

## 35. Knobelaufgabe #19

Die *unendliche* Folge

0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 4 0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 5 ...

ist invariant unter der folgenden Transformation.

- ▶ Jedes Element wird um 1 erhöht.
- ▶ An den Anfang und zwischen je zwei Elemente wird eine 0 gesetzt.

---

Programmieren Sie eine Funktion, die alle Elemente der Folge nacheinander ausgibt.

*Assignment is undoubtedly the most characteristic feature of programming a digital computer, and one that most clearly distinguishes it from other branches of mathematics.*

— C.A.R. Hoare (1934–)

# 35. Motivation

*Wunsch*: Die Protokollierung der Auswertung via *Return* und *Call* soll sich an- und ausschalten lassen.

☞ Ein *globaler Schalter* lässt sich mit den bisherigen Mitteln nicht realisieren. Funktionen lassen sich bis dato nur über ihre Funktionsargumente steuern — oder über Eingaben der Benutzer\*in.

☞ Erweitert man *Return* und *Call* um ein Boolesches Argument, dann hat man viele *lokale Schalter*, aber keinen globalen.

# 35. Motivation

*Idee:* ein Gedächtnis in Form einer *Speicherzelle* einführen, die abgefragt und manipuliert werden kann.

---

☞ Ein Schalter kann durch eine Speicherzelle realisiert werden, die einen Booleschen Wert enthält.

```
let trace = ref false
```

☞ Der Ausdruck `ref false` legt eine neue Speicherzelle an, einen Behälter für einen Wert. Der initiale *Inhalt* der allokierten Speicherzelle ist `false`. Als Ergebnis des Aufrufs gibt `ref` die *Adresse* der allokierten Speicherzelle zurück.

## 35. Motivation

☞ Mit Hilfe der Adresse können wir den Inhalt der Speicherzelle abfragen.

```
!trace
```

☞ Wertet `e` zu einer Adresse aus, so ermittelt `!e` den Inhalt der adressierten Speicherzelle (lies: „bang e“).

---

Damit können wir eine Version von `putline` definieren, die sich von außen über den Schalter steuern lässt.

```
let traceline (s : String) =  
  if !trace then putline s
```

☞ Nur wenn der Inhalt der Speicherzelle `true` ist, erfolgt die Ausgabe. (Wo ist das `else`?)

## 35. Motivation

☞ Mit Hilfe der *Zuweisung*  $e_1 := e_2$  können wir den Inhalt der von  $e_1$  adressierten Speicherzelle auf den Wert von  $e_2$  setzen.

```
trace := true
```

Der Ausdruck schaltet die Protokollierung an.

```
trace := false
```

Der Ausdruck schaltet die Protokollierung aus.

☞ Der *Wert* der Zuweisung ist jeweils '()', unser liebgewonnener Dummywert.

In der Definition von *Return* und *Call* ersetzen wir *putline* durch *traceline*.

## 35. Demo

```
Mini> factorial 1
1
Mini> trace := true
()
Mini> !trace
true
Mini> factorial 1
call 0
return 1
return 1
1
Mini> trace := false
()
Mini> !trace
false
Mini> factorial 1
1
```

☞ '!' ist keine Funktion im mathematischen Sinne: der gleiche Aufruf, *!trace*, führt zu zwei unterschiedlichen Ergebnissen.

## 35. Exkursion: Module

☞ Namen zu erfinden ist schwer!

F# erlaubt es, Definitionen in sogenannten Modulen zusammenzufassen, die jeweils einen eigenen „Namensraum“ bilden.

```
module Values.Modules
```

```
module Ann =
```

```
  let name = "Ann"
```

```
module Bob =
```

```
  let name = "Bob"
```

```
let hello = "Hello " ^ Ann.name ^ " and " ^ Bob.name ^ "!"
```

Das Programm definiert das Modul *Values.Modules*, das zwei lokale Module, *Ann* und *Bob*, enthält. Der Bezeichner *name* wird zweimal definiert, lebt aber in zwei unterschiedlichen Namensräumen.

