Typsysteme für Programmiersprachen

Johannes Waldmann

Institut für Informatik, Universität Leipzig

- Benjamin C. Pierce: *Types and Programming Languages*, MIT Press, 2002
- Henk Barendregt: *Lambda Calculi with Types*, in: Handbook of Logic in Computer Science
- Martin Abadi und Luca Cardelli: *A Theory of Objects*, Springer 1996

Wer, wie, was; wieso, weshalb, warum?

Ein Typsystem ist ein syntaktisches Verfahren, das Programmbestandteile danach klassifiziert, welche Art von Werten sie berechnen.

Damit kann man automatisch bestimmte unerwünschte Verhaltensweisen des Programms ausschließen.

Typsysteme spielen deswegen eine bedeutende Rolle bei: Software-Entwicklung, Entwurf von Sprachen, hoch performanten Compilern, und Sicherheit.

(Pierce: Types and Programming Languages, 2002)

Einfache Typen

G: Grund-Daten-Typen (Bsp: Zahl, Zeichen, Wahrheitswert)

Jeder Programm-Teil-Ausdruck hat einen Typ.

Welchen Typ haben Funktionen?

Wenn $x :: a$ und $f :: a \to b$, dann $f(x) :: b$.

Welche Typen kommen vor?

$t \in T$, wenn $t \in G$ oder $t = t_1 \to t_2$ für $t_1, t_2 \in T$.

Eigenschaften von Typsystemen

Beziehung zur Semantik (definiert durch Reduktion $\to_\beta$)

subject reduction: Wenn $A :: t$ und $A \to_\beta B$, dann $B :: t$

Entscheidbarkeit/Konstruierbarkeit von:

- type checking: gegeben A, t, gilt $A :: t$?
- type inference: gegeben A, finde t mit $A :: t$
- type inhabitation: gegeben t, finde A mit $A :: t$

Deduktionsregeln für Typen

Kontexte $\Gamma = (x_1 :: t_1, \dots, x_n :: t_n)$

Kontext benutzen: $\frac{(x :: t) \in \Gamma}{\Gamma \vdash x :: t}$

Applikation: $\frac{\Gamma \vdash f :: a \to b, \Gamma \vdash x :: a}{\Gamma \vdash (fx) :: b}$

Abstraktion: $\frac{\Gamma, (x :: a) \vdash M :: b}{\Gamma \vdash \lambda x.M :: (a \to b)}$

In diesem System (einfach getypter Lambda-Kalkül) gilt: jeder getypte Term besitzt eine Normalform ($\sim$ jedes Programm hält).

Einfache Funktionen

Es gibt Grundfunktionen (eingebaut) und nutzerdefinierte:

$g(x) = M$, wobei M ein Ausdruck ist, der (evtl.) x enthält

klassische Schreibweise: $g = (x \mapsto M) = \lambda x.M$

Imperative $\subseteq$ funktionale Programmierung

$x_1 := a_1; x_2 := a_2; \dots; \text{return}\, e \Rightarrow (\lambda x_1.((\lambda x_2.(\dots e \dots))a_2))a_1$

Zusammengesetzte Typen

Produkt: $t = t_1 \times \ldots \times t_n$

Disjunkte Summe: $t = C_1 t_1 + \ldots + C_m t_m$.

Namensgleichheit oder Strukturgleichheit?

Rekursive Typen

Bsp. Zeichenketten

String $= A() + B($Char $\times$ $\boxed{\text{String}}$$)$

existiert? ist eindeutig? Fixpunkt-Operator für Typen!

Iteration/Rekursion

$f = \lambda x. \ldots \boxed{f}(g(x)) \ldots$ vgl. $q = 1 - 2 \cdot \boxed{q}$ oder $z = 1 + \boxed{z}^2$

f existiert? f hat Typ? f ist Fixpunkt von F:

Bsp. $F = \lambda f. \lambda x \ldots \boxed{f}(g(x)) \ldots$

$f = \mathsf{Fix}\, F$ mit $\mathsf{Fix} = \lambda g.(\lambda x.g(xx))(\lambda x.g(xx))$.

Fix hat keinen (einfachen) Typ.

$\forall a : \mathsf{Fix} :: (a \to a) \to a$ ist nicht ableitbar.

System PCF = einfach getypter Kalkül + Axiom für Fix.

Universelle Typen

Welches ist der Typ von $\lambda x.x$? Für alle t gilt $\lambda x.x :: (t \to t)$.

Schreibweise: $\lambda x.x :: (\forall t.t \to t)$

Curry-Howard-Isomorphie

$x :: t \sim x$ ist ein Beweis für Allgemeingültigkeit der Aussage t.

Bsp: $\lambda x.x$ beweist $\forall t : t \to t$.

