Unterlagen: Grundlagen der Programmierung 2, Sommersemester 2018

Prof. Dr. Manfred Schmidt-Schauß
Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt

E-mail:schauss@ki.informatik.uni-frankfurt.de
URL: http://www.ki.informatik.uni-frankfurt.de/
Tel: 069 / 798 28597
Fax: 069/ 798 28919

9. Mai 2018

Kapitel 3

Haskell, Typen und Typberechnung

3.1 Typisierung in Haskell

In diesem Kapitel sind die Typisierungsmethoden in Haskell beschrieben, wobei auch kurz auf die Typisierung in anderen Programmiersprachen eingegangen wird.

In Haskell gilt, dass jeder gültige Ausdruck einen Typ haben muss. Der Typchecker sorgt dafür, dass es keine Ausnahmen gibt. Diese Art des Typchecks nennt man *starke statische Typisierung*. Die Theorie sagt dann, dass nach erfolgreichem Typcheck keine Typfehler zur Laufzeit mehr auftreten können.

Das Gegenteil ist *dynamische Typisierung*, die eine Typüberprüfung zur Laufzeit macht. D.h. es gibt ein Typsystem, alle Datenobjekte haben einen Typ, meist durch die Implementierung der Daten gegeben. Bei falschem Aufruf wird ein Laufzeit-Typfehler gemeldet.

Die *schwache, statische Typisierung* hat einen Typcheck zur Kompilierzeit, aber es sind Löcher gelassen für den Programmierer, so dass in bestimmten Fällen entweder ein dynamischer Typcheck notwendig ist und eventuell doch ein Typfehler zur Laufzeit auftreten kann. Normalerweise haben alle Konstanten, Variablen (Bezeichner), Prozeduren und Funktionen einen im Programm festgelegten Typ. Typisch für diese Sprachen ist, dass man zur Laufzeit den Typ per Programmbefehl abfragen kann.

Monomorphe Typisierung erzwingt, dass alle Objekte und insbesondere Funktionen einen eindeutigen Typ haben, d.h. Typvariablen sind verboten bzw. es gibt keine Allquantoren in den Typen. In Haskell würde das bedeuten, dass man für Listen von Zahlen und Listen von Strings verschiedene Längenfunktionen bräuchte.

Polymorphe Typisierung erlaubt es, Funktionen zu schreiben, die auf mehreren Typen arbeiten können, wobei man schematische Typen meint. In Haskell sind die Typvariablen verantwortlich für die Schematisierung. Man

nennt dies auch *parametrischen Polymorphismus*. In dieser Typisierung ist es möglich, eine einzige Längenfunktion für alle Listen zu verwenden.

Oft interpretiert man die polymorphe Typisierung auch so: Ein Ausdruck kann eine Menge von Typen haben (vielgestaltig, polymorph). Die (teilweise unendliche) Menge der Typen kann man dann oft in einem einzigen schematischen Typausdruck notieren.

Die weitere Strukturierung der Typen in Typklassen ignorieren wir zunächst mal. Wir geben die Syntax von Typen in Haskell an:

Definition 3.1.1 *Syntax der Haskell-Typen (ohne Typklassen-Information)*

$$\begin{array}{lll} \langle \mathit{Typ} \rangle & ::= & \langle \mathit{Basistyp} \rangle \mid (\langle \mathit{Typ} \rangle) \mid \langle \mathit{Typvariable} \rangle \\ & & \mid \langle \mathit{Typkonstruktor}_n \rangle \{ \langle \mathit{Typ} \rangle \}^n \\ & & \quad \textit{(n ist dessen Stelligkeit)} \\ \langle \mathit{Basistyp} \rangle & ::= & \texttt{Int} \mid \texttt{Integer} \mid \texttt{Float} \mid \texttt{Rational} \mid \texttt{Char} \end{array}$$

