Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Datenbanken und Informationssysteme

Studienarbeit

# Typüberprüfung in einer polymorphen objektorientierten Programmiersprache

Analyse, Design und Implementierung eines Typprüfers für Tycoon-2

eingereicht bei:
Prof. Dr. Florian Matthes
Arbeitsbereich Softwaresysteme
TU Hamburg Harburg

vorgelegt von:
Matthias Ernst
Hagenbeckstraße 50
22527 Hamburg
Tel.: 040/4908397

Hamburg, den 5. ßuni 1998

**Zusammenfassung**

Diese Arbeit beschreibt die Implementierung eines Typüberprüfungsalgorithmus für die objektorientierte Programmiersprache TL-2, welche im Rahmen des Tycoon-2-Projektes entwickelt wurde.

TL-2 enthält ein reiches Typsystem höherer Ordnung mit Subtyp- und parametrischem Polymorphismus sowie kovarianter *Self*-Typisierung. Die Arbeit stellt das polymorphe Typsystem vor und erläutert dabei die Besonderheiten, die sich im objektorientierten Paradigma ergeben. Außerdem enthält sie in Auszügen formale Typregeln.

Weiterhin gibt die Arbeit einen Überblick über Implementierungstechniken für die Typüberprüfung, also die Abbildung von Typregeln auf Entscheidungsalgorithmen, da zu diesem Thema wenig Literatur zur Verfügung steht. Neben der Umsetzung des Typsystems findet hierbei vor allem die Realisierung eines Typinferenzalgorithmus Erwähnung.

# Inhaltsverzeichnis


**1 Ausgangspunkt der Arbeit** **3**
1.1 Ziel der Arbeit . . . . . 4

**2 Das Tycoon-2-System** **6**
2.1 Die Programmiersprache TL-2 . . . . . 6
2.2 Die Programmierumgebung . . . . . 7

**3 Typsysteme** **9**
3.1 Formalisierung von Typsystemen . . . . . 10
3.1.1 Deduktionssysteme . . . . . 10
3.1.2 Typaussagen . . . . . 11
3.2 Ein Beispiel . . . . . 11

**4 Typisierung in TL-2** **13**
4.1 Die abstrakte Syntax von TL-2 . . . . . 13
4.2 Index der Typregeln . . . . . 16
4.3 Grundlegende Konzepte . . . . . 16
4.3.1 Subtypisierung . . . . . 16
4.3.2 Subsumption . . . . . 18
4.3.3 Wert- und Typbezeichner . . . . . 18
4.3.4 Substitutionen . . . . . 19
4.3.5 Signaturen . . . . . 19
4.4 Systemvalidität . . . . . 20
4.4.1 Das initiale Environment . . . . . 21
4.4.2 System und Klassen . . . . . 21
4.5 Klassen und Objekte . . . . . 21
4.5.1 Objekttypen . . . . . 21
4.5.2 Klassen und Typen . . . . . 22
4.5.3 Metaklassen . . . . . 22
4.5.4 Methoden . . . . . 23
4.6 Subtypisierung auf Interfaces . . . . . 24
4.7 Vererbung . . . . . 25
4.7.1 Die Class Precedence List . . . . . 25

4.7.2 Typsichere Vererbung . . . . . 26
4.8 Parametrischer Polymorphismus . . . . . 28
4.8.1 Eingeschränkter parametrischer Polymorphismus . . . 28
4.8.2 Generische Klassen . . . . . 30
4.8.3 Generische Methoden . . . . . 31
4.9 Ausdrücke . . . . . 32
4.9.1 Literale . . . . . 32
4.9.2 Blöcke . . . . . 32
4.9.3 let-Ausdrücke . . . . . 33
4.9.4 Methodenaufrufe . . . . . 33
4.9.5 Funktionsabstraktionen . . . . . 33
4.9.6 Zuweisungsausdrücke . . . . . 34
4.9.7 Ausnahmen . . . . . 34
4.9.8 Zusicherungen . . . . . 36

**5 Der Typüberprüfungsalgorithmus von TL-2 37**
5.1 Implementierungstechniken . . . . . 37
5.1.1 Continuation Passing Style . . . . . 37
5.1.2 Fehlerbehandlung . . . . . 38
5.2 Der abstrakte Syntaxbaum . . . . . 39
5.2.1 Bezeichner . . . . . 39
5.2.2 Wertausdrücke . . . . . 40
5.2.3 Typausdrücke . . . . . 40
5.2.4 Substitutionen . . . . . 44
5.3 Typprüfung von Werten . . . . . 45
5.3.1 LiteralValue . . . . . 45
5.3.2 IdeValue . . . . . 46
5.3.3 BlockValue . . . . . 46
5.3.4 LetValue . . . . . 46
5.3.5 SendValue . . . . . 46
5.3.6 CompiledFun . . . . . 47
5.3.7 AssignValue . . . . . 47
5.3.8 Kontrollstrukturen . . . . . 47
5.3.9 AssertValue . . . . . 48
5.4 Typinferenz . . . . . 48
5.4.1 Unifikation . . . . . 49

**6 Zusammenfassung und Ausblick 52**

# Kapitel 1

# Ausgangspunkt der Arbeit

Das Tycoon-2-System ist als Nachfolger des Tycoon[1] Systems entstanden, welches am Arbeitsbereich Datenbanken und Informationssysteme im Rahmen des europäischen FIDE-Projektes entwickelt wurde [Matthes et al. 97]. Beide Systeme betonen die Unterstützung für die Entwicklung langlebiger und datenintensiver Anwendungen sowie die Integration unabhängig voneinander entwickelter externer Dienste in einem verbesserten sprachlichen und architektonischen Rahmen. Dies wird erreicht durch die Bereitstellung höherer sprachlichen Konzepte wie Polymorphismus, Funktionen höherer Ordnung, Massendatenstrukturen und Iterations- und Persistenzabstraktion.

Gegenüber Tycoon, welches mit TL eine funktional-imperative polymorphe Programmiersprache zur Verfügung stellt [Matthes, Schmidt 92], realisiert Tycoon-2 das objekt-orientierte Paradigma, das sich im Hinblick auf Wiederverwendung und Erweiterbarkeit überlegen für die Konstruktion großer Softwaresysteme zeigt.

Tycoon-2 zeichnet sich durch folgende Systemeigenschaften aus:

- Virtuelle Maschine: Tycoon-2-Programme werden in maschinenunabhängigen Bytecode übersetzt, der von einer virtuellen Maschine ausgeführt wird.
- Orthogonale Persistenz: das gesamte Tycoon-2-System läuft innerhalb eines Objektspeichers ab, der Code, Daten sowie laufende Berechnungen (Threads) unabhängig von deren Definition über mehrere Programmabläufe persistent speichern und wiederherstellen kann.
- Uniforme Datenrepräsentation: alle Daten im System sind uniform als markierte Datenstrukturen repräsentiert. Dies erweist sich für Systemkomponenten wie den Bytecodeinterpreter und den Garbagecollector als vorteilhaft.

[1] TYped Communicating Objects in Open eNvironments

- Reflexion: bis auf den Kern des Systems, d.h. virtuelle Maschine und Objektspeicher, sind alle Komponenten des Laufzeitsystems (z.B. der Compiler) als Tycoon-2-Objekte realisiert und langlebig im Objektspeicher abgelegt und damit von anderen Programmen als typisierte Objekte nutzbar.

- Multithreading: Tycoon-2 erlaubt die effiziente nebenläufige Ausführung und Synchronisation mehrerer Berechnungen. Die einzelnen Ausführungspfade werden dabei direkt auf Betriebssytem-Threads abgebildet.

- Portabilität des Systems: Aufgrund des kleinen Systemkerns ist Tycoon-2 sehr einfach auf POSIX-konforme Systeme portierbar und läuft bisher auf mehreren wichtigen UNIX-Plattformen (Solaris, HP-UX, Linux).

- Portabilität von Objektspeichern: Objektspeicher mit Daten, Code und aktiven Threads können unabhängig von der Architektur, auf der sie entstanden sind, auf jeder von Tycoon-2 unterstützten Plattform wiederaufsetzen.

Eine erste prototypische Implementierung wurde auf Basis des Vorgängersystems Tycoon vorgenommen. In dieser Version waren die Laufzeit- und Programmierumgebung einschließlich eines Typprüfers in der Sprache TL implementiert und TL-2 Code wurde in TL-Bytecode des "alten" Bytecodeinterpreters übersetzt. Nach der optimierten Neuimplementierung des Bytecodeinterpreters sowie der Laufzeit- und Entwicklungsumgebung in TL-2 (mit Ausnahme des Typprüfers) entstand das neue System durch einen Bootstrap [Wienberg 97].

## 1.1 Ziel der Arbeit

Aufgabe der vorliegenden Arbeit ist, einen Typprüfer für TL-2 zu entwickeln und als eine weitere Komponente in die Programmierumgebung zu integrieren. Sie wird wie die anderen Komponenten in TL-2 implementiert und arbeitet auf dem bereits gescopten Syntaxbaum, setzt also die lexikalische und die Syntaxanalyse sowie die Scopingphase voraus.

Ein wichtiges Ziel ist dabei, möglichst kleine Quelltexteinheiten separat voneinander prüfen zu können, sowie bei Änderungen nur die betroffenen Teile des Systems neu zu prüfen. Insbesondere soll für parametrisierte Klassen nur genau einmal deren Definition typgeprüft werden, nicht jede Instantiierung mit Aktualparametern, wie dies z.B. bei C++ Templates [Stroustrup 97] geschieht. Außerdem soll eine typsichere Vererbung für Methoden sichergestellt werden, ohne daß ererbte Methoden neu geprüft werden müssen.

Die prototypische Implementierung des Typüberprüfers weist zudem Schwächen in der Typinferenz auf, die durch die Neuimplementierung ausgeräumt werden sollen.

Als statische Analyse von Programmen soll die Typüberprüfung dem Entwickler eine wichtiges Werkzeug in der Softwareentwicklung geben, das es erlaubt, mögliche Programmierfehler noch vor der Laufzeit zu erkennen und zu beheben.

Ein interessanter Aspekt der Arbeit ist außerdem, inwieweit die Anwendung objektorientierter Techniken die Implementierung des Typprüfers gegenüber der existierenden funktional-imperativen Version in TL vereinfacht und übersichtlicher macht.

# Kapitel 2

# Das Tycoon-2-System

Dieses Kapitel bietet einen Überblick über das Tycoon-2-System. Es beschreibt die Programmiersprache TL-2 sowie die Programmierumgebung. Einzelne Konstrukte der Programmiersprache werden hierbei bewußt nicht erläutert, dies geschieht im Zusammenhang mit der Beschreibung ihrer Typisierung in Kapitel 4.

## 2.1 Die Programmiersprache TL-2

TL-2 ist eine objektorientierte Programmiersprache, das heißt, Objekte und Nachrichten zwischen ihnen bilden das Programmierparadigma. Für eine grundlegende Einführung in die programmiersprachlichen Aspekte von Objektorientierung siehe [Meyer 97].

Jedes Objekt in TL-2 ist Exemplar einer Klasse. In Klassen sind Methoden und Variablen definiert; diese bestimmen Verhalten und Zustand der Objekte. Klassen sind also Abstraktionen gleichartiger Objekte. Klassen können voneinander erben, die erbende Klasse wird dabei als Subklasse der beerbten Klassen bezeichnet; diese selbst heißen Superklassen. Durch Vererbung übernimmt die Subklasse Methoden- und Variablendefinitionen von ihren Superklassen.

Klassen in TL-2 sind selbst Objekte erster Klasse, d.h. sie können an Wertvariablen gebunden werden, und an ihnen können Methoden gerufen werden. Daher sind auch sie Exemplare einer Klasse, ihrer *Metaklasse*, die bei der Definition der Klasse angegeben wird. In ihnen werden vor allem Erzeugungsmethoden definiert, die ein neues Exemplar der Klasse liefern, sogenannte *Konstruktoren*. Diese nehmen also keinen Sonderstatus wie in anderen objektorientierten Sprachen ein.

Aus dem funktionalen Programmierparadigma übernimmt TL-2 das Konzept von Funktionen als Werte erster Klasse. Hierbei wird ein unter Umständen parametrisiertes Programmstück inklusive seiner Variablenbindungen zur späteren Ausführung verpackt und kann als Wert an Variablen

gebunden werden. Mit solchen Funktionsabschlüssen (*Closures*) lassen sich z.B. elegant Iterationsabstraktionen und andere Kontrollstrukturen realisieren. Funktionen werden in TL-2 als Objekte repräsentiert. Ihre Klasse definiert eine entsprechend parametrisierte Methode (`[]`, *"apply"*), mit der die Funktion ausgewertet werden kann.

## 2.2 Die Programmierumgebung

Die Komponenten der TL-2 Programmierumgebung sind selbst in TL-2 implementiert und als typisierte Objekte im Objektspeicher abgelegt. Im einzelnen sind dies:

- Der *Klassenlader* verwaltet die Abhängigkeiten zwischen Klassenobjekten und den Quelltextdateien, die deren Klassendefinitionen enthalten. Ändert sich eine dieser Dateien, so kann der Klassenlader automatisch eine Neuübersetzung anstoßen.

- *Lexikalische und Syntaxanalyse* sind mit Hilfe eines flexiblen Toolkits zur Scanner- und Parsergenerierung implementiert. Der Scanner erzeugt aus der Eingabe einen Strom von Tokens, der an den Parser weitergereicht wird. Dieser baut daraus den entsprechenden abstrakten Syntaxbaum auf. Die Struktur des Syntaxbaumes wird bei der Beschreibung der Implementierung des Typprüfers in 5 näher erläutert.

- Eine separate *Scopingkomponente* löst Vorkommen von lokalen Variablen sowie von Typbezeichnern gegen deren jeweilige Deklaration auf. Dies geschieht, indem im Syntaxbaum die Verwendungen der Bezeichner durch direkte Objektreferenzen auf die der Bezeichnerdeklaration entsprechenden Objekte erweitert werden. Somit werden mehrdeutige Bezeichner aufgelöst und den späteren Phasen (Kompilation und Typüberprüfung) ein direkter Zugriff auf die Deklaration ohne Verwendung einer Symboltabelle oder von de-Bruijn-Indizes [de Bruijn 72] ermöglicht.

- Der *TL-2 Compiler* übersetzt den abstrakten Syntaxbaum in ausführbaren Bytecode, erzeugt ein Klassenobjekt (ein Exemplar der Metaklasse) und macht dieses im System bekannt. Im Gegensatz zu anderen kompilierten Sprachen wie C++ oder Java wird also kein Objektcode zur späteren Verwendung in Programmen im Dateisystem abgelegt, sondern dieser wird direkt im laufenden System wirksam.

- Mit Hilfe einer terminal-orientierten Eingabe, dem *Toplevel*, können TL-2 Ausdrücke interaktiv ausgewertet werden. Diese benutzt reflektiv den Compiler des laufenden Systems, um die Eingabe übersetzen, auswerten und binden zu können.

Da sämtliche Daten im Objektspeicher uniform repräsentiert sind, sind keine Typinformationen für die Übersetzung und Ausführung erforderlich, das System ist ohne Typüberprüfung bereits lauffähig. Durch die Typprüfung werden lediglich Programme als fehlerhaft angemahnt, bei denen potentiell folgende Fehler auftreten können:

- Aufruf einer nicht bekannten Methode. Diese Fehler manifestieren sich bei der Ausführung durch einen Laufzeitfehler ("message not understood"). Dies schließt insbesondere einen Aufruf mit der falschen Anzahl von Parametern ein.

