

Teil VII

Effekte

VII Effekte
Ralf Hinze

Ein- und
Ausgabe
Zustand
Listenbeschrei-
bungen
Kontrollstruktu-
ren
Ausnahmen

707

33. Knobelaufgabe #18

Ist es möglich, ein *nicht-rekursives* Programm zu schreiben, das *nicht terminiert*?

Nicht-rekursiv bedeutet, dass in dem Programm weder rekursive Funktionsdefinitionen

```
let rec f (x1 : t1) : t2 = ... f ...
```

noch rekursive Variantentypen

```
type T = | ... | ... T ... | ...
```

verwendet werden dürfen.

VII Effekte
Ralf Hinze

Ein- und
Ausgabe
Zustand
Listenbeschrei-
bungen
Kontrollstruktu-
ren
Ausnahmen

708

33. Gliederung

- 34 Ein- und Ausgabe
- 35 Zustand
- 36 Listenbeschreibungen
- 37 Kontrollstrukturen
- 38 Ausnahmen

VII Effekte
Ralf Hinze

Ein- und
Ausgabe
Zustand
Listenbeschrei-
bungen
Kontrollstruktu-
ren
Ausnahmen

709

33. Lernziele

Nach Durcharbeitung dieses Kapitels sollten Sie

- ▶ die Semantik externer Effekte erklären können,
- ▶ das Konzept des Speichers verstanden haben,
- ▶ Kontrollstrukturen kennen und verwenden können,
- ▶ Ausnahmen und deren Semantik erklären können,
- ▶ effektfreie und effektvolle Ausdrücke unterscheiden können.

VII Effekte
Ralf Hinze

Ein- und
Ausgabe
Zustand
Listenbeschrei-
bungen
Kontrollstruktu-
ren
Ausnahmen

710

33. Überblick

Dieses Kapitel ist dem Studium von Effekten gewidmet.

- ▶ *Bisher:* Eine Mini-F# Funktion ist eine Funktion im mathematischen Sinne: das Funktionsargument bestimmt das Funktionsergebnis.
- ▶ Mathematische Funktionen sind im gewissen Sinne autistisch.
- ▶ Im folgenden erweitern wir die Idee des Rechnens: ein Ausdruck kann neben dem Wert zusätzlich einen *Effekt* haben.
- ▶ *Externe Effekte:*
 - ▶ Ausgabe auf dem Bildschirm,
 - ▶ die Anforderung einer Eingabe,
 - ▶ das Einlesen von Sensordaten,
 - ▶ die Steuerung eines Motors usw.
- ▶ *Interne Effekte:*
 - ▶ eine Rechnung kann von einem Gedächtnis abhängen oder das Gedächtnis verändern;
 - ▶ eine Rechnung kann ergebnislos abgebrochen werden und an anderer Stelle wiederaufgenommen werden.

VII Effekte
Ralf Hinze

Ein- und
Ausgabe
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

711

33. Eine Warnung vorneweg

☞ Effekte verändern die Natur des Rechnens!

- ▶ Ist eine Funktion effektiv, dann handelt es sich nicht mehr um eine Funktion im mathematischen Sinne.
- ▶ Eine effektvolle Funktion kann bei gleichen Argumenten unterschiedliche Resultate liefern!
- ▶ *Gefahr:*
Setzt man die neuen Sprachkonstrukte nicht mit Bedacht ein, dann
 - ▶ leidet die Lesbarkeit von Programmen;
 - ▶ leidet die Wartbarkeit von Programmen.

Wenn eine Funktion auf vielfältigen Wegen mit ihrer Umwelt interagiert, dann kann die Funktion nicht mehr isoliert verstanden werden, sondern die vielfältigen Verflechtungen müssen zusätzlich berücksichtigt werden.

VII Effekte
Ralf Hinze

Ein- und
Ausgabe
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

712

34. Motivation

Aufgabe: In Kapitel 3 haben wir ein 2-Personenspiel programmiert, bei dem Spielerin B eine von Spieler A ausgedachte Zahl raten musste. Beide Parteien wurden bisher vom Rechner gestellt. Das wollen wir jetzt ändern: die Benutzer*in soll die Rolle von Spieler A übernehmen.

