



Institut für Informatik  
Prof. Dr. Peter Thiemann  
Jochen Walter

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

Freiburg, den 11. Januar 2001

## Informatik 1, WiSe 2000/2001

### Übungsblatt 9

Die Aufgaben werden in den Übungs- und Programmiergruppen  
vom 15.1. bis zum 19.1. 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 (\*\*):

Spezifizieren Sie den ADT Wab der Wörter über dem Alphabet  $\{a, b\}$ , indem Sie ein geeignetes Rangalphabet und eine Menge von Axiomen angeben. Beachten Sie, daß  $\varepsilon \in \{a, b\}^*$ !

#### Lösung zu Aufgabe 1:

Als Rangalphabet wählen wir  $\Sigma = \{\underline{a}, \underline{b}, \underline{\varepsilon}\}$  mit  $\sigma(\underline{a}) = \sigma(\underline{b}) = 1$  und  $\sigma(\underline{\varepsilon}) = 0$

Die Axiomenmenge  $Q = \emptyset$  ist die leere Menge.

Dann ist die  $\Sigma$ -Algebra  $(\{a, b\}^*, \alpha)$  mit

$$\begin{aligned}\alpha(\underline{\varepsilon})(w) &= \varepsilon \\ \alpha(\underline{a})(w) &= aw \\ \alpha(\underline{b})(w) &= bw\end{aligned}$$

eine Implementierung von Wab =  $(\Sigma, Q)$ .

#### Aufgabe 2 (\*\*):

Spezifizieren Sie den ADT Bool der Booleschen Algebra mit den Operationssymbolen true, false, not, and, or.

#### Lösung zu Aufgabe 2:

Als Rangalphabet wählen wir  $\Sigma = \{\underline{\text{true}}, \underline{\text{false}}, \underline{\text{not}}, \underline{\text{and}}, \underline{\text{or}}\}$  mit

$$\begin{aligned}\sigma(\underline{\text{true}}) &= 0 \\ \sigma(\underline{\text{false}}) &= 0 \\ \sigma(\underline{\text{not}}) &= 1 \\ \sigma(\underline{\text{and}}) &= 2 \\ \sigma(\underline{\text{or}}) &= 2.\end{aligned}$$

Die Axiomenmenge ist

$$Q = \left\{ \begin{array}{l} a \text{ or } b = b \text{ or } a, \\ a \text{ and } b = b \text{ and } a, \\ a \text{ or } (b \text{ and } c) = (a \text{ or } b) \text{ and } (a \text{ or } c), \\ a \text{ and } (b \text{ or } c) = (a \text{ and } b) \text{ or } (a \text{ and } c), \\ a \text{ or } \underline{\text{false}} = a \\ a \text{ and } \underline{\text{true}} = a \\ a \text{ or } (\underline{\text{not}} a) = \underline{\text{true}} \\ a \text{ and } (\underline{\text{not}} a) = \underline{\text{false}} \end{array} \right\}.$$

(Dabei wurden die Terme in Infixschreibweise geschrieben.)

### Aufgabe 3 (\*):

Geben Sie zwei verschiedene Implementierungen gemäß Definition B3.2 von Bool an.

### Lösung zu Aufgabe 3:

(a) Die  $\Sigma$ -Algebra  $(\{L, 0\}, \alpha)$  mit

$$\begin{aligned} \alpha(\underline{\text{true}})() &= L \\ \alpha(\underline{\text{false}})() &= 0 \\ \alpha(\underline{\text{not}})(a) &= \begin{cases} L & ; a = 0 \\ 0 & ; a = L \end{cases} \\ \alpha(\underline{\text{and}})(a, b) &= \begin{cases} L & ; a = b = L \\ 0 & ; \text{sonst} \end{cases} \\ \alpha(\underline{\text{or}})(a, b) &= \begin{cases} 0 & ; a = b = 0 \\ L & ; \text{sonst} \end{cases} \end{aligned}$$

ist eine Implementierung von Bool.

(b) Die  $\Sigma$ -Algebra  $(\{(0, 0), (0, L), (L, 0), (L, L)\}, \alpha)$  mit

$$\begin{aligned} \alpha(\underline{\text{true}})() &= (L, L) \\ \alpha(\underline{\text{false}})() &= (0, 0) \\ \alpha(\underline{\text{not}})((a, b)) &= (\neg a, \neg b) \\ \alpha(\underline{\text{and}})((a, b), (c, d)) &= (a \wedge c, b \wedge d) \\ \alpha(\underline{\text{or}})((a, b), (c, d)) &= (a \vee c, b \vee d) \end{aligned}$$

ist ebenfalls eine Implementierung von Bool.

