

Technische Universität München

Proseminar
Softwaredesaster
und wie man sie verhindern kann

Typsysteme und Dimensionstypsysteme

Dennis Pagano

Kursleiter: Prof. Tobias Nipkow
Betreuer: Norbert Schirmer
27.11.2002

Inhaltsverzeichnis

1	Laufzeitfehler und sichere Sprachen	3
1.1	Typfehler	3
1.2	Dimensionsfehler.....	4
2	Fehlervermeidung.....	4
2.1	Typsysteme	5
2.1.1	Typen	5
2.1.2	Fehlervermeidung mit Hilfe von Typen	5
2.1.3	Typregeln.....	6
2.1.4	Der Typcheck	8
2.1.5	Typinferenz	9
2.1.6	Übersicht über Typsysteme	9
2.1.7	Typisierte und untypisierte Sprachen.....	9
2.2	Dimensionstypen	10
2.2.1	Dimension, Einheit und Repräsentation.....	10
2.2.2	Vereinbarkeit mit den konventionellen Typkonzepten.....	11
3	Dimensionstypsysteme.....	11
3.1	Monomorphe und polymorphe Dimensionstypsysteme	11
3.2	Typinferenz	12
3.3	Wichtige Funktionen und formale Dimensionsausdrücke.....	12
3.4	Wichtige Regeln	13
3.5	Ein Dimensionstypsystem nach Andrew Kennedy	13
3.5.1	Das Problem der abhängigen Typen.....	14
3.5.2	Erweiterung um mehrfache Einheiten.....	14
4	Fazit.....	14
	Literatur.....	15

1 Laufzeitfehler und sichere Sprachen

Laufzeitfehler stellen eine hohe Gefahr in der Programmierung dar, denn ihre Tücke liegt vor allem darin, dass sie im Allgemeinen nicht vor Ausführung des Programms bemerkt werden und oft nicht einmal während der Laufzeit. Dies führt zu unvorhersehbaren Konsequenzen, vor allem bei portablen Programmen, da hier zusätzlich das Umfeld variiert.

Man kann grundsätzlich zwei verschiedene Arten von Laufzeitfehlern unterscheiden: Auf der einen Seite solche, die zu einem augenblicklichen Abbruch des ausgeführten Programms führen, und auf der anderen diejenigen Laufzeitfehler, die unbemerkt bleiben und willkürliche, unvorhersehbare Ereignisse hervorrufen. Fehler der ersten Art bezeichnet man als *abgefangen (trapped)*, die anderen als *nicht abgefangen (untrapped)*. Ein Beispiel für einen nicht abgefangenen Fehler wäre ein unbeabsichtigter Zugriff auf eine existierende Adresse im Speicher, wie etwa ein Zugriff auf Daten außerhalb der Grenzen eines Arrays, falls eine entsprechende Prüfung zur Laufzeit nicht durchgeführt wird (*buffer overrun*).

Ein anderes Beispiel für einen unabgefangenen Fehler ist ein Sprung an die falsche Adresse. Der Speicherinhalt an der angesprungenen Stelle wird als Instruktion interpretiert, egal, ob er wirklich eine Instruktion darstellt oder nicht. Dieser Laufzeitfehler kann ebenfalls eine Zeit lang völlig unbemerkt bleiben.

Abgefangene Fehler können etwa Division durch Null oder Adressierung einer unzulässigen Adresse sein.

Man bezeichnet ein Programmfragment als *sicher*, wenn es keine unabgefangenen Fehler hervorruft. Dementsprechend nennt man eine Sprache, in der alle Programmfragmente sicher sind, eine *sichere Sprache*.

Sichere Sprachen bewahren uns also vor den schlimmsten Fällen von Laufzeitfehlern, nämlich vor denen, die wir nicht bemerken.

1.1 Typfehler

Unter einem Typfehler versteht man das Auftreten von bestimmten Typen in einem für diese nicht vorgesehenen Umfeld. Ein einfaches Beispiel wäre der Versuch, eine Integer-Zahl zu einem String zu addieren. Die Addition ist für diese beiden Typen standardmäßig nicht definiert. Sie sieht es vor, entweder zwei Integer zu addieren, oder – falls durch den überladenen Additions-Operator realisiert – zwei Zeichenketten zu konkatenieren.

Treten jedoch beide Typen auf, so kollidiert der eine Typ mit dem anderen.

Sicherlich kann ein solcher Typfehler bereits vom Compiler abgefangen werden, doch dazu später.

Im Grunde repräsentiert diese Art von Kollision einen Spezialfall einer Fehlinterpretation eines bestimmten Bereichs im Speicher. So könnte beispielsweise bei einer expliziten Typumwandlung (casting) ein ursprünglich als Integer zu interpretierender Bereich in einen Zeiger umgewandelt werden. Ein solcher „wilder“ Zeiger kann großen Schaden anrichten und führt meist zu unvorhersehbaren Ereignissen.

Wir werden uns später sehr genau mit Typfehlern und deren Vermeidung auseinandersetzen.

1.2 Dimensionsfehler

Entfernen wir uns für einen kurzen Augenblick von Programmiersprachen.

