Tafelübung 09

Algorithmen und Datenstrukturen

Lehrstuhl für Informatik 2 (Programmiersysteme)

Friedrich-Alexander-Universität Erlangen-Nürnberg

Wintersemester 2016/2017

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Übersicht

Generics – Einführung

- Motivation und Grundlagen
- Generics in Java

Abstrakte Datentypen

- Grundlagen
- Beispiel (alte Klausuraufgabe)
- Umsetzung in Java-Code

Objektvergleich – Wiederholung

- `==` vs. `equals`
- Besonderheiten bei ADTs

Grundlagen der Logik

wp-Kalkül

- Motivation und Grundlagen
- Zuweisungen
- Fallunterscheidungen
- Datentypen
- Schleifen

Generics – Einführung

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Generics: Motivation

Szenario

In unserer Anwendung benötigen wir...

- eine Liste zur Speicherung von Integer-Werten
- eine Liste zur Speicherung von Strings
- eine Liste zur Speicherung von `WasAuchImmer`-Objekten

Schlechte Lösung (1)

drei Listen-Implementierungen $\leadsto$ viel duplizierter Code $\lightning$

Schlechte Lösung (2)

Listen-Elemente vom Typ `Object` $\leadsto$ keine Typsicherheit $\lightning$

Generics: Grundlagen

- Generics $\equiv$ *generische Typen*
  - Abstraktion von den zu Grunde liegenden Typen
  - Erstellung eines Datentyps unter Angabe eines Typparameters
    - Typparameter als „Platzhalter“ für konkreten Typ
    - sog. Typparametrisierung
  - Verwendung eines generischen Typs $\leadsto$ Angabe der Typparameter

  $\leadsto$ stellt Typsicherheit schon zur Übersetzungszeit sicher

Im Listen-Beispiel

Eine Listen-Implementierung mit Typparameter für Typen der zu speichernden Objekte.

Generics in Java

Typparametrisierbare Listen-Implementierung

```java
public class List<T> {
    // ...
    public void add(T item) { /* ... */ }
    public T get(int idx) { /* ... */ }
}
```

Anmerkungen

- `List` heißt generischer Typ
- T heißt Typparameter
- T kann innerhalb `List` als „ganz normaler Typ“ verwendet werden
  - allerdings: T kann nicht zur Objekt-Instanziierung verwendet werden

Verwendung der Listen-Implementierung

```java
List<Integer> intList = new List<Integer>();
```

Einschränkung des Typparameters

- mögliche Typen für Typparameter können eingeschränkt werden (*Bounds*)
  - nur Klassen, die von bestimmter Klasse erben
  - nur Klassen, die bestimmtes Interface implementieren
- Nutzen? Nutzen!
  - stellt „Mindest-Anforderung“ an Typen dar
  - konkrete Typen haben „mindestens“ die entsprechende Schnittstelle

  $\leadsto$ erlaubt den Aufruf dieser Methoden, ohne konkreten Typ zu kennen
- in Java: Einschränkung des Typparameters mittels `extends`
  - auch bei Interfaces...

Beispiele

```
class Foo<T extends Exception> { /* ... */ }
class Bar<T extends Comparable<T>> { /* ... */ }
```

Statische generische Methoden (1)

Folgender Code kompiliert nicht

```
public class Foo<T> {
    public Foo(T param) { /* ... */ }
    public static Foo<T> getFoo(T param) {
        if (param == null) { /* Fehlerbehandlung */ }
        return new Foo<T>(param);
    }
}
```

Fehlermeldung beim Übersetzen

```
cannot make a static reference to the non-static type T
```

Was ist da schiefgegangen?

- bei der Instanziierung wird ein konkreter Typ für T eingesetzt
- Klassenmethoden sind nicht an ein Objekt gebunden
  - $\rightsquigarrow$ keine Instanziierung hat stattgefunden
  - $\rightsquigarrow$ für T existiert kein konkreter Typ

Statische generische Methoden (2)

Folgender Code kompiliert nicht

```
public static Foo<T> getFoo(T param) { /* ... */ }
```

Lösung

- bei der Methodendeklaration den Typparameter mit angeben
- Compiler kann dann beim Methodenaufruf einen passenden Typ einsetzen

... im Beispiel:

```
public static <T> Foo<T> getFoo(T param) { /* ... */ }
//=========================================================
Foo<Integer> intFoo = Foo.getFoo(4711);
```

Abstrakte Datentypen

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Abstrakte Datentypen (ADTs)

Abstrakter Datentyp [Wikipedia]

Ein Abstrakter Datentyp ist ein Verbund von Daten zusammen mit der Definition aller zulässigen Operationen, die auf sie zugreifen. Ein ADT beschreibt, *was* die Operationen tun (Semantik), aber noch nicht, *wie* sie es tun sollen (Implementierung).

