Compilerbau Vorlesung Wintersemester 2008, 09, 10, 11

Johannes Waldmann, HTWK Leipzig

30. Januar 2012

Einleitung

Beispiel

Eingabe (≈ Java):

```
{ int i;
  float prod;
  float [20] a;
  float [20] b;
  prod = 0;
  i = 1;
  do {
    prod = prod
      + a[i]*b[i];
    i = i+1;
  } while (i <= 20);
}
```

Ausgabe
(Drei-Adress-Code):

```
L1: prod = 0
L3: i = 1
L4: t1 = i * 8
    t2 = a [ t1 ]
    t3 = i * 8
    t4 = b [ t3 ]
    t5 = t2 * t4
    prod = prod + t5
L6: i = i + 1
L5: if i <= 20 goto L4
L2:
```

Inhalt

- Motivation, Hintergründe
- lexikalische und syntaktische Analyse (Kombinator-Parser)
- syntaxgesteuerte Übersetzung (Attributgrammatiken)
- Code-Erzeugung ($+$ Optimierungen)
- statische Typsysteme
- Laufzeitumgebungen

Sprachverarbeitung

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

Compiler und andere Werkzeuge

- Quellprogramm
- Präprozessor $\rightarrow$ modifiziertes Quellprogramm
- Compiler $\rightarrow$ Assemblerprogramm
- Assembler $\rightarrow$ verschieblicher Maschinencode
- Linker, Bibliotheken $\rightarrow$ ausführbares Maschinenprogramm

Phasen eines Compilers

- Zeichenstrom
- lexikalische Analyse $\rightarrow$ Tokenstrom
- syntaktische Analyse $\rightarrow$ Syntaxbaum
- semantische Analyse $\rightarrow$ annotierter Syntaxbaum
- Zwischencode-Erzeugung $\rightarrow$ Zwischencode
- maschinenunabhängige Optimierungen $\rightarrow$ Zwischencode
- Zielcode-Erzeugung $\rightarrow$ Zielcode
- maschinenabhängige Optimierungen $\rightarrow$ Zielcode

Methoden und Modelle

- lexikalische Analyse: reguläre Ausdrücke, endliche Automaten
- syntaktische Analyse: kontextfreie Grammatiken, Kellerautomaten
- semantische Analyse: Attributgrammatiken
- Code-Erzeugung: bei Registerzuordnung: Graphenfärbung

Anwendungen von Techniken des Compilerbaus

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

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', `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/`

Organisation

- pro Woche eine Vorlesung, eine Übung.
- Prüfungszulassung: regelmäßiges und erfolgreiches Bearbeiten von Übungsaufgaben
- Prüfung: Klausur (120 min, keine Hilfsmittel)

Beispiel: Interpreter (I)

arithmetische Ausdrücke:

```
data Exp = Const Integer
         | Plus Exp Exp | Times Exp Exp
    deriving ( Show )
ex1 :: Exp
ex1 = Times ( Plus ( Const 1 ) ( Const 2 ) )
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
```

Beispiel: Interpreter (II)

