Typüberprüfung in polymorphen Programmiersprachen: Aufgaben und Lösungsansätze

Diplomarbeit

eingereicht bei

Prof. Dr. Joachim W. Schmidt
und
Dr. Martin Lehmann

Universität Hamburg
Fachbereich Informatik
Datenbanken und Informationssysteme

von
Gerd Bremer
Rademachergang 3
20255 Hamburg

22. August 1996

"$F_{\leq}$'s score of dead and wounded is still rising"
Benjamin Pierce, Correspondence about $F_{\leq}$ from the Types list

"I don't really like de Bruijn numbers myself"
N.G. de Bruijn

Für meine Eltern und für meine Freundin Hanni.

Inhaltsverzeichnis

1. Einführung und Überblick **1**

1.1 Ausgangspunkt der Arbeit und Problemstellung 1

1.2 Aufgabenbeschreibung und Gliederung der Arbeit 3

2. Einordnung der Typüberprüfung im Compilerbau **5**

2.1 Das klassische Compilermodell 5

2.2 Architektur des Tycoon–Systems 11

3. Typen, Datenabstraktion und Polymorphismus **16**

3.1 Einfache Datentypen 16

3.2 Typkonstruktoren 18

3.3 Potenzmengen 24

3.4 Parametrischer Polymorphismus 26

3.5 Typoperatoren 27

3.6 Abstrakte Datentypen 28

3.7 Rekursive Typen 30

3.8 Dynamische Typen 33

4. Typüberprüfung in Lambda–Kalkülen **35**

4.1 Ungetypter Lambda–Kalkül 35

4.2 Einfach getypter Lambda–Kalkül 38

4.3 Getypter Lambda–Kalkül zweiter Ordnung 42

4.4 Getypte Lambda–Kalküle endlicher und unendlicher Ordnung 46

4.5 Getypter Lambda–Kalkül mit Subtypisierung 52

5. TLMIN **60**

5.1 Syntax der Sprache 60
5.1.1 Konkrete Syntax 60
5.1.2 Syntaxerweiterung zu TL 61
5.1.3 Normalisierung von TLMIN Programmen 62
5.1.4 Abstrakte Syntax 63
5.2 Bezeichner 67
5.2.1 De Bruijn–Indizierung 68
5.2.2 Objektreferenzierung 70
5.2.3 Qualifizierte Typbezeichner 72
5.2.4 Nichtabstrakte Typbezeichner 75
5.3 Subtypbeziehungen 76
5.3.1 Subtypisierung bei strukturierten Datentypen 76
5.3.2 Subtypisierung bei Typbezeichnern 77
5.3.3 Subtypisierung bei Typoperatorapplikationen 79
5.4 Subsignaturbeziehungen 82
5.4.1 Subsignaturregeln 82
5.4.2 Subsignaturregeln mit de Bruijn–Indizes 83
5.4.3 Typregeln für Tupel-, Record- und Ausnahmesignaturen 84
5.5 Substitution in Rückgabetypen 85
5.5.1 Substitutionsoperation 85
5.5.2 Nichtanonyme Funktionsargumente 86
5.6 Typdefinitionen in Programmblöcken 88
5.7 Rekursive Typen 89
5.7.1 Syntaxbaumrepräsentation 89
5.7.2 Wohlgeformtheit wechselseitig rekursiver Typdefinitionen 90
5.7.3 Kontraktive Typen 92
5.7.4 Zyklenerkennung 96
5.7.5 Subsignaturbeziehungen 103

6. Einsatz des neuentwickelten Typüberprüfungsalgorithmus **105**

6.1 Vergleich der Prüfzeiten in TL und TLMIN 106
6.2 Vergleich der ausgeführten Subtyptests 107

7. Fazit und Ausblick **109**

7.1 Fazit 109
7.2 Ausblick 112

A. Der untypisierte Lambda–Kalkül F_0 **114**

B. Der einfach typisierte Lambda–Kalkül F_1 **116**

C. Der getypte Lambda–Kalkül zweiter Ordnung F_2 **119**

D. Der typisierte Lambda–Kalkül höherer Ordnung F_∞ **122**

E. Der typisierte Lambda–Kalkül $F_\infty^\leq$ mit Subtypisierung **125**

F. Der typisierte Lambda–Kalkül $F_\infty^\leq$ ohne Sortennotation **130**

G. Algorithmische Typregeln **135**

H. TLMIN **136**

I. Messergebnisse **163**

Literaturverzeichnis **187**

1. Einführung und Überblick

1.1 Ausgangspunkt der Arbeit und Problemstellung

Am Arbeitsbereich Datenbanken und Informationssysteme (DBIS) der Universität Hamburg wurde die integrierende Datenbankprogrammierumgebung Tycoon[1] [Matthes 93; Matthes et al. 94] entwickelt, die einen verbesserten sprachlichen und architektonischen Rahmen für die Integration verschiedenartiger, unabhängig entwickelter generischer Dienste und Werkzeuge in einer offenen Systemumgebung bereitstellt. Zur Datenmodellierung, System- und Anwendungsprogrammierung wird im Tycoon–System die persistente, datenmodellneutrale, vollständige, polymorphe und strikt typisierte Programmiersprache TL[2] [Matthes, Schmidt 92; Mathiske et al. 93] eingesetzt, die im Kern auf einem Lambda–Kalkül höherer Ordnung mit Subtypisierung basiert. Die Sprache unterstützt:

- imperative, funktionale, relationale und objektorientierte Modellierungsparadigmen,
- die uniforme und orthogonale Benennung, Bindung und Typisierung der in datenintensiven Anwendungen auftretenden Dienste und Objekte,
- die Datenbanksystem– und Anwendungsprogrammierung durch Konzepte wie Polymorphismus, Iterations– und Persistenzabstraktion, Funktionen höherer Ordnung, Kontroll- und Datenstrukturen oder Modulbibliotheken, die die Definition, Integration, Erweiterung und Benutzung generischer Diensterbringer gestattet, und
- die Realisierung des Tycoon–Systems selbst, da der überwiegende Teil der Systemkomponenten (Parser, Typprüfer, Zwischencodegenerator) in TL implementiert ist.

TL ist eine typisierte Sprache. Typen definieren Restriktionen bzw. Zusicherungen und verhindern inkonsistente Interaktionen zwischen inkompatiblen Werten. Sie kategorisieren Wertobjekte gemäß Verwendungszweck, Struktur und Verhalten. Die Einhaltung von Typrestriktionen und die Lokalisierung von Typfehlern wird in TL durch einen Typüberprüfungsalgorithmus garantiert. Ein Typüberprüfungsalgorithmus sollte im Idealfall die drei folgenden Merkmale aufweisen:

- Alle typkorrekten Ausdrücke einer Programmiersprache werden als korrekt identifiziert (Vollständigkeit, *completeness*).

[1] Tycoon: TYped Communicating Objects in Open eNvironments
[2] TL: Tycoon Language

- Die als typkorrekt akzeptierten Ausdrücke sind in der Sprache typkorrekt (Korrektheit, *soundness*).
- Der Algorithmus terminiert für jeden zu prüfenden Ausdruck.

Die Theorie der Programmiersprachen zeigt, daß diese idealen Eigenschaften nicht immer erreichbar sind. Schon ein vergleichsweise einfach getyptes Lambda–Kalkül mit Subtypisierung und eingeschränktem parametrischem Polymorphismus zweiter Ordnung wie $F_{<:}$ [Ghelli 90; Curien, Ghelli 91] ist semi–entscheidbar [Pierce 92a]. Die Terminierung ist nur für typkorrekte Ausdrücke gesichert. Eine orthogonal um rekursive Typoperatoren erweiterte Sprache ist auch für typkorrekte Terme unentscheidbar [Solomon 78]. Vor diesem Hintergrund kommt der Definition von Typregeln und der Realisierung von Typüberprüfern eine besondere Rolle im Sprachentwurf zu.

Typregeln leiten typkorrekte Ausdrücke der Sprache induktiv aus typkorrekten Teilausdrücken ab und bilden so analog zur EBNF–Notation der Sprachsyntax ein Produktionensystem. Die Rückwärtsinterpretation einer Typregel entspricht einem Korrektheitsbeweis, so daß die Typüberprüferimplementierung als inverse Anwendung von Typregeln des Typsystems verstanden werden kann (siehe Abschnitt 2.1 auf Seite 10).

Die Vorteile einer formalen Sprachspezifikation gegenüber einer umgangsprachlichen Formulierung sind:

- Unklarheiten und Mehrdeutigkeiten der Sprache können bereits bei der Entwicklung erkannt und beseitigt werden.
- Eine formale Spezifikation erleichtert die Standardisierung einer Programmiersprache und unterstützt die Portierung von Programmen und Übersetzern auf unterschiedliche Rechnerarchitekturen.
- Wichtige Theoreme wie Entscheidbarkeit, Vollständigkeit und Korrektheit sind mathematisch ableitbar.

Da TL strikt typisiert ist, kann die Typüberprüfung statisch zur Übersetzungszeit erfolgen, um eine effiziente und von Typfehlern freie Laufzeitcodierung zu ermöglichen. Die Fehlersuche, Erweiterung und Anwendung des bereits implementierten Typüberprüfungsalgorithmus wird durch die folgenden Eigenschaften erschwert:

- Durch die Repräsentation von Bezeichnern durch de Bruijn–Indizes [de Bruijn 72; Abadi et al. 90] ist das Lesen und Verstehen des Typprüferprogrammcodes mühsam und das Aufspüren und Beseitigen von Fehlern schwierig.
- Der Aufwand für die Realisierung mächtiger Sprachkonzepte ist bei Verwendung der de Bruijn Notation aufgrund des niedrigen Abstraktionsniveaus umständlich und umfangreich. Dies wird besonders deutlich bei der Implementierung von dynamischen Typen [Geisler 95; Koehler 96] und rekursiven Typen.
- Die Programmcodierung ist teilweise unübersichtlich und weicht teilweise von der systematischen Sprachdefinition ab.
- Neben de Bruijn–Indizes erschweren komplexe Optimierungen das Verstehen der Implementierung (Caching, absolute Indizes bei Unifikationstypen).

1.2 Aufgabenbeschreibung und Gliederung der Arbeit

Die Ziele dieser Diplomarbeit sind:

- die Aufarbeitung des aktuellen Wissensstandes auf dem Gebiet der Implementierung von Typüberprüfungsalgorithmen für funktionale Programmiersprachen;
- die Untersuchung der systematischen Konstruktion von Typüberprüferalgorithmen basierend auf Typsystemen von typisierten Lambda–Kalkülen beliebiger Ordnung;
- die Analyse höhersprachlicher Konzepte wie Subtypisierung, Polymorphismus, Rekursion, Substitution, Typqualifizierung und Typinferenz hinsichtlich ihrer Auswirkungen auf den Typüberprüfungsalgorithmus;
- die Entwicklung einer uniformen Repräsentation statischer und dynamischer Typen in TL mit Hilfe von direkten Objektreferenzen statt de Bruijn–Indizes;
- die Integration der gewonnenen Ergebnisse in Form eines Typüberprüfungsalgorithmus für eine gegenüber TL im Umfang reduzierte Sprache TLMIN;
- ein Vergleich der beiden Typprüferimplementierungen und ihres Laufzeitverhaltens.

Die Arbeit gliedert sich neben der Einleitung in die im folgenden übersichtsartig vorgestellten Kapitel:

Einordnung der Typüberprüfung im Compilerbau: Dieses Kapitel beschreibt im ersten Teil das klassische Compilermodell nach [Aho et al. 88], das aus mehreren Übersetzungsphasen besteht, zu denen auch die statische Typüberprüfungsphase gehört. Für jede Übersetzungsphase der Compileranalyse bestehen formale Spezifikationsmethoden, die als Grundlage zur automatischen Generierung von Compilerkomponenten dienen. Eine wichtige formale Spezifikation der Typüberprüfungsphase ist das Typsystem, das aus einer Menge von Typregeln besteht. Der zweite Teil des Kapitels beschreibt die Architektur des Tycoon–Systems.

Typen, Datenabstraktion und Polymorphismus: Das dritte Kapitel stellt das *Ideal*–Modell vor, das eine Semantik für einfache Datentypen und Typkonstruktoren wie Tupel und Records definiert und das die Bedeutung von Typsystemen höherer Ordnung mit Subtypisierung, parametrischem Polymorphismus und Typoperatoren erklärt. Desweiteren werden abstrakte, rekursive und dynamische Typen angesprochen.

Typüberprüfung in Lambda–Kalkülen: TLMIN ist ein getyptes Lambda–Kalkül höherer Ordnung mit Subtypisierung. Dieses Kapitel untersucht einfachere getypte Lambda–Kalküle unterschiedlicher Mächtigkeit, bei deren Typüberprüfung bereits zahlreiche Aufgaben zu lösen sind. Zu jedem Kalkül ist die Syntax, das Typsystem und ein Typüberprüfungsalgorithmus aufgeführt. Es wird gezeigt, wie der Typüberprüfungsalgorithmus systematisch aus dem Typsystem abgeleitet werden kann.

Typüberprüfung in TLMIN: In diesem Kapitel werden die wichtigsten Aufgaben und Lösungsansätze bei der Typüberprüfung in TLMIN dargestellt. Hierzu gehört die Repräsentation von Bezeichnern durch Objektreferenzen, die Qualifizierung von Bezeichnern, der Subtyptest, der Subsignaturtest, die Instanziierung von Rückgabetypen, Typdefinitionen in Programmblöcken, die Repräsentation sowie der Subtyptest rekursiver Typen und der rekursive Polymorphismus (*f–bounded–polymorphism*).

Vergleich der Typprüferimplementierungen: Die Ausführungszeiten von TL und TLMIN Typüberprüfer werden verglichen und analysiert.

Fazit und Ausblick: Das abschließende Kapitel faßt die bei Typprüferimplementierung gewonnene Erfahrung zusammen und nennt die noch ausstehenden Aufgaben.

2. Einordnung der Typüberprüfung im Compilerbau

2.1 Das klassische Compilermodell

In diesem Abschnitt wird ein Compilermodell, das bekannte Phasenmodell nach [Aho et al. 88] vorgestellt. Es werden die einzelnen Phasen beschrieben und die zur Implementierung nutzbaren Compilerbau-Technologien präsentiert. Eine dieser Phasen beschreibt die Typüberprüfung.

Ein Compiler ist ein Programm, das ein in einer bestimmten Quellsprache geschriebenes Programm liest und es in ein semantisch äquivalentes Programm einer Zielsprache übersetzt. Diese Aufgabe läßt sich in verschiedene Teilaufgaben zerlegen, die hintereinander ausgeführt werden können. Die Abarbeitung einer Teilaufgabe wird als Phase bezeichnet. Jede Phase transformiert eine Repräsentation des Quelltextes in eine andere, wobei die letzte Repräsentation dem Zielprogramm entspricht.

Grundsätzlich kann der Übersetzungsprozeß, wie in Abbildung 2.1 dargestellt, in zwei Phasen mit drei Repräsentationen unterteilt werden.

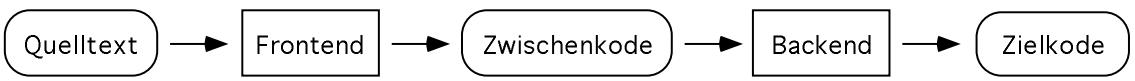


Abbildung 2.1: Grobstruktur eines Compilers

1. Die Analysephase (*Frontend*) zerlegt das Quellprogramm in seine Bestandteile und erzeugt eine maschinenunabhängige Zwischenrepräsentation des Quellprogramms.

2. Die Synthesephase (*Backend*) generiert aus dem Zwischencode das maschinenabhängige Zielprogramm.

Während der Analysephase werden Fehler im Quellprogramm lokalisiert und möglichst präzise Fehlermeldungen ausgegeben. Nur wenn im Frontend keine Fehler gefunden wurden, wird Zielcode generiert.

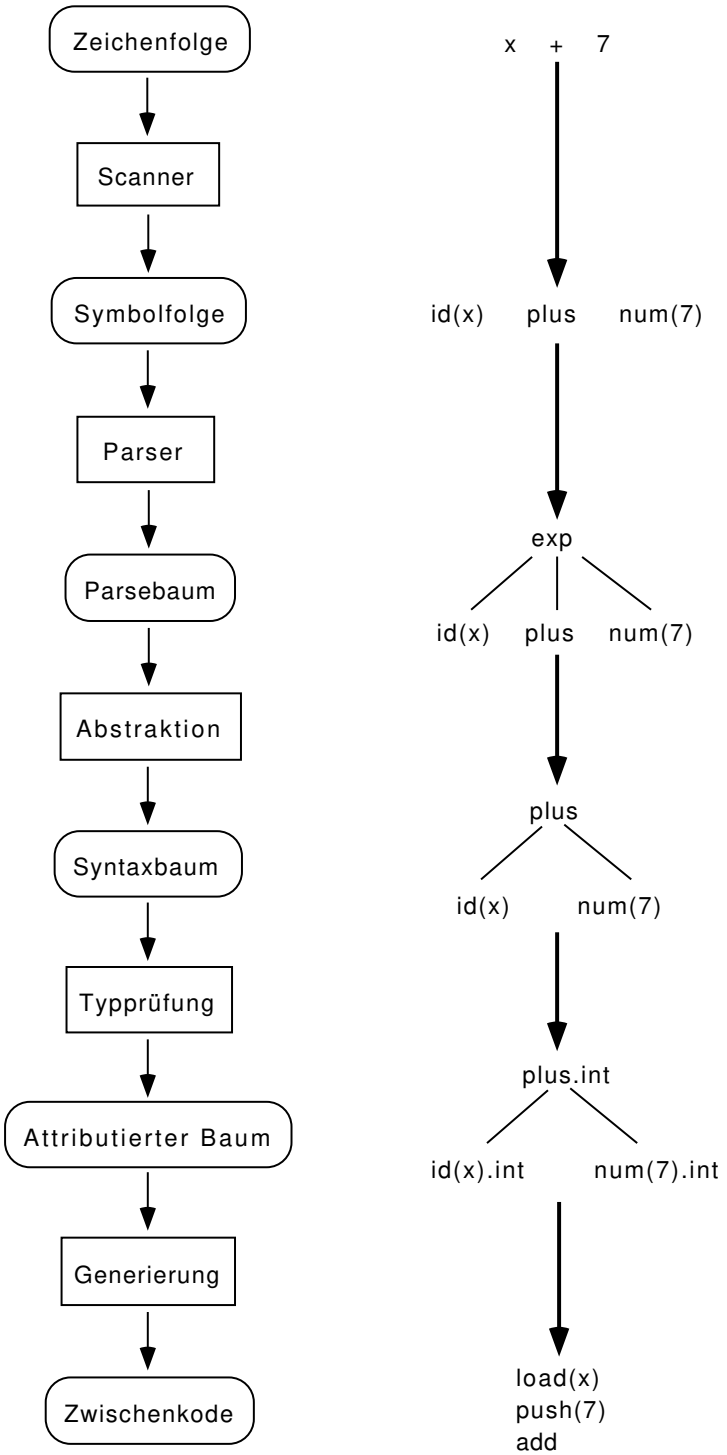


Abbildung 2.2: Feinstruktur der Analysephase des Compilers

Da die Typüberprüfung Teil der Analysephase ist, wird die Synthesephase in dieser Arbeit nicht weiter betrachtet. Abbildung 2.2 a) zeigt die einzelnen Phasen und Repräsentationen des Frontends. Abbildung 2.2 b) enthält ein Beispiel für die Transformation der Datenstrukturen.

Jede Übersetzungsphase kann als Funktion definiert werden.

$scan : Characters \rightarrow Symbols$
: Die lexikalische Analyse zerlegt den Quelltext, der als Zeichenreihe betrachtet wird, in eine Symbolfolge.

$parse : Symbols \rightarrow ParseTree$
: Die syntaktische Analyse analysiert die Symbolfolge auf syntaktische Konstrukte, die als Parsebaum repräsentiert werden.

$abstract : ParseTree \rightarrow AST$
: Durch die Abstraktion wird der Parsebaum in einen abstrakten Syntaxbaum (AST) transformiert.

$check : AST \rightarrow AST'$
: Die statische semantische Analyse überprüft, ob die Konstrukte semantisch korrekt verwendet werden, und fügt semantische Informationen als Attribute in den Syntaxbaum ein.

$generate : AST' \rightarrow IntermediateCode$
: Die Zwischencode-Erzeugung generiert aus dem attributierten Syntaxbaum einen Zwischencode, der vom Backend weiterverarbeitet wird.

Mit den beschriebenen Funktionen und einem Quelltext (*source*) als Eingabe läßt sich der logische Aufbau des Compilers funktional folgendermaßen beschreiben:

$$backend(frontend(source)) = backend(generate(check(abstract(parse(scan(source))))))$$

Diese Beschreibungsform betont, daß die einzelnen Phasen (Funktionen) nur vom Ergebnis der vorherigen Phase abhängen. In einer Implementation können aus Effizienzgründen mehrere Phasen zu einem Lauf zusammengefaßt und verschiedene Repräsentationen in einer Datenstruktur dargestellt werden.

In den folgenden Unterabschnitten werden den einzelnen Phasen formale Beschreibungstechniken zugeordnet. Formale Beschreibungen können Grundlage für die automatische Generierung von entsprechenden Compilerkomponenten sein.

Lexikalische Analyse

Die lexikalische Analyse (*scanning*) gruppiert Zeichenreihen zu Symbolfolgen. Die zulässigen Symbole einer Sprache werden mittels regulärer Ausdrücke spezifiziert. Ein Scanner-Generator erzeugt aus einer Menge von regulären Ausdrücken einen endlichen Automaten. Programme enthalten lexikalische Fehler, wenn sie vom Scanner nicht in Symbolfolgen umgewandelt werden können. Eine weitere Aufgabe des Scanners ist beispielsweise das Entfernen von Leerzeichen, Tabulatoren, Zeilenwechseln und Kommentaren.

Im folgenden werden Beispiele von Symboldefinitionen vorgestellt:

Symbol	regulärer Ausdruck	informelle Beschreibung
"if"	if	Schlüsselwort: **if**
"+"	+	Additionszeichen
int	$[0–9]^+$	ganze Zahl: mindestens eine Ziffer
id	$[a–zA–Z][a–zA–Z0–9]^*$	Bezeichner: Buchstabe gefolgt von Buchstaben und Ziffern

Syntaxanalyse

Die syntaktische Analyse (*parsing*) erzeugt aus den erlaubten Symbolfolgen einer Sprache Parsebäume. Die Menge der gültigen Parsebäume wird durch eine kontextfreie Grammatik formalisiert und konkrete Syntax genannt. Aus einer kontextfreien Grammatik erstellt ein Parser-Generator einen Parser. Symbolfolgen, die sich nicht gemäß der kontextfreien Grammatik zerteilen lassen, enthalten Syntaxfehler und repräsentieren keine korrekten Programme.

Eine kontextfreie Grammatik G enthält folgende Komponenten:

- Eine Menge N von nichtterminalen Elementen.
- Eine Menge T von terminalen Elementen, die der Scanner als Symbole der Sprache akzeptiert.
- Eine endliche Menge P von Produktionsregeln. Jede Produktionsregel ist von der Form $X \rightarrow \alpha$, wobei X ein nichtterminales Element und α ein Wort aus $(N \cup T)^*$ ist. Produktionsregeln mit dem gleichen Nichtterminal auf der linken Regelseite können zu einer Produktionsregel zusammengefaßt werden, indem die rechten Regelseiten, durch das Zeichen "|" getrennt, hintereinander dargestellt werden.
- Das Wurzel–Nichtterminal S der Grammatik.

Beispiel: Die kontextfreie Grammatik mit der folgenden Produktion definiert einfache arithmetische Ausdrücke.

$$valG \quad \rightarrow \quad int \mid id \mid valG \text{ "+" } valG$$

$valG$ ist das Wurzel–Nichtterminal. Die Nichtterminale int und id bezeichnen ganze Zahlen bzw. Bezeichner. Ein gültiger Ausdruck kann eine ganze Zahl, ein Bezeichner oder die Addition zweier Ausdrücke sein.

In Abbildung 2.2 b) ist ein Parsebaum für den arithmetischen Ausdruck $id + 7$ dargestellt.

Abstraktion

Eine durch Abstraktion komprimierte Form eines Parsebaums ist der abstrakte Syntaxbaum. Abstrakte Syntaxbäume werden durch Konstruktoren beschrieben, die als Parameter die Werte der Unterbäume bzw. Attribute des Knotens erhalten. Die Konstuktion eines Syntaxbaums

aus dem Parsebaum ist durch semantische Aktionen in den Produktionen der kontextfreien Grammatik darstellbar, wie das folgende Beispiel zeigt:

$$
\begin{array}{lcll}
valG & ::= & int & => intNode(int) \\
 & | & id & => idNode(id) \\
 & | & valG_1 \text{ "+" } valG_2 & => addNode(valG_1\ valG_2)
\end{array}
$$

Die Menge der wohlgeformten Syntaxbäume einer Sprache heißt abstrakte Syntax. Die abstrakte Syntax definiert die innere Struktur von Programmen unabhängig von Schlüsselwörtern oder anderen syntaktischen Konventionen. Sie bildet den Ausgangspunkt für die Definition der formalen statischen Semantik.

Semantische Analyse

Attributierte Grammatiken

Die Analyse der statischen Semantik läßt sich mit attributierten Grammatiken formulieren [Aho et al. 88]. Jeder Knoten des Syntaxbaums erhält weitere Attribute, deren Werte durch eine Menge von Berechnungsvorschriften in der Grammatik bestimmt werden.

Das folgende Beispiel zeigt eine attributierte Grammtik zur Typbestimmung von Wertausdrücken in einem abstrakten Syntaxbaum:

$$
\begin{array}{lcll}
val & ::= & intNode(int) & => val.type := integer \\
 & | & idNode(id) & => val.type := lookup(val.env\ id) \\
 & | & addNode(val_1\ val_2) & => val_1.env := val.env; \\
 & & & \quad val_2.env := val.env; \\
 & & & \quad if\ val_1.type = integer \\
 & & & \quad andif\ val_2.type = integer \\
 & & & \quad then\ val.type := integer \\
 & & & \quad else\ val.type := wrong \\
 & & & \quad end
\end{array}
$$

Die Knoten in dem Syntaxbaum enthalten zwei zusätzliche Attribute. *type* ist der Typ des entsprechenden Wertausdrucks. *env* ist eine Umgebung, in der die im Programmtext jeweils gültigen Bezeichner mit ihren Typen assoziiert sind. Der Knoten *intNode* hat den Typ *integer*. Die Funktion *lookup* sucht in der Umgebung von *val* nach dem Typ des Bezeichners *id*. Ist der Bezeichner nicht bekannt, wird der Typ *wrong* zurückgegeben, der das Auftreten eines Typfehlers markiert. Ein Typfehler liegt ebenfalls vor, wenn bei der Addition im *addNode* die beiden Argumente nicht vom Typ *integer* sind.

Bei den Attributen einer Grammatik unterscheidet man zwischen geerbten (*inherited*) und abgeleiteten (*derived*) Attributen eines Knotens. Der Wert eines abgeleiteten Attributs errechnet sich aus den Werten der Attribute der Nachfolger dieses Knotens im Syntaxbaum. Ein Beispiel ist *val.type* im obigen Beispiel, da der Typ der Addition aus den Typen der Argumente berechnet wird. Der Wert eines ererbten Attributs wird aus den Werten der Geschwister und des Vorgängers dieses Knoten berechnet. Das Attribute *val.env* ist ererbt, da es im *addNode*-Knoten an die Nachfolger weitergereicht wird.

Typsysteme

Ein formales System zur Spezifikation der statischen Semantik ist das Typsystem, das aus einer Menge von Typregeln zusammengesetzt ist [Matthes 93; Hennessy 90]. Jede Typregel ist durch eine horizontale Linie zweigeteilt. Oberhalb der Linie stehen die Prämissen, unterhalb die Konklusion. Die Prämissen können eventuell durch ergänzende mathematische Aussagen präzisiert sein. Eine Typregel wird durch einen eindeutigen Namen in eckigen Klammern benannt. Prämissen und Konklusion enthalten Aussagen (*judgements*) über Typen, Werte und deren Beziehungen. Die Aussagen sind von einem statischen Kontext (*environment*) S abhängig, einer geordneten Liste von Wert- und Typsignaturen. Die Liste enthält die in einem Ausdruck sichtbaren und verschatteten freien Bezeichner, wobei spätere Deklarationen rechts von früheren Deklarationen liegen.

Zwei Interpretationen einer Typregel sind möglich:

Vorwärtsinterpretation: Sind die Prämissen erfüllt, dann gilt die Konklusion (Produktionsregel). Aus einem Typsystem können alle möglichen typkorrekten Terme einer Sprache beginnend mit den Axiomen abgeleitet werden.

Rückwärtsinterpretation: Um die Konklusion zu beweisen, müssen alle Prämissen bewiesen werden (Beweisregel). Beginnend mit dem gesamten Programm als Konklusion ist ein Beweis im Typsystem zurück bis zu den Axiomen zu finden.

Das folgende Beispiel zeigt das Typsystem für einfache arithmetische Ausdrücke, das aus einem Axiom und vier Deduktionsregeln besteht:

[Sig empty]
$$\frac{}{\vdash \emptyset \ sig}$$

[Sig id]
$$\frac{\vdash S \ sig \qquad id \notin Dom(S)}{\vdash S, idNode(id) : Int \ sig}$$

[Value int]
$$\frac{\vdash S \ sig}{S \vdash intNode(int) : Int}$$

[Value id]
$$\frac{\vdash S, idNode(id) : Int, S' \ sig}{S, idNode(id) : Int, S' \vdash idNode(id) : Int}$$

[Value add]
$$\frac{S \vdash val_1 : Int \qquad S \vdash val_2 : Int}{S \vdash addNode(val_1 \ val_2) : Int}$$

Das Typsystem formalisiert die folgenden statischen Semantikregeln:

Sig empty: Die leere Signatur ist ein wohlgeformter statischer Kontext.

Sig id: Wenn ein statischer Kontext S wohlgeformt ist, dann ist S gefolgt von der Signatur $idNode(id) : Int$ ebenfalls eine zulässige Signaturfolge, vorausgesetzt der Bezeichner id ist nicht in S definiert.

Value int: In jedem wohlgeformten Kontext S hat der Knoten $intNode$ den Typ Int.

Value id: Ein Knoten $idNode$ hat den Typ Int, falls er im statischen Kontext enthalten ist.

Value add: Der Knoten $addNode$ besitzt im Kontext S den Typ Int, falls die beiden Knoten val_1 und val_2 im selben Kontext S den Typ Int haben.

Sofern alle Typregeln eines Typsystems eindeutig rückwärts interpretierbar sind, kann ein automatischer Beweiser implementiert werden, der den Wahrheitsgehalt einer Typaussage bestimmt. Der Beweiser versucht, die Aussage mit den Konklusionen der Typregeln des Typsystems zu vergleichen. Bei Übereinstimmung mit einer Typregel müssen gegebenenfalls Variablen in den Prämissen der Typregeln mit Ausdrücken aus der Konklusion unifiziert werden. Anschließend verifiziert der Beweiser die Prämissen der zutreffenden Typregel. Sind keine Prämissen vorhanden, ist die Aussage wahr. Besteht keine Übereinstimmung mit den Typregeln ist die zu beweisende Aussage falsch. Bei der Erstellung von Typsystemen ist zu beachten, daß Aussagen in Ausnahmefällen zu Nichtentscheidbarkeit, d.h. zu Nichtterminierung im Beweiser führen (siehe Abschnitt 4.5). Ein Typüberprüfungsalgorithmus ist ein automatischer Beweiser, der die Korrektheit von Typaussagen anhand der Typregeln eines Typsystems bestimmt.

Beginnend mit dem Axiom *[Sig empty]* unter Verwendung der Deduktionsregeln kann die Wohlgeformtheit der Aussage $idNode(x) : Int \vdash addNode(idNode(x)\ intNode(7)) : Int$ abgeleitet werden. Umgekehrt kann die Wohlgeformtheit der Aussage auf das Axiom zurückgeführt, also bewiesen werden.

$$\dfrac{\dfrac{\dfrac{}{\vdash \oslash\ sig}\ [Sig\ empty]}{\oslash \vdash idNode(x) : Int\ sig}\ [Sig\ id] \qquad \dfrac{\dfrac{\dfrac{}{\vdash \oslash\ sig}\ [Sig\ empty]}{\oslash \vdash idNode(x) : Int\ sig}\ [Sig\ id]}{idNode(x) : Int \vdash intNode(7) : Int}\ [Value\ int]}{idNode(x) : Int \vdash idNode(x) : Int}\ [Value\ id]$$

$$idNode(x) : Int \vdash addNode(idNode(x)\ intNode(7)) : Int\ [Value\ add]$$

Zwischencode-Erzeugung

Die Erzeugung des Zwischencodes aus dem Syntaxbaum läßt sich wie die Typüberprüfung syntaxgesteuert mit attributierten Grammatiken beschreiben. Der Zwischencode von arithmetischen Ausdrücken läßt sich durch folgende Regeln, die ein neues Attribut *val.code* verwenden, aus einem abstrakten Syntaxbaum generieren:

$$\begin{array}{lcll} val & ::= & intNode(int) & => \quad val.code := push(int) \\ & | & idNode(id) & => \quad val.code := load(val.env\ id) \\ & | & addNode(val_1\ val_2) & => \quad val_1.code := seq(val_1.code\ val_2.code\ add) \end{array}$$

Die Funktion *push* legt die Zahl *int* auf einem Keller ab. Die *load*-Funktion legt den Wert, der durch den Bezeichner *id* in der Umgebung *val.env* bestimmt ist, auf den Keller. *add* addiert die ersten zwei Elemente des Kellers und legt das Ergebnis dort wieder zurück. Eine Hinteranderausführung von Operationen wird durch die Funktion *seq* ausgedrückt.

2.2 Architektur des Tycoon–Systems

Die Architektur des Tycoon–Systems verfolgt das Ziel, ein hohes Maß an Skalierbarkeit und Interoperabilität durch die Integration von persistenten Daten, Programmen und Threads in einem offenen sprachlichen und architektonischen Rahmen zu erreichen [Matthes et al. 95b].

Die hohe Skalierbarkeit erlaubt es, Systemimplementierungen zu realisieren, die von einer Einzelimplementierung im Hauptspeicher auf einem Personal Computer bis zu einer mehrbenutzerfähigen, optimierenden und persistenten Implementierung in einem Netzwerk reicht. Anwendungen in Tycoon, die nur eine limitierte Systemfunktionalität benötigen, können auf einer effizienten und schlanken Systemversion arbeiten, die bei wachsenden operationalen Anforderungen erweiterbar ist. Die Systemoffenheit des Tycoon–Systems ermöglicht die Integration von externen generischen Diensten wie Fenstersystemen (OpenWindows, NeWS), Datenbanken (Ingres, Oracle) und Authentifikationsdiensten (Kerberos).

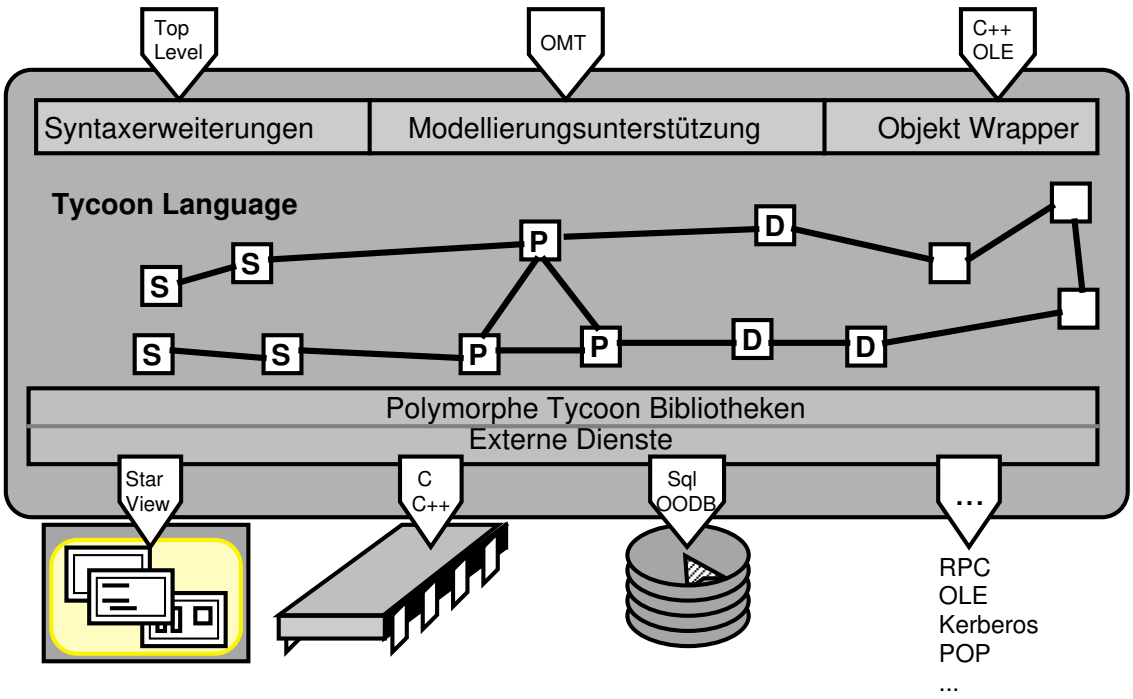


Abbildung 2.3: Interoperabilität im Tycoon–System

Ein hohes Maß an Interoperabilität wird erzielt, indem bestehende und neu entwickelte generische Diensten in uniformer Weise als polymorphe Tycoon–Bibliotheken im Tycoon–System integriert sind, die einen typsicheren Zugriff auf externe Daten und Programmcode sicherstellen (vgl. Abb. 2.3). Innerhalb von TL Anwendungen existieren uniforme Benennungs-, Typ-, Bindungs- und Lebensdauerregeln, z.B. für Bildschirm-, Daten- und Programmobjekte (S,D,P in Abb. 2.3), so daß integrierte Dienste von in Tycoon selbst implementierten Diensten nicht unterscheidbar sind und Typsicherheit und Persistenzabstraktion auch für externe Dienste vollständig erlangt wird. Wie durch die Pfeile am oberen Rand von Abb. 2.3 angedeutet wird, kann das Tycoon–System selbst als Subsystem in größere Systemen aufgerufen werden, die in anderen Sprachen, beispielsweise in C, geschrieben sind.

Abbildung 2.4 veranschaulicht die Trennung der Aufgaben Datenmodellierung, Datenmanipulation und Datenspeicherung in der Architektur des Tycoon–Systems durch ein Schichtenmodell.

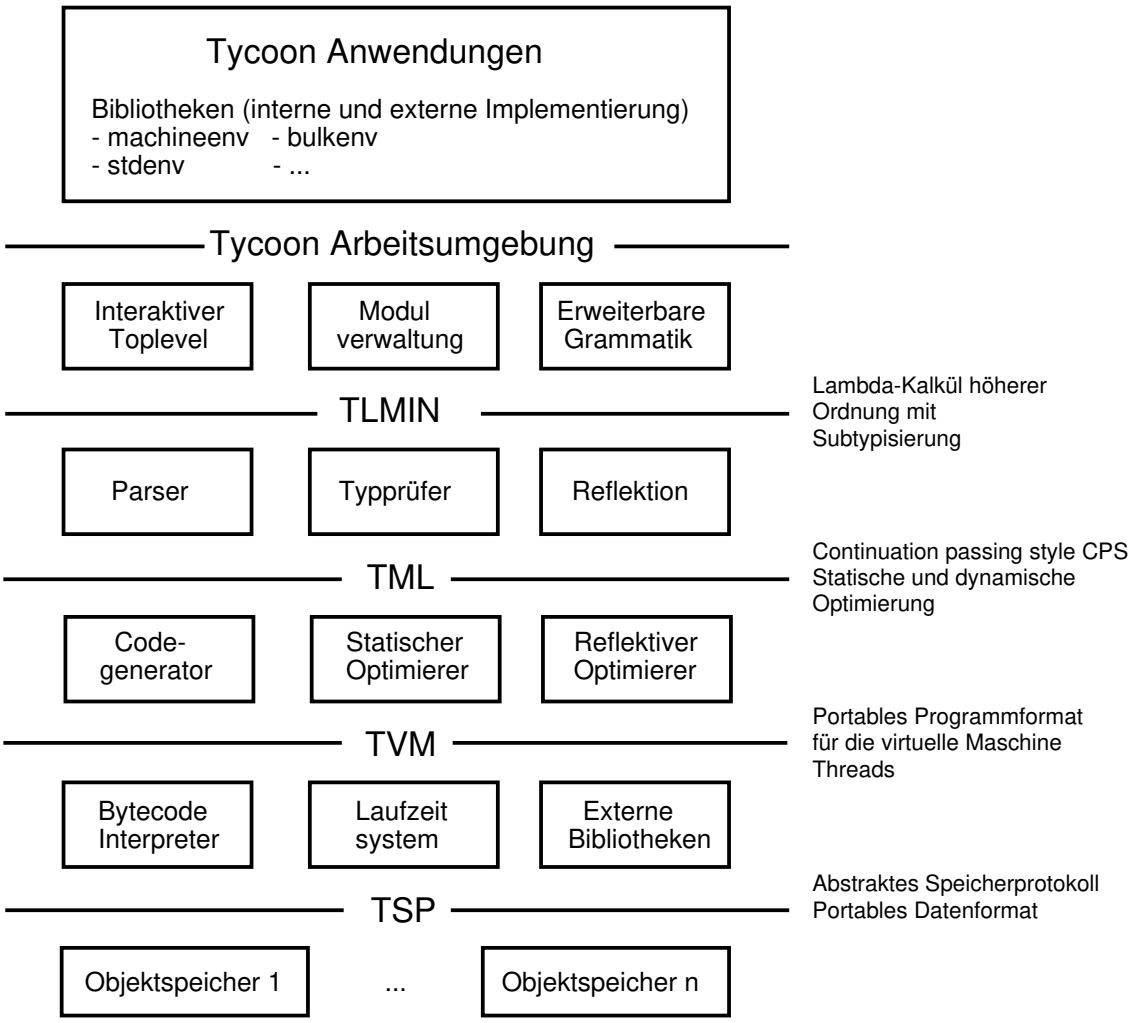


Abbildung 2.4: Architektur des Tycoon–Systems

Tycoon Arbeitsumgebung: Der interaktive Toplevel des Tycoon–Systems erlaubt den Zugriff auf sämtliche Werkzeuge und Dienste in der persistenten Programmierumgebung. Diese Dienste sind als Bibliotheken zugreifbar, die Schnittstellen und Modulkomponenten beinhalten. Gegenwärtig wird eine generische Modulverwaltung aufgebaut [**?**], die über dynamische Typisierung, Environments erster Klasse, Abhängigkeitskontrolle und Versionsmanagement verfügt. Um ein weites Spektrum von Daten und Exekutionsmodellen zu unterstützen, ist der Parser der Systemsprache durch das Konzept erweiterbarer Grammatiken modifizierbar [Cardelli et al. 94a; Cardelli et al. 94b], ohne den Compiler neu übersetzen zu müssen.

TLMIN Schnittstelle: Die Programmiersprache TLMIN ist eine polymorphe, strikt typisierte, persistente, im Kern funktionale Programmiersprache höherer Ordnung, die Funktionen und Typen als Objekte erster Klasse behandelt. TLMIN ist eine im Sprachumfang minimierte TL Variante. Die Syntax der Sprache ist mit Hilfe der erweiterbaren Gram-

matiken definiert und kann zum vollen TL Sprachumfang erweitert werden.

TML Schnittstelle: Die *Tycoon Machine Language* (TML) ist ein einfaches untypisiertes Lambda–Kalkül in *Continuation Passing Style*, das im Tycoon–System als quellsprachen– und zielmaschinenunabhängige Zwischenrepräsentation für statische und dynamische Optimierungen dient [Gawecki, Matthes 95]. Zu den wenigen Sprachkonstrukten gehören Wertbezeichner, Funktionsabstraktion, Funktionsapplikation und primitive Funktionsapplikation. Die Optimierer und der TVM Codegenerator sind in TL geschrieben.

TVM Schnittstelle: Die *Tycoon Virtual Machine* (TVM) ist eine Schnittstelle unterhalb der TML Schicht, die einen Satz von Bytecode Instruktionen für ein funtionales Exekutionsmodell höherer Ordnung definiert. TVM Bytecode kann entweder von der virtuellen Maschine interpretiert oder direkt in Maschinencode der Zielmaschine übersetzt werden. Der TVM Interpreter und das zugehörige Laufzeitsystem sind in ANSI–C geschrieben und auf verschiedenen Systemplattformen verfügbar (SunOs4.1.x, Solaris 2.x, Linux, Windows–NT und MacOs). Die Plattformunabhängigkeit des TVM Modells erlaubt es, Bytecode dynamisch in einem heterogenen Netzwerk zu transferieren. Dabei ist es möglich, persistente Threads über Systemgrenzen hinaus migrieren zu lassen [Mathiske et al. 95a; Mathiske et al. 95b; Matthes, Schmidt 95].

TSP Schnittstelle: Die TSP Schnittstelle ist als ANSI–C header file implementiert, das eine Teilmenge von Datenstrukturen und Funktionen exportiert, die von weit verbreiteten kommerziellen und public–domain Objektspeichern angeboten werden. Die Schnittstelle ist datenmodellunabhängig und abstrahiert von Implementierungen spezieller Objektspeicher. Ein TSP Klient kann frei zwischen verschiedenen Speichern wählen und mit mehr als einem persistenten Speicher gleichzeitig arbeiten. Um einen TSP Server zu installieren, müssen TSP Datenstrukturen und Funktionen auf Datenstrukturen und Operationen des jeweiligen Objektspeichers abgebildet werden. Dies ist bisher für den kommerziellen mehrbenutzerfähigen Speicher *ObjectStore* (*ObjectDesign*) und den wissenschaftlichen Prototypen *NapierStore* der Universität von St. Andrews geschehen. Desweiteren existieren zwei in der DBIS Gruppe entwickelte Speicherprototypen, die hauptsächlich für die Anwendung auf den PC geeignet sind. Eine detaillierte Beschreibung der TSP Schnittstelle findet sich in [Matthes et al. 95a].

Das Diagramm in Abb. 2.5 beschreibt den Datenfluß zwischen den einzelnen Schichten des Tycoon–Systems. Der Quellcode aus Programmdateien oder Benutzereingaben werden vom erweiterbaren Parser basierend auf der existierenden TLMIN oder TL Syntaxdefinition gelesen. Neue Grammatikdefinitionen werden vom Grammatikprüfer auf Korrektheit geprüft und in Parseregeln des erweiterbaren Parsers transformiert. Die vom Parser generierten Typ- und Wertausdrücke, die als abstrakte Syntaxbäume vorliegen, werden vom Typprüfer hinsichtlich ihrer statische Wohlgeformtheit geprüft. Die Wertsyntaxbäume dienen anschließend als Eingabe für die TML Zwischencodegenerierung. Die neu erzeugten TML Terme werden optimiert, bevor sie an die Zielcodegenerierung weitergereicht werden. Die TVM Maschine behandelt die Auswahl von zu interpretierenden Bytecode und direkt auszuführenden Maschinencode transparent. Um über Abstraktionsgrenzen hinweg zu optimieren, kann der TVM Generator ausführbaren TVM Code mit einer persistenten Bindung auf den zugehörigen TML Baum versehen. Zur Laufzeit kann so durch einen reflektiven Zugriff auf den Optimierer Programmcode weiter reduziert werden.

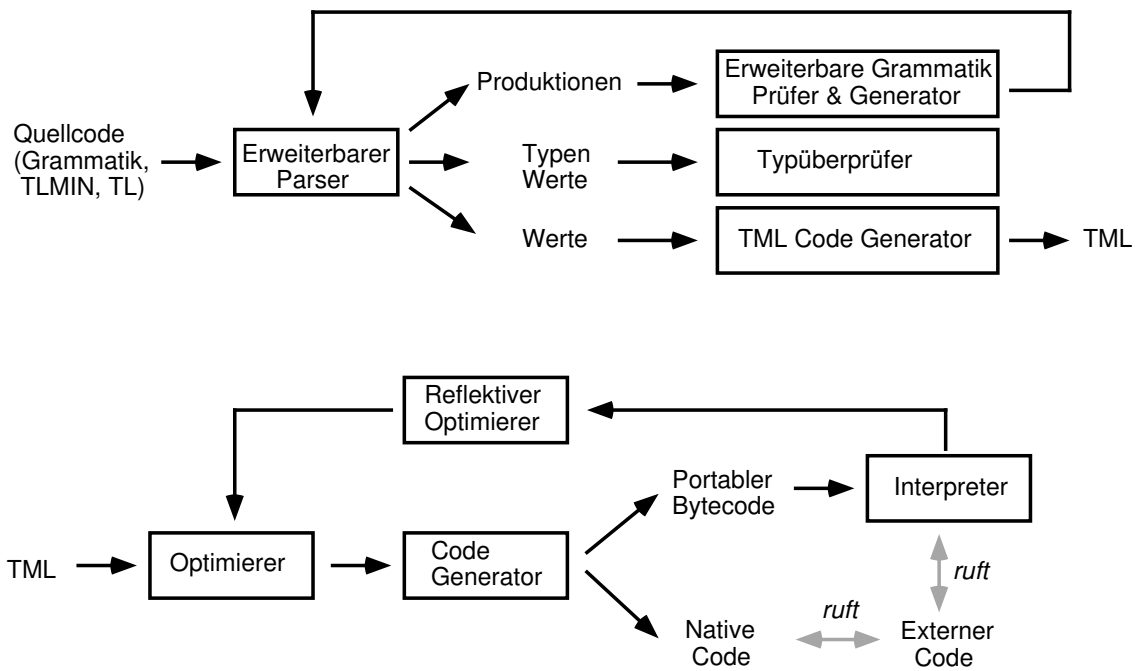


Abbildung 2.5: Datenfluß zwischen den Schichten des Tycoon–Systems

3. Typen, Datenabstraktion und Polymorphismus

Ein Datentyp ist eine Menge von Werten zusammen mit einer Menge von Operationen, die bestimmten Gesetzen gehorchen. Der Begriff Datentyp entspricht dem mathematischen Begriff der Algebra. Die Wertemenge eines Datentyps heißt Typ. Ein Wert x hat den Typ T, wenn x Element von T ist. Die Interpretation eines Typs als Menge von Werten mit gemeinsamen strukturellen Eigenschaften wird das *Ideal–Modell* genannt [Hindley 69; Milner 78; Damas, Milner 82; MacQueen et al. 86].

Im folgenden werden wichtige Typkonzepte betrachtet, die in Algol-ähnlichen Sprachen (Pascal, Algol68, C, Modula-2, Ada, TL) enthalten sind.

3.1 Einfache Datentypen

Typen, deren Werte atomar und nicht weiter strukturierbar sind, bezeichnet man als einfach. Zu jeder Sprache gehört ein Satz vordefinierter einfacher Typen, aus denen alle anderen Typen hergeleitet werden. Dazu gehören im allgemeinen folgende Typen:

Bool: Wahrheitswerte aus der Menge { true, false }.

Int: Die Integerzahlen, eine Menge von ganzen Zahlen.

Char: Eine Menge von Buchstaben.

Real: Eine Menge von reelwertigen Zahlen.

Die Werte dieser Typen können direkt auf Maschinenebene interpretiert werden. Nicht selten umfassen die vordefinierten Typen auch numerische Typen mit festgelegter Genauigkeit, wie *longreal*, *double*, *longint*, *short*, *shortreal*. Einige Sprachen besitzen unterschiedliche Formate für Real-Typen, z.B. *fixed* und *float*. Es können spezielle Integer-Bereiche definiert sein, wie *unsigned int* oder *Cardinal*.

Als Operationen auf diesen Typen stehen vordefinierte Funktionen zur Verfügung. Beispiele aus TL:

not(x : Bool) :Bool
/\, \/, <+>, => (x,y :Bool) :Bool
<, >, <=, >= (x,y :Int) :Bool
*+, –, *, /, %, ↑ (x,y :Int) :Int*

Es gibt einfache Typen, die nicht vordefiniert sind. Aufzählungstypen sind Mengen, deren Elemente bei der Spezifikation explizit benannt und aufgeführt werden. Es folgt ein Beispiel in Pascal.

```
type Colors = (red, blue, green)
```

Da Aufzählungstypen durch die Position ihrer Elemente geordnet sind, gibt es partielle Nachfolger- und Vorgänger-Operationen:

```
succ(c :Colors) :Colors
pred(c :Colors) :Colors

succ(red) = blue
pred(red) ⇒ Laufzeitfehler
```

Unterbereichstypen sind zusammenhängende Teilmengen einfacher Typen, die durch Angabe eines kleinsten und eines größten Elements spezifiziert werden. Die Typen *Digit* und *Byte* werden in Pascal folgendermaßen deklariert:

```
type    Digit   = 0..9;
        Byte    = 0..255;
```

Die Basistypangabe des Unterbereichs, im Beispiel der Integer-Typ, fehlt und muß vom Übersetzer bestimmt werden. Als Basistypen sind nur Mengen zulässig, auf denen eine Ordnungsrelation definiert ist (Ordinaltypen). Der Übersetzer kann die Elementbeziehung zwischen Werten und Bereichstypen nicht statisch prüfen. Sie muß dynamisch zur Laufzeit erfolgen, da diese Werte als beliebige Ausdrücke auftreten können.

Einfache Datentypen in TL: In TL sind weder Aufzählungstypen noch Bereichstypen als einfache Datentypen vorhanden. Aufzählungstypen sind mit varianten Tupeltypen darstellbar. Beispielsweise wird der Typ *Colors* in TL folgendermaßen definiert:

```
Let Colors = Tuple case red, blue, green end
```

Als Alternative zu Bereichstypen können abstrakte Datentypen definiert werden, die die Einhaltung der Bereichsgrenzen im Programmcode garantieren. Beispiel:

```
Let SubInt = Tuple
    T <:Int
    error :Exception end
    test(:Int) :T
end

let new(from,to :Int) :SubInt = tuple
    Let T = Int
    let error = exception "SubInt error" end
    let test(i :Int) :T =
```

```
        if i >= from andif i <= to then i
        else raise error end
        end
    end

    let digit :SubInt = new(0 9)
    let x :digit.T = digit.test(5)
    let y :digit.T = digit.test(x + x)  ⇒ exception "SubInt error" end
```

Der abstrakte Datentyp $SubInt$ definiert ganzzahlige Unterbereiche. Die Datenstruktur enthält die abstrakte Typvariable T, die als Subtyp von Int spezifiziert ist, eine Ausnahmekomponente $error$ und eine Funktion $test$, die Werte aus Int auf Werte aus T abbildet. Die Funktion new nimmt als Argumente die Unterbereichsgrenzen $from$ und to und gibt einen Wert vom Typ $SubInt$ zurück. Der Unterbereich $digit$ enthält die ganzzahligen Werte von 0 bis 9. Die Funktion $digit.test$ angewendet auf die Zahl 5 liefert 5 vom Typ digit.T zurück, während der Aufruf von $digit.test(x + x)$ mit $x = 5$ zu einer Ausnahme evaluiert, da der erlaubte Unterbereich überschritten ist.

3.2 Typkonstruktoren

In Programmiersprachen werden neue Typen mit Typkonstruktoren aus vorhandenen Typen gebildet. Da Typen Mengen modellieren, ist ein Typkonstruktor als Mengenkonstruktor zu interpretieren. Konstruktoren erzeugen Produktmengen, Vereinigungsmengen, Teilmengen, Potenzmengen und Funktionsmengen.

Produktmengen

Produktmengen lassen sich unterteilen in

- kartesische Produkte (Tupel) und
- benannte kartesische Produkte (Records).

Gegeben seien Mengen U_i mit $i = 1 \ldots n$. Als kartesisches Produkt $U_1 \times \ldots \times U_n$, auch Kreuzprodukt genannt, wird die Menge der geordneten Elementtupel von U_i bezeichnet, kurz:

$$U_1 \times \ldots \times U_n = \{(u_1, \ldots, u_n) \mid u_i \in U_i, i = 1 \ldots n\}$$

Beispiel:

```
    let t :Int × Bool = (3,true)
```

Als Operationen für Tupelwerte existieren Selektionsfunktionen, die den Zugriff auf die einzelnen Komponenten über die entsprechende Position ermöglichen.

```
    first(t)    ⇒ 3
    second(t)   ⇒ true
```

Ein Record ist eine ungeordnete Menge von benannten Werten. Ein Recordtyp wird durch die Angabe von Bezeichnern und ihren zugehörigen Typen spezifiziert.
Beispiel:

Let *R* = {a :*Int*, *b* :*Bool*}
let *r* :*R* = {a = *4*, *b* = *false*}

Die Selektion von Recordkomponenten erfolgt nicht über die Position, sondern über den eindeutigen Namen der Komponente.

r.a ⇒ *4*
r.b ⇒ *false*

Während die Typen

{a :*Int*, *b* :*Bool*} *und* {*b* :*Bool*, a :*Int*}

die gleiche Menge von Werten repräsentieren, bezeichnen die Typen

Int × *Bool* *und* *Bool* × *Int*

zwei unterschiedliche disjunkte Mengen.
Produktmengen in TL: In TL wird ein Tupeltyp mit dem Schlüsselwort **Tuple** und ein Tupelwert mit dem Schlüsselwort **tuple** definiert. Recordtypen und Recordwerte werden mit den Schlüsselwörtern **Record** bzw. **record** eingeleitet. Der Zugriff auf Tupelkomponenten erfolgt wie bei Recordkomponenten über den Namen.

Let *T* = ***Tuple*** a :*Int* *b* :*Bool* ***end***
let *t* :*T* = ***tuple*** ***let*** a = *7* ***let*** *b* = *true* ***end***
t.a ⇒ *7*
t.b ⇒ *true*

Dies hat zwei Konsequenzen:

1. Wenn Tupelwerte mit anonymen Komponenten definiert werden, ist eine spätere Selektion nicht direkt möglich. Im folgenden Beispiel kann auf den Wert der Tupelkomponente von *a* nicht zugegriffen werden. Dies ist jedoch indirekt über eine Wertzuweisung von *a* an einen Tupelwert *b* und dem Zugriff auf die benannte Komponente von *b* möglich:

   Let *Anonym* = ***Tuple*** :*Int* ***end***
   let a :*Anonym* = ***tuple*** *1* ***end***
   let *b* :***Tuple*** *x* :*Int* ***end*** = a

   a.*?*
   b.x ⇒ *1*

2. Da die Namen in einem Tupel nicht eindeutig sein müssen, ist nur die Selektion über den Bezeichner möglich, der zum Zeitpunkt der Selektion sichtbar ist.

   Let *Double* = ***Tuple*** a :*Int* a :*Bool* ***end***
   let *d* :*Double* = ***tuple*** ***let*** a = *1* ***let*** a = *false* ***end***
   d.a ⇒ *false*

Die obigen Anmerkungen zu TL beziehen sich nur auf die Sprachsyntax. Im TL-Compiler werden Tupelkomponenten aus Gründen der Effizienz wieder mit Positionen repräsentiert.

Vereinigungsmengen

Wie Produktmengen können auch Vereinigungsmengen auf zwei Arten in Abhängigkeit von der Zugriffsmethode, d.h. geordnet über Position oder ungeordnet über Namen, gebildet werden. Man unterscheidet

- variante Tupel und
- variante Records

Gegeben seien Mengen U_i, $i = 1 \ldots n$. Ein variantes Tupel $U_1 \cup \ldots \cup U_n$ ist die Vereinigung der Elemente aus U_i, kurz:

$$U_1 \cup \ldots \cup U_n = \{u \mid u \in \bigcup U_i\}.$$

Beispiel:

```
let v :Int ∪ Bool = 1
```

Als Operation für variante Tupel ist nur die Variantenselektion vorhanden, für die die Ordnung der Komponenttypen in der Vereinigungsmenge Int ∪ Bool entscheidend ist.

```
case v
when firstvariant then 2 > v
when secondvariant then v
end
```

Obwohl die Typen

Int ∪ Bool *und* *Bool ∪ Int*

mathematisch die gleiche Wertemenge beschreiben, müssen sie dennoch wegen der benötigten Ordnung unterschieden werden, was dem Ideal–Modell widerspricht. Somit sollte das Konzept varianter Tupel in einer Programmiersprache ausgeschlossen werden.

