Compilerbau
Vorlesung
Wintersemester 2008, 09, 10, 11

Johannes Waldmann, HTWK Leipzig

31. Januar 2013

Einleitung

Beispiel

Eingabe (≈ Java):	Ausgabe (Drei-Adress-Code):

```
{ 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);
}
```

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

Compiler und andere Werkzeuge

- Quellprogramm
- Präprozessor → modifiziertes Quellprogramm
- Compiler → Assemblerprogramm
- Assembler → verschieblicher Maschinencode
- Linker, Bibliotheken → ausführbares Maschinenprogramm

Phasen eines Compilers

- Zeichenstrom
- lexikalische Analyse → Tokenstrom
- syntaktische Analyse → Syntaxbaum
- semantische Analyse → annotierter Syntaxbaum
- Zwischencode-Erzeugung → Zwischencode
- maschinenunabhängige Optimierungen → Zwischencode
- Zielcode-Erzeugung → Zielcode
- maschinenabhängige Optimierungen → 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
```

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 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? $(-1,5)$? $(2,4)$?

Inferenz-Systeme (Beispiel 2)

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

$$\overline{01}$$

- Regel-Schemata (für jedes u,v):

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

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$_k$: $\frac{[\dots,x_k,x_{k+1},\dots]}{[\dots,x_{k+1}+1,x_k-1,\dots]}$
  - rotate: $\frac{[x_1,\dots,x_n]}{[x_2,\dots,x_n,x_1]}$

Aufgaben: • Ableitungen für $[5,3,1,3]$, $[7,7,1]$

- jede Folge der Form $[z,0,\dots,0]$ 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 $p \in$ `Exp` , $z \in \mathbb{Z}$

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

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

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

Umgebungen (Spezifikation)

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

Umgebungen (Implementierung)

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

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

Spezifikation:

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

Aussagenlogische Resolution

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

folgendes Inferenzsystem heißt *Resolution*:

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

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

Resolution (Vollständigkeit)

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

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

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:

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

konkrete Syntax (Alternative):

```
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
```

Let und Lambda

- `let { x = A } in Q`

  kann übersetzt werden in

  `(\ x -> Q) A`

- `let { x = a , y = b } in Q`

  wird übersetzt in …

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

Mehrstellige Funktionen

… simulieren durch einstellige:

- mehrstellige Abstraktion:

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

- mehrstellige Applikation:

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

  (die Applikation ist links-assoziativ)

- der Typ einer mehrstelligen Funktion:

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

  (der Typ-Pfeil ist rechts-assoziativ)

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 -> ...
```


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

Lambda-Kalkül

Motivation

gebundene (lokale) Variablen in der …

- Analysis: $\int x^2 \, \mathrm{d}\, x, \sum_{k=0}^{n} k^2$
- Logik: $\forall x \in A : \forall y \in B : P(x, y)$
- Programmierung:
  `static int foo (int x) { ... }`

Der Lambda-Kalkül

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

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

Beispiel: $(\lambda x.x * x)(3 + 2) \to (3 + 2) * (3 + 2)$
Im reinen Lambda-Kalkül gibt es *nur* Funktionen—keine Zahlen

Lambda-Terme

Menge Λ der Lambda-Terme (mit Variablen aus einer Menge V):

- (Variable) wenn $x \in V$, dann $x \in \Lambda$
- (Applikation) wenn $F \in \Lambda, A \in \Lambda$, dann $(FA) \in \Lambda$
- (Abstraktion) wenn $x \in V, B \in \Lambda$, dann $(\lambda x.B) \in \Lambda$

das sind also Lambda-Terme:
$x, (\lambda x.x), ((xz)(yz)), (\lambda x.(\lambda y.(\lambda z.((xz)(yz)))))$

verkürzte Notation

- Applikation als links-assoziativ auffassen:

$$(\dots((FA_1)A_2)\dots A_n) \sim FA_1A_2\dots A_n$$

  Beispiel: $((xz)(yz)) \sim xz(yz)$

- geschachtelte Abstraktionen unter ein Lambda schreiben:

$$\lambda x_1.(\lambda x_2.\dots(\lambda x_n.B)\dots) \sim \lambda x_1 x_2 \dots x_n.B$$

  Beispiel: $\lambda x.\lambda y.\lambda z.B \sim \lambda xyz.B$

- die vorigen Abkürzungen sind sinnvoll, denn $(\lambda x_1 \dots x_n.B)A_1 \dots A_n$ verhält sich wie eine Anwendung einer mehrstelligen Funktion.

