Theorie und Implementation objektorientierter Programmiersprachen

Andreas Abel
Ludwig-Maximilians-Universität

Rohfassung vom 1. Oktober 2002

Skript zur Vorlesung *Theorie und Implementation objektorientierter Programmiersprachen*. Weiteres Material ist bereitgestellt unter
`http://www.tcs.informatik.uni-muenchen.de/~abel/OO/index.html`.

Kommentare und Verbesserungsvorschläge bitte an `abel@informatik.uni-muenchen.de`.

Dieses Skript ist in Bearbeitung. Die Präsentation des Stoffes ist unvollständig und höchstwahrscheinlich fehlerbehaftet. Zitate verwandter Arbeiten fehlen. Bitte dieses Dokument nicht zitieren.

Inhaltsverzeichnis

1 Einführung **1**
- 1.1 Geschichte objektorientierter Programmierung . . . 2
- 1.2 Wesentliche Konzepte der Objektorientierung . . . 2
- 1.3 Sicherheit durch formale Methoden . . . 2
- 1.4 Typ-Systeme . . . 3
- 1.5 Statische und dynamische Typprüfung . . . 3
- 1.6 Typprüfung für objekt-orientierte Sprachen . . . 3
- 1.7 Ziel und Inhalt der Vorlesung . . . 6

2 Ungetypte arithmetische Ausdrücke **9**
- 2.1 Externe Syntax . . . 9
- 2.2 Interne Syntax . . . 10
  - 2.2.1 Induktiv definierte Mengen . . . 11
  - 2.2.2 Rekursion über induktive Mengen . . . 13
  - 2.2.3 Beweise durch Induktion . . . 14
- 2.3 Semantik . . . 15
  - 2.3.1 Abstrakte Maschine . . . 15
  - 2.3.2 Natürliche Semantik . . . 18

3 Getypte arithmetische Ausdrücke **19**

4 Der ungetypte Lambda-Kalkül **21**
- 4.1 Die λ-Notation . . . 21
- 4.2 Syntax des reinen Lambda-Kalküls . . . 23
- 4.3 Operationale Semantik . . . 23
- 4.4 Programmieren im Lambda-Kalkül . . . 25
  - 4.4.1 Church-Booleans . . . 25
  - 4.4.2 Paare . . . 26
  - 4.4.3 Church-Ziffern . . . 27
  - 4.4.4 Divergenz . . . 28
  - 4.4.5 Funktionen mit mehreren Argumenten und Currying . . . 28
- 4.5 Formale Behandlung des Lambda-Kalküls . . . 29

5 Der getypte Lambda-Kalkül **33**
5.1 Booleans . . . 33
5.2 Einfach-getypter λ-Kalkül . . . 34

6 Erweiterungen des getypten Lambda-Kalküls **37**
6.1 Der ein-elementige Typ . . . 37
6.2 Befehlssequenzen und `let` . . . 37
6.3 Produkttypen . . . 37
6.4 Datensatztypen . . . 37
6.5 Rekursion . . . 37
6.6 Zusammenfassung . . . 37
6.6.1 Syntax . . . 37
6.6.2 Auswertung mit Call-by-Value . . . 38
6.6.3 Typisierung . . . 40

7 Referenzen **43**

8 Untertypen **47**
8.1 Subsumption . . . 47
8.2 Regeln der Untertyp-Beziehung . . . 48
8.3 Entscheidung der Untertyp-Beziehung . . . 48

9 Objekt-orientierte Programmierung **51**
9.1 Objekte, Methoden . . . 51
9.2 Konstruktor . . . 52
9.3 Subtyping . . . 52
9.4 Bündelung der Instanzen-Variablen . . . 52
9.5 Klassen . . . 52
9.6 Zusätzliche Instanzenvariablen . . . 53
9.7 `self` (ohne dynamische Bindung) . . . 53
9.8 `self` (mit dynamischer Bindung) . . . 54

10 Featherweight Java **57**

Literatur **59**

Kapitel 1

Einführung

Objektorientierte Programmiersprachen haben sich einen festen Platz in professioneller Softwareentwicklung erobert. Herausragend sind die Sprachen C++, die eine Erweiterung des "Marktführers" C um objektorientierte Konstrukte darstellt, und JAVA, die besonders zur Entwicklung portabler und platformübergreifender Software verwendet wird. Beide Sprachen sind vergleichsweise jung. Die erste Definition von C++ war 1986 abgeschlossen, JAVA kam 1995 auf den Markt. Das Prinzip der Objektorientierung ist jedoch beinahe vierzig Jahre alt und findet sich bereits bei SIMULA 1 (1964) und Smalltalk (1972) [Zus99].

Ein *Objekt*, die Grundeinheit objekt-orientierter Programmiersprachen, ist eine Entität mit einem gewissen Verhalten, das durch seine *Methoden* bestimmt wird. Eine Liste von Methoden, über die ein Objekt gesteuert werden kann, wird als *Schnittstelle (Interface)* bezeichnet. Idealerweise spezifiziert diese Schnittstelle das Verhalten des Objekts unter Aufruf seiner Methoden. In realen Programmiersprachen ist jedoch meist nur für jede Methode der *Typ* gegeben, der angibt, wie die Methode syntaktisch korrekt benutzt werden kann. Es gibt jedoch Spezifikationssprachen wie z.B. die Unified Modelling Language (UML), oder Spracherweiterungen wie ESC JAVA [DLNS98], die eine teilweise Definition des Verhaltens ermöglichen.

Ein weiteres Kernkonzept in objektorientierten Programmiersprachen sind *Klassen*. Eine Klasse implementiert ein Schnittstelle; sie definiert die internen Datenelemente *(Felder)* von Objekten eben dieser Klasse und implementiert die Methoden der Schnittstelle. Jedes Objekt wird genau einer Klasse zugeordnet, kann aber mehreren Schnittstellen genügen. Zur Laufzeit eines objektorientierten Programmes muss klar sein, welcher Klasse ein bestimmtes Objekt angehört, um Aufrufe von Methoden dieses Objekts ausführen zu können. Die Klasse eines Objekts kann auch einfach als ihr *Typ* angesehen werden.

[Bla bla... Hier folgen nun die Folien des ersten Vortrags.]

Rohfassung vom 1. Oktober 2002

1.1 Geschichte objektorientierter Programmierung

1. Vorläufer: SIMULA 1 (1964): Sprache für Simulationen.
2. Smalltalk (1972): Erste rein objektorientierte Sprache.
3. C++ (1986): Hybridsprache: Prozedural und objektorientiert.
4. JAVA (1995): Typsichere (?) OO-Sprache für platformunabhängige Anwendungen.
5. weitere: ObjectPascal, MODULA-3, Oberon ...

1.2 Wesentliche Konzepte der Objektorientierung

Operationelle Konzepte:

1. Methoden und dynamische Bindung
2. Vererbung
3. Späte Bindung

Sicherheitskonzepte:

4. Datenkapselung
5. Subtyping (Interfaces, Subclasses)

1.3 Sicherheit durch formale Methoden

- Anforderung an moderne Programmiersprachen: zur Verlässlichkeit und Robustheit von Software beitragen.
- Vollständige Korrektheitsbeweise durch formale Methoden sind sehr aufwendig und meist zu teuer.
- Schlanke formale Methoden: beschränken sich auf automatisch testbare Eigenschaften von Programmen.
  1. Typ-Systeme
  2. Model-Checker
  3. Laufzeitüberwachungssysteme

1.4 Typ-Systeme

- Definition (Benjamin Pierce [Pie02]):

  > A type system is a syntactic method for automatically proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute.

- Typsicherheit:

  1. Jeder korrekte Programmausdruck hat einen Typ.
  2. Der Typ eines Ausdrucks ändert sich durch Auswertung nicht.
  3. Ein wohlgetypter Ausdruck erzeugt keine unerwarteten Laufzeitfehler.

- Slogan: *Well-typed programs cannot go wrong.*

1.5 Statische und dynamische Typprüfung

	Statisch	Dynamisch
Typprüfung	beim Übersetzen	zur Laufzeit
Vorteil	Fehler früher erkannt	Nur wirkliche Fehler
Nachteil	Korrekte Programme abgelehnt	Fehler "schlummern"
Sprachen	ML, Haskell C, C++, JAVA	LISP, Scheme JAVA

Bemerkung: Nicht alle Sprachen mit Typprüfung sind auch typsicher. Unsicher sind z.B. C und C++.

1.6 Typprüfung für objekt-orientierte Sprachen

- Typsysteme für objekt-orientierte Sprachen sind kompliziert:
- Das Typ-System von JAVA ist stellenweise unnötig restriktiv.
- Beispiel: Cloning.

```
interface Move {}

/* A game board with undo */

abstract class Board {

    Board prev;

    void copyFrom (Board board) {
```

```
        this.prev = board.prev;
    }
    abstract Board copy();

    abstract void doMove (Move move);

    void back () {
        this.copyFrom (this.prev);
    }
    void forward (Move move) {
        Board board = this.copy();
        this.prev = board;
        this.doMove (move);
    }
}

class ChessBoard extends Board {
    boolean turn; /* ... further instance variables */

    ChessBoard () { /* initialize */ }

    void copyFrom (ChessBoard chessBoard) {
        super.copyFrom (chessBoard);
        this.turn = chessBoard.turn; /* ... further code */
    }
    ChessBoard copy() {
        ChessBoard chessBoard = new ChessBoard();
        chessBoard.copyFrom (this);
        return chessBoard;
    }

    void doMove (Move move) { /* perform chess move */ }
}
```

Die abgeleitete Klasse `ChessBoard` kopiert auch sich selbst, deswegen sollte die Methode `copy` ein Objekt vom Typ `ChessBoard` zurückgeben. Dies scheitert jedoch am Compiler:

```
cloning.java:36: The method ChessBoard copy() declared in class
ChessBoard cannot override the method of the same signature
declared in class Board.  They must have the same return type.
    ChessBoard copy() {
               ^
1 error
```

Eine akzeptierte Implementation von `ChessBoard` lautet:

```
class ChessBoard extends Board {
    boolean turn; /* ... further instance variables */

    ChessBoard () { /* initialize */ }

    void copyFrom (Board chessBoard) {
        super.copyFrom (chessBoard);
        this.turn = ((ChessBoard)chessBoard).turn; /* ... further code */
    }
    Board copy() {
        ChessBoard chessBoard = new ChessBoard();
        chessBoard.copyFrom (this);
        return (Board)chessBoard;
    }

    void doMove (Move move) { /* perform chess move */ }
}
```

- Manchmal ist das operationelle Verhalten überraschend.
- Beispiel: Equality.

```
/* classes with equality */

class A {
    int x;

    A (int x) { this.x = x; }
    boolean eq(A a) {
        return this.x == a.x;
    }
}

class B extends A {
    int y;

    B (int x, int y) { super(x); this.y = y; }
    boolean eq(B b) {
        return super.eq(b) && this.y == b.y;
    }
}

class Eq {
    public static void main (String[] args) {
        A a1 = new A(1);
        B b1 = new B(1,2);
```

```
        if (a1.eq(b1)) System.out.println ("a1 = b1");
        else System.out.println ("NOT a1 = b1");
        if (b1.eq(a1))  System.out.println ("b1 = a1");
        else System.out.println ("NOT b1 = a1");
    }
}
```

Das Ergebnis verblüfft und ist bedingt durch overloading.

```
a1 = b1
b1 = a1
```

- Das Typ-System von JAVA enthält fehlerhafte Regeln.
- Beispiel: Arrays.

```
class A { }

class B extends A { void bla() { System.out.println("Class B blabla."); } }

class C {

    A a = new A();

    void assign (A[] v) {
        v[0] = a;
    }

    public static void main (String[] args) {
        C c = new C();
        B[] v = new B[1];
        c.assign (v);
        v[0].bla();
    }
}
```

Das Programm produziert einen Typfehler zur Laufzeit, obwohl es keine kritischen Casts enthält. Ursache ist eine fehlerhafte Typregel für Arrays.

```
Exception in thread "main" java.lang.ArrayStoreException
        at C.assign(Compiled Code)
        at C.main(Compiled Code)
```

1.7 Ziel und Inhalt der Vorlesung

Ziel: Beherrschung von Typsystemen für OO-Sprachen.

1. Theorie
   - Schrittweise Einführung in Typen und korrekte Typregeln.
   - Beweis von Typsicherheit für zunehmend mächtigere Systeme.
2. Implementation
   - Umsetzen der theoretischen Erkenntnisse in funktionierende Typprüfer.
   - Entwicklung eines typsicheren Interpreters für eine objektorientierte Sprache.

Kapitel 2

Ungetypte arithmetische Ausdrücke

In diesem Kapitel führen wir eine simple "Programmiersprache" für Ganzzahlen und Zeichenketten ein. Die Sprache ist so rudimentär und hat den Namen Programmiersprache eigentlich nicht verdient. Sie hilft uns aber, wichtige Konzepte der formalen Behandlung von Programmiersprachen zu verdeutlichen.

In unserer Metasprache seinen die ganzen Zahlen 0, 1, -1, 2, -2, ... und Zeichenketten `"hallo"`, `"bla"`, ... vorhanden. Mit *Metasprache* meinen wir die Sprache, in der wir unsere *Objektsprachen*, d.h. die uns interessierenden Fragmente von objektorientierten Programmiersprachen, definieren und analysieren. Diese Metasprache ist die Sprache der Mathematik, und in einigen Fällen die Programmiersprache, die wir für die Implementation eines Interpreters oder Typprüfers verwenden. In diesem Kapitel benutzen wir *Haskell* [**?**].

In der Mathematik wird die Menge der ganzen Zahlen mit $\mathbb{Z}$ bezeichnet, die Menge der Zeichenkette bezeichnen wir mit $\mathbb{S}$. In Haskell heißt der Typ der ganzen Zahlen `Int` und der Typ der Zeichenketten `String`. Für ganze Zahlen verwenden wir die Buchstaben i, j, k und für Zeichenketten s.

2.1 Externe Syntax

Die Sprache, die wir in diesem Kapitel vorstellen, behandelt einen kleinen Teil der gültigen Ausdrücke von JAVA. Dieser Teil kann durch folgende (BNF-ähnliche) Grammatik beschrieben werden.

e	::=	n	konstante ganze Zahl 0, 1, -1, ...
	|	s	konstante Zeichenkette, z.B. `"hallo"`
	|	$e_1 + e_2$	Addition
	|	$e_1 - e_2$	Subtraktion
	|	e.`length`()	Länge der Zeichenkette e
	|	`Integer.toString`(e)	Verwandlung der Zahl e in eine Zeichenkette

Beispiel 2.1 *Folgende JAVA-Ausdrücke werden von der Grammatik erfasst.*

```
5 + 4
"hallo".length() + 1
Integer.toString(753).length()
"hallo" + 1.length()
```

Dabei ist der letzte Ausdruck "unsinnig" und wird vom JAVA-Compiler zurückgewiesen. Wie man die sinnvollen von den unsinnigen Ausdrücken unterscheidet, werden wir in Kapitel 3 besprechen.

