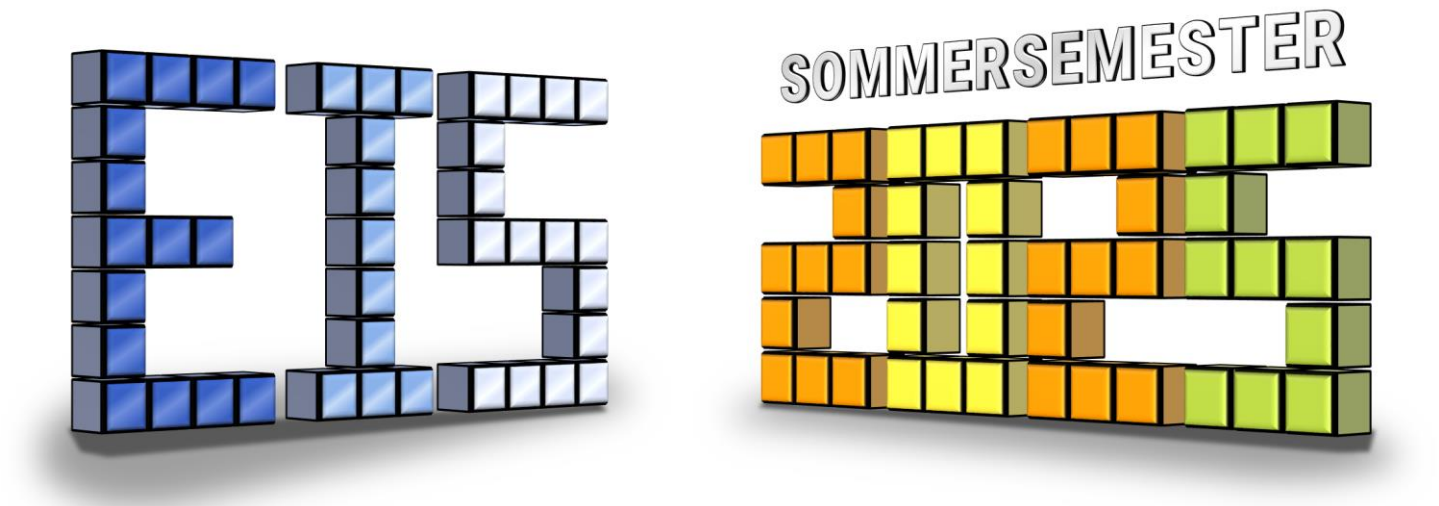
EINFÜHRUNG IN DIE SOFTWAREENTWICKLUNG

Sommersemester 2025



Foliensatz #9

Funktionale Programmierung

Michael Wand
Institut für Informatik
Michael.Wand@uni-mainz.de

Techniken

Strukturierung von Programmen

- Prozedural
- Objekt-orientiert
- Funktional
- Meta-Programmierung

Funktionale Programmierung

Funktionale Programmierung

Grundidee

- Genauso wie prozedural

Also wo ist der Unterschied?

- Erweiterung:
  Funktionen als Datentyp („Code als Daten“)

Funktionale Programmierung

„Pure Functional Programming“

- Spezielle Variante von FP
  - Richtung *innerhalb* von FP
  - Manchmal synonym mit FP verwendet
- Mathematisch angehaucht

Prinzipien

- Keine (oder eingeschränkte) Änderung von Variablen
  - „Avoiding mutable state“
- Dies impliziert: Rekursion statt Schleifen

Funktionen

Funktion

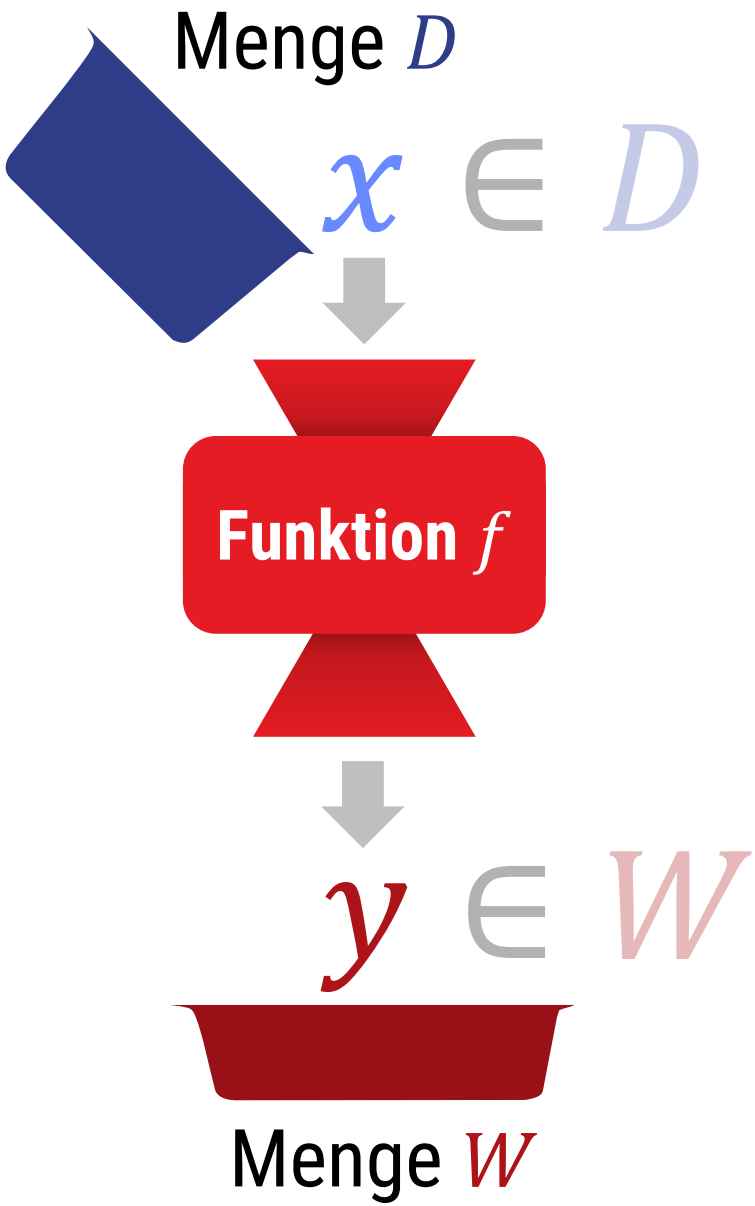
- Zuordnung von Elementen

$$f: D \to W$$

$$x \mapsto y = f(x)$$

$$x \in D, \quad y \in W$$

- Deterministisch
  - Gleiche Eingabe → gleiche Ausgabe
- Ergebnis muß für alle Eingaben definiert sein
  - D = “Definitionsmenge” (domain)
  - W = “Ziel/Wertemenge” (codomain / target set)



Funktionen

Zusammengesetzte Ein- / Ausgabe

- Signatur:

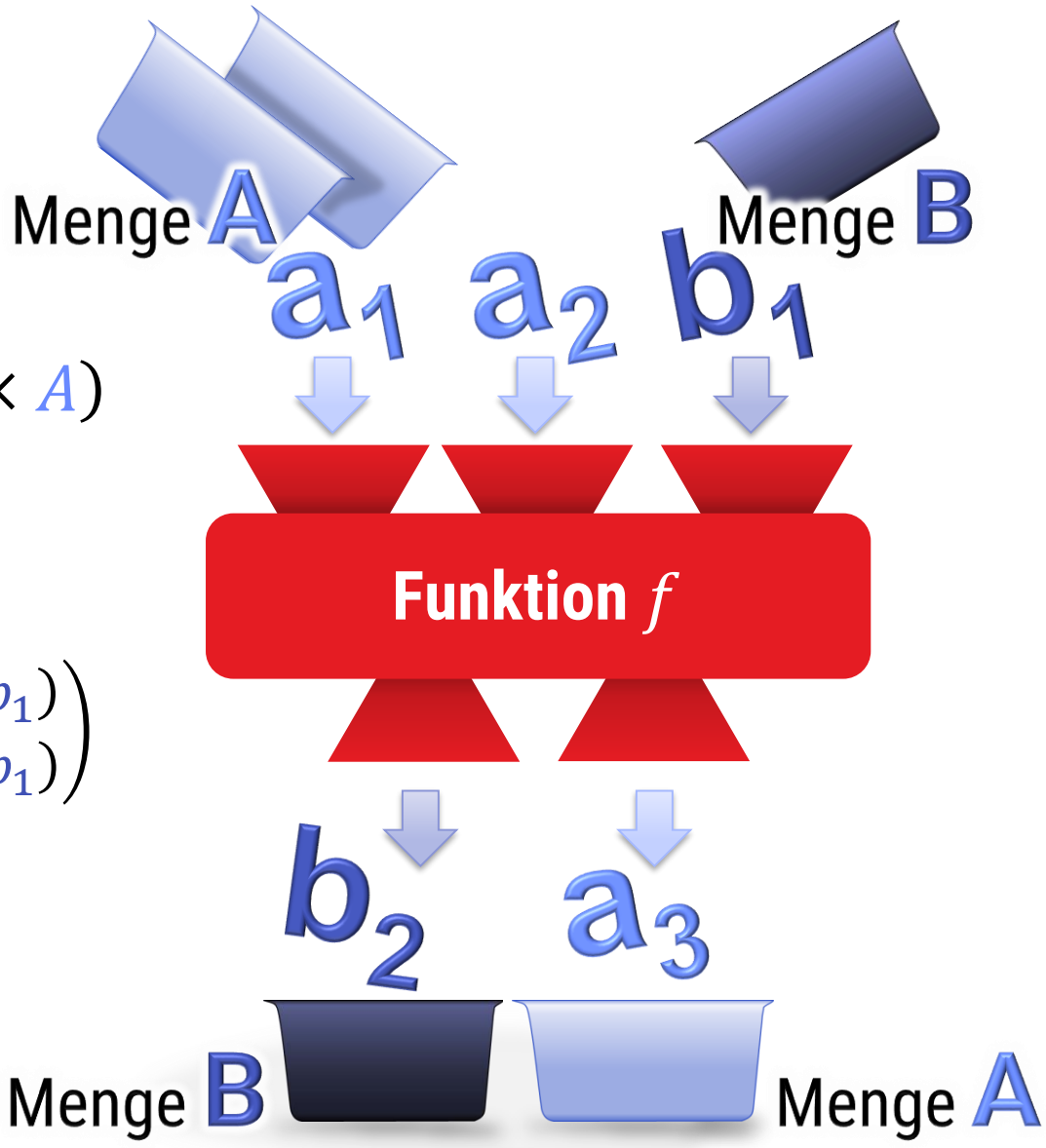
  $f : (A \times A \times B) \rightarrow (B \times A)$

- Zuordnung:

