

Pragmatik von Typsystemen, Objektorientierung und EIFFEL

Studienarbeit

Bearbeiter: Robert Will, 29335
Betreuer: Dr.-Ing. Günter Hübel

Fachgebiet Softwaretechnik und Programmiersprachen
Technische Universität Ilmenau

Kurz gefasst: Nach einer kurzen Einführung in den Zweck der Typisierung, werden zunächst einige praktisch relevante und theoretisch fundierte Typsysteme vorgestellt, zuletzt das der Sprache EIFFEL mit seinen drei Problemen. Dann werden verschiedene Lösungen zu diesen Problemen aus der Literatur zusammengefasst; diese werden zueinander und zu den anderen Typsystemen in Beziehung gesetzt, wobei der Zweck der Typisierung im Vordergrund steht. Die Arbeit schließt mit einer Sammlung von Argumenten gegen die Covarianz.

Inhalt

1	Einleitung	1
2	Grundlegende Typsysteme	3
2.1	Verwirrung über Typen	3
2.2	Typ-Pragmatik.....	4
2.3	Typen als maximale disjunkte Mengen.....	7
2.4	Die Mutter aller Typsysteme: PASCAL.....	8
2.5	Subtypen und Generizität.....	10
2.6	Zwei moderne Typsysteme: HASKELL und EIFFEL.....	11
2.7	Cardellilogie.....	14
2.8	Subtypen und Generizität im Detail.....	16
2.9	Beschränkte Generizität.....	19
3	EIFFELS drei kleine Probleme	22
3.1	Exporteinschränkung (descendant hiding)	23
3.2	Generische Konformität	25
3.3	Covarianz	29
4	Vertragsstreitigkeiten: bisherige Lösungen	31
4.1	Konformitätslücke — Die erweiterte Typprüfung	32
4.2	Monomorphe Typen.....	35
4.3	Neubesetzung mit recast	36
5	Typing by Contract	37
5.1	Abschwächung ankündigen.....	37
6	Anwendungsfälle der Covarianz	39
6.1	Perfekte Beispiele für schlechten Entwurf	39
6.2	Peer Routines	41
6.3	Denk mal einer an die Erben	42
7	Vertragsverhandlungen: bessere Typsysteme	46
7.1	Rekursive Generizität	46
7.2	Matching	48
7.3	Eine andere Interpretation.....	49
7.4	Pragmatische Regeln	51
7.5	Exkurs: Generische Routinen	53
8	Abschließende Empfehlungen	54
8.1	Formale Typregeln.....	54
8.2	Covarianz: Eine Lösung ohne Probleme.....	55
8.3	Schluss.....	56
9	Literatur	57

1 Einleitung

Die beste Programmiersprache der Welt hat Löcher in ihrem Typsystem! Mit dieser desillusionierenden Feststellung fing alles an. Wie konnte so etwas möglich sein? Typen sind doch gar keine so komplizierte Angelegenheit, und ein Loch im Typsystem, das würde ja bedeuten, dass Programme willkürlich abstürzen, dass man sich nicht mehr auf das Typsystem als Hilfe beim entwerfen und dokumentieren von Programmen verlassen kann, und überhaupt was nützt uns ein Typsystem, wenn wir uns nicht darauf verlassen können? Eine Sprache mit Löchern im Typsystem kann schwerlich die „beste Programmiersprache der Welt“ sein.

Bei näherer Betrachtung stellt sich die Lage als weit weniger dramatisch dar: Eiffel hat durch ihre konsequente Objektorientierung und durch die Unterstützung beschränkter Generizität eines der ausdrucksstärksten objektorientierten Typsysteme überhaupt. An manchen Stellen lässt dieses Typsystem, Dinge zu die theoretisch eigentlich verboten sein sollten, weil sie zu Inkonsistenzen führen können, dass heißt zu Laufzeitfehlern, die eigentlich gar nicht vorkommen dürften. Glücklicherweise gibt es aber noch „dem Typsystem nachgeschaltete“ Konsistenzregeln, mit denen der Compiler die Abwesenheit dieser Klasse von Laufzeitfehlern überprüfen kann, bevor ein Programm ausgeführt wird.

In dieser Arbeit wird gezeigt, dass man es noch besser machen kann, wenn man die Typisierung aus ausreichend großem Abstand betrachtet. Folglich kreist die Arbeit um zwei Themen: Zum Einen die Darstellung grundlegender praktischer Typsysteme und möglicher Erweiterungen, zum anderen die konkreten „Löcher“ in der Sprache Eiffel und Möglichkeiten, sie zu schließen.

Zum Schließen der Typlöcher gibt es im Grund zwei „Richtungen“: einerseits kann man sich die vorhandenen Typregeln hernehmen und die Stellen suchen, die an der Entstehung von Inkonsistenzen beteiligt sind und diese leicht ändern. Andererseits kann man neue Prinzipien aus anderen, insbesondere theoretisch gut untersuchten, Typsystemen übernehmen. Die erste Möglichkeit ist eher pragmatisch und *ad hoc*, sie hat den Vorteil, dass man Programmierer noch in den alten Konzepten denken können, die sich in der Praxis als vorteilhaft erwiesen haben. Insbesondere wenn die neuen leicht geänderten Regeln gar nicht so oft gebraucht werden, ist das ein Vorteil. Die zweite Möglichkeit hat den Vorteil, dass das neue Typsystem „aus einem Guss“ ist und auf einheitlichen, einfachen Konzepten basiert. Wenn diese Konzepte nicht allzu weit von den Anforderungen der Praxis entfernt sind, kann dies ein ganz

besonderer Vorteil sein: dann können Programmierer nämlich lernen, in diesen Konzepten zu denken, und damit ein höheres Abstraktionsniveau und Fehlerfreiheit durch Konstruktion erreichen.

Wenn man den genannten zweiten Weg zur Schließung der Typlöcher gehen will, muss man sich zwangsläufig mit allgemeinen Konzepten der Typisierung beschäftigen. So sind die beiden Themenkreise dieser Arbeit verbunden. Außerdem werden wir sehen, dass spätere Lösungsvorschläge zu Eiffels Problemen sich wesentlich dadurch unterscheiden, welchen Kompromiss zwischen *ad hoc* Modifikation und neuen Konzepten sie wählen. Auch verschiedene Denkweisen und Programmierstile werden explizite Rollen spielen.

Gliederung dieser Arbeit

Zunächst werden in Abschnitt 2 einige schon existierende Typsysteme vorgestellt. Wir beginnen dabei mit dem der Sprache PASCAL, weil es spätere Typtheorien wesentlich beeinflusst hat. Eine Theorie auf die wir uns wesentlich beziehen ist die von Cardelli (1997). Wir gehen dann zu den Sprachen HASKELL und EIFFEL über, die im wesentlichen den Stand der Praxis repräsentieren. Dieser Teil der Arbeit schließt mit einer Betrachtung von beschränkter Generizität, einem Konzept, das für die späteren Teil von großer Bedeutung sein wird.

Dann wenden wir uns gezielt der Sprache Eiffel zu: Abschnitt 3 widmet sich den berüchtigten Typlöchern von Eiffel und Abschnitt 4 stellt ein paar *ad hoc* Lösungen dazu vor. Die späteren Teile beschäftigen sich dann nur noch mit dem dritten Problem: der Covarianz. Abschnitt 5 betrachtet sie aus methodischer Sicht. Wir nehmen die Sichtweise der vertraglichen Programmierens und des Softwareentwurfs ein, und wir schauen uns näher an, wozu Covarianz überhaupt verwendet wird. Als wesentlicher Anwendungsfall stellen sich die sogenannten *peer routines* heraus und in Abschnitt 6 benutzen wir unser Wissen über andere Typsysteme, um diesen Routinen genauere Typen zu geben. Es stellt sich heraus, dass diese erweiterten Typsysteme sehr eng zusammenhängen, und dass sie (wenn gut umgesetzt) von hohem praktischen Nutzen sind.

Die letzten Abschnitte erläutern dann die Hauptthesen dieser Arbeit:

1. Covarianz im allgemeinen verstößt gegen das vertragliche Programmieren und hat folglich in einer Programmiersprache nichts zu suchen. Meistens erreicht man besser strukturierte Lösung durch Verwendung von Generizität. **Wir brauchen eine neue Denkweise, die frei ist von Covarianz.**

2. Generizität und insbesondere eine Erweiterung der beschränkten Generizität reichen aus, um Covarianz und Eiffels verankerte Typen völlig zu ersetzen. **Wir brauchen eine Erweiterung des Typsystems, um *peer routines* genauer zu typisieren.**

2 Grundlegende Typsysteme

2.1 Verwirrung über Typen

Ziel dieser Arbeit ist es ja, Typsysteme in einem pragmatischen Licht zu sehen. Um zu zeigen, wie viele andere Sichten es noch gibt und welch großes Durcheinander man damit bekommt, und um gleich schon am Anfang mal die zitierte Literatur zu übersetzen, präsentiere ich hier die Sicht von Wegener (1990, Seite 33). Er schreibt:

Der hauptsächliche Gegensatz liegt zwischen der Sicht des Compilerautors von Typen als Eigenschaften von Ausdrücken und der Sicht des Applikationsprogrammierers von Typen als Arten von Verhalten. Eine tiefere Analyse ergibt die folgenden Sichten:

Sicht des Applikationsprogrammierers: Typen sind ein Mechanismus zur Klassifikation von Werten nach deren Eigenschaften und Verhalten.

Sicht der Systemevolution (objektorientiert): Typen sind ein Werkzeug der Softwaretechnik zur Verwaltung von Softwareentwicklung und -Evolution.

Sicht der Systemprogrammierung (Sicherheit): Typen sind ein Anzug von Kleidung (Anzug von Rüstung), um rohe Daten gegen unbeabsichtigte Repräsentation zu beschützen.

Sicht der Typprüfung: Typen sind syntaktische Restriktionen von Ausdrücken, um die Kompatibilität von Operanden und Operatoren zu sichern.

Sicht der Typinferenz: Ein Typsystem ist eine Menge von Regeln, die jedem Teilausdruck einen eindeutigen generellsten Typ zuordnet, der die Menge aller sinnvollen Kontexte wiedergibt, in denen der Teilausdruck vorkommen kann.

Sicht der Verifikation: Typen bestimmen Verhaltensinvarianten, die von den Instanzen des Typs erfüllt werden müssen.

Sicht des Implementierers: Typen beschreiben Speicherabbildungen für Werte.

Die ist bereits eine bunte Bowle von Sichten und dabei hat Wegener noch mindestens die **Sicht der intuitionistischen Typtheorie** vergessen: Typen sind Aussagen, Werte sind Beweise.

2.2 Typ-Pragmatik

Typsysteme dienen hauptsächlich dazu, Fehler in Softwaresystemen möglichst früh zu finden, oder gar nicht erst hineinzulassen. Ein paar Beispiele für solche Fehler sind:

- Addition von Zahlen und Strings
- Benutzung nicht existierender Felder eines Records
- Manipulation von Referenzen
- Verwechseln von lokalen und globalen Variablen
- Verletzen von Invarianten durch Zugriff auf Interna eines Moduls

Die ersten beiden Beispiele sind noch sehr konkret, sie stammen aus den frühen Zeiten der Programmierung, das letzte Beispiel ist schon sehr abstrakt, eine konkrete Variante davon wäre zum Beispiel die Manipulation der internen Zeiger eines Objekts vom Typ Liste — abstrakte Datentypen und Module hängen ja untrennbar zusammen. Genauso abhängig ist die Typisierung vom Geltungsbereich (*scope*) von Variablen, daher muss man Fragen von global und lokal bei der Diskussion von Typen immer mit behandeln.

Besonders wichtig ist auch der Aspekt "Manipulation von Referenzen", denn wenn man auf diese Operationen wie zum Beispiel Addition anwenden kann, wird *Garbage Collection* unmöglich! Außerdem sind Programmierfehler bei Referenzen besonders gefährlich, weil diese den Speicher des gesamten Programms betreffen können. Das Betriebssystem schützt zwar den Speicher verschiedener Prozesse (Programme) voreinander, aber wenn man die schiere Größe auch nur eines einzelnen modernen Programms in Kauf nimmt, reicht das einfach nicht aus. Programmierfehler mit Referenzen sind die aller schlimmsten, weil Ursache und Wirkung völlig ohne Zusammenhang sind. Einen String mit einer Zahl zu addieren ist dagegen überhaupt kein Problem: so ein Fehler ist durch gute Tests (relativ) leicht zu finden. (Trotzdem ist es natürlich pro-

duktiver, den Fehler noch früher zu finden, oder gar nicht erst zu machen.)

Wahrscheinlich ist gerade dieser Unterschied in der Schwere der Fehler der Grund warum Smalltalk Programmierer behaupten, dass sie prima ohne statische Typisierung auskommen, weil sie stattdessen intensive Tests durchführen. In Smalltalk gibt es keine Zeigeroperationen und dadurch sind die möglichen Fehler alle noch relativ lokal. Die Unit-Tests sind dann eine indirekte Art von Schnittstellen-Verträgen. Smalltalk Programmierer behaupten auch, dass sie ihre Programme öfter ausführen und testen, als andere Programmierer die Programme überhaupt kompilieren (und Typ-Prüfen). Hier sieht man wieder einmal den Zusammenhang zwischen Methode und Werkzeugen (Programmiersprache). Letztendlich vergessen Smalltalk Programmierer dabei aber mindestens die konstruktive Seite der Typisierung: einfache, formale Dokumentation und Unterstützung schon bei der Programmentwicklung.

Man kann diese Überlegungen zusammenfassen, indem man sagt, dass die Sicherung gegen Referenzierungsfehler Module schützt, so dass Zugriffe nur über die Schnittstellen erfolgen können, und nicht "kreuz und quer". Der nächste Aspekt der Typisierung ist dann, dass die Typen helfen, diese Schnittstellen besser zu beschreiben. Man kann dabei Typen als eine spezialisierte Art von Verträgen (oder Schnittstellen) sehen. Manche gehen sogar so weit, Typen als die Spezifikation selbst zu betrachten. Besser ist es aber, die Typen wiederum als Voraussetzung der Spezifikation zu sehen: erst wenn man weiß, welcher Art die Ein- und Ausgaben eines Programms oder eine Routine sind, kann man diese genauer beschreiben. Wenn man in einer Zusicherung schreibt " $x > 0$ " dann hängt dessen Bedeutung ja vom Typ von " x " ab: Wenn es eine Ganzzahl ist, ist es gleichbedeutend mit " $x = 1$ ".

Und damit kommen wir schon zu einer weiteren Aufgabe für Typsysteme. Sowohl im richtigen Leben, als auch in der Mathematik kommt es oft vor, dass ein Begriff verschiedene Bedeutungen hat, je nachdem worauf er sich bezieht. Mathematiker sprechen ganz selbstverständlich von der Abgeschlossenheit eines Graphen und von der Abgeschlossenheit einer Menge von Graphen, obwohl die formalen Definitionen natürlich völlig verschieden sind. Im Text ist aber immer klar, was gemeint ist, wenn sie schreiben "G ist abgeschlossen" oder "S ist abgeschlossen", solange der Typ von G und S (Menge oder Graph?) immer klar ist. Dies nennt man "Polymorphismus" und ohne Typisierung ist es nur auf eine ad hoc Art und Weise möglich. (Zu den ad hoc Methoden zählt zum Beispiel die Funktionsüberladung nach Argument-Typen, wie sie in manchen Programmiersprachen üblich ist, aber selbst diese basiert auf Typen.)

Im einfachsten Fall spiegelt sich Überladung im Typsystem wieder, indem ein Begriff in verschiedenen Modulen mit verschiedener Bedeutung verwendet wird. Dies ist noch eine einfache Angelegenheit von Geltungsbereichen. Wirklicher Polymorphismus existiert aber erst, wenn ein Begriff an derselben Stelle eines Programms verschiedene Bedeutungen annehmen kann. Ein klassisches Beispiel dafür ist ein Sortieroutine, die den Operator "<" verwendet, welcher eine andere Bedeutung hat, je nachdem was man sortiert — der Programmtext zum Sortieren ist aber stets derselbe. Man kann diese Art der Überladung also durch implizite Parametrisierung erklären: so als hätte die Sortieroutine den Operator als weiteren Parameter. In diesem Sinne ist die Überladung auch nur eine Abkürzung (die ziemlich viel auf einmal abkürzt), aber dieses System hat noch einen weiteren Vorteil: man kann verschiedene Funktionen zu einer gemeinsamen Abstraktion zusammenschließen und dann nicht nur mit einem Namen auf alle auf einmal Bezug nehmen, sondern man kann auch *gemeinsame Eigenschaften* der verschiedenen Begriffe wieder verwenden. Zum Beispiel die Regel " $x < y = y > x$ " oder " $(x < y \text{ oder } x = y) = x = y$ ". Und das ist ein wirklicher Abstraktionsgewinn.

Schließlich gehört es auch zu den guten Eigenschaften eines Typsystems, dass es die Denkprozesse des Programmierers unterstützt. Gegner von strenger Typisierung behaupten zwar, die Typen würden ihr kreatives Denken stören, aber andere behaupten das Gegenteil: beim Spezifizieren und Lösen eines Problems ist es hilfreich über die Typen von Ein- und Ausgabe nachzudenken, um sich einer Lösung anzunähern. Dazu muss das Typsystem aber auch wirklich ein gutes sein, und die essentiellen Konzepte der Programmierung widerspiegeln. Später werden wir noch eine Reihe von Beispielen sehen, bei denen die Typen eher verwirren — dies liegt aber daran, dass das Thema dieser Arbeit ja gerade die Randzonen sind, bei denen die Typen nicht so gut auf die Probleme passen.

