Formal Verification of Control Flow MiniJava Programs

Funktionale Programmierung (Technische Universität München)

Verification

- **Äquivalenzen**
  - Indempotenz: $(F \wedge F) = F$ and $(F \vee F) = F$
  - Kommutativität: $(F \wedge G) \equiv (G \wedge F)$ and $(F \vee G) \equiv (G \vee F)$
  - Assoziativität: $((F \wedge G) \wedge H) \equiv (F \wedge (G \wedge H))$ and $((F \vee G) \vee H) \equiv (F \vee (G \vee H))$
  - Absorption: $(F \wedge (F \vee G)) \equiv F$ and $(F \vee (F \wedge G)) \equiv F$
  - Distributivität: $(F \wedge (G \vee H)) \equiv ((F \wedge G) \vee (F \wedge H))$ and $(F \vee (G \wedge H)) \equiv ((F \vee G) \wedge (F \vee H))$
  - Doppelnegation: $\neg\neg F \equiv F$
  - deMorgan: $\neg(F \wedge G) \equiv (\neg F \vee \neg G)$ and $\neg(F \vee G) \equiv (\neg F \wedge \neg G)$
  - Triviale Kontradiktion: $(F \wedge \neg F) \equiv false$
  - Triviale Tautologie: $(F \vee \neg F) \equiv true \rightarrow$ entire solution space
  - Dominanz: $(F \wedge false) \equiv false$ and $(F \vee true) \equiv true$
  - Identität: $(F \wedge true) \equiv F$ and $(F \vee false) \equiv F$
  - Implikation: $(F \rightarrow G) \equiv (\neg F \vee G)$
  - Dikonditional: $(F \leftrightarrow G) \equiv ((F \rightarrow G) \wedge (G \rightarrow F)) \equiv \neg(F \oplus G)$
  - XOR, note $\oplus = \mathrm{XOR}$: $(F \oplus G) \equiv ((F \vee G) \wedge (\neg F \vee \neg G)) \equiv ((F \wedge \neg G) \vee (\neg F \wedge G))$

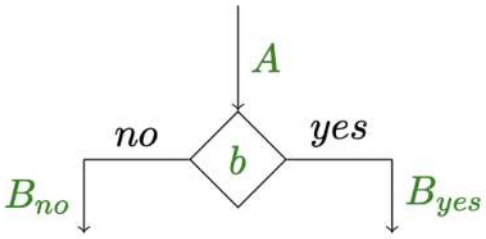
p	q	$p \oplus q$
T	T	F
T	F	T
F	T	T
F	F	F

Aussagenlogik - Beispiele

$x = y \vee x = 3 \Rightarrow y = 3$: statement does not hold as implying expression is related by OR-relation which means y mustn't be 3

WP-Kalkül

- assertions give precise description of the possible values of a current program state, the stronger, i.e. smaller the solution space of an assertion, the better is an assertion
- an assertion *A* is stronger than another assertion *B* if it includes / extends the information contained in *B*, i.e. whenever A is satisfied B is as well: **A stronger than B :=** $A \implies B$
- since $false \implies A$ and $true \implies A$ hold for all A, $false$ and $true$ are the strongest and weakest assertions ; $true$ is always satisfied, while $false$ can never be satisfied while a program may not violate $false$ by simply not reaching the respective point
- Wekeast Precondition Operator $\mathbf{WP}[\![\mathbf{s}]\!](B)$ transformiert Nachbedingung B für jede Anweisung s in eine minimale Anforderung A, die vor Ausführung erfüllt sein muss, damit B nach der Ausführung gilt
- Vorbedingung A für eine Anweisung s ist lokal konsistent sofern $A \Rightarrow \mathbf{WP}[\![s]\!](B)$, i.e. A implies weakest pre-condition for B
- Wenn alle Vorbedingungen lokal konsistent sind & Vorbedingung des Progamms true, dann ist eine Eigenschaft bewiesen
- Wenn alle Zusicherungen lokal konsistent sind, dann ist die Zusicherung am Startknoten NICHT true, das Program muss nicht zwangsweise für alle Eingaben terminieren, da A nicht unbedingt true sein muss
- If all assertions are locally consistent and the assertion Z is annotated at the stop node, then if *true* is annotated at the start node, then Z holds for all executions ; also replacing Z by a weaker assertion keeps all assertions locally consistent
- **[SPICK]** Operations $\mathbf{WP}[\![;]\!](B) \equiv B$, $\mathbf{WP}[\![\mathbf{x = e;}]\!](B) \equiv B[e/x]$ substituting x by e everywhere in B ; $\mathbf{WP}[\![\mathbf{x = read();}]\!](B) \equiv \forall x.B$; $\mathbf{WP}[\![\mathbf{write(e);}]\!](B) = B$