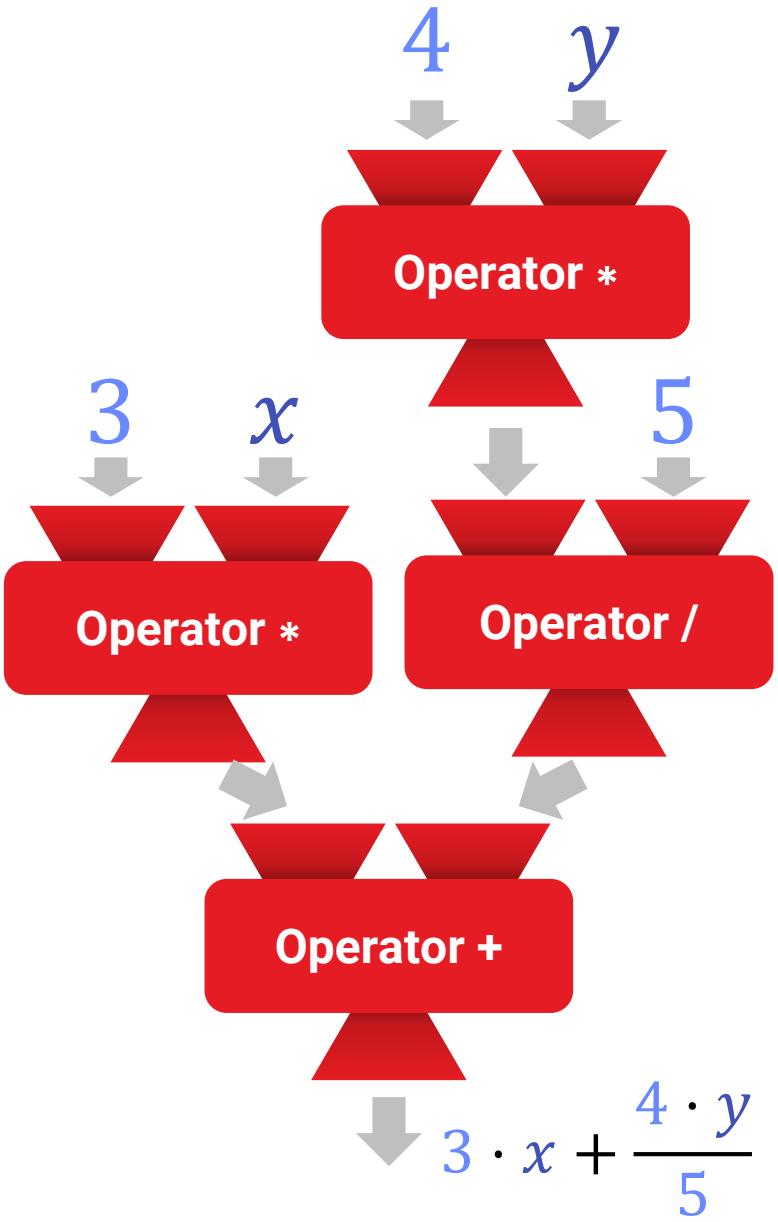
  $a_1, a_2, b_1 \mapsto f(a_1, a_2, b_1)$

  $= \begin{pmatrix} f_1(a_1, a_2, b_1) \\ f_2(a_1, a_2, b_1) \end{pmatrix}$

  $= \begin{pmatrix} b_2 \\ a_3 \end{pmatrix}$

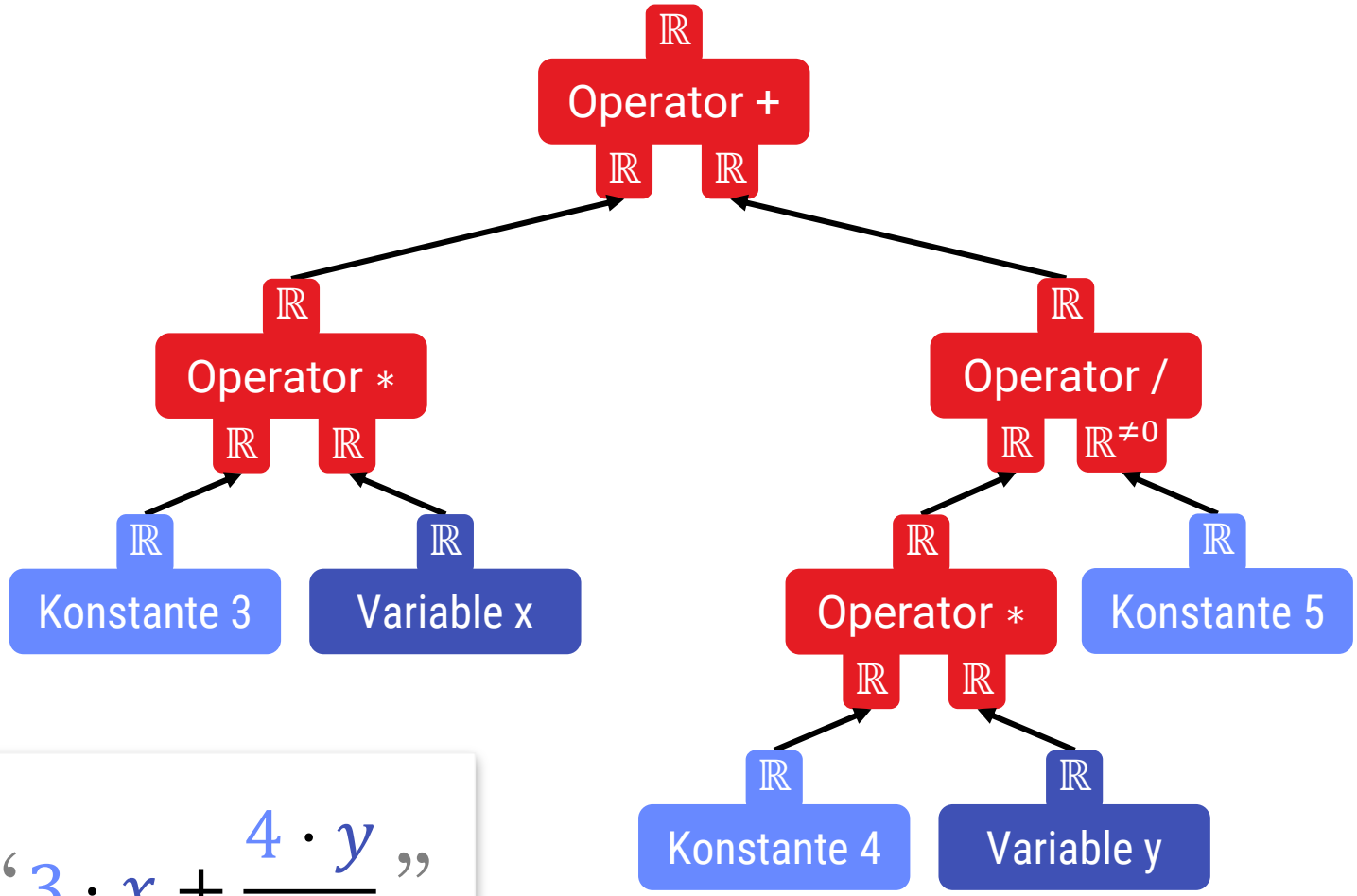


Arithmetische Ausdrücke



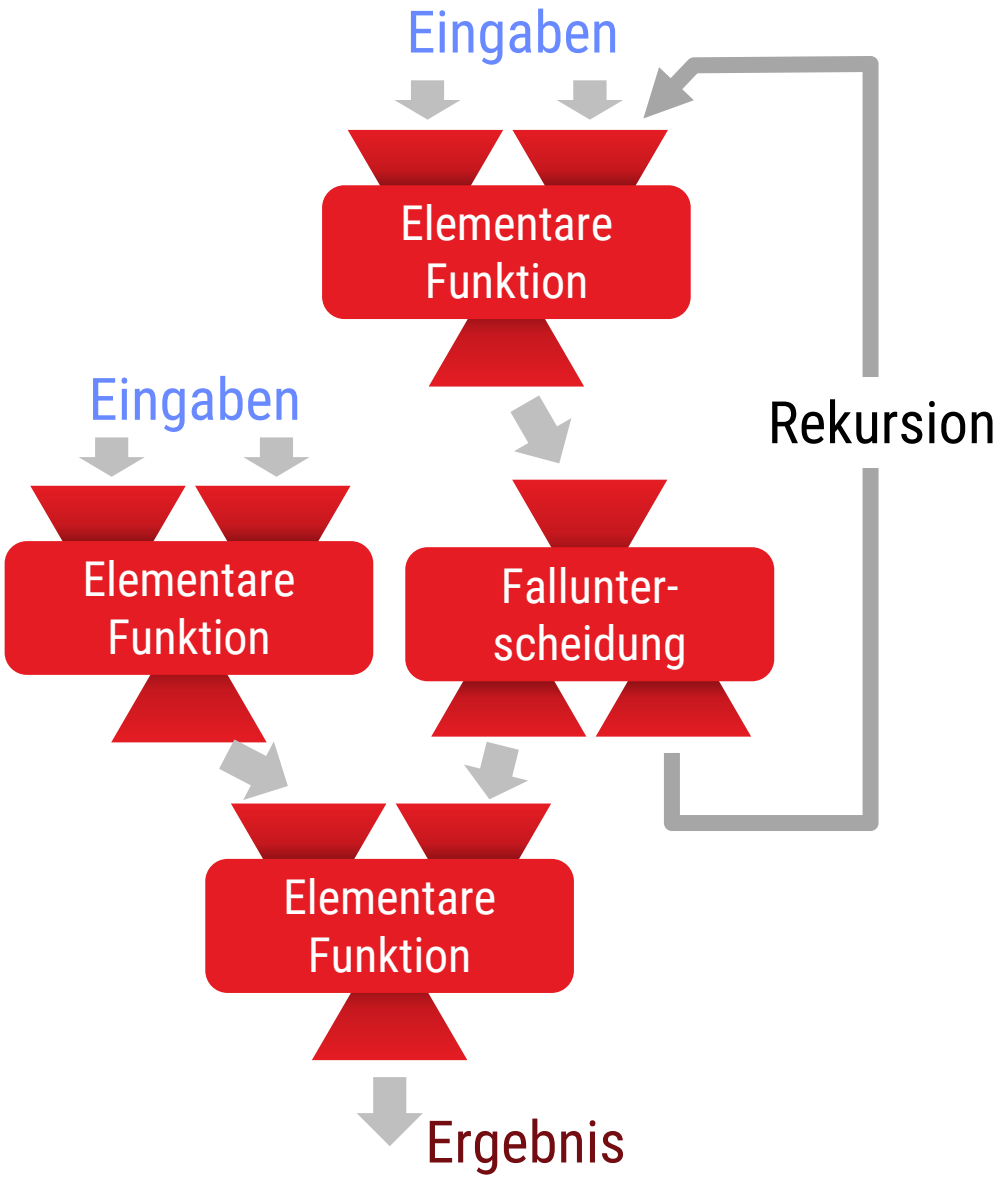
“$3 \cdot x + \frac{4 \cdot y}{5}$”

Statisch Typisiert



$$\text{“}3 \cdot x + \frac{4 \cdot y}{5}\text{”}$$

Funktionale Programmierung



Bausteine

- Elementare Funktionen
- Rekursion

Beispiel

$$f(x) = \begin{cases} 1, & \text{falls } x = 0 \\ x \cdot f(x-1), & \text{sonst} \end{cases}$$

Turing-mächtig

- „Funktionale Programmierung“
- Modell ignoriert Interaktion, I/O u.ä.