Ähnlich wie bei den Typfehlern - nur in einem anderen Zusammenhang - können Fehler entstehen, wenn Dimensionen unerlaubt in einem bestimmten Kontext auftreten oder fehlen.

So kann man beispielsweise in der Physik eine Gleichung sofort falsifizieren, wenn die Dimensionen auf der linken und auf der rechten Seite nicht übereinstimmen.

Es gibt in der Tat verschiedene Gesetze, was mit dimensionsbehafteten Werten erlaubt oder falsch ist.

Dementsprechend dürfen Werte mit verschiedenen Dimensionen niemals addiert oder voneinander subtrahiert werden, wohingegen deren Multiplikation oder Division auch die Multiplikation bzw. Division der beiden Dimensionen zur Folge hat.

Die Summe zweier Werte mit den Dimensionen Geschwindigkeit und Zeit ist also ein Dimensionsfehler, wohingegen deren Produkt einen Wert mit der Dimension Länge ergibt.

Werte aus der Naturwissenschaft sind zumeist mit bestimmten Dimensionen behaftet.

Muss man nun solche Werte in Programmen verwenden, so geht dadurch die Dimension der Werte bei herkömmlichen Programmen verloren.

Dadurch ist - wenn wir uns nun den Programmiersprachen wieder zuwenden – keinerlei Sicherheit für die dimensionale Korrektheit von Werten innerhalb der Programme mehr gegeben. Allein die numerische Richtigkeit bleibt bei einer derartigen Verwendung von dimensionsbehafteten Werten sicher.

Desaster können bekanntermaßen die Folge sein, wenn Dimensionen bzw. Einheiten kollidieren.

2 Fehlervermeidung

Es stellt sich nun die Frage, wie man solche Fehler am besten vermeiden kann.

Dafür gibt es zunächst zwei grundlegend verschiedene Ansätze.

Auf der einen Seite kann das Programm selbst während der Laufzeit Prüfungen durchführen, die Ausdrücke immer wieder verifizieren, um so nicht Gefahr zu laufen, dass ungültige Ausdrücke unbemerkt auftreten. Hier spricht man von *dynamischen* Checks.

Auf der anderen Seite sind Programme, die Laufzeitfehler verursachen, formal gesehen nur eine Teilmenge aller gültigen Programme. Fehler könnten also auch verhindert werden, indem die Menge aller gültigen Programme eingeschränkt wird.

Dies geschieht sinngemäß bereits vor der Ausführung, man nennt diese Art Fehler zu verhindern deshalb auch *statisch*.

Oft wird die dynamische Variante für zu teuer im Sinne der Leistungsstärke der hervorgebrachten Programme befunden. Jedoch bekommt man selbst bei statisch verifizierenden Sprachen die Sicherheit der Programme nicht umsonst: Tests, die auf die Wahrung von Arraygrenzen überprüfen beispielsweise, können nicht vollständig zur Compilezeit übernommen werden. Die Tatsache, dass eine Sprache statisch verifiziert bedeutet noch lange nicht, dass eine Programmausführung völlig unbeaufsichtigt vor sich gehen kann. Sicherlich ist es jedoch auch sinnvoll, sowohl statisch als auch dynamisch zu überprüfen, wie es etwa bei Java der Fall ist.

Wir beschäftigen uns im Folgenden mit einem speziellen Mittel gegen Laufzeitfehler, das vor allem bei den statischen Checks Anwendung findet, den Typsystemen.

2.1 Typsysteme

Typsysteme stellen, ausgehend von der Idee der Typen, eine Reihe von Regeln zur Verfügung, anhand derer Ausdrücke jeglicher Art innerhalb von Programmen auf Korrektheit überprüft werden können. Bei einer Überprüfung wird ein Algorithmus verwendet, um die Einhaltung der Regeln zu gewährleisten.

Hauptgegenstand von Typsystemen sind also offensichtlich Typen, weshalb es erforderlich ist, dass wir uns zunächst ein Bild von diesem abstrakten Begriff machen. Ferner werden wir Charakteristika von Typsystemen benennen und erklären, und uns mit ihren Auswirkungen auf Programmiersprachen befassen.

2.1.1 Typen

Beim Programmieren arbeiten wir im Allgemeinen mit Werten. Diese sind zunächst für sich gesehen nicht mit einer bestimmten Bedeutung behaftet, sondern einfach Träger einer Information. Um sinnvoll programmieren zu können, messen wir ihnen jedoch üblicherweise bestimmte Bedeutungen zu. Dies wird besonders dann offensichtlich, wenn wir Variablen für unsere Werte vorsehen, die wir gewöhnlich der Bedeutung entsprechend benennen.

Zunächst gibt es viele mögliche Bedeutungen die wir uns vorstellen können.

Nun kann man jedoch sinnvollerweise verschiedene Bedeutungen kategorisieren.

Es macht beispielsweise grundlegend Sinn, Zahlen als eine Kategorie zu betrachten.

Auf diese Weise erhält man gewisse Grundkategorien für die Werte in Programmen.

