Compilerbau
Vorlesung
WS 2008–11,13,15,17,19,SS 22

Johannes Waldmann, HTWK Leipzig

10. Juni 2022

Beispiel: C-Compiler

- ```
  int gcd (int x, int y) {
    while (y>0) { int z = x%y; x = y; y = z; }
    return x; }
  ```
- `gcc -S -O2 gcd.c` erzeugt `gcd.s`:

  ```
  .L3:    movl    %edx, %r8d ; cltd  ; idivl   %r8d
          movl    %r8d, %eax  ;   testl   %edx, %edx
          jg      .L3
  ```

  Ü: was bedeutet `cltd`, warum ist es notwendig?
  Ü: welche Variable ist in welchem Register?
- identischer (!) Assembler-Code für

  ```
  int gcd_func (int x, int y) {
    return y > 0 ? gcd_func (y,x % y) : x;
  }
  ```
- vollständige Quelltexte:
  `https://gitlab.imn.htwk-leipzig.de/waldmann/cb-ws19/blob/master/gcd.c`
- Bsp Java-Kompilation: `https://www.imn.htwk-leipzig.de/~waldmann/etc/safe-speed/`
  ```

Sprachverarbeitung

- mit Compiler:
  - Quellprogramm $\rightarrow$ Compiler $\rightarrow$ Zielprogramm
  - Eingaben $\rightarrow$ Zielprogramm $\rightarrow$ Ausgaben
- mit Interpreter:
  - Quellprogramm, Eingaben $\rightarrow$ Interpreter $\rightarrow$ Ausgaben
- Mischform:
  - Quellprogramm $\rightarrow$ Compiler $\rightarrow$ Zwischenprogramm
  - Zwischenprogramm, Eingaben $\rightarrow$ virtuelle Maschine $\rightarrow$ Ausgaben

Gemeinsamkeit: syntaxgesteuerte Semantik (Ausführung bzw. Übersetzung)

Inhalt der Vorlesung

Konzepte von Programmiersprachen

- Semantik von einfachen (arithmetischen) Ausdrücken
- lokale Namen, • Unterprogramme (Lambda-Kalkül)
- Zustandsänderungen (imperative Prog.)
- Continuations zur Ablaufsteuerung

realisieren durch

- Interpretation, • Kompilation

Hilfsmittel:

- Theorie: Inferenzsysteme (f. Auswertung, Typisierung)
- Praxis: Haskell, Monaden (f. Auswertung, Parser)

Literatur

- Franklyn Turbak, David Gifford, Mark Sheldon: *Design Concepts in Programming Languages*, MIT Press, 2008. http://cs.wellesley.edu/~fturbak/
- Guy Steele, Gerald Sussman: *Lambda: The Ultimate Imperative*, MIT AI Lab Memo AIM-353, 1976 (the original 'lambda papers', https://web.archive.org/web/20030603185429/http://library.readscheme.org/page1.html)
- Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman: *Compilers: Principles, Techniques, and Tools (2nd edition)* Addison-Wesley, 2007, http://dragonbook.stanford.edu/
- J. Waldmann: *Das M-Wort in der Compilerbauvorlesung*, Workshop der GI-Fachgruppe Prog. Spr. und Rechnerkonzepte, 2013 http://www.imn.htwk-leipzig.de/~waldmann/talk/13/fg214/

Anwendungen von Techniken des Compilerbaus

- Implementierung höherer Programmiersprachen
- architekturspezifische Optimierungen (Parallelisierung, Speicherhierarchien)
- Entwurf neuer Architekturen (RISC, spezielle Hardware)
- Programm-Übersetzungen (Binär-Übersetzer, Hardwaresynthese, Datenbankanfragesprachen)
- Software-Werkzeuge (z.B. Refaktorisierer)
- domainspezifische Sprachen

Organisation der Vorlesung

- pro Woche eine Vorlesung, eine Übung.
- in Vorlesung, Übung und Hausaufgaben:
  - Theorie,
  - Praxis: Quelltexte (weiter-)schreiben (erst Interpreter, dann Compiler)
- Prüfungszulassung: regelmäßiges und erfolgreiches Bearbeiten von Übungsaufgaben
- Prüfung: Klausur (120 min, keine Hilfsmittel)

Beispiel: Interpreter f. arith. Ausdrücke

```
data Exp = Const Integer
         | Plus Exp Exp | Times Exp Exp
    deriving ( Show )

ex1 :: Exp
ex1 =
  Times ( Plus ( Const 1 ) ( Const 2 ) ) ( Const 3 )

value :: Exp -> Integer
value x = case x of
    Const i -> i
    Plus x y -> value x + value y
    Times x y -> value x * value y
```

das ist syntax-gesteuerte Semantik:
Wert des Terms wird aus Werten der Teilterme kombiniert

Beispiel: lokale Variablen und Umgebungen

```
data Exp = ... | Let String Exp Exp | Ref String
ex2 :: Exp
ex2 = Let "x" ( Const 3 )
      ( Times ( Ref "x" ) (Ref "x" ) )
type Env = ( String -> Integer )
extend n w e = \ m -> if m == n then w else e m
value :: Env -> Exp -> Integer
value env x = case x of
    Ref n -> env n
    Let n x b -> value (extend n (value env x) env) b
    Const i -> i
    Plus x y -> value env x + value env y
    Times x y -> value env x * value env y
test2 = value (\ _ -> 42) ex2
```

Übung (Haskell)

- Wiederholung Haskell
  - Interpreter/Compiler: ghci `http://haskell.org/`
  - Funktionsaufruf nicht `f(a,b,c+d)`, sondern
    `f a b (c+d)`
  - Konstruktor beginnt mit Großbuchstabe und ist auch eine Funktion
- Wiederholung funktionale Programmierung/Entwurfsmuster
  - rekursiver algebraischer Datentyp (ein Typ, mehrere Konstruktoren)
    (OO: Kompositum, ein Interface, mehrere Klassen)
  - rekursive Funktion
- Wiederholung Pattern Matching:
  - beginnt mit `case ... of`, dann Zweige
  - jeder Zweig besteht aus Muster und Folge-Ausdruck
  - falls das Muster paßt, werden die Mustervariablen gebunden und der Folge-Ausdruck auswertet

Übung (Interpreter)

- Benutzung:
  - Beispiel für die Verdeckung von Namen bei geschachtelten Let
  - Beispiel dafür, daß der definierte Name während seiner Definition nicht sichtbar ist
- Erweiterung:
  Verzweigungen mit C-ähnlicher Semantik:
  Bedingung ist arithmetischer Ausdruck, verwende 0 als Falsch und alles andere als Wahr.

```
data Exp = ...
         | If Exp Exp Exp
```

- Quelltext-Archiv: siehe `https://gitlab.imn.htwk-leipzig.de/waldmann/cb-ws19`

Motivation

- inferieren $=$ ableiten
- Inferenzsystem I, Objekt O,
  Eigenschaft $I \vdash O$ (in I gibt es eine Ableitung für O)
- damit ist I eine *Spezifikation* einer Menge von Objekten
- man ignoriert die *Implementierung* ($=$ das Finden von Ableitungen)
- Anwendungen im Compilerbau:
  Auswertung von Programmen, Typisierung von Programmen

Definition

ein *Inferenz-System* I besteht aus

- Regeln (besteht aus Prämissen, Konklusion)
  Schreibweise $\frac{P_1,\ldots,P_n}{K}$
- Axiomen ($=$ Regeln ohne Prämissen)

eine *Ableitung* für F bzgl. I ist ein Baum:

- jeder Knoten ist mit einer Formel beschriftet
- jeder Knoten (mit Vorgängern) entspricht Regel von I
- Wurzel (Ziel) ist mit F beschriftet

Def: $I \vdash F :\iff \exists$ I-Ableitungsbaum mit Wurzel F.

Regel-Schemata

- um unendliche Menge zu beschreiben, benötigt man unendliche Regelmengen
- diese möchte man endlich notieren
- ein *Regel-Schema* beschreibt eine (mglw. unendliche) Menge von Regeln, Bsp: $\dfrac{(x, y)}{(x - y, y)}$
- Schema wird *instantiiert* durch Belegung der Schema-Variablen
  Bsp: Belegung $x \mapsto 13, y \mapsto 5$
  ergibt Regel $\dfrac{(13, 5)}{(8, 5)}$

Inferenz-Systeme (Beispiel 1)

- Grundbereich $=$ Zahlenpaare $\mathbb{Z} \times \mathbb{Z}$
- Axiom:

$$\frac{}{(13, 5)}$$

- Regel-Schemata:

$$\frac{(x, y)}{(x - y, y)}, \quad \frac{(x, y)}{(x, y - x)}$$

kann man $(1, 1)$ ableiten? $(-1, 5)$? $(2, 4)$?

Das Ableiten als Hüll-Operation

- für Inferenzsystem I über Bereich O
  und Menge $M \subseteq O$ definiere
  $M^{\vdash} := \{K \mid \frac{P_1, \ldots, P_n}{K} \in I, P_1 \in M, \ldots, P_n \in M\}$.
- Übung: beschreibe $\emptyset^{\vdash}$.
- Satz: $\{F \mid I \vdash F\}$ ist die bzgl. $\subseteq$ kleinste Menge M mit $M^{\vdash} \subseteq M$
  Bemerkung: „die kleinste“: Existenz? Eindeutigkeit?
- Satz: $\{F \mid I \vdash F\} = \bigcup_{i \geq 0} M_i$ mit $M_0 = \emptyset, \forall i : M_{i+1} = M_i^{\vdash}$

Aussagenlogische Resolution

Formel $(A \vee \neg B \vee \neg C) \wedge (C \vee D)$ in konjunktiver Normalform dargestellt als $\{\{A, \neg B, \neg C\}, \{C, D\}\}$
(Formel = Menge von Klauseln, Klausel = Menge von Literalen, Literal = Variable oder negierte Variable)

folgendes Inferenzsystem heißt *Resolution*:

- Axiome: Klauselmenge einer Formel,
- Regel:
  - Prämissen: Klauseln K_1, K_2 mit $v \in K_1, \neg v \in K_2$
  - Konklusion: $(K_1 \setminus \{v\}) \cup (K_2 \setminus \{\neg v\})$

Eigenschaft (Korrektheit): wenn $\frac{K_1, K_2}{K}$, dann $K_1 \wedge K_2 \rightarrow K$.

Resolution (Vollständigkeit)

die Formel (Klauselmenge) ist nicht erfüllbar $\iff$ *die leere Klausel ist durch Resolution ableitbar.*
Bsp: $\{p, q, \neg p \vee \neg q\}$
Beweispläne:

- $\Rightarrow$: Gegeben ist die nicht erfüllbare Formel. Gesucht ist eine Ableitung für die leere Klausel. Methode: Induktion nach Anzahl der in der Formel vorkommenden Variablen.
- $\Leftarrow$: Gegeben ist die Ableitung der leeren Klausel. Zu zeigen ist die Nichterfüllbarkeit der Formel. Methode: Induktion nach Höhe des Ableitungsbaumes.

Die Abtrennungsregel (modus ponens)

. . . ist das Regelschema $\dfrac{P \to Q, P}{Q}$,

Der *Hilbert-Kalkül* für die Aussagenlogik
ist das Inferenz-System mit modus ponens
und Axiom-Schemata wie z. B.

- $A \to (B \to A)$
- $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$
- $(\neg A \to \neg B) \to ((\neg A \to B) \to A)$

(es gibt verschiedene Varianten des Kalküls) — Man zeigt:

- Korrektheit: jede ableitbare Aussage ist allgemeingültig
- Vollständigkeit: jede allgemeing. Aussage ist ableitbar

Inferenz von Werten

- Grundbereich: Aussagen der Form wert(p, z) mit $p \in$ `Exp` , $z \in \mathbb{Z}$

