Technische Universität Braunschweig

Institute of Operating Systems and Computer Networks
Reliable System Software

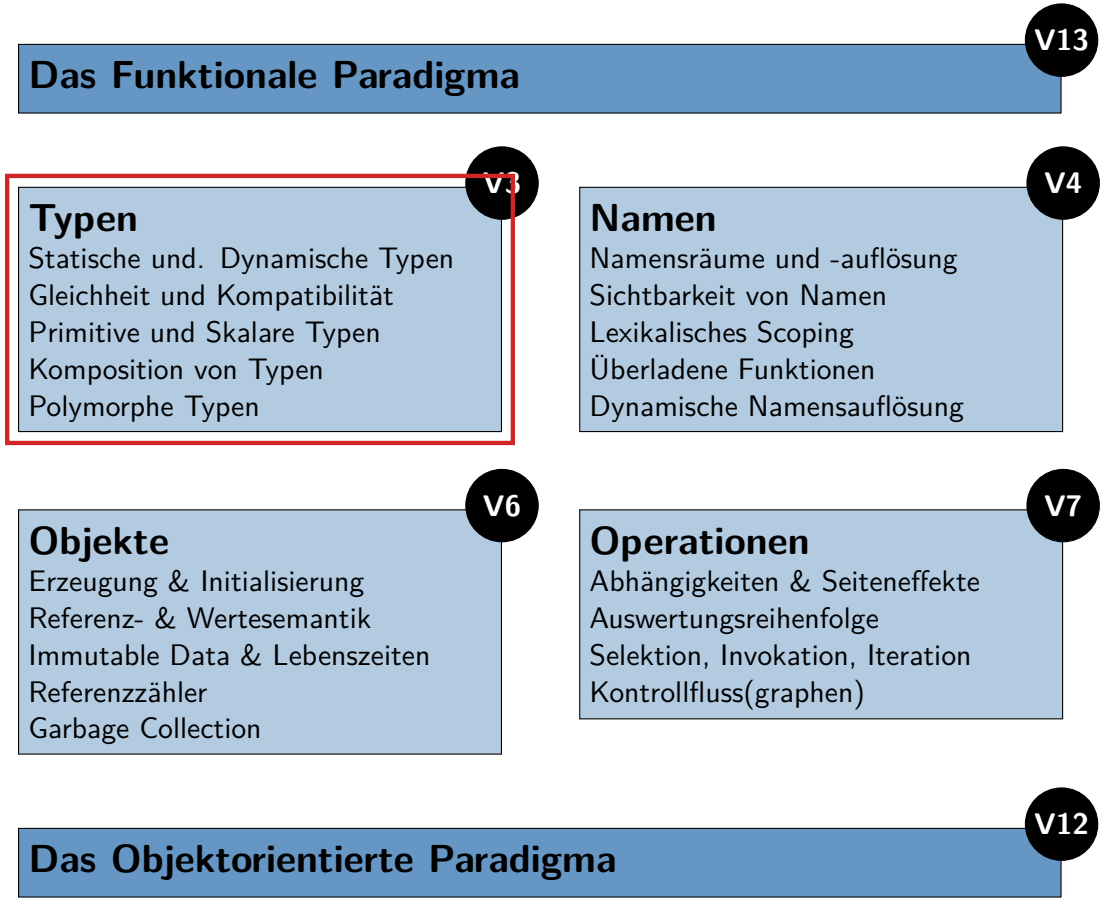
Programmiersprachen und Übersetzer

03 - Typen

Christian Dietrich

Sommersemester 2025

Wieso sollte ich mich mit Typen beschäftigen?



- Typen sind das Rückgrat moderner Programmiersprachen.
  - Je mehr das Typsystem abfängt, desto weniger muss man denken.
  - Das Typsystem ist eine wichtige Quelle für automatische Optimierungen.

Was bieten einem Typen?

- Typen bieten **Kontext** und transportieren Informationen quer zum AST.
  - Ohne Typen: Wir müssten die Operation immer ganz genau angeben.

```
var a,b; ... float32_add(a, b)
```

```
int32_t a, b; .... a+b
```

  - Mit Typen: Der Übersetzer wählt die passende Operation entsprechend.

- Typen **schützen vor Bugs** und verhindern invalide Aktionen.
  - `Cat k1; ... k1 = Dog();` ⇒ „Type mismatch: Cannot assign 'Dog' to 'Cat'"
  - Mit Typen macht ein Programmierer Zusicherungen über Objekte und Variablen.

- Typen **verbessern die Lesbarkeit** von Code und sind Dokumentation.
  - Mit statischen Typen kommunizieren wir Intention und angedachte Verwendung.
  - Eine gute API ist bereits anhand der Typnamen verständlich:

```
____ regex_match(________, ___________, ______________)
```

Was bieten einem Typen?

- Typen bieten **Kontext** und transportieren Informationen quer zum AST.
  - Ohne Typen: Wir müssten die Operation immer ganz genau angeben.

```
var a,b; ... float32_add(a, b)
```

```
int32_t a, b; .... a+b
```

  - Mit Typen: Der Übersetzer wählt die passende Operation entsprechend.

