Übung zur Vorlesung

Automatisierte Logik und Programmierung

Prof. Chr. Kreitz

Universität Potsdam, Theoretische Informatik — Sommersemester 2004

Blatt 2 — Abgabetermin: 05.06.2009

Aufgabe 2.1 (Die Mühen des Beweisens ohne Entscheidungsprozeduren I)

Beweisen Sie die folgende Sequenz ohne Verwendung der `arith`-Entscheidungsprozedur:

$$\texttt{x:}\mathbb{Z}\texttt{, y:}\mathbb{Z}\texttt{, 4-y}\leq\texttt{1-x, y-2<6} \vdash \texttt{x<7}$$

Formulieren Sie hierzu die erforderlichen Gesetze über $\leq$, `<`, Addition, Subtraktion, Monotonie, etc. und wenden Sie diese dann in Ihrem Beweis mit der Taktik `InstLemma` an. Die Lemmata brauchen nicht bewiesen zu werden, sollten aber möglichst allgemeingültiger Natur sein. So könnte man z.B. ein spezielles Lemma formulieren, das `4-y`$\leq$`1-x` in einem Schritt in `4+x`$\leq$`1+y` überführt, aber eigentlich ist hierfür die Kombination von drei elementareren Lemmata erforderlich.

`InstLemma: tok->term list->tactic` (Manual, S. 128) instantiiert Lemmata der Form $\forall x_1\texttt{:}T_1\texttt{..}x_n\texttt{:}T_n\texttt{.}P \Rightarrow Q$ mit den gelisteten Termen und wendet dann modus ponens (`impE`) an.

Mit der `arith` Prozedur wird die Sequenz in einem Schritt gelöst.

Aufgabe 2.2 (Theoretisches Fundament von `arith`)

2.2–a Zeigen Sie, daß eine Sequenz $\Gamma \vdash G_1 \vee \texttt{..} \vee G_n$ genau dann gültig ist, wenn die Sequenz $\Gamma\texttt{,}\neg G_1\texttt{,..,}\neg G_n \vdash \mathsf{ff}$ gültig ist, sofern alle Formeln G_i entscheidbar sind.

2.2–b Zeigen Sie, daß entscheidbare Formeln abgeschlossen sind unter den aussagenlogischen Konnektiven $\neg$, $\wedge$, $\vee$, $\Rightarrow$.

2.2–c Es sei $\Gamma = v_1 \geq u_1 + c_1, \ldots, v_n \geq u_n + c_n$ eine Menge von atomaren arithmetischen Formeln, wobei die v_i und u_i Variablen (oder `0`) und die c_i ganzzahlige Konstanten sind, und $\mathcal{G}$ der Graph, der die Ordnungsrelation zwischen den Variablen von Γ beschreibt.

Zeigen Sie, daß Γ genau dann widersprüchlich ist, wenn $\mathcal{G}$ einen positiven Zyklus besitzt.

Aufgabe 2.3 (Grundlagen der `arith`-Prozedur)

2.3–a Beschreiben Sie einen Algorithmus, der überprüft, ob ein gerichteter Graph einen positiven Zyklus besitzt.

2.3–b Beschreiben Sie (informal) eine Taktik, welche elementar-arithmetische Formeln in konjunktive Normalform (als Vorbereitung für `arith`) umwandelt.

Aufgabe 2.4 (Die Mühen des Beweisens ohne Entscheidungsprozeduren II)

Beweisen Sie die Gültigkeit der Formel

$$\texttt{.., a=f(b)}\in\mathbb{Z}\texttt{, c=f(f(b))}\in\mathbb{Z} \vdash \texttt{h(g(a,f(c)),f(a)) = h(g(a,f(f(a))),c)} \in \mathbb{Z}$$

ohne Verwendung der `equality` Regel. Verwenden Sie hierzu Elementartaktiken wie `D, HypSubst, Declaration` und Tacticals wie `THEN`, `Repeat` etc. und gehen Sie davon aus, daß die Tactic `Wf` alle bei `HypSubst` anfallenden Wohlgeformtheitsziele lösen kann.

