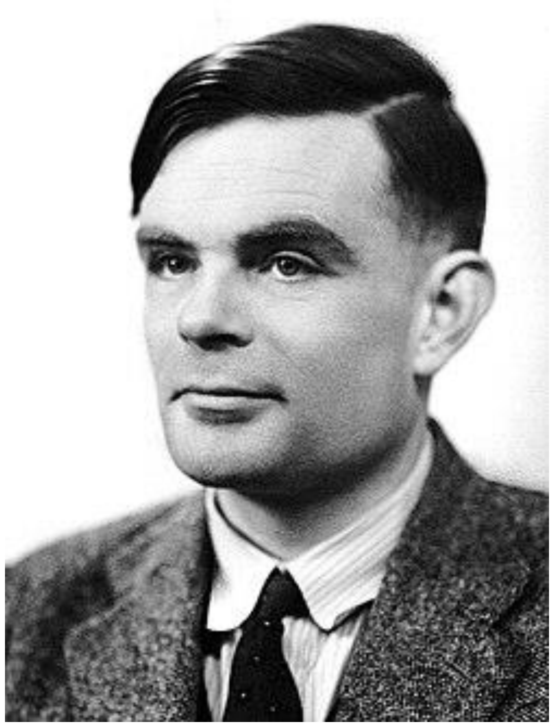
Type Checking

Niklas de Bruyn

Malte Schulze Balhorn

Robin Thielking

Church-Turing-These



λ-calculus

- Erfunden/Entdeckt von Alonzo Church
- Beschreibt Funktionen mit gebundenen Variablen
- Regeln:
  - λx.M (Abstraction)
  - (λx.M) N (Application)
  - (λx.x) 3 -> 3 (Reduction)

λ-calculus

- Wie stellt man Funktionen mit mehreren Variablen dar?
- Lösung: In λx.M kann M ebenfalls eine λ-Funktion sein
- Bsp.: λx.λy.x+y
    1. (λx.λy.x+y) 2 3
    2. (λy.2+y) 3
    3. 2 + 3
    4. 5
- Currying: Aus einer Funktion von (int,int)=> int wird eine Funktion von int => (int => int) bzw. int => int => int

λ-calculus

- In vielen Programmiersprachen vertreten (Anonyme Funktion)
- Bsp.: $\lambda x.x+1$
  - Python: lambda x: x + 1
  - Javascript: (x) => {return x+1}
  - Haskell: \x -> x + 1

λ-calculus

- Mit λ-calculus lassen sich logische Funktionen wie TRUE und FALSE darstellen
- TRUE := $\lambda x.\lambda y.x$
- FALSE := $\lambda x.\lambda y.y$
- IFELSE := $\lambda b.\lambda x.\lambda y.b\ x\ y$
    1. IFELSE TRUE A B
    2. TRUE A B
    3. A

λ-calculus

- λ-calculus erlaubt viele Sachen wie z.B.
- $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$ divergiert
- $(\lambda x.\ x + 1)\ (\lambda y.\ y\ y)$ macht keinen Sinn

Simply typed λ-calculus

- Lösung: Einführung von Typen
- $\lambda x. x\ +\ 1\ \ => \ (\lambda x{:}\,\mathbb{N}. x\ +\ 1){:}\,\mathbb{N} -> \mathbb{N}$
- Wie überprüft man ob diese Typen korrekt sind?

Simply typed λ-calculus

$$\frac{\mathrm{x}:\sigma \in \Gamma}{\Gamma \vdash \mathrm{x}:\sigma} \quad \text{(Variable)}$$

$$\frac{\Gamma, \mathrm{x}:\sigma \vdash \mathrm{e} : \tau}{\Gamma \vdash (\lambda \mathrm{x}:\sigma.\mathrm{e}):(\sigma \to \tau)} \quad \text{(Abstraction)}$$

$$\frac{\Gamma \vdash \mathrm{e}_1:\sigma \to \tau \qquad \Gamma \vdash \mathrm{e}_2:\sigma}{\Gamma \vdash \mathrm{e}_1\mathrm{e}_2:\tau} \quad \text{(Application)}$$

Simply typed λ-calculus

- Beispiel: Identität

$$\frac{\mathrm{x}: \alpha \vdash \mathrm{x}: \alpha}{\emptyset \ \vdash (\lambda \mathrm{x}: \alpha. \mathrm{x}): \alpha \to \alpha}$$

Simply typed λ-calculus

- Nicht lösbar, wir wissen nicht was y ist.

$$\frac{\mathrm{x}{:}\,\alpha \vdash \ \mathrm{y}{:}\,\alpha \to \sigma \qquad\qquad \mathrm{x}{:}\,\alpha \vdash \ \mathrm{x}{:}\ \alpha}{\dfrac{\mathrm{x}{:}\,\alpha \vdash \ \mathrm{y}\ \mathrm{x} :\ \sigma}{\emptyset\ \vdash (\lambda \mathrm{x}{:}\,\alpha.\,\mathrm{y}\ \mathrm{x}){:}\,\alpha \to \sigma}}$$

Simply typed λ-calculus

- Lösung: Kontext erweitern

$$\frac{\mathrm{y}: \alpha \to \sigma, \mathrm{x}: \alpha \vdash \mathrm{y}: \alpha \to \sigma \qquad \mathrm{y}: \alpha \to \sigma, \mathrm{x}: \alpha \vdash \mathrm{x}: \alpha}{\dfrac{\mathrm{y}: \alpha \to \sigma, \mathrm{x}: \alpha \vdash \mathrm{y}\,\mathrm{x} : \sigma}{\mathrm{y}: \alpha \to \sigma \vdash (\lambda \mathrm{x}: \alpha.\, \mathrm{y}\,\mathrm{x}): \alpha \to \sigma}}$$

Simply typed λ-calculus

$$\frac{\dfrac{\dfrac{x : \alpha, y : \beta \vdash x : \beta \to \rho \qquad x : \alpha, y : \beta \vdash y : \beta}{x : \alpha, y : \beta \vdash x\, y : \rho}}{x : \alpha \vdash \lambda y : \beta .\, x\, y : \beta \to \rho}}{\emptyset \vdash \lambda x : \alpha .\, \lambda y : \beta .\, x\, y : \alpha \to \beta \to \rho}$$

