

Compilerbau

Vorlesung

Johannes Waldmann, HTWK Leipzig

WS 08–11,13,15,17,19,SS 22,24,WS 25

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

Ü: was bedeutet `cld`, 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: siehe Repo
- Bsp Java-Kompilation: <https://www.imn.htwk-leipzig.de/~waldmann/etc/safe-speed/>
- Bsp Haskell-Kompilation:

MicroHS (Lennart Augustsson, Haskell Symposium 2024,  
<https://github.com/augustss/MicroHs/blob/master/doc/hs2024.pdf> )

# 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)

# Einleitung: Sprachverarbeitung

- mit Interpreter:
  - Quellprogramm, Eingaben  $\xrightarrow{\text{Interpreter}}$  Ausgaben
- mit Compiler:
  - Quellprogramm  $\xrightarrow{\text{Compiler}}$  Zielprogramm
  - Eingaben  $\xrightarrow{\text{Zielprogramm}}$  Ausgaben
- Mischform:
  - Quellprogramm  $\xrightarrow{\text{Compiler}}$  Zwischenprogramm
  - Zwischenprogramm, Eingaben  $\xrightarrow{\text{virtuelle Maschine}}$  Ausgaben
- reale Maschine (CPU) ist Interpreter für Maschinensprache (Interpretation in Hardware, in Microcode)
- gemeinsam ist: syntaxgesteuerte Semantik (Ausführung oder Übersetzung)

# 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, Datenbankabfragesprachen)
- 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
- Prüfungszulassung: regelmäßiges und erfolgreiches Bearbeiten von Übungsaufgaben
- Prüfung: Klausur (120 min, keine Hilfsmittel)  
Bei Interesse und nach voriger Absprache: Ersatz eines Teiles der Klausur durch vorherige Hausarbeit  
z.B. Reparaturen an autotool-Aufgaben oder anderem open-source-Projekt (Ihrer Wahl), bei denen Techniken des Compilerbaus angewendet werden

# 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
```

# Bezeichner sind Strings — oder nicht?

- ... | `Let String Exp Exp` — wirklich?
- es gilt `type String = [Char]`, also
  - einfach verkettete Liste von Zeichen
  - mit Bedarfsauswertung (lazy Konstruktoren)
- das ist
  - ineffizient (in Platz *und* Zeit)
  - egal (für unseren einfachen Anwendungsfall)
  - gefährlich (wenn man es für andere Anwendungen übernimmt)
- deswegen jetzt schon Diskussion ...
  - von alternativen Implementierungen
  - und wie man diese versteckt

# Datentypen für Folgen (von Zeichen)

- `type String = [Char]`: einfach verkettet, lazy: ist in den allermeisten Fällen unzweckmäßig
- `data Vector a`: Array (d.h., zusammenhängender Speicherbereich, deswegen effiziente Indizierung) mit kostenlosem *slicing* (Abschnitt-Bildung)
- `data ByteString`:  $\approx$  Vektor von Bytes (d.h., für rein binären Datenaustausch)
- `data Text` (aus `Modul Data.Text`) *efficient packed, immutable Unicode text type*, (d.h., Zeichen = Bytefolge)
- `Modul Data.Text.Lazy`: lazy Liste von (strikten) `Text`-Abschnitten, für Stream-Verarbeitung

# Verstecken von Implementierungsdetails

- Implementierung direkt sichtbar:

```
data Exp = ... | Let Text Exp Exp
```

- Verschieben der Implementierungs-Entscheidung:

```
type Id = Text; data Exp = ... | Let Id Exp
```

bleibt aber sichtbar (`type`-Deklarationen werden bei Kompilation immer expandiert)

- Verstecken der Entscheidung:

```
modul Id (Id) where data Id = Id Text
```

exportiert wird Typ-Name, aber nicht der Konstruktor der Anwender (Importeur) von `Id` sieht `Text` nicht

- `data`-Deklaration mit genau einem Konstruktor:

```
erstetzen durch newtype Id = Id Text
```

dieser kostet *gar nichts* (keine Zeit, keinen Platz)

# Verwendung standardisierter Namen

- alle benötigten Funktionen (einschl. Konstruktoren) für `Id` implementieren und exportieren (es sind nicht viele)

```
eqId :: Id -> Id -> Bool; eqId (Id s) (Id t) = s == t
```

- diese spezifischen Namen will sich keiner merken  $\Rightarrow$  verwende standardisierte Typklassen, Bsp.

```
instance Eq Id where (Id s) == (Id t) = s == t
```

der Importeur von `Id` sieht den Namen `(==)` bereits, weil er in `Prelude` definiert ist

- wenn die Implementierung einer standardisierten Klasse eine einfache Delegation ist, kann sie vom Compiler erzeugt werden

```
newtype Id = Id Text deriving Eq
```

# Einsparung von Konstruktor-Aufrufen

- `-- Implementierung des Konstruktors`  
`import qualified Data.Text as T`  
`fromString :: String -> Id; fromString s = Id (T.pack s)`  
`-- Anwendung:`  
`foo :: Id ; foo = fromString "bar"`
- **der Schreibaufwand wird verringert durch**

```
 -- bei Implementierung:
import Data.String;
instance IsString Id where fromString = T.pack
 -- bei Anwendung:
{-# language OverloadedStrings #-}
foo :: Id ; foo = "bar"
```

**String-Literale sind dann *überladen*  $\Rightarrow$  Compiler setzt `fromString` vor jedes (`"bar"`  $\Rightarrow$  `fromString "bar"`)**

# 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 ausgewertet

# 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
```

# Übung (effiziente Imp. von Bezeichnern)

- welche Operationen auf `Id` werden benötigt?
  - Konstruktion (`fromString`)
  - Gleichheit
  - Ausgabe (nur für Fehlermeldungen!)
- für `newtype Id = Id Text deriving Eq`:  
wie teuer ist Vergleich? wie könnte man das verbessern?
- für `type Env` und `extend` wie angegeben: wie teuer ist das Aufsuchen des Wertes eines Namens in einer Umgebung, die durch  $n$  geschachtelte `extend` entsteht?  
wie könnte man das verbessern?  
Hinweis: mit `Env` als Funktion: gar nicht.  
Welcher andere Typ könnte verwendet werden?

# 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  $\frac{P_1, \dots, P_n}{K}$

- Axiomen (= Regeln ohne Prämissen)

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

- jeder Knoten ist mit einem Objekt beschriftet
- jeder Knoten (mit Vorgängern) entspricht Regel von  $I$
- Wurzel (Ziel) ist mit  $F$  beschriftet

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

# 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:  $\frac{(x, y)}{(x - y, y)}$
- Schema wird *instantiiert* durch Belegung der Schema-Variablen

Bsp: Belegung  $x \mapsto 13, y \mapsto 5$

ergibt Regel  $\frac{(13, 5)}{(8, 5)}$

# Inferenz-Systeme (Beispiel)

- Grundbereich = Zahlenpaare  $\mathbb{Z} \times \mathbb{Z}$
- Axiom:  $\frac{}{(13, 5)}$
- Regel-Schemata:  $\frac{(x, y)}{(x - y, y)}$ ,  $\frac{(x, y)}{(x, y - x)}$
- gilt  $I \vdash (1, 1)$  ?
- Ü: Beziehung zu einem alten Algorithmus (früh im Studium, früh in der Geschichte der Menschheit)

# Primalitäts-Zertifikate

- **Satz:**  $p \in \mathbb{P} \iff \exists g : g \text{ ist primitive Wurzel mod } p:$   
 $[g^0, g^1, g^2, \dots, g^{p-2}]$  ist Permutation von  $[1, 2, \dots, p-1]$   
  
let {p = 7; g = 3}  
in map (`mod` p) \$ take (p-1) \$ iterate (\*g) 1  
 $[1, 3, 2, 6, 4, 5]$
- **Inferenzregel**  $\frac{g_1 : p_1, \dots, g_k : p_k}{g : p}$ ,  
falls  $p-1 = q_1^{e_1} \dots q_k^{e_k}$  und  $\forall i : g^{(p-1)/q_i} \neq 1 \pmod p$
- Vaughan Pratt, *Each Prime has a Succinct Certificate*,  
SIAM J. Comp. 1975
- es folgt  $\mathbb{P} \in \text{NP} \cap \text{co-NP}$ , aber  $\mathbb{P}$  *not known to be in P*
- Agrawal, Kayal, Saxena: *Primes is in P*, 2004

# Inferenzsystem: (aussagenlog.) Resolution

- Grundbereich: disjunktive Klauseln
- Inferenz-Regel: 
$$\frac{p_1 \vee \dots \vee p_i \vee q, \neg q \vee r_1 \vee \dots \vee r_j}{p_1 \vee \dots \vee p_i \vee r_1 \vee \dots \vee r_j}$$
- Beispiel:  $\{p \vee q, \neg q \vee r\} \vdash p \vee r$
- Def. Formel  $F$  folgt aus Formelmenge  $M$ :  $M \models F := \forall b : (\forall G \in M : \text{Wert}(G, b) = 1) \Rightarrow \text{Wert}(F, b) = 1$
- Beziehungen zw. Syntax (Resolution) und Semantik (Folgerung)
  - Resolution ist korrekt:  $(M \vdash F) \Rightarrow (M \models F)$
  - Resolution ist widerlegungsvollständig:  
 $(M \models \emptyset) \Rightarrow (M \vdash \emptyset)$

# Inferenz von Typen

- später implementieren wir das, als statische Analyse im Interpreter/Compiler,

jetzt geben wir nur die Regel an: 
$$\frac{f : T_1 \rightarrow T_2, x : T_1}{fx : T_2}$$

- Bsp. für Verwendung eines Inferenzsystems: Manuel Chakravarty, Gabriele Keller, Simon Peyton Jones, Simon Marlow, *Associated Types with Class*, POPL 2005

Absch. 4.2 (Fig. 2) Grundbereich:  $\Theta | \Gamma \vdash e : \sigma$

means that in type environment  $\Gamma$  and instance environment  $\Theta$  the expression  $e$  has type  $\sigma$

Bsp. für ein Regelschema: 
$$\frac{(v : \sigma) \in \Gamma}{\Theta | \Gamma \vdash v : \sigma} (var)$$

# Inferenz von Werten

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

`data Exp = Const Integer`

`| Plus Exp Exp | Times Exp Exp`

- Axiome:  $\text{wert}(\text{Const } z, z)$

- Regeln:

$$\frac{\text{wert}(X, a), \text{wert}(Y, b)}{\text{wert}(\text{Plus } X \ Y, a + b)},$$

$$\frac{\text{wert}(X, a), \text{wert}(Y, b)}{\text{wert}(\text{Times } X \ Y, a \cdot b)}, \dots$$

- das ist syntaxgesteuerte Semantik:

für jeden Konstruktor von  $p \in \text{Exp}$

gibt es genau eine Regel mit Konklusion  $\text{wert}(p, \dots)$

# Umgebungen (Spezifikation)

- Grundbereich: Aussagen der Form  $\text{wert}(E, p, z)$   
(in Umgebung  $E$  hat Programm  $p$  den Wert  $z$ )

Umgebungen konstruiert aus  $\emptyset$  und  $E[v := b]$

- Regeln für Operatoren  $\frac{\text{wert}(E, X, a), \text{wert}(E, Y, b)}{\text{wert}(E, \text{Plus } XY, a + b)}, \dots$
- Regeln für Umgebungen  $\frac{}{\text{wert}(E[v := b], \text{Ref } v, b)}$   
 $\frac{\text{wert}(E, \text{Ref } v', b')}{\text{wert}(E[v := b], \text{Ref } v', b')}$  für  $v \neq v'$
- Regeln für Bindung:  $\frac{\text{wert}(E, X, b), \text{wert}(E[v := b], Y, c)}{\text{wert}(E, \text{let } v = X \text{ in } Y, c)}$

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

# Aufgaben Inferenz

## 1. Primalitäts-Zertifikate

- welche von 2, 4, 8 sind primitive Wurzel mod 101?
- vollst. Primfaktorzerlegung von 100 angeben
- ein vollst. Prim-Zertifikat für 101 angeben.
- bestimmen Sie  $2^{(101-1)/5} \pmod{101}$  von Hand  
Hinweise: 1. das sind *nicht* 20 Multiplikationen,  
2. es wird *nicht* mit riesengroßen Zahlen gerechnet.

## 2. Geben Sie den vollständigen Ableitungsbaum an für die Auswertung von

```
let {x = 5} in let {y = 7} in x
```

## 3. warum ist aussagenlog. Resolution nicht vollständig? (es

gilt nicht:  $(M \models F) \Rightarrow (M \vdash F)$ .) Ein einfaches Gegenbeispiel reicht.

4. ein Paper aus POPL heraussuchen, das Inferenzsysteme verwendet zur Beschreibung von statischer oder dynamische Semantik einer Programmiersprache

# Semantische Bereiche

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

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

- typische Verarbeitung:

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

# Continuations

- Programmablauf-Abstraktion durch Continuations:

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

$k$  ist die *continuation* (die Fortsetzung im Erfolgsfall)

- eben geschriebenen Code refaktorisieren zu:

```
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

## 1. Bool im Interpreter

- Boolesche Literale
- relationale Operatoren (`==`, `<`, o.ä.),
- Inferenz-Regel(n) für Auswertung des `if`
- Implementierung der Auswertung von `if/then/else` mit `with_bool`,

## 2. Striktheit der Auswertung

- einen Ausdruck  $e :: \text{Exp}$  angeben, für den `value undefined e` eine Exception ist (zwei mögliche Gründe: nicht gebundene Variable, Laufzeit-Typfehler)
- mit diesem Ausdruck: diskutiere Auswertung von `let {x = e} in 42`

### 3. bessere Organisation der Quelltexte

- Cabalisierung (Quelltexte in `src/`, Projektbeschreibungsdatei `cb.cabal`), Anwendung: `cabal repl` usw.
- **separate** Module für `Exp`, `Env`, `Value`,

# 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:

$$\lambda x y z . z \rightarrow B := \lambda x . \rightarrow (\lambda y . \rightarrow (\lambda z . \rightarrow B))$$

- mehrstellige Applikation:

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

(die Applikation ist links-assoziativ)

- der Typ einer mehrstelligen Funktion:

$$T1 \rightarrow T2 \rightarrow T3 \rightarrow T4 := \\ T1 \rightarrow (T2 \rightarrow (T3 \rightarrow T4))$$

(der Typ-Pfeil ist rechts-assoziativ)

# Semantik mit Closures

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

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

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

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

# Closures (Spezifikation)

- Closure konstruieren (Axiom-Schema):

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

- Closure benutzen (Regel-Schema, 3 Prämissen)

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

- 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 selbstgebaudem Interpreter ausrechnen
- mit Haskell ausrechnen
- in JS (node) ausrechnen

# Repräsentation von Fehlern

- Fehler explizit im semantischen Bereich des Interpreters repräsentieren (anstatt als Exception der Gastsprache)

