

# Compilerbau Vorlesung Wintersemester 2008

Johannes Waldmann, HTWK Leipzig

12. Oktober 2010

## 1 Einleitung

### Beispiel

Eingabe ( $\approx$  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 → 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:
  - Hausaufgaben (klein),
  - Projekt (mittelklein)
- Prüfung: Projektverteidigung  
(enthält auch Compilerbau-Fragen außerhalb des Projektes)

## Projekt-Themen

- Erkennen von Codeverdopplungen, Extrahieren von Mustern, Erkennen der Anwendbarkeit von Mustern
- Refactorings als Eclipsefp-Plugin <http://eclipsefp.sourceforge.net/>
- Erweiterung der autotool-Aufgabe zu Java-Methodenaufrufen <https://autotool.imn.htwk-leipzig.de/cgi-bin/Trial.cgi?topic=TypeCheck-Quiz>  
(static → nonstatic, generic)
- SQL-Syntaxbäume (Query-Objekte)

## 2 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 Verketteten 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  $S \rightarrow aSbS, S \rightarrow \epsilon$  entspricht

```

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

```

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

## Parser für (geschachtelte) Listen

konkrete Syntax:

```
(foo (bar baz ()) 1234)
```

abstrakte Syntax:

```
data Expression
  = Identifier String
  | List [ Expression ]
```

- `parse: String -> Expression`
- `print: Expression -> String`
- `pretty-print: Expression -> Doc`

## Robuste Parser-Bibliotheken

Designfragen:

- Nichtdeterminismus einschränken
- Fehlermeldungen (Quelltextposition)

Beispiel: Parsec (Autor: Daan Leijen) <http://www.haskell.org/haskellwiki/Parsec>

Übungen:

- parsec-Parser aufrufen
- parsec-Parser selbst schreiben (elementare, Kombinatoren)
- `buildExpressionParser`

## 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`

## 3 Konkrete abstrakte Syntax

### Motivation (I)

- wollen uns in (diesem) Compilerbau nicht mit *konkreter Syntax* beschäftigen,  
(ist zwar ein klassisches Gebiet: kontextfreie Grammatiken, Kellerautomaten, LL/LR-Parser usw., aber leider haben wir keine Zeit—und mit Parsec kommt man schon sehr sehr weit)
- sondern mit *Semantik* (Interpretation, Kompilation)  
dazu muß nur die abstrakte Syntax möglichst einfach und systematisch repräsentiert werden.

### Motivation (II)

Datenaustausch

- in einem Programm: Objekte der Programmiersprache (Graphen, Bäume)
- zwischen Programmen gleicher Sprache: binäre Serialisierung
- zwischen Programmen verschiedener Sprachen: textuelle, sprachunabhängige Serialisierung

Ziel: Anwendungs- und sprachunabhängige konkrete Syntax.

## Varianten

- 199?: XML(-RPC)
- 200?: JSON
- 196?: LISP

die konkrete Syntax von LISP (für Daten *und* Programme) löst das Problem bereits so gut wie möglich

vgl. Phil Wadler: *The Essence of XML* (slides of 2002 POPL talk) <http://homepages.inf.ed.ac.uk/wadler/topics/xml.html>

## JSON

<http://json.org/>

Java Script Object Notation

value →

- string, number, (true, false, null)
- object { foo = value1, bar = value2 }
- array [ value1, value2, value3 ]

## Zweistufige Syntax

(für die Beispielsprachen aus dem Compilerbau, wie im Buch Design Concepts of Programming Languages)

- konkret: String
- konkret abstrakt: LISP, S(ymbolic) Exp(ressions)
- abstrakt: anwendungsspezifisch

Transformationen:

- String ↔ Sexp:  
Typklassen Autolib.Reader/ToDoC
- Sexp ↔ abstrakt:  
Typklassen DCPL.Input/Output

## 4 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), Syntax besteht aus:
  - Variablen  $x, y, \dots$
  - Applikationen  $xx, (\lambda x.xx)y$
  - Abstraktionen  $\lambda x.xx, \lambda x.(\lambda y.x)$

### Konkrete Abstrakte Syntax

abstrakt:

```
data Exp = I S.Id -- Bezeichner (Variable)
  | Abstraction
      { formal :: S.Id , body :: Exp }
  | Application
      { rator  :: Exp , rand  :: Exp }
```

konkret:

```
(lam g (lam a (lam b (app (app g b) a))))
```

(nach Turbak, Gifford: Design Concepts in Programming Languages)