Auf die definierten Werte kann mittels qualifizierter Namen zugegriffen werden: *Ann.name* und *Bob.name*. (Von „außen“ entsprechend: *Values.Modules.Ann.name*.)

## 35. Exkursion: Module

Die Verwendung qualifizierter Namen ist oft mühselig.

```
module Values.Modules
```

```
module Ann =
```

```
  let name = "Ann"
```

```
module Bob =
```

```
  let name = "Bob"
```

```
open Ann
```

```
let hello = "Hello " ^ name ^ " and " ^ Bob.name ^ "!"
```

Mit **open** werden Namensräume geöffnet.

# 35. Bankkonto

☞ Speicherzellen können beliebige Werte enthalten:

- ▶ Boolesche Werte,
- ▶ natürliche Zahlen,
- ▶ Funktionen,
- ▶ Adressen anderer Speicherzellen usw.

☞ Eine Speicherzelle, die eine *natürliche* Zahl enthält, kann zum Beispiel verwendet werden, um ein *Bankkonto* zu modellieren: der Inhalt repräsentiert den Kontostand.

## 35. Bankkonto

```
module Account =  
  let private funds = ref 0  
  
  let deposit (amount : Nat) =  
    funds := !funds + amount  
  
  let withdraw (amount : Nat) =  
    let old = !funds  
    funds := !funds ÷ amount  
    old ÷ !funds  
  
  let balance () = !funds
```

 '÷' bezeichnet die Subtraktion auf den natürlichen Zahlen („minus“). Die Differenz zwischen dem alten und dem neuen Kontostand wird zurückgegeben.

## 35. Kapselung

- ▶ Die Speicherzelle *funds* ist lokal zu *deposit*, *withdraw* und *balance*.
- ▶ Der Zusatz *private* stellt sicher, dass der Bezeichner *funds* nur innerhalb des Moduls sichtbar ist.
- ▶ Man sagt auch, der Zustand ist *gekapselt*; von außen ist nicht sichtbar, dass die Funktionen eine Speicherzelle verwenden.
- ▶ Der Kontostand kann nur mit Hilfe der Funktion *balance* eingesehen werden.
- ▶ Der Kontostand kann nicht direkt über eine Zuweisung verändert werden, sondern nur indirekt mit *deposit* und *withdraw*.
- ▶ Heimliche Kontomanipulationen sind in der Bankenwelt sehr ungern gesehenen.

## 35. Demo

```
Mini> Account.deposit 4711  
()  
Mini> Account.withdraw 815  
815  
Mini> Account.withdraw 2765  
2765  
Mini> Account.withdraw 2765  
1131  
Mini> Account.withdraw 2765  
0
```

☞ *withdraw* ist keine mathematische Funktion: drei Aufrufe der Form *withdraw 2765*, drei unterschiedliche Funktionsergebnisse.



Endlich mal was, das einigermaßen bekannt aussieht. Statt  
 $\text{funds} := !\text{funds} + \text{amount}$  würde ich ja schreiben

$$\text{funds} = \text{funds} + \text{amount}$$

Ein- und  
Ausgabe

Zustand

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische  
Semantik

Vertiefung

Blick über den  
Tellerrand

Listenbeschrei-  
bungen

Kontrollstruktu-  
ren

Ausnahmen

Und ich habe schon gedacht, du hättest C++ vergessen ;-).



Mon dieu, dosch nischt '='. Dasch ischt doch kein Vergleich,  
sondern eine Zuweisung.

$$\text{funds} := \text{funds} + \text{amount}$$


Na ja, es *ist* nicht gleich, aber es wird doch gleich *gemacht*.



Aber die Ausdrücke links und rechts haben doch noch nicht  
einmal den gleichen Typ!