- Typen **schützen vor Bugs** und verhindern invalide Aktionen.
  - `Cat k1; ... k1 = Dog();` ⇒ „Type mismatch: Cannot assign 'Dog' to 'Cat'"
  - Mit Typen macht ein Programmierer Zusicherungen über Objekte und Variablen.

- Typen **verbessern die Lesbarkeit** von Code und sind Dokumentation.
  - Mit statischen Typen kommunizieren wir Intention und angedachte Verwendung.
  - Eine gute API ist bereits anhand der Typnamen verständlich:

```
bool regex_match(regex_t*, const char*, regex_flags_t)
```

⇒ Typen sind Freunde und Helfer in der Not!

Was bedeutet ein Typ? Wie interpretieren wir Typen?

- Strukturell: Es gibt eingebaute und daraus kombinierte Typen.
  - Eingebaute Typen: `bool`, `int`, `int32_t`, `float`, `double`, `char`, ...
  - Kompositionen: Referenzen (`char *` = Pointer auf `char`), Arrays (`int[10]`).
  - „Wie baue ich meine Datenstrukturen aus einfacheren Datentypen?“
- Denotationell: Typen *T* sind Mengen von Objekten.
  - Der Typ `uint16_t` *ist* die Menge aller positiver Ganzzahlen $0 \leq x < 65536$.
  - Typen sind Prädikate über die Mengen-Mitgliedschaft `T :: object ↦ bool` .
  - „Welche Regeln und Invarianten gelten für die Objekte hinter diesem Typ?“
- Abstraktional: Ein Typ ist ein Interface von konsistenten Operationen.
  - `int32_t` ist das Interface $\{+_{int32_t}, -_{int32_t}, *_{int32_t}, \ldots\}$.
  - Der Typ `stack_t` besteht aus drei Operationen: {**`push()`**, **`pop()`**, **`empty()`**}.
  - „Mit welchen Operationen kann ich Objekte dieses Typs manipulieren?“

Alle drei Sichtweisen sind nützlich und keine ist richtiger als die andere.

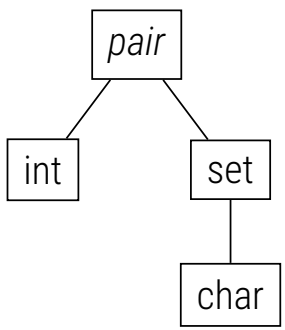
Typausdrücke: Strukturelle Notation für Typen

- Wir können Typen strukturell als Typausdrücke aufschreiben.
  - Komplexere Typen durch die wiederholte Anwendung von Typkonstruktoren
    T ist die Menge aller validen Typausdrücke. (Rekursiv!)

$$\text{pair} :: T \times T \mapsto T$$

  - „Ein Paar aus Ganzzahl und einer Menge von Zeichen“
  - Mit Typausdrücken wird das formeller: pair(int, set(char))

- Typausdrücke sind Bäume und Konstruktoren erzeugen innere Knoten.
  - Typausdrücke sind strukturell sehr ähnlich zu AST-Unterbäumen.
  - Umwandlung von AST-Knoten durch rekursiven Besucher
  - Nicht nur built-in Typen als Blätter: array(int, 2)



Äquivalenz von Typen

- Oft gestellte Frage bei Sprachen: Wann sind zwei Typen äquivalent?
  - **Äquivalenz ist**: Ununterscheidbarkeit unter einem bestimmten Aspekt
  - Beispiel: ”abc” und ”foo” sind äquivalent im Speicherverbrauch.
  - Unterschiedliche Sprachen verwenden unterschiedliche Äquivalenzen!

- Strukturelle Äquivalenz: Gleicher Typausdruck = Gleicher Typ
  - Typnamen sind nur Abkürzung für den definierten Typausdruck.

```pascal
type R1 = record
   a, b : integer;
end;

type R2 = record
   a : integer;
   b : integer;
end;
```

Pascal

Beide Definitionen liefern in Pascal:

record((”a”, integer), (”b”, integer))

```pascal
type student = record
   name: string;
   age: integer;
end;

type school = record
   name: string;
   age: integer;
end;
```

Pascal

Mit reiner struktureller Äquivalenz sind beide Typen nicht unterscheidbar. Sollten sie das sein?

Namensäquivalenz von Typen

- Rekursive Typdefinitionen sind ein Problem für strukturelle Äquivalenz.
  - Ersetzt man jedes Auftreten eines Typnamens durch den Typausdruck, kommt man zu einer endlos tiefen Verschachtelung:

```c
struct list {
  struct list *next;
};
```

C

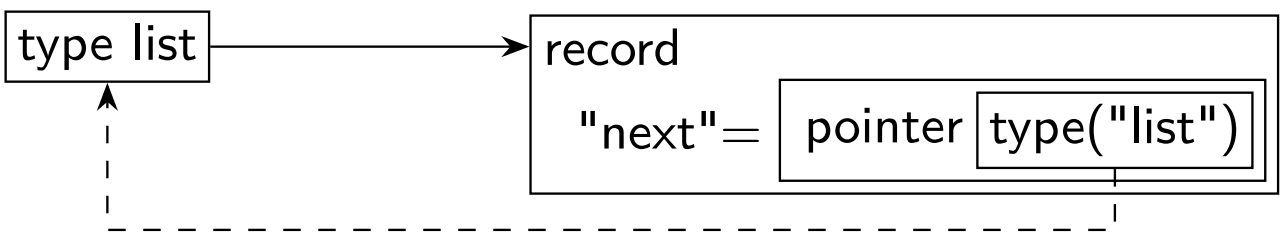
```
record(("next", pointer(
  record(("next", ...)))
))
```

- Namensäquivalenz: Jede Typdefinition erzeugt einen neuen Typen.