Propositions as Types

- STLC ähnelt sehr stark Natural Deduction aus der Mathematik $$(E \to)\ \frac{A \qquad A \to B}{B}$$
- Tatsächlich tritt dieses Phänomen häufiger aus und ist bekannt als Curry-Howard correspondence (oder Isomorphismus)

Propositions as Types

Aussagen als Typen

Beweise als Programme

Beweisvereinfachung als Programmauswertung

Propositions as Types

Natural Deduction	$\leftrightarrow$	Typed Lambda Calculus
Gentzen (1935)		Church (1940)
Type Schemes	$\leftrightarrow$	ML Type System
Hindley (1969)		Milner (1975)
System F	$\leftrightarrow$	Polymorphic Lambda Calculus
Girard (1972)		Reynolds (1974)
Modal Logic	$\leftrightarrow$	Monads (state, exceptions)
Lewis (1910)		Kleisli (1965), Moggi (1987)
Classical-Intuitionistic Embedding	$\leftrightarrow$	Continuation Passing Style
Gödel (1933)		Reynolds (1972)
Linear Logic	$\leftrightarrow$	Session Types
Girard (1987)		Honda (1993)

Quelle: https://homepages.inf.ed.ac.uk/wadler/papers/propositions-as-types/stanford.pdf

Warum ist λ-calculus so mächtig?

Es bildet das Fundament der funktionalen Programmierung (z.B. LISP, Haskell, ...)

```lisp
(defun fakultaet (n)
  (if (<= n 1)
      1
      (* n (fakultaet (- n 1)))))
```

LISP (~1960)

```haskell
fakultaet :: Integer -> Integer
fakultaet n =
  if n <= 1
    then 1
    else n * fakultaet (n - 1)
```

Haskell (~1990)

Die **Kernkonzepte** spielen dabei eine wichtige Rolle**:**

- Konzept 1: "***Variable Binding***" ⇒ λx.x+1

  → Die Variable "x" wird durch "λx" an den Ausdruck "x + 1", dem Körper der Funktion, gebunden

- Konzept 2: "***β-Reduction***" ⇒ (λx.x+1) 5 → 5 + 1 → 6

  → Anwendung der Funktion auf das Argument "5" und ersetzen der gebundenen Variable durch diesen

- λ-calculus ist außerdem “**Turing complete**”

Konzept 1: „Variable Binding“ im Detail

Was bedeutet dies genau?

<u>Mathematischer Exkurs:</u>

1. $f(x) = x^n$ → Geb. Var “**x**”, Freie Var “**n**”
2. $g(y) = yn$ → Geb. Var “**y**”, Freie Var “**n**”
3. $h(x) = x^c$ → Geb. Var “**x**”, Freie Var “**c**”

Durch die gegebenen Beispiele können wir behaupten, dass die Funktion f dasselbe ist wie g.

Wir können aber nicht behaupten, dass g und h dasselbe ist, da unterschiedliche freie Variablen **n** und **c** existieren.

Bildung eines Scopes ⇒ Wichtig für Funktionen und funktionale Programmiersprachen

Diese Eigenschaft wird **alpha equivalence** genannt ⇒ f und g sind **semantisch identisch**

Konzept 2: „β-Reduction“

Bisher Zahlen immer direkt dargestellt.

Vorheriges Beispiel → (λx.x+1) 5

Wie werden sie im λ-calculus dargestellt?

Mit **Church encoding** → n := $\lambda f.\lambda x.f^n(x)$

0 := λ*f*.λ*x*.*x*

1 := λ*f*.λ*x*.*f x*

2 := λ*f*.λ*x*.*f* (*f x*)

3 := λ*f*.λ*x*.*f* (*f* (*f x*))

Daraus folgt für unser Beispiel in richtiger Notation:

(λx.x+1) 5 ⇒ (λx.+ x λ*f*.λ*x*.*f x*) λ*f*.λ*x*.*f* (*f* (*f* (*f* (*f x*))))

Demo: Lambda Calculus Calculator

OR FALSE TRUE = (λxy.IF x TRUE y) FALSE TRUE
→β (λy.IF FALSE TRUE y) TRUE
→β IF FALSE TRUE TRUE
= (λbtf.btf) FALSE TRUE TRUE
→β (λtf.FALSE tf) TRUE TRUE
→β (λf.FALSE TRUE f) TRUE
→β FALSE TRUE TRUE
= (λxy.y) TRUE TRUE
→β (λy.y) TRUE
→β TRUE

Hinweise:

1. e_1 →β e_2 bedeutet e_1 reduziert unmittelbar zu e_2
2. λxy.y ist äquivalent zu λx.λy.y

λ-calculus in Programmiersprachen

Warum behandeln wir ein derart theoretisches Thema?
Wie hilft uns dies bei Programmiersprachen und deren Compiler?

- Konzepte lassen sich auf Programmiersprachen übertragen:

Variables ⇔ Variables

Abstraction ⇔ Anonymous Function

Application ⇔ Function Call

- Viele weitere Sprachkonzepte darunter Closures und High-Order Functions lassen sich daraus ableiten
- Type Systems, Type Inference und Type Checking bilden sich aus den stetigen Erweiterungen
- Untyped λ-calculus besitzt keine Typen und jede Anwendung ist somit erlaubt ⇒ führt zu Laufzeitfehlern
- Daher Typed λ-calculus, auf welchem das Hindley-Milner Typsystem basiert

Was sind Typen und Typsysteme?