```
data Exp = Const Integer
         | Plus Exp Exp
         | Times Exp Exp
```

- Axiome: wert(`Const`z, z)
- Regeln:

$$\frac{\text{wert}(X, a), \text{wert}(Y, b)}{\text{wert}(\texttt{Plus}\ X\ Y, a + b)}, \quad \frac{\text{wert}(X, a), \text{wert}(Y, b)}{\text{wert}(\texttt{Times}\ X\ Y, a \cdot b)}, \ldots$$

Umgebungen (Spezifikation)

- Grundbereich: Aussagen der Form $\mathrm{wert}(E, p, z)$
  (in Umgebung E hat Programm p den Wert z)
  Umgebungen konstruiert aus $\emptyset$ und $E[v := b]$
- Regeln für Operatoren $\dfrac{\mathrm{wert}(E, X, a), \mathrm{wert}(E, Y, b)}{\mathrm{wert}(E, \texttt{Plus}XY, a + b)}, \ldots$
- Regeln für Umgebungen

  $$\frac{}{\mathrm{wert}(E[v := b], v, b)}, \quad \frac{\mathrm{wert}(E, v', b')}{\mathrm{wert}(E[v := b], v', b')} \text{ für } v \neq v'$$
- Regeln für Bindung: $\dfrac{\mathrm{wert}(E, X, b), \mathrm{wert}(E[v := b], Y, c)}{\mathrm{wert}(E, \texttt{let } v = X \texttt{ in } Y, c)}$

Umgebungen (Implementierung)

Umgebung ist (partielle) Funktion von Name nach Wert
Realisierungen: `type Env = String -> Integer`
Operationen:

- `empty :: Env` leere Umgebung
- `lookup :: Env -> String -> Integer`
  Notation: $e(x)$
- `extend :: String -> Integer -> Env -> Env`
  Notation: $e[v := z]$

Beispiel

```
lookup (extend  "y" 4 (extend "x" 3 empty)) "x"
```

entspricht $(\emptyset[x := 3][y := 4])x$

Semantische Bereiche

bisher: Wert eines Ausdrucks ist Zahl.
jetzt erweitern (Motivation: if-then-else mit richtigem Typ):

```
data Val = ValInt Int
         | ValBool Bool
```

Dann brauchen wir auch

- `data Val = ... | ValErr String`
- vernünftige Notation (Kombinatoren) zur Einsparung von Fallunterscheidungen bei Verkettung von Rechnungen

  ```
  with_int  :: Val -> (Int  -> Val) -> Val
  ```

Continuations

Programmablauf-Abstraktion durch Continuations:
Definition:

```
with_int  :: Val -> (Int  -> Val) -> Val
with_int v k = case v of
    ValInt i -> k i
    _ -> ValErr "expected ValInt"
```

Benutzung:

```
value env x = case x of
    Plus l r ->
        with_int ( value env l ) $ \ i ->
        with_int ( value env r ) $ \ j ->
        ValInt ( i + j )
```

Aufgaben

- Bool:
  - Boolesche Konstanten.
  - relationale Operatoren (==, <, o.ä.),
  - Inferenz-Regel(n) für Auswertung des `If`
  - Implementierung der Auswertung von if/then/else mit `with_bool`,
- Refaktorisierung
  - neue Hilfsfunktionen zur kürzeren Notation der Auswertung von `Plus`,`Times`,
  - evtl. relationale Operatoren

Beispiele

- in verschiedenen Prog.-Sprachen gibt es verschiedene Formen von Unterprogrammen:
  - Prozedur, sog. Funktion, Methode, Operator, Delegate, anonymes Unterprogramm
- allgemeinstes Modell:
  - Kalkül der anonymen Funktionen (Lambda-Kalkül),

Interpreter mit Funktionen

- abstrakte Syntax:

```
data Exp = ...
  | Abs { par :: Name , body :: Exp }
  | App { fun :: Exp  , arg :: Exp }
```

- konkrete Syntax:

```
let { f = \ x -> x * x }  in  f (f 3)
```

- konkrete Syntax (Alternative):

```
let { f x = x * x }  in  f (f 3)
```

Semantik (mit Funktionen)

- erweitere den Bereich der Werte:

```
data Val = ... | ValFun ( Val -> Val )
```

- erweitere Interpreter:

```
value :: Env -> Exp -> Val
value env x = case x of
    ... | Abs n b -> _ | App f a -> _
```

- mit Hilfsfunktion `with_fun :: Val -> ...`
- Testfall (in konkreter Syntax)

```
let { x = 4 } in  let { f = \ y -> x * y }
    in  let { x = 5 }  in  f x
```

Let und Lambda

- `let { x = A } in Q`
  kann übersetzt werden in
  `(\ x -> Q) A`
- `let { x = a , y = b } in Q`
  wird übersetzt in . . .

- beachte: das ist nicht das `let` aus Haskell

Mehrstellige Funktionen

. . . simulieren durch einstellige:

- mehrstellige Abstraktion:

```
\ x y z -> B := \x -> (\y -> (\z -> B ))
```

- mehrstellige Applikation:

```
f P Q R     :=    ((f P) Q) R
```

(die Applikation ist links-assoziativ)

- der Typ einer mehrstelligen Funktion:

```
T1 -> T2 -> T3 -> T4    :=
   T1 -> (T2 -> (T3 -> T4))
```

(der Typ-Pfeil ist rechts-assoziativ)

Semantik mit Closures

- bisher: `ValFun` ist Funktion als Datum der Gastsprache

```
value env x = case x of ...
  Abs n b -> ValFun $ \ v ->
    value (extend n v env) b
  App f a ->
    with_fun ( value env f ) $ \ g ->
    with_val ( value env a ) $ \ v -> g v
```

- alternativ: Closure: enthält Umgebung `env` und Code `b`

```
value env x = case x of ...
  Abs n b -> ValClos env n b
  App f a -> ...
```

Closures (Spezifikation)

- Closure konstruieren (Axiom-Schema):

$$\frac{}{\text{wert}(E, \lambda n.b, \text{Clos}(E, n, b))}$$

- Closure benutzen (Regel-Schema, 3 Prämissen)

$$\frac{\begin{array}{c}\text{wert}(E_1, f, \text{Clos}(E_2, n, b)), \\ \text{wert}(E_1, a, w), \text{wert}(E_2[n := w], b, r)\end{array}}{\text{wert}(E_1, fa, r)}$$

- Ü: Inferenz-Baum für Auswertung des vorigen Testfalls (geschachtelte Let) zeichnen
- . . . oder Interpreter so erweitern, daß dieser Baum ausgegeben wird

Rekursion?

- Das geht nicht, und soll auch nicht gehen:

```
let { x = 1 + x } in x
```

- aber das hätten wir doch gern:

```
let { f = \ x -> if x > 0
                 then x * f (x -1) else 1
    } in  f 5
```

  (nächste Woche)

- aber auch mit nicht rekursiven Funktionen kann man interessante Programme schreiben:

Testfall (2)