Wieso das denn nicht? Links *Nat* (na ja, eigentlich `unsigned int`) und rechts *Nat*.

Dann darf ich auch schreiben:

$$4711 = 815$$

Links *Nat*, rechts *Nat*.



Natürlich nicht! Links muss eine Variable stehen!

Genau. Links muss ein Ausdruck stehen, der zu einer *Adresse* ausgewertet. Der neue *Inhalt* der adressierten Speicherzelle steht dann auf der rechten Seite.



Aber dann ergibt doch `funds + amount` keinen Sinn: `funds` ist eine Adresse und `amount` eine Zahl.

Genau. Deswegen schreiben wir in Mini-F# ja auch `funds := !funds + amount`.



# 35. Abstrakte Syntax

Wir erweitern Mini-F# um Speicheroperationen.

|                                  |                   |
|----------------------------------|-------------------|
| $e ::= \dots$                    | <i>Ausdrücke:</i> |
| <b>ref</b> e                     | Allokation        |
| !e                               | Dereferenzierung  |
| e <sub>1</sub> := e <sub>2</sub> | Zuweisung         |

 **ref** e allokiert eine Speicherzelle und gibt die Adresse der bzw. eine Referenz auf die Speicherzelle zurück. Aus diesem Grund heißt der Zugriff !e auch Dereferenzierung.

## 35. Statische Semantik

Adressen erhalten einen Referenztyp; dieser ist mit dem Typ des Inhaltes parametrisiert.

$$t ::= \dots \\ | \text{Ref}\langle t \rangle$$

Typen:  
Referenztyp

Typregeln:

$$\frac{\Sigma \vdash e : t}{\Sigma \vdash \text{ref } e : \text{Ref}\langle t \rangle}$$

$$\frac{\Sigma \vdash e : \text{Ref}\langle t \rangle}{\Sigma \vdash !e : t}$$

$$\frac{\Sigma \vdash e_1 : \text{Ref}\langle t \rangle \quad \Sigma \vdash e_2 : t}{\Sigma \vdash e_1 := e_2 : \text{Unit}}$$

## 35. Statische Semantik

Zur Erinnerung: Ausdrücke können beliebig miteinander kombiniert werden.

```
let p = (ref false, ref 0)
```

  $p$  hat den Typ  $Ref\langle Bool \rangle * Ref\langle Nat \rangle$ .

Alle folgenden Ausdrücke sind zulässig:

```
!(fst p)
```

```
snd p := 4711
```

```
fst (swap p) := !(snd p) + 1
```

# 35. Dynamische Semantik

Die drei neuen Konstrukte manipulieren einen sogenannten (Haupt-) Speicher.

Ein Speicher ist eine endliche Abbildung von Adressen auf Werte.

$a \in \text{Addr}$  *Adressen*

$\sigma \in \text{Addr} \rightarrow_{\text{fin}} \text{Val}$  *Speicher*

☞ Den Bereich der Adressen lassen wir abstrakt; wir fordern nur, dass es unendlich viele Adressen gibt.

## 35. Dynamische Semantik

- ▶ Die Auswertung verändert sich mit dem Einzug von Effekten.
- ▶ Wie müssen wir die Auswertungsregeln modifizieren, um Speichermodifikationen modellieren zu können?
- ▶ Ein Ausdruck hat neben einem *Wert* zusätzlich einen *Effekt*. Die dreistellige Relation

$$\delta \vdash e \Downarrow \nu$$

taugt nicht mehr für dieses Szenario.

- ▶ *Idee*: Wir erweitern die Auswertungsrelation zu einer *fünfstelligen* Relation

$$\delta \vdash \sigma \parallel e \Downarrow \nu \parallel \sigma'$$

Der Ausdruck  $e$  wertet zu  $\nu$  aus und bewirkt zusätzlich eine Zustandsänderung:  $\sigma$  ist der Speicher *vor* der Auswertung von  $e$  und  $\sigma'$  ist der Speicher *nach* der Auswertung.

