Grundlagen der Programmierung

Vorlesung 9 vom 16.12.2004

Sebastian Iwanowski
FH Wedel

Konstruktion von Schleifen

Lösungsverfahren:

1) Finde zu gegebenem ψ eine Zerlegung in I und ¬β: $I \wedge \neg\beta \Rightarrow \psi$
2) Sorge dafür, dass I schon vor der Schleife gilt: Finde S_0 mit: $\{\varphi\}\ S_0\ \{I\}$
3) Finde eine Rumpfanweisung S derart, dass:

a) $\{I \wedge \beta\}$	S	$\{I\}$	(Invariantenbedingung)
b) $\{I \wedge \beta \wedge (z=z_0)\}$	S	$\{z<z_0\}$	(Fortschrittsbedingung)
c) $I \wedge (z \leq 0)$	$\Rightarrow$	$\neg\beta$	(Terminierungsbedingung)

```
{φ}
S0
{I}
while β do
begin
        {I ∧ β ∧ (z=z0} }
        S
        {I ∧ (z<z0) }
end
{I ∧ ¬β}
{ψ}
```

*Die Zahl z heißt **Variante***
(in jedem Schleifendurchlauf ändert sie sich)

Konstruktion von Schleifen

Lösungsverfahren:

1) Finde zu gegebenem ψ eine Zerlegung in I und ¬β: $I \wedge \neg\beta \Rightarrow \psi$
2) Sorge dafür, dass I schon vor der Schleife gilt: Finde S_0 mit: $\{\varphi\}\ S_0\ \{I\}$
3) Finde eine Rumpfanweisung S derart, dass:

a) $\{I \wedge \beta\}$	S	$\{I\}$	(Invariantenbedingung)
b) $\{I \wedge \beta \wedge (z=z_0)\}$	S	$\{z<z_0\}$	(Fortschrittsbedingung)
c) $I \wedge (z \leq 0)$	$\Rightarrow$	$\neg\beta$	(Terminierungsbedingung)

```
{φ}
S0
{I}
while β do
begin
        {I ∧ β ∧ (z=z0} }
        S
        {I ∧ (z<z0) }
end
{I ∧ ¬β}
{ψ}
```

Die Gültigkeit dieses allgemeinen Verfahrens 1) bis 3) beweist man einmalig durch vollständige Induktion.

Danach muss man für jede spezielle Aufgabe nur noch die Gültigkeit der einzelnen geforderten Bedingungen nachweisen.

Konstruktion von Schleifen

Lösungsverfahren:

1) Finde zu gegebenem ψ eine Zerlegung in I und ¬β: $I \wedge \neg\beta \Rightarrow \psi$
2) Sorge dafür, dass I schon vor der Schleife gilt: Finde S_0 mit: $\{\varphi\}\ S_0\ \{I\}$
3) Finde eine Rumpfanweisung S derart, dass:

a) $\{I \wedge \beta\}$	S	$\{I\}$	(Invariantenbedingung)
b) $\{I \wedge \beta \wedge (z=z_0)\}$	S	$\{z<z_0\}$	(Fortschrittsbedingung)
c) $I \wedge (z \leq 0)$	$\Rightarrow$	$\neg\beta$	(Terminierungsbedingung)

```
{φ}
S0
{I}
while β do
begin
        {I ∧ β ∧ (z=z0)}
        S
        {I ∧ (z<z0)}
end
{I ∧ ¬β}
{ψ}
```

Dieses Verfahren kann man bei bekanntem S_0, β, S und ψ auch für die Verifikation einsetzen: Prüfe die Bedingungen in 1), 2), 3a), 3b), 3c) nach. Dann erübrigt sich die Anwendung einer vollständigen Induktion im Einzelfall.