2.2 Interne Syntax

Um knapp und präzise über Programmausdrücke reden zu können, führt man eine *interne* oder *abstrakte Syntax* ein. Manchmal dient sie auch dazu, eine gedachte Programmiersprache zu definieren und zu analysieren, ohne sich über eine genaue Definition der Syntax den Kopf zerbrechen zu müssen.

t	::=	n	konstante ganze Zahl
	|	s	konstante Zeichenkette
	|	t_1 `plus` t_2	Addition
	|	t_1 `minus` t_2	Subtraktion
	|	`len` t	String-Länge
	|	`str` t	Verwandlung in Zeichenkette

Diese rekursive Definition der gültigen Terme t lässt sich in Haskell durch den folgenden *rekursiven Datentyp* beschreiben.

```
data Term = CInt Int
          | CStr String
          | Plus Term Term
          | Minus Term Term
          | Len Term
          | Str Term
```

Das Symbol `Term` bezeichnet den neuen Typ der Terme, den wir durch die sechs Klauseln definieren. Die Symbole `CInt`, `CStr`, `Plus`, `Minus`, `Len` und `Str` sind sogenannte Konstruktoren, durch die Ausdrücke vom Typ `Term` erzeugt werden können. In unserem Fall haben sie je ein bis zwei Argumente vom Typ `Int`, `String` oder `Term`. Die dritte Klausel z.B. ist so zuverstehen: sind t_1 und t_2 Ausdrücke vom Typ `Term`, so ist `Plus` t_1 t_2 wieder ein Ausdruck vom Typ `Term`.

Beispiel 2.2 *Die JAVA-Ausdrücke aus dem letzten Bespiel lassen sich wie folgt in interne Syntax übersetzen und in Ausdrücke vom Typ* `Term` *übersetzen:*

```
5 plus 4                    Plus (CInt 5) (CInt 4)
(len "hallo") plus 1        Plus (Len (CStr "hallo")) (CInt 1)
len (str 753)               Len (Str (CInt 753))
"hallo" plus (len 1)        Plus (CStr ("hallo")) (Len 1)
```

Wie aus den Beispielen deutlich geworden ist, betrachten wir *Klammern* zwar nicht als primären Bestandteil unserer internen Syntax, benutzen sie aber, um Ausdrücke zu disambiguieren. So ist z.B. die Struktur des Ausdrucks

$$1\ \texttt{minus}\ 2\ \texttt{minus}\ 3$$

nach unserer Grammatik nicht eindeutig; durch Klammersetzung entscheiden wir uns für eine der Möglichkeiten

$$1\ \texttt{minus}\ (2\ \texttt{minus}\ 3)$$

oder

$$(1\ \texttt{minus}\ 2)\ \texttt{minus}\ 3.$$

2.2.1 Induktiv definierte Mengen

Die oben gegebene Grammatik der internen Syntax ist eine Kurzschreibweise für die folgende *induktive Definition* der Menge von Termen Tm.

Definition 2.1 (Terme, induktiv)

1. $\mathbb{Z} \subseteq \mathsf{Tm}$
2. $\mathbb{S} \subseteq \mathsf{Tm}$
3. *Sind* $t_1, t_2 \in \mathsf{Tm}$*, so auch* $t_1\ \texttt{plus}\ t_2,\ t_1\ \texttt{minus}\ t_2 \in \mathsf{Tm}$.
4. *Ist* $t \in \mathsf{Tm}$*, so sind* $\texttt{len}\, t,\ \texttt{str}\, t \in \mathsf{Tm}$.

Diese Definition kann auch durch folgende *Inferenzregeln* ausgedrückt werden:

Definition 2.2 (Terme, durch Inferenzregeln)

$$\frac{i \in \mathbb{Z}}{i \in \mathsf{Tm}} \qquad \frac{s \in \mathbb{S}}{s \in \mathsf{Tm}}$$

$$\frac{t_1 \in \mathsf{Tm} \qquad t_2 \in \mathsf{Tm}}{t_1\ \texttt{plus}\ t_2 \in \mathsf{Tm}} \qquad \frac{t_1 \in \mathsf{Tm} \qquad t_2 \in \mathsf{Tm}}{t_1\ \texttt{minus}\ t_2 \in \mathsf{Tm}}$$

$$\frac{t \in \mathsf{Tm}}{\texttt{len}\, t \in \mathsf{Tm}} \qquad \frac{t \in \mathsf{Tm}}{\texttt{str}\, t \in \mathsf{Tm}}$$

Mit Hilfe dieser Inferenzregeln lassen sich Beweisbäume oder *Herleitungen*, generieren, die belegen, warum ein bestimmter Ausdruck t in Tm ist (bzw. wie er da "hereingekommen" ist). Dabei ist jede Regel so zu lesen: Wenn die *Prämissen* über dem Balken erfüllt sind, so gilt die *Konklusion* unter dem Balken.

Beispiel 2.3 *Wir beweisen, das* `(len "hallo") plus 1` *ein Term unserer Sprache ist.*

$$\dfrac{\dfrac{\dfrac{\texttt{"hallo"} \in \mathbb{S}}{\texttt{"hallo"} \in \mathsf{Tm}}}{\texttt{len}\,\texttt{"hallo"} \in \mathsf{Tm}} \qquad \dfrac{1 \in \mathbb{Z}}{1 \in \mathsf{Tm}}}{(\texttt{len}\,\texttt{"hallo"})\,\texttt{plus}\,1 \in \mathsf{Tm}}$$

Genauso kann man beweisen, dass der unsinnige Ausdruck `"hallo" plus (len 1)` $\in \mathsf{Tm}$*.*

Übung 2.1 *Geben Sie eine Herleitung von* `(len (str "bla")) minus (5 plus 3)` $\in \mathsf{Tm}$ *an.*

Die Menge Tm ist *definiert* als die *kleineste Menge*, die unter den Inferenzregeln abgeschlossen ist.

Wir können die Menge der Terme schrittweise von unten *konstruieren*, mit Hilfe der folgenden Definition.

Definition 2.3 (Terme, schrittweise) *Für jede natürliche Zahl n, definiere eine Menge T_n wie folgt:*

$$\begin{array}{lcl} T_0 & = & \emptyset \\ T_{n+1} & = & \mathbb{Z} \cup \mathbb{S} \\ & \cup & \{t_1\ \texttt{plus}\ t_2, t_1\ \texttt{minus}\ t_2 \mid t_1, t_2 \in T_n\} \\ & \cup & \{\texttt{len}\,t, \texttt{str}\,t \mid t \in T_n\} \end{array}$$

Schliesslich sei $T = \bigcup_{n\in\mathbb{N}} T_n$ der Limes der Folge $(T_n)_{n\in\mathbb{N}}$.

Übung 2.2 *Geben Sie einen Ausdruck t an mit $t \in T_3$ aber $t \notin T_2$.*

Satz 2.1 *Die Mengen T_n sind* kumulativ*, d.h. $T_n \subseteq T_{n+1}$ für beliebiges $n \in \mathbb{N}$.*

Übung 2.3 *Beweisen Sie Satz 2.1.*

Die Konstruktion der Menge der Terme ist korrekt, d.h. T ist die kleinste Menge, die unter den Inferenzregeln in Definition 2.2 abgeschlossen ist.

Satz 2.2 $T = \mathsf{Tm}$*. Das entspricht:*

1. *T ist unter den Regeln in Def. 2.2 abgeschlossen.*
2. *Sei T' unter den Inferenzregeln abgeschlossen. Dann $T \subseteq T'$.*

Beweis.

1. Beispielhaft zeigen wir das t_1 `plus` $t_2 \in T$, falls $t_1, t_2 \in T$. Nach Voraussetzung gibt es Indices $n_1, n_2 \in \mathbb{N}$ mit $t_1 \in T_{n_1}$ und $t_2 \in T_{n_2}$. Sei $n = \max(n_1, n_2)$. Da die Folge $(T_m)_{m\in\mathbb{N}}$ kumulativ ist (Satz 2.1), sind $t_1, t_2 \in T_n$ und damit ist t_1 `plus` $t_2 \in T_{n+1} \subseteq T$.

2. Es genügt zu zeigen, dass für alle Indices $n \in \mathbb{N}$ und alle Terme $t \in T_i$ auch $t \in T'$ gilt. Wir beweisen diese Aussage durch vollständige Induktion über n.

   Sei z.B. $\texttt{len}\, t \in T^n$. Dann ist $n > 1$ und $t \in T_{n-1}$. Nach Induktionsvoraussetzung ist $t \in T'$. Da T' unter den Inferenzregeln abgeschlossen ist, gilt auch $\texttt{len}\, t \in T'$.

□

Übung 2.4 *Vervollständigen Sie diesen Beweis für die noch nicht behandelten Fälle.*

2.2.2 Rekursion über induktive Mengen

In unserer Metasprache können wir nun Funktionen definieren, die Terme manipulieren. Beispielweise eine Funktion $|\cdot| : \mathsf{Tm} \to \mathbb{N}$, die die Größe eines Terms bestimmt. Wir definieren diese Funktion rekursiv, und durch Fallunterscheidung über die verschiedenen Termformen. Wir geben eine Definition in mathematischer Notation und eine in Haskell.

$\lvert\cdot\rvert$	$: \mathsf{Tm} \to \mathbb{N}$	`size :: Term -> Int`
$\lvert i\rvert$	$= 1$	`size (CInt i) = 1`
$\lvert s\rvert$	$= 1$	`size (CStr s) = 1`
$\lvert t_1\ \texttt{plus}\ t_2\rvert$	$= 1 + \lvert t_1\rvert + \lvert t_2\rvert$	`size (Plus  t1 t2) = 1 + size t1 + size t2`
$\lvert t_1\ \texttt{minus}\ t_2\rvert$	$= 1 + \lvert t_1\rvert + \lvert t_2\rvert$	`size (Minus t1 t2) = 1 + size t1 + size t2`
$\lvert\texttt{len}\, t\rvert$	$= 1 + \lvert t\rvert$	`size (Len t) = 1 + size t`
$\lvert\texttt{str}\, t\rvert$	$= 1 + \lvert t\rvert$	`size (Str t) = 1 + size t`

Diese Definition ist dadurch gerechtfertigt, dass eine rekursive Verwendung immer nur mit "kleinerem" Argument auftritt. In der Definition der Größe von t_1 `plus` t_2 z.B. wird sich nur auf die Größe der *Subterme* t_1 und t_2 bezogen. Diese Intuition können wir auch mathematisch präzise fassen. Wir definieren eine Folge von Funktionen $(\mathsf{size}_n)_{n\in\mathbb{N}}$:

$$
\begin{array}{lcll}
\mathsf{size}_0 & : & T_0 \to \mathbb{N} & \\
\ldots & & & \\
\mathsf{size}_{n+1} & : & T_{n+1} \to \mathbb{N} & \text{verwendet } \mathsf{size}_n : T_n \to \mathbb{N} \\
\ldots & & &
\end{array}
$$

Jede dieser Funktion ist *nicht* rekursiv und *approximiert* die zu konstruierende Funktion $|\cdot| : T \to \mathbb{N}$, welche wir als Grenzwert der Folge erhalten.

Funktionen über Terme wie $|\cdot|$, die sich selbst nur mit Subtermen als Argument aufrufen, nennt man auch *strukturell rekursiv*. Eine weitere strukturell

rekursive Funktion berechnet die Höhe eines Terms.

$$
\begin{array}{lcl}
\mathsf{height} & : & \mathsf{Tm} \to \mathbb{N} \\
\\
\mathsf{height}(i) & = & 1 \\
\mathsf{height}(s) & = & 1 \\
\mathsf{height}(t_1\ \texttt{plus}\ t_2) & = & 1 + \max(\mathsf{height}(t_1), \mathsf{height}(t_2)) \\
\mathsf{height}(t_1\ \texttt{minus}\ t_2) & = & 1 + \max(\mathsf{height}(t_1), \mathsf{height}(t_2)) \\
\mathsf{height}(\texttt{len}\, t) & = & 1 + \mathsf{height}(t) \\
\mathsf{height}(\texttt{str}\, t) & = & 1 + \mathsf{height}(t)
\end{array}
$$

Übung 2.5 *Schreiben sie eine strukturell rekursive Funktion, die einen Ausdruck der internen Syntax in externe Syntax übersetzt.*

2.2.3 Beweise durch Induktion

Es gibt drei Arten, beweise für Aussagen über Terme t zu führen:

1. Induktion über die Termhöhe $\mathsf{height}(t) \in \mathbb{N}$.
2. Induktion über die Termgrösse $|t| \in \mathbb{N}$.
3. Strukturelle Induktion über die Konstruktion von $t \in \mathsf{Tm}$.

Alternative 3 vermeidet den Umweg über die natürlichen Zahlen und führt meistens zu einer klaren Beweisstruktur. Wann immer möglich, werden wir strukturelle Induktion verwenden.

Satz 2.3 *Für alle Terme* $t \in \mathsf{Tm}$ *gilt* $\mathsf{height}(t) \leq |t|$.

Beweis. Durch strukturelle Induktion über $t \in \mathsf{Tm}$. Wir müssen nun alle sechs Fälle abhandeln:

Fall $t = i$. Es gilt $\mathsf{height}(i) = 1 \leq 1 = |i|$.

Fall $t = s$. Analog.

Fall $t = t_1\ \texttt{plus}\ t_2$. Nach Induktionsvoraussetzung ist $\mathsf{height}(t_n) \leq |t_n|$ für $n \in \{1, 2\}$. Somit gilt

$$\mathsf{height}(t) = 1 + \max(\mathsf{height}(t_1), \mathsf{height}(t_2)) \leq 1 + |t_1| + |t_2| = |t|.$$

Fall $t = t_1\ \texttt{minus}\ t_2$. Analog.

Fall $t = \texttt{len}\, t'$. Nach Induktionsvoraussetzung ist $\mathsf{height}(t') \leq |t'|$. Damit auch

$$\mathsf{height}(t) = 1 + \mathsf{height}(t') \leq 1 + |t'| = |t|.$$

Fall $t = \texttt{str}(t')$. Analog. □

Übung 2.6 *Beweisen Sie* $|t| \leq 2^{\mathsf{height}(t)} - 1$ *für alle Terme* $t \in \mathsf{Tm}$.

2.3 Semantik

Bis jetzt haben wir die Terme als reine syntaktische Objekte behandelt. Obwohl die Namen der Termkonstruktoren eine gewisse Bedeutung suggerieren, haben wir doch die Semantik nicht definiert.