Zusammenfassung *Aufgaben eines Typsystems*:

- Beschränkung der Kommunikation zwischen Modulen auf explizite Schnittstellen
- Abgrenzung dieser Schnittstellen durch "Abschätzung nach oben"
- Strukturierung der Programmiersprache und Erlauben von gezieltem Polymorphismus
- Ermöglicht effizientere Kompilierung

Diese Aufzählung mag wieder etwas subjektiv gefärbt sein — der starke Zusammenhang von Typen, Modulen und Schnittstellen kommt ja

aus dem Bertrand Meyer'schen Weltbild (1988). Allerdings geben andere zitierten Werke oft keine eigene Definition an. Cardelli (1997) gibt ähnliche Aufgaben an und Cardelli (1991) konzentriert sich auf die "geordnete Evolution" von Software, und damit schließt sich der Kreis in gewissem Sinne, denn genau diese Evolution ist eines der Ziele von Meyers Modularität und Verträgen.

Typen haben also eine Doppelrolle als Basis von Verträgen (indem sie den Diskursbereich festlegen) und als statischer Teil von Verträgen. Die wesentliche Abgrenzung von Typen und allgemeinen Verträgen ist also nicht nur, dass man Typen immer voll statisch prüfen können muss, sondern auch, dass Typen auf einem einfach Modell basieren sollen, dass sich zur Dokumentation und Strukturierung eines Programms eignet.

2.3 Typen als maximale disjunkte Mengen

Man ist sich darüber einig, dass Typen die Mengen von Werten sind, die in einem Programm auftreten können. Jeder Wert soll dabei genau einen Typ haben (oder mindestens einen, wenn man Polymorphismus in Betracht zieht). Das heißt, ohne Polymorphismus kann kein Wert zu mehreren Typen gehören — die Typmengen sind disjunkt.

Mit der Mengentheorie kann man das komplette Verhalten eines Softwaresystems spezifizieren. Eine uneingeschränkte Verwendung der Mengentheorie würde einen Typprüfer also ganz klar überfordern. Welche Einschränkung ist es also, die das Prüfen der Typen wieder vereinfacht? Antwort: alle vorkommenden Funktionen sind über den Mengen, die die Typen darstellen, *abgeschlossen*. Das heißt, dass die Nachbereiche der Funktionen immer Untermengen der Typen sind. Es treten also keine Mengen auf, die "größer" als ein Typ sind, deswegen nennt man die Typen maximale Mengen¹. Der Typprüfer kann also alle im Programm auftretenden Wertemengen durch die Typen nach oben abschätzen — er hat also wirklich nicht viel zu tun.

¹ Ich kenne die Gleichung "Typen = maximale Mengen" aus dem Umfeld der Spezifikationssprache Z. Ich weiß nicht, ob das auch noch woanders so verwendet wird.

Typgrundregel

Typen sind Mengen von (zulässigen) Werten aus einer Programmiersprache. Jeder Wert gehört (mindestens) zu einem Typ.

Die Typsysteme, die ich im Folgenden vorstelle, unterscheiden sich dadurch, was es für Typen gibt und durch welche Prinzipien sie zusammenhängen. Wie Werte zu Typen kommen folgt stets denselben Prinzipien, unterschiedlich ist aber, welche Typregeln im Vordergrund stehen.

Man kann sich fragen, inwiefern dieses Bild noch auf polymorphe Typen zutrifft. Bei Subtypen bleibt es bestehen: die Menge zu jedem Typ enthält dann die Mengen, die zu allen seinen Subtypen gehören. Man kann also in der Tat diese Mengen nachträglich erweitern, indem man einen neuen Subtypen einführt. (Vorteil der Objektorientierung...) Natürlich sind solcherlei Subtypen, die Untermengen einer gemeinsamen Menge sind nicht mehr maximal und schon gar nicht mit ihrem Supertypen disjunkt. In Unterabschnitt 6.1 werden wir über dieses Problem stolpern.

Bei parametrischem Polymorphismus wird es schwieriger: hier ist es in der Tat am besten sich parametrische Typen als Funktionen vorzustellen, die für jeden zulässigen Wert der Parameter einen Typ, also eine Menge darstellen. Ein Wert, der einen parametrischen Typ hat, kann also zu jedem Typ (jeder Menge) aus dem Nachbereich des parametrischen Typs gehören. Aber da Eiffel keine parametrischen Typen hat (siehe Unterabschnitt 2.8), müssen wir uns darum keine Sorgen machen.

2.4 Die Mutter aller Typsysteme: PASCAL

Basierend auf den Datentyptheorien aus den *Notes on data structuring* (Dahl, Dijkstra, Hoare 1972) war PASCAL die erste Programmiersprache, die ein signifikantes Typsystem enthielt. PASCAL lieferte damit die Grundlage für spätere Typtheorien und begründete die *konstruktivistischen Typen*, wie ich sie nenne. Diese Bezeichnung wird klar, wenn man sich das PASCAL'sche Typsystem anschaut: da gibt es ein paar vordefinierte Typen und Typkonstruktoren wie Felder und Records mit denen man neuen Typen definieren kann.

PASCALs primitive Typen sind *Integer*, *Real*, *Double*, *Char* und *String*. PASCAL definiert für jeden dieser Typen die entsprechenden Konstanten und primitiven Operationen. Die Überladung von Operatoren wie + und die Konversion von numerischen Typen wird dabei ad hoc ge-

regelt. Es gibt also explizite Regeln der Programmiersprache, wie „Real plus Integer ist legal und ergibt Real“.

In späteren Sprachen wird String bereits ein zusammengesetzter Typ sein (`List of Char`) oder ein abstrakter Typ (der Typ zur Klasse `STRING`, möglicherweise Unterklasse von `LIST[CHAR]`), aber in PASCAL sind wir noch nicht so weit. Trotzdem hat PASCAL einen großen Fortschritt in der Typisierung erzielt, weil zum ersten Mal verschiedene Typkonstruktoren (relativ) frei kombiniert werden können. (Eine der Einschränkungen ist zum Beispiel, dass Records nicht als Funktionsergebnisse dienen können. Neuere Versionen der Sprache (bzw. der Compiler) haben diese Einschränkung nicht mehr.)

PASCALS Typkonstruktoren sind Felder, geschrieben als `array[Bereich] of Typ`, Records, geschrieben als `record Name : Typ; ... end` und Zeiger, geschrieben einfach als `?Typ`. Eine weitere Besonderheit ist dabei noch, dass Pascal die besondere Gruppe von *Ordinaltypen* unterscheidet und zwei spezielle Typkonstruktoren dafür einführt: *Aufzählungstypen*, deren Elemente abstrakte Werte sind, die sich nur durch Namen unterscheiden, und *Teilbereichstypen*, die eine Untermenge eines anderen Ordinaltypen darstellen. Die geniale Struktur des PASCAL'schen Typsystems kommt dadurch zum Ausdruck, dass genau die Ordinaltypen als Indextypen eines Feldes zugelassen sind. Dabei spielt es keine Rolle, ob man den Typ dabei gerade erst definiert (wie in `array[1..10] of Real`) oder ob man einen schon benannten Typ verwendet (wie in `array[Twochentag] of Real`).

Auch PASCALS Zeiger sind ein Novum: zum ersten Mal kann man in einer imperativen Sprache auch rekursive Daten darstellen, ohne sich direkt mit Speicheradressen zu beschäftigen. Um diese Rekursion zu erlauben, dürfen Zeigertypen als einzige schon verwendet werden bevor sie deklariert werden. Die Einschränkung, dass das für andere Typen nicht geht, hat nicht nur mit der einfacheren Kompilierbarkeit in einem Durchlauf zu tun, sondern es verhindert auch, dass Records rekursiv definiert werden — dies würde ja zu unendlichen Datenstrukturen führen.

Zu guter Letzt hat PASCAL auch noch so genannte *variante Records* (oder *Recordvarianter?*). Deren Syntax hebt dabei die Parallelität zur endlichen Fallunterscheidung hervor:

```
record case Name : Typ of
  Wert :
    Name : Typ; ...
end
```

Damit hat PASCAL schon alle Typkonstruktoren, die wir auch bei Cardelli (1997) finden, nur die Generizität und die Subtypen fehlen noch; leider sind das genau die Konstrukte, die zu unseren Covarianz-Problemen führen.

2.5 Subtypen und Generizität

Eine Forschung über Subtypen begann erst mit der steigenden Popularität objektorientierter Programmiersprachen. Deswegen vergleicht auch Meyer (1986) noch *Generizität und Vererbung* und nicht Generizität und Subtypen. Dass der Unterschied zwischen den beiden noch nicht völlig verstanden wurde, zeigt sich einerseits darin, dass die meisten OO Sprachen meinen Generizität völlig durch Subtypen ersetzen zu können, andererseits darin, dass auch EIFFELS Typprobleme in diesem Bereich auftreten.

Die Problematik hat wohl auch damit zu tun, dass beide Konzepte intuitiv völlig einleuchtend sind: *Generische Typen* haben Typparameter, die man verwenden kann wie Typen. Generische Typen kann man in normale Typen umwandeln, indem man die Parameter mit Typen instanziiert. Dies ist ein ganz normaler Ersetzungsprozess, leicht zu verstehen für alle, die mit Programmierung zu tun haben.

Es besteht allerdings ein wichtiger Unterschied zwischen der Typ-Ersetzung und einer allgemeinen Ersetzung einer Variablen durch einen Text, wie sie in vielen Sprachen durch *Makros* oder *Templates* vertreten ist. Bei der Typ-Ersetzung kann man im inneren Geltungsbereich stets voraussetzen, dass die Variable für einen gültigen Typ steht (nicht nur für einen beliebigen Text), und im erweiterten Fall kann man sogar noch mehr Voraussetzungen über diesen Typ machen, die natürlich am Beginn des Bereiches deklariert sein müssen. Im Gegensatz zur einfachen Textersetzung führt das zu robusterer Software und leichter verwendbaren Komponenten.

Subtypen andererseits etablieren eine ganz einfache Beziehung zwischen zwei Typen: ein Typ unterstützt alle Operationen, die der andere Typ auch unterstützt, folglich kann man den einen Typ auch überall dort einsetzen, wo man den anderen Typ einsetzen kann, dies nennt man auch *Ersetzbarkeit* (*substitutability*).

Das hier ebenfalls verwendete Wort „ersetzen“ deutet schon auf die Ähnlichkeit der beiden Mechanismen hin, es macht aber auch auf prägnante Art den Unterschied klar: bei der Generizität ersetzen wir Variablen durch Typen, bei Subtypen ersetzen wir Typen durch Typen. Und was noch schöner ist: man kann sogar aus diesem Vergleich schon die wesentlichen Vor- und Nachteile der beiden Mechanismen erkennen!

Mit Subtypen kann man ein Programm ergänzen, ohne dass diese Ergänzung vorher vorgesehen war. Der „alte Code“ wird einfach für einen konkreten Typ geschrieben, später verwendet man diesen Code auch für neue Subtypen des alten Typs. Variablen hingegen muss man von Anfang an vorsehen. Andererseits kann man zwar jeden Typ als eine Typvariable auffassen, die für den Typ selbst und alle seine Subtypen steht, aber eine solche Variable hat natürlich keine Identität, sie kann nur einmal vorkommen. Man kann damit also nicht ausdrücken, dass aus einer Datenstruktur immer Werte desselben Typs herauskommen, wie auch hineingegangen sind, oder dass eine Identitätsfunktion natürlich wieder denselben Typ für Argument und Resultat hat.

Generische Typen helfen auch besonders bei der Dokumentation von Softwaresystemen, es ist einfach eine nützliche und völlig einleuchtende Information, zu sagen „hier eine Liste von Ganzzahlen, da eine Tabelle von Strings zu Adressobjekten“. In EIFFEL wären das zum Beispiel `LIST[INTEGER]` und `TABLE[STRING, ADRESS]`. Ebenso leicht fällt es (man verzeihe mir die Vorwärtsreferenz zu Teilabschnitt 2.8) aus einer generischen Klassenschnittstelle die Schnittstellen der zugehörigen Typen abzulesen.

Die jüngste Entwicklung von JAVA hat gezeigt, dass Subtypen allein keine Generizität ersetzen können: Ab Version 1.5. wird JAVA generische Klassen haben. Möglicherweise hat die Generizität auch noch mehr Potential, als selbst Bertrand Meyer dachte, und generische Routinen in EIFFEL lösen die Probleme von Covarianz und verankerten Typen.

2.6 Zwei moderne Typsysteme: HASKELL und EIFFEL

Bevor ich im nächsten Teilabschnitt die modernen Ausprägungen von Subtypen und Generizität erläutere kommen hier erstmal die Grundlagen dazu: zwei Typsysteme, an denen man's erklären kann. Beide Typsysteme nehmen die Aufgabe sehr ernst, die Programmiersprache zu vereinfachen. Sie basieren auf einfachen Prinzipien und haben nur wenige aber universelle primitive Konstrukte.

HASKELLS Typsystem ist ein besonders einfacher Vertreter der Kategorie der konstruktiven Systeme. HASKELL ist ein gutes Beispiel, weil es auch eine sehr *nützliche* Programmiersprache ist. Das System veranschaulicht Cardellis Prinzipien (1997) mit der Ausnahme, dass HASKELL über keinerlei Subtypen verfügt. Dass man trotzdem große Softwaresysteme in HASKELL schreiben kann zeigt, dass Subtypen möglicherweise gar nicht so essentiell sind.

In HASKELL existieren Datentypen und Funktionen unabhängig voneinander als Werte der Programmiersprache, das heißt Funktionen haben Typen (nicht nur Signaturen) und alle Variablen können sowohl für Funktionen als auch für Daten stehen. Funktionsapplikation und Datenzugriff sind die einzigen primitiven Operationen. Alle Datentypen sind algebraisch, sie werden definiert mit der universellen Formel `data Typkons param = Wertkons Typ ... | Wertkons ...`. *Typkons* ist dabei der Name für den neuen Typ (man nennt den Namen Typkonstruktor, weil man sich ihn als Funktion von den Typparametern zu einem neuen Typ vorstellen kann), *Wertkons* sind die Wertkonstruktoren für diesen Typ. Man muss alle algebraischen Typen deklarieren, weil damit automatisch die Wertkonstruktor Funktionen definiert werden. Vordefinierte Datentypen wie Listen, Tupel und Zeichen / Ganzzahlen lassen sich im Prinzip auch auf diese algebraische Art definieren. Nur der Konstruktor \rightarrow für Funktionstypen muss separat eingeführt werden. Dann allerdings lassen sich aus diesen Konstruktoren Typen beliebig kombinieren.

Zum Beispiel ergeben sich Funktionen mit mehreren Parametern durch Kombination von Funktionen untereinander oder mit Tupeln:

`(Integer \rightarrow Integer) \rightarrow Integer` oder
`(Integer, Integer) \rightarrow Integer`

sind beides Typen für Funktionen mit zwei Parametern. Genauso leicht erhält man eine Funktion mit zwei Ergebnissen:

`Integer \rightarrow (Integer, Integer)`

Oder eine Funktion über Listen:

`(a \rightarrow b) \rightarrow [a] \rightarrow [b]`

Oder eine Liste von Funktionen:

`[a \rightarrow String]`

In den beiden letzten Beispielen habe ich schon Typvariablen verwendet, wie sie im Unterabschnitt 2.8 erklärt werden. Die elementaren Typregeln von HASKELLS Systems sind die Funktionsanwendung und die Funktionsbildung. Beide finden sich auch bei Cardelli (1997), sie lauten vereinfacht:

$f : A \rightarrow B, x : A$

$f x : B$

und

$$x : A \Rightarrow e(x) : B$$
$$(\lambda x : A \rightarrow e(x)) : A \rightarrow B$$

Die zweite Regel konstruiert dabei eine anonyme Funktion und sie enthält eine Typannotation im Code („das linke ‚A‘ unter dem Strich“). HASKELL erfordert diese Annotation nicht, aber in OO Sprachen und EIFFEL ist sie notwendig, deswegen habe ich sie gleich mal mit eingezeichnet. Das was ich oben „Datenzugriff“ nannte ist wieder die primitive endliche Fallunterscheidung, die ich auch schon mal bei PASCAL erwähnte. Sie spielt für OO Sprachen keine so große Rolle, da man dort die Fallunterscheidungen hauptsächlich implizit über dynamische Bindung abwickelt.

EIFFEL ist ein besonders gutes Beispiel für ein, wie ich es nenne: *abstraktives Typsystem*, weil es quasi die Sprache ist, die dieses Prinzip zum ersten Mal explizit gemacht hat. Außerdem ist EIFFEL neben TRELIS/OWL die einzige (rein) objektorientierte Sprache, die auch generische Typen unterstützt. In EIFFEL haben Routinen nicht den gleichen Status, wie Werte eines Typs, dafür spielen sie aber eine entscheidende Rolle bei der Definition von Typen, denn genau das ist der Grundgedanke der abstraktiven Typsysteme: ein Typ wird nicht durch seine Struktur (mit Hilfe von Konstruktoren) definiert, sondern direkt durch Angabe der anwendbaren Operationen. Sprachen mit konstruktiven Typsystemen definieren abstrakte Typen indirekt, indem sie einfach den Typnamen und die Operationen aus einem Modul exportieren, die Typstruktur (d.h. die primitiven Konstruktoren) aber nicht.

Das Konzept des abstrakten Datentyps in EIFFEL ist folgendermaßen rekursiv definiert: Ein Typ besteht aus einer Menge von Elementen (*features*) und einer Menge von Schöpfungsprozeduren (*creation procedures*). Die Elemente sind entweder Anfragen (*queries*) oder Befehle (*commands*). Schöpfungsprozeduren und Befehle haben eine Signatur bestehend aus einer Liste von Parametern wiederum bestehend aus Name und Typ. Anfragen haben eine Signatur bestehend aus einer ebensolchen Liste und einem Ergebnistyp.

Und das war schon alles! Alle Typen von der Ganzzahl über Felder, Tabellen und Strings, bis zum HTML-Dokument lassen sich so darstellen. Selbst Funktionen und Prozeduren kann man dadurch wieder zu Werten machen, indem deren Typ einfach nur ein Element anwenden (*apply*) bzw. aufrufen (*call*) enthält — genau das macht EIFFEL mit seinen Agenten. Die Rekursion in obiger Definition besteht übrigens darin, Typen durch Signaturen zu definieren und Signaturen wieder durch Typen.