$$\begin{aligned}\mathbf{WP}[\![b]\!]\,(B_{no}, B_{yes}) &\equiv ((\neg b) \Rightarrow B_{no}) \wedge (b \Rightarrow B_{yes}) \\ &\equiv (b \vee B_{no}) \wedge (\neg b \vee B_{yes}) \\ &\equiv (\neg b \wedge B_{no}) \vee (b \wedge B_{yes}) \vee (B_{no} \wedge B_{yes}) \\ &\equiv (\neg b \wedge B_{no}) \vee (b \wedge B_{yes})\end{aligned}$$

$$\mathbf{WP}[\![b]\!]\,(B_0, B_1) \equiv ((\neg b) \Rightarrow B_0) \wedge (b \Rightarrow B_1)$$

The weakest pre-condition can be rewritten into:

$$\begin{aligned}\mathbf{WP}[\![b]\!]\,(B_0, B_1) &\equiv (b \vee B_0) \wedge (\neg b \vee B_1) \\ &\equiv (\neg b \wedge B_0) \vee (b \wedge B_1) \vee (B_0 \wedge B_1) \\ &\equiv (\neg b \wedge B_0) \vee (b \wedge B_1)\end{aligned}$$

Summary of the Approach

- (1) Annotate each program point with an assertion.
- (2) Program start should receive annotation **true**.
- (3) Verify for each statement s between two assertions A and B, that A implies the weakest pre-condition of s for B i.e.,

$$A \Rightarrow \mathbf{WP}[\![s]\!](B)$$

- (4) Verify for each conditional branching with condition b, whether the assertion A before the condition implies the weakest pre-condition for the post-conditions B_0 and B_1 of the branching, i.e.,

$$A \Rightarrow \mathbf{WP}[\![b]\!]\,(B_0, B_1)$$

An annotation with the last two properties is called locally consistent.

	Strongest Postcondition	Weakest Precondition
+	- Präziseste Information über den Programmstatus - Natürliche Art über Programme nachzudenken (da Ausführungsreihenfolge beachtet wird)	- Regeln zur Berechnung sind klar definiert - Nur notwendige Informationen enthalten - Notwendige Startbedingungen sind leicht ablesbar - Kann zum Beweis weiterer Eigenschaften (Terminierung) verwendet werden
-	- Regeln zur Berechnung sind schwer zu definieren - Formeln können sehr lang werden und unnötige Informationen enthalten	

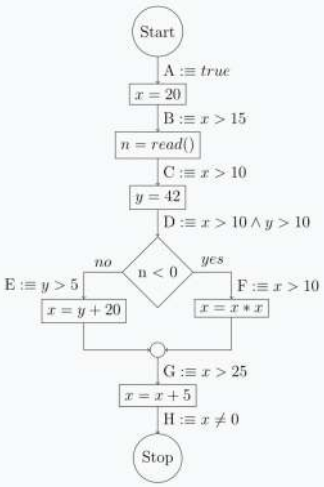
Damit eine assertion B nach einem Statement s hält, muss vor s die WP **WP[** s **]**(B) gelten. Falls vor s die Assertion A hält und die Bedingung $A \Longrightarrow \mathbf{WP}[\![s]\!](B)$ gilt, sagen wir die Assertions sind $lokal\ konsistent$. Um zu beweisen, dass H am Ende immer hält, müssen wir zeigen, dass:

1. alle assertions lokal konsistent sind
2. der Startknoten mit $true$ annotated ist

Wir beginnen bei $G \Longrightarrow \mathbf{WP}[\![x = x + 5]\!]$(H) und arbeiten uns rückwärts bis zu A vor.

Local Consistency

In the following control flow graph assertions are annotated to all the edges.