```
data Val = ... | ValErr Text
```

- strikte Semantik: `ValErr` *niemals* in Umgebung (bei Let-Bindung oder UP-Aufruf)
- **Ü**: realisieren durch Aufruf (an geeigneten Stellen) von

```
with_val :: Val -> (Val -> Val) -> Val
with_val v k = case v of
 ValErr _ -> v
 _ -> k v
```

# Übungen

1. 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?

2. alternative Implementierung von Umgebungen

- **bisher** `type Env = Id -> Val`

- **jetzt** `type Env = Data.Map.Map Id Val` oder `Data.HashMap`

Messung der Auswirkungen: 1. Laufzeit eines Testfalls, 2. Laufzeiten einzelner UP-Aufrufe (profiling)



# 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:  $\int x^2 dx$ ,  $\sum_{k=0}^n k^2$
- Logik:  $\forall x \in A : \forall y \in B : P(x, y)$
- Programmierung:  

```
static int foo (int x) { ... }
```

# Der Lambda-Kalkül

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

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

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

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

# Lambda-Terme

- Menge  $\Lambda$  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)

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

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

# Eigenschaften der Reduktion

- $\rightarrow_\beta$  auf  $\Lambda$  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  $\Lambda$  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

- $\lambda$ -Kalkül: Rel.  $\rightarrow_{\beta}$  substituiert Variablen im Term  
schwache Reduktion: wie  $\rightarrow_{\beta}$ , aber niemals unter  $\lambda$   
unser Interpreter: realisiert schwache Reduktion, Regeln  
für  $\text{wert}(E, X, w)$  speichern Substitutionen in Umgebung
- ein Zusammenhang wird hergestellt durch *Kalküle für explizite Substitutionen*,

Pierre-Louis Curien: *An Abstract Framework for Environment Machines*, TCS 82 (1991),

[https://doi.org/10.1016/0304-3975\(91\)90230-Y](https://doi.org/10.1016/0304-3975(91)90230-Y)

Abadi, Cardelli, Curien, Levy: *Explicit Substitutions*, JFP 1991, <https://doi.org/10.1017/S0956796800000186>,

# Daten als Funktionen

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

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

Ü: überprüfen für  $k = 2$

- Anwendungen:
  - Modellierung von Listen, Zahlen
  - Auflösung simultaner Rekursion

# Lambda-Kalkül als universelles Modell

- Wahrheitswerte:

$\text{True} := \lambda xy.x, \text{False} := \lambda xy.y$

- Verzweigung:  $\text{if } b \text{ then } x \text{ else } y := bxy$

- natürliche Zahlen als iterierte Paare (Ansatz)

$(0) := \langle \text{True}, \lambda x.x \rangle; (n + 1) := \langle \text{False}, n \rangle$

- $s_2^2$  ist partielle Vorgänger-Funktion:  $s_2^2(n + 1) = n$

- Verzweigung:  $\text{if } a = 0 \text{ then } x \text{ else } y := s_1^2axy$

- Ü: nachrechnen. Ü: das geht sogar mit  $(0) = \lambda x.x$

- Rekursion?

# Fixpunkt-Kombinatoren (Motivation)

- Beispiel: die Fakultät

$f = \lambda x \rightarrow \text{if } x=0 \text{ then } 1 \text{ else } x * f(x-1)$

erhalten wir als Fixpunkt einer Fkt. 2. Ordnung

$g = \lambda h x \rightarrow \text{if } x=0 \text{ then } 1 \text{ else } x * h(x-1)$

$f = \text{fix } g \quad \text{-- d.h., } f = g f$

- Ü:  $g(\lambda z.z)7$ , Ü:  $\text{fix } g 7$

- Implementierung von  $\text{fix}$  mit Rekursion:

$\text{fix } g = g (\text{fix } g)$

- es geht aber auch *ohne Rekursion*. Ansatz:  $\text{fix} = AA$ ,

dann  $\text{fix } g = AAg = g(AAg) = g(\text{fix } g)$

eine Lösung ist  $A = \lambda xy \dots$

# 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.  $\Theta f$  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

- $c : \mathbb{N} \rightarrow \Lambda : n \mapsto \lambda f x. f^n(x)$   
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  $s = \lambda n f x. f(n f x)$   
1. auf Papier beweisen, 2. mit leancheck prüfen  
benutze `check $ \ (Natural x) -> ...`
- Addition: `plus c_a c_b = c_{a+b}` für `plus = \ a b f x. a f (b f x)`
- implementiere die Multiplikation, beweise, prüfe
- Potenz: `pow c_a c_b = c_{a^b}` für `pow = \ a b. b a`

# Visualisierung und Sonifizierung

- Claude Heiland-Allen: *GULCII, my Graphical Untyped Lambda Calculus Interactive Interpreter*,  
[https://mathr.co.uk/blog/2017-10-26\\_gulcii\\_in\\_edinburgh.html](https://mathr.co.uk/blog/2017-10-26_gulcii_in_edinburgh.html)
- vorgeführt auf FARM 2017 (Workshop on Functional Art, Music, Modeling and Design), Oxford  
<https://functional-art.org/2017/>
- Inhalt:
  - Kodierung von Zahlen nach Church
  - ... Scott (einfacher?, aber mit Rekursion/Fixpunkt)
  - Äquivalenz dieser Kodierungen

# 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:  $\text{Rec } x \ B$  ist Fixpunkt von  $(\lambda x. B)$

- Semantik: 
$$\frac{\text{wert}(E, (\lambda x. B)(\text{Rec } x \ B), v)}{\text{wert}(E, \text{Rec } x \ B, v)}$$

- Ü: verwende `Let` statt `App` (`Abs ..`) ..

- Ü: das Beispiel mit dieser Regel auswerten

# Direkte Realisierung von Tupeln

- bisher: Simulation von Tupeln (Konstruktor, Selektor) durch Funktionen
- jetzt: Realisierung im Interpreter
  - abstrakte Syntax:

```
data Exp = ..
 | Tuple [Exp]
 | Select Integer Exp
```

Index für **Select** ist **Literal** (nicht **Exp**), damit wir später statisch typisieren können
  - **Werte**:

```
data Val = .. | ValTuple [Val]
```
- Anwendung: Tupel realisiert Umgebung (nur Werte, ohne Namen) zur Laufzeit der Zielsprache der Kompilation

# Simultane Rekursion: letrec

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

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

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

- weitere Beispiele:

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

# letrec nach rec

- Teilausdrücke (für jedes  $i$ )

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

äquivalent vereinfachen zu  $t (\backslash n1 .. nk \rightarrow xi)$

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

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

# 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  $\Theta$ )
  - in Javascript (ohne Rekursion, mit  $\Theta$ )
- Kodierung von Wahrheitswerten und Zahlen (nach Church)
  - implementiere Test auf 0: `iszero cn = 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  $F = \lambda xy.FyxF = (\lambda fxy.fyx f)F$ ,

also  $F = \Theta(\lambda fxy.fyx f)$

- (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  $(\rightarrow_\beta \cup \rightarrow_{\bar{\beta}})^*$ )

- 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)

$X = \lambda abcdefghijklmnopqrstuvvxyzr.$

$r(\text{thisisa fixedpoint combinator})$

$$Y = X^{27} = \underbrace{X \dots X}_{27}$$

Zeige, daß  $Y$  ein Fixpunktkombinator ist.

# 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:**  $State \rightarrow (State, Val)$   
(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
```