Typkonstruktoren können benutzerdefiniert sein (z.B. `Baum .`). Eingebaute Typkonstruktoren sind: Liste $[\cdot]$, Tupel `(.,...,.)` für Stelligkeiten ≥ 2 und Funktion $\rightarrow$ (Stelligkeit 2).
Typen mit $\rightarrow$ sind sogenannte *Funktionstypen*. Die Konvention ist, dass man normalerweise $a \rightarrow b \rightarrow c \rightarrow d$ schreiben darf, und der gemeinte Typ dann durch Rechtsklammerung entsteht, d.h. $a \rightarrow (b \rightarrow (c \rightarrow d))$. Es gibt spezielle syntaktische Konventionen für Tupel und Listen. Zum Beispiel ist $(\texttt{Int}, \texttt{Char})$ der Typ eines Paars von Zahlen und Zeichen; $[\texttt{Int}]$ ist der Typ einer Liste von Zahlen des Typs `Int`.

Man verwendet den doppelten Doppelpunkt zur Typangabe. $t :: \tau$ soll bedeuten, dass der Ausdruck t den Typ τ hat. Dies ist syntaktisch auch in Haskell-Programmen erlaubt.

- Basistypen nennt man auch *elementare Typen*.
- Einen Typ nennt man auch *Grundtyp*, wenn er keine Typvariablen enthält. Einen solchen Typ nennt man manchmal auch *monomorph*.
- Einen Typ nennt man *polymorph*, wenn er Typvariablen enthält.

Beispiel 3.1.2

- $\texttt{length} :: [a] \rightarrow \texttt{Int}$
  D.h. die Funktion `length` *hat als Argument ein Objekt vom Typ* $[a]$*, wobei* a *noch frei ist. Das Ergebnis muss vom Typ* `Int` *sein.*
  Will man den Typ als Formel ausdrücken, so könnte man schreiben:

  $$\forall a.\texttt{length} :: [a] \rightarrow \texttt{Int}$$

  Das kann man interpretieren als: Für alle Typen a *ist* `length` *eine Funktion, die vom Typ* $[a] \rightarrow \texttt{Int}$ *ist.*

  Da das eine Aussage über Argumente und Resultate ist, sollte folgendes gelten: $\forall x.x :: [a] \Rightarrow (\texttt{length x}) :: \texttt{Int}$.

- $(:) :: a \to [a] \to [a]$.
  Der Listenkonstruktor hat zwei Argumente. Das erste hat irgendeinen Typ a. Das zweite Argument ist eine Liste, deren Elemente vom Typ a, d.h. von gleichem Typ wie das erste Argument sein müssen. Das Ergebnis ist eine Liste vom Typ $[a]$.

- `map` $:: (a \to b) \to [a] \to [b]$ *beschreibt den Typ der Funktion* `map`.

3.1.1 Typregeln

Die wichtigste Typregel in Haskell, die auch in anderen Programmiersprachen mit monomorphem Typsystem gilt, betrifft die Anwendung von Funktionen auf Argumente: Wenn σ, τ Typen sind, dann gilt die Regel:

$$\frac{s :: \sigma \to \tau \ , \quad t :: \sigma}{(s\ t) :: \tau}$$

Zum Beispiel ist die Anwendung der Funktion `quadrat` :: `Int` $\to$ `Int` auf eine Zahl 2 : `Int`. D.h. `quadrat 2` :: `Int`.
Wenn man z.B. die Funktion `+` anwendet, mit dem Typ + :: `Int` $\to$ `Int` $\to$ `Int`, dann erhält man für $(1 + 2)$ die voll geklammerte Version $((+\ 1)\ 2)$. $(+\ 1)$ hat dann den Typ (`Int` $\to$ `Int`), und $((+\ 1)\ 2)$ den Typ `Int`. Man kann auch die Regel im Fall mehrerer Argumente etwas einfacher handhaben, indem man sofort alle Argumenttypen einsetzt. Die zugehörige Regel ist dann

$$\frac{s :: \sigma_1 \to \sigma_2 \ldots \to \sigma_n \to \tau \quad , \quad t_1 :: \sigma_1 \ , \ \ldots \ , \ t_n :: \sigma_n}{(s\ t_1\ \ldots\ t_n) :: \tau}$$

Zum Beispiel ergibt das für `(+ 1 2)`:

$$\frac{+ :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}, \quad 1 :: \texttt{Int}\ , 2 :: \texttt{Int}}{(+\ \texttt{1}\ \texttt{2}) :: \texttt{Int}}$$

Dies lässt sich noch nicht so richtig auf die Funktion `length` oder `map` anwenden. Da die Haskelltypen aber Typvariablen enthalten können, formulieren wir die erste Regel etwas allgemeiner, benötigen dazu aber den Begriff der Instanz eines Typs.

