Ablaufsteuerung

Markus Roggenbach

2. Dezember 2002

Aktuelle Vorbemerkung

M.Broy, J.Siedersleben:
Objektorientierte Programmierung und Softwareentwicklung,
Informatik Spektrum, Februar 2002.

Thesen:

OO ist das populärste Programmier-Paradigma

OO wird den heutigen Bedürfnissen nicht gerecht

S.Jähnichen, S.Herrmann:
Was, bitte, bedeutet Objektorientierung?
Informatik Spektrum, Juli (?) 2002.

These: OO-Konzepte müssen verändert/ergänzt werden

- Dimensionen: Programmiersprache – Methodik – Infrastruktur
- Problemfelder: Objektidentität – Referenzierung, Aliasing, Kapselung – Vererbung – Modularisierung

Fazit: OO ist gute Diskussionsbasis

Thema der VL

Befehle zur Ablaufsteuerung:
Verändern des (normalen) Kontrollflusses

Inhalt

Sprünge

Ausgänge

Ausnahmen

Sprünge (Goto)

'n Sprung, wat'n dat'n?

Befehle bislang ...

```
C ::=  skip
     | V := E
     | new (V)
     | p(E1, ... , Ek)
     | begin C1; ... ; Cn end
     | if E then C else C
     | case E of v1 : C1  ... vn: Cn end
     | while E do C
     | repeat C until B
     | for i in L do C end
```

nun auch:

```
C ::= ...
    |  M: C -- Markierung
    |  goto M -- Sprung (Markenbenutzung)
```

Eine alte Debatte ...

- E.W. Dijkstra: *Goto statement considered harmful*, CACM 11,147–148, 1968.

  "The goto statement as it stands is just too primitive;
  it is too much an invitation
  to make a mess of one's program."

- D.E. Knuth: *Structured Programming with goto statements*, ACM Computing Surveys 6,261–302, 1974.

  "There are the occasions
  when the efficiency of the goto
  outweighs its harm to readability."

Und was ist so schlimm am “goto”?

zu diskutieren sind

- Ein/Ausgänge von Befehlen
- Sprünge aus Blöcken in andere Blöcke
  (Block: Programmstück, das die Reichweite von Bindungen begrenzt)
- Sprünge aus (rekursiven) Prozeduren

$\rightsquigarrow$

Programme mit Sprüngen sind nicht leicht zu verstehen (?)

Bsp.: Ein/Ausgänge

```
if E1 then C1
      else begin C2;
                 goto L
           end;
C3;
while E2 do begin C4;
                  L:C5
            end;
```

Bsp: Rekursive Prozedur

```
procedure print (num, width: Natural);
begin
  if width <= 0  then goto L;
  if num   >= 10 then print(num div 10, width-1);
  write(chr(ord('0') + (num mod 10))
end;

  ...
  print (n,w);
  ...
  L:
```

Entwurfsentscheidungen in Programmiersprachen

- kein goto: Modula-2, Java
- mit goto:

Sprache	Sprungziel
Algol 60, C, Ada	identifier
Fortran, Pascal	integer Konstanten
PL/I	Markenvariablen

“Zähmung” in Pascal

Markierungen:

- müssen deklariert werden
- gelten im umgebenden Block
- lassen sich nicht
  - als Parameter übergeben,
  - speichern,
  - verändern.

Erlaubte Sprungziele:

- Konstanten (als keine Ausdrücke!)
- Sprünge nur in ‘aktive Befehlsgruppen’

$\rightsquigarrow$ Befehlsgruppen haben nur einen Eintrittspunkt

Befehlsgruppe: zusammengesetzter Befehl, Unterprogramm
aktiv: in Ausführung begriffen

Ein in Pascal erlaubtes Beispiel

```
procedure sub1;
  label 100;

  procedure sub2;
  begin ...
        goto 100;
  end; { of sub2 }

  begin ...
        100: ...
end; { of sub1 }
```

Ausgänge
(exit, return, continue, . . .)

Das Prinzip

Ein Programmteil wird (vorzeitig) verlassen,
entspricht einem Sprung an das Ende dieses Programmteils

Vorteile gegenüber Sprüngen

- kein Rückwärtssprung – keine Probleme mit Termination
- kein Sprung über Prozedur/Blockgrenzen
- Alle Befehle haben einen Eingang,
  aber evtl. mehrere Ausgänge

Formen von Ausgänge

Verlassen von Schleifen

`exit Id [ when E ]` (Ada)

Rückkehr aus Prozeduren und Funktionen

`return [ E ]` (Ada)

Übergang zum nächsten Schleifendurchlauf

`continue` (C, C++, Java)

Anhalten des Programms

`halt` (Ada)

