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

Johannes Waldmann, HTWK Leipzig

27. Januar 2020

1 Einleitung

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 → Compiler → Zielprogramm
  - Eingaben → Zielprogramm → Ausgaben
- mit Interpreter:
  - Quellprogramm, Eingaben → Interpreter → Ausgaben
- Mischform:
  - Quellprogramm → Compiler → Zwischenprogramm
  - Zwischenprogramm, Eingaben → virtuelle Maschine → 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`

2 Inferenz-Systeme

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: $I \vdash F : \Longleftrightarrow \exists\, I$-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 $K_1 \wedge K_2 \to K$.

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 $\frac{P \to Q, P}{Q}$,

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 $\mathsf{wert}(p, z)$ mit $p \in \texttt{Exp}$, $z \in \mathbb{Z}$

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

- Axiome: $\mathsf{wert}(\texttt{Const}\, z, z)$
- Regeln:
  $\dfrac{\mathsf{wert}(X, a), \mathsf{wert}(Y, b)}{\mathsf{wert}(\texttt{Plus}\ X\ Y, a + b)}$, $\dfrac{\mathsf{wert}(X, a), \mathsf{wert}(Y, b)}{\mathsf{wert}(\texttt{Times}\ X\ Y, a \cdot b)}, \ldots$

Umgebungen (Spezifikation)

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

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

3 Unterprogramme

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):

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

- Closure benutzen (Regel-Schema, 3 Prämissen)

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

- 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

- alternative Implementierung von Umgebungen
  - bisher `type Env = String -> Val`
  - jetzt `type Env = Data.Map.Map String Val`
- alternative Implementierung von Namen

  der Vergleich von Strings (als Schlüssel im Suchbaum) ist teuer. Ist für diesen Interpreter zeitkritisch!

- statt (implizie) `type Name = String` besser

  ```
  data Name =
    Name { hash :: Int, form :: String }
    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)`
- Compiler benutzt `Name` und `Env` nur zur Übersetzungszeit!
  Zur Laufzeit ist Umgebung ein Vektor von Vektoren (siehe VL PPS: Frames, statische Kette),
  die Längen und alle Indizes sind statisch bekannt.

- eingebaute primitive Rekursion (Induktion über Peano-Zahlen):
  implementieren Sie die Funktion `fold :: r -> (r -> r) -> N -> r`
  Testfall: `fold 1 (\x -> 2*x) 5 == 32`
  durch `data Exp = .. | Fold ..` und neuen Zweig in `value`
  Wie kann man damit die Fakultät implementieren?
- entwerfen und realisieren Sie einen entsprechenden Operator zur Simulation von `while`

4 Lambda-Kalkül (Wdhlg.)

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:

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

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

- 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)
  - $(\ldots((FA_1)A_2)\ldots A_n) \sim FA_1A_2\ldots A_n$
  - $\lambda x_1.(\lambda x_2.\ldots.(\lambda x_n.B)\ldots) \sim \lambda x_1x_2\ldots x_n.B$

  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*
  $\mathsf{wert}(E, X, v)$ ist *big-step-Semantik* für Interpreter
- Diese „passen zusammen.“ — Formal (Ansatz):
  $\forall X, Y \in \mathsf{Exp}(\mathsf{Abs}, \mathsf{App}, \mathsf{Ref}) : X \rightarrow_\beta^* Y \stackrel{?}{\Leftrightarrow} \mathsf{wert}(\emptyset, X, Y)$
- ist falsch (sinnfrei): Typen von Y passen nicht! Besser:
  $\forall X \in \mathsf{Exp}(\mathsf{Abs}, \mathsf{App}, \mathsf{Ref}, \mathsf{ConstBool}), B \in \{\mathsf{False}, \mathsf{True}\} : X \rightarrow_\beta^* \mathsf{ConstBool}\, B \stackrel{?}{\Longleftrightarrow} \mathsf{wert}(\emptyset, X, \mathsf{ValBool}\ B)$
- 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:
  $\mathsf{True} := \lambda xy.x, \mathsf{False} := \lambda xy.y$
- Verzweigung: `if` b `then` x `else` $y := bxy$
- natürliche Zahlen als iterierte Paare (Ansatz)
  $(0) := \langle \mathsf{True}, \lambda x.x \rangle;\ (n+1) := \langle \mathsf{False}, n \rangle$
- 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

- $c : \mathbb{N} \rightarrow \Lambda : n \mapsto \lambda fx.f^n(x)$
  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)

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

  Zeige, daß Y ein Fixpunktkombinator ist.

5 Fixpunkte

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 $\mathbb{R}^2$
- 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 $f : B \to B_\bot$

- 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:

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

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

Welche Funktionale sind stetig?

- die Erfahrung (der Programmierung mit rekursiven Funktionen) lehrt: alle Operatoren `\g -> \x -> .. g ..` sind stetig.
- denn die (von uns bisher benutzten) AST-Konstruktoren bilden stetige Ftk. auf stetige Fkt. ab
- Ü: $(\lambda x.\, \mathsf{if}\, x = \perp \mathsf{then}\, 42 \,\mathsf{else}\, \perp)$ 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: $\frac{\mathsf{wert}(E, (\lambda x.B)(\mathsf{Rec}\ x\ B), v)}{\mathsf{wert}(E, \mathsf{Rec}\ x\ B, v)}$
- 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
  ($\perp$ 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)

```
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 { x = 3 + 4 , y = x * x } 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

- Limes der Folge $F^k(\bot)$ für

  ```
  F h = \ x -> if x > 23 then x - 11
               else h (h (x + 14))
  ```

- 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?

- $C =$ 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)?
  $h : C \to C : L \mapsto \{a\} \cup L \cdot \{b\} \cdot (\Sigma^* \setminus L)$

6 Verpackung und Verteilung

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`

7 Zustand/Speicher

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

8 Monaden

... 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 $m \in \text{Mor}_C(S, T)$
- für jedes $O \in \text{Obj}_C$ gibt es $\text{id}_O : O \to O$
- für $f : S \to M$ und $g : M \to T$ gibt es $f \circ g : S \to T$.
  es gilt immer $f \circ \text{id} = f$, $\text{id} \circ g = g$, $f \circ (g \circ h) = (f \circ g) \circ h$

Beispiele:

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

Kategorische Definitionen und Sätze

Beispiel: Isomorphie

- eigentlich: Abbildung, die die Struktur (der abgebildeten Objekte) erhält
- Struktur von $O \in \text{Obj}(C)$ 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 $\mathrm{proj}_1 : P \to A_1, \mathrm{proj}_2 : P \to A_2$,
  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 $g \circ \mathrm{proj}_1 = f_1$ und $g \circ \mathrm{proj}_2 = f_2$
- 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:
  $\text{Obj}(K) = \text{Obj}(C)$, Pfeile: $s \to Ft$
- $\ldots K$ 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 $g \circ \text{proj}_i = f_i$ folgt $x_i = f_i(x) = (g \circ \text{proj}_i)(x) = \text{proj}_i(g(x)) = \text{proj}_i(g(y)) = (g \circ \text{proj}_i)(y) = f_i(y) = y_i$, Widerspruch.