Definition 3.1.3 *Wenn γ eine Funktion auf Typen ist, die Typen für Typvariablen einsetzt, dann ist γ eine* Typsubstitution.
Wenn τ ein Typ ist, denn nennt man $\gamma(\tau)$ eine Instanz von τ.

Beispiel 3.1.4 *Man kann mit $\gamma = \{a \mapsto \texttt{Char}, b \mapsto \texttt{Float}\}$ die Instanz $\gamma([a] \to$ `Int`$) =$ [`Char`] $\to$ `Int` bilden.*

Die **Regel für die Anwendung** lautet dann:

$$\frac{s : \sigma \to \tau, \quad t : \rho \text{ und } \gamma(\sigma) = \gamma(\rho)}{(s\ t) :: \gamma(\tau)} \qquad \text{wobei die Typvariablen } \rho \text{ umbenannt sind}$$

Die Umbenennung soll so sein, dass die Mengen der Typvariablen von $sigma \to \tau$ und ρ disjunkt sind, damit keine ungewollten Konflikte entstehen.

Beispiel 3.1.5 *Die Anwendung für den Ausdruck* `map quadrat` *ergibt folgendes. Zunächst ist*

$$\texttt{map} :: (a \to b) \to [a] \to [b]$$

Instanziiert man das mit der Typsubstitution $\{a \mapsto \texttt{Int}, b \mapsto \texttt{Int}\}$*, dann erhält man, dass* `map` *u.a. den Typ* $(\texttt{Int} \to \texttt{Int}) \to [\texttt{Int}] \to [\texttt{Int}]$ *hat. Damit kann man dann die Regel verwenden:*

$$\frac{\texttt{map} : (\texttt{Int} \to \texttt{Int}) \to [\texttt{Int}] \to [\texttt{Int}], \quad \texttt{quadrat} :: (\texttt{Int} \to \texttt{Int})}{(\texttt{map quadrat}) :: [\texttt{Int}] \to [\texttt{Int}]}$$

Die Erweiterung auf eine Funktion mit n Argumenten, wenn γ eine Typsubstitution ist, sieht so aus:

$$\frac{s :: \sigma_1 \to \sigma_2 \ldots \to \sigma_n \to \tau, \quad t_1 : \rho_1, \ldots, t_n : \rho_n \text{ und } \forall i : \gamma(\sigma_i) = \gamma(\rho_i)}{(s\ t_1 \ldots t_n) :: \gamma(\tau)}$$

Auch hierbei müssen die Mengen der Typvariablen von $\sigma_1 \to \sigma_2 \ldots \to \sigma_n \to \tau$ einerseits und $\rho_1, \ldots, \rho_n$ andererseits, disjunkt sein.

Bei diesen Regeln ist zu beachten, dass man das allgemeinste γ nehmen muss, um den Typ des Ergebnisses zu ermitteln. Nimmt man irgendeine andere, passende Typsubstitution, dann erhält man i.a. einen zu speziellen Typ.

Beispiel 3.1.6 *Betrachte die Funktion* `id` *mit der Definition* `id x = x` *und dem Typ* $a \to a$*. Um Konflikte zu vermeiden, nehmen wir hier* $a' \to a'$ *Der Typ von* `(map id)` *kann berechnet werden, wenn man obige Regel mit der richtigen Typsubstitution benutzt. Der Typ des ersten Arguments von* `map` *muss* $a \to b$ *sein, und* `id` *erzwingt, dass* $a = b$*. Die passende Typsubstitution ist* $\gamma = \{b \mapsto a, a' \mapsto a\}$*. Das ergibt unter Anwendung der Regel*

$$\frac{\texttt{map} : (a \to b) \to ([a] \to [b]), \quad \texttt{id} :: a' \to a'}{(\texttt{map id}) :: \gamma([\texttt{a}] \to [\texttt{b}])}$$

d.h. $(\texttt{map id}) :: ([a] \to [a])$.

