13 Abstrakte Datentypen

Bisher: *Konkrete Datentypen*

- Menge von Elementen
- Operationen auf den Elementen (Konstruktoren, Selektoren, Typprädikate)
- Eigenschaften abgeleitet

Jetzt: *Abstrakte Datentypen* (ADT)

- Operationen und Eigenschaften vorgegeben
- Menge von Elementen (Repräsentation) *uninteressant*
- Repräsentation wird vor Anwender des ADT verborgen
  (*information hiding*, Datenkapselung)
- Implementierung durch passenden konkreten Datentyp
  - jede Implementierung der Operationen kann eigene Repräsentation wählen
  - abhängig von verfügbaren Implementierungen, nicht-funktionalen Anforderungen (wie Geschwindigkeit oder Speicherverbrauch), etc

13.1 Definition: Abstrakter Datentyp (ADT)

Ein *abstrakter Datentyp* A ist gegeben durch

- eine Menge von Operationen auf A
  (beschrieben durch ihre Verträge)
- eine Menge von Eigenschaften der Operationen
  (beschrieben durch Gleichungen).

Bemerkung: Die Operationen können eingeteilt werden in

- Konstruktoren (Konstruktion eines Werts vom Typ A),
- Selektoren (Zugriff auf Komponente eines Werts vom Typ A) und
- Observatoren (Eigenschaften eines Werts vom Typ A, z.B. Länge einer Liste)
- Transformatoren (Umformung eines Werts vom Typ A, z.B. Listenverkettung)

13.2 Mengen als ADT: Operationen

Der Datentyp `(set %X)` von Mengen mit Elementen vom Typ `%X` sei gegeben durch die Verträge der Operationen

```
(: set-empty? ((set %X) -> boolean))
(: set-insert ((set %X) %X -> (set %X)))
(: set-remove ((set %X) %X -> (set %X)))
(: set-member ((set %X) %X -> boolean))
```

Der Datentyp `(set %X)` ist parametrisch: der Typ `%X` der Elemente ist beliebig.

13.3 Mengen als ADT: Konstruktor

Problem:

- Der Datentyp `(set %X)` ist parametrisch.
- Alle Implementierungen verlangen eine Gleichheitsrelation auf `%X`, manche sogar eine totale Ordnung.
- Woher kommen diese Relationen, wenn `%X` unbekannt ist?

Lösung:

- Die Relationen werden als Prädikate zu Parametern des Konstruktors und als Teil der Datenstruktur abgelegt.

```
(define pred (lambda (x) (contract (x x boolean))))
(: make-empty-set ((pred %X) (pred %X) -> (set %X)))
```

- Die Argumente von `(make-empty-set = <)` sind
  - eine Gleichheitsrelation, `(: = (pred %X))`, und
  - eine Kleiner-Relation, `(: < (pred %X))`, jeweils auf Datentyp `%X`.

13.4 Mengen als ADT: Eigenschaften

Zu den Verträgen der Operationen kommen noch Eigenschaften der Operationen. Z.B. gelten die beiden Eigenschaften

```
(for-all ((s (set %X)) (x %X))
  (set-member x (set-insert s x)))
(for-all ((s (set %X)) (x %X))
  (not (set-member (set-remove s x) x)))
```

Die Verträge und die Eigenschaften zusammengenommen definieren den abstrakten Datentyp *Menge*.

Weitere Eigenschaften (Auszug):

```
(for-all ((x integer))
  (not (set-member (make-empty-set = <) x)))
(set-empty? (make-empty-set = <))
(for-all ((s (set %X)) (x %X))
  (not (set-empty? (set-insert s x))))
```

13.5 Implementierung eines ADT

Eine Implementierung eines ADT A besteht aus

1. einer Menge (Sorte, konkreter Datentyp) M, deren Elemente die Elemente von A repräsentieren und
2. Implementierungen der Operationen des ADT für M, so dass die Eigenschaften/Gleichungen erfüllt sind.