Bei varianten Records erfolgt die Variantenselektion wie bei gewöhnlichen Records über eindeutige Bezeichner. Durch Vertauschen von ungeordneten Komponenten in varianten Records ändert sich der Typ nicht.

```
Let V = [a :Int  b : Bool]
let v :V = [a = 1]

case v
when a then a > 2
when b then b
end
```

Vereinigungsmengen in TL: Aufgrund von Effizienzüberlegungen können in TL variante Tupel definiert werden. Zudem sind die Komponenttypen immer Tupeltypen. Diese weniger elegante Eigenschaft bedeutet aber keine Einschränkung der Sprachmächtigkeit, da einfache Typen bijektiv auf Tupeltypen mit einer Komponente abgebildet werden können.

Es folgt ein Beispiel für variante Typeltypen, variante Tupelwerte und die Variantenselektion in TL:

```
Let T = Tuple
            case firstvariant with i :Int
            case secondvariant with b :Bool
        end
let v = tuple_case firstvariant of T with
            let i = 1
        end

case v
when firstvariant with l then 2 > l.i
when secondvariant with l then l.b
end
```

Variante Records sind in TL nicht implementiert.

Funktionsmengen

Die Menge der Funktionen $f : U \rightarrow V$ kann als

- Funktionstyp oder als
- Feldtyp (Arraytyp)

erzeugt werden. Wenn U ein Ordinaltyp ist, entspricht f einem Array mit dem Indextyp U und dem Komponententyp V. Die ganze Funktion kann durch die Folge ihrer Werte $f(low) \ldots f(high)$ dargestellt werden, wobei low das kleinste Element in U repräsentiert und $high$ das größte. Das Beispiel

```
TYPE        Digit       = [0..9];
          DigitToREAL   = ARRAY Digit OF REAL;
```

definiert eine Abbildung von $Digit$ auf $REAL$ in Modula-2.

Funktionstypen werden durch Angabe der Argumenttypen und des Rückgabetypen spezifiziert.

Funktionsmengen in TL:

In TL wird ein Feldtyp mit dem vordefinierten Typoperator *Array* definiert. Beim Aufruf des Typoperators *Array* ist nur die Angabe des Komponententyps möglich:

```
Let IntArray = Array(Real)
```

Der Indextyp ist ein Integerbereich, der mit der Zahl 0 beginnt und dessen Größe beim Anlegen eines neuen Feldes angegeben werden muß. Die Korrektheit der Feldindizierung wird dynamisch überwacht.

Funktionstypen werden in TL mit dem **Fun**–Konstruktor gebildet. Zum Beispiel beschreibt die Definition

Let *IntFun* = ***Fun***(*x :Int y :Int) :Int*

einen Funktionstyp, der zwei Integerargumente x und y auf einen Integerwert abbildet.

Teilmengen

Sind $T1$ und $T2$ Typen mit der Eigenschaft $T1 \subset T2$, so ist $T1$ ein Subtyp von $T2$, kurz $T1 <: T2$. Hat der Wert x den Typ $T1$ und ist $T1 <: T2$, so hat x ebenfalls den Typ $T2$ (Subsumptionsprinzip). Ein Wert vom Typ $T1$ kann daher überall typkorrekt verwendet werden, wo ein Wert vom Typ $T2$ erwartet wird. Die allgemeingültige Definition der Subtypbeziehung läßt sich für die verschiedenen Typkonstruktoren einer Programmiersprache präzisieren. Die Subtypisierung von Typkonstruktoren beruht auf *struktureller* Kompatibilität und betrifft

- Unterbereichstypen,
- Funktionstypen,
- Tupel,
- Records und
- Varianten.

Für Bereichstypen gilt die triviale Subtypbeziehung:

$$n..m <: n'..m' \iff n' \leq n \quad \textit{und} \quad m \leq m'$$

Im folgenden Beispiel ist die Funktionsanwendung $f(3)$ typkorrekt, da die Konstante 3 den Typ 3..3 hat und 3..3 <: 2..5 gilt.

```
let f :Fun(x :2..5) :3..6 = fun(x :2..5) x + 1
f(3)
```

Die Kontravarianzregel definiert die Subtypbeziehung von Funktionstypen. Die Parametertypen stehen in kontravarianter, die Rückgabetypen in kovarianter Beziehung. Beispiel:

Fun(*x :1..9) :4..5 <: **Fun**(x :2..5) :3..6*

Die Kontravarianz der Parametertypen ist notwendig. Sprachen, die eine kovariante Beziehung der Parametertypen vorschreiben, sind typunsicher [Cook 89]. So erzeugt der Aufruf der Funktion *test* mit dem Argument f im unteren Beispiel einen Typfehler.

Let *F* = ***Fun***(*x* :***Tuple*** *i* :*Int* ***end***) :*String*
let *test*(*f* :*F*) = *f*(***tuple let*** *i* = *1* ***end***)
let *f* :*F* = ***fun***(*x* :***Tuple*** *i* :*Int s* :*String* ***end***) *x.s*
test(f) ⇒ *Laufzeitfehler*

Im Rumpf der Funktion *test* wird die Funktion *f* mit einem Tupelwert aufgerufen, der nur die Komponente *i* enthält. Die Funktion *f* erwartet aber Tupelwerte mit einer zusätzlichen Stringkomponente *s*.

Die Subtypbeziehung für Tupeltypen ist durch die Subtypbeziehung der Komponenttypen festgelegt. Im nächsten Beispiel gilt $T <: S$, da *Int* und *String* Subtypen von **Ok** sind:

Let *T* = ***Tuple*** a :*Int* *b* :*String* ***end***
Let *S* = ***Tuple*** a :***Ok*** *b* :***Ok*** ***end***

Es fällt auf, daß ein Hinzufügen weiterer Komponenten am Ende des Tupelsubtyps keine Typfehler produziert. So kann der Typ Car im folgenden TL–Beispiel überall verwendet werden, wo der Typ $Vehicle$ erwartet wird.

Let *Car* = ***Tuple*** *age* :*Int* *speed* :*Int* *fuel* :*String* ***end***
Let *Vehicle* = ***Tuple*** *age* :*Int* *speed* :*Int* ***end***

Diese Subtypbeziehung widerspricht jedoch der Interpretation eines Tupeltyps als geordnetes kartesisches Produkt mit Elementen gleicher Länge. Damit die Subtyprelation zwischen Car und $Vehicle$ gilt, muß die Wertemenge, die den $Vehicle$ Typ beschreibt, alle Werte aus Car enthalten. $Vehicle$ ist demnach die Menge der geordneten Tupelwerte, die *mindestens* zu Anfang zwei Integerkomponenten besitzen.

Entsprechendes trifft auf Recordtypen zu. Wegen der fehlenden Ordnung können weitere Komponenten in Subtypen an beliebiger Stelle eingestreut werden. Im nachfolgenden Beispiel ist $Car2$ ein Subtyp von $Vehicle2$.

Let *Car2* = ***Record*** *age* :*Int* *fuel* :*String* *speed* :*Int* ***end***
Let *Vehicle2* = ***Record*** *age* :*Int* *speed* :*Int* ***end***

In [Cardelli, Wegner 85] wird der Typkonstruktor & eingeführt, um zwei Records zu konkatenieren:

Let *Car3* = *Vehicle* & ***Record*** *fuel* :*String* ***end***

Die Namen der Komponenten müssen disjunkt sein. TL ermöglicht die Bildung von Recordsubtypen mit Hilfe des Repeat-Konstrukts, das Signaturen aus strukturierten Typen extrahiert.

Let *Car4* = ***Record*** *Repeat Vehicle* *fuel* :*String* ***end***

Der Subtyp von Varianten enthält im Gegensatz zum Tupelprodukt weniger Komponenten. Als Nachteil erweißt sich die Ordnung auf den Komponenten. In einem Subtyp können nur diejenigen Komponenten auftreten, die ein Präfix der Supertypkomponenten darstellen.

So gilt $X <: Y$ im folgenden Beispiel:

Let $X = Int \cup String$
Let $Y = Int \cup String \cup Bool$

Die TL Varianten $Weekday$ und $Weekend$ stehen somit nicht in Subtypbeziehung.

```
Let Weekday = Tuple
        case mon, tue, wed, thu, fri, sat, sun
        end

Let Weekend = Tuple
        case sat,sun
        end
```

Als Lösung bieten sich wieder variante Records an, die in TL nicht existieren.

3.3 Potenzmengen

Als Potenzmenge einer Menge A wird die Menge aller Teilmengen von A bezeichnet. Zwei Interpretationen einer Potenzmenge und ihrer Teilmengen sind denkbar. Eine Möglichkeit ist, die Potenzmenge als Typ aufzufassen. Die Teilmengen sind die zugehörigen Werte. Ein Beispiel sind Set-Typen in Modula-2:

```
TYPE     Digit    = [0..9];
         DigitSet = SET OF Digit;

CONST    digitSet = DigitSet{0,5,9};
```

Mengen werden in Modula-2 durch Aufzählung ihrer Elemente angegeben. Auf Mengen können Operationen wie Vereinigung, Differenz, Durchschnitt und symmetrische Differenz vorgenommen werden. Weiterhin existieren Vergleichsoperatoren und ein Relationsoperator IN, der überprüft, ob ein Wert in einer Menge enthalten ist.

Eine zweite Möglichkeit interpretiert die Teilmengen einer Potenzmenge als Typen. Zusätzlich zur Werte- und Typebene kommt hierbei die Ebene der Typmengen, auch Sorte (*kind*) genannt, hinzu.

Abbildung 3.1 stellt das Ebenendiagramm von Quest dar. Die Pfeile innerhalb der Abbildung stehen für die Beziehungen zwischen Werten und Typen bzw. Typen und Sorten. So hat der Wert 3 den Typ Int, kurz: $3 : Int$ und der Typ Int die Sorte **TYPE**, kurz: $Int :: \mathbf{TYPE}$. **TYPE** ist in Quest die Typmenge ohne Typoperatoren. Der Sortenoperator **POWER** bildet aus einem Typ B die Potenzmenge von B. Es gilt $B :: \mathbf{POWER}(B)$ und, falls $A <: B$ ist, auch $A :: \mathbf{POWER}(B)$. Die Aussage $\mathbf{TYPE} :: \mathbf{TYPE}$ führt zu theoretischen [Meyer, Reinhold 86] und praktischen Problemen [Cardelli 87] und macht eine Unterscheidung sowohl zwischen Typen und Sorten als auch zwischen Werten und Typen unmöglich [Cardelli 88a; Cardelli 88b]. Sie ist in Quest nicht zulässig.

TL hat das Konzept der Sorten implizit übernommen. Anstelle von $A :: \mathbf{POWER}(B)$ wird die dazu isomorphe Subtypsignatur $A <: B$ benutzt. Die explizite Sortennotation entfällt. Mittels Subtypsignaturen ist in TL die universelle und existentielle Quantifizierung von Typvariablen realisiert, die konkret die Definition von

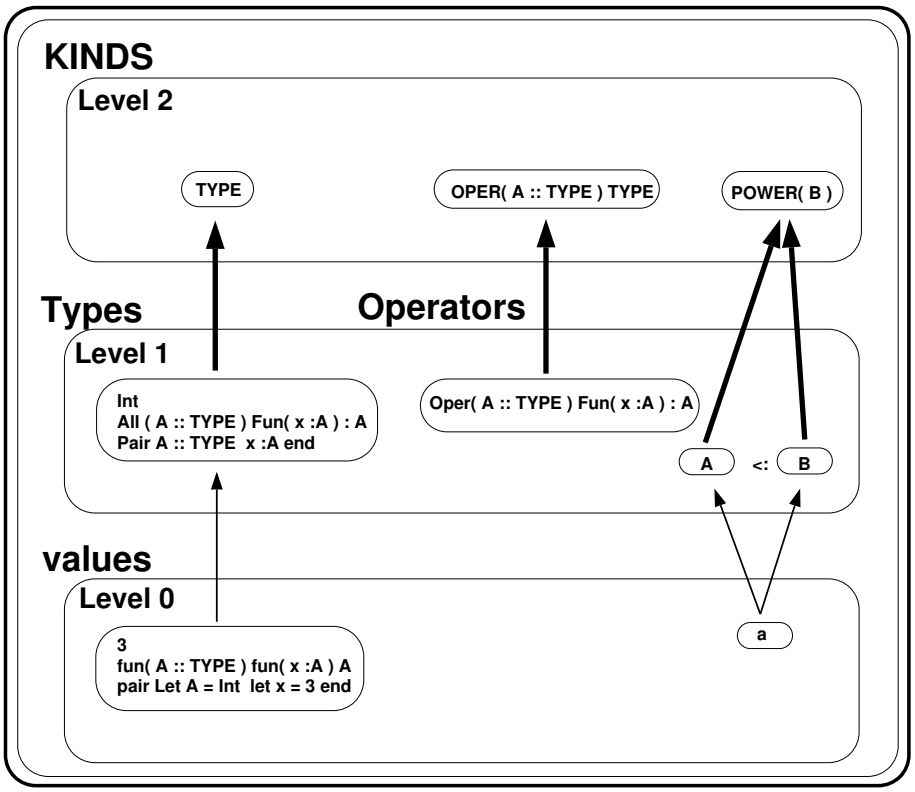


Abbildung 3.1: Das Ebenendiagramm von Quest

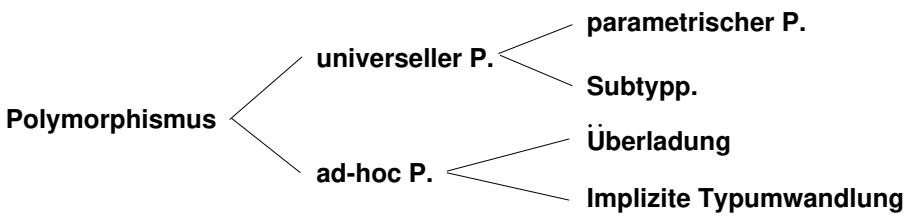


Abbildung 3.2: Arten von Polymorphismus

- parametrisch polymorphen Funktionen,
- Typoperatoren und
- abstrakten Datentypen

erlaubt. Die Bedeutung dieser Typen wird in den folgenden Abschnitten vorgestellt.

3.4 Parametrischer Polymorphismus

Polymorphismus heißt Vielgestaltigkeit. In Programmiersprachen ermöglicht Polymorphismus ein von speziellen Typen unabhängiges Programmverhalten zu definieren. Abbildung 3.2 unternimmt eine Klassifizierung von Polymorphismus gemäß [Cardelli, Wegner 85].

Parametrischer Polymorphismus bezeichnet ein uniformes, von beliebigen Typparametern unabhängiges Verhalten von Funktionen.

Subtyppolymorphismus ermöglicht ein typkorrektes Verhalten auf Werten mit beliebigem Subtyp.

Überladung erlaubt es, für eine endliche Anzahl von Typen unterschiedliche Funktionalität unter einem Begriff zusammenzufassen.

Implizite Typumwandlung wird ausgeführt, wenn der Typ der aktuellen Funktionsargumente nicht mit dem Typ der formalen Parameter übereinstimmt.

Überladung und die implizite Typumwandlung bilden zusammen den ad-hoc Polymorphismus, der hier nicht weiter ausgeführt wird, weil dieses Konzept das Verständnis einer Programmiersprache übermäßig erschwert und deshalb in TL nicht realisiert ist.

Der universelle Polymorphismus setzt sich aus dem parametrischen und dem Subtyppolymorphismus zusammen. Das Konzept des Subtyppolymorphismus wird in Abschnitt 3.2 vorgestellt. Der Vorteil polymorpher gegenüber monomorph typisierten Sprachen ist eine größere Flexibilität beim Programmentwurf ohne Aufgabe der strikten Typisierung.

Parametrisch polymorphe Funktionen in TL sind Funktionen, die Typvariablen als Parameter besitzen, die bei der Funktionsapplikation gebunden (instanziiert) werden. Eine Typvariable ist dabei durch die Angabe eines Supertypen eingeschränkt. Die Funktion *poly* im folgenden Beispiel bezeichnet die polymorphe Identitätsfunktion, die mit verschiedenen Typen und entsprechenden Werten aufgerufen werden kann.

```
Let Poly = Fun(A <:Ok  a :A) :A
let poly :Poly = fun(A <:Ok  a : A) :A a
poly(:Int 1)          ⇒  1
poly(:String  "a")    ⇒  "a"
```

Eine wichtige Rolle bei der Verwendung von Subtypsignaturen spielt der Supertyp der nicht–parametrisierten Typen: **Ok**. Die TL-Signatur $A <:$ **Ok** entspricht der Quest–Signatur $A ::$ **TYPE**. Die Funktion *poly* kann mit allen Typen, die Suptyp von **Ok** sind, aufgerufen werden. Durch Wahl eines Subtypen anstelle von **Ok** in der Subtypsignatur läßt sich der parametrische Polymorphismus einschränken. Im folgenden TL Beispiel sind die möglichen Typargumente beim Aufruf der Funktion *polyPerson* auf Subtypen von *Person* eingeschränkt.

```
Let NamedThing = Tuple name :String end
let car :NamedThing = tuple let name = "Ford" end
Let Person = Tuple name :String  age :Int end
Let Student <: Person = Tuple name :String  age :Int  semester :Int end
let peter :Student = tuple
                       let name = "peter"
                       let age = 27
                       let semester = 10
                     end
let polyPerson(P <:Person  p :P) :P = p
polyPerson(:Student peter)       ⇒  peter
polyPerson(:NamedThing car)      ⇒  Typfehler
```

Der Aufruf von *polyPerson* mit dem Typbezeichner *NamedThing* erzeugt einen Typfehler, da *NamedThing* nicht Subtyp von *Person* ist. Der Rückgabetyp einer polymorphen Funktion kann, wie das Beispiel zeigt, vom aktuellen Typargument abhängig sein.

3.5 Typoperatoren

Typoperatoren sind Funktionen, die Typen und/oder Werte auf Typen abbilden, und die vollständig zum Übersetzungszeitpunkt ausgewertet werden können. Das folgende Beispiel in TL zeigt den Identitätsoperator:

```
Let Op = Oper(A <: Ok) A
:Op(Int)       ⇒  Int
:Op(String)    ⇒  String
```

Die Subtypbeziehung auf Typoperatoren entspricht der Kontravarianzregel bei Funktionstypen. Typoperatoren sind nicht Subtyp von **Ok**, so daß sich der Typraum aus einer unendlichen Menge von unterschiedlichen disjunkten Sortenuniversen zusammensetzt. Die folgende Abbildung zeigt die Typhierachien von TL:

Der Typ **Ok** ist Supertyp aller nichtparametrisierten Typen. Der Typ **Nok** ist Subtyp der nichtparametrisierten Typen. Die Basistypen stehen zueinander nicht in einer Subtypbeziehung (vgl. Abb. 3.3 links). Unterschiedliche Typhierachien mit Typoperatoren erster Ordnung

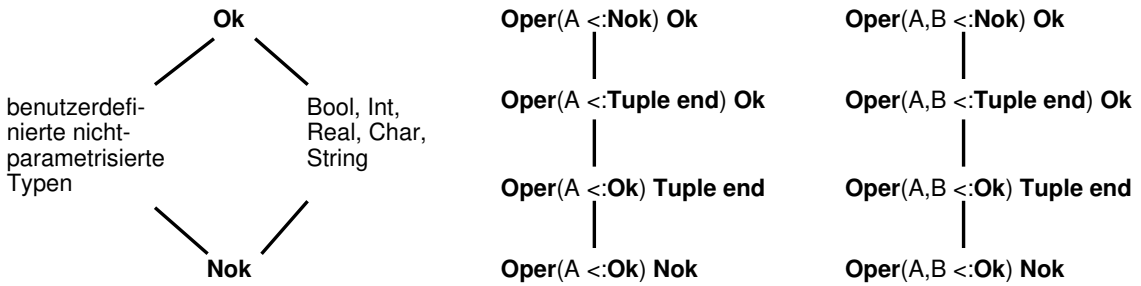


Abbildung 3.3: TL Typhierachien

lassen sich anhand der Zahl ihrer Argumente bilden. In Abbildung 3.3 sind Hierachien für Typoperatoren mit einem bzw. zwei Argumenten dargestellt, jeweils mit den höchst– und niedrigstliegenden Vertretern und zwei Beispielen benutzerdefinierter Typoperatoren.

Theoretisch denkbar und praktisch sinnvoll sind ebenso Wertbezeichner als Argumente von Typoperatoren. Das Beispiel unten benutzt den abstrakten Datentypen Nat, der im nächsten Abschnitt auf Seite 29 definiert ist.

```
Let Op2(nat :Nat) = nat.T
:Op2(nat1) ⇒ nat1.T     (* Syntaxfehler in TL *)
```

Die Einschränkung auf Wertbezeichner ist erforderlich, um in der Übersetzungsphase keine Evaluationssemantik für Werte einzuführen. Die Typoperatorapplikation mit Wertargumenten ist in TL syntaktisch nicht erlaubt. In TLMIN ist dies aus Kompatibilität zu TL ebenfalls unzulässig. Wenn Wertargumente erlaubt sind, müssen Typargumente wie bei der Funktionsapplikation mit einem Doppelpunkt markiert werden.

3.6 Abstrakte Datentypen

Abstrakte Datentypen unterstützen das Konzept der strukturierten Programmierung und leisten damit einen wesentlichen Beitrag zur Realisierung größerer Programmsysteme. Ein abstrakter Datentyp ist ein spezieller Datentyp, der

- die Implementierung des Repräsentationstyps der Datenstruktur vor dem Benutzer versteckt (*information hiding*) und
- den Zugriff auf die Datenstruktur nur über die Operationen des Datentyps erlaubt (Datenkapselung, *encapsulation*).

Eine Datenkapselung ist zweigeteilt in einen für den Benutzer sichtbaren Teil, der die Spezifikation der Datenstruktur und der Zugriffsoperationen enthält, und den für den Benutzer unsichtbaren Teil, der alle Implementierungsdetails der gekapselten Datenobjekte und die algorithmische Struktur der Zugriffsoperationen umfaßt.

Mit dem Prinzip der Datenkapselung eng verbunden ist das Konzept der Modularisierung als Zerlegungsmechanismus zur strukturierten Programmierung im Großen. Das Modul unterstützt die Implementierung abstrakter Datentypen. In Modula-2 werden die Spezifikation und die Implementierung eines abstrakten Datentyps in ein Definitionsmodul (*DEFINITION MODULE*) und ein Implementationsmodul (*IMPLEMENTATION MODULE*) getrennt. Im Definitionsmodul können opaque Typen, d.h. Typen mit verborgener Implementierung, deklariert werden. Ada verwendet zur Modularisierung das Paket (*package*), das dem Modulkonzept von Modula-2 ähnlich ist. Im öffentlichen Teil des Pakets kann der Repräsentationstyp der Datenstruktur als privat (*private*) gekennzeichnet werden. Die Implementierung ist einem Benutzer nicht zugreifbar, muß aber dennoch im öffentlichen Teil erfolgen.

Abstrakte Datentypen in TL: Die Subtypisierung von abstrakten Typvariablen beruht in TL weder auf struktureller noch auf einfacher Namenskompatibilität. Sie basiert auf Objektidentität. Für zwei abstrakte Typvariablen A und B gilt $A <: B$ genau dann, wenn entweder A und B identisch sind oder A explizit als Subtyp von B definiert ist. Die Subtyprelation gilt nicht, wenn A nicht identisch zu B ist, auch wenn beide Typen den gleichen Namen haben, und wenn A nicht explizit als Subtyp von B definiert ist. In TL können abstrakte Datentypen ohne zusätzliche Sprachkonstrukte bereits mit Subtypsignaturen in Tupel- oder Recordtypen deklariert werden. Entsprechende Tupel- oder Recordwerte enthalten die Datentypimplementierung. Es können daher mehrere Instanzen eines abstrakten Datentyps gleichzeitig existieren, anders als in Modula-2 oder Ada, wo Module keine Typen, sondern eigene Objekte mit je einer Instanz sind.

```
Let Nat = Tuple T <:Ok  zero :T  succ(x :T) :T end

let nat1 :Nat = tuple
  Let T = Int  let zero = 0  let succ(x :T) :T = x+1
end

let nat2 :Nat = tuple
  Let T = String
  let zero = ""
  let succ(x :T) :T = string.concat(x "1")
end
```

Das obige TL-Beispiel definiert den abstrakten Datentyp Nat der natürlichen Zahlen als Tupeltyp. Neben der abstrakten Typvariable T gibt es einen Anfangswert $zero$ vom Typ T und die Zugriffsoperationen $succ$, die aus einer natürlichen Zahl x den Nachfolger berechnet. Die Werte $nat1$ und $nat2$ sind verschiedene Implementierungen des Datentyps. Mit der Definition von $nat1$ und $nat2$ existieren drei abstrakte Typvariablen:

Nat.T ist die Typvariable des Tupeltyp Nat. Sie ist ein Subtyp von **Ok**. Es existieren *keine* Werte dieses Typs.

nat1.T ist die Typvariable des Tupelwertes $nat1$. Sie ist ein Subtyp von **Ok**, der typinkompatibel zu $Nat.T$, $nat2.T$ und Int ist. $nat1.zero$ hat diesen Typ, außerdem die Werte, die durch eine Hintereinanderausführung von $nat1.succ$ auf $nat1.zero$ entstehen.

nat2.T ist die Typvariable des Tupelwertes $nat2$. Sie ist ein Subtyp von **Ok**, der typinkompatibel zu $Nat.T$, $nat1.T$ und $String$ ist. $nat2.zero$ hat diesen Typ, außerdem die Werte, die durch eine Hintereinanderausführung von $nat2.succ$ auf $nat2.zero$ entstehen.

Die folgenden Beispiele verdeutlichen diese Aussagen:

```
:Op(Nat.T)                    (* typkorrekt ,da Nat.T <: Ok *)
poly(:nat1.T nat1.zero)       (* typkorrekt ,da nat1.T <: Ok und nat1.zero :nat1.T *)
polyPerson(:nat1.T nat1.zero) (* Typfehler, da nat1.T ≮: Person *)
nat1.zero + 1                 (* Typfehler, da nat1.T inkompatibel zu Int *)
nat1.succ(nat1.zero)          (* typkorrekt ,da nat1.succ ein Argument vom Typ nat1.T erwartet *)
nat1.succ(nat2.zero)          (* Typfehler, da nat1.T inkompatibel zu nat2.T *)
```

Der Typ $Nat.T$ hat keine praktische Bedeutung, da keine Werte von diesem Typ erzeugt werden können. Im Beispiel hat $nat1.zero$ den Typ $nat1.T$ und nicht T. Das Voranstellen einer Tupelvariable vor der abstrakten Typvariable heißt Typqualifizierung und wird im Typüberprüfungsalgorithmus vollzogen (siehe Abschnitt 5.2.3). Der Typ $nat1.T$ im oberen Beispiel ist inkompatibel zu dem Typ $nat1.T$, der im folgenden definiert wird:

```
let f(n :nat1.T) = ...
let nat1 :Nat = tuple
  Let T = Real  let zero = 0.0  let succ(x :T) :T = x+1.0
end

f(nat1.T)       (* Typfehler, da der von der Funktion f erwartete Typ nat1.T
                   nicht identisch mit dem Funktionsargument nat1.T ist *)
```

Ein abstrakter Datentyp kann in TL, angelehnt an Modula-2, auch mit einem Definitionsmodul (*interface*) und einem Implementationsmodul (*module*) erstellt werden, was die separate Übersetzung von Programmkomponenten und den Aufbau von Programmbibliotheken erlaubt. Im Übersetzer werden diese Module intern auf Typen und Werte abgebildet.

3.7 Rekursive Typen

Typen sind rekursiv, falls sie sich selbst direkt oder indirekt in ihrer Definition referenzieren. Mit rekursiven Typen lassen sich induktiv definierte Datenstrukturen wie Listen, Bäume oder zyklische Graphstrukturen aufbauen. In TL werden folgende sprachliche Konzepte durch rekursive Typen unterstützt:

Typvollständigkeit unter besonderer Berücksichtigung von Massendaten (*bulk data types*).

Iterationsabstraktion z.B. durch quantifizierte Prädikate, mengenwertige Ausdrücke, Funktoren oder unifikationsbasierte Datendeduktion.

Datenmodellneutralität um funktionale, relationale, imperative und objekt-orientierte Modellierungen zu ermöglichen, die die Integration einer großen Menge externer Diensterbringer erlaubt.

Reflektion des TL Übersetzers durch vordefinierte Systemkomponenten, die ihrerseits als TL Module implementiert sind (*bootstrap*) und zur Datenbanksystemprogrammierung verwendet werden können.

Nachfolgend wird dies mit Beispielen rekursiver Typdefinitionen belegt. Da TL weder vordefinierte Typkonstruktoren für Massendaten noch vordefinierte Sprachkonzepte zur Iterationsabstraktion bietet, wird die Funktionalität zur mengenorientierten Datenverarbeitung im Rahmen eines *add-on* Ansatzes [Matthes, Schmidt 91] in Tycoon Bibliotheken zur Verfügung gestellt. Die Definition von Massendatentypen erfolgt in Bibliotheksmodulen oftmals rekursiv. Ein Beispiel hierfür ist der generische Listentyp, der als Typoperator im Modul *list* realisiert ist:

```
Let Rec T(A <: Ok) <: Ok =
    Tuple
    case nil
    case cons with head : A    tail : T(A)
    end
```

Den Kern der Iterationsabstraktion bildet der Iterator $IterRep.T(E)$, der eine homogene Sammlung von Elementen des Typs E definiert:

```
Let Rec T(E <: Ok) <: Ok =
    Tuple
        empty() : Bool
        get() : E
        rest() : T(E)
    end
```

Um die gewünschte Datenmodellneutralität zu gewährleisten, verfügt TL über keinen vordefinierten Klassenbegriff von objekt-orientierten Sprachen wie z.B. Smalltalk. Eine besondere Eigenschaft dieser Sprachen ist der hohe Wiederverwendungsgrad von Programmcode durch die Erweiterung bestehender Klassen um Subklassen, die die Methoden ihrer Superklassen erben. Für die typsichere Vererbung binärer Methoden ist das Konzept der Typähnlichkeit *type matching* [Black, Hutchinson 90; Bruce 93; Bruce et al. 95; Abadi, Cardelli 95] erforderlich, das wie Subtypisierung induktiv eine strukturelle Ordnung auf Typen definiert. Typähnlichkeit ist in TLMIN mit Hilfe des rekursivem Polymorphismus (*f-bounded polymorphism*) [Canning et al. 89] realisiert, einer Erweiterung des parametrischen Polymorphismus um rekursive Signaturen. In TL existiert dieses Typkonzept bisher nicht.

Das folgende Beispiel zeigt die Methode *pointMove* einer Klasse zweidimensionaler Punkte.

```
let pointMove(Rec Self <: Ok  <: PointOp(Self)    self : Self) (x,y : Int) : Self =
    begin
        self.x := self.x + x
        self.y := self.y + y
        self
    end
```

Die Funktion *pointMove* ist rekursive polymorph. Der formale Typparameter $Self$ wird auf der rechten Seite der Signatur bereits benutzt. Der Typ **Ok** ist ein Platzhaltertyp, der für die Typüberprüfung notwendig ist. Der Typoperator *PointOp* ist rekursiv in der Punktklasse definiert und liefert als Ergebnis einen Recordtyp mit den Punktkoordinaten x und y sowie den Methoden *move* und *equal* zurück:

```
Let Rec PointOp(Self <:Ok) <:Ok =
    Record
        var x,y :Int
        move(a,b :Int) :Self
        equal(p :Self) :Bool
    end
```

Eine Subklasse der Punkteklasse möge gefärbte Punkte beschreiben. Der rekursiv definierte Typoperator $ColorpointOp$ der Subklasse gibt einen Recordtyp mit einem zusätzlichen Attribut *color* vom Typ $String$ zurück:

```
Let Rec ColorpointOp(Self <:Ok) <:Ok =
    Record
        var x,y :Int
        var color :String
        move(a,b :Int) :Self
        equal(p :Self) :Bool
    end
```

$ColorpointOp$ ist gemäß der Kontravarianzregel ein Subtyp von $PointOp$. Die Methode $colorpointEqual$ ist wie die Methode $pointEqual$ rekursiv polymorph definiert und kann nur deshalb die Implementierung von $pointEqual$ erben:

```
let colorpointEqual :Fun(Rec Self <:Ok <:ColorPointOp(Self)
                         self :Self) :Fun(p :Self) :Bool = pointEqual
```

Das letztes Beispiel für rekursiven Typen betrifft die Repräsentationen von Typen, Signaturen, Werten, Bindungen, etc. im TLMIN Übersetzer, die als wechselseitig rekursive Typen in den Schnittstellen $TLPreAST$ und $TLAST$ (siehe Anhang H) definiert sind:

```
Let Rec Type <:Ok =
    Tuple
        pos :Source.Position
        (* of type if declared; of value if inferred *)

    case wrongCase
      (* type error, suppress subsequent messages *)
    ...
    end

and Signature <:Ok =
    (* Note: identity is important to distinguish abstract
             type variables *)
    Tuple
        ide :TLIde.T
        var type :Type
        ...
    end

and Value <:Ok =
```

```
    Tuple
      pos :Source.Position

    case okCase
    ...
    end

and Binding <:Ok =
    Tuple
      ide :TLIde.T
      type :Type
      ...
    end

and ...
```

3.8 Dynamische Typen

In typisierten Programmiersprachen finden sich zwei Anwendungsgebiete, in denen die Typüberprüfung nicht vollständig zur Übersetzungszeit möglich ist und dynamisch zur Laufzeit erfolgen muß [Koehler 96]:

1. Die Manipulation von Werten, deren Typinformation außerhalb der statischen Umgebung berechnet wird.
2. Die typgesteuerte Berechnung, um generische, datenstrukturunabhängige Algorithmen zu entwickeln.

Typisches Anwendungsbeispiel für Punkt 1 ist die *eval*–Funktion, die ein Stück Programmcode als String erhält und unter Ausnutzung von Laufzeitreflektion übersetzt und ausführt. Ein weiteres Beispiel sind Operationen, die Speicherobjekte in linearisierter Form auf Dateien schreiben (*extern*) und von Dateien wieder typsicher zurück in den Speicher laden (*intern*).

Beispiele für Punkt 2 sind generische Datenvisualisierungswerkzeuge und Generatoren zum Erzeugen von Programmcode.

Als Lösung bieten sich automorphe, selbstbeschreibende Werte an, die aus einem Wert und der Laufzeitrepräsentation des Typs dieses Wertes bestehen. Die Laufzeittypprepräsentation wird in der Literatur in der Regel als dynamischer Typ bezeichnet [Matthews 87; Abadi et al. 92; Leroy, Mauny 91], obgleich es sich um einen Wert für eine statische Typrepräsentation handelt.

Die Implementierung dynamischer Typen in TL ist in [Geisler 95] und [Koehler 96] beschrieben. Ein Kernproblem bei der Implementierung ist die Wahl einer geeigneten Darstellungsform für dynamische Typen. Die Typrepräsentation während der Übersetzungszeit innerhalb des TL Compilers ist ungeignet, da Typbezeichner durch de Bruijn–Indizes (siehe Abschnitt 5.2.1) dargestellt sind, die sich auf einen gemeinsamen statischen Kontext beziehen. Die Repräsentation von dynamischen Typen muß jedoch unabhängig von einem gemeinsamen statischen Kontexten sein, damit sie in jedem möglichen Kontext gültig ist.

Es wird daher eine alternative Darstellung gewählt, in der Typbezeichner durch Objektreferenzen auf die deklarierende Typsignatur repräsentiert sind. Die Verwendung unterschiedlicher

Typdarstellungen hat den Nachteil, daß zeitaufwendige Transformationen zwischen den beiden Repräsentationen notwendig sind. Wie bereits in der Einleitung motiviert, vereinfacht die Darstellung von Typen basierent auf Objektrefenzen im TLMIN Typprüfer die Implementierung von dynamischen Typen, da die aufwendigen Transformationen entfallen. Typrepräsentationen zur Übersetzungszeit und zur Laufzeit können in TLMIN identisch sein.

4. Typüberprüfung in Lambda–Kalkülen

Der Lambda–Kalkül ist ein formales System, das ursprünglich zur Untersuchung mathematischer Funktionen in der Mathematik diente. Neuerdings wird es zum Studium von Algorithmen und ihrer Berechenbarkeit herangezogen. In den letzten Jahren wurden zahlreiche Lambda–Kalküle untersucht, mit denen wichtige programmiersprachliche Konzepte einfach, umfassend und ohne Einschränkungen darstellbar sind und die sich daher von kommerziellen Programmiersprachen stark unterscheiden [Reynolds 74; Girard 72; Cardelli, Wegner 85; Ghelli 90; Curien, Ghelli 91; Cardelli et al. 91; Steffen, Pierce 94]. Der Lambda–Kalkül bildet die Grundlage für funktionale Programmiersprachen, wie Lisp, Miranda, Haskell und TL. Eine besondere Bedeutung für die Verifikation der statischen Semantik von Programmiersprachen besitzen typisierte Lambda–Kalküle. In den folgenden Abschnitten werden der untypisierte Lambda–Kalkül, der einfach typisierte Lambda–Kalkül, der Lambda–Kalkül zweiter Ordnung, der Lambda–Kalkül endlicher Ordnung und der Lambda–Kalkül unendlicher Ordnung mit und ohne Subtypisierung vorgestellt. Dabei wird jeweils zur Spezifikation der Syntax eine Grammatik und zur Spezifikation der statischen Semantik ein Typsystem aufgestellt. Mit dem jeweiligen Typsystem wird ein Typüberprüfungsalgorithmus präsentiert.

4.1 Ungetypter Lambda–Kalkül

Der gewöhnliche Lambda–Kalkül wurde von A. Church in den dreißiger Jahren aufgestellt. In diesem Kalkül gibt es nur Funktionen, die Zahlen (*church numbers*), Boolesche Werte und Datenstrukturen wie Tupel oder Varianten repräsentieren können [Cardelli et al. 91]. Es existiert nur der universelle Funktionstyp, der von Werten auf Werte abbildet, die ihrerseits Funktionen dieses Typs sind. Beispiel:

```
fun(x) x                          (* Identitätsfunktion *)
fun(y) fun(x) y (y (x)).          (* Doppelte Hintereinanderausführung *)
```

In der ersten Beispielzeile ist die Identitätsfunktion dargestellt. Die Funktion in Zeile zwei erwartet als Argument eine Funktion y und gibt als Ergebnis eine Funktion zurück, die die Hintereinanderausführung $y(y(x))$ berechnet.

Syntax

Die Menge syntaktisch gültiger Lambda-Ausdrücke ist durch folgende Grammatik definiert:

Value ::=
Value-Identifier | *Value-Constant*
| ***fun*** *"(" Value-Identifier ")" Value* *Abstraktion*
| *Value "(" Value ")"* *Applikation*
| *"{" Value "}"*

Ein Wert ist im einfachsten Fall eine Variable, z.B. x, oder eine Konstante, z.B. 1. Funktionsabstraktionen können gebildet und durch Funktionsapplikation eliminiert werden. Die Variable x in dem Ausdruck $\mathbf{fun}(x)\,a$ bezeichnet man als durch die Funktionsabstraktion gebunden. Der Gültigkeitsbereich der Bindung ist der Ausdruck a. Ein Vorkommen einer Variablen außerhalb des Gültigkeitsbereichs jeder ihrer Bindungen durch eine Funktionsabstraktion ist ein freies Vorkommen. Ein Vorkommen, das nicht frei ist, ist ein gebundenes Vorkommen. Zum Beispiel ist in dem Ausdruck $\mathbf{fun}(x)\,\{+\,(y)\,(x)\}$ die Variable x gebunden und die Variable y frei.

Substitution

Zur Transformation von Lambda-Ausdrücken existieren im Lambda–Kalkül Reduktionsregeln. Bei der Funktionsapplikation wird im Rumpf der Funktion der Formalparameter durch den Aktualparameter substituiert (*Beta-Reduktion*):

$$\{\mathbf{fun}(x)\,a\}\,(b) \;=\; \langle b \rightarrow x\rangle a$$

Eine Funktionsapplikation mit dem Parameter b als aktuelles Argument ist äquivalent zum Rumpf der Funktion, indem jedes freie Vorkommen des formalen Parameters x durch b ersetzt ist. Die Substitutionsoperation $\langle b \rightarrow x\rangle a$ ist induktiv über der Struktur von a definiert:

1)	$\langle b \rightarrow x\rangle x$	=	b	
2)	$\langle b \rightarrow x\rangle y$	=	y	*falls* $x \neq y$
3)	$\langle b \rightarrow x\rangle \mathbf{fun}(x)\,a$	=	$\mathbf{fun}(x)\,a$	
4)	$\langle b \rightarrow x\rangle \mathbf{fun}(y)\,a$	=	$\mathbf{fun}(y)\,\langle b \rightarrow x\rangle a$	*falls* $x \neq y$ *und* y *ist nicht frei in* b
5)	$\langle b \rightarrow x\rangle f\,(a)$	=	$\langle b \rightarrow x\rangle f(\langle b \rightarrow x\rangle a)$	
6)	$\langle b \rightarrow x\rangle (a)$	=	$\{\langle b \rightarrow x\rangle a\}$	

In Punkt 4) wird die Annahme *"y ist nicht frei in b"* vorausgesetzt, um Namenskonflikte (*name clash problem*) zu umgehen und das ungewollte Binden von Bezeichnern (*variable capture*) zu verhindern (siehe Abschnitt 5.2).

Typsystem

Die im folgenden nach Konklusionen geordnet aufgelisteten Typregeln bilden das Typsystem des untypisierten Lambda–Kalküls. Es besteht aus einem Axiom und vier Deduktionsregeln:

Kontext:

$$\frac{}{\vdash \oslash \ sig}\ [\textit{Sig empty}] \qquad \frac{\vdash S \ sig \quad x \notin Dom(S)}{\vdash S, x \ sig}\ [\textit{Sig ide}]$$

Value:

$$\frac{\vdash S, x, S' \ sig}{S, x, S' \vdash x \ value}\ [\textit{Value ide}] \qquad \frac{S, x \vdash b \ value}{S \vdash \mathbf{fun}\,(x)\,b \ value}\ [\textit{Value} \rightarrow \textit{introduction}] \qquad \frac{S \vdash f \ value \quad S \vdash a \ value}{S \vdash f\,(a) \ value}\ [\textit{Value} \rightarrow \textit{elimination}]$$

Die Typregel *[Sig empty]* definiert einen leeren Kontext als wohlgeformt. Die Regel *[Sig ide]* erlaubt es, einen Wertbezeichner x in einen wohlgeformten Kontext einzutragen. Dabei können niemals zwei Variablen mit demselben Namen in den Kontext eingefügt werden. Dies wird in der Regel durch die Beziehung $x \notin Dom(S)$ ausgedrückt. Diese Einschränkung stellt sicher, daß die Bedingung *"y ist nicht frei in b"* in Punkt 4) der Substitutionsoperation gültig ist. $Dom(S)$ bezeichnet die Menge der in S enthaltenen Bezeichner. Die Regel *[Value → introduction]* definiert eine Funktionsabstraktion im Kontext S als wohlgeformt, falls im Kontext S erweitert mit x der Wert b gültig ist. Eine Funktionsapplikation ist korrekt, falls im selben Kontext sowohl die Funktion als auch das Funktionsargument korrekt sind (*[Value → elimination]*). Nach Regel *[Value ide]* ist ein Bezeichner gültig, wenn er im statischen Kontext enthalten ist.

Die allgemeine Syntax von Kontexten im Kalkül kann durch eine Grammatik spezifiziert werden:

Context	::=	$\oslash$	*leerer Kontext*
	|	*Context, Value-Identifier*	*Wertsignatur*

Ein statischer Kontext ist entweder leer ($\oslash$) oder er enthält eine Liste von Wertbezeichnern, die durch Kommata getrennt sind.

Typüberprüfungsalgorithmus

Der Typüberprüfungsalgorithmus für den ungetypten Lambda–Kalkül kann aus den Typregeln abgeleitet werden, da die Typregeln in eindeutiger Weise rückwärts anwendbar sind. Der in Abbildung 4.1 in einer Pseudocodenotation dargestellte Algorithmus zeigt einen automatischen Beweiser, der nur Wahrheitswerte ausgibt. Im Fehlerfall werden keine aussagekräftigen Fehlermeldungen ausgegeben.

Die Funktionen *issig* und *isvalue* beweisen die Korrektheit der Aussagen $\vdash S \ sig$ bzw. $S \vdash v \ value$. Die Funktion *lookupValueIde* sucht im statischen Kontext den Bezeichner x (*scoping*).

```
let issig(S :_Env) :Bool =
    if S = ∅ then true                                  (* Regel: [Sig empty] *)
    elsif S = S1,x then issig(S1)                       (* Regel: [Sig ide] *)
    else false
    end

let lookupValueIde(S :_Env  x :_Ide) :Bool =
    if S = ∅ then false                                 (* Regel: Sig empty *)
    elseif S = S1,x then true                           (* Regel: Sig ide *)
    elseif S = S1,y then lookupValueIde(S1 x)
    else false
    end

let isvalue(S :_Env  a :_Value) :Bool =
    if a = x and lookupValueIde(S x) then issig(S)      (* Regel: [Value ide] *)
    elsif a = fun(x) b then isvalue(S,x b)              (* Regel: [Value → introduction alg] *)
    elsif a = f(b) and isvalue(S f) then isvalue(S b)   (* Regel: [Value → elimination] *)
    else false
    end
```

Abbildung 4.1: Typüberprüfungsalgorithmus des ungetypten Lambda-Kalküls.

Die durch einen Unterstrich gekennzeichneten Typen $_Env$, $_Ide$ und $_Value$ identifizieren die Syntaxbaumrepräsentationen von statischen Kontexten, von Wertbezeichnern und Werten des Kalküls. Die Syntaxbäume sind in precompilierter Form aufgeführt. Der Operator '=' im Programmcode vergleicht zwei Syntaxbäume auf strukturelle Gleichheit und liefert einen Wahrheitswert zurück.

Für jede Typregel einer Aussageart existiert in der entsprechenden Funktion des Typprüfers eine Fallunterscheidung. Im Programmcode steht die entsprechende Typregel als Kommentar neben der Bedingung. Die Überprüfung einer formalen Aussage führt zu drei Resultaten.

- Es wird die Wohlgeformtheit eines leeren Kontextes bestimmt. Das Ergebnis ist *true*.
- Es trifft keine der Fallunterscheidungen in den jeweiligen Funktionen zu. Das Ergebnis ist *false*.
- Es werden weitere Funktionen aufgerufen, um die Korrektheit von Teilaussagen zu ermitteln. Das Ergebnis ist die logische Konjunktion der Einzelresultate.

4.2 Einfach getypter Lambda–Kalkül

Im einfach typisierten Lambda–Kalkül wird das definierende Auftreten einer Wertvariable mit einer expliziten Typangabe annotiert. Jeder Lambda-Ausdruck hat einen Typ. Beginnend mit

einer Menge von Basistypen wie *Boolean*, *Integer* oder *Real* können mit dem Konstruktor '$\rightarrow$' Funktionstypen konstruiert werden. Beispiele für Lambda Ausdrücke sind:

$\textbf{fun}\,(x : Integer)\,x$ *(* Integer Identitätsfunktion *)*
$\textbf{fun}\,(y : \textbf{Fun}\,(x : Boolean) : Boolean)\,\textbf{fun}\,(x : Boolean)\,y\,(y\,(x))$ *(* Boolean Verdopplung *)*

In der ersten Beispielzeile ist die Identitätsfunktion dargestellt, die *Integer*–Werte auf *Integer*–Werte abbildet. Die Funktion in Zeile zwei erwartet als Argument eine Boolesche Funktion y und gibt als Ergebnis eine Funktion zurück, die die Hintereinanderausführung $y(y(x))$ berechnet.

Syntax

Die Wertsyntax des einfach getypten Lambda–Kalküls unterscheidet sich von der Syntax des ungetypten Lambda–Kalküls durch eine Typannotation in der Funktionsabstraktion:

Value ::=
Value-Identifier | *Value-Constant*
| **fun** *"(" Value-Identifier ":" Type ")" Value* *Abstraktion mit Typangabe*
| *Value "(" Value ")"* *Applikation*
| *"{" Value "}"*

Die Typsyntax läßt sich ebenfalls mit einer Grammatik beschreiben:

Type ::=
Type-Constant
| **Fun** *"(" Value-Identifier ":" Type ")" ":" Type* *Funktionstyp*
| *"{" Type "}"*

Typen können Konstanten, Funktionstypen oder geklammerte Typausdrücke sein.

Typsystem

Die folgenden Typregeln bilden das Typsystem des einfach getypten Lambda–Kalküls:

Kontext:

[Sig empty]
$$\frac{}{\vdash \emptyset\ sig}$$

[Sig ide]
$$\frac{S \vdash A :: \mathbf{TYPE} \quad x \notin Dom(S)}{\vdash S, x : A\ sig}$$

Typen:

$$\frac{\vdash S\ sig \quad A \in \{Bool, Int, String\}}{S \vdash A :: \mathbf{TYPE}}\ [Type\ builtin] \qquad \frac{S, x : A \vdash B :: \mathbf{TYPE}}{S \vdash \mathbf{Fun}\,(x : A) : B :: \mathbf{TYPE}}\ [Type \rightarrow]$$

Werte:

$$\frac{\vdash S, x : A, S'\ sig}{S, x : A, S' \vdash x : A}\ [Value\ ide] \qquad \frac{S, x : A \vdash b : B}{S \vdash \mathbf{fun}\,(x : A)\,b : \mathbf{Fun}\,(x : A) : B}\ [Value \rightarrow introduction]$$

$$\frac{S \vdash f : \mathbf{Fun}\,(x : A) : B \quad S \vdash a : A}{S \vdash f\,(a) : B}\ [Value \rightarrow elimination]$$

Die Typregel *[Sig empty]* definiert den leeren Kontext als wohlgeformt. Der Kontext bleibt wohlgeformt, wenn er um eine Wertsignatur $x : A$ mit einem wohlgeformten Typ A erweitert wird, vorausgesetzt, daß x noch nicht im Kontext bekannt ist (*[Sig ide]*). Die Aussage $A :: \mathbf{TYPE}$ drückt die Wohlgeformtheit von A aus. Eine Variable x ist gültig und hat den Typ A, falls die Signatur $x : A$ im Kontext bekannt ist (*[Value ide]*). **TYPE** ist die Sorte der nichtparametrisierten Typen. Funktionswerte haben einen Funktionstyp mit gleicher Funktionssignatur (*[Value → introduction]*). Regel *[Value → elimination]* bestimmt als Rückgabetyp von $f(a)$ den Typ B, falls f eine Funktion ist und der Typ von a mit dem formalen Argumenttyp von f übereinstimmt. Gegenüber dem ungetypten Lambda–Kalkül neu sind die Typregeln *[Type builtin]* und *[Type →]*, die die Wohlgeformtheit von vordefinierten Typkonstanten bzw. von Funktionstypen festlegen.

Die allgemeine Syntax von Kontexten im Kalkül kann durch eine Grammatik spezifiziert werden:

Context	::=	∅	*leerer Kontext*
	|	*Context, Value-Identifier ":" Type*	*Wertsignatur*

Der Ausdruck $S, x : T$ erweitert die bestehende Signatur S mit einem Wertbezeichner x vom Typ T. Der leere Kontext ∅ enthält keine Signatur.

Typüberprüfungsalgorithmus

Abbildung 4.2 zeigt den Typüberprüfungsalgorithmus, der sich unmittelbar aus dem Typsystem erstellen läßt. Die Funktionen *issig* und *lookupValueIde* sind bereits vom Algorithmus des ungetypten Lambda–Kalküls auf Seite 38 bekannt. Die Funktion *hastype* bestimmt die Wohlgeformtheit von Werten. Die Funktion *haskind* überprüft die Wohlgeformtheit von Funktionstypen gemäß der Typregeln *[Type builtin]* und *[Type →]*.

Der obige Algorithmus arbeitet nicht effizient. Die Überprüfung eines Wertes oder eines Typs auf Wohlgeformtheit endet mit der Überprüfung des aktuellen Kontextes, was die Überprüfung der Typen im Kontext impliziert u.s.w. Die Kontextüberprüfung kann jedoch sofort mit dem Ergebnis *true* terminieren, wenn die Typüberprüfung mit einem leeren Kontext

```
let issig(S :_Env) :Bool =
   if S = ∅ then true                                   (* Regel: [Sig empty] *)
   elsif S = S1,x:A then haskind(S1 A TYPE)             (* Regel: [Sig ide] *)
   else false
   end

let lookupValueIde(S :_Env  x :_Ide  A :_Type) :Bool =
   if S = ∅ then false                                  (* Regel: Sig empty *)
   elseif S = S1,x:A then true                          (* Regel: Sig ide *)
   elseif S = S1,y:B then lookupValueIde(S1 x A)
   else false
   end

let hastype(S :_Env  a :_Value  A :_Type) :Bool =
   if a = x and lookupValueIde(S x A) then issig(S)     (* Regel: [Value ide] *)
   elsif a = fun(x:B) c and                             (* Regel: [Value → introduction alg] *)
      haskind(S B TYPE) and
      hastype(S,x:B c C) then A = Fun(x:B):C
   elsif a = f(b) and hastype(S f Fun(x:B):C) and       (* Regel: [Value → elimination] *)
      hastype(S b B) then A = C
   else false
   end

let haskind(S :_Env  A :_Type  K :_Kind) :Bool =
   if A ∈ {Bool,Int,String} and issig(S) then K = TYPE  (* Regel: [Type builtin] *)
   elsif A = Fun(x:B):C and haskind(S B TYPE)           (* Regel: [Type → alg] *)
      and haskind(S,x:B C TYPE) then K = TYPE
   else false
   end
```

Abbildung 4.2: Typüberprüfungsalgorithmus des einfach getypten Lambda-Kalküls.

beginnt und in den Kontext nur überprüfte Typen einfügt. An zwei Positionen wird im Algorithmus der statische Kontext mit nicht geprüften, eventuell fehlerhaften Typen erweitert. Dies sind die Implementierungen der Typregeln *[Type →]* und *[Value → introduction]*. Durch das Hinzufügen einer zusätzlichen Prämisse, die die Korrektheit der Typen überprüft, die in der zweiten Prämisse in den Kontext eingefügt werden, ist garantiert, daß nur wohlgeformte Kontexte aufgebaut werden. Im folgenden sind die erweiterten Typregeln *[Type → alg]* und *[Value → introduction alg]* dargestellt:

[Type → alg]

$$\frac{S \vdash A :: \mathbf{TYPE} \quad S, x : A \vdash B :: \mathbf{TYPE}}{S \vdash \mathbf{Fun}\,(x : A) : B :: \mathbf{TYPE}}$$

[Value → introduction alg]

$$\frac{S \vdash A :: \mathbf{TYPE} \quad S, x : A \vdash b : B}{S \vdash \mathbf{fun}\,(x : A)\, b : \mathbf{Fun}\,(x : A) : B}$$

Durch die Modifikation kann die Funktion *issig* im Algorithmus entfallen. Der Algorithmus steht in Abbildung 4.3.

Ein im Lambda–Kalkül geschriebenes Programm p vom Typ A ist korrekt, wenn der Typprüfer die Aussage $\emptyset \vdash p : A$ mit *true* beantwortet. Wenn der Typ A nicht bekannt ist, kann er aus p inferiert werden, wenn der Algorithmus, um Unifikationstypen und eine Resolutionsfunktion erweitert wird.

4.3 Getypter Lambda–Kalkül zweiter Ordnung

Girard [Girard 72] und Reynolds [Reynolds 74] führen unabhängig von einander den getypten Lambda–Kalkül zweiter Ordnung ein, der die Formulierung parametrisch polymorpher Funktionen mittels Typabstraktion gestattet. Durch Einführung von Typvariablen läßt sich beispielsweise die polymorphe Identitätsfunktion definieren und auf beliebige Typargumente anwenden:

$\mathbf{fun}\,(X)\,\mathbf{fun}\,(x : X)\,x$	$:\mathbf{Fun}\,(X) : \mathbf{Fun}\,(x : X) : X$
$\{\mathbf{fun}\,(X)\,\mathbf{fun}\,(x : X)\,x\}\,(Integer)$	$:\mathbf{Fun}\,(x : Integer) : Integer$
$\{\mathbf{fun}\,(X)\,\mathbf{fun}\,(x : X)\,x\}\,(Boolean)$	$:\mathbf{Fun}\,(x : Boolean) : Boolean$

Das Beispiel zeigt in der ersten Zeile die polymorphe Identitätsfunktion mit ihrem Typ. In Zeile zwei und drei wird die Funktion mit dem Argument *Integer* bzw. *Boolean* aufgerufen. Der Ergebnistyp der Funktionsanwendung ist vom Typargument abhängig.

Syntax

Die Syntax von Wertausdrücken im getypten Lambda–Kalkül zweiter Ordnung ist die Erweiterung des einfach getypten Lambda–Kalküls um Typabstraktion und Typapplikation.

Value ::= ...	
| **fun** *"(" Type-Identifier ")" Value*	*Typabstraktion*
| *Value "(" Type ")"*	*Typapplikation*

```
let lookupValueIde(S :_Env  x :_Ide  A :_Type) :Bool =
   if S = ∅ then false                                   (* Regel: Sig empty *)
   elseif S = S1,x:A then true                           (* Regel: Sig ide *)
   elseif S = S1,y:B then lookupValueIde(S1 x A)
   else false
   end

let hastype(S :_Env  a :_Value  A :_Type) :Bool =
   if a = x and lookupValueIde(S a A) then issig(S)      (* Regel: [Value ide] *)
   elsif a = fun(x:B) b and                              (* Regel: [Value → introduction] *)
      A = Fun(x:B):C then hastype(S,x:B b C)
   elsif a = f(b) and hastype(S f Fun(x:B):C)            (* Regel: [Value → elimination] *)
      and A = C then hastype(S b B)
   else false
end

let haskind(S :_Env  A :_Type  K :_Kind) :Bool =
   if A ∈ {Bool,Int,String} and                          (* Regel: [Type builtin] *)
      issig(S) then K = TYPE
   elsif A = Fun(x:B):C and                              (* Regel: [Type →] *)
      haskind(S,x:B C TYPE) then K = TYPE
   else false
   end
```

Abbildung 4.3: Optimierter Typüberprüfungsalgorithmus des einfach getypten Lambda-Kalküls.

