



Institut für Informatik
Prof. Dr. Peter Thiemann
Jochen Walter

Georges-Köhler-Allee 79
D-79110 Freiburg i. Br.

Freiburg, den 5. Februar 2001

Informatik 1, WiSe 2000/2001

Übungsblatt 10

Die Aufgaben werden in den Übungs- und Programmiergruppen
vom 22.1. bis zum 26.1. besprochen.

Die Lösungen müssen nicht abgegeben werden!

Die Aufgaben auf diesem Blatt können in Teams bearbeitet werden. Bevor drscheme gestartet werden kann, muß setup lang eingegeben werden. Der Sprachumfang sollte auf „Full Scheme“ eingestellt werden. Die Aufgaben sind mit einem bis drei Sternen versehen, wobei die Aufgaben mit einem Stern am einfachsten, die mit dreien am schwierigsten sind.

Aufgabe 1 (***):

Gegeben sei der folgende ADT:

datatype Nat2
sorts Nat2
operations zero : \rightarrow Nat2
succ : Nat2 \rightarrow Nat2
equations succ(succ(x)) = x

- (a) Beweisen Sie: Für die Terme $s = \text{succ}(\text{succ}(\text{succ}(\text{zero})))$ und $t = \text{succ}(\text{zero})$ gilt $[s]_{\equiv_Q} = [t]_{\equiv_Q}$. Dabei ist Q die Axiomenmenge von Nat2.
- (b) Geben Sie drei Σ -Algebren an, in denen Q gilt. (Σ bezeichnet dabei das Rangalphabet von Nat2.) Die Trägermenge der ersten Σ -Algebra $\mathcal{A} = (A, \alpha)$ soll ein Element haben, die der zweiten Σ -Algebra $\mathcal{B} = (B, \beta)$ zwei und die der dritten $\mathcal{C} = (C, \gamma)$ mehr als zwei Elemente.
- (c) Wir wählen nun als Menge der Konstruktoren $\Gamma = \Sigma$. Welche der Funktionen

$$\hat{\alpha} : T_\Gamma / \equiv_Q \rightarrow A$$

$$\hat{\beta} : T_\Gamma / \equiv_Q \rightarrow B$$

$$\hat{\gamma} : T_\Gamma / \equiv_Q \rightarrow C$$

sind injektiv? Welche sind surjektiv? Welche der Σ -Algebren ist deshalb eine Implementierung (nach Teil B3.8 der Vorlesung) von Nat2?

Lösung zu Aufgabe 1:

- (a) Zunächst gilt zero \equiv_Q zero wegen Punkt 1 der Definition von \equiv_Q . Wegen Punkt 4 gilt auch succ(zero) \equiv_Q succ(zero).

Wir betrachten nun die Menge der Variablen in Q $X = \{x\}$ und die Variablenbelegung $g : X \rightarrow T_\Sigma$ mit $g(x) = \underline{\text{succ}}(\underline{\text{zero}})$. Nach Punkt 5 der Definition von \equiv_Q gilt $\hat{f}(\underline{\text{succ}}(\underline{\text{succ}}(x))) \equiv_Q \hat{f}(x)$ für alle Variablenbelegungen f , also auch für unsere Variablenbelegung g , d.h. es gilt $\underline{\text{succ}}(\underline{\text{succ}}(\underline{\text{succ}}(\underline{\text{zero}}))) = \underline{\text{succ}}(\underline{\text{zero}})$. Damit gilt $[s]_{\equiv_Q} = [t]_{\equiv_Q}$.

(b) (a) $\mathcal{A} = (\{0\}, \alpha)$ mit

$$\begin{aligned}\alpha(\underline{\text{zero}})() &= 0 \\ \alpha(\underline{\text{succ}})(x) &= 0\end{aligned}$$

(b) $\mathcal{B} = (\{0, 1\}, \beta)$ mit

$$\begin{aligned}\beta(\underline{\text{zero}})() &= 0 \\ \beta(\underline{\text{succ}})(x) &= (x + 1) \bmod 2\end{aligned}$$

(c) $\mathcal{C} = (\mathbb{N}, \gamma)$ mit

$$\begin{aligned}\gamma(\underline{\text{zero}})() &= 0 \\ \gamma(\underline{\text{succ}})(x) &= x\end{aligned}$$

(c) Zunächst gilt $T_\Gamma / \equiv_Q = \{[\underline{\text{zero}}]_{\equiv_Q}, [\underline{\text{succ}}(\underline{\text{zero}})]_{\equiv_Q}\}$

(a) $\hat{\alpha}$ ist surjektiv, aber nicht injektiv.

