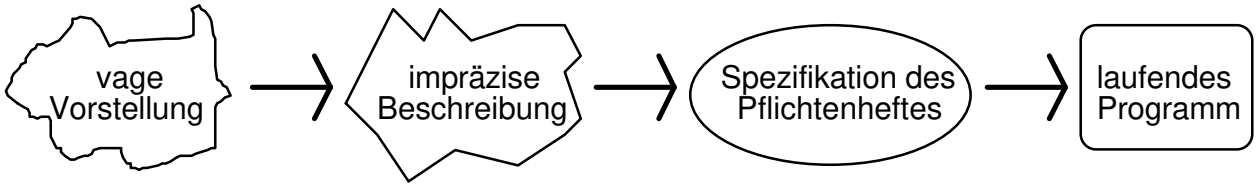
Einführung
in die
Konstruktion fehlerfreier Software

Mitschrift von
Josefine Schuhmann
und
Frank Pilhofer

Wintersemester 1994/1995

Einführung in die Konstuktion fehlerfreier Software



Diese Vorlesung befaßt sich mit den letzten beiden Abschnitten der Konstuktion von Software, der Spezifikation des Pflichtenheftes und dem Entwurf des laufenden Programms. In vielen Fällen kommt es jedoch vor, daß das laufende Programm im Prinzip ein "Abfallprodukt" einer korrekten Spezifikation ist.

Diese Vorlesung

- mathematische, theoretische Grundlagen
  - Programmvariablen, Datenumgebung
  - Programmanweisungen und ihre Zusammensetzungen als Funktionen auf der Menge der Datenumgebungen
  - Vorbedingungen und Nachbedingungen (Spezifikation)
  - Beweisregeln (Beziehungen zwischen Vor- und Nachbedingungen)
- ingenieurmäßige Konstruktion
  - Anwendung der Beweisregeln
  - Analyse und Verifikation
  - Bedeutung für die Konstruktionsaufgabe
  - nützliche Richtlinien für die Konstruktion
- Zusammenfassung
- Zukunft als Software-Ingenieure

Mathematisch-theoretische Grundlagen

Definition: Eine Programmvariable ist ein Tripel (Name,Menge,Element) mit Element∈ Menge; Element≈"Wert". Beispiel: (x, $\Re$, 0.5)

Definition: Eine Datenumgebung ist eine Folge von Programmvariablen. D: Menge aller Datenumgebungen.

Datenumgebungen

$d^0:\left[\left(N_0^0,M_0^0,W_0^0\right),\left(N_1^0,M_1^0,W_1^0\right),\ldots\right]$, $d^1:\left[\left(N_0^1,M_0^1,W_0^1\right),\left(N_1^1,M_1^1,W_1^1\right),\ldots\right]$ sind Datenumgebungen

- $d^a = d^b \Leftrightarrow \forall i:\left(N_i^a = N_i^b \wedge M_i^a = M_i^b \wedge W_i^a = W_i^b\right)$
- $d^a \overset{strukturell}{=} d^b \quad \Leftrightarrow \quad \forall i:\left(N_i^a = N_i^b \wedge M_i^a = M_i^b\right)$
- $x \in d^j \quad \Leftrightarrow \quad \exists i: N_i^j = x$

Man beachte, daß zwei Datenumgebung nur gleich sind, wenn sie die gleichen Daten in der gleichen Reihenfolge enthalten. Zwei Datenumgebungen sind strukturell gleich, wenn sie die gleichen Variablen enthalten, sich die Werte der Variablen jedoch unterscheiden.

Schreibweisen

- $F(x) \quad \Leftrightarrow \quad F.x$
- $F(x,y) \quad \Leftrightarrow \quad F.(x,y)$
- (*Operator Variable*: *Ausdruck 1* : *Ausdruck 2*) ⇔ Führe *Ausdruck 2* mit allen durch *Ausdruck 1* erlaubten Kombinationen mit *Variable* durch und verknüpfe mit *Operator*. Operatoren sind z.B. $\forall, \exists, \cdot, \cap$ etc.

  Beispiel: $\sum_{i=1}^{n} x(i) = \sum_{i=1}^{n} x.i = (+i : i \in Z, 1 \le i \le n : x.i)$

Später wird oft von Mengen von Datenumgebungen die Rede sein. Da diese natürlich schlecht aufzählbar (da i.a. überabzählbar) sind, wird die Schreibweise $\{B\}$ verwendet, womit alle Datenumgebungen gemeint sind, in denen B gilt: $\{B\} = \{d \in D | B.d = wahr\}$. Diese Bedingung B muß natürlich keinesfalls atomar sein. Beispiel: $\{x > 5 \wedge z < 5\}$.

Verwirrend sind auf den ersten Blick auch Schreibweisen wie $B \cap V$, wobei B ein logischer Ausdruck und V eine Menge von Datenumgebungen ist. Auch hier drückt B wieder im obigen Sinne eine Menge von Datenumgebungen aus. Gemeint ist: $B \cap V = \{d \in V | B.d = wahr\}$.

Definition: Der Wert einer (gewöhnlichen) Variable in einer Datenumgebung ist der Wert der ersten Programmvariable mit dem fraglichen Namen.

"Variablenname" ist eine Funktion $D \rightarrow Wert$ (z.B. *x.d*). Ab und zu wird auch "Datenumgebung" als eine Funktion $Name \rightarrow Wert$ (z.B. *d.x*) angesehen.

Definition: Wert eines Ausdrucks A in einer Datenumgebung d:

- Jeder Variablenname wird durch den Wert der Variable in d ersetzt

- Der Wert des sich daraus ergebenden Ausdrucks wird ermittelt
- "Ausdruck" ist wiederum eine Funktion $D \to Wert$ $(A.d)$

Beispiel: Sei $d:[(x,Z,1),(y,Z,2)]$ und $A = x + y^2$. Die Funktion A ist traditionell eine Funktion $A: Z^2 \to Z$. Hier wird sie jedoch zu einer Funktion $A: D \to Z$. Es ist $A.d = 5$.

