14 Zuweisungen und Zustand

Bisher: funktionale / wertorientierte Programmierung

```
(let ((new-set (set-insert old-set new-element)))
  ... new-set
  ... old-set
  ...)
```

- alte und neue Version sind gleichzeitig verfügbar
- jede Funktion kann so programmiert werden
- meistens ohne Performanzverlust
- einfaches Berechnungsmodell (Substitutionsmodell)
- einfach nachzuvollziehen und zu verifizieren (Induktion)
- große Vorteile beim Programmieren von Parallelrechnern

Zustandsänderung

- Funktionale Sicht
  - Neue Versionen werden durch **Kopieren, Erweitern und Neuerstellen** von Datenobjekten hergestellt
  - Alte Versionen bleiben unverändert verfügbar
  - Datenstrukturen sind *immutable* und *persistent*
- Imperative Sicht
  - Neue Versionen werden durch **Veränderung** von existierenden Datenobjekten hergestellt
  - Alte Versionen sind nicht mehr verfügbar
  - Datenstrukturen sind *mutable* und *ephemeral*

Die Welt ist veränderlich

Heraklit: panta rei




Imperatives Programmieren

- Modellierung von veränderlichen Objekten
  - geringerer Speicherplatzbedarf
  - direkte Änderung manchmal schneller
- Programme, die reale Objekte mit einbeziehen
- Höchste Effizienz erforderlich
- Nachteile
  - neues Berechnungsmodell erforderlich
  - schwieriger nachzuvollziehen
  - schwieriger korrekt zu beweisen
  - Probleme bei paralleler Ausführung

14.1 Das Bankkonto — Zustandsvariable

- Ein Bankkonto enthält immer einen bestimmten Geldbetrag.
- Es können Abhebungen vorgenommen werden.
- Der Kontostand darf dadurch nicht unter Null fallen.

```
; Zustandsvariable: aktueller Kontostand
; balance : number
(define balance 90)

; vom Konto einen gewissen Betrag abheben und anzeigen, ob dies möglich war
; withdraw : number -> boolean
; Effekt: verändert die Zustandsvariable balance
```

Gewünschtes Verhalten von `withdraw`

```
> ; (= balance 90)
> (withdraw 1000)
#f
> ; (= balance 90)
> (withdraw 40)
#t
> ; (= balance 50)
> (withdraw 40)
#t
> ; (= balance 10)
> (withdraw 40)
#f
```

- `withdraw` liefert bei gleicher Eingabe unterschiedliche Ausgaben
- `withdraw` ist keine Funktion im mathematischen Sinn

Definition: Zuweisung

`(set!` ⟨*variable*⟩ ⟨*expression*⟩`)`

- wertet ⟨*expression*⟩ aus
- überschreibt den Wert von ⟨*variable*⟩ mit dem Wert von ⟨*expression*⟩
- Wert: *unspecified* (wird nicht gedruckt)

Beispiel

```
> (define raab 64)
> raab
64
> (set! raab 0)
> raab
0
> (set! raab 42)
> raab
42
```

Bemerkung: `set!` kann *nicht* mithilfe des Substitutionsmodells spezifiziert werden!

Konvention: Das Ausrufezeichen **!** „Bang“ signalisiert einen direkten Effekt.

Code für `withdraw`, 1. Versuch

```
; vom Konto einen gewissen Betrag abheben und anzeigen, ob dies möglich war
; withdraw : number -> boolean
; Effekt: verändert die Zustandsvariable balance
(define withdraw
  (lambda (amount)
    (if (>= balance amount)
        #t
        #f)))
```

- Passt zum Vertrag
- Richtiger Rückgabewert
- Alle Ellipsen ausgefüllt
- Aber es passiert kein Effekt!
- Muss noch im `#t`-Zweig eingefügt werden.

Definition: Block — Sequentielle Ausführung

`(begin` $\langle$*expression*$\rangle_1$ `...` $\langle$*expression*$\rangle_n$`)`

- wertet $\langle$*expression*$\rangle_1$ bis $\langle$*expression*$\rangle_n$ **von links nach rechts** aus
- Wert: Wert von $\langle$*expression*$\rangle_n$; *unspecified*, falls $n = 0$

Beispiel

```
   (begin (set! z (* 3 5)) 42)
=> (begin (set! z 15) 42)
=> (begin 42)
=> 42
```

Anmerkung: Für die meisten Formen ist die Reihenfolge der Auswertung der Teilausdrücke **nicht** spezifiziert!