  Wenn der Entwickler den Aufwand treibt zwei Definitionen zu schreiben, wird er schon Unterschiedliches gemeint haben.

  - Berechnung der Typausdrücke stoppt an benannten Typen.

type list = record(("next", pointer(type("list"))))



Typnamen und Typaliase

- Bei Namensäquivalenz: Kann ich ein Typalias erzeugen?
  - Typalias: Zweiter Name für den gleichen Typen.
  - `typedef old_type new_type`: Typdefinition oder Typalias?

```c
typedef double celsius_t;
typedef double fahrenheit_t;
void heating_set(celsius_t);

// Typdefinition:   Übersetzerfehler
// Typalias:        Gegrillte Zimmerpflanzen
fahrenheit_t target = (fahrenheit_t) 100;
heating_set(target);
```

C

  - C nutzt nur für `struct`s Namensäquivalenz, ansonsten Strukturäquivalenz.

- Manche Sprachen: sowohl Typaliase als auch **abgeleiteten Typen**

```ada
subtype mode_t is integer;         -- Typalias für Integer
type celsius_t is new integer;     -- Eigener Typ
type fahrenheit_t is new integer; -- Eigener Typ
```

ADA

Skalare Typen: Aufzählungen und Zahlen

- Skalare Typen beschreiben nicht-weiter zerlegbare Objekte
  - Kein Teil eines skalaren Objekts macht eigenständig Sinn.
  - Diskreter Wertebereich, oft mit totaler Ordnung

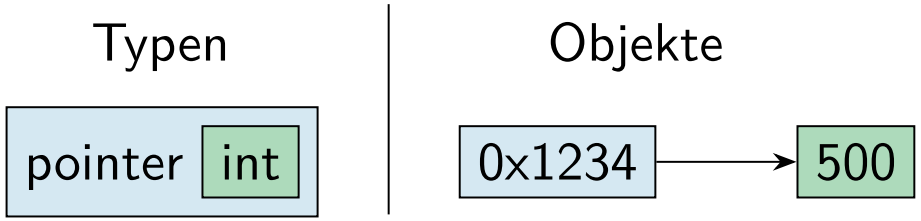
- Aufzählungen oder Enumerations (`enum{wert0, wert1, ...}`)
  - Explizite Angabe des Werteberreichs: `enum {red, green, blue}`
  - Wahrheitswerte sind ein häufiger Spezialfall: `typedef enum{false, true} bool;`
  - Closed-World Assumption ist manchmal schädlich: `enum {female, male}`

- Numerische Typen: Ganzzahlen, Gleitkommazahlen, und Dezimalzahlen
  - Endlicher, aber meist sehr großer Wertebereich (`uint32_t` : $0 \leq n < 2^{32}$)
  - Speicherung mit (`signed`) und ohne (`unsigned)` Vorzeichen
  - Sehr oft Übereinstimmung mit den Wortbreiten der realen Maschine
  - Manche Sprachen erlauben explizite Einschränkung des Wertebereichs:

```
type water_temperature = 273..373 (* Kelvin *)
```
Pascal

Skalare Typen: Zeiger und Funktionen

- Zeiger sind gespeicherte Referenzen auf Objekte vom Basistyp.



  - Minimales Interface (abstraktional): dereference `::` pointer(T) $\mapsto$ T
  - Verschiedene Zeigertypen geben unterschiedliche Referenzen.
    - Universeller Zeiger in C garantiert nichts (`T*`)
    - C++-Referenzen zeigen immer auf valides Objekt (`T&`).
    - Für jedes Objekt maximal ein **eindeutiger Zeiger** (`std::unique_ptr<T>`)

- Funktionstypen beschreiben den Rückgabetyp und die Parametertypen
  - Strukturell: Typausdruck für `strlen` ist func(size_t, pointer(char))
  - Abstraktional: call `::` func(R,…) $\times$ … $\mapsto$ R („Funktionen sind aufrufbar“)
  - Denotational: Maschinencode der Funktion ist ein Objekt dieser Typen.

