Skriptum zur Vorlesung

Typsysteme

im Wintersemester 2015/2016

Franz Puntigam
Technische Universität Wien
Institut für Computersprachen
Programmiersprachen und Übersetzer
http://www.complang.tuwien.ac.at/franz/typsysteme.html

Inhaltsverzeichnis

Kapitel 1

Begriffsbestimmungen und Überblick

Typsysteme sind Bestandteile von Programmiersprachen und können als solche nur im Zusammenhang mit Programmiersprachen sinnvoll betrachtet werden. Dennoch gibt es theoretische Grundlagen von Typsystemen, die einheitlich auf viele Sprachen und Sprachklassen anwendbar sind. Typsysteme entwickelten sich in den letzten Jahrzehnten stetig weiter und gewannen zunehmend an Bedeutung—ein Prozess, dessen Ende noch nicht absehbar ist. Typen können als ein Aspekt gesehen werden, der gemeinsame und unterschiedliche Eigenschaften von Sprachen in allen Sprachklassen herausstreicht. Das Verstehen von Typsystemen trägt daher wesentlich zum Verstehen von Programmiersprachen und der Softwareentwicklung und -wartung im Allgemeinen bei.

1.1 Was sind Typen?

Jeder, der schon einmal eine typisierte Programmiersprache verwendet hat, weiß ungefähr, was ein Typ ist. Allerdings gibt es zahlreiche Definitionen des Begriffs „Typ“, die auf den ersten Blick keinerlei Gemeinsamkeiten aufweisen. Für einen Anwendungsprogrammierer bedeuten Typen sicher etwas anderes als für einen Compilerentwickler oder Programmiersprachendesigner. Folgende Antworten auf die Frage „Was sind Typen?“ spiegeln verschiedene Interessen und Sichtweisen wider:

1. *Typen sind Werkzeuge zur Klassifikation von Werten aufgrund ihrer Eigenschaften, ihres Verhaltens und ihrer Verwendungsmöglichkeiten.*

   Die Zuordnung von Werten (z.B. `1`, `2`, `true` und der Implementierung von `print`) zu Typen (z.B. `Integer`, `Boolean` und `Procedure`) hilft bei der Strukturierung von Programmen und vereinfacht damit die Programmerstellung. Der Typ einer Variablen beschreibt nicht nur welche Werte sie enthalten kann, sondern auch welche Operationen darauf sinnvoll anwendbar sind.

2. *Typen sind Schutzschilder gegen unbeabsichtigte bzw. falsche Interpretation roher Daten.*

   Bitmuster in Speicherzellen können auf verschiedene Arten (z.B. als Adressen, ganze Zahlen oder Gleitkommazahlen) interpretiert werden. Typen schränken die erlaubten Interpretationsmöglichkeiten ein und verhindern dadurch eine Klasse von Programmierfehlern.

3. *Typen sind Werkzeuge des Softwareentwicklers zur Unterstützung der Programmerstellung, Weiterentwicklung und Wartung.*

   Diese Antwort erweitert die vorhergehenden Antworten aus der Sicht eines Softwareentwicklers, der neben der Erstellung vor allem auch die Weiterentwicklung und Wartung von Software im Auge hat. Wichtige Konzepte objektorientierter Programmiersprachen (z.B. Klassen und Vererbung) stehen in enger Beziehung zu Typsystemen.

4. *Typen schränken Ausdrücke syntaktisch ein, sodass Kompatibilität zwischen Operatoren und Operanden zugesichert wird.*

   In vielen Programmiersprachen braucht nicht jeder Operator mit allen Operanden kompatibel zu sein. Typüberprüfungen stellen sicher, dass Operatoren und Operanden verträglich sind und die Operationen daher tatsächlich ausgeführt werden können. Dadurch wird eine mögliche Fehlerquelle ausgeschaltet. Falls die Typüberprüfung vom Compiler vorgenommen wird, kann das Programm möglicherweise effizienter ausgeführt werden.

5. *Typen bestimmen Speicherauslegungen für Werte.*

   Aus der Sicht eines Compilerentwicklers ist es notwendig zu wissen, welchen Speicherplatz ein Wert belegt und wie dieser strukturiert ist. Typen sind eine gute Quelle für diese Information.

6. *Typen legen Verhaltens-Invarianten fest, die alle Instanzen der Typen erfüllen müssen.*

Typen dienen auch der Verifikation. Dadurch, dass alle Instanzen (= Konstanten, Objekte, Werte) eines Typs dasselbe invariante (= unveränderliche) Verhalten zeigen, wird die Verifikation vereinfacht. Auf dieser Grundlage beruht das Ersetzbarkeitsprinzip in der objektorientierten Programmierung.

7. *Ein Typsystem ist eine Menge von Regeln, die jedem Ausdruck einen eindeutigen allgemeinsten Typ zuordnet. Dieser Typ beschreibt die Menge aller Umgebungen, in denen der Ausdruck vorkommen kann und eine Bedeutung hat.*

   Vor allem in funktionalen Programmiersprachen werden die Typen von Ausdrücken oft nicht explizit angegeben. Trotzdem haben einige dieser Sprachen ein ausgeklügeltes Typsystem, das es erlaubt, jedem Ausdruck einen eindeutig berechenbaren Typ zuzuordnen. Der vom Compiler berechnete Typ eines Ausdrucks (z.B. einer Funktion) erlaubt dem Programmierer, Fehler zu erkennen und ermöglicht Optimierungen.

Die ersten drei Antworten stammen am ehesten von Benutzern einer Sprache bzw. eines Compilers, die zwei nächsten von Compilerentwicklern und die beiden letzten von Leuten, die sich mit der Theorie der Programmierung und Programmiersprachen beschäftigen. Eine gute Programmiersprache kann nur das Produkt einer Zusammenarbeit dieser drei Disziplinen sein. Eine Sprache, die aus der Sicht des Programmierers große Vorteile bietet, aber nur sehr ineffizient implementiert werden kann oder keine wohldefinierte Semantik hat, wird längerfristig genausowenig Erfolg haben wie eine, die effiziente Implementierungen eines schönen theoretischen Modells erlaubt, aber praktisch kaum verwendbar ist. Die Frage, welche der oben genannten Definitionen die richtige oder die wichtigere sei, ist daher überflüssig: Ein Typkonzept soll den Definitionen aus allen Sichtweisen entsprechen. Falls eine davon nicht vollständig erfüllt werden kann, muss es zumindest eine überzeugende Erklärung dafür geben. Tatsächlich können sich einige der genannten Definitionen teilweise widersprechen, sodass jedes Typsystem einen Kompromiss darstellt.

Als generelles Ziel für den Einsatz von Typsystemen in Programmiersprachen wird häufig die verbesserte Lesbarkeit, Sicherheit und Effizienz von Programmen genannt. Die obigen Antworten Nr. 1, 3 und 6 zielen auf verbesserte Lesbarkeit ab, die Antworten Nr. 2, 4, 6 und 7 auf erhöhte Sicherheit und die Antworten Nr. 4, 5 und 7 auf verbesserte Effizienz von Programmen. Daneben hat Antwort Nr. 3 zusammen mit 7 auch effektivere Softwareentwicklungsprozesse zum Ziel.

Im folgenden Text wird hauptsächlich diese Definition des Typbegriffs implizit verwendet werden:

Ein Typ beschreibt eine Menge von Instanzen.

Diese Definition ist so allgemein gehalten, dass sie jeder der weiter oben angeführten Definitionen entspricht. Sie gibt nicht an, auf welche Art die Menge beschrieben ist, welche Elemente sie enthält, in welcher Beziehung diese Elemente zueinander stehen und welche gemeinsame Eigenschaften sie haben. Genausowenig legt sie die Relationen zwischen den Typen eines Typsystems fest. Ein konkretes Typsystem wird durch die Festlegung dieser offenen Punkte definiert.

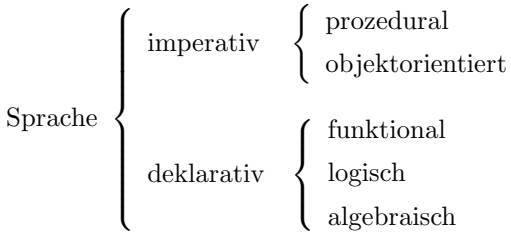
Wenn man vom Typ eines Wertes oder Objekts spricht, meint man die Menge der Instanzen zu denen der Wert bzw. das Objekt zählt. Die vielfältige Verwendung des Typ-Begriffs hat jedoch zu einer etwas „schlampigeren“ Terminologie geführt. So spricht man häufig vom „Typ einer Variablen“ und meint damit Einschränkungen der Werte oder Objekte, die durch diese Variable bezeichnet werden können. Wenn man es genau nimmt ist der „Typ einer Variablen“ eine Menge von Variablennamen; das was salopp „Typ einer Variablen“ genannt wird ist genaugenommen eine „Typeinschränkung auf einer Variablen“. In diesem Skriptum wird oft diese saloppe Terminologie verwendet, da sie natürlich klingt und in der Regel nicht zu Mehrdeutigkeiten führt.

1.2 Kategorisierung

Die folgenden Einteilungen von Programmiersprachen aufgrund ihrer Paradigmen und Typsysteme sollen einen systematischen Überblick über die zahlreichen Möglichkeiten für den Entwurf von Programmiersprachen geben, wobei Typen im Mittelpunkt stehen.

1.2.1 Paradigmen und Sprachklassen

Weltweit gibt es zwei- bis dreitausend Programmiersprachen, und ständig kommen neue dazu. Die meisten davon, insbesondere alle häufig verwendeten, lassen sich anhand der von ihnen am besten unterstützten Programmierparadigmen in einige wenige Sprachklassen einteilen:



Imperative Sprachen werden dadurch charakterisiert, dass Programme aus *Anweisungen* (= Befehlen) aufgebaut sind. Diese werden in einer festgelegten Reihen-

folge ausgeführt, in parallelen imperativen Sprachen teilweise auch gleichzeitig bzw. in beliebiger Reihenfolge. Grundlegende Sprachelemente sind Variablen, Konstanten, Routinen (= Funktionen bzw. Prozeduren) und—in fast allen imperativen Sprachen—Typen. Der wichtigste Befehl in diesen Sprachen ist die *destruktive Zuweisung*: Eine Variable bekommt einen neuen Wert, unabhängig vom Wert den sie vorher hatte. Die Menge der Werte in allen Variablen im Programm sowie ein (oder mehrere) Zeiger auf den (die) nächsten auszuführenden Befehl(e) beschreiben den *Programmzustand,* der sich mit der Ausführung jeder Anweisung ändert.

Der wichtigste Abstraktionsmechanismus in *prozeduralen Sprachen* (z.B. Algol, Fortran, Cobol, C, Pascal, Modula-2, Ada, u.s.w.) ist die Routine (bzw. Prozedur). Programme werden aufgrund funktionaler Abhängigkeiten zwischen Daten in sich gegenseitig aufrufende und den globalen Programmzustand verändernde Routinen zerlegt. „Saubere“ prozedurale Programme werden mittels *strukturierter Programmierung* geschrieben.

Die *objektorientierte Programmierung* ist eine Weiterentwicklung der strukturierten prozeduralen Programmierung, die den abstrakten Begriff des *Objekts* in den Mittelpunkt stellt. Ein Objekt hat eine *Identität,* einen *Zustand* und ein festgelegtes *Verhalten.* Objekte kommunizieren miteinander durch den Austausch von Nachrichten. Man unterscheidet zwischen *aktiven* und *passiven* Objekten. Bei passiven Objekten, die in allen sequentiellen objektorientierten Sprachen verwendet werden, entspricht das Senden einer Nachricht in etwa einem Prozeduraufruf. Das Verhalten eines passiven Objekts wird durch eine Menge von Operationen, die den Objektzustand ändern können, beschrieben. Aktive Objekte sind Prozesse, die sich auch über andere Kommunikationsmechanismen wie z.B. synchrones oder asynchrones *Message-Passing* verständigen. Das Verhalten eines aktiven Objekts wird meist durch ein Programmstück, das in einer Endlosschleife beliebig oft ausgeführt werden kann, festgelegt. Wie bei prozeduralen Sprachen erfolgen Zustandsänderungen sowohl von passiven als auch aktiven Objekten durch destruktive Zuweisungen. Der wesentliche Unterschied zur prozeduralen Programmierung ist der, dass zusammengehörende Operationen, Prozesse und Daten zu Objekten zusammengefasst werden. Dadurch ist es in vielen Fällen möglich, die Programmausführung anhand der Zustandsänderungen in den einzelnen Objekten zu beschreiben, ohne globale Änderungen der Programmzustände betrachten zu müssen. Als weitere Abstraktionsmechanismen bieten objektorientierte Sprachen (z.B. C++, Eiffel, Smalltalk, Modula-3, Java, C#, Ada 95, u.s.w.) *Klassen* und *Vererbung* oder vergleichbare Sprachkonstrukte. Klassen dienen der Klassifikation und Beschreibung von Objekten und stehen oft in enger Beziehung zu Typen. Vererbung ist ein häufig verwendeter—obwohl nicht der einzig mögliche—Mechanismus zur Erstellung neuer Klassen durch Verwendung bereits existierender Klassen.

Deklarative Programme beschreiben ein statisches Modell durch die Beziehungen zwischen Ausdrücken dieses Modells. Im Prinzip gibt es in deklarativen Sprachen keine zustandsändernden Anweisungen. Deklarative Sprachen entstanden aus mathematischen Sprachen zur Beschreibung von Modellen und stehen daher in der Regel auf einem höheren Abstraktionsniveau als imperative Sprachen, die sich aus Maschinensprachen und hardwarenahen Berechnungsmodellen entwickelten. Grundlegende Sprachelemente sind Symbole, die sich manchmal in mehrere Klassen (z.B. Variablensymbole, Funktionssymbole, Prädikate, u.s.w.) einteilen lassen.

Eines der für die Informatik bedeutendsten theoretischen Modelle ist der Lambda-Kalkül, der den mathematischen Begriff *Funktion* formal definiert. Programmiersprachen, die auf diesem Kalkül beruhen, heißen *funktionale Sprachen.* Beispiele sind Lisp, ML, Miranda und Haskell. Alle Ausdrücke in diesen Sprachen werden als Funktionen aufgefasst, und der wesentliche Berechnungsschritt besteht in der Anwendung einer Funktion auf einen Ausdruck. Der Lambda-Kalkül hatte einen großen Einfluss auf die historische Entwicklung der imperativen Sprachen. Manchmal werden funktionale Sprachen als „saubere“ Varianten prozeduraler Sprachen angesehen, die ohne „unsaubere“ destruktive Zuweisung auskommen.

Logische Sprachen beruhen auf der Klausel-Logik, einer Teilmenge der Prädikatenlogik. Die Menge aller wahren Aussagen in einem Modell wird mittels Fakten und Regeln beschrieben. Um einen Berechnungsvorgang zu starten, wird eine Anfrage gestellt. Das Ergebnis der Berechnung besagt, ob und unter welchen Bedingungen die in der Anfrage enthaltene Aussage wahr ist. Der wichtigste Vertreter dieser Sprachen, Prolog, hat eine prozedurale Interpretation; das heißt, Fakten und Regeln können als Prozeduren aufgefasst und wie in prozeduralen Sprachen ausgeführt werden.

Algebraische Sprachen, die auf freien Algebren beruhen, wurden bisher fast ausschließlich als Spezifikationssprachen verwendet. Sie sind hier dennoch als eigene Sprachklasse angeführt, da ihre Konzepte in vielen Sprachen in Form von Modulen und in objektorientierten Sprachen als Klassen vorkommen.

Die Zuordnung von Sprachen zu Sprachklassen ist nicht immer so eindeutig wie man vielleicht glauben möchte. Zum Beispiel gibt es in Lisp, einer im Wesentlichen funktionalen Sprache, auch eine destruktive Zuweisung, und CLOS (= Common-Lisp-Object-System) ist eine bekannte objektorientierte Erweite-

rung von Lisp. Natürlich kann man auch in C++ oder Java funktionale oder prozedurale Programme schreiben. Und auch in C oder Modula-2 kann man, allerdings meist nur mit hohem Aufwand, objektorientierte Programme schreiben. Die Art der Verwendung einer Programmiersprache lässt sich oft leichter einem Paradigma zuordnen als die Sprache selbst. Da Programmiersprachen im Hinblick auf bestimmte Paradigmen entwickelt werden, ist die Zuordnung von Sprachen zu Paradigmen aufgrund ihrer vorherrschenden Verwendungen jedoch meist problemlos möglich.

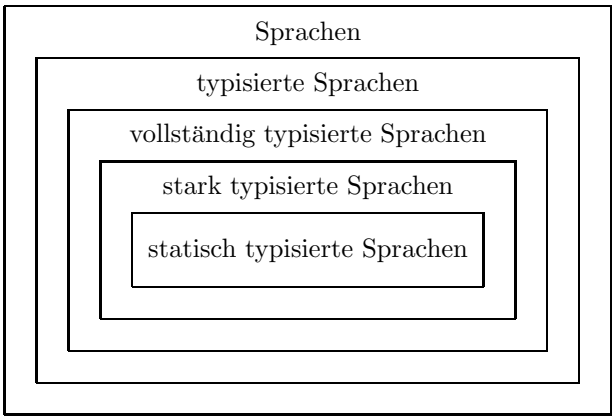
1.2.2 Einteilung von Typsystemen

Typsysteme können nach einer Reihe von Kriterien eingeteilt werden:

Kriterium	Ausprägungen
Definierbarkeit	vordefiniert, definierbar
Art der Definition	extensional, intensional
Typfestlegung	statisch, stark, schwach, ...
Mächtigkeit	flexibel, sicher, einfach, ...
Typzwang	konservativ, optimistisch
Durchlässigkeit	dicht, löchrig
Typäquivalenz	struktur-, namensgleich
Abstraktionsgrad	abstrakt, konkret
Ausdrücklichkeit	implizit, explizit
Überschneidungen	monomorph, polymorph

Prinzipiell können Typen *extensional* oder *intensional* spezifiziert sein. Eine extensionale Spezifikation zählt die Instanzen eines Typs auf. Durch die Angabe seiner Eigenschaften relativ zu anderen Typen wird ein Typ intensional spezifiziert. Eine Analogie gibt es in der Mathematik bei der Spezifikation von Mengen: $\{2, 3, 5, 7, 11, 13\}$ ist eine extensional spezifizierte Menge. $\{i \in \{1, \ldots, 16\} \mid p(i)\}$ beschreibt dieselbe Menge relativ zur Menge $\{1, \ldots, 16\}$ der ganzen Zahlen im Bereich von 1 bis 16, wobei das Prädikat $p(i)$ genau dann wahr ist wenn i eine Primzahl ist. Bei extensional spezifizierten Mengen (bzw. Typen) ist die Zugehörigkeit eines Elements (einer Instanz) stets einfach feststellbar. Dagegen ist bei intensional spezifizierten Mengen (Typen) nicht immer entscheidbar, ob ein bestimmtes Element in der Menge enthalten ist (zum Typ gehört) oder nicht. Intensionale Spezifikationen sind wesentlich mächtiger als extensionale. Zum Beispiel beschreibt $\{i \in \mathbb{N} \mid p(i)\}$, wobei $\mathbb{N}$ die Menge der natürlichen Zahlen bezeichnet, die unendliche Menge aller Primzahlen, obwohl diese nicht auf einfache Weise rekursiv aufzählbar ist.

Eine Programmiersprache, in der es Typen gibt, heißt *typisiert.* Eine Programmiersprache, in der der eindeutige Typ jedes Ausdrucks durch statische Programmanalyse ermittelt werden kann, heißt *statisch typisiert.* Statische Typisiertheit ist eine brauchbare Eigenschaft, da Typen die möglichen Verwendungen von Ausdrücken angeben und dadurch zur Lesbarkeit von Programmen beitragen. Aber die Forderung, dass jedem Ausdruck zur Compilezeit ein eindeutiger Typ zugeordnet sein muss, ist in einigen Fällen zu restriktiv. Eine schwächere Forderung, *vollständige Typisiertheit*, ist, dass alle Ausdrücke garantiert *typkonsistent* sind, obwohl nicht alle Typen dem Compiler bekannt sein müssen. Typkonsistenz muss spätestens während der Programmausführung überprüft werden können. Sprachen, in denen die Typkonsistenz aller Ausdrücke durch einen Compiler garantiert werden kann, heißen *stark typisiert.* In stark typisierten Sprachen können während der Programmausführung niemals Typfehler auftreten. Jede statisch typisierte Programmiersprache ist auch stark typisiert, da dem Compiler die zur Typüberprüfung notwendige Information vorliegt. Und jede stark typisierte Sprache ist vollständig typisiert. Die Umkehrungen gelten nicht: Nicht jede vollständig typisierte Sprache ist stark typisiert, und nicht jede stark typisierte Sprache ist statisch typisiert. Diese Beziehungen können grafisch veranschaulicht werden—Rechtecke stellen Mengen dar:



Typisierte Sprachen, die nicht statisch typisiert sind, werden *dynamisch typisiert* genannt. Solche, die nicht stark typisiert aber typisiert sind, heißen *schwach typisiert.*

Die im vorigen Absatz eingeführten Begriffe und Beziehungen sind etwas verwirrend und können leicht zu Verwechslungen führen. Daher werden sie hier nocheinmal gegenübergestellt:

Statisch bzw. dynamisch typisiert: Diese Eigenschaften beziehen sich darauf, ob jedem Ausdruck eines Programms statisch (d.h. zur Compilezeit) ein eindeutiger Typ *zugeordnet* werden kann oder nicht. Sie bezieht sich nicht darauf, ob die Typkonsistenz von einem Compiler überprüft werden kann, obwohl in der Praxis jedes statisch typisierte Programm auch stark typisiert ist.

Von einer Sprache, in der mehr Ausdrücken statisch ein eindeutiger Typ zugeordnet werden kann als in einer zweiten Sprache, sagt man manchmal, sie sei „statischer“ typisiert als diese zweite Sprache. Man sagt von Sprachen auch, sie seien bis auf gewisse Sprachkonstrukte oder Aspekte statisch typisiert, wenn die entsprechenden Sprachen ohne diese Konstrukte bzw. unter Vernachlässigung dieser Aspekte statisch typisiert sind.

Stark bzw. schwach typisiert: Diese Eigenschaften beschreiben, ob ein Compiler *Typkonsistenz garantieren* kann oder nicht. Man beachte, dass die Typüberprüfung zur Compilezeit auch *statische Typüberprüfung* und jene zur Laufzeit *dynamische Typüberprüfung* genannt wird. Eine Sprache, die eine statische Typüberprüfung ermöglicht, heißt *stark typisiert*, aber sie ist nicht notwendigerweise statisch typisiert. Eine Sprache, die eine statische Typüberprüfung nicht ermöglicht, heißt *schwach typisiert.*[1]

Von einer Sprache, in der mehr Ausdrücke statisch auf Typkonsistenz geprüft werden können als in einer zweiten Sprache, sagt man oft, sie sei stärker typisiert als diese zweite Sprache. Eine stark typisierte Sprache ist stärker typisiert als jede nicht stark typisierte Sprache. Man sagt von Sprachen auch, sie seien bis auf gewisse Konstrukte oder Aspekte stark typisiert, wenn diese Sprachen ohne diese Konstrukte bzw. unter Vernachlässigung dieser Aspekte stark typisiert sind.

Vollständigkeit: Eine Sprache ist vollständig typisiert wenn eine Typüberprüfung (statisch oder dynamisch) möglich ist. Von einer Sprache, in der Programme nur zum Teil auf Typkonsistenz überprüft werden können, sagt man, sie habe ein *durchlässiges* oder *löchriges* Typsystem, bzw. sie sei bis auf diese oder jene Konstrukte oder Aspekte vollständig typisiert. Nicht vollständig typisierte Sprachen haben mehr oder weniger Löcher, bzw. sind mehr oder weniger typisiert.

Eine Sprache ist typisiert, wenn es in der Sprache ein Typsystem gibt. Eine Sprache, für die überhaupt kein Typsystem definiert ist, heißt *untypisiert.*

In (bis auf einige Konstrukte) statisch typisierten Sprachen sind verschiedene Aspekte eines Programms statisch festgelegt. Diese Aspekte können für Programmanalysen und Optimierungen herangezogen werden und haben häufig auch den Charakter einer zuverlässigen Programmdokumentation. Dynamische Aspekte sind statischen Analysen—aber auch der statischen Betrachtung durch den Programmierer—nur in beschränktem Umfang zugänglich. Statisch festgelegte Typen sind praktisch immer durch die Sprache *syntaktisch* festgelegt. Programme werden also ausschließlich auf syntaktische Konsistenz hin überprüft; die Bedeutung (= *Semantik*) eines Programms bzw. Typs bleibt dem Compiler unbekannt. In allen in der Praxis eingesetzten (bis auf einige Konstrukte) statisch typisierten Sprachen ist es möglich, Programme zu schreiben, deren Typen syntaktisch konsistent sind, obwohl die intuitiven Bedeutungen der Typen inkonsistent sein können. Statische Typüberprüfung ist also kein Allheilmittel gegen Laufzeitfehler. Der Begriff *Typfehler* bezieht sich nur auf syntaktische Einschränkungen, die von Sprache zu Sprache stark divergieren.

Eines der Ziele jedes Typsystems ist die Ausschaltung einiger Fehlerarten in Programmen. Allerdings schränken Typsysteme, die eine große Anzahl von Fehlerarten abfangen, die Freiheit der Programmierer stark ein. Daher hat nicht jede Programmiersprache ein umfangreiches und sicheres Typsystem. Zum Beispiel gibt es in Forth im Wesentlichen nur den Typ `cell`, der je nach Operation als ganze Zahl, Fließkommazahl oder Adresse aufgefasst wird. Dieses sehr einfache Typkonzept erlaubt große Flexibilität und erlaubt einfache, effiziente Implementierungen von Interpretern auf Kosten der Typsicherheit. Das Ziel der meisten neueren Programmiersprachen ist jedoch, sowohl Typsicherheit als auch Flexibilität zu bieten. Man versucht, die möglichen Fehlerquellen immer genauer einzugrenzen, sodass die Anzahl der vom Typsystem akzeptierten Programme ebenso wie die der abgefangenen Fehlerarten vergrößert wird. Die Kombination von Typsicherheit und Flexibilität ist in weitem Umfang möglich, jedoch häufig nur auf Kosten der Einfachheit und Verständlichkeit. In einigen Sprachen mit einem sehr mächtigen Typkonzept können sogar erfahrene Programmierer kaum begründen, warum ein bestimmtes Programm vom Compiler akzeptiert wird oder nicht. Viele Sprachen verzichten deswegen zugunsten eines überschaubaren, einigermaßen sicheren Typsystems zum Teil auf Flexibilität.

Im vorigen Absatz wurde implizit davon ausgegangen, dass der Compiler nur solche Programme übersetzen kann, die typkonsistent sind. Bei Typfehlern muss der Programmierer das Programm so umschreiben, dass es nachweislich keine Typfehler enthält. Diese Vorgangsweise nennt man *konservativ*. Da Typsysteme nur einen Teil der Ausdruckskraft einer Sprache umfassen können, schließen konservative Typüberprüfungen immer eine größere Anzahl von Programmen von der Ausführung aus als tatsächlich Typfehler enthalten. Im Gegensatz dazu gehen *optimistische* Typüber-

[1] Diese verwirrende Terminologie hat historische Gründe. Es war nicht von Anfang an klar, dass es stark typisierte Sprachen gibt, die nicht auch statisch typisiert sind.

prüfungen davon aus, dass Programme keine Typfehler enthalten. Sie warnen höchstens vor möglichen Typfehlern, weisen aber nur solche Programme zurück, die nachweislich für alle möglichen Ausführungen fehlerhaft sind. Optimistische Typüberprüfungen findet man zum Beispiel in einigen Lisp-Dialekten.

In jeder vollständig typisierten Sprache werden, im Prinzip, alle Typfehler spätestens bei der Programmausführung entdeckt. In der Praxis wird aus Effizienzgründen jedoch häufig auf Typüberprüfungen während der Ausführung verzichtet. Ein klassisches Beispiel ist die Überprüfung, ob ein Index innerhalb der Feldgrenzen liegt. Solche Löcher in der Typüberprüfung können leicht Fehler unerkannt bleiben lassen und zu falschen Resultaten führen. Dies ist besonders dann unangenehm wenn der Programmierer sich darauf verlässt, dass die Typen auf Konsistenz überprüft werden. Es ist daher wichtig, alle solchen Löcher, also alle Konstruktionen, die aufgrund der Sprachdefinition zwar verboten sind aber durch die Typüberprüfung nicht als verboten erkannt werden, zu kennen.

Oft besteht ein Typsystem nur aus einigen vordefinierten einfachen Typen (z.B. ganze Zahlen, Fließkommazahlen) und wenigen Typkonstruktoren (z.B. Felder, Verbunde, Paare). Man kann Typen als äquivalent ansehen, wenn sie gleich strukturiert sind (z.B. in Fortran, Lisp und zu einem großen Teil in C), oder wenn explizit definierte Typen den gleichen, an nur einer Stelle eingeführten Namen haben (z.B. teilweise in Pascal und praktisch allen objektorientierten Sprachen). *Typäquivalenz* aufgrund von *Strukturgleichheit* hat den Vorteil der einfacheren Handhabung. Es braucht nicht darauf geachtet zu werden, wie der Typ heißt. Andererseits kann Typäquivalenz aufgrund von *Namensgleichheit* als zusätzlicher Abstraktionsmechanismus zur Programmstrukturierung eingesetzt werden. Zum Beispiel ist es nicht möglich, die Instanzen der Typen `Apfel` und `Birne` in unerwünschter Weise zu vermischen, auch wenn beide Typen ganze Zahlen sind. Man sagt, `Apfel` und `Birne` sind zwei von `Integer` *abgeleitete* Typen. Zusammen mit Verbunden stellt dieser Abstraktionsmechanismus eine Grundlage für Vererbung in vielen objektorientierten Sprachen dar.

Typäquivalenz aufgrund von Namensgleichheit erlaubt die Einführung von *abstrakten Datentypen*, kurz ADT. Ein ADT „versteckt“ die interne Darstellung seiner Instanzen. Nach außen hin wird eine Instanz als abstraktes Objekt ohne innere Struktur, z.B. als Adresse, repräsentiert. Auf diesen Objekten sind nur die vom ADT exportierten Operationen anwendbar. Manchmal wird ein ADT als Modul implementiert, das Instanzen als „opaque“ Zeiger auf Instanzen eines nicht exportierten Typs darstellt. Diese Zeiger und einige darauf anwendbare Routinen werden exportiert. In vielen objektorientierten Sprachen ist ein ADT ein Verbund, der neben Daten auch Routinen (Methoden) enthält. Einige Komponenten werden durch das Typsystem vor Zugriffen von außen geschützt.

Von abstrakten Datentypen zu unterscheiden sind *abstrakte Typen.* Abstrakte Typen stellen unvollständig spezifizierte Typen dar, aus denen *konkrete Typen* erzeugt werden können. Abstrakte Typen werden als Strukturierungsmittel zur Erzeugung vieler gleich oder ähnlich strukturierter konkreter Typen und als Schnittstellenbeschreibungen in objektorientierten Sprachen verwendet.

Nicht alle Sprachelemente in einer typisierten Sprache müssen einen Typ haben. So sind zum Beispiel Variablen in Lisp oder Smalltalk nicht typisiert; d.h. es gibt keine Typeinschränkungen auf Variablen. Dagegen sind die Werte bzw. Objekte in diesen Sprachen vollständig typisiert; die Typkompatibilität von Werten bzw. Objekten wird zur Laufzeit lückenlos überprüft. In statisch typisierten Sprachen müssen jedoch auch Variablen typisiert sein.

Typen von Variablen werden auf verschiedene Arten deklariert. In einigen älteren imperativen Sprachen ist die Typdeklaration implizit. Zum Beispiel enthalten in Fortran Variablen, die mit den Buchstaben `i` bis `n` oder `I` bis `N` beginnen, falls kein anderer Typ deklariert ist, eine ganze Zahl, solche, die mit einem anderen Zeichen beginnen, eine Fließkommazahl. In praktisch allen neueren imperativen Sprachen muss der Typ der meisten typisierten Sprachelemente explizit deklariert werden. Ausgenommen sind in der Regel nur Literale. In sehr vielen deklarativen Sprachen ist der Typ von Ausdrücken nach wie vor implizit definiert. Zum Beispiel ist in Lisp `1` eine ganze Zahl, `a` ein Symbol und `(a 1)` eine Liste. Letzterem Ausdruck kann man aber auch den etwas spezielleren Typ „zweielementige Liste von einem Symbol und einer ganzen Zahl“ zuordnen. Typen sind also nicht eindeutig, sondern können in einem weiten Bereich zwischen sehr allgemein und ganz speziell liegen.

Funktionen in funktionalen Sprachen können beliebige Ausdrücke als Argumente übergeben werden. Mittels *Typinferenz* lässt sich in neueren funktionalen Sprachen für jede vom Compiler akzeptierte Funktion ein allgemeinster, für jedes Argument gültiger Ergebnistyp berechnen.

1.2.3 Polymorphe Typsysteme

Konventionelle typisierte Sprachen wie z.B. Pascal beruhen darauf, dass jede Variable oder Routine genau einen eindeutigen Typ hat. Solche Sprachen heißen *monomorph.* Im Gegensatz dazu können Variablen und Routinen in *polymorphen Sprachen* mehrere Typen haben. Jeder formale Parameter einer *polymorphen Routine* kann an Argumente von mehr als nur einem Typ

gebunden werden. *Polymorphe Typen* sind solche, deren Operationen auf Operanden mehrerer Typen anwendbar sind.

In einer Sprache mit polymorphem Typsystem hat eine Variable oder ein Parameter meist gleichzeitig folgende Typen: Der *deklarierte Typ* ist jener, mit dem die Variable deklariert wurde. Dieser existiert natürlich nur in Sprachen, in denen Typen von Variablen deklariert werden. Der *statische Typ* ist jener, der vom Compiler (z.B. durch Typinferenz) ermittelt wird. Dieser Typ kann spezieller sein als der deklarierte Typ. In vielen Fällen ordnet der Compiler ein und derselben Variablen an verschiedenen Stellen in einem Programm verschiedene statische Typen zu. Statische Typen werden für die statische Typüberprüfung verwendet. Der *dynamische Typ* ist der speziellste Typ, den der in der Variable gerade gespeicherte Wert hat. Dynamische Typen werden unter anderem für die dynamische Typüberprüfung und für dynamisches Binden in objektorientierten Sprachen verwendet.

Man kann verschiedene Arten von polymorphen Typsystemen unterscheiden:

- polymorphes Typsystem
  - universell quantifiziert
    - parametrisch
    - Subtyping
  - ad hoc polymorph
    - überladen
    - umwandelnd

Parametrische Typsysteme—auch *generische* Typsysteme genannt—heißen so, weil die Gleichförmigkeit der Typen durch Typparameter erreicht wird. Das heißt, Typausdrücke können Typparameter enthalten, für die Typausdrücke eingesetzt werden. Zum Beispiel kann im Ausdruck `List`$[a] \to a$, dem Typ aller Funktionen von `List`$[a]$ in a, für den Typparameter a der Typausdruck `Integer` eingesetzt werden. Das Ergebnis, `List`[`Integer`] $\to$ `Integer`, ist der (generierte) Typ einer Funktion, die Listen von ganzen Zahlen in ganze Zahlen abbildet. Ein Typausdruck mit freien Typparametern bezeichnet die Menge aller Typausdrücke, die durch Einsetzen von Typausdrücken generiert werden können. Anders gesagt, Typparameter werden implizit als universell über die Menge aller Typausdrücke quantifizierte Variablen betrachtet. Daher werden parametrische Typsysteme den *universell quantifizierten* Typsystemen zugerechnet.

Vererbung in objektorientierten Programmiersprachen ist mit *Subtyping* verwandt. (Wir verwenden hier den englischen Begriff, da es keinen adäquaten deutschen Begriff gibt.) Zum Beispiel hat der Typ `Person`, ein ADT, die beiden *Untertypen* `Student` und `Angestellter`. An eine Routine mit einem formalen Parameter vom Typ `Person` kann ein Argument vom Typ `Student` oder `Angestellter` übergeben werden, da jeder Student und jeder Angestellte auch eine Person ist. Die Menge der Instanzen von `Person` beinhaltet alle Instanzen von `Student` und `Angestellter`. Die Routine akzeptiert alle Argumente vom Typ t, wobei t eine universell über `Person` und dessen Untertypen quantifizierte Variable ist. Daher zählen auch Typsysteme mit Subtyping zu den universell quantifizierten Typsystemen.

Wenn ein ADT eine Routine (Methode) exportiert, dann exportiert auch jeder seiner Untertypen eine Routine mit einer dazu passenden Schnittstelle. Nicht die Routinen selbst, sondern nur deren Schnittstellen müssen übereinstimmen. Der Compiler sieht nur den statischen Typ einer Variablen (oder eines Parameters). Der dynamische Typ wird erst während der Ausführung festgelegt. Daher kann der Compiler auch nicht immer feststellen, welche konkrete Routine des in der Variable enthaltenen Wertes gegebenenfalls ausgeführt werden muss, da ja nur die Schnittstelle bekannt ist. Die Routine kann manchmal erst während der Programmausführung festgelegt werden. Dies ist als *dynamisches Binden* (engl. „dynamic binding“) bekannt. *Statisches Binden* (engl. „static binding“) bedeuted, dass bereits der Compiler die auszuführende Routine festlegt.

Im Gegensatz zu universell quantifizierten Typsystemen haben *ad hoc polymorphe* Systeme nicht notwendigerweise eine gleichförmige Struktur. Eine Routine ist ad hoc polymorph wenn sie Argumente mehrerer verschiedener Typen, die in keiner Relation zueinander stehen müssen, akzeptiert und sich für jeden dieser Typen anders verhalten kann. *Überladene* Typsysteme erlauben, dass ein und derselbe Name verschiedene Routinen bezeichnet, die sich durch die statischen Typen ihrer formalen Parameter unterscheiden. Die statischen Typen der übergebenen Argumente bzw. Variablen entscheiden, welche Funktion ausgeführt wird. Das Überladen dient häufig nur der syntaktischen Vereinfachung, da für Operationen mit ähnlicher Funktionalität nur ein gemeinsamer Name vorgesehen werden braucht. Zum Beispiel bezeichnet „/“ in vielen Programmiersprachen sowohl die ganzzahlige Division als auch die Division von Fließkommazahlen, obwohl diese Funktionen sich im Detail sehr stark voneinander unterscheiden.

Typumwandlung ist im Gegensatz zum Überladen eine semantische Operation. Sie dient hauptsächlich zur Umwandlung eines Wertes in ein Argument eines dynamischen Typs, der von einer Funktion erwartet wird. Zum Beispiel wird in C jede ganze Zahl des Typs `char`, `short` oder `long` bei der Argumentübergabe implizit in eine Zahl vom Typ `int` umgewandelt wenn der formale Parameter vom Typ `int` ist. Typumwandlungen können auch explizit erfolgen. Einige Programmier-

sprachen (z.B. C++) definieren durch oft recht diffizile Regeln, wie Typen von Argumenten umgewandelt werden, wenn zwischen mehreren überladenen Routinen gewählt werden kann.

1.3 Literaturhinweise

Die Definitionen des Begriffs Typ aus verschiedenen Sichtweisen wurden aus [29] übernommen. Dieser Artikel enthält eine thematisch umfangreiche Einführung in objektorientierte Programmiersprachen, in der auch Typen nicht zu kurz kommen. Es gibt zahlreiche allgemeine Einführungen in Programmiersprachen und deren Klassifikation. Als Beispiel sei [11] angeführt. Einige Klassifikationskriterien, insbesondere die Einteilung polymorpher Typsysteme, wurden aus [6] übernommen. Eine gute, verständliche Einführung in polymorphe Typsysteme wird auch in [8] gegeben.

Leider ist die Terminologie hinsichtlich der Klassifikation von Typsystemen nicht ganz einheitlich. Dies wird vor allem beim Lesen von Web-Seiten, die sich (auch) mit Programmiersprachen befassen, immer wieder deutlich. Vor allem die Begriffe „statisches Typsystem“ und „starkes Typsystem“ werden oft in zahlreichen unterschiedlichen Bedeutungen verwendet. Die in diesem Skriptum verwendete Terminologie ist aber insofern etabliert, als die meisten klassischen Bücher und Artikel auf diesem Gebiet (wie die oben zitierte Literatur) diese Terminologie verwenden.

Kapitel 2

Einfache theoretische Modelle

Wer Typsysteme genauer kennen und verstehen will, braucht grundlegendes Wissen über die Programmiersprachen, in die Typsysteme eingebettet sind, ebenso wie Wissen über die formalen Modelle, auf denen diese Typsysteme beruhen. Dieses Kapitel gibt einen kurzen Überblick sowohl über die wichtigsten Berechnungsmodelle als auch über einfache Typmodelle, auf denen komplexere Typsysteme aufbauen.

2.1 Der Lambda-Kalkül

Der Lambda-Kalkül ist wahrscheinlich das einflussreichste theoretische Modell der Informatik. Entsprechend umfangreiche Literatur über verschiedene Varianten des Lambda-Kalküls und seine Typsysteme gibt es. In diesem Abschnitt werden eine häufig verwendete untypisierte Variante und eine darauf aufbauende monomorph typisierte Variante beschrieben. Die typisierte Variante wird dann um einige strukturierte Typen erweitert, die in späteren Kapiteln verwendet werden. Ansätze für polymorphe Typsysteme im Lambda-Kalkül werden in Kapitel 4 behandelt.

2.1.1 Der untypisierte Lambda-Kalkül

Syntaktisch sind alle Elemente des Lambda-Kalküls *Lambda-Ausdrücke.* Wir definieren die Menge aller Lambda-Ausdrücke *Exp* folgendermaßen:

1. $x \in \mathit{Exp}$ wenn $x \in \mathit{Id}$; *Id* ist eine vorgegebene Bezeichner-Menge;
2. $(e_1\, e_2) \in \mathit{Exp}$ wenn $e_1 \in \mathit{Exp}$ und $e_2 \in \mathit{Exp}$;
3. $(\lambda x.e) \in \mathit{Exp}$ wenn $x \in \mathit{Id}$ und $e \in \mathit{Exp}$.

Runde Klammern veranschaulichen die Struktur von Lambda-Ausdrücken und können weggelassen werden wenn die Struktur eindeutig ist. Die Elemente der Menge *Id* sind einfach nur Namen, denen keine weitere Bedeutung beigemessen wird. Ausdrücke der Form $(e_1\, e_2)$ werden *Anwendungen* genannt, da e_1, als Funktion interpretiert, auf das Argument e_2 angewandt wird. Ausdrücke der Form $\lambda x.e$ heißen *Abstraktionen*; sie stellen einstellige Funktionen mit dem formalen Parameter x und dem Funktionsrumpf e dar.

Die Semantik des Lambda-Kalküls wird mittels Regeln bestimmt, die wiederholt auf Lambda-Ausdrücke angewandt werden um diese zu verändern. Veränderungen sind hauptsächlich *Ersetzungen* aller Vorkommen eines Namens x in einem Lambda-Ausdruck e_2 durch einen Lambda-Ausdruck e_1; das Ergebnis ist wieder ein Lambda-Ausdruck, der mit $[e_1/x]e_2$ (gelesen als „e_1 ersetzt x in e_2“) bezeichnet wird. Obwohl diese Definition von Ersetzung sehr einleuchtend erscheint, muss man aus mathematischer Sicht einen größeren Aufwand betreiben, um mögliche Namenskonflikte zu vermeiden. Zu diesem Zweck wird die Menge der *freien Variablen* $\mathit{free}(e)$ eines Lambda-Ausdrucks e definiert:

$$\begin{aligned} \mathit{free}(x) &= \{x\} \text{ für alle } x \in \mathit{Id} \\ \mathit{free}(e_1\, e_2) &= \mathit{free}(e_1) \cup \mathit{free}(e_2) \\ \mathit{free}(\lambda x.e) &= \mathit{free}(e) \setminus \{x\} \end{aligned}$$

Damit kann die Ersetzung $[e_1/x]e_2$ induktiv definiert werden:

$$[e/x_1]x_2 = \begin{cases} e & \text{für } x_1 = x_2 \\ x_2 & \text{für } x_1 \neq x_2 \text{ und } x_2 \in \mathit{Id} \end{cases}$$

$$[e_1/x](e_2\, e_3) = ([e_1/x]e_2)\,([e_1/x]e_3)$$

$$[e_1/x_1](\lambda x_2.e_2) = \begin{cases} \lambda x_2.e_2 & \text{für } x_1 = x_2 \\ \lambda x_2.[e_1/x_1]e_2 & \text{für } x_1 \neq x_2 \text{ und } x_2 \notin \mathit{free}(e_1) \\ \lambda x_3.[e_1/x_1]([x_3/x_2]e_2) & \text{sonst, wobei} \\ & x_1 \neq x_3 \neq x_2 \text{ und } x_3 \notin \mathit{free}(e_1) \cup \mathit{free}(e_2) \end{cases}$$

In der dritten Regel werden Namenskonflikte durch Namensänderungen beseitigt. Das folgende Beispiel zeigt die Anwendung aller drei Reglen:

$$[y/x](((\lambda y.x)\,(\lambda x.x))\,x) \equiv ((\lambda z.y)\,(\lambda x.x))\,y$$

Drei weitere Regeln genügen, um die Semantik des Lambda-Kalküls vollständig zu beschreiben:

1. α-Konversion (Umbenennung):

$$\lambda x_1.e \Longleftrightarrow \lambda x_2.[x_2/x_1]e \quad \text{wobei } x_2 \in \mathit{Id} \text{ und } x_2 \notin \mathit{free}(e)$$

Diese Regel erlaubt uns, Namen gebundener Variablen (= formale Parameter) beliebig gegen andere, noch nicht verwendete Namen auszutauschen. Namen formaler Parameter haben also keine globale Bedeutung.

2. β-Konversion (Anwendung):

$$(\lambda x.e_1)\, e_2 \Longleftrightarrow [e_2/x]e_1$$

Das ist die wichtigste Regel, die die Funktionsweise des Lambda-Kalküls zum Großteil erklärt. Das Argument e_2 einer Anwendung (eines Funktionsaufrufs) ersetzt jedes Vorkommen des formalen Parameters x im Rumpf e_1 der Funktion.

3. η-Konversion:

$$\lambda x.(e\, x) \Longleftrightarrow e \quad \text{für} \quad x \in \mathit{Id} \quad \text{und} \quad x \notin \mathit{free}(e)$$

Diese Regel wurde erst später zum Lambda-Kalkül dazugefügt. Sie beseitigt einige subtile Unvollständigkeiten im ursprünglichen Kalkül, auf die hier nicht weiter eingegangen wird. Wenn hinter $\lambda x.(e\, x)$ ein weiterer Ausdruck folgt, ist statt der η-Konversion stets mit gleichem Ergebnis auch die β-Konversion verwendbar.

Die gerichteten Varianten der beiden letztgenannten Konversionen, sie werden *Reduktionen* genannt, sind von größerer praktischer Bedeutung:

1. β-Reduktion:

$$(\lambda x.e_1)\, e_2 \Longrightarrow [e_2/x]e_1$$

2. η-Reduktion:

$$\lambda x.(e\, x) \Longrightarrow e \quad \text{für} \quad x \in \mathit{Id} \quad \text{und} \quad x \notin \mathit{free}(e)$$

Ein Lambda-Ausdruck ist in *Normalform* wenn er weder mit β-Reduktion noch mit η-Reduktion weiter reduzierbar ist. Nicht für jeden Lambda-Ausdruck gibt es einen entsprechenden Ausdruck in Normalform. Zum Beispiel ist im Ausdruck

$$(\lambda x.(x\, x))\, (\lambda x.(x\, x))$$

nur eine β-Reduktion anwendbar, die zum selben Ausdruck führt. Wenn es für einen Ausdruck aber eine Normalform gibt, dann gilt folgender Satz:

> Kein Lambda-Ausdruck kann in zwei verschiedene Normalformen konvertiert werden, wenn man Namensunterschiede aufgrund von α-Konversionen unberücksichtigt lässt.

Ein weiterer Satz, das *Fixpunkt-Theorem*, erlaubt es, im Lambda-Kalkül Rekursion auszudrücken:

> Jeder Lambda-Ausdruck e hat einen Fixpunkt e', sodass $(e\, e')$ zu e' konvertiert werden kann.

Für dieses Theorem gibt es einen überraschend einfachen Beweis, der hier als Beispiel für Konversionen im Lambda-Kalkül angeführt wird:

Für e' nehmen wir $(Y\, e)$, wobei

$$Y \equiv \lambda f.((\lambda x.(f\, (x\, x)))\, (\lambda x.(f\, (x\, x))))$$

der sogenannte Y-Kombinator ist. Dann gilt

$$\begin{aligned} (Y\, e) &= (\lambda x.(e\, (x\, x)))\, (\lambda x.(e\, (x\, x))) \\ &= e\, ((\lambda x.(e\, (x\, x)))\, (\lambda x.(e\, (x\, x)))) \\ &= e\, (Y\, e) \end{aligned}$$

In der Ableitung wurde zweimal β-Reduktion auf dem jeweiligen äußeren Ausdruck angewandt.

Jede rekursive Funktion kann im Lambda-Kalkül leicht durch eine rekursionsfreie (und nichtiterative) Funktion ersetzt werden. Nehmen wir an, f sei eine rekursive Funktion, die folgendermaßen definiert sei:

$$f \equiv \cdots f \cdots$$

Dieser Ausdruck kann leicht umgeschrieben werden in

$$f \equiv (\lambda f.\ \cdots f \cdots)\, f$$

sodass das innere Vorkommen von f nun gebunden ist. Diese Gleichung besagt, dass f ein Fixpunkt des Lambda-Ausdrucks $(\lambda f.\ \cdots f \cdots)$ ist. Das ist genau das, was Y berechnen kann. Wir erhalten also folgende rekursionsfreie Definition von f:

$$f \equiv Y\, (\lambda f.\ \cdots f \cdots).$$

Die Mächtigkeit des Lambda-Kalküls wird durch *Churchs These* angedeutet:

> Genau jene Funktionen von den natürlichen Zahlen in die natürlichen Zahlen sind effektiv berechenbar, die im Lambda-Kalkül ausgedrückt werden können.

Die Bedeutung dieser These wird sofort ersichtlich, wenn man weiss, dass auf den Begriff der „effektiven Berechenbarkeit“ der Satz über die Turing-Vollständigkeit anwendbar ist. Das heißt, alles was berechenbar ist, ist auch im Lambda-Kalkül berechenbar.

Eine abschließende Bemerkung zur Syntax: Funktionen im Lambda-Kalkül sind auf ein Argument beschränkt. Mehrstellige Funktionen können leicht durch „geschachtelte“ Ausdrücke der Form $\lambda x.e$ dargestellt werden. So schreibt man im Lambda-Kalkül—statt der in der Mathematik üblichen Schreibweise $f(a, b)$—für eine zweistellige Funktion $(f\, a\, b)$ oder geklammert $((f\, a)\, b)$ oder etwas ausführlicher $(((\lambda x.(\lambda y.e))\, a)\, b)$.

2.1.2 Der typisierte Lambda-Kalkül

Wir wollen den untypisierten Lambda-Kalkül um Typen erweitern. Dazu führen wir die Mengen *BasTyp* von Basistypen (z.B. `Boolean`, `Integer` und `String`) und *Typ* von strukturierten Typen ein. (*BasTyp* ist eine Teilmenge von *Typ*.) In diesem Abschnitt wird implizit ein monomorphes Typsystem verwendet. Daher können wir der Einfachheit halber annehmen, dass jeder Typ eine Menge von Werten beschreibt und je zwei verschiedene Typen keine gemeinsamen Werte beschreiben.[1] Jedem Lambda-Ausdruck wird ein Typ, das ist das Element von *Typ*, in dem der durch den Lambda-Ausdruck festgelegte Wert enthalten ist, zugeordnet. Wir schreiben $e{:}t$ wenn der Lambda-Ausdruck e den Typ $t \in \mathit{Typ}$ hat. Der strukturierte Typ $t_1{\to}t_2$ bezeichnet den Typ aller Funktionen von Werten des Typs t_1 in Werte des Typs t_2.

Beim Übergang vom untypisierten in den typisierten Lambda-Kalkül sind Modifikationen der Syntax und Semantik notwendig. Wir definieren daher einige Begriffe um. In den folgenden Abschnitten werden wir, wenn nichts anderes gesagt wird, die Definitionen des typisierten Lambda-Kalküls verwenden.

Die Menge *BasTyp* ist als gegeben angenommen. Die Menge *Typ* ist induktiv definiert:

1. $b \in \mathit{Typ}$ wenn $b \in \mathit{BasTyp}$;
2. $t_1{\to}t_2 \in \mathit{Typ}$ wenn $t_1 \in \mathit{Typ}$ und $t_2 \in \mathit{Typ}$.

Die Menge *Id* von typisierten Bezeichnern der Form $x{:}t$ mit $t \in \mathit{Typ}$ ist als gegeben angenommen. Die Menge *Exp* aller typisierten Lambda-Ausdrücke ist induktiv definiert:

1. $x{:}t \in \mathit{Exp}$ wenn $x{:}t \in \mathit{Id}$;
2. $(e_1{:}t_2{\to}t_1\ e_2{:}t_2){:}t_1 \in \mathit{Exp}$ wenn $e_1{:}t_2{\to}t_1 \in \mathit{Exp}$ und $e_2{:}t_2 \in \mathit{Exp}$;
3. $(\lambda x{:}t_2.e{:}t_1){:}t_2{\to}t_1 \in \mathit{Exp}$ wenn $x{:}t_2 \in \mathit{Id}$ und $e{:}t_1 \in \mathit{Exp}$.

Die Konversions-Regeln und entsprechenden Reduktions-Regeln sind analog zu denen im untypisierten Lambda-Kalkül definiert:

1. α-Konversion:

$$(\lambda x_1{:}t_2.e{:}t_1){:}t_2{\to}t_1 \iff (\lambda x_2{:}t_2.[x_2/x_1]e{:}t_1){:}t_2{\to}t_1$$
wobei $x_2{:}t_2 \in \mathit{Id}$ und $x_2 \notin \mathit{free}(e)$

2. β-Konversion:

$$((\lambda x{:}t_2.e_1{:}t_1){:}t_2{\to}t_1\ e_2{:}t_2){:}t_1 \iff [e_2/x]e_1{:}t_1$$

3. η-Konversion:

$$(\lambda x{:}t_2.(e{:}t_2{\to}t_1\ x{:}t_2){:}t_1){:}t_2{\to}t_1 \iff e{:}t_2{\to}t_1$$
wenn $x{:}t_2 \in \mathit{Id}$ und $x \notin \mathit{free}(e)$

Die Einführung von Typen in den Lambda-Kalkül hat schwerwiegende Auswirkungen auf seine Mächtigkeit. Für jeden Ausdruck im typisierten Lambda-Kalkül gibt es genau eine Normalform, die effektiv berechenbar ist. Folglich ist es aber nicht möglich, alle effektiv berechenbaren Funktionen darzustellen. Insbesondere kann im typisierten Lambda-Kalkül kein Fixpunktoperator definiert werden, sodass rekursive Funktionen nicht berechenbar sind. Der Grund dafür ist, dass im Lambda-Ausdruck $(e\,e)$ der Teilausdruck e sowohl einen Typ $t_2{\to}t_1$ als auch t_2 haben muss, was in den oben definierten Strukturen ausgeschlossen ist.

Glücklicherweise gibt es einen Ausweg aus diesem Dilemma. Anstatt sich auf die Selbstanwendung zu verlassen, kann man den Lambda-Kalkül um eine vierte Regel erweitern, die typisierte Versionen des Y-Kombinators implementiert. Für jeden Typ $t \in \mathit{Typ}$ definieren wir einen *typisierten Fixpunktoperator* Y_t vom Typ $(t{\to}t){\to}t$, den wir als Konstante zum typisierten Lambda-Kalkül hinzufügen. Die Semantik der Fixpunktoperatoren wird durch eine δ-Konversionsregel festgelegt:

$$(Y_t{:}(t{\to}t){\to}t\ e{:}t{\to}t){:}t \iff (e{:}t{\to}t\ (Y_t{:}(t{\to}t){\to}t\ e{:}t{\to}t){:}t){:}t$$

Wenn man die Typinformation ignoriert kann man leicht feststellen, dass diese δ-Regel der Konversion $(Y\,f) \iff (f\,(Y\,f))$ im untypisierten Kalkül entspricht. Zur Implementierung der Rekursion kann derselbe Trick wie in Abschnitt 2.1.1 verwendet werden.

2.1.3 Strukturierte Typen

In Abschnitt 2.1.2 wurden strukturierte Typen nur für Funktionen verwendet. Mit Hilfe der Technik, die zur Einführung der typisierten Fixpunktoperatoren verwendet wurde, nämlich dem Dazufügen von δ-Regeln[2] zum Lambda-Kalkül, kann eine Reihe weiterer strukturierter Typen definiert werden. Genaugenommen werden nicht die Typen selbst, sondern Zugriffsfunktionen auf und Erzeugungsfunktionen von Instanzen strukturierter Typen über gerichtete δ-Regeln definiert. Im Prinzip kann man mit δ-Regeln den Lambda-Kalkül beliebig verändern. Allerdings ist dabei größte Vorsicht

[1] Tatsächlich müssen als Instanzen der Typen etwas kompliziertere Strukturen als nur Mengen von Werten verwendet werden, um grundlegende Eigenschaften des Lambda-Kalküls beibehalten zu können. Eine Diskussion dieses Themas würde den Rahmen des Skriptums aber bei Weitem sprengen.

[2] Der Begriff δ-Regel steht einfach für alle weiteren Regeln außer den α-, β- und η-Regeln.

geboten, wenn man grundlegende Eigenschaften des Lambda-Kalküls beibehalten will. Die hier beschriebenen Erweiterungen haben sich als einigermaßen gut verträglich mit dem ursprünglichen Lambda-Kalkül erwiesen und werden in vielen imperativen und funktionalen Sprachen verwendet.

Zu den einfachsten strukturierten Typen gehören jene für *kartesische Produkte* (Kreuzprodukte): Für je zwei Typen t_1 und t_2 in *Typ* enthält *Typ* auch den Typ $t_1 \times t_2$. Beispiele sind `Integer`×`Boolean` und (`Integer`→`Integer`)×(`Integer`×`Boolean`). Instanzen dieses Typs heißen *Paare* und werden in der Form (e_1, e_2) dargestellt. Auf ihnen sind die Zugriffsfunktionen `first` und `rest` sowie die Erzeugungsfunktion `cons` definiert:

$$(\texttt{first}{:}(t_1 \times t_2) \to t_1\ (e_1, e_2){:}t_1 \times t_2){:}t_1 \\ \quad \Longrightarrow e_1{:}t_1$$

$$(\texttt{rest}{:}(t_1 \times t_2) \to t_2\ (e_1, e_2){:}t_1 \times t_2){:}t_2 \\ \quad \Longrightarrow e_2{:}t_2$$

$$((\texttt{cons}{:}t_1 \to (t_2 \to (t_1 \times t_2))\ e_1{:}t_1){:}t_2 \to (t_1 \times t_2)\ e_2{:}t_2){:}t_1 \times t_2 \\ \quad \Longrightarrow (e_1, e_2){:}t_1 \times t_2$$

Beachten Sie in der δ-Regel für `cons`, dass ein Typ der Form $t_1 \to (t_2 \to t_3)$ nicht nur für eine Funktion steht, die ein Argument vom Typ t_1 nimmt und als Ergebnis eine Funktion von t_2 in t_3 liefert, sondern auch für eine Funktion, die zwei Argumente der Typen t_1 und t_2 nimmt und als Ergebnis eine Instanz des Typs t_3 liefert. Eigentlich handelt es sich bei `first`, `rest` und `cons` um drei Familien von Funktionen, da die Funktionsnamen, wie bei den typisierten Fixpunktoperatoren, mit den jeweiligen Typen als Indizes versehen werden müssten. Wir verzichten zugunsten einer besseren Lesbarkeit auf das Anschreiben dieser Indizes.

Eine weitere strukturierte Datenstruktur ist der *Verbund* (engl. *„record“*). Das ist eine ungeordnete Menge indizierter Werte. Als Indexmenge kann jede beliebige Menge genommen werden. Jedes Element dieser Menge wird als *Label* bezeichnet. Der Typ eines Wertes im Verbund wird durch den Typ bestimmt, der dem Label des Wertes zugeordnet ist. Ein Verbundtyp wird in geschwungenen Klammern als Folge von durch Komma getrennten, typisierten Labels angeschrieben. Ein Beispiel ist {`a`:`Integer`, `b`:`Boolean`, `c`:`String`}. Einen Verbund dieses Typs erzeugt man, indem man jedem Label einen Wert zuordnet, wie dieses Beispiel zeigt:

$$\{\texttt{a}{=}3, \texttt{b}{=}\texttt{true}, \texttt{c}{=}\texttt{"abcd"}\}{:} \\ \quad \{\texttt{a}{:}\texttt{Integer}, \texttt{b}{:}\texttt{Boolean}, \texttt{c}{:}\texttt{String}\}$$

Die einzige Operation auf Verbunden ist die Auswahl eines Feldes, dargestellt durch die übliche Punkt-Notation ($i \in \{1, \ldots, n\}$):

$$\{l_1{=}e_1, \ldots, l_n{=}e_n\}{:}\{l_1{:}t_1, \ldots, l_n{:}t_n\}.l_i \Longrightarrow e_i{:}t_i$$

Tupel kann man als Verallgemeinerung von Paaren und Spezialisierung von Verbunden sehen: Zum Beispiel ist ein Tupel `(3, true, "abcd")` vom Typ `Integer` × `Boolean` × `String` äquivalent zu einem Verbund {`1`=3, `2`=`true`, `3`=`"abcd"`} vom entsprechenden Typ {`1`:`Integer`, `2`:`Boolean`, `3`:`String`}.

Als letzter strukturierter Typ wird in diesem Abschnitt der *Variantentyp* eingeführt. Syntaktisch ist ein Variantentyp genauso wie ein Verbundtyp definiert, außer dass (statt der geschwungenen) eckige Klammern verwendet werden. Ein Beispiel für einen Variantentyp ist [`a`:`Integer`, `b`:`Boolean`, `c`:`String`]. Jede Instanz dieses Typs ordnet genau einem der Label einen Wert zu. Drei mögliche Instanzen des Beispieltyps sind also

$$[\texttt{a}{=}3]{:}[\texttt{a}{:}\texttt{Integer}, \texttt{b}{:}\texttt{Boolean}, \texttt{c}{:}\texttt{String}] \\ [\texttt{b}{=}\texttt{true}]{:}[\texttt{a}{:}\texttt{Integer}, \texttt{b}{:}\texttt{Boolean}, \texttt{c}{:}\texttt{String}] \\ [\texttt{c}{=}\texttt{"abcd"}]{:}[\texttt{a}{:}\texttt{Integer}, \texttt{b}{:}\texttt{Boolean}, \texttt{c}{:}\texttt{String}]$$

Die einzige Operation auf Varianten ist die Mehrfachverzweigung (engl. *„case statement“*). Seine Syntax und Semantik sind am besten aus der entsprechenden δ-Regel ersichtlich ($i \in \{1, \ldots, n\}$):

$$(\texttt{case}\ [l_i{=}e_i]{:}[l_1{:}t_1, \ldots, l_n{:}t_n]\ \texttt{of} \\ \quad l_1 \Rightarrow f_1{:}t_1 \to t, \ldots, l_n \Rightarrow f_n{:}t_n \to t){:}t \\ \Longrightarrow (f_i{:}t_i \to t\ e_i{:}t_i){:}t$$

Eine Mehrfachverzweigung gibt für jedes Label eines Variantentyps eine entsprechend typisierte Funktion an. Je nach dem Label der in der Mehrfachverzweigung angegebenen Instanz dieses Typs wird eine Funktion ausgewählt. An diese Funktion wird der Wert der Instanz übergeben.

2.2 Logik

Logiken, insbesondere verschiedene Formen der Klausel-Logik, für die automatische Beweissysteme existieren, spielen eine zweifache Rolle im Zusammenhang mit Typen. Einerseits gibt es logische Programmiersprachen, in die Typen eingebaut wurden, um verschiedene Arten von Programmierfehlern leichter finden zu können. Andererseits kann man Logiken zur Spezifikation von Typen verwenden.

2.2.1 Die Prädikatenlogik erster Stufe

Die Prädikatenlogik erster Stufe ist eine Grundlage für die meisten Logiken. Durch die Beschränkung auf die erste Stufe können in der Logik selbst keine Aussagen über die Variablen der Logik getroffen werden; es gibt keine Metavariablen. *Wohlgeformten Formeln* (= Ausdrücke) der Prädikatenlogik erster Stufe sind folgendermaßen aufgebaut. [Übliche Schreibweisen sind in eckigen Klammern angeführt.]

1. Vorgegebene disjunkte Mengen:
   (a) eine Menge von *Variablen* V $[x, y, z]$;
   (b) eine Menge von *Funktionssymbolen* F $[a^{(n)}, b^{(n)}, \ldots; 0 \leq n$ gibt die *Stelligkeit* an]; 0-stellige Funktionssymbole werden *Konstanten* genannt;
   (c) eine Menge von *Prädikaten* P $[p^{(n)}, q^{(n)}, \ldots; 0 \leq n$ gibt wieder die Stelligkeit an].

2. Die Menge aller *Terme* $[u, v, w]$ wird induktiv konstruiert: $x \in V$ ist ein Term; und $a(u_1, \ldots, u_n)$ ist ein Term wenn $a^{(n)} \in F$ und $u_1, \ldots, u_n$ Terme sind. [Bei $n = 0$ werden die Klammern weggelassen.] Ein Term, der keine Variablen enthält, heißt *Grundterm.*

3. $p(u_1, \ldots, u_n)$ ist eine *Atomformel* $[A, B, \ldots]$ wenn $p^{(n)} \in P$ ein Prädikat ist und $u_1, \ldots, u_n$ Terme sind. [Bei $n = 0$ werden die Klammern weggelassen.] Wenn $u_1, \ldots, u_n$ keine Variablen enthalten, heißt $p(u_1, \ldots, u_n)$ *Grundatomformel.*

4. Die Menge aller *wohlgeformten Formeln* der Prädikatenlogik $[U, V, W, \ldots]$ ist induktiv definiert:
   (a) Eine Atomformel ist ein wohlgeformte Formel.
   (b) Wenn U und V wohlgeformte Formeln sind, dann sind auch $\neg U$, $U \wedge V$, $U \vee V$, $U \rightarrow V$, $U \leftarrow V$ und $U \leftrightarrow V$ wohlgeformte Formeln. Die *logischen Operatoren* stehen (in der Reihenfolge ihres Auftretens) für Negation, Konjunktion (Und), Disjunktion (Oder), Implikation nach rechts bzw. links und Äquivalenz.
   (c) Wenn U eine wohlgeformte Formel und x eine Variable ist, dann sind auch $\forall x.U$ und $\exists x.U$ wohlgeformte Formeln. $\forall$ ist der universelle und $\exists$ der existenzielle Quantor. Eine wohlgeformte Formel heißt *geschlossen,* wenn alle darin vorkommenden Variablen im Bereich eines Quantors liegen.

Ein *prädikatenlogisches Modell* ist eine Abbildung aller wohlgeformten Formeln in die Menge der Wahrheitswerte $\{\texttt{true}, \texttt{false}\}$. Das heißt, jede wohlgeformte Formel entspricht im Modell einer wahren oder falschen Aussage. Modelle müssen den vordefinierten Bedeutungen der logischen Operatoren und Quantoren, die als allgemein bekannt vorausgesetzt werden, entsprechen. Hier sind zur Erinnerung einige Eigenschaften angeführt:

$$\begin{aligned}
\neg\neg U &\Leftrightarrow U \\
U \wedge (\neg U) &\Leftrightarrow \texttt{false} \\
\neg(U \wedge V) &\Leftrightarrow (\neg U) \vee (\neg V) \\
\neg(\forall x.U) &\Leftrightarrow \exists x.(\neg U) \\
U \wedge (V \vee W) &\Leftrightarrow (U \wedge V) \vee (U \wedge W) \\
U \rightarrow V &\Leftrightarrow (\neg U) \vee V \\
U \leftrightarrow V &\Leftrightarrow (U \rightarrow V) \wedge (U \leftarrow V)
\end{aligned}$$

Logische Programmiersprachen wie z.B. Prolog verwenden als Modell das *Herbrand-Modell*, in dem jeder Grundterm und jede Grundatomformel als dieser Grundterm bzw. diese Grundatomformel selbst interpretiert werden. Daher bestimmen ausschließlich die logischen Operatoren und Quantoren, ob eine Aussage aus einer Menge als wahr angenommener wohlgeformter Formeln ableitbar ist. In neueren logischen Sprachen und „constraint"-Sprachen werden teilweise auch andere Modelle verwendet.

2.2.2 Definite Horn-Klausel-Logik

Definite Horn-Klauseln, hier kurz Klauseln genannt, sind eine relativ mächtige Teilmenge der Prädikatenlogik erster Stufe. Eine solche Klausel ist eine geschlossene wohlgeformte Formel der Form

$$\forall x_1. \cdots \forall x_m.(A \vee \neg B_1 \vee \cdots \vee \neg B_n)$$

wobei $A, B_1, \ldots, B_n$ Atomformeln und $x_1, \ldots, x_m$ alle darin vorkommenden Variablen sind $(m, n \geq 0)$. Der Einfachheit halber werden Klauseln meist in der Form

$$A \leftarrow B_1, \ldots, B_n$$

angeschrieben; das heißt, Quantoren werden weggelassen, die Äquivalenz zwischen $A \vee \neg B_1 \vee \cdots \vee \neg B_n$ und $A \leftarrow (B_1 \wedge \cdots \wedge B_n)$ wird ausgenützt, und Konjunktion wird mittels Beistrich ausgedrückt. Die Aussage einer Klausel kann in Deutsch etwa so umschrieben werden: „Wenn es Belegungen für alle Variablen in $A, B_1, \ldots, B_n$ gibt, sodass B_1 bis B_n alle zugleich wahr sind, dann ist auch A mit diesen Variablenbelegungen wahr." A wird als *Kopf* und $B_1, \ldots, B_n$ als *Rumpf* der Klausel bezeichnet. Eine Klausel mit leerem Rumpf $(n = 0)$ heißt *Faktum*, eine mit nichtleerem Rumpf heißt *Regel.* Ein *definites Programm* ist eine Menge definiter Horn-Klauseln, die als wahr angenommen werden.

Eine logische Programmiersprache kann *Ziele* (das sind Atomformeln) aus einem logischen Programm (das ist eine Menge als wahr angenommener Klauseln) ableiten. Welche Ziele sind nun ableitbar? Wir versuchen, diese Frage vorerst nur für Grundatomformeln zu beantworten. Eine Klausel, die Variablen enthält, kann man als eine Menge von Klauseln ohne Variablen auffassen; in jedem Element dieser Menge sind alle Variablen mit Grundtermen belegt. Diese Menge ist fast

immer unendlich groß, da es unendliche viele mögliche Variablenbelegungen mit Grundtermen gibt. Wir nennen das Programm, das dadurch entsteht, dass alle Klauseln mit Variablen durch alle ihnen entsprechenden Klauseln ohne Variablen ersetzt werden, *GP.* Es ist leicht einsichtig, dass der Kopf jedes Faktums in GP eine auch im ursprünglichen Programm ableitbare Grundatomformel ist. Wir bezeichnen die Menge dieser aus Fakten ableitbaren Grundformeln mit R_0. Aus jedem R_i mit $0 \leq i$ kann leicht die Menge von ableitbaren Grundformeln R_{i+1} errechnet werden:

$$R_{i+1} \quad = \quad R_i \ \cup \ \{A \mid (A \leftarrow B_1, \ldots, B_n) \in GP \\ \wedge \ B_1 \in R_i \ \wedge \cdots \wedge \ B_n \in R_i\}$$

Die Menge aller aus dem ursprünglichen Programm ableitbaren Grundatomformeln ist der kleinste Fixpunkt dieser Iteration, das ist jenes R_m (mit kleinstmöglichem m) für welches $R_m = R_{m+1}$ gilt.

Im vorigen Absatz wurde eine einfache „bottom-up"-Methode zur Ableitung von Zielen beschrieben. Viele logische Programmiersprachen, wie z.B. Prolog, verwenden jedoch eine „top-down"-Methode. Eine wichtige Teilaufgabe ist dabei die *Unifikation.* Zwei Terme sind unifizierbar, wenn die Variablen in den beiden Termen einheitlich so mit Termen belegt werden können, dass die beiden Terme gleich werden. Eine Menge von Variablenbindungen, die dies zustande bringt, heißt *Unifikator.* Ein Unifikator, der nur die für die Unifikation unbedingt notwendigen Variablenbindungen enthält, heißt *allgemeinster Unifikator.* Ebenso können zwei Atomformeln unifiziert werden wenn sie das gleiche Prädikat haben und ein allgemeinster Unifikator existiert, der die Argumentterme der beiden Atomformeln gleich macht, sodass auch die Atomformeln selbst gleich werden. Wenn θ ein allgemeinster Unifikator von zwei Atomformeln A und B ist, dann schreiben wir formal θA bzw. θB für die Atomformeln die entstehen, wenn alle Variablenbindungen von θ auf A bzw. B angewandt werden. Natürlich gilt $\theta A = \theta B$. Die Antwort auf eine Anfrage, nennen wir das Ziel G, wird so berechnet: Es werden Mengen S_i noch ungelöster Atomformeln berechnet, wobei der Index i über die einzelnen Schritte der Berechnung läuft. Die Anfangsmenge $S_0 = \{G\}$ enthält nur das Ziel. Die Menge S_i $(0 < i)$ wird aus $S_{i-1} = \{G_1, \ldots, G_j, \ldots, G_m\}$ folgendermaßen erzeugt: Es werden eine Atomformel G_j $(1 \leq j \leq m)$ aus S_i und eine Klausel $A \leftarrow B_1, \ldots, B_n$ aus dem Programm gewählt, sodass G_j mit A mittels eines allgemeinsten Unifikators θ_i unifizierbar ist; S_i ist dann die Menge

$$\{\theta_i G_1, \ldots, \theta_i G_{j-1}, \theta_i B_1, \ldots, \theta_i B_n, \theta_i G_{j+1}, \ldots, \theta_i G_n\}.$$

In anderen Worten, in jedem Schritt wird eine noch unbewiesene Atomformel aus der Menge gewählt und durch eine Bedingung—ausgedrückt durch eine (möglicherweise leere) Menge unbewiesener Atomformeln—, unter der die zu beweisende Atomformel wahr ist, ersetzt. Wenn es einen Index k gibt, sodass S_k die leere Menge ist, dann ist $\theta_k \cdots \theta_1 G$ aus dem Programm ableitbar. Die Antwort enthält also nicht nur die prinzipielle Aussage, dass es Variablenbelegungen gibt, unter denen G ableitbar ist, sondern liefert solche Variablenbindungen gleich mit. Wenn S_k nicht leer ist, aber im Programm keine Klausel vorkommt deren Kopf mit einer Atomformel in S_k unifizierbar ist, dann ist der Beweisversuch gescheitert. G könnte aber trotzdem beweisbar sein und der Beweis wäre gefunden worden wenn in einem früheren Schritt eine andere Klausel gewählt worden wäre. In Prolog werden in diesem Fall systematisch die letzten Berechnungsschritte zurückgenommen und mit einer anderen Klausel wiederholt, bis ein Beweis gefunden ist oder keine alternative Klausel mehr existiert. Diesen Vorgang nennt man *„backtracking".*

Die Ableitung von Zielen aus logischen Programmen ist Halbentscheidbar. Es gibt Ziele, die weder ableitbar noch widerlegbar sind. In unentscheidbaren Fällen kommt die Programmausführung in eine Endlosschleife. Es ist jedoch im Allgemeinen nicht entscheidbar, ob sich eine Ausführung in einer Endlosschleife befindet oder nur mehr Ressourcen braucht, um zu einem Ergebnis zu kommen.