Definition: Der Wert einer Feldvariable $x(A)$ ist $x(A.d).d$, d.h. zunächst wird der Ausdruck *A* in der Datenumgebung *d* ausgewertet. Man beachte, daß diese Definition nicht vorschreibt, von welchem Typ der Feldindex sein muß. Der Feldindex ist in gewissem Sinne Teil des Namens der Feldvariable.

Definition: Eine Programmanweisung ist eine Abbildung $D \to D$. Diese Abbildung ist im allgemeinen partiell, d.h. die Anweisung ist nur auf bestimmte Eingaben definiert.

Definition: Die Auswertung einer Zuweisung $x := A$ (d.h. einer speziellen Programmanweisung) angewendet auf die Datenumgebung *d* wird geschrieben als $(x := A).d$ und läuft wie folgt ab:

1.) Der Wert von *A* in *d* wird ermittelt (*A.d*).

2.) Der Wert der ersten Variablen in *d* mit dem Namen *x* wird durch *A.d* ersetzt.

d.h.: $d' = (x := A).d$ mit $\forall i: N_i' = N_i$, $\forall i: M_i' = M_i$ und $\forall i \neq j: W_i' = W_i$ und $W_j' = A.d$, wobei *j* der Index der ersten Variablen in *d* mit dem Namen *x* ist. Im Klartext: die Zuweisung verändert die Datenumgebung nur in genau einem Wert.

Auch Zuweisungen sind nicht immer definiert, eine Zuweisung kann durchaus eine partielle Funktion sein. Damit eine Zuweisung $x := A$ in einer Datenumgebung *d* definiert ist, muß

- der Ausdruck *A* muß in *d* definiert sein: $d \in \mathrm{Def}_A$
- eine Variable mit dem Namen *x* in *d* existieren: $\exists i: N_i = x$
- *A.d* vom Typ des ersten *x* in *d* sein: $A.d \in W_i \wedge N_i = x$

Insgesamt: $\mathrm{Def}_{x:=A} = \left\{ d^j \in D \middle| d^j \in \mathrm{Def}_A \wedge \exists i: \left(N_i^j = x \wedge \left(\forall k: \left(k \geq i \vee N_k^j \neq x \right) \right) \wedge A.d \in W_i \right) \right\}$

In dieser Beschreibung wurde zusätzlich noch berücksichtigt, daß die erste Variable des Namens *x* verwendet wird "(entweder ist der Index *k* einer Variablen größer als *i* oder der Name der Variablen *k* ist ungleich *x*)". Durch eine Zuweisung kann der Typ (die Menge) einer Variable nicht geändert werden.

Diese Definition einer Zuweisung verbietet Ausdrücke mit Nebeneffekten, da sich bei einer Zuweisung nur der Wert der einen Variable ändern darf. Diese Bedingung und Beschränkung wird im folgenden Korrektheitsbeweise vereinfachen. Die Ergebnisse, die sich im folgenden ergeben, lassen sich nicht immer auf Programme (Programmanweisungen) mit Nebeneffekten übertragen.

Die Zuweisung zu einer Feldvariable wird geschrieben als $\big(x(A_1):=A_2\big).d=\big(x(A_1.d):=A_2\big).d$. Wiederum wird zuerst der Feldindex in der Datenumgebung berechnet und dann als Teil des Variablennamens interpretiert.

Es gibt auch die Mehrfachzuweisung $\big((x,y,z):=(A_1,A_2,A_3)\big).d$. Sie wird quasiparallel ausgeführt:

- $\big((x,y,z):=(A_1,A_2,A_3)\big).d=(x:=A_1).d;(y:=A_2).d;(z:=A_3).d$

Alle Zuweisungen greifen also auf die gleiche Datenumgebung zu. Bei der Auswertung von A_2 besitzt x z.B. noch den Wert, den x vor der ersten Zuweisung hatte. Probleme entstehen, wenn die gleiche Variable mehrfach auf der linken Seite der Mehrfachzuweisung auftaucht. Es erweist sich als praktisch, diese Art der Mehrfachzuweisung zu gestatten, solange der Variablen mehrmals der *gleiche Wert* zugewiesen wird.

Die Mehrfachzuweisung ist nicht definiert, wenn der gleichen Variable verschiedene Werte zugewiesen werden sollen, oder wenn ein einzelner Ausdruck nicht definiert ist. Der Definitionsbereich der mehrfachen Zuweisung ist daher formal:

$$\mathrm{Def}_{(x_1\ldots x_n):=(A_1\ldots A_n)}=\left\{d\in\bigcap_{i=1}^{n}\mathrm{Def}_{x_i:=A_i}\;\middle|\;\forall(i,j):\big("x_i"="x_j"\Rightarrow A_i.d=A_j.d\big)\right\}$$

Die Austauschanweisung $(x:=:y).d$ wird als $\big((x,y)=(y,x)\big).d$ interpretiert. Der Definitionsbereich der Austauschanweisung ergibt sich aus dem Definitionsbereich der Mehrfachzuweisung.

Definition: $\big(\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ endif}\big).d=\begin{cases}S_1.d & \text{falls } B.d \textit{ wahr}\\ S_2.d & \text{falls } B.d \textit{ falsch}\end{cases}$

Auch zur Definiertheit der if-Anweisung gelten einige Bedingungen:

- Die Bedingung muß definiert sein.
- Falls die Bedingung wahr ist, so muß der then-Zweig definiert sein.
- Falls die Bedingung falsch ist, so muß der else-Zweig definiert sein.

$$\mathrm{Def}_{\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ endif}}=\left\{d^j\in D\middle|d^j\in\mathrm{Def}_B\wedge\Big(\big(B.d^j\big)\wedge\big(d^j\in\mathrm{Def}_{S_1}\big)\vee\overline{\big(B.d^j\big)}\wedge\big(d^j\in\mathrm{Def}_{S_2}\big)\Big)\right\}$$

Definition: $(S_1,S_2).d=S_2(S_1.d)$. Dies ist die *Komposition* der Funktionen S_1 und S_2.