Zusammengesetzte Typen: Records

```c
typedef struct {
  int    feld_1;
  double feld_2;
} foo_t;
```

```c
union {
  int    feld_1;
  double feld_2;
};
```

- Records subsumieren endlich viele jedoch heterogen typisierte Elemente
  - Die einzelnen Elemente heißen Felder und haben einen Typ und einen Namen.
  - Namen müssen normalerweise statische Bezeichner sein.
  - Minimales Interface: dot_operator :: record(..., (name, T), ...) $\times$ name $\mapsto$ T
- Records kommen in fast allen Sprachen in diversen Geschmäckern vor:
  - Java Klassen sind auch Records, aber noch viel viel mehr...
  - Fortran90: Der Übersetzer darf Felder umsortieren, um Speicher zu sparen.
  - Haskell: Tupel sind so etwas wie Records ohne Feldnamen.
- Bei varianten Records (Unions) ist maximal ein Feld gültig.
  - Alle Felder des Records nehmen den gleichen Speicher ein.
  - Unions sind oft die einzige (erlaubte) Möglichkeit das Typsystem zu umgehen.

Zusammengesetzte Typen: Sequenzen

```haskell
a = [2, 3] -- list(int)
b = 1 : a  -- = cons(1, [2, 3])
T = tail b -- list(int), [2,3]
H = head b -- int, 1
```

Haskell

```c
struct list {
  int           value;
  struct list *next;
};
```

C

- Sequenztypen sind homogen getypte und (un)geordnete Container.
  - Bekanntester Sequenztyp ist die (einfach verkettete) Liste: list(T)
  - Minimales Interface: Nur Zugriff auf das erste Element und die Restelemente:
    head :: list(T) ↦ T tail :: list(T) ↦ list(T)
- Listen sind rekursiv definiert und werden oft mit Records nachgebaut
  - Die Restliste ist selbst wieder eine Liste oder `nil` (leere Liste)
  - Mittels rekursivem Record kann man Listentypen emulieren
- Abweichungen von strikter Homogenität und Einfügeordnung möglich
  - Python-Listen (`[1, "x", 3.4]`) garantieren nur minimalen Typ: list(object)
  - Sets sind ungeordnete Mengen (Java: `HashSet<Integer>`)

Zusammengesetzte Typen: Abbildungen

```c
// array(int, 10), map(int, int)
int rgb[10];
// map(int, map(int, char))
char matrix[20][20];
```

C

```cpp
std::map<std::string, int> x;
x["y"] = 23;
x["z"] = x.at("y") + 100;
```

C++

- Abbildungstypen: Zuordnung zwischen einem Index- und einem Wertetyp
  - Indexmenge und Wertemengen sind homogen typisiert
  - Vordefiniert-beschränkte oder dynamisch-wachsende Abbildungsgröße
  - Wahlfreier Zugriff mit dynamischem Index `get :: map(K, V) × K ↦ V`
- Arrays sind die einfachste (und effizienteste) Abbildung.
  - Indextyp ist immer `int`, Indexmenge ist zusammenhängend und beschränkt
  - Effizienter Elementzugriff und dichte Speicherung möglich
  - Manche Sprachen erlauben variabel lange Zeilen in mehrdimensionalen Arrays.
- Assoziative Arrays wachsen und erlauben beliebige Indextypen.
  - Namen: Dictionary (Python), Hash (Ruby), Table (Lua), Map (Rust, C++)
  - In Skriptsprachen wird oft die Homogenität aufgegeben: `map(object,object)`

Polymorphismus macht Code flexibel

Bisher: Zwei Typen sind entweder gleich oder haben nichts gemein.

- Typen haben jedoch oft gemeinsame Aspekte und sind irgendwie ähnlich
  - Strukturell: `struct point2D {int x,y}` `struct point3D {int x,y,z}`
  - Denotational: uint8_t $\subset$ uint16_t $\subset$ uint32_t $\subset$ uint64_t
  - Abstraktional: lengthIntList `::` list(int) $\mapsto$ int lengthFloatList `::` list(float) $\mapsto$ int
- Monomorphe Typsysteme beachten nur Äquivalenz und sind **inflexibel**
  - Codeduplikate: `printX_2D()`, `isEven_u8()`, `lenghtFoobarList()`
  - Ständige explizite Konvertierungen zwischen ähnlichen Typen:
    ```
    struct point3D p; struct point2D p_ = {p.x, p.y};
    ```
- Polymorphismus erlaubt es ähnliche anstatt nur gleiche Typen zu haben
  - Polymorpher Code darf sich nur auf die Gemeinsamkeiten verlassen.
  - Subtypen: „Jeder 3D-Punkt ist auch ein 2D-Punkt“
  - Parametrisch: „Funktion verarbeitet alle Listen unabhängig vom Elementtyp“

Typkompatibilität

- Neben Typäquivalenz brauchen wir auch noch die Typkompatibilität.

```C++
S x;
T y = x;
```

Substitution: Kann ich ein `S` da einsetzen wo ein `T` gefordert ist?