Fakultät Funktional

Fakutät – imperativ (prozedural, Python)

```
def factorial(n: int) -> int:
    result: int = 1                    # mutable state
    for result: int in range(n+1, 2):
       result = result * i             # mutation of state
    return result
}
```

Fakutät – „rein“ funktional (Python)

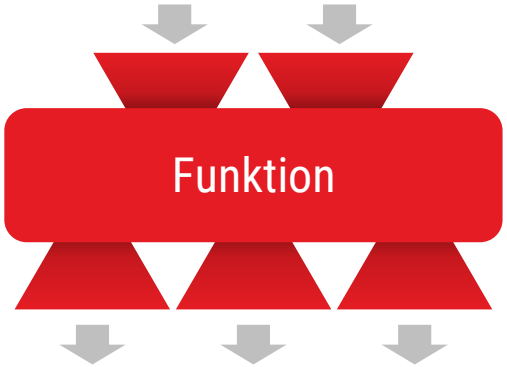
```
def factorial(n: int) -> int:
    return 1 if n <= 1 else factorial(n-1)*n  # no (explicit) mutable state
    # Python “if … else” Ausdruck wertet nur zutreffenden Fall aus!
    # Python unterstützt damit “pure functional style”
```

C++ – for fun

```
int factorial(int n) { return (n <= 1) ? 1 : factorial(n-1)*n; }
```

Funktionen als Bausteine

Bausteine



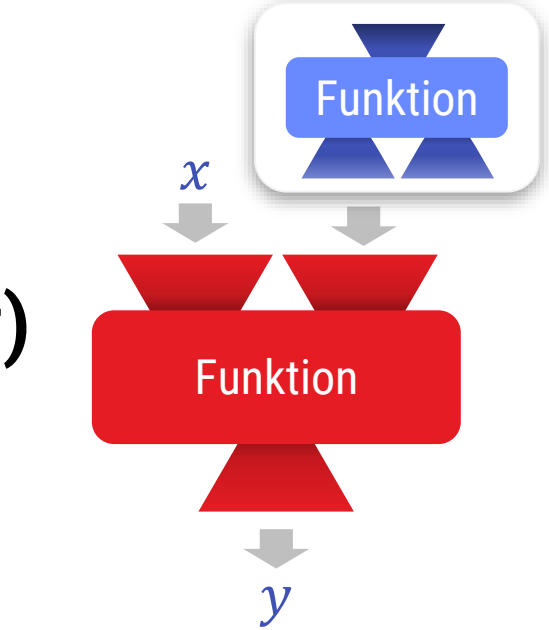
- Funktionen kapseln „Codeschnipsel“
- Definierte Schnittstelle
  - Ein- und Ausgabe spezifiziert
  - Statisch typisiert (in Scala, C, C++, Pascal, etc.)
  - Dynamisch typisiert in Python, JavaScript u.ä.
- Composition von Funktionen ergibt bei Bedarf ein „neu zusammengestelltes“ Programm

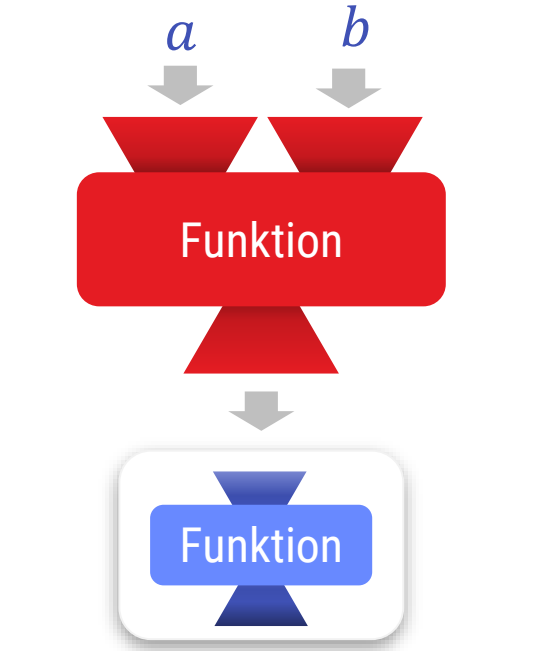
Higher Order…

Higher-Order Functions
(dt.: Funktionen höherer Ordnung)

- Funktionen, die andere Funktionen
  - Als Parameter übergeben bekommen
  - Als Rückgabe-„Wert“ haben
  - Oder beides

→ **Konfiguration zur Laufzeit**





Rein Funktionale Programmierung

Nochmal: „Pure Functional Programming“

- Keine (oder eingeschränkte) Änderung von Variablen
  - „Avoiding mutable state“
- „Rekursion statt Schleifen“

Warum?

Problem von Variablenänderung („mutable state“)

- Problematisch sind globale Variablen, bezogen auf aktuellen Kontext

Herausforderung beim Verstehen

- Hoch: Variablen außerhalb der Funktion, die implizit geändert werden (keine Parameter)
- Mittel: Referenzen auf äußere Variablen (in OOP auch: `self`)
- Gering: Lokale Variabeln, z.B. in Schleifen

Reasoning about mutable state

Es ist schwierig(er) zu verstehen…

- …wie Code funktioniert, der nicht-lokale Effekte hat
  - Änderungen von Variablen verändern Bedeutung
- „Pure Functions“: kein Schreiben, nur Rückgabe
- Reine Funktionen verhalten sich einfacher
  - gleiche Eingaben → gleiche Ausgabe
  - „Referentielle Transparenz“
    - Man könnte Werte einsetzen
- Dies erleichtert
  - Verständnis (nicht immer! – es gibt einige Gegenbeispiele)
  - Beweis mit Invarianten (auch nicht immer)
  - Parallelisierung (vermeided Zugriffskonflikte bez. shared memory)

Allgemeine Leitlinien

Gezielte Änderungen

- Reine Funktionen sind oft einfacher zu verstehen
  - Nicht immer, da Algorithmus komplizierter werden kann
- Globale Seiteneffekte (state mutation) zu minimieren ist immer eine gute Idee
  - Lokale Seiteneffekte (z.B. Schleifenvariablen) eher unkritisch
- Prozedurale / OOP Entwürfe profitieren auch davon

Architekturen, die Änderungen kapseln

- Speziell ausgezeichnete Mechanismen, wenn persistente/globale Daten geändert werden

Entwurfsstrategien

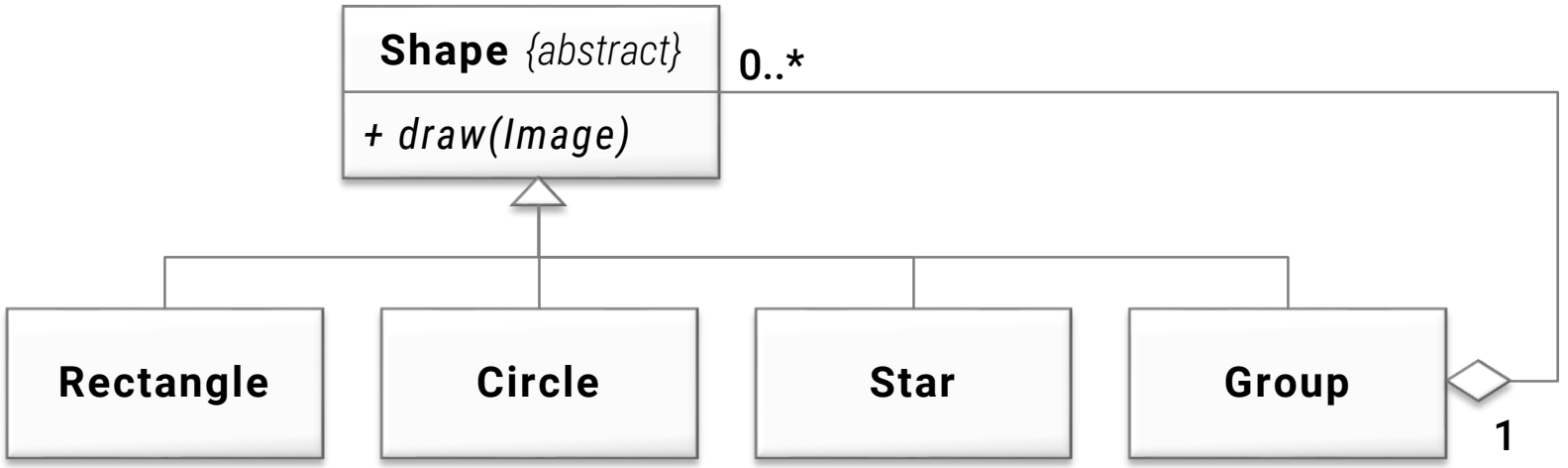
Entwurfsstrategie

- Ähnlich wie prozedurale Programmierung
- Datenfluss (ggF. rekursiv)
- Vermeidung von Änderungen von Variablen (die global sind bez. auf aktuelle Funktion)
  - Statt dessen: Parameterübergabe
  - „Streng funktional (pure)“: Rekursion statt Schleifen

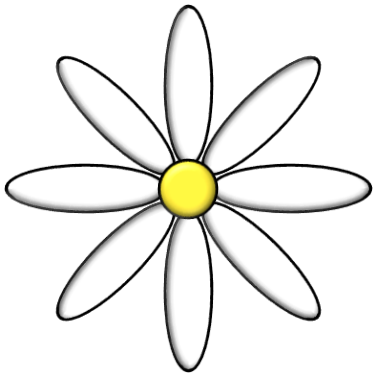
Funktionale Variante des Zeichenprograms

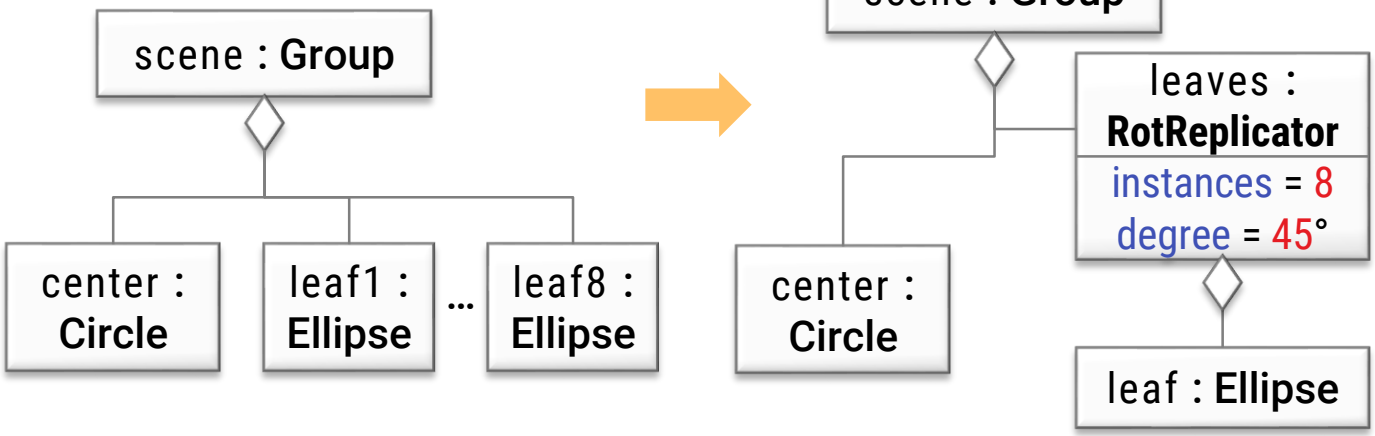
„Datenflussgraphen“

Komponentenhierarchie



Feature: Programmable Repetition?

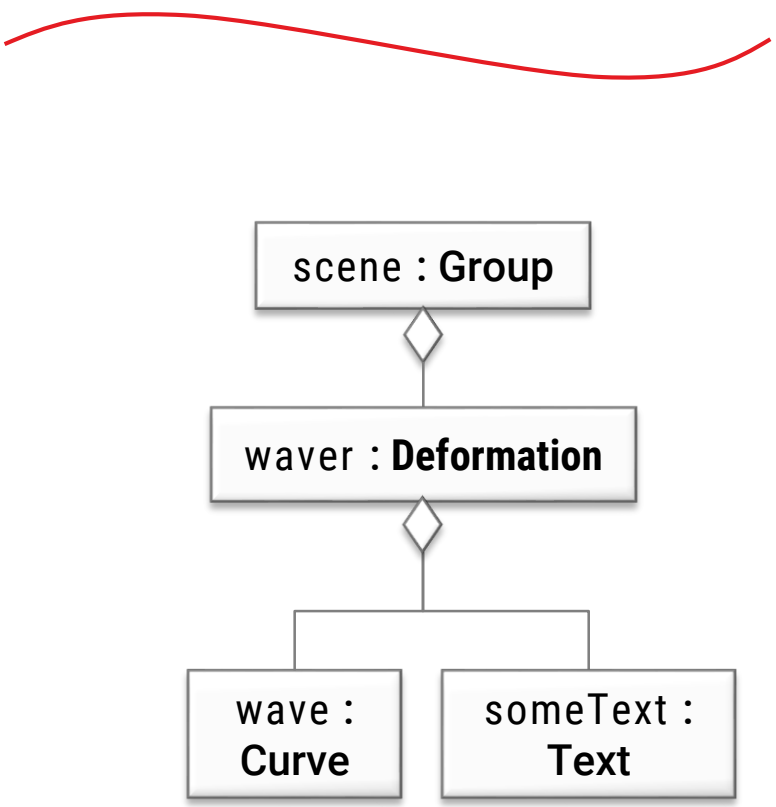




Anderes Bespiel

Wavy-Text!

