FR Informatik
Fakultät für Mathematik und Informatik
Universität des Saarlandes

ML mit Typklassen

Diplomarbeit (überarbeitete Fassung)

Angefertigt unter der Leitung von Prof. Dr. Gert Smolka

Gerhard Schneider

4. Juni 2000

Leitung:	Prof. Dr. Gert Smolka
Erstgutachter:	Prof. Dr. Gert Smolka
Zweitgutachter:	Prof. Dr. Reinhard Wilhelm
Betreuung:	Dipl.-Inform. Andreas Rossberg
	Dipl.-Inform. Leif Kornstaedt

Hiermit erkläre ich, Gerhard Schneider, daß ich die vorliegende Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Saarbrücken, 31. Mai 2000

Zusammenfassung

Diese Arbeit beschreibt Design und Implementierung von TML. TML vereinigt das Typklassenkonzept von Haskell mit Standard ML. Haskell und Standard ML sind statisch getypte funktionale Programmiersprachen. Im Gegensatz zu Haskell verfügt ML über ein expressives, parametrisches Modulsystem, das die Entwicklung großer Softwaresysteme hervorragend unterstützt. Andererseits verfügt Haskell mit dem Typklassenkonzept über einen expressiven parametrischen Polymorphismus, der ML fehlt.

Bei TML handelt es sich um eine Erweiterung von SML um Typklassen, die mit dem Modulsystem von ML verträglich ist. Durch die Integration ins Modulsystem bieten sich interessante Programmiertechniken, die so weder in Haskell, noch in SML möglich sind.

Danksagung

Hiermit möchte ich mich bei Prof. Dr. Gert Smolka und seinen Mitarbeitern bedanken. Der Lehrstuhl bot neben einer ausgezeichneten Infrastruktur ein sehr angenehmes Umfeld zum Arbeiten.
Meine besonderer Dank gilt Leif Kornstaedt und Andreas Rossberg, die meine Diplomarbeit hervorragend betreut haben.
Außerdem möchte ich Thorsten Brunklaus und Andreas Simon für die sehr hilfreiche Kritik danken.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Typsysteme von Programmiersprachen

Typsysteme dienen in Programmiersprachen dazu, verschiedene Sorten von Werten zu unterscheiden. Mit dieser Unterscheidung können Programme in zwei Klassen aufgeteilt werden: In sogenannte wohlgeformte Programme, die vom Typsystem akzeptiert werden und, in Programme, die das Typsystem zurückweist. Es gibt viele Arten von Typsystemen. Allesamt besitzen aber dieselben prinzipiellen Vorteile:

- Statische Typsysteme bieten automatische Fehlererkennung zur Übersetzungszeit: zum Beispiel darf eine ganze Zahl nicht mit einer Zeichenkette addiert werden.
- Verbessertes Laufzeitverhalten: Steht der Typ einer Berechnung schon zur Übersetzungszeit fest, so können kostspielige Laufzeittests entfallen.
- Typen besitzen dokumentarischen Charakter.

Der Hauptnachteil von Typsystemen besteht darin, daß sie nie vollständig sind. Das heißt es wird immer Programme geben, die zurückgewiesen werden, obwohl sie bei der Ausführung ohne Einbeziehung der Typinformation zu sinnvollen Resultaten führen. Zum Beispiel hat der Ausdruck

```
1 + (if true then 1 else "foo")
```

ein wohldefiniertes Resultat. Trotzdem weisen die meisten existierenden Typsysteme den Ausdruck als ungültig zurück.

Die Hauptansprüche an Typsysteme sind somit einerseits, daß ungültige Programme zurückgewiesen werden, also die Korrektheit des Typsystems. Andererseits sollte ein Typsystem möglichst vollständig sein, das heißt es sollte die meisten sinnvollen Programme akzeptieren.

1.2 Polymorphe Typsysteme und Typinferenz

Polymorphe Typen spezifizieren Familien von Typen. Zum Beispiel ist es für die Berechnung der Länge einer Liste nicht von Bedeutung, den Elementtyp der Liste zu kennen.

```
fun len nil = 0
  | len (x :: xr) = 1 + len xr
```

Deshalb wird der Längenfunktion der polymorphe Typ

$$\forall\alpha.\alpha\ list \rightarrow int$$

zugewiesen. Das α steht dabei als Platzhalter für einen beliebigen Typ und erlaubt dadurch die Längenfunktion auf beliebige Listen anzuwenden.

Polymorphe Typsysteme haben ihren Ursprung in dem Typsystem von Hindley [Hin69]. Milner hat dieses später wiederentdeckt und einen Algorithmus für eine auf dem Lambda-Kalkül basierende Programmiersprache in [Mil78] vorgestellt, der den Typ eines Programms ableiten kann, obwohl das Programm keine oder nur teilweise Typannotationen enthält. Ein einfaches Beispiel ist die Folgerung, daß der Typ des Bezeichners `x` in folgendem Ausdruck *int* sein muß.

```
x + 1
```

Im Allgemeinen leitet der Algorithmus von Milner den sogenannten *prinzipalen* Typ her [DM82]. Der prinzipale Typ ist der allgemeinste Typ, der einem Programm zugewiesen werden kann. Im Beispiel der Längenfunktion gerade $\forall\alpha.\alpha\ list \rightarrow int$.

1.3 Typklassen

Oft reichen polymorphe Typen in der Praxis nicht aus. Zum Beispiel kann man mit Hilfe von polymorphen Typen die Funktion `double` nicht zufriedenstellend typisieren.

```
fun double x = x + x
```

Weist man ihr den Typ $int \rightarrow int$ zu, so kann sie nicht auf Gleitkommazahlen angewendet werden. Man darf ihr aber auch nicht den Typ $\forall\alpha.\alpha \rightarrow \alpha$ zuweisen, da sie dann auf Werte angewendet werden könnte, auf denen keine Addition definiert ist.

Der Grund für diese Schwäche polymorpher Typsysteme liegt darin, daß sie entweder *alle* Typen oder nur *einen* erlauben. Bei der Funktion `double` muß aber unterschieden werden, ob auf dem Argumenttyp die arithmetische Operation `+` definiert ist oder nicht.

Die Typklassen von Haskell [HJW+92] bieten ein generisches Mittel, um ein Typsystem um sogenannten *qualifizierten parametrischen Polymorphismus* zu erweitern. Allgemeiner definiert eine *Typklasse* eine Menge von Methoden über einem unbekannten Typ. Eine Typklasse kann abstrakt als eine Menge von Typen aufgefaßt werden, auf die die Methoden der Klasse angewendet werden können. Ein Typ ist ein *Element der Klasse*, wenn eine Implementierung der Methoden für diesen Typ vorliegt. Eine solcher Typ wird auch als *Instanz* der Klasse bezeichnet.

Ein Standardbeispiel ist die Typklasse, deren Instanzen genau die Typen sind, auf denen Gleichheit definiert ist. Die folgende Klassendeklaration zeigt, wie man dieses in einer konkreten Syntax hinschreiben könnte.

```
class 'a EQ with
  val == : 'a * 'a -> bool
end
```

Die Klasse besteht aus zwei Teilen:

- Der Ausdruck `'a EQ` in der ersten Zeile führt den Klassenbezeichner `EQ` ein und kennzeichnet `'a` als Platzhalter für die Typen, die eine Instanz der Klasse sind.
- In der zweiten Zeile ist die Typsignatur der einzigen Methode der Klasse angegeben. Die Typsignatur drückt aus, daß für jede Instanz `'a` der Klasse `EQ` der Gleichheitsoperator als eine Methode definiert ist, die zwei Argumente vom Typ `'a` nimmt und einen Boole'schen Wert zurückgibt.

Außerhalb der Klassendeklaration wird der Operator wie eine Funktion verwendet. Er hat den Typ:

$$\forall\alpha\ \mathrm{EQ}.\alpha \times \alpha \rightarrow \mathit{bool}$$

Der Klassenkontext auf der Klassentypvariable besagt, daß die Typen, die für α eingesetzt werden dürfen, eine Instanz der Klasse `EQ` sein müssen.

Eine Implementierung der Methoden der Klasse `EQ` kann mit einer sogenannten *Instanzdeklaration* angegeben werden. Eine Instanz für ganze Zahlen der Klasse `EQ` könnte so deklariert werden:

```
instance int EQ with
  val op== = Int.eq
end
```

Bei der Definition von Funktionen wird die Verwendung von Methoden von Typklassen im Typ der Funktion sichtbar. Zum Beispiel hat der Elementtest auf Listen

```
fun member y nil = false
  | member y (x :: xr) = y == x orelse member y xr
```

den Typ:

$$\forall\alpha \text{ EQ}.\alpha \times \alpha \; list \rightarrow int$$

1.4 Standard ML

Standard ML [MTHM97] ist eine funktionale Programmiersprache, die ursprünglich entwickelt wurde, um im Bereich der Logik für computergestütztes Beweisen eingesetzt zu werden.

Sie ist statisch getypt und war die erste Programmiersprache mit polymorphen Typen. Das Typsystem von SML verwendet den Typinferenzalgorithmus von Hindley/Milner, um die Typen in der Kernsprache automatisch herzuleiten.

SML verfügt außerdem über ein ausgefeiltes parametrisches Modulsystem. Insbesondere wurde das automatische Herleiten von Typinformation auf das Modulsystem erweitert. SML leitet automatisch die Signatur einer Struktur her und überprüft beim Kombinieren verschiedener Module, ob diese auch zusammenpassen. Diese Signaturtechnik bewahrt einen Programmierer davor, Module fehlerhaft zu verwenden und qualifiziert SML für die Entwicklung großer Software-Systeme.

In SML entspricht ein Modul einer Struktur. Eine Struktur ist ein Paket von Deklarationen zusammengehöriger Typen, Werte und Funktionen. Mit Signaturen kann man spezifizieren, welche Komponenten eine Struktur enthalten muß. Strukturen können gegen Signaturen abgeglichen werden. Dabei stehen zwei Arten von *Abgleich* zur Verfügung: Beim *transparenten* Abgleich scheinen enthaltene Typen durch und man kann auf die Implementierung dieser Typen zugreifen. Beim *opaken* Abgleich können Typen abstrakt gemacht werden, das heißt die Implementierung wird verborgen. Durch opaken Signaturabgleich werden so neue Typen erzeugt. In der Praxis werden Signaturen dazu verwendet, die Schnittstelle eines Moduls exakt zu definieren.

Zudem gibt es in SML das Konzept eines Funktors. Funktoren sind Funktionen über Strukturen. Ein Funktor ermöglicht es, Strukturen über andere Strukturen zu parametrisieren.

Das folgende Beispiel zeigt einen Funktor, mit dem sich Mengen über beliebigen geordneten Typen realisieren lassen:

```
functor Set (Elem : ORDERED) :> SET =
  struct
     type set = Elem.t list
     type elem = Elem.t
     val empty = nil
     fun insert (x, set) = ...
     fun member (x, set) = ...
  end
```

Die Signatur `ORDERED` spezifiziert die minimalen Komponenten der Parametersignatur, welche den Elementtyp festlegt und die Signatur `SET` beschreibt die Schnittstelle der resultierenden Struktur. Die Schreibweise `:>` steht dabei für opaken Signaturabgleich, womit hier die Implementierung von Mengen als Listen verborgen werden kann.

1.5 Beitrag dieser Arbeit

SML verfügt nicht über qualifizierten parametrischen Polymorphismus. Lediglich für die Gleichheit gibt es eine spezielle Form von Typvariablen, die nur Typen zulassen, auf denen Gleichheit definiert ist.

Diese Arbeit beschreibt die Integration von Typklassen in SML. Der dafür entwickelte Sprachdialekt wird im folgenden mit TML bezeichnet. Dabei wird detailliert auf folgende Aspekte eingegangen:

Integration in die Kernsprache. Die Erweiterung der Kernsprache orientiert sich an der Programmiersprache Haskell, in der Typklassen bereits seit mehreren Jahren fester Bestandteil sind.

Integration in das Modulsystem. Bisher wurde offenbar noch nie versucht, Typklassen in ein so komplexes Modulsystem zu integrieren, wie es SML besitzt. Im Vergleich mit SML besitzt Haskell nur ein sehr primitives Modulsystem. Klassen und Instanzen dürfen in Haskell nur auf oberster Ebene deklariert werden.

Diese Arbeit zeigt, daß man SML um Typklassen so erweitern kann, daß die Erweiterung verträglich mit dem Modulsystem bleibt.

Bei der Integration ins Modulsystem wurde besonders auf Vollständigkeit wert gelegt: Klassen und Instanzen können in Strukturen und im Rumpf eines Funktors deklariert werden. Es ist möglich, Klassen und Instanzen ähnlich wie Datentypen zu replizieren. Die Signaturspezifikationen wurden um Spezifikation von Klassen und Instanzen erweitert. Analog zu abstrakten Typen können Klassen abstrakt spezifiziert werden.

Definition der statischen Semantik. Da bisher keine Integration von Typklassen in ein Modulsystem existiert, gibt es dafür auch keine Definition der statischen Semantik. Selbst für die Typklassen von Haskell wurde die statische Semantik nur für eine Teilsprachen in [JW92] formalisiert.

In dieser Arbeit wird die statische Semantik für TML vollständig im Stile der Definition von SML [MTHM97] angegeben.

Dynamische Semantik. Zur Erklärung der dynamischen Semantik von TML wird eine Teilsprache von TML auf SML zurückgeführt. In der Teilsprache sind dabei die für Typklassen relevanten Teile des Modulsystems enthalten. Die Rückführung geschieht durch Angabe von formalen Transformationsregeln, die den exakten Bezug zur statischen Semantik herstellen.

Insbesondere für Implementatoren ist dies sehr nützlich, da man an den Regeln genau ablesen kann, wie die grundlegenden Ideen der Transformationen aussehen und welche Typinformationen dazu benötigt werden.

Übersetzer. Im Rahmen dieser Arbeit wurde ein Übersetzer für TML entwickelt, um die Ausdrucksmöglichkeiten von TML praktisch erproben zu können. Dabei wurde auf Effizienz keinen Wert gelegt. Der Übersetzer war die Vorlage für die Definition der statischen und dynamischen Semantik.

1.6 Verwandte Arbeiten

Historisch gehen die ersten Überlegungen zu Typklassen auf Stefan Kaes [Kae88], sowie Wadler und Blott [WB89] zurück. Ziel dieser Arbeiten war die Entwicklung einer generischen Methode zur Behandlung überladener Funktionen, wie + und =.

Als generisches Mittel zum Überladen benutzerdefinierter Funktionen fanden Typklassen erstmals Einzug in eine vollwertige Programmiersprache beim Design von Haskell [HJW+92].

In [OWW95] kritisieren die Autoren, daß die dynamische Semantik eines Programmes von der statischen Semantik abhängt. Sie stellen einen weiteren Ansatz vor, bei dem die statische Semantik wieder von der dynamischen Semantik getrennt werden kann. Dabei geht aber ein Großteil der Expressivität verloren.

Formalisierungen von Teilsprachen von Haskell findet man in [JW92]und [HHPW96]. Weitere Arbeiten zur Formalisierung von Typsystemen mit Typklassen sind [NP93], [NS91] und [NP95].

In [JHS98] wird die Integration von Typklassen in die logische Programmiersprache Mercury diskutiert.

Ein umfassender Überblick über qualifizierte Typen, der theoretischen Grundlage des Typklassenkonzepts, wird in [Jon92] gegeben.

In [JJM97] werden ausführlich Erweiterungen wie zum Beispiel Multi-Parameter-Typklassen diskutiert.

1.7 Aufbau dieser Arbeit

In Kapitel 2 wird informal beschrieben, wie sich das Konzept der Typklassen in SML integrieren läßt. Insbesondere wird dabei auf die Erweiterungen im Modulsystem und die Probleme, die dadurch entstehen, eingegangen. Das Kapitel endet mit einer Gegenüberstellung von Typklassen mit den bestehenden Konzepten von SML und der Behandlung eines Funktors, der eine ähnliche Expressivität ermöglicht wie die in [Läu94] beschriebenen existentiell quantifizierten Typen.

In Kapitel 3 wird eine formale Definition der Syntaxerweiterungen aufbauend auf der Sprachdefinition von Standard ML [MTHM97] angegeben.

In Kapitel 4 wird formal, im Stile der Definition von Standard ML, die statische Semantik definiert. Im wesentlichen lassen sich Typklassen in Bezug auf die statische Semantik orthogonal in ML einbauen. Es gibt keine Konflikte, die einen Konzeptionswechsel bei der Definition der statischen Semantik erfordern würden.

In Kapitel 5 wird die dynamische Semantik von TML durch Rückführung auf dynamisch getyptes SML formalisiert. Im Gegensatz zur Definition von Standard ML kann die dynamische Semantik nicht mehr ohne die Verwendung von Typinformationen definiert werden.

Im Rahmen der Arbeit ist eine Prototypimplementierung von TML entstanden, die die in Kapitel 5 vorgestellten Transformationen implementiert. In Kapitel 6 werden die wichtigsten Aspekte der Implementierung vorgestellt.

Kapitel 7 bietet einen Ausblick auf mögliche Erweiterungen von TML.

Kapitel 2

Typklassen in TML

Dieses Kapitel beschreibt informal Typklassen in TML. In den ersten beiden Abschnitten wird die Integration in die Kernsprache und in das Modulsystem vorgestellt. Anschließend wird der Zusammenhang zwischen Typklassen und dem Modulsystem, sowie Gleichheit in ML ohne und mit Typklassen, diskutiert. In dem folgenden Abschnitt wird ein Problem diskutiert, daß durch die Integration von Typklassen in das Modulsystem verursacht wird. Das Kapitel endet mit der Vorstellung eines speziellen Funktors, der eine ähnliche Ausdruckskraft ermöglicht, wie existentiell quantifizierte Typen.

2.1 Integration von Typklassen in die Kernsprache

Dieser Abschnitt stellt zunächst kurz die Idee und das Konzept von Typklassen vor. Anschließend wird die Verwendung von Typkassen in der Kernsprache diskutiert. Dabei werden die für Typklassen benötigten Spracherweiterungen eingeführt.

2.1.1 Was sind Typklassen?

ML ist eine statisch getypte Programmiersprache. Zum Beispiel hat die Nachfolgerfunktion auf ganzen Zahlen den Typ `int -> int`.

```
fun succ x = x + 1
```

Es gibt aber auch ML-Funktionen, bei denen man den Typ nicht eindeutig festlegen kann. Ein Beispiel dafür ist die Längenfunktion auf Listen. Mit ihr kann man sowohl die Länge einer Zahlenliste, als auch die Länge einer Liste von Zeichenketten berechnen.

```
fun length (_ :: xr) = 1 + length xr
  | length nil       = 0
```

Es wäre viel zu restriktiv, den Typ der Längenfunktion etwa auf `int list -> int` festzulegen. In ML werden solchen Funktionen allgemeinere Typen zugewiesen. Genauer wird ein ihnen ein sogenanntes *Typschema* zugeordnet. Ein Typschema ist ein Typ, in dem die Komponenten über die man keine Aussage treffen möchte, durch *Typvariablen* ersetzt sind. Eine Typvariable steht als Platzhalter für einen beliebigen Typ. In der Syntax von ML werden Typvariablen durch ein Apostroph gekennzeichnet, etwa `'a`.

Für die Längenfunktion erhält man somit das folgende Typschema:

```
length : 'a list -> int
```

Die Längenfunktion kann nur in Ausdrücken verwendet werden, wenn eine gültige Substitution der Typvariablen durch konkrete Typen existiert, so daß der Ausdruck typkorrekt ist. Das Ersetzen der Typvariablen bezeichnet man als *Instantiieren* und die Typen, die durch Instantiieren entstehen, als *Instanzen* des Typschema. Zum Beispiel wird in `length [1]` die Typvariable im Typschema der Längenfunktion mit `int` instantiiert.

Für den Rest dieses Kapitels wird zwischen Typen und Typschemata der Einfachheit halber nicht mehr unterschieden.

Eine weitere Besonderheit im Typsystem von ML ist die Unterscheidung zwischen Typen, auf deren Werte der Gleichheitstest zugelassen ist, und Typen, auf denen keine Gleichheit definiert ist. Daß diese Unterscheidung Sinn macht, sieht man daran, daß man zum Beispiel Zahlen miteinander vergleichen kann, aber für Funktionen mit dem Typ `int -> int` die Gleichheit im Allgemeinen unentscheidbar ist.

Diese Unterscheidung verkompliziert beispielsweise die Typisierung des Elementtests:

```
fun member x (y :: yr) = (x = y) orelse (member x yr)
  | member x nil       = false
```

Es ist klar, daß der Elementtest nicht den Typ `'a * 'a list -> bool` haben kann, denn dann müßte man auch zwei Funktionen mit dem Typ `int-> int` vergleichen können.

Aus diesem Grund gibt es in ML eine zweite Sorte Typvariablen. Diese Typvariablen werden als *Gleichheitstypvariablen* bezeichnet und können ausschließlich mit Typen, auf denen Gleichheit definiert ist, instantiiert werden. Gleichheitstypvariablen werden in ML-Syntax mit zwei Apostrophen am Anfang notiert, wie etwa `''a`.

Damit kann man den Elementtest für Listen wie folgt typisieren:

```
member : ''a * ''a list -> bool
```

Allgemeiner lassen Gleichheitstypvariablen genau die Typen zu, auf denen die Funktion = definiert ist. Man kann also Gleichheitstypvariablen mit der Menge aller Typen identifizieren, auf deren Werte man die Funktion = anwenden kann.

Typklassen verallgemeinern diese Sichtweise. Eine *Typklasse* spezifiziert eine Menge von Typen, auf denen bestimmte Funktionen definiert sind. Zum Beispiel wird durch die folgende Typklasse `EQ` die Menge aller Typen spezifiziert, für die die Funktion `==` definiert ist.

```
class 'a EQ with
  val == : 'a * 'a -> bool
end
```

Ein Typ `t` ist ein Element einer Typklasse, wenn für ihn eine Implementierung der Funktion `==` existiert und diese Implementierung den Typ `t * t -> bool` hat. Ein solcher Typ wird auch als *Instanz* der Klasse `EQ` bezeichnet.

Betrachten wir noch einmal den Elementtest auf Listen. Dieses Mal wird für die Implementierung der Gleichheitstest `==` aus der Klasse `EQ` verwendet.

```
fun member x (y :: yr) = (x == y) orelse (member x yr)
  | member x nil       = false
```

Der obige Elementtest auf Listen wird durch die Verwendung von `==` aus der Klasse `EQ` auf Listen beschränkt, deren Elementtyp eine Instanz der Klasse `EQ` ist. Deswegen wird die Syntax für Typen um einen sogenannten *Klassenkontext* erweitert.

Zum Beispiel spezifiziert `'a EQ => 'a -> int` eine Funktion, deren Argumenttyp eine Instanz der Klasse `EQ` ist. Hier ist `'a EQ` der Klassenkontext.

Mit der neuen Typsyntax erhalten wir für den Elementtest den Typ:

```
member : 'a EQ => 'a * 'a list -> bool
```

Um mit der Funktion `member` zu testen, ob die Zahl `1` ein Element der Liste `[2, 3, 4]` ist, benötigt man eine Implementierung der Funktion `==` auf ganzen Zahlen. Diese kann mit Hilfe einer *Instanzdeklaration* angegeben werden.

```
instance int EQ with
  fun x == y = Int.EQ (x, y)
end
```

2.1.2 Deklaration einer Typklasse

Abbildung 2.1 zeigt den allgemeinen Aufbau einer Typklassendeklaration. Eine *Typklassendeklaration* besteht aus zwei Teilen, nämlich dem *Kopf* und dem *Rumpf*.

Abbildung 2.1 Aufbau einer Klassendeklaration

```
class tyvar classid with
  methspec
end
```

Im Kopf steht eine Typvariable gefolgt von einem *Klassenbezeichner*. Die Typvariable im Kopf wird als *Klassentypvariable* bezeichnet. Im Rumpf werden die Funktionen einer Klasse spezifiziert. Die Funktionen einer Klasse werden als *Methoden* bezeichnet. Der Typ jeder Methodenspezifikation muß die Klassentypvariable mindestens einmal enthalten.

Eine Klassendeklaration führt die Klasse und alle Methoden in den Skopus ein. Die im Kopf deklarierte Klassentypvariable wird durch die Klassendeklaration gebunden und ist in den Methodenspezifikationen frei.