Gebundene Variablen

Def: Menge $\mathrm{FV}(t)$ der *freien Variablen* von $t \in \Lambda$

- $\mathrm{FV}(x) = \{x\}$
- $\mathrm{FV}(FA) = \mathrm{FV}(F) \cup \mathrm{FV}(A)$
- $\mathrm{FV}(\lambda x.B) = \mathrm{FV}(B) \setminus \{x\}$

Def: Menge $\mathrm{BV}(t)$ der *gebundenen Variablen* von $t \in \Lambda$

- $\mathrm{BV}(x) = \emptyset$
-
-

Substitution

$A[x := N]$ ist (eine Kopie von) A, wobei jedes freie Vorkommen von x durch N ersetzt ist.
Definition durch strukturelle Induktion

- A ist Variable (2 Fälle)
- A ist Applikation
- A ist Abstraktion
  - $(\lambda x.B)[x := N] = \lambda x.B$
  - $(\lambda y.B)[x := N] = \lambda y.(B[x := N])$, falls $x \neq y$ und ...

Das falsche Binden von Variablen

Diese Programme sind *nicht* äquivalent:

```
int f (int y) {
  int x = y + 3; int sum = 0;
  for (int y = 0; y<4; y++)
        { sum = sum + x    ; }
  return sum;
}
int g (int y) {
                int sum = 0;
  for (int y = 0; y<4; y++)
        { sum = sum + (y+3); }
  return sum;
}
```


Gebundene Umbenennungen

Relation $\to_\alpha$ auf Λ:

- Axiom: $(\lambda x.B) \to_\alpha (\lambda y.B[x := y])$ falls $y \notin V(B)$.
- Abschluß unter Kontext:
  $\dfrac{F \to_\alpha F'}{(FA) \to_\alpha (F'A)}, \dfrac{A \to_\alpha A'}{(FA) \to_\alpha (FA')}, \dfrac{B \to_\alpha B'}{\lambda x.B \to_\alpha \lambda x.B'}$

$\equiv_\alpha$ ist die durch $\to_\alpha$ definierte Äquivalenzrelation
(die transitive, reflexive und symmetrische Hülle von $\to_\alpha$)

Bsp. $\lambda x.\lambda x.x \equiv_\alpha \lambda y.\lambda x.x$, $\lambda x.\lambda x.x \not\equiv_\alpha \lambda y.\lambda x.y$

wir betrachten ab jetzt $\Lambda / \equiv_\alpha$
(d. h., Äquivalenzklassen von Termen)
(vgl. rationale Zahlen als Äquivalenzklassen von Paaren)

Ableitungen

Absicht: Relation $\to_\beta$ auf $\Lambda / \equiv_\alpha$ (Ein-Schritt-Ersetzung):

- Axiom: $(\lambda x.B)A \to_\beta B[x := A]$
  ein Term der Form $(\lambda x.B)A$ heißt *Redex* (= reducible expression)
- Abschluß unter Kontext:
  $\dfrac{F \to_\beta F'}{(FA) \to_\beta (F'A)}, \dfrac{A \to_\beta A'}{(FA) \to_\beta (FA')}, \dfrac{B \to_\beta B'}{\lambda x.B \to_\beta \lambda x.B'}$

Vorsicht:
$(\lambda x.(\lambda y.xyx))(yy) \to_\beta (\lambda y.yx)[x := (yy)] \stackrel{?}{=} \lambda y.y(yy)$
das freie y wird fälschlich gebunden
die Substitution ist nicht ausführbar, man muß vorher lokal umbenennen

Eigenschaften der Reduktion

$\to$ auf Λ ist

- konfluent
  $\forall A, B, C \in \Lambda : A \to_\beta^* B \wedge A \to_\beta^* C \Rightarrow \exists D \in \Lambda : B \to_\beta^* D \wedge C \to_\beta^* D$
- (Folgerung: jeder Term hat höchstens eine Normalform)
- aber 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 $K = \lambda xy.x, I = \lambda x.x$

Daten als Funktionen

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

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

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

- Auflösung simultaner Rekursion
- Modellierung von Zahlen

Lambda-Kalkül als universelles Modell

- Wahrheitswerte:

  True $= \lambda xy.x$, False $= \lambda xy.y$

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

- natürliche Zahlen:

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

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

- Rekursion?