Zusammenfassung: Schleifen

Wesentliche Elemente bei der Verifikation von Schleifen:

1) **Beweise, dass die Berechnungen in der Schleife so fortschreiten, dass <u>nach</u> der Schleife das Richtige berechnet ist.**

   (falls das Programm dort jemals ankommt)

   Wesentliches Beweiselement: ***Invariantenbedingung***

2) **Beweise, dass die Schleife zum Ende kommt.**

   Wesentliche Beweiselemente:
   Variante und Terminierungsbedingung

$\{\varphi\}$
S_0
$\{I\}$
while β do
begin
  $\{I \wedge \beta \wedge (z=z_0)\}$
  S
  $\{I \wedge (z<z_0)\}$
end
$\{I \wedge \neg\beta\}$
$\{\psi\}$

Grundlagen der Programmierung

1. Einführung
   - Grundlegende Eigenschaften von Algorithmen und Programmen
2. Logik
   - Aussagenlogik
   - Prädikatenlogik
3. Programmentwicklung und –verifikation
   - Grundlagen der Programmverifikation
   - Zuweisungen und Verbundanweisungen
   - Verzweigungen
   - Schleifen
   - ➔ Modularisierung
   - Rekursion
4. Entwurf und Analyse von Algorithmen
   - Klassifikation von Algorithmen
   - Programmierung von Algorithmen
   - Bewertung von Algorithmen

Modularisierung

Grundlegende Idee: Zerlege ein Problem in Teilprobleme

1. Beispiel: Schreib einen Brief

```
Briefkopf;
Anrede;
Briefinhalt;
Briefschluss
```

} **Prozeduren**

Die 4 Teilprobleme können unabhängig voneinander gelöst werden und wiederverwendet werden.

Vorteile:

1) **Übersichtlichkeit**
2) **Wiederverwendbarkeit**

Modularisierung

Grundlegende Idee: Zerlege ein Problem in Teilprobleme

2. Beispiel: Berechne zu einem Kalenderdatum D und einer Tageszahl n das Kalenderdatum D+n

```
LiesEingabedatum;
LiesTageszahl;
BerechneAusgabedatum;
GibAusgabedatumAus;
```

Problem: Hier müssen Daten zwischen den Prozeduren ausgetauscht werden.

Lösung: Prozeduren mit Parametern

```
LiesEingabedatum (in);
LiesTageszahl (n);
BerechneAusgabedatum (in, n, out);
GibAusgabedatumAus (out);
```

Modularisierung

Prozeduren mit Parametern

```
LiesEingabedatum (in);
LiesTageszahl (n);
BerechneAusgabedatum (in, n, out);
GibAusgabedatumAus (out);
```

Eingabeparameter:

- dienen der Übermittlung von Werten aus dem aufrufenden Programm an die Prozedur

Ausgabeparameter:

- dienen der Übermittlung von Werten aus der Prozedur an das aufrufende Programm

1. Übergabetechnik: Call by reference (Variablenparameter)

- Prozedur und aufrufendes Programm teilen sich denselben Speicherplatz.
- In Pascal wird diese Übergabetechnik bei den VAR-Parametern angewandt.
- In manchen Programmiersprachen gibt es nur diese Übergabetechnik.

Modularisierung

Prozeduren mit Parametern

```
LiesEingabedatum (in);
LiesTageszahl (n);
BerechneAusgabedatum (in, n, out);
GibAusgabedatumAus (out);
```

Eingabeparameter:

- dienen der Übermittlung von Werten aus dem aufrufenden Programm an die Prozedur