(b) $\hat{\beta}$ ist surjektiv und injektiv.

(c) $\hat{\gamma}$ weder injektiv noch surjektiv.

Damit ist \mathcal{B} eine Implementierung von Nat2 .

Aufgabe 2 (**):

Für zwei Terme $s, t \in T_\Sigma(X)$ für ein Rangalphabet Σ und eine Variablenmenge X sagt man t *paßt auf* s (t *matches* s), wenn es eine Substitution σ gibt, so daß $\hat{\sigma}(s) = t$. Man nennt dann t auch eine Instanz oder Spezialisierung von s und sagt s *umfaßt* t .

Zeigen Sie, daß die Relation t *paßt auf* s reflexiv und transitiv, aber nicht antisymmetrisch ist.

Lösung zu Aufgabe 2:

Wir betrachten ein beliebiges Rangalphabet (Σ, σ) und eine ebenso beliebig Variablenmenge X .

Reflexivität: Man betrachtet die Substitution $\sigma : X \rightarrow T_\Sigma(X)$ mit

$$(\forall x \in X) \quad \sigma(x) = x.$$

Für diese Substitution gilt für jeden Term $t \in T_\Sigma(X)$ $\hat{\sigma}(t) = t$ und folglich gilt t *paßt auf* t , d.h. die Relation ist reflexiv.

Transitivität: Wenn gilt u paßt auf t und t paßt auf s , dann gibt es zwei Substitutionen σ_1 und σ_2 mit $\hat{\sigma}_1(s) = t$ und $\hat{\sigma}_2(t) = u$.

Nun betrachtet man die Komposition $\hat{f} = \hat{\sigma}_2 \circ \hat{\sigma}_1$. Natürlich gilt $\hat{f}(s) = u$. Zu zeigen ist nur noch, daß es eine Substitution $\sigma : X \rightarrow T_\Sigma(X)$ gibt, deren homomorphe Fortsetzung $\hat{\sigma} = \hat{f}$ ist.

Dazu betrachtet man die Substitution $\sigma = \hat{\sigma}_2 \circ \sigma_1$. Für jedes $x \in X$ gilt $\hat{f}(x) = \sigma(x)$. Durch die Konstruktion ist dann auch $\hat{\sigma}$ und \hat{f} gleich. Damit gilt u paßt auf s , d.h. die Relation ist transitiv.

Antisymmetrie: Für zwei Variablen $x, y \in X$ gilt x paßt auf y und y paßt auf x , aber $x \neq y$. Damit ist die Relation nicht antisymmetrisch.

Aufgabe 3 (Programmierung, **):

In dieser Aufgabe soll eine einfache Simulation einer Verkehrsampel programmiert werden.

- (a) Geben Sie Scheme-Code für eine Klasse `bulb` an. Diese Klasse soll eine Instanzvariable `color` enthalten, in der eine Zeichenkette steht, die die Farbe der Glühbirne angibt.

Weiter soll die Klasse eine Methode `turn-on: -> VOID` und eine Methode `turn-off: -> VOID` haben, die jeweils die Farbe der Birne und die Art des Zustandswechsels ausgeben, etwa in der Form `gelb wurde eingeschaltet`.

Die Konstruktorfunktion `make-bulb: string -> bulb` hat ein Argument, mit dem die `color`-Variable initialisiert wird.

- (b) Geben Sie Scheme-Code für eine Klasse `traffic-light` an. Die Objekte dieser Klasse haben fünf Instanzvariable `red`, `yellow`, `green`, `state` und `name`. Die ersten drei enthalten jeweils ein `bulb`-Objekt in der entsprechenden Farbe, die vierte eine natürliche Zahl, die den Zustand der Ampel darstellt. Dabei bedeutet die Zahl 0 den Zustand aus, die Zahl 1 rot, die Zahl 2 rot-gelb, die Zahl 3 grün und die Zahl 4 gelb. Die fünfte Variable enthält eine Zeichenkette, die den Namen der Ampel enthält.