3.1.2 Berechnen der Typsubstitution

Im folgenden zeigen wir, wie man die Typsubstitution γ nicht nur geschickt rät, sondern ausrechnet, wenn man Typen $\delta_i, \rho_i, i = 1, \ldots, n$ gegeben hat, die nach der Einsetzung gleich sein müssen. Diese Berechnung nennt man auch *Unifikation*.

γ muss so gewählt werden, dass folgendes gilt $\forall i : \gamma(\delta_i) = \gamma(\rho_i)$. Man kann dieses Gleichungssystem umformen bzw. zerlegen.

Der Algorithmus operiert auf einem Paar aus einer Lösung G und einer Multimenge E von Gleichungen. TC bezeichnet Typkonstruktoren, a eine

Typvariable, σ, τ (polymorphe) Typen.

Wir dürfen folgende Regeln zur Umformung des Gleichungssystems $\delta_i \doteq \rho_i, i = 1, \ldots, n$ benutzen, wobei wir mit $G = \emptyset$ starten.

$$(Dekomposition) \quad \frac{G \ ; \ \{(TC \ \sigma_1 \ldots \sigma_m) \doteq (TC \ \tau_1 \ldots \tau_m)\} \cup E}{G \ ; \ \{\sigma_1 \doteq \tau_1, \ldots, \sigma_m \doteq \tau_m\} \cup E}$$

Wenn die Typkonstruktoren rechts und links verschieden sind, dann kann man die Berechnung abbrechen; es kann keine Lösung geben.

$$(Ersetzung) \quad \frac{G \ ; \ \{a \doteq \sigma\} \cup E}{G[\sigma/a] \cup \{a \mapsto \sigma\} \ ; \ E[\sigma/a]} \quad \text{Wenn } a \text{ nicht in } \sigma \text{ vorkommt}$$

wobei $E[\sigma/a]$ bedeutet: Ersetze alle Vorkommen der Typvariablen a durch den Typ σ, und $G[\sigma/a]$ bedeutet: Ersetze in alle Lösungskomponenten $x \mapsto s$ durch $x \mapsto s[\sigma/a]$.
Hierbei ist wegen der Terminierung der Berechnung darauf zu achten, dass σ die Typvariable a nicht enthält, da man sonst bei der Berechnung keinen Fortschritt erzielt, bzw. die Berechnung nicht terminiert. Man kann auch zeigen, dass es in diesem Fall keine Lösung gibt.

$$(Vereinfachung) \quad \frac{G \ ; \ \{a \doteq a\} \cup E}{G \ ; \ E}$$

$$(Vertauschung) \quad \frac{G \ ; \ \{\sigma \doteq a\} \cup E}{G \ ; \ \{a \doteq \sigma\} \cup E} \quad \text{Wenn } \sigma \text{ keine Typvariable ist.}$$

Wir sind fertig, wenn E leer ist. Dann hat G die Form $a_1 \mapsto \tau_1, \ldots, a_k \mapsto \tau_k$t, wobei a_i Typvariablen sind, und die a_i in keiner rechten Seite τ_j einer Gleichung auftreten. Die Typsubstitution ist dann direkt ablesbar.

Wenn E nicht leer ist, aber keine Regel anwendbar, dann ist das Verfahren nicht erfolgreich und es wird keine Lösung gefunden.
Es gibt auch Regeln, die direkt anzeigen, dass die Gleichungen nicht lösbar sind:
Es gibt keine Lösung, wenn:

- $x \doteq t$ in E, $x \neq t$ und x kommt in t vor.
- $(f \ \ldots) \doteq (g \ \ldots)$ kommt in E vor und $f \neq g$.