### Primitive Operationen

(= Funktionen, die keine Lambda-Ausdrücke sind, weil sie direkt auf primitiven Werten rechnen)

abstrakt:

```
data Exp = ...
  | Literal Integer
  | PrimitiveApplication
      { primop :: PrimOp , arg :: [ Exp ] }
data PrimOp
  = PrimOp { name :: S.Id }
```

konkret:

`(prim + 2 (prim * 3 4))`

### **Wiederholung: Operationale Semantik**

(SOS - small step operational semantics)

Ein-Schritt-Relation

- $\beta$ -Reduktion:  $(\lambda x.M)N \rightarrow_{\beta} M[x := N]$

falls  $\text{bound-vars}(M)$  und  $\text{free-vars}(N)$  disjunkt.

bei Bedarf gebundene Var. umbenennen

- $\alpha$ -Reduktion:  $\lambda x.M \rightarrow_{\alpha} \lambda y.M[x := y]$

Abschluß von  $\rightarrow_{\beta}$  unter Kontexten:

- $A_1 \rightarrow_{\beta} A_2 \Rightarrow A_1B \rightarrow_{\beta} A_2B, BA_1 \rightarrow_{\beta} BA_2, \lambda xA_1 \rightarrow_{\beta} \lambda xA_2$

operationale Semantik (= Programmausführung) ist reflexive transitive Hülle  $\rightarrow_{\beta}^*$   
(Mehr-Schritt-Relation)

### **Denotationale Semantik**

Denotationale Semantik eines Unterprogrammes (in einer deklarativen Programmiersprache) ist eine *Funktion*.

- Semantik von Literalen:  
bezeichnen (primitive) Objekte selbst
- Semantik primitiver Operationen:
  - (Zahlen-)Werte der Argumente bestimmen,
  - dann Operation anwenden
- Semantik von Lambda-Ausdrücken (Abstraktionen, Applikationen)?

## Denotationale Semantik (II)

Semantik von *lokalen* Unterprogrammen hängt ab von weiter außen gebundenen Variablen.

zutreffendes Modell ist deswegen:

- Semantik  $:: \text{Exp} \times \text{Env} \rightarrow \text{Value}$
- Exp ... Menge der Ausdrücke
- Value ... Menge der Werte
- Umgebungen  $\text{Env} :: \text{Var} \rightarrow \text{Value}$
- Var ... Menge der Variablen

(Vorsicht, das ist teilw. vereinfacht, Verallgemeinerung folgt)

## Umgebungen

- Umgebungen  $\text{Env} :: \text{Var} \rightarrow (\text{Value} \cup \text{Error})$

(Variable nicht gebunden: Error)

Operationen mit Umgebungen:

- lookup  $:: \text{Var} \rightarrow \text{Env} \rightarrow (\text{Value} \cup \text{Error})$   
lookup  $i e = ei$
- leere Umgebung = const Error =  $\lambda j. \text{Error}$
- neue Bindung:  
 $e[i/v] := \lambda j. \text{if } i = j \text{ then } v \text{ else } ej$

## Semantische Bereiche

Value = ...

- Zahlen (für prim. Op. )
- Wahrheitswerte (für prim. Op.)
- in Applikation ( $\text{app } f \ a$ ) ist der Wert von  $f$  eine Funktion (Sprechweise hier: procedural value)

unvollständig und vereinfacht:

```

data Value
  = VInt Integer
  | VBool Bool
  | VProc ( Value -> Value )

```

### call-by-value

Semantik von Applikation (app f a) in Umgebung e

- Sem. von  $f$  in  $e$  ist  $VProc\ p :: Value$
- Sem. von  $a$  in  $e$  ist  $v :: Value$
- berechne  $p\ v$

Semantik von Abstraktion (lam i b) in Umgebung e

- ist Funktion  $\lambda x. \dots$
- Semantik von  $b$  in  $e[i/x]$

Semantik von Bezeichner i in Umgebung e

- lookup i e

### Andere Semantiken?

Beispiel:

```

(prim / 1 0) ==> (error divide-by-zero)
(app (lam x 3) (prim / 1 0)) =?=> 3

```

Umgebung speichert für Namen ( $x$ ):

- call-by-value:  
*Wert* des Arguments
- call-by-name:  
*Rechnung*, die diesen Wert ergibt