Die Syntax von Typen wird um Typvariablen und polymorphe Funktionstypen erweitert:

Type ::=
  Type-Identifier
| ***Fun*** *"(" Type-Identifier ")" ":" Type* *polymorpher Funktionstyp*

Typsystem

Das Typsystem des getypten Lambda–Kalküls zweiter Ordnung ist eine Erweiterung des einfach getypten Kalküls um zusätzliche Typregeln für Typvariablen, polymorphe Funktionen und deren Typen. Die zusätzlichen Regeln sind:

Kontext:

[Sig Ide]

$$\frac{\vdash S\ sig \qquad X \notin Dom(S)}{\vdash S, X :: \mathbf{TYPE}\ sig}$$

Typen:

[Type Ide]

$$\frac{\vdash S, X :: \mathbf{TYPE}, S'\ sig}{S, X :: \mathbf{TYPE}, S' \vdash X :: \mathbf{TYPE}}$$

[Type ∀]

$$\frac{S, X :: \mathbf{TYPE} \vdash B :: \mathbf{TYPE}}{S \vdash \mathbf{Fun}(X) : B :: \mathbf{TYPE}}$$

Werte:

[Value ∀ introduction]

$$\frac{S, X :: \mathbf{TYPE} \vdash b : B}{S \vdash \mathbf{fun}(X)\, b : \mathbf{Fun}(X) : B}$$

[Value ∀ elimination]

$$\frac{S \vdash f : \mathbf{Fun}(X) : A \qquad S \vdash B :: \mathbf{TYPE}}{S \vdash f(B) : \langle B \rightarrow X \rangle A}$$

Die Regel *[Sig Ide]* erweitert den bestehenden gültigen Kontext um polymorphe Typvariablen. Der Name der Typvariable darf nicht im aktuellen Kontext enthalten sein. Gemäß Regel *[Type Ide]* sind polymorphe Typvariablen nur gültig, wenn sie im Kontext enthalten sind. Regel *[Type ∀]* legt **TYPE** als Sorte für den polymorphen Funktionstyp fest. Regel *[Value ∀ introduction]* definiert eine polymorphe Funktion mit ihrem Typ. Zur Bestimmung des Rückgabetyps einer polymorphen Funktion wird in der Regel *[Value ∀ elimination]* im Rückgabetyp der Funktion die formale Typvariable durch den aktuellen Typparameter substituiert.

Zur Syntax von Kontexten kommt die Typsignatur hinzu:

Context ::= ...
| *Context, Type-Identifier "::" Kind* *Typsignatur*

Als einzige Sorte (*kind*) ist nur **TYPE**, die Menge der Typen, für die Werte existieren, zulässig. Im Lambda–Kalkül zweiter Ordnung ist das die gesamte Typmenge. Das Konzept der Sorten gehört noch nicht explizit zur Sprache und tritt nur im Kontext der Typregeln in Erscheinung.

```
let lookupValueIde(S :_Env  x :_Ide  A :_Type) :Bool =
   if S = ∅ then false                                   (* Regel: Sig empty *)
   elsif S = S1,x:A then true                            (* Regel: Sig ide *)
   elsif S = S1,y:B then lookupValueIde(S1 x A)
   elsif S = S1,x :: K then false
   elsif S = S1,Y :: K then lookupValueIde(S1 x A)
   end

let lookupTypeIde(S :_Env  X :_Ide  K :_Kind) :Bool =
   if S = ∅ then false                                   (* Regel: Sig empty*)
   elsif S = S1,X :: K then true                         (* Regel: Sig Ide *)
   elsif S = S1,Y :: L then lookupTypeIde(S1 X K)
   elsif S = S1,X:B then false
   elsif S = S1,y:B then lookupValueIde(S1 X K)
   end

let hastype(S :_Env  a :_Value  A :_Type) :Bool =
   ...
   elsif a = fun (X) b and                               (* Regel: Value ∀ introduction *)
      hastype(S,X :: TYPE b B) then A = Fun (X): B
   elsif a = f(B) and hastype(S f Fun (X): C) and        (* Regel: Value ∀ elimination *)
      haskind(S B TYPE) then A = ⟨B → X⟩C
   ...

let haskind(S :_Env  A :_Type  K :_Kind) :Bool =
   ...
   elsif A = X then lookupTypeIde(S X K)                 (* Regel: Type Ide *)
   elsif A = Fun (X): B and                              (* Regel: Type ∀ *)
      haskind(S,X :: TYPE B lkindbasic) then K = TYPE
   ...
```

Abbildung 4.4: Typüberprüfungsalgorithmus des getypten Lambda-Kalküls zweiter Ordnung.

Substitution

In der Typregel *[Value ∀ elimination]* wird zur Bestimmung des aktuellen Rückgabetyps beim Aufruf einer polymorphen Funktion f mit dem Parameter B im formalen Rückgabetyp A jedes freie Vorkommen des Typbezeichners X durch B ersetzt. Die Substitution $\langle B \rightarrow X \rangle A$ ist wie die Termsubstitution in Abschnitt 4.1 induktiv auf der Struktur von A definiert:

1)	$\langle B \rightarrow X \rangle X$	$=$	B	
2)	$\langle B \rightarrow X \rangle Y$	$=$	Y	*falls* $X \neq Y$
3)	$\langle B \rightarrow X \rangle \mathbf{Fun}(x : C) : D$	$=$	$\mathbf{Fun}(x : \langle B \rightarrow X \rangle C) : \langle B \rightarrow X \rangle D$	
4)	$\langle B \rightarrow X \rangle \mathbf{Fun}(X) : C$	$=$	$\mathbf{Fun}(X) : C$	
5)	$\langle B \rightarrow X \rangle \mathbf{Fun}(Y) : C$	$=$	$\mathbf{Fun}(Y) : \langle B \rightarrow X \rangle C$	*falls* $X \neq Y$ *und* Y *ist nicht frei in* B

In Zeile 5) wird die Bedingung *"Y ist nicht frei in B"* vorausgesetzt, um das ungewollte Binden von freien Variablen (*name clash problem*) zu umgehen (siehe Abschnitt 5.2).

Typüberprüfungsalgorithmus

Der Typüberprüfungsalgorithmus für den getypten Lambda–Kalkül zweiter Ordnung kann aus den Typregeln konstruiert werden. Der Algorithmus ist eine Erweiterung des Typprüfers für den einfach getypten Lambda–Kalkül. Im Programmfragment in Abbildung 4.4 ist aus Platzgründen nur diese Erweiterung dargestellt. Die im Algorithmus neue Funktion *lookupTypeIde* sucht im statischen Kontext eine Typvariable X mit der Sorte K. Wird der Bezeichner mit der Sorte gefunden, ist das Ergebnis *true*. Ist der angetroffene Bezeichner eine Wertvariable, ist das Ergebnis *false*. Die gegenüber dem einfach getypten Lambda–Kalkül modifizierte Funktion *lookupValueIde* unterscheidet ebenfalls zwischen Wert– und Typsignaturen. Ist der angetroffene Bezeichner ein Typbezeichner ist das Ergebnis *false*. Die Funktionen *hastype* und *haskind* enthalten jeweils zwei weitere Fallunterscheidungen. Jede Fallunterscheidung implementiert eine Typregel.

4.4 Getypte Lambda–Kalküle endlicher und unendlicher Ordnung

Die Lambda–Kalküle ab der dritten Ordnung enthalten als neue Sprachkonzepte Sorten und Typoperatoren. Das folgende Beispiel enthält Definitionen und Anwendungen von Typoperatoren:

$$\{\mathbf{Oper}(X :: \mathbf{TYPE})\, X\}\,(Integer) \;=\; Integer$$
$$\{\mathbf{Oper}(X :: \mathbf{TYPE})\, X\}\,(Boolean) \;=\; Boolean$$

$$f : \mathbf{Fun}(Y :: \mathbf{TYPE}) : Z$$
$$f\,(X) : \{\mathbf{Oper}(Y :: \mathbf{TYPE})\, Z\}\,(X)$$

Zur Klassifizierung von Typen und Typoperatoren treten Sorten in polymorphen Funktionswerten und Funktionstypen sowie in Typoperatoren auf. Die Auflistung unten beschreibt Sortenbeispiele:

TYPE ist die Sorte der gewöhnlichen Typen.

OPER (X :: TYPE): TYPE ist die Sorte der Typoperatoren, die von gewöhnlichen Typen auf gewöhnliche Typen abbilden.

OPER (X :: OPER (X :: TYPE): TYPE): TYPE ist die Sorte der Typoperatoren, die von einfachen Typoperatoren auf gewöhnliche Typen abbilden.

OPER (X :: TYPE): OPER (X :: TYPE): TYPE ist die Sorte der Typoperatoren, die von gewöhnlichen Typen auf einfache Typoperatoren abbilden.

Zur Vereinfachung der Schreibweise wird im folgenden der getypte Lambda–Kalkül n-ter Ordnung mit F_n abgekürzt. Die Notation für die Vereinigung der Sprachen F_i mit einer natürlichen Zahl i ist F_∞.

Syntax

Die folgende Grammatik definiert die Sortensyntax von F_3 und F_∞.

Kind ::=	*(in F_3)*
TYPE	*Basissorte*
| ***OPER*** *"(" Type-Identifier "::" **TYPE** ")" ":" **TYPE***	*Typoperatorsorte*
| *"{" Kind "}"*	

Kind ::=	*(in F_∞)*
TYPE	*Basissorte*
| ***OPER*** *"(" Type-Identifier "::" Kind ")" ":" Kind*	*Typoperatorsorte*
| *"{" Kind "}"*	

Die Basissorte ist **TYPE**. Daneben gibt es in F_3 die Sorte **OPER** (*Type-Identifier*:: **TYPE**) : **TYPE** der Typoperatoren, die einfache Typen auf einfache Typen abbilden. In F_∞ gibt es unendlich viele Sorten. Aus einfachen Sorten werden durch Anwendung des Sortenoperators **OPER** komplexere Sorten gebildet. Die Syntax von Typen und Werten in F_3 und F_∞ sind identisch:

Type ::=	
Type-Constant | *Type-Identifier*	
| ***Fun*** *"(" Value-Identifier ":" Type ")" ":" Type*	*Funktionstyp*
| *"{" Type "}"*	
| ***Fun*** *"(" Type-Identifier "::" Kind ")" ":" Type*	*polymorpher Funktionstyp*
| ***Oper*** *"(" Type-Identifier "::" Kind ")" Type*	*Typoperator*
| *Type "(" Type ")"*	*Applikation*

Im Vergleich mit F_2 enthalten Typvariablen von polymorphen Funktionen in F_3 und F_∞ eine Sortenangabe.

Value ::=
Value-Identifier | *Value-Constant*
| ***fun*** *"(" Value-Identifier ":" Type ")" Value* *Abstraktion*
| *Value "(" Value ")"* *Applikation*
| *"{" Value "}"*
| ***fun*** *"(" Type-Identifier "::" Kind ")" Value* *Typabstraktion mit Sortenangabe*
| *Value "(" Type ")"* *Typapplikation*

Wird **TYPE** als Sorte in polymorphen Funktionen von F_2 gewählt, so ist F_2 eine Teilmenge von F_3, bzw. F_∞. Die Kalküle F_n und F_{n+1} mit $n \geq 3$ unterscheiden sich nur in der Ordnung der Sorten. Sie ist induktiv auf der Sortenstruktur definiert.

- Die Sorte der Basistypen **TYPE** besitzt die Ordnung 1.
- Die Sorte **OPER** $(X :: K) : L$ hat die Ordnung $1 + max(k,l)$, wobei k die Ordnug von K und l die Ordnung von L ist.

In F_n können Sorten mit der maximalen Ordnung $n-1$ auftreten. Die folgende Auflistung verdeutlicht dies nochmals:

F_1: Typen sind keine Objekte erster Klasse. Sie dienen nur zur Beschreibung der Funktionalität von Werten.

F_2: Es können polymorphe Funktionen definiert werden, die Typen als Argumente nehmen und auf Werte abbilden.

F_3: Die Sprache enthält Typoperatoren, die einfache Typen auf einfache Typen abbilden.

F_4: Es existieren Typoperatoren, die Typoperatoren aus F_3 als Argumente nehmen und auf ebensolche abbilden.

F_n: Typoperatoren nehmen als Parameter Typen mit einer Sorte der Ordnung $n-2$. Von derselben Ordnung ist die Sorte des Rückgabetyps.

Typsystem

Das im folgenden aufgeführte Typsystem von F_∞ zeigt nur die gegenüber F_2 hinzugefügten oder modifizierten Typregeln. In den modifizierten Typregeln wird von der speziellen Sorte **TYPE** abstrahiert. Neu sind Typregeln, die die Wohlgeformtheit von Sorten beschreiben und Typregeln, die die Beziehung zwischen Typen und Sorten ausdrücken.

Kontext:

[Sig Ide]

$$\frac{S \vdash K \ kind \qquad X \notin Dom(S)}{\vdash S, X :: K \ sig}$$

Sorten:

$$\frac{\vdash S\ sig}{S \vdash \mathbf{TYPE}\ kind}\ [Kind\ basic] \qquad \frac{S, X :: K \vdash L\ kind}{S \vdash \mathbf{OPER}\,(X :: K) : L\ kind}\ [Kind\ oper]$$

Typen:

$$\frac{\vdash S, X :: K, S'\ sig}{S, X :: K, S' \vdash X :: K}\ [Type\ Ide] \qquad \frac{S, X :: K \vdash A :: \mathbf{TYPE}}{S \vdash \mathbf{Fun}\,(X :: K) : A :: \mathbf{TYPE}}\ [Type\ \forall]$$

$$\frac{S, X :: K \vdash A :: L}{S \vdash \mathbf{Oper}\,(X :: K)\,A :: \mathbf{OPER}\,(X :: K) : L}\ [Type\ introduction] \qquad \frac{S \vdash A :: \mathbf{OPER}\,(X :: K) : L \quad S \vdash B :: K}{S \vdash A\,(B) :: L}\ [Type\ elimination]$$

Werte:

$$\frac{S, X :: K \vdash b : B}{S \vdash \mathbf{fun}\,(X :: K)\,b : \mathbf{Fun}\,(X :: K) : B}\ [Value\ \forall\ introduction] \qquad \frac{S \vdash f : \mathbf{Fun}\,(X :: K) : A \quad S \vdash B :: K}{S \vdash f\,(B) : \langle B \to X \rangle A}\ [Value\ \forall\ elimination]$$

Die Regeln *[Kind basic]* und *[Kind oper]* legen die Wohlgeformtheit von Sorten fest. Die Typregeln *[Type introduction]* und *[Type elimination]* sind entsprechend den Regeln *[Value → introduction]* und *[Value → elimination]* definiert. *[Type introduction]* zeigt, wie die Sorte eines Typoperators abgeleitet wird. Die Regel *[Type elimination]* besagt, daß die Typoperatorapplikation $A\,(B)$ die Sorte L besitzt, wenn der Typoperator A die Sorte $\mathbf{OPER}\,(X :: K) : L$ und der Typ B die Sorte K hat.

Typüberprüfungsalgorithmus

In Abschnitt 4.2 werden die Typregeln *[Type →]* und *[Value → introduction]* modifiziert, um die wiederholte Validierung des statischen Kontextes im Algorithmus zu vermeiden. Aus demselben Grund sind die Typregeln *[Kind oper]* und *[Type introduction]* jeweils um eine weitere Prämisse zu ergänzen, die die Sorte K auf Wohlgeformtheit prüft, so daß in der zweiten Prämisse der K enthaltende statische Kontext ebenfalls korrekt ist:

$$\frac{S \vdash K\ kind \quad S, X :: K \vdash L\ kind}{S \vdash \mathbf{OPER}\,(X :: K) : L\ kind}\ [Kind\ oper\ alg] \qquad \frac{S \vdash K\ kind \quad S, X :: K \vdash A :: L}{S \vdash \mathbf{Oper}\,(X :: K)\,A :: \mathbf{OPER}\,(X :: K) : L}\ [Type\ introduction\ alg]$$

Der in Abbildung 4.5 für F_∞ vorgestellte Typüberprüfungsalgorithmus zeigt die gegenüber F_1 vorgenommenen Erweiterungen der Funktionen *hastype* und *haskind* und die zur Überprüfung der Wohlgeformtheit von Sorten dienende Funktion *iskind*.

```
let hastype(S :_Env  a :_Value  A :_Type) :Bool =
    ...
    elsif a = fun (X :: K) b and iskind(S K) and          (* Regel: Value ∀ introduction *)
        hastype(S,X :: K b B) then A = Fun (X :: K): B
    elsif a = b (B) and hastype(S b Fun (X :: K): C) and  (* Regel: Value ∀ elimination *)
        haskind(S B K) then A = ⟨B → X⟩C
    ...

let haskind(S :_Env  A :_Type  K :_Kind) :Bool =
    ...
    elsif A = Fun (X :: L): B and iskind(S L) and         (* Regel: Type ∀ *)
        haskind(S,X :: L B TYPE) then K = TYPE
    elsif A = Oper (X :: L) B and iskind(S L) and         (* Regel: Type introduction *)
        haskind(S,X :: L B M) then K = OPER (X :: L): M
    elsif A = B (C) and haskind(S B OPER (X :: L): M)     (* Regel: Type elimination alg *)
        and haskind(S C L) then K = ⟨C → X⟩M
    ...

let iskind(S :_Env  K :_Kind) :Bool =
    if K = TYPE then true                                 (* Regel: Kind basic *)
    elsif K = OPER (X :: K): L and                        (* Regel: Kind oper alg *)
        iskind(S K) then iskind(S,X :: K L)
    else false
    end
```

Abbildung 4.5: Typüberprüfungsalgorithmus von F_∞.

Die Funktion *iskind* implementiert die Typregeln *[Kind basic]* und *[Kind oper alg]* und wird in den Funktionen *hastype* und *haskind* benutzt. Gegenüber F_2 bleibt die Funktion *lookupTypeIde* unverändert. In F_∞ können zwei Typen A und B textuell verschieden und dennoch äquivalent sein, kurz $A \stackrel{\beta}{=} B$, wenn A das Ergebnis der β-Reduktion einer Typoperatorapplikation in B ist, wobei der Typoperator ein **Oper**–Ausdruck sein muß. Der **Oper**–Ausdruck kann in beliebiger Tiefe in B auftreten. So gilt beispielsweise:

$$\{\mathbf{Oper}(X :: \mathbf{TYPE})\, X\}\,(Int) \stackrel{\beta}{=} \langle Int \to X\rangle X \;=\; Int$$
$$A\,(B\,(\{\mathbf{Oper}(X :: K)\, C\}\,(D))) \stackrel{\beta}{=} A\,(B\,(\langle D \to X\rangle C))$$

Formal ist die β-Äquivalenz $\stackrel{\beta}{=}$ die reflexive, transitive und symmetrische Hülle der rekursiv über Typen und Sorten wie folgt definierten '$\stackrel{\beta}{\longrightarrow}$' Reduktion:

$$\frac{}{(\mathbf{Oper}(X :: K)\, A)\,(B) \stackrel{\beta}{\longrightarrow} \langle B \to X\rangle A}\ (\beta)$$

$$\frac{A \stackrel{\beta}{\longrightarrow} A'}{A\,(B) \stackrel{\beta}{\longrightarrow} A'\,(B)} \qquad \frac{B \stackrel{\beta}{\longrightarrow} B'}{A\,(B) \stackrel{\beta}{\longrightarrow} A\,(B')}$$

$$\frac{A \stackrel{\beta}{\longrightarrow} A'}{\mathbf{Fun}(x : A) : B \stackrel{\beta}{\longrightarrow} \mathbf{Fun}(x : A') : B} \qquad \frac{B \stackrel{\beta}{\longrightarrow} B'}{\mathbf{Fun}(x : A) : B \stackrel{\beta}{\longrightarrow} \mathbf{Fun}(x : A) : B'}$$

$$\frac{K \stackrel{\beta}{\longrightarrow} K'}{\mathbf{Fun}(X :: K) : A \stackrel{\beta}{\longrightarrow} \mathbf{Fun}(X :: K') : A} \qquad \frac{A \stackrel{\beta}{\longrightarrow} A'}{\mathbf{Fun}(X :: K) : A \stackrel{\beta}{\longrightarrow} \mathbf{Fun}(X :: K) : A'}$$

$$\frac{K \stackrel{\beta}{\longrightarrow} K'}{\mathbf{Oper}(X :: K)\, A \stackrel{\beta}{\longrightarrow} \mathbf{Oper}(X :: K')\, A} \qquad \frac{A \stackrel{\beta}{\longrightarrow} A'}{\mathbf{Oper}(X :: K)\, A \stackrel{\beta}{\longrightarrow} \mathbf{Oper}(X :: K)\, A'}$$

$$\frac{}{\mathbf{TYPE} \stackrel{\beta}{\longrightarrow} \mathbf{TYPE}} \qquad \frac{K \stackrel{\beta}{\longrightarrow} K'}{\mathbf{OPER}(X :: K) : L \stackrel{\beta}{\longrightarrow} \mathbf{OPER}(X :: K') : L}$$

$$\frac{L \stackrel{\beta}{\longrightarrow} L'}{\mathbf{OPER}(X :: K) : L \stackrel{\beta}{\longrightarrow} \mathbf{OPER}(X :: K) : L'}$$

Die '$\stackrel{\beta}{\longrightarrow}$' Reduktion führt in der Regel (β) Typoperatorapplikationen aus, falls der Typoperator kein Bezeichner, sondern ein **Oper** Ausdruck ist. Die restlichen Regeln definieren die '$\stackrel{\beta}{\longrightarrow}$' Reduktion induktiv für alle Sorten und Typen aus der '$\stackrel{\beta}{\longrightarrow}$' Reduktion der Typkomponenten. Die Funktion $betanorm(A)$ mit dem Argument A bildet die eindeutige und immer existierende Normalform von A bezüglich der Relation '$\stackrel{\beta}{=}$'. Der Beweis der strengen Normalisierung übersteigt den Rahmen dieser Arbeit und findet sich in [Steffen, Pierce 94]. In dem Typ $betanorm(A)$ sind die in A enthaltenen Typoperatorapplikationen der Form $\{\mathbf{Oper}\,(X :: K)\,B\}\,(C)$ aufgelöst. Ein Programm p mit dem Typ A ist vor der Typüberprüfung zu normalisieren:

hastype$(\oslash\ betanorm(p)\ A)$

4.5 Getypter Lambda–Kalkül mit Subtypisierung

Der Lambda–Kalkül $F_{\infty}^{\leq}$ ist eine Erweiterung von F_{∞}, das in vielen wissenschaftlichen Veröffentlichungen die Grundlage zur Untersuchung objekt-orientierter Programmiersprachen ist [Bruce 94; Hofmann, Pierce 92; Hofmann, Pierce 95; Pierce 92b; Pierce, Turner 94]. Die Kernerweiterung stellt das Subtypkonzept dar, das in Abschnitt 3.2 vorgestellt wird. Die Sprache enthält zusätzlich den Typ **Top**, das größte Typelement in der Sorte **TYPE**. Eine neue Sorte in der Sprache ist die *Potenzsorte* (*power-kind*), die die Potenzmenge eines gewöhnlichen Typs der Sorte **TYPE** beschreibt. Beispielsweise bezeichnet die Sorte

$$\boldsymbol{POWER}(\boldsymbol{Fun}\,(x : Int) : Int)$$

die Menge von Funktionstypen, die eine Potenzmenge von $\mathbf{Fun}\,(x : Int) : Int$ ist.

Syntax

Die Grammatik für Werte in $F_{\infty}^{\leq}$ bleibt gegenüber der Grammatik von F_{∞} unverändert. Ein neuer Typ ist der Typ **Top**:

Type ::= ...
  | ***Top*** *Maximaltyp in* ***TYPE***

Als neue Sorte ist die Potenzsorte in der Sprache definiert:

Kind ::= ...
  | ***POWER*** *"(" Type ")"* *Potenzmengensorte*

Typsystem

Nachfolgend ist das gegenüber F_{∞} veränderte Typsystem von $F_{\infty}^{\leq}$ dargestellt:

Sorten:

[Kind power]

$$\frac{S \vdash A :: \mathbf{TYPE}}{S \vdash \mathbf{POWER}(A)\ kind}$$

Subsorten:

[Subkind type type]

$$\frac{\vdash S\ sig}{S \vdash \mathbf{TYPE} <:: \mathbf{TYPE}}$$

[Subkind oper]

$$\frac{S \vdash L_1 <:: K_1 \qquad S, X :: L_1 \vdash K_2 <:: L_2}{S \vdash \mathbf{OPER}\,(X :: K_1) :\ K_2 <:: \mathbf{OPER}\,(X :: L_1) :\ L_2}$$

[Subkind power type]
$$\frac{S \vdash A :: \mathbf{TYPE}}{S \vdash \mathbf{POWER}(A) <:: \mathbf{TYPE}}$$

[Subkind power power]
$$\frac{S \vdash A <: B}{S \vdash \mathbf{POWER}(A) <:: \mathbf{POWER}(B)}$$

Typen:

[Type top]
$$\frac{\vdash S \ sig}{S \vdash \mathbf{Top} :: \mathbf{TYPE}}$$

[Type subsumption]
$$\frac{S \vdash A :: K \quad S \vdash K <:: L}{S \vdash A :: L}$$

[Type elimination]
$$\frac{S \vdash A :: \mathbf{OPER}\,(X :: K) : L \quad S \vdash B :: K}{S \vdash A\,(B) :: \langle B \to X \rangle L}$$

Subtypen:

[Subtype top]
$A :: \mathbf{TYPE} \Longleftrightarrow A <: \mathbf{Top}$

[Subtype power]
$A :: \mathbf{POWER}(B) \Longleftrightarrow A <: B$

[Subtype reflexive]
$$\frac{S \vdash A :: \mathbf{TYPE}}{S \vdash A <: A}$$

[Subtype transitive]
$$\frac{S \vdash A <: B \quad S \vdash B <: C}{S \vdash A <: C}$$

[Subtype beta]
$$\frac{S \vdash A <: B \quad S \vdash B \stackrel{\beta}{=} C}{S \vdash A <: C}$$

[Subtype $\to$]
$$\frac{S \vdash B_1 <: A_1 \quad S \vdash A_2 <: B_2}{S \vdash \mathbf{Fun}\,(x : A_1) : A_2 <: \mathbf{Fun}\,(x : B_1) : B_2}$$

[Subtype $\forall$]
$$\frac{S \vdash L <:: K \quad S, X :: L \vdash A <: B}{S \vdash \mathbf{Fun}\,(X :: K) : A <: \mathbf{Fun}\,(X <: L) : B}$$

[Subtype app]
$$\frac{\begin{array}{c} S \vdash A :: \mathbf{OPER}\,(X :: K) : \mathbf{POWER}(D) \\ S \vdash B :: \mathbf{OPER}\,(X :: K) : \mathbf{POWER}(E) \\ S \vdash D <: E \quad S \vdash C :: K \end{array}}{S \vdash A\,(C) <: B\,(C)}$$

Werte:

[Value subsumption]
$$\frac{S \vdash a : A \quad S \vdash A <: B}{S \vdash a : B}$$

Die Typregel *[Kind power]* fügt Potenzsorten als neue Sortenart in das Typsystem ein. Die Regeln *[Subkind type type]*, *[Subkind power type]*, *[Subkind power]* und *[Subkind oper]* drücken eine Teilmengenbeziehung zwischen Sorten aus, die der Subtypisierung auf Typebene entspricht. Neu ist die Regel *[Type top]*: **Top** ist in jedem wohlgeformten Kontext ein gültiger Typ. Ebenfalls neu im Typsystem ist die Typregel *[Type subsumption]*. Für Obermengen L einer Sorte K bleibt die '::'–Beziehung erhalten. Die Typregel *[Type elimination]* unterscheidet sich in der Konklusion von der F_∞ Variante, um eine zusätzliche Typsubstitution in Sortenstrukturen. Bei den Werten hinzugekommen ist die Subsumptionsregel. Ein Wert a vom Typ A hat alle Supertypen von A als Typ. Die Subtypregeln definieren Subtypbeziehungen zwischen Typen. Die Subtypisierung ist reflexiv (*[Subtype reflexiv]* und transitiv (*[Subtype transitive]*). Für Funktionstypen gilt die Kontravarianzregel (*[Subtype →]*, *[Subtype ∀]*). Gemäß der Regel *[Subtype power]* kann eine Subtypbeziehung $A <: B$ durch eine Aussage $A :: \mathbf{POWER}(B)$ mit einer Potenzsorte ersetzt werden und umgekehrt. Ebenfalls gleichwertig sind die Aussagen $A <: \mathbf{Top}$ und $A :: \mathbf{TYPE}$. Die Regel *[Subtype beta]* besagt, daß die Subtypbeziehung für zwei β-äquivalente Typen erhalten bleibt. Mit der Regel können beispielsweise Subtypaussagen wie:

$$S \vdash \{\boldsymbol{Oper}(X :: \boldsymbol{TYPE})\, X\}\,(Int) <: Int \quad oder$$
$$S \vdash Int <: \{\boldsymbol{Oper}(X :: \boldsymbol{TYPE})\, X\}\,(Int)$$

abgeleitet werden.

Substitution

Die Substitution in Sorten in der Regel *[Type elimination]* kann induktiv auf der Sortenstruktur erfolgen. Eine Typersetzung kann nur in Potenzsorten erfolgen, da nur diese Sortenart Typbezeichner enthalten kann.

$$\begin{array}{lllll}
1) & \langle A \to X\rangle \boldsymbol{TYPE} & = & \boldsymbol{TYPE} \\
2) & \langle A \to X\rangle \boldsymbol{POWER}(B) & = & \boldsymbol{POWER}(\langle A \to X\rangle B) \\
3) & \langle A \to X\rangle \boldsymbol{OPER}\,(X :: K) : L & = & \boldsymbol{OPER}\,(X :: K) : L \\
4) & \langle A \to X\rangle \boldsymbol{OPER}\,(Y :: K) : L & = & \boldsymbol{OPER}\,(Y :: \langle A \to X\rangle K) : \langle A \to X\rangle L
\end{array}$$

Typüberprüfungsalgorithmus

Das Typsystem von $F^{\leq}_{\infty}$ ist nicht eindeutig rückwärts interpretierbar. Die Regeln *[Type subsumption]*, *[Type transitive]*, *[Value subsumption]*, *[Subtype transitive]* und *[Subtype beta]* müssen eliminiert werden, da ihre wiederholte Ausführung in automatischen Beweisern zur Nichtterminierung führt. Dafür sind andere Typregeln zu modifizieren.

Um die Typregel *[Value subsumption]* streichen zu können, ist nur die Regel *[Value → elimination]* mit der Subsumptionsregel zu vereinen. Die Funktion f darf mit einem Wert a aufgerufen werden, dessen Typ A' ein Subtyp des formalen Parametertyps A ist. Gleichzeitig kann f einen Funktionstyp F besitzen, der Subtyp eines Funktionstyps ist. Die Regel *[Value subsumption]* wird demnach zweifach in der Regel *[Value → elimination]* angewendet, einmal für das Funktionsargument von f und ein weiteres Mal für die Funktion f selbst:

[Value → elimination alg]

$$\frac{S \vdash f : F \quad S \vdash F <: \mathbf{Fun}(x : A) : B \quad S \vdash a : A' \quad S \vdash A' <: A}{S \vdash f(a) : B}$$

Die Typregel *[Subtype transitive]* kann entfallen und wird durch die Regeln *[Subtype ide alg]* und *[Subtype app alg]* ersetzt:

[Subtype ide alg]

$$\frac{S_1, X :: \mathbf{POWER}(A), S_2 \vdash A <: B}{S_1, X :: \mathbf{POWER}(A), S_2 \vdash X <: B}$$

[Subtype app alg]

$$\frac{S \vdash A :: \mathbf{OPER}(X :: K) : \mathbf{POWER}(B) \quad S \vdash \langle D \to X \rangle B <: C \quad S \vdash D :: K}{S \vdash A(D) <: C}$$

Da die Transitivität von Funktionstypen aus der Kontravarianzregel und der Transitivität von Argument- und Rückgabetypen folgt, sind nur für Typbezeichner und Typoperatorapplikation Regeln anzugeben. In Regel *[Subtype ide alg]* ist, um $X <: B$ zu testen, der Subtyptest $A <: B$ zu beweisen. Der Typ A ist das größte Typelement aus der Potenzmenge $\mathbf{POWER}(A)$. Aus Regel *[Subtype power]* folgt bereits $X <: A$. Aus $A <: B$ und der Transitivität folgt schließlich die Aussage $X <: B$, die auch in der Konklusion der Typregel steht. Entsprechend kann bei der Typoperatorapplikation argumentiert werden. In *[Subtype app alg]* ist, um $A(D) <: C$ zu testen, $\langle D \to X \rangle B <: C$ zu zeigen, wobei $\langle D \to X \rangle B$ das größte Typelement aus der Potenzmenge von $A(D)$ ist. Durch Transitivität folgt schließlich die Konklusion der Typregel.

Die Typregeln *[Value ∀ elimination]*, *[Type elimination]* und *[Subtype app alg]* sind zu modifizieren, um die Typregel *[Type subsumption]* streichen zu können. Die Subsumptionsregel wird dazu in die drei Typregeln integriert:

[Value ∀ elimination alg]

$$\frac{S \vdash f : F \quad S \vdash F <: \mathbf{Fun}(X :: K) : A \quad S \vdash B :: K' \quad S \vdash K' <:: K}{S \vdash f(B) : \langle B \to X \rangle A}$$

[Type elimination alg]

$$\frac{S \vdash A :: \mathbf{OPER}(X :: K) : L \quad S \vdash B :: K' \quad S \vdash K' <:: K}{S \vdash A(B) :: \langle B \to X \rangle L}$$

[Subtype app alg]

$$\frac{S \vdash A :: \mathbf{OPER}(X :: K) : \mathbf{POWER}(B) \quad S \vdash \langle D \to X \rangle B <: C \quad S \vdash D :: K' \quad S \vdash K' <:: K}{S \vdash A(D) <: C}$$

Die Subtypregel *[Subtype beta]* ist durch die rückwärts interpretierbare Regel *[Subtype beta alg]* zu ersetzen:

[Subtype beta alg]

$$\frac{S \vdash betanorm(A) <: betanorm(B)}{S \vdash A <: B}$$

Um die Subtypbeziehung von zwei Typen A und B zu prüfen, sind diese zu normalisieren.

```
let lookupValueIde(S :_Env  x :_Ide  A :_Type) :Bool =
    ...

let lookupTypeIde(S :_Env  X :_Ide  K :_Kind) :Bool =
    ...

let hastype(S :_Env  a :_Value  A :_Type) :Bool =
    if a = x then lookupValueIde(S x A)                          (* Regel: [Value ide] *)
    elsif a = fun (x:B) c and                                    (* Regel: [Value → introduction alg] *)
        haskind(S B TYPE) and
        hastype(S,x:B c C) then A = Fun (x:B): C
    elsif a = f(b) and hastype(S f F) and                        (* Regel: [Value → elimination alg] *)
        subtype(S F Fun (x:B): C) and hastype(S b B')
        and subtype(S B' B) then A = C
    elsif a = fun (X :: K) b and iskind(S K) and                 (* Regel: Value ∀ introduction *)
        hastype(S,X :: K b B) then A = Fun (X :: K): B
    elsif a = f(B) and hastype(S f F) and                        (* Regel: Value ∀ elimination alg *)
        subtype(S F Fun (X :: K): C) and haskind(S B K')
        and subkind(S K' K) then A = ⟨B → X⟩C
    else false
end

let haskind(S :_Env  A :_Type  K :_Kind) :Bool =
    if A ∈ {Bool,Int,String} then K = TYPE                       (* Regel: [Type builtin] *)
    elsif A = X then lookupTypeIde(S X K)                        (* Regel: Type Ide *)
    elsif A = Fun (x:B): C and haskind(S B TYPE) and             (* Regel: [Type →] *)
        haskind(S,x:B C TYPE) then K = TYPE
    elsif A = Fun (X :: L): B and iskind(S L) and                (* Regel: Type ∀ *)
        haskind(S,X :: L B TYPE) then K = TYPE
    elsif A = Oper (X :: L) B and iskind(S L) and                (* Regel: Type introduction alg *)
        haskind(S,X :: L B M) then K = OPER (X :: L): M
    elsif A = B (C) and haskind(S B OPER (X :: L): M)            (* Regel: Type elimination alg *)
        and haskind(S C L) then K = ⟨C → X⟩M
    else false
    end
```

Abbildung 4.6: Typüberprüfungsalgorithmus von $F^{\leq}_{\infty}$.

```
let iskind(S :_Env  K :_Kind) :Bool =
    if K = TYPE then true                                   (* Regel: Kind basic *)
    elsif K = OPER (X :: K):L and                           (* Regel: Kind oper alg *)
        iskind(S K) then iskind(S,X :: K L)
    elsif K = POWER(A) then haskind(S A TYPE)               (* Regel: Kind power *)
    else false
    end

let subtype(S :_Env  A :_Type  B :_Type) :Bool =
    if A = B then true                                      (* Regel: Subtype reflexive *)
    elsif A = Fun (x:C1): C2 and B = Fun (x:D1): D2 then    (* Regel: Subtype → *)
        subtype(S D1 C1) then subtype(S C2 D2)
    elsif A = Fun (X :: K): C and B = Fun (X :: L): D and   (* Regel: Subtype ∀ *)
        subkind(S L K) then subtype(S,X :: L C D)
    elsif A = X and lookupTypeIde(S X POWER(C))             (* Regel: Subtype ide alg *)
        then subtype(S C B)
    elsif A = C(D) and haskind(S D K') and                  (* Regel: Subtype app alg *)
        haskind(S C OPER (X :: K):POWER(E)) and
        subkind(S K' K) then subtype(S ⟨D → X⟩E B)
    else false
    end

let subkind(S :_Env  K :_Kind  L :_Kind) :Bool =
    if K = TYPE and L = TYPE then true                      (* Regel: Subkind type type *)
    elsif K = POWER(A) and L = TYPE then true               (* Regel: Subkind power type *)
    elsif K = POWER(A) and L = POWER(B) then                (* Regel: Subkind power *)
        subtype(S A B)
    elsif K = OPER (X :: K1): K2 and                        (* Regel: Subkind oper *)
        L = OPER (X :: L1): L2 and
        subkind(S L1 K1) then subkind(S,X :: K1 K2 L2)
    else false
    end
```

Fortsetzung von Abbildung 4.6: Typüberprüfungsalgorithmus von $F^{\leq}_{\infty}$.

Der aus dem geänderten Typsystem von $F_\infty^\leq$ abgeleitete Typüberprüfungsalgorithmus ist in Abbildung 4.6 dargestellt. Wie bereits in F_∞ ist ein Programm p vor der Typüberprüfung zu normalisieren.

$F_\infty^\leq$ ohne Sortennotation

Durch konsequente Anwendung der Äquivalenzregeln *[Subtype power]* und *[Subtype top]* kann mit der Transformationsfunktion τ die Sprache $F_\infty^\leq$ mit expliziter Sortennotation in eine äquivalente Sprache ohne Sortennotation überführt werden:

$$
\begin{array}{lcl}
\tau(\mathbf{TYPE}) & = & \mathbf{Top} \\
\tau(\mathbf{POWER}(A)) & = & \tau(A) \\
\tau(\mathbf{OPER}\,(X :: K) :\, L) & = & \mathbf{Oper}\,(X <: \tau(K))\,\tau(L) \\
 & & \\
\tau(X) & = & X \\
\tau(\mathbf{Fun}\,(X :: K) :\, A) & = & \mathbf{Fun}\,(X <: \tau(K)) : \tau(A) \\
\tau(\mathbf{Fun}\,(x : A) :\, B) & = & \mathbf{Fun}\,(x : \tau(A)) : \tau(B) \\
\tau(A\,(B)) & = & \tau(A)\,(\tau(B)) \\
 & & \\
\tau(x) & = & x \\
\tau(\mathbf{fun}\,(X :: K)\,a) & = & \mathbf{fun}\,(X <: \tau(K))\,\tau(a) \\
\tau(\mathbf{fun}\,(x : A)\,a) & = & \mathbf{fun}\,(x : \tau(A))\,\tau(a) \\
\tau(f\,(B)) & = & \tau(f)\,(\tau(B)) \\
\tau(f\,(a)) & = & \tau(f)\,(\tau(a))
\end{array}
$$

Die Funktion τ ersetzt Sortensignaturen der Form X :: K in Subtypsignaturen $X <: \tau(K)$. Sorten werden dabei auf Typen abgebildet. Die Sorte **TYPE** wird auf den Typ **Top** abgebildet. τ angewendet auf Potenzsorten liefert das Argument des Potenzsortenkonstruktors. Typoperatorsorten werden auf Typoperatortypen abgebildet. Zum Beispiel gilt:

$$
\begin{array}{lcl}
\tau(\mathbf{Fun}\,(X :: \mathbf{POWER}(Int)) : X) & = & \mathbf{Fun}\,(X <: Int) : X \\
\tau(\mathbf{Oper}(X :: \mathbf{TYPE})\,\mathbf{POWER}(X)) & = & \mathbf{Oper}\,(X <: \mathbf{Top})\,X
\end{array}
$$

Die Syntax, das Typsystem und der Typüberprüfungsalgorithmus von $F_\infty^\leq$ ohne Sortennotation sind im Anhang F aufgeführt. Das Typsystem ist kleiner, da auf Typregeln verzichtet werden kann, die die Wohlgeformtheit und die Teilmengenbeziehung von Sorten ausdrücken. Für Typoperatoren wird mit der Regel *[Subtype oper]* eine Subtypbeziehung definiert, die der Kontravarianz von Funktionstypen entspricht:

[Subtype oper]

$$
\frac{S \vdash B_1 <: A_1 \qquad S, X <: B_1 \vdash A_2 <: B_2}{S \vdash \mathbf{Oper}\,(X <: A_1)\,A_2 <: \mathbf{Oper}\,(X <: B_1)\,B_2}
$$

Funktionstypen und Basistypen sind Subtypen von **Top** . Für Typoperatoren gilt diese Eigenschaft nicht:

[Subtype builtin ok]

$$\frac{\vdash S \ sig \quad A \in \{Bool, Int, String\}}{S \vdash A <: \mathbf{Top}}$$

[Subtype → ok]

$$\frac{S \vdash \mathbf{Fun}(x : A) : B \ type}{S \vdash \mathbf{Fun}(x : A) : B <: \mathbf{Top}}$$

[Subtype ∀ ok]

$$\frac{S \vdash \mathbf{Fun}(X <: A) : B \ type}{S \vdash \mathbf{Fun}(X <: A) : B <: \mathbf{Top}}$$

Die Wohlgeformtheit von Typen wird im Typüberprüfungsalgorithmus mit der Funktion *istype* geprüft. Dazu ist ein Vergleich zwischen Sorten nicht notwendig.

Die theoretischen Eigenschaften von $F^{\leq}_{\infty}$ werden in [Steffen, Pierce 94] beschrieben. Es wird ein Algorithmus präsentiert, um die Aussage $S \vdash A <: B$ für zwei wohlgeformte Typen zu entscheiden. Dieser Algorithmus ist vollständig (*complete*) und korrekt (*sound*), d.h. er terminiert für typkorrekte Terme.

Pierce beweißt in [Pierce 92a] die Unentscheidbarkeit der Subtypaussage in $F^{\leq}_{2}$, einem getypten Lambda–Kalkül zweiter Ordnung mit gebundenem parametrischen Polymorphismus und einem Maximaltypen *Top*. Er wiederlegt damit eine zuvor in [Ghelli 90] aufgestellte Behauptung, daß die Typüberprüfung in $F^{\leq}_{2}$ entscheidbar ist. Als Grundlage des Beweises dient die Beschreibung divergierender Terme der Sprache, bei denen der Typprüfer nicht terminiert [Ghelli 95]. Die Divergenz der Terme beruht auf der impliziten Substitution in der Regel *[Subtype ∀]*.

[Subtype ∀]

$$\frac{S \vdash B_1 <: A_1 \quad S, X <: B_1 \vdash A_2 <: B_2}{S \vdash \mathbf{Fun}(X <: A_1) : A_2 <: \mathbf{Fun}(X <: B_1) : B_2}$$

Die Rückwärtsanwendung der Regel ermöglicht die Konstruktion einer Prämisse mit einer im Vergleich zur Konklusion komplexeren Struktur. Aus der Unentscheidbarkeit von $F^{\leq}_{2}$ folgt die Unentscheidbarkeit von $F^{\leq}_{\infty}$. Aufgrund der künstlichen Erscheinungsform von divergierenden Termen hat die Unentscheidbarkeit von $F^{\leq}_{\infty}$ auf die Programmierpraxis keinen Einfluß.

Das folgende Programmstück zeigt einen einfach strukturierten divergierenden TL-Term:

```
Let A = Fun(X <:Ok)
          :Fun(f :Fun( <:X)
                     :Fun(f :X):Ok
               ) :Ok

Let B = Fun(Z <:A)
          :Fun(f :A):Ok
Let X <:B = A                ⇒  Nichtterminierung
```

5. TLMIN

Dieses Kapitel beschreibt Aufgaben und Lösungsansätze der Typüberprüfung in der polymorphen, strikt typisierten und im Kern funktionalen Programmiersprache TLMIN[1]. Die Sprache ist gegenüber TL im Umfang reduziert und kann durch Syntaxerweiterung zu TL erweitert werden. Die Sprache ist ein Lambda–Kalkül höherer Ordnung, das im Gegensatz zu $F_{\infty}^{\leq}$ effiziente und leistungsfähige Sprachkonstrukte enthält, die sich besonders zur Entwicklung datenintensiver Anwendungen eignen. Ein neu in TLMIN implementiertes Konzept zur Modellierung von Vererbungsbeziehungen objektorientierter Programmiersprachen ist der rekursive Polymorphismus (*f–bounded–polymorphism*).

5.1 Syntax der Sprache

Dieser Abschnitt stellt die konkrete und abstrakte Syntax von TLMIN vor. Desweiteren werden die während der Syntaxanalyse durchgeführten Normalisierungen präsentiert. Ein weiterer Unterabschnitt befaßt sich mit der Syntaxerweiterung zu TL.

5.1.1 Konkrete Syntax

Die konkrete Syntax von TLMIN ist im Anhang H aufgeführt. Die Sprache ist gegenüber $F_{\infty}^{\leq}$ um folgende Sprachelemente erweitert:

Typen: Neben Funktionstypen, Typoperatoren und einfachen Typbezeichnern enthält die Sprache:

- Tupel-, Record- und Ausnahmetypkonstruktoren (**Tuple**, **Record**, **Exception**),
- Qualifizierte Typbezeichner, die Tupel-, Record- und Ausnahmetypkomponenten eindeutig identifizieren, z.B. $list.T$,
- den Subtypen **Nok** aller nichtparametrisierten Typen und
- rekursive Typen.

Werte: Neben Funktionswerten, Funktionsapplikationen und Wertbezeichnern enthält die Sprache:

- den Feldkonstruktor **array** und die Feldelementselektion '[]',

[1] TL minimized

- den Tupelkonstruktor **tuple**, den Variantenkonstruktor **tuple_case**, den Recordkonstruktor **record** und den Ausnahmekonstruktor **exception**,
- die Tupel-, Record- und Ausnahmeselektion $a.x$ mit einem Wert a und dem Bezeichner x,
- die Recorderweiterung **extend**,
- die bedingte Auswertung **if**, das Schleifenkonstrukt **loop** und die Fallunterscheidung für Varianten **case**,
- den sequentiellen Programmblock **begin end**,
- Konstrukte zur Ausnahmebehandlung **try**, **raise** und **reraise** und
- den kanonischen Wert **ok** des Typs **Ok**.

Bindungen: für (wechselseitig) rekursive und nichtrekursive Werte und Typen.

Signaturen: Neben der von $F_{\infty}^{\leq}$ bekannten Wert- und Typsignatur gibt es:

- eine weitere Typsignatur, die nichtabstrakte Typbezeichner an Typen bindet. Diese Typsignaturen entsprechen den Typbindungen auf der Wertebene. Beispiel:

  Let *A* = ***Tuple Let*** *T* = *Int* *t :T* ***end***
  let *f(**Let** X = Int x :X)* = *x*

  Der Typbezeichner A wird in der ersten Beispielzeile in einer Typbindung definiert und der Typ T in einer Signatur. Typdefinierende Signaturen sind orthogonal in die Sprache integriert, d.h. sie können überall verwendet werden, wo auch Subtypsignaturen zulässig sind. In der zweiten Beispielzeile tritt eine Typsignatur in der formalen Parameterliste einer Funktion auf.
- rekursive Varianten der beiden Typsignaturarten (siehe Abschnitt 5.7.5 und 5.7.2).

5.1.2 Syntaxerweiterung zu TL

Im Vergleich mit TL ist TLMIN eine im Umfang reduzierte Sprache. Es fehlen in TLMIN die folgenden Sprachkonstrukte:

- Werte: **andif**, **orif**, **while**, **elsif**, **assert**, **for**, Variantentest und -projektion '?' bzw. '!'
- Bindungen: Parallele Bindungen mit **and** und das **open** Konstrukt
- Signaturen: das **Repeat** Konstrukt
- Schnittstellen und Module

Die fehlenden Sprachelemente können durch Syntaxerweiterung auf den Sprachkern von TLMIN abgebildet werden, so daß neben dem bereits bestehenden TL Übersetzer ein zweiter Übersetzer zur Verfügung steht. Dies hat für die Typprüferimplementierung zwei Vorteile. Zum einen können bei der Übersetzung von bestehenden TL Programmen Implementierungsfehler sowohl im TLMIN Typprüfer als auch im TL Typprüfer entdeckt werden. Zum zweiten ist es möglich, das Laufzeitverhalten der beiden Typprüfer zu vergleichen, um Stärken und Schwächen der beiden Systeme zu erkennen (vgl. Kapitel 6).

Im folgenden Beispiel ist die Syntaxerweiterung des **while** Konstrukts dargestellt:

```
grammar
   value3:Value |==
      "while" v=value "do" b=bindings "end"
                  => value<<| loop
                                 if v then b else exit end
                              end |>>
end
```

Die Syntaxerweiterung wird mit dem Schlüsselwort **grammar** eingeleitet. Die Grammatik von Werten *value3* wird um einen Ausdruck erweitert, der aus dem Terminal *while*, einem Wert v, dem Terminal *do*, Bindungen b und dem abschließenden Terminal *end* besteht. Dieser Ausdruck wird in der Syntaxanalysephase auf eine **loop** Schleife abgebildet, die einen **if** Ausdruck enthält.

5.1.3 Normalisierung von TLMIN Programmen

Die abstrakte TLMIN Syntax stellt nur eine Teilmenge der konkreten TLMIN Syntax dar. Die erweiterte konkreten Syntax verringert den Programmieraufwand und erhöht die Lesbarkeit des Programmtextes. Eine minimierte abstrakte Syntax vereinfacht zudem die Typprüferimplementierung. Bei der Umformung eines Parsebaums der konkreten Syntax in einen äquivalenten abstrakten Syntaxbaum sind die folgenden Normalisierungsschritte erforderlich:

- Einführung anonymer Bezeichner ("?") in Bindungen und Signaturen:

:A a	$\Rightarrow$	***Let*** *? = A* ***let*** *? = a*
<:A :B	$\Rightarrow$	*? <:A ?:B*

  Ein anonymer Bezeichner "?" ist gleichwertig zu nichtanonymen Bezeichnern.

- Expansion abkürzender Notationen in Bindungen:

let $x(S_1)...(S_n)$ *:A = a*	$\Rightarrow$	***let*** $x =$ ***fun***$(S_1)...$***fun***(S_n) *:A a*
let $x(S_1)...(S_n)$ *= a*	$\Rightarrow$	***let*** $x =$ ***fun***$(S_1)...$***fun***(S_n) *a*
Let $X(S_1)...(S_n)$ *<:A = B*	$\Rightarrow$	***Let*** $X =$ ***Oper***$(S_1)...$***Oper***(S_n) *<:A B*
Let $X(S_1)...(S_n)$ *= A*	$\Rightarrow$	***Let*** $X =$ ***Oper***$(S_1)...$***Oper***(S_n) *A*

- Expansion abkürzender Notationen in Signaturen:

$x_1,...,x_n$ *:A*	$\Rightarrow$	x_1 *:A*$...x_n$ *:A*
var $x_1,...,x_n$ *:A*	$\Rightarrow$	***var*** x_1 *:A*...***var*** x_n *:A*
$X_1,...,X_n$ *<:A*	$\Rightarrow$	X_1 *<:A*$...X_n$ *<:A*
$x(S_1)...(S_n)$ *:A*	$\Rightarrow$	x *:*$\mathbf{Fun}(S_1)...\mathbf{Fun}(S_n)$ *:A*
$X(S_1)...(S_n)$ *<:A*	$\Rightarrow$	X *<:*$\mathbf{Oper}(S_1)...\mathbf{Oper}(S_n)$ *A*

- Umwandlung von Infix und Listfix Funktionsapplikationen in die Präfixnotation:

a infixOp b	$\Rightarrow$	{***infix*** *infixOp(a b)*}
f ***of*** $a_1...a_n$ ***end***	$\Rightarrow$	*f(*$\mathbf{array}$ $a_1...a_n$ ***end****)*

- Umwandlung von Infix Typoperatorapplikationen in die Präfixnotation:

  A InfixOp B $\Rightarrow$ {**infix** *InfixOp(A B)*}

- Ersetzung fehlender **else** Zweige in **if** Ausdrücken durch eine leere Sequenz, die zu **ok** evaluiert:

  if a ***then*** b ***end*** $\Rightarrow$ ***if*** a ***then*** b ***else end***

- Ersetzung fehlender **else** Zweige in **try** Ausdrücken durch einen **reraise** Ausdruck.

- Ersetzung fehlender **else** Zweige in **case** Ausdrücken durch **raise** *_builtin.caseError*.

5.1.4 Abstrakte Syntax

Tabelle 5.1 zeigt die abstrakte Syntax für Signaturen, Bindungen und Typen von TLMIN. Dabei werden syntaktische Objekte definiert, die mit folgenden Metavariablen benannt werden:

$S \in Sigs$	Signaturen
$D \in Binds$	Bindungen
$A \in Type$	Typen
$C \in Case$	Varianten in Tupeltypen
$z \in CaseIde$	Fallmarken in Varianten
$X, Y \in Ide$	Typbezeichner
$x, y \in ide$	Wertbezeichner
$a, b \in Value$	Werte
$p \in Qualifier$	Signaturselektoren

Ein Vergleich mit der abstrakten Syntax von TL in [Matthes 93] zeigt, daß in TLMIN neben nichtrekursiven Signaturen und Bindungen auch rekursive Typsignaturen und Typbindungen existieren, mit deren Hilfe rekursive Typen repräsentiert werden. Eine explizite Notation für rekursive Typen fehlt in TLMIN. Beispielsweise werden die wechselseitig rekursiven Signaturen

Let Rec $X <:$***Ok*** $= F(Y)$ ***and*** $Y <:$***Ok*** $= G(X)$

in der abstrakten Syntax von TLMIN durch zwei rekursive Signaturen dargestellt:

$X \overset{Rec}{<:} F(Y),\ Y \overset{Rec}{<:} F(X)$

In TL ist eine vergleichsweise aufwendige Notation für rekursive Typen erforderlich, um wechselseitig rekursive Typbindungen direkt darstellen zu können. Die rekursive Typdefinition im Beispiel ist in der abstrakten Syntax als eine parallele Bindung von zwei Typvariablen X und Y an zwei unabhängige rekursive Typausdrücke dargestellt, die beide das gesamte rekursive Gleichungssystem beschreiben:

$X =$ ***Rec***$(X, X <:$***Ok*** $= F(Y), Y <:$***Ok*** $= G(X))$

und

$$Y = \textbf{Rec}(Y, X <: \textbf{Ok} = F(Y), Y <: \textbf{Ok} = G(X))$$

Tupeltypen besitzen in der abstrakten Typsyntax immer eine nicht–leere Variantenliste. Ein in der konkreten Syntax definierter Tupeltyp ohne Variante wird durch einen Tupeltyp mit einer anonymen Variante mit leerer Signaturliste repräsentiert. Beispiel:

$$\textbf{Tuple}\ x : Int\ \textbf{end} \quad \Rightarrow \quad \textbf{Tup}(x : Int; \textbf{Case}(?, \varnothing))$$

Mit dieser Notation kann die Subtypisierung von Tupeltypen mit und ohne Varianten in einer Typregel formalisiert werden. Außerdem läßt sich die Subtypbeziehung zwischen einem Tupeltyp ohne Varianten und einem Tupeltyp mit Varianten leichter beschreiben.

In Tabelle 5.2 ist die abstrakte Syntax für Werte von TLMIN aufgestellt. Unterschiede zu TL bezüglich der Notation gibt es nicht.

S	::=	$\oslash$	Leere Signatur
	|	$S, x : A$	Wertsignatur
	|	$S, X <: A$	Subtypsignatur
	|	$S, X = A$	Typdefinition
	|	$S, X = A <: B$	Einschränkende Typdefinition
	|	$S, X \overset{Rec}{<:} A$	Rekursive Subtypsignatur
	|	$S, X \overset{Rec}{=} A$	Rekursive Typdefinition
	;		
D	::=	$\oslash$	Leere Bindung
	|	$D, x = a$	Wertbindung
	|	$D, x : A = a$	Einschränkende Wertbindung
	|	$D, X = A$	Typbindung
	|	$D, X <: A = B$	Einschränkende Typbindung
	|	$S, X \overset{Rec}{=} A$	Rekursive Typbindung
	;		
A	::=	**Ok**	Supertyp aller Typen
	|	**Nok**	Subtyp aller Typen
	|	X	Typbezeichner
	|	$p.X$	Qualifizierter Typbezeichner
	|	Bool | Int | String	Basistypbezeichner
	|	Arr(A)	Feldtyp
	|	**Fun**$(S) : A$	(Polymorpher) Funktionstyp
	|	**Tup**$(S; C)$	Tupeltyp mit Varianten
	|	**Rcd**(S)	Recordtyp
	|	**Exc**(S)	Ausnahmetyp
	|	**Var**(A)	Typ veränderlicher Bindungen
	|	**Oper**$(S)A$	Typoperator
	|	$A(D)$	Typoperatorapplikation
	;		
C	::=	**Case**(z_1, S_1)	Varianten in Tupeltypen
	;		
p	::=	x	Wertbezeichner
	|	X	Typbezeichner
	|	$p.x \mid p.X$	Feldbezeichner
	;		

Tabelle 5.1: Die abstrakte TLMIN Syntax für Signaturen, Bindungen und Typen

a, b	::=	**ok**	Der kanonische Wert des Typs **Ok**
	|	int	Ganzzahlliterale
	|	$real$	Fließkommaliterale
	|	$char$	Zeichenliterale
	|	$string$	Zeichenkettenliterale
	|	x	Wertbezeichner
	|	$a.x$	Tupel-, Record- und Ausnahmefeldselektion
	|	**fun**$(S)a$	(Polymorpher) Funktionskonstruktor
	|	$a(D)$	Funktionsapplikation
	|	**arr**(D)	Feldkonstruktor
	|	$a[b]$	Feldelementselektion
	|	**tup**(D)	Tupelkonstruktor
	|	**tup**$(D; z; A)$	Tupelkonstruktor mit Variante
	|	**rcd**(D)	Recordkonstruktor
	|	**extend**(a, D)	Recorderweiterung
	|	**exc**(a, S)	Ausnahmewertgenerierung
	|	**seq**(D)	Sequentielle Auswertung
	|	**if**(a, b_1, b_2)	Bedingte Auswertung
	|	**case**$(a,$ $(z_{11} \ldots z_{1k_1}, y_1, b_1) \ldots$ $(z_{n1} \ldots z_{nk_n}, y_n, b_n)$,b$)$	Fallunterscheidung $(0 \leq n, 1 \leq k_i)$
	|	**case**$(a,$ $(z_{11} \ldots z_{1k_1}, y_1, b_1) \ldots$ $(z_{n1} \ldots z_{nk_n}, y_n, b_n))$	Vollständige Fallunterscheidung $(1 \leq n, 1 \leq k_i)$
	|	**loop**(D)	Schleife
	|	**exit**	Schleifenterminierung
	|	**try**$(a,$ $(a_1, y_1, b_1) \ldots (a_n, y_n, b_n), b)$	Ausnahmebehandlung $(0 \leq n)$
	|	**raise**(a, D)	Ausnahmeerzeugung
	|	**reraise**	Ausnahmepropagierung
	;		

Tabelle 5.2: Die abstrakte TLMIN Syntax für Werte

5.2 Bezeichner

Die Möglichkeit der Redefinition von Bezeichnern in einer polymorphen Programmiersprache ist entscheidend für die Syntaxbaumrepräsentation der Bezeichner im Typprüfer. Es können drei Möglichkeiten zur Redefinition von Bezeichnern bestehen:

1. Die Redefinition von Bezeichnern ist grundsätzlich nicht erlaubt.
2. Die Redefinition von Bezeichnern ist in einem statischen Kontext nicht erlaubt, falls diese Bezeichner bereits im Kontext enthalten sind.
3. Die Redefinition von Bezeichnern ist uneingeschränkt erlaubt.