Die Klasse `EQ` in folgendem Beispiel deklariert zwei Methoden `==` und `!=`.

```
class 'a EQ with
  val == : 'a * 'a -> bool
  val != : 'a * 'a -> bool
end
```

In der Verwendung unterscheiden sich Methoden nicht von normalen Werten. Sie können zum Beispiel nach der Klassendeklaration zur Definition einer Funktion verwendet werden.

```
fun member y nil = false
  | member y (x :: xr) = (x == y) orelse (member y xr)
```

Bei der Applikation obiger Funktion `member` auf Objekte mit einem konkreten Typ, muß natürlich eine Implementierung der Methode `==` für den entsprechenden Typ vorliegen. Diese wird jeweils mit einer sogenannte Instanzdeklaration angegeben.

2.1.3 Deklaration von Instanzen

Instanzen dienen dazu, Implementierungen der Methoden einer Klasse für einen bestimmten Typ anzugeben. Eine Instanz besteht wie eine Klasse aus einem *Kopf* und einem *Rumpf*.

Im Kopf steht ein sogenannter *Instanztyp* und der Klassenbezeichner. Im Rumpf werden die Implementierungen der Methoden der im Kopf bezeichneten Klasse angegeben. Abbildung 2.2 zeigt den Aufbau einer Instanz.

Abbildung 2.2 Instanzdeklaration

```
instance instty classid with
  methdec
end
```

Der Instanztyp beschreibt die Menge der Typen für die die Implementierung der Methoden verwendet werden soll. Abbildung 2.3 zeigt die erlaubten Instanztypen. Dabei gilt die syntaktische Einschränkung, daß Typvariablen und Labels paarweise disjunkt sein müssen.

Abbildung 2.3 Instanztypen

```
('a1, ..., 'ak) tycon
('a1 -> 'a2)
{label1 : 'a1, ..., labelk : 'ak}
```

Zum Beispiel ist `('a, 'a) pair` kein gültiger Instanztyp, da die Typvariable `'a` mehrfach vorkommt.

Ein Typ *paßt* auf einen Instanztyp, wenn eine Substitution der Typvariablen im Instanztyp existiert, so daß das Ergebnis der Substitution gleich dem Typ ist. Wird eine Instanz für einen konkreten Typ benötigt, so wird diejenige Instanz gewählt, auf deren Instanztyp der Typ paßt.

Zwei Instanztypen sind gleich, wenn sie sich bis auf Umbenennung der Typvariablen und Vertauschung der Labels nicht unterscheiden. Des weiteren besitzen Instanztypen folgende wichtige Eigenschaft: Paßt ein Typ auf zwei Instanztypen, so sind diese gleich.

Ist im Skopus einer Instanzdeklaration bereits eine Instanz mit gleichem Instanztyp für dieselbe Klasse definiert, so verdeckt die Instanzdeklaration die vorherige Instanz. Dadurch ist es ausgeschlossen, daß für einen Typ zwei Instanzen im Bezugsrahmen stehen, auf deren Instanztyp der Typ paßt. Das heißt, man kann jedem Typ höchstens eine Instanz zuordnen.

Bei der Applikation einer Methode wird die Instanz verwendet, auf die der konkrete Typ paßt, der für die Klassentypvariable im Methodentyp instantiiert wurde.

Bei der Implementierung der Methoden muß darauf geachtet werden, daß der Typ der Implementierung die Spezifikation der Klasse erfüllt. Eine Implementierung einer Methode erfüllt die Spezifikation, wenn der Typ der Implementierung polymorpher ist als der spezifizierte Typ nach dem Einsetzen des Instanztyps für die Klassentypvariable. Man darf die Instanztypfunktion für die Klassentypvariable einsetzen, da die Implementierung nur für Typen verwendet wird, die auf den Instanztyp passen, und diese somit mindestens die Struktur des Instanztyps besitzen.

In folgendem Beispiel wird die Klasse `EQ` für den Typ `int` instantiiert. Insbesondere ist hier der Typ von `== : int * int -> int`. Es wird also ausgenutzt, daß durch die Instanztypfunktion bekannt ist, daß diese Implementierung nur für Applikationen verwendet wird, bei denen die Klassentypvariable als Platzhalter für den Typ `int` steht.

```
instance int EQ with
  val op== = Int.EQ
  fun x != y = not (x == y)
end
```

2.1.4 Applikation von Methoden

Bei der Applikation von Methoden können zwei Fälle auftreten:

Statische Bindung. In diesem Fall wird die Instanz bereits zur Übersetzungszeit statisch gebunden. Dies geschieht immer, wenn zur Übersetzungszeit anhand der Typinformation die passende Instanz eindeutig feststeht. Zum Beispiel wird bei der Applikation `1 == 2` die `int`-Instanz der Klasse `EQ` eingesetzt.

Dynamische Bindung. In dem folgenden Fall kann nicht mit Hilfe der Typinformation entschieden werden, welche Instanz zu verwenden ist. Deshalb wird die Instanz zur Laufzeit in Abhängigkeit vom Argumenttyp von `f` dynamisch gebunden.

```
val f = fn x => x == x
```

Etwa bei der Applikation von `f` auf `"Orca"` wird die `string` Instanz der Klasse `EQ` dynamisch eingesetzt. Dabei hat die Funktion `f` den Typ den Typ `'a EQ => 'a -> bool`.

Dynamische Bindung korrespondiert also genau damit, daß das zugeordnete Typschema einen Kontext trägt.

2.1.5 Abgeleitete Instanzen

Als nächstes soll Gleichheit auf Listen definiert werden. Nach Definition sind zwei Listen genau dann gleich, wenn sie dieselben Elemente in derselben Reihenfolge enthalten. Die Gleichheit von Listen wird also auf die Gleichheit ihrer Elemente zurückgeführt.

Diese Bedingung wird im Kopf einer Instanz, im sogenannten *Instanzkontext*, festgehalten. Abbildung 2.4 zeigt den Aufbau einer Instanzdeklaration mit Kontext. Ein Kontext im Instanzkopf besteht aus einer Sequenz von Paaren von Typvariable und Klassenbezeichner. Jede Typvariable im Kontext muß einmal im Instanztyp vorkommen.

Abbildung 2.4 Aufbau einer abgeleitete Instanz

```
instance context => instty classid with
  methdec
end
```

Das folgende Beispiel zeigt die Instantiierung der Klasse `EQ` für Listen. Der Instanzkontext fordert, daß der Typ der Listenelemente bereits eine Instanz von `EQ` ist. Deswegen darf die Implementierung für den Listenvergleich den Gleichheitstest `==` auf den Listenelementen verwenden.

```
instance 'a EQ => 'a list EQ with
  fun (x :: xr) == (y :: yr) = x == y andalso xr == yr
    |       nil == nil       = true
    |         _ == _         = false
  fun (x != y) = not (x == y)
end
```

Genauer ist die Instanz für Listen über die Instanz der Klasse `EQ` für den Typ der Listenelemente parametrisiert.

Wird zum Beispiel eine Instanz für Zahlenlisten benötigt, so kann diese aus obiger Instanz abgeleitet werden. Dazu wird der Vergleich der Listenelemente zur Übersetzungszeit durch einen Dispatch an die entsprechende Methode in der `int`-Instanz der Klasse `EQ` ersetzt.

2.1.6 Rekursive Instanzen

Um Instanzen über verschränkt rekursive Datentypen, wie zum Beispiel

```
type var = int

datatype exp =
    VarExp of var
  | AbsExp of var * exp
  | AppExp of exp * exp
  | LetExp of dec * exp

and dec = Dec of var * exp
```

definieren zu können, werden verschränkt rekursive Instanzen benötigt. Verschränkt rekursive Instanzen werden über eine mit `and` verkettete Folge von Instanzdeklarationen definiert.

Folgendes Beispiel definiert eine Klasse mit einer Methode zur Berechnung der freien Vorkommen von Bezeichnern. Die Instanzen müssen verschränkt rekursiv sein, damit in der Instanz für Ausdrücke die Instanz für Deklarationen und umgekehrt verwendet werden kann.

```
class 'a FREE_VARS with
  val freeVars : 'a -> int list
end


local
  fun boundVar (Dec (var, exp)) = var
in
  instance exp FREE_VARS with
    fun freeVars (VarExp var) = [var]

      | freeVars (AbsExp (var, exp)) =
        List.filter (fn var' => var != var') (freeVars exp)

      | freeVars (AppExp (exp, exp')) =
        freeVars exp @ freeVars exp'
```

```
        | freeVars (LetExp (dec, exp)) =
          let
            val bv = boundVar dec
            val fv = List.filter
                       (fn var => var != bv)
                       (freeVars exp)
          in
            freeVars dec @ fv
          end
    end
    and dec FREE_VARS with
      fun freeVars (Dec (var, exp)) =
          List.filter (fn var' => var != var') (freeVars exp)
    end
  end
```

Konsequenterweise sind Instanzen immer rekursiv zu sich selbst. Dies führt dazu, daß die beiden folgenden Instanzdeklarationen äquivalent sind.

```
instance t EQ with
  val op== = fn (x, y) = x == y
end

instance t EQ with
  fun x == y = x == y
end
```

2.1.7 Einschränkungen

Klassen und Instanzen dürfen nur auf Strukturebene deklariert werden. Das heißt, sie dürfen nicht in `let`-Ausdrücken deklariert werden.

Um Klassen- und Instanzdeklarationen in `let`-Ausdrücken zu erlauben, müßte von der statischen Semantik gefordert werden, daß sie bei jeder Wertbindung die prinzipalen Typen herleitet. Für das folgende Programm existiert aufgrund der Wertrestriktion kein prinzipaler Typ für `x`. Damit wäre das Programm unzulässig.

```
instance 'a EQ => 'a list EQ ...

let
  local
    instance int EQ ...
  in
    val x = (fn () => nil == nil) ()
  end
in
  3 :: x
end
```

Da derartige Ausdrücke in der Praxis sehr oft auftreten, müßten häufig Typannotationen vorgenommen werden. Dies ist aber nicht wünschenswert.

Das äquivalente Problem tritt auf Strukturebene auf.

```
instance 'a EQ => 'a list EQ ...

local
  structure S = struct end
  local
    instance int EQ ...
  in
    val x = (fn () => nil == nil) ()
  end
in
  1 :: x
end
```

Solche Programme werden durch die statische Semantik von TML nicht akzeptiert, da für jede Deklaration auf Strukturebene verlangt wird, daß die bei der Elaboration hergeleiteten Wertumgebungen prinzipal bezüglich dieser Deklaration ist.

Dadurch werden außerdem Programme verweigert, die bisher nach der Definition von SML gültig gewesen sind.

Zum Beispiel:

```
val f = (fn x => x) (fn x => (x == x))
structure S = struct end
val x = f 7
```

In der Praxis scheint dieses Problem auf Strukturebene nur selten aufzutreten und kann durch einfache Typannotationen beseitigt werden.

Da `open` in `let`-Ausdrücken erlaubt ist, könnten über diese Hintertür Instanzen lokal eingeführt werden. Dies führt zu dem Kompromiß, daß zwischen `open` auf Kernsprachenebene und Strukturebene unterschieden wird: `open` auf Kernspracheneben führt keine Klassen- und Instanzdeklarationen in den Skopus ein.

2.2 Integration von Typklassen in das Modulsystem

Typklassen sind in TML vollständig in das Modulsystem integriert. Klassen und Instanzen können in Strukturen und Funktoren deklariert werden. Die Signaturspezifikationen werden um Spezifikationen von Klassen und Instanzen erweitert. Außerdem kommt Syntax zum Replizieren von Klassen und Instanzen hinzu.

2.2.1 Klassen- und Instanzdeklaration

In TML werden die Deklarationen auf Strukturebene um Klassen- und Instanzdeklarationen erweitert. Das heißt, sie können überall deklariert werden, wo auch Strukturen deklariert werden können.

Im Gegensatz zu `open` in Deklarationen führt `open` auf Strukturebene auch Instanzen und Klassen ein.

2.2.2 Replikation von Klassen und Instanzen

Genau wie Datentypen und Werte kann man Klassen und Instanzen in den aktuellen Sichtbarkeitsbereich ziehen. Eine Klasse kann wie folgt repliziert werden.

Abbildung 2.5 Klassenreplikation

```
class classid = class longclassid
```

Da Instanzen nicht an einen Bezeichner gebunden sind, ist das Replizieren von Instanzen komplizierter. Um eine Instanz eindeutig zu identifizieren, müssen Instanztyp und Klasse feststehen. Zum Replizieren muß außerdem angegeben werden, von wo die Instanz repliziert werden soll. Daraus ergibt sich folgende Replikationssyntax für Instanzen:

Abbildung 2.6 Instanzreplikation

```
instance instty classid from longstrid
```

Im folgenden Beispiel wird die `int` Instanz der Klasse `TO_STRING` aus der Struktur `String` repliziert.

```
instance int TO_STRING from String
```

Mit der bisher vorgestellten Replikationssyntax können nur Instanzen aus einer Struktur repliziert werden. Um eine Instanz aus dem aktuellen Sichtbarkeitsbereich in eine Struktur zu replizieren, wird die `from`-Angabe weggelassen.

In nachstehendem Beispiel wird die `int`-Instanz der Klasse `EQ` in die Struktur `S` repliziert.

```
instance int EQ with ... end

structure S = struct
                 instance int EQ
              end
```

Es ist möglich, die Replikation mit einer `from`-Angabe auf die vereinfachte Variante zurückzuführen. Hierzu wird die Struktur in einer `local`-Deklaration geöffnet und die entsprechende Instanz mit der vereinfachten Replikationssyntax repliziert.

Somit ist

```
instance instty classid from longstrid
```

äquivalent zu:

```
local
  open longstrid
in
  instance instty classid
end
```

2.2.3 Spezifikation von Klassen

Eine Klassenspezifikation hat den gleichen Aufbau wie eine Klassendeklaration. Eine Struktur paßt auf eine Signatur mit einer Klassenspezifikation, wenn sie eine Klassendeklaration mit gleichem Klassenbezeichner enthält und alle Methoden der Klasse dem in der Struktur spezifizierten Typ exakt entsprechen.

Abbildung 2.7 Klassenspezifikation

```
class tyvar classid with
  methspec
end
```

Im Gegensatz zu Werten dürfen die Methoden in der Struktur nicht allgemeiner sein, da dadurch die Korrektheit des Typsystems verloren geht. Folgendes Beispiel verdeutlicht das Problem.

Zunächst wird eine Klasse `C` mit einer Methode `m` deklariert. Die Methode `m` hat dabei den Typ `'a * 'b -> unit`. Insbesondere akzeptiert sie auf dem zweiten Argument Werte von beliebigem Typ.

```
class 'a C with
  val m : 'a * 'b -> unit
end
```

Die folgende Instantiierung der Klasse `C` für `int` ist sicherlich gültig.

```
instance int C with
  fun m (x, y) = ()
end
```

Die Funktion `f` hat den Typ `'a C => 'a -> unit`.

```
fun f x = m (x, 1)
```

Die Spezifikation der Klasse `C` in der Signatur `G` ist ähnlich zur Deklaration oben. Allerdings sind bei der Methode `m` als zweites Argument nur noch Werte zugelassen, deren Typ eine Instanz von `'b list` ist.

```
signature G =
  sig
     class 'a C with
        val m : 'a * 'b list -> unit
     end
  end
```

Die unten stehende Struktur `S` erfüllt die Signatur `G`, sofern man erlaubt, daß Methodenspezifikationen in Signaturen spezieller sein dürfen als die zugehörige Spezifikation in der Struktur.

```
structure S : G =
  struct
     class c = class C
  end

open S
```

Nach dem Öffnen der Struktur `S` definiert die folgende Deklaration eine gültige Instanz der Klasse `C`. Insbesonder wird auf dem zweiten Argument der Methode `m` jetzt eine Liste erwartet.

```
instance int C with
  fun m (x, nil) = ()
end
```

Sicherlich ist die Applikation `f 1` typkorrekt. Innerhalb der Funktion `f` wird die Methode `m` auf `(1, 1)` appliziert. Für die Methodenapplikation wird aber die zuletzt definierte Instanz verwendet, deren Methode `m` auf dem zweiten Argument eine Liste erwartet. Damit ist das Programm nicht typkorrekt, obwohl es vom Typsystem akzeptiert wurde.

2.2.4 Spezifikation von Instanzen

Eine Instanz wird wie in Abbildung 2.8 spezifiziert.

Abbildung 2.8 Instanzspezifikation

```
instance context => instty classid
```

Eine gültige Implementierung für eine Instanzspezifikation muß denselben Instanzkontext aufweisen wie die Spezifikation.

2.2.5 Abstrakte Klassenspezifikation

Eine Klassenspezifikation kann auch abstrakt sein, das heißt es wird keine Typsignatur der Methoden angegeben. Abbildung 2.9 zeigt die Syntax einer abstrakten Klassenspezifikation.

Abbildung 2.9 Abstrakte Klassenspezifikation

```
class classtyvar classid
```

Eine abstrakte Klasse kann nicht instantiiert werden.

Allerdings kann die Klasse in Typschemata auftreten. Zum Beispiel kann man die Funktion `f` in folgender Struktur nur mit den vordefinierten Instanzen der Struktur verwenden.

```
structure S : sig
                class 'a c
                instance int c
                instance string c
                val f : 'a c => 'a -> unit
              end
            = struct ... end
```

2.2.6 Erweiterte Wertspezifikationen

In TML ist es möglich, qualifizierte Typschemata, also Typschemata mit Kontext, für Werte in Signaturen zu spezifizieren. Die verallgemeinerte Syntax einer Wertspezifikation hat die Form:

Abbildung 2.10 Wertspezifikationen

```
val vid : context => ty
```

Dabei ist ein Kontext eine Sequenz von Paaren von Typvariable und Klassenbezeichner. Folgende Signatur spezifiziert zum Beispiel die typischen Mengenoperationen über einem abstrakten Typ `'a set`.

```
signature SET = sig
  type 'a set
  val empty  : 'a EQ => 'a set
  val insert : 'a EQ => 'a * 'a set -> 'a set
  val member : 'a EQ => 'a * 'a set -> bool
  val subset : 'a EQ => 'a set * 'a set -> bool
end
```

2.2.7 Signaturabgleich

In TML ist eine Struktur eine zulässige Implementierung einer Signatur, wenn diese polymorpher ist und mindestens alle Komponenten der Signatur enthält.

Zum Beispiel ist

```
'a EQ => 'a -> unit
```

allgemeiner als

```
('a EQ, 'a ORD) => 'a -> unit
```

In Standard ML erfolgt der Signaturabgleich, ohne die Laufzeitumgebung einzubeziehen. In ML mit Typklassen hängt der Signaturabgleich von der Instanzumgebung ab. Zum Beispiel kann folgende Spezifikation

```
val f : int -> unit
```

mit

```
val f : 'a EQ => 'a -> unit
```

abgeglichen werden, wenn im Sichtbarkeitsbereich eine `int` Instanz der Klasse `EQ` liegt.

Damit kann man in TML die Laufzeitsemantik einer Struktur spezialisieren.

Zum Beispiel hat die Struktur

```
structure S = struct
  fun f x = (x == x)
end
```

die hergeleitete Signatur:

```
sig
  val f : 'a EQ => 'a -> bool
end
```

Die Struktur stellt somit eine zulässige Implementierung der Signatur `G` dar, sofern eine `int`-Instanz der Klasse `EQ` existiert:

```
signature G = sig
  val f : int -> bool
end
```

Beim Signaturabgleich von `G` mit `S`

```
structure T : G = S
```

wird die Struktur `S` insofern spezialisiert, als sie die beim Abgleich sichtbare `int`-Instanz der Klasse `EQ` statisch bindet.

2.3 Beziehung zwischen Funktoren und Typklassen

Dieser Abschnitt untersucht die Beziehung zwischen dem Modulsystem und Typklassen. Im Ergebnis stellt sich heraus, daß ein näherer Zusammenhang zwischen Typklassen und Modulsystem existiert, und daß sich die Konzepte teilweise überlagern.

2.3.1 Mengen mit Funktoren

Folgendes Beispiel wurde aus [Oka98] übernommen. Die Signatur definiert die leere Menge, eine Operation zum Einfügen von Elementen, sowie den Elementtest über Mengen. Der Mengentyp und der Typ der Elemente sind abstrakt.

```
signature SET = sig
  type set
  type elem
  val empty  : set
  val insert : elem * set -> set
  val member : elem * set -> bool
end
```

Folgender Funktor implementiert die Signatur SET für Mengen. Die interne Repräsentation der Mengen erfordert eine Ordnung auf den Elementen. Der Funktor wurde deswegen über eine Struktur, die eine solche Ordnung bietet, parametrisiert.

```
functor UnbalancedSet(Element : ORDERED) : SET = struct
  type elem = Element.t
  datatype tree = E | T of tree * elem * tree
  type set = tree

  val empty = E

  fun member ... Element.compare ...
  fun insert ... Element.compare ...
end
```

Dabei ist ORDERED wie folgt definiert:

```
signature ORDERED = sig
  type t
  val compare : t * t -> order
end
```

Die tatsächliche Implementierung der Funktionen ist hier nicht weiter von Bedeutung und kann in [Oka98] nachgelesen werden.

Um die Implementierung für Mengen verwenden zu können, muß zunächst für den gewünschten Elementtyp eine Implementierung der Signatur ORDERED zur Verfügung gestellt werden. Danach wird der Funktor appliziert und man erhält eine Struktur für den gewünschten Typ.

```
structure IntOrder : ORDERED = struct ... end

structure IntSet = UnbalancedSet(IntOrder)
```

Danach läßt sich die erzeugte Struktur `IntSet` wie folgt verwenden:

```
IntSet.insert (1, IntSet.insert (1, IntSet.empty))
```

2.3.2 Mengen mit Typklassen

Mit Hilfe von Typklassen kann der Funktor entfallen. Dazu muß man die Signatur `SET` so abändern, daß die Funktionen ausschließlich für Elementtypen verwendet werden können, die Instanz der folgenden Klasse `ORD` sind.

```
class 'a ORD with
  val compare : 'a * 'a -> order
end
```

Damit erhalten wir die folgende Signatur für `SET`:

```
signature SET = sig
  type 'a set
  val empty  : 'a ORD => 'a set
  val insert : 'a ORD => 'a * 'a set -> 'a set
  val member : 'a ORD => 'a * 'a set -> bool
end
```

Eine Implementierung der Signatur `SET` kann die Methoden der Klasse `ORD` verwenden. Damit wird ein Funktor überflüssig.

```
structure Set :> SET =
  datatype 'a tree = E | T of tree * 'a * tree
  type 'a set = 'a tree
  val empty = E
  fun insert ... compare ...
  fun member ... compare ...
```

Um die Struktur für Mengen zu benutzen, muß zunächst eine `int`-Instanz der Klasse `ORD` gebildet werden. Eine Funktorapplikation ist nicht nötig.

```
instance int ORD with
  fun compare ...
end
```

Der Aufruf der Mengenoperationen gestaltet sich dann ähnlich wie oben:

```
Set.insert (1, Set.insert (1, Set.empty))
```

Für spezielle Anforderungen kann die Struktur `Set` auch spezialisiert werden. Dies geschieht einfach, indem man mit einer spezielleren Signatur abgleicht.

```
signature INTSET =
  sig
    type 'a set
    val empty : int set
    val insert : int * int set -> int set
    val member : int * int set -> bool
  end

structure IntSet :> INTSET = Set
```

Beim Abgleich mit der Signatur `INTSET` wird dabei die im Geltungsbereich liegende `int`-Instanz der Klasse `ORD` statisch gebunden. Die Struktur `Set` wird also gewissermaßen partiell ausgewertet.

2.3.3 Gegenüberstellung

Macht man eine Gegenüberstellung zwischen dem Typklassenkonzept und dem Modulsystem, so ergeben sich folgende Korrespondenzen.

Eine Klasse entspricht einer Signatur:

Typklassen	Modulsystem
`class 'a ORD`	`signature ORDERED`

Eine Instanz ohne Kontext entspricht einer Struktur, eine mit Kontext einem Funktor.

Typklassen	Modulsystem
`instance int ORD`	`structure IntOrd : ORDERED`
`instance 'a ORD => 'a list ORD`	`functor ListOrd(Elem : ORDERED)` `  : ORDERED`

Eine Struktur, deren Wertsignaturen Constraints enthalten, entspricht einem Funktor, der über die Constraits parametrisiert ist.

Typklassen

```
structure Set :
sig
  type 'a set
  val empty  : 'a ORD => 'a set
  val insert : 'a ORD => 'a * 'a set -> 'a set
  val member : 'a ORD => 'a * 'a set -> bool
end
```

Modulsystem

```
functor Set (Ord : ORDERED) :
sig
  type set
  type elem
  val empty  : set
  val insert : elem * set -> set
  val member : elem * set -> bool
end
```

Der Abgleich mit einer spezielleren Signatur entspricht der Applikation des Funktors.