Dadurch wird alles angenehm einfach. Man erhält einen rekursiven Typ, wenn eine der Signaturen sich wieder auf den definierten Typen selbst bezieht (und nicht nur auf andere Typen). Solche rekursiven Typen werden wir bald als spezielle Problemklasse kennen lernen.

Wie wir bei Meyer (1995) nachlesen können, ist der *Elementaufruf* (*feature call*) die wichtigste Typregel in EIFFEL, dazu kommt noch die Zuweisung.

a) $x.f(\text{arg})$ b) $y := \text{exp}$

Beim Aufruf **a)** ist sicherzustellen, dass das Element f auch wirklich existiert, und bei der Zuweisung und Parameterübergabe **b)** ist sicherzustellen, dass alle Entitäten (formale Argumente in f bzw. Ziel der Zuweisung y) nur an passende Objekte (arg bzw. exp) gebunden sind. Die erste Frage sollte sich einfach beantworten lassen, indem man auf den Typ von x schaut: gibt es dort f mit der passenden Zahl von Argumenten? Bei der zweiten Frage sollte es eigentlich reichen, dass der (mit üblichen funktionalen Regeln hergeleitete) Typ von arg bzw. exp konform zu dem (deklarierten) Typ von y ist. Ausnahmen in EIFFELs Typsystem machen allerdings beide diese Fragen noch viel komplizierter.

Wenn wir später Ergänzungen wie Generizität, covariante Redefinitionen und verankerte Typen zum Typsystem machen, lassen sich deren Typregeln alle danach beurteilen, ob sie der Erhaltung der Aufrufkonsistenz a) dienen. Die Zuweisung b) hat dabei eine Hilfsfunktion: wenn alle Variablen stets an konforme Objekte gebunden sind, kann man die a) Frage in der Tat so einfach beantworten wie oben beschrieben: schaue auf den Typ der Variable, dort muss das zu rufende Element stehen.

2.7 Cardellilogie

Es ist schon nicht so einfach, eine fremde Arbeit zusammenzufassen, ohne einfach deren Zusammenfassung abzuschreiben, oder sie aus dem Zusammenhang zu reißen; man muss sich da schon ganz schön zusammen reißen. Ich habe das Buchkapitel von Cardelli (1997) ja hier schon öfter erwähnt, aber noch nicht geschrieben was drin steht. Jetzt, nach 14 Seiten Text, sind wir bereit es knapp auszudrücken: Cardelli beschreibt ein konstruktivistisches Typsystem ähnlich dem von PASCAL und von HASKELL, mit generischen Typen wie in HASKELL, allerdings mit beschränkter Generizität auf der Basis von Subtypen (also wie in EIFFEL, das kommt im nächsten Teilabschnitt). Diese Cardelli'sche Mischung kommt auch der Sprache ML ziemlich nahe; zumindest deren Variante OCAML hat auch Subtypen.

Cardellis Ansatz ist deshalb so wichtig, weil er von den meistens Forschern so verwendet wird. In der Regel baut man Objektorientierung in diesen Ansatz ein, indem man die Objekte einfach wie Records behandelt.

Einige Nachfolgesprachen von PASCAL, wie MODULA-2 und -3 sowie OBERON und COMPONENT PASCAL basieren auch auf diesem Prinzip. Sie brauchen aber stets noch weitere Mechanismen, um die Abstraktion und Kapselung dieser Records zu sichern. Sonst kommt es zu solchen Absurditäten wie bei OBJECT PASCAL, wo die privaten Teile eines Objekts im öffentlichen Teil einer Unit deklariert werden müssen. COMPONENT PASCAL und MODULA-3 (die von Cardelli mitentwickelt wurde) sind übrigens Beispiele für Sprachen, bei denen die Typsysteme sehr komplex sind — und wenn man bedenkt, dass es auch einfacher geht und dass einfacher besser ist, sind deren Typsysteme einfach zu komplex.

Wie wir aber im vorigen Teilabschnitt gesehen haben, ist es einfacher die Typen einer rein objektorientierten Sprache direkt als primitive abstrakte Typen zu definieren. Wenigstens sind aber die wesentlichen theoretischen Ergebnisse vom Record-Modell auf das einfachere Modell übertragbar. Cardelli definiert nämlich Subtypen-Regeln für jedes der Typkonstrukte, die sich auf die Record-Objekte anwenden lassen und das reicht schon um eine triviale, aber wesentlich Schlussfolgerung zu ziehen.

Die Regel für Records lautet: Recordtyp S ist Subtyp von Recordtyp R, falls die Feldnamen von S eine Obermenge der Feldnamen von R sind, und die Typen aller gemeinsamen Felder in S Subtypen des entsprechenden Feldes von R sind. Man kann das in Cardellis Stil als logische Schlussregel schreiben (= steht für „Subtyp von“):

$$S_1 = T_1, \dots, S_n = T_n$$

$$\mathbf{record\ f_1 : S_1, \dots, f_n : S_n \dots f_m : S_m\ end = record\ f_1 : T_1, \dots, f_n : T_n\ end}$$

Dazu nehmen wir jetzt noch die Regel für Funktionen:

$$A = B, C = D$$

$$B \rightarrow C = A \rightarrow D$$

Wir müssen also den Vorbereitungsbereich der Funktion (höchstens) vergrößern ($B = A$) und den Nachbereich verkleinern, damit man eine Funktion wirklich problemlos für eine andere einsetzen kann. Weil beim Argumenttyp das Relationssymbol umgedreht ist, spricht man von Contravarianz. Der Ergebnistyp hingegen ändert sich covariant.

Für Objekttypen zieht man mit Hilfe dieser Regeln die einfache Schlussfolgerung, dass die Elemente von Objekttypen (deren Signaturen man auch als Records auffassen kann) ihre Argumente nur contravariant redefinieren dürfen, um noch ein Subtyp zu sein. Obwohl die Regel so trivial erscheint, stellt sie das Haupthindernis für Meyers Typsystem dar. Man kann sich das auch leicht veranschaulichen: was sollte denn eine Funktion zurückgeben (und was soll eine Prozedur tun), wenn eines ihrer Argumente außerhalb seines Wertebereichs liegt?

2.8 Subtypen und Generizität im Detail

Im letzten Teilabschnitt haben wir gesehen, wie Subtypen bei Cardelli aussehen. In diesem Teilabschnitt werden wir sehen, wie Subtypen in EIFFEL aussehen, wie Generizität in EIFFEL, HASKELL und anderen Sprachen aussieht und wie beschränkte Generizität funktioniert — mit oder ohne Subtypen.

Beginnen wir bei einem der Grundprinzipien der Objektorientierung: *Vererbung erzeugt Subtypen*. Früher war das mal eine tolle Universalregel, heute ist man voller Zweifel. Ganz einfach deshalb, weil Vererbung in vielen Sprachen (so auch EIFFEL) nicht immer Typen erzeugt, die Subtypen nach dem offensichtlichen Cardelli-Kriterium sind. Trotzdem ist in (den meistens) OO Sprachen die Erzeugung von Subtypen an Vererbung gebunden (nur in SMALLTALK zum Beispiel nicht, da gibt es aber auch überhaupt keine Typen).

Auf der streng typisierten Seite ist die Sprache OCAML ein Beispiel für das Cardelli'sche Lager: zwei beliebige, unabhängige Typen können in Subtyp-Beziehung stehen, sofern ihre Struktur das zulässt.

In JAVA erzeugt jede Unterklasse auch Subtypen, in EIFFEL erzeugte früher jede Klasse Untertypen. Mittlerweile gibt es da aber auch die so genannte *nicht-konforme Vererbung*, die keine Subtypen erzeugt. (Im Abschnitt 3 sehen wir, dass EIFFEL auch eine Subtyp-Regel hat, die nicht auf Vererbung basiert.) Der Grund für die Einschränkung der Subtypen auf Vererbung ist ganz einfach, dass man davon ausgeht, Typen können ohne explizite Vererbungsrelation nur zufällig kompatibel sein. Außerdem liefert die Vererbung dem Subtypen gleichzeitig die Implementierung der geerbten Operationen; wenn diese also nicht mutwillig überschrieben werden, erfüllt der per Vererbung erstellte Subtyp also automatisch seine Ersetzbarkeitsverpflichtung!

Spekulationsmodus: Ich schrieb oben schon, dass es ein Vorteil der Subtypen ist, dass man keine speziellen Typvariablen deklarieren muss, sondern jedweden Typ einfach so durch einen Subtyp ersetzen

kann. Hier sehen wir zusätzlich noch, dass man den passenden Subtyp ohne Zusatzaufwand erstellen kann: man fügt einfach die neuen Aspekte hinzu und der zur Konformität notwendige Teil wird einfach geerbt. Diese Einfachheit mag erklären warum die Vererbung stets im Vordergrund der objektorientierten Propaganda gestanden hat, wo doch eigentlich die Module und die Schnittstellen das wichtigste Prinzip sind (siehe OOSC, die ersten beiden Kapitel).

Schnittstellen waren (und sind wohl noch) in den Köpfen der Programmierer immer mit einem hohem Preis verbunden: Man muss erst die Schnittstelle separat beschreiben und jedes Mal muss man sie komplett implementieren. JAVA-Interfaces schaffen es auf elegante Art, beide diese Nachteile miteinander zu vereinen. Nur in EIFFEL werden Schnittstellenbeschreibungen automatisch generiert (mit kompletten Verträgen, die JAVA immer noch fremd sind!) und die Implementierung wird erleichtert, weil man beliebige Teile der Implementierung in Oberklassen faktorisieren kann — alles transparent für den Nutzer der Schnittstelle.

Soviel zu den Subtypen, nun zur Generizität. Diese wird in EIFFEL einfach dadurch realisiert, dass man jede Klasse mit Typparametern versehen kann: den *formal generics*. In der Klasse können die *formal generics* dann verwendet werden, wie normale Typen (zu denen allerdings kein anderer Typ konform ist); aus der Klasse kann man dann verschiedene Typen erhalten, indem man sie mit *actual generics* instanziiert. Die generische Klasse selbst zählt dabei nicht als Typ; es gibt also keine „universellen Typen“ oder „polymorphen Typen“ wie wir sie gleich in HASKELL und Cardellis System finden werden. Trotzdem kann man natürlich innerhalb einer Klasse in der es zum Beispiel den *formal generic T* gibt, einen Typ `SOMETHING[T]` verwenden.

Die generischen Klassen von EIFFEL enthalten auf diese Weise automatisch generische Routinen — damit ist das System fast so ausdrucksstark, wie eines bei dem Datentypen und Funktionen separat generisch getypt werden. Leider führt, wie wir noch sehen werden, gerade dieser Mangel an Ausdrucksstärke zu einem der Probleme der Covarianz.

Der wesentliche Unterschied zwischen der Generizität à la EIFFEL und der in HASKELL und bei Cardelli ist, dass die Geltungsbereiche von Variablen bei letzteren viel kleiner sind: bei Cardelli gibt es explizite Quantoren für die Typvariablen innerhalb eines Typs, in HASKELL stehen implizite Quantoren vor jedem Typ, so dass der Geltungsbereich einer Typvariable jeweils ein Typ ist. In den beiden Beispielen von oben ...

```
Map :: (a -> b) -> [a] -> [b]
repeat show :: [a -> String]
```

... bezeichnen die Typvariablen `a` in den beiden verschiedenen Typausdrücken auch verschiedene Typen. (Das Zeichen `::` steht dabei für „hat den Typ“.) Durch die impliziten Quantoren sind die Typen `[a]` und `[b]` auch dasselbe (außer sie kommen als Teile eines einzigen Typs vor).

Dieser Unterschied scheint theoretisch nichts auszumachen (formal wurde der Unterschied meines Wissens noch nicht untersucht), es spielt aber für die Praxis eine große Rolle, weil es die Lesbarkeit und den Deklarationsaufwand wesentlich mitbestimmt. Auch wenn es also auf die Semantik keinen Einfluss hat, kann es doch für die Akzeptanz verschiedener Lösungen entscheidend sein. Man muss eben in EIFFEL die *formal generics* nur ein einziges Mal pro Klasse deklarieren.

Zunächst erscheinen die Konzepte ähnlich, denn sie erlauben beide das Schreiben von Code für verschiedene Typen und im Gegensatz zu Polymorphismus durch Subtypen tun sie dies beide durch explizite Typvariablen. Trotzdem gibt es einen einfachen, fundamentalen Unterschied, der zwei verschiedene Begriffe rechtfertigt.

Der Unterschied zwischen Generizität und parametrischem Polymorphismus

Generizität parametrisiert nur Code, aber parametrischer Polymorphismus erlaubt parametrisierte Typen für alle Werte.

In den funktionalen Sprachen, in denen der parametrische Polymorphismus meist eingesetzt wird, sind Funktionen Werte der Programmiersprache, sie haben also Typen wie andere Werte auch. Dadurch werden die Funktionen generisch. Aber andere Werte können auch polymorph sein: so hat die leere Liste den Typ „ $\forall X$ •Liste von X“; in Haskell `empty_list :: [a]`.

In imperativen Sprachen hingegen spricht man von Routinen *signaturen* und von *Klassenschnittstellen* und diese sind nicht Bestandteil des Werte- und Typsystems. In gewissem Sinn macht das die imperativen Sprachen einfacher: man hat Typvariablen nur, wo man sie am meisten braucht: im Code. Und wenn nur die Klassen generisch sind, hat man zusätzlich noch die Vereinfachung, dass die Typparameter bei Benutzung immer explizit gemacht werden. Bei generischen Routinen setzt man bereits eine Art von Typinferenz ein, weil dort die explizite Angabe

der *actual generics* keinen Dokumentationsgewinn bringt, sondern eher lästiges Rauschen erzeugt: die Typen lassen sich leicht aus den Routinenargumenten ablesen, und genau das formalisiert die Typinferenz.

Sprachen mit generischen Routinen stehen direkt an der Grenze zum parametrischen Polymorphismus, wenn sie Mechanismen wie Funktionszeiger oder Agenten besitzen, die Funktionen doch noch zu Werten machen. `GENERICJAVA` hat sich aber explizit dagegen entschieden: Wenn der allgemeinste Typ ein parametrischer Typ wäre, gibt die Typinferenz diesen nicht an, sondern der Programmierer muss selbst einen nicht parametrisierten Typ angeben.

2.9 Beschränkte Generizität

Hierzu zunächst ein Überblick, wie er sonst nirgends in der Literatur zu finden ist:

Drei Arten der beschränkten Generizität	
1. Beschränkung durch explizite Angabe benötigter Operationen.	ADA, POLYJ
2. Beschränkung durch Subtypen.	EIFFEL, GENERICJAVA, Cardellis System
3. Beschränkung durch Typklassen, bzw. Match-Typen.	HASKELL, LOOM

Die explizite Beschränkung hat jüngst durch die Ablehnung von `POLYJ` (oder besser gesagt: die Annahme von `GENERICJAVA`) eine Abfuhr erhalten. Sie hat die Nachteile, dass sie zu sehr langen Deklarationen führt, und dass sie wenig robust gegen Erweiterungen und Änderungen ist, weil diese an den Schnittstellen explizit gemacht werden müssen. Kurz: sie bietet zu wenig Abstraktion. Deswegen soll dies auch schon ihre letzte Erwähnung in dieser Arbeit gewesen sein.

Schauen wir uns mal die Beschränkung durch Subtypen näher an. Sie funktioniert, indem man jedem Typparameter (*formal generic*) bei der Deklaration einen geforderten Supertypen zuordnet. In `EIFFEL` gibt es dafür die Syntax `[T -> TC, ..]` wobei `T` der *formal generic* ist und `TC` ist ein Typ (keine Klasse!). Innerhalb der Klasse, die diese Deklaration erhält kann man dann davon ausgehen, dass der Typ `T` ein Subtyp von `TC` ist. (Aber nicht umgekehrt, ich schrieb ja eben schon, dass nur `T` selbst

zu sich konform ist.) Dadurch kann man dann alle von `TC`s Operationen mit `T` verwenden und `T` „überall hinschicken“ wo auch ein `TC` akzeptiert würde. Falls man keine Beschränkung angibt, wird automatisch der Typ `ANY` als Beschränkung angenommen. Dadurch wird sichergestellt, dass stets jeder Typ zu `ANY` konform ist. (Andernfalls ist `T` dies indirekt, da `TC` auch schon zu `ANY` konform sein muss.) Wir sehen hier, wie sich die beschränkte Generizität in die Sprache einfügt und auf den bereits bestehenden Typeregeln und dem Konzept der Konformität aufbaut.

Bei der dritten Art der beschränkten Generizität habe ich verschiedene Mechanismen zusammengefasst, die unabhängig voneinander entwickelt wurden. Dass diese Mechanismen wirklich äquivalent sind, behaupte ich hier zum ersten Mal, für eine Erklärung sind wir erst in Abschnitt 7 bereit. Bis dahin kann ich aber schon mal die `HASKELL`'schen Typklassen vorstellen: dies sind Mengen von Bezeichner-Typ-Paaren, wobei eine der vorkommenden Typvariablen besonders gekennzeichnet ist. Ein einfaches Beispiel:

```
class Eq a where
  (==) :: a -> a -> Bool
```

Man kann dann einen Typ zu einer Instanz einer Typklasse machen, indem man die geforderten Funktionen implementiert. Das kann zum Beispiel so aussehen:

```
instance Eq Boolean where
  True == True = True
  False == False = False
  _ == _ = False
```

Um dieses Beispiel zu verstehen, muss man wissen, dass hier der Datenzugriff mittels `case` ersetzt wurde durch den syntaktischen Zucker rekursiver Gleichungen und Pattern Matching: auf der linken Seite der Gleichungen steht ein Muster, und auf der rechten Seite steht das passende Funktionsergebnis. Das einfache `=` steht für die Funktionsdefinition, während das doppelte `==` für die zu definierende Funktion steht.