- ▶ (Im Interesse der Lesbarkeit lassen wir die Umgebung  $\delta$  unter den Tisch fallen — ebenso ignorieren wir Ein- und Ausgaben.)

# 35. Dynamische Semantik

Wir müssen den Bereich der Werte um Adressen erweitern.

$$\nu ::= \dots$$

$$| a$$

Werte:  
Adresse

Auswertungsregeln:

$$\frac{\sigma \parallel e \Downarrow \nu \parallel \sigma'}{\sigma \parallel \mathbf{ref} \ e \Downarrow a \parallel \sigma', \{a \mapsto \nu\}} \quad a \notin \text{dom } \sigma'$$

$$\frac{\sigma \parallel e \Downarrow a \parallel \sigma'}{\sigma \parallel !e \Downarrow \sigma'(a) \parallel \sigma'}$$

$$\frac{\sigma \parallel e_1 \Downarrow a \parallel \sigma_1 \quad \sigma_1 \parallel e_2 \Downarrow \nu \parallel \sigma_2}{\sigma \parallel e_1 := e_2 \Downarrow () \parallel \sigma_2, \{a \mapsto \nu\}}$$

 !e ist ein *lesender Speicherzugriff*;  $e_1 := e_2$  ein *schreibender Speicherzugriff*. Beachte:  $\sigma'(a)$  ist stets definiert.

Ein- und  
Ausgabe

Zustand

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische  
Semantik

Vertiefung

Blick über den  
Tellerrand

Listenbeschrei-  
bungen

Kontrollstruktu-  
ren

Ausnahmen

## 35. Beispiel

$$\frac{\frac{\emptyset \parallel 0 \Downarrow 0 \parallel \emptyset}{\emptyset \parallel \mathit{ref} 0 \Downarrow a_1 \parallel \{a_1 \mapsto 0\}}}{\emptyset \parallel \mathit{ref} (\mathit{ref} 0) \Downarrow a_2 \parallel \{a_1 \mapsto 0, a_2 \mapsto a_1\}}$$

☞ Zunächst wird die Auswertung von  $\mathit{ref} 0$  und dann die Auswertung von  $0$  angestoßen;  $0$  wertet zu  $0$  aus, ohne den (leeren) Speicher zu verändern; dann wird eine Speicherzelle angelegt  $\{a_1 \mapsto 0\}$  und schließlich eine zweite  $\{a_2 \mapsto a_1\}$ .

## 35. Auswertungsregeln

☞ Da wir die Auswertungsrelation um zwei Argumente erweitert haben, müssen wir *alle* bisher aufgeführten Auswertungsregeln anpassen!

Jeder Teilausdruck kann einen Effekt haben:

*(trace := true; factorial 9) + (trace := false; factorial 10)*

Die Auswertungsregel wird wie folgt abgeändert.

$$\frac{\sigma_0 \parallel e_1 \Downarrow \nu_1 \parallel \sigma_1 \quad \sigma_1 \parallel e_2 \Downarrow \nu_2 \parallel \sigma_2}{\sigma_0 \parallel e_1 + e_2 \Downarrow \nu_1 + \nu_2 \parallel \sigma_2}$$

☞ Beide Teilausdrücke verändern potentiell den Speicher:  $e_1$  ändert  $\sigma_0$  zu  $\sigma_1$ ,  $e_2$  „sieht“ den modifizierten Speicher und ändert ihn zu  $\sigma_2$ .