+



Wavy-Text!

Reifikation von „draw()“-Befehlen

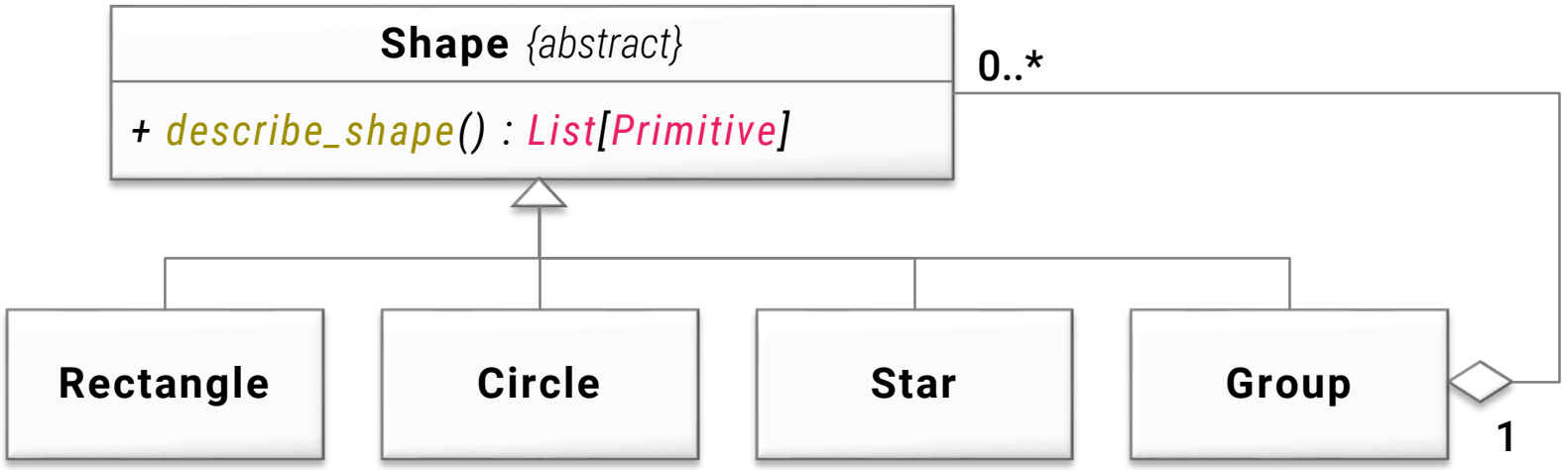
Limitierung

- Zeichnen via „`draw(img: Painter)`“, z.B.
  - `img.drawEllpise(center, radius1, radius2)`
- Methodenaufrufe kann man wiederholen, aber nicht verändern

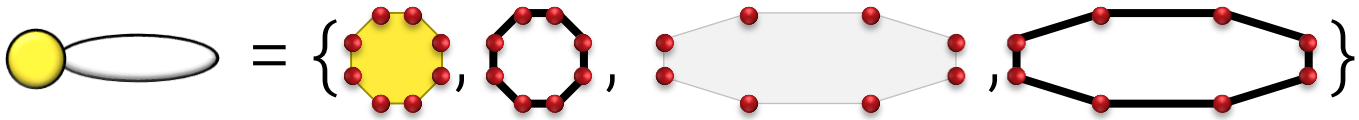
Reifikation

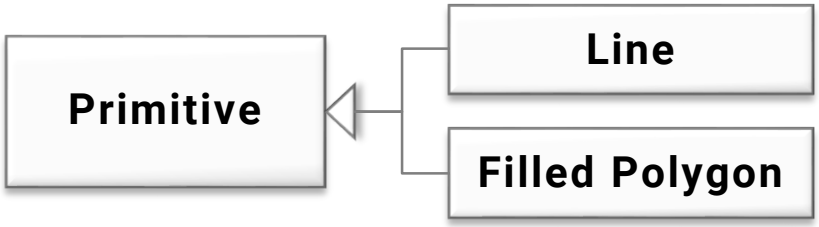
- Liste von Datenobjekten
- Elementare „Primitive“: Linien, Polygone

Komponentenhierarchie



Repräsentation: Liste von Primitiven





Flexibler

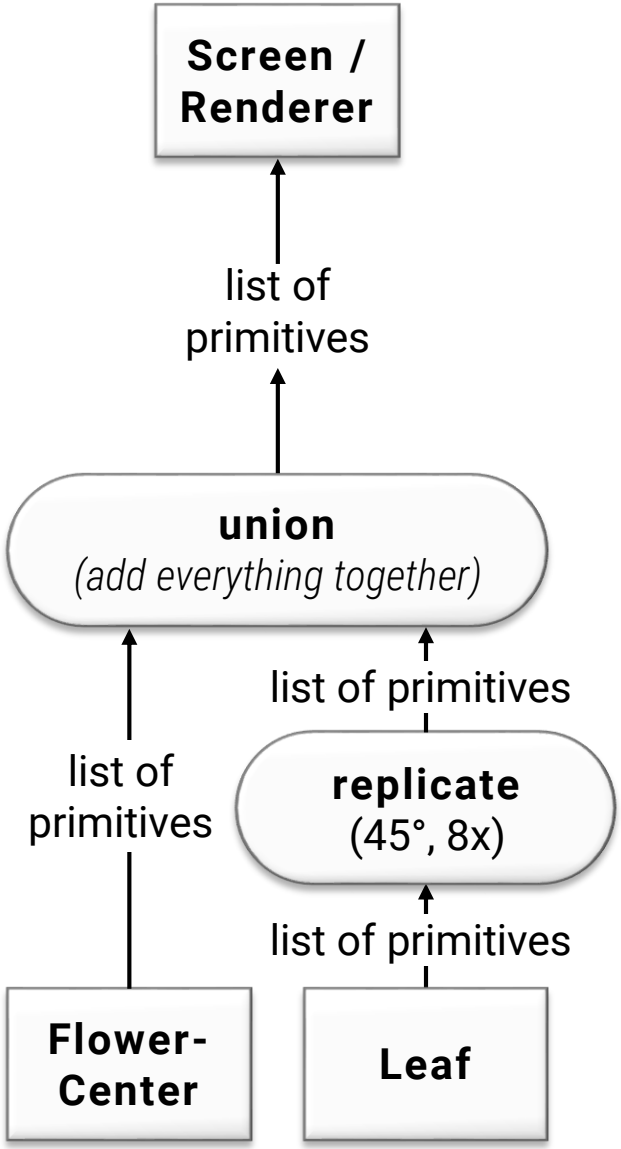
Nun können wir unser Problem lösen

- Jeder Knoten im Objektbaum kann eine geometrische Beschreibung
  - Aufruf „`describe_shape()`“ des Kindknoten
  - Liste von Primitiven (zuvor: Operationen in „`draw()`“)
- Beliebige Berechnungen damit möglich
- Zusammenstellen einer neuen Liste

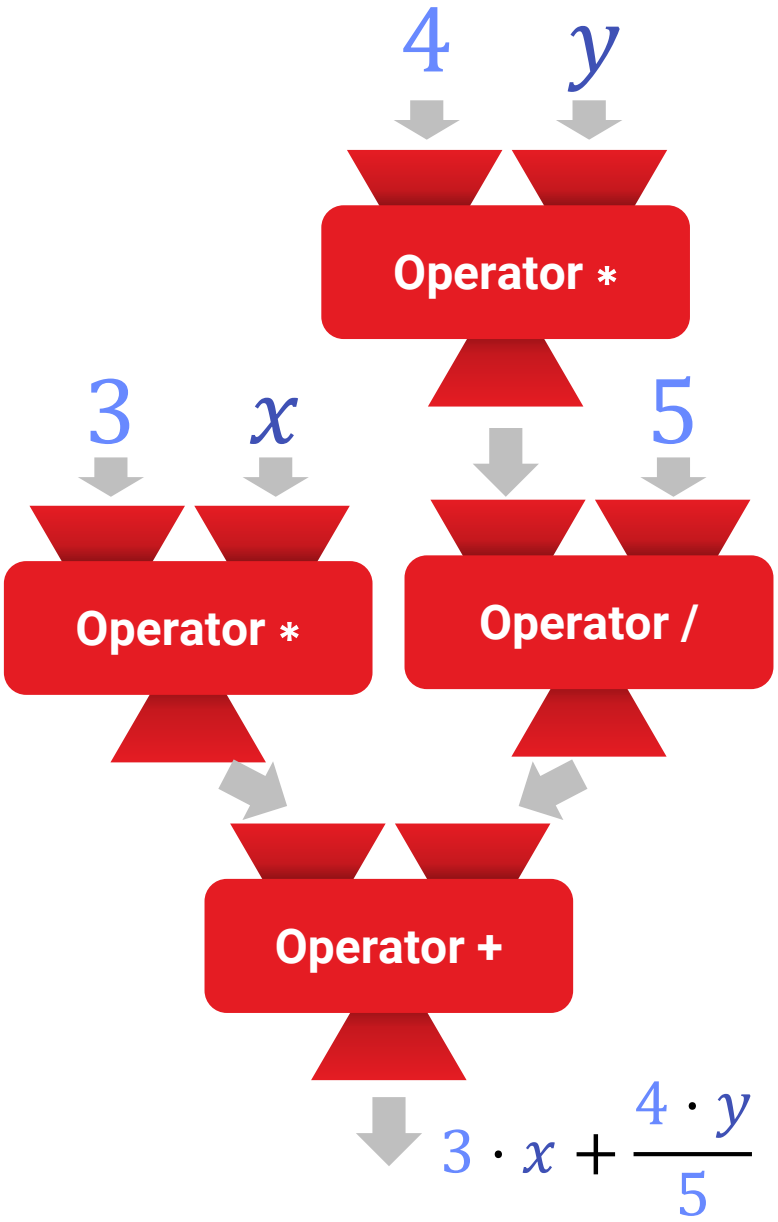
Datenflussgraph

Architektur

- Knotenobjekte sind „Funktionen“
  - 0…* Eingaben
  - 0…* Ausgaben
  - Beliebige Berechnung, um Eingaben in Ausgaben zu transformieren
- Jedes Knotenobjekt repräsentiert eine Funktion
- „Funktionales“ Muster: Kompositionshierarchie/Graph
- Oft in der Praxis angewandt

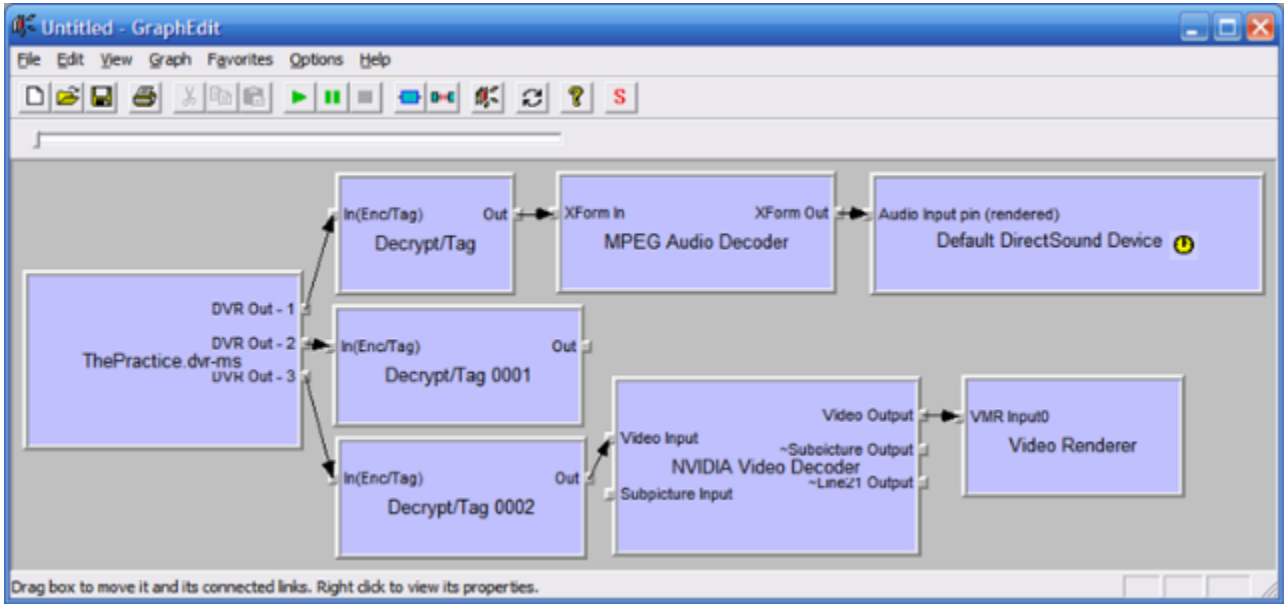


Datenflussgraphen



“$3 \cdot x + \frac{4 \cdot y}{5}$”