2.2.3 Typisierte logische Programme

Typen können in logischen Sprachen im Prinzip auf die gleiche Weise eingeführt werden wie im Lambda-Kalkül. Wir können annehmen, dass ein Typ eine Menge von Termen beschreibt.

Der auf den ersten Blick einfachste Ansatz ist, jeder Variable in einem logischen Programm einen Typ zuzuordnen. Zum Beispiel schreiben wir $x{:}t$ für eine Variable x vom Typ t. Typkompatibilität bedeutet dann einfach, dass jedes Vorkommen einer Variable denselben Typ hat und als Variablenbelegungen nur Terme aus den Typen der Variablen in Frage kommen. Die Variablen sind also nicht über alle Terme, sondern nur über die Menge T_i der zum jeweiligen Typ t_i gehörenden Terme quantifiziert $(1 \leq i \leq m)$:

$$\forall x_1 \in T_1. \cdots \forall x_m \in T_m.(A \leftarrow (B_1 \wedge \cdots \wedge B_n))$$

Während jeder Unifikation muss überprüft werden, ob die an Variablen gegundenen Terme mit dem Typ der Variablen kompatibel sind. Falls diese Überprüfung fehlschlägt, wird mittels „backtracking" ein anderer Lösungsweg gesucht.

Daraus kann man einen formal anderen, aber inhaltlich identischen Ansatz ableiten: Jeder Typ wird über ein einstelliges Prädikat definiert, das für genau jene

Terme wahr ist, die zum Typ gehören. Damit kann obige Klausel äquivalent in folgender Form ausgedrückt werden, wobei $t_i^{(1)}$ das Prädikat ist, das den Typ t_i beschreibt ($1 \leq i \leq m$):

$$\forall x_1. \cdots \forall x_m.(A \leftarrow (t_1(x_1) \wedge \cdots \wedge t_m(x_m) \wedge B_1 \wedge \cdots \wedge B_n))$$

oder in abgekürzter Schreibweise

$$A \leftarrow t_1(x_1), \ldots, t_m(x_m), B_1, \ldots, B_n.$$

Daraus ergibt sich eine Reihe weiterer Möglichkeiten. Es muss zum Beispiel nicht jede Variable einen bestimmten Typ haben, und beliebig komplexe Typen—auch solche, die von mehreren Argumenten abhängen—können definiert werden. Andererseits stellt sich dabei die Frage, wodurch sich Typen von gewöhnlichen Prädikaten unterscheiden. Eine mögliche Antwort darauf wäre, dass es tatsächlich keinen Unterschied gibt. Ein logisches Programm kann als eine Menge von Typspezifikationen—also als Typsystem—gesehen werden, und das Ergebnis einer Anfrage besagt, ob und unter welchen Bedingungen ein Ziel den Typspezifikationen entspricht. Ein Prolog-Interpreter ist demgemäß ein einziges großes Typüberprüfungssystem und Prädikatenlogik eine mächtige Typbeschreibungssprache.

Nicht jeder gibt sich mit dieser simplifizierenden Antwort zufrieden. Es ist ja technisch kein Problem, Typspezifikationen und sonstige Atomformeln in einer Klausel syntaktisch zu trennen. Genauso einfach kann man sicherstellen, dass für jede Variable eine Atomformel zur Typüberprüfung eingeführt wird und jedes Argument aller Vorkommen eines Prädikats im Programm denselben Typ hat. Eine Motivation dafür ist, dass ein Typfehler von Natur aus etwas anderes als eine Einschränkung der Lösungsmenge ist. Einer aufgetretenen Typinkompatibilität sollte also nicht stillschweigend durch „backtracking“ begegnet werden, sondern eine unübersehbare Fehlermeldung sollte auf den Programmfehler aufmerksam machen. Die verschiedenen Ansätze in diese Richtung unterscheiden sich in der Syntax, der Flexibilität und im Zeitpunkt der Typüberprüfung. Praktisch alle Ansätze erlauben polymorphe Typen, die wir in Kapitel 4 behandeln werden.

2.3 Algebren

Algebra ist ein Begriff aus der Mathematik. Eine *universelle Algebra* ist ein Paar $\langle A, \Omega \rangle$, wobei die *Trägermenge* A eine beliebige nichtleere Menge und Ω ein System von *Operationen* auf A ist. Unter dem *Typ* einer Algebra versteht man in der Mathematik die Stelligkeiten der Operationen in Ω. Diese Typen sind nur entfernt mit Typen in Programmiersprachen verwandt. Zum Beispiel ist der Ring über den ganzen Zahlen $\langle \mathbb{Z}, \{+, \cdot, -, 0\} \rangle$ eine Algebra des Typs $\langle 2, 2, 1, 0 \rangle$, da $+$ und $\cdot$ binäre Operationen, die Negation ($-$) eine unäre Operation und 0 eine nullstellige Operation bzw. eine ausgewählte Konstante ist. In dieser Algebra gelten neben anderen auch folgende Gesetze:

$$(a + b) + c = a + (b + c) \qquad a + b = b + a$$
$$a + 0 = a \qquad a + (-a) = 0$$
$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Alle in Ringen geltenden Gesetze lassen sich aus solchen Gesetzen herleiten. Diese Gesetze bestimmen das Verhalten der Operationen. Man könnte statt den ganzen Zahlen genausogut die Menge $\mathbb{R}$ der reellen Zahlen einsetzen, ohne dass sich die Gesetze dadurch ändern würden. Für die Beschreibung des Verhaltens der Operationen ist es also gar nicht nötig, die Trägermenge zu kennen. In der Mathematik bezeichnet der Begriff *Varietät*, angeschrieben als $V(\Omega, \Delta)$, eine Familie universeller Algebren, die durch eine Menge von Operationen Ω und eine Menge dazugehöriger Gesetze Δ bestimmt ist. Algebren in $V(\Omega, \Delta)$ können sich durch ihre Trägermengen unterscheiden, und in einigen Algebren können noch weitere, nicht aus Δ ableitbare Gesetze gelten.

2.3.1 Abstrakte Datentypen

Aufgrund ihrer Eigenschaften werden universelle Algebren und deren Varietäten manchmal als theoretische Grundlage für abstrakte Datentypen (ADT) verwendet. Die Menge der Operationen Ω und deren Typen (= Stelligkeiten) werden zusammen als *Signatur* bezeichnet, die im Wesentlichen der Schnittstellenbeschreibung eines ADT entspricht. Die Menge der Gesetze Δ spezifiziert das Verhalten des ADT nur unvollständig, da in einigen Algebren der Varietät $V(\Omega, \Delta)$ neben Δ weitere Gesetze gelten können. Von besonderer praktischer Bedeutung sind daher die sogenannten *freien Algebren* in $V(\Omega, \Delta)$, in denen außer den Gesetzen in Δ und den daraus ableitbaren Gesetzen keine weiteren Gesetze gelten. Im nächsten Beispiel wird der ADT `WW` (Wahrheitswert) spezifiziert:

Operationen:
$\Omega = \{$`wahr`, `falsch`, `nicht`, `und`, `oder`$\}$
Typ der Algebra: $\langle 0, 0, 1, 2, 2 \rangle$
Gleichungen:

```
nicht(wahr) = falsch
nicht(falsch) = wahr
und(w, wahr) = w
und(w, falsch) = falsch
und(v, w) = und(w, v)
oder(w, wahr) = wahr
oder(w, falsch) = w
oder(v, w) = oder(w, v)
```

Für viele praktische Anwendungen (vor allem in Programmiersprachen) ist es sinnvoll, mehrere Trägermengen haben zu können. Eine *heterogene Algebra* (engl. *„many-sorted algebra“*) ist ein Tupel $\langle A_1, \ldots, A_n, \Omega \rangle$, wobei $A_1, \ldots, A_n$ Trägermengen sind und Ω ein System von Operationen ist. Jede Trägermenge hat einen Namen, der üblicherweise *Sorte* (engl. *„sort“*) genannt wird. Benennen wir also A_i mit s_i (für $i \in \{1, \ldots, n\}$). Offensichtlich sind heterogene Algebren Erweiterungen der universellen Algebren (engl. *„single-sorted algebra“*). Wir müssen die Definition des oben eingeführten Begriffs Signatur entsprechend erweitern: Die Signatur einer heterogenen Algebra beschreibt neben dem Typ der Algebra auch die Menge der Sorten $S = \{s_1, \ldots, s_n\}$ und den *Index* jeder Operation, die dabei als Funktion aufgefasst wird. Das Beispiel der algebraischen Spezifikation einer Liste von Wahrheitswerten veranschaulicht diese Definition:

Sorten: `WListe`, `WW`

Operationen:

```
leerw : WListe
 cons : WW × WListe → WListe
first : WListe → WW
 rest : WListe → WListe
 wahr : WW
falsch : WW
nicht : WW → WW
  und : WW × WW → WW
 oder : WW × WW → WW
```

Gleichungen:

```
first(cons(w, l)) = w
 rest(cons(w, l)) = l
      rest(leerw) = leerw
      nicht(wahr) = falsch
    nicht(falsch) = wahr
     und(w, wahr) = w
   und(w, falsch) = falsch
        und(v, w) = und(w, v)
    oder(w, wahr) = wahr
   oder(w, falsch) = w
        oder(v, w) = oder(w, v)
```

Wie man an diesem Beispiel sieht, entspricht eine Sorte einem Typ in einer Programmiersprache. Allerdings kann man einen solchen Typ nur beschränkt als ADT bezeichnen, da mehrere Typen in einer einzigen Datenkapsel spezifiziert sind. Andererseits wird der Typ `WW` für die Spezifikation des Typs `WListe` gebraucht. Man kann `WListe` gar nicht unabhängig von `WW` spezifizieren. Wenn man diese Überlegungen weiter treibt erkennt man, dass im Endeffekt jedes ausreichend komplexe Programm in einer einzigen Datenkapsel dargestellt werden muss. Dadurch geht natürlich viel von der Attraktivität von Algebren zur Spezifikation von abstrakten Datentypen verloren.

Es gibt aber eine einfache Lösung für dieses Problem: Wie man am Beispiel sieht, hängt `WListe` von `WW` ab, aber `WW` hängt nicht von `WListe` ab. Daher kann man die Sorte `WW` als einen Parameter einer *generischen* Sorte `Liste` auffassen:

Sorten: `Liste`

Parameter-Sorten: `Element`

Operationen:

```
 leer : Liste
 cons : Element × Liste → Liste
first : Liste → Element
 rest : Liste → Liste
```

Gleichungen:

```
first(cons(e, l)) = e
 rest(cons(e, l)) = l
       rest(leer) = leer
```

Der ursprüngliche Typ `WListe` entspricht dem Typ `Liste[WW]`, wobei `WW` für den Parameter `Element` eingesetzt ist. Mit parametrischen Typen sind heterogene Algebren tatsächlich mächtige Werkzeuge zur Spezifikation von Programmen im Allgemeinen und von abstrakten Datentypen im Besonderen.

2.3.2 Prozessalgebra

Vor allem für die Spezifikation nebenläufiger Prozesse wurden in den letzten Jahren sogenannte Prozessalgebren entwickelt. Eine einfache Prozessalgebra mit den Operationen $\{*, \|, +, \varepsilon\}$ vom Typ $\langle 2, 2, 2, 0 \rangle$ ist zum Beispiel über folgende Gesetze definiert, wobei $*$ die höchste und $+$ die niedrigste Priorität hat:

$$\begin{aligned}
x + y &= y + x \\
(x + y) + z &= x + (y + z) \\
x + x &= x \\
(x + y) * z &= x * z + y * z \\
(x * y) * z &= x * (y * z) \\
x * \varepsilon &= x \\
\varepsilon * x &= x \\
x \parallel y &= y \parallel x \\
(x \parallel y) \parallel z &= x \parallel (y \parallel z) \\
x \parallel \varepsilon &= x
\end{aligned}$$

Neben ε gibt es noch eine Reihe weiterer Konstanten in der Algebra, die nicht ausdrücklich genannt werden. Jede dieser Konstanten entspricht einer atomaren Aktion (= einem Befehl) einer virtuellen Maschine. Prozesse entstehen durch wiederholte Anwendungen der Operatoren $*$, $\|$ und $+$ auf atomare Aktionen.

Ein Ausdruck $x * y$ bedeutet, dass zuerst die atomaren Aktionen von x und danach die atomaren Aktionen von y ausgeführt werden. Im Ausdruck $x \parallel y$ können sich die Ausführungen der atomaren Aktionen von x und y beliebig überlappen, d.h. x und y werden parallel (bzw. quasi-parallel oder nebenläufig) ausgeführt. Ein Ausdruck $x + y$ bedeutet, dass entweder x oder y, aber nicht beide, ausgeführt werden. ε ist eine leere Aktion, die nichts macht. Mit dieser Interpretation lassen sich obige Gesetze intuitiv einfach erklären. Beachten Sie, dass diese Gesetze aus Gründen der Einfachheit gewählt wurden. Es wurden keine Gesetze berücksichtigt, die den Zusammenhang zwischen sequentieller und paralleler Ausführung herstellen. Zum Beispiel gilt für alle atomaren Aktionen a und b häufig $a * b + b * a = a \parallel b$ (Interleavings-Semantik), obwohl dies nicht aus den Gesetzen ableitbar ist. Vollständige Prozessalgebren sind in der Literatur beschrieben.

2.4 Literaturhinweise

Die Beschreibung des Lambda-Kalküls wurde zum Teil aus [14] übernommen. Dieser Artikel gibt einen umfangreichen und verständlichen Überblick über moderne funktionale Programmiersprachen. Kurze Beschreibungen strukturierter Typen und der Semantik von Typen im Lambda-Kalkül finden sich in [6]. Generell gibt es umfangreiche Literatur zum Thema Lambda-Kalkül. Da typisierte Versionen des Lambda-Kalküls nach wie vor Gegenstand wissenschaftlicher Abhandlungen sind, enthalten viele aktuelle Ausgaben von Zeitschriften über (funktionale) Programmiersprachen auch Artikel zu diesem Thema. Eine umfangreiche theoretische Einfhrung mit zahlreichen Literaturangaben vor allem über Typen in funktionalen Sprachen gibt Mitchell in [23].

Eine kompakte, formale und inhaltlich umfangreiche, allerdings für mathematisch weniger Interessierte schwer verständliche Einführung in die logische Programmierung gibt Lloyd in [17]. Darin werden auch Typen und die Eigenschaften typisierter logischer Sprachen beschrieben. Praktische Einführungen in die logische Programmierung, meist jedoch ohne auf Typen einzugehen, bieten zahlreiche Bücher über Prolog. Hier seien die „Klassiker“ [27] und [7] erwähnt. Tiefergehende, aber nicht leicht verständliche Beschreibungen verschiedener Ansätze zur Einführung von Typen in logische Sprachen werden in [24] gegeben.

Viele Einführungen in die Algebra beinhalten auch Beschreibungen universeller und heterogener Algebren. Es gibt auch zahlreiche Abhandlungen über algebraische Spezifikation, in denen Zusammenhänge zwischen Algebren und abstrakten Datentypen erläutert werden. Als Beispiel sei [9] erwähnt. Vergleiche zwischen algebraisch spezifizierten und über erweiterte Lambda-Kalküle definierte Typsysteme werden in [8] angestellt.

Gute Einführungen in Prozessalgebren sind z.B. [5] und [20]. Beide Bücher sind jedoch ohne mathematische Grundkenntnisse nur schwer verständlich. Als ein wichtiger Vertreter der Prozessalgebren gilt heute der π-Kalkül [21, 22].

Kapitel 3

Typen in imperativen Sprachen

In diesem Kapitel werden einige praktische Aspekte von Typen in herkömmlichen imperativen Sprachen untersucht. Als Grundlage wurde die Programmiersprache *Ada* gewählt, da Ada ein—im Vergleich zu anderen herkömmlichen imperativen Sprachen wie Fortran, Cobol, C, Pascal und Modula-2—relativ umfangreiches Typsystem hat, das zu einem guten Teil auf der im vorigen Kapitel eingeführten Theorie beruht und bereits einige für die objektorientierte Programmierung wichtige Eigenschaften vorwegnimmt. In diesem Kapitel werden wir uns auf Ada 83, die 1983 standardisierte Ada-Version, beschränken. In Kapitel 5 werden wir weitere Eigenschaften von Ada 95 behandeln. Auf aktuelle Neuerungen gehen wir nicht ein, da wir uns auf die Entwicklung des Typsystems konzentrieren.

Ein weiterer Grund für die Wahl von Ada ist, dass dessen Typsystem, mehr als Typsysteme in vielen anderen imperativen Sprachen, umfangreiche dynamische Komponenten enthält, die Typüberprüfungen zur Laufzeit notwendig machen. Damit soll gezeigt werden, dass statische Überprüfbarkeit nicht in allen praktisch verwendeten imperativen Sprachen gegeben sein muss. Dennoch ist Ada, wie alle imperativen Sprachen, eher konservativ; das heißt, alle Programme, die möglicherweise Typfehler im statisch überprüften Teil des Typsystems enthalten, lassen sich nicht übersetzen. Man teilt das Typsystem also in zwei Teile. Typkompatibilität im einen Teil wird vom Compiler überprüft, die im anderen Teil während der Ausführung. Die Zuordnung von Überprüfungen zu diesen beiden Teilen erfolgt aus rein pragmatischen Gesichtspunkten.

3.1 Datentypen in Ada

Das Typsystem von Ada wurde stark von Algebren beeinflusst, sodass abstrakte Datentypen zu den wichtigsten Strukturierungsmitteln in der Sprache zählen. Sogar die einfachen vordefinierten Typen werden durch eine implizit vorgegebene Algebra spezifiziert. Die zusammengesetzten Datentypen entsprechen im Großen und Ganzen den im Abschnitt über den Lambda-Kalkül beschriebenen, obwohl Ada die funktionale Programmierung nur in einer eingeschränkten Form unterstützt.

Das folgende Schema gibt einen Überblick über die in diesem Abschnitt angesprochenen Datentypen. Daneben werden noch Unterbereichstypen und abgeleitete Typen jeder Typkategorie in diesem Schema behandelt. (Typen von „tasks“ zählen in Ada zwar auch zu den Typen, werden aber erst in Abschnitt 3.2 erläutert.)

- Datentypen
  - elementar
    - skalar
      - diskret
        - Aufzählungen
          - Zeichen
          - Wahrheitswerte
          - andere
        - ganze Zahlen
          - vorzeichenbehaftet
          - modular
      - reelle Zahlen
        - Gleitkommazahlen
        - Festkommazahlen
          - gewöhnlich
          - dezimal
    - Zeiger
  - zusammengesetzt
    - Verbunde
      - einfache Verbunde
      - diskriminante Verbunde
        - mit Varianten
        - ohne Varianten
    - Felder
      - mit spezifizierten Grenzen
      - ohne spezifizierte Grenzen

3.1.1 Skalare Typen

Zu den skalaren Typen zählen Aufzählungstypen und Zahlen. Jeder dieser Typen beschreibt eine Menge von

Werten ohne sichtbare innere Struktur und die auf diesen Werten ausführbaren Operationen.

Aufzählungstypen. Der Wertebereich eines Aufzählungstyps wird, wie der Name schon sagt, extensional durch Aufzählung der Werte festgelegt. Die Werte eines Typs sind vollständig geordnet; je zwei Werte sind anhand ihrer relativen Position in der Aufzählung vergleichbar. Die Vergleichsoperatoren `<`, `<=`, `=`, `>=`, `>` und `/=`, alle vom Typ $t \times t \to$ `BOOLEAN`, sind auf jedem Aufzählungstyp t definiert. Weiters sind einige sogenannte *Attribute* von Aufzählungstypen vordefiniert; das sind Familien von Funktionen, die einen Aufzählungstyp als Index haben. Für jeden Aufzählungstyp t sind diese null- oder einstelligen Attribute vorgegeben:

Funktion	Typ	Beschreibung
t`'FIRST`	t	kleinster Wert in t
t`'LAST`	t	größter Wert in t
t`'SUCC`	$t \to t$	nachfolgender Wert
t`'PRED`	$t \to t$	vorausgehender Wert
t`'POS`	$t \to$ `CARDINAL`	Position des Wertes
t`'VAL`	`CARDINAL` $\to t$	Wert an Position

Allgemeine Aufzählungstypen sind frei definierbare Aufzählungstypen, die keine weiteren Zugriffsfunktionen haben. Ein Beispiel zeigt, wie sie definiert werden:[1]

```
type TAG is (MO, DI, MI, DO, FR, SA, SO);
type ARBEITSTAG is (MO, DI, MI, DO, FR);
```

An diesem Beispiel ist zu beachten, dass `TAG` und `ARBEITSTAG` zwei unabhängige Typen sind, die keine gemeinsamen Werte enthalten. `MO` in `TAG` ist also ein anderer Wert als `MO` in `ARBEITSTAG`. Wenn aus dem Kontext, in dem `MO` verwendet wird, nicht klar hervorgeht, welches `MO` gemeint ist, dann muss der gewünschte Typ mittels *Typqualifikation* in der Form `TAG'(MO)` bzw. `ARBEITSTAG'(MO)` angegeben werden. Von Aufzählungstypen können Unterbereiche gebildet werden. Zum Beispiel beschreibt der Typausdruck `(TAG'(MO)..TAG'(FR))` einen Unterbereich von `TAG`.

Wahrheitswerte. Der Typ `BOOLEAN` ist ein vordefinierter Aufzählungstyp, der nur die Werte `TRUE` und `FALSE` enthält. Neben den für alle Aufzählungstypen definierten Funktionen gibt es noch die Operationen `not`, `and`, `or` und `xor` („exclusive or“) mit den üblichen Definitionen. Die Kurzschlussoperatoren `and then` und `or else` werten, im Unterschied zu `and` und `or`, den rechten Operanden nur aus, wenn die Auswertung des linken Operanden `TRUE` (für `and then`) bzw. `FALSE` (für `or else`) ergibt. An Hand der Wahrheitswerte wird besonders deutlich, dass in Ada, wie in den meisten imperativen Sprachen, das Typsystem untrennbar in die Sprache integriert ist. Die Eigenschaften von Wahrheitswerten werden z.B. in Kontrollstrukturen für Optimierungen genutzt, sodass jede Änderung der Definition von Wahrheitswerten zwangsläufig große Änderungen der Sprache nach sich ziehen würde.

Zeichen. Auch der vordefinierte Datentyp `CHARACTER` ist ein Aufzählungstyp, der als Wertebereich die Zeichen des ASCII-Codes hat. Syntaktisch werden Zeichen in einfache Hochkomma eingeschlossen; beispielsweise sind `'a'` und `'1'` Zeichen. Es lassen sich beliebige Aufzählungstypen definieren, deren Werte Zeichen sind. Ein Beispiel ist

```
type GeradeZiff is ('0','2','4','6','8');
```

Zahlen. Ada unterstützt drei Klassen von Zahlen: ganze Zahlen, Gleitkomma- und Festkommazahlen.

Ganze Zahlen. Ähnlich wie in anderen Sprachen sind auf ganzen Zahlen dieselben Vergleichsoperatoren wie für Aufzählungstypen, die unären Operatoren `+` und `-`, die binären Operatoren `+`, `-`, `*`, `/` (ganzzahlige Division), `mod` (Divisionsrest, Vorzeichen des rechten Operanden), `rem` (Divisionsrest, Vorzeichen des linken Operanden) und `**` (Exponentiation) sowie die einstellige Funktion `ABS` (Absolutwert) definiert. Operanden und Ergebnisse (außer bei Vergleichsoperatoren) sind jeweils vom selben Typ; es finden keine impliziten Typumwandlungen statt. Dies ist besonders deshalb interessant, weil es mehrere ganzzahlige Datentypen gibt. Zu den vordefinierten ganzzahligen Datentypen zählen `SHORT_INTEGER`, `INTEGER` und `LONG_INTEGER`. Mehrere ganzzahlige Datentypen sind in imperativen Sprachen häufig anzutreffen, und fast so häufig sind die absoluten Wertebereiche, wie auch in Ada, implementierungsabhängig. Das kann bei Programmportierungen zu einer großen Hürde werden. Eine Teillösung bietet die Möglichkeit in Ada, Bereiche für ganze Zahlen anzugeben. Zum Beispiel wird durch

```
type SEITENZAHL is range 1..10000;
```

ein ganzzahliger Typ `SEITENZAHL` im Bereich von 1 bis 10000 definiert. Bei dieser Form der Typdefinition gibt es die Einschränkungen, dass die untere und obere Grenze statisch (durch den Compiler) berechenbar sein und der definierte Bereich in dem von `LONG_INTEGER` liegen muss. Auf jedem ganzzahligen Typ t sind die Funktionen t`'FIRST` und t`'LAST` vom Typ $t \to t$, die die kleinste bzw. größte Zahl im Wertebereich von t zurückliefern und vom Compiler ausgewertet werden können, definiert. Zum Beispiel kann der weiter oben verwendete Typ `CARDINAL` definiert werden durch

```
type CARDINAL is range 0..INTEGER'LAST;
```

[1] Groß- und Kleinschreibung wird in Ada nicht unterschieden. Wir folgen hier der Konvention, dass nur Schlüsselwörter mit einem Kleinbuchstaben beginnen.

Alle obigen Beispiele für ganzzahlige Typen fallen in die Klasse der *vorzeichenbehafteten ganzzahligen Typen.* Wird der Wertebereich eines solchen Typs überschritten, so wird in Ada eine Ausnahmebehandlung („exception“) ausgelöst. Im Gegensatz dazu gibt es bei *modularen ganzzahligen Typen* keine Wertebereichsüberschreitungen, sondern der Wert wird an den vorgegebenen Wertebereich angepasst. Zum Beispiel umfasst der modulare Wertebereich des Typs `ZWEI_ZIFFERN` definiert durch

```
type ZWEI_ZIFFERN is mod 100;
```

die ganzen Zahlen von 0 bis 99 modulo 100. Die Zahlen -90, 10, 110, ... werden als äquivalent angesehen. In Sprachen wie C sind alle ganzzahligen Typen modular; d.h. bei Überlauf werden die Zahlen automatisch auf einen vorgegebenen modularen Wertebereich verkürzt. Anders als in Ada ist der Wertebereich stets eine Zweierpotenz und kann auch negative Zahlen umfassen.

Gleitkommazahlen sind auf ähnliche Weise definiert wie ganze Zahlen. Außer `rem` und `mod` sind alle Operationen auf ganzen Zahlen auch auf Gleitkommazahlen anwendbar, wobei Argument- und Ergebnistypen wieder übereinstimmen müssen. Die vordefinierten Typen sind `SHORT_FLOAT`, `FLOAT` und `LONG_FLOAT`. Es können eigene Gleitkommatypen definiert werden, wobei neben dem Wertebereich auch die Genauigkeit angegeben werden kann:

```
type MY_FLOAT is digits 8 range -1.0..1.0E30;
```

Wiederum sind sowohl der Wertebereich als auch die Genauigkeit durch die Beschränkungen eines vordefinierten Typs `LONG_FLOAT` begrenzt. Auf jeder Gleitkommazahl t sind einige Attribute vordefiniert, die den Typ genauer spezifizieren:

Funktion	Typ	Beschreibung
t'DIGITS	CARDINAL	berechenb. Dezimalstellen
t'MANTISSA	CARDINAL	Länge der Binär-Mantisse
t'EMAX	CARDINAL	maximaler Exponent
t'SMALL	t	kleinste positive Zahl
t'LARGE	t	größte positive Zahl
t'EPSILON	t	max. Fehler/Rechenschritt

Festkommazahlen. Für Festkommazahlen gibt es keine vordefinierte Typen. Alle solchen Typen müssen explizit definiert werden. Die Definition eines gewöhnlichen Festkommatyps sieht folgendermaßen aus:

```
type SPANNUNG is delta 0.1 range 0.0..10.0;
```

Die `delta`-Klausel gibt die Mindestgenauigkeit an. Obwohl Festkommazahlen oft für dieselben Aufgaben wie Gleitkommazahlen herangezogen werden, sind ihre Eigenschaften und Implementierungen doch—aus der Sicht der Ada-Designer—ausreichend verschieden, um darauf leicht voneinander abweichende Operationsmengen zu definieren. So kann bei Festkommazahlen einer der Operanden der Multiplikation und der zweite Operand der Division eine ganze Zahl sein, da es für diese Fälle einfache, effiziente Implementierungen gibt. Diese Attribute sind auf Festkommazahlen vordefiniert:

Funktion	Typ	Beschreibung
t'DELTA	CARDINAL	Mindestgenauigkeit
t'ACTUAL_DELTA	CARDINAL	tatsächl. Genauigkeit
t'BITS	CARDINAL	Speicherbedarf
t'LARGE	t	größte Zahl

Für Anwendungen im Finanzbereich benötigt man häufig dezimale Festkommazahlen. Deren Typen werden folgendermaßen definiert:

```
type PREIS is delta 0.01 digits 5;
```

Der Wertebereich dieses Typs reicht von -999.99 bis 999.99 (fünf Dezimalstellen). Da im Finanzbereich genau gerechnet und gerundet werden muss, entspricht die Mindestgenauigkeit von dezimalen Festkommazahlen der tatsächlichen Genauigkeit.

3.1.2 Zusammengesetzte Typen

Zusammengesetzte (= strukturierte) Typen werden aus skalaren und anderen zusammengesetzten Datentypen (und Typen von „tasks“, siehe Abschnitt 3.2.2) aufgebaut. Die wichtigsten zusammengesetzten Typen sind ein- und mehrdimensionale Felder mit spezifizierten oder unspezifizierten Grenzen und Verbunde mit und ohne Varianten.

Felder. Ein Feld wird, wie üblich, durch den Typ seiner Komponenten und den (oder die) Indexbereich(e) definiert. Als Indexbereich kann jeder *diskrete* Typ dienen; das ist ein Aufzählungstyp oder ein ganzzahliger Typ, oder ein Unterbereich davon. Hier sind einige Beispiele:

```
type VEKTOR is array (1..5) of INTEGER;
type MATRIX is array
     (1..K*J,Func(N)..Func(N*M)) of FLOAT;
type STUNDENTAFEL is array
     (TAG range MO..SA, 1..8) of TEXT;
```

Die generelle Form einer Felddefinition ist

```
type t is array (t_1, ..., t_n) of t';
```

wobei t der definierte Typ des n-dimensionalen Feldes ist, $t_1, \ldots, t_n$ die n Indexbereiche sind und t' der Komponentyp ist.

Eine Besonderheit von Ada ist, dass Feldgrenzen nicht statisch festgelegt werden müssen. Daher können

auch Bereiche wie `Func(N)..Func(N*M)` angegeben werden, deren Grenzen erst zur Laufzeit durch Anwendung der Funktion `Func` berechenbar sind. In diesen Fällen sind natürlich auch die von den Grenzen abhängigen Attribute erst während der Ausführung berechenbar. Diese Attribute sind (für $i \in \{1, \ldots, n\}$) vordefiniert:

Funktion	Typ	Beschreibung
t'FIRST(i)	t_i	untere Indexgrenze
t'LAST(i)	t_i	obere Indexgrenze
t'LENGTH(i)	CARDINAL	Anzahl der Werte

Eine weitere Besonderheit sind Feldtypen mit unspezifizierten Grenzen. Beispielsweise ist diese Typdefinition erlaubt:

```
type MATR is array
     (INTEGER range <>, INTEGER range <>)
     of FLOAT;
```

Allerdings können von diesem abstrakten Typ direkt keine Instanzen erzeugt werden. Bei der Deklaration einer Variable dieses Typs müssen die fehlenden Bereichsangaben nachgetragen werden. Zum Beispiel ist `MY_MATRIX` eine Variable eines Typs mit dynamisch festgelegten Grenzen:

```
MY_MATRIX: MATR(1..20,FUNC(N)..FUNC(N*M));
```

Der vordefinierte Typ `STRING` ist ein eindimensionales Feld von Zeichen mit unspezifizierten Grenzen. Die Länge eines Strings muss bei Variablendeklarationen angegeben werden. Die Längenangabe kann aber entfallen, wenn die Länge aus dem Kontext herleitbar ist. Auch hier spielen pragmatische Entscheidungen eine größere Rolle als ein klares Konzept.

Verbundtypen. Diese werden ähnlich wie in vielen anderen Sprachen definiert:

```
type DATUM is
     record
       TAG:   INTEGER range 1..31;
       MONAT: MONATSNAME;
       JAHR:  INTEGER range 1800..2200;
     end record;
```

Verbundtypen können *Diskriminanten* enthalten; das sind spezielle Komponenten, die dazu beitragen, die Struktur des Verbundtyps zu bestimmen. Im nächsten Beispiel werden sie dazu verwendet, die Bereichsgrenzen von Feldern des weiter oben definierten Typs `MATR` zu spezifizieren.

```
type MATRIZEN (L: INTEGER; R: INTEGER) is
     record
       M1: MATR(1..2*L, 1..3*R);
       M2: MATR(1..3*R, 1..2*L);
     end record;
```

Ein Spezialfall sind Verbunde mit Varianten. Varianten werden mittels `case`-Anweisung auf einer (oder mehreren) Diskriminanten festgelegt. Der variante Verbundtyp `PERIPHERIE` im folgenden Beispiel enthält die Komponente `ZEILE` genau dann, wenn die Diskriminante `SORTE` den Wert `DRUCKER` enthält. In den beiden anderen Fällen sind statt dessen die Komponenten `ZYLINDER` und `SPUR` vorhanden.

```
type GERAET is (DRUCKER,PLATTE,TROMMEL);

type PERIPHERIE (SORTE: GERAET) is
     record
       STATUS: ZUSTAND;
       case SORTE is
            when DRUCKER =>
                 ZEILE:    INTEGER;
            when others =>
                 ZYLINDER: INTEGER;
                 SPUR:     INTEGER;
       end case;
     end record;
```

Ada bietet zahlreiche Möglichkeiten um die Speicherauslegung für Instanzen von Typen genau zu spezifizieren. Beispiele:

```
type Device_Register is range 0..2**8-1;
D: Device_Register;
for D'Size use 8;
for D'Address use To_Address(16#FF00#);

type Colors is (Red,Green,Blue);
for Colors use (Red=>10,Green=>20,Blue=>30);

type Status_Word is record
    Mask      : array (0..7) of Boolean;
    Protection: Integer range 0..3;
    Something : Address;
  end record;
for Status_Word use record
    Mask      : at 0*Word range 0..7;
    Protection: at 0*Word range 10..11;
    Something : at 1*Word range 8..31;
  end record;
for Status_Word'Alignment use 8;
```

3.1.3 Unterbereichstypen, abgeleitete Typen und Zeiger

Von jedem Typ (oder Unterbereichstyp) können verschiedene *Unterbereichstypen* erzeugt werden. Ein Unterbereichstyp enthält zusätzliche Einschränkungen gegenüber dem Typ, dessen Unterbereichstyp er ist. Es können Wertebereiche (`range MO..FR`, `range 1..9`, etc.) und Genauigkeiten (`digits 6`, `delta 0.1`, etc.) stärker eingeschränkt, bzw. bei Feldern Bereichsangaben und bei Verbunden Diskriminanten festgelegt wer-

den. Hier sind einige Beispiele für Typen, die Unterbereichstypen von weiter oben definierten Typen sind:

```
subtype WERKTAG is TAG'(MO)..TAG'(SA);
        (entspricht TAG range MO..SA)
subtype SMALL is INTEGER range -100..100;
subtype BETRIEBS_SPANNUNG is
        SPANNUNG delta 0.5 range 0.0..5.0;
subtype PP is PREIS range 0.01..999.99;
subtype SMALL_VEKTOR is VEKTOR(2..4);
subtype MATRIX1 is MATR(1..100,1..100);
subtype PER_DR is PERIPHERIE(DRUCKER);
```

Unterbereichstypen stellen keine neuen Typen im eigentlichen Sinn dar. Sie legen nur Einschränkungen fest, die zur Laufzeit überprüft werden. Falls eine Einschränkung nicht erfüllt ist, wird eine Ausnahmebehandlung (engl. *„exception handling“*) eingeleitet. Instanzen verschiedener Unterbereichstypen eines gemeinsamen Typs dürfen in Ausdrücken gemischt vorkommen. (Wie wir später sehen werden, sind Unterbereichstypen wie Untertypen verwendbar; jedoch erfolgt die Überprüfung der Einschränkungen in Ada zur Laufzeit, während bei Untertypen meisst statische Typüberprüfungen möglich sind.)