Lösung 2.0 *Ziele dieses Übungsblattes ist es, den praktischen Nutzen zu erkennen, den Entscheidungsprozeduren schon in relativ leichten Problemstellungen bringen. Außerdem sollen einige Hintergründe der Entscheidungsprozeduren genauer betrachtet werden.*

Lösung 2.1

Der Beweis verwendet sehr elementare Gesetze. Sie wurden während der Beweisführung formuliert.

```
x:ℤ, y:ℤ, 4-y≤1-x, y-2<6 ⊢ x<7
BY InstLemma ‘sub_le_r‘ [4-y,1,x]
\
 x:ℤ, y:ℤ, 4-y+x≤1, y-2<6 ⊢ x<7
 BY InstLemma ‘sub_add_comm‘ [4,y,x]
 \
  x:ℤ, y:ℤ, 4+x-y≤1, y-2<6 ⊢ x<7
  BY InstLemma ‘sub_le_l‘ [4+x,1,y]
  \
   x:ℤ, y:ℤ, 4+x≤1+y, y-2<6 ⊢ x<7
   BY InstLemma ‘le_less‘ [y-2,6]
   \
    x:ℤ, y:ℤ, 4+x≤1+y, y-2≤6-1 ⊢ x<7
    BY InstLemma ‘sub_6_1‘ []
    \
     x:ℤ, y:ℤ, 4+x≤1+y, y-2≤5 ⊢ x<7
     BY InstLemma ‘sub_le_l‘ [y,5,2]
     \
      x:ℤ, y:ℤ, 4+x≤1+y, y≤5+2 ⊢ x<7
      BY InstLemma ‘add_5_2‘ []
      \
       x:ℤ, y:ℤ, 4+x≤1+y, y≤7 ⊢ x<7
       BY InstLemma ‘add_comm‘ [4,x]
       \
        x:ℤ, y:ℤ, x+4≤1+y, y≤7 ⊢ x<7
        BY InstLemma ‘add_comm‘ [1,y]
        \
         x:ℤ, y:ℤ, x+4≤y+1, y≤7 ⊢ x<7
         BY InstLemma ‘sub_le_r‘ [x,4,y+1]  *** reversed ***
         \
          x:ℤ, y:ℤ, x≤y+1-4, y≤7 ⊢ x<7
          BY InstLemma ‘le_add_mono‘ [y,7,1-4]
          \
           x:ℤ, y:ℤ, x≤y+1-4, y+1-4≤7+1-4 ⊢ x<7
           BY InstLemma ‘le_trans‘ [y,23,55]
           \
            x:ℤ, y:ℤ, x≤7+1-4 ⊢ x<7
            BY InstLemma ‘add_7_1‘ []
            \
             x:ℤ, y:ℤ, x≤8-4 ⊢ x<7
             BY InstLemma ‘sub_8_4‘ []
             \
              x:ℤ, y:ℤ, x≤4 ⊢ x<7
              BY InstLemma ‘le_less‘ [x,7]  *** reversed ***
              \
               x:ℤ, y:ℤ, x≤4 ⊢ x≤7-1
               BY InstLemma ‘sub_7_1‘ []
               \
                x:ℤ, y:ℤ, x≤4 ⊢ x≤6
                BY InstLemma ‘le_pos_add‘ [x,4,2]
                |\
                | x:ℤ, y:ℤ, x≤4+2 ⊢ x≤6
                | BY InstLemma ‘add_4_2‘ []
                | \
                |  x:ℤ, y:ℤ, x≤6 ⊢ x≤6
                |  BY hypothesis 3
                 \
                  x:ℤ,y:ℤ, x<5, y≤7 ⊢ 0≤2
                  BY InstLemma ‘pos_2‘ []
```

