Seminar: Spezifikation und Verifikation
SS 2001
Prof. Dr. K. Madlener

Robert Eschbach
Referenten: Jens Brandt, Rüdiger Grammes

AG Grundlagen der Informatik, FB Informatik, Universität Kaiserslautern, 67653 Kaiserslautern
URL: `http://www-madlener.informatik.uni-kl.de/`

ZUSAMMENFASSUNG. Dieses Dokument liefert Informationen zum Seminar 'Spezifikation und Verifikation', SS 2001, AG Grundlagen der Informatik. Das Dokument wird im SS 2001 laufend aktualisiert.

Stand: 19. Juni 2001

Inhaltsverzeichnis

KAPITEL 1

Einführung

1. Aktuelles

(1) 09.05.01: Termine hinzugefügt (siehe Durchführung/Zeitlicher Ablauf).

2. Eintrag in das Vorlesungsverzeichnis

(1) Seminar: Spezifikation und Verifikation
(2) 2 Std., n.V., s. Aushang
(3) A) In diesem Seminar wird die algebraische Semantik imperativer Programme behandelt, und somit die Methoden der algebraischen Spezifikation verwendet um Verifikationsaufgaben imperativer Programme zu behandeln. Die Teilnehmer sollen anhand von Originalliteratur einen Vortrag erarbeiten und halten.
(4) B) Joseph A. Goguen, Grant Malcom: Algebraic Semantics of Imperative Programs, MIT Press, 1996.
(5) C) Vordiplom.
(6) D) Seminarschein nach erfolgreichem Vortrag.
(7) E) Kann als integriertes Seminar zum Praktikum 89-722 gewählt werden.

3. Zielsetzung

Die Teilnehmer sollen lernen,

(1) einen längeren Vortrag vor Publikum zu halten,
(2) innerhalb des Vortrags auf Fragen zu reagieren und
(3) entstehende Diskussionen zu moderieren.

Dabei wird thematisch der Bereich der Semantik und Verifikation von Programmen aufgegriffen.

4. Durchführung

4.1. Zeitlicher Ablauf. Zu Beginn des Semesters wird von uns eine Einführung in folgende Gebiete gegeben:

(1) Algebraische Spezifikation,
(2) Termersetzung,
(3) Gleichungsdeduktion.

Danach halten die Teilnehmer ihre Vorträge (jeweils in 34/420).

(1) 14.05.01: 08:30 - 10:00 Uhr.
Referent Rüdiger Grammes.
Thema: Iteration.
(2) 21.05.01: 08:30 - 10:00 Uhr.
Referent Rüdiger Grammes.
Thema: Arrays.
(3) 28.05.01:
Referent: Jens Brand
Thema: Nicht-rekursive Prozeduren.
(4) 29.05.01:
Referent: Jens Brand
Thema: Rekursive Prozeduren.

4.2. Arbeiten der Teilnehmer.

(1) Themen:
   (a) Iteration, Felder (arrays) ([**GM96**], Kapitel 5, 6),
   (b) Prozeduren ([**GM96**], Kapitel 7).

(2) Vortrag: Jedes Thema wird in zwei Vorträgen von dem entsprechenden Referenten vorgestellt. Ein Vortrag soll dabei zeitlich einer Vorlesung, also ungefähr 90 Min., entsprechen. Der Vortrag umfasst ebenfalls die praktische Vorführung von Beispielen an dem System OBJ ([**GM96**]) und Diskussionen mit den Teilnehmern. Als Vorbereitung wird den Teilnehmern die Lektüre von [**Par93**] empfohlen.

(3) Dokumentation: Um eine einheitliche Dokumentation und Präsentation zu erhalten, sollen die Folien in LaTeX(Klasse `slides.cls` bzw. `foils`) erstellt werden.

KAPITEL 2

Iteration

Referent: Rüdiger Grammes
Datum: 14.05.01

1. Gliederung

- Einleitung
- Spezifikation: Syntax und Semantik der While-Schleife
- Verifikation: Invarianten und Partielle Korrektheit
- Verifikation: Bewertungsfunktion und Totale Korrektheit
- Zusammenfassung

2. Einleitung

Beispiel einer While-Schleife:

```
while (i != 3) {
   i = i + 1;
}
```

- Anzahl der Schleifendurchläufe hängt vom Startwert von i ab .
- Für i $\leq$ 3 gilt i = 3 nach Abarbeitung der Schleife .
- Terminiert nicht für i $>$ 3 .

3. Syntax und Semantik der While-Schleife

Syntax:

```
obj PGM3 is ex PGM2 .
  op while_do_od : Tst Pgm -> Pgm .
endo
```

Semantik:

```
obj SEM3 is pr SEM2 . pr PGM3 .
  var S : Store . var T : Tst . var P : Pgm .
  cq S ; while T do P od = S ; P ; while T do P od if S[[T]] .
  cq S ; while T do P od = S if not S[[T]] .
endo
```

4. Verifikation

Beispiel: Programm zur Berechnung der Fakultät von n:
Es gilt: S[['N]] ist n und 0 $\leq$ n .

```
'C := 0 ; 'X := 1 ;
while 'C < 'N do
     'C := 'C + 1 ; 'X := 'X * 'C
od .
```

Initial und nach jedem Schleifendurchlauf ist S[['X]] = S[['C]] ! .

5. Invarianten

Eine Invariante ist ein Prädikat auf Stores, das in jedem Schleifendurchlauf gültig ist.
while T do P od Schleife, I Pradikat auf Stores.
Dann I(S) und S[[T]] $\Rightarrow$ I(S ; P).

6. Partielle Korrektheit von While-Schleifen

init ; while T do P od ist hinsichtlich Vorbedingung pre und
Nachbedingung post partiell korrekt, falls ein Prädikat inv existiert, mit:

- ($\forall$ S : Store) pre(S) $\Rightarrow$ inv(S ; init) .
- ($\forall$ S : Store) inv(S) and not(S[[T]]) $\Rightarrow$ post(S) .
- ($\forall$ S : Store) inv(S) and S[[T]] $\Rightarrow$ inv(S ; P) .