## Call-by-name

Semantik von Applikation ( $\text{app } f \ a$ ) in Umgebung  $e$

- Sem. von  $f$  in  $e$  ist  $\text{VProc } p :: \text{Value}$
- wende  $p$  an auf Sem. von  $a$  in  $e$

Semantik von Abstraktion ( $\text{lam } i \ b$ ) in Umgebung  $e$

- ist Funktion  $\lambda x. \dots$
- Semantik von  $b$  in  $e[i/x]$

## Zusammenfassung bisher

allgemein:

- Value (Werte): Zahl, Proc, ...
- Comp (Rechnung): Resultat der Semantik-Funktionen
- Nameable: kann benannt werden, steht in Umgebungen  
beachte:  $\text{Value} = \dots \text{VProc} (\text{Nameable} \rightarrow \text{Comp})$

nebenwirkungsfreie (deklarative) Sprachen:

- $\text{Comp} = (\text{Value} \cup \text{Error})_{\perp} \dots (\text{OK}, \text{Fehler}, \text{Divergenz})$

call-by-value:  $\text{Nameable} = \text{Value}$

call-by-name:  $\text{Nameable} = \text{Comp}$

später: Nebenwirkungen:  $\text{Comp} = \text{State} \rightarrow (\text{Value}, \text{State})$

# 5 Syntactic Sugar

## Motivation

Semantik für Spracheigenschaft:

- direkt implementieren (z. B. Interpreter)
- indirekt implementieren (durch Transformation auf Kernsprache)

Transformationen sind „nur Syntax“, daher die Bezeichnung *syntactic sugar*.

## Mehrfach-Abstraktion/Applikation

bereits im Lambda-Kalkül vereinbart:

$$((\lambda xy.M)AB) \equiv (((\lambda x.(\lambda y.M))A)B)$$

bisher: konkret: `(lam x b)`, `(app f a)`, abstrakt:

```
| Abstraction { formal :: S.Id , body :: Exp }
| Application { rator :: Exp, rand :: Exp }
```

neu: konkret: `(abs (x y) b)`, `(f a b c)`, abstrakt:

```
| MultiAbstraction
  { mformal :: [ S.Id ] , body :: Exp }
| MultiApplication
  { rator :: Exp, mrand :: [ Exp ] }
```

## Mehrfach-Abstraktion/Applikation

Desugaring durch Ersetzungsregeln:

```
(abs [] b) -> b
(abs (x:xs) b) -> (lam x (abs xs b))
```

```
(f [] ) -> f
(f (x:xs)) -> ((app f x) xs)
```

nützliche Abkürzung für primitive Operationen:

```
(prim + 3 4) -> (@+ 3 4)
```

## Lokale Bindungen

konkret:

```
(let ((n1 x1) (n2 x2)) y)
```

abstrakt:

```
data Exp = ...
| LocalBinding
  { binders :: [ Binder ] , body :: Exp }
data Binder =
  Binder { bname :: S.Id, bdefn :: Exp }
```

Übersetzung in Applikation/Abstraktion

Was folgt daraus über Sichtbarkeiten?

## Lokale Bindungen (Übersetzung)

Übersetzung in Multi-Applikation/Abstraktion

```
(let [(n1, x1), ..., (nk, xk)] y)
  -> ((abs [n1, ..., nk] y) x1 .. xk)
```

Beachte: das geht nicht:

```
(let ((x (@+ 3 4)) (y (@* x x))) (@- x y))
```

die  $x$  in Definition von  $y$  beziehen sich nicht auf das  $x$  aus der ersten Definition

Abhilfe:

```
(let ((x (@+ 3 4)))
      (let ((y (@* x x)) (@- x y)) )
```

## 6 Fixpunkte

### Motivation

Das geht bisher gar nicht:

```
(let ((f (lam x (if (@= x 0)
                    1
                    (@* x (app f (@- x 1)))))) )
      (app f 3) )
```

(Bezeichner  $f$  ist nicht sichtbar)

Lösung:

```
( (rec f (lam x (if ... (app f ...)))) 3)
```

mit neuem primitiven Knotentyp `rec`

### Rekursion

abstrakt:

```
data Exp = ...
| Recursion { rname :: S.Id, body :: Exp }
```

konkret:

```
(rec n b)
```

Semantik von `rec n b` in Umgebung  $E$   
 ist der Fixpunkt (vom Typ  $\text{Comp}$ )  
 der Funktion (vom Typ  $\text{Comp} \rightarrow \text{Comp}$ )

