Programmiersprachen

Vorlesung 4 vom 08.11.21

Kontrollabstraktion

Christoph Lüth

Universität Bremen

Wintersemester 2021/22

Wo sind wir?

- Einführung
- Werte und Typen
- Anweisungen, Variablen und Zustand
- Kontrollabstraktion
- Datenabstraktion
- Fortgeschrittene Typsysteme
- Nebenläufigkeit
- Objektorientierung
- Skriptsprachen
- Beispielsprache II
- Ab Woche 11: Studentische Vorträge.

Abstraktion

- Definition Wikipedia:

> Das Wort Abstraktion[a] bezeichnet meist den [...] Denkprozess des erforderlichen Weglassens von Einzelheiten und des Überführens auf etwas Allgemeineres oder Einfacheres. Daneben gibt es spezifische sowie unspezifische Verwendungen des Begriffes in bestimmten Einzelwissenschaften und einzelnen Theorien, Thesen sowie Behauptungen.
>
> [a] lat. *abstractus* "abgezogen", Partizip Perfekt Passiv von abs-trahere "abziehen", "trennen"

- Weiter: "In der Mathematik [...] werden Abstrakta meist mit Äquivalenzklassen identifiziert."
- Im Lambda-Kalkül ist Abstraktion $\lambda x.\ t$ — die Einführung des Funktionsparameters

Abstraktion

- Kontrollabstraktion (heute):
  - Jenseits der direkten Auswertung
  - Prozeduren und Parameter
  - Sprünge
- Datenabstraktion (nächstes Mal):
  - Abstrakte Datentypen
  - Verkapselung und Objekte
  - Module
  - Packages

Prozeduren

Prozeduren und Funktionen

- **Prozeduren** sind benannte, parameterisierte **Blöcke**
  - Meist ohne Rückgabewert
- **Funktionen** sind Prozeduren mit Rückgabewert
  - **Reine** Funktionen (pure functions): referentiell transparent, ohne Seiteneffekt
  - Meist **mit** Seiteneffekten
- Viele Programmiersprachen unterscheiden das nicht
- Funktionsdefinition hat **(formale) Parameter**, beim Aufruf **Parameterwerte** (Argumente)

```
int f(x) { return x*10; }   // ‘x‘ ist formaler Parameter

... f(29+2) ... // ‘29+2‘ ist Parameterwert
```

Parameterübergabe (Parameter Passing)

- Aus **konzeptioneller** Sicht gibt es drei Arten von Parametern:
  - **Eingabeparameter** — erlaubt Kommunikation vom Aufrufer an die Funktion
  - **Ausgabeparameter** — erlaubt Kommunikation von der Funktion an den Aufrufer
  - **Ein/Ausgabeparameter** — erlaubt bidirektionale Kommunikation

Parameterübergabe (Parameter Passing)

- Aus **konzeptioneller** Sicht gibt es drei Arten von Parametern:
  - **Eingabeparameter** — erlaubt Kommunikation vom Aufrufer an die Funktion
  - **Ausgabeparameter** — erlaubt Kommunikation von der Funktion an den Aufrufer
  - **Ein/Ausgabeparameter** — erlaubt bidirektionale Kommunikation
- Beispiel (Ada; Eingabeparater `v`, `w`, Ausgabeparameter `sum`)

```
type Vector is array (1 .. n) of Float;

procedure add (v, w: in Vector; sum: out Vector) is
  begin for in 1 .. n loop
    sum(i) := v(i) + w(i);
  end loop
```

Parameterübergabe (Parameter Passing)

- Aus **konzeptioneller** Sicht gibt es drei Arten von Parametern:
  - **Eingabeparameter** — erlaubt Kommunikation vom Aufrufer an die Funktion
  - **Ausgabeparameter** — erlaubt Kommunikation von der Funktion an den Aufrufer
  - **Ein/Ausgabeparameter** — erlaubt bidirektionale Kommunikation
- Beispiel (Ada; Eingabeparater `v`, `w`, Ausgabeparameter `sum`)

```
type Vector is array (1 .. n) of Float;

procedure add (v, w: in Vector; sum: out Vector) is
  begin for in 1 .. n loop
    sum(i) := v(i) + w(i);
  end loop
```

- Aus **operationaler** Sicht gibt verschiedene Arten der **Parameterübergabe**

Call by Value

- **Parameterwert** ist **beliebiger** Ausdruck (R-Wert)
- **Funktionsaufruf**:
  - Parameter wird zu v ausgewertet
  - Formaler Parameter wird lokale Variable im Funktionsrumpf, mit v initialisert
  - Funktionsrumpf wird ausgeführt
- Für **Eingabeparameter**
- Klare Semantik (kein Effekt auf Aufrufer)
- Effizient für "kleine" v, ineffizient für große Datenstrukturen (Felder etc.)
- Wertet eventuell zu viel aus