Definition Typen (τ)

- Abstraktion über Werte in einem formalen System
  ⇒ beschreibt Art von Daten, die eine Variable oder ein Ausdruck repräsentieren kann:
  Boolean = {true, false}

Definition Typsysteme

- Menge von Regeln, die jedem Ausdruck $\boldsymbol{e}$ einen Typen zuordnen:
  $\Gamma \vdash e : \tau$ ⇒ "In der Umgebung $\boldsymbol{\Gamma}$ hat $\boldsymbol{e}$ den Typ $\boldsymbol{\tau}$"

> The fundamental purpose of a **type system** is to prevent the occurrence of execution errors during the running of a program. –*Cardelli, Type Systems 2004*

Typüberprüfung in Programmiersprachen

```
fn add(a: u8, b: u8) -> u8 {
    a + b
}

fn main() {
    let x = add(2, 3);        // Funktioniert
    let y = add("2", 3);      // Compile Error:
                              // expected `u8`, found `&str`
}
```

Type Checking

```
fn main() {
    // Compiler versteht Typ durch Annotation "u8"
    let elem = 5u8;

    // Initialisierung eines leeren Vektors
    // Genauer Typ aber unbekannt (`Vec<_>`)
    let mut vec = Vec::new();


    // Elem unseren Vektor hinzufügen
    vec.push(elem);
    // Compiler versteht nun, dass Vektor-Typ gleich Element-Typ (`Vec<u8>`)

}
```

Type Inference

Typüberprüfung in Programmiersprachen

```rust
fn add(a: u8, b: u8) -> u8 {
    a + b
}

fn main() {
    let x = add(2, 3);        // Funktioniert
    let y = add("2", 3);      // Compile Error:
                              // expected `u8`, found `&str`
}
```

Type Checking

```rust
fn main() {
    // Compiler versteht Typ durch Annotation "u8"
    let elem = 5u8;

    // Initialisierung eines leeren Vektors
    // Genauer Typ aber unbekannt (`Vec<_>`)
    let mut vec = Vec::new();


    // Elem unseren Vektor hinzufügen
    vec.push(elem);
    // Compiler versteht nun, dass Vektor-Typ gleich Element-Typ (`Vec<u8>`)

}
```

Type Inference

Arten der Typüberprüfung im Vergleich

Static Type Checking

- Typen werden zur Kompilierzeit überprüft
- Compiler analysiert Code und stellt sicher, dass das Nutzen von Werten hinsichtlich ihrer Typregelungen konsistent bleibt

\+ Vorteil liegt in der Früherkennung von Fehlern und der generellen Unterstützung bei der Programmierung selbst

\- Nachteil liegt in der gefühlten Einschränkung und Umständlichkeit, insbesondere wenn keine Typinferenz stattfindet

Beispiele: C, C++, Java, Rust, Haskell

Dynamic Type Checking

- Typen werden zur Laufzeit überprüft

\+ Höhere Flexibilität

\- Kann zu Runtime Fehlern führen

Beispiele: Perl, Ruby, Python, JavaScript

Arten der Typüberprüfung im Vergleich

Static Type Checking

- Typen werden zur Kompilierzeit überprüft
- Compiler analysiert Code und stellt sicher, dass das Nutzen von Werten hinsichtlich ihrer Typregelungen konsistent bleibt

+ Vorteil liegt in der Früherkennung von Fehlern und der generellen Unterstützung bei der Programmierung selbst

- Nachteil liegt in der gefühlten Einschränkung und Umständlichkeit, insbesondere wenn keine Typinferenz stattfindet

Beispiele: C, C++, Java, Rust, Haskell

Dynamic Type Checking

- Typen werden zur Laufzeit überprüft

+ Höhere Flexibilität

- Kann zu Runtime Fehlern führen

Beispiele: Perl, Ruby, Python, JavaScript

	Typed	Untyped
Safe	ML, Java	LISP
Unsafe	C	Assembler

Entnommen aus Type Systems, Cardelli 2004

Typüberprüfung in Programmiersprachen

```rust
fn add(a: u8, b: u8) -> u8 {
    a + b
}

fn main() {
    let x = add(2, 3);        // Funktioniert
    let y = add("2", 3);      // Compile Error:
                              // expected `u8`, found `&str`
}
```

Type Checking

```rust
fn main() {
    // Compiler versteht Typ durch Annotation "u8"
    let elem = 5u8;

    // Initialisierung eines leeren Vektors
    // Genauer Typ aber unbekannt (`Vec<_>`)
    let mut vec = Vec::new();


    // Elem unseren Vektor hinzufügen
    vec.push(elem);
    // Compiler versteht nun, dass Vektor-Typ gleich Element-Typ (`Vec<u8>`)

}
```

Type Inference

Einführung in das Hindley–Milner Typsystem

Hindley-Milner ist ein Typsystem, welches auf dem λ-calculus aufbaut und parametrischen Polymorphismus einführt.

⇒ Dient der automatischen Typinferenz funktionaler Sprachen

⇒ Ermöglicht Typprüfung ohne Annotationen

> „A well-typed program won‘t go wrong.“ *-Milner*

Symboldefinitionen:

$\boldsymbol{\tau}$ ⇒ Repräsentiert monomorphe Typen (z.B. Int, Bool oder Int → Bool)
α,β,γ ⇒ Typvariablen für unbekannte Typen während der Inferenz

$\boldsymbol{\Gamma}$ ⇒ Repräsentiert die Typumgebung, in welcher bekannte Variablen und deren Typen gemapped werden

Einführung in das Hindley–Milner Typsystem

Symboldefinitionen (Forts.):

σ ⇒ Repräsentiert polymorphe Typschemata (z.B. ∀α.α→α)

∀α ⇒ Universelle Quantifikation über Typvariablen → damit wird Polymorphismus ausgedrückt

[τ/α]σ ⇒ Typsubstitution → Ersetzen aller vorkommenden Typvariablen **α** mit Typ **τ** in Schema **σ**

S ⇒ Zuordnung von Typvariablen zu Typen, repräsentiert gefundene Lösungen durch Unifikation

gen(Γ,τ) ⇒ Generalisation, verwandelt Monotyp in Polytyp, indem sie über Typvariablen quantifiziert, die nicht in der Umgebung vorhanden sind

inst(σ) ⇒ Universelle Quantifikation über Typvariablen, damit wird Polymorphismus ausgedrückt

ftv(τ) ⇒ Typsubstitution → Ersetzen aller vorkommenden Typvariablen α

S ⇒ Zuordnung von Typvariablen zu Typen, repräsentiert gefundene Lösungen durch Unifikation