Typen stellen nun eine Begrenzung der möglichen Werte dar. Dabei zwingt ein Typ Werte in die Kategorie, die durch ihn induziert wird, weshalb wir oft Typ und Kategorie gleichsetzen. Im eigentlichen Sinn jedoch stellen Typen nur Grenzen dar, die für die möglichen Werte von Variablen oder Funktionen gelten.

So begrenzt zum Beispiel der Typ *byte* die Werte einer Variable dieses Typs auf ganze Zahlen aus dem Intervall $[-128,127]$.

2.1.2 Fehlervermeidung mit Hilfe von Typen

Durch die Klassifikation bestimmter Variablen mit Hilfe der ihnen zugewiesenen Typen, erhalten wir sowohl Grenzen für zulässige Bereiche innerhalb der möglichen Werte als auch Unterscheidungskriterien für verschieden typisierte Variablen. Somit können Operationen oder Funktionen auf bestimmte Typen festgelegt werden. Durch die Einschränkung der gültigen Werte sind wir also auch imstande, die gültigen Operationen und Funktionen einzuschränken, indem wir Operanden, Parameter und Ergebnisse typisieren.

Damit ist es uns folglich möglich, gewisse Regeln aufzustellen, die das Auftreten von unerlaubten Ausdrücken verhindern können.

Wir erhalten somit die Möglichkeit, bestimmte Ausdrücke als korrekt durchgehen zu lassen, und andere als falsch im Sinne unserer Ideen zu blockieren. Dabei soll explizit

darauf geachtet werden, sinnvolle Regeln zu finden, also insbesondere Regeln, mit denen Laufzeitfehler verhindert werden.

Insgesamt helfen uns Typen also dabei, die Menge der gültigen Programme unseren Wünschen entsprechend einzuschränken. Dieser Standpunkt ist natürlich aus der Sichtweise des Sprachen-Designers zu betrachten, denn der Programmierer muss sich letztendlich dieser Sichtweise beugen.

Die Standard-Typen, die wir etwa in Java benutzen, bilden für sich gesehen sinnvoll voneinander abgegrenzte logische Einheiten, wodurch es uns intuitiv leicht fällt, den richtigen Typ für einen bestimmten Zweck zu finden. Zusätzlich wirken sie als Schutz vor Fehlinterpretationen des Speichers und garantieren definitionsgemäße Anwendung von Funktionen und Operationen.

2.1.3 Typregeln

Der Hauptbestandteil eines Typsystems sind seine Regeln. Diese werden mit Hilfe eines Formalismus festgelegt, den wir an dieser Stelle einführen.

Zunächst werden einige formale Ausdrücke beschrieben, die man logische Urteile nennt.

Ein typisches logisches Urteil hat die Form:

$$\Gamma \vdash \beta \qquad \text{Aus } \Gamma \text{ folgt } \beta$$

wobei β eine Behauptung ist und die freien Variablen von β in Γ deklariert sind.

Wir sagen in diesem Fall, Γ hat β zur Folge.

Γ bezeichnet an dieser Stelle ein *statisches Typumfeld*. In einem Typumfeld sind alle freien Variablen deklariert, also handelt es sich dabei beispielsweise um eine geordnete Liste von verschiedenen Variablen und deren Typen in der Form $\emptyset, x_1: A_1, \dots, x_n: A_n$.

An dieser Stelle wird das leere Umfeld durch das Symbol \emptyset dargestellt.

Da das leere Umfeld Teilmenge jedes Umfeldes ist, lassen wir es in dieser Notation auch oft weg. Die formale Gestalt der Behauptung β ist von Urteil zu Urteil verschieden, jedoch sind alle freien Variablen stets in Γ zu deklarieren.

Für unsere Belange ist im Augenblick das Typurteil am wichtigsten, welches besagt, dass ein Term M vom Typ A ist, wenn die freien Variablen von M wie in der statischen Typumgebung deklariert sind. Ein Typurteil hat die Form:

$$\Gamma \vdash M:A \qquad M \text{ hat Typ } A \text{ in } \Gamma$$

Beispiele hierfür wären:

$$\emptyset \vdash true: Bool \qquad true \text{ hat immer den Typ } Bool$$

$$x: Nat \vdash x + 1: Nat \qquad x + 1 \text{ hat Typ } Nat, \text{ wenn } x \text{ Typ } Nat \text{ hat}$$

Jedes Urteil kann dabei entweder als *gültig* oder als *ungültig* betrachtet werden.

Die obigen Beispiele sind alle gültig. Ein Beispiel für ein ungültiges Urteil wäre:

$$\Gamma \vdash true: Nat \qquad true \text{ hat nicht den Typ } Nat$$

Ein weiteres Urteil, das oft von großem Nutzen ist, ist die Behauptung, dass die gewählte Umgebung wohlgeformt ist:

$$\Gamma \vdash \diamond \qquad \Gamma \text{ ist wohlgeformt, also korrekt konstruiert}$$

Mit Hilfe dieses Logikkalküls können wir also bestimmte Forderungen formulieren und so die Typregeln eines Typsystems aufbauen.