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 )
value :: Env -> Exp -> Integer
value env x = case x of
    Ref n -> env n
    Let n x b -> value ( \ m ->
      if n == m then value env x else env m )  b
    Const i -> i
    Plus x y -> value env x + value env y
    Times x y -> value env x * value env y
```

Übung (Haskell)

- Wiederholung Haskell

  - Interpreter/Compiler: ghci `http://haskell.org/`
  - Funktionsaufruf nicht `f(a,b,c+d)`, sondern `f a b (c+d)`
  - Konstruktor beginnt mit Großbuchstabe und ist auch eine Funktion

- Wiederholung funktionale Programmierung/Entwurfsmuster

  - rekursiver algebraischer Datentyp (ein Typ, mehrere Konstruktoren)
    (OO: Kompositum, ein Interface, mehrere Klassen)

- rekursive Funktion

- Wiederholung Pattern Matching:
  - beginnt mit `case ... of`, dann Zweige
  - jeder Zweig besteht aus Muster und Folge-Ausdruck
  - falls das Muster paßt, werden die Mustervariablen gebunden und der Folge-Ausdruck auswertet

Übung (Interpreter)

- Benutzung:
  - Beispiel für die Verdeckung von Namen bei geschachtelten Let
  - Beispiel dafür, daß der definierte Name während seiner Definition nicht sichtbar ist
- Erweiterung:

  Verzweigungen mit C-ähnlicher Semantik:

  Bedingung ist arithmetischer Ausdruck, verwende 0 als Falsch und alles andere als Wahr.

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

Umgebungen

Umgebung ist (partielle) Funktion von Name nach Wert

Realisierungen: `type Env = String -> Int`

`import Data.Map ; type Env = Map String Int`

Operationen:

- `empty :: Env` leere Umgebung
- `lookup :: Env -> String -> Int`

  Notation: $e(x)$
- `extend :: Env -> String -> Int -> Env`

  Notation: $e[x/v]$

Spezifikation:

- $e[x/v](x) = v, \quad x \neq y \Rightarrow e[x/v](y) = e(y)$

Inferenz-Systeme

Motivation

- inferieren $=$ ableiten
- Inferenzsystem I, Objekt O,

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

  Auswertung von Programmen, Typisierung von Programmen

Definition

ein *Inferenz-System* I besteht aus

- Regeln (besteht aus Prämissen, Konklusion)

  Schreibweise $\frac{P_1,\ldots,P_n}{K}$

- Axiomen ($=$ Regeln ohne Prämissen)

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

- jeder Knoten ist mit einer Formel beschriftet

- jeder Knoten (mit Vorgängern) entspricht Regel von I

- Wurzel ist mit F beschriftet

Schreibweise: $I \vdash F$

Inferenz-Systeme (Beispiel 1)

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

- Axiom:

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

- Regel-Schemata:

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

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

Inferenz-Systeme (Beispiel 2)

- Grundbereich: Zeichenketten aus $\{0, 1\}^*$
- Axiom:

$$\frac{}{01}$$

- Regel-Schemata (für jedes u, v):

$$\frac{0u, v0}{u1v}, \quad \frac{1u, v1}{u0v}, \quad \frac{u}{\mathrm{reverse}(u)}$$

Leite 11001 ab. Wieviele Wörter der Länge k sind ableitbar?

Inferenz-Systeme (Beispiel 3)

- Grundbereich: endliche Folgen von ganzen Zahlen
- Axiome: jede konstante Folge (Bsp. $[3, 3, 3, 3]$)
- Schlußregeln:
  - swap$_k$: $\dfrac{[\ldots, x_k, x_{k+1}, \ldots]}{[\ldots, x_{k+1} + 1, x_k - 1, \ldots]}$
  - rotate: $\dfrac{[x_1, \ldots, x_n]}{[x_2, \ldots, x_n, x_1]}$

Aufgaben:
- Ableitungen für $[5, 3, 1, 3]$, $[7, 7, 1]$
- jede Folge der Form $[z, 0, \ldots, 0]$ ist ableitbar
- Invarianten, $[5, 3, 3]$ ist nicht ableitbar

praktische Realisierung: `http://www.siteswap.org/` und HTWK-Hochschulsport

Inferenz von Werten

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

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

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

- Regeln:

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

Umgebungen

- Grundbereich: Aussagen der Form $\mathsf{wert}(E, p, z)$

  (in Umgebung E hat Programm p den Wert z)

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

- Regeln für Operatoren $\dfrac{\mathsf{wert}(E, X, a), \mathsf{wert}(E, Y, b)}{\mathsf{wert}(E, \mathtt{Plus} XY, a + b)}, \ldots$

- Regeln für Umgebungen

  $$\frac{}{\mathsf{wert}(E[v := b], v, b)}, \quad \frac{\mathsf{wert}(E, v', b')}{\mathsf{wert}(E[v := b], v', b')} \text{ für } v \neq v'$$

- Regeln für Bindung: $\dfrac{\mathsf{wert}(E, X, b), \mathsf{wert}(E[v := b], Y, c)}{\mathsf{wert}(E, \mathtt{let}\ v = X\ \mathtt{in}\ Y, c)}$

Aussagenlogische Resolution

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

folgendes Inferenzsystem heißt *Resolution*:

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

Resolution (Eigenschaften)

die Formel (Klauselmenge) ist nicht erfüllbar $\iff$ *die leere Klausel ist durch Resolution ableitbar.*

Beweispläne:

- $\Rightarrow$: Gegeben ist die nicht erfüllbare Formel. Gesucht ist eine Ableitung für die leere Klausel. Methode: Induktion nach Anzahl der in der Formel vorkommenden Variablen.

- $\Leftarrow$: Gegeben ist die Ableitung der leeren Klausel. Zu zeigen ist die Nichterfüllbarkeit der Formel. Methode: Induktion nach Höhe des Ableitungsbaumes.

Semantische Bereiche

bisher: Wert eines Ausdrucks ist Zahl.

jetzt erweitern (Motivation: if-then-else mit richtigem Typ):

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

Dann brauchen wir auch

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

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

Continuations

Programmablauf-Abstraktion durch Continuations:

Definition:

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

Benutzung:

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

Aufgabe: if/then/else mit `with_bool`

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 { formal :: Name , body :: Exp }
  | App { rator  :: Exp  , rand :: Exp }
```

konkrete Syntax (Beispiel):

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

Semantik

erweitere den Bereich der Werte:

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

erweitere Interpreter:

```
value :: Env -> Exp -> Val
value env x = case x of
    ...
    Abs { } ->
    App { } ->
```

mit Hilfsfunktion

```
with_fun :: Val -> ...
```

Testfall (1)

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

Closures

bisher:

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

alternativ: die Umgebung von `Abs` in die Zukunft transportieren:

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

Der Lambda-Kalkül

(Alonzo Church, 1936) Syntax:

- Variablen $x, y, \ldots$
- Applikationen xx, $(\lambda x.xx)y$
- Abstraktionen $\lambda x.xx$, $\lambda x.(\lambda y.x)$

Begriffe: freie Variablen (FV), gebundene Variablen (BV),

Small-Step-Semantik des Lambda-Kalküls

Basis-Operation ist $A[x/B]$:
ersetze jedes freie Vorkommen von x in A durch B

- β-Konversion:

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

  wobei FV(B) disjunkt zu BV(A)

- α-Konversion: gebundene Umbenennung

  $(\lambda x.A) \to_\alpha (\lambda y.A[x/y])$

…und Abschluß unter Kontext (Inferenz-System?)

