Prüfer:	Prof. Dr. Gunzenhäuser
Betreuer:	Dipl.-Inform. Markus Geltz

Beginn am:	1.10.1993
Beendet am:	31.3.1994
CR-Nummer:	D.2.2, D.2.6, D.3.3

Studienarbeit Nr. 1299

Verwendung von Typen in Smalltalk

Fritz Hohl

Verwendung von Typen in Smalltalk

Kurzfassung

Der Text beschäftigt sich mit den Folgen, der Einführung von Typen in das eigentlich ungetypte Smalltalk. Nach einer kurzen Einführung in die Eigenschaften von Typsystemen für Smalltalk und dem Versuch deren Klassifikation werden vier existierende Ansätze, die in diese Richtung gehen, vorgestellt. Anschließend wird ein weiterer Ansatz und dessen Implementierung vorgestellt sowie einige Werkzeuge, die es einem Programmierer erlauben bestehenden und neuen Quellcode zu typisieren. Der neue Ansatz, NTST, beruht dabei auf einem Typfragment in Smalltalk-80 4.1. Es folgt eine Untersuchung einiger Möglichkeiten, die Unterstützung des Programmierers durch die Verwendung der Information zu verbessern, die durch Typen gewonnen werden können. Es zeigt sich, daß diese Verbesserung u.a. dazu führt, daß ein Programmierer weniger Code lesen muß um die Wirkungsweise einer Methode zu verstehen.

Aufgabenstellung

Verwendung von Typen in Smalltalk

Smalltalk ist als vollkommen ungetypte Sprache bekannt, dennoch enthalten die letzten Implementierungen von Smalltalk-80 Möglichkeiten zur Deklaration von Typen, die jedoch nicht genutzt werden. Dabei wäre eine Typinformation für Variablen öfters von Vorteil, z.B. beim Browsen von Nachrichtenräumen.

In der Studienarbeit soll untersucht werden, an welchen Stellen des Systems Unterstützung durch Typen sinnvoll wäre. Dies soll an einem kleineren, schon bestehenden Beispiel (ein Mini-Bibliothekssytem) untersucht werden.

Der Kern der Arbeit wird dabei von folgenden Fragen umrissen:

- Wo könnte der Benutzer bei der Untersuchung bereits bestehender Programme (beim Browsen) unterstützt werden?
- Sind Vorteile in einem gemischten System (getypte und ungetypte Variablen) zu erwarten?

Die Arbeit kann dabei erweitert werden durch eine Veränderung des Browsers, wodurch der Benutzer auch beim Erstellen seiner Software unterstützt wird. Zu ergänzen ist hier z.B. eine Protokollierung noch fehlender Methoden.

Anmerkung zur obigen Aufgabenstellung:
Im Verlauf der Studienarbeit erwies es sich, daß eine Bezugnahme auf das Mini-Bibliothekssystem keine neuen Erkenntnisse bringen würde, so daß es in der folgenden Arbeit nicht mehr erwähnt wird. Da kein Typsystem für einen Smalltalkdialekt vorlag und die Einflußnahme eines solchen auf die Programmierung kaum "trocken" vorstellbar ist, mußte ein derartiges Typsystem erst geschrieben werden. Die Komplexität einer solchen Sprachkomponente in einer objekt-orientierten Sprache ist relativ hoch, daher nimmt in der vorliegenden Arbeit die Beschreibung des Typsystems und seiner Implementierung einen erheblichen Anteil des Platzes ein (analog zur Zeit, die dieser Komplex zur Verwirklichung benötigte).

Inhaltsverzeichnis

1 Einleitung 1
2 Begriffe 3
2.1 Was ist Smalltalk? 3
2.2 Was ist Typen? 5
3 Typisierung und Objekt-Orientiertes Programmieren 9
4 Typsysteme für Smalltalk 12
4.1 Klassifikation von Typsystemen in Smalltalk 12
4.2 Borning und Ingalls 1982 15
4.3 TS 1986-1990 17
4.4 Palsberg und Schwartzbach 1991 18
4.5 Strongtalk 1993 19
5 Das Typfragment in Smalltalk-80 4.1 20
5.1 Die Typsprache 20
5.2 Die Semantik der Typausdrücke 22
5.3 Diskussion des Typfragmentes 22
6 Eine Erweiterung des Fragmentes in Smalltalk-80: NTST 23
6.1 Ausdehnung auf andere Sprachmittel 23
6.2 Das Typsystem von NTST 24
6.3 Diskussion des Typsystems 28
6.4 Implementierung 30
7 Werkzeuge zur Unterstützung der Typisierung 35
7.1 Der TypeClassBrowser 35
7.2 Der GlobalsBrowser 40
7.3 Implementierung 40
8 Programmierunterstützung durch Typen in Smalltalk-80 42
8.1 Änderungen im Browser 45
8.2 Implementierung 48
8.3 Ein Erfahrungsbericht 49
9 Zusammenfassung 53
10 Ausblicke 54

Anhänge 55

A Anmerkung zum Quellcode 55
B Installation der Software 57
C Sonstiges 59
C.1 Beispiele von Typausdrücken aus den Klassenkommentaren 59
C.2 Smalltalk-80 4.1 Parsebäume 59
C.3 Bemerkungen zu dieser Ausarbeitung 61

1 Einleitung

Smalltalk ist als Konzept wegweisend für den Bereich der graphischen Benutzerführung[1] gewesen. Viele der Elemente des Smalltalkdesigns finden sich heute in Anwendungen und Entwicklungen wieder, die mit dem Einzug von Computern in das tägliche Leben von vielen Menschen benutzt werden. Dazu zählen die graphische Interaktion einer Anwendung mit dem Benutzer durch die Verwendung von Fenstern und Menüs ebenso wie die Verwendung von Zeigeinstrumenten, vor allem Mäusen (in der Tat war dieser Aspekt so erfolgreich, daß sich heute alle kommerziellen Smalltalkdialekte den bestehenden Fenstersystemen anpassen, anstatt wie bis vor wenigen Jahren, ihr eigenes mitzubringen).

Smalltalk ist auch als Programmiersprache ein wichtiger Schrittmacher der objekt-orientierten Programmierung gewesen und stellt heute den wichtigsten Vertreter eines dynamischen, objektorientierten Programmierparadigmas dar. Trotzdem erlangte Smalltalk nie die Verbreitung einer Sprache wie C++, obwohl sie auf einem einheitlichen, einfachen Konzept beruht. Dies ist zum Teil darauf zurückzuführen, daß erst in letzter Zeit Smalltalksysteme zur Verfügung stehen, die den effizienten Code in der monolithischen Form herstellen können, die die softwareproduzierende Industrie fordert.

Ein anderer Grund für die mangelnde Verbreitung mag aber auch darin begründet sein, daß die Programmierung in Smalltalk zwar alte Probleme löst, aber neu hervorruft und andere auch weiter ungelöst läßt. Ein Beispiel für ein neu enstandenes Problem ist das der Navigation durch das Klassensystem. Eine Umgebung wie Smalltalk-80 stellt dem Programmierer eine Vielzahl von Klassen zur Verfügung und eine noch größere Menge an Methoden. Der Programmierer muß darin die Klasse und die Methode finden, die ein Teilproblem seiner Anwendung zu lösen in der Lage ist oder er muß feststellen können, daß noch kein Code dazu exisitiert. Im ersten Fall muß er herausfinden, wie Objekte dieser Klasse zu erzeugen und anzuwenden sind und im zweiten Fall muß er den benötigten Code schreiben, indem er das Problem in weitere Teilprobleme aufspaltet und wieder nach deren Lösungen im System sucht. Ein Smalltalkprogrammierer verbringt daher den größten Teil seiner Zeit damit, herauszufinden, was bestehender Code tut und wie er zu modifizieren ist.

Ein großes Problem dabei ist herauszufinden, von welcher Klasse ein Objekt ist, das an eine Variable gebunden wird, das das Resultat einer Methode ist oder das als Parameter für eine Methode gebraucht wird. Die Schwierigkeit dabei ist, daß die Variablen in Smalltalk, also die Orte, an denen Objekte referenziert werden, ungetypt sind, das heißt, das an eine Variable jedes beliebige Objekt gebunden werden darf und das System erst zur Ausführungszeit feststellen kann, um welches es sich im Moment handelt. Um trotzdem an diese Information zu kommen gibt es zwei Wege: Der eine ist, sich auf eine Dokumentation zu verlassen, in der Hinweise zu den Variablen verzeichnet sind, dazu muß eine Dokumentation, also zumindest ein Klassen- bzw. Methodenkommentar vorliegen, und diese Angaben müssen stimmen. Vor allem ersteres ist schon im Smalltalk-80 Grundsystem nicht bei allen Methoden der Fall. Der zweite Weg ist das Ermitteln der Klassenzugehörigkeit durch das Durchspielen von Smalltalk-

1. Jede Rollenbezeichnung in diesem Text wie z.B. “Benutzer” muß geschlechtsneutral gelesen werden

code im Kopf. Dazu wird der Aufrufweg, der zu einem Objekt führt bis zu einer Nachricht ermittelt, von der man weiß, welches Objekt sie erzeugt. Dieses Vorgehen zwingt den Programmierer bisweilen mehrere Methoden durchzulesen und bedeutet erhebliche mentale Arbeit für ihn.

Ein anderes Problem ist das kooperative Programmieren mehrerer Menschen in gemeinsamen Klassenhierarchien. Hier muß sichergestellt sein, daß jede Referenz die Art von Objekten bezeichnet, an die der Programmierer zum Programmierzeitpunkt gedacht hat. Das ist ein nichttrivialer Aspekt, weil die Klassenzugehörigkeit in Smalltalk nur besagt, daß sich ein Objekt zu einem Zeitpunkt genauso verhält, wie es die Klasse in diesem Augenblick durch ihre Implementation definiert. Das Verhalten einer Klasse kann sich aber in einem dynamischen System von Zeit zu Zeit wandeln, so daß nicht sichergestellt ist, daß der Code, der diese Referenz verwendet, noch richtig arbeitet. Dieses Problem wird in einem verteilten Smalltalk, einem immer wichtiger werdenden Gebiet, noch potenziert.

Falls man versucht, diese Probleme zu lösen, muß man darauf achten, die Vorteile, die Smalltalk einem Programmierer bietet, wie die Möglichkeit in kurzer Zeit mächtige Anwendungen schreiben zu können und die Freiheit einer nicht reglementierenden Umgebung, nicht allzu sehr zu beschneiden.

Eine Lösung für das erste Problem und vielleicht die Basis für die Lösung des zweiten wäre die Einführung von Typen in Smalltalk. Typen sind Restriktionen über dem Inhalt von Variablen; sie stellen sicher, das an diese nur Objekte genau spezifizierter Klassen gebunden werden können. Einen Versuch, Smalltalk-80 mit Typen auszustatten stellt der folgende Text vor.

2 Begriffe

Um in die Materie einsteigen zu können sind einige Begriffe und Sachverhalte zu erklären, um auf dieser Basis die für den Themenkomplex notwendigen Ideen entwicklen zu können. Der nächste Abschnitt gibt daher einen kurzen Überblick über Smalltalk und der übernächste beschäftigt sich mit dem Gebiet der Typen.

2.1 Was ist Smalltalk?

Smalltalk ist ein Programmiersystem, das etwa ab 1970 am Palo Alto Research Center (PARC) der Firma Xerox entwickelt wurde [Ho87]. Dieses Programmiersystem bedingte eine für die damalige Zeit ungewöhnlich leistungsstarke und teure Hardware, die zum Teil erst für dieses Projekt in einem Rechner zusammengefasst wurde. Sie bestand aus einem hochauflösenden Rasterbildschirm und einer Maus und sollte an jedem Arbeitsplatz soviel Verarbeitungskapazität bereitstellen, daß dort das Smalltalksystem lokal betrieben werden konnte. Das Ziel des Smalltalkprojektes war es, eine benutzerfreundliche und interaktive Oberfläche zur Verfügung zu stellen, die der Benutzer nach seinen Bedürfnissen erweitern kann. Den Referenztext zu Smalltalk stellt heute [GR89] dar, ein Buch das das originale Smalltalk-80 von Xerox beschreibt.

Das Programmiersystem Smalltalk-80 beinhaltet als wichtigste Komponente die Programmiersprache Smalltalk-80, obwohl es nur wenige Implementierungen gibt, die nur letzteres bereitstellen, so daß beides nur schwer voneinander zu trennen ist. Smalltalk ist eine reine objekt-orientierte Programmiersprache, d.h. die Elemente der Programmierung sind Objekte.

Objekte sind “Dinge”, die auf bestimmte Botschaften mit bestimmten Antworten reagieren. Dazu können sie auf private Daten zugreifen, die von anderen Objekten nicht gesehen werden können (das Prinzip des “Information Hiding”). Das Verhalten von Objekten in Smalltalk wird dabei von ihrer Klasse bestimmt. Eine Klasse besteht aus einem “Bauplan” für Objekte und aus Codestücken, die das Verhalten von Objekten definieren. Da die Klassen in Smalltalk ebenfalls Objekte sind, haben auch sie private Daten und eine Klasse, die ihr Verhalten bestimmt (die sogenannte Metaklasse). Diese Codestücke werden *Methoden* genannt. Methoden entsprechen Prozeduren bzw. Funktionen in anderen Programmiersprachen, d.h. sie besitzen eventuell Parameter und geben Resultate zurück, wobei sie Seiteneffekte (wie das Öffnen eines Fensters) haben können. Auf Objekte kann nur dadurch Einfluß genommen werden, daß man ihnen sogenannte *Nachrichten* schickt. Diese Nachrichten, die aus einem Schlüsselwort (*Selektor* genannt) und Referenzen auf die Parameter, die selbst wieder Objekte sind, bestehen, lösen beim Empfängerobjekt den Aufruf der Methode aus, die durch diesen Selektor referenziert wurde. Das Ergebnis einer Nachrichtenübermittlung ist ein Resultatsobjekt, das weiter verwendet werden kann. Die möglichen Anweisungen in Smalltalk sind neben Nachrichtensendungen Zuweisungen und die Rückgabe eines Wertes in einer Methode. Das hat zur Folge, daß Befehle, die in Sprachen wie C oder Pascal eigene Kategorien sind, wie z.B. Schleifen oder arithmetische Funktionen in Smalltalk als Nachrichten an Objekte erklärt und implementiert werden. Ein Beispiel dafür ist der Ausdruck `1 + 2`. In Smalltalk bedeutet er, daß dem Objekt 1 (ein Zahlenobjekt) die Nachricht ‘+ 2’, also ein Selektor ‘+’ und ein

Parameter geschickt wird. Dieser Parameter ist wieder ein Objekt. Diese Nachricht stößt die Methode '+' an, die in der Klasse von 1 definiert ist. Das Rückgabeobjekt dieser Methode ist wieder eine Zahl, die die Summe von Empfänger und Parameter darstellt, in diesem Fall 3.

Der "Bauplan" einer Methode bestimmt die Struktur von Objekten dieser Klasse, die *Instanzen* dieser Klasse genannt werden. Diese Struktur ist eine Liste von Variablen, den *Instanzenvariablen*, die zusammen die private Datenbasis eines Objektes darstellen. Variablen sind die einzige Möglichkeit in Smalltalk Objekte zu referenzieren, ein Objekt, das an keine Variable gebunden ist, kann nicht mehr erreicht werden und darf daher gelöscht zu werden. Variablen in Smalltalk sind **ungetypt**, d.h. eine Variable kann Objekte jeder Klasse binden.

Smalltalk bietet allerdings für einige Klassen Möglichkeiten an, Objekte durch eine eigene Darstellungsform direkt zu erzeugen. So werden wie im Beispiel oben Zahlen aus ihrer Darstellung im Quelltext direkt in Objekte der entsprechenden Klassen verwandelt (ansonsten müßte man ja an eine Variable das Ergebnis einer Objekterzeugungsnachricht binden). Das Gleiche gilt für Objekte der Klassen Symbol, Array, String, Character und Block.

Klassen sind in eine Hierarchie eingebunden, in der jeder Knoten außer dem Wurzelknoten genau einen Vaterknoten, sie sogenannte *Superklasse* hat (einfache Vererbung im Gegensatz zur Mehrfachvererbung). Demgegenüber heißen die Tochterknoten der Superklasse *Subklassen* dieser Klasse. Die Hierarchie dient dazu, Implementation in Form von Methoden und "Bauplänen" zu *vererben*. Vererbung bedeutet in diesem Zusammenhang, daß Subklassen alle Methoden sowie den Objektbauplan der Superklasse ohne weitere Definition benutzen können, d.h. ein Objekt einer Subklasse enthält mindestens die gleichen Instanzenvariablen wie ein Objekt der Superklasse und es reagiert auf dieselben Nachrichten. Damit nun Subklassen weitere Funktionalität anbieten können, dürfen sie weitere Instanzenvariablen definieren, weitere Methoden anbieten und ererbte Methoden durch eigene Methoden *überdefinieren*. In letzterem Fall wird dann bei einer entsprechenden Nachricht nicht die Methode der Superklasse ausgeführt, sondern die neue Methode, obwohl der Selektor in diesem Fall gleich ist (allerdings können andere Parameter verlangt werden). Die Gesamtheit aller ausführbaren Methoden einer Klasse wird das *Protokoll dieser Klasse* genannt.

Das Programmieren in Smalltalk erfolgt auf folgende Weise: Der Programmierer identifiziert die Arten von Objekten, die er für die Lösung seines Problemes benötigt. Er versucht eine Antwort auf die Frage zu finden, ob es diese Arten von Objekten bereits fertig gibt, ob es Arten gibt, die zumindest einen Teil der Funktionalität bereitstellen oder ob er durch eine Klassendefinition eine neue Art von Objekten erschaffen muß. Falls es dann auf die eine oder andere Weise eine Klasse gibt, die die gewünschten Objekte bereitstellt, braucht er nur noch ein Objekt dieser Klasse zu erzeugen und kann dann dessen Funktionalität benutzen. Dieser Prozeß ist rekursiv, weil er um eine Klasse von Objekten zu erschaffen oder zu verändern wieder Teilprobleme durch die Benutzung von anderen Objekten lösen muß. Die Programmierung in Smalltalk bedingt also auch für erfahrene Programmierer ein stetiges Durchsuchen des Klassenraumes. Das aus diesem Grund benötigte Werkzeug nennt sich in Smalltalk *Browser*. Mit ihm kann man nach Klassennamen suchen, sich Klassen anzeigen lassen und neue definieren, nach Methoden suchen und neue schreiben und weitere Werkzeuge benutzen. Wenn ein Programmierer wissen will, was Objekte einer Klasse tun oder *wie* sie es tun, dann

muß er die Dokumentation und den Quellcode studieren. Beim Studium der Dokumentation muß ein Benutzer in etwa wissen, was er wissen will (etwa wenn er den richtigen Band eines Handbuches sucht), und nicht immer sind solche Texte phänomenologisch geordnet. Daher ist es üblich, daß ein Programmierer Auszüge aus einer Methode verwendet, die einen ähnlichen Effekt zeigt wie der, den er erzielen will. Diese Vorgehensweise kann man an folgendem Beispiel nachvollziehen: Ein Programmierer will, daß ein Fenster aufgeht, wenn einem bestimmten Objekt eine Nachricht geschickt wird. In diesem soll die Struktur dieses Objektes in einer bestimmten Weise dargestellt werden. Er weiß aus seiner Erfahrung daß eine ähnliche Funktionalität durch die Nachricht `inspect` erbracht wird, die jedes Objekt versteht (falls er diese Erfahrung nicht hat, kann er beim Durchsuchen des Klassenraumes auch zufällig daraufstoßen). Er versucht die Methode herauszufinden, die bei `inspect` ausgeführt wird. Dazu kann er sich z.B. im Browser alle die Klassen anzeigen lassen, die genau das tun. Hat er die richtige Methode gefunden, dann kann er diese kopieren und seinen Wünschen entsprechend modifizieren. Diese Änderung wird in den allermeisten Fällen zu einer neuen Methode führen, die das Gesamtssystem etwas erweitert und die er separat testen kann (das ist der Aspekt der inkrementellen Programmierung in Smalltalk).

Die Vorteile, die für Smalltalk-80 geltend gemacht werden können sind:

- Smalltalk-80 enthält eine komplette Programmierungebung
- Es stellt viele grundlegende Datenklassen (Arrays, Strings, Dictionaries, ..) bereit
- auch umfangreiche Anwendungen können in vergleichweise kurzer Zeit erstellt werden
- Programme in Smalltalk sind vergleichsweise einfach zu warten

Dem stehen einige Nachteile gegenüber:

- Smalltalk ist auf einen Benutzer ausgerichtet und es gibt keine Unterstützung für kooperatives Arbeiten
- Die Effizienz einer Anwendung kann nicht vollständig kontrolliert werden; sie ist gewöhnlich geringer als die von vergleichbaren Programmen in C
- Anwendungen sind nur schwer von der Programmierumgebung zu trennen und auszuliefern

2.2 Was sind Typen?

Historisch gesehen sind Typen Zusagen über die Struktur oder den Wertebereich eines Datums. Diese Ansicht finden wir in den prozeduralen Programmiersprachen, die Typen verwenden. Was bedeutet das? Prozedurale Sprachen beruhen auf dem Prinzip, daß es eine Liste von Anweisungen für einen Rechner gibt, die dieser in einer bestimmten Reihenfolge durchläuft. Diese Liste nennt man Programm. Den Anweisungen sind Daten zugeordnet, auf denen die Anweisungen operieren. Diese Daten werden durch Variablen referenziert. Variablen sind Platzhalter für Daten und haben einen in einem Bereich eindeutigen Namen.

Um sicherzustellen, daß die tatsächliche Variablenbelegung mit Daten im Programmlauf mit der Belegung in Einklang steht, die sich der Programmierer vorgestellt hat, taucht in der

Begriffswelt der Programmiersprachen der "Typ einer Variable" auf. Um diesen Begriff zu erkären muß man darauf eingehen, wie Daten in prozeduralen Programmiersprachen aufgebaut werden. Zum einen gibt es die Datenarten, die Symbole mit einer Bedeutung darstellen. Zu diesen Datenarten zählen alle Zahlenarten, Buchstaben und sonstige lineare Größen. Dann gibt es die Strukturierungsmöglichkeiten dieser "atomaren" Datenarten. Dazu zählt die Sequenz von Daten einer Art (oft Array genannt) oder die Kollokation von Daten verschiedener Arten (oft als Record bezeichnet) sowie Mengen von Daten (Sets). Datenkomplexe, selbst wieder eine Datenart, bestehen aus Strukturen von Datenkomplexen oder atomaren Datenarten. Variablen können solche Datenkomplexe als Wert haben. Diese Komplexe kann man sich als Strukturbaum vorstellen, in dem die atomaren Datenarten die Blätter des Baums darstellen. Wenn man nun will, daß der Programmierer definieren muß, wie ein Komplex aussieht, der an eine Variable gebunden werden kann, dann muß man ihm ein Mittel in die Hand geben, Datenkomplexe zu beschreiben. Diese Beschreibungen sind die Typen. Analog zu den atomaren Datenarten gibt es atomare Datentypen, die nicht weiter beschrieben werden können und die von der Programmiersprache vorgegeben sind. Diese atomaren Datentypen beschreiben die Semantik ihrer Daten und nicht ihre Struktur, da diese nur aus einem einzelnen Wert besteht. Darauf aufbauend gibt es komplexe Datentypen, die nicht die Gesamtsemantik ihrer Daten beschreiben, sondern ihre Struktur.