Alle Typen haben einen Namen. Das heißt, mit Ausnahme von Feldern und Zeigern muss es für jeden verwendeten und nicht vordefinierten Typ eine explizite Typdefinition geben. (Für Feld- und Zeigertypen wird implizit ein nach außen nicht sichtbarer Name angenommen falls kein Name explizit definiert ist.) Unterbereichstypen können dagegen direkt in Variablendeklarationen vorkommen. Siehe zum Beispiel obige Deklaration von `MY_MATRIX`.

Neben den Unterbereichstypen gibt es die sogenannten *abgeleiteten* Typen. Im Gegensatz zu Unterbereichstypen sind abgeleitete Typen tatsächlich eigenständige Typen, die in Ausdrücken nicht gemischt vorkommen dürfen. Der Compiler prüft auf Namensgleichheit. Auf jedem abgeleiteten Typ sind dieselben Operationen anwendbar, die auf dem Typ, aus dem er abgeleitet wurde, anwendbar sind (= Überladen). Folgendes Beispiel zeigt die Syntax abgeleiteter Typdeklarationen:

```
type APFEL is new INTEGER range 0..100;
type BIRNE is new INTEGER range 0..100;
```

Obwohl `APFEL` und `BIRNE` dieselbe Struktur haben und aus demselben Typ erzeugt wurden, stellen sie zwei völlig verschiedene Typen dar. Instanzen dieser Typen werden jedoch durch explizite Typumwandlung vergleichbar. Dies ist möglich, da die interne Repräsentation eines abgeleiteten Typs im übersetzten Programm der des Typs, von dem abgeleitet wurde, entspricht. Allerdings ist die Typumwandlung nur zwischen je zwei Typen, die gemeinsam in einer Definition eines abgeleiteten Typs vorkommen, erlaubt. Daher kann aus dem Inhalt einer Variable `A` vom Typ `Apfel` nur durch diese zweifache Typumwandlung eine Instanz des Typs `BIRNE` erzeugt werden: `BIRNE(INTEGER(A))`.

Zum Abschluss seien *Zeigertypen* erwähnt. Auf jeden beliebigen Datentyp kann auch ein Zeigertyp definiert werden. Zum Beispiel ist `APFEL_P` der Typ eines Zeigers auf eine Instanz des Typs `Apfel`:

```
type APFEL_P is access APFEL;
MY_APFEL: APFEL_P := new APFEL;
```

Ein Zeiger dieses Typs auf eine neue Instanz von `APFEL` im Heap wird mittels `new APFEL` erzeugt. Der Typ, der nach `new` steht, muss so genau spezifiziert sein, dass eine neue Instanz davon erzeugt werden kann; die Grenzen von Arrays und Diskriminanten von Verbunden müssen festgelegt werden. Bei der Definition eines Zeigertyps brauchen diese Einschränkungen nicht bekannt zu sein.

3.2 Prozeduren und Prozesse

Nicht nur Daten im eigentlichen Sinn haben einen Typ. Auch Funktionen, Prozeduren und Prozessen kann ein Typ zugeordnet werden. In manchen Sprachen haben auch Module einen Typ—z.B. in ML, in Ada aber nur ansatzweise.

3.2.1 Funktionen und Prozeduren

Hier sind einige Beispiele für Funktions- und Prozedur-Deklarationen in Ada:

```
procedure PUSH (EL: in ELEMENT_T;
                ST: in out STACK_T);
procedure POP (EL: out ELEMENT_T;
               ST: in out STACK_T);
function MAX (X,Y: in FLOAT) return FLOAT;
function "*" (X: in MATR(A,B);
              Y: in MATR(B,C))
             return MATR(A,C);
```

Die Wörter `in`, `out` und `in out` legen den *Modus* (= Flussrichtung der Daten) der Parameter fest. In Ada können vordefinierte Operatoren, wie z.B. `*`, überladen werden.

Der Kopf einer Routine kann als Typdefinition aufgefasst werden, die den Namen der Routine, die Anzahl, Typen und Modi der Parameter und, bei Funktionen, den Ergebnistyp festlegt. In Ada 83 wird jedem Funktions- oder Prozedurnamen genau eine Implementierung zugeordnet, die man als die einzige Instanz des

entsprechenden Typs der Routine ansehen kann. Wegen dieser strikten eins-zu-eins-Beziehung zwischen einer Implementierung und dessen Typ reicht ein gemeinsamer Name für beide. Es ist wenig sinnvoll, Funktions- oder Prozedurtypen wie gewöhnliche Datentypen zu behandeln, da die Angabe eines solchen Typs die dazugehörige Implementierung eindeutig festlegt.

Im Gegensatz zu Ada 83 behandeln einige imperative Sprachen (z.B. C, Modula 2 und auch Ada 95) Typen von Routinen wie (gewöhnliche) Datentypen. Diese Typen erlauben uns, Routinen in Feldern und Verbunden zu speichern und als Argumente an Routinen zu übergeben. Dadurch wird die eins-zu-eins-Beziehung zwischen Typen und Implementierungen aufgehoben, sodass es für jeden Typ mehrere Implementierungen geben kann. Es ist notwendig, Implementierungen einen Namen zu geben, der sich vom Typnamen unterscheidet. In C und Ada 95 wird dies durch Zeiger auf Implementierungen bewerkstelligt. Zum Beispiel:

```
type OP_P is access function (X,Y: in FLOAT)
                              return FLOAT;
MY_OPERATION: OP_P := MAX'Access;
```

Welcher der beiden oben beschriebenen Ansätze ist besser? Diese Frage lässt sich nicht allgemein beantworten. (In Sprachen wie C und Ada 95 werden daher beide Ansätze verwendet—eine eins-zu-eins-Beziehung zwischen Typ und Implementierung für gewöhnliche Routinen und Trennung zwischen Typ und Implementierung bei Verwendung von Zeigern auf Routinen.) Einerseits bietet die Trennung von Typ und Implementierung größere Flexibilität, andererseits stellt sie auch eine zusätzliche Fehlerquelle dar. In fast allen Sprachen, die mehrere Implementierungen eines Typs von Routinen erlauben, wird Typäquivalenz als Strukturgleichheit aufgefasst. Daher haben alle Implementierungen mit gleich strukturierten Schnittstellen denselben Typ, auch wenn die Implementierungen ansonsten nichts miteinander zu tun haben.

Eine eins-zu-eins-Beziehung zwischen Typ und Implementierung bedeutet dagegen immer Typäquivalenz aufgrund von Namensgleichheit. Erst in letzter Zeit wurden Sprachen entwickelt, die Äquivalenz von Funktionstypen aufgrund von Namensgleichheit unterstützen und trotzdem mehrere Implementierungen eines Typs zulassen. Dabei müssen getrennte Namen für Routinentypen und Implementierungen definiert werden, und jede Implementierung muss sich explizit auf einen Routinentyp beziehen, wodurch sich manchmal der Schreibaufwand stark erhöht (siehe Kapitel 5).

Ein bedeutendes neues Typmodell für die objektorientierte Programmierung zeigt, dass Typäquivalenz aufgrund von Strukturgleichheit neben übereinstimmenden Schnittstellen auch explizit spezifizierte übereinstimmende semantische Eigenschaften einschließen kann (siehe ebenfalls Kapitel 5). Dieses Typmodell ist daher bezüglich der Äquivalenz von Prozedurtypen noch aussagekräftiger als ein auf Typäquivalenz aufgrund von Namensgleichheit beruhendes System, da Namensgleichheit zwar das Zusammenfassen beliebiger Prozeduren zu einem Typ erlaubt, aber diese Zuordnung in keiner Weise begründet.

3.2.2 Inkarnationen und Prozesse

Eine Besonderheit von Ada sind Prozesse (*„tasks“* in Ada-Terminologie) für die nebenläufige Programmierung (*„concurrency“*). Jeder Prozess hat einen Typ, der wie jeder Datentyp behandelt wird und von dem mehrere Instanzen erzeugt werden können, wie folgendes Beispiel zeigt:

```
task type DRUCKER_TREIBER is
  entry DRUCKE_ZEILE(ZL: in STRING(80));
  entry SEITENVORSCHUB;
  entry DRUCKER_STATUS(F: out STATUS);
end;

DRUCKER_POOL: array(1..MAX_DRUCKER)
              of DRUCKER_TREIBER;
DRUCKER_POOL(1).DRUCKE_ZEILE("Hallo!");
```

Die Deklaration eines solchen Typs spezifiziert Einstiegspunkte (*„entries“*) in einen Prozess dieses Typs. Ein Prozess ist die Ausführung eines Programmstücks, in der Regel in einer Schleife. Wenn ein *„accept statement“* für einen Einstiegspunkt ausgeführt wird, wartet der Prozess so lange, bis ein anderer Prozess einen Aufruf dieses Einstiegspunktes macht und dadurch das „accept statement“ und der Aufruf in Interaktion treten. Der aufrufende Prozess wartet so lange, bis das „accept statement“ vollständig abgearbeitet ist (*„Rendezvous“*). Wie bei Prozeduraufrufen können Argumente in beide Richtungen übergeben werden. Für jeden Prozess p und jeden seiner Einstiegspunkte e gibt es einige Attribute:

Funktion	Typ	Beschreibung
p'TERMINATED	BOOLEAN	p beendet?
p'PRIORITY	CARDINAL	Priorität von p
p'STORAGE_SIZE	CARDINAL	Speicherbedarf für p
e'COUNT	CARDINAL	Anzahl der Aufrufe

Wie bei Funktionen und Prozeduren besteht eine eins-zu-eins-Beziehung zwischen dem Typ eines „task“ und dem von allen Prozessen dieses Typs ausgeführten Programmstück, der Implementierung des Prozesses. Wie bei Funktionen und Prozeduren könnte man auch bei Prozessen diese fixe eins-zu-eins-Beziehung auflösen. Dadurch würde eine dreistufige Hierarchie

entstehen: Ein Prozesstyp hat mehrere Implementierungen, und jede Implementierung kann mehrere Prozesse aufspannen.

Auch bei Routinen kann man sich eine solche Dreiteilung vorstellen, wenn man Implementierungen von Routinen als Typen und deren Inkarnationen (= durch Routinenaufrufe erzeugte „stack frames“) als Instanzen der Implementierungen betrachtet. Die Instanzen von Typen sind also wiederum Typen, die Instanzen einer niedrigeren Ebene beschreiben:

Typ einerer Routine oder eines „task“
↑ (ist vom Typ)
Implementierung
↑ (ist vom Typ)
Routinen-Inkarnation oder „task“

Tatsächlich werden in einigen Sprachen, z.B. einigen Dialekten von Lisp und Smalltalk, Inkarnationen von Routinen wie gewöhnliche Daten behandelt. In einigen Sprachen wird auch versucht, Prozeduren, Funktionen und Prozesse—und manchmal auch Module—zu einem einzigen Konstrukt zu vereinigen, sodass die oben dargestellte Hierarchie einheitlicher wird. Natürlich ist es bei Verwendung polymorpher Typsysteme auch möglich, dass eine Implementierung mehrere Typen hat und eine Inkarnation oder ein „task“ mehreren Implementierungen entspricht, obwohl letzteres recht unpraktisch ist.

Eine mehr als zweistufige Hierarchie ist als Typ-Instanz-Beziehung aber oft hinderlich. Eine Lösung bietet hier Vererbung, wobei eine konkrete Implementierung von einem abstrakten Routinen- oder Prozesstyp erbt (siehe Kapitel 5).

3.3 Generische Pakete

Ada hat die Signaturen des theoretischen Konzepts der heterogenen Algebren mit Parametersorten (siehe Abschnitt 2.3) fast unverändert zur Definition der Schnittstellen von abstrakten Datentypen übernommen.

Das Beispiel in Abbildung 3.1 veranschaulicht die Definition eines *generischen Paketes* (= Ada-Terminologie für Module mit Parametern). Das generische Paket `STACK` hat zwei Parameter. Der erste gibt den Typ der Stackelemente an, der zweite die maximale Anzahl von Elementen in einem Stack. Der Typ eines Stacks, `STACK_T`, ist, wie die im öffentlichen Teil des Paketes deklarierten Prozeduren und Funktionen, allgemein verwendbar. Da `STACK_T` hier als `limited private` deklariert ist, ist seine Struktur nach außen unsichtbar, und seine Instanzen können nur von den von `STACK` zur Verfügung gestellten Prozeduren und Funktionen gesehen und manipuliert werden. Im privaten Teil wird die

```
generic
  type ELEMENT_T is private;
  MAX_ELEM: CARDINAL;
package STACK is
  type STACK_T is limited private;
  procedure PUSH (EL: in ELEMENT_T;
                  ST: in out STACK_T);
  procedure POP (EL: out ELEMENT_T;
                 ST: in out STACK_T);
  function IS_EMPTY (ST: in STACK_T)
                     return BOOLEAN;
  function IS_FULL (ST: in STACK_T)
                     return BOOLEAN;
private
  type STACK_T is
       record
         EINTRAEGE: array (1..MAX_ELEM)
                    of ELEMENT_T;
         INDEX: INTEGER range 0..MAX_ELEM;
       end record;
end STACK;

package body STACK is
  ...
end STACK;

package I_STACK is new STACK(INTEGER,100);

MY_STACK: I_STACK.STACK_T;
```

Abbildung 3.1: Beispiel für generisches Paket

Typdefinition vervollständigt, sodass sie der Compiler sehen kann.

Die Implementierung der Prozeduren und Funktionen eines Paketes erfolgt im „package body“, der eine getrennte Übersetzungseinheit sein kann. In Ada gibt es auch bei Modulen eine eins-zu-eins-Beziehung zwischen dem Modultyp (= Signatur des ADT = Deklarationsteil eines Paketes) und seiner Implementierung (= Gesetze einer freien Algebra = „package body“). Auch bei Modulen könnte man, wie in einigen objektorientierten Sprachen, mehrere Implementierungen eines Modultyps zulassen.

Ein generisches Paket ist nicht direkt verwendbar. Statt dessen muss daraus ein neues (nicht generisches) Paket, im obigen Beispiel `I_STACK`, erzeugt werden, in dem alle Parameter durch konkrete Typen bzw. Werte ersetzt sind. Dieses Paket kann man verwenden; z.B. kann man eine Variable `MY_STACK` vom von `I_STACK` exportierten Typ `STACK_T` deklarieren. Die Möglichkeit, beliebig viele neue Pakete aus `STACK` zu erzeugen, kann man auf vielfältige Weise nutzen. Zum Beispiel kommt die einfachere Variante in Abbildung 3.2 ohne `STACK_T` aus, wodurch jede Prozedur und Funktion ein Argument weniger hat. Die benötigten Datenstrukturen werden nicht mehr über Argumente übergeben,

```
generic
  type ELEMENT_T is private;
  MAX_ELEM: CARDINAL;
package STACK1 is
  procedure PUSH (EL: in ELEMENT_T);
  procedure POP (EL: out ELEMENT_T);
  function IS_EMPTY return BOOLEAN;
  function IS_FULL return BOOLEAN;
end STACK1;

package body STACK1 is
  EINTRAEGE: array (1..MAX_ELEM)
             of ELEMENT_T;
  INDEX: INTEGER range 0..MAX_ELEM;
  ...
end STACK1;

package I_STACK1 is new STACK1(INTEGER,100);

MY_STACK1: I_STACK1;
```

Abbildung 3.2: Beispiel für vereinfachten ADT

sondern stehen in globalen Variablen. Durch Erzeugen neuer Pakete aus `STACK1` werden auch neue Versionen dieser globalen Variablen erzeugt, sodass sie sich nicht gegenseitig beeinflussen können.

In Ada gibt es in diesem Zusammenhang ein Problem: Da für jede Variable wie `MY_STACK1` auch ein neues Paket erzeugt werden muss, ist es im Allgemeinen nicht möglich, ein Feld von solchen Stacks zu erzeugen. Im Prinzip wäre es nur nötig, implizite Definitionen neuer Pakete aus `STACK1` bei der Initialisierung von Zeigern zuzulassen, um dieses Problem zu vermeiden. Zum Beispiel könnte man dann ein Feld von Zeigern auf Stacks etwa so deklarieren:

```
type ST_P is access STACK1;
F: array (1..10) of ST_P :=
    (1..10 => new STACK1(INTEGER,100));
```

In jenen objektorientierten Sprachen, in denen Module und Klassen zusammenfallen, werden auf ähnliche Weise neue Objekte erzeugt. (In Ada 95 wurde jedoch ein anderer Weg gewählt.)

Die Entwickler von Ada 83 geben als Begründung für die Nichtunterstützung von Routinen als Parameter und Zeigern auf Module an, dass durch diese Konzepte die statische (= textuelle) Programmsicherheit gefährdet wäre, da der Programmierer sich nicht mehr auf die strikte eins-zu-eins-Beziehung zwischen Namen und Routinen bzw. Modulen verlassen könnte. Mittlerweile wurde diese Entscheidung teilweise revidiert: Ada 95 unterstützt Routinen als Parameter und bietet neue Sprachkonstrukte mit ähnlicher Funktionalität wie Zeiger auf Module (siehe Abschnitt 5.1.1). Wegen der geforderten statischen Programmsicherheit gibt es jedoch auch weiterhin keine Zeiger auf Module.

Während `STACK` einer Algebra sehr genau entspricht, kann `STACK1` nicht einfach als Algebra aufgefasst werden. Der Grund liegt in den globalen Variablen, die in Algebren unbekannt sind. Andererseits sind solche globale Variablen zusammen mit der Möglichkeit, beliebig viele Versionen davon zu erzeugen, ein mächtiges und häufig verwendetes Werkzeug in modernen Programmiersprachen. Daher entwickelte man neben Algebren auch andere theoretische Modelle, die den tatsächlichen Gegebenheiten in vielen imperativen Sprachen besser entsprechen.

3.4 Literaturhinweise

Es gibt zahlreiche Bücher und Zeitschriftenartikel über Ada. In der TU-Hauptbibliothek sind über 80 relevante Bücher, Zeitschriften und Konferenzberichte vorhanden. Der offizielle Sprachstandard von Ada 83 ist in [2] beschrieben, der von Ada 95 in [3]. Informationen über aktuelle Standards und Entwicklungen zu Ada sind unter `http://ada-auth.org/standards/` zu finden. Darüber hinaus enthält praktisch jedes Buch über eine typisierte Sprache auch ein (oder mehrere) Kapitel über Typen in der jeweiligen Sprache.

Kapitel 4

Modelle polymorpher Typsysteme und Typen in funktionalen Sprachen

In diesem Kapitel werden die Beschreibungen aus Kapitel 2 in Richtung universell quantifizierter polymorpher Typsysteme verfeinert und Beispiele von Typen in funktionalen Sprachen behandelt. Abschnitt 4.1 führt Subtyping in Algebren ein. (Parametrische algebraische Typsysteme wurden bereits in Abschnitt 2.3 eingeführt.) Abschnitt 4.2 gibt einen kurzen Überblick über Typinferenz und das Damas-Milner Typsystem, eine Erweiterung des einfachen typisierten Lambda-Kalküls. In diesem Abschnitt wird auch das Typsystem der Sprache ML kurz erläutert. Abschnitt 4.3 beschäftigt sich mit Ansätzen von Subtyping in funktionalen Sprachen. Diese Ansätze werden auch als eine Grundlage für Typen in der objektorientierten Programmierung verwendet. Schließlich wird in Abschnitt 4.4 das PER-Modell kurz angerissen um ein tieferes Verständnis des Typbegriffs zu vermitteln.

4.1 Order-sorted Algebras

Heterogene Algebren (siehe Abschnitt 2.3) sind eine von mehreren theoretischen Grundlagen für die Einführung einer großen Zahl von Typen in Programmiersprachen. Im vorigen Kapitel wurde die Flexibilität dieser Grundlage zur Definition von allgemeinen Modulen und abstrakten Datentypen genutzt. Diese große Flexibilität ist jedoch nicht immer erwünscht, da sie leicht zu unstrukturierten, komplexen Programmen führt, in denen Routinen auf vielfältige und oft undurchschaubare Weise überladen sind. Man versucht daher, zusätzliche Strukturierungsmechanismen einzuführen, die einen „Wildwuchs“ an Typen und Routinen zu vermeiden helfen. In heterogenen Algebren geschieht dies oft dadurch, dass eine Ordnung auf Sorten eingeführt wird. Man spricht dann von einer *„order-sorted algebra“*. Durch die zusätzliche Strukturierung wird auch Subtyping ermöglicht.

Zur Wiederholung: Eine heterogene Algebra ist ein Tupel $(A_1, \ldots, A_n, \Omega)$, wobei $A_1, \ldots, A_n$ Trägermengen sind und Ω ein System von Operationen ist, deren Bedeutungen durch eine Menge von Gesetzen Δ festgelegt werden. Jede Trägermenge A_i entspricht genau einer Sorte s_i aus der Menge der Sorten $S = \{s_1, \ldots, s_n\}$. Jede Operation in Ω hat einen Typ und einen Index. Zum Beispiel kann man Typ und Index einer m-stelligen Operation $\omega \in \Omega$ in der Form

$$\omega : s_{i_1} \times \cdots \times s_{i_m} \to s_j$$

anschreiben, wobei die Indizes $j, i_1, \ldots, i_m \in \{1, \ldots, n\}$ und $s_j, s_{i_1}, \ldots, s_{i_m} \in S$. Die Signatur einer Algebra entspricht in etwa der Schnittstelle eines ADT. Sie besteht aus S, Ω und den Typen und Indizes der Operationen in Ω. Sie enthält weder die Trägermengen noch die Gesetze.

4.1.1 Verbände über Sortenmengen

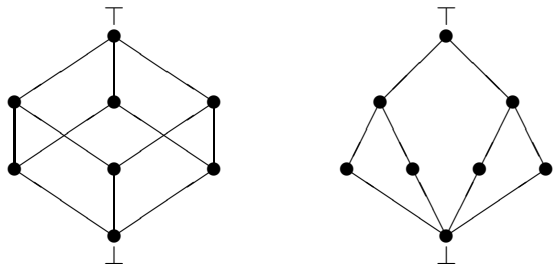
Für die Einführung einer Ordnung auf einer Algebra bietet sich ihre Signatur an, da nur sie außerhalb eines ADT sichtbar ist. Hier bietet sich vor allem die Menge S an, da ihre Elemente den Typen in Programmiersprachen entsprechen. Wir nehmen an, dass S die beiden Sorten $\top$ und $\bot$ enthält und führen einen Verband $\langle S, \leq, \sqcup, \sqcap, \top, \bot \rangle$ über S ein. Dieser hat folgende Eigenschaften:

1. $\leq$ ist eine reflexive, transitive und antisymmetrische Relation auf S (= Halbordnung von S), sodass $\bot \leq s$ und $s \leq \top$ für alle $s \in S$ gilt.

2. Für jedes $s_1 \in S$ und $s_2 \in S$ sind das Supremum $s_1 \sqcup s_2 = s_3 \in S$ (s_3 ist das kleinste Element von S mit $s_1 \leq s_3$ und $s_2 \leq s_3$) und Infimum $s_1 \sqcap s_2 = s_4 \in S$ (s_4 ist das größte Element von S mit $s_4 \leq s_1$ und $s_4 \leq s_2$) eindeutig definiert. Dabei gilt: $s \sqcup s' = s' \Leftrightarrow s \leq s' \Leftrightarrow s \sqcap s' = s$.

Ein Verband kann leicht als ein zusammenhängender Graph (*Hasse-Diagramm*) dargestellt werden. In diesen

beiden Beispielen stellt jeder Punkt ein Element von S dar:



Für je zwei Sorten s und s' gilt $s \leq s'$ genau dann wenn es im Graphen mindestens einen stets von unten nach oben führenden Pfad zwischen s und s' gibt. Auch Supremum und Infimum zweier Sorten sind im Graphen leicht zu finden, wenn man die Pfade nach oben bzw. unten verfolgt.

Die obigen Graphen kann man so deuten, dass sie das Enthaltensein der Instanzenmengen von Typen bzw. Vererbung in objektorientierten Programmen darstellen. Sorten entsprechen ja Typen in Programmiersprachen. Für $s, s' \in S$ mit $s \leq s'$ entspricht s einem Untertyp von s' und s' einem Obertyp von s. Der linke Graph stellt Mehrfach- und der rechte, abgesehen von der Sorte $\bot$, Einfachvererbung dar.

Einige wichtige Eigenschaften heterogener Algebren, auf deren Sortenmengen Verbände definiert sind, lassen sich leicht verstehen. Zwischen den Trägermengen $A_1, \ldots, A_n$ (= Wertemengen) und den entsprechenden Sorten $s_1, \ldots, s_n$ (= Typen) gelten folgende Beziehungen (für alle $i, j, k \in \{1, \ldots, n\}$):

$$\begin{aligned} s_i \leq s_j \quad &\Leftrightarrow \quad A_i \subseteq A_j \\ s_i = s_j \sqcap s_k \quad &\Leftrightarrow \quad A_i \subseteq A_j \cap A_k \end{aligned}$$

Intuitiv bedeutet das, dass jeder Untertyp eine Menge von Werten beschreibt, die eine Teilmenge der von seinem Obertyp beschriebenen Wertemenge ist. Weiters gilt, dass je zwei von Typen beschriebene Wertemengen nur solche gemeinsame Werte enthalten, die auch in der Wertemenge eines gemeinsamen Untertyps dieser zwei Typen enthalten sind. Die durch $\bot$ beschriebenen Werte sind in jeder Trägermenge enthalten; und $\top$ beschreibt die Menge aller Werte. Das heißt, für $s_i = \top$ und $s_j = \bot$ gilt

$$A_i = \bigcup_{k=1,\ldots,n} A_k \qquad \text{und} \qquad A_j = \bigcap_{k=1,\ldots,n} A_k.$$

Meist enthält A_j nur einen Wert, der oft ebenfalls durch $\bot$ dargestellt wird. Er repräsentiert undefinierte Ergebnisse von Operationen, also (nicht existierende) Ergebnisse von Endlosrekursionen bzw. Fehlerzustände.

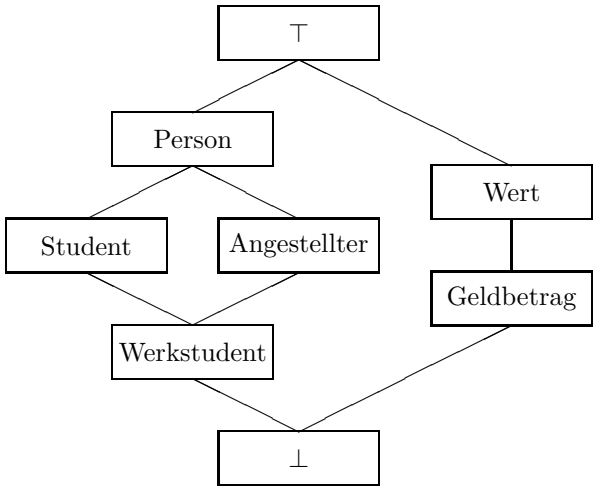
4.1.2 Polymorphe Operationen

Dadurch, dass Trägermengen teilweise ineinander enthalten sind, sind auch die Operationen automatisch polymorph. Angenommen, eine Operation $\omega \in \Omega$ mit $\omega : s_{i_1} \times \cdots \times s_{i_m} \to s_j$ ist durch die Gesetze in Δ vollständig definiert. Dann ist ω durch dieselben Gesetze gleichzeitig auch als (für $k \in \{1, \ldots, m\}$)

$$\omega : s_{i'_1} \times \cdots \times s_{i'_m} \to s_{j'} \quad \text{mit} \quad s_j \leq s_{j'} \quad \text{und} \quad s_{i'_k} \leq s_{i_k}$$

vollständig definiert. Das ist auch intuitiv klar, da jede Operation eine Menge von Eingangswerten in eine Menge von Ergebniswerten abbildet. Natürlich ist die Operation auch auf jeder beliebigen Teilmenge der Eingangswerte definiert, und das Ergebnis liegt stets auch in jeder Obermenge der tatsächlich möglichen Ergebniswerte.

Wenn man typisierte Variablen betrachtet, bedeutet das für eine Programmiersprache Folgendes: In einem Aufruf einer Routine kann an der Position eines Eingangsparameters jede Variable stehen, deren statischer Typ ein Untertyp des statischen Typs des entsprechenden formalen Parameters ist. Der aktuelle Wert der Variable zum Zeitpunkt des Routinenaufrufs ist dann in jedem Fall eine Instanz des Parametertyps. An der Position eines Ausgangsparameters kann jede Variable stehen, deren statischer Typ ein Obertyp des entsprechenden Parametertyps ist, damit der von der Prozedur zurückgegebene Wert in jedem Fall eine Instanz des statischen Variablentyps ist. Ein Durchgangsparameter muss genau denselben statischen Typ haben wie eine Variable an dieser Position, da die Variable sowohl ein Eingangsargument liefert als auch einen Ergebniswert übernimmt. Ein Beispiel verdeutlicht dies. Die Typen des Programms seien folgendermaßen angeordnet:



Damit wäre dieses Programmstück in einer Pseudosprache erlaubt:

```
procedure Beitrag(V: in Person;
                  S: in out Wert;
                  B: out Geldbetrag);
VERSICHERT: Student := ...;
VERS_LEIST: Wert := ...;
BEITRAG: Wert;
Beitrag(VERSICHERT, VERS_LEIST, BEITRAG);
```

Nehmen wir an, dass in diesem Beispiel die Prozedur `Beitrag` für einen Versicherten (erstes Argument) die Versicherungsleistungen (zweites Argument) anpasst und die Höhe des Versicherungsbeitrages (drittes Argument) ermittelt. Die Prozedur berechnet die Ergebnisse aufgrund der Werte der übergebenen Argumente. Das Ergebnis kann nur dann richtig berechnet werden, wenn alle Argumente in den von der Prozedur erwarteten Wertemengen liegen. Nachdem jede Instanz von `Student` auch eine Instanz von `Person` ist, kann man der Prozedur jeden beliebigen Wert in einer Variable vom statischen Typ `Student` als erstes Argument übergeben. Die Prozedur liefert an dritter Argumentposition einen `Geldbetrag` zurück. Da jede Instanz von `Geldbetrag` auch eine Instanz von `Wert` ist, kann eine Variable vom statischen Typ `Wert` das Ergebnis übernehmen. Für den Durchgangsparameter an zweiter Position kann nur eine Variable vom statischen Typ `Wert` verwendet werden, da diese Variable sowohl das Eingangsargument liefert als auch das Ergebnis aufnimmt.

Zusammengefasst gilt in jeder stark typisierten Sprache: Wird der Wert einer Variable x an eine andere Variable y zugewiesen, muss der statische Typ von x ein Untertyp des statischen Typs von y sein. Entsptechendes gilt auch für die Parameterübergabe. Durch diese Regel kann bei der Zuweisung zwar Information über den Typ eines Wertes verloren gehen, aber es ist niemals notwendig, Typüberprüfungen zur Laufzeit vorzunehmen.

4.2 Typinferenz und das Typsystem von ML

Im typisierten Lambda-Kalkül (siehe Abschnitt 2.1) können monomorphe Funktionen auf einfache Weise ausgedrückt werden. Beispielsweise kann man leicht Funktionen der beiden Typen `Integer` → `Integer` und `Float` → `Float` erzeugen. Man kann aber keine *generische* Funktion schreiben, die ein Argument eines beliebigen (unbekannten) Typs in ein Ergebnis vom Argumenttyp abbildet. Wir hätten zum Beispiel gerne die Möglichkeit, Funktionen des Typs $a \rightarrow a$ auszudrücken, wobei a ein Typparameter ist, für den jeder beliebige Typ eingesetzt werden kann. Solche Typparameter werden implizit als universell über die Menge aller Typen quantifiziert angesehen.

Als einen ersten Versuch zur Einführung von Typparametern in den Lambda-Kalkül erweitern wir die Konversionsregeln entsprechend. Zum Beispiel fügen wir zu den Regeln aus Abschnitt 2.1.2 unter anderem die Regeln

$$\begin{aligned}&((\lambda x{:}v.e_1{:}t_1){:}v \rightarrow t_1\ e_2{:}t_2){:}t_1\\ &\qquad\qquad \Longleftrightarrow ([t_2/v]([e_2/x]e_1{:}t_1)){:}t_1\\ &((\lambda x{:}v.e_1{:}v){:}v \rightarrow v\ e_2{:}t){:}t\\ &\qquad\qquad \Longleftrightarrow ([t/v]([e_2/x]e_2{:}t)){:}t\end{aligned}$$

ein, wobei v jeweils ein Typparameter ist. Allerdings haben wir jetzt ein Problem: Es ist im Allgemeinen nicht feststellbar, ob ein Programm typkonsistent ist oder nicht. Dieses Typüberprüfungsproblem ist genauso unentscheidbar wie viele ähnliche Ansätze.

4.2.1 Typinferenz

Statt zu versuchen, das schwierige Typüberprüfungsproblem direkt zu lösen, kann man einen Schritt weiter gehen: Alle Typbezeichner werden aus den Lambda-Ausdrücken entfernt, sodass nur mehr der ursprüngliche untypisierte Lambda-Kalkül übrig bleibt. Dann wird versucht, die Typen der Lambda-Ausdrücke aus diesen untypisierten Ausdrücken zu berechnen. Diesen Berechnungsvorgang nennt man *Typinferenz*. Praktisch bedeutet das, dass der Programmierer seine Programme schreibt, ohne Typen angeben zu müssen. Der Compiler berechnet die fehlenden Typangaben aus der Verwendung der einzelnen Programmteile. Viele moderne funktionale Programmiersprachen, vor allem ML und dessen Dialekte sowie Haskell, beruhen tatsächlich auf dem Prinzip der Typinferenz. Hier ist ein kurzer Ausschnitt aus einer Interaktion mit einem Standard-ML-Interpreter:

```
Eingabe:   fun pluszwei x = x + 2 ;
Ergebnis:  val pluszwei = fn : int -> int
Eingabe:   fun ident x = x ;
Ergebnis:  val ident = fn : 'a -> 'a
```

Die erste Eingabe definiert (aufgrund des Schlüsselwortes `fun`) die Funktion `pluszwei` mit dem Argument `x`, die als Ergebnis `x + 2` liefert, wobei `+` die ganzzahlige Addition bezeichnet. Das Ergebnis dieser Definition in der zweiten Zeile besagt, dass `pluszwei` eine Funktion von `int` nach `int` ist. Der eingebaute Compiler hat also anhand der Addition mit `2` festgestellt, dass `pluszwei` nur auf `int` definiert ist und ein Ergebnis vom Typ `int` liefert. In der dritten Zeile wird die einfache Identitätsfunktion definiert. Hier erkennt der Compiler, dass `ident` für alle Argumenttypen definiert ist und das Funktionsergebnis denselben Typ wie das Argument hat. Der Typ der Funktion ist also `'a -> 'a`, wobei `'a` einen Typparameter bezeichnet.

4.2.2 Typen in ML

Neuere funktionale Sprachen wie ML bieten vielfältige Möglichkeiten zur Typdefinition. Neben den üblichen vordefinierten Typen wie `bool`, `int`, `real` und `string` gibt es zahlreiche Typkonstruktoren. Vordefinierte Typkonstruktoren erlauben die Definition von Listen, Verbunden, Tupeln (unter anderem Paare) und Funktionen mit einfacher Syntax. Zum Beispiel sind `[1,2,3]`, `["a","b"]` und `nil` Listen der Typen `int list`, `string list` bzw. `'t list`. Letzterer Typ is polymorph, da `nil` eine Instanz von `int list`, `string list`, `bool list`, etc. ist.

Tupel werden benutzt um Instanzen unterschiedlicher Typen miteinander zu kombinieren. Zum Beispiel haben die Tupel `(5, 6)`, `(true, "fact", 7)` und `(true, nil)` die Typen `int*int`, `bool*string*int` bzw. `bool*('t list)`. Typen von Verbunden und Funktionen haben die in Abschnitt 2.1.2 und 2.1.3 eingeführte Form. Diese Typen können aber Typparameter enthalten. So ist z.B. {`name:string,id:int`} ein Verbundtyp und `'t list -> bool` ein Funktionstyp. (Tupel sind eigentlich nur Verbunde mit den Labeln $1, 2, \ldots$ in abgekürzter Schreibweise.)

Zusätzlich zu diesen vordefinierten Typen und Typkonstruktoren gibt es drei weitere Möglichkeiten, Typen zu definieren. Die erste Möglichkeit erlaubt einfach nur, neue Namen für existierende Typen einzuführen, wie diese Beispiele zeigen:

```
type intpair = int*int;
type 'a pair = 'a * 'a;
type boolpair = bool pair;
```

Die zweite Möglichkeit, einen Typ zu definieren, beschreibt, wie Werte des neuen Typs aussehen und erzeugt werden. Hier sind einige Beispiele:

```
datatype color = red | green | blue;
datatype publications =
    nopubs | journal of int | conf of int;
datatype 't stack =
    empty | push of 't * 't stack;
```

Die erste Zeile definiert einfach nur einen Aufzählungstyp mit den gegebenen Instanzen. Der Typ in der zweiten Zeile spezifiziert, dass einige Instanzen des Typs mit Parametern versehen sind. Zum Beispiel sind `journal(3)` und `conf(7)` Instanzen dieses Typs. Solche Datentypen können auch rekursiv sein, wie die dritte Zeile zeigt. Instanzen dieses Typs sind `empty`, `push(2,empty)`, `push(7,push(2,empty))`, usw. Auch der vordefinierte Typ `'a list` kann auf diese Weise definiert werden:

```
datatype 'a list = nil | :: of 'a * 'a list;
```

Die dritte Möglichkeit, einen Typ zu definieren, versteckt die interne Repräsentation eines ADT:

```
abstype 'a lifo = Stack of 'a list
with exception error;
     val create = Stack nil;
     fun push(x, Stack xs) = Stack(x::xs);
     fun pop(Stack nil) = raise error;
     |   pop(Stack(x::xs)) = Stack xs;
     fun top(Stack nil) = raise error;
     |   top(Stack(x::xs)) = x;
end;
```

Der ADT `'a lifo` exportiert nur die Ausnahme `error`, den Wert `create` und die Funktionen `push`, `pop` und `top`. Die interne Darstellung (= die Liste im Argument von `Stack`) ist nach außen nicht sichtbar. Daher sind auch die Listenoperationen auf Instanzen von `'a lifo` nicht direkt anwendbar, sonder nur über `Stack`.