Bemerkungen:

- warum ist die Bedingung bei $\to_\beta$ nötig? (Beispiele)

Mehrstellige Funktionen

…simulieren durch einstellige:

- Abstraktion: $\lambda xyz.B := \lambda x.\lambda y.\lambda z.B$

- Applikation: $f PQR := ((f P)Q)R$

weiterer *syntactic sugar*:

```
let { f x =          A } in B
let { f   = \ x -> A } in B
```

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

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 Haskell ausrechnen
- mit selbstgebautem Interpreter ausrechnen

Fixpunkte

Motivation

Das geht bisher gar nicht:

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

(Bezeichner f ist nicht sichtbar)

Lösung:

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

mit neuem AST-Knotentyp `rec`

Rekursion

abstrakt:

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

Semantik von `Rec n b` in Umgebung E

ist der Fixpunkt der Funktion (vom Typ Val $\rightarrow$ Val)

$$\lambda c.\text{Semantik von } b \text{ in } E[n/c]$$

Existenz von Fixpunkten

Fixpunkt von $f :: C \to C$ ist $x :: C$ mit $fx = x$.

Existenz? Eindeutigkeit? Konstruktion?

Satz: Wenn C *pointed CPO* und f *stetig*, dann besitzt f genau einen kleinsten Fixpunkt.

Begriffe:

- CPO $=$ complete partial order $=$ vollständige Halbordnung
- complete $=$ jede monotone Folge besitzt Supremum ($=$ kleinste obere Schranke)
- pointed: C hat kleinstes Element $\perp$
- stetig: $f(\sup \vec{x}) = \sup \overrightarrow{f(x)}$

Dann fix$(f) = \sup[\perp, f(\perp), f^2(\perp), \ldots]$

Funktionen als CPO

- partielle Funktionen $C = (B \to B)$
- Bereich $B \cup \bot$ geordnet durch $\forall x \in B : \bot < x$
- C geordnet durch $f \leq g \iff \forall x \in B : f(x) \leq g(x)$,
- d. h. g ist Verfeinerung von f
- Das Bottom-Element von C ist die überall undefinierte Funktion.

Funktionen als CPO, Beispiel

Wert von

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

ist Fixpunkt der Funktion $F =$

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

Iterative Berechnung des Fixpunktes:

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

Rechnen im Lambda-Kalkül

Daten als Funktionen

Simulation von Daten (Tupel)
durch Funktionen (Lambda-Ausdrücke):

- Konstruktor: $\langle D_1, \ldots, D_k \rangle \Rightarrow \lambda s.sD_1 \ldots D_k$
- Selektoren: $s_i \Rightarrow \lambda t.t(\lambda d_1 \ldots d_k.d_i)$

dann gilt $s_i\langle D_1, \ldots, D_k\rangle \rightarrow_\beta^* D_i$
Anwendungen:

- Auflösung simultaner Rekursion
- Modellierung von Zahlen

Lambda-Kalkül als universelles Modell

- Wahrheitswerte:

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

  (damit läßt sich if-then-else leicht aufschreiben)

- natürliche Zahlen:

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

  (damit kann man leicht $x > 0$ testen)

- Rekursion?

Fixpunkt-Kombinatoren

- Definition: $Y := \lambda f.((\lambda x.f(xx))(\lambda x.f(xx)))$
- Satz: $Y f$ ist Fixpunkt von f
- d.h. Y ist *Fixpunkt-Kombinator*
- Beweis: vergleiche $(Y f)$ und $f(Y f)$

wir benutzen eine Variante des Y:
$Y' = \lambda f.(\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy)))$,
weil sonst die Aufrufe (xx) nicht halten würden.
Ü: weitere Fixpunktkombinatoren,
$\Theta = (\lambda xy.(y(xxy)))(\lambda xy.(y(xxy)))$

Lambda-Berechenbarkeit

Satz: (Church, Turing)

Menge der Turing-berechenbaren Funktionen

(Zahlen als Wörter auf Band)

$=$ Menge der while-berechenbaren Funktionen

(Zahlen als Registerinhalte)

$=$ Menge der Lambda-berechenbaren Funktionen

(Zahlen als Lambda-Ausdrücke)

Übung Fixpunkte

- Fixpunkt der Folge $F^k(\perp)$ für

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

- Fixpunkt der Folge $F^k(\perp)$ für

```
F h = \ x -> if x > 10 then x + 11
                 else h (2 * x - 8)
```

- Turing-Fixpunkt-Kombinator mit Lambda-Calculator

```
http:
//joerg.endrullis.de/lambdaCalculator/
```

- Tupel-Konstruktor und Selektoren als Ausdrücke im Interpreter
- gegenseitige Rekursion (f, g) als Fixpunkt (`Rec`) einer geeigneten Funktion (benutzt Tupel)
- Hausaufgabe: vorige Aufgabe (im Interpreter) ohne `Rec`, sondern mit einem Fixpunkt-Kombinator

letrec

Beispiel (aus: D. Hofstadter, GEB)

```
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 $f(x) \neq g(x)$?

AST-Knoten:

```
data Exp = ... | LetRec [(Name, Exp)] Exp
```

weitere Beispiele:

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

letrec nach rec

mithilfe der Lambda-Ausdrücke für select und tuple

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

Zustand/Speicher

Motivation

bisherige Programme sind nebenwirkungsfrei, das ist nicht immer erwünscht:

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

Dazu muß semantischer Bereich geändert werden.

- bisher: `Val`, jetzt: `State -> (State, Val)`

Semantik von (Teil-)Programmen ist Zustandsänderung.

Speicher

Modellierung:

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

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

Auswertung von Ausdrücken

Ausdrücke (mit Nebenwirkungen):

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

Resultattyp des Interpreters ändern:

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

semantischen Bereich erweitern:

```
data Val = ...
    | ValAddr Addr
    | ValFun ( Val -> Action Val )
```

Aufruf des Interpreters:

```
run Store.empty $ evaluate undefined $ ...
```

Änderung der Hilfsfunktionen

bisher:

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

jetzt:

```
with_int :: Action Val
  -> ( Int -> Action Val ) -> Action Val
with_int m k = m >>= \ v -> case v of ...
```

Hauptprogramm muß kaum geändert werden (!)

Speicher-Aktionen als Monade

generische Aktionen/Verknüpfungen:

• nichts tun (return), • nacheinander (bind, `>>=`)

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a
        -> (a -> m b) -- Continuation
        -> m b
instance Monad Action where
  return x = Action $ \ s -> ( s, x )
  Action a >>= f = Action $ \ s -> ...
```

Rekursion

. . . wird benötigt für Wiederholungen (Schleifen).

mehrere Möglichkeiten zur Realisierung

- mit Fixpunkt-Kombinator
- semantisch (in der Gastsprache des Interpreters)
- (neu:) operational unter Benutzung des Speichers

Rekursion (operational)

Idee: eine Speicherstelle anlegen und als Vorwärtsreferenz auf das Resultat der Rekursion benutzen

```
evaluate env x = case x of ...
  Rec n ( Abs x b ) ->
    new ( ValErr "Rec" ) >>= \ a ->
    with ( evaluate
             ( extend env n ... )
             ( Abs x ... ) ) $ \ v ->
    put a v >>= \ () ->
    return v
```

Rekursion (semantisch)

bisher:

```
fix :: ( a -> a ) -> a
fix f = f ( fix f )
```

jetzt:

```
import Control.Monad.Fix
class MonadFix m where
    mfix :: ( a -> m a ) -> m a

instance MonadFix Action where
mfix f = Action $ \ s0 ->
    let Action a = f v
        ( s1, v ) = a s0
    in  ( s1, v )
```

Speicher—Übung

Code aus Vorlesung:

```
http://dfa.imn.htwk-leipzig.de/cgi-bin/
gitweb.cgi?p=ws11-cb.git
git clone git://dfa.imn.htwk-leipzig.de/srv/
```

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

Die Konstruktorklasse Monad

Definition:

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

Benutzung der Methoden:

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

Do-Notation für Monaden

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

do-Notation (explizit geklammert):

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

do-Notation (implizit geklammert):

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

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

Die Zustands-Monade

Implementierung:

```
data State s a = State ( s -> ( s, a ))
instance Monad ( State s ) where ...
```

Benutzung:

```
import Control.Monad.State

tick :: State Integer ()
tick = do c <- get ; put $ c + 1

evalState ( do tick ; tick ; get ) 0
```

List als Monade

```
instance Monad [] where
    return = \ x - > [x]
    m >>= f = case m of
        []       -> []
        x : xs -> f x ++ ( xs >>= f )
do a <- [ 1 .. 4 ]
   b <- [ 2 .. 3 ]
   return ( a * b )
```

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
```

Elementare Parser (I)

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

Monadisches Verketten von Parsern

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

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

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

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 = do
    c <- satisfy Data.Char.isDigit
    return $ fromIntegral
           $ fromEnum c - fromEnum '0'
```

Kombinatoren für Parser (I)

- Folge (and then) (ist `>>=` aus der Monade)
- Auswahl (or)

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

- Wiederholung (beliebig viele)

```
many, many1 :: Parser c a -> Parser c [a]
many p = many1 p <|> return []
many1 p = do x <- p; xs <- many p; return $ x : xs
```

```
zahl :: Parser Char Integer = do
   zs <- many1 ziffer
   return $ foldl ( \ a z -> 10*a+z ) 0 zs
```

Kombinator-Parser und Grammatiken

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

```
s :: Parser Char ()
s =   do { expect 'a' ; s ; expect 'b' ; s }
  <|> return ()
```

Anwendung: `exec "abab" $ do s ; eof`

Robuste Parser-Bibliotheken

Designfragen:

- asymmetrisches `<|>`
- Nichtdeterminismus einschränken
- Fehlermeldungen (Quelltextposition)

Beispiel: Parsec (Autor: Daan Leijen)

`http://www.haskell.org/haskellwiki/Parsec`

Asymmetrische Komposition

gemeinsam:

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

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

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

Nichtdeterminismus einschränken

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

Festlegung (in Parsec): wenn `p` wenigstens ein Zeichen verbraucht, dann wird `q` nicht benutzt (d. h. `p` muß erfolgreich sein)

Backtracking dann nur durch `try p <|> q`

Fehlermeldungen

- Fehler $=$ Position im Eingabestrom, bei der es „nicht weitergeht“

- und auch durch Backtracking keine Fortsetzung gefunden wird

- Fehlermeldung enthält:
  - Position
  - Inhalt (Zeichen) der Position
  - Menge der Zeichen mit Fortsetzung

Pretty-Printing (I)

John Hughes's and Simon Peyton Jones's Pretty Printer Combinators

Based on *The Design of a Pretty-printing Library* in *Advanced Functional Programming*, Johan Jeuring and Erik Meijer (eds), LNCS 925

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

Pretty-Printing (II)

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

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

- Kombinatoren:

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

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

Ablaufsteuerung/Continuations

Definition

(alles nach: Turbak/Gifford Ch. 17.9)

CPS-Transformation (continuation passing style):

- original: Funktion gibt Wert zurück

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

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

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

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

Motivation

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

CPS als einheitlicher Mechanismus für

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

CPS für Linearisierung

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

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

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

später tatsächlich als Programmtransformation (Kompilation)

CPS für Resultat-Tupel

wie modelliert man Funktion mit mehreren Rückgabewerten?