Die Komposition ist i.a. nicht kommutativ, d.h. $(S_1,S_2).d\neq(S_2,S_1).d$, sie ist jedoch assoziativ:

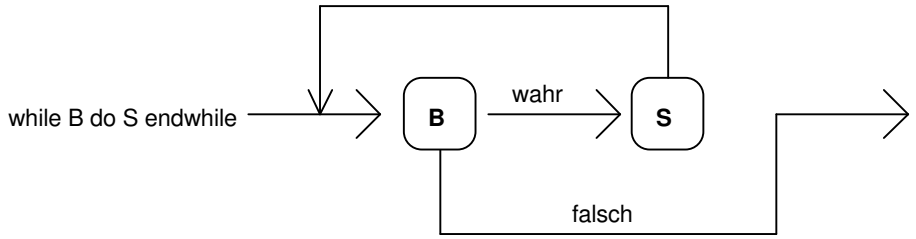
- $\big(S_1,(S_2,S_3)\big).d=(S_2,S_3)(S_1.d)=S_3\big(S_2(S_1.d)\big)=S_3\big((S_1,S_2).d\big)=\big((S_1,S_2),S_3\big).d$

Die Schreibweise $(S_1,S_2,S_3).d$, wobei die Funktionen sequentiell von links nach rechts abgearbeitet werden, ist daher eindeutig definierbar als $(S_1,S_2,S_3).d=\big(S_1,(S_2,S_3)\big).d$.

Definition: $S^n.d=\big(S,S^{n-1}\big).d$, und $S^1.d=S.d$, sowie $S^0.d=d$

Die Komposition zweier Anweisungen $(S_1, S_2).d$ ist nur dann definiert, falls $S_1.d = d'$ und $S_2.d'$ definiert sind. Also: $\text{Def}_{(S_1,S_2)} = \left\{ d^j \middle| d^j \in \text{Def}_{S_1} \wedge S_1.d^j \in \text{Def}_{S_2} \right\}$. Diese Erkenntnis läßt sich rekursiv auf Folgen beliebiger Länge verallgemeinern: $\text{Def}_{(S_1,\ldots,S_n)} = \left\{ d^j \middle| d^j \in \text{Def}_{S_1} \wedge S_1.d^j \in \text{Def}_{(S_2,\ldots,S_n)} \right\}$.

Definition: $(\text{while } B \text{ do } S \text{ endwhile}).d = \begin{cases} (S, \text{while } B \text{ do } S \text{ endwhile}).d & \text{falls } B.d \textit{ wahr} \\ d & \text{falls } B.d \textit{ falsch} \end{cases}$



Lemma: $(S, \text{while } B \text{ do } S \text{ endwhile}).d = (\text{while } B \text{ do } S \text{ endwhile}).(S.d)$

Bew: folgt direkt aus der Definition der Komposition.

Lemma: $(\text{while } B \text{ do } S \text{ endwhile}).d = S^n.d$, wobei $B.(S^n.d) = \textit{falsch}$ und $\wedge_{j=0}^{n-1} B.(S^j.d) = \textit{wahr}$.

Beweis: Zur Verdeutlichung definiere $d^i := S^i.d$, d.h. d^i ist die Datenumgebung nach *i*-maligem Durchlaufen der Schleife.
Beweis durch Induktion:

Verankerung: Ist $B.d^0$ falsch, so bricht die Schleife mit $(S, \text{while } B \text{ do } S \text{ endwhile}).d = d$ ab.
$\wedge_{j=0}^{-1} B.(S^j.d) = \textit{wahr}$, damit ist das Lemma erfüllt.
Im anderen Fall ist $B.d^0$ wahr, und es muß iteriert werden.

Annahme: $(\text{while } B \text{ do } S \text{ endwhile}).d = (\text{while } B \text{ do } S \text{ endwhile}).(S^n.d) \Leftrightarrow$
$\wedge_{j=0}^{n-1} B.(S^j.d) = \textit{wahr}$, d.h. insbesondere $B.d^i = \textit{wahr} \quad 1 \le i \le n-1$.

Schritt: Ist $B.d^n$ falsch, so ist $(\text{while } B \text{ do } S \text{ endwhile}).d = S^n.d$ mit - nach Annahme -
$\wedge_{j=0}^{n-1} B.(S^j.d) = \textit{wahr}$, d.h. das Lemma ist erfüllt.
Ist $B.d^n$ wahr, so muß die Schleife ein weiteres mal durchlaufen werden:
$(\text{while} B \text{ do } S \text{ endwhile}).d = (\text{while} B \text{ do } S \text{ endwhile}).(S.(S^n.d)) = (\text{while} B \text{ do } S \text{ endwhile}).(S^{n+1}.d)$
Nach Annahme und mit $B.d^n = \textit{wahr}$ gilt dann auch $\wedge_{j=0}^{n} B.(S^j.d) = \textit{wahr}$, damit ist das Lemma dann auch für n+1 erfüllt.

Damit die while-Anweisung definiert ist, muß
- die Bedingung in jeder Iteration definiert (und vom Typ Boolean) sein
- das Programmstück in jeder Iteration definiert sein und
- nach endlich vielen Iterationen muß die Bedingung falsch werden.

$$\text{Def}_{\text{w } B \text{ do } S \text{ end}} = \left\{ d \in D \middle| \exists n: \left(B.(S^n.d) \wedge \overline{\left(\bigvee_{j=0}^{n-1} B.(S^j.d) \right)} \wedge \left(\bigwedge_{j=0}^{n} S^j.d \in \text{Def}_B \right) \wedge \left(\bigwedge_{j=0}^{n-1} S^j.d \in \text{Def}_S \right) \right) \right\}$$

Definition: Konkatenation von Folgen: $[a,b]\&[c,d] \hat{=} [a,b,c,d]$. Die Konkatenation ist nicht kommutativ, wohl aber assoziativ. Man beachte, daß die Komposition Programmstücke aneinanderhängt; die Konkatenation hängt Mengen (z.B. Datenumgebungen) aneinander.