Check whether the annotated assertions prove that the program computes an $\mathbf{x} \neq 0$ and discuss why this is the case.

$$\begin{aligned} &\mathbf{WP}[\mathrm{x} = \mathrm{x} + 5](\mathrm{H}) \\ &\equiv \mathbf{WP}[\mathrm{x} = \mathrm{x} + 5](\mathrm{x} \neq 0) \\ &\equiv \mathrm{x} + 5 \neq 0 \\ &\Longleftarrow \mathrm{x} > 25 \quad \equiv \mathrm{G} \end{aligned}$$

die nächste assertion x > 25 impliziert, dass egal was x ist (es muss aber grösser 25 sein) und wir dazu 5 addieren, dann kann das Ergebnis auf gar keinen Fall 0 sein

$$\begin{aligned} &\mathbf{WP}[\mathrm{x} = \mathrm{x} * \mathrm{x}](\mathrm{G}) \\ &\equiv \mathbf{WP}[\mathrm{x} = \mathrm{x} * \mathrm{x}](\mathrm{x} > 25) \\ &\equiv \mathrm{x} * \mathrm{x} > 25 \\ &\equiv |\mathrm{x}| > 5 \\ &\Longleftarrow \mathrm{x} > 10 \quad \equiv \mathrm{F} \end{aligned}$$

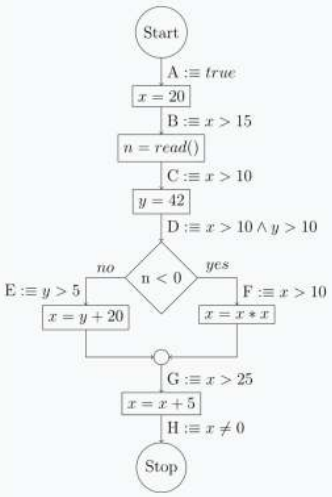
hier genauso: wenn x > 10 ist, dann muss der Absolutwert von x auch definitiv grösser 5 sein, also ist die Aussage x > 10 ==> |x| > 5 wahr

$$\begin{aligned} &\mathbf{WP}[\mathrm{x} = \mathrm{y} + 20](\mathrm{G}) \\ &\equiv \mathbf{WP}[\mathrm{x} = \mathrm{y} + 20](\mathrm{x} > 25) \\ &\equiv \mathrm{y} + 20 > 25 \\ &\equiv \mathrm{y} > 5 \quad \equiv \mathrm{E} \end{aligned}$$

Es darf von oben nach unten impliziert werden, z.B. G darf H implizieren.

Local Consistency

In the following control flow graph assertions are annotated to all the edges.



Check whether the annotated assertions prove that the program computes an $\mathbf{x \neq 0}$ and discuss why this is the case.

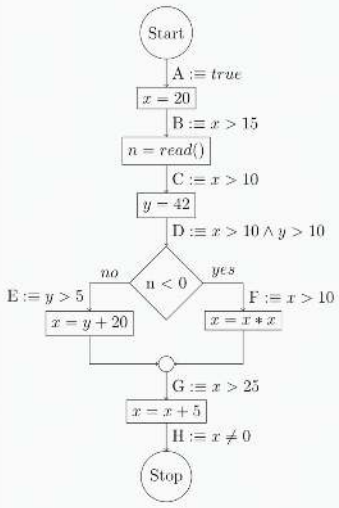
$$
\begin{aligned}
&\mathbf{WP}[\![\mathrm{n} < 0]\!](\mathrm{E}, \mathrm{F}) \\
\equiv\ &\mathbf{WP}[\![\mathrm{n} < 0]\!](\mathrm{y} > 5, \mathrm{x} > 10) \\
\equiv\ &(\mathrm{n} \geq 0 \wedge \mathrm{y} > 5) \vee (\mathrm{n} < 0 \wedge \mathrm{x} > 10) \\
\Longleftarrow\ &\mathrm{x} > 10 \wedge \mathrm{y} > 10 \quad \equiv \mathrm{D}
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{WP}[\![\mathrm{y} = 42]\!](\mathrm{D}) \\
\equiv\ &\mathbf{WP}[\![\mathrm{y} = 42]\!](\mathrm{x} > 10 \wedge \mathrm{y} > 10) \\
\equiv\ &\mathrm{x} > 10 \wedge 42 > 10 \\
\equiv\ &\mathrm{x} > 10 \quad \equiv \mathrm{C}
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{WP}[\![\mathrm{n} = \mathrm{read}()]\!](\mathrm{C}) \\
\equiv\ &\mathbf{WP}[\![\mathrm{n} = \mathrm{read}()]\!](\mathrm{x} > 10) \\
\equiv\ &\forall \mathrm{n}.\mathrm{x} > 10 \\
\equiv\ &\mathrm{x} > 10 \\
\Longleftarrow\ &\mathrm{x} > 15 \quad \equiv \mathrm{B}
\end{aligned}
$$

Local Consistency

In the following control flow graph assertions are annotated to all the edges.