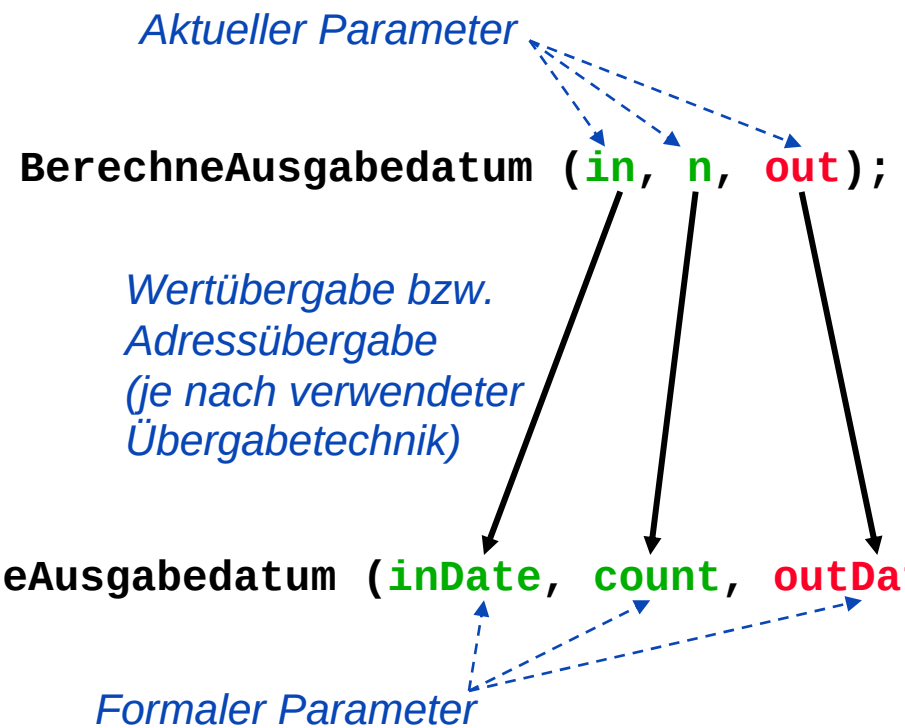
2. Übergabetechnik: Call by value (Wertparameter)

- Prozedur und aufrufendes Programm legen für denselben Parameter verschiedene Speicherplätze an.
- Beim Aufruf wird der Wert des aufrufenden Programms in den entsprechenden Speicherplatz der Prozedur kopiert.
- Eine Rückgabe von Werten durch die Prozedur ist auf diese Weise nicht möglich (also nur als Eingabeparameter verwendbar).
- In Pascal wird dieser Übergabetechnik bei den Nicht-VAR-Parametern angewandt.
- In manchen Programmiersprachen gibt es nur diese Übergabetechnik.

Modularisierung

Parameteridentifikation zwischen aufrufendem Programm und Prozedur

im aufrufenden Programm:



in der Prozedur:

```
BerechneAusgabedatum (inDate, count, outDate);
```

Formaler Parameter

- Es werden jeweils die Parameter identifiziert, die an der gleichen Position in der Parameterliste stehen.

Modularisierung

Anforderungen an die Parameter

an die formalen Parameter (in der Prozedur):

- Parameter müssen Variablennamen sein.
- Bei Call by reference steht dieser Variablenname für die Speicheradresse, die übergeben wurde.
- Bei Call by value steht dieser Variablenname für einen in der Prozedur neu eingerichteten Speicherplatz, in den der Wert des aufrufenden Programms kopiert wird.

an die aktuellen Parameter (im aufrufenden Programm):

- Bei Call by reference müssen die Parameter Variablennamen sein. Sie stehen für die Speicheradresse, die an die Prozedur übergeben wird.
- Bei Call by value dürfen die Parameter beliebige Wertausdrücke sein. Der Ausdruck muss vor Prozedurbeginn ausgewertet werden und wird dann in den neu eingerichteten Speicherplatz der Prozedur kopiert.

Modularisierung

Warum gibt es 2 Parameterübergabetechniken in derselben Programmiersprache ?

für Pascal: Warum gibt es Wertparameter ?

- Wertparameter können bequemer mit aktuellen Parametern bedient werden (beliebige Ausdrücke sind zulässig).

*Programmieren wird **bequemer***

- Falls eine Variable als Wertparameter übergeben wird, dann kann man sich sicher sein, dass diese auch von einer nicht ganz durchsichtigen Prozedur auf keinen Fall verändert wird.

*Programmieren wird **zuverlässiger***

Die mathematisch exakte Verifikation von Programmen mit call-by-reference-Parametern ist praktisch unmöglich !

Modularisierung

Wann sollte man welche Technik einsetzen ?

Empfehlung: ***Parameter sollten nur zur Übergabe von Werten an die Prozedur benutzt werden und nicht zur Übergabe von Rückgabewerten an das aufrufende Programm.***

aus Übersichtlichkeits- und Zuverlässigkeitsgründen

Folgerung: Dann braucht man ja eigentlich nur call by value ... **?**

Achtung: Call by reference ist wesentlich effizienter bezüglich des Speicherplatzverbrauchs !

