Sprachkonzepte

Teil 5: Typsysteme

Typprüfung, Typinferenz, parametrische Polymorphie

Typsysteme

Das **Typsystem** einer Programmiersprache besteht aus

- einer Menge von Typen, die mit Sprachkonstrukten assoziiert werden können

  Sprachen geben üblicherweise einige Typen vor (= primitive Datentypen) und erlauben darauf aufbauend die Konstruktion benutzerdefinierte Typen

  typisierte Sprachkonstrukte sind solche, die einen Wert haben oder sich auf Werte beziehen, z.B. Literale, Variablen, Ausdrücke, Funktionen

- einer Menge von Regeln für die Äquivalenz, Kompatibilität und Inferenz von Typen

Äquivalenz	Wann haben zwei Werte den gleichen Typ?
Kompatibilität	Werte welcher Typen dürfen wo verwendet werden?
Inferenz	Welchen Typ hat ein Ausdruck abhängig von seinen Bestandteilen und dem Kontext?

Typen

Es gibt verschiedene Sichten auf **Typen**, die man nach Bedarf einnehmen kann:

- denotationale Sicht:

  ein Typ bezeichnet eine Menge von Werten

  ein Sprachkonstrukt hat den gegebenen Typ, wenn sein Wert garantiert in der bezeichneten Menge liegt

- strukturelle Sicht

  neue Typen werden aus vordefinierten Typen konstruiert

  übliche Typkonstruktionen sind etwa Array und Record / Struct

- abstraktionsbasierte Sicht

  ein Typ legt die zulässigen Operationen auf seinen Werten fest

  besonders bei der objektorientierten Programmierung üblich (Interface-Methoden)

Klassifikation von Datentypen

skalare Datentypen:

- boolescher Typ
- Zeichentyp(en):
  Zeichencodierung?
- Zahltypen: ganze Zahlen, Gleitkommazahlen, ...
  Vorzeichen? Zahlbereiche? Genauigkeit? Dezimal?

zusammengesetzte Datentypen:

- Array
- Record / Struktur *(mathematische Sprechweise: Tupel)*
- Liste
- ...

Typprüfung

Die **Typprüfung** stellt fest, ob die Regeln zur Typkompatibilität eingehalten sind.

die Verletzung einer Regel wird als Typenkonflikt (type clash) bezeichnet

Eine Programmiersprache ist

- stark typisiert *(strongly typed)*, wenn ihre Typprüfung garantiert, dass auf Werte nur die laut deren Typen zugelassenen Operationen anwendbar sind

  es gibt auch andere Definitionen starker Typisierung als diese von Michael L. Scott

  viele Sprachen enthalten unterschiedlich stark typisierte Bereiche

- statisch typisiert *(statically typed)*, wenn sie stark typisiert ist und die Typprüfung (fast) vollständig zur Compilezeit stattfindet

  erfordert explizite Variablendeklarationen, erleichtert automatische Codeoptimierungen

- dynamisch typisiert *(dynamically typed)*, wenn sie stark typisiert ist und die Typprüfung (fast) vollständig zur Laufzeit stattfindet

  typisch beim Scripting, ermöglicht Verzicht auf Variablendeklarationen

Typprüfung: Typäquivalenz

Typäquivalenz ist die einfachste Form der Typkompatibilität.
Man unterscheidet zwei Arten:

- Namensäquivalenz

  jeder Typname steht für einen anderen Typ

  in C sind etwa `struct s { int a, b; };` *und* `struct t { int a, b; };`
  zwei strikt verschiedene Typen

- strukturelle Äquivalenz

  gleich aufgebaute Typen sind unabhängig vom Namen äquivalent

  in C kann man mit `typedef` *definierte Typen als strukturell äquivalent auffassen*
  (alternativ kann man sie aber auch als reine Aliasnamen betrachten)

Typprüfung: Typumwandlung (1)

Typumwandlung ermöglicht Typkompatibilität für nicht äquivalente Typen:

- implizite Typumwandlung *(type coercion)*

  viele Sprachen erlauben in Ausdrücken Werte mit anderen als den vom Kontext verlangten Typen

  typische Beispiele sind gemischte Ausdrücke mit ganzen Zahlen und Gleitkommazahlen oder die Verwendung von Unterklassereferenzen anstelle von Oberklassereferenzen

  in C++ kann man die implizite Typumwandlung für Objekte durch Konstruktoren und überladene typecast-Operatoren klassenspezifisch regeln

- explizite Typumwandlung *(type cast)*

  in vielen Sprachen kann man den Typ eines Ausdrucks per Operator auf einen vom Kontext verlangten Typ wandeln

  typische Beispiele sind Downcast und Crosscast von Objektreferenzen bei der objektorientierten Programmierung

Typprüfung: Typumwandlung (2)

Implementierung von Typumwandlungen:

- Typaufprägung *(type punning)*

  der Zieltyp wird dem unveränderten Speicherinhalt einfach aufgeprägt,
  d.h. der Speicherinhalt wird im Sinne des Zieltyps uminterpretiert

  der Compiler muss dafür keinen Code erzeugen

  z.B. Zweierkomplementdarstellung einer negativen ganzen Zahl als positive Zahl

  z.B. Teile eines Byte-Arrays als Verwaltungsstruktur mit ganzen Zahlen und Zeigern (typisch in der Systemprogrammierung, etwa in einer Speicherverwaltung)

- Typabbildung

  jedem Wert des Quelltyps wird ein Wert des Zieltyps zugeordnet

  der Compiler erzeugt Code, der zur Laufzeit den zugeordneten Wert bestimmt
  (je nach Sprache und Typ kann es dabei auch zu Laufzeitfehlern kommen)

  z.B. ganze Zahl mit 2 Byte auf 4 Byte erweitern

  z.B. ganze Zahl in entsprechende Gleitkommazahl umrechnen

Typinferenz

Typinferenz ist die Herleitung des Typs eines Sprachkonstrukts aus seinen Bestandteilen und dem Kontext.

