

# Laufzeit-Monomorphisierung

## Dynamische Spezialisierung Polymorpher Funktionen

Martin Grabmüller

Fachgebiet Übersetzerbau und Programmiersprachen  
Fakultät IV – Elektrotechnik und Informatik  
Technische Universität Berlin

ÜBB-Kolloquium, 28. November 2006

## Einführung

- Polymorphie in Programmiersprachen
- Implementierung polymorpher Funktionen
- Statische Monomorphisierung

## Laufzeit-Monomorphisierung

- Allgemeines Prinzip
- Quellsprache
- Implementierung
- Erweiterungen

## Abschluss

- Weitere Arbeiten
- Zusammenfassung

## Einführung

- Polymorphie in Programmiersprachen
- Implementierung polymorpher Funktionen
- Statische Monomorphisierung

## Laufzeit-Monomorphisierung

- Allgemeines Prinzip
- Quellsprache
- Implementierung
- Erweiterungen

## Abschluss

- Weitere Arbeiten
- Zusammenfassung

polymorph  $\Leftrightarrow$  vielgestaltig

- ▶ Eine Implementierung arbeitet auf verschiedenartigen Objekten (parametrische Polymorphie, Generizität)

polymorph  $\Leftrightarrow$  vielgestaltig

- ▶ Eine Implementierung arbeitet auf verschiedenartigen Objekten (parametrische Polymorphie, Generizität)
- ▶ Ein Name steht für verschiedene Implementierungen (ad-hoc Polymorphie, Überlagerung)

polymorph  $\Leftrightarrow$  vielgestaltig

- ▶ Eine Implementierung arbeitet auf verschiedenartigen Objekten (parametrische Polymorphie, Generizität)
- ▶ Ein Name steht für verschiedene Implementierungen (ad-hoc Polymorphie, Überlagerung)

polymorph  $\Leftrightarrow$  vielgestaltig

- ▶ Eine Implementierung arbeitet auf verschiedenartigen Objekten (parametrische Polymorphie, Generizität)
- ▶ Ein Name steht für verschiedene Implementierungen (ad-hoc Polymorphie, Überlagerung)

## Beispiel

*Eine Definition:*

`id ::  $\forall \alpha. \alpha \rightarrow \alpha$`

`id =  $\lambda x:\alpha. x$`

*... beliebig viele Verwendungsmöglichkeiten:*

`id True  $\hookrightarrow$  True`

`id 'a'  $\hookrightarrow$  'a'`

`id 42  $\hookrightarrow$  42`

`...`

# Polymorphie in Programmiersprachen

polymorph  $\Leftrightarrow$  vielgestaltig

- ▶ Eine Implementierung arbeitet auf verschiedenartigen Objekten (parametrische Polymorphie, Generizität)
- ▶ Ein Name steht für verschiedene Implementierungen (ad-hoc Polymorphie, Überlagerung)

## Beispiel

*Eine Definition:*

$\text{id} :: \forall \alpha. \alpha \rightarrow \alpha$

$\text{id} = \lambda x:\alpha. x$

*... beliebig viele Verwendungsmöglichkeiten:*

$\text{id True} \hookrightarrow \text{True}$

$\text{id 'a'} \hookrightarrow \text{'a'}$

$\text{id 42} \hookrightarrow 42$

...

Mittel der Abstraktion: Konzeptionelle Vorteile

Mittel der Wiederverwendung: Praktische Vorteile

# Implementierung polymorpher Funktionen

Jede polymorphe Funktion wird in eine Maschinencode-Sequenz übersetzt.

- + Pro polymorpher Definition nur eine übersetzte Version
  - ▶ geringe Code-Größe
  - ▶ schnelle Kompilierung
- Erfordert uniforme Repräsentation verschiedener Datentypen

# Implementierung polymorpher Funktionen

Jede polymorphe Funktion wird in eine Maschinencode-Sequenz übersetzt.

- + Pro polymorpher Definition nur eine übersetzte Version
  - ▶ geringe Code-Größe
  - ▶ schnelle Kompilierung
- Erfordert uniforme Repräsentation verschiedener Datentypen

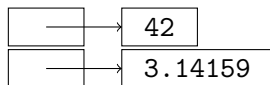
## Beispiel

Funktion `id` kennt weder die Repräsentation ihres Arguments noch ihres Resultats.

Aufrufe `id 42` und `id 3.14159` können nicht verschiedene Register verwenden.