Es gibt verschiedene Methoden, die Bedeutung von Programmkonstrukten mathematisch präzise zu erklären. Man unterscheidet zwischen

1. *operationeller Semantik*, welche für jeden Term beschreibt, wie er von einer abstrakten Maschine ausgewertet wird,
2. *denotationeller Semantik*, die Terme in mathematische Objekte übersetzt, und
3. *axiomatischer Semantik*, die das Programmverhalten durch logische Axiome beschreibt.

Wir werden uns ausschließlich mit der ersten Alternative befassen, da sie gleichzeitig einen Grundbaustein zur Implementation eines Interpreters darstellt.

In der Praxis findet man häufig eine von zwei weiteren Alternativen.

- Natürlich-sprachliche Semantik. Programmkonstrukte werden informell erklärt.
- Gar keine Semantik. Es gibt keine genaue Sprachbeschreibung. Die Bedeutung eines Programms ergibt sich, in dem man es kompiliert und ausführt.

Diese Ansätze haben entscheidende Schwächen: Falls keine genaue Spezifikation existiert, werden verschiedene Übersetzer ein- und dasselbe Programm mit großer Wahrscheinlichkeit leicht unterschiedlich interpretieren. Anwendungen, die unter einem Compiler entwickelt wurden, funktionieren mit einem anderen Compiler plötzlich nicht mehr. Die Programmiersprache ist nicht portabel.

2.3.1 Abstrakte Maschine

Wir definieren die *Auswertung* eines Programmausdrucks durch eine *abstrakte Maschine*. Die Maschine ist abstrakt in dem Sinne, dass wir konkrete Details ignorieren, z.B. wie Speicher repräsentiert ist etc. Das Ergebnis der Berechnung durch die Maschine ist ein *Wert*. In unserem Fall gibt es nur zwei verschiedene Kategorien von Werten.

$$\begin{array}{rcll} v & ::= & i & \text{Ganzzahl-Konstante} \\ & | & s & \text{String-Konstante} \end{array}$$

Der Zustand der Maschine wird vollständig bestimmt durch den Ausdruck, der gerade ausgewertet werden soll. Durch einen Auswertungsschritt wird die Maschine in den nächsten Zustand überführt, den wir wiederum durch einen Ausdruck charakterisieren. Die Zustandsübergangsrelation können wir daher als binäre Relation zwischen Termen schreiben.

$$t \longrightarrow t' \qquad t \text{ wertet in einem Schritt zu } t' \text{ aus}$$

Welche Terme sollen wir nun in Relation zueinander setzen? Zuerst einmal definieren wir, wie sich unsere Operationen `plus, minus, len, str` auf Werten verhalten.

Definition 2.4 (Auswertungssemantik I: Berechnungsregeln)

$$\frac{}{i_1 \texttt{ plus } i_2 \longrightarrow i_3}\ \text{E-PlusVV} \qquad \textit{wobei } i_3 \textit{ die Zahl } i_1 + i_2 \textit{ bezeichnet}$$

$$\frac{}{i_1 \texttt{ minus } i_2 \longrightarrow i_3}\ \text{E-MinusVV} \qquad \textit{wobei } i_3 \textit{ die Zahl } i_1 - i_2 \textit{ bezeichnet}$$

$$\frac{}{\texttt{len } s \longrightarrow i_s}\ \text{E-LenV} \qquad \textit{wobei } i_s \textit{ die Länge von } s \textit{ bezeichnet}$$

$$\frac{}{\texttt{str } i \longrightarrow s_i}\ \text{E-StrV} \qquad \textit{wobei } s_i \textit{ die Dezimalnotation von } i \textit{ bezeichnet}$$

Einen Ausdruck t, der mit einer der *Berechnungs*regeln in einen Ausdruck t' überführt werden kann ($t \longrightarrow t'$), nennt man *Redex*. Den Term t' nennt man *Redukt*. In unserer Sprache sind Redices von der folgenden Form.

$$\begin{array}{rcl} r & ::= & i_1 \texttt{ plus } i_2 \\ & | & i_1 \texttt{ minus } i_2 \\ & | & \texttt{len } s \\ & | & \texttt{str } i \end{array}$$

Beispiel 2.4 *Redices und ihre Redukte.*

$$\begin{array}{lcl} 5 + 4 & \longrightarrow & 9 \\ \texttt{len "hallo"} & \longrightarrow & 5 \\ \texttt{str } 753 & \longrightarrow & \texttt{"753"} \end{array}$$

Noch ungeklärt ist, wie z.B. der Ausdruck (5 `plus` 4) `minus` (4 `plus` 3) ausgewertet wird. Es fehlen noch Regeln, die angeben, was z.B. mit Termen t_1 `minus` t_2 passieren soll, wenn t_1 oder t_2 keine Werte sind. Im Prinzip haben wir im obigen Beispiel zwei Möglichkeiten:

$$(5 \texttt{ plus } 4) \texttt{ minus } (4 \texttt{ plus } 3) \longrightarrow 9 \texttt{ minus } (4 \texttt{ plus } 3) \longrightarrow 9 \texttt{ minus } 7 \longrightarrow 2,$$

oder

$$(5 \texttt{ plus } 4) \texttt{ minus } (4 \texttt{ plus } 3) \longrightarrow (5 \texttt{ plus } 4) \texttt{ minus } 7 \longrightarrow 9 \texttt{ minus } 7 \longrightarrow 2.$$

Wir könnten nun beide zulassen, dann wäre unsere Auswertung *nicht-deterministisch*. Wir bevorzugen jedoch eine *deterministische Auswertungsstrategie*, in der Fachliteratur under dem Namen *left-most outer-most reduction* oder *head reduktion* bekannt. Diese bearbeitet immer den Redex, der "am weitesten links" im Term bzw. im sogenannten *Kopf* (engl. *head*) des Termes steht. Die folgenden Regeln ermöglichen die Auflösung eines Redexes tief innerhalb eines Terms, jedoch nur des am weitesten links stehenden Redexes.

Definition 2.5 (Auswertungssemantik II: Kongruenzregeln)

$$\frac{t_1 \longrightarrow t_1'}{t_1 \texttt{ plus } t_2 \longrightarrow t_1' \texttt{ plus } t_2}\ \text{E-PlusTT} \qquad \frac{t \longrightarrow t'}{i \texttt{ plus } t \longrightarrow i \texttt{ plus } t'}\ \text{E-PlusVT}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 \texttt{ minus } t_2 \longrightarrow t_1' \texttt{ minus } t_2}\ \text{E-MinusTT} \qquad \frac{t \longrightarrow t'}{i \texttt{ minus } t \longrightarrow i \texttt{ minus } t'}\ \text{E-MinusVT}$$

$$\frac{t \longrightarrow t'}{\texttt{len}\, t \longrightarrow \texttt{len}\, t'}\ \text{E-LenT} \qquad \frac{t \longrightarrow t'}{\texttt{str}\, t \longrightarrow \texttt{str}\, t'}\ \text{E-StrT}$$

Die Kongruenzregeln haben je eine Hypothese: Z.B. liest sich die Regel E-LenT folgendermaßen: Wenn die Maschine t in einem Schritt zu t' auswerten könnte, dann ist der Übergang von $\texttt{len}\, t$ nach $\texttt{len}\, t'$ auch möglich. Die Möglichkeit eines Auswertungsschrittes können wir durch eine Herleitung belegen, an deren Anfang eine Berechnungsregel steht, gefolgt von Kongruenzregeln.

Beispiel 2.5

$$\dfrac{\dfrac{\dfrac{}{\texttt{str}\, 753 \longrightarrow \texttt{"753"}}\ \text{E-StrV}}{\texttt{len}(\texttt{str}\, 753) \longrightarrow \texttt{len}\, \texttt{"753"}}\ \text{E-LenT}}{\texttt{len}(\texttt{str}\, 753) \texttt{ plus } (5 \texttt{ plus } 4) \longrightarrow (\texttt{len}\, \texttt{"753"}) \texttt{ plus } (5 \texttt{ plus } 4)}\ \text{E-PlusTT}$$

Übung 2.7 *Für die folgenden Terme t, entscheiden Sie ob ein Auswertungsschritt zu einem Term t' möglich ist. Wenn ja, geben Sie eine Herleitungen für $t \longrightarrow t'$ an.*

1. $t = 5 \texttt{ minus } ((\texttt{len}\, \texttt{"hallo"}) \texttt{ plus } (2 \texttt{ minus } 3))$.
2. $t = (\texttt{len}\, 753) \texttt{ plus } (\texttt{str}\, \texttt{"hallo"})$.
3. $t = \texttt{str}((4 \texttt{ plus } 1) \texttt{ minus } (\texttt{len}\, \texttt{"hallo"}))$.

Definition 2.6 (Reduktion) *Die durch die Berechnungs- und Kongruenzregeln definierte Relation $\longrightarrow$ heißt* Ein-Schritt-Reduktion *oder* Ein-Schritt-Auswertung *und ist eine* small-step semantics. *Die Regeln nennen wir auch* Reduktionsregeln.

Alle Terme t, die nicht mit irgeneinem anderen Term t' in der Relation $t \longrightarrow t'$ stehen, können auch nicht weiter ausgewertet werden. Diese nennen wir *Normalformen*. Darunter fallen z.B. alle Werte v, aber auch unsinnige Terme wie $;\texttt{minus } 5$.

Definition 2.7 (Normalform) *Gibt es kein t' mit $t \longrightarrow t'$ so ist t in Normalform.*

Satz 2.4 *Jeder Wert v ist in Normalform.*

Beweis. Durch Überprüfen aller Auswertungsregeln. □
Unsere Auswertungsrelation ist in der Tat deterministisch.

Satz 2.5 (Auswertung ist deterministisch) *Wenn $t \longrightarrow t'$ und $t \longrightarrow t''$, dann ist $t' = t''$.*

Beweis. Durch Induktion nach $t \longrightarrow t'$. □

[Beweis ausführen.]

Definition 2.8 (Mehr-Schritt-Reduktion) *Wir definieren die Relationen:*

$\longrightarrow^+$ *transitive Hülle von* $\longrightarrow$
$\longrightarrow^*$ *reflexiv-transitive Hülle von* $\longrightarrow$

Übung 2.8 *Definieren Sie $\longrightarrow^+$ und $\longrightarrow^*$ durch Inferenzregeln. Dabei dürfen Sie $\longrightarrow$ als gegeben voraussetzen.*

Satz 2.6 (Mehr-Schritt-Auswertung ist deterministisch) *Sei t ein Term und u, u' Normalformen. Wenn $t \longrightarrow^* u$ und $t \longrightarrow^* u'$, dann $u = u'$.*

2.3.2 Natürliche Semantik

Im letzten Teil haben wir eine operationelle Semantik in Form einer *small-step* Auswertungsrelation kennengelernt. Dabei wurden die einzelnen Schritte spezifiziert, die eine abstrakte Maschine durchführen kann, um einen Ausdruck in Normalform überzuführen. Alternativ kann man die Semantik auch *natürlich* bzw. in Form einer *big-step* Auswertungsrelation definieren. Diese Relation gibt direkt an, zu welchem Wert ein Term auswertet.

Definition 2.9 (Big-step Semantik) *Die Auswertungsrelation $t \Downarrow v$ ist durch folgende Regeln gegeben.*

$$\frac{}{v \Downarrow v}\ \text{B-Value}$$

$$\frac{t_1 \Downarrow i_1 \qquad t_2 \Downarrow i_2}{t_1\ \texttt{plus}\ t_2 \Downarrow (i_1 + i_2)}\ \text{B-Plus} \qquad \frac{t_1 \Downarrow i_1 \qquad t_2 \Downarrow i_2}{t_1\ \texttt{minus}\ t_2 \Downarrow (i_1 - i_2)}\ \text{B-Minus}$$

$$\frac{t \Downarrow s}{\texttt{len}\, t \Downarrow i_s}\ \text{B-Len} \qquad \frac{t \Downarrow i}{\texttt{str}\, t \Downarrow s_i}\ \text{B-Str}$$

Dabei sind i_s und s_i wie oben definiert.

Für unsere Sprache sind small- und big-step Semantik äquivalent.

Satz 2.7 *Sei t ein Term und v ein Wert.*

1. *Wenn $t \Downarrow v$, dann $t \longrightarrow^* v$.*
2. *Wenn $t \longrightarrow^* v$, dann $t \Downarrow v$.*

Übung 2.9 *Beweisen Sie Satz 2.7. Teil 1 geht direkt mittels struktureller Induktion; für Teil 2 müssen Sie erst die folgende Hilfsaussage beweisen:*

Wenn $t \longrightarrow t'$ und $t' \Downarrow v$, dann $t \Downarrow v$.

Kapitel 3

Getypte arithmetische Ausdrücke

[Mitschrift von Cyril Bitterich:]

Slogan "well-typed programs do not go wrong".

$$T ::= Int \mid String$$

Definition 3.1 *Die Typisierungsrelation $t : T$ ist gegeben durch folgende Schlussregeln (Inferenzregeln):*
$\frac{}{i:Int} T-Int$ $\frac{}{s:String} T-String$ $\frac{t_1:Int \;\; t_2:Int}{t_1 \; plus \; t_t 2:Int} T-Plus$ $\frac{t_1:Int \;\; t_2:Int}{t_1 \; minus \; t_2:Int} T-Minus$
$\frac{t:String}{t:Int} T-Len$ $\frac{t:Int}{t:String} T-Str$

Definition 3.2 *Ein Typsystem ist eine Term-/Programmiersprache zusammen mit einer Typisierungsrelation.*

Definition 3.3 *Ein Typsystem ist deterministisch, wenn zu jedem Term t höchstens ein Typ T existiert mit $t : T$.*

Definition 3.4 *Ein Typsystem ist entscheidbar, wenn ein Algorithmus existiert, der für jeden Term t und jeden Typ T entscheidet, ob $t : T$. Ein solcher Algorithmus heißt Typprüfungs- bzw. Type-Checking-Algorihmus.*

Definition 3.5 *Ein Algorithmus, der zu einem gegebenen Term t einen Typ T berechnet, heißt Typinferenz.*

Beispiel 3.1 $\dfrac{\dfrac{\dfrac{}{''hallo'':String} T-String}{len''hallo'':Int} T-Len \quad \dfrac{}{1:Int} T-Int}{(len''hallo''+1):Int} T-Plus}{str(len''hallo'')+1):String} T-Str$

Definition 3.6 *Ein Typsystem ist sicher, falls es folgende zwei Eigenschaften besitzt.*

1. *Fortschritt (Progress). Ist ein Term wohlgetypt und kein Wert, so kann man ihn auswerten.*
2. *Typerhaltung (Preservation, Subject Reduction). Ist ein Term wohlgetypt, besitzt er nach Auswertung noch den selben Typ.*

Satz 3.1 *(Typerhaltung)*
Falls $t : T$ *und* $t \to t'$*, dann* $t' : T$*.*

Beweis. Induktion über $t : T$.