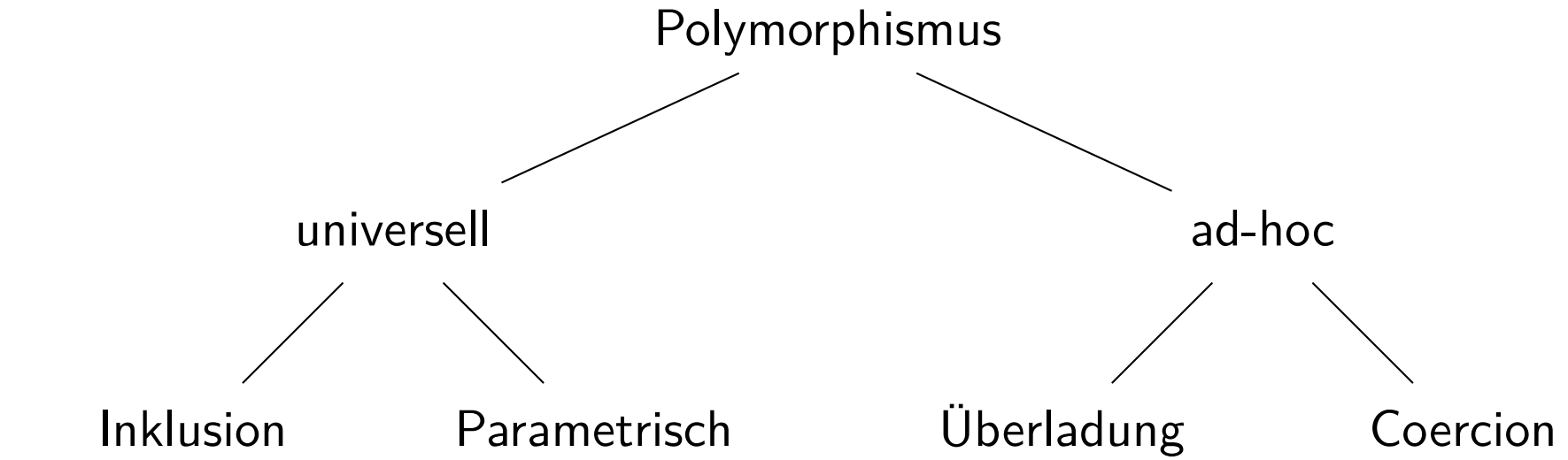
- „Sicher“ hat dabei viele unterschiedliche Interpretationen.
  - Speicher: Die S-Datenstruktur ist kompatibel zur T-Datenstruktur. *strukturell*
  - Semantik: S verhält sich in jedem Kontext entsprechend wie T. *denotationell*
  - Abstraktionen: S kann in jedem Kontext wie T verwendet werden. *abstraktional*

- Kein Typsystem kann alles abfangen und komplette Sicherheit bieten.

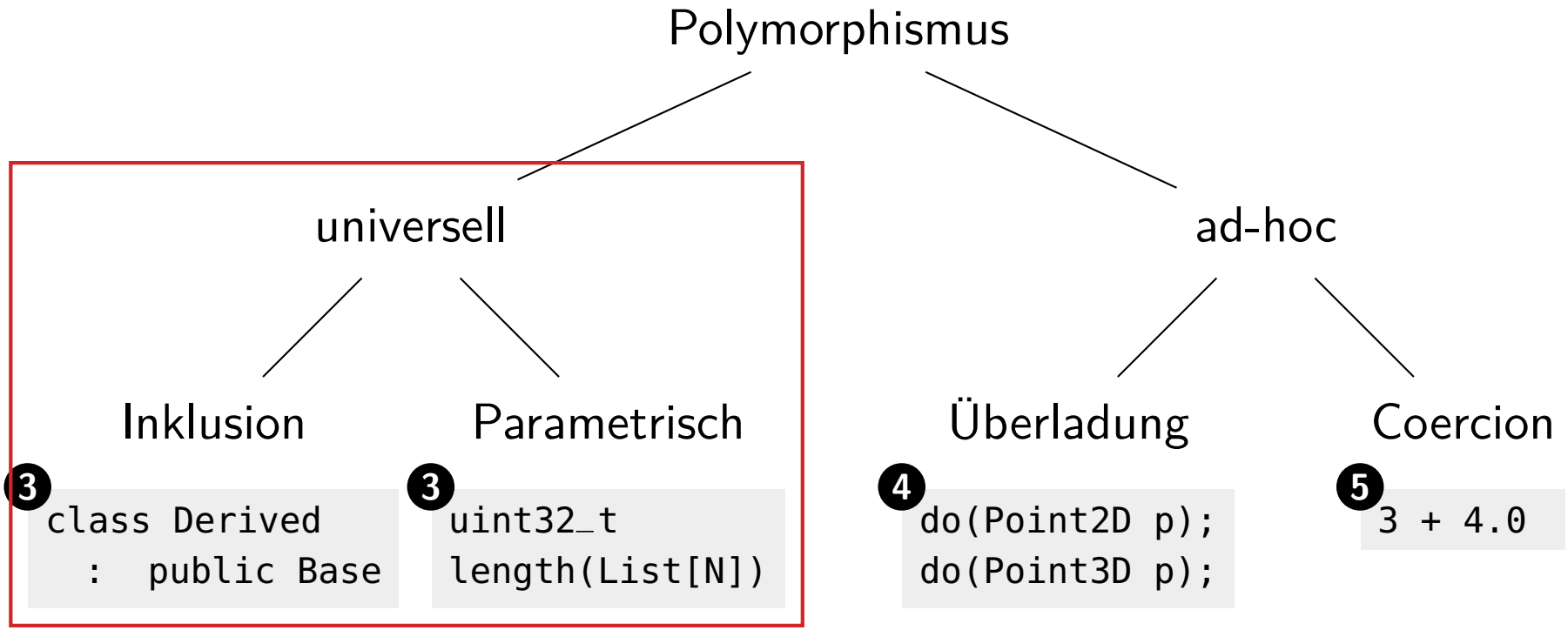
```Base.java
class Base {
  public MyObj alloc() {
    return new MyObj(23);
  };
}
```

```Derived.java
class Derived extends Base {
  public MyObj alloc() {
    eraseUserDisk();
    return null;
  };
}
```