Natürlich wird Spieler A weiterhin durch eine Mini-F# Funktion realisiert,

```
let human-player (guess : Nat) : Bool
```

aber es soll eine Funktion sein, die über Ein- und Ausgaben mit der Benutzer*in interagiert und deren Antworten weiterleitet.

Dazu benötigen wir grundlegende Funktionen zur Ein- und Ausgabe.

VII Effekte
Ralf Hinze

Ein- und
Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

713

34. Demo

```
Mini> player-B (human-player, 0, 99)  
Ist die Zahl gleich oder kleiner als 49? ja  
Ist die Zahl gleich oder kleiner als 24? nein  
Ist die Zahl gleich oder kleiner als 37? nein  
Ist die Zahl gleich oder kleiner als 43? nein  
Ist die Zahl gleich oder kleiner als 46? nein  
Ist die Zahl gleich oder kleiner als 48? ja  
Ist die Zahl gleich oder kleiner als 47? ja  
47
```

☞ Nach sieben Runden hat der Rechner die Zahl ermittelt.

VII Effekte
Ralf Hinze

Ein- und
Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

714

34. Ausgabe

Ausgaben auf dem Bildschirm werden mit Hilfe der Funktion *putstring* getätigt. Der Aufruf

```
putstring "Hello, world!"
```

wertet zu dem leeren Tupel '()' aus und hat zusätzlich den Effekt, dass der String "Hello, world!" ausgegeben wird.

☞ Die Funktion *putstring* ist das erste Beispiel für eine nicht-mathematische Funktion. Der Funktionswert steht schon vor dem Aufruf fest; das Verhalten entspricht dem der Funktion *fun* ($s : \text{String} \rightarrow ()$).

☞ Die Funktion *putstring* wird allein wegen ihres Effektes aufgerufen.

VII Effekte
Ralf Hinze

Ein- und Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

715

34. Eingabe

Eingaben von der Tastatur können mit Hilfe der Funktion *getline* eingefangen werden. Der Aufruf

```
getline ()
```

liest eine einzelne Zeile ein, eine Folge von Zeichen, die von einem Zeilenvorschub abgeschlossen wird. Der String *ohne* den Zeilenvorschub wird als Ergebnis zurückgegeben.

☞ Auch *getline* ist keine mathematische Funktion. Wäre sie eine, dann müsste sie stets den gleichen String zurückgeben.

VII Effekte
Ralf Hinze

Ein- und Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

716

34. „Query the user“

Wir können *putstring* und *getline* kombinieren, um eine Funktion zu programmieren, die die Benutzer*in zu einer Eingabe auffordert.

```
let query (prompt : String) : String =  
  let () = putstring prompt in getline ()
```

☞ Die lokale Bindung *let* $() = e_1$ *in* e_2 dient dazu, die Auswertung von zwei Ausdrücken und damit das Auftreten von Effekten zu *sequentialisieren*:

- ▶ Zunächst wird e_1 ausgerechnet,
- ▶ dann wird das Ergebnis mit dem Muster '()' abgeglichen,
- ▶ anschließend wird e_2 ausgerechnet.

VII Effekte
Ralf Hinze

Ein- und Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

717

34. Syntaktischer Zucker

☞ „Bindungen ohne Bindungen“ können mit dem bekannten und beliebten Semikolonoperator abgekürzt werden.

```
let query (prompt : String) : String =  
  putstring prompt; getline ()
```

Weiterhin erlauben wir die Definition *let* $() = e$ mit *do* e abzukürzen. (Wird in Teil VIII häufiger verwendet.)

VII Effekte
Ralf Hinze

Ein- und Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

718

34. Motivation

Mit Hilfe von *query* können wir *human-player* kurz und knapp definieren.

```
let human-player (guess : Nat) : Bool =  
  query ("Ist die Zahl gleich oder kleiner als "  
    ^ show guess ^ "? ") = "ja"
```

Der Ratekandidat *guess* wird ausgegeben; die Eingabe der Benutzer*in wird in einen Booleschen Wert verwandelt.