ML bietet auch ein umfangreiches Modulkonzept. Zum Beispiel kann man einen Stack auch folgendermaßen definieren:

```
structure S = struct
  exception error;
  type 't Stack = 't list;
  val create = Stack nil;
  fun push(x, Stack xs) = Stack(x::xs);
  fun pop(Stack nil) = raise error;
  |   pop(Stack(x::xs)) = Stack xs;
  fun top(Stack nil) = raise error;
  | top(Stack(x::xs)) = x;
end;
```

Im Gegensatz zu abstrakten Datentypen verstecken Module in ML die interne Datenrepresentation nicht. Auf alle Komponenten des Moduls kann mittels Punkt-Notation zugegriffen werden. Ein Stack mit einem Element kann also so erzeugt werden:

```
val v = S.push(2, S.create);
```

Als Alternative dazu kann ein Modul mittels `open` explizit geöffnet werden, sodass alle Namen im Modul daraufhin auch ohne Punkt-Notation zugänglich sind:

```
open S;
val v = push(2, create);
```

Sehr oft möchte man aber einen Teil des Inhalts eines Moduls vor unberechtigten Zugriffen schützen. Für diesen Zweck bietet ML Signaturen, die einem Typ eines Moduls entsprechen und die sichtbare Funktionalität des Moduls einschränken können. Zum Beispiel beschreibt die Signatur `st`, dass ein Modul dieses Typs einen nicht weiter beschriebenen Typ `q` und die Funktionen `push` und `pop` der Typen `'t*q->q` bzw. `q->q` exportiert:

```
signature st = sig
  type q;
  val push: 't * q -> q;
  val pop: q -> q;
end;
```

Ein neues Modul mit dieser Funktionalität kann man leicht erzeugen, indem man `st` mit `S` in Beziehung setzt:

```
structure S1:st = S;
```

In `S1` sind `create` und `top` nicht mehr zugreifbar. Auf jedes Modul kann über jede gewünschten Signatur zugegriffen werden, solange der Inhalt des Moduls mit der Signatur in Übereinstimmung gebracht werden kann. Eine Signatur kann beispielsweise als Parametertyp einer Funktion angegeben sein. Jedes Modul, das dieser Signatur entspricht, kann dann als Argument übergeben werden. Ein Modul entspricht auch mehreren Signaturen, wie das nächste Beispiel zeigt:

```
signature stringStack = sig
  exception error;
  type string Stack;
  val create: string Stack;
  val push: string * string Stack
                  -> string Stack;
  val pop: string Stack -> string Stack;
  val top: string Stack -> string;
end;

structure S2:stringStack = S;
```

Durch `stringStack` wird der Inhalt der Stacks in `S2` auf Zeichenketten eingeschränkt. `S2` ist also nicht mehr polymorph.

4.2.3 Ein Typinferenz-Algorithmus

Für eine einfache funktionale Sprache kann ein Algorithmus zur Typinferenz leicht formal definiert werden. Als Beispiel nehmen wir eine Sprache, in der es neben Parameternamen und Funktionen nur noch Konstanten (wie Zahlen und Zeichenketten, aber auch vordefinierte Operationen darauf) gibt; es werden also nur Basistypen und Funktionstypen, aber keine strukturierten Typen unterstützt. In der Literatur wird eine solche Sprache oft durch $\lambda_t^{\rightarrow}$ bezeichnet. Typen in dieser Sprache sollen durch einen Typinferenz-Algorithmus berechnet werden. Die Funktion PT, die diesen Algorithmus implementiert, ist folgendermaßen definiert, wobei c eine Konstante, x einen Parameternamen, v einen Typparameter, s bzw. t einen Typ, e einen untypisierten Lambda-Ausdruck und f einen typisierten Lambda-Ausdruck bezeichnet:

$$
\begin{aligned}
\mathrm{PT}(c) &= \emptyset \triangleright c{:}t \quad (t \text{ enthält keine Typparameter})\\
\mathrm{PT}(x) &= \{x{:}v\} \triangleright x{:}v \quad (v \text{ neu})\\
\mathrm{PT}(e\,e') &= \text{let } \Gamma \triangleright f{:}t = \mathrm{PT}(e);\\
&\qquad \Gamma' \triangleright f'{:}t' = \mathrm{PT}(e');\\
&\qquad \theta = \mathrm{unify}(\{s = s' \mid x{:}s \in \Gamma;\ x{:}s' \in \Gamma'\}\\
&\qquad\qquad \cup \{t = t' \rightarrow v\}) \quad (v \text{ neu})\\
&\quad \text{in } \theta\Gamma \cup \theta\Gamma' \triangleright \theta(f\ f'){:}\theta v\\
\mathrm{PT}(\lambda x.e) &= \text{let } \Gamma \triangleright f{:}t = \mathrm{PT}(e);\\
&\quad \text{in if } x{:}s \in \Gamma \text{ für ein beliebiges } s\\
&\qquad \text{then } \Gamma \setminus \{x{:}s\} \triangleright (\lambda x{:}s.f){:}s \rightarrow t\\
&\qquad \text{else } \Gamma \triangleright (\lambda x{:}v.f){:}v \rightarrow t \quad (v \text{ neu})
\end{aligned}
$$

Γ (Gamma) ist eine Umgebung mit (also eine Menge von) Typzuweisungen an Parameternamen. Die erste Gleichung ordnet einer Konstante c einfach nur den (parameterfreien) Typ dieser Konstante zu, wobei die Umgebung leer bleibt. Dieser Typ muss eindeutig definiert sein. Zum Beispiel ist `int` der Typ von `1` und `int` $\rightarrow$ (`int` $\rightarrow$ `int`) der Typ von `+`, dem Operator zur Addition zweier ganzer Zahlen.

Die zweite Gleichung weist einem Parameternamen einen beliebigen neuen Typparameter als Typ zu. Wenn dieser Typ später im Algorithmus nicht weiter eingeschränkt wird, kann der Parameter durch beliebige Ausdrücke ersetzt werden.

Die dritte Gleichung behandelt Anwendungen (= Funktionsaufrufe). Sowohl für die Funktion als auch dessen Parameter werden durch rekursive Aufrufe von PT entsprechende Umgebungen und typisierte Ausdrücke berechnet. Als Kern des Algorithmus müssen die den Parameternamen in den Umgebungen zugeordneten Typen miteinander unifiziert werden. Es wird ein allgemeinster Unifikator θ berechnet, der auf alle Typen in den Umgebungen und Ausdrücken angewendet wird. Durch die Unifikation werden Typparameter, sofern notwendig, aneinander und an Typen gebunden.

Die letzte Gleichung behandelt Abstraktionen. Falls der Parameter x im Funktionsrumpf e vorkommt, kommt eine Typzuweisung an x auch in der e entsprechenden Umgebung vor. In diesem Fall entspricht der zugewiesene Typ dem Typ des Parameters. Die Typzuweisung wird aus der Umgebung entfernt, da x durch die Abstraktion gebunden wird, also weiter außen nicht in derselben Bedeutung vorkommt. Falls x nicht in e vorkommt, ist der Parametertyp beliebig; es kann dafür ein neuer Typparameter verwendet werden.

Da die Berechnung eines allgemeinsten Unifikators eine zentrale Bedeutung für die Typinferenz hat, wollen wir im Folgenden die Funktion "unify" formal definieren. Das Ergebnis der Funktion ist entweder eine Liste von Typparameterersetzungen $[t_1/v_1]\circ\cdots\circ[t_n/v_n]$ oder "fail". In letzterem Fall schlägt die Unifikation fehl; das Programm, auf das PT angewendet wird, ist in diesem

Fall nicht typkonsistent.

$$\begin{aligned}
\text{unify}(\emptyset) &= \emptyset \\
\text{unify}(E \cup \{b = b'\}) &= \text{if } b \neq b' \text{ then fail} \\
&\quad\ \text{else unify}(E) \\
\text{unify}(E \cup \{v = t\}) &= \text{if } v = t \text{ then unify}(E) \\
&\quad\ \text{else if } v \text{ occurs in } t \text{ then fail} \\
&\quad\ \text{else unify}([t/v]E) \circ [t/v] \\
\text{unify}(E \cup \{s \to s' = t \to t'\}) &= \\
&\quad\ \text{unify}(E \cup \{s = t,\ s' = t'\})
\end{aligned}$$

Nach demselben Prinzip wurden zahlreiche weitere Typinferenz-Algorithmen für komplexere Sprachen wie ML entwickelt. Eine Analyse dieser Algorithmen würde aber den Rahmen des Skriptums sprengen.

4.2.4 Transparenz und Entscheidbarkeit von Typinferenz

Typinferenz führt nicht immer zum gewünschten Ergebnis. Sehr oft ist der Compiler nicht in der Lage, einen allgemeinsten Typ eines Ausdrucks zu bestimmen. In diesen Fällen gibt der Compiler eine Fehlermeldung aus. Zum Beispiel, in der Funktion

```
fun max(x:int, y) = if x > y then x else y;
```

kann der Compiler nur anhand der expliziten Typangabe von `x` (`:` gefolgt von `int`) erkennen, dass eine Funktion `>` vom Typ `(int*int)->bool` verlangt wird. Ohne diese Information wäre auch `(real*real)->bool` ein möglicher Kanditat, aber nicht `('t*'t)->bool`, da `>` nicht von allen Typen `'t` akzeptiert wird. Ohne explizite Typangabe von `x` würde der Compiler eine Fehlermeldung liefern. In der Praxis ist für einen Programmierer leider nicht immer gleich erkennbar wo der Fehler liegt, da die Ursache dafür nicht unbedingt an der Stelle liegen muss, an der der Fehler gemeldet wird. Eine scheinbar harmlose Änderung eines Programms kann zu einer Fehlermeldung an einer von der Änderung anscheinend gar nicht betroffenen Stelle führen.

Das Problem, dass ein Parameter mehr als nur einen konkreten Typ aber doch nicht jeden beliebigen Typ haben kann (z.B. nur `int` oder `real`), wurde in Haskell durch Typklassen zu einem großen Teil gelöst: Eine Typklasse definiert eine Klasse von Typen, die gemeinsame Operationen haben. Beispielsweise gehören die Typen `Int` und `Real` zur Typklasse `Num`, die Operationen wie `+` und `>` definiert. Trifft der Typinferenzalgorithmus auf einen dieser Operatoren, dann führt er einen neuen Typparameter ein, beschränkt diesen jedoch auf `Num`. Solche eingeschränkten Typparameter sind nur mit Typen unifizierbar, die zur Typklasse `Num` gehören (Instanzen davon sind), oder mit anderen Typparametern, auf denen es keine oder nur kompatible Einschränkungen gibt. Das Konzept der Typklassen entspricht im Wesentlichen der gebundenen Generizität (siehe Kapitel 5). Ursachen für vom Compiler gefundene Typfehler sind jedoch auch zusammen mit Typklassen oft nur schwer zu verstehen und zu finden.

Theoretische Ergebnise besagen, dass das Typinferenzproblem im Allgemeinen unentscheidbar ist. Jedoch gibt es ein etwas eingeschränkteres, dafür aber entscheidbares polymorphes Typsystem, das fast so mächtig wie das uneingeschränkte ist. Es wird in den oben genannten funktionalen Sprachen im Zusammenhang mit Typinferenz verwendet. Die Einschränkung dieses Typsystems ist, dass rekursive Funktionen innerhalb des Rumpfes der Funktionsdefinition nur monomorph verwendet werden dürfen. Das heißt, alle direkt rekursiven Aufrufe müssen dieselben Argumenttypen haben. Alle anderen (rekursionsfreien oder indirekt rekursiven) Aufrufe können polymorph sein, sodass jeder Aufruf andere Argumenttypen haben kann. Alle Versuche, diese Einschränkung aufzuheben, führten bislang zu Typsystemen, in denen das Typüberprüfungsproblem unentscheidbar ist. Es gibt aber Tendenzen, diese unentscheidbaren Typsysteme in der Praxis einzusetzen, da die Unentscheidbarkeit nur in wenigen praktisch verwendbaren Programmen zum Problem wird. Bei unentscheidbaren Programmen kommt der Compiler in eine Endlosschleife oder bricht ab weil er zu wenig Speicher hat.

4.3 Funktionale Ansätze für Subtyping

Algebren können sowohl mit Generizität als auch mit Untertypen umgehen, wie wir in Abschnitt 2.3 und 4.1 gesehen haben. Aber Algebren haben einen Nachteil: Es fehlt ihnen an Ausdrucksstärke. Einerseits können Algebren, wie bereits erwähnt, nicht mit globalen Zuständen (Variablen) umgehen, was besonders für objektorientierte Sprachen ein großer Nachteil ist. Andererseits ist die Äquivalenz zweier Algebren innerhalb der Theorie in der Regel nicht formal nachweisbar, auch wenn die Algebren (mit ausreichendem Meta-Wissen) ganz offensichtlich äquivalent sind.

Funktionale Ansätze haben diese Nachteile nicht; sie sind daher für die meisten Anwendungen ausdrucksstärker. In Abschnitt 4.2 haben wir gesehen, dass funktionale Sprachen mit einem sehr mächtigen generischen Typsystem ausgestattet sein können. Bisher fehlt jedoch jede Unterstützung für Subtyping. Im Folgenden werden funktionale Ansätze für Subtyping eingeführt.

4.3.1 Einfache Untertypbeziehungen

Ähnlich wie bei „order-sorted algebras“ beschreibt ein Typ im Wesentlichen eine Wertemenge, und über den

Typen ist ein Verband definiert, der der Enthaltensein-Relation der Wertemengen entspricht.[1] Dieser Verband ist jedoch nicht direkt vorgegeben, sondern leitet sich aus den Eigenschaften der Typen ab. Wir müssen also für alle Arten von Typen Regeln einführen, die die Untertypbeziehungen zwischen gegebenen Typen herleiten. Eine Untertypbeziehung $s \leq t$ besagt, dass der Typ s ein Untertyp des Typs t ist. Die Untertypbeziehungen müssen dem Ersetzbarkeitsprinzip entsprechen:

> Ein Typ s ist ein Untertyp eines Typs t wenn eine Instanz von s überall verwendet werden kann wo eine Instanz von t erwartet wird.

Dieses Ersetzbarkeitsprinzip definiert im Wesentlichen den Begriff Subtyping.

Die erste Regel besagt, dass jeder Typ t ein Untertyp von sich selbst ist; die Untertyprelation ist reflexiv:

$$\Gamma \vdash t \leq t \tag{4.1}$$

Diese Regel wird gelesen als „Γ leitet $t \leq t$ ab“, wobei Γ eine Umgebung mit Subtyping-Annahmen ist. Diese Umgebung wird erst in Abschnitt 4.3.2 gebraucht. Wir führen sie aber schon hier ein, damit wir die Regeln dann nicht mehr ändern müssen.

Jeder Typ t ist Untertyp des allgemeinsten Typs $\top$:

$$\Gamma \vdash t \leq \top \tag{4.2}$$

$\top$ hat keinerlei interne Struktur, ebenso wie der Typ $\bot$, der Fehlerzustände bezeichnet. $\bot$ ist ein Untertyp jedes Typs:

$$\Gamma \vdash \bot \leq t \tag{4.3}$$

Die Untertyprelation ist auch transitiv:

$$\frac{\Gamma \vdash s \leq t' \quad \Gamma \vdash t' \leq t}{\Gamma \vdash s \leq t} \tag{4.4}$$

Diese Regel ist zu lesen als „Γ leitet $s \leq t$ ab wenn es einen Typ t' gibt, sodass Γ auch $s \leq t'$ und $t' \leq t$ ableitet“, wobei s und t beliebige Typen sind. Der horizontale Strich steht für die logische Implikation. Mehrere Ausdrücke über dem Strich sind konjunktiv (mittels Und) verknüpft.

Wir führen nun eine Regel für Funktionstypen ein:

$$\frac{\Gamma \vdash s' \leq s \quad \Gamma \vdash t \leq t'}{\Gamma \vdash s \to t \leq s' \to t'} \tag{4.5}$$

Ein Funktionstyp ist also ein Untertyp eines anderen Funktionstyps wenn die Einschränkungen auf die Eingangsparameter des ersten Funktionstyps weniger streng als die auf die Eingangsparameter des zweiten sind. Bei den Ergebnistypen ist es genau umgekehrt: Ein Untertyp eines Funktionstyps hat auch einen Untertyp des Ergebnistyps des ersten Funktionstyps als Ergebnistyp. Dies kann man mit einem kleinen Beispiel rechtfertigen: Der ganzzahlige Bereich $\{4, \ldots, 6\}$ ist offensichtlich ein Untertyp von $\{3, \ldots, 7\}$, da eine Zahl im Bereich von 4 bis 6 überall verwendet werden kann wo eine Zahl im Bereich von 3 bis 7 erwartet wird. Eine Funktion vom Typ $\{3, \ldots, 7\} \to \{4, \ldots, 6\}$ kann überall verwendet werden, wo eine Funktion vom Typ $\{4, \ldots, 6\} \to \{3, \ldots, 7\}$ erwartet wird, da das Ergebnis der Funktion auf jeden Fall im erwarteten Bereich liegt und zumindest der erwartete Argumenttyp-Bereich abgedeckt ist. Diese Art der Untertypbeziehung für Eingangsparameter heißt *kontravariant*, da Funktionen in Untertypen Obertypen als Eingangsparameter erlauben—die Typen variieren in entgegengesetzte Richtungen. Ergebnistypen sind dagegen *kovariant*, da Funktionen in Untertypen auch Untertypen zurückgeben—Typen variieren in dieselbe Richtung.

Für Verbundtypen gilt diese Regel:

$$\frac{(\forall\, i \leq m)\, \Gamma \vdash s_i \leq t_i}{\Gamma \vdash \{l_1{:}s_1, \ldots, l_n{:}s_n\} \leq \{l_1{:}t_1, \ldots, l_m{:}t_m\}} \quad (m \leq n) \tag{4.6}$$

Auch dies wird durch ein Beispiel klar:

$$t = \{\texttt{name:string}, \texttt{alter:}\{0, \ldots, 120\}\}$$
$$s = \{\texttt{name:string}, \texttt{alter:}\{18, \ldots, 65\}, \texttt{mnr:int}\}$$

Der Verbundtyp t könnte allgemeine Informationen zu einer Person enthalten, s zu einem Studenten. Offensichtlich gilt $s \leq t$, da es bei Verwendung einer Instanz von s (wo eine Instanz von t erwartet wird) folgende Möglichkeiten gibt: Es kann nur auf die Komponenten `name` und `alter` zugegriffen werden. Auf `mnr` wird nicht zugegriffen, da diese Komponente in t nicht vorkommt. Ein lesender Zugriff auf eine dieser beiden Komponenten liefert auf jeden Fall ein Ergebnis vom in t spezifizierten Typ. Schreibende Zugriffe auf einen Verbund gibt es in einer rein funktionalen Sprache nicht; diesen Fall brauchen wir also nicht betrachten. Aber Achtung: Wenn die Regel 4.6 für imperative bzw. objektorientierte Sprachen adaptiert wird, werden auch Schreibzugriffe möglich. Dann müssen schreib- und lesbare Komponenten des Unter- und Obertyps jeweils gleiche Typen haben; $s \leq t$ würde also nicht gelten, wenn `alter` eine schreibbare Komponente wäre.

Die Regel für Tupel entspricht der für Verbunde, da Tupel als Spezialfall von Verbunden gesehen werden.

Für Variantentypen gilt diese Regel:

$$\frac{(\forall\, i \leq m)\, \Gamma \vdash s_i \leq t_i}{\Gamma \vdash [l_1{:}s_1, \ldots, l_m{:}s_m] \leq [l_1{:}t_1, \ldots, l_n{:}t_n]} \quad (m \leq n) \tag{4.7}$$

[1] Genaugenommen muss es sich nicht nur um einen Verband, sondern um Ideale handeln, um wichtige Eigenschaften funktionaler Sprachen beibehalten zu können. Aber auf diese Details wollen wir hier nicht eingehen, da wir eher am Prinzip und (später) dessen Realisierung in imperativen Sprachen interessiert sind als an der Theorie.

Einen Spezialfall dieser Regel haben wir schon einige Male verwendet. Zum Beispiel ist [Di:Tag, Mi:Tag] ein Untertyp von [Mo:Tag, Di:Tag, Mi:Tag, Do:Tag]. Angenommen, es wird eine Variante eines Untertyps in einer Case-Anweisung verwendet wo eine Instanz (Variante) eines Obertyps erwartet wird. Dann enthält die Case-Anweisung Alternativen für alle Label im Obertyp. Die Menge der Label im Untertyp ist eine Teilmenge davon, sodass die Case-Anweisung auf jeden Fall eine Alternative für das in der Variante vorhandene Label enthält. Der Typ des Wertes mit diesem Label ist ein Untertyp des erwarteten Typs. Da nur lesend auf diesen Wert zugegriffen wird, erfüllt dieser Wert auf jeden Fall die Erwartungen.

Obige Regeln definieren implizit auch eine strukturelle Typäquivalenz: Zwei Typen s und t sind genau dann äquivalent wenn sowohl $s \leq t$ als auch $t \leq s$ gilt.

4.3.2 Rekursive Typen

Typen, vor allem Verbundtypen, sind in der Praxis sehr häufig rekursiv. Zum Beispiel ist der Typ eines Knotens in einem Baum rekursiv, da der Name des Typs im Verbundtyp selbst vorkommt:

$$\texttt{baum} = \{\texttt{i:int}, \texttt{l:baum}, \texttt{r:baum}\}$$

Rekursive Typen werfen einige Probleme auf. Ein sehr offensichtliches Problem ist das der Typdarstellung: Wir wollen einen Typ unabhängig von dessen Namen beschreiben, um strukturelle Typäquivalenz nutzen zu können. Das ist in der oben gezeigten Form aber nicht ohne weiteres möglich, da der Name im Typ verwendet wird. Es können nicht alle Vorkommen dieses Namens durch den entsprechenden Typ ersetzt werden; der Typ wäre sonst unendlich groß.

Ein einfacher „Trick“ hilft bei der Beseitigung dieses Problems: Statt des Typnamens wird ein Typparameter zur Kennzeichnung aller rekursiver Vorkommen des Typs verwendet. Obiger Typ wird zum Beispiel so dargestellt:

$$\mu v.\{\texttt{i:int}, \texttt{l:}v, \texttt{r:}v\}$$

Der Fixpunktoperator μ bindet den Typparameter v an den Typausdruck.

Auf den ersten Blick mag es scheinen, als ob sich dadurch nicht sehr viel geändert hätte: Der Name, mit dessen Hilfe Rekursion ausgedrückt wird, ist nur auf syntaktisch andere Weise eingeführt worden. Bedeutung bekommt diese Änderung erst durch das Hinzufügen zweier Regeln, über die Äquivalenzen rekursiver Typen definiert werden:

1. α-Konversionsregel:

$$\mu v.t = \mu u.[u/v]t \quad (u \notin \mathit{free}(\mu v.t))$$

2. Faltungsregel:

$$\mu v.t = [\mu v.t/v]t$$

Ersetzungen und die Funktion *free* sind so wie für den Lambda-Kalkül in Abschnitt 2.1 definiert, jedoch wird statt λ stets μ verwendet, und Namen stehen statt für formale Parameter stets für Typparameter. Ähnlich wie im Lambda-Kalkül können durch Anwendung der ersten Regel Typparameter beinahe beliebig umbenannt werden. Der neue Typparameter darf im Typausdruck t nur noch nicht frei (also nicht durch μ gebunden) vorkommen. Die zweite Regel hat mehrere Bedeutungen:

- Wenn der Typparameter v in t nicht vorkommt, ist $\mu v.t$ äquivalent zu t.
- Sonst (v kommt in t vor) können alle Vorkommen von v in t beliebig durch den Typ selbst ($\mu v.t$) ersetzt werden; das heißt, der Typ wird aufgefaltet. Dieser Vorgang ist beliebig oft wiederholbar, sodass der Typ im Extremfall (kleinster Fixpunkt) unendlich groß wird. In umgekehrter Richtung wird die Regel zum Falten rekursiver Typen eingesetzt.

Die folgenden Typausdrücke sind äquivalent, wie sich durch wiederholte Anwendungen obiger Regeln leicht zeigen lässt:

- $\mu v.\{\texttt{i:int}, \texttt{l:}v, \texttt{r:}v\}$
- $\{\texttt{i:int}, \texttt{l:}\mu u.\{\texttt{i:int}, \texttt{l:}u, \texttt{r:}u\},$
  $\texttt{r:}\mu v.\{\texttt{i:int}, \texttt{l:}v, \texttt{r:}\mu w.v\}\}$
- $\{\texttt{i:int}, \texttt{l:}\mu u.\{\texttt{i:int}, \texttt{l:}u, \texttt{r:}u\},$
  $\texttt{r:}\{\texttt{i:int}, \texttt{l:}\mu v.\{\texttt{i:int}, \texttt{l:}v, \texttt{r:}v\},$
  $\texttt{r:}\mu v.\{\texttt{i:int}, \texttt{l:}v, \texttt{r:}v\}\}\}$

Ein weiteres Problem ist die Definition von Subtyping für rekursive Typen. Es ist oft nicht gleich einsichtig, wie das Ersetzbarkeitsprinzip und die oben definierten Regeln auf rekursive Typen anwendbar sind, da hierbei möglicherweise unendliche Strukturen miteinander verglichen werden müssen. Trotzdem gibt es eine überraschend einfache und entscheidbare Regel zur Definition von Subtyping für rekursive Typen, die davon ausgeht, dass auch nicht identische Typparameter zueinander in einer Untertypbeziehung stehen können. Dazu benötigen wir die bereits oben eingeführte Umgebung Γ. Diese enthält Subtyping-Annahmen für Typparameter: Wenn $u \leq v$ in Γ enthalten ist kann man davon ausgehen, dass jeder Typ, für den u steht, ein Untertyp jeden Typs ist, für den v steht. Dies wird durch folgende Regel ausgedrückt:

$$\Gamma \cup \{u \leq v\} \vdash u \leq v \tag{4.8}$$

Die Subtyping-Regel für rekursive Typen ist:

$$\frac{\Gamma \cup \{u \leq v\} \vdash s \leq t}{\Gamma \vdash \mu u.s \leq \mu v.t} \quad \begin{array}{r}(u \notin \mathit{free}(t); v \notin \mathit{free}(s);\\ u, v \notin \mathit{free}(\Gamma))\end{array} \tag{4.9}$$

Die Untertypbeziehung zwischen zwei rekursiven Typen ist also genau dann gegeben, wenn die rekursionsfreien Varianten dieser Typen (mit freien Typparametern statt der gebundenen Typparameter) zueinander in einer Untertypbeziehung stehen, wobei Subtyping-Annahmen für die Typparameter gelten. Die Nebenbedingungen der Regel besagen, dass für die Typparameter Namen zu wählen sind, die weder im anderen Typ frei vorkommen noch in Γ bereits enthalten sind. Mit Hilfe dieser Regel ist die Untertypbeziehung zwischen rekursiven Typen entscheidbar, ohne auf unendliche Strukturen zurückgreifen zu müssen.

Beispielsweise gilt die Untertypbeziehung

$$\mu u.\{\texttt{i:int}, \texttt{f:int}\to u, \texttt{next:}u\} \leq \mu v.\{\texttt{i:int}, \texttt{f:int}\to v\}.$$

Aber die beiden Typen $\mu u.\{\texttt{i:int}, \texttt{f:}u\to u, \texttt{next:}u\}$ und $\mu v.\{\texttt{i:int}, \texttt{f:}v\to v\}$ können durch obige Regeln nicht in eine Untertypbeziehung gesetzt werden. Der Grund dafür ist, dass die Untertyprelation zwischen Funktionstypen kontravariante Eingangs- und kovariante Ergebnisparametertypen verlangt. Tatsächlich würde eine Untertypbeziehung zwischen diesen beiden Typen auch dem Ersetzbarkeitsprinzip widersprechen. Wie wir in Kapitel 5 sehen werden, sind kontravariante Eingangsparametertypen ein Grund dafür, dass Subtyping nicht so mächtig sein kann wie man es sich in der Praxis wünschen würde.

An dieser Stelle des Skriptums wäre die Diskussion einer Sprache angebracht, die Typinferenz (mit parametrischen Typen) wie in ML mit Subtyping verbindet. Leider haben sich bisher alle Ansätze für eine Verbindung von Typinferenz mit Subtyping als nicht zielführend erwiesen. Aus theoretischen Gründen dürfte es kaum möglich sein, eine sinnvolle Verbindung dieser Sprachkonzepte zu erzielen. Es gibt zwar funktionale Sprachen mit Typinferenz, die auch objektorientierte Erweiterungen mit Subtyping anbieten, aber in diesen Sprachen ist Typinferenz dort nicht anwendbar, wo Subtyping verwendet wird. Die Kombination von parametrischem Polymorphismus mit Subtyping ist aber möglich, wie wir in Kapitel 5 sehen werden.

4.4 Das PER-Modell

Im Großteil dieses Skriptums steht die statische Überprüfbarkeit von Typkonsistenz im Mittelpunkt der Betrachtungen. In diesem Abschnitt wollen wir aber kurz auf die semantische Bedeutung von Typen eingehen. Bisher sind wir davon ausgegangen, dass ein Typ einfach nur eine Menge von Instanzen beschreibt. Das PER-Modell beschreibt die Semantik von polymorphen Typen etwas genauer. PER steht für *„partial equivalence relation“*. Eine Relation R auf einer Menge W ist eine Menge von Paaren (v, w) mit $v, w \in W$; man schreibt $v\,R\,w$ für (v, w) in R. Eine Relation R ist eine partielle Äquivalenzrelation (PER) auf W genau dann wenn R eine symmetrische ($v\,R\,w \Leftrightarrow w\,R\,v$) und transitive ($u\,R\,v \wedge v\,R\,w \Rightarrow u\,R\,w$), aber nicht notwendigerweise reflexive ($w\,R\,w$) Relation auf W ist. Jeder Typ A entspricht einer PER, und die Menge der durch A beschriebenen Werte ist die Menge der Äquivalenzklassen von A, das ist die Menge $\{\{w\}_A \mid w\,A\,w\}$, wobei $\{w\}_A = \{v \mid v\,A\,w\}$.

Um zu verstehen wozu PERs dienen betrachten wir Untertypbeziehungen in funktionalen Sprachen. Wie wir gesehen haben ist bei Verbunden ein Typ ein Untertyp eines anderen Typs wenn

1. die Instanzen des Untertyps eine Teilmenge der des Obertyps sind und/oder
2. der Untertyp alle Komponenten seines Obertyps und möglicherweise zusätzliche Komponenten hat.

Der erste Fall lässt sich auf einfache Weise durch Teilmengen von Wertemengen beschreiben, wie dies in den vorhergehenden Abschnitten gemacht wurde. Beim zweiten Fall stoßen wir dabei aber auf Probleme. Mit Hilfe des PER-Modells lassen sich beide Fälle, wenn auch weniger anschaulich, problemlos beschreiben. Angenommen, A und B beschreiben Typen als PER, sodass $B \leq A$ gilt, B also ein Untertyp von A ist. Wenn, wie im ersten Fall, B eine Teilmenge der Instanzen von A beschreiben soll, kann man für B einfach eine Teilmenge der Paare in A nehmen, so dass B nur eine Teilmenge der Äquivalenzklassen von A beschreibt. Wenn z.B. $B = \{(a,a),(a,b),(b,a),(b,b)\}$ und $A = B \cup \{(c,c),(c,d),(d,c),(d,d)\}$, dann ist die Menge der Äquivalenzklassen von B gleich $\{\{a,b\}\}$ und die der von A gleich $\{\{a,b\},\{c,d\}\}$.

Der zweite Fall ist etwas komplizierter: Wenn $B \leq A$ gilt weil A weniger Komponenten als B hat, dann unterscheidet B genauer zwischen verschiedenen Werten als A, da B mehr Information als A hat. Daher trifft B genauere Einteilungen in Äquivalenzklassen. Wenn v und w für B in derselben Äquivalenzklasse liegen, dann müssen v und w natürlich auch für A in derselben Äquivalenzklasse liegen. Wenn v und w für A in unterschiedlichen Äquivalenzklassen liegen, dann liegen sie auch für B in unterschiedlichen Äquivalenzklassen. Trotzdem ist es möglich, dass v und w für B in verschiedenen Äquivalenzklassen liegen, obwohl sie für A, aufgrund der eingeschränkteren Information, zur selben Äquivalenzklasse gehören. Auch in diesem Fall ist B eine Teilmenge von A. Wenn z.B. $B = A \setminus \{(c,d),(d,c)\}$ (A wie oben), dann ist die Menge der Äquivalenzklassen von B gleich $\{\{a,b\},\{c\},\{d\}\}$. Zusammengefasst

gilt daher $B \leq A$ genau dann wenn $B \subseteq A$ gilt. Allerdings gilt im Allgemeinen nicht, dass die Menge der Instanzen von B eine Teilmenge der von A ist, da Instanzen Äquivalenzklassen sind, die sich für A und B unterscheiden können.

Um die Unterschiede zwischen den Modellen von Typen als Mengen bzw. Relationen klarer zu machen gehen wir davon aus, dass W die Menge aller Werte im Mengen-Modell ist. Ein Wert im PER-Modell ist eine Teilmenge von W. Im Mengen-Modell könnte man einen Verbundtyp so interpretieren, dass seine Instanzen beliebig viele Komponenten haben können, von denen aber nur die im Verbundtyp spezifizierten sichtbar sein müssen. Ein Untertyp dieses Verbundtyps enthält daher auch eine Teilmenge dieser Instanzen. In Abschnitt 5.3 werden wir eine objektorientierte Erweiterung einer Logik-Sprache sehen, die auf genau dieser Typinterpretation aufbaut. In klassischen imperativen Sprachen ist diese Typinterpretation jedoch nur teilweise richtig, da jede Instanz eines Typs ja nur einen gewissen, vom Typ der Instanz abhängigen Speicherplatz verbrauchen darf. In objektorientierten Sprachen wie Smalltalk, Eiffel und Java, in denen bei der Parameterübergabe nicht die Objekte selbst kopiert werden sondern nur Zeiger darauf, ist diese Typinterpretation aber durchaus verwendbar.

Im PER-Modell wird dieses Problem dadurch gelöst, dass man den Begriff Wert als die Menge aller Elemente von W auffasst, die die gleichen sichtbaren Komponenten haben. Die unsichtbaren Komponenten spielen keine Rolle. Daher ist das PER-Modell für klassische imperative Sprachen, in denen Objekte bei der Parameterübergabe (teilweise) kopiert werden, besser geeignet als das Mengen-Modell. Vor allem lassen sich mit dieser Typinterpretation explizite und implizite Typumwandlungen besser beschreiben. Aber mit Typumwandlungen kann man nie die Mächtigkeit erzielen, die man mit Subtyping und dem Übergeben von Zeigern bei der Parameterübergabe hat.

4.5 Literaturhinweise

Die meisten polymorphen Typsysteme für funktionale Sprachen haben ihren Ursprung im Damas-Milner-Typsystem. Eine Einführung in dieses Typsystem und seine Anwendung in modernen funktionalen Sprachen wird in [14] gegeben. Eine detailiertere theoretische Darstellung und neuere Ergebnisse werden z.B. in [13] und [23] geboten. Es gibt zahlreiche Bücher über ML, Miranda, Haskell und ähnliche Sprachen. Die meisten dieser Bücher beschreiben Typen in funktionalen Sprachen aus einer eher praktischen Sichtweise.

Leider gibt es keine einfach lesbare und trotzdem umfassende Einführung in Subtyping für funktionale Sprachen. Eine umfangreiche theoretische Abhandlung über polymorphe Typen in funktionalen Sprachen ist [23]. Mehrere Verfeinerungen und Erweiterungen solcher Typen, einschließlich des PER-Modells, werden unter anderem in [12] formal behandelt. Zahlreiche Aspekte von Subtyping und Generizität vor allem in Zusammenhang mit objektorientierter Programmierung werden in der (leider nur teilweise gut lesbaren) theoretischen Abhandlung [1] angesprochen.

Kapitel 5

Typen in objektorientierten Sprachen

Objektorientierte Sprachen mit polymorphen Typsystemen gehören zu den wesentlichsten Errungenschaften der Informatik in den letzten Jahrzehnten. Ein großer Teil dieses Fortschritts wäre ohne entsprechende Entwicklungen auf dem Gebiet der Typsysteme unvorstellbar. In diesem Kapitel werden einige dieser Entwicklungen vorgestellt, wobei der Schwerpunkt auf Subtyping, Vererbung und ähnlichen Konzepten liegt.

5.1 Untertypen in Beispielen

In Kapitel 3 wurde das Typsystem von Ada 83 als Beispiel für ein Typsystem in einer imperativen Sprache vorgestellt. Mittlerweile hat sich Ada weiterentwickelt, und neuere Standards wurden verabschiedet. Ada 95 enthält objektorientierte Konstrukte, die sich auch in einem Typsystem, das Subtyping und Vererbung unterstützt, niederschlagen. Ada 95 wird in Abschnitt 5.1.1 vorgestellt, da sich an dieser Sprache die Entwicklung von Typsystemen nachvollziehen lässt. Ähnliche Entwicklungen fanden auch bei anderen Sprachen statt, zum Beispiel beim Übergang von C nach C++ bzw. Objective C, von Modula-2 nach Modula-3 bzw. Oberon-2 und so weiter. In Abschnitt 5.1.2 wird mit Eiffel eine Sprache vorgestellt, in der Module und Typen im Wesentlichen zu einem Sprachkonstrukt, nämlich Klassen, zusammengefasst sind. Ähnliche Konstrukte gibt es zum Beispiel auch in C++ und Java. Eiffel wurde gewählt, da die Sprache die Festlegung von *Zusicherungen* (engl. *assertions*) erlaubt, mit deren Hilfe sich grundlegende Probleme der Typspezifikation erklären lassen.