Die folgenden Lemmata spielen hierbei eine Rolle. Sie müssten nun noch mit den elementareren Regeln ganzer Zahlen (meist mit Induktion) bewiesen werden. Die Gezetze der Konstantenarithmetik müssten hierbei auf die interne Darstellung der Kosntanten zurückgreifen. All dies ist implizit in der Prozedur `arith` enthalten.

```
sub_le_r     :     ∀a,b,c:ℤ. a≤b-c ⇔ a+c≤b
sub_le_l     :     ∀a,b,c:ℤ. a-c≤b ⇔ a≤b+c
sub_add_comm :     ∀a,b,c:ℤ. a-b+c = a+c-b
add_comm     :     ∀a,b:ℤ.   a+b = b+a
le_less      :     ∀a,b:ℤ.   a<b ⇔ a≤b-1
le_add_mono  :     ∀a,b,c:ℤ. a≤b ⇒ a+c≤b+c
le_pos_add   :     ∀a,b,c:ℤ. 0≤c ∧ a≤b ⇒ a≤b+c
le_trans     :     ∀a,b,c:ℤ. a≤b ∧ b≤c ⇒ a≤c
sub_6_1 :           6-1 = 5
sub_7_1 :           7-1 = 6
sub_8_4 :           8-4 = 4
add_4_2 :           4+2 = 6
add_5_2 :           5+2 = 7
add_7_1 :           7+1 = 8
pos_2   :           0 ≤ 2
```

Lösung 2.2 *Ziel dieser Aufgabe ist es, die theoretischen Fundamente der* `arith`*-Prozedur zu überprüfen und dafür bekannte Ergebnisse aus verschiedenen Teilgebieten der Informatik im Sinne eines neuen Verwendungszwecks zusammenzusetzen.*

2.2–a Dies ist Standard-Ergebnis der Logik.

Wegen der Entscheidbarkeit der G_i ist auch $G_1 \vee \ldots \vee G_n$ entscheidbar (siehe Teilaufgabe (b)) und damit gilt $G_1 \vee .. \vee G_n \;\Leftrightarrow\; \neg\neg(G_1 \vee .. \vee G_n) \;\Leftrightarrow\; \neg(\neg G_1 \wedge .. \neg \wedge G_n)$ auch in einer konstruktiven Logik. Der Rest entspricht dann einer Anwendung der Einführungsregel für $\neg$ und der Eliminationsregel für $\wedge$.

2.2–b Der Beweis ist ein Standardresultat der theoretischen Informatik: Wir müssen nur zeigen, daß aus der Entscheidbarkeit von beliebigen Formeln A und B die Entscheidbarkeit von $A \wedge B$, $A \vee B$, $A \Rightarrow B$ und $\neg A$ folgt.

Eine Formel A mit freien Variablen $x_1, .., x_n$ ist entscheidbar, wenn es eine berechenbare totale(!) Funktion χ_A:$\mathbb{N}^n \to \mathbb{N}$ (die *charakteristische Funktion* von A) gibt mit der Eigenschaft

$\chi_A(x_1, .., x_n) = 0$ genau dann, wenn $A[x_1, .., x_n]$ gültig ist.

(Man könnte auch $\chi_A(x_1, .., x_n) = 1$ fixieren, aber mit 0 ist es praktischer.)

Es seien χ_A und χ_B die charakteristischen Funktionen von A und B mit (o.B.d.A.) freien Variablen $x_1, .., x_n$. Dann ergeben sich folgende charakteristische Funktionen

$A \wedge B$: $\lambda x_1, .., x_n . \chi_A(x_1, .., x_n)$ `+` $\chi_B(x_1, .., x_n)$

$A \vee B$: $\lambda x_1, .., x_n . \chi_A(x_1, .., x_n)$ `*` $\chi_B(x_1, .., x_n)$

$A \Rightarrow B$: $\lambda x_1, .., x_n .$ `(1-`$\chi_A(x_1, .., x_n)$`)` `*` $\chi_B(x_1, .., x_n)$

$\neg A$: $\lambda x_1, .., x_n .$ `1-`$\chi_A(x_1, .., x_n)$ (Man beachte, daß x-y=0 ist, wenn x<y gilt.)