Typklassen

```
structure IntSet :
sig
  type 'a set
  val empty : int set
  val insert : int * int set -> int set
  val member : int * int set -> bool
end = Set
```

Modulsystem

```
structure IntSet = Set(IntOrd)
```

Untersucht man obigen Vergleich genauer, stellt sich heraus, daß Typklassen einfacher zu verwenden sind, da beim Aufruf einer Funktion immer dieselbe Struktur `Set` verwendet wird. Bei der Funktorlösung hängt die aufzurufende Struktur vom Typ der Mengenelemente ab und zudem muß diese zuvor mittels Funktorapplikation erzeugt werden.

Ein weiterer Vorteil von Typklassen ist das automatische Erzeugen der Instanzen. Bildet man zum Beispiel Listen von Listen von ganzen Zahlen, muß in der Funktorlösung durch

mehrmalig Funktorapplikationen eine Struktur mit einer passenden Ordnung erzeugt werden. Bei Typklassen entfällt dies vollständig weil die Listen Instanz der Klasse `ORD` universell für jede Art von Listen funktioniert. Die Universalität der Listeninstanz würde einer impliziten und automatischen Funktorapplikation auf Modulseite entsprechen.

Bei der Funktorlösung werden für jeden Elementtyp eigene abstrakte Typen `elem` und `set` erzeugt. Will man aufsetzend auf den Mengenoperationen weitere Operationen wie zum Beispiel `subset` definieren, die nicht vom Elementtyp abhängen, so muß man diese in einem Funktor über Mengen parametrisieren.

```
functor AdvancedSet (Set : SET) :>
sig
  include SET
  val subset : set * set -> bool
end
```

Der Mengentyp der Typklassenlösung ist hingegen polymorph über dem Typ der Mengenelemente. Das heißt es gibt nur einen Typkonstruktor für alle Arten von Mengen. So können Funktionen wie der Teilmengentest leicht für beliebige Mengen definiert werden.

2.4 Mißbrauch von Typklassen

Dieser Abschnitt diskutiert Probleme, die durch den Mißbrauch von Typklassen entstehen können. Offensichtlich werden diese Probleme durch die Möglichkeit, Instanzen zu verdecken, verursacht. Allerdings stellt sich bei näherer Betrachtung der Integration von Typklassen ins Modulsystem heraus, daß dieselben Probleme auch durch das Modulsystem verursacht werden können.

2.4.1 Mißbrauch durch Verdecken in der Kernsprache

Obwohl der hier vorgestellte Dialekt das Verdecken von Instanzen zuläßt, ist damit große Vorsicht geboten.

Betrachten wir noch einmal die Signatur für Mengen mit Hilfe von Typklassen.

```
signature SET = sig
  type 'a set
  val empty : 'a ORD => 'a set
  val insert : 'a ORD => 'a * 'a set -> 'a set
  val member : 'a ORD => 'a * 'a set -> bool
end
```

Eine gültige Implementierung der Signatur könnte wie folgt aussehen. Dabei werden Mengen intern als geordnete Binärbäume dargestellt.

```
structure Set : SET =
    struct
        datatype 'a tree =
            E
          | T of 'a tree * 'a * 'a tree
        type 'a set = 'a tree
        val empty = E
        fun member (x, E) = false
          | member (x, s as T (a, y, b)) =
            (case compare (x, y) of
                 LESS => member (x, a)
               | GREATER => member (x, b)
               | EQUAL => true)
        fun insert (x, E) = ....
          | insert (x, s as T (a, y, b)) = ...
```

Nun wird eine Menge mit Zeichenketten erzeugt. Die im Geltungsbereich liegende Ordnung auf Zeichenketten ist die lexikographische.

```
(* Lexikographische Ordnung *)
instance string ORD with ... end

val mySet = Set.insert ("Grizzly",
                        Set.insert ("Orca",
                                    Set.insert ("Elephant",
                                                Set.empty)))
```

Danach wird die Ordnung umdefiniert und getestet, ob `"Orca"` ein Element der Menge ist. Da die neue Ordnung umgekehrt lexikographisch ist, steigt die Implementierung des Elementtests auf dem falschen Pfad ab. Schließlich liefert der Elementtest fälschlicherweise `false`.

```
(* Umgekehrte lexikographische Ordnung *)
instance string ORD with ... end;

Set.member ("Orca", mySet)
```

2.4.2 Mißbrauch mit Hilfe des Modulsystems

Die gleiche Situation wie im vorherigen Abschnitt, läßt sich mit Hilfe des Modulsystems ohne Verdecken von Instanzen nachstellen.

```
structure S : sig
                val mySet : int Set.set
              end
  = struct
      instance int ORD with ... (* lexikographische Ord-
nung *) end
       val mySet = Set.insert ("Grizzly", ...
    end
```

Durch Öffnen der Struktur S wird lediglich `mySet` in den Skopus eingeführt. Die Ordnungs-Instanz wurde aufgrund des Signaturconstraints nicht exportiert.

Nun kann man eine neue Ordnungs-Instanz für `int` deklarieren. Auch in diesem Falle wird der Elementtest fehlschlagen.

```
instance int ORD with ... (* umgekehrt-lexikographische Ord-
nung *)

Set.member ("Orca", mySet)
```

2.4.3 Mißbrauch mit Hilfe von `local`

Analog zur Modulsystem-Variante läßt sich das Problem mit `local` reproduzieren.

```
local
  instance int ORD with ... (* lexikographische Ord-
nung *) ... end
in
  val mySet = ...
end

instance int ORD with ... (* umgekehrt lexikographische Ord-
nung *) ... end
```

2.4.4 Lösung des Problems

In diesem Abschnitt wird kurz auf mögliche Sprachrestriktionen eingegangen, die obigem Problem entgegen wirken.

Einschränkung 1: Instanzen nur noch auf dem Toplevel

Die bisherige Diskussion hat gezeigt, daß es nicht ausreicht, das Verdecken von Instanzen zu verbieten. Um das beschriebene Problem vollständig zu beseitigen, müßte man das Deklarieren von Klassen und Instanzen auf Strukturebene verbieten. Danach wären Klassen- und Instanzdeklarationen nur noch auf dem Toplevel erlaubt. Durch diese Einschränkung verliert man alle interessanten neuen Eigenschaften, die TML durch die Integration von Typklassen in das Modulsystem mit sich bringt.

Einschränkung 2: Verdecken verbieten

Letztendlich könnte man nur das Verdecken von Instanzen verbieten. Der Vorteil dieser Lösung besteht darin, daß man nicht durch schlechten Programmierstil in die Situation der ersten Variante des Problems gerät.

Leider hilft dies nicht gegen die zweite und dritte Variante. Gerade aber die zweite Variante scheint extrem gefährlich zu sein. Man denke an ein großes Softwaresystem, indem des öfteren Gebrauch von `open` gemacht wird. Ein Programmfehler obiger Art könnte unter Umständen nur sehr schwer zu finden sein. Für Anfänger könnte dieses Problem unlösbar werden.

Keine Einschränkung

In TML gibt es keine Einschränkung, so daß die Verantwortung beim Programmierer liegt.

2.5 Gleichheit in TML

In diesem Abschnitt wird der Zusammenhang zwischen Gleichheit in Standard ML und ML mit Typklassen erläutert.

2.5.1 Typspezifikationen

Ein Spezifikation `eqtype 'a t` ist in ML so definiert, daß ein Wert vom Typ `s t` Gleichheit zuläßt, wenn `s` Gleichheit zuläßt. In TML kann dies mit einer abgeleiteten Instanz ausgedrückt werden: `instance 'a EQ => 'a t EQ`.

Außerdem kann man in TML spezifizieren, daß auf einem Typ `s t` Gleichheit definiert ist, ohne daß auf `s` Gleichheit definiert ist. Dies ist in Standard ML nicht möglich. Man erhält die Spezifikation, in dem man eine Instanz für `t` der Klasse `EQ` spezifiziert, deren Kontext leer ist:

```
instance 'a t EQ
```

Allgemeiner kann man beliebige Abhängigkeiten definieren. In folgendem Beispiel wird die Gleichheit nur auf der ersten Komponente des Paares verlangt.

```
instance 'a EQ => ('a, 'b) pair EQ
```

2.5.2 Wertspezifikationen

Eine Wertspezifikation mit Gleichheitstypvariablen kann äquivalent in TML spezifiziert werden, indem für jede Gleichheitstypvariable der Kontext der Typvariable um `EQ` erweitert wird. Zum Beispiel kann `''a -> unit` in `'a EQ => 'a -> unit` umgeschrieben werden.

2.6 Exists-Funktors

In diesem Abschnitt wird ein Funktor vorgestellt, der eine ähnliche Ausdruckskraft ermöglicht, wie die in [Läu94] vorgestellten existentiell quantifizierten Typen. Der Funktor selbst kann nicht in TML implementiert werden.

2.6.1 Signatur des Exists-Funktor

Die Signatur des Exists-Funktors wird durch Angabe der Argument und Resultatsignatur gegeben.

```
functor Exists (class 'a C) :
sig
  type pack
  instance pack C
  val inject : 'a C => 'a -> pack
end
```

Die Idee besteht darin, daß man mit Hilfe der Funktion `inject` einen Wert, dessen Typ eine Instanz der Klasse `C` ist, in ein Paket umwandeln kann. Das Paket enthält dabei die Instanz zu dem Typ des Wertes der Klasse `C` und den Wert selbst.

Wird nun eine Methode der Klasse `C` auf ein Paket, also auf einen Wert mit dem Typ `pack` angewendet, so wird die `pack`-Instanz verwendet. Diese leitet den Aufruf der Methode an die im Paket enthaltene Instanz weiter, indem sie die entsprechende Methode aus der Instanz auswählt und dieser den Wert im Paket übergibt.

2.6.2 Anwendung des Exists-Funktors

Angenommen, eine Klasse `TO_STRING` mit einer Methode `toString` wurde für ganze Zahlen, Zeichenketten und Listen instantiiert. Dann liefert die Funktorapplikation

```
structure EToString = Exists (class C = class TO_STRING)
```

eine Struktur mit folgender Signatur:

```
EToString :
sig
  type pack
  instance pack TO_STRING
  val inject : 'a TO_STRING => 'a -> pack
end

open EToString
```

Nach dem Öffnen der Struktur kann man eine homogene Liste vom Typ `pack list` aufbauen, deren Elemente Pakete aus einer Instanz für `TO_STRING` und einem Wert sind.

```
val pack_list =
    [inject 1, inject "Hello World", inject [1, 2, 3]]
    : pack list
```

Da der Typ `pack` eine Instanz der Klasse `TO_STRING` ist, kann die Methode `toString` auf jedes Element der Liste angewendet werden. Die `pack`-Instanz leitet dabei jeden Aufruf an die Instanz im Paket weiter. Je nach Implementierung der Instanzen könnte das Ergebnis der Abbildung obiger Liste mit der Methode `toString`

```
List.map toString pack_list
```

wie folgt aussehen:

```
["1", "Hello World!", "[1, 2, 3]"] : string list
```

2.6.3 Erlaubte Klassen

Enthält eine Methode der Klasse `C` als Ergebnis die Klassentypvariable, so müßte diese wieder in ein Paket umgewandelt werden. Dies ist mit Typinformation immer dann möglich, wenn die Klassentypvariable im Argumenttyp vorkommt.

Kommt die Klassentypvariable ausschließlich im Resultat vor, so kann kein Paket erzeugt werden, da unklar ist, welchen Wert und welche Instanz in das Paket aufgenommen werden sollen.

```
class 'a C with
  val m : unit -> 'a
end

structure EC = Exists (class c = class C)

val pack = m () : EC.pack    (* Semantik völlig unklar *)
```

2.6.4 Probleme durch Verdecken von Instanzen

Ein weiteres Problem wird durch die Möglichkeit, Instanzen zu verdecken, verursacht. In folgendem Beispiel wird für eine Klasse `C` die Existenzstruktur erzeugt.

```
class 'a C with
  val m : 'a * 'a -> bool
end

structure EC = Exists (class c = class C)

open EC
```

Nun wird eine Instanz definiert, deren Methode `m` immer `true` zurück gibt und anschließend ein Paket erzeugt.

```
instance int C with
  fun m (_, _) = true
end

val pack1 = inject 1
```

Weil man Instanzen verdecken kann, ist es möglich, ein weiteres Paket zu erzeugen, bei dem die Methode `m` aber `false` zurückgibt.

```
instance int C with
  fun m (_, _) = false
end

val pack2 = inject 1
```

Das Problem tritt auf, wenn die Methode `m` auf `(pack1, pack2)` appliziert wird. Die Pakete enthalten unterschiedliche Instanzen, so daß die Semantik von der `Exists`-Funktorimplementierung abhängt.

Kapitel 3

Syntax von ML mit Typklassen

Dieses Kapitel erweitert die Definition von Standard ML von 1997 [MTHM97] um die für Typklassen notwendige Syntax. Es werden die aus der Definition üblichen Abkürzungen verwendet. Kenntnis der Kapitel 2 und 3 der Definition wird vorausgesetzt. Die Struktur dieses Kapitels orientiert sich im Wesentlichen an der der Definition, so daß beide auf einfache Weise zu vergleichen sind.

Normalerweise werden die hier angegebenen Grammatikregeln zu denen der Standard ML Definition hinzugenommen. Bei Überlappung gilt die Regel in diesem Dokument. Regeln der Form

$$\begin{array}{lll} x & += & Alt_1 \\ & | & \ldots \\ & | & Alt_N \end{array}$$

erweitern die in der Standard Definition um die Alternativen Alt_1 bis Alt_N.

3.1 Kernsprache

3.1.1 Bezeichner

Aus der Menge der symbolischen Bezeichner wird = der Einfachheit halber herausgenommen. = als Bezeichner zuzulassen, würde zu Komplikationen bei der Syntaxanalyse führen (zum Beispiel bei `val = = = =`). Da = kein Bezeichner mehr ist, entfällt die Sonderregel, daß er nicht neu gebunden werden darf.

Zu den Bezeichnerkategorien[1] werden die Typklassenbezeichner hinzugenommen. Die folgende Abbildung listet alle Bezeichnerkategorien der Kernsprache auf. Die mit *long* markierten Bezeichnerklassen können qualifiziert werden. `vid`, `tyvar`, `longvid`, `longtycon`,

[1] Um Verwechslungen zwischen Typ- und Bezeichnerklassen zu vermeiden, wurde “identifier class” mit “Bezeichnerkategorie” übersetzt.

usw. denotieren einen Bezeichner aus der entsprechenden Bezeichnerkategorie `VId`, `TyVar`, `LongVId`, `LongTyCon`, usw.

`VId`	(Wertbezeichner)	*long*
`TyVar`	(Typvariablen)	
`TyCon`	(Typkonstruktoren)	*long*
`Lab`	(Recordlabels)	
`StrId`	(Strukturbezeichner)	*long*
`ClassId`	(Typklassenbezeichner)	*long*

Gleichheitstypvariablen verlieren ihren Sonderstatus und werden als ganz normale Typvariablen aufgefaßt.

3.1.2 Grammatik der Kernsprache

Die Grammatik der Kernsprache wird nur bei Wertbindungen zum Annotieren von Klassenkontexten an die Typvariablen modifiziert.

In `dec` wird die Wertbindung herausgenommen und durch folgende ersetzt:

$$\begin{array}{rcll} \textit{dec} & += & \texttt{val}\ \langle \textit{constraints}\ \texttt{=>} \rangle\ \textit{tyvarseq}\ \textit{valbind} & \\ \textit{constraint} & ::= & \textit{tyvar}\ \textit{longclassid} & \\ \textit{constraints} & ::= & \textit{constraint} & \\ & | & (\ \textit{constraint}_1\ ,\ \ldots\ ,\ \textit{constraint}_n\ ) & n \geq 1 \end{array}$$

3.2 Klassensprache

Dieser Abschnitt definiert die Klassensprache von TML. Es gibt keine Korrespondenz in der Definition von Standard ML. Neben Regeln für die Deklarationen und Replikation von Klassen und Instanzen kommt eine Regel für Typausdrücke mit Klassenbedingungen hinzu.

3.2.1 Schlüsselwörter der Klassensprache

Die Klassensprache benutzt zusätzlich die folgenden Schlüsselwörter. Dies sind zugleich alle Schlüsselwörter, die zu TML hinzugenommen werden.

```
class instance from
```

3.2.2 Grammatik der Klassensprache

classdec ::= `class` *classbind*
| `instance` *instbind*
| `class` *classid* = `class` *longclassid*
| `instance` *longtyfcn longclassid*

classbind ::= *tyvar classid* `with` *methspec* `end`

methspec ::= ϵ leer
| `val` *methdesc*
| *methspec*$_1$ ⟨;⟩ *methspec*$_2$

instbind ::= ⟨*constraints* `=>`⟩ *instty longclassid*
`with` *methdec* `end` ⟨`and` *instbind*⟩

methdec ::= ϵ leer
| `val` ⟨*constraints* `=>`⟩ *tyvarseq valbind*
| *methdec*$_1$ ⟨`;`⟩ *methdec*$_2$

qualty ::= ⟨*constraints* `=>`⟩ *ty*

methdesc ::= *vid* `:` *qualty*

instty ::= *tyvarseq longtycon*
| { ⟨*insttyrow*⟩ }
| *tyvar*$_1$ `->` *tyvar*$_2$

insttyrow ::= *lab* `:` *tyvar* ⟨`,` *insttyrow*⟩

3.3 Modulsprache

Die Deklarationen auf Strukturebene werden um die Deklarationen aus der Klassensprache erweitert. Außerdem wird für TML zwischen `open` in Strukturdeklarationen (*strdec*) und in Deklarationen (*dec*) der Kernsprache unterschieden. Bei den Spezifikationen kommen zusätzliche Regeln zum Spezifizieren von Klassen und Instanzen hinzu. Die Wertbeschreibungen werden um Annotationen für Klassenbedingungen erweitert.

3.3.1 Grammatik der Modulsprache

strdec	$+=$	*classdec*
	|	`open` *longstrid*$_1$ ... *longstrid*$_n$
spec	$+=$	`class` *classdesc*
	|	`class` *classid* `=` `class` *longclassid*
	|	`instance` *instspec*
classdesc	::=	*tyvar classid* ⟨`with` *methspec* `end`⟩
instspec	::=	⟨*constraints* `=>`⟩ *instty longclassid*
valdesc	::=	`val` *vid* `:` *qualty*

3.4 Abgeleitete Syntax

Die abgeleitete Syntax von Standard ML wird um eine abgeleitete Form zum Replizieren von Instanzen erweitert. In folgender Tabelle steht links die abgeleitete Form und rechts die zurückgeführte Form.

Deklarationen auf Strukturebene *strdec*

`instance` *instty longclassid*
`from` *longstrid*
	`local`
	`open` *longstrid*
	`in`
	`instance` *instty longclassid*
	`end`

3.5 Syntaktische Einschränkungen

Zu den Einschränkungen der Definition von Standard ML kommt die folgende hinzu:

Eine Typvariable darf nur einmal in einem Instanztyp auftreten. Zum Beispiel ist `('a, 'a) pair` kein gültiger Instanztyp.

Kapitel 4

Statische Semantik

Dieses Kapitel definiert die statische Semantik von TML ausgehend von der statischen Semantik von Standard ML. Der Aufbau dieses Kapitels orientiert sich an der Definition von Standard ML [MTHM97].

4.1 Statische Semantik der Kernsprache

4.1.1 Primitive Werte

Zu den primitiven Werten der statischen Semantik, aus denen alle semantischen Werte aufgebaut sind, kommt die Menge der Klassennamen. Da in TML die Spezialbehandlung der Gleichheit entfällt, tragen Typvariablen und Typnamen kein Gleichheitsattribut mehr.

Abbildung 4.1 Primitive Werte

α or *tyvar*	$\in$	TyVar	Typvariablen
t	$\in$	TyName	Typnamen
is	$\in$	IdStatus $=$ {c, e, v}	Bezeichnerstatus
γ	$\in$	ClassName	Klassennamen
cs	$\in$	ClassStatus $=$ {a, k}	Klassenstatus

Typklassen tragen in TML einen Klassenstatus, der angibt, ob die Klasse abstrakt (a) oder konkret (k) ist.

4.1.2 Zusammengesetzte semantische Objekte

Zu den zusammengesetzten Objekte der Definition von Standard ML werden eine Klassenumgebung *CE*, eine Instanzumgebung *IE* und eine Umgebung zum Aufsammeln von Typvariablenconstraints *TC* hinzugenommen.

Abbildung 4.2 Zusammengesetzte Objekte der statischen Semantik

$$
\begin{array}{rcl}
\tau & \in & \mathrm{Type} = \mathrm{TyVar} \cup \mathrm{RowType} \cup \mathrm{FunType} \cup \mathrm{ConsType} \\
(\tau_1, \ldots, \tau_k) \text{ or } \tau^{(k)} & \in & \mathrm{Type}^k \\
(\alpha_1, \ldots, \alpha_k) \text{ or } \alpha^{(k)} & \in & \mathrm{TyVar}^k \\
\\
\varrho & \in & \mathrm{RowType} = \mathrm{Lab} \xrightarrow{fin} \mathrm{Type} \\
\tau \to \tau' & \in & \mathrm{FunType} = \mathrm{Type} \times \mathrm{Type} \\
 & & \mathrm{ConsType} = \bigcup_{k \geq 0} \mathrm{ConsType}^{(k)} \\
\tau^{(k)} t & \in & \mathrm{ConsType}^{(k)} = \mathrm{Type}^k \times \mathrm{TyName}^{(k)} \\
\theta \text{ or } \Lambda\alpha^{(k)}.\tau & \in & \mathrm{TypeFcn} = \bigcup_{k \geq 0} \mathrm{TyVar}^k \times \mathrm{Type} \\
\sigma \text{ or } \forall(\alpha\ \Gamma)^{(k)}.\tau & \in & \mathrm{TypeScheme} = \bigcup_{k \geq 0} (\mathrm{TyVar} \times \mathrm{Constraints})^k \times \mathrm{Type} \\
(\Theta, VE) & \in & \mathrm{TyStr} = \mathrm{TypeFcn} \times \mathrm{ValEnv} \\
\\
(\gamma_1, \ldots, \gamma_k) \text{ or } \Gamma & \in & \mathrm{Constraints} = \bigcup_{k \geq 0} \mathrm{ClassName}^{(k)} \\
(\Gamma_1, \ldots, \Gamma_k) \text{ or } (\Gamma)^{(k)} & \in & \mathrm{InstConstraints} = \bigcup_{k \geq 0} \mathrm{Constraints}^k \\
\iota & \in & \mathrm{InstTyFcn} = \{\to\} \cup \mathrm{TyName} \cup \bigcup_{k \geq 0} \mathrm{Lab}^k \\
(\gamma, \xi\alpha.VE, cs) & \in & \mathrm{ClassStr} = \mathrm{ClassName} \times \mathrm{TyVar} \times \mathrm{ValEnv} \times \mathrm{ClassStatus}
\end{array}
$$

$$
\begin{array}{rcl}
SE & \in & \mathrm{StrEnv} = \mathrm{StrId} \xrightarrow{fin} \mathrm{Env} \\
TE & \in & \mathrm{TyEnv} = \mathrm{TyCon} \xrightarrow{fin} \mathrm{TyStr} \\
CE & \in & \mathrm{ClassEnv} = \mathrm{ClassId} \xrightarrow{fin} \mathrm{ClassStr} \\
IE & \in & \mathrm{InstEnv} = \mathrm{InstTyFcn} \times \mathrm{ClassName} \xrightarrow{fin} \mathrm{InstConstraint} \\
VE & \in & \mathrm{ValEnv} = \mathrm{VId} \xrightarrow{fin} \mathrm{TypeScheme} \times \mathrm{IdStatus} \\
TC & \in & \mathrm{TyVarConstraints} = \mathrm{TyVar} \xrightarrow{fin} \mathrm{Constraints}
\end{array}
$$

$$
\begin{array}{rcl}
E \text{ or } (SE, TE, VE, CE, IE) & \in & \mathrm{Env} = \\
 & & \mathrm{StrEnv} \times \mathrm{TyEnv} \times \mathrm{ValEnv} \times \\
 & & \mathrm{ClassEnv} \times \mathrm{InstEnv}
\end{array}
$$

$$
\begin{array}{rcl}
T & \in & \mathrm{TyNameSet} = \mathrm{Fin}(\mathrm{TyName}) \\
U & \in & \mathrm{TyVarSet} = \mathrm{Fin}(\mathrm{TyVar}) \\
C \text{ or } (T, K, TC, U, E) & \in & \mathrm{Context} = \\
 & & \mathrm{TyNameSet} \times \mathrm{ClassNameSet} \times \\
 & & \mathrm{TyVarSet} \times \mathrm{Env}
\end{array}
$$

Klassenumgebung *CE*. Die Klassenumgebung bildet einen Klassenbezeichner auf einen Klassennamen, eine Wertumgebung und den Klassenstatus ab. Der Klassenstatus gibt an, ob die Klasse abstrakt (a) oder konkret (k) ist. Die Wertumgebung enthält die in der Klassendeklaration spezifizierten Methoden. Dabei ist die Klassentypvariable durch die Umgebung $\xi\alpha.VE$ gebunden und tritt somit in den Methodentypen frei auf.