- Aufruf einer nicht öffentlichen Methode. Der Bytecodeinterpreter führt einen solchen Aufruf ohne Hinsicht auf den Aufrufer immer aus (eine für die interaktive Fehlersuche sehr nützliche Eigenschaft !).

Ohne Einsatz der Typprüfung weist das Tycoon-2-System unter diesem Aspekt starke Ähnlichkeit mit Smalltalk-Systemen [Goldberg, Robson 89] auf.

Eine ausführliche Beschreibung der Programmierumgebung findet sich in [Wienberg 97].

# Kapitel 3

# Typsysteme

Die überwiegende Zahl von Programmiersprachen beinhaltet das Konzept des *Typs*. Die gängige mengentheoretische Interpretation identifiziert Typen mit Mengen von Werten [Scott 72]. Ist ein Wert $v$ Element eines Typs $A$, so sagt man auch "$v$ hat den Typ $T$", oder in Formelschreibweise: $v : A$.

In einer *statisch typisierten* Programmiersprache kann durch ein Regelwerk, das Typsystem, jedem wohlgeformten Wertausdruck statisch durch eine textuelle Analyse ein Typ zugeordnet werden. Korrekter Entwurf des Typsystems und der Auswertungssemantik garantieren dann, daß der Ausdruck zur Laufzeit tatsächlich zu einem Wert dieses Typs evaluiert. Somit kann ein Laufzeitfehler (wie z.B. "message not understood") aufgrund von Operationen auf inkompatiblen Werten bereits statisch ausgeschlossen werden.

Schlägt die Typzuordnung fehl, so kann in dem gegebenen Typsystem nicht bewiesen werden, daß ein Ausdruck zu einem Wert eines bestimmten Typs evaluiert, das Programm ist *nicht typkorrekt*. Das bedeutet nicht zwingend, daß ein Programmfehler vorliegen muß, z.B. kann die benötigte Typinformation nur nicht in dem Typsystem ausgedrückt werden.

Beispielhaft hierfür sind nicht-generische Behälterklassen, wie sie z.B. in der Programmiersprache Java[Gosling et al. 96] zur Verfügung stehen: Exemplare der Klasse `Vector` erlauben es, Elemente des Typs `Object` in einer indizierten Liste zu speichern. Folgerichtig bietet eine solche Liste Operationen zum Speichern von Objekten (`addElement (Object)`) und zum Wiederauffinden (`Object elementAt (int)`) an. Dies hat jedoch folgende Implikation: obwohl der Programmierer weiß (und tatsächlich jeder mögliche Programmablauf dazu führt), daß in einer Liste nur Objekte eines speziellen Typs $A$ gespeichert werden, kann das Typsystem für ein Element, das wieder aus dieser Liste entnommen wird, nur den allgemeinen Typ `Object` berechnen. Der Programmierer muß sich dann mit unsicheren Mitteln behelfen und unterwandert damit die statische Prüfung. In einem Typsystem zweiter Ordnung, d.h. mit generischen Typen, wäre diese Information statisch

ausdrückbar.

Eine Menge entspricht immer einem Prädikat bzw. einer Aussage über die Zugehörigkeit eines Wertes zu dieser Menge. Daher ist Typprüfung eine Form der Beweisführung über Programme, oder der Programmverifikation. Tatsächlich bestehen zwischen Typsystemen und Logiken enge Verwandtschaften (Curry-Howard-Isomorphismus [Howard 80]).

Mit den Eigenschaften von Logiken geht auch eine unerwünschte einher: Nichtentscheidbarkeit. Aus diesem Grund ist es für einen Typüberprüfungsalgorithmus beispielsweise nicht möglich sicherzustellen, daß Array-Zugriffe nur auf erlaubte Indices erfolgen, obwohl diese Eigenschaft auf Typebene (z.B. durch einen Intervalltyp *[0..n]*) durchaus ausgedrückt werden könnte.

Die Herausforderung beim Entwurf eines Typsystems für eine Programmiersprache besteht zu einem großen Teil also darin, dem Programmierer so viel Ausdrucksmächtigkeit wie möglich zur Verfügung zu stellen, die Typprüfung jedoch als ein entscheidbares und effizient lösbares Problem zu gestalten.

## 3.1 Formalisierung von Typsystemen

### 3.1.1 Deduktionssysteme

Die Verwandtschaft von Typsystemen und Logiken legt nahe, ähnliche formale Beschreibungstechniken zu nutzen. Daher formuliert man Typsysteme durch *Deduktionssysteme* über Typaussagen.

Eine Deduktionsregel hat die Form

$$\boxed{\frac{P_1 \ldots P_n}{C}\ Name}$$

wobei die $P_i$ *Prämissen* und $C$ *Konklusion* genannt werden. Für Deduktionsregeln lassen sich zwei Lesarten finden:

**vorwärts** (also von oben nach unten) in der Bedeutung, daß aus der Ableitbarkeit der Prämissen die Konklusion folgt, oder

**rückwärts** als "um die Konklusion zu beweisen, müssen folgende Prämissen bewiesen werden."

Die zweite Lesart läßt sich direkt als abstrakter Algorithmus für einen automatischen Beweiser verstehen: man setze die zu beweisende Aussage als Konklusion einer Deduktionsregel ein und beweise rekursiv alle entsprechenden Prämissen, bis man bei Axiomen angelangt (ein Axiom ist eine Deduktionsregel ohne Prämissen).

### 3.1.2 Typaussagen

Prämissen und Konklusion treffen Aussagen (*judgements*) über Typen, Werte und deren Beziehungen. Diese Aussagen beruhen zumeist auf einem Kontext, dem *statischen Environment*, das die Zuordnung von freien Variablen übernimmt. Im folgenden sei das Environment durch den Buchstaben $\Gamma$ gekennzeichnet. Typaussagen haben daher die Form $\Gamma \vdash A$, gelesen als "im Environment $\Gamma$ gilt $A$".

In einem polymorphen Typsystem werden sowohl Werten Typen zugeordnet als auch Aussagen über Beziehungen zwischen Typen gemacht. Daher sind sind drei wichtige Formen von Deduktionsregeln zu unterscheiden:

- Wohlgeformte Typen. Diese Regeln bestimmen die Menge der legalen Typen. Sie sind von der Form

  $$\boxed{\frac{\cdots}{\Gamma \vdash A}\ \mathit{Type\ xx}}$$

  ("$A$ ist ein wohlgeformter Typ.")

- Wert-Typ-Zuordnungen. Diese Regeln weisen Werten Typen zu:

  $$\boxed{\frac{\cdots}{\Gamma \vdash a : A}\ \mathit{Val\ xx}}$$

  ("$a$ hat den Typ $A$.")

- Subtypisierung. Diese Regeln bestimmen die Subtyprelation '<:' auf Typen:

  $$\boxed{\frac{\cdots}{\Gamma \vdash A <: B}\ \mathit{Subtype\ xx}}$$

  ("$A$ ist Subtyp von $B$.")

## 3.2 Ein Beispiel

Gegeben sei eine einfache Programmiersprache über ganze Zahlen mit der Konstanten `0` und der Nachfolgerfunktion `succ`. Außerdem können Bezeichner eingeführt werden durch: `let` *`ide`* `=` *`value`* `in` *`value`*. Als einziger Typ lassen sich die natürlichen Zahlen $Nat$ identifizieren.

Das Typsystem sieht dann wie folgt aus:

**Die Zahl 0 hat den Typ** $Nat$.

$$\boxed{\frac{}{\Gamma \vdash \mathtt{0} : Nat}\ \mathit{Val\ zero}}$$

**Der Nachfolger jeder natürlichen Zahl ist eine natürliche Zahl.**

$$\boxed{\frac{\Gamma \vdash a : Nat}{\Gamma \vdash \texttt{succ}(a) : Nat}\ \mathit{Val\ succ}}$$

**Lokale Definition eines Bezeichners innerhalb eines Ausdrucks.**

$$\boxed{\frac{\Gamma \vdash e1 : A \quad \Gamma, x : A \vdash e2 : B}{\Gamma \vdash \texttt{let}\ v = e1\ \texttt{in}\ e2 : B}\ \mathit{Val\ let}}$$

Man beachte, wie die Sichtbarkeit des Bezeichners durch eine lokale Erweiterung des Environments $(\Gamma, x : A)$ ausgedrückt wird.

**Lookup im Environment.**

$$\boxed{\frac{}{..., x : A, \Gamma \vdash x : A}\ (x \notin \mathit{domain}(\Gamma))\quad \mathit{Val\ ide}}$$

Durch die Bedingung wird ausgedrückt, daß später definierte Bezeichner früher definierte überschatten.

Der Typ von "zwei" (`let eins = succ(0) in succ(eins)`) kann dann wie folgt abgeleitet werden:

$$\frac{\dfrac{\dfrac{}{\Gamma \vdash 0 : Nat}\ \mathit{Val\ zero}}{\Gamma \vdash succ(0) : Nat}\ \mathit{Val\ succ} \qquad \dfrac{\dfrac{}{\Gamma, eins : Nat \vdash eins : Nat}\ \mathit{Val\ ide}}{\Gamma, eins : Nat \vdash succ(eins) : Nat}\ \mathit{Val\ succ}}{\Gamma \vdash \texttt{let}\ eins = succ(0)\ \texttt{in}\ succ(eins) : Nat}\ \mathit{Val\ let}$$

# Kapitel 4

# Typisierung in TL-2

Im folgenden werde ich die abstrakte Syntax von TL-2 darstellen sowie typrelevante Sprachkonzepte identifizieren. Ich werde ihren Einfluß auf die Typprüfung erläutern und entsprechende Typregeln aufstellen.

Die Typregelen habe ich dabei möglichst so ausgewählt, wie sie der operationalen Semantik des Typprüfers entsprechen. Zum Beispiel findet sich im Typprüfer keine direkte Entsprechung der Transitivitätsregel der Subtypisierung, da diese rückwärts gelesen zu einem nichtdeterministischen Algorithmus führt. Sind solche Regeln der Vollständigkeit halber trotzdem aufgeführt, so sind sie mit einem entsprechenden Hinweis versehen.

Ich verwende folgende Metavariablen:

| **Variable** | **Bedeutung** |
|---|---|
| $a, b, c, ...$ | Wertausdruck |
| $t, u...$ | Wertbezeichner |
| $A, B, C, ...$ | Typausdruck |
| $T, U, ...$ | Typbezeichner |
| $S$ | Signatur |
| $m$ | Methodenselektor |
| | |
| $\overrightarrow{\mathcal{A}}$ | Liste von $\mathcal{A}$, d.h. |
| $[]$ | leere Liste oder |
| $\mathcal{A} : \overrightarrow{\mathcal{B}}$ | Element $\mathcal{A}$ vor Liste $\overrightarrow{\mathcal{B}}$ |
| $[\mathcal{A}, \mathcal{B}, \mathcal{C}]$ | Liste mit drei Elementen $= \mathcal{A} : \mathcal{B} : \mathcal{C} : []$ |

## 4.1 Die abstrakte Syntax von TL-2

### System

*System*: $\overrightarrow{Klasse}$ (Wurzelproduktion)

## Klassendefinitionen

| | | |
|---|---|---|
| *Klasse*: | class $X(\overrightarrow{Typsignatur})$ | (Klassenbezeichner und Parameter) |
| | super $\overrightarrow{Superklasse}$ | (Superklassen) |
| | meta *Metaklasse* | (Metaklasse) |
| | Self *Self-Signatur* | (Self-Bound) |
| | variables $\overrightarrow{Variable}$ | (Exemplarvariablen) |
| | methods $\overrightarrow{Methode}$ | (Methoden) |
| *Superklasse*: | $X[(\overrightarrow{Typ})]$ | |
| *Metaklasse*: | $X[(\overrightarrow{Typ})]$ | |
| *Self-Signatur*: | Self <: *Typ* \| | (Signatur des Bezeichners `Self`) |
| | Self = *Typ* | |
| *Variable*: | *Sichtbarkeit Wertsignatur* | |
| *Methode*: | *Methodensignatur Methodenkörper* | |
| *Methodensignatur*: | *Sichtbarkeit* $m(\overrightarrow{Signatur})$ : *Typ* | |
| *Sichtbarkeit*: | public \| private | |
| *Methodenkörper*: | deferred \| *Block* | |
| *Block*: | $\overrightarrow{Ausdruck}$ | |

## Ausdrücke

| | | |
|---|---|---|
| *Ausdruck*: | *x* \| | |
| | *Literal* \| | (Integer, Real, Character, String, ...) |
| | *Nachricht* \| | (Methodenaufruf) |
| | *Funktion* \| | (Funktionsabschluß) |
| | *Let* \| | (lokale Bezeichnerdeklaration) |
| | *x* := *Ausdruck* \| | (lokale Zuweisung) |
| | assert *Ausdruck* \| | (Zusicherung) |
| | try *Block Funktion* | (Ausnahmebehandlung) |
| *Nachricht*: | $Ausdruck.m(\overrightarrow{Parameter})$ | |
| *Parameter*: | *Ausdruck* \| :*Typ* | |
| *Funktion*: | fun$(\overrightarrow{Signatur})$ :*Typ Block* | |
| *Let*: | let *Wertsignatur* = *Ausdruck* in *Ausdruck* | |

## Typen und Typoperatoren

| | | |
|---|---|---|
| *Typ*: | $X$ \| | (Typvariable) |
| | $\sigma[\overrightarrow{Methodensignatur}]$ | (Interface) |
| | $\mathrm{Fun}(\overrightarrow{Signatur}) : Typ$ \| | (Funktionstyp) |
| | *Typapplikation* \| | |
| | Void \| | |
| | Nil \| | |
| | *Typoperator* | |
| *Typapplikation*: | $Typ(\overrightarrow{Typ})$ | |
| *Typoperator*: | $\Lambda(\overrightarrow{Typsignatur}).Typ$ | |

## Signaturen

| | |
|---|---|
| *Signatur*: | *Wertsignatur* \| *Typsignatur* |
| *Wertsignatur*: | $x : Typ$ |
| *Typsignatur*: | $X <: Typ$ \| |
| | $X = Typ$ |

## 4.2 Index der Typregeln

| *Beschreibung* | *Name* | *Abschnitt* | *Seite* |
|---|---|---|---|
| Allgemeine Typregeln | Subtype Refl | 4.3.1 | 16 |
| | Subtype Trans | 4.3.1 | 17 |
| | Val Subsumption | 4.3.2 | 18 |
| Systemvalidität | System valid | 4.4.2 | 21 |
| Der Typ *Void* | Type Void | 4.3.1 | 17 |
| | Subtype Void | 4.3.1 | 17 |
| Der Typ *Nil* | Type Nil | 4.3.1 | 17 |
| | Subtype Nil | 4.3.1 | 17 |
| Bezeichner und Environment | Val Ide | 4.3.3 | 18 |
| | Type Ide | 4.3.3 | 19 |
| | Subtype Ide | 4.3.3 | 19 |
| Signaturen | Subsig Empty | 4.3.5 | 19 |
| | Subsig Value | 4.3.5 | 19 |
| | Subsig Type | 4.3.5 | 19 |
| | Sig Param | 4.3.5 | 20 |
| Interfaces | Subsig Method | 4.6 | 24 |
| | Subtype Interface | 4.6 | 24 |
| CPL | Valid CPL | 4.7.2 | 27 |
| Parametrischer Polymorphismus | Type apply | 4.8.2 | 30 |
| | Sub Oper | 4.8.2 | 31 |
| Wertausdrücke | Val Block | 4.9.2 | 32 |
| | Val let | 4.9.3 | 33 |
| | Val send | 4.9.4 | 33 |
| | Val fun | 4.9.5 | 34 |
| | Val assign | 4.9.6 | 34 |
| | Val assert | 4.9.8 | 36 |

## 4.3 Grundlegende Konzepte

### 4.3.1 Subtypisierung

Wie die Einführung in die Typregeln bereits nahelegt, existiert zwischen Typen in TL-2 eine Subtyprelation, notiert durch das Relationszeichen $<:$. Die Subtyprelation bildet eine Halbordnung, d.h. sie ist reflexiv und transitiv. Weiterhin hat sie ein größtes Element, den Top-Typ *Void*, und ein kleinstes, den Bottom-Typ *Nil*.