Einführung in das Hindley–Milner Typsystem

<u>Symboldefinitionen (Forts.):</u>

∅	⇒ Leere Substitution → repräsentiert keine Änderung des Typen
$[\alpha \mapsto \tau]$	⇒ Substitution, bei der die Typvariable **α** auf den Typ **τ** abgebildet wird
$S_1 \circ S_2$	⇒ Zusammensetzung von Substitutionen, zuerst **S_1** ,dann **S_2**

Hindley-Milner Typsystem (λ→HM)

$$e ::= \quad x \quad | \quad \lambda x.e \quad | \quad e_1\, e_2$$

Variable

Abstraction

Application

```rust
pub enum Expr {
    Var(String),
    Abs(String, Box<Expr>),
    App(Box<Expr>, Box<Expr>),
}
```

Entnommen aus Typechecker Zoo, Diehl 2025

Zu schwer für direkte Programmierung!

⇒ Daher Erweiterung des Kerns durch zusätzliche Ausdruckstypen

Hindley-Milner Typsystem (AST)

Zu den Erweiterungen zählen zum Beispiel:

- eine let-Bindung für lokale Variablen
- Literale für konkrete Werte wie Zahlen und Strings
- Datenstrukturen wie Tupel

Damit haben wir unseren algebraischen Datentyp für Ausdrücke implementiert, welcher von unserem expression AST dargestellt wird

```
pub enum Expr {
    Var(String),
    Abs(String, Box<Expr>),
    App(Box<Expr>, Box<Expr>),
    Let(String, Box<Expr>, Box<Expr>),
    Lit(Lit),
    Tuple(Vec<Expr>),
}
```

Entnommen aus Typechecker Zoo, Diehl 2025

Hindley-Milner Typsystem (AST)

Analog dazu wird ein weiterer algebraischer Datentyp für unsere Typen implementiert und orientiert sich am vorherigen Aufbau.

Type::Var dient als Platzhalter während Inferenz

Type:Arrow repräsentiert Funktionstypen, genauer Parametertyp und Rückgabetyp

Type:Int und **Type::Bool** dienen als Fundament

Type::Tuple unterstützt strukturierte Daten

```
pub enum Type {
    Var(String),
    Arrow(Box<Type>, Box<Type>),
    Int,
    Bool,
    Tuple(Vec<Type>),
}
```

Entnommen aus Typechecker Zoo, Diehl 2025

Hindley-Milner Typsystem (Inference Alg.)

TyVar / TmVar → Unbekannte Typen & Programmvariablen

Env → ordnet Variablen Typ-Schemata zu (für Let-Polymorphismus)

Subst → speichert konkrete Typzuweisungen (Unifikation)

Fresh_tyvar / TypeInference → verwaltet Zustand, erzeugt frische Typvariablen und ist zählerbasiert → garantiert eindeutige Namen (t0, t1, ...)

Hinweis:
Strings für Einfachheit, reale Systeme nutzen optimierte Repräsentationen

```rust
pub type TyVar = String;
pub type TmVar = String;
pub type Env = BTreeMap<TmVar, Scheme>;
pub type Subst = HashMap<TyVar, Type>;

pub struct TypeInference {
    counter: usize,
}


fn fresh_tyvar(&mut self) -> TyVar {
    let var = format!("t{}",
self.counter);
    self.counter += 1;
    var
}
```

Entnommen aus Typechecker Zoo, Diehl 2025

Hindley-Milner Typsystem (Inference Alg.)

```
pub fn infer(&mut self, env: &Env, expr: &Expr) -> Result<(Subst, Type, InferenceTree)> {
    match expr {
        Expr::Lit(Lit::Int(_)) => self.infer_lit_int(env, expr),
        Expr::Lit(Lit::Bool(_)) => self.infer_lit_bool(env, expr),
        Expr::Var(name) => self.infer_var(env, expr, name),
        Expr::Abs(param, body) => self.infer_abs(env, expr, param, body),
        Expr::App(func, arg) => self.infer_app(env, expr, func, arg),
        Expr::Let(var, value, body) => self.infer_let(env, expr, var, value, body),
        Expr::Tuple(exprs) => self.infer_tuple(env, expr, exprs),
    }
}
```

```
/// T-Var: x : σ ∈ Γ    τ = inst(σ)
///        ──────────────────────────
///                Γ ⊢ x : τ
fn infer_var(
    &mut self,
    env: &Env,
    expr: &Expr,
    name: &str,
) -> Result<(Subst, Type, InferenceTree)> {
    let input = format!("{} ⊢ {} ⇒", self.pretty_env(env), expr);

    match env.get(name) {
        Some(scheme) => {
            let instantiated = self.instantiate(scheme);
            let output = format!("{}", instantiated);
            let tree = InferenceTree::new("T-Var", &input, &output, vec![]);
            Ok((HashMap::new(), instantiated, tree))
        }
        None => Err(InferenceError::UnboundVariable {
            name: name.to_string(),
        }),
    }
}
```