Die Klassendeklaration

```
class 'a C with
  val m : 'a * 'b -> 'b
end
```

resultiert in folgender *CE*-Umgebung:

$$CE = \{\mathrm{C} \mapsto (c, \xi\alpha.\{\mathrm{m} \mapsto (\forall\beta.\{1 \mapsto \alpha, 2 \mapsto \beta\} \to \beta\}, \mathrm{v})\}, \mathrm{k})\}$$

Dabei ist c ein neuer Klassenname und α die Klassentypvariable.

In der statischen Semantik wird nicht zwischen Methoden und anderen Werten unterschieden. Beim Einführen einer Methode in einen Kontext C wird über die Klassentypvariable generalisiert. Dabei wird der Klassenconstraint über der eigenen Klasse und der Klassentypvariable hinzugefügt. Das heißt, eine Methode wird als ganz normale Funktion aufgefaßt. Mittels des Constraints auf der Klassentypvariable wird die Tatsache berücksichtigt, daß bei einer Methodenapplikation eine Instanz der zugehörigen Klasse benötigt wird. Dieses Vorgehen erlaubt es, daß in den semantischen Regeln der Kern- und Modulsprache von TML Methoden nicht speziell behandelt werden müssen.

Zum Beispiel hat obige Methode `m` im Kontext C den Typ:

$$\forall\alpha\ (c), \beta().\{1 \mapsto \alpha, 2 \mapsto \beta\} \to \beta\}$$

Instanzumgebung *IE*. Eine Instanzumgebung ist eine Abbildung von einem Paar aus einer Instanztypfunktion und einem Klassennamen zu einem k-Tupel von Constraints. Dabei ist eine Instanztypfunktion entweder ein Typname, der Funktionstypkonstruktor $\to$ oder eine endliche, geordnete Folge von paarweise verschiedenen Labeln. Das k-Tupel von Klassenconstraints wird als InstConstraints bezeichnet und stellt die semantische Repräsentation des Instanzkontextes einer Instanzdeklaration dar. Klassenconstraints sind endliche, geordnete Folgen von Klassennamen.

Die beiden Instanzdeklarationen

```
instance int C with ... end

instance ('a C, 'b C) => ('a, 'b) pair C with ... end
```

resultieren folgender Instanzumgebung:

$$\begin{aligned} IE \quad = \quad & \{(t_{int}, c) \to (), \\ & (t_{pair}, c) \to ((c), (c))\} \end{aligned}$$

wobei t_{pair} der Typname des Typkonstruktors `pair` und t_{int} der Typname des Typkonstruktors `int` ist.

Typvariablenconstraints *TC*. Zuletzt wird eine Abbildung von Typvariablen auf endliche Teilmengen von Klassennamen benötigt. Diese wird mit Typvariablenconstraints *TC* bezeichnet und dient zum Aufsammeln der Klassenconstraints auf den Typvariablen in Ausdrücken.

Ordnung auf Lab. Die Menge der Labels Lab ist total geordnet. Die Ordnungsrelation wird mit $<$ notiert.

Gleichheit auf Klassenconstraints Γ. Zwei Klassenconstraints sind genau dann gleich, in Zeichen $\Gamma = \Gamma'$, wenn sie die gleichen Elemente enthalten. Dabei spielt die Reihenfolge *keine* Rolle.

4.1.3 Projektions-, Injektions- und Modifikationsabbildungen

Neben den bereits in der Definition von Standard ML definierten Abbildungen wird in diesem Kapitel eine Modifikationsabbildung zum Aufsammeln der Klassenconstraints benötigt. Sie wird mit $\bowtie$ bezeichnet und ist wie folgt definiert:

$$(TC \bowtie TC')(\alpha) := TC(\alpha) \bowtie TC'(\alpha),$$

wobei

$$(\gamma_1, \ldots, \gamma_m) := (\gamma_1, \ldots, \gamma_n) \bowtie (\gamma_{n+1}, \ldots, \gamma_m)$$

$\bowtie$ wird erweitert auf beliebige endliche Zusammenschlüsse:

$$\bowtie_{1 \le i \le n} TC_i := \begin{cases} TC_1 \bowtie \ldots \bowtie TC_n & n \ge 1 \\ \{\} & n = 0 \end{cases}$$

Zwei Typvariablenconstraints *TC* und *TC* sind äquivalent, in Zeichen $TC \equiv TC'$, wenn gilt:

$$TC(\alpha) = TC'(\alpha) \quad \forall \alpha \in \mathrm{Dom}(TC) \cap \mathrm{Dom}(TC')$$

4.1.4 Typen und Typfunktionen

Das Kriterium in der Definition von ML, das die Zulässigkeit von Gleichheit auf Typen spezifiziert, entfällt für TML. Eine Typfunktion $\theta = \Lambda\alpha^{(k)}.\tau$ hat die Stelligkeit k. Zwei Typfunktionen werden als gleich betrachtet, wenn sie sich nur in der Wahl der gebunden Variablen (Alpha-Konversion) unterscheiden. Hat t die Stelligkeit k so steht t für die Typfunktion $\Lambda(\alpha\ \Gamma)^{(k)}.\tau$ (Eta-Konversion). $\tau^{(k)}\theta$ denotiert die Applikation einer Typfunktion

θ auf einen Vektor $\tau^{(k)}$ (Beta-Konversion). Für das Resultat der Substitution von Typfunktionen für Typnamen wird $\tau\{\theta^{(k)}/t^{(k)}\}$ geschrieben. Alle Beta-Konversionen werden direkt nach der Substitution durchgeführt.

4.1.5 Typschemata

Die Typschemata tragen in TML auf den generalisierten Typvariablen Klassenconstraints. Um eine Typvariable mit einem nichtleeren Klassenconstraint zu instantiieren, muß für jedes Element des Klassenconstraints eine passende Instanz im Geltungsbereich liegen.

Zunächst wird eine Funktion κ definiert, die zu jedem Typ die zugehörige Instanztypfunktion und ihre Argumente zurückliefert.

$$\kappa : \text{Type} \to \text{InstTyFcn} \times \bigcup_{k\geq 0} \text{Type}^k$$

mit

$$\begin{array}{rcl}
\kappa(\tau^{(k)}t) & = & (t, \tau^{(k)}) \\
\kappa(\{lab_1 : \tau_1, \ldots, lab_n : \tau_n\}) & = & ((lab_{k_1}, \ldots, lab_{k_n}), (\tau_{k_1}, \ldots, \tau_{k_n})) \\
 & & \text{mit } lab_{k_i} \leq lab_{k_j} \text{ wenn } i \leq j \\
 & & n \geq 0, k_i \neq k_j, k_l \in \{1, \ldots, n\} \\
\kappa(\tau_1 \to \tau_2) & = & (\to, (\tau_1, \tau_2))
\end{array}$$

Die Funktion κ ist nicht total, da sie nicht über Typvariablen definiert ist.

Ein Typ τ ist eine Instanz des Typschema σ unter der Instanzumgebung *IE*, wenn

$$IE \vdash \sigma \succ \tau \Rightarrow TC$$

unter folgenden Regeln ableitbar ist. Dabei werden die für die Instantiierung nötigen Typvariablenconstraints *TC* hergeleitet.

Instantiierung $\boxed{IE \vdash \sigma \succ \tau \Rightarrow TC}$

$$\frac{\sigma = \forall(\alpha\ \Gamma)^{(k)}.\tau' \quad \tau = \tau'\{\tau^{(k)}/\alpha^{(k)}\} \quad IE \vdash (\tau_1\Gamma_1, \ldots, \tau_k\Gamma_k) \Rightarrow TC}{IE \vdash \sigma \succ \tau \Rightarrow TC} \tag{I.1}$$

In (I.1) wird zunächst eine Substitution zum Instantiieren des Typschemas zum Typ τ bestimmt. Die letzte Prämisse in der Regel überprüft die Zulässigkeit der Instantiierung und gibt die dafür notwendigen Typvariablenconstraints zurück.

$\boxed{IE \vdash (\tau\ \Gamma)^{(k)} \Rightarrow TC}$

$$\frac{IE \vdash \tau_i\ \Gamma_i \Rightarrow TC_i\ (1 \leq i \leq k)}{IE \vdash (\tau\ \Gamma)^{(k)} \Rightarrow \bowtie_{1\leq i\leq k} TC_i} \tag{I.2}$$

In Regel (I.2) kann $k = 0$ sein. Damit degeneriert diese zu einem Axiom.

$$\boxed{IE \vdash \tau\ \Gamma \Rightarrow TC}$$

Die folgenden beiden Regeln überprüfen, ob der Typ τ unter der Instanzumgebung *IE* die Klassenconstraints in Γ erfüllt.

$$\frac{TC = \{\alpha \mapsto \Gamma\}}{IE \vdash \alpha\ \Gamma \Rightarrow TC} \tag{I.3}$$

In Regel (I.3) ist der zu überprüfende Typ eine Typvariable. Deshalb werden die Klassenconstraints in Γ als Typvariablenconstraints festgehalten.

$$\frac{\begin{array}{rl} \tau \neq \alpha & \kappa(\tau) = (\iota, \tau^{(k)}) \\ \Gamma^k_\gamma = IE(\iota, \gamma) & \forall \gamma \in \Gamma \\ IE \vdash \tau_i\ \Gamma_{i,\gamma} \Rightarrow TC_{i,\gamma} & \forall \gamma \in \Gamma, 1 \leq i \leq k \end{array}}{IE \vdash \tau\ \Gamma \Rightarrow \bowtie_{1 \leq i \leq k, \gamma \in \Gamma} TC_{i,\gamma}} \tag{I.4}$$

In Regel (I.4) ist τ keine Typvariable. Zunächst wird die zugehörige Instanztypfunktion ι und ihre Argumente $(\tau)^{(k)}$ bestimmt. Anschließend wird rekursiv für jeden Typ τ_i überprüft, ob dieser die Klassenconstraints $(IE(\iota, \gamma))_i$ für alle $\gamma \in \Gamma$ unter der Instanzumgebung *IE* erfüllt. Die Typvariablenconstraints werden als Ergebnis der Regel zusammengefaßt.

Die dritte Prämisse kann nur erfüllt werden, wenn *IE* für (ι, γ) definiert ist.

Beispiel zum Instantiieren von Typschemata

Dieses Beispiel zeigt das Instantiieren von

$$\forall(\alpha\ eq).\alpha \to \alpha$$

mit

$$(\beta, t_{int})t_{pair} \to (\beta, t_{int})t_{pair}$$

unter der Umgebung *IE* von Seite 55.

$$\dfrac{\dfrac{}{IE \vdash \beta(c) \Rightarrow \{\beta \to (c)\}} \quad \dfrac{\kappa(t_{int}) = (t_{int}, ())\quad IE(t_{int}, c) = ()}{IE \vdash (c) \Rightarrow \{\}}}{\dfrac{\kappa((\beta, t_{int})t_{pair}) = (t_{pair}, (\beta, t_{int}))\qquad IE(t_{pair}, c) = (c, c)}{\dfrac{IE \vdash (\beta, t_{int})t_{pair}(c) \Rightarrow \{\beta \to (c)\} \bowtie \{\}}{\dfrac{IE \vdash (\beta, t_{int})t_{pair}(c) \Rightarrow \{\beta \to (c)\}}{IE \vdash \alpha(c) \Rightarrow \alpha \to \alpha \succ (\beta, t_{int})t_{pair} \to (\beta, t_{int})t_{pair} \Rightarrow \{\beta \to (c)\}}}}}$$

Gleichheit auf Typschemata

Zwei Typschemata σ und σ' werden als gleich betrachtet, wenn sie durch Umbenennen und Umordnen der gebundenen Typvariablen ineinander überführt werden können. Dabei dürfen Typvariablen, die nicht im Rumpf des Typschematas vorkommen weggelassen werden. Zum Beispiel sind

$$\forall\alpha_1\Gamma_1.\alpha_1 \to \alpha_1$$

und

$$\forall\alpha_2\Gamma_2.\alpha_2 \to \alpha_2$$

nur dann gleich, wenn $\Gamma_1 = \Gamma_2$.

Generalisieren

Ein Typschema $\sigma = \forall(\alpha\ \Gamma)^{(k)}.\tau$ generalisiert einen Typ τ unter der Umgebung *IE*, wenn ein *TC* existiert, so daß $IE \vdash \sigma \succ \tau \Rightarrow TC$ gilt.

Ein Typschema σ generalisiert ein Typschema σ' unter der Umgebung *IE* wenn für alle τ'' gilt:

$$IE \vdash \sigma' \succ \tau'' \Rightarrow TC' \text{ dann gilt } (IE \vdash \sigma \succ \tau'' \Rightarrow TC \text{ und } TC = TC \bowtie TC')$$

Die Bedingung $TC = TC \bowtie TC'$ besagt, daß die Constraints von σ allgemeiner sein müssen, als die von σ'.

4.1.6 Geltungsbereich von expliziten Typvariablen

Am Geltungsbereich von expliziten und impliziten Typvariablen ändert sich nichts. An explizite Typvariablen können in Wertbindungen Klassenconstraints annotiert werden.

4.1.7 Nicht-expansive Ausdrücke

An der Definition der nicht-expansiven Ausdrücke von Standard ML in Abschnitt 4.7 hat sich nichts geändert.

4.1.8 Abschluß

Sei τ ein Typ und *A* ein semantisches Objekt. Dann ist $Clos^A_{TC}(\tau)$, der Abschluß von τ hinsichtlich *A* und *TC*, das Typschemata $\forall(\alpha\ TC(\alpha))^{(k)}.\tau$, wobei $\alpha^{(k)} \in tyvars(\tau)\backslash tyvars(A)$. Der totale Abschluß $Clos^{TC}_{\{\}}\tau$ wird mit $Clos^{TC}\tau$ abgekürzt. Für Wertumgebungen *VE* deren Kodomäne nur Typen enthält, schreiben wir

$$Clos_A^{TC} VE = \{vid \mapsto (Clos_A^{TC}\tau, is); VE(vid) = (\tau, is)\}$$

Beim Abschließen einer Wertumgebung, werden die Typvariablen, die in expansiven Ausdrücken vorkommen, nicht generalisiert.

Wenn

$$VE(vid) = (\tau, is),$$

dann ist

$$Clos_{C,valbind}^{TC}(VE(vid)) = (\forall(\alpha\ TC(\alpha))^{(k)}.\tau, is),$$

wobei

$$\alpha^{(k)} = \begin{cases} tyvars(\tau)\backslash tyvars(C), & \text{falls } exp \text{ nicht-expansiv in } C \\ (), & \text{falls } exp \text{ expansiv in } C \end{cases}$$

4.1.9 Typstrukturen und Typumgebungen

Für Typstrukturen und Typumgebungen werden die Definitionen aus der Definition von Standard ML, Abschnitt 4.9 übernommen. Kriterien, die die Gleichheit betreffen, entfallen einfach.

4.1.10 Inferenzregeln der Kernsprache

Die Numerierung der Inferenzregeln orientiert sich an der der Definition von SML. Dabei ersetzt eine Regel mit der Nummer (n) die Regel mit der Nummer (n) in der Definition von SML. Neue Regeln wurden mit $(C1)$, $(C2)$, usw. durchnumeriert. Dieses Numerierungsschema gilt auch im späteren Teil über die statische Semantik der Modulsprache.

Obwohl nicht in den Inferenzregeln annotiert, wird immer angenommen, daß für jeden Kontext $C = T, K, U, E$ gilt:

$$tynamesE \subseteq T$$

$$classnamesE \subseteq T$$

Intuitiv bedeutet dies, daß T jeden generierten Typnamen und K jeden generierten Klassennamen enthält. Es ist notwendig T und K explizit im Kontext aufzunehmen, weil in *tynamesE* und in *classnamesE* nicht alle Typnamen bzw. Klassennamen enthalten sein müssen. Ein Grund dafür ist, daß der Kontext $T, \emptyset, \emptyset, E$ eine Projektion der Basis $B = T, K, F, G, E$ ist, deren Komponenten F und G weitere Typnamen und Klassennamen enthalten können, die in T bzw. K aufgenommen wurden, aber nicht in E enthalten sind.

Der folgende Satz kann leicht gezeigt werden:

Ist S ein Judgement $T, K, U, E \vdash phrase \Rightarrow A$, so daß $tynamesE \subseteq T$, und ist S' ein Judgement $T', K'U', E' \vdash phrase' \Rightarrow A'$, das in einem Beweis von S vorkommt, dann gilt:

$$tynamesE' \subseteq T' \text{ und } classnamesE' \subseteq T'$$

Dabei sei *phrase* entweder ein Satz der Kernsprache oder der Klassensprache.

Bei den Regeln für Ausdrücke und Muster ändert sich die rechte Seite. Neben dem Typ des atomaren Ausdrucks werden in TML die beim Instantiieren von Typschemata entstehenden Typvariablenconstraints hergeleitet.

Atomare Ausdrücke $\boxed{C \vdash atexp \Rightarrow \tau, TC}$

$$\frac{}{C \vdash scon \Rightarrow type(scon), \{\}} \quad (1)$$

$$\frac{C(longvid) = (\sigma, is) \qquad IE \text{ of } C \vdash \sigma \succ \tau \Rightarrow TC}{C \vdash longvid \Rightarrow \tau, TC} \quad (2)$$

In Regel (2) erfolgt die Instantiierung des Typschemas unter Berücksichtigung der Instanzumgebung *IE*. Nur durch diese Regel können neue Constraints auf Typvariablen eingeführt werden.

$$\frac{C \vdash \langle exprow \Rightarrow \varrho, TC\rangle}{C \vdash \{\langle exprow\rangle\} \Rightarrow \{\}\langle +\varrho\rangle \text{ in Type}, \{\}\langle \rhd\!\lhd TC\rangle} \quad (3)$$

$$\frac{C \vdash dec \Rightarrow E, TC \quad C \oplus E \vdash exp \Rightarrow \tau, TC' \quad tynamesE \subseteq T \text{ of } C}{C \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Rightarrow \tau, TC \bowtie TC'} \quad (4)$$

$$\frac{C \vdash exp \Rightarrow \tau, TC}{C \vdash \texttt{(}exp\texttt{)} \Rightarrow \tau, TC} \quad (5)$$

Ausdrucksreihen $\boxed{C \vdash exprow \Rightarrow \varrho, TC}$

$$\frac{C \vdash exp \Rightarrow \tau, TC \quad \langle C \vdash exprow \Rightarrow \varrho, TC'\rangle}{C \vdash lab\texttt{=}exp\langle\texttt{,}\ exprow\rangle \Rightarrow \tau, TC \bowtie TC'} \quad (6)$$

Ausdrücke $\boxed{C \vdash exp \Rightarrow \tau, TC}$

$$\frac{C \vdash atexp \Rightarrow \tau, TC}{C \vdash atexp \Rightarrow \tau, TC} \quad (7)$$

$$\frac{C \vdash exp \Rightarrow \tau' \to \tau, TC \quad C \vdash atexp \Rightarrow \tau', TC'}{C \vdash exp\ atexp \Rightarrow \tau, TC \bowtie TC'} \tag{8}$$

In der Regel (8) können Typvariablen verschwinden, das heißt

$$tyvars(\tau') \backslash (tyvars(\tau) \cap tyvars(\tau')) \neq \emptyset.$$

Trotzdem bleiben die Constraints auf diesen Typvariablen in *TC* erhalten.

$$\frac{C \vdash exp \Rightarrow \tau, TC \quad C \vdash ty \Rightarrow \tau}{C \vdash exp\ \texttt{:}\ ty \Rightarrow \tau, TC} \tag{9}$$

$$\frac{C \vdash exp \Rightarrow \tau, TC \quad C \vdash match \Rightarrow exn \to \tau, TC'}{C \vdash exp\ \texttt{handle}\ match \Rightarrow \tau, TC \bowtie TC'} \tag{10}$$

$$\frac{C \vdash exp \Rightarrow exn}{C \vdash \texttt{raise}\ exp \Rightarrow \tau, \{\}} \tag{11}$$

$$\frac{C \vdash match \Rightarrow \tau, TC}{C \vdash \texttt{fn}\ match \Rightarrow \tau, TC} \tag{12}$$

Muster $\boxed{C \vdash mrule \Rightarrow \tau, TC}$

$$\frac{C \vdash mrule \Rightarrow \tau, TC \quad \langle C \vdash match \Rightarrow \tau, TC' \rangle}{C \vdash mrule\ \langle \texttt{|}\ match \rangle \Rightarrow \tau, TC \bowtie TC'} \tag{13}$$

Musterregel $\boxed{C \vdash match \Rightarrow \tau, TC}$

$$\frac{C \vdash pat \Rightarrow (VE, \tau) \quad C + VE \vdash exp \Rightarrow \tau', TC \quad tynamesVE \subseteq T \text{ of } C}{C \vdash pat\ \texttt{=>}\ exp \Rightarrow \tau \to \tau', TC} \tag{14}$$

Deklarationen $\boxed{C \vdash dec \Rightarrow E, TC}$

$$\frac{\begin{array}{ll} TC' = TC_{|tyvars(VE')} & \\ \langle C \vdash constraints \Rightarrow TC'' \rangle & \langle TC \equiv TC'' \rangle \\ U = tyvars(tyvarseq) & U \cap tyvars(VE') = \emptyset \\ C + U \vdash valbind \Rightarrow VE, TC & VE' = Clos^{TC}_{C,valbind} VE \end{array}}{C \vdash \texttt{val}\ \langle constraints\ \texttt{=>} \rangle\ tyvarseq\ valbind \Rightarrow VE' \text{ in Env}, TC'} \tag{15}$$

In Regel (15) werden beim Abschließen der Wertumgebung *VE* zu VE' die Typvariablenconstraints *TC* in den Typschemata an die generalisierten Typvariablen annotiert. TC' ist die Einschränkung der Typvariablenconstraints *TC* auf die freien Typvariablen von *VE*.

$$\frac{C \vdash \mathit{typbind} \Rightarrow \mathit{TE}}{C \vdash \texttt{type}\ \mathit{typbind} \Rightarrow \mathit{TE}\ \text{in Env}, \{\}} \tag{16}$$

Da man in Regel (16) und einigen der folgenden Regeln nicht zu Ausdrücken absteigen kann, entstehen keine Constraints of Typvariablen. Somit ist *TC* auf der rechten Seite immer leer.

$$\frac{C \oplus \mathit{TE} \vdash \mathit{datbind} \Rightarrow \mathit{VE}, \mathit{TE} \quad \forall (t, \mathit{VE}') \in \mathrm{Ran}\mathit{TE}, t \notin (T \text{ of } C)}{C \vdash \mathit{datatype}\ \mathit{datbind} \Rightarrow (\mathit{VE}, \mathit{TE})\ \text{in Env}, \{\}} \tag{17}$$

$$\frac{C(\mathit{longtycon}) = (\Theta, \mathit{VE}) \qquad \mathit{TE} = \{\mathit{tycon} \mapsto (\Theta, \mathit{VE})\}}{C \vdash \texttt{datatype}\ \mathit{tycon} = \texttt{datatype}\ \mathit{longtycon} \Rightarrow (\mathit{VE}, \mathit{TE})\ \text{in Env}, \{\}} \tag{18}$$

$$\frac{\begin{array}{c} C \oplus \mathit{TE} \vdash \mathit{datbind} \Rightarrow \mathit{VE}, \mathit{TE} \qquad \forall (t, \mathit{VE}') \in \mathrm{Ran}\mathit{TE},\ t \notin (T \text{ of } C) \\ C \oplus (\mathit{VE}, \mathit{TE}) \vdash \mathit{dec} \Rightarrow E, \mathit{TC} \end{array}}{C \vdash \texttt{abstype}\ \mathit{tycon}\ \texttt{with}\ \mathit{dec}\ \texttt{end} \Rightarrow \mathrm{Abs}(\mathit{TE}, E), \mathit{TC}} \tag{19}$$

In Regel (17) und (19) entfällt die Seitenbedingung, daß *TE* Gleichheit maximiert.

$$\frac{C \vdash \mathit{exbind} \Rightarrow \mathit{VE}}{C \vdash \texttt{exception}\ \mathit{exbind} \Rightarrow \mathit{VE}\ \text{in Env}, \{\}} \tag{20}$$

$$\frac{C \vdash \mathit{dec}_1 \Rightarrow E_1, \mathit{TC}_1 \quad C \oplus E_1 \vdash \mathit{dec}_2 \Rightarrow E_2, \mathit{TC}_2}{C \vdash \texttt{local}\ \mathit{dec}_1\ \texttt{in}\ \mathit{dec}_2\ \texttt{end} \Rightarrow E_2, \mathit{TC}_1 \bowtie \mathit{TC}_2} \tag{21}$$

$$\frac{\begin{array}{c} C(\mathit{longstrid}_1) = E_1 \quad \ldots \quad C(\mathit{longstrid}_n) = E_n \\ E_i' = E_i - (\mathit{IE} \text{ of } E_i) - (\mathit{CE} \text{ of } E_i) \end{array}}{C \vdash \texttt{open}\ \mathit{longstrid}_1 \ldots\ \mathit{longstrid}_n \Rightarrow E_1' + \ldots + E_n', \{\}} \tag{22}$$

Regel (22) führte keine Instanzen und Klassen ein. TML unterscheidet zwischen `open` auf `dec` und `strdec` Ebene.

$$\frac{}{C \vdash \quad \Rightarrow \{\}\ \text{in Env}, \{\}} \tag{23}$$

$$\frac{C \vdash \mathit{dec}_1 \Rightarrow E_1, \mathit{TC}_1 \quad C \oplus E_1 \vdash \mathit{dec}_2 \Rightarrow E_2, \mathit{TC}_2}{C \vdash \mathit{dec}_1\ \langle\texttt{;}\rangle\ \mathit{dec}_2 \Rightarrow E_1 + E_2, \mathit{TC}_1 + \mathit{TC}_2} \tag{24}$$

Am Rest der statischen Semantik der Kernsprache ändert sich nichts.