Beispiel: Abbruch von Schleifen

```
search:
for m in Month loop
  for d in Day loop
    if matches (diary(m,d),i)
    then
      keydate:= (m, d);
      exit search;
    end if;
  end loop;
end loop;
```

Beispiel: Rückkehr aus einer Funktionen

```
function gcd (in i,j: Positive) return Positive is
  r: Positive;
begin
  loop
    r := i mod j;
    if r=0 then return j; end if;
    i := j;
    j := r;
  end loop;
end;
```

Ausnahmen (exception)

Ausnahmesituationen

durch Hardware/Laufzeitsystem feststellbarer Zustand,
der möglicherweise eine spezielle Behandlung erfordert

- arithmetischer Überlauf
- Division durch Null
- Speicherfehler
- unvollständige Ein-Ausgabe-Operation
- unvollständige Funktionsdefinitionen (SML)
- Verletzen von Indexgrenzen
- fehlgeschlagene Speicherallokation
- . . .

Behandlung von Ausnahmesituationen

- “Panik”: Programm anhalten

Behandlung von Ausnahmesituationen

- “Panik”: Programm anhalten
- “Do it Yourself”

Behandlung von Ausnahmesituationen

- “Panik”: Programm anhalten
- “Do it Yourself”
  (Behandlung programmiert der Benutzer – wenn sie/er dran denkt)

Ausnahmen in Programmiersprachen

Form:

`throw A`	– Auslösen (werfen, wecken, raise)
`when A then C`	– Behandeln (handle, fangen, catch)

zentrale Fragen:

- Wo wird die Behandlung spezifiziert?
- Wie wird die Ausnahme ihrer Behandlung zugeordnet?
- Fortsetzung nach der Ausnahmebehandlung?

Weitere Fragen:

- Gibt es eingebaute Ausnahmen?
- Können eingebaute Ausnahmen selbst ausgelöst werden?
- Sind Hardwarefehler als Ausnahmen realisiert?
- Wie können neue Ausnahmen spezifiziert werden?
- Standard-Behandlungen für nicht abgefangene Ausnahmen?
- Können Ausnahmen abgeschaltet werden?

Ausnahmen in Ada

Wo wird die Behandlung spezifiziert?

Block/Unitende.

```
begin
-- Block/Unit
exception
  when exception_name_1 => -- erster Handler
  when exception_name_2 => -- zweiter Handler
end;
```

Wie wird die Ausnahme ihrer Behandlung zugeordnet?

Ausnahme x tritt auf in Block/Unit B

1. B hat Handler für x – lokale Behandlung.
2. B hat keinen Handler für x : Propagieren von x

Propagieren:

- Prozedur $\leadsto$ "point of call" in der aufrufenden Unit
- Unit $\leadsto$ aufrufende Unit (bis hin zum Hauptprogramm)
- Block $\leadsto$ aufrufenden Block, direkt nach dem aufrufenden Block.
- . . .

Ausnahme im Deklarationsteil

```
procedure River is
   current_flow : FLOAT := GET_FLOW;
   ...
   begin
      ...
   end River
```

Fortsetzung nach der Ausnahmebehandlung?

Block/Unit, die eine Ausnahme auslöst, wird beendet –

zusammen mit allen Units,
an welche die Ausnahme propagiert wurde,
die sie aber nicht behandeln konnten.

Beispiel: Wdh. eines Statements nach Ausnahme

```
type AGE_TYPE is range 0...125;
type AGE_LIST_TYPE is array (1..4) of AGE_TYPE;
package AGE_IO is new INTEGER_IO (AGE_TYPE);
use AGE_IO;
AGE_LIST: AGE_LIST_TYPE;
...
begin
```

```
for AGE_COUNT in 1..4 loop
  loop
    EXECEPT_BLK:
      begin
        PUT_LINE(‘‘Enter an integer in the range 0...125’’);
        GET(AGE_LIST(AGE_COUNT));
        exit;
      exception
        when DATA_ERROR => ...
        when CONSTRAINT_ERROR => ..
      end EXECTPT_BLK;
  end loop;
end loop;
```

Antworten auf weitere Fragen:

- Gibt es eingebaute Ausnahmen?

  `CONSTRAINT_ERROR, NUMERIC_ERROR,`
  `PROGRAM_ERROR, STORAGE_ERROR,`
  `TASKING_ERROR`

- Können eingebaute Ausnahmen selbst ausgelöst werden?

  raise `exception_name`

- Wie können neue Ausnahmen spezifiziert werden?

  `exception_name_list:` **exception**

- Standard-Behandlungen für nicht abgefangene Ausnahmen?

  default Handler für eingebaute Ausnahmen:
  Führen zum Terminieren des Programms

- Können Ausnahmen abgeschaltet werden?

  durch Übersetzerdirektiven im Programm
  pragma SUPPRESS (check_name)