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

3.1.2 Regeln für numbers:

Regel

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

3.2 Beispiel:

$$\dfrac{\dfrac{\dfrac{}{0:\mathrm{Nat}}\,T-\mathrm{Zero}}{\mathrm{iszero}\,0:\mathrm{Bool}}\,T-\mathrm{IsZero} \qquad \dfrac{}{0:\mathrm{Nat}}\,T-\mathrm{Zero} \qquad \dfrac{\dfrac{}{0:\mathrm{Nat}}\,T-\mathrm{Zero}}{\mathrm{pred}\,0:\mathrm{Nat}}\,T-\mathrm{Pred}}{\mathrm{if}\ \mathrm{iszero}\,0\ \mathrm{then}\ 0\ \mathrm{else}\ \mathrm{pred}\,0:\mathrm{Nat}}\,T-\mathrm{If}$$

Die Regeln werden von unten nach oben gelesen. Unter dem Strich steht die Folgerung, darüber die Voraussetzung.

In diesem Beispiel soll die Folgerung „if iszero 0 then 0 else pred 0 : Nat “ geprüft werden. . Da die Folgerung ein „If- Fall“ ist, wenden wir als erstes die Regel T-If an. Der Regel nach unterteilen wir die Folgerung nun in drei Voraussetzungen: zum einen in den If-Fall vom Typ Bool und zum anderen in den then-else-Fall, der aus zwei Werten besteht (beide vom Typ Nat).
Im nächsten Schritt werden die drei Voraussetzungen weiter analysiert.

„iszero 0:Bool“: T-IsZero gibt dem Term der Form isZero t_1 den Typ Bool, solange t_1 vom Typ Nat ist. In unserem Beispiel ist „IsZero 0“ vom Typ Bool, da der Wert 0 vom Typ Nat ist, was man durch die Anwendung der Regel T-Zero sofort sieht.

„0:Nat“: Die Regel T-Zero wird benutzt, um die Null zu Typen

„pred 0:Nat“: T-Pred gibt dem Term der Form pred t_1 den Typ Nat, solange t_1 vom Typ Nat ist. In unserem Beispiel ist „pred 0“ vom Typ Nat, da der Wert 0 vom Typ Nat ist.

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

Die Fälle T-True, T-False und T-Zero lassen sich sofort zeigen, da ja t in diesen Fällen ein Wert ist. Für die anderen Fälle kann wie folgt zeigen:

Fall T-If

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

Es gibt ein t´, so dass t_1 then t_2 else $t_3 \rightarrow t'$.
Entweder ist t_1 ein Wert oder es existiert ein t'_1 für das $t_1 \rightarrow t'_1$ gilt.
Falls t_1 ein Wert ist, dann können wir auf das oben genannte Lemma zurück greifen. Da t_1 vom Typ Bool ist, muss t_1 laut Lemma somit entweder true oder false ergeben. In diesen beiden Fällen bezieht sich t auf die Fälle E-IfTrue (if true then t_2 else $t_3 \rightarrow t_2$) oder E-IfFalse (if false then t_2 else $t_3 \rightarrow t_3$).
Anderseits gibt es noch den Fall $t_1 \rightarrow t'_1$, so dass für T-If gilt: $t \rightarrow$ if t'_1 then t_2 else t_3.

Ähnlich wie der Fall T-If werden auch folgende Fälle hergeleitet:

Fall T-Succ

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

Fall T-Pred

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

Fall T-IsZero

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

Auf diese Fälle wird hier aber nicht näher eingegangen.

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

Wenn die letzte Regel einer Ableitung T-True ist, dann wissen wir, dass t eine Konstante true ist. Somit muss T vom Typ Bool sein. Somit ist t ein Wert, so dass der Fall $t \rightarrow t'$ für alle t´ nicht eintreten kann.

Fall T-If

t = if t_1 then t_2 else t_3	t_1 : Bool t_2 : T t_3 : T

Wenn die letzte Regel einer Ableitung T-If ist, so hat t die Form if t_1 then t_2 else t_3. Hier muss jetzt eine Fallunterscheidung vorgenommen werden, denn t_1 kann die Werte true (Fall E-IfTrue) oder false (E-IfFalse) annehmen. Andererseits kann auch die Regel (E-If) angewendet werden, so dass wir ein t_1 brauchen, dass sich auf t'_1 abbilden lässt.

1.Fall: E-IfTrue:	Die E-IfTrue-Regel besagt: if true then t_2 else $t_3 \rightarrow t_2$. Wenn also t_1= true, dann gibt es ein $t' = t_2$. Durch die Regel T-If wissen wir, dass t_2 : T.
2.Fall: E-IfFalse:	Hier gilt dasselbe wie bei E-IfTrue, da if false then t_2 else $t_3 \rightarrow t_3$.
3.Fall: E-If:	$t_1 \rightarrow t'_1$ und $t' =$ if t'_1 then t_2 else t_3 Durch den Fall T-If wissen wir, dass t_1 : Bool. Hier können wir die Induktionsvoraussetzung anwenden, so dass wir t'_1 : Bool erhalten. Dies kombiniert mit den Fakten, dass t_2 : T und t_3 : T (Fall T-If) ist, kann man unter Anwendung des Falles T-If darauf schließen, dass if t'_1 then t_2 else t_3 : T und somit t´: T.

Fall T-Zero

t = 0	T = Nat

Kann nicht passieren (Begründung siehe T-True)

5. Einfach getypte Lambda-Kalkül

5.1 Typisierung:

Typisierungsrelation: $\Gamma \vdash t : T$

Regel

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

Abstraktionsregel: $$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}\ T_1.t_2 : T_1 \to T_2}\ \text{T-Abs}$$

Anwendungsregel: $$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}\ \text{T-App}$$

5.2 Beispiel:

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

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 $t \to t'$.

Die Fälle sind hier genauso zu behandeln wie bei den arithmetischen Gleichungen. Der einzige Unterschied, den es hier gibt, ist, dass wir jetzt geschlossene Terme haben und nicht wie vorher mit freien Variablen arbeiten.

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

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

Hier sind zwei Fälle zu berücksichtigen: Entweder ist z gleich x oder eine andere Variable.
Falls z = x, dann $[x \to s]z = s$. Das erforderliche Resultat ist dann $\Gamma \vdash s : S$.
Anderenfalls ist $[x \to s]z = z$. Hier ist das gewünschte Ergebnis gleich sichtbar.

Fall T-Abs

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

Der Fall wird hier nicht näher behandelt, da etliche neue Lemmas, wie z.B. weakening und Lemma 5.3.4, neu eingeführt werden müssten. Diese Lemmas wären nur für diesen Beweis nötig und würde meineserachtens den Zeitrahmen sprengen.

Fall T-True

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

Dann ist $[x \to s]t = \text{true}$. Damit ist das gewünschte Ergebnis $\Gamma \vdash [x \to s]t : T$ sofort sichtbar.

Fall T-False

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

Dieser Fall wird exakt so bewiesen wie der Fall T-True.

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“

Das schön an der „Curry-Howard Correspondance“ ist, dass es nicht auf ein spezielles Typsystem limitiert ist. D.h. man kann es auf eine große Anzahl von Typsystemen und der Logik anwenden.