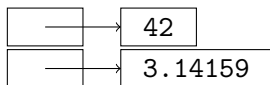
# Implementierung polymorpher Funktionen – Boxing

**Boxing:** unterschiedlich große Speicherblöcke werden indirekt referenziert, alle polymorphen Werte werden als Zeiger gleicher Größe manipuliert:



# Implementierung polymorpher Funktionen – Boxing

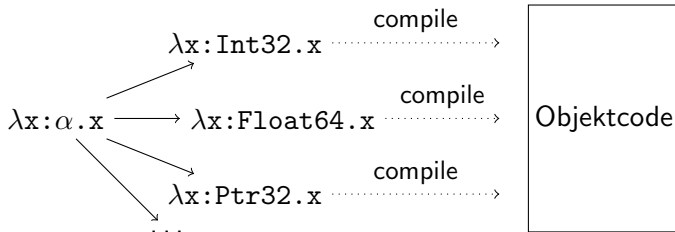
**Boxing:** unterschiedlich große Speicherblöcke werden indirekt referenziert, alle polymorphen Werte werden als Zeiger gleicher Größe manipuliert:



- Benötigt mehr Speicherplatz
- Speicherreservierung und Boxing/Unboxing kosten Zeit
- Indirekte Zugriffe kosten Zeit
- Garbage Collection kostet Zeit
- + Einfache Implementierung

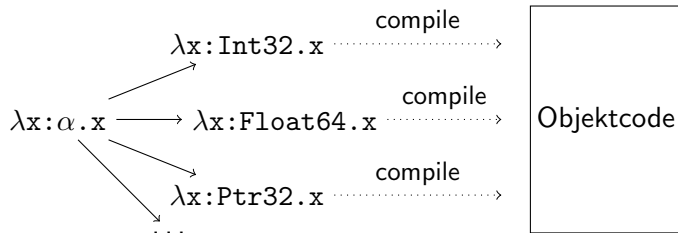
# Statische Monomorphisierung

Spezialisierung zur Übersetzungszeit



# Statische Monomorphisierung

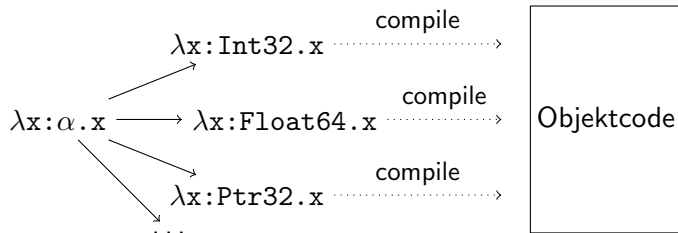
Spezialisierung zur Übersetzungszeit



- Für jede Datenrepräsentation wird separater Code erzeugt

# Statische Monomorphisierung

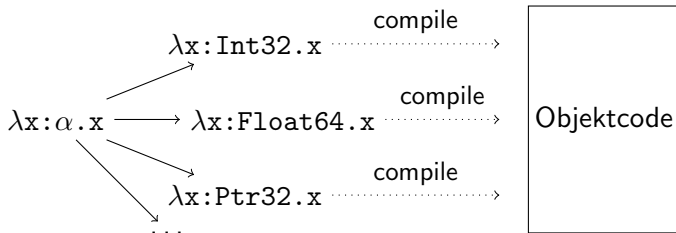
Spezialisierung zur Übersetzungszeit



- ▶ Für jede Datenrepräsentation wird separater Code erzeugt
- + Sehr effizient zur Laufzeit
- Code-Duplikation
- Längere Übersetzungszeiten
- Evtl. Probleme bei unterschiedlichen Compiler-Versionen und gemischten Kompilaten

# Statische Monomorphisierung

Spezialisierung zur Übersetzungszeit



- ▶ Für jede Datenrepräsentation wird separater Code erzeugt
- + Sehr effizient zur Laufzeit
- Code-Duplikation
- Längere Übersetzungszeiten
- Evtl. Probleme bei unterschiedlichen Compiler-Versionen und gemischten Kompilaten

Beispiele: C++ Templates, MLton, JHC, Ada...

## Einführung

- Polymorphie in Programmiersprachen
- Implementierung polymorpher Funktionen
- Statische Monomorphisierung

## Laufzeit-Monomorphisierung

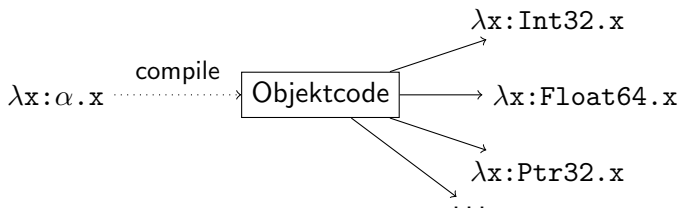
- Allgemeines Prinzip
- Quellsprache
- Implementierung
- Erweiterungen