Zum Knobeln: Kann die menschliche Gegenspielerin *mogeln*?

VII Effekte
Ralf Hinze

Ein- und
Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische
Semantik
Vertiefung
Zustand
Listenbeschrei-
bungen
Kontrollstruktu-
ren
Ausnahmen

719

34. Reihenfolge

Effekte verändern die Natur des Rechnens: die Reihenfolge und die Multiplizität von Rechnungen spielen nunmehr eine Rolle.

Bisher: Die Komponenten des Paarausdrucks

```
(factorial 9, factorial 10)
```

konnten in beliebiger Reihenfolge ausgerechnet werden.

Jetzt: Wenn *factorial* zusätzlich einen Effekt hat, dann spielt die Reihenfolge der Teilrechnungen sehr wohl eine Rolle.

VII Effekte
Ralf Hinze

Ein- und
Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische
Semantik
Vertiefung
Zustand
Listenbeschrei-
bungen
Kontrollstruktu-
ren
Ausnahmen

720

34. Multiplizität

Ebenso ist relevant, wie oft eine Rechnung durchgeführt wird.

Bisher: Der Ausdruck

```
let f = factorial 9 in (f, f * 10)
```

ist äquivalent zu dem Ausdruck

```
(factorial 9, factorial 10)
```

Jetzt: Die Ausdrücke haben zwar den gleichen Wert aber wahrscheinlich einen anderen Effekt.

VII Effekte
Ralf Hinze

Ein- und
Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische
Semantik
Vertiefung
Zustand
Listenbeschrei-
bungen
Kontrollstruktu-
ren
Ausnahmen

721

34. Demo

```
let rec factorial (n : Nat) : Nat =  
  putstring (show n ^ "\n");  
  if n = 0 then 1 else factorial (n - 1) * n
```

```
Mini> (factorial 0, factorial 1)
```

```
0
```

```
1
```

```
0
```

```
(1, 1)
```

```
Mini> (factorial 1, factorial 0)
```

```
1
```

```
0
```

```
0
```

```
(1, 1)
```

```
Mini> let f = factorial 0 in (f, f * 1)
```

```
0
```

```
(1, 1)
```

VII Effekte
Ralf Hinze

Ein- und
Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische
Semantik
Vertiefung
Zustand
Listenbeschrei-
bungen
Kontrollstruktu-
ren
Ausnahmen

722

34. Motivation

Beispiel: Eingabe von Personendaten.

```
let input-person () : Person =
  if contains (query "gender: ") ["f"; "female"] then
    Female { name = query "name:  " }
  else
    Male { name = query "name:  ";
           bald = contains (query "bald?: ") ["y"; "yes"] }
```

☞ Die Funktion $contains : String \rightarrow List \langle String \rangle \rightarrow Bool$ überprüft, ob das erste Argument in der angegebenen Liste enthalten ist.

VII Effekte
Ralf Hinze

Ein- und Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

723

34. Demo

```
Mini> input-person ()
gender : male
name   : Ralf
bald?  : yes
Male { name = "Ralf"; bald = true }
```

☞ Warum haben wir $input-person$ als Funktion des Typs $Unit \rightarrow Person$ definiert und nicht einfach als $Person$?