Call by Reference (Call by Variable)

- Parameterwert muss ein L-Wert sein
- Funktionsaufruf:
  - Umgebung des Funktionsrumpfes wird erweitert
  - Formaler Parameter wird zu L-Wert aufgelöst (aliasing)
  - Funktionsrumpf wird ausgeführt
- Für **Ausgabeparameter** und **Ein/Ausgabeparameter**
- Funktion kann Parameterwert verändern
- Effizient aber fehleranfällig (wegen Aliasing)

```
void foo(reference int x)
{ x= x+1; }

char V[10];
i= 2;
V[2]= 5;
foo(V[i]);
// V[2] == 6
```

Call By Name

- Parameterwert ist beliebiger Ausdruck `e` (R-Wert)
- Aufruf von Funktion `f(x)` ist semantisch äquivalent zur Ausführung des Rumpfes, in dem alle `x` durch `e` ersetzt werden.
- Semantisch sauber, aber subtil
- Stammt von ALGOL-60, wird heute nur noch wenig benutzt

```
int x= 0;
int foo(name int y)
{
  int x= 2;
  return x+y;
}
...
int a= foo(x+1);
  // = {int x= 2; x+ x+ 1} ???
```

Jensen's Device

- Call-by-Name erlaubt **Metaprogrammierung** (Macros)
- Beispiel: Jensen's Device

```
int sum(name int exp; name int i; int fr; int to)
{
  int acc= 0;
  for (i= fr; i<= to; i++) acc= acc+ exp;
  return acc;
}

int x= ...;
int y= sum(2*x*x- 2*x+ 1, x, 1, 10)
```

- Berechnet

$$y = \sum_{x=1}^{10} 2x^x - 2x + 1$$

Variationen

- Call by constant:
  - Wenn Funktionsrumpf den formalen Parameter nicht modifiziert kann call-by-value durch call-by-reference implementiert werden.
- Call by need (Haskell):
  - Ähnlich call-by-name, Parameterwert wird **nur** ausgewertet, wenn er benutzt wird
- Call by value mit Zeigern (C, Java, Python)
  - Wenn Werte Zeiger (Referenzen) sind kann der Aufruf Seiteneffekte haben
  - Parameter vom Typ Pointer (C) oder `Object` (Java, Python) sind Ein/Ausgabe-Parameter

Parameterübergabe in C

C-Standard (C99, §6.5.2.2)

1. The expression that denotes the called function[77] shall have type pointer to function returning void or returning an object type other than an array type.
4. An argument may be an expression of any object type. In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.[a]

[a] A function may change the values of its parameters, but these changes cannot affect the values of the arguments. On the other hand, it is possible to pass a pointer to an object, and the function may change the value of the object pointed to. A parameter declared to have array or function type is adjusted to have a pointer type as described in 6.9.1.

Parameterübergabe und Auswertungsstrategie

- Auswertungsstrategien in funktionalen Sprachen:
  - Innermost-first
  - Outermost-first
- Innermost-first $\sim$ call-by-value, eager evaluation
- Outermost-first $\sim$ call-by-need, lazy evaluation
- Outermost-first: nicht-strikt

```
f 7 undefined  ⇝  14
```

Beispiel:

```
f x y = x+ x
```

Auswertung innermost:

```
    f (f 7 3) (f 5 9)
⇝   f (7+7) (5+ 5)
⇝   f 14 20
⇝   14+14  ⇝  28
```

Auswertung outermost:

```
    f (f 7 3) (f 5 9)
⇝   f 7 3+ f 7 3
⇝   (7+ 7)+ (7+ 7)
⇝   14+14  ⇝  28
```

Funktionen höherer Ordnung

Funktionen Höherer Ordnung

- Funktionen höherer Ordnung sind Funktionen $A \to B$ mit A oder B eine Funktion.
- Funktion als Argument, Beispiel (Python):

```
map(str, [1, 18, true, "foo"])
```

- Funktion als Resultat, Beispiel `3 <=` (vom Typ `Int-> Bool`) in (Haskell):

```
filter (3 ≤) [0,7,1,8,2,9,-2]
```

- Dabei hilfreich: "anonyme" Funktionen (Lambda-Ausdrücke), Beispiel (Python):

```
filter (lambda x: 3<= x, [0,7,1,8,2,9,-2])
```

- Komplikationen: Scoping
- Python und besonders Haskell unterstützen Funktionen höher Ordnung

Funktionen höherer Ordnung in C

- Auch C unterstützt Funktionen höherer Ordnung — durch Zeiger
- Beispiel:

```
typedef struct list_t {
     void          *elem;
     struct list_t *next;
     } list_t;
extern list_t *filter(int f(void *x), list_t *l);
extern list_t *map(void *f(void *x), list_t *l);
```

- Problem: Speicherverwaltung, Typsystem nicht expressiv genug
- Wird genutzt für Sprungtabellen, Signalhandler, Callbacks.