- Grundlage für die Typprüfung

  Ist ein Ausdruck, der als Initialisierer / Operand / Argument auftritt, kompatibel mit dem erwarteten Typ?

- Grundlage für eine deklarationsfreie Typisierung

  Welcher Typ soll einer Variablen zugeordnet werden, damit sie mit dem Wert eines angegebenen Ausdrucks initialisiert werden kann?

  erspart bei komplizierten Typen Schreibarbeit und macht Code allgemeingültiger, z.B. in Java:

```
var v = new HashMap<String, Integer>(); // don't repeat yourself :-)
for (var key : v.keySet()) { … } // unabhängig vom Schlüsseltyp!
```

Parametrische Polymorphie (1)

Funktionen oder Datentypen sind **parametrisch polymorph**, wenn sie generisch programmiert sind, d.h. die Typen der behandelten Werte Parameter der Deklaration sind.

man muss für alle vorkommenden Typen nur eine einzige einheitliche Implementierung der Funktion bzw. des Datentyps bereitstellen

im Gegensatz dazu stellt man bei Overloading und Overriding typabhängig unterschiedliche Implementierungen von Funktionen bzw. Datentypen bereit

Typische Anwendungsgebiete:

- abstrakte Datentypen wie Listen, Maps, Mengen, ...

  z.B. in Java: `class LinkedList<E> ... // Elementtyp als Typparameter E`

- Operationen und Algorithmen wie Minimum / Maximum, Suchen, Sortieren, ...

  z.B. in Java: `static <T> void sort(T[] a, Comparator<? super T> c)`

Parametrische Polymorphie (2)

Für die Verwendung müssen parametrisch polymorphe Funktionen und Datentypen mit konkreten Typen instanziiert werden:

- explizite Instanziierung

  bei der Verwendung generischer Datentypen

  z.B. in C++: `std::vector<int> v; // Instanziierung mit Typparameter T = int`

- implizite Instanziierung

  bei der Verwendung generischer Funktionen,
  realisiert mittels Typinferenz für die Aufrufargumente der Funktion

  z.B. in C++: `std::min(1.2, 3.4); // Instanziierung mit Typparameter T = double`

Parametrische Polymorphie (3)

Implementierungsalternativen für die Instanziierung generischer Datentypen und Funktionen:

- Implementierung per Typlöschung *(type erasure)*

  der Compiler ersetzt Typparameter durch einen allgemeinen Typ und ergänzt notwendige Typanpassungen

  z.B. bei Java Generics

- Implementierung per Copy & Paste

  für jeden als Argument vorkommenden Typ erstellt der Compiler eine spezifische Kopie der Implementierung

  z.B. bei C++ Templates

Parametrische Polymorphie: Constraints

Für Typargument können Einschränkungen (**Constraints**) gelten:

- bei Implementierung mit Typlöschung muss der als Argument übergebene Typ von einem allgemeinen Typ abgeleitet sein

  z.B. bei Java nur Unterklassen von java.lang.Object, keine primitiven Grundtypen

- übergebene Typen müssen Operationen unterstützen, die in der generischen Implementierung genutzt werden

  z.B. muss eine Ordnungsrelation implementiert sein, wenn Werte des übergebenen Typs sortiert werden

Die Einschränkungen für Typargument können explizit oder implizit gegeben sein.

Beispiel: Java Generics

Bei Java Generics sind Constraints an die Ableitungshierarchie von Klassen gekoppelt:

- Constraint bei einem unbeschränkter *(unbound)* Typparameter `<T>`

  das Argument für T muss eine Unterklasse von `java.lang.Object` sein

- Constraint bei einem Typparameter mit oberer Schranke `<T extends S>`

  das Argument für T muss eine Unter- bzw. Implementierungsklasse von S sein

  die generische Implementierung darf in diesem Fall Methoden von S verwenden

  in der folgenden generischen Methode aus java.util.Arrays muss das Argument für T Comparable<C> implementieren, mit C gleich T oder Oberklasse von T, damit die Methode die natürliche Ordnung von T verwenden kann:

  ```
  static <T extends Comparable<? super T>> int compare(T[] a, T[] b)
  ```

  Argument für C ist hier kein konkreter Typ, sondern ein Wildcard mit unterer Schranke T

Beispiel: C++ Templates

Bei C++ Templates sind Constraints an die verwendeten Operationen gekoppelt:

- implizite Constraints bei einem Typparameter **`<typename T>`**

  das Argument für T muss alle Operationen unterstützen, die von der generischen Implementierung in der konkreten Anwendung genutzt werden

  das Konzept wird auch "Duck Typing" genannt: alles was wie eine Ente läuft und wie eine Ente quakt, gilt als Ente

- ab C++20 auch explizite Constraints bei einem Typparameter **<**Constraint T**>**

  das angegebene Constraint muss dazu als sogenanntes **concept** definiert sein

  Sortierfunktion aus <algorthm> bis C++17 mit impliziten Constraints:

  ```
  template<typename T> void sort(T first, T last);
  ```

  das gleiche bei C++20 mit expliziten Constraints (etwas vereinfacht):

  ```
  template<std::random_access_iterator T> void sort(T first, T last);
  ```