Ob man diesen Code versteht oder nicht, man kann ihn in jedem Fall sicher anwenden:

```
? 1 == 1
True :: Bool

? "Hinz" == "Kunz"
False :: Bool
```

```
? 1 == "Hinz"
error: TODO

? \x -> x == x
<<function>> :: Num a ==> a -> Bool
```

Zwischenbilanz

Weil die (beschränkte) Generizität später noch eine wichtige Rolle spielen wird, hier eine kurze Nennung von Vor- und Nachteilen.

Für eine objektorientierte Sprache scheint die Beschränkung für Subtypen die beste Lösung zu sein, denn sie passt prima zum Rest der Sprache, was sich im günstigen Fall in einfacheren Typregeln äußert. Allerdings scheinen Subtypen allein noch nicht auszureichen, da die Programmiersprache zusätzlich noch zu oft auf Typzwänge (*type casts*) verweist (wie JAVA), oder auf zwielichtige Methoden wie Kovarianz und verankerte Typen zurückgreifen muss (wie EIFFEL).

HASKELLS Typklassensystem hat den bereits erwähnten Nachteil, dass Typen und Typklassen getrennte Konzepte sind, was die Akzeptanz bei faulen Programmierern wesentlich erschwert, denn man kann nicht mehr einfach Unterklassen von allen Typen bilden. Der Vorteil der Typklassen ist allerdings, dass sie in der Praxis alle Anforderungen zufrieden stellen, ohne dass besondere Ausnahmen nötig sind! (Möglicherweise wurde auf der Welt noch nicht so viel HASKELL Code geschrieben, wie EIFFEL Code, oder noch nicht so große Projekte. Trotzdem und weil HASKELL grundlegende Beispiele meistert, bin ich hier zuversichtlich.)

Als ich jünger war, habe ich auch eine Ähnlichkeit zwischen Javas Interfaces und den HASKELL'schen Typklassen gesehen, weil beide im Kontrast zu konkreten Typen stehen und nicht mehr enthalten als die Typen einiger Funktionen bzw. Routinen. Wie ich aber eben schon schrieb, sind die Interfaces nur ein schlechter Abklatsch von allgemeinen Klassen. Auch bei der hier behandelten Problematik verhalten sie sich genau wie (objektorientierte) Klassen und nicht wie Typklassen. Übrigens erlauben es die HASKELL'schen Typklassen genau wie Klassen in EIFFEL und im Gegensatz zu JAVA Interfaces, einige (oder alle) Implementierung in die Typklasse zu faktorisieren (natürlich, ohne dass sie dadurch gleich Teil der Schnittstelle werden, für den Nutzer bleibt es transparent).

Der Leser ist jetzt bestimmt schon ganz gespannt, wie wohl ein Kompromiss zwischen den beiden Ansätzen aussehen mag. Bevor ich aber dazu komme, sollte ich erstmal die Probleme vorstellen, die damit gelöst werden sollen...

3 EIFFELS drei kleine Probleme

Alle drei Probleme in EIFFEL sind Typregeln, die es erlauben einen Typ A als konform zu B zu bezeichnen, obwohl A nicht Subtyp von B nach Cardellis konstruktivistisch-struktureller Definition ist. Zwei der Probleme erzeugen den angeblichen Subtyp durch Vererbung, das dritte Problem erzeugt ihn durch eine explizite Typregel. Keine der Typregeln finden sich in anderen mir bekannten Programmiersprachen oder Typtheorien, sie alle wurden in EIFFEL eingeführt, weil sie (angeblich) für die Praxis der objektorientierten Programmierung unerlässlich sind, aber dazu mehr in Teilabschnitt 7.4.

Die drei Probleme sind:

Exporteinschränkungen: Eine Unterklasse bietet ein Element des Supertyps nicht mehr an. Was soll passieren, wenn das Element über die Schnittstelle des Supertyps aufgerufen wird?

Konformität generischer Typen: Die folgende Regel führt manchmal zur Konformität von Nicht-Subtypen...

$$A_1 = B_1, \dots, A_n = B_n$$

$$T[A_1, \dots, A_n] = T[B_1, \dots, B_n]$$

Kovarianz: Argumente einer geerbte Routine können covariant (d.h. einschränkend) redefiniert werden. Was soll passieren, wenn die Routine über die Schnittstelle des Supertyps mit einem Argument aufgerufen wird, das die Subtyp-Variante nicht mehr akzeptiert?

Ich habe die Probleme absichtlich mit der „Was soll passieren ...?“-Frage charakterisiert, obwohl es ja hier nicht um Programmiersprachen-Semantik, sondern nur um Typen geht. Hätten wir semantische Gestaltungsbefugnis, könnten wir auf diese Frage einfach „Gar Nichts.“ antworten und das Problem wäre gelöst!

Eine bessere Formulierung wäre also „Wie können wir verhindern, dass...?“. Dann aber wäre die einfachste Antwort: „einfach die Typregel abschaffen, die das Problem verursacht!“ So naiv wie das klingt, kommen dem doch die gleich vorgestellten Lösungen sehr, sehr nahe. Dieser Weg wird gangbar, weil entweder die Typregel durch eine ähnliche ersetzt wird, die dasselbe erreicht (dies wird bei den generischen Typen passieren), oder weil die Anwendungsfälle der alten Regel auch mit anderen Mechanismen bearbeitet werden können. Entweder durch besseren Entwurf mit schon vorhanden Mechanismen (Exporteinschränkung

und ähnliche Anwendungsfälle der Kovarianz), oder durch neue Mechanismen (insbesondere der Anwendungsfall *peer routine* der Kovarianz).

Die beiden ersten Probleme wurden meines Erachtens in der Literatur schon hinreichend gelöst, ich gebe die Lösungen gleich zusammen mit der Problemvorstellung in diesem Abschnitt an. Die Kovarianz hingegen macht von ihrer Vorstellung an den ganzen Rest dieser Arbeit voll.

3.1 Exporteinschränkung (descendant hiding)

Dieses Problem ist so einfach, dass es schon wieder schwierig ist. Ich mache es am besten mal anhand seines kanonischen Beispiels konkret:

```
class VOGEL feature {ANY} flieg is ... end

class STRAUSS inherit VOGEL export {NONE} flieg end ... end

voegel : LIST[VOGEL]
...
voegel.add(create {STRAUSS})
voegel.do_all(agent flieg)
```

Wenn wir uns hier die von den Klassen erzeugten Typen anschauen, so sehen wir, dass `VOGEL` ein Element `flieg` hat, aber sein Untertyp `STRAUSS` hat dieses Element nicht. William Cook (1989) hat sich bei Betrachtung dieses Problems gedacht: „Wenn wir den `STRAUSS` über eine `VOGEL`-Variable ansprechen, können wir ja `flieg` in jedem Fall aufrufen. Also ist es sinnlos, der Klasse `STRAUSS` zu erlauben, `flieg` nicht zu exportieren und diese Export-Einschränkungsmöglichkeit sollte abgeschafft werden.“

Leider ist die Sache nicht so einfach, denn Meyer argumentiert genau anders herum: Weil `flieg` nicht vom `STRAUSS` exportiert wird, sollte es auch nicht aufrufbar sein, wenn der `STRAUSS` über eine `VOGEL`-Variable angesprochen wird. Der Grund ist, dass nicht-exportierte Routinen nicht zur Erhaltung der Invariante verpflichtet sind; wenn man eine solche von außen aufruft, kann man die Invariante beschädigen. Hier sehen wir, wie eng Typisierung und Kapselung zusammen wirken. Das nächste Beispiel verdeutlicht das etwas besser:

```
class POLYGON feature {ANY}
  add_vertex(...) is ...
    ensure num_vertices = old num_vertices + 1
  end
  ...
end

class RECTANGLE inherit POLYGON export {NONE} add_vertex end
invariant num_vertices = 4
end
```

Hier sehen wir ganz klar: die Prozedur `add_vertex` verletzt die Invariante des `RECTANGLE`, sie darf unter keinen Umständen aufgerufen werden, über welche Schnittstelle auch immer.

Die meisten Arbeiten über objektorientierte Typsysteme behandeln dieses Thema nicht, außer Cook gehen meines Wissens nur noch Meyer und ich darauf ein. Meyer hat das Problem zunächst über die erweiterte Typanalyse gelöst (siehe Abschnitt 4), aber in seiner jüngsten Lösung (2003) schließt er sich Cook an: Exporteinschränkungen werden einfach verboten. Ganz so hart ist es allerdings nicht, weil EIFFEL in der Zwischenzeit auch nicht-konforme Vererbung unterstützt: man kann (mit Hilfe des `expanded` Schlüsselwortes in der Elternliste einer Klasse) angeben, dass eine Unterklasse keine Subtypen der Typen seiner Elternklasse erzeugt. Dementsprechend können nicht-konforme Unterklassen Exporteinschränkungen durchführen soviel sie wollen. Meyer zeigt, dass dies besonders bei der Implementierungsvererbung den praktischen Anforderungen entspricht: wenn man ein Elternteil nur hat, um dessen Elemente im Eigengebrauch aufzurufen, dann erbt man natürlich nicht-konform und exportiert keines der geerbten Elemente. Ein Beispiel dafür findet sich in Meyers Artikel.

In meiner Arbeit (2003) schließe ich mich dieser Lösung, denn sie passt in die von mir interpretierte Theorie des vertraglichen Programmierens. Vereinfacht gesagt: wenn eine Klasse ein Element anbietet, dann müssen das auch alle ihre Unterklassen anbieten. Andernfalls könnte sich ja kein Nutzer der Klasse mehr darauf verlassen, dass überhaupt ein angebotenes Element verfügbar ist.

Meyers Abkehr von der alten Regel der Exporteinschränkung ist wegweisend, denn die Begründung der Abschaffung steht in krassem Widerspruch zur ursprünglichen Begründung der Einführung der Regel. 1995 schreibt Meyer noch: „Die Möglichkeit der Exporteinschränkung wird der menschlichen Unfähigkeit zur perfekten Klassifizierung gerecht. Zu jeder Regel gibt es Ausnahmen und dies muss sich in der Programmiersprache widerspiegeln.“ Vertragliches Programmieren basiert aber

gerade darauf, dass alle Ausnahmen vorher deklariert werden: entweder die Routine flieg bekommt eine Vorbedingung, oder man trennt sorgsam in VOGEL und FLIEGENDER_VOGEL. Wir werden im Teilabschnitt 7.4 noch sehen, warum dies nicht zu der von Meyer befürchteten Explosion der Klassen oder Vorbedingungen führt. Kurz gesagt liegt es daran, dass die Software ja gar nicht verpflichtet ist, die Anwendungswelt zu vollständig zu modellieren, es kommt vielmehr darauf an, dass die funktionalen Anforderungen erfüllt werden.

Meyer hat mit der Abschaffung der Exporteinschränkung einen entscheidenden Paradigmenwechsel vollzogen; er beginnt dem Prinzip treu zu werden, dass er in seinen Büchern selbst aufgestellt hat, nämlich dass die passende Vererbungshierarchie eines Projektes der Funktionalität folgen muss. Konformität in EIFFEL rückt näher ihren formalen Grundlagen — und dieses Prinzip wende ich auch auf die anderen beiden Probleme an.

3.2 Generische Konformität

Ich habe ja am Anfang des Abschnitts schon die problematische Konformitätsregel genannt. Hier möchte ich zunächst mit einem Beispiel die Regel motivieren und auch gleich ihre Schwachstelle aufzeigen. Die Regel hat ja keinerlei Entsprechung in Cardellis formalem System: zwei Typen, die auf demselben generischen Typ basieren (bzw. derselben Klasse in EIFFEL) können zwar in Subtyp-Beziehung stehen, dies ist aber nicht zwangsläufig so, nur weil die generischen Parameter (*actual generics*) in Subtyp-Beziehung stehen.

Die intuitive Motivation für die Regel ist, dass man eine Liste von Störchen selbstverständlich auch als eine Liste von Vögeln auffassen kann (man kann sie zum Beispiel alle fliegen lassen). Der intuitive Fehler an der Regel ist, dass man natürlich in die Liste von Störchen keinen allgemeinen Vogel einfügen kann (dann ist es ja keine Liste von Störchen mehr). Die intuitive Lösung des Problems ist es, das Einfügen in eine solche Liste einfach zu verbieten. Genau dies ist auch das Resultat der hier vorgestellten Lösung und auch der erweiterten Typprüfung von Abschnitt 4.

In meiner anderen Arbeit (Abschnitt C) diskutiere ich das Problem relativ ausführlich, insbesondere die Anwendungsfälle dieser Konformitätsregel. Hier möchte ich mal näher auf die formalen Aspekte eingehen, denn mit Bezug auf Cardellis System lässt sich eine einfache Lösung und ein gutes Verständnis für das Problem finden.

Wir haben ja schon gesehen, dass in Cardellis System beliebige Typen in Subtyp-Beziehung stehen können, sofern sie strukturell kompatibel sind. Ein Kunde im Schuhladen mit Name und (Schuh-)Größe kann Subtyp von jedem Typ sein, der nur eine Größe (von konformem Typ) hat. In OO Sprachen hingegen sollen Subtypen immer auf durch Vererbung verwandten Klassen beruhen, man möchte ja nicht dass Schuhkäufer plötzlich zu Schrauben, Schriftarten oder anderen Dingen mit einer Größe kompatibel sind. (Außerdem führt diese Regel auch zu einer hohen Flüchtigkeit: allein durch kleine Änderungen können die Subtyp-Beziehungen entstehen oder beendet werden.) Die Regel der konformen Generizität macht nun von diesem Prinzip eine Ausnahme. Erstens, weil ja Typen, die auf derselben Klasse basieren immer noch hinreichend verwandt sind, und zweitens weil, wie wir gesehen haben, diese Konformität auch sehr nützlich ist. (Über die Gründe kann ich allerdings nur spekulieren, weil Meyer die Regel anscheinend einfach nur als Unterpunkt der großen Konformitätsregel angegeben hat, ohne sie einzeln zu diskutieren, oder überhaupt zu erwähnen, dass sie etwas Besonderes ist.)

Wenn wir also den Wunsch haben, die Konformität etwas weiter aufzufassen, als nur über Vererbung, und der erste, naive Ansatz schlägt fehl, dann können wir uns doch eigentlich direkt an Cardelli wenden und fragen: Wir wollen erlauben, dass $C[T_1, \dots]$ ein Subtyp von $C[S_1, \dots]$ wird; wann geht das nach Deinen Regeln? Um die Antwort zu finden, können wir uns ganz einfach die entsprechenden Typen als Records anschauen, und nach den Cardelli'schen Strukturregeln vergleichen. Natürlich ist es einem Benutzer nur schwer zuzumuten, sich jedes Mal die beiden kompletten Typen anzuschauen, nur um zu entscheiden, ob der eine ein Subtyp vom anderen ist. Außerdem könnte sich auch hier eine hohe Flüchtigkeit einstellen. Hier können wir aber von der Information Gebrauch machen, dass beide Typen ja auf derselben Klasse basieren; wenn wir diese Klasse schon vorher analysieren können wir in der Tat ganz einfach über die Konformität entscheiden, indem wir nur anschauen, ob (und in welcher Richtung) die S_i und T_i zueinander konform sind.

Es stellt sich heraus, dass wir aus der Klassenschnittstelle zu jedem der Typparameter eine kleine Information herauslesen können: gleich werden wir formal definieren, wann Typparameter *covariant*, *contravariant* oder *nichtvariant* in der Klassenschnittstelle ist. Mit dieser Information stellt sich die neue Typregel wie folgt dar:

Sei $C[G_i]$ eine generische Klasse mit *formal generics* G_i (für $i = 1..n$).

Seien T_i und S_i Typen, wobei für jedes $i = 1..n$ entweder ($T_i = S_i$ und C ist covariant in G_i) oder ($S_i = T_i$ und C ist contravariant in G_i) gilt.

Dann gilt: $C[T_i] = C[S_i]$

Diese Regel sieht zwar kompliziert aus, sie ist aber immer noch relativ einfach, weil man sich alle Typparameter einzeln anschauen kann. Und falls C nichtvariant in einem der G_i ist, braucht man die Typen T_i und S_i überhaupt gar nicht erst anschauen, Konformität kommt dann für keinen der auf C basierenden Typen in Frage. Wie 1989 schon William Cook bemerkte, ist EIFFEL's alte Regel ein Spezialfall davon, der annimmt alle Klassen sind covariant in allen Typparametern, was natürlich völlig haltlos ist.

Schauen wir uns nun mal an, wie wir die Varianz der Typparameter definieren müssen, damit obige Regel auch aufgeht. Im Prinzip müssen wir nur die zusätzlichen Voraussetzungen finden, die nötig sind, damit unsere Regel eine logische Folgerung aus Cardellis Regeln ist.

Einfach sind dabei die Fälle, wenn der Typparameter, nennen wir ihn T , direkt in einer Signatur auftaucht. Findet sich dabei T nur als Typ von Argumenten einer Anfrage oder eines Befehls, so ist die Klasse contravariant in T , denn eine Ersetzung von T durch einen Supertypen macht die Klassensignatur (gesehen als Record) zum Subtypen. Umgekehrt, wenn T nur als Typ von Ergebnissen von Anfragen vorkommt, so ist die Klasse covariant in T . Nur wenn T überhaupt nicht in der Schnittstelle vorkommt, ist die Klasse sowohl contra- als auch covariant in T . Dies mag zunächst als nutzlose Beobachtung erscheinen, aber wenn wir später mal implizite Typparameter einführen (die in der Schnittstelle nicht vorkommen), ist es sehr nützlich, denn es sorgt dafür dass diese „interne Generizität“ die Nutzer der Klasse nicht betrifft.

Nun betrachten wir die Fälle, dass T in der Schnittstelle als *actual generic* einer benutzten Klasse vorkommt. Hier gilt: wenn diese Klasse covariant in T ist, und sie erscheint in covariantem Kontext (also so dass ein covarianter Typparameter dort auftauchen könnte), dann bleibt die Klasse covariant in T ; und wenn die Klasse des parametrisierten Typs contravariant in T ist, muss sie in contravariatem Kontext auftauchen, damit die Klasse covariant in T ist. Das umgekehrte gilt wieder, um die Klasse contravariant in T sein zu lassen.