Entnommen aus Typechecker Zoo, Diehl 2025

Hindley-Milner Typsystem (Ausblick)

$$\frac{x : \sigma \in \Gamma \quad \tau = \mathrm{inst}(\sigma)}{\Gamma \vdash x : \tau}\text{(T-Var)}$$

$$\frac{\Gamma, x : \alpha \vdash e : \tau \quad \alpha \text{ fresh}}{\Gamma \vdash \lambda x.\, e : \alpha \to \tau}\text{(T-Lam)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \alpha \text{ fresh} \quad S = \mathrm{unify}(\tau_1, \tau_2 \to \alpha)}{\Gamma \vdash e_1\, e_2 : S(\alpha)}\text{(T-App)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \sigma = \mathrm{gen}(\Gamma, \tau_1) \quad \Gamma, x : \sigma \vdash e_2 : \tau_2}{\Gamma \vdash \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 : \tau_2}\text{(T-Let)}$$

$$\frac{}{\Gamma \vdash n : \mathrm{Int}}\text{(T-LitInt)}$$

$$\frac{}{\Gamma \vdash b : \mathrm{Bool}}\text{(T-LitBool)}$$

$$\frac{}{\mathrm{unify}(\tau, \tau) = \emptyset}\text{(U-Refl)}$$

$$\frac{\alpha \notin \mathrm{ftv}(\tau)}{\mathrm{unify}(\alpha, \tau) = [\alpha \mapsto \tau]}\text{(U-VarL)}$$

$$\frac{\alpha \notin \mathrm{ftv}(\tau)}{\mathrm{unify}(\tau, \alpha) = [\alpha \mapsto \tau]}\text{(U-VarR)}$$

$$\frac{S_1 = \mathrm{unify}(\tau_1, \tau_3) \quad S_2 = \mathrm{unify}(S_1(\tau_2), S_1(\tau_4))}{\mathrm{unify}(\tau_1 \to \tau_2, \tau_3 \to \tau_4) = S_2 \circ S_1}\text{(U-Arrow)}$$

$$\frac{S_1 = \mathrm{unify}(\tau_1, \tau_3) \quad S_2 = \mathrm{unify}(S_1(\tau_2), S_1(\tau_4))}{\mathrm{unify}((\tau_1, \tau_2), (\tau_3, \tau_4)) = S_2 \circ S_1}\text{(U-Tuple)}$$

$$\frac{}{\mathrm{unify}(\mathrm{Int}, \mathrm{Int}) = \emptyset}\text{(U-Int)}$$

$$\frac{}{\mathrm{unify}(\mathrm{Bool}, \mathrm{Bool}) = \emptyset}\text{(U-Bool)}$$

Typing Rules

Entnommen aus Typechecker Zoo, Diehl 2025

⇒ [Type Systems - Typechecker Zoo]()

On Understanding Types Data Abstraction and Polymorphism

- *On Understanding Types, Data Abstraction, and Polymorphism* (Cardelli & Wegner, 1985)
- Vereinheitlichung von Typbegriffen in Programmiersprachen
- Einführung der Modellsprache **Fun** (auf λ-calculus basierend)
- Brücke zwischen funktionalen und objektorientierten Konzepten
- Grundstein für moderne Typsysteme (ML, Java, Rust ...)

Motivation & Problemstellung

- Warum brauchen wir Typen in Programmiersprachen?
- Schutz vor fehlerhaften Operationen („type safety“)
- Weg von **ungetypten** zu **getypten** Universen
- Typen als Mittel zur Strukturierung und Abstraktion
- Problem: Viele unverbundene Typkonzepte (OOP ↔ Funktional)
- Ziel: Einheitliches, formales Modell für **Typen & Polymorphie**

Arten von Polymorphismus

- *Polymorphismus* = ein Wert oder eine Funktion hat mehrere Typen
- Zwei Hauptkategorien: **Universal** und **Ad-hoc**
- **Universal:** Parametrisch ($\forall$) und Inklusion ($<$)
- **Ad-hoc:** Überladung und Coercion (Typkonvertierung)
- Neu bei Cardelli/Wegner: **Inclusion Polymorphism = Vererbung**
- Basis für generische Funktionen + Objekt-Hierarchien

Universal & Existential Quantification

- Zwei zentrale Erweiterungen in *Fun*: ∀ und ∃
- **∀ (Universal)** → parametrische Polymorphie
- **∃ (Existentiell)** → Datenabstraktion & Information Hiding
- ∀ = eine Funktion für alle Typen (generisch)
- ∃ = es gibt einen versteckten Typ (abstrakt)
- Kombination beider ⇒ generische abstrakte Datentypen (z. B. Stack)

Subtyping

- Bedeutet: Ein Typ S ist ein Subtyp eines Typs T, geschrieben S <: T.
- Werte des Subtyps können überall dort verwendet werden, wo der Supertyp erwartet wird.
- Beispiel: Hund <: Tier

Bounded Quantification

- Typvariable ist durch eine **Subtyp-Grenze** eingeschränkt
- Kombination aus **Polymorphie** und **Subtyping**
- Formal: **$\forall$ T <: Tier** („für alle T, die Untertypen von Tier sind“)
- Erlaubt **generische Funktionen**, die nur für bestimmte Typbereiche gelten
- Unterschied zu Subtyping: Einschränkung in der **Definition**, nicht Beziehung zwischen Typen

Parametrische Polymorphie – ∀ („für alle“)

- Funktion funktioniert **für alle Typen T** gleich
- Nimmt etwas vom Typ T und gibt wieder denselben Typ zurück
- universelle Quantifizierung (∀T.T→T)

```python
from typing import TypeVar

T = TypeVar("T")

def identity(a: T) -> T:
    return a

print(identity(42))          # Int
print(identity("Hello"))     # String
print(identity([1, 2, 3]))   # List[Int]
```

Existenzielle Typen – ∃ („es gibt“)

- Die Klasse Stack kapselt interne Repräsentation (_data) vollständig.
- Von außen ist nur das Interface sichtbar (push, pop), nicht aber wie die Daten gespeichert sind.
- Das entspricht einem existentiellen Typ:
  - „Es gibt einen Typ, aber die Außenwelt muss ihn nicht kennen.“
- Datenabstraktion ($\exists T.T \times (T \rightarrow T)$)

```python
class Stack:
    def __init__(self):
        self._data = []   # versteckte Repräsentation

    def push(self, x):
        self._data.append(x)

    def pop(self):
        return self._data.pop()

stack = Stack()
stack.push(10)
stack.push(20)
print(stack.pop())
```