4.2 Statische Semantik der Klassensprache

Für die Klassensprache werden keine weiteren semantischen Objekte benötigt.

4.2.1 Inferenzregeln der Klassensprache

Klassendeklarationen $\boxed{C \vdash \mathit{classdec} \Rightarrow E}$

$$\frac{C \vdash \mathit{classbind} \Rightarrow CE, VE}{C \vdash \texttt{class}\ \mathit{classbind} \Rightarrow (CE, VE)\ \text{in Env}} \tag{C.1}$$

$$\frac{C(\mathit{longclassid}) = (c, \xi\alpha.VE, cs) \qquad VE' = Clos^{\{\alpha \to (c)\}} VE}{\begin{array}{c} C \vdash \texttt{class}\ \mathit{classid} = \texttt{class}\ \mathit{longclassid} \Rightarrow \\ \{\mathit{classid} \to (c, \xi\alpha.VE, cs)\}, VE')\ \text{in Env} \end{array}} \tag{C.2}$$

In Regel (C.2) wird die Wertumgebung *VE* über der Klassentypvariable abgeschlossen, weil die Methoden in der Wertumgebung später in den Kontext *C* eingeführt werden und Methoden im Kontext nicht von Werten unterschieden werden. Der Constraint über der Klassentypvariable berücksichtigt die Tatsache, daß bei der Verwendung der Methode die entsprechende Instanz ihrer Klasse im Skopus sein muß.

$$\frac{C + IE \vdash \mathit{instbind} \Rightarrow IE}{C \vdash \texttt{instance}\ \mathit{instbind} \Rightarrow IE\ \text{in Env}} \tag{C.3}$$

Das Modifizieren von *C* durch *IE* in Regel (C.2) entspricht der rekursiven Natur von Instanzbindungen.

$$\frac{\begin{array}{cc} C \vdash \mathit{instty} \Rightarrow \tau & C(\mathit{longclassid}) = (c, \xi\alpha.VE, cs) \\ \kappa(\tau) = (\iota, \alpha^{(k)}) & C(\iota, c) = (\Gamma_1, \ldots, \Gamma_k) \end{array}}{C \vdash \texttt{instance}\ \mathit{instty}\ \mathit{longclassid} \Rightarrow \{(\iota, c) \to \Gamma^{(k)}\}\ \text{in Env}} \tag{C.4}$$

In Regel (C.4) werden beim Replizieren die Methoden der zugehörigen Klasse nicht in den Skopus eingeführt. Instanzen können also in den Skopus eingeführt werden, ohne daß die Methoden der Klasse im Skopus sind.

Klassenbindung $\boxed{C \vdash \mathit{classbind} \Rightarrow CE, VE}$

$$\frac{\begin{array}{c} \mathit{tyvar} = \alpha \qquad c \notin K\ \text{of}\ C \qquad TC = \{\alpha \mapsto (c)\} \\ C, \alpha \vdash \mathit{methspec} \Rightarrow VE \end{array}}{\begin{array}{c} C \vdash \mathit{tyvar}\ \mathit{classid}\ \texttt{with}\ \mathit{methspec}\ \texttt{end} \Rightarrow \\ \{\mathit{classid} \to (c, \xi\alpha.VE, \mathrm{k})\}, Clos^{TC} VE \end{array}} \tag{C.5}$$

Beim Binden einer Klasse in Regel (C.5) wird analog zu Regel (C.2) die Wertumgebung über der Klassentypvariable abgeschlossen, bevor sie später in den Kontext *C* eingeführt wird.

Methodenspezifikation $\boxed{C, \alpha \vdash \mathit{methspec} \Rightarrow VE}$

$$\frac{}{C, \alpha \vdash \Rightarrow \{\}} \tag{C.6}$$

$$\frac{C \vdash \mathit{methdesc} \Rightarrow \mathit{VE}, \mathit{TC} \quad \mathit{TC}(\alpha) = () \quad \mathit{VE}' = \mathit{Clos}_{\alpha}^{\mathit{TC}}\mathit{VE}}{C, \alpha \vdash \texttt{val}\ \mathit{methdesc} \Rightarrow \mathit{VE}'} \tag{C.7}$$

Bei einer Methodenspezifikation in (C.7) dürfen keine Constraints über der Klassentypvariable spezifiziert werden. Implizit trägt die Klassentypvariable immer einen Constraint, der ihrer eigenen Klasse entspricht. Beim Abschließen der Wertumgebung wird die Klassentypvariable ausgeschlossen, so daß die Klassentypvariable in den Methodenspezifikationen frei ist.

$$\frac{\begin{array}{c}\mathrm{Dom}\mathit{VE}_1 \cap \mathrm{Dom}\mathit{VE}_2 = \emptyset \\ C, \alpha \vdash \mathit{methspec}_1 \Rightarrow \mathit{VE}_1 \quad C, \alpha \vdash \mathit{methspec}_2 \Rightarrow \mathit{VE}_2\end{array}}{C, \alpha \vdash \mathit{methspec}_1\ \langle\texttt{;}\rangle\ \mathit{methspec}_2 \Rightarrow \mathit{VE}_1 + \mathit{VE}_2} \tag{C.8}$$

Methodenbeschreibung $\boxed{C \vdash \mathit{methdesc} \Rightarrow \mathit{VE}, \mathit{TC}}$

$$\frac{C \vdash \mathit{qualty} \Rightarrow \tau, \mathit{TC} \qquad \langle C \vdash \mathit{methdesc} \Rightarrow \mathit{VE}, \mathit{TC}'\rangle}{C \vdash \mathit{vid}\ \texttt{:}\ \mathit{qualty}\ \langle\texttt{and}\ \mathit{methdesc}\rangle \Rightarrow \{\mathit{vid} \rightarrow (\tau, \texttt{v})\}\langle{+}\mathit{VE}\rangle, \mathit{TC}\langle\bowtie \mathit{TC}'\rangle} \tag{C.9}$$

Instanzbindungen $\boxed{C \vdash \mathit{instbind} \Rightarrow \mathit{IE}}$

$$\frac{\begin{array}{c}C \vdash \mathit{instty} \Rightarrow \tau \\ C(\mathit{longclassid}) = (c, \xi\beta.\mathit{VE}, \mathrm{k}) \\ \kappa(\tau) = (\iota, \alpha^{(k)}) \\ \langle C \vdash \mathit{constraints} \Rightarrow \mathit{TC}\rangle \\ \langle \mathit{tyvars}(\mathit{TC}) \subseteq \mathit{tyvars}(\tau)\rangle \\ \Gamma^{(k)} = (\{\}\langle \bowtie\ \mathit{TC}(\alpha_1)\rangle, \ldots, \{\}\langle \bowtie\ \mathit{TC}(\alpha_k)\rangle) \\ C + \mathit{Clos}^{\{\beta \mapsto (c)\}}\mathit{VE} \vdash \mathit{methdec} \Rightarrow \mathit{VE}' \\ \mathit{IE}\ \text{of}\ C \vdash \mathit{Clos}^{\{\}\langle\bowtie \mathit{TC}\rangle}((\tau)\mathit{VE}) = \mathit{VE}' \\ \langle\langle C \vdash \mathit{instbind} \Rightarrow \mathit{IE}'\rangle\rangle\end{array}}{\begin{array}{l}C \vdash \texttt{instance}\ \langle\mathit{constraints}\ \texttt{=>}\rangle\ \mathit{instty}\ \mathit{longclassid} \\ \quad \texttt{with}\ \mathit{methdec}\ \texttt{end}\ \langle\langle\texttt{and}\ \mathit{instbind}\rangle\rangle \Rightarrow \\ \quad \{(t, c) \rightarrow \Gamma^{(k)}\}\langle\langle{+}\mathit{IE}'\rangle\rangle\end{array}} \tag{C.10}$$

Die Regel (C.10) wird von oben nach unten erklärt:

1. Zunächst wird der *instty* zum Instanztyp elaboriert.
2. Im Kontext C wird mittels die Klasse *longclassid* nachgeschlagen.
3. Enthält der Instanzkopf einen Instanzkontext, so wird dieser elaboriert. Das Ergebnis ist die Typvariablenconstraintumgebung *TC*. Die in *TC* aufgesammelten Constraints dürfen von den Methodenimplementationen verwendet werden.
4. Dabei müssen alle Typvariablen im Instanzkontext *constraints* im Instanztyp vorkommen. Es gilt die syntaktische Einschränkung, daß die Typvariablen im Instanztyp paarweise disjunkt sind.

5. $\Gamma^{(k)}$ steht für die geordneten Instanzconstraints.

6. Bevor die Methodendeklarationen elaboriert werden, werden die Methoden innerhalb der Instanz sichtbar gemacht. Da bei der Methodendeklaration die Methoden als ganz normale Wertbezeichner verwendet werden sollen, werden die Methodentypen vor dem Modifizieren über der Klassentypvariable abgeschlossen. Die Klassentypvariable erhält dabei die eigene Klasse als Constraint.

7. Die vorletzte Prämisse verifiziert, daß die Typen der Methodendeklarationen den in der Klassendeklaration spezifizierten Typen entsprechen, nachdem die Klassentypvariable durch den Instanztyp ersetzt wurde. Die Typvariablen im Instanztyp tragen nach dem Abschluß die Constraints in *constraints*.
   In der statischen Semantik würde es genügen statt der Gleichheit nur zu verlangen, daß die Methodendeklarationen polymorpher sind als die Methodenspezifikationen in der Klasse. Dies würde aber für die im folgenden Kapitel definierte dynamische Semantik bedeuten, daß für die Übergabe der Methodentabellen Anpassungscode eingefügt werden müßte.
   Die Gleichheit an dieser Stelle verträgt sich nicht mit dem Prinzip, daß die semantischen Regeln für Ausdrücke die benötigten Constraints herleiten. Aus diesem Grund mußte in Regel (C.12) erlaubt werden, daß beliebige Constraints hinzugenommen werden dürfen.

8. Die letzte Prämisse elaboriert weitere Instanzdeklarationen.

Methodendeklaration $\boxed{C \vdash \mathit{methdec} \Rightarrow \mathit{VE}}$

$$\frac{}{C \vdash \Rightarrow \{\}} \tag{C.11}$$

$$\frac{\begin{array}{ll}\langle C \vdash \mathit{constraints} \Rightarrow TC' \rangle & \langle TC \equiv TC' \rangle \\ U = \mathit{tyvars}(\mathit{tyvarseq}) & C + U \vdash \mathit{valbind} \Rightarrow \mathit{VE}, TC \\ \mathit{VE}' = \mathit{Clos}^{TC \bowtie TC''}_{C,\mathit{valbind}} \mathit{VE} & \mathit{tyvars}(\mathit{VE}') = \{\}\end{array}}{C \vdash \texttt{val}\ \langle \mathit{constraints}\ \texttt{=>} \rangle\ \mathit{tyvarseq}\ \mathit{valbind} \Rightarrow \mathit{VE}'} \tag{C.12}$$

In Regel (C.12) dürfen beliebige Typvariablenconstraints aufgenommen werden. Dies ist notwendig, damit in Regel (C.10) die Gleichheit auf den Wertumgebungen verlangt werden kann. Die Methoden werden in dieser und in Regel (C.13) nicht in den Kontext C eingeführt, so daß nie die in Regel (C.10) in den Kontext eingeführten Methoden überschrieben werden.

$$\frac{\begin{array}{c}C \vdash \mathit{methdec}_1 \Rightarrow \mathit{VE}_1 \qquad C \vdash \mathit{methdec}_2 \Rightarrow \mathit{VE}_2 \\ \mathrm{Dom}\mathit{VE}_1 \cap \mathrm{Dom}\mathit{VE}_2 = \emptyset\end{array}}{C \vdash \mathit{methdec}_1\ \langle\texttt{;}\rangle\ \mathit{methdec}_2 \Rightarrow \mathit{VE}_1 + \mathit{VE}_2} \tag{C.13}$$

Klassenannahmen $\boxed{C \vdash \mathit{constraints} \Rightarrow TC}$

$$\frac{\mathit{tyvar} = \alpha \quad C(\mathit{longclassid}) = (c, \xi\beta.VE, cs)}{C \vdash \mathit{tyvar}\ \mathit{longclassid} \Rightarrow \{\alpha \mapsto (c)\}} \tag{C.14}$$

$$\frac{C \vdash \mathit{constraint}_i \Rightarrow TC_i \quad \forall i \in \{1, \ldots, n\} \quad n \geq 1}{C \vdash (\mathit{constraint}_1, \ldots, \mathit{constraint}_n) \Rightarrow \bowtie_{1 \leq i \leq n} TC_i} \tag{C.15}$$

In den Regeln (C.14) und (C.15) werden die Klassenconstraints elaboriert. Dabei ist die Reihenfolge beim Aufsammeln für die in Kapitel 5 definierte dynamische Semantik von Bedeutung.

Qualifizierte Typen $\boxed{C \vdash \mathit{qualty} \Rightarrow \tau, TC}$

$$\frac{\langle C \vdash \mathit{constraints} \Rightarrow TC\rangle \qquad C \vdash \mathit{ty} \Rightarrow \tau}{C \vdash \langle \mathit{constraints}\texttt{=>}\rangle\ \mathit{ty} \Rightarrow \tau, \{\}\langle + TC\rangle} \tag{C.16}$$

Instanztypen $\boxed{C \vdash \mathit{instty} \Rightarrow \tau}$

$$\frac{\mathit{tyvar} = \alpha}{C \vdash \mathit{tyvar} \Rightarrow \alpha} \tag{C.17}$$

$$\frac{\langle C \vdash \mathit{insttyrow} \Rightarrow \rho\rangle}{C \vdash \{\langle \mathit{insttyrow}\rangle\} \Rightarrow \{\}\langle + \rho\rangle} \tag{C.18}$$

$$\frac{C \vdash \mathit{ty} \Rightarrow \tau \qquad C \vdash \mathit{ty}' \Rightarrow \tau'}{C \vdash \mathit{ty}\ \texttt{->}\ \mathit{ty}' \Rightarrow \tau \to \tau'} \tag{C.19}$$

Instanztypreihen $\boxed{C \vdash \mathit{insttyrow} \Rightarrow \tau}$

$$\frac{C \vdash \mathit{ty} \Rightarrow \tau \quad \langle C \vdash \mathit{tyrow} \Rightarrow \rho\rangle}{C \vdash \mathit{lab}\ \texttt{:}\ \mathit{ty}\ \langle\ \texttt{,}\ \mathit{tyrow}\rangle \Rightarrow \{\mathit{lab} \mapsto \tau\}\langle + \rho\rangle} \tag{C.20}$$

4.3 Statische Semantik der Modulsprache

Dieser Abschnitt beschreibt die Änderungen gegenüber der Definition von Standard ML.

4.3.1 Semantische Objekte

Grundsätzlich ändern sich die semantischen Objekte der Modulsprache nicht. Jedoch bindet ein Präfix jetzt nicht mehr nur Typnamen, sondern auch Klassennamen.

Konkret bedeutet dies, daß in der Definition von Standard ML, Abbildung 11, (T) durch (T, K), (T') durch (T', K') und TyNameSet durch TyNameSet $\times$ ClassNameSet ersetzt werden.

4.3.2 Realisierung

Die Definitionen zu Typrealisierungen φ werden aus der Definition von Standard ML übernommen. Eine *Klassenrealisierung* ist eine Abbildung ζ : ClassName $\rightarrow$ ClassName. Analog zu Typrealisierung werden *Träger* Supp ζ und *Ausmaß* Yield ζ definiert. Eine Realisierung ist eine Abbildung

$$\psi : \text{ClassName } \dot{\cup} \text{ Tyname} \rightarrow \text{ClassName } \dot{\cup} \text{ TypeFcn}$$

wobei

$$\psi(x) = \begin{cases} \varphi(x) & \text{falls } x \text{ Typname} \\ \zeta(x) & \text{falls } x \text{ Klassenname} \end{cases}$$

Realisierungen ψ werden auf alle semantischen Objekte erweitert. Sie dienen dazu jeden Typnamen t durch $\psi(t)$ und jeden Klassennamen γ durch $\psi(\gamma)$ zu ersetzen. Bevor eine Realisierung auf ein semantisches Objekt mit gebundenen Namen angewendet werden kann, müssen die gebundenen Namen umbenannt werden, so daß für jeden Präfix (T, K) gilt:

$$(T \dot{\cup} K) \cap (\text{Supp } \psi \cup \text{Yield} \psi)$$

4.3.3 Signaturinstantiierung

Eine Umgebung E_2 *ist eine Instanz der Signatur* $\Sigma_1 = (T_1, K_1)E_1$, geschrieben $\Sigma_1 \geq E_2$, wenn eine Realisierung ψ existiert, so daß $\psi(E_1) = E_2$ und Supp $\psi \subseteq (T_1 \dot{\cup} K_1)$.

4.3.4 Funktorinstantiierung

Die Funktorinstantiierung wird analog zu der Signaturinstantiierung geändert.

4.3.5 Bereicherung

Beim Abgleich einer Umgebung mit einer Signatur ist es erlaubt, daß die Umgebung mehr Komponenten enthält und daß die Umgebung polymorpher ist, als eine Instanz der Signatur.

Bereicherung wird wie folgt über Umgebungen und Typstrukturen rekursiv definiert: Eine Umgebung

$$E_1 = (SE_1, TE_1, VE_1, CE_1, IE_1)$$

bereichert eine Umgebung

$$E_2 = (SE_2, TE_2, VE_2, CE_2, IE_2)$$

unter der Umgebung *IE*, geschrieben, $IE \vdash E_1 \succ E_2$, wenn

1. Dom $SE_1 \supseteq$ Dom SE_2, und $IE \vdash SE_1(strid) \succ SE_2(strid)$ für alle $strid \in$ Dom SE_2
2. Dom $TE_1 \supseteq$ Dom TE_2, und $TE_1(tycon) \succ TE_2(tycon)$ für alle $tycon \in$ Dom TE_2
3. Dom $VE_1 \supseteq$ Dom VE_2, und $IE \vdash VE_1(vid) \succ VE_2(vid)$ für alle $vid \in$ Dom VE_2 wobei $IE \vdash (\sigma_1, is_1) \succ (\sigma_2, is_2)$ falls $IE \vdash \sigma_1 \succ \sigma_2$ und $is_1 = is_2$ oder $is_2 = \mathrm{v}$
4. Dom $CE_1 \supseteq$ Dom CE_2, und $CE_1(classid) \succ CE_2(classid)$ für alle $classid \in$ Dom CE_2
5. Dom $IE_1 \supseteq$ Dom IE_2, und $IE_1(insttyfcn, classname) = IE_2(insttyfcn, classname)$ für alle $(insttyfcn, classname) \in$ Dom IE_2

Eine Typstruktur (Θ_1, VE_1) bereichert eine Typstruktur (Θ_2, VE_2), geschrieben

$$(\Theta_1, VE_1) \succ (\Theta_2, VE_2),$$

wenn gilt:

1. $\Theta_1 = \Theta_2$
2. Entweder $VE_1 = VE_2$ oder $VE_2 = \{\}$

Eine Klassenstruktur $(\gamma_1, \xi\alpha.VE_1, cs_1)$ bereichert eine Klassenstruktur $(\gamma_2, \xi\beta.VE_2, cs_2)$, wenn

1. $\gamma_1 = \gamma_2$
2. Entweder $((t)VE_1)(vid) = ((t)VE_2(vid))$ für alle $vid \in VE_2$ wobei t ein neuer Typname oder $cs_2 = \mathrm{a}$

4.3.6 Signaturabgleich

Eine Signatur Σ_1 gleicht eine Umgebung E unter IE ab, wenn eine Umgebung E^- existiert, so daß $\Sigma_1 \geq E^-$ und $IE \vdash E^- \prec E$. Damit ist der Signaturabgleich eine Kombination aus Instantiieren von Signaturen und Bereicherung von Umgebungen. Sind Σ_1, E und IE gegeben, so gibt es höchstens ein E^-.

4.3.7 Prinzipale Umgebungen

Für diesen Abschnitt soll $E \succ E'$ bedeuten, daß zwischen den Umgebungen gilt:
Sei $VE = VE$ of E und $VE' = VE$ of E',

$$\mathrm{Dom}VE = \mathrm{Dom}VE' \text{ und } \forall vid \in \mathrm{Dom}VE : VE(vid) \succ VE'(vid)$$

Ist C ein Kontext und angenommen, es gilt $C \vdash dec \Rightarrow E$. Dann ist

$$E \textit{ principal (für dec im Kontext C)},$$

wenn für alle E', mit $C \vdash dec \Rightarrow E'$, gilt:

$$E \succ E'$$

In Regel (56) wird verlangt, daß wenn *dec* überhaupt zu einer Umgebung E in C elaboriert, dann elaboriert *dec* in C zu einer prinzipalen Umgebung.

Prinzipale Umgebungen waren bereits in der Definition von SML '90 [MTH90], sind aber in der Definition '97 weggefallen. In TML werden prinzipale Umgebungen wieder benötigt, damit mit Hilfe der statischen Semantik, jedem gültigen Programm eine eindeutige dynamische Semantik zugeordnet werden kann.

Der Grund wird anhand des folgenden Beispiels verdeutlicht:

```
class 'a C with
  val m : 'a -> 'a
end

instance int C ... (* 1. Instanz *)
local
  structure ...
  local
    instance int C ... (* 2. Instanz *)
  in
    fun f x = m x
  end
in
  val z = f 1
end
```

Nach der Definition von SML könnte für die Funktion `f` der Typ `int -> int` hergeleitet werden, da die Funktion nur auf ganze Zahlen appliziert wird. Unter diesem Typ würde die zweite Instanz statisch gebunden. Der allgmeinste Typ, der `f` zugeordnet werden kann, ist `'a C => 'a -> 'a`. Dies bedeutet aber, daß die Instanz dynamisch gebunden und somit bei der Applikation `f 1` die erste Instanz gebunden wird.

4.3.8 Inferenzregeln der Modulsprache

In jeder Basis $B = T, K, F, G, E$, in der ein *topdec* elaboriert wird, gilt:

$$\textit{tynames}F \cup \textit{tynames}G \cup \textit{tynames}E \subseteq T$$
$$\textit{classnames}F \cup \textit{classnames}G \cup \textit{classnames}E \subseteq K$$

Es gelten die folgenden Sätze:

- Ist S ein Judgement $B \vdash \mathit{topdec} \Rightarrow B'$, bei der B obige Bedingungen erfüllt, so erfüllt auch B' obige Bedingungen.
- Ist S' ein Judgement $B'' \vdash \mathit{phrase} \Rightarrow A$, wobei *phrase* ein Satz der Modulsprache und wobei S' irgendwo im Beweis von S vorkommt, so erfüllt B'' ebenfalls obige Bedingungen.
- Kommt $T, K, U, E \vdash \mathit{phrase} \Rightarrow A$ irgendwo im Beweis von S vor und ist *phrase* ein Satz der Modul-, Klassen- oder Kernsprache, so gilt $\mathit{tynamesE} \subseteq T$ und $\mathit{classnamesE} \subseteq K$.