Zuletzt bleibt der Fall, dass T als generischer Parameter der gerade betrachteten Klasse selbst auftritt. (Beispiel folgt sofort.) Hier wissen wir ja noch nicht, ob diese Klasse co- oder contravariant in T ist, wir wollen es ja gerade herausfinden. Man kann es auch herausfinden, indem man einfach annimmt, die Klasse wäre covariant (bzw. contravariant) in T . Wenn man dann auch herausbekommt, dass die Klasse covariant (bzw. contravariant) ist, so ist das auch so. Falls man das Gegenteil herausbekommt (einschließlich Nichtvarianz), schlägt die Annahme fehl, die Klasse ist nicht covariant (bzw. contravariant).

Nehmen wir uns gleich mal den schwierigsten Fall:

```
class LIST[T] feature
  append ( other : LIST[T] ) is ...
end -- class LIST[T]
```

Obwohl der Parameter T nur ein einziges Mal in der Klasse vorkommt, ergibt die Analyse, dass die Klasse nichtvariant in T ist. Weder die Annahme der Co- noch Contravarianz bestätigt sich. Intuitiv liegt das daran, dass die Konformität bei Anwendung der Klasse zweimal zu schlagen kann: für Objekte auf denen wir die Routine rufen und für die Argumentobjekte. Nehmen wir mal an $A = B$ und die Klasse wäre covariant:

```
la, lb : LIST[B]
...
create {LIST[A]} la -- legal da LIST[A] = LIST[B]
la.append(lb) -- fügt Bs in A-Liste
```

Andererseits, falls die Klasse contravariant wäre:

```
la : LIST[A]
lb : LIST[B]
...
la.append(lb) -- legal, da LIST[B] = LIST[A]
-- fügt aber Bs in A-Liste
```

Ein positives Beispiel für eine rekursive Klasse ist die folgende. Sie ist covariant in ihrem Typargument und kann folglich sorgenfrei verwendet werden:

```
class SEQUENCE[T] feature
  tail : SEQUENCE[T] is ...
end -- class SEQUENCE[T]
```

Wir haben hier also gesehen, dass man die alte EIFFEL'sche generische Konformität durch eine einfache und formal sichere Typregel erset-

zen kann. In meiner anderen Arbeit zeige ich, dass die neue Typregel praktisch auch ausreicht: sie arbeitet gut mit *immutable data structures* (unveränderlichen bzw. funktionalen Datenstrukturen) und bei EIFFELS Agententypen führt es sogar genau zu der Contravarianten Konformität, die von den SmartEIFFEL-Machern schon separat vorgeschlagen wurde (leider ist mir keine Veröffentlichung bekannt). Dies könnte letztendlich ein Hauptargument für die neue Regel sein: dass man sich nämlich zusätzliche Regelungen für die Agenten spart.

Natürlich kann man auch dem benötigten zusätzlichen Attribut einer Klassenschnittstelle (nämlich der Information über Co- bzw. Contravarianz) eine gewisse Flüchtigkeit unterstellen: nur durch eine kleine Änderung der Klasse kann sich die Varianz eines ihrer Typparameter ändern. Dasselbe trifft aber auch auf den *verzögert / wirksam* Status (*effective / deferred*) zu: mir nur einem verzögerten Element wird die ganze Klasse verzögert. Der Punkt ist, dass der Status ein verbindlicher Teil der Klassenschnittstelle ist. Dadurch wird sich der Programmierer bewusst, wie wichtig diese kleinen Änderungen für die Nutzer der Klasse sind und er hat ein Interesse daran, den Zustand zu erhalten.

3.3 Covarianz

Wir haben jetzt schon zwei von EIFFELS drei kleinen Problemen gelöst, indem wir die Sprache in Übereinstimmung mit den Cardelli'schen Typregeln brachten. Die Covarianz ist weit komplexer als die anderen beiden Probleme, nach neuesten Erkenntnissen liegt das an der Vermischung so vieler Anwendungsfälle im selben Mechanismus, die wir erst im Teilabschnitt 7.4 entwirren. Hier will ich deshalb erstmal nur den Anwendungsfall vorstellen, auf welchen sich die meisten Vorschläge in anderen Arbeiten beziehen.

```
class EQ feature
  is_equal ( other : like Current ) : BOOLEAN is ...
end -- class EQ

class TRAVERSABLE[T] inherit EQ feature ...
end -- class TRAVERSABLE[T]

class LIST[T] inherit TRAVERSABLE[T] feature ...
end -- class LIST[T]
```

Zugegeben, eine Klasse `EQ` braucht man in EIFFEL gar nicht, weil `is_equal` schon in der Klasse `ANY` drin ist. Wir haben aber in der Praxis dasselbe Problem mit der Klasse `COMPARABLE`, nur dass diese wegen ihrer größeren Elementzahl nicht so gut als Beispiel taugt.

Diese simplen Deklarationen reichen schon, um unser Problem zu erläutern, denn der Typ `like Current` steht in jeder Klasse für die Klasse selbst — er führt also zu einer automatischen covarianten Redefinition des Arguments `other` der Routine `is_equal`: in `TRAVERSABLE` hat `other` also den Typ `TRAVERSABLE[T]` und in `LIST` hat es den Typ `LIST[T]`. Weil die Typen in EIFFEL trotzdem zueinander konform sind, führt folgender Code zu einem Fehler:

```
t, l : TRAVERSABLE[T]
...
create t
create {LIST[T]} l
l.is_equal(t)
```

`l` ist eine Liste, aber wir sprechen sie über die Schnittstelle `TRAVERSABLE` an, dadurch können wir ihrer `is_equal` Routine ein anderes `TRAVERSABLE` übergeben, obwohl sie eigentlich nur andere Listen akzeptiert. Der Aufruf führt also dazu, dass die Variable `other` an ein nicht-konformes Objekt gebunden wird — der Anfang einer Katastrophe.

Mehr müssen wir erstmal über die Covarianz nicht wissen, um die nachfolgenden Lösungsangebote zu verstehen. Ich möchte hier aber noch auf einen Umstand hinweisen, der für die Covarianz zu sprechen scheint, obwohl diese doch sonst ganz den harten Cardelli'schen Subtypen-Regeln widerspricht. Wenn man es nämlich erlaubt in einer Unterklasse Routinenargumente covariant zu redefinieren, Funktionsergebnisse können sowieso covariant redefiniert werden, und die Klasse selbst ist auch ein Subtyp ihrer Elternklasse, dann kann folglich alles, was überhaupt redefiniert werden kann, covariant redefiniert werden. Als Konsequenz gilt dies dann auch für die verankerten Typen, da diese sich entweder auf `Current`, auf ein Routinenargument oder ein Anfragergebnis beziehen. Daraus folgt wiederum, dass verankerte Typen an jeder dieser Stellen problemlos verwendet werden können, weil sie in Unterklassen ja höchstens covariant redefiniert wurden! Dies ist ein erfreulicher Umstand, der die Typregeln erheblich vereinfacht (aber leider fand ich ihn bisher nirgendwo erwähnt).

Aber auch ein Argument gegen die Covarianz, oder besser: gegen verankerte Typen, will ich hier gleich angeben. Obwohl diese zunächst ganz einleuchtend erscheinen, stellt sich heraus, dass ihre Semantik relativ komplizierten Regeln unterliegt, und zwar für jeden der drei Typen verschiedenen! Ein `like argument` Typen verhält sich zum Beispiel innerhalb der Routine anders als außerhalb. Bei den `like query` Typen ist die genaue Semantik sogar strittig. Ein Teil der Arbeit wird dem-

zufolge auch mit der Semantik der verankerten Typen beschäftigt sein. Kann man vielleicht sogar ganz ohne sie auskommen?

4 Vertragsstreitigkeiten: bisherige Lösungen

In diesem Abschnitt möchte ich die drei Lösungen vorstellen, die Bertrand Meyer im Laufe der Zeit vorgestellt hat und am Ende noch eine, die in zwei EIFFEL-Varianten verwendet wird. Diese lösen sozusagen alle drei Probleme (auch die bereits im vorigen Abschnitt anders gelöst), nicht nur die Kovarianz.

Wenn man die Typen als primitive Verträge auffasst, so kann man sagen, dass sich die Vorschläge in diesem Abschnitt damit beschäftigen, Verträge zu interpretieren: Wer ist für Fehler verantwortlich? Wie verhindern wir Betrug? Die Lösungen in den folgenden Abschnitten beschäftigen sich damit, Verträge flexibler zu machen: Welcher Kompromiss zwischen Kunde und Anbieter ist möglich? Welche statisch prüfbareren Vertragsmuster gibt es?

Die ersten beiden Lösungen in diesem Abschnitt stammen jeweils aus ETL2 und OOSC2 und wurden beide lange Zeit in der Praxis eingesetzt: die eine ab erstmaliger Lösung des Problems, die andere bis Heute. Beide Lösungen basieren auf einer erweiterten Typprüfung, die nach fehlerhaften polymorphen Aufrufen sucht. Es bleiben also die Typminen im Programm, und die Analyse soll sicherstellen, dass niemand drauf tritt.

Die dritte Lösung wurde in einer Arbeit von Colnet und Liquori vorgeschlagen, und auch im SMARTEIFFEL Compiler implementiert. Aber nur experimentell und nicht in der „normalen“ Version. SATHER verwendet, soweit ich weiß, eine ähnliche Lösung.

Die vierte Lösung wurde erst im letzten April vorgestellt und war für mich der Anlass zur Beschäftigung mit dem Problem. Die Lösung wird zurzeit im ECMA-Gremium zur Standardisierung von EIFFEL diskutiert, kam in *comp.lang.EIFFEL* vorbei, ohne dass sich jemand näher damit beschäftigt hätte; und was sonst noch so für Forschungen laufen, weiß ich nicht. Im Vergleich mit den Typprüfungen dreht sie den Spieß radikal um: die Verantwortung zur Vermeidung von Typfehlern wird jetzt vollständig auf die Autoren der Anbieter-Klassen geschoben. Das neue **recast** Konstrukt in der Programmiersprache soll ihnen dabei helfen.

4.1 Konformitätslücke — Die erweiterte Typprüfung

Von allen hier vorgestellten und nicht vorgestellten Lösungen der EIFFEL'schen Probleme ist die Erweiterte Typprüfung im Prinzip die einzige, die wirklich in der Praxis verwendet wurde, und dies nicht zu wenig: sie ist seit über zehn Jahren Bestandteil im EIFFEL-Standard. Währenddessen wurden andere Lösungen oft nicht einmal als Regeln einer Programmiersprache spezifiziert, oder nicht implementiert, oder sie verließen dann das wissenschaftliche Labor nicht. Nur in SATHER wurde noch relativ viel Code geschrieben (aber wohl nicht dieselbe Größenordnung wie in EIFFEL). HASKELL verfügt auch über große „praktische Erfahrung“, aber ihre konzeptuelle Distanz zur Objektorientierung ist dafür auch sehr gering. (Andere hier erwähnte Programmiersprachen haben auch noch viel Code, eine ganz bestimmte hat sogar bald mehr als alle anderen auf der Welt — aber von diesen Sprachen wollen wir ja für EIFFEL nicht lernen.)

Die Idee ist einfach: man findet sich damit ab, dass zwischen manchen Typen eine Konformitäts-Beziehung besteht, obwohl sie keine Subtypen im Cardelli'schen Sinn sind. Diese kleinen Inkonsistenzen toleriert man, solange es **a)** niemals zum Aufruf eines nicht existierenden Elements kommt, und **b)** niemals ein nicht-konformes Objekt an eine Entität gebunden wird. Letzteres könnte man auch tolerieren, aber man geht davon aus, dass dies schon ein unerwünschter Zustand ist, der nur durch einen Programmfehler entstehen kann. Außerdem ist es von diesem Zustand viel zu „kurz“ bis zu einem nicht-existierenden Aufruf. Hätten wir in b) „Cardelli'scher Subtyp“ an Stelle von „EIFFEL'scher Konformität“, würde die Aussage a) implizieren. Da wir das nicht haben, muss die erweiterte Prüfung die Differenz von a) und b) wettmachen.

Zunächst wurde dazu eine Methode verwendet, die ich *feine Analyse* nenne (der Algorithmus aus ETL2). 1995 hat Meyer dann die *grobe Analyse* vorgestellt, die in OOSC2 einging und Heute noch in EIFFEL verwendet wird.

Feine Analyse: Um solche Aufrufe und Parameterübergaben zu verhindern, muss der erweiterte Prüfer einfach ein bisschen genauer hinschauen, welche Objekt-Typen sich wirklich hinter einer Variable verbergen können, wenn ein potentiell gefährliches Element aufgerufen wird. Er wird also zu jeder Variable die Menge der möglich enthaltenen Typen feststellen (diese vergrößert sich mit jeder Zuweisung und Objektkonstruktion). Diese Analyse ist konservativ (Meyer sagt *pessimistic*) in dem Sinn, dass alle Anweisungen in allen Routinen betrachtet werden, ob diese nun ausgeführt werden, oder nicht. Aber sie ist sehr fein, in dem Sinn, dass zur Bestimmung dieser Typmengen wirklich ein ganzes Sys-

tem untersucht wird: also zu jeder Routine werden alle anderen Routinen mit einbezogen, die sie aufrufen. Zuletzt wird einfach für jeden Element-Aufruf festgestellt, ob dieser Aufruf für alle aktuell möglichen Zieltypen verfügbar ist. Das war's schon.

In einem solcherart geprüften System kann man CATs aufrufen, wie man will und man kann auch semi-konforme Objekte überall im System verbreiten. Nur wenn man einen CAT aufruft und die Zielvariable kann wirklich genau so ein Objekt enthalten, indem gerufene Element auch CAT ist, nur dann wird der Aufruf für ungültig erklärt.

Mit dem Ziel möglichst nicht zu grob zu sein, hat die Feine Analyse einige Nachteile in Kauf genommen, die Meyer (1995) aufzählt:

- Durch die Behandlung von maximalen Mengen ist sie ziemlich kompliziert.
- Durch die Behandlung des gesamten Systems auf einmal ist die Überprüfung langsam und flüchtig und bietet keine Unterstützung für vorkompilierte Bibliotheken.

Die vollständigen Regeln für die **grobe Analyse** sind auch ziemlich detailreich, aber die Version, die Meyer auf der OOPSLA Vortrag gibt die Grundidee genial knapp wieder:

Definition: polymorphe Entität

Eine Entität x ist polymorph, wenn sie eine der folgenden Eigenschaften erfüllt:

- Sie erscheint in einer Zuweisung $x := y$ wobei y einen anderen Typ hat, oder selbst (rekursiv) polymorph ist.
- Sie ist ein formales Routinenargument.
- Sie ist eine externe Funktion.

Wir sehen schon worauf es hinaus läuft: das Markieren aller Entitäten (Variablen) die Objekte von mehr als nur einem Typ enthalten können. Damit wird die maximale Menge aus dem feinen Algorithmus auf ein einzelnes Bit zusammengeschrumpft.

Definition: CAT, catcall

Eine Routine ist ein CAT (*Changing Availability or Type*), wenn eine Redefinition in einer der Unterklassen ihren Exportstatus oder den Typ eines ihrer Argumente ändert.

Ein Aufruf ist ein *catcall* falls ein solcher CAT aufgerufen wird.

Damit ist auch endlich der Begriff CAT definiert, den ich oben schon verwendet habe. Und die neue Typregel können wir ganz einfach formulieren:

Die neue Typregel

Catcalls auf polymorphen Entitäten sind verboten.

Einen bestehenden Nachteil dieses zweiten Verfahrens kann man sich hieraus direkt ableiten: jeder Aufruf ist genau dann gültig wenn er nicht polymorph ist, oder kein CAT. Der erweiterte Prüfer findet heraus, welches von beidem er nicht ist, aber aus dem Code ist es nicht direkt ersichtlich: es fehlt also Dokumentation.

Man kann übrigens auch andere Methoden der Typprüfung in das Schema von grob und fein einordnen. Zum Beispiel wäre es in einem lokalen Sinn die größte Methode, covariante Redefinitionen zu verbieten, dann gäbe es keine CATs mehr und folglich auch keine Catcalls. Generell sind Systeme mit mehr Deklarationen größer, weil sie dem Typprüfer weniger Wahl lassen. Die feinste vorstellbare Methode wäre demzufolge, gar keine Deklarationen zu verwenden. Man kann sich hier einen statischen Typprüfer für SMALLTALK vorstellen, der ähnlich funktioniert wie die obige feine Analyse, aber an Stelle von maximalen Mengen von Typen einer Variable findet er die maximale Menge von Nachrichten, die von allen den möglichen Objekttypen hinter einer Variable verstanden wird. (Man beachte: „Nachricht schicken“ ist SMALLTALK für „Element aufrufen“.) Dieser Prüfer könnte die Konsistenzbedingung a) sicherstellen, ohne dabei b) zu benötigen! (Ich kann im Moment allerdings nicht sagen, ob so ein Prüfer wirklich genau die derart konsistenten Programme erkennt, oder ob er an einigen Stellen so pessimistisch sein muss, der er in Wirklichkeit gar nicht mehr so fein ist.)

EIFFEL als Programmiersprache — und um das zu sehen, muss man es nicht mit dem völlig typlosen SMALLTALK vergleichen — gehörte schon immer zu der Seite: mehr Deklarationen, einfachere Regeln. Dazu passt auch der 1995er Umstieg von der feinen auf die grobe Analyse. Und im

selben Sinn beschäftigt sich der Rest dieser Arbeit damit, eine bessere Dokumentation durch Typen zu erreichen — aber zuvor müssen wir noch durch eine eher düstere Phase...