```
let { t f x = f (f x) }
in  let { s x = x + 1 }
    in  t t t t s 0
```

- auf dem Papier den Wert bestimmen
- mit selbstgebautem Interpreter ausrechnen
- mit Haskell ausrechnen
- in JS (node) ausrechnen

Übungen

1. Schreiben und benutzen Sie die Hilfsfunktion `with_val`, die im Skript an einigen Stellen schon benutzt wurde. Realisieren Sie damit die strikte Semantik für Let-Bindung und Funktions-Anwendung.
2. alternative Implementierung von Umgebungen
   - bisher `type Env = String -> Val`
   - jetzt `type Env = Data.Map.Map String Val`
3. alternative Implementierung von Namen
   der Vergleich von Strings (als Schlüssel im Suchbaum) ist teuer. Ist für diesen Interpreter zeitkritisch!
   - statt `data Name = Name T.Text` besser
     ```
     data Name =
       Name { hash :: Int, form :: T.Text }
       deriving (Eq, Ord)
     ```
   - mit *smart constructor* `name :: String -> Name`
     dann `Times (Ref (name "x")) (Const 3)`
   - und `instance IsString Name where ...`
     dann (wie früher!) `Times (Ref "x") (Const 3)`
   - für später: der Compiler benutzt `Name` und `Env` nur zur Übersetzungszeit!

Motivation

1. Modellierung von Funktionen:
   - intensional: Fkt. ist Berechnungsvorschrift, Programm
   - (extensional: Fkt. ist Menge v. geordneten Paaren)
2. Notation mit gebundenen (lokalen) Variablen, wie in
   - Analysis: $\int x^2 \,\mathrm{d}\, x, \sum_{k=0}^{n} k^2$
   - Logik: $\forall x \in A : \forall y \in B : P(x, y)$
   - Programmierung: `static int foo (int x) { ... }`

Der Lambda-Kalkül

- ist der Kalkül für Funktionen mit benannten Variablen
- Alonzo Church, 1936 . . . Henk Barendregt, 1984 . . .
- die wesentliche Operation ist das Anwenden einer Funktion:

$$(\lambda x.B)A \rightarrow_\beta B[x := A]$$

Beispiel: $(\lambda x.x * x)(3 + 2) \rightarrow_\beta (3 + 2) * (3 + 2)$

- Im reinen Lambda-Kalkül gibt es *nur* Funktionen (keine Zahlen, Wahrheitswerte usw.)

Lambda-Terme

- Menge Λ der Lambda-Terme
  (mit Variablen aus einer Menge V):
  - (Variable) wenn $x \in V$, dann $x \in \Lambda$
  - (Applikation) wenn $F \in \Lambda, A \in \Lambda$, dann $(FA) \in \Lambda$
  - (Abstraktion) wenn $x \in V, B \in \Lambda$, dann $(\lambda x.B) \in \Lambda$

  Beispiele: $x, (\lambda x.x), ((xz)(yz)), (\lambda x.(\lambda y.(\lambda z.((xz)(yz)))))$
- verkürzte Notation (Klammern weglassen)
  - $(\ldots((FA_1)A_2)\ldots A_n) \sim FA_1A_2\ldots A_n$
  - $\lambda x_1.(\lambda x_2.\ldots.(\lambda x_n.B)\ldots) \sim \lambda x_1 x_2\ldots x_n.B$

  mit diesen Abkürzungen simuliert $(\lambda x_1\ldots x_n.B)A_1\ldots A_n$
  eine mehrstellige Funktion und -Anwendung

Eigenschaften der Reduktion

- $\rightarrow_\beta$ auf Λ ist nicht terminierend (es gibt Terme mit unendlichen Ableitungen)
  $W = \lambda x.xx, \Omega = WW$.
- es gibt Terme mit Normalform und unendlichen Ableitungen, $KI\Omega$ mit $K = \lambda xy.x, I = \lambda x.x$
- $\rightarrow_\beta$ auf Λ ist konfluent
  $\forall A, B, C \in \Lambda : A \rightarrow_\beta^* B \wedge A \rightarrow_\beta^* C \Rightarrow \exists D \in \Lambda : B \rightarrow_\beta^* D \wedge C \rightarrow_\beta^* D$
- Folgerung: jeder Term hat höchstens eine Normalform

Beziehung zur Semantik des Interpreters

- die Relation $\rightarrow_\beta$ ist eine *small-step-Semantik*
  wert(E, X, v) ist *big-step-Semantik* für Interpreter
- Diese „passen zusammen.“ — Formal (Ansatz):
  $\forall X, Y \in \text{Exp}(\text{Abs}, \text{App}, <= * : X \rightarrow_\beta^* Y \stackrel{?}{\Leftrightarrow} \text{wert}(\emptyset, X, Y)$
- ist falsch (sinnfrei): Typen von Y passen nicht! Besser:
  $\forall X \in$
  $\text{Exp}(\text{Abs}, \text{App}, <= *\text{ConstBool}), B \in \{\text{False}, \text{True}\} : X \rightarrow_\beta^* \text{ConstB}$
- gilt nur für bestimmte Reduktionsstrategie (nicht für $\rightarrow_\beta^*$)
- ist für induktiven Beweis nicht geeignet
  (nicht alle X sind geschlossen, nicht alle E sind $\emptyset$)

Daten als Funktionen

- Simulation von Daten (Tupel)
  durch Funktionen (Lambda-Ausdrücke):
  - Konstruktor: $\langle D_1, \ldots, D_k \rangle := \lambda s.sD_1 \ldots D_k$
  - Selektoren: $s_i^k := \lambda t.t(\lambda d_1 \ldots d_k.d_i)$

  es gilt $s_i^k \langle D_1, \ldots, D_k \rangle \rightarrow_\beta^* D_i$

  Ü: überprüfen für $k = 2$
- Anwendungen:
  - Modellierung von Listen, Zahlen
  - Auflösung simultaner Rekursion

Lambda-Kalkül als universelles Modell

- Wahrheitswerte:
  True $:= \lambda xy.x$, False $:= \lambda xy.y$
- Verzweigung: `if` b `then` x `else` $y := bxy$
- natürliche Zahlen als iterierte Paare (Ansatz)
  $(0) := \langle \text{True}, \lambda x.x \rangle$; $(n+1) := \langle \text{False}, n \rangle$
- s_2^2 ist partielle Vorgänger-Funktion: $s_2^2(n+1) = n$
- Verzweigung: `if` $a = 0$ `then` x `else` $y := s_1^2 axy$
- Ü: nachrechnen. Ü: das geht sogar mit $(0) = \lambda x.x$
- Rekursion?

Fixpunkt-Kombinatoren (Motivation)

- Beispiel: die Fakultät

  ```
  f = \ x -> if x=0 then 1 else x*f(x-1)
  ```

  erhalten wir als Fixpunkt einer Fkt. 2. Ordnung

  ```
  g = \ h x -> if x=0 then 1 else x * h(x-1)
  f = fix g   -- d.h., f = g f
  ```

- Ü: $g(\lambda z.z)7$, Ü: `fix` g 7
- Implementierung von `fix` mit Rekursion:

  ```
  fix g = g (fix g)
  ```

- es geht aber auch *ohne Rekursion*. Ansatz: `fix` $= AA$,
  dann `fix` $g = AAg = g(AAg) = g($`fix` $g)$
  eine Lösung ist $A = \lambda xy\ldots$.

Fixpunkt-Kombinatoren (Implementierung)

- Definition (der *Fixpunkt-Kombinator* von Turing)
  $\Theta = (\lambda xy.(y(xxy)))(\lambda xy.(y(xxy)))$
- Satz: $\Theta f \rightarrow_\beta^* f(\Theta f)$, d. h. Θf ist Fixpunkt von f
- Folgerung: im Lambda-Kalkül kann man simulieren:
  - Daten: Zahlen, Tupel von Zahlen
  - Programmablaufsteuerung durch:
    - Nacheinander„ausführung“:
      Verkettung von Funktionen
    - Verzweigung,
    - Wiederholung: durch Rekursion (mit Fixpunktkomb.)

Lambda-Berechenbarkeit

Satz: (Church, Turing)

Menge der Turing-berechenbaren Funktionen

(Zahlen als Wörter auf Band)

Alan Turing: On Computable Numbers, with an Application to the Entscheidungsproblem, Proc. LMS, 2 (1937) 42 (1) 230–265
`https://dx.doi.org/10.1112/plms/s2-42.1.230`

$=$ Menge der Lambda-berechenbaren Funktionen

(Zahlen als Lambda-Ausdrücke)

Alonzo Church: A Note on the Entscheidungsproblem, J. Symbolic Logic 1 (1936) 1, 40–41

$=$ Menge der while-berechenbaren Funktionen

(Zahlen als Registerinhalte)

Kodierung von Zahlen nach Church

- $c : \mathbb{N} \to \Lambda : n \mapsto \lambda fx.f^n(x)$
  mit $f^0(x) := x, f^{n+1}(x) := f(f^n(x))$
- in Haskell: `c n f x = iterate f x !! n`
- Decodierung: `d e = e (\x -> x+1) 0`
- Nachfolger: $s(c_n) = c_{n+1}$ für $s = \lambda nfx.f(nfx)$
  1. auf Papier beweisen, 2. mit leancheck prüfen
  benutze `check $ \ (Natural x) -> ...`
- Addition: `plus` c_a $c_b = c_{a+b}$ für `plus` $= \lambda abfx.af(bfx)$
- implementiere die Multiplikation, beweise, prüfe
- Potenz: `pow` c_a $c_b = c_{a^b}$ für `pow` $= \lambda ab.ba$

Übung Lambda-Kalkül (I)

- die Fakultät (z.B. von 7) ...
  - in Haskell (ohne Rekursion, aber mit `Data.Function.fix`)
  - in unserem Interpreter (ohne Rekursion, mit Turing-Fixpunktkombinator Θ)
  - in Javascript (ohne Rekursion, mit Θ)
- Kodierung von Wahrheitswerten und Zahlen (nach Church)
  - implementiere Test auf 0: `iszero` c_n = if $n = 0$ then True else False
  - implementiere Addition, Multiplikation, Fakultät ohne `If`, `Eq`, `Const`, `Plus`, `Times`
  - für nützliche Ausgaben: das Resultat nach `ValInt` dekodieren (dabei muß `Plus` und `Const` benutzt werden)

Übung Lambda-Kalkül (II)

folgende Aufgaben aus H. Barendregt: *Lambda Calculus*

- (Abschn. 6.1.5) gesucht wird F mit $Fxy = FyxF$. Musterlösung: es gilt $F = \lambda xy.FyxF = (\lambda fxy.fyxf)F$, also $F = \Theta(\lambda fxy.fyxf)$
- (Aufg. 6.8.2) Konstruiere $K^\infty \in \Lambda^0$ (ohne freie Variablen) mit $K^\infty x = K^\infty$ (hier und in im folgenden hat $=$ die Bedeutung $(\rightarrow_\beta \cup \rightarrow_\beta^-)^*$
- Konstruiere $A \in \Lambda^0$ mit $Ax = xA$
- beweise den Doppelfixpunktsatz (Kap. 6.5) $\forall F, G : \exists A, B : A = FAB \wedge B = GAB$
- (Aufg. 6.8.17, B. Friedman) Konstruiere Null, Nulltest, partielle Vorgängerfunktion für Zahlensystem mit Nachfolgerfunktion $s = \lambda x.\langle x \rangle$ (das 1-Tupel)
- (Aufg. 6.8.14, J. W. Klop)

$$X = \lambda abcdefghijklmnopqstuvvxyzr.$$
$$r(thisisafixedpointcombinator)$$
$$Y = X^{27} = \underbrace{X \ldots X}$$

Motivation

Das ging bisher gar nicht:

```
let { f = \ x -> if x > 0
                  then x * f (x -1) else 1
    } in  f 5
```

Lösung 1: benutze Fixpunktkombinator

```
let { Theta =  ... } in
let { f = Theta ( \ g -> \ x -> if x > 0
                  then x * g (x - 1) else 1 )
    } in f 5
```

Lösung 2 (später): realisiere Fixpunktberechnung im Interpreter (neuer AST-Knotentyp)

Existenz von Fixpunkten

Fixpunkt von $f :: C \to C$ ist $x :: C$ mit $fx = x$.
Existenz? Eindeutigkeit? Konstruktion?
Satz: Wenn *C pointed CPO* und *f stetig*,
dann besitzt *f* genau einen kleinsten Fixpunkt.

- CPO = complete partial order = vollständige Halbordnung
- complete = jede monotone Folge besitzt Supremum (= kleinste obere Schranke)
- pointed: *C* hat kleinstes Element $\bot$

Beispiele f. Halbordnungen, CPOs

Halbordnung? pointed? complete?

- $\leq$ auf $\mathbb{N}$
- $\leq$ auf $\mathbb{N} \cup \{+\infty\}$
- $\leq$ auf $\{x \mid x \in \mathbb{R}, 0 \leq x \leq 1\}$
- $\leq$ auf $\{x \mid x \in \mathbb{Q}, 0 \leq x \leq 1\}$
- Teilbarkeit auf $\mathbb{N}$
- Präfix-Relation auf Σ^*
- $\{((x_1, y_1), (x_2, y_2)) \mid (x_1 \leq x_2) \vee (y_1 \leq y_2)\}$ auf $\mathbb{R}^2$
- $\{((x_1, y_1), (x_2, y_2)) \mid (x_1 \leq x_2) \wedge (y_1 \leq y_2)\}$ auf $\mathbb{R}^2$
- Relation $\subseteq$ auf $\{\{A\}, \{B\}, \{A, B\}\}$
- identische Relation id_M auf einer beliebigen Menge M
- $\{(\bot, x) \mid x \in M_\bot\} \cup \mathrm{id}_M$ auf $M_\bot := \{\bot\} \cup M$

Stetige Funktionen

f ist stetig :=

- f ist monoton: $x \leq y \Rightarrow f(x) \leq f(y)$
- und für monotone Folgen $[x_0, x_1, \ldots]$ gilt: $f(\sup[x_0, x_1, \ldots]) = \sup[f(x_0), f(x_1), \ldots]$

Beispiele: in $(\mathbb{N} \cup \{+\infty\}, \leq)$

- $x \mapsto 42$ ist stetig
- $x \mapsto$ if $x < +\infty$ then $x + 1$ else $+\infty$
- $x \mapsto$ if $x < +\infty$ then 42 else $+\infty$

Satz: Wenn *C pointed CPO* und $f : C \to C$ *stetig*, dann besitzt f genau einen kleinsten Fixpunkt . . .
. . . und dieser ist $\sup[\bot, f(\bot), f^2(\bot), \ldots]$

Funktionen als CPO

- Menge der partiellen Funktionen von B nach B: $C = (B \hookrightarrow B)$
- partielle Funktion $f : B \hookrightarrow B$ entspricht totaler Funktion $f : B \to B_\perp$
- C geordnet durch $f \leq g \iff \forall x \in B : f(x) \leq g(x)$, wobei $\leq$ die vorhin definierte CPO auf $B_\perp$
- $f \leq g$ bedeutet: g ist Verfeinerung von f
- Das Bottom-Element von C ist die überall undefinierte Funktion. (diese heißt auch $\perp$)

Funktionen als CPO, Beispiel

- der Operator $F =$

```
\ g -> ( \ x -> if (x==0) then 0
                else 2 + g (x - 1) )
```

  ist stetig auf $(\mathbb{N} \hookrightarrow \mathbb{N})$ (Beispiele nachrechnen!)

- Iterative Berechnung des Fixpunktes:

$$\begin{aligned} \bot &= \emptyset \quad \text{überall undefiniert} \\ F\bot &= \{(0,0)\} \\ F(F\bot) &= \{(0,0),(1,2)\} \\ F^3\bot &= \{(0,0),(1,2),(2,4)\} \end{aligned}$$

- $\sup[\ldots, F^k\bot, \ldots] = \{(x, 2x) \mid x \in \mathbb{N}\}$

Welche Funktionale sind stetig?

- die Erfahrung (der Programmierung mit rekursiven Funktionen) lehrt: alle Operatoren `\g -> \x -> .. g ..` sind stetig.
- denn die Semantik der (von uns bisher benutzten) AST-Konstruktoren bildet stetige Ftk. auf stetige Fkt. ab
- Ü: $(\lambda x.\, \text{if } x = \bot \text{ then } 42 \text{ else } \bot)$ ist nicht stetig.
- Diskussion: eine solche Funktion kann nicht die Semantik eines Programms sein, weil das Halteproblem nicht entscheidbar ist

Fixpunktberechnung im Interpreter

- Erweiterung der abstrakten Syntax:

```
data Exp = ... | Rec Name Exp
```

- Beispiel

```
App
  (Rec g (Abs v (if v==0 then 0 else 2 + g(v-1))))
  5
```

- Bedeutung: `Rec x B` ist Fixpunkt von $(\lambda x.B)$
- Semantik: $\frac{\text{wert}(E, (\lambda x.B)(\text{Rec } x\ B), v)}{\text{wert}(E, \text{Rec } x\ B, v)}$
- Ü: verwende `Let` statt `App (Abs ..) ..`
- Ü: das Beispiel mit dieser Regel auswerten

Fixpunkte und Laziness

Fixpunkte existieren in pointed CPOs.

- Zahlen: nicht pointed
  (arithmetische Operatoren sind strikt)
- Funktionen: partiell $\Rightarrow$ pointed
  ($\bot$ ist überall undefinierte Funktion)
- Daten (Listen, Bäume usw.): pointed:
  (Konstruktoren sind nicht strikt)

Beispiele in Haskell:

```
fix f = f (fix f)
xs = fix $ \ zs -> 1 : zs
ys = fix $ \ zs ->
     0 : 1 : zipWith (+) zs (tail zs)
```

Arithmetik mit Bedarfsauswertung

- über $\mathbb{Q}$ hat $f = \lambda x.1 + x/4$ einen Fixpunkt (4/3), aber $\sup_k f^k \bot = \bot$, weil die Operationen strikt sind.
- wirklich? Kommt auf die Repräsentation der Zahlen an! Op. auf Maschinenzahlen sind strikt. — Aber:
- Zahl als lazy Liste von Ziffern (Bsp: Basis 2)

```
x=plus(1:repeat 0)(0:0:x)=[1,0,1,0,1,0..]
```

- Ü: bestimme $y = \sqrt{2} - 1$ aus $2 = (1 + y)^2$, d.h., als Fixpunkt von $\lambda y.(1 - y^2)/2$
- Kombiniere `https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.4249` (Jerzy Karczmarczuk 1998), `http://joerg.endrullis.de/publications/productivity/`

Simultane Rekursion: letrec

- Beispiel (aus: D. Hofstadter, Gödel Escher Bach, 1979)

```
letrec { f = \ x -> if x == 0 then 1
                    else x - g(f(x-1))
       , g = \ x -> if x == 0 then 0
                    else x - f(g(x-1))
} in f 15
```

  Bastelaufgabe: für welche x gilt $f(x) \neq g(x)$?

- weitere Beispiele:

```
letrec { y = x * x, x = 3 + 4  } in x - y
letrec { f = \ x -> .. f (x-1) } in f 3
```

letrec nach rec (falsch)

- Plan: mit Lambda-Ausdrücken für Konstruktor, Selektor

```
LetRec [(n1,x1), .. (nk,xk)] y
=> ( rec t ( let n1 = select1 t
                ...
                nk = selectk t
             in  tuple x1 .. xk ) )
   ( \ n1 .. nk -> y )
```

- benutzt $\langle x_1, \ldots, x_k \rangle f = f x_1 \ldots x_k$
- terminiert nicht, die Auswertungsstrategie des Interpreters ist dafür zu eifrig (*eager*)
- Lösung: `tuple` direkt unter `rec t`,
  `let .. tuple ..` $\Rightarrow$ `tuple (let ..) (let ..)`

letrec nach rec (richtig)

- Teilausdrücke (für jedes *i*)