Exceptions

Ausnahmen (Exceptions)

- Motivation:
  - Fehlerbehandlung in geschachtelten Funktionen
  - Ohne Sprünge nur umständliche Fallunterscheidungen

Ausnahmen (Exceptions)

- Motivation:
  - Fehlerbehandlung in geschachtelten Funktionen
  - Ohne Sprünge nur umständliche Fallunterscheidungen
  - Nicht alle Fehlermöglichkeiten **können** im Vorfeld ausgeschlossen werden
  - Klassisches Beispiel: Dateizugriff

```
if os.path.exists(filenm):
  # someone deletes filenm
  fd= open(filenm) # FEHLER!
```

```
int main() {
  fd= open("data");
  ... P(fd); ...;
  close(fd);
  }

void P(int fd) { ... Q(fd); ... }

void Q(int fd) { ... R(fd); ... }

void R(int fd) {
  if ((x= read(fd, 1024))== -1)
    // Fehler!
  ...
  }
```

Ausnahmen

- Konzeptionell: Ausnahmen sind **Erweiterung des Definitionsbereichs**:

$$\texttt{f} : A \to B \texttt{ throws } C \;=\; \texttt{f} : A \to B + C$$

$$B[f(x)] \texttt{ catch } e \Rightarrow E \;=\; \begin{cases} B[b] & f(x) = b \\ E & f(x) = e \end{cases}$$

- Unterstützung von Ausnahmen durch eine Programmiersprache benötigt:
  - Ausnahmen **deklarieren** — meist eigener Typ (SML) oder Klasse (Java, Haskell)
  - Ausnahmen **auslösen** (`throw`, `raise`)
  - Ausnahmen **fangen** (`catch`)

Ausnahmen in Java

- Ausnahmen sind Objekte der Klassen `Throwable`, `Exception`
- Geworfene Ausnahmen müssen **deklariert** werden (`throws ...`)
  - Warum? Static Scoping (siehe Beispiel)
- Schema:

```
try
  block
catch (exception_type e)
  block
catch (exception_type e)
  block
finally
  block
```

Ausnahmen in Haskell

- Ausnahmen sind Instanzen der Typklasse `Exception` (Modul `Control.Exception`)
  - Situation in Haskell98 anders
- Werfen und fangen:

```
throw :: Exception e ⇒ e → a
catch :: Exception e ⇒ IO a → (e → IO a) → IO a
```

- Exceptions können überall geworfen werden, aber nur als Aktion (`IO`) gefangen werden.
  - Warum? Bricht referentielle Transparenz
- Durch Nicht-Striktheit (verzögerte Auswertung) werden Ausnahmen später geworfen als man denkt (siehe Beispiel)

Ausnahmen in C

- C hat **keine** Exceptions
- Alternativen:
  - `goto`
  - `setjmp` and `longjmp`
  - `switch`, siehe Duff's Device

Duff's Device:

```
void send(short *to, short *from, i
{
  int n = (count+ 7)/8;
  switch (count % 8) {
   case 0: do {
              *to = *from++;
   case 7:    *to = *from++;
   case 6:    *to = *from++;
   case 5:    *to = *from++;
   case 4:    *to = *from++;
   case 3:    *to = *from++;
   case 2:    *to = *from++;
   case 1:    *to = *from++;
           } while (--n > 0);
} }
```

Quelle: Wikipedia

Programmieren mit Ausnahmen

Ausnahmen sollten **unerwartete** und **seltene** Situationen modellieren.

1. "Ask forgiveness, not permission"
   - Bessere Robustheit
2. "Let it fail"

3. Nur Ausnahmen fangen, die auch behandelt werden
   - Unbehandelte Fehler werden nur schlimmer
4. Ausnahmen können auch der Effizienz dienen.

Zusammenfassung

- Abstraktion ist die Kunst des Weglassens unerheblicher Details
- Heute: Kontrollabstraktion
- Prozeduren — parametrisierte Blöcke
  - Parameter: In, Out, In/Out
  - Parameterübergabemechanismen: call-by-value, call-by-reference, call-by-name
- Ausnahmen: reglementierte Sprünge
  - In Java, Python, Haskell recht ähnlich
  - In C nicht vorhanden
  - Sollten **unerwartete** und **seltene** Situationen behandeln