Strukturdeklarationen $\boxed{B \vdash \mathit{strdec} \Rightarrow \mathrm{Env}}$

Regel (C.21) wird den Strukturdeklarationsregeln der Definition von Standard ML hinzugefügt.

$$\frac{C \text{ of } B \vdash \mathit{classdec} \Rightarrow E}{B \vdash \mathit{classdec} \Rightarrow E} \tag{C.21}$$

$$\frac{C \text{ of } B \vdash \mathit{dec} \Rightarrow E, \{\} \qquad E \textit{ principal für dec in } (C \textit{ of } B)}{C \vdash \mathit{classdec} \Rightarrow \mathit{dec} \Rightarrow E} \tag{56}$$

In Regel (56) besagt die Prämisse, daß die Typschemata in E so allgemein wie möglich sind. Dies ist notwendig, da durch die Möglichkeit, Instanzen auf Strukturebene zu deklarieren, ansonsten keine eindeutige dynamische Semantik für TML mit Hilfe der statische Semantik angegeben werden könnte.

Die Prämisse führt dazu, daß es Programme gibt, die in SML wohlgeformt sind, aber nicht in TML. Ein Beispiel dafür ist folgendes Programm:

```
val f = (fn x => x) (fn x => x)
structure A = struct end
val y = f 7
```

Mit der Struktur wird erzwungen, daß die Deklarationen Strukturdeklarationen sind, so daß die Prämisse in Regel (56) bei jeder Deklaration gelten muß. `f` kann aber wegen der Wertrestriktion kein prinzipaler Typ zugewiesen werden, so daß dieses Programm in TML ungültig ist.

In SML gibt es diese Prämisse nicht, so daß der für die freie Typvariable einzusetzende Typ durch den Kontext festgelegt werden kann. In diesem Beispiel wird durch die Applikation `f 7` festgelegt, daß für die freie Typvariable der Typ `int` einzusetzen ist.

Strukturausdrücke $\boxed{B \vdash \mathit{strexp} \Rightarrow E}$

$$\frac{B \vdash \mathit{strexp} \Rightarrow E \quad B \vdash \mathit{sigexp} \Rightarrow \Sigma \quad \Sigma \geq E' \quad IE \text{ of } B \vdash E' \prec E}{B \vdash \mathit{strexp} : \mathit{sigexp} \Rightarrow E'} \tag{52}$$

In Regel (52) wird der Signaturabgleich unter der *IE* Umgebung vorgenommen.

$$\frac{\begin{array}{cc} B \vdash \mathit{strexp} \Rightarrow E & B \vdash \mathit{sigexp} \Rightarrow (T', K')E' \\ (T', K')E' \geq E'' & \mathit{IE} \text{ of } B \vdash E'' \prec E \\ T' \cap (T \text{ of } B) = \emptyset & K' \cap (K \text{ of } B) = \emptyset \end{array}}{B \vdash \mathit{strexp} \texttt{ :> } \mathit{sigexp} \Rightarrow E'} \tag{53}$$

$$\frac{\begin{array}{c} B \vdash \mathit{strexp} \Rightarrow E \\ B(\mathit{funid}) \geq (E'', (T', K')E'), \mathit{IE} \text{ of } B \vdash E \succ E'' \\ (\mathit{tynames}\ E\ \cup T \text{ of } B) \cap T' = \emptyset \\ (\mathit{classnames}\ E\ \cup K \text{ of } B) \cap K' = \emptyset \end{array}}{B \vdash \mathit{funid} \texttt{ (} \mathit{strexp} \texttt{) } \Rightarrow E'} \tag{54}$$

In Regel (53), (54) und in einigen der folgenden Regeln unterliegen die Klassennamen den analogen Bedingungen wie die Typnamen. Zum Beispiel wird bei einem opaken Abgleich die Ergebnisumgebung mit neuen Klassennamen instantiiert.

Signaturausdrücke $\boxed{B \vdash \mathit{sigexp} \Rightarrow E}$

$$\frac{B(\mathit{sigid}) = (T, K)E \quad T \cap (T \text{ of } B) = \emptyset \quad K \cap (K \text{ of } B) = \emptyset}{B \vdash \mathit{sigid} \Rightarrow E} \tag{63}$$

$\boxed{B \vdash \mathit{sigexp} \Rightarrow \Sigma}$

$$\frac{\begin{array}{c} B \vdash \mathit{sigexp} \Rightarrow E \\ T = \mathit{tynamesE} \backslash (T \text{ of } B) \qquad K = \mathit{classnamesE} \backslash (K \text{ of } B) \end{array}}{B \vdash \mathit{sigexp} \Rightarrow (T, K)E} \tag{66}$$

Spezifikationen $\boxed{B \vdash \mathit{spec} \Rightarrow E}$

$$\frac{C \text{ of } B \vdash \mathit{TE} \qquad \forall (t, \mathit{VE}) \in \mathrm{Ran}\mathit{TE}}{B \vdash \texttt{type}\ \mathit{typdesc} \Rightarrow \mathit{TE} \text{ in Env}}$$

Die Regel (70) der Definition von Standard ML entfällt vollständig.

Zu den Spezifikationen der Definition von Standard ML werden folgende Regeln hinzugenommen.

$$\frac{C \text{ of } B \vdash \mathit{classdesc} \Rightarrow \mathit{CE}, \mathit{VE}}{B \vdash \texttt{class}\ \mathit{classdesc} \Rightarrow \mathit{CE}, \mathit{VE} \text{ in Env}} \tag{C.22}$$

$$\frac{B(\mathit{longclassid}) = (c, \xi\alpha.\mathit{VE})}{\begin{array}{l} B \vdash \texttt{class}\ \mathit{classid} = \texttt{class}\ \mathit{longclassid} \Rightarrow \\ \quad \{\mathit{classid} \rightarrow (c, \xi\alpha.\mathit{VE})\}, \mathit{Clos}^{\{\alpha \rightarrow (c)\}}\mathit{VE} \text{ in Env} \end{array}} \tag{C.23}$$

$$\frac{C \text{ of } B \vdash \mathit{instspec} \Rightarrow \mathit{IE}}{B \vdash \texttt{instance}\ \mathit{instspec} \Rightarrow \mathit{IE} \text{ in Env}} \tag{C.24}$$

Klassenspezifikation $\boxed{C \vdash \mathit{classdesc} \Rightarrow \mathit{CE}, \mathit{VE}}$

$$\frac{\mathit{tyvar} = \alpha \quad c \notin K \text{ of } C}{C \vdash \texttt{class}\ \mathit{tyvar}\ \mathit{classid} \Rightarrow \{\mathit{classid} \rightarrow (c, \emptyset)\}, \emptyset} \tag{C.25}$$

$$\frac{\begin{array}{c}\mathit{tyvar} = \alpha \qquad c \notin K \text{ of } C \qquad TC = \{\alpha \mapsto (c)\} \\ C, \alpha, c \vdash \mathit{methspec} \Rightarrow \mathit{VE}\end{array}}{\begin{array}{c}C \vdash \mathit{tyvar}\ \mathit{classid}\ \texttt{with}\ \mathit{methspec}\ \texttt{end} \Rightarrow \\ \{\mathit{classid} \rightarrow (c, \xi\alpha.\mathit{VE})\}, \mathit{Clos}^{TC}\mathit{VE}\end{array}} \tag{C.26}$$

Instanzspezifikation $\boxed{C \vdash \mathit{instspec} \Rightarrow \mathit{IE}}$

$$\frac{\begin{array}{rl} C \vdash \mathit{instty} \Rightarrow \tau & \kappa(\tau) = (\iota, \alpha^{(k)}) \\ \langle C \vdash \mathit{constraints} \Rightarrow TC \rangle & \langle \mathit{tyvars}(TC) \subseteq \mathit{tyvars}(\tau) \rangle \\ C(\mathit{longclassid}) = (c, \xi\beta.\mathit{VE}') & \gamma^{(k)} = (\{\}\langle \cup TC(\alpha_1)\rangle, \ldots, \{\}\langle \cup TC(\alpha_k)\rangle) \end{array}}{C \vdash \langle \mathit{constraints}\ \texttt{=>} \rangle\ \mathit{instty}\ \mathit{longclassid} \Rightarrow \{(\iota, c) \rightarrow \gamma^{(k)}\}} \tag{C.27}$$

4.4 Beispiele

4.4.1 Einfache Klassendeklaration

```
class 'a c with
  val m : 'a -> unit
end
```

$$\dfrac{\texttt{'a} = \alpha, \gamma \notin K \text{ of } C \qquad \dfrac{\dfrac{\dfrac{\dfrac{\ldots}{C \vdash \texttt{'a -> unit} \Rightarrow \alpha \rightarrow \{\}, \{\}}\,\text{C.16}}{C \vdash \texttt{m : 'a -> unit} \Rightarrow \mathit{VE}', \{\}}\,\text{(C.9)}}{C, \alpha \vdash \texttt{val m : 'a -> unit} \Rightarrow \mathit{VE}', \{\}}\,\text{(C.7)}}{\dfrac{C \vdash \texttt{'a c with val m : 'a -> unit} \Rightarrow \{\texttt{c} \mapsto (\gamma, \xi\alpha.\mathit{VE}', \text{n})\}, \mathit{VE}'}{C \vdash \texttt{class 'a c with val m : 'a -> unit} \Rightarrow}\,\text{(C.1)}}\,\text{(C.5)}$$

wobei $\mathit{VE}' = \{\texttt{m} \mapsto (\forall.\alpha \rightarrow \{\}, \text{v})\}$

Kapitel 5

Dynamische Semantik

Dieses Kapitel beschreibt die dynamische Semantik von TML. Dazu wird TML auf die in der dynamischen Semantik der Definition von ML [MTHM97] definierte reduzierte Sprache zurückgeführt. Die Transformationen erhalten die Korrektheit eines TML Programms im Sinne der im vorangegangenen Kapitel vorgestellten statischen Semantik.

Die Transformationen der Kern- und Klassensprache orientieren sich an [PJ93]. In [PJ93] werden die Regeln anhand von Beispielen für Haskell beschrieben, wobei der Zusammenhang zwischen statischer und dynamischer Semantik nur informell angesprochen wird. Die hier vorgestellten Regeln stellen den formalen Zusammenhang mit der statischen Semantik für eine Teilsprache von TML her. Dadurch ist es möglich, die für die Übersetzung benötigte Typinformation genau abzulesen.

Bei den Regeln für die Kern- und Klassensprache werden drei Fälle unterschieden: Instanzdeklarationen werden in Methodentabellen übersetzt; bei der Definition eines Bezeichners werden für den Klassenkontext formale Parameter zur Übergabe der Methodentabellen eingeführt; ein Bezeichner in einem Ausdruck wird auf die benötigten Methodentabellen appliziert.

Für das Modulsystem kommen zwei weitere Transformationsregeln hinzu. Erstens müssen beim Abgleichen einer Signatur mit einer Struktur die formalen Parameter angepaßt werden, da zum Beispiel eine Wertspezifikation in der Signatur mehr Kontextelemente aufweisen könnte als der Typ in der Struktur. Zweitens müssen Instanzen umbenannt werden, da diese über Instanztypfunktion und Klassennamen identifiziert werden, aber Instanztypfunktion oder Klassenname beim Abgleich ersetzt werden können. Diese Transformationen werden im zweiten Teil des Kapitels beschrieben.

Zur Definition der Transformationen wird auf die mit Hilfe der statischen Semantik zugeordneten Typinformationen zurückgegriffen. Dies bedeutet zugleich, daß die Trennung zwischen statischer und dynamischer Semantik, wie sie in der Sprachdefinition von ML üblich ist, in TML nicht aufrecht erhalten werden kann.

5.1 Mini TML

Ein TML-Programm ist ein gültiges Mini TML-, kurz MTML-, Programm, wenn es nach Weglassen der Klassendeklarationen, Signaturen und Klassenkontexte in Instanzköpfen ein Wort der Sprache ist, die durch untenstehende Grammatik definiert wird.

Die hier vorgestellten Transformationsregeln steigen analog zur statischen Semantik über den Syntaxbaum rekursiv ab. Es sind nur Programme erlaubt, für die die Korrektheit mittels der statischen Semantik zuvor bewiesen werden kann. Der Korrektheitsbeweis kann als Beweisbaum dargestellt werden. Deswegen kann jedem syntaktischen Konstrukt ein Teilbeweis des Beweisbaumes zugeordnet werden.

Zum Beispiel ist dem Ausdruck `f 1` der Beweisbaum oberhalb von $C \vdash \texttt{f 1} \Rightarrow \tau, TC$ zugeordnet. Damit kann zum Beispiel das Typschema von `f` mit $C(\texttt{f})$ referenziert werden.

5.2 Syntax von MTML

Die hier verwendeten Bezeichnerkategorien entsprechen denen von TML.

strdec ::= ϵ (leer)
| `structure` *strid* = *strexp*
| `instance` *instty longclassid* `with` *methdec* `end`
| `instance` *instty longclassid*
| $strdec_1$ $strdec_2$
| *dec*

dec ::= ϵ (leer)
| `val` *valbind*
| dec_1 dec_2

methdec ::= ϵ (leer)
| `val` *vid* = *exp*
| $methdec_1$ $methdec_2$

valbind ::= *vid* = *exp* ⟨`and` *valbind*⟩
| `rec` *valbind*

exp ::= *longvid*
| *exp exp*
| `fn` *vid* `=>` *exp*
| `let` *dec* `in` *exp* `end`

instty ::= *tyvarseq longtycon*

strexp ::= *strid*
| *strexp* `:` *sigid*
| *strexp* `:>` *sigid*
| `struct` *strdec* `end`

5.3 *strdec*-Transformation

Leere Alternative

$$[\![\epsilon]\!] \equiv \epsilon$$

Struktur

$$[\![\texttt{structure}\ \textit{strid} = \textit{strexp}\ \textit{strdec}]\!] \equiv \texttt{structure}\ \textit{strid} = [\![\textit{strexp}]\!]\ [\![\textit{strdec}]\!]$$

Instanzdeklarationen

Instanzdeklarationen werden in Funktionen zum Erzeugen von Methodentabellen übersetzt. Eine Methodentabelle ist ein Record, dessen Labels den Methodenbezeichnern der zugehörigen Klasse entsprechen. Die Darstellung der Instanzen als Records erfordert die Erweiterung der zulässigen Labels um symbolische Labels. Zum Beispiel wäre `==` kein gültiges Label in Standard ML.

Der Instanzdeklaration entspricht im Beweisbaum der statischen Semantik eine Regelanwendung der Form:

$$\frac{C + IE \vdash \textit{instbind} \Rightarrow IE}{C \vdash \texttt{instance}\ \textit{instbind} \Rightarrow IE \text{ in Env}}$$

Da Instanzen keinen eigenen Bezeichner haben, wird ihnen ein eindeutiger Bezeichner mittels der Instanztypfunktion und dem Klassennamen zugeordnet. Instanztypfunktion und Klassennamen sind im Kontext C definiert. Hat *instty* die Form *tyvarseq tycon*, mit $\textit{tyvarseq} = (\alpha_1, \ldots, \alpha_k)$, dann ist die Instaztypfunktion $\iota = C(\textit{tycon})$ und der Klassenname $\gamma = C(\textit{classid})$. Für $k = 0$ ist die Folge der Typvariablen leer.

Der zugehörige Instanzbezeichner wird wie folgt denotiert:

$$[\iota, \gamma]$$

Instanzbezeichner liegen in einem eigenen Namensraum. Die Funktion [., .] wird im ersten Argument auf Typvariablen erweitert. Somit bezeichnet auch $[\alpha, \gamma]$ einen Instanzbezeichner. [., .] ist injektiv und total. Diese Erweiterung wird später zum Bezeichnen formaler Argumente benötigt.

Für jedes Kontextelement im Kopf der Instanzdeklaration erhält die Transformation ein formales Argument. Die formalen Argumente dienen zur Übergabe der für die dynamische Bindung benötigten Methodentabellen.

Mittels *IE* aus obigem Judgement wird der Kontext der *instty*-Instanz der Klasse *classid* bestimmt: $IE(\iota, \gamma) = (\Gamma_1, \ldots, \Gamma_k)$. Kontexte tauchen hier in der Syntax nicht mehr auf, da sie beim Übergang zu MTML weggelassen werden.

Instanzdeklarationen mit $k \geq 0$

$$
\begin{array}{l}
[\![\texttt{instance}\ \textit{instty longclassid}\ \texttt{with}\ \textit{methdec}\ \texttt{end}]\!] \equiv \\
\quad \texttt{val rec}\ [\iota, \gamma] = \texttt{fn}\ ([\![\alpha_1, \Gamma_1]\!], \ldots, [\![\alpha_n, \Gamma_n]\!]) \texttt{=>} \\
\qquad \texttt{let} \\
\qquad\qquad [\![\textit{methdec}]\!] \\
\qquad \texttt{in} \\
\qquad\qquad \{m_1 \texttt{=} m_1, \ldots, m_n \texttt{=} m_n\} \\
\qquad \texttt{end}
\end{array}
$$

Dabei seien $m_1, \ldots, m_n$ die Methoden der Klasse *classid*.

Ist der Kontext leer, also $k = 0$, so ist die Funktion zum Erzeugen der Methodentabelle über `()` parametrisiert. Dies reflektiert, daß Instanzen rekursiv sind.

Instanzreplikation

Einer Instanzreplikation ist ein Judgement der Form

$$C \vdash \texttt{instance}\ \textit{instty longclassid} \Rightarrow \textit{IE}\ \text{in Env}$$

im Beweisbaum zugeordnet.

Sei $\textit{instty} = \textit{tyvarseq longtycon}$, $\iota = C(\textit{longtycon})$ und $\gamma = C(longclassid)$.

$$[\![\texttt{instance}\ \textit{instty longclassid}]\!] \equiv \texttt{val}\ [\iota, \gamma] = [\iota, \gamma]$$

Da bei 1. Typvariablen allquantifiziert sind, ist die rechte Seite der Wertbindung nicht expansiv. Das heißt sie hat keinen Seiteneffekt, so daß es erlaubt ist, zusätzliche formale Argumente zur Übergabe der Methodentabellen einzuführen.

Beispiel zur Transformation von Instanzen

Für die Beispiele werden die beiden folgenden Klassendeklarationen vorausgesetzt. γ sei der Klassenname der Klasse `c` und ζ der Klassenname der Klasse `d`. In den Beispielen werden auch die Klassenkontexte in den Instanzdeklarationen notiert. Bei dem Übergang zu MTML fallen Klassendeklarationen und -kontext weg.

```
class 'a c with
  val m : 'a -> 'a
end

class 'a d with
  val n : 'a -> 'a
end
```

Zunächst eine Instanzdeklaration ohne Constraints:

```
instance int c with
  val m = fn x => 23
end
```

Die Instanztypfunktion von `int` ist der Typname von `int`, der hier mit t_{int} denotiert wird. γ ist der Klassenname von `c`.

Nach der statischen Semantik ist $IE(t_{int}, \gamma) = ()$, da die Instanzdeklaration keine Kontext besitzt.

```
val rec [t_int, γ] = fn () =>
  let
     val m = fn x => 23
  in
     {m = m}
  end
```

Die nächste Instanzdeklaration definiert eine Instanz für Paare der Klasse `c`. Für die erste Komponente des Paares wird verlangt, daß sie eine Instanz der Klasse `c` ist und für die zweiten Komponente, daß sie eine Instanz der Klasse `d` ist.

Aus dokumentarischen Zwecken wurde im Beispiel der Instanzkontext annotiert, obwohl dieser eigentlich beim Übergang zu MTML wegfällt.

```
instance ('a c, 'b d) => ('a, 'b) pair c with
  val m = fn (Pair (x, y)) => Pair (x, y)
end
```

Nach der statischen Semantik ist $IE(pair, \gamma) = ((\gamma), (\zeta))$, so daß man folgende Übersetzung erhält.

```
val rec [t_pair, γ] =
  fn (([α, γ]), ([β, ζ])) =>
    let
      val m = fn (Pair (x, y)) => x
    in
      {m = m}
    end
```

Sequenz von Strukturdeklarationen

$$[\![strdec_1\ strdec_2]\!] \equiv [\![strdec_1]\!][\![strdec_2]\!]$$

5.4 *dec*-Transformationen

Leere Alternative

$$[\![\epsilon]\!] \equiv \epsilon$$

Wertdeklaration

Einer Wertdeklaration entspricht im Beweisbaum einer Regelanwendung (15):

$$\frac{\begin{array}{cc}\cdots & \cdots \\ C + U \vdash valbind \Rightarrow VE, TC & VE' = Clos^{TC}_{C,valbind} VE\end{array}}{C \vdash \texttt{val}\ tyvarseq\ valbind \Rightarrow VE' \text{ in Env}, TC'}$$

Bei Wertdeklarationen werden zur Übergabe der Methodentabellen zusätzliche formale Argumente, eines für jedes Element des Kontextes im Typschema, eingefügt. Deswegen erfolgt die Transformation von *valbind* unter Einbeziehung der Umgebung VE'.

$$[\![\texttt{val}\ valbind]\!] \equiv \texttt{val}\ [\![valbind]\!]^{VE'}$$

5.5 *valbind*-Transformation

Die statische Semantik ordnet jedem gebundenen Bezeichner ein allgemeinstes Typschema zu. Für jeden Constraint auf den quantifizierten Typvariablen wird die entsprechende Wertbindung um ein zusätzliches formales Argument erweitert. Die formalen Argumente dienen dazu, bei der Applikation eines Bezeichners die benötigten Methodentabellen zur Verfügung zu stellen.

Diese Information kann direkt am Typschema abgelesen werden. Enthält das Typschema keine quantifizierten Variablen oder sind alle Constraints leer, so werden keine Argumente eingeführt.

1. $VE(vid) = (\forall\alpha_1\ \Gamma_1, \ldots, \alpha_n\ \Gamma_n.\tau, is)$, $n \geq 1$

$$[\![vid = exp\ \langle \texttt{and}\ valbind\rangle]\!]^{VE} \equiv \begin{array}{l}\texttt{val}\ vid = \texttt{fn}\ (\texttt{[}\alpha_1, \Gamma_1\texttt{]}, \ldots, \texttt{[}\alpha_1, \Gamma_n\texttt{]})\ \texttt{=>}\ [\![exp]\!] \\ \langle \texttt{and}\ [\![valbind]\!]^{VE}\rangle\end{array}$$

2. $VE(vid) = \tau$

$$[\![vid = exp\ \langle \texttt{and}\ valbind\rangle]\!]^{VE} \equiv\ \texttt{val}\ vid\ \texttt{=>}\ [\![exp]\!]\langle \texttt{and}\ [\![valbind]\!]^{VE}\rangle$$

Beispiel zur Transformation von Wertbindungen

Die Funktion `f` hat den Typ $\forall\alpha\ (\gamma).\alpha \to \alpha$.

$$[\![\texttt{val f = fn x => m x}]\!]^{\{\texttt{f}\mapsto(\forall\alpha\ (\gamma).\alpha\to\alpha,\mathrm{v})\}} \equiv \texttt{val f = fn}\ [\alpha, \gamma]\ \texttt{=>}\ [\![\texttt{fn x => m x}]\!]$$

5.6 *methdec*-Transformationen

Leere Alternative

$$[\![\ ]\!] \equiv$$

Methodenbindung

In Regel (C.10) der statischen Semantik wird erzwungen, daß die Methodendeklarationen den Typ haben, der in der Klassendeklaration spezifiziert wurde, nachdem in der Spezifikation die Klassentypvariable durch den Instanztyp substituiert wurde. Deshalb muß kein Programmcode eingefügt werden, der die Methoden an die Klassenspezifikation anpaßt.

Bei einer Methode wird für jedes Kontextelement des in der Klassendeklaration spezifizierten Typschema ein formales Argument eingefügt. Lediglich für den Constraint über der Klassentypvariable wird kein formales Argument eingeführt.

Bei der Elaboration des Rumpfes einer Instanzdeklaration in der statischen Semantik, Regel (C.10), wird die Wertumgebung der Klasse C hinzugefügt. Dabei wird über die Klassentypvariable abgeschlossen, so daß diese nicht frei auftritt.