7. Berechnung des größten gemeinsamen Teilers I

Definition des größten gemeinsamen Teilers:

```
obj GCD is ex ZZ .
  op gcd : Int Int -> Int [comm] .
  vars I J : Int .
  cq gcd(I,I) = I if 0 < I .
  cq gcd(I,J + (- I)) = gcd(I,J) if 0 < I and I < J .
endo
```

8. Berechnung des größten gemeinsamen Teilers II

Schrittweise Konstruktion des Programms
Spezifikation:

```
pre(x,y,S) = S[['X]] is x and S[['Y]] is y and
             0 < x and 0 < y .

post(x,y,S) = (S[['X]]) is gcd(x,y) .
```

9. Berechnung des größten gemeinsamen Teilers III

Führe Schleifenrumpf aus, bis der triviale Fall `gcd(I,I)` erreicht ist.

```
while not('X is 'Y) do ... od .
```

Wähle den Rumpf derart, daß folgende Invariante gilt.

```
inv(S) = gcd(S[['X]],S[['Y]]) is gcd(x,y) and
         0 < x and 0 < y .
```

Die Invariante folgt direkt aus der Vorbedingung. Weiterhin folgt aus
der Invariante und der Negation des Guards direkt die Nachbedingung.

10. Berechnung des größten gemeinsamen Teilers IV

Benutze Eigenschaft des gcd:

```
gcd(I,J) = gcd(I,J + (-I)) if I < 0 and I < J .
```

Ziehe den kleineren Wert vom größeren ab:

```
if 'X < 'Y then 'Y := 'Y - 'X else 'X := 'X - 'Y fi
```

Damit ergibt sich folgendes Programm:

```
while not('X is 'Y) do
  if 'X < 'Y then 'Y := 'Y - 'X else 'X := 'X - 'Y fi
od .
```

11. Terminierung: Definitionen

Ein Programm terminiert auf einem Store S gdw. S ; P gleich einem Term von Typ Store ist.
P↓S gdw. (∃ S' : Store) S ; P = S' .
Ein Programm terminiert für ein Prädikat I gdw. das Programm auf allen Stores S terminiert für die I(S) gilt.
P↓I gdw. (∀ S : Store) I(S) ⇒ P↓S

12. Bewertungsfunktion

loop = while T do P od   While-Schleife.

Führe Bewertungsfunktion M : Store -> Int ein, die eine Aussage trifft wie weit die Schleife von der Terminierung entfernt ist.
zu zeigen:

- M ist nach unten beschränkt .
- M wird in jedem Schleifendurchlauf reduziert .

13. Terminierung

Theorem
if

(∀ S : Store)   inv(S) and S[[T]]
⇒ inv(S ; P) and P↓S and
$0 \leq$ M(S) and M(S ; P) $<$ M(S)

then

(∀ S : Store)   inv(S)
⇒ inv(S ; loop) and not(S ; loop [[T]]) and
loop↓S .

14. Totale Korrektheit von While-Schleifen

init ; while T do P od ist hinsichtlich Vorbedingung pre und
Nachbedingung post korrekt, falls ein Prädikat inv und eine Funktion
M : Store -> Int existiert, mit:

- (∀ S : Store) pre(S) ⇒ inv(S ; init) .
- (∀ S : Store) inv(S) and not(S[[T]]) ⇒ post(S) .
- (∀ S : Store)   inv(S) and S[[T]]
  ⇒ inv(S ; P) and P↓S and
  $0 \leq$ M(S) and M(S ; P) $<$ M(S) .

15. Beispiel : Terminierung des gcd-Programms

```
while not('X is 'Y) do
  if 'X < 'Y then 'Y := 'Y - 'X else 'X := 'X - 'Y fi
od .
```

Auswahl einer Bewertungsfunktion:

- Entweder 'X oder 'Y wird durch den Schleifenrumpf reduziert .
- Es ist 'X $>$ 0 und 'Y $>$ 0 .

Bewertungsfunktion: M(S) = S[[X + Y]] .

16. Zusammenfassung

- Spezifikation
  - Definition der Syntax und Semantik von While-Schleifen
- Verifikation
  - Beweis der Korrektheit eines Programms hinsichtlich Vor- und Nachbedingung mittels Invarianten

– Durch While-Schleifen können nichtterminierende Programme entstehen
– Nachweis der Terminierung eines Programms mittels einer nach unten beschränkten, streng monoton fallenden Bewertungsfunktion auf Stores.

17. OBJ

```
obj GCD is ex ZZ .
   op gcd : Int Int -> Int [comm] .
   vars I J : Int .
   cq gcd(I,I) = I if 0 < I .
   cq gcd(I,J + (- I)) = gcd(I,J) if 0 < I and I < J .
endo

obj GCD1 is ex ZZ .
   op gcd1 : Int Int -> Int [comm] .
   vars I J : Int .
   cq gcd1(I,I) = I if 0 < I .
   cq gcd1(I,J) = gcd1(I,J + (- I)) if 0 < I and I < J .
endo

obj GCD2 is ex ZZ .
   op gcd2 : Int Int -> Int [comm] .
   vars I J : Int .
   cq gcd2(I,I) = I if 0 < I .
   cq gcd2(I,J + I) = gcd2(I,J) if 0 < I and 0 < J .
endo

obj GCDP is pr SEM .
   let body = if 'X < 'Y then 'Y := 'Y - 'X
                      else 'X := 'X - 'Y fi .
   let gcdp = while not('X is 'Y) do body od .
endo

th PROOF is pr GCDP .
            pr GCD .
  ops x y : -> Int .
  ops inv post : Store -> Bool .
  var S : Store .
  eq post(S) = (S[['X]]) is gcd(x,y) .
  eq inv(S) = gcd(S[['X]],S[['Y]]) is gcd(x,y) and
              0 < (S[['X]]) and 0 < (S[['Y]]) .
  op s : -> Store .
endth


*** Zeige pre -> inv ***
open PROOF .

eq s[['X]] = x .
  eq s[['Y]] = y .
  eq 0 < x = true .
  eq 0 < y = true .
  red inv(s) .
close
```

```
*** Ab nun gilt die Invariante ***
openr PROOF .
  eq gcd(s[['X]],s[['Y]]) = gcd(x,y) .
  eq 0 < (s[['X]]) = true .
  eq 0 < (s[['Y]]) = true .
close

*** zeige ! T -> post ***
open PROOF .
  eq s[['Y]] = s[['X]] .
*** Lemma ***
  red inv(s) .
  eq s[['X]] = gcd(x,y) .
  red post(s) .
close

*** Ab nun gilt T ***
openr PROOF .
  eq not((S[['Y]]) is (S[['X]])) = true .
close

*** Zeige Invariante ***

*** Fall 'X < 'Y
open PROOF .
  eq (s[['X]]) < (s[['Y]]) = true .
  red inv(s ; body) .
close

*** Fall not('X < 'Y) ***
open PROOF .
  eq (s[['X]]) < (s[['Y]]) = false .
*** da not('X is 'Y) (wegen T) ***
  eq (s[['Y]]) < (s[['X]]) = true .
  red inv(s ; body) .
close



*** Terminierung ***


openr PROOF .
  op M : Store -> Int .
  eq M(S) = S[['X + 'Y]] .
  red 0 <= M(s) .
close


*** Fall ('X < 'Y) ***
open PROOF .
  eq (s[['X]]) < (s[['Y]]) = true .
  red M(s ; body) < M(s) .
```

```
close

*** Fall not('X < 'Y) ***
open PROOF .
  eq (s[['X]]) < (s[['Y]]) = false .
  red M(s ; body) < M(s) .
close
```