Definition: $(\text{declare}(x,M,A)).d = [(x,M,A.d)]\&d$. Die Declare-Funktion definiert eine Variable, indem sie die neue Variable der Datenumgebung hinzufügt. Die neue Variable wird am Anfang hinzugefügt, d.h. sie erhält Vorrang vor allen anderen Definitionen von *x*. Das Ergebnis der Funktion ist nur definiert, falls $A.d \in M$.

Definition: $(\text{release } x).d = d'$, wobei $j = (\min k: N_k = x: k)$ und

- $N_i' = N_i;$ $M_i' = M_i;$ $W_i' = W_i$ $i < j$
- $N_i' = N_{i+1};$ $M_i' = M_{i+1};$ $W_i' = W_{i+1}$ $i \geq j$

Die Funktion ist nur definiert, falls in der Datenumgebung eine Variable des Namens *x* existiert.

Mit anderen Worten, release entfernt die erste Variable mit dem Namen *x* aus der Datenumgebung. Mit den beiden Prozeduren declare und release kann eine Datenumgebung als Stack verwendet werden. Beispiel für beide Funktionen:

$$(\text{declare}(x,Z,2), \text{release } y).[(y,R,1.5),(y,Q,\tfrac{3}{4})] = (\text{release } y).[(x,Z,2),(y,R,1.5),(y,Q,\tfrac{3}{4})]$$
$$= [(x,Z,2),(y,Q,\tfrac{3}{4})]$$

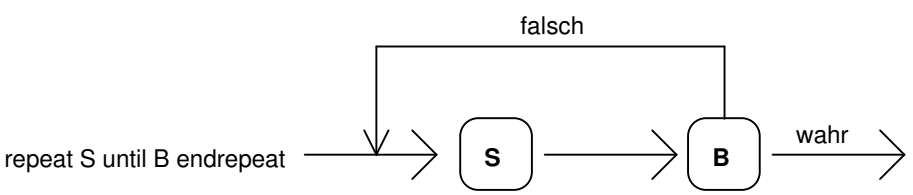
Die Deklaration ist nur definiert, wenn der Ausdruck in der Datenumgebung definiert ist und das Ergebnis von *A* mit dem Wertebereich von *x* übereinstimmt. Release ist nur dann definiert, wenn in der Datenumgebung eine Variable des gegebenen Namens existiert. Insgesamt:

$$\text{Def}_{\text{declare}(x,W,A)} = \{d \in D \mid d \in \text{Def}_A \wedge A.d \in W\} \text{ bzw. } \text{Def}_{\text{release } x} = \{d^j \in D \mid \exists i: N_i^j = x\}$$

Definition: $(\text{call } P).d = S_P.d$ wobei S_P die Prozedur mit Namen *P* ist. Das Ergebnis ist nur dann definiert, wenn $S_P.d$ definiert ist und *P* terminiert.

Definition: Null.*d*=*d*, wobei Null die Nullanweisung (leere Prozedur) ist.

Definition: $(\text{repeat } S \text{ until } B \text{ endrepeat}).d = \begin{cases} (S, \text{repeat } S \text{ until } B \text{ endrepeat}).d & \text{falls } B.(S.d) \textit{ falsch} \\ S.d & \text{falls } B.(S.d) \textit{ wahr} \end{cases}$



repeat ist keine Sonderform von while. Bei der repeat-Anweisung wird das Programmstück *S* mindestens einmal ausgeführt, während es bei while nicht ausgeführt werden muß. Jedes repeat-Konstrukt läßt sich jedoch als while umschreiben:

- $(\text{repeat } S \text{ until } B \text{ endrepeat}).d = (S, \text{while } B \text{ do } S \text{ endwhile }).d$

Zusammenfassung der Anweisungen:

- Atomare Anweisungen
  - Zuweisung
  - declare
  - release
  - Nullanweisung
- Zusammengesetzte Anweisungen
  - if
  - while, repeat und ähnliche Schleifenkonstrukte
  - Folge von Anweisungen
  - call (Prozeduraufruf)

Definition: Eine Ausführungsgeschichte ist eine Folge von Datenumgebungen. Interpretation: Eine Ausführungsgeschichte eines Programmes ist die Folge von Datenumgebungen, die bei der Ausführung der Einzelanweisungen des Programmes erzeugt werden (die Historie).

Beispiel: $\big((\text{declare}(x, Z, y-2)), x := x+2, x := y\big).\big[(y, Z, 4)\big]$ erzeugt die Ausführungsgeschichte

$$\big[\big[(y, Z, 4)\big], \big[(x, Z, 2), (y, Z, 4)\big], \big[(x, Z, 4), (y, Z, 4)\big], \big[(x, Z, 4), (y, Z, 4)\big]\big]$$

Definition: Die Menge aller Ausführungsgeschichten ist $D^* = \Big(\bigcup_{n \in N} D^n\Big) \cup D^\infty$. Diese Definition erlaubt es, die Ausführungsgeschichte eines nicht terminierenden Programmes zu betrachten.

Definition: letzt.*F* ist das letzte Element in der Folge *F*. Ist *F* leer oder ist *F* unendlich, so ist die Funktion nicht definiert.

Wir werden jetzt unsere Schreibweise ändern. Bislang haben wir alle Funktionen und Anweisungen so definiert, daß sie auf einer Datenumgebung operieren. Nun werden die Funktionen so umdefiniert, daß sie ihre Werte aus dem letzten Element einer Ausführungsgeschichte erhalten und ihr Ergebnis (wieder eine Datenumgebung) an die Ausführungsgeschichte anhängen.

Schreibweise: *S* ist eine Funktion auf *D*. S^* ist die Funktion *S* in obigem Sinne angewandt auf D^*.