Beispiel:

```
A : Integer
```

A ist eine Variable vom atomaren Datentyp Integer. Integer sind ganze Zahlen zwischen z.B. -32767 und +32768.

```
B : Array of Integer
```

B ist eine Variable vom komplexen Datentyp Array of Integer, d.h. sie enthält einen Datenkomplex, der aus einer geordneten Sequenz von Integerdaten besteht.

Daten werden Anweisungen in Form von Parametern zugeordnet. Parameter werden bei der Definition von Anweisungen durch Variablen und beim Abschicken von Anweisungen durch die Daten selber dargestellt (diese können wieder durch Variable gebunden sein). Das Typsystem eines Übersetzers prüft bei der Übersetzung von Anweisungen, ob die Daten beim Aufruf einer Anweisung (bzw. der Typ der Variable oder der Funktion, die für die Daten stehen) kompatibel mit den Typen sind, die für die Parametervariablen der Prozedurdefinition angegeben sind.

Diese Regeln lauten:

1) Ein Typ ist mit einem anderen kompatibel, falls sie gleich sind. Das ist für atomare Typen trivial festzustellen, für komplexe Typen muß entweder der Strukturaufbau verglichen werden oder die Typen müssen die gleichen Bezeichnungen haben (in einem System, in dem man Typen Namen geben kann). Zwei komplexe Strukturen sind gleich, falls die Knoten beider Typen gleich sind und die Blätter des Strukturbaumes kompatibel sind.

2) Ein atomarer Typ ist mit einem anderen kompatibel, falls sein Wertebereich eine Teilmenge des Wertebereiches des anderen Typs ist. Man beachte, daß diese Relation anti-

symmetrisch ist, da auch die Teilmengenrelation antisymmetrisch ist.

Die letzte Regel gilt nur für bestimmte atomare Datentypen (etwa Ganzzahlen).

Typen sind in prozeduralen Sprachen also Zusagen über die Struktur eines Datenkomplexes, der an eine bestimmte Variable gebunden werden kann oder Zusagen über den Wertebereich dieser Daten.

Betrachten wir nun noch einige Begriffe im Zusammenhang mit Typen, die wir später benötigen werden:

Ein Typsystem kann **strenges Typisieren** (strong typing) fordern oder **schwaches Typisieren** erlauben (weak typing). Beim strengen Typisieren kann der Typprüfer alle Tests zur Übersetzungszeit vornehmen, da er bereits zu diesem Zeitpunkt alle Informationen über Variablenwerte besitzt, während er beim schwachen Typisieren vieles erst zur Laufzeit einer Anwendung überprüfen kann. Die Folge davon ist ein größeres Laufzeitsystem (das ja die Prüfungsalgorithmen enthalten muß) und die Möglichkeit, daß Typfehler erst zur Laufzeit (und womöglich dem Anwender) gemeldet werden [MMM90].

Ein Typsystem macht eine Programmiersprache **typsicher** (type-safe), falls die möglichen Werte einer Variablen immer durch den Typ dieser Variablen beschrieben werden.

Ein komplexer Typ, der ja einem Datenkomplex entspricht, hat eine innere Struktur. Um diese Struktur auszudrücken besitzt er Attribute, die seinen Aufbau repräsentieren. Diese Attribute, soweit sie wieder Typen referenzieren heißen Parameter. Typen, die Parameter haben, heißen **parametrische Typen**.

Beispiel:

```
C : Array of Array of Integer
```

ist ein Typ, dessen Parameter das Array of Integer ist. Dieser Parameter ist wieder ein Typ mit einem Parameter (Integer).

Typsysteme können **Typvariablen** benutzen, um Typen zu referenzieren. Damit kann z.B. sichergestellt werden, daß zwei Variablen vom selben Typ sind, ohne daß diese Typen durch einen Namen referenziert werden müssen.

Typsysteme können sich der **Typinferenz** bedienen, um die Typen zu erraten, die durch Typvariablen dargestellt werden. Typinferenz wird ebenfalls dazu benutzt um Typen von Variablen zu errechnen, die nicht typisiert sind (Typinferenz in ungetypten Sprachen wird dazu verwendet, Code besser optimieren zu können).

Als Grund für die Einführung von Typen wurde oben die geforderte Übereinstimmung der Ist-Belegung der Variablen mit der Soll-Belegung angeführt. Welche Vorteile bietet diese Übereinstimmung? Da ist zum einen die Tatsache, daß Programme sehr viel mehr gelesen als geschrieben werden, d.h. es ist für ein Programm nicht nur wichtig, fehlerfrei zu sein; es muß

auch nach Jahren wieder vom Programmierer oder anderen Menschen verstanden werden können. Der zweite Aspekt ist die Programmierung einer Anwendung in Modulen oder Objekten. Falls ein System durch das (eventuell kooperative) Programmieren von einzelnen Teilen entsteht, die einander benutzen, dann muß *vor* der Programmierung der Module feststehen, wie die sichtbare Hülle dieser Module aussieht. Hier helfen Typen, da sie Fehler aufdecken können, die durch die falsche Verwendung dieser Hüllen oder Schnittstellen entstehen können. Wenn z.B. ein Programmierer ein Modul verwendet, indem er eine Prozedur darin aufruft und diesem Aufruf einen Parameter von Typ Integer mitgibt, obwohl die Schnittstellendefinition besagt, daß der Parameter ein String sein muß, dann kann dieser Fehler bereits bei der Übersetzung des Aufrufs entdeckt werden. Neben diesen Vorteilen bei der Programmierung können Typen jedoch auch dazu beitragen, die Performanz von Programmen zu erhöhen. So kann der Code für eine Addition kleiner oder schneller sein, wenn bereits zur Übersetzungszeit klar ist, von welchem Typ eine Variable ist (im Gegensatz zu einer Sprache ohne Typen, bei der erst zur Laufzeit über den Inhalt einer Variablen entschieden wird).

Noch ein Wort zum Sprachgebrauch in diesem Text: Ein Programmierer typisiert eine Variable, indem er für sie einen Typ deklariert. Diese Variable heißt dann typisiert oder getypt. Ein Teil des Typsystems, der Typprüfer überprüft die Kompatibilität von Typen zu einem bestimmten Zeitpunkt. Tritt dabei ein Verstoß gegen die Typregeln auf, dann informiert er den Programmierer über einen Typfehler.

3 Typisierung und Objekt-Orientiertes Programmieren

Im Gegensatz zur Ansicht, daß Typen in prozeduralen Sprachen Zusagen über die Struktur oder den Wertebereich eines Datums sind, bedeuten Typen in objekt-orienterten Sprachen Zusagen über das *Verhalten* von Daten. Dies ist notwendig, weil Daten in diesen Sprachen opak sind, d.h. die Struktur von Datenkomplexen ist nicht von außen einsehbar. Außerdem sind diese Datenkomplexe “aktiv”, d.h. sie reagieren auf bestimmte “Botschaften” in einer bestimmten Weise. Diese Daten heißen in objekt-orientierten Programmiersprachen *Objekte*. Objekte sind also nicht nur passive Elemente, sie allein bestimmen die Dynamik einer Anwendung, sie sind die “Urheber” von Aktionen. Falls eine Variable als von einem Typ deklariert ist, dann bedeutet das, daß nur Objekte an diese Variablen gebunden werden dürfen, die eine bestimmte *Rolle* übernehmen können. Die Typsysteme für objekt-orientierte Sprachen unterscheiden sich im Wesentlichen darin, wie diese Rollen definiert sind, d.h. was genau sie unter einem Typ verstehen. Sehen wir uns die verschiedenen Typbegriffe einmal an.

Im einfachsten Fall wird der Typbegriff mit dem Verhalten von Objekten gleichgesetzt, d.h. ein Typ ist durch die Referenz auf ein Objekt oder eine Klasse von Objekten definiert. Wenn sich die Implementierung der Objekte auf ihr Verhalten auswirkt, dann ändert sich auch die Definition des Typs, weil dieser Zusagen über ein bestimmtes Verhalten macht. Die Folge davon ist, daß die Werte von Variablen, die mit diesem Typ deklariert worden sind, plötzlich anders reagieren, als dies der Programmierer vorausgesehen hat. Dieser Typbegriff ist offensichtlich unbrauchbar, denn er verläßt sich darauf, daß Änderungen in Objektimplementierungen “gutartig” sind, und kein Typsystem realisiert ihn. Für Programmiersprachen, die kein Typsystem haben und dadurch Implementierung und Typ gleichsetzen (wie Smalltalk), ist dieses Problem jedoch charakteristisch.

Der nächste Typbegriff setzt ebenfalls Implementierung mit Typ gleich, trägt aber gleichzeitig dafür Sorge, daß Änderungen in der Klassenimplementierung nur dann durchgeführt werden dürfen, wenn sie den Typ der Klasse nicht ändern. Dazu betrachten die Übersetzer die Methoden der Objekte vor der Änderung und verhindern die Änderung, wenn sich die Anzahl der Parameter, der Typ der Parameter oder der Typ des Rückgabewertes so verändert, daß vorhandener Code diese Methoden nicht mehr korrekt benutzen kann. Hier beruht der Typbegriff also auf dem Verhalten von Objekten zu einem Programmierzeitpunkt und alle Änderungen werden abgewiesen, die dieses Verhalten modifizieren.

Der letzte Typbegriff trennt die Klasse vom Typ. Eine Klasse (bzw. die Implementierung eines Objektes) wird als der Ist-Zustand des Verhaltens von Objekten aufgefasst, während der Typ den Soll-Zustand kennzeichnet. Die Typen werden dann durch die Methodensignaturen definiert, die ein Typ mindestens aufweisen muß. **Methodensignaturen** bestehen aus dem Namen einer Methode, der Anzahl ihrer Parameter und dem Typ der Parameter und des Rückgabewertes. Werden diese Typen unter eigenen Namen als Element in die Sprache eingeführt, dann heißen sie **Interfaces**. Falls sie nur aus der Aufzählung der Methodensignaturen bestehen, heißen sie **Signaturtypen**.

All diese Typbegriffe leiden jedoch unter dem Problem, daß sie die eigentliche Idee von Typen in objekt-orientierten Sprachen nicht so recht verwirklichen. Was heißt das? Ein Typ ist der Eindruck, den ein Programmierer vom Verhalten eines Objektes zu einem Zeitpunkt, nämlich dem Programmierzeitpunkt, hat. Dieses Verhalten ist in einer Sprache, die eine Änderung der Objektimplementierung nach diesem Zeitpunkt nicht mehr zuläßt, für die Laufzeit der Anwendung garantiert (das gilt für viele klassische Compilersprachen). In einer Sprache jedoch, die eine Änderung des Verhaltens auch nach diesem Zeitpunkt zuläßt, müssen Maßnahmen getroffen werden, daß sich diese Modifikationen nicht störend auf den existierenden Code auswirken. Betrachten wir dazu das Verhalten von Objekten näher. Das Verhalten von Objekten besteht aus dem Verhalten seiner Methoden (eingeschränkt auf die Methoden, die der Programmierer wirklich verwendet). Das Verhalten der Methoden besteht aus der Reaktion auf:

- den Methodennamen
- die Anzahl der Parameter
- die Reihenfolge der Parameter
- den Typ der Parameter.

Diese Reaktion besteht in der Rückgabe eines bestimmten Objektes sowie den Seiteneffekten. Das Problem ist, daß die bekannten objekt-orientierten Programmiersprachen nicht spezifizieren können, wie das Rückgabeobjekt von den Parametern abhängt und wie die Seiteneffekte aussehen. Diese Semantik von Methoden kann daher weder deklariert noch bei Bedarf nachgeprüft werden. Das hat zur Folge, daß Typsysteme zwar prüfen können, ob der Typ von Parametern vor und nach einer Änderung übereinstimmt, nicht aber, ob diese Methode auf einmal etwas völlig anderes macht.

Beispiel:

Vorher:

```
blabla: a<Integer> returns: Boolean
(a = 5) ifTrue: [^true]
```

Nachher:

```
blabla: a<Integer> returns: Boolean
(a = 5) ifTrue: [^false]
```

Diese Verhaltensänderung kann von den im Folgenden vorgestellten Typsystemen nicht erkannt werden; sie beschränken sich alle auf die Prüfung der Erhaltung der “sichtbaren” kriterien wie Parameter- und Rückgabetypen.

Ein Schritt in die Richtung, die Erhaltung der Semantik zu überprüfen stellt das Konzept dar, zu jeder Methode eine Vor- oder Nachbedingung sowie eine Invariante angeben zu können, wie dies in Eiffel der Fall ist. Änderungen der Methoden sind nur dann zulässig, falls diese Regeln auch nach der Änderung gültig sind. Auch mit diesem Mechanismus kann jedoch nicht geprüft werden, ob alle Seiteneffekte (wie das Öffnen eines Fensters) erhalten bleiben. Dieser Aspekt bleibt zukünftigen Forschungen vorbehalten.

Einen wesentlichen Teil der Flexibilität ungetypter objekt-orientierter Sprachen stellt die

Möglichkeit dar, jedes Objekt, das bestimmte Nachrichten versteht, in dem Kontext zu verwenden, der durch diese Nachrichten gegeben ist [CCHO89]. Dies sind in Sprachen mit Klassen im Wesentlichen die Objekte der Subklasse der Klasse, dessen Objekt sich der Programmierer bei der Programmierung einer Nachricht vorgestellt hat. Natürlich können dazu auch Objekte dienen, die nicht Subtypen dieser Klasse sind und natürlich ist es in einigen Sprachen (besonders in Smalltalk) möglich, Subklassen so zu kontruieren, daß ihre Objekte nicht statt Objekten ihrer Superklassen eingesetzt werden können, aber dieser Gedanke ist eine der Grundlagen der objekt-orientierten Programmierung: Ein Subklassenobjekt kann immer für ein Superklassenobjekt stehen.

Dieser Grundsatz muß auch mit einem Typsystem verwirklicht werden können. Dazu gibt es das Konzept des **Subtyps**. Ein Subtyp kennzeichnet Objekte, die bedenkenlos statt einem Supertyp eingesetzt werden können, d.h. an eine Variable, die als von einem Typ deklariert worden ist, dürfen auch Objekte gebunden werden, die von einem Subtyp dieses deklarierten Typs sind. Anders als z.B. in Smalltalk garantiert das Typsystem, daß durch diese Ersetzung kein Schaden entsteht (dieser Dualität von Implementierung und Spezifikation werden wir im nächsten Kapitel noch einmal begegnen). Diese Subtypen entsprechen in ihrem Gedanken den Unterbereichstypen in prozeduralen Typsystemen. Sofern der Typbegriff nicht auf einer Spezifikation der Klassensignatur, sondern auf der Referenz auf eine separate Entität Typ beruht, ist es darüber hinaus auch möglich, Typen in einer Hierarchie zu klassifizieren. Wie bei den Klassenhierarchien in den Programmiersprachen hat dies den Vorteil, daß Typen durch Vererbung die Definition ihres Supertyps erben (und gegebenenfalls modifizieren) können.

Sehen wir uns nun die Typsysteme an, die für Smalltalk relevant sind.

4 Typsysteme für Smalltalk

Smalltalk ist nach der eigentlichen Sprachdefinition eine ungetypte Sprache. Ein Smalltalk-programmierer kennt die Restriktionen, die Typsysteme in Programmiersprachen einbringen nicht, und er ist eine Umgebung gewohnt, in der alles erlaubt ist, was technisch möglich ist. Wenn man ihn dazu bringen will, Typen zu benutzen, dann darf man seinen Programmierstil und sein Programmiermodell nicht zu sehr einschränken, oder man muß ihm im Gegenzug Vorteile anbieten, die diese Umstellung aufwiegen. Man kann sich darüber streiten, ob getyptes Smalltalk noch Smalltalk ist, ein Dialekt davon oder eine komplett neue Programmiersprache, die sich an Smalltalk anlehnt. Entscheidend wird auch hier sein, ob ein neues Smalltalk günstig und auf vielen Plattformen angeboten wird, ob man das System gern benutzt und ob das Ergebnis den Anforderungen nach Performanz, Größe und Form genügen kann.

Ein Typsystem für Smalltalk muß folgende Fragen klären:

- Wie ändert sich die Sprache?
- Können auch bestehende Klassen typisiert werden?
- Zu welchem Zeitpunkt wird der Typprüfungsprozeß angestoßen?
- Wie stehen Klassen- und Typhierarchien zueinander?

Im Folgenden wird der Versuch einer Klassifikation von Typsystemen unternommen, danach werden vier Typsysteme für Smalltalk vorgestellt.

4.1 Klassifikation von Typsystemen in Smalltalk

Ab hier wollen wir uns nur noch mit Typsystemen beschäftigen, die auf Smalltalk anwendbar sind, insbesondere gehen wir nicht auf Fragen ein, die durch Mehrfachvererbung enstehen.

Die Typsysteme, die für Smalltalk entwickelt wurden, lassen sich nach der Art der Elemente des Typsystems und dem Verhältnis zwischen Art der Vererbung und Art der Typhierarchie unterscheiden.

Die Elemente, die in einem Typsystem eingesetzt werden können, lauten:

- Klassennamen
- Typmengen
- Strukturparametrische Typen
- Typen mit allgemeine Parametern
- Typvariablen
- Pseudotypen wie SELF
- Signaturtypen
- Spezifizierungstypen

Klassennamen als Typen bezeichnen den Typ, den eine Instanz dieser Klasse oder eines Sub-

typs davon hat, z.B. bezeichnet `<Integer>` Instanzen, die von der Klasse `Integer` oder von einer ihrer Subtypen (vielleicht `SmallInteger`) ist.

Typmengen sind Mengen von Typen, die zusammengenommen wieder als Typ einer Variablen deklariert werden können. Das ist z.B. bei den Typen von Rückgabewerten nötig, wenn ausgedrückt werden soll, daß hier entweder eine bestimmte Instanz oder `nil` zurückgegeben wird.

Beispiel: `<Integer | nil>`

Strukturparameter von Typen befriedigen das Bedürfnis, nicht nur den Typ eines Objektes anzugeben, sondern auch Aussagen über den Inhalt dieses Objektes zu machen, wie es u.a. bei Collections der Fall ist. Ohne Strukturparameter kann man von einer Collection nur sagen, daß sie eine Kollokation von Objekten ist, mit Strukturparametern läßt sich ausdrücken, daß sie beispielsweise nur Integerobjekte enthält.

Beispiel: `<Collection of: Integer>`

Häufig will man jedoch nicht nur Aussagen darüber treffen, von welchem Typ die Elemente von Collections sind, sondern man will *alle* Informationen spezifizieren, die über die innere Struktur von Objekten gemacht werden müssen um ihr Verhalten zu beschreiben. Ein schönes Beispiel für dieses Bedürfnis sind die informellen Typkommentare (siehe Anhang C.1), die man in den Klassenkommentaren von Smalltalk-80 4.1 findet. Das Problem bei diesen Parametern ist allerdings, daß man für verschiedene Objekte auch verschiedene Dinge ausdrücken will, bei Collections etwa bestimmte Längen, bei Zahlen bestimmte Intervalle usw. Da in Smalltalk neue Objekte hinzugefügt werden können, müßte es daher auch möglich sein, das Typsystem um neue Parameter zu erweitern.Unter diesem Aspekt sind Strukturparameter auch nur spezielle allgemeine Parameter, aber da sie (als einzige) in allen Typsystemen benutzt werden erscheint diese Trennung zweckmäßig.

Beispiel: `< Integer rangeBetween: 1 and: 15>`

Typvariablen benötigt man, wenn man ein Typsystem höherer Ordnung haben will. Ohne dieses Konstrukt kann man z.B. nicht ausdrücken, daß zwei Variablen vom gleichen Typ sind, da man ohne Typvariablen Typen nur über einen Namen oder eine Spezifikation ihrer Eigenschaften referenzieren kann und nach dem Übersetzungszeitpunkt ja auch durchaus noch Klassen definiert werden können. Wenn man Typvariablen einsetzt, dann muß man auch nach dem Übersetzen der Methode mit der Typvariablen diese Methode typprüfen können, da die konkrete Instantiierung mit einem Typwert ja erst in einer anderen Methode erfolgt; außerdem müssen diese Werte durch einen Unifikationsprozeß gewonnen werden.

Beispiel:

```
Typevariable: P

= aNumber<P> returns: <P>
     ^aNumber
```

Mit dem Pseudotyp SELF kann eine weitere Relation über Typen ausgedrückt werden, die nämlich, daß der Typ einer Variable der gleiche sein soll wie der des Empfängers der Methode, in der SELF vorkommt. Um SELF-Typen bei der Vererbung richtig zu behandeln, ist es notwendig, diese bei jeder Vererbung auf den richtigen Typ zu setzen, im Allgemeinen ist das der Typ des Empfängers der ererbten Methode, wie dies z.B. in [Gr89] beschrieben wird.

Bsp.: <SELF>

Signaturtypen dienen der Darstellung der Typen von Methoden. Sie werden meist durch ein Konstrukt spezifiziert, das den Namen der Methode sowie die Typen der Argumente und des Resultats enthält.

Bsp.: <= <Integer> ^ <Integer>>

Spezifizierungstypen sind Typen, die Objekte nicht durch Namen spezifizieren, sondern durch die Angabe der Eigenschaften, die Objekte dieses Typs haben müssen. Diese Eigenschaft kann z.B. eine Liste von Signaturtypen sein. Der dadurch spezifizierte Typ ist dann die Menge von Klassen, von denen jede alle diese Signaturen durch die Definition von Methoden mit dieser Signatur erfüllen kann. Beschränkt sich die Spezifierung auf die Prüfung des Enthaltenseins bestimmter Signaturen in einem Typ, dann nennt man diesen Mechanismus *Bounded Parametric Polymorphism* [CCHO89].