Typregeln fordern dabei die Gültigkeit bestimmter Urteile, ausgehend von anderen Urteilen, von denen bereits bekannt ist, dass diese gültig sind. Dieser Prozess beginnt dann schließlich bei einem intrinsisch richtigen Urteil, nämlich meistens bei:

$$\emptyset \vdash \diamond \qquad \text{Das leere Umfeld ist wohlgeformt}$$

Um eine Typregel auszudrücken benötigen wir also mehrere Urteile, die wir schlussendlich aufeinander aufbauen lassen. Somit hat eine allgemeine Typregel die Form:

$$\frac{\Gamma_1 \vdash \beta_1 \dots \Gamma_n \vdash \beta_n}{\Gamma \vdash \beta} \qquad \text{Allgemeine Form einer Typregel}$$

Jede Typregel besteht formal aus einer Anzahl von *Prämissen* $\Gamma_i \vdash \beta_i$, die oberhalb einer horizontalen Linie geschrieben werden, und einer einzigen *Schlussfolgerung* $\Gamma \vdash \beta$ unterhalb der Linie.

Dabei bedeutet die Schreibweise: Wenn alle Prämissen erfüllt sind, so ist auch die Schlussfolgerung korrekt.

Wichtig ist außerdem, dass die Menge der Prämissen auch leer sein darf, was bei intrinsisch richtigen Regeln der Fall sein muss. Zusätzlich bekommt jede Typregel einen Namen und manchmal werden auch Anmerkungen oder Abkürzungen mit notiert.

Sehen wir uns nun einfache Beispiele für Typregeln an.

$$\frac{(\text{Val } n) \ (n = 0, 1, \dots)}{\Gamma \vdash \diamond} \\ \Gamma \vdash n: \text{Nat}$$

Die Typregel (Val n) besagt, dass jedes Zahlzeichen in einer beliebigen, wohlgeformten Umgebung Γ ein Ausdruck vom Typ *Nat* ist.

$$\frac{(\text{Val } +) \quad \Gamma \vdash M: \text{Nat} \quad \Gamma \vdash N: \text{Nat}}{\Gamma \vdash M + N: \text{Nat}}$$

Die Typregel (Val $+$) sagt aus, dass zwei Ausdrücke M und N , die natürliche Zahlen darstellen, zu einem größeren Ausdruck $M + N$ kombiniert werden können, der damit ebenfalls für eine natürliche Zahl steht.

(Env \emptyset)

$$\frac{}{\emptyset \vdash \diamond}$$

Die intrinsische Regel (Env \emptyset) besagt, dass das leere Typumfeld stets wohlgeformt ist.

Eine Menge von Typregeln bezeichnet man als ein (formales) Typsystem.
Damit haben wir den wichtigsten Bestandteil der Typsysteme erklärt.

2.1.4 Der Typcheck

Was geschieht jedoch nun während einer Prüfung auf mögliche Laufzeitfehler, oder besser gefragt, was macht eine Sprache mit ihrem Typsystem?

Sinnvollerweise wird ein Typsystem dazu verwendet, um anhand seiner Typregeln Ausdrücke, Funktionen und Operationen zu verifizieren.

Diesen Vorgang nennt man einen Typcheck.

Den Typcheck selbst wiederum muss natürlich ein Algorithmus durchführen, diesen bezeichnet man aufgrunddessen als Typchecker.

Um zu verstehen, was ein Typchecker genau leistet, müssen wir selbst einen Typcheck an einem Beispiel vornehmen. Sehen wir uns dazu ein Programmfragment an:

$$y = (fx) + 1$$

Wir gehen aufgrund der (hier nicht aufgeführten) Funktionsdeklaration und der Deklaration der Variablen x und y von folgendem Kontext aus:

$$\Gamma = (x: \text{Nat}, y: \text{Nat}, f: \text{Nat} \rightarrow \text{Nat})$$

Also können wir ein Urteil aufstellen, das es zu überprüfen gilt:

$$\Gamma = (x: \text{Nat}, y: \text{Nat}, f: \text{Nat} \rightarrow \text{Nat}) \vdash (fx) + 1: \text{Nat}$$

Wir wollen hier mit Hilfe unserer Typregeln zeigen, dass der Funktionswert der Funktion f an der Stelle x zuzüglich des festen Wertes 1 tatsächlich vom geforderten Typ Nat ist.

Ein Typchecker wird nun folgendes Konstrukt von sukzessive aufeinander aufbauenden Urteilen anhand der ihm vorliegenden Regeln bilden:

$$\frac{\Gamma \vdash x: \text{Nat} \quad \Gamma \vdash f: \text{Nat} \rightarrow \text{Nat}}{\Gamma \vdash (fx): \text{Nat}} \quad \text{mit (Val +)}$$

$$\frac{\Gamma \vdash (fx): \text{Nat}}{\Gamma \vdash (fx) + 1: \text{Nat}}$$

Dabei ist zu beachten, dass wir im letzten Schritt die Typregel (Val +) angewendet haben, die wir weiter oben eingeführt haben.