Bemerkungen

- Ein abstrakter Datentyp A kann mehrere Implementierungen mit unterschiedlichen Charakteristiken (Effizienz) haben.
- Ein Klient eines ADT
  - verwendet nur die ADT Operationen mit den festgelegten Verträgen und Eigenschaften, aber
  - weiß nicht, welche Implementierung verwendet wird.

13.6 Implementierungen von `(set X)`, Invarianten

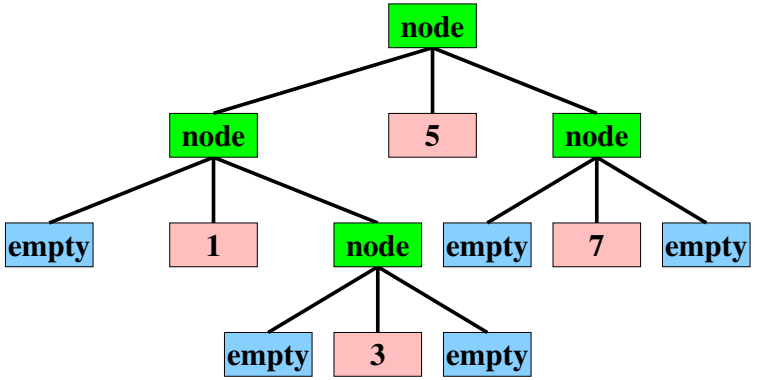
Für die Repräsentation der Elemente des Datentyps `(set X)` gibt es viele Möglichkeiten, demonstriert mit der Repräsentation der Menge $\{1, 3, 5, 7\}$.

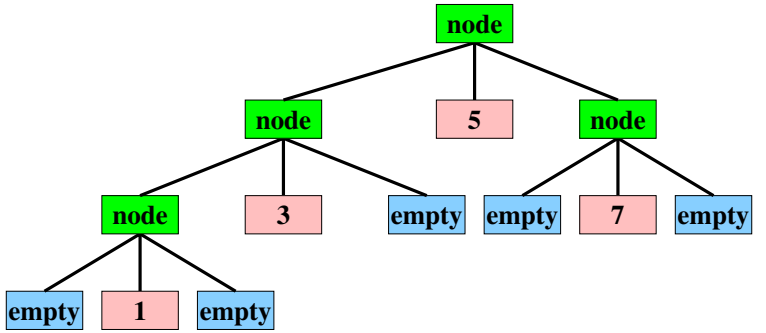
`list-set`: Liste der Elemente
`(list 1 3 5 7)`, `(list 3 5 3 7 3 1)`, `(list 7 7 3 3 1 5)`

`unique-list-set`: Liste der Elemente
Invariante: keine Duplikate
`(list 1 3 5 7)`, `(list 3 5 7 1)`, `(list 7 3 1 5)`

`sorted-list-set`: Liste der Elemente
Invariante: aufsteigend sortiert, ohne Duplikate
`(list 1 3 5 7)`

`btree-set:` binärer Baum, *Invariante*: Suchbaum





`function-set:` als charakteristische Funktion

```
(lambda (x)
  (or (= x 1) (= x 3) (= x 5) (= x 7)))
```

```
(lambda (x)
  (and (odd? x) (< 0 x 8)))
```

13.6.1 Implementierung: Menge durch ungeordnete Liste mit Wiederholungen

Ein `list-set` ist ein Wert der Form

`(make-list-set eq? rep)`

wobei `(: eq? (pred %X))` und `(: rep (list %X))` sind.
Dabei ist eq? die Gleichheitsrelation auf `%X`.

```
; Leere Menge
(define make-empty-list-set
  (lambda (= <)
    (make-list-set = empty)))

; Element einfügen
(define list-set-insert
  (lambda (s x)
    (make-list-set (list-set-eq? s)
                   (make-pair x (list-set-rep s)))))
```