$K = \lambda xy.x :: \forall a, b : a \to (b \to a)$

Objekte, Typen, Kinds

Terme (für Programme *und* Typen):
$T \Rightarrow V \mid C \mid TT \mid \lambda(x : T).T \mid \Pi(x : T).T$

Sorten $S = \{*, \square\} \subset C$, sprich: * Type, $\square$ Kind.

$x :: A :: *$ bedeutet:
"x ist ein A, und A ist ein Typ (also x ein Objekt)",

$x :: A :: \square$ bedeutet:
"x ist ein A, und A ist ein Kind (also x ein Typ)".

Nicht reguläre Typen

```
data List a = Nil | Cons a (List a)

data D a b = Nil | Cons a (D b a)
Cons 1 (Cons 'a' (Cons 5 (Cons 'c' (Cons 7 Nil))))
    :: D Integer Char

data B a = Leaf a | Node (B (a,a))
Node (Node (Leaf ((1,2), (3,4)))) :: B Integer
```


Parametrisierte Typen: Typkonstruktoren

Bsp: Liste, Menge, Suchbaum; Parameter: Elementtyp.

```
reverse :: forall a . List a -> List a
length  :: forall a . List a -> Integer
```

System F2 (second order polymorphic lambda calculus)

wer das nicht kann, muß es umständlich und/oder unsicher simulieren:

Java: class Object, C: Macros, C++: Templates, Ada: Generics

Eingeschränkte Polymorphie: Typ-Klassen

```
sort :: forall a . Ord a => List a -> List a

class Ord a where (<) :: a -> a -> Bool
```

. . . und Konstruktor-Klassen (= Haskell)

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)
instance Functor List ...
instance Functor Tree ...
```


Spezielles Axiom für den Lambda-Würfel

für $(s_1, s_2) \in S^2 : \dfrac{\Gamma \vdash A : s_1,\ \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi(x :: A).B) :: s_2}$

Instanzen davon:

λ: $(*, *)$ einfach getypt

$\lambda 2$: $(*, *), (\Box, *)$

$\lambda\omega$: $(*, *), (\Box, \Box)$: $\sim$ einfach getypter Lambda-Kalkül auf Typ-Ausdrücken

λP: $(*, *), (*, \Box)$: dependent types (Cayenne . . . hotter than Haskell)

Allgemeine Axiome für den Lambda-Würfel

Axiom: $\vdash * :: \Box$

Applikation: $\dfrac{\Gamma \vdash f :: (\Pi(x :: A).B), \Gamma \vdash a : A}{\Gamma \vdash fa :: B[x := A]}$

Abstraktion: $\dfrac{\Gamma, x :: A \vdash b :: B, \Gamma \vdash (\Pi(x :: A).B) :: s}{\Gamma \vdash (\lambda(x :: A).b) :: (\Pi(x :: A).B)}$

Konversion: $\dfrac{\Gamma \vdash A :: B,\ \Gamma \vdash B' :: s,\ \Gamma \vdash B =_\beta B'}{\Gamma \vdash A :: B'}$

Thesen zum Design von menschenfreundlichen Programmiersprachen

(weitgehend unabhängig von den Daten "erster Ordnung")

- *Abstraktionen* über die Grunddaten der Sprache sollen *in der Sprache selbst* natürlich und sicher ausdrückbar sein.

  (⇒ parametrisierte Daten, polymorphe Funktionen)

- zusammengehörende (Bezeichner für) Daten, Funktionen, Abstraktionen sollen *gruppierbar* sein

  (⇒ hierarchisches Modulsystem)

Existenziell quantifizierte Typen

```
data E = forall a . E a (a -> Integer)
[ E "foo" length, E 4 succ ] :: [ E ]

E x f :: E  ... existst a : x :: a, f :: a -> b
```


Überlappende Instanzen

```
class Show a where show :: a -> String
instance Show a => Show [ a ] where ...
instance Show Char ...
instance Show [ Char ] ... ??
```

mehrparametrige Klassen

```
class C a b where f :: a -> [b] -> [a]
```

funktionale Abhängigkeiten

```
class Container c a | c -> a where ...
```

Stroustrup über Sprach-Design

Wozu dient eine Programmiersprache?

- zur Steuerung von Maschinen?
- zur Kommunikation zwischen Programmierern?
- zum Ausdrücken von Entwürfen? • von Algorithmen?
  - von Beziehungen zwischen Begriffen?
- zum Experimentieren?

Gehört Informatik zu

- Mathematik? • Ingenieurwissenschaften? • Architektur?
- Kunst? • Biologie? • Soziologie? • Philosophie?

Bjarne Stroustrup: *Design and Evolution of C++*, 1994