Fall $\frac{}{c:Int} T - Int$ denn $i \not\to$

Fall $\frac{}{s:String} T - String.\ s \not\to$

Fall $\frac{t_1:Int\ \ t_2:Int}{t_1\ plus\ t_2:Int} T - Plus.$

Unterfall $\frac{t_1 \to t_1'}{t_1 plus t_2 \to t_1' plus t_2} E - PlusTT \qquad \frac{t_1':Int\ t_2:Int}{t_1' plus t_2:Int} T - Plus$

Unterfall $\frac{t_2 \to t_2'}{i_1\ plus\ t_2 \to i_2\ plus\ t_2'} E - PlusVT$ Nach I.V. $t_2' : Int$
Somit

$$\frac{\frac{}{i_1:Int} T - Int\ t_2':Int}{i_1\ plus\ t_2':Int} T - Plus$$

Unterfall $\frac{}{i_1\ plus\ i_2 \to (i_1+i_2)} E - PlusVV \qquad \frac{}{(i_1+i_2):Int} T - Int$

□

Satz 3.2 *(Fortschritt)*
Falls $t : T$*, dann entweder* $t = v$ *oder* $t \to t'$

Kapitel 4

Der ungetypte Lambda-Kalkül

Fast alle Programmiersprachen stellen Konstrukte für Unterprogramme, Sub-Routinen, Prozeduren, Methoden oder Funktionen, um den Programmcode zu modularisieren und zu *abstrahieren*, um wiederkehrende Berechnungen nicht jedes Mal neu implementieren zu müssen. Die Abstraktion ermöglicht erst die Entwicklung grösserer Softwarepakete. Auch in der Mathematik sind die Konzepte Abstraktion und Funktionsbildung von essentieller Bedeutung.

Der Lambda-Kalkül ist eine syntaktische Theorie der Funktionenbildung und -anwendung. Er wurde in den 1920er Jahren von Alonzo Church und Mitarbeitern entwickelt ihm Rahmen der Formalisierung von mathematischen Grundlagen, und fand in der frühen höheren Programmiersprache LISP (ab 1950er, John McCarthy) Verwendung. Peter Landin benutze ihn in den 1960er Jahren als Kern-Programmiersprache, auf die sich alle anderen Programmkonstrukte zurückführen lassen. Einige Beispiele dazu werden wir in Teil 4.4 behandeln.

Heute ist der Lambda-Kalkül Grundlage von Programmiersprachentheorie und Implementation. Er ist Kern von funktionalen Sprachen wie Scheme, ML und Haskell und Teil von PIZZA, einer Erweiterung von JAVA. Er dient dem Studium von Variablenbehandlung, Bindung, Auswertungsstrategien und Typsystemen.

4.1 Die λ-Notation

In der Mathematik sehen wir Funktionen oft dadurch definiert, was bei ihrer *Anwendung* auf ein Argument zurückgegeben wird.

$$\begin{aligned} f(x) &= a \cdot x + b \\ f(0) &= a \cdot 0 + b \end{aligned}$$

Die Anwendung auf Argument 0 ist entsprechend direkt zu berechnen. Seltener wird die Funktion als eigenständiges Objekt definiert. Hier ist die Anwendung

auf ein Argument eine Operation, die erklärt werden muss.

$$\begin{array}{lcl} f & = & x \mapsto a \cdot x + b \\ f(0) & = & (x \mapsto a \cdot x + b)(0) = a \cdot 0 + b \end{array}$$

Der Pfeil $x \mapsto t$ wird gelesen als "x wird abgebildet auf t". Er erzeugt eine Funktion, indem er x im Ausdruck t *abstrahiert*. Der Vorteil dieser Notation ist, dass sie *namenlose* Funktionen erlaubt, die auch innerhalb eines größeren Ausdruckes vorkommen können, wie z.B. $x \mapsto a \cdot x + b$ in der zweiten Zeile. Somit macht die $\mapsto$-Notation Funktionen zu *"Bürgern erster Klasse"*: Sie können so frei verwendet werden wie z.B. relle Zahlen und beliebig in Ausdrücken vorkommen.

Im Lambda-Kalkül schreibt man für die Abstraktion $x \mapsto t$ den Term $\lambda x t$, daher der Name. Die Anwendung (oder *Applikation*) schreibt man durch Hintereinanderstellung ohne Klammerung des Arguments, also $f\ 0$ statt $f(0)$.

$$\begin{array}{lcl} f & = & \lambda x(a \cdot x + b) \\ f\ 0 & = & (\lambda x(a \cdot x + b))\ 0 = a \cdot 0 + b \end{array}$$

Funktionale Programmiersprachen kennen Notationen für anonyme Funktionen, die der mathematischen bzw. der des Lambda-Kalküls ähnlich sind.

```
SML:      fn x => a * x + b
Haskell:  \x -> a * x + b
```

Dabei ist der Schrägstrich \ die beste Approximation an ein λ im ASCII-Zeichensatz.

Um Klammern zu sparen, verwenden wir die Punkt-Notation. *Ein Punkt "." öffnet eine Klammer, die sich so weit rechts schließt wie syntaktisch möglich.* Normalerweise wird der Punkt direkt hinter der abstrahierten Variable eingesetzt.

$$f = \lambda x.\ a \cdot x + b$$

Folgende drei Ausdrücke sind syntaktisch äquivalent.

$$\begin{array}{l} (\lambda x(x))\ (\lambda x(\lambda f(f\ (x\ f)))) \\ (\lambda x\ x)\ (\lambda x \lambda f.\ f\ (x\ f)) \\ \lambda x\ x\ \lambda x \lambda f.\ f.\ x\ f \end{array}$$

Der erste Ausdrück verwendet weder keine Punktnotation und auch nicht die Regel, dass $\lambda x \ldots$ genau einen Ausdruck erwartet, in dem x abstrahiert wird. Zur Sicherheit wurden überall Klammern gesetzt, was die Lesbarkeit beeinträchtigt. Der zweite Ausdruck macht sich die Parsing-Regel für λ teilweise zunutze und verwendet die Punkt-Notation maßvoll. Der letzte Ausdruck hingegen macht vollen Gebrauch von Parsing-Regel und Punkt-Notation und spart alle Klammern ein, allerdings auf Kosten der Lesbarkeit. Wir werden den Punkt nur hinter einer Abstraktion verwenden wie im zweiten Ausdruck.

4.2 Syntax des reinen Lambda-Kalküls

Der reine Lambda-Kalkül kennt als alleinige Operationen Abstraktion und Applikation. *Everything is a function.*

Definition 4.1 ((Interne) Syntax) *Die Terme t des ungetypten Lambda-Kalküls sind durch folgende Grammatik gegeben.*

$$\begin{array}{rcll} t & ::= & x & \textit{Variable} \\ & | & \lambda x t & \textit{Abstraktion, Funktionskonstruktion, -einführung} \\ & | & t_1\ t_2 & \textit{Applikation, Funktionsanwendung, -elimination} \end{array}$$

Die Abstraktion $\lambda x t$ *bindet* die Variable x. Variablen, die durch kein vorangestelltes λ gebunden werden, bezeichnen wir als *frei*. Eine präzise Definition folgt in Teil 4.5.

Beispiel	Freie Variablen	Gebundene Variablen
$\lambda x.\ a \cdot x + b$	a, b	x
$\lambda x.\ y\ (\lambda z.\ x\ z)$	y	x, z
$\lambda x.\ y\ (\lambda y.\ x\ y)$	y	x, y

Tabelle 4.1: Beispiele für freie und gebundene Variablen.

Tabelle 4.1 zeigt einige λ-Terme mit freien und gebunden Variablen. Das letzte Beispiel $\lambda x.\ y\ (\lambda y.\ x\ y)$ enthält y einmal frei, einmal gebunden. In diesem Fall sprechen wir von *freien* und *gebundenen Vorkommen* einer Variable.

Gebundene Variablen dürfen mit einem neuen Namen versehen werden, der in dem Term noch nicht vorkommt. Die Bedeutung des Terms ändert sich dabei nicht, und wir betrachten die entstehenden Terme sogar als syntaktisch gleich. Es gibt also keinen Unterschied zwischen $\lambda x\ x$ und $\lambda y\ y$. Die Umbenennung von gebundenen Variablen bezeichnet man als α-*Konversion* und zwei Terme, die sich nur in den Namen der gebundenen Variablen unterscheiden, als α-*äquivalent*. Solche Variablenumbenennungen sind auch aus der Integralrechnung bekannt, z.B.

$$\left(\int_{-\infty}^{+\infty} e^{-x^2} dx\right)^2 = \int_{-\infty}^{+\infty} e^{-x^2} dx \int_{-\infty}^{+\infty} e^{-y^2} dy = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} e^{-x^2-y^2} dx dy$$

Die Integralnotation $\int t dx$ bindet die Variable x. Sie kann also in y umbenannt werden.

4.3 Operationale Semantik

Die Ausführung der Anwendung einer Funktion auf ein Argument bezeichnet man als β-*Reduktion*. Z.B.

$$(\lambda x.\ a \cdot x + b)\ 0 \longrightarrow a \cdot 0 + b$$

Dabei mussten alle freien Vorkommen von x durch 0 ersetzt werden. Diese Operation bezeichen als *Substitution* und schreiben allgemein $[s/x]t$ (lies "s für x in t). Hier ersetzen wir die freien Vorkommen von x in t durch s. Eine genaue Definition von Substitution folgt in Teil 4.5.

Definition 4.2 (β-Reduktion) *Seien s, t λ-Terme und x eine Variable. Der Term $(\lambda xt)\ s$) wertet in einem Schritt zu $[s/x]t$ aus und wir schreiben*

$$(\lambda xt)\ s \longrightarrow [s/x]t.$$

Dabei bezeichnen wir $(\lambda xt)\ s$ als Redex *(*red*ucible* ex*pression) und $[s/x]t$ als* Redukt*.*

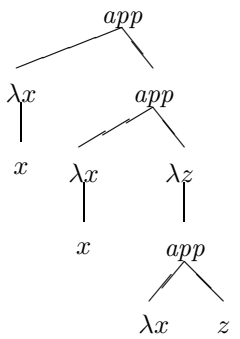
Definition 4.3 (β-Gleichheit) *Zwei Terme t_1 und t_2 sind β-gleich (β-äquivalent), in Zeichen $t_1 =_\beta t_2$, falls man in t_1 und t_2 β-Reduktionen durchführen kann, so dass man Terme t_1' und t_2' erhält, die syntaktisch gleich sind $t_1' = t_2'$.*

Ein Term kann nun viele Redexe enthalten, die entfernt werden müssen um die *Normalform* zu erreichen. Die Reihenfolge, in der Redexe entfernt werden, wird durch ein *Auswertung-Strategie* bestimmt.

Wir unterscheiden 4 Auswertungsstrategien:

Beispiel 4.1
$(\lambda xx)\ \underline{((\lambda xx)\lambda z.\underline{(\lambda xx)z})}$ $\quad \mathtt{id} = \lambda xx$

$\underline{\mathtt{id}(\underline{\mathtt{id}\,\lambda z.\underline{\mathtt{id}\,z}})}$



A Strategien, die alle Redexe entfernen

1. Volle β-Reduktion
   (Redexe könne in beliebiger Reihenfolge bearbeitet werden.) Nicht deterministisch.
   z.B. $\mathtt{id}(\mathtt{id}\,\lambda z.\underline{\mathtt{id}\,z}) \longrightarrow \mathtt{id}(\underline{\mathtt{id}\,\lambda z.z}) \longrightarrow \underline{\mathtt{id}(\lambda z.z)} \longrightarrow \lambda z.z$

2. Normale Auswertung / Kopf- (head-)Reduktion/ Leftmost-outermost
   Entfernt den Redex, der am weitesten links-außen steht, zuerst.
   z.B. $\underline{\mathtt{id}(\mathtt{id}\,\lambda z.\,\mathtt{id}\,z)} \longrightarrow \underline{\mathtt{id}(\lambda z.\,\mathtt{id}\,z)} \longrightarrow \lambda z.\underline{\mathtt{id}\,z} \longrightarrow \lambda z.z$

B Strategien, die nicht unter λ reduzieren

3. Call-by-name / lazy / verzögerte Auswertung
   Wie normale Auswertung; stoppt, wenn Wurzel des Termbaums eine Abstraktion.
   z.B. $\underline{\mathtt{id}(\mathtt{id}\,\lambda z.\,\mathtt{id}\,z)} \longrightarrow \underline{\mathtt{id}\,\lambda z.\,\mathtt{id}\,z} \longrightarrow \lambda z.\,\mathtt{id}\,z$

4. Call-by-value / strict / eager / Applikative Reihenfolge
   Zuerst wird Argument ausgewertet, dann in Funktion eingesetzt.
   $\mathtt{id}(\underline{\mathtt{id}\,\lambda z.\,\mathtt{id}\,z}) \longrightarrow \underline{\mathtt{id}\,\lambda z.\,\mathtt{id}\,z} \longrightarrow \lambda z.\,\mathtt{id}\,z$

Beispiel 4.2 $\mathtt{power3}(x) = x \cdot x \cdot x$

$\mathtt{power3}(100!) \xrightarrow{cbn} 100! \cdot 100! \cdot 100!$

$\mathtt{power3}(100!) \xrightarrow{cbv} v \cdot v \cdot v (v = 100!)$

$\mathtt{ignore}(x) = false$

$\mathtt{ignore}(100!) = \xrightarrow{cbn} false$

$\mathtt{ignore}(100!) = \xrightarrow{cbv} false (v = 100!)$

4.4 Programmieren im Lambda-Kalkül

Der Lambda-Kalkül ist eine Turing-vollständige Programmiersprache, d.h. jede berechenbare Funktion lässt sich im Lambda-Kalkül repräsentieren. Weitere Beispiele für Turing-mächtige Sprachen sind Turing-Maschinen, WHILE- und GOTO-Sprachen, die Peano-Arithmetik, kombinatorische Algebren, jede vernünftige Programmiersprache, aber auch z.B. das Textsatzsystem TEX und die Seitenbeschreibungssprache POSTSCRIPT.

Im Folgenden werden wir einige elementare Datentypen und die wichtigsten Anwendungen dieser Datentypen im Lambda-Kalkül definieren.