2. (Aufgabe) $|P| \leq |A_1 \times A_2|$

- Kategorie mit gerichteten Graphen als Objekte,

  $f : \mathsf{V}(G) \to \mathsf{V}(H)$ ist Morphismus, falls $\forall x, y : (x, y) \in \mathsf{E}(G) \iff (f(x), f(y)) \in \mathsf{E}(H)$.

  Was bedeuten dort mono, epi?

  Produkt, Summe? Welche Graph-Operationen (aus VL Modellierung) haben die passenden Eigenschaften?

- Funktor- und Monadengesetze (mit leancheck in ghci)
- falsche Functor-/Monad-Instanzen für Maybe, List, Tree (d.h. typkorrekt, aber semantisch falsch)
- `StateT Int` zum Zählen der Auswertungsschritte (Aufrufe von `value`) einbauen
- `WriterT [String]` zur Protokollierung (Argumente und Resultat von `value`)
- Lesen der Umgebung mit `ReaderT Env`. Änderung der Umgebung (in Let, App) durch welche Funktion?

9 Kombinator-Parser

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?

10 Ablaufsteuerung/Continuations

Definition

(alles nach: Turbak/Gifford Ch. 17.9)

CPS-Transformation (continuation passing style):

- original: Funktion gibt Wert zurück

```
f == (abs (x y) (let ( ... ) v))
```

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

  ```
  f-cps == (abs (x y k) (let ( ... ) (k v))
  ```

  aus `g (f 3 2)` wird `f-cps 3 2 g-cps`

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 ->
  value (E, label y y) (value (E, label x x))
= \ k -> ( \ k0 -> k0 k0 ) ( \ k1 -> k1 k1 )
```

Semantik

semantischer Bereich:

```
type Continuation a = a -> Action Val
date CPS a
   = CPS ( Continuation a -> Action Val )
evaluate :: Env -> Exp -> CPS Val
```

Plan:

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

CPS als Monade

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

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

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

Der CPS-Transformator

- `https://hackage.haskell.org/package/transformers-0.5.6.2/docs/Control-Monad-Trans-Cont.html`
- `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.Cla`
- Ü: delimited Continuations (reset/shift)

Beispiele/Übung KW 50: Parser

- Parser für `\x y z -> ...`, benutze `foldr`
- Parser für `let { f x y = ... } in ...`
- Parser für `let { a = b ; c = d ; ... } in ..`
- `Text.Parsec.Combinator.notFollowedBy` zur Erkennung von Schlüsselwörtern
- Ziffern in Bezeichnern

Beispiele/Übung KW 50: CPS

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 = thd p
                } in if  0 == n then a
                     else loop (triple loop (n - 1) (n * a))
    } in f 5
```

11 Typen

Grundlagen

Typ = statische Semantik
(Information über mögliches Programm-Verhalten, erhalten ohne Programm-Ausführung)
formale Beschreibung:

- P: Menge der Ausdrücke (Programme)
- T: Menge der Typen
- Aussagen $p :: t$ (für $p \in P$, $t \in T$)
  - prüfen oder
  - herleiten (inferieren)

Inferenzsystem für Typen (Syntax)

- Grundbereich: Aussagen der Form $E \vdash X : T$
  (in Umgebung E hat Ausdruck X den Typ T)
- Menge der Typen:
  - primitiv: Int, Bool
  - zusammengesetzt:
    - Funktion $T_1 \to T_2$
    - Verweistyp $\mathsf{Ref}T$
    - Tupel $(T_1, \ldots, T_n)$, einschl. $n = 0$
- Umgebung bildet Namen auf Typen ab

Inferenzsystem für Typen (Semantik)

- Axiome f. Literale: $E \vdash$ Zahl-Literal : Int, ...
- Regel für prim. Operationen: $\frac{E \vdash X : \text{Int}, E \vdash Y : \text{Int}}{E \vdash (X + Y) : \text{Int}}, \ldots$
- Abstraktion/Applikation: ...
- Binden/Benutzen von Bindungen: ...

hierbei (vorläufige) Design-Entscheidungen:

- Typ eines Ausdrucks wird inferiert
- Typ eines Bezeichners wird ...
  - in Abstraktion: deklariert
  - in Let: inferiert

Inferenz für Let
(alles ganz analog zu Auswertung von Ausdrücken)

- Regeln für Umgebungen
  - $E[v := t] \vdash v : t$
  - $\frac{E \vdash v' : t'}{E[v := t] \vdash v' : t'}$ für $v \neq v'$
- Regeln für Bindung:
$$\frac{E \vdash X : s, \quad E[v := s] \vdash Y : t}{E \vdash \texttt{let}\ v = X\ \texttt{in}\ Y : t}$$

Applikation und Abstraktion

- Applikation:
$$\frac{E \vdash F : T_1 \to T_2, \quad E \vdash A : T_1}{E \vdash (FA) : T_2}$$
  vergleiche mit *modus ponens*

- Abstraktion (mit deklariertem Typ der Variablen)

$$\frac{E[v := T_1] \vdash X : T_2}{E \vdash (\lambda(v :: T_1)X) : T_1 \to T_2}$$

  dazu Erweiterung der abstrakten Syntax

Eigenschaften des Typsystems

Wir haben hier den *einfach getypten Lambda-Kalkül* nachgebaut:

- jedes Programm hat höchstens einen Typ
- nicht jedes Programm hat einen Typ.

  Der Y-Kombinator $(\lambda x.xx)(\lambda x.xx)$ hat keinen Typ
- jedes getypte Programm terminiert

  (Begründung: bei jeder Applikation FA ist der Typ von FA kleiner als der Typ von F)

Übung: typisiere `t t t t succ 0` mit `succ = \ x -> x + 1` und `t = \ f x -> f (f x)`

Übung (Wiederholung/Nachtrag)

1. Pretty-Printer für `Exp`,
2. Pretty-Printer und Parser für `Type`
3. `ExceptT` in `Sem` verwenden, bessere Fehlermeldungen in `module With`
4. `ContT` aus Transformers-Bibliothek, damit Nachnutzung der dort definierten Instanzen

12 Typ-Rekonstruktion

Motivation

- Bisher: Typ-Deklarationspflicht für Variablen in Lambda.
- scheint sachlich nicht nötig. In vielen Beispielen kann man die Typen einfach rekonstruieren:

```
let { t = \ f x -> f (f x)
    ; s = \ x -> x + 1
    } in t s 0
```

- Diesen Vorgang automatisieren!

Realisierung mit Constraints

Inferenz für Aussagen der Form $E \vdash X : (T, C)$

- E: Umgebung (Name $\rightarrow$ Typ)
- X: Ausdruck (Exp)
- T: Typ
- C: Menge von Typ-Constraints

wobei

- Menge der Typen T erweitert um Unbekannte U
- Constraint: Paar von Typen (T_1, T_2)
- Lösung eines Constraints-System C: Substitution $\sigma : U \rightarrow T$ mit $\forall (T_1, T_2) \in C : T_1\sigma = T_2\sigma$

Bsp: $U = \{u, v, w\}, C = \{(u, \mathsf{Int} \rightarrow v), (w \rightarrow \mathsf{Bool}, u)\}$,
$\sigma = \{(u, \mathsf{Int} \rightarrow \mathsf{Bool}), (v, \mathsf{Bool}), (w, \mathsf{Int})\}$

Inferenzregeln f. Rekonstruktion (Plan)

Plan:

- Aussage $E \vdash X : (T, C)$ ableiten,
- dann C lösen (allgemeinsten Unifikator σ bestimmen)
- dann ist $T\sigma$ der (allgemeinste) Typ von X (in Umgebung E)

Für (fast) jeden Teilausdruck eine eigene („frische“) Typvariable ansetzen, Beziehungen zwischen Typen durch Constraints ausdrücken.