## Abschluss

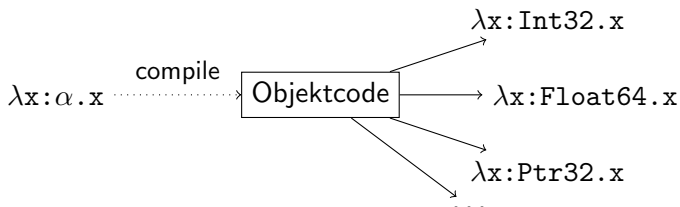
- Weitere Arbeiten
- Zusammenfassung

**Code-Erzeugung zur Laufzeit!**

## Code-Erzeugung zur Laufzeit!



## Code-Erzeugung zur Laufzeit!



- ▶ Für jede Datenrepräsentation wird seperater Code erzeugt – aber zur Laufzeit
- ▶ Spezialisierung erfolgt nur nach Bedarf und so spät wie möglich

- ▶ Vorteile:
  - ▶ Nur tatsächlich benötigter Code wird generiert
  - ▶ Compiler kann adaptiv optimieren
    - ▶ abhängig von Konfiguration
    - ▶ abhängig von Maschineneigenschaften
  - ▶ Laden und Kompilieren bei Bedarf ohne Verzögerungen im laufenden Betrieb

# Laufzeit-Monomorphisierung – Vor- und Nachteile

- ▶ Vorteile:
  - ▶ Nur tatsächlich benötigter Code wird generiert
  - ▶ Compiler kann adaptiv optimieren
    - ▶ abhängig von Konfiguration
    - ▶ abhängig von Maschineneigenschaften
  - ▶ Laden und Kompilieren bei Bedarf ohne Verzögerungen im laufenden Betrieb
- ▶ Nachteile:
  - ▶ Langsamere Ausführung
  - ▶ Optimierter Code muss Aufwand der Kompilierung ausgleichen
  - ▶ aufwändiges Laufzeitsystem: Compiler, Code-Management

|         |                           |                  |
|---------|---------------------------|------------------|
| $t ::=$ | Int                       | Ganzzahltyp      |
|         | Float                     | Gleitkommatyp    |
|         | $\alpha$                  | Typvariable      |
|         | $\bar{t} \rightarrow t_2$ | Funktionsstyp    |
|         | $\forall \alpha. t$       | Universeller Typ |

# Quellsprache – Ausdrücke und Programme

|          |       |  |                         |
|----------|-------|--|-------------------------|
| $e$      | $::=$ | $x$                                      | Variable                |
|          |       | $\lambda \overline{x} : \overline{t}. e$ | Abstraktion             |
|          |       | $e_1 \ \overline{e}$                     | Applikation             |
|          |       | $\Lambda \alpha. e$                      | Typabstraktion          |
|          |       | $e [t]$                                  | Typapplikation          |
|          |       | $i$                                      | Ganzzahlkonstante       |
|          |       | $g$                                      | Gleitkommazahlkonstante |
|          |       | $e_1 \oplus [t] e_2$                     | Arithmetische Operation |
| $\oplus$ | $::=$ | $+ \mid - \mid *$                        | Arithmetik              |
| $f$      | $::=$ | $x = e$                                  | Funktionsgleichung      |
| $\Pi_s$  | $::=$ | $f_1; \dots; f_n; e \quad (n \geq 0)$    | Quellprogramm           |

## Beispiel

Programm:

```
double =  $\Lambda\alpha.\lambda x: \alpha. x + [\alpha] x;$   
double [Int] 21
```

## Beispiel

Programm:

```
double =  $\Lambda\alpha.\lambda x: \alpha. x +[\alpha] x$ ;  
double [Int] 21
```

*double* [Int] 21  $\hookrightarrow$  ( $\Lambda\alpha.\lambda x: \alpha. x +[\alpha] x$ ) [Int] 21

## Beispiel

Programm:

```
double =  $\Lambda\alpha.\lambda x: \alpha. x +[\alpha] x$ ;  
double [Int] 21
```

```
double [Int] 21       $\hookrightarrow$  ( $\Lambda\alpha.\lambda x: \alpha. x +[\alpha] x$ ) [Int] 21  
                      $\hookrightarrow$  ( $\lambda x: \text{Int}. x +[\text{Int}] x$ ) 21
```

## Beispiel

Programm:

```
double =  $\Lambda\alpha.\lambda x: \alpha. x +[\alpha] x;$   
double [Int] 21
```

```
double [Int] 21       $\hookrightarrow$  ( $\Lambda\alpha.\lambda x: \alpha. x +[\alpha] x$ ) [Int] 21  
                     $\hookrightarrow$  ( $\lambda x: \text{Int}. x +[\text{Int}] x$ ) 21  
                     $\hookrightarrow$  21 +[Int] 21
```