VII Effekte
Ralf Hinze

Ein- und Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

724

34. Abstrakte Syntax

Die Ein- und Ausgabeoperationen $putstring$ und $getline$ lassen sich auf Funktionen zurückführen, die ein einzelnes Zeichen ausgeben bzw. einlesen (zur Übung).

| | |
|---------------|-------------------------|
| $e ::= \dots$ | <i>Ausdrücke:</i> |
| $getchar\ e$ | Einlesen eines Zeichens |
| $putchar\ e$ | Ausgabe eines Zeichens |

VII Effekte
Ralf Hinze

Ein- und Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

725

34. Statische Semantik

Die Ein- und Ausgabeoperationen verarbeiten einzelne Zeichen.

$$\frac{\Sigma \vdash e : Unit}{\Sigma \vdash getchar\ e : Char} \quad \frac{\Sigma \vdash e : Char}{\Sigma \vdash putchar\ e : Unit}$$

VII Effekte
Ralf Hinze

Ein- und Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

726

34. Dynamische Semantik

- ▶ Die Auswertung verändert sich mit dem Einzug von Effekten.
- ▶ Wie müssen wir die Auswertungsregeln modifizieren, um Interaktionen mit der Umwelt modellieren zu können?
- ▶ Beweisregeln sind genauso wenig interaktiv wie mathematische Funktionen!
- ▶ 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 *vierstelligen* Relation

$$\delta \vdash e \Downarrow_t \nu$$

die eine Umgebung mit einem Ausdruck, einem *externen Effekt* t und einem Wert in Beziehung setzt.

- ▶ Was ist ein externer Effekt?

34. Ereignisse

Wir modellieren einen externen Effekt als *Sequenz* von Ereignissen, wobei ein einzelnes Ereignis die Ein- oder Ausgabe eines Zeichens ist.

$c \in \text{Unicode}$

$$t \in \text{Event} ::= \begin{array}{l} \text{Ereignis} \\ | \text{Eingabe von } c \\ | \text{Ausgabe von } c \end{array}$$

34. Auswertungsregeln

Auswertungsregeln:

$$\frac{\delta \vdash e \Downarrow_t ()}{\delta \vdash \text{getchar } e \Downarrow_{t \cdot \text{in}(c)} c}$$

$$\frac{\delta \vdash e \Downarrow_t c}{\delta \vdash \text{putchar } e \Downarrow_{t \cdot \text{out}(c)} ()}$$

34. Beispiel

$$\frac{\frac{\emptyset \vdash () \Downarrow_e ()}{\emptyset \vdash \text{getchar } () \Downarrow_{\text{in}(h)} 'h'}}{\emptyset \vdash \text{putchar } (\text{getchar } ()) \Downarrow_{\text{in}(h) \cdot \text{out}(h)} ()}$$

$\text{in}(h) \cdot \text{out}(h)$ ist nicht die einzige mögliche Ereignisfolge: auch $\text{in}(a) \cdot \text{out}(a)$ oder $\text{in}(1) \cdot \text{out}(1)$ usw. sind denkbar, nicht aber $\text{in}(h) \cdot \text{out}(a)$ oder $\text{in}(a) \cdot \text{out}(h)$.

Die Auswertungsrelation ist eine „echte“ Relation und trägt damit der Tatsache Rechnung, dass viele unterschiedliche Interaktionen mit der Benutzer*in möglich sind.

34. Auswertungsregeln

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

Jeder Teilausdruck kann einen Effekt haben:

$(\text{putstring "Hello, "; 4700}) + (\text{putstring "world!"; 11})$

Die Auswertungsregel wird wie folgt abgeändert.

$$\frac{\delta \vdash e_1 \Downarrow_{t_1} n_1 \quad \delta \vdash e_2 \Downarrow_{t_2} n_2}{\delta \vdash e_1 + e_2 \Downarrow_{t_1 \cdot t_2} n_1 + n_2}$$

☞ Beide Teilausdrücke haben einen Effekt, t_1 bzw. t_2 ; der kumulierte Effekt der Summe ist $t_1 \cdot t_2$. Somit treten die Effekte des ersten Summanden vor den Effekten des zweiten Summanden auf.

VII Effekte
Ralf Hinze

Ein- und Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

731

34. Auswertungsregeln

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

$$\frac{\delta_1 \vdash e_1 \Downarrow_{\nu_1} \quad \dots \quad \delta_n \vdash e_n \Downarrow_{\nu_n}}{\delta \vdash e \Downarrow_{\nu}}$$

wird wie folgt erweitert:

$$\frac{\delta_1 \vdash e_1 \Downarrow_{t_1} \nu_1 \quad \dots \quad \delta_n \vdash e_n \Downarrow_{t_n} \nu_n}{\delta \vdash e \Downarrow_{t_1 \dots t_n} \nu}$$

☞ Die Reihenfolge der Effekte wird durch die Konkatenation der Ereignissequenzen festgelegt.