Inferenzregeln? Implementierung? — Testfall:

```
\ f g x y ->
   if (f x y) then (x+1) else (g (f x True))
```

Inferenzregeln f. Rekonstrukion

- primitive Operationen (Beispiel)

$$\frac{E \vdash X_1 : (T_1, C_1), \quad E \vdash X_2 : (T_2, C_2)}{E \vdash X_1 + X_2 : (\text{Int}, \{T_1 = \text{Int}, T_2 = \text{Int}\} \cup C_1 \cup C_2)}$$

- Applikation

$$\frac{E \vdash F : (T_1, C_1), \quad E \vdash A : (T_2, C_2)}{E \vdash (FA) : \ldots}$$

- Abstraktion

$$\frac{\ldots}{E \vdash \lambda x.B : \ldots}$$

- (Ü) Konstanten, Variablen, if/then/else

Substitutionen (Definition)

- Signatur $\Sigma = \Sigma_0 \cup \ldots \Sigma_k$,
- $\mathrm{Term}(\Sigma, V)$ ist kleinste Menge T mit $V \subseteq T$ und $\forall 0 \leq i \leq k, f \in \Sigma_i, t_1 \in T, \ldots, t_i \in T : f(t_1, \ldots, t_i) \in T$.

  (hier Anwendung für Terme, die Typen beschreiben)
- Substitution: partielle Abbildung $\sigma : V \to \mathrm{Term}(\Sigma, V)$,

  Definitionsbereich: $\mathrm{dom}\,\sigma$, Bildbereich: $\mathrm{img}\,\sigma$.
- Substitution σ auf Term t anwenden: $t\sigma$
- σ heißt *pur*, wenn kein $v \in \mathrm{dom}\,\sigma$ als Teilterm in $\mathrm{img}\,\sigma$ vorkommt.

Substitutionen: Produkt

Produkt von Substitutionen: $t(\sigma_1 \circ \sigma_2) = (t\sigma_1)\sigma_2$

Beispiel 1:
$\sigma_1 = \{X \mapsto Y\}, \sigma_2 = \{Y \mapsto a\}, \sigma_1 \circ \sigma_2 = \{X \mapsto a, Y \mapsto a\}$.
Beispiel 2 (nachrechnen!):
$\sigma_1 = \{X \mapsto Y\}, \sigma_2 = \{Y \mapsto X\}, \sigma_1 \circ \sigma_2 = \sigma_2$
Eigenschaften:

- σ pur $\Rightarrow$ σ idempotent: $\sigma \circ \sigma = \sigma$
- σ_1 pur $\wedge$ σ_2 pur impliziert nicht $\sigma_1 \circ \sigma_2$ pur

Implementierung:

```
import Data.Map
type Substitution = Map Identifier Term
times :: Substitution -> Substitution -> Substition
```

Substitutionen: Ordnung

Substitution σ_1 ist *allgemeiner als* Substitution σ_2:
$\sigma_1 \underset{\sim}{<} \sigma_2 \iff \exists \tau : \sigma_1 \circ \tau = \sigma_2$
Beispiele:

- $\{X \mapsto Y\} \underset{\sim}{<} \{X \mapsto a, Y \mapsto a\}$,
- $\{X \mapsto Y\} \underset{\sim}{<} \{Y \mapsto X\}$,
- $\{Y \mapsto X\} \underset{\sim}{<} \{X \mapsto Y\}$.

Eigenschaften

- Relation $\underset{\sim}{<}$ ist Prä-Ordnung (..., ..., aber nicht ...)
- Die durch $\underset{\sim}{<}$ erzeugte Äquivalenzrelation ist die ...

Unifikation—Definition

Unifikationsproblem

- Eingabe: Terme $t_1, t_2 \in \mathrm{Term}(\Sigma, V)$
- Ausgabe: ein allgemeinster Unifikator (mgu): Substitution σ mit $t_1\sigma = t_2\sigma$.

(allgemeinst: infimum bzgl. $\underset{\sim}{<}$)
Satz: jedes Unifikationsproblem ist

- entweder gar nicht
- oder bis auf Umbenennung eindeutig

lösbar.

Unifikation—Algorithmus

$\mathrm{mgu}(s, t)$ nach Fallunterscheidung

- s ist Variable: ...
- t ist Variable: symmetrisch
- $s = (s_1 \to s_2)$ und $t = (t_1 \to t_2)$: ...

```
mgu :: Term -> Term -> Maybe Substitution
```

Unifikation—Komplexität

Bemerkungen:

- gegebene Implementierung ist korrekt, übersichtlich, aber nicht effizient,
- (Ü) es gibt Unif.-Probl. mit exponentiell großer Lösung,
- eine komprimierte Darstellung davon kann man aber in Polynomialzeit ausrechnen.

Bsp: Signatur $\{f/2, a/0\}$,
unifiziere $f(X_1, f(X_2, f(X_3, f(X_4, a))))$ mit $f(f(X_2, X_2), f(f(X_3, X_3), f(f(X_4, X_4), f(a, a))))$

Übung Rekonstruktion

- Bsp Unifikation Komplexität
- Testfälle `test/e{15,16}.prog`
- diskutiere (ggf. implementiere) Typrekonstruktion für Store-Operationen,
- ... für Continuations

13 Polymorphe Typen

Motivation

ungetypt:

```
let { t = \ f x -> f (f x)
    ; s = \ x -> x + 1
    } in (t t s) 0
```

einfach getypt nur so möglich:

```
let { t2 = \ (f :: (Int -> Int) -> (Int -> Int))
             (x :: Int -> Int) -> f (f x)
    ; t1 = \ (f :: Int -> Int) (x :: Int) -> f (f x)
    ; s = \ (x :: Int) -> x + 1
    } in (t2 t1 s) 0
```

wie besser?

Typ-Argumente (Beispiel)

- Typ-Abstraktion, Typ-Applikation:

```
let { t = \ (t :: *)
          ->  \ ( f :: t -> t ) ->
                \ ( x :: t ) ->
                      f ( f x )
    ; s = \ ( x :: int ) -> x + 1    }
in  (((t @(int -> int)) (t @int)) s) 0
```

  konkrete Syntax wie in Haskell (`-XTypeApplications`)

- zur Laufzeit werden die Typ-Abstraktionen und Typ-Applikationen *ignoriert*
- . . . besser: nach statischer Analyse *entfernt*

Typ-Argumente (Inferenz-Regeln)