4.4.1 Church-Booleans

Die ersten Datenobjekte, den wir betrachten, sind die Wahrheitswerte `true` und `false` und ihre Verwendung mit `if`. Da im Lambda-Kalkül "alles Funktion" ist, müssen wir die drei Konstrukte durch Funktionen realisieren. Wollen wir Datentypen im Lambda-Kalkül repräsentieren, müssen wir folgende Schlüssel-Frage beantworten:

> Wie kann ein Objekt des betrachteten Typs verwendet werden?

Es geht also nicht um die *Konstruktion (Einführung)*, sondern um die *Verwendung (Elimination)* eines Datenobjekts. Ein Datenobjekt im Lambda-Kalkül wird bestimmt durch seine Verwendungsmöglichkeiten.

Ein Wahrheitswert steht für eine *Entscheidung*, welche von zwei alternativen Berechnungen fortgesetzt werden soll. Das aus Programmiersprachen bekannte `if-then-else` ist also die allgemeinste Verwertung eines boolschen Wertes. Alle

anderen Funktionen, wie z.B. logische Verknüpfungen, können mit Hilfe von `if` ausgedrückt werden.

Der Wert `true` ist eine Funktion, die angewandt auf zwei Berechnungen, den *then*-Zweig und den *else*-Zweig, stets den *then*-Zweig wählt. Der Wert `false` verfolgt immer den *else*-Zweig.

$$\begin{array}{lcl} \texttt{true} & = & \lambda t \lambda e.\, t \\ \texttt{false} & = & \lambda t \lambda e.\, e \\ \texttt{if} & = & \lambda b \lambda t \lambda e.\, b\, t\, e \end{array}$$

Der Destruktor `if` tut nichts Besonderes, er wendet den übergebenen Wahrheitswert b auf t, denn *then*-Zweig, und e, den *else*-Zweig an. Die Negationsfunktion können wir nun wie folgt definieren:

$$\texttt{not} \quad = \quad \lambda b.\ \texttt{if}\ b\ \texttt{false}\ \texttt{true}$$

Die Definition is korrekt, wie aus den folgenden Auswertungssequenzen ersichtlich:

$$\begin{array}{l} \texttt{not true} = (\lambda b.\ \texttt{if}\ b\ \texttt{false true})\ \texttt{true} \\ \longrightarrow \texttt{if true false true} = (\lambda b \lambda t \lambda e.\ b\ t\ e)\ \texttt{true false true} \\ \longrightarrow (\lambda t \lambda e.\ \texttt{true}\ t\ e)\ \texttt{false true} \\ \longrightarrow (\lambda e.\ \texttt{true false}\ e)\ \texttt{true} \\ \longrightarrow \texttt{true false true} = (\lambda t \lambda e.\ t)\ \texttt{false true} \\ \longrightarrow (\lambda e.\ \texttt{false})\ \texttt{true} \longrightarrow \texttt{false} \end{array}$$

$$\begin{array}{l} \texttt{not false} = (\lambda b.\ \texttt{if}\ b\ \texttt{false true})\ \texttt{false} \\ \longrightarrow^4 (\texttt{false false true}) = (\lambda t \lambda e.\ e)\ \texttt{false true} \\ \longrightarrow^2 \texttt{true} \end{array}$$

Die Funktion `not` kann noch knapper definiert werden durch den Term $\lambda b.\, b\, \texttt{false}\, \texttt{true}$, welcher sich auch ergibt, wenn man die ursprüngliche Definition normalisiert. Auch folgende Definition ist denkbar:

$$\texttt{not} \quad = \quad \lambda b \lambda t \lambda e.\ b\ e\ t$$

Boolsche Funktionen lassen sich durch (geschachtelte) `if`-Ausdrücke implementieren. Deswegen sind sie auch im Lambda-Kalkül problemlos kodierbar. Z.B. Konjunktion:

$$\texttt{and} \quad = \quad \lambda b \lambda c.\ b\ c\ \texttt{false}$$

Der Ausdruck `and` b c liesst sich: Wenn b, dann c, sonst falsch.

4.4.2 Paare

Ein *Paar* ist der Zusammenschluss zweier Komponenten. Bei der Elimination eines Paares erhälten wir eine der Komponenten zurück. (Wollen wir beide Komponenten, müssen wir das Paar zweimal verwenden.) Welche der beiden Komponenten wir erhalten wollen, geben wir durch einen Wahrheitwert an. Ein

Paar ist also eine Funktion, die ein Bit erwartet und eine Komponente zurückliefert.

$$\begin{array}{lcl} \texttt{pair} & = & \lambda c_{\texttt{true}} \lambda c_{\texttt{false}} \lambda b.\ b\ c_{\texttt{true}}\ c_{\texttt{false}} \\ \texttt{fst} & = & \lambda p.\ p\ \texttt{true} \\ \texttt{snd} & = & \lambda p.\ p\ \texttt{false} \end{array}$$

Die Selektoren `fst` und `snd` tun nichts weiter, als entweder die Nachricht `true` oder `false` an das Paar p zu übergeben. Wieder sind also alle Verwendungsmöglichkeiten im Konstruktor `pair` enthalten. Korrektheit wird ersichtlich durch die Reduktionsfolgen:

$$\begin{array}{l} \texttt{fst}\ (\texttt{pair}\ t_1\ t_2) = (\lambda p.\ p\ \texttt{true})\ (\texttt{pair}\ t_1\ t_2) \\ \longrightarrow (\texttt{pair}\ t_1\ t_2)\ \texttt{true} = (\lambda c_{\texttt{true}} \lambda c_{\texttt{false}} \lambda b.\ b\ c_{\texttt{true}}\ c_{\texttt{false}})\ t_1\ t_2\ \texttt{true} \\ \longrightarrow^3 \texttt{true}\ t_1\ t_2 \longrightarrow^2 t_1 \end{array}$$

$$\begin{array}{l} \texttt{snd}\ (\texttt{pair}\ t_1\ t_2) = (\lambda p.\ p\ \texttt{false})\ (\texttt{pair}\ t_1\ t_2) \\ \longrightarrow^4 \texttt{false}\ t_1\ t_2 \longrightarrow^2 t_2 \end{array}$$

4.4.3 Church-Ziffern

Auch unendliche Datentypen wie die natürlichen Zahlen lassen sich im Lambda-Kalkül repräsentieren. (Dabei ist jede Zahl ein endliches Gebilde.) Eine natürliche Zahl n gibt allgemein an, *wie oft* eine Prozedur ausgeführt wird bzw. eine Funktion angewendet wird. Die Repräsentation der Zahl n im Lambda-Kalkül, die Church-Ziffer $\underline{n}$ ist also eine Funktion, die eine beliebige Funktion f als Argument nimmt und n-mal iteriert. Dabei ist 0-fache Iteration die Identität. Die n-te Iterierte einer Funktion f können wir allgemein so definieren.

$$\begin{array}{lcl} f^0 & = & \lambda x.\ x \\ f^{n+1} & = & f \circ f^n = \lambda x.\ f\ (f^n\ x) \end{array}$$

Damit erhalten wir die Church-Ziffern nach folgendem Schema:

$$\begin{array}{lcl} \underline{0} & = & \lambda f \lambda x.\ x \\ \underline{1} & = & \lambda f \lambda x.\ f\ x \\ \underline{2} & = & \lambda f \lambda x.\ f\ (f\ x) \\ \vdots & & \\ \underline{n} & = & \lambda f \lambda x.\ f^n\ x \end{array}$$

Die Applikation $\underline{n}\ f$ einer Church-Ziffer ist also f^n. Tabelle 4.2 listet einige einfache Funktionen über natürlichen Zahlen auf.

Die Vorgänger-Funktion `pred` mit $\texttt{pred}\ \underline{n} = \underline{n-1}$ ist nicht direkt implementierbar. Man benötigt Paare.

$$\begin{array}{lcl} \texttt{x} & = & \texttt{pair}\ \underline{0}\ \underline{0} \\ \texttt{f} & = & \lambda p.\ \texttt{pair}\ (\texttt{snd}\ p)\ (\texttt{succ}\ (\texttt{snd}\ p)) \\ \texttt{pred} & = & \lambda n.\ \texttt{fst}\ (n\ \texttt{f}\ \texttt{x}) \end{array}$$

Übung 4.1 *Definieren Sie Multiplikation* `mult` *auf Church-Ziffern.*

Funktion	Beschreibung	Spezifikation	Implementation
`succ`	Nachfolger	`succ` $\underline{n} = \underline{n+1}$	$\lambda n \lambda f \lambda x.\ f\ (n\ f\ x)$
`add`	Addition	`add` $\underline{n}\ \underline{m} = \underline{n+m}$	$\lambda n \lambda m \lambda f \lambda x.\ n\ f\ (m\ f\ x)$
`iszero`	Test auf 0	`iszero` $\underline{0}$ = `true` `iszero` $\underline{n+1}$ = `false`	$\lambda n.\ n\ (\lambda\_.\ \texttt{false})\ \texttt{true}$

Tabelle 4.2: Funktionen über natürliche Zahlen.

Übung 4.2 *Definieren Sie Exponentiation* `exp` *auf Church-Ziffern.*

Übung 4.3 *Definieren Sie Subtraktion* `sub` *und euklidische Division*[1] `div` *auf Church-Ziffern.*

4.4.4 Divergenz

Nicht alle Terme des ungetypten Lambda-Kalküls lassen sich auf Normalform reduzieren. Einfachstes Beispiel für einen *divergenten* Term ist $\Omega = (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$, der durch Selbstanwendung entsteht. Ω ist invariant unter Reduktion:

$$\Omega = (\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \longrightarrow [\lambda x.\ x\ x/x](x\ x) = (\lambda x.\ x\ x)\ (\lambda x.\ x\ x) = \Omega$$

4.4.5 Funktionen mit mehreren Argumenten und Currying

Funktionen mit mehreren Argumenten (z.B. `pair`, `add`) haben wir mit Hilfe von mehrfacher Lambda-Abstraktion definiert. Streng genommen ist z.B. `pair` eine Funktion, die die erste Komponente als Argument nimmt und eine Funktion zurückgibt, die wieder ein Argument, die zweite Komponente, erwartet, und dann ein Paar erzeugt. Diese Darstellung hat den Vorteil, dass man `pair` auch *teilweise* anwenden kann. So kann man eine Funktion `pair0` definieren, die nur ein Argument t erwartet und dann ein Paar mit konstanter erster Komponente $\underline{0}$ und zweiter Komponente t erzeugt.

$$\begin{aligned}\texttt{pair0} &= \texttt{pair}\ \underline{0} = (\lambda c_{\texttt{true}} \lambda c_{\texttt{false}} \lambda b.\ b\ c_{\texttt{true}}\ c_{\texttt{false}})\ \underline{0} \longrightarrow \lambda c_{\texttt{false}} \lambda b.\ b\ \underline{0}\ c_{\texttt{false}} \\ \texttt{pair0}\ t &= (\lambda c_{\texttt{false}} \lambda b.\ b\ \underline{0}\ c_{\texttt{false}})\ t \longrightarrow \lambda b.\ b\ \underline{0}\ t =_\beta \texttt{pair}\ \underline{0}\ t\end{aligned}$$

Im Fall von Paaren mag das nicht besonders sinnvoll sein, aber nehmen wir an, wir hätten eine Exponentiationsfunktion `power` definiert mit

$$\begin{aligned}\texttt{power}\ \underline{n}\ \underline{m} &= \underline{m^n} \\ \texttt{power} &= \lambda n \lambda m \ldots\end{aligned}$$

[1] Das ist herkömmliches Teilen mit Rest, wie aus der Grundschule bekannt.

Dann können wir Quadratur $\texttt{sqr}\ \underline{n} = \underline{n^2}$ sehr knapp definieren durch teilweise Applikation von `power`:

$$\texttt{sqr} = \texttt{power}\ \underline{2}$$

Eine Funktion, die teilweise Applikation erlaubt, nennt man ge*curry*t (nach Haskell Curry).

Alternativ können wir im Lambda-Kalkül Funktionen mit mehreren Argumenten über *Paare* definieren, z.B. die Additionsfunktion

$$\texttt{add} = \lambda p.\ \lambda f \lambda x.\ (\texttt{fst}\ p)\ f\ ((\texttt{snd}\ p)\ f\ x)$$

$$\begin{array}{lcl} \texttt{add}\ (\texttt{pair}\ n\ m) & \longrightarrow & \lambda f \lambda x.\ \texttt{fst}\ (\texttt{pair}\ n\ m)\ f\ (\texttt{snd}\ (\texttt{pair}\ n\ m)\ f\ x) \\ & \longrightarrow^2 & \lambda f \lambda x.\ n\ f\ (m\ f\ x) \end{array}$$

Im Lambda-Kalkül sieht diese Variante etwas umständlich aus, jedoch ist dies in den meisten Programmiersprachen die gängige Form. Etwas eleganter wird sie mit der gebräuchlichen Notation $(a_1, \ldots, a_n)$ für n-Tupel. Im SML können wir `add` dann folgendermaßen implementieren.

```
add = fn (n, m) => fn f => fn x => n f (m f x)
```

Das Transformieren einer über Tupel definierte Funktion mit mehreren Argumenten in eine mit iterierter λ-Abstraktion nennt man *currying*, die Gegentransformation *uncurrying*.

Übung 4.4 *Definieren Sie eine λ-Funktion* `curry`*, die eine Funktion f als Argument nimmt, welche wiederum ein Paar von Argumenten erwartet, und eine Funktion zurückliefert, die zwei einzelne Argumente erwartet, aber sonst das Gleiche leistet wie f.*

Übung 4.5 *Definieren Sie auch eine Funktion* `uncurry`*.*

4.5 Formale Behandlung des Lambda-Kalküls

Nach dem wir gesehen haben, was wir *im* Lambda-Kalkül programmieren können, wollen wir den Lambda-Kalkül *von außen* betrachten und schon benutzte Begriffe wie *freie Variable* und *Substitution* präzisieren. Die folgende Definition erfasst wohlgeformte Terme des ungetypten Lambda-Kalküls.

Definition 4.4 (Terme des ungetypten Lambda-Kalküls) *Sei V eine abzählbare[2] Menge für Namen von Variablen. Die Menge Tm der wohlgeformten Terme ist die kleinste Menge, die unter folgenden Regeln abgeschlossen ist:*

1. *Wenn $x \in \mathsf{V}$, dann $x \in \mathsf{Tm}$.*
2. *Wenn $x \in \mathsf{V}$ und $t \in \mathsf{Tm}$, dann $\lambda x t \in \mathsf{Tm}$.*
3. *Wenn $t_1 \in \mathsf{Tm}$ und $t_2 \in \mathsf{Tm}$, dann $t_1\ t_2 \in \mathsf{Tm}$.*

[2] *Abzählbar* impliziert immer *unendlich*.

