Einführung in die Informatik 2

Prof. Dr. Seidl, J. Kranz, N. Hartmann, J. Brunner
WS 2016/17

Übungsblatt 8
Abgabefrist: 9.1.2017

In diesem Übungsblatt sollen die beiden Themen der Vorlesung (Verifikation und OCaml) zusammengeführt werden. Sie werden deshalb ein Programm in OCaml schreiben, das in der Lage ist ein gegebenes Programm durch Berechnung der schwächsten Vorbedingungen (siehe Vorlesungsscript Seite 27ff) zu verifizieren. In den Präsenzaufgaben werden die dazu benötigten Datentypen und Ausgabefunktionen vorgestellt und der Umgang mit ihnen geübt. In den Hausaufgaben werden Sie dann den Verifikator implementieren.

Aufgabe 8.1 (P) Formeln

Um Aussagen über das Programm zu machen, Zusicherungen zu beschreiben und Vorbedingungen zu berechnen, werden zuerst Beschreibungen von Variablen sowie arithmetischen und logischen Formeln benötigt. Um beliebig komplexe Formeln zu vermeiden, werden Formeln im Folgenden lediglich als Konjunktion von Implikationen betrachtet. Eine Formel hat also stets die Form $(a_0 \implies b_0) \wedge \cdots \wedge (a_n \implies b_n)$. Alle a_k, b_k sind dabei entweder $true$, $false$ oder Gleichungen bzw. Ungleichungen der Form $l \odot r$ mit $\odot \in \{=, \neq, <, \leq\}$ und l, r sind entweder einfache numerische Konstanten oder lineare Ausdrücke der Form $ax + b$, wobei x eine Variable und $a, b \in \mathbb{Z}$ sind. Die folgenden OCaml-Typen sind dafür gegeben.

```
1  type var                    = string
2  type linear_expression      = Constant of int
3                              | LinExpr of int * var * int
4  type boolean_expression_op  = Eq | Neq | Le | Leq
5  type boolean_expression     = True
6                              | False
7                              | BoolExpr of linear_expression
8                                  * boolean_expression_op
9                                  * linear_expression
10 type implication            = { assumption : boolean_expression;
11                                 conclusion : boolean_expression }
12 type formula                 = implication list
```

Konstruieren Sie die folgenden Formeln in OCaml:

1. $a = b \implies a = 3$
2. $x \geq 17 \implies y \leq z$
3. $u \neq v$
4. $a = b \wedge c < k$
5. $x + 4 < 8 \implies false$
6. $-5i + 8 \neq i \wedge g \neq n$
7. $(9h - 1 \geq u \implies 8 > 9) \wedge (7 = 2 \implies u \leq 21) \wedge (t = r \implies b \neq 21)$

8. $false \implies s = 42$
9. $false$
10. $true$

Lösungsvorschlag 8.1

Die Lösung befindet sich in der Datei `ocaml/p8_sol.ml`.

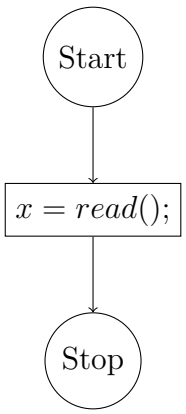
Aufgabe 8.2 (P) Graphen

Bevor Sie nun schwächste Vorbedingungen berechnen können, benötigen Sie noch eine geeignete Repräsentation von Programmen. Dazu werden - wie bisher - Kontrollflussgraphen verwendet. Ein Kontrollfussgraph (CFG) enthält dabei eine Menge von Knoten und eine Menge von Kanten. Jedem Knoten wird dabei eine eindeutige Nummer zugeordnet, welche verwendet werden kann um die Kanten zu Beschreiben. Kanten können außerdem optional eine Guard (bei Verzweigungen), sowie die bereits berechnete Zusicherung, in Form einer wie in 8.1 beschriebenen Formel, enthalten.

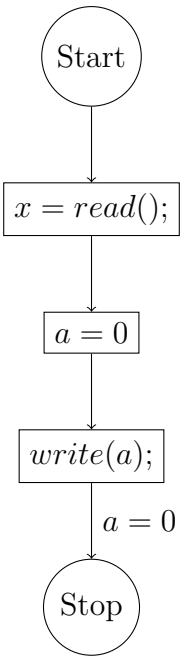
```
1  type instruction = Assignment of var * linear_expression
2                   | Read of var
3                   | Write of var
4  type node        = Statement of instruction
5                   | Branch of boolean_expression
6                   | Join
7                   | Start
8                   | Stop
9  type node_id     = int
10 type guard       = No | Yes
11 type edge        = node_id * guard option * formula option * node_id
12 type cfg         = { nodes : (node_id * node) list; edges : edge list }
```

Konstruieren Sie mit diesen Typen die folgenden Graphen in OCaml.