Beispiel: In ($\langle$*operator*$\rangle$ $\langle$*operand*$\rangle_1$ ... $\langle$*operand*$\rangle_n$) ist **nicht** festgelegt, in welcher Reihenfolge $\langle$*operator*$\rangle$ sowie $\langle$*operand*$\rangle_1$ ... $\langle$*operand*$\rangle_n$ auszuwerten sind.

Code für `withdraw`

```
; vom Konto einen gewissen Betrag abheben und anzeigen, ob dies möglich war
; withdraw : number -> boolean
; Effekt: verändert die Zustandsvariable balance
(define withdraw
  (lambda (amount)
    (if (>= balance amount)
        (begin
          (set! balance (- balance amount))
          #t)
        #f)))
```

- Passt zum Vertrag
- Richtiger Rückgabewert
- Effekt geschieht (nur) vor Rückgabe von `#t`

Mantra #12 (Effekte)

Der Effekt einer Prozedur **muss** unter dem Vertrag durch einen Kommentar beschrieben werden.

14.2 Zustand kapseln

- Die bisherige Implementierung für ein Bankkonto ist nicht zufriedenstellend:
- Jedes Bankkonto erfordert eine eigene Zustandsvariable und eine eigene `withdraw`-Prozedur.

$\Rightarrow$ Die Anzahl der Konten muss vorab bekannt sein.

$\Rightarrow$ Die Verwendung von verschiedenen Namen für die gleiche Prozedur ist unnatürlich.

$\Rightarrow$ Die mehrfache Implementierung der gleichen Funktionalität erschwert die Wartung (Codeduplikation sollte vermieden werden).

Ansatz: Definiere ein Konto als einen Wert, der die Zustandsvariable für den Kontostand *enthält*.

Definition: Records mit Zustandsvariablen

`(define-record-procedures-2` t
c p
$(f_1 \ \ldots \ f_n))$

definiert einen zusammengesetzten Datentyp (Record) mit

- t ist der Name des definierten Typs
- c : $t_1 \ \ldots \ t_n$ `->` t ist der Vertrag des Konstruktors
- p : `value -> boolean` ist der Name des Typprädikats
- f_i kann sein:
  - entweder der Name des Selektors s_i : t `->` t_i oder
  - eine Liste $(s_i \ m_i)$ bestehend aus dem Namen s_i des Selektors und dem Namen eines *Mutators* m_i. (Konvention: endet mit **!**)
    In diesem Fall ist das i-te Feld eine Zustandsvariable, der Mutator hat den Vertrag m_i : t t_i `->` `unspecified` und $(m \ (c \ v_1 \ldots v_n) \ w)$ **ändert** die i-te Komponente nach w.

Gekapselter Zustand für Bankkonto

```
; Ein Bankkonto ist ein Wert
;  (make-account b)
; wobei b : number der Kontostand ist (veränderlich)
(define-record-procedures-2
  account
  make-account account?
  ((account-balance set-account-balance!)))
```

Vertrag von `set-account-balance!`

```
; Den Kontostand ändern
; set-account-balance! : account number -> unspecified
; Effekt: (set-account-balance! a n) setzt den Kontostand auf n
```

Verwendung von `account`

```
> (define a1 (make-account 90))
> (balance a1)
90
> (set-account-balance! a1 777)
> (balance a1)
777
```

Geld abheben

```
; Geld abheben und anzeigen, ob das möglich war
; account-withdraw : account number -> boolean
; Effekt: (account-withdraw a n) ändert den Kontostand von a
(define account-withdraw
  (lambda (a n)
    (if (>= (account-balance a) n)
      (begin
        (set-account-balance! a (- (account-balance a) n))
        #t)
      #f)))
```

Verwendung von `account-withdraw`

```
> (define a2 (make-account 90))
> (account-withdraw a2 1000)
#f
> (account-withdraw a2 40)
#t
> (account-withdraw a2 40)
#t
> (account-withdraw a2 40)
#f
```