  ```
  let { n1 = select1 t, ..  nk = selectk t
      } in xi
  ```

  äquivalent vereinfachen zu `t (\ n1 .. nk -> xi)`

- ```
  LetRec [(n1,x1), .. (nk,xk)] y
  => ( rec t
         ( tuple ( t ( \ n1 .. nk -> x1 ))
                 ...
                 ( t ( \ n1 .. nk -> xk )) ))
     ( \ n1 .. nk -> y )
  ```

- Ü: implementiere `letrec {f = _, g = _} in f 15`
  ```

Übung Fixpunkte

1. Limes der Folge $F^k(\bot)$ für
   ```
   F h = \ x -> if x > 23 then x - 11
                 else h (h (x + 14))
   ```
2. Ist F stetig? Gib den kleinsten Fixpunkt von F an:
   ```
   F h =  \ x -> if x >= 2 then 1 + h(x-2)
       else if x == 1 then 1 else h(4) - 2
   ```
   Hat F weitere Fixpunkte?
3. $C =$ Menge der Formalen Sprachen über $\Sigma = \{a, b\}$, halbgeordnet durch $\subseteq$. ist CPO? pointed?
   $g : C \to C : L \mapsto \{\epsilon\} \cup \{a\} \cdot L \cdot \{b\}$ ist stetig? Fixpunkt(e)?
   $h : C \to C : L \mapsto \{a\} \cup L \cdot \{b\} \cdot (\Sigma^* \setminus L)$
4. in der Relation $\subseteq$ auf $\{\{A\}, \{B\}, \{A, B\}\}$: geben Sie eine stetige Funktion an, die zwei verschiedene kleinste Fixpunkte besitzt.
5. Geben Sie Argumente aus dieser Diskussion wieder: *. . . distinguish bindings that are self-referentially recursive from non-recursive bindings* `https://github.com/ghc-proposals/ghc-proposals/pull/401` (O.

Haskell-Software-Pakete

- Paket (package) := Quelltexte + Beschreibung von
  - herstellbaren Artefakten
    (Bibliotheken, Executables, Testfälle)
  - benötigten Paketen (und Versionen-Constraints)

  Format der Beschreibung: Textdatei `foo.cabal`
- Paket-Sammlung: `https://hackage.haskell.org/`
- Build-Werkzeug: `cabal install|test|repl`
- Installation in `$HOME/.cabal/{bin,lib,*}`,
  ist projekt-übergreifender Cache

Kuratierte Paket-Mengen

- Versions-Constraints in Cabal-Files können streng bis großzügig sein (oder ganz fehlen),
- gebaut wird mit den letzten passenden Hackage-Versionen, kann zu nicht reproduzierbaren Builds führen (oder schlimmer, „cabal hell“)
- `https://stackage.org/` definiert *Resolver* (Bsp. LTS-14.14), das sind exakte Versionsnummern von konsistenten Paketmengen.
- Datei `stack.yaml` enthält Resolver und ggf. weitere Paketquellen (z.B. git-clone-URL)
- siehe auch `https://www.imn.htwk-leipzig.de/~waldmann/etc/untutorial/haskell/build/`

Continuous Intergration

- lokal:
  - im Editor: mit HIE `https://github.com/haskell/haskell-ide-engine`, benutzt Language Server Protocol `https://microsoft.github.io//language-server-protocol/specifications/specification-3-14/`
  - CLI: `ghcid`, siehe `https://github.com/ndmitchell/ghcid`
- remote: Beispiel siehe `https://gitlab.imn.htwk-leipzig.de/waldmann/haskell-ci-test`

Haskell-Module

- Modul: benannte Menge von Definitionen (Werte, Typen)

```
module M (M (), foo) where
import C (f); import qualified D ;
data M = A | B
foo :: M -> Bool ; foo x = ... f ..
bar :: Int -> M ; bar y = ... D.g ..
```

- Ziele:
  - Abstraktion (z.B. Konstruktoren von `M` nicht exportiert)
    damit kann man Invarianten (hier: von `M`) erzwingen
  - getrennte Kompilation,
    aus `M.hs` entstehen `M.o` und `M.hi`

Modul-Abhängigkeiten

- Abhängigkeitsgraph G: Knoten $=$ Module,
  Kante $C \to M$, falls `module M where import C`
- Kompilation verwendet topologische Ordnung auf G
  (Kompilation von `M` benötigt `C.hi`) $\Rightarrow$ G kreisfrei
- Kreise entfernen: `Env = Map Name Val`,
  `Val = .. | ValClos Name Exp Env`
  durch Typargumente `Env k v = Map k v`
  Ü: Beziehung zw. `Val` und `Action`?
- Kreiskante $C \to M$ entfernen: File `M.hs-boot`
  Typ- und data-Deklarationen *ohne* Konstruktoren
  `module C where import {-# SOURCE #-} M`
  `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/separate_compilation.html#how-to-compile-mutually-recursive-modules`

Motivation

bisherige Programme sind nebenwirkungsfrei, das ist nicht immer erwünscht:

- direktes Rechnen auf von-Neumann-Maschine: Änderungen im Hauptspeicher
- direkte Modellierung von Prozessen mit Zustandsänderungen ((endl.) Automaten)

Dazu muß semantischer Bereich geändert werden.

- bisher: `Val`, jetzt: `State -> (State, Val)` (dabei ist `(A,B)` die Notation für $A \times B$)

Semantik von (Teil-)Programmen ist Zustandsänderung.

Speicher (Daten)

- Implementierung benutzt größenbalancierte Suchbäume
  `http://hackage.haskell.org/package/containers/docs/Data-Map.html`
- Notation mit qualifizierten Namen:

```
import qualified Data.Map as M
newtype Addr = Addr Int
type Store = M.Map Addr Val
```

- `newtype`: wie `data` mit genau einem Konstruktor, Konstruktor wird zur Laufzeit *nicht* repräsentiert
- Aktion: liefert neue Speichernbelegung und Resultat

```
newtype Action a =
    Action ( Store -> ( Store, a ))
```

Speicher (Operationen)

- `newtype Action a = Action ( Store -> ( Store, a ))`
- spezifische Aktionen:

```
new :: Val -> Action Addr
get :: Addr -> Action Val
put :: Addr -> Val -> Action ()
```

- Aktion ausführen, Resultat liefern (Zielzustand ignorieren)

```
run :: Store -> Action a -> a
```

- Aktionen nacheinander ausführen

```
bind :: Action a ->  ... Action b -> Action b
```

Aktion ohne Zustandsänderung

```
result :: a -> Action a
```

Auswertung von Ausdrücken

- Ausdrücke (mit Nebenwirkungen):

```
date Exp = ...
    | New Exp | Get Exp | Put Exp Exp
```

- Resultattyp des Interpreters ändern:

```
value     :: Env -> Exp ->          Val
          :: Env -> Exp -> Action Val
```

- semantischen Bereich erweitern:

```
data Val = ... | ValAddr Addr
```

- Aufruf des Interpreters:

```
run Store.empty $ value env0 $ ...
```

Änderung der Hilfsfunktionen

- bisher:

```
with_int :: Val -> ( Int -> Val ) -> Val
with_int v k = case v of
    ValInt i -> k i
```

- jetzt:

```
with_int :: Action Val
  -> ( Int -> Action Val ) -> Action Val
with_int a k = bind a $ \ v -> case v of
    ValInt i -> k i
```

- Hauptprogramm muß kaum geändert werden (!)

Variablen?

in unserem Modell haben wir:

- veränderliche Speicherstellen,
- aber immer noch unveränderliche „Variablen“ (lokale Namen)

$\Rightarrow$ der Wert eines Namens kann eine Speicherstelle sein, aber dann immer dieselbe.

Imperative Programmierung

es fehlen noch wesentliche Operatoren:

- Nacheinanderausführung (Sequenz)
- Wiederholung (Schleife)

diese kann man:

- simulieren (durch `let`)
- als neue AST-Knoten realisieren (Übung)

Rekursion

mehrere Möglichkeiten zur Realisierung

- im Lambda-Kalkül (in der interpretierten Sprache)
  mit Fixpunkt-Kombinator
- durch Rekursion in der Gastsprache des Interpreters
- simulieren (in der interpretierten Sprache)
  durch Benutzung des Speichers

Rekursion (operational)

- Idee: eine Speicherstelle anlegen und als Vorwärtsreferenz auf das Resultat der Rekursion benutzen
- ```
  Rec n (Abs x b)  ==>
      a := new 42
      put a ( \ x -> let { n = get a } in b )
      get a
  ```
  ```

Speicher—Übung

Fakultät imperativ:

```
let { fak = \ n ->
        { a := new 1 ;
          while ( n > 0 )
            { a := a * n ; n := n - 1; }
          return a;
        }
    } in  fak 5
```

1. Schleife durch Rekursion ersetzen und Sequenz durch `let`:

   ```
   fak = let { a = new 1 }
         in  Rec f ( \ n -> ... )
   ```

2. Syntaxbaumtyp erweitern um Knoten für Sequenz und Schleife

. . . unter verschiedenen Aspekten

- unsere Motivation: semantischer Bereich, `result :: a -> m a` als wirkungslose Aktion, Operator `bind :: m a -> (a -> m b) -> m b` zum Verknüpfen von Aktionen
- auch nützlich: `do`-Notation (anstatt Ketten von `>>=`)
- die Wahrheit: *a monad in X is just a monoid in the category of endofunctors of X*
- die ganze Wahrheit:
  `Functor m => Applicative m => Monad m`
- weitere Anwendungen: `IO`, Parser-Kombinatoren, weitere semant. Bereiche (Continuations, Typisierung)

Die Konstruktorklasse Monad

- Definition (in Standardbibliothek)

