

Solutions for Problem Set 1

DPL Seminar, Summer 2001

Handout 3

Konstantine Arkoudas

July 12, 2001

Problem 1

We first issue some declarations and define a few auxiliary methods that will simplify life:

```
(domain Thing)
(declare P (-> (Thing) Boolean))
(declare Q (-> (Thing) Boolean))
(declare R (-> (Thing) Boolean))
(declare S (-> (Thing) Boolean))
(declare R2 (-> (Thing Thing) Boolean)) ;; A binary R
(declare Q2 (-> (Thing Thing) Boolean)) ;; A binary Q

;; A method for modus tollens:
(define (mt P Q)
  (dmatch [P Q]
    ([([if P1 P2] (not P2))] (suppose-absurd P1
                             (!absurd (!imp P P1) Q))))))

;; The ds method implements the so-called "disjunctive syllogism":
;; it takes two premises of the form P1  $\vee$  P2 and  $\sim$ P1 and derives P2.
(define (ds P Q)
  (dmatch [P Q]
    ([([or P1 P2] (not P1))]
      (dlet ((imp1 (assume P1
                       (!dn (suppose-absurd (not P2) (!absurd P1 (not P1))))))
             (imp2 (assume P2
                       (!claim P2))))
              (!claim P2))))
      (!cd P imp1 imp2))))))

;; A method that successively specializes a premise of the
;; form (forall x1 (forall x2 (... (forall xn P) ...))) with
;; a list of terms [t1 ... tn].
(define (uspec* P terms)
  (dmatch terms
    ([[]] (!claim P))
    ((list-of t rest) (!uspec* (!uspec P t) rest))))
```

Part a

```
(define premise1 (forall ?x (if (and (P ?x) (Q ?x)) (R ?x))))
(define premise2 (exists ?x (and (Q ?x) (not (R ?x)))))
(define conclusion (exists ?x (not (P ?x))))
(assert premise1 premise2)
(pick-witness w premise2)
(dlet ((w-property (and (Q w) (not (R w))))
      (w-is-not-P (suppose-absurd (P w)
                                   (!absurd (!imp (!uspec premise1 w)
                                                  (!both (P w) (!left-and w-property)))
                                       (!right-and w-property))))))
      (!egen conclusion w)))
```

Part b

```
(define premise1 (forall ?x (if (P ?x) (Q ?x))))
(define premise2 (exists ?x (and (R ?x) (not (Q ?x)))))
(define premise3 (forall ?x (if (R ?x) (or (P ?x) (S ?x)))))
(define conclusion (exists ?x (and (R ?x) (S ?x))))
(assert premise1 premise2 premise3)
```

```

(pick-witness w premise2
  (dlet ((w-property (and (R w) (not (Q w))))
    (L1 (!mt (!uspec premise1 w)
      (!right-and w-property)))
    (L2 (!dn (suppose-absurd (not (S w))
      (!absurd (!ds (!mp (!uspec premise3 w)
        (!left-and w-property))
          L1)
        (not (S w))))))
    (L3 (!both (!left-and w-property) L2)))
    (!egen conclusion w)))

```

Part c

```

(define premise1 (exists ?y (forall ?x (R2 ?x ?y))))
(assert premise1)
(define conclusion (forall ?x (exists ?y (R2 ?x ?y))))
(pick-witness w premise1
  (pick-any foo
    (dbegin (!uspec (forall ?x (R2 ?x w)) foo)
      (!egen (exists ?y (R2 foo ?y)) w))))

```

Part d

```

(define premise1 (forall ?x (forall ?y (if (and (P ?x) (Q ?y)) (R2 ?x ?y)))))
(define premise2 (forall ?x (forall ?y (if (and (P ?x) (R2 ?x ?y)) (S ?y)))))
(assert premise1 premise2)
(define conclusion (exists ?x (if (P ?x) (forall ?x (if (Q ?x) (S ?x))))))
(dbegin (assume (P ?foo)
  (pick-any a
    (assume (Q a)
      (dbegin (!mp (!uspec* premise1 [?foo a])
        (!both (P ?foo) (Q a)))
        (!mp (!uspec* premise2 [?foo a])
          (!both (P ?foo) (R2 ?foo a))))))
    (!egen conclusion ?foo))