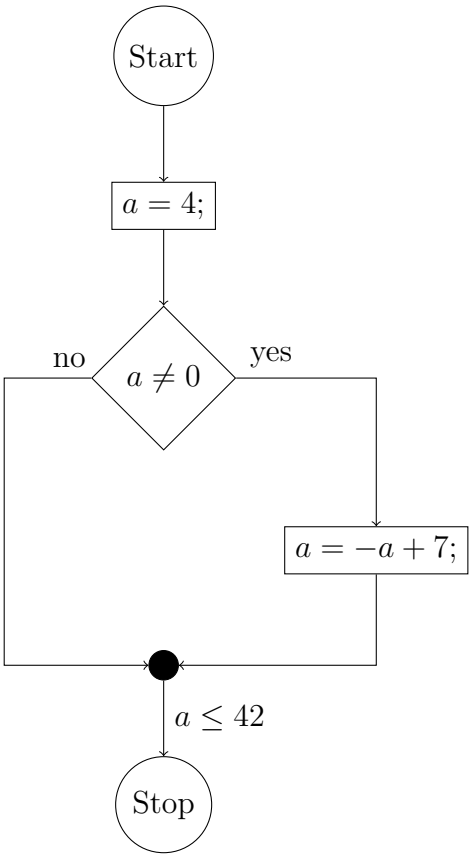
1.



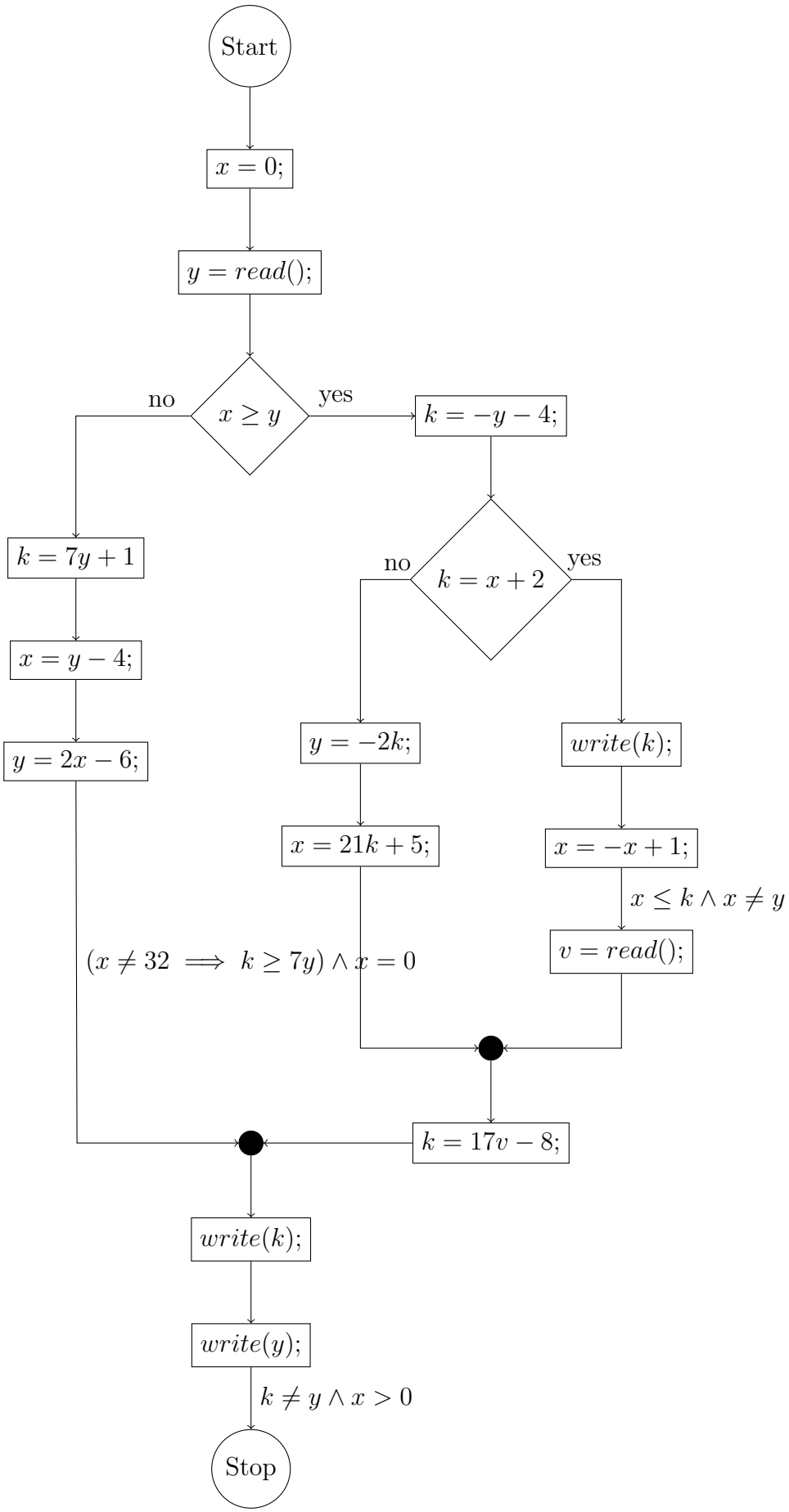
2.


Start
$x = read();$
$a = 0$
$write(a);$
$a = 0$
Stop

3.


Start
$a = 4;$
$a \neq 0$
no
yes
$a = -a + 7;$
$a \leq 42$
Stop

4.


Start
$x = 0;$
$y = read();$
$x \geq y$
no
yes
$k = -y - 4;$
$k = 7y + 1$
$k = x + 2$
no
yes
$x = y - 4;$
$y = 2x - 6;$
$y = -2k;$
$write(k);$
$x = 21k + 5;$
$x = -x + 1;$
$x \leq k \wedge x \neq y$
$v = read();$
$(x \neq 32 \implies k \geq 7y) \wedge x = 0$
$k = 17v - 8;$
$write(k);$
$write(y);$
$k \neq y \wedge x > 0$
Stop

Lösungsvorschlag 8.2

Die Lösung befindet sich in der Datei `ocaml/p8_sol.ml`

Allgemeine Hinweise zur Hausaufgabenabgabe

Die Hausaufgabenabgabe bezüglich dieses Blattes erfolgt über das Ocaml-Abgabesystem. Sie erreichen es unter `https://vmnipkow3.in.tum.de`. Sie können hier Ihre Abgaben einstellen und erhalten Feedback, welche Tests zu welchen Aufgaben erfolgreich durchlaufen. Sie können Ihre Abgabe beliebig oft testen lassen. Bitte beachten Sie, dass Sie Ihre Abgabe stets als **eine einzige UTF8-kodierte Textdatei mit dem Namen *ha8.ml* hochladen müssen**. Die hochgeladene Datei muss ferner den Signaturen in *ha8.mli* entsprechen.

Aufgabe 8.3 (H) Verifikation [20 Punkte]

Implementieren Sie ein OCaml-Programm, das die Korrektheit eines Programms überprüft, indem es die schwächsten Vorbedingungen aller Anweisungen in einem gegebenen Kontrollflussgraphen berechnet. Benutzen Sie dafür die in den Präsenzaufgaben vorgestellten Datentypen. Des Weiteren stehen Ihnen die folgenden Funktionen zur Verfügung, welche String-Repräsentationen der einzelnen Datentypen erzeugen bzw. den Kontrollfussgraphen (inklusive aller bereits berechneten Zusicherungen) als .dot Datei exportieren.

```
1 val string_of_linear_expression : linear_expression -> string
2 val string_of_boolean_expression : boolean_expression -> string
3 val string_of_implication : implication -> string
4 val string_of_formula : formula -> string
5 val string_of_instruction : instruction -> string
6 val print_cfg : cfg -> string -> unit
```

Hinweis: Nutzen Sie diese Funktionen um die Funktion und Korrektheit Ihrer einzelnen Zwischenschritte zu überprüfen. In der Angabe sind außerdem die folgenden Beispielgraphen gegeben mit denen Sie Ihre Implementierungen überprüfen können. Wenn Sie alle Funktionen korrekt implementieren, erhalten Sie außerdem die entsprechenden Zusicherungen an der Kante, die den Start-Knoten verlässt:

- `cfg0` - $1 \leq 2i$
- `cfg1` - *true*
- `cfg2` - *true*
- `cfg3` - $a = 0 \wedge b = 0$

Implementieren Sie nun das Verifikationsprogramm, indem Sie der folgenden Anleitung Schritt für Schritt folgen.