Im ersten Fall und zweiten Fall identifizieren die Bezeichnernamen in eindeutiger Weise das ihnen zugewiesene Programmobjekt. Beispiele für Programmiersprachen, die eine Redefinition von Bezeichnern gemäß Punkt zwei erlauben, sind die im letzten Kapitel vorgestellten Lambda–Kalküle. In $F^{\leq}_{\infty}$ ohne Sortennotation erfolgt die Definition von Bezeichnern mit den Typregeln *[Sig ide]* und *[Sig Ide]*:

$$\frac{S \vdash A <: \mathbf{Top} \quad x \notin Dom(S)}{\vdash S, x : A \ sig}\ [Sig\ ide] \qquad \frac{S \vdash A \ type \quad X \notin Dom(S)}{\vdash S, X <: A \ sig}\ [Sig\ Ide]$$

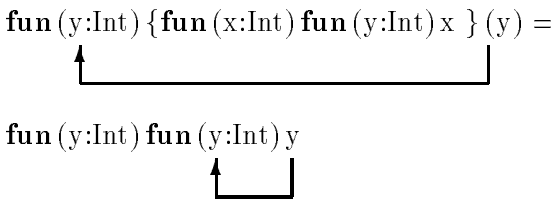
Die Aussage $x \notin Dom(S)$ in der Prämisse von *[Sig ide]* bzw. $X \notin Dom(S)$ in der Prämisse von *[Sig Ide]* verbietet die Redefinition eines Bezeichners x bzw. X, falls dieser Bezeichner bereits im aktuellen statischen Kontext S enthalten ist. Das folgende Beispiel zeigt, daß eine wiederholte Deklaration desselben Bezeichners dennoch möglich ist. Es wird die Wohlgeformtheit eines Funktionstyps abgeleitet, in dem der Bezeichner x zweimal deklariert ist. Die beiden Deklarationen von x erfolgen unabhängig voneinander im leeren Kontext:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \emptyset \ sig}\ [Sig\ empty]}{\emptyset \vdash Int <: \mathbf{Top}}\ [Subtype\ builtin\ ok]}{\vdash x : Int \ sig}\ [Sig\ ide]}{x : Int \vdash Int <: \mathbf{Top}}\ [Subtype\ builtin\ ok]}{\emptyset \vdash \mathbf{Fun}(x : Int) : Int \ type}\ [Type \rightarrow]}{\emptyset \vdash \mathbf{Fun}(x : Int) : Int <: \mathbf{Top}}\ [Subtype \rightarrow ok]}{\vdash x : \mathbf{Fun}(x : Int) : Int \ sig}\ [Sig\ ide]}{x : \mathbf{Fun}(x : Int) : Int \vdash Bool <: \mathbf{Top}}\ [Subtype\ builtin\ ok]}{\emptyset \vdash \mathbf{Fun}(x : \mathbf{Fun}(x : Int) : Int) : Bool \ type}\ [Type \rightarrow]$$

Die Ableitung verdeutlicht das Konzept der Blockstrukturierung im Lambda–Kalkül. Das äußere x im Funktionstyp $\mathbf{Fun}(x : \mathbf{Fun}(x : Int) : Int) : Bool$ ist zum Zeitpunkt der Deklaration des inneren x noch nicht gültig. Obwohl Bezeichner mehrfach deklariert werden können,

sind Namenskonflikte beim Typäquivalenztest nicht möglich. Daraus folgt, daß die Korrektheit der Typüberprüfung von Typbezeichnern gewahrt bleibt, die über den Bezeichnernamen identifiziert werden.

Durch die uneingeschränkte Redefinition von Bezeichnern können jedoch Namenskonflikte entstehen (*name clash problem*) [Field, Harrison 88], die zu einer fehlerhaften Bindung von Bezeichnern an Wert- und Typsignaturen führen. Im folgenden TL Beispiel wird der Wertbezeichner y in den Sichtbarkeitsbereich einer weiteren Deklaration von y substituiert:



Die Pfeile im Beispiel zeigen die Bindung zur deklarierenden Signatur von y. Diese Bindung wechselt bei der Funktionsapplikation. Bezeichnernamen sind zur eindeutigen Indentifizierung von Bezeichnern in TLMIN und TL nicht mehr ausreichend. Drei Lösungsmöglichkeiten bieten sich an:

- Bezeichner werden bei der Substitution umbenannt (α–Konversion).
- Bezeichner werden durch de Bruijn-Indizes repräsentiert. Dieses Konzept wird im TL Typprüfer angewendet.
- Bezeichner werden durch festverzeigerte Objektreferenzen repräsentiert. Dieses Konzept wird im TLMINTypprüfer benutzt.

Die folgenden Abschnitte beschreiben die beiden letzten Konzepte genauer.

5.2.1 De Bruijn–Indizierung

5.2.1.1 Repräsentation von de Bruijn–Indizes

Ein de Bruijn–Index ist eine positive ganze Zahl, geschrieben @i, die den Abstand zwischen definierendem und referenzierendem Auftreten eines Bezeichners im statischen Kontext angibt [de Bruijn 72; Abadi et al. 90]. Jedes referenzierende Auftreten eines Bezeichners im Syntaxbaum wird durch einen de Bruijn–Index ersetzt. Die folgenden Datentypen definieren die Repräsentation von de Bruijn–Indizes im Typprüfer:

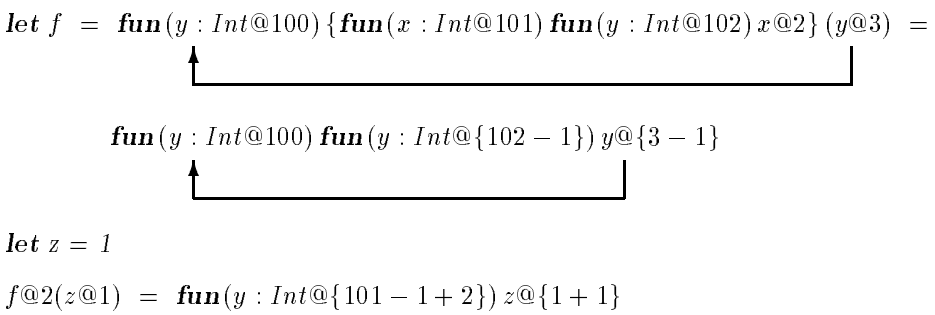
```
Let Ref = Tuple
  name :String
  pos :Source.Position  (* of reference in source text *)
  var index :Int        (* de Bruijn Index, set in scoping phase *)
end

Let Rec T <: Ok = Tuple
```

```
    pos :Source.Position (* of type if declared; of value if inferred *)
case ideCase with
    var ideRefList :list.T(Ref)
    ...
case ...
end
```

Der Typ Ref aus der Schnittstelle $TLIde$ repräsentiert einen einfachen Bezeichner mit den Komponenten $name$ für den Bezeichnernamen, pos für die Position im Quelltext und $index$ für den de Bruijn–Index. Da ein Bezeichner qualifiziert auftreten kann, wird er durch eine Liste $ideRefList$ von einfachen Bezeichnern in der $ideCase$ Variante des Typs $TLType.T$ im TL Typprüfer repräsentiert.

5.2.1.2 Substitution mit de Bruijn–Indizes

Das folgende Beispiel zeigt, wie Namenskonflikte mit de Bruijn–Indizes vermieden werden:



Die Substitutionsoperation $\langle B \rightarrow X \rangle A$ vereinfacht sich zu einer schematischen Indexmanipulation $\langle B \rightarrow i \rangle A$. Bei der Substitution müssen in den folgenden Fällen die Indizes von freien Bezeichnern zusätzlich manipuliert werden (*lifting*), damit die richtigen Bindungen erhalten bleiben:

- Bei der Elimination eines Binders in A wie Funktionswerte, Funktionstypen oder Typoperatoren, müssen alle geschachtelten Indizes von freien Bezeichnern dekrementiert werden. Im Beispiel in Zeile eins wird durch die Funktionsapplikation der Binder **fun** $(x : Int@101)$ eliminiert. Im Resultat sind die Indizes $Int@102$ und $y@3$ um 1 erniedrigt.
- Beim Substituieren von B in den Gültigkeitsbereich eines Binders in A müssen alle freien Indizes in B inkrementiert werden. In der letzten Beispielzeile wird $z@1$ um 1 erhöht, da z in den Gültigkeitsbereich des Binders **fun** $(y : Int@102)$ substituiert wird.

Die Substitutionsoperation im TLTypüberprüfungsalgorithmus ist in [Matthes 93] definiert. Sie ist für alle Syntaxbaumrepräsentationen definiert, die Bezeichner enthalten können, also für Typen, Signaturen und Tupelvarianten. Gleiches gilt für das Inkrementieren und Dekrementieren von Bezeichnern.

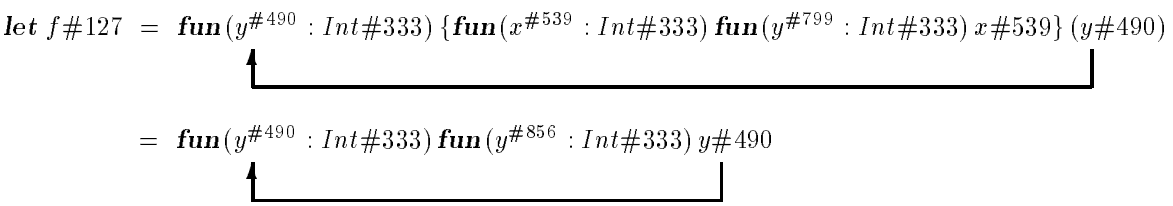
5.2.2 Objektreferenzierung

5.2.2.1 Repräsentation von Objektreferenzen

Objekte im Objektspeicher von TL besitzen eine eindeutige und unveränderliche Objektidentität (Oid), geschrieben #i, die zur Modellierung von Objektbeziehungen verwendet werden kann. Die Objektidentität kann zur Unterscheidung von Bezeichnern verwendet werden, wenn die Speicherobjekte der Signaturen verschiedene Objektidentitäten besitzen. Bei der Objektreferenzierung existiert eine Verzeigerung im Objektspeicher zwischen dem Speicherobjekt, das einen Typ- oder Wertbezeichner repräsentiert, und dem Speicherobjekt für die deklarierende Signatur. Die folgenden Programmzeilen zeigen die Repräsentation von Typbezeichnern im TLMINTypprüfer.

```
Let Rec Type <:Ok=
   Tuple
      pos :Source.Position
   case ideCase with
      first :Signature      (* Objektreferenz zur deklarierenden Signatur *)
      path :Path  (* empty if simple identifier *)
   end
```

Die Tupelvariante *ideCase* aus dem Syntaxbaumtyp $TLAST.Type$ enthält neben einer Quelltextposition *pos* für Fehlerausgaben die Repräsentation der deklarierenden Signatur $first$. Bei qualifizierten Bezeichnern kommt eine nichtleere Pfadangabe *path* hinzu. Das folgende Beispiel zeigt, daß Namenskonflikte mit Objektreferenzen nicht auftreten:



let $z\#668 = 1$

$f\#127(z\#668) \;=\; \mathbf{fun}(y^{\#916} : Int\#333)\, z\#668$

Die korrekte Bindung von $y\#490$ an die deklarierende Signatur $y^{\#490} : Int$ bleibt auch nach der Funktionsapplikation bestehen. Ein zusätzliches Inkrementieren oder Dekrementieren von Werten ist nicht notwendig, da der Objektspeicher die Wahrung von Objektreferenzen garantiert.

5.2.2.2 Substitution mit Objektreferenzen

Die Substitutionsoperation $\langle B \to X^i \rangle A$ ist für alle möglichen Varianten der Typrepräsentation $TLAST.Type$ definiert. Die folgenden Regeln zeigen, in welchen Fällen die Substitution $\langle A \to X^i \rangle Y^j$ bei einfachen und $\langle A \to X^i \rangle Y^j.Z$ bei qualifizierten Typbezeichnern vollzogen wird:

$$
\begin{array}{lcll}
\langle A \rightarrow X^i \rangle Y^j & = & Y^j & , \; if \; i \neq j \\
\langle A \rightarrow X^i \rangle Y^j & = & A & , \; if \; i = j \\
\langle A \rightarrow X^i \rangle Y^j.Z & = & Y^j.Z & , \; if \; i \neq j \\
\langle A \rightarrow X^i \rangle Y^j.Z & = & ide(A).Z & , \; if \; i = j
\end{array}
$$

Der einfache Bezeichner Y^j wird durch A und der qualifizierte Bezeichner $Y^j.Z$ durch $ide\{A\}.Z$ ersetzt, falls die deklarierenden Signaturen von X^i und Y^j dieselbe Identität besitzen, d.h. falls $i = j$ gilt. Da es in TL wertabhängige Typen gibt, ist die Substitutionsoperation auch für Wertbezeichner definiert, die in Typpfaden auftreten:

$$
\begin{array}{lcll}
\langle a \rightarrow x^i \rangle y^j.Z & = & y^j.Z & , \; if \; i \neq j \\
\langle a \rightarrow x^i \rangle y^j.Z & = & ide\{a\}.Z & , \; if \; i = j
\end{array}
$$

Die Funktion $ide\{p\}$ verhindert während der Substitution die Konstruktion syntaktisch unerlaubter Typen wie **tuple Let** T = Int **end**.T oder **Tuple** T <: **Ok end**.T. Sie ist auf Werten und Typen folgendermaßen definiert:

$ide\{x\} = x$
$ide\{X\} = X$
$ide\{p.x\} = ide\{p\}.x$
$ide\{p.X\} = ide\{p\}.X$
$ide\{a\} = ?^i$ mit $?^i : A$, A ist der Typ von a und $i = newOid()$
$ide\{A\} = ?^i$ mit $?^i = A$ und $i = newOid()$

Ist das Argument A von *ide* kein Typbezeichner, so wird ein neuer immer einzigartiger Bezeichner ? mit der Signatur $? = A$ generiert. Wenn das Argument a kein Wertbezeichner ist, wird die Signatur $? : A$ generiert, wobei A der Typ von a ist. Die vollständigen Regeln für die Substitution stehen im Anhang H.

5.2.2.3 Substitutionsalgorithmus in TLMIN

Im Typprüfer von TLMIN ist der Substitutionsalgorithmus im Modul *tlSubstitute* implementiert. Zur Substitution von Typen dient die Funktion *substituteType* und zum Ersetzen in einer Signatur die Funktion *substituteSignature*:

```
Let Substitute = Tuple
                   formal :TLAST.Signature
                   actual :TLAST.Argument
                 end
  (* A substitution contains a 'formal' signature
     which has to be substituted for an 'actual'
     part which is either a type or a value identifier
     (possibly with path). *)

Let Substitutes = list.T(Substitute)

substituteType(type :TLAST.Type
```

```
                substitutes :Substitutes) :TLAST.Type
    (* Substitute 'type' according to 'substitutes'. *)

substituteSignature(signature :TLAST.Signature
                    substitutes :Substitutes) :TLAST.Signature
    (* Substitute 'signature' according to 'substitutes'. *)
```

Die Funktion *substituteType* wird mit dem zu substituierenden Typ *type* aufgerufen. *type* wird kopiert und Substitutionen gemäß einer Substitutionsliste *substitutes* durchgeführt. Die Liste enthält Paare $(formal, actual)$. $formal$ ist die Signatur des zu ersetzenden Typ– oder Wertbezeichners. *actual* bezeichnet den Syntaxbaum, der für *actual* eingesetzt werden soll. Die Funktion *substituteSignature* substituiert innerhalb einer Signatur *signature* gemäß der angegebenen Substitutionsliste.

5.2.3 Qualifizierte Typbezeichner

In TL werden strukturierte Werte und Typen mit den vordefinierten Wert- und Typkonstruktoren Tupel, Tupelvarianten und Record erzeugt (siehe Abschnitt 3.2). Dabei werden Bindungen und Signaturen aggregiert. Aufgrund der Orthogonalität von TL sind neben Wertbindungen und Wertsignaturen ebenfalls Typbindungen und Typsignaturen zulässig. In Abschnitt 3.6 wird gezeigt, daß bei der Typinferenz von aggregierten Werten wie $nat1.zero$ eine Qualifizierung der Typvariablen T mit $nat1$ notwendig ist, um die Abhängigkeit zu der Wertvariablen (*value dependent type*) sicherzustellen und damit die abstrakten Datentypen $nat1.T$ und $nat2.T$ unterscheiden zu können.

5.2.3.1 Repräsentation von qualifizierten Typbezeichnern

Qualifizierte Typbezeichner werden wie einfache Typbezeichner durch den $ideCase$ von $TLAST.Type$ repräsentiert (siehe Abschnitt 5.2.2.1). Bei qualifizierten Typbezeichnern ist die Pfadliste *path* vom Typ $Path$ nicht leer. Ein Pfadelement wird dabei durch den Typ $Field$ repräsentiert:

```
Let Field =
  Tuple
    name :String (* 'name' in tupCase only for debugging *)
  case tupCase with
    index :Int
      (* position of field in tuple type,
         debugging:
           do set printIndices true
           format: @index *)
  case rcdCase with
  end
```

Bei Pfadelementen, die auf Recordkomponenten verweisen, ist der Name *name* relevant, bei Pfadelementen, die Tupelkomponenten referenzieren, die Position *index* im Tupeltyp. Beispielsweise wird der Typ $nat1.T$ dargestellt mit einer Signatur $first$, die die deklarierende Signatur von $nat1$ repräsentiert, und einem Pfad *path*, dessen einziges Element eine *tupCase* Variante vom Typ $Field$ mit $index = 1$ ist, da der Bezeichner T als erstes Element im Tupeltyp Nat auftritt.

5.2.3.2 Qualifizierung von Typbezeichnern

Beim Feldzugriff auf Wert- oder Typkomponenten eines Tupels oder Records müssen alle lokal extrahierten Typinformationen durch einen Pfad qualifiziert werden. Das folgende Beispiel dient als Motivation zur Beschreibung des Qualifizierungsalgorithmus:

```
a :Tuple                                  1.) a.b   :Tuple
    T1 <:Ok                                           T2 <:a.T1
    b :Tuple                                          c :T2
        T2 <:T1     ⟹                              end
        c :T2                             2.) a.b.c :a.b.T2
    end                                   3.) a.d   :a.b.T2
    d :b.T2
end
```

Der Typ von $a.b$ wird bestimmt, indem der Tupeltyp von b kopiert und dabei jeder Bezeichner X, der im Tupeltyp von a deklariert ist, durch $a.X$ ersetzt wird, kurz:

$$\langle a.T1 \rightarrow T1 \rangle \textbf{Tuple}\ T2\ <: T1\ \ c\ \ : T2\ \textbf{end}$$

Die Substitution erfolgt dabei rekursiv über die Pfadlänge des Wertes. Der Typ von $a.b.c$ wird berechnet, indem der inferierte Typ von $a.b$ nochmals kopiert und dabei der Bezeichner $T2$ durch $a.b.T2$ substituiert wird. Bei der Substitution ist zu beachten, daß der zu ersetzende Bezeichner bereits qualifiziert sein kann. Im Beispiel besitzt d den qualifizierten Typ $b.T2$. Der Wert $a.d$ hat dann den zusätzlich mit a qualifizierten Typ $a.b.T2$.

Typbezeichner werden in den TLMINTypregeln *[Value tuple dot]*, *[Value record dot]* und *[Subtype Dot]* qualifiziert.

[Value tuple dot]

$$\frac{S \vdash a : \mathbf{Tup}(S', x : A, S''; C) \quad x \notin Dom(S'')}{S \vdash a.x : \mathrm{qualify}(A; a; S')}$$

[Value record dot]

$$\frac{S \vdash a : \mathbf{Rcd}(S', x : A, S'')}{S \vdash a.x : \mathrm{qualify}(A; a; S')}$$

[Subtype Dot]

$$\frac{S \vdash p \overset{Sig}{::} S', X <: A, S'' \quad X \notin Dom(S'')}{S \vdash p.X <: \mathrm{qualify}(A; p; S')}$$

Ein qualifizierter Typ A, in dem jedes Auftreten eines $X \in Dom(S)$ durch $p.X$ ersetzt ist, abgekürzt $\mathrm{qualify}(A; p; S)$, ist wie folgt definiert:

$$\begin{aligned}
\mathrm{qualify}(A; p; \varnothing) &= A \\
\mathrm{qualify}(A; p; S, x : B) &= \mathrm{qualify}(\langle p.x \rightarrow x \rangle A; p; S) \\
\mathrm{qualify}(T; p; S, X <: B) &= \mathrm{qualify}(\langle p.X \rightarrow X \rangle A; p; S) \\
\mathrm{qualify}(T; p; S, X = B) &= \mathrm{qualify}(\langle p.X \rightarrow X \rangle A; p; S)
\end{aligned}$$

5.2.3.3 Qualifizierungsalgorithmus in TLMIN

Die Funktion $qualifyType$ im Modul $tlQualify$ qualifiziert Typbezeichner im Typ einer Signatur, die durch die Funktionsargumente $first$, $prefix$ und $path$ referenziert wird:

```
qualifyType(first :TLAST.Signature  prefix, path :TLAST.Path
            type :TLAST.Type  state :State) :TLAST.Type

  (* Return qualified type in signature indicated by 'first', 'prefix'
     and 'path'. 'type' belongs to signature indicated by 'first' and 'prefix'. *)
```

Die Typqualifizierung geschieht rekursiv in Abhängigkeit von der Pfadlänge von $path$. Solange der Pfad nicht leer ist, werden Typen qualifiziert, die durch den Typ $type$ und das erste Pfadelement in $path$ referenziert werden. Dabei werden Typbezeichner gegebenenfalls mit $first$ und $prefix$ qualifiziert. Während sich die Pfadlänge von $path$ mit jedem Rekursionsschritt verkleinert, wächst die Pfadlänge im abgearbeiteten Pfad $prefix.\ prefix$ und $path$ bilden verknüft immer den gleichen Pfad.

Um beispielsweise den qualifizierten Typ von $list.cons$ zu berechnen, wird die Funktion $qualifyType$ mit folgenden Argumenten aufgerufen:

first bezeichnet die Wertsignatur $list : List$

prefix ist ein leerer Bezeichnerpfad

path enthält das Pfadelement, das die Tupelkomponente $cons$ beschreibt

type ist der Typbezeichner $List$

Die Funktion bestimmt die Signatur von $cons$ aus $type$ und $path$. Sie kopiert den Signaturtyp

```
Fun(A <:Ok  head :A  tail :T(A)) :T(A)
```

und ersetzt den Typbezeichner T durch $list.T$. Das Präfix $list$ in $list.T$ wird aus $first$ ($list$ Signatur) und $prefix$ (leerer Pfad) gebildet. Im zweiten Rekursionsschritt terminiert die Typqualifizierung, da der Pfad $path$ leer ist.

Die während eines Rekursionsschrittes ausgeführte Typqualifizierung entspricht den im letzten Abschnitt aufgestellten formalen Qualifizierungsregeln.

5.2.3.4 Alternative zu qualifizierten Tybbezeichnern

Die im folgende vorgestellte Alternative zu qualifizierten Typbezeichnern resultiert aus einer anfänglichen Typüberprüferimplementierung, in der Typbezeichner in der abstrakten Syntax ohne Pfadangabe repräsentiert werden.

Da der Pfad in der abstrakten Syntax entfällt, werden Typbezeichner ausschließlich durch die Identität der deklarierenden Typsignatur unterschieden, die durch den Typpfad in der konkreten Syntax referenziert wird. Beispielsweise haben $nat1.zero$ und $nat2.zero$ zwei gleichnamige Typen T, deren deklarierende Signaturen aufgrund der Typinkompatibilität verschiedene Identitäten besitzen müssen. Um unterschiedliche Instanzen von T zu erhalten, ist der Typ Nat bei der Deklaration von neuen Werten dieses Typs wiederholt zu kopieren. Damit entstehen zwei Probleme:

- Während die Typvariablen T von $nat1$, $nat2$ und nat unterschiedlich sind, sollten ihre Typen Nat gleich sein. So soll der Typprüfer sofort feststellen, daß der Aufruf $f(nat1)$ im folgenden Beispiel typkorrekt ist, weil die Typen des formalen und des aktuellen Parameters identisch sind.

  let *nat1 :Nat = ...*
  let *f(nat :Nat) = nat.zero*
  f(nat1) (typkorrekt, wegen Typidentität Nat == Nat *)*

  Bei unterschiedlichen Kopien von Nat ist jedoch ein Vergleich der gesamten Tupelstruktur erforderlich. Eine mögliche Lösung ist, sich für jeden Wert sowohl den Originaltyp als auch die Typkopie zu merken. Beim Zugriff auf den gesamten Typ Nat wird das Original herangezogen, beim Zugriff auf Komponenten die Kopie.

- Die Wertabhängigkeit ist auch für die korrekte Substitution von Bezeichnern entscheidend, z.B. bei der Berechnung des Rückgabetyps bei der Funktionsapplikation, wie der Aufruf $g(nat1)$ verdeutlicht:

  let *nat1 :Nat = ...*
  let *g(nat :Nat) :*T_{nat} *= nat.zero*
  g(nat1) :$\langle nat1 \rightarrow nat \rangle T_{nat}$

  Die Funktion g hat nicht den Rückgabetyp $nat.T$ sondern T_{nat}. Die Indizierung soll dabei die Wertabhängigkeit zu nat ausdrücken. Zur Berechnung des Typs von $g(nat1)$ muß der Rückgabetyp T_{nat} durch die Typkopie T_{nat1} von $nat1$ ersetzt werden. Dies ist ein Problem, da keine unmittelbare Abhängigkeit zwischen nat und der Typvariablen T_{nat} von nat besteht. Eine mögliche Lösung ist, im Rückgabetyp von g nicht nur den Formalparameter durch den Aktualparameter zu ersetzen, sondern zusätzlich alle in den Parametern auftretenden Typ- und Wertsignaturen. Der korrekte Rückgabetyp von $g(nat1)$ ist dann:

  $$\langle nat1 \rightarrow nat \rangle \langle T_{nat1} \rightarrow T_{nat} \rangle \langle zero_{nat1} \rightarrow zero_{nat} \rangle \langle succ_{nat1} \rightarrow succ_{nat} \rangle T_{nat} \;=\; T_{nat1}$$

Diese Alternative erweist sich im Vergleich zu qualifizierten Typbezeichnern als unpraktikabel: Das Kopieren der abstrakten Tupeltypen kostet Zeit und Speicherplatz. Noch aufwendiger ist die Erstellung der Substitutionsliste in der Substitutionsoperation.

5.2.4 Nichtabstrakte Typbezeichner

Während in $F_{\infty}^{\leq}$ nur die Definition abstrakter Typbezeichner möglich ist, können in TLMIN durch das Schlüsselwort **Let** Typausdrücke statisch an nichtabstrakte Typbezeichner gebunden werden. Jede Verwendung eines nichtabstrakten mit **Let** definierten Typbezeichners ist äquivalent zu dem gebundenen Typausdruck. Im folgenden Beispiel ist der Bezeichner T äquivalent zu dem Funktionstyp $\mathbf{Fun}(x : Int) : Bool$:

Let $T = \mathbf{Fun}(x : Int) : Bool$

Die Typregel *[Sig Ide Let]* führt einen neuen nichtabstrakten Bezeichner in den statischen Kontext ein:

[Sig Ide Let]

$$\frac{S \vdash A \ type \quad i = newOid()}{\vdash S, X^i = A \ sig}$$

Ein nichtabstrakter Typbezeichner kann referenziert werden, falls er im statischen Kontext definiert ist. Dies drückt die Regel *[Type Ide Let]* aus:

[Type Ide Let]

$$\frac{\vdash S, X = A, S' \ sig \quad X \notin Dom(S')}{S, X = A, S' \vdash X \ type}$$

Die in Abschnitt 4.4 für F_∞ aufgestellten Regeln für die '$\xrightarrow{\beta}$' Reduktion sind in TLMIN um eine zusätzliche kontextabhängige Regel für nichtabstrakte Typbezeichner zu erweitern:

$$\frac{S, X = A, S' \vdash A \xrightarrow{\beta} B}{S, X = A, S' \vdash X \xrightarrow{\beta} B}$$

Wenn A sich zu B reduzieren läßt, dann läßt sich auch X zu B reduzieren.

5.3 Subtypbeziehungen

5.3.1 Subtypisierung bei strukturierten Datentypen

Zu den strukturierten Datentypen in TLMIN gehören Tupel-, variante Tupel-, Record-, Ausnahme-, Funktions- und Typoperatortypen. Außer Typoperatoren sind alle Typen Subtypen von **Ok**. Ansonsten gilt die bereits in Abschnitt 3.2 besprochene strukturelle Kompatibilität. Für den Subtyptest bedeutet dies, daß weitere Subsignaturtest und/oder Subtyptests auf den Typkomponenten erforderlich sind. Bei varianten Tupeln kommt noch ein Subvariantentest hinzu.

So gilt beispielsweise für Funktionstypen die Kontravarianzregel:

[Subtype Fun]

$$\frac{S \vdash S'' <:: S' \quad S, S'' \vdash \langle S'' \Rightarrow S' \rangle A <: B}{S \vdash \mathbf{Fun}(S') : A <: \mathbf{Fun}(S'') : B}$$

Um die Subtypbeziehung von Funktionstypen zu prüfen, muß der Typprüfer einen Subsignaturtest und einen weiteren Subtyptest durchführen. Nachfolgend ist die Implementierung des Subtyptests von Funktionstypen in TLMIN dargestellt:

```
case big
when funCase with bigL then
  case small
  when funCase with smallL then
    if areSubSignatures(bigL.signatures smallL.signatures ...) then
      let substitutes = buildSubstitutes(smallL.signatures bigL.signatures)
```

```
        let smallRange = substituteType(smallL.range substitutes)
        isSubType(smallRange bigL.range ...)
    else
        false
    end
...
```

Falls der Subtyp *small* und der Supertyp *big* Funktionstypen ($funCase$) sind, wird zuerst der Subsignaturtest $areSubsignatures$ mit den Funktionssignaturen ausgeführt. Bei erfolgreichem Test wird die Kopie $smallRange$ des Rückgabetyps $smallL.range$ generiert, in der jedes Auftreten einer Signatur aus $smallL.signatures$ durch die entsprechende Signatur aus $bigL.signatures$ ersetzt ist. Als letztes wird die Subtypbeziehung $smallRange <: bigL.range$ geprüft. Der Subtyptest $isSubType$ und der Subsignaturtest $areSubSignatures$ befinden sich im Modul $tlCheck$.

5.3.2 Subtypisierung bei Typbezeichnern

5.3.2.1 Subtypregeln bei Typbezeichnern

Die Subtypisierung bei Typbezeichnern ist durch die Typregeln *[Subtype reflexive]*, *[Subtype transitive]* und *[Subtype beta]* festgelegt. Regel *[Subtype reflexive]* definiert eine Subtypbeziehung zwischen zwei gleichen Typen:

[Subtype reflexive]

$$\frac{S \vdash A \ type}{S \vdash A <: A}$$

Dies gilt insbesondere für abstrakte Typvariablen. Beispielsweise gilt

$$nat1.T \quad <: \quad nat1.T\,,$$

1. falls beide Bezeichner $nat1$ die identische Signatur $nat1 : Nat$ referenzieren und
2. falls beide Pfade T gleich sind.

Die Gleichheit von Pfaden ist durch dieselbe Position der referenzierten Tupelkomponenten bzw. durch Namensgleichheit bei Recordkomponenten definiert.

Bei verschiedenen Typbezeichnern existieren zwei weitere Möglichkeiten einer Subtypisierung. Wie bereits in Abschnitt 4.5 beschrieben, kann die nicht rückwärts anwendbare Typregel *[Subtype transitive]* durch die zwei rückwärts interpretierbaren Typregeln *[Subtype app alg]* und *[Subtype ide alg]* ersetzt werden. Letztere beschreibt eine Subtypbeziehung mit einem abstrakten Bezeichner als Subtyp. Im folgenden ist diese Regel in der sortenfreien Notation aufgeführt:

[Subtype ide alg]

$$\frac{S_1, X <: A, S_2 \vdash A <: B}{S_1, X <: A, S_2 \vdash X <: B}$$

Um die Subtypbeziehung $X <: B$ zu testen, wenn B ungleich X ist, wird mit dem Supertyp A von X anstelle von X weitergeprüft. Dabei ist zu beachten, daß X auch ein qualifizierter Typbezeichner sein kann. In diesem Fall sind eventuell in A auftretene Typbezeichner zu qualifizieren.

Die zweite Möglichkeit einer Subtypbeziehung bei unterschiedlichen Typbezeichnern betrifft den Subtyptest von nichtabstrakten Typbezeichnern, die mit dem Schlüsselwort **Let** definiert sind. Die Typregel *[Subtype beta]* definiert die Subtypbeziehung zwischen nichtabstrakten Typbezeichnern:

[Subtype beta]

$$\frac{S \vdash A <: B \qquad S \vdash B \stackrel{\beta}{=} C}{S \vdash A <: C}$$

Nichtabstrakte Typbezeichner können als Subtyp und als Supertyp in einer Subtypbeziehung auftreten. Die Subtypbeziehung von zwei gleichen nichtabstrakten Typbezeichnern kann bereits durch den Typgleichheitstest bewiesen werden. Bei Ungleichheit gibt es zwei Alternativen:

- Es wird anstelle des nichtabstrakten Typbezeichners mit dem durch ihn gebundenen Typausdruck weitergeprüft (einstufige Expansion).
- Nichtabstrakte Typbezeichner werden vor dem Subtyptest vollständig durch die dazu äquivalenten Typausdrücke ersetzt (vollständige Expansion). Wie in $F^{\leq}_{\infty}$ beschrieben, bleiben nur noch normalisierte Typausdrücke übrig.

In TLMIN wird der erste Weg beschritten, da durch rekursive Typen eine Normalisierung nicht möglich ist (unendliche Expansion).

5.3.2.2 Subtyptestalgorithmus von Typbezeichnern in TLMIN

Die im letzten Abschnitt beschriebene Dreiteilung der Typüberprüfung von Typbezeichnern findet sich im TLMIN Typprüfer wieder:

```
case big
when ideCase with bigL then
   case small
   when ideCase with smallL then
      if smallL.first == bigL.first andif
         tlAST.samePath(smallL.path bigL.path) then        (* Typgleichheit *)
         true
      else
         try
            isSubType(small stripIde(big true state) state)    (* big ist nichtabstrakt *)
         when exactError then
            isSubType(stripIde(small false state) big state)   (* small ist abstrakt oder *)
         end                                                   (* nichtabstrakt *)
      end
   else
      ...
```

Zwei Typbezeichner *small* und *big* sind gleich, wenn ihre Signaturen identisch ($smallL.first == bigL.first$) und ihre Pfade gleich sind ($samePath$). Wenn die Bezeichner verschieden sind, und *big* nichtabstrakt ist, wird mit dem zu *big* äquivalenten Typ weitergetestet ($stripIde(big\ true...)$). Wenn *big* abstrakt ist, wird die Ausnahme $exactError$ ausgelöst, und mit dem an *small* gebundenem Typausdruck ($stripIde(small\ false...)$) weitergeprüft. Der Typbezeichner *small* darf in diesem Fall abstrakt oder nichtabstrakt sein. Die booleschen Parameter *true* oder *false* beim Funktionsaufruf von *stripIde* geben an, ob der Typbezeichner nichtabstrakt sein muß.

Mit dem Übersetzerkommando *do set traceSubType true* werden die im TLMIN Typprüfer durchgeführten Subtyptests ausgegeben. Im folgenden Beispiel sind die Ausgaben dargestellt, die bei der Definition einer Funktion *f* auftreten. Daneben ist aufgeführt, welcher der drei möglichen Subtyptests im Fall von Typbezeichnern angewendet wird:

let *f(X <:nat1.T* ***Let*** *Y = X y :Y) = nat1.succ(y);*

## small: Y	*Y wird expandiert*
*## big : **Ok***	
#### small: X	*X wird expandiert*
*#### big : **Ok***	
###### small: nat1.T	*nat1.T wird expandiert*
*###### big : **Ok***	
*######## small: **Ok***	*Typgleichheit*
*######## big : **Ok***	
## small: Y	*Y wird expandiert*
## big : nat1.T	*nat1.T ist abstrakt und wird nicht expandiert*
#### small: X	*X wird expandiert*
#### big : nat1.T	*nat1.T ist abstrakt und wird nicht expandiert*
###### small: nat1.T	*Typgleichheit*
###### big : nat1.T	

Der erste Subtyptest $Y <:$ **Ok** prüft die Korrektheit der Wertsignatur $y : Y$ und endet mit dem Test **Ok** $<:$ **Ok**. Anschließend wird der Funktionsaufruf $nat1.succ(y)$ getestet. Dazu ist der Subtyptest $Y <: nat1.T$ notwendig, dem zwei weitere Subtyptests folgen. In allen drei Tests ist der Typ $nat1.T$ der Supertyp. Dieser abstrakte Typ wird nicht weiter expandiert. Im Algorithmus werden bei den zwei ersten Tests die Ausnahme $exactError$ ausgelöst. Danach wird jeweils mit dem Typausdruck weitergeprüft, der an den Subtypen gebunden ist.

5.3.3 Subtypisierung bei Typoperatorapplikationen

5.3.3.1 Subtypregeln bei Typoperatorapplikationen

Der Subtyptest mit Typoperatorapplikationen im Subtyp und/oder im Supertyp ist wie der Subtyptest bei Typbezeichnern dreigeteilt, da abstrakte und nichtabstrakte Bezeichner an Typoperatoren gebunden werden können, so daß bei der Typoperatorapplikation die gleiche Bezeichnerproblematik hinsichtlich Transitivität und Typäquivalenz auftritt.

Der erste Fall einer Subtypbeziehung mit Typoperatorapplikationen folgt aus der Regel *[Subtype Apply]*:

[Subtype Apply]

$$\frac{\begin{array}{c} S \vdash A <: \mathbf{Oper}(S')C' \\ S \vdash B <: \mathbf{Oper}(S')C'' \\ S, S' \vdash C' <: C'' \qquad S \vdash D :: S' \end{array}}{S \vdash A(D) <: B(D)}$$

Zwei Typoperatorapplikationen stehen in Subtypbeziehung, falls die Signaturen der Supertypoperatoren gleich sind und deren Rückgabetypen in Subtypbeziehung stehen. Diese Typregel enthält in der Prämisse eine schwächere Aussage (Subtypbeziehung der Rückgabetypen) als die Typregel *[Subtype reflexive]* (Gleichheit der Rückgabetypen). Die Typregel *[Subtype reflexive]* ist damit für den Subtyptest von Typoperatorapplikationen bedeutungslos.

Ist die Regel *[Subtype Apply]* nicht anwendbar, so kann der eventuell abstrakte Typoperator im Subtyp ausgeführt und mit dem Ergebnistyp $\langle D \Rightarrow S' \rangle B$ weitergetestet werden:

[Subtype Apply alg]

$$\frac{\begin{array}{c} S \vdash A <: \mathbf{Oper}(S')B \\ S \vdash D :: S' \qquad \langle D \Rightarrow S' \rangle B <: C \end{array}}{S \vdash A\,(D) <: C}$$

Diese Typregel ersetzt die Transitivitätsregel (siehe Abschnitt 4.5). Der dritte Subtyptest betrifft Typoperatoren, die an nichtabstrakte Bezeichner gebunden sind. Eine derartige Typoperatorapplikation ist typäquivalent zu dem Ergebnistyp der Applikation. Wie bei nichtabstrakten Typbezeichnern kann in diesem Fall mit den äquivalenten Typausdrücken weitergetestet werden. Wie dort auch ist die zuständige Typregel *[Subtype beta]*.

5.3.3.2 Subtyptestalgorithmus von Typoperatorapplikationen in TLMIN

Das folgende Programmfragment zeigt den TLMIN Subtyptest für zwei Typoperatorapplikationen *small* und *big*:

```
case big
when appCase with bigL then
   case small
   when appCase with smallL then                    (* first chance: *)
      if isSubType(smallL.oper bigL.oper ...) andif (* Subtyptest für Typoperatoren *)
         areSubSignatures(smallL.signatures
                          bigL.signatures ...) andif (* Gleichheitstest für Signaturen *)
         areSubSignatures(bigL.signatures
                          smallL.signatures ...) then true
      else
         try                                        (* second chance: *)
            isSubType(small applyOper(big true ...) ...)  (* Typoperator ist nicht abstrakt *)
         when exactError then
            isSubType(applyOper(small false ...) big ...) (* Typoperator ist abstrakt *)
         end                                        (* oder nichtabstrakt *)
      end
   ...
```

Zuerst wird die Subtypbeziehung der Typoperatoren und die Gleichheit der Signaturen (*first chance*) geprüft. Die Gleichheit wird durch zwei Subsignaturtests bestimmt. Wenn einer der drei Tests fehlschlägt und der Typoperator von *big* nichtabstrakt ist, wird dieser ausgeführt (*applyOper*(*big true*...)) und mit dem Ergebnistyp weitergeprüft. Wenn der Typoperator abstrakt ist, wird die Ausnahme *exactError* ausgelöst und mit dem Ergebnistyp der Typoperatorapplikation von *small* (*applyOper*(*small false*...)) weitergeprüft. Die Funktion *applyOper* berechnet den Ergebnistyp einer Typoperatorapplikation. Die booleschen Parameter *true* oder *false* beim Funktionsaufruf von *applyOper* geben an, ob der Typoperator nichtabstrakt sein muß.

In [Matthes 93] wird behauptet, daß die Laufzeit des TL Typüberprüfungsalgorithmus linear in der Größe des konstruierten Ableitungsbaums ist. Diese Behauptung gilt nur, wenn in TL wie in TLMIN beim Subtyptest von Typoperatorapplikationen ein Zurücksetzen des Algorithmus (*backtracking*) nicht erforderlich ist, wenn also die *first chance* zu *true* evaluiert. Daher kann die Typüberprüfung von Programmen, in denen zahlreiche Typoperatoranwendungen enthalten sind, überdurchschnittlich lange dauern, wie das folgende Beispiel verdeutlicht:

```
Let T1 = Oper()Ok;
Let T2 = Oper(X <:Ok)Ok;
Let A <:T1() = T2(Ok);

## small: Ok
## big   : Ok
## small: T2(Ok)
## big   : T1()
#### small: T2                          Beginn der first chance
#### big   : T1
###### small: T2
###### big   : Oper() Ok
######## small: Oper(X <:Ok) Ok
######## big   : Oper() Ok              first chance schlägt fehl ⇒ backtracking:
#### small: T2(Ok)                      T() wird ausgeführt
#### big   : Ok
###### small: Ok
###### big   : Ok
```

Der Subtyptest $T2(\mathbf{Ok}) <: T1()$ scheitert in der *first chance*. Es wird daraufhin mit dem Ergebnistyp von $T()$ weitergeprüft. Bei der Implementierung ist zu beachten, daß beim Scheitern in der *first chance* keine Typfehlermeldungen generiert werden dürfen.

Es stellt sich die Frage, ob der aufwendige Subtyptest vereinfacht werden kann, indem auf die *first chance* verzichtet wird und Typoperatorapplikationen, wenn möglich, sofort ausgeführt werden. Das geht jedoch nicht, wie das folgende Beispiel mit dem abstrakten Typoperator *Array* verdeutlicht:

```
Let X <:Array(Int) = Array(Int)
```

Die Gültigkeit der Subtypbeziehung $Array(Int) <: Array(Int)$ kann nur in der *first chance* bestimmt werden. Andernfalls scheitert der Subtyptest, weil mit **Ok** als Subtyp weitergetestet wird.

5.4 Subsignaturbeziehungen

5.4.1 Subsignaturregeln

Signaturen S heißen Subsignaturen von Signaturen S', kurz $S <:: S'$, wenn erstens die geordneten Folgen S und S' die gleiche Länge besitzen, und zweitens die Typen A_i in den Signaturen S Subtypen der Typen A'_i an gleicher Position in S' sind. So gilt beispielsweise:

$$name : String,\ age : Int <:: name : \mathbf{Ok},\ age : Int <:: name : \mathbf{Ok},\ age : \mathbf{Ok}$$

Die folgenden Subsignaturregeln formalisieren die Subsignaturbeziehung von TLMIN:

[Subsig reflexive]
$$\frac{\vdash S, S'\ sig}{S \vdash S' <:: S'}$$

[Subsig ide]
$$\frac{S \vdash A <: B \qquad S, x : A \vdash S' <:: \langle x \to y \rangle S''}{S \vdash x : A, S' <:: y : B, S''}$$

[Subsig Ide]
$$\frac{S \vdash A <: B \qquad S, X <: A \vdash S' <:: \langle X \to Y \rangle S''}{S \vdash X <: A, S' <:: Y <: B, S''}$$

[Subsig Ide Let]
$$\frac{S \vdash A <: B \qquad S, X = A \vdash S' <:: \langle X \to Y \rangle S''}{S \vdash X = A, S' <:: Y <: B, S''}$$

[Subsig Ide Let2]
$$\frac{S \vdash A \stackrel{\beta}{=} B \qquad S, X = A \vdash S' <:: \langle X \to Y \rangle S''}{S \vdash X = A, S' <:: Y = B, S''}$$

Gemäß Regel *[Subsig reflexive]* stehen zwei gleiche Signaturen S' in Subsignaturbeziehung. Dies gilt insbesondere für die leere Signatur $\emptyset$ (Induktionsanfang). Die weiteren vier Typregeln zeigen die Möglichkeiten auf, bestehenden Signaturlisten unter Beibehaltung der Subsignaturbeziehung zu erweitern (Induktionsschritt). Dabei ist jeweils in der Supersignaturliste der Bezeichner der ergänzten Supersignaturkomponente durch den Bezeichner der ergänzten Subsignaturkomponente zu ersetzen. Diese Regel sind besser zu verstehen, wenn sie rückwärts von den Konklusionen zu den Prämissen gelesen werden. Um die Subsignaturen auf Subsignaturbeziehung zu testen, wird zuerst überprüft, ob $A <: B$ bzw. $A \stackrel{\beta}{=} B$ gilt. Danach werden in einem erweiterten Kontext die Signaturreste geprüft. Das folgende Beispiel verdeutlicht die erforderliche Substitution:

$$T1 <: \mathbf{Ok},\ t1 : T1 <:: T2 <: \mathbf{Ok},\ t2 : T2$$

Der Subsignaturtest im TLMIN Typprüfer läuft wie folgt ab:

1. Prüfe $T1 <: \mathbf{Ok} <:: T2 <: \mathbf{Ok}$
2. Prüfe $\mathbf{Ok} <: \mathbf{Ok}$

3. Substituiere $\langle T1 \rightarrow T2 \rangle t2 : T2$ in der folgenden Supertypsignatur.
4. Prüfe $t1 : T1 <:: t2 : T1$
5. Prüfe $T1 <: T1$

Im folgenden sind die entsprechenden Typprüferausgaben dargestellt:

Let *X <:***Tuple** *T2 <:***Ok** *t2 :T2* ***end*** *=* ***Tuple*** *T1 <:***Ok** *t1 :T1* ***end****;*

small: ***Tuple*** *T1 <:***Ok** *t1 :T1* ***end***
big : ***Tuple*** *T2 <:***Ok** *t2 :T2* ***end***
small: ***Ok***
big : ***Ok***
small: T1
big : T1

In der Typregel *[Subsig Ide Let2]* steht die Prämisse $A \stackrel{\beta}{=} B$, während in den anderen Typregeln die schwächere Prämisse $A <: B$ ausreichend ist. Die strenge Typgleichheit ist jedoch notwendig, um Laufzeitfehler zu vermeiden. Das folgende Beispiel definiert eine Funktion f, die als Argument beim Aufruf einer Funktion g übergeben wird:

let *f(t :***Tuple** *x :Int* ***end****) = t.x*
let *g(***Let** *T =* ***Tuple end*** *f(t :T):Int) = f(***tuple end****)*
*g(***Let** *T =* ***Tuple*** *x :Int* ***end*** *f)* $\Rightarrow$ *Laufzeitfehler*

Die Funktion f erwartet als Parameter einen Tupelwert t und extrahiert dessen Komponente x. Die Funktion g erwartet einen Typparameter T und eine Funktion f. Das Argument von f muß vom Typ T sein. Im Rumpf von g wird f auf den Tupelwert **tuple end** angewendet. Die Funktionsapplikation von g in der dritten Beispielzeile bewirkt die Funktionsapplikation von f mit dem Tupelwert **tuple end**. Der Zugriff auf die fehlende Komponente im Tupelwert erzeugt einen Laufzeitfehler.

5.4.2 Subsignaturregeln mit de Bruijn–Indizes

Die Substitution $\langle X \rightarrow Y \rangle S''$ von Typbezeichnern beim Subsignaturtest ist nicht erforderlich. Dabei wird ausgenutzt, daß Syntaxbaum und statischer Kontext zusammen einen Typausdruck repräsentieren. Der Syntaxbaum bleibt unverändert. Nur der Kontext wird manipuliert. Um beispielsweise $S \vdash T1 <: \mathbf{Ok}, t1 : T1@1 <:: T2 <: \mathbf{Ok}, t2 : T2@1$ zu beweisen, sind die folgenden Einzelschritte durchzuführen:

1. Prüfe $S \vdash T1 <: \mathbf{Ok} <:: T2 <: \mathbf{Ok}$
2. Prüfe $S \vdash \mathbf{Ok} <: \mathbf{Ok}$
3. Prüfe $S, T1 <: \mathbf{Ok} \vdash t1 : T1@1 <:: t2 : T2@1$
4. Prüfe $S, T1 <: \mathbf{Ok} \vdash T1@1 <: T2@1$

Der Subtyptest $T1@1 <: T2@1$ ist erfolgreich, weil die de Bruijn–Indizes von $T1@1$ und $T2@1$ gleich sind. Beide Typen referenzieren die Signatur $T1 <: \mathbf{Ok}$, die als oberstes Element im statischen Kontext steht.

5.4.3 Typregeln für Tupel-, Record- und Ausnahmesignaturen

Die Typregeln *[Subtype Tup]*, *[Subtype Rcd]* und *[Subtype Exc]*, die die Subtypbeziehung von Tupel-, Record- und Ausnahmetypen definieren, fordern in ihren Prämissen schwächere Subsignaturbedingungen, als sie für Funktions- oder Typoperatortypen festgelegt sind.

Für Tupel- und Ausnahmetypen gilt die Tupelsubsignaturbeziehung '$\overset{Tup}{<::}$', die durch die Typregel *[Subsig Tup]* definiert ist:

[Subsig Tup]

$$\frac{\vdash S, S_1, S_2 \ sig \qquad S \vdash S_1 <:: S_1'}{S \vdash S_1, S_2 \overset{Tup}{<::} S_1'}$$

Die Signatur S_1, S_2 ist eine Tupelsubsignatur von S_1', falls es ein Präfix S_1 gibt, das Subsignatur von S_1' ist und falls alle Signaturen wohlgeformt sind. So gilt

$$age : Int, speed : Int, fuel : String \overset{Tup}{<::} age : Int, speed : Int$$

für das Beispiel aus Abschnitt 3.2. Im Typüberprüfungsalgorithmus unterscheidet sich der Tupelsubsignaturtest vom einfachen Subsignaturtest nur durch den fehlenden Längenvergleich.

Die Typregeln *[Subsig Rcd empty]* und *[Subsig Rcd]* definieren die Recordsubsignaturbeziehung '$\overset{Rcd}{<::}$':

[Subsig Rcd empty]

$$\frac{S \vdash S' \ rcdsig}{S \vdash S' \overset{Rcd}{<::} \oslash}$$

[Subsig Rcd]

$$\frac{S \vdash S_1, S_2, S_3 \ rcdsig \qquad S \vdash S_1, S_2, S_3 \overset{Rcd}{<::} S_1' \qquad S, S_1' \vdash S_2 \overset{Tup}{<::} S_2'}{S \vdash S_1, S_2, S_3 \overset{Rcd}{<::} S_1', S_2'}$$

Eine Signatur S' ist Recordsubsignatur der leeren Signatur $\oslash$ (Induktionsanfang). Diese Beziehung bleibt bestehen, falls in der Supersignatur eine Signatur S_2' angefügt wird, die eine Tupelsupersignatur der Infixsignatur S_2 ist (Induktionsschritt). Für das Beispiel

$$S \vdash age : Int, fuel : String, speed : Int \overset{Rcd}{<::} age : Int, speed : Int$$

sind folgende Teilableitungen auszuführen:

$$S \vdash age : Int, fuel : String, speed : Int \overset{Rcd}{<::} \oslash$$
$$S \vdash age : Int, fuel : String, speed : Int \overset{Rcd}{<::} age : Int$$
$$S \vdash age : Int, fuel : String, speed : Int \overset{Rcd}{<::} age : Int, speed : Int$$

Die Typregel *[Subsig Rcd]* gilt für Recordsignaturen, deren einzelne Elemente voneinander unabhängig sind und deshalb beliebig vertauscht werden dürfen, ohne daß sich ihre Bedeutung verändert. Die Recordsubsignaturbeziehung kann jedoch von der Reihenfolge der Signaturelemente abhängig sein, wie das folgende Beispiel zeigt:

$$S, T <: Int \vdash T <: \mathbf{Ok}, \quad t : T \overset{Rcd}{<::} t : T, \quad T <: \mathbf{Ok}$$

Der Typ T in der Wertsignatur $t : T$ referenziert in der Subsignaturliste die Signatur $T <: \mathbf{Ok}$ und in der Supersignaturliste die Signatur $T <: Int$ im statischen Kontext. Die Signaturen stehen daher nicht in Recordsubsignaturbeziehung. Im folgenden Beispiel stehen die Recordsignaturen in Subsignaturbeziehung, da sie voneinander unabhängig sind:

$$a : Int, b : String \overset{Rcd}{<::} b : String, a : Int$$

Anders als in TLMIN gilt im Typprüfer von TL diese Beziehung nicht, weil dort die Reihenfolge von Subsignaturliste und Supersignaturliste übereinstimmen muß. In der Subsignaturliste dürfen nur zusätzliche Signaturen eingefügt sein. In TL gilt daher die folgende Typregel:

[Subsig Rcd tl0]

$$\frac{S \vdash S_1, S_2, S_3 \ \mathit{rcdsig} \qquad S \vdash S_1 \overset{Rcd}{<::} S_1' \qquad S, S_1' \vdash S_2 \overset{Tup}{<::} S_2'}{S \vdash S_1, S_2, S_3 \overset{Rcd}{<::} S_1', S_2'}$$

Sie unterscheidet sich von der TLMIN Typregel nur in der zweiten Prämisse, wo bereits mit S_1 als Subsignatur eine Recordsubsignaturbeziehung bestehen muß, so daß eine Permutation der Subsignaturliste ausgeschlossen ist.

5.5 Substitution in Rückgabetypen

5.5.1 Substitutionsoperation

Die Berechnung des Rückgabetyps beim Aufruf einer polymorphen Funktion oder eines Typoperators erfolgt in Abhängigkeit von den Argumenten. Im Rückgabetyp ist jedes Auftreten eines Formalparameters durch den zugehörigen Aktualparameter zu ersetzen. Die Substitutionsoperation $\langle D \Rightarrow S \rangle A$ in den Typregeln *[Value apply]* und *[Subtype App alg]* sowie in der Betareduktion ersetzt im Rückgabetyp A freie Variablen X^i bzw. $x^i \in Dom(S)$ durch Typen oder Wertbezeichner aus D:

[Value apply]

$$\frac{S \vdash a : \mathbf{Fun}(S') : A \qquad S \vdash D :: S'}{S \vdash a(D) : \langle D \Rightarrow S' \rangle A}$$

[Subtype Apply alg]

$$\frac{\begin{array}{c} S \vdash A <: \mathbf{Oper}(S')B \\ S \vdash D :: S' \qquad \langle D \Rightarrow S' \rangle B <: C \end{array}}{S \vdash A\,(D) <: C}$$

$$\frac{}{(\mathbf{Oper}(S)A)\,(D) \xrightarrow{\beta} \langle D \Rightarrow S \rangle A} \ (\beta)$$

Beispielsweise gelten für die in Abschnitt 3.4 definierte Funktion *poly* und für den in Abschnitt 3.5 definierten Typoperator Op die folgenden Typaussagen:

$poly(:Int\ 1)\ :\langle ? = Int, ? = 1 \Rightarrow A <: \mathbf{Ok}, a : A\rangle A\ =\ Int$

$Op(Int)\ =\ \langle ? = Int \Rightarrow A <: \mathbf{Ok}\rangle A\ =\ Int$

Die Substitutionsoperation ist für jede Bindungsvariante definiert:

$$\langle \varnothing \Rightarrow \varnothing\rangle A = A$$
$$\langle y = b, D \Rightarrow x^i : B, S\rangle A = \langle D \Rightarrow S\rangle\langle b \rightarrow x^i\rangle A$$
$$\langle y : B = b, D \Rightarrow x^i : B, S\rangle A = \langle D \Rightarrow S\rangle\langle b \rightarrow x^i\rangle A$$
$$\langle Y = B, D \Rightarrow X^i = B, S\rangle A = \langle D \Rightarrow S\rangle\langle B \rightarrow X^i\rangle A$$
$$\langle Y <: C = B, D \Rightarrow X^i = B, S\rangle A = \langle D \Rightarrow S\rangle\langle B \rightarrow X^i\rangle A$$

So gilt beispielsweise:

$\langle ? = Int, ? = 1 \Rightarrow A <: \mathbf{Ok}, a : A\rangle A\ =\ \langle ? \rightarrow a\rangle\langle Int \rightarrow A\rangle A\ =\ Int$

$\langle ? = Int \Rightarrow A <: \mathbf{Ok}\rangle A\ =\ \langle Int \rightarrow A\rangle A\ =\ Int$

Die Funktion *buildAppSubstitutes* im Modul *tlSubstitute* des TLMIN Typprüfers erstellt aus den Bindungen und den Formalsignaturen die Substitutionsliste gemäß der aufgeführten Regeln. Danach wird die Funktion *substituteType* mit der Substitutionsliste und dem formalen Rückgabetyp als Argument aufgerufen, die den Typ kopiert und die Ersetzung ausführt.

5.5.2 Nichtanonyme Funktionsargumente

In der abstrakten Syntax von TLMIN und TL werden Funktionsargumente als Wert– bzw. Typbindungen repräsentiert. Die konkrete Syntax erlaubt den Funktionsaufruf mit Wert– bzw. Typargumenten. Im Parser werden in diesem Fall mit den Wert– und Typargumenten anonyme Bindungen generiert (vgl. Abschnitt 5.1.3). Ein Funktionsaufruf mit nichtanonymen Argumentbindungen ist ebenfalls möglich. Beispiel:

let poly(X <:Ok x :X) :X = x
poly(:Int 1) normalisiert zu id(Let ? = Int let ? = 1) ⇒ 1 :Int
poly(Let X = Int let x :X = 1) ⇒ 1 :Int

Die polymorphe Identitätsfunktion *poly* wird in der zweiten Beispielzeile mit den Argumenten *Int* und 1 aufgerufen. Die Syntaxanalyse formt die Argumente in anonyme Bindungen um, die den Bezeichner "?" an die übergebenen Funktionsargumente bindet. In der dritten Beispielzeile ist eine Funktionsapplikation mit nichtanonymen Funktionsargumenten dargestellt.

