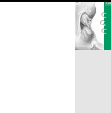



Informatik II

Einige Aspekte von Typsystemen (am Beispiel von Python)

G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de



Definition eines Typsystems

- Ziel: **Semantische** Bedeutung einem Speicherblock zuordnen → Konzept des **Typs** bzw. **Typsystems**
- 3 Definitionsarten von **Typen** :
 - **Denotationale Definition:** $Typ :=$ Menge von Werten
 - Beispiel: **enum, char, int**
 - **Konstruktive Definition:** $Typ :=$
 - entweder: primitiver, eingebauter Typ (int, float, ...),
 - oder: zusammengesetzt aus anderen Typen (struct, class, array, ...).
 - **Abstrakte Definition:** $Typ :=$ Interface, bestehend aus Menge von Operationen (Funktionen, Operatoren), die auf Werte dieses Typs angewendet werden können

G. Zachmann Informatik 2 – SS 2011 Typsysteme 4

- *Type Error* :=
Operation (= Operand oder Funktionsaufruf) ist für beteiligte Typen (Variablen, Ausdrücke, Rückgabewerte) nicht definiert
 - Beispiel: Integer / String
- *Type Checking* :=
Type-Errors finden, dann Fehlermeldung ausgeben oder auflösen
 - Beispiel: Typen zweier Operanden gleich machen
- Auflösen von Type-Errors: *Coercion, Promotion, implicit type cast ...*
 - Beispiele:
- $1 * 1.2 \rightarrow 1.0 * 1.2$

```
int i;
...
if ( i )
{
  ...
}
```

→

```
int i;
...
if ( static_cast<bool>(i) )
{
  ...
}
```

G. Zachmann Informatik 2 – SS 2011
Typsysteme 5

Die zwei wichtigsten Arten des Type-Checking

- Programmiersprachen können bzgl. Type-Checking in 2 Klassen eingeteilt werden (Klassifikation ist nicht immer scharf!)
- *Static type checking / statically typed* :=
Type checking zur Compile-Zeit durch den Compiler
- *Dynamic type checking / dynamically typed* :=
Type checking zur Laufzeit durch den Interpreter (bzw. die Virtual Machine)
 - Geht auch in compiliertem Code
- *Type Inference* :=
Typ der Variablen wird aus dem Kontext hergeleitet;
Typen dürfen unvollständig sein.

G. Zachmann Informatik 2 – SS 2011
Typsysteme 6

Static Typing (z.B. C++ / Java)


- Konsequenz:
 - Compiler muß zu jeder Zeit den Typ eines Wertes im Speicher kennen
 - Relativ einfach machbar, wenn Variablen einem Speicherblock fest zugeordnet sind
 - Typ(Speicherbereich) = Typ(Variable)
 - Bessere Performance (Compiler generiert sofort den richtigen Code)
- Konsequenz für die Sprache bzgl. Variablen:
 - Variable = **unveränderliches Triple** (Name, Typ, Speicherbereich)
 - Variablen müssen vom Programmierer vorab deklariert werden
 - Syntax in C++: `Type Variable;`
 - Beispiele:


```
int i;           // the variable 'i'
float f1, f2;
char c1,        // comment
      c2;        // comment
```

G. Zachmann Informatik 2 – SS 2011 Typsysteme 7

- Beispiel:


```
float i = 3.0;
i = 1; // überschr. 3.0
i = "hello"; // error
int i = 1; // error
```


- Unschön ist: Typ von Speicherblock wird durch Typ der darübergerlegten Variable bestimmt!
- Gründe für *static typing*:
 - Zur Laufzeit ist das Programm schneller
 - Programmierer machen weniger Fehler (Bugs) – oder? ...

G. Zachmann Informatik 2 – SS 2011 Typsysteme 8



Ein Beispiel aus einer sehr statisch typisierten Sprache (Ada) ...

```
...
declare
  vertical_veloc_sensor: float;
  horizontal_veloc_sensor: float;
  vertical_veloc_bias: integer;
  horizontal_veloc_bias: integer;
  ...
begin
  declare
    pragma suppress(numeric_error, horizontal_veloc_bias);
  begin
    sensor_get( vertical_veloc_sensor );
    sensor_get( horizontal_veloc_sensor );
    vertical_veloc_bias := integer( vertical_veloc_sensor );
    horizontal_veloc_bias := integer(horizontal_veloc_sensor);
    ...
  exception
    when numeric_error => calculate_vertical_veloc();
    when others => use_irs1();
  end;
end irs2;
```

... und das Resultat





Start der ersten Ariane-5, 1996



Static types give me the same feeling of safety as the announcement that my seat cushion can be used as a flotation device.

Don Roberts, OOPSLA 2005
Reported in Martin Fowlers' blog on OOPSLA 2005

G. Zachmann Informatik 2 – SS 2011 Typsysteme 11



Dynamic Typing (z.B. in Python)

- Idee: Typ aus der Symboltabelle entfernen, und dafür zum Wert direkt mit in den Speicher schreiben
- Konsequenzen:
 - Variablen haben keinen deklarierten Typ mehr!
 - Werte im Speicher müssen **unveränderlichen** "Type-Tag" haben
 - Typ von Variable/Wert wird zur Laufzeit (jedesmal) überprüft
 - Variable muß nicht an festen Speicherbereich **gebunden** werden
 - Welcher Operator, wird jeweils zur Laufzeit entschieden
 - "**Generic Programming**" bzw. **Polymorphie** wird (fast) trivial
 - Wesentlich kürzere Compile-Zeiten, aber Performance penalty
 - Debugger mit höherer Funktionalität ("*to program in the debugger*")
 - Konsequenz für die Sprache bzgl. Variablen:
 - Variable = **veränderliches Paar** (Name, Speicherbereich)
 - Variablen müssen vom Programmierer **nicht** deklariert werden

G. Zachmann Informatik 2 – SS 2011 Typsysteme 12

Beispiele:

```

i = 3.0
i = 1; // bindet i neu
i = "hello"; // dito

```

```

list = [1,2,3]
listref = list
listcopy = list[:]
list.append( 4 )

```

Zuordnung zwischen Variablenname und Wert/Objekt im Speicher heißt auch *Binding*

G. Zachmann Informatik 2 – SS 2011 Typsysteme 13

Ändern eines Integers

Achtung: manche Operationen erzeugen eine Kopie!

```

a = 1

```

```

b = a

```

neues int-Objekt erstellt durch den Operator (1+1)

```

a = a+1

```

alte Referenz gelöscht durch Zuweisung (a=...)

Analog bei Listen

G. Zachmann Informatik 2 – SS 2011 Typsysteme 14

Strongly Typed / Weakly Typed

- Achtung: ex. keine einheitliche und strenge Definition!

- Sprache ist *strongly typed* (stark typisiert), wenn es schwierig/unmöglich ist, einen Speicherblock (der ein Objekt enthält) als anderen Typ zu interpretieren (als den vom Objekt vorgegebenen).
 - Uminterpretierung funktioniert nur mit Casts und Pointern, oder Unions (s. C aus Info 1)
- Sprache ist *strongly typed* (stark typisiert), wenn es wenig **automatische** Konvertierung (*coercion*) für die eingebauten Operatoren und Typen gibt.
 - Insbesondere: wenn es keine Coercions gibt, die Information verlieren

- Def. 1 ist sinnvoller, Def. 2 leider häufig bei Programmierern

G. Zachmann Informatik 2 – SS 2011 Typsysteme 15

Beispiele

- Verlust von Information in C++ (Definition 2):

<code>int i = 3.0;</code>	<code>i = 3.0</code>
In C++: liefert nur Warnung	In Python: Konstante ist Float, i ist nur Name, der an die Konstante gebunden wird
- Uminterpretierung in C++ (Definition 1):

<code>float f = 3; printf("%d\n", f);</code>	<code>f = 3.0 print "%d" % f</code>
In C++: höchstens Warnung	In Python: Konvertierung

G. Zachmann Informatik 2 – SS 2011 Typsysteme 16

- Beispiel: Visual Basic

```
Visual BASIC is a hybrid language. In addition to including statically typed variables, it includes a "Variant" data type that can store data of any type. Its implicit casts are fairly liberal where, for example, one can sum string variants and pass the result into an integer literal
```

- Bemerkungen:

- Mit OO-Sprachen kann man die Stärke der Typisierung gemäß Definition 2 (fast) beliebig weit abschwächen
- Resultat der automatischen Konvertierung ist nicht immer klar

- Beispiel:

```
string s = "blub";  
s += 3.1415;      // gültig  
s = 3.1415;      // dito
```

In einer hypothetischen,
eigenen String-Klasse in C++

Typing-Quadrant

- Orthogonalität:** Achtung: *static/dynamic typing* und *strong/weak typing* sind orthogonal zueinander!!

*) ohne C-style casts und ohne `reinterpret_cast`

- Häufig falscher Sprachgebrauch! ("strong" = "static und strong", oder gar: "strong" = "static")
- Achtung: weak \leftrightarrow strong ist eher Kontinuum

G. Zachmann Informatik 2 – SS 2011 Typsysteme 19

Hierarchie bzgl. Typisierung

```

graph TD
    A[Programmiersprachen] --> B[Untyped]
    A --> C[Typed]
    B --> B1[Beispiele: Assembler (z.B. TOY), (Perl)]
    C --> D[Statically typed]
    C --> E[Dynamically typed]
    D --> F[Declared types]
    D --> G[Inferred types]
    E --> E1[Beispiele: Python, Smalltalk, Ruby]
    F --> F1[Beispiele: C++, Java, Pascal]
    G --> G1[Beispiele: ML, Haskell]
  
```

G. Zachmann Informatik 2 – SS 2011 Typsysteme 20

Exkurs: Inferred Typing

- Typ-Deklarationen sind ...
 - lästig
 - unnötig
 - schränken ein
- Idee: der Compiler kann den Typ (fast immer) folgern
- Sehr simple Form von *Type-Inference* in Python:

```
x = 1          # 1 wird als int interpr.
x = 1.0        # 1.0 wird als float interpr.
```
- Und in C++:

```
float f = 3 / 2;
```

 - Compiler schließt, wir wollten Integer-Division haben

G. Zachmann Informatik 2 – SS 2011 Typsysteme 22

Exkurs: Inferred Typing

- Type-Inference in einer Funktion:

```
def fact( n ):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```
- **Angenommen**, Python wäre eine statisch typisierte Sprache mit Type-Inference:
 - Test 'n == 0' → n muß **int** sein,
 - Zeile 'return 1' → fact muß Funktion sein, die **int** liefert
 - Zusammen → **fact** muß Funktion von **int** nach int sein
 - Letzte Zeile: klassisches, statisches Type-Checking
- Macht nur für statisch typisierte Sprachen Sinn

G. Zachmann Informatik 2 – SS 2011 Typsysteme 23

Etwas komplexeres Beispiel

```
def mkpairs(x):  
    if x.empty():  
        return x  
    else:  
        xx = [x[0], x[0]]  
        xx.append(mkpairs(x[1:]))  
        return xx
```

Liefert neue Sequenz bestehend aus den Elementen 1-... von Sequenz x

Schlußfolgerungen:

1. `x.empty()` → `x` muß Sequenztyp sein, da `empty()` nur auf solchen definiert ist
2. `return x` → Funktion `mkpairs` muß Typ Sequenz→Sequenz haben
3. Rest ist Type Checking

Beispiel wo Inferenz nicht so ohne weiteres funktioniert:

```
def sqr(x):  
    return x*x
```

- Multiplikation verlangt, daß `x` vom Typ `Float` oder `Int` ist
- Also `sqr: Float → Float` oder `sqr: Int → Int`
- Nicht eindeutig definierter Typ
- `sqr` kann auch kein parametrisierter Typ sein, also vom Typ `T → T`, da ja nicht jeder beliebige Typ `T` erlaubt ist
- Lösung: mehrere, überladene Funktionen erzeugen, falls benötigt (à la Templates in C++)

Konvertierungen

- Totales Strong Typing gemäß Def. 2 → nur Operanden genau gleichen Typs erlaubt