KAPITEL 3

Arrays

Referent: Rüdiger Grammes
Datum: 21.05.01

1. Gliederung

- Einleitung
- Spezifikation
  - Arrays
  - Syntax von Arrayprogrammen
  - Semantik von Arrayprogrammen
- Verifikation
  - Beispiel
- Zusammenfassung

2. Einleitung

Erweiterung der Sprache um Zuweisungen zu Array-Komponenten:

```
a['X + 2] := 2 * a['X + 1] .
```

Array-Variablen repräsentieren in einem gegebenen Store ein Array.

3. Syntax und Semantik von Arrays

```
obj ARRAY is pr ZZ .
   sort Array .
   op zero : -> Array .
   op  _[_] : Array Int -> Int [prec 5] .
   op  _[_<-_] : Array Int Int -> Array .
   var A : Array .
   var I J K : Int .
   eq zero[I] = 0 .
   eq  (A [ I <- J ])[I]  =  J .
   cq  (A [ I <- J ])[K]  =  A[K]    if  not(I is K) .
endo
```

4. Syntax von Arrayprogrammen

```
obj ARRPGM is ex PGM3 .
  sorts  Arvar Arcomp .
  subsort  Arcomp < Exp .
  ops a b c : -> Arvar .
  op _[_] : Arvar Exp -> Arcomp [prec 1] .
  op _:=_ : Arcomp Exp -> BPgm [prec 20] .
endo
```

5. Semantik

Sei s[[a]] = A und s[['X]] = 0 . Für diesen Store sollte gelten:

```
s[[ a['X] ]] = A[0]
             = (s[[a]])[ s[['X]] ] .
```

Auswertung des Programms a['X] := 4 sollte ergeben:

```
(s ; a['X] := 4)[[a]] = A[0 <- 4]
                      = (s[[a]])[ s[['X]]] <- s[[4]] ] .
```

6. Semantik von Arrayprogrammen I

```
obj SEMARR is pr ARRAY .
              pr ARRPGM .
              pr SEM3 .
  op _[[_]] : Store Arvar -> Array [prec 65] .

  var AV AV' : Arvar .
  var X : Var .
  var E E' : Exp .
  var S : Store .
```

7. Semantik von Arrayprogrammen II

```
  eq initial[[AV]] = zero .
  eq S[[ AV[E] ]] = (S[AV])[ S[[E]] ] .
  eq S ; AV[E] := E' [[AV]] =
       (S[[AV]])[ S[[E]] <- S[[E']] ] .
  cq S ; AV[E] := E' [[AV']] = S[[AV']] if AV =/= AV' .
  eq S ; X := E [[AV]] = S[[AV]] .
  eq S ; AV[E] := E' [[X]] = S[[X]] .
endo
```

8. Zusammenfassung

- Spezifikation
  - Array-Variablen repräsentieren einen Array in einem gegebenen Store.
  - Verwendung von Array-Komponenten als Komponenten denen ein Wert zugewiesen werden kann oder als Ausdrücke.

9. OBJ

```
obj GCD is ex ZZ .
  op gcd : Int Int -> Int [comm] .
  vars I J : Int .
  cq gcd(I,I) = I if 0 < I .
  cq gcd(I,J + (- I)) = gcd(I,J) if 0 < I and I < J .
endo

obj GCD1 is ex ZZ .
  op gcd1 : Int Int -> Int [comm] .
  vars I J : Int .
  cq gcd1(I,I) = I if 0 < I .
  cq gcd1(I,J) = gcd1(I,J + (- I)) if 0 < I and I < J .
endo
```

```
obj GCD2 is ex ZZ .
   op gcd2 : Int Int -> Int [comm] .
   vars I J : Int .
   cq gcd2(I,I) = I if 0 < I .
   cq gcd2(I,J + I) = gcd2(I,J) if 0 < I and 0 < J .
endo

obj GCDP is pr SEM .
   let body = if 'X < 'Y then 'Y := 'Y - 'X
     else 'X := 'X - 'Y fi .
   let gcdp = while not('X is 'Y) do body od .
endo

th PROOF is pr GCDP .
    pr GCD .
  ops x y : -> Int .
  ops inv post : Store -> Bool .
  var S : Store .
  eq post(S) = (S[['X]]) is gcd(x,y) .
  eq inv(S) = gcd(S[['X]],S[['Y]]) is gcd(x,y) and
         0 < (S[['X]]) and 0 < (S[['Y]]) .
  op s : -> Store .
endth


*** Zeige pre -> inv ***
open PROOF .

eq s[['X]] = x .
  eq s[['Y]] = y .
  eq 0 < x = true .
  eq 0 < y = true .
  red inv(s) .
close

*** Ab nun gilt die Invariante ***
openr PROOF .
  eq gcd(s[['X]],s[['Y]]) = gcd(x,y) .
  eq 0 < (s[['X]]) = true .
  eq 0 < (s[['Y]]) = true .
close

*** zeige ! T -> post ***
open PROOF .
  eq s[['Y]] = s[['X]] .
*** Lemma ***
  red inv(s) .
  eq s[['X]] = gcd(x,y) .
  red post(s) .
close

*** Ab nun gilt T ***
openr PROOF .
  eq not((S[['Y]]) is (S[['X]])) = true .
```

```
close

*** Zeige Invariante ***

*** Fall 'X < 'Y
open PROOF .
  eq (s[['X]]) < (s[['Y]]) = true .
  red inv(s ; body) .
close

*** Fall not('X < 'Y) ***
open PROOF .
  eq (s[['X]]) < (s[['Y]]) = false .
*** da not('X is 'Y) (wegen T) ***
  eq (s[['Y]]) < (s[['X]]) = true .
  red inv(s ; body) .
close



*** Terminierung ***


openr PROOF .
  op M : Store -> Int .
  eq M(S) = S[['X + 'Y]] .
  red 0 <= M(s) .
close


*** Fall ('X < 'Y) ***
open PROOF .
  eq (s[['X]]) < (s[['Y]]) = true .
  red M(s ; body) < M(s) .
close

*** Fall not('X < 'Y) ***
open PROOF .
  eq (s[['X]]) < (s[['Y]]) = false .
  red M(s ; body) < M(s) .
close


th MAXP is pr SEM .
  ops l u : -> Int .
  eq l < u = true .
  let init = 'X := l + 1 ; 'M := a[l] .
  let body = if 'M < a['X] then 'M := a['X] else skip fi ;
 'X := 'X + 1 .
  let maxp = init ; while 'X < u do body od .
endth

th PROOF is pr MAXP .
  op s : -> Store .
```

```
  op a0 : -> Array .

  *** assume guard ***
  eq (s[['X]]) < u = true .
  *** assume inv1(s) ***
  eq (s[['X]]) <= u = true .
  eq s[[a]] = a0 .
  *** assume inv2(s) ***
  var I : Int .
  cq a0[I] <= (s[['M]]) = true if l <= I and I < (s[['X]]) .
endth

***Zeige inv1(s) ***
open PROOF .
  eq (s[['M]]) < a0[ s[['X]] ] = true .
  red (s ; body[['X]]) <= u .
  red (s[[a]]) == a0 .
close

open PROOF .
  eq (s[['M]]) < a0[ s[['X]] ] = false .
  red (s ; body[['X]]) <= u .
  red (s[[a]]) == a0 .
close

*** Zeige inv2(s) ***

*** case s[['M]] < a0[ s[['X]] ] ***
th CASE1 is pr PROOF .
  eq (s[['M]]) < a0[ s[['X]] ] = true .
  *** show inv2(s ; body) ***
  op i : -> Int .
  eq l <= i = true .
  eq i <= (s[['X]]) = true .
endth

*** case i = s[['X]] ***
open CASE1 .
  eq i = s[['X]] .
  red a0[i] <= (s ; body [['M]]) .
close

*** case i < s[['X]] ***
open CASE1 .
  eq i < (s[['X]]) = true .
  red a0[i] <= (s[['M]]) .
  red (s[['M]]) <= (s ; body [['M]]) .
*** => a0[i] <= s ; body [['M]] ***
close
```

```
obj ATMOST is pr ARRAY .
  op atmost : Array Int Int Int -> Bool .
  var A : Array . vars L U M : Int .
  cq atmost(A, L, U, M) = true if U <= L .
  cq atmost(A, L, U, M) = atmost(A, L, U - 1, M) and A[U - 1] <= M if L < U .
endo

*** inv2(S) ^= atmost(a0,l,s[['X]],s[['M]]) ***

th PROOF2 is pr MAXP .
             pr ATMOST .
  op inv : Store -> Bool .
  op a0 : -> Array .
  var S : Store .
  eq inv(S) = l <= (S[['X]]) and (S[['X]]) <= u and
              ((S[[a]]) == a0) and
              atmost(a0, l, S[['X]], S[['M]]) .
  op s : -> Store .
  eq (s[['X]]) < u = true .
  eq l <= (s[['X]]) = true .
  eq (s[['X]]) <= u = true .
  eq s[[a]] = a0 .
  eq atmost(a0, l, s[['X]], s[['M]]) = true .
endth

*** case s[['M]] < a0[ s[['X]] ] ***
open PROOF2 .
  eq (s[['M]]) < a0[ s[['X]] ] = true .
  *** therefore ***
  [lemma] eq  atmost(a0, l, s[['X]], a0[ s[['X]] ]) = true .
  red inv(s ; body) .
close

*** case a0[ s[['X]] ] <= s[['M]] ***
open PROOF2 .
  eq a0[ s[['X]] ] <= (s[['M]]) = true .
  red inv(s ; body) .
close
```

KAPITEL 4

Nicht-rekursive Prozeduren

Referent: Jens Brandt Datum: 28.05.01

1. Inhaltsübersicht

- Einführung
- Prozeduren ohne Parameter
- Prozeduren mit `Var`-Parametern
- Prozeduren mit `Exp`-Parametern

2. Einführung

```
program swapdemo;
  var a,b : int;
  begin
    a := 42; b := 23; swap(a,b);
  end.

procedure swap(var x,y : int);
  var t : int;
  begin
    t := x ; x := y ; y := t ;
  end;
```

3. Beispiel swap

```
obj SWAP is pr SEM .

  op swap : -> Pgm.

    *** uses 'X 'Y 'T : Var
    *** for all x y : Int
    *** pre(S,x,y)  = (S[['X]]) is x and (S[['Y]]) is y
    *** post(S,x,y) = (S[['Y]]) is x and (S[['X]]) is y

  eq swap = 'T := 'X ; 'X := 'Y ; 'Y := 'T .

endo
```

4. Parameterlose Prozeduren - Spezifikation

```
obj PROC is pr SEM .

  op proc : -> Pgm .
    *** uses Y^ : Var
    *** uses A^ : Arvar
    *** for all e^ : Int
    *** for all a^ : Array
```

```
    *** pre(S,e^,a^) = ...
    *** post(S,e^,a^) = ...

  eq proc = <body> .
endo
```

5. Parameterlose Prozeduren - Verifikation

Es gelte: $(\forall \vec{s} : \texttt{Store})(\forall \vec{e} : \texttt{Int})(\forall \vec{a} : \texttt{Array})\ pre(\texttt{s}, \vec{e}, \vec{a})$
Die Definition der Prozedur `proc` heißt *korrekt*, wenn:

- `proc` $\downarrow$ `s` (`proc` terminiert auf `s`.)
- $post($`s ; proc`$, \vec{e}, \vec{a})$ (`proc` erfüllt die Nachbedingung.)
- Für alle `V` (vom Typ `Var` oder `Arvar`) nicht in der Liste $\vec{Y}$ oder $\vec{A}$:
  `s ; proc [[V]] = s [[V]]`. (`proc` verändert nur spezifizierte Variablen.)