Mehrere Konten sind voneinander unabhängig

- Jedes durch `make-account` erzeugte Konto besitzt eine **eigene Identität**. Es kann sich (genauer: seinen Stand) unabhängig von allen anderen Konten ändern.

```
> (define a3 (make-account 50))
> (define a4 (make-account 100))
> (account-withdraw a3 60)
#f
> (account-withdraw a4 60)
#t
> (account-withdraw a4 50)
#f
> (account-withdraw a3 50)
#t
> (account-balance a3) (account-balance a4)
0 40
```

Gekapselter Zustand vs. globaler Zustand

- Globaler Zustand
  - ist genau einmal vorhanden,
  - ist überall sichtbar und
  - kann überall verändert werden.
- Gekapselter Zustand
  - kann mehrfach mit verschiedenen Identitäten vorhanden sein,
  - ist nur über den zugehörigen Record-Wert sichtbar,
  - kann nur über dieses Record verändert werden,
  - Zugriffskontrolle ist möglich

Mantra #12 (Gekapselter Zustand)

Gekapselter Zustand ist besser als globaler Zustand.

14.2.1 Konstruktionsanleitung 10 (Gekapselter Zustand)

Falls ein Wert Zustandskomponenten enthalten soll, schreibe eine Datendefinition wie bei zusammengesetzten Daten und lege fest, welche der Felder veränderbar sein sollen.

Die zugehörige Record-Definition wird mit `define-record-procedures-2` erstellt. Für die veränderbaren Felder müssen Mutatoren definiert werden.

Konvention: Der Mutator zum Feld mit Selektor s heißt `set-`s`!`.

Falls sich an der Position k ein veränderbares Feld befindet, so lautet die Definition

```
(define-record-procedures t
  c p
  (s_1 ... (s_k m_k) ... s_n))
```

In der Schablone einer Prozedur, die den Zustand von Feld k eines Records r vom Typ t auf den Wert a ändert, muss der Mutator in der Form `(`m_k r a`)` vorkommen.

Die Form `begin` dient zur Veränderung von mehreren Komponenten in einer Prozedur oder zur Definition eines Rückgabewerts nach einer Mutation.

14.3 Berechnungsmodell

- Bei Auswertung nach dem Substitutionsmodell hängt der Wert eines Ausdrucks nur von der Form des Ausdrucks selbst ab.
- Die Beispiele zeigen, dass dies bei Berechnungen mit Zustand nicht mehr gilt.

$\Rightarrow$ Ein erweitertes Berechnungsmodell ist erforderlich.

Das Substitutionsmodell mit Speicher

- Ein Speicher ist eine Abbildung von *Adressen* auf beliebige Scheme-Werte.
- Adressen werden einer beliebigen unendlichen Menge (z.B. den natürlichen Zahlen) entnommen.
- Die Auswertung einer Adresse bewirkt ihre *Dereferenzierung*, d.h. das Nachschlagen ihres Inhalts im Speicher.
- Eine Adresse ist *frisch*, falls sie im Speicher noch nicht belegt ist.

$\Rightarrow$ **Änderungen am Substitutionsmodell**

  - Variable und Komponenten von zusammengesetzten Datenobjekten werden **ausschließlich** an Adressen gebunden.
  - Das Ausführen einer Variablenbindung erzeugt eine frische Adresse. Das betrifft `define`, `let`, `letrec` und `lambda`.
  - Der Konstruktor eines Datenobjekts erzeugt für jede Komponente eine frische Adresse.

- Alle anderen Auswertungsregeln bleiben gleich.

14.3.1 Beispiel mit globalem Zustand/1

Speicher

Bindungen

Kommandosequenz

```
(define balance 90)
(set! balance (- balance 40))
balance
```

Beispiel mit globalem Zustand/2

Speicher

```
L1 |-> 90
```

Bindungen

```
balance = L1
```

Kommandosequenz

```
(set! balance (- balance 40))
balance
```

Beispiel mit globalem Zustand/3

Speicher

```
L1 |-> 90
```

Bindungen

```
balance = L1
```

Kommandosequenz

```
(set! L1 (- L1 40))
balance
```

Beispiel mit globalem Zustand/4

Speicher

```
L1 |-> 90
```

Bindungen

```
balance = L1
```

Kommandosequenz

```
(set! L1 (- 90 40))
balance
```

Beispiel mit globalem Zustand/5

Speicher

```
L1 |-> 90
```

Bindungen

```
balance = L1
```

Kommandosequenz

```
(set! L1 50)
balance
```

Beispiel mit globalem Zustand/6

Speicher

```
L1 |-> 50
```

Bindungen

```
balance = L1
```

Kommandosequenz

```
balance
```

Beispiel mit globalem Zustand/7

Speicher

```
L1 |-> 50
```

Bindungen

```
balance = L1
```

Kommandosequenz

```
L1
```

Beispiel mit globalem Zustand/8

Speicher

```
L1 |-> 50
```

Bindungen

```
balance = L1
```

Kommandosequenz

```
50
```

14.3.2 Beispiel mit gekapseltem Zustand/1

Speicher

Bindungen

Kommandosequenz

```
(define a3 (make-account 50))
(define a4 (make-account 100))
(account-withdraw a3 60)
(account-withdraw a4 60)
(account-withdraw a4 50)
(account-withdraw a3 50)
(account-balance a3)
(account-balance a4)
```

Beispiel mit gekapseltem Zustand/2

Speicher

```
L1 |-> 50
```

Bindungen

Kommandosequenz

```
(define a3 (record:account L1))
(define a4 (make-account 100))
(account-withdraw a3 60)
(account-withdraw a4 60)
(account-withdraw a4 50)
(account-withdraw a3 50)
(account-balance a3)
(account-balance a4)
```

Beispiel mit gekapseltem Zustand/3

Speicher

```
L1 |-> 50; L3 |-> (record:account L1)
```

Bindungen

```
a3 = L3
```

Kommandosequenz

```
(define a4 (make-account 100))
(account-withdraw a3 60)
(account-withdraw a4 60)
(account-withdraw a4 50)
(account-withdraw a3 50)
(account-balance a3)
(account-balance a4)
```

Beispiel mit gekapseltem Zustand/4

Speicher

```
L1 |-> 50;  L3 |-> (record:account L1)
L2 |-> 100
```

Bindungen

```
a3 = L3
```

Kommandosequenz

```
(define a4 (record:account L2))
(account-withdraw a3 60)
(account-withdraw a4 60)
(account-withdraw a4 50)
(account-withdraw a3 50)
(account-balance a3)
(account-balance a4)
```

Beispiel mit gekapseltem Zustand/5

Speicher

```
L1 |-> 50;  L3 |-> (record:account L1)
L2 |-> 100; L4 |-> (record:account L2)
```

Bindungen

```
a3 = L3
a4 = L4
```

Kommandosequenz

```
(account-withdraw a3 60)
(account-withdraw a4 60)
(account-withdraw a4 50)
(account-withdraw a3 50)
(account-balance a3)
(account-balance a4)
```

Beispiel mit gekapseltem Zustand/6

Speicher

```
L1 |-> 50;  L3 |-> (record:account L1)
L2 |-> 100; L4 |-> (record:account L2)
```

Bindungen

```
a3 = L3
a4 = L4
```

Kommandosequenz

```
(account-withdraw L3 60)
(account-withdraw a4 60)
(account-withdraw a4 50)
(account-withdraw a3 50)
(account-balance a3)
(account-balance a4)
```

Beispiel mit gekapseltem Zustand/7

Speicher

```
L1 |-> 50;  L3 |-> (record:account L1)
L2 |-> 100; L4 |-> (record:account L2)
```

Bindungen

```
a3 = L3
a4 = L4
```

Kommandosequenz

```
#f
(account-withdraw a4 60)
(account-withdraw a4 50)
(account-withdraw a3 50)
(account-balance a3)
(account-balance a4)
```

Beispiel mit gekapseltem Zustand/8

Speicher

```
L1 |-> 50;  L3 |-> (record:account L1)
L2 |-> 100; L4 |-> (record:account L2)
```

Bindungen

```
a3 = L3
a4 = L4
```

Kommandosequenz

```
(account-withdraw L4 60)
(account-withdraw a4 50)
(account-withdraw a3 50)
(account-balance a3)
(account-balance a4)
```