- neuer Konstruktor `Pi Name Type` in `data Type`:
  $t \in \text{Var} \wedge T \in \text{Typ} \Rightarrow \Pi t.T \in \text{Typ}$,
  dabei kann t in T vorkommen (Konstruktor `Ref Name`)
- neue Konstruktoren in `data Exp`, mit Inferenz-Regeln:
  - Typ-Abstraktion (`TypeAbs Name Exp`)
    erzeugt parametrischen Typ $\dfrac{E \vdash X : T_1}{E \vdash \lambda(t :: *) \to X : \Pi t.T_1}$
  - Typ-Applikation (`TypeApp Exp Type`)
    instantiiert param. Typ $\dfrac{E \vdash F : \Pi t.T_1}{E \vdash F\ @T_2 : T_1[t := T_2]}$,
    dabei $[t := \dots]$ Ersetzen aller freien Vorkommen von t
- Typen von Ausdrücken sind (sollen sein) geschlossen:
  keine freien Typvariablen, vgl. $\lambda(a :: *)(x :: b).x$

Rekonstruktion polymorpher Typen

. . . ist im Allgemeinen nicht möglich:

Joe Wells: *Typability and Type Checking in System F Are Equivalent and Undecidable*, Annals of Pure and Applied Logic 98 (1998) 111–156, `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.6.6483`

übliche Einschränkung (ML, Haskell): *let-Polymorphismus*:

Typ-Abstraktionen nur für let-gebundene Bezeichner:

```
let { t = \ f x -> f(f x) ; s = \ x -> x+1 }
in t t s 0
```

folgendes ist dann nicht typisierbar (`t` ist monomorph):

```
( \ t s -> t t s 0 )
  ( \ f x -> f (f x) ) (\ x -> x+1)
```

Implementierung

Luis Damas, Roger Milner: *Principal Type Schemes for Functional Programs* 1982,

- Inferenzsystem ähnlich zu Rekonstruktion monomorpher Typen mit Aussagen der Form $E \vdash X : (T, C)$
- Umgebung E ist jetzt partielle Abbildung von Name nach *Typschema* (nicht wie bisher: nach Typ).
- Bei Typinferenz für let-gebundene Bezeichner wird über die freien Typvariablen generalisiert.
- Dazu Teil-Constraint-Systeme lokal lösen.

Jones 1999 `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.134.7274` Grabmüller 2006 `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.65.7733`,

14 Dependent Types

Funktionen zwischen Typen und Daten

- Funktionen von Daten nach Daten
  `\ (x :: Int) -> x + 1`

- Funktionen von Typ nach Daten
  `\ (t :: Type) -> \ (x :: t) -> x`
- Funktionen von Typ nach Typ (ML, Haskell, Java, C#)
  `\ (t :: Type) -> List t`
- Funktionen von Daten nach Typ (*dependent types*)
  `\ (t :: Typ) (n :: Int) -> Array t n`
  Sprachen: Cayenne, Coq, Agda

Implementierung

- Quelle: Lennart Augustsson: *Simpler, Easier!* `http://augustss.blogspot.com/2007/10/`
- kein Unterschied zw. `data Exp` und `data Type`,
- Typisierungs-Aussagen (Bsp): $0 :: \mathbb{N},\ \mathbb{N} :: *,\ * :: \Box$
- Applikation: $\dfrac{E \vdash f : \Pi(x :: A_1).B, E \vdash a : A_2}{E \vdash (fa) : B[x := a]}$
  mit $A_1 \equiv_\beta A_2$, (Vergleich von Normalformen)
- Abstraktion, mit $(s_1, s_2) \in \{*, \Box\}^2$ oder Teilmenge
  $$\frac{E \vdash A : s_1,\ E[x := A] \vdash b : B,\ E[x := A] \vdash B : s_2}{E \vdash (\lambda(x :: A).b) : \Pi(x :: A).B}$$
- Eigenschaften: 1. X hat Typ $\implies$ X terminiert,
  2. diese Sprachen sind nicht Turing-vollständig

Anwendung: Peano-Zahlen, Fold

- (Wdhlg) Peano-Zahlen mit Rekursionsschema

```
data N = Z | S N
fold :: r -> (r -> r) -> N -> r
```

- verallgemeinert: mglw. unterschiedliche Resultat-Typen

```
fold :: (f::N->*) -> f 0
  -> ((k::N)->f k->f(k+1))->(n::Nat)->f n
```

Ü: `fold` in den Interpreter einbauen (Auswertung und Typisierung)

- Anw.: Listen (geschachtelte Paare) bestimmter Länge:

```
list :: * -> N -> *
list a n = fold Unit (\t -> Pair a t) n
```

Übung: Tupel

- abstrakte Syntax

  ```
  data Exp = .. | Tuple [Exp] | Nth Int Exp
  ```

  beachte: *nicht* `Nth Exp Exp` wg. Typisierung
- konkrete Syntax (runde Klammern, Kommas, keine 1-Tupel)
- dynamische Semantik: `data Val = .. , value`
- Anwendung: das Beispiel mit label/jump umschreiben
- statische Semantik (Typen)
  - abstrakte Syntax
  - konkrete Syntax
  - Typisierung (Inferenzregeln, Implementierung)

15 Plan für Compiler

Transformationen/Ziel

- continuation passing (Programmablauf explizit)
- closure conversion (alle Umgebungen explizit)
- lifting (alle Unterprogramme global)
- Registervergabe (alle Argumente in Registern)

Ziel: maschinen(nahes) Programm mit

- globalen (Register-)Variablen (keine lokalen)
- Sprüngen (kein return)
- automatischer Speicherbereinigung

16 CPS-Transformation

CPS-Transformation: Spezifikation
(als Schritt im Compiler)

- Eingabe: Ausdruck X, Ausgabe: Ausdruck Y
- Semantik: Wert von $X =$ Wert von $Y(\lambda v.v)$
- Syntax:
  - $X \in$ Exp (fast) beliebig,
  - $Y \in$ Exp/CPS stark eingeschränkt:
    - keine geschachtelten Applikationen
    - Argumente von Applikationen und Operationen $(+, *, >)$ sind Variablen oder Literale

CPS-Transformation: Zielsyntax
drei Teilmengen von `data Exp`:

```
Exp_CPS ==> App Identifier Exp_Value^*
    | If Exp_Value Exp_CPS Exp_CPS
    | Let Identifier Exp_Letable Exp_CPS
Exp_Value ==> Literal | Identifier
Exp_Letable ==> Literal
    | Abs Identifier Exp_CPS
    | Exp_Value Op Exp_Value
```

Übung 1: Übersetze von `Exp` nach `Exp_CPS`:

```
(0 - (b * b)) + (4 * (a * c))
```

Übung 2: wegen CPS brauchen wir tatsächlich:

```
\ k -> k ((0 - (b * b)) + (4 * (a * c))
```

Beispiel

Lösung 1:

```
(0 - (b * b)) + (4 * (a * c))
==>
let { t.3 = b * b } in
  let { t.2 = 0 - t.3 } in
    let { t.5 = a * c } in
       let { t.4 = 4 * t.5 } in
         let { t.1 = t.2 + t.4 } in
            t.1
```

Lösung 2:

```
\ k -> let  ... in k t.1
```

CPS-Transf. f. Abstraktion, Applikation

vgl. Sect. 6 in: Gordon Plotkin: *Call-by-name, call-by-value and the λ-calculus*, Th. Comp. Sci. 1(2) 1975, 125–159 http://dx.doi.org/10.1016/0304-3975(75)90017-1 , http://homepages.inf.ed.ac.uk/gdp/

- $\mathrm{CPS}(v) = \lambda k.kv$
- $\mathrm{CPS}(FA) = \lambda k.(\mathrm{CPS}(F)(\lambda f.\,\mathrm{CPS}(A)(\lambda a.fak)))$
- $\mathrm{CPS}(\lambda x.B) = \lambda k.k(\lambda x.\,\mathrm{CPS}(B))$

dabei sind k, f, a *frische* Namen.

Bsp. $\mathrm{CPS}(\lambda x.9) = \lambda k_2.k_2(\lambda x.\,\mathrm{CPS}(9)) = \lambda k_2.k_2(\lambda x k_1.k_1 9)$,
$\mathrm{CPS}((\lambda x.9)8) = \lambda k_4.(\lambda k_2.k_2(\lambda x k_1.k_1 9))(\lambda f.((\lambda k_3.k_3 8)(\lambda a.fak_4)))$
Ü: Normalform von $\mathrm{CPS}((\lambda x.9)8)(\lambda z.z)$

Realisierung der CPS-Transformation

- $\mathrm{CPS}(X) = \mathrm{CPS}_{\mathsf{K}(X,\lambda x.x)}$ wobei

  $\mathrm{CPS}_{\mathsf{K:Exp->(ExpValue->ExpCPS)->ExpCPS}}$
- $\mathrm{CPS}_{\mathsf{K}(X,k)}$ erzeugt Ausdruck `let { .. } in .. let { v = .. } in` k `v`

  geschachtelte `Let`, der (letzte) Name v bezeichnet den Wert von X, Continuation k wird auf v angewendet

- Beispiel: $\mathrm{CPS}_{\mathsf{K}(X+Y,k)=\mathrm{CPS}_{\mathsf{K}(X,\lambda a.\ \mathrm{CPS}_{\mathsf{K}(Y,\lambda b.}\texttt{let\{v=a+b\}in }_{k(\mathsf{V}))}}}$
- dabei werden frische Namen (hier: v) benötigt, deswegen Rechnung in Zustandsmonade: $\mathrm{CPS}_{\mathsf{K}:}$ `Exp -> (ExpValue -> Fresh ExpCPS) -> Fresh ExpCPS`

Namen

- Bei der Übersetzung werden „frische“ Variablennamen benötigt (= die im Eingangsprogramm nicht vorkommen).
- ```
  module Control.Monad.State where
  data State s a = State ( s -> ( a, s ) )
  get :: State s s ; put :: s -> State ()
  evalState :: State s a -> s -> a

  fresh :: State Int String
  fresh = do k <- get ; put (k+1)
             return $ "f." ++ show k
  type Fresh a = State Int a
  ```
- wegen Zähler k sind alle diese Namen paarweise verschieden, können wegen `"."` nicht im Quelltext vorkommen.

Anwendung der Namens-Erzeugung

- ```
  fresh_let :: ExpLetable
  -> (ExpValue->Fresh ExpCPS) -> Fresh ExpCPS
  fresh_let a k  = do
    f <- fresh "l" ; b <- k ( Ref f )
    return $ Let f a b
  ```
- ```
  cpsk (Plus x y) k =
    cpsk x $ \ u -> cpsk y $ \ v ->
    fresh_let (Plus u v) k
  ```

Kompilation des If

- warum (wann) funktioniert das nicht?
  ```

```
cpsk (If b j n) k =
    cps b $ \ b' ->
    cps j $ \ j' -> cps n $ \ n' ->
    k (If b' j' n')
```

- es wird Code erzeugt, der beide Zweige auswertet, weil unser `Let` strikt ist.
  wenn einer der Zweig eine Rekursion enthält, dann erzeugt das Nichttermination.
- richtig ist: `cps b $ \ b' ->`

  ```
  If b' <$> cps j k <*> cps n k
  ```

Kompilation des Let

- ```
  cps (Let n a b) k =   -- FALSCH:
   cps a $ \a'->cps b $ \b'->k (Let n a' b')
  ```
  Testfall `cps (let a=9 in a+2) return`
  ergibt `let {l.0=a+2} in let {a=9} in l.0`
- U: auch falsch: `cps b $ \ b'->cps a $ \a'->..`
- besser (semantisch korrekt, aber Ziel-Syntax falsch):

  ```
  cps a $ \a'->do b'<-cps b k; return(Let n a' b')
  ```

- richtig (`a'` ist `ExpValue`, mglw. nicht `ExpLetable`)

  ```
  cps a $ \a'->do b'<-cps b k;return $ subst n a' b'
  ```

Umrechnung zw. Continuations (App)

- die Continuation (Funktion) im Compiler repräsentieren als Ausdruck (Abstraktion) der Zielsprache
- ```
  type Cont = ExpValue -> Fresh ExpCPS
  cont2exp :: Cont -> Fresh ExpCPS
  cont2exp k = do
      e <- fresh "e" ; out <- k (Ref e)
      return $ MultiAbs [e] out
  ```

- ```
  cpsk (App f a) k =
    cps f $ \ f' -> cps a $ \ a' ->
    cont2exp k >>= \ x -> fresh_let x $ \ c ->
     return $ MultiApp f' [ a', c ]
  ```

Umrechnung zw. Continuations (Abs)

- Continuation im Quelltext → Continuation im Compiler

  ```
  type Cont = ExpValue -> Fresh ExpCPS
  exp2cont :: Name -> Cont
  exp2cont c = \ v -> return $ MultiApp (Ref c) [v]
  ```

- Anwendung bei Abstraktion

  ```
  Abs x b -> \ k -> do
      c <- fresh "k"
      b' <- cps b $ exp2cont c
      fresh_let (MultiAbs [ x, c ] b') k
  ```

Vergleich CPS-Interpreter/Transformator

Wiederholung CPS-Interpreter:

```
type Cont = Val -> Action Val
eval :: Env -> Exp -> Cont -> Action Val
eval env x = \ k -> case x of
    ConstInt i -> ...
    Plus a b -> ...
```

CPS-Transformator:

```
type Cont = ExpValue -> Fresh ExpCPS
cps :: Exp -> Cont -> Fresh ExpCPS
cps x = \ m -> case x of
    ConstInt i -> ...
    Plus a b -> ...
```
```

Übung CPS-Transformation

- Transformationsregeln für Ref, App, Abs, Let nachvollziehen (im Vergleich zu CPS-Interpreter)
- Transformationsregeln für if/then/else, new/put/get hinzufügen
- anwenden auf eine rekursive Funktion (z. B. Fakultät),
  wobei Rekursion durch Zeiger auf Abstraktion realisiert wird

17 Closure Conversion

Motivation

(Literatur: DCPL 17.10) — Beispiel:

```
let { linear = \ a -> \ x -> a * x + 1
    ; f = linear 2 ; g = linear 3
    }
in  f 4 * g 5
```

beachte nicht lokale Variablen: `(\ x -> .. a .. )`

- Semantik-Definition (Interpreter) benutzt Umgebung
- Transformation (closure conversion, environment conversion) (im Compiler) macht Umgebungen explizit.

