

# Compilerbau Vorlesung Wintersemester 2008

Johannes Waldmann, HTWK Leipzig

20. November 2009



# 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
- ▶ syntaxgesteuerte Übersetzung
- ▶ Zwischencode-Erzeugung
- ▶ Laufzeitumgebungen
- ▶ Zielcode-Erzeugung

# Inhