



Informatik I

Das Typsystem (am Bsp. Python vs. C++)

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




Wozu Typen?

- Sicherheit der Software (Internet, Atomanlagen, etc.)
 - Wird die Platte formatiert, wenn man sich eine Webseite mit einem JPEG-Bild im Browser ansieht?

Microsoft Security Bulletin MS04-028
Buffer Overrun in JPEG Processing (GDI+) Could Allow Code Execution (833987)

Issued: September 14, 2004
Updated: October 12, 2004
Version: 2.0

Summary
Who should read this document: Customers who use any of the affected operating systems, affected software programs, or affected components.

Impact of Vulnerability: Remote Code Execution
Maximum Severity Rating: Critical

G. Zachmann Informatik 1 - WS 05/06 Typsysteme 2



Definitionen

- Ziel: **Semantische** Bedeutung einem Speicherblock zuordnen → Konzept des Typs bzw. Typsystems
- 3 Definitionen von **Typ** :
 - **Denotational:** *Typ* := Menge von Werten.
 - **Konstruktiv:** *Typ* :=
 - entweder: primitiver, eingebauter Typ (int, float, ...),
 - oder: zusammengesetzt aus anderen Typen.
 - **Abstrakt:** *Typ* := Interface, bestehend aus Menge von Operationen (Funktionen, Operatoren), die auf Werte dieses Typs angewendet werden können

G. Zachmann Informatik 1 - WS 05/06 Typsysteme 3



- **Type Error** := Operation (Operand / Funktion) ist für Werte (Variablen, Ausdrücke, Rückgabewerte) nicht definiert
 - Beispiel: Zahl / Uhrzeit
- **Type Checking** := Type-Errors finden, dann Fehlermeldung ausgeben oder auflösen
 - Beispiel: Typen zweier Operanden müssen gleich sein

G. Zachmann Informatik 1 - WS 05/06 Typsysteme 4

2 Arten von Type-Checking

- *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)
- *Type Inference* := Typ der Variablen wird aus dem Kontext hergeleitet; Typen dürfen unvollständig sein.

G. Zachmann Informatik 1 - WS 05/06 Typsysteme 5

Static Typing (C++)

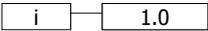
- Konsequenz:
 - Compiler muß zu jeder Zeit Typ eines Wertes im Speicher kennen
 - Relativ einfach machbar, wenn Variablen einem Speicherblock fest zugeordnet sind
 - Typ(Variable) = Typ(Speicherbereich)
 - Bessere Performance (Compiler generiert sofort richtigen Code)
- Konsequenz für die Sprache bzgl. 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 1 - WS 05/06 Typsysteme 6

Beispiel:

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



- Problem: Typ von Speicherblock wird durch Typ der darübergelegten Variable bestimmt!

G. Zachmann Informatik 1 - WS 05/06 Typsysteme 7

Dynamic Typing (Python)

- Konsequenzen:
 - Typ von Variable/Wert wird zur Laufzeit (jedesmal) überprüft
 - Werte im Speicher müssen *unveränderlichen* Type-Tag haben
 - Variable muß nicht an festen Speicherbereich gebunden werden
 - Welcher Operator, wird jeweils zur Laufzeit entschieden
 - "Generic Programming" wird einfacher
 - Variablen müssen nicht deklariert werden
 - Wesentlich kürzere Compile-Zeiten, Performance penalty
 - Debugger mit höherer Funktionalität ("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 1 - WS 05/06 Typsysteme 8

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 1 - WS 05/06 Typsysteme 9

Ändern eines Integers

Achtung: manche Operationen erzeugen eine Kopie!

```
a = 1
```

```
b = a
```

neues int Object erstellt durch Addieren des Operators (1+1)

```
a = a+1
```

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

Analog bei Listen

G. Zachmann Informatik 1 - WS 05/06 Typsysteme 10

Strongly Typed / Weakly Typed

Achtung: 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
- 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 häufig bei Programmierern