Bsp.: <understands: #(<= <Integer> ^ <Integer>>)>

Typsysteme lassen sich aber auch nach dem Verhältnis der Art der Vererbung zur Art der Typhierarchie unterscheiden [Gr89]:

- Typsysteme können spezifikationsbasiert sein, d.h. darauf beruhen, daß Instanzen bestimmte Spezifikationen erfüllen müssen um von einem Typ zu sein, oder sie können klassenbasiert sein, d.h. die Klassenzugehörigkeit bestimmt den Typ und der Klassenname ist der Typname.

- Klassen können Spezifikationen und Implementierungen vererben oder nur Implementierungen. Wenn eine Klasse auch seine Spezifikationen vererbt, dann muß jede Subklasse die Spezifikationen der Superklasse erfüllen und eine Instanz dieser Klasse kann daher immer anstelle einer Instanz der Superklasse benutzt werden, d.h. falls es ein Typsystem in diesem System gibt muß die Subklasse ein Subtyp der Superklasse sein.

Spezifikationsbasierte Typsysteme auf Klassensystemen, die auch Spezifikationen vererben

Hier sind Typ- und Klassenhierarchie völlig entkoppelt. Es gibt Typspezifikationen, und es gibt Klassen, die sie erfüllen. Falls allerdings eine Klasse eine solche Spezifikation erfüllt, dann erfüllt der ganze Teilbaum in der Klassenhierarchie unterhalb dieser Klasse die Spezifikation, weil Subklassen in diesem Ansatz für Superklassen einsetzbar sind.

Spezifikationsbasierte Typsysteme auf Klassensystemen, die nur Implementierungen vererben
Im Gegensatz zum obigen Ansatz erfüllt hier der Klassenbaum unterhalb einer Klasse, die die Spezifikationen erfüllt nicht automatisch auch die Spezifikationen.

Klassenbasierte Typsysteme auf Klassensystemen, die auch Spezifikationen vererben
Da Subklassen in diesen Systemen alle Eigenschaften ihrer Superklassen aufweisen, verweisen Typen in diesem Ansatz auf den Teilbaum der Typhierarchie, dessen Wurzel durch den Typnamen (der ja ein Klassennamen ist) bestimmt wird. Dieser Teilbaum wird mit jeder neuen Subklasse einer der Klassen im Teilbaum erweitert und ist daher potentiell unendlich groß. Dieser Ansatz kann auch ohne Typmengen auskommen, weil der Typ, der durch zwei Typen definiert wird, immer durch den nearest-common-supertype ersetzt werden kann. Der nearest-common-supertype ist der Supertyp der beiden Typen, der von der Wurzel des Typbaumes am weitesten entfernt ist.

Klassenbasierte Typsysteme auf Klassensystemen, die nur Implementierungen vererben
In diesen Typsystemen denotieren Typnamen die Instanzen genau der Klasse, dessen Namen der Typnamen ist und es gibt keine Verbindung zwischen Klassen- und Typhierarchie.

Spezifikationsbasierte Typsysteme lassen sich laut [Gr89] nicht zu Optimierungszwecken nutzen, da zur Übersetzungszeit von Methoden nicht klar ist, welcher Empänger eine bestimmte Methode ausführt, weil die Menge der Klassen, die eine bestimmte Spezifikation erfüllen, auch nach dem Zeitpunkt der Übersetzung dieser Methode erweiterbar ist.

Systeme, die auch eine Vererbung von Spezifikationen praktizieren, haben die unangenehme Eigenschaft, daß die Argumente von überdefinierten Methoden Supertypen der Argumente derjenigen Methode sein müssen, die sie überdefinieren [CCHO89]. Damit ist es nicht möglich, Methoden zu schreiben, deren Argumente ebenfalls Subtypen der ‘Originalmethoden’ sind, wie dies für spezialisierte Methoden bisweilen sinnvoll wäre.

Smalltalk ist eine Sprache, in der nur die Implementation vererbt wird, daher können Instanzen von Subklassen nicht immer anstelle von Instanzen von Superklassen verwendet werden. Trotzdem gibt es auch für Smalltalk Typsysteme, die sich auf die Vererbung von Spezifikationen verlassen. Da dies keine Eigenschaft des unmodifizierten Smalltalks ist, versuchen diese Ansätze denselben Effekt dadurch zu erreichen, daß Subklassen Restriktionen bezüglich der Argumenttypen oder Resultatstypen von überdefinierten Methoden auferlegt werden.

4.2 Borning und Ingalls 1982

Borning und Ingalls stellen in [BI82] eine Erweiterung von Smalltalk-80 vor, die auf der Trennung von (Typ-)Interface und Klassenimplementierung beruht. Interfaces besitzen eine unabhängige Vererbungshierarchie, in der erbende Typen das Interface additiv und manipulativ verändern können.

Die Typen können Parameter (in der Regel andere Typen) oder Typvariablen enthalten, so z.B. `<Collection of: Integer>` (Eine Collection, in der nur Integers stehen) oder

`<Collection of: elementType>` (eine Collection, in der nur Instanzen stehen, die von dem Typ sind, dessen Name die Typvariable `elementType` enthält).

Das entstehende Typsystem läßt die Typdeklaration von temporären, globalen und Klassen- sowie Instanzenvariablen zu und die von Methodenparametern und -resultaten. Temporäre Variablen können allerdings vom System auf Wunsch auch inferiert werden.

Mit der Einführung des nächsten gemeinsamen Supertyps (nearest common supertype, NCS) zweier Typen kann auf Typmengen verzichtet werden. Der nächste gemeinsame Supertyp zweier Typen ist der gemeinsame Supertyp, der in der Typhierarchie am weitesten von der Wurzel der Hierachie (`Object`) entfernt ist.

Der Zeitpunkt, zu dem Typen geprüft werden, ist der, an dem eine Methode übersetzt wird, d.h. auch der Typprüfungsvorgang ist inkrementell und führt zu keiner großen Veränderung des Zeitverhaltens.

Da auch Borning und Ingalls nicht das gesamte Smalltalk-80 System typisieren, bevor sie mit der Programmierung der Anwendungen beginnen, kann es der Fall sein, daß der Typprüfer auf ungetypten Code (also Methodensignaturen und nicht-lokale Variablen) trifft. In diesem Fall wird der Benutzer informiert und der Typ der Parameter bzw. Variablen wird als vom Typ `UndefinedObject` angenommen, der zu jedem anderen Typ kompatibel ist.

Die Autoren sehen außerdem das Problem, daß die Änderung einer Klasse dazu führen kann, daß sie nicht mehr vom selben Typ ist wie vor der Änderung, und daß sich die Änderung auch auf die Subtypen dieser Klasse auswirken kann, implementieren jedoch keinen Lösungsvorschlag.

Graver [Gr89] kritisiert in diesen Ansatz vor allem drei Dinge:

1) Wenn eine Klasse eine ererbte Methode überdefiniert, dann muß der Rückgabetyp dieser neuen Methode ein Subtyp der alten sein. Diese Restriktion ist vollkommen unüblich in Smalltalk und bedeutet einige Arbeit, falls alte Methoden getypt werden sollen.

2) Der Ansatz benutzt das Konstrukt des nearest common supertypes, um den Supertyp zweier Typen zu bestimmen. Das Problem dabei ist, daß dadurch sehr viel Information verloren gehen, da der NCS sehr weit oben in der Typhierarchie angesiedelt sein kann, was sich problematisch z.B. auf Optimierungsbemühungen auswirken kann.

3) Dadurch, daß keine NIL-Werte erkannt werden, muß der Typ von NIL, UndefinedObject, Subtyp aller anderen Typen sein. Falls man aber durch ein Typsystem gerade den Fall ausschließen will, daß keine ‘Does not understand’-Meldungen mehr auftauchen, das System also typ-safe ist, dann muß die Klasse von UndefinedObject *alle* Methoden implementieren; sicherlich keine ernsthaft aufrecht zu erhaltende Forderung.

4.3 TS 1986-1990

In [Joh86], [Gr89], [GJ90] und [JGZ88] stellen R. E. Johnson, J. O. Graver und L. W. Zurawski einen getypten Smalltalk-Dialekt, das TS-System vor.

Das Typsystem von TS ist in der Lage, die meisten großen Methoden in Smalltalk zu prüfen, stellt sicher, daß die geprüften Methoden type-safe sind (daß also die Werte von Variablen immer vom deklarierten Typ sind) und ermöglicht den Einsatz eines optimierenden Compilers auf der Basis der Information, die der Typprüfer erarbeitet.

Dazu benutzt das Typsystem Typmengen, parametrische Typen, Typvariablen, Signaturtypen sowie die Pseudotypen SELF, SELF CLASS und INSTANCETYPE. Die Parameter von Typen sind allerdings auf Strukturparameter beschränkt (`Collection of: Integer`). Die Inhalte von Typvariablen werden vom System inferiert, wobei es nicht immer eine eindeutige Lösung geben muß, aber wenn es keine Lösung gibt, dann liegt ein Typfehler vor. Die Pseudotypen wie SELF sind Makros, die zur Übersetzungszeit durch den Typ des Empfängers der Methoden ersetzt werden, in denen SELF usw. verwendet wird. Um Problemen mit vererbten Methoden aus dem Weg zu gehen, werden diese bei der Vererbung an die Subklassen ein weiteres Mal verarbeitet, so daß die SELF-Makros der ererbten Methoden auf die aktuellen Empfänger zeigen.

Um Smalltalkprogrammierern auch weiterhin exploratives Programmieren zu ermöglichen ist TS bemüht, dem Programmierer einiges von der Arbeit Typen zu deklarieren, abzunehmen, und bietet deshalb die Möglichkeit, die Typen von Methodenargumenten, temporären Variablen und Blockargumenten zu inferieren. Die Autoren berichten allerdings, daß inferierte Typen häufig zu allgemein sind, was sich mindernd auf mögliche Optimierungen auswirkt.

Aus der Erkenntnis heraus, daß in Smalltalk (im Gegensatz zu anderen objektorientierten Sprachen) Subklassen die Implementierung einer Klasse und nicht deren Spezifikation erben, wird Subtyping in diesem Ansatz nicht mit der Subklassenrelation gleichgesetzt, d.h. eine Subklasse einer Klasse ist kein Subtyp dieses Typs. Da aber Typen durchaus durch Klassen definiert werden (kein Interface-Ansatz), ergibt sich das Problem, daß Instanzen der Subklasse einer Klasse, die in einem pragmatischen Sinn durchaus sinnvoll als Subtypen dieser Klasse angesehen werden können, nicht als Werte in einer Variablen stehen können, die als vom Typ der Superklasse deklariert ist.

Beispiel:

```
| zahl<Integer> |
zahl := 5
```
Typfehler, weil SmallInteger kein Subtyp von Integer ist!

Will der Programmierer daher, daß auch Instanzen anderer Klassen als Werte für Variablen zur Verfügung stehen, dann kann er den Typ dieser Variable nicht durch Nennung eines Klassennamens definieren, sondern muß spezifizieren, wie die Struktur der Klassen aussehen muß, die als Typen dieser Variable genommen werden können. Dies ist mit Signaturtypen möglich. Signaturtypen spezifizieren eine Liste von Methodensignaturen, die ein Typ erfüllen muß:

Beispiel:

```
typeVariable: P.
| zahl<understands: (P add: P returns: P)> |
zahl := 5
Kein Typfehler, weil SmallInteger eine Methode mit der Signatur
<SmallInteger add: SmallInteger returns: Integer> besitzt!
```

Signaturtypen sind allerdings wieder eine Art Interface auf der Ebene von Methodenmengen, die z.B. nicht die Semantik der Methoden definiert, die diese Methoden haben müssen, was dann problematisch ist, wenn Methoden gleiche Namen haben, aber nicht das gleiche tun.

Aus dem getypten Programm macht der Compiler des TS-Systems nach einigen Optimierungen Code für eine Registertransfersprache, optimiert diesen Code noch einmal und erzeugt schließlich direkt Maschinencode. Die Optimierungen umfassen u.a. die Umwandlung von Nachrichtenversendungen in Case-Anweisungen, in denen direkt bestimmte Unterprogramme aufgerufen werden, die In-line Substitution von Unterprogrammaufrufen, die Eliminierung von End-Rekursionen und eine Beta-Reduktion genannte Methode, die Erzeugung von Blöcken und deren Evaluierung zu reduzieren. Der sich so ergebende Code ist 5 bis 10 mal schneller als der damalige Smalltalk-80 Compiler, obwohl die Übersetzung zwischen 15 und 30 Sekunden pro Methode benötigt [JGZ88].

4.4 Palsberg und Schwartzbach 1991

Das Ziel von [PS91] ist es, Typen in untypisiertem Smalltalkprogrammen zu inferieren, um mit dieser Information einem optimierenden Compiler die Möglichkeit zu geben, schnelleren Code zu generieren. Der Compiler filtert dazu Typimplikationen (Type Constraints) aus den Befehlen heraus und versucht, eine Lösung dieser Implikationen zu finden (ein Object>>new impliziert z.B. die Erzeugung einer Instanz der Klasse des Empfängers von new). Falls ihm das gelingt, ist das Programm mit diesem Ansatz typisierbar und die Lösung kann in im schlechtesten Falle exponentieller Zeit gefunden werden, gelingt dies nicht, kann dieses Programm nicht typisiert werden, obwohl es frei von Typfehlern sein kann. Dies soll allerdings bei den meisten gebräuchlichen Programmen nicht der Fall sein.

Typen sind hier endliche Mengen von Klassen und die Subtyp-Relation ist die Teilmengenbeziehung.

Die Beispielsprache im Text erinnert an ein vereinfachtes Smalltalk, dem Metaklassen, Blöcke und Primitive fehlen.

Anmerkung:
Kurz vor Abschluß dieser Studienarbeit stieß ich auf ein Buch von Palsberg und Schwartzbach ([PS94]). Es kam 1994 heraus und trägt den Titel "Object-Oriented Type Systems". In ihm entwickeln die Autoren aus der Kenntnis der Typsysteme von Simula, Smalltalk, C++ und Eiffel ein Typsystem für eine einfache objekt-orientierte Sprache, die sie BOPL (Basic Object

Programming Language) nennen. Anhand dieser Sprache zeigen sie die Eigenschaften von Typsystemen für objekt-orientierte Sprachen auf. Dazu beschreiben sie acht Dialekte von BOPL, die sich im Vorhandensein von Vererbung, Generizität und Typen unterscheiden und die sich zum Teil automatisch ineinander überführen lassen (so läßt sich der getypte Dialekt aus dem ungetypten durch eine Operation `infer` *erzeugen, bei der die Typen der Variablen inferiert werden). Es existiert sogar eine Workbench, die diese Operationen auf BOPL anwenden kann, und die per FTP geholt werden kann (FTP: //daimi.aau.dk/pub/oots/workbench).*

4.5 Strongtalk 1993

Bei Strongtalk [BG93] handelt es sich um einen statisch und stark getypten, kommerziellen Smalltalk-Dialekt, dessen Programme mit wenig Mühe wieder in Smalltalk-80 Programme umgewandelt werden können (allerdings unter Verlust von Funktionalität).

Das Typmodell beruht auf Interfaces, die in Strongtalk Protocols genannt werden und von denen es gewissermaßen Soll-Interfaces gibt, die separat definiert werden und Ist-Interfaces, die durch die Implementierung von Klassen formuliert sind. Diese Interfaces können kompatibel miteinander sein oder auch nicht. Die Typen von Subklassen sind ererbte Typen, die nicht unbedingt Subtypen ihrer Superklassen sein müssen [CHC90].

Der Nachteil dieser strukturellen Typisierung ist, daß Klassen zufällig ein bestimmtes Interface erfüllen können, ohne jedoch den semantischen Anforderungen an dieses Interface zu genügen. Um diese Möglichkeit auszuschließen, benutzt Strongtalk ein Konzept von Modula-3, das sich **Brands** nennt. Brands erweitern Referenzen auf Interfaces um eindeutige Stempel, die vom System vergeben werden. Referenziert ein Typ ein bestimmtes Interface, dann prüft das System, ob der Stempel des Typs und der des Interface gleich sind und verweigert sonst den Zugriff darauf.

Neben generischen Klassen gibt es Typmengen sowie die Pseudo-Typen `Self` und `Instance`, die auf den Typ der Klasse, in der diese Variablen verwendet wird bzw. auf die Instanz dieser Klasse verweist.

Als Besonderheit unterstützt Strongtalk typpolymorphe Methoden, Code also, der sich nur im Typ der Methodenparameter oder des -resultats unterscheidet.

Da es bei kommerziellen Smalltalkprodukten nicht immer wünschenswert ist, den Sourcecode zu einer Anwendung mitzuliefern, muß ein Typprüfer in der Lage sein, Code typisieren zu können, den er nicht einsehen kann. Das ist mit Strongtalk deshalb möglich, weil man die Typsignaturen von Methoden ohne Kenntnis des inneren Aufbaus dieser Methoden erstellen kann und auf dieser Basis eigenen Code typprüfen lassen kann.

5 Das Typfragment in Smalltalk-80 4.1

Wenn man in einer Initialisierungsmethode der Klasse Parser den Wert einer Variable auf `true` setzt (siehe Anhang A), dann erkennt der Parser von Smalltalk-80 ab der Version 4.0 an einigen Stellen im Quellcode Typausdrücke. Der Parser baut aus dem Quellcode einen Parsebaum auf, in dem auch die Typinformationen enthalten sind, aber der Übersetzer verwendet diese Informationen an keiner Stelle (es findet auch keine Typprüfung statt). Es gibt auch keine Stelle im ausgelieferten Virtual Image, an der Typen im Code vorkommen (die Typerkennung ist ja auch ausgeschaltet). Die einzige Stelle, an der so etwas wie Typen vorkommen ist der Kommentar der Klassen des Virtual Image (siehe Anhang C.1). Hier beschreiben Typausdrücke informal, von welcher Klasse die Objekte sein sollen, die Klassen- und Instanzenvariablen zugewiesen werden können. Außer diesen Stellen und dem Kommentar der Methode, die Typausdrücke parst, gibt es nirgends im System weitere Hinweise auf ein Typsystem, auch die Dokumentation weiß zu diesem Thema nur, daß Smalltalk eine ungetypte Sprache ist (eine schriftliche Anfrage per Email an ParcPlace zu diesem Thema wurde zwar prompt beantwortet, schwieg sich aber ansonsten zu diesem Thema aus). In den nachfolgenden Abschnitten werden die Syntax des Typfragmentes wie es sich aus dem Parser ergibt vorgestellt, einige Vermutungen über die sich hieraus ergebende Semantik gemacht und dieser vermutete Ansatz diskutiert.

5.1 Die Typsprache

Die Syntax der Typsprache läßt sich dem Kommentar der Methode `Parser>>typeExpression` entnehmen:

```
typeExpression =
 simpleType =
 (keyword simpleType)+ ( `;' (keyword simpleType)+ )* |
 ( `|' simpleType)+ |
 `->' simpleType
 ]
simpleType = `(` typeExpression `)' |
             classname [`class'] |
             literal

keyword = character* `:'
```

In dieser regulären Grammatik werden Terminalsymbole durch ‘’-Paare geklammert, []-Paare stehen für optionale Anteile. Ein simpleType besteht also aus einem Klassennamen oder einer durch ()-Paare geklammerten typeExpression. Diese wiederum bestehen entweder aus einem simpleType oder aus einem simpleType und weiteren Elementen. Die weiteren Elemente können Kombinationen aus Schlüsselwörtern und simpleTypes sein, wie sie auch als Nachrichten in Smalltalk vorkommen (wobei diese durch ein ‘;’-Trennzeichen auch noch kaskadiert sein dürfen, ebenfalls wie Selektoren in Smalltalk), oder sie können aus Aneinanderreihungen von simpleTypes bestehen, die durch ‘|’-Zeichen getrennt sind, oder sie bestehen aus einem ‘->’-

Zeichen und einem simpleType.

Wie man weiter aus den Beispielen sehen kann, werden Typausdrücke zwischen eckige Klammern geschrieben. Mit dieser Grammatik ergeben sich also zum Beispiel fogende gültige Typausdrücke:

```
<Integer>
<Integer | nil>
<Integer -> String>
<Array of: (Integer | String)>
<Array of: Integer; size: 5>
```

Nicht zulässig dagegen wären:

`<Integer -> String | UndefinedObject>`
Alle Ausdrücke müssen eindeutig geklammert werden, da der Parser keine Präzedenzrelationen kennt

`<Array of Integer>`
Schlüsselwörter müssen mit ‘:’ enden

Die Schlüsselwörter werden nirgends näher definiert, aber aus den Klassenkommentaren kann man ersehen, daß nur wenige wie `of:` und `size:` benutzt werden.

Mit der so definierten Sprache und den Möglichkeiten, an bestimmten Stellen diese Typen hinter Variablen anzugeben, kann man Methoden schreiben wie diese:

```
gauss: ende<Integer>

 "Gibt die Summe der Zahlen zwischen 1 und ende zurueck"

 | summe<Integer> |
 summe := 0.
 (1 to: ende) do: [:i<Integer> | summe := summe + i].
 ^summe
```

Wenn der Parser Typen erkennt, dann ist er nicht mehr in der Lage Primitive, die ebenfalls zwischen spitzen Klammern stehen, parsen zu können. Dieser Fehler weist ebenfalls auf die mangelnde Einbettung der Typerkennung in das Gesamtsystem hin und läßt die Vermutung zu, daß es sich hierbei um ein Feature handelt, das entweder noch nicht voll eingebaut wurde, oder aber wieder teilweise entfernt wurde.

Nach der Darstellung der Syntax von Typausdrücken stellt sich die Frage,was aber Typen in einem solchen System bedeuten können? Wie sieht die Beziehung zwischen Typhierarchie (wenn es eine gibt) und Klassenhierarchie aus? Gibt es eine Subtyprelation und wenn ja, wie ist sie definiert?

5.2 Die Semantik der Typausdrücke

Indizien für die semantische Interpretation der Typausdrücke liefern auch hier wieder die Kommentare der vordefinierten Klassen. Betrachte man diese intuitiv, so könnten beispielsweise folgende Interpretationen herauskommen:

`<Integer>`
In dieser Variable kann ein Objekt der Klasse Integer enthalten sein.

`<Integer | nil>`
In dieser Variable steht ein Integer oder aber nil.

`<Integer -> String>`
Der Inhalt dieser Variable ist eine Assoziation von Integer auf String.

`<Array of: Integer>`
Das Objekt ist ein Array, das nur Elemente vom Typ Integer enthält.

Diese intuitive Erfassung scheint auch der Hauptzweck der Verwendung von informalen Typen im Klassenkommentar zu diesen; sie dienen der uniformen Dokumentation möglicher Werte in Variablen.