Wir haben also bewiesen, dass die Zuweisung an die Variable y vom Typ Nat gültig ist. Ein Typchecker führt also formale Beweise von Ausdrücken aus, die er anhand der Typregeln des Typsystems zeigen kann. Somit erkennt er rechtzeitig ungültige Programme und kann je nach Check den Compilervorgang verweigern (statisch) oder das laufende Programm abbrechen (dynamisch).

2.1.5 Typinferenz

Ein weiterer wichtiger Begriff im Zusammenhang mit Typsystemen ist die Typinferenz. Darunter versteht man das Folgern eines bestimmten Typs für einen gegebenen Ausdruck, obwohl der Typ nicht explizit angegeben ist. Dabei wird ein weiteres Mal in hohem Maße Gebrauch von den Typregeln gemacht.

Typinferenz ist bei manchen Sprachen, wie etwa bei Gofer, ein fester Bestandteil. Dort muss man Typen nicht explizit zu den Variablen- oder Funktionsdeklarationen angeben. Stattdessen kann ein Algorithmus auf die richtigen Typen schließen. Kehren wir zum besseren Verständnis zu unserem obigen Beispiel zurück.

$$y = (fx) + 1$$

Diesmal gehen wir aber davon aus, dass y nicht deklariert wurde, und der Kontext somit

$$\Gamma = (x: \text{Nat}, f: \text{Nat} \rightarrow \text{Nat})$$

ist. Wir müssen nun also den Typ erschließen, der für den Ausdruck in Frage kommt:

$$\Gamma = (x: \text{Nat}, f: \text{Nat} \rightarrow \text{Nat}) \vdash (fx) + 1: ?$$

Dazu können wir exakt dieselbe Schlussfolgerung wie oben verwenden, woraus sich ergibt, dass y eine natürliche Zahl sein muss.

Somit ist die Typinferenz sehr stark vom jeweiligen Typsystem abhängig. Ein entsprechender Algorithmus kann leicht, schwer oder überhaupt nicht zu finden sein, je nach Typsystem. Und selbst wenn ein Algorithmus gefunden werden kann, so kann dieser sehr effizient oder aber hoffnungslos langsam sein.

Gute Typinferenz zeichnet sich dadurch aus, dass sie immer den allgemeinsten Typ für einen untypisierten Ausdruck findet.

2.1.6 Übersicht über Typsysteme

Typsysteme bestehen aus einer Menge von Typregeln.

Dabei ist das Typsystem völlig unabhängig von Typcheck-Algorithmen zu sehen:

Typsysteme sind eine Eigenschaft von Programmiersprachen und gehören in die Rubrik Sprachendefinition, wohingegen Typchecker zu den Compilern gehören.

Es ist sogar so, dass verschiedene Compiler verschiedene Typchecker für ein und dasselbe Typsystem benutzen können.

Jedes Typsystem sollte beweisbar sein, das bedeutet, es sollte möglich sein, einen Typchecker zu formulieren, der für die Regeleinhaltung der Programme garantieren kann, denn der Sinn eines Typsystems ist nach wie vor die Vermeidung von Laufzeitfehlern.

Außerdem sollte ein Typsystem durchschaubar sein, damit ein Programmierer sich leicht an die Regeln halten bzw. bei Fehlern schnell die Ursache erkennen kann.

2.1.7 Typisierte und untypisierte Sprachen

Was hat das Vorhandensein eines Typsystems nun für Auswirkungen auf eine Sprache?

Die Tatsache, dass eine Sprache über ein Typsystem verfügt, macht sie zu einer *typisierten Sprache*.

Auf der anderen Seite haben *untypisierte Sprachen* keine Typen bzw. nur einen Universaltyp, der alle Werte beinhaltet.

Die meisten untypisierten Sprachen sind notwendigerweise vollkommen sicher. Denn das Programmieren wäre zu frustrierend, müsste man sowohl auf statische als auch auf dynamische Checks verzichten, die Schutz vor Fehlern bieten.

Assembler-Sprachen gehören zu der leidvollen Kategorie von Sprachen, die sowohl untypisiert als auch unsicher sind.

	Typisiert	Untypisiert
Sicher	Gofer	LISP
Unsicher	C	Assembler

Tabelle 1. Typisierte und untypisierte Sprachen.

Auch C wird zu den unsicheren Sprachen gezählt, obwohl Typen vorhanden sind, auf deren Korrektheit auch getestet wird. Jedoch existieren in ebenso gewisse Sprachkonstrukte, mit denen man das Typsystem zur Laufzeit umgehen kann. Dabei seien vor allem ungeprüfte Typumwandlungen erwähnt, die so genannten Typecasts. Typisierte Sprachen haben den Vorteil, leistungsstärkere Programme hervorbringen zu können, da Typinformationen zur Compilezeit verfügbar sind.

Das Typsystem einer typisierten Sprache gibt zudem Aufschluss über ihre Sicherheit: Eine typisierte Sprache heißt *typsicher*, wenn Kraft ihres Typsystems keine unabgefangenen Typfehler passieren können.

2.2 Dimensionstypen

Bevor wir uns mit speziellen Eigenschaften von Dimensionstypen beschäftigen können, müssen wir diesen Begriff zunächst spezifizieren. Dabei lässt sich schon vermuten, dass es sich hierbei um eine Verknüpfung von Dimensionen auf der einen Seite und Typen auf der anderen Seite handelt. In der Tat lassen sich Dimensionstypen in unsere Definition von allgemeinen Typen einfügen.