Arten des Polymorphismus



Arten des Polymorphismus



Inklusion: Subtypen

Ziel: Behandlung unterschiedlicher Typen **ohne** Konvertierung

Der Code für Supertypen soll auch Subtypen verarbeiten können.

Subtypen sind Untermengen: $S \subseteq T$

Da ein Typ T die Menge seiner Objekte (denotational), ist ein Subtyp S eine Untermenge vom (Super)typ T. Jedes S ist auch ein T.

- Subranges: Subtyping durch Einschränkung des Werteberreichs
  - Ada: `subtype percent_t is integer range 0..100;`
  - Wer `integer` verarbeiten kann, kann auch `percent_t` verarbeiten.

```
type percent_t = 0..100;
procedure PrintInteger(j : integer);
...
var completed : percent_t;
completed := 37;
PrintInteger(completed);
```

Pascal

Inklusion: Subtyping durch Vererbung

Technische Universität Braunschweig

Man proklamiert, intentional, dass es eine Subtyp-Beziehung gibt!
Was muss daraus folgen um Ersetzbarkeit zu garantieren?

- Zur Erhaltung der Ersetzbarkeit müssen alle Felder geerbt werden:

```C++
struct Base {
  int x;
};
```

type Base = record(("x", int))

```C++
struct Derived : public Base {
  int y;
};
```

type Derived = record(("x", int), ("y", int))

- Die Subtypen müssen das Interface der Supertypen anbieten:

```Java
void print(Base o) {
  o.printX();
}
...
print(new Base());
print(new Derived());
```

Parametrisch: Generische Typen

Bisher: Alle Typkonstruktoren waren vorgegeben (`record()` , `map()`).

- Parametrischer Polymorphismus erlaubt eigene Konstruktoren.
  - Erinnerung: Konstruktoren sind Typ$^n \to$Typ Funktionen: $\text{map} :: T \times T \mapsto T$
  - Wir führen Typvariablen in der Sprache ein, um Typausdrücke zu „speichern“.
  - Trivialste Sicht: Eigene Typkonstruktoren sind Abkürzungen:

```java
public class Pair<A> {
  public A f;
  public A s;
}
```
Java

```
Pair(A) := record(("f", A), ("s", A))
```

```
type IntPair = Pair(int)
             = record(("a", int"), ("b", int))
```

- Mit eigenen Konstruktoren lässt sich Schreibarbeit sparen.
  - `class intLst { int v; intLst* n }`
  - `class charLst { char v; charLst* n }`
  - `class boolLst { bool v; boolLst* N }`

```cpp
template<typename T>
class List<T> {
  T        v;
  List<T>* n;
};
```
C++

Parametrisch: Typkonstruktoren werden selbst Typen

Generische Typen als First-Class Citizens

Die Stärke von parametrischem Polymorphismus ist, dass generische Typen nicht nur Abkürzungen sind, **sondern** vollwertige Typen.

- Variablen, Parameter und Rückgabewerte mit generische Typen
  - Listenlängen sind unabhängig vom Elementtyp: $\text{length} :: \text{List[N]} \mapsto \text{int}$
  - Mehrfach auftretende Typvariablen propagieren Typen zur rechten Seite: $\text{concat} :: \text{List[N]} \times \text{List[N]} \mapsto \text{List[N]}$
- Alle **konkreten Instanzen** sind kompatibel zum generischen Typen.
  - Im Gegensatz zu Vererbung gibt es (potentiell) unendlich viele Instanzen.
  - `Pair(int)`, `Pair(bool)` sind kompatibel mit `Pair(A)`.
  - Wie die einzelnen Instanzen zueinander stehen, ist deutlich komplizierter.

Instanztypen: Kovarianz, Kontravarianz, und Invarianz

Bei der Kombination von Subtyping und parametrischen Polymorphismus können die Instanztypen `K[N]` unterschiedlich kompatibel sein:

- Kovarianz: Kompatibilität läuft entlang der Spezialisierungsrichtung.
  - Möglich, wenn `K[N]` nur Quelle für Instanzen von `N` ist.
  - `Baker[Cake]` ist Subtyp von `Baker[Bread]`.
  - Jedoch: `Baker[Bread]` ist inkompatibel mit `eatAll :: Baker[Cake]` $\mapsto \bot$

`Cake` ist ein spezielles (erbt von) `Bread`.
`Baker[T]` bäckt Gebäck vom Typ T.
`Customer[T]` kauft Gebäck vom Typ T.

