Programmierparadigmen

by

Dr. Günter Kolousek

Programmierparadigmen

- Art der Programmierung
- grobe Einteilung
  - imperative Programmierung: Programm legt die Abarbeitung von Operationen fest
    - prozedurale Programmierung
    - modulare Programmierung
    - objektorientierte Programmierung
    - generische Programmierung
  - deklarative Programmierung: Programm legt *nicht* die Abarbeitung von Operationen fest
    - funktionale Programmierung
    - logische Programmierung

Imperative Programmierung

- Programm besteht aus einer Folge von Befehlen
  - z.B. Addieren, I/O-Befehle, Sprungbefehle
- Befehle werden nacheinander abgearbeitet
  - und verändern dabei Daten (Speicherzellen)
- Programmiersprachen
  - frühe Assemblersprachen

Prozedurale Programmierung

- Prozeduren (auch: Unterprogramm, Routine, Subroutine): Zusammenfassung von Befehlen
  - auch Funktionen (nicht im mathematischen Sinne!), d.h. mit Rückgabewert
- Record (auch Struktur): Zusammenfassung von Daten
  - werden an Prozeduren weitergegeben
- Sinn
  - logische Gliederung
  - Wiederverwendung von Code
- Programmiersprachen
  - C, Pascal, Fortran, COBOL, Basic,…

Modulare Programmierung

- Aufteilung der Gesamtfunktionalität (an Prozeduren und Datenstrukturen) in Module
- Ein Modul
  - fasst Prozeduren und Daten zu einer logischen Einheit zusammen
  - bietet eine Schnittstelle an
- Sinn
  - weitergehende Strukturierung des Systems
- Programmiersprachen
  - Oberon (und Oberon-2, Oberon-07), Modula-2 (und Modula-3), Ada

Objektorientierte Programmierung

- Zusammenfassung von Befehlen und Daten zu Klasse
  - Vererbung und Polymorphie
  - objektbasierte Programmierung: keine Vererbung
    - z.B. Javascript
- Sinn
  - Zusammenbringen was zusammen gehört
- Programmiersprachen
  - Python, Java, C#, C++, Ruby,…
- siehe Foliensatz *Objektorientierung*

Generische Programmierung

- Definition einer Funktion (oder auch Klasse samt Methoden) enthält Typvariablen
  - aber: unabhängig von Klassen oder Vererbung!
- Ziel: Verwendung einer Funktion mit verschiedenen Typen
- Beispiel: Entwicklen einer Funktion `add(x, y)`
  - Lösung in C++ mit mehreren überladenen Funktionen:

```
int add(int x, int y) {
    return x + y;
}
double add(double x, double y) {
    return x + y;
}
// ...
```

  - $\rightarrow$ immer der ”gleiche” Code!

Generische Programmierung – 2

- Beispiel:
  - Lösung mit einem (Funktions)Template:

```
// Voraussetzung: Operator + ist überladen
// anderenfalls Compilerfehler!
template <typename T>
T add(T x, T y) { return x + y; }

int main() {
    cout << add(1, 2) << endl;
    cout << add(1.0, 2.0) << endl;
    // cout << add("abc", "def") << endl;  // -> error
    cout << add(string{"abc"}, string{"def"}) << endl;
}
```

- $\rightarrow$ Reduzierung der Implementierung

(Template) Meta-Programming

```cpp
#include <iostream>

using namespace std;
using ull = unsigned long long;

template <ull n>
struct Factorial {
    static constexpr ull value{n * Factorial<n - 1>::value};
};

template <>
struct Factorial<0> {
    static constexpr ull value{1};
};

int main() {
    Factorial<0> f0;
    cout << f0.value << endl;  // -> 1
    cout << Factorial<1>::value << endl;  // -> 1
    cout << Factorial<2>::value << endl;  // -> 2
    cout << Factorial<64>::value << endl;  // -> 9223372036854775808
}
```

Funktionale Programmierung

- basiert nicht Berechnung eines inneren Zustandes eines Berechnungsprozesses
  - keine Nebeneffekte (side effects) möglich
- Funktionen im mathematischen Sinn
  - gleiche Eingabe $\rightarrow$ gleiche Rückgabe
  - keine Nebeneffekte
- Funktionale Programme
  - keine Folge von Anweisungen sondern ineinander verschachtelte Funktionsaufrufe
  - Higher-order functions
  - partial application
  - Closures
- Programmiersprachen
  - Haskell, Lisp, Erlang,…
  - aber auch Erweiterungen in ”normalen” Programmiersprachen: Python, Java, C#, C++,…

Funktionale Programmierung – 2

- Higher-order functions
  - Mathematik, Informatik
  - zumindest eines der beiden Kriterien
    - hat eine oder mehrere Funktionen als Parameter
    - liefert eine Funktion als Ergebnis zurück
  - Beispiel

```
def f(x):
    return 2 * x + 1
def g(x):
    return -2 * x - 1
def add_functions(u, v, x):
    return u(x) + v(x)
print(add_functions(f, g, 1))
```

  - Implementierung entweder als
    - first-class object
    - Adresse der Funktion