Definition: Falls *S* eine Basisanweisung ist, so ist $S^*.d^* = d^* \,\&\, \big[S.\big(\text{letzt}.d^*\big)\big]$ mit $d^* \in D^*$. Die Funktion ist nur definiert, wenn

- $|d^*| < \infty$ (ansonsten ist letzt nicht definiert)
- $S.\big(\text{letzt}.d^*\big)$ definiert ist

Beispiel: $\text{Null}^*.[[(x,Z,42)]] = [[(x,Z,42)],[(x,Z,42)]]$

Definition: $(S_1,S_2)^*.d^* = S_2^*.(S_1^*.d^*)$. Die Komposition ist nur definiert, wenn $|d^*| < \infty$; $S_1.(\text{letzt}.d^*)$ definiert und $S_2.(S_1.(\text{letzt}.d^*))$ definiert ist.

Definition: $(\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ endif})^*.d^* = \begin{cases} S_1^*.d^* & \text{falls } B.(\text{last}.d^*) \textit{ wahr} \\ S_2^*.d^* & \text{falls } B.(\text{last}.d^*) \textit{ falsch} \end{cases}$

Es gelten analoge Bedingungen wie oben und bei der ursprünglichen Definition von if.

Definition: $(\text{while } B \text{ do } S \text{ endwhile})^*.d^* = \begin{cases} (S, \text{while } B \text{ do } S \text{ endwhile})^*.d^* & \text{falls } B.(\text{last}.d^*) \textit{ wahr} \\ \text{Null}^*.d^* & \text{falls } B.(\text{last}.d^*) \textit{ falsch} \end{cases}$

Es gelten analoge Bedingungen wie oben und bei der ursprünglichen Definition von while.

Definition: $(\text{call } P)^*.d^* = S_P^*.d^*$ wobei S_P die Prozedur mit Namen P ist. Das Ergebnis ist nur dann definiert, wenn $S_P.(\text{last}.d^*)$ definiert ist und P mit dieser Eingabe terminiert.

Eingabe und Ausgabeanweisungen lassen sich als Funktionen auf speziellen (Feld-) Variablen verstehen. So läßt sich eine Anweisung wie `print x,y;` als `bildschirm(Zeile) := str(x)&str(y); Zeile:=Zeile+1` umschreiben. `seek#1, x; read#1,y;` wird zu `pos(1):=x; y:=dateisatz(1,pos(1));`.

Formale Parameterübergabe

In vielen Fällen werden bei Prozeduraufrufen *Parameter* übergeben, auf denen die Prozedur operieren soll. Für die Parameterübergabe gibt es zwei Möglichkeiten. Die eine ist *call-by-value*, die andere ist *call-by-reference*. Im ersten Fall erhält die aufgerufene Prozedur den Wert einer Variablen als Parameter, im zweiten Fall die Variable selber.

Als Beispiel definieren wir eine Funktion `procedure P(value f1, reference f2)`, wobei der erste Parameter by value, der zweite bei reference übergeben werden soll. Bei einem Aufruf der Funktion P mit den Parametern `a1` und `a2` wird nun eine neue Variable des Namens `f1` angelegt, die Typ und Wert von a1 erhält. `f2` hingegen wird als spezielle Variable vom Typ Referenz angelegt, die auf `a2` verweist. So operieren alle Zugriffe auf `f2` in Wirklichkeit auf `a2`.

Dieser Unterschied zwischen beiden Methoden ist fundamental und soll daher hier ausführlich an einem Beispiel demonstriert werden.

```
procedure P(value f1, reference f2)
     f1:=f1+f2;
     f2:=f1+f2;
end procedure
```

Sei die Datenumgebung $d = [(a1, Z, 3), (a2, Z, 2)]$. Wir rufen nun `call P(a1,a2)`. Für `f1` wird eine neue Ganzzahlvariable mit dem Wert 3 definiert. Für `f2` wird eine neue Variable vom Typ Referenz erzeugt. Beim Eintritt in die Prozedur `P` haben wir daher die Datenumgebung $d^1 = [(f2, \text{Ref}, "a2"), (f1, Z, 3), (a1, Z, 3), (a2, Z, 2)]$. Im ersten Befehl der Prozedur werden der Wert von `f1` und `f2` addiert, wobei beim Zugriff von `f2` festgestellt wird, daß es sich um eine Referenz auf `a2` handelt, und deshalb auf dessen Wert zurückgegriffen wird. Das Ergebnis wird der neu definierten Variablen `f2` zugewiesen. Nach Ausführung der Zuweisung erhalten wir $d^2 = [(f2, \text{Ref}, "a2"), (f1, Z, 5), (a1, Z, 3), (a2, Z, 2)]$. Der nächste Befehl läuft zunächst ähnlich ab. Die Werte von `f1` und `f2` alias `a2` werden addiert. Bei der Zuweisung des Ergebnisses an `f2` wird jedoch wieder festgestellt, das `f2` eine Referenz auf `a2` ist; deshalb wird das Ergebnis `a2` selber zugewiesen. Wir erhalten $d^3 = [(f2, \text{Ref}, "a2"), (f1, Z, 5), (a1, Z, 3), (a2, Z, 7)]$. Die Prozedur wird nun wieder verlassen, wobei die beim Eintritt in die Funktion definierten Variablen wieder gelöscht werden: $d^4 = [(a1, Z, 3), (a2, Z, 7)]$. Spätestens hier sollte der Unterschied zwischen der Übergabe *by-reference* und *by-value* auffallen. Änderungen auf Parameter, die *by-value* übergeben wurden, sind nur während der Ausführung der Unterprozedur sichtbar. Mit *by-reference* übergebenen Parametern kann eine Prozedur jedoch die Datenumgebung des aufrufenden Programmes ändern. Daher sollte die Übergabeart sorgfältig ausgewählt werden. *By-reference*-Übergaben sind, insbesondere bei "größeren" Variablen, meist weniger aufwendig, dafür werden Seiteneffekte ermöglicht.