- benutze Datentyp Tupel (Paar):

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

- benutze Continuation:

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

CPS/Tupel-Beispiel

erweiterter Euklidischer Algorithmus:

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

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

vervollständige, übersetze in CPS

CPS für Ablaufsteuerung

Beispiel label/jump

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

Vergleiche:

- `label <name>` deklariert Exception-Handler
- `jump <name>` springt zum Handler

Semantik für CPS

Semantik von Ausdruck `x` in Umgebung E

ist Funktion von Continuation nach Wert (Action)

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

Beispiel 1:

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

Beispiel 2

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

Semantik

semantischer Bereich:

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

Plan:

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

CPS als Monade

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

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

feed ( return x ) c = c x

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

Übung CPS

- Parser für `Label String Exp`, `Jump Exp Exp`
- Continuations als Werte (von Argumenten und Resultaten von Unterprogrammen)
- Rekursion (bzw. Schleifen) mittels Label/Jump (und ohne Rec oder Fixpunkt-Kombinator)
- `jump (label x x) (label y y)`

Typen

Grundlagen

Typ $=$ statische Semantik

(Information über mögliches Programm-Verhalten, erhalten ohne Programm-Ausführung)

formale Beschreibung:

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

Inferenzsystem für Typen (Syntax)

- Grundbereich: Aussagen der Form $E \vdash X : T$

  (in Umgebung E hat Ausdruck X den Typ T)

- Menge der Typen:
  - primitiv: Int, Bool
  - zusammengesetzt:
    * Funktion $T_1 \to T_2$
    * Verweistyp $\mathrm{Ref}\, T$
    * Tupel $(T_1, \ldots, T_n)$, einschl. $n = 0$

- Umgebung bildet Namen auf Typen ab

Inferenzsystem für Typen (Semantik)

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

hierbei (vorläufige) Design-Entscheidungen:

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

Inferenz für Let

(alles ganz analog zu Auswertung von Ausdrücken)

- Regeln für Umgebungen
  - $E[v := t] \vdash v : t$
  - $\dfrac{E \vdash v' : t'}{E[v := t] \vdash v' : t'}$ für $v \neq v'$
- Regeln für Bindung:

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

Applikation und Abstraktion

- Applikation:

$$\frac{E \vdash F : T_1 \to T_2, \quad E \vdash A : T_1}{E \vdash (F A) : T_2}$$

vergleiche mit *modus ponens*

- Abstraktion (mit deklariertem Typ der Variablen)

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

Eigenschaften des Typsystems

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

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

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

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

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

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 :: Type )
         ->  \ ( f :: t -> t ) ->
                \ ( x :: t ) ->
                       f ( f x )
    ; s = \ ( x :: Int ) -> x + 1
    }
in  (((t [Int -> Int]) (t [Int])) s) 0
```

zur Laufzeit werden die Abstraktionen und Typ-Applikationen *ignoriert*

Typ-Argumente (Regeln)

neuer Typ-Ausdruck $\forall t.T$, Inferenz-Regeln:

- Typ-Abstraktion: erzeugt parametrischen Typ

$$\frac{E \vdash \ldots}{E \vdash \lambda(t :: \text{Type} \to X : \ldots}$$

- Typ-Applikation: instantiiert param. Typ

$$\frac{E \vdash F : \ldots}{E \vdash F[T_2] : \ldots}$$

Ü: Vergleich Typ-Applikation mit expliziter Instantiierung von polymorphen Methoden in C#

Inferenz allgemeingültige Formeln

Grundbereich: aussagenlogische Formeln (mit Variablen und Implikation)

Axiom-Schemata:

$$\frac{}{X \to (Y \to X)}, \frac{}{(X \to (Y \to Z)) \to ((X \to Y) \to (X \to Z))}$$

Regel-Schema (*modus ponens*): $\frac{X \to Y, X}{Y}$

Beobachtungen/Fragen:

- Übung (autotool): Leite $p \to p$ ab.
- (*Korrektheit*): jede ableitbare Formel ist allgemeingültig
- (*Vollständigkeit*): sind alle allgemeingültigen Formeln (in dieser Signatur) ableitbar?

Typen und Daten

- bisher: Funktionen von Daten nach Daten

  `\ (x :: Int) -> x + 1`

- heute: Funktionen von Typ nach Daten

  `\ (t :: Type) -> \ (x :: t) -> x`

- Funktionen von Typ nach Typ (ML, Haskell, Java, C#)

  `\ (t :: Type) -> List t`

- Funktionen von Daten nach Typ (*dependent types*)

  `\ (t :: Typ) (n :: Int) -> Array t n`

  Sprachen: Cayenne, Coq, Agda

  Eigenschaften: Typkorrektheit i. A. nicht entscheidbar, d. h. Programmierer muß Beweis hinschreiben.

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!

(zunächst für einfaches (nicht polymorphes) Typsystem)

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 Variablen
- Constraint: Paar von Typen (T_1, T_2)
- Lösung eines Constraints: Substitution σ mit $T_1\sigma = T_2\sigma$

Inferenzregeln f. Rekonstruktion (Plan)

Plan:

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

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

Inferenzregeln? Implementierung? — Testfall:

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

Inferenzregeln f. Rekonstrukion

- primitive Operationen (Beispiel)

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

- Applikation

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

- Abstraktion

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

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

Substitutionen (Definition)

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

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

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

Substitutionen: Produkt

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

Beispiel 1:

$\sigma_1 = \{X \mapsto Y\}, \sigma_2 = \{Y \mapsto a\}, \sigma_1 \circ \sigma_2 = \{X \mapsto a, Y \mapsto a\}$.

Beispiel 2 (nachrechnen!):

$\sigma_1 = \{X \mapsto Y\}, \sigma_2 = \{Y \mapsto X\}, \sigma_1 \circ \sigma_2 = \sigma_2$

Eigenschaften:

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

Implementierung:

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

Substitutionen: Ordnung

Substitution σ_1 ist *allgemeiner als* Substitution σ_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 (. . . , . . . , aber nicht . . .)
- Die durch $\lesssim$ erzeugte Äquivalenzrelation ist die . . .

Unifikation—Definition

Unifikationsproblem

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

(allgemeinst: infimum bzgl. $\lesssim$)

Satz: jedes Unifikationsproblem ist

- entweder gar nicht
- oder bis auf Umbenennung eindeutig

lösbar.

Unifikation—Algorithmus

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

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

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

Unifikation—Komplexität

Bemerkungen:

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

Rekonstruktion polymorpher Typen

…ist im Allgemeinen nicht möglich:

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

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

Typ-Abstraktionen nur für let-gebundene Bezeichner:

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

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

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

Implementierung

let-Polymorphie, Hindley/Damas/Milner

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

Beispiel

```
let { c = ... }
in let { g = \ f x -> f (if b c x) } in ..
```

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

CPS-Transformation: Zielsyntax

```
Exp_CPS ==> App Id Exp_Value^*
    | If Exp_Value Exp_CPS Exp_CPS
    | Let Id Exp_Letable Exp_CPS

Exp_Value ==> Literal | Identifier

Exp_Letable ==> Literal
    | Abs Id^* Exp_CPS
    | Exp_Value Op Exp_Value
```

Übung: Übersetze

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

Beispiel

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

Transformation f. Applikation

```
CPS[ (app f a1 ... an) ] =
(abs (k)
   (app CPS[f] (abs (i_0)
      (app CPS[a1] (abs (i_1)
         ...
            (app CPS[an] (abs (i_n)
               (app i_0 i_1 ... i_n k)))...)))))
```

dabei sind `k, i_0, .. i_n` *frische* Namen (= die im gesamten Ausdruck nicht vorkommen)

Ü: ähnlich für Primop (Unterschied?)

Transformation f. Abstraktion

```
CPS[ (abs (i_1 ... i_n) b) ] =
(abs (k)
   (let ((i (abs (i_1 .. i_n c)
                 (app CPS[b] c))))
     (app k i)))
```

Ü: Transformation für `let`

Namen

Bei der Übersetzung werden „frische“ Variablennamen benötigt (= die im Eingangsprogramm nicht vorkommen).

```
import Control.Monad.State
data State s a = State ( s -> ( a, s ) )
get :: State s s ; put :: s -> State ()

fresh :: State Int String
fresh = do
    k <- get ; put (k+1)
    return $ "f." ++ show k

type Transform a = State Int a
cps :: Exp -> Transform Exp
```

Vereinfachungen

um geforderte Syntax (ExpCPS) zu erreichen:

- implicit-let

```
(app (abs (i_1 .. i_n) b) a_1 .. a_n)
==>
(let ((i_1 a_1)) ( .. (let ((i_n a_n)) b)..
```

Umbenennungen von Variablen entfernen:

- copy-prop

```
(let ((i i')) b) ==> b [i:=i']
```

aber kein allgemeines Inlining

Teilweise Auswertung

- Interpreter (bisher): komplette Auswertung

  (Continuations sind Funktionen, werden angewendet)

- CPS-Transformator (heute): gar keine Auswertung,

  (Continuations sind Ausdrücke)

- gemischter Transformator: benutzt sowohl
  - Continuations als Ausdrücke (der Zielsprache)
  - als auch Continuations als Funktionen (der Gastsprache)

  (compile time evaluation, partial evaluation)

Partial Evaluation

- bisher:

```
transform  :: Exp -> State ... Exp
transform x = case x of ...
 ConstInt i -> do
  k<-fresh; return $ Abs k (app (Ref k) x)
```

- jetzt:

```
type Cont = Exp -> State ... Exp
transform
    :: Exp -> ( Cont -> State ... Exp )
transform x k = case x of ...
   ConstInt i -> k x
```

Partial Evaluation (II)

```
CPS[ (app f a1 ... an) ] =
(m-abs (K)
   (m-app CPS[f] (m-abs (i_0)
        ...
          (m-app CPS[an] (m-abs (i_n)
    ???      (app i_0 i_1 ... i_n k)))...)))))
```

ändere letzte Zeile in

```
(let ((i (abs (temp) K[temp])))
      (app i_0 .. i_n i))
```

Erklärung CPS-Transformation

Spezifikation (Vorsicht, enthält Typfehler - welche?):

```
    eval env x              (\ y -> k2 (k1 y))
=?= eval env (cps x k1) k2
```

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 -> Transform Exp
cps :: Exp -> Cont -> Transform Exp
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 (Code, $\underbrace{W_1, \ldots, W_n}_{\text{Umgebung}}$)

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

Closure-Konstruktion?

Komplette Übersetzung des Beispiels?

Transformation

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

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

```
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
- Ausgabe: äquivalentes Programm P',
  bei dem alle Abstraktionen global sind

Motivation: in Maschinencode gibt es nur globale Sprungziele

(CPS-Transformation: Unterprogramme kehren nie zurück
$\Rightarrow$ globale Sprünge)

Realisierung

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

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

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

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

Kombinatorische Logik

Motivation

- Lambda-Kalkül zur Modellierung von Abstraktion und Applikation,

  wesentliches Merkmal: benutzerdefinierte Funktionen mit gebundene Variablen.