Die Korrektheit des Verfahrens wollen wir hier nicht zeigen, aber es ist offensichtlich, dass die so berechnete Substitution das letzte Gleichungssystem erfüllt.

Die **Regel für die Anwendung mit Unifikation** ist:

$$\frac{s : \sigma \to \tau, \quad t : \rho \text{ und } \gamma(\sigma) = \gamma(\rho)}{(s\ t) :: \gamma(\tau)}$$

wenn γ allgemeinster Unifikator von $\sigma \doteq \rho$ ist,
wobei die Typvariablen in ρ umbenannt sind

Beispiel 3.1.7 *Bei der Typisierung für* `(map id)` *zeigt sich, dass man unterschiedliche Typsubstitutionen finden kann:*

Die eigentliche Typisierung ist:

$$\frac{\texttt{map} :: (a \to b) \to [a] \to [b], \texttt{id} :: a' \to a'}{(\texttt{map id}) :: \gamma([a] \to [b])}$$

Berechnen von γ:

G	E
	$a \to b \doteq a' \to a'$
	$a \doteq a', b \doteq a'$
$a \mapsto a'$	$b \doteq a'$
$a \mapsto a', b \mapsto a'$	

Das ergibt die Typsubstitution $\gamma = \{a \mapsto a', b \mapsto a'\}$

Eine andere Berechnung von γ mit einer Vertauschung ergibt:

G	E
	$a' \to a' \doteq a \to b$
	$a' \doteq a, a' \doteq b$
$a' \mapsto a$	$a \doteq b$
$a' \mapsto a, a \mapsto b$	

Das ergibt die Typsubstitution

$$\gamma = \{a' \mapsto a, a \mapsto b\}$$

Wir erhalten dann entweder `(map id)`$:: [a'] \to [a']$ *oder* `(map id)`$:: [b] \to [b]$*, was jedoch aufgrund der All-Quantifizierung bis auf Umbenennung identische Typen sind.*

Definition 3.1.8 *Sei V eine Menge von Typvariablen und γ, γ' zwei Typsubstitutionen. Dann ist γ allgemeiner als γ' (bzgl. V), wenn es eine weitere Typsubstitution δ gibt, so dass für alle Variablen $x \in V$: $\delta(\gamma(x)) = \gamma'(x)$.*

Um das zum Vergleich von Typsubstitutionen anzuwenden, nimmt man normalerweise V als Menge der Typvariablen die in der Typgleichung vor der Unifikation enthalten sind.

Beispiel 3.1.9 *Nimmt man $V = \{a_1\}$, dann ist $\gamma = \{a_1 \mapsto (a_2, b_2)\}$ allgemeiner als $\gamma' = \{a_1 \mapsto (\texttt{Int}, \texttt{Int})\}$, denn man kann γ' durch weitere Instanziierung von γ erhalten: Nehme $\delta = \{a_2 \mapsto \texttt{Int}, b_2 \mapsto \texttt{Int}\}$. Dann ist $\delta(\gamma(a_1)) = (\texttt{Int}, \texttt{Int}) = \gamma'(a_1)$.*

Beispiel 3.1.10 *Der Typ der Liste* $[1]$ *kann folgendermaßen ermittelt werden:*

- $[1] = 1 : []$
- $1 :: \texttt{Int}$ *und* $[] :: [b]$ *folgt aus den Typen der Konstanten.*
- $(:) :: a \to [a] \to [a]$
- *Anwendung der Regel mit* $\gamma = \{a \mapsto \texttt{Int}\}$ *ergibt:*
  $(1 :) :: [\texttt{Int}] \to [\texttt{Int}]$
- *Nochmalige Anwendung der Regel mit* $\gamma = \{b \mapsto \texttt{Int}\}$ *ergibt:*
  $(1 : []) :: [\texttt{Int}]$

Beispiel 3.1.11 *Wir weisen nach, dass es keinen Typ von* $[1, '\texttt{a}']$ *gibt: Der voll geklammerte Ausdruck ist* $1 : ('\texttt{a}' : [])$.
$1 :: \texttt{Int}$, $[] :: [b]$ *und* $'\texttt{a}' :: \texttt{Char}$ *folgen aus den Typen der Konstanten.*
Wie oben ermittelt man: $(1 :) :: [\texttt{Int}] \to [\texttt{Int}]$ *und*
`'a':[]` *::* $[\texttt{Char}]$.