Fixpunkt-Kombinatoren

- Definition: $\Theta = (\lambda xy.(y(xxy)))(\lambda xy.(y(xxy)))$
- Satz: $\Theta f \to_\beta f(\Theta f)$, d. h. Θf ist Fixpunkt von f
- d.h. Θ ist *Fixpunkt-Kombinator*, (T wegen Turing)
- Beweis (ausrechnen)
- Folgerung: im Lambda-Kalkül kann man beliebige Wiederholung (Schachtelung) von Rechnungen beschreiben

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 Lambda-Kalkül

- Konstruktor und Selektoren für Paare
- Test, ob der Nachfolger von 0 gleich 0 ist

  (mit λ-kodierten Zahlen)

- Fakultät mittels Θ

  (mit „echten“ Zahlen und Operationen)

Fixpunkte

Motivation

Das ging bisher gar nicht:

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

Lösung 1: benutze Fixpunktkombinator

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

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

Existenz von Fixpunkten

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

- CPO $=$ complete partial order $=$ vollständige Halbordnung
- complete $=$ jede monotone Folge besitzt Supremum ($=$ kleinste obere Schranke)
- pointed: C hat kleinstes Element $\perp$
- stetig: $x \le y \Rightarrow f(x) \le f(y)$ und für monotone Folgen $[x_0, x_1, \ldots]$ gilt: $f(\sup[x_0, x_1, \ldots]) = \sup[f(x_0), f(x_1), \ldots]$

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

Beispiele f. Halbordnungen, CPOs

Halbordnung? pointed? complete?

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

Funktionen als CPO

- Menge der partiellen Funktionen von B nach B:

  $C = (B \hookrightarrow B)$

- partielle Funktion $f : B \hookrightarrow B$
  entspricht totaler Funktion $f : B \to B_\perp$
- C geordnet durch $f \le g \iff \forall x \in B : f(x) \le g(x)$,

  wobei $\le$ die vorhin definierte CPO auf $B_\perp$

- $f \le g$ bedeutet: g ist Verfeinerung von f
- Das Bottom-Element von C ist die überall undefinierte Funktion. (diese heißt auch $\perp$)

Funktionen als CPO, Beispiel

der Operator $F =$

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

ist stetig auf $(\mathbb{N} \hookrightarrow \mathbb{N})$ (Beispiele nachrechnen!)
Iterative Berechnung des Fixpunktes:

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

Fixpunktberechnung im Interpreter

Erweiterung der abstrakten Syntax:

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

Beispiel

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

Bedeutung: `Rec x B` bezeichnet den Fixpunkt von $(\lambda x.B)$
Definition der Semantik:

```
value (E, Rec x B) =
    fixpoint $ \ v -> value (E[x:=v],B)
```


Fixpunkte und Laziness

Fixpunkte existieren in pointed CPOs.

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

Beispiele in Haskell:

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


Simultane Rekursion: letrec

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

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

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

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


letrec nach rec

mittels 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 )
```


Übung Fixpunkte

- Limes der Folge $F^k(\bot)$ für

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

- Limes der Folge $F^k(\bot)$ für

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

- gegenseitige Rekursion (f, g) als Fixpunkt (`Rec`) einer geeigneten Funktion (benutzt Tupel)
- (Ergänzung zu Lambda-Kalkül:)
  Turing-Fixpunkt-Kombinator mit
  `http://joerg.endrullis.de/lambdaCalculator/`

Zustand/Speicher

Motivation

bisherige Programme sind nebenwirkungsfrei, das ist nicht immer erwünscht:

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

Dazu muß semantischer Bereich geändert werden.

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

Semantik von (Teil-)Programmen ist Zustandsänderung.

Speicher

```
import qualified Data.Map as M
http://hackage.haskell.org/packages/archive/container
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 -> ...
```


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

- mit Fixpunkt-Kombinator (bekannt)
- in der Gastsprache des Interpreters
  (dabei neu: Fixpunkte von Aktionen)
- (neu:) simulieren (in der interpretierten Sprache)
  durch Benutzung des Speichers

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 )
```


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

Die Konstruktorklasse Monad

Definition:

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

Beispiele für Monaden

- Aktionen mit Speicheränderung (vorige Woche)