5.1.1 Ada 95

Vererbung und damit einhergehende Untertypen lassen sich in Ada 95 auf folgende Kurzformel bringen:

Vererbung	=	Typableitung
	+	Überschreiben
	+	Typerweiterung
	+	dynamische Typerkennung

Abgeleitete Typen wurden bereits in Abschnitt 3.1.3 vorgestellt. Dieser Mechanismus ist die Grundlage für Vererbung in Ada 95. Zur Wiederholung: Aus jedem bereits existierenden Typ kann ein neuer Typ abgeleitet werden, der dieselbe Struktur und dieselben darauf anwendbaren Routinen wie sein Elterntyp hat, jedoch einen eigenständigen Typ mit einem eigenen Namen darstellt. Eine Instanz eines abgeleiteten Typs kann durch explizite Typumwandlung in eine Instanz des Elterntyps umgewandelt werden und umgekehrt.

```
type Int is range 0..10000;
function "+"(Links, Rechts: Int)
           return Int;
type Laenge is new Int;
-- function "+"(Links, Rechts: Laenge)
--             return Laenge;
```

In diesem Programmstück ist `+` implizit auch für `Laenge` deklariert. Die entsprechende Deklaration ist daher auskommentiert (durch `--`). Auch die Implementierung der Funktion wird *ererbt.* Ada 95 erlaubt auch das *Überschreiben* von Funktionen:

```
type Winkel is new Int;
function "+"(Links, Rechts: Winkel)
           return Winkel;
```

Hier bewirkt die explizite Angabe der Typdeklaration, dass die ererbte Implementierung von `+` durch eine neue Implementierung ersetzt wird. Auf diese Weise können auch neue Routinen, die im Elterntyp nicht deklariert sind, dazugefügt werden. Es ist aber nicht möglich, ererbte Routinen gänzlich zu entfernen.

Zur Wiederholung: Kontravarianz bedeutet, dass die Parametertypen von Routinen durch Vererbung (bzw. Subtyping) gleich bleiben oder allgemeiner werden. Kovarianz bedeutet, dass Parametertypen durch Vererbung gleich bleiben oder spezieller werden. Und Invarianz bedeutet, dass Parametertypen durch Vererbung nicht verändert werden.

Die Schnittstellenbeschreibungen von Routinen auf einem abgeleiteten Typ unterscheiden sich von den

Schnittstellenbeschreibungen der entsprechenen Routinen auf dem ursprünglichen Typ genau dadurch, dass alle Vorkommen des ursprünglichen Typbezeichners durch den Bezeichner des abgeleiteten Typs ersetzt werden. Alle anderen Typbezeichner bleiben gleich. Abgeleitete Typen sind Untertypen des ursprünglichen Typs. Sowohl `Laenge` als auch `Winkel` sind Untertypen von `Int`. In Ada sind daher sowohl Eingangs- als auch Ausgangsparametertypen kovariant (oder invariant). Routinen auf abgeleiteten Typen widersprechen aus diesem Grund dem Ersetzbarkeitsprinzip. Kovariante Typen von Eingangsparametern decken nur einen Teil des Eingangswertebereiches ab. In Ada ist es daher auch nicht uneingeschränkt möglich, eine Instanz eines Untertyps überall zu verwenden, wo eine Instanz eines Obertyps erwartet wird. Statt dessen muss eine Instanz des erwarteten Typs (also keines Untertyps davon) übergeben werden.

Unter *Typerweiterung* versteht man in Ada das Hinzufügen neuer Komponenten zu abgeleiteten Verbundtypen. Solche erweiterbaren Verbunde müssen mit dem Wort `tagged` gekennzeichnet werden:

```
type Person is tagged record
    Vorname: String (1..20);
end record;

type Mann is new Person with record
    Barttraeger: Boolean;
end record;

type Frau is new Person with null record;
```

Verbundtypen, die nicht durch `tagged` gekennzeichnet sind, können nicht erweitert werden. Abgeleitete erweiterbare Verbundtypen müssen entweder eine Klausel `with record ... end record` oder, falls man ohne Zusätze auskommt, `with null record` enthalten. Typumwandlung ist für Instanzen erweiterbarer Verbundtypen nur in Richtung Vorgängertyp möglich, da Instanzen nicht genügend Information für Umwandlungen in die andere Richtung enthalten. Routinen auf erweiterbaren Verbundtypen können auf die genau gleiche Weise überladen werden wie die auf anderen Typen:

```
function Anrede (P: in Person)
                return String; -- ""
function Anrede (M: in Mann)
                return String; -- "Herr "
function Anrede (P: in Frau)
                return String; -- "Frau "
```

Erweiterbare Verbundtypen sind eine Voraussetzung für dynamische Typerkennung. Zu jedem erweiterbaren Verbundtyp t ist implizit ein sogenannter *klassenweiter Typ* `t'Class` definiert. Die durch t definierte Klasse (in Ada-Terminologie) umfasst t und alle von t abgeleiteten Typen. Zu `Person'Class` gehören demnach `Person`, `Mann` und `Frau`, und zu `Frau'Class` gehört nur `Frau`. Jede Instanz eines klassenweiten Typs enthält eine unsichtbare Komponente, ein *„tag“*, das den *spezifischen* (= nicht klassenweiten) Typ der Instanz angibt. Da diese Komponente während der Programmausführung gebraucht wird, muss jede Variable eines klassenweiten Typs mit einer Instanz eines entsprechenden spezifischen Typs initialisiert werden:

```
P: Person'Class := Frau'(Vorname => "Anna");
```

Der spezifische Typ von `P` kann zur Laufzeit nicht geändert werden, da sich dadurch der Platzbedarf für `P` ändern würde. Wenn man den spezifischen Typ ändern will, muss man auf Zeiger zurückgreifen:

```
type PersonRef is access Person'Class;
P_Ref: PersonRef;
P_Ref := new Frau'(Vorname => "Anna");
P_Ref := new Mann'(Vorname => "Stefan",
                   Barttraeger => False);
```

Wenn die spezifischen Typen von Argumenten dem Compiler bekannt sind, kann die auszuführende überladene Operation statisch festgelegt werden—statisches Binden. Bei Verwendung klassenweiter Typen als Argumenttypen kann die auszuführende Operation jedoch erst zur Laufzeit anhand der „tag“-Komponente bestimmt werden, falls der entsprechende formale Parametertyp ein spezifischer Typ ist—dynamisches Binden. Diese „tag“-Komponente ist auch direkt über den `in`-Operator abfragbar:

```
procedure Display (P: in Person'Class) is
begin
    Put (Anrede (P));
    Put (P.Vorname);
    if P in Mann'Class then
        Put (" (maennlich)")
    elsif P in Frau'Class then
        Put (" (weiblich)")
    end if;
end Display;
```

Für jeden spezifischen Typ in `Person'Class` gibt es eine Funktion `Anrede`. Es muss dynamisch entschieden werden, welche dieser Funktionen auszuführen ist. Jede Instanz eines derartigen Typs hat die Komponente `Vorname`.

Ein klassenweiter Typ gilt als vom entsprechenden spezifischen Typ verschieden. Daher bleiben bei einer Typableitung klassenweite Parametertypen stets unverändert. Alle klassenweiten Parametertypen sind daher invariant und entsprechen dem Ersetzbarkeitsprinzip.

Beim Aufruf einer Routine dürfen Eingangsargumente einen klassenweite Typ haben, obwohl die entsprechenden formalen Parameter einen spezifischen erweiterbaren Typ haben. Über die dynamischen Typen

der Argumente erfolgt in diesem Fall dynamisches Binden. In Ada geht (im Gegensatz zu C++, Java und Eiffel) aus der Syntax nicht hervor, über welches Argument einer Routine statisches bzw. dynamisches Binden erfolgt. Es gibt daher eine Einschränkung die sicherstellt, dass die Semantik trotzdem eindeutig definiert ist: Alle spezifischen erweiterbaren Parametertypen einer Routine müssen gleich sein. Beim dynamischen Binden müssen alle diese Argumente denselben dynamischen Typ haben. Dies wird zur Laufzeit kontrolliert bevor dynamisch gebunden wird. In der Funktion `Anrede` ist diese Bedingung z.B. einfach dadurch erfüllt, dass es nur einen einzigen Parameter gibt.

Ada 95 unterstützt im Zusammenhang mit Vererbung auch *abstrakte Typen*—nicht zu verwechseln mit abstrakten Datentypen. Abstrakte Typen definieren klassenweite Typen, ohne selbst als spezifische Typen auftreten zu können. Diese Typen werden durch das Schlüsselwort `abstract` gekennzeichnet:

```
type A is abstract tagged null record;
procedure Q (X: in A) is abstract;

type B is new A with record
    I: Integer;
end record;
procedure Q (X: in B) is ...
```

`A` ist ein abstrakter und `B` ein davon abgeleiteter konkreter Typ. `Q` ist auf `A` als abstrakte Prozedur deklariert. Jeder von `A` abgeleitete konkrete Typ muss `Q` überschreiben. Abstrakte Typen verwendet man zusammen mit klassenweiten Typen, wenn z.B. Prozeduren für `Mann` und `Frau`, nicht aber für `Person` benötigt werden.

Schließlich soll ein weiterer Mechanismus von Ada 95 vorgestellt werden, der eher mit Generizität als mit Vererbung und Subtyping zu tun hat. Das nächste Beispiel zeigt eine generische Funktion:

```
generic
  type T is private;
  with function "<"(X,Y:T) return Boolean;
function Max(X,Y:T) return T is
begin
  if X < Y
    then return Y;
    else return X;
  end if;
end Max;
```

Diese generische Funktion nimmt eine Funktion `<` als generischen Parameter. `<` muss auf dem anderen generischen Parameter definiert sein. Damit die generische Funktion verwendbar wird, muss man eine nichtgenerische Funktion daraus ableiten:

```
function MaxI is new Max(Integer,"<");
```

`MaxI` ist also eine Funktion auf ganzen Zahlen, die das Maximum der beiden Argumente zurückliefert. `<` ist die Vergleichsfunktion auf `Integer`. Es lässt sich aus `Max` aber auch leicht `MinF` ableiten:

```
function MinF is new Max(Float,">");
```

In diesem Fall wird für den generischen Parameter `<` die Funktion `>` auf Fließkommazahlen eingesetzt, sodass `MinF` die kleinere der beiden Zahlen zurückliefert.

In Ada sind die Mechanismen für Vererbung und Datenkapselung getrennt. Für Vererbung werden abgeleitete Typen und für Datenkapselung Pakete verwendet. Eine ähnliche Trennung zwischen Vererbung und Kapselung gibt es zum Beispiel in Oberon-2. In Sprachen wie Java, C++ und Eiffel sind diese Mechanismen jedoch zu einem Konzept kombiniert.

5.1.2 Eiffel

Die Deklarationseinheiten in Eiffel sind *Klassen*. Eine Klasse enspricht einem (parametrischen) Modul, spezifiziert gleichzeitig aber auch einen Typ oder eine Familie von Typen. Hier ist ein Beispiel einer Klasse in Eiffel:

```
class KONTO [T -> GELD_BETRAG]
feature {ANY}
    guthaben: T;
    ueberziehungsrahmen: T;
    besitzer: PERSON;

    einzahlen (summe: T) is
        do addieren (summe)
    end; -- einzahlen

    abheben (summe: T) is
        do addieren (-summe)
    end; -- abheben

feature {NONE}
    addieren (summe: T) is
        do guthaben := guthaben + summe
    end; -- addieren
end -- class KONTO
```

Das Programmstück beschreibt eine *generische Klasse* namens `KONTO`. Typparameter werden in eckigen Klammern geschrieben. `KONTO` hat einen Typparameter `T`, der universell über `GELD_BETRAG` und dessen Untertypen quantifiziert ist. Man nennt diese Art von Polymorphismus *gebundene Generizität* (oder *gebundenen parametrischen Polymorphismus*) da der Typparameter nicht frei gewählt werden kann, sondern ein Untertyp eines gegebenen Typs sein muss. Es handelt sich um Generizität, da Typen für Typparameter einzusetzen sind. Geeignete Werte für `T` sind etwa `SCHILLING_BETRAG` und `DOLLAR_BETRAG`. Es wird angenommen, dass `GELD_BETRAG` die binären Operationen

`+` und `-` als Infixoperatoren sowie die unäre Operation `-` als Präfixoperator definieren. Auch alle Untertypen von `GELD_BETRAG` stellen diese Operationen zur Verfügung. Gebundener parametrischer Polymorphismus wird also dann verwendet, wenn von einem Typparameter erwartet wird, dass Instanzen des dafür eingesetzten Typs bestimmte Operationen und Verbund-Komponenten bereitstellen. In Ada wird dafür, wie wir gesehen haben, ein anderer Mechanismus eingesetzt.

Der Rest der Klassendefinition sollte, abgesehen von der `feature`-Klausel, selbsterklärend sein. Dem Schlüsselwort `feature` kann eine Liste von Klassennamen in geschwungenen Klammern folgen. Alle nachfolgend definierten Variablen und Routinen sind nur innerhalb der Klassen in dieser Liste und deren Unterklassen sichtbar. `ANY` ist die allgemeinste Klasse, von der alle anderen Klassen erben. Daher sind im Beispiel die Variablen `guthaben`, `ueberziehungsrahmen` und `besitzer` sowie die Funktionen `einzahlen` und `abheben` überall sichtbar. `NONE` ist dagegen eine vordefinierte Klasse ohne Instanzen, von der keine Klasse erbt. Daher gibt es im Beispiel keine Objekte außer den Instanzen von `KONTO`, die `addieren` sehen. Genaugenommen sind in anderen Klassen, die keine Unterklassen sind, höchstens die Schnittstellen von Routinen, aber nicht deren Implementierungen sichtbar. Obwohl Schnittstelle und Implementierung textuell nicht getrennt sind, werden sie konzeptuell sehr wohl unterschieden.

An der Beschreibung des Programmstücks im vorigen Absatz fällt auf, dass es schwierig ist, die Begriffe „Klasse“ und „Typ“ klar voneinander zu trennen. Zum Beispiel wurde „Unterklasse“ in äquivalenter Bedeutung zu „Untertyp“ verwendet. Tatsächlich gibt es in Eiffel-Terminologie eine eins-zu-eins-Beziehung zwischen parameterfreien Klassen und Typen. Eine generische Klasse beschreibt entsprechende nicht-generische Typen. Zum Beispiel sind `KONTO[SCHILLING_BETRAG]` und `KONTO[DOLLAR_BETRAG]` zwei Typen, die durch die Klasse `KONTO` beschrieben werden.

Alle Objekte in Eiffel sind Instanzen eines Typs. Jede Variable enthält einen Zeiger auf ein Objekt und hat einen deklarierten Typ. Das Objekt, auf das gezeigt wird, muss auf jeden Fall einen (dynamischen) Typ haben, der ein Untertyp des deklarierten Typs der Variable ist. Beim Aufruf einer Routine, die zu einem Objekt gehört, wird das Objekt explizit durch Punktnotation angegeben:

```
konto:  KONTO[SCHILLING_BETRAG];
betrag: SCHILLING_BETRAG;
...
konto.einzahlen (betrag);
```

Dynamisches Binden erfolgt nur über dieses Objekt, in diesem Fall dem Inhalt der Variable `konto`. Dieses Objekt kann als spezielles, implizites Argument der Routine angesehen werden.

Eine Besonderheit von Eiffel sind *Zusicherungen.* Sie werden als *Vorbedingungen* (engl. „preconditions, als `require`-Klauseln in Eiffel) und *Nachbedingungen* (engl. „postconditions“, `ensure`-Klauseln) zu Implementierungen von Routinen (`do`-Klauseln) dazugefügt. Daneben gibt es noch Invarianten. Jede Zusicherung ist eine Liste Boolescher Ausdrücke, die durch Strichpunkte getrennt sind. Der Strichpunkt steht implizit für eine Konjunktion (Und-Verknüpfung). Daher kann jede Zusicherung entweder zu „ja“ oder „nein“ ausgewertet werden. Wenn eine Zusicherung zu „nein“ ausgewertet wird, erfolgt eine Fehlermeldung bzw. Ausnahmebehandlung. Eine Vorbedingung beschreibt die Menge der erlaubten Eingangswerte genauer als dies durch Parametertypen alleine möglich wäre. Ebenso beschreiben Nachbedingungen mögliche Mengen von Ergebniswerten. Invarianten müssen stets gelten wenn der Zustand eines Objekts von außen sichtbar ist. In Nachbedingungen ist die Bezugnahme auf Variablen- und Argumentwerte zum Zeitpunkt des Funktionsaufrufs erlaubt. Zum Beispiel bezeichnet `old guthaben` den Wert der Variable `guthaben` zum Zeitpunkt des Aufrufs von `einzahlen` oder `abheben`. Zusicherungen auf diesen beiden Routinen können beispielsweise so aussehen:

```
einzahlen (summe: T) is
    require summe >= 0
    do addieren (summe)
    ensure guthaben = old guthaben + summe
    end; -- einzahlen

abheben (summe: T) is
    require summe >= 0;
            guthaben >= summe -
                        ueberziehungsrahmen
    do addieren (-summe)
    ensure guthaben = old guthaben - summe
    end; -- abheben
```

Zusicherungen können als Teil des Typsystems angesehen werden. Konzeptuell gehört eine Zusicherung zur Schnittstelle und nicht zur Implementierung. Daher ist sie dem Aufrufer einer Routine bekannt. Zusicherungen müssen in der Regel zur Laufzeit überprüft werden. Vorbedingungen werden vor dem Aufruf einer Routine überprüft, Nachbedingungen am Ende der Ausführung einer Routine und Invarianten sowohl vor als auch nach der Ausführung. In einigen Fällen können Zusicherungen zur Optimierung eingesetzt werden.

Eiffel unterstützt Vererbung von Klassen und damit auch von Typen:

```
class SCHILLING_BETRAG
inherit GELD_BETRAG
feature ...
```

Dies entspricht der Ableitung des Typs (bzw. der Klasse) `SCHILLING_BETRAG` aus `GELD_BETRAG`. Ebenso wie in Ada gibt es die Möglichkeit, Routinen zu überschreiben, Klassen—entsprechend den erweiterbaren Verbunden—zu erweitern und abstrakte Typen zu definieren. Außerdem entsprechen in Eiffel alle Variablen ohne zusätzliche Deklarationen Zeigern auf klassenweite Typen in Ada. (Beachten Sie, dass sich die Bedeutungen des Begriffs „Klasse“ in Eiffel und Ada wesentlich voneinander unterscheiden.) Im Gegensatz zu Ada bietet Eiffel Mehrfachvererbung auf der Sprachebene, das heißt, eine Klasse kann von mehr als nur einer anderen Klasse erben. Um Namenskonflikte zu vermeiden, bietet Eiffel umfangreiche Möglichkeiten, ererbte Variablen und Routinen umzubenennen.

Die Frage, ob Eiffel typsicher ist, lässt sich nicht so einfach beantworten. Betrachten wir die Vererbung genauer: Die formalen Parameter der Routinen in einer Unterklasse haben, falls die Routinen nicht überschrieben sind, genau dieselben Typen wie die entsprechenden Parameter im dazugehörigen Obertyp. Das heißt, im Gegensatz zu Ada werden Typen nicht automatisch verändert. Sie sind in diesem Fall invariant. Eiffel unterstützt aber auch Kovarianz auf allen Parametern: Beim Überschreiben ererbter Routinen können die Typen formaler Parameter und Ergebnistypen durch beliebige Untertypen der ursprünglichen Typen ersetzt werden. Dadurch, dass alle Parameter kovariant sein können, kann der Compiler nicht mehr garantieren, dass der gesamte Eingangswertebereich abgedeckt ist; das Ersetzbarkeitsprinzip ist verletzt. In Eiffel wird beim Auftreten nicht abgedeckter Eingangswerte eine Ausnahmebehandlung durchgeführt. Es ist aber auch möglich, die erlaubten Aufrufe so einzuschränken, dass der Compiler (bzw. Linker) Typkonsistenz garantieren kann. Dynamisches Binden ist dann nur in den Fällen erlaubt, in denen alle Parametertypen invariant sind.

Nachbedingungen überschriebener Funktionen sind in Eiffel strenger oder gleich streng wie die der ursprünglichen Funktionen. Dadurch beschreiben überschriebene Nachbedingungen kleinere Wertemengen. Bei den Vorbedingungen ist es umgekehrt: Überschriebene Vorbedingungen dürfen nicht strenger, können aber allgemeiner sein und beschreiben größere Wertemengen. Vererbung in Eiffel ist hinsichtlich der Zusicherungen auf den Eingangsparametern kontravariant. Invarianten sind im Wesentlichen invariant. Daher entsprechen Zusicherungen dem Ersetzbarkeitsprinzip.

5.2 Allgemeine Konzepte

Nach Beispielen für Sprachen mit einem objektorientierten Typkonzept sollen nun einige wichtige Konzepte unabhängig von konkreten Sprachen behandelt werden.

5.2.1 Varianz von Parametertypen

Die Begriffe Kovarianz, Kontravarianz und Invarianz wurden bereits des öfteren verwendet. Da sie bei Typen in objektorientierten Sprachen eine große Rolle spielen, wollen wir sie hier noch einmal untersuchen. Zur Wiederholung: Wir schreiben $s \leq t$ wenn s ein Untertyp von t ist. Für Typen von Routinen schreiben wir

$$s_1 \times \cdots \times s_k \to s_{k+1} \times \cdots \times s_n \;\leq\; t_1 \times \cdots \times t_k \to t_{k+1} \times \cdots \times t_n.$$

Die Typen von Eingangsparametern stehen links vom Pfeil, solche von Ausgangsparametern rechts vom Pfeil. Durchgangsparameter werden dupliziert und deren Typen stehen sowohl links als auch rechts vom Pfeil. Der i-te Parameter (für $1 \leq i \leq n$) in diesem Typ ist *kovariant* wenn $s_i \leq t_i$, *kontravariant* wenn $t_i \leq s_i$ und *invariant* wenn $s_i = t_i$. Da die Ausgangsparametertypen praktisch immer kovariant (oder invariant) sind, sagt man oft, eine Routine (bzw. Subtyping oder Vererbung) sei kontra- bzw. kovariant wenn die Eingangsparametertypen kontra- bzw. kovariant sind. Von invariantem Subtyping spricht man wenn alle Parametertypen invariant sind.

Eine Programmiersprache sollte sicherstellen, dass nur solche Varianzen erlaubt sind, die mit dem Ersetzbarkeitsprinzip kompatibel sind:

> Eine Instanz eines Untertyps kann überall verwendet werden wo eine Instanz eines entsprechenden Obertyps erwartet wird.

Insbesondere sollte keine Änderung des Aufrufs einer zu einem Objekt gehörenden Routine nötig sein wenn dieses Objekt durch ein anderes Objekt eines Untertyps ersetzt wird.

Wie in Abschnitt 4.1.2 erklärt, muss der Aufruf einer Routine vom Typ

$$t_1 \times \cdots \times t_k \to t_{k+1} \times \cdots \times t_n$$

Variablen der statischen Typen $u_1, \ldots, u_n$ als Argumente übergeben, wobei $u_i \leq t_i$ für $1 \leq i \leq k$ und $t_j \leq u_j$ für $k < j \leq n$. Wenn t_i und t_j gemeinsam den Typ eines Durchgangsparameters beschreiben gilt zwangsläufig $u_i = t_i = t_j = u_j$. Diese Beziehungen zwischen Variablen- und Parametertypen müssen auch dann gelten wenn die Routine durch eine andere Routine eines beliebigen Untertyps

$$s_1 \times \cdots \times s_k \to s_{k+1} \times \cdots \times s_n$$

ersetzt wird. Für alle Typen $u_1, \ldots, u_n$ und $s_1, \ldots, s_n$ muss Folgendes gelten: $u_i \leq s_i$ für $1 \leq i \leq k$ und $s_j \leq u_j$ für $k < j \leq n$. Da wir bereits wissen, dass $u_i \leq t_i$ für $1 \leq i \leq k$ und $t_j \leq u_j$ für $k < j \leq n$, sind diese Beziehungen erfüllt wenn $t_i \leq s_i$ für $1 \leq$

$i \leq k$ und $s_j \leq t_j$ für $k < j \leq n$. Eingangsparametertypen müssen demnach kontravariant und Ausgangsparametertypen kovariant sein. Durchgangsparametertypen sind zwangsläufig invariant.

Eine andere Sichtweise von Subtyping beruht auf Familien von Einzelroutinen, die zusammengesetzt eine Gesamtroutine ergeben. Eine Gesamtroutine vom Typ

$$t = t_{\text{in}} \rightarrow t_{\text{out}} = t_1 \times \cdots \times t_k \rightarrow t_{k+1} \times \cdots \times t_n$$

kann als eine Familie von Einzelroutinen der Typen

$$t^i = t^i_{\text{in}} \rightarrow t^i_{\text{out}} = t^i_1 \times \cdots \times t^i_k \rightarrow t^i_{k+1} \times \cdots \times t^i_n$$

(für $1 \leq i \leq m$) aufgefasst werden, wobei $t^i_j \leq t_j$ für $1 \leq j \leq n$. Wenn $W(s)$ die Wertemenge (= Menge aller Instanzen) jedes Typs s bezeichnet, muss gelten:

$$W(t_{\text{in}}) \;=\; \bigcup_{1 \leq i \leq m} W(t^i_{\text{in}}).$$

Dadurch deckt die Vereinigung aller m Einzelroutinen den ganzen Eingangswertebereich der Gesamtroutine ab. Aufgrund der dynamischen Typen der Eingangsargumente kann zur Laufzeit entschieden werden, welche der Einzelroutinen ausgeführt zu werden hat. Falls es mehrere Möglichkeiten für die Auswahl der Einzelroutine gibt, werden weitere Kriterien zur Entscheidung herangezogen. In der Regel wird die passende Routine mit den spezifischsten Typen gewählt. Wenn die Auswahl der Einzelroutinen nur über ein einziges Eingangsargument erfolgt, spricht man von *Einfach-Binden* (engl. „single dispatching“). Wenn die Auswahl über das erste Argument erfolgt, gilt z.B. $t_j = t^i_j$ für alle $1 \leq i \leq m$ und $1 < j \leq k$. Einfach-Binden liegt (wie in Ada) auch vor wenn in jeder Einzelroutine mehrere gekennzeichnete Eingangsparameter einen gemeinsamen Typ haben und die entsprechenden Parameter der Gesamtroutine einen entsprechenden gemeinsamen Obertyp haben. Es muss dynamisch überprüft werden ob tatsächlich alle Argumente denselben dynamischen Typ haben. Die Auswahl erfolgt dann über den gemeinsamen dynamischen Typ dieser Eingangsargumente. Im allgemeinen Fall, wenn die Auswahl der Einzelroutine über die dynamischen Typen mehrerer beliebiger Argumente erfolgen kann, spricht man von *Mehrfach-Binden* (engl. „multiple dispatching“).

Den Aufruf einer zu einem Objekt gehörenden Routine kann man auch so auffassen, dass eine um einen Parameter erweiterte globale Routine aufgerufen wird, wobei dieses Objekt als zusätzliches Argument, das die Umgebung (engl. „environment“) beschreibt, übergeben wird. Der zusätzliche Parameter ist in der Regel ein Durchgangsparameter. Man kann die globale Routine als Gesamtroutine verstehen. Durch Einfach-Binden über den zusätzlichen Parameter wird jene Einzelroutine bestimmt, die dem Typ der Umgebung entspricht. Damit kann ein alternatives Regelsystem für ein dem Ersetzungsprinzip entsprechendes Subtyping aufgestellt werden. Der Typ der Gesamtroutine sei

$$\top \times t_1 \times \cdots \times t_k \rightarrow t_{k+1} \times \cdots \times t_n \times \top.$$

Der erste Eingangs- und letzte Ausgangsparameter stellen den durchgereichten zusätzlichen Parameter dar. Für jeden Typ s^i ($1 \leq i \leq m$), der eine entsprechende Routine unterstützt, gibt es eine Einzelroutine vom Typ

$$s^i \times t^i_1 \times \cdots \times t^i_k \rightarrow t^i_{k+1} \times \cdots \times t^i_n \times s^i.$$

In der einfachsten Form gilt $t^i_j = t_j$ für alle $1 \leq i \leq m$ und $1 \leq i \leq k$. Diese Form von Subtyping mit invarianten Eingangsparametertypen—oft auch mit invarianten Ausgangsparametertypen—wird in vielen Sprachen wie z.B. C++, Java und Oberon-2 verwendet. Die entsprechenden Typüberprüfungen können in jedem Fall vom Compiler durchgeführt werden. Allerdings ist diese Form von Subtyping stärker eingeschränkt als notwendig. Die zusätzlichen Möglichkeiten von kontravariantem Subtyping werden aber kaum benötigt.

Die in Ada verwendete Form ist eine Erweiterung davon: $t^i_j = s^i$ falls s^i ein von t_j abgeleiteter Typ ist, sonst $t^i_j = t_j$. Auch für diese Form ist nur Einfach-Binden notwendig, aber im Allgemeinen muss man dabei Typen dynamisch prüfen: Bei einem Aufruf muss garantiert werden, dass alle Argumente an Parameterpositionen des Typs s^i einen *dynamischen* Typ s^i haben. Die dynamischen Typen der Argumente bzw. Variablen sind in der Regel nicht zur Compilezeit bekannt.

Recht kompliziert sind die Verhältnisse bei der in Eiffel verwendeten Form des kovarianten Subtyping, wobei das Binden nur über den Typ *eines* Objekts erfolgt. Die einzige Regel ist $t^i_j \leq t_j$ für $1 \leq j \leq n$. Die Bedingung $W(t_{\text{in}}) = \bigcup_{1 \leq i \leq m} W(t^i_{\text{in}})$ und das Ersetzbarkeitsprinzip sind im Allgemeinen nicht erfüllt. Bei einer Typüberprüfung zur Compilezeit muss eine statische Analyse des gesamten Programms durchgeführt werden um sicherzustellen, dass alle möglicherweise aufgerufenen Routinen tatsächlich vorhanden sind. Der damit verbundene Aufwand wird häufig geschäut. Außerdem ist es für einen Programmierer sehr oft nicht leicht zu verstehen, wodurch ein möglicher Typfehler verursacht wird; eine kleine Änderung an einer Stelle eines Programms kann einen Typkonflikt an einer ganz anderen, mit der Änderung scheinbar in keinerlei Zusammenhang stehenden Stelle verursachen.

Kovariantes Subtyping lässt sich gut mittels „multiple dispatching“ realisieren. Ein Problem dabei ist jedoch, dass die Anzahl der Einzelroutinen rasch explodieren (übermäßig groß werden) kann. Diese Code-Explosion kann man nur dadurch umgehen, dass man bei der Vererbung Heuristiken verwendet, die aber

nicht in jedem Fall den Erwartungen entsprechen. Außerdem wird durch „multiple dispatching“ die Zuordnung von Einzelroutinen zu Klassen bzw. Typen erschwert; die auszuführende Einzelroutine wird ja nicht mehr über ein ausgewähltes s_i, sondern über eine Menge gleichwertiger dynamischer Typen bestimmt.

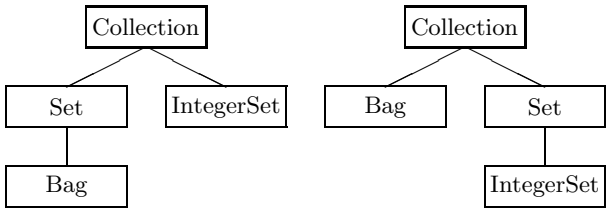
Für einen Softwareentwickler bietet kovariantes Subtyping oft eine gute Lösung, da sich damit einige in der Praxis häufige Beziehungen zwischen Typen relativ einfach ausdrücken lassen. Zum Beispiel gibt es einen Typ `Tier` mit einer Routine `füttere`, die als Eingangsargument eine Instanz von `Nahrung` erwartet. Im Untertyp `Löwe` von `Tier` soll `füttere` eine Instanz von `Fleisch` erwarten, wobei `Fleisch` ein Untertyp von `Nahrung` ist. Diese Beziehung lässt sich nur mittels Kovarianz ausdrücken. Wie wir gesehen haben, kann Kovarianz in uneingeschränkter Form von einer Programmiersprache nicht zur Verfügung gestellt werden, wenn statische Typüberprüfung und getrennte Übersetzbarkeit von Modulen gefordert wird und die Nachteile von „multiple dispatching“ nicht in Kauf genommen werden sollen. Die Kunst des Programmiersprachendesigns liegt auch hierbei zu einem großen Teil darin, gute Kompromisse zu finden.

Ein möglicher Ausweg besteht vielleicht darin, Typen in eingeschränkter Weise als Werte zu betrachten. (Dabei ist Vorsicht geboten, da dies leicht zu unentscheidbaren Typsystemem führen kann.) Der Typ `Tier` kann z.B. eine Konstante `frisst` enthalten, deren Wert der Typ `Nahrung` ist. Die Routine `füttere` erwartet sich dann eine Instanz von `frisst` als Argument. In `Löwe` kann `frisst` den Wert `Fleisch` haben; `füttere` bleibt unverändert. Dieser Ansatz löst das Problem nur scheinbar: `füttere` in `Löwe` akzeptiert noch immer nicht alle Werte, die von `füttere` in `Tier` als Argument erwartet werden. Dennoch kann dieser Ansatz unter Umständen Vorteile bringen, da die Beziehung zwischen `Tier` und `Nahrung` über `frisst` von vornherein festgelegt ist. Damit kann eine notwendige dynamische Typüberprüfung bereits früher, vor dem Aufruf erfolgen. In einigen Fällen, aber nicht immer, kann überhaupt darauf verzichtet werden.

5.2.2 Untertypen und Vererbung

In vielen objektorientierten Sprachen, z.B. in Ada, Java, C++ und Eiffel, werden Subtyping und Vererbung als zusammengehörige Mechanismen verstanden. Bei genauerer Betrachtung stellt sich aber heraus, dass sich dadurch Probleme ergeben können. In objektorientierten Sprachen entspicht ein Typ in etwa der *Spezifikation der Schnittstelle und des nach außen sichtbaren Verhaltens eines Objekts.* Alle Instanzen eines Typs bieten deshalb nach außen hin dieselbe Schnittstelle an und zeigen dasselbe Verhalten. Ein Untertyp bietet mehr Möglichkeiten in der Schnittstelle und beschreibt das Verhalten genauer als ein entsprechender Obertyp. Der Begriff „Vererbung“ bezieht sich dagegen auf Implementierungen. Zum Beispiel erbt ein abgeleiteter Typ die Implementierungen seiner Routinen, falls sie nicht überschrieben werden. Man könnte meinen, dass die Implementierung das Verhalten formal beschreibe und Vererbung daher eine direkte Folge der Einführung von Untertypen darstelle. Wenn dem so wäre, müsste man das Überschreiben von Routinen verbieten, da Überschreiben mit einer Verhaltensänderung gleichzusetzen wäre. Überschreiben ist aber für die praktische Programmierung wichtig, sodass man es nicht einfach verbieten kann. Implementierung und Festlegung des nach außen sichtbaren Verhaltens sind daher zwei verschiedene Dinge. Während sich die Beschreibung des Verhaltens bei Typen nur auf die nach außen hin sichtbaren und relevanten Aspekte beschränkt, beschreibt die Implementierung auch alle anderen Aspekte im Detail. Untertypen beziehen sich daher auf eine konzeptuelle und Implementierungen auf die konkrete Ebene. Man spricht daher auch häufig von *Vererbung von Konzepten* (gleichbedeutend mit Subtyping) im Gegensatz zur *Vererbung von Implementierungen.*

Den wesentlichen Unterschied zwischen Vererbung von Konzept und Implementierung versteht man am besten anhand eines Beispiels, in dem sich die beiden Hierarchien unterscheiden:



Vererbung von Impl. Subtyping

Ziel der Vererbung von Implementierungen ist die direkte Wiederverwendung so vieler Programmteile wie möglich. Die Implementierungen von `Set` und `Bag` zeigen so starke Ähnlichkeiten, dass sich die Wiederverwendung von Programmteilen lohnt. In diesem Fall erbt `Bag` große Teile der Implementierung von `Set`. Für diese Entscheidung ist nur der pragmatische Gesichtspunkt, dass sich `Bag` einfacher aus `Set` ableiten lässt als umgekehrt, ausschlaggebend. Für `IntegerSet` wurde eine von `Set` unabhängige Implementierung gewählt, da durch die Beschränkung auf ganze Zahlen eine wesentlich effizientere Implementierung möglich ist.

Wenn wir uns von Konzepten (bzw. Typen) leiten lassen, schaut die Hierarchie anders aus. `Bag` und `Set` stehen in keinem Verhältnis zueinander, da die Routinen für das Hinzufügen bzw. Löschen von Elementen einander ausschließende Bedeutungen haben, obwohl `Set` und `Bag` dieselbe Schnittstelle haben können.