## Beispiel

Programm:

```
double =  $\Lambda\alpha.\lambda x: \alpha. x +[\alpha] x;$   
double [Int] 21
```

```
double [Int] 21       $\hookrightarrow$  ( $\Lambda\alpha.\lambda x: \alpha. x +[\alpha] x$ ) [Int] 21  
                     $\hookrightarrow$  ( $\lambda x: \text{Int}. x +[\text{Int}] x$ ) 21  
                     $\hookrightarrow$  21 +[Int] 21  
                     $\hookrightarrow$  42
```

## Beispiel

Programm:

```
double =  $\Lambda\alpha.\lambda x: \alpha. x +[\alpha] x;$   
double [Int] 21
```

```
double [Int] 21  $\hookrightarrow$  ( $\Lambda\alpha.\lambda x: \alpha. x +[\alpha] x$ ) [Int] 21  
 $\hookrightarrow$  ( $\lambda x: \text{Int}. x +[\text{Int}] x$ ) 21  
 $\hookrightarrow$  21 +[Int] 21  
 $\hookrightarrow$  42
```

```
double [Float] 11.5  $\hookrightarrow$  ( $\Lambda\alpha.\lambda x: \alpha. x +[\alpha] x$ ) [Float] 11.5
```

## Beispiel

Programm:

```
double =  $\Lambda\alpha.\lambda x: \alpha. x +[\alpha] x;$   
double [Int] 21
```

```
double [Int] 21  $\hookrightarrow$  ( $\Lambda\alpha.\lambda x: \alpha. x +[\alpha] x$ ) [Int] 21  
 $\hookrightarrow$  ( $\lambda x: \text{Int}. x +[\text{Int}] x$ ) 21  
 $\hookrightarrow$  21 +[Int] 21  
 $\hookrightarrow$  42
```

```
double [Float] 11.5  $\hookrightarrow$  ( $\Lambda\alpha.\lambda x: \alpha. x +[\alpha] x$ ) [Float] 11.5  
 $\hookrightarrow$  ( $\lambda x: \text{Float}. x +[\text{Float}] x$ ) 11.5
```

## Beispiel

Programm:

```
double =  $\Lambda\alpha.\lambda x: \alpha. x +[\alpha] x$ ;  
double [Int] 21
```

```
double [Int] 21  $\hookrightarrow$  ( $\Lambda\alpha.\lambda x: \alpha. x +[\alpha] x$ ) [Int] 21  
 $\hookrightarrow$  ( $\lambda x: \text{Int}. x +[\text{Int}] x$ ) 21  
 $\hookrightarrow$  21 +[Int] 21  
 $\hookrightarrow$  42
```

```
double [Float] 11.5  $\hookrightarrow$  ( $\Lambda\alpha.\lambda x: \alpha. x +[\alpha] x$ ) [Float] 11.5  
 $\hookrightarrow$  ( $\lambda x: \text{Float}. x +[\text{Float}] x$ ) 11.5  
 $\hookrightarrow$  11.5 +[Float] 11.5
```

## Beispiel

Programm:

```
double =  $\Lambda\alpha.\lambda x: \alpha. x +[\alpha] x;$   
double [Int] 21
```

```
double [Int] 21  $\hookrightarrow$  ( $\Lambda\alpha.\lambda x: \alpha. x +[\alpha] x$ ) [Int] 21  
 $\hookrightarrow$  ( $\lambda x: \text{Int}. x +[\text{Int}] x$ ) 21  
 $\hookrightarrow$  21 +[Int] 21  
 $\hookrightarrow$  42
```

```
double [Float] 11.5  $\hookrightarrow$  ( $\Lambda\alpha.\lambda x: \alpha. x +[\alpha] x$ ) [Float] 11.5  
 $\hookrightarrow$  ( $\lambda x: \text{Float}. x +[\text{Float}] x$ ) 11.5  
 $\hookrightarrow$  11.5 +[Float] 11.5  
 $\hookrightarrow$  23.0
```

- ▶ Typ-Abstraktion wird in Code-Generator übersetzt
- ▶ Typ-Applikation wird in Aufruf eines Code-Generators übersetzt

# Implementierung – Grundidee

- ▶ Typ-Abstraktion wird in Code-Generator übersetzt
- ▶ Typ-Applikation wird in Aufruf eines Code-Generators übersetzt

## Beispiel

|   |               |  |
|---|---------------|--|
| $\Lambda\alpha.\lambda x: \alpha. x + [\alpha] x$ | $\Rightarrow$ | Code-Generator für $\lambda x: \alpha. x + [\alpha] x$ |
| <i>double</i> [Int]                               | $\Rightarrow$ | Aufruf des Code-Generators mit Int                     |

Der Ausdruck

*double* [Int] 21

wird in folgende Code-Sequenz übersetzt:

1. Berechnung des Wertes von *double* (der Code-Generator),
2. Aufruf des Generators mit dem Typ Int und
3. Aufruf der generierten Funktion mit Parameter 21.