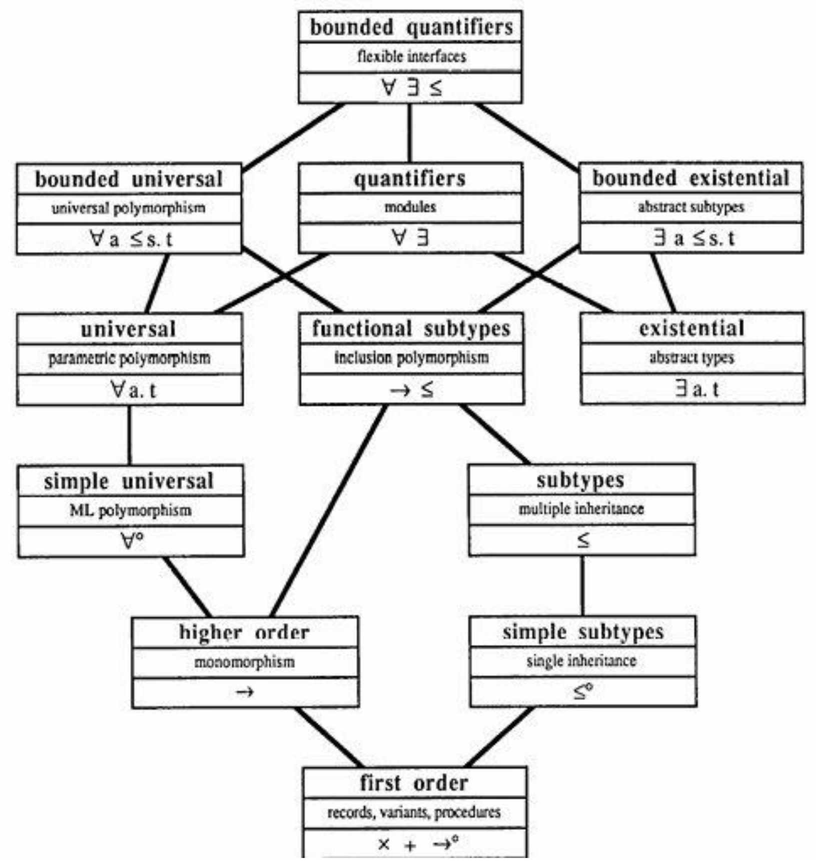
Inklusionspolymorphie
– Subtyping

- Inklusionspolymorphie: Funktion akzeptiert alle Subtypen von Tier.
- Dynamische Bindung: Ausführung der passenden Methode zur Laufzeit.
- Prinzip: Ein Interface – viele Implementierungen.

```python
class Tier:
    def __init__(self, name):
        self.name = name

    def laerm_machen(self):
        """Standard-Methode für Geräusche."""
        return f"{self.name} macht ein Geräusch."

class Hund(Tier):
    def laerm_machen(self):
        """Überschriebene Methode für Hunde."""
        return f"{self.name} bellt: Wau Wau!"

class Katze(Tier):
    def laerm_machen(self):
        """Überschriebene Methode für Katzen."""
        return f"{self.name} miaut: Miau!"

def tier_unterhalten(tier):
    """
    Diese Funktion akzeptiert ein 'Tier'-Objekt,
    aber auch *jeden* Subtyp von Tier (Hund, Katze).
    Das ist Inklusionspolymorphie.
    """
    print(tier.laerm_machen())

# Erstellung der Objekte
bello = Hund("Bello")
felix = Katze("Felix")
eine_echse = Tier("Echse")

# Aufruf der Funktion mit verschiedenen Subtypen
print("--- Test der Polymorphie ---")
tier_unterhalten(bello)      # Hund-Objekt wird als Tier behandelt
tier_unterhalten(felix)      # Katze-Objekt wird als Tier behandelt
tier_unterhalten(eine_echse) # Tier-Objekt
```

Hierarchie der Typensysteme

- Das Diagramm ordnet **alle bekannten Typkonzepte** entlang zweier Achsen:
  - **Horizontal:** von *Universalität (∀)* zu *Existentialität (∃)*
    → *also von generischen Typen zu abstrakten, verborgenen Typen.*
  - **Vertikal:** von *einfachen Typen* unten bis zu *flexiblen, quantifizierten Systemen* oben.

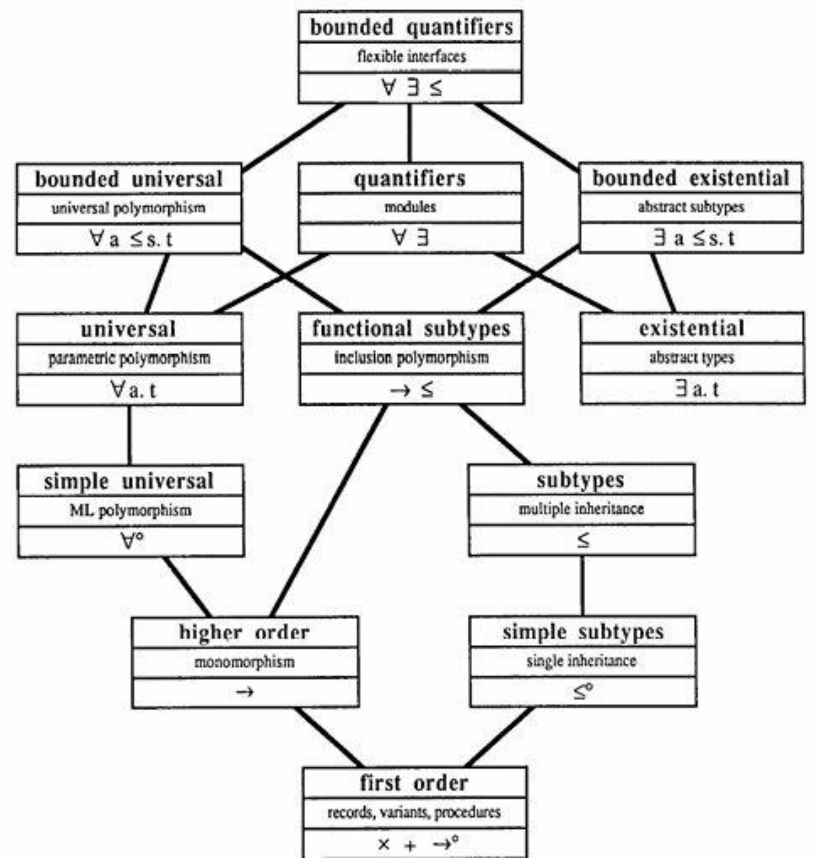


Unterste Ebene – First Order

Records, Variants, Procedures:

- → Die einfachsten Typkonstrukte, wie Strukturen, Alternativen und Funktionen.