**Aktion ohne Zustandsänderung**

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

# Auswertung von Ausdrücken

- Ausdrücke (mit Nebenwirkungen):

```
data 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)

⇒ 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



# Monaden (Motivation, Kategorien)

## ... 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  $\text{Obj}_C$  und Morphismen  $\text{Mor}_C$ , jeder Morphismus  $m$  hat als Start ( $S$ ) und Ziel ( $T$ ) ein Objekt, Schreibweise:  $m : S \rightarrow T$  oder  $m \in \text{Mor}_C(S, T)$
- für jedes  $O \in \text{Obj}_C$  gibt es  $\text{id}_O : O \rightarrow O$
- für  $f : S \rightarrow M$  und  $g : M \rightarrow T$  gibt es  $f \circ g : S \rightarrow T$ .  
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:  $\text{Obj}_{\text{Set}} = \text{Mengen}$ ,  $\text{Mor}_{\text{Set}} = \text{totale Funktionen}$
- Grp:  $\text{Obj}_{\text{Grp}} = \text{Gruppen}$ ,  $\text{Mor}_{\text{Grp}} = \text{Homomorphismen}$
- für jede Halbordnung  $(M, \leq)$ :  $\text{Obj} = M$ ,  $\text{Mor} = (\leq)$
- Hask:  $\text{Obj}_{\text{Hask}} = \text{Typen}$ ,  $\text{Mor}_{\text{Hask}} = \text{Funktionen}$

# 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:  $f : A \rightarrow B$  ist *Isomorphie* (kurz: ist iso),

falls es ein  $g : B \rightarrow A$  gibt mit  $f \circ g = \text{id}_A \wedge g \circ f = \text{id}_B$

## weiteres Beispiel

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

# Produkte

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

# Dualität

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

# Aufgaben Kategorien

SS 24 : zur VL KW 22: Aufgaben 1, 2, 3 (b, c)

1. der identische Morphismus jedes Objektes ist eindeutig  
(das bedeutet: falls es  $\text{id}_1, \text{id}_2$  gibt mit  
 $\forall f : \text{id}_k \circ f = f = f \circ \text{id}_k$ , dann  $\text{id}_1 = \text{id}_2$ )

2. Definieren Sie den duale Begriff zu Monomorphismus.  
Geben Sie Beispiele und Gegenbeispiele für Monom.  
und duale Monom. in der Kategorie der Mengen an

3. Beweise: für das (kategorisch definierte) Produkt  $P$  von  
endlichen Mengen  $A_1, A_2$  gilt:  $|P| = |A_1 \times A_2|$ .

(a) (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.

(b) (Aufgabe)  $|P| \leq |A_1 \times A_2|$

(c) Geben Sie eine entsprechende Aussage für das duale Produkt endlicher Mengen an und beweisen Sie diese.

4. Kategorie mit gerichteten Graphen als Objekte,

$f : V(G) \rightarrow V(H)$  ist Morphismus, falls

$\forall x, y : (x, y) \in E(G) \iff (f(x), f(y)) \in E(H)$ .

Was bedeuten dort mono, epi?

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

# Monaden (Definition, Beispiele)

## Funktoren

- Def: Funktor  $F$  von Kategorie  $C$  nach Kategorie  $D$ :
  - Wirkung auf Objekte:  $F_{\text{Obj}} : \text{Obj}(C) \rightarrow \text{Obj}(D)$
  - Wirkung auf Pfeile:  
$$F_{\text{Mor}} : (g : s \rightarrow t) \mapsto (g' : F_{\text{Obj}}(S) \rightarrow F_{\text{Obj}}(T))$$
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 Halbordnungen?
- `class Functor f where`  
`fmap :: (a -> b) -> (f a -> f b)`  
**Beispiele:** `List`, `Maybe`, `Action`

# Die Kleisli-Konstruktion

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

Diese Komposition ist

$$f \gg g = \lambda x \rightarrow (f \ x \ \gg g)$$

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

# Aufgaben Monaden

SS 24: zu KW 22: Aufgaben 1, 3, 4

## 1. (Functor, Monad für Action)

(a) implementieren Sie `instance Functor Action` direkt.

überprüfen Sie die Funktor-Gesetze an Beispielen.

(b) eine indirekte Implementierung ist

```
instance Functor m where fmap f a = a >>=
```

**Beweisen Sie: für jede Monade `m` erfüllt dieses `fmap` die Funktor-Gesetze.**

## 2. die Konstruktorklasse `Applicative` hat unter anderem diese Methoden

```
class Functor f => Applicative f where
```

`pure :: a -> f a`

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

`liftA2 :: (a -> b -> c) -> f a -> f b -> f c`

im folgenden geht es nur um statische Semantik  
(Typkorrektheit), (noch) nicht um dynamische Semantik  
(Axiome)

- (a) Implementieren Sie `<*>` durch `liftA2`
- (b) Implementieren Sie `liftA2` durch `<*>`
- (c) geben Sie eine Implementierung für `liftA2` mit  
Monaden-Operationen an. (Beachte: andersherum geht  
das nicht, denn nicht jedes `Applicative` ist auch  
Monade)
- (d) was ist *the monad of no return*?

### 3. Funktor- und Monadengesetze (mit leancheck in ghci)

```
cabal install --lib leancheck
```

```
import Test.LeanCheck
```

```
import Test.LeanCheck.Function
```

```
-- Kleisli-Komposition:
```

```
import Control.Monad ((>=>))
```

```
:set -XPatternSignatures
```

```
-- return ist rechts neutral
```

```
check $ \ (f :: Bool -> Maybe Bool) x -> (f x == (re
```

```
-- ein Test, der zeigt, daß eine falsche Aussage er
```

```
check $ \ (f :: Bool -> Maybe Bool) x -> (f x == (f
```

## 4. falsche Functor-/Monad-Instanzen für Maybe, List, Tree (d.h. typkorrekt, aber semantisch falsch)

Standard-Datentyp mit `newtype` verpacken

```
newtype List a = List [a]
instance Functor List where
 -- korrekte Instanz für List
 fmap f (List xs) =
 -- benutzt korrekte Instanz für []
 List (fmap f xs)
```

oder strukturgleichen Typ selbst definieren

```
import Prelude hiding (Maybe)
data Maybe a = Nothing | Just a deriving (Eq, Show)
instance Monad Maybe where ...
```

## 5. StateT Int zum Zählen der Auswertungsschritte (Aufrufe von value) einbauen

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

```
type Domain = S.StateT Natural Action Val
```

```
value :: Env Name Val -> Exp -> Domain
```

```
...
```

```
Put e f -> with_addr (value env e) $ \ a ->
 with_value (value env f) $ \ v ->
 T.lift (put a v) >>= \ _ ->
 return ValUnit
```