Es sollte bemerkt werden, daß die oben nahegelegte Vorstellung, eine Referenz würde den expliziten Namen der referenzierten Variablen enthalten, falsch ist. Nehmen wir an, in `P` würde eine neue Variable `a2` definiert. Auch in diesem Fall würde `f2` noch auf die als Parameter übergebene Variable `a2` zeigen, und nicht auf das lokale `a2`, welches bei Referenzierung als Name den Vorrang hätte. Sicherer wäre in diesem Fall der Verweis mittels eines von hinten gezählten Index.

In nur wenigen Sprachen ist es möglich, die Form der Parameterübergabe selbst zu wählen. In anderen Sprachen, oder bei nicht expliziter Anweisung, werden Parameter immer by-reference (FORTRAN), immer by-value (C) oder je nach Typ verschieden übergeben. Teilweise gibt es sogar innerhalb der Sprachen auf verschiedenen Systemen Unterschiede. Es kann daher durchaus sein, daß eine Prozedur P auf System A korrekt arbeitet, auf System B jedoch nicht.

Es gibt auch noch Unterschiede bei der Behandlung von Ausdrücken, die *by-reference* übergeben werden (z.B. oben `call P(a1,a1+a2)`). In manchen Sprachen sind solche Aufrufe unzulässig, in anderen wird eine temporäre Variable mit dem Ergebnis des Ausdrucks definiert; die Prozedur erhält dann eine Referenz auf diese temporäre Variable.

Vorbedingungen und Nachbedingungen

Um Beweise bezüglich der Korrektheit einer Prozedur führen zu können, gibt es das Konzept der Vor- und Nachbedingungen. Die Denkweise läßt sich an einem einfachen wenn-dann-Konstrukt verdeutlichen. Wenn $x < 5$, dann ist $x^2 < 25$. In diesem Fall ist $x < 5$ eine Vorbedingung. Falls die Vorbedingung erfüllt ist, so ist die Nachbedingung, eine Aussage, erfüllt.

In Bezug auf Prozeduren ist die Nachbedingung eine Aussage über das partiell korrekte Verhalten der Prozedur; die Vorbedingung fordert eine bestimmte Struktur der Umgebung, z.B. der Eingabewerte, bezüglich derer korrektes Verhalten erreicht wird. Formal und allgemein:

Definition: Eine Teilmenge V von D ist eine Vorbedingung einer Nachbedingung P (P ist wiederum eine Teilmenge von D) bezüglich der Programmierung S, falls $S.d \in P$ für alle $d \in V$ und $d \in DEF_S$.

Man schreibt $\{V\}S\{P\}$, was soviel bedeutet wie $\left(\forall d : d \in V \wedge d \in S^{-1}.D : S.d \in P\right)$.

V ist in diesem Fall eine gewöhnliche Vorbedingung.

Lemma: $\{V\}S\{P\} \Leftrightarrow V \cap S^{-1}.D \subseteq S^{-1}.P$

Bew.: $\{V\}S\{P\} \quad \Leftrightarrow \quad \left(\forall d : d \in V \wedge d \in S^{-1}.D : S.d \in P\right)$

$\Leftrightarrow \quad \left(\forall d : d \in V \cap S^{-1}.D : d \in S^{-1}.P\right)$

$\Leftrightarrow \quad V \cap S^{-1}.D \subseteq S^{-1}.P$

Definition: V ist eine <u>strikte</u> Vorbedingung für P bezüglich S, wenn $\{V\}S\{P\}$ und $V \subseteq S^{-1}.D$.

Lemma: $\{V\}S\{P\}$ strikt $\Leftrightarrow V \subseteq S^{-1}.P$.

Bew.: $\{V\}S\{P\}$ strikt $\Leftrightarrow \quad \{V\}S\{P\}$ und $V \subseteq S^{-1}.D$

$\Leftrightarrow \quad V \cap S^{-1}.D \subseteq S^{-1}.P$ und $V \subseteq S^{-1}.D$

$\Leftrightarrow \quad V \subseteq S^{-1}.P$ und $V \subseteq S^{-1}.D$

Wegen $P \subseteq D$ folgt $S^{-1}.P \subseteq S^{-1}.D$, also

$\Leftrightarrow \quad V \subseteq S^{-1}.P$

Eine strikte Vorbedingung stellt die totale Korrektheit des Programmes S sicher.

Definition: Eine Teilmenge V von D heißt <u>umfassende</u> Vorbedingung einer Nachbedingung P, falls

- $\{V\}S\{P\}$ und
- $S^{-1}.P \subseteq V$

Lemma: $\{V\}S\{P\}$ umfassend $\Leftrightarrow V \cap S^{-1}.D = S^{-1}.P$.

Bew.: $\{V\}S\{P\}$ umf. $\Leftrightarrow \quad \{V\}S\{P\}$ und $S^{-1}.P \subseteq V$

$\Leftrightarrow \quad V \cap S^{-1}.D \subseteq S^{-1}.P$ und $S^{-1}.P \subseteq V$

Wieder folgt mit $S^{-1}.P \subseteq S^{-1}.D$

$\Leftrightarrow \quad V \cap S^{-1}.D \subseteq S^{-1}.P$ und $S^{-1}.P \subseteq V \cap S^{-1}.D$

$\Leftrightarrow \quad V \cap S^{-1}.D = S^{-1}.P$

Lemma: $\{V\}S\{P\}$ strikt und umfassend $\Leftrightarrow V = S^{-1}.P$.