Dies ist der *Beweis* dafür, daß die Theorie $\mathcal{A}$ prinzipiell entscheidbar ist. Prinzipiell ist hiermit auch ein Verfahren zur Überprüfung elementar-arithmetischer Formeln gegeben. Dieses aber wäre extrem ineffizient. Beweis und Entscheidungsverfahren sind verschiedene Aspekte: ein einfacher Beweis liefert oft nur ein komplexes Verfahren und umgekehrt.

2.2–c Das eigentliche Argument ist verhältnismäßig einfach. Γ ist genau dann widersprüchlich, wenn aus Γ für eines der v_i folgt, daß $v_i > v_i$ ist. Dies bedeutet, daß es eine Kette von Ungleichungen $v_i \geq x_1 + g_1$, $x_i \geq x_{i+1} + g_{i+1}$, ..., $x_k \geq v_i + g_{k+1}$ aus Γ geben muß mit $\sum_{i=1}^{k+1} g_i > 0$. Per Konstruktion ist dies genau dann der Fall, wenn es im Ordnungsgraphen eine Serie von Kanten $[v_i \xrightarrow{g_1} x_1, x_1 \xrightarrow{g_2} x_2, .., x_k \xrightarrow{g_{k+1}} v_i]$ gibt mit Gewicht $\sum_{i=1}^{k+1} g_i > 0$, also einen positiven Zyklus von v_i nach v_i.

Im Detail wird der Beweis relativ aufwendig. Einen ausführlichen Beweis findet man im Artikel "*An algorithm for checking PL/CV arithmetic inferences*", der im Appendix D des folgenden Buches erschienen ist, auf Seite 238ff.

Robert L. Constable, Scott D. Johnson, and Carl D. Eichenlaub. *Introduction to the PL/CV2 Programming Logic*, Lecture Notes in Computer Science 135, Springer Verlag, 1982.

Lösung 2.3 *Auch in dieser Aufgabe sollen bekannte Ergebnisse aus verschiedenen Teilgebieten der Informatik neu zusammengesetzt werden.*

2.3–a Der naive Algorithmus, schrittweise alle Pfade der Länge 1.. $|V|$ zu erzeugen und das Gewicht der identifizierten Zyklen zu berechnen, ist zu ineffizient, da bei maximalem Verzweigungsgrad k bis zu $k^{|V|}$ Pfade erzeugt werden.

Da die einzelnen Pfade für die konkrete Fragestellung ("gibt es einen positiven Zyklus") überhaupt nicht relevant sind, bietet es sich an, global die *maximalen Gewichte* aller Pfade zu bestimmen, die in einem Knoten v_i beginnen und in einem Knoten v_j des Graphen enden. Anschließend prüft man, ob das maximale Gewicht eines Pfad von v_i nach v_i für ein i positiv ist.

Für die Berechnung der maximalen Gewichte bietet sich ein Pfadanalyseverfahren an, das in ähnlicher Form schon einmal in der Theoretischen Informatik zur Umwandlung von Automaten in reguläre Ausdrücke vorgestellt wurde. Man definiere $C_{i,j}^k$ als das maximale Gewicht eines Pfades von v_i nach v_j bei dem (außer v_i und v_j selbst) nur die Knoten $v_1..v_k$ durchlaufen werden dürfen.

Dabei ist $C_{i,j}^0$ das Gewicht der Kante von v_i nach v_j, falls eine solche existiert, und -∞ andernfalls. Man macht sich schnell klar, daß $C_{i,j}^{k+1}$ entweder identisch ist mit $C_{i,j}^k$ oder das maximale Gewicht eines Pfades von v_i nach v_j, der über v_k läuft – also die Summe von $C_{i,k}^k$ und $C_{k,j}^k$, je nachdem, welches Gewicht höher ist.