Beispiel mit gekapseltem Zustand/9

Speicher

```
L1 |-> 50;  L3 |-> (record:account L1)
L2 |-> 100; L4 |-> (record:account L2)
```

Bindungen

```
a3 = L3
a4 = L4
```

Kommandosequenz

```
(if (>= (account-balance L4) 50)
      (begin
        (set-account-balance! L4 (- (account-balance L4) 50))
        #t)
      #f)
(account-withdraw a4 50)
(account-withdraw a3 50)
(account-balance a3)
```

```
(account-balance a4)
```

Beispiel mit gekapseltem Zustand/10

Speicher

```
L1 |-> 50;  L3 |-> (record:account L1)
L2 |-> 100; L4 |-> (record:account L2)
```

Bindungen

```
a3 = L3
a4 = L4
```

Kommandosequenz

```
(if (>= 100 50)
      (begin
        (set-account-balance! L4 (- (account-balance L4) 50))
        #t)
      #f)
(account-withdraw a4 50)
(account-withdraw a3 50)
(account-balance a3)
```

```
(account-balance a4)
```

Beispiel mit gekapseltem Zustand/11

Speicher

```
L1 |-> 50;  L3 |-> (record:account L1)
L2 |-> 100; L4 |-> (record:account L2)
```

Bindungen

```
a3 = L3
a4 = L4
```

Kommandosequenz

```
(begin
  (set-account-balance! L4 (- (account-balance L4) 50))
  #t)
(account-withdraw a4 50)
(account-withdraw a3 50)
(account-balance a3)
(account-balance a4)
```

Beispiel mit gekapseltem Zustand/12

Speicher

```
L1 |-> 50;  L3 |-> (record:account L1)
L2 |-> 100; L4 |-> (record:account L2)
```

Bindungen

```
a3 = L3
a4 = L4
```

Kommandosequenz

```
(begin
  (set-account-balance! L4 50)
  #t)
(account-withdraw a4 50)
(account-withdraw a3 50)
(account-balance a3)
(account-balance a4)
```

Beispiel mit gekapseltem Zustand/13

Speicher

```
L1 |-> 50;  L3 |-> (record:account L1)
L2 |-> 50;  L4 |-> (record:account L2)
```

Bindungen

```
a3 = L3
a4 = L4
```

Kommandosequenz

```
(begin
  #t)
(account-withdraw a4 50)
(account-withdraw a3 50)
(account-balance a3)
(account-balance a4)
```

Beispiel mit gekapseltem Zustand/14

Speicher

```
L1 |-> 50;  L3 |-> (record:account L1)
L2 |-> 50;  L4 |-> (record:account L2)
```

Bindungen

```
a3 = L3
a4 = L4
```

Kommandosequenz

```
(account-withdraw a4 50)
(account-withdraw a3 50)
(account-balance a3)
(account-balance a4)
```

Beispiel mit gekapseltem Zustand/15

Speicher

```
L1 |-> 50;  L3 |-> (record:account L1)
L2 |-> 0;   L4 |-> (record:account L2)
```

Bindungen

```
a3 = L3
a4 = L4
```

Kommandosequenz

```
(account-withdraw a3 50)
(account-balance a3)
(account-balance a4)
```

Beispiel mit gekapseltem Zustand/16

Speicher

```
L1 |-> 0;    L3 |-> (record:account L1)
L2 |-> 0;    L4 |-> (record:account L2)
```

Bindungen

```
a3 = L3
a4 = L4
```

Kommandosequenz

```
(account-balance a3)
(account-balance a4)
```

Beispiel mit gekapseltem Zustand/17

Speicher

```
L1 |-> 0;    L3 |-> (record:account L1)
L2 |-> 0;    L4 |-> (record:account L2)
```

Bindungen

```
a3 = L3
a4 = L4
```

Kommandosequenz

```
0
(account-balance a4)
```

Beispiel mit gekapseltem Zustand/18

Speicher

```
L1 |-> 0;   L3 |-> (record:account L1)
L2 |-> 0;   L4 |-> (record:account L2)
```

Bindungen

```
a3 = L3
a4 = L4
```

Kommandosequenz

```
0
```

14.3.3 Problem: Geteilter Zustand — Aliasing und Sharing

Betrachte folgende Kommandosequenz

```
(define acc1 (make-account 1000))
(define acc2 acc1)
(account-withdraw acc1 100)
(= (account-balance acc1)
   (account-balance acc2))
```

Was ist das Ergebnis des letzten Ausdrucks?