Die Menge der freien Variablen eines Terms lässt sich durch die folgende strukturell rekursive Funktion implementieren.

Definition 4.5 (Freie Variablen) *Für $t \in \mathsf{Tm}$ bezeichnet $\mathsf{FV}(t) \subseteq \mathsf{V}$ die Menge der freien Variablen von t.*

$$\begin{array}{lcl} \mathsf{FV}(x) & = & \{x\} \\ \mathsf{FV}(\lambda x t) & = & \mathsf{FV}(t) \setminus \{x\} \\ \mathsf{FV}(t_1\ t_2) & = & \mathsf{FV}(t_1) \cup \mathsf{FV}(t_2) \end{array}$$

Nachdem wir Substitution schon intuitiv verwendet haben, wollen wir sie jetzt formal definieren. Um in einem Termbaum t eine Variable x durch einen Term s zu ersetzen, müssen wir in t alle mit x bezeichneten Blätter finden und anstatt ihrer s einhängen. Die Substitutionsfunktion verfährt natürlicherweise also durch Rekursion über t. Hier eine erste Definition:

$$\begin{array}{lcll} [s/x]x & = & s & \\ [s/x]y & = & y & x \neq y \\ [s/x]\lambda y t & = & \lambda y.\ [s/x]t & \\ [s/x](t_1\ t_2) & = & ([s/x]t_1)\ ([s/x]t_2) & \end{array}$$

Der Fall $\lambda y t$ hat noch zwei Schwachstellen. Erstens, auch gebundene Variablen werden ersetzt (falls $y = x$), z.B.

$$[z/x]\lambda x\ x = \lambda x.\ [z/x]x = \lambda x\ z$$

Die Bedeutung dieses Terms hat sich geändert, eine gebundene Variable ist plötzlich frei geworden. Der umgekehrte Fall kann auch eintreten, es können freie Variablen in s "eingefangen" werden (engl. *variable capture*), z.B.

$$[x/z]\lambda x.\ z\ x = \lambda x([x/z]z x) = \lambda x.\ x\ x$$

Beide Probleme können behoben werden, wenn wir bei der Substitution die gebundene Variable y durch eine *frische* Variable y' ersetzen, die weder in s noch in t vorkommt. Solche neuen Variablen sind immer verfügbar, da wir eine unendliche Variablenmenge V zugrunde gelegt haben. Außerdem ändert sich gemäß α-Regel die Bedeutung des Termes nicht. Nun also die verbesserte Definition der Substitution.

Definition 4.6 (Substitution) *Für Terme $s, t \in \mathsf{Tm}$ und eine Variable $x \in V$ bezeichnet $[s/x]t \in \mathsf{Tm}$ den Term, der durch Ersetzung aller freien Vorkommen von x in t entsteht.*

$$\begin{array}{lcll} [s/x]x & = & s & \\ [s/x]y & = & y & x \neq y \\ [s/x]\lambda y t & = & \lambda y'.\ [s/x][y'/y]t & y' \text{ neue Variable} \\ [s/x](t_1\ t_2) & = & ([s/x]t_1)\ ([s/x]t_2) & \end{array}$$

Falls in der dritten Zeile $x = y$, kann t keine freien Vorkommen von x enthalten, die substituiert werden müssten. An dieser Stelle kann man die Substitution beenden. Außerdem, falls $y \neq x$ und $y \notin \mathsf{FV}(s)$, kann man sich das Umbennenen sparen. Damit ergeben sich folgende Optimierungen:

$$\begin{array}{lcl} [s/x]\lambda xt & = & \lambda xt \\ [s/x]\lambda yt & = & \lambda y.\ [s/x]t \quad \text{falls } y \neq x \text{ und } y \notin \mathsf{FV}(s) \end{array}$$

Kapitel 5

Der getypte Lambda-Kalkül

[Hier behandeln wir Teile 9.1-9.3 aus Pierce [Pie02]. Ich bin Cyril Bitterich sehr verbunden, der die folgende Mitschrift (und weitere) zur Verfügung gestellt hat.]

5.1 Booleans

t ::= true | false | if t_1 then t_2 else t_3
v ::= true | false

Berechnungs-Regeln:

$$\frac{}{\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2}\ \text{E-IfTrue}$$

$$\frac{}{\texttt{if false then } t_2 \texttt{ else } t_3 \longrightarrow t_3}\ \text{E-IfFalse}$$

Kongruenz-Regel:

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3}\ \text{E-If}$$

Typisierung:

$$\frac{}{\texttt{true} : \texttt{Bool}}\ \text{T-True} \qquad \frac{}{\texttt{false} : \texttt{Bool}}\ \text{T-False}$$

$$\frac{t_1 : \texttt{Bool} \qquad t_2 : T \qquad t_3 : T}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \ : \ T}\ \text{T-If}$$

5.2 Einfach-getypter λ-Kalkül

Beispiel 5.1 $\lambda xt : Fun$
$(\lambda xx)true : Bool$
$(\lambda x \lambda yy)true :\ Fun$
(Hier sind (λxx) *und* $(\lambda x \lambda yy)$ *vom Typ Fun)*

$\lambda xx : Bool \to Bool$ $\quad$ $\lambda xx : Int \to Int$
$\lambda x \lambda yy : Bool \to (Bool \to Bool)$ $\quad$ $\lambda x \lambda yy : Bool \to Int \to Int)$

$T_1 \to (T_2 \to T_3)$: Typ von Funktionen, die ein Argument vom Typ T_1 erwarten und eine Funktion vom Typ $T_2 \to T_3$ zurückliefern.
$(T_1 \to T_2) \to T_3$: Typ der Funktionen, die eine Funktion vom Typ $T_1 \to T_2$ als Argument erwarten und etwas vom Typ T_3 zurückliefern.

$\to R$
$n(2x)$
$.sin(2x) : R \to R$

$\lambda xx : A \to A$ SML: $(fnx \Rightarrow x) :' a \to' a$
$\lambda x : Bool.x : Bool \to Bool$
$\underbrace{\lambda x : Int.x}_{t} : \underbrace{Int \to Int}_{T}$

Definition 5.1 *Syntax des reinen einfach getypten Lambda-Kalküls* ($\lambda^{\to}$)

Terme: $t ::= x$
$\quad\quad | \ \lambda\ x : T\ t$
$\quad\quad | \ t_1\ t_2$

Typen: $T ::= T \to T$

Kontexte: $T ::= .$ $\quad$ *leerer Kontext*
$\quad\quad | \ \Gamma\ ,\ x{:}\ T$ $\quad$ *erweiterter Kontext*

$\lambda x : Bool$ $\quad \lambda y : Int\ x : Bool \to Int \to Bool$
$\quad\quad\quad\quad \lambda y : Int\ x : \quad\quad Int \to Bool$
$\quad\quad\quad\quad\quad\quad x : \quad\quad\quad\quad Bool$

$$\frac{\frac{}{x:Bool, y:Int \vdash x:Bool}\, T-VAR}{\frac{}{x:Bool \vdash \lambda y:Int\ x:Int \to Bool}\, T-LAM}\, T-LAM$$
$$\vdash \lambda x : Bool\ \lambda y : Int\ x : Bool \to Int \to Bool$$

$$\frac{\Gamma \vdash t_1 : T' \to T \quad \Gamma \vdash t_2 : T'}{\Gamma \vdash t_1\ t_2 : T}$$

Definition 5.2 *Typisierungs-Relation* $\Gamma \vdash t : T$
"Im Kontext Γ *hat* t *den Typ* T*".*

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\, T - Var$$

$$\frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda x : S.t : S \to T} T - LAM$$

$$\frac{\Gamma \vdash t : S \to T \quad \Gamma \vdash s : S}{\Gamma \vdash ts : T} T - APP$$

Definition 5.3 *($\lambda^{\to,Bool}$)*
Der Kalkül $\lambda^{\to,Bool}$ entsteht durch Hinzunehmen von Booleans zu $\lambda^{\to}$

Beispiel 5.2

In $\lambda^{\to,Bool}$ gibt es folgende Typen:	*Rang*	
$Bool$	*0*	
$Bool \to Bool$, $Bool \to Bool \to Bool$	*1*	*"first-order"*
$(Bool \to Bool) \to Bool, ...$	*2*	↓
$((Bool \to Bool) \to Bool) \to Bool$	*3*	*"higher-order"*

Definition 5.4 *(Rang, Ordnung)*
$rk(Bool) = 0$
$rk(T_1 \to T_2) = max(rk(T_1) + 1, rk(T_2))$

Nun interessiert uns auch wieder die Frage der Typsicherheit: Dafüer benötigen wir auch wieder:

Definition 5.5 *Auswertung cbv für geschlossene Terme*
Werte $v ::= \lambda x : S.t$

$$\frac{}{(\lambda x : S.t)v \to [v/x]t} E - BetaV$$

$$\frac{t_1 \to t_1'}{t_1 t_2 \to t_1' t_2} E - AppT$$

$$\frac{t \to t'}{vt \to vt'} E - AppV$$

Satz 5.1 *Fortschritt*
Wenn $\vdash t : T$, dann $t \to t'$ oder $t = v$.

Lemma 5.1 *Inversion, Erzeugung, Generation*

1. *Wenn $\Gamma \vdash x : T$, dann $x : T \in T$*
2. *Wenn $\Gamma \vdash \lambda x : S.t : R$, dann $R = S \to T$ und $\Gamma, x : S \vdash t : T$*
3. *Wenn $\Gamma \vdash t_1 t_2 : T$, dann $\Gamma \vdash t_1 : S \to T$ für ein S und $T \vdash t_2 : S$*

4. *Wenn* $\Gamma \vdash true : T$ *, dann* $T = Bool$
5. *Wenn* $\Gamma \vdash false : T$ *, dann* $T = Bool$
6. *Wenn* $\Gamma \vdash if\ t_1\ then\ t_2\ else\ t_3 : T$ *, dann* $\Gamma \vdash t_1 : Bool$ *und* $\Gamma \vdash t_2 : T$, $\Gamma \vdash t_3 : T$

Beweis. Durch Fallunterscheidung über die Typherleitung. □

Kapitel 6

Erweiterungen des getypten Lambda-Kalküls

6.1 Der ein-elementige Typ

6.2 Befehlssequenzen und `let`

6.3 Produkttypen

6.4 Datensatztypen

6.5 Rekursion

6.6 Zusammenfassung

6.6.1 Syntax

Typen.

T	::=	$T_1 \to T_2$	Funktionstyp
	|	`Int`	Typ der Ganzzahlen
	|	`Bool`	Typ der Wahrheitswerte
	|	`Unit`	Ein-elementiger Typ
	|	$T_1 \times T_2$	Produkttyp
	|	$\{\overrightarrow{l{:}T}\}$	Datensatz-Typ

Terme.

t	::=	$x \mid \lambda x{:}T\ t \mid t_1\ t_2$	Variablen, Funktionen, Anwendung
	$\mid$	$i \mid t_1 + t_2 \mid t_1 - t_2 \mid \texttt{iszero}\ t$	Ganzzahl-Konstanten, Addition, Subtraktion, Test auf 0
	$\mid$	$\texttt{true} \mid \texttt{false} \mid \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3$	Wahrheitswerte, Fallunterscheidung
	$\mid$	$()$	Leeres Tupel
	$\mid$	$(t_1, t_2) \mid t.1 \mid t.2$	Paarbildung und Projektionen
	$\mid$	$\{l_1{=}t_1, \ldots, l_n{=}t_n\} \mid t.l$	Datensatz und Projektion
	$\mid$	$\texttt{let}\ x{=}t_1\ \texttt{in}\ t_2$	Lokale Variablen
	$\mid$	$\texttt{fix}\ f{:}T\ t$	Rekursion

Werte.

v	::=	$\lambda x{:}T t$	Funktionen
	$\mid$	i	Ganzzahl-Konstanten
	$\mid$	$\texttt{true} \mid \texttt{false}$	Wahrheitswerte
	$\mid$	$()$	Leeres Tupel
	$\mid$	(v_1, v_2)	Paar von Werten
	$\mid$	$\{l_1{=}v_1, \ldots, l_n{=}v_n\}$	Datensatz von Werten

6.6.2 Auswertung mit Call-by-Value

Einzelschritt-Auswertung.

$$t \longrightarrow t'$$

Lambda-Kalkül Berechnung.

$$\frac{}{(\lambda x{:}T\ t)\ v \longrightarrow [v/x]t}\ \text{E-Beta}$$

Kongruenz.

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ \text{E-AppTT} \qquad \frac{t \longrightarrow t'}{v\ t \longrightarrow v\ t'}\ \text{E-AppVT}$$

Ganzzahlen Berechnung.

$$\frac{i_3 = i_1 + i_2}{i_1 + i_2 \longrightarrow i_3}\ \text{E-PlusVV} \qquad \frac{i_3 = i_1 - i_2}{i_1 - i_2 \longrightarrow i_3}\ \text{E-MinusVV}$$

$$\frac{}{\texttt{iszero}\ 0 \longrightarrow \texttt{true}}\ \text{E-IsZeroTrue} \qquad \frac{i \neq 0}{\texttt{iszero}\ i \longrightarrow \texttt{false}}\ \text{E-IsZeroFalse}$$

Kongruenz.

$$\frac{t_1 \longrightarrow t_1'}{t_1 + t_2 \longrightarrow t_1' + t_2}\ \text{E-PlusTT} \qquad \frac{t_1 \longrightarrow t_1'}{t_1 - t_2 \longrightarrow t_1' - t_2}\ \text{E-MinusTT}$$

$$\frac{t \longrightarrow t'}{i + t \longrightarrow i + t'}\ \text{E-PlusVT} \qquad \frac{t \longrightarrow t'}{i - t \longrightarrow i - t'}\ \text{E-MinusVT}$$

$$\frac{t \longrightarrow t'}{\texttt{iszero}\ t \longrightarrow \texttt{iszero}\ t'}\ \text{E-IsZero}$$

Wahrheitswerte Berechnung.

$$\frac{}{\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2}\ \text{E-IfTrue}$$

$$\frac{}{\texttt{if false then } t_2 \texttt{ else } t_3 \longrightarrow t_3}\ \text{E-IfFalse}$$

Kongruenz.

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3}\ \text{E-If}$$

Leeres Tupel Keine Auswertungsregeln.