Spezifikation

closure conversion:

- Eingabe: Programm P
- Ausgabe: äquivalentes Programm P', bei dem alle Abstraktionen *geschlossen* sind
- zusätzlich: P in CPS $\Rightarrow$ P' in CPS

geschlossen: alle Variablen sind lokal

Ansatz:

- Werte der benötigten nicht lokalen Variablen $\Rightarrow$ zusätzliche(s) Argument(e) der Abstraktion
- auch Applikationen entsprechend ändern

closure passing style

- Umgebung = Tupel der Werte der benötigten nicht lokalen Variablen
- Closure = Paar aus Code und Umgebung
  realisiert als Tupel (Code, $\underbrace{W_1, \ldots, W_n}_{\text{Umgebung}}$)

```
\ x -> a * x + 1
==>
\ clo x ->
  let { a = nth 1 clo } in a * x + 1
```

Closure-Konstruktion? Komplette Übersetzung des Beispiels?

Transformation

```
CLC[ \ i_1 .. i_n -> b ] =
   (tuple ( \ clo i_1 .. i_n ->
            let { v_1 = nth 1 clo ; .. }
            in  CLC[b]
          ) v_1 .. )
```

wobei $\{v_1, \ldots\}$ = freie Variablen in $(\lambda i_1 \ldots i_n \to b)$

```
CLC[ (f a_1 .. a_n) ] =
  let { clo = CLC[f]
      ; code = nth 0 clo
  } in  code clo CLC[a_1] .. CLC[a_n]
```

- für alle anderen Fälle: strukturelle Rekursion
- zur Erhaltung der CPS-Form: Spezialfall bei `let`

18 Lifting

Spezifikation

(lambda) lifting:

- Eingabe: Programm P, bei dem alle Abstraktionen geschlossen sind

- Ausgabe: äquivalentes Programm P', bei dem alle Abstraktionen (geschlossen und) global sind

  Motivation: in Maschinencode gibt es nur globale Sprungziele
  (CPS-Transformation: Unterprogramme kehren nie zurück $\Rightarrow$ globale Sprünge)

Realisierung

nach closure conversion sind alle Abstraktionen geschlossen, diese müssen nur noch aufgesammelt und eindeutig benannt werden.

```
let { g1 = \ v1 .. vn -> b1
    ...
    ; gk = \ v1 .. vn -> bk
} in b
```

dann in `b1, .., bk, b` keine Abstraktionen gestattet

- Zustandsmonade zur Namenserzeugung ($g_1, g_2, \ldots$)
- Ausgabemonade (`WriterT`) zum Aufsammeln
- $g_1, \ldots, g_k$ dürften nun sogar rekursiv sein (sich gegenseitig aufrufen)

Lambda-Lifting (Plan)

um ein Programm zu erhalten, bei dem alle Abstraktionen global sind:

- bisher: closure conversion + lifting:

  (verwendet Tupel)

- Alternative: lambda lifting

  (reiner λ-Kalkül, keine zusätzlichen Datenstrukturen)

Lambda-Lifting (Realisierung)

- verwendet Kombinatoren (globale Funktionen)

  $I = \lambda x.x, S = \lambda xyz.xz(yz), K = \lambda xy.x$

- und Transformationsregeln

  $\mathsf{lift}(FA) = \mathsf{lift}(F)\,\mathsf{lift}(A), \mathsf{lift}(\lambda x.B) = \mathsf{lift}_x(B);$

- Spezifikation: $\mathsf{lift}_x(B)x \to_\beta^* B$
- Implementierung:
  falls $x \notin \mathsf{Free}(B)$, dann $\mathsf{lift}_x(B) = KB$;
  sonst $\mathsf{lift}_x(x) = I, \mathsf{lift}_x(FA) = S\,\mathsf{lift}_x(F)\,\mathsf{lift}_x(A)$

Beispiel: $\mathsf{lift}(\lambda x.\lambda y.yx) = \mathsf{lift}_x(\mathsf{lift}_y(yx)) = \mathsf{lift}_x(SI(Kx)) = S(K(SI))(S(KK)I)$

19 Registervergabe

Motivation

- (klassische) reale CPU/Rechner hat nur *globalen* Speicher (Register, Hauptspeicher)
- Argumentübergabe (Hauptprogramm → Unterprogramm) muß diesen Speicher benutzen

  (Rückgabe brauchen wir nicht wegen CPS)
- Zugriff auf Register schneller als auf Hauptspeicher ⇒ bevorzugt Register benutzen.

Plan (I)

- Modell: Rechner mit beliebig vielen Registern $(R_0, R_1, \ldots)$
- Befehle:
  - Literal laden (in Register)
  - Register laden (kopieren)
  - direkt springen (zu literaler Adresse)
  - indirekt springen (zu Adresse in Register)
- Unterprogramm-Argumente in Registern:
  - für Abstraktionen: $(R_0, R_1, \ldots, R_k)$
    (genau diese, genau dieser Reihe nach)
  - für primitive Operationen: beliebig
- Transformation: lokale Namen → Registernamen

Plan (II)

- Modell: Rechner mit begrenzt vielen realen Registern,
  z. B. $(R_0, \ldots, R_7)$
- falls diese nicht ausreichen: *register spilling*
  virtuelle Register in Hauptspeicher abbilden
- Hauptspeicher (viel) langsamer als Register:
  möglichst wenig HS-Operationen:
  geeignete Auswahl der Spill-Register nötig

Registerbenutzung

- Allgemeine Form der Programme:

```
let { r1 = .. ; r2 = .. } in r4 ..
```

- für jeden Zeitpunkt ausrechnen: Menge der *freien* Register (= deren aktueller Wert nicht (mehr) benötigt wird)
- nächstes Zuweisungsziel ist niedrigstes freies Register (andere Varianten sind denkbar)
- vor jedem UP-Aufruf: *register shuffle* (damit die Argumente in $R_0, \ldots, R_k$ stehen)

Compiler-Übungen

1. Testen Sie die richtige Kompilation von

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

2. implementieren Sie fehlende Codegenerierung/Runtime für

```
let { p = new 42
    ; f = \ x -> if (x == 0) then 1
                 else  (x * (get p) (x-1))
    ; foo = put p f
    } in f 5
```

3. ergänzen Sie das Programm, so daß $5!$ ausgerechnet wird

```
let { f = label x (x, 5, 1) }
in  if ( 0 == nth 1 f )
    then nth 2 f
    else jump ...
              (..., ..., ...)
```

   Kompilieren Sie das Programm. Dazu muß für `label`, `jump`, `tuple`, `nth` die CPS-Transformation implementiert werden.

4. Bei der CPS-Transformation von `If` wird die Continuation k kopiert:

```
If b' <$> cps j k <*> cps n k
```

   Geben Sie ein Beispiel dafür an, daß dadurch der Ausgabetext groß werden kann.

   Geben Sie eine andere Übersetzung für `If` an, bei der k nur einmal angewendet wird. (Benutzen Sie `cont2exp`.)

   Überprüfen Sie für das eben angegebene Beispiel.

5. Alle Tests bis hier kompilieren konstante Ausdrücke. Diese könnte man auch sofort auswerten.

   Ändern Sie Compiler und Laufzeitsystem, so daß Funktionen kompiliert werden, die zur Laufzeit Argumente (Zahlen) von der Kommandozeile bekommen.

6. Register Allocator in GCC: siehe `https://gcc.gnu.org/onlinedocs/gccint/RTL-passes.html`.

   Suchen Sie die entsprechende Beschreibung/Implementierung in `clang` (`llvm`).