Spezifikation der Semantik

Es gibt verschiedene Möglichkeiten zur Spezifikation der Semantik. In AuD verwenden wir eine Beschreibung mit Hilfe von Axiomen.

Spezifikation eines ADTs

- eine Spezifikation eines ADTs hat vier Bestandteile:

  adt Name des ADTs

  sorts Liste verwendeter Datentypen

  ops Schnittstellen der Operationen
  - Name, Parameter-Typen und Ergebnis-Typ
  - rein syntaktische Festlegung!

  axs Axiome zur Beschreibung der Semantik
  - vollständige Beschreibung des Verhaltens
  - rekursive Formulierung mit „Basisfall“ und „Rekursionsfall“
  - so allgemein wie möglich, so ausführlich wie nötig
    - keine Fälle doppelt abdecken
    - keine Fälle vergessen

Beispiel für einen ADT

Beispiel: Abstrakter Datentyp *Boolean*

adt Boolean
sorts Boolean
ops

true:			$\rightarrow$ Boolean
false:			$\rightarrow$ Boolean
not:		Boolean	$\rightarrow$ Boolean
and:	Boolean $\times$	Boolean	$\rightarrow$ Boolean
or:	Boolean $\times$	Boolean	$\rightarrow$ Boolean

axs

not(true())	= false()
not(false())	= true()
and(false(), A)	= false()
and(true(), true())	= true()
and(A, B)	= and(B, A)
or(A, B)	= not(and(not(A), not(B))

end Boolean

Beispiel: Klausuraufgabe aus dem WS 2013/14 (1)

ADT *FunList*

Folgender ADT ist vorgegeben und soll erweitert werden:

adt FunList
sorts FunList, Pred, T
ops
  nil: $\rightarrow$ FunList
  cons: T $\times$ FunList $\rightarrow$ FunList
  ...
axs
  ...
end FunList

nil erzeugt eine leere Liste, *cons* fügt ein neues Element am Anfang der Liste an.

Beispiel: Klausuraufgabe aus dem WS 2013/14 (2)

Teilaufgabe a)

FunList soll um zwei Operationen erweitert werden:

- *head* gibt das Kopfelement der Liste zurück, also das zuletzt mittels *cons* hinzugefügte Element, oder `null`, falls die Liste leer (*nil*) ist
- *tail* gibt die Restliste ohne Kopfelement zurück, wobei die Restliste der leeren Liste ebenfalls die leere Liste ist

Beispiel: Klausuraufgabe aus dem WS 2013/14 (3)

Erweiterter ADT *FunList*

adt FunList
sorts FunList, Pred, T
ops

nil:		$\to$	FunList
cons:	T $\times$ FunList	$\to$	FunList
head:	FunList	$\to$	T
tail:	FunList	$\to$	FunList

axs

head(nil())	= `null`
head(cons(t, l))	= t
tail(nil())	= nil()
tail(cons(t, l))	= l

end FunList

Beispiel: Klausuraufgabe aus dem WS 2013/14 (4)

Teilaufgabe c)

Es sei der folgende ADT zur Darstellung allgemeiner Prädikate gegeben:

adt Pred
sorts Pred, T, Boolean
ops
  apply: Pred $\times$ T $\rightarrow$ Boolean
end Pred

FunList soll um eine Operation *filter* erweitert werden, die ein Prädikat erhält und eine neue *FunList* zurückgibt, die alle Elemente der aktuellen *FunList* beinhaltet, die das Prädikat erfüllen.

Beispiel: Klausuraufgabe aus dem WS 2013/14 (5)

Signatur und Axiome der Operation *filter*

adt FunList
sorts FunList, Pred, T
ops

...

filter: $\quad$ Pred $\times$ FunList $\rightarrow$ FunList

axs

...

$$\text{filter}(p, \text{nil}()) = \text{nil}()$$

$$\text{filter}(p, \text{cons}(t, l)) = \begin{cases} \text{cons}(t, \text{filter}(p, l)) & \text{if apply}(p, t) \\ \text{filter}(p, l) & \text{else} \end{cases}$$

end FunList

Umsetzung in Java-Code

- **ops** $\leadsto$ Schnittstellen der Java-Methoden
- **axs** $\leadsto$ Implementierung der Java-Methoden