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

- Aktionen mit Welt-Änderung: `IO a`
- Transaktionen (Software Transactional Memory) `STM a`
- Aktionen, die möglicherweise fehlschlagen:

  `data Maybe a = Nothing | Just 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
```


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 )
```

Gesetze für Monaden

das Wort *Monade* ist abgeleitet von *Monoid*,
für jede Implementierung muß `return` und >=> folgende Eigenschaften erfüllen:

- Grundbereich: Funktionen des Typs `a -> m b`
- Verküpfung (Kleisli-Komposition)

```
(>=>) ::
    (a -> m b) -> (b -> m c) -> (a -> m c)
(f >=> g) = \ x -> (f x >>= \ y -> g y)
```

- `return` ist (links- und rechts-)neutral für >=>
- >=> ist assoziativ

Beispiele? Beweise?

Monaden: Zusammenfassung

- verwendet zur Abstraktion vom Programmablauf
  (das Semikolon, das Anweisungen verknüpft, kann umdefiniert werden)
- Notation `do { x <- foo ; bar ; .. }`
  ähnlich zu imperativen Programmen
- Grundlagen: Kategorien-Theorie (ca. 1960),
  in Funktl. Prog. seit ca. 1990
  `http://homepages.inf.ed.ac.uk/wadler/topics`
- in anderen Sprachen: *Workflows* in F#, LINQ-Syntax in C#

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

Definition:

```
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

Anwendungsbeispiel:

```
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.`

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: $\dfrac{E \vdash X : \text{Int}, E \vdash Y : \text{Int}}{E \vdash (X + Y) : \text{Int}}, \ldots$
- Abstraktion/Applikation: ...
- Binden/Benutzen von Bindungen: ...

hierbei (vorläufige) Design-Entscheidungen:

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

Inferenz für Let

(alles ganz analog zu Auswertung von Ausdrücken)

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

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

  vergleiche mit *modus ponens*
- Abstraktion (mit deklariertem Typ der Variablen)

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

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

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

- Typ-Applikation: instantiiert param. Typ

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

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 → 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)

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

- 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

Substitutionen (Definition)

- Signatur $\Sigma = \Sigma_0 \cup \ldots \Sigma_k$,
- $\operatorname{Term}(\Sigma, V)$ 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 $\sigma : V \to \operatorname{Term}(\Sigma, V)$,

  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 $\Rightarrow \sigma$ idempotent: $\sigma \circ \sigma = \sigma$
- σ_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 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 $t_1, t_2 \in \operatorname{Term}(\Sigma, V)$
- 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

$\operatorname{mgu}(s, t)$ 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 Undeci-*
dable, Annals of Pure and Applied Logic 98 (1998) 111–156, `http://citeseerx.ist.psu.edu/viewdoc/summary`

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

Plan für Compiler

CPS-Transformation: Zielsyntax

drei Teilmengen von `data Exp`:

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

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

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

Übung 2: wegen CPS brauchen wir tatsächlich:

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


Beispiel

Lösung 1:

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

Lösung 2:

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


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 ()
evalState :: State s a -> s -> a

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
```


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`

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: Applikation zur Laufzeit

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

- jetzt: Applikation während der Transformation

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

für jedes Abs/App entscheiden, ob Lauf- oder Compilezeit

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 -> Transform Exp
cps :: Exp -> Cont -> Transform Exp
cps x = \ m -> case x of
    ConstInt i -> ...
    Plus a b -> ...
```


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))
```


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.

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

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 $(\lambda i_1 \ldots i_n \to b)$

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

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

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)

Registervergabe

Motivation

- (klassische) reale CPU/Rechner hat nur *globalen* Speicher (Register, Hauptspeicher)
- Argumentübergabe (Hauptprogramm $\to$ 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 $\to$ 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 $\mathrm{mark}(c) = \dots$

- 0: $c' = \mathrm{head}(c); \mathrm{head}'(c) = p; \mathrm{mark}'(c) = 1; p' = c;$
- 1,2,3: falls $\mathrm{mark}(p) = \dots$
  - 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 $u \rightarrow_G v$: 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/ec`
- Aufgabe:
  `http://www.imn.htwk-leipzig.de/~waldmann/ec`

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)

Prüfungsvorbereitung

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

- 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

- 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)

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`)

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

- 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

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