren:

Typ	Dokument
Typvariablen	T < Drucker
Protokoll	

```
druckenAuf: einDrucker <T> ^ <Self>
```

Die Deklaration von `Zeichnung` mit Typparameter `T` als Subtyp von `Dokument` vorausgesetzt, lassen sich die folgenden Variablendeklarationen bilden:

```
| z <Zeichnung[Plotter]> p <Plotter> l <Zeilendrucker> |
```

Weiterhin die Deklarationen von `Plotter` und `Zeilendrucker` als Subtypen von `Drucker` vorausgesetzt wäre ein Methodenaufruf

```
z druckenAuf: p
```

typkorrekt,

```
z druckenAuf: l
```

hingegen nicht. Allerdings ist die Assoziation von `Zeichnung` mit `Plotter`, die Kovarianz, in keiner Typdefinition festgehalten, sondern lediglich in der Deklaration von `z`. Es hindert einen insbesondere nichts daran, dieselbe oder eine andere Variable als vom Typ `Zeichnung[Zeilendrucker]` zu deklarieren. Man beachte, daß es anders als im obigen Beispiel von `Equatable`, wo ja der Typparameter auf den definierten Typ selbst eingeschränkt wurde, hier keine Möglichkeit gibt, einen bestimmten Wert für einen Typparameter vorzuschreiben.

Was man allerdings tun könnte, ist, `Zeichnung` als Subtyp von `Dokument[Plotter]` zu definieren. Dies hat jedoch den Nachteil, daß `Zeichnung` damit kein Subtyp mehr von `Dokument` und, wie auch zuvor schon `Zeichnung [Plotter]` kein Subtyp von `Dokument[Drucker]` ist (s. Abschnitt 3.12.3), wodurch die Zuweisungskompatibilität mit entsprechend deklarierten Variablen verlorengeht. Kovariante Redefinition bei gleichzeitiger Inklusionspolymorphie läßt sich auch mittels parametrischer Typen nicht hinbekommen.

3.14 Grenzen der Typisierung

Wie Sie sehen, ist das Problem der kovarianten Redefinition ziemlich hartnäckig. Man muß aber gar nicht so weit gehen, um an die praktischen Grenzen der Typisierung zu gelangen: Bereits der Ausdruck

```
x reciprocal
```

beinhaltet einen Typfehler, wenn nicht sichergestellt ist, daß `x` nicht `0` enthält. Nun könnte man einen Typ `NotZero` definieren und `x` als von diesem Typ deklarieren, womit der obige Ausdruck kein Problem mehr wäre; mit den hier vorgestellten Mitteln der statischen Typprüfung wäre dann aber schon die einfache Zuweisung

```
x := y - z
```

nicht mehr auf Zulässigkeit prüfbar. Selbst wenn es Typsysteme gibt, die das können[68], so sind diese kaum praxistauglich.

3.15 Weiterführende Literatur

Typsysteme sind immer noch Gegenstand aktiver Forschung. Während die prozeduralen und objektorientierten Programmiersprachen eher pragmatisch an das Thema herangehen, sind auf dem Gebiet der funktionalen Programmiersprachen ausgefeilte Theorien entwickelt worden, die nach und nach auf andere Programmiersprachen übertragen werden. Die meisten der heute in Gebrauch befindlichen objektorientierten Programmiersprachen sind hingegen nicht die Quintessenz ausgereifter theoretischer Überlegungen, sondern vielmehr das Produkt von Ideen, Experimenten und einer ganzen Menge praktischer Zwänge.

Die Originalarbeit zu Polymorphismus ist die von Strachey, die bereits im Herbst 1967 verfaßt, allerdings erst 2000 veröffentlicht wurde. Die darin noch nicht enthaltene Inklusionspolymorphie, die für die objektorientierte Programmierung charakteristisch ist, wurrde erst von Cardelli und Wegner 1985 hinzugefügt. Eine sehr gute, nicht allzu theorielastige Einführung in Typsysteme für die objektorientierte Programmierung ist das Buch von Palsberg und Schwartzbach; es ist recht dünn und dabei noch einigermaßen angenehm zu lesen. Sehr viel weitergehend ist das Werk von Pierce, das ich hiermit aber nicht empfohlen haben will.

[Strachey 2000]
C Strachey „Fundamental concepts in programming languages“ *Higher-Order and Symbolic Computation* 13:1–2 (2000) 11–49.

[Cardelli & Wegner 1985]
L Cardelli, P Wegner „On understanding types, data abstraction, and polymorphism“ *ACM Computing Surveys* 17:4 (1985) 471–523.

[Palsberg & Schwartzbach 1994]
J Palsberg, MI Schwartzbach *Object-Oriented Type Systems* (John Wiley & Sons, 1994).

[Pierce 2002]
BC Pierce *Types and Programming Languages* (MIT Press 2002).

---

[68] Tatsächlich gibt es Typsysteme, die selbst Turing-äquivalent sind, mit denen man also alle Prüfungen durchführen kann, die auch vom Programm selbst durchgeführt werden könnten.