- grundsätzlich zwei Möglichkeiten zur Umsetzung:
  - Operationen als Klassen-Methoden:
    - Signatur entspricht ADT-Signatur
  - Operationen als Instanz-Methoden:
    - ein Parameter vom Typ des ADTs wird zur `this`-Referenz

- für Instanz-erzeugende Operationen gilt:
  - *können* als Konstruktor implementiert werden
  - allerdings geht dann die Benennung verloren

  $\leadsto$ besser: Konstruktor nur aufrufen

Umsetzung mit Klassen-Methoden

Ausschnitt aus dem ADT *FunList*

ops

...

nil:		$\rightarrow$ FunList
cons:	T $\times$ FunList	$\rightarrow$ FunList

Umsetzung mit Klassen-Methoden

```java
public class FunList<T> {
    private T head;
    private FunList<T> tail;
    private static final FunList<?> nil = new FunList<>(null, null);

    private FunList(T head, FunList<T> tail) {
        this.head = head; this.tail = tail;
    }
    public static FunList<?> nil() {
        return nil;
    }
    public static <T> FunList<T> cons(T head, FunList<T> tail) {
        return new FunList<T>(head, tail);
    }
}
```

Umsetzung mit Instanz-Methoden

Ausschnitt aus dem ADT *FunList*

ops

...
nil: $\rightarrow$ FunList
cons: T $\times$ FunList $\rightarrow$ FunList

Umsetzung mit Instanz-Methoden

```
public class FunList<T> {
    private T head;
    private FunList<T> tail;
    private static final FunList<?> nil = new FunList<>(null, null);

    private FunList(T head, FunList<T> tail) {
        this.head = head; this.tail = tail;
    }
    public static FunList<?> nil() { // static -> ohne Objekt aufrufbar
        return nil;
    }
    public FunList<T> cons(T head) {
        return new FunList<T>(head, this);
    }
}
```

Wichtiger Hinweis

Achtung!

In der Klausur und in den Hausaufgaben ist i.d.R. eingeschränkt, welche Operatoren und Datentypen verwendet werden dürfen. Sind beispielsweise sämtliche Java-API-Klassen verboten, darf auch kein `System.out.println()` im abgegebenen Code stehen.
Wenn es nicht explizit erlaubt ist, dürfen oft keine mathematischen Symbole (wie $<$, $\geq$, $\neq$, $\cdot$, ...) verwendet werden. Genauso sind meist jegliche Arten von Schleifen verboten!

Tipps

- genau lesen, was erlaubt ist und was nicht
- viel „Rekursion“ verwenden, immer an Basis- und Rekursionsfall denken!

Objektvergleich – Wiederholung

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Vergleiche bei primitiven Datentypen

- Java stellt Vergleichs-Operatoren für primitive Datentypen bereit
  - ==, !=, <, <=, >, >=
- implementieren Vergleich hinsichtlich natürlicher Ordnung
  - bei Zahlen beispielsweise nach der Wertigkeit

Beispiel

```
int a = 4711;
int b = 42;
int c = 42;
System.out.println(a < b); // false
System.out.println(b == c); // true
```

Vergleiche bei Referenz-Typen

- für Referenz-Typen kennt Java nur die Operatoren == und !=
  - vergleichen nur die Referenzen, kein „inhaltlicher“ Vergleich!

Beispiel

```
BigInteger a = new BigInteger("42");
BigInteger b = new BigInteger("42");
System.out.println(a == b); // false
```

- Java erlaubt (leider?) kein *Operator Overloading*

$\leadsto$ „inhaltliche“ Vergleiche werden auf Methoden-Aufrufe abgebildet

Inhaltlicher Vergleich von Objekten

- die Klasse `Object` beinhaltet die Methode `equals()`
  - wird u.a. von der Java-API zum inhaltlichen Vergleich verwendet
  - kann in Unterklassen überschrieben werden
  - ⤳ eigene Definition von „Gleichheit“

Beispiel

```java
public class Rechteck {
    private int breite, hoehe;
    // ...
    public boolean equals(Object other) {
        if (!(other instanceof Rechteck)) { return false; }
        return this.breite == ((Rechteck)other).breite
                && this.hoehe == ((Rechteck)other).hoehe;
    }
}
```

Objektvergleich bei ADTs

- manchmal kann man mit == doch „inhaltlich“ vergleichen
  ⤳ genau dann, wenn maximal ein Objekt mit entsprechenden Werten existiert