1. `[0.5+0.5 Punkte]` Implementieren Sie die Funktionen `val get_in_edges : cfg -> node_id -> edge list` und `val get_out_edges : cfg -> node_id -> edge list`, die eine Liste mit allen eingehenden bzw. ausgehenden Kanten eines Knotens liefern.

2. `[0.5 Punkte]` Impelementieren Sie die Funktion `val must_compute_wp : cfg -> node_id -> bool`, welche entscheidet, ob die schwächste Vorbedingung an einem Knoten noch berechnet werden muss (diese also noch nicht berechnet ist).

3. `[0.5 Punkte]` Überlegen Sie, wann die schwächste Vorbedingung eines Knotens berechnet werden kann. Implementieren Sie die Funktion `val can_compute_wp : cfg -> node_id -> bool`, die angibt, ob die schwächste Vorbedingung für den übergebenen Knoten berechnet werden kann.

4. `[1 Punkt]` Schreiben Sie die Funktion `val set_wp : cfg -> node_id -> formula -> cfg`, welche die Vorbedingung eines Knotens an den entsprechenden Kanten setzt und den modifizierten Graphen zurückgibt.

5. `[1 Punkt]` Implementieren Sie die Funktion

   ```
   val normalize_boolean_expression : boolean_expression ->
                                      boolean_expression,
   ```

   die einen Boolschen Ausdruck in eine einheitliche Form bringt, welche die weitere Vereinfachung erleichtert. Dazu werden einerseits alle Ungleichungen der Form $e_0 < e_1$ in Ungleichungen über dem Operator $\leq$ (also $e_0 \leq e_1'$) umgeformt. Überlegen Sie sich, wie e_1' aussehen muss. Außerdem sollen alle variablen-unabhängigen linearen Ausdrücke, also diejenigen, welche die Form $0x + b$ haben, in Konstanten umgewandelt werden.

6. `[1 Punkt]` Sie benötigen außerdem eine Funktion, die einen Boolschen Ausdruck negiert. Schreiben Sie dazu die Funktion

   ```
   val negate_boolean_expression : boolean_expression ->
                                   boolean_expression.
   ```

7. `[3 Punkte]` Während der Berechnung der Vorbedingungen werden ständig neue Formeln konstruiert, welche vereinfacht werden müssen, bevor Sie mit diesen weiterrechnen können. Implementieren Sie dazu die Funktion

   ```
   val simplify_boolean_expression : boolean_expression ->
                                     boolean_expression,
   ```

   welche den Boolschen Ausdruck zuerst normalisiert und anschließend mindestens die folgenden Vereinfachungen durchführt:

   - Für die beiden Konstanten c_0, c_1 und einen beliebigen Operator $\odot \in \{=, \neq, <, \leq\}$ muss der Ausdruck $c_0 \odot c_1$ so weit wie möglich vereinfacht werden.
   - Die Gleichung $a_0x + b = a_1x + b$ muss vereinfacht werden. Überlegen Sie sich, welche Informationen Sie gewinnen können wenn a_0 gleich bzw. ungleich a_1 ist.
   - Die Ausdrücke $a_0x_0 + b_0 \odot e$ mit beliebigem e und $c \odot a_0x_0 + b_0$ mit beliebiger Konstante c sollen in die Form $x_0 \odot e'$ bzw. $e' \odot x_0$ vereinfacht werden, falls dies möglich ist. Ansonsten soll nach $a_0x_0 \odot e'$ bzw. $e' \odot a_0x_0$ umgeformt werden.

8. [1 Punkt] Mit Hilfe von `simplify_boolean_expression` können nun Implikationen vereinfacht werden. Implementieren Sie dazu die Funktion

   `val simplify_implication : implication -> implication`,

   welche zum einen beide Seiten der Implikation vereinfacht und zum anderen Implikationen der Form $b \implies false$ in $true \implies b'$ umwandelt.