Bemessen wir einem Wert eine Bedeutung zu, die aus der Naturwissenschaft kommt, so trägt diese zumeist zusätzlich Informationen über die Dimension des Ausdrucks.

Der Wert *25m* beispielsweise trägt (zunächst) mehr Information als der Wert *25*, denn wir wissen dabei zusätzlich, dass es sich bei dem numerischen Wert *25* um eine *Länge* handelt und dass diese in der Einheit *Meter* bemessen ist. Wir fassen nun die Dimensionen als neue Typen auf. Variablen von verschiedenen Dimensionstypen können so gegeneinander abgegrenzt werden und die ursprüngliche Idee der Einschränkung von Operationen oder Funktionen auf bestimmte Typen der Operanden bzw. Parameter kann analog fortgeführt werden.

2.2.1 Dimension, Einheit und Repräsentation

Dimension, Einheit und Repräsentation sind drei verschiedene Begriffe, die jedoch eng zusammenhängen.

Eine *Dimension* beschreibt eine bestimmte Eigenschaft und Interpretation einer Größe. Größen mit derselben Dimension stellen dieselbe Art von Eigenschaft dar, sei es Masse, Kraft, Geschwindigkeit oder sonstiges.

Zwei Größen mit verschiedenen *Einheiten*, aber derselben Dimension unterscheiden sich lediglich um einen Skalierungsfaktor. Ein in nautischen Meilen gemessener Wert ist etwa 1,852-mal so groß wie derselbe Wert in Kilometer, beide jedoch haben die Dimension

Länge. Diese Einheiten sind noch recht leicht umzurechnen, schwieriger sind beispielsweise Temperaturen in Grad Celsius und Fahrenheit.

Basisdimensionen sind diejenigen Dimensionen, die nicht durch andere Dimensionen ausgedrückt werden können. Das *International System of Units* (SI) definiert deren sieben, nämlich *Länge, Masse, Zeit, elektrischer Strom, thermodynamische Temperatur, Stoffmenge* und *Lichtstärke*.

Dimensionen, die sich durch bereits existierende Dimensionen ausdrücken lassen, nennt man *abgeleitet*.

Ein weiterer wichtiger Begriff ist die *Repräsentation*, also der Typ, der verwendet wird um den numerischen Wert einer Größe zu speichern. Elektrische Größen werden dabei oft durch komplexe Zahlen repräsentiert, wohingegen Längen beispielsweise meistens durch reelle Zahlen dargestellt werden.

Außerdem sei noch erwähnt, dass dimensionslose Größen, die genauso häufig in der Wissenschaft vorkommen (z.B. als Skalierungsfaktor), in unserem Sinn ebenfalls durch eine Dimension dargestellt werden können, nämlich durch die Einheitsdimension.

2.2.2 Vereinbarkeit mit den konventionellen Typkonzepten

Bei den konventionellen Typkonzepten fungieren Typsysteme als Garant für die Korrektheit von Werten, Operationen und Funktionen, indem sie gegebene Ausdrücke anhand ihrer Regeln überprüfen.

Auf ähnliche Weise müssen naturwissenschaftliche Ausdrücke dimensional konsistent sein. Aus den anfangs erwähnten Gesetzen für dimensionsbehaftete Werte können nun entsprechende Regeln formuliert werden, die ein dimensionsbehaftetes Typsystem bilden sollen. Dazu verwenden wir Dimensionstypen. Ein Typchecker kann dann für die dimensionale Korrektheit von gegebenen Dimensionsausdrücken garantieren.

Damit erhalten wir ein Konzept, das ohne Probleme im Konzept der Typsysteme aufgeht. Dies führt uns zum Begriff der Dimensionstypsysteme.

3 Dimensionstypsysteme

Wie wir gesehen haben, ist es uns möglich, unsere Anforderungen an dimensionale Korrektheit in ein Typsystem zu integrieren. Regeln können wir typsystemkonform etablieren, hier sind jedoch einige Spezialfälle zu betrachten, welchen wir uns später widmen. Der Typchecker kann damit die an ihn gestellten Anforderungen bewältigen. Als nächstes müssen wir eine weitere wichtige Entscheidung für das Dimensionstypsystem treffen, nämlich ob es monomorpher oder polymorpher Gestalt sein soll.

3.1 Monomorphe und polymorphe Dimensionstypsysteme

Bei einem *monomorphen* Typsystem sind alle Typen explizit festzulegen, wohingegen ein *polymorphes* Typsystem die Verwendung von Typvariablen erlaubt.

Welche Art ist aber für Dimensionstypsysteme geeignet?

