

Polymorphe Typsysteme

Typsysteme:

- sichern Korrektheit
- erzwingen eine einheitliche Struktur
- vermeiden Fehler, die bei durch den Programmierer „intuitiv“ geschaffenen Typsystemen entstehen können
- ermöglichen Optimierungen, z.B. auf die Hardware
- bieten eine schützende Hülle, indem sie die zugrundeliegende Repräsentation der Objekte der Sprache abschirmt und ihre Interaktionsmöglichkeiten einschränkt

Typ:

Menge von Werten

Programmiersprachen:

Es gibt *monomorphe* und *polymorphe* Programmiersprachen:

	<i>monomorphe</i>	<i>polymorphe</i>
Sprachen	jeder Wert gehört zu genau einem Typ	Werte können mehrere Typen haben
Funktionen	Operanden haben nur einen einzigen Typ	Operanden können mehr als einen Typ haben
Beispiel	<pre>int add (int a, int b) { ... }</pre> <pre>c = add (2,3);</pre>	<pre>public Object clone (Object a) { ... }</pre> <pre>clone (Vector...) clone (Jbutton...)</pre>

statisch getypt:

Typ jedes Ausdrucks kann durch statische Programmanalyse, also bereits **vor** der Ausführung ermittelt werden

streng getypt:

Ausdrücke müssen **typkonsistent** sein, müssen zur Laufzeit bestimmt werden können

Polymorphismus:

zwei unterschiedliche Arten von Polymorphismus: *ad-hoc* und *universeller* Polymorphismus

– *ad-hoc*:

„scheinbarer“ Polymorphismus

– *Überladen*:

verschiedene Funktionen haben denselben **Namen**, es ist also ein rein syntaktischer Mechanismus. Bsp.:

```
real add (real..., real...) {...};  
int add (int..., int...) {...};
```

– *Koerzion*:

semantisch notwendige Typkonvertierungen können weggelassen werden, z.B. kann int als real aufgefaßt werden. Bsp. (s. o.):

```
add (2.5, 4);
```

– *universeller*: „echter“ Polymorphismus

– *inklusionsbasierter*:

ein *Subtyp* (z.B. Toyota) hat die Eigenschaften seines *Supertyps* (z.B. Auto) und mehr, ein Objekt eines *Subtyps* kann also im Kontext eines *Supertyps* verwendet werden. Bsp.:

Angenommen `run` schalte eine Objekt des Typs Auto ein. Obwohl `run` nur für den Typ Auto definiert ist, läßt sich ein Objekt des Typs Toyota verwenden: `run (Toyota...)`

– *parametrischer*:

– eine Funktion hat bei jedem Argument eine implizite oder explizite Typangabe

– dieselbe Funktion kann für alle Argumente jedes Types ohne Änderungen verwendet werden, z.B.

kann eine `length`-Funktion auf Listen mit beliebigem Inhalt angewandt werden:

```
(length (list 1 2 3 4 5)) → 5
```

```
(length (list 'a 'b 'c)) → 3
```

– für die Implementation: eine einheitliche Datenrepräsentation muß vorausgesetzt werden, z.B. wird alles als Objekt repräsentiert, auf das mittels Zeiger zugegriffen wird

Schlüsselwörter:

- type:
 - definiert einen Typ
 - schafft keinen neuen Typ, sondern gibt einem Typausdruck einen Namen. Bsp.:
type IntPair = Int x Int
- value: definiert einen Wert (bzw. eine Funktion)
- rec: rekursive Typdefinition. Bsp.:
rec type IntList = [nil: Unit,
 cons: { head: Int, tail: IntList }]

Universelle Quantifikation:

- ohne sie lassen sich parametrisch polymorphe Funktionen nicht modellieren. Bsp.: Identitätsfunktion:
muß für jeden Typ einzeln definiert werden, also
id: Int → Int
id: Real → Real
- die Einführung von *Typvariablen* ermöglicht es, eine Funktion für einen beliebigen Typ zu definieren:
 - Bsp.:
(define id
 (lambda (x:a)
 x))
 - hierbei steht implizit der Parameter all[a]:
(define id
 (lambda (x:a all[a])
 x))
also: id: $\forall a.a \rightarrow a$

Typoperator:

- arbeitet auf Typen
- versucht gemeinsame Struktur herauszuziehen, so daß zum Beispiel nicht
type IntPair = Int x Int
type BoolPair = Bool x Bool
...
einzeln definiert werden müssen, sondern die Struktur des „Pair“ (Typ x Typ) erfaßt wird:
type Pair[T] = T x T;
⇒ type IntPair = Pair[Int];
⇒ type BoolPair = Pair[Bool];
„Pair“ ist hier ein Typoperator
- Bsp.: List als Typoperator:
rec type List[Item] = [nil: Unit,
 cons: { head: Item, tail: List[Item] }]
⇒ type IntList = List[Int];
⇒ type RealList = List[Real];

Inferenz:

- Herleitung/Erschließung des Typs eines Ausdrucks. Bsp.:
(define add
 (lambda (a b)
 (+ a b)))
- Voraussetzung: primitive Funktionen müssen getypt sein, hier sei +: Int x Int → Int
- der Typ von add, a und b läßt sich dann wie folgt inferieren (herleiten):
(+ a b) ⇒ a: Int, b: Int ⇒ add: Int x Int → Int

Monomorphismus → Polymorphismus:

- *Überladen* (Overloading): + ist z.B. sowohl als +: Int x Int → Int als auch als +: Real x Real → Real definiert
- *Koerzion* (Coercion): z.B. kann Int als Real aufgefaßt werden: add: Real x Real → Real, add (2.5, 3)
- *value sharing*: z.B. ist null in C eine Konstante, die durch alle Zeiger genutzt wird
- *Subtyping*: z.B. können in Java abgeleitete Klassen in den Kontext der Oberklasse eingesetzt werden (s.o.)