Instanztypen: Kovarianz, Kontravarianz, und Invarianz

Bei der Kombination von Subtyping und parametrischen Polymorphismus können die Instanztypen `K[N]` unterschiedlich kompatibel sein:

- Kovarianz: Kompatibilität läuft entlang der Spezialisierungsrichtung.
  - Möglich, wenn `K[N]` nur Quelle für Instanzen von `N` ist.
  - `Baker[Cake]` ist Subtyp von `Baker[Bread]`.
  - Jedoch: `Baker[Bread]` ist inkompatibel mit `eatAll :: Baker[Cake] ↦ ⊥`
- Kontravarianz: Kompatibilität läuft entgegen der Spezialisierung.
  - Möglich, wenn der `K[N]` nur Senke für Instanzen von `N` ist.
  - `Customer[Bread]` ist Subtyp von `Customer[Cake]`.
  - Jedoch: `Customer[Cake]` ist inkompatibel mit `feed :: Customer[Bread] ↦ ⊥`
- Invarianz: Instanztypen sind inkompatibel zueinander.
  - Fallback, wenn `K[N]` sowohl Quelle als auch Senke ist.
  - Alle Instanzen von (veränderbaren) `Tüte[N]` sind inkompatibel zueinander.

„Typen von Typen“: Parameterbeschränkungen

`Baker[Car]` ist sinnlos: Autos backt man nicht bei 180°C Umluft.

- Generische Typen können Einschränkungen für ihre Parameter haben.
  - Meta-Typen man kann strukturell, abstraktional und denotional auffassen.
  - Strukturell: Typparameter muss bestimmtes Feld haben.
  - Abstraktional: Typparameter muss ein bestimmtes Interface haben.
  - Denotional: Typparameter muss Subtyp von bestimmter Klasse sein.

Beispiel: Generics bei Java

```
class Baker<T extends Bread> {
  void bake(T b) {
    oven.heat(b.degrees());
  }
}
```

Ein `Baker` kann nur Unterklassen von `Bread` backen.

Beispiel: Type Traits in Rust

```
pub trait Bakable {
  fn degrees(&self) -> f64;
}

impl Bakable for Cake {
  fn degrees(&self) -> f64 {
    return 180;
  }
}
fn bake<T: Bakable>(b: T);
```

Typsysteme: Verwendung von Typen in der Sprache

- Das **Typsystem** einer Sprache beschreibt die Verwendung der Typen.
  - Welche eingebauten Typen und Typkonstruktoren stehen bereit?
  - Wie ist die Äquivalenz und Kompatibilität von Typen definiert?
  - Ist das Typsystem monomorph oder enthält es Polymorphismus?

- Typsysteme haben weitere charakteristische Eigenschaften:
  - **Typsicherheit**: Kann ich das Typsystem (unbemerkt) umgehen?
  - **Dynamik**: Zu welchem Zeitpunkt werden die Typen geprüft?
  - **Automatisierung**: Wie viel Arbeit nimmt der Übersetzer uns ab?

Handreichung für den effektiven Neuling

Um eine Sprache effektiv zu erlernen, muss ich das Typsystem der neuen Programmiersprache prägnant beschreiben und mit bereits bekannten Sprachen vergleichen können.

Typsysteme: Starke vs. Schwache Typisierung

"Wie strikt forciert die Sprache ihr Typsystem?"

- Das Typsystem einer Programmiersprache heißt stark typisiert, wenn:
  1. Es gibt unterschiedliche Typen.
  2. Implizite (automatische) Konvertierung nur zwischen ähnlichen Typen
  3. Explizite Typkonvertierungen sind im Regelfall notwendig.

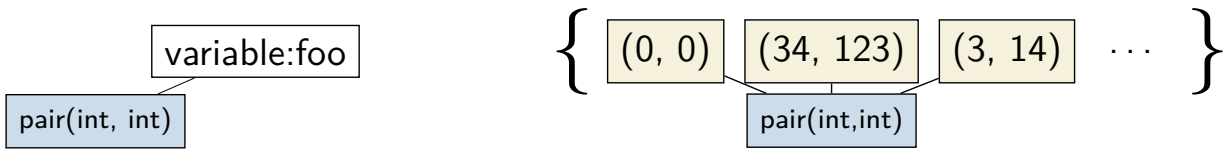
- Python ist ein Beispiel für eine stärker typisierten Sprache.
  - Jedes Objekt hat genau einen Typ, der bei der Erstellung festgelegt wird.
  - Wenige implizite Konvertierung. Zum Beispiel: `int`→`float`: `1+4.5` $\Rightarrow 5.5$
  - Konvertierungen explizit sichtbar: `int("23")` $\Rightarrow 23$, `str([1,2])` $\Rightarrow$ `"[1, 2]"`

- Beispiele für schwache Typisierung
  - TCL: Alles ist ein String. Jede Operation interpretiert den String anders.
  - PHP (< 7): Implizite Konvertierung von `str`→`int`: `"23"+4` $\Rightarrow$ 27
  - C: Arrays werden, falls nötig, zu einem Pointer auf ihr erstes Element.