Der Ausdruck

*double* [Int] 21

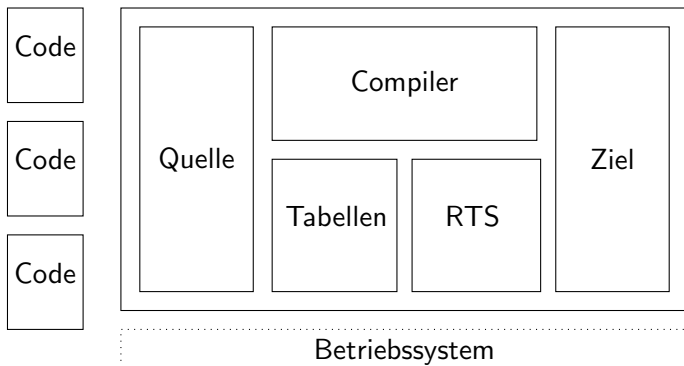
wird in folgende Code-Sequenz übersetzt:

1. Berechnung des Wertes von *double* (der Code-Generator),
2. Aufruf des Generators mit dem Typ Int und
3. Aufruf der generierten Funktion mit Parameter 21.

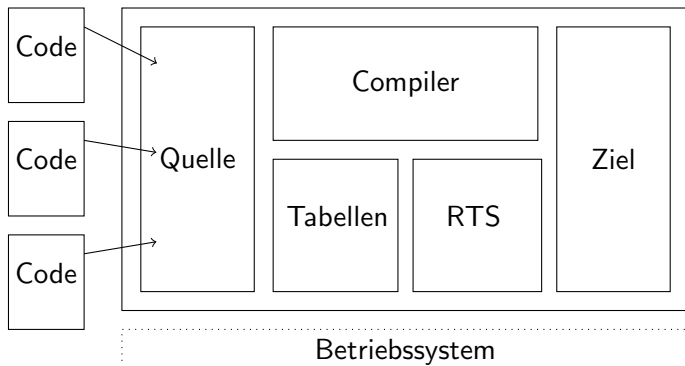
Die Definition von *double* wird in eine Funktion übersetzt, die

1. die abstrakte Syntax des Ausdrucks  $\lambda x: \alpha. x + [\alpha] x$  verwaltet,
2. als Parameter einen Typ erwartet,
3. diesen Typ für  $\alpha$  einsetzt,
4. diesen Ausdruck in Maschinencode übersetzt und
5. einen Funktionszeiger auf diesen Code zurückliefert.