Integriert man nun den Test, ob ein positiver Zyklus gefunden wurde, in den Algorithmus, so ergibt sich folgendes Programmstück (in ALGOL-ähnlicher Notation)

```
FOR i:=1 TO n DO
  FOR j:=1 TO n DO
    A_{i,j} := C^0_{i,j};
poscycle := false;

FOR k:=1 TO n WHILE NOT poscycle DO
  FOR i:=1 TO n WHILE NOT poscycle DO
    FOR j:=1 TO n WHILE NOT poscycle DO
      A_{i,j} := max(A_{i,j}, A_{i,k}+A_{k,j});
      poscycle := j=i AND A_{i,j}>0;
```

Die Laufzeit des Algorithmus ist kubisch und daher können problemlos Graphen von bis zu

1000 Knoten analysiert werden, während bei dem naiven Verfahren die Graphengröße maximal 30 sein darf.

Weitere Details findet man im Artikel "*An algorithm for checking PL/CV arithmetic inferences*" auf Seite 241ff.

2.3–b Im Prinzip muß die Taktik nur die folgenden Konversionen der Reihe nach rekursiv anwenden.

$$
\begin{array}{lll}
A \Rightarrow B & \mapsto & \neg A \vee B \\
\neg(A \wedge B) & \mapsto & \neg A \vee \neg B \\
\neg(A \vee B) & \mapsto & \neg A \wedge \neg B \\
A \vee (B \wedge C) & \mapsto & A \vee B \ \wedge \ A \vee C \\
\neg\neg A & \mapsto & A
\end{array}
\qquad
\begin{array}{lll}
\neg(s{=}t) & \mapsto & s{\neq}t \\
\neg(s{\neq}t) & \mapsto & s{=}t \\
\neg(s{<}t) & \mapsto & s{\geq}t \\
\neg(s{>}t) & \mapsto & s{\leq}t \\
\neg(s{\geq}t) & \mapsto & s{<}t \\
\neg(s{\leq}t) & \mapsto & s{>}t
\end{array}
$$

Im Detail ist dies aber relativ mühsam, da eine Substitutionsregel für logische Äquivalenzen nicht existiert. So muß man sich damit behelfen, die Substitution separat auszurechnen und die modifizierte Formel mit der Regel `cut` einzuführen.

Man betrachtet hierzu die Formel von außen und analysiert ihre Struktur. Im ersten Schritt sucht man in der Konklusion C nach Teilformeln der Gestalt $A \Rightarrow B$, sobald eine solche gefunden wird, ersetzt man diese durch $\neg A \vee B$. Man erhält somit eine neue Formel F und ruft die Regel `cut (-1)` F auf.

Im ersten Teilziel ist nun $\vdash F$ zu zeigen und man wiederholt die Konversion, bis keine Formeln der Teilformeln der Gestalt $A \Rightarrow B$ mehr da sind. Anschließend konvertiert man Teilformeln der Gestalt $\neg(A \wedge B)$ usw.

Im zweiten Teilziel muß man $F \vdash C$ zeigen. Hierzu kann man die Taktik `simple_prover` einsetzen, die genau zu dem Teilziel $\neg A \vee B \vdash A \Rightarrow B$ führen wird, da F und C bis auf diese Teilformeln ja identisch sind und nur zerlegt werden müssen. Nun benötigt man ein Lemma der Gestalt $\forall$`A,B:`$\mathbb{P}_1$`. Decidable(A)` $\wedge$ `Decidable(B)` $\Rightarrow A \Rightarrow B \Leftrightarrow \neg A \vee B$ und Lemmata über die Entscheidbarkeit elementar-arithmetischer Formeln (siehe Aufgabe 2.2). Damit läßt sich dann das restliche Teilziel beweisen

Dieses Verfahren ist natürlich zu ineffizient um in der Praxis Einzug zu finden. Jedoch ist hiermit gezeigt, daß `arith` – die ja nur auf Disjunktionen atomarer Formeln anwendbar ist – zu einer vollständigen Entscheidungsprozedur für die Theorie $\mathcal{A}$ ergänzt werden kann.

In der Praxis ist es effizienter, auf die Normalisierung zu verzichten und die zu untersuchenden Formeln von Hand freizulegen. Im Prinzip braucht man dazu nur die Tactic `prover` so zu erweitern, daß in `simple_prover` der Aufruf von `arith` früh integriert wird. Hierdurch werden Implikationen, Negationen und Konjunktionen automatisch zerlegt und Disjunktionen separat verfolgt.

