11 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)
- vereinfacht die Programmierung von Multi-core Rechnern

Zustandsänderung

Funktionale Sicht

- Neue Versionen werden durch **Kopieren, Erweitern und Neuerstellen** von Datenobjekten hergestellt
- Alte Versionen bleiben unverändert verfügbar
  sie werden automatisch entfernt, wenn sie nicht mehr zugreifbar sind (*garbage collection*)

⇒ Funktionale Datenstrukturen sind *immutable* und *persistent*

Imperative Sicht

- Neue Versionen werden durch **Veränderung** von Datenobjekten hergestellt
- Alte Versionen sind nicht mehr verfügbar

⇒ Imperative Datenstrukturen sind *mutable* und *ephemeral*

Die Welt ist veränderlich

Heraklit: panta rei

Imperatives Programmieren

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

11.1 Programmieren mit Zustandsvariablen

- In DrScheme sind Zustandsvariable spezielle Felder von zusammengesetzten Datentypen.
- Der einfachste Fall ist eine Referenz:

```
(define-record-procedures-parametric-2 (ref X)
  make-ref ref?
  ((get-ref set-ref!)))
```

- Datenstruktur mit einem Feld, das als Zustandsvariable dient
- Selektor `get-ref`
- Mutator `set-ref!` (verändert den Inhalt des Feldes)
- Referenz hat Vertrag `(ref X)`, wobei `X` der Vertrag des Inhalts der Referenz ist

Definition: Records mit Zustandsvariablen

(define-record-procedures-parametric-2 (t t_1 ... t_n)
  c p
  (f_1 ... f_n))

definiert einen zusammengesetzten Datentyp (Record) mit

- t ist der Name des definierten Typs
- $t_1, \ldots, t_n$ Platzhalter für Verträge für die Felder des Typs
- (: c (t_1 ... 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 **!**)

    Das i-te Feld ist eine Zustandsvariable, der Mutator (m_i (c $v_1 \ldots v_n$) w) **ändert** die i-te Komponente v_i nach w und hat den Vertrag (: m_i (t t_i -> unspecified)).

Beispiel: Bankkonto

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

```
; Zustandsvariable: aktueller Kontostand
(: balance (ref real))
(define balance (make-ref 90))

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

Gewünschtes Verhalten von `withdraw`

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

- `withdraw` liefert bei gleicher Eingabe unterschiedliche Ausgaben

$\Rightarrow$ `withdraw` ist keine Funktion im mathematischen Sinn

Definition: Zuweisung durch Mutator

(`set-ref!` ⟨*expression1*⟩ ⟨*expression2*⟩)

- wertet ⟨*expression1*⟩ aus (muss record vom Typ `ref` sein)
- überschreibt den Wert des Feldes von `ref` mit dem Wert von ⟨*expression2*⟩
- Wert: *unspecified* (wird nicht gedruckt)

Beispiel

```
> (define raab (make-ref 64))
> (get-ref raab)
64
> (set-ref! raab 0)
> (get-ref raab)
0
> (set-ref! raab 42)
> (get-ref raab)
42
```

Bemerkung: `set-ref!` 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 (real -> boolean))
; Effekt: verändert die Zustandsvariable balance
(define withdraw
  (lambda (amount)
    (if (>= (get-ref 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*$\rangle_n$

Beispiel

```
   (begin (set-ref! z (* 3 5)) 42)
=> (begin (set-ref! 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 (real -> boolean))
; Effekt: verändert die Zustandsvariable balance
(define withdraw
  (lambda (amount)
    (if (>= (get-ref balance) amount)
        (begin
          (set-ref! balance (- (get-ref balance) amount))
          #t)
        #f)))
```

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

MANTRA

Mantra #12 (Effekte)

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

11.2 Zustand kapseln

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

⇒ Die Anzahl der Konten muss vorab bekannt sein.

⇒ Die Verwendung von verschiedenen Namen für die gleiche Prozedur ist unnatürlich.

⇒ 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* und übergebe dies explizit als Parameter an die Operationen.

Gekapselter Zustand für Bankkonto

```
; Ein Bankkonto ist ein Wert
;  (make-account b)
; wobei b : real 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 real -> 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 real -> 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

Mantra #12 (Gekapselter Zustand)

Gekapselter Zustand ist besser als globaler Zustand.

11.2.1 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.

11.3 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.

- Neue Komponente: Speicher

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.
- Eine Adresse ist *frisch*, falls sie im Speicher noch nicht belegt ist.

⇒ **Änderungen am Substitutionsmodell**

  - Komponenten von zusammengesetzten Datenobjekten werden **ausschließlich** an Adressen gebunden.
  - Der Konstruktor eines Datenobjekts erzeugt für jede Komponente eine frische Adresse und speichert die Werte der Komponenten an diesen Adressen.
  - Die Auswertung eines Selektors bewirkt die *Dereferenzierung* der dort abgelegten Adresse, d.h. das Nachschlagen ihres Inhalts im Speicher.
  - Die Auswertung eines Mutators bewirkt die Änderung des Inhalts der dort abgelegten Adresse.