```
test3 = print $ run state0 $ flip S.runStateT 0 $ v
```

```
$ Let "a" (New $ LitInt 0) ...
```

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







# 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 Verketteten 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:

$$\begin{aligned} (<*>) &:: \text{Parser } c \ (a \rightarrow b) \\ &\rightarrow \text{Parser } c \ a \rightarrow \text{Parser } c \ b \end{aligned}$$

- eines der Resultate wird exportiert, anderes ignoriert

$$\begin{aligned} (<*) &:: \text{Parser } a \rightarrow \text{Parser } b \rightarrow \text{Parser } a \\ (*>) &:: \text{Parser } a \rightarrow \text{Parser } b \rightarrow \text{Parser } b \end{aligned}$$

Eselsbrücke: Ziel des „Pfeiles“ wird benutzt

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

$$(<\$>) :: (a \rightarrow b) \rightarrow \text{Parser } a \rightarrow \text{Parser } b$$

# Kombinator-Parser und Grammatiken

- CFG-Grammatik mit Regeln  $S \rightarrow aSbS, S \rightarrow \epsilon$  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

- `chainl1 :: Parser c a -> Parser c (a -> a -> a) -> Parser c a`  
`chainl1 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 -> chainl1 ... ) atom ops`
- `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:

$$\begin{aligned} (<|>) &:: \text{Parser } c \ a \ -> \text{Parser } c \ a \\ &\quad -> \text{Parser } c \ a \end{aligned}$$
$$\text{Parser } p \ <|> \text{Parser } q = \text{Parser } \$ \ \backslash \ s \ -> \dots$$

- **symmetrisch:**  $p \ s \ ++ \ q \ s$
- **asymmetrisch:**  $\text{if null } p \ s \ \text{then } q \ s \ \text{else } p \ s$

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

# Applicative, aber nicht Monad (Beispiele)

- Roman Cheplyaka, 2015 <https://hackage.haskell.org/package/regex-applicative>

```
data RE s a = ...
instance Applicative (RE s) where ...
-- | Attempt to match a string of symbols
-- against the regular expression:
match :: RE s a -> [s] -> Maybe a
match re = let obj = compile re in \str ->
 listToMaybe $ results $ foldl (flip step) obj s
```

Erläuterung (geraten): regulärer Ausdruck wird in endlichen nicht-det. Automaten übersetzt (`compile re`), *unabhängig* von der Eingabe (`str`) — deswegen keine Monad-Instanz,

Rechnung des dazu äquivalenten det. Automaten wird simuliert (für diese eine Eingabe, ohne den gesamten Potenzmengenautomaten auszurechnen), vgl.

`https://www.imn.htwk-leipzig.de/~waldmann/edu/ss23/afs/folien/#\(61\)`

- **Paolo Capriotti, Huw Campbell, 2012–**

`https://hackage.haskell.org/package/optparse-applicative`

**Anwendungs-Beispiel:** `https://git.imn.`

`htwk-leipzig.de/waldmann/pure-matchbox/-/blob/master/src/Matchbox/Config.hs#L118`

# Übung Kombinator-Parser

1. Bezeichner parsen (alphabetisches Zeichen, dann Folge von alphanumerischen Zeichen),

Hinweis:

```
fmap fromString $ (:) <$> _ <*> many _
```

2. Whitespace ignorieren (verwende `Data.Char.isSpace`)

Hinweis: jeder Token-Parser `t` (Bsp: Bezeichner, Zahl-Literal) wird abgeschlossen durch `t <*> space` (warum nicht `space *> t`?)

3. konkrete Haskell-ähnliche Syntax realisieren

- `if .. then .. else ..`

- `let { .. = .. } in ..`
- Abstraktion (`\ x -> ..`) einstellig
- Applikation (`f a b c`) mehrstellig

Hinweis:

```
exp = foldl App <$> atom <*> many atom
atom = Const <$> number <|> _ <|> parens e
```

- Abstraktion (`\ x y z -> ..`) mehrstellig (im Parser übersetzen in geschachtelte einstellige)

4. Implementierung `chainl1`, `expression` ergänzen,  
Operator-Parser vervollständigen für Ausdrücke mit  
Multiplikation, Addition, Vergleichen

Hinweis: Operatoren sollen schwächer binden als  
Applikation (Bsp: `f a + b` bedeutet `(f a) + b`),

realisieren durch

```
exp = expression _ app
app = foldl App <$> atom ... -- wie oben
atom = ... <|> parens exp -- wie oben
```

## 5. den Typ des Parsers ändern

```
type Parser c = StateT [c] [] -- vorher
type Parser c = StateT [c] Maybe -- nachher
```

und Auswirkung diskutieren.

## 6. Fehlermeldungen mit

```
type Parser c = StateT [c] (Except String)
```

```
char c = satisfy (== c)
 <|> throwError ("expected: " <> [c])
```

7. **den Eigenbau-Parser** `Parser Char` durch

`Text.Parsec.String.Parser` **ersetzen** <https://hackage.haskell.org/package/parsec/>

**Operator-Ausdrücke** dann mit `Text.Parsec.Expr`

# Pretty-Printing

## Motivation

- Node "foo" (Node "baz" Leaf Leaf) (Node "foobaz" Leaf Leaf)  
much easier to read if it is presented as

```
Node "foo" (Node "baz" Leaf Leaf)
 (Node "foobaz" Leaf Leaf)
```

A pretty-printer's job is to lay out structured data appropriately (John Hughes 1995)

- inappropriate wäre `deriving Show`, denn Ausgabe ist dort immer einzeilig
- Anwendung: Code-Ausgabe in Kompilation, (Mensch soll es lesen, hat es aber nicht geschrieben)

# Spezifikation

- `data Doc` abstrakter Dokumententyp für Textblöcke

- **Konstruktor:** `text :: String -> Doc`

- **Kombinatoren:**

`vcat :: [ Doc ] -> Doc -- vertikal`

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

- **Ausgabe:** `render :: Doc -> String (oder Text)`

- der wesentliche Punkt ist

`cat :: [ Doc ] -> Doc`

vertikal *oder* horizontal je nach verfügbarem Platz

- nachvollziehbare Spezifikation, effiziente Implement.

# Implementierung von Hughes, Peyton Jones

- John Hughes: *The Design of a Pretty-printing Library* in *Advanced Functional Programming*, 1995 Johan Jeuring and Erik Meijer (eds), LNCS 925 `https:`

`//www.cse.chalmers.se/~rjmh/Papers/pretty.html`

- verwendet in GHC (Glasgow Haskell Compiler) von Anfang an

Übung: wirklich?

`https://downloads.haskell.org/~ghc/0.29/`

und GHC heute?

- alleinstehende Implementierung

`https://hackage.haskell.org/package/pretty`

# Implementierung von Wadler und Leijen

- Phil Wadler: *A Prettier Printer*, 1997 <https://homepages.inf.ed.ac.uk/wadler/papers/prettier/prettier.pdf>
- Implementierung (Daan Leijen 2000)  
<https://hackage.haskell.org/package/wl-pprint>
- „modern“ (David Luposchajski)  
<https://hackage.haskell.org/package/prettypainter>  
dort auch Vergleich verschiedener Prettyprinter  
vgl. <https://xkcd.com/927/> **Standards**

# Klasse, Generierung, Präzedenzen

- `class Pretty a where pretty :: a -> Doc`

- (z.B. in autotool) automatische Code-Erzeugung für `instance Pretty T` für algebraische Datentypen

[https://git.imn.htwk-leipzig.de/waldmann/autotool/-/blob/master/todoc/src/Autolib/ToDoc/Class.hs?ref\\_type=heads](https://git.imn.htwk-leipzig.de/waldmann/autotool/-/blob/master/todoc/src/Autolib/ToDoc/Class.hs?ref_type=heads)

- Klammern weglassen (in Operator-Ausdrücken) durch Berücksichtigung von Präzedenzen