Beispiel

```
public class FunList<T> {
    private T head; private FunList<T> tail;
    private static final FunList<?> nil = new FunList<>(null, null);

    private FunList(T head, FunList<T> tail) {
        this.head = head; this.tail = tail;
    }
    public static FunList<?> nil() { return nil; }
    public static <T> FunList<T> cons(T head, FunList<T> tail) {
        return new FunList<T>(head, tail);
    }
}
```

- die leere Liste `nil` ist immer dasselbe Objekt
  ⤳ `(myList == nil)` ist genau dann `true`, wenn myList eine (die) leere Liste ist
- aber: `(myList == myOtherList)` vergleicht weiterhin nur Referenzen!
  ⤳ im Allgemeinen wird für zwei „inhaltlich gleiche“ `FunLists` false herauskommen

Grundlagen der Logik

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Klassische Aussagenlogik

- die (klassische) Aussagenlogik beschäftigt sich mit...
  - Elementaraussagen (*Atome*)
    - haben einen von zwei Wahrheitswerten ($\top$: *wahr* oder $\bot$: *falsch*)
    - oft mit Großbuchstaben bezeichnet (A, B, C, ...)
  - und deren Verknüpfung durch sog. Junktoren
    - Verknüpfung erzeugt zusammengesetzte Aussage
    - Wahrheitswert lässt sich aus den Wahrheitswerten der verknüpften Aussagen und der Bedeutung der verwendeten Junktoren ableiten
- im Folgenden: kurzer Überblick über die wichtigsten Grundlagen
  - nicht vollständig!
  - bei Bedarf selber recherchieren!
  - mehr in Mathe, GTI, GLoIn, ...

Wichtige Junktoren (1)

$\neg$ Negation („Nicht“)

- 1-stelliger Junktor
- Ergebnis ist wahr gdw. Operand falsch ist

$\wedge$ Konjunktion („Und“)

- 2-stelliger Junktor
- Ergebnis ist wahr gdw. beide Operanden wahr sind

$\vee$ Disjunktion („Oder“)

- 2-stelliger Junktor
- Ergebnis ist wahr gdw. mindestens ein Operand wahr ist

Wichtige Junktoren (2)

$\oplus$ Kontravalenz („Exklusives Oder“)

- 2-stelliger Junktor
- Ergebnis ist wahr gdw. genau ein Operand wahr ist

$\rightarrow$ Implikation („Hinreichende Bedingung“)

- 2-stelliger Junktor, links: *Prämisse*, rechts: *Konklusion*
- Ergebnis ist wahr gdw.
  - Prämisse falsch ist (*ex falso quodlibet*), *oder*
  - Prämisse und Konklusion richtig sind

$\leftrightarrow$ Äquivalenz („Genau-Dann-Wenn“)

- 2-stelliger Junktor
- Ergebnis ist wahr gdw. beide Operanden denselben Wahrheitswert haben

Regeln der Logik (1)

- Vereinfachen von Aussagen:
  - Umformungen mittels „Rechenregeln“
  - Ziel: bestimmen, ob Aussage wahr oder falsch ist
  - $\rightsquigarrow$ falls nicht möglich: in eine möglichst „einfache“ Form bringen

Allgemeine Vereinfachungen

Sei A ein beliebiger logischer Ausdruck. Dann gilt:

- $\neg\neg A \equiv A$, $A \wedge A \equiv A$, $A \vee A \equiv A$
- $A \wedge \top \equiv A$, $A \wedge \bot \equiv \bot$
- $A \vee \top \equiv \top$, $A \vee \bot \equiv A$
- $A \wedge \neg A \equiv \bot$ (*Kontradiktion*)
- $A \vee \neg A \equiv \top$ (*Tautologie*)

Regeln der Logik (2)

Kommutativgesetze

Seien A und B beliebige logische Ausdrücke. Dann gilt:

- $A \wedge B \equiv B \wedge A$
- $A \vee B \equiv B \vee A$

Assoziativgesetze

Seien A, B und C beliebige logische Ausdrücke. Dann gilt:

- $(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$
- $(A \vee B) \vee C \equiv A \vee (B \vee C)$

Regeln der Logik (3)

Distributivgesetze

Seien A, B und C beliebige logische Ausdrücke. Dann gilt:

- $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$
- $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$

De Morgan'sche Gesetze

Seien A und B beliebige logische Ausdrücke. Dann gilt:

- $\neg(A \wedge B) \equiv \neg A \vee \neg B$
- $\neg(A \vee B) \equiv \neg A \wedge \neg B$

wp-Kalkül

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Motivation

- Frage: Liefert eine bestimmte Methode das korrekte Ergebnis?
  - insb. bei sicherheitskritischen Systemen unbedingt notwendig
- Problem beim (naiven) Testen:
  - Methode müsste für alle Parameterwerte getestet werden
  - insbesondere bei großem Parameterraum praktisch unmöglich
    - ein `int`-Parameter: 2^{32} mögliche Aufrufe
    - zwei `int`-Parameter: $2^{32} \cdot 2^{32} = 2^{64}$ mögliche Aufrufe
    - ...