- Diese Ebene ist nicht polymorph – jedes Objekt hat genau einen Typ.

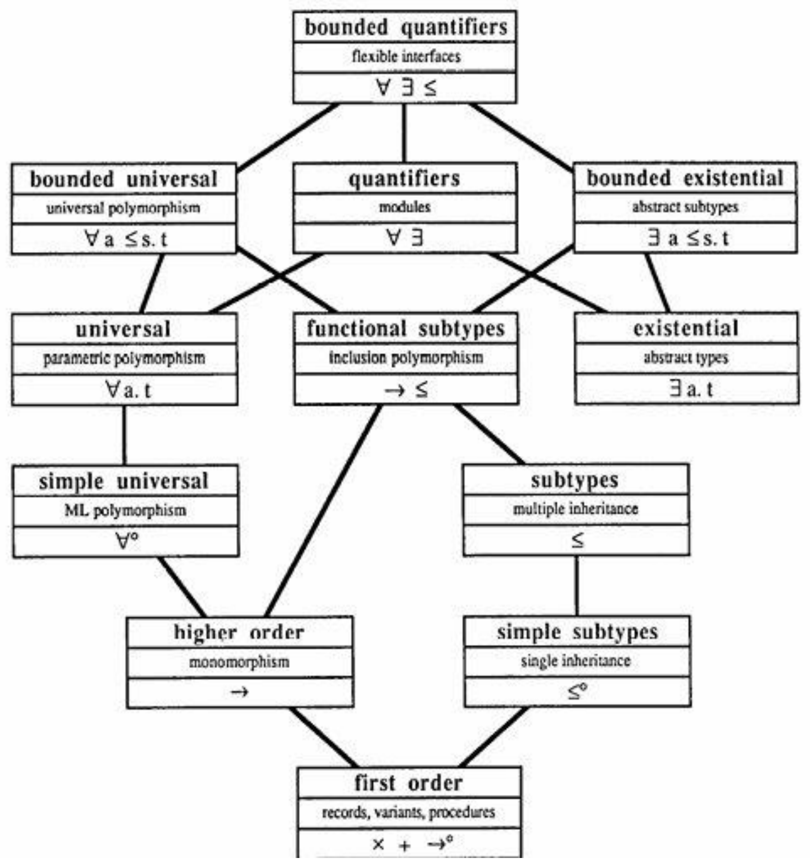


Simple Subtypes und Higher Order

Records, Variants, Procedures:

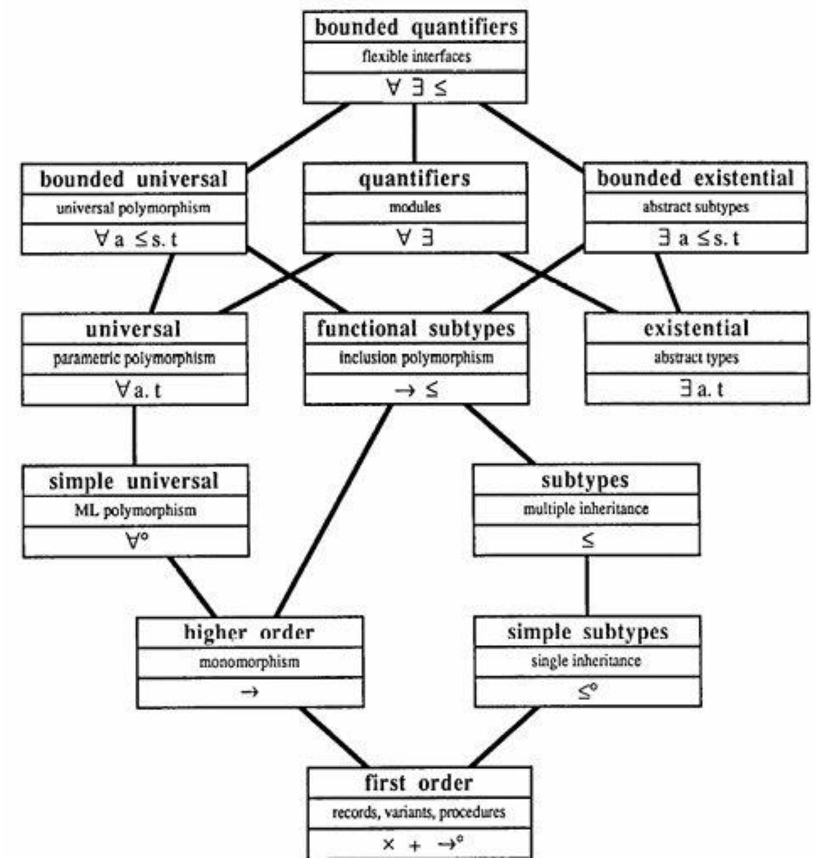
- → Die einfachsten Typkonstrukte, wie Strukturen, Alternativen und Funktionen.

- Diese Ebene ist nicht polymorph – jedes Objekt hat genau einen Typ.



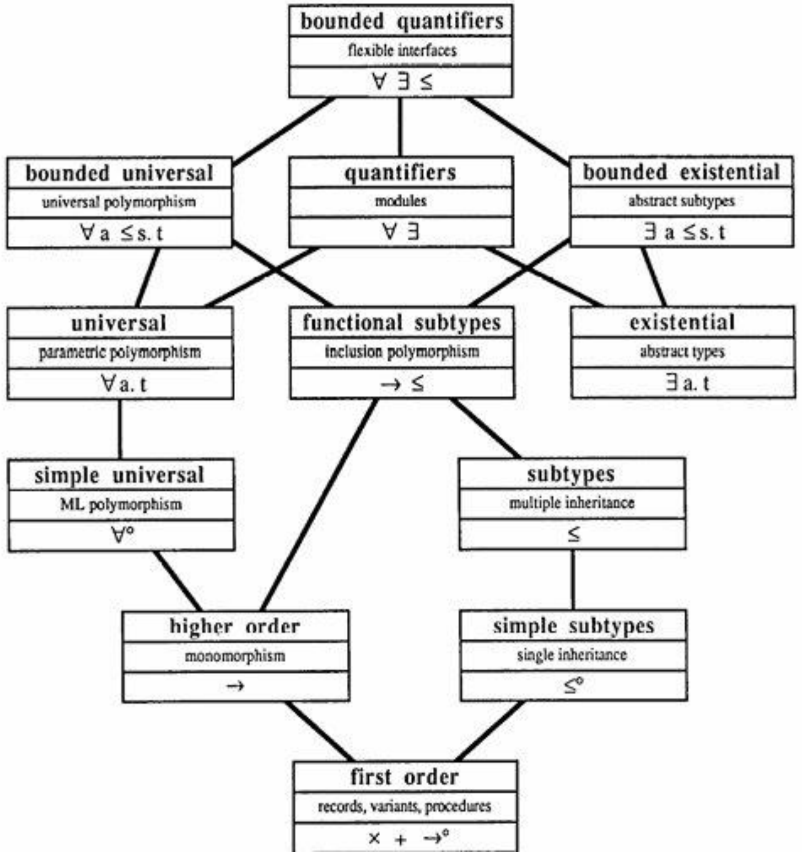
Universal und Existential

- Universal (∀ a. t): parametrische Polymorphie – eine Definition funktioniert für alle Typen.
- Existential (∃ a. t): Datenabstraktion – es gibt einen verborgenen Typ, der intern genutzt wird.
- → Diese beiden sind komplementär: Universal = generisch, Existential = kapselnd.



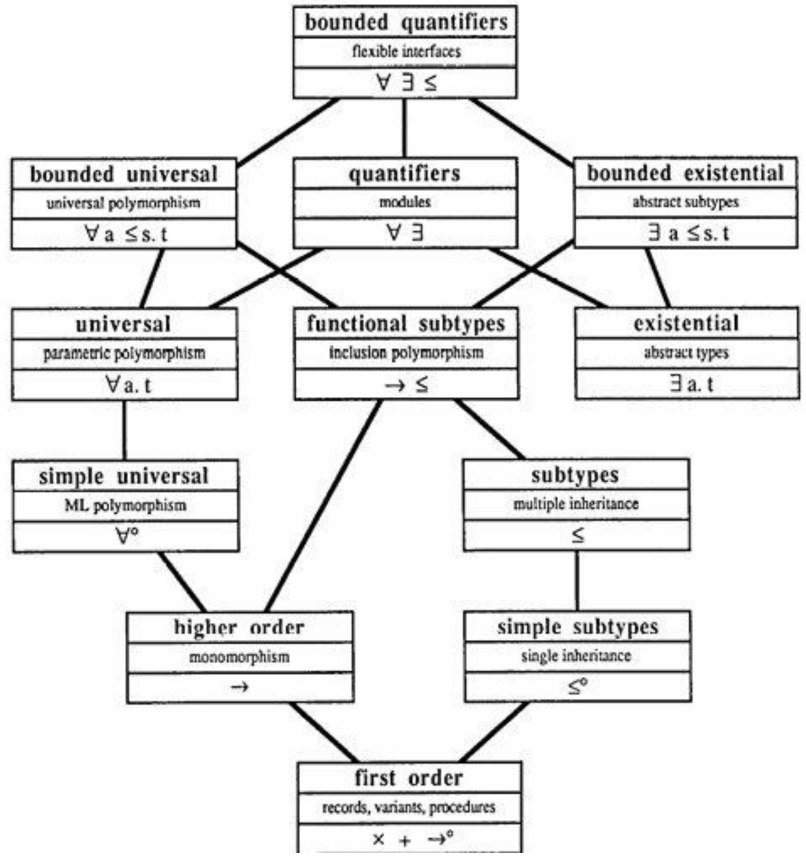
Functional Subtypes

- Inklusionspolymorphie (≤) – also Vererbung und Subtyping.
- Diese Ebene verbindet funktionale und objektorientierte Konzepte.



Bounded Quantifiers

- Systeme, in denen Polymorphie und Subtyping kombiniert werden.
    - Bounded Universal (∀ a ≤ s. t): generische Definitionen mit Typgrenzen („für alle Untertypen von s“).
    - Bounded Existential (∃ a ≤ s. t): abstrakte Subtypen – z. B. verborgene Implementierungen mit Vererbung.
- Diese Systeme erlauben flexible Interfaces und modulare Typdefinitionen



Fun

```
type Point2 =
    ∃Point.
        {makepoint: (Real × Real) → Point,
         x_coord: Point → Real,
         y_coord: Point → Real
        }
```

- Theoretische Modellsprache von **Cardelli & Wegner (1985)**
- basiert auf **getyptem λ-Calculus**
- **Grundtypen:** Int, Bool, String, Unit
- **Strukturierte Typen:** Record, Variant, Function, Recursion
- **Erweiterungen:** ∀ (*für alle*), ∃ (*es gibt*), *bounded quantification*
- **Ziel:** gemeinsamer Rahmen für **funktionale & objektorientierte Typen**
- Beispiel: **∃** – Datenabstraktion und *Information Hiding*

Bedeutung & Einfluss

- Vereinheitlichung der Typkonzepte in Programmiersprachen
- Theoretische Grundlage für **Abstraktion, Polymorphie & Subtyping**
- Einführung einer **formalen Typsemantik** über den λ-Calculus
- Brücke zwischen **funktionaler** und **objektorientierter** Typauffassung
- Einfluss auf moderne Typsysteme: **Java, C#, Scala, Rust, Haskell**
- Cardelli gilt als **Mitbegründer der objektorientierten Typentheorie**