```
class Pretty a where pretty_prec :: Int ->
```

# 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?

# Ablaufsteuerung/Continuations

## Definition

(alles nach: Turbak/Gifford Ch. 17.9)

CPS-Transformation (continuation passing style):

- original: Funktion gibt Wert  $v$  zurück

$$f = \lambda x y \rightarrow \text{let } \{ \dots \} \text{ in } v$$

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

$$f\_cps = \lambda x y k \rightarrow \text{let } \{ \dots \} \text{ in } k v$$

aus  $g (f 3 2)$

wird  $\lambda k \rightarrow f\_cps 3 2 \$ \lambda v \rightarrow g\_cps v k$

# Motivation

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

CPS als einheitlicher Mechanismus für

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

# CPS für Linearisierung

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

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

$\text{plus } x \ y \ k = k \ (x + y)$

$\text{mal } x \ y \ k = k \ (x * y)$

später tatsächlich als Programmtransformation  
(Kompilation)

# CPS für Resultat-Tupel

wie modelliert man Funktion mit mehreren Rückgabewerten?

- benutze Datentyp Tupel (Paar):

$$f : A \rightarrow (B, C)$$

- benutze Continuation:

$$f/cps : A \rightarrow (B \rightarrow C \rightarrow D) \rightarrow D$$

# 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)

$$\begin{aligned} \text{value}(E, \text{label } L \ B) &= \lambda k \rightarrow \\ &\quad \text{value}(E[L:=k], B) \ k \\ \text{value}(E, \text{jump } L \ B) &= \lambda k \rightarrow \\ &\quad \text{value}(E, L) \ \$ \ \lambda k' \rightarrow \\ &\quad \text{value}(E, B) \ k' \end{aligned}$$

**Beispiel 1:**

$$\begin{aligned} \text{value}(E, \text{label } x \ x) & \\ &= \lambda k \rightarrow \text{value}(E[x:=k], x) \ k \\ &= \lambda k \rightarrow k \ k \end{aligned}$$

## 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 ->
 (\ k0 -> k0 k0) $ \ k' ->
 value (E, label y y) k'
= \ k ->
 (\ k0 -> k0 k0) $ \ k' ->
 (\ k1 -> k1 k1) k'
```

# Semantik

semantischer Bereich:

```
type Continuation a = a -> Action Val
```

```
newtype CPS a
```

```
 = CPS (Continuation a -> Action Val)
```

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

Plan:

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

# CPS als Monade

- ein CPS-transformiertes Programm ausführen

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

- Spezifikation der Monaden-Operationen

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

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

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

# Der CPS-Transformator

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

# Übungsaufgaben

Rekursion (bzw. Schleifen) mittels Label/Jump  
(und ohne Rec oder Fixpunkt-Kombinator)

folgende Beispiele sind aus Turbak/Gifford, DCPL, 9.4.2

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

```
let { d = \ f -> \ x -> f (f x) }
in let { f = label 1 (\ x -> jump 1 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
 } in if 0 == n then a
 else loop (triple loop
 } in f 5

```







# 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  $T_1 \rightarrow T_2$
    - \* Verweistyp  $\text{Addr } T$
    - \* Tupel  $(T_1, \dots, T_n)$ , einschl.  $n = 0$
- Umgebung bildet Namen auf Typen ab

# Inferenzsystem für Typen (Semantik)

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

hierbei (vorläufige) Design-Entscheidungen:

- Typ eines Ausdrucks wird inferiert
- Typ eines Bezeichners wird  $\dots$ 
  - in Abstraktion: deklariert
  - in Let: inferiert

# Inferenz für Let

(alles ganz analog zu Auswertung von Ausdrücken)

- Regeln für Umgebungen

- $E[v := t] \vdash v : t$

- $$\frac{E \vdash v' : t'}{E[v := t] \vdash v' : t'} \text{ für } v \neq v'$$

- Regeln für Bindung:

$$\frac{E \vdash X : s, \quad E[v := s] \vdash Y : t}{E \vdash \text{let } v = X \text{ in } Y : t}$$

# Applikation und Abstraktion

- Applikation: 
$$\frac{E \vdash F : T_1 \rightarrow T_2, \quad E \vdash A : T_1}{E \vdash (FA) : T_2}$$

vergleiche mit *modus ponens*

- Abstraktion (mit deklariertem Typ der Variablen)

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

dazu Erweiterung der abstrakten Syntax

- basiert auf: Alonzo Church: Simple Theory of Types, J. Symb. Logic 1940

# Beziehung zw. dyn. und stat. Semantik

- Beschreibung der dynamische Semantik als:
  - big-step: Relation zw. Programm (Term) und Wert (Normalform), realisiert durch Inferenzbaum (äq.: durch Aufruf von `value`)
  - small-step  $\xrightarrow{S}$ : Schritt bei Durchquerung des Inferenzbaums (äq.: bei Folge der Aufrufe von `value`, repräsentiert Folge von Beta-Reduktionsschritten)
- Eigenschaften (Beziehung small-step zu Typisierung)
  - Sicherheit:  $e_0 : T$  und  $e_0 \xrightarrow{S} e_1 \Rightarrow e_1 : T$ .
  - Fortschritt:  $e_0 : T$  und  $e_0$  kein Wert  $\Rightarrow \exists e_1 : e_0 \xrightarrow{S} e_1$ .
  - Termination:  $e_0 : T \Rightarrow$  Folge  $e_0 \xrightarrow{S} e_1 \xrightarrow{S} \dots$  ist endlich
- *well-typed programs don't go wrong*  
(Robin Milner 1978, über polymorphe Typisierung)

# Grenzen der einfachen Typisierung

- nicht jedes Programm hat einen Typ.  
 $(\lambda x.xx)(\lambda x.xx)$  hat keinen Typ  
Ü: versuche die Bestimmung von  $T_1, T_2$  in  
 $(\lambda(x : T_1).xx)(\lambda(x : T_2).xx)$   
... soll auch nicht, denn es terminiert nicht
- nicht jedes terminierende Programm hat einen Typ  
 $(\lambda x.xx)(\lambda y.y)0$
- es gibt ausdrucksstärkere Typsysteme,  
aber wenn gilt  $\forall e : e \text{ hat Typ} \iff e \text{ terminiert}$ ,  
dann ist „ $e$  hat Typ“ nicht entscheidbar,  
weil das Halteproblem nicht entscheidbar ist

# Übung

## 1. Typisierung für

- Vergleichs-Operatoren für Zahlen
- If-Then-Else,
- New, Get, Put — Vorsicht: dann gibt es getypte, aber trotzdem nicht terminierende Programme.