Beweis: Sei $\{V\}S\{P\}$ strikt und umfassend

$\Leftrightarrow \quad V \subseteq S^{-1}.P$ und $S^{-1}.P \subseteq V$

$\Leftrightarrow \quad V = S^{-1}.P$

Zusammenfassung

◆ $\{V\}S\{P\}$	$\Leftrightarrow$	$\left(\forall d : d \in V \wedge d \in S^{-1}.D : S.d \in P\right)$	$\Leftrightarrow$	$V \cap S^{-1}.D \subseteq S^{-1}.P$
◆ $\{V\}S\{P\}$ strikt	$\Leftrightarrow$	$\{V\}S\{P\}$ und $V \subseteq S^{-1}.D$	$\Leftrightarrow$	$V \subseteq S^{-1}.P$
◆ $\{V\}S\{P\}$ umfassend	$\Leftrightarrow$	$\{V\}S\{P\}$ und $S^{-1}.P \subseteq V$	$\Leftrightarrow$	$V \cap S^{-1}.D = S^{-1}.P$
◆ $\{V\}S\{P\}$ strikt & umf.	$\Leftrightarrow$			$V = S^{-1}.P$

Es gibt genau eine strikte umfassende Vorbedingung. Diese ist Vereinigungsmenge aller strikten Vorbedingungen und Schnittmenge aller umfassenden Vorbedingungen.

Beweisregeln

Arten von Beweisaufgaben:
- Ermitteln der Vorbedingung
- Ermitteln des Programmstückes
- Ermitteln der Nachbedingung

Die dritte Art tritt selten auf. Üblicherweise wird eine Nachbedingung nicht ermittelt, sondern in einer Spezifikation vorausgesetzt. Das Finden eines Tripels (V, S, P) mit $\{V\}S\{P\}$ nennt man Korrektheitsbeweisführung.

Beweisregel B1: Falls $V \Rightarrow V'$, $\{V'\}S\{P'\}$ und $P' \Rightarrow P$, dann gilt $\{V\}S\{P\}$.

Beweis: $\{V\}S\{P\} \quad \Leftrightarrow \left(\forall d : d \in V \wedge d \in S^{-1}.D : S.d \in P\right)$

$\Leftarrow \left(\forall d : d \in V' \wedge d \in S^{-1}.D : S.d \in P'\right)$ und $V \subseteq V'$ und $P' \subseteq P$

$\Leftrightarrow \{V'\}S\{P'\}$ und $V \subseteq V'$ und $P' \subseteq P$

Beweis B1 arbeitet nach dem Überflußprinzip. Das gewünschte, $\{V\}S\{P\}$, zeigen wir dadurch, daß wir die Nachbedingung P selbst dann einschränken können, wenn wir die Vorbedingungen erleichtern. $V \subseteq V'$ heißt in diesem Falle, daß in V' *mehr Freiheitsgrade* existieren als in V.

Beweisregel DC1: Falls $\{V\}S\{P\}$ und $\{V'\}S\{P'\}$, dann gilt ebenfalls $\{V \text{ und } V'\}S\{P \text{ und } P'\}$.

Beweisregel DC2: Falls $\{V\}S\{P\}$ und $\{V'\}S\{P'\}$, dann gilt ebenfalls $\{V \text{ oder } V'\}S\{P \text{ oder } P'\}$.

Lemma: Sei $d' = (x := A).d$. Für alle d aus dem Bereich der Zuweisung gilt:
$x.d' = A.d$ und $\forall y \neq x : y.d' = y.d$.

Beweisregel Z1: Sei P ein gegebener Ausdruck über alle Variablennamen. Die Zuweisung $x := A$ sei auch gegeben. Bildet man den Ausdruck V, in dem der Variablenname x überall in P durch (A) ersetzt wird (geschrieben $P^x A$), dann ist, V eine umfassende Vorbedingung von P bzgl. $x := A$.

Satz: Beweisregel Z1 funktioniert. Zu zeigen: $\left\{P^x A\right\}x := A\{P\}$ umfassene Vorbedingung.
Beweis: OBdA sei $P(x, y)$ eine Funktion in zwei Variablen. Setze $V = P\big(A(x, y), y\big)$.
Es ist $\big(V = P\big(A(x, y), y\big)\big).d \Leftrightarrow V.d = P\big(A(x.d, y.d), y.d\big) = P(x.d', y.d')$ für alle

$d \in (x:=A)^{-1}.D.$

Betrachte: wahr $\Leftrightarrow$ $\left(\forall d : d \in V \cap (x:=A)^{-1}.D : d \in V\right)$

Nun ist $d \in V = P.((x:=A).d) \Leftrightarrow (x:=A).d \in P \Leftrightarrow d \in (x:=A)^{-1}.P$. Daher

$\Leftrightarrow$ $\left(\forall d : d \in V \cap (x:=A)^{-1}.D : (x:=A).d \in P\right)$

$\Leftrightarrow$ $V \cap Z^{-1}.D \subseteq Z^{-1}.P$

$\Leftrightarrow$ $\{V\}S\{P\}$ $\Rightarrow$ Vorbedingung gezeigt.

z.Z.: Vorbedingung ist umfassend. Betrachte

wahr $\Leftrightarrow$ $\left(\forall d : d \in (x:=A)^{-1}.P \cap (x:=A)^{-1}.D : d \in (x:=A)^{-1}.P\right)$

wie oben gezeigt ist $d \in V \Leftrightarrow d \in (x:=A)^{-1}.P$. Damit gilt

$\Leftrightarrow$ $Z^{-1}.P \cap Z^{-1}.D \subseteq V$

$\Leftrightarrow$ $Z^{-1}.P \subseteq V$ $\Rightarrow$ umfassend gezeigt

Zusammen ist also gezeigt, daß $\{P^x A\}x:=A\{P\}$ eine umfassende Vorbedingung ist.

Beispiel zur Ermittlung einer umfassenden Vorbedingung

- $\{?\}x(i):=1\{x(i)=x(j)\}$

Bezeichne nun ein einfach gestrichener Ausdruck den Wert dieses Ausdrucks vor Ausführung von $x(i):=1$, und ein zweifach gestrichener Ausdruck den Wert nach der Ausführung. Es gelten nach Definition der Feldzuweisung $x''(i')=1;\ x''(k)=x'(k)\ \ \forall k \neq i;\ j''=j'$ und $i''=i'$. Mit diesen Bedingungen kann die Nachbedingung umgeformt werden:

- $x''(i'')=x''(j'')$ $\Leftrightarrow$ $x''(i')=x''(j')$

$\Leftrightarrow$ $(x''(i')=x''(j'))$ und $(i'=j'$ oder $i' \neq j')$

$\Leftrightarrow$ $(x''(i')=x''(j')$ und $i'=j')$ oder $(x''(i')=x''(j')$ und $i' \neq j')$

$\Leftrightarrow$ $(1=1$ und $i'=j')$ oder $(1=x''(j')$ und $i' \neq j')$

$\Leftrightarrow$ $i'=j'$ oder $(1=x''(j')$ und $i' \neq j')$

$\Leftrightarrow$ $i'=j'$ oder $1=x''(j')$

Eine umfassende Vorbedingung ist daher $\{j=i$ oder $x(j)=1\}$.