Lösung 2.4 Da alle Wohlgeformtheitsziele durch `Wf` gelöst werden, wird der Übersichtlichkeit wegen der Typ $\mathbb{Z}$ im folgenden weggelassen.

```
.., a=f(b), c=f(f(b)) ⊢ h(g(a,f(c)),f(a)) = h(g(a,f(f(a))),c)
BY HypSubst  (-1)  0 THEN Wf
\.., a=f(b), c=f(f(b)) ⊢ h(g(a,f(f(f(b)))),f(a)) = h(g(a,f(f(a))),f(f(b)))
 BY HypSubst  (-2)  0 THEN Wf
 \.., a=f(b), c=f(f(b)) ⊢ h(g(f(b),f(f(f(b)))),f(f(b))) = h(g(f(b),f(f(f(b)))),f(f(b)))
  BY D 0
  |\
  | .... ⊢ h=h
  | BY Declaration
   \
    .... ⊢ (g(f(b),f(f(f(b)))),f(f(b))) = (g(f(b),f(f(f(b)))),f(f(b)))
    BY D 0
    |\
    | .... ⊢ g(f(b),f(f(f(b)))) = g(f(b),f(f(f(b))))
    | BY D 0 THENL [Declaration; D 0]
    | |\
    | | .... ⊢ f(b) = f(b)
    | | BY D 0 THEN Declaration
    |  \
    |   .... ⊢ f(f(f(b))) = f(f(f(b)))
    |   BY Repeat (D 0 Orelse Declaration)
    \
     .... ⊢ f(f(b)) = f(f(b))
     BY  Repeat (D 0 Orelse Declaration)
```

Die Vorgehensweise ist ganz schematisch und läßt sich leicht programmieren, da im wesentlichen Terme innerhalb von Gleichheiten zerlegt werden (Dekomposition). Im Prinzip wird nach den Substitution nur noch zerlegt bzw. auf die Typdeklarationen zugegriffen.

Etwas ineffizient wirkt, daß die Substitutionen jedes Vorkommen von `c` bzw. `a` ersetzen, auch wenn dies nicht erforderlich ist. Effizienter ist ein Beweis, der mit Dekompositionen beginnt.

```
.., a=f(b), c=f(f(b)) ⊢ h(g(a,f(c)),f(a)) = h(g(a,f(f(a))),c)
BY Repeat (D 0 Orelse Declaration)
|\
| .., a=f(b), c=f(f(b)) ⊢ a=a
|\
| .., a=f(b), c=f(f(b)) ⊢ c=f(a)
 \.., a=f(b), c=f(f(b)) ⊢ f(a)=c
```

Das erste Teilziel wird eigentlich von `Declaration` gelöst wird. Die anderen Teilziele sind i.w. identisch, so daß es sich lohnt, den Beweis mit `Assert f(a)=c` zu beginnen.

```
.., a=f(b), c=f(f(b)) ⊢ h(g(a,f(c)),f(a)) = h(g(a,f(f(a))),c)
BY Assert f(a)=c
|\
| .., a=f(b), c=f(f(b)) ⊢ f(a)=c
| BY HypSubst  (-1)  0 THEN Wf THEN HypSubst  (-2)  0 THEN Wf
| \
|  ...., ⊢ f(f(b)) = f(f(b))
|  BY Repeat (D 0 Orelse Declaration)
 \
  ....,  f(a)=c ⊢  h(g(a,f(c)),f(a)) = h(g(a,f(f(a))),c)
  BY HypSubst  (-1)  0 THEN Wf
  \
   ....,  ⊢  h(g(a,f(c)),c)) = h(g(a,f(c)),c)
   BY Repeat (D 0 Orelse Declaration)
```

Bei dieser Beweisführung ist der interne Beweis der kürzeste. Da aber insgesamt mehr einzugeben ist, werden die meisten Benutzer zur ersten Lösung tendieren und als dritte Regel `Repeat (D 0 Orelse Declaration)` eingeben.