Für nichttriviale Programme wollen wir allgemein verwendbare Funktionen schreiben können, die über einer ganzen Reihe von Dimensionen arbeiten. Aber sogar etwas derart einfaches wie eine Quadrierungsfunktion könnten wir mit einem monomorphen

Dimensionstypsystem nicht realisieren, stattdessen müssten wir für jede Dimension eine eigene Quadrierungsfunktion schreiben. In einem polymorphen Dimensionstypsystem können wir Typvariablen benutzen, um auszudrücken, dass die Funktion die Dimension ihres Arguments quadriert – und zwar für jede Dimension. Deshalb macht es grundlegend Sinn, ein polymorphes Dimensionstypsystem zu erarbeiten.

3.2 Typinferenz

Eine wünschenswerte Eigenschaft eines Dimensionstypsystems wäre außerdem die Möglichkeit der Typinferenz. Bei Operationen und Funktionen wollen wir nicht jedes Mal den Dimensionstyp mit angeben. Besonders bei polymorphen Funktionen müssten wir stets die Dimensionstypen mitführen, was natürlich nicht angenehm wäre. Allerdings bekommt man einen Typinferenz-Algorithmus wie schon besprochen nicht umsonst, wenn es überhaupt einen gibt. Eine weitere Aufgabe beim Erstellen eines Dimensionstypsystems kann es also sein, zu beweisen, dass es einen solchen Algorithmus gibt bzw. gleich einen entsprechenden Algorithmus zu finden. Grundsätzlich jedoch ist Typinferenz auch bei Dimensionstypsystemen möglich.

3.3 Wichtige Funktionen und formale Dimensionsausdrücke

An dieser Stelle gilt es einige grundlegend wichtige Funktionen und Operationen zu betrachten. Dabei müssen wir zunächst eine Notation für Dimensionsausdrücke festlegen: Wir schreiben Dimensionen innerhalb von eckigen Klammern wie z.B. $[M]$ für Masse. Außerdem verwenden wir als Repräsentation einen Datentyp *real*. Diese Notation ist willkürlich und muss je nach Sprache angepasst werden, in Gofer bezeichnen eckige Klammern beispielsweise schon Listen und sind daher dort nicht brauchbar. Hier geht es jedoch nicht um eine spezielle Implementierung, sondern nur um die formale Gestalt.

Wir beginnen unser System mit einer Menge von Basisdimensionen, beispielsweise den sieben Grunddimensionen der SI.

Um polymorphe Funktionen schreiben zu können, benötigen wir so genannte Dimensionstypvariablen. Das sind Variablen, deren Werte Dimensionstypen sind, die also als Platzhalter in Funktionsdeklarationen fungieren sollen. Dabei ist zu beachten, dass ein Variablenname eindeutig ist, dass also derselbe Variablenname an zwei verschiedenen Stellen denselben Dimensionstyp repräsentiert. Wir verwenden im Folgenden als Dimensionstypvariablen die Bezeichner d , d_1 und d_2 , um nicht in Konflikt mit anderen Variablen zu geraten. Die Einheitsdimension für dimensionslose Größen stellen wir durch $\mathbf{1}$ dar.

Betrachten wir nun die gewöhnlichen arithmetischen Operationen mit dimensionsbehafteten Operanden. Deren Typdeklarationen müssen wie folgt lauten:

$+$, $-$	$: [d] \text{ real} \times [d] \text{ real} \rightarrow [d] \text{ real}$
$*$	$: [d_1] \text{ real} \times [d_2] \text{ real} \rightarrow [d_1 d_2] \text{ real}$
$/$	$: [d_1] \text{ real} \times [d_2] \text{ real} \rightarrow [d_1 d_2^{-1}] \text{ real}$
sqrt	$: [d^2] \text{ real} \rightarrow [d] \text{ real}$
exp, ln, sin, cos, tan	$: [\mathbf{1}] \text{ real} \rightarrow [\mathbf{1}] \text{ real}$

Weiterhin benötigen wir eine polymorphe Null:

zero : [d] real

Ohne eine polymorphe Null könnten wir beispielsweise keine Absolutbetragsfunktion formulieren, wir könnten nicht einmal das Vorzeichen eines dimensionsbehafteten Ausdrucks feststellen. Dies ist mit Hilfe der polymorphen Null möglich, sehen wir uns etwa folgende Implementierung einer Absolutbetragsfunktion an:

abs x = if x < zero then zero - x else x

Diese hat den Typ $[d] \text{ real} \rightarrow [d] \text{ real}$.

An dieser Stelle können wir schließlich formale Dimensionsausdrücke definieren.

Ein Dimensionsausdruck ist definiert durch:

$\delta ::= d \mid \mathbf{B} \mid \delta \cdot \delta \mid \delta^{-1} \mid \mathbf{1}$

wobei \mathbf{B} eine beliebige Basisdimension und d eine beliebige Dimensionsvariable ist. Anstelle von $d_1 \cdot d_2$ schreiben wir auch $d_1 d_2$.

3.4 Wichtige Regeln

Beim Umgang mit Dimensionsausdrücken sind zusätzlich einige wichtige Regeln zu beachten, deren Herkunft in der Naturwissenschaft liegt.