Wenn wir jetzt die Typregel anwenden wollen, dann stellen wir fest: es gibt kein γ*, das* $[\texttt{Int}]$ *und* $[\texttt{Char}]$ *gleichmacht. D.h. die Regel ist nicht anwendbar.*

Da das auch der allgemeinste Versuch war, einen Typ für diese Liste zu finden, haben wir nachgewiesen, dass die Liste $[1, 'a']$ *keinen Typ hat; d.h. der Typchecker wird sie zurückweisen.*

```
Prelude> [1,'a'] ↩

<interactive>:1:1:
    No instance for (Num Char)
      arising from the literal ‘1’ at <interactive>:1:1
    Possible fix: add an instance declaration for (Num Char)
    In the expression: 1
    In the expression: [1, 'a']
    In the definition of ‘it’: it = [1, 'a']
```

Beispiel 3.1.12 *Wir zeigen, wie der Typ von* `(map quadrat [1,2,3,4])` *ermittelt wird.*

- `map` *hat den Typ* $(a \to b) \to [a] \to [b]$
  `quadrat` *den Typ* $\texttt{Integer} \to \texttt{Integer}$*, und* $[1, 2, 3, 4] :: [\texttt{Integer}]$.
- *Wir nehmen* $\gamma = \{a \mapsto \texttt{Integer}, b \mapsto \texttt{Integer}\}$.
- *Das ergibt* $\gamma(a) = \texttt{Integer}$, $\gamma([a]) = [\texttt{Integer}]$, $\gamma([b]) = [\texttt{Integer}]$.

- *Damit ergibt sich mit obiger Regel, dass das Resultat vom Typ* $\gamma([b]) =$ [`Integer`] *ist.*

Etwas komplexer ist die Typisierung von definierten Funktionen, insbesondere von rekursiv definierten. Man benötigt auch weitere Regeln für Lambda-Ausdrücke, `let`-Ausdrücke und List-Komprehensionen. In Haskell und ML wird im wesentlichen der sogenannte Typcheckalgorithmus von Robin Milner verwendet. Dieser ist i.a. schnell, aber hat eine sehr schlechte worst-case Komplexität: in seltenen Fällen hat er exponentiellen Zeitbedarf. Dieser Algorithmus liefert allgemeinste Typen im Sinne des Milnerschen Typsystems. Allerdings nicht immer die allgemeinsten möglichen polymorphen Typen.
Folgender Satz gilt im Milner-Typsystem. Wir formulieren ihn für Haskell.

Satz 3.1.13 *Sei t ein getypter Haskell-Ausdruck, der keine freien Variablen enthält (d.h. der geschlossen ist). Dann wird die Auswertung des Ausdrucks t nicht mit einem Typfehler abbrechen.*

Dieser Satz sollte auch in allen streng typisierten Programmiersprachen gelten, sonst ist die Typisierung nicht viel wert.

Wir untermauern den obigen Satz, indem wir uns die Wirkung der Beta-Reduktion auf die Typen der beteiligten Ausdrücke anschauen.

Erinnerung Beta-Reduktion: $\dfrac{((\lambda x.t)\ s)}{t[s/x]}$.

Der Typ von $\lambda x.t$ sei $\tau_1 \to \tau_2$. Der Typ von s sei σ. Damit der Ausdruck $((\lambda x.t)\ s)$ einen Typ hat, muss es eine (allgemeinste) Typsubstitution γ geben, so dass $\gamma(\tau_1) = \gamma(\sigma)$ ist. Der Ausdruck hat dann den Typ $\gamma(\tau_2)$.