In Abhängigkeit vom Typschema und den Klassenkonstraints erhält man folgende Transformationsregeln:

1. $C(vid) = (\forall\alpha\ (\gamma), \alpha_1\ \Gamma_1, \ldots, \alpha_n\ \Gamma_n.\tau, is)$, $n \geq 1$, α Klassentypvariable:

$$[\![\texttt{val}\ vid = exp]\!] \equiv \texttt{val}\ vid = \texttt{fn}\ ([\![\alpha_1, \Gamma_1]\!], \ldots, [\![\alpha_n, \Gamma_n]\!])\ \texttt{=>}[\![exp]\!]$$

2. $C(vid) = \forall\alpha\ (\gamma).\tau$, α Klassentypvariable:

$$[\![\texttt{val}\ vid = exp]\!] \equiv \texttt{val}\ vid = [\![exp]\!]$$

5.7 *exp*-Transformationen

Die Transformation eines Ausdrucks *exp* hängt von dem Typ τ ab, den die statische Semantik im Beweis durch das Judgement $C \vdash exp \Rightarrow \tau$ dem Ausdruck zuordnet.

Bezeichner

Bei Bezeichnern muß unterschieden werden, ob der Bezeichner ein Methodenbezeichner ist oder nicht.

Transformationsregel für normale Bezeichner

Ein Bezeichner *longvid* wird durch die statische Semantik an ein Typschema

$$C(longvid) = (\forall\alpha_1\ \Gamma_1, \ldots, \alpha_n\ \Gamma_n.\tau', is)$$

gebunden. Wie zuvor seien die Typvariablen durch die depth-first, left-most Ordnung geordnet. Da der Ausdruck den Typ τ hat, muß eine Substitution $\psi = \{\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n\}$ existieren, so daß $\tau = \tau'\psi$ ist.

Mit Hilfe dieser Substitution kann man die benötigten Methodentabellen aus den Instanzdeklarationen erzeugen und den Wertbezeichner damit versorgen.

Dabei muß wieder unterschieden werden, ob überhaupt Constraints vorhanden sind order nicht.

1. $n = 0$, das heißt, es wird über keine Typvariable quantifiziert:

$$[\![longvid]\!] \equiv longvid$$

2. $n \geq 1$

$$[\![longvid]\!] \equiv (\ longvid\ ([\![\tau_1, \Gamma_1]\!]^{IE}\ \text{of}\ ^{C}, \ldots, [\![\tau_n, \Gamma_n]\!]^{IE}\ \text{of}\ ^{C})\ )$$

Transformationsregel für Methodenbezeichner

Bei Methodenbezeichner wird zunächst die Methode aus einer Methodentabelle selektiert und dann die Methode mit den Methodentabellen für die Kontextelemente auf den gebundenen Typvariablen im Typschema der Methode versorgt.

Da sich das Konstrukt nicht innerhalb einer Klassendeklaration befindet, ist die Klassentypvariable im Typschema gebunden. Das einzige Kontextelement auf der Klassentypvariable bezieht sich auf die Klasse, durch die die Methode eingeführt wurde.

Sei $C(longvid) = (\forall\alpha\ (\gamma), \alpha_1\ \Gamma_1, \ldots, \alpha_n\ \Gamma_n.\tau', is)$ und α die Klassentypvariable. Dann ist γ der Klassename der zugehörigen Klasse. Da der Ausdruck den Typ τ hat, muß eine Substitution $\psi = \{\tau''/\alpha, \tau_1/\alpha_1, \ldots, \tau_n/\alpha_n\}$ existieren, so daß $\tau = \tau'\psi$ ist.

Sei *m* das Methodenlabel. Dieses kann syntaktisch aus *longvid* extrahiert werden.

Mit Hilfe der Substitution für die Klassentypvariable α und dem Klassennamen γ wird wird die Methodentabelle erzeugt, aus der die Methode *m* zu selektieren ist.

1. $n = 0$:
$$[\![longvid]\!] \equiv (\#m\ [\![\tau'', \gamma]\!]^{IE}\ \text{of}\ ^{C})$$

2. $n \geq 1$:
$$[\![longvid]\!] \equiv ((\#m\ [\![\tau'', \gamma]\!]^{IE}\ \text{of}\ ^{C})([\![\tau_1, \Gamma_1]\!]^{IE}\ \text{of}\ ^{C}, \ldots, [\![\tau_n, \Gamma_n]\!]^{IE}\ \text{of}\ ^{C}))$$

Bemerkung: Die Klassentypvariable kann nicht mit Hilfe der statischen Semantik herausgefunden werden, da diese nicht zwischen Methoden- und normalen Wertbezeichnern unterscheidet. Da die Unterscheidung aber notwendig ist, wird in der echten Implementierung zwischen normalen Wertbezeichnern und Methoden unterschieden.

Applikation

$$[\![exp_1\ exp_2]\!] \equiv [\![exp_1]\!]\ [\![exp_2]\!]$$

Abstraktion

$$[\![\texttt{fn}\ vid\ \texttt{=>}\ exp]\!] \equiv \texttt{fn}\ vid\ \texttt{=>}\ [\![exp]\!]$$

`let`-Ausdruck

$$[\![\texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end}]\!] \equiv \texttt{let}\ [\![dec]\!]\ \texttt{in}\ [\![exp]\!]\ \texttt{end}$$

5.8 Erzeugen von Methodentabellen

Bei Typvariablen ist die Methodentabelle an den Instanzbezeichner $[\alpha, \gamma]$ gebunden.

$$[\![\alpha, \gamma]\!]^{IE} \equiv [\alpha, \gamma]$$

Für einen konstruierten Typ wird die Methodentabelle mit Hilfe der folgenden Transformationsregeln erzeugt:

$$[\![\tau, \Gamma]\!]^{IE} \equiv \texttt{(}[\![\tau, \gamma_1]\!]^{IE}\texttt{, } \ldots \texttt{, } [\![\tau, \gamma_n]\!]^{IE}\texttt{)}$$

wobei $(\gamma_1, \ldots, \gamma_n)$ die geordneten Constraints in Γ.

$$[\![(\tau_1, \ldots, \tau_n)t, \gamma]\!]^{IE} \equiv \texttt{ ([}t, \gamma\texttt{] (}[\![\tau_1, \Gamma_1]\!]^{IE}\texttt{, } \ldots \texttt{, } [\![\tau_n, \Gamma_n]\!]^{IE}\texttt{))}$$

wobei $IE(t, \gamma) = (\Gamma_1, \ldots, \Gamma_n)$.

Ein Beispiel zum Erzeugen von Methodentabellen

Dieses Beispiel zeigt, wie eine Instanz für den Typ `(int, 'a) pair` der Klasse `c` erzeugt wird. t_{int} bzw. t_{pair} denotieren dabei die jeweiligen Typnamen. Das Beispiel verwendet die *IE*-Umgebung.

$$IE = \{(t_{int}, \gamma) \mapsto (), (t_{pair}, \gamma) \mapsto ((\gamma), (\zeta))\}$$

$$[\![(t_{int}, \alpha)t_{pair}, \gamma]\!] \equiv \texttt{([}t_{pair}, \gamma\texttt{] (([}t_{int}, \gamma\texttt{]),([}\alpha, \zeta\texttt{]))}$$

5.9 *strexp*-Transformation

Einem Strukturausdruck ist im Beweisbaum ein Judgement der Form:

$$B \vdash \mathit{strexp} \Rightarrow E$$

Strukturbezeichner

$$[\![\mathit{strid}]\!] \equiv \mathit{strid}$$

Transparenter Signaturabgleich

Eine Struktur `S` ist eine gültige Implementierung einer Signatur `G`, wenn die hergeleitete Signatur der Struktur polymorpher ist als `G`. Das heißt insbesondere, daß der Typ eines Wertes in der Struktur eine unterschiedliche Anzahl von Klassenconstraints tragen kann, als der in der Signatur spezifizierte Typ.

Da bei der Bindung von Wertbezeichnern ein formales Argument für jeden Klassenconstraint hinzugefügt wird, bedeutet dies, daß der in der Struktur definierte Wert andere formale Argumente besitzt, als der durch die Signatur spezifiziert Wert erwarten würde.

In folgendem Beispiel erwartet der in der Struktur spezifizierte Wert eine Methodentabelle der Klasse `EQ`. Durch den Signaturabgleich wird aber die Struktur spezialisiert, so daß die Methodentabelle für ganze Zahlen der Klasse `EQ` übergeben werden muß.

```
structure S : sig val f : int -> bool end =
  fun f x = x == x
end
```

Folgende Transformationsregeln passen beim transparenten Signaturabgleich die formalen Argumente zur Übergabe der Methodentabellen an.

Nach statischer Semantik, Regel (52) gilt:

$$\frac{B \vdash \mathit{strexp} \Rightarrow E \quad B \vdash \mathit{sigexp} \Rightarrow \Sigma \quad \Sigma \geq E' \quad \mathit{IE} \text{ of } B \vdash E' \prec E}{B \vdash \mathit{strexp} : \mathit{sigexp} \Rightarrow E'}$$

Mit Hilfe der statischen Information erhält man folgende Transformationsregel:

$$\begin{array}{l}[\![\mathit{strexp} : \mathit{sigid}]\!] \equiv \\ \quad \texttt{let} \\ \quad\quad \texttt{structure S} = [\![\mathit{strexp}]\!] \\ \quad \texttt{in} \\ \quad\quad [\![E : E']\!]^{\texttt{S}}_{\mathit{IE}} \text{ of } B \\ \quad \texttt{end}\end{array}$$

Opaken Signaturabgleich

Beim opaken Signaturabgleich sind zwei Fälle speziell zu behandeln:

1. Bei Wertfeldern müssen genau wie beim transparenten Signaturabgleich die formalen Argumente zur Übergabe der Methodentabellen angepaßt werden.
2. Da beim opaken Signaturabgleich neue Typnamen und Klassennamen eingeführt werden können, erhalten die Instanzbezeichner neue Namen, die von der Realisierung abhängen.

In folgedem Beispiel wird durch den opaken Abgleich ein neuer Typname für `t` eingeführt. Da der spezifizierte Instanztyp `t` ist und somit die Instanz durch den Typnamen von `t` adressiert wird, muß die Instanz unter dem neuen Typnamen zur Verfügung gestellt werden. Intern wird die Instanz mittels des Instanztyps `int` adressiert.

```
structure S :> sig
                 type t
                 instance t EQ
               end =
  struct
     type t = int
     instance int EQ with ...
  end
```

Bei einem opaken Signaturabgleich kommt die Regel (53) zur Anwendung:

$$\frac{\begin{array}{ll} B \vdash \mathit{strexp} \Rightarrow E & B \vdash \mathit{sigexp} \Rightarrow (T', K')E' \\ (T', K')E' \geq E'' & IE \text{ of } B \vdash E'' \prec E \\ T' \cap (T \text{ of } B) = \emptyset & K' \cap (K \text{ of } B) = \emptyset \end{array}}{B \vdash \mathit{strexp} \texttt{ :> } \mathit{sigexp} \Rightarrow E'}$$

T' und K' sind dabei neue Typnamen bzw. Klassennamen. Die bei der Instantiierung

$$(T', K')E' \geq E''$$

verwendete Realisierung ψ wird zum Umbenennen der Instanzbezeichner benötigt. Die Instanzbezeichner müssen umbenannt werden, da das Resultat der Regel E' ist.

$$\begin{array}{l} [\![\mathit{strexp} \texttt{ : } \mathit{sigid}]\!] \equiv \\ \quad \texttt{let} \\ \quad\quad \texttt{structure S =} [\![\mathit{strexp}]\!] \\ \quad \texttt{in} \\ \quad\quad [\![E :> E']\!]^{\psi,\texttt{S}}_{IE \text{ of } B} \\ \quad \texttt{end} \end{array}$$

5.10 $[\![E : E']\!]$, $[\![E :> E']\!]$-Transformation

Mit $E = (T, K, VE, TE, SE, CE, IE)$ und $E' = (T', K', VE', TE', SE', CE', IE')$ sei

$$[\![E : E']\!]^{longid}_{IE} \equiv [\![VE : VE']\!]^{longid}_{IE} [\![SE : SE']\!]^{longid}_{IE} [\![IE : IE']\!]^{longid}$$

$$[\![E :> E']\!]^{\psi,longid}_{IE} \equiv [\![VE : VE']\!]^{longid}_{IE} [\![SE :> SE']\!]^{\psi,longid}_{IE} [\![IE :> IE']\!]^{\psi,longid}$$

5.11 $[\![VE : VE']\!]$-Transformationen

Sei $VE' = \{vid_1 \mapsto (\sigma'_1, \mathrm{is}_1), \ldots, vid_r \mapsto (\sigma'_r, \mathrm{is}_r)\}$ und $\sigma_i = VE(vid_i)$, für alle $i \in \{1, \ldots, r\}$.

Die Transformationsregel für die *VE*-Umgebung:

$$[\![VE : VE']\!]^{longid} \equiv \begin{array}{l} \texttt{val } vid_1 \texttt{ =} [\![vid_1 : (\sigma_1, \sigma'_1)]\!]^{longid} \\ \ldots \\ \texttt{val } vid_r \texttt{ =} [\![vid_r : (\sigma_r, \sigma'_r)]\!]^{longid)} \end{array}$$

5.12 $[\![vid : (\sigma, \sigma')]\!]$-Transformation

Bei Wertfeldern wird Programmcode zum Anpassen der formalen Argumente, die zur Übergabe der Methodentabellen dienen, eingefügt. Die Konvertierung kann mittels einer Substitution ermittelt werden, die den Zusammenhang zwischen spezifiziertem Typ und dem Typ in der anzupassenden Struktur, herstellt.

Sind $\sigma' = \forall\alpha_1\ \Gamma_1, \ldots, \alpha_n\ \Gamma_n.\tau'$ und $\sigma = \forall\beta_1\ \Pi_1, \ldots, \beta_m\ \Pi_m.\tau$, dann existiert eine Substitution $\varphi = \{\tau_1/\beta_1, \ldots, \tau_m/\beta_m\}$, so daß $\tau' = \tau\varphi$. Eine solche Substitution existiert, weil die statischen Semantik zusichert, daß die Struktur die Signatur erfüllt.

Im folgenden Beispiel ist für `f` die Substitution $\varphi = \{t_{int}/\alpha\}$.

```
signature G =
  val f : int -> int
end

structure S : G =
  val f : 'a -> 'a
end
```

Abkürzend wird $[\![\tau_i, \Gamma_i]\!]^m$ für $[\![\tau_1, \Gamma_1]\!]$, ..., $[\![\tau_m, \Gamma_m]\!]$ geschrieben.

1. $n, m \geq 1$

   `val` *vid* `=` `fn (` $[\![\alpha_i, \Gamma_i]\!]^n$ `) => (` *longid* `.` *vid* `(` $[\![\tau_i, \Pi_i]\!]^m$ `))`

2. $n = 0, m \geq 1$

   `val` *vid* `= (` *longid* `.` *vid* `(` $[\![\tau_i, \Pi_i]\!]^m$ `))`

3. $n = m = 0$

   `val` *vid* `=` *longid* `.` *vid*

5.13 $[\![IE : IE']\!]$, $[\![IE :> IE']\!]$-Transformation

Sei $IE' = \{(\iota_1, \gamma_1) \mapsto \text{cons}_1, \ldots, (\iota_r, \gamma_r) \mapsto \text{cons}_r\}$.

$$[\![IE : IE']\!]^{longid} \equiv \begin{array}{l} \texttt{val}\ [\iota_1, \gamma_1] = longid\texttt{.}[\iota_1, \gamma_1] \\ \ldots \\ \texttt{val}\ [\iota_r, \gamma_r] = longid\texttt{.}[\iota_r, \gamma_r] \end{array}$$

Beim opaken Ableich ist die resultierende Umgebung, die mit neuen Typnamen und Klassennamen instantiierte Signaturumgebung (Regel (53) der statischen Semantik). Da dabei auch Instanzbezeichner umbenannt werden können, müssen die Instanzbezeichner angepaßt werden.

Dazu wird die Umkehrabbildung der Realisierung ψ benötigt. Streng genommen existiert diese aber nicht, weil Typnamen durch beliebige Typfunktionen substituiert werden können. Da als Instanztypfunktionen nur Typnamen zugelassen sind, kann die Umkehrabbildung aber für Instanztypfunktionen gebildet werden. ψ^{-1} wird als die partielle Umkehrabbildung aufgefaßt, die nur über Intanztypfunktionen und Klassennamen definiert ist.

$$[\![IE :> IE']\!]^{\psi,longid} \equiv \begin{array}{l} \texttt{val}\,[\psi^{-1}(\iota_1), \psi^{-1}(\gamma_1)] = longid\,.\,[\iota_1, \gamma_1] \\ \ldots \\ \texttt{val}\,[\psi^{-1}(\iota_r), \psi^{-1}(\gamma_r)] = longid\,.\,[\iota_r, \gamma_r] \end{array}$$

In MTML sind nur die Typnamen als Instanztypfunktion erlaubt. Die Erweiterung auf alle Instanztypfunktionen ist einfach, wie man sich leicht überlegt. Gerechtfertigt wird diese Behauptung auch durch die Erkenntnis, daß man die zusätzlichen Instanztypfunktionen injektiv auf Typnamen abbilden kann.

5.14 $[\![SE : SE']\!]$, $[\![SE :> SE']\!]$-Transformation

Bei Strukturen wird rekursiv abgestiegen.
Sei $SE' = \{strid_1 \mapsto E'_1, \ldots, strid_n \mapsto E'_n\}$ und $E_i = SE(strid_i)$.

$$[\![SE : SE']\!]^{(\psi,\texttt{LONDID})}_{IE} \equiv \begin{array}{l} strid_1 = \texttt{struct}\,[\![E_1 : E'_1]\!]^{(\psi,\texttt{LONDID}.strid_1)}_{IE}\,\texttt{end} \\ \ldots \\ strid_n = \texttt{struct}\,[\![E_n : E'_n]\!]^{(\psi,\texttt{LONDID}.strid_n)}_{IE}\,\texttt{end} \end{array}$$

$$[\![SE :> SE']\!]^{(\psi,\psi',\texttt{LONDID})}_{IE} \equiv \begin{array}{l} strid_1 = \texttt{struct}\,[\![E_1 :> E'_1]\!]^{(\psi,\psi',\texttt{LONDID}.strid_1)}_{IE}\,\texttt{end} \\ \ldots \\ strid_n = \texttt{struct}\,[\![E_n :> E'_n]\!]^{(\psi,\psi',\texttt{LONDID}.strid_n)}_{IE}\,\texttt{end} \end{array}$$

5.15 Beispiel

Das Beispiel wird unter folgender Instanzumgebung transformiert:

$$IE = \{(t_{int}, \gamma) \mapsto (), (t_{pair}, \gamma) \mapsto ((\gamma), (\zeta))\}$$

$$\begin{array}{ll} & [\![\texttt{val f = fn x => m (pair (1,x))}]\!] \\ \equiv & \texttt{val f = fn (([}\alpha,\zeta\texttt{])) => }[\![\texttt{fn x => m (pair (1,x))}]\!] \quad (1) \\ \equiv & \texttt{val f = fn (([}\alpha,\zeta\texttt{])) => fn x => }[\![\texttt{m (pair (1,x))}]\!] \quad (2) \\ \equiv & \texttt{val f = fn (([}\alpha,\zeta\texttt{])) => fn x => }[\![\texttt{m}]\!]\,[\![\texttt{(pair (1,x))}]\!] \quad (3) \\ \equiv & \texttt{val f = fn (([}\alpha,\zeta\texttt{])) => fn x => (\#m }[\![(t_{int},\alpha)t_{pair},\gamma]\!]\texttt{)}\,[\![\texttt{(pair (1,x))}]\!] \quad (4) \\ \equiv^{+} & \texttt{val f = fn (([}\alpha,\zeta\texttt{])) => fn x =>} \\ & \texttt{(\#m ([}t_{pair},\gamma\texttt{] (([}t_{int},\gamma\texttt{]),((([}\alpha,\zeta\texttt{])))))) (pair (1,x))} \quad (5) \end{array}$$

In Zeile (3) hat m den Typ $(t_{int}, \alpha)t_{pair} \rightarrow (t_{int}, \alpha)t_{pair}$. Das heißt das Typschema von m

$$\forall\beta\ (\gamma).\beta \rightarrow \beta$$

wurde mittels der Substitution $\xi = \{\beta/(t_{int}, \alpha)t_{pair}\}$ instantiiert. β ist dabei die Klassentypvariable, so daß sich die obige Transformation ergibt.

5.16 Funktoren

In diesem Kapitel wurde die Transformation von Funktoren nicht beschrieben. Die Transformation von Funktoren stellt letztendlich eine Kombination aus den Transformationen für transparenten und opaken Signaturabgleich dar. Dabei ist zu beachten, daß auf die Argumentstruktur mittels der Signatur zugegriffen wird. Somit sind die Typnamen und Klassennamen im Präfix aus der Sicht des Funktorrumpfes quasi abstrakt, so daß bei der Applikation die Argumentstruktur ähnlich wie bei einem opaken Abgleich angepaßt werden muß.

Kapitel 6

Implementierung

Im Rahmen der Diplomarbeit ist eine prototypische Implementierung entstanden, die gezeigt hat, wie die in den vorherigen Kapiteln beschriebenen Erweiterungen umgesetzt werden können.

6.1 Phasen der Übersetzung

Die Übersetzung eines TML-Programms wird in drei Phasen vorgenommen. Die erste Phase umfaßt die Syntaxanalyse und die Auflösung von Infix-Operatoren.

In der zweiten Phase erfolgt die Typrekonstruktion nach dem Hindley/Milner Algorithmus. Für Typklassen wurden die Unifikation und das Instantiieren von Typvariablen, wie in [PJ93] beschrieben, modifiziert. Im Gegensatz zu dem Vorgehen in [PJ93] werden während der Typinferenz die Typklassen noch nicht auf ML ohne Typklassen zurückgeführt. Wesentlicher Grund für die Aufspaltung der Typinferenz und der Transformation in zwei Phasen war die Erkenntnis, daß die Zusammenfassung in eine Phase zu sehr schwer verständlichem Programmcode führt.

In der dritten Phase wird die Zwischensprache in die Zielsprache übersetzt. Dabei wird mit Hilfe der Typinformation die Typklassenelemente in ML ohne Typklassen übersetzt. Zielsprache ist die `INTERMEDIATE_GRAMMAR` des Alice-Projektes [KR00].

6.2 Komponenten des Übersetzers

Der Übersetzer besteht aus einem Modul zur Syntaxanalyse, einem Elaborationsmodul und einem Übersetzungsmodul.

6.3 Syntaxanalyse

Für die Syntaxanalyse wurde der Parser der Hamlet-Implementierung [Ros99] verwendet. Die meisten Syntaxerweiterungen konnten einfach hinzugefügt werden. Lediglich beim Parsen der Klassenkontexte mußte ein kleiner Trick angewendet werden, da die Syntax an dieser Stelle nicht mit einem Look-ahead von 1 auskommt.

Genauer kann der Parser in folgendem Beispiel bei der Analyse des Kontextelementes nicht entscheiden, ob er bereits mit der Analyse des Instanztyps fertig ist oder nicht, so daß an dieser Stelle ein Shift/Reduce-Konflikt ensteht.

```
val m : int . list EQ => 'a -> unit
```

Das Problem kann leicht umgangen werden, da Instanztypen und Klassenbezeichner zusammen genommen den gleichen syntaktisch Aufbau aufweisen wie ein Typ. Nach dem Parsen wird dann der Instanzbezeichner aus dem Typ extrahiert.

6.4 Typinferenz

Die Typinferenz funktioniert nach dem Prinzip des Hindley/Milner Algorithmus [Mil78] und ist für eine Teilsprache von ML in [Car87] ausführlich beschrieben. Da die Typinferenz sehr gut verstanden ist, wird hier nur auf die wesentlichen Aspekte der Implementierung eingegangen.

6.4.1 Repräsentation von Typen

Folgende Abbildung zeigt die idealisierte Typrepresentation im TML-Typinferenzmodul:

Abbildung 6.1 Repräsentation der Typen

```
type context = classname list

datatype tyterm =
         TRef of ty
       | TVar of {level : int,
                  explicit : bool,
                  constraints : constraints}
       | TCon of ty list * tyname

withtype ty = tyterm ref
```

Funktionstypen werden einfach als zweistellige konstruierte Typen dargestellt. Das heißt, es gibt einen speziellen Typnamen, der zum Konstruieren von Funktionstypen dient.

6.4.2 Repräsentation von Typvariablen

Eine Typvariable trägt einen sogenannten *Level*, ein `explicit`-Flag und ihren Klassenkontext.