Die Klasse soll eine Methode `step: -> VOID` enthalten, die die Zustände durchschaltet. (Dabei kommt nach Zustand 4 wieder 1.) Bei jedem Schaltvorgang wird zunächst eine Meldung auf dem Bildschirm ausgegeben, aus der der Name der Ampel hervorgeht und danach die entsprechenden Birnen ein- bzw. ausgeschaltet. Eine zweite Methode der Klasse ist `reset: -> VOID`, bei deren Aufruf die Ampel in den Zustand 1 versetzt wird.

Die Konstruktorfunktion `make-traffic-light: string -> traffic-light` erzeugt ein `traffic-light`-Objekt, bei dem der Name auf den übergebenen Wert und der Zustand auf 0 gesetzt wird.

- (c) (Optional) Bilden Sie eine Unterklasse `traffic-light-counterpart` von der Klasse `traffic-light`, die sich von `traffic-light` dadurch unterscheidet, daß die Ampel bei Aufruf ihrer `reset`-Methode in Zustand 3 geht. (Das Problem einer richtigen Koordination zweier Ampeln, etwa wenn bei einer Baustelle Verkehr und Gegenverkehr über eine Spur geleitet werden müssen, ist erheblich komplizierter. Dabei muß sichergestellt werden, daß die Fahrzeuge in einer Fahrtrichtung genug Zeit haben, den kritischen Bereich zu verlassen, bevor die Strecke für die Fahrzeuge in der anderen Richtung freigegeben wird.)
- (d) (Optional) Geben Sie eine Klasse `traffic-light-group` an. Sie enthält eine Instanzvariable `traffic-lights`, in der eine Liste aller Ampeln der Ampelgruppe steht. (Das

können `traffic-light`-Objekte und wegen der Subsumption auch `traffic-light-counterpart`-Objekte sein.)

Die Klasse soll eine Methode `step: -> VOID` enthalten, die für alle in ihr enthaltenen Ampeln die jeweilige `step`-Methode aufruft. Eine zweite Methode der Klasse ist `reset: -> VOID`, bei deren Aufruf alle Ampeln zurückgesetzt werden.

Die Konstruktorfunktion `make-traffic-light-group: list(traffic-light) -> traffic-light-group` erzeugt aus einer Liste von Ampeln ein `traffic-light-group`-Objekt.

Lösung zu Aufgabe 3:

(a) `(load "send.scm")` ; Enthält die Definitionen von `no-method`, `get-method` und `send`

```
;;; SIGNATUR
;;; make-bulb: string -> bulb
;;; ERKLÄRUNG
;;; Konstruktorfunktion für Objekte der Klasse bulb. Das Argument ist die
;;; Farbe der Glühbirne.
;;; Die Klasse kennt folgende Methoden:
;;; turn-on: -> VOID
;;; (send b 'turn-on) gibt eine Meldung aus, aus der die Farbe hervorgeht und
;;; die Tatsache, daß die Birne eingeschaltet wurde.
;;; (send b 'turn-off) gibt eine Meldung aus, aus der die Farbe hervorgeht und
;;; die Tatsache, daß die Birne ausgeschaltet wurde.
;;; DEFINITION
(define make-bulb
  (lambda (color)
    (lambda (message)
      (case message
        ((turn-on)
         (lambda (this)
           (display color)
           (display " wurde eingeschaltet")
           (newline)))
        ((turn-off)
         (lambda (this)
           (display color)
           (display " wurde ausgeschaltet")
           (newline)))
        (else
         (make-no-method 'bulb message)))))))
```