Die Operationen für `Set` und `IntegerSet` haben dagegen dieselbe Bedeutung. `IntegerSet` kann, aufgrund der Einschränkung auf ganze Zahlen, eine gegenüber `Set` erweiterte Schnittstelle anbieten und ist daher ein Untertyp von `Set`. Beispielsweise kann man für `IntegerSet` eine Routine definieren, die die größte Zahl in der Menge findet. Eine vergleichbare Operation kann es in `Set` nicht geben, da auf den Elementen dieser Menge keine Ordnung definiert ist.

Obiges Beispiel demonstriert unterschiedliche Argumentationen für die Vererbung von Implementierungen und Konzepten. Die Unterschiede zwischen den Argumentationen sind wichtiger als jene zwischen den Hierarchien, da die Hierarchien selbst letztendlich von Details der Sprache und beabsichtigten Verwendungen abhängen. Beim Implementierungsbeispiel wurde davon ausgegangen, dass sich `Bag` relativ einfach durch Verwendung von `Set` implementieren lässt, was nicht für jede Sprache zutreffen muss. Bei der Vererbung von Konzepten wurde implizit ein Typsystem vorausgesetzt, das eine entsprechende Spezialisierung zwischen `Set` und `IntegerSet` erlaubt.

Es wurden experimentelle Sprachen entwickelt, in denen Subtyping von einem Mechanismus für die Vererbung von Implementierungen getrennt ist. Während in der Typhierarchie das Ersetzbarkeitsprinzip beachtet werden muss, braucht dieses in der Vererbungshierarchie nicht zu gelten. Damit ist es einfach, Vererbung sehr allgemein zu definieren. Zum Beispiel kann man auch erlauben, dass bei der Vererbung nur Teile einer Klasse in die neue Klasse übernommen werden. Das folgende Beispiel ist in der experimentellen Sprache *Portlandish* geschrieben:

```
type Person signature
    Name(): String,
    Versnr(): Integer
endType
```

Diese Typdefinition legt nur eine Schnittstelle fest. Die entsprechende Implementierung wird getrennt davon spezifiziert:

```
implementation PersonImpl of Person
    fields
        N: String;
        V: Integer;
    methods
        Name(): String { return N }
        Versnr(): Integer { return V }
endImplementation
```

Durch diese Trennung, vor allem durch getrennte Namen für Typen und Implementierungen, ist es möglich, alternative Implementierungen zu definieren:

```
implementation AnderePersonImpl of Person
    fields ...
    methods ...
endImplementation
```

Vererbung erfolgt getrennt für Typdefinitionen und Implementierungen:

```
type Student
    supertypes Person
    signature ...
endType

implementation StudentImpl of Student
    superImplementations PersonImpl
    fields ...
    methods ...
endImplementation
```

In diesem Beispiel stimmen die Hierarchien von Typen und Implementierungen überein. Da dieser Fall häufig auftritt, stellt sich die Frage, ob sich der zusätzliche Aufwand lohnt. Eine andere Möglichkeit ist, dass man nur Subtyping verwendet. Zur Vermeidung des Duplizierens von Programmcode können Sprachkonstrukte genutzt werden, die, wie z.B. Prozeduraufrufe, in der Sprache ohnehin vorhanden sind. Vor allem Sprachen, die Module und Typen zu Klassen vereinen, machen dafür aber gezielte Lockerungen bei den Sichtbarkeitsregeln notwendig.

Obiges Beispiel könnte man in einer sehr ähnlichen Form auch in Java oder C# schreiben. Diese Sprachen erlauben die Definition von Schnittstellen unabhängig von Klassen. Allerdings müssen auch Klassen in Java und C#, soweit dies überprüft werden kann, dem Ersetzbarkeitsprinzip entsprechen. Damit ist es in diesen Sprachen kaum möglich, Typ- und Vererbungshierarchien klar voneinander zu trennen.

Neben Subtyping und Vererbung von Implementierungen gibt es noch eine dritte Hierarchie, eine „is-a“-Hierarchie, die üblicherweise in der Analyse- bzw. Designphase eines Softwareprojekts erstellt wird. Dabei handelt es sich um eine konzeptuelle Hierarchie. Im Gegensatz zu Subtyping sind in der „is-a“-Hierarchie die Schnittstellen und das Verhalten nur ansatzweise festgelegt, und auf Einschränkungen wie Ko- und Kontravarianz wird verzichtet. Diese Hierarchie stellt oft einen Vorläufer der Subtyping-Hierarchie dar.

Typen werden in objektorientierten Sprachen durch ihre Schnittstellen und sichtbaren Verhaltensweisen festgelegt. (Typen in funktionalen Sprachen beschreiben nur Schnittstellen.) Es stellt sich die Frage, wie man das Verhalten eines Typs beschreiben kann wenn Implementierungen dafür nicht geeignet sind. Stand der Technik ist, das Verhalten eines Typs in der dazugehörigen Dokumentation zu beschreiben. In manchen Fällen werden dazu auch formale Spezifikationssprachen verwendet. Ein Ziel sollte sein, die formale

Spezifikation eines Typs dem Compiler zugänglich zu machen, damit die Übereinstimmung zwischen Spezifikation und Implementierung überprüft werden kann. Ein Ansatz dafür sind Zusicherungen wie in Eiffel. Allerdings erlauben diverse Einschränkungen nur die Darstellung eines Teils der gesamten Spezifikation eines komplexeren Typs. Die Darstellung komplexerer Spezifikationen ist noch immer Gegenstand diverser Forschungsaktivitäten.

Untertypen müssen dasselbe Verhalten wie Obertypen beschreiben, können dabei aber mehr Details berücksichtigen. Da Verhaltensbeschreibungen zur Zeit einem Compiler kaum zugänglich sind, muss irgendwie anders festgelegt werden, welche Typen aufgrund ihres Verhaltens kompatibel sind und welche nicht. Am häufigsten geschieht dies durch die explizite Angabe der Obertypen in jedem Typ. Z. B. folgen Ada, Eiffel, C++ und Java diesem Ansatz, der aber nur funktioniert wenn Typkompatibilität durch Namensgleichheit festgelegt ist. Wenn Typkompatibilität durch Strukturgleichheit bestimmt ist, kann ein Ansatz wie in Modula-3 verwendet werden: Neben global sichtbaren Variablen und Signaturen von Routinen spezifiziert ein Typ auch eine Menge zusätzlicher Namen, die das Verhalten in einer sehr abstrakten Form beschreiben. Ein Typ B ist ein Untertyp eines Typs A wenn es für jede Variable, jede Signatur und jeden verhaltensbeschreibenden Namen in A eine entsprechende Variable oder Signatur bzw. den gleichen Namen in B gibt. In jedem Fall kann das Verhalten durch einen beliebig gewählten Namen abstrahiert werden.

5.2.3 Subtyping und Verhalten

Liskov und Wing [16] haben eine vielfach zitierte Definition von Subtyping erarbeitet. Diese Definition beruht darauf, dass Typen formale Spezifikationen des sichtbaren Verhaltens aller Instanzen sind. Eine solche Typspezifikation sieht zum Beispiel so aus:

bag = **type**
  uses BBag (bag **for** B)
  for all b: bag
    put = **proc** (i: int)
      requires $|b_{pre}.elems| < b_{pre}.bound$
      modifies b
      ensures $b_{post}.elems = b_{pre}.elems \cup \{i\}$
        $\wedge\ b_{post}.bound = b_{pre}.bound$
    get = **proc** () **returns** (int)
      requires $b_{pre}.elems \neq \{\}$
      modifies b
      ensures $b_{post}.elems = b_{pre}.elems - \{result\}$
        $\wedge\ result \in b_{pre}.elems$
        $\wedge\ b_{post}.bound = b_{pre}.bound$

Die **uses**-Klausel gibt eine Sorte B—im algebraischen Sinn—mit dessen Spezifikation BBag an. Damit werden grundlegende mathematische Symbole wie $|x|$ (Anzahl der Elemente in der Menge x), $\cup$, $=$, $\neq$, $\leq$, etc. eingeführt, die in der Typspezifikation gebraucht werden. Die mit **for all** beginnende Klausel besagt, dass die folgende Spezifikation für alle Instanzen b des Typs bag gilt. Es werden zwei Routinen spezifiziert. Für jede Routine gibt es eine Vorbedingung (**requires**-Klausel) und Nachbedingungen (als **modifies**-Klausel, die mögliche Änderungen einschränkt, und als **ensures**-Klausel). Die Instanz b tritt in den Zusicherungen in zwei Versionen auf: b_{pre} steht für den Wert von b vor Ausführung der Routine, b_{post} für den Wert nach der Ausführung. Die Semantik dieser Spezifikation sollte intuitiv klar sein.

Eine (informale) Definition von Subtyping für solche Typspezifikationen lautet wie folgt:

s ist ein Untertyp von t wenn

1. jede Invariante von s eine entsprechene Invariante von t impliziert; Invarianten in s sind also mindestens so stark wie die in t;
2. die Routinen von s sich wie die Routinen von t verhalten:
   (a) Eingangsparametertypen sind kontravariant;
   (b) Ergebnistypen sind kovariant;
   (c) die Menge der von s erlaubten Ausnahmebehandlungen liegt in der Menge der von t erlaubten Ausnahmebehandlungen;
   (d) Vorbedingungen von t implizieren die entsprechenden Vorbedingungen von s (Kontravarianz); Vorbedingungen in s können also nur schwächer sein als die in t;
   (e) Nachbedingungen von s implizieren die entsprechenden Nachbedingungen von t (Kovarianz); Nachbedingungen in s sind also mindestens so stark wie die in t;
3. die *Einschränkungen* auf s entsprechende Einschränkungen auf t implizieren; Einschränkungen auf s sind also mindestens so stark wie die auf t.

Diese Definition verwendet *Einschränkungen* (engl. „constraints“ bzw. „history constraints“), auf die wir bis jetzt noch nicht eingegangen sind. Solche Einschränkungen verhindern Probleme, die durch *„aliasing“* entstehen können: Angenommen, ein Objekt wird über mehrere Variablen referenziert. In einer Sprache mit einem polymorphen Typsystem können diese Variablen verschiedene Typen haben. Jeder dieser Typen kann eine unterschiedliche Menge von Routinen anbieten. Jedoch werden Veränderungen auf dem Objekt, die

durch die Sichtweise der einen Variable erfolgen, auch über die andere Variable sichtbar. In einigen Fällen können diese verschiedenen Sichtweisen problematisch sein. Zum Beispiel, wenn t der Typ einer Menge ist, die nur Routinen zum Einfügen von Elementen in die Menge und zum Lesen der Elemente in der Menge anbietet, kann fälschlich angenommen werden, dass ein eingefügtes Element immer in der Menge bleibt. Man kann einen Untertyp s von t definieren, der eine zusätzliche Routine zum Löschen anbietet. Nun kann eine Instanz von s aber auch über eine Variable vom Typ t referenziert werden. Die über den Typ t gemachte Annahme, dass alle Elemente stets in der Menge bleiben, ist also nicht notwendigerweise erfüllt.

Durch die explizite Spezifikation von Einschränkungen sind solche Fälle ausgeschlossen. Einschränkung beschreiben, welche Eigenschaften von Instanzen des Typs erwartet werden dürfen. Zum Beispiel ist die Annahme, dass alle Elemente stets in der Menge bleiben, nur erlaubt wenn es eine entsprechende Einschränkung, etwa $b_{pre}.elems \subseteq b_{post}.elems$, auf t gibt. Wenn es jedoch eine solche Einschränkung gibt ist es nicht möglich, einen Untertyp s von t einzuführen, der diese Einschränkung verletzt; eine Routine zum Löschen von Elementen kann es in s daher nicht geben. Die üblichen Zusicherungen (vor allem Vorbedingungen) können nicht alle nötigen Einschränkungen ausdrücken, da Aufrufer nur den nach außen sichtbaren Objektzustand sehen und sich Zustände durch Nebenläufigkeit zwischen Überprüfung der Vorbedingungen und Ausführung einer Routine ändern können.

Genau genommen beziehen sich diese Einschränkungen nur auf die weitere Entwicklung eines Objekts (daher „history constraint“). Andere Formen von Einschränkungen, die immer gleichmäßig gelten, stellen ja Invarianten dar.

5.2.4 Untertypen und Generizität

Die beiden Arten des universell quantifizierten Polymorphismus, nämlich Subtyping und Generizität (bzw. parametrischer Polymorphismus), wurden auf sehr unterschiedliche Weise eingeführt. Subtyping ist nach dem Ersetzbarkeitsprinzip so definiert, dass Instanzen eines Untertyps überall eingesetzt werden können, wo Instanzen eines entsprechenden Obertyps erwartet werden. Generizität ist ein Machanismus zur Beschreibung einer Familie von Typen, die sich nur durch die für Typparameter eingesetzten Typen unterscheiden. Es stellt sich daher die Frage, wie sich diese beiden Arten von Polymorphismus zueinander verhalten.

Betrachten wir beispielsweise Listen von Elementen gleichförmiger Typen. Dies lässt sich in Ada auf natürliche Weise durch die Verwendung von Generizität ausdrücken:

```
generic
    type Data is private;
package Lists is
    type Elem is private;
    function Append (A,B: access Elem)
        return access Elem;
private
    type Elem is record
        Item: Data;
        Next: access Elem
    end record;
end Lists;

type Person is ...
package PersonLists is new Lists (Person);
```

Für jeden Typ kann damit generisch ein Listentyp mit den notwendigen Operationen erzeugt werden. Abgesehen von der Kapselung im Paket lässt sich ein entsprechender Effekt auch durch Vererbung erreichen:

```
type Elem is tagged record
    Next: access Elem
end record;
function Append (A,B: access Elem)
    return access Elem;
type PersonElem is new Elem with ...
```

Dieses Beispiel veranschaulicht, dass zumindest in einigen Fällen Subtyping und Generizität gegeneinander austauschbar sind. Kann man vielleicht mit einem der beiden Mechanismen alles ausdrücken, was man im jeweils anderen ausdrücken kann? Ohne Subtyping, also nur mit Generizität, ist es zum Beispiel nicht möglich, heterogene Listen auszudrücken; das sind Listen, die Elemente von mehr als nur einem Typ enthalten können. Heterogene Listen kommen in der Praxis aber vor. Daher ist Generizität sicher nicht mächtiger als Subtyping.

Dagegen hat man mit Subtyping alleine andere Probleme. Wenn man kontravariantes Subtyping verwendet, kann man beispielsweise die Schnittstelle der Funktion `Append` in obigem Beispiel nicht in einem Untertyp übernehmen, da die Eingangs- und Ausgangsparameter jeweils denselben Typ haben sollen. Jede Form von kovariantem Subtyping unterliegt aber, wie in Abschnitt 5.2.1 erklärt, verschiedenen unerwünschten Einschränkungen, die es bei Generizität nicht gibt. Daraus folgt, dass Generizität und Subtyping einander ergänzen.

Die Kombination von Generizität und Subtyping drückt sich sehr anschaulich in gebundener Generizität aus. Der Ausdruck `KONTO[T->GELDBETRAG]` in Eiffel besagt zum Beispiel, dass für den Typparameter `T` in der Klasse `KONTO` jeder beliebige Untertyp von `GELDBETRAG` eingesetzt werden kann. Durch die Beschränkung auf Untertypen von `GELDBETRAG` sind einige der Eigenschaften aller Instanzen von `T`, nämlich

jene, die auch alle Instanzen von `GELDBETRAG` haben, bekannt und können verwendet werden. Zum Beispiel können je zwei Instanzen von `T` addiert werden und liefern als Ergebnis wieder eine Instanz von `T`.

Gebundene Generizität verwendet Subtyping als ein Hilfsmittel. Wir fragen uns nun, unter welchen Bedingungen ein generischer Typ ein Untertyp eines anderen generischen Typs ist, also überall, wo eine Instanz des zweiten generischen Typs erwartet wird, auch eine Instanz des ersten generischen Typs verwendet werden kann. Im Fall von Eiffel ist die Frage einfach zu beantworten: Ein Typ (bzw. eine Klasse) x`[`a_1`->`$s_1, \ldots, a_n$`->`s_n`]` ist ein Untertyp (bzw. eine Unterklasse) von y`[`b_1`->`$t_1, \ldots, b_n$`->`t_n`]`, wenn für alle $1 \leq i \leq n$ gilt, dass s_i ein Untertyp von t_i und x`[`$s_1, \ldots, s_n$`]` ein Untertyp von y`[`$s_1, \ldots, s_n$`]` ist. Es ist klar, dass bei der Ersetzung der Parameter immer nur die spezifischeren konstanten Typausdrücke s_i verwendet werden können, da für alle anderen Typen eine Bedingung in x verletzt wäre.

Mit dieser Version von gebundenen Typparametern bzw. Subtyping für generische Typen gibt es aber ein Problem: Sie ist nur dann vollständig bzw. anwendbar, wenn alle Parametertypen kovariant sind, was in der Regel nicht gilt. Trotzdem werden solche Beziehungen in der Praxis für wichtig erachtet. Beispielsweise sind auf Arrays in Java und C# implizit solche Untertypbeziehungen definiert; `Student[]` ist ein Untertyp von `Person[]` wenn `Student` ein Untertyp von `Person` ist, obwohl es dadurch zu Typfehlern zur Laufzeit kommen kann: Enthält eine Variable vom deklarierten Typ `Person[]` ein Array des Typs `Student[]`, dann erlaubt die statische Typüberprüfung das Schreiben einer Instanz von `Person` in das Array; zur Laufzeit wird eine Ausnahmebehandlung ausgelöst, falls der dynamische Typ des in das Array geschriebenen Objekts ungleich `Student` ist.

Für Sprachen mit kontravariantem Subtyping werden wesentlich mächtigere Mechanismen benötigt, um komplexe Beziehungen zwischen den Typparametern ausdrücken zu können. Mit herkömmlichen Mitteln kann nur ausgedrückt werden, dass ein für einen Typparameter eingesetzter Typ ein Untertyp einer gegebenen Typkonstante ist. Man will zumindest auch festlegen können, dass ein eingesetzter Typ ein Obertyp einer Typkonstante ist, und das nicht nur bei der Deklaration eines Typparameters. Dafür erforderliche Sprachmechanismen wurden in der Version 1.5 zusammen mit Generizität zu Java hinzugefügt: Die Ausdrücke `? extends T` (beliebiger Untertyp von `T`) und `? super T` (beliebiger Obertyp von `T`) können in vielen Fällen statt konkreter Typen Typparameter ersetzen. Das geht aber nur, wenn auf entsprechende Variablen nur lesend (im Fall von `? extends T`, also Zugriffe, für die kovariante Typen gefordert werden) bzw. nur schreibend (im Fall von `? super T`, kontravariante Typen) zugegriffen wird. Dazu ein Beispiel in Java:

```
class List<T> {
   copyFrom (List<? extends T> f) {...}
   copyTo (List<? super T> t) {...}
   ...
}
```

Beide Methoden kopieren den Inhalt generischer Listen, `copyFrom` von der Liste `f` in die aktuelle Liste `this` vom Typ `T` und `copyTo` von `this` nach `t`. Auf `f` wird nur lesend und auf `t` nur schreibend zugegriffen. Dabei wird `List<Student>` nur dann als Untertyp von `List<Person>` aufgefasst, wenn auf den ersten Typ nur lesend oder auf den zweiten nur schreibend zugegriffen werden darf.

In der Theorie gibt es im Wesentlichen zwei Möglichkeiten, Klassen bzw. generische Typen zueinander in Beziehung zu setzen, nämlich F-gebundene Generizität und Subtyping höherer Ordnung. Angenommen, T bezeichnet einen generischen Typ und $T[S]$ einen Typ, der durch Ersetzung eines Typparameters in T durch den Typ S aus T abgeleitet wird. Im *F-gebundenen Polymorphismus* gilt die Beziehung $S \leq T[S]$, der Untertyp kann also den Typparameter des (generischen) Obertyps ersetzen. Einige aktuelle Sprachen mit Generizität verwenden F-gebundene Generizität und ermöglichen damit rekursive Definitionen wie in $S \leq T[S]$. In Java 1.5 sind beispielsweise viele Klassen und Schnittstellen folgendermaßen definiert:

```
interface Comparable<T> {
   int compareTo (T that);
}
class Integer implements Comparable<Integer> {
   public int compareTo (Integer that) {...}
}
```

Der Typ des formalen Parameters `that` wird durch einen Typparameter bestimmt, nicht durch Subtyping. Damit werden Einschränkungen beim Subtyping (kovariante bzw. binäre Routinen) vermieden und es ist trotzdem sichergestellt, dass `this` in `compareTo` denselben dynamischen Typ wie `that` hat. Dieser Ansatz hat aber einen Haken: Wenn $S \leq T[S]$ gilt und ein Untertyp $U \leq S$ existiert, kann $U \leq T[U]$ nicht mehr gelten, sondern nur $U \leq T[S]$ (wie aus S übernommen). Der Einsatz von Wildcard-Typen hilft manchmal weiter (z.B. $S \leq T$[`? extends` S] statt $S \leq T[S]$), schränkt aber die Art der Zugriffe (lesend oder schreibend) stark ein.

Eine alternative Beziehung mit ähnlicher Ausdrucksstärke ist *Subtyping höherer Ordnung*: Zwei generische Typen S und T sind in einer Untertypbeziehung höherer Ordnung, wenn $S[U] \leq T[U]$ für alle Typen U gilt.

Subtyping höherer Ordnung scheint mathematisch einfacher handhabbar zu sein als F-gebundener Polymorphismus und wird z.B. in Haskell und C++ verwendet.

Man kann nicht-generische Typen einfach nur als Spezialfall generischer Typen (mit einer leeren Menge von Typparametern) ansehen. Dann braucht man in einer Sprache nicht zwischen nicht-generischen und generischen Typen unterscheiden. Als Grundlage für die Vererbung kann man dann $<\#$—man nennt diese Beziehung in diesem Fall *Matching* definiert durch $S <\# T$ wenn $S[U] \leq T[U]$ für alle Typen U—anstatt der Untertypbeziehung verwenden. Leider erfüllt aber $<\#$ das Ersetzbarkeitsprinzip nicht. Der Grund dafür liegt darin, dass gebundene Typparameter in rekursiven Typen (siehe Abschnitt 4.3.2) kovariant als Eingangsparametertypen verwendet werden dürfen. Zum Beispiel gilt die Beziehung

$$\{\texttt{i:int}, \texttt{f:}u\to u, \texttt{next:}u\} <\# \{\texttt{i:int}, \texttt{f:}v\to v\}.$$

Eine derartige Beziehung ist in vielen Fällen durchaus erwünscht. Durch die Verletzung des Ersetzbarkeitsprinzips dürfen Aufrufe von Routinen aber nur dann polymorph sein wenn garantiert ist, dass kein Argument einen Typ mit möglicherweise kovarianten Eingangsparametertypen hat.

Zusammengefasst: Auch Subtyping höherer Ordnung unterstützt (wie F-gebundene Generizität) binäre Methoden. Aber auch dabei gibt es einen Haken: Man muss entweder statisch sicherstellen, dass in Aufrufen binärer Methoden bestimmte übergebene Argumente gleiche Typen haben (wodurch die Flexibilität eingeschränkt wird) oder akzeptieren, dass bei Matching Typfehler erst zur Laufzeit erkannt werden.

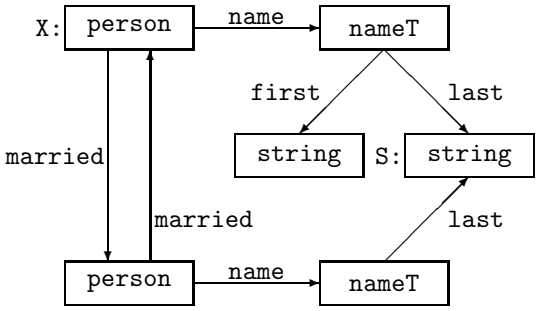
5.3 Logik und Subtyping

Typen und Subtyping können nicht nur in imperativen Sprachen gewinnbringend eingesetzt werden. In diesem Abschnitt wird dies an der Programmiersprache LIFE demonstriert, deren Wurzeln in der logischen Programmierung liegen.

ψ-Terme kombinieren logische Terme (siehe Abschnitt 2.2) mit Typen, auf denen ein Verband definiert ist (siehe Abschnitt 4.1). Stark vereinfacht ausgedrückt erweitern ψ-Terme gewöhnliche Terme so, dass erweiterbare verbundähnliche Datenstrukturen in logischen Sprachen einfacher repräsentiert werden können. Ein Beispiel veranschaulicht dies:

```
X:person(name =>
           nameT(first => string,
                 last => S:string),
         married =>
           person(name => nameT(last => S),
                  married => X))
```

`person`, `nameT` und `string` sind Typsymbole. In LIFE werden sie, in Anlehnung an Algebren, *Sorten* genannt. Der Begriff Typ wird vermieden, da ψ-Terme selbst, in gewisser Weise, Typen sind. `name`, `first`, `last` und `married` sind Label zur Bezeichnung von Verbundkomponenten, die *„features“* oder *Attribute* heißen. Die dritte Klasse von Symbolen sind Variablennamen, zu der auch `X` und `S` gehören. Wie in Prolog werden Variablennamen mit großem Anfangsbuchstaben und alle anderen Symbole klein geschrieben. Jede Sorte entspricht ungefähr einem Verbundtyp, wobei die Komponenten und deren Typen jedoch nicht festgelegt sind, sondern stets neue Komponenten dazugefügt werden können. Die Sruktur des obigen ψ-Terms lässt sich grafisch veranschaulichen:



Jedes Rechteck stellt einen Teilausdruck des ψ-Terms dar und enthält die Sorte dieses Teilausdrucks. Einigen Teilausdrücken ist ein Variablenname zugeordnet. Pfeile mit den entsprechenden Bezeichnungen veranschaulichen Attribute. Der ψ-Term besagt, dass `X` eine Instanz von `person` ist, die zumindest die Komponenten `name` und `married` der Sorten `nameT` bzw. `person` enthält, wobei der Name der Person aus je einem Vor- und Nachnamen besteht. Jene Person, mit der `X` verheiratet ist, ist auch mit `X` verheiratet und hat einen Namen, dessen Nachnamen-Komponente `S` mit dem Nachnamen von `X` ident ist.

Ein ψ-Term ist ein Ausdruck der Form x oder $x{:}t$ oder t, wobei x eine Variable und t ein „untagged“ ψ-Term ist. Ein „untagged“ ψ-Term ist ein Ausdruck der Form s oder $s(a_1 \Rightarrow t_1, \ldots, a_n \Rightarrow t_n)$, wobei s eine Sorte ist und $a_1, \ldots, a_n$ paarweise verschiedene Attribute und $t_1, \ldots, t_n$ ψ-Terme sind. Eine Variable x darf in einem ψ-Term höchstens einmal in der Form $x{:}t$ auftreten.

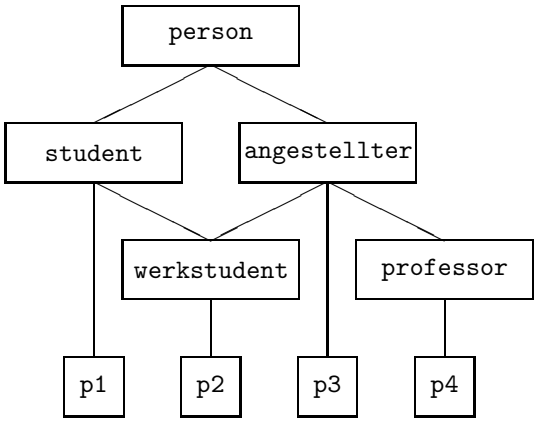
Auf ψ-Termen gibt es eine Halbordnung $\preceq$, die sich aus der Halbordnung $\leq$ auf der Menge der Sorten ableiten lässt. $t \preceq t'$ zwischen zwei ψ-Termen t und t' gilt genau dann, wenn

1. $s \leq s'$ für die äußersten Sorten s und s' von t bzw. t' gilt, und
2. alle Attribute $a_1, \ldots, a_n$ von t' auch Attribute von t sind, und

3. $t_i \preceq t'_i$ für $i \in \{1, \ldots, n\}$ und jedes $a_i \Rightarrow t_i$ in t und $a'_i \Rightarrow t'_i$ in t' gilt, und

4. Teilausdrücke in t identisch sind falls die entsprechenden Teilausdrücke in t' identisch sind.

Wenn $\leq$ auf der Menge der Sorten einen Verband beschreibt, dann beschreibt auch $\preceq$ auf der Menge aller ψ-Terme, abgesehen von Variablenbenennungen, einen Verband. Das bedeutet, dass es für je zwei ψ-Terme ein eindeutig bestimmtes Infimum gibt, das dem Ergebnis der Unifikation zwischen diesen beiden Termen entspricht. Für das folgende Beispiel verwenden wir diesen Ausschnitt aus einem Verband über Sorten:



Die Unifikation von

```
student(betreuer =>
          professor(assistent =>
                      angestellter),
        freund => person)
```

und

```
X:person(betreuer =>
           p4(assistent => X),
         vermieter => person)
```

liefert das Ergebnis

```
X:werkstudent(betreuer =>
                p4(assistent => X),
              freund => person,
              vermieter => person)
```

`p1` bis `p4` sind spezielle Sorten, die jeweils genau eine Instanz beschreiben. Man kann `p1` bis `p4` selbst als Werte auffassen. In LIFE werden Werte und Sorten nicht unterschieden. Jede Sorte stellt auch einen Wert dar und umgekehrt.

LIFE-Programme sehen wie Prolog-Programme aus und werden, abgesehen von der unterschiedlichen Unifikation, gleich ausgeführt. Zum Beispiel liefert das Programmstück

```
prueft(p4, p1).

fleissig(angestellter).
fleissig(X:student) :- prueft(professor, X).
```

auf die Anfrage „`fleissig(Y:student)`“ nacheinander die beiden Antworten „`Y=werkstudent`“ und „`Y=p1`“.

5.4 Typen für aktive Objekte

Nebenläufige (parallele) objektorientierte Sprachen beruhen manchmal auf *aktiven Objekten*. Ein aktives Objekt ist im Wesentlichen ein Prozess, der mit anderen Objekten über „message passing“ kommuniziert. Wie jedes Objekt besitzt auch ein aktives Objekt eine Identität, einen veränderlichen Zustand und ein Verhalten, das durch ein Programmstück festgelegt ist. Aktive Objekte können, genauso wie passive Objekte, typisiert sein. Über den Typen der aktiven Objekte lässt sich auch Subtyping realisieren, wie in Abschnitt 5.4.1 beschrieben. Allerdings ist die Vererbung des Programmcodes aktiver Objekte oft schwierig, wie wir in Abschnitt 5.4.2 sehen werden.

5.4.1 Prozesstypen

Man kann „tasks“ in Ada als aktive Objekte ansehen, denen ein Typ zuordenbar ist. Hier ist ein kleines Beispiel für einen Prozess und dessen Typ:

```
task type Buffer is
    entry Put (E: in Element);
    entry Get (E: out Element);
end Buffer;

task body Buffer is
    X: Element;
begin
    loop
        accept Put (E: in Element)
            do X := E;
        end;
        accept Get (E: out Element)
            do E := X;
        end;
    end loop;
end Buffer;
```

Der Typ beschreibt die Nachrichten, die von einem Prozess akzeptiert werden können. Die in der Typdeklaration bereitgestellte Information ist jedoch sehr lückenhaft. Vor allem fehlt die Information, wann und unter welchen Umständen eine Nachricht vom Prozess akzeptiert wird. Das Beispiel beschreibt einen Puffer, der zu jedem Zeitpunkt nur ein einziges Element enthalten kann. Das heißt, die Nachrichten `Put` und `Get` werden nur abwechselnd akzeptiert, wobei die erste Nachricht `Put` sein muss. Das Programmstück

```
B: Buffer; E1, E2: Element;
B.Put(E1); B.Put(E2); B.Get(E1);
```

würde also zu einem „deadlock“ führen, falls es keinen weiteren Prozess gibt, der entsprechende Nachrichten an `B` sendet. Aus dem Typ von `Buffer` ist dieses Verhalten aber nicht ersichtlich.

Ein geeignetes Typsystem für aktive Objekte müsste in der Lage sein, das Verhalten seiner Instanzen (Prozesse) so weit zu beschreiben, wie es für ihre richtige Verwendung notwendig ist. *Prozesstypen*, die auf einer Prozessalgebra (siehe Abschnitt 2.3.2) beruhen, können die notwendige Information beschreiben. Alle atomaren Aktionen in Prozesstypen beschreiben Nachrichten, die erwartet werden. Atomare Aktionen werden mittels sequentieller Komposition ($*$), paralleler Komposition ($\|$) und alternativer Komposition ($+$) zu Prozesstypen zusammengesetzt. Der folgende Prozesstyp entspricht obigem Ada-Beispiel:

```
Buffer = Put (in Element) *
         Get (out Element) *
         Buffer
```

Eine Instanz dieses Typs erwartet zuerst eine Nachricht mit der Schnittstelle `Put(in Element)`, danach eine mit `Get(out Element)`. Wegen des rekursiven Aufrufs von `Buffer` wird diese Sequenz unbeschränkt oft wiederholt. Daher ist obige Definition von `Buffer` äquivalent zu dieser:

```
Buffer = Put (in Element) *
         Get (out Element) *
         Put (in Element) *
         Get (out Element) *
         ...
```

Das nächste Beispiel zeigt den Prozesstyp eines Puffers, der beliebig viele Elemente enthalten kann:

```
IBuffer = ( Put (in Element) *
            Get (out Element) ) ||
          IBuffer
```

Ein aktives Objekt dieses Typs erwartet also gleichzeitig beliebig viele `Put`-Nachrichten, aber höchstens so viele `Get`-Nachrichten wie gerade Elemente im Puffer sind. Beliebig viele und beliebig geordnete `Put`- und `Get`-Nachrichten werden von den Instanzen des folgenden Prozesstyps erwartet:

```
UBuffer = Put (in Element) ||
          Get (out Element) ||
          UBuffer
```

Bei der Verwendung von Variablen eines Prozesstyps ist Vorsicht geboten: Durch bestimmte Zugriffsarten auf solche Variablen ändern sich die Typen. Wird eine Nachricht an den durch die Variable referenzierten Prozess geschickt, ändert sich der Typ implizit; der neue Typ ist jener, den der Prozess nach Verarbeitung der Nachricht haben wird. Hat die Variable `B` zum Beispiel den Prozesstyp `Buffer`, so wird durch Ausführung des Ausdrucks `B.Put(E1)` der Typ von `B` zu `Get(out Element)` $*$ `Buffer` verändert. `B` erwartet danach also eine `Get`-Nachricht.

Wenn mehrere Prozesse gleichzeitig Nachrichten an einen weiteren Prozess schicken, können sie sich gegenseitig beeinflussen. Um sicherzustellen, dass dadurch keine Typinkompatibilitäten auftreten, werden *Rechte* für das Senden von Nachrichten an Variablen vergeben. Diese Rechte werden durch Prozesstypen ausgedrückt. Über eine Variable darf nur dann eine Nachricht an den durch die Variable referenzierten Prozess gesendet werden, wenn der Prozesstyp der Variable diese Nachricht als nächste erwartet. Ein neu aufgespannter Prozess wird nur über *eine* Variable referenziert, deren Prozesstyp das nach außen sichtbare Verhalten des Prozesses vollständig beschreibt. Wird eine neue Variable, die auf denselben Prozess zeigt, angelegt (z.B. durch Verwendung der Variable als Argument einer Routine), muss der ursprüngliche Prozesstyp auf die beiden Variablen aufgeteilt werden; das heißt, wenn der ursprüngliche Prozesstyp von der Form $\tau \parallel \tau'$ ist, dann ist der neue Prozesstyp der einen Variable τ und jener der anderen τ'. Die Aufspaltung darf nur an einer durch den Operator „$\|$“ gekennzeichneten Stelle erfolgen. Dadurch können sich die Nachrichten, die gemäß τ und τ' gesendet werden, beliebig überlappen, ohne sich gegenseitig zu stören.

Auf Prozesstypen lässt sich durch einige Regeln eine Untertyp-Relation $\leq$ definieren. Mit Hilfe der folgenden Regeln lässt sich zum Beispiel zeigen, dass `UBuffer` ein Untertyp von sowohl `Buffer` als auch `IBuffer` ist und `IBuffer` ein Untertyp von `Buffer` ist. Dies ist auch intuitiv klar, da man auch einen unendlich großen Puffer so verwenden kann als ob er nur ein Element beinhalten könnte. Einen unbeschränkten Puffer kann man auch so verwenden, als ob er durch eine Bedingung eingeschränkt wäre. Bei solchen Verwendungen werden zwar nicht alle Möglichkeiten ausgeschöpft, aber es können dabei keine Inkonsistenzen auftreten.

$$\tau \leq \tau$$

$$\tau \leq \varepsilon \quad (\varepsilon \text{ entspricht dem leeren Typ } \top)$$

$$\frac{\tau_1' \leq \tau_1 \ \ldots \ \tau_n' \leq \tau_n}{m(\tau_1, \ldots, \tau_n) \ \leq \ m(\tau_1', \ldots, \tau_n')}$$

$$\frac{\tau' \leq \tau''}{\tau * \tau' \ \leq \ \tau * \tau''} \qquad \frac{\tau \leq \tau'' \quad \tau' \leq \tau'''}{\tau + \tau' \ \leq \ \tau'' + \tau'''}$$

$$\frac{\tau \leq \tau'' \quad \tau' \leq \tau'''}{\tau \parallel \tau' \ \leq \ \tau'' * \tau'''} \qquad \frac{\tau \leq \tau'' \quad \tau' \leq \tau'''}{\tau \parallel \tau' \ \leq \ \tau'' \parallel \tau'''}$$

Die Typkonsistenz von Prozesstypen, die auf Prozessalgebren beruhen, lässt sich statisch überprüfen. In

einem typkonsistenten Programm ist garantiert, dass alle gesendeten Nachrichten vom Empfänger verstanden werden, und zwar in der Reihenfolge, in der die Nachrichten gesendet wurden. Eine neuere Generation von Prozesstypen, deren Syntax nicht mehr auf Prozessalgebren beruht sondern eher der von konventionellen Typen ähnelt, beseitigt einige kleinere Probleme und bietet dem Programmierer mehr Flexibilität. Die oben erläuterten Grundprinzipien wie Änderung eines Typs beim Senden von Nachrichten und Typaufspaltung beim Erzeugen neuer Referenzen blieben auch in der neueren Generation erhalten.