5.3 Diskussion des Fragmentes

Die vorhandene Typunterstützung ist kein vollständiges Typsystem, sondern bleibt ein Fragment, solange man es nicht um die Elemente erweitert, die zu einem minimalen Typsystem erweitert. Diese Erweiterungen müssen erreichen, daß getypter Code type-safe ist und die Semantik der Konstrukte der Typsprache erklären, vor allem die der Schlüsselworte. Das Fragment kann nicht dafür sorgen, daß der Code vollständig type-safe ist, da einige Variablenarten nicht typisiert werden. Dazu gehören globale Variablen sowie Klassen- und Instanzenvariablen. Der Typ dieser Variablen könnte zwar inferiert werden (schließlich inferieren [PS91] *alle* Typen), aber die erratene Information kann dann nur noch der Performanzsteigerung einer Anwendung dienen.Die Semantik der Schlüsselworte steht darüber hinaus noch vor dem Problem, daß sie nur bestimmte Parameter zulässt. So steht z.B. hinter einem `size:`-Schlüsselwort sicher ein Literal, genauer eine Zahl und kein simpleType. Diese Restriktionen haben den Effekt, daß Schlüsselworte eigentlich nicht global eingeführt werden können (wie in Smalltalk), es müßte eine grammatische Regel für jedes einzelne Schlüsselwort geben und neue Schlüsselworte würden die Grammatik erweitern.

Das nächste Kapitel stellt daher eine Erweiterung des Typfragmentes vor, die es zu einem vollständigen Typsystem ausbaut.

6 Eine Erweiterung des Fragmentes: NTST

Weil das Typfragment von Smalltalk-80 4.1 nur wenig Funktionalität besitzt, wurde es im Laufe der Studienarbeit klar, daß es wesentlich erweitert werden mußte, um ein den anderen Ansätzen vergleichbares Typsystem zu haben. Das so entstandene Typsystem mit dem Namen NTST wird in diesem Kapitel beschrieben.

6.1 Ausdehnung auf andere Sprachmittel

Das Typfragment läßt eine Typisierung von Temporär- und Parametervariablen in Methoden und Blöcken zu. Ohne Erweiterungen müßte ein Typsystem daher die Typen der anderen Variablenarten, nämlich die von globalen und Instanzenvariablen inferieren. Weiterhin müßte es den Rückgabetyp von Methoden herausfinden. Dies ist möglich, widerspricht aber einem der Ziele der Einführung von Typen in Smalltalk: dem des Herausfindens von Fehlern zur Übersetzungzeit. In NTST muß der Programmierer daher den Typ aller Variablen deklarieren, ebenso wie den Rückgabetyp von Methoden. Dies zwingt ihn sich die Inhalte aller Variablen bewußt zu machen und genau zu recherchieren, was Methoden alles zurückliefern können. Daher mußte es in NTST möglich sein, daß der Programmierer auch für die anderen Variablen und die Rückgabewerte Typen zu definieren.

Ersteres wurde durch eine Modifikation des Klassenübersetzungsmechanismus in Smalltalk erreicht. Wenn man jetzt das Typsystem einschaltet, dann können Klassendefinitionen nun wie folgt aussehen:

```
Object subclass: #DTE
instanceVariables: ' type<Integer> speed<Float> '
classVariables: ' Channel<String> '
category: 'Communication elements'
```

Letzteres wurde ohne Modifikation dadurch erreicht, daß per Konvention die temporäre Variable `Return` in einer Methode den Typ des Rückgabewertes der Methode enthält, z.B.:

```
+ smallInteger<SmallInteger>

    | Return<SmallInteger> |
    ^self value addbin: smallInteger
```

NTST kann in Smalltalk-80 nicht von einem vollständig getypten Smalltalk ausgehen, daher muß es Mechanismen geben, die es erlauben, Methoden in einer ungetypten Umgebung zu typisieren. Aufgrund des Kapselungsprinzips von Objekten gibt es neben den Typen von Variablen nur noch eine Information, die einem Typsystem zur Typprüfung fehlt: der Typ der Parameter von ungetypten Methoden sowie der ihrer Rückgabewerte. Diese Typen bilden die Typsignatur von Methoden und müssen für Methoden deklariert werden, die in zu typenden Methoden verwendet werden. Die Angabe dieser Signatur erfolgt über ein Werkzeug von NTST, den TypeBrowser, der in einem späteren Kapitel beschrieben wird.

6.2 Das Typsystem von NTST

Die Syntax der Typsprache ist der Syntax des Typfragmentes sehr ähnlich. Geändert wurden lediglich:

- der Typ `simpleType '->' simpleType` fällt weg
- Schlüsselworte und Kaskadierungen derselben fallen weg

Der Grund für die erste Änderung ist darin zu suchen, daß er durch geeignete Schlüsselwortkonstruktionen wie etwa: `<Association from: Integer to: String>` ersetzt werden kann und es keinen Grund (etwa eine häufige Verwendung) gibt, diesen Typ speziell zu behandeln.

Der Grund für die zweite Änderung ist formaler Natur. Eine Syntax, in der Schlüsselworte 'generisch' vorkommen, kann keine Restriktionen über die Verwendung von Schlüsselworten mit bestimmten simpleTypes machen oder Aussagen darüber, welche Parameter ein Schlüsselwort bekommen kann. Die Regeln für Schlüsselworte werden daher einzeln in die Grammatik aufgenommen. Damit ergibt sich folgende Grammatik für NTST-Typen:

```
simpleType = literal |
             classname ['class']|
             '(' typeExpression ')'

typeExpression = simpleType | keywordtype | typeSet

typeSet = simpleType ( '|' ( simpleType | keywordtype ))*

keywordtype = blocktype | ...

blocktype = 'BlockClosure' 'of:' number

number=('1'|'2'|'3'|'4''5'|'6'|'7'|'8'|'9')
       ('1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'|'0')*
```

Die Semantik der komplexeren Typen lautet wie folgend:

<Integer | Float | nil >
Eine Variable dieser Typmenge kann entweder Instanzen der Klasse Integer oder der Klasse Float enthalten oder nil sein. Typmengen haben mindestens die Kardinalität 2, Typmengen mit nur einem Element degenerieren zu einem simpleType. Eine Typmenge mit zwei Typen, von denen der eine ein Subtyp des anderen ist, degeneriert zu einer Typmenge ohne den Subtyp. Die Elemente der Typmenge dürfen alle Typen außer Typmengen sein, d.h. Typmengen, die Typmengen enthalten degenerieren zu einer Typmenge, die alle simpleTypes und keywordtypes der Ursprungsmengen enthalten.

<BlockClosure of: 2>

Eine Variable dieses Typs besteht aus einem Block, der zwei Parameter nimmt.

Bevor wir darauf zu sprechen kommen, wie der Typprüfungsprozeß abläuft, wollen wir uns ansehen, welcher Typbegriff für simpleTypes in NTST überhaupt verwendet wird.

Ein Typ ist in NTST die Zusage, daß ein Objekt bestimmte Methoden kennt, von denen die Signatur bekannt ist. Diese Kenntnis leitet sich direkt aus dem Funktionsumfang ab, den eine bestimmte Klasse zum Typisierungszeitpunkt zur Verfügung stellt. Diese Klasse ist daher mit dem Typ assoziiert und der Typ trägt den Namen dieser Klasse. So besagt der Typ `<SmallInteger>` zum Beispiel, daß er nur auf Objekte referenzieren wird, die unter andere die Nachricht + verstehen, wobei der Typ des Objektes, das das Ergebnis dieser Nachricht ist, ebenfalls `<Integer>` ist.

Ein Typ ist also die Zusage über das Verhalten von Objekten. Damit diese Zusage nicht darin mündet, daß Klassen nach einer Referenz auf eines ihrer Objekte nie wieder geändert werden kann, muß ein Weg gefunden werden, diese Zusage auch nach Klassenänderungen gültig zu halten. Um einen weiteren wichtigen Aspekt von Smalltalk, nämlich die Wiederverwendung von Code durch Vererbung nicht durch die Verwendung von Typen wegfallen zu lassen, muß es darüber hinaus möglich sein, auszudrücken, daß eine Subklasse mit dem gleichen Verhalten der Superklasse (im Sinne einer “Verhaltensinklusion”) ein Subtyp dieses Typs ist.

Aus diesen Gründen basiert die Typhierarchie auf der Klassenhierarchie, obwohl in Smalltalk Implementationen vererbt werden und keine Spezifikationen. Die Typhierarchie ist aber nicht gleich der Klassenhierarchie: Eine Subklasse ist nur dann ein Subtyp einer Superklasse, wenn sie bestimmte Voraussetzungen erfüllt. Weiter gilt, daß Klassenänderungen nur dann vorgenommen werden dürfen, wenn sich dadurch der Typ einer Klasse (im Vergleich zu dem Typ der Klasse vor einer Änderung) nicht ändert. Leider wäre für diesen Test der formale Nachweis nötig, daß sich auch die Semantik von Methoden bzw. das Klassenmodell nicht ändert, eine Forderung, die durch bestehende Typsysteme nicht gelöst wird. Daher beschränken sich die Tests in NTST ebenfalls auf die automatisch durchzuführende Prüfung der Kompatibilität der Signaturen von Klassen und Methoden.

Die Regeln dafür lauten ([Gr89]): Eine Signatur ist subtyp-kompatibel zu der Signatur einer Supertypmethode, falls

- die Anzahl der benötigten Parameter gleich ist
- die Typen der Parameter der Methode des Subtyps Supertypen der Parameter sind, die die Methode des Supertyps benötigt
- der Typ des Resultats der Methode des Subtyps ein Subtyp des Resultatstyps der Methode des Supertyps ist

Eine Subklasse ist ein Subtyp einer Superklasse, falls die Subklasse auch alle Methoden der Superklasse zur Verfügung stellt, diese Methoden gleich heißen und alle Methodensignaturen der Subklasse subtyp-kompatibel zu denen der Superklasse sind. Dabei muß der Subtyp keine direkte Subklasse sein; es reicht wenn er im Vererbungsbaum irgendwo unterhalb im Teilbaum des Supertyps steht.

Ebenfalls an dieser Stelle genannt werden muß die Regel für Klassenänderungen: Eine Klassenänderung darf durchgeführt werden, wenn die geänderte Klasse ein Subtyp der ungeänderten Klasse ist und sich dadurch die Menge der Subtypbeziehungen dieser Klasse nicht ändert.

Auf der Basis der Subtypbeziehung für simpleTypes werden auch die Regeln erklärt, die definieren, welche Typen mit welchen anderen Typen kompatibel sind und was das Ergebnis der Kompatibilitätsprüfung der beiden Typen ist. Man beachte dabei, daß die Kompatibilität eine antisymmetrische Relation ist. In der folgenden Aufzählung steht B für den deklarierten Typ etwa einer Variable und A für den Typ, der gegen B geprüft werden soll, etwa der Typ eines Smalltalkausdrucks.

1) Ein **simpleType** A ist mit einem **simpleType**e B kompatibel, falls A eine Subtyp von B ist. Das Ergebnis ist dann Typ A

2) Ein **simpleType** A ist mit einer **Typmenge** B kompatibel, falls A ein Subtyp mindestens eines Typs der Menge B ist. Das Ergebnis ist Typ A

3) Ein **simpleType** A ist nie mit einem **Blocktyp** kompatibel, weil Blöcke keine simpleTypes sind

4) Eine **Typmenge** A ist nie mit einem **simpleType** B kompatibel, weil B nur von genau einem Typ sein darf

5) Eine **Typmenge** A ist mit einer **Typmenge** B kompatibel, falls einer der Typen aus A ein Subtyp mindestens einer der Typen aus B ist. Das Ergebnis ist eine Typmenge C mit den Typen aus A, die einen Supertyp in B haben

6) Eine **Typmenge** A ist nie mit einem **Blocktyp** B kompatibel, weil B genau einen Typ erwartet

7) Ein **Blocktyp** A ist nie mit einem **simpleType** B kompatibel, weil A kein simpleTyp ist.

8) Ein **Blocktyp** A ist mit einer **Typmenge** B kompatibel, falls A kompatibel mit einem der Typen aus B kompatibel ist, und das Ergebnis ist A

9) Ein **Blocktyp** A ist mit einem **Blocktyp** B kompatibel, falls die Anzahl der Parameter übereinstimmt

Kommen wir nun zum Mechanismus der Typprüfung in NTST. In einer nicht-inkrementellen Umgebung würde ein Typprüfer den gesamten Code einer Applikation prüfen und könnte am Ende sagen, ob der gesamte Code typfehlerfrei ist oder nicht. Dieses Verhalten folgt dem Programmiermodell von *einem* Zeitpunkt, an dem der gesamte Code fertig bearbeitet ist. Diesen Zeitpunkt gibt es in einem inkrementellen System nicht, eine Anwendung ensteht nach und nach aus dem Zusammenspiel einzelner Klassen. Die Einheit, um die das System erweitert

wird, ist neben den Klassendefinitionen die Methode. Eine Methode wird in Smalltalk-80 zu einem Zeitpunkt übersetzt und steht danach zur Verfügung. Diesem Modell der Programmierung in Schritten von einer Methode (bzw. einer Klassendefinition) folgt NTST durch die inkrementelle Typprüfung jeweils einer Methode. Falls ein Programmierer nach dem Schreiben des Textes einer Methode `accept` wählt und damit dem Übersetzer die Kontrolle und Übersetzung des Quellcodes dieser Methode überlässt, dann läuft nach dem Parser der Typprüfer los und testet den Code. Dazu bekommt der Typprüfer den Parsebaum des Parser als Eingabe (siehe Abbildung 1). Er kennt von allen in der Methode verwendeten Methoden die Signatur und von allen verwendeten Variablen den Typ (ansonsten beschwert sich NTST). Um die Ausgabe des Typprüfers, die Signatur der definierten Methode, zu berechnen, traversiert er den Parsebaum und löst bei jedem Knoten bestimmte Prüfungen und Aktionen aus.

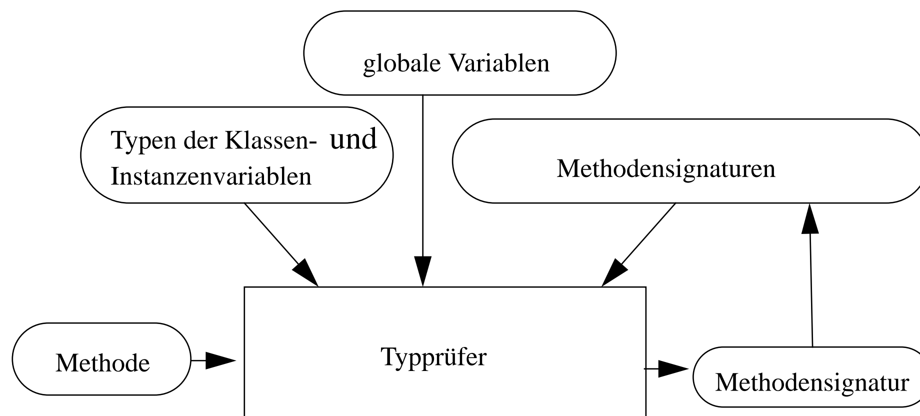

Abbildung 1

Die möglichen Knoten des Parsebaumes sind (die folgende Darstellung ist gegenüber der Realität leicht verkürzt):

- **Literale** wie `'eins'`, `1`, `1.0` oder `#Eins`
- **Variablen** wie `aString` oder `i`
- **Zuweisungen** wie `a := 5`
- **Returns** wie `^1 + 1`
- **Nachrichten** wie `aStream next`
- **Blöcke** wie `[:i | Transcript show: i; cr]`
- **Methodendefinitionen**

Trifft der Typprüfer auf einen solchen Knoten, dann errechnet er den Typ dieses Knotens und gibt ihn nach oben hin weiter. Ausserdem löst er manchmal bestimmte Aktionen aus. Es folgt daher eine Beschreibung der errechneten Typen und der ausgelösten Aktionen:

Der Typ eines Literals ist die Klasse, dessen Instanz es ist. Der Typ des Literals 5 ist z.B. `SmallInteger`, der von #Eins ist `Symbol`.

Der Typ einer Variablen wird zuerst im aktuellen Variablendictionary gesucht, in dem alle

methodenlokalen Variablen verzeichnet sind. Wird sie dort nicht gefunden, wird zuerst bei den Instanzenvariablen der Klasse der Methode gesucht, dann bei deren Klassenvariablen. Wird der Typ der Variable schließlich auch bei den globalen Variablen gefunden, wird ein Fehler ausgegeben.

Bei einer Zuweisung sucht der Typprüfer rekursiv den Typ des zuzuweisenden Wertes, dann den der Variable, der zugewiesen wird. Ist der Variablentyp ein Supertyp des Typs des Wertes fährt der Typprüfer fort, ansonsten gibt es einen Fehler.

Bei einem Rückgabebefehl wird erst der Typ des zurückgegebenen Wertes ausgerechnet und wie bei einer Zuweisung gegen den Typ des Rückgabetyps geprüft. Der Typ des Rückgabewertes wird in die Typmenge der Rückgabetypen der Methode eingefügt.

Bei Nachrichten wird erst der Typ des Empfängers der Nachricht ausgerechnet. Es wird überprüft, ob dieser Typ diese Nachricht versteht. Falls er sie versteht, wird die Signatur dieser Methode geholt und es wird geprüft, ob die Typen der Parameter Subtypen der Typen der Signaturparameter sind. Ist das der Fall, dann ist der Rückgabetyp der Signatur der Typ dieser Nachricht.

Bei Blöcken wird ein neues Dictionary mit Variablen und Typen erstellt, das den Inhalt des alten Directories hat und bei dem die eventuell neu hinzugekommenen Variablen eingetragen sind. Dieses Directory bildet dann die Umgebung für den Parsebaum, der den Block darstellt. Der Typ eines Blockes ist `BlockClosure`.

Methoden besitzen temporäre Variablen sowie eine Typsignatur (sofern sie nicht zum ersten Mal definiert werden oder sich rekursiv aufrufen). Aus den temporären Variablen wird ein Dictionary mit den Typen gemacht, in dem der Typprüfer nach dem Typ einer Variable suchen kann. Tritt unter diesen Variablen ein "Return" auf, dann wird der Typ dieser Variable zum deklarierten Rückgabetyp der Methode erklärt.

Mit dem Typsystem von NTST ist es möglich, Code zu schreiben der type-safe ist, falls erkannt werden kann, ob eine Variable nil enthält, obwohl der Typ dieser Variable nicht `UndefinedObject` ist. Ohne diese zusätzliche Forderung würde bei nicht initialisierten Variablen die Möglichkeit bestehen, daß ihnen Nachrichten geschickt würden, die einen anderen Empfänger als nil erwarten. Das hätte zur Folge, daß eine "Does not understand"-Meldung erscheinen würde, ein Effekt, den ein getyptes Smalltalk zu verhindern sucht. Mit Hilfe der nichtlokalen Datensflußanalyse ist es möglich, nicht initialisierte Variablen zu finden, aber die Kosten dafür sind aufgrund der Nichtlokalität hoch [BG93].

6.3 Diskussion des Typsystems

Das Typsystem entstand aus der Überlegung heraus einen Formalismus zu schaffen, der

1) auf dem Typfragment von Smalltalk-80 aufbaut
2) schnell realisiert werden kann
3) wesentliche Eigenschaften eines Typsystems hat und daher als Grundlage für eine

Untersuchung der möglichen Programmierunterstützung durch Typen dienen kann
4) keine Änderung bestehenden Smalltalkcodes bedingt, und der es damit erlaubt auch bestehende Klassen zu typisieren

Bis auf die letzte Forderung erfüllt NTST diese Vorgaben. Dennoch ist es kein “gutes” Typsystem in dem Sinne, daß es Vorteile gegenüber bestehenden Formalismen bietet oder gar für die Praxis besonders tauglich wäre. Der Ansatz von NTST ist auch nicht einfach zu einem konsequenten Typsystem zu erweitern. Dies liegt daran, daß die Designentscheidung die Typhierarchie auf die Klassenhierarchie zu legen, die in einem sehr frühen Stadium der Studienarbeit getroffen wurde, im Betrieb einige Probleme bereitet. Da ist zum einen der seltsame Fall, daß die Subklasse C einer Superklasse B, die wieder eine Superklasse A hat, ein Subtyp von A sein kann, ohne daß B ein Subtyp von A sein muß (siehe Abbildung 2). Diese Möglichkeit ist kontraintuitiv (jedenfalls für einen Smalltalkprogrammierer), wird aber bisweilen benötigt um bestehende Klassen zu typisieren. Das zweite Problem hängt stark mit dem ersten zusammen. Falls man die Subtyprelation einer Klasse erweitert, dann versucht NTST diese Information an alle Supertypen dieser Klasse zu transportieren. Das klappt auch, aber sobald man die Menge der Supertypen erweitert, müsste NTST eigentlich die Subtyprelationen der Supertypen aktualisieren (sonst könnte es sein, daß diese unvollständig werden), und das ist zur Zeit noch nicht der Fall..

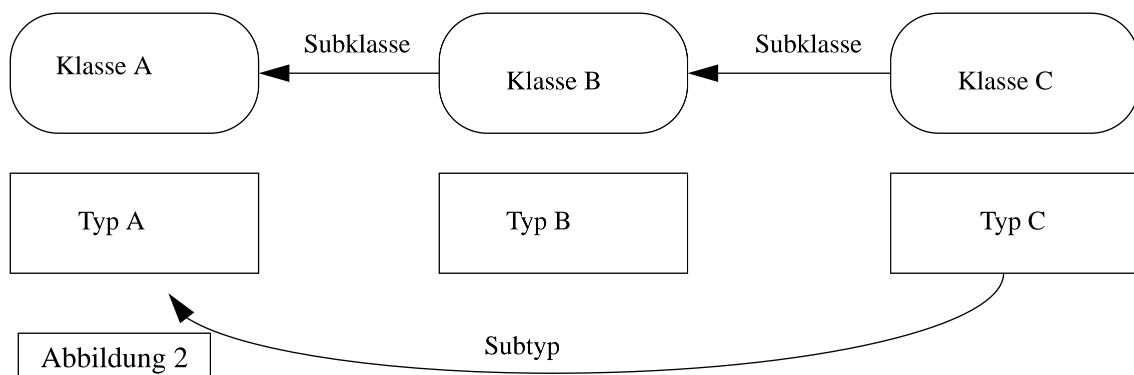

Abbildung 2

Ein weit gravierenderes Problem ist, daß Subklassen in Smalltalk nicht sehr oft auch Subtypen sind weil sie die Kriterien für die Subtypeneigenschaften (Resultatstypinklusion, umgekehrte Parametertypinklusion) nicht erfüllen. Die Forderung nach der Subtypeigenschaft für Subklassen, deren Objekte statt denen der Superklasse verwendet werden kann bei neuen Klassen vielleicht erfüllt werden, trifft aber auf die bestehende Klassenhierarchie sicher nicht zu. Das Resultat davon ist die unangenehme Eigenschaft von NTST, bestehenden Code nicht immer ohne Änderungen typisieren zu können.Ein Beispiel dafür ist folgende Situation: Eine Klasse A definiert eine Methode “+”, die als Parameter ein weiteres Objekt der Klasse A verlangt. Es gibt eine Subklasse B von A, die diese Methode ererbt. Empfängt ein Objekt des Typs B nun die Nachricht “+” mit einem weiteren Objekt vom Typ B als Parameter, so ist diese Nachricht nur dann zulässig, wenn B ein Subtyp von A ist. Das ist aufgrund der häufig nicht erfüllten Kriterien für die Subtypeneigenschaft nicht immer gegeben. Eine Lösung dieses Problems ist eine Technik, die immer dann eingesetzt werden muß wenn das Typsystem keine adäquate Ausdrucksmöglichkeit bietet. Dazu kopiert man die aufzurufende Methode in die Klasse des

Empfängers dieser Methode im Code. Man ersetzt den Methodenkörper durch eine Anweisung der Art `^super (Methodename und Parameter)`, wodurch diese Methode die “richtige” Methode aufruft. Nun kann man allerdings die Typen der Parameter (oder andere kritische Stellen auf den benötigten Typ setzen, in unserem Beispiel wäre das dann

```
+ aNumber<B>

      ^super + aNumber
```

Diese Methode ist zwar unschön, aber sie funktioniert und man könnte diese Ersetzung automatisch vom System vornehmen lassen.