(b) `(load "ub10-3a.scm")`

```
;;; SIGNATUR
```

```

;;; make-traffic-light: string -> traffic-light
;;; ERKLÄRUNG
;;; Konstruktorfunktion für Objekte der Klasse traffic-light. Das Argument ist
;;; der Name der Ampel. Jede Ampel hat fünf verschiedene Zustände. Aus, rot,
;;; rot-gelb, grün und gelb.
;;; Die Klasse kennt folgende Methoden:
;;; step: -> VOID
;;; (send t 'step) führt dazu, daß die Ampel in den nächsten Zustand übergeht.
;;; Falls die Ampel im Zustand aus ist, wird eine Fehlermeldung ausgegeben.
;;; (send t 'reset) setzt die Ampel in den Zustand rot.
;;; DEFINITION
(define make-traffic-light
  (lambda (name)
    (let ((red (make-bulb "rot"))
          (yellow (make-bulb "gelb"))
          (green (make-bulb "grün"))
          (state 0))
      (lambda (message)
        (case message
          ((step)
           (lambda (this)
             (if (= state 0)
                 (begin
                  (display name)
                  (display " ist ausgeschaltet!")
                  (newline))
                 (begin
                  (display "Schaltvorgänge bei ")
                  (display name)
                  (display ":")
                  (newline)
                  (case state
                    ((1)
                     (send yellow 'turn-on))
                    ((2)
                     (send red 'turn-off)
                     (send yellow 'turn-off)
                     (send green 'turn-on))
                    ((3)
                     (send green 'turn-off)
                     (send yellow 'turn-on))
                    ((4)
                     (send yellow 'turn-off)
                     (send red 'turn-on)))
                  (set! state (if (= state 4)

```

```

                                                    (add1 state))))))
((reset)
 (lambda (this)
  (display name)
  (display " wird initialisiert")
  (newline)
  (case state
   ((0)
    (send red 'turn-on))
   ((1)
    #f)
   ((2)
    (send yellow 'turn-off))
   ((3)
    (send green 'turn-off)
    (send red 'turn-on))
   ((4)
    (send yellow 'turn-off)
    (send red 'turn-on)))
  (set! state 1)))
(else
 (make-no-method 'traffic-light message))))))

```

(c) (load "ub10-3b.scm")

```

;;; SIGNATUR
;;; make-traffic-light-counterpart: string -> traffic-light-counterpart
;;; ERKLÄRUNG
;;; Konstruktorfunktion für Objekte der Klasse traffic-light-counterpart.
;;; Ein Objekt der Klasse traffic-light-counterpart unterscheidet sich
;;; von einem Objekt der Klasse traffic-light dadurch, daß es bei Aufruf der
;;; Methode reset in Zustand rot geht.
;;; DEFINITION
(define make-traffic-light-counterpart
 (lambda (name)
  (let ((super (make-traffic-light name)))
   (lambda (message)
    (case message
     ((reset)
      (lambda (this)
       (send super 'reset)
       (send super 'step)

```

```

        (send super 'step))
      (else
        (get-method super message))))))
(d) (load "ub10-3c.scm")

;;; SIGNATUR
;;; make-traffic-light-group: list(traffic-light) -> traffic-light-group
;;; ERKLÄRUNG
;;; Konstruktorfunktion für Objekte der Klasse traffic-light-group. Das
;;; Argument ist eine Liste von traffic-light-Objekten.
;;; Die Klasse kennt folgende Methoden:
;;; step: -> VOID
;;; (send g 'step) führt dazu, daß alle Ampeln in der Ampelgruppe in den
;;; nächsten Zustand übergehen.
;;; (send g 'reset) setzt alle Ampeln in den Zustand rot.
;;; DEFINITION
(define make-traffic-light-group
  (lambda (traffic-lights)
    (lambda (message)
      (letrec ((send-to-list
                (lambda (lst msg)
                  (if (not (null? lst))
                      (begin
                        (send (car lst) msg)
                        (send-to-list (cdr lst) msg)))))))
        (case message
          ((step)
           (lambda (this)
             (send-to-list traffic-lights 'step)))
          ((reset)
           (lambda (this)
             (send-to-list traffic-lights 'reset)))
          (else
           (make-no-method 'traffic-light-group message)))))))

(define g (make-traffic-light-group
  (list
    (make-traffic-light "Ampel 1")
    (make-traffic-light "Ampel 2")
    (make-traffic-light-counterpart "Ampel 3"))))

(send g 'step) (newline)
(send g 'reset) (newline)
(send g 'step) (newline)
(send g 'step) (newline)

```