Übung zur Vorlesung

Automatisierte Logik und Programmierung II

Prof. Chr. Kreitz

Universität Potsdam, Theoretische Informatik — Sommersemester 2014

Blatt 1 — Abgabetermin: 7.05.2014

Aufgabe 1.1 (ML Spielereien - Funktionen, die später noch gebraucht werden)

Programmieren Sie die folgenden elementaren ML Funktionen

1.1–a `member: * -> * list -> bool`
testet, ob ein Element `x` in einer Liste `l` vorkommt.

1.1–b `select: int -> * list -> *`
bestimmt das `n`-te Element einer Liste `l`.

1.1–c `replicate: * -> int -> * list`
erzeugt bei Eingabe von `x` und `n` die `n`-elementige Liste `[x;...x]`

1.1–d `find: (*-> bool) -> * list -> *`
findet bei Eingabe einer Funktion `f` und einer Liste `l` das erste Element `x` von `l`, für das `f(x)=true` ist.

1.1–e `map: (*->**) -> * list -> ** list`
wendet eine Funktion `f` auf alle Elemente einer Liste `l` an.

1.1–f `lookup: (*#**) list -> * -> **`
findet bei Eingabe eines Labels `x` das erste Element `y`, für das ⟨`x,y`⟩ in einer Liste `l` liegt.

Aufgabe 1.2 (Einbettung von Logikregeln als Taktiken)

In Einheit 9 hatten wir die Curry-Howard Isomorphie zwischen den logischen Regeln und den Regeln der Typentheorie besprochen und die Logikoperatoren als definitorische Erweiterung der Sprache der Typentheorie eingeführt. Wir wollen nun auch das Inferenzsystem der Typentheorie um logische Inferenzregeln erweitern.

1.2–a Überlegen Sie zunächst, welche Schritte notwendig sind, um die Inferenzregeln der Typentheorie in Taktiken umzuwandeln, welche die logischen Inferenzregeln nachbilden.

Sie dürfen davon ausgehen, daß Basistaktiken wie `D 0` die logischen Definitionen wie `and`, `or`, `implies` etc. automatisch auffalten und dann das entsprechende typentheoretische Konstrukt (Produkt, Disjunkte Vereinigung, Funktionenraum, ...) zerlegen. Sie müssen aber verhindern, daß eine Regel wie `andR` auch auf eine Implikation anwendbar ist.

Da 15 Inferenzregeln zu programmieren sind, lohnt sich eine Higher-Order Konzeption. Die folgenden Teilaufgaben beschreiben eine mögliche Vorgehensweise

1.2–b Programmieren Sie ein Tactical
`TryOn: ((int -> tactic) -> int -> (term -> bool) -> tactic`,
das bei Eingabe einer (parametrisierten) Taktik tac, einer Klauselummer i (0 steht für die Konklusion), und einer Testfunktion $test$ auf Termen zunächst überprüft, ob der Term in Klausel i den Test erfüllt, im Erfolgsfall die Taktik tac auf Klausel i anwendet und im Mißerfolgsfall mit Fehlermeldung abbricht.

Die übliche Art der Definition einer solchen wäre `let TryOn tac i test p = ...`, wobei `p` für das aktuelle Beweisobjekt steht, das Sie ggf. mit Funktionen wie `conclusion` oder `hypotheses` analysieren müssen. Für die Erzeugung einer Fehlermeldung können Sie die Taktik `Fail` verwenden. Achten Sie darauf, daß `TryOn tac i test` den Typ `tactic` hat.

1.2–c Programmieren Sie beispielhaft die Testfunktion
`is_and_term: term -> bool`,
welche bei Eingabe eines Terms t prüft, ob es sich um eine Konjunktion handelt.

1.2–d Implementieren Sie mit Hilfe von `TryOn`, der Taktik `D`, den Funktionen zum Einfügen von Steuerungsparametern und Testfunktionen wie der obigen die Regeln der Prädikatenlogik als Taktiken. Programmieren Sie insbesondere die Regeln `andL`, `orR1`, exR und `allL`.