Geteilter Zustand: Aliasing

```
(define acc1 (make-account 1000))
(define acc2 acc1)
(account-withdraw acc1 100)
(= (account-balance acc1)
   (account-balance acc2))
```

- Das Substitutionsmodell mit Speicher liefert das Ergebnis `#t`, da `acc1` und `acc2` beide an die gleiche Adresse gebunden sind und demnach die beiden Konten identisch sind.
- In einem solchen Fall heißen `acc1` und `acc2` **Aliase**, da sie unterschiedliche Namen für dasselbe Objekt sind.
- Bemerkung: Aliase sind ungefährlich, solange die Objekte keine Zustandskomponenten beinhalten, d.h. keine eigene Identität besitzen.

Geteilter Zustand: Sharing

Betrachte folgenden Personendatentyp

```
; Eine Person ist ein Wert
;  (make-person n a)
; wobei n : string der Name ist
;   und a : account das zugehörige Bankkonto ist
(define-record-procedures-2 person
  make-person person?
  (person-name person-account))
```

Geteiltes Konto ist halbes Konto?

Die folgende Kommandosequenz

```
(define shared-account (make-account 77))
(define sarah (make-person "Sarah" shared-account))
(define mark  (make-person "Mark"  shared-account))
```

bewirkt, dass Sarah und Mark ein gemeinsames Konto haben.

Nach den Buchungen

```
(account-withdraw (person-account sarah) 33)
(account-withdraw (person-account mark) 44)
```

ist das gemeinsame Konto leer.

Hier teilen (*share*) sich die Datenstrukturen Sarah und Mark ein gemeinsames `shared-account` Objekt. Änderungen durch Sarah sind auch für Mark sichtbar und umgekehrt.

14.3.4 Exkurs: parallele Ausführung

Problem

Betrachte folgende Kommandosequenz in einer erweiterten Sprache:

```
(define x 0)
(in-parallel
  (let ((y (+ x 1)))
    (set! x y))
  (let ((z (- x 1)))
    (set! x z)))
```

- Jeder Rechenschritt von `(in-parallel` e_1 e_2`)` wählt zufällig einen der Teilausdrücke e_1 bzw e_2 aus und macht dort einen Schritt
- Frage: welchen Wert hat `x` zum Schluss?

Analyse

- Das Problem ist die globale Zustandsvariable `x`, die von beiden Teilausdrücken von `in-parallel` gelesen und geschrieben wird.
- Es gibt vier interessante Auswertungsschritte, die `x` betreffen

  L1 in e_1: `let ((y (+ x 1)))` liest `x`

  S1 in e_1: `(set! x y)` schreibt `x`

  L2 in e_2: `let ((z (- x 1)))` liest `x`

  S2 in e_2: `(set! x z)` schreibt `x`
- Die Berechnung der Auswertungsschritte von e_1 und e_2 kann beliebig verzahnt geschehen.

Mögliche Abläufe

- L1: `let ((y (+ x 1)))`
- L2: `let ((z (- x 1)))`
- S1: `(set! x y)`
- S2: `(set! x z)`

Ablauf	Ergebnis `x`
L1, S1, L2, S2	0
L1, L2, S1, S2	-1
L1, L2, S2, S1	1
L2, L1, S2, S1	1
L2, L1, S1, S2	-1
L2, S2, L1, S1	0

$\Rightarrow$ Das Ergebnis hängt von der zufälligen Verzahnung der Berechnungsprozesse ab.

Mantra #14 (Vermeide Zustand)

Verwende Zustand nur, wenn es nicht zu vermeiden ist.

14.3.5 Zusammenfassung

Zustand ist manchmal erforderlich:

- Modellierung von veränderlichen Objekten
- Erzielung der optimalen Effizienz

Aber Zustand ist nicht unproblematisch:

- komplizierteres Berechnungsmodell
- schwieriger nachzuvollziehen und zu verifizieren
- interessante Effekte durch Aliasing und Sharing
- unvorhersehbare Effekte durch Parallelität