```
int i = 1;
unsigned int u = i + 1; // error
float f = u + 1.0; // error
```

- Automatische Konvertierungen im Typsystem:
 - Begriffe: *automatic conversion*, *coercion*, *implicit cast*, *promotion*
 - Definition: *coercion* := autom. Konvertierung des Typs eines Ausdrucks zu dem Typ, der durch den Kontext benötigt wird.
 - Sprachgebrauch:
 - *coercion* bei built-in Types
 - *implicit cast* bei user-defined Types (Klassen)

G. Zachmann Informatik 2 – SS 2011 Typsysteme 26

- **Widening:**
 - Coercion vom "kleineren" Typ in den "größeren" Typ.
 - "kleinerer" Typ = kleinerer Wertebereich nicht notw. mehr Bits)
 - Beispiel:


```
int i = 1;
unsigned int u = i + 1; // 1 → 1u
float f = u + 1.0f; // 1u → 1.0f
```
- **Promotion-Hierarchie**, definiert im C++-Standard:



```
bool → char → int → unsigned int → long int →
float → double → long double
```

G. Zachmann Informatik 2 – SS 2011 Typsysteme 27

- **Narrowing:**
 - Coercion vom "größeren" in den "kleineren" Typ
 - Gefährlich: was passiert mit Werten außerhalb des kleineren Wertebereichs ?!
 - Beispiel:

```
float f = 1E30;
int j = f;
char c = j;
```
- Achtung: Coercion ist "lazy", d.h., so spät wie möglich, von innen nach außen!
- Beispiel:

```
float f = 1/2; // f == 0.0 !
```



G. Zachmann Informatik 2 – SS 2011 Typsysteme 28

"Typarithmetik" in Ausdrücken

- Zur Compile-Zeit muß der Baum eines Ausdruckes mit Typinformationen annotiert werden
 - Zum Type-Checking
 - Um die richtigen Assembler-Befehle zu erzeugen
- Beispiel:

```
1 + sin( Pi / 2 )
```

G. Zachmann Informatik 2 – SS 2011 Typsysteme 29

Funktionen eines (statischen) Typsystems

- Ungültige Operationen verhindern:
 - Beispiel: "hello world" / 3.0
 - Bei statischer / dynamischer Typisierung wird dies zur Compile-Zeit / Laufzeit abgefangen
- Optimierung: Compiler kann zur Compile-Zeit z.B. Operationen zusammenfassen
- Dokumentation: ein Typname kann (sollte) etwas über seine Bedeutung aussagen

```
typedef NumSecondsAfter1970 unsigned int;
struct ComplexNumber { float r, i; };
```

- Noch wichtiger sind aber gut gewählte Variablennamen! ist genauso wichtig wie eine ausführliche Beschreibung, wozu die Variable verwendet wird!

G. Zachmann Informatik 2 – SS 2011 Typsysteme 30

Charakteristika von Programmiersprachen

- Objekt-orientierte / funktionale / logische Sprache (s. Info 1)
- Statisch typisiert ↔ dynamisch typisiert
- Stark typisiert ↔ schwach typisiert
- Mit *type inference* ↔ mit deklarierten Typen
- Low level ↔ high level

- Compiliert ↔ interpretiert ("Skriptsprache")

G. Zachmann Informatik 2 – SS 2011 Typsysteme 31



- Systeme, die mit dynamisch-typisierten Sprachen gebaut werden, sind um ca. einen Faktor 5-10 kleiner (weniger LoC) als Systeme in statisch-typisierten Sprachen →
 - Weniger Code = weniger Bugs
 - Weniger Code = leichter zu warten (erweitern, modifizieren)
- Auch mit statisch-typisierten Sprachen benötigt man Unit-Tests (= Test-Code für einzelne Funktionen); mit diesen findet man aber auch schon die Bugs, die der Compiler durch Type-Checks gefunden hätte