1. Typen

Ein Typsystem ist ein praktikables, syntaktisches Verfahren, mit dem man die Abwesenheit gewisser Laufzeit-Eigenschaften eines Programms beweisen kann, indem man Programmbestandteile danach klassifiziert, welche Art Werte sie berechnen.

Strenges Typisieren verhindert nicht nur Fehler, sondern trägt (bereits in der Entwurfsphase) zur Klarheit der Software bei.

Typsysteme beschleunigen Programme, d.h. der Compiler kann Typinformationen benutzen, um effizienten (d.h. spezialisierten) Code zu generieren.

Bei Entwurf und Anwendung eines Typsystems balanciert man zwischen

- Ausdrucksstärke: alle „sinnvollen“ Funktionen, Datentypen, und Abstraktionen davon sollen auch typisierbar sein (aber alle „unsinnigen“ nicht)
- Effektivität: wichtige Fragen (hat der Term x den Typ t?) sollen effektiv entscheidbar sein.

1.1 Typsicherheit:

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

1.2 Typprüfung:

Es gibt zwei Arten der Typprüfung: die statische und die dynamische Typprüfung.
Wird ein Programm auf syntaktische Korrektheit überprüft, bevor es ausgeführt wird, spricht man von einer „statischen Typprüfung“. Programmiersprachen, die die „statische Typprüfung“ durchführen sind u.a. SML, Haskel, C++ und Java.
Erfolgt die Typprüfung zur Laufzeit, d.h. ein Programm wird ausgeführt und die Ausdrücke werden ausgewertet, dann spricht man von der sog. „dynamischen Typprüfung“. Dieses System wird in folgenden Programmiersprachen angewendet: Lisp, Scheme, Pascal und Perl.

2. Einführung in Typen

t ::=	Term
true	Konstante true
false	Konstante false
if t then t else t	Konditional
0	Konstante
succ t	Successor
pred t	Predecessor
iszero t	Zero Test

succ erhöht den Wert immer um eins und pred zieht immer eins des angegebenen Wertes ab.
Beispiel: succ(0) = 1, succ(succ(0)) = 2
pred(succ(succ(0))) = 1
Damit kann man die komplette Menge der natürlichen Zahlen „Nat“ darstellen.

2.1 Unterschied zwischen „numeric value“ und „boolean“:

numeric value	boolean
Typ: Nat	Typ: Bool
z.B. pred, succ, iszero	z.B.: true, false

2.2 Beispiele:

Term t ist Element von T (geschrieben: "t : T")
pred(succ(pred(succ0))) (Typ Nat)
if true then false else true (Typ Bool)

Kein Typ:
true + 6

3. Typisierungsrelation

3.1 Regeln

3.1.1 Regeln für booleans:

Regel

T::=	Typ
Bool	Typ boolean
$\text{true} : \text{Bool}$	T-True
$\text{false} : \text{Bool}$	T-False
$\dfrac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$	T-If

3.1.2 Regeln für numbers:

Regel

T::=	Typ
Nat	Typ für natürliche Zahlen
$0 : \text{Nat}$	T-Zero
$\dfrac{t_1 : \text{Nat}}{\text{succ } t_1 : \text{Nat}}$	T-Succ
$\dfrac{t_1 : \text{Nat}}{\text{pred } t_1 : \text{Nat}}$	T-Pred
$\dfrac{t_1 : \text{Nat}}{\text{iszero } t_1 : \text{Bool}}$	T-IsZero

4. Typ-System

Satz von Pierce: "well-typed terms do not go wrong".

Ein Typsystem ist eine Programmiersprache zusammen mit einer Typisierungsrelation.

Damit ein Typ-System sicher ist, gelten folgende Regeln:

Regel

Sicherheit = Fortschritt + Typerhaltung

Fortschritt (Progress): Wenn ein Term kein Wert und wohlgetypt ist, dann kann man auswerten.