$\leadsto$ wir brauchen eine formale, analytische Methode

$\leadsto$ wp-Kalkül

Grundlagen des wp-Kalkül

- zur Analyse eines Algorithmus A:

  1. Formulierung einer Nachbedingung Q (*Postcondition*)
     - logischer Ausdruck über den Variablen des Algorithmus
     - Anforderung an Methode; soll nach Abarbeitung von A erfüllt sein
  2. Bestimmung der Vorbedingung P (*Precondition*)
     - logischer Ausdruck über den Eingabewerten von A
     - Welche „Eigenschaften“ müssen die Eingabewerte haben, damit nach Ausführung von A die Nachbedingung Q auch tatsächlich erfüllt ist?

- A ist korrekt, falls...
  - alle Eingabewerte, die tatsächlich auftreten können, P erfüllen

$\rightsquigarrow$ Rückwärtsanalyse: Analyse „von unten nach oben“

Schwächste Vorbedingung

- *wp*: *weakest precondition*
  - Vorbedingung stellt Anforderungen an die Eingabewerte dar
  - diese Anforderungen sollen „so gering wie möglich“ sein

$\rightsquigarrow$ Suche nach der schwächsten Vorbedingung

- möglichst viele Eingabewerte sollen Vorbedingung P erfüllen
- genauer: jene Eingabewerte, die P erfüllen, sollen genau die sein, die zu einer Erfüllung der Nachbedingung Q nach Ausführung von A führen

Schwächste Vorbedingung: Beispiel

Beispiel

```java
public static int addFifty(int v) {
  int res = v + 50;
  return res;
}
```

(Ausgedachte) Nachbedingung Q

Q: res > 100

Vorbedingung P

P: v > 50

Achtung

z.B. P: v > 75 ist auch eine Vorbedingung, aber nicht die schwächste!

Formale Definition

- E: Menge der möglichen Eingabedaten
- O: Menge der möglichen Ausgabedaten

- Vorbedingung P: Prädikat auf E
- Nachbedingung Q: Prädikat auf $E \times O$

- A: Algorithmus, der Eingabedaten in Ausgabedaten überführt
  - ist also eine Abbildung: $A : E \to O$
  - $A(x)$ bezeichnet die Ausgabedaten zu den konkreten Eingabedaten x
  - zunächst unter der Annahme, dass A terminiert

- falls gilt: $\forall x \in E : \; P(x) \to Q(x, A(x))$
  dann schreibt man: $\{P\}A\{Q\}$

  Lies: *Für alle Startzustände x, für die P gilt, gilt nach Ausführung von A auch Q.*

- das schwächste Prädikat P mit $\{P\}A\{Q\}$ heißt $wp(A, Q)$

Allgemeines Vorgehen

- seien *A* und *Q* gegeben; gesucht ist *wp*(*A*, *Q*)
- Rückwärtsanalyse $\rightsquigarrow$ *A* schrittweise „von hinten“ abarbeiten
  - in jedem Schritt letzte Anweisung *X* in *A* „verarbeiten“
  - dazu „Auswirkungen“ von *X* auf Nachbedingung bestimmen
    - Was muss *vor* *X* gelten, damit *nach* *X* Nachbedingung erfüllt ist?
    - Ergebnis wird neue Nachbedingung für restliche Anweisungen in *A*

Beispiel

```
// wp("X; Y; Z;", Q) = wp("X; Y;", Q') = wp("X;", Q'') =: P
X;
// wp("Y; Z;", Q) = wp("Y;", Q') =: Q''
Y;
// wp("Z;", Q) =: Q'
Z;
// Q
```

Zuweisungen $\rightsquigarrow$ Substitution

Zuweisungen der Form `l = r;`

In der Nachbedingung werden alle Vorkommen von `l` durch `r` ersetzt.