   Vergleichen Sie an Beispielen die Anzahl der Register in dem von unserem Compiler erzeugten Code und dem daraus durch GCC (clang) erzeugten Assemblercode.

   Wo findet die Register-Allokation für Java-Programme statt?

und wer Zeit hat ...

- Typprüfung vor Kompilation
- exakte Spezifikation und Beweis der CPS-Transformation
- Verbesserung der Registerzuweisung

- anderes Back-End (nicht C): Maschinencode für reale oder virtuelle Maschine, z.B. LLVM IR `http://llvm.org/docs/LangRef.html#introduction`, WASM `https://webassembly.org/`
- was fehlt, bis der Compiler seinen eigenen Quelltext übersetzen kann?

20 Automatische Speicherverwaltung

Motivation

Speicher-Allokation durch Konstruktion von

- Zellen,Tupel, Closures

Modell: Speicherbelegung = gerichteter Graph
Knoten *lebendig*: von Register aus erreichbar.
sonst tot ⇒ automatisch freigeben

Gliederung:

- mark/sweep (pointer reversal, Schorr/Waite 1967)
- twospace (stop-and-copy, Cheney 1970)
- generational (JVM)

Mark/Sweep

Plan: wenn Speicher voll, dann:

- alle lebenden Zellen markieren
- alle nicht markierten Zellen in Freispeicherliste

Problem: zum Markieren muß man den Graphen durchqueren, man hat aber keinen Platz (z. B. Stack), um das zu organisieren.

Lösung:
H. Schorr, W. Waite: *An efficient machine-independent procedure for garbage collection in various list structures*, Communations of the ACM, 10(8):481-492, August 1967.
temporäre Änderungen im Graphen selbst (pointer reversal)

Pointer Reversal (Invariante)

ursprünglicher Graph G_0, aktueller Graph G:
Knoten (cons) mit zwei Kindern (head, tail), markiert mit

- 0: noch nicht besucht
- 1: head wird besucht (head-Zeiger ist invertiert)
- 2: tail wird besucht (tail-Zeiger ist invertiert)
- 3: fertig

globale Variablen p (parent), c (current).
Invariante: man erhält G_0 aus G, wenn man

- head/tail-Zeiger aus 1/2-Zellen (nochmals) invertiert
- und Zeiger von p auf c hinzufügt.

Pointer Reversal (Ablauf)

- pre: $p =$ null, $c =$ root, $\forall z : \mathrm{mark}(z) = 0$
- post: $\forall z : \mathrm{mark}(z) =$ if (root $\rightarrow^* z$) then 3 else 0

Schritt (neue Werte immer mit $'$): falls $\mathrm{mark}(c) = \ldots$

- 0: $c' = \mathrm{head}(c); \mathrm{head}'(c) = p; \mathrm{mark}'(c) = 1; p' = c;$
- 1,2,3: falls $\mathrm{mark}(p) = \ldots$
  - 1: $\mathrm{head}'(p) = c; \mathrm{tail}'(p) = \mathrm{head}(p); \mathrm{mark}'(p) = 2; c' = \mathrm{tail}(p); p' = p$
  - 2: $\mathrm{tail}'(p) = c; \mathrm{mark}'(p) = 3; p' = \mathrm{tail}(p); c' = p;$

Knoten werden in Tiefensuch-Reihenfolge betreten.

Eigenschaften Mark/Sweep

- benötigt 2 Bit Markierung pro Zelle, aber keinen weiteren Zusatzspeicher
- Laufzeit für mark $\sim$ | lebender Speicher |
- Laufzeit für sweep $\sim$ | gesamter Speicher |
- Fragmentierung (Freispeicherliste springt)

Ablegen von Markierungs-Bits:

- in Zeigern/Zellen selbst

  (Beispiel: Rechner mit Byte-Adressierung, aber Zellen immer auf Adressen $\equiv 0 \pmod 4$: zwei LSB sind frei.)

- in separaten Bitmaps

Stop-and-copy (Plan)

Plan:

- zwei Speicherbereiche (Fromspace, Tospace)
- Allokation im Fromspace
- wenn Fromspace voll, kopiere lebende Zellen in Tospace und vertausche dann Fromspace $\leftrightarrow$ Tospace

auch hier: Verwaltung ohne Zusatzspeicher (Stack)

C. J. Cheney: *A nonrecursive list compacting algorithm*, Communications of the ACM, 13(11):677–678, 1970.

Stop-and-copy (Invariante)

fromspace, tospace : array [$0 \ldots N$] of cell

Variablen: $0 \leq$ scan $\leq$ free $\leq N$

einige Zellen im fromspace enthalten Weiterleitung ($=$ Adresse im tospace)

Invarianten:

- scan $\leq$ free
- Zellen aus tospace [0 . . . scan-1] zeigen in tospace
- Zellen aus tospace [scan . . . free-1] zeigen in fromspace
- wenn man in G (mit Wurzel tospace[0]) allen Weiterleitungen folgt, erhält man isomorphes Abbild von G_0 (mit Wurzel fromspace[0]).

Stop-and-copy (Ablauf)

- pre: tospace[0] = Wurzel, scan = 0,free = 1.
- post: scan = free

Schritt: while scan $<$ free:

- für alle Zeiger p in tospace[scan]:
  - falls fromspace[p] weitergeleitet auf q, ersetze p durch q.
  - falls keine Weiterleitung
    - kopiere fromspace[p] nach tospace[free],
    - Weiterleitung fromspace[p] nach free eintragen,
    - ersetze p durch free, erhöhe free.
- erhöhe scan.

Besucht Knoten in Reihenfolge einer Breitensuche.

Stop-and-copy (Eigenschaften)

- benötigt „doppelten“ Speicherplatz
- Laufzeit $\sim |$ lebender Speicher $|$
- kompaktierend
- Breitensuch-Reihenfolge zerstört Lokalität.

Speicher mit Generationen

Beobachtung: es gibt

- (viele) Zellen, die sehr kurz leben
- Zellen, die sehr lange (ewig) leben

Plan:

- bei den kurzlebigen Zellen soll GC-Laufzeit $\sim$ Leben (und nicht $\sim$ Leben + Müll) sein
- die langlebigen Zellen möchte man nicht bei jeder GC besuchen/kopieren.

Lösung: benutze Generationen, bei GC in Generation k: betrachte alle Zellen in Generationen $> k$ als lebend.

Speicherverwaltung in JVM

Speicheraufteilung:

- Generation 0:
  - Eden, Survivor 1, Survivor 2
- Generation 1: Tenured

Ablauf

- minor collection (Eden voll):

  kompaktierend: Eden + Survivor 1/2 $\rightarrow$ Survivor 2/1 ...