## 2. für Tupel mit abstrakter Syntax

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

- dynamische Semantik,
- statische Semantik (einschl. Prüfung des Index, deswegen nicht `Nth Exp Exp`)
- konkrete Syntax (Parser)

### 3. konkrete Syntax für `AbsTyped`

(dazu Parser für Typ-Ausdrücke, dabei soll  $(->)$  rechts-assoziativ sein)

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

# Inferenzregeln f. Rekonstruktion (Plan)

Plan:

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

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

Inferenzregeln? Lösbarkeit? Implementierung? — Testfall:

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

# Inferenzregeln f. Rekonstruktion

- 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

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

- Abstraktion

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

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

# Implementierung Constraint-Erzeugung

- durch Stapelung von Monaden-Transformatoren

```
type Domain a =
 WriterT [Con] (StateT Int (Except T.Text)) a
```

- `StateT Int` zum Erzeugen *frischer* Unbekannter
- `WriterT [Constraint]` zum Aufsammeln der Constraints
- Programm zur Typisierung bleibt (im wesentlichen), aber  
`if t1 == t2 then return foo else reject ..`  
wird ersetzt durch  
`do tell (Con t1 t2); return foo`
- damit `Except` beim Erzeugen der Constraints unnötig  
(aber beim Lösen der Constraints)

# Substitutionen (Definition)

- Signatur  $\Sigma = \Sigma_0 \cup \dots \cup \Sigma_k$ ,
- $\text{Term}(\Sigma, V)$  ist kleinste Menge  $T$  mit  $V \subseteq T$  und  $\forall 0 \leq i \leq k, f \in \Sigma_i, t_1 \in T, \dots, t_i \in T : f(t_1, \dots, t_i) \in T$ .  
(hier Anwendung für Terme, die Typen beschreiben)
- Substitution: partielle Abbildung  $\sigma : V \rightarrow \text{Term}(\Sigma, V)$ ,  
Definitionsbereich:  $\text{dom } \sigma$ , Bildbereich:  $\text{img } \sigma$ .
- Substitution  $\sigma$  auf Term  $t$  anwenden:  $t\sigma$
- $\sigma$  heißt *pur*, wenn kein  $v \in \text{dom } \sigma$  als Teilterm in  $\text{img } \sigma$  vorkommt.

Beispiele (pur oder nicht?)

$\{(X, f(Y)), (Y, a)\}, \{(X, f(a)), (Y, a)\}, \{(X, Y), (Y, f(X))\}$

# 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:

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

Implementierung:

```
import Data.Map
```

```
type Substitution = Map Identifier Term
```

```
times :: Substitution -> Substitution -> Sub
```

# Substitutionen: Ordnung

Substitution  $\sigma_1$  ist *allgemeiner als* Substitution  $\sigma_2$ :

$$\sigma_1 \lesssim \sigma_2 \iff \exists \tau : \sigma_1 \circ \tau = \sigma_2$$

Beispiele:

- $\{X \mapsto Y\} \lesssim \{X \mapsto a, Y \mapsto a\}$ ,
- $\{X \mapsto Y\} \lesssim \{Y \mapsto X\}$ ,
- $\{Y \mapsto X\} \lesssim \{X \mapsto Y\}$ .

Eigenschaften

- Relation  $\lesssim$  ist Prä-Ordnung ( $\dots$ ,  $\dots$ , aber nicht  $\dots$ )
- Die durch  $\lesssim$  erzeugte Äquivalenzrelation ist die  $\dots$

# Unifikation—Definition

## Unifikationsproblem

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

(allgemeinst: infimum bzgl.  $\lesssim$ )

Satz: jedes Unifikationsproblem ist

- entweder gar nicht
- oder bis auf Umbenennung eindeutig lösbar.

# Unifikation—Algorithmus

- Typ

`mg_u` :: Term -> Term -> Maybe Substitution  
-- oder ... -> Except Text Substitution

- Implementierung:  $\text{mg}_u(s, t)$  nach Fallunterscheidung

- $s$  ist Variable: ...
- $t$  ist Variable: symmetrisch
- $s = (s_1 \rightarrow s_2)$  und  $t = (t_1 \rightarrow t_2)$ : ...
- für andere zusammengesetzte Typen (Tupel, Verweis) entsprechend

# 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  
 $f(f(X_2, X_2), f(f(X_3, X_3), f(f(X_4, X_4), f(a, a))))$

# Hausaufgaben

- Typrekonstruktion: Quelltexte zur Berechnung von Typ und Constraints ergänzen. (Quelltexte zur Lösung der Constraints werden noch gegeben.)

Insbesondere: die alte Implementierung

```
with_int
 :: Domain Val -> (() -> Domain Val) ->
with_int a f = a >>= \ v -> case v of
 Type.Int -> f ()
 _ -> reject "Integer expected"
```

ist jetzt falsch, man darf hier nicht ablehnen, sondern muß ein Constraint ausgeben.

- diskutiere Typisierung (evtl. Typrekonstruktion) für Continuations.

Vergleiche mit Typ von `callCC` aus `transformers` (dieser Type ist aber polymorph, wir sind hier monomorph)

- (noch von früher) Typisierung für Tupel
- konkrete Syntax (Parser, Prettyprinter)

# 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 -> I
 (x :: Int -> Int) -> f (f x)
 ; t1 = \ (f :: Int -> Int) (x :: Int) ->
 ; 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  $\text{Pi Name Type in data Type}$ :  
 $t \in \text{Var} \wedge T \in \text{Typ} \Rightarrow \Pi t.T \in \text{Typ}$ ,  
dabei kann  $t$  in  $T$  vorkommen (Konstruktor  $\text{Ref Name}$ )
- neue Konstrukteure in  $\text{data Expr}$ , mit Inferenz-Regeln:
  - Typ-Abstraktion ( $\text{TypeAbs Name Expr}$ )  
$$\frac{E \vdash X : T_1}{E \vdash \lambda(t :: *) \rightarrow X : \Pi t.T_1}$$
erzeugt parametrischen Typ
  - Typ-Applikation ( $\text{TypeApp Expr Type}$ )  
$$\frac{E \vdash F : \Pi t.T_1}{E \vdash F @T_2 : T_1[t := T_2]}$$
instantiiert param. Typ  
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,

übliche Einschränkung (ML, Haskell): Typ-Abstraktionen nur für let-gebundene Bezeichner:

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

```
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)
```

# Diskussion: Variablen

- wir haben Daten-Variablen (DV) und Typ-Variablen (TV)
- werden unterschiedlich behandelt in Typisierung:
  - Typ einer DV: entsteht durch Deklaration in Abstraktion oder durch Inferenz in Let, wird realisiert durch Umgebung,
  - Belegung einer TV: entsteht durch Typ-Applikation, wird realisiert durch Substitution.
- das ist: 1. nicht uniform, 2. nicht allgemein genug: wir können damit nur über Typen abstrahieren, nicht über Typkonstruktoren.

# Diskussion: Kinds (in Haskell)

- höhere Kinds, Bsp:

```
import Data.Kind (Type)
data List (a :: Type) = Nil | Cons a (List a)
type List :: * -> *
```

- Kind-Inferenz,

```
data T f a = Node a (f (T f a))
type T :: (* -> *) -> * -> *
```

- Kind-Polymorphie

```
data T f g = C (f (g Bool))
type T :: forall {k}. (k -> *) -> (* -> k) -> *
```

# Diskussion: Agda

- in Haskell strikte Trennung von Daten, Typen, Kinds
- in Agda werden Daten, Typen, Kinds, ... uniform behandelt. <https://agda.readthedocs.io/en/v2.6.4.3-r1/getting-started/what-is-agda.html>

Damit sind auch Funktionen von Daten nach Typen möglich: *dependent types*.

- für zukünftiges *dependent Haskell* ist ein Plan: Datum  $\stackrel{?}{=}$  Type = Kind =...

Richard Eisenberg: *design for dependent types* <https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0378-dependent-type-design.rst>

# Übung Polymorphie

- (Beispiel aus Vorlesung) Der Lambda-Ausdruck  $(\lambda x.xx)(\lambda y.y)0$  terminiert, aber ist nicht monomorph typisierbar, denn bereits der Teilausdruck  $xx$  ist es nicht. Eine Typisierung ist möglich, wenn  $x$  polymorph ist und sein Typ-Argument  $a$  unterschiedlich instantiiert wird:

```
((\x -> x @ (b->b) (x @b)) -- <-- hier wird a belegt
 :: forall b . (forall a . a -> a) -> (b -> b)
)
@Int -- <-- hier wird b belegt
((\y -> y) :: forall c . c -> c)
0
```

Übertragen Sie auf unsere Sprache, falls möglich.