**Reflexivität**:

$$\boxed{\frac{\Gamma \vdash A}{\Gamma \vdash A <: A}\ \textit{Subtype Refl}}$$

Reflexivität wird in der Implementierung nur insofern ausgenutzt, daß Typobjekte auf Identität verglichen werden, d.h. nicht alle Ausprägungen

dieser Regel werden angewandt. Diese Regel wird jedoch durch die anderen impliziert, so daß der Algorithmus weiterhin vollständig ist.

**Transitivität**:

$$\frac{\Gamma \vdash A <: B \quad \Gamma \vdash B <: C}{\Gamma \vdash A <: C}\ \mathit{Subtype\ Trans}$$

Zur Transitivitätsregel gibt es keine Entsprechung im Typüberprüfungsalgorithmus. Sie wird nur für den speziellen Fall von Typbezeichnern angewandt (siehe 4.3.3).

**Void**:

$$\frac{}{\Gamma \vdash \mathit{Void}}\ \mathit{Type\ Void}$$

$$\frac{}{\Gamma \vdash \sigma[...] <: \mathit{Void}}\ \mathit{Subtype\ Void}$$

Tatsächlich kann *Void* behandelt werden wie das leere Interface $\sigma[]$.

**Nil**:

$$\frac{}{\Gamma \vdash \mathit{Nil}}\ \mathit{Type\ Nil}$$

$$\frac{\Gamma \vdash A <: \mathit{Void}}{\Gamma \vdash \mathit{Nil} <: A}\ \mathit{Subtype\ Nil}$$

### Rekursive Subtypisierung

Durch (u.U. indirekten) Selbstbezug in Typen kann es ohne geeignete Maßnahmen zu unendlicher Rekursion im Ableitungsbaum bzw. im Algorithmus kommen. Man betrachte beispielsweise die folgende Klassendefinition:

```
class List(E <: Object)
{
  ...
  tail :List(E) { ... }
  ...
}
```

Ein Subtyptest $List(A) <: List(B)$ müsste die Ergebnistypen der Methode `tail` auf Subtyprelation testen und geriete somit in eine nicht terminierende Rekursion. Aus diesem Grund wird eine zu beweisende Beziehung in ein zweites *Environment bewiesener bzw. als wahr angenommener Subtypbeziehungen* $\Sigma$ aufgenommen. Der tatsächliche Subtyptest wird dann mit diesem erweiterten Environment durchgeführt, der Algorithmus terminiert daher auch bei rekursiven Typen.

Folglich lauten alle Subtypregeln mit dem zweiten Environment $\Sigma$:

$$\frac{\Sigma, A <: B \quad \Gamma \vdash ...}{\Sigma \qquad\qquad \Gamma \vdash A <: B}\ \mathit{Subtype\ XXX}$$

Der Einfachheit halber sind die Regeln jedoch ohne $\Sigma$ formuliert, der Leser möge die Erweiterung in Gedanken nachvollziehen.

$\Sigma$ wird vor jedem Subtyptest konsultiert:

$$\frac{(A <: B) \in \Sigma}{\Sigma\ \Gamma \vdash A <: B}\ \textit{Subtype Sigma}$$

### 4.3.2 Subsumption

Eines der zentralen Konzepte polymorpher, dabei vorrangig objektorientierter Sprachen ist die *Subsumption*, d.h. die Möglichkeit, ein Objekt eines Typs $A$ dort einzusetzen, wo ein Objekt eines anderen Typs $B$ erwartet wird. Damit diese Ersetzung keinen späteren Laufzeitfehler verursacht, muß das ersetzende Objekt mindestens alle Nachrichten verstehen, die der erwartete Objekttyp spezifiziert. Dies ist genau dann der Fall, wenn $A <: B$. Aus diesem Grund ist das eingesetzte Objekt auch Element des erwarteten Typs $B$: ein Objekt ist Element aller Typen, die Supertypen seines Erzeugungstyps sind.

Die Subsumptionsbeziehung wird durch folgende Regel ausgedrückt:

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A <: B}{\Gamma \vdash a : B}\ \textit{Val Subsumption}$$

Auch die Subsumptionsregel führt zu einem nichtdeterministischem Algorithmus und ist daher nicht direkt im Typprüfungsalgorithmus implementiert. Statt dessen sind alle betroffenen Regeln entsprechend formuliert, um Subsumption einzubeziehen (*Val let* (4.9.3), *Val send*(4.9.4) in Verbindung mit *Subsig Value*(4.3.5), *Val fun*(4.9.5), *Val assign*(4.9.6)).

In der Mengeninterpretation entspricht Subtypisierung genau der Teilmengenrelation. Subsumption ist somit äquivalent zu der folgenden Implikation in der Mengenlehre:

$$a \in A \wedge A \subseteq B \Rightarrow a \in B$$

### 4.3.3 Wert- und Typbezeichner

Bezeichner dienen als Platzhalter für Werte und Typen, die an sie gebunden werden können. In einem gegebenem statischen Kontext hat ein Wertbezeichner immer den gleichen, deklarierten Typ; ein Typbezeichner ist, je nach Deklaration, entweder gleich oder Subtyp eines anderen Typs. Typgleichheit wird interpretiert als wechselseitige Subtypbeziehung ( $X = T \Leftrightarrow X <: T \wedge T <: X$ ).

Diese Typinformation finden sich im statischen Environment wieder; die folgenden Regeln spezifizieren den Zugriff auf dieses Environment:

$$\frac{}{\ldots, x : A, \ldots \vdash x : A}\ \textit{Val Ide}$$

$$\frac{}{\ldots, T <: A, \ldots \vdash T} \; \textit{Type Ide}$$

$$\frac{}{\ldots, T = A, \ldots \vdash T} \; \textit{Type Ide 2}$$

$$\frac{\ldots, T <: A, \ldots \vdash A <: B}{\ldots, T <: A, \ldots \vdash T <: B} \; \textit{Subtype Ide}$$

$$\frac{\ldots, T = A, \ldots \vdash A <: B}{\ldots, T = A, \ldots \vdash T <: B} \; \textit{Subtype Ide 2}$$

$$\frac{\ldots, T = A, \ldots \vdash B <: A}{\ldots, T = A, \ldots \vdash B <: T} \; \textit{Subtype Ide 3}$$

Die drei Subtypregeln sind dabei Sonderfälle der Transitivitätsregel.

### 4.3.4 Substitutionen

Unter Substitutionen versteht man die Ersetzung freier Typbezeichner durch andere Typen innerhalb von Typausdrücken oder Signaturen.

$$A[B/X]$$

bezeichnet dabei den Typausdruck, der entsteht, wenn alle freien Vorkommen des Bezeichners $X$ in $A$ durch $B$ ersetzt werden. $List(T)[Int/T]$ ist also gleich $List(Int)$.

Substitutionen werden in dieser Arbeit benutzt, um das Einsetzen aktueller für formale Typparameter auszudrücken, sowie bei der Behandlung des Bezeichners *Self*.

### 4.3.5 Signaturen

Eine *Signatur* $S$ ist eine Deklaration eines Bezeichners. Eine Signatur hat die Form $x : A$, $T <: A$ oder $T = A$. Sie ist für spätere Typregeln für Parameterlisten und -übergabe sowie für Methodenüberschreibung von Bedeutung und wird daher jetzt eingeführt. Entsprechend definiert man Signaturlisten $\overrightarrow{S}$. Basierend auf Subtypbeziehungen wird eine Subsignaturbeziehung auf Signaturlisten definiert, repräsentiert durch das Relationszeichen $<::$.

$$\frac{}{\Gamma \vdash [] <:: []} \; \textit{Subsig Empty}$$

$$\frac{\Gamma \vdash A <: A' \quad \Gamma \vdash \overrightarrow{S} <:: \overrightarrow{S}'}{\Gamma \vdash (x : A) : \overrightarrow{S} <:: (x' : A') : \overrightarrow{S}'} \; \textit{Subsig Value}$$

$$\frac{\Gamma \vdash A <: A' \quad \Gamma, T <: A \vdash \overrightarrow{S} <:: \overrightarrow{S}'[T/T']}{\Gamma \vdash (T <: A) : \overrightarrow{S} <:: (T' <: A') : \overrightarrow{S}'} \; \textit{Subsig Type}$$

$$\frac{\Gamma \vdash A <: A' \quad \Gamma, T = A \vdash \overrightarrow{S} <:: \overrightarrow{S}'[T/T']}{\Gamma \vdash (T = A) : \overrightarrow{S} <:: (T' <: A') : \overrightarrow{S}'} \textit{ Subsig Type 2}$$

$$\frac{\Gamma \vdash A <: A' \quad \Gamma \vdash A' <: A \quad \Gamma, T = A \vdash \overrightarrow{S} <:: \overrightarrow{S}'[T/T']}{\Gamma \vdash (T = A) : \overrightarrow{S} <:: (T' = A') : \overrightarrow{S}'} \textit{ Subsig Type 3}$$

Man beachte, daß die Namen der Bezeichner in diesen Regeln nicht relevant sind. Praktisch ermöglicht dies, beim Überschreiben von Methoden in Subklassen Parameternamen zu verändern. Benannte (oder *keyword*) Parameter würden andere Regeln erfordern.

Die Substitutionen in den Regeln *Subsig Type* sind erforderlich, da sich Typen im Rest der Signaturliste $\overrightarrow{S}'$ auf den Typbezeichner $T'$ beziehen können.

Zum Beispiel gilt:

$$[(U = String), (u : String)] <:: [(T <: Object), (t : T)]$$

(unter der Annahme $String <: Object$)

Um nun bei Typoperatorapplikationen (4.8.2) oder Methodenaufrufen (4.9.4) Aktualparameter gegen Formalparameter zu testen, wird aus den Aktualparametern $\overrightarrow{p}$ eine Signaturliste anonymer Bezeichner $Sig_p(\overrightarrow{p})$ aufgebaut[1]; steht diese in Subsignaturbeziehung zu der Signaturliste der Formalparameter, so handelt es sich um legale Parameter:

$$\frac{}{\Gamma \vdash Sig_p([]) = []} \textit{ Sig Param}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash Sig_p(a : \overrightarrow{p}) = (? : A) : Sig_p(\overrightarrow{p})} \textit{ Sig Param 2}$$

$$\frac{}{\Gamma \vdash Sig_p((: A) : \overrightarrow{p}) = (? = A) : Sig_p(\overrightarrow{p})} \textit{ Sig Param 3}$$

Beispielsweise lauten die Formalparametersignaturen der polymorphen Identität: $[(T <: Void), (t : T)]$. Ein Aufruf mit Parametern $[:Int, 3]$ ist legal, denn

$$Sig_p([:Int, 3]) = [(? = Int), (? : Int)]$$
$$[(? = Int), (? : Int)] <:: [(T <: Void), (t : T)]$$

## 4.4 Systemvalidität

Systemvalidität zu beweisen, ist das Ausgangsziel des Typprüfungsalgorithmus. Dazu müssen alle Klassendefinitionen auf Typkorrektheit überprüft werden.

[1] Diese etwas komplex wirkende Definition erweist sich bei der Einführung benannter Parameter von Vorteil.

### 4.4.1 Das initiale Environment

Alle durch Klassen definierte Typen (s. 4.5) sind global sichtbar. Da Klassen andere Klassen bzw. ihre Typbezeichner referenzieren, müssen diese in einem initialen Environment $\Gamma_0$ definiert werden, in dem die einzelnen Klassen dann typgeprüft werden.

$S$ bezeichne das System, d.h. alle definierten Klassen. $\sigma$ bilde eine Klasse auf ihr Interface bzw. ihren Typoperator ab. Das initiale Environment wird folgendermaßen definiert:

$$\Gamma_0([]) = []$$
$$\Gamma_0(K : S) = (K = \sigma(K)) : \Gamma_0(S)$$

### 4.4.2 System und Klassen

Damit das System typkorrekt ist, müssen alle Klassendefinitionen im initialen Environment typkorrekt sein.

$$\frac{\forall K \in S : \Gamma_0(S) \vdash K \ \mathit{valid}}{S \ \mathit{valid}} \ \mathit{System\ valid}$$

Eine Klassendefinition ist typkorrekt, wenn

- ihre Parametereinschränkungen wohlgeformte Typen bilden, die Subtyp von *Void* sind,
- ihre Superklassendefinitionen wohlgeformte Typen bilden, die Subtyp von *Void* sind,
- ihre Metaklassendefinition einen wohlgeformte Typ bildet, der Subtyp von *Void* ist und den Beschränkungen bezüglich abstrakten und konkreten Klassen genügt (s. 4.5.3 u. 4.5.4),
- ihre *Self*-Signatur einen wohlgeformte Typ bildet, der Subtyp von *Void* ist,
- ihre CPL valide ist (s. *Valid CPL*, 4.7.2) sowie
- alle Methodenkörper in einem um die Klassenparameter und *Self* erweiterten Environment typkorrekt und von dem jeweils deklarierten Ergebnistyp sind.

## 4.5 Klassen und Objekte

### 4.5.1 Objekttypen

Ein Objekt definiert sich durch seinen internen Zustand und die Nachrichten, die es versteht. Nach außen sichtbar sind nur die Methoden des Objekts, deren Signaturen definieren daher seinen Typ (sein *Interface*).

Ein Interface besteht folglich aus einer Menge von Methodensignaturen. Exemplarvariablen definieren je eine Lese- und Schreibmethode. Einer Exemplarvariablen `size :Int` entsprechen z.B. die Methoden `size :Int` (Lesemethode) und `"size:="(newSize :Int)` (Schreibmethode). Die Definition dieser *Accessor*-Methoden hat zwei Vorteile: Uniformität des Typsystems und vollständige Kapselung der Objektimplementierung. Eine Exemplarvariable kann wie in Eiffel[Meyer 92] ohne Änderung des Interfaces (und damit ohne Änderung der Klienten dieser Klasse) durch eine parameterlose Methode ersetzt werden; umgekehrt kann z.B. eine aufgeschobene parameterlose Methode durch die Definition einer Exemplarvariablen "implementiert" werden.