Um eventuell entstandene Wohlgeformtheitsziele zu bearbeiten, können Sie nach Ausführung der Basistaktik das Konstrukt `THENW Auto` verwenden. Das Tactical `THENW` arbeitet ähnlich wie `THEN`, wendet die zweite Taktik aber nur auf die verbleibenden Wohlgeformtheitsziele der ersten an. Die Taktik `D` markiert derartige Ziele.

Aufgabe 1.3

In Einheit 13 haben wir die Tacticals `Try`, `Progress` und `Repeat` beschrieben. Geben Sie eine ML-Implementierung dieser Tacticals an.

Aufgabe 1.4 (Die Mühen des Beweisens ohne Entscheidungsprozeduren)

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

`x:ℤ, y:ℤ, 4-y≤1-x, y-2<6 ⊢ 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≤1-x` in einem Schritt in `4+x≤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 $\forall x_1 : T_1 \,..\, x_n : T_n \,.\, P \Rightarrow Q$ mit den gelisteten Termen und wendet dann modus ponens (`imp`) an.

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

Aufgabe 1.5 (Theoretisches Fundament von `arith`)

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

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

1.5–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 1.6 (Grundlagen der `arith`-Prozedur)

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

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

Lösung 1.1 *Ziel dieser Aufgabe ist es, Erfahrungen im Umgang mit* ML *zu sammeln*

1.1–a `member: * -> * list -> bool`: Es gibt mehrere Lösungsmöglichkeiten

```
letrec member x l = if l=[] then false
                            else if x = hd l then true
                                             else member x (tl l)
                    ;;
letrec member x l = let a.rest = l
                    in  if x = a then true
                                 else member x rest
                    ?  false
                    ;;
letrec member x l = let a.rest = l
                    in  (x = a) or (member x rest)
                    ?  false
                    ;;
```

1.1–b `select: int -> * list -> *`

```
letrec select pos list = if pos = 1
                         then hd list
                         else select (pos-1) (tl list)
                       ;;
```

1.1–c `replicate: * -> int -> * list`

```
let replicate x n  = if n<0 then failwith `replicate`
                            else letrec aux x n res =
                                    if n=0 then res
                                           else aux x (n-1) (x.res)
                                 in  aux x n []
                     ;;
```

1.1–d `find: (*-> bool) -> * list -> *`

```
letrec find f l =  if l=[] then fail
                           else if f (hd l) then hd l
                                            else find f (tl l)
                   ;;
letrec find f l =  let a.rest = l
                   in if f a then a
                             else find f rest
                   ;;
```

Der Fehler wird im zweiten Fall durch das fehlschlagende Matching erzeugt.

1.1–e `map: (*->**) -> * list -> ** list`

```
letrec map f l = if l = if l=[] then []
                                else (f (hd l)) .  (map f (tl l))
                 ;;
letrec map f l = if l = let a.rest = l
                        in   (f a) .  (map f rest)
                        ?  []
                        ;;
```

1.1–f `lookup: (*#**) list -> * -> **`

```
let lookup table x   = let eqfst x (y,z) = (x=y)
                       in  snd (find (eqfst x) table)
                       ;;
```

Lösung 1.2 *Ziel dieser Aufgabe ist es, einfache Taktiken zu konstruieren und vorbereitende Tacticals zu schreiben, welche die Programmierung erleichtern. Geübt werden soll dabei auch die funktionale Denkweise: Tacticals sind Funktionen, die Funktionen (Taktiken) in Funktionen umwandeln. Funktionsdefinitionen brauchen nur so viele Parameter, wie für eine eindeutige Beschreibung nötig.*

1.2–a ergibt sich aus der Programmierung

1.2–b
```
let TryOn tac i test p =
  let t = (if i=0 then conclusion p
                  else select (get_pos_hyp_num i p) (hypotheses p))
  in if test t then tac i p
               else Fail p
```