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

### Existenz von Fixpunkten

Fixpunkt von  $f :: C \rightarrow C$  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:  $f(\sup \vec{x}) = \sup f(\vec{x})$

Dann  $\text{fix}(f) = \sup[\perp, f(\perp), f^2(\perp), \dots]$

### Funktionen als CPO

- partielle Funktionen  $C = (B \rightarrow B)$
- Bereich  $B \cup \perp$  geordnet durch  $\forall x \in B : \perp < x$
- $C$  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 (lam x
  (if (@= x 0) 1 (@* x (app f (@- x 1))))))
```

ist Fixpunkt der Funktion  $F =$

```
(lam f (lam x
  (if (@= x 0) 1 (@* x (app f (@- x 1)))) )))
```

Iterative Berechnung des Fixpunktes:

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

### letrec

konkret:

```
(letrec ((n1 x1) (n2 x2)) y)
```

wobei  $n_1, n_2$  sichtbar in  $x_1, x_2$

abstrakt:

```
data Exp = ...
| RecursiveBinding
  { binders :: [ Binder ] , body :: Exp }
```

Beispiele:

```
(letrec ((x (@+ 3 4)) (y (@* x x))) (@- x y))
(letrec ((f (lam x (... (f (@- x 1)) ...)))) (f 3))
```

### letrec: Transformation nach rec

(DCPL Fig. 6.7 p. 233)

```
(letrec ((n1 x1) .. (nk xk)) y)
->
(app (rec t
      (lam s
        (s (t (abs (n1..nk) x1)
              ...
              (abs (n1..nk) xk) ))))
      (abs (n1..nk) y) )
```

## 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  $f(Y f)$
- Folgerung: `rec` wird eigentlich nicht benötigt

## 7 Daten

### Paare

Syntax:

```
(pair x y) -- Konstruktor
```

```
(fst p)    -- Destruktor
```

```
(snd p)    -- Destruktor
```

Semantik (Plan):

für alle  $x, y$ :

```
(fst (pair x y)) == x
```

```
(snd (pair x y)) == y
```

### Listen

mit solchen Paaren kann man im Prinzip alles ausdrücken, z. B. Listen durch syntactic sugar:

```
(list [] ) --> #u
```

```
(list (x:xs) ) --> (pair x (list xs))
```

### Semantik für Paare

was soll die Semantik dieses Ausdrucks sein?

```
(fst (pair 1 (@/ 1 0)))
```

- `1` : pair-Konstruktor ist *nicht strikt*
- `(error divide-by-zero)` : pair-Konstruktor ist *strikt*

## Strikte Paare

```
data Value = ...
           | Pair Value Value
```

Semantik von `(pair X1 X2)` in Umgebung  $E$  ist die folgende Rechnung:

- führe Semantik von  $X1$  in  $E$  aus, liefert  $v1 :: Value$
- führe Semantik von  $X2$  in  $E$  aus, liefert  $v2 :: Value$
- Resultat ist `Pair v1 v2`

Semantik von `(fst p)`: führe Semantik von  $p$  aus, liefert `Pair v1 v2`, Resultat ist  $v1$ .

## Nicht strikte Paare

```
data Value = ...
           | Pair Comp Comp
```

Semantik von `(pair X1 X2)` in Umgebung  $E$  ist diese Rechnung:

- Semantik von  $X1$  in  $E$  ist  $c1 :: Comp$
- Semantik von  $X2$  in  $E$  ist  $c2 :: Comp$
- Resultat ist `Pair c1 c2`

Semantik von `(fst p)`: führe Semantik von  $p$  aus, liefert `Pair c1 c2`, führe  $c1$  aus, Resultat ist  $v1$ .

## Simulation von nicht strikten Paaren

```
(ns-pair X Y)
--> (pair (abs (u) X) (abs (u) Y))
```

```
(ns-fst P) --> ((fst P) #u)
```

- in CBV müssen das desugaring-Regeln sein, `ns-pair` ist keine CBV-Funktion.
- $u$  darf nicht in  $X, Y$  vorkommen
- `(abs (u) X)` heißt *thunk*

mit *thunking* nur für zweites `pair`-Argument erhält man (lazy) streams, vgl. `Iterator/Enumerator` in Java/C#

## 8 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: `Comp = Value + Error`
- jetzt: `Comp = State -> ( Value, State )`

Semantik von (Teil-)Programmen ist Zustandsänderung.

### Speicher