Implementierung: Element suchen

```
(define list-set-member
  (lambda (s x)
    (let ((= (list-set-eq? s)))

      (letrec ((loop-member
        (lambda (l)
          (cond
           ((empty? l)
            #f)
           ((pair? l)
            (or (= x (first l))
                (loop-member (rest l))))))))))

      (loop-member (list-set-rep s))))))
```

13.6.2 Implementierung: Mengen durch Listen ohne Wiederholung

Ein `unique-list-set` ist ein Wert der Form

```
(make-unique-list-set eq? rep)
```

wobei `(: eq? (pred %X))` und `(: rep (list %X))` eine Liste ohne wiederholte Elemente ist.
Dabei ist eq? die Gleichheitsrelation auf `%X`.

Implementierung: Leere Menge

```
(define make-empty-unique-list-set
  (lambda (= <)
    (make-unique-list-set = empty)))
```

Implementierung: Element suchen

```
(define unique-list-set-member ...)
```

13.6.3 Implementierung: Mengen durch sortierte Listen ohne Wiederholung

Ein `sorted-list-set` ist ein Wert der Form

`(make-sorted-list-set eq? lt? rep)` wobei

- `(: eq? (pred %X))` die Gleichheitsrelation auf X,
- `(: lt? (pred %X))` die Kleiner-als-Relation auf X und
- `(: rep (list %X))` eine aufsteigend sortierte Liste ohne wiederholte Elemente ist.

```
; Leere Menge
(define make-empty-sorted-list-set
  (lambda (= <)
    (make-sorted-list-set = < empty)))

; Element einfügen
(define sorted-list-set-insert (lambda (s x) ...))
```

Implementierung: Element suchen

```
(define sorted-list-set-member
  (lambda (s x)
    (let ((= (sorted-list-set-eq? s)))
         ((< (sorted-list-set-lt? s)))

      (letrec ((loop-member
        (lambda (l)
          (cond
           ((empty? l)
            #f)
           ((pair? l)
            (or (= x (first l))
                (and (< (first l) x)
                     (loop-member (rest l))))))))))

      (loop-member (sorted-list-set-rep s))))))
```

13.6.4 Implementierung: Mengen durch binäre Suchbäume

Ein `search-tree-set` ist ein Wert der Form

`(make-search-tree eq? lt? rep)` wobei

- `(: eq? (pred %X))` die Gleichheitsrelation auf X,
- `(: lt? (pred %X))` die Kleiner-als-Relation auf X und
- `(: rep (btree %X))` ein binärer Suchbaum ist.

Implementierung: Leere Menge

```
(define make-empty-search-tree-set
  (lambda (= <)
    (make-search-tree = < the-empty-tree)))
```

Implementierung: Element einfügen

```
(define search-tree-set-insert (lambda (s x) ...))
(define search-tree-set-member (lambda (s x) ...))
```

13.7 Generische Implementierungen

- Klientenprogramme eines ADT dürfen sich nicht auf eine spezifische Implementierung beziehen, sondern müssen *unabhängig* davon sein.
- D.h. ein Klientenprogramm darf nicht direkt `search-tree-set-insert` oder `sorted-list-set-member` verwenden, da das Programm sonst auf diese eine Implementierung festgelegt wäre.

$\Rightarrow$ Benötigen *generische Schnittstelle*, die direkt die Operationen des abstrakten Datentyps benutzt.

- Drei Probleme
  - Konstruktion bricht Abstraktion:
    der Konstruktor verrät die verwendete Implementierung
  - Implementierung der generischen Schnittstelle
  - Implementierung der Verkapselung

13.7.1 Konstruktion bricht Abstraktion

Problem: Bei Konstruktion wird die Repräsentation erwähnt.

$\Rightarrow$ Bruch der Abstraktion!