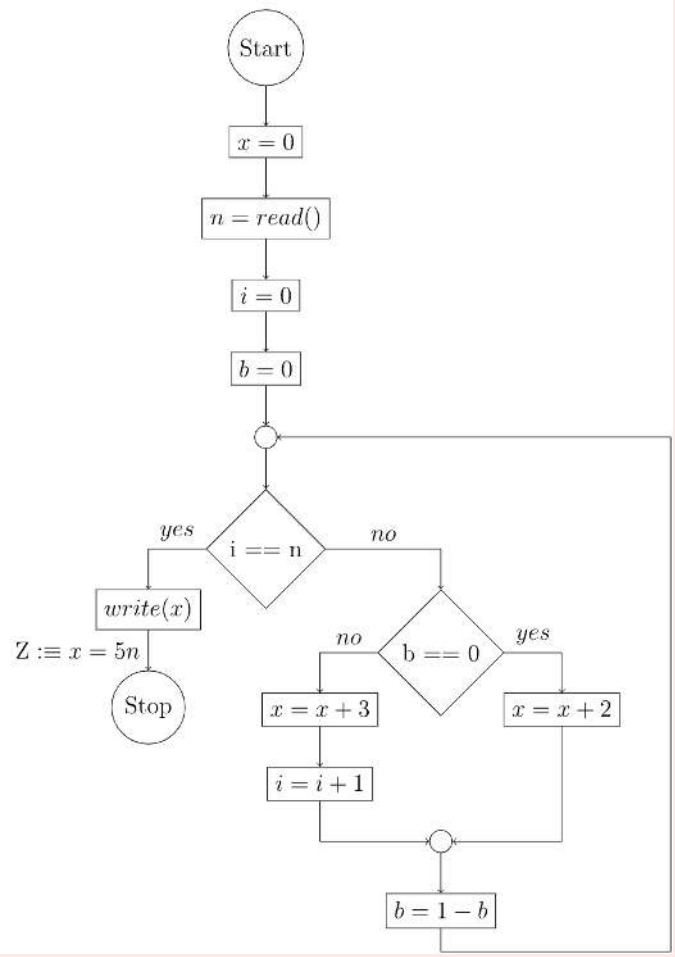
Check whether the annotated assertions prove that the program computes an $\mathbf{x \neq 0}$ and discuss why this is the case.

$$
\begin{aligned}
&\mathbf{WP}[\![\mathrm{x} = 20]\!](\mathrm{B}) \\
\equiv\ &\mathbf{WP}[\![\mathrm{x} = 20]\!](\mathrm{x} > 15) \\
\equiv\ &20 > 15 \\
\equiv\ &\mathrm{true} \quad \equiv \mathrm{A}
\end{aligned}
$$

- **Loop Invariants**: um Programme zu beweisen mit einer Loop benötigen wir **LOOP INVARIANTEN**, eine assertion die zu jedem zeitpunkt der loop execution true sein muss (vorher und nachher)
  - Loop Invariante:
    - lopp inv muss Ziel ingewisser weise enthalten
    - Schleife betrachten und Beziehung der einzelnen Variablen herstellen und gucken

- wenn r definiert wird, ein sogenannter Schleifenzähler, dann versucht es auf 0 zu setzen und darauf achten, dass es kleiner wird

- **Tipps zur Bildung der Schleifeninvariante**
  - Variablen mitaufnehmen, die zur Berechnung (meistens von x) wichtig sind
  - x steht in den aller meisten Fällen in Abhängigkeit zum Schleifenzähler (meistens i, wenn kein i als klassischer Schleifenzähler existiert, kann eig immer einer aus anderen Variablen erstellt werden)
  - Das Gegenteil der Abbruchbedingung der Schleife mit aufnehmen
  - Variablen aus Beweisziel (normalerweise Z) muss in Verbindung mit Variablen in Invariante stehen
  - enthält eine Schleife eine Verzweigung (eine weitere Fallunterscheidung), dann sollte die Variable der Verzweigung mit in die Invariante durch eine Fallunterscheidung aufgenommen werden, z.B.