Ein weiteres Manko ist das Fehlen der SELF bzw. SUPER Pseudotypen. Diese werden bei jeder Vererbung oder Definition einer Methode auf den Empfängertyp dieser Methode gesetzt. Dadurch kann man die Typgleichheit von Variablen ausdrücken, was oft eine erhebliche Vereinfachung beim Typisierungsprozess darstellt. Allerdings kann dieses Problem ebenfalls durch das Herunterkopieren der Methoden lösen (in diesem Fall simuliert der Programmierer sogar das Vorgehen des Typsystems, wenn es auf SELF usw. trifft).

Die beschriebenen Probleme des Typsystems beeinflussen allerdings ihren Wert für die weitere Untersuchung nicht.

6.4 Implementierung

Betrachten wir nun, wie NTST implementiert wurde. Die Basis des Typsystems, die Typen werden an verschiedenen Stellen und zu verschiedenen Zeiten gespeichert. Die Typen, die der Parser erkennen kann, nämlich die der Temporär- und Parametertypen werden nur zum Typprüfungsprozeß benötigt, der in Aktion tritt, wenn eine Methode geändert wurde oder wenn ein Empfängertyp einer Nachricht gesucht wird. Das Gleiche gilt für den Rückgabetyp einer Methode. Das hat zur Folge, daß diese Typen nur dann im Quelltext präsent sein müssen, wenn die Methode geprüft wird. Die Speicherung im Quelltext übernimmt das Smalltalksystem, so daß hier keine Maßnahmen getroffen werden mussten. Die zweite Gruppe von Typen sind die der globalen Variablen. Hier musste die Klasse `SystemDictionary`, die die Zuordnung von globalen Variablen zu deren Inhalten mit der Instanz `Smalltalk` erledigt, um eine Instanzenvariable erweitert werden, die ein zweites Dictionary enthält. In diesem Dictionary sind jedem Variablennamen die entsprechenden Typen zugeordnet. Die letzte Gruppe von Variablen sind die Klassen- und Instanzenvariablen. Diese werden in Smalltalk bei der Klasse gespeichert, in der sie definiert sind. Daher ist es vernünftig auch dort die Typinformation dieser Variablen in Form von zwei neuen Instanzenvariablen der Klasse `ClassDescription` abgelegt sind.

Aber von welcher Klasse sind Typen überhaupt? Dazu muß man wissen, daß die Typinformation im Parsebaum in Form von Objekten der Klasse `VariableNode` gespeichert sind. Es lag also nahe, überall dort, wo Typinformation gebraucht wurde, VariableNodes zu benutzen. VariableNodes bestehen eigentlich nur aus ihrem Namen, einem String. Dieser ist mit einem Klassennamen von Smalltalk identisch. Die zwei komplexeren Typenarten, Blocktypen und

Typmengen sind Subklassen von VariableNode. Sie heißen `TypeBlock` und `TypeSet`. Die erstere Klasse hat eine Instanzenvariable, `number`, in der die Anzahl der Parametervariablen in diesem Block gespeichert ist. TypeSet hingegen enthält (u. U. rekursive) Arrays der Größe zwei, in denen andere Typen enthalten sind. Typsysteme, die Typmengen verarbeiten können, haben normalerweise keine Unterscheidung zwischen Typmengen und Einzeltypen, in NTST ist diese Trennung historisch bedingt und sollte bei einem Redesign entfallen. NTST kann einfach um weitere Schlüsselworttytpen erweitert werden, in dem die Klasse, die den "Empfänger" in einem Typ spielt, eine Methode bereitstellt, die genauso heißt, wie das Schlüsselwort. Das Ergebnis dieser Methode ist ein Objekt einer Subklasse von VariableNode (das kann auch eine neue Subklasse sein), die die Nachricht `match2: aVariableNode` versteht. Diese Nachricht gibt ein Objekt der Klasse VariableNode zurück, falls der Test auf Kompatibilität (um einen solchen handelt es sich bei match2:) erfolgreich war und `nil` sonst.

Während Typinformationen bei den Variablen gespeichert sind, werden die Menge der Subtypen einer Klasse in einer weiteren Instanzenvariable von `ClassDescription` gespeichert. Die Signatur von Methoden steht in einer Instanzenvariable, die die Definition der Klasse `CompiledMethod` erweitert. Dort wird auch vermerkt, wenn die entsprechende Methode erfolgreich typgeprüft wurde, die Methode im NTST-Sprachgebrauch also 'getypt' ist. Das Gleiche gilt für Klassen, diese gelten als getypt, wenn alle Klassen- und Instanzenvariablen typisiert sind. Die Information, ob eine Klasse oder Methode getypt ist, wird für einige Zwecke wie das Abspeichern der Typinformation bei einem speziellen FileOut verwendet.

Die Information, ob sich NTST gerade in einem Zustand befindet, in dem neue Methoden oder Klassen überhaupt typgeprüft werden, wird durch den Wert der Klassenvariable `IsTypingOn` in der Klasse `Type` angegeben. Dieser Wert kann durch einen Knopf im NTST-Browser gesetzt werden (siehe entsprechendes Kapitel).

Der Typprüfungsprozeß ist in die Methode `compile: in: notifying: ifFail:` der Klasse `SmalltalkCompiler` "implantiert". Hier wird der zuvor aufgebaute Parsebaum an die Methode `testMethod:` der Klasse `Type` übergeben. Diese stößt die Typprüfung an. Dazu existiert eine zentrale "Verteilermethode" `selector: environment: indent: returnType:`, die einen Teilbaum eines Parsebaumes bekommt und ihn der Methode übergibt, die für die Art des Wurzelknotens zuständig ist. Diese Methoden nehmen den Teilbaum und arbeiten die Kinder der Wurzel rekursiv ab. Diese Abarbeitung geht dabei immer über die Verteilermethode. Neben den Ästen werden den Methoden auch die für diesen Teilbaum gültigen lokalen Variablen und deren Typen in Form eines Dictionarys übergeben sowie der erwartete Rückgabetyp. Welche Knotenarten gibt es in einem Parsebaum und was tun die Methoden, die für diese Arten zuständig sind? An Elementen eines Parsebaums gibt es:

- Literale
- VariableNodes
- AssignmentNodes
- ReturnNodes
- MessageNodes
- CascadeNodes
- SequenceNodes

- BlockNodes

Die Methoden, die diese Nodes bearbeiten werden im Folgenden beschrieben. Dabei gilt für alle, daß sie den Typ, den ihr Node repräsentiert, errechnen und zurückgeben. Falls in `Type>>debugging` der Rückgabewert auf true gesetzt ist, kann der Aufruf der Methoden und ihre Rückgabewerte im Transcript beobachtet werden.

Literale
Literale sind Objekte, die direkt aus ihrer textuellen Repräsentation im Quelltext heraus erzeugt werden. Hierzu zählen alle Zahlen wie 15, 3.3, 2/3, -77, Strings wie `'hello, world'`, Symbole wie `#do:` und Arrays wie `#( 12 'hallo' #( 1 2 ) )`. Die dafür zuständige Methode `Type>>literalNode:..` nimmt einfach die Klasse des Literalobjektes und erzeugt aus dem Namen der Klasse einen VariableNode. Der Typ eines Literals ist der Typ seiner Klasse.

VariableNodes
VariableNodes repräsentieren das Auftreten von Variablen. Der Typ einer Variablen ist dabei einfach der Typ, mit dem diese Variable deklariert wurde (NTST verlangt ja die Deklaration des Typs von jeder Variable, die in einer zu typisierenden Methode auftritt). Dazu wird nachgesehen, *wo* diese Variable definiert bzw. deklariert wurde. Dazu wird erst das Dictionary durchsucht, den der Typprüfer für temporäre und Parametervariablen anlegt, dann die Instanzenvariablen der Klasse, in der die zu typisierende Methode steht, danach deren Klassenvariablen. Wird auch dort nichts gefunden, sieht die Methode nach, ob der VariableNode `self` oder `super` ist (in diesem Fall ist der Typ der der Empfängerklasse der Methode bzw. deren Superklasse). Wenn dann die Variable auch in den globalen Variablen nicht gefunden wird, erscheint eine Fehlermeldung.

AssignmentNodes
Dieser Knoten repräsentiert Zuweisungen wie `a := 1 + 1`. Er hat zwei Unterbäume: Den der Variablen auf der linken Seite und den des Ausdrucks auf der rechten Seite der Zuweisung. Der “linke” Unterbaum enthält einen VariableNode und der rechte einen Knoten, der den Ausdruck repräsentiert, etwa einen MessageNode oder einen LiteralNode. Von jedem dieser Unterbäume wird der Typ dadurch ermittelt, daß er der Verteilermethode `select:..` geschickt wird (die ihn entsprechend weiterleitet). Der Rückgabewert ist dann der Typ des Unterbaumes. Falls der Typ des Ausdrucks kompatibel zu dem der Variable ist, ist die Zuweisung typkorrekt, sonst gibt es eine entsprechende Fehlermeldung.

ReturnNodes
Ein ReturnNode repräsentiert eine Rückgabeanweisung in einer Methode. Im Prinzip gilt hier dasselbe wie für Zuweisungen (eine Rückgabe ist eine Zuweisung an eine spezielle Variable mit dem Seiteneffekt des Rücksprungs aus der Methode). Dabei wird der Typ des Rückgabeausdrucks gegen den Rückgabetyp der Methode geprüft. Falls dieser Test erfolgreich war, wird der Typ des Ausdrucks dem Array hinzugefügt, das den tatsächlich möglichen Rückgabetyp der Methode enthält.

MessageNodes

MessageNodes repräsentieren Nachrichten wie z.B. `Transcript show: 'hallo'`. Sie besitzen einen Empfänger, einen Selektor und eventuelle Argumente. In unserem Fall ist die Variable Transcript der Empfänger, der Selektor heißt 'show:' und das Argument ist ein String.Die Empfänger einer Nachricht und die Parameter können wieder MessageNodes sein, so wird z.B. `1 + 1 * 2` als MessageNode geparst, in dem der Empfänger der MessageNode ist, der `1 + 1` darstellt (in Smalltalk gelten keine Operatorpräzedenzregeln, und alle Nachrichten, die die gleiche Anzahl von Argumenten benötigen, werden von links nach rechts abgearbeitet).Die Typen des Empfängers und der Parameter werden durch den Aufruf von `Type>>select:..` mit den entsprechenden Ästen als Parameter errechnet. Dann wird die Methode gesucht, die durch den Selektor bestimmt wird. Dazu wird in der Klasse, die durch den Typ des Empfängers dargestellt wird, nach dieser Methode gesucht. Falls sie dort nicht gefunden wird, wird sie die Vererbungshierarchie nach oben nach dieser Methode durchsucht. Wird die Methode nicht gefunden erscheint eine Fehlermeldung. Falls die Methode gefunden wurde, wird die Signatur dieser Methode (die ja dort gespeichert ist) ermittelt. Die Parameter des MessageNodes werden gegen die Typen der Parameter in der Signatur geprüft (erstere müssen kompatibel mit letzteren sein). Falls dieser Test nicht erfolgreich war, erscheint ebenfalls eine Fehlermeldung (das ist oft dann der Fall, falls diese Methode keine eigene Signatur aufweisen kann, d.h. wenn diese noch nicht gesetzt wurde). Falls der Empfänger in einem MessageNode `nil` ist, erscheint eine Fehlermeldung. Dies ist vor allem dann der Fall, wenn der Empfänger wieder ein MessageNode ist und die Signatur der dort aufgerufenen Methode `nil` zurückgibt. Falls der Selektor eines MessageNodes rekursiv auf die Methode verweist, in der der MessageNode verwendet wird, erkennt das der Mechanismus und gibt eine Warnmeldung aus, die besagt, daß man die Methode, die eben typgeprüft wird, anschließend noch einmal typprüfen lassen soll. Dies ist deshalb notwendig, weil die Signatur dieser Methode im ersten Anlauf noch nicht feststehen kann. Der Typ eines MessageNodes ist der Rückgabetyp in der Signatur der Methode, die durch den Selektor (und den Empfänger) repräsentiert wird.

CascadeNodes

Kaskaden sind Sequenzen von Nachrichten, die einem Empfängerobjekt geschickt werden und durch Semikolon getrennt werden, wie z.B. `Transcript show: 'hello'; show: ', world'; cr`. Diese bestehen in einem CascadeNode durch noch aus einem Array von MessageNodes, die einzeln an `Type>>select:..` geschickt werden. Der Typ eines CascadeNodes ist der des letzten auftretenden MessageNodes.

SequenceNodes

Ein SequenceNode steht für eine Sequenz von verschiedenen Anweisungen, wie sie in Methoden oder Blöcken stehen können. Diese Sequenzen können von der Deklaration von temporären Variablen angeführt werden. So würde z.B.

```
| a<Integer> b<Float> |
a := 5.
b := 3.0 + 1.0.
```

eine Sequenz mit zwei Anweisungen und zwei Deklarationen von temporären Variablen sein. Die Variablendeklarationen gelten genau für die Anweisungen der Sequenz, daher wird bei

einem SequenceNode ein neues Dictionary erzeugt, in das die Variablen des alten Dictionarys kopiert werden. Anschließend werden die Variablen-Typ-Zuordnungen der Variablendeklaration hineinkopiert. Dieses neue Dictionary wird an `Type>>select:..` beim Aufruf mit jeder einzelnen Anweisung weitergegeben.

BlockNodes

BlockNodes repräsentieren Blöcke oder eine Methode. Sie besitzen eventuell Parametervariablen. Diese Parameter sind für den Block oder die Methode wie temporäre Variablen zu behandeln, daher wird ein neues Dictionary angelegt, in die die alten lokalen Variablen kopiert werden. In dieses Dictionary werden nun die Parametervariablen eingetragen (wobei diese auch gleichnamige Variable überdecken können). Dieses Dictionary wird als Argument für den "Body" des BlockNodes mit an `Type>>select:..` übergeben. Der Body enthält die Anweisungen des Blocks oder der Methode.Der Typ eines Blockes ist `<BlockClosure of: n>`, wobei n für die Anzahl der Parameter steht.

Die Wurzel eines Parsebaums ist dabei immer ein MethodNode, der einen BlockNode enthält, daher wird der Verteilermethode am Anfang ein Blocknode übergeben.

Wenn man die Implementierung kritisieren will, dann muß man zuerst darauf verweisen, das NTST ohne eine NIL-Erkennung nicht type-safe ist. Dies ist aber wie schon angesprochen kein struktureller Mangel, und selbst ein kommerzielles System wie Strongtalk implementiert sie nicht [BG93]. Die vorliegende Implementierung kann keine Poolvariablen und keine Instanzenvariablen von Metaklassen typisieren, aber diese werden auch nur sehr selten verwendet und man könnte eine Typunterstützung dieser Variablen relativ schnell implementieren. Der hinderlichste Fehler der Implementierung ist der Mangel an Schlüsselworten, hier wäre z.B. ein `of:` angebracht, das in Collections eine Struktur ausdrücken könnte, in der nur Objekte eines einzigen Typs gespeichert sind, wie etwa `<Array of: Integer>`, also ein Array, das nur Integers speichert (analog zur Arraydeklaration in vielen Programmiersprachen). Wenn man NTST neu implementieren würde, würde man sicher viel weniger die existierenden Klassen wie `Parser` und `ClassDescription` modifizieren, sondern durch eine saubere Verlagerung der Änderung in neue Subklassen sicherstellen, daß bei einem Programmierfehler das gesamte System unbrauchbar werden könnte. Man würde die an sich unnütze Klasse `Type`, von der es keine Instanzen gibt auflösen und deren Funktionalität sinnvoller unterbringen und die Methoden vor allem in `Type` würden durch eine regelgerechte Problemanalyse nicht so groß werden. Ebenfalls nützlich wäre die Einführung eines Zeitpunktes, an dem die Typinformationen wieder auf einen homogenen Stand gebracht werden können. Dazu zählt die erneute Typprüfung der Methoden, falls diese durch eine Änderung der Typdefinitionen ungültig gemacht werden würde und ein vollständiges Durchpropagieren der Subtypenrelationen. Das könnte durch eine Methode `synchronizeNTST` verwirklicht werden, nach deren Aufruf NTST die Gültigkeit aller Typinformationen erklären könnte.

7 Werkzeuge zur Unterstützung der Typisierung

Für die Akzeptanz eines Typsystems in Smalltalk ist nicht nur die Mächtigkeit der Typsprache wichtig, sondern vor allem auch die Möglichkeiten, die ein Programmierer hat, schnell und ohne unnötige Mühe Variablen, Methoden und Klassen typisieren zu können. Die dazu notwendigen Werkzeuge müssen in das Bild passen, das die Programmierumgebung Smalltalk-80 im Programmierer erzeugt. Daher muß ein Typwerkzeug grafisch orientiert sein, es muß eventuell mit den anderen Werkzeugen interagieren können und es muß die gleichen Aktionen anbieten, die vergleichbare Aufgaben in anderen Werkzeugen erfordern (im Sinne einer Orthogonalität der Teilaufgaben).

Da der Typbegriff eng an Variablen und an die Klassenhierarchie geknüpft ist, ist es sicher eine gute Idee, das Werkzeug zu erweitern, das diese Programmierelemente darstellbar und zugreifbar macht: den Browser. Der nächste Abschnitt beschreibt daher dieses Werkzeug.Im übernächsten wird dann ein kleines Tool vorgestellt, mit dem man globale Variablen typisieren kann.

7.1 Der TypeClassBrowser

Der TypeBrowser (siehe Abbildung 3) ist eine Erweiterung des globalen Smalltalkbrowsers des Systems. Er wurde um zwei kleine Fenster erweitert, von denen das obere einen Knopf enthält, mit dem man dem Typsystem anzeigen kann, das von nun an zu schreibende Klassen bzw. Methoden typgeprüft werden sollen. Es gibt keine Restriktionen bezüglich seiner Verwendung, man kann ihn jederzeit an- und ausschalten.

Das untere der beiden neuen Fenster enthält für gewöhnlich die Typinformation zu dem Systemobjekt, das gerade im Browser selektiert ist, das ist entweder eine Klassenkategorie, eine Klasse, ein Protokoll, eine Methode oder keines der vier. Falls keines dieser Elemente ausgewählt ist, zeigt dieses Fenster (von nun an das Typfenster genannt) nichts an. Wählt man jedoch eine Klassenkategorie aus, dann bekommt man gemäß der Browsertradition im Typfenster ein “Formular” für die Typinformation einer Klasse. Diese Information besteht aus der Subtyprelation zu dieser Klasse (“alle Subtypen der Klasse werden aufgelistet”) und der Information, ob die Klassen- und Instanzenvariablen dieser Klasse typisiert sind, die Klasse im NTST-Sprachgebrauch also getypt ist. Wählt man nun eine Klasse aus, dann wird dieses Formular durch die Anzeige der tatsächlichen Typinformationen (Subtypenmenge und “Getyptsein der Klasse”) ersetzt. Es steht jetzt aber nicht alle Typinformation im Typfenster. Wie bei den Variablen, die innerhalb einer Methode deklariert sind, steht der Typ der Klassen- und Instanzenvariablen hinter den entsprechenden Variablennamen. Jetzt können Typinformationen auch zum ersten Mal geändert werden: Ein Typ einer Klassen- bzw. Instanzenvariable wird einfach in spitzen Klammern hinter den Variablennamen gesetzt. Falls nun der Typschalter an ist und man alle auftretenden Variablen typisiert hat, kann man Accept aus dem mittleren Mausmenü auswählen. Jetzt sind die Typen dieser Variablen gesetzt. Falls bei diesem Vorgang kein Fehler aufgetreten ist, kann man sehen, daß der Status der Klasse auf “getypt

gesetzt wird.

Abbildung 3

Die Subtypenrelation zu ändern ist etwas umständlicher. Zuerst muß nämlich die Zeile mit der Information aus dem Typfenster gelöscht werden, in der drin steht, ob die Klasse getypt ist. Dann kann man das Array mit den Subtypen beliebig kürzen oder erweitern. Man muß allerdings beachten, daß die Typen als String ausgelegt sind und als umschließendes Paar die spitzen Klammern verlangen (das Kennzeichen für Typen in NTST). Auch hier führt der Weg zum Erfolg über die Selektion von Accept aus dem mittleren Mausmenü.

Auch nach der Selektion eines Protokolls (also einer Methodenkategorie) bekommt man ein Formular zu sehen, diesmal ist es das Typsignaturformular für eine Methode. Selektiert man nun eine Methode wird das Formular wiederum durch die tatsächliche Typsignatur der Methode ausgetauscht. Außerdem steht auch hier wieder die Information, ob diese Methode bereits einmal erfolgreich eine Typprüfung durchlaufen hatte, die Methode also getypt ist. Auch hier führt der Weg zur Manipulation der Signatur durch den Programmierer über das Löschen der “ist-getypt”-Information und das Einsetzen der gewünschten Typen zur Beendigung dieser Aktion durch ein Accept. Dabei steht das Array mit den Strings, die der Typsprache gemäß aufgebaut ist für die Reihenfolge der Parametertypen und ein einzelner String mit diesem Aufbau für den Resultatstyp dieser Methode. Man muß dabei allerdings beachten, daß die eingegebene Signatur nicht vom System überprüft werden kann, wenn die Methode ungetypt ist (und selbst bei diesen prüft das System zur Zeit nicht auf Signaturkompatibilität nach).