Typerhaltung (Preservation): Wenn ein Term wohlgetypt ist, dann behält er nach der Auswertung auch noch den selben Typ.

Lemma

1. Falls v ein Wert vom Typ Bool ist, dann ergibt v entweder true oder false.

2. Falls v ein Wert vom Typ Nat ist, dann ergibt v einen numerischen Wert.

4.1 Fortschritt (Progress):

Falls $t : T$, dann $t = v$ oder $t \rightarrow t'$
Beweis: Induktion über $t : T$

Fall T-If

$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$ $\quad t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T$

Fall T-Succ

$t = \text{succ } t_1$ $\quad t_1 : \text{Nat}$

Fall T-Pred

$t = \text{pred } t_1$ $\quad t_1 : \text{Nat}$

Fall T-IsZero

$t = \text{iszero } t_1$ $\quad t_1 : \text{Nat}$

4.2 Typerhaltung (Preservation):

Falls $t : T$ und $t \rightarrow t'$, dann $t' : T$
Beweis: Induktion über $t : T$

Fall T-True

$t = true$	$T = Bool$

Fall T-If

$t = if\ t_1\ then\ t_2\ else\ t_3$	$t_1 : Bool \quad t_2 : T \quad t_3 : T$

Fall T-Zero

$t = 0$	$T = Nat$

5. Einfach getypte Lambda-Kalkül

5.1 Typisierung:

Typisierungsrelation: $\Gamma \vdash t : T$

Regel

Variablenregel:	$\dfrac{x : T \in \Gamma}{\Gamma \vdash x : T}$	T-Var
Abstraktionsregel:	$\dfrac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}\ T_1.t_2 : T_1 \rightarrow T_2}$	T-Abs
Anwendungsregel:	$\dfrac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}$	T-App

5.2 Beispiel:

$$\dfrac{\dfrac{\dfrac{x : \text{Bool} \in x : \text{Bool}}{x : \text{Bool} \quad x : \text{Bool}}\, T-\text{Var}}{\lambda x : \text{Bool}.x : \text{Bool} \to \text{Bool}}\, T-\text{Abs} \qquad \dfrac{}{\text{true} : \text{Bool}}\, T-\text{True}}{(\lambda x : \text{Bool}.x)\text{true} : \text{Bool}}\, T-\text{App}$$

Lemma

1. Falls v ein Wert vom Typ Bool ist, dann ergibt v entweder true oder false.

2. Falls v ein Wert vom Typ $T_1 \to T_2$, dann ist $v = \lambda x : T_1.t_2$

5.3 Fortschritt (Progress):

Angenommen t ist ein geschlossener, wohlgetypter Term ($\vdash t : T$). Dann ist t entweder ein Wert oder es existiert ein t´ mit $t \to t'$.

5.4 Typerhaltung (Preservation):

Falls $\Gamma, x : S \vdash t : T$ und $\Gamma \vdash s : S$, dann $\Gamma \vdash [x \to s]t : T$.
Beweis erfolgt durch Induktion über die Anweisung $\Gamma, x : S \vdash t : T$.

Fall T-Var

$t = z \quad \text{mit } z : T \in (\Gamma, x : S)$

Fall T-Abs

$t = \lambda y : T_2.t_1 \qquad T = T_2 \to T_1 \qquad \Gamma, x : S, y : T_2 \vdash t_1 : T_1$

Fall T-True

$t = \text{true} \qquad T = \text{Bool}$

Fall T-False

$t = \text{false} \qquad T = \text{Bool}$

6. Curry – Howard Correspondance

Was Curry und Howard entdeckten, war das jeder Beweis ein stark rechenbetontes Gefühl hat. Dies wird in folgender Tabelle gezeigt:

Logik	Programmiersprache
Propositionen	Typen
Proposition $P \supset Q$	Typ $P \rightarrow Q$
Proposition $P \wedge Q$	Typ $P \times Q$
Prüfen der Proposition P	Term t vom Typ P
Proposition P ist prüfbar	Typ P ist „bewohnt“