Die in den Argumentbindungen lokal definierten anonymen oder nichtanonymen Bezeichner werden in den Substitutionsregeln ignoriert, um ein Auftreten im Ergebnistyp zu vermeiden. So besitzt der Funktionsaufruf von *poly* in der dritten Beispielzeile den Typ *Int* und nicht *X*. Dennoch ist es möglich, daß lokal definierte Bezeichner im Rückgabetyp auftreten können, wie das folgende Beispiel zeigt:

let *f(A <:**Ok*** *B <:**Ok*** *b :B) :B = b;*
*f(**Let** X = Int* ***Let*** *Y = X 1)* $\Rightarrow$ *1 :X*

Das Ergebnis des Funktionsaufrufs hat den lokal definierten Typ X. Da X als Int definiert ist, mag dieses Beispiel noch nicht gravierend sein. Im folgenden Beispiel wird jedoch ein falscher Ergebnistyp berechnet:

let *f(A <:**Ok*** a :A) = (***fun***(***Let*** B = A* ***Let*** C = A a :A) :C a)(**Let** X = A :X a)*
*f :**Fun**(A <:**Ok*** a :A) :X*
f(:Int 3) $\Rightarrow$ *3 :X*

Die Funktion f hat den formalen Rückgabetyp X, der auf den formalen Argumenttyp A zeigt. Beim Funktionsaufruf $f(:Int\ 3)$ wird der Rückgabetyp wie folgt berechnet:

$$f(:Int\ 3) :\langle ? = Int, ? = 3 \Rightarrow A <: \mathbf{Ok}, a : A\rangle X \;=\; \langle ? \rightarrow a\rangle\langle Int \rightarrow A\rangle X \;=\; X$$

X wird nicht substituiert und referenziert weiterhin den formalen abstrakten Argumenttyp A. Es gibt mehrere Möglichkeiten, dieses Problem zu lösen:

1. Im Funktionsaufruf sind nur anonyme Bindungen zulässig, d.h. explizite Typ– oder Wertbindungen sind nicht erlaubt. Beispiel:

f(:Int :Int 1)	*ist erlaubt*
*f(**Let** X = Int* ***Let*** *Y = Int* ***let*** *x = 1)*	*ist verboten*

2. Im Funktionsaufruf dürfen nichtanonyme Bindungen definiert aber nicht referenziert werden. Beispiel:

*f(**Let** X = Int* ***Let*** *Y = Int* ***let*** *x = 1)*	*ist erlaubt*
*f(**Let** X = Int* ***Let*** *Y <: X = Int 1)*	*ist verboten*

3. Im Funktionsaufruf dürfen nichtanonyme Bindungen definiert und referenziert werden. Die definierten Bezeichner dürfen nicht im Rückgabetyp erscheinen. Beispiel:

*f(**Let** X = Int* ***Let*** *Y <:X = Int 1)*	*ist erlaubt*
*f(**Let** X = Int* ***Let*** *Y = X 1)*	*ist verboten, da X Ergebnistyp ist*

Im Typüberprüfungsalgorithmus von TLMIN wurde die letzte Alternative gewählt, die der Programmierung den größten Freiraum gewährt. Im TL Typprüfer wird die Problematik der nichtanonymen Bindungen in Funktionsaufrufen nicht behandelt. Dort kommt es zum Systemabsturz.

5.6 Typdefinitionen in Programmblöcken

Der letzte Abschnitt hat gezeigt, daß beim Funktionsaufruf mit nichtanonymen Funktionsargumenten die durch die Argumentbindungen definierten Bezeichner nicht im Rückgabetyp auftreten dürfen. Neben Bezeichnern, die in einer Argumentliste definiert sind, können Bezeichner aus einem **begin end** Konstrukt den sie umgebenen Programmblock verlassen, wie das folgende Beispiel zeigt:

```
begin
   Let T = Int
   let t :T = 1
end
⇒  t :T
```

Der Wert einer **begin end** Sequenz entspricht dem Wert der letzten Wertbindung innerhalb dieser Sequenz. Im Beispiel ist dies der Wert t vom Typ T. Das Verlassen des Programmblocks führt zu einen fehlerhaften Berechnung des Rückgabetyps bei der Funktionsapplikation:

```
let f(T <:Ok  t :T) = begin
   Let Local = T
   let local :Local = t
end
f            ⇒    f : Fun(T  <: Ok  t  : T)  : Local
f(:Int 1)    ⇒    local : Local
```

Da eine explizite Rückgabetypangabe bei der Funktionsdefinition von f fehlt, wird der inferierte Typ *Local* des Funktionsrumpfes als Rückgabetyp von f gesetzt. Bei der Funktionsapplikation von f wird *Local* nicht substituiert, so daß im Typprüfer ein fehlerhafter Rückgabetyp berechnet wird.

Dieser Fehler tritt nicht auf, falls zur Rückgabetypberechnung der gesamte Funktionsrumpf ausgewertet wird. Bei der Auswertung des Rumpfes wird in diesem Fall *Local* mit dem Funktionsargument *Int* gleichgesetzt. Die Rückgabetypspezifikation bei der Funktionsdefinition ist als Optimierung zu verstehen, um nicht bei jeder Funktionsapplikation den gesamten Funktionsrumpf auszuwerten. Die Optimierung ist jedoch nur korrekt, wenn lokale Bezeichner den Funktionsrumpf nicht verlassen.

Im Typüberprüfungsalgorithmus testet die Funktion *leaveScope* aus dem Modul *tlCheckUtil*, ob ein Bezeichner einen Programmblock verläßt. Im folgenden ist die Signatur von *leaveScope* dargestellt:

```
leaveScope(type :TLAST.Type
        signatures :TLAST.Signatures
        errLog :errorLog.T
        typeCheck :TLSwitches.TypeCheck) :Bool
```

Die Funktion wird mit vier Argumenten aufgerufen. Die Funktion inspiziert den Typsyntaxbaum *type*. Wird ein in *type* auftretender Bezeichner gefunden, der eine Signatur aus den Signaturen *signatures* des Programmblocks referenziert, erfolgt ein Fehlereintrag in die Fehlerliste *errLog* des Typprüfers. Der Parameter *typeCheck* enthält eine Menge von Typprüferschaltern.

5.7 Rekursive Typen

5.7.1 Syntaxbaumrepräsentation

Rekursive Typen werden in TLMIN durch Bezeichner repräsentiert, die zyklische Signatursyntaxbäume referenzieren. Anders als in TL existiert keine Variante für rekursive Typen in der Typsyntaxbaumrepräsentation. Im folgenden Beispiel ist der für die Rekursion relevante Teil aus der Syntaxbaumrepräsentation von Signaturen dargestellt:

```
Let Rec Signature <: Ok =
   Tuple
      ...
      var type : Type
      recIndex : Int
      ...
end
```

Der Typ *Signature* ist als nichtabstrakter Typ in der Schnittstelle *TLAST* definiert. Die Variable *recIndex* vom Typ *Integer*, die in jeder Signaturvariante definiert ist, zeigt an, ob eine rekursive Signatur vorliegt. Der Wert dieser Variablen ist

- = 0, falls die Signatur nichtrekursiv ist,
- = 1, für die erste Signatur in der wechselseitig rekursiven Signaturliste,
- = n, für die n–te rekursive Signatur.

Ein boolscher Wert zur Unterscheidung zwischen rekursiven und nicht–rekursiven Signaturen ist unzureichend, da die erste Signatur in einer Liste wechselseitig rekursiver Signaturen identifizierbar sein muß, wie folgendes Beispiel zeigt:

```
Let Rec X1 <:Ok = F(X1)    ⇒  recIndex = 1
Let Rec Y1 <:Ok = G(Y1)    ⇒  recIndex = 1

Let Rec X2 <:Ok = F(X2)    ⇒  recIndex = 1
and Y2 <:Ok = G(Y2)        ⇒  recIndex = 2
```

Der Wert von *recIndex* in zwei aufeinanderfolgenden unabhängigen rekursiven Signaturen ist jeweils 1, während *recIndex* in zwei wechselseitig rekursiven Signaturen den Wert 1 und 2 besitzt. Eine dreiwertige Variable ist jedoch zur eindeutigen Identifizierung bereits ausreichend.

Um einen zyklischen Syntaxbaum für Signaturen zu konstruieren, ist in *Signature* die Komponente *type*, die den Typ in der Signatur repräsentiert, als veränderlich deklariert. Der Wert von *type* ist anfangs mit einem Initialwert (Platzhalter, *placeholder*) belegt, der später mit dem *tatsächlichen* Wert überschrieben wird. Der tatsächliche Wert seinerseits kann den Syntaxbaum der Signatur mit dem Platzhalter referenzieren. Durch die destruktive Zuweisung wird der Platzhalter überschrieben und der Zyklus geschlossen. Dieses Verfahren zur Bildung von zyklischen Strukturen wird im Typprüfer verwendet bei der

- Überprüfung der Wohlgeformtheit rekursiver Signaturen bzw. Typbindungen und der
- Substitution in rekursiven Signaturen bzw. Typbindungen.

Abbildung 5.1 verdeutlicht die Verzeigerung des referenzierenden Auftretens von Typbezeichnern mit ihren deklarierenden zyklischen Signaturen am Beispiel des Listenkonstruktors.

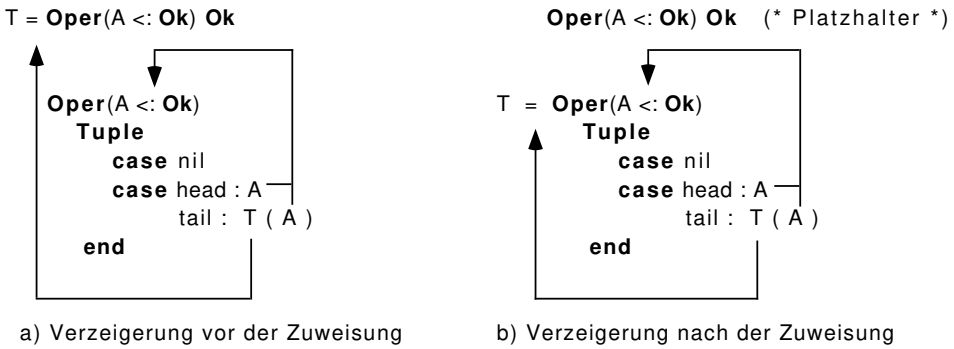


Abbildung 5.1: Verzeigerung im abstrakten Syntaxbaum des Listenkonstruktors

In Abbildung a) ist die Verzeigerung vor der destruktiven Zuweisung dargestellt. Der Bezeichner T ist an den Platzhalter **Oper**(A <:**Ok**) **Ok** gebunden, der in der konkreten Syntax von TLMIN bei der Definition rekursiver Typen immer angegeben werden muß. Der eigentliche Typoperatorausdruck, der an T gebunden werden soll, enthält eine Referenz auf die Signatur von T. Mit der destruktiven Zuweisung an die veränderliche Typsignatur von T entsteht ein Zyklus, wie er in Abbildung b) zu sehen ist.

5.7.2 Wohlgeformtheit wechselseitig rekursiver Typdefinitionen

Beim Test auf Wohlgeformtheit von wechselseitig rekursiven Typdefinitionen ist zu beachten, daß alle Typbezeichner parallel definiert werden und mit der ersten Typdefinition gültig sind. Der Bezeichner *Group* im folgenden Beispiel kann also bereits in der Typdefinition von *Manager* benutzt werden:

```
Let Rec Manager <:Ok = Tuple
    name :String  group :Group
end
and Group <:Ok = Tuple
    name :String  manager :Manager
end
```

Im Typüberprüfungsalgorithmus von TLMIN wird die Wohlgeformtheit der beiden Typdefinitionen wie folgt bestimmt:

1. Es wird die Wohlgeformtheit der Supertypen (**Ok**, **Ok**) von *Manager* und *Group* im bestehenden Kontext S überprüft:

$S \vdash \textbf{Ok}\ type$
$S \vdash \textbf{Ok}\ type$

2. Der bestehende statische Kontext wird mit den Signaturen $Manager = \textbf{Ok}$ und $Group = \textbf{Ok}$ erweitert. Als Platzhaltertypen werden die Supertypen (**Ok**, **Ok**) gesetzt. Anschließend wird im erweiterten Kontext S' die Wohlgeformtheit der Tupeltypen geprüft:

$S' \vdash \textbf{Tuple}\ name\ : String\ \ group\ : Group\ \textbf{end}\ type$
$S' \vdash \textbf{Tuple}\ name\ : String\ \ manager\ : Manager\ \textbf{end}\ type$
$mit\ S' = S, Manager{=}\textbf{Ok}, Group{=}\textbf{Ok}$

3. Die Platzhaltertypen in den Syntaxbäumen der Signaturen werden destruktiv durch die Tupeltypen überschrieben. Die Identität der Signatursyntaxbäume bleibt unverändert, so daß zyklische Signaturen entstehen.

4. Im erweiterten statischen Kontext wird die Kontraktivität der Tupeltypen von $Manager$ und $Group$ getestet:

$S' \vdash \textbf{Tuple}\ name\ : String\ \ group\ : Group\ \textbf{end} \downarrow Manager$
$S' \vdash \textbf{Tuple}\ name\ : String\ \ manager\ : Manager\ \textbf{end} \downarrow Group$
$mit\ S' = S, Manager \overset{Rec}{<:} \textbf{Tuple...}, Group \overset{Rec}{<:} \textbf{Tuple...}$

5. Als letztes wird die Subtypbeziehung von $Manager$ und $Group$ mit ihren Supertypen überprüft:

$S, Manager \overset{Rec}{<:} \textbf{Tuple...}, Group \overset{Rec}{<:} \textbf{Tuple...} \vdash Manager <: \textbf{Ok}$
$S, Manager \overset{Rec}{<:} \textbf{Tuple...}, Group \overset{Rec}{<:} \textbf{Tuple...} \vdash Group <: \textbf{Ok}$

Die Typregeln *[Sig Rec]* und *[Sig Rec Let]* definieren die Wohlgeformtheit für zyklische Typsignaturen, die abstrakte bzw. nichtabstrakte Bezeichner an Typausdrücke binden:

[Sig Rec]

$$\frac{\begin{array}{c} S \vdash B_i\ type \\ S, X_1 <: B_1, \ldots, X_n <: B_n \vdash A_i\ type \\ S, X_1 \overset{Rec}{<:} A_1, \ldots, X_n \overset{Rec}{<:} A_n \vdash A_i \downarrow X_i \\ S, X_1 \overset{Rec}{<:} A_1, \ldots, X_n \overset{Rec}{<:} A_n \vdash A_i <: B_i \\ i = 1 \ldots n \quad 1 \leq n \end{array}}{\vdash S, X_1 \overset{Rec}{<:} A_1, \ldots, X_n \overset{Rec}{<:} A_n\ sig}$$

[Sig Rec Let]

$$\frac{\begin{array}{c} S \vdash B_i\ type \\ S, X_1 = B_1, \ldots, X_n = B_n \vdash A_i\ type \\ S, X_1 \overset{Rec}{=} A_1, \ldots, X_n \overset{Rec}{=} A_n \vdash A_i \downarrow X_i \\ S, X_1 \overset{Rec}{=} A_1, \ldots, X_n \overset{Rec}{=} A_n \vdash A_i <: B_i \\ i = 1 \ldots n \quad 1 \leq n \end{array}}{\vdash S, X_1 \overset{Rec}{=} A_1, \ldots, X_n \overset{Rec}{=} A_n\ sig}$$

Die erste Prämisse sichert die Wohlgeformtheit der Supertypen B_i zu. In der zweiten Prämisse wird die Wohlgeformtheit der Typausdrücke A_i in dem mit X_i Signaturen erweiterten Kontext getestet. Im Kontext der dritten Prämisse werden die Typen in den Signaturen ersetzt, angedeutet durch das Wort *Rec*, und damit der Kontrativitätstest ausgeführt, der im folgenden Abschnitt vorgestellt wird. In der letzten Prämisse wird die Subtypbeziehung von A_i und B_i für alle i geprüft.

5.7.3 Kontraktive Typen

Die TLMIN Typregeln *[Sig Rec]* und *[Sig Rec Let]* enthalten die Aussage $S \vdash A \downarrow X$, um sicherzustellen, daß nur kontraktive Typen A an Bezeichner X gebunden werden. Ein Typ A ist nicht–kontraktiv, wenn er aufgrund seiner Trivialität keine eindeutige Spezifikation liefert. Das folgende Beispiel zeigt unerlaubte Typdefinitionen mit nicht–kontraktiven Typen:

Let Rec $X <:$***Ok*** $=X$
Let Rec $F(X <:$***Ok***$) <:$***Ok*** $= G(X)$ ***and*** $G(X <:$***Ok***$) <:$***Ok*** $= F(X)$
Let Rec $A <:$***Ok*** $= F(A)$ ***and*** $F(X <:$***Ok***$) <:$***Ok*** $= X$

Nicht–kontraktive Typen sind im Typprüfer von TLMIN statisch verboten, da sie dem Super- und Subtyp aller Typen und Typoperatoren entsprechen und damit die elementare Bedingung verletzen, daß eine korrekte Typisierung keine Laufzeitfehler produzieren kann (Korrektheit der Typisierung bezüglich der Evaluationssemantik, *soundness*). Das folgende Beispiel zeigt, was passieren kann:

Let Rec $X <:$***Ok*** $= X$
let $x : X =$ *"hallo"* *(* ok *)*
$1 + x$ $\Rightarrow$ *Laufzeitfehler*

5.7.3.1 Typregeln

Formal gelten gemäß [Matthes 93] in jedem wohlgeformten Kontext S die folgenden Kontraktivitätsaussagen für einen Typbezeichner X^i mit Signaturidentität i:

$Y^j \downarrow X^i \Leftrightarrow i \neq j$ $\quad$ $x^j.Z \downarrow X^i$ $\quad$ $\mathbf{Ok} \downarrow X^i$ $\quad$ $\mathbf{Nok} \downarrow X^i$ $\quad$ $\mathbf{Var}(A) \downarrow X^i$
$\mathbf{Tup}(S';C) \downarrow X^i$ $\quad$ $\mathbf{Rcd}(S') \downarrow X^i$ $\quad$ $\mathbf{Exc}(S') \downarrow X^i$ $\quad$ $\mathbf{Fun}(S') : B \downarrow X^i$

Die restlichen zwei Typregeln beziehen sich auf ihren statischen Kontext S:

[Contractive Oper]

$$\frac{S, S' \vdash B \downarrow X^i}{S \vdash \mathbf{Oper}(S')B \downarrow X^i}$$

[Contractive Apply]

$$\frac{S \vdash A <: \mathbf{Oper}(S')B \quad S \vdash D :: S' \quad S \vdash \langle D \Rightarrow S' \rangle B \downarrow X^i}{S \vdash A(D) \downarrow X^i}$$

Die folgenden Beispiele zeigen jedoch, das die Typregeln unvollständig und zum Teil fehlerhaft sind.

1. **Regel:** $Y^j \downarrow X^i \Leftrightarrow i \neq j$
   Beispiel:

Let Rec X ***<:Ok*** $= Y$ ***and*** Y ***<:Ok*** $= X$

Der Bezeichner Y ist nicht–kontraktiv in X und der Bezeichner X ist nicht–kontraktiv in Y, im Widerspruch zur Typregel. Die Regel ist durch die folgenden Typregeln zu ersetzen:

[Contractive Ide]

$$\frac{S, Y^j <: A, S' \vdash A \downarrow X^i \quad i \neq j}{S, Y^j <: A, S' \vdash Y^j \downarrow X^i}$$

[Contractive Ide Let]

$$\frac{S, Y^j = A, S' \vdash A \downarrow X^i \quad i \neq j}{S, Y^j = A, S' \vdash Y^j \downarrow X^i}$$

[Contractive Dot]

$$\frac{S \vdash Y^j \downarrow X^i \quad i \neq j}{S \vdash Y^j.T \downarrow X^i}$$

Bezeichner Y bzw. $Y.T$ sind kontraktiv in X, falls die an ihnen gebundenen Typausdrücke A kontraktiv in X sind.

Um im Beispiel $Y \downarrow X$ zu beweisen, ist Regel *[Contractive Ide Let]* rückwärts anzuwenden und $X \downarrow X$ zu beweisen. Diese Aussage ist im Typsystem jedoch nicht ableitbar. Ein Typbezeichner X ist niemals kontraktiv in sich selbst.

2. **Regel: Var**$(A) \downarrow X^i$
   Beispiel:

   Let Rec X ***<:Ok*** $=$ *Var(X)*

   Der Typ X ist gemäß der Regel kontraktiv in X. Diese Regel ist falsch, da die Kontraktivität eines Typs unabhängig von der Veränderlichkeit ist. Es gilt die folgende Typregel:

   Var$(A) \downarrow X^i \Leftrightarrow A \downarrow X^i$

3. **Kontraktivitätstest in lokal definierten Typbezeichnern** Beispiel:

   Let Rec X ***<: Tuple*** R ***<:Ok*** T ***<:Ok end*** $=$
   Tuple Let $R = X$ ***Let*** $T = R.T$ ***end*** $\Rightarrow$ *Nichtterminierung*

   Der an X gebundene Tupeltyp ist gemäß der Typregeln kontraktiv. Der nicht auf Kontraktivität geprüfte Typ T im Inneren des Tupeltyps jedoch nicht. Der Subtyptest der beiden Tupeltypen, genauer der Subtyptest $R.T <:$ **Ok** führt in TLMIN zur Nichtterminierung:

   ## small: ***Tuple*** $R =A$ $T =R.T$ ***end***
   ## big : ***Tuple*** $R <:$***Ok*** $T <:$***Ok end***
   #### small: A
   #### big : ***Ok***
   ###### small: ***Tuple*** $R =A$ $T =R.T$ ***end***
   ###### big : ***Ok***
   #### small: $R.T$
   #### big : ***Ok***
   ###### small: $R.R.T$

```
###### big   : Ok
######## small: R.R.R.T
######## big   : Ok
########## small: R.R.R.R.T
########## big   : Ok
########### small: R.R.R.R.R.T
########### big   : Ok        (* Nichtterminierung *)
...
```

Um diesen Fall in die Typregeln einzubeziehen, müßten alle lokalen Typdefinitionen ebenfalls auf Kontraktivität getestet werden, im Beispiel $T \downarrow R.T$, bevor der Subtyptest ausgeführt wird. Dieser Fall wird sowohl im TL Typprüfer wie auch im TLMIN Typprüfer nicht behandelt.

5.7.3.2 Algorithmus

Der Kontraktivitätstest ist im Modul *tlCheckUtil* implementiert. Die Funktion *contractive* erwartet drei Argumente.

```
contractive(type :TLAST.Type
        parents :Iter.T(Signature)
        state :TLPreAST.State) :Bool
```

Der Parameter *state* enthält Übersetzerschalter und das Verzeichnis *errLog*, in dem die im Typprüfer entdeckten Typfehler eingetragen werden. Der Typ *type* wird inspiziert und auf Kontraktivität bezüglich der in der Signaturliste *parents* stehenden Typbezeichner untersucht. Eine Liste ist notwendig, um im Programmverlauf Zyklen zu erkennen und die Nichtterminierung des Algorithmus zu sichern. Der Kontraktivitätstest schlägt fehl, falls Typsignaturen auftreten, die bereits in der Liste *parents* stehen. Beispiel:

```
Let Rec X <:Ok = Y and Y <:Ok = Y
Let Rec X(A <:Ok) <:Ok = X(A)
```

In der ersten Beispielzeile enthält die Signaturliste *parents* beim ersten Test $Y \downarrow X$ zunächst die Signatur von X und beim zweiten Test $Y \downarrow X$ die Signatur von X und Y. Da die Signatur von Y in der Liste steht, bricht der Algorithmus mit einer Fehlermeldung ab. Wie die zweite Beispielzeile verdeutlicht, müssen auch Typoperatorbezeichner in die Signaturliste eingetragen werden. Beim Aufruf der Funktion *contractive* kann die Signaturliste leer sein oder die Signaturen der wechselseitig rekursiv definierten Typen enthalten. Im letzten Fall terminiert der Algorithmus genau einen Zyklus früher.

5.7.3.3 Nicht–kontraktive Werte

Da in TLMIN neben rekursiven Typen auch rekursive Werte definiert werden können, ist die Problematik der Kontraktivität auf die Werteebene übertragbar. So kann in TLMIN der folgende Wert definiert sein:

```
let rec x :Ok = x
Type checking
Translating
placeholder found in tml environment lookup
Target Code Generation
Running

ok :Ok
```

Die Ausgabe *placeholder found in tml environment lookup* während der Codegenerierungsphase zeigt, daß eine uninitialisierte Variable x referenziert wird, für die noch kein Code generiert wurde. Die Definition von x auf der Wertebene entspricht auf der Typebene der Definition eines nicht–kontraktiven Typen X:

```
Let Rec X <:Ok = X
```

Die Definition von x und y im folgenden Beispiel durchläuft hingegen ohne Fehlermeldungen die Codegenerierung und führt im Value–Unparser zum Systemabsturz:

```
let rec x :Tuple t :Tuple a :Int end end = tuple let t = y.t end
    and y :Tuple t :Tuple b :Int end end = tuple let t = x.t end

Type checking
Translating
Target Code Generation
Running

Bus error
```

Die Tupelkomponenten t in x und y referenzieren sich gegenseitig, ohne daß ein spezieller Wert spezifiziert wird. Im Value–Unparser wird auf die Tupelkomponenten a und b von t zugegriffen, obwohl dafür in der Codegenerierungsphase kein Code generiert wird. Auch zu diesem Beispiel kann eine entsprechende Typdefinition mit nicht–kontraktiven Typen T konstruiert werden:

```
Let Rec X <:Tuple T <:Tuple A <:Int end end = Tuple Let T = Y.T end
    and Y <:Tuple T <:Tuple B <:Int end end = Tuple Let T = X.T end
```

Die Beispiele lassen erkennen, daß die Definition nicht–kontraktiver Werte im Übersetzer nicht behandelt wird. Eine Lösung zur Überprüfung rekursiver Wertbindung ist der in [Mathiske 92] beschriebene einfache endliche Automat, der erlaubte und kritische Zustände unterscheidet. Befindet sich der Automat nach Eingabe eines syntaktischen Konstruktes in einem erlaubten Zielzustand, schließt dies das Vorhandensein eines uninitialisierten Bezeichners aus, während in einem kritischen Zustand ein uninitialisierter Bezeichner existieren kann. Ein erschöpfendes allgemeines Verfahren existiert nicht, da das Problem uninitialisierter Bezeichner unentscheidbar ist.

5.7.4 Zyklenerkennung

In Typsystemen, die Typkompatibilität über Namensäquivalenz definieren, führen rekursive Deklarationen zu keinen neuen Problemen bei der Typüberprüfung. Bei dem Test, ob ein Wert a den Typ A besitzt, oder ob ein Typ A Subtyp eines Typs B ist, kann der Typüberprüfungsalgorithmus bei jedem Auftreten eines Typnamens sofort die Typkompatibilität entscheiden, ohne die Definition des Typbezeichners expandieren zu müssen.

5.7.4.1 Subtypisierung

Bei Typsystemen mit strikter Strukturäquivalenz führt die Expansion von Bezeichnern beim Subtyptest zur Notwendigkeit, Zyklen zu erkennen, um die Terminierung des Algorithmus sicherzustellen [Amadio, Cardelli 92]. Ein entdeckter Zyklus führt, anders als beim Kontraktivitätstest, zu einem positiven Ergebnis, d.h. alle Paare $(small, big)$ in einem Zyklus stehen in Subtypbeziehung. Das folgende Beispiel zeigt den Subtyptest von zwei strukturäquivalenten Listentypen $List1$ und $List2$:

```
Let Rec List1 <:Ok = Tuple hd :Int  tl :List1 end
Let Rec List2 <:Ok = Tuple hd :Int  tl :List2 end
Let X <:List1 = List2

## small: List2
## big   : List1              (* List2 <:List1 gilt als bewiesen *)
#### small: List2
#### big   : Tuple hd :Int tl :List1  end
###### small: Tuple hd :Int tl :List2  end
###### big   : Tuple hd :Int tl :List1  end
######## small: Int
######## big   : Int
######## small: List2   (* Zyklus, da List2 <:List1 schon getestet wurde *)
######## big   : List1
```

Der Subtyptest $List2 <: List1$ führt nach mehreren Schritten auf denselben Test zurück. Nach dem ersten Schritt gilt die Annahme $List2 <: List1$ als bewiesen, so daß der Test bei der Entdeckung eines Zyklus abbricht. Der Zyklus wird beim Subtyptest von zwei Typbezeichnern erkannt. Im folgenden Beispiel ist bei der Zyklenerkennung ein Tupeltyp beteiligt:

```
Let Rec List1 <:Ok = Tuple hd :Int  tl :Tuple hd :Int  tl:List1 end end
Let List2 = Tuple hd :Int  tl :List1 end
Let X <:List1 = List2

## small: List2
## big   : List1
#### small: List2
#### big   : Tuple hd :Int tl :Tuple hd :Int tl :List1  end end
###### small: Tuple hd :Int tl :List1  end
###### big   : Tuple hd :Int tl :Tuple hd :Int tl :List1  end end
######## small: Int
```

```
######## big   : Int
######## small: List1  (* List1 <: Tuple... gilt als bewiesen *)
######## big   : Tuple hd :Int tl :List1  end
⋮
################ small: List1  (* Zyklus, da List1 <: Tuple... schon getestet wurde *)
################ big   : Tuple hd :Int tl :List1  end
```

Im Subtyptest trifft der Bezeichner $List1$ in tieferen Rekursionsstufen immer auf einen Tupeltyp. Der Bezeichner $List2$ ist nicht–rekursiv und wird im zweiten Schritt expandiert. Nach dem zweiten Schritt wird nur noch der innere Tupeltyp von $List1$ gegen den äußeren Tupeltyp von $List1$ getestet. Als Vergleichsoperation bei Tupeltypen dient die Objektidentität. Ein weiteres Beispiel zeigt, daß ein Subtyptest mit Typoperatorapplikationen einen Zyklus einleiten kann:

```
Let Rec Op(A <:Ok) <:Ok = Tuple hd :A tl :Op(A) end
Let X <:Op(Ok) = Op(Int)

## small: Op(Int)
## big   : Op(Ok)      (* Op(Int) <:Op(Ok) gilt als bewiesen *)
#### small: Op        (* Beginn der first chance *)
#### big   : Op
#### small: Int
#### big   : Ok
#### small: Ok        (* Scheitern der first chance *)
#### big   : Int
#### small: Op(Int)
#### big   : Tuple hd :Ok tl :Op(Ok)  end
###### small: Tuple hd :Int tl :Op(Int)  end
###### big   : Tuple hd :Ok tl :Op(Ok)  end
######## small: Int
######## big   : Ok
######## small: Op(Int)    (* Zyklus, da Op(Int) <:Op(Ok) schon getestet wurde *)
######## big   : Op(Ok)
```

Der Zyklus schließt sich mit dem Subtyptest von zwei Typoperatorapplikationen $Op(Int) <: Op(\mathbf{Ok})$. Die letzten zwei Beispiele haben gezeigt, daß Tupeltypen und Typoperatorapplikationen einen Zyklus einleiten können. Dabei ist die Identität der Tupeltypen wichtig. Bei der Typoperatorapplikation wird der Rückgabetyp kopiert, so daß die Identität von Tupeltypen verloren geht. Dies läßt den Schluß zu, daß im Subtyptest zu jeder einmal ausgeführten Typoperatorapplikation der Rückgabetyp gespeichert werden muß, so daß der Rückgabetyp kein zweites Mal berechnet wird. Daß die Speicherung von Rückgabetypen notwendig ist, zeigt das folgende Beispiel:

```
Let Rec List1(A <:Ok) <:Ok = Tuple hd :A tl :Tuple hd :Int tl :List1(A) end end
Let List2 <:Ok = Tuple hd :Int tl :List1(Int) end
Let X <:List1(Int) = List2

## small: List2
## big   : List1(Int)
```

```
#### small: List2
#### big   : Tuple hd :Int tl :Tuple hd :Int tl :List1(Int)  end  end
###### small: Tuple hd :Int tl :List1(Int)  end
###### big   : Tuple hd :Int tl :Tuple hd :Int tl :List1(Int)  end  end
######## small: Int
######## big   : Int
######## small: List1(Int)
######## big   : Tuple hd :Int tl :List1(Int)  end
########## small: Tuple hd :Int tl :Tuple hd :Int tl :List1(Int)  end  end
########## big   : Tuple hd :Int tl :List1(Int)  end
########### small: Int
########### big   : Int
########### small: Tuple hd :Int tl :List1(Int)  end                (* Zyklusbeginn *)
########### big   : List1(Int)
⋮
#################### small: Tuple hd :Int tl :List1(Int)  end
#################### big   : List1(Int)     (* Zykluserkennung *)
```

Der Subtyptest terminiert, weil der Rückgabetyp von $List1(Int)$ nur einmal berechnet wird. Mit der Übersetzeroption *do set cacheExposing false* wird die Speicherung von Rückgabetypen unterbunden. Der Typprüfer terminiert in diesem Fall nicht. Neben der Typoperatorapplikation darf auch die Typexpansion qualifizierter Typbezeichner nur einmal berechnet werden:

```
Let X = Tuple Let Rec A <:Ok = Tuple a1 :Tuple a2 :A end end end
Let Y = Tuple Let B = Tuple b :X.A end end
Let T <:Y.B = X.A

## small: X.A
## big   : Y.B
#### small: X.A
#### big   : Tuple b :X.A  end
###### small: Tuple a1 :Tuple a2 :X.A  end  end
###### big   : Tuple b :X.A  end
######## small: Tuple a2 :X.A  end                  (* Zyklusbeginn *)
######## big   : X.A
⋮
################ small: Tuple a2 :X.A  end          (* Zykluserkennung *)
################ big   : X.A
```

Der Typprüfer terminiert, da der expandierte Tupeltyp von $X.A$ nur einmal evaluiert wird. Auch dies kann durch die Option *do set cacheExposing false* kontrolliert werden.

5.7.4.2 Typregeln

Für den Subtyptest rekursiver Typen existieren keine Typregeln, da es in der abstrakten Syntax von TLMIN keine rekursive Typen gibt. Rekursive Typen werden mit Hilfe von rekursiven Signaturen repräsentiert. Der Subtyptest für rekursive Typen entspricht dem Subtyptest für nicht–rekursive Typen, wobei zusätzlich ein Algorithmus zur Zyklenerkennung vorhanden ist.

5.7.4.3 Speicherung von Subtypbeziehungen

Um Zyklen beim strukturellen Subtyptest von rekursiven Typen zu erkennen, müssen die zu beweisenden Subtypbeziehungen gespeichert werden [Amadio, Cardelli 92]. Eine gespeicherte Subtypbeziehung, die im weiteren Verlauf angetroffen wird, gilt als bewiesen. Im TLMIN Typprüfer werden Subtypbeziehungen als Typsyntaxbaumpaar des Typs *TypePair* in einer Menge des Typs *LocalCache* im Modul *tlCheck* abgelegt:

```
Let TypePair = Tuple small,big :Type end
Let LocalCache = set.T(TypePair)

let newLocalCache() :LocalCache =
  set.new(:TypePair
          fun(a,b :TypePair) :Bool
            tlAST.equalTypes(a.small b.small) andif
            tlAST.equalTypes(a.big b.big))
```

Als Operationen auf dem Subtypspeicher stehen die Operationen aus dem Mengenmodul *set* zur Verfügung. Im Typprüfer werden jedoch nur die Funktionen *member*, *insert* und *delete* verwendet, die im Modul *tlCheck* in die Funktionen *memberLocal*, *insertLocal* und *deleteLocal* umbenannt werden und deren Funktionalität sich mit dem Schalter *typeCheck-.switch.cacheLocalSubtyping* aufheben läßt. Der Speicheralgorithmus beim Subtyptest von zwei Typen *small* und *big* ist wie folgt:

1. Es wird mit der Funktion *memberLocal* getestet, ob das Paar mit *small* und *big* im Speicher enthalten ist. Wenn ja, dann terminiert der Subtyptest.
2. Wenn das Paar nicht gespeichert ist, wird es mit der Funktion *insertLocal* abgespeichert und der strukturelle Subtyptest fortgesetzt.
3. Nach Beendigung des Subtyptests von *small* und *big* wird das Paar aus dem Speicher mit der Funktion *deleteLocal* gelöscht.

Das Prädikat zum Vergleich von Typsyntaxbäumen in der Subtypmenge ist die Funktion *equalTypes* aus dem Modul *tlAST*:

```
let rec equalTypes(type1,type2 :Type) :Bool =
  case type1
  when ideCase with l then
    case type2
    when ideCase with l2 then
      l.first == l2.first andif samePath(l.path l2.path)
      else false
    end
  when appCase with l then
    case type2
    when appCase with l2 then
      equalTypes(l.oper l2.oper) andif
      equalSignatures(l.signatures l2.signatures)
    else false
```

```
  end
when okCase then type2?okCase
else type1 == type2
end
```

Zwei Bezeichner sind gleich, falls ihre deklarierenden Signaturen identisch und ihre Typpfade gleich sind. Zwei Typoperatorapplikationen sind gleich, falls die Typoperatoren und die Argumentsignaturen gleich sind. Die Funktion *equalSignatures* ist nicht dargestellt. Zwei Signaturlisten sind gleich, falls die Länge, die Signaturart und die Typen der entsprechenden Signaturen gleich sind. Zwei **Ok** Typen sind gleich. Für alle anderen Typen gilt Typgleichheit bei Objektidentität.

Wie in Abschnitt 5.7.4.1 gezeigt, darf eine Typoperatorapplikation und die Typexpansion eines qualifizierten Typbezeichners nur einmal während eines Subtyptests berechnet werden. Das berechnete Ergebnis muß daher abgespeichert werden. Hierzu dient der abstrakte Datentyp T im Modul $tlExposeCache$. Die Funktion *lookupIde* sucht im Speicher *cache* nach der Typexpansion eines qualifizierten Typbezeichners mit deklarierender Signatur *sig* und Typpfad *path*:

```
lookupIde(cache :T  sig :TLAST.Signature
        path :TLAST.Path  exact :Bool
        typeCheck :TLSwitches.TypeCheck) :TLAST.Type
```

Die Funktion *insertIde* fügt die Typexpansion *stripped* eines Bezeichners mit deklariender Signatur *sig* und Typpfad *path* in *cache* ein:

```
insertIde(cache :T  sig :TLAST.Signature  path :TLAST.Path
        exact :Bool  stripped :TLAST.Type
        typeCheck :TLSwitches.TypeCheck) :Ok
```

Die Funktion *lookupApp* sucht in *cache* nach der Typexpansion einer Typoperatorapplikation mit Typoperatorsignatur $operFirst$ und Typoperatorpfad $operPath$ sowie Argumentbindungen $bindings$ und Argumentsignaturen $signatures$:

```
lookupApp(cache :T  operFirst :TLAST.Signature
        operPath :TLAST.Path  signatures :TLAST.Signatures
        bindings :TLAST.Bindings  exact :Bool
        typeCheck :TLSwitches.TypeCheck) :TLAST.Type
```

Die Funktion *insertApp* fügt die Typexpansion *applied* einer Typoperatorapplikation in den Speicher ein:

```
insertApp(cache :T  operFirst :TLAST.Signature
        operPath :TLAST.Path  signatures :TLAST.Signatures
        bindings :TLAST.Bindings  exact :Bool
        applied :TLAST.Type
        typeCheck :TLSwitches.TypeCheck) :Ok
```

Falls die *lookup*–Funktionen das gewünschte Element nicht findet, wird der Typsyntaxbaum $notCached$ zurückgegeben. Falls die Typexpansion nichtabstrakt sein muß ($exact = true$) und nur die abstrakte Typexpansion im Speicher ist ($exact = false$), wird die Ausnahme $abstractError$ ausgelößt. Die Speichersemantik ist wie folgt:

- Wenn sich die Typexpansion nicht im Speicher befindet, wird die nichtabstrakte Typexpansion berechnet ($exact = true$) und, falls vorhanden, in den Speicher eingefügt. Falls nicht vorhanden, wird die immer verfügbare abstrakte Typexpansion berechnet ($exact = false$) und in den Speicher eingefügt.
- Wird die abstrakte Typexpansion mit $exact = false$ im Speicher gefunden und ein nichtabstrakter Typ benötigt ($exact = true$), dann wird die Ausnahme $abstractError$ ausgelößt.
- Ansonsten wird die im Speicher gefundene Typexpansion als Ergebnis zurückgeliefert.

Diese Speichersemantik hat den Vorteil, daß die Typexpansion mit $exact = true$ nicht erneut berechnet werden muß, falls eine Typexpansion mit $exact = false$ im Speicher gefunden wird. Diese Berechnung ist bereits einmal fehlgeschlagen.

Zumindest für die Dauer eines rekursiven Subtyptest müssen Subtypbeziehungen und Typexpansionen von Typoperatorapplikationen bzw. qualifizierten Typbezeichnern gespeichert werden. Im TLMIN Typprüfer bleiben die berechneten Typexpansionen für die Dauer einer Übersetzungsschleife (*toplevel execution loop*) gespeichert. Mit jeder neuen Schleife wird ein neuer Speicher angelegt und der alte Speicher verworfen. Diese Zeitspanne hat sich als vorteilhaft für die Übersetzung von Schnittstellen und Modulen erwiesen. Ein Modul wird in einer Übersetzungsschleife übersetzt. Da die Wahrscheinlichkeit für das wiederholte Auftreten derselben Typbezeichner bzw. Typoperatoren in verschiedenen Modulen gering ist und aufgrund der Größe und Anzahl der zu speichernden Objekte, ist eine Speicherung über diese Zeitspanne hinaus nicht sinnvoll.

Gewisse Subtypbeziehungen sollten dagegen länger als nur für die Dauer eines rekursiven Subtyptests gespeichert bleiben. So kann die Zeit für die wiederholte Typprüfung von Ausdrücken mit wechselseitig rekursiven Typen und Typoperatoren dramatisch reduziert werden, wenn Subtypbeziehungen für die Dauer einer ganzen Übersetzungschleife gespeichert werden.

Im TLMIN Typprüfer ist daher ein zweiter, effizient implementierter Speicher für Subtypbeziehungen enthalten, der als abstrakter Datentyp im Modul $tlSubTypeCache$ implementiert ist. Als Operationen stehen die Funktionen *new* zum Erzeugen eines Subtypspeichers, die Funktion *member* zur Abfrage eines Speichereintrags und die Funktion *insert* zum Einfügen eines Eintrags zur Verfügung. Im folgenden ist die Schnittstellenbeschreibung dargestellt:

```
T <:Ok
(* cache for saving subtype relations *)

new() :T
(* create new subtype cache *)

member(cache :T  small,big :TLAST.Type
       typeCheck :TLSwitches.TypeCheck) :Bool
```

```
(* test if subtype relation 'small' <: 'big' is member of 'cache' *)

insert(cache :T  small,big :TLAST.Type
       typeCheck :TLSwitches.TypeCheck) :Ok
(* insert subtype relation 'small' <: 'big' into 'cache' *)
```

Der Speicher wird mit jeder neuen Übersetzungsschleife neu generiert und der alte Speicher verworfen. Es werden nur Typpaare mit einem Bezeichner, einer Typoperatorapplikation oder **Ok** als Supertyp und einem Bezeichner oder einer Typoperatorapplikation als Subtyp gespeichert.

5.7.4.4 Speicherung von Subtypbeziehungen in TL

In TL werden Subtypbeziehungen von rekursiven Typbezeichnern für die Dauer eines rekursiven Subtyptests abgespeichert. Da die Gleichheit von de Bruijn–Indizes als Gleichheitsprädikat ungeeignet ist, wird die Identität der rekursiv definierten Typbezeichner verwendet. Der Subtyptest ist aus folgenden Gründen aber weder vollständig noch korrekt:

- Es werden nur Paare von rekursiven Typbezeichnern gespeichert. Typpaare mit Tupel– oder Typoperatorapplikationen, die Zyklen einleiten, werden nicht erkannt und führen zur Nichtterminierung des Typprüfers. Beispiel:

  ```
  Let Rec T(E <:Ok) <:Ok =
     Tuple
        empty() :Bool
        get() :E
        rest() :T(E)
     end
  Let X <:T(Ok) = T(Int)  ⇒  Nichtterminierung
  ```

  Die rekursive Definition von T entspricht der Definition des Typoperator $Iter.T$, mit dessen Hilfe das Iterationskonzept in TL verwirklicht ist. Der Subtyptest T(Int) <: T(**Ok**) terminiert nicht, da er zyklisch ist und Typoperatorapplikationen während des Subtyptests nicht abgespeichert werden.

- Die Identität der rekursiv definierten Typbezeichner ist als Prädikat zum Erkennen von Zyklen ungeeignet, da sie während der Qualifizierung von Typbezeichnern unverändert bleibt. Beispiel:

  ```
  Let T = Tuple
             R <:Ok
             Let Rec Y <:Ok = Tuple Z <:R end
          end
  let f(a,b :T) = tuple Let X <:a.Y = b.Y end
  ```

  Die typinkorrekte Subtypbeziehung $b.Y <: a.Y$ wird im Typprüfer nicht erkannt, da die Identität von Y in beiden Typen gleich ist.

Die richtige Wahl eines Prädikats zur Feststellung der Identität von Typen in unterschiedlichen Kontexten bleibt ein ungelöstes Problem.

5.7.5 Subsignaturbeziehungen

Rekursive Typen werden wie in Abschnitt 3.7 beschrieben mit rekursiven Signaturen definiert. Da insgesamt zwei Typsignaturarten in TLMIN existieren, gibt es auch zwei Arten von rekursiven Signaturen, die im folgenden Beispiel dargestellt sind:

Let F = ***Fun***(***Rec*** X <:***Ok*** = ***Tuple*** x :X ***end*** x :X) :X
Let G = ***Fun***(***Rec*** Y <:***Ok*** <:***Tuple*** y :Y ***end*** y :Y) :Y

Der Funktionstyp in der ersten Zeile enthält eine nichtabstrakte rekursive Typdefinition X. In der zweiten Beispielzeile ist Y im Funktionstyp mit einer rekursiven Subtypsignatur als abstrakter Typ definiert. Damit die Subtypbeziehung $G <: F$ gilt, ist die Subsignaturbeziehung

Rec X = ***Tuple*** x :X ***end*** x :X <:: ***Rec*** Y <:***Tuple*** y :Y ***end*** y :Y

zu prüfen. Der Subsignaturtest für rekursive Signaturen durchläuft folgende Einzelschritte:

1. Prüfe $X \overset{Rec}{<:} \mathbf{Tuple}\ x : X \mathbf{end} <:: Y \overset{Rec}{<:} \mathbf{Tuple}\ y : Y \mathbf{end}$
2. Substituiere $\langle X \to Y \rangle \mathbf{Tuple}\ y : Y \mathbf{end}$
3. Prüfe $tlsubtype \mathbf{Tuple} x : X \mathbf{end} \mathbf{Tuple} y : X \mathbf{end}$
4. Substituiere $\langle X \to Y \rangle y : Y$ in der restlichen Supertypsignatur.
5. Prüfe $x : X <:: y : X$
6. Prüfe $X <: X$

Die Substitution von Y durch X in Punkt zwei fehlt im Subsignaturtest für nichtrekursive Signaturen in Abschnitt 5.4.1. Da ein referenzierendes Auftreten eines Typbezeichners im Typ seiner deklarierenden Signatur nur in rekursiven Signaturen möglich ist, findet die Substitution niemals statt. Die folgenden drei Typregeln formalisieren den Subsignaturtest für rekursive Signaturen:

[Subsig Rec]

$$\frac{\begin{array}{c} S, X_1 \overset{Rec}{<:} A_1, \ldots, X_n \overset{Rec}{<:} A_n \vdash S' <:: \langle X_n \to Y_n \rangle \langle X_1 \to Y_1 \rangle S'' \ldots \\ S, X_1 \overset{Rec}{<:} A_1, \ldots, X_n \overset{Rec}{<:} A_n \vdash A_i <: \langle X_n \to Y_n \rangle \langle X_1 \to Y_1 \rangle B_i \ldots \qquad i = 1 \ldots n \quad 1 \leq n \end{array}}{S \vdash X_1 \overset{Rec}{<:} A_1, \ldots, X_n \overset{Rec}{<:} A_n, S' <:: Y_1 \overset{Rec}{<:} B_1, \ldots, Y_n \overset{Rec}{<:} B_n, S''}$$

[Subsig Rec Let]

$$\frac{\begin{array}{c} S, X_1 \overset{Rec}{=} A_1, \ldots, X_n \overset{Rec}{=} A_n \vdash S' <:: \langle X_n \to Y_n \rangle \langle X_1 \to Y_1 \rangle S'' \ldots \\ S, X_1 \overset{Rec}{=} A_1, \ldots, X_n \overset{Rec}{=} A_n \vdash A_i <: \langle X_n \to Y_n \rangle \langle X_1 \to Y_1 \rangle B_i \ldots \qquad i = 1 \ldots n \quad 1 \leq n \end{array}}{S \vdash X_1 \overset{Rec}{=} A_1, \ldots, X_n \overset{Rec}{=} A_n, S' <:: Y_1 \overset{Rec}{<:} B_1, \ldots, Y_n \overset{Rec}{<:} B_n, S''}$$

[Subsig Rec Let2]

$$\frac{\begin{array}{c} S, X_1 \overset{Rec}{=} A_1, \ldots, X_n \overset{Rec}{=} A_n \vdash S' <:: \langle X_n \to Y_n \rangle \langle X_1 \to Y_1 \rangle S'' \ldots \\ S, X_1 \overset{Rec}{=} A_1, \ldots, X_n \overset{Rec}{=} A_n \vdash A_i = \langle X_n \to Y_n \rangle \langle X_1 \to Y_1 \rangle B_i \ldots \qquad i = 1 \ldots n \quad 1 \leq n \end{array}}{S \vdash X_1 \overset{Rec}{=} A_1, \ldots, X_n \overset{Rec}{=} A_n, S' <:: Y_1 \overset{Rec}{=} B_1, \ldots, Y_n \overset{Rec}{=} B_n, S''}$$

In den Typregeln ist die Subsignaturbeziehung von wechselseitig rekursiven Signaturen angegeben. Hiervon ist die Deklaration eines einzelnen rekursiven Typs ein Spezialfall. Genauso sind die Typregeln für den Subsignaturtest nichtrekursiver Signaturen eine Spezialisierung der hier dargestellten Typregeln. Dies muß so sein, da jeder nichtrekursive Typ auch rekursiv definiert werden kann.

6. Einsatz des neuentwickelten Typüberprüfungsalgorithmus

In diesem Kapitel werden die Ergebnisse einer Messreihe präsentiert, die beim Einsatz des TLMIN- und des TL Typüberprüfungsalgorithmus erzielt wurden. Insgesamt wurden 9 Bibliotheken mit 411 Schnittstellen und Modulen übersetzt. Eine vollständige Darstellung der Einzelmesswerte befindet sich in Anhang I. Die folgende Tabelle zeigt die übersetzten Bibliotheken:

Verzeichnis	Komponenten	Schnittstellen	Module	Zeilen	Worte	Zeichen
machineenv	35	18	17	3224	10839	87715
stdenv	48	24	24	6314	22301	167186
bulkenv	73	37	36	11921	41543	350259
unixenv	16	8	8	1830	6052	43739
compenv	50	23	27	8192	26653	231416
tml0	24	12	12	8177	25167	207486
tl0	79	39	40	22522	72643	693453
egrammar	10	5	5	4163	14005	132357
tlmin	76	38	38	15339	44004	437446
insgesamt:	411	204	207	81579	263207	2351057

Die Verzeichnisse *machineenv*, *stdenv* und *bulkenv* bauen aufeinander auf und bilden die Basiskomponenten des Tycoon Systems. Das Verzeichnis *machineenv* enthält rechnerabhängige Komponenten. Im Verzeichnis *stdenv* werden Operationen für die Basisdatentypen *Int*, *Real*, *Bool*, *Char* und *String* zur Verfügung gestellt. Weitere Module behandeln die Ein- und Ausgabe von Zeichenströmen und die Definition von Zeitformaten. Das Verzeichnis *bulkenv* enthält eine Vielzahl von Modulen zur Definition von Massendatentypen wie Listen, B–Bäume und Streutabellen. Das Verzeichnis *unixenv* definiert Komponenten, die vom *Unix* Betriebssystem abhängig sind. Das Verzeichnis *compenv* stellt generische Funktionen zur automatischen Generierung von Parsern bereit. Das Verzeichnis *tml0* enthält Module zur Behandlung der untypisierten Sprache TML, die als Zwischenrepräsentation im Übersetzer von TL und TLMIN dient. Im größten Verzeichnis *tl0* stehen die Module und Schnittstellen des TL Übersetzers. Parser mit erweiterbarer Syntax können mit Hilfe des *egrammar*–Verzeichnis generiert werden. Der Programmcode des TLMIN Übersetzers steht schließlich im Verzeichnis *tlmin*.

Die letzten drei Spalten in der Tabelle enthalten die aufsummierte Anzahl der Zeilen, Wörter und Zeichen im jeweiligen Verzeichnis. Im folgenden Abschnitt werden die Typprüferzeiten des TL und des TLMIN Übersetzers analysiert und verglichen.

6.1 Vergleich der Prüfzeiten in TL und TLMIN

Die im folgenden aufgeführten Übersetzungszeiten für die beiden Typprüferimplementierungen wurden auf einer *Sun SparcStation* 20 gemessen. Die Werte sind jeweils in Sekunden mit zwei Nachkommastellen angegeben. Da durch den Mehrbenutzerbetrieb des Rechners Schwankungen in den Messzeiten möglich sind, sollten die Nachkommastellen in den hier zusammengefaßten Messzeiten nicht weiter beachtet werden. Die hier für Verzeichnisse aufgeführten Messwerte wurden unverändert aus den Ergebniszeilen der Messtabellen im Anhang I übernommen. Die folgende Tabelle enthält die nach Verzeichnissen gegliederten Typprüferzeiten:

Verzeichnis	TL	TLMIN	Differenz abs.	Differenz v. H.
machineenv	86.08	69.42	16.66	1.24
stdenv	258.77	164.65	94.12	1.57
bulkenv	932.11	670.01	262.10	1.39
compenv	414.69	229.31	185.38	1.81
tml0	1501.62	491.06	1010.56	3.06
tl0	4232.71	1188.11	3044.60	3.56
unixenv	63.90	65.70	- 1.80	0.97
egrammar	365.72	205.94	159.78	1.78
tlmin	3378.02	937.90	2440.12	3.60
insgesamt:	11233.62	4022.10	7211.52	2.79
	= 187.23 min	= 67.03 min	= 120.19 min	

Die Tabelle zeigt, daß das Verhältnis der Übersetzungszeiten von TLMIN und TL Typprüfer nicht genau bestimmt werden kann. Während der TL Typprüfer das *unixenv* Verzeichnis schneller prüft als der TLMIN Typprüfer, ist er bei der Übersetzung der Verzeichnisse *tl0* und *tlmin* um mehr als das 3.5 fache langsamer.

Die zwei folgenden Tabellen klassifizieren die Messwerte nach Zeitintervallen:

< sec	1	2	3	4	5	6	7	8	9	10	20	30	40	50	60
TL	53	22	40	19	21	20	19	16	18	14	69	28	16	5	2
TLMIN	154	62	29	18	8	16	6	6	6	8	38	22	10	9	5

< min	2	3	4	5	6	7	8	9	10	11	12
TL	28	7	7	2	2	1	0	1	0	0	1
TLMIN	14	1	0	0	0	0	0	0	0	0	0

In der ersten Tabelle sind in der ersten (zweiten,...) Spalte die Anzahl der geprüften Komponenten mit einer Typprüferzeit kleiner als eine (zwei,...) Sekunden aufgeführt. Jede übersetzte Komponente tritt dabei nur einmal als Eintrag auf. Beispielsweise wird eine Prüfzeit von weniger als einer Sekunde nur in der ersten Spalte und nicht nochmals in der zweiten, dritten,... Spalte eingetragen. Die letzte Spalte enthält die Anzahl der Komponenten mit Messzeiten zwischen 50 und 60 Sekunden. Darüber hinausgehende Zeiten stehen in der zweiten Tabelle, die Einträge in Minutenintervallen klassifiziert.

In TLMIN dauert die aufwendigste Typüberprüfung für eine Komponente weniger als drei Minuten. In TL dauert die längste Typüberprüfungszeit mehr als elf Minuten. Insgesamt gibt es 14 Komponenten mit einer Messzeit von mehr als drei Minuten.

Die folgende Tabelle zeigt die fünf Komponenten mit den größten Messzeiten:

TL		TLMIN	
tl0\tlSyntaxValue	661.43	tl0\tl2tml	134.24
tl0\tl2tml	537.72	tlmin\tl2tml	113.95
tlmin\tl2tml	418.67	tml0\tml2tvm	104.21
tlmin\tlMinSyntax	324.05	bulkenv\dDimDictionary1	100.70
tml0\tml2tvm	300.12	egrammar\eGramRep	93.06

Die Prüfzeiten sind jeweils in Sekunden angegeben. Ein wichtiger Grund für die teilweise sehr langen Typüberprüfungszeiten in TL ist die unzulängliche Speicherung von Subtypbeziehungen (vgl. Abschnitt 5.7.4.4). Die fehlende Speicherung wird besonders deutlich, wenn umfangreiche Module typgeprüft werden, bei denen die gleichen Subtyptests wiederholt durchzuführen sind.

6.2 Vergleich der ausgeführten Subtyptests

Beim Einsatz der beiden Typprüferimplementierungen wurde neben den Typprüferzeiten die Anzahl der ausgeführten Subtyptests und die Anzahl der auf Wohlgeformtheit geprüften Typen und Werte bestimmt. Desweiteren wurde die Anzahl der erfolgreichen (*first chance*) und fehlgeschlagenen (*second chance*) Subtyptests von Typoperatorapplikationen ermittelt.

Die folgende Tabelle enthält das nach Verzeichnissen unterteilte Ergebnis der Typüberprüfung in TLMIN:

Verzeichnis	TLMIN	<:	Typen	Werte	1.st	2.nd
machineenv	69.42	7943	3157	3579	155	0
stdenv	164.65	18914	6435	9850	663	0
bulkenv	670.01	71688	16652	27545	4057	193
compenv	229.31	24061	4909	12026	853	0
tml0	491.06	52105	3670	20218	792	0
tl0	1188.11	90615	12178	48301	3145	0
unixenv	65.70	3591	1394	2002	36	0
egrammar	205.94	16438	2677	7957	647	0
tlmin	937.90	63986	8435	30881	2434	0
insgesamt:	4022.10	349341	59507	162359	12782	193

In der zweiten Spalte (TLMIN) der Tabelle stehen nochmals die Messzeiten des TLMIN Typprüfers. Die meisten Subtypbeziehungen (Spalte: <:) werden im Verzeichnis *tl*0 geprüft, gefolgt von *bulkenv* und *tlmin*. Die meisten Typen werden im Verzeichnis *bulkenv* und die meisten Werte im Verzeichnis *tl*0 auf Wohlgeformtheit getestet. Die vorletzte Spalte enthält die Anzahl der Subtyptests von Typoperatorapplikationen, deren Gültigkeit durch den Subtyptest der Typoperatoren und durch die Signaturgleichheit der Operatorargumente (*first*

chance bestimmt werden konnte, ohne die Typoperatoren anwenden zu müssen. Wie oft Typoperatoren ausgeführt werden müssen, ist in der letzte Spalte ersichtlich. Nur in vier Modulen im Verzeichnis *bulkenv* treffen im Subtyptest Typoperatorapplikationen mit inkompatiblen Typoperatoren aufeinander.

Die folgende Tabelle zeigt die entsprechenden Daten für den Typprüfer von TL:

Verzeichnis	TL	<:	Typen	Werte	1.st	2.nd
machineenv	86.08	6549	3074	3456	151	0
stdenv	258.77	20673	7108	9467	936	0
bulkenv	932.11	153579	16239	22389	17017	340
compenv	414.69	25723	4659	11408	1194	0
tml0	1501.62	30095	3270	17617	1244	0
tl0	4232.71	94204	11850	46847	4455	0
unixenv	63.90	2823	886	2488	7	0
egrammar	365.72	44504	2264	7768	1724	0
tlmin	3378.02	63554	11150	30269	2663	0
insgesamt:	11233.62	441704	60500	151709	29391	340

Im Vergleich mit der vorherigen Tabelle wird deutlich, daß die Anzahl der geprüften Werte und Typen bei beiden Implementierungen etwa gleich ist. Große Unterschiede sind bei den Subtyptests in den Verzeichnissen *bulkenv* und *egrammar* erkennbar, wo der TL Typprüfer wesentlich mehr Subtyptests ausführt. Der Grund hierfür ist das bereits beschriebene Problem der Speicherung von Subtypbeziehungen. Allerdings wird in einigen Verzeichnissen (*machineenv*, *tml*0, *unixenv* und *tlmin*) aus folgenden Gründen z. T. erheblich weniger getestet:

- Der TL Typprüfer testet nicht, ob Typen in Wertsignaturen Subtypen von **Ok** sind.
- Der TLMIN Typprüfer testet beim Subsignaturtest von mit **Let** definierten Typbindungen die Signaturtypen auf Gleichheit durch zwei Subtyptests (Typregel *[Subsig Ide Let2]*), während der TL Typprüfer nur einen einfachen Subtyptest ausführt.
- Durch die fehlerhafte Speicherung von Subtypbeziehungen rekursiver Typbezeichner in TL werden weniger Subtypbeziehungen geprüft.

Insgesamt kann festgehalten werden, daß die Speicherung von bereits getesteten Subtypbeziehungen besonders bei der Überprüfung umfangreicher Anwendungen die Typprüferzeiten teilweise drastisch reduzieren kann. Bei kleineren Anwendungen und in Programmen, bei denen zahlreiche Subtypbeziehungen mit unterschiedlichen Typen auftreten, erweißt sich die Datenspeicherung hingegen als Nachteil.

7. Fazit und Ausblick

7.1 Fazit

Die statische Typüberprüfung in strikt typisierten polymorphen Programmiersprachen garantiert die Abwesenheit einer Vielzahl von Fehlersituationen zur Programmlaufzeit. Beispiele für Fehler, die während der statischen Programmanalyse entdeckt werden, sind inkompatible Funktionsparameteranzahl oder -typen, inkompatible Funktionsergebnistypen, inkompatible Wertbindungen und der Zugriff auf undefinierte Bezeichner oder nicht vorhandene Tupel-, Record- und Ausnahmekomponenten. Als Folge der statischen Typprüfung kann durch die Minimierung der dynamischen Laufzeitüberprüfungen effizienter und zuverlässiger Programmcode generiert werden. Beispiele für verbleibende dynamische Laufzeittests in TLMIN sind unzulässige Indizes beim Feldzugriff (x[i]) und Namenskonflikte bei Recorderweiterungen.

Zum Erfolg einer typisierten Programmiersprache trägt ein möglichst kleiner Satz an vordefinierten Typ- und Wertkonstruktoren bei, die durch eine einfache und klare Semantik gekennzeichnet sind. Funktionstypen, kartesische Produkttypen (Tuple, Record) und Typvereinigungen (Varianter Tuple bzw. Record) werden im *Ideal* Modell als Wertemengen mit gleichen strukturellen Eigenschaften interpretiert, die durch Mengenoperationen (Produktbildung, Mengenvereinigung) aus Basistypen induktiv zusammengesetzt werden.

Die nahe semantische Verwandtschaft von Record- und Tupeltypen in TL hat zum geringen Nutzen von Recordtypen in den vorhandenen TL Bibliotheken geführt. Es stellt sich die Frage, ob Recordstrukturen nicht besser aus der Sprache vollständig verbannt werden sollten. Diese Frage stellt sich aufgrund der unpraktischen Semantik ebenfalls für variante Tupeltypen, in denen die Reihenfolge der Tupelvarianten für die Subtypisierung wichtig ist. Eine sinnvollere Alternative sind hingegen variante Recordtypen, die von der Variantenreihenfolge abstrahieren, aber in TL und TLMIN nicht vorhanden sind.

Die erweiterbare Syntax von TLMIN unterstützt die Möglichkeit zur flexiblen *add–on* Spracherweiterung unter Beibehaltung des minimalen Typsystems. So läßt sich die Syntax von TLMIN zur TL Syntax einfach erweitern. Beispielsweise wird das **while** Konstrukt von TL in TLMIN als **loop** Schleife mit einem **if** Konstrukt abgebildet. Die Minimalität des Sprachkerns sowie die Unterstützung uniformer und flexibler Benennungs-, Bindungs- und Sichtbarkeitsregeln für alle semantischen Objekte in TLMIN hat einen positiven Einfluß auf den Umfang und die Komplexität des formalen Typsystems und damit auch auf den Umfang und die Komplexität des Typüberprüfungsalgorithmus.

Trotz des minimalen Sprachkerns erweist sich die Implementierung eines Typprüfers für TLMIN aufgrund der höhersprachlichen Konzepte wie abstrakte Typbezeichner, Typoperatoren, rekursive Typen und Subtypisierung immer noch als große Herausforderung. Ein Blick auf

die Typtheorie zeigt, daß eine Vielzahl von Typüberprüfungsaufgaben bereits bei der Untersuchung von einfacheren Lambda–Kalkülen vorzufinden sind. Beispielsweise ist die Sichtbarkeitsanalyse von Wertbezeichnern im untypisierten und von Typbezeichnern im einfach typisierten Lambda–Kalkül erforderlich. Typkompatibilitäten sind in allen typisierten Kalkülen zu bestimmen. Die Äquivalenz von Typbezeichnern reicht von einem einfachen Stringvergleich im typisierten Lambda–Kalkül zweiter Ordnung über die β–Äquivalenz bei Einführung von Typoperatoren im Lambda–Kalkül höherer Ordnung bis zum Vergleich von de Bruijn–Indizes oder Objektreferenzen in TL. Die Substitutionsoperation definiert im untypisierten Lamda–Kalkül das Ergebnis einer Funktionsapplikation. Eine entsprechende Operation zur Berechnung von Funktionsrückgabetypen findet sich im typisierten Lambda–Kalkül zweiter Ordnung und zur Berechnung von Typoperatorergebnissen in $F_{\infty}^{\leq}$. Die Subtypregeln für Typbezeichner, Funktionstypen und Typoperatoren sind bereits in $F_{\infty}^{\leq}$ definiert. Obwohl dieser typisierte Lambda–Kalkül höherer Ordnung gegenüber TLMIN einen deutlich kleineren Sprachkern hat, ist die Komplexität des Typsystems enorm und die Untersuchung der typtheoretischen Eigenschaften Gegenstand der aktuellen Forschung. Ein Ergebnis der Forschung ist die Unentscheidbarkeit der Subtyprelation für bestimmte fehlerhafte Subtypbeziehungen, die sich im TLMIN Typprüfer durch Nichtterminierung bemerkbar macht.

Ein wichtiges Ziel der Neuimplementierung des TLMIN Typprüfers ist die Wahl einer geeigneten Repräsentation von Typbezeichnern auf der Basis von Objektreferenzen statt de Bruijn–Indizes. Letztere verfügen über theoretisch wohlverstandene Eigenschaften, sind jedoch für den praktischen Einsatz bei der Implementierung von Typprüfern mit rekursiven und dynamischen Typen aus folgenden Gründen nur bedingt geeignet:

- Die Wahrung der korrekten Indizierung von de Bruijn–Indizes an die an sie gebundenen Objekte im statischen Kontext muß explizit durch Indexmanipulationen (*lifting*) sichergestellt werden.
- Eine fehlerhafte Indizierung kann fatale Folgen haben und der Fehlerursprung nur mühsam ermittelt werden.
- Schon die Repräsentation von rekursiven Typen in TL, in denen sämtliche wechselseitig rekursiven Signaturen einer Typdefinition enthalten sind, ist kompliziert und umständlich.
- Es ist unklar, wie Subtypbeziehungen mit de Bruijn–Indizes beim Subtyptest von rekursiven Typen korrekt gespeichert werden können. Die Lösung in dem vor Begin dieser Arbeit vorliegenden Typprüfer von TL ist in den meisten Anwendungen akzeptabel aber fehlerhaft und unvollständig.
- Die Repräsentation semantischer Objekte durch abstrakte Syntaxbäume, die Signaturen in einem einzigen statischen Kontext indizieren, erschweren eine kontextübergreifende Typüberprüfung, z.B. bei dynamischer Typisierung oder Codemigration.

Der große Vorteil der Objektreferenzierung besteht in der kontextunabhängigen Repräsentation semantischer Objekte ausschließlich über abstrakte Syntaxbäume mit folgenden Vorteilen für die Typprüferimplementierung:

- Es sind keine zusätzlichen Indexmanipulationen erforderlich, die das Verständnis der Implementierung erschweren.

- Typbezeichner referenzieren die deklarierende Signatur des Bezeichners durch eine stabile Verzeigerung im Objektspeicher.
- Rekursive Typbezeichner führen zu zyklischen Signaturen.
- Compilerwerte in Form von abstrakten Syntaxbäumen können zur Laufzeit unverändert genutzt und über Speichergrenzen hinaus versendet werden.

Bei Einsatz von Objektreferenzen ergeben sich im Vergleich zu de Bruijn–Indizes folgende Nachteile:

- Die Substitution von Bezeichnern beim Subsignaturtest und beim Subtyptest von Funktionstypen bzw. Typoperatoren geschieht durch Kopieren und Ersetzen von Syntaxbäumen, während bei der de Bruijn–Implementierung eine Manipulation im statischen Kontext ausreichend ist, ohne Syntaxbäume zu kopieren.
- Die auf Bezeichnern induzierte Ordnungsrelation

  $i < j \;\Leftrightarrow\;$ *"i ist später definiert als j"*

  für zwei de Bruijn–Indizes i und j kann für Optimierungen im Typüberprüferalgorithmus ausgenutzt werden. Eine Ordnung für Objektidentitäten, z.B. unter Ausnutzung von Datum und Urzeit der generierten Speicherobjekte, fehlt im Tycoon–System.

Die Repräsentation von qualifizierten Typbezeichnern durch die deklarierende Signatur des ersten Pfadelementes und eines restlichen Pfades, bestehend aus ganzzahligen Positionswerten bei Tupelkomponenten bzw. Stringwerten bei Recordkomponenten ist vorteilhaft für die Konkatenation von Pfaden innerhalb des Qualifizierungsalgorithmus. Um Pfade beim Subtyptest von abstrakten Typbezeichnern zu vergleichen, brauchen nur Zahlen oder Strings verglichen zu werden. Abstrakte Datentypen sind wertabhängig, d.h. Typen wie $nat1.T$ und $nat2.T$ sind grundsätzlich inkompatibel, obwohl $nat1$ und $nat2$ denselben Typ besitzen. Die Wertabhängigkeit wird im TLMIN Typprüfer bewußt auf Wertbezeichner eingeschränkt, um Übersetzungsphase und Programmlaufzeit klar zu trennen.

Die Repräsentation von rekursiven Typbezeichnern durch zyklische Signatursyntaxbäume entspricht dem Wesen dieser Typen und bereitet bei der Implementierung im Typüberprüfungsalgorithmus keine Probleme. Da nichtrekursive Typen in TLMIN immer auch als rekursive Typen definierbar sind, können nichtrekursive Typen als ein verzichtbarer Spezialfall im Typsystem angesehen werden. Im TLMIN Typsystem und in der Typprüferimplementierung werden rekursive und nichtrekursive Typen aus Gründen der Optimierung im Compiler und einer besseren Darstellung der Sprache bewußt getrennt. Der gegenüber TL in TLMIN ergänzte rekursive Polymorphismus (*f–bounded–polymorphism*) in Form von rekursiven Subtypsignaturen ist im Typprüfer äquivalent zu den schon vorhandenen rekursiven Typdefinition in Form von rekursiven **Let**–Signaturen implementiert.

Zur Beschreibung der formalen statischen Semantik dienen Typsysteme, die aus einer Menge von Typregeln bestehen. Eine Rückwärtsinterpretation der Typregeln definiert einen Beweiseralgorithmus. Hierbei ist zu beachten, daß

- es Typregeln gibt, wie *[Value Subsumption]* oder *[Subtype transitive]*, die gewissermaßen "nicht–kontraktiv" sind, da ihre Anwendung unmittelbar zur gleichen Problemstellung und damit zur rekursiven Anwendung der Typregel zurückführt, so daß der Algorithmus nicht terminiert. Wie in Abschnitt 4.5 gezeigt, können alternative Regeln konstruiert werden, die sich problemlos anwenden lassen.
- beim Kontraktivitätstest und beim Subtyptest rekursiver Typen ebenfalls Zyklen auftreten können, die erkannt werden müssen, um die Nichtterminierung des Typprüfers zu vermeiden. Zyklische Strukturen werden erkannt, indem der aktuelle Strukturknoten mit der Menge der bereits durchlaufenen Knoten verglichen wird. Ist der aktuelle Knoten in der Menge enthalten, liegt ein Zyklus vor. Ansonsten wird der aktuelle Knoten in die Knotenmenge eingetragen und die Abarbeitung der Struktur fortgesetzt.

Die enge Verbindung von Typsystem und Typüberprüfungsalgorithmus wird in Kapitel 4 deutlich, wo zu jedem dargestellten Typsystem ein direkt aus den Typregeln abgeleiteter Typüberprüfungsalgorithmus vorgestellt wird. Das im Anhang H stehende Typsystem von TLMIN formalisiert den im Rahmen dieser Arbeit implementierten Typprüfer und abstrahiert von konkreten Implementierungsdetails.

7.2 Ausblick

In dieser Arbeit wurden die Kernprobleme der Typüberprüfung in TLMIN vorgestellt. Die Lösungsansätze resultieren zum einen in dem Versuch einer formalen Spezifikation der semantischen Analyse durch ein Typsystem und zum zweiten in der Implementierung eines Typüberprüfungsalgorithmus. Die Erfahrung hat gezeigt, daß die Korrektheit und Vollständigkeit des TLMIN Typsystems ohne formalen Beweis nur durch "Anschauen" der Typregeln nicht gewährleistet werden kann. Der Grund hierfür liegt in der Anzahl und Komplexität der Typregeln. Das im Anhang H präsentierte Typsystem von TLMIN ist die Synthese aus dem bestehenden TL Typsystem und der Erfahrung, die bei der Implementierung und beim Testen des TLMIN Typüberprüfungsalgorithmus gesammelt wurde.

Das folgende Beispiel zeigt einen bisher im Typsystem und im Typüberprüfungsalgorithmus von TLMIN und TL noch unberücksichtigte Fall, der zu einem Laufzeitfehler führt:

```
Let Abs = Tuple T <:Ok  t :T  f(t :T) :Int end
let abs :Abs = tuple Let T = Int  let t :T = 1  let f(t :T) :Int = t end
let var abs2 = abs
let abs3 :Abs = tuple Let T = String  let t :T = "a"  let f(t :T) :Int = string.size(t) end
let x = abs2.t
abs2 := abs3
abs2.f(x)     ⇒ Laufzeitfeher
```

Der Bezeichner *abs*2 wird in der dritten Beispielzeile durch das Schlüsselwort **var** destruktiv an den Bezeichner *abs* gebunden. Die veränderliche Wertbindung von *abs*2 wird in der vorletzten Zeile überschrieben. Vor der Zuweisung erfolgt ein Zugriff auf die Komponente $abs2.t$, die an den Bezeichner x gebunden wird. Der Aufruf der Funktion $abs2.f$ mit dem Argument x wird vom Typprüfer akzeptiert und führt zu einem Laufzeitfehler, da die Implementierung von f ein Stringargument erwartet und mit einem ganzzahligen Argument aufgerufen wird.

Bei der Definition von veränderlichen Wertbezeichnern muß im Typprüfer getestet werden, ob der zugewiesene Wert oder Komponenten davon einen abstrakten Datentyp besitzen. In diesem Fall ist ein Typfehler zu erzeugen. Neben dem Algorithmus muß das Typsystem entsprechend geändert werden.

Eine weitere Aufgabe betrifft die Typüberprüfung von rekursiven Typen. Wie in Abschnitt 3 auf Seite 93 beschrieben, muß der Kontraktivitätstests für rekursive Typen im Typüberprüfungsalgorithmus verbessert werden. Typbezeichner, die in lokalen Typsignaturen deklariert werden, müssen ebenfalls auf Kontraktivität untersucht werden. Gleichzeitig sind die Typregeln im Typsystem entsprechend anzupassen. Der nicht vollständig korrekte Kontraktivitätstest kann in Sonderfällen zur Nichtterminierung im TLMIN Typprüfer führen.

In TL und TLMIN sind gegenwärtig als Typoperatorargumente nur Typbindungen erlaubt (vgl. Abschnitt 3.5 auf Seite 27). Damit Wertbindungen zulässig sind, muß die Syntax der Sprache geändert werden. Beim Typoperatoraufruf muß wie bei polymorphen Funktionsaufrufen vor anonymen Typargumenten ein Doppelpunkt stehen, um Typ- und Wertargumente syntaktisch unterscheiden zu können.

Die Typüberprüfung in TLMIN geht über die Implementierung des TLMIN Typsystems hinaus. Beim Aufruf von polymorphen Funktionen und beim Aufruf von Typoperatoren können Typargumente fehlen. Sie werden durch partiell instantiierte Typvariablen ersetzt, die im weiteren Verlauf der Typüberprüfung durch einen rudimentären Unifikationsalgorithmus vollständig instantiiert werden. Beispiel:

```
let poly(X <:Ok  x :X) :X = x
poly(1)  ⇒  1 :Uni[Int]
```

Die polymorphe Funktion *poly* erwartet beim Funktionsaufruf ein Typ- und ein Wertargument. Beim Aufruf mit nur einem Wertargument wird das Typargument inferiert, das den Rückgabetyp bestimmt. Der Typ $Uni[Int]$ kennzeichnet eine vollständig instantiierte Typvariable und ist äquivalent zu Int. Die Korrektheit und Leistungsfähigkeit des Unifikationsalgorithmus sollte analysiert und entsprechende Typregeln erstellt werden.

Im TLMIN Typprüfer werden Bezeichner durch eine Referenz auf die deklarierende Signatur repräsentiert. Diese Darstellung hat gegenüber de Bruijn–Indizes den großen Vorteil vom statischen Kontext unabhängig zu sein. Da im Tycoon System der statische Kontext und die Laufzeitumgebung im gleichen persistenten Speicher angesiedelt sind, kann dieselbe Typrepräsentation zur Übersetzungszeit und zur Laufzeit verwendet werden. Die in TLMIN noch ausstehende Implementierung von dynamischen Typen auf der Grundlage von Objektreferenzen sollte im Vergleich zu einer Implementierung basierend auf de Bruijn–Indizes problemlos möglich sein.

In TL wird die separate Übersetzung von Programmeinheiten durch eine im Übersetzer eingebaute Modulverwaltung realisiert. Die Modulverwaltung berechnet die Abhängigkeiten von Modulen (*modules*), Schnittstellen (*interfaces*) und Bibliotheken (*libraries*) und führt gegebenenfalls automatisch eine Neuübersetzung und Neubindung von Programmkomponenten aus. Der TLMIN Übersetzer verfügt derzeit noch nicht über eine Modulverwaltung. Im Rahmen einer Diplomarbeit [?] wird eine außerhalb des Übersetzers implementierte *add-on* Lösung vorbereitet.

A. Der untypisierte Lambda–Kalkül F_0

Syntax

Value ::=
Value-Identifier | *Value-Constant*
| **fun** "(" *Value-Identifier* ")" *Value* *Abstraktion*
| *Value* "(" *Value* ")" *Applikation*
| "{" *Value* "{"
;

Typsystem

Kontext:

[Sig empty]
$$\frac{}{\vdash \emptyset \ sig}$$

[Sig ide]
$$\frac{\vdash S \ sig \qquad x \notin Dom(S)}{\vdash S, x \ sig}$$

Value:

[Value ide]
$$\frac{\vdash S, x, S' \ sig}{S, x, S' \vdash x \ value}$$

[Value → introduction]
$$\frac{S, x \vdash b \ value}{S \vdash \mathbf{fun}\,(x)\,b \ value}$$

[Value → elimination]
$$\frac{S \vdash f \ value \qquad S \vdash a \ value}{S \vdash f\,(a) \ value}$$

Typüberprüfungsalgorithmus

```
let issig(S :_Env) :Bool =
    if S = ∅ then true                                  (* Regel: [Sig empty] *)
    elsif S = S1,x then issig(S1)                       (* Regel: [Sig ide] *)
    else false
    end

let lookupValueIde(S :_Env  x :_Ide) :Bool =
    if S = ∅ then false                                 (* Regel: Sig empty *)
    elseif S = S1,x then true                           (* Regel: Sig ide *)
    elseif S = S1,y then lookupValueIde(S1 x)
    else false
    end

let isvalue(S :_Env  a :_Value) :Bool =
    if a = x and lookupValueIde(S x) then issig(S)      (* Regel: [Value ide] *)
    elsif a = fun(x) b then isvalue(S,x b)              (* Regel: [Value → introduction alg] *)
    elsif a = f(b) and isvalue(S f) then isvalue(S b)   (* Regel: [Value → elimination] *)
    else false
    end
```

B. Der einfach typisierte Lambda–Kalkül F_1

Syntax

Type ::=
    Type-Constant
  | ***Fun*** *"(" Value-Identifier ":" Type ")" ":" Type* *Funktionstyp*
  | *"{" Type "}"*
;
Value ::=
    Value-Identifier | Value-Constant
  | ***fun*** *"(" Value-Identifier ":" Type ")" Value* *Abstraktion*
  | *Value "(" Value ")"* *Applikation*
  | *"{" Value "}"*
;

Typsystem

Kontext:

$$\frac{}{\vdash \emptyset \ sig}\ [\textit{Sig empty}] \qquad \frac{S \vdash A :: \mathbf{TYPE} \quad x \notin Dom(S)}{\vdash S, x : A \ sig}\ [\textit{Sig ide}]$$

Typen:

$$\frac{\vdash S \ sig \quad A \in \{Bool, Int, String\}}{S \vdash A :: \mathbf{TYPE}}\ [\textit{Type builtin}] \qquad \frac{S, x : A \vdash B :: \mathbf{TYPE}}{S \vdash \mathbf{Fun}\,(x : A) : B :: \mathbf{TYPE}}\ [\textit{Type} \rightarrow]$$

Werte:

$$\frac{\vdash S, x : A, S' \ sig}{S, x : A, S' \vdash x : A}\ [\textit{Value ide}] \qquad \frac{S, x : A \vdash b : B}{S \vdash \mathbf{fun}\,(x : A)\, b : \mathbf{Fun}\,(x : A) : B}\ [\textit{Value} \rightarrow \textit{introduction}]$$

$$\frac{S \vdash f : \mathbf{Fun}\,(x : A) : B \quad S \vdash a : A}{S \vdash f\,(a) : B}\ [\textit{Value} \rightarrow \textit{elimination}]$$

Typüberprüfungsalgorithmus

```
let lookupValueIde(S :_Env  x :_Ide  A :_Type) :Bool =
    if S = ∅ then false                                   (* Regel: Sig empty *)
    elseif S = S1,x:A then true                           (* Regel: Sig ide *)
    elseif S = S1,y:B then lookupValueIde(S1 x A)
    else false
    end

let hastype(S :_Env  a :_Value  A :_Type) :Bool =
    if a = x then lookupValueIde(S x A)                   (* Regel: [Value ide] *)
    elsif a = fun(x:B) c and                              (* Regel: [Value → introduction alg] *)
        haskind(S B TYPE) and
        hastype(S,x:B c C) then A = Fun(x:B):C
    elsif a = f(b) and hastype(S f Fun(x:B):C) and        (* Regel: [Value → elimination] *)
        hastype(S b B) then A = C
    else false
    end

let haskind(S :_Env  A :_Type  K :_Kind) :Bool =
    if A ∈ {Bool,Int,String} then K = TYPE                (* Regel: [Type builtin] *)
    elsif A = Fun(x:B):C and haskind(S B TYPE)            (* Regel: [Type → alg] *)
        and haskind(S,x:B C TYPE) then K = TYPE
    else false
    end
```

C. Der getypte Lambda–Kalkül zweiter Ordnung F_2

Syntax

Type ::=
    Type-Constant | *Type-Identifier*
  | ***Fun*** *"(" Value-Identifier ":" Type ")" ":" Type* *Funktionstyp*
  | *"{" Type "}"*
  | ***Fun*** *"(" Type-Identifier ")" ":" Type* *polymorpher Funktionstyp*
;
Value ::=
    Value-Identifier | *Value-Constant*
  | ***fun*** *"(" Value-Identifier ":" Type ")" Value* *Abstraktion*
  | *Value "(" Value ")"* *Applikation*
  | *"{" Value "}"*
  | ***fun*** *"(" Type-Identifier ")" Value* *Typabstraktion*
  | *Value "(" Type ")"* *Typapplikation*
;

Typsystem

Kontext:

$$\frac{}{\vdash \emptyset \ sig}\ [\textit{Sig empty}] \qquad \frac{S \vdash A :: \mathbf{TYPE} \quad x \notin Dom(S)}{\vdash S, x : A \ sig}\ [\textit{Sig ide}] \qquad \frac{\vdash S \ sig \quad X \notin Dom(S)}{\vdash S, X :: \mathbf{TYPE} \ sig}\ [\textit{Sig Ide}]$$

Typen:

$$\frac{\vdash S \ sig \quad A \in \{Bool, Int, String\}}{S \vdash A :: \mathbf{TYPE}}\ [\textit{Type builtin}] \qquad \frac{\vdash S, X :: \mathbf{TYPE}, S' \ sig}{S, X :: \mathbf{TYPE}, S' \vdash X :: \mathbf{TYPE}}\ [\textit{Type Ide}]$$

$$\frac{S, x : A \vdash B :: \mathbf{TYPE}}{S \vdash \mathbf{Fun}\,(x : A) : B :: \mathbf{TYPE}}\ [\textit{Type} \rightarrow] \qquad \frac{S, X :: \mathbf{TYPE} \vdash B :: \mathbf{TYPE}}{S \vdash \mathbf{Fun}\,(X) : B :: \mathbf{TYPE}}\ [\textit{Type} \forall]$$

Werte:

$$\frac{\vdash S, x : A, S' \ sig}{S, x : A, S' \vdash x : A}\ [\textit{Value ide}] \qquad \frac{S, x : A \vdash b : B}{S \vdash \mathbf{fun}\,(x : A)\, b : \mathbf{Fun}\,(x : A) : B}\ [\textit{Value} \rightarrow \textit{introduction}]$$

$$\frac{S \vdash f : \mathbf{Fun}\,(x : A) : B \quad S \vdash a : A}{S \vdash f\,(a) : B}\ [\textit{Value} \rightarrow \textit{elimination}] \qquad \frac{S, X :: \mathbf{TYPE} \vdash b : B}{S \vdash \mathbf{fun}\,(X)\, b : \mathbf{Fun}\,(X) : B}\ [\textit{Value} \forall \textit{ introduction}]$$

$$\frac{S \vdash f : \mathbf{Fun}\,(X) : A \quad S \vdash B :: \mathbf{TYPE}}{S \vdash f\,(B) : \langle B \rightarrow X \rangle A}\ [\textit{Value} \forall \textit{ elimination}]$$

Typüberprüfungsalgorithmus

```
let lookupValueIde(S :_Env  x :_Ide  A :_Type) :Bool =
   if S = ∅ then false                                   (* Regel: Sig empty *)
   elsif S = S1,x:A then true                            (* Regel: Sig ide *)
   elsif S = S1,y:B then lookupValueIde(S1 x A)
   elsif S = S1,x :: K then false
   elsif S = S1,Y :: K then lookupValueIde(S1 x A)
   end

let lookupTypeIde(S :_Env  X :_Ide  K :_Kind) :Bool =
   if S = ∅ then false                                   (* Regel: Sig empty*)
   elsif S = S1,X :: K then true                         (* Regel: Sig Ide *)
   elsif S = S1,Y :: L then lookupTypeIde(S1 X K)
   elsif S = S1,X:B then false
   elsif S = S1,y:B then lookupValueIde(S1 X K)
   end

let hastype(S :_Env  a :_Value  A :_Type) :Bool =
   if a = x then lookupValueIde(S x A)                   (* Regel: [Value ide] *)
   elsif a = fun (x:B) c and                             (* Regel: [Value → introduction alg] *)
      haskind(S B TYPE) and
      hastype(S,x:B c C) then A = Fun (x:B): C
   elsif a = f(b) and hastype(S f Fun (x:B): C) and      (* Regel: [Value → elimination] *)
      hastype(S b B) then A = C
   elsif a = fun (X) b and                               (* Regel: Value ∀ introduction *)
      hastype(S,X :: TYPE b B) then A = Fun (X): B
   elsif a = f(B) and hastype(S f Fun (X): C) and        (* Regel: Value ∀ elimination *)
      haskind(S B TYPE) then A = ⟨B → X⟩C
   else false
   end

let haskind(S :_Env  A :_Type  K :_Kind) :Bool =
   if A ∈ {Bool,Int,String} then K = TYPE                (* Regel: [Type builtin] *)
   elsif A = Fun (x:B): C and haskind(S B TYPE)          (* Regel: [Type → alg] *)
      and haskind(S,x:B C TYPE) then K = TYPE
   elsif A = X then lookupTypeIde(S X K)                 (* Regel: Type Ide *)
   elsif A = Fun (X): B and                              (* Regel: Type ∀ *)
      haskind(S,X :: TYPE B TYPE) then K = TYPE
   else false
   end
```

D. Der typisierte Lambda–Kalkül höherer Ordnung F_∞

Syntax

```
Kind ::=
      TYPE                                              Basissorte
    | OPER "(" Type-Identifier "::" Kind ")" ":" Kind   Typoperatorsorte
    | "{" Kind "}"
;
Type ::=
      Type-Constant | Type-Identifier
    | Fun "(" Value-Identifier ":" Type ")" ":" Type    Funktionstyp
    | "{" Type "}"
    | Fun "(" Type-Identifier "::" Kind ")" ":" Type    polymorpher Funktionstyp
    | Oper "(" Type-Identifier "::" Kind ")" Type       Typoperator
    | Type "(" Type ")"                                 Applikation
;
Value ::=
      Value-Identifier | Value-Constant
    | fun "(" Value-Identifier ":" Type ")" Value       Abstraktion
    | Value "(" Value ")"                               Applikation
    | "{" Value "}"
    | fun "(" Type-Identifier "::" Kind ")" Value       Typabstraktion
    | Value "(" Type ")"                                Typapplikation
;
```

Typsystem

Kontext:

[Sig empty]
$$\frac{}{\vdash \emptyset \ sig}$$

[Sig ide]
$$\frac{S \vdash A :: \mathbf{TYPE} \quad x \notin Dom(S)}{\vdash S, x : A \ sig}$$

[Sig Ide]
$$\frac{S \vdash K \ kind \quad X \notin Dom(S)}{\vdash S, X :: K \ sig}$$

Sorten:

[Kind basic]
$$\frac{\vdash S \ sig}{S \vdash \mathbf{TYPE} \ kind}$$

[Kind oper]
$$\frac{S, X :: K \vdash L \ kind}{S \vdash \mathbf{OPER}\,(X :: K) : L \ kind}$$

Typen:

[Type builtin]
$$\frac{\vdash S \ sig \quad A \in \{Bool, Int, String\}}{S \vdash A :: \mathbf{TYPE}}$$

[Type →]
$$\frac{S, x : A \vdash B :: \mathbf{TYPE}}{S \vdash \mathbf{Fun}\,(x : A) : B :: \mathbf{TYPE}}$$

[Type Ide]
$$\frac{\vdash S, X :: K, S' \ sig}{S, X :: K, S' \vdash X :: K}$$

[Type ∀]
$$\frac{S, X :: K \vdash A :: \mathbf{TYPE}}{S \vdash \mathbf{Fun}\,(X :: K) : A :: \mathbf{TYPE}}$$

[Type introduction]
$$\frac{S, X :: K \vdash A :: L}{S \vdash \mathbf{Oper}\,(X :: K)\,A :: \mathbf{OPER}\,(X :: K) : L}$$

[Type elimination]
$$\frac{S \vdash A :: \mathbf{OPER}\,(X :: K) : L \quad S \vdash B :: K}{S \vdash A\,(B) :: L}$$

Werte:

[Value ide]
$$\frac{\vdash S, x : A, S' \ sig}{S, x : A, S' \vdash x : A}$$

[Value → introduction]
$$\frac{S, x : A \vdash b : B}{S \vdash \mathbf{fun}\,(x : A)\,b : \mathbf{Fun}\,(x : A) : B}$$

[Value → elimination]
$$\frac{S \vdash f : \mathbf{Fun}\,(x : A) : B \quad S \vdash a : A}{S \vdash f\,(a) : B}$$

[Value ∀ introduction]
$$\frac{S, X :: K \vdash b : B}{S \vdash \mathbf{fun}\,(X :: K)\,b : \mathbf{Fun}\,(X :: K) : B}$$

[Value ∀ elimination]
$$\frac{S \vdash f : \mathbf{Fun}\,(X :: K) : A \quad S \vdash B :: K}{S \vdash f\,(B) : \langle B \to X \rangle A}$$

Typüberprüfungsalgorithmus

```
    let lookupValueIde(S :_Env  x :_Ide  A :_Type) :Bool =
    if S = ∅ then false                                   (* Regel: Sig empty *)
    elsif S = S1,x:A then true                            (* Regel: Sig ide *)
    elsif S = S1,y:B then lookupValueIde(S1 x A)
    elsif S = S1,x :: K then false
    elsif S = S1,Y :: K then lookupValueIde(S1 x A)
    end

let lookupTypeIde(S :_Env  X :_Ide  K :_Kind) :Bool =
    if S = ∅ then false                                   (* Regel: Sig empty*)
    elsif S = S1,X :: K then true                         (* Regel: Sig Ide *)
    elsif S = S1,Y :: L then lookupTypeIde(S1 X K)
    elsif S = S1,X:B then false
    elsif S = S1,y:B then lookupValueIde(S1 X K)
    end

let hastype(S :_Env  a :_Value  A :_Type) :Bool =
    if a = x then lookupValueIde(S x A)                   (* Regel: [Value ide] *)
     elsif a = fun (x:B) c and                            (* Regel: [Value → introduction alg] *)
        haskind(S B TYPE) and
        hastype(S,x:B c C) then A = Fun (x:B): C
    elsif a = f(b) and hastype(S f Fun (x:B): C) and      (* Regel: [Value → elimination] *)
        hastype(S b B) then A = C
    elsif a = fun (X :: K) b and iskind(S K) and          (* Regel: Value ∀ introduction *)
        hastype(S,X :: K b B) then A = Fun (X :: K): B
    elsif a = b (B) and hastype(S b Fun (X :: K): C) and  (* Regel: Value ∀ elimination *)
        haskind(S B K) then A = ⟨B → X⟩C
    else false
    end

let haskind(S :_Env  A :_Type  K :_Kind) :Bool =
    if A ∈ {Bool,Int,String} then K = TYPE                (* Regel: [Type builtin] *)
    elsif A = Fun (x:B): C and haskind(S B TYPE)          (* Regel: [Type → alg] *)
        and haskind(S,x:B C TYPE) then K = TYPE
    elsif A = X then lookupTypeIde(S X K)                 (* Regel: Type Ide *)
    elsif A = Fun (X :: K): B and iskind(S K) and         (* Regel: Type ∀ *)
        haskind(S,X :: K B TYPE) then K = TYPE
    elsif A = Oper (X :: L) B and iskind(S L) and         (* Regel: Type introduction alg *)
        haskind(S,X :: L B M) then K = OPER (X :: L): M
    elsif A = B (C) and haskind(S B OPER (X :: L): M)     (* Regel: Type elimination *)
        and haskind(S C L) then K = M
    else false
    end

let iskind(S :_Env  K :_Kind) :Bool =
    if K = TYPE then true                                 (* Regel: Kind basic *)
    elsif K = OPER (X :: K): L and                        (* Regel: Kind oper alg *)
        iskind(S K) then iskind(S,X :: K L)
    else false
    end
```

E. Der typisierte Lambda–Kalkül $F^{\leq}_{\infty}$ mit Subtypisierung

Syntax

Kind ::=

TYPE	*Basissorte*
| ***OPER*** *"(" Type-Identifier "::" Kind ")" ":" Kind*	*Typoperatorsorte*
| *"{" Kind "}"*	
| ***POWER*** *"(" Type ")"*	*Potenzmengensorte*

;

Type ::=

Type-Constant | *Type-Identifier*	
| ***Fun*** *"(" Value-Identifier ":" Type ")" ":" Type*	*Funktionstyp*
| *"{" Type "}"*	
| ***Fun*** *"(" Type-Identifier "::" Kind ")" ":" Type*	*polymorpher Funktionstyp*
| ***Oper*** *"(" Type-Identifier "::" Kind ")" Type*	*Typoperator*
| *Type "(" Type ")"*	*Applikation*
| ***Top***	*Maximaltyp in* ***TYPE***

;

Value ::=

Value-Identifier | *Value-Constant*	
| ***fun*** *"(" Value-Identifier ":" Type ")" Value*	*Abstraktion*
| *Value "(" Value ")"*	*Applikation*
| *"{" Value "}"*	
| ***fun*** *"(" Type-Identifier "::" Kind ")" Value*	*Typabstraktion*
| *Value "(" Type ")"*	*Typapplikation*

;

Typsystem

Kontext:

[Sig empty]
$$\frac{}{\vdash \emptyset \ sig}$$

[Sig ide]
$$\frac{S \vdash A :: \mathbf{TYPE} \quad x \notin Dom(S)}{\vdash S, x : A \ sig}$$

[Sig Ide]
$$\frac{S \vdash K \ kind \quad X \notin Dom(S)}{\vdash S, X :: K \ sig}$$

Sorten:

[Kind basic]
$$\frac{\vdash S \ sig}{S \vdash \mathbf{TYPE} \ kind}$$

[Kind oper]
$$\frac{S, X :: K \vdash L \ kind}{S \vdash \mathbf{OPER}\,(X :: K) : L \ kind}$$

[Kind power]
$$\frac{S \vdash A :: \mathbf{TYPE}}{S \vdash \mathbf{POWER}(A) \ kind}$$

SubSorten:

[Subkind type type]
$$\frac{\vdash S \ sig}{S \vdash \mathbf{TYPE} <:: \mathbf{TYPE}}$$

[Subkind oper]
$$\frac{S \vdash L_1 <:: K_1 \quad S, X :: L_1 \vdash K_2 <:: L_2}{S \vdash \mathbf{OPER}\,(X :: K_1) : K_2 <:: \mathbf{OPER}\,(X :: L_1) : L_2}$$

[Subkind power type]
$$\frac{S \vdash A :: \mathbf{TYPE}}{S \vdash \mathbf{POWER}(A) <:: \mathbf{TYPE}}$$

[Subkind power power]
$$\frac{S \vdash A <: B}{S \vdash \mathbf{POWER}(A) <:: \mathbf{POWER}(B)}$$

Typen:

[Type top]
$$\frac{\vdash S \ sig}{S \vdash \mathbf{Top} :: \mathbf{TYPE}}$$

[Type subsumption]
$$\frac{S \vdash A :: K \quad S \vdash K <:: L}{S \vdash A :: L}$$

[Type builtin]
$$\frac{\vdash S \ sig \quad A \in \{Bool, Int, String\}}{S \vdash A :: \mathbf{TYPE}}$$

[Type Ide]
$$\frac{\vdash S, X :: K, S' \ sig}{S, X :: K, S' \vdash X :: K}$$

[Type →]
$$\frac{S, x : A \vdash B :: \mathbf{TYPE}}{S \vdash \mathbf{Fun}\,(x : A) : B :: \mathbf{TYPE}}$$

[Type introduction]
$$\frac{S, X :: K \vdash A :: L}{S \vdash \mathbf{Oper}\,(X :: K)\,A :: \mathbf{OPER}\,(X :: K) : L}$$

[Type ∀]
$$\frac{S, X :: K \vdash A :: \mathbf{TYPE}}{S \vdash \mathbf{Fun}\,(X :: K) : A :: \mathbf{TYPE}}$$

[Type elimination]
$$\frac{S \vdash A :: \mathbf{OPER}\,(X :: K) : L \quad S \vdash B :: K}{S \vdash A\,(B) :: \langle B \to X \rangle L}$$

Werte:

[Value ide]

$$\frac{\vdash S, x : A, S' \; sig}{S, x : A, S' \vdash x : A}$$

[Value subsumption]

$$\frac{S \vdash a : A \qquad S \vdash A <: B}{S \vdash a : B}$$

[Value → introduction]

$$\frac{S, x : A \vdash b : B}{S \vdash \mathbf{fun}\,(x : A)\, b : \mathbf{Fun}\,(x : A) : B}$$

[Value → elimination]

$$\frac{S \vdash f : \mathbf{Fun}\,(x : A) : B \qquad S \vdash a : A}{S \vdash f\,(a) : B}$$

[Value ∀ introduction]

$$\frac{S, X :: K \vdash b : B}{S \vdash \mathbf{fun}\,(X :: K)\, b : \mathbf{Fun}\,(X :: K) : B}$$

[Value ∀ elimination]

$$\frac{S \vdash f : \mathbf{Fun}\,(X :: K) : A \qquad S \vdash B :: K}{S \vdash f\,(B) : \langle B \to X \rangle A}$$

Subtypen:

[Subtype top]

$$A :: \mathbf{TYPE} \Longleftrightarrow A <: \mathbf{Top}$$

[Subtype power]

$$A :: \mathbf{POWER}(B) \Longleftrightarrow A <: B$$

[Subtype reflexive]

$$\frac{S \vdash A :: \mathbf{TYPE}}{S \vdash A <: A}$$

[Subtype transitive]

$$\frac{S \vdash A <: B \qquad S \vdash B <: C}{S \vdash A <: C}$$

[Subtype beta]

$$\frac{S \vdash A <: B \qquad S \vdash B \stackrel{\beta}{=} C}{S \vdash A <: C}$$

[Subtype →]

$$\frac{S \vdash B_1 <: A_1 \qquad S \vdash A_2 <: B_2}{S \vdash \mathbf{Fun}\,(x : A_1) : A_2 <: \mathbf{Fun}\,(x : B_1) : B_2}$$

[Subtype ∀]

$$\frac{S \vdash L <:: K \qquad S, X :: L \vdash A <: B}{S \vdash \mathbf{Fun}\,(X :: K) : A <: \mathbf{Fun}\,(X <: L) : B}$$

[Subtype app]

$$\frac{\begin{array}{c} S \vdash A :: \mathbf{OPER}\,(X :: K) : \mathbf{POWER}(D) \\ S \vdash B :: \mathbf{OPER}\,(X :: K) : \mathbf{POWER}(E) \\ S \vdash D <: E \qquad S \vdash C :: K \end{array}}{S \vdash A\,(C) <: B\,(C)}$$

Typüberprüfungsalgorithmus

```
let lookupValueIde(S :_Env  x :_Ide  A :_Type) :Bool =
   if S = ∅ then false                                   (* Regel: Sig empty *)
   elsif S = S1,x:A then true                            (* Regel: Sig ide *)
   elsif S = S1,y:B then lookupValueIde(S1 x A)
   elsif S = S1,x :: K then false
   elsif S = S1,Y :: K then lookupValueIde(S1 x A)
   end

let lookupTypeIde(S :_Env  X :_Ide  K :_Kind) :Bool =
   if S = ∅ then false                                   (* Regel: Sig empty*)
   elsif S = S1,X :: K then true                         (* Regel: Sig Ide *)
   elsif S = S1,Y :: L then lookupTypeIde(S1 X K)
   elsif S = S1,X:B then false
   elsif S = S1,y:B then lookupValueIde(S1 X K)
   end

let hastype(S :_Env  a :_Value  A :_Type) :Bool =
   if a = x then lookupValueIde(S x A)                   (* Regel: [Value ide] *)
   elsif a = fun (x:B) c and                             (* Regel: [Value → introduction alg] *)
      haskind(S B TYPE) and
      hastype(S,x:B c C) then A = Fun (x:B): C
   elsif a = f(b) and hastype(S f F) and                 (* Regel: [Value → elimination alg] *)
      subtype(S F Fun (x:B): C) and hastype(S b B')
      and subtype(S B' B) then A = C
   elsif a = fun (X :: K) b and iskind(S K) and          (* Regel: Value ∀ introduction *)
      hastype(S,X :: K b B) then A = Fun (X :: K): B
   elsif a = f(B) and hastype(S f F) and                 (* Regel: Value ∀ elimination alg *)
      subtype(S F Fun (X :: K): C) and haskind(S B K')
      and subkind(S K' K) then A = ⟨B → X⟩C
   else false
end

let haskind(S :_Env  A :_Type  K :_Kind) :Bool =
   if A ∈ {Bool,Int,String} then K = TYPE                (* Regel: [Type builtin] *)
   elsif A = X then lookupTypeIde(S X K)                 (* Regel: Type Ide *)
   elsif A = Fun (x:B): C and haskind(S B TYPE) and      (* Regel: [Type →] alg *)
      haskind(S,x:B C TYPE) then K = TYPE
   elsif A = Fun (X :: L): B and iskind(S L) and         (* Regel: Type ∀ *)
      haskind(S,X :: L B TYPE) then K = TYPE
   elsif A = Oper (X :: L) B and iskind(S L) and         (* Regel: Type introduction alg *)
      haskind(S,X :: L B M) then K = OPER (X :: L): M
   elsif A = B (C) and haskind(S B OPER (X :: L): M)     (* Regel: Type elimination alg *)
      and haskind(S C L) then K = ⟨C → X⟩M
   else false
   end
```

```
let iskind(S :_Env  K :_Kind) :Bool =
   if K = TYPE then true                                   (* Regel: Kind basic *)
   elsif K = OPER (X :: K):L and                           (* Regel: Kind oper alg *)
      iskind(S K) then iskind(S,X :: K L)
   elsif K = POWER(A) then haskind(S A TYPE)               (* Regel: Kind power *)
   else false
   end

let subtype(S :_Env  A :_Type  B :_Type) :Bool =
   if A = B then true                                      (* Regel: Subtype reflexive *)
   elsif A = Fun (x:C1): C2 and B = Fun (x:D1): D2 then    (* Regel: Subtype → *)
      subtype(S D1 C1) then subtype(S C2 D2)
   elsif A = Fun (X :: K): C and B = Fun (X :: L): D and   (* Regel: Subtype ∀ *)
      subkind(S L K) then subtype(S,X :: L C D)
   elsif A = X and lookupTypeIde(S X POWER(C))             (* Regel: Subtype ide alg *)
      then subtype(S C B)
   elsif A = C(D) and haskind(S D K') and                  (* Regel: Subtype app alg *)
      haskind(S C OPER (X :: K):POWER(E)) and
      subkind(S K' K) then subtype(S ⟨D → X⟩E B)
   else false
   end

let subkind(S :_Env  K :_Kind  L :_Kind) :Bool =
   if K = TYPE and L = TYPE then true                      (* Regel: Subkind type type *)
   elsif K = POWER(A) and L = TYPE then true               (* Regel: Subkind power type *)
   elsif K = POWER(A) and L = POWER(B) then                (* Regel: Subkind power *)
      subtype(S A B)
   elsif K = OPER (X :: K1):K2 and                         (* Regel: Subkind oper *)
      L = OPER (X :: L1):L2 and
      subkind(S L1 K1) then subkind(S,X :: K1 K2 L2)
   else false
   end
```

F. Der typisierte Lambda–Kalkül $F^{\leq}_{\infty}$ ohne Sortennotation

Syntax

Type ::=
    Type-Constant | *Type-Identifier*
  | ***Fun*** *"(" Value-Identifier ":" Type ")" ":" Type* — *Funktionstyp*
  | *"{" Type "}"*
  | ***Fun*** *"(" Type-Identifier "<:" Type ")" ":" Type* — *polymorpher Funktionstyp*
  | ***Oper*** *"(" Type-Identifier "<:" Type ")" Type* — *Typoperator*
  | *Type "(" Type ")"* — *Applikation*
  | ***Top*** — *Maximaltyp in* ***TYPE***
;
Value ::=
    Value-Identifier | *Value-Constant*
  | ***fun*** *"(" Value-Identifier ":" Type ")" Value* — *Abstraktion*
  | *Value "(" Value ")"* — *Applikation*
  | *"{" Value "}"*
  | ***fun*** *"(" Type-Identifier "<:" Type ")" Value* — *Typabstraktion*
  | *Value "(" Type ")"* — *Typapplikation*
;

Typsystem

Kontext:

$$\frac{}{\vdash \emptyset \ sig}\ [\textit{Sig empty}] \qquad \frac{S \vdash A <: \mathbf{Top} \quad x \notin Dom(S)}{\vdash S, x : A \ sig}\ [\textit{Sig ide}] \qquad \frac{S \vdash A \ type \quad X \notin Dom(S)}{\vdash S, X <: A \ sig}\ [\textit{Sig Ide}]$$

Typen:

$$\frac{\vdash S \ sig}{S \vdash \mathbf{Top} \ type}\ [\textit{Type top}] \qquad \frac{\vdash S \ sig \quad A \in \{Bool, Int, String\}}{S \vdash A \ type}\ [\textit{Type builtin}] \qquad \frac{\vdash S, X <: A, S' \ sig}{S, X <: A, S' \vdash X \ type}\ [\textit{Type Ide}]$$

$$\frac{S, x : A \vdash B <: \mathbf{Top}}{S \vdash \mathbf{Fun}\,(x : A) : B \ type}\ [\textit{Type} \rightarrow] \qquad \frac{S, X <: A \vdash B <: \mathbf{Top}}{S \vdash \mathbf{Fun}\,(X <: A) : B \ type}\ [\textit{Type} \forall]$$

$$\frac{S, X <: A \vdash B \ type}{S \vdash \mathbf{Oper}\,(X <: A)\,B \ type}\ [\textit{Type introduction}] \qquad \frac{S \vdash A <: \mathbf{Oper}\,(X <: B)\,C \quad S \vdash D <: B}{S \vdash A\,(D) \ type}\ [\textit{Type elimination}]$$

Werte:

$$\frac{\vdash S, x : A, S' \ sig}{S, x : A, S' \vdash x : A}\ [\textit{Value ide}] \qquad \frac{S \vdash a : A \quad S \vdash A <: B}{S \vdash a : B}\ [\textit{Value subsumption}]$$

$$\frac{S, x : A \vdash b : B}{S \vdash \mathbf{fun}\,(x : A)\,b : \mathbf{Fun}\,(x : A) : B}\ [\textit{Value} \rightarrow \textit{introduction}] \qquad \frac{S \vdash f : \mathbf{Fun}\,(x : A) : B \quad S \vdash a : A}{S \vdash f\,(a) : B}\ [\textit{Value} \rightarrow \textit{elimination}]$$

$$\frac{S, X <: A \vdash b : B}{S \vdash \mathbf{fun}\,(X <: A)\,b : \mathbf{Fun}\,(X <: A) : B}\ [\textit{Value} \forall \textit{ introduction}] \qquad \frac{S \vdash f : \mathbf{Fun}\,(X <: A) : B \quad S \vdash C <: A}{S \vdash f\,(C) : \langle C \rightarrow X \rangle B}\ [\textit{Value} \forall \textit{ elimination}]$$

Subtypen:

$$\frac{S \vdash A \ type}{S \vdash A <: A}\ [\textit{Subtype reflexive}] \qquad \frac{S \vdash A <: B \quad S \vdash B <: C}{S \vdash A <: C}\ [\textit{Subtype transitive}] \qquad \frac{S \vdash A <: B \quad S \vdash B \stackrel{\beta}{=} C}{S \vdash A <: C}\ [\textit{Subtype beta}]$$

[Subtype builtin ok]

$$\frac{\vdash S\ sig \qquad A \in \{Bool, Int, String\}}{S \vdash A <: \mathbf{Top}}$$

[Subtype → ok]

$$\frac{S \vdash \mathbf{Fun}\,(x : A) : B\ type}{S \vdash \mathbf{Fun}\,(x : A) : B <: \mathbf{Top}}$$

[Subtype ∀ ok]

$$\frac{S \vdash \mathbf{Fun}\,(X <: A) : B\ type}{S \vdash \mathbf{Fun}\,(X <: A) : B <: \mathbf{Top}}$$

[Subtype →]

$$\frac{S \vdash B_1 <: A_1 \qquad S \vdash A_2 <: B_2}{S \vdash \mathbf{Fun}\,(x : A_1) : A_2 <: \mathbf{Fun}\,(x : B_1) : B_2}$$

[Subtype ∀]

$$\frac{S \vdash B_1 <: A_1 \qquad S, X <: B_1 \vdash A_2 <: B_2}{S \vdash \mathbf{Fun}\,(X <: A_1) : A_2 <: \mathbf{Fun}\,(X <: B_1) : B_2}$$

[Subtype oper]

$$\frac{S \vdash B_1 <: A_1 \qquad S, X <: B_1 \vdash A_2 <: B_2}{S \vdash \mathbf{Oper}\,(X <: A_1)\,A_2 <: \mathbf{Oper}\,(X <: B_1)\,B_2}$$

[Subtype app]

$$\frac{\begin{array}{c} S \vdash A <: \mathbf{Oper}\,(X <: C)\,D \\ S \vdash B <: \mathbf{Oper}\,(X <: C)\,E \\ S \vdash D <: E \qquad S \vdash F <: C \end{array}}{S \vdash A\,(F) <: B\,(F)}$$

Typüberprüfungsalgorithmus

```
let lookupValueIde(S :_Env  x :_Ide  A :_Type) :Bool =
   if S = ∅ then false                                          (* Regel: Sig empty *)
   elsif S = S1,x:A then true                                   (* Regel: Sig ide *)
   elsif S = S1,y:B then lookupValueIde(S1 x A)
   elsif S = S1,x <: B then false
   elsif S = S1,Y <: B then lookupValueIde(S1 x A)
   end

let lookupTypeIde(S :_Env  X :_Ide) :Bool =
   if S = ∅ then false                                          (* Regel: Sig empty*)
   elsif S = S1,X <: A then true                                (* Regel: Sig Ide *)
   elsif S = S1,Y <: A then lookupTypeIde(S1 X)
   elsif S = S1,X:A then false
   elsif S = S1,y:A then lookupValueIde(S1 X)
   end

let hastype(S :_Env  a :_Value  A :_Type) :Bool =
   if a = x then lookupValueIde(S x A)                          (* Regel: [Value ide] *)
   elsif a = fun(x:B) c and                                     (* Regel: [Value → introduction] *)
      istype(S B) and hastype(S,x:B c C)
      then A = Fun(x:B): C
   elsif a = f(b) and hastype(S f F) and                        (* Regel: [Value → elimination alg] *)
      subtype(S F Fun(x:B): C) and hastype(S b B')
      and subtype(S B' B) then A = C
   elsif a = fun(X <: B) c and istype(S B) and                  (* Regel: Value ∀ introduction *)
      hastype(S,X <: B c C) then A = Fun(X <: B) : C
   elsif a = f(B) and hastype(S f F) and                        (* Regel: Value ∀ elimination alg *)
      subtype(S F Fun(X <: C) : D) and
      subtype(S B C) then A = ⟨B → X⟩D
   else false
   end

let istype(S :_Env  A :_Type) :Bool =
   if A = X then lookupTypeIde(S X)                             (*Regel: Type Ide *)
   elsif A = Top then true                                      (* Regel: Type top *)
   elsif A = Fun(x:B): C and                                    (* Regel: Type → *)
      istype(S B) then istype(S,x:B C)
   elsif A = Fun(X <: B) : C and                                (* Regel: Type ∀ *)
      istype(S B) then istype(S,X <: B C)
   elsif A = Oper(X <: B)C and                                  (* Regel: Type introduction *)
      istype(S B) then istype(S,X <: B C)
   elsif A = B(C) and subtype(S B Oper(X <: D) E)               (* Regel: Type elimination *)
      then subtype(S C D)
   end
```

```
let bound(S :_Env  X :_Ide) :_Type =
   if S = ∅ then undefined                                   (* Regel: Sig empty*)
   elsif S = S1,X <: B then B                                (* Regel: Sig Ide *)
   elsif S = S1,Y <: B then lookupTypeIde(S1 X A)
   elsif S = S1,y:B then bound(S1 X)
   else undefined
   end

let subtype(S :_Env  A :_Type  B :_Type) :Bool =
   if A = B then true                                        (* Regel: Subtype reflexive *)
   elsif A = Fun(x:C1): C2 and B = Fun(x:D1): D2 and          (* Regel: Subtype → *)
      subtype(S D1 C1) then subtype(S C2 D2)
   elsif A ∈ {Bool,Int,String} and B = Top then true         (* Regel: Subtype builtin top *)
   elsif A = Fun(x:C): D and B = Top then true               (* Regel: Subtype → top *)
   elsif A = Fun(x <: C) : D and B = Top then true           (* Regel: Subtype ∀ top *)
   elsif A = X then subtype(S bound(S X) B)                  (* Regel: Subtype ide alg *)
   elsif A = Fun(X <: C1) : C2 and B = Fun(X <: D1) : D2     (* Regel: Subtype ∀ *)
      and subtype(S D1 C1) then subtype(S,X <: D1 C2 D2)
   elsif A = Oper(X <: C1) C2 and B = Oper(X <: D1) D2       (* Regel: Subtype oper *)
      and subtype(S D1 C1) then subtype(S,X <: D1 C2 D2)
   elsif A = C(D) and subtype(S D E) and                     (* Regel: Subtype app alg *)
      subtype(S C Oper(X <: E) F)
      then subtype(S ⟨D → X⟩E B)
   else false
   end
```

G. Algorithmische Typregeln

$\mathbf{F}_1$:

[Value → introduction alg]

$$\frac{S \vdash A :: \mathbf{TYPE} \quad S, x : A \vdash b : B}{S \vdash \mathbf{fun}\,(x : A)\,b : \mathbf{Fun}\,(x : A) : B}$$

[Type → alg]

$$\frac{S \vdash A :: \mathbf{TYPE} \quad S, x : A \vdash B :: \mathbf{TYPE}}{S \vdash \mathbf{Fun}\,(x : A) : B :: \mathbf{TYPE}}$$

$\mathbf{F}_\infty$:

[Kind oper alg]

$$\frac{S \vdash K\ kind \quad S, X :: K \vdash L\ kind}{S \vdash \mathbf{OPER}\,(X :: K) : L\ kind}$$

[Type introduction alg]

$$\frac{S \vdash K\ kind \quad S, X :: K \vdash A :: L}{S \vdash \mathbf{Oper}\,(X :: K)\,A :: \mathbf{OPER}\,(X :: K) : L}$$

$\mathbf{F}^{\leq}_{\infty}$:

[Value → elimination alg]

$$\frac{S \vdash f : F \quad S \vdash F <: \mathbf{Fun}\,(x : A) : B \quad S \vdash a : A' \quad S \vdash A' <: A}{S \vdash f\,(a) : B}$$

[Value ∀ elimination alg]

$$\frac{S \vdash f : F \quad S \vdash F <: \mathbf{Fun}\,(X :: K) : A \quad S \vdash B :: K' \quad S \vdash K' <:: K}{S \vdash f\,(B) : \langle B \to X \rangle A}$$

[Type elimination alg]

$$\frac{S \vdash A :: \mathbf{OPER}\,(X :: K) : L \quad S \vdash B :: K' \quad S \vdash K' <:: K}{S \vdash A\,(B) :: \langle B \to X \rangle L}$$

[Subtype app alg]

$$\frac{S \vdash A :: \mathbf{OPER}\,(X :: K) : \mathbf{POWER}(B) \quad S \vdash \langle D \to X \rangle B <: C \quad S \vdash D :: K' \quad S \vdash K' <:: K}{S \vdash A\,(D) <: C}$$

[Subtype ide alg]

$$\frac{S_1, X :: \mathbf{POWER}(A), S_2 \vdash A <: B}{S_1, X :: \mathbf{POWER}(A), S_2 \vdash X <: B}$$

[Subtype beta alg]

$$\frac{S \vdash betanorm(A) <: betanorm(B)}{S \vdash A <: B}$$

H. TLMIN

Reservierte Schlüsselworte

Die folgenden Bezeichner und Infixsymbole sind reservierte Schlüsselworte und können nicht als benutzerdefinierte (Infix-) Bezeichner in TLMIN Programmen verwendet werden.

and begin case else end exception exit fun if infix let loop of ok raise reraise rec then try tuple tuple_case var when with Exception Fun Let Nok Ok Oper Rec Tuple = <: :

Produktionen

Die Grammatik von TLMIN-Programmen läßt sich durch die folgenden Produktionen beschreiben.