6. Verifikation swap (1/2)

```
obj SWAP is pr SEM .

  op swap : -> Pgm.

    *** uses 'X 'Y 'T : Var
    *** for all e1 e2 : Int
    *** pre(S,x,y)  = (S[[X1]]) is e1 and (S[[X2]]) is e2
    *** post(S,x,y) = (S[[X2]]) is e1 and (S[[X1]]) is e2

  eq swap = 'T := 'X ; 'X := 'Y ; 'Y := 'T .

endo
```

7. Verfikation swap (2/2)

Es gelte: `(s[['X]]) is e1 and (s[['Y]]) is e2`

- `('T := 'X ; 'X := 'Y ; 'Y := 'T)`$\downarrow$ `s` $\surd$
- `s ; 'T := 'X ; 'X := 'Y ; 'Y := 'T[['X]] is e2 and`
  `s ; 'T := 'X ; 'X := 'Y ; 'Y := 'T[['Y]] is e1)` $\surd$
- `('T := 'X ; 'X := 'Y ; 'Y := 'T)` verändert nur `'X`, `'Y` und `'T`. $\surd$

8. Beispiel swap(X,Y)

```
obj SWAP is pr SEM .
  op swap : Var Var -> Pgm .
  vars X1 X2 : Var .

    *** uses 'T : Var
    *** for all e1 e2 : Int
    *** pre(S,X1,X2,e1,e2)  = (S[[X1]]) is e1 and (S[[X2]]) is e2
    *** post(S,X1,X2,e1,e2) = (S[[X2]]) is e2 and (S[[X1]]) is e1

  eq swap(X1,X2) = 'T := X1 ; X1 := X2 ; X2 := 'T .
endo
```

9. Prozeduren mit `Var`-Parametern: Spezifikation

```
obj PROC is pr SEM .
  op proc : Var^K Arvar^L -> Pgm .
  vars X^ : Var .
  vars AV^ : Arvar .
    *** uses Y^ : Var
    *** uses A^ : Arvar
    *** for all e^ : Int
    *** for all a^ : Array
    *** pre(S,X^,AV^,e^,a^) = ...
    *** post(S,X^,AV^,e^,a^) = ...
  eq proc(X^,AV^) = <body> .
endo
```

10. Prozeduren mit `Var`-Parametern: Verifikation

Es gelte: $pre(\mathtt{s}, \vec{X}, \vec{AV}, \vec{e}, \vec{a})$
Die Definition der Prozedur `proc` heißt *korrekt*, wenn:

- `proc(`$\vec{X}$`,`$\vec{AV}$`)` $\downarrow$ `s`
- $post($`s ; proc(`$\vec{X}$`,`$\vec{AV}$`)`$, \vec{e}, \vec{a})$
- Für alle `V` (vom Typ `Var` oder `Arvar`) nicht in der Liste $\vec{Y}$ oder $\vec{A}$:
  `s ; proc(`$\vec{X}$`,`$\vec{AV}$`) [[V]] = s [[V]]`.
- Die Variablen aus $\vec{X}, \vec{Y}, \vec{V}, \vec{AV}$ sind paarweise verschieden.

11. Beispiel set(E) (1/2)

```
obj SETEXP is pr SEM .
  op set : -> Pgm.
  var E : Exp .

  *** uses 'X 'Y : Var
  *** for all e : Int
  *** pre(S,E,e)  = (S[[E]]) is e
  *** post(S,E,e) = (S[['X]]) is e and (S[['Y]]) is e

  eq set(E) = 'X := E ; 'Y := E .
endo
```

12. Beispiel set(E) (2/2)

- Variante 1
  ```
  eq set(E) = 'X := E ; 'Y := E .
  red s ; set('X+1) [['Y]] .
  ```
  $\rightarrow$ `Result: s[['X]] + 2`
- Variante 2
  ```
  eq set(E) = 'E := E ; 'X := 'E ; 'Y := 'E .
  red s ; set('X+1) [['Y]] .
  ```
  $\rightarrow$ `Result: s[['X]] + 1`

13. Pascal-Ansatz

```
program setdemo;
  var a,b,x,y : int;
  begin
    set(a+b);
  end.
```

```
procedure set(e : int);
  var t : int;
  begin
    x := e ; y :=e ;
  end;
```

14. set(E) mit setbody (1/2)

```
obj SETBODY is pr SEM .

  op setbody : Int -> Pgm .
  var I : Int .

  *** uses 'X 'Y : var
  *** pre(S,I)  = true
  *** post(S,I) = S[['X]] is I and S['Y]] is I

  eq setbody(I) = 'X := I ; 'Y := I .

endo
```

15. set(E) mit setbody (2/2)

```
obj SETEXP is pr SETBODY .

  op set : Exp -> Pgm .
  var E : Exp .

  var S : Store .
  eq S ; set(E) = S ; setbody(S[[E]]) .
endo
```

16. Prozeduren mit `Exp`-Parametern: Spezifikation (1/2)

```
obj PROCBODY is pr SEM .
  op procbody : Var^K Arvar^L Int^M -> Pgm .
  vars X^ : Var .
  vars AV^ : Arvar .
  vars I^ : Int .
    *** uses Y^ : Var  *** uses A^ : Arvar
    *** for all e^ : Int
    *** for all a^ : Array
    *** pre(S,X^,AV^,I^,e^,a^) = ...
    *** post(S,X^,AV^,I^,e^,a^) = ...
  eq procbody(X^,AV^,I^) = <body> .
endo
```

17. Prozeduren mit `Exp`-Parametern: Spezifikation (2/2)

```
obj PROC is pr PROCBODY .

  op proc : Var^K Arvar^L Exp^M -> Pgm .
  vars X^ : Var .
  vars AV^ : Arvar .
  vars E^ : Exp .
  var S : Store ;
```

```
  eq S ; proc(X^,AV^,E^) = S ; procbody(X^,AV^,S[[E^]]) .

endo
```