  ... falls dabei Überlauf $\rightarrow$ Tenured
- major collection (Tenured voll):

  alles nach Survivor 1 (+ Tenured)

Speicherverwaltung in JVM (II)

- richtige Benutzung der Generationen:
  - bei minor collection (in Gen. 0) gelten Zellen in Tenured (Gen. 1) als lebend (und werden nicht besucht)
  - Spezialbehandlung für Zeiger von Gen. 1 nach Gen. 0 nötig (wie können die überhaupt entstehen?)
- Literatur: `http://www.imn.htwk-leipzig.de/~waldmann/edu/ws09/pps/folien/main/node78.html`
- Aufgabe: `http://www.imn.htwk-leipzig.de/~waldmann/edu/ws09/pps/folien/main/node79.html`

21 Ausblick

Informative Typen

in $X :: T$, der deklarierte Typ T kann eine schärfere Aussage treffen als aus X (Implementierung) ableitbar.

```
data T a = C a -- C :: a -> T a
data T a where C :: Bool -> T Bool
```

das ist u.a. nützlich bei der Definition und Implementierung von (eingebetteten) domainspezifischen Sprachen

- generalized algebraic data types GADTs
- (parametric) higher order abstract syntax (P)HOAS
- Dependent Types (in Haskell)

GADT

- üblich (algebraischer Datentyp, ADT)

```
data Tree a =
  Leaf a | Branch (Tree a) (Tree a)
```

- äquivalente Schreibweise:

```
data Tree a where
  Leaf :: a -> Tree a
  Branch :: Tree a -> Tree a -> Tree a
```

- Verallgemeinerung (generalized ADT)

```
data Exp a where
  ConsInt :: Int -> Exp Int
  Greater :: Exp Int -> Exp Int -> Exp Bool
```

- Dimitrios Vytiniotis, Stephanie Weirich, Simon Peyton Jones: *Simple . . . Type Inference for GADTs*, ICFP 2006

Higher Order Abstract Syntax

```
data Exp a  where
  Var :: a -> Exp a
  Abs :: (a -> Exp b) -> Exp (a -> b)
  App :: Exp (a -> b) -> Exp a -> Exp b

App (Abs $ \x -> Plus (C 1) (Var x)) (C 2)
```

```
value :: Exp a -> a
value e = case e of
  App f a -> value f ( value a )
```

Ü: vervollständige Interpreter

Quelle: Frank Pfenning, Conal Elliott: *HOAS*, PLDI 1988 `http://conal.net/papers/`

22 Zusammenfassung

Methoden

- Inferenzsysteme
- Lambda-Kalkül
- (algebraischen Datentypen, Pattern Matching, Funktionen höherer Ordnung)
- Monaden

Semantik

- dynamische (Programmausführung)
  - Interpretation
    - funktional,
    - imperativ (Speicher)
    - Ablaufsteuerung (Continuations)
  - Transformation (Kompilation)
    - CPS transformation
    - closure passing, lifting, Registerzuweisung
- statische: Typisierung (Programmanalyse)
  - monomorph/polymorph
  - deklariert/rekonstruiert

Monaden zur Programmstrukturierung

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

Anwendungen:

- semantische Bereiche f. Interpreter,
- Parser,
- Unifikation

Testfragen (für jede Monad-Instanz):

- Typ (z. B. Action)
- anwendungsspezifische Elemente (z. B. new, put)
- Implementierung der Schnittstelle (return, bind)

Wozu braucht man den Compilerbau?

- jedes Programm verwendet Methoden des Compiler-(d.h. Übersetzer)baus

  (nach Definition übersetzt es Eingabe in Ausgabe)

  Gerhard Goos zugeschrieben, `https://dblp.uni-trier.de/pers/hd/g/Goos:Gerhard`
- G.Goos: *Issues in Compiling*, JUCS 2001, `http://dx.doi.org/10.3217/jucs-007-05-0410`
- *Ask HN: Are compiler engineers still in demand, and what do they work on?* 20. 1. 2020, `https://news.ycombinator.com/item?id=22096628`

Prüfungsvorbereitung

Beispielklausur `http://www.imn.htwk-leipzig.de/~waldmann/edu/ws11/cb/klausur/`

- was ist eine Umgebung (`Env`), welche Operationen gehören dazu?
- was ist eine Speicher (`Store`), welche Operationen gehören dazu?

- Gemeinsamkeiten/Unterschiede zw. Env und Store?
- Für $(\lambda x.xx)(\lambda x.xx)$: zeichne den Syntaxbaum, bestimme die Menge der freien und die Menge der gebundenen Variablen. Markiere im Syntaxbaum alle Redexe. Gib die Menge der direkten Nachfolger an (einen Beta-Schritt ausführen).
- Definiere Beta-Reduktion und Alpha-Konversion im Lambda-Kalkül. Wozu wird Alpha-Konversion benötigt? (Dafür Beispiel angeben.)
- Wie kann man Records (Paare) durch Funktionen simulieren? (Definiere Lambda-Ausdrücke für `pair, first, second`)
- welche semantischen Bereiche wurden in den Interpretern benutzt? (definieren Sie `Val, Action Val, CPS Val`)
- welches sind die jeweils hinzukommenden Ausdrucksmöglichkeiten der Quellsprache (`Exp`)?
- wie lauten die Monad-Instanzen für `Action, CPS, Parser`, was bedeutet jeweils das bind (>>=)?
- warum benötigt man call-by-name für Abstraktionen über den Programmablauf (warum kann man `if` oder `while` nicht mit call-by-value implementieren)?
- wie kann man call-by-name simulieren in einer call-by-value-Sprache?
- wie kann man call-by-value simulieren in einer call-by-name-Sprache (Antwort: durch CPS-Transformation)
- Definiere Fakultät mittels Fixpunktoperator (Definiere das `f` in `fak = fix f`)
- Bezüglich welcher Halbordnung ist dieses `f` monoton? (Definiere die Ordnung, erläutere Monotonie an einem Beispiel.)
- Wie kann man Rekursion durch get/put simulieren? (Programmbeispiel ergänzen)
- Wie kann man Rekursion durch label/jump simulieren? (Programmbeispiel ergänzen)
- Für die Transformationen CPS, Closure Conv., Lifting, Registervergabe: welche Form haben jeweils Eingabe- und Ausgabeprogramm? Auf welchem Maschinenmodell kann das Zielprogramm ausgeführt werden? (Welche Operationen muß das Laufzeitsystem bereitstellen?)

- Was sind die Bestandteile eines Inferenzsystems (Antwort: Grundbereich, Axiome, Regeln), wie kann man ein Axiom als Spezialfall einer Regel auffassen?
- wie lauten die Inferenzregeln für das Nachschlagen eines Namens in einer Umgebung?
- Inferenzregeln für Applikation, Abstraktion, Let, If/Then/Else im einfach getypten Kalkül
- Geben Sie ein Programm an, das sich nicht einfach (sondern nur polymorph) typisieren läßt. Geben Sie den polymorphen Typ an.
- Inferenz-Regeln für Typ-Applikation, Typ-Abstraktion im polymorphen Kalkül
- für Typ-Rekonstruktion im einfach getypten Kalkül: Welches ist der Grundbereich des Inferenzsystems?
- geben Sie die Inferenzregel für Typrekonstruktion bei If/Then/Else an
- Geben Sie eine Inferenzregel für Typrekonstruktion an, durch die neue Variablen eingeführt werden.
- Wann ist σ ein Unifikator von zwei Termen s, t?
- Geben Sie zwei verschiedene Unifikatoren von $f(a, X)$ und $f(Y, Z)$ an. Einer davon soll streng allgemeiner als der andere sein. Begründen Sie auch diese Beziehung.
- Bestimmen Sie einen Unifikator von $f(X_n, f(X_{n-1}, \ldots, f(X_0, a)\ldots))$ und $f(f(X_{n-1}, X_{n-1}), f(f(X_{n-$