Bindungen

Bindings::=
{ ***Let*** *TypeBinding* | ***Let Rec*** *TypeRecBinding* | ***let*** *ValueBinding* |
let rec *ValueRecBinding* | *":" Type* | ***var*** *Value* | *Value* };
TypeBinding::=
TypeIde Parameters [*"<:" Type*] *"=" Type;*
TypeRecBinding::=
TypeIde Parameters "<:" Type "=" Type { ***and*** *TypeRecBinding* };
ValueBinding::=
[***var***] *ValueIde Parameters* [*":" Type*] *"=" Value;*
ValueRecBinding::=
[***var***] *ValueIde Parameters ":" Type "=" Value* { ***and*** *ValueRecBinding* };

Werte

Value::=
fun *"(" Signatures ")"* [*":" Type*] *Value* |
$Value_1$ { *colonInfix* $Value_1$ };
$Value_1$*::=*
$Value_2$ { *infix* $Value_2$ };
$Value_2$*::=*
$Value_3$ { *"(" Bindings ")"* | *"." FieldIde* | *"[" Value "]"* | ***of*** *Bindings* ***end*** };

Value$_3$ ::=
    ValueIde |
    ok |
    int | *char* | *string* | *real* | *longreal* |
    tuple *Bindings* ***end*** |
    tuple_case *CaseIde* ***of*** *Type* [***with*** *Bindings*] ***end*** |
    record *Bindings* ***end*** |
    extend *Value* [***with*** *Bindings*] ***end*** |
    array *Bindings* ***end*** |
    exception *Value* [***with*** *Signatures*] ***end*** |
    begin *Bindings* ***end*** |
    loop *Bindings* ***end*** |
    if *Value* ***then*** *Bindings* [***else*** *Bindings*] ***end*** |
    case [***of***] *Value* { ***when*** *CaseIdeList* [***with*** *ValueIde*] ***then*** *Bindings* }
        [***else*** *Bindings*] ***end*** |
    exit |
    try *Bindings* { ***when*** *Value* [***with*** *ValueIde*] ***then*** *Bindings* }
        [***else*** *Bindings*] ***end*** |
    raise *Value* [***with*** *Bindings*] ***end*** |
    reraise;

Signaturen

Signatures::=
    { *TypeSignatures* | ***Rec*** *TypeRecSignatures* | ***Let*** *TypeBinding* |
      Let Rec *TypeRecBinding* | *ValueSignatures* };
TypeSignatures::=
    [*TypeIdeList Parameters*] "<:" *Type*;
TypeRecSignatures::=
    TypeIdeList Parameters "<:" *Type* "<:" *Type*;
ValueSignatures::=
    [***var***] [*ValueIdeList Parameters*] ":" *Type*;
Parameters::=
    { "(" *Signatures* ")" };

Typen

Type::=
    Oper "(" *Signatures* ")" *Type* |
    Fun "(" *Signatures* ")" ":" *Type* |
    Type$_1$ { *colonInfix Type*$_1$ };
Type$_1$ ::=
    Type$_2$ { *infix Type*$_2$ };
Type$_2$ ::=
    Type$_3$ { "(" { *Type* } ")" };
Type$_3$ ::=
    "{" *Type* "}" |
    { *ValueOrTypeIde* "." } *TypeIde* |
    Ok | ***Nok*** |
    Tuple *Signatures* { ***case*** *CaseIdeList* [***with*** *Signatures*] } ***end*** |
    Record *Signatures* ***end*** |
    Exception *Signatures* ***end***;

Bezeichner

ValueIdeList, TypeIdeList, CaseIdeList::=
Ide { "," Ide };
Ide, ValueIde, TypeIde, FieldIde, CaseIde, ValueOrTypeIde::=
identifier | infix | colonInfix | "{" Ide "}";

Typsystem

Die nachfolgenden Typregeln formalisieren die statische Semantik von TLMIN. Zunächst eine Übersicht der verwendeten Notationen:

S, S'	Die Signaturen S gefolgt von den Signaturen S'
D, D'	Die Bindungen D gefolgt von den Bindungen D'
$i = newOid()$	Eine neue einzigartige Objektidentität i
$x \in Dom(S)$	Die Wertvariable x ist in S definiert
$X \in Dom(S)$	Die Typvariable X ist in S definiert
$S \vdash J$	In dem statischen Kontext S ist die Aussage J wahr
$S\ sig$	Die Signaturen S sind wohlgeformt
$S\ rcdsig$	Die Signaturen S sind wohlgeformte Recordsignaturen
$A\ type$	Der Typ A is wohlgeformt
$A \overset{Sig}{::} S$	Der Typ A hat Signaturen S
$p \overset{Sig}{::} S$	Der Pfad p identifiziert Signaturen S
$S <:: S'$	S sind Subsignaturen von S'
$S \overset{Tup}{<::} S'$	S sind Tupelsubsignaturen von S'
$S \overset{Rcd}{<::} S'$	S sind Recordsubsignaturen von S'
$A <: B$	A ist ein Subtyp von B
$A \overset{Rec}{<:} B$	A ist ein rekursiver Subtyp von B
$D :: S$	Die Bindungen D haben Signaturen S
$a : A$	Der Wert a hat Typ A
seqtype(S)	Der Typ der Signatursequenz S
$\langle B \rightarrow X \rangle A$	Substitution freier Variablen X in A durch B
$\langle Y \rightarrow X \rangle S$	Substitution freier Variablen X in S durch Y
$\langle D \Rightarrow S \rangle A$	Iterierte Substitution von S in A gemäß D
$\langle S'' \Rightarrow S' \rangle A$	Iterierte Substitution von S' in A gemäß S''
qualify$(T; p; S)$	Substitution freier Variablen $X \in Dom(S)$ durch $p.X$
$A \downarrow X$	Der Typ A ist kontraktiv in der Variablen X

Wohlgeformte Signaturen

[Sig init]

$$\frac{BoolOid, IntOid, StringOid, ArrOid = newOid()}{\vdash Bool^{BoolOid} <: \mathbf{Ok}, Int^{IntOid} <: \mathbf{Ok}, String^{StringOid} <: \mathbf{Ok}, Arr^{ArrOid} <: \mathbf{Oper}(E <: \mathbf{Ok})\mathbf{Ok}\ sig}$$

[Sig ide]

$$\frac{S \vdash A <: \mathbf{Ok} \quad i = newOid()}{\vdash S, x^i : A\ sig}$$

[Sig Ide]

$$\frac{S \vdash A\ type \quad i = newOid()}{\vdash S, X^i <: A\ sig}$$

[Sig Ide Let]

$$\frac{S \vdash A\ type \quad i = newOid()}{\vdash S, X^i = A\ sig}$$

[Rcdsig]

$$\frac{\vdash S, S'\ sig \quad (S' = S_1, S_2 \wedge X \in Dom(S_1) \Rightarrow X \notin Dom(S_2))}{S \vdash S'\ rcdsig}$$

[Sig Rec]

$$\frac{\begin{array}{c} S \vdash B_i\ type \\ S, X_1 <: B_1, \ldots, X_n <: B_n \vdash A_i\ type \\ S, X_1 \overset{Rec}{<:} A_1, \ldots, X_n \overset{Rec}{<:} A_n \vdash A_i \downarrow X_i \\ S, X_1 \overset{Rec}{<:} A_1, \ldots, X_n \overset{Rec}{<:} A_n \vdash A_i <: B_i \\ i = 1 \ldots n \quad 1 \leq n \end{array}}{\vdash S, X_1 \overset{Rec}{<:} A_1, \ldots, X_n \overset{Rec}{<:} A_n\ sig}$$

[Sig Rec Let]

$$\frac{\begin{array}{c} S \vdash B_i\ type \\ S, X_1 = B_1, \ldots, X_n = B_n \vdash A_i\ type \\ S, X_1 \overset{Rec}{=} A_1, \ldots, X_n \overset{Rec}{=} A_n \vdash A_i \downarrow X_i \\ S, X_1 \overset{Rec}{=} A_1, \ldots, X_n \overset{Rec}{=} A_n \vdash A_i <: B_i \\ i = 1 \ldots n \quad 1 \leq n \end{array}}{\vdash S, X_1 \overset{Rec}{=} A_1, \ldots, X_n \overset{Rec}{=} A_n\ sig}$$

Wohlgeformte Typen

[Type Ok]

$$\frac{\vdash S\ sig}{S \vdash \mathbf{Ok}\ type}$$

[Type Nok]

$$\frac{\vdash S\ sig}{S \vdash \mathbf{Nok}\ type}$$

[Type Ide]

$$\frac{\vdash S, X <: A, S'\ sig \quad X \notin Dom(S')}{S, X <: A, S' \vdash X\ type}$$

[Type Ide Let]

$$\frac{\vdash S, X = A, S'\ sig \quad X \notin Dom(S')}{S, X = A, S' \vdash X\ type}$$

[Type Dot]

$$\frac{S \vdash p \overset{Sig}{::} S' \quad S, S' \vdash X\ type \quad X \in Dom(S')}{S \vdash p.X\ type}$$

[Type Fun]

$$\frac{\vdash S, S'\ sig \quad S, S' \vdash B <: \mathbf{Ok}}{S \vdash \mathbf{Fun}(S') : B\ type}$$

[Type Tup]

$$\frac{1 \leq n \quad z_i \neq z_j\ \ i \neq j \quad \vdash S, S'\ sig \quad \vdash S, S', S_i\ sig \quad i = 1 \ldots n}{S \vdash \mathbf{Tup}(S'; \mathbf{Case}(z_1, S_1) \ldots \mathbf{Case}(z_n, S_n))\ type}$$

[Type Rcd]

$$\frac{S \vdash S'\ rcdsig}{S \vdash \mathbf{Rcd}(S')\ type}$$

[Type Exc]

$$\frac{\vdash S, S'\ sig}{S \vdash \mathbf{Exc}(S')\ type}$$

[Type Var]

$$\frac{S \vdash A <: \mathbf{Ok}}{S \vdash \mathbf{Var}(A)\ type}$$

[Type Oper]

$$\frac{\vdash S, S'\ sig \quad S, S' \vdash B\ type}{S \vdash \mathbf{Oper}(S')B\ type}$$

[Type Apply]

$$\frac{S \vdash A <: \mathbf{Oper}(S')B \quad S \vdash D :: S'}{S \vdash A(D)\ type}$$

Wert- und Typsignaturen

[Select Val]

$$\frac{S \vdash a : A \quad S \vdash A \overset{Sig}{::} S'}{S \vdash a \overset{Sig}{::} S'}$$

[Select Tup]

$$\frac{S \vdash A <: \mathbf{Tup}(S'; C)}{S \vdash A \overset{Sig}{::} S'}$$

[Select Rcd]

$$\frac{S \vdash A <: \mathbf{Rcd}(S')}{S \vdash A \overset{Sig}{::} S'}$$

Subsignaturen

[Subsig reflexive]
$$\frac{\vdash S, S'\ sig}{S \vdash S' <:: S'}$$

[Subsig ide]
$$\frac{S \vdash A <: B \quad S, x : A \vdash S' <:: \langle x \to y \rangle S''}{S \vdash x : A, S' <:: y : B, S''}$$

[Subsig Ide]
$$\frac{S \vdash A <: B \quad S, X <: A \vdash S' <:: \langle X \to Y \rangle S''}{S \vdash X <: A, S' <:: Y <: B, S''}$$

[Subsig Ide Let]
$$\frac{S \vdash A <: B \quad S, X = A \vdash S' <:: \langle X \to Y \rangle S''}{S \vdash X = A, S' <:: Y <: B, S''}$$

[Subsig Ide Let2]
$$\frac{S \vdash A \stackrel{\beta}{=} B \quad S, X = A \vdash S' <:: \langle X \to Y \rangle S''}{S \vdash X = A, S' <:: Y = B, S''}$$

[Subsig Tup]
$$\frac{\vdash S, S_1, S_2\ sig \quad S \vdash S_1 <:: S_1'}{S \vdash S_1, S_2 \stackrel{Tup}{<::} S_1'}$$

[Subsig Rcd empty]
$$\frac{S \vdash S'\ rcdsig}{S \vdash S' \stackrel{Rcd}{<::} \oslash}$$

[Subsig Rcd]
$$\frac{S \vdash S_1, S_2, S_3\ rcdsig \quad S \vdash S_1, S_2, S_3 \stackrel{Rcd}{<::} S_1' \quad S, S_1' \vdash S_2 \stackrel{Tup}{<::} S_2'}{S \vdash S_1, S_2, S_3 \stackrel{Rcd}{<::} S_1', S_2'}$$

[Subsig Rec]
$$\frac{\begin{array}{c} S, X_1 \stackrel{Rec}{<:} A_1, \ldots, X_n \stackrel{Rec}{<:} A_n \vdash S' <:: \langle X_n \to Y_n \rangle \langle X_1 \to Y_1 \rangle S'' \ldots \\ S, X_1 \stackrel{Rec}{<:} A_1, \ldots, X_n \stackrel{Rec}{<:} A_n \vdash A_i <: \langle X_n \to Y_n \rangle \langle X_1 \to Y_1 \rangle B_i \ldots \qquad i = 1 \ldots n \quad 1 \leq n \end{array}}{S \vdash X_1 \stackrel{Rec}{<:} A_1, \ldots, X_n \stackrel{Rec}{<:} A_n, S' <:: Y_1 \stackrel{Rec}{<:} B_1, \ldots, Y_n \stackrel{Rec}{<:} B_n, S''}$$

[Subsig Rec Let]
$$\frac{\begin{array}{c} S, X_1 \stackrel{Rec}{=} A_1, \ldots, X_n \stackrel{Rec}{=} A_n \vdash S' <:: \langle X_n \to Y_n \rangle \langle X_1 \to Y_1 \rangle S'' \ldots \\ S, X_1 \stackrel{Rec}{=} A_1, \ldots, X_n \stackrel{Rec}{=} A_n \vdash A_i <: \langle X_n \to Y_n \rangle \langle X_1 \to Y_1 \rangle B_i \ldots \qquad i = 1 \ldots n \quad 1 \leq n \end{array}}{S \vdash X_1 \stackrel{Rec}{=} A_1, \ldots, X_n \stackrel{Rec}{=} A_n, S' <:: Y_1 \stackrel{Rec}{<:} B_1, \ldots, Y_n \stackrel{Rec}{<:} B_n, S''}$$

[Subsig Rec Let2]
$$\frac{\begin{array}{c} S, X_1 \stackrel{Rec}{=} A_1, \ldots, X_n \stackrel{Rec}{=} A_n \vdash S' <:: \langle X_n \to Y_n \rangle \langle X_1 \to Y_1 \rangle S'' \ldots \\ S, X_1 \stackrel{Rec}{=} A_1, \ldots, X_n \stackrel{Rec}{=} A_n \vdash A_i = \langle X_n \to Y_n \rangle \langle X_1 \to Y_1 \rangle B_i \ldots \qquad i = 1 \ldots n \quad 1 \leq n \end{array}}{S \vdash X_1 \stackrel{Rec}{=} A_1, \ldots, X_n \stackrel{Rec}{=} A_n, S' <:: Y_1 \stackrel{Rec}{=} B_1, \ldots, Y_n \stackrel{Rec}{=} B_n, S''}$$

Subtypen

[Subtype reflexive]
$$\frac{S \vdash A\ type}{S \vdash A <: A}$$

[Subtype transitive]
$$\frac{S \vdash A <: A' \quad S \vdash A' <: A''}{S \vdash A <: A''}$$

[Subtype beta]
$$\frac{S \vdash A <: B \quad S \vdash B \stackrel{\beta}{=} C}{S \vdash A <: C}$$

[Subtype Nok]
$$\frac{S \vdash A <: \mathbf{Ok}}{S \vdash \mathbf{Nok} <: A}$$

[Subtype Ok Fun]
$$\frac{S \vdash \mathbf{Fun}(S') : B \ type}{S \vdash \mathbf{Fun}(S') : B <: \mathbf{Ok}}$$

[Subtype Ok Tup]
$$\frac{S \vdash \mathbf{Tup}(S'; C) \ type}{S \vdash \mathbf{Tup}(S'; C) <: \mathbf{Ok}}$$

[Subtype Ok Rcd]
$$\frac{S \vdash \mathbf{Rcd}(S') \ type}{S \vdash \mathbf{Rcd}(S') <: \mathbf{Ok}}$$

[Subtype Ok Exc]
$$\frac{S \vdash \mathbf{Exc}(S') \ type}{S \vdash \mathbf{Exc}(S') <: \mathbf{Ok}}$$

[Subtype Ok Var]
$$\frac{S \vdash \mathbf{Var}(A) \ type}{S \vdash \mathbf{Var}(A) <: \mathbf{Ok}}$$

[Subtype Ide]
$$\frac{\vdash S, X <: A, S' \ sig \qquad X \notin Dom(S')}{S, X <: A, S' \vdash X <: A}$$

[Subtype Ide Let]
$$\frac{\vdash S, X = A, S' \ sig \qquad X \notin Dom(S')}{S, X = A, S' \vdash X = A}$$

[Subtype Dot]
$$\frac{S \vdash p \overset{Sig}{::} S', X <: A, S'' \qquad X \notin Dom(S'')}{S \vdash p.X <: \mathrm{qualify}(A; p; S')}$$

[Subtype Fun]
$$\frac{S \vdash S'' <:: S' \qquad S, S'' \vdash \langle S'' \Rightarrow S' \rangle A <: B}{S \vdash \mathbf{Fun}(S') : A <: \mathbf{Fun}(S'') : B}$$

[Subtype Tup]
$$\frac{\begin{array}{c} S \vdash S_0, S_i \overset{Tup}{<::} S'_0, S'_i \quad i = 1 \ldots n \quad 1 \leq n \\ S \vdash S'_0, S''_j \ sig \quad j = 1 \ldots m \quad 1 \leq m \end{array}}{\begin{array}{ll} S \vdash & \mathbf{Tup}(S_0; \mathbf{Case}(z_1, S_1) \ldots \mathbf{Case}(z_n, S_n)) <: \\ & \mathbf{Tup}(S'_0; \mathbf{Case}(z_1, S'_1) \ldots \mathbf{Case}(z_n, S'_n) \, \mathbf{Case}(z'_1, S''_1) \ldots \mathbf{Case}(z'_m, S''_m)) \end{array}}$$

[Subtype Rcd]
$$\frac{S \vdash S' \overset{Rcd}{<::} S''}{S \vdash \mathbf{Rcd}(S') <: \mathbf{Rcd}(S'')}$$

[Subtype Exc]
$$\frac{S \vdash S' \overset{Tup}{<::} S''}{S \vdash \mathbf{Exc}(S') <: \mathbf{Exc}(S'')}$$

[Subtype Var]
$$\frac{S \vdash A \ type}{S \vdash \mathbf{Var}(A) <: A}$$

[Subtype Oper]
$$\frac{S \vdash S'' <:: S' \qquad S, S'' \vdash \langle S'' \Rightarrow S' \rangle A <: B}{S \vdash \mathbf{Oper}(S')A <: \mathbf{Oper}(S'')B}$$

[Subtype Apply]
$$\frac{\begin{array}{c} S \vdash A <: \mathbf{Oper}(S')C' \\ S \vdash B <: \mathbf{Oper}(S')C'' \\ S, S' \vdash C' <: C'' \qquad S \vdash D :: S' \end{array}}{S \vdash A(D) <: B(D)}$$

Signaturen von Bindungen

[Bind empty]
$$\frac{\vdash S \ sig}{S \vdash \oslash :: \oslash}$$

[Bind ide]
$$\frac{S \vdash a : A \qquad S, x : A \vdash D :: S'}{S \vdash x : A = a, D :: x : A, S'}$$

[Bind Ide]
$$\frac{S \vdash A \ type \qquad S, X = A \vdash D :: S'}{S \vdash X = A, D :: X = A, S'}$$

[Bind Ide restrict]
$$\frac{S \vdash A <: B \qquad S, X = A \vdash D :: S'}{S \vdash X <: B = A, D :: X = A, S'}$$

[Bind rec]
$$\frac{\begin{array}{c} S, x_1 : A_1, \ldots, x_n : A_n \vdash a_i : A_i \\ S, x_1 : A_1, \ldots, x_n : A_n \vdash D :: S' \\ i = 1 \ldots n \quad 1 \leq n \end{array}}{S \vdash x_1 : A_1 \overset{rec}{=} a_1, \ldots, x_n : A_n \overset{rec}{=} a_n, D :: x_1 : A_1, \ldots, x_n : A_n, S}$$

[Bind Rec]

$$\frac{\vdash S, X_1 \overset{Rec}{=} A_1, \ldots, X_n \overset{Rec}{=} A_n \ sig \quad S, X_1 \overset{Rec}{=} A_1, \ldots, X_n \overset{Rec}{=} A_n \vdash D :: S'}{S \vdash X_1 \overset{Rec}{=} A_1, \ldots, X_n \overset{Rec}{=} A_n, D :: X_1 \overset{Rec}{=} A_1, \ldots, X_n \overset{Rec}{=} A_n, S'}$$

Typen von Werten

[Value subsumption]

$$\frac{S \vdash a : A \quad S \vdash A <: B}{S \vdash a : B}$$

[Value ok]

$$\frac{\vdash S \ sig}{S \vdash \mathbf{ok} : \mathbf{Ok}}$$

[Value ide]

$$\frac{\vdash S, x : A, S' \ sig \quad x \notin Dom(S')}{S, x : A, S' \vdash x : A}$$

[Value fun]

$$\frac{S, S' \vdash a : A}{S \vdash \mathbf{fun}(S')a : \mathbf{Fun}(S') : A}$$

[Value apply]

$$\frac{S \vdash a : \mathbf{Fun}(S') : A \quad S \vdash D :: S'}{S \vdash a(D) : \langle D \Rightarrow S' \rangle A}$$

[Value array]

$$\frac{S \vdash D :: x_1 : A \ldots x_n : A \quad 1 \leq n}{S \vdash \mathbf{arr}(D) : Arr^{ArrOid}(A)}$$

[Value array empty]

$$\frac{\vdash S \ sig}{S \vdash \mathbf{arr}() : Arr^{ArrOid}(\mathbf{Nok})}$$

[Value index]

$$\frac{S \vdash a : Arr^{ArrOid}(A) \quad S \vdash b : Int^{IntOid}}{S \vdash a[b] : A}$$

[Value tuple]

$$\frac{S \vdash D :: S'}{S \vdash \mathbf{tup}(D) : \mathbf{Tup}(S'; \mathbf{Case}(?, \oslash))}$$

[Value tuple variant]

$$\frac{S \vdash A = \mathbf{Tup}(S'; \mathbf{Case}(z_1, S_1) \ldots \mathbf{Case}(z_n, S_n)) \quad S \vdash D :: S', S_i \quad 1 \leq i \leq n}{S \vdash \mathbf{tup}(D; z_i; A) : A}$$

[Value tuple dot]

$$\frac{S \vdash a : \mathbf{Tup}(S', x : A, S''; C) \quad x \notin Dom(S'')}{S \vdash a.x : \mathrm{qualify}(A; a; S')}$$

[Value record]

$$\frac{S \vdash D :: S' \quad S \vdash S' \ rcdsig}{S \vdash \mathbf{rcd}(D) : \mathbf{Rcd}(S')}$$

[Value record dot]

$$\frac{S \vdash a : \mathbf{Rcd}(S', x : A, S'')}{S \vdash a.x : \mathrm{qualify}(A; a; S')}$$

[Value extend]

$$\frac{S \vdash a : \mathbf{Rcd}(S') \quad S \vdash D :: S'' \quad S \vdash S', S'' \ rcdsig}{S \vdash \mathbf{extend}(a, D) : \mathbf{Rcd}(S', S'')}$$

[Value exception]

$$\frac{S \vdash a : String^{StringOid} \quad S \vdash D :: S'}{S \vdash \mathbf{exc}(a, D) : \mathbf{Exc}(S')}$$

[Value seq]

$$\frac{S \vdash D :: S'}{S \vdash \mathbf{seq}(D) : \mathrm{seqtype}(S')}$$

[Value if]

$$\frac{S \vdash a : Bool^{BoolOid} \quad S \vdash b : B \quad S \vdash b' : B}{S \vdash \mathbf{if}(a, b, b') : B}$$

[Value loop]

$$\frac{S \vdash a : \mathbf{Ok}}{S \vdash \mathbf{loop}(a) : \mathbf{Ok}}$$

[Value exit]

$$\frac{\vdash S \ sig}{S \vdash \mathbf{exit} : \mathbf{Nok}}$$

[Value try]

$$\frac{\begin{array}{c} S \vdash a : B \\ S \vdash b : B \\ S \vdash a_i : \mathbf{Exc}(S_i) \quad i = 1 \ldots n \quad 0 \leq n \\ S, x_i : \mathbf{Tup}(S_i; \mathbf{Case}(?, \oslash)) \vdash b_i : B \quad i = 1 \ldots n \end{array}}{S \vdash \mathbf{try}(a, (a_1, x_1, b_1) \ldots (a_n, x_n, b_n), b) : B}$$

[Value raise]

$$\frac{S \vdash a : \mathbf{Exc}(S') \quad S \vdash D :: S'}{S \vdash \mathbf{raise}(a, D) : \mathbf{Nok}}$$

[Value reraise]

$$\frac{\vdash S \ sig}{S \vdash \mathbf{reraise} : \mathbf{Nok}}$$

[Value case]

$$\frac{\begin{array}{c} S \vdash a : \mathbf{Tup}(S'; \mathbf{Case}(z'_1, S_1) \ldots \mathbf{Case}(z'_m, S_m)) \\ S \vdash b : B \\ S, y_i : \mathbf{Tup}(S', S''; \mathbf{Case}(?, \emptyset)) \vdash b_i : B \quad 0 \leq n \quad i = 1 \ldots n \\ S, S' \vdash S'' <:: S_{z_{ij}} \quad j = 1 \ldots k_i \\ \bigcup_{i=1}^{n} \bigcup_{j=1}^{k_i} z_{ij} \subseteq \bigcup_{l=1}^{m} z'_l \quad z_{ij} \neq z_{i'j'} \ i \neq i', j \neq j' \end{array}}{S \vdash \mathbf{case}(a, (z_{11} \ldots z_{1k_1}, y_1, b_1) \ldots (z_{n1} \ldots z_{nk_n}, y_n, b_n), b) : B}$$

[Value case exhaustive]

$$\frac{\begin{array}{c} S \vdash a : \mathbf{Tup}(S'; \mathbf{Case}(z'_1, S_1) \ldots \mathbf{Case}(z'_m, S_m)) \\ S, y_i : \mathbf{Tup}(S', S''; \mathbf{Case}(?, \emptyset)) \vdash b_i : B \quad 1 \leq n \quad i = 1 \ldots n \\ S, S' \vdash S'' <:: S_{z_{ij}} \quad j = 1 \ldots k_i \\ \bigcup_{i=1}^{n} \bigcup_{j=1}^{k_i} z_{ij} = \bigcup_{l=1}^{m} z'_l \quad z_{ij} \neq z_{i'j'} \ i \neq i', j \neq j' \end{array}}{S \vdash \mathbf{case}(a, (z_{11} \ldots z_{1k_1}, y_1, b_1) \ldots (z_{n1} \ldots z_{nk_n}, y_n, b_n)) : B}$$

Typen von Sequenzen

Die Funktion seqtype() berechnet den Typ einer Blockstruktur.

$\text{seqtype}(\oslash) = \mathbf{Ok}$

$\text{seqtype}(S, x : A) = A$

$\text{seqtype}(S, X = A) = \text{seqtype}(S)$

Substitution in Typen

Die Substitution von freien Bezeichnern X^i in einem Typ B durch einen anderen Typ A, kurz: $\langle A \to X^i \rangle B$ stellt eine Basisfunktion dar, die für die Berechnung von Funktionsrückgabetypen, den Subsignaturtest und die Qualifizierung von Typen erforderlich ist. Sie ist für jede Typvariante explizit definiert.

$\langle A \to X^i \rangle \mathbf{Ok} = \mathbf{Ok}$

$\langle A \to X^i \rangle \mathbf{Nok} = \mathbf{Nok}$

$\langle A \to X^i \rangle Y^j = Y^j \ , \ if \ i \neq j$

$\langle A \to X^i \rangle Y^j = A \ , \ if \ i = j$

$\langle A \to X^i \rangle Y^j.Z = Y^j.Z \ , \ if \ i \neq j$

$\langle A \to X^i \rangle Y^j.Z = ide(A).Z \ , \ if \ i = j$

$\langle A \to X^i \rangle \mathbf{Fun}(S) : B = \mathbf{Fun}(\langle A \to X^i \rangle S) : \langle A \to X^i \rangle B$

$\langle A \to X^i \rangle \mathbf{Tup}(S; \mathbf{Case}(z_1, S_1) \ldots \mathbf{Case}(z_n, S_n)) =$
$\mathbf{Tup}(\langle A \to X^i \rangle S; \mathbf{Case}(z_1, \langle A \to X^i \rangle S_1) \ldots \mathbf{Case}(z_n, \langle A \to X^i \rangle S_n))$

$\langle A \to X^i \rangle \mathbf{Rcd}(S) = \mathbf{Rcd}(\langle A \to X^i \rangle S)$

$\langle A \to X^i \rangle \mathbf{Exc}(S) = \mathbf{Exc}(\langle A \to X^i \rangle S)$

$\langle A \to X^i \rangle \mathbf{Var}(B) = \mathbf{Var}(\langle A \to X^i \rangle B)$

$\langle A \to X^i \rangle \mathbf{Oper}(S)B = \mathbf{Oper}(\langle A \to X^i \rangle S)\langle A \to X^i \rangle B$

$\langle A \to X^i \rangle B(D) = \langle A \to X^i \rangle B(\langle A \to X^i \rangle D)$

Da es in TL wertabhängige Typen gibt, ist die folgende Substitutionsoperation für Wertbezeichner definiert, die in Typpfaden auftreten:

$\langle a \to x^i \rangle y^j.Z = y^j.Z \ , \ if \ i \neq j$

$\langle a \to x^i \rangle y^j.Z = ide\{a\}.Z \ , \ if \ i = j$

Die Funktion $ide\{p\}$ verhindert während der Substitution die Konstruktion syntaktisch unerlaubter Typen wie **tuple Let** $T \ = \ Int$ **end**.T oder **Tuple** $T \ <:$ **Ok end**.T. Sie ist auf Werten und Typen folgendermaßen definiert:

$ide\{x\} = x$, $ide\{X\} = X$,

$ide\{p.x\} = ide\{p\}.x$, $ide\{p.X\} = ide\{p\}.X$,

$ide\{a\} = ?^i$, mit $?^i : A$, A ist der Typ von a und $i = newOid()$

$ide\{A\} = ?^i$ mit $?^i = A$ und $i = newOid()$

Im Typüberprüfungsalgorithmus ist die Substitutionsfunktion im Modul *tlSubstitute* realisiert. Zur Substitution von Typen dient die Funktion *substituteType*, zum Ersetzen in einer Signatur die Funktion *substituteSignature* und zum Qualifizieren die Funktion *qualifiedSubstitute*. Der Ausdruck $\langle S'' \Rightarrow S' \rangle A$ in den Typregeln *[Subtype Fun]* und *[Subtype Oper]* beschreibt die wiederholte Substitution von Bezeichnern aus S' durch Bezeichner aus S'' in dem Typ A:

$\langle y^j : B'', S'' \Rightarrow x^i : B', S' \rangle A = \langle S'' \Rightarrow S' \rangle \langle y^j \rightarrow x^i \rangle A$

$\langle Y^j = B'', S'' \Rightarrow X^i = B', S' \rangle A = \langle S'' \Rightarrow S' \rangle \langle Y^j \rightarrow X^i \rangle A$

$\langle Y^j = B'', S'' \Rightarrow X^i <: B', S' \rangle A = \langle S'' \Rightarrow S' \rangle \langle Y^j \rightarrow X^i \rangle A$

$\langle Y^j <: B'', S'' \Rightarrow X^i <: B', S' \rangle A = \langle S'' \Rightarrow S' \rangle \langle Y^j \rightarrow X^i \rangle A$

Substitution in Rückgabetypen

Die Typregeln für die Instantiierung polymorpher Funktionen *[Value apply]* und Typoperatorapplikation *[Subtype apply alg]* verwenden den Ausdruck $\langle D \Rightarrow S \rangle A$ zur Beschreibung der Substitution freier Variablen X^i bzw. $x^i \in Dom(S)$ innerhalb eines Rückgabetyps A durch Typen oder Wertbezeichner aus D.

$\langle \oslash \Rightarrow \oslash \rangle A = A$

$\langle y = b, D \Rightarrow x^i : B, S \rangle A = \langle D \Rightarrow S \rangle \langle b \rightarrow x^i \rangle A$

$\langle y : B = b, D \Rightarrow x^i : B, S \rangle A = \langle D \Rightarrow S \rangle \langle b \rightarrow x^i \rangle A$

$\langle Y = B, D \Rightarrow X^i = B, S \rangle A = \langle D \Rightarrow S \rangle \langle B \rightarrow X^i \rangle A$

$\langle Y <: C = B, D \Rightarrow X^i = B, S \rangle A = \langle D \Rightarrow S \rangle \langle B \rightarrow X^i \rangle A$

Substitution in Signaturen

Der Ausdruck $\langle X^i \rightarrow Y^j \rangle S$ steht für die Substitution eines freien Bezeichners Y^j durch X^i in den Typen einer Signatur S. Diese Substitution ist beim Subsignaturtest erforderlich.

$\langle X^i \rightarrow Y^j \rangle (S, z^k : A) = \langle X^i \rightarrow Y^j \rangle S, z^l : \langle X \rightarrow Y \rangle A$ mit $l = newOid()$

$\langle X^i \rightarrow Y^j \rangle (S, Z^k = A) = \langle X^i \rightarrow Y^j \rangle S, Z^l = \langle X \rightarrow Y \rangle A$ mit $l = newOid()$

$\langle X^i \rightarrow Y^j \rangle (S, Z^k <: A) = \langle X^i \rightarrow Y^j \rangle S, Z^l <: \langle X \rightarrow Y \rangle A$ mit $l = newOid()$

Qualifizierte Typvariablen

Beim Feldzugriff auf Wert- oder Typkomponenten eines Tupels oder Records in den Typregeln *[Subtype Dot]*, *[Value tuple dot]* oder *[Value record dot]* müssen alle lokal extrahierten Typinformationen durch einen Pfad qualifiziert werden, um abstrakte Datentypen unterscheiden zu können.

Ein qualifizierter Typ A, indem jedes Auftreten eines $X^i \in Dom(S)$ durch $p.X$ ersetzt ist, abgekürzt $\text{qualify}(A; p; S)$, ist wie folgt definiert:

$$\text{qualify}(A; p; \emptyset) = A$$
$$\text{qualify}(A; p; S, x^i : B) = \text{qualify}(\langle p.x \to x^i \rangle A; p; S)$$
$$\text{qualify}(T; p; S, X^i <: B) = \text{qualify}(\langle p.X \to X^i \rangle A; p; S)$$
$$\text{qualify}(T; p; S, X^i = B) = \text{qualify}(\langle p.X \to X^i \rangle A; p; S)$$

Kontraktive Typen

Der Typ A ist im Kontext S kontraktiv in dem Typbezeichner X^i, abgekürzt $S \vdash A \downarrow X^i$, falls er aufgrund seiner *Nichttrivialität* eine eindeutige Spezifikation liefert. Formal gelten in jedem wohlgeformten Kontext S die folgenden Kontraktivitätsaussagen für einen Typbezeichner X^i:

$$Y^j \downarrow X^i \Leftrightarrow i \neq j \qquad Y^j.Z \downarrow X^i \qquad \mathbf{Ok} \downarrow X^i \qquad \mathbf{Nok} \downarrow X^i \qquad \mathbf{Var}(A) \downarrow X^i$$
$$\mathbf{Tup}(S'; C) \downarrow X^i \qquad \mathbf{Rcd}(S') \downarrow X^i \qquad \mathbf{Exc}(S') \downarrow X^i \qquad \mathbf{Fun}(S') : B \downarrow X^i$$

Die restlichen zwei Typregeln beziehen sich auf ihren statischen Kontext S:

[Contractive Oper]
$$\frac{S, S' \vdash B \downarrow X^i}{S \vdash \mathbf{Oper}(S')B \downarrow X^i}$$

[Contractive Apply]
$$\frac{S \vdash A <: \mathbf{Oper}(S')B \qquad S \vdash D :: S' \qquad S \vdash \langle D \Rightarrow S' \rangle B \downarrow X^i}{S \vdash A(D) \downarrow X^i}$$

Mit nicht-kontraktiven Typen können in TLMIN typkorrekte Aussagen zu Laufzeitfehlern führen.

Restriktionen

Die Regel [Subtype Var] ist in Funktionssignaturen nicht anwendbar, da die Implementierung von Variablenparametern (Übergabe einer Objektreferenz und eines Offsets) inkompatibel mit der Implementierung von Wertparametern ist. Die Einhaltung dieser Restriktion wird statisch überprüft.

Ausgewählte Schnittstellen

Syntaxbaumrepräsentation vor der Typüberprüfung

```
interface TLPreAST
(* System : Tycoon Compiler TLMin
   File   : TLPreAST.ti
   Author : Gerd Bremer, Florian Matthes, Gerald Schroeder
   Date   : 10-Jun-1995
   Purpose: Abstract syntax trees for TLMin generated during
            parsing and inspected during type checking.
    The type checker creates a new syntax tree
    representation (see TLAST.ti)
*)

import
  list
  optional
  :Source
  :TLIde
  tlEnv
  errorLog
  tlSwitches
  :TLAST
  tlSubTypeCache
  tlExposeCache

export


  (* -- Types for Reflection: *)

  Let Cache = Tuple
                subTypeCache :tlSubTypeCache.T
                exposedCache :tlExposeCache.T
              end

  Let State = Tuple
                env :tlEnv.T
                errLog :errorLog.T
                typeCheck :tlSwitches.TypeCheck
                cache :Cache
              end


  (* -- Abbreviation: *)
```

```
Let Pos = Source.Position


(* -- Types: *)

Let Rec Type <:Ok =
  Tuple
    pos :Pos
      (* of type if declared; of value if inferred *)

  case wrongCase
    (* type error, suppress subsequent messages *)

  case okCase
    (* top element of type lattice  *)

  case nokCase
    (* bottom element of type lattice *)

  case uniCase
    (* used by type checker *)

  case ideCase with
    ideList :TLIde.List
      (* size = 1 => typeIde
         size > 1 => abstract typeIde
                       [typeIde | valueIde] {fieldIde} fieldIde *)

  case funCase with
    signatures :Signatures
    range :Type (* <:Ok *)

  case tupCase with
    signatures :Signatures
    variants :Variants
      (* non-empty, use tlAST.emptyVariants for plain tuples *)

  case rcdCase with
    signatures :Signatures

  case excCase with
    signatures :Signatures

  case operCase with
    signatures :Signatures
    range :Type
    rangeBound :optional.T(Type)
```

```
    case appCase with
      oper :Type
      bindings :Bindings

    case varCase with
      type :Type
    end


  (* -- Variants: *)

  and Variant <:Ok =
    Tuple
      label :TLIde.T
      signatures :Signatures
    end

  and Variants <:Ok = list.T(Variant)


  (* -- Signatures: *)

  and Signature <:Ok =
    Tuple
    case typeCase with
      (* ide <:type *)
      ide :TLIde.T
      type :Type

    case typeEqualCase with
      (* ide =type *)
      ide :TLIde.T
      type :Type

    case valueCase with
      (* ide :type *)
      ide :TLIde.T
      type :Type

    case typeBoundCase with
      (* ide <: bound <: type *)
      ide :TLIde.T
      type :Type
      bound :Type

    case typeEqualBoundCase with
```

```
    (* ide <: bound = type *)
    ide :TLIde.T
    type :Type
    bound :Type

  case recursiveCase with
    (* Rec ... *)
    signatures :Signatures

  case reflectCase with
    f :Fun(state :State) :TLAST.Signatures
    (* example in 'tlMinSyntax' when declaring
       Repeat-Construct *)
  end

and Signatures <:Ok = list.T(Signature)


(* -- Bindings: *)

and Binding <:Ok =
  Tuple
  case typeCase with
    (* Let ide = type *)
    ide :TLIde.T
    type :Type

  case typeBoundCase with
    (* Let ide <: bound = type *)
    ide :TLIde.T
    type :Type
    bound :Type

  case valueCase with
    (* let [var] ide = value *)
    ide :TLIde.T
    value :Value
    mutable :Bool

  case valueTypeCase with
    (* let [var] ide :type = value *)
    ide :TLIde.T
    value :Value
    mutable :Bool
    type :Type

  case recursiveCase with
```

```
      (* let rec ... / Let Rec ... *)
      bindings :Bindings

    case reflectCase with
      f :Fun(state :State isFun :Bool) :TLAST.CheckedBindings
      (* example in 'tlMinSyntax' when declaring
         open-Construct *)
    end

  and Bindings <:Ok = list.T(Binding)


  (* -- Values: *)

  and Value <:Ok =
    Tuple
      pos :Pos

    case wrongCase

    case okCase

    case intCase with
      val :Int

    case realCase with
      val :Real

    case longrealCase with
      val :String

    case charCase with
      val :Char

    case stringCase with
      val :String

    case ideCase with
      ide :TLIde.T

    case funCase with
      signatures :Signatures
      value :Value
      rangeType :optional.T(Type)

    case applyCase with
      value :Value
```

```
  bindings :Bindings

case arrayCase with
  bindings :Bindings

case indexCase with
  value :Value
  index :Value

case tupleCase with
  bindings :Bindings

case tupleCaseCase with
  bindings :Bindings
  label :TLIde.T
  type :Type

case recordCase with
  bindings :Bindings

case fieldCase with
  (* value.field *)
  value :Value
  field :TLIde.T

case extendCase with
  value :Value
  bindings :Bindings

case exceptionCase with
  value :Value
  signatures :Signatures

case seqCase with
  bindings :Bindings

case ifCase with
  condition :Value
  thenBindings :Bindings
  elseBindings :Bindings

case caseCase with
   (* error if not exhaustive *)
  value :Value
  branches :CaseBranches
  elseBindings :optional.T(Bindings)
```

```
    case loopCase with
      bindings :Bindings

    case exitCase with

    case tryCase with
      bindings :Bindings
      branches :TryBranches
      elseBindings :Bindings
      (* never empty, when missing else in source text
      then parser inserts 'else let ? :Nok = reraise' *)

    case raiseCase with
      value :Value
      bindings :Bindings

    case reraiseCase with
  end

  and CaseBranch <:Ok =
    Tuple
      pos :Pos
      labels :TLIde.List
      ide :TLIde.T
        (* TLIde.wrong if no ide in branch *)
      bindings :Bindings
    end

  and CaseBranches <:Ok = list.T(CaseBranch)

  and TryBranch <:Ok =
    Tuple
      pos :Pos
      value :Value
      ide :TLIde.T
        (* TLIde.wrong if no ide in branch *)
      bindings :Bindings
    end

  and TryBranches <:Ok = list.T(TryBranch)

  ...

end
```

Syntaxbaumrepräsentation nach der Typüberprüfung

```
interface TLAST
(* System : Tycoon Compiler TLMin
   File   : TLAST.ti
   Author : Gerd Bremer, Florian Matthes, Gerald Schroeder
   Date   : 10-Jun-1995
   Purpose: Representation of abstract syntax trees for TLMin
*)

import
  list
  optional
  :Source
  :TLIde

export

  (* -- Abbreviation: *)

  Let Pos = Source.Position


  (* -- Status of uniCase-Types: *)

  (* initial:
       UniCase-Type is unknown
       Debugging: Uni[?]
       Uni[?]-Types are build from UniCase-PreTypes during type
       validation (see 'when uniCase ...' in 'tlCheckType.checkType').
       They occur in function or type operator applications when
       type parameters are omitted. Uni[?]-Types are set to 'transient'
       (the compiler switch 'typeCheck' must be set) in the following
       subsignature test with the formal parameter type as upper bound
       (see 'when applyCase ...' in 'tlCheckValue.checkValue'
       and 'when appCase ...' in 'tlCheckType.checkType').

     transient:    UniCase-Type has an upper bound
                   Debugging: Uni[<:Type]

     instantiated: UniCase-Type is wellknown
                   Debugging: Uni[Type]
  *)

  Let Status = Int

  initial,transient,instantiated :Status
```

```
(* -- Identifier path: *)

(* Example ideCase-Type 'list.T':
   first = signature list :List
   path  = list.singleton(
             tuple_case tupCase of Field with
               let name = "T"
               let index = 1
             end)
*)

Let Field =
  Tuple
    name :String
      (* 'name' in tupCase only for debugging *)
  case tupCase with
    index :Int
      (* position of field in tuple type,
           debugging:
           do set printIndices true
           format: @index *)
  case rcdCase with
  end

Let Path = list.T(Field)


(* -- Types: *)

Let Rec Type <:Ok =
  Tuple
    pos :Pos
      (* of type if declared; of value if inferred *)

  case wrongCase
    (* type error, suppress subsequent messages *)

  case okCase
    (* top element of type lattice  *)

  case nokCase
    (* bottom element of type lattice *)

  case ideCase with
    first :Signature
```

```
    path :Path
      (* empty if simple identifier *)

  case uniCase with
    (* has to remain shared while not(instantiated) *)
    var status :Status
    var type :Type
      (* if 'status' == 'initial' then 'initialUniType'
         elsif 'status' == 'transient' then the bound
           specified for the formal type parameter
         else the instantiation type
         end
      *)

  case funCase with
    signatures :Signatures
    range :Type (* <:Ok *)

  case tupCase with
    signatures :Signatures
    variants :Variants
      (* non-empty, use 'emptyVariants' for plain tuples *)

  case rcdCase with
    signatures :Signatures

  case excCase with
    signatures :Signatures

  case varCase with
    type :Type (* <:Ok *)

  case operCase with
    signatures :Signatures
    range :Type
    rangeBound :optional.T(Type)

  case appCase with
    oper :Type
    signatures :Signatures
    bindings :Bindings
  end

and Variant <:Ok =
  Tuple
    label :TLIde.T
    signatures :Signatures
```

```
    end

  and Variants <:Ok = list.T(Variant)


(* -- Signatures: *)

  and Signature <:Ok =
    (* Note: identity is important to distinguish abstract
             type variables *)
    Tuple
      ide :TLIde.T
      var type :Type
        (* mutable to allow cyclic syntax trees *)
      recIndex :Int
        (* == 0 : no recursion
           == 1 : first recursive signature
           == 2 : second recursive signature
           ... *)
      count :Int
        (* only for debugging:
             do set printIndices true
             format: @index *)

    case typeCase
      (* ide <:type *)

    case typeEqualCase
      (* ide = type *)

    case valueCase
      (* ide :type *)
  end

  and Signatures <:Ok = list.T(Signature)


(* -- Bindings: *)

  and Binding <:Ok =
    Tuple
      ide :TLIde.T
      type :Type
      recIndex :Int
        (* same meaning as 'recIndex' in 'Signature' *)

    case typeCase
```

```
      (* Let ide = type *)

    case valueCase with
      value :Value
      var isVarParameter :Bool
        (* needed for backend; initially false; set if binding occurs
           in function application and expected formal parameter
           type is varCase (see 'setVarParameter' in 'tlCheckValue'
           and 'when applyCase ...' in 'tlCheckValue.checkValue'. *)
  end

  and Bindings <:Ok = list.T(Binding)
  (* First definition in source code is first element in list. *)


(* -- Values: *)

  and Value <:Ok =
    Tuple
      pos :Pos

    case wrongCase

    case okCase

    case intCase with
      val :Int

    case realCase with
      val :Real

    case longrealCase with
      val :String

    case charCase with
      val :Char

    case stringCase with
      val :String

    case ideCase with
      index :Int
        (* needed for backend;
           relative position in type environment;
           debugging:
             do set printBackendIndices true
             do set printTLMin true
```

```
            format: #index *)
  signature :Signature

case funCase with
  signatures :Signatures
  value :Value
  rangeType :optional.T(Type)

case applyCase with
  value :Value
  bindings :Bindings

case arrayCase with
  bindings :Bindings

case indexCase with
  value :Value
  index :Value

case tupleCase with
  bindings :Bindings

case tupleCaseCase with
  bindings :Bindings
  label :TLIde.T
  index :Int
    (* needed for backend;
       position of variant in Variant Tuple Type;
       debugging:
         do set printBackendIndices true
         do set printTLMin true
         format: #index *)
  type :Type

case recordCase with
  bindings :Bindings

case fieldCase with
  (* value.field *)
  value :Value
  type :Type (* of 'value' *)
  field :Field
  fieldOffset :Int
    (* needed for backend;
       if field/tupCase:
         position in tuple type ignoring type fields
       if field/rcdCase:
```

```
          initialFieldOffset
        debugging:
          do set printBackendIndices true
          do set printTLMin true
          format: #index *)

  case extendCase with
    value :Value
    bindings :Bindings

  case exceptionCase with
    value :Value
    signatures :Signatures

  case seqCase with
    bindings :Bindings

  case ifCase with
    condition :Value
    thenBindings :Bindings
    elseBindings :Bindings

  case caseCase with
    value :Value
    branches :CaseBranches
    elseBindings :optional.T(Bindings)

  case loopCase with
    bindings :Bindings

  case exitCase

  case tryCase with
    bindings :Bindings
    branches :TryBranches
    elseBindings :Bindings
      (* never empty, when missing else in source text
         the parser inserts 'else let ? :Nok = reraise' *)

  case raiseCase with
    value :Value
    bindings :Bindings

  case reraiseCase
end

and CaseBranch <:Ok =
```

```
    Tuple
      pos :Pos
      labels :TLIde.List
      indices :list.T(Int)
        (* needed for backend,
           for l in labels:
             index[l] = position of variant l in Variant Tuple Type,
           debugging:
             do set printBackendIndices true
             do set printTLMin true
             format: #index *)
      ide :TLIde.T
        (* TLIde.wrong if no ide in branch *)
      bindings :Bindings
    end

  and CaseBranches <:Ok = list.T(CaseBranch)

  and TryBranch <:Ok =
  Tuple
    pos :Pos
    value :Value
    ide :TLIde.T
      (* TLIde.wrong if no ide in branch *)
    bindings :Bindings
  end

  and TryBranches <:Ok = list.T(TryBranch)

  ...

end
```

I. Messergebnisse

Die auf den nächsten Seiten präsentierten Tabellen enthalten Messergebnisse, die beim Übersetzen von Schnittstellen und Modulen mit dem TL und TLMIN Compiler ermittelt wurden. Die Messung erfolgte auf einer *Sun SparcStation* 20. Insgesamt wurden 411 Komponenten aus 9 Bibliotheken übersetzt, die in der folgenden Tabelle aufgeführt sind:

Verzeichnis	Komponenten	Schnittstellen	Module	Zeilen	Worte	Zeichen
machineenv	35	18	17	3224	10839	87715
stdenv	48	24	24	6314	22301	167186
bulkenv	73	37	36	11921	41543	350259
unixenv	16	8	8	1830	6052	43739
compenv	50	23	27	8192	26653	231416
tml0	24	12	12	8177	25167	207486
tl0	79	39	40	22522	72643	693453
egrammar	10	5	5	4163	14005	132357
tlmin	76	38	38	15339	44004	437446
insgesamt:	411	204	207	81579	263207	2351057

Die letzten drei Spalten enthalten die summierte Anzahl der Zeilen, Wörter und Zeichen in dem jeweiligen Verzeichnis. Die Analyse der Ergebnisse erfolgt in Abschnitt 6. Die Tabellen auf den nächsten Seiten enthalten folgende Information:

Spalte 1 (Komponente): Die übersetzten Komponenten. Schnittstellen enden mit der Bezeichnung ".ti' und Module mit ".tm"

Spalte 2 (Zeilen): Die Anzahl der Programmzeilen.

Spalte 3 (TLMin/TL): Die Typüberprüfungszeit in Sekunden mit dem TLMIN bzw. mit dem TL Typprüfer.

Spalte 4 (<:): Die Anzahl der durchgeführten Subtyptests während der Typüberprüfung.

Spalte 5 (Typen): Die Anzahl der auf Wohlgeformtheit geprüften Typen während der Typüberprüfung.

Spalte 6 (Werte): Die Anzahl der auf Wohlgeformtheit geprüften Werte während der Typüberprüfung.

Spalte 7 (1.st): Die Anzahl der Subtyptests von Typoperatorapplikationen, die in der *first chance* mit dem Ergebnis *true* endeten.

Spalte 8 (2.nd): Die Anzahl der Subtyptests von Typoperatorapplikationen, die in der *first chance* mit dem Ergebnis *false* endeten und zur *second chance* führten.