18. Prozeduren mit `Exp`-Parametern: Verifikation

Es gelte: $pre(\mathtt{s}, \vec{X}, \vec{AV}, \vec{I}, \vec{e}, \vec{a})$
Die Definition der Prozedur `proc` heißt *korrekt*, wenn:

- `proc(`$\vec{X}$`,`$\vec{AV}$`,`$\vec{I}$`)` $\downarrow$ `s`
- $post($`s ; proc(`$\vec{X}$`,`$\vec{AV}$`,`$\vec{I}$`)`$, \vec{e}, \vec{a})$
- Für alle `V` (vom Typ `Var` oder `Arvar`) nicht in der Liste $\vec{Y}$ oder $\vec{A}$:
  `s ; proc(`$\vec{X}$`,`$\vec{AV}$`,`$\vec{I}$`) [[V]] = s [[V]].`
- Die Variablen aus $\vec{X}$, $\vec{Y}$, $\vec{V}$, $\vec{AV}$ sind paarweise verschieden.

19. Zusammenfassung - Nichtrekursive Prozeduren

- Grundsätzliche Spezifikation und Verifikation von Prozeduren
- Prozeduren als Makros
- Auswerung von Parametern
- Auswerung von Parametern

20. OBJ

```
***
*** Seminar-Beispiele.obj
***
*** OBJ-Beispiele zum Seminarvortrag: Prozeduren
*** entnommen aus: Joseph A. Goguen, Grant Malcom: Algebraic Semantics of Im-
perative Programs, MIT Press, 1996
***

*** -------------------------------------------------------------------
------
***
*** Beispiel 1: swap/swap
***

open SEM .

op swap : -> Pgm .

var S : Store .

eq S ; swap [['X]] = S [['Y]] .
eq S ; swap [['Y]] = S [['X]] .
var V : Var .
cq S ; swap [[V]] = S [[V]] if (( V =/= 'T ) and ( V =/= 'X )) and ( V =/= 'Y ) .

op s : -> Store .
red s ; swap [['X]] . ***> erwartet: s[['Y]]
red s ; swap [['Y]] . ***> erwartet: s[['X]]
red (s ; swap) ; swap [['X]] . ***> erwartet: s[['X]]
red (s ; swap) ; swap [['Y]] . ***> erwartet: s[['Y]]
```

```
close


*** --------------------------------------------------------------------
------
***
*** Beispiel 2: swap(X1,X2)
***

obj SWAP is pr SEM .

  op swap : Var Var -> Pgm .
  vars X1 X2 : Var .

  *** uses 'T : Var
  *** for all e1 e2 : Int
  *** pre(S,X1,X2,e1,e2)  = (S[['X]]) is e1 and (S[['Y]]) is e2
  *** post(S,X1,X2,e1,e2) = (S[['Y]]) is e2 and (S[['X]]) is e1

  eq swap(X1,X2) = 'T := X1 ; X1 := X2 ; X2 := 'T .

endo

open SWAP .

ops x1 x2 : -> Var .
ops e1 e2 : -> Int .
op s : -> Store .

*** Vorbedingung pre(s,x1,x2,e1,e2) voraussetzen
eq s[[x1]] = e1 .
eq s[[x2]] = e2 .

*** Nachbedingung post(s ; swap8x1,x2) , x1,x2,e1,e2) zeigen ***
red s ; swap(x1,x2) [[x1]] . ***> erwartet: e2
red s ; swap(x1,x2) [[x2]] . ***> erwartet: e1

close


*** --------------------------------------------------------------------
------
***
*** Beispiel 3: set(E)
***

obj SETEXP is pr SEM .
 op set : -> Pgm.
  var E : Exp .

  *** uses 'X 'Y : Var
  *** for all e : Int
  *** pre(S,E,e)  = (S[[E]]) is e
  *** post(S,E,e) = (S[['X]]) is e and (S[['Y]]) is e
```

```
  eq set(E) = 'X := E ; 'Y := E .
endo

open SETEXP
red s ; set('X+1) [['Y]] .
close


obj SETEXP2 is pr SEM .
 op set : -> Pgm.
  var E : Exp .

  *** uses 'X 'Y 'E: Var
  *** for all e : Int
  *** pre(S,E,e)  = (S[[E]]) is e
  *** post(S,E,e) = (S[['X]]) is e and (S[['Y]]) is e

  eq set(E) = 'E := E ; 'X := 'E ; 'Y := 'E .
 endo

open SETEXP
red s ; set('X+1) [['Y]] .
close
```

KAPITEL 5

Rekursive Prozeduren

Referent: Jens Brandt Datum: 29.05.01

1. Inhaltsübersicht

- Einführung
- Prozeduren ohne Parameter
- Prozeduren mit `Var`-Parametern

2. Einführung

```
program idpdemo;
  var x : int;
  begin
    x := 5;
    idp;
  end.

procedure idp;
  begin
    if ( x<>0 )
      begin x := x - 1; idp; x := x + 1; end;
  end;
```

3. Beispiel idp

```
obj IDP is pr SEM .
  op idp : -> Pgm .
  *** uses 'X : Int
  *** for all e : Int
  *** pre(S,e)  = (S[['X]]) is e and 0 <= e
  *** post(S,e) = (S[['X]]) is e
  *** M(S) = S[['X]]

  eq idp = if 'X is 0 then skip
           else 'X := 'X - 1 ; idp ; 'X := 'X + 1 fi .
endo
```

4. Parameterlose rekursive Prozeduren - Spezifikation

```
  obj PROC is pr SEM .
      op proc : -> Pgm .
        *** uses Y^ : Var
        *** uses A^ : Arvar
        *** for all e^ : Int
        *** for all a^ : Array
        *** pre(S,e^,a^) = ...
        *** post(S,e^,a^) = ...
```

```
        *** M(S) : Store -> Int = ...

      eq proc = <body> .
  endo
```

5. Wohlfundierte Induktion (1/2)

Wohlfundiertheit:

Jede nichtleere Menge natürlicher Zahlen hat ein kleinstes Element.