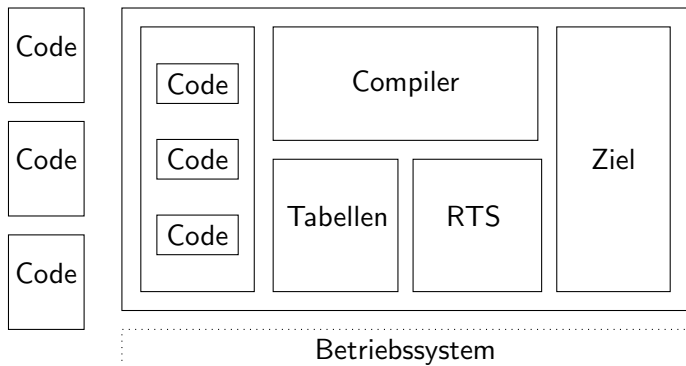
# Systemaufbau



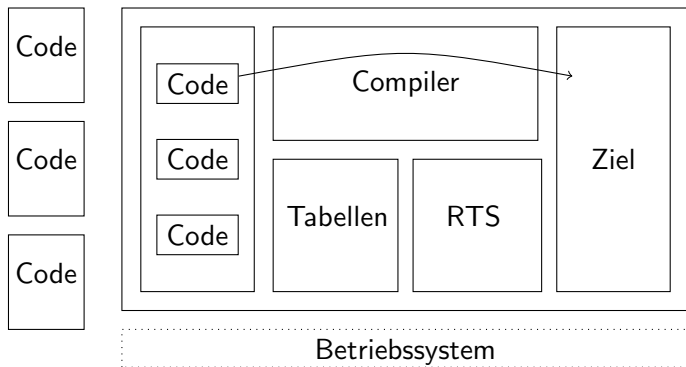
# Systemaufbau



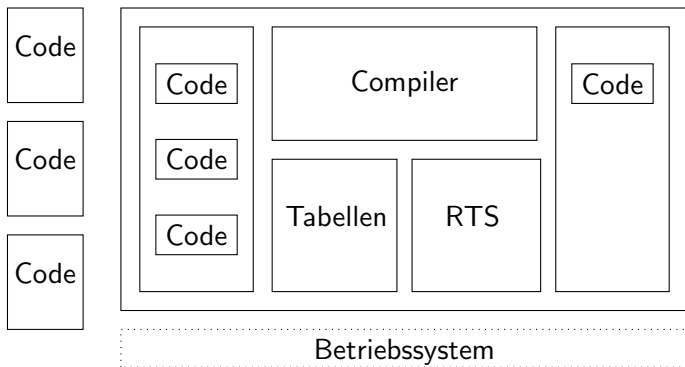
# Systemaufbau



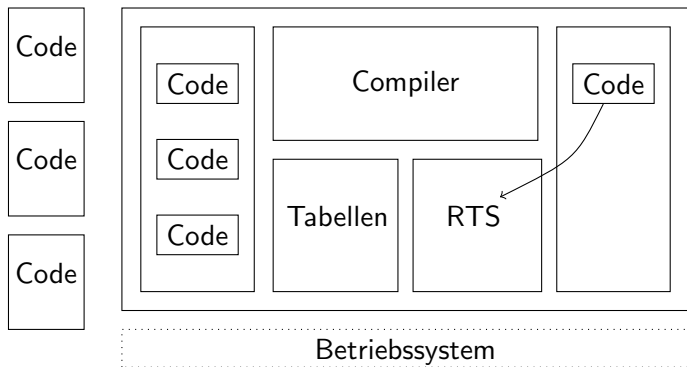
# Systemaufbau



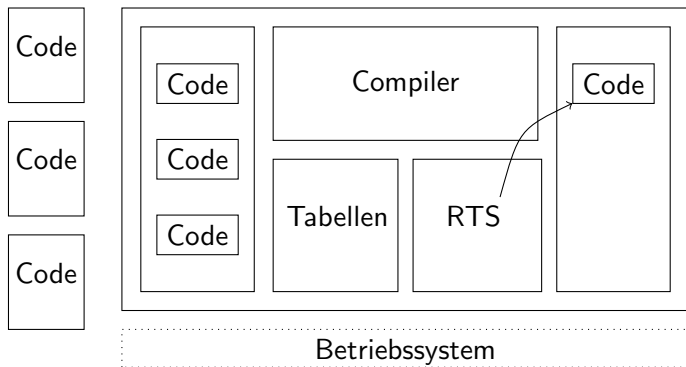
# Systemaufbau



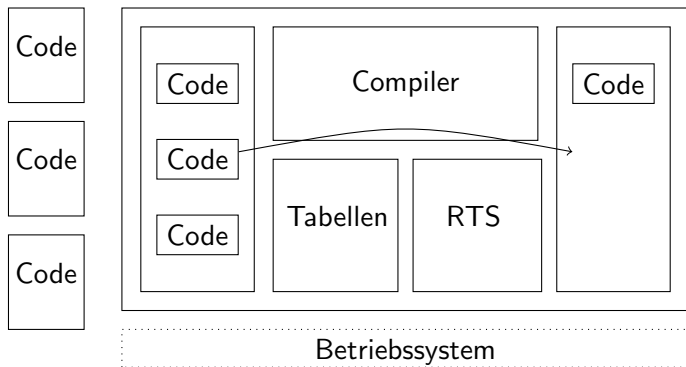
# Systemaufbau



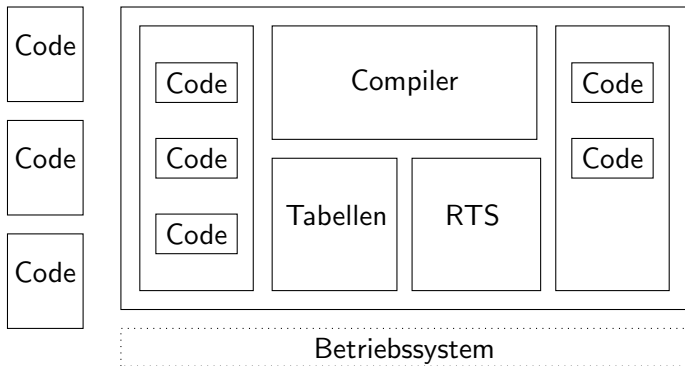
# Systemaufbau



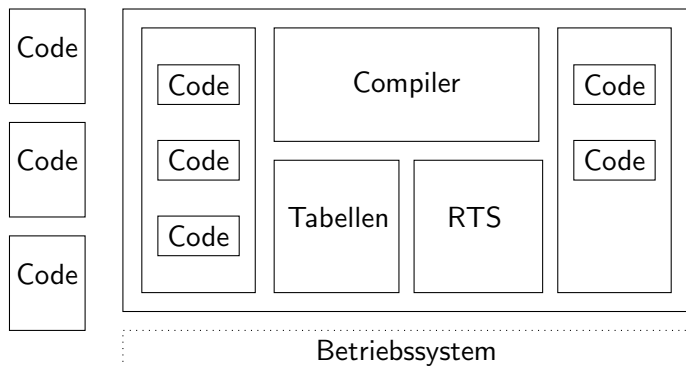
# Systemaufbau



# Systemaufbau



# Systemaufbau



Verwaltung:

- ▶ Kompilierte Code-Fragmente
- ▶ Abbildung (Code-Generator  $\times$  Typ)  $\rightarrow$  Maschinencode
- ▶ Abbildung berechneter Werte  $\rightarrow$  Maschinencode zur Vermeidung von Mehrfach-Übersetzung
- ▶ Geladene Module (beim Laden nach Bedarf)

- ▶ Funktionierender Prototyp
- ▶ Unterstützung von Berechnungen mit Ganz- und Gleitkommazahlen
- ▶ Leistungsvergleich steht noch aus
- ▶ Nur rudimentäres Laufzeitsystem

# Ein Problem der Quellsprache

Beispieldefinition

$$\mathit{double} = \Lambda\alpha.\lambda x : \alpha.x + [\alpha] x$$

# Ein Problem der Quellsprache

Beispieldefinition

$$double = \Lambda \alpha. \lambda x : \alpha. x + [\alpha] x$$

ist **nicht typsicher!**

# Ein Problem der Quellsprache

Beispieldefinition

$$double = \Lambda \alpha. \lambda x : \alpha. x + [\alpha] x$$

ist **nicht typsicher!**

$$double[\forall \alpha. \alpha \rightarrow \alpha] double$$

# Ein Problem der Quellsprache

Beispieldefinition

$$double = \Lambda \alpha. \lambda x : \alpha. x + [\alpha] x$$

ist **nicht typsicher!**

$$\begin{aligned} & double[\forall \alpha. \alpha \rightarrow \alpha] double \\ \hookrightarrow & (\Lambda \alpha. \lambda x : \alpha. x + [\alpha] x)[\forall \alpha. \alpha \rightarrow \alpha] double \end{aligned}$$