Komponente	Zeilen	TLMIN	<:	Typen	Werte	1.st	2.nd
RuntimeCore.ti	41	0.09	22	18	0	0	0
runtimeCore.tm	42	0.35	71	25	54	0	0
Main.ti	15	0.07	11	6	0	0	0
Parameters.ti	55	0.05	7	5	0	0	0
parameters.tm	18	0.14	20	5	11	1	0
Session.ti	30	0.02	12	13	0	0	0
session.tm	36	0.18	42	11	42	0	0
WordOp.ti	273	2.85	316	305	0	0	0
word.tm	349	13.41	1017	325	719	8	0
CCallback.ti	52	0.10	24	23	0	0	0
cCallback.tm	29	0.39	70	24	29	3	0
Unsafe.ti	46	0.24	41	36	0	0	0
unsafe.tm	62	0.88	148	58	78	3	0
TSP.ti	218	2.76	301	291	0	0	0
tsp.tm	278	11.61	1069	327	472	12	0
Store.ti	143	3.32	262	243	0	0	0
store.tm	222	11.17	755	223	508	18	0
TM.ti	77	0.81	129	114	0	0	0
tm.tm	146	3.50	379	107	269	0	0
Thread.ti	134	0.59	136	139	0	0	0
thread.tm	162	4.65	687	225	278	40	0
Charrain.ti	57	0.32	85	86	0	0	0
charrain.tm	226	6.00	1155	179	563	36	0
Volatile.ti	31	0.15	46	46	0	0	0
volatile.tm	106	1.83	373	97	161	14	0
Reference.ti	25	0.06	18	22	0	0	0
reference.tm	113	1.96	446	77	237	18	0
XLink.ti	20	0.14	14	9	0	0	0
xlink.tm	25	0.29	37	12	19	0	0
Checkpoint.ti	18	0.02	8	9	0	0	0
checkpoint.tm	41	0.38	53	12	51	0	0
Atom.ti	34	0.15	34	28	0	0	0
atom.tm	50	0.60	104	33	65	0	0
Symloc.ti	27	0.07	12	12	0	0	0
symloc.tm	23	0.27	39	12	23	2	0
insgesamt:	3224	69.42	7943	3157	3579	155	0

Abbildung I.1: Komponenten im Verzeichnis machineenv

Komponente	Zeilen	TLMIN	<:	Typen	Werte	1.st	2.nd
Poly.ti	58	0.22	48	43	0	0	0
poly.tm	59	1.14	129	39	101	0	0
Order.ti	40	0.42	92	89	0	0	0
order.tm	42	1.37	242	69	107	9	0
IterRep.ti	54	0.07	15	15	0	0	0
Iter.ti	278	5.66	635	663	0	0	0
iter.tm	563	34.93	3587	1242	1486	265	0
Optional.ti	48	0.26	49	53	0	0	0
optional.tm	70	1.30	191	61	117	12	0
Ascii.ti	28	0.66	37	35	0	0	0
ascii.tm	48	0.44	37	1	70	0	0
BoolOp.ti	37	0.21	40	35	0	0	0
bool.tm	43	0.80	146	36	81	0	0
IntOp.ti	70	0.86	130	130	0	0	0
int.tm	150	4.82	630	184	317	10	0
CharOp.ti	64	0.75	89	81	0	0	0
char.tm	111	3.62	421	85	342	5	0
StringOp.ti	94	1.09	119	111	0	0	0
string.tm	205	6.00	738	147	536	10	0
String0.ti	112	1.52	150	141	0	0	0
string0.tm	212	6.91	768	183	503	10	0
ArrayOp.ti	73	0.82	109	108	0	0	0
arrayOp.tm	104	4.19	653	178	276	51	0
Pair.ti	26	0.33	75	78	0	0	0
pair.tm	33	1.24	207	71	61	9	0
RealOp.ti	69	1.06	137	132	0	0	0
real.tm	140	5.23	589	141	371	6	0
Writer.ti	59	0.46	63	56	0	0	0
writer.tm	95	2.38	320	79	204	8	0
Reader.ti	61	0.52	59	50	0	0	0
reader.tm	114	3.32	408	92	240	12	0
Fmt.ti	82	0.95	79	67	0	0	0
fmt.tm	186	5.41	589	127	426	22	0
Scan.ti	93	0.99	97	84	0	0	0
scan.tm	231	7.70	1000	144	725	9	0
Print.ti	53	0.43	58	43	0	0	0
print.tm	50	1.17	153	36	115	4	0
Time.ti	60	0.83	92	84	0	0	0
time.tm	214	6.57	641	84	673	0	0
Chronos.ti	27	0.05	9	7	0	0	0
userChronos.tm	24	0.32	37	10	18	1	0
cpuChronos.tm	22	0.32	40	7	20	0	0
Directory.ti	86	0.78	64	61	0	0	0
directory.tm	484	12.07	1441	220	1030	58	0
TypeRep.ti	135	1.24	128	133	0	0	0
typeRep.tm	444	9.58	1240	358	545	61	0
Dynamic.ti	169	1.55	162	151	0	0	0
dynamic.tm	794	22.09	2171	391	1486	101	0
insgesamt:	6314	164.65	18914	6435	9850	663	0

Abbildung I.2: Komponenten im Verzeichnis stdenv

Komponente	Zeilen	TLMIN	<:	Typen	Werte	1.st	2.nd
List.ti	52	0.41	76	78	0	0	0
Cell.ti	152	3.65	461	483	0	0	0
Position.ti	21	0.11	30	33	0	0	0
Switches.ti	94	1.15	103	96	0	0	0
Set.ti	49	0.49	82	81	0	0	0
MutSet.ti	74	0.84	139	145	0	0	0
VarList.ti	90	0.71	125	124	0	0	0
Stack.ti	44	0.46	80	79	0	0	0
BndStack.ti	47	0.58	87	86	0	0	0
DynArray.ti	82	1.11	136	137	0	0	0
Dictionary.ti	71	0.80	145	145	0	0	0
IdDict.ti	67	0.70	125	125	0	0	0
Queue.ti	72	0.59	104	104	0	0	0
HashTable.ti	85	1.05	189	189	0	0	0
Assoc.ti	61	1.57	210	210	0	0	0
BTreeUtil.ti	91	1.47	180	183	0	0	0
BTree.ti	104	2.13	280	277	0	0	0
BTreeDelete.ti	17	0.08	9	10	0	0	0
BTreeElements.ti	41	1.18	124	125	0	0	0
BTreeInsert.ti	17	0.08	10	11	0	0	0
BTreeReorg.ti	19	0.12	12	10	0	0	0
Cache.ti	68	0.51	88	88	0	0	0
Sort.ti	17	0.09	13	11	0	0	0
BitSet.ti	88	1.57	146	154	0	0	0
DynBitSet.ti	88	1.02	144	152	0	0	0
Memoize.ti	41	0.30	67	64	0	0	0
list.tm	77	2.63	447	126	135	43	0
cell.tm	319	27.24	3527	913	1031	289	0
position.tm	28	1.04	89	40	19	10	0
set.tm	68	2.68	448	106	136	38	0
mutSet.tm	117	5.44	797	193	250	74	0
varList.tm	319	11.31	1488	258	898	53	0
bndStack.tm	98	3.84	566	117	225	41	0
dynArray.tm	173	8.33	1035	197	485	70	0
stack.tm	88	3.72	507	115	166	47	0
dictionary.tm	131	5.38	897	214	277	67	0
idDict.tm	197	9.98	1469	267	583	85	0
queue.tm	149	5.88	831	211	321	59	0
hashTable.tm	236	10.71	1615	309	683	70	0
assoc.tm	196	15.82	1878	358	469	149	0

Abbildung I.3: Komponenten im Verzeichnis bulkenv

Komponente	Zeilen	TLMIN	<:	Typen	Werte	1.st	2.nd
bTreeUtil.tm	282	22.10	2826	369	805	209	0
bTreeDelete.tm	360	24.18	3375	365	1585	158	0
bTreeElements.tm	317	45.79	5715	728	896	524	0
bTreeInsert.tm	233	17.63	2469	320	835	172	0
bTree.tm	154	12.15	1634	369	219	140	0
bTreeReorg.tm	51	0.60	63	38	13	2	0
cache.tm	160	3.39	425	106	165	18	0
sort.tm	42	1.52	265	23	126	9	0
switches.tm	129	5.56	623	119	241	28	0
bitSet.tm	198	12.21	1447	203	778	74	0
dynBitSet.tm	207	11.03	1340	219	748	78	0
memoize.tm	133	2.55	463	120	171	16	0
DDimDicPartList.ti	104	0.93	142	146	0	0	0
dDimDicPartList.tm	357	12.31	1711	305	966	59	0
DDimDicUtil.ti	109	2.20	226	237	0	0	0
dDimDicUtil.tm	296	15.34	1742	433	793	103	0
DDimDicBuild.ti	43	1.27	103	99	0	0	0
dDimDicBuild.tm	330	21.81	2006	383	660	149	19
DDimDictionary1.ti	69	2.27	163	145	0	0	0
dDimDictionary1.tm	1008	100.70	6966	823	4816	233	90
DDimDictionary.ti	223	3.39	246	242	0	0	0
dDimDictionary.tm	826	73.66	5909	829	3007	278	50
DDimDicDebug.ti	51	1.61	155	146	0	0	0
dDimDicDebug.tm	414	32.50	2461	353	1591	130	34
DirectedGraph.ti	126	1.04	118	120	0	0	0
directedGraph.tm	225	8.68	1144	219	396	69	0
Graph.ti	37	1.11	45	41	0	0	0
graph.tm	140	5.59	535	84	330	31	0
Iter2.ti	313	10.43	759	772	0	0	0
List2.ti	39	0.49	43	41	0	0	0
LinkedList.ti	37	0.27	16	20	0	0	0
iter2.tm	842	55.80	5353	1375	2177	295	0
linkedList.tm	218	23.13	2441	236	549	187	0
insgesamt:	11921	670.01	71688	16652	27545	4057	193

Abbildung I.4: Fortsetzung: Komponenten im Verzeichnis bulkenv

Komponente	Zeilen	TLMIN	<:	Typen	Werte	1.st	2.nd
Fingerprint.ti	71	0.61	59	48	0	0	0
CharClass.ti	51	0.36	37	29	0	0	0
Memory.ti	84	0.66	81	75	0	0	0
Source.ti	68	0.68	57	45	0	0	0
ErrorLog.ti	98	0.75	80	71	0	0	0
Symbol.ti	26	0.13	10	8	0	0	0
SymbolSet.ti	47	0.48	44	40	0	0	0
Unparser.ti	117	0.96	79	73	0	0	0
fingerprint.tm	91	2.50	236	45	179	0	0
charClass.tm	425	4.34	429	66	356	2	0
source.tm	147	4.71	567	66	339	2	0
errorLog.tm	127	3.91	475	78	260	16	0
memory.tm	179	4.81	670	94	270	31	0
symbol.tm	23	0.42	48	6	14	2	0
symbolSet.tm	117	5.95	659	92	326	43	0
unparser.tm	422	13.13	1115	210	838	15	0
Scanner.ti	57	0.40	55	54	0	0	0
RegExprRep.ti	65	0.25	40	34	0	0	0
RegExpr.ti	56	0.45	41	36	0	0	0
LexicalAnalyzer.ti	61	0.32	47	49	0	0	0
DFA.ti	114	0.35	48	45	0	0	0
RegToLex.ti	137	0.19	34	32	0	0	0
scanner.tm	32	1.33	171	26	27	9	0
regExprRep.tm	81	1.62	183	30	137	0	0
regExpr.tm	69	2.38	304	50	171	16	0
lexicalAnalyzer.tm	31	1.14	144	22	27	8	0
dfa.tm	326	8.51	933	180	541	19	0
regToLex.tm	419	9.84	959	129	772	30	0
Parser.ti	48	0.52	66	62	0	0	0
LL1Grammar.ti	62	0.48	48	38	0	0	0
Grammar.ti	152	3.63	436	435	0	0	0
parser.tm	61	1.99	237	47	57	8	0
lL1Grammar.tm	440	19.00	1984	264	1002	99	0
grammar.tm	499	26.13	3490	647	1070	182	0
Lexicon.ti	53	0.47	50	41	0	0	0
lexicon.tm	100	2.60	352	76	159	17	0
LRParserGen.ti	14	0.15	8	6	0	0	0
LRParser.ti	95	1.13	101	96	0	0	0
LRGrammar.ti	98	1.80	216	212	0	0	0
lrParserGen.tm	27	0.71	40	8	36	0	0

Abbildung I.5: Komponenten im Verzeichnis compenv

Komponente	Zeilen	TLMIN	<:	Typen	Werte	1.st	2.nd
lrParser.tm	700	29.19	2832	413	1658	124	0
lrGrammar.tm	405	20.02	2752	364	866	100	0
QParser.ti	70	0.36	36	34	0	0	0
Dependencies.ti	20	0.01	4	5	0	0	0
qParser.tm	308	9.84	792	81	678	25	0
dependencies.tm	410	15.55	1046	115	899	30	0
depend.tm	242	10.63	866	99	601	31	0
depend2.tm	311	13.33	1068	123	725	42	0
dependQuest.tm	22	0.30	16	5	9	1	0
dependTycoon.tm	23	0.29	16	5	9	1	0
insgesamt:	7701	229.31	24061	4909	12026	853	0

Abbildung I.6: Fortsetzung: Komponenten im Verzeichnis compenv

Komponente	Zeichen	TLMIN	<:	Typen	Werte	1.st	2.nd
TVM.ti	106	3.98	111	105	0	0	0
tvm.tm	596	20.80	1536	179	1280	19	0
TML.ti	282	3.96	256	247	0	0	0
tml.tm	651	41.63	5363	254	1703	48	0
AstTools.ti	36	0.48	12	9	0	0	0
astTools.tm	219	10.61	1473	27	387	10	0
TML2tvm.ti	46	1.25	55	51	0	0	0
TMLPrimitives.ti	68	0.88	57	54	0	0	0
TMLUnparse.ti	18	0.43	7	7	0	0	0
PTML.ti	36	0.78	23	18	0	0	0
tmlPrimitives.tm	872	43.51	4899	248	2196	85	0
tmlUnparse.tm	282	21.07	2466	64	945	24	0
ptml.tm	887	48.69	3993	337	2726	57	0
tml2tvm.tm	1390	104.21	7866	926	3852	244	0
TMLException.ti	29	0.42	4	4	0	0	0
tmlException.tm	482	36.30	4397	325	1395	108	0
TMLOptControl.ti	66	1.71	67	62	0	0	0
TMLReduce.ti	36	0.57	18	15	0	0	0
TMLExpand.ti	21	0.59	10	7	0	0	0
TMLOptimize.ti	27	0.70	20	14	0	0	0
tmlOptControl.tm	98	5.92	797	34	107	15	0
tmlReduce.tm	841	54.51	6938	294	2304	87	0
tmlExpand.tm	941	74.62	9436	334	2960	78	0
tmlOptimize.tm	147	13.44	2301	55	363	17	0
insgesamt:	8177	491.06	52105	3670	20218	792	0

Abbildung I.7: Komponenten im Verzeichnis tml0

Komponente	Zeilen	TLMIN	<:	Typen	Werte	1.st	2.nd
TyNames.ti	60	0.81	34	25	0	0	0
TyGlobal.ti	73	2.30	94	86	0	0	0
TLReflect.ti	27	0.43	26	22	0	0	0
tyNames.tm	49	1.01	70	13	72	0	0
tyGlobal.tm	65	2.78	148	11	59	0	0
tlReflect.tm	76	1.73	157	43	93	3	0
TLIde.ti	71	0.97	47	42	0	0	0
TLType.ti	201	1.39	118	122	0	0	0
TLValue.ti	234	2.88	208	212	0	0	0
TLUnit.ti	65	0.25	29	23	0	0	0
tlIde.tm	84	2.25	193	32	94	4	0
tlType.tm	262	11.32	1291	131	467	44	0
tlValue.tm	99	9.72	1537	77	32	27	0
TLEnv.ti	135	1.81	113	98	0	0	0
TLUnparse.ti	48	1.23	55	38	0	0	0
tlUnparse.tm	871	28.02	3561	217	2321	162	0
tlEnv.tm	477	26.50	2549	272	1358	60	0
TMLEnv.ti	41	0.66	26	22	0	0	0
TyRuntime.ti	49	0.23	27	24	0	0	0
tmlEnv.tm	70	5.59	489	40	178	13	0
tyRuntime.tm	92	2.53	161	22	156	1	0
TimeMeasure.ti	70	1.42	84	73	0	0	0
timeMeasure.tm	121	4.89	550	76	223	14	0
TLIndex.ti	154	2.83	125	104	0	0	0
TL2tml.ti	45	0.57	16	9	0	0	0
TLBuiltin.ti	20	0.51	11	11	0	0	0
TLCheck.ti	117	1.65	89	74	0	0	0
TLCheckType.ti	57	0.67	44	30	0	0	0
TLCheckUnit.ti	39	0.27	17	10	0	0	0
TLInitialEnv.ti	74	2.67	91	83	0	0	0
tlInitialEnv.tm	231	11.34	1105	52	471	32	0
tlIndex.tm	1267	84.99	7650	636	3481	188	0
tlBuiltin.tm	671	44.02	3161	496	2124	116	0
tlCheck.tm	1367	86.84	5856	543	3713	170	0
tlCheckType.tm	435	23.15	2231	184	1101	87	0
tlCheckUnit.tm	103	5.38	461	38	158	25	0
TLDynamicImpl.ti	118	1.59	99	93	0	0	0
tlDynamicImpl.tm	708	36.21	3319	291	1783	91	0
TLDynamic.ti	28	0.88	11	6	0	0	0
TLDynEnv.ti	38	0.08	16	14	0	0	0

Abbildung I.8: Komponenten im Verzeichnis tl0

Komponente	Zeilen	TLMIN	<:	Typen	Werte	1.st	2.nd
tlDynEnv.tm	75	1.87	176	26	128	6	0
tlDynamic.tm	1281	68.86	4963	731	2956	177	0
TLClosure.ti	64	0.93	57	50	0	0	0
TLTypedClosure.ti	88	3.29	83	81	0	0	0
TLCheckValue.ti	35	0.97	19	12	0	0	0
TLLibrary.ti	108	1.20	81	74	0	0	0
TLScanner.ti	107	0.76	37	29	0	0	0
TLParser.ti	30	0.54	37	27	0	0	0
TLSyntaxType.ti	78	1.61	98	94	0	0	0
TLSyntaxValue.ti	18	0.07	6	5	0	0	0
tlClosure.tm	102	4.14	442	44	187	10	0
tlTypedClosure.tm	679	32.67	2596	325	1474	99	0
tlCheckValue.tm	1386	86.21	6115	548	4041	154	0
tl2tml.tm	1779	134.24	8232	1261	4553	304	0
tlLibrary.tm	732	28.26	1905	281	1555	91	0
tlScanner.tm	374	14.99	1015	122	1030	20	0
tlParser.tm	86	3.96	237	37	143	5	0
tlSyntaxType.tm	545	42.26	4474	796	1057	309	0
tlSyntaxValue.tm	752	83.90	6533	973	1389	376	0
tlSyntaxUnit.tm	252	15.51	1944	317	448	125	0
TLValueUnparse.ti	43	0.78	34	24	0	0	0
tlValueUnparse.tm	580	28.56	2039	215	1508	89	0
TLObject.ti	211	2.03	135	114	0	0	0
TySourceFPCache.ti	34	0.14	17	14	0	0	0
tlObject.tm	1182	60.19	4016	445	2941	141	0
tySourceFPCache.tm	65	1.85	178	34	86	6	0
TyDoCommand.ti	65	1.85	55	42	0	0	0
TyDepend.ti	69	0.72	40	28	0	0	0
TyExec.ti	53	0.73	48	31	0	0	0
TyTopLevel.ti	44	0.73	32	25	0	0	0
Tycoon.ti	14	0.31	14	9	0	0	0
tyExec.tm	553	27.50	1576	85	1350	24	0
tyDepend.tm	550	29.21	1839	207	1484	49	0
tyDoCommand.tm	956	53.86	2846	385	2550	46	0
tySyntaxCommand.tm	119	6.26	605	103	168	34	0
tyTopLevel.tm	220	11.61	743	77	447	16	0
tycoon.tm	219	10.88	795	51	518	14	0
TLOptimize.ti	36	0.75	30	22	0	0	0
tlOptimize.tm	137	9.54	654	39	404	13	0
insgesamt:	22533	1188.11	90615	12178	48301	3145	0

Abbildung I.9: Fortsetzung: Komponenten im Verzeichnis tl0

Komponente	Zeilen	TLMIN	<:	Typen	Werte	1.st	2.nd
LongIntOp.ti	78	0.77	80	75	0	0	0
Process.ti	17	0.17	9	4	0	0	0
UxEnvironment.ti	25	0.29	11	9	0	0	0
UxFile.ti	73	1.73	71	63	0	0	0
longInt.tm	304	10.94	1237	643	149	29	0
process.tm	35	0.82	35	10	35	0	0
uxEnvironment.tm	45	1.89	70	13	73	0	0
uxFile.tm	111	5.39	256	61	187	0	0
Date.ti	154	3.36	94	85	0	0	0
NewDate.ti	112	3.19	100	91	0	0	0
date.tm	300	13.78	576	97	581	2	0
newDate.tm	190	12.54	478	119	404	2	0
Audio.ti	58	0.54	37	32	0	0	0
Image.ti	49	0.27	22	17	0	0	0
audio.tm	200	8.15	413	63	440	3	0
image.tm	79	1.87	102	12	133	0	0
insgesamt:	1830	65.70	3591	1394	2002	36	0

Abbildung I.10: Komponenten im Verzeichnis unixenv

Komponente	Zeilen	TLMIN	<:	Typen	Werte	1.st	2.nd
SymbolQueue.ti	30	0.17	20	17	0	0	0
symbolQueue.tm	81	2.55	248	46	117	7	0
EGrammar.ti	247	5.57	345	349	0	0	0
EGramRep.ti	377	9.39	492	491	0	0	0
eGramRep.tm	1557	93.06	6975	941	3804	254	0
eGrammar.tm	1063	66.78	5290	374	2457	224	0
EGramSyntax.ti	154	0.21	16	11	0	0	0
eGramSyntax.tm	444	19.90	2361	299	1324	142	0
EGramParser.ti	61	0.74	47	42	0	0	0
eGramParser.tm	149	7.57	644	107	255	20	0
insgesamt:	4163	205.94	16438	2677	7957	647	0

Abbildung I.11: Komponenten im Verzeichnis egrammar

Komponente	Zeilen	TLMIN	<:	Typen	Werte	1.st	2.nd
TLIde.ti	47	0.61	30	26	0	0	0
TLSwitches.ti	106	2.71	99	94	0	0	0
TLAST.ti	559	7.70	453	442	0	0	0
TLEnv.ti	58	1.06	66	57	0	0	0
tlIde.tm	55	1.86	159	20	76	2	0
tlSwitches.tm	93	5.78	642	13	66	0	0
tlAST.tm	647	41.08	3184	423	885	125	0
tlEnv.tm	144	12.05	982	85	278	40	0
TLReflect.ti	27	0.58	26	22	0	0	0
tlReflect.tm	76	2.31	157	43	93	3	0
TLUnparse.ti	40	1.08	47	35	0	0	0
TLTrace.ti	37	0.40	37	30	0	0	0
TLSubTypeCache.ti	33	0.33	20	16	0	0	0
TLExposeCache.ti	57	1.05	51	40	0	0	0
tlUnparse.tm	618	27.63	2677	270	1622	107	0
tlTrace.tm	75	2.35	115	31	99	0	0
tlSubTypeCache.tm	172	17.40	1287	93	353	76	0
tlExposeCache.tm	146	12.80	942	92	208	35	0
TLPreAST.ti	429	6.22	433	411	0	0	0
tlPreAST.tm	396	20.89	1969	294	335	43	0
TLPreUnparse.ti	38	1.06	41	31	0	0	0
TLPreTrace.ti	38	0.60	31	25	0	0	0
tlPreUnparse.tm	514	15.31	1598	176	1291	71	0
tlPreTrace.tm	61	2.58	97	27	70	0	0
TLSubstitute.ti	72	1.08	56	44	0	0	0
TLQualify.ti	68	0.68	37	29	0	0	0
tlSubstitute.tm	475	32.39	2598	229	1002	121	0
tlQualify.tm	245	15.69	1210	88	519	39	0
TLCheck.ti	65	0.93	47	33	0	0	0
TLCheckUtil.ti	87	2.44	96	70	0	0	0
TLCheckType.ti	38	0.67	32	20	0	0	0
TLCheckValue.ti	38	1.29	24	15	0	0	0
TLCheckBinding.ti	21	0.26	12	6	0	0	0
tlCheck.tm	870	50.68	3209	280	1988	67	0
tlCheckUtil.tm	460	45.26	3076	273	1171	140	0
tlCheckType.tm	474	31.31	2309	171	1181	108	0
tlCheckValue.tm	749	38.25	2685	169	1951	90	0
tlCheckBinding.tm	312	31.51	2236	123	808	119	0
TLCheckOpenAndRepeat.ti	25	0.71	21	13	0	0	0
TLCheckUnit.ti	21	0.27	15	8	0	0	0

Abbildung I.12: Komponenten im Verzeichnis tlmin

Komponente	Zeilen	TLMIN	<:	Typen	Werte	1.st	2.nd
tlCheckOpenAndRepeat.tm	235	20.04	1530	86	519	56	0
tlCheckUnit.tm	181	26.88	1942	29	913	48	0
TLScanner.ti	107	1.18	37	29	0	0	0
TLCmdSyntax.ti	43	0.51	22	17	0	0	0
TLMinSyntax.ti	55	2.02	75	61	0	0	0
TLParser.ti	44	0.65	29	22	0	0	0
tlScanner.tm	383	17.61	1017	120	1032	20	0
tlCmdSyntax.tm	144	7.71	581	107	228	35	0
tlMinSyntax.tm	695	78.93	5654	714	2305	416	0
tlParser.tm	207	9.47	897	114	351	52	0
TimeMeasure.ti	70	1.99	84	73	0	0	0
TMLEnv.ti	46	1.00	30	26	0	0	0
timeMeasure.tm	121	5.95	550	76	223	14	0
tmlEnv.tm	66	8.32	505	45	178	13	0
TLBuiltin.ti	20	0.52	11	11	0	0	0
TLInitialEnv.ti	68	2.63	53	46	0	0	0
TL2tml.ti	48	0.53	12	8	0	0	0
tlInitialEnv.tm	181	8.42	728	40	334	21	0
tlBuiltin.tm	674	48.56	3158	495	2120	116	0
tl2tml.tm	1450	113.95	6756	1050	3546	273	0
TLTopLevel.ti	69	1.10	65	58	0	0	0
tlTopLevel.tm	156	7.72	547	94	313	21	0
TLValueUnparse.ti	51	1.06	43	32	0	0	0
TyGlobal.ti	25	0.12	13	11	0	0	0
TyDoCommand.ti	52	0.74	35	24	0	0	0
TyRuntime.ti	49	0.32	27	24	0	0	0
TyExec.ti	31	0.36	24	15	0	0	0
TyTopLevel.ti	34	0.33	20	13	0	0	0
Tycoon.ti	18	0.39	14	9	0	0	0
tlValueUnparse.tm	472	37.25	2300	161	1193	83	0
tyGlobal.tm	23	0.23	22	3	20	0	0
tyRuntime.tm	92	3.54	161	22	156	1	0
tyExec.tm	315	18.41	769	53	797	4	0
tyDoCommand.tm	724	51.02	2272	305	1953	47	0
tyTopLevel.tm	190	11.96	720	38	396	14	0
tycoon.tm	144	7.62	477	37	308	14	0
insgesamt:	15769	937.90	63986	8435	30881	2434	0

Abbildung I.13: Fortsetzung: Komponenten im Verzeichnis tlmin

Komponente	Zeilen	TL	<:	Typen	Werte	1.st	2.nd
RuntimeCore.ti	41	0.10	7	18	0	0	0
runtimeCore.tm	42	0.86	63	25	54	0	0
Main.ti	15	0.08	2	6	0	0	0
Parameters.ti	55	0.07	2	5	0	0	0
parameters.tm	18	1.22	14	5	11	1	0
Session.ti	30	0.05	6	13	0	0	0
session.tm	36	0.68	40	9	42	0	0
WordOp.ti	273	1.83	76	306	0	0	0
word.tm	349	9.09	1052	326	708	10	0
CCallback.ti	52	0.80	7	23	0	0	0
cCallback.tm	29	2.19	58	24	29	3	0
Unsafe.ti	46	0.86	9	36	0	0	0
unsafe.tm	62	2.54	115	56	78	3	0
TSP.ti	218	2.58	66	291	0	0	0
tsp.tm	278	9.38	947	324	469	4	0
Store.ti	143	4.67	70	245	0	0	0
store.tm	222	10.37	553	223	508	14	0
TM.ti	77	0.45	37	114	0	0	0
tm.tm	146	6.04	388	106	261	0	0
Thread.ti	134	0.29	49	140	0	0	0
thread.tm	162	4.50	650	211	278	47	0
Charrain.ti	57	0.19	25	87	0	0	0
charrain.tm	226	8.45	1260	132	497	35	0
Volatile.ti	31	0.11	17	47	0	0	0
volatile.tm	106	3.95	363	94	161	14	0
Reference.ti	25	0.08	7	23	0	0	0
reference.tm	113	4.68	452	61	202	18	0
XLink.ti	20	0.81	1	9	0	0	0
xlink.tm	25	2.14	21	12	19	0	0
Checkpoint.ti	18	0.06	2	9	0	0	0
checkpoint.tm	41	2.67	55	9	51	0	0
Atom.ti	34	0.11	8	29	0	0	0
atom.tm	50	1.58	87	32	65	0	0
Symloc.ti	27	0.64	5	12	0	0	0
symloc.tm	23	1.96	35	12	23	2	0
insgesamt:	3224	86.08	6549	3074	3456	151	0

Abbildung I.14: Komponenten im Verzeichnis machineenv

Komponente	Zeilen	TL	<:	Typen	Werte	1.st	2.nd
Poly.ti	58	0.13	14	43	0	0	0
poly.tm	59	4.83	106	35	101	0	0
Order.ti	40	0.23	29	90	0	0	0
order.tm	42	1.40	212	67	107	9	0
IterRep.ti	54	0.08	7	22	0	0	0
Iter.ti	278	3.99	215	663	0	0	0
iter.tm	563	23.53	4229	1666	1456	362	0
Optional.ti	48	0.17	17	54	0	0	0
optional.tm	70	1.15	155	60	72	15	0
Ascii.ti	28	0.29	0	35	0	0	0
ascii.tm	48	0.99	70	1	70	0	0
BoolOp.ti	37	0.72	9	35	0	0	0
bool.tm	43	1.78	189	28	78	0	0
IntOp.ti	70	2.60	37	130	0	0	0
int.tm	150	6.44	738	188	317	16	0
CharOp.ti	64	2.39	22	81	0	0	0
char.tm	111	5.75	525	77	327	6	0
StringOp.ti	94	2.59	30	111	0	0	0
string.tm	205	9.77	780	141	489	18	0
String0.ti	112	2.78	36	141	0	0	0
string0.tm	212	10.12	800	176	473	14	0
ArrayOp.ti	73	2.38	33	108	0	0	0
arrayOp.tm	104	6.07	684	173	260	61	0
Pair.ti	26	0.24	30	79	0	0	0
pair.tm	33	1.29	193	67	59	11	0
RealOp.ti	69	2.58	38	132	0	0	0
real.tm	140	7.31	659	129	369	8	0
Writer.ti	59	2.25	15	56	0	0	0
writer.tm	95	8.76	280	69	190	8	0
Reader.ti	61	2.33	16	50	0	0	0
reader.tm	114	8.66	417	97	240	16	0
Fmt.ti	82	2.39	16	67	0	0	0
fmt.tm	186	10.38	641	124	414	26	0
Scan.ti	93	3.21	26	84	0	0	0
scan.tm	231	11.40	1097	132	660	17	0
Print.ti	53	2.81	14	43	0	0	0
print.tm	50	5.47	136	33	115	6	0
Time.ti	60	0.95	19	84	0	0	0
time.tm	214	8.79	786	88	655	0	0
Chronos.ti	27	0.08	2	7	0	0	0
userChronos.tm	24	2.07	25	10	18	0	0
cpuChronos.tm	22	4.10	26	7	20	0	0
Directory.ti	86	3.05	20	61	0	0	0
directory.tm	484	14.54	1705	146	970	66	0
TypeRep.ti	135	3.77	33	135	0	0	0
typeRep.tm	444	23.27	1565	748	584	126	0
Dynamic.ti	169	6.63	41	153	0	0	0
dynamic.tm	794	32.26	3936	382	1423	151	0
insgesamt:	6314	258.77	20673	7108	9467	936	0

Abbildung I.15: Komponenten im Verzeichnis stdenv

Komponente	Zeilen	TL	<:	Typen	Werte	1.st	2.nd
List.ti	52	2.29	27	79	0	0	0
Cell.ti	152	3.83	150	484	0	0	0
Position.ti	21	0.15	11	34	0	0	0
Switches.ti	94	2.45	28	96	0	0	0
Set.ti	49	2.27	26	82	0	0	0
MutSet.ti	74	2.37	48	146	0	0	0
VarList.ti	90	2.31	40	125	0	0	0
Stack.ti	44	2.26	27	80	0	0	0
BndStack.ti	47	2.26	28	87	0	0	0
DynArray.ti	82	2.51	43	138	0	0	0
Dictionary.ti	71	2.51	51	146	0	0	0
IdDict.ti	67	2.48	44	126	0	0	0
Queue.ti	72	2.29	35	105	0	0	0
HashTable.ti	85	2.44	65	190	0	0	0
Assoc.ti	61	3.57	79	211	0	0	0
BTreeUtil.ti	91	5.35	62	192	0	0	0
BTree.ti	104	3.70	108	278	0	0	0
BTreeDelete.ti	17	4.86	3	10	0	0	0
BTreeElements.ti	41	5.78	54	125	0	0	0
BTreeInsert.ti	17	4.81	3	11	0	0	0
BTreeReorg.ti	19	4.72	3	10	0	0	0
Cache.ti	68	0.90	25	89	0	0	0
Sort.ti	17	2.70	3	11	0	0	0
BitSet.ti	88	2.48	51	155	0	0	0
DynBitSet.ti	88	2.44	51	153	0	0	0
Memoize.ti	41	0.21	23	65	0	0	0
list.tm	77	4.37	413	155	105	46	0
cell.tm	319	16.20	4020	1063	955	366	0
position.tm	28	0.91	83	40	19	10	0
set.tm	68	4.75	430	90	136	39	0
mutSet.tm	117	6.20	855	164	249	82	0
varList.tm	319	12.52	2505	210	664	224	0
bndStack.tm	98	6.10	556	105	211	43	0
dynArray.tm	173	8.57	1181	145	468	72	0
stack.tm	88	5.90	520	101	166	49	0
dictionary.tm	131	7.18	927	169	275	73	0
idDict.tm	197	12.66	1541	196	539	89	0
queue.tm	149	6.52	1014	191	279	90	0
hashTable.tm	236	9.77	1674	220	639	69	0
assoc.tm	196	13.48	2550	305	469	221	0

Abbildung I.16: Komponenten im Verzeichnis bulkenv

Komponente	Zeilen	TL	<:	Typen	Werte	1.st	2.nd
bTreeUtil.tm	282	29.64	4115	354	691	290	0
bTreeDelete.tm	360	61.81	9495	233	1241	815	0
bTreeElements.tm	317	46.60	8887	787	748	672	0
bTreeInsert.tm	233	38.69	5789	234	772	479	0
bTree.tm	154	19.87	2878	311	219	305	0
bTreeReorg.tm	51	6.29	46	41	13	2	0
cache.tm	160	4.98	366	94	165	17	0
sort.tm	42	4.11	314	20	122	9	0
switches.tm	129	8.04	626	88	241	27	0
bitSet.tm	198	9.68	1316	155	620	77	0
dynBitSet.tm	207	10.50	1155	152	590	81	0
memoize.tm	133	6.04	391	104	150	15	0
DDimDicPartList.ti	104	2.44	49	147	0	0	0
dDimDicPartList.tm	357	15.57	3623	281	722	404	0
DDimDicUtil.ti	109	3.99	101	264	0	0	0
dDimDicUtil.tm	296	17.04	3238	417	745	325	0
DDimDicBuild.ti	43	4.51	51	109	0	0	0
dDimDicBuild.tm	330	34.03	8326	408	620	1153	53
DDimDictionary1.ti	69	4.65	67	159	0	0	0
dDimDictionary1.tm	1008	145.81	34821	781	2803	4969	206
DDimDictionary.ti	223	2.93	79	244	0	0	0
dDimDictionary.tm	826	120.86	31593	890	2038	4391	55
DDimDicDebug.ti	51	7.14	68	157	0	0	0
dDimDicDebug.tm	414	35.10	5602	369	1309	596	26
DirectedGraph.ti	126	3.35	39	121	0	0	0
directedGraph.tm	225	10.04	1123	203	390	77	0
Graph.ti	37	4.07	14	41	0	0	0
graph.tm	140	14.63	638	61	327	45	0
Iter2.ti	313	6.69	302	780	0	0	0
List2.ti	39	2.60	18	48	0	0	0
LinkedList.ti	37	3.22	6	20	0	0	0
iter2.tm	842	41.91	5908	1605	2146	332	0
linkedList.tm	218	16.21	3178	179	543	463	0
insgesamt:	11921	932.11	153579	16239	22389	17017	340

Abbildung I.17: Fortsetzung: Komponenten im Verzeichnis bulkenv

Komponente	Zeilen	TL	<:	Typen	Werte	1.st	2.nd
Fingerprint.ti	71	0.20	13	48	0	0	0
CharClass.ti	51	0.15	8	29	0	0	0
Memory.ti	84	2.40	28	79	0	0	0
Source.ti	68	2.87	11	45	0	0	0
ErrorLog.ti	98	3.69	17	71	0	0	0
Symbol.ti	26	2.08	2	8	0	0	0
SymbolSet.ti	47	2.26	12	40	0	0	0
Unparser.ti	117	2.95	22	73	0	0	0
fingerprint.tm	91	6.68	240	44	178	0	0
charClass.tm	425	9.71	446	64	337	1	0
source.tm	147	7.68	569	39	336	3	0
errorLog.tm	127	8.38	510	61	244	17	0
memory.tm	179	8.29	591	65	270	26	0
symbol.tm	23	3.52	37	6	14	2	0
symbolSet.tm	117	6.50	688	71	284	44	0
unparser.tm	422	16.58	1110	214	760	44	0
Scanner.ti	57	3.40	28	62	0	0	0
RegExprRep.ti	65	0.91	8	36	0	0	0
RegExpr.ti	56	1.62	12	36	0	0	0
LexicalAnalyzer.ti	61	3.41	27	57	0	0	0
DFA.ti	114	4.81	17	47	0	0	0
RegToLex.ti	137	6.33	11	39	0	0	0
scanner.tm	32	8.77	166	72	47	14	0
regExprRep.tm	81	5.98	166	35	131	0	0
regExpr.tm	69	8.42	369	44	171	18	0
lexicalAnalyzer.tm	31	8.67	155	67	47	13	0
dfa.tm	326	15.69	1050	173	500	33	0
regToLex.tm	419	22.44	873	125	750	20	0
Parser.ti	48	4.81	28	72	0	0	0
LL1Grammar.ti	62	3.61	8	94	0	0	0
Grammar.ti	152	6.47	158	436	0	0	0
parser.tm	61	16.27	218	101	83	14	0
lL1Grammar.tm	440	27.05	2262	180	967	127	0
grammar.tm	499	28.21	3903	578	1066	237	0
Lexicon.ti	53	6.43	15	41	0	0	0
lexicon.tm	100	11.52	389	56	159	21	0
LRParserGen.ti	14	0.10	1	6	0	0	0
LRParser.ti	95	4.36	24	98	0	0	0
LRGrammar.ti	98	6.08	60	213	0	0	0
lrParserGen.tm	27	1.38	31	8	36	0	0

Abbildung I.18: Komponenten im Verzeichnis compenv

Komponente	Zeilen	TL	<:	Typen	Werte	1.st	2.nd
lrParser.tm	700	29.48	2660	344	1454	137	0
lrGrammar.tm	405	29.51	4148	302	796	275	0
QParser.ti	70	2.21	8	34	0	0	0
Dependencies.ti	20	0.09	2	5	0	0	0
qParser.tm	308	11.05	949	71	641	28	0
dependencies.tm	410	16.68	1304	94	841	36	0
depend.tm	242	15.29	1062	77	583	35	0
depend2.tm	311	17.16	1285	89	695	47	0
dependQuest.tm	22	1.26	11	5	9	1	0
dependTycoon.tm	23	1.28	11	5	9	1	0
insgesamt:	7701	414.69	25723	4659	11408	1194	0

Abbildung I.19: Fortsetzung: Komponenten im Verzeichnis compenv

Komponente	Zeilen	TL	<:	Typen	Werte	1.st	2.nd
TVM.ti	106	1.33	11	105	0	0	0
tvm.tm	596	17.78	1595	145	1202	29	0
TML.ti	282	22.92	75	576	0	0	0
tml.tm	651	118.85	2112	207	1394	48	0
AstTools.ti	36	8.64	2	9	0	0	0
astTools.tm	219	23.89	279	32	376	4	0
TML2tvm.ti	46	8.84	12	51	0	0	0
TMLPrimitives.ti	68	7.51	3	54	0	0	0
TMLUnparse.ti	18	5.67	2	7	0	0	0
PTML.ti	36	7.13	5	18	0	0	0
tmlPrimitives.tm	872	172.51	3996	142	1844	119	0
tmlUnparse.tm	282	44.14	823	68	845	22	0
ptml.tm	887	90.32	3432	224	2453	98	0
tml2tvm.tm	1390	300.12	6473	833	3376	469	0
TMLException.ti	29	10.47	1	4	0	0	0
tmlException.tm	482	183.50	2392	326	1107	195	0
TMLOptControl.ti	66	10.12	11	62	0	0	0
TMLReduce.ti	36	6.92	4	15	0	0	0
TMLExpand.ti	21	6.11	1	7	0	0	0
TMLOptimize.ti	27	6.11	3	14	0	0	0
tmlOptControl.tm	98	14.44	304	31	107	12	0
tmlReduce.tm	841	199.25	3411	165	1932	128	0
tmlExpand.tm	941	196.85	4520	144	2618	106	0
tmlOptimize.tm	147	38.20	628	31	363	14	0
insgesamt:	8177	1501.62	30095	3270	17617	1244	0

Abbildung I.20: Komponenten im Verzeichnis tml0

Komponente	Zeilen	TL	<:	Typen	Werte	1.st	2.nd
TyNames.ti	60	1.31	5	25	0	0	0
TyGlobal.ti	73	3.40	1	86	0	0	0
TLReflect.ti	27	0.12	3	25	0	0	0
tyNames.tm	49	2.14	94	13	71	0	0
tyGlobal.tm	65	4.33	188	11	59	0	0
tlReflect.tm	76	9.85	353	145	137	23	0
TLIde.ti	71	3.92	9	42	0	0	0
TLType.ti	201	9.20	65	488	0	0	0
TLValue.ti	234	16.34	117	817	0	0	0
TLUnit.ti	65	7.35	2	23	0	0	0
tlIde.tm	84	5.92	168	32	90	2	0
tlType.tm	262	34.67	1070	131	467	64	0
tlValue.tm	99	19.13	488	80	32	34	0
TLEnv.ti	135	6.47	16	98	0	0	0
TLUnparse.ti	48	9.82	11	38	0	0	0
tlUnparse.tm	871	97.77	3766	208	2237	199	0
tlEnv.tm	477	36.51	2360	205	1239	64	0
TMLEnv.ti	41	13.04	5	22	0	0	0
TyRuntime.ti	49	0.72	6	24	0	0	0
tmlEnv.tm	70	24.70	315	41	178	14	0
tyRuntime.tm	92	13.17	144	18	142	1	0
TimeMeasure.ti	70	0.29	12	73	0	0	0
timeMeasure.tm	121	14.23	593	103	243	25	0
TLIndex.ti	154	17.31	22	104	0	0	0
TL2tml.ti	45	13.97	1	9	0	0	0
TLBuiltin.ti	20	7.15	3	11	0	0	0
TLCheck.ti	117	9.98	9	74	0	0	0
TLCheckType.ti	57	10.02	4	30	0	0	0
TLCheckUnit.ti	39	8.42	2	10	0	0	0
TLInitialEnv.ti	74	21.17	21	83	0	0	0
tlInitialEnv.tm	231	27.91	743	33	443	20	0
tlIndex.tm	1267	177.89	6990	455	3352	270	0
tlBuiltin.tm	671	228.98	4515	415	2109	288	0
tlCheck.tm	1367	180.40	5472	437	3538	219	0
tlCheckType.tm	435	57.82	2035	161	1074	111	0
tlCheckUnit.tm	103	29.33	360	30	156	23	0
TLDynamicImpl.ti	118	9.41	41	293	0	0	0
tlDynamicImpl.tm	708	87.05	3256	292	1738	114	0
TLDynamic.ti	28	10.90	1	6	0	0	0
TLDynEnv.ti	38	8.62	4	14	0	0	0

Abbildung I.21: Komponenten im Verzeichnis tl0

Komponente	Zeilen	TL	<:	Typen	Werte	1.st	2.nd
tlDynEnv.tm	75	21.91	173	20	121	4	0
tlDynamic.tm	1281	188.24	5603	729	2818	244	0
TLClosure.ti	64	6.33	11	54	0	0	0
TLTypedClosure.ti	88	11.96	20	81	0	0	0
TLCheckValue.ti	35	8.56	2	12	0	0	0
TLLibrary.ti	108	11.89	20	76	0	0	0
TLScanner.ti	107	4.44	5	29	0	0	0
TLParser.ti	30	5.80	8	27	0	0	0
TLSyntaxType.ti	78	12.47	45	94	0	0	0
TLSyntaxValue.ti	18	10.38	2	5	0	0	0
tlClosure.tm	102	7.83	316	35	141	8	0
tlTypedClosure.tm	679	164.36	2509	276	1353	105	0
tlCheckValue.tm	1386	298.55	6708	442	3954	333	0
tl2tml.tm	1779	537.72	10163	1330	4376	623	0
tlLibrary.tm	732	47.70	2293	264	1487	129	0
tlScanner.tm	374	14.72	1227	52	1006	19	0
tlParser.tm	86	10.24	212	33	143	6	0
tlSyntaxType.tm	545	83.65	5027	496	1050	362	0
tlSyntaxValue.tm	752	661.43	6852	604	1386	441	0
tlSyntaxUnit.tm	252	29.74	2239	269	484	146	0
TLValueUnparse.ti	43	10.82	4	24	0	0	0
tlValueUnparse.tm	580	194.46	2526	191	1426	109	0
TLObject.ti	211	20.79	20	114	0	0	0
TySourceFPCache.ti	34	0.69	4	14	0	0	0
tlObject.tm	1182	80.82	4341	328	2873	145	0
tySourceFPCache.tm	65	6.89	189	24	86	6	0
TyDoCommand.ti	65	29.11	11	42	0	0	0
TyDepend.ti	69	31.35	5	28	0	0	0
TyExec.ti	53	33.28	5	31	0	0	0
TyTopLevel.ti	44	4.08	5	25	0	0	0
Tycoon.ti	14	0.13	3	9	0	0	0
tyExec.tm	553	101.92	1722	36	1326	20	0
tyDepend.tm	550	63.30	2016	161	1421	51	0
tyDoCommand.tm	956	97.96	3303	295	2536	82	0
tySyntaxCommand.tm	119	46.00	715	115	184	55	0
tyTopLevel.tm	220	77.29	1057	210	495	49	0
tycoon.tm	219	22.85	1194	24	515	39	0
TLOptimize.ti	36	0.18	4	22	0	0	0
tlOptimize.tm	137	32.14	375	24	371	8	0
insgesamt:	22533	4232.71	94204	11850	46857	4455	0

Abbildung I.22: Fortsetzung: Komponenten im Verzeichnis tl0

Komponente	Zeilen	TL	<:	Typen	Werte	1.st	2.nd
LongIntOp.ti	78	0.72	20	75	0	0	0
Process.ti	17	0.07	1	4	0	0	0
UxEnvironment.ti	25	0.08	2	9	0	0	0
UxFile.ti	73	0.26	13	63	0	0	0
longInt.tm	304	10.25	703	151	635	1	0
process.tm	35	1.22	27	10	35	0	0
uxEnvironment.tm	45	4.90	58	12	73	0	0
uxFile.tm	111	5.06	265	60	187	0	0
Date.ti	154	1.60	13	85	0	0	0
NewDate.ti	112	1.57	18	91	0	0	0
date.tm	300	9.81	594	93	592	2	0
newDate.tm	190	7.48	513	114	395	2	0
Audio.ti	58	1.97	10	32	0	0	0
Image.ti	49	1.88	5	17	0	0	0
audio.tm	200	10.78	482	58	438	2	0
image.tm	79	6.25	99	12	133	0	0
insgesamt:	1830	63.90	2823	886	2488	7	0

Abbildung I.23: Komponenten im Verzeichnis unixenv

Komponente	Zeilen	TL	<:	Typen	Werte	1.st	2.nd
SymbolQueue.ti	30	5.02	5	17	0	0	0
symbolQueue.tm	81	7.45	200	38	114	8	0
EGrammar.ti	247	8.01	94	358	0	0	0
EGramRep.ti	377	18.09	121	495	0	0	0
eGramRep.tm	1557	114.66	7287	727	3749	313	0
eGrammar.tm	1063	148.96	33379	263	2349	1201	0
EGramSyntax.ti	154	7.67	3	11	0	0	0
eGramSyntax.tm	444	30.39	2912	225	1303	184	0
EGramParser.ti	61	9.11	11	43	0	0	0
eGramParser.tm	149	16.36	492	87	253	18	0
insgesamt:	4163	365.72	44504	2264	7768	1724	0

Abbildung I.24: Komponenten im Verzeichnis egrammar

Komponente	Zeilen	TL	<:	Typen	Werte	1.st	2.nd
TLIde.ti	47	3.86	6	26	0	0	0
TLSwitches.ti	106	0.37	6	94	0	0	0
TLAST.ti	559	73.15	173	2028	0	0	0
TLEnv.ti	58	16.23	14	57	0	0	0
tlIde.tm	55	5.73	162	20	72	1	0
tlSwitches.tm	93	1.40	397	13	66	0	0
tlAST.tm	647	125.71	2591	394	864	94	0
tlEnv.tm	144	20.14	523	71	264	22	0
TLReflect.ti	27	0.12	3	25	0	0	0
tlReflect.tm	76	10.22	353	145	137	23	0
TLUnparse.ti	40	10.18	8	35	0	0	0
TLTrace.ti	37	10.81	7	30	0	0	0
TLSubTypeCache.ti	33	8.00	3	16	0	0	0
TLExposeCache.ti	57	10.24	5	40	0	0	0
tlUnparse.tm	618	64.95	2697	315	1628	105	0
tlTrace.tm	75	15.21	127	31	131	0	0
tlSubTypeCache.tm	172	38.12	1135	59	331	64	0
tlExposeCache.tm	146	20.26	560	76	204	20	0
TLPreAST.ti	429	91.77	179	2129	0	0	0
tlPreAST.tm	396	103.76	1020	291	335	10	0
TLPreUnparse.ti	38	15.75	7	31	0	0	0
TLPreTrace.ti	38	16.59	6	25	0	0	0
tlPreUnparse.tm	514	71.15	2073	195	1285	77	0
tlPreTrace.tm	61	21.48	106	27	102	0	0
TLSubstitute.ti	72	9.95	9	44	0	0	0
TLQualify.ti	68	12.99	5	29	0	0	0
tlSubstitute.tm	475	79.59	2488	224	978	118	0
tlQualify.tm	245	51.39	905	103	482	22	0
TLCheck.ti	65	14.88	3	33	0	0	0
TLCheckUtil.ti	87	22.42	13	70	0	0	0
TLCheckType.ti	38	14.48	4	20	0	0	0
TLCheckValue.ti	38	13.35	2	15	0	0	0
TLCheckBinding.ti	21	13.22	1	6	0	0	0
tlCheck.tm	870	113.80	3020	300	1842	79	0
tlCheckUtil.tm	460	88.82	2611	265	1128	114	0
tlCheckType.tm	474	108.21	2178	164	1154	92	0
tlCheckValue.tm	749	137.59	3045	152	1935	79	0
tlCheckBinding.tm	312	83.76	1754	105	767	94	0
TLCheckOpenAndRepeat.ti	25	14.70	4	13	0	0	0
TLCheckUnit.ti	21	14.29	2	8	0	0	0

Abbildung I.25: Komponenten im Verzeichnis tlmin

Komponente	Zeilen	TL	<:	Typen	Werte	1.st	2.nd
tlCheckOpenAndRepeat.tm	235	60.42	1138	85	517	43	0
tlCheckUnit.tm	181	33.47	1546	21	911	15	0
TLScanner.ti	107	4.50	5	29	0	0	0
TLCmdSyntax.ti	43	7.58	5	17	0	0	0
TLMinSyntax.ti	55	24.48	13	61	0	0	0
TLParser.ti	44	22.07	5	22	0	0	0
tlScanner.tm	383	16.43	1231	49	1008	19	0
tlCmdSyntax.tm	144	18.31	655	88	226	48	0
tlMinSyntax.tm	695	324.05	7081	367	2287	423	0
tlParser.tm	207	40.25	1525	89	330	82	0
TimeMeasure.ti	70	0.29	12	73	0	0	0
TMLEnv.ti	46	14.41	5	26	0	0	0
timeMeasure.tm	121	14.46	593	103	243	25	0
tmlEnv.tm	66	28.31	319	47	178	14	0
TLBuiltin.ti	20	7.45	3	11	0	0	0
TLInitialEnv.ti	68	19.82	11	46	0	0	0
TL2tml.ti	48	15.51	1	8	0	0	0
tlInitialEnv.tm	181	29.93	490	24	306	16	0
tlBuiltin.tm	674	232.66	4506	414	2105	287	0
tl2tml.tm	1450	418.67	7851	1101	3382	447	0
TLTopLevel.ti	69	16.41	16	58	0	0	0
tlTopLevel.tm	156	22.79	570	66	313	22	0
TLValueUnparse.ti	51	11.82	5	32	0	0	0
TyGlobal.ti	25	0.12	0	11	0	0	0
TyDoCommand.ti	52	31.89	4	24	0	0	0
TyRuntime.ti	49	0.73	6	24	0	0	0
TyExec.ti	31	34.46	3	15	0	0	0
TyTopLevel.ti	34	3.88	3	13	0	0	0
Tycoon.ti	18	0.13	3	9	0	0	0
tlValueUnparse.tm	472	100.35	2290	146	1150	70	0
tyGlobal.tm	23	0.76	40	3	20	0	0
tyRuntime.tm	92	13.81	144	18	142	1	0
tyExec.tm	315	79.49	923	23	779	5	0
tyDoCommand.tm	724	76.47	2749	211	1935	76	0
tyTopLevel.tm	190	83.97	986	71	424	35	0
tycoon.tm	144	19.23	612	21	308	21	0
insgesamt:	15769	3378.02	63554	11150	30269	2663	0

Abbildung I.26: Fortsetzung: Komponenten im Verzeichnis tlmin

Literaturverzeichnis

Abadi et al. 90: Abadi, M., Cardelli, L., Curien, P.-L., und Lévy, J.-J. „Explicit Substitutions". Technical Report 54, Digital Equipment Corporation, Systems Research Center, Palo-Alto, California, Februar 1990.

Abadi et al. 92: Abadi, M., Cardelli, L., Pierce, B., und Rémy, D. „Dynamic Typing in Polymorphic Languages". In: *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, Juni 1992.

Abadi, Cardelli 95: Abadi, M. und Cardelli, L. „On Subtyping and Matching". In: *Proceedings ECOOP'95*. Springer-Verlag, 1995.

Aho et al. 88: Aho, A.V., Sethi, R., und Ullmann, J.D. *Compilerbau.* Addison-Wesley Publishing Company, 1988.

Amadio, Cardelli 92: Amadio, R.M. und Cardelli, L. „Subtyping Recursive Types". *ACM Transactions on Programming Languages and Systems*, Jg. 15, September 1992, Nr. 4.

Appelrath et al. 92: Appelrath, H.-J., Lockemann, P.C., Neuhold, E., Reuter, A., Schek, H.-J., und Schweppe, H. „Interoperable Informationssysteme: Dienste für verteilte, datenintensive Anwendungen". (unveröffentlicht), Mai 1992.

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

Blaser 90: Blaser, A. (Hrsg.). *Database Systems of the 90s, Lecture Notes in Computer Science*, Bd. 466, November 1990.

Bruce et al. 95: Bruce, K.B., Schuett, A., und Gent, R. van. „PolyTOIL: a type-safe polymorphic object-oriented language". In: *Proceedings ECOOP'95*. Springer-Verlag, 1995.

Bruce 93: Bruce, K. „Safe type checking in a statically typed object-oriented language". In: *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, 1993.

Bruce 94: Bruce, Kim B. „A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics". *Journal of Functional Programming*, Jg. 4, April 1994, Nr. 2. A preliminary version appeared in POPL 1993 under the title "Safe Type Checking in a Statically Typed Object-Oriented Programming Language".

Canning et al. 89: Canning, P.S., Cook, W.R., Hill, W.L., und 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 et al. 91: Cardelli, L., Martini, S., Mitchell, J.C., und Scedrov, A. „An Extension of System F with Subtyping". In: Ito, T. und Meyer, A.R. (Hrsg.), *Theoretical Aspects of Computer Software, TACS'91*, Lecture Notes in Computer Science. Springer-Verlag, 1991, S. 750–770.

Cardelli et al. 94a: Cardelli, L., Matthes, F., und Abadi, M. „Extensible Grammars for Language Specialization". In: Beeri, C., Ohori, A., und Shasha, D.E. (Hrsg.), *Proceedings of the Fourth International Workshop on Database Programming Languages, Manhatten, New York*, Workshops in Computing. Springer-Verlag, Februar 1994, S. 11–31.

Cardelli et al. 94b: Cardelli, L., Matthes, F., und Abadi, M. „Extensible Syntax with Lexical Scoping". Technical Report 121, Digital Equipment Corporation, Systems Research Center, Palo-Alto, California, Februar 1994.

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

Cardelli 87: Cardelli, Luca. „Typechecking dependent types and subtypes". In: *Proc. of the Workshop on Foundations of Logic and Functional Programming*, Trento, Italy, Dezember 1987.

Cardelli 88a: Cardelli, L. „Phase Distinctions in Type Theory". Manuscript, 1988.

Cardelli 88b: Cardelli, L. „Structural Subtyping and the Notion of Power Type". In: *Proceedings of the Fifteenth ACM Symposium on Principles of Programming Languages, San Diego, California*, 1988.

Cattell 91: Cattell, R.G.G. „Next-Generation Database Systems". *Communications of the ACM*, Jg. 34, Oktober 1991, Nr. 10.

Cook 89: Cook, W. „A Proposal for Making Eiffel Type-Safe". In: *ECOOP 90 Proc. European Conference on Object Oriented Programming*, 1989.

Curien, Ghelli 91: Curien, P.-L. und Ghelli, G. „Subtyping + Extensionality: Confluence of $\beta\eta$top reduction in $F_{\leq}$". In: Ito, T. und Meyer, A.R. (Hrsg.), *Theoretical Aspects of Computer Software, TACS'91*, Lecture Notes in Computer Science. Springer-Verlag, 1991, S. 731–749.

Damas, Milner 82: Damas, L. und Milner, R. „Principal type-schemes for functional programs". In: *Proceedings of the Ninth ACM Symposium on Principles of Programming Languages*, 1982, S. 207–212.

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.

Field, Harrison 88: Field, A.J. und Harrison, P.G. *Functional Programming.* Addison-Wesley Publishing Company, 1988.

Gawecki, Matthes 95: Gawecki, A. und Matthes, F. „Integrating Query and Program Optimization Using Persistent CPS Representations". In: Atkinson, M.P. (Hrsg.), *Fully Integrated Data Environments.* Springer-Verlag (to appear), 1995.

Geisler 95: Geisler, A. „Basisdienste zur Gestaltung einer reflektiven grafischen Entwicklungsumgebung für eine persistente Programmiersprache". Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, 1995.

Ghelli 90: Ghelli, G. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism.* Dissertation, Università di Pisa, Dipartimento di Informatica, Maerz 1990.

Ghelli 95: Ghelli, Giorgio. „Divergence of $F_{\leq}$ Type Checking". *Theoretical Computer Science*, Jg. 139, 1995, Nr. 1,2, S. 131–162.

Girard 72: Girard, J.-Y. *Interprétation fonctionelle et élimination des coupures dans l'arithétique d'ordre supérieur.* Dissertation, Université Paris VII, 1972.

Hennessy 90: Hennessy, M. *The Semantics of Programming Languages.* John Wiley & Sons, 1990.

Hindley 69: Hindley, J. R. „The Principal Type-Scheme of an Object in Combinatory Logic". *Trans. AMS*, Jg. 146, 1969, S. 29–60.

Hofmann, Pierce 92: Hofmann, Martin und Pierce, Benjamin. „An Abstract View of Objects and Subtyping (Preliminary Report)". Technical Report ECS-LFCS-92-226, University of Edinburgh, LFCS, 1992.

Hofmann, Pierce 95: Hofmann, Martin und Pierce, Benjamin. „A Unifying Type-Theoretic Framework for Objects". *Journal of Functional Programming*, Jg. 5, Oktober 1995, Nr. 4, S. 593–635. Previous versions appeared in the Symposium on Theoretical Aspects of Computer Science, 1994, (pages 251–262) and, under the title "An Abstract View of Objects and Subtyping (Preliminary Report)," as University of Edinburgh, LFCS technical report ECS-LFCS-92-226, 1992.

Koehler 96: Koehler, H. „Generische Daten- und Funktionsvisualisierung in einer persistenten Programmierumgebung". Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Maerz 1996.

Leroy, Mauny 91: Leroy, X. und Mauny, M. „Dynamics in ML". Rapport de Recherche 1491, INRIA, Domaine de Voluceau, Rocquencourt 78153 Le Chesnay Cedex, France, Juli 1991.

MacQueen et al. 86: MacQueen, D.B., Plotkin, G.D., und Sethi, R. „An ideal model for recursive polymorphic types". *Information and Control*, Jg. 71, 1986, S. 95–130.

Mathiske et al. 93: Mathiske, B., Matthes, F., und Müßig, S. „The Tycoon System and Library Manual". DBIS Tycoon Report 212-93, Fachbereich Informatik, Universität Hamburg, Germany, Dezember 1993.

Mathiske et al. 95a: Mathiske, B., Matthes, F., und Schmidt, J.W. „On Migrating Threads". (Accepted for publication.), Juni 1995.

Mathiske et al. 95b: Mathiske, B., Matthes, F., und Schmidt, J.W. „Scaling Database Languages to Higher-Order Distributed Programming". (Submitted for publication.), Maerz 1995.

Mathiske 92: Mathiske, B. „Kodegenerierung für Programmiersprachen mit Persistenz, Polymorphie und Funktionen höherer Ordnung". Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Dezember 1992.

Matthes et al. 94: Matthes, F., Müßig, S., und Schmidt, J.W. „Persistent Polymorphic Programming in Tycoon: An Introduction". FIDE Technical Report FIDE/94/106, Fachbereich Informatik, Universität Hamburg, Germany, August 1994.

Matthes et al. 95a: Matthes, F., Müller, R., und Schmidt, J.W. „Towards a Unified Model of Untyped Object Stores: Experience with the Tycoon Store Protocol". In: Atkinson, M.P. (Hrsg.), *Fully Integrated Data Environments*. Springer-Verlag (to appear), 1995.

Matthes et al. 95b: Matthes, F., Schröder, G., und Schmidt, J.W. „Tycoon: A Scalable and Interoperable Persistent System Environment". In: Atkinson, M.P. (Hrsg.), *Fully Integrated Data Environments*. Springer-Verlag (to appear), 1995.

Matthes, Schmidt 91: Matthes, F. und Schmidt, J.W. „Bulk Types: Built-In or Add-On?". In: *Database Programming Languages: Bulk Types and Persistent Data*. Morgan Kaufmann Publishers, September 1991.

Matthes, Schmidt 92: Matthes, F. und 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.

Matthes, Schmidt 95: Matthes, F. und Schmidt, J.W. „Persistent Threads". In: Atkinson, M.P. (Hrsg.), *Fully Integrated Data Environments*. Springer-Verlag (to appear), 1995. (Reprint of [MaSc94]).

Matthes 93: Matthes, F. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung*. Springer-Verlag, 1993. (In German.).

Matthews 87: Matthews, D. „Static and Dynamic Type Checking". In: *Proceedings of the First International Workshop on Database Programming Languages, Roscoff, Finistere, France*, September 1987, S. 43–52.

Meyer, Reinhold 86: Meyer, Albert R. und Reinhold, Mark B. „"Type" is not a Type: Preliminary Report". In: *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*. ACM, ACM, Januar 1986, S. 287–295.

Milner 78: Milner, R. „A Theory of Type Polymorphism in Programming". *Journal of Computer and System Sciences*, Jg. 17, 1978, S. 348–375.

Pierce, Turner 94: Pierce, Benjamin C. und Turner, David N. „Simple Type-Theoretic Foundations for Object-Oriented Programming". *Journal of Functional Programming*, Jg. 4,

April 1994, Nr. 2, S. 207–247. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

Pierce 92a: Pierce, B. C. „Bounded Quantification is Undecidable". In: *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages*, Januar 1992, S. 305–315.

Pierce 92b: Pierce, Benjamin C. „F-Omega-Sub User's Manual Version 1.0". Documentation, Computer Science Dept., University of Edinburgh, Oktober 1992.

Reynolds 74: Reynolds, J.C. „Towards a theory of type structure". In: *Colloquium sur la programmation, Lecture Notes in Computer Science*, Bd. 19. Springer-Verlag, 1974, S. 408–423.

Solomon 78: Solomon, M. „Type Definitions with Parameters". In: *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, Tucson, Arizona*, Januar 1978, S. 31–38.

Steffen, Pierce 94: Steffen, Martin und Pierce, Benjamin. „Higher-Order Subtyping". Technical Report ECS-LFCS-94-280, LFCS, University of Edinburgh, Januar 1994. Also available as Universität Erlangen-Nürnberg Interner Bericht IMMD7-01/94. To appear in Theoretical Computer Science.

Erklärung

Ich versichere hiermit, die vorliegende Arbeit selbständig und ausschließlich unter Zuhilfenahme der angegebenen Quellen durchgeführt zu haben.

Ort, Datum	Gerd Bremer