Funktionale Programmierung – 3

- Partial application
  - auch *currying* genannt (nach Haskell Curry)
    - Teilweise Anwendung einer Funktion bzgl. der Parameter
    - ergibt Funktion mit einer geringeren Anzahl an Parameter
  - Beispiel 1

```
print(int('1111', 2))  # -> 15
from functools import partial
base2 = partial(int, base=2)
print(base2('1111'))  # -> 15
```

  - Beispiel 2

```
def make_adder(n):
    return lambda k: n + k

add = make_adder(2)
add(3)  # -> 5
```

Funktionale Programmierung – 4

- Closures (Funktionsabschluss)
  - Funktion, die Kontext beim Aufruf speichert
    - Kontext sind die nicht lokalen Variablen
    - auch wenn der Kontext nicht mehr exisitiert
    - Closures ”konservieren” also ihren Kontext

```
def create_incrementer(start, inc):
    cnt = start
    def increment(inc=inc):  # closure!
        nonlocal cnt
        cnt += inc
        return cnt
    return increment

inc12 = create_incrementer(1, 2)
print(inc12(), inc12(), inc12(1)) # -> 3 5 6
```

Funktionale Programmierung – 5

- Berechnung der Faktoriellen
  - nicht funktional

```
def factorial(n):
    res = 1
    while n >= 1:
        res *= n
        n -= 1
    return res
```

  - funktional

```
def factorial(n):
    return 1 if n <= 1 else n * factorial(n - 1)
```

  - funktional – 2 (in Python am schnellsten!)

```
from functools import reduce
from operator import mul
def factorial(n):
    return reduce(mul, range(1, n + 1), 1)
```

Logische Programmierung

- Fakten und Regeln vorgegeben
  - Regelinterpreter leitet für eine Fragestellung (query) eine Antwort ab
- basiert auf mathematischer Logik
- Programmiersprachen
  - Prolog
    - Teilmenge der Prädikatenlogik erster Ordnung
  - SQL

Logische Programmierung – 2

Prolog:

```
mann(adam).
mann(tobias).
mann(frank).
frau(eva).
frau(daniela).
frau(ulrike).
vater(adam,tobias).
vater(tobias,frank).
vater(tobias,ulrike).
mutter(eva,tobias).
mutter(daniela,frank).
mutter(daniela,ulrike).
```

Quelle: Wikipedia

Logische Programmierung – 3

Abfragen:

```
?- mann(tobias).
yes.
?- mann(heinrich).
no.
?- frau(X).  % Variable -> beginnen groß!
X=eva
X=daniela
X=ulrike
?- mann(heinrich).  % negativ -> keine Ableitung gefunden!
no
?- frau(heinrich).
no.
```

Logische Programmierung – 4

Regeln:

```
grossvater(X,Y) :-  % X Großvater von Y väterlicherseits
    vater(X,Z),     % , -> AND
    vater(Z,Y).     % ; -> OR
grossvater(X,Y) :-  % mütterlicherseits
    vater(X,Z),
    mutter(Z,Y).
```

Abfragen:

```
?- grossvater(adam,ulrike).
yes.
?- grossvater(X,frank).
X=adam
```

Programmiersprachen

- C: prozedural
  - Nachfolger von B (B von BCPL)
  - Systemprogrammierung, embedded systems
- Java: objektorientiert mit funktionalen Elementen
  - beeinflusst von: C++, Smalltalk, C#
  - Netzwerk- und Serverprogrammierung
- C#: objektorientiert mit funktionalen und deklarativen Elementen
  - beeinflusst von: Java, C++, Delphi (Object-Pascal)
  - Anwendungsentwicklung, Webentwicklung
- C++: generische Programmierung, objektorientiert mit funktionalen Elementen
  - beeinflusst von: C, Simula, Ada
  - Anwendungsentwicklung, Systemprogrammierung, embedded systems, HPC

Programmiersprachen – 2

- Erlang: funktional
  - beeinflusst von: Smalltalk, Prolog, Lisp
  - nebenläufige Anwendungen (ursprünglich für Telekommunikation), Serverprogrammierung
- Go: objektorientiert
  - beeinflusst von: Pascal, C, Smalltalk
  - nebenläufige Programmierung
- Haskel: rein funktional
  - beeinflusst viele Programmiersprachen…
- Lisp: funktional und prozedural
  - beeinflusst von Smalltalk
  - für KI, Emacs ;-)

Programmiersprachen – 3

- Python: objektorientiert, prozedural, funktionale Elemente
  - beeinflusst von: ABC, C und C++, Lisp, Haskell
  - Anwendungsprogrammierung, Webprogrammierung, wissenschaftliche Anwendungen
- Ruby: objektorientiert
  - beeinflusst von: Perl, Smalltalk, Eiffel
  - Webprogrammierung
- Rust: generische Programmierung
  - beeinflusst von: C++, Erlang, Haskell
  - Systemprogrammierung
- Smalltalk: rein objektorientiert
  - beeinflusst von: Simula, Lisp
  - Anwendungsprogrammierung
- weitere: Nim, Lua, D, Eiffel, Forth, Fortran,…