- Alle anderen Auswertungsregeln bleiben gleich.

Beispiel mit gekapseltem Zustand/1

Speicher

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
```

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
```

Kommandosequenz

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

Beispiel mit gekapseltem Zustand/4

Speicher

```
L1 |-> 50
L2 |-> 100
```

Kommandosequenz

```
(define a4 (record:account L2))
(account-withdraw (record:account L1) 60)
(account-withdraw a4 60)
(account-withdraw a4 50)
(account-withdraw (record:account L1) 50)
(account-balance (record:account L1))
(account-balance a4)
```

Beispiel mit gekapseltem Zustand/5

Speicher

```
L1 |-> 50
L2 |-> 100
```

Kommandosequenz

```
(account-withdraw (record:account L1) 60)
(account-withdraw (record:account L2) 60)
(account-withdraw (record:account L2) 50)
(account-withdraw (record:account L1) 50)
(account-balance (record:account L1))
(account-balance (record:account L2))
```

Beispiel mit gekapseltem Zustand/7

Speicher

```
L1 |-> 50
L2 |-> 100
```

Kommandosequenz

```
#f
(account-withdraw (record:account L2) 60)
(account-withdraw (record:account L2) 50)
(account-withdraw (record:account L1) 50)
(account-balance (record:account L1))
(account-balance (record:account L2))
```

Beispiel mit gekapseltem Zustand/8

Speicher

```
L1 |-> 50
L2 |-> 100
```

Kommandosequenz

```
(account-withdraw (record:account L2) 60)
(account-withdraw (record:account L2) 50)
(account-withdraw (record:account L1) 50)
(account-balance (record:account L1))
(account-balance (record:account L2))
```

Beispiel mit gekapseltem Zustand/9

Speicher

```
L1 |-> 50
L2 |-> 100
```

Kommandosequenz

```
(if (>= (account-balance (record:account L2)) 50)
      (begin
        (set-account-balance! (record:account L2) (- (account-balance (recor
        #t)
      #f)
(account-withdraw (record:account L2) 50)
(account-withdraw (record:account L1) 50)
(account-balance (record:account L1))
(account-balance (record:account L2))
```

Beispiel mit gekapseltem Zustand/10

Speicher

```
L1 |-> 50
L2 |-> 100
```

Kommandosequenz

```
(if (>= 100 50)
      (begin
        (set-account-balance!  (record:account L2) (- (account-balance  (rec
        #t)
      #f)
(account-withdraw (record:account L2) 50)
(account-withdraw (record:account L1) 50)
(account-balance (record:account L1))
(account-balance (record:account L2))
```

Beispiel mit gekapseltem Zustand/11

Speicher

```
L1 |-> 50
L2 |-> 100
```

Kommandosequenz

```
(begin
  (set-account-balance! (record:account L2) (- (account-balance (record:acco
  #t)
(account-withdraw (record:account L2) 50)
(account-withdraw (record:account L1) 50)
(account-balance (record:account L1))
(account-balance (record:account L2))
```

Beispiel mit gekapseltem Zustand/12

Speicher

```
L1 |-> 50
L2 |-> 100
```

Kommandosequenz

```
(begin
  (set-account-balance! (record:account L2) 50)
  #t)
(account-withdraw (record:account L2) 50)
(account-withdraw (record:account L1) 50)
(account-balance (record:account L1))
(account-balance (record:account L2))
```

Beispiel mit gekapseltem Zustand/13

Speicher

```
L1 |-> 50
L2 |-> 50
```

Kommandosequenz

```
(begin
  #t)
(account-withdraw (record:account L2) 50)
(account-withdraw (record:account L1) 50)
(account-balance (record:account L1))
(account-balance (record:account L2))
```

Beispiel mit gekapseltem Zustand/14

Speicher

```
L1 |-> 50
L2 |-> 50
```

Kommandosequenz

```
(account-withdraw (record:account L2) 50)
(account-withdraw (record:account L1) 50)
(account-balance (record:account L1))
(account-balance (record:account L2))
```

Beispiel mit gekapseltem Zustand/15

Speicher

```
L1 |-> 50
L2 |-> 0
```

Kommandosequenz

```
(account-withdraw (record:account L1) 50)
(account-balance (record:account L1))
(account-balance (record:account L2))
```

Beispiel mit gekapseltem Zustand/16

Speicher

```
L1 |-> 0
L2 |-> 0
```

Kommandosequenz

```
(account-balance (record:account L1))
(account-balance (record:account L2))
```