Paare Berechnung.

$$\frac{}{(v_1, v_2).1 \longrightarrow v_1}\ \text{E-PairBeta}_1 \qquad \frac{}{(v_1, v_2).2 \longrightarrow v_2}\ \text{E-PairBeta}_2$$

Kongruenz.

$$\frac{t_1 \longrightarrow t_1'}{(t_1, t_2) \longrightarrow (t_1', t_2)}\ \text{E-PairTT} \qquad \frac{t \longrightarrow t'}{(v, t) \longrightarrow (v, t')}\ \text{E-PairVT}$$

$$\frac{t \longrightarrow t'}{t.1 \longrightarrow t'.1}\ \text{E-Proj}_1 \qquad \frac{t \longrightarrow t'}{t.2 \longrightarrow t'.2}\ \text{E-Proj}_2$$

Datensätze Berechnung.

$$\frac{}{\{\overrightarrow{l=v}\}.l_i \longrightarrow v_i}\ \text{E-RecordBeta}$$

Kongruenz.

$$\frac{t \longrightarrow t'}{\{\overrightarrow{l=v}, l'=t, \overrightarrow{l''=t''}\} \longrightarrow \{\overrightarrow{l=v}, l'=t', \overrightarrow{l''=t''}\}}\ \text{E-Record}$$

$$\frac{t \longrightarrow t'}{t.l \longrightarrow t'.l}\ \text{E-Proj}$$

Lokale Variablen Berechnung.

$$\frac{}{\texttt{let } x=v \texttt{ in } t \longrightarrow [v/x]t}\ \text{E-LetBeta}$$

Kongruenz.

$$\frac{t_1 \longrightarrow t_1'}{\texttt{let } x=t_1 \texttt{ in } t_2 \longrightarrow \texttt{let } x=t_1' \texttt{ in } t_2}\ \text{E-Let}$$

Rekursion Berechnung.

$$\frac{}{\texttt{fix}\, f{:}T\ t \longrightarrow [\texttt{fix}\, f{:}T\ t/f]t}\ \text{E-Fix}$$

6.6.3 Typisierung

Kontexte.

$$\begin{array}{rcll} \Gamma & ::= & \cdot & \text{leerer Kontext} \\ & | & \Gamma, x{:}T & \text{erweiterter Kontext} \end{array}$$

Typisierungsrelation.

$$\Gamma \vdash t : T$$

Lambda-Kalkül

$$\frac{(x{:}T) \in \Gamma}{\Gamma \vdash x : T}\ \text{T-Var}$$

Einführung.

$$\frac{\Gamma, x{:}S \vdash t : T}{\Gamma \vdash \lambda x{:}S\ t\ : S \to T}\ \text{T-Lam}$$

Beseitigung.

$$\frac{\Gamma \vdash t : S \to T \qquad \Gamma \vdash s : S}{\Gamma \vdash t\ s\ :\ T}\ \text{T-App}$$

Ganzzahlen Einführung.

$$\frac{}{\Gamma \vdash i : \texttt{Int}}\ \text{T-Int}$$

Beseitigung.

$$\frac{\Gamma \vdash t_1 : \texttt{Int} \qquad \Gamma \vdash t_2 : \texttt{Int}}{\Gamma \vdash t_1 + t_2 : \texttt{Int}}\ \text{T-Plus} \qquad \frac{\Gamma \vdash t_1 : \texttt{Int} \qquad \Gamma \vdash t_2 : \texttt{Int}}{\Gamma \vdash t_1 - t_2 : \texttt{Int}}\ \text{T-Minus}$$

$$\frac{\Gamma \vdash t : \texttt{Int}}{\Gamma \vdash \texttt{iszero}\ t : \texttt{Bool}}\ \text{T-IsZero}$$

Wahrheitswerte Einführung.

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}}\ \text{T-True} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}}\ \text{T-False}$$

Beseitigung.

$$\frac{\Gamma \vdash t_1 : \texttt{Bool} \qquad \Gamma \vdash t_2 : T \qquad \Gamma \vdash t_3 : T}{\Gamma \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3\ :\ T}\ \text{T-If}$$

Leeres Tupel Einführung.

$$\frac{}{\Gamma \vdash () : \texttt{Unit}}\ \text{T-Unit}$$

Keine Beseitigung.

Paare Einführung.

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1, t_2) : T_1 \times T_2}\ \text{T-Pair}$$

Beseitigung.

$$\frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash t.1 : T_1}\ \text{T-Proj}_1 \qquad \frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash t.2 : T_2}\ \text{T-Proj}_2$$

Datensätze Einführung.

$$\frac{\Gamma \vdash t_i : T_i \ \text{ für } \ i = 1 \ldots n}{\Gamma \vdash \{\overrightarrow{l{=}t}\} : \{\overrightarrow{l{:}T}\}}\ \text{T-Record}$$

Beseitigung.

$$\frac{\Gamma \vdash t : \{\overrightarrow{l{:}T}\}}{\Gamma \vdash t.l_i : T_i}\ \text{T-Proj}$$

Lokale Variablen

$$\frac{\Gamma \vdash s : S \qquad \Gamma, x{:}S \vdash t : T}{\Gamma \vdash \texttt{let}\ x{=}s\ \texttt{in}\ t\ :\ T}\ \text{T-Let}$$

Rekursion

$$\frac{\Gamma, f{:}T \vdash t : T}{\Gamma \vdash \texttt{fix}\, f{:}T\ t\ :\ T}\ \text{T-Fix}$$

Kapitel 7

Referenzen

Bislang wurden nur rein funktionale Programmkonstrukte behandelt. Der Wert v eines Ausdrucks t war allein durch die Gestalt von t bestimmt, d.h. es gab immer höchstens ein v mit $t \longrightarrow^* v$. Dies ändert sich, wenn man *Seiteneffekte* zur Programmiersprache hinzunimmt. Der Wert eines Ausdrucks t is dann abhängig von einem Zustand μ der Maschine, der nicht durch t allein determiniert ist. Zudem kann die Auswertung eines Ausdrucks eine Änderung des Zustandes verursachen. Beispiele für Zustand in einem realen Rechnersystem sind Register-, Hauptspeicher- und Festplatteninhalt, der Zustand von Peripheriegeräten wie z.B. der Tastatur. Wir behandeln im folgenden nur den Hauptspeicher.

[Hier wird der Stoff aus Pierce [Pie02], Kapitel 13, behandelt. Die folgende Mitschrift ist von Cyril Bitterich.]

Operationen

- Allozierung + Initialisierung einer neuen Zelle
  ref v
- Auslesen einer Zelle !v
- Neubeschreiben einer Zelle $v_1 := v_2$

Beispiel 7.1 *C/JAVA*

```
int bla() {
  int x = 5;
  x = x+1;
  return x;
}
```

In λ^{Ref}*:*

$bla = \lambda u : Unit$

let x = ref 5

in (x := !x+1;

!x)

Auswertung bla()
ref 5 $\longrightarrow$ l (l ist neue Adresse "location")
!x $\longrightarrow$!l $\longrightarrow$ 5
5 +1 $\longrightarrow$ 6
x:=6 $\longrightarrow$ ()
!x $\longrightarrow$ 6

Beschreibung der Speicherbelegung

μ Zustand des Speichers, bildet Adressen auf Werte ab d.h. !l $\longrightarrow \mu(l)$

Operationen auf den Speicher

- Allokieren: $(\mu, l \mapsto v)$: neue Adresse l, Initialisierung mit v
- Schreiben: $[l \mapsto v]\mu$: v neuer Inhalt von l
- Lesen: $\mu(l) = v$: v Inhalt von l
- Leerer Speicher: $\emptyset$

Neue Auswertungsrelation

$$t \mid \mu \longrightarrow t' \mid \mu'$$

Übung 7.1 $ref5 \mid \mu_0 \longrightarrow l \mid \underbrace{(\mu_0, l \mapsto 5)}_{\mu_1} \qquad \mu_1 \geq \mu_0$

$!l \mid \mu_1 \longrightarrow \underbrace{\mu_1(l)}_{5} \mid \mu_1$

$l := 6 \mid \mu_1 \longrightarrow () \mid \underbrace{[l \mapsto 6]\mu_1}_{\mu_2} \qquad \mu_2 \not\geq \mu_1$

$!l \mid \mu_2 \longrightarrow \underbrace{\mu_2(l)}_{6} \mid \mu_2$

Terme

t:: = ...
| ref t
| !t
| $t_1 := t_2$
| l (nur intern)

Werte

v :: = ...
| l

Auswertungsregeln

- Erweitere bisherige Regeln im Speicher μ
  z.B.

$$\frac{}{(\lambda x : T.t)v \mid \mu \longrightarrow [v/x]t \mid \mu} E - Beta$$

  z.B.

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{t_1 t_2 \mid \mu \longrightarrow t_1' t_2 \mid \mu'}$$

- Neue Regeln:
  - Allokation

$$\frac{}{ref\ \ v \mid \mu \longrightarrow l \mid (\mu, m \mapsto v)} E - RefV$$

$$\frac{t \mid \mu \longrightarrow t' \mid \mu'}{ref\ \ t \mid \mu \longrightarrow ref\ \ t' \mid \mu'} E - Ref$$

  - Auslesen

$$\frac{}{!l \mid \mu \longrightarrow \mu(l) \mid \mu} E - DerefV$$

$$\frac{t \mid \mu \longrightarrow t' \mid \mu'}{!t \mid \mu \longrightarrow t' \mid \mu'} E - Deref$$

  - Zuweisung

$$\frac{}{l := v \mid \mu \longrightarrow () \mid [l \mapsto]\mu} E - AssignVV$$

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{t_1 := t_2 \mu \longrightarrow t_1' := t_2 \mid \mu} E - AssignTT$$

$$\frac{t \mid \mu \longrightarrow t' \mid \mu'}{l := t \mid \mu \longrightarrow l := t' \mid \mu'} E - AssignVT$$

Typisierung

T :: = ...
    | Ref T

Regeln

$$\frac{\Gamma \mid \Sigma \vdash t : T}{\Gamma \mid \Sigma \vdash ref\ t : Ref\ T} T - Ref$$

$$\frac{\Gamma \mid \Sigma \vdash t : Ref\ T}{\Gamma \mid \Sigma \vdash !t : T} T - Deref$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : Ref\ T \qquad \Gamma \mid \Sigma \vdash t_2 : T}{\Gamma \mid \Sigma \vdash t_1 := t_2 : Unit} T - Assign$$

$$\frac{\Sigma(l) = T}{\Gamma \mid \Sigma \vdash l :} T - Loc$$

Übung 7.2 *let x = ref 5 in x:= !x+1*

$$\dfrac{\dfrac{}{5:Int}}{ref\ 5:Ref\ Int} \dfrac{\dfrac{}{x\ldots\vdash x:Ref\ Int}T-Var \quad \dfrac{\dfrac{\dfrac{}{x:\ldots\vdash x:Ref\ Int}T-Var}{x:\ldots\vdash !x:Int}T-Deref \quad \ldots\vdash 1:Int}{x:\ldots\vdash !x+1:Int}T-Assign}{X:Ref\ Int\vdash x:=!x+1:Unit}}{\vdash let\ x = ref 5\ in x :=!x+1 : Unit}$$

Durch Auswertung treten auch Terme l aud, Was ist ihr Typ?
z.B.: $ref 5 \longrightarrow l$

Lösung Typisiere den Speicher μ. Merke für jede Adresse l den Typ der zugehörigen Zelle.

Speichertyp

$\Sigma ::= \emptyset$
$\quad | \ \Sigma, l : T$

Übung 7.3 $\dfrac{\dfrac{5:Int}{\cdot|\emptyset\vdash ref\ 5:Ref\ Int}T-Ref}{\cdot|l:Int\vdash l:Ref\ Int}T-Loc$ $\qquad ref\ 5 \mid \emptyset \longrightarrow l \mid (\emptyset, l \mapsto 5)$

Satz 7.1
Wenn

$$\Gamma \mid \Sigma \vdash t : T$$
$$t \mid \mu \longrightarrow t' \mid \mu'$$
$$\Gamma \vdash \mu : \Sigma$$

dann gibt es ein Σ' *mit* $\Sigma' \ni \Sigma$ *und*

$$\Gamma \mid \Sigma' \vdash t' : T$$
$$\Gamma \vdash \mu' : \Sigma'$$

Definition 7.1 $\Gamma \vdash \mu : \Sigma$

$$\frac{}{\Gamma \vdash \emptyset : \emptyset}T - Empty$$

$$\frac{\Gamma \vdash \mu : \Sigma \qquad \Gamma \mid \Sigma \vdash v : T}{\Gamma \vdash (\mu, l \mapsto v) : (\Sigma, l : T)}T - Alloc$$

$$\frac{\Gamma \vdash \mu : \Sigma \qquad \Gamma \mid \Sigma \vdash v : \Sigma(l)}{\Gamma \vdash [l \mapsto v]\mu : \Sigma}T - Write$$

Kapitel 8

Untertypen

Der Begriff des *Untertyps*[1] ist essentiell für objekt-orientierte Programmierung, findet sich aber schon in einfachen imperativen Programmiersprachen wie z.B. PASCAL. Dort gibt es zwei Divisionsoperatoren: `div` für Ganzzahlen und "/" für Fließkommazahlen. In unserer Schreibweise:

$$\frac{i : \texttt{Int} \qquad j : \texttt{Int}}{i \ \texttt{div} \ j : \texttt{Int}} \qquad \frac{a : \texttt{Float} \qquad b : \texttt{Int}}{a/b : \texttt{Float}}$$

PASCAL erlaubt die Anwendung von "/" auch auf Ganzzahlen; der Compiler führt intern eine Konvertierung nach Fließkomma durch. Eine Typherleitung für den Term i/j sieht dann so aus:

$$\dfrac{\dfrac{i : \texttt{Int}}{i : \texttt{Float}}\ \text{T-Sub} \qquad \dfrac{j : \texttt{Int}}{j : \texttt{Float}}\ \text{T-Sub}}{i/j : \texttt{Float}}\ \text{T-DivFloat}$$

An den Stellen T-Sub wird eine Typkonversion durchgeführt. Die Konversion ist zulässig, weil `Int` als ein Untertyp von `Float` angesehen werden kann, d.h. jede Ganzzahl kann als Fließkommazahl interpretiert werden. Die zentrale Frage im Zusammenhang mit Untertypen ist: Welche Konversionen sind zulässig und sollen vom Typ-Prüfer bzw. Übersetzer automatisch durchgeführt werden?

8.1 Subsumption

Aus der Mathematik ist bekannt, dass $\mathbb{Z} \subseteq \mathbb{R}$. Diese semantische Relation können wir auf der Typebene reflektieren durch die Untertyp-Relation

$$\texttt{Int} <: \texttt{Float}$$

[1] auch *Teiltyp*.

Konversion ist möglich von einem Term eines Typs S in dessen Obertyp T, was wir durch die *Subsumptions-Regel* erfassen:

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T}\ \text{T-Sub}$$

Prinzip I (Teilmenge) Ein Typ S kann als Untertyp von T aufgefasst werden, wenn die Menge der Bewohner von S als eine Teilmenge der Bewohner von Typ T angesehen werden kann.