4.2 Monomorphe Typen

Ich weiß auch nicht, woran es liegt, dass ich die meisten Unterabschnitte hier immer mit den gleichen Worten beginne, aber schon wieder ist die Idee ganz einfach: man führt einfach verschiedene Typen ein, für Variablen, die auch Objekte konformer Typen enthalten können (also polymorph sein), und Variablen, die immer nur Objekte genau des deklarierten Typs enthalten: das sind dann die monomorphen Variablen. Natürlich können die polymorphen Variablen auch Objekte genau des deklarierten Typs enthalten, deswegen stellen die polymorphen Typen eine echte Obermenge der monomorphen Typen dar. Nach der selben Logik wie in Meyers grober Analyse, verbietet man dann einfach *catcalls* auf polymorphen Typen, d.h. man erlaubt sie nur für explizit als monomorph deklarierten Typen.

Die Regeln für die konforme Zuweisung sehen dann folgendermaßen aus:

1. An eine Entität polymorphen Typs kann man zuweisen, wie bisher: Objekte von Entitäten aller konformen Typen.
2. An eine Entität monomorphen Typs kann man nur von Entitäten des *selben* und ebenfalls monomorphen Typs zuweisen.

Die Restriktion an monomorphe Entitäten nur von anderen monomorphen Entitäten zuzuweisen ist sehr streng, aber nötig, weil man ja sonst indirekt ein Objekt eines konformen aber nicht identischen Typs in die Variable mogeln kann.

Diese Regeln scheinen also meiner Forderung aus dem letzten Unterabschnitt nachzukommen, den Algorithmus durch zusätzliche Deklarationen explizit zu machen. Diese Art der Deklaration ist aber eher lästig als nützlich: man macht nur die Arbeit, die Meyers grobe Analyse bereits automatisch macht und zudem sind die Deklarationen auch relativ flüchtig: es ist ja gerade der Vorteil der Subtypen, dass man jede Variable im Nachhinein zu einer polymorphen Variable machen kann. Dies wird von den monomorphen Typen direkt unterwandert! Wie man besser dokumentieren kann, sehen wir erst nach einer genaueren Analyse des Problems. Man muss auch bedenken, dass diese Lösung noch mit allen Ar-

ten von *catcalls* umgehen kann, während zukünftige Lösungen viel spezifischer sein werden. Dann enthalten Typdeklarationen (hoffentlich) nützlichere Informationen.

In MATCH-O, einer EIFFEL-Variante von Colnet und Liquori (2000) werden die monomorphen Typen als *match types* bezeichnet. Dies liegt daran, dass die Autoren die monomorphe Semantik auf einem Umweg erreicht haben, ausgehend von der Sprache LOOM, welche überhaupt keine Subtypen, aber dafür Match Typen unterstützt. Match Typen werden wir noch im Abschnitt 7 kennen lernen, hier genügt es erst einmal zu wissen, dass es sich dabei um eine Art der beschränkten Generizität handelt. Das heißt, in LOOM sind alle Typen monomorph, aber es gibt Match-beschränkte Typvariablen, die für alle Typen stehen, die einen anderen „matchen“. Weil man die Typvariablen sehr oft braucht und das zu viel Schreiberei führt, hat LOOM ein abkürzendes Konstrukt, sogenannte *hash types* (weil man sie mit dem # Zeichen schreibt), das eine implizite Typvariable darstellt. Diese *hash types* haben eine Semantik ähnlich den normalen polymorphen Typen der Objekt-Orientierung und daraus haben Colnet und Liquori ihr Typsystem abgeleitet. Letztendlich ist ihr System MATCH-O aber trotz des Namens näher an monomorphen Typen, als an Match Typen.

4.3 Neubesetzung mit recast

Wieder ist die Idee einfach, nur diesmal ist sie nicht genial: jede Routine, die ein potenzieller Catcall sein könnte muss eine *recast* Klausel enthalten. Diese verweist auf eine so genannte *recast function*, welche die Argumente von ihrem „zu großen“ Typ der äußeren Schnittstelle auf den kleineren Typ der ausgeführten „presst“. Wie die *recast function* dies machen soll, wird nicht gesagt. Man kann sich aber leicht vorstellen, dass die betroffene Routine diese Umwandlung selbst durchführen kann, sie ist ja in derselben Situation. (Ein Beispiel dafür findet sich in Abschnitt 9 meiner anderen Arbeit.)

Recast hilft den Routinen also nicht wirklich ihren neuen Verpflichtungen nachzukommen. Und diese Verpflichtungen sind nicht unwesentlich: covariante Redefinitionen werden dabei so aufwendig gemacht, dass man besser auf sie verzichtet und bei Nichtvarianz bleibt. Und was noch schlimmer ist: weil *recast* auch die alte Regel der generischen Konformität verteidigen soll, muss auch jede Routine mit *recast* versehen werden, die generische Argumente hat. Das ist ein Aufwand! Und dabei muss man bedenken, dass Programmierer bisher so trainiert waren, Catcalls zu vermeiden (siehe voriger Teilabschnitt). Das heißt, all dieser neue *recast*-Code wird eigentlich nie aufgerufen. Wahrschein-

lich wird man den Code deshalb auch nicht besonders gut schreiben. Und wenn sich die Programmierer dann an `recast` gewöhnen, wird er vielleicht doch mal aufgerufen...

Alles in allem entsteht durch `recast` nur mit viel Aufwand eine riesige Grauzone, selbst gegenüber einer Welt ohne obige erweiterte Prüfung, wo also alle Catcalls in Laufzeitfehlern enden, ist `recast` noch ein großer Rückschritt. Insbesondere zeigt es genau in die dem vertraglichen Programmieren entgegen gesetzte Richtung, die mit der Abschaffung der Exporteinschränkung im selben Artikel eingeschlagen wurde. Darüber darf man sich wundern. Für die nun folgenden, hoffentlich besseren Vorschläge, kann man `recast` aber erstmal wieder vergessen.

5 Typing by Contract

Wenn man sich erst einmal das folgende überlegt hat, kann man Covarianz nie wieder im selben Licht sehen: Nach den Regeln des vertraglichen Programmierens in der Vererbung muss jede Redefinition einer Routine den Vertrag ihres Originals einhalten. Und das heißt (man sollte es wissen) Vorbedingung höchstens abschwächen, und Nachbedingung höchstens verstärken. Aber wenn nun eine Routine plötzlich deklariert sie möchte sich nur noch mit Argumenten eines Subtyps des eigentlichen Argument-Typs abfinden, so ist das eine Verstärkung der Vorbedingung. Sie verlangt plötzlich mehr, als ihr zusteht! Covarianz im allgemeinen Sinn steht also auf Kriegsfuß mit dem vertraglichen Programmieren.

Ausgehend von dieser Beobachtung aus meinem vorhergehenden Artikel, die ich im nächsten Unterabschnitt vertiefe, untersuche ich im folgenden Abschnitt die Anwendungsfälle der Covarianz. Daraus ergeben sich zunächst methodische Ratschläge an den Programmierer, und dann auch die Anforderungen an ein Typsystem. Erst die Methode, dann die Werkzeuge — so sollte die Ingenieurwissenschaft immer vorgehen.

5.1 Abschwächung ankündigen

Nachdem ich also festgestellt hatte, dass man die Typeinschränkungen als Verträge auffassen konnte, habe ich daraus gleich eine Lösung gemacht: Man macht einfach die Vorbedingung im Original einer Routine stark genug und schwächt sie dann in den Redefinitionen ab! Dieses Verfahren nutzt natürlich die volle Generalität des Design by Contract und ist daher eher wie mit Kanonen auf Spatzen zu schießen. Andererseits kann man da aber die Probleme präzise behandeln, ohne sich den

Beschränkungen eines Typsystems hinzugeben, und das ist besonders nützlich, da wir ja anscheinend ein ausreichendes Typsystem noch nicht haben. Außerdem machen die Verträge endlich mal explizit, was sich die ganze Zeit im Typsystem versteckt hat. Dies ist die Voraussetzung, um in den nächsten Unterabschnitten dann zu erkennen, dass Lösungen mit Covarianz im Grunde oft komplizierter sind, als solche ohne. Zum Verstehen eines Entwurfs, muss man eben die impliziten Teile mitrechnen, und wenn man's genau nimmt, sind diese sogar noch komplizierter als das Explizite, weil man sie ja nicht direkt vor sich hat.

Wenn man die Zusicherungen benutzt, um die fehlenden Typaussagen zu notieren sieht unser Standard-Beispiel so aus:

```
class EQ feature
  is_equal ( other : like Current ) : BOOLEAN is
    require other.same_type(Current)
  ...
end -- class EQ

class TRAVERSABLE[T] inherit EQ feature ...
end -- class TRAVERSABLE[T]

class LIST[T]
  inherit TRAVERSABLE[T] redefine is_equal end
feature
  is_equal ( other : LIST[T] ) : BOOLEAN is
    require else True
  ...
end -- class LIST[T]
```

Die Vorbedingung wird naturgemäß von den Unterklassen geerbt. In der Klasse `LIST` sehen wir die explizite Unterscheidung ab sofort alle Subtypen von `LIST` als Argumente zu akzeptieren. Die Zauberformel `require else True` heißt nichts anderes, als: die Vorbedingung wird fallen gelassen.

Im Artikel habe ich auch darüber spekuliert, dass man solche Vorbedingungen statisch überprüfen kann. Der Algorithmus wäre wohl ähnlich denen, die bereits in Eiffel verwendet werden. Ich habe allerdings keine genauen Regeln dafür angegeben, denn mein Vorschlag ist nicht abhängig davon, ob so ein Algorithmus existiert. Falls mal eine solche Vorbedingung verletzt wird, so ist das Resultat ein Laufzeitfehler. Ich habe argumentiert, dass solche Laufzeitfehler im vertraglichen Programmieren sowieso unvermeidbar sind, und es ist besser etwas mehr Laufzeitfehler zu haben, als Fehler die noch länger unentdeckt bleiben. (Wenn man mehr Fehler statisch finden kann, ist das natürlich noch besser.)

Weiterhin habe ich in dem Artikel den Entwurf von solchen „Redefinitionshierarchien“ explizit gemacht. Gerade bei der Klasse `EQ` stellt sich die Frage, ob es nicht insgesamt praktischer für die Nutzer der Klassen wäre, wenn `is_equal` einfach immer alle Argumente akzeptiert und bei nicht vergleichbaren Argumenten einfach `False` liefert.

Dieses Vorgehen hat den direkten Vorteil, dass den Erben von (in Eiffel sind das alle Klassen!) die schwierige Entscheidung abgenommen wird, ab welcher Hierarchiestufe die Vorbedingung fallengelassen wird. Als Nachteil ist anzuführen, dass ein Vergleich von einer Liste mit einem U-Boot oder ähnliches wahrscheinlich ein Programmierfehler ist, der uns durch eine solche Definition durch die Lappen geht. Aber `is_equal` ist wirklich ein undankbares Beispiel: die Routinen muss es allen möglichen Eiffel-Programmierern recht machen und das überfordert eigentlich jeden.

Diese Entwurfsfragen sind essentiell und unabhängig davon, wie wir die beschlossenen Verträge ausdrücken. Wir müssen solche Überlegungen also immer anstellen, selbst wenn wir überhaupt kein Typsystem haben!

Wie wir schon an obigem Beispiel sehen, ist die vertragliche Variante der Typdokumentation redundant mit dem Typsystem: die Verwendung von `like Current` geht einher mit der Deklaration einer Vorbedingung, und die Ersetzung von `like Current` in der Unterklasse geht einher mit der Aufhebung der Vorbedingung. Könnten wir nicht vielleicht die Semantik von `like Current` so justieren, dass sie genau das aussagt? (Antwort im Abschnitt 7.4.)

6 Anwendungsfälle der Covarianz

Wie die Überschrift schon sagt, sammelt dieser Abschnitt ein paar Anwendungsfälle der Covarianz auf (die anderen beiden Typ-Probleme betrachte ich ja als geklärt) und bespricht sie aus methodischen Gesichtspunkten. Wir beginnen mit den Kühen und Skifahrern aus dem Reich der akademischen Beispiele.

6.1 Perfekte Beispiele für schlechten Entwurf

Ein typischer Fehler von Anfängern in der OO Programmierung ist die so genannte Über-Modellierung: die Anfänger sind so entzückt von der Ausdrucksstärke von Vererbung und Assoziationen, dass sie mit Freude große Diagramme malen, mit vielen Klassen drauf und vielen Beziehungen zwischen diesen Klassen. Die Objektorientierung verlockt dazu mög-

lichst viele Aussagen über die Anwendungswelt mit Hilfe von Objekten auszudrücken.

Natürlich ist dieses Vorgehen unsinnig: sobald man dann den Code sieht sind die überflüssigen Klassen im Weg, oft erfordert der Code auch eine andere Struktur, als man schon hat. In einer fertigen Software dienen die Klassen dazu, den Code zu strukturieren und einzelne Teile gegeneinander zu kapseln. Daraufhin muss man auch beim Entwurf schon arbeiten. Ebenso ist die Aufgabe des Typsystems zu verstehen:

Das Programmierer – Modellierer Prinzip

Typsysteme dienen nicht in erster Linie der Beschreibung und Strukturierung der Anwendungswelt, sondern der Beschreibung und Strukturierung der Softwarelösung.

In einem der grundlegenden Werke der Objektorientierung (Meyer 1988) können wir nachlesen: Klassen sind Module, die uns über eine Schnittstelle bestimmte Funktionalitäten bereitstellen. Ein Modul kapselt Daten und der erstwichtigste Vorteil der Objektorientierung besteht darin, dass man die Module beliebig oft instanzieren kann: jedes Objekt erhält seine eigene Kopie der Daten.

Umso erstaunlicher ist es, dass derselbe Autor uns in anderen Werken (1995, 2003) dem ganz entgegen gesetzte Beispiele präsentiert: da ist von jungen Skifahrern und –innen die Rede, und von Booten und ihrem Skipper. Schauen wir uns mal das erste Beispiel an:

```
class SKIER feature
  roommate : like Current
end -- class SKIER

class BOY inherit SKIER end
class GIRL inherit SKIER end
```

Hier drückt Meyer die Anforderung aus, dass immer nur Jungs mit Jungs und Mädchen mit Mädchen den Raum teilen sollen. Aber ganz offensichtlich sind die Klassen `BOY` und `GIRL` hier überflüssig, besser würde man schreiben:

```

class SKIER feature
  gender : GENDER
  roommate : SKIER
invariant
  gender = roommate.gender
end -- class SKIER

```

Wer's noch einfacher mag, kann statt dem Typ `GENDER` auch `BOOLEAN` nehmen, und das Attribut `gender` entsprechend `is_boy` nennen.

Wie wir sehen, ist der Entwurf jetzt viel *direkter* geworden: wenn wir später programmieren (zum Beispiel eine Routine zum Verteilen der Kinder auf ihre Zimmer), können wir ganz einfach das Attribut `gender` verwenden und müssen uns nicht mit dynamischer Typprüfung herumplagen, wie Meyer (1995) das tut. Und das Beste an dem direkten Ansatz ist: wir können ihn immer noch um zusätzliche Klassen `BOY` und `GIRL` erweitern, falls sich das als nötig erweisen sollte. Der Entwurf bleibt dabei natürlich frei von Covarianz. Mehr dazu findet sich in Abschnitt 13 meines ersten Artikels zum Thema (Will 2003), im dortigen Abschnitt 9 wird auch das andere Beispiel besprochen.

Genau wie bei diesen beiden Beispielen stellt sich auch bei anderen heraus, dass Covarianz bei der Konstruktion von Lösungen keine Rolle spielt. Covarianz mag in der Anwendungswelt „natürlich“ sein, aber in der Softwarewelt ist sie nicht natürlich. Genauso wenig, wie es Exporteinschränkungen sind. Als Softwareentwickler braucht man keine besonderen Modellierungsmittel, um mit `STRAUSS`envögeln zurechtzukommen, die nicht fliegen können. Entweder man behandelt nur fliegende Vögel, oder man führt eine explizite Vorbedingung für die Flug-Routine ein: in jedem Fall muss man die Entscheidung an einem Punkt des Entwicklungsprozesses treffen und dieser Punkt ist nicht in der letzten Minute — dann wäre Softwareentwicklung nämlich ganz trivial.

Ich behaupte hier also, dass das wichtigste Argument für die Covarianz, nämlich ihr Bezug zur Anwendungswelt, nichtig ist.

6.2 Peer Routines

Mit dem Beispiel, dass ich in den vorigen Abschnitten verwendet habe, nahm ich schon den wichtigsten Anwendungsfall der Covarianz vorweg. *Peer Routines* sind Routinen einer Klasse, die Argumente oder Ergebnisse vom Typ der Klasse haben — also der klassische Anwendungsfall von `like Current`. Die Argumente der Routine haben dabei den gleichen Status wie der Typ der Klasse selbst, daher der Begriff

Peer Routine (*peer* = Gleicher unter Gleichen). Wenn man Subtypen verwendet, wird diese Gleichheit zerstört: der Apfel muss sich plötzlich mit allen Früchten vergleichen. Man hat also die Idee den rekursiven Charakter des ursprünglichen Typen zu erhalten und so kommt man auf die Matching-Beziehung.

Im Gegensatz zu Covarianz spiegeln diese Ideen wirklich die Essenz der Typen wieder, leider wurden sie noch in keiner populären OO Sprache implementiert.

Hier ist zu bemerken, dass `GENERICJAVA` bereits rekursive generische Beschränkungen unterstützt, wie wir sie in Unterabschnitt 7.1 sehen werden. Aus dieser Richtung erleben wir demnächst vielleicht eine positive Überraschung, obwohl ich irgendwie bezweifle, dass `JAVA`-Programmierer diesen Mechanismus wirklich gut ausnutzen werden, es ist ja doch nicht ganz trivial.

Im Abschnitt 7 werden wir ein paar der Mechanismen sehen, die dem modernen *language designer*, d.h. dem Entwickler von Programmiersprachen, zur Behandlung von *peer routines* zur Seite stehen. Im Anhang gebe ich eine Kurzanleitung, wie man sie in `EIFFEL` behandeln kann. Eine generelle Möglichkeit zur Covarianz jederzeit, wie sie von `EIFFEL` geboten wird, ist dabei aber ganz sicher nicht nötig.