Nun muß über die Wege berichtet werden, eine Methode zu typen, einem der komplexeren Mechanismen des TypeBrowsers.

Durch das Einschalten der erweiterten Syntax im Parser ist Smalltalk-80 4.1 in der Lage, Typinformationen bei Temporär- und Parametervariablen zuzulassen. Diese werden in spitzen Klammern dann hinter die Variablennamen geschrieben, wenn diese deklarierend genannt werden. Bei den Temporärvariablen ist das zwischen den senkrechten Strichen der Fall und bei den Parametervariablen im Methodenkopf. Die Syntax der Typausdrücke folgt dabei der Typsyntax von Smalltalk-80 4.1. Um diesen Mechanismus auch in NTST zu verwenden, muß man sich bei der Typsyntax auf eine Teilmenge der in Smalltalk-80 zulässigen beschränken, die oben beschrieben wurde. In NTST ist es außerdem Konvention, daß die Rückgabetypen von Methoden durch den Typ der Temporärvariablen `Return` bestimmt werden, so daß diese Variable besser nicht für eigene Zwecke benutzt wird. Wenn man eine bestehende Methode typisieren will, dann muß man alle auftretenden Temporär- und Parametervariablen sowie die Returnvariable typisieren. Ist das geschehen, dann wählt man wieder Accept aus dem mittleren Mausmenü. Nun beginnt erst der Smalltalk-80 Parser seine Arbeit. Er bringt die gewohnten Meldungen und wird sich zudem über die unnütze Variable Return beschweren, wobei wir diese Warnung ignorieren. Nach einem erfolgreichen Parserlauf läuft der Typprüfer los.

Tritt der (seltene) Fall ein, daß er nichts zu beanstanden hatte, dann überprüft er ob die Signatur, die er aus der Methode errechnet hat mit derjenigen übereinstimmt, die im Typfenster angegeben ist. Falls dies nicht der Fall ist, fragt er, ob man die neue Signatur eintragen lassen will. Ist das der Fall, dann trägt er die neue Signatur ein und setzt den Status der Methode auf "getypt", was sich im neuen Text des Typfensters widerspiegelt.

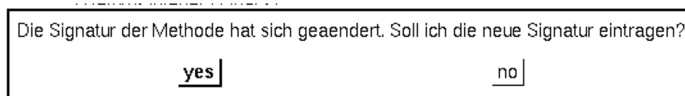

Abbildung 4

Verwendet eine Methode sich rekursiv selbst, dann bemerkt das der Typprüfer und bringt eine entsprechende Warnung. Diese besagt, daß die Signatur dieser Methode nicht typgeprüft werden kann, und daß man daher diese Methode nach dem ersten Durchlauf ein zweites Mal typprüfen lassen soll (das ist notwendig, weil der Typprüfer in einem solchen Fall die Prüfung auf Signaturverträglichkeit ausschaltet.

Meist wird jedoch der Typprüfer auf ein Problem laufen (das nicht immer ein Typfehler sein muß!). Dazu kennt er sechs Fehlermeldungen, die man dann in Kenntnis der Typhierarchie interpretieren muß:

1) Dieser Typ kennt diese Methode nicht
2) Die Syntax eines Typs stimmt nicht
3) Einem UndefinedObject wird eine Nachricht geschickt
4) die Parameter einer Nachricht stimmen nicht mit der Signatur der entsprechenden Methode überein
5) Der Typ des Wertes einer Zuweisung stimmt nicht mit dem Typ der Variablen überein, der zugewiesen wird
6) Der Typ des Rückgabewertes stimmt nicht mit dem Rückgabetyp der Methode

überein

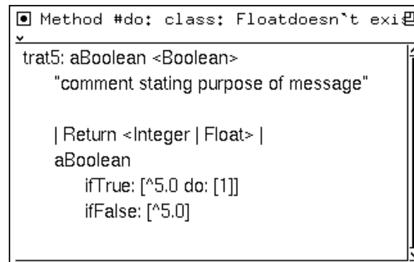

Abbildung 5

Die erste Fehlermeldung besagt, daß ein Typ eine Methode nicht kennt (siehe Abbildung 5). Dafür kann es zwei Gründe geben:

1) Es liegt wirklich ein Typfehler vor, der Programmierer hat dann vielleicht einen anderen Typ gemeint
2) Die Methode ist in einer Superklasse definiert, aber die Klasse des Typs ist kein Subtyp dieses Supertyps.

Im zweiten Fall kann es entweder sein, daß die Typhierarchie durchaus recht hat und der Programmierer diese Methode nicht aufrufen darf (weil sich vielleicht die Semantik der Methode geändert hat), oder aber (und das ist im augenblicklichen Zustand von NTST wahrscheinlicher), daß zwischen Sub- und Superklasse durchaus ein Subtypbeziehung herrscht, daß diese aber nicht bei der Superklasse vermerkt ist. Im letzteren Fall muß die Menge der Subtypen bei der entsprechenden Superklasse wie oben beschrieben erweitert werden.

Die zweite Fehlermeldung ist einfach zu interpretieren: Es liegt ein Verstoß gegen die Typsyntax von NTST vor. Entweder entspricht dabei einen Typnamen keine gleichnamige Klasse oder es wurde falsch geklammert oder ein unerlaubtes Schlüsselwort verwendet.

Falls einen `UndefinedObject` eine Nachricht geschickt wurde, liegt dies meist daran, daß die Methode, die `nil` als Resultatstyp hatte, nicht getypt ist. Dann muß deren Signatur gesetzt werden.

Falls die Parametertypen einer Methodensignatur nicht mit den tatsächlich übergebenen Typen übereinstimmen, kann auch wieder entweder ein echter Typfehler vorliegen oder aber die Signatur dieser Methode falsch sein. Letzteres kann dadurch geschehen sein, daß die Signatur dieser Methode noch auf den Defaulteinstellungen (keine Parameter) gesetzt sind, oder aber irrtümlich auf die falschen Typen gesetzt wurden. Im letzteren Fall ist darauf zu achten, daß dann diese Signatur bestimmt schon einmal zur Typprüfung eingesetzt worden ist, und eine Typprüfung für eine andere Methode vom Typsystem unbemerkt ungültig werden kann wenn man diese Signatur ändert

Liegt ein Zuweisungsfehler vor (siehe Abbildung 6), dann ist der Typ des zugewiesenen Wertes kein Subtyp des Typs der Variablen, der zugewiesen wird. Dies kann ebenfalls wieder daran liegen, daß ein echter Typfehler vorliegt, oder daran, daß die Typhierarchie noch nicht der Realität entsprechend nachgebildet ist.

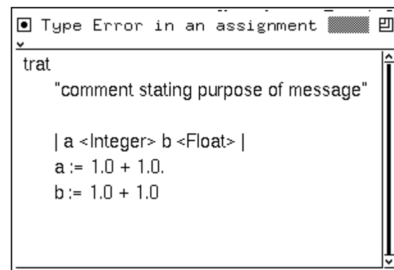

Abbildung 6

Eine spezielle Abart des Zuweisungsfehlers ist die fehlende Kompatibilität des Typs des Rückgabewertes zum Rückgabetyp der Methode und es gilt das Gesagte aus dem letzten Abschnitt.

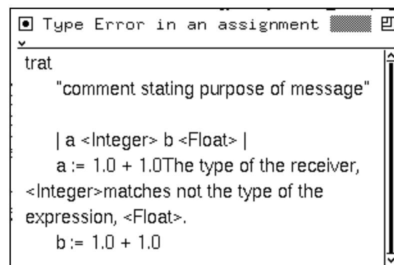

Abbildung 7

Alle diese Fehler führen zu einem Abbruch des Accept. Sie müssen in einer Änderung der Methode oder einer Änderung in der Typinformation von NTST, das sind Subtypenrelationen und Methodensignaturen enden. Im letzteren Fall muß die Methodendefinition aufgegeben und zu einer anderen Klasse oder Methode gewechselt werden (es gibt derzeit ja nur eine Instanz des TypeBrowsers). Um nicht alle Änderungen zu verlieren, ist es bisweilen nützlich, den Typschalter auszuschalten und die Methode regulär (also ohne Typprüfung) zu übersetzen. Da auch der normale Parser die Typen im Quelltext behält, geht dann nichts verloren. Fast alle Fehlermeldungen (wie etwa die des Zuweisungsfehlers, siehe Abbildung 7) besitzen ein Menü, das am mittleren Mausknopf hängt und in dem es einen Punkt "Explain why" gibt. Wird dieser ausgewählt, dann spezifiziert das Typsystem den aufgetretenen Fehler näher. Diese Indirektionsstufe von Fehlerinformationen wurde gewählt um Meldungen nicht durch eine Flut von Daten unlesbar zu machen.

Außer den Typen von Temporär-, Parameter- und Resultatswerten ist die Typinformation von Klassen und Methoden noch nicht so in das Smalltalksystem eingebunden, wie dies für ein nützliches Typsystem wünschenswert wäre. Damit diese Information jedoch auch außerhalb von Images erhalten werden kann, besteht die Möglichkeit Typinformationen separat in eine Datei zu schreiben. Der Mechanismus arbeitet dabei analog zum "File-Out" von Smalltalkcode: Man wählt den entsprechenden Menüeintrag (File out type information as...) aus dem Methoden-, Protokoll-, Klassen- oder Kategorienfenster aus, kann dann einen Dateinamen wählen, wobei ein Default vorgegeben wird, und dann wird die Datei geschrieben. Typinformationsdateien sollen die Endung .stt (für Smalltalk Types) haben. Um die gesamten Typinformationen, die im ganzen Smalltalksystem vorliegen in eine Datei schreiben zu können,

benutzt man den Eintrag “File out ALL type information as..” aus dem mittleren Mausmenü des Kategorienfensters. Das Laden der Informationen geschieht wie gewohnt durch ein “File In” etwa im Werkzeug FileList.

Eine nützliche Eigenschaft des GlobalsBrowsers wäre die Möglichkeit, auf Wunsch den Typ von Temporärvariablen vorzuschlagen. Dies wäre technisch relativ einfach machbar (Inferenz dieser Typen bei Zuweisungen), und würde dem Verhalten des Browsers entsprechen, nicht deklarierte Temporärvariablen auf Wunsch zu deklarieren.

7.2 Der GlobalsBrowser

Es gibt in Smalltalk-80 kein grafisches Werkzeug, um globale Variablen, die selber einem SystemDictionary gehalten werden, das an die globale Variable Smalltalk gebunden ist, anzusehen und zu verändern. Um dennoch die Typen von globalen Variablen bestimmen und manipulieren zu können wurde ein kleines interaktives Tool, der GlobalsBrowser zu diesem Zweck programmiert (siehe Abbildung 8).

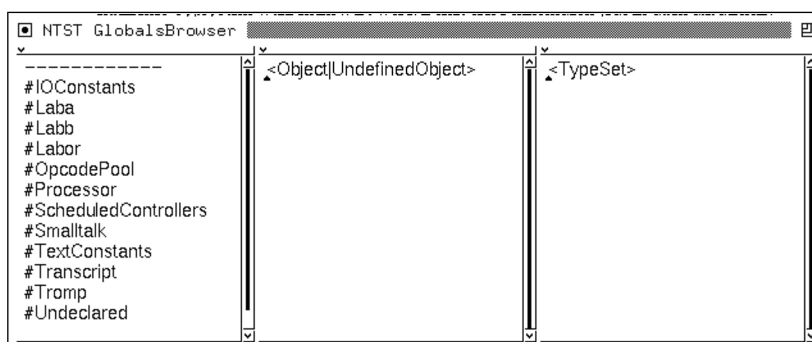

Abbildung 8

Er zeigt auf der linken Seite all diejenigen globalen Variablen aus Smalltalk an, die keine Klassen sind (und die daher durch dem Browser erreichbar wären). In der rechten Seite steht zu jeder der Variablen der deklarierte Typ, den man mittels Accept aus dem mittleren Mausmenü verändern kann.

7.3 Implementierung

Der GlobalsBrowser ist eine Subklasse von `DictionaryInspector`, die einfach um eine zweite Spalte erweitert wurde und das mittlere Mausmenü bzw. die Methoden überdefiniert, die den Text zurückgeben, der in den Fenstern angezeigt wird.

Der TypeBrowser ist eine Subklasse von Browser, in dem zwei weitere Fenster eingehängt wurden (was sich auch in einem vergrößerten Platzbedarf auf dem Bildschirm äußert). Im Typfenster wurde die Acceptmethode, `acceptText:from:` überdefiniert, so daß sie die Subtypen und die Methodensignaturen behandeln kann. Diese Umdefinition erwies sich am Anfang als relativ problematisch, da entweder das geänderte Menü auch in anderen Fenstern erschien oder aber (bei einem etwas anderen Code) im Typfenster das unmodifizierte Menü

des Hauptbrowserfensters. Dieser Effekt kommt daher, daß einige Smalltalkwerkzeuge wie der Browser aus Performanzgründen z.B. Menüs in Klassenvariablen halten, die bei jedem neuen Dialogobjekt neu besetzt werden, was sich auch auf das Verhalten der anderen Dialogobjekte auswirkt und etwas unintuitiv für den Programmierer ist. Die übrigen Änderungen zu den Methoden des Browsers erfolgten im Zusammenhang mit der Programmierunterstützung durch Typen. Daher sind diese Details im entsprechenden Abschnitt des Kapitels erwähnt, das sich mit der Unterstützung beschäftigt. Die Implementierungsaspekte der Typprüfung wurden bereits in Kapitel 6.4 behandelt.

8 Programmierunterstützung durch Typen in Smalltalk-80

Die Ansätze, Typen in Smalltalk einzusetzen, haben ihre Typsysteme und deren Vorteile und Nachteile beschrieben. Wenig bekannt ist jedoch, wie sich die Benutzung von Typen auf die Programmierung auswirkt. Einige Autoren [CCHO89] erwähnen die Vorteile, die einer Programmierumgebung aus dem Gebrauch von Typen erwachsen würden, lassen diesen Punkt aber entweder im Dunkeln oder verweisen darauf, daß dies ein zukünftiges Forschungsthema sein könnte.

Eine derartige Arbeit muß von den Nutzen berichten, die die Verwendung vonTypen in Smalltalk bringt, aber auch von den Kosten dieser Verwendung. Der Nutzen läßt sich in vier Bereiche einteilen:

1) Verbesserte Dokumentation von Quellcode
2) Verbesserte Fehlerfreiheit im Programm
3) Verbessertes Laufzeitverhalten (Performanz) der Anwendung
4) Verbesserte Programmierunterstützung

Dabei wirkt sich die verbesserte Performanz nur am Rande auf die Programmierung aus, dann nämlich, wenn das Testen von Programmen durch einen schnelleren Testdurchlauf verbessert wird; außerdem behandelt [JGZ88] den Aspekt der Performanz (außer auf die Programmierung) eingehend. Daher will ich auf die Darstellung dieses Bereiches in meiner Arbeit verzichten.

Die Verbesserung der Quellcodedokumentation zerfällt in zwei Teile: Da ist zum einen die traditionelle Dokumentation: ein externer Text, der Programmteile, insbesondere deren Schnittstelle nach außen hin beschreibt und den andere Programmierer benötigen, um Programme zu schreiben, die mit den beschriebenen interagieren. Darunter fällt in Smalltalk-80 auch der Klassenkommentar, ein Text, der zu jeder Klasse hinzugegeben werden kann, auch wenn er in gewisser Weise "im Code" steht. Hier kann ein Kommentarprüfer automatisch nach Variablennamen und Typen suchen und diese gegen die tatsächlich vorhandenen Typen und Variablennamen prüfen und gegebenenfalls ergänzen oder berichtigen, wodurch die Dokumentation ein gutes Stück aktueller (durch den Automatismus) und genauer (durch die Typen) wird. Der zweite Bereich ist die Selbstdokumentation von Code, der gerade in Smalltalk-80, wo der Quellcode zu beinahe jeder Methode zur Verfügung steht, eine wichtige Rolle spielt. Typisches Programmieren in Smalltalk besteht im Untersuchen von Quelltext und in der Modifikation desselben. Eines der Probleme, das dabei entsteht, ist die Suche nach der Klasse von Objekten, die durch Variablen gebunden sind. Im normalen ungetypten Smalltalk kann die Suche danach durch das Studium mehrerer Methoden führen und wird bisweilen durch undokumentierte Klassen behindert. Typen, die bei der Deklaration jeder Variable genau definieren, welche Werte diese Variable annehmen kann, helfen Methoden richtig anzuwenden und Code richtig zu modifizieren.

Einer der Vorteile, die am häufigsten für die Verwendung von Typen genannt werden, ist das Aufdecken von Programmierfehlern zur Übersetzungszeit, die sonst erst später zu ungewolltem Verhalten von Programmen führen können. Die Fehler, die von einem Typsystem aufgedeckt werden können, sind die, die aus der Kluft zwischen der Belegung von Variablen entstehen, die der Programmierer annimmt und der Belegung, die wirklich in einem Programm auftreten können. Ein Beispiel dafür ist die Verwendung von Methoden als Funktionen, die gar keinen Rückgabebefehl haben. In einem solchen Fall gibt Smalltalk self zurück, also den Empfänger der Methode. Da dieser Fall nicht vom Programmierer in Erwägung gezogen worden ist, kommt ein Objekt einer anderen Klasse zurück als der, die er erwartet hat und irgendwann wird diesem Objekt eine Nachricht geschickt, die es nicht versteht, worauf Smalltalk ein “Does not understand”-Fenster öffnet. Dieser Fall kann mit einem geeigneten Typsystem nicht auftauchen, da der Programmierer hier den Typ des Rückgabewertes spezifiert hat und direkt bei der Programmierung mit der Information konfrontiert wird, daß dieser Typ gar nicht zurückgegeben wird. Nicht alle Typsysteme können alle diese Fehler entdecken (sind also Type-safe), weil z.B. die Erkennung ob eine Variable initialisiert worden ist oder nil enthält nur durch nicht-lokale Mechanismen entdeckt werden kann, aber im Großteil der Fälle dürften sie solche Fehler erkennen. Ein anderes Beispiel für die verbesserte Fehlerfreiheit in Programmen sind die Strukturparameter von Typen, die beschreiben, wie die *innere* Struktur der Objekte beschaffen ist und die durch die Spezifikation dieser Struktur helfen können, Programmierfehler aufzudecken. Ein Beispiel dafür ist der einzige derzeit in NTST verwirklichte Typparameter für Blocktypen, der angibt, wieviele Parameter einem Block übergeben werden. Beispielsweise hätte Array>>do: die Signatur do: aBlock<BlockClosure of: 1>, d.h. der Programmierer von do: verlangt, daß Blöcke, die als Parameter an do: übergeben werden genau einen Parameter haben. Damit würde der Typprüfer #(1 2 3) do: [:i | Transcript show: i] akkzeptieren, nicht aber #(1 2 3) do: [:i :j | Transcript show: i; show: j] obwohl das der Übersetzer nicht merkt und erst bei der Evaluierung des Blockes eine Fehlermeldung bringt. Ein weiterer verbreiteter Strukturparameter (der allerdings nicht in NTST realisiert wurde) sind Konstrukte wie <Array of: Integer>, das hier besagt, daß die Elemente eines Arrays ausschließlich vom Typ Integer sind.

Ein typloses System wie Smalltalk weiß über die Empfänger von Nachrichten in Methoden nicht sehr viel. Wenn ein Programmierer z.B. wissen will, in welchen Klassen die Methoden definiert sind, die seine Methode benutzt (messages), dann bekommt er lediglich eine Liste mit den Klassen, die Methoden anbieten, die die gleichen *Namen* haben wie die in seiner Methode. Daher werden mit einiger Sicherheit auch solche Klassen aufgelistet, die er überhaupt nicht benutzt. Durch die Verwendung von Typen kann die Programmierumgebung zur Übersetzungszeit sehr viel mehr Information über die Methodenempfänger erhalten und könnte deshalb bei messages nur die Klassen bekommen, an die seine Methode wirklich Nachrichten verschickt. Das gleiche gilt für alle Funktionen des Browsers, die aufgrund des fehlenden Typsystems nur über die Methodenselektoren se-lektieren können. Ein anderer Aspekt der verbesserten Programmierunterstützung könnte der Einsatz von über das Typsystem gewonnenen Informationen im Debugger sein.

Die möglichen Nachteile lassen sich ebenfalls in Bereiche einteilen:

- Belastung des Programmierers durch Tätigkeiten, die nur durch das Typsystem entstanden sind
- Zwang zu Programmieren mit Typen
- Einschränkung des Programmiermodelles und der Programmiertraditionen und -konventionen durch das Typsystem
- Störungen durch bereits getypten Code

Variablen mit Typinformationen zu versehen ist sicher mehr Arbeit als dies nicht zu tun, auch das Beseitigen der dadurch aufgedeckten Fehler und die Arbeiten, die mit der Typhierarchie entstehen, äußern sich sicher in einer geringeren Anzahl von getippten Zeilen Code. Diese sehr kurzfristigen Nachteile werden sicher durch größere längerfristige aufgehoben, aber gerade Programmierer, die zum ersten Mal mit einem Typsystem konfrontiert werden, werden diese Vorteile erst erfahren müssen. Neben diesen 'natürlichen' Problemen der Mehrarbeit durch Typen besteht weiterhin die Gefahr, daß ein Smalltalk mit Typen auch dem Programmierer Arbeit abverlangt, der gar keinen getypten Code schreibt.

In einem Smalltalk, das kein Nebeneinander von getyptem und ungetyptem Code erlaubt (gemischter Code), muß ein Programmierer alle seine Variablen typisieren. Dieses frühe Einbringen von Typinformationen bringt wahrscheinlich besseren Code mit sich, es ist allerdings die Frage, ob die Produktivität und die Motivation des Programmierers nicht unter den erschwerten Bedingungen leidet, was sich in Ablehnung des Typsystems und damit dem gesamten Nutzen desselben äußern könnte. Außerdem gibt es kein Smalltalk-80 System, das vollständig getypt ist, was sicher eine Voraussetzung für ein solches strenges Typsystem ist. Ein vollständiges Typisieren bestehenden Smalltalk-80 Codes ist darüber hinaus vielleicht unmöglich (weil die Typdeklaration ja auch sagt, welche Objekte an eine Variable gebunden sein müssen und es durchaus sein kann, daß es kein vernüftiges Typsystem gibt, das die Subtypbeziehungen zur Verfügung stellt, die der bestehende Code benötigt) oder wenig aussagekräftig (Typisierung aller Variablen mit `<Object>`. Bei neuen Smalltalkdialekten besteht natürlich die Möglichkeit, den gesamten Code zu typsystemgerecht aufzubauen.

