

Automatisierte Logik und Programmierung II

Prof. Chr. Kreitz

Universität Potsdam, Theoretische Informatik — Sommersemester 2006

Blatt 1 — Abgabetermin: —

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 Programmieren Sie mit Hilfe der Taktik `Unfold` ein Tactical `CC: tactic -> tactic`, welches eine angegebene Taktik zur Anwendung bringt, nachdem zuvor eine Definition in der Konklusion einer Sequenz aufgelöst wurde.

Programmieren Sie unter Verwendung der Taktiken `Fold` und `Unfold` ein Tactical `CH: (int->tactic) -> int -> tactic`, welches dasselbe für Taktiken leistet, die auf Hypothesen operieren.

1.2-b Implementieren Sie mit Hilfe von `CC` und `CH` die Regeln der Prädikatenlogik als Taktiken.

Hinweis: Bedenken Sie beim Auffalten, daß Eliminationsregeln oft eine Kopie der eliminierten Hypothese stehen lassen. hinterlassen, Universenlevel nicht anzugeben sind, und – außer bei den Quantoren – keine Variablendeklarationen hinterlassen.

Verwenden Sie die Variable `Umax` um das maximale Universenlevel zu bestimmen, wo dies nötig ist, und die vordefinierte Taktik `Wf`, um Wohlgeformtheitsziele beweisen zu lassen. Mit der Taktik `mkInv i` können Sie die deklarierte Variable der Hypothese `i` unsichtbar machen (es entsteht eine Variable `%x...`, die in Deklarationen vom Display-Mechanismus unterdrückt wird).

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 = [] then []
                 else (f (hd l)) . (map f (tl l))
;;
letrec map f l = if l = [] then []
                 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 enthalten, wie für eine eindeutige Beschreibung nötig ist.

1.2-a Um CC und CH zu programmieren, muß man jeweils die Top-Level Definition in Konklusion bzw. in der genannten Hypothese `pos` auflösen. Deren Name ist genau der Name des Operators, den man durch Anwendung der Funktion `opid` auf den entsprechenden Term (`conclusion pf` bzw. `type_of_hyp pos pf`) erhält. Bei CH muß ggf. die Hypothese wieder zurückgefaltet werden.

```

let CC tactic pf      = let defname = opid (conclusion pf)
                        in
                        (      Unfold defname 0
                          THEN tactic
                        ) pf
                        ;;

let type_of_hyp pos pf = snd (dest_declaration (select pos (hypotheses pf))) ;;

let CH tactic pos pf  = let defname = opid (type_of_hyp pos pf)
                        in
                        (      Unfold defname pos
                          THEN tactic pos
                          THEN Try (Fold defname pos)
                        ) pf
                        ;;

```

Die Lösung für CH funktioniert in der angegebenen Form nur für Taktiken, die keine große Umsortierung der Hypothesen vornehmen. Andernfalls wird eventuell die falsche Hypothese zurückgefaltet. Eine präzisere Lösung würde sich beim Auf falten den Variablenamen der aufgefalteten Hypothese (der ändert sich normalerweise nicht!) merken und die entsprechende Hypothese am Ende wieder aufspüren und zurückfalten.

1.2-b Die Implementierungen sind naheliegend, da nur die zugehörigen typentheoretischen Regeln mit CC (bzw. CH) aufgerufen werden müssen. Zuweilen müssen Wohlgeformtheitsziele behandelt werden, da man diese nicht mehr sehen möchte.

```

let andI      = CC independent_pairFormation      ;;
let orI1      = CC (inlFormation Umax) THENL [Id;Wf] ;;
let orI2      = CC (inrFormation Umax) THENL [Id;Wf] ;;
let impI      = CC (lambdaFormation Umax) THENL [mkInv (-1);Wf];;
let notI      = CC impI                           ;;
let allI      = CC (lambdaFormation Umax) THENL [Id;Wf] ;;
let exI term  = CC (dependent_pairFormation Umax term) THENL [Wf;Id;Wf] ;;

```

Man beachte, daß bei extrem komplizierten Datentypen die Taktik `Wf` nicht zum Erfolg

führt. Diese kommen im Normalfall aber nicht vor (üblicherweise werden sie vorher separat definiert und dabei als wohlgeformt nachgewiesen).

Wenn man ganz präzise ist, müsste man Regeln wie `lambdaFormation` zuvor mittels `refine` in eine Taktik umwandeln. Dies bedeutet aber auch, daß die Parameter einer regel korrekt eingefügt werden müssen, was wir in der Vorlesung nicht besprochen haben. Die tatsächliche Implementierung wäre somit

```

let exI term      = CC (refine (dependent_pairFormation
                               [mk_level_exp_arg Umax; mk_term_arg term]))
                               THENL [Wf;Id;Wf]      ;;

1.2-c let andE      pos =      CH productElimination pos
                               THEN thin pos
                               THEN mkInv pos
                               THEN mkInv (pos+1)      ;;
let orE          pos = CH unionElimination pos      ;;
let impE          pos =      CH independent_functionElimination pos
                               THENL [Id; thin pos THEN mkInv (-1)]      ;;
let falseE        pos = CH voidElimination pos      ;;
let notE          pos =      CH impE pos
                               THENL [Id;falseE (-1)]      ;;
let alle pos term =      CH (\h.dependent_functionElimination h term) pos
                               THENL [Wf; thin (-1) THEN mkInv (-1)]      ;;
let exE          pos =      CH productElimination pos
                               THEN thin pos
                               THEN mkInv (pos+1)      ;;

```

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 ;;

```