Dies wären im Einzelnen:

$\delta_1 \delta_2 =_{\mathbf{D}} \delta_2 \delta_1$: Kommutativität
$(\delta_1 \delta_2) \delta_3 =_{\mathbf{D}} \delta_1 (\delta_2 \delta_3)$: Assoziativität
$\mathbf{1} \cdot \delta =_{\mathbf{D}} \delta$: Identität
$\delta \delta^{-1} =_{\mathbf{D}} \mathbf{1}$: Inverse Elemente

Bezüglich der Dimensionsmultiplikation bilden Dimensionsausdrücke also eine abelsche Gruppe.

Die Kommutativität erschwert den Typcheck natürlich, da zusätzlich jede mögliche Anordnung der gegebenen Dimensionen überprüft werden muss.

3.5 Ein Dimensionstypsystem nach Andrew Kennedy

Andrew Kennedy von der University of Cambridge hat ein Dimensionstypsystem beschrieben, das alle Anforderungen, die wir besprochen haben erfüllt.

Es ist polymorph und verfügt über alle Dimensionstypregeln, die das Rechnen mit Dimensionen notwendig macht. Zusätzlich hat Kennedy bewiesen, dass Typcheck und Typinferenz in seinem Dimensionstypsystem möglich und vor allem anwendbar und implementierbar sind. Der Typinferenz-Algorithmus, den Kennedy zusätzlich angibt, berücksichtigt bereits alle wichtigen Gesetze, inklusive Kommutativität, und findet ferner stets den allgemeinsten Dimensionstyp, falls es diesen gibt.

Dieses Dimensionstypsystem wurde bereits erfolgreich implementiert.

3.5.1 Das Problem der abhängigen Typen

Sehen wir uns eine Funktion an, die Werte natürlichzahlig potenziert:

$$\begin{aligned} \text{power } 0 \ x &= 1.0 \\ \text{power } n \ x &= x * \text{power } (n - 1) \ x \end{aligned}$$

Da die Dimension des Ergebnisses von einem natürlichzahligen Wert abhängig ist, kann Kennedys System diesem keinen besseren Typ als den dimensionslosen Typ geben:

$$int \rightarrow [1] \ real \rightarrow [1] \ real$$

Er selbst erkennt die Einschränkung, versucht diese jedoch wettzumachen, indem er erklärt, dass variable Exponenten in der Naturwissenschaft wirklich selten sind (Ausgenommen Potenzreihen, jedoch sind diese dimensionslos).

3.5.2 Erweiterung um mehrfache Einheiten

Ein Schwachpunkt bei Kennedys Dimensionstypsystem ist die Verwendung von nur einer einzigen Einheit für jede Dimension. Diese Einheit ist innerhalb eines Programms jedoch konsequent zu verwenden, weshalb Einheitskonflikte innerhalb einer Dimension auch ausgeschlossen sind. Eine praktikable Erweiterung wäre hier jedoch durch das Hinzufügen der Möglichkeit von mehreren Einheiten innerhalb einer Dimension gegeben. Wie weiter oben bereits erwähnt, differieren Einheiten innerhalb einer Dimension lediglich um einen Skalierungsfaktor, den es an dieser Stelle mit einzubeziehen gilt. Insgesamt stellt dies jedoch keine außerordentliche Schwierigkeit dar. Lediglich die Programme müssten geändert werden, damit Umrechnungen durchgeführt werden können, wobei sich eventuell numerische Fragestellungen ergeben: Würde man beispielsweise Werte in *cm* und *m* addieren, so macht es grundlegend Unterschied, ob man das Ergebnis in *m* oder in *cm* weiterführt. In der Praxis ist es daher wahrscheinlich sinnvoller, diese Entscheidung dem Programmierer zu überlassen. Allerdings ist die Entscheidung dann auch zu fordern, indem nur erlaubt wird, in der gleichen Einheit zu rechnen. Dabei sollten dann natürlich explizite Typumwandlungen innerhalb einer Dimension möglich sein.

4 Fazit

Wir haben gesehen, inwiefern Typsysteme Laufzeitfehler verhindern können, und wie sie die Menge der gültigen Programme intelligent einschränken. Weiterhin konnten wir Dimensionstypen erfolgreich in die konventionellen Typkonzepte eingliedern und so die Idee der Dimensionstypsysteme erreichen. Ferner war es uns möglich, Dimensionstypsysteme innerhalb der konventionellen Typsysteme zu verstehen und wichtige Eigenschaften zu benennen. Dimensionstypsysteme können uns demnach helfen, Fehler innerhalb von dimensionsbehafteten Ausdrücken zu vermeiden.

Literatur

- [1] Type Systems, Luca Cardelli,
Digital Equipment Corporation, Systems Research Center.
Allen B. Tucker (Ed.): *The Computer Science and Engineering Handbook*.
CRC Press, 1997, ISBN: 0-8493-2909-4. Chapter 103, pp 2208-2236.
[http://www.luca.demon.co.uk/Bibliography.html#Type systems](http://www.luca.demon.co.uk/Bibliography.html#Type%20systems)

- [2] Dimension Types, Andrew Kennedy, University of Cambridge.
In *Proceedings of the 5th European Symposium on Programming*:
Lecture Notes in Computer Science volume 788, Springer-Verlag, 1994.
<http://citeseer.nj.nec.com/kennedy94dimension.html>