Der spezielle Typbezeichner *Self* steht für den Typ des Empfängerobjekts. Er kann Teil der Parameter- oder Ergebnistypen innerhalb des Interfaces sein. *Self* ist relevant für die typsichere Vererbung von Methodenimplementierungen und wird näher behandelt in 4.7.

### 4.5.2 Klassen und Typen

TL-2 ist eine klassenbasierte Sprache, d.h. Objekte werden nicht separat definiert, sondern als Exemplare ihrer Klasse. Diese legt fest, welche Exemplarvariablen das Objekt besitzt und welche Nachrichten es versteht. Somit bestimmt die Klasse den Typ ihrer Exemplare. Daher sind die Klassenbezeichner gleichzeitig auch Typbezeichner und stehen für den Typ, den sie bilden. Im Falle unparametrisierter Klassen ist dies ein Interface. Dieser enthält die öffentlichen Methoden der Klasse und ihrer Superklassen. Genaueres im Abschnitt über Vererbung (4.7). Ist die Klasse parametrisiert, so definiert sie einen gleichnamigen Typoperator, der seine Argumente auf ein Interface abbildet. Beispielsweise führt die Klassendefinition `class List(E <: Object) ...` zur folgenden Definition des Typbezeichners: $List = \Lambda(E <: Object)...$ .

### 4.5.3 Metaklassen

Auf den Typ der Exemplare einer Klasse hat deren Metaklasse zunächst keinen Einfluß. Die Klasse der Exemplare kann zwar über die Methode `"class"` erfragt werden, doch der Rückgabetyp dieser Methode ist `Class`, die Oberklasse aller Metaklassen. Grund dafür ist, daß Metaklassen nicht zwingend Subklassen voneinander sind, auch wenn ihre Exemplare dies sind.

Ein Beispiel hierfür wären eine unsortierte Liste `List` und eine sortierte Liste `SortedList`, die ihre Sortierung durch eine Funktion bestimmt. Die Metaklasse von `List`, `ListClass`, bietet dann einen unparametrisierten Konstruktor an, `new :List`, während der Konstruktor von `SortedList` einen Parameter, die Sortierungsfunktion, erwartet. Um eine Subtypbeziehung zwischen `List` und `SortedList` zu gewährleisten, kann die `class`-

Methode folglich nicht spezifischer typisiert werden.

Zwei wichtige Standard-Metaklassen sind `AbstractClass`, von der keine Exemplare erzeugt werden können, und `ConcreteClass`. Letztere enthält eine private Methode `_new`, die ein neues, uninitialisiertes Exemplar ihrer Exemplarklasse im Objektspeicher anlegt. Klassen, von denen Exemplare erzeugt werden können, definieren daher eine Metaklasse, die von `ConcreteClass` erbt und besagte Methode zur Erzeugung neuer Exemplare nutzt. `ConcreteClass` ist eine mit dem Exemplartyp parametrisierte Klasse, den die Methode `_new` als Rückgabetyp trägt. Siehe auch den Abschnitt über parametrischen Polymorphismus (4.8).

Definiert eine Klasse `K` eine Subklasse von `ConcreteClass(T)` als ihre Metaklasse, so können durch die Methode `_new` Exemplare von K erzeugt werden. `_new` ist jedoch mit `T` als Rückgabetyp deklariert. Es muß daher sichergestellt werden, daß `K` Subtyp von `T` ist.

Neben einem globalen Typbezeichner definiert jede Klasse auch eine Methode gleichen Namens in einem globalen Namensraum, dem *Pool*, die es erlaubt, auf das Klassenobjekt zuzugreifen. Diese ist im Gegensatz zu der `class`-Methode mit dem Typbezeichner der Metaklasse typisiert. Für die Klasse `String` existiert im Pool also eine Methode `String :StringClass`.

### 4.5.4 Methoden

Methoden definieren Schnittstelle und Verhalten von Objekten. Eine Methode besteht aus Parameter- und Ergebnistypdefinition sowie dem Methodenkörper, der die Methodenimplementierung bildet. Klassen definieren zwei Sichtbarkeitsbereiche für Methoden, *public* und *private*. Benutzer einer Klasse dürfen nur öffentliche Methoden (d.h. solche, die im *public* Bereich der Klasse definiert wurden) aufrufen. Private Methoden dürfen sowohl vom aktuellen Empfängerobjekt durch eine Nachricht an `self` bzw. `super` als auch vom Klassenobjekt gerufen werden[2].

Statt eines Implementierungsköpers können Methoden durch das Schlüsselwort `deferred` als aufgeschoben deklariert werden. Hierbei handelt es sich um eine reine Spezifikation, die Methodensignatur wird zwar in das Interface aufgenommen, eine Implementierung muß jedoch in Subklassen erfolgen. Klassen mit aufgeschobenen Methoden heißen *abstrakte Klassen.* Von ihnen sollten keine Exemplare erzeugt werden können, denn der Aufruf einer solchen Methode führt zu einem Laufzeitfehler. Deklariert daher eine abstrakte Klasse eine Metaklasse, die Subklasse von `ConcreteClass` ist, so wird dies als Typfehler gemeldet. Eine abstrakte Klasse dient folglich nur zur Vererbung und zur Typdefinition.

Liegt hingegen eine Methodenimplementierung vor, so muß geprüft werden, ob der gelieferte Rückgabewert von einem Subtyp des deklarierten Er-

[2]Auf diese Weise kann das Klassenobjekt neu erzeugte Exemplare initialisieren.

gebnistyps ist.

## 4.6 Subtypisierung auf Interfaces

Gängige statisch getypte objektorientierte Sprachen wie C++, Java oder Eiffel identifizieren Subtypisierung mit der Vererbungsrelation zwischen Klassen. Der einer Klasse entsprechende Typ ist genau dann Subtyp des einer anderen Klasse entsprechenden Typs, wenn die erstere Klasse von der letzteren direkt oder indirekt erbt.

TL-2 hingegen basiert, ähnlich Sprachen wie Emerald [Black, Hutchinson 90] oder Polytoil [Bruce et al. 95], auf *struktureller Subtypisierung*. Die Subtyprelation ergibt sich hierbei allein induktiv aus der Struktur der beteiligten Typen.

Ein Interface $A$ ist dann Subtyp eines anderen Interfaces $B$, wenn

- $A$ für jede Methode in $B$ eine entsprechende Methode definiert.
- die Methode in $A$ *mindestens* die gleichen Parameter wie die korrespondierende Methode in $B$ akzeptiert. Dies ist genau dann der Fall, wenn die Parametersignaturen der Methode aus $B$ Subsignaturen der Signaturen aus $A$ sind, entsprechend der kontravarianten Stellung der Methodenparameter.
- die Methode in $A$ nur Ergebnisse zurückliefert, die dem in $B$ definierten Typ entsprechen, d.h. der Ergebnistyp in $A$ muß Subtyp des Ergebnistyps in $B$ sein.

Sind diese Bedingungen erfüllt, dann kann sicher ein Objekt vom Typ $A$ unter $B$ subsumiert werden.

Die Subtyprelation auf Interfaces läßt sich somit wie folgt formalisieren: bezeichne *pubSel(T)* die Menge der öffentlichen Methodenselektoren im Typ $T$, sowie *meth(T, m)* die Methodensignatur der Methode mit dem Selektor $m$ in $T$.

$$\frac{\Gamma \vdash \overrightarrow{S'} <:: \overrightarrow{S} \quad \Gamma \vdash T <: T'}{m(\overrightarrow{S}) : T <: m(\overrightarrow{S'}) : T'} \; \textit{Subsig Method}$$

Dann gilt mit $T, T'$ in der Form $\sigma[\overrightarrow{m(\overrightarrow{S}) : T}]$:

$$\frac{\Gamma \vdash \mathit{pubSel}(T') \subseteq \mathit{pubSel}(T) \quad \Gamma \vdash \forall m \in \mathit{pubSel}(T') : \mathit{meth}(T, m) <: \mathit{meth}(T', m)}{\Gamma \vdash T <: T'} \; \textit{Subtype Interface}$$

Im Gegensatz zum vererbungsbasierten Ansatz erfasst strukturelle Subtypisierung auch Subtypbeziehungen, die sich nicht direkt im Sourcecode ausdrücken lassen. Dies gilt insbesondere für parametrisierte Typen. So ist es

zum Beispiel wünschenswert, eine Liste eines speziellen Elementtyps überall dort verwenden zu können, wo eine Liste eines allgemeineren Typs erwartet wird:

$$A <: B \Rightarrow List(A) <: List(B)$$

Diese Beziehung lässt sich nicht in C++ ausdrücken, in Eiffel ist sie durch eine spezielle Regel immer gültig. Dadurch entstehen jedoch Lücken im Eiffel-Typsystem, dann nämlich, wenn der Typparameter in einer kovarianten Position, z.B. als Methodenparameter, steht. Strukturelle Subtypisierung hingegen läßt nur genau die typsicheren Subtypbeziehungen zu.

## 4.7 Vererbung

Klassen können von anderen Klassen, ihren *Superklassen*, erben und von ihnen so Methoden übernehmen. Handelt es sich um parametrisierte Klassen, so müssen entsprechende Typparameter eingesetzt werden. TL-2 ermöglicht es, von mehreren Klassen zu erben (*multiple inheritance*).

### 4.7.1 Die Class Precedence List

Betrachtet man die transitive Hülle der Vererbungsrelation einer Klasse, so entsteht ein gerichteter, azyklischer Graph. Ähnlich wie in CLOS [Bobrow et al. 88] wird dieser Graph in TL-2 in eine lineare Form gebracht, die *class precedence list* (CPL) der Klasse. Die CPL bestimmt die Reihenfolge, in der der Bytecodeinterpreter nach einer aufzurufenden Methode sucht, daher berechnet sich aus ihr der von der Klasse gebildete Typ.

Die CPL besteht aus (möglicherweise) parametrisierten Klassenbezeichnern und somit gleichzeitig aus Interfaces bezeichnenden Typausdrücken. Die definierte Klasse steht dabei am Anfang der CPL. Die CPL der Klasse `Sequence` lautet beispielsweise:

`[Sequence(T), Container(T), Collection(T), Object,` *Pool*[3]`]`

Der Algorithmus zur Berechnung der CPL lautet:

$$CPL(K) = K : unique\ (flatten\ ([CPL(S) \mid S \leftarrow superclasses(K)]))$$

(*unique* entfernt mehrfache Vorkommen eines Listenelements bis auf das letzte, *flatten* "klopft" geschachtelte Listen flach.)

Existieren Methoden gleichen Namens in mehreren Klassen in der CPL, so gilt die vorderste Definition. Diese Methode überschreibt alle ererbten Methoden gleichen Namens. Redefinition von Methoden erfordert die Wahrung bestimmter Konsistenzkriterien, welche im folgenden erläutert werden.

[3]Die "ererbten" Poolmethoden ermöglichen es, Klassenobjekte uniform durch Methodenaufrufe zu erreichen.

### 4.7.2 Typsichere Vererbung

#### Der Typ *Self*

Neben Methodenaufrufen an andere Objekten kann eine Methodenimplementierung auch Methodenaufrufe an das aktuelle Empfängerobjekt, denotiert durch das Schlüsselwort `self`, schicken. Methodenaufrufe an `self` könnten zwar gegen die Klasse geprüft werden, in der die Methode definiert ist, jedoch ändert sich der Typ von `self`, wenn diese Methode an eine Subklasse vererbt wird. Daher weist man `self` einen vordefinierten Typ *Self* zu.

Man betrachte beispielsweise folgende Klassen:

```
class A {
public:
  one { self.two }
  two { tycoon.stdout << "A::two" }
}

class B : A {
public:
  two { tycoon.stdout << "B::two" }
}
```

Die Methode `one` enthält einen Methodenaufruf an `self`. Betrachtet man ein Exemplar der Klasse `A`, so hat `self` den Typ `Self = A`. Andererseits hat `self` durch Vererbung an die Klasse `B` den Typ `Self = B`.

Um sicherzustellen, daß vererbte Methoden auch in einer Subklasse noch typkorrekt sind, werden die Methoden einer Klasse $K$ unter folgenden Annahmen geprüft:

$$\mathit{Self} <: K.$$

Alle Subklassen von $K$ müssen diese Annahmen sicherstellen, da sonst keine typsichere Vererbung garantiert werden kann.

Diese Regel hat folgende Implikation: erbt $L$ von $K$, so gilt für Exemplare von $L$: $\mathit{Self} = L$. Aus obiger Regel folgt dann: $L <: K$. Vererbung impliziert also Subtypisierung. Unglücklicherweise verhindert dies die typsichere Vererbung sogenannter *binärer* Methoden, d.h. Methoden mit mit *Self* typisierten Parametern. Verschiedene Ansätze zur Lösung werden diskutiert: F-bounded parametrischer Polymorphismus [Canning et al. 89], type matching [Abadi, Cardelli 95] sowie multiple dispatch [Castagna 94]. TL-2 ist eine Sprache mit single dispatch und Subtypisierung als einziger Typrelation und erfordert daher die Anwendung des ersten zitierten Ansatzes. (Tatsächlich muß eine solche Klasse (z.B. `Ordered(T)`) unter einer strengeren Annahme, nämlich $\mathit{Self} <: T$, geprüft werden. Dies ist durch die optionale *Self*-Signatur im Klassenkopf möglich.)

### Der Typ *Super*

Neben Nachrichten an `self` können durch sogenannte *super*-Nachrichten Methodenaufrufe an das aktuelle Empfängerobjekt gesendet werden. Dabei wird allerdings nicht die ganze CPL nach der gerufenen Methode durchsucht, sondern nur hinter der Fundstelle der aktuell ausgeführten Methode. Dadurch können innerhalb von Vererbungshierarchien die überschriebenen Versionen von Methoden ausgeführt werden. Syntaktisch werden super-Nachrichten wie Nachrichten an einen vordefinierten Bezeichner `super` ausgedrückt, man kann sich `super` wie ein Objekt mit einem vordefinierten Typ *Super* vorstellen, obwohl `super` für sich allein keinen legalen Wertausdruck darstellt.

Der Typ *Super* läßt sich damit statisch aus der CPL der Klasse ohne das erste Element ableiten. Durch Mehrfachvererbung und damit verbundener Streichung von Klassen aus der CPL kann sich die zum Lookup verwendete Restliste allerdings gegenüber der CPL der beerbten Klasse ändern. Damit diese Liste weiterhin dem statischen Typ *Super* entspricht, darf in der CPL einer Klasse jede Methode nur durch eine subtypkonforme Version überschrieben werden.

Beispiel: Man betrachte folgende Klassen:

```
class A     { public: x :Nil  {  nil    } }
class B     { public: x :Int  {   3     } }
class C : A { public: x :Nil  { super.x } }

class D : C, B, A { }
```

Die CPL von `D` lautet `[D,C,B,A]`, und es gilt $D <: A,\ B,\ C$. Trotzdem ist `D` nicht typsicher, da sich die Klasse `B` in der CPL zwischen `C` und `A` geschoben hat und somit eine nicht konforme Methode beim Aufruf `super.x` liefert.