# Ein Problem der Quellsprache

Beispieldefinition

$$\mathit{double} = \Lambda\alpha.\lambda x : \alpha.x + [\alpha] x$$

ist **nicht typsicher!**

$$\mathit{double}[\forall\alpha.\alpha \rightarrow \alpha] \mathit{double}$$

$$\hookrightarrow (\Lambda\alpha.\lambda x : \alpha.x + [\alpha] x)[\forall\alpha.\alpha \rightarrow \alpha] \mathit{double}$$

$$\hookrightarrow (\lambda x : \forall\alpha.\alpha \rightarrow \alpha.x + [\forall\alpha.\alpha \rightarrow \alpha] x) \mathit{double}$$

# Ein Problem der Quellsprache

Beispieldefinition

$$\mathit{double} = \Lambda\alpha.\lambda x : \alpha.x +[\alpha] x$$

ist **nicht typsicher!**

$$\begin{aligned} & \mathit{double}[\forall\alpha.\alpha \rightarrow \alpha] \mathit{double} \\ \hookrightarrow & (\Lambda\alpha.\lambda x : \alpha.x +[\alpha] x)[\forall\alpha.\alpha \rightarrow \alpha] \mathit{double} \\ \hookrightarrow & (\lambda x : \forall\alpha.\alpha \rightarrow \alpha.x +[\forall\alpha.\alpha \rightarrow \alpha] x) \mathit{double} \\ \hookrightarrow & \mathit{double} +[\forall\alpha.\alpha \rightarrow \alpha] \mathit{double} \end{aligned}$$

# Ein Problem der Quellsprache

Beispieldefinition

$$\mathit{double} = \Lambda\alpha.\lambda x : \alpha.x + [\alpha] x$$

ist **nicht typsicher!**

$$\begin{aligned} & \mathit{double}[\forall\alpha.\alpha \rightarrow \alpha] \mathit{double} \\ \hookrightarrow & (\Lambda\alpha.\lambda x : \alpha.x + [\alpha] x)[\forall\alpha.\alpha \rightarrow \alpha] \mathit{double} \\ \hookrightarrow & (\lambda x : \forall\alpha.\alpha \rightarrow \alpha.x + [\forall\alpha.\alpha \rightarrow \alpha] x) \mathit{double} \\ \hookrightarrow & \mathit{double} + [\forall\alpha.\alpha \rightarrow \alpha] \mathit{double} \\ \hookrightarrow & ??? \end{aligned}$$