Jetzt verwenden wir γ um etwas über den Typ des Resultatterms $t[s/x]$ herauszufinden. Dazu muss man wissen, dass der Milner-Typcheck nur den Typ der Unterterme beachtet, nicht aber deren genaue Form. Verwendet man γ, dann hat unter γ der Ausdruck s und die Variable x den gleichen Typ; jeweils als Unterterme von t betrachtet. D.h. Der Typ von $t[s/x]$ ist unter der Typsubstitution γ gerade $\gamma(\tau_2)$.

Da man das für jede Typsubstitution machen kann, kann man daraus schließen, dass die Beta-Reduktion den Typ erhält.

Leider ist die Argumentation etwas komplizierter, wenn die Beta-Reduktion innerhalb eines Terms gemacht wird, d.h. wenn t Unterausdruck eines anderen Ausdrucks ist, aber auch in diesem Fall gilt die Behauptung, dass die Beta-Reduktion den Typ nicht ändert.

Analog kann man die obige Argumentation für andere Auswertungsregeln verwenden.

3.1.3 Bemerkungen zu anderen Programmiersprachen

In den funktionalen Programmiersprachen Lisp und Scheme, die beide höherer Ordnung sind, gibt es in den Standardvarianten kein statisches Typsystem. Der Grund ist, dass schon in den ersten Konzeptionen dieser Sprachen keine statische Typdisziplin eingehalten wurde: Z.B. ist es erlaubt und wird auch genutzt,

dass man alle Werte als Boolesche Werte verwenden darf: Die Konvention ist: `Nil` = `False`, alles andere gilt als `True`. Z.B. ist der Wert von 1 = 0 in Lisp die Konstante `Nil`, während 1 = 1 die Konstante 1 ergibt. Damit hat man Ausdrücke, die sowohl Boolesche Ausdrücke sind als auch Listen; und Ausdrücke, die Boolesche Ausdrücke sind und Zahlen.

Als weitere Schwierigkeit kommen die Typeffekte von Zuweisungen hinzu. D.h. um einen vernünftigen Typisierungsalgorithmus für diese Programmiersprachen zu konzipieren, müsste man zu viel ändern.

Programmiersprachen, die keine Funktionen oder Prozeduren höherer Ordnung haben, begnügen sich meist mit einem monomorphen Typcheck.

Bei Verwendung der arithmetischen Operatoren, z.B. des Additionszeichens (+) will man oft erreichen, dass dies für alle Zahlentypen wie `Int, Rational` usw. funktioniert. In manchen Programmiersprachen wird vom Compiler automatisch eine Typkonversion eingefügt, wenn unverträgliche Zahltypen benutzt werden.
In Haskell sind eingegebene Zahlkonstanten normalerweise überladen, indem z.B. bei Eingabe der Zahl 1 vom Compiler `fromInteger 1` eingefügt wird. Wenn unverträgliche Operanden benutzt werden, muss man die Typkonversion von Hand einfügen. Z.B. `pi + 2%3` ergibt einen Typfehler, während `pi + (fromRational (2%3))` korrekt `3.80825932::Double` ergibt.

Python ist objektorientiert und hat ein Klassensystem. Damit kann man Klassen als Typen verwenden, was aber nicht zu einem starken, sondern zu einem dynamisch Typsystem führt. Man könnte vermutlich ein monomorphes Typsystem zu Python hinzufügen. Python hat aber keine syntaktischen Elemente, um den Typ einer Variablen festzulegen; und als weitere Schwierigkeit kommt hinzu, dass Python Lambda-Ausdrücke zulässt. Diese Kombination erlaubt mit Sicherheit kein strenges Typsystem, es sei denn, man nimmt hin, dass viele in normalem Python erlaubten Programme in getypten Python verboten wären.

Die Programmiersprache **Java** ist streng getypt. Das Typsystem ist monomorph, allerdings entsprechen die Java-Klassen den elementaren Typen. Zudem gibt es einen Untertyp-Begriff für Klassen, d.h. für die elementaren Typen. Die Typfehler, die jetzt noch auftreten, sind meist von der Art, dass ein syntaktisch geforderter Typ nicht mit dem aktuellen Objekt übereinstimmt, d.h. Fehler beim (cast).