VII Effekte
Ralf Hinze

Ein- und Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

732

34. Auswertungsregeln: Beispiel

Beispiel: Die Regel für *in*-Ausdrücke

$$\frac{\delta \vdash d \Downarrow_{\delta'} \quad \delta, \delta' \vdash e \Downarrow_{\nu}}{\delta \vdash (d \text{ in } e) \Downarrow_{\nu}}$$

wird zu

$$\frac{\delta \vdash d \Downarrow_{t'} \delta' \quad \delta, \delta' \vdash e \Downarrow_t \nu}{\delta \vdash d \text{ in } e \Downarrow_{t' \cdot t} \nu}$$

VII Effekte
Ralf Hinze

Ein- und Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

733

34. Auswertungsregeln: Diskussion

- ▶ Die Einführung von effektvollen Ausdrücken hat einen dramatischen Effekt auf die Semantik von Mini-F#.
- ▶ Die Auswertungsregeln legen nunmehr pedantisch fest, in welcher Reihenfolge ein Programm abgearbeitet wird.
- ▶ Das ist in gewisser Weise ein Rückschritt:

$e_1 + e_2$

konnte bis dato gleichzeitig oder im Fachjargon *parallel* ausgerechnet werden.

- ▶ Wenn wir weiterhin eine parallele Auswertung wegen des möglichen Geschwindigkeitsvorteils anstreben, dann müssen wir sicherstellen, dass e_1 keine Effekte hat: $\delta \vdash e_1 \Downarrow_{\epsilon} \nu_1$.
- ▶ Diese Eigenschaft ist wie viele andere nicht formal entscheidbar!
- ▶ Die Parallelisierung von Programmen ist eine der großen Herausforderungen der Informatik.
- ▶ „The free lunch is over.“

VII Effekte
Ralf Hinze

Ein- und Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

734

34. Modularität

☞ Interaktive Programme sind wegen ihrer Interaktionen schwieriger zu lesen und zu verstehen sind als effektfreie Programme.

Aus diesem Grund sollte man versuchen, Effekte auf einige wenige Funktionen zu beschränken und so viel wie möglich effektfrei zu rechnen.

Die Funktion *query* prüft die Eingabe nicht auf Plausibilität.

```
Mini) query "age: "  
age: Hello, world!  
"Hello, world!"
```

VII Effekte
Ralf Hinze

Ein- und Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

735

34. Eingabe mit Validierung

Idee: *query* mit einem *Validator* parametrisieren.

```
let checked-query (prompt : String,  
                  check : String → Result ⟨'value⟩) : 'value
```

☞ Ein *Validator* bildet einen *String* auf ein Element des folgenden Datentyps ab.

```
type Result ⟨'value⟩ =  
  | Okay of 'value  
  | Error of String
```

☞ Ist der *String* zulässig, wird *Okay value* zurückgegeben, wobei *value* der semantische Wert des *Strings* ist, zum Beispiel eine natürliche Zahl; schlägt die Validierung fehl, wird *Error msg* zurückgegeben, wobei *msg* eine aussagekräftige Fehlermeldung ist.

VII Effekte
Ralf Hinze

Ein- und Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

736

34. Eingabe mit Validierung

☞ Eine validierende Version von *query*:

```
let rec checked-query (prompt : String,  
                     check : String → Result ⟨'value⟩) : 'value =  
  match check (query (prompt ^ ": ")) with  
  | Okay v → v  
  | Error msg → putline ("*** " ^ msg);  
    checked-query (prompt, check)
```

☞ Es werden solange Eingaben angefordert, bis die Eingabe von *check* abgesegnet wird. Im Fehlerfall wird die Benutzer*in auf den Fehler hingewiesen.

