Übung zur Vorlesung

# Automatisierte Logik und Programmierung II

Prof. Chr. Kreitz

Universität Potsdam, Theoretische Informatik — Sommersemester 2009

Blatt 1 — Abgabetermin: 15.05.2009

## Aufgabe 1.1 (ML Spielereien)

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

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

**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  ;;
```