Weitere Beweisregeln

Beweisregel Z2: Falls $V \Rightarrow P^x A$, dann gilt $\{V\}x:=A\{P\}$.

Beweis: $\{P^x A\}x:=A\{P\}$ ist nach (Z1) umfassend. $P \Rightarrow P$ gilt trivialerweise.

Mit $V \Rightarrow P^x A$; $\{P^x A\}x:=A\{P\}$ und $P \Rightarrow P$ gilt nach (B1) $\{V\}x:=A\{P\}$.

Beweisregel F1: Falls $\{V\}S_1\{P_1\}$ und $\{P_1\}S_2\{P\}$, dann gilt $\{V\}(S_1,S_2)\{P\}$.

Beweis: Sei $d \in V \cap \mathrm{Def}_{(S_1,S_2)}$. Es ist

$S_1.d \in P_1$ nach Definition von $\{V\}S_1\{P_1\}$. Damit ist

$S_2.(S_1.d) \in P$ nach Definition von $\{P_1\}S_2\{P\}$. Damit gilt auch

$(S_1,S_2).d \in P$.

Beweisregel F1': Falls $\{V\}S_1\{P_1\}$ umfassend und $\{P_1\}S_2\{P\}$ umfassend, dann ist $\{V\}(S_1,S_2)\{P\}$ ebenfalls umfassend.

Beweis: $\{V\}(S_1,S_2)\{P\}$ ist klar nach (F1). Zu zeigen: $(S_1,S_2)^{-1}.P \subseteq V$.

Es gilt nach Definition $S_1^{-1}.P_1 \subseteq V$ und $S_2^{-1}.P \subseteq P_1$. Daraus folgt $S_1^{-1}.(S_2^{-1}.P) \subseteq V$ und damit $(S_1,S_2)^{-1}.P \subseteq V$.

Beweisregel F1": Falls $\{V\}S_1\{P_1\}$ strikt und $\{P_1\}S_2\{P\}$ strikt, dann ist $\{V\}(S_1,S_2)\{P\}$ ebenfalls strikt. Zu zeigen: $V \subseteq (S_1,S_2)^{-1}.P$.

Bekannt ist nach Definition $V \subseteq S_1^{-1}.P_1$ und $P_1 \subseteq S_2^{-1}.P$.

Daraus folgt direkt $V \subseteq S_1^{-1}.(S_2^{-1}.P) = (S_1,S_2)^{-1}.P$.

Beweisregel IF1: Sei $S :=$ if B then S_1 else S_2 endif . Ist $\{V \text{ und } B\}S_1\{P\}$ und $\{V \text{ und nicht } B\}S_2\{P\}$, so gilt $\{V\}S\{P\}$.

Beweis: Sei $d \in V \cap \mathrm{Def}_S$.

Fall 1: $B.d = wahr: S.d = S_1.d; d \in B \cap V \Rightarrow S.d = S_1.d \in P$

Fall 2: $B.d = falsch: S.d = S_2.d; d \in \neg B \cap V \Rightarrow S.d = S_2.d \in P$

Damit gilt in allen Fällen $\{V\}S\{P\}$.

Beweisregel IF2: Sei wiederum $S :=$ if B then S_1 else S_2 endif . Falls $\{V_1\}S_1\{P\}$ und $\{V_2\}S_2\{P\}$, dann gilt $\{V_1 \text{ und } B \text{ oder } V_2 \text{ und nicht } B\}S\{P\}$.

Beweis: Sei $V = V_1 \text{ und } B \text{ oder } V_2 \text{ und nicht } B$.

$$\{V\}S\{P\} \overset{(IF1)}{\Leftarrow} \{V \text{ und } B\}S_1\{P\} \text{ und } \{V \text{ und nicht } B\}S_2\{P\}$$

$$\Leftrightarrow \left\{\begin{pmatrix} V_1 \text{ und } B \\ \text{oder} \\ V_2 \text{ und nicht } B \end{pmatrix} \text{und } B\right\}S_1\{P\} \text{ und } \left\{\begin{pmatrix} V_1 \text{ und } B \\ \text{oder} \\ V_2 \text{ und nicht } B \end{pmatrix} \text{und nicht } B\right\}S_2\{P\}$$

$$\Leftrightarrow \{V_1 \text{ und } B\}S_1\{P\} \text{und} \{V_2 \text{ und nicht } B\}S_2\{P\}.$$

Diese *Aussage* wird durch das weglassen von B bzw. nicht B verstärkt:

$$\overset{(B1)}{\Leftarrow} \{V_1\}S_1\{P\} \text{und} \{V_2\}S_2\{P\}.$$

Insgesamt liest sich die Zeile also als

$$\{V_1\}S_1\{P\} \text{und} \{V_2\}S_2\{P\} \Rightarrow \{V_1 \text{ und } B \text{ oder } V_2 \text{ und nicht } B\}S\{P\}.$$

Die Beweisregeln (IF1) und (IF2) gelten auch in umfassender (strikter) Form, d.h. wenn die Vorbedingungen V_1 und V_2 umfassend (strikt) sind, so ist auch das resultierende V umfassend (strikt). Für die strikten Formen muß allerdings auch noch der Definitionsbereich der Bedingung B berücksichtigt werden.

Die Datenumgebungen bei der If-Anweisung lassen sich wie folgt verdeutlichen.