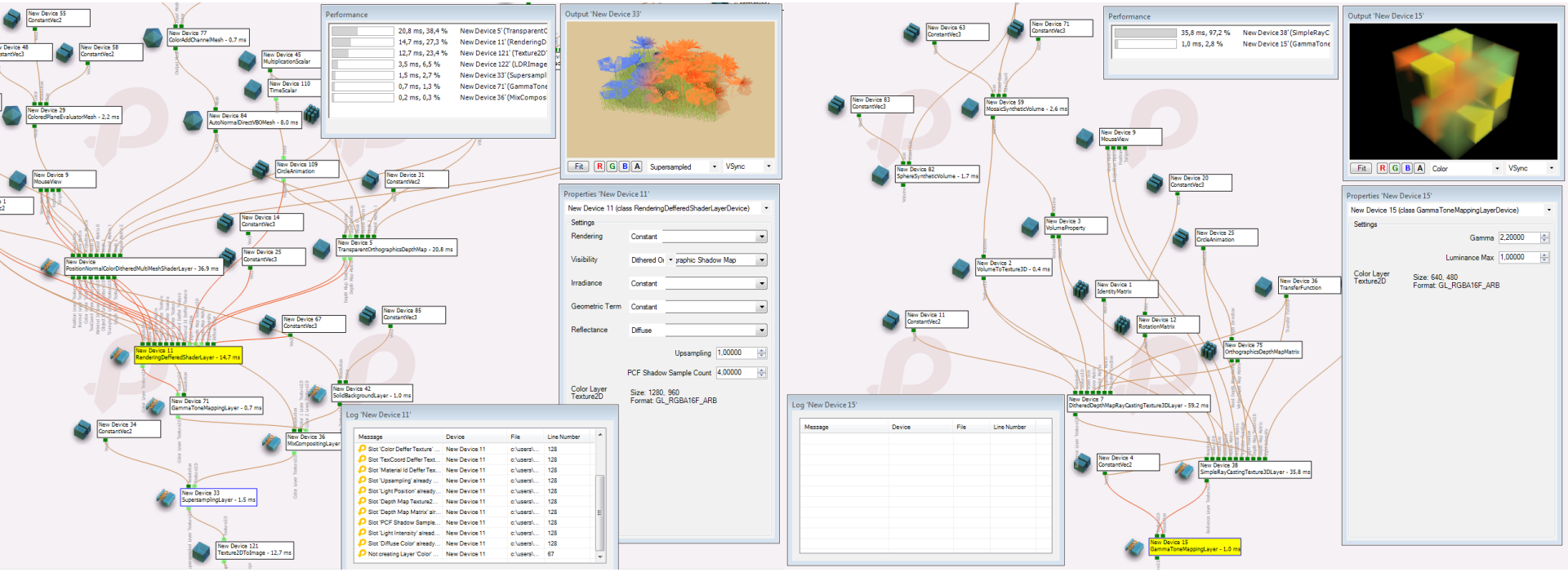
Beispiele



Microsoft „DirectShow“

- Graph von Audio / Video Prozessoren
- Data Flow: Bild-, Video-, Audio-Puffer

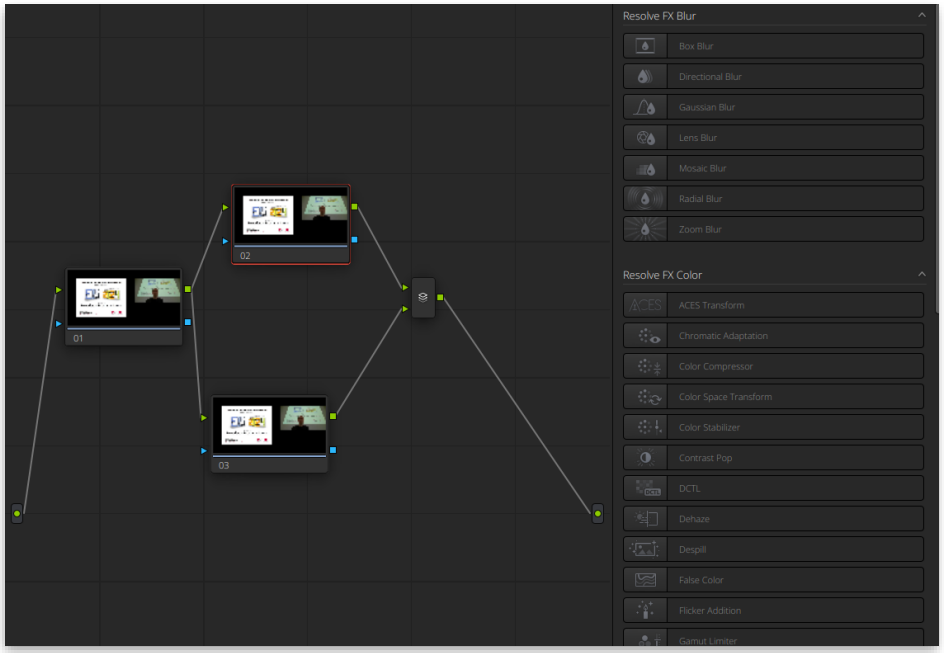
Beispiele



„**Plexus**“ (Tobias Ritschel, UCL/MPI Saarbrücken)

- 2D Bildverarbeitung und 3D Rendering
- Daten werden mit der GPU verarbeitet

Beispiele



Blackmagic Design „Davinci Resolve“

- Datenflußgraph (z.B.) für Color-Grading Operationen
- Flexible Gestaltung komplexer Bildverarbeitung
  - EIS-Vorlesungsvideos (hier nur minimales Colorgrading :-))

Beispiele

Datenflussgraphen

- „DirectShow“ Graphen (Microsoft DirectX Framework)
- Plexus Framework (Tobias Ritschel, UCL/MPI Informatik)
- Amira Visualisierungssoftware
  (ZIB Berlin / Thermo Fischer Scientific)
- Davinci Resolve: Fusion-Compositor, Color-Grading
  (Blackmagic Design)
- TensorFlow (Deep Learning Framwork von Google)

“OOP”, “Functional”, und das “Expression Problem”

Zwei Varianten

Objekt-orientierte Variante

- Oberklasse mit festen Methoden (z.B. „draw()“)
- Neue Datentypen durch Ableiten
  - Möglich, ohne Oberklasse zu ändern/neu zu übersetzen
- Methoden erweitern ist aufwendig
  - Nicht via Plug-Ins: Neuübersetzung notwendig

Funktionale Variante

- Fester Satz an Datentypen (z.B. Primitive)
- Neue Operationen durch hinzufügen von Funktionen
  - „Funktionsobjekte“ in einer OOP-Sprache
- Datentypen erweitern ist aufwendig (auch: Neuübersetzung)

„Erweiterbarkeit“

Wan ist Code „leicht erweiterbar“

- Wir nehmen an, dass wir nicht allen Source Code haben
- Wir wollen Module nachladen und integrieren
  - Neue Klassen bereitstellen
  - Neue Funktionen bereitstellen
- Basis-Code soll nicht neu übersetzt / geändert werden

Ohne Plug-Ins?

- Kein „hartes“ Problem, trotzdem:
  - Leichter wartbar, wenn keine/kaum Änderungen nötig

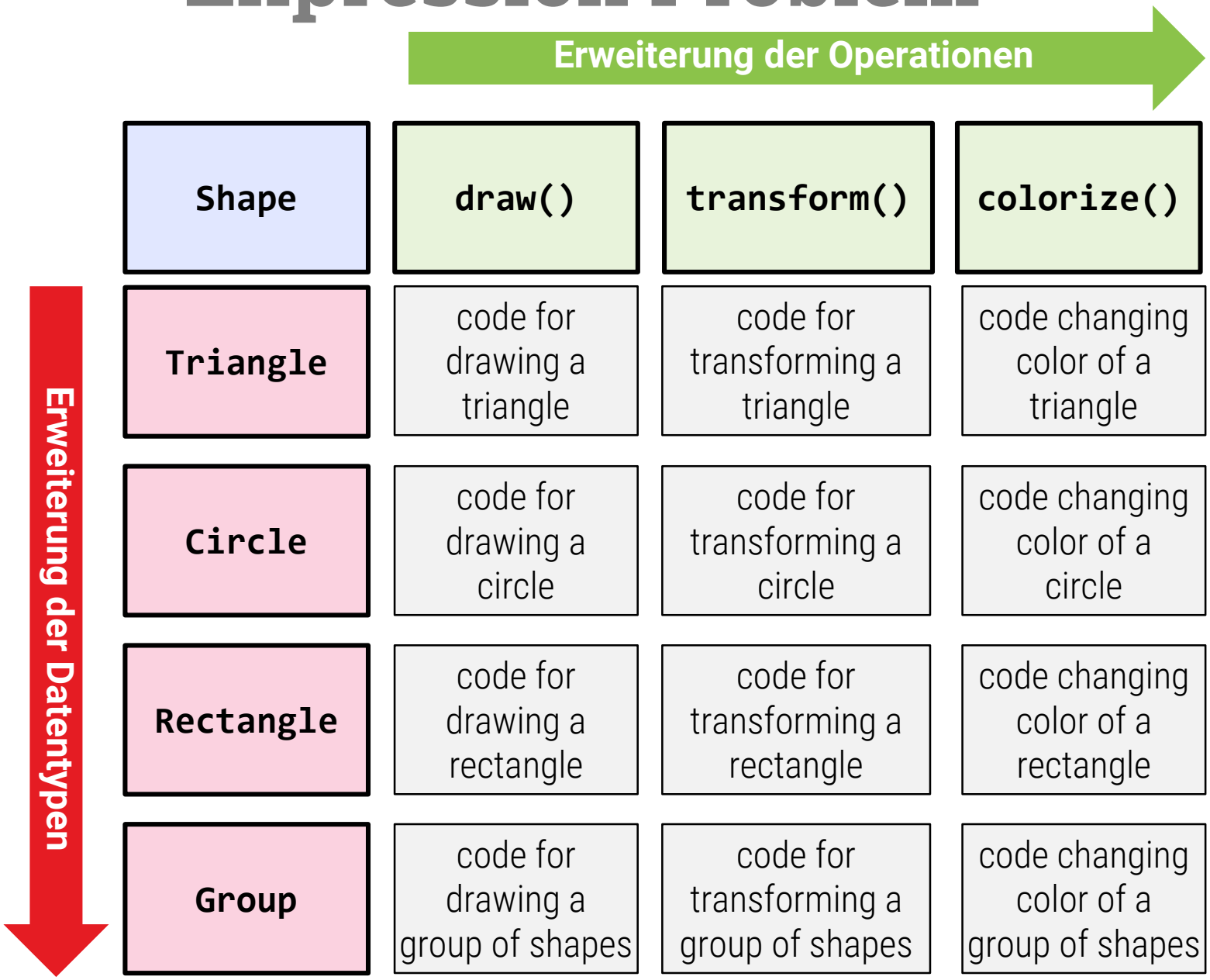
Funktional

```
Shape: TypeAlias = Circle | Star | Rectangle

def draw_circle(c: Circle, painter: QPainter) -> None:
    …
def draw_rectangle(r: Rectangle, painter: QPainter) -> None:
    …
def draw_star(r: Star, painter: QPainter) -> None:
    …

def render_scene(s: Scene, painter: QPainter) -> None:
    for shape in s.all_shapes:
        match shape:
            case Circle():
                draw_circle(shape, painter)
            case Rectangle():
                draw_rectangle(shape, painter)
            case Star():
                draw_star(shape, painter)
            case _:
                assert False # I do not like this!
```