Level einer Typvariable

Anhand des Levels einer Typvariable wird bestimmt, ob diese bei einer Wertbindung generalisiert werden soll oder nicht.

Beim Generalisieren eines Typs, wird eine Typvariable generalisiert, wenn sie nicht im Typ in einer Variable vorkommt, die durch eine umschließende Abstraktion gebunden ist [Car87]. In [Car87] werden dazu alle Typannahmen in der Umgebung traversiert.

Für Programmiersprachen wie Standard ML ist dies nicht praktikabel, da unter Umständen sehr viele Typen zu untersuchen sind. Deswegen kann folgender Trick angewendet werden: Jeder Typvariable wird beim Erzeugen ein Level zugewiesen. Der Level ergibt sich durch die Abstiegstiefe auf dem Syntaxbaum. Er erhöht sich um eins, wenn über ein Konstrukt abgestiegen wird, an dem beim Verlassen die Typen von Bezeichnern generalisiert werden. In ML erhöht sich der Level zum Beispiel um 1 beim Abstieg ins *valbind* in Regel (15) der statischen Semantik [MTHM97] und wird um 1 dekrementiert beim Verlassen. Bevor der Level dekrementiert wird, werden alle Typvariablen in den Typen von Bezeichnern generalisiert, deren Level gleich dem aktuellen Level ist.

In folgendem Beispiel sind die Level als Indizes annotiert.

```
val1 f = fn1 x =>
   let1
     val2 rec g = fn2 y = y
   in1
     g x
   end
```

In dem Beispiel wird `g` beim Abstieg zunächst der Typ α_2 und `y` der Typ β_2 zugewiesen. Durch die Unifikation wird β_2 mit $\alpha_2 \rightarrow \alpha_2$ instantiiert. Beim Verlassen der Wertbindung wird der Level zu 1 decrementiert und dabei alle Typvariablen mit Level 2 generalisiert, so daß man für `g` den Typ

$$\forall\beta.\beta \rightarrow \beta$$

erhält.

Wird eine Typvariable α_i mit einem Typ τ unifiziert, so muß der Level aller Typvariablen im Typ τ, die einen Level größer als i besitzen, auf i gesetzt werden.

Dies führt dazu, daß in folgendem Beispiel nicht alle Typvariablen im Typ von `g` generalisiert werden. Der Grund liegt darin, daß die Variable `x` im Rumpf von `g` verwendet wird und somit der Typ von `g` vom Typ von `x` abhängt. Oder anders ausgedrückt: Der Typ von `g` wird sozusagen durch den Typ von `x` mitbestimmt.

```
val f = fn x =>
  let
     val g = fn h => h x
  in
     g
  end
```

$$\mathtt{g} : \forall\beta.(\alpha \to \beta) \to \beta$$

Typvariablen werden nicht generalisiert, wenn sie im Typ eines *expansiven* Ausdrucks vorkommen, im Sinne von [MTHM97], Abschnitt 4.7. Bei solchen Typvariablen wird der Level um 1 decrementiert, so daß sie gegebenenfalls beim Generalisieren an einer umschließenden Wertbindung generalisiert werden können.

Klassenkontext auf Typvariablen

Ein Kontext wird als Liste von Klassennamen repräsentiert. Durch die Liste ist ein Kontext kanonisch geordnet. Diese Ordnung ist für die Übersetzung wichtig, da die Reihenfolge der formalen Argumente für die Methodentabelle davon abhängt. Siehe dazu auch die dynamische Semantik in Kapitel 5.

`explicit`-Flag

Das `explicit`-Flag kennzeichnet explizite Typvariablen. Explizite Typvariablen sind alle Typvariablen, die im Programmtext auftauchen. Zum Beispiel bei expliziten Annotationen an Wertbindungen oder bei der Spezifikation von Signaturen. Diese dürfen nicht instantiiert werden und auch keine weiteren Kontextelemente aufnehmen als die spezifizierten.

6.4.3 Typschemata

Für Typschemata wurde keine eigene Repräsentation eingeführt. Beim Generalisieren wird der Level jeder generalisierten Typvariable auf -1 gesetzt, so daß man die in einem Typschema quantifizierten Typvariablen anhand des Levels erkennt. Zum Beispiel wird $\forall\alpha.\alpha$ als α_{-1} repräsentiert.

6.4.4 Bezeichnerstatus für Methoden

Die Implementierung verwendet einen zusätzlichen Bezeichnerstatus für Methoden. Der Status für Methoden wird benötigt, da die Übersetzung von Methodenbezeichnern sich von der von anderen Wertbezeichner unterscheidet, wie in 5.6 beschrieben.

6.4.5 Unifikationsalgorithmus

Der Unfikationsalgorithmus ändert sich an zwei Stellen:

- Beim Unifizieren zweier Typvariablen werden die Klassenconstraints vereinigt.
- Beim Unifizieren einer Typvariable mit einem konkreten Typ, werden die Klassenconstraints auf den Typvariablen durch die sogenannte *Kontextreduktion* unter Berücksichtigung der Instanzumgebung *IE* über den Typ propagiert.

Der hier angegebene Ausschnitt des Unifikationsalgorithmus berücksichtigt nicht den Level der Typvariable und auch nicht das Flag, das eine explizite Typvariable markiert.

```
fun unify IE (var as ref (TVAR {constraints=c, ...}),
              var' as ref (TVAR {constraints=c', ...})) =
    (var := TVAR {constraits=(c @ c'), ...};
     var':= TREF var)
  | unify IE (var as ref (TVAR {constraints=c, ...}), tau) =
    (propagateConstraits IE (c, tau);
    var := TREF tau)
  | ...
```

Beim propagieren der Kontexte sind zwei Fälle zu unterscheiden:

- Bei einer Typvariablen werden die propagierten Klassenconstraints zu den Constraints der Typvariable hinzugenommen.
- Bei einem Typ, wird zunächst die Typfunktion extrahiert und anschließend für jedes Element des zu propagierenden Constraints der Instanzkontext nachgeschlagen. Das Nachschlagen schlägt fehl, wenn keine passende Instanz im Skopus liegt. War das Nachschlagen erfolgreich werden die nachgeschlagenen Constraints auf die Argumente des konstruierten Typs propagiert.

```
fun propContext IE
       (c, var as ref (TVar {constraints=c', ...})) =
    var := TVAR {constraint=(c @ c'), ...}

  | propContext IE (c, TREF tau) =
    propContext IE (c, tau)
  | propContext IE (c, TCON (tys, tyname)) =
    let
        val instanceConstraintsList =
            List.map (fn classname =>
                      IE.lookup (tyname, classname)) c
            (* IE.lookup kann Exception werfen, wenn *)
            (* keine passende Instanz gefunden wird. *)
    in
```

```
          List.app
            (propagateContext IE)
            (ListPair.zip (instanceConstraintList, tys))
      end
```

6.4.6 Typannotationen am Syntaxbaum

Die in Kapitel 5 verwendete Typinformation wird während der Typinferenz am Syntaxbaum annotiert.

Während der Übersetzung werden die folgenden Annotationen zum Auflösen der Typklassenkonstrukte benötigt:

Wertbezeichner in Ausdrücken Bei der Elaboration von Ausdrücken wird das Typschema des Wertbezeichners mit neuen Typvariablen instantiiert. Die neuen Typvariablen werden gepaart mit den Constraints in depth-first, left-most Ordnung bezüglich des instantiierten Typschemas an den Syntaxbaum annotiert. Durch die Unifikation werden die Typvariablen instantiiert.

Bei der späteren Übersetzung kann dann leicht abgelesen werden, für welche Typen entsprechende Methodentabellen zu erzeugen sind. Die Ordnung der Typvariablen ist wichtig, damit die Methodentabellen in der richtigen Reihenfolge übergeben werden können.

Ist der Wertbezeichner eine Methode, so wird die Klassentypvariable gesondert am Syntaxbaum annotiert. Bei der Übersetzung wird aus der für die Klassentypvariable erzeugten Methodentabelle die Methode selektiert.

Wertbindungen An Wertbindungen werden die generalisierten Typvariablen an den Syntaxbaum in depth-first, left-most Ordnung annotiert. Diese Information wird beim Übersetzen zum Einfügen der formalen Argumente für die Übergabe der Methodentabellen benötigt.

Transparenter Signaturabgleich Beim transparenten Signaturabgleich werden Annotationen zum Anpassen der Wertfelder an den Syntaxbaum annotiert. Dies geschieht rekursiv für jedes Wertfeld. Die Annotation wird aus dem in der Signatur spezifizierten und dem tatsächlichen Typ der Struktur wie folgt berechnet: Zunächst wird der Typ in der Struktur neu instantiierrt und dann das Paar aus neuen Typvariablen und zugehörigen Constraints in der Annotation aufgenommen. Die Annotation berücksichtigt dabei die depth-first, left-most Ordnung bezüglich des Typs in der Struktur. Nun wird der neu instantiierte Typ mit dem in der Signatur spezifizierten Typ instantiiert. Dies ist möglich, da die Signatur spezieller ist, als die Struktur. Durch die Unifikation werden die Typvariablen in der Annotation spezialisiert, so daß bei der Übersetzung mit Hilfe der instantiierten Typvariablen die benötigten Methodentabellen erzeugt werden können.

Opaker Signaturabgleich Beim opaken Signaturabgleich werden zunächst für die Wertbezeichner die gleichen Annotationen vorgenommen, wie beim transparenten Signaturabgleich. Anschließend wird für jede Instanz ein Paar, das aus dem Bezeichner der Instanz in der Struktur und dem neuen Bezeichner besteht, annotiert. Die Berechnung der neuen Bezeichner erfolgt dabei wie Kapitel 5 beschrieben. Auch das Annotieren der Instanzbezeichner erfolgt rekursiv über die Struktur.

6.5 Übersetzung von TML

Im Übersetzungsmodul wurden exakt die in Kapitel 5 formalisierten Transformationsregeln umgesetzt. Die für die Transformation der Übersetzung benötigte Typinformation wird während der Typinferenz an den Syntaxbaum der Zwischendarstellung annotiert.

Im folgenden wird nur auf Punkte eingegangen, die in keinem der beiden vorausgehenden Kapiteln behandelt wurden.

6.5.1 Formale Argumente zum Übergeben der Methodentabellen

Die dynamische Semantik in Kapitel 5 führt bei Wertbindungen und Instanzdeklarationen formale Argumente zur Übergabe der Methodentabellen ein. Bei der Implementierung ist zu beachten, daß die Ordnung berücksichtigt wird.

6.5.2 Repräsentation der Methodentabellen

Der Prototyp repräsentiert die Methodentabellen als Records. Die Methoden werden an die gleichnamigen Labels gebunden. Dabei wurde Ausgenutzt, daß das Backend symbolische Bezeichner unterstützt. Die Unterstützung symbolischer Bezeichner ist nicht zwingend für das Backend, da man die Labels ohne Probleme umbenennen kann.

6.5.3 Übersetzung rekursiver Wertbindungen

Bei der Übersetzung einer rekursiven Wertbindung werden alle durch die Wertbindung gebundenen Bezeichner durch ein Flag für rekursive Bezeichner gekennzeichnet. Diese Kennzeichnung ist notwendig, da die Annotationen an rekursive Bezeichner in Ausdrücken erst nach vollständiger Elaboration des Ausdrucks erfolgen können. Der Grund hierfür liegt darin, daß der Typ während der Elaboration der Wertbindung durch die Unifikation ständig spezieller werden kann. Erst wenn der Typ stabil ist, können die entsprechenden Annotationen vorgenommen werden. Bei der Definition der Transformationsregeln in Kapitel 5 tritt dieses Problem nicht auf, da die Transformationsregeln sozusagen schon vorher den Typ genau kennen.

Nach der Elaboration einer rekursiven Wertbindung bekommen alle rekursiven Bezeichner den normalen Wertbezeichnerstatus.

6.5.4 Benchmarks

Bei der Auswahl der Benchmarks wurden auf [Sch98] zurückgegriffen. Für die Benchmarks wurde das Mozart-Backend [Con99] verwendet. Die Benchmarks wurden auf auf einem Celeron (Medoncino), 466Mhz, 128KB Cache und 256MB Speicher durchgeführt.

Da der Übersetzer prototypisch ist und bei der Implementierung auf Optimierungen keinen großen Wert gelegt wurde, sind die Benchmarks im Allgemeinen nicht aussagekräftig.

Takeushi

Der Takeushi ist ein Standard-Benchmark für funktionale Sprachen. Er zeichnet sich durch sehr viele rekursive Aufrufe aus. Bei jeder Rekursion werden ein paar arithmetische Operationen ausgeführt. Dadurch ist dieser Benchmark hervorragend geeignet, das Laufzeitverhalten bei statischer Bindung zu untersuchen.

Für die Tests wird folgende Klasse und `int`-Instanz verwendet:

```
class 'a TAKEUSHI_OP with
  val - : 'a * 'a -> 'a
  val < : 'a * 'a -> bool
  val E : 'a
end

instance int TAKEUSHI_OP with
  val op- = Int.-
  val op< = Int.<
  val E = 1
end
```

Der Takeushi-Benchmark wurde auf zwei Arten implementiert. Die erste Variante verwendet direkt die arithmetischen Operationen aus dem `Int`-Modul und die zweite Variante bindet obige Instanz statisch.

Der Takeushi-Benchmark wurde mit `tak (24, 16, 8)` aufgerufen, was zu ca. 2,5 Millionen Funktionsaufrufen führt. Die Messung wurde auf den oben genannten Systemen jeweils 5 mal durchgeführt.

Die folgende Tabelle zeigt die Zeitmessungen in Sekunden. In der dritten Spalte stehen die Zeiten für eine von Hand optimierte Version.

	Direkter Aufruf	Statische Bindung	Hand-optimierte
1. Messung	14	25	19
2. Messung	15	24	19
3. Messung	14	24	19
4. Messung	14	24	19
5. Messung	14	24	19
Arith. Mittel	14,2	24, 2	19

Die statische gebundene Variante benötigt ca. 70% mehr Zeit in Bezug auf die Variante, bei der die arithmetischen Funktionen direkt aufgerufen werden. Der Grund dafür liegt in der schlechten Übersetzung des statischen Aufrufs. Und zwar wird für jede Verwendung einer Methode, die zugehörige Methodentabelle extra berechnet. Diese Berechnung ist extrem teuer.

Bei der von Hand optimierten Version wurde das Erzeugen der Methodentabelle herausgezogen, so daß nicht bei jeder Verwendung einer Methode die Methodentabelle neu erzeugt wird. Die von Hand optimierte Version benötigt trotzdem noch ca. 30% mehr Zeit, als die Implementierung, die die arithmetischen Operationen direkt aufruft. Diese Kosten sind der Selektion der Methode aus der Methodentabelle zuzurechnen.

Die statische Bindung läßt sich also offensichtlich wie folgt optimieren: Zunächst wird das Erzeugen der Methodentabellen herausgezogen, so daß jede Methodentabelle nur noch einmal erzeugt wird. Außerdem könnte es unter Umständen sinnvoll sein, die benötigten Methoden in den lokalen Skopus einzuführen und direkt aufzurufen.

6.5.5 Weitere Optimierungen

Bei der Transformation von TML in ML ohne Typklassen werden bei der Deklaration von Funktionen formale Argumente zur Übergabe von Methodentabellen eingeführt.

Es gibt drei Varianten, die formalen Argumente einzufügen:

- als ein Tupel,
- als ein Tupel inklusive der ursprünglichen Argumente der Funktion,
- in gecurrieter Form.

Da verschiedene Backends verschiedene Varianten optimieren, wäre es die eleganteste Lösung, die Entscheidung dem Backend zu überlassen. Dazu muß die Zwischendarstellung geeignet definiert werden.

6.6 Einschränkungen der Implementierung

Folgende Merkmale von TML konnten aus Zeitgründen nicht mehr vollständig in die Implementierung eingebaut werden. Alle lassen sich jedoch ohne Probleme realisieren.

- Instanzen über Recordtypen.
- Das Auflösen impliziter Typvariablen.
- Constraintannotationen an explizite Typvariablen bei Wertbindungen.
- Structure Sharing Constraints.

Kapitel 7

Ausblick

In diesem Kapitel werden kurz mögliche Erweiterungen von TML vorgestellt.

7.1 Konstruktorklassen

Eine der wichtigsten Erweiterungen des Typklassenkonzepts von Haskell war die Erweiterung um Konstruktorklassen. Konstruktorklassen erlauben es, Typklassen über Typkonstruktoren zu parametrisieren.

Zum Beispiel definiert die Klasse `MAPABLE` eine Methode `map` über beliebigen Datenstrukturen.

```
class 'c MAPABLE with
  val map : ('a -> 'b) -> 'a 'c -> 'b 'c
end
```

Denkbare Instanz-Kandidaten könnten `list` oder `tree` sein.

7.2 Automatisches Ableiten von Instanzen

Ähnlich wie in Haskell wäre es wünschenswert, für manche Klassen, wie zum Beispiel für die Klasse `EQ`, Instanzen automatisch zu erzeugen.

7.3 Superklassen

Superklassen dienen dazu, Relationen ähnlich der Vererbungsrelation in objektorientierten Programmiersprachen auszudrücken. Außerdem vereinfachen sie Typschemata, da man die Kontexte auf den Typvariablen durch Weglassen der Superklassen vereinfachen kann.

Betrachten wir zunächst ein einfaches Beispiel, in dem die Klasse A eine Superklasse der Klasse B ist.

```
class 'a A with
  val m : 'a -> 'a
end

class 'a A => 'a B with
  val n : 'a -> 'a
end
```

Beim Instantiieren von B muß eine entsprechende Instanz von A bereits vorhanden sein. Folgendes Beispiel führt zu einem Fehler, da noch keine `int`-Instanz der Klasse A definiert wurde.

```
instance int B with ... end
```

7.4 Verschränkt rekursive Klassen

In TML können in Klassenspezifikationen Klassenkontexte an Typvariablen notiert werden, die nicht der Klassentypvariable entsprechen. Derzeit gibt es aber keine Möglichkeit verschränkt, rekursiv Methoden zu spezifizieren.

```
class 'a A with
  val m : 'b B => 'a * 'b -> unit
and 'b B with
  val n : 'a A => 'a * 'a -> unit
end
```

7.5 Default-Methoden

Default-Methoden dienen dazu, bereits bei der Deklaration für bestimmte Methoden eine Implementierung anzugeben. Zum Beispiel wird in folgender Klasse EQ der Ungleichheitstest auf den Gleichheitstest zurückgeführt.

```
class 'a EQ with
  val == : 'a * 'a -> bool
  val != : 'a * 'a -> bool
  fun x != y = x == y
end
```

7.6 Multi-Parameter-Typklassen

Die in [JJM97] diskutierten Multi-Parameter-Typklassen ermöglichen die Definition von Klassen mit mehreren Klassentypvariablen.

Zum Beispiel:

```
class ('a, 'b) ISOMORPHISM with
  val iso : 'a -> 'b
  val osi : 'b -> 'a
end
```

Dabei soll `osi` die Umkehrabbildung zu `iso` sein.

Eine Typklasse mit einer Klassentypvariable teilt die Menge aller Typen in zwei Äquivalenzklassen. Dabei entspricht die eine Äquivalenzklasse der Menge der Typen, über die die in der Klasse spezifizierten Methoden definiert sind und die andere gerade der Menge der Typen, über die die Methoden nicht definiert sind.

Im Vergleich dazu kann man mit Multi-Parameter-Klassen beliebige Relationen zwischen Typen ausdrücken.

7.7 Existentielle Typen

Objektorientierte Sprachen haben in der Vergangenheit gezeigt, daß sie sich hervorragend dazu eigenen, Gemeinsamkeiten heterogener Objekte zu modellieren. Wesentliches Merkmal ist dabei die Möglichkeit, erweiterbare Subtyp-Hierarchien aufzubauen. Im Gegensatz zu statisch getypten funktionalen Programmiersprachen ist die Typinferenz bei objektorientierten Sprachen gar nicht oder nur sehr schwach entwickelt. Außerdem leiden alle objektorientierte Sprachen unter der Kontravarianz-Regel [AC96].

Konstantin Läufer stellt in [Läu94] eine Möglichkeit vor, existentiell quantifizierte Typen mit Hilfe von Typklassen zu realisieren. Der in Kapitel 2 vorgestellte `Exists`-Funktor demonstriert bereits die Expressivität dieses Konzeptes für eine eingeschränkte Menge von Typklassen.

Anhang A

Benchmarks

A.1 Takeushi

Folgende Klasse und Instanz wurden zur Implementierung des Takeushi-Benchmarks verwendet.

```
  class 'a TAKEUSHI_OP with
    val - : 'a * 'a -> 'a
    val < : 'a * 'a -> bool
    val E : 'a
  end

  instance int TAKEUSHI_OP with
    val op- = Int.-
    val op< = Int.<
    val E = 1
  end
```

Takeushi mit statischer Bindung

```
stak : int * int -> int

  val rec stak =
     fn (x:int, y:int, z:int) =>
        if y<x then stak (stak (x-E, y, z), stak (y-
  E, z, x), stak (z-E, x, y))
        else z
```

Takeushi mit direkten Integer-Operationen

```
ntak : int * int -> int
```

```
local
    val op- = Int.-
    val op< = Int.<
in
    val rec ntak =
       fn (x, y, z) =>
  if y<x then ntak (ntak (x-1, y, z), ntak (y-
1, z, x), ntak (z-1, x, y))
  else z
end
```

Anhang B

Hilfsmittel

Schriftlicher Teil

- Diese Arbeit wurde mit der Dokumentenbeschreibungssprache LaTeX erstellt.
- Das Unix Tool Awk wurde zum automatischen Formatieren der Beispiele verwendet.

Implementierung

- Standard ML of New Jersey [Lab] und ML-Lex und ML-Yacc.
- Mozart VM [Con99] als Zielplatform.
- Standard Unix-Tools.

Literaturverzeichnis

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[Car87] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, April 1987.

[Con99] Mozart Consortium. The Mozart Programming System, 1999. `http://www.mozart-oz.org/`.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM, January 1982.

[HHPW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

[Hin69] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc*, 146:29–60, 1969.

[HJW+92] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partian, and J. Peterson. Report on the programming language Haskell, version 1.2. *Sigplan*, 27(5), May 1992.

[JHS98] David Jeffery, Fergus Henderson, and Zoltan Somogyi. Type classes in Mercury. Technical report, Department of Computer Science, University of Melbourne, Australia, September 1998.

[JJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: Exploring the design space. In *Haskell Workshop*, 1997.

[Jon92] Mark Philip Jones. *Qualified Types: Theory and Practice*. PhD thesis, Oxford University, July 1992. Also available as Programming Research Group technical report 106.

[JW92] Simon L. Peyton Jones and Philip Wadler. A static semantics for Haskell. Technical report, University of Glasgow, February 1992.

[Kae88] Stefan Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 131–144. Springer Verlag, 1988.

[KR00] Leif Kornstaedt and Andreas Rossberg. Stockhausen Operette 1, 2000. `http://www.ps.uni-sb.de/stockhausen/operette1/`.

[Lab] Lucent Technologies; Bell Laboratories. SML of New Jersey. `http://cm.bell-labs.com/cm/cs/what/smlnj/`.

[Läu94] Konstantin Läufer. Combining type classes and existential types. In *Proceedings of the Latin American Informatics Conference*, Mexico, 1994.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MTH90] Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (revised)*. MIT Press, Cambridge, Massachusetts, London, England, 1997.

[NP93] Tobias Nipkow and Christian Prehofer. Type checking type classes. In ACM, editor, *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Charleston, South Carolina, January 10–13, 1993*, pages 409–418, New York, NY, USA, 1993. ACM Press.

[NP95] Tobias Nipkow and Christian Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, April 1995.

[NS91] Tobias Nipkow and Gregor Snelting. Type classes and overloading resolution via order-sorted unification. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 1–14. Springer Verlag, June 1991.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK, 1998.

[OWW95] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 135–146, La Jolla, California, June 25–28, 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.

[PJ93] John Peterson and Mark Jones. Implementing type classes. *SIGPLAN Notices*, 28(6):227–236, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*.

[Ros99] Andreas Rossberg. Hamlet: A model implementation of Standard ML, 1999. `http://www.ps.uni-sb.de/~rossberg/hamlet/`.

[Sch98] Ralf Scheidhauer. *Design, Implementierung und Evaluierung einer virtuellen Maschine für Oz*. Doctoral thesis, Fachbereich Informatik,Universität des Saarlandes, Saarbrücken, Deutschland, December 1998.

[WB89] Philip Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. *16th ACM Symposium on Principles of Programming Languages, Austin Texas, ACM*, 1989.