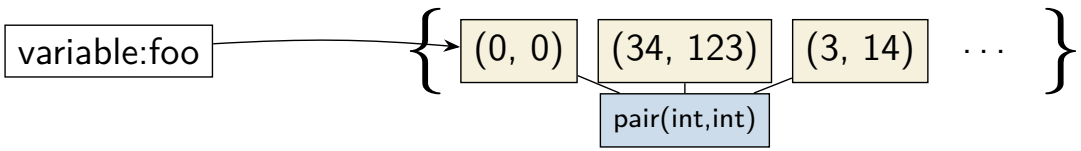
Typsysteme: Statische vs. Dynamische Typisierung

"Werden Typen statisch im Quellcode annotiert?"

- Statische Typisierung: C, C++, Rust, Java, Haskell,…
  - Bei jeder Definition/Deklaration wird ein statischer Typ festgelegt.
  - Variablen können niemals ein Objekt falschen Typs beinhalten.
  - Variablen und Funktionen haben einen Typen: `pair<int, int> foo`



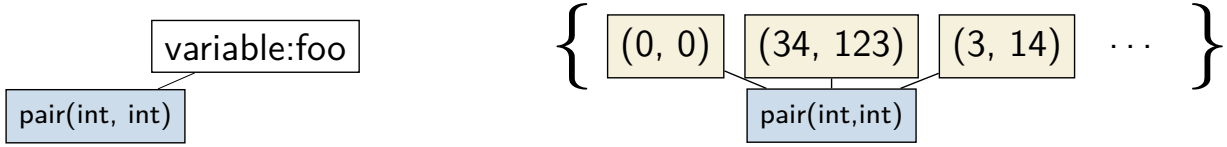
- Dynamische Typisierung: Python, PHP, JavaScript, Lisp, …
  - Variablen haben keine fest assoziierten statischen Typen.
  - Nur die Objekte, die in den Variablen „leben", haben eine Typ.



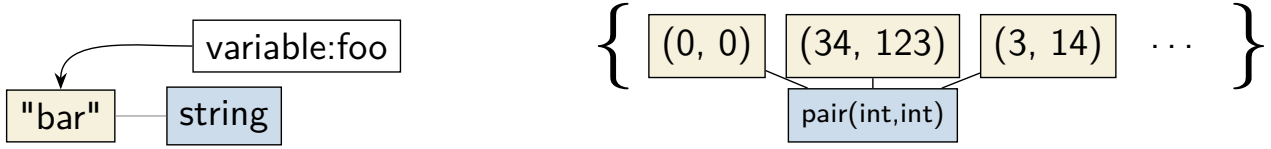
Typsysteme: Statische vs. Dynamische Typisierung

”Werden Typen statisch im Quellcode annotiert?”

- Statische Typisierung: C, C++, Rust, Java, Haskell,…
  - Bei jeder Definition/Deklaration wird ein statischer Typ festgelegt.
  - Variablen können niemals ein Objekt falschen Typs beinhalten.
  - Variablen und Funktionen haben einen Typen: `pair<int, int> foo`



- Dynamische Typisierung: Python, PHP, JavaScript, Lisp, …
  - Variablen haben keine fest assoziierten statischen Typen.
  - Nur die Objekte, die in den Variablen „leben“, haben eine Typ.



Typsysteme: Explizite vs. Implizite Typisierung

”Muss ich alle Typen wirklich hinschreiben?”

- Statische Typisierung erhöht die Menge an *explizit* getippten Zeichen.

```
std::vector<int> vec;
for (std::vector<int>::iterator it : vec) {...}
```

C++11

  - Teilweise wäre der Datentyp aus dem Kontext heraus klar.
  - Beim Programmierer entsteht das Gefühl von Boilerplate:
    „Der Übersetzer weiß das doch eh! Warum muss ich das hinschreiben?“

- Implizite Typisierung und **Typinferenz** macht das Leben einfacher.

```
std::vector<int> vec;
for (auto it : vec) {...}
```

C++11

  - Der Übersetzer errechnet den Typen für `it` aus dem Kontext.
  - Fortgeschrittene Typinferenz mittels Unifikation möglich (später mehr).

Zusammenfassung

- Das formale Konzept Typen findet Anwendung in Programmiersprachen
  - Typen haben eine Struktur, eine Objektmenge und ein Interface.
  - Typkonstruktoren komponieren komplexe Typen aus einfachen.
- Geläufige Typen werden oft direkt von der Sprache unterstützt.
  - Skalare Typen sind unteilbar mit endlichem Wertebereich.
  - Zusammengesetzte Typen können mehrere Subobjekte halten.
- Polymorphe Typsysteme sind flexibler als monomorphe Typsysteme.
  - Subtypen definieren eine Hierarchie von kompatiblen Untermengen.
  - Generische Typen sind benutzerdefinierte Typkonstruktoren.
- Typsysteme beschreiben die Integration von Typen in die Sprache.
  - Strikte Typisierung verhindert, dass wir das Typsystem umgehen.
  - Statische Typisierung gibt jeder Variable einen festen Typen.