```

Part e

```

(define premise1 (exists ?x (and (P ?x) (forall ?y (if (Q ?y) (R2 ?x ?y)))))
(define premise2 (forall ?x (forall ?y (if (and (P ?x) (R2 ?x ?y)) (S ?y)))))
(define premise3 (not (exists ?x (and (S ?x) (not (Q ?x)))))
(assert premise1 premise2 premise3)
(define conclusion (forall ?x (iff (Q ?x) (S ?x))))
(pick-any x
  (dlet ((L1 (assume (Q x)
    (pick-witness w premise1
      (dlet ((w-property (and (P w) (forall ?y (if (Q ?y) (R2 w ?y)))))
        (L1 ((P w) BY (!left-and w-property)))
        (L2 ((R2 w x) BY (!mp (!uspec (!right-and w-property) x) (Q x))))
        (!mp (!uspec* premise2 [w x]) (!both L1 L2))))))
    (L2 (assume (S x)
      (!dn (suppose-absurd (not (Q x))
        (dbegin (!both (S x) (not (Q x))
          (!absurd (!egen (exists ?x
            (and (S ?x) (not (Q ?x)))) x)
            premise3))))))
    (!equiv L1 L2)))

```

Part f

```
(define premise1 (exists ?x (and (P ?x) (forall ?y (if (and (P ?y) (R2 ?x ?y))
(Q2 ?y ?a))))))
(define premise2 (exists ?x (and (P ?x) (not (Q2 ?x ?a))))))
(define premise3 (exists ?x (and (not (P ?x)) (Q2 ?x ?a))))
(assert premise1 premise2 premise3)
(define conclusion (exists ?x (exists ?y (and (P ?x) (and (P ?y) (not (R2 ?x ?y)))))))
(pick-witness w1 premise1
  (dlet ((w1-property (and (P w1) (forall ?y (if (and (P ?y) (R2 w1 ?y))
(Q2 ?y ?a))))))
    (pick-witness w2 premise2
      (dlet ((w2-property (and (P w2) (not (Q2 w2 ?a))))
            (L1 ((if (and (P w2) (R2 w1 w2)) (Q2 w2 ?a)) BY
              (!uspec (!right-and w1-property) w2)))
            (L2 ((not (and (P w2) (R2 w1 w2))) BY
              (!mt L1 (!right-and w2-property))))
            (L3 ((not (R2 w1 w2)) BY
              (suppose-absurd (R2 w1 w2)
                (!absurd (!both (!left-and w2-property) (R2 w1 w2)) L2))))
            (L4 (!both (!left-and w1-property) (!both (!left-and w2-property) L3)))
            (L5 (!egen (exists ?y (and (P w1) (and (P ?y) (not (R2 w1 ?y)))) w2)))
              (!egen conclusion w1))))))
```

Problem 2

```
(define (rd L)
  (match L
    ((split L1 (list-of x (split L2 (list-of x L3)))) (rd (join L1 [x] L2 L3)))
    (_ L)))
```

Problem 3

We first define the `map` functional:

```
(define (map f L)
  (match L
    ([[] []]
      ((list-of x rest) (add (f x) (map f rest)))))
```

Then we define a function `rv-in-term` (“replace variable in term”) that replaces every occurrence of a variable `v` in a term `t` by some other term `new`; and a similar function `rv-in-prop` that replaces every *free* occurrence of a variable `v` in a proposition `P` by some term `new`:

```
(define (rv-in-term v new t)
  (match t
    ((some-var x) (check ((equal? x v) new)
      (else t)))
    (((some-symbol f) (some-list args))
      (make-term f (map (function (s) (rv-in-term v new s)) args))))))

(define (rv-in-prop v new P)
  (match P
    ((some-atom t) (rv-in-term v new t))
    ((not Q) (not (rv-in-prop v new Q)))
    (((some-prop-con pc) P1 P2) (pc (rv-in-prop v new P1) (rv-in-prop v new P2)))
    (((some-quant q) (val-of v) body) (q v body))
    (((some-quant q) x body) (q x (rv-in-prop v new body)))))
```

Finally, the definition of `rename-prop`:

```
(define (rename-prop P)
  (match P
    ((not Q) (not (rename-prop Q)))
    (((some-prop-con pc) P1 P2) (pc (rename-prop P1) (rename-prop P2)))
    (((some-quant q) x B) (let ((x' (fresh-var)))
                           (q x' (rename-prop (rv-in-prop x x' B)))))
    (_ P)))
```

Problem 4

```
(!claim (fresh-var))
```