```
type Store = Map Location Value
```

Syntax: Ausdrücke:

```
data Exp = ...
  | (cell Exp) -- CellCreation
  | (begin Exp Exp) -- SimpleSequencing
```

Primops:

- lesen ( $@^ E$ )
- schreiben ( $@ := E1 E2$ ),
- testen (ist Location?) ( $@cell? E$ ),
- test (gleiche Location?) ( $cell=? E1 E2$ ).

Beachte: explizite Dereferenzierung ( $@^$ )

## Syntactic Sugar

- Anweisungsfolgen beliebiger Länge

```
(begin) --> #u ; (begin E) --> E
(begin (E:Es)) --> (begin E (begin Es))
```

- „if ohne else“

```
(if test yeah) --> (if test yeah #u)
```

- Wiederholung

```
(while test body) -->
  (letrec ((loop (...))) loop)
```

## Semantik f. Speicher

```
type Store = Map Location Value
type State = Store
```

```
type Comp = State -> ( Value, State )
```

Das ist genau die Zustands-Monade aus Haskell.

primitive Operationen (vgl. DCPL.Store)

- allocating
- fetching
- update

## Veränderliche Variablen

(bis jetzt sind unsere „Variablen“ konstant und nur die Zellen veränderlich.)

neue Syntax (set! I E), I ... Identifier, E ... Exp

erfordert Änderung in der Semantik: bisher: Nameable = Value, jetzt: Nameable = Location

Realisierung von (Name  $\rightarrow$  Wert) durch *zwei* Bindungen:

- (in Umgebung:) Name  $\rightarrow$  Location
- (im Speicher:) Location  $\rightarrow$  Value

Ausblick:

- CBV, CBN, CBReference
- Übersetzung: veränd. Var  $\rightarrow$  veränd. Zellen

### Assignment Conversion

(als Transformationsschritt im Compiler):

- Eingabe: Programm mit veränderlichen Variablen
- Ausgabe: Programm mit „konstanten Variablen“ und veränderlichen Zellen

Variable ersetzen durch (Verweis auf) Zelle:

```
(begin (set! x (@* 2 y)) x)
==>
(let ((x (cell x)) (y (cell y)))
  (begin (@:= x (@* 2 (@^ y))) (@^ x)))
```

Bessere Implementierung: nur für die Variablen, die tatsächlich Zuweisungsziel sind (im Bsp: y nicht)

## 9 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 Resultat-Tupel

wie modelliert man Funktion mit mehreren Rückgabewerten?

- benutze Datentyp Tupel (Paar):

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

- benutze Continuation:

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

## CPS/Tupel-Beispiel

erweiterter Euklidischer Algorithmus:

```
prop_egcd x y =
  let (p,q) = egcd x y
  in abs (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))))))
```

semantischer Bereich:

```
data Value = ... | VCtrlPoint Expcont
```

d. h. Continuations sind Werte

```
Expcont = Value -> State Store Value
Comp     = Expcont -> State Store Value
```

Modul: DCPL.Semantics.FLICK.Cont

## Semantik für CPS

Semantik von Ausdruck  $x$

in Umgebung  $E$  mit Continuation  $k$

- $x = (\text{app } f \ a)$   
Semantik von  $f$  in  $E$  mit Continuation:  $\lambda p.$   
Semantik von  $a$  in  $E$  mit Continuation:  $\lambda v.p \ v \ k$
- $x = (\text{label } L \ B)$   
Semantik von  $B$  in Umgebung  $E[L/k]$  mit  $k$
- $x = (\text{jump } L \ B)$   
let  $k' =$  gebundener Wert von  $L$  in  $E$ ,  
Semantik von  $B$  in  $E$  mit  $k'$

## CPS-Transformation: Spezifikation

(als Schritt im Compiler)

- Eingabe: Ausdruck  $X$ , Ausgabe: Ausdruck  $Y$
- Semantik:  $X \equiv Y(\lambda v.v)$   
(triviale top-level continuation)
- Syntax:
  - $X \in \text{Exp}$  (fast) beliebig,
  - $Y \in \text{ExpCPS}$  stark eingeschränkt
    - \* keine geschachtelten Applikationen
    - \* Argumente von Applikationen und Primops 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)
             | (error Msg)
Exp/Value ==> Literal + Identifier
Exp/Letable ==> Literal
              | (abs Id^* Exp/CPS)
              | (prim Primop Exp/Value^*)
```

### Übersetze

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

### Beispiel

```
(@+ (@- 0 (@* b b)) (@* 4 (@* a c)))
==>
(let ((t.3 (@* b b)))
  (let ((t.2 (@- 0 t.3)))
    (let ((t.5 (@* a c)))
      (let ((t.4 (@* 4 t.5)))
        (let ((t.1 (+ t.2 t.4)))
          (app ktop.0 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  $k, i_0, \dots, i_n$  *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

## Besser: Meta-Continuations

- bisher: CPS:  $\text{Exp} \rightarrow \text{Exp}$
- jetzt: CPS:  $\text{Exp} \rightarrow \text{MetaCont} \rightarrow \text{Exp}$

```
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))
```

## 10 Closure Conversion

### Motivation

(Literatur: DCPL 17.10) — Beispiel:

```
(let ((linear
      (abs(a b) (abs (x) (@+ (@* a x) b))))))
  (let ((f (linear 4 5)) (g (linear 6 7)))
    (@+ (f 8) (g 9)) ))
```

beachte nicht lokale Variablen: `(abs (x) .. a .. b )`

- 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 (Code, Wert<sub>1</sub>, ..., Wert<sub>n</sub>)

```
(abs (x) (@+ (@* a x) b))
==>
(abs (clo x)
  (let ((a (@get 2 clo))
        (b (@get 3 clo)))
    (@+ (@* a x) b) ))
```

Closure-Konstruktion? Komplette Übersetzung des Beispiels?

### Transformation

```
CLP[ (abs (i_1 .. i_n) b) ] =
  (@prod (abs (clo i_1 .. i_n)
            (let ((v_1 (@get 2 clo)) .. )
              CLP[b] ))
        v_1 .. )
```

wobei  $\{v_1, \dots\}$  = freie Variablen in  $b$

```
CLP[ (app f a_1 .. a_n) ] =
  (let ((clo CLP[f])
        (code (@get 1 clo)))
    (app code clo CLP[a_1] .. CLP[a_n]) )
```

zur Erhaltung der CPS-Form: anstatt erster Regel

```
CLP[ (let ((i (abs (..) ..))) b) ] = ...
```

### Zuweisungen und Closures

die Werte der nicht lokalen Variablen werden kopiert (in die Closures).  
falls Zuweisungen an Variablen erlaubt sind (`set!`):

- *erst* assignment conversion
- *danach* closure conversion

### Vergleich mit inneren Klassen

```
interface Fun { int app (int x); }
class Linear {
  static Fun linear (int a, int b) {
    return new Fun() {
      int app (int x) { return a * x + b; }
    }
  }
  static int example() {
    Fun f = linear (4,5);
    Fun g = linear (6,7);
    return f.app(8) + g.app(9);
  }
}
```

Fehler? Bytecode?

## 11 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.

Syntax-Erweiterung:

```
(program (a_1 .. a_n)
  -- body:
  (let ( ... ) ...)
  -- ab hier neu:
  (def sub0 (abs (...) body0))
  ...
)
```

dann in `body*` keine Abstraktionen gestattet

## 12 Kombinatorische Logik

### Motivation

bisher: closure conversion + lifting:

beliebiges Programm (Lambda-Ausdruck)  $\Rightarrow$  Programm mit nur globalen Funktionen (und Tupeln)

man kann sogar jeden Lambda-Ausdruck in äquivalentes Programm mit wenigen und feststehenden globalen Funktionen (Kombinatoren) übersetzen.

### Beispiel

vordefinierte Kombinatoren:

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

Programm:  $P = \lambda x.xx$

Übersetzung:  $P' = SII$

Begründung:  $P'x = SIIx \rightarrow Ix(Ix) \rightarrow x(Ix) \rightarrow xx$

### Systematische Übersetzung

Spezifikation:

- Eingabe: geschlossener Lambda-Ausdruck  $P$

- Ausgabe: äquivalenter Kombinator-Ausdruck  $[P]$   
(Applikationen mit  $S, K, I$ ; sonst keine Variablen und Lambdas)

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

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

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?  $I = \dots$

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/>

### Rechnen mit Kombinatoren

- Standard-Basis:  $Kxy \rightarrow x, Ix \rightarrow x, Sxyz \rightarrow xz(yz)$
- Tupel:  $\langle A_1, \dots, A_n \rangle := \lambda x.xA_1 \dots A_n$
- Projektionen:  $P_i^n \langle A_1, \dots, A_n \rangle \rightarrow A_i$
- $X = \langle K, S, K \rangle$
- $XXX \rightarrow?, X(XX) \rightarrow?$
- Zahlen (nach Church):  $[n] := \lambda f.x.f^n(x)$
- Nachfolger? Summe?

(Barendregt, S. 140, 166)

## 13 Registervergabe

### Motivation

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

### Plan (I)

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

### Plan (II)

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

## Registerbenutzung

Allgemeine Form der Programme:

```
(let* ((r1 (...))
      (r2 (...))
      (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  $R_0, \dots, R_k$  stehen)

## Registervergabe und Graphenfärbung

Gegeben Programm (das Let innerhalb einer Abstraktion), konstruiere Graph  $G = (V, E)$

- Knoten  $V$ : die lokal gebundenen Namen
- Kanten  $E$ : falls  $x$  und  $y$  gleichzeitig benötigt, dann  $xy \in E$ .

gesucht ist konfliktfreie Färbung mit geringer Farbzahl:

- Farben  $C$ : (virtuelle) Register
- Färbung: Abbildung  $f : V \rightarrow C$
- konfliktfrei:  $\forall xy \in E : f(x) \neq f(y)$

## Algorithmen zur Färbung

Das Entscheidungsproblem COL

- Eingabe:  $(G, k)$
- Frage: existiert konfliktfreie Färbung  $f$  für  $G$  mit  $|\text{img } f| \leq k$

... ist NP-vollständig

⇒ kein effizienter Algorithmus bekannt

Näherungsverfahren (für Farben  $\{1, 2, \dots\}$ ):

- färbe der Reihe nach jeden Punkt mit der kleinsten freien Farbe (= die nicht unter seinen Nachbarn vorkommt)

Heuristik für gute Reihenfolge?

## Graphenparameter

- Maximalgrad  $\Delta(G)$
- chromatische Zahl  $\chi(G)$   
kleinstes  $k$ , für das  $G$  eine konfliktfreie  $k$ -Färbung besitzt
- Cliquenzahl  $\omega(G)$ : maximale Knotenzahl einer Clique  
(= vollständig verbundener Teilgraph)

Anwendungen:

- Folgerung aus Algorithmus:  $\chi(G) \leq \Delta(G) + 1$ .  
Für welche Graphen gilt Gleichheit?
- größter Fehler des heuristischen Algorithmus?
- trivial  $\omega(G) \leq \chi(G)$ . Möglicher Abstand?

## Register-Files

SPARC-Architektur:

- Register-File besteht aus Blöcken von je 8 Registern
- Register-Window zeigt je drei benachbarte Blöcke
- bei UP-Aufruf/Rückkehr wandert Window um zwei (!) Blöcke nach rechts/links (d. h. die Fenster überlappen)

Registersatz besteht aus

- global  $G_0, \dots, G_7$
- in  $I_0, \dots, I_7$  (links im Fenster)
- lokal  $L_0, \dots, L_7$  (mitte)
- out  $O_1, \dots, O_7$  (rechts)

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

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

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

Knoten werden in Tiefensuch-Reihenfolge betreten.

### Eigenschaften Mark/Sweep

- benötigt 2 Bit Markierung pro Zelle, aber keinen weiteren Zusatzspeicher
- Laufzeit für mark  $\sim | \text{lebender Speicher} |$
- Laufzeit für sweep  $\sim | \text{gesamter Speicher} |$
- 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:  $0 \leq \text{scan} \leq \text{free} \leq N$

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

Invarianten:

- $\text{scan} \leq \text{free}$
- 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/edu/ws09/pps/fohlen/main/node78.html>
- Aufgabe: <http://www.imn.htwk-leipzig.de/~waldmann/edu/ws09/pps/fohlen/main/node79.html>

# 15 Zusammenfassung

## Semantik definiert durch...

- Interpretation
  - funktional (CBV, CBN), Fixpunkte
  - imperativ (Speicher)
  - Ablaufsteuerung (Continuations)
- Transformation
  - desugaring (let  $\rightarrow$  lambda)
  - assignment conversion
  - CPS transformation
  - closure passing, lifting

```
git clone git://dfa.imn.htwk-leipzig.de/var/www/dcpl
```