# Ein Problem der Quellsprache

Beispieldefinition

$$double = \Lambda \alpha. \lambda x : \alpha. x + [\alpha] x$$

ist **nicht typsicher!**

$$\begin{aligned} & double[\forall \alpha. \alpha \rightarrow \alpha] double \\ \hookrightarrow & (\Lambda \alpha. \lambda x : \alpha. x + [\alpha] x)[\forall \alpha. \alpha \rightarrow \alpha] double \\ \hookrightarrow & (\lambda x : \forall \alpha. \alpha \rightarrow \alpha. x + [\forall \alpha. \alpha \rightarrow \alpha] x) double \\ \hookrightarrow & double + [\forall \alpha. \alpha \rightarrow \alpha] double \\ \hookrightarrow & ??? \end{aligned}$$

Lösung mit Typklassen (à la Haskell):

$$double = \Lambda \text{Num } \alpha \Rightarrow \alpha. \lambda x : \alpha. x + [\alpha] x$$

Variable  $\alpha$  kann nur noch mit Typen instanziiert werden, die zur Klasse Num gehören.

Implementierung von Typklassen:

- ▶ Jede Klasse definiert einen Record-Typ (*dictionary type*)
- ▶ Jede Instanz einen Wert dieses Typs (*dictionary*)
- ▶ Benutzung einer Operation wird in Record-Zugriff übersetzt
- ▶ Pro Klassen-Constraint wird ein Parameter hinzugefügt

# Dictionary Passing

Implementierung von Typklassen:

- ▶ Jede Klasse definiert einen Record-Typ (*dictionary type*)
- ▶ Jede Instanz einen Wert dieses Typs (*dictionary*)
- ▶ Benutzung einer Operation wird in Record-Zugriff übersetzt
- ▶ Pro Klassen-Constraint wird ein Parameter hinzugefügt

## Beispiel

$$\begin{aligned} double &= \Lambda \text{Num } \alpha \Rightarrow \alpha. \lambda x : \alpha. x + [\alpha] x \Rightarrow \\ double &= \lambda dNum. \lambda x. x \ dNum. + x \end{aligned}$$

# Dictionary Passing

Implementierung von Typklassen:

- ▶ Jede Klasse definiert einen Record-Typ (*dictionary type*)
- ▶ Jede Instanz einen Wert dieses Typs (*dictionary*)
- ▶ Benutzung einer Operation wird in Record-Zugriff übersetzt
- ▶ Pro Klassen-Constraint wird ein Parameter hinzugefügt

## Beispiel

$$\begin{aligned} double &= \Lambda \text{Num } \alpha \Rightarrow \alpha. \lambda x : \alpha. x + [\alpha] x \Rightarrow \\ double &= \lambda dNum. \lambda x. x \ dNum. + x \end{aligned}$$

Dynamischer Funktionsaufruf  $dNum.+$  (entspricht virtuellem Methodenaufruf in OO-Sprachen) kann durch Laufzeit-Monomorphisierung in statischen Aufruf übersetzt werden. (Ggf. weitere Optimierung.)

## Einführung

- Polymorphie in Programmiersprachen
- Implementierung polymorpher Funktionen
- Statische Monomorphisierung

## Laufzeit-Monomorphisierung

- Allgemeines Prinzip
- Quellsprache
- Implementierung
- Erweiterungen

## Abschluss

- Weitere Arbeiten
- Zusammenfassung

- ▶ Einsatzfähige Implementierung
- ▶ Erweiterung auf Typklassen
- ▶ Code-Speicher-Management
- ▶ Vollständiges Laufzeitsystem
- ▶ Dynamische Leistungsmessung und -anpassung
- ▶ Optimierung
- ▶ Leistungsvergleiche mit anderen Ansätzen

## Vorgestellte Arbeiten:

- ▶ Allgemeines Prinzip zur Laufzeit-Monomorphisierung
- ▶ Anwendung auf konkrete Quellsprache
- ▶ Beschreibung eines Prototyps
- ▶ Aussicht auf Erweiterungen: Typklassen

Vorgestellte Arbeiten:

- ▶ Allgemeines Prinzip zur Laufzeit-Monomorphisierung
- ▶ Anwendung auf konkrete Quellsprache
- ▶ Beschreibung eines Prototyps
- ▶ Aussicht auf Erweiterungen: Typklassen

**Vielen Dank!**