1.2–c `let is_and_term t =  let opid,parm,bterms = dest_term t in opid = 'and'`

1.2–d Bei der Verwendung von `Sel` und `With` müssen wir die Argumente etwas umsortieren um typkorrekt zu bleiben.

```
let SEL j tac i  = Sel j (tac i)
and WITH t tac i = With t (tac i)
;;
let andR       = TryOn D 0           is_and_term
and orR1       = TryOn (SEL 1 D) 0   is_or_term     THENW Auto
and orR2       = TryOn (SEL 2 D) 0   is_or_term     THENW Auto
and impR       = TryOn D 0           is_imp_term    THENW Auto
and falseR     = TryOn D 0           is_false_term  THENW Auto
and notR       = TryOn D 0           is_not_term    THENW Auto
and allR       = TryOn D 0           is_all_term    THENW Auto
and exR t      = TryOn (WITH t D) 0  is_ex_term
and andL   i   = TryOn D i           is_and_term
and orL    i   = TryOn D i           is_or_term
and falseL i   = TryOn D i           is_false_term
and exL    i   = TryOn D i           is_ex_term
and allL   i t = TryOn (WITH t D) i  is_all_term    THENW Auto
and impL   i   = TryOn D i           is_imp_term
and notL   i   = TryOn D i           is_not_term
```

Lösung 1.3 *Ziel dieser Aufgabe ist es, imperative Vorgehensweisen in einer funktionalen Programmiersprache auszudrucken.*

```
let Try (tac:tactic) = tac ORELSE Id ;;

let Progress (tac:tactic) (pf:proof) =
 let proofs, val = tac pf in
   if length proofs=1
      then let subgoal=hd proofs in
               if (hypotheses subgoal) = (hypotheses pf) &
                  (conclusion subgoal) = (conclusion pf)
                  then failwith 'NO PROGRESS'
                  else proofs, val
      else proofs, val
;;

letrec Repeat (tac:tactic)  = (Progress tac  THEN  Repeat tac) ORELSE Id  ;;
```

Lösung 1.4

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 1.5 *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.*

1.5–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 $\wedge$.

1.5–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 $\chi_A{:}\mathbb{N}^n{\to}\mathbb{N}$ (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

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

$A \vee B$: $\lambda x_1, .., x_n \,.\, \chi_A(x_1, .., x_n) \;\star\; \chi_B(x_1, .., x_n)$

$A \Rightarrow B$: $\lambda x_1, .., x_n \,.\, (1{-}\chi_A(x_1, .., x_n)) \;\star\; \chi_B(x_1, .., x_n)$

$\neg A$: $\lambda x_1, .., x_n \,.\, 1{-}\chi_A(x_1, .., x_n)$ (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.

1.5–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 $v_i \geq x_1 + g_1, x_i \geq x_{i+1} + g_{i+1}, \ldots, x_k \geq v_i + g_{k+1}$ 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 1.6 *Auch in dieser Aufgabe sollen bekannte Ergebnisse aus verschiedenen Teilgebieten der Informatik neu zusammengesetzt werden.*

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

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

$A \Rightarrow B$	$\mapsto \neg A \vee B$	$\neg(s{=}t)$	$\mapsto s{\neq}t$
$\neg(A \wedge B)$	$\mapsto \neg A \vee \neg B$	$\neg(s{\neq}t)$	$\mapsto s{=}t$
$\neg(A \vee B)$	$\mapsto \neg A \wedge \neg B$	$\neg(s{<}t)$	$\mapsto s{\geq}t$
$A \vee (B \wedge C)$	$\mapsto A \vee B \ \wedge \ A \vee C$	$\neg(s{>}t)$	$\mapsto s{\leq}t$
$\neg\neg A$	$\mapsto A$	$\neg(s{\geq}t)$	$\mapsto s{<}t$
		$\neg(s{\leq}t)$	$\mapsto s{>}t$

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.