Prinzip II (Ersetzung) Ein Typ S kann als Untertyp von T aufgefasst werden, wenn an jeder Stelle, wo ein Term vom Typ T erwartet wird, ein Term vom Typ S eingesetzt werden kann ohne die Typsicherheit zu gefährden.

8.2 Regeln der Untertyp-Beziehung

[Hier fehlt die Motivation der einzelnen Typregeln]

Definition 8.1 (Deklarative Untertyp-Beziehung) *Die Relation $S <: T$ ist gegeben durch die folgenden Inferenzregeln:*

$$\frac{}{T <: T}\ \text{S-Refl} \qquad \frac{R <: S \qquad S <: T}{R <: T}\ \text{S-Trans}$$

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}\ \text{S-Arrow} \qquad \frac{S_1 <: T_1 \qquad S_2 <: T_2}{S_1 \times S_2 <: T_1 \times T_2}\ \text{S-Prod}$$

$$\frac{}{\{\overrightarrow{l : T}, \overrightarrow{k : R}\} <: \{\overrightarrow{l : T}\}}\ \text{S-RWidth} \qquad \frac{S_i <: T_i \quad \textit{für } i = 1..|\overrightarrow{l}|}{\{\overrightarrow{l : S}\} <: \{\overrightarrow{l : T}\}}\ \text{S-RDepth}$$

8.3 Entscheidung der Untertyp-Beziehung

Satz 8.1 *Die Untertyp-Relation ist entscheidbar.*

Zum Beweis dieses Satzes müssen wir einen Algorithmus angeben, der für beliebige Typen S und T entscheidet ob $S <: T$.

Aus den bisherigen Untertyp-Regeln lässt sich nicht direkt ein Algorithmus ableiten. Insbesondere die Transitivitäts-Regel bereitet Kopfzerbrechen, denn sie ist nicht Syntax-gerichtet. Um zu entscheiden ob $R <: T$, empfiehlt uns die Transitivitäts-Regel, einen Typ S zu finden und dann $R <: S$ und $S <: T$ zu testen. Da es unendlich viele Typen S gibt, kann unsere Suche endlos dauern.

Wir entwickeln im folgenden neue Regeln für Untertypen, die ohne eine Transitivitäts-Regel auskommen, deterministisch und Syntax-gerichtet sind.

Definition 8.2 (Algorithmische Untertyp-Beziehung) *Die Relation $S \lhd: T$ ist gegeben durch die folgenden Inferenzregeln:*

$$\frac{}{A \lhd: A}\ \text{SA-Refl} \quad \textit{für Grundtypen } A$$

$$\frac{T_1 \lhd: S_1 \qquad S_2 \lhd: T_2}{S_1 \to S_2 \lhd: T_1 \to T_2}\ \text{SA-Arrow} \qquad \frac{S_1 \lhd: T_1 \qquad S_2 \lhd: T_2}{S_1 \times S_2 \lhd: T_1 \times T_2}\ \text{SA-Prod}$$

$$\frac{S_i \lhd: T_i \quad \textit{für } i = 1..|\overrightarrow{l}|}{\{\overrightarrow{l : S}, \overrightarrow{k : R}\} \lhd: \{\overrightarrow{l : T}\}}\ \text{SA-Record}$$

Aus diesen Regeln lässt sich nun direkt ein Algorithmus ablesen. Bleibt zu zeigen, dass die beiden Untertyp-Beziehungen äquivalent sind. Die eine Richtung ist einfach: Jede Untertyp-Beziehung $S \lhd: T$, die mit den algorithmischen Regeln hergeleitet wurde, kann auch mit den deklarativen gezeigt werden. Hierfür muss man nur einsehen, dass sich jede Anwendung der Regel SA-Record durch eine Kombination von S-RWidth, S-RDepth, S-Trans und S-Refl ausdrücken lässt.

Lemma 8.1 (Algorithmische Untertyp-Relation ist korrekt) *Wenn $S \lhd: T$, dann $S <: T$.*

Beweis. Induktion nach $S \lhd: T$. □

Die andere Richtung ist etwas schwieriger. Hier machen Anwendungen der Regeln S-Refl und S-Trans Umstände. Wir zeigen also zuerst, dass die algorithmische Relation reflexiv und transitiv ist. Die Beweise dieser Aussagen geben uns Algorithmen, wie wir Vorkommen von S-Refl und S-Trans eliminieren können.

Lemma 8.2 (Reflexivität) *Für alle Typen T gilt $T \lhd: T$.*

Beweis. Durch Induktion nach T. Wir demonstrieren einen Fall:

Fall $T = T_1 \to T_2$. Nach Induktionsvoraussetzung gilt $T_1 \lhd: T_1$ und $T_2 \lhd: T_2$. Mit Regel SA-Arrow folgt damit $T_1 \to T_2 \lhd: T_1 \to T_2$. □

Lemma 8.3 (Transitivität) *Wenn $R \lhd: S$ und $S \lhd: T$, dann $R \lhd: T$.*

Beweis. Durch Induktion nach $R \lhd: S$ mit Fallunterscheidung über $S \lhd: T$. Zum Beispiel:

Fall $R = R_1 \to R_2$, $S = S_1 \to S_2$ und

$$\frac{S_1 \lhd: R_1 \qquad R_2 \lhd: S_2}{R_1 \to R_2 \lhd: S_1 \to S_2}\text{SA-Arrow} \qquad S_1 \to S_2 \lhd: T$$

Die einzige Möglichkeit ist $T = T_1 \to T_2$ mit $T_1 \lhd: S_1$ und $S_2 \lhd: T_2$. Induktionsvoraussetzung liefert $T_1 \lhd: R_1$ und $R_2 \lhd: T_2$. Mit Regel SA-Arrow folgt also $R_1 \to R_2 \lhd: T_1 \to T_2$. □

Lemma 8.4 (Algorithmische Untertyp-Relation ist vollständig) *Wenn* $S <: T$*, dann* $S <: T$*.*

Beweis. Durch Induktion nach $S <: T$, unter Verwendung der Lemmata über Reflexivität und Transitivität. □

Satz 8.1 folgt nun leicht. Die algorithmische Untertyp-Relation lässt sich direkt in ein Programm umwandeln, dass für gegebene S, T entscheidet, ob $S \lhd: T$. Nach Lemmata 8.1 und 8.4 ist dies äquivalent zu $S <: T$.

Kapitel 9

Objekt-orientierte Programmierung

Objekt-orientierung ist ein *Paradigma* bzw. ein Programmier*stil*. Manche Programmiersprachen stellen Sprachkonstrukte zur Verfügung, die objekt-orientiertes Programmieren begüngstigen; allerdings kann man auch in funktionalen Sprachen objekt-orientiert Programmieren (siehe Abelson und Sussman [ASS91]). In realen objekt-orientierten Sprachen vereinigt das Konzept der *Klasse* eine Vielzahl von Aspekten objekt-orientierter Programmierung auf sich. Wie Pierce [Pie02] beschränken wir uns auf fünf wesentliche Kennzeichen.

Polymorphie Ein Interface kann mehrere Implementationen besitzen. Dynamic dispatch: Objekt entscheidet, welcher Code ausgeführt wird.

Kapselung Nur Methoden des Objektes können auf Instanzenvariablen direkt zugreifen.

Subtyping Ein Objekt mit mehr Funktionalität (Methoden) kann an jeder Stelle verwendet werden, an der ein Objekt mit weniger Funktionalität erwartet wird.

Vererbung Verhalten und Implementation kann an Subklasse weitergegeben werden.

Späte Bindung (`self`, `this`) Vituelle Methoden.

Im folgenden zeigen wir am Beispiel eines Zählers (`counter`), wie wir die obigen Konzepte in unserer funktionalen Sprache simulieren können. Der Programmcode ist direkt von Pierce [Pie02] übernommen.

9.1 Objekte, Methoden

Interface.

```
Counter = { get: Uni -> Int , inc: Unit -> Unit }
```

Objekt.

```
c = let x = ref 0 in
        { get = \lambda _:Unit. !x,
          inc = \lambda _:Unit. x := !x + 1 }
c : Counter
```

Methodenaufruf.

```
c.get(), c.inc()
inc3 = \lambda c: Counter. (c.inc(); c.inc(); c.inc())
```

9.2 Konstruktor

Objektkonstruktoren bezeichnen wir mit `new...`

```
newCounter : Unit -> Counter
newCounter = \lambda _:Unit. let x = ref 0 in
        { get = \lambda _:Unit. !x,
          inc = \lambda _:Unit. x := !x + 1 }
```

9.3 Subtyping

```
ResetCounter={get...; inc..., reset:Unit -> Unit}
newResetCounter = \lambda _:Unit. let x = ref 0 in
        { get   = \lambda _:Unit. !x,
          inc   = \lambda _:Unit. x := !x + 1,
          reset = \lambda _:Unit. x := 0 }
```

9.4 Bündelung der Instanzen-Variablen

```
CounterRep = { x: Ref Int }
newCounter = \lambda _:Unit.
    let r = {x=ref 0 } in
        { get = \lambda _:Unit. !(r.x),
          inc = \lambda _:Unit. r.x := !(r.x) + 1 }
```

9.5 Klassen

Definition 9.1 *Interface*
Ein Datensatz-Typ, in dem jeder Eintrag ein Funktionstyp ist.

Definition 9.2 *Objekt*
Ein Term vom Typ "Interface".

Definition 9.3 *Methode*
Ein Bestandteil eines Objektes.

Definition 9.4 *Klasse*
Die Sammlung der Objekte mit identischer Implementation der Methoden.

```
CounterRep = { x: Ref Int}
counterClass: CounterRep -> Counter
counterClass
     = \lambda r: CounterRep.
        {get = \lambda _ : Unit. !(r.x),
         inc = \lambda _ : Unit. r.x := !(r.x) + 1 }
```

Vererbung

```
ResetCounter = { get..., inc..., reset: Unit ->Unit }

resetCounterClass : CounterRep -> ResetCounter
resetCounterClass = \lambda r: CounterRep.
     let super = counterClass r in
         { get = super.get,
           inc = super.inc,
           reset = \lambda _: Unit. r.x := 0 }

newResetCounter = \lambda _: Unit. let r = {x = ref 0} in resetCounterClass r
```

9.6 Zusätzliche Instanzenvariablen

```
BackupCounterRep = { x: Ref Int, b: Ref Int}
BackupCounter = { get..., inc..., reset..., backup: Unit -> Unit }

backupCounterClass : BackupCounterRep -> BackupCounter
backupCounterClass = \lambda r : BackupCounterRep.
     let super = counterClass r in
         { get = super.get,
           inc = super.inc,
           reset = \lambda _: Unit. r.x := !(r,b),
           backup = \lambda _: Unit. r.b := !(r.x) }
```

9.7 `self` (ohne dynamische Bindung)

Ziel: Eine Methode kann andere Methoden des gleichen Objektes aufrufen. (Rekursion)

```
SetCounter = { get : Unit -> Unit,
               set : Int -> Unit,
```

```
               inc : Unit -> Unit }

setCounterClass : CounterRep -> SetCounter
setCounterClass = \lambda r : CounterRep.
      fix self : SetCounter.
      { get = \lambda _ : Unit. !(r.x),
        set = \lambda i : Int. r.x := i,
        inc = \lambda _ : Unit.
              self.set (self.get()+1) }

newSetCounter : Unit -> SetCounter
newSetCounter = \lambda _:Unit
      let r = {x = ref 0 } in setCounterClass r
```

9.8 `self` (mit dynamischer Bindung)

```
SetCounter = { get : Unit -> Unit,
               set : Int -> Unit,
               inc : Unit -> Unit }

setCounterClass : CounterRep -> SetCounter -> SetCounter
setCounterClass = \lambda r : CounterRep.
      \lambda self : SetCounter.
      { get = \lambda _ : Unit. !(r.x),
        set = \lambda i : Int. r.x := i,
        inc = \lambda _ : Unit.
              self.set (self.get()+1) }

newSetCounter : Unit -> SetCounter
newSetCounter = \lambda _:Unit
      let r = {x = ref 0 } in
        fix self : SetCounter. setCounterClass r self
```

Problem: Mit cbv-Auswertung hängt sich der Rechner beim Instantiieren der Klasse mit `newSetCounter` auf:

```
fix self : SetCounter. setCounterClass r self
--> setCounterClass r (fix self... setCounterClass r self)
--> setCounterClass r (setCounterClass r (fix self... setCounterClass r self ))
--> ...
```

Abhilfe:

1. Man verwendet eine “lazy” Sprache, die Funktionsargumente nur bei Bedarf auswertet.

2. Man kodiert verzögerte Auswertung per Hand durch "dummy"-Abstraktionen `\lambda _:Unit` und -Applikationen `()`.

```
self : Unit -> SetCounter
self () : SetCounter

setCounterClass : CounterRep -> (Unit -> SetCounter) -> SetCounter
setCounterClass = \lambda r : CounterRep.
      \lambda self : Unit -> SetCounter.
      { get = \lambda _ : Unit. !(r.x),
        set = \lambda i : Int. r.x := i,
        inc = \lambda _ : Unit. self().set (self().get() + 1) }

newSetCounter = \lambda _:Unit
      let r = {x = ref 0 } in
        fix self : Unit -> SetCounter.
          \lambda _ : Unit. setCounterClass r self
```

Beispiel: "instrumented Counter"; zählt die Anzahl der Schreibzugriffe mit `set`. Das in der Oberklasse `setCounterClass` mit später Bindung implementierte `inc` kann unverändert übernommen werden, obwohl `set` überschrieben wird.

Achtung! Diese Implementation ist in einer strikten cbv-Sprache inkorrekt und muss nach dem Vorbild von `setCounterClass` umgearbeitet werden.

```
InstrCounterRep = { x : RefInt, a: RefInt }
InstrCounter = { get..., set..., inc..., accesses: Unit -> Int }

instrCounterClass = InstrCounterRep -> InstrCounter -> InstrCounter
instrCounterClass =
    \lambda r : InstrCounterRep \lambda self : InstrCounter.
      let super = setCounterClass r self
      in { get = super.get ,
           set = \lambda i : Int. ( super.set(i) ; r.a := !(r.a)+1),
           inc = super.inc ,
           accesses = \lambda _ : Unit. !(r.a) }
```

Kapitel 10

Featherweight Java

[Hier wird der Stoff aus Pierce [Pie02], Kapitel 19, behandelt.]

Rohfassung vom 1. Oktober 2002

Literaturverzeichnis

[ASS91] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Struktur und Interpretation von Computerprogrammen*. Springer-Verlag, 1991.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq SRC, 1998.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Zus99] Horst Zuse. Geschichte der Programmiersprachen. Technical Report 1999-1, Technische Universität Berlin, 1999.