Beispiel mit gekapseltem Zustand/17

Speicher

```
L1 |-> 0
L2 |-> 0
```

Kommandosequenz

```
0
(account-balance (record:account L2))
```

Beispiel mit gekapseltem Zustand/18

Speicher

```
L1 |-> 0
L2 |-> 0
```

Kommandosequenz

```
0
```

11.4 Problem: Geteilter Zustand — Aliasing und Sharing

11.4.1 Aliasing

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/1

Speicher

Kommandosequenz

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

Geteilter Zustand/2

Speicher

```
L1 |-> 1000
```

Kommandosequenz

```
(define acc1 (record:account L1))
(define acc2 acc1)
(account-withdraw acc1 100)
(= (account-balance acc1)
   (account-balance acc2))
```

Geteilter Zustand/3

Speicher

```
L1 |-> 1000
```

Kommandosequenz

```
(define acc2 (record:account L1))
(account-withdraw (record:account L1) 100)
(= (account-balance (record:account L1))
   (account-balance acc2))
```

Geteilter Zustand/4

Speicher

```
L1 |-> 1000
```

Kommandosequenz

```
(account-withdraw (record:account L1) 100)
(= (account-balance (record:account L1))
   (account-balance (record:account L1)))
```

Geteilter Zustand/5

Speicher

```
L1 |-> 900
```

Kommandosequenz

```
(= (account-balance (record:account L1))
   (account-balance (record:account L1)))
```

Geteilter Zustand/6

Speicher

```
L1 |-> 900
```

Kommandosequenz

```
(= 900
   (account-balance (record:account L1)))
```

Geteilter Zustand/7

Speicher

```
L1 |-> 900
```

Kommandosequenz

```
(= 900
   900)
```

Geteilter Zustand/8

Speicher

```
L1 |-> 900
```

Kommandosequenz

```
#t
```

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 das Record mit der gleichen 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.

11.4.2 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.

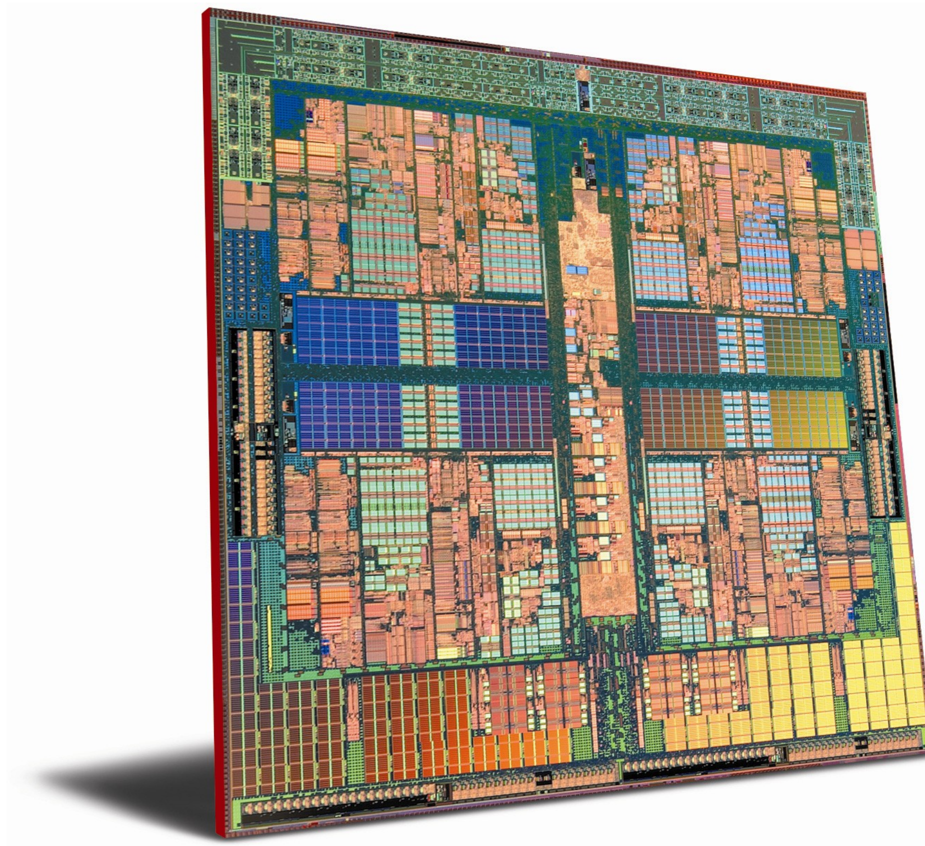
Nach den Buchungen

```
(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.

11.4.3 Exkurs: parallele Ausführung



Problem

Betrachte folgende Kommandosequenz in einer erweiterten Sprache:

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

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

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

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

Mögliche Abläufe

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

Mantra #14 (Vermeide Zustand)

Verwende Zustand nur, wenn es nicht zu vermeiden ist.

11.5 Zusammenfassung

Zustand ist manchmal erforderlich:

- Modellierung von veränderlichen Objekten
- Erzielung der größtmöglichen Effizienz

Aber Zustand ist nicht unproblematisch:

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