Expression Problem



Leicht in funktionalem Design, schwerer in OOP

Leicht in OOP-Design, schwerer in funktionalem

Shape	draw()	transform()	colorize()
Triangle	code for drawing a triangle	code for transforming a triangle	code changing color of a triangle
Circle	code for drawing a circle	code for transforming a circle	code changing color of a circle
Rectangle	code for drawing a rectangle	code for transforming a rectangle	code changing color of a rectangle
Group	code for drawing a group of shapes	code for transforming a group of shapes	code changing color of a group of shapes

OOP

Oberklasse ändern, alle Nachfahren anpassen, alles neu kompilieren, C++-Binär-Bibliotheken nicht mehr kompatibel, Source-Code der Ober-Kl. nötig

Klasse ableiten, implementieren

`Shape`	`draw()`	`transform()`	`colorize()`
`Triangle`	code for drawing a triangle	code for transforming a triangle	code changing color of a triangle
`Circle`	code for drawing a circle	code for transforming a circle	code changing color of a circle
`Rectangle`	code for drawing a rectangle	code for transforming a rectangle	code changing color of a rectangle
`Group`	code for drawing a group of shapes	code for transforming a group of shapes	code changing color of a group of shapes

FP

Neue Funktion schreiben, alle Typen berücksichtigen

alle Funktionen durchgehen und Implementation hinzufügen
Source-Code nötig, nicht mehr kompatibel

Shape	draw()	transform()	colorize()
Triangle	code for drawing a triangle	code for transforming a triangle	code changing color of a triangle
Circle	code for drawing a circle	code for transforming a circle	code changing color of a circle
Rectangle	code for drawing a rectangle	code for transforming a rectangle	code changing color of a rectangle
Group	code for drawing a group of shapes	code for transforming a group of shapes	code changing color of a group of shapes

Lösungen für das Expression-Problem

Ad-Hoc Lösungen: z.B.

- „Visitor“-Pattern in OOP (Wechsel zu „FP-Situation“)
  - Gleiche Nachteile, aber umständlicher
  - Nützlich zur Kombination von Paradigmen
- „Nachbau“ von VMTs in FP
  - Etabliert z.B. Closures für single-Method Objekte

Visitor Pattern

```
class ShapeVisitor:
   def visitTriangle(t: Triangle) -> None: pass
   def visitRectangle(r: Rectangle) -> None: pass
   def visitCircle(c: Circle) -> None: pass
   def visitGroup(g: Group) -> None: pass
};

class DrawShape (ShapeVisitor): ...
class TransformShape(ShapeVisitor): ...
// ...extend as you like... (but adding types now becomes inflexible)

def visitListOfShapes(sl: list[Shape], v: ShapeVisitor) -> None:
  for s in sl:
  match s:
     case Triangle(): v.visitTriangle(s)
     case Circle(): v.visitCircle(s)
     case ... // all cases
     case _: print("too bad – we cannot extend types easily anymore now")
```

Lösungen für das Expression-Problem

Beispiele für systematische Lösungen

- Offene Klassen
  - Methoden nicht Teil des Typs (z.B. in Closure-“Multi“)
  - Auch in Julia (dyn. Dispatch, aber keine Klassen)
- Patching
  - Methoden Teil eines *dynamisch änderbaren* Types
  - Methoden dynamisch hinzufügen in Python („Monkey-Patching“)
  - Extension-Methods in Scala, C#

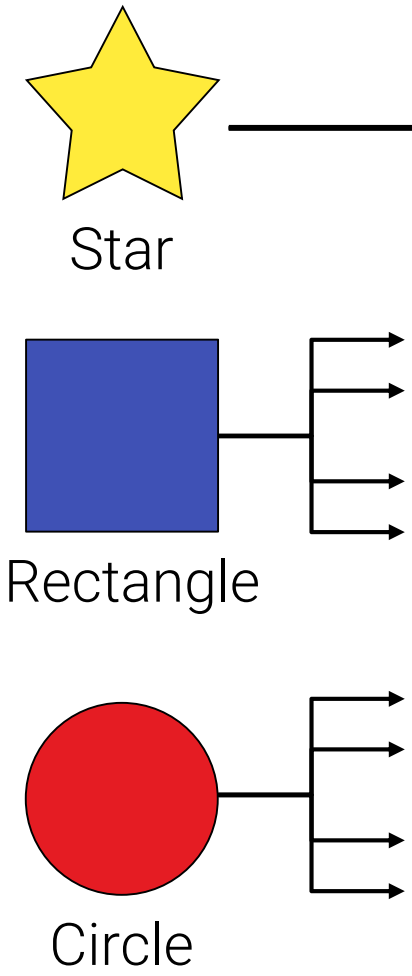
Höhere Komplexität: In der Praxis seltener

Verwandtes Problem

Beispiel

- Textsatzsystem mit strukturellen Objekten
  - `TextObjekt`: Überschriften, Gleichungen, Abbildungen, Fließtext etc.
    (ähnlich Latex, Word, o.ä.)
- Anforderung: Übersetzung in verschiedene Formate
  - `Format`: HTML, SVG, PDF, LaTex Code
- Kernelement
  `translate(input: TextObject, output: Format)`
  - Was zu tun ist, hängt vom Typ von Input und Output ab
  - Lösung: „**Multiple Dispatch**“
    - Auswahl von translate-implementation nach beiden Typen

Konvertierung von Zeichnungen



SVG

```
<svg
  <path d="M 80.271615,111.42732 59.421474,100.02115
38.205078,110.73066 42.609945,
  87.376298 25.868365,70.507728 49.440864,67.480111
60.310395,46.345246 70.474133,67.82844
  93.933453,71.634946 76.642489,87.939907 Z" style="stroke-
width:0.25;fill:#ffff00" />
</svg>
```

Qt/Screen

```
QPainter.drawPolygon(…)
```

OpenGL/Screen

```
glBegin(GL_POLYGON); glVertex2f(…); … glEnd(…);
```

PDF

```
%PDF-1.5

%% Original object ID: 1 0
3 0 obj
<<
 /Count 1
 /Kids [
  4 0 R
 ]
 /Type /Pages
>>
endobj

%% Page 1
%% Original object ID: 2 0
4 0 obj
<<
 /Contents 5 0 R
 /Group <<
  /CS /DeviceRGB
  /I true
  /S /Transparency
  /Type /Group
 >>
 /MediaBox [
  0
  0
  595.275591
  841.889764
 ]
 /Parent 3 0 R
 /Resources 7 0 R
 /StructParents 0
 /Type /Page
>>
endobj

%% Contents for page 1
%% Original object ID: 4 0
5 0 obj
<<
 /Length 6 0 R
>>
stream
1 0 0 -1 0 841.889764 cm
q
1 1 0 rg /a0 gs
227.543 315.855 m 168.438 283.523 l
108.297 313.883 l 120.785 247.68 l
73.328 199.863 l 140.148 191.281 l 170.957
131.371 l 199.77 192.27 l 266.27
 203.059 l 217.254 249.277 l h
227.543 315.855 m f
Q
endstream
endobj

6 0 obj
244
endobj

%% Original object ID: 3 0
7 0 obj
<<
 /ExtGState <<
  /a0 <<
   /CA 1
   /ca 1
  >>
 >>
>>
endobj

xref
0 8
0000000000 65535 f
0000000053 00000 n
0000000135 00000 n
0000000335 00000 n
0000000444 00000 n
0000000748 00000 n
0000001047 00000 n
0000001094 00000 n
trailer <<
 /Info 2 0 R
 /Root 1 0 R
 /Size 8
 /ID
[<98c5f2964a8f62228395ab4944b2b28d><98
c5f2964a8f62228395ab4944b2b28d>]
>>
startxref
1179
%%EOF
```

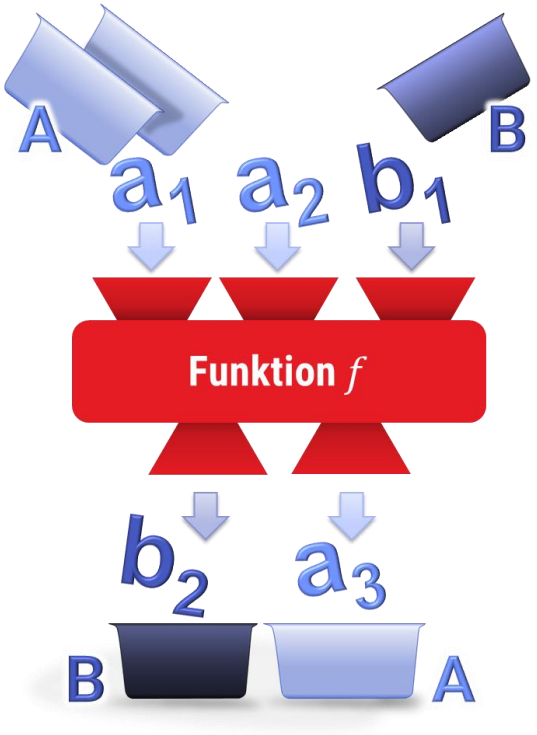
Multiple Dispatch

Multiple Dispatch

- Logische Erweiterung von Standard-OO-Polymorphie
  - Auswahl der „spezifischsten“ Methode nach Typen der Parameter
- Kernproblem: Prioritäten bei mehreren Methoden, die gleich gut „passen“
  - Auflösen z.B. die Vorrang nach Parameterreihenfolge
- Verfügbar in „extravaganteren“ Sprachen, z.B.
  - CLOS (Common-LISP Object System)
  - Python (Nachgerüstet via Bibliothek[en] mit Dekorator)
  - JULIA (kein klassisches OO, aber MD überall)

Weitere

Funktionale Ideen & Entwurfsmuster



Techniken speziell für FP

Bereits gesehen (mit Trade-Offs)

- Algebraic Data Types + Pattern Matching
- Data Flow Architectures (Datenflussgraphen)

Weitere Ideen (non-exhaustive)