Wohlfundierte Induktion:

Ein Prädikat P auf den natürlichen Zahlen gilt für alle diese, falls:
$(\forall x \in \mathbb{N})\,[(\forall y \in \mathbb{N})\, y < x \Rightarrow P(y)] \Rightarrow P(x)$

6. Wohlfundierte Induktion (2/2)

$(\forall x \in \mathbb{N})\,[(\forall y \in \mathbb{N})\, y < x \Rightarrow P(y)] \Rightarrow P(x)$

- $R := \{r \mid \texttt{not}\,(P(r))\,\}$
- $r_0 := \min(R)$
- $P(y)$ gilt für alle y mit $y < r_0$
- $\Rightarrow P(r_0) \rightsquigarrow$ Widerspruch

7. Parameterlose rekursive Prozeduren - Verifikation (1/4)

- $(\forall \vec{e} : \texttt{Int})(\forall \vec{a} : \texttt{Array})(\forall S : \texttt{Store})\ \mathit{pre}(\texttt{S}, \vec{e}, \vec{a})\ \wedge\ IV$
  $\Rightarrow \mathit{post}(\texttt{S ; proc}, \vec{e}, \vec{a}) \wedge \texttt{proc} \downarrow S$
- mit $IV =$
  $(\forall \vec{e_2} : \texttt{Int})(\forall \vec{a_2} : \texttt{Array})(\forall S_2 : \texttt{Store})\ \mathit{pre}(S_2, \vec{e_2}, \vec{a_2})\ \wedge\ (M(S_2) < M(S))$
  $\Rightarrow \mathit{post}(S_2\texttt{;proc}, \vec{e_2}, \vec{a_2}) \wedge \texttt{proc} \downarrow S_2$

8. Parameterlose rekursive Prozeduren - Verifikation (2/4)

- $(\forall \vec{e} : \texttt{Int})(\forall \vec{a} : \texttt{Array})(\forall S : \texttt{Store})\ \mathit{pre}(S, \vec{e}, \vec{a})$
  $\Rightarrow M(S) \geq 0$
- <body> verändert nur die Variablen aus $\vec{Y}$ und $\vec{A}$.

9. Verifikation idp (1/4)

```
obj IDP is pr SEM .
  op idp : -> Pgm .
  *** uses 'X : Int
  *** for all e : Int
  *** pre(S,e)  = (S[['X]]) is e and 0 <= e
  *** post(S,e) = (S[['X]]) is e
  *** M(S) = S[['X]]

  eq idp = if 'X is 0 then skip
           else 'X := 'X - 1 ; idp ; 'X := 'X + 1 fi .
endo
```

10. Verifikation idp (2/4)

- $(\forall \vec{e} : \texttt{Int})(\forall \vec{a} : \texttt{Array})(\forall S : \texttt{Store})\quad \mathit{pre}(S, \vec{e}, \vec{a}) \Rightarrow M(S) \geq 0 \quad \surd$
- `if 'X is 0 then skip else 'X := 'X - 1 ; idp ; 'X := 'X + 1 fi .`
  Nur `'X` wird benutzt. $\surd$
- $(\forall e : \texttt{Int})(\forall S : \texttt{Store})$
  $(\texttt{(S[['X]]) is e and 0 <= e}) \wedge IV \Rightarrow (\texttt{S ; idp [['X]] is e}) \wedge \texttt{idp} \downarrow S$
  $IV = (\forall e_2 : \texttt{Int})(\forall S_2 : \texttt{Store})(S_2\texttt{[['X]] is } e_2 \texttt{ and 0 <= } e_2) \wedge (e_2 \leq e)$
  $\Rightarrow (S_2 \texttt{ ; idp [['X]] = } S_2\texttt{[[X]]}) \wedge \texttt{idp} \downarrow S_2$

11. Verifikation idp (3/4)

Fall 1:: $e = 0$

`S ; idp [['X]] is e = S ; skip [['X]] is e = S [['X]] is e` $\surd$

Fall 2:: $e > 0$

`(0 <=` S_2`[['X]] < e) = (0 <= S[['X]] - 1 < e) = (0 <= e-1 < e)` $\surd$

$\to S_2$`; idp [['X]] =` S_2`[['X]] and idp` $\downarrow S_2$

$\to S$`; 'X := 'X -1 ; idp [['X]] = e-1 and idp` $\downarrow$ `(S ; 'X :='X -1)`

12. Verifikation idp (4/4)

Fall 2 (Fortsetzung):: $e < 0$

```
     S ; idp [['X]] is e
=  S ; 'X:= 'X - 1 ; idp; 'X := X + 1 ; [['X]] is e
=  S ; 'X:= 'X - 1 ; idp; [['X]] + 1 is e  =  (e-1) + 1 is e √

     ( idp ; 'X := 'X + 1 )↓( S ; 'X := 'X - 1 )
=('X := 'X - 1 ; idp ; 'X := 'X + 1)↓S.√
```

13. Rekursive Prozeduren mit `Var`-Parametern - Spezifikation

```
obj PROC is pr SEM .
  op proc : Var^K Arvar^L -> Pgm .
  vars X^ : Var .
  vars AV^ : Arvar .
    *** uses Y^ : Var  *** uses A^ : Arvar
    *** for all e^ : Int
    *** for all a^ : Array
    *** pre(S,X^,AV^,e^,a^) = ...
    *** post(S,X^,AV^,e^,a^) = ...
    *** M(S,X^,AV^) = ...
  eq proc(X^,AV^) = <body> .
endo
```

14. Rekursive Prozeduren mit `Var`-Parametern - Verifikation

- $(\forall \vec{X} : \texttt{Var})(\forall \vec{AV} : \texttt{Arvar})(\forall \vec{e} : \texttt{Int})(\forall \vec{a} : \texttt{Array})(\forall S : \texttt{Store})$
  $pre(S, \vec{X}, \vec{AV}, \vec{e}, \vec{a}) \Rightarrow M(S, \vec{X}, \vec{AV}) \geq 0$
- <body> verändert nur die Variablen aus $\vec{Y}$ und $\vec{A}$.
- $(\forall \vec{X} : \texttt{Var})(\forall \vec{AV} : \texttt{Arvar})(\forall \vec{e} : \texttt{Int})(\forall \vec{a} : \texttt{Array})(\forall S : \texttt{Store}) pre(S, \vec{X}, \vec{AV}, \vec{e}, \vec{a})$
  $\wedge\ IV \Rightarrow post(\texttt{S;proc}(\vec{X}, \vec{AV}), \vec{X_2}, \vec{AV_2}, \vec{e}, \vec{a}) \wedge \texttt{proc}(\vec{X}, \vec{AV}) \downarrow S$
  mit $IV = (\forall \vec{X_2} : \texttt{Var})(\forall \vec{AV_2} : \texttt{Arvar})(\forall \vec{e_2} : \texttt{Int})(\forall \vec{a_2} : \texttt{Array})(\forall S_2 : \texttt{Store})$
  $pre(S_2, \vec{X_2}, \vec{AV_2}, \vec{e_2}, \vec{a_2})\ \wedge\ M(S_2, \vec{X_2}, \vec{AV_2}) < M(S, \vec{X}, \vec{AV})$
  $\Rightarrow post(S_2; \texttt{proc}(\vec{X}, \vec{AV}), \vec{X_2}, \vec{AV_2}, \vec{e_2}, \vec{a_2}) \wedge \texttt{proc}(\vec{X_2}, \vec{AV_2}) \downarrow S_2$