### Validität von Class Precedence Lists

Bezeichnet man den aus einer CPL $L$ ableitbaren Typ als *Type(L)*, so ergibt sich folgende Validitätsregel:

$$\boxed{\frac{}{\Gamma \vdash \mathit{valid}([])}\ \mathit{Valid\ CPL}}$$

$$\boxed{\frac{\Gamma \vdash \mathit{Type}(T : L) <: \mathit{Type}(L) \quad \Gamma \vdash \mathit{valid}(L)}{\Gamma \vdash \mathit{valid}(T : L)}\ \mathit{Valid\ CPL\ 2}}$$

Man beachte, daß in diesem Fall *alle* Methodensignaturen am Subtyptest beteiligt sind, nicht nur die öffentlichen wie in 4.6.

Diese Regel erzwingt Typsicherheit sowohl für Aufrufe an `self` wie auch an `super`, denn sie impliziert die obigen Bedingungen.

## 4.8 Parametrischer Polymorphismus

Das Typsystem von TL-2 beinhaltet *parametrischen Polymorphismus* [Cardelli, Wegner 85] in Form von typparametrisierten oder *generischen* Klassen und Methoden. Parametrischer Polymorphismus erlaubt es, generische Klassen und Algorithmen zu implementieren, die in verschiedenen Instantiierungen mit konkreten Typparametern eingesetzt werden können. Dies ist insbesondere für die Realisierung von Behälterklassen von Bedeutung, die von dem Elementtyp abstrahieren.

Der benannte Typparameter erlaubt es, Typinformation über das aktuelle Typargument in der Typprüfung weiterzuverwenden. Dies ist allein durch die Verwendung von Subtyppolymorphismus nicht möglich, da durch Subsumption eine Generalisierung des Typs und damit ein Informationsverlust eintritt.

Als Beispiel diene die polymorphe Identität. Sie läßt sich auf zwei Wegen formulieren:

- durch Subtyppolymorphismus. Generizität wird erreicht durch Subsumption eines spezielleren Typs durch `Object`.
  $\texttt{id}_{sub}$`(x :Object) :Object { x }`
- durch parametrischen Polymorphismus:
  $\texttt{id}_{parametric}$`(T <: Object, x :T) :T { x }`

Beide Methoden liefern als Ergebnis das Wertargument `x`, bei der ersten Variante ist als Typinformation für das Ergebnis allerdings nur der allgemeine Typ `Object` verfügbar. Die typsichere Verwendung des Ergebnisses ist dadurch stark eingeschränkt und zwar um so mehr, je generischer und damit allgemeiner der Algorithmus getypt ist. Mit der zweiten Variante lassen sich hingegen auch Ausdrücke wie `id(:Int, 3) + 4` statisch typprüfen.

### 4.8.1 Eingeschränkter parametrischer Polymorphismus

C++ realisiert parametrischen Polymorphismus durch *Templates* [Stroustrup 97]. Um zu prüfen, ob ein aktuelles Typargument eine typsichere Instantiierung eines Templates darstellt, ist eine makroartige Expansion der Templatedefinition notwendig. Dies führt zu langen Kompilationszeiten und erfordert, daß die Templatedefinition im Sourcecode vorliegt.

Es ist wünschenswert, im Gegensatz dazu die Definition einer generischen Klasse oder eines generischen Algorithmus nur einmal zu prüfen, und dabei sicherzustellen, daß alle möglichen Instantiierungen typkorrekt sein werden. Dies wird in TL-2, wie z.B. in Eiffel, durch eingeschränkten parametrischen Polymorphismus (*bounded parametric polymorphism*) erreicht. Hierbei wird für das formale Typargument eine obere Schranke, d.h. ein Supertyp, angegeben.

Die generische Definition kann nun mit dieser Information typgeprüft werden. Praktisch gesehen heißt das, daß die generische Klasse alle Operationen auf Werten des Typparameters ausführen kann, die die Typbeschränkung aufführt.

Soll beispielsweise aus einer Liste von Angestellten der bestbezahlte ausgewählt werden, so kann dies folgendermaßen implementiert werden:

```
bestPaid(T <: Employee, employees :Sequence(T)) :T
{
  employees.sort(fun(e1 :T, e2 :T) {    ; Sortierfunktion
    e1.salary > e2.salary
  })

  employees[0]
}
```

Die Typeinschränkung `T <: Employee` garantiert dann, daß die Implementierung typsicher die Nachricht `salary` senden kann.

Bei jeder Instantiierung einer generischen Klasse bzw. einer generischen Methode ist nun lediglich noch zu prüfen, ob das Aktualargument der Typbeschränkung genügt.

In manchen Fällen, insbesondere im Zusammenhang mit binären Methoden, ist es vonnöten, daß das Typargument nicht nur Subtyp eines festen Typs ist, sondern daß seine Methoden in ihren Argumenten den Typ selbst wieder enthalten. Es muß also möglich sein, eine Selbstbezüglichkeit des Typarguments auszudrücken. Eine Typschranke mit *Self* als Parametertyp in den Methoden entfällt, da durch Kovarianz keine Subtypbeziehung herrscht. Statt dessen zieht man selbstbezüglichen Typ aus der Typschranke heraus, indem diese selber generisch gemacht wird, und erlaubt einen rekursiven Bezug der Typschranke auf den Typparameter. Diese Technik nennt sich *F-bounded parametric polymorphism*[Canning et al. 89] (siehe dazu auch 4.7.2).

Ein klassisches Beispiel hierfür sind sortierbare Behälter, deren Elemente sich mit Elementen *des gleichen Typs* vergleichen können. Dafür definiert man ein generisches Sortierinterface:

```
class Sorted(T <: Object) : Object
  meta AbstractClass
{
public:
  "<"(t :T) :Bool deferred
}
```

Der Behälter wird dann wie folgt parametrisiert:

```
class SortedList(T <: Sorted(T)) ...
```

Diese Parametrisierung stellt nun sicher, daß jeder eingesetzte Typ $T$ eine Methode `"<"(t :T) :Bool` zur Verfügung stellt, die für die Sortierung verwendet werden kann.

### 4.8.2 Generische Klassen

Während alle Exemplare unparametrisierter Klassen den gleichen Typ haben, ermöglichen parametrisierte Klassen die Instantiierung von Objekten unterschiedlicher Typen. Bei der Erzeugung müssen die Aktualargumente der Typparameter eingesetzt werden. Das bedeutet, ein Exemplar einer generischen Klasse ist in allen Typargumenten festgelegt und nicht selbst generisch.

#### Typoperatoren

Bei einer parametrisierte Klasse `class X(T <: A) ...` stellt für alle Typausdrücke $B$ mit $B <: A$ der Ausdruck $X(B)$ einen wohlgeformten Typausdruck dar. $X$ realisiert also eine Abbildung von Typen auf Typen und bildet somit einen *Typoperator*. Typoperatoren werden wie folgt notiert:

$$\Lambda(\overrightarrow{T <: A}).B$$

Folglich gilt: Bildet eine parametrisierte Klasse `class X(T <: A) ...` wie oben beschrieben einen Typ $Ifc = \sigma[\overrightarrow{m(\overrightarrow{S}) : B}]$, dann lautet die Signatur des Bezeichners $X$:

$$X = \Lambda(\overrightarrow{T <: A}).Ifc$$

Entsprechend lassen sich Anwendungen von Typoperatoren wieder reduzieren:

$$\frac{\Gamma \vdash O <: \Lambda(\overrightarrow{S}).A \quad \Gamma \vdash Sig_p(\overrightarrow{P}) = \overrightarrow{S'} \quad \Gamma \vdash \overrightarrow{S'} <:: \overrightarrow{S}}{\Gamma \vdash O(\overrightarrow{P}) <: A[\overrightarrow{P}/\overrightarrow{S}]} \; \textit{Type apply}$$

Typoperatoren werden einzig durch parametrisierte Klassen eingeführt, es gibt keine Syntax, um diese in einem TL-2 Quelltext auszudrücken. Aus diesem Grund lassen sich in TL-2 auch keine Operatoren konstruieren, die Typen auf andere Operatoren abbilden.

#### Subtypisierung höherer Ordnung

Das Typsystem von TL-2 definiert induktiv auf der Subtyprelation aufbauend eine Relation zwischen Typoperatoren. Streng genommen handelt es sich um um eine Familie von Relationen zwischen Operatoren gleicher Parameterzahl und Parametern gleichen "Grades". Aufgrund der Ähnlichkeit zu Typen wird diese Relation allerdings mit dem gleichen Relationszeichen

$<:$ bezeichnet und man spricht von *Subtypisierung höherer Ordnung (higher order subtyping)*.

$<:$ ist auf Operatoren wie folgt definiert:

$$\frac{\Gamma \vdash \overrightarrow{S'} <:: \overrightarrow{S} \quad \Gamma \vdash A[\overrightarrow{S'}/\overrightarrow{S}] <: B}{\Gamma \vdash \Lambda(\overrightarrow{S}).A <: \Lambda(\overrightarrow{S'}).B} \; \textit{Sub Oper}$$

Die Subsignaturbeziehung $\overrightarrow{S'} <:: \overrightarrow{S}$ stellt sicher, daß der "kleinere" Operator links des Relationszeichens mindestens alle Typargumente akzeptiert, die auch der größere akzeptiert. Die zweite Prämisse besagt, daß die Anwendung des kleineren Operators einen Subtyp der Anwendung des größeren auf die gleichen Argumente ergibt. Diese Regel ist analog zur Regel zur Subtypisierung von Methoden (4.6).

Entsprechend ist es möglich, nicht nur mit Typen parametrisierte Klassen und Methoden zu definieren, sondern auch mit Typoperatoren. Die Formalparameter tragen dann entsprechend einen Typoperator als Beschränkung.

Beispiel: $\Lambda(C <: \mathit{Container}).C(\mathit{Int})$.

#### Generische Klassen im Pool

Wie im Abschnitt über Metaklassen (4.5.3) beschrieben, wird für jede Klasse im System eine globale Methode im Pool angelegt, die das Klassenobjekt liefert.

Im Falle generischer Klassen ist dies nun selbst eine mit den Klassenparametern parametrisierte Methode und erzwingt somit die Festlegung der Typparameter bei der Selektion des Klassenobjektes.

Beispielsweise ist die Klasse `List` mit dem Elementtyp `E <: Object` parametrisiert und hat die Metaklasse `ListClass(E)`. Dies führt zu folgender Definition im Pool:

```
List(E <: Object) :ListClass(E)
```

Somit kann eine leere Integerliste wie folgt erzeugt werden: `List(:Int).new`.

Man beachte, daß auch diese Typinformation nur zum Zeitpunkt der Typprüfung bekannt ist. `List(:Int)` und `List(:String)` evaluieren zu dem selben Klassenobjekt.

### 4.8.3 Generische Methoden

Während, wie oben beschrieben, Exemplare generischer Klassen in den Typparametern bereits festgelegt sind, so kann Generizität durch typparametrisierte Methoden auch "innerhalb" von Objekten erreicht werden. Typparameter für Methoden werden genauso definiert wie für Klassen und nachfolgende Wertparameter können sich auf diese beziehen. Zum Beispiel defi-

nieren alle Behälterklassen in der TL-2 Standardbibliothek eine Abbildungsmethode

```
map(T <: Object, map :Fun(e :E)T) :Reader(T),
```

die einen Typparameter `T` sowie eine Abbildungsfunktion von `E` (dem Elementtyp des Behälters) nach `T` übernimmt, diese auf alle Elemente anwendet und eine Aufzählung der Abbildungsergebnisse zurückliefert.

Typparameter in Methoden verursachen keine Veränderung in den Subtypregeln für Methoden, außer daß nun auch Subsignaturbeziehungen zwischen Typbezeichnern zu beachten sind.

## 4.9 Ausdrücke

Ausdrücke bilden die Methodenimplementierungen. Aufgabe des Typüberprüfers ist es, sicherzustellen, daß nur typkonforme Methodenaufrufe auf den Objekten ausgeführt werden und daß eine Methode nur Objekte ihres deklarierten Ergebnistyps zurückliefert, da die Ergebnistypen ihrerseits in der Typüberprüfung anderer Methoden eingesetzt werden.

Im folgenden wird die Typisierung aller Ausdrücke von TL-2 beschrieben.

### 4.9.1 Literale

Folgende Literale existieren in TL-2.

| Beschreibung | Klasse | Beispiel |
|---|---|---|
| Ganzzahlen $\in [-2^{31}, 2^{31}-1]$ | `Int` | `3` |
| Ganzzahlen $\in [-2^{63}, 2^{63}-1]$ | `Long` | `3L` |
| IEEE double precision Fließkommazahlen | `Real` | `3.4` |
| ISO-Latin-1 Zeichen | `Char` | `'x'` |
| Zeichenketten | `String` | `"Hello"` |

### 4.9.2 Blöcke

Blockwerte bilden Methoden- und Funktionskörper und bestehen aus einer Sequenz von Wertausdrücken $\overrightarrow{a}$. Ein wohltypisierter Blockwert besteht aus wohltypisierten Werten, und sein Typ berechnet sich aus dem Typ des letzten Teilausdrucks. Ein leerer Block liefert den Typ *Void*.

$$\frac{}{\Gamma \vdash [] : \mathit{Void}} \; \mathit{Val\ Block}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash [a] : A} \; \mathit{Val\ Block\ 2}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash L : B}{\Gamma \vdash (a : L) : B} \; \mathit{Val\ Block\ 3}$$

### 4.9.3 let-Ausdrücke

Ein `let`-Ausdruck führt einen lokalen Bezeichner ein, der im folgenden Blockwert (bis zum Ende des umschließenden Blocks) sichtbar ist. Der Typ bestimmt sich aus dem Typ des Blockwerts, wenn dieser leer ist, so gilt der deklarierte Typ des Bezeichners. Der initial gebundene Wertausdruck muß von einem Subtyp des deklarierten Typs sein. Die Typdeklaration des Bezeichners kann auch weggelassen werden, der fehlende Typ bestimmt sich dann aus dem Typ des zugewiesenen Ausdrucks.

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A <: B}{\Gamma \vdash (let\ x\ :B = a\ in\ []) : B}\ \textit{Val let}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A <: B \quad \Gamma, x : B \vdash c : C}{\Gamma \vdash (let\ x\ :B = a\ in\ c) : C}\ \textit{Val let 2}$$

### 4.9.4 Methodenaufrufe

Bei der Typisierung eines Methodenaufrufs wird in dem Interface des Empfängerobjekts die Methode gleichen Namens nachgesehen. Ist weder der Empfänger der Nachricht `self` noch der Absender das Klassenobjekt, so muß die Methode öffentlich sein. Danach werden die Aktualparameter gegen die Formalparameter abgeglichen. Der Typ des Methodenaufrufs ist der deklarierte Ergebnistyp der Methode, eventuell mit instantiierten Typparametern. Außerdem wird der Bezeichner `Self` durch den Typ des Empfängerobjektes ersetzt.

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A <: \sigma[m(\overrightarrow{S}) : T] \quad \Gamma \vdash Sig_p(\overrightarrow{p}) = S' \quad \Gamma \vdash S' <:: \overrightarrow{S}}{\Gamma \vdash a.m(\overrightarrow{p}) : T[\overrightarrow{p}/\overrightarrow{S}][A/Self]}\ \textit{Val send}$$

Typparameter können beim Aufruf weggelassen werden. Der Typüberprüfer versucht dann, anhand der übrigen Parameter oder anhand des Kontexts den fehlenden Parameter herauszufinden. Diese Technik nennt sich *Typinferenz*. Daher kann die generische Identität auch wie folgt aufgerufen werden:

```
id( (* fehlender Typparameter *) 3) + 4
```

Der Inferenzalgorithmus ist im Detail beschrieben in 5.4.