Beispiel:

```
; work-with-set : ... -> ...
(define work-with-set
  (lambda (...)
    ... make-empty-search-tree-set ...
    ... set-empty? ... set-insert ...))
```

ADT Fabriken

Lösung: Parametrisiere/abstrahiere über den Konstruktor

⇒ in OO-Sprachen ist das ein **Entwurfsmuster** (*design pattern*), das **Fabrikmuster** (*factory pattern*)

D.h. an eine Funktion, die Mengen erzeugt, wird der gewünschte Mengenkonstruktor als Parameter übergeben.

Beispiel:

```
; work-with-set : ((pred X) (pred X) -> (S X)) ... -> ...
;       (S X) muss (set X) implementieren
(define work-with-set
  (lambda (make-empty-set ...)
    ... make-empty-set ...
    ... set-empty? ... set-insert ...))

; Verwendung mit Mengen implementiert durch binäre Suchbäume
(work-with-set make-empty-search-tree-set ...)
```

13.7.2 Implementierung der generischen Schnittstelle durch datengesteuerte Programmierung

- Bei der Verwendung von datengesteuerter Programmierung wählt jede Funktion anhand der Repräsentation die richtige Implementierung aus.
- Implementierung entsprechend dem Muster für gemischte Typen.

Element einfügen

```
(define set-insert
  (lambda (s x)
    (cond
     ((list-set? s)
      (list-set-insert s x))
     ((unique-list-set? s)
      (unique-list-set-insert s x))
     ((sorted-list-set? s)
      (sorted-list-set-insert s x))
     ((search-tree-set? s)
      (search-tree-set-insert s x)))))
```

Element suchen

```
(define set-member
  (lambda (s x)
    (cond
     ((list-set? s)
      (list-set-member s x))
     ((unique-list-set? s)
      (unique-list-set-member s x))
     ((sorted-list-set? s)
      (sorted-list-set-member s x))
     ((search-tree-set? s)
      (search-tree-set-member s x)))))
```

Nachteile der datengesteuerten Implementierung der generischen Schnittstelle:

- mühsame, uninteressante Implementierung (daher fehleranfällig)
  boilerplate code
- unflexibel: schlecht erweiterbar um neue Implementierungen des ADT
- nicht wartbar

Grund für die Nachteile:

- Jede generische Operation muss sämtliche Implementierungen kennen.

13.7.3 Verkapselung

- Ein Klientenprogramm darf nicht in der Lage sein, die Abstraktion zu brechen.
- Unter Verwendung der Typprädikate `list-set?`, `unique-list-set?`, `sorted-list-set?` usw. sowie der zugehörigen Selektoren könnte ein Klientenprogramm beliebige Operationen auf der Repräsentation der Elemente des ADT durchführen.

$\Rightarrow$ Invariante der Repräsentation kann verletzt werden

$\Rightarrow$ Operationen können falsche Ergebnisse liefern

- Beispiel: Falsche Konstruktion einer sortierten Liste

```
(define sk (make-sorted-list-set = < (list 99 1 2 3 4)))
(sorted-list-set-member sk 99) ⇒ #t
(sorted-list-set-member sk 3) ⇒ #f
```

13.8 Generische Implementierung durch Message-Passing

- Vermeidung der Nachteile der datengesteuerten Programmierung
- Erreicht Verkapselung und Flexibilität
- Ansatz:
  - Jedes Element eines ADT wird durch eine Funktion repräsentiert
  - Diese Funktion verkapselt die Repräsentation und die Operationen
  - Implementierung des ADT $=$ Operationen $\times$ Repräsentation

  $\Rightarrow$ Kapselung: Implementierung ist versteckt vor dem Programm
- Grundidee der objekt-orientierten Programmierung

13.8.1 Message Passing

- Idee:
  - Jede Operation eines ADT erhält einen internen Namen
  - Ein Element ist eine Prozedur, die den Namen einer Operation auf die Operation selbst abbildet

  $\Rightarrow$ eine Interpretation im Sinne der Σ-Algebra
- Implementierung: Strings als Namen