```
class Monad m where
    return  :: a -> m a
    ( >>= ) :: m a -> (a -> m b) -> m b
```

- Instanz (für benutzerdefinierten Typ)

```
instance Monad Action where
  return = result ; (>>=) = bind
```

- Benutzung der Methoden:

```
value e l >>= \ a ->
value e r >>= \ b ->
return ( a + b )
```

Do-Notation für Monaden

```
value e l >>= \ a ->
value e r >>= \ b ->
return ( a + b )
```

do-Notation (explizit geklammert):

```
do { a <- value e l
   ; b <- value e r
   ; return ( a + b )
   }
```

do-Notation (implizit geklammert):

```
do a <- value e l
   b <- value e r
   return ( a + b )
```

Haskell: implizite Klammerung nach `let`, `do`, `case`, `where`

Beispiele für Monaden

- Aktionen mit Speicheränderung (vorige Woche)

  ```
  Action (Store -> (Store, a))
  ```

- Aktionen mit Welt-Änderung: `IO a`
- Transaktionen (Software Transactional Memory) `STM a`
- Vorhandensein oder Fehlen eines Wertes

  ```
  data Maybe a = Nothing | Just a
  ```

- . . . mit Fehlermeldung

  ```
  data Either e a = Left e | Right a
  ```

- Nichtdeterminismus (eine Liste von Resultaten): `[a]`
- Parser-Monade (nächste Woche)

Die IO-Monade

```
data IO a -- abstract
instance Monad IO -- eingebaut

readFile :: FilePath -> IO String
putStrLn :: String -> IO ()
```

Alle „Funktionen“, deren Resultat von der Außenwelt (Systemzustand) abhängt, haben Resultattyp `IO ...`, sie sind tatsächlich *Aktionen*.
Am Typ einer Funktion erkennt man ihre möglichen Wirkungen bzw. deren garantierte Abwesenheit.

```
main :: IO ()
main = do
   cs <- readFile "foo.bar" ; putStrLn cs
```

Grundlagen: Kategorien

- *Kategorie C* hat Objekte Obj_C und Morphismen Mor_C, jeder Morphismus m hat als Start (S) und Ziel (T) ein Objekt, Schreibweise: $m : S \to T$ oder $m \in \mathrm{Mor}_C(S, T)$
- für jedes $O \in \mathrm{Obj}_C$ gibt es $\mathrm{id}_O : O \to O$
- für $f : S \to M$ und $g : M \to T$ gibt es $f \circ g : S \to T$.
  es gilt immer $f \circ \mathrm{id} = f$, $\mathrm{id} \circ g = g$, $f \circ (g \circ h) = (f \circ g) \circ h$

Beispiele:

- Set: $\mathrm{Obj}_{\mathrm{Set}}$ = Mengen, $\mathrm{Mor}_{\mathrm{Set}}$ = totale Funktionen
- Grp: $\mathrm{Obj}_{\mathrm{Grp}}$ = Gruppen, $\mathrm{Mor}_{\mathrm{Grp}}$ = Homomorphismen
- für jede Halbordnung $(M, \leq)$: $\mathrm{Obj} = M, \mathrm{Mor} = (\leq)$
- Hask: $\mathrm{Obj}_{\mathrm{Hask}}$ = Typen, $\mathrm{Mor}_{\mathrm{Hask}}$ = Funktionen

Kategorische Definitionen und Sätze

Beispiel: Isomorphie

- eigentlich: Abbildung, die die Struktur (der abgebildeten Objekte) erhält
- Struktur von $O \in \mathrm{Obj}(C)$ ist aber unsichtbar
- Eigenschaften von Objekten werden beschrieben durch Eigenschaften ihrer Morphismen (vgl. abstrakter Datentyp, API)
  Bsp: $f : A \to B$ ist *Isomorphie* (kurz: ist iso), falls es ein $g : B \to A$ gibt mit $f \circ g = \mathrm{id}_A \wedge g \circ f = \mathrm{id}_B$

weiteres Beispiel

- Def: $m : a \to b$ monomorph: $\forall f, g : f \circ m = g \circ m \Rightarrow f = g$
- Satz: f mono $\wedge$ g mono $\Rightarrow$ $f \circ g$ mono.

Produkte

- Def: P ist *Produkt* von A_1 und A_2
  mit Projektionen $\text{proj}_1 : P \to A_1, \text{proj}_2 : P \to A_2$,
  wenn für jedes B und Morphismen $f_1 : B \to A_1, f_2 : B \to A_2$
  es existiert genau ein $g : B \to P$
  mit $g \circ \text{proj}_1 = f_1$ und $g \circ \text{proj}_2 = f_2$
- für Set ist das wirklich das Kreuzprodukt
- für die Kategorie einer Halbordnung?
- für Gruppen? (Beispiel?)

Dualität

- Wenn ein Begriff kategorisch definiert ist,
  erhält man den dazu *dualen* Begriff
  durch Spiegeln aller Pfeile
- Bsp: dualer Begriff zu *Produkt* (heißt *Summe*)
  Definition hinschreiben, Beispiele angeben
- Ü: dualer Begriff zu: mono (1. allgemein, 2. Mengen)
- entsprechend: die *duale Aussage*
  diese gilt gdw. die originale (primale) Aussage gilt
- Ü: duale Aussage für Komposition von mono.

Funktoren

- Def: Funktor F von Kategorie C nach Kategorie D:
  - Wirkung auf Objekte: $F_{\text{Obj}} : \text{Obj}(C) \to \text{Obj}(D)$
  - Wirkung auf Pfeile:
    $F_{\text{Mor}} : (g : s \to t) \mapsto (g' : F_{\text{Obj}}(S) \to F_{\text{Obj}}(T))$

  mit den Eigenschaften:
  - $F_{\text{Mor}}(\text{id}_o) = \text{id}_{F_{\text{Obj}}(o)}$
  - $F_{\text{Mor}}(g \circ_C h) = F_{\text{Mor}}(g) \circ_D F_{\text{Mor}}(h)$
- Bsp: Funktoren zw. Kategorien von Halbordnungnen?
- `class Functor f where`

  `fmap :: (a -> b) -> (f a -> f b)`

  Beispiele: `List, Maybe, Action`

Die Kleisli-Konstruktion

- Plan: für Kategorie C, Endo-Funktor $F : C \to C$
  definiere sinnvolle Struktur auf Pfeilen $s \to Ft$
- Durchführung: die Kleisli-Kategorie K von F:
  $\mathrm{Obj}(K) = \mathrm{Obj}(C)$, Pfeile: $s \to Ft$
- ... K ist tatsächlich eine Kategorie, wenn:
  - identische Morphismen (`return`), Komposition (`>=>`)
  - mit passenden Eigenschaften

  (F, `return`, `(>=>)`) heißt dann *Monade*

Diese Komposition ist `f >=> g = \ x -> (f x >>= g)`
Aus o.g. Forderung (K ist Kategorie) ergibt sich Spezifikation für `return` und `>>=`

Functor, Applicative, Monad

https://wiki.haskell.org/Functor-Applicative-Monad_Proposal

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> (f a -> f b)
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
```

eine Motivation: effizienterer Code für `>>=`,
wenn das rechte Argument eine konstante Funktion ist
(d.h. die Folge-Aktion hängt nicht vom Resultat der ersten Aktion ab: dann ist Monad nicht nötig, es reicht Applicative)

Die Maybe-Monade

```
data Maybe a = Nothing | Just a
instance Monad Maybe where ...
```

Beispiel-Anwendung:

```
case ( evaluate e l ) of
    Nothing -> Nothing
    Just a  -> case ( evaluate e r ) of
        Nothing -> Nothing
        Just b  -> Just ( a + b )
```

mittels der Monad-Instanz von Maybe:

```
evaluate e l >>= \ a ->
    evaluate e r >>= \ b ->
        return ( a + b)
```

Ü: dasselbe mit do-Notation

List als Monade

```
instance Monad [] where
    return = \ x - > [x]
    m >>= f = case m of
        []     -> []
        x : xs -> f x ++ ( xs >>= f )
```

Beispiel:

```
do a <- [ 1 .. 4 ]
   b <- [ 2 .. 3 ]
   return ( a * b )
```

Anwendung: Ablaufsteuerung für Suchverfahren

Monaden: Zusammenfassung

- verwendet zur Definition semantischer Bereiche,
- Monade = Monoid über Endofunktoren in Hask,
  (Axiome für `return`, `>=>` bzw. `>>=`)
- Notation `do { x <- foo ; bar ; .. }`
  (`>>=` ist das benutzer-definierte Semikolon)
- Grundlagen: Kategorien-Theorie (ca. 1960),
  in Funktl. Prog. seit ca. 1990 `http://homepages.inf.ed.ac.uk/wadler/topics/monads.html`
- in anderen Sprachen: F#: *Workflows*, C#: LINQ-Syntax
- GHC ab 7.10: `Control.Applicative`: `pure` und `<*>`
  (= `return` und eingeschränktes `>>=`)

Monaden-Transformatoren

- Motivation: Kombination verschiedener Semantiken:
  - Zustands-Änderung (`Action`)
  - Fehlschlagen der Rechnung (`Maybe`)
  - Schritt-Zählen
  - Logging

  in *einer* Monade
- diese nicht selbst schreiben,
  sondern Monaden-Transformatoren verwenden, z.B.
  ```
  type Sem = ExceptT Err (StateT Store Identity)
  ```
- Standard-Bibliotheken dafür: transformers, mtl