## 35. Auswertungsregeln

☞ Allgemein werden Ausdrücke von *links nach rechts* abgearbeitet und Speicheränderungen werden in dieser Reihenfolge sichtbar. Die Auswertungsregel

$$\frac{e_1 \Downarrow \nu_1 \quad e_2 \Downarrow \nu_2 \quad \dots \quad e_n \Downarrow \nu_n}{e \Downarrow \nu}$$

wird wie folgt angepasst:

$$\frac{\sigma_0 \parallel e_1 \Downarrow \nu_1 \parallel \sigma_1 \quad \sigma_1 \parallel e_2 \Downarrow \nu_2 \parallel \sigma_2 \quad \dots \quad \sigma_{n-1} \parallel e_n \Downarrow \nu_n \parallel \sigma_n}{\sigma_0 \parallel e \Downarrow \nu \parallel \sigma_n}$$

☞ Der Zustand wird jeweils von Teilausdruck zu Teilausdruck weitergereicht. Man sagt auch, der Zustand wird *durchgefädelt* (engl. *threading*).

## 35. Vertiefung

☞ Eine Speicherzelle vom Typ  $\text{Ref}\langle \text{Nat} \rangle$  können wir verwenden, um zu zählen, wie oft eine bestimmte Funktion aufgerufen wird. (Zum Zwecke der Programmoptimierung.)

```
let counter = ref 0
```

```
let rec fibonacci (n : Nat) : Nat =
```

```
  counter := !counter + 1;
```

```
  if n ≤ 1 then
```

```
    n
```

```
  else
```

```
    fibonacci (n ÷ 1) + fibonacci (n ÷ 2)
```

## 35. Demo

Die Berechnung von *fibonacci n* ist aufwändig:

```
Mini) counter := 0; let f = fibonacci 10 in (f, !counter)  
(55, 177)
```

```
Mini) counter := 0; let f = fibonacci 20 in (f, !counter)  
(6765, 21891)
```

☞ Die Anzahl der Aufrufe übersteigt den Wert der Fibonaccifunktion in beiden Fällen.

# 35. Vertiefung

Wir können auch effektfrei zählen:

```

let rec counting-fibonacci (n : Nat) : Nat * Nat =
  if n ≤ 1 then
    (n, 1)
  else
    let (f1, c1) = counting-fibonacci (n ÷ 1)
    let (f2, c2) = counting-fibonacci (n ÷ 2)
    (f1 + f2, c1 + c2 + 1)
  
```

☞ Die Zahl  $c$  der Aufrufe für die Berechnung von *fibonacci*  $n$  ist fast doppelt so groß wie der Funktionswert von *fibonacci*  $(n + 1)$ , es gilt:  $c = 2 \cdot \text{fibonacci}(n + 1) - 1$ .

## 35. Memoisierung

*Problem:* Die gleichen Funktionswerte werden wiederholt berechnet: *fibonacci* 8 wird bei der Berechnung von *fibonacci* 20 insgesamt 233 mal (= *fibonacci* 12) neu ausgerechnet.

*Idee:* Wir merken uns die Aufrufe und greifen später auf die memorierten Werte zurück.

```
let memo-fibonacci = [| for i in 0..99 → fibonacci i |]
```

☞ Statt *fibonacci* *n* verwenden wir *memo-fibonacci*.[*n*].

## 35. Memoisierung

Abstrahieren wir von  $0..99$  und von *fibonacci*, erhalten wir

```
let memo (dom : Nat, func : Nat → 'v) : Nat → 'v =  
  let  
    memo-table = [| for i in 0..dom ÷ 1 → func i |]  
  in  
    fun (n : Nat) → memo-table.[n]  
let memo-fibonacci = memo (100, fibonacci)
```

*fsi Memo.fs*

...

☞ Unglücklicherweise dauert die Auswertung von *fibonacci* extrem (!) lange, da zunächst die Tabelle vollständig gefüllt wird.

☞ Eigentlich schwebt uns eine *bedarfsgetriebene* Füllung der Tabelle vor: erst wenn ein Funktionswert angefordert wird, berechnet *memo* den entsprechenden Tabelleneintrag.