Beispiel

$$
\begin{aligned}
& wp(\text{“}\texttt{b = 2 - 2*a; b += a + 3; a = a - 2;}\text{“},\ a < 0 < b) \\
\equiv\ & wp(\text{“}\texttt{b = 2 - 2*a; b += a + 3;}\text{“},\ a - 2 < 0 < b) \\
\equiv\ & wp(\text{“}\texttt{b = 2 - 2*a; b = b + a + 3;}\text{“},\ a - 2 < 0 < b) \\
\equiv\ & wp(\text{“}\texttt{b = 2 - 2*a;}\text{“},\ a - 2 < 0 < b + a + 3) \\
\equiv\ & wp(\text{““},\ a - 2 < 0 < (2 - 2a) + a + 3) \\
\equiv\ & (a - 2 < 0 < (2 - 2a) + a + 3) \\
\equiv\ & (a - 2 < 0 < 5 - a) \\
\equiv\ & (a - 2 < 0) \land (0 < 5 - a) \quad \equiv \quad (a < 2) \land (a < 5) \quad \equiv \quad (a < 2) \quad =: \quad P
\end{aligned}
$$

Inkrement-/Dekrement-Operatoren

Inkrement-/Dekrement-Operatoren

- Präinkrement/-dekrement: nach [sic!] der eigentlichen Substitution behandeln
- Postinkrement/-dekrement: vor [sic!] der eigentlichen Substitution behandeln

Beispiel

$$
\begin{aligned}
& wp(\text{“}\texttt{b = 2 - 2*a; b += a + a++; a -= --b + b;}\text{“},\ a < 0 < b) \\
\equiv\ & wp(\text{“}\texttt{b = 2 - 2*a; b += a + a++; --b; a -= b + b;}\text{“},\ a < 0 < b) \\
\equiv\ & wp(\text{“}\texttt{b = 2 - 2*a; b += a + a++; --b;}\text{“},\ a - 2b < 0 < b) \\
\equiv\ & wp(\text{“}\texttt{b = 2 - 2*a; b += a + a++;}\text{“},\ a - 2(b-1) < 0 < b - 1) \\
\equiv\ & wp(\text{“}\texttt{b = 2 - 2*a; b += a + a; a++;}\text{“},\ a - 2b + 2 < 0 < b - 1) \\
\equiv\ & wp(\text{“}\texttt{b = 2 - 2*a; b += a + a;}\text{“},\ a - 2b + 3 < 0 < b - 1) \\
\equiv\ & wp(\text{“}\texttt{b = 2 - 2*a;}\text{“},\ a - 2(b + 2a) + 3 < 0 < (b + 2a) - 1) \\
\equiv\ & wp(\text{““},\ a - 2(2 - 2a + 2a) + 3 < 0 < (2 - 2a + 2a) - 1) \\
\equiv\ & (a - 1 < 0 < 1) \quad \equiv \quad (a < 1) \quad =: \quad P
\end{aligned}
$$

Fallunterscheidungen $\leadsto$ Aufspalten

Fallunterscheidungen der Form `if (b) {X;} else {Y;}`

- zwei Möglichkeiten zum Erfüllen der Nachbedingung $\leadsto$ Aufspalten
  - Bedingung wahr $\leadsto$ b erfüllt *und* `X` führt zur Nachbedingung
  - Bedingung falsch $\leadsto$ b nicht erfüllt *und* `Y` führt zur Nachbedingung

Beispiel

$$
\begin{aligned}
& wp(\text{“}\texttt{if (a < 11) \{a = a+17;\} else \{a = 40-a;\}}\text{“},\ 25 < a \leq 31) \\
\equiv\ & ((a < 11) \wedge wp(\text{“}\texttt{a = a+17;}\text{“},\ 25 < a < 31)) \\
& \qquad\qquad \vee ((a \geq 11) \wedge wp(\text{“}\texttt{a = 40-a;}\text{“},\ 25 < a \leq 31)) \\
\equiv\ & ((a < 11) \wedge (25 < a + 17 \leq 31)) \vee ((a \geq 11) \wedge (25 < 40 - a \leq 31)) \\
\equiv\ & ((a < 11) \wedge (8 < a \leq 14)) \vee ((a \geq 11) \wedge (9 \leq a < 15)) \\
\equiv\ & (8 < a < 11) \vee (11 \leq a < 15) \quad \equiv \quad (8 < a < 15) \quad =: \quad P
\end{aligned}
$$

Datentypen $\rightsquigarrow$ Vereinfachen

Datentypen

In manchen Fällen kann eine Bedingung weiter vereinfacht werden, wenn man die Datentypen der vorkommenden Variablen kennt.