G. Zachmann Informatik 1 - WS 05/06 Typsysteme 11

Beispiele

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

```
int i = 3.0;
```

In C++: liefert nur Warning

```
i = 3.0
```

In Python: Konst. ist Float, i ist nur Name, der an Konst. gebunden wird

Uminterpretierung in C++ (Definition 1):

```
float f = 3;
printf( "%d\n", f );
```

In C++: nur Warning

```
f = 3.0
print "%d" % f
```

In Python: Konvertierung

G. Zachmann Informatik 1 - WS 05/06 Typsysteme 12

- Mit OO-Sprachen kann man die Stärke der Typisierung gemäß Definition 2 (fast) beliebig weit "aufweichen".
Bsp.: Resultat der automatischen Konvertierung nicht klar:

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

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

```
s = "12"
s += 1 # Fehlermeldung
```

Mit eingebautem String-Typ in Python

G. Zachmann Informatik 1 - WS 05/06
Typsysteme 13

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↔strong ist eher Kontinuum

G. Zachmann Informatik 1 - WS 05/06
Typsysteme 14

Hierarchie bzgl. Typisierung

```

graph TD
    A[Programmiersprachen] --> B[untyped]
    A --> C[typed]
    B --- B_ex[Beispiele: Assembler, Perl]
    C --> D[Statically typed]
    C --> E[Dynamically typed]
    D --> F[Declared types]
    D --> G[Inferred types]
    E --- E_ex[Beispiele: Python, Java]
    F --- F_ex[Beispiele: C++, Java, Pascal]
    G --- G_ex[Beispiele: ML, Haskell]
  
```

G. Zachmann Informatik 1 - WS 05/06
Typsysteme 15

Exkurs: Inferred Typing

- Type-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 1 - WS 05/06
Typsysteme 17

▪ 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:

- 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 1 - WS 05/06 Typsysteme 18

▪ 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

▪ Schlußfolgerung

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

G. Zachmann Informatik 1 - WS 05/06 Typsysteme 19

▪ 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 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++)

G. Zachmann Informatik 1 - WS 05/06 Typsysteme 20

▪ Konvertierungen

▪ Absolutes 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 Ausdruckes 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 1 - WS 05/06 Typsysteme 21

- Widening:
 - Coercion vom "kleineren" Typ in den "größeren" Typ.
 - "kleinerer" Typ = kleinerer Wertebereich
 - 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 1 - WS 05/06 Typsysteme 22

- 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 1 - WS 05/06 Typsysteme 23

"Typarithmetik" in Ausdrücken

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


```
graph TD
    Root["+ float"] --- L["1.0 float"]
    Root --- R["sin float"]
    L --- LKonv["Konv."]
    LKonv --- LInt["1 int"]
    R --- RDiv["/ float"]
    RDiv --- RPi["Pi float"]
    RDiv --- RTwo["2.0 float"]
    RTwo --- RKonv["Konv."]
    RKonv --- RInt["2 int"]
```

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

G. Zachmann Informatik 1 - WS 05/06 Typsysteme 24

Funktionen eines Typsystems

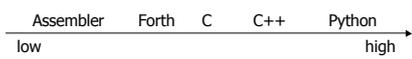
- Sicherheit: ungültige Operationen verhindern
 - Beispiel: "hello world" / 3.0
- Optimierung
- Dokumentation: ein gut gewählter Typname ist genauso wichtig wie eine ausführliche Beschreibung, wozu der Typ verwendet wird!


```
typedef NumSecondsAfter1970 unsigned int;
struct ComplexNumber { float r, i; };
```
- Abstraktion
- Wesentlicher Bestandteil jedes APIs (von z.B. Libraries)

G. Zachmann Informatik 1 - WS 05/06 Typsysteme 25

Charakteristika von Programmiersprachen

- Statisch typisiert ↔ dynamisch typisiert
- Stark typisiert ↔ schwach typisiert
- Mit *type inference* ↔ mit deklarierten Typen
- Low level ↔ high level



- Objekt-orientierte / funktionale / logische Sprache
- Compiliert ↔ interpretiert (Skriptsprache)