Einige Typsysteme beschränken Mechanismen, die bei ungetyptem Smalltalk nicht restringiert wären. Ein Beispiel für diesen Fall ist [BI82], die in ihrem System verlangen, daß die Rückgabetypen von überdefinierten Methoden Subtypen der Rückgabetypen der Superklasse sind. Solche Einschränkungen behindern das Typisieren von bestehendem Code (und enden mit einer Neustrukturierung des Programmes, eine sehr drastische Maßnahme) oder verlangen vom Programmierer eine Änderung seines Programmiermodells.

Die erweiterte Syntax von Smalltalk, die nötig werden könnte, um typisierte Ausdrücke auszudrücken kann bei ungeschickter Einbindung in ein Smalltalk Probleme bereiten. Noch wichtiger ist allerdings das mögliche Problem, daß der Typprüfungsmechanismus die Lauffähigkeit einer Smalltalkanwendung behindert; daher sollten alle Prozeße in diesem Zusammenhang zur Übersetzungszeit abgearbeitet sein.

Ein für den Programmierer nützliches Typsystem muß also soviel Nutzen haben, daß es die Mehrarbeit, die durch die Typisierung entsteht zumindest ausgleicht, es sollte ihm die Möglichkeit geben, zu entscheiden, ob er getypten oder ungetypten Code schreibt und diese beiden

Bereiche müssen “friedlich” miteinander leben können.

Um Erfahrungen mit dem Verhältnis von Vor- und Nachteilen eines Typsystems in Smalltalk zu machen wurden einige Erweiterungen des Browsers vorgenommen, die ohne ein Typsystem nicht möglich gewesen wären. Davon berichtet der nächste Abschnitt. Danach wurde ein Teil des Codes des NTST-Systems (also eine kleinere Anwendung mit Methoden, die eher länger sind, als dies unter Smalltalk üblich wäre) typisiert; der übernächste Abschnitt berichtet von diesem Bemühungen. Obwohl die Erweiterungen und der Erfahrungsbericht auf NTST beruhen, will ich versuchen zu zeigen, ob diese Ergebnisse auch auf andere Typsysteme anwendbar sind.

8.1 Änderungen im Browser

Der Systembrowser ist in Smalltalk-80 *das* Werkzeug zur Programmierung; mit ihm kann man sich den Code von Klassen und Methoden ansehen, sich Informationen über die Typhierarchie und das Vorkommen von Variablen und Nachrichten anzeigen lassen, den Code in eine Datei abspeichern und schließlich auch eigene Klassen und Methoden definieren.

Der Browser ist daher auch der Punkt, an dem neue oder veränderte Werkzeuge zur Unterstützung der Programmierung ansetzen müssen.

Exemplarisch für diese erweiterten Werkzeuge wurden drei Möglichkeiten des System Browsers modifiziert:

1) `senders`: Wer sendet eine Nachricht, die die selektierte Methode aufruft?
2) `messages`: Welche Klassen stellen die Methoden zur Verfügung, die die selektierte Methode aufruft?
3) `comment`: Der Kommentar zur selektierten Klasse

Bei `senders` gibt der Browser einen weiteren Browser mit all den Methoden aus, die Nachrichten mit den Selektoren versenden wie der, der den Kopf der aktuell selektierten Methode definiert. Die Anzahl der Methoden, die im neuen Browser erscheinen ist hier normalerweise größer als die der Methoden, die die Methode wirklich aufrufen, weil zwischen verschiedenen Empfängern einer Methode nicht vor der Laufzeit unterschieden werden kann. Da ein Typsystem unter anderen diese Information liefern kann, falls die Methode, in der der gewünschte Selektor verschickt wird, getypt ist, kann sich hier getypter Code reduzierend auf die Anzahl der aufgelisteten Methoden bemerkbar machen. Leider versagt dieser Mechanismus dann, wenn die zu untersuchenden Methoden ungetypt sind. Daher ist das erweiterte `senders`-Fenster zweigeteilt (siehe Abbildung 9): Im oberen Fenster stehen all die Methoden, von denen das Typsystem weiß, daß sie die selektierte Methode aufrufen, weil sie getypt sind und darauf hin untersucht wurden, und im unteren Fenster steht der Rest der Methoden, die der unmodifizierte `senders`-Aufruf erbracht hätte, nur ohne die Methoden aus dem oberen Fenster und ohne die Methoden, die getypt waren und für die entschieden werden konnte, daß sie andere Empfänger als die aktuelle Klasse referenzieren.

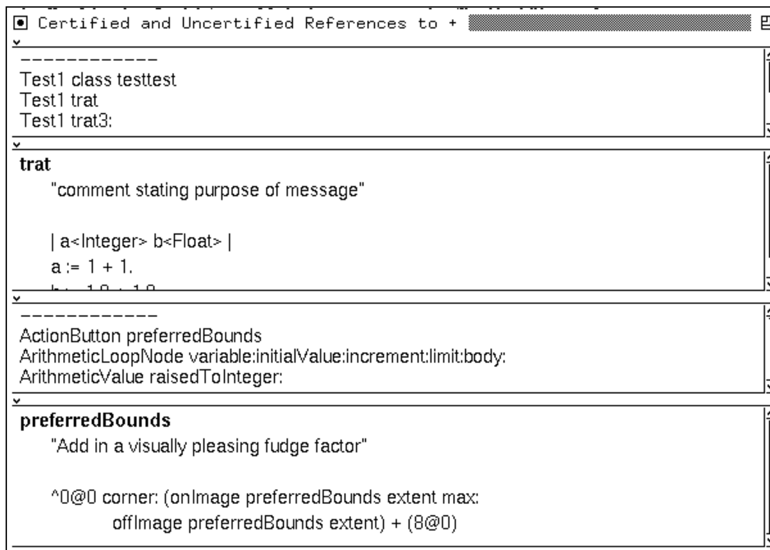

Abbildung 9

Der Menüpunkt `messages` im Methodenmenü gibt normalerweise eine Liste mit den Nachrichten aus, die in der selektierten Methode verschickt werden. Der Programmierer kann eine Nachricht aus dieser Liste auswählen und das System öffnet darauf hin einen Browser, in dem alle Klassen stehen, die eine entsprechende Methode zur Verfügung stellen. Diese Liste enthält für gewöhnlich mehr Klassen als die, die wirklich relevant sind, da der Browser ohne Typen natürlich nicht weiß, welchen Empfänger diese Nachricht beim Programmablauf haben wird (siehe Abbildung 10).

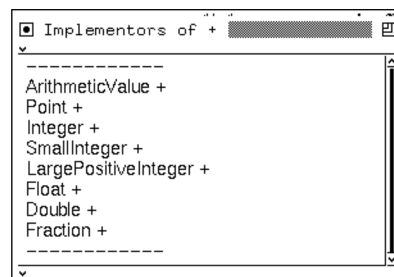

Abbildung 10

An dieser Stelle hilft ein Typsystem, da durch die Typdeklarationen klar ist, welche Klassen die möglichen Empfänger sind, nämlich alle Subtypen des Typs des Empfängers der Methode. Genau das macht dann auch die Erweiterung von `messages`, die oft zu einer Einschränkung des Suchraums führt (Abbildung 11), wobei sich die Genauigkeit der Typdeklaration direkt

auf diese Reduktion auswirkt. Die Vorgehensweise bei einer ungetypten Methode ist die des unmodifizierten Browsers. Bei Typsystemen auf der Basis von Interfaces würden hier statt der Liste mit den Subtypen eine Liste mit den Klassen auftauchen, die das Interface erfüllen.

Im Klassenkommentar wird normalerweise jede Klassen- und Instanzenvariable samt informellem Pseudotyp und einer kurzen textuellen Erklärung aufgeführt. Wenn man ein Typsystem hat, dann können die informellen Typen verifizierter Information weichen. Dazu wird

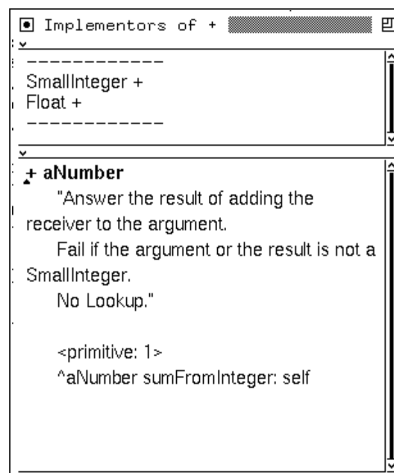

Abbildung 11

der Klassenkommentar nach dem Vorkommen von Paaren von Strings und Typdeklarationen abgesucht, die mindestens durch ein Leerzeichen getrennt sind. Dann wird der String als Variablennamen interpretiert, wobei Strings, die mit einem Großbuchstaben anfangen, als Klassenvariablen und solche, die mit Kleinbuchstaben anfangen als Instanzenvariablen gedeutet. Falls die gefundenen Variablen tatsächlich definiert sind, prüft das System, ob die Typen im Kommentar und die der Variablendeklaration übereinstimmen. Ist dies nicht der Fall, fragt das System den Programmierer, ob der Kommentar entsprechend angepaßt werden soll.

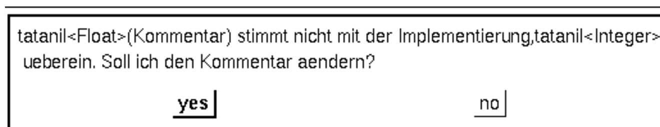

Abbildung 12

Ist das der Fall, dann wird der Kommentar geändert. Wird ein Variablennamen nicht als Variable der Klasse gefunden (falls sie also noch nicht realisiert ist), fragt das System, ob dieser Eintrag aus dem Kommentar genommen werden soll und tut dies gegebenenfalls.

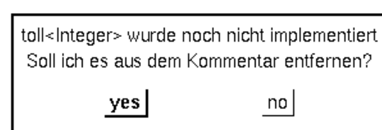

Abbildung 13

Wird eine Variable gefunden, die nicht im Kommentar auftaucht, wird diese samt Typ in den Kommentar mit aufgenommen, falls der Programmierer das wünscht. Für den Kommentar gilt, daß jede Klassen- bzw. Instanzenvariable eine eigene Zeile bekommen muß, damit das System korrekt arbeitet.

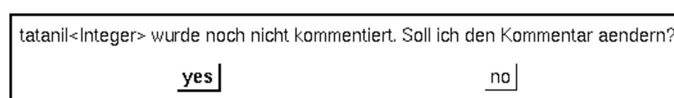

Abbildung 14

Neben den oben aufgeführten Erweiterungen des Browsers gibt es natürlich eine Reihe von weiteren möglichen Erweiterungen, etwa die Ersetzung des allgemeinen Klassenformulars im Browser durch ein Methodenskelett (Methodenkopf samt Typen und Returntyp) bei Methoden, die zwar schon referenziert werden, aber noch nicht implementiert sind.

Es erwies sich als nicht sinnvoll, alle Browserwerkzeuge zu modifizieren. Als Beispiel möge der Punkt `implementors` dienen, mit dem man sich anzeigen lassen kann, welche Klassen noch eine Methode mit dem gleichen Selektor implementieren. Man könnte natürlich den Suchraum auf die Klassen einschränken, die Methoden mit der gleichen *Signatur* anbieten, aber dann würde man wahrscheinlich zu wenig Klassen bekommen, wenn man daran interessiert ist, zu sehen, wie diese Methode von anderen Klassen verwirklicht wird, auch wenn diese die gleiche Funktionalität für andere Parameter verwirklichen könnten.

8.2 Implementierung

Alle nötigen Erweiterungen wurden in der Klasse TypeBrowser vorgenommen, der Subklasse von Browser, die den NTST Type Browser realisiert.

Verifizierung der Typen im Klassenkommentar
Hier wurde die Methode `acceptMessage:` so modifiziert, daß der neue Kommentar zuerst der Methode `typecheckComment` der Klasse gegeben wird, die gerade im Browser selektiert ist. Diese Methode scannt den Kommentar nach den Variablenname-Typ-Paaren und modifiziert diese gegebenenfalls (falls z.B. der Typ nicht stimmt). Der Rückgabewert dieser Methode ist ein Kommentar, der den Vorstellungen des Benutzers gerecht werden sollte. Bemerkungen, die hinter den Paaren stehen, werden in die modifizierten Zeilen herübergerettet. Vor der Untersuchung der Paare werden alle Tabulatoren und Leerzeichen vor und zwischen dem Paar auf ein Leerzeichen reduziert. Die modifizierte Zeile wird mit einem führenden Tabulatorzeichen versehen, so daß die Paare einheitlich eingerückt sind.

Änderungen im Browser: senders
Hier wurde der Listbrowser so modifiziert, daß er zwei verschiedene Listen von Methoden anzeigen kann. In der unteren Liste sind all diejenigen Methoden, die nicht getypt sind und die Smalltalk bei einem normalen `senders`-Aufruf gezeigt hätte minus die Methoden, die in Frage kommen, aber getypt sind. Die getypten Methoden werden nochmals typgeprüft. Dabei werden die Empfängertypen der Nachricht ermittelt, deren Sender gerade gesucht werden. Stimmt der Typ der Empfänger mit der aktuellen Klasse überein, so wird die so ermittelte Methode im oberen Fenster angezeigt. Je mehr Methoden im System getypt sind, umso weniger Methoden erscheinen im unteren Fenster.

Änderungen im Browser: messages
Nachdem der Benutzer den Selektor ausgesucht hat, dessen Code er zu sehen wünscht, sucht die Methode `TypeBrowser>>allImplementorsOf:` im Code der gerade im Browser aktiven Methode nach den Typen der Empfänger dieses Selektors. Das geht natürlich nur, wenn die gerade aktive Methode getypt ist (ansonsten arbeitet `messages` wie gewohnt). Nach die Empfänger mittels `Type>>method:class:request:` gefunden wurden, werden sie und ihre Subklassen (die ja ebenfalls den Selektor bekommen könnten) in der Reihen-

folge angezeigt, wie sie im Quelltext vorkommen.

8.3 Ein Erfahrungsbericht

Als “Versuchskaninchen” wurde die (ungetypte) Klasse `TypeBrowser` ausgesucht, und zwar aus drei Gründen:

1) Die Klasse implementiert eine typische Smalltalkanwendung, ein grafisches, interaktives Objekt
2) Die Klasse enthält einige lange Methoden
3) TypeBrowser ist eine Subklasse von Browser, einer häufig benutzten Standardklasse von Smalltalk

Aus dieser Klasse wurden einige Methoden nacheinander typisiert, wobei der Bedarf an Zeit für die Typsisierung über der Komplexität der Methoden gemessen werden sollte. Das Typsystem und seine Werkzeuge sollten außerdem zeigen, wie praxistauglich sie sind.

Schon bald nach Beginn dieser Typisierung wurde klar, daß ein gemessener Zeitbedarf im vorliegenden System wenig aussagekräftig sein würde. Dies liegt daran, daß NTST von allen Nachrichten, die in einer zu typisierenden Methode verschickt werden können verlangt, daß ihre Signatur bekannt ist. Das führt bei einem System, in dem nur wenig Methoden im unmodifizierten Smalltalkkern typisiert sind dazu, daß die Typisierung einer Methode die Deklaration aller ihrer aufrufbaren Methoden nach sich zieht. Sind einmal genügend Methoden auf diese Weise bearbeitet worden, wird es immer unwahrscheinlicher eine Methode zu benutzen, deren Signatur nicht bekannt ist (die Signaturen werden ja gespeichert). Daher fließt die Zeit für die Erstellung der Signaturen dann auch nicht mehr so stark in den Zeitaufwand für die Typisierung ein. Das Gleiche gilt für die Subtypenrelationen, die noch von Hand eingetragen werden müssen, wenn der Typprüfer meldet, daß ein Selektor von einem Typ nicht verstanden wird.

Auf der anderen Seite wurden die Methoden vom Programmierer des Typsysstems bearbeitet, der sich über die Reaktion des Typsystems relativ klar war.

Ebenfalls als problematisch erwies sich das Fehlen der Pseudotypen wie SELF. Ein illustrierendes Beispiel für diese Problematik stellt die Verwendung der Methode new in der Klasse Object dar. Diese Methode erzeugt eine Instanz des Empfängers und gibt diese zurück. Bei den Klassen, die new unmodifiziert erben ist daher der Rückgabetyp von new der Typ einer Instanz dieser Klasse. Weil jedoch NTST derzeit keinen Pseudotyp SELF, der den Typ der Empfängerklasse referenziert, kennt, muß der Typ von new in Object `<Object>` sein. Da dies aber der allgemeinste Typ ist, trägt er keine Typinformation und kann daher auch keine möglichen Typfehler aufdecken. Eine wenig elegante Lösung des Problems ererbter Methoden ist die Nachbildung des Mechanismus, den das Typsystem bei der Verwendung von SELF leisten müßte. Dazu wird new in allen Klassen überdefiniert. Das Smalltalk-Statement für diese Methode ist dann `^super new` und der Rückgabetyp kann bei jeder einzelnen Klasse auf den Typ einer Instanz dieser Klasse gesetzt werden.

Beispiel:

```
TypeBrowser>>new argumenttypes:#() resulttype:'<TypeBrowser>'

new
     ^super new
```

Die Notwendigkeit, wegen der Typisierung unnötig Code schreiben zu müssen, schränkt den praktischen Wert der Vererbung in solchen Fällen stark ein, aber bei Systemen mit einem Pesudotyp SELF kann dieses manuelle Vorgehen transparent für den Programmierer automatisiert werden [Gr89], auch wenn dann eine Codeveränderung in der vererbten Methode eine Neuübersetzung aller Subklassen mit sich zieht.

Bedingt durch die Integration der Deklaration des Rückgabetyps über die temporäre Variable Return kommt es bei der Übersetzung in solchen Fällen immer zu einer Warnung durch den Übersetzer, der (zu Recht) findet, daß Return nie ein Wert zugewiesen wird. Dies ist bisweilen etwas lästig und bedeutet eine unnötige Aktion durch den Programmierer (Wegklicken der Warnung). Zumindest unästhestisch ist auch die Eigenart des Typprüfers, in der nach der Prüfung einer Methode gefunden Signatur Rückgabetypen mehrfach zurückzugeben. Dies dürfte im Betrieb zwar zu keinen Problemen führen, verursacht aber zumindest etwas längere Prüfzeiten.

Als unangenehme Eigenschaft der vorliegenden NTST-Realisierung ist auch die Beschränkung auf einen TypeBrowser anzusehen sowie die Tatsache, daß es keine Teilbrowser (Methoden-, Klassen, Kategorienbrowser) mit Typunterstützung gibt. Die Folge davon ist, daß häufig zwischen der zu typenden Methode und den Klassen und Signaturen hin- und hergeschaltet werden muß, die für die Typprüfung ergänzt werden müssen. Das verlangt vom Programmierer Denkarbeit, da er die noch zu ändernden Einträge im Gedächtnis halten muß. Hier wäre ein zweiter TypeBrowser vonnöten oder ein Typprüfer, der in einem Fenster die benötigten Einträge (Signaturen oder Subtypen) vorschlägt und dem Programmierer die Möglichkeit läßt, diesen Vorschlag zu ändern, um den neuen Eintrag gleich in der Typprüfung vornehmen zu können.

Da der Benutzer Signaturen und Subtypenrelationen deklarieren kann und diese vom Typsystem nicht oder nur in Grenzfällen überprüft werden können, kann es nicht nur vorkommen, daß diese Deklarationen nicht der Wirklichkeit entsprechen, es kann auch sein, daß die Änderung einer Signatur eine erneute Typprüfung von solchen Methoden verlangen müßte, die sich auf die ursprüngliche Signatur bezogen hatten. Letzteres Problem kann man dadurch umgehen, daß Typen in Signaturen nur so geändert werden, daß als resultierender Typ des ursprünglichen und des neu einzutragenden Typs der NCS genommen wird (dann sind erfolgreich geprüfte Methoden auch weiterhin getypt).

Abgesehen von den Kosten des Typsystems und den Problemen seiner Implementierung ist auf der Habenseite die Auffindung eines Programmfehlers zu verbuchen, der durch das Typsystem gefunden werden konnte. Es handelte sich einen Befehl, der in einem Failure-Block einer Nachricht enthalten und noch nie ausgeführt worden war. An dieser Stelle wurde dem Wert einer Variable eine Nachricht geschickt, die in den Objekten, die als mögliche Werte in

Frage kommen konnten nicht implementiert war und die irgendwann im laufenden Betrieb eventuell eine Does-not-understand-Botschaft ausgelöst hätte. Der Effekt eines Typsystems Fehler in Code zu finden, der noch nicht ausgeführt worden ist, wird in der benutzten Literatur nicht erwähnt. Er rührte in diesem Fall von der Tatsache her, daß ein strenges Typsystem weitergehende Informationen über den möglichen Wertebereich von Variablen hat als ein normaler Smalltalkübersetzer, der nicht eruieren kann, ob eine Nachricht, die an das Objekt geschickt werden wird, das an eine Variable gebunden wird, erfolgreich ausgeführt werden kann oder nicht. Die Erkenntnis über diesen Programmierfehler muß der Benutzer allerdings selber machen, ein Typsystem kann immer nur von Verstößen gegen die Typregeln berichten. Was diese Verletzung ausgemacht hat, bleibt nicht nur in NTST der Exploration der Programmierers unterworfen.

Schließlich typisiert wurden 7 Methoden von TypeBrowser. Tabelle 1 gibt Auskunft über die Komplexität der Methoden indem sie die Anzahl der sendbaren Nachrichten und die der möglichen Rückgabebefehle neben den Zeitbedarf für die Typisierung stellt. Dabei konnte gerade am Anfang der Zeitbedarf nicht für alle Methoden gemessen werden, da durch die Typisierung noch Fehler des NTST-Systems aufgedeckt wurden, die danach beseitigt werden mußten.

Die Zeitdauer zwischen einem Accept der Methode und dessen Beendigung nahm bei den kurzen Methoden subjektiv nicht zu, allein bei der langen Methode stieg er von 1 Sekunde für die ungetypte Version auf etwa 3 Sekunden, was aber im Verhältnis zum Zeitbedarf der eigentlichen Typisierung als nicht sehr bedeutend empfunden wurde.

Table 1: Zeitbedarf

Methode	Anzahl der Nachrichten	Anzahl der Returns	Zeitbedarf der Typisierung
textMode	2	0	7 min
typeText	103	17	?
typeTextMenu	4	1	?
t.AcceptTextFrom:	15	3	?
protocolList	10	2	10 min
protocolMenu	8	2	10 min
evaluateString	3	1	?