5.4.2 Vererbungsanomalie

Vor längerer Zeit war die Untersuchung einiger Zusammenhänge zwischen der objektorientierten und negenläufigen Programmierung ein sehr aktives Forschungsgebiet. Bereits sehr früh wurde erkannt, dass die Wiederverwendung von Code für nebenläufige Programme viel schwieriger und weniger erfolgreich realisiert werden kann als für sequentielle Programme. Ein Grund dafür liegt vermutlich in der notwendigen Synchronisation nebenläufiger Prozesse: Wenn eine Unterklasse (bzw. ein abgeleiteter Typ) eine Klasse durch das Dazufügen neuer Operationen erweitert, ergeben sich meist gänzlich neue Synchronisationsbedingungen, sodass der Synchronisationscode für viele ansonsten nicht veränderte Operationen umgeschrieben werden muss. Und die Erweiterung der Funktionalität durch das Dazufügen von Operationen ist eine häufige Ursache für die Verwendung von Unterklassen. Dieses Problem wird häufig als *Vererbungsanomalie* in nebenläufigen, objektorientierten Sprachen bezeichnet.

Die Trennung des Synchronisationscodes vom restlichen Code kann bei der Umgehung der Vererbungsanomalie hilfreich sein, da damit wenigstens der restliche Code uneingeschränkt wiederverwendet werden kann. Auch Synchronisationscode kann durch diese Trennung einfacher wiederverwendet werden, da dafür eine eigenständige Darstellung in einer unabhängigen Klassenhierarchie eingeführt werden kann. In der Praxis zeigt sich jedoch, dass eine klare Trennung zwischen diesen Code-Arten oft nur schwer möglich ist.

Ein bekannter Vorschlag zur Lösung des Problems der Vererbungsanomalie ist, die Synchronisationsbedingungen in Unterklassen gegenüber den entsprechenden Synchronisationsbedingugen in Oberklassen einzuschränken. Eine Operation in einer Unterklasse hat demnach sicher nicht mehr Freiheiten als eine entsprechende Operation in einer Oberklasse. Dieser Vorschlag ist aber nur von begrenzter praktischer Bedeutung, da Typen auf aktiven Objekten (z.B. Prozesstypen) genau die umgekehrte Struktur haben, Untertypen also mehr Freiheiten erlauben als die entsprechenden Obertypen.

5.5 Eigenschaften ausgewählter objektorientierter Sprachen

In diesem Abschnitt sind einige charakteristische Merkmale der Typsysteme einiger bekannter objektorientierter Programmiersprachen kurz zusammengefasst. Ziel dieser Zusammenfassung ist, die in diesem Skriptum verwendete Terminologie sowie generelle Eigenschaften und Einschränkungen von Typsystemen anhand einiger ausgewählter Sprachen, von denen angenommen wird, dass sie den Studierenden teilweise bekannt sind, noch einmal zu demonstrieren. Es wird keinerlei Anspruch auf Vollständigkeit erhoben. Die Beurteilungen der Sprachen erfolgten hauptsächlich nach subjektiven Kriterien.

5.5.1 Smalltalk

Smalltalk ist eine bereits recht alte, aber in einigen Bereichen noch immer beliebte objektorientierte Sprache, die in eine sehr umfangreiche Bibliothek von Klassen und Objekten eingebettet ist. Jede Klasse definiert implizit auch einen Typ. Variablen sind in (den meisten Dialekten von) Smalltalk nicht typisiert; d.h. eine Variable kann einen Zeiger auf ein Objekt jedes beliebigen Typs enthalten. Deswegen—und aufgrund des interpretativen Charakters der Sprache—sind Typüberprüfungen im Wesentlichen nur zur Laufzeit durchführbar. Im Prinzip kann jede beliebige Nachricht an jedes beliebige Objekt geschickt werden. Das Objekt entscheidet aufgrund der Nachricht, ob eine der Nachricht entsprechende Methode oder eine speziell für unverstandene Nachrichten vorgesehene Methode ausgeführt wird. Es wird Einfachvererbung unterstützt. Klassen haben hauptsächlich die Aufgabe, abstrakte Datentypen zu realisieren. Wegen der untypisierten Variablen hat Generizität keine Bedeutung; jede Klasse kann prinzipiell als generisch angesehen werden, wobei Typkonsistenz aber nicht vom System, sondern nur vom Programmierer sicherzustellen ist. Ähnlich verhält es sich mit Subtyping: Da jedes Objekt prinzipiell mit jeder Nachricht umgehen kann, spielt Subtyping aus der Sicht des Systems keine Rolle. Um das gewünschte Verhalten zu erreichen, müssen sich Programmierer selbst ohne Systemunterstützung darum kümmern, dass das Ersetzbarkeitsprinzip erfüllt ist.

Zusammenfassend kann man sagen, dass Smalltalk eine sehr vielseitige objektorientierte Sprache ist, in der Typen eine (im Vergleich zu anderen objektorientierten Sprachen) eher untergeordnete Rolle spielen. Allerdings gibt es keine Löcher im Typsystem; Smalltalk ist also vollständig typisiert. Eine ähnlich nebensächliche Rolle spielen Typen in Objective C, einer objektorientierten Erweiterung von C, ebenso wie in Self, einer Sprache, in der sogar die Vererbungshierarchie

(bzw. „delegation“) dynamisch änderbar ist, und in CLOS, einer vielzitierten Erweiterung von Lisp. Modernere dynmische objektorientierte Sprachen wie Ruby haben zwar eine aktuellere Syntax und sind weniger stark in ein System integriert, aber die Wurzeln in Smalltalk sind noch immer deutlich zu erkennen. In diesen Sprachen müssen sich Programmierer ohne Systemunterstützung darum kümmern, dass nur erwartete Nachrichten an Objekte geschickt werden.

5.5.2 C++

In C++ sind, wie in vielen ähnlichen Sprachen, Typen, Module und erweiterbare Verbunde zum Konzept der Klassen vereint. Die Sichtbarkeit und Vererbbarkeit von Verbundkomponenten wird durch die Schlüsselwörter `public`, `private` und `protected` festgelegt. Zwischen Vererbung und Subtyping wird auf der Sprachebene nicht unterschieden. Parametertypen sind stets invariant. Um einige Einschränkungen durch die fehlende Trennung zwischen Typen und Modulen zu umgehen, wurden „friends“ eingeführt. Sie erlauben den benannten „Freunden“ einer Klasse, auch auf solche Klassenkomponenten zuzugreifen, auf die sie sonst keinen Zugriff hätten. Der Zugriff auf überschriebene Implementierungen von Funktionen ist möglich.

C++ hat sich aus der Sprache C entwickelt, in der Typumwandlungen eine große Rolle spielen. Diese Eigenschaft ist von C++ beibehalten und sogar uneingeschränkt auf Klassen ausgeweitet worden. Durch Typumwandlungen kann der Programmierer die Einschränkungen des Typsystems beinahe beliebig umgehen. Die formale Überprüfung der Typkonsistenz durch den Compiler hat daher in manchen Fällen nur sehr wenig mit der Konsistenz der semantischen Bedeutungen der Typen zu tun. Typüberprüfungen zur Laufzeit sind in C bzw. C++ nicht vorgesehen. Neben beliebigen expliziten Typumwandlungen zählen auch fehlende Überprüfungen der Bereichsgrenzen bei Feldzugriffen, Varianten ohne Angabe der verwendeten Labels sowie Probleme bei der dynamischen Speicherverwaltung (keine „garbage collection“) zu den bekannten Lücken im Typsystem, mit denen der Programmierer umgehen muss.

Demgegenüber bietet C++ ein relativ reichhaltiges Typsystem. Es werden alle Arten von Polymorphismus unterstützt: Neben Typumwandlungen gibt es überladene Operatoren, Subtyping (zusammen mit Mehrfachvererbung) und Generizität durch „templates“. Letztere nützen „macros“ zur typkonformen Ersetzung von (gebundenen) Typparametern durch Typen. In C++ haben auch Funktionen Typen. Weiters gibt es abstrakte Typen, die durch Vererbung zu konkreten Typen erweitert und in Schnittstellenbeschreibungen eingesetzt werden können. Auch die explizite dynamische Typerkennung wird durch RTTI („Run Time Type Information“) in einem gewissen Maß unterstützt.

Insgesamt ergibt sich ein etwas zwiespältiges Bild: Einerseits bietet C++ ein sehr umfangreiches Typsystem, das bis auf einige Konstrukte Typüberprüfungen zur Compilezeit ermöglicht. Andererseits ist es dem Programmierer freigestellt, dieses Typsystem beliebig zu umgehen—sicherlich oft unabsichtlich bzw. ohne die Konsequenzen abschätzen zu können. Zur Nutzung der Vorteile des Typsystems in der Softwareentwicklung und -wartung ist daher sehr viel Erfahrung und eine strenge Programmierdisziplin notwendig. Ein wichtiger Punkt auf der Negativseite ist ein kaum vorhandenes Modulkonzept, woraus sich eine schlechte Überprüfbarkeit der Modulintegrität bei unabhängiger Übersetzung und (trotz eigener Sprachkonstrukte) schlechte Unterstützung getrennter Namensräume ergibt.

5.5.3 Java und C#

Die Programmiersprache Java wird heute in ganz großem Umfang eingesetzt. Die Typsysicherheit von Java wird des öfteren als großer Fortschritt gegenüber C++ angepriesen. Tatsächlich sind in Java die Lücken im Typsystem geschlossen. Gründe dafür sind der Verzicht auf Typumwandlungen zugunsten von dynamischer Typerkennung, dynamische Überprüfungen der Bereichsgrenzen bei Feldzugriffen und eine komplette Reorganisation der dynamischen Speicherverwaltung. Dadurch kann der Programmierer Typüberprüfungen nicht mehr umgehen, sondern höchstens zur Laufzeit hin verschieben. Funktionen in Unterklassen können im Normalfall überschrieben werden. Durch Einschränkung auf Einfachvererbung werden alle Probleme der Mehrfachvererbung vermieden.

Generizität wurde erst spät (mit Version 1.5) zu Java hinzugefügt. Es war eine ingenieurmäßige Meisterleistung, diese Erweiterung so zustande zu bringen, dass die ursprünglichen Standardbibliotheken unverändert sowohl mit als auch ohne Generizität einsetzbar sind. Andererseits mussten dabei auch Kompromisse eingegangen werden, durch die die Generizität aus heutiger Sicht nicht mehr so mächtig wie in vergleichbaren anderen Sprachen ist.

Ein weiteres Problem ist das Fehlen der Möglichkeit, Komponenten von Klassen umzubenennen, was leicht zu Namenskonflikten führen kann. Das Konzept der Pakete ist in Java nicht sehr ausgefeilt und führt gelegentlich zu Problemen.

Einige vielgepriesene Eigenschaften des Typsystems erweisen sich bei genauerem Hinsehen als weniger überzeugend als erwartet. Zum Beispiel sind Subtyping und Vererbung zwar auf der Ebene der Sprachbeschreibung getrennt, in der Realisierung jedoch sehr eng miteinander verknüpft. Die „interfaces“ in Java sind nichts

anderes als eine eingeschränkte Form abstrakter Typen. Dass Mehrfachvererbung zwar auf „interfaces“ aber nicht auf Klassen definiert ist, stellt eine Einschränkung gegenüber einigen anderen Sprachen dar.

Zusammengefasst kann man sagen, dass Javas Typsystem zwar in mancher Hinsicht sicher einen Fortschritt gegenüber dem von C++ darstellt, insgesamt jedoch weit hinter den heutigen Möglichkeiten zurück bleibt. Es gibt zahlreiche praktikable Vorschläge zur Verbesserung von Java, welche die meisten gegenwärtigen Unzulänglichkeiten beseitigen, die Sprache aber auch komplizierter machen würden.

Die Programmiersprache C# ist Java sehr ähnlich. In Details bietet C# kleine Vorteile gegenüber Java, vor allem hinsichtlich der Unterstützung von Paketen und der Generizität. Verbesserungen in der Unterstützung der Generizität mussten jedoch auch duch bedeutende Änderungen in den Standardbibliotheken erkauft werden. Der größte Unterschied zwischen C# und Java liegt in den verwendeten Standardbibliotheken. Der Zwischencode und die Bibliotheken von C# (.NET) werden von mehreren Programmiersprachen verwendet und sind ganz auf das Betriebssystem Windows zugeschnitten, während Java-Zwischencode großen Wert auf Kompatibilität zwischen ganz unterschiedlichen Betriebssystemen legt.

5.5.4 Ada

Ada hat ein recht umfangreiches Typsystem, bei dem sehr viele Typüberprüfungen bereits vom Compiler durchgeführt werden können und alle anderen Überprüfungen (bei entsprechenden Compileroptionen) zur Laufzeit erfolgen. Typen und Implementierungen sind zu einem Konzept vereint, Module aber unabhängig davon. Einfachvererbung wird direkt unterstützt, wobei Parametertypen auf eingeschränkte Weise kovariant sind. Es werden alle Arten von Polymorphismus unterstützt. Zwischen Subtyping und Vererbung wird nicht unterschieden. Abstrakte Typen, die Unterstützung abstrakter Datentypen und Bereichseinschränkungen, die eine einfache Form von Zusicherungen darstellen, gehören zu den Stärken von Ada.

Insgesamt macht das Typsystem aktueller Ada-Versionen einen ausgereiften Eindruck. Allerdings mussten aus Kompatibilitätsgründen zu Ada 83 Kompromisse eingegangen werden, welche die Qualitäten des Typsystems nicht einfach erkennen lassen. Ada verwendet eine Terminologie, die sich teilweise drastisch von der üblichen Terminologie unterscheidet. Außerdem ist Ada eine sehr umfangreiche Sprache, sodass viel Erfahrung notwendig ist, um die Stärken von Ada ausnützen zu können. Nicht zuletzt erfordert die Verwendung objektorientierter Techniken in Ada einen, im Vergleich zu anderen Sprachen, relativ hohen Schreibaufwand.

5.5.5 Eiffel

Auch Eiffel hat ein recht umfangreiches Typsystem, bei dem überraschend viele Typüberprüfungen zur Compilezeit durchführbar sind, obwohl Subtyping mit kovarianten Eingangsparametertypen verwendet wird. Trotzdem müssen viele Überprüfungen zur Laufzeit vorgenommen werden. Dynamische Typüberprüfungen lassen sich ausschalten. Typen, Module und erweiterbare Verbunde sind zu Klassen vereint. Die fehlende Trennung zwischen Typen und Modulen wird durch das gezielte Sichtbarmachen von „features“ (Routinen und Variablen) in anderen Klassen großteils kompensiert. Zwischen Mehrfachvererbung und Subtyping wird nicht unterschieden. Allgemeine Probleme bei der Mehrfachvererbung werden durch komplexe Mechanismen zum Umbenennen und Überschreiben von „features“ weitgehend beseitigt. Abstrakte Typen und gebundene Generizität werden unterstützt. Die wichtigste Besonderheit von Eiffel ist ein ausgefeiltes Konzept von Zusicherungen, die sich ausreichend gut mit Subtyping vertragen, aber zum größten Teil nur dynamisch überprüfbar sind.

Der semantischen Bedeutung eines Typs wird große Bedeutung beigemessen. Durch kovariante Eingangsparametertypen und fehlende Trennung zwischen Typen und Modulen bzw. Subtyping und Vererbung werden jedoch manchmal recht diffizile Regeln notwendig, deren Auswirkungen für den Programmierer nicht immer leicht verständlich sind. Außerdem wird die getrennte Übersetzung von Programmteilen nur unzureichend unterstützt.

5.6 Literaturhinweise

Die offiziellen Sprachstandards von Ada sind auch im Internet verfügbar [3]. Daneben gibt es umfangreiche weitere Literatur darüber. Als Beispiel sei eine Artikelserie über Ada in der Zeitschrift „Communications of the ACM“ erwähnt [28]. Eiffel ist in [19] beschrieben, der neue Standard in [10]. Zur Unterscheidung zwischen Vererbung von Implementierung und Subtyping gibt es eine große Zahl an Publikationen. Eine kurze und verständliche Beschreibung des Problems findet sich in [15]. Die Sprache Portlandish wird in [25] vorgestellt.

Es gibt zahlreiche Publikationen zu den Themen Subtyping und Vererbung. Zum Beispiel geben Liskov und Wing [16] eine noch immer aktuelle, auf Verhaltensspezifikationen basierende und relativ gut lesbare Definition von Subtyping. Viele leicht zugängliche Artikel zu diesem Thema finden sich in der Zeitschrift „The Journal of Object-Oriented Programming“ und in den Proceedings der jährlich abgehaltenen Konferenzen ECOOP („European Conference on Object-

Oriented Programming“, Proceedings meist bei Springer in der Serie „Lecture Notes in Computer Science“ erschienen) und OOPSLA („Object-Oriented Programming Systems, Languages and Applications“), seit einigen Jahren unter dem neuen Namen SPLASH, meist als Oktoberausgabe der „ACM SIGPLAN Notices“ erschienen.

Auch zur Vererbungsanomalie in nebenläufigen objektorientierten Sprachen wurden zahlreiche Artikel veröffentlicht. Eine gut Beschreibung des Problems wird in [18] gegeben.

Die Sprache LIFE [4] beruht auf ψ-Termen. Da dieses Thema einige theoretisch interessante Probleme aufwirft, sind die meisten Artikel dazu, wie der oben zitierte, sehr formal und eher schwer verständlich.

Prozesstypen (der neueren Generation) sind z.B. in [26] beschrieben. Da es sich dabei um ein aktuelles Forschungsthema am Institut für Computersprachen, Arbeitsgruppe Programmiersprachen und Übersetzerbau handelt, sind Sie herzlich eingeladen, sich entsprechende weiterführende Literatur direkt dort zu besorgen oder über ein Praktikum oder eine Diplomarbeit selbst mitzumachen.

Literaturverzeichnis

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects.* Springer, 1996.

[2] *The Programming Language Ada: Reference Manual.* ANSI/MIL-STD-1815A-1983, American National Standards Inst., Inc., Springer, 1983.

[3] *Annotated Ada Reference Manual.* International Standard ISO/IEC 8652:1995(E). Verfügbar unter „http://lglwww.epfl.ch/Ada/“.

[4] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. *The Journal of Logic Programming,* 16:195–234, 1993.

[5] J. C. M. Baeten and W. P. Weijland. *Process Algebra.* Cambridge University Press, 1990.

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

[7] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* 2nd edition, Springer, 1984.

[8] Scott Danforth and Chris Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys,* 20(1):29–72, March 1988.

[9] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer, 1985.

[10] Standard ECMA-367. *Eiffel analysis, design and programming Language.* Download: „http://www.ecma-international.org/publications/standards/Ecma-367.htm“.

[11] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts.* 3rd edition, Wiley, 1998.

[12] Carl A. Gunter and John C. Mitchell (eds.). *Theoretical Aspects of Object-Oriented Programming. Types, Semantics, and Language Design.* The MIT Press, 1994.

[13] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems,* 15(2):253–289, 1993.

[14] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys,* 21(3):360–411, 1989.

[15] Wilf LaLonde and John Pugh. Subclassing $\neq$ subtyping $\neq$ is-a. *Journal of Object-Oriented Programming,* 3(5):57–62, January 1991.

[16] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems,* 16(6):1811–1841, November 1994.

[17] John W. Lloyd. *Foundations of Logic Programming.* 2nd extended edition, Springer, 1987.

[18] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*, G. Agha et al. (eds.), The MIT Press, Cambridge, 1993.

[19] Bertrand Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

[20] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[21] Robin Milner. The Polyadic π-Calculus: A Tutorial. In *Logic and Algebra of Specification,* pp. 203–246, Springer-Verlag, 1992.

[22] R. Milner, J. Parrow und D. Walker. A Calculus of Mobile Processes (Parts I and II). *Information and Computation,* Vol. 100, pp. 1–77, 1992.

[23] John C. Mitchell. *Foundations for Programming Languages.* The MIT Press, 1996.

[24] Frank Pfenning (editor). *Types in Logic Programming.* The MIT Press, 1992.

[25] Harry H. Porter. Separating the subtype hierarchy from the inheritance of implementation. *Journal of Object-Oriented Programming,* 4(9):20–29, February 1992.

[26] Franz Puntigam. Coordination Requirements Expressed in Types for Active Objects. In *Proceedings ECOOP'97,* Springer-Verlag, LNCS 1241, Jyväskylä, Finland, June 1997.

[27] Leon Sterling and Ehud Shapiro. *Prolog: fortgeschrittene Programmiertechniken.* (Deutsch) Addison-Wesley, 1988.

[28] S. Tucker Taft. Ada 9X: A technical summary. *Communications of the ACM,* 35(11):77–82, November 1992.

[29] Peter Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger,* 1(1):7–87, August 1990.

Glossar

abgeleiteter Typ (engl. „derived type“). Durch Typableitung wird ein neuer Typ erzeugt, der dieselbe Struktur, aber einen anderen Namen wie ein vorgegebener Typ hat. → 3.1.3, 5.1.1.

abstrakter Datentyp (engl. „abstract data type“, abgek. ADT). Ein ADT ist ein Typ, dessen Instanzen abgeschlossene Einheiten sind; auf einer Instanz sind nur die vom ADT exportierten Operationen anwendbar. → 1.2.2.

abstrakter Typ (engl. „abstract type“). Ein abstrakter Typ ist ein, im Gegensatz zu einem konkreten Typ, unvollständig spezifizierter Typ. → 1.2.2, 5.1.1.

Attribut (engl. „attribute“). In vielen objektorientierten Sprachen versteht man unter einem Attribut den Namen einer Komponente zur Beschreibung des Objektzustands; oft ist das einfach der Name einer Verbundkomponente. In Ada ist ein Attribut eine einem Typ oder Wert zugeordnete Funktion. → 3.1.1.

Aufzählungstyp (engl. „enumeration type“). Ein Aufzählungstyp beschreibt eine geordnete, endliche Menge nichtstrukturierter Werte. Beispiele sind Wahrheitswerte (engl. „Boolean“) und Zeichen (engl. „character“). → 3.1.1.

Ausnahmebehandlung (engl. „exception handling“). Das ist die Ausführung eines speziell dafür vorgesehenen Programmstückes beim Auftreten einer Ausnahmesituation während der Programmausführung. Eine solche Ausnahmesituation ist z. B. das Erkennen eines Typfehlers zur Laufzeit.

definite Horn-Klausel (engl. „definite horn clause“). Ein Ausdruck in der von Prolog verwendeten Untermenge der Prädikatenlogik. → 2.2.2.

diskreter Typ (engl. „discrete type“). Zu den diskreten Typen zählen die Typen von ganzer Zahlen und Aufzählungen. Instanzen diskreter Typen eignen sich zur Indizierung von Feldern. → 3.1.

Diskriminante (engl. „discriminant part“). Das ist in Ada eine spezielle Komponente eines Verbunds, die dazu beiträgt, die Struktur des entsprechenden Verbundtyps zu bestimmen. → 3.1.2.

dynamisches Binden (engl. „dynamic binding“). Es wird erst zur Laufzeit festgelegt, welche Routine an einen Aufruf „gebunden“ wird. Gegenteil: statisches Binden. → 1.2.3.

Einfach-Binden (engl. „single dispatching“). Die Auswahl der auszuführenden Routine erfolgt über den dynamischen Typ eines einzigen Parameters. Gegenteil: Mehrfach-Binden. → 5.2.1.

Ersetzbarkeit (engl. „substitutability“). Prinzip der Ersetzbarkeit: „Eine Instanz eines Untertyps kann überall dort verwendet werden, wo eine Instanz eines entsprechenden Obertyps erwartet wird.“ → 4.3.1, 5.2.1.

Feld (engl. „array“). Das ist eine indizierte Menge von Werten oder Objekten gleichen Typs. Die Indexmenge ist durch einen diskreten Typ beschrieben. → 3.1.2.

freie Algebra (engl. „free algebra“). Das ist eine Algebra, in der neben den durch eine Menge Δ beschriebenen und aus Δ ableitbaren Gesetzen keine weiteren Gesetze gelten. → 2.3.

generischer Typ (engl. „generic type“). Siehe „parametrischer Typ“.

heterogene Algebra (engl. „multi-sorted algebra“). Das ist eine mathematische Struktur bestehend aus einem Tupel von durch Sorten benannten Trägermengen und einem System von Operationen darauf. → 2.3.

indiziert (engl. „indexed“ oder „labeled“). Mit einem Index bzw. „label“ versehen. Z. B. sind die Komponenten eines Verbunds mit den Komponentennamen indiziert.

Instanz (engl. „instance“). Ein durch einen Typ beschriebenes Element (Wert oder Objekt) ist eine Instanz dieses Typs.

Invariante (engl. „invariant“). Eine formal festgelegte Bedingung, die während der Ausführung eines bestimmten Programmstückes stets erfüllt sein muss. Meist wird nur gefordert, dass eine Invariante am Beginn und Ende der Ausführung eines Programmstücks erfüllt ist.

klassenweiter Typ (engl. „class-wide type“). In Ada: Der klassenweite Typ `t'Class` eines erweiterbaren Typs t umfasst alle Instanzen von t und der von t abgeleiteten Typen. Gegenteil: spezifischer Typ. → 5.1.1.

Lambda-Kalkül (engl. „lambda-calculus“). Der λ-Kalkül (λ ist der griechische Buchstabe „Lambda“) ist ein formales Modell zur Beschreibung von Funktionen. → 2.1.

Mehrfach-Binden (engl. „multiple dispatching“). Die Auswahl der auszuführenden Routine erfolgt über die dynamischen Typen mehrerer Argumente. Gegenteil: Einfach-Binden. → 5.2.1.

monomorphes Typsystem (engl. „monomorphic type system“). Ein monomorphes Typsystem ist ein Typsystem, in dem jeder Wert bzw. jedes Objekt eine Instanz von genau einem Typ ist. Gegenteil: polymorphes Typsystem. → 1.2.3.

Nachbedingung (engl. „postcondition“). Das ist eine Zusicherung, die nach der Ausführung einer bestimmten Operation erfüllt sein muss. → 5.1.2.

Obertyp (engl. „supertype“). Ein Typ, der eine Obermenge der Instanzen eines entsprechenden Untertyps beschreibt; der Obertyp ist allgemeiner als der Untertyp. Für t ist ein Obertyp von s schreiben wir $s \leq t$.

Objekt (engl. „object“). Ein Objekt ist eine abgeschlossene Einheit mit eigenständiger Identität, einem (veränderlichen) Zustand und einem (manchmal veränderlichen) Verhalten. Ein **passives Objekt** wird meist als Verbund dargestellt; das Verhalten wird durch eine Menge von Methoden (Routinen) auf diesem Objekt beschrieben. Ein **aktives Objekt** entspricht einem Prozess; das Verhalten wird durch das vom Prozess ausgeführte Programmstück beschrieben.

parametrischer Typ (engl. „parametric type“, auch *generischer Typ* genannt). Das ist ein Typ bzw. Typausdruck, in dem Typparameter verwendet werden, welche durch Typausdrücke ersetzt werden können. Ein **gebundener parametrischer Typ** schränkt die Menge der Typausdrücke, die für Typparameter eingesetzt werden können, ein. → 1.2.3, 2.3.1, 3.3, 4.2, 5.2.4.

PER-Modell. Ein auf partiellen Äquivalenzrelationen (engl. „partial equivalence relation“ = PER) beruhendes semantisches Modell für Typen in objektorientierten Sprachen. → 4.4.

polymorphes Typsystem (engl. „polymorphic type system“). In einem polymorphen Typsystem kann jeder Wert bzw. jedes Objekt eine Instanz von mehr als nur einem Typ sein. Die Operationen polymorpher Typen sind auf Operanden mehrerer Typen anwendbar. Gegenteil: Monomorphes Typsystem. → 1.2.3, 4, 5.

Prozessalgebra (engl. „process algebra“). Das ist eine Algebra, die dazu geeignet ist, das Verhalten nebenläufiger (paralleler) Prozesse zu beschreiben. → 2.3.2.

Prozesstyp (engl. „process type“). Allgemein: Der Typ eines Prozesses (z.B. in Ada). Speziell: Ein Typ, der das dynamische Verhalten eines Prozesses partiell beschreibt. → 3.2.2, 5.4.1.

Psi-Term (engl. „ψ-term“). Term in der logischen Programmiersprache LIFE. → 5.3.

Signatur (engl. „signature“). Eine Signatur ist die Schnittstellenbeschreibung einer Routine oder Klasse in einer Programmiersprache bzw. einer Operation in einer Algebra. → 2.3.

skalarer Typ (engl. „scalar type“). Ein skalarer Typ beschreibt eine Menge von Werten ohne sichtbare innere Struktur und die auf diesen Werten ausführbaren Operationen. → 3.1.1.

spezifischer Typ (engl. „specific type“). In Ada: Ein (nicht klassenweiter, erweiterbarer) Typ. Gegenteil: klassenweiter Typ. → 5.1.1.

statisches Binden (engl. „static binding“). Es wird bereits vom Übersetzer festgelegt, welche Funktion oder Prozedur an einen Aufruf „gebunden“ wird. Gegenteil: dynamisches Binden. → 1.2.3.

Subtyping. Ein Konzept eines polymorphen Typsystems, in dem ein Untertyp eine Teilmenge der Instanzen beschreibt, die durch einen entsprechenden Obertyp beschrieben sind. → 1.2.3, 4, 5.

Typ (engl. „type“). Allgemeine Definition: *Ein Typ beschreibt eine Menge von Instanzen.* Es gibt zahlreiche unterschiedliche konkretere Definitionen. In objektorientierten Sprachen: *Ein Typ ist eine Spezifikation der Schnittstelle und (in vielen Fällen) des nach außen sichtbaren Verhaltens seiner Instanzen (= Objekte).* → 1.1, 5.2.2.

Typäquivalenz (engl. „type equivalence“). Das ist eine Relation, die angibt, ob je zwei Typen als gleich betrachtet werden, d. h. äquivalent sind. Typäquivalenz kann auf Namensgleichheit (zwei Typen haben denselben Namen) oder Strukturgleichheit (zwei Typausdrücke haben dieselbe Struktur) beruhen. → 1.2.2.

Typerweiterung (engl. „type extension“). In Ada: Die Erweiterung eines erweiterbaren („tagged“) Verbundtyps um neue Verbundkomponenten. → 5.1.1.

Typfehler (engl. „type error“). Ein Typfehler liegt vor, wenn ein Ausdruck mehrere Typen hat, die nicht zusammenpassen. Zum Beispiel, wenn man einer Funktion vom Typ `Integer` → `Integer` das Argument `true` vom Typ `Boolean` übergibt, bekommt `true` die nicht zusammenpassenden Typen `Integer` und `Boolean`.

Typinferenz (engl. „type inference“). Typinferenz ist der Vorgang, bei dem aus einem Programm der allgemeinste Typ jedes Ausdrucks errechnet wird. → 4.2.

typisierte Sprache (engl. „typed language“). Eine Sprache ist typisiert wenn es in der Sprache Typen gibt; sonst ist sie **untypisiert**. Sie ist **vollständig typisiert** wenn jeder Ausdruck spätestens zur Laufzeit auf Typkonsistenz überprüft werden kann. Eine Sprache ist **stark typisiert** (engl. „strongly typed“) wenn alle möglichen Typfehler bereits während der Übersetzung erkannt werden können; sonst ist sie **schwach typisiert** (engl. „weakly typed“). Eine typisierte Sprache ist **statisch typisiert** (engl. „statically typed“) wenn der Typ jedes in einem Programm vorkommenden Ausdrucks bereits bei der Übersetzung bekannt ist; sonst heisst sie **dynamisch typisiert** (engl. „dynamically typed“). → 1.2.2.

typisierte Variable (engl. „typed variable“). Eine Variable ist typisiert, wenn auf ihr eine Typeinschränkung definiert ist. Der **deklarierte Typ** einer Variable ist jene Typeinschränkung, die explizit im Programmtext steht. Der **statische Typ** ist jene Typeinschränkung, die bei einer statischen Programmanalyse erwittelt wird. Der **dynamische Typ** ist der Typ des zur Laufzeit in der Variable enthaltenen Wertes.

Typkonsistenz (engl. „type consistency“). Das ist die Eigenschaft eines Programms, frei von Typfehlern zu sein. → 1.2.2.

Typqualifikation (engl. „type qualification“). Ada: Durch Typqualifikation wird der Typ eines Wertes angegeben; dadurch werden Namenskonflikte aufgelöst. → 3.1.1.

Typsystem (engl. „type system“). Ein Typsystem ist eine Menge von Regeln, die Typen zueinander und zu Ausdrücken in einer Programmiersprache in Beziehung setzen.

Typüberprüfung (engl. „type checking“). Das ist jener Vorgang, bei dem ein Programm auf Typkonsistenz überprüft wird. Die **dynamische Typüberprüfung** erfolgt zur Laufzeit; die **statische Typüberprüfung** zur Compilezeit. Eine Typüberprüfung ist **konservativ** wenn nur garantiert typkonsistente Programme akzeptiert werden; sie ist **optimistisch** wenn nur sicher nicht typkonsistente Programme zurückgeworfen werden. → 1.2.2.

Typumwandlung (engl. „type coercion“). Durch Typumwandlung wird ein Argument einer Routine automatisch in ein Argument jenes Typs umgewandelt, der von der Routine erwartet wird. → 1.2.3.

Überladen (engl. „overloading“). Ein und derselbe Name kann verschiedene Routinen beschreiben, die sich nur durch die Typen der Parameter unterscheiden. → 1.2.3.

Überschreiben (engl. „overwriting“). Das Ersetzen einer ererbten Operation durch eine neue Operation. → 5.1.1.

universelle Algebra (engl. „universal algebra“ oder „single-sorted algebra“). Das ist eine mathematische Struktur bestehend aus einer *Trägermenge* und einem System von *Operationen*. Die Operationen sind durch eine Menge von *Gesetzen* beschrieben. → 2.3.

Unterbereichstyp (in Ada engl. „subtype“; Vorsicht: „subtype“ steht üblicherweise für „Untertyp“). Ein Unterbereichstyp enthält zusätzliche Einschränkungen gegenüber dem Typ, dessen Unterbereichstyp er ist. Es können Wertebereiche und Genauigkeiten eingeschränkt bzw. bei Feldern Bereichsangaben und bei Verbunden Diskriminanten festgelegt werden. → 3.1.3.

Untertyp (engl. „subtype“). Ein Typ, der eine Teilmenge der Instanzen eines entsprechenden Obertyps beschreibt; der Untertyp ist spezifischer als der Obertyp. Für s ist ein Untertyp von t schreiben wir $s \leq t$.

Variante (engl. „variant“). Eine Variante ist eine Instanz eines Variantentyps. → 2.1.3.

Variantentyp (engl. „variant type“). Ein Variantentyp ist eine Menge indizierter Typen. Jede Instanz hat genau einen Typ in dieser Menge. → 2.1.3.

Varianz (engl. „variance“). Die Varianz eines Parametertyps einer Routine, die zu einem Objekt gehört, beschreibt, wie sich der Parametertyp

beim Ersetzen des Objekts durch ein Objekt eines entsprechenden Untertyps verhält. Bei **Kovarianz** (engl. „covariance“) wird der Parametertyp spezifischer, bei **Kontravarianz** (engl. „contravariance“) allgemeiner und bei **Invarianz** (engl. „invariance“) bleibt er unverändert. → 5.2.1.

Verband (engl. „lattice“). Ein Verband ist eine algebraische Struktur in der auf einer Menge eine reflexive, transitive und antisymmetrische Relation definiert ist, so dass je zwei Elemente der Menge ein eindeutig definiertes *Supremum* und *Infimum* haben. → 4.1.1.

Verbund (engl. „record“). Ein Verbund ist eine Instanz eines Verbundtyps. → 2.1.3.

Verbundtyp (engl. „record type“). Ein Verbundtyp ist eine Menge indizierter Typen. Jede Instanz des Verbundtyps ist eine Menge indizierter Instanzen der entsprechend indizierten Typen im Verbundtyp. → 2.1.3.

Vererbung (engl. „inheritance“). Vererbung ist ein Mechanismus, der die Erstellung von Programmteilen durch gezielte Modifikation bereits existierender Programmteile unterstützt. → 5.2.2.

Vererbungsanomalie (engl. „inheritance anomaly“). Das ist die Bezeichnung eines durch die Notwendigkeit von Synchronisationscode verursachten Problems, welches die Vererbung in nebenläufigen objektorientierten Sprachen behindert. → 5.4.2.

Vorbedingung (engl. „precondition“). Das ist eine Zusicherung, die vor der Ausführung einer bestimmten Operation erfüllt sein muss. → 5.1.2.

Wert (engl. „value“). Ein Wert ist ein Symbol oder eine aus Werten konstruierte Struktur, das bzw. die keinen inneren Zustand besitzt. Zwei Werte sind genau dann gleich, wenn sie dasselbe Symbol bzw. dieselbe Struktur darstellen.

Zusicherung (engl. „assertion“). Eine Zusicherung ist eine formal festgelegte Bedingung, die bei der Ausführung eines festgelegten Programmstücks erfüllt sein muss. → 5.1.2.