Wichtig!

Wenn eine Vereinfachung auf Grund eines Datentyps verwendet wird, muss dies explizit hingeschrieben werden!

Beispiel

$$\begin{aligned}
& wp(\text{“}\texttt{a = a * 2;}\text{“},\ \ a \geq 1) \\
\equiv\ & wp(\text{““},\ \ 2a \geq 1) \\
\equiv\ & (a \geq \frac{1}{2}) \\
\text{(a ist Integer)} \quad \equiv\ & (a > 0) \quad =: \quad P
\end{aligned}$$

Korrektheit von Schleifen

- die Korrektheit von Schleifen ist schwerer zu beweisen
  - Anweisungen werden mehrfach ausgeführt
  - zur Korrektheit einer Schleife gehört auch deren Terminierung
  - $\leadsto$ Schleifen-Bedingung muss nach endlich vielen Durchläufen *falsch* werden

$\leadsto$ zwei Werkzeuge zum Korrektheitsbeweis von Schleifen

  - Schleifen<u>in</u>variante:
    - logischer Ausdruck; muss vor/nach jedem Durchlauf *wahr* sein
    - beschreibt das „Verhalten“ der Schleife
  - Schleifenvariante:
    - Funktion (im mathematischen Sinne) über den Programm-Variablen
    - Beweis der Terminierung der Schleife

$\leadsto$ Finden einer Schleifeninvariante und -variante

Schleifeninvariante

Schleifeninvariante

Sei eine Schleife der Form `while (b) { A; }` gegeben. Ein Prädikat I ist eine *gültige Schleifeninvariante*, falls gilt: $\{I \wedge b\}A\{I\}$.

Folgende Implikation muss also erfüllt sein: $(I \wedge b) \rightarrow wp(A, I)$

„Umgangssprachlich“

Falls *vor* der Ausführung des Rumpfes die Invariante I und die Schleifen-Bedingung b erfüllt sind (d.h. die Schleife wird auch tatsächlich ausgeführt), dann muss *nach* Ausführung des Rumpfes zumindest die Invariante erfüllt sein.
Die Anweisungen im Rumpf sorgen also dafür, dass die Invariante nach Ausführung noch immer erfüllt ist, falls sie es vor der Ausführung war.

Schleifeninvariante: Beispiel

Beispiel

```
public static int mul(int a, int b) {
  /* P: T */
  if (a < 0) {
    a = -a;
    b = -b;
  }
  int r = 0, x = a;
  while (x >= 1) {
    r = r + b;
    --x;
  }
  /* Q: r = a * b */
  return r;
}
```

Welche dieser Prädikate sind gültige Schleifeninvarianten?

$\top$ **?** $x \geq 0$ **?** $r = a \cdot b$ **?** $r = (a - x) \cdot b \land x \geq 0$ **?**

Schleifeninvariante: Lösungs-Skizze

Welche dieser Prädikate sind gültige Schleifeninvarianten?

$\top$ **?** $x \geq 0$ **?** $r = a \cdot b$ **?** $r = (a - x) \cdot b \land x \geq 0$ **?**

Bestimmung der gültigen Schleifeninvarianten

- $I := \top$
  - $wp(\text{“...“}, \ \top) \equiv \top$
  - $(I \land b) \equiv (\top \land x \geq 1) \rightarrow \top$
- $I := (x \geq 0)$
  - $wp(\text{“...“}, \ x \geq 0) \equiv (x - 1 \geq 0) \equiv (x \geq 1)$
  - $(I \land b) \equiv (x \geq 0 \land x \geq 1) \rightarrow (x \geq 1)$
- $I := (r = a \cdot b)$
  - $wp(\text{“...“}, \ r = a \cdot b) \equiv (r + b = a \cdot b)$
  - $(I \land b) \equiv (r = a \cdot b \land x \geq 1) \not\rightarrow (r + b = a \cdot b)$
- $I := (r = (a - x) \cdot b \land x \geq 0)$
  - $wp(\text{“...“}, \ r = (a - x) \cdot b \land x \geq 0) \equiv (r + b = (a - (x - 1)) \cdot b \land x - 1 \geq 0) \equiv (r = (a - x) \cdot b \land x \geq 1)$
  - $(I \land b) \equiv (r = (a - x) \cdot b \land x \geq 0 \land x \geq 1) \rightarrow (r = (a - x) \cdot b \land x \geq 1)$

Geeignete Schleifeninvariante (1)