### 4.9.5 Funktionsabstraktionen

Eine Funktionsabstraktion repräsentiert einen verzögert ausgewerteten, parametrisierten Blockausdruck. Ihr Typ setzt sich aus den Parametersignaturen und dem deklarierten Ergebnistyp zusammen. Im laufenden System ist

es nur möglich, Funktionen mit Wertparametern zu versehen. Grundsätzlich wären Typparameter aber ohne viel Aufwand realisierbar. Die formalen Typparameter sind im Rumpf sowie im Ergebnistyp der Funktion sichtbar. Der Typ des Funktionskörpers muß Subtyp des deklarierten Ergebnistyps sein. Der Ergebnistyp kann weggelassen werden, er wird dann inferiert.

$$\frac{\Gamma, \overrightarrow{S} \vdash \overrightarrow{a} : A \quad \Gamma, \overrightarrow{S} \vdash A <: B}{\Gamma \vdash (fun(\overrightarrow{S}) : B\{\overrightarrow{a}\}) : Fun(\overrightarrow{S})B} \; \textit{Val fun}$$

Funktionstypen sind äquivalent zu Interfaces:

$$Fun(\overrightarrow{S})T = \sigma[[](\overrightarrow{S}) : T]$$

Beispiel: Der Ausdruck `fun(s :String) { s.size }` hat den Typ

$$Fun(s : String)Int = \sigma[[](s : String) : Int]$$

Funktionsapplikationen sind daher durch Methodenaufrufe definiert.

### 4.9.6 Zuweisungsausdrücke

Der an eine lokale Variable gebundene Wert kann durch eine Zuweisung geändert werden. Der zugewiesene Wert muß dabei von einem Subtyp des deklarierten Bezeichnertyps sein. Der Typ einer Zuweisung ist *Void*.

Alle anderen "syntaktischen" Zuweisungen werden bereits in der Scopingphase in Methodenaufrufe umgewandelt. Die Zuweisung auf eine Exemplarvariable `a := b`, die aussieht wie eine Zuweisung an eine lokale Variable, stellt also tatsächlich einen Methodenaufruf `self."a:="(b)` dar und wird entsprechend typisiert.

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash x : B \quad \Gamma \vdash A <: B \quad x \textit{ ist lokaler Bezeichner}}{\Gamma \vdash (x := a) : Void} \; \textit{Val assign}$$

### 4.9.7 Ausnahmen

TL-2 benutzt eine Standardklasse `Exception`, um Ausnahmen zu erzeugen. Diese Klasse definiert eine Methode `raise`, die die Ausnahme auslöst. Dadurch wird der "normale" Kontrollfluß verlassen und die Ausnahme wird durch den Aufrufstack "nach oben" gereicht. Da der Typprüfer kein spezielles Wissen über diese Funktionalität hat, muß `raise` so typisiert sein, daß eine ausgelöste Exception anstelle eines jeden anderen Wertausdruckes akzeptiert wird. Folgerichtig lautet die Signatur: `raise :Nil`. Dies ermöglicht z.B. folgenden Ausdruck:

```
let item :Item = table.containsKey("item") ? {
  table["item"]
} : {
  ItemNotFound.new("item").raise
}
```

Um eine Ausnahme zu behandeln, steht ein `try`-Konstrukt zur Verfügung. Dies ist als private Methode der Basisklasse `Object` realisiert und steht somit allen Methodenimplementierungen durch eine Nachricht an `self` zur Verfügung. Die Signatur lautet:

```
try(T <: Void, body :Fun()T, handler :Fun(e :Exception)T) :T
```

Evaluiert die `body`-Funktion zu einem Wert, so ist dieser das Ergebnis des `try`-Konstrukts. Wird statt dessen eine Ausnahme ausgelöst, so wird die `handler`-Funktion mit dem Ausnahmeobjekt als Parameter ausgewertet. Da beide Funktionen einen Wert vom Typ `T` zurückliefern, ist das Ergebnis auch von diesem Typ.

Beispiel:

```
let l :Int := try(:Int, fun() :Int {
  x.length
}, fun(e :Exception) :Int {
  0
})
```

Dank Typinferenz läßt sich der gleiche Ausdruck auch kürzer schreiben:

```
let l = try({
  x.length
}, fun(e :Exception) {
  0
})
```

Unglücklicherweise ist es in der momentanen Implementierung nicht möglich, nur Ausnahmen einer bestimmten Klasse respektive eines bestimmten Typs zu behandeln. Die hierzu erforderlichen dynamischen Typtests sind bisher nicht Bestandteil des Systems. Da die Typinformation über die Klassenobjekte zur Laufzeit jedoch vorhanden ist, dürfte dies kein unüberwindliches Hindernis darstellen. Allerdings ist zur Laufzeit nicht einfach feststellbar, mit welchen Parametern ein Exemplar einer parametrisierte Klasse instantiiert wurde. Eine mögliche Lösung wäre die Einführung einer Form von existentieller Typquantifizierung, eine ähnliche Lösung wie in Pizza [Odersky, Wadler 97]: Ist $l$ Exemplar der Klasse `List`, so gilt $l : \exists(T <: Object).List(T)$.

### 4.9.8 Zusicherungen

Zusicherungen ermöglichen es, Aussagen über erwartete Programmzustände zu machen. Sind diese Aussagen während eines Programmablaufes nicht erfüllt, so wird die Auswertung abgebrochen. Die entsprechende `assert` Anweisung in TL-2 ist mit *Void* typisiert, ihre Bedingung muß einen Boolschen Wert darstellen.

$$\frac{\Gamma \vdash c : Bool}{\Gamma \vdash assert\ c : Void}\ Val\ assert$$

# Kapitel 5

# Der Typüberprüfungsalgorithmus von TL-2

Dieses Kapitel beschreibt die Implementierung des Typprüfers in TL-2. Hauptsächlich wurde dies durch Erweiterung der Klassen erreicht, die den abstrakten Syntaxbaum repräsentieren, insbesondere die Klassenhierarchie der Wertausdrücke (Subklassen von `Value`) und der Typausdrücke (Subklassen von `Type`).

## 5.1 Implementierungstechniken

Zunächst werden verwendete Implementierungstechniken erläutert, soweit sie für das Verständnis von Codestücken von Bedeutung sind.

### 5.1.1 Continuation Passing Style

Einige der im folgenden beschriebenen Methoden liefern konzeptuell mehrere Ergebniswerte, insbesondere ein Paar bestehend aus einem Typ und einer Substitution. TL-2 unterstützt weder mehrere Ergebniswerte noch call-by-reference Semantik, so daß eine alternative Technik eingesetzt wird. Statt die Werte zurückzugeben, erwartet die Methode einen weiteren Parameter, eine Funktion, an die die betreffenden Werte zur Weiterverarbeitung weitergegeben werden. Nach Aufruf der Funktion wird die Methode typischerweise beendet, so daß abstrakt gesehen die Funktion *nach* der Methode weiterarbeitet. Tatsächlich kann der Funktionsaufruf in einen endrekursiven Aufruf übersetzt werden. Diese Sichtweise führt zu der Bezeichnung *continuation* für die Funktion, die Technik als solche wird als *continuation passing style* (CPS) [Appel, Jim 89] bezeichnet.

Mit Hilfe von parametrischem Polymorphismus kann vom Ergebnistyp der Funktion abstrahiert werden und eine ebenso flexible Programmierung wie mit "konventioneller" Ergebniswertübergabe erreicht werden.
Beispiel:

```
m(T <: Void, cont :Fun(Int, Int) :T) :T
{
  cont[3, 4]                ; Rufe die Funktion und
}                           ; gib das Ergebnis zurueck
...
m(fun(i :Int, j :Int) {     ; Die Summe der beiden von m
  i + j                     ; gelieferten Zahlen
})                          ; Ergebnis: 7 :Int
...
m(fun(i :Int, j :Int) {     ; Ist das erste Ergebnis
  i > j                     ; groesser als das zweite ?
})                          ; Ergebnis: false :Bool
```

Der Typprüfer kann für den ersten Ausdruck den Typ `Int` inferieren, für den zweiten `Bool`.

### 5.1.2 Fehlerbehandlung

Die Implementierung des Typüberprüfers benutzt Ausnahmen, um Typfehler zu behandeln. Statt Prädikaten wie `isSubTypeOf` gibt es Methoden, die die erwarteten Beziehungen überprüfen und bei Nichteinhaltung eine Ausnahme der Klasse `TypeCheckError` auslösen. Dies erweist sich für die Erzeugung detaillierter Fehlermeldungen von Vorteil, da die Hierarchie der Typtests beim Propagieren der Ausnahme durch den Aufrufstapel "aufgesammelt" werden kann.

Dies wird weiterhin erleichtert durch die Verwendung von Funktionen höherer Ordnung: aus der Klasse `TypeCheckMixin` kann eine Methode `trace` geerbt werden, die die Erstellung der Testhierarchie übernimmt. Die Implementierung eines Typtests sieht dann etwa wie folgt aus:

```
trace(arg.pos, "While checking actual argument "+arg.name, {
  ... tatsaechlicher Test ...
})
```

Tritt während des Typtests (d.h. der Auswertung der übergebenen Funktion) ein Typfehler auf, übernimmt die Methode `trace` die angegebene Sourcecodeposition und den Fehlertext in die ausgelöste Ausnahme und damit in die erzeugte Fehlermeldung.

## 5.2 Der abstrakte Syntaxbaum

Die Syntaxanalyse erzeugt aus dem Eingabestrom den abstrakten Syntaxbaum. Dieser entspricht entweder einer Klassendefinition oder bei interaktiver Eingabe einem Wertausdruck. In der Scopingphase werden Objektreferenzen zwischen Elementen des Syntaxbaumes hergestellt, so daß die entstehende Struktur strenggenommen keinen Baum, sondern einen allgemeinen gerichteten Graphen darstellt[1].

Die Klassen des abstrakten Syntaxbaumes lassen sich systematisch auf folgende Weise aus der abstrakten Syntax ableiten:

- Nonterminale, also Symbole auf der linken Seite einer Produktion, werden repräsentiert als (oft abstrakte) Oberklassen.
- Produktionen werden auf Subklassifizierung abgebildet: eine rechte Seite einer Produktion wird repräsentiert als Unterklasse der entsprechenden Nonterminalklasse der linken Seite. Ihre Struktur definiert sich unter anderem durch Referenzen auf Objekte, die den Elementen der rechten Produktionsseite entsprechen.

Beispiel:
Das Nonterminal *Ausdruck* ist repräsentiert durch eine Klasse `Value`. Diese hat unter anderem eine Klasse `SendValue` für Methodenaufrufe als Unterklasse. Entsprechend der (expandierten) Produktion

$$Wert \rightarrow Wert.m(\overrightarrow{Wert})$$

enthält diese Exemplarvariablen für den Empfängerwert, den Methodenselektor und die Parameterwerte.

Im folgenden werden die einzelnen Elemente des Syntaxbaums erklärt.

### 5.2.1 Bezeichner

Bezeichnerdeklarationen, Signaturen also, sind durch die Klassen `ValueIde` für Wertbezeichner $x : A$ sowie `TypeIde` und `SubtypeIde` für Typbezeichner $T = A$ bzw. $T <: A$ realisiert.

Sie definieren zwei wichtige Methoden:

- `name :String`: Der Name des deklarierten Bezeichners.
- `type :Type`: Der Typausdruck rechts des Relationszeichens.

[1] F-bounded rekursiv gebundene Typparameter sowie indirekt rekursive Interfacetypen führen auch Zyklen in diesen Graphen ein.

### 5.2.2 Wertausdrücke

Wie bereits aus der abstrakten Syntax ersichtlich, gibt es in TL-2 im Gegensatz zu anderen Sprachen keine Unterscheidung zwischen Werten und Anweisungen. Ausdrücke, die keinen sinnvollen Wert zurückliefern, werden als vom Typ *Void* getypt. Der Typ Void enthält zwar alle Werte, aber keine Methoden, um diese zu manipulieren. Selbst der Identitätsvergleich (`==`) ist erst in der Klasse `Object` definiert.

Alle Wertausdrücke sind als Subklassen von `Value` implementiert. Betrachtet man Typregeln der Form *Val xxx*, so sieht man, daß diese jeweils die Typisierung einer bestimmten Klasse von Typausdrücken bestimmen: *Val send* typisiert z.B. Methodenaufrufe, *Val let* Bezeichnerbindungen. Entsprechend diesem Schema implementiert jede der Wertklassen eine Methode

```
check(T <: Void, cont :Fun(Type, Subst)T) :T
```

die der jeweiligen Typregel entsprechend den Wertausdruck typprüft und ein Paar *(Typ, Substitution)* an die Continuation weitergibt.

Folgende Klassen existieren entsprechend der abstrakten Syntax:

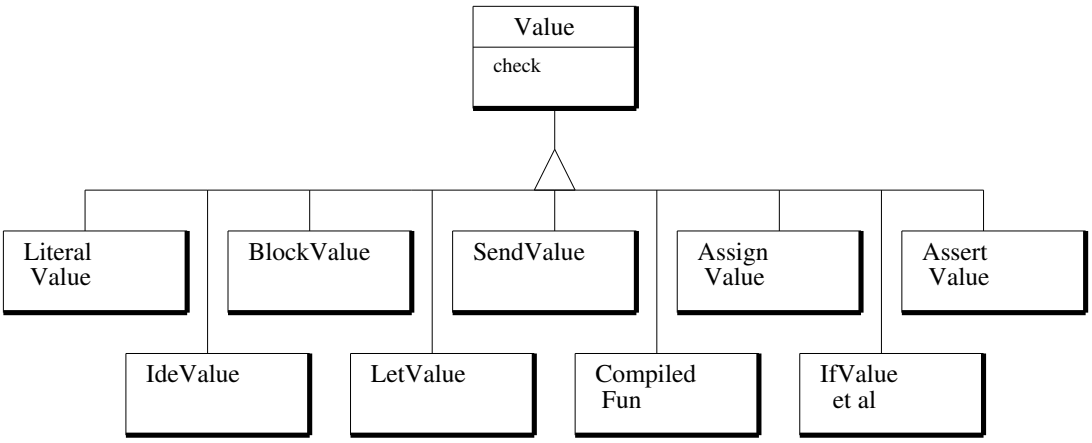


Abbildung 5.1: Klassenhierarchie für Wertausdrücke

### 5.2.3 Typausdrücke

Entsprechend den Wertausdrücken existiert eine Klassenhierarchie für Typausdrücke. Neben den in TL-2-Code formulierbaren Typausdrücken (Bezeichner, Operatorapplikation, *Nil, Void*) existieren weitere Klassen, die vom Typprüfer intern verwaltet werden:

- `ClassInstanceType` für alle aus Klassen gebildeten Interfaces. Dies sind die "eigentlichen" Typen, durch die transitiv z.B. alle Bezeichner definiert sind.