VII Effekte
Ralf Hinze

Ein- und Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

737

34. Validierung

☞ *Validator*en sind einfache, effektfreie Funktionen:

```
let is-nat (s : String) : Result ⟨Nat⟩ =  
  if s <> "" && String.forall Char.IsDigit s then  
    Okay (Nat.Parse s)  
  else  
    Error "natural number expected"
```

☞ *forall* : (Char → Bool) → String → Bool überprüft, ob alle Zeichen eines *Strings* das angegebene Prädikat erfüllen.

☞ *is-nat* ist im Prinzip ein einfacher Parser!

VII Effekte
Ralf Hinze

Ein- und Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

738

34. Demo

```
Mini) checked-query ("age", is-nat)
age: Hello, world !
*** natural number expected
age: 4711
4711
```

Wir sollten zusätzlich verlangen, dass die Altersangabe kleiner als 123 ist.

```
Mini) checked-query ("age", both (is-nat, is-less 123))
age: Ralf
*** natural number expected
age: 4711
*** number must be less than 123
age: 41
41
```

VII Effekte
Ralf Hinze

Ein- und Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

739

34. Validierung

Die Funktion *both* kombiniert zwei Validatoren: *both (is-nat, is-less 123)* fordert, dass die Eingabe eine Folge von Ziffern ist *und* dass die korrespondierende Zahl kleiner als 123 ist.

```
let both (first : 'a → Result ⟨'b⟩,
         second : 'b → Result ⟨'c⟩) : 'a → Result ⟨'c⟩ =
  fun x → match first x with
  | Okay y → second y
  | Error msg → Error msg
```

☞ Um den String nicht wiederholt in eine Zahl umwandeln zu müssen, wird der semantische Wert des ersten Validators an den zweiten Validator weitergereicht:

- ▶ *is-nat* : *String* → *Result* ⟨*Nat*⟩
- ▶ *is-less 123* : *Nat* → *Result* ⟨*Nat*⟩
- ▶ *both (is-nat, is-less 123)* : *String* → *Result* ⟨*Nat*⟩

VII Effekte
Ralf Hinze

Ein- und Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

740

34. Validierung

Die Funktion *is-less* kleidet die Vergleichsoperation *<* in *Okay* bzw. *Error* ein.

```
let is-less (n : Nat) : Nat → Result ⟨Nat⟩ = fun m →
  if m < n then Okay m
  else Error ("number must be less than " ^ show n)
```

VII Effekte
Ralf Hinze

Ein- und Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

741

34. Anwendungen

☞ Mit Hilfe von *checked-query* können wir z. B. Eingaben auf eine vorgegebene Auswahl von Strings beschränken.

```
let choice (prompt : String, choices : List ⟨String⟩) : String =
  checked-query (prompt,
  fun s →
    if contains s choices
    then Okay s
    else Error ("choices: " ^ concat " ", " choices))
```

☞ Die Funktion *concat* : *String* → *List* ⟨*String*⟩ → *String* konkateniert eine Liste von Strings und fügt zwischen je zwei Elemente den angegebenen Separator, erstes Argument, ein.

VII Effekte
Ralf Hinze

Ein- und Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische Semantik
Vertiefung
Zustand
Listenbeschreibungen
Kontrollstrukturen
Ausnahmen

742

34. Anwendungen

☞ Mit diesen Zutaten können wir die Funktion *input-person* neu definieren, diesmal inklusive Validierung der getätigten Eingaben.

```
let input-person () : Person =  
  if contains  
    (choice ("gender", ["f", "m", "female", "male"]))  
    ["f", "female"]  
  then  
    Female { name = checked-query ("name ", is-name) }  
  else  
    Male { name = checked-query ("name ", is-name);  
          bald = contains  
            (choice ("bald? ", ["y", "n", "yes", "no"]))  
            ["y", "yes"] }
```

☞ Die Funktion *is-name* überprüft, ob die Eingabe ein gültiger Name ist (zur Übung).

