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

Definitionen

- Ziel: Semantische Bedeutung einem Speicherblock zuordnen → Konzept des Typs bzw. Typsystems
- 3 Definitionen von *Typ* :
  - Denotational: *Typ* := Menge von Werten.
  - Konstruktiv: *Typ* :=
    - entweder: primitver, 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

- *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

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.

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++:

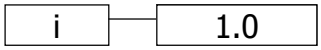
```
Type Variable;
```

  - Beispiele:

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

- Beispiel:

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



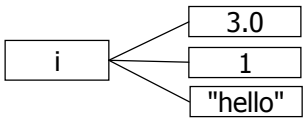
- Problem: Typ von Speicherblock wird durch Typ der darübergelegten Variable bestimmt!

Dynamic Typing (Python)

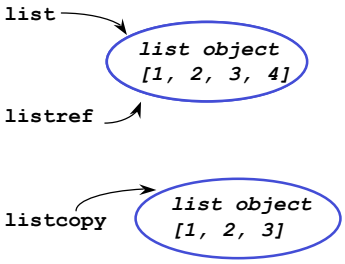
- Konsequenzen:
  - →Typ von Variable/Wert wird zur Laufzeit (jedesmal) überprüft
  - →Werte im Speicher müssen *unveränderlichen* Type-Tag haben
  - →Variable muß nicht an festen Speicherbereich gebunden werden
  - →Welcher Operator, wird jeweils zur Laufzeit entschieden
  - →"Generic Programming" wird 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

- Beispiele:

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



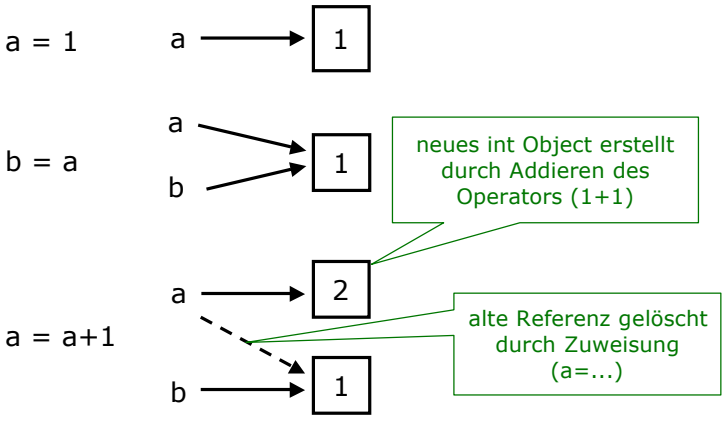

```
list = [1,2,3]
listref = list
listcopy = list[:]
list.append( 4 )
```



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

Ändern eines Integers

- Achtung: manche Operationen erzeugen eine Kopie!



- Analog bei Listen

Strongly Typed / Weakly Typed

- Achtung: keine einheitliche und strenge Definition

1. 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
2. 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

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

- 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
```

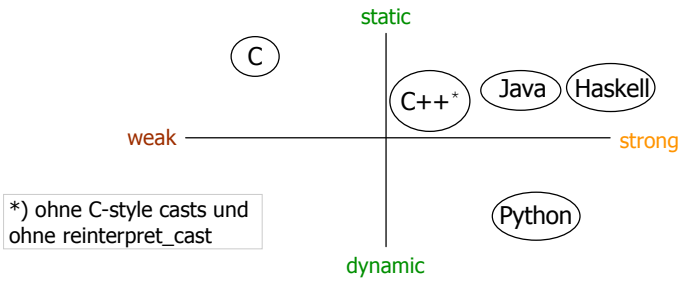
In einer hypothetischen, eigenen String-Klasse in C++

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

Mit eingebautem String-Typ in Python

Typing-Quadrant

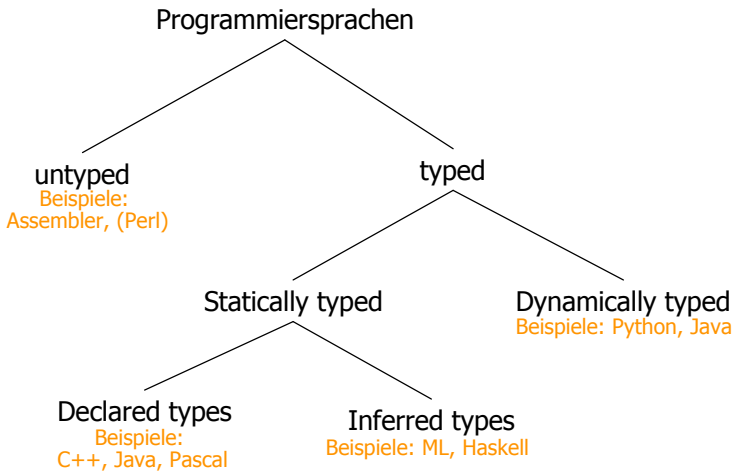
- Orthogonalität: Achtung: *static/dynamic typing* und *strong/weak typing* sind orthogonal zueinander!!



*) 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

Hierarchie bzgl. Typisierung



- Beispiel für static weak typing (in C++):

```
float f = 1.9;
printf( "%d\n", f );
```

  - Moderne Compiler geben hier immerhin Warning aus
- Beispiel für dynamic weak typing (immer noch C++):

```
float f = 1.9;
char format[] = "%d\n";
printf( format, f );
```

  - Format-String-Interpreter betrachtet Typ erst zur Laufzeit

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

- 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

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

- 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++)

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)

- 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`

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

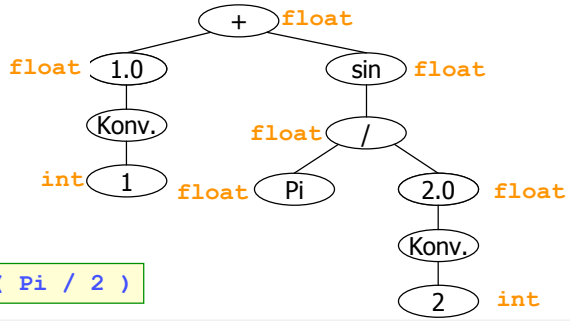
```
float f = 1E30;
int j = f;
char c = j;
```

- Achtung: Coercion ist "lazy", d.h., so spät wie möglich, von innen nach außen!
- Beispiel:

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

"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 )
```

Funktionen eines Typsystems

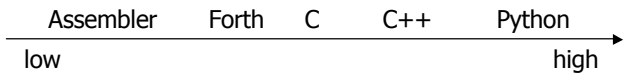
- Sicherheit: ungültige Operationen verhindern
  - Beispiel: "hello world" / 3.0
- Optimierung
- Dokumentation: ein gut gewählter Typname ist genauso wichtig wie eine ausührliche Beschreibung, wozu der Typ verwendet wird!

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

- Abstraktion
- Wesentlicher Bestandteil jedes APIs (von z.B. Libraries)

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)