Der Zustands-Transformator

- ```
  newtype StateT s m a
    = StateT { runStateT :: s -> m (a,s) }
  ```
- die elementaren Operationen:

  ```
  get :: Monad m => StateT s m s
  put :: Monad m => s -> StateT s m ()
  ```
- ```
  instance Monad m => Monad (StateT s m)
  ```
- Anwendung auf

  ```
  newtype Identity a = Identity { runIdentity :: a}
  ```

  ergibt ```type State s = StateT s Identity```

Der Ausnahme-Transformator MaybeT

- ```
  newtype MaybeT m a =
    MaybeT { runMaybeT :: m (Maybe a) }
  ```
- Anwendung im Interpreter:

  ```
  type Sem = StateT Store (MaybeT Identity)
  with_int :: Sem Val->(Val -> Sem r)->Sem r
  with_int a k = a >>= \ case
    ValInt i -> k i   ;  _ -> lift Nothing
  ```
- benutzt Einbettung der inneren Monade:

  ```
  class MonadTrans t where
    lift :: Monad m => m a -> t m a
  instance MonadTrans (StateT s)
  ```
  ```

Der Ausnahme-Transformator ExceptT

- wie MaybeT, aber mit Fehlermeldungen (Exceptions)

```
newtype ExceptT e m a = ExceptT (m (Either e a))
instance Monad m => Monad (ExceptT e m)
instance MonadTrans (ExceptT e)
```

- Operationen:

```
runExceptT :: ExceptT e m a -> m (Either e a)
throwE :: Monad m => e -> ExceptT e m a
catchE :: Monad m => ExceptT e m a -> (e -> ..) -> .
```

- Ü: verwende `Sem = StateT (ExceptT Err Id.)` mit sinnvollem `data Err = ..`
- Ü: Unterschied zu `ExceptT Err (StateT Id.)`

Bibliotheken für Monaden-Transformatoren

- gemeinsame Grundlage: Mark P Jones: *Functional Programming with Overloading and Higher-Order Polymorphism*, AFP 1995, `http://web.cecs.pdx.edu/~mpj/pubs/springschool.html` (S. 36 ff.)
- `https://hackage.haskell.org/package/transformers`: wie hier beschrieben
- `https://hackage.haskell.org/package/mtl`: zusätzliche Typklassen und Instanzen

```
class Monad m => MonadState s m | m -> s where
  get :: ... ; put :: ...
instance MonadState s m => MonadState s (MaybeT m)
```

dadurch braucht man fast keine `lift` zu schreiben

Übung

- der identische Morphismus jedes Objektes ist eindeutig
- was ist der duale Begriff zu Monomorphismus?
  Def., Bedeutung für Mengen
- Beweise: für das (kategorisch definierte) Produkt P von endlichen Mengen A_1, A_2 gilt: $|P| = |A_1 \times A_2|$.
  1. (Musterlösung) $|P| \geq |A_1 \times A_2|$: indirekter Beweis.
     Falls $|P| < |A_1 \times A_2|$, wähle $B = A_1 \times A_2$, $f_i(x_1, x_2) = x_i$.
     Dann ist g nicht injektiv: es gibt $x = (x_1, x_2) \neq (y_1, y_2) = y$, mit $g(x) = g(y)$.
     Für diese x, y gilt $x_i \neq y_i$ für wenigstens ein $i \in \{1, 2\}$.
     Aus $g \circ \mathrm{proj}_i = f_i$ folgt $x_i = f_i(x) = (g \circ \mathrm{proj}_i)(x) = \mathrm{proj}_i(g(x)) = \mathrm{proj}_i(g(y)) = (g \circ \mathrm{proj}_i)(y) = f_i(y) = y_i$, Widerspruch.
  2. (Aufgabe) $|P| \leq |A_1 \times A_2|$
- Kategorie mit gerichteten Graphen als Objekte,
  $f : \mathrm{V}(G) \to \mathrm{V}(H)$ ist Morphismus, falls
  $\forall x, y : (x, y) \in \mathrm{E}(G) \iff (f(x), f(y)) \in \mathrm{E}(H)$.
  Was bedeuten dort mono, epi?
  Produkt, Summe? Welche Graph-Operationen (aus VL Modellierung) haben die passenden Eigenschaften?

- Funktor- und Monadengesetze (mit leancheck in ghci)

```
cabal install --lib leancheck

import Test.LeanCheck
import Test.LeanCheck.Function
-- Kleisli-Komposition:
import Control.Monad ((>=>))

:set -XPatternSignatures

-- return ist rechts neutral
check $ \ (f:: Bool -> Maybe Bool) x -> (f x == (retu

-- ein Test, der zeigt, daß eine falsche Aussage erka
check $ \ (f:: Bool -> Maybe Bool) x -> (f x == (f >=
```

- falsche Functor-/Monad-Instanzen für Maybe, List, Tree (d.h. typkorrekt, aber semantisch falsch)
- `StateT Int` zum Zählen der Auswertungsschritte (Aufrufe von `value`) einbauen

```
import qualified Control.Monad.Trans.State as S
import qualified Control.Monad.Trans.Class as T
```

```
import Numeric.Natural

type Domain = S.StateT Natural Action Val

value :: Env Name Val -> Exp -> Domain
...
  Put e f -> with_addr (value env e) $ \ a ->
    with_value (value env f) $ \ v ->
    T.lift (put a v) >>= \ _ ->
    return ValUnit

test3 = print $ run state0 $ flip S.runStateT 0 $ val
   $ Let "a" (New $ LitInt 0) ...
```

- `WriterT [String]` zur Protokollierung (Argumente und Resultat von `value`)
- Lesen der Umgebung mit `ReaderT Env`. Änderung der Umgebung (in Let, App) durch welche Funktion?

Datentyp für Parser

- ```
  data Parser c a =
       Parser ( [c] -> [ (a, [c]) ] )
  ```
  - über Eingabestrom von Zeichen (Token) *c*,
  - mit Resultattyp *a*,
  - nichtdeterministisch (List).
- Beispiel-Parser, aufrufen mit:

```
parse :: Parser c a -> [c] -> [(a,[c])]
parse (Parser f) w = f w
```

- alternative Definition:

```
type Parser c a = StateT [c] [] a
```
```

Elementare Parser (I)

```
-- | das nächste Token
next :: Parser c c
next = Parser $ \ toks -> case toks of
    [] -> []
    ( t : ts ) -> [ ( t, ts ) ]
-- | das Ende des Tokenstroms
eof :: Parser c ()
eof = Parser $ \ toks -> case toks of
    [] -> [ ( (), [] ) ]
    _  -> []
-- | niemals erfolgreich
reject :: Parser c a
reject = Parser $ \ toks -> []
```

Monadisches Verketten von Parsern

Definition:

```
instance Monad ( Parser c ) where
  return x = Parser $ \ s -> return ( x, s )
  p >>= g = Parser $ \ s -> do
    ( a, t ) <- parse p s
    parse (g a) t
```

beachte: das *return/do* gehört zur List-Monade
Anwendungsbeispiel:

```
p :: Parser c (c,c)
p = do x <- next ; y <- next ; return (x,y)
```

mit Operatoren aus `Control.Applicative`:

```
p = (,) <$> next <*> next
```

Elementare Parser (II)

```
satisfy :: ( c -> Bool ) -> Parser c c
satisfy p = do
    x <- next
    if p x then return x else reject

expect :: Eq c => c -> Parser c c
expect c = satisfy ( == c )

ziffer :: Parser Char Integer
ziffer = ( \ c ->  fromIntegral
           $ fromEnum c - fromEnum '0' )
  <$> satisfy Data.Char.isDigit
```

Kombinatoren für Parser (I)

- Folge (and then) (ist `>>=` aus der Monade)
- Auswahl (oder) (ist Methode aus `class Alternative`)

```
( <|> ) :: Parser c a -> Parser c a -> Parser c a
Parser f <|> Parser g = Parser $ \ s -> f s ++ g s
```

- Wiederholung (beliebig viele, wenigstens einer)

```
many, some :: Parser c a -> Parser c [a]
many p = some p <|> return []
some p = (:) <$> p <*> many p
```

```
zahl :: Parser Char Integer =
  foldl (\ a z -> 10*a+z) 0 <$> some ziffer
```

Kombinatoren für Parser (II)

(aus `Control.Applicative`)

- der zweite Parser hängt nicht vom ersten ab:

```
(<*>) :: Parser c (a -> b)
      -> Parser c a -> Parser c b
```

- eines der Resultate wird exportiert, anderes ignoriert

```
(<*) :: Parser a -> Parser b -> Parser a
(*>) :: Parser a -> Parser b -> Parser b
```

  Eselsbrücke: Ziel des „Pfeiles“ wird benutzt

- der erste Parser ändert den Zustand nicht (`fmap`)

```
(<$>) :: (a -> b) -> Parser a -> Parser b
```

Kombinator-Parser und Grammatiken

- CFG-Grammatik mit Regeln $S \to aSbS$, $S \to \epsilon$ entspricht

```
s :: Parser Char ()
s =  (expect 'a' *> s *> expect 'b' *> s )
  <|> return ()
```

  Anwendung: `parse (s <* eof) "abab"`
- CFG: Variable $=$ Parser, nur `<*>` und `<|>` benutzen
- höherer Ausdrucksstärke (Chomsky-Stufe 1, 0) durch
  - Argumente ($\approx$ unendlich viele Variablen)
  - Monad (bind) statt Applicative (abhängige Fortsetzung)