Insgesamt läßt sich sagen, daß ein benutzbares Smalltalk mit Typen den Aufwand des Typisierens von Methoden durch eine (zumindest durch Signaturen) durchgetyptes Virtual Image im überschaubaren Bereich halten sollte. Es muß unbedingt Pseudotypen wie SELF unterstützen und es sollte möglichst wenig Denkarbeit vom Programmierer verlangen. Die Werkzeuge

sollten an vielen Stellen anbieten, benötigte Signaturen automatisch einzutragen, aber nur dann, wenn die Korrektheit dieser Signaturen entweder automatisch festgestellt wurde oder dem Programmierer die Gelegenheit gegeben wurde, diese Korrektheit selber festzustellen.

9 Zusammenfassung

Smalltalk ist eine vollkommen ungetypte Sprache, d.h. an jede Variable können Objekte jeder Klasse gebunden werden. Dennoch gibt es verschiedene Ansätze, Smalltalk mit einem Typsystem auszustatten. Typen sind in Smalltalk Restriktionen über den Klassen von Objekten, die an Variable gebunden werden können. Diese Restrriktionen werden entweder durch die Zugehörigkeit zu einem Teilast des Klassenbaumes oder bestimmte Eigenschaften von Klassen bestimmt. Falls der erste Ansatz gewählt wird, müssen Subklassen bestimmte Eigenschaften erfüllen, die sie in ungetyptem Smalltalk nicht haben müssen, weil dadurch angenommen wird, daß Klassen neben ihrer Implementierung auch ihre Spezifikation vererben. Das hat zur Folge, daß bestehender Code geändert werden muß, falls er typisiert werden soll. Typsysteme lassen sich außerdem noch nach ihren Sprachelementen kategorisieren. Jedes Sprachelement erhöht die Mächtigkeit der Typsprache und die Menge an Wissen über ein Objekt, die durch Typen ausgedrückt werden kann.

Die Gründe für die Verwendung von Typen sind:

- Die bessere Lesbarkeit von Quellcode
- Die Möglichkeit, die Dokumentation aktueller zu halten
- Die Möglichkeit des Vermeidens von “Does-not-understand”-Meldungen zur Laufzeit
- Die Erkennung einiger Programmierfehler zur Übersetzungszeit
- Die Möglichkeit der Performanzsteigerung von Anwendungen
- Die verbesserte Programmierunterstützung
- Die verbesserten Möglichkeiten des kooperativen Programmierens

Diese Gründe müssen dabei den Mehraufwand rechtfertigen, der durch die Typisierung von Code entsteht.

Es wurden vier verschiedene Ansätze eines Typsystems in Smalltalk vorgestellt. Es wurde gezeigt, daß in Smalltalk-80 4.1 Möglichkeiten bestehen, bestimmte Variablen zu typisieren und daß diese nicht nützlich sind und nicht benutzt werden. Es wurde ein weiteres Typsystem, NTST, vorgestellt und diskutiert, das dieses Typfragment um die notwendige Funktionalität erweitert. Es wurden zwei Werkzeuge vorgestellt, die die Schnittstelle von NTST zum Programmierer darstellen. Dann wurden die Möglichkeiten untersucht, die Programmierung durch die Verwendung von Typen zu unterstützen. Dabei zeigte es sich, daß diese Verwendung bei geeigneter Modifikation der Programmierwerkzeuge dazu führen kann, daß

- der Suchraum auf der Klassenhierarchie beim Browsen verkleinert werden kann
- der Programmierer wesentlich weniger Code durchsuchen muß, bis er versteht wie Methoden arbeiten

Die Präsentation des Typsystems und die Vorteile, die dessen Verwendung dem Programmierer bringen, sind dabei jenseits von objektiven Nützlichkeitsüberlegungen entscheidend für die Akzeptanz durch die Programmierer.

10 Ausblicke

Die vorliegende Arbeit versuchte deutlich zu machen, daß Typen zu einem Smalltalk führen könnten, das Vorteile gegenüber der ungetypten Sprache hat. Es ist jedoch noch offen, ob ein solches System von den Programmierern angenommen und benutzt wird. Hier sollte eine Arbeit ansetzen, die diese Fragestellung untersucht und im Dialog mit den Benutzern das Typsystem und seine Implementierung modifiziert, wie auch Smalltalk selber im Dialog mit seinen Anwendern entstanden ist. Ein weitere interessante Untersuchung in diesem Zusammenhang wäre die Frage, ob Programmierer getypten Code besser und schneller verstehen als ungetypten.

Es scheint sich aber schon jetzt abzuzeichnen, daß objekt-orientierte Sprachen im Zusammenhang mit verteilten Systemen eine immer wichtigere Rolle spielen werden und unter diesem Aspekt sind Typen eine der Voraussetzungen für die Kooperation mehrerer Programmierer in einem System, weil dann der Aspekt der Definition von Programmschnittstellen und die Prüfung auf deren Einhaltung essentiell wird.

Ein weiterer Gesichtspunkt sind die Möglichkeiten, die Typen im Zusammenhang mit der Optimierung und der Übersetzung in Code bieten, der nur ein kleines Laufzeitsystem benötigt. Hier liegt der Schlüssel zu einer massenhaften industriellen Verwendung von Smalltalk, die sich in einer Zeit, in der man noch mehr Fortran als andere Sprachen einsetzt, nur positiv auf die Wartbarkeit und Erweiterbarkeit von Software auswirken kann.

Anhänge

A Anmerkung zum Quellcode

Der Quellcode für das NTST-System ist ein File-Out der entsprechenden Klassen für Smalltalk-80 Version 4.1; er steht beim Betreuer dieser Arbeit, Herrn Markus Geltz, zur Verfügung. Er lief auf einer SparcStation 10 mit SunOS 4.1.3 . Die Anweisungen zur Installation (siehe Anhang B) beschreiben, wie die Software in ein Virtual Image eingebaut wird.

Zur Qualität des Codes sei bemerkt, daß er das Ergebnis von unüberarbeitetem, explorativem Programmieren ist und nur dem Zweck gedient hat, Smalltalk-80 mit irgendeinem Typsystem auszustatten, um an diesem Beispiel Überlegungen zu Thema Typen in Smalltalk zu machen. Diesen Zweck hat der Code voll erfüllt, aber das Ergebnis ist sicher nicht die Software, die man für praktische Zwecke einsetzen kann. Abgesehen davon, daß es wahrscheinlich sinnvoller wäre, einen anderen Ansatz für das Typsystem zu verwenden (etwa den von Strongtalk), fehlt dem Programm die Basis einer Problemanalyse (weil eben die Programmierung erst die Elemente der Analyse herauskristallisiert hat), mit dem Ergebnis, daß Klassen z.T. nur als Empfänger von Nachrichten dienen (Type class), von einigen Klassen nur einzelne Instanzen möglich sind (TypeBrowser) oder daß Methoden viel zu groß sind. Würde man daher wirklich ein Typsystem für Smalltalk auf der Basis von NTST schreiben wollen, müßte man den vorliegenden Quellcode wegwerfen und das System neu programmieren.

Nachfolgend einige Stellen aus dem Quellcode, die interessant sind:

Initialisierung des Parsers

Original

```
Parser methodsFor: 'initialize-release'

initScanner
   super initScanner.
   typeTable := TypeTable.
   "Default language choice:"
   oldLanguage := true.
   newLanguage := true.
   extendedLanguage := false
```

Modifikation

```
Parser methodsFor: 'initialize-release'

initScanner
   "initializes the Scanner.
```

```
Last Programmer: Fritz Hohl, Date: 17.3.94”

super initScanner.
typeTable := TypeTable.
“Default language choice:”
oldLanguage := true.
newLanguage := true.
extendedLanguage := true
```

B Installation der Software

Dieser Abschnitt beschreibt, welche Probleme sich bei der Erstellung eines Konfigurationsskriptes ergaben und wie das NTST-System installiert wird.

Das erste Problem ist ein Fehler in Smalltalk-80 4.1. Wenn man nämlich den Parser durch das Ändern eines Boolschen Wertes (siehe Anhang A) dazu bewegt, Typen in Methoden zu parsen, dann kann kein Code mehr compiliert werden, der ein *Primitive* benutzt, weil die Syntax der *Primitives* der der Typen ähnelt (z.B. <primitive: 321>) und dies nicht abgefangen wird. Die Einschränkung, im laufenden System keine weiteren *Primitive*-Befehle mehr eingeben zu können, könnte man noch dulden, wenn man jedoch eine Klasse lädt, die Primitive benutzt (wie dies beim Laden einiger NTST-Komponenten der Fall ist), dann würde diese Hürde das Laden dieser Klassen verhindern. Daher mußte dieses Problem behoben werden, die Klassen Scanner und Parser wurden so angepaßt, daß Primtive und Typen zusammen geparst werden können.

Das zweite Problem ist das, daß der Compiler im laufenden Betrieb erweitert wird, und dann eventuell Klassen und Methoden benötigt, die noch gar nicht geladen sind. Die Lösung dieses Problems bestand darin, zuerst eine Version des Compilers zu laden, der zwar eine erweiterte Funktionalität hat, diese aber nicht benutzt (weil das entsprechende Startkommando kommentiert wurde).

Die letzte Hürde ist das Laden eines Typsystems, das getypten Code enthält. Das Problem ist, ein System zu laden, das selber wieder dieses System zum Laden benötigt.Die Lösung dieser Probleme ist eine Vorgehensweise, die man *Bootstrapping* nennt. Darunter versteht man den Aufbau eines immer mächtigeren Systems mit den Mitteln, die ein schwächeres zur Verfügung stellt; eine Situation, die z.B. dann entsteht, wenn ein Rechner eingeschaltet wird und zuerst nichts kann, als Dateien über ein Netz zu laden und zu starten, und dann damit das eigentliche Betriebssystem lädt, das dem Rechner die eigentliche Funktionalität gibt. Bezogen auf NTST heißt das, daß erst ein Typsystem gestartet wird, das die Typen ignoriert und in dem das Typsystem ausgeschaltet ist und dann damit ein System zu laden, das eine Typprüfung durchführen kann.

Das Konfigurationsskript beachtet alle diese Abhängigkeiten. Es wird dadurch ausgeführt, daß man in einem geladenen Smalltalk-80 Version 4.1 das Kommando `(Filename named: 'NTST.st') fileIn` ausführt. Dies kann entweder in einem Workspace so gemacht werden, daß diese Zeile eingetippt wird und dann im mittleren Mausmenü `do it` ausgewählt wird oder mit dem Werkzeug Filelist aus dem Launcher, indem man auf den Dateinamen klickt und dann "`file in`" aus dem mittleren Mausmenü auswählt.Diese Datei muß dazu in dem Dateiverzeichnis stehen, von dem aus Smalltalk gestartet wurde. In diesem Verzeichnis muß das Verzeichnis `bootstrapping` mit den Dateien `Type_class-getName.st`, `SmalltalkCompiler2.st` und `Type.st` stehen.

Beispiel:

```
>st80 &
>pwd
/home/hohlfz
>ls
NTST.st
bootstrapping
> ls bootstrapping
SmalltalkCompiler2.st Type_class-getName.st Type.st
```

Die Datei `NTST.st` enthält die Klassendefinition einer Dummy-Klasse namens NTST, deren einzige Methode `initialize` ist. Diese Methode wird bei einem File-In einer Klasse immer automatisch ausgeführt. Diese Methode lädt zuerst die Klassen Scanner und Parser, in denen der Primitive-Fehler behoben worden ist und die das Parsen der erweiterten Smalltalk-Syntax einschalten, so daß Methoden mit Typdeklarationen geladen werden können. Dann wird eine von Funktionalität befreite Version der Klasse Type geladen, die einige Methoden bereitstellt, die andere Klassen nachher benötigen. Danach wird ein SmalltalkCompiler geladen, der die Typprüfung ausgeschaltet lässt und dann werden die Klassen des NTST-Systems nacheinander geladen. Am Schluß wird die Version des SmalltalkCompilers geladen, der die Typrüfung zuläßt und die Typinformation des aktuellen Systems wird nachgefahren.

Jetzt ist NTST einsatzbereit.

C Sonstiges

C.1 Beispiele von Typausdrücken aus den Klassenkommentaren

Diese Beispiele sind Auszüge aus den informellen Typkommentaren, die in den Klassenkommentaren des ausgelieferten Smalltalk-80 4.1 vorkommen.

```
<IntegerArray | (Array of: Integer)>
<BlockClosure argument: Symbol>
<BlockClosure with: Integer>
<"at least 2" SequenceableCollection of: MessageNode>
<(Array of: Symbol) size: 256>
<Dictionary key: Character value: Array>
<Dictionary keys: Character values: Symbol>
<Integer between: 1 and: 15>
```

C.2 Smalltalk-80 4.1 Parsebäume

Der Parser von Smalltalk-80 Version 4.1 erzeugt aus dem Quellcode einer Methode einen Parsebaum, bei dem `VariableNodes` Variablen kennzeichnen. Die einzige wesentliche Instanzenvariable dieser Objekte ist ein String, der den Namen der Variablen kennzeichnet. Wird im Parser nun die Typerkennung eingeschaltet, dann steht statt eines `VariableNode` ein `ParameterNode` oder ein `MessageNode` im Baum. Bei ersterem gibt es zwei Attribute, `name` und `type`, die den Namen der Variablen bzw. den Namen der Klasse enthalten, die diesen Typ kennzeichnet. Bei letzterem wird der (komplexe) Typ wie eine Smalltalk-Nachricht geparst.

Abbildung 15 erhält man, wenn man den folgenden Ausdruck mit `inspect` untersucht:

```
SmalltalkCompiler parse:
'test: einParameter<Integer>

     | temp<Collection of: Integer> foo<nil | Integer> |

     temp := Collection new.
     temp add: einParameter.
     temp do: [:i | | foo<Integer> | foo := i].
     ^einParameter'
class: Object
```

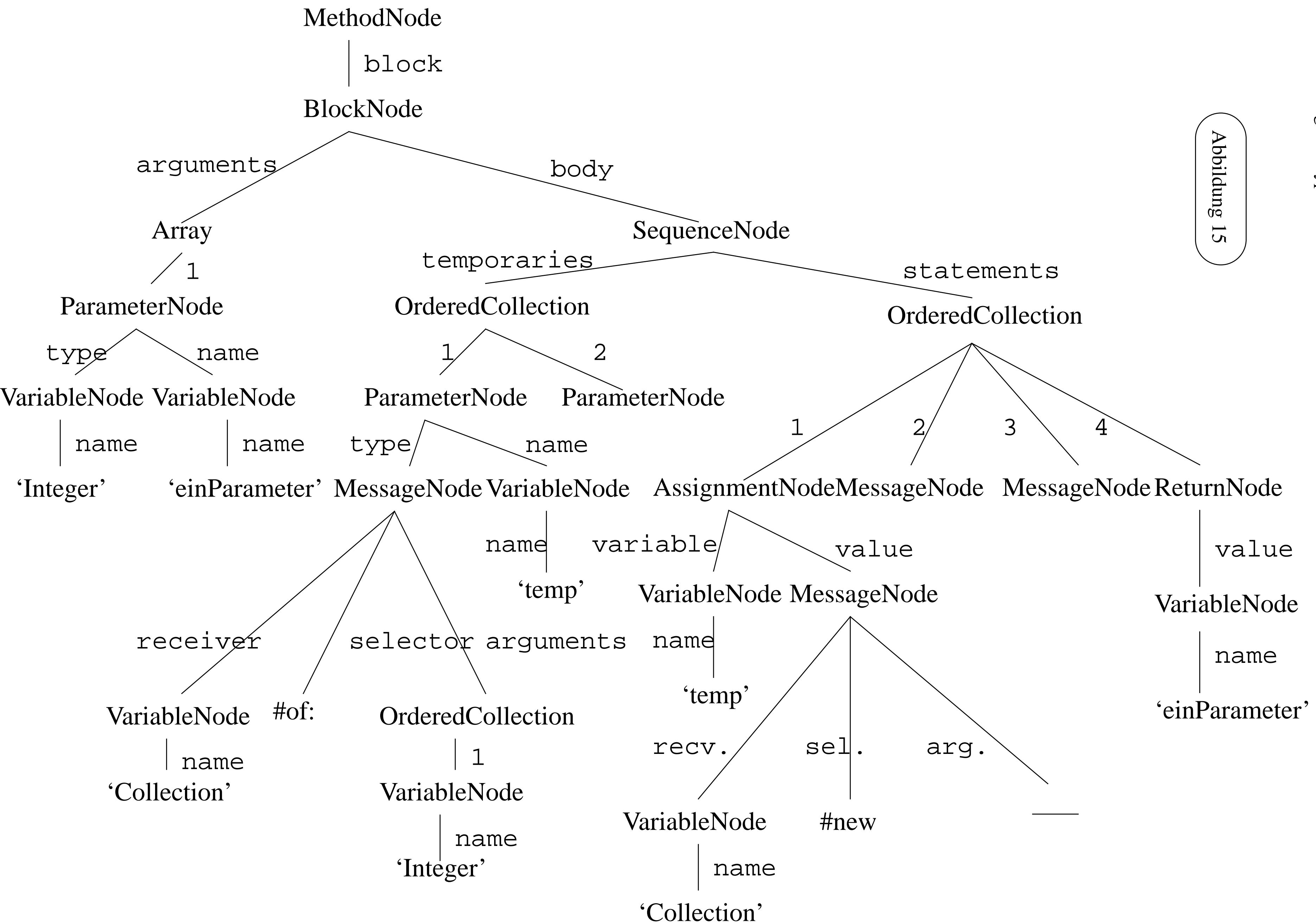
Abbildung 15
MethodNode
block
BlockNode
arguments
body
Array
1
SequenceNode
temporaries
statements
ParameterNode
OrderedCollection
OrderedCollection
type
name
1
2
VariableNode
VariableNode
ParameterNode
ParameterNode
1
2
3
4
name
name
type
name
'Integer'
'einParameter'
MessageNode
VariableNode
AssignmentNode
MessageNode
MessageNode
ReturnNode
name
variable
value
value
'temp'
VariableNode
MessageNode
VariableNode
receiver
selector
arguments
name
name
VariableNode
#of:
OrderedCollection
'temp'
'einParameter'
name
1
recv.
sel.
arg.
'Collection'
VariableNode
name
VariableNode
#new
'Integer'
name
'Collection'

C.3 Bemerkungen zu dieser Ausarbeitung

Die Texte zu dieser Studienarbeit wurde bereits kurz nach Beginn der Dokumentation in HTML (Hypertext Markup Language) weitergeführt, da dieser Formalismus durch die Möglichkeit von Hypertextlinks hierarchische Dokumentation und Verweise auf andere Arbeiten unterstützt.

Mit der zunehmenden Verfügbarkeit von elektronischen Texten etwa durch HTTP (Hypertext Transfer Protocol) sollten Verweise aus Dokumenten wie diesem zunehmend auf andere Dokumente verweisen, auf die dann ebenfalls im Volltext zugegriffen werden könnte.

Dabei zeigte es sich, daß die Arbeit mit Mosaic (einem HTTP-Client) dann nützlich war, wenn es um die Text*struktur* ging, und etwas lästig, sobald fertige Sätze formuliert werden mußten, da noch kein Editor existiert, der die interne Darstellung von Umlauten und Links beim Edieren verbirgt (einen Text zu ändern, der bereits mit HTML-Strukturinformationen ausgestattet ist, bedarf meiner Erfahrung höherer Aufmerksamkeit als normaler ISO-Text). Dieser Komfortverlust sollte allerdings mit der Verfügbarkeit von richtigen HTML-Editoren behoben sein.

Ein Problem von HTML ist es, daß bestehende (HTML-)Texte nicht so von außen referenziert werden können, daß ein Bezug zu einer bestimmten Stelle (Seite, Zeile) oder einem bestimmten Absatz (Abschnitt) genommen werden kann, was eine sehr nützliche Eigenschaft wäre.

Nach Abschluß der Dokumentation wurde diese Struktur dann in die lineare Form einer papiergebundenen Ausarbeitung überführt. Links wurden dabei entweder durch den Text, den sie verbargen (Kapitel), oder durch textuelle Verweise (Literaturstellen) ersetzt.

Literaturverweise

[BI82] A. H. Borning und D. H. H. Ingalls. A Type Declaration and Inference System for Smalltalk. In Proceedings of Conference of Principles of Programming Languages 1982, Seite 133ff

[BG93] G. Bracha, D. Griswold. Strongtalk: Typechecking Smalltalk in a Production Envi ronment. In Proceedings of the ACM Conf. on Object-Oriented Programming, Systems, Lan guages and Applications, Seite 215ff, 1993

[CCHO89] P.S. Canning, W. Cook, W. Hill, W. Olthoff. Interfaces for Strongly-Typed Object-Oriented Programming. In Proceedings of the ACM Conf. on Object-Oriented Pro gramming, Systems, Languages and Applications, Seite 457-467, 1989

[CHC90] W. R. Cook, W. Hill, P. Canning. Inheritance Is Not Subtyping. In Proc. of the ACM Symp. on Principles of Programming Languages, Seite 125-135, 1990

[GJ90] J. O. Graver, R. E. Johnson. A Type System for Smalltalk. In Proc. of the ACM Symp. on Principles of Programming Languages, Seite 136-150, 1990

[Gr89] J. O. Graver. Type-Checking and Type-Inference for Object-Oriented Programming Languages. Doktorarbeit, UIUC, 1989. (FTP://st.cs.uiuc.edu/pub/papers/graver-thesis.ps)

[GR89] A. Goldberg, D. Robson. Smalltalk-80: the language. Addison-Wesley, 1989

[Ho87] H.-J. Hoffmann. Smalltalk verstehen und anwenden. Carl Hanser Verlag, 1987

[JGZ88] R. E. Johnson, J. Graver, L. Zurawski. TS: An Optimizing Compiler for Smalltalk. In Proceedings of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications, Seite 18-25, 1988

[Joh86] R. E. Johnson. Type-Checking Smalltalk. In Proceedings of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications, 1986

[MMM90] O. Madsen, B. Magnusson, B. Moller-Pedersen. Strong Typing of Object-Ori ented Languages Revisited. In Proc. OOPSLA/ECOOP'90, ACM SIGPLAN Fith Annual Conference on Object-Oriented Programming Systems, Languages and Applications; Euro pean Conference on Object-Oriented Programming, 1990

[PS91] J. Palsberg, M. L. Schwartzbach. Object-Oriented Type Inference. In Proceedings of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications, Seite 146ff, 1991

[PS94] J.Palsberg, M. I. Schwartzbach. Object-Oriented Type Systems. John Wiley & Sons, 1994

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfaßt und nur die angegebenen Quellen benutzt zu haben.

____

(Fritz Hohl)