### Aufgabe 4 (\*):

Für ein Rangalphabet  $\bar{\Sigma} = (\Sigma, \sigma)$  und eine Variablenmenge  $X$  ist eine *Substitution* eine Funktion

$$s : X \rightarrow T_{\bar{\Sigma}}(X).$$

Man kann  $s$  zu einer Funktion

$$\hat{s} : T_{\bar{\Sigma}}(X) \rightarrow T_{\bar{\Sigma}}(X)$$

fortsetzen (vgl. Satz B2.26):

$$\begin{aligned}\hat{s}(x) &= s(x) \quad \text{für } x \in X \\ \hat{s}(f t_1 \dots t_n) &= f \hat{s}(t_1) \dots \hat{s}(t_n)\end{aligned}$$

Gegeben sei  $\Sigma = \{0, \underline{+}, \underline{\text{succ}}\}$  mit  $\sigma(0) = 0$ ,  $\sigma(\underline{+}) = 2$  und  $\sigma(\underline{\text{succ}}) = 1$  sowie  $X = \{u, v, w, x, y, z\}$ . Die Funktion  $s$  sei gegeben durch  $s(u) = u$ ,  $s(v) = \underline{\text{succ}} v$ ,  $s(w) = \underline{\text{succ}} v$ ,  $s(x) = y$ ,  $s(y) = x$ ,  $s(z) = \underline{\text{succ}} \underline{\text{succ}} 0$

Wenden Sie die Substitution  $\hat{s}$  auf die folgenden Ausdrücke an:

- (a)  $u$
- (b)  $x$
- (c)  $\underline{+} x \underline{\text{succ}} y$
- (d)  $\underline{+} v \underline{+} w z$

#### Lösung zu Aufgabe 4:

- (a)  $u$
- (b)  $y$
- (c)  $\underline{+} y \underline{\text{succ}} x$
- (d)  $\underline{+} \underline{\text{succ}} v \underline{+} \underline{\text{succ}} v \underline{\text{succ}} \underline{\text{succ}} 0$

#### Aufgabe 5 (Programmierung, \* \* \*):

- (a) Definieren Sie eine Klasse `point`, die die  $x$ - und die  $y$ -Koordinate eines Punktes enthält. Der Konstruktorfunktion `make-point` sollen dabei die Koordinaten des Punktes übergeben werden. Die Objekte dieser Klasse sollen die Methode `move` beinhalten. Diese Methode erwartet eine Verschiebung  $d_x$  in  $x$ - und eine Verschiebung  $d_y$  in  $y$ -Richtung. Sie verändert die Koordinaten des Objektes auf den neuen Wert  $x + d_x$  und  $y + d_y$ . Der Rückgabewert soll das Objekt sein. Die Objekte sollen weiterhin die Methoden `x-of` und `y-of` kennen, die die aktuelle  $x$ - bzw.  $y$ -Koordinate zurückliefern.  
Geben Sie Scheme-Code an, der ein `point`-Objekt erzeugt, den Punkt verschiebt und danach seine Koordinaten bestimmt.
- (b) Erweitern Sie die Klasse `point` durch Vererbung zu einer Klasse `colored-point`. Der Konstruktorfunktion `make-colored-point` sollen dabei die Koordinaten und die Farbe (als `rgb`-Wert) des Punktes übergeben werden. Die Objekte dieser Klasse sollen zusätzlich eine Methode `update-color` kennen, die als Argument einen Farbwert erwartet und die Farbe des Objektes auf diesen Farbwert setzt. Die Methode soll das Objekt zurückgeben. Weiterhin sollen die Objekte noch eine Methode `color-of` kennen, die die Farbe des Objekts zurückliefert.
- (c) Erweitern Sie die Klasse `colored-point` zu einer Klasse `new-colored-point`. Diese Klasse soll die Methode `move` überschreiben. Zusätzlich zur Verschiebung des Punktes wird die Farbe des Objekts invertiert. Auch diese Methode soll das Objekt zurückliefern.

Ein `rgb`-Wert wird durch die Funktion

```
(define invert-color
  (lambda (c)
    (make-rgb (- 1.0 (rgb-red c))
              (- 1.0 (rgb-green c))
              (- 1.0 (rgb-blue c)))))
```

invertiert.