VII Effekte
Ralf Hinze

Ein- und
Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische
Semantik
Vertiefung
Zustand
Listenbeschrei-
bungen
Kontrollstruktu-
ren
Ausnahmen

743

34. Demo

```
Mini) input-person ()  
gender : sehr maskulin  
*** choices : f, m, female, male  
gender : m  
name : Ralf  
bald : ja  
*** choices : y, n, yes, no  
bald : y  
Male { name = "Ralf"; bald = true }
```

VII Effekte
Ralf Hinze

Ein- und
Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische
Semantik
Vertiefung
Zustand
Listenbeschrei-
bungen
Kontrollstruktu-
ren
Ausnahmen

744

34. Trace

☞ Ausgaben können auch beim Testen von Programmen nützliche Dienste leisten.

```
let rec factorial (n : Nat) : Nat =  
  if n = 0 then  
    Return 1  
  else  
    Return (Call factorial (n - 1) * n)
```

Mit Hilfe von *Return* wird der (Rückgabe-) Wert eines Ausdrucks protokolliert; mit Hilfe von *Call* der aktuelle Parameter einer Funktion.

VII Effekte
Ralf Hinze

Ein- und
Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische
Semantik
Vertiefung
Zustand
Listenbeschrei-
bungen
Kontrollstruktu-
ren
Ausnahmen

745

34. Demo

```
Mini) Call factorial 10  
call 10  
call 9  
call 8  
call 7  
call 6  
call 5  
call 4  
call 3  
call 2  
call 1  
call 0  
return 1  
return 1  
return 2  
return 6  
return 24  
return 120  
return 720  
return 5040  
return 40320  
return 362880  
return 3628800
```

VII Effekte
Ralf Hinze

Ein- und
Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische
Semantik
Vertiefung
Zustand
Listenbeschrei-
bungen
Kontrollstruktu-
ren
Ausnahmen

746

34. Trace

Die *Fibonacci-Funktion* zeigt ein lebhafteres Auf und Ab.

```
let rec fibonacci (n : Nat) : Nat =  
  Return (if n ≤ 1 then  
    n  
  else  
    Call fibonacci (n ÷ 1) + Call fibonacci (n ÷ 2))
```

Die Funktion ist nach dem italienischen Mathematiker Leonardo da Pisa, genannt Fibonacci, benannt, der mit dieser Funktion das Wachstum einer Kaninchenpopulation modellierte.

Die Funktion beantwortet die Frage „Wie viele Kaninchenpaare entstehen nach n Monaten aus einem einzigen Paar, wenn jedes Paar ab dem zweiten Lebensmonat ein weiteres Paar auf die Welt bringt?“.

VII Effekte
Ralf Hinze

Ein- und
Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische
Semantik
Vertiefung
Zustand
Listenbeschrei-
bungen
Kontrollstruktu-
ren
Ausnahmen

747

34. Demo

```
Mini) Call fibonacci 4  
call 4  
call 3  
call 2  
call 1  
return 1  
call 0  
return 0  
return 1  
call 1  
return 1  
return 2  
call 2  
call 1  
return 1  
call 0  
return 0  
return 1  
return 3  
3
```

VII Effekte
Ralf Hinze

Ein- und
Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische
Semantik
Vertiefung
Zustand
Listenbeschrei-
bungen
Kontrollstruktu-
ren
Ausnahmen

748

34. Trace

```
let Return (x : 'a) : 'a =  
  putline ("return " ^ show x); x
```

Wertmäßig ist *Return* die Identität: das Argument wird als Ergebnis zurückgegeben.

```
let Call (f : 'a → 'b) : 'a → 'b =  
  fun x → putline ("call " ^ show x); f x
```

Wertmäßig ist *Call* die Identität von Funktionen.

VII Effekte
Ralf Hinze

Ein- und
Ausgabe
Motivation
Abstrakte Syntax
Statische Semantik
Dynamische
Semantik
Vertiefung
Zustand
Listenbeschrei-
bungen
Kontrollstruktu-
ren
Ausnahmen

749