Parser für Operator-Ausdrücke

- ```
  chainl :: Parser c a -> Parser c (a -> a -> a)
         -> Parser c a
  chainl p op = (foldl ...)
    <$> p <*> many ((,) <$> op <*> p)
  ```
- ```
  expression :: [[Parser c (a -> a -> a)]]
    -> Parser c a -> Parser c a
  expression opss atom =
    foldl ( \ p ops -> chainl ... ) atom opss
  ```
- ```
  exp = expression
    [ [ string "*" *> return Times ]
    , [ string "+" *> return Plus  ] ]
    ( Exp.Const <$> zahl )
  ```
  ```

Robuste Parser-Bibliotheken

- Designfragen:
  - Nichtdeterminismus einschränken
  - Backtracking einschränken
  - Fehlermeldungen (Quelltextposition)
- klassisches Beispiel: Parsec (Autor: Daan Leijen)
  `http://hackage.haskell.org/package/parsec`
- Ideen verwendet in vielen anderen Bibliotheken, z.B.
  `http://hackage.haskell.org/package/attoparsec`
  (benutzt z.B. in
  `http://hackage.haskell.org/package/aeson`)

Asymmetrische Komposition

gemeinsam:

```
(<|>) :: Parser c a -> Parser c a
      -> Parser c a
Parser p <|> Parser q = Parser $ \ s -> ...
```

- symmetrisch: `p s ++ q s`
- asymmetrisch: `if null p s then q s else p s`

Anwendung: `many` liefert nur maximal mögliche Wiederholung (nicht auch alle kürzeren)

Nichtdeterminismus einschränken

- Nichtdeterminismus = Berechnungsbaum = Backtracking
- asymmetrisches `p <|> q` : probiere erst `p`, dann `q`
- häufiger Fall: `p` lehnt „sofort“ ab

Festlegung (in Parsec): wenn `p` wenigstens ein Zeichen verbraucht, dann wird `q` nicht benutzt (d. h. `p` muß erfolgreich sein)
Backtracking dann nur durch `try p <|> q`

Fehlermeldungen

- Fehler = Position im Eingabestrom, bei der es „nicht weitergeht“
- und auch durch Backtracking keine Fortsetzung gefunden wird
- Fehlermeldung enthält:
  - Position
  - Inhalt (Zeichen) der Position
  - Menge der Zeichen mit Fortsetzung

Übung Kombinator-Parser

- Bezeichner parsen (alphabetisches Zeichen, dann Folge von alphanumerischen Zeichen)
- Whitespace ignorieren (verwende `Data.Char.isSpace`)
- Operator-Parser vervollständigen (chainl,...)
- konkrete Haskell-ähnliche Syntax realisieren
  - `if .. then .. else ..`
  - `let { .. = .. } in ..`
  - `\ x y z -> ..`
  - `f a b c`

Pretty-Printing (I)

John Hughes's and Simon Peyton Jones's Pretty Printer Combinators
Based on *The Design of a Pretty-printing Library* in *Advanced Functional Programming*, Johan Jeuring and Erik Meijer (eds), LNCS 925

`http://hackage.haskell.org/packages/archive/pretty/1.0.1.0/doc/html/Text-PrettyPrint-HughesPJ.html`

Pretty-Printing (II)

- `data Doc` abstrakter Dokumententyp, repräsentiert Textblöcke
- Konstruktoren:

```
text :: String -> Doc
```

- Kombinatoren:

```
vcat         :: [ Doc ] -> Doc -- vertikal
hcat, hsep :: [ Doc ] -> Doc -- horizontal
```

- Ausgabe: `render :: Doc -> String`

Bidirektionale Programme

Motivation: parse und (pretty-)print aus *einem* gemeinsamen Quelltext
Tillmann Rendel and Klaus Ostermann: *Invertible Syntax Descriptions*, Haskell Symposium 2010
`http://lambda-the-ultimate.org/node/4191`
Datentyp

```
data PP a = PP
  { parse :: String -> [(a,String)]
  , print :: a -> Maybe String
  }
```

Spezifikation, elementare Objekte, Kombinatoren?

Definition

(alles nach: Turbak/Gifford Ch. 17.9)
CPS-Transformation (continuation passing style):

- original: Funktion gibt Wert `v` zurück

```
f = \ x y -> let { ... } in v
```

- cps: Funktion erhält zusätzliches Argument, das ist eine *Fortsetzung* (continuation), die den Wert verarbeitet:

```
f_cps = \ x y k -> let { ... } in k v
```

aus `g (f 3 2)`
wird `\ k -> f_cps 3 2 $ \ v -> g_cps v k`

Motivation

Funktionsaufrufe in CPS-Programm kehren nie zurück, können also als Sprünge implementiert werden!

CPS als einheitlicher Mechanismus für

- Linearisierung (sequentielle Anordnung von primitiven Operationen)
- Ablaufsteuerung (Schleifen, nicht-lokale Sprünge)
- Unterprogramme (Übergabe von Argumenten und Resultat)
- Unterprogramme mit mehreren Resultaten

CPS für Linearisierung

`(a + b) * (c + d)` wird übersetzt (linearisiert) in

```
( \ top ->
   plus a b $ \ x ->
   plus c d $ \ y ->
   mal  x y top
) ( \ z -> z )
```

```
plus x y k = k (x + y)
mal  x y k = k (x * y)
```

später tatsächlich als Programmtransformation (Kompilation)

CPS für Resultat-Tupel

wie modelliert man Funktion mit mehreren Rückgabewerten?

- benutze Datentyp Tupel (Paar):

```
f : A -> (B, C)
```

- benutze Continuation:

```
f/cps : A -> (B -> C -> D) -> D
```

CPS/Tupel-Beispiel

erweiterter Euklidischer Algorithmus:

```
prop_egcd x y =
    let (p,q) = egcd x y
    in  (p*x + q*y) == gcd x y

egcd :: Integer -> Integer
     -> ( Integer, Integer )
egcd x y = if y == 0 then ???
           else let (d,m) = divMod x y
                    (p,q) = egcd y m
                in  ???
```

vervollständige, übersetze in CPS

CPS für Ablaufsteuerung

Wdhlg: CPS-Transformation von `1+(2*(3-(4+5)))` ist

```
\ top -> plus 4 5 $ \ a ->
         minus 3 a $ \ b ->
         mal 2 b $ \ c ->
         plus 1 c top
```

Neu: `label` und `jump`

```
1 + label foo (2 * (3 - jump foo (4 + 5)))
```

Semantik: durch `label` wird die aktuelle Continuation benannt:
`foo = \ c -> plus 1 c top`
und durch `jump` benutzt:

```
\ top -> plus 4 5 $ \ a -> foo a
```

Vergleiche: `label`: Exception-Handler deklarieren,
`jump`: Exception auslösen

Semantik für CPS

Semantik von Ausdruck `x` in Umgebung *E*
ist Funktion von Continuation nach Wert (Action)

```
value(E, label L B) = \ k ->
  value (E[L:=k], B) k
value (E, jump L B) = \ k ->
  value (E, L) $ \ k' ->
  value (E, B) k'
```

Beispiel 1:

```
value (E, label x x)
  = \ k -> value (E[x:=k], x) k
  = \ k -> k k
```

Beispiel 2

```
value (E, jump (label x x) (label y y))
= \ k ->
  value (E, label x x) $ \ k' ->
  value (E, label y y) k'
= \ k ->
```

Semantik

semantischer Bereich:

```
type Continuation a = a -> Action Val
newtype CPS a
    = CPS ( Continuation a -> Action Val )
value :: Env -> Exp -> CPS Val
```

Plan:

- abstrakte Syntax: `Label, Jump`, konkrete S. (Parser)
- Semantik:
  - Verkettung durch `>>=` aus `instance Monad CPS`
  - Einbetten von `Action Val` durch `lift`
  - `value` für bestehende Sprache
  - `value` für `label` und `jump`

CPS als Monade

- ein CPS-transformiertes Programm ausführen

```
feed :: CPS a -> ( a -> Action Val )
     -> Action Val
feed ( CPS s ) c = s c
```

- Spezifikation der Monaden-Operationen

```
feed ( return x ) c = c x
feed ( s >>= f ) c =
    feed s ( \ x -> feed ( f x ) c )
```

- Einbettung von Aktionen (Bsp: `put`,`get`,`new`)

```
lift :: Action a -> CPS a
```

Der CPS-Transformator

- `https://hackage.haskell.org/package/transformers/docs/Control-Monad-Trans-Cont.html`
  (Ross Paterson, 2001)
- ```
  newtype ContT r m a =
    ContT { runContT :: (a -> m r) -> m r }
  ```
- Beziehung zu voriger Folie:
  `CPS = ContT Val Action`, `feed = runContT`,
  `lift = Control.Monad.Trans.Class.lift`
- Ü: vergleiche unser label/jump mit `callCC`
- Ü: delimited Continuations (`reset`,`shift`)
  Kenichi Asai and Oleg Kiselyov, CW 2011
  `https://okmij.org/ftp/continuations/#tutorial`
  ```

Übungsaufgaben

Rekursion (bzw. Schleifen) mittels Label/Jump
(und ohne Rec oder Fixpunkt-Kombinator)
folgende Beispiele sind aus Turbak/Gifford, DCPL, 9.4.2

- Beschreibe die Auswertung (Datei `ex4.hs`)

```
let { d = \ f -> \ x -> f (f x) }
in  let { f = label l ( \ x -> jump l x ) }
    in  f d ( \ x -> x + 1 ) 0
```

- `jump (label x x) (label y y)`
- Ersetze `undefined`, so daß `f x = x!` (Datei `ex5.hs`)

```
let { triple x y z = \ s -> s x y z
    ; fst t = t ( \ x y z -> x )
    ; snd t = t ( \ x y z -> y )
    ; thd t = t ( \ x y z -> z )
    ; f x = let { p = label start undefined
                ; loop = fst p ; n = snd p ; a =
                } in if  0 == n then a
                     else loop (triple loop (n -
} in f 5
```