Die lokale Quantifikation erfordert *System F*, in ghc

anschalten durch `-XRank2Types`. Im System Hindley-Milner sind Quantoren global, aber der Typ `forall b a . (a -> a) -> (b -> b)` trifft hier nicht zu. Ein Rank-2-Typ aus der Praxis ist

```
newtype LogicT m a =
 LogicT { unLogicT :: forall r. (a -> m r -> m r
```

daraus folgt dieser Typ von `unLogicT`

```
:: forall {k} (m :: k -> *) a
 . LogicT m a
-> forall (r :: k). (a -> m r -> m r) -- <-- loka
-> m r -> m r
```

<https://hackage.haskell.org/package/logict-0.8.1.0/docs/Control-Monad-Logic.html#t:LogicT>



# 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

# CPS-Transformation

## CPS-Transformation: Spezifikation

(als Schritt im Compiler)

- Eingabe: Ausdruck  $X$ , Ausgabe: Ausdruck  $Y$
- Semantik: Wert von  $X = \text{Wert von } Y(\lambda v.v)$
- Syntax:
  - $X \in \text{Exp}$  (fast) beliebig,
  - $Y \in \text{Exp/CPS}$  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:

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

$\Rightarrow$

```
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  $\lambda$ -calculus*, Th. Comp. Sci. 1(2) 1975, 125–159

[http://dx.doi.org/10.1016/0304-3975\(75\)90017-1](http://dx.doi.org/10.1016/0304-3975(75)90017-1) ,

<http://homepages.inf.ed.ac.uk/gdp/>

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

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

Bsp.  $\text{CPS}(\lambda x.9) = \lambda k_2.k_2(\lambda x.\text{CPS}(9)) = \lambda k_2.k_2(\lambda xk_1.k_19)$ ,

$\text{CPS}((\lambda x.9)8) =$

$\lambda k_4.(\lambda k_2.k_2(\lambda xk_1.k_19))(\lambda f.((\lambda k_3.k_38)(\lambda a.fak_4)))$

Ü: Normalform von  $\text{CPS}((\lambda x.9)8)(\lambda z.z)$

# Realisierung der CPS-Transformation

- $\text{CPS}(X) = \text{CPS}_{\mathcal{K}}(X, \lambda x.x)$  wobei  
$$\text{CPS}_{\mathcal{K}} : \text{Exp} \rightarrow (\text{ExpValue} \rightarrow \text{ExpCPS}) \rightarrow \text{ExpCPS}$$
- $\text{CPS}_{\mathcal{K}}(X, k)$  erzeugt Ausdruck  
$$\text{let } \{ \dots \} \text{ in } \dots \text{ let } \{ v = \dots \} \text{ in } k v$$

geschachtelte `let`, der (letzte) Name  $v$  bezeichnet den Wert von  $X$ , Continuation  $k$  wird auf  $v$  angewendet
- Beispiel:  $\text{CPS}_{\mathcal{K}}(X + Y, k) =$   
$$\text{CPS}_{\mathcal{K}}(X, \lambda a. \text{CPS}_{\mathcal{K}}(Y, \lambda b. \text{let } \{ v = a + b \} \text{ in } k(v)))$$
- dabei werden frische Namen (hier:  $v$ ) benötigt, deswegen  
Rechnung in Zustandsmonade: 
$$\text{CPS}_{\mathcal{K}} : \text{Exp} \rightarrow (\text{ExpValue} \rightarrow \text{Fresh ExpCPS}) \rightarrow \text{Fresh ExpCPS}$$

# 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 {1.0=a+2} in let {a=9} in 1.0

- 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 \rightarrow 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

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 $(\text{Code}, \underbrace{W_1, \dots, W_n}_{\text{Umgebung}})$

```
\ 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 $\{v_1, \dots\}$ = freie Variablen in $(\lambda i_1 \dots i_n \rightarrow b)$

```
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`

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 b_1, \dots, b_k, b keine Abstraktionen gestattet

- Zustandsmonade zur Namenserverzeugung (g_1, g_2, \dots)
- Ausgabemonade (`WriterT`) zum Aufsammeln
- g_1, \dots, 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

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

- Spezifikation: $\text{lift}_x(B)x \rightarrow_{\beta}^* B$

- Implementierung:

$$\text{falls } x \notin \text{Free}(B), \text{ dann } \text{lift}_x(B) = KB;$$

$$\text{sonst } \text{lift}_x(x) = I, \text{lift}_x(FA) = S \text{lift}_x(F) \text{lift}_x(A)$$

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

Registervergabe

Motivation

- (klassische) reale CPU/Rechner hat nur *globalen* Speicher (Register, Hauptspeicher)
- Argumentübergabe (Hauptprogramm \rightarrow Unterprogramm) muß diesen Speicher benutzen
(Rückgabe brauchen wir nicht wegen CPS)
- Zugriff auf Register schneller als auf Hauptspeicher \Rightarrow bevorzugt Register benutzen.

Plan (I)

- Modell: Rechner mit beliebig vielen Registern (R_0, R_1, \dots)
- 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, \dots, R_k)
(genau diese, genau dieser Reihe nach)
 - für primitive Operationen: beliebig
- Transformation: lokale Namen \rightarrow Registernamen

Plan (II)

- Modell: Rechner mit begrenzt vielen realen Registern,
z. B. (R_0, \dots, 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, \dots, R_k stehen)

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/>

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

$f(X_n, f(X_{n-1}, \dots, f(X_0, a) \dots))$ und

$f(f(X_{n-1}, X_{n-1}), f(f(X_{n-2}, X_{n-2}), \dots, f(a, a) \dots))$.

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 == at 1 f )
    then at 2 f
    else jump ...
        (... , ... , ...)
```

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

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

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

Geben Sie ein Beispiel dafür an, daß dadurch der Ausgabertext 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. Testen Sie auch die Kompilation von Ausdrücken, die zur Laufzeit Argumente (Zahlen) von der Kommandozeile bekommen (Zugriff mit `argv`)
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?

7. (fehlende) Speicherverwaltung: unser Laufzeitsystem verwendet `malloc` ohne `free`. Das ist für kurze Programme akzeptabel, aber für länger laufende eventuell nicht.

(a) schreiben Sie ein Programm, das im Interpreter einen Wert hat (weil dort das Haskell-Laufzeitsystem den

Speicher bereinigt), die kompilierte C-Version aber nicht.

Begrenzen Sie die Allokation für das Maschinenprogramm durch `ulimit -m ...`

(Die Allokation für das Haskell-Programm kann mit `+RTS -M... -RTS` beschränkt werden)

(b) ersetzen Sie nun in `runtime.c` jedes `malloc` durch `GC_MALLOC` aus dem Boehm-Collector (Hans-Juergen Boehm, Mark Weiser, SPE 1988)

<https://www.hboehm.info/gc/>

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?

Quellen zu verifizierter Kompilation:

- Compcert (formal verification of realistic compilers usable for critical embedded software)
<https://compcert.org/>
- Nipkow und Klein: *Concrete Semantics* Kap. 8

`http://concrete-semantics.org/` (in
Isabelle/HOL)

- **Wadler: Programming Language Foundations in Agda**

`https://plfa.github.io/`