Beispiel: `list-set`

```
(: make-generic-list-set ((pred %a) (pred %a) -> (string -> %op)))
(define make-generic-list-set
  (lambda (= <)
    (letrec ((wrap (lambda (rep)
                     (lambda (m)
                       (cond
                        ((string=? m "empty?")
                         (lambda ()
                           (empty? (list-set-rep rep))))
                        ((string=? m "member?")
                         (lambda (x)
                           (list-set-member rep x))
                        ((string=? m "insert")
                         (lambda (x)
                           (wrap (list-set-insert rep x)))))))))))
      (wrap (make-list-set = < empty)))))
```

Beispiel: `sorted-list-set`

```
(: make-generic-sorted-list-set ((pred %a) (pred %a) -> (string -> %op)))
(define make-generic-sorted-list-set
  (lambda (= <)
    (letrec ((wrap (lambda (rep)
                     (lambda (m)
                       (cond
                        ((string=? m "empty?")
                         (lambda ()
                           (empty? (sorted-list-set-rep rep))))
                        ((string=? m "member?")
                         (lambda (x)
                           (sorted-list-set-member rep x))
                        ((string=? m "insert")
                         (lambda (x)
                           (wrap (sorted-list-set-insert rep x)))))))))
      (wrap (make-sorted-list-set = < empty)))))
```

Verwendung

```
(define l0 = (make-empty-sorted-list-set = <))
(define l1 = ((l0 "insert") 17))
(define l2 = ((l1 "insert") 42))
(define l3 = ((l2 "insert") 0))
((l3 "member?") 21) ⇒ #f
((l3 "member?") 17) ⇒ #t
```

- Implementierung bewahrt die Verkapselung
- Beobachtung: jeder Aufruf einer Operation hat das gleiche Muster
  ((*object message*) *arguments*)

⇒ Abstraktion des Musters führt zur generischen Implementierung

13.8.2 Generische Implementierung

```
; generic methods
(define set-empty?
  (lambda (s)
    ((s "empty?"))))
(define set-member?
  (lambda (s x)
    ((s "member?") x)))
(define set-insert
  (lambda (s x)
    ((s "insert") x)))
```

- Diese ADT-Operationen funktionieren mit jeder message-passing Implementierung.

Beispiel: Anwendung der neuen Mengenoperationen

```
   (define ul-s (set-insert (make-generic-list-set = <) 1))
   (set-rep (set-insert ul-s 1))
=> (list 1 1)
   (set-member ul-s 7)
=> #f

   (define ol-s (set-insert (make-generic-ordered-list-set = <) 1))
   (set-rep (set-insert ol-s 1))
=> (list 1)
   (set-member ol-s 1)
=> #t
```

13.9 Interne Operationen

- Erweitere den ADT `(set X)` um
  `(: set-union ((set X) (set X) -> (set X)))`
- Problem: Die beiden Argumente könnten `(set X)` jeweils unterschiedlich repräsentieren!
- Lösungsmöglichkeit: Erweitere ADT um Konversionsoperation
  `(: set->list ((set X) -> (list X)))`

Erklärung: `(set->list s)` liefert Liste der Elemente von s.

Definition:

```
(define set->list
  ...)
; DAMIT
(define set-union
  (lambda (s1 s2)
    (list-fold set-insert s1 (set->list s2))))
```

13.10 Zusammenfassung

- Abstrakte Datentypen spezifizieren
  - eine Menge von Operationen
  - Eigenschaften der Operationen
- ADTs lassen mehrere Implementierungen zu
- Implementierung wählt Repräsentation mit Invariante
- Verkapselung
  - Klienten können die Abstraktion nicht brechen
  - Invariante bleibt bewahrt
- Klienten sind unabhängig von Implementierung
- Fabrikmuster
- Datengetriebene Implementierung möglich
- Generische Implementierung durch Message Passing