## 35. Memoisierung

☞ Wenn wir Tabelleneinträge ändern wollen, müssen wir statt einem Array von Zahlen ein Array von Speicherzellen verwenden.

Jede Speicherzelle nimmt einen von zwei möglichen Zuständen an:

- ▶ „der Funktionswert wurde noch nicht berechnet“ oder
- ▶ „der Wert wurde berechnet und er lautet ...“.

☞ Die beiden Zustände können wir mit Elementen des Datentyps *Option* modellieren:

- ▶ *None* bzw.
- ▶ *Some value*, wobei *value* der berechnete Wert ist.

☞ Die Memotabelle hat somit insgesamt den furchteinflößenden Typ *Array*  $\langle \text{Ref} \langle \text{Option} \langle 'a \rangle \rangle \rangle$  statt *Array*  $\langle 'a \rangle$  wie bisher.

## 35. Memoisierung

```
let memo (dom : Nat, func : Nat → 'v) : Nat → 'v =  
  let  
    memo-table = [| for i in 0 .. dom ÷ 1 → ref None |]  
  in  
    fun (n : Nat) →  
      match !memo-table.[n] with  
      | None → let v = func n  
        do memo-table.[n] := Some v  
        v  
      | Some v → v  
let memo-fibonacci = memo (100, fibonacci)
```

*fsi Memo.fs*

```
Mini> memo-fibonacci 99
```

...

☞ Die Auswertung der beiden Definitionen dauert nunmehr keinen Fingerschnips.

☞ Jetzt tritt eine lange (!) Stille ein, wenn wir *memo-fibonacci 99* aufrufen: die *rekursiven* Aufrufe der Fibonacci Funktion werden *nicht* memoisiert.

## 35. Memoisierung

Was ist zu tun?

- ▶ Wertedefinitionen dürfen nicht rekursiv sein.
- ▶ Die Wertedefinition in eine Funktionsdefinition zu überführen,

```
let rec fibonacci (n : Nat) : Nat =
  memo (100, fun fib (n : Nat) →
    if n ≤ 1 then n
      else fibonacci (n ÷ 1) + fibonacci (n ÷ 2)) n
```

ist verführerisch, aber nicht sinnvoll: Für jeden rekursiven Aufruf wird eine *neue* Memotabelle angelegt!

- ▶ Wir können die rekursiven Aufrufe memoisieren, indem wir *memo* keine Funktion des Typs  $Nat \rightarrow 'v$  übergeben, sondern eine Funktion des Typs

```
(Nat → 'v) → (Nat → 'v)
```

die über die rekursiven Aufrufe abstrahiert.

Ein- und  
Ausgabe

Zustand

Motivation

Abstrakte Syntax

Statische Semantik

Dynamische

Semantik

Vertiefung

Blick über den

Tellerrand

Listenbeschrei-  
bungen

Kontrollstruktu-  
ren

Ausnahmen

## 35. Memoisierung

```
let rec-memo (dom : Nat, functional : (Nat → 'v) → (Nat → 'v)) : Nat → 'v =  
  let memo-table = [| for i in 0..dom ÷ 1 → ref None |]  
  
  let rec memo-f (n : Nat) : 'v =  
    match !memo-table.[n] with  
    | None → let v = functional memo-f n  
      do memo-table.[n] := Some v  
      v  
    | Some v → v  
  
  in  
  memo-f  
  
let memo-fibonacci =  
  rec-memo (100,  
    fun fib → fun n →  
      if n ≤ 1  
      then n  
      else fib (n ÷ 1) + fib (n ÷ 2))
```

# 35. Memoisierung

*fsi Memo.fs*

Mini) *memo-fibonacci* 99

354224848179261915075

☞ Der Lohn der Anstrengungen: sowohl die Definition von *memo-fibonacci* als auch alle Aufrufe von *memo-fibonacci* sind in Windeseile ausgerechnet; auch die rekursiven Aufrufe füllen die Tabelle.