In der Literatur spricht man übrigens von *binary methods*, weil die binären Operatoren kanonische Beispiele von *peer routines* sind. Der Begriff ist aber nicht so gut, weil die Binarität keine essentielle Eigenschaft des Phänomens ist. Meine Referenz ist Bruce et. al. (1995).

6.3 Denk mal einer an die Erben

Im *recast proposal* (2003) bemerken Meyer et. al. so ganz nebenbei, dass man sich ja bei den *catcalls* nur um die qualifizierten Aufrufe kümmern muss (d.h. die von einer Klasse zur anderen mit `Syntax.target.routine`) und nicht um die unqualifizierten (internen, `Syntax` ohne qualifizierendes Aufrufziel), da ja in letzterem Fall der Typ des Ziels immer bekannt ist. Damit kehren die Autoren ein Problem unter den Teppich, das da unten schon ein paar gute Bekannte hat.

Prinzipiell haben sie mit dieser Beobachtung natürlich Recht: innerhalb einer Klasse sind alle Typen und Signaturen exakt statisch bekannt, egal auf welchen komplizierten Vererbungswegen sie ihren Weg in diese Klasse genommen haben. Es ist gerade ein Vorteil der Sprache, dass der sehr mächtige Vererbungsmechanismus völlig statisch ist: alles wird zur Compile-Zeit ausgewürfelt, Inkonsistenzen werden sofort festgestellt. Allerdings wird dieser Mechanismus auch über Gebühr belastet: man

legt kaum Wert auf die interne Struktur einer Klasse, das heißt falls mal ein Fehler auftritt, sieht man die Inkonsistenz zweier Teil und kann nicht sagen, welcher Teil „Schuld“ hat.

Einen Teil der *catcall*-Probleme unter den „Klassen-intern“ Teppich zu schieben ist wieder ein Fall von besonders „feiner“ Prüfung, es fehlt Dokumentation. In diesem Teilabschnitt möchte ich mich mal den Klassen-internen Aspekten der Covarianz widmen. Insbesondere der dritten Art von verankerten Typen, den `like query`s, die sehr beliebt sind, aber etwas unterspezifiziert. Auch sie kann man mit Generizität besser verstehen.

Die covariante Redefinition von Attributen ist exakt so ein Beispiel von unfeststellbarer Schuld bei Inkonsistenzen. Das Problem ist einfach: Routinen der Oberklasse könnten Objekte an ein Attribut zuweisen, die nicht mehr konform zu dem neuen, covariant redefinierten Typ des Attributs sind. Die Lösung ist auch einfach: die sich beide Konflikt-Parteien in derselben Klasse befinden, kann eine triviale statische Prüfung diesen Fall aufdecken. Die Frage ist nur: wer hat Schuld? Außerdem ist diese Prüfung wieder relativ flüchtig: das Vorhandensein von Zuweisungen innerhalb von Routinen einer Oberklasse entscheidet darüber, ob man ein Attribut covariant redefinieren kann. Da kann ich eine einfachere Regel vorschlagen: Attribute sollten immer nichtvariant sein. Wenn man sie unbedingt redefinieren will, kann man sie in den Oberklassen immer noch als verzögerte Anfragen deklarieren, die man dann später zu Attributen redefiniert.

Ich schrieb schon bei der Einführung in die Covarianz im Abschnitt 0, dass die Verlockung der Covarianz in EIFFEL darin besteht, eine besonders einfache Regel haben zu wollen, die sagt „alles kann covariant redefiniert werden“. Dies geht aber eben so nicht auf, weil jedes einzelne Ding dann doch wieder seine eigenen Regeln braucht, um zu funktionieren. Das Redefinieren von Attributen zum Beispiel ist den Extra-Aufwand nicht wert, den es an zusätzlichen Regeln erfordert, also sollte es sich auch nicht hinter der allgemeinen Covarianz-Regel verstecken.

Die `like query` Typen sind ein weiteres Beispiel für solcherart versteckte Komplexität. Schon William Cook hat 1989 festgestellt, dass deren Typeregeln nicht wirklich funktionieren. Nehmen wir mal die Deklaration `x : T` an, dann sollte der Typ `like x` zu `T` konform sein, aber nicht umgekehrt, denn `like x` kann ja nach späteren Redefinitionen von `x` für jeden beliebigen Subtypen von `T` stehen. Wenn man diese Regel aber ernst nimmt, kann man nicht einmal den Wert von `x` an eine Variable vom Typ `like x` zuweisen: `x'` Typ ist ja `T` und der ist, wie gesagt zu

`like x` nicht konform. Die folgenden beiden Variablen verhalten sich auch auf unangenehme Art unsymmetrisch:

```
x : T
y : like x
```

Wenn man darüber etwas länger nachdenkt, stellt sich als Konflikt heraus, dass `like x` gleichzeitig für einen variablen Typ steht und für den Typ von `x`. Die Frage ist, an welcher Stelle vom Code man „weiß“, dass `like x` dem Typ von `x` entspricht (man muss das wissen, um den Typ verwenden zu können) und an welcher Stelle man es nicht weiß (man darf das nicht wissen, um von der Redefinition von `x` unabhängig zu sein).

Im jetzigen EIFFEL sind diese beiden Zonen nicht voneinander getrennt, man kann eine Typprüfung also immer nur im Ganzen durchführen, für jede Klasse, die `x` erbt, aufs Neue. Diese Prüfung ist ähnlich der Prüfung von Makros oder C++ Templates: diese haben keine Typschnittstellen, man muss sie also bei jeder Instantiierung neu überprüfen! Dabei kann mit geschickt eingesetzter Generizität genau das erreichen, was der `like query` Typ nicht schafft. Man lagert einfach all den Code, der den Typ von `like x` nicht kennen muss, in eine generische Klasse aus. Diesen Code kann man dann mehrfach instanzieren und der umgebende Code kennt dann natürlich den zugeordneten Typ. So hat man den variablen und den invarianten Teil der Klasse sauber voneinander getrennt. Die Mechanismen Attributdefinition und verankerter Typ sind dagegen nur ad hoc Lösungen, die nicht weit tragen.

Als Beispiel für dieses Vorgehen können wir uns die einzige Stelle im Code von SmartEiffel anschauen, die Colnet und Liquori (2000) nicht auf ihre Variante von monomorphen Typen umstellen konnten. Dort stehen die Klassen `DECLARATION_LIST`, `FORMAL_ARG_LIST` und `LOCAL_VAR_LIST` in genau der beschriebenen Beziehung: `DECLARATION_LIST` ist ein gemeinsamer Vorfahre der beiden anderen Klassen und diese nutzen bei der Vererbung die covariante Redefinition. Vereinfacht sieht das Ganze so aus:

```
class DECLARATION_LIST feature
  name ( i : INTEGER ) : LOCAL_ARGUMENT is deferred end

  flat_list : ARRAY[like name]
  ...
end -- class DECLARATION_LIST
```

```

class FORMAL_ARG_LIST inherit DECLARATION_LIST feature
  name ( i : INTEGER ) : ARGUMENT_NAME is
    do Result := flat_list.item(i) end
  ...
end -- class FORMAL_ARG_LIST

class LOCAL_VAR_LIST inherit DECLARATION_LIST feature
  name ( i : INTEGER ) : LOCAL_NAME is
    do Result := flat_list.item(i) end
  ...
end -- class LOCAL_VAR_LIST

```

Hier stellen wir zunächst erst einmal fest, dass der Code von `name` in den beiden Unterklassen dupliziert wurde. Hier lohnte es sich nicht, den Code auszulagern, denn die Redefinition des Typs der Routine macht eine Wiederholung des Codes in jedem Fall erforderlich. Dies ist schon ein Anzeichen in Richtung der besseren Lösung: wiederholter Code bei dem nur Typen verändert sind, weist auf Generizität. Wir schreiben also:

```

class DECLARATION_LIST[T -> LOCAL_ARGUMENT] feature
  name ( i : INTEGER ) : T is
    do Result := flat_list.item(i) end

  flat_list : ARRAY[T]
  ...
end -- class DECLARATION_LIST

class FORMAL_ARG_LIST
  inherit DECLARATION_LIST[ARGUMENT_NAME]
  feature ...
end -- class FORMAL_ARG_LIST

class LOCAL_VAR_LIST
  inherit DECLARATION_LIST[LOCAL_NAME]
  feature ...
end -- class LOCAL_VAR_LIST

```

Dabei müssen wir nur alle Vorkommen von `like name` in `DECLARATION_LIST` ersetzen durch `T`, und in den anderen Klassen brauchen wir sonst überhaupt nichts mehr ändern! Leider sind Colnet und Liquori selbst nicht auf diese Idee gekommen: sie haben stattdessen die 300 Zeilen Code der Klasse `DECLARATION_LIST` komplett in den beiden anderen Klassen dupliziert. Das ist auf jeden Fall keine Werbung für Ihren Ansatz der Typisierung!

Mal so ganz hypothetisch: würde man die Covarianz einfach abschaffen, verschwänden all die in diesem Teilabschnitt genannten Probleme und die Programmierer müssten sich gezwungenermaßen bessere Lö-

sungen für die genannten Angelegenheiten suchen. Mann, habe ich perverse Vorstellungen...

7 Vertragsverhandlungen: bessere Typsysteme

Soweit ich das erfasst habe, gibt es über Match-Typen gar nicht so viel Material. Die erste Erwähnung scheint bei Kim Bruce zu sein, der sie als syntaktische Abkürzung für F-gebundenen Polymorphismus eingeführt hat. Bruce hat mehrere objektorientierte Programmiersprachen darauf aufgebaut: zunächst TOOPLE (1994), dann PolyTOIL (1995), und zuletzt LOOM (1997), bei welcher Match-Typen Subtypen völlig ersetzen. Wir werden dann sehen, wie Match-Typen Subtypen simulieren können, und das Resultat sieht aus, wie die monomorphen Typen, die wir in Abschnitt 4.2 schon gesehen haben.

7.1 Rekursive Generizität

Wieder eine ganz einfache Idee: alle Klassen erben von `EQ` und wir wollen sicherstellen, dass Listen nur mit Listen verglichen werden und Äpfel nicht mit Birnen, gleichzeitig sollen aber Array-Listen und verkettete Listen vergleichbar sein (die Subtypen von Liste). Also müssen wir einfach nur genauer sagen, mit was ein `EQ` vergleichbar sein soll! Versuchen wir es mal mit Generizität:

```

class EQ[C] feature
  is_equal ( other : C ) : BOOLEAN is ...
end -- class EQ[C]

```

Dann behält man den Typparameter so lange in den Klassen der Hierarchie, wie man möchte, und wenn man dann komplette Subtypen aus den Unterklassen generieren will, lässt man den generischen Parameter einfach weg. Ich gehe im Beispiel jetzt mal von den Klassen `TRAVERSABLE` und `LIST` auf `FRUCHT` und `APFEL` über, dann stört uns der Typparameter nicht, aber die Aussage bleibt erhalten:

```

class FRUCHT[C] inherit EQ[C] feature ...
end -- class FRUCHT[C]

class APFEL
  inherit FRUCHT[APFEL] feature ...
end -- class APFEL

```

In der Klasse `APFEL` wird die Signatur von `is_equal` hier automatisch zu `(other : APFEL) : BOOLEAN`. Damit drücken wir aus, dass

die Subtypen von `APFEL` alle miteinander vergleichbar sein sollen (wir wollen dann natürlich keine Kovarianz mehr zulassen!) und gleichzeitig haben wir die problematische Konformität von `APFEL` und `FRUCHT` beseitigt, weil `FRUCHT` jetzt kein Typ mehr ist, sondern eine ganze Klasse von Typen. (Zur Erinnerung: weil `APFEL` konform zu `FRUCHT` war, konnten wir ihn ja an eine `FRUCHT`-Variable zuweisen und dann mit `BIRNE` vergleichen, obwohl sein `is_equal` nur `APFEL` akzeptiert.)

Nachdem wir die Konformität beseitigt haben, steht natürlich in Frage, ob überhaupt noch eine sinnvolle Relation zwischen `APFEL` und `FRUCHT` (beziehungsweise `EQ`) besteht. Zum Beispiel sollte es noch möglich sein Routinen zu benutzen, die mit allen Unterklassen von `EQ` funktionieren.

```
class SEARCHABLE[T -> EQ[T]] feature
  index_of ( x : T ) : INTEGER is ...
end -- class SEARCHABLE[T -> EQ[T]]
```

Diese Klasse benutzt beschränkte Generizität, um den Typparameter einzuschränken auf Typen, die mit sich selbst vergleichbar sind. Damit kann man durchsuchbare Strukturen von `ÄPFELn` und `BIRNEn` definieren, aber nicht allgemein von `FRÜCHTEN` (bzw. man kann keinen Schaden mit letzteren anrichten, denn `ÄPFEL` und `BIRNE` sind ja nicht konform zu `FRUCHT`).

Die rekursiven Typ-Beschränkungen haben mal einen an F-Algebren erinnert (was immer das ist), da hat er sie F-gebundenen Polymorphismus genannt. In EIFFEL-Terminologie wäre das F-beschränkte Generizität, aber das erklärt uns auch nichts, denn leider schien sich die Forschung auf die „Interpretation“ solcher Typen zu konzentrieren (also Fixpunkte usw.) und hat sich wenig mit der Pragmatik beschäftigt. Meine Hauptreferenz (Canning et. al., 1989) verwendet auch wieder Cardelli-Records und haufenweise strukturelle Subtypen.

Wenn man in der nunmehr generischen Klasse selbst den Typparameter verwenden möchte, muss man noch eine kleine Änderung hinzufügen:

```
class EQ[C -> EQ[C]] feature
  is_equal ( other : C ) : BOOLEAN is
    deferred
    ensure other.is_equal(Current)
  end
end -- class EQ[C]
```

Schauen wir uns den Code in der Nachbedingung an. Damit `other` dort auch ein Element `is_equal` hat, müssen wir schon wissen, dass `C` ein Subtyp von `EQ` ist! Aber nichts einfacher als das, wir geben diese Restriktion einfach bei Definition `EQ` von schon mit an (unterstrichener Teil)! Dadurch erhalten wir auch schon für die benutzte Klasse selbst eine rekursive Beschränkung.

Hier sollte auch klar werden, warum ich im ersten Abschnitt so viel Wert auf HASKELL gelegt habe: die hier gezeigte beschränkte Generizität entspricht nämlich ziemlich genau dem HASKELL'schen beschränkten Polymorphismus. Der zusätzliche generische Parameter in EIFFEL entspricht genau der Typvariable in der HASKELL'schen Definition einer Typklasse. Und weil man die HASKELL'schen Typklassen ja *ausschließlich* zusammen mit dem beschränkten Polymorphismus einsetzt, gibt dieser einfache Zusammenhang schon Hoffnung, dass dieser simple Mechanismus schon ausreicht, um die meisten Anwendungen von Kovarianz genauer zu typisieren!

7.2 Matching

Die oben genannten Sprachen von Kim Bruce sind alle sehr Theorie-lastig, Klassen sind Werte und dadurch muss zwischen Klassen-Typen und Objekt-Typen unterschieden werden. Ich werde hier einfach Matching auf die Sprache Eiffel beziehen...

Stellen wir uns einfach mal vor, wir wollen uns die zusätzlichen generischen Parameter der obigen Lösung sparen. Wir könnten dazu ein spezielles Symbol einführen, das innerhalb einer Klasse immer für den aktuellen Typ steht. Dann leiten wir aus den eben angegebenen Regeln für rekursive Beschränkungen, Regeln für unser Symbol her und schon können wir die Vorteile der obigen Methode nutzen, ohne uns um die rekursiven Hintergründe zu kümmern.

Die meisten Werke über Match Typen, nennen das Symbol `MyType` oder so ähnlich. Wir können aber auch ganz mutig sein, und das Symbol `like Current` nennen. Auf diese Weise geben wir diesem einen von drei verankerten Typen eine neue Semantik, so wie ich das im Abschnitt

5 schon angedeutet habe. So entwickelt man die Sprache weiter, ohne zu große Veränderungen durchzuführen.

Welche Regeln erhalten wir also: `like Current` steht für einen impliziten generischen Parameter, der zum aktuellen Typ konform ist (weil auf selbigen generisch beschränkt). `like Current` steht also überall in der Klasse für den selben Typ, aber nicht für den selben Typ wie der aktuelle. Das folgende Beispiel illustriert den Sachverhalt:

```
class FRUCHT feature
  beispiel is
    local a : like Current; b : FRUCHT
    do
      b := a -- legal
      a := b -- illegal
    end
end -- class APFEL
```

Man kann also gezielt `like Current` und den aktuellen Typ verwenden, je nachdem welche Semantik gewünscht ist. Damit wäre bereits erreicht, was ich im Abschnitt 5.1 gewünscht habe.

Leider können wir einen Typen wie `FRUCHT`, der implizite generische Parameter hat, nicht einfach so in Code verwenden, weil ja sein Parameter nicht instanziiert wurde. Nur wenn man die Klasse für generische Beschränkungen benutzt, kann man ihr eine offensichtliche Semantik geben: `class SOMETHING[T -> FRUCHT]` wird zu `class SOMETHING[T -> FRUCHT[T]]`.

Dies ist die sogenannte Match-beschränkte Generizität. In Bruce' Programmiersprachen wird sie geschrieben als match-gebundener Polymorphismus: " $T \triangleleft\# \text{FRUCHT} \bullet \text{SOMETHING}[T]$ ". In diesem Abschnitt haben wir die Semantik dieses Konstrukts mit Hilfe der rekursiv beschränkten Generizität erklärt, im nächsten Unterabschnitt findet sich eine Interpretation mit Subtypen höherer Ordnung und im übernächsten Unterabschnitt sind wir dann weise genug, um eine eigene Semantik für Match-Typen zu definieren.