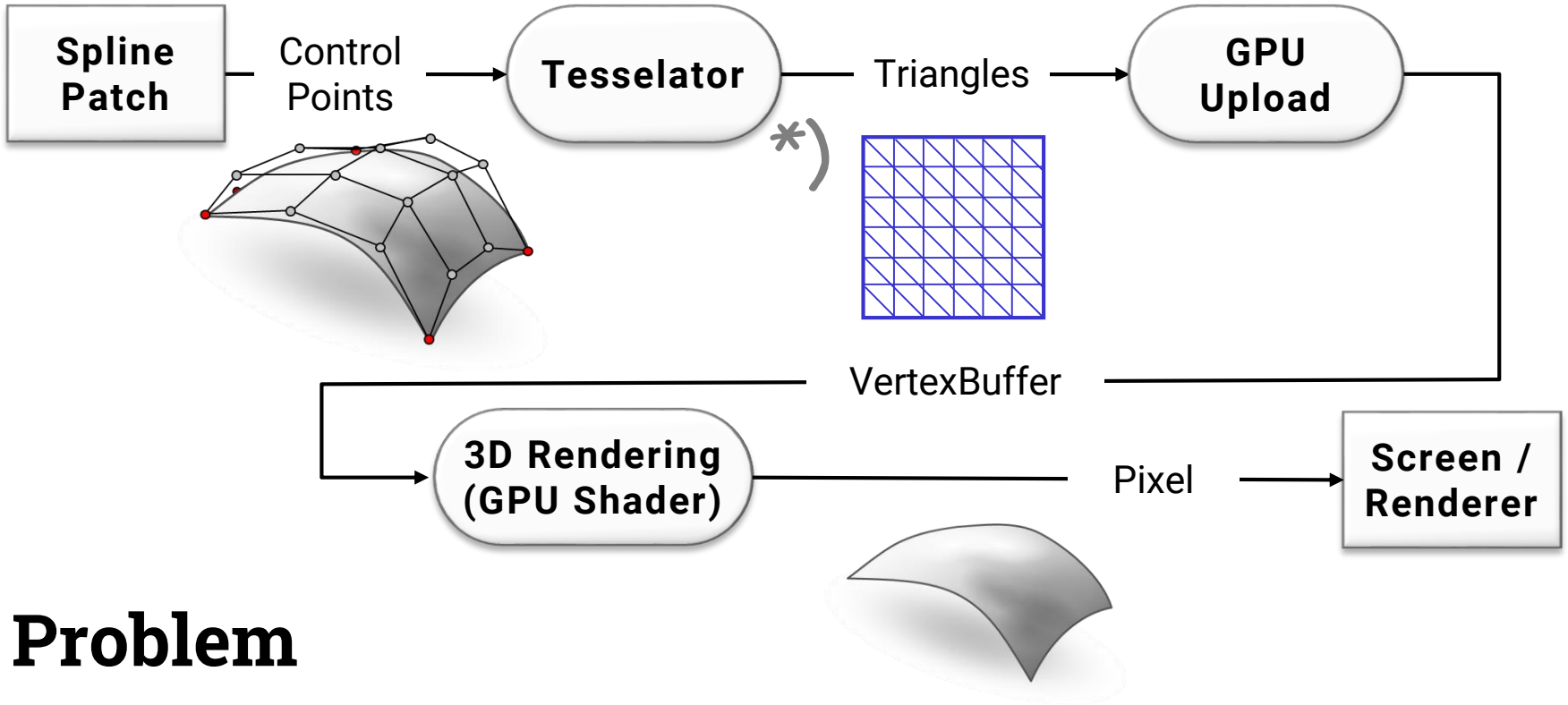
- Caching, z.B. für DFGs
- Unbenannte Funktionen („Lambda Expressions“)
- Closures (Variablenbindung an Funktionskontext)
- Partial Application & Currying
- Lazy Evaluation
- Continuations

Techniken speziell für FP

Weitere Ideen (non-exhaustive)

- Caching, z.B. für DFGs
- Unbenannte Funktionen („Lambda Expressions“)
- Closures (Variablenbindung an Funktionskontext)
- Partial Application & Currying
- Lazy Evaluation
- Continuations

Caching für DFGs



Problem

- Datenflussgraphen berechnen alles „on Demand“ neu
  - Je nach Problem kann das sehr teuer sein
- Beispiel hier: 3D Game Engine
  - Tesselierung (CPU) viel langsamer als Rendering (GPU)

…Tesselierung…

*) Daher gehen wir „einfacher“ vor: Wir teilen einfach die Objekte in kleine Vierecke oder Dreiecke auf, die wir dann einfach eckpunktweise verformen: Wenn eine Form aus vielen kleinen Fragmenten besteht, sieht man gar nicht mehr, dass der Rand nicht glatt ist. Hier ein Beispiel mit dem Viereck:

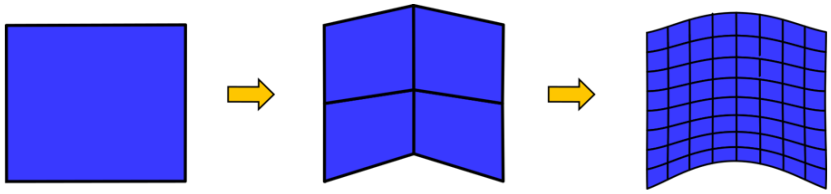
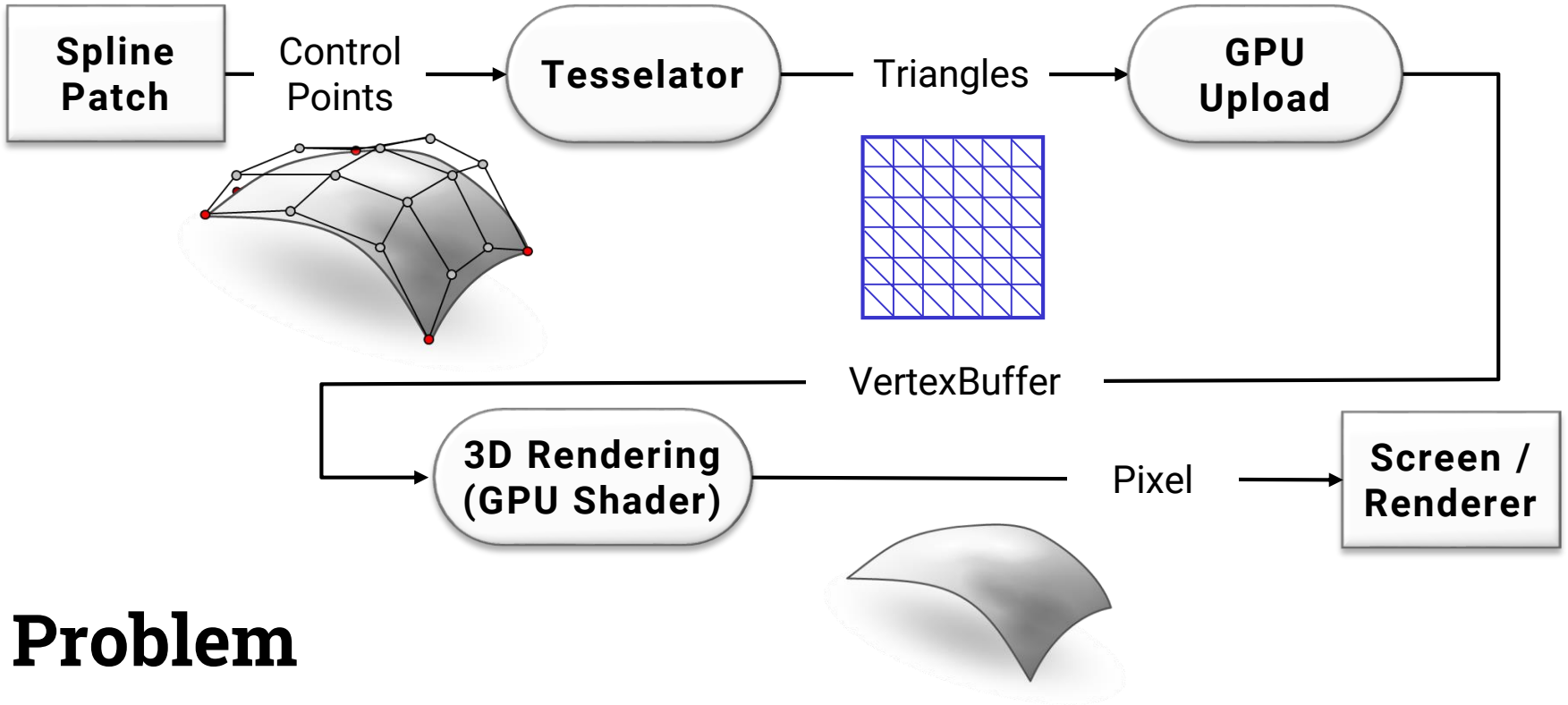


Abbildung 2: Wenn man die Formen in viele kleine Fragmente zerteilt, fällt gar nicht auf, dass deren Kanten eigentlich gerade sind (alle drei Formen bestehen nur als Vierecken mit geraden Seiten)

Die Implementation einer solchen Zerteilung kann etwas aufwendig sein; am einfachsten ist es, wenn man alle Formen bereits in Dreiecke zerlegt (einfacher als Vierecke), und diese dann durch mehrere, wiederholte „1:4“-Splits (Vierfachaufteilungen) verfeinert.

Abbildung 3: Man kann so ziemlich alle Formen leicht durch Dreiecke annähern (siehe rechts), und Dreiecke kann man dann leicht durch 1:4-Splits verfeinern.

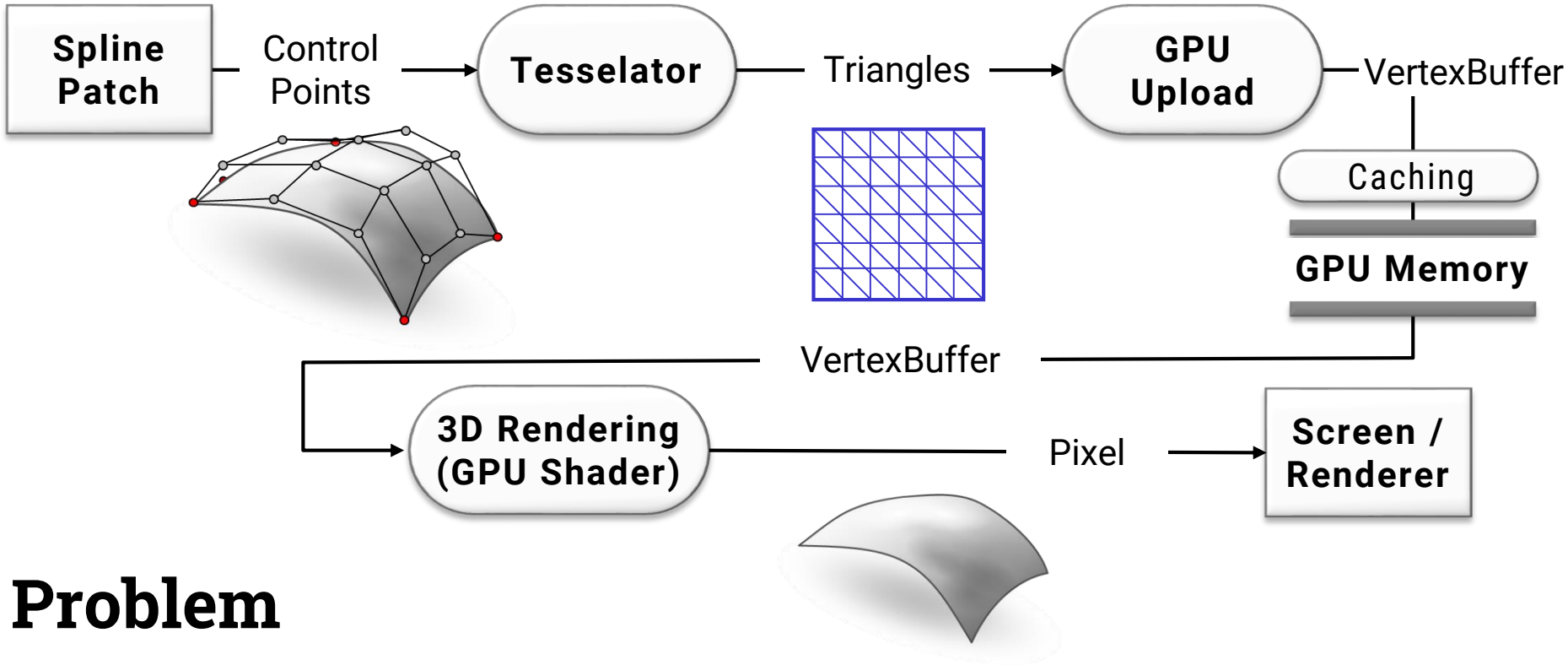
Caching für DFGs



Problem

- Datenflussgraphen berechnen alles „on Demand“ neu
  - Je nach Problem kann das sehr teuer sein
- Beispiel hier: 3D Game Engine
  - Tesselierung (CPU) viel langsamer als Rendering (GPU)

Caching für DFGs



Problem

- Caching: Zwischenergebniss speichern, solange sich nichts ändert
  - Speicher knapp? z.B. LRU Scheduling für Freigabe
- Allgemein ein nützliches Muster („Vorberechnung“)

Techniken speziell für FP

Weitere Ideen (non-exhaustive)

- Caching, z.B. für DFGs
- Unbenannte Funktionen („Lambda Expressions“)
- Closures (Variablenbindung an Funktionskontext)
- Partial Application & Currying
- Lazy Evaluation
- Continuations

Unbenannte Funktionen (λ-Expr.)