Idealfall:
1. Benutze nur call by reference.
2. Verwende den Parameter dennoch nur als Eingabeparameter.

Aber wer gewährleistet die Einhaltung der zweiten Forderung ?

Kompromiss: Benutze call by value für kleine (einfache) Daten und call by reference für große (zusammengesetzte).

Modularisierung

Rückgabe von Werten von der Prozedur an das aufrufende Programm

Wie soll man das bei Befolgung der eben gegebenen Empfehlung bewerkstelligen ?

Antwort:

entsprechend der Schreibweise der Mathematik: $y = f(x_1, x_2, \ldots, x_k)$

- Prozeduren erhalten Rückgabewerte
- Wenn der Rückgabewert ein beliebig komplexer Wert sein darf, so können wir uns auf die Forderung nach **einem** Rückgabewert beschränken.
- In Pascal heißen Prozeduren mit Rückgabewert **Funktionen**

Modularisierung

Prozeduren mit Rückgabewert (Funktionen)

```
in := LiesEingabedatum ();
n := LiesTageszahl ();
out := BerechneAusgabedatum (in, n);
GibAusgabedatumAus (out);
```

kein Rückgabewert ----→

in der Prozedur (Funktion):

```
BerechneAusgabedatum (inDate, count);
begin
      .
      .
      .
      return outDate;
end
```

im aufrufenden Programm:

```
out := BerechneAusgabedatum (in, n);
```

- Die Prozedur darf in der return-Anweisung einen beliebigen Wertausdruck haben.
- Dieser wird bei der Ausführung der return-Anweisung ausgewertet
- Nach der return-Anweisung wird die Prozedur beendet und der Rückgabewert im Hauptprogramm an der entsprechenden Stelle eingesetzt.

Modularisierung

Variablen in Prozeduren

- Da Prozeduren alle Funktionalitäten von Programmen haben, darf man in ihnen auch Variablen definieren.
  (falls die Programmiersprache das erlaubt)
- Variablen sollten grundsätzlich nur lokal verwendet werden, auch wenn die Programmiersprache mehr erlaubt (wie Pascal)
- Die Kommunikation zwischen zwei verschiedenen Programmteilen (Prozeduren) hat in der Regel über Parameter zu erfolgen (und nicht durch Benutzung gemeinsamer Variablen)
- Daher behandeln wir hier keine Sichtbarkeitsregeln.

Unterscheide Parametervariablen von anderen lokalen Variablen !

Modularisierung

Verifikation bei Prozeduren

- Die Verifikation innerhalb von Prozeduren unterscheidet sich nicht von der Verifikation allgemeiner Programme.

Einziger neuer Punkt der Beachtung: **Parameterübergabe**

- Die aktuellen Parameter müssen die Vorbedingungen der entsprechenden formalen Parameter erfüllen.

Die meisten Compiler unterstützen das durch Typprüfungen.

- Der Rückgabewert muss die Vorbedingungen für die zugewiesene Variable des Hauptprogramms erfüllen.

Analoges gilt bei der Benutzung von Variablenparametern.

Rekursion

oder: Wie programmiert man wirklich elegant ?

```
procedure f (n: Integer): Integer
if (n=0)
  then
    return 1
  else
    return n • f(n-1)
end {f}
```

Was berechnet diese Prozedur ?

Gibt es irgendwelche Vorbedingungen ?

Wie beweist man das alles ?

Rekursion

Verifikation von rekursiven Prozeduren

Rekursive Prozeduren haben viel mit Schleifen gemeinsam

- Daher bietet sich eine ähnliche Verifikationsstechnik an:

1) **Beweise, dass die Berechnungen in jedem Durchlauf der Prozedur so fortschreiten, dass nach Abbruch der Rekursion das Richtige berechnet ist.**

   (falls die Rekursion irgendwann einmal abbricht)

   Mögliches Beweiselement: ***Invariantenbedingung***

2) **Beweise, dass die Rekursion irgendwann einmal abbricht.**

   Essentielle Beweiselemente: ***Variante und Terminierungsbedingung***

Wichtigste Beweistechnik: Vollständige Induktion

Beim nächsten Mal:

Rekursion (Vertiefung)