7.3 Eine andere Interpretation

Cardelli und Abadi (1996) haben Match-Typen innerhalb ihrer Theorie untersucht, mit dem Ziel es auf andere Konzepte zurückzuführen und dadurch Typeregeln abzuleiten. Außer der offensichtlichen Interpretation durch F-gebundenen Polymorphismus haben sie noch eine weitere Interpretation untersucht. In jedem Fall betrachten sie Typen als Fixpunkte von Gleichungen der Form $A = A_{Op}(A)$, wobei A_{Op} ein Typoperator ist, al-

so eine Funktion, die aus einem Typen (z.B. X) einen anderen Typen $A_{Op}(X)$ macht. Wir können uns vorstellen, dass jede generische Klasse einen solchen Typoperator definiert, auch wenn dieser in Eiffel nicht explizit gemacht wird.

Wir erinnern uns daran, dass \triangleleft : das Symbol für Subtypen ist. Dann ist die F-gebundene Interpretation für Matching: $A \triangleleft\# B \gg A \triangleleft : B_{Op}(A)$. Die andere Interpretation ergibt sich, wenn man Subtypen höherer Ordnung definiert: $A_{Op} \triangleleft B_{Op} == "X \bullet A_{Op}(X) \triangleleft : B_{Op}(X)$. Dann definiert man Matching durch $A \triangleleft\# B \gg A_{Op} \triangleleft B_{Op} (= "X \bullet A_{Op}(X) \triangleleft : B_{Op}(X)$). Diese Interpretation ist intuitiv strenger, als die erste.

Die beiden Interpretationen liefern unterschiedliche Typeregeln für die Match-Typen, die aber beide ihren Zweck erfüllen. Außerdem gibt es noch andere Regeln für Match-Typen, die auch den Zweck erfüllen. Trotzdem sind die beiden Interpretationen echt verschieden. Der größte Unterschied, den ich verstanden habe, ist dass bei den Subtypen höherer Ordnung `like Current` nur mit sich selbst kompatibel ist, während es in der anderen Interpretation ein Subtyp des Klassentyps selbst (und also auch dessen Supertypen) ist. Dieser Sachverhalt führt allerdings dazu, dass die Matching-Relation in der F-gebundenen Interpretation nicht mehr transitiv ist. Zum Beispiel:

```
class A feature
  p ( other : like Current ) : INTEGER
  q : INTEGER
end
class B feature
  p ( other : like Current ) : INTEGER
end
class C feature
  p ( other : B ) : INTEGER
end
```

Hier haben wir $B \triangleleft\# C$ und $A \triangleleft\# B$, aber nicht $A \triangleleft\# C$ ☹. In einer klassischen OO Sprache ist dies aber kein Problem, denn da basieren ja Subtypen und auch Match-Typen immer auf Vererbung. Wir bräuchten also `class B inherit C ...` und dann würde `B` die Signatur von `C` erben.

Nach Cardelli und Abadi hat die Interpretation mit Subtypen höherer Ordnung verschiedene theoretische Vorteile, einer davon ist die Transitivität und Reflexivität der Match-Relation, ein anderer ist, dass sie nicht die „Entfaltungseigenschaft rekursiver Typen“ voraussetzt, um zu funktionieren, was das bedeutet, weiß ich allerdings nicht ☹.

Cardelli und Abadi schreiben am Schluss auch, dass ihre Interpretationen zum direkten Einsatz in einer Programmiersprache wohl zu kompliziert sind, und ermutigen die Verwendung einer primitiven Matching-Relation, also die direkte Angabe von Typregeln. Und genau das mache ich auch im nächsten Unterabschnitt.

7.4 Pragmatische Regeln

Nach dem Matching-Ansatz über rekursiv beschränkte Generizität aus dem Unterabschnitt 7.2, erhalten Klassen in deren Schnittstelle `like Current` vorkommt eine besondere Stellung in der Programmiersprache: sie können nur noch in generischen Beschränkungen verwendet werden. Dies ist natürlich eine ganz harte Restriktion. Man könnte jetzt auf die Idee kommen, auch diesen Sachverhalt besser zu dokumentieren, indem man solchen Klassen ein Attribut `matching` in die Schnittstelle schreibt, ähnlich wie das Attribut `deferred` für Klassen, die verzögerte Routinen enthalten. Allerdings ist hier die Lage unterschiedlich: verzögerte Routinen kann man ja in der Schnittstelle nicht erkennen, das Vorkommen von `like Current` allerdings schon. Und außerdem stellt sich heraus, dass es noch eine viel weniger restriktive Lösung gibt.

Wenn wir Klassen mit implizitem generischen Parameter auch zur Deklaration von Entitäten erlauben wollen, müssen wir uns fragen, welche Semantik sie haben sollen. Eine Möglichkeit wäre, einen impliziten Typ zu benutzen. Nehmen wir an, die Klasse `x` steht für das implizite `x[T]`. Dann definieren wir `class MONO_X inherit X[MONO_X] end` und lassen jegliche Vorkommen von `x` in Deklarationen implizit stehen für `MONO_X`. Die Typregeln entsprechen dann denen von monomorphen Typen: man kann Routinen mit `like Current` Argumenten aufrufen, aber man kann keine konformen Objekte zuweisen, ganz einfach weil die implizite Klasse `MONO_X` keine Erben hat, also sind nur Objekte der Klasse selbst konform zu ihr. Das wäre Möglichkeit Eins.

Eine andere Möglichkeit ist `x` in Deklarationen die Semantik `x[NONE]` zu geben. Dann kann man Routinen mit `like Current` Argumenten nicht aufrufen (außer mit `void` als Argument). Die Konsequenzen dieser Semantik auf die Konformität lägen dann allerdings in den Händen der Regel über generische Konformität (Abschnitt 3.2) und wären sowohl kompliziert, als auch restriktiver als nötig. Wir wagen jetzt also den Quantensprung und definieren auf Basis dieser Semantik einfach zwei ad hoc Regeln:

Match-Robert Semantik von `like Current`

1. Routinen mit Argumenten vom Typ `like Current` dürfen nur aufgerufen werden, wenn das Ziel-Objekt `Current` oder vom Typ eines generischen Parameters ist.
2. Klassen die Routinen-Argument vom `like Current` in ihrer Schnittstelle enthalten, dürfen nicht als Instanzen von Typparametern verwendet werden.

Diese Regeln ist alles was von der Matching-Theorie in der Praxis gebraucht wird! Mit ihnen ist unmittelbar klar, dass im Endeffekt niemals eine Routine aufgerufen wird, deren Argument vom Typ `like Current` ist. Trotzdem können Klassen, die solche Routinen enthalten fast uneingeschränkt verwendet werden, auch Subtypen werden nicht eingeschränkt.

Dass man *peer routines* nur innerhalb eines generischen Kontexts aufrufen kann, kommt daher dass Match-Typen überhaupt nur durch den match-gebundenen Polymorphismus definiert sind. Natürlich bekommt man generische Parameter in EIFFEL nicht so leicht, wie in HASKELL (siehe Abschnitt 2.8), aber in einem wichtigen Fall sind sie doch günstig: nämlich wenn man Routinen über einer Klasse in der Klasse selbst definiert. Diese Regeln haben also auch überhaupt kein Problem mit „interner Kovarianz“ (siehe Abschnitt 6.3).

Mit den hier gegebenen Regeln lassen sich alle Beispiele aus dieser und anderen Arbeiten zur Kovarianz richtig typisieren, und ich behaupte auch jegliche Software aus der Praxis. Die Regeln vereinen Subtypen (durch leichte Einschränkung von polymorphen Aufrufen) und Match Typen (durch simple Verwendung der normalen beschränkten Generizität). Sie sind (offensichtlich) total einfach, sie unterstützen das vertragliche Programmieren und *last-not-least* in Verbindungen mit den beiden Regeln aus Abschnitt 3 reichen sie aus, um jegliche *catcalls* auszuschließen — man braucht keine erweiterte Typprüfung, keine neuen Typen (man spart sogar die `like query` Typen) und ganz bestimmt kein *recast*.

Wir erhalten also als Ergebnis aller Theorie, zwei einfache Regeln, die nicht einmal eine genaue Entsprechung in der Theorie haben, die aber genau ausdrücken was wir brauchen. Ist das nicht schön?

7.5 Exkurs: Generische Routinen

In den vorigen Unterabschnitten habe ich Typregeln vorgestellt, die Covarianz obsolet machen und damit auch verankerte Typen. Nicht so allerdings die `like argument` Typen, die nicht der Covarianz dienen, sondern... Die Semantik eines `like x` Argument-verankerten Typs ist gleich dem Typ des aktuellen Argumentes für `x`, also verschieden für jeden Aufruf der Routine. Es handelt sich also um einen Spezialfall einer generischen Routinen-Signatur.

Das Konzept der Generischen Routinen kennt man aus anderen Sprachen, ich muss dazu nicht viel sagen, außer vielleicht ganz provokativ: GENERICJAVA hat auch welche. Hier zeige ich wieder eine Parallele zu EIFFELS verankerten Typen auf. Man schaue sich die folgenden Beispiele an:

```
class ANY feature
  clone ( x : ANY ) : like x is ... end
  equal ( x : ANY; y : like x ) : BOOLEAN is ... end
end -- class ANY
```

Wenn man weiß, dass `like x` hier nicht einfach nur für ANY steht, sondern bei jedem Aufruf verschieden jeweils für den Typ des aktuellen Parameters, der an `x` übergeben wird, dann ist die Ähnlichkeit zu generischen Funktionen verblüffend:

```
id :: a → a
(==) :: a → a → Bool
```

Man könnte jetzt wieder auf die Idee kommen, dem verankerten Typ wieder eine Semantik zu verpassen, die auf der generischen basiert. Aber leider ist er viel zu ausdrücksschwach. Von einer simplen Funktion höherer Ordnung kann er zum Beispiel nur träumen:

```
map :: (a → b) → [a] → [b]
```

Und in EIFFEL fragt man sich, was man an Stelle des Fragezeichens schreiben soll. Es gibt keine Lösung:

```
class SEQUENCE[T] feature
  map ( f : FUNCTION[T, ?] ) : SEQUENCE[?] is ...
end -- class SEQUENCE[T]
```

Man kann sich schon fragen, wie EIFFEL-Programmierer 18 Jahre lang mit so einem billigen verankerten Typ an Stelle von generischen Routinen ausgekommen sind. Ich kann mir fast schon vorstellen, dass

die beiden Routinen aus `ANY` die einzigen sind, die diesen verankerten Typ überhaupt benutzen.

8 Abschließende Empfehlungen

Es zeigt sich, dass zu einer Beseitigung von EIFFELS Typlöchern zwei Dinge notwendig sind: Erstens eine Umstellung auf einfache und vollständig konsistente Typregeln, und Zweitens eine Umstellung der Denkweise hinweg von Covarianz. (`Recast` können wir dabei ganz vergessen.) Im dritten Unterabschnitt werden die beiden Aspekte verbunden, hier zunächst einzeln:

8.1 Formale Typregeln

Die aktuelle EIFFEL-Regel ist nicht (konstruktiv) einfach, wenn man die finale statische Analyse mitrechnet, und nicht konsistent, wenn man sie nicht mitrechnet. Glücklicherweise habe ich in dieser Arbeit direktere Regeln vorgestellt.

- Exporteinschränkungen werden einfach abgeschafft. (Abschnitt 3.1 und Meyer et. al. 2003)
- Für generische Konformität gilt die Regel aus Abschnitt 3.2: sie ist einfach und sicher und Varianz-Informationen zu formal generics sind eine gute Dokumentation.
- Für `peer routines` gilt die Regel aus Abschnitt 7.4: ebenfalls einfach und sicher.
- Generische Routinen sind eine mögliche Ergänzung von Eiffel, aber sie bringen ein paar Schwierigkeiten, die ich in dieser Arbeit völlig außer Acht gelassen habe. (Tipp: polymorphe Werte, Bacha et. al. 1998).

Für die beiden „kleinen Typ-Probleme“ (1 und 2) sind die beiden vorgestellten Lösungen offensichtlich perfekt und können direkt umgesetzt werden. Sie führen nicht einmal zu größeren Inkompatibilitäten bei der Umstellung von EIFFEL. Die Regel für die `peer routines` ist auch genial einfach und nützlich, aber da sie mit der kompletten Abschaffung von Covarianz verbunden ist, wird sie sicher auf einige Ablehnung stoßen. Es ist ein wesentlicher Wechsel der Denkweise und der Programmiermuster notwendig — und die neuen Muster zu lernen ist dabei noch viel einfacher, als die alten Muster abzulegen, und das Gehacke mit Covarianz und verankerten Typen aufzugeben.

8.2 Covarianz: Eine Lösung ohne Probleme

Wir haben ja in Teilabschnitt 4.1 gesehen, dass die aktuelle Lösung in EIFFEL eigentlich gar nicht so schlecht ist: alle Typfehler werden statisch gefunden, trotzdem ist die Sprache nicht zu restriktiv. Warum also wollen wir dann Covarianz abschaffen? Weil wir sie nicht brauchen! Warum brauchen wir sie nicht? Wie wir in diesem Abschnitt gesehen haben, lassen sich alle Anwendungsfälle der Covarianz besser mit anderen Mitteln lösen:

- Wir erhalten bessere Entwürfe durch den Verzicht auf Übermodellierung.
- Wir erhalten klarere Schnittstellen durch die Verwendung von Generizität und „Matching“.
- Wir erhalten klarere Typregeln durch den Verzicht auf verankerte Typen.

Covarianz mag in der Modellierung der Welt nützlich sein. Aber zur Modellierung der Software taugt sie nicht, weil sie dem einfachsten Prinzip der vertraglichen Programmierung widerspricht. Die Covarianz hat in EIFFEL so viele (angebliche) Anwendungsfälle, dass man kaum glauben kann, sie könnten auch alle ohne Covarianz gelöst werden. Sich von der Covarianz zu lösen, ist ein erheblicher Paradigmenwechsel, wie ihn meines Wissens bisher noch nie eine Programmiersprache geschafft hat.

Ich glaube Covarianz richtet nicht deshalb so großen Schaden an, weil sie zu Typlöchern führt — diese sind beherrschbar. Nein, Covarianz ist deshalb schädlich weil es dem vertraglichen Denken so stark widerspricht. Wie kann ich denn ernsthaft *Design by Contract* als Konzept als Methodologie verkaufen wollen, wenn schon mein Typsystem dem Konzept widerspricht? *Menschen lernen zu 80% per Nachahmung*. Daraus folgt, dass die Standardbibliotheken vorbildlich sein müssen. Für Typsysteme gilt das umso mehr.

8.3 Schluss

Viele der Empfehlungen sind schon einzeln viel versprechend und außerdem passen sie noch alle zusammen. Wenn man sich aber die großen Veränderungen anschaut, die die Vorschläge zusammengenommen mit sich bringen, so mag man sich schon fragen, ob man nicht einfach bei der Typanalyse à la Meyer (1995) und OOSC2 bleiben sollte.

Andererseits scheinen Meyer und andere mit dieser Lösung unzufrieden zu sein, und da *recast* ein großer Rückschritt wäre, bleibt eigentlich nur die Flucht nach vorn. EIFFEL war bisher trotz seiner Typ-Lücken eine der OO Sprachen mit bestem Typsystem. Mit den obigen Vorschlägen würde es noch einen Schritt wesentlich besser werden: ausdrucksstark und nützlich und immer noch einfach im Vergleich mit MODULA-3, OCAML oder COMPONENT PASCAL. Welchen Weg wirst Du wählen, oh wundervolles EIFFEL: wird's *recast*-Hölle oder der generische Himmel?

9 Literatur

- Martín Abadi and Luca Cardelli, 1996: *On Subtyping and Matching*, In Proceedings of ECOOP'95, LNCS 952, pp. 145..167. Springer-Verlag
- Gilad Bracha, Martin Odersky, David Stoutamire, Philip Wadler, May 1998: *GJ Specification*, <http://www.research.avayalabs.com/user/wadler/gj/Documents/#gj-specification>
- Bruce et. al., May 1995: *On Binary Methods*, Theory and Practice of Object Systems 1 (3), pp. 221..242
- K. Bruce, L. Petersen & A. Fiech, 1997: *Subtyping is not a good `Match' for object-oriented languages*. In ECOOP Proceedings. LNCS 1241. Springer, Berlin, pp. 104..127
- Peter Canning et. al., 1989, *F-Bounded Polymorphism for Object-Oriented Programming*, Fourth International Conference on Functional Programming Languages and Computer Architecture, pp. 273..280
- Luca Cardelli, 1997: *Type Systems*, Handbook of Computer Science and Engineering, CRC Press
- Dominique Colnet, Luigi Liquori, 2000, *Match-O, a Dialect of Eiffel with Match-Types*, <http://citeseer.nj.nec.com/colnet00matcho.html>
- William Cook, 1989: *A Proposal for Making Eiffel Type Safe* The Computer Journal
- Paul Hudak et. al., 1998: *A Gentle Introduction to Haskell*, <http://haskell.org/tutorial>
- Bertrand Meyer, 1986: *Genericity versus Inheritance*, Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications, pages 391..405
- Bertrand Meyer, 1988 (OOSC): *Object-Oriented Software Construction*, 1st ed., Prentice Hall
- Bertrand Meyer, 1992 (ETL): *Eiffel: The Language*, 1st ed., Prentice Hall
- Bertrand Meyer, 1995: *Static Typing and other Mysteries of Life*, Keynote at OOPSLA, Austin/Texas
- Bertrand Meyer, 1997 (OOSC2): *Object-Oriented Software Construction*, 2nd ed., Prentice Hall
- Bertrand Meyer et. al., 2003 (recast proposal): *Competent Compilers can catch all Catcalls*, <http://www.inf.ethz.ch/~meyer/ongoing/-covariance/recast.pdf>, 25. April 2003
- Peter Wegner, June 1990: Concepts and Paradigms of Object-Oriented Programming, expanded script of OOPSLA'89 keynote
- Robert Will, 2003: *Caring for Covariance with Complete Contract Conformance*, <http://www.stud.tu-ilmenau.de/~robertw/headings/cgi/~robertw/eiffel/covariance3.htm>, 16. Juli 2003