I:=== x = 5i + 2b AND b in {0, 1}

- In order to end up at defined loop invariant when combining conditional statements, e.g. WP[i>0](x= |n|a, x = a*(|n| -i) AND i≥0), the clue is to define an expression which is stronger than the previous expression and therefore implies it, because in WP proofs we are allowed to perform implications, this expression, usually the left-side of the OR clause, should then be very similar to the right-side, such that it can be combined considering the statement *s* and its negated counter-part to arrive at the loop invariant

  EXAMPLE

  I:=== WP[i>0](D, B) === (i ≤ 0 AND x= |n|a) OR (i>0 AND x = a*(|n| -i) AND i≥0)

  === (**i ≤ 0 AND x= |n|a**) OR (i>0 AND **x = a|n| - ai** AND i ≥ 0)

  <== (**i=0 AND x= |n|a - ai**) OR (i>0 AND x = |n|a - ai AND i ≥ 0)

  === x = a(|n| - i) AND i ≥ 0

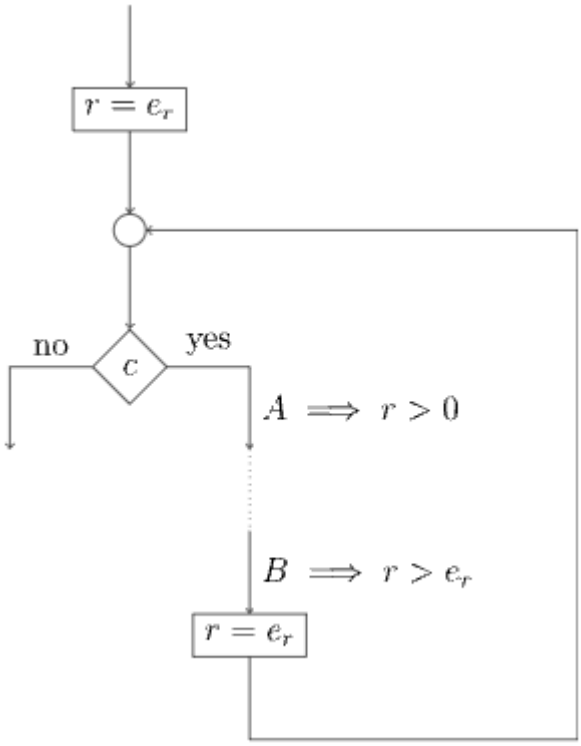
- Special Rules of logarithm

Rule or special case	Formula
Product	$\ln(xy) = \ln(x) + \ln(y)$
Quotient	$\ln(x/y) = \ln(x) - \ln(y)$
Log of power	$\ln(x^y) = y\ln(x)$
Log of e	$\ln(e) = 1$
Log of one	$\ln(1) = 0$
Log reciprocal	$\ln(1/x) = -\ln(x)$

  - $\ln(e^k) = k$ and $e^{\ln c} = c$

Proving Termination of a Program

- general structure of a loop



- loop-free programs always terminate
- General approach
  - Make sure that each loop is executed only finitely often ...
  - For each loop, identify an indicator value r, that has two properties
    - (1) $r > 0$ whenever the loop is entered;
    - (2) r is decreased during every iteration of the loop.
  - Transform the program in a way that, alongside ordinary program execution, the indicator value r is computed.
  - Verify that properties (1) and (2) hold!
- loop terminates iff number of iterations is finite; choose variable of program that increases (decreases) in every iteration, prove there is an upper (or lower) bound that variable cannot reach, thus loop must run only finite times otherwise it would reach bound, common strategy is to choose variable increased/decreased by 1
- if no such variable exist introduce new variable r and compute it such that it modules the behavior of such an iterator

- find local concistent annotations in program such that assertion A enforces lower bound ($A \Rightarrow r > 0$) and assertion B at end of loop (before recomputation of r in e_r) guarantees that r decreased
- loop invariante typically contains $r = e_r$ as well as relations between relations used in e_r
- Das ein Programm keine Schleife enthält ist eine hinreichende Bedingung, dass dieses terminiert; Es reicht nicht, dass min eine Schleife endlich oft durchläuft, wenn eine zweite im Programm dies nicht tut, dann terminiert das Programm trotzdem nicht.