15. Verifikation lrep (1/5)

```
obj LREP is pr SEM .

   op lrep : Var Var -> Pgm .
   vars X Y : Var .

   *** uses 'A 'B : Var
   *** for all x y : Int
   *** pre(S,X,Y,x,y) = S[[X]] is x and 0 < x and S[[Y]] is y and 0 < y
   *** post(S,x,y) = S[['A]] * x - S[['B]] * y = gcd(x,y)
   *** M(S,X,Y) = S [[X + Y]]
```

16. Verifikation lrep (2/5)

```
   eq lrep(X,Y) =
        if X is Y then 'A := 1 ; 'B := 0
        else if X < Y
          then Y := Y - X ; lrep(X,Y); 'A := 'A + 'B
          else X := X - Y ; lrep(X,Y); 'B := 'A + 'B
          fi
        fi .
endo
```

17. Verifikation lrep (3/5)

Fall 1:: $x = y$

```
post( S ; 'A :=1 ; 'B := 0, x,y)
= (S ; 'A := 1 ; 'B := 0 [['A]]) * x
- (S ; 'A := 1 ; 'B := 0 [['B]]) * y is gcd(x,y)
= 1 * x - 0 * y is gcd(x,x)
```
$\surd$

18. Verifikation lrep (4/5)

Fall 2:: $x < y$

post(S ; Y := Y - X ; lrep(X,Y) ; 'A := 'A + 'B, x,y)

$($0 < S_2[[X]] $) = ($0 <= x$)$ $\surd$

$($0 < S_2[[Y]] $) = ($0 <= y$)$ $\surd$

$M(S_2, X_2, Y_2) < M(S, X, Y)$ $\surd$

$\rightarrow S_3$[['A]]*(S_2[[X]])-S_3[['B]]*(S_2[[Y]]) = gcd(S_2[[X]],S_2[[Y]])

mit S_3; = S_2 ; lrep(X,Y)

19. Verifikation lrep (5/5)

Fall 2 (Fortsetzung):: $x < y$

S_3[['A]]*(S_2[[X]])-S_3[['B]]*(S_2[[Y]]) = gcd(S_2[[X]],S_2[[Y]])

$=S_3$[['A]] * x - S_3[['B]] * (y-x) = gcd(x,y-x)

= (S_3[['A]]+S_3[['B]]) * x - S_3[['B]] * y = gcd(x,y)

= (S_3; A := 'A +'B [['A]]) * x

- S_3; A := 'A +'B [['B]] * y = gcd(x,y) $\surd$

20. Zusammenfassung - Rekursive Prozeduren in OBJ

$\rightarrow$ *Spezifikation*
- Bewertungsfunktion

$\rightarrow$ *Verifikation*
- Wohlfundierte Induktion
  - Korrektheit
  - Terminierung

21. OBJ

```
*** ------------------------------------------------------------------
------
***
*** Beispiel 4: idp
***

th IDPPROOF is pr SEM .
```

```
        op idp : -> Pgm .
        let body =      if 'X is 0
                                then skip
                                else 'X := 'X - 1 ; idp ; 'X  := 'X + 1
                                fi .

        op s : -> Store .
        op x : -> Int .


*** pre(s,x) voraussetzen
        eq s[['X]] = x .
        eq 0 <= x = true .


*** IV annehmen **
        op idp-result : Store -> Store .
        var S : Store .


*** idp gibt einen Store zurück, falls
*** pre(S,S [['X]]) and M(s) < M(s)

        cq S ; idp = idp-result(S)
                if 0 <= (S[['X]]) and (S[['X]]) < (s[['X]] ) .

*** idp erfüllt die Nachbedingung, falls
*** pre(S,S[['X]]) and M(s) < M(s)

        cq (idp-result(S))[['X]] = S [['X]]
                if 0 <= (S [['X]]) and (S [['X]]) < (s [['X]]) .
endth


***
*** Fall 0 = x
***

open IDPPROOF .
        eq      x = 0 .
        red (s ; body [['X]]) is x .
close



***
*** Fall 0 < x
***

open IDPPROOF .
        eq 0 < x = true .
        red (s ; body [['X]]) is x .
close
```

```
*** ----------------------------------------------------------------------
------
***
*** Beispiel 5: lrep
***

obj LREP is pr SEM .

        op lrep : Var Var -> Pgm .
        vars X Y : Var .

        *** uses 'A 'B : Var
        *** for all x y : Int
        *** pre(S,X,Y,x,y) = S[[X]] is x and 0 < x and S[[Y]] is y and 0 < y
        *** post(S,x,y) = S [[X + Y]]

        eq lrep(X,Y) =
                if X is Y
                then 'A := 1 ; 'B := 0
                else if X < Y
                         then Y := Y - X ; lrep(X,Y); 'A := 'A + 'B
                         else X := X - Y ; lrep(X,Y); 'B := 'A + 'B
                         fi
                fi .
endo

*** Ausführen

open LREP .

ops x1 x2 : -> Var .
ops e1 e2 : -> Int .
op s : -> Store .

eq s[[x1]] = 15 .
eq s[[x2]] = 9 .

red s ; lrep(x1,x2) .

close
```

Literaturverzeichnis

[GM96] Joseph A. Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. Foundations of Computing. MIT Press, 1996.

[Par93] Ian Parberry. How to present a paper in theoretical computer science. Technical report, Department of Computer Sciences, University of North Texas, 1993.