9. [2 Punkte] Als nächstes gilt es diejenigen Implikationen zu erkennen, welche mit Sicherheit immer erfüllt sind. Implementieren Sie dazu die Funktion `val is_tautology : implication -> bool`, die mindestens die folgenden Fälle erkennt:

   - $false \implies b$
   - $b \implies true$
   - $b \implies b$
   - $e_0 = e_1 \implies e_1 = e_0$
   - $e_0 \neq e_1 \implies e_1 \neq e_0$
   - $e_0 \leq e \implies e_0 \leq e'$, wenn $e \leq e'$
   - $e \leq e_1 \implies e' \leq e_1$, wenn $e \geq e'$

   Hinweis: Die letzten beiden Fälle sind zumindest dann zu erkennen, wenn sowohl e als auch e' Konstanten sind oder sich lediglich durch einen konstanten Offset unterscheiden (also $e \equiv ax + b$ und $e' \equiv ax + b'$).

10. [1 Punkt] Jetzt können Sie die Funktion `val simplify_formula : formula -> formula` implementieren, welche zuerst die einzelnen Implikationen der Formel vereinfacht und anschließend die Formel vereinfacht, indem Tautologien aus der Konjunktion entfernt werden.

11. [1+1 Punkte] Die Berechnung der schwächsten Vorbedingung von Zuweisungen wurde in der Vorlesung wie folgt definiert: $\mathsf{WP}[\![\ \texttt{x = e;}\ ]\!](B) = B[e/x]$. In der Formel B müssen also alle Vorkommen der Variable x durch den Term e ersetzt werden. Implementieren Sie die Funktionen

    ```
    val replace_var_in_linear_expression : linear_expression -> var ->
                  linear_expression -> linear_expression
    ```

    und

    ```
    val replace_var_in_boolean_expression : boolean_expression -> var ->
                  linear_expression -> boolean_expression,
    ```

    die diese Ersetzungen durchführen.

12. [4 Punkte] Sie können jetzt die Funktion `val compute_wp : cfg -> node_id -> formula` implementieren, welche die schwächste Vorbedingung am übergebenen Knoten berechnet. Gehen Sie dabei von folgenden Voraussetzungen aus:

    - Der Kontrollflussgraph enthält keine Schleifen.
    - Die Funktion `compute_wp` wird nur für Knoten aufgerufen, für die eine Berechnung der Vorbedingung möglich ist.
    - Die Funktion wird niemals für den Start- oder Stop-Knoten aufgerufen.

Wenn Sie diese (oder andere unerwartete) Fälle dennoch abfangen möchten, verlassen Sie die Funktion indem Sie `failwith "..."` mit einer entsprechenden Fehlermeldung aufrufen. Die schwächste Vorbedingung an einer Verzweigung mit Bedingung b und Nachbedingungen B_0 und B_1 an den No- bzw. Yes-Zweigen, wurde wie folgt definiert: $\mathsf{WP}[\![\ \texttt{b}\ ]\!](B_0, B_1) = (\neg b \implies B_0) \wedge (b \implies B_1)$. Wenn Sie die übrigen Funktionen korrekt implementiert haben, dann können Sie davon ausgehen, dass alle Implikationen in B_0 und B_1 die Form $\mathit{true} \implies a$ haben. Machen Sie sich die Äquivalenzen $a \implies (\mathit{true} \implies b) \equiv a \implies b$ und $a \implies (b_0 \wedge \cdots \wedge b_n) \equiv (a \implies b_0) \wedge \cdots \wedge (a \implies b_n)$ zu Nutze, um die schwächste Vorbedingung an Verzweigungen in der in 8.1 beschriebenen Form zu konstruieren. Für Read-Anweisungen benutzen Sie den vereinfachten WP-Operator $\mathsf{WP}[\![\ \texttt{x = read();}\ ]\!](B) = B$ anstelle der Definition der Vorlesung (da sich $\forall x : B$ nicht mit den in 8.1 beschriebenen Formeln ausdrücken lässt).

13. `[2 Punkte]` Abschließend können Sie nun die Hauptfunktion `val verify : cfg -> cfg` implentieren, die das übergebene Programm (bzw. dessen Kontrollflussgraphen) verifiziert. Überprüfen Sie dabei zuerst an welchen Knoten noch Zusicherungen berechnet werden müssen und an welchen dieser Knoten eine Berechnung bereits möglich ist. Berechnen Sie dann diese Zusicherung (schwächste Vorbedingung) und annotieren Sie die entsprechenden Kanten im CFG. Wiederholen Sie dies mit dem modifizierten CFG solange bis entweder alle Zusicherungen berechnet sind oder keine weitere Berechnung möglich ist. Geben Sie den Graphen zurück, der alle berechneten Zusicherungen enthält.

Lösungsvorschlag 8.3

Die Lösung befindet sich in der Datei `ocaml/ha8_sol.ml`.