- nicht jede Schleifeninvariante kann für Korrektheitsbeweis verwendet werden
  - *gültige* Schleifeninvariante $\leftrightsquigarrow$ *geeignete* Schleifeninvariante
- eine geeignete Schleifeninvariante I hat zusätzlich folgende Eigenschaften:
  1. das Prädikat I ist vor der Schleife erfüllt
  2. die Nachbedingung Q lässt sich nach der Schleife aus I implizieren
- also: eine geeignete Schleifeninvariante...
  - gilt vor dem ersten Durchlauf,
  - gilt nach jedem Durchlauf, *und*
  - impliziert, dass nach dem letzten Durchlauf Q erfüllt ist

Geeignete Schleifeninvariante (2)

Schema

```
{P}
C
{I}
while (b)
  {I ∧ b} A {I}
{I ∧ ¬b}
D
{Q}
```

1. das Prädikat I ist vor der Schleife erfüllt
   $\leadsto$ es muss gelten: $P \to wp(\mathtt{C}, I)$
2. die Nachbedingung Q lässt sich nach der Schleife aus I implizieren
   $\leadsto$ es muss gelten: $(I \land \neg b) \to wp(D, Q)$

Geeignete Schleifeninvariante: Beispiel

Beispiel

```
public static int mul(int a, int b) {
  /* P: T */
  if (a < 0) {
    a = -a;
    b = -b;
  }
  int r = 0, x = a;
  while (x >= 1) {
    r = r + b;
    --x;
  }
  /* Q: r = a * b */
  return r;
}
```

Welche dieser Prädikate sind vor der Schleife erfüllt?

$\top$ **?** $x \geq 0$ **?** $r = a \cdot b$ **?** $r = (a - x) \cdot b \wedge x \geq 0$ **?**

Geeignete Schleifeninvariante: Beispiel

Beispiel

```
public static int mul(int a, int b) {
  /* P: T */
  if (a < 0) {
    a = -a;
    b = -b;
  }
  int r = 0, x = a;
  while (x >= 1) {
    r = r + b;
    --x;
  }
  /* Q: r = a * b */
  return r;
}
```

Welche dieser Prädikate implizieren nach der Schleife *Q*?

$\top$ **?** $x \geq 0$ **?** $r = a \cdot b$ **?** $r = (a - x) \cdot b \land x \geq 0$ **?**

Schleifenvariante

- zur Korrektheit einer Schleife gehört auch deren Terminierung

$\leadsto$ Schleife muss nach endlich vielen Durchläufen abbrechen

- um Schleifen-Terminierung zu zeigen: Schleifenvariante finden
- Schleifenvariante: (mathematische) Funktion V mit folgenden Eigenschaften:
  - V ist eine Funktion über den in der Schleife benutzten Variablen
  - V ist ganzzahlig
  - V ist streng monoton fallend
  - V ist nach unten durch eine Konstante c beschränkt
    - lies: sobald V den Wert c annimmt, bricht die Schleife ab

Schleifenvariante: Formale Definition

- sei eine Schleife der Form `while (b) { A; }` gegeben
  - sei I eine Invariante für die Schleife
- Σ: Zustandsraum der Schleife
  - bezieht sich i.d.R. auf die in der Schleife verwendeten Variablen
- Schleifenvariante: $V : \Sigma \to \mathbb{Z}$
  - also: bildet Programmzustand auf ganze Zahl ab
- V muss nach unten beschränkt sein:
  - $I \to V \geq c$, mit $c \in \mathbb{Z}$
- V muss streng monoton fallend sein:
  - $\{I \wedge b \wedge V =: z\}A\{c \leq V < z\}$, mit $c, z \in \mathbb{Z}$

Schleifenvariante: Beispiel

Beispiel

```
public static int mul(int a, int b) {
  /* P: T */
  if (a < 0) {
    a = -a;
    b = -b;
  }
  int r = 0, x = a;
  while (x >= 1) {
    r = r + b;
    --x;
  }
  /* Q: r = a * b */
  return r;
}
```

Mögliche Schleifenvariante

- x

Partielle/Totale Korrektheit

- eine Funktion mit einer Schleife ist...
  - partiell korrekt, falls...
    - eine Schleifeninvariante I existiert *und*
    - I vor der Schleife gilt *und*
    - Q nach der Schleife aus I impliziert werden kann
  - total korrekt, falls...
    - sie partiell korrekt ist *und*
    - eine Schleifenvariante existiert

Fragen? Fragen!

(hilft auch den anderen)