- *Kombinatorische Logik* ist ein Berechnungsmodell mit einer kleinen, fixierte Menge von (globalen) Funktionen,

  diese heißen *Kombinatoren*.

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

Beispiele

- vordefinierte Kombinatoren:

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

  Ü: Berechne Normalform von $SKKx$, von $SIIx$

- weitere, z. B.
  $B = \lambda xyz.x(yz), C = \lambda xyz.xzy, J = \lambda xyzw.xy(xwz)$

  Ü: simuliere B und C durch I und J

Systematische Übersetzung

Spezifikation:

- Eingabe: geschlossener Lambda-Ausdruck P
- Ausgabe: äquivalenter Kombinator-Ausdruck $[P]$

  (Applikationen mit S, K, I; sonst keine Variablen und Lambdas)

benutzt $[\lambda x.A] = \mathrm{lift}_x(A)$ mit Spezifikation: $\mathrm{lift}_x(A)x \to^* A$

- $\mathrm{lift}_x(y) =$ falls $x = y$ dann I sonst Ky
- $\mathrm{lift}_x(AB) = S\,\mathrm{lift}_x(A)\,\mathrm{lift}_x(B)$
- $\mathrm{lift}_x(\lambda y.A) = \mathrm{lift}_x(\mathrm{lift}_y(A))$

Beispiele: $\lambda x.xx, \lambda xy.y, \lambda xy.yx$ — Vereinfachungen?

Kombinator-Basen

Def: Eine Menge M von Kombinatoren heißt *Basis*, falls es zu jedem Lambda-Ausdruck einen äquivalenten Ausdruck nur aus Applikationen und Kombinatoren aus M gibt.

Satz: $\{S, K, I\}$ ist Basis.
Satz: $\{S, K\}$ ist Basis. — Beweis? $I = \ldots$
Satz: es gibt eine Basis mit nur einem Element. (Schwer.)

Literatur:

- Henk Barendregt: The Lambda Calculus, its Syntax and Semantics, 1984. `http://www.cs.ru.nl/~henk/`

- Raymond Smullyan: How To Mock a Mockingbird, 1985. `http://www.raymondsmullyan.com/`

Anwendungen

- Übersetzung von λ nach CL

  entspricht Closure-Conversion und Lifting (beides, gleichzeitig)

- CL: feste Menge (Basis) von Kombinatoren,

  CC+L: Kombinatoren hängen vom Programmtext ab

- CL als Programmiersprache: `http://www.madore.org/~david/programs/unlambda/`

Bsp: Berechnung von Fibonacci-Zahlen in Unlambda:

```
```s``s``sii`ki `k.*``s``s`ks
 ``s`k`s`ks``s``s`ks``s`k`s`kr``s`k`sikk
  `k``s`ksk
```
```

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

Plan (II)

- Modell: Rechner mit begrenzt vielen realen Registern,

  z. B. $(R_0, \ldots, R_7)$

- falls diese nicht ausreichen: *register spilling*

  virtuelle Register in Hauptspeicher abbilden

- Hauptspeicher (viel) langsamer als Register:

  möglichst wenig HS-Operationen:

  geeignete Auswahl der Spill-Register nötig

Registerbenutzung

Allgemeine Form der Programme:

```
(let* ((r1 (...))
       (r2 (...))
       (r3 (...)))
       ...
  (r4 ...))
```

für jeden Zeitpunkt ausrechnen: Menge der *freien* Register ($=$ deren aktueller Wert nicht (mehr) benötigt wird)

nächstes Zuweisungsziel ist niedrigstes freies Register (andere Varianten sind denkbar)

vor jedem UP-Aufruf: *register shuffle* (damit die Argumente in $R_0, \ldots, R_k$ stehen)

Automatische Speicherverwaltung

Motivation

Speicher-Allokation durch Konstruktion von

- Zellen,Tupel, Closures

Modell: Speicherbelegung $=$ gerichteter Graph

Knoten *lebendig*: von Register aus erreichbar.

sonst tot $\Rightarrow$ automatisch freigeben

Gliederung:

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

Mark/Sweep

Plan: wenn Speicher voll, dann:

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

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

Lösung:

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

temporäre Änderungen im Graphen selbst (pointer reversal)

Pointer Reversal (Invariante)

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

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

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

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

Pointer Reversal (Ablauf)

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

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

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

Knoten werden in Tiefensuch-Reihenfolge betreten.

Eigenschaften Mark/Sweep

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

Ablegen von Markierungs-Bits:

- in Zeigern/Zellen selbst

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

Stop-and-copy (Plan)

Plan:

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

auch hier: Verwaltung ohne Zusatzspeicher (Stack)

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

Stop-and-copy (Invariante)

fromspace, tospace : array [0 . . . N] of cell

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

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

Invarianten:

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

Stop-and-copy (Ablauf)

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

Schritt: while scan $<$ free:

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

Besucht Knoten in Reihenfolge einer Breitensuche.

Stop-and-copy (Eigenschaften)

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

Breiten- und Tiefensuche

put (Wurzel(G));
while Speicher nicht leer:
 $u \leftarrow$ get; wenn u nicht markiert:
  markiere u;
  für alle v mit $u \to_G v$: put(v);

dabei ist Speicher (mit Operationen put/get):

- Stack (LIFO) (push/pop) $\Rightarrow$ Tiefensuche,
- Queue (FIFO) (enqueue/dequeue) $\Rightarrow$ Breitensuche.

woran erkennt man, daß eine Knotenreihenfolge eines gerichteten Graphen G bei einer Breiten/Tiefensuche entstanden sein könnte? (wenn man Reihenfolge der Nachfolger eines Knoten jeweils beliebig wählen kann)

Speicher mit Generationen

Beobachtung: es gibt

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

Plan:

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

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

Speicherverwaltung in JVM

Speicheraufteilung:

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

Ablauf

- minor collection (Eden voll):

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

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

  alles nach Survivor 1 ($+$ Tenured)

Speicherverwaltung in JVM (II)

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

Zusammenfassung

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)

Parser

Ansätze:

- eingebettet: Kombinator-Parser (Parsec)
- separat: lex/yacc (flex/bison), javacc, antlr, ...

Unterschiede:

- eingebettet: benutzt Typsystem, Abstraktionsmechanismen, Bibliotheken der Gastsprache (separat: diese Ausdrucksmittel fehlen oder müssen simuliert werden)
- diese Einschränkung der Ausdruckskraft (z. B. LR(k)-Grammatiken) gestattet effizientere Realisierung (z. B. deterministische Kellerautomaten)

Anhang

Prüfungsvorbereitung

- was ist eine Umgebung (`Env`), welche Operationen gehören dazu?
- was ist eine Speicher (`Store`), welche Operationen gehören dazu?
- Gemeinsamkeiten/Unterschiede zw. Env und Store?
- Für $(\lambda x.xx)(\lambda x.xx)$: zeichne den Syntaxbaum, bestimme die Menge der freien und die Menge der gebundenen Variablen. Markiere im Syntaxbaum alle Redexe. Gib die Menge der direkten Nachfolger an (einen Beta-Schritt

ausführen).

- Definiere Beta-Reduktion und Alpha-Konversion im Lambda-Kalkül. Wozu wird Alpha-Konversion benötigt? (Dafür Beispiel angeben.)
- Wie kann man Records (Paare) durch Funktionen simulieren? (Definiere Lambda-Ausdrücke für `pair, first, second`)
- welche semantischen Bereiche wurden in den Interpretern benutzt? (definieren Sie `Val, Action Val, CPS Val`)
- welches sind die jeweils hinzukommenden Ausdrucksmöglichkeiten der Quellsprache (`Exp`)?

- wie lauten die Monad-Instanzen für `Action`, `CPS`, `Parser`, was bedeutet jeweils das bind (>>=)?
- warum benötigt man call-by-name für Abstraktionen über den Programmablauf (warum kann man `if` oder `while` nicht mit call-by-value implementieren)?
- wie kann man call-by-name simulieren in einer call-by-value-Sprache?
- wie kann man call-by-value simulieren in einer call-by-name-Sprache (Antwort: durch CPS-Transformation)
- Definiere Fakultät mittels Fixpunktoperator (Definiere das

`f` in `fak = fix f`)

- Bezüglich welcher Halbordnung ist dieses `f` monoton? (Definiere die Ordnung, erläutere Monotonie an einem Beispiel.)

- Wie kann man Rekursion durch get/put simulieren? (Programmbeispiel ergänzen)

- Wie kann man Rekursion durch label/jump simulieren? (Programmbeispiel ergänzen)

- Für die Transformationen CPS, Closure Conv., Lifting, Registervergabe: welche Form haben jeweils Eingabe- und Ausgabeprogramm? Auf welchem Maschinenmodell kann das Zielprogramm ausgeführt werden? (Welche Operationen muß das Laufzeitsystem bereitstellen?)

- Was sind die Bestandteile eines Inferenzsystems (Antwort: Grundbereich, Axiome, Regeln), wie kann man ein Axiom als Spezialfall einer Regel auffassen?

- wie lauten die Inferenzregeln für das Nachschlagen eines Namens in einer Umgebung?

- Inferenzregeln für Applikation, Abstraktion, Let, If/Then/Else im einfach getypten Kalkül

- Geben Sie ein Programm an, das sich nicht einfach (sondern nur polymorph) typisieren läßt. Geben Sie den polymorphen Typ an.

- Inferenz-Regeln für Typ-Applikation, Typ-Abstraktion im polymorphen Kalkül

- für Typ-Rekonstruktion im einfach getypten Kalkül: Welches ist der Grundbereich des Inferenzsystems?

- geben Sie die Inferenzregel für Typrekonstruktion bei If/Then/Else an

- Geben Sie eine Inferenzregel für Typrekonstruktion an, durch die neue Variablen eingeführt werden.

- Wann ist σ ein Unifikator von zwei Termen s, t?

- Geben Sie zwei verschiedene Unifikatoren von $f(a, X)$ und $f(Y, Z)$ an. Einer davon soll streng allgemeiner als der andere sein. Begründen Sie auch diese Beziehung.

- Bestimmen Sie einen Unifikator von $f(X_n, f(X_{n-1}, \ldots, f(X_0, a) \ldots))$ und $f(f(X_{n-1}, X_{n-1}), f(f(X_{n-2}, X_{n-2}), \ldots, f(a, a) \ldots))$.