- (d) (Freiwillige Zusatzaufgabe.) Erweitern Sie die Klasse `new-colored-point` um eine graphische Darstellung. Bei Update-Operationen soll auch die graphische Darstellung abgeändert werden.

### Lösung zu Aufgabe 5:

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

```
;;; SIGNATUR
;;; make-point: number number -> point
;;; ERKLÄRUNG
;;; Konstruktorfunktion für Objekte der Klasse point. Die Argumente sind die
;;; x- und y-Koordinate des Punktes.
;;; Die Klasse kennt folgende Methoden:
;;; move: number number -> point
;;; (send p 'move dx dy) ändert die Koordinaten des Punktes um den
;;; Verschiebungsvektor (dx, dy) und liefert p zurück. Das Objekt p wird dabei
;;; verändert!
;;; x-of: -> number
;;; Liefert die x-Koordinate des Punktes zurück
;;; y-of: -> number
;;; Liefert die y-Koordinate des Punktes zurück
;;; DEFINITION
(define make-point
  (lambda (x y)
    (lambda (message)
      (case message
        ((move)
         (lambda (this dx dy)
           (set! x (+ x dx))
           (set! y (+ y dy))
           this))
         ((x-of)
          (lambda (this)
            x))
         ((y-of)
          (lambda (this)
            y))
         (else
```

```

                (make-no-method 'point message))))))

(define p (make-point 10 20))
(send p 'x-of )
(send p 'y-of )
(send p 'move 3 7)
(send p 'x-of )
(send p 'y-of )
(b) (load "ub9-5a.scm")

;;; SIGNATUR
;;; make-colored-point: number number color -> colored-point
;;; ERKLÄRUNG
;;; Konstruktorfunktion für Objekte der Klasse colored-point, die eine
;;; Unterklasse von point ist. Die Argumente sind die x- und y-Koordinate
;;; sowie die Farbe des Punktes.
;;; Zusätzlich zu den Methoden von point kennt diese Klasse folgende Methoden:
;;; update-color: color -> colored-point
;;; (send p 'update-color c) ändert die Farbe von p zu c und liefert p zurück.
;;; Das Objekt p wird dabei verändert!
;;; color-of: -> color
;;; Liefert die Farbe des Punktes zurück.
;;; DEFINITION
(define make-colored-point
  (lambda (x y color)
    (let ((super (make-point x y)))
      (lambda (message)
        (case message
          ((update-color)
           (lambda (this c)
             (set! color c)
             this))
          ((color-of)
           (lambda (this)
             color))
          (else
           (get-method super message))))))))

(define p (make-colored-point 10 20 (make-rgb 1 0 0)))
(send p 'x-of )
(send p 'y-of )
(let ((c (send p 'color-of)))
  (list (rgb-red c)
        (rgb-green c)
        (rgb-blue c)))
(send p 'move 3 7)

```

```

(send p 'x-of )
(send p 'y-of )
(send p 'update-color (make-rgb 0 1 0))
(let ((c (send p 'color-of)))
  (list (rgb-red c)
        (rgb-green c)
        (rgb-blue c)))

```

```

(c) (load "ub9-5-0.scm")
     (load "ub9-5b.scm")

```

```

;;; SIGNATUR
;;; make-new-colored-point: number number color -> colored-point
;;; ERKLÄRUNG
;;; Konstruktorfunktion für Objekte der Klasse new-colored-point, die eine
;;; Unterklasse von colored-point ist. Die Argumente sind die x- und
;;; y-Koordinate sowie die Farbe des Punktes.
;;; Die Methode move von colored-point wird überschrieben durch
;;; move: number number -> new-colored-point
;;; (send p 'move dx dy) ändert die Koordinaten des Punktes um den
;;; Verschiebungsvektor (dx, dy) und invertiert die Farbe von p. Rückgabewert
;;; ist das Objekt p, das durch den Aufruf von p verändert wird.
;;; DEFINITION
(define make-new-colored-point
  (lambda (x y color)
    (let ((super (make-colored-point x y color)))
      (lambda (message)
        (case message
          ((move)
           (lambda (this dx dy)
             (send super 'move dx dy)
             (send this 'update-color (invert-color (send super 'color-of)))
             this))
          (else
           (get-method super message))))))))

(define p (make-new-colored-point 10 20 (make-rgb 1 0 0)))
(send p 'move 3 7)
(send p 'x-of )
(send p 'y-of )
(let ((c (send p 'color-of)))
  (list (rgb-red c)
        (rgb-green c)
        (rgb-blue c)))

```