Feature / Problem in funktionalen Sprachen

- Konfiguration von Code via Funktionen
- Viele Funktionen müssen definiert werden
  - „Boiler-Plate-Code“ vermeiden

„Lambda“-Expressions

„lambda-Ausdruck“

- Definition einer unbenannten Funktion
- Python: `f = lambda x: float, y: float : x * y`
- Scala: `def f = (x: Float, y: Float) => x * y`

Nichts fundamentales, aber sehr praktisch

Techniken speziell für FP

Weitere Ideen (non-exhaustive)

- Caching, z.B. für DFGs
- Unbenannte Funktionen („Lambda Expressions“)
- Closures (Variablenbindung an Funktionskontext)
- Partial Application & Currying
- Lazy Evaluation
- Continuations

Closures

„Lambda“-Expressions again

- Python:

```
z: float = 42;
f = lambda x: float, y: float : x * y * z
```

- Scala:

```
val z: Float = 42;
def f = (x: Float, y: Float) => x * y * z;
```

Was nun?

- `z` ist kein offizieller Parameter der Funktion
- `z` wird mit in Funkt‘referenz `f` „eingepackt“ (*closure*)

Objekte mit einer Methode

Closure

Daten:

```
z: float = 42.0
```

Funktion:

```
f: Callable[[float,float], float] = return x * y * z
```

Speicher Daten für Funktionsaufruf

- Keine Parameter, entnommen aus Kontext

Zitat aus dem Web (sinngemäß)

- „Objekte sind Daten mit Funktionen (Methoden) dran, Closures sind Funktionen mit Daten dran“ *)

*) engl. orig. Zitat: http://mrevelle.blogspot.com/2006/10/closure-on-closures.html

Feinheiten

In Scala

- Bindung von **`val`** und **`var`** möglich
  - Man sieht hier, warum immutable sauberer ist :-)
- Bei **`var`** Referenzsemantik
  - Objekt kann geändert werden
  - Änderungen bei nächstem Aufruf sichtbar
- Bei **`val`**: Problem existiert nicht

Feinheiten

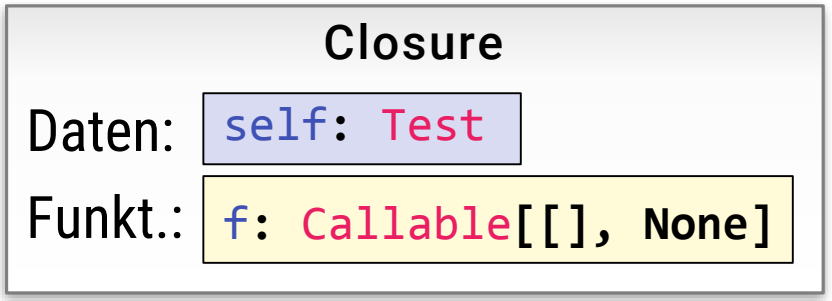
Mutability für Closures in Python

- Referenzsemantik, Änderungen an Objekt möglich
- Immutable Variablen (z.B. `int`) via „**`nonlocal`**“-Deklaration änderbar
  - Erlaubt, Variablen in closure *neu zu binden*

Weiteres, wichtiges Feature: Methodenzeiger

```
class Test:
    def method(): pass

obj: Test = Test()
f: Callable[[], None] = obj.method
```



`self` im closure!

Und die Anderen?

C++ kann das (inzwischen) auch

- Closures deklarieren bei Lambda-Ausdrücken:
  - Alles geht (Werte, Referenzen, Zeiger; const oder nicht)
  - Alles ohne GC, man muss vorsichtig sein
- Closure Objekte:
  - Modul: `#include <functional>`
  - Danach `std::functional<…>` als Zeiger auf Funktion (mit allen nötigen Daten)
  - Sehr praktisch!
    - Empfohlen statt „raw function pointers“

Techniken speziell für FP

Weitere Ideen (non-exhaustive)

- Caching, z.B. für DFGs
- Unbenannte Funktionen („Lambda Expressions“)
- Closures (Variablenbindung an Funktionskontext)
- Partial Application & Currying
- Lazy Evaluation
- Continuations

Currying & Partial Application

Partial Application

- Einige Argumente ausfüllen
  - Fast OOP: „Felder im Closure setzen“
- Scala Partial Application

```
def f = (x: Float, y: Float) => x * y;
def g = (x: Float) => f(x, 23.0);
println(g(2) - 4); // Ausgabe: 42.0 (=2*23-4)
```

Currying & Partial Application

Currying *(Name nach Haskell Brooks Curry)*

- Funktion mit mehreren Parametern
  → Funktion,
  die Funktion zurück gibt,
  die Funktion zurück gibt,
  die Funktion zurück gibt, ….
- Scala z.B. via

```
def f = (x: Float) => ((y: Float) =>
        ((z: Float) => x * y * z;))
def g = f(2)  // g = Funktion (Float) => (Float) => (Float)
def h = g(3)  // h = Funktion (Float) => (Float)
val r = h(4)  // r = 24
```

Currying vs. Partial Application

Partial Application

- Vorteil: Beliebige Parameter „ausfüllen“
- Vorteil: Reihenfolge egal

Currying

- Feste Reihenfolge zum „Ausfüllen“
- Vorteil: Funktion in Schritte aufteilen
  - Kann Modularisierung verbessern
  - Nicht bei `x*y*z`, obviously
- (Auch für Transformationen in Programmanalyse)

Techniken speziell für FP

Weitere Ideen (non-exhaustive)

- Caching, z.B. für DFGs
- Unbenannte Funktionen („Lambda Expressions“)
- Closures (Variablenbindung an Funktionskontext)
- Partial Application & Currying
- Lazy Evaluation
- Continuations

Lazy Evaluation

„On Demand“ Compuation

- Funktionsaufruf oder Auswertung erst, wenn Ergebnis gebraucht wird
- Scala:

```
lazy val onDemand = loadData("pic.png");
showPic(onDemand); // erst hier wird Datei geladen
```

Oft genutztes Muster

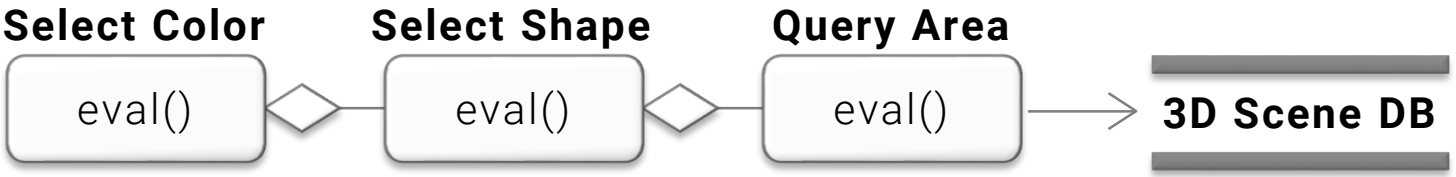
- Berechnung / Aktion erst bei Bedarf
  - Beispiele: Paging, Delayed load für Hypertext mit Bildern
  - Auch in imperativen Sprachen beliebt (z.B. Java `eval()` Interface)

Scala kann noch mehr…

Lazy Evaluation

- Unendliche Datenstrukturen / Datenströme
  - Berechnet oder geladen (vom Netz, von Disk)
- Beispiel: Python 3 „`for i in range(…)`“
  - `range`-Funktion erzeugt Sequenzobjekt
  - Werte werden „on-demand“ berechnet
  - Kein Speicherverbrauch (anders als in Python 2)
- Kombination mit Datenflussgraphen
  - Beispiel (eigene Erfahrung):
    - Iterator sucht (geometrische) Objekte aus Datenbank
    - Weitere Verarbeitung als Kette von lazy-eval-Funktionen

Kernidee?



Also was ist die Kernidee?

- Man definiert Operationen als Funktionen
  - „Funktionsobjekte“ in OO-Sprachen wie Python
  - Zusammenbau möglich, ohne Auswertung zu starten
  - Gesonderter Mechanismus, um Auswertung auszulösen
- Auswertung
  - Oft bei Benutzung des Wertes automatisch
    - z.B. C++ Feldzugriff-Operator „`.`“ oder „`->`“ überladen
    - „Smart Pointers“
  - Oft als Kaskade (Auswertung triggered Auswertung triggert…)

Scala kann noch mehr…

Scala kann noch mehr

- z.B. „By-Name“ Parameter:
  - Code-Blöcke als Parameter übergeben
  - Aber noch nicht auswerten
  - Erst bei Funktionsaufruf mit Variablenbindung an Kontext
  - Beyond this lecture…

Techniken speziell für FP

Weitere Ideen (non-exhaustive)

- Caching, z.B. für DFGs
- Unbenannte Funktionen („Lambda Expressions“)
- Closures (Variablenbindung an Funktionskontext)
- Partial Application & Currying
- Lazy Evaluation
- Continuations

Abgefahrenes Muster :-)

„Rein“ funktionale Sprachen

- Rekursion statt Schleifen
- „Tail-Call-Recursion“ (Rekursiver Aufruf am Ende) wird optimiert, so dass kein Stack-Speicher nötig wird
  - Ohne das wäre die Sprache unbrauchbar
  - Keine längern „Schleifen“ möglich

Not a bug, a feature

- Tail calls kosten wirklich nichts
- Fortsetzung des Programms als Parameter übergeben

Continuation Passing

Sieht ungefähr so aus

Python (kein TCO)

```
def do_something(x: int, next: Callable[[], None]):
    print(x*x)
    next()


do_something(4, lambda: print("Hooray, done!"))
```

Ergebnis:

```
16
Hooray, done!
```

Anwendungen

Einfache Anwendungen

- Callbacks (für ereignisoriente Programmierung)
  - z.B. bei GUIs
- „Asynchrone Funktionsaufrufe“
  - Funktionsaufruf wird an äußere Eventloop delegiert
  - Wartet z.B. auf Netzwerkeingabe oder anderen I/O
  - Aufruf, wenn fertig

„Volles Programm“

- Unbeschränkte Ketten: Nur mit TCO (FP Laufzeitsystem)
- Anwendungen z.B. als Zwischencode im Compilerbau

Funktionale Muster

Zusammenfassung

Wichtige Muster

- Lambdas: Tipparbeit sparen
- Closures + Partial Application
  - In etwa: OOP für FP
    - Felder ähnlich Parameter
  - Anders herum: Nützlich in imperativen Sprachen
    - z.B. Eventmodellierung in Python (Qt/PySide)
- Lazy Evaluation
  - Häufiges Entwurfsmuster in imperativen Systemen
    - Gezielt einsetzen, unerwartete Seiteneffekte
  - Ganze FP-Sprachen komplett „lazy“ (z.B. Haskell)
    - Ablaufreihenfolge kann unklar werden
    - Pure FP: Seiteneffekte und IO muss man in „Monaden“ kapseln

Zusammenfassung

Wichtige Muster

- Continuations
  - Häufig in OO+FP z.B. für Eventhandler oder andere „asynchrone Muster“
    - Kontrollfluss von außen gesteuert
  - Komplete Steuerung des Kontrollflusses: eher selten

Fazit:

- Konzepte aus verschiedenen Welten lernen
  - Einfacher auszudrücken in passender Sprache
  - Trotzdem nützlich in anderen Paradigmen