- `OperType` als Repräsentanten für Typoperatoren
- `UnifyType` für Unifikationsvariablen zur Typinferenz
- `SubstType` entsprechen einem verpacktem Paar *(Typ, Substitution)*
- `MethodType` als Repräsentation von Methodensignaturen

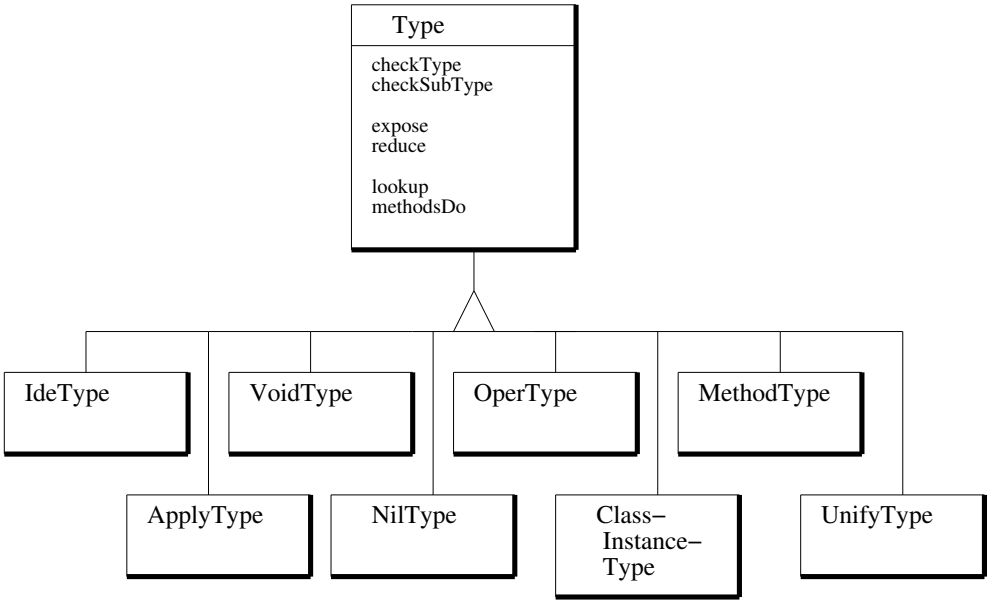


Abbildung 5.2: Klassenhierarchie für Typausdrücke

### Wohlgeformte Typen

Entsprechend den Regeln *Type XXX* implementiert jede Typklasse eine Methode `checkType`, die den Typausdruck auf Wohlgeformtheit überprüft. Es zeigt sich, daß nur für Typoperatorapplikationen Typtests entsprechen *Type apply* (4.8.2) durchgeführt werden müssen, nämlich, ob der vermeintliche Typoperator tatsächlich einer ist, und ob die Argumente seinen Parametersignaturen entsprechen. Nicht deklarierte Typbezeichner werden schon in der Scopingphase entdeckt, und die anderen Typen werden nur vom Typüberprüfer selbst erzeugt, der die Wohlgeformtheit zuvor sicherstellt.

### Subtyptests

Zum Testen der Subtypbeziehung existiert in `Type` die Methode `checkSubType`, die als Parameter den vermeintlichen Supertyp `big` sowie zwei Substitutionen, anzuwenden auf `self` und `big`, erhält. In mehreren

Fällen muß eine Unterscheidung basierend auf der Klasse von `big` durchgeführt werden, so daß über einen Visitor-Mechanismus [Gamma et al. 95] der spezielle Typ von `big` festgestellt wird und eine entsprechende Methode `checkSub`*XXX*`Type`, z.B. `checkSubApplyType` oder `checkSubIdeType` gerufen wird. Eine oder mehrere dieser Methoden müssen dann von der jeweiligen Typklasse den Typregeln entsprechend implementiert werden.

Diese Funktionalität wird von `Type` geerbt und steht somit allen Typklassen zur Verfügung. Damit lassen sich auch bereits allgemeine Zusammenhänge als Standardimplementierung bereitstellen, nämlich:

- `checkSubVoidType`: ist immer erfolgreich, da alle Typen Subtyp von *Void* sind. Nur für Typoperatoren muß diese Methode überschrieben werden, um einen Fehler zu liefern.
- `checkSubNilType`: kein Typ ist Subtyp von *Nil*, außer *Nil* selbst.
- `checkSubOperType`: kein Typ ist Subtyp eines Typoperators, außer einem Operator selbst.

### Rekursive Subtyptests und der Subtypcache

Das in 4.3.1 beschriebene Environment $\Sigma$ ist durch eine Tabelle im Typprüfer implementiert. Zur Vermeidung von Redundanzen werden nur Exemplare der Klassen `IdeType, ApplyType, NilType, VoidType, UnifyType` darin gespeichert.

Zusätzlich zum Abfangen rekursiver Subtyptests verbleiben bewiesene Instanzen der Subtyprelation in dieser Tabelle und bilden somit einen *Cache* für erfolgreiche Subtyptests. Allerdings sind insbesondere Einträge mit Unifikationsvariablen wertlos, da sich deren "Lebensdauer" auf einen Methodenrumpf beschränkt und vergrößern somit den Cache unnötig. Dieser kann nach der Typprüfung umfangreicher Programmquellen im Objektspeicher eine Größe von mehreren Megabyte einnehmen. Eine sorgfältigere Auswahl "brauchbarer" Cacheeinträge wäre daher wünschenswert.

### Aufdeckung

Für einige Operationen (Methodenselektion, Operatorapplikation, Subtypisierung mit Interfacetypen) ist es notwending, die innere Struktur, d.h. den Interfacetypen *aufzudecken*, der entweder gleich oder Supertyp des betrachteten Typausdrucks ist. Betrachtet man beispielsweise

```
m(P <: Person, p :P) :Money { p.salary }
```

so ist zu prüfen, ob der deklarierte Typ `P` des Bezeichners `p` die Methode `salary` anbietet. Dazu muß der Interfacetyp aufgedeckt werden und zwar über die folgende Sequenz:

$$\texttt{P} \rightarrow \texttt{Person} \rightarrow \sigma[\mathit{salary} : \mathit{Money} \ldots]$$

Für diese Operation existiert folgende Methode in `Type`:

```
expose(T <: Void,
       subst :Subst,
       exact :Bool,
       cont  :Fun(Type, Subst)T) :T
```

Der aufgedeckte Typ, also der Parameter an `cont`, ist dabei entweder *Nil*, *Void*, ein Interface oder ein Typoperator.

Der Parameter `exact` bestimmt dabei, ob der aufgedeckte Typ $B$ gleich dem Ausgangstyp $A$ sein muß, oder ob eine Subtypbeziehung $A <: B$ ausreichend ist. Zum Methodenlookup ist Subtypbeziehung ausreichend, so daß hier `false` eingesetzt wird.

Grob gesagt dient `expose` zur Umsetzung von Typregeln der Form:

$$\frac{a : A \quad A <: \sigma[\ldots]}{\ldots}$$

Der hier als Supertyp angegebene Interfacetyp entspricht dabei dem Typen, der von `expose` aus $A$ ermittelt wird.

### Methodenlookup

Zwei Methoden implementieren die Selektion von und Iteration über Methoden.

- ```
  lookup(T <: Void,
         selector    :Symbol,
         subst       :Subst,
         withPrivate :Bool,
         cont        :Fun(Method, Subst)T) :T
  ```

- ```
  methodsDo(subst       :Subst,
            withPrivate :Bool,
            callback    :Fun(Method, Subst)Void)
  ```

Die erste Methode erlaubt das Auffinden einer bestimmten Methode, dies wird zum Prüfen von Methodenaufrufen benötigt. Die zweite Methode iteriert über alle Methoden des Interfaces und ruft die Callbackfunktion mit ihnen auf. Das Argument `withPrivate` bestimmt dabei, ob das private oder öffentliche Interface benutzt wird.

Implementiert sind diese Methoden mit Hilfe von `expose`, ein Methodenlookup auf einem Nicht-Interfacetypen wird einfach an das Ergebnis von `expose` weitergereicht (vereinfacht dargestellt[2]):

[2]Tatsächlich muß noch für die Anpassung von `Self` Sorge getragen werden.

```
lookup(...) {
  ; expose self to type 't'
  ; and forward the lookup operation to 't'
  self.expose(..., fun(t :Type, s :Subst) {
    t.lookup(...)
  })
}
```

### Typapplikation

Um Typoperatorapplikationen zu prüfen und aufzulösen, existieren zwei weitere Methoden:

- `checkApplication(args      :List(Type),`
  `subst     :Subst,`
  `argsSubst :Subst)`

- `apply(T <: Void,`
  `args      :List(Type),`
  `subst     :Subst,`
  `argsSubst :Subst,`
  `exact     :Bool,`
  `cont      :Fun(Type, Subst)T) :T`

Auch Applikationen bedienen sich der `expose` Methode, um den Typoperator zu ermitteln.

### Reduktion

Um einen Vergleich zweier Typen zu ermöglichen (z.B. im Subtypcache, siehe 5.2.3), ist es notwendig, die entstandenen Substitutionen teilweise auszuwerten, da Substitutionen nicht ohne großen Aufwand auf Äquivalenz überprüft werden können. Dazu existiert in `Type` eine Methode `reduce` mit folgender Signatur:

```
reduce(T <: Void,
       subst :Subst,
       cont :Fun(Type, Subst)T) :T
```

Die übergebene Substitution wird auf den Empfängertyp soweit wie möglich angewendet, das Ergebnis wird an die Continuation `cont` weitergegeben.

## 5.2.4 Substitutionen

Substitutionen beschreiben das Einsetzen von Typausdrücken für freie Typbezeichner. Dies geschieht an mehreren Stellen:

- Anwendung von Typoperatoren: die Aktualargumente werden für die Formalparameter im Ergebnistyp ersetzt.
- Methodenaufrufe: die aktuellen Typargumente werden für die Formalparameter im Ergebnistyp ersetzt.
- Methodenaufrufe: der Bezeichner `Self` ist durch den Typ des Empfängers zu ersetzen.

Um ein aufwendiges Kopieren z.B. eines ganzen Interfacetypen zu vermeiden, werden Substitutionen explizit während der Typprüfung umhergereicht und verzögert ausgewertet. Aus diesem Grund tragen die meisten Methoden, die Typen behandeln, zwei Parameter, den Typen und eine noch auf ihn anzuwendende Substitution.

Substitutionen sind implementiert durch eine Klassenhierarchie mit der Oberklasse `Subst` als unveränderliche verkettete Listen. Diese Struktur erlaubt die gemeinsame Nutzung von Teilsträngen, ohne daß Substitutionsobjekte kopiert werden müßten. Die wichtigste Methode von `Subst` ist

```
subst(T <: Void,
      ide :Ide,
      ifFound :Fun(Type, Subst)T),
      ifNotFound :Fun()T) :T
```

Diese Methode versucht, den Parameter `ide` zu ersetzen. Gelingt dies, wird die Funktion `ifFound` mit dem Resultat und der noch nicht bearbeiteten Restsubstitution aufgerufen, sonst wird `ifNotFound` aufgerufen.

## 5.3 Typprüfung von Werten

Im folgenden werden alle Wertklassen aufgeführt. Die relevanten Methoden werden erläutert und abstrakt beschrieben, wie die Typprüfung realisiert ist.

### 5.3.1 LiteralValue

Exemplare dieser Klasse entsprechen Literalen im Programmcode. Wichtige Methoden sind:

- `value :Object`. Der Literalwert. Da das System in TL-2 selbst implementiert ist, kann bereits der Parser den richtigen Wert einsetzen.

Zur Typprüfung wird anhand der Klasse des Wertes (`value."class"`) entschieden, welchem Typ der Ausdruck angehört.

### 5.3.2 IdeValue

Referenzen auf Wertbezeichner werden durch Exemplare dieser Klasse repräsentiert. Wichtige Methoden sind:

- `ide :ValueIde`. Der referenzierte Wertbezeichner, d.h. lokale Variable, Methodenparameter oder `self`.

Durch die Scopingphase ist die Referenz auf das `ValueIde`-Objekt bereits auf das "richtige" Bezeichnerobjekt gesetzt worden, so daß der deklarierte Typ einfach ermittelt werden kann, nämlich durch `ide.type`. Die entsprechende Typregel ist *Val Ide*(4.3.3).

### 5.3.3 BlockValue

Blockwerte sind repräsentiert durch Objekte der Klasse `BlockValue`. Sie bilden Methoden- und Funktionskörper. Wichtige Methoden sind:

- `values :List(Value)`. Die Liste der enthaltenen Werte.

Um den Block typzuprüfen, wird jeder Teilausdruck typgeprüft und das letzte Ergebnis ist der Typ des Blocks. Dies entspricht der Regel *Val block*(4.9.2).

### 5.3.4 LetValue

Lokale Bezeichnerdeklarationen sind im AST dargestellt durch Exemplare dieser Klasse. Wichtige Methoden sind:

- `ide :ValueIde`. Der deklarierte Bezeichner.
- `value :Value`. Der initial gebundene Wertausdruck.
- `body :Value`. Der darauffolgende Block, in dem der Bezeichner sichtbar ist.

Der Typprüfer prüft den Typ des Wertausdruckes gegen die Bezeichnerdeklaration und bestimmt dann den Typ des `body` Ausdrucks, welcher das Gesamtergebnis darstellt. Die entsprechend Regel ist *Val let*(4.9.3).

Man beachte, daß aufgrund der vorher erfolgten Scopingphase bereits alle Referenzen auf das Bezeichnerobject `ide` innerhalb von `body` aufgelöst sind, so daß für die Prüfung des `body` Ausdrucks keine Bezeichnermanipulation mehr erforderlich ist.

### 5.3.5 SendValue

Exemplare dieser Klasse repräsentieren Methodenaufrufe. Sie bieten folgende wichtige Methoden an:

- `receiver :Value`. Das Zielobjekt des Aufrufs links vom Punkt.

- `selector :Symbol`. Der Name der gerufenen Methode.
- `args :List(Value)`. Die Liste der Argumente.

Der Typprüfer bestimmt den Typ des Zielobjekts und ermittelt daraus die Signatur der gerufenen Methode. Dann werden die Argumente gegen die Parametersignaturen der Methode abgeglichen und der Ergebnistyp der Methode zurückgegeben. Dies entspricht der Regel *Val send*(4.9.4).

### 5.3.6 CompiledFun

Exemplare dieser Klasse repräsentieren Funktionsabstraktionen. Ihre wichtigen Methoden sind:

- `domain :List(Ide)`. Die Parametersignaturen der Funktion.
- `range :Type`. Der deklarierte Ergebnistyp oder `nil`, falls weggelassen.
- `body :Value`. Der Funktionskörper.

Zur Typprüfung wird der Typ des Funktionskörpers bestimmt und auf Subtypbeziehung mit dem deklarierten Ergebnistyp untersucht. Aus `domain` und `range` wird dann der Funktionstyp $Fun(\overrightarrow{S})T$ konstruiert. Siehe *Val fun*(4.9.5).

### 5.3.7 AssignValue

Wertzuweisungen an lokale Variablen werden durch diese Klasse repräsentiert. Zur Scopingzeit wird festgestellt, ob es sich bei der LHS tatsächlich um einen lokalen Bezeichner handelt. Ist dies nicht der Fall, wird dieser Knoten durch einen Methodenaufruf (`SendValue`) ersetzt. Wichtige Methoden sind:

- `lhs :ValueIde`. Die left hand side der Zuweisung.
- `rhs :Value`. Der zuzuweisende Ausdruck.

Der Typ der RHS muß Subtyp des deklarierten Bezeichnertyps sein, Ergebnistyp ist *Void*, entsprechend Regel *Val assign*(4.9.6).

### 5.3.8 Kontrollstrukturen

Kontrollstrukturen wie `if`, `while` oder `try` sind durch Methodensignaturen in der Klasse `Object` repräsentiert. Um eine Kompilation in effizienteren Bytecode zu erlauben, werden für sie jedoch eigene AST-Knoten konstruiert. Dementsprechend erfolgt die Typprüfung auch separat, vollzieht aber die gleichen Schritte, die für den Methodenaufruf vonnöten wären.

Beispielhaft sei hier die Klasse `IfValue` für Fallunterscheidungen dargestellt. Sie definiert folgende Methoden:

- `condition :Value`. Die Bedingung.
- `trueBranch :Value`. Der zu wählende Wert, falls die Bedingung zu wahr evaluiert.
- `falseBranch :Value`. Der Wert für den Fall `false`.

Um einen `IfValue` typzuprüfen, muß die Bedingung von einem Subtyp von `Bool` sein. Um den Ergebnistyp festzustellen, wird Typinferenz angewandt. Dazu wird eine Unifikationsvariable angelegt und gegen die beiden Zweige geprüft. Die Implementierung der Typinferenz wird im Detail erläutert in Abschnitt 5.4.

### 5.3.9 AssertValue

Die Klasse für Zusicherungen definiert eine Methode:

- `value :Value`. Die zugesicherte Bedingung.

Entsprechend Regel *Val assert*(4.9.8) ist der Ausdruck typsicher, wenn der Typprüfer für `value` einen Typ bestimmt, der Subtyp von `Bool` ist. Ergebnistyp ist *Void*.

## 5.4 Typinferenz

Typinferenz bezeichnet die Möglichkeit, Typannotationen auszulassen und vom Typprüfer berechnen zu lassen. Auf diese Weise lässt sich der oft erhebliche Mehraufwand gegenüber untypisierten Sprachen vermindern. Während Typannotationen an vielen Stellen bedeutungstragend sind und das Verständnis des Programmtextes erleichtern, stellen sie in einigen Situationen unnötigen Ballast dar. Herausragendes Beispiel sind in diesem Zusammenhang Bezeichnerdeklarationen mit Konstruktoraufrufen oder Literalen:

```
let c :Compiler = Compiler.new          let c = Compiler.new
let i :Int      = 3                     let i = 3
```

Beide Typen der Beispiele auf der linken Seite ergeben sich sofort aus den gebundenen Ausdrücken und erleichtern somit nicht das Verständnis gegenüber den Deklarationen auf der rechten Seite.

In TL-2 können folgende Typannotationen inferiert werden:

- Typen deklarierter Bezeichner
- Ergebnistypen von Funktionsabstraktionen
- Typparameter bei Methodenaufrufen

- (als Spezialfall von Methodenaufrufen) Typen in Kontrollstrukturen wie *if* und *try*

Zur Inferenz werden zwei Strategien angewendet:

- bei Bezeichnerdeklarationen und Ergebnistypen von Funktionsabstraktionen wird direkt der aus dem entsprechenden Wert (RHS bzw. Funktionskörper) berechnete Typ eingesetzt.

- bei Methodenaufrufen und Kontrollstrukturen ist das zu inferierende Typargument oftmals Teil der Typen *mehrerer* folgender Argumente oder des Ergebniswertes, so daß hier mehrere Typen in Einklang gebracht werden müssen. Beispielsweise lautet die Signatur der Fallunterscheidung in der Klasse `Bool`:

  ```
  "?:"(T <: Void, ifTrue :Fun()T, ifFalse :Fun()T) :T
  ```

  Die prototypische Implementierung des Typprüfers wählt für `T` immer den Ergebnistyp des *true*-Zweiges und testet daraufhin, ob der *false*-Zweig dieser Wahl entspricht. Dies schlägt vor allem dann fehl, wenn der *true*-Zweig einen spezielleren Typ hat als erwünscht. Ein häufiges Beispiel:

  ```
  (a > 10) ? { nil } : { a.asString }
  ```

  Der erste Zweig hat den Typ `Nil`, der zweite den Typ `String`. Da `Nil` Subtyp von `String` ist, wäre `String` eine typkorrekte Belegung für `T`, diese wird jedoch vom prototypischen Typprüfer nicht erkannt, da dieser sich "zu früh" auf den Typ `Nil` festlegt.

  Die mit dieser Arbeit vorgelegte Implementierung des Typprüfers bereinigt das Problem auf folgende Weise: für den ausgelassenen Typparameter wird eine frische Unifikationsvariable `U?` eingesetzt. Wird im folgenden ein Subtyptest `U? <: A` oder `A <: U?` ausgeführt, so wird dies als *Constraint* in die Unifikationsvariable aufgenommen. Lassen sich alle Constraints einer Unifikationsvariablen miteinander vereinbaren, so ist die Typprüfung erfolgreich, andernfalls wird ein Typfehler gemeldet. Zur detaillierten Funktionsweise von Unifikationsvariablen siehe den nächsten Abschnitt.

### 5.4.1 Unifikation

Unifikationsvariablen in der vorliegenden Implementierung repräsentieren einen Typbereich mit einer Untergrenze $S$ und Obergrenze $B$:

$$S <: U? <: B \quad \text{, wobei } S <: B$$

Dies bedeutet folgendes: jedes Programmstück, in dem $U?$ bisher vorkam, ist typkorrekt **für jeden Typ $T$ innerhalb der gegebenen Grenzen**. Initialer Typbereich ist $Nil <: U? <: Void$.

Die Schranken einer Unifikationsvariablen dürfen daher nur derart verändert werden, daß der neue Bereich von $U?$ ein Unterbereich des alten ist, damit alle Typen innerhalb des neuen Bereichs weiterhin typkorrekte Lösungen für $U?$ sind. Auf dieser Feststellung beruht der Unifikationsalgorithmus: (gegeben sei $S <: U? <: B$)

- sei $U? <: T$ ein neues Constraint:
  1. gilt $B <: T$, dann ist $U? <: T$ ohnehin gewährleistet und die Schranken werden nicht verändert.
  2. gilt $S <: T <: B$, dann wird $T$ als neue obere Schranke gewählt: $S <: U? <: T$
  3. andernfalls wird als gemeinsamer Subtyp von $T$ und $B$ der Bottom-Typ *Nil* als neue obere Schranke eingesetzt. Um die Invariante *Untergrenze* $<:$ *Obergrenze* nicht zu verletzen, muß gelten: $B = Nil$, sonst schlägt die Unifikation fehl.
- analoge Überlegungen gelten für ein neues Constraint $T <: U?$:
  1. gilt $T <: S$, dann ist $T <: U?$ ohnehin gewährleistet und die Schranken werden nicht verändert.
  2. gilt $S <: T <: B$, dann wird $T$ als neue untere Schranke gewählt: $T <: U? <: B$
  3. andernfalls werden als gemeinsame Supertypen von $T$ und $B$ zunächst *Object* und, falls dies fehlschlägt, der Top-Typ $Void$ als neue untere Schranke eingesetzt. Auch hier muß die obige Invariante gewahrt bleiben. Der Test mit *Object* erweist sich als sinnvoll, da nahezu alle Klassen von `Object` erben und die Typen ihrer Exemplare daher Subtypen von *Object* sind.

Der Algorithmus könnte z.B. dahingehend verfeinert werden, daß ein spezielleren gemeinsamer Supertyp (Punkt 3) z.B. über gemeinsame Superklassen berechnet wird. Dies würde die Komplexität des Algorithmus jedoch weiter erhöhen. Insbesondere muß der Programmierer im Fehlerfall in der Lage sein, die Unifikationsentscheidungen nachzuvollziehen. Je flexibler der Inferenzalgorithmus ist, desto später werden Typfehler gemeldet und sind umso schwerer verständlich.

Besondere Schwierigkeiten treten auf, wenn die Subtypbeziehung *zweier Unifikationsvariablen* zu testen ist. Eine Variable als Unter- oder Obergrenze der anderen einzusetzen, erweist sich als algorithmisch nicht handhabbar. Stellt man sich beide Variablen stark vereinfacht als Intervalle in einem eindimensionalen Typraum vor, so müssen die Schranken so verändert werden,

daß sich die Intervalle nicht überlappen. Auf eine genauere Darstellung des gewählten Algorithmus wird hier verzichtet.

# Kapitel 6

# Zusammenfassung und Ausblick

Die vorliegende Arbeit beschreibt das polymorphe Typsystem höherer Ordnung der objektorientierten Sprache TL-2 und formalisiert dieses in weiten Teilen. Es wird der Zusammenhang von Klassen und Typen erläutert sowie die Subtyprelation auf Interfaces induktiv definiert. Der Zusammenhang von Typisierung und Vererbung wird dargestellt, insbesondere die Rolle des Typbezeichners *Self*. Weiterhin wird parametrischer Polymorphismus in Form von generischen Klassen und Methoden erläutert.

Eine entsprechende Implementierung des Typprüfers in TL-2 setzt diese Ergebnisse praktisch um. Dazu werden Änderungen in den Klassenhierarchien `Type` und `Value` des abstrakten Syntaxbaumes vorgenommen. Ein großer Teil der dabei implementierten Methoden hat dabei direkte Entsprechungen in den Typregeln.

Folgende Punkte erweisen sich in der Implementierung als problematisch und verlangen nach weitergehender Untersuchung:

- bei der Nutzung externer Dienste mit anderen Typsystemen oder Datenmodellen kann es zu Schwierigkeiten kommen. Beispielsweise sind die Typen *Int* und *Long* strukturkompatibel, bei Nutzung des C-Interfaces darf jedoch keine Subsumption verwendet werden. Statt Interfacekompatibilität ist hier Implementierungs- bzw. Darstellungskompatibilität erforderlich. Einen Lösungsansatz bietet hier die Programmiersprache *Modula 3* mit *branded types*([Nelson 91]).

- Klassenobjekten wird der Zugriff auf private Methoden ihrer Exemplare gewährt. Da Subtypisierung und damit Subsumption nur auf öffentlichen Schnittstellen beruht, ist für den Typprüfer schwer nachweisbar, daß ein entsprechend typisiertes Objekt *tatsächlich* auch diese privaten Methoden anbietet. Der in der vorliegenden Version realisierte Mechanismus ist zwar typsicher, schränkt die Möglichkeiten, aus

Klassenobjekten private Exemplarmethoden zu rufen jedoch teilweise stark ein.

- Die Erzeugung aussagekräftiger Fehlermeldungen erweist sich als schwierig. Insbesondere die Hierarchie erfolgter Typtests muß übersichtlich dargestellt werden, weiterhin müssen die getroffenen Typinferenzentscheidungen nachvollziehbar sein. Der vorliegende Algorithmus ist in dieser Hinsicht verbesserungswürdig.

Eine Erweiterung des Typprüfers um dynamische Typtests sollte abgesehen von grundsätzlichen Schwierigkeiten (s. *Ausnahmen*, 4.9.7) unproblematisch sein, da aufgrund der reflektiven und selbstenthaltenden Systemarchitektur der Typprüfer mit allen Typinformationen zur Laufzeit verfügbar ist.

# Literaturverzeichnis


*Abadi, Cardelli 95:* Abadi, M. and Cardelli, L. *On Subtyping and Matching.* In: *Proceedings of the European Conference on Object-Oriented Programming, Aarhus, Denmark.* Springer-Verlag, 1995.

*Appel, Jim 89:* Appel, A.W. and Jim, T. *Continuation-passing, closure-passing style.* In: *Proceedings of the Sixteenth ACM Symposium on Principles of Programming Languages*, 1989.

*Black, Hutchinson 90:* Black, A.P. and Hutchinson, N.C. *Typechecking Polymorphism in Emerald.* Technical Report TR-90-34, Department of Computer Science, University of Arizona, Tucson, Arizona, Dezember 1990.

*Bobrow et al. 88:* Bobrow, D.G., De Michiel, L.G., Gabriel, R.P., Keene, S.E., Kiczales, G., and Moon, D.A. *Common Lisp Object System Specification.* ACM SIGPLAN Notices, Jg. 23, September 1988.

*Bruce et al. 95:* Bruce, K.B., Schuett, A., and Gent, R. van. *PolyTOIL: a type-safe polymorphic object-oriented language.* In: *Proceedings of the European Conference on Object-Oriented Programming, Aarhus, Denmark.* Springer-Verlag, 1995.

*Canning et al. 89:* Canning, P.S., Cook, W.R., Hill, W.L., and Olthoff, W. *F-Bounded Polymorphism for Object-Oriented Programming.* In: *Proceedings of the Conference on Functional Proramming Languages and Computer Architecture, Imperial College, London*, September 1989, S. 273–280.

*Cardelli, Wegner 85:* Cardelli, L. and Wegner, P. *On Understanding Types, Data Abstraction, and Polymorphism.* ACM Computing Surveys, Jg. 17, Dezember 1985, Nr. 4, S. 471–522.

*Castagna 94:* Castagna, G. *Covariance and contravariance: conflict without a cause.* Technical Report liens-94-18, LIENS, Oktober 1994.

*de Bruijn 72:* de Bruijn, N.G. *Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem.* Indag. Math., Jg. 34, 1972, Nr. 5, S. 381–392.

*Gamma et al. 95:* Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, 1995.

*Goldberg, Robson 89:* Goldberg, Adele and Robson, David. *Smalltalk-80: The Language.* Addison-Wesley Publishing Company, 1989.

*Gosling et al. 96:* Gosling, J., Joy, B., and Steele, G. *The Java Language Specification.* Addison-Wesley Publishing Company, 1996.

*Howard 80:* Howard, W.A. *The formulae-as-types notion of construction.* In: Seldin, J.P. and Hindley, J.R. (Eds.). *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalisms.* Academic Press, New York and London, 1980, S. 479–490.

*Matthes et al. 97:* Matthes, F., Schröder, G., and Schmidt, J.W. *Tycoon: A Scalable and Interoperable Persistent System Environment.* In: Atkinson, M.P. (Ed.). *Fully Integrated Data Environments.* Springer-Verlag (to appear), 1997.

*Matthes, Schmidt 92:* Matthes, F. and Schmidt, J.W. *Definition of the Tycoon Language TL – A Preliminary Report.* Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

*Meyer 92:* Meyer, B. *Eiffel. The Language.* Prentice Hall, Englewood Cliffs, New Jersey, 1992.

*Meyer 97:* Meyer, B. *Object-oriented Software Construction, 2nd edition.* International Series in Computer Science. Prentice Hall, Englewood Cliffs, New Jersey, 1997.

*Nelson 91:* Nelson, G. (Ed.). *Systems programming with Modula-3.* Series in innovative technology. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

*Odersky, Wadler 97:* Odersky, M. and Wadler, P. *Pizza into Java: Translating theory into practice.* In: *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Januar 1997.

*Scott 72:* Scott, D. *Lattice Theory, Data Types and Semantics.* In: Rustin, Randall (Ed.). *Formal Semantics of Algorithmic Languages.* Prentice Hall, Englewood Cliffs, New Jersey, 1972, S. 65–106.

*Stroustrup 97:* Stroustrup, B. *The C++ Programming Language, 3rd edition.* Addison-Wesley Publishing Company, 1997.

*Wienberg 97:* Wienberg, A. *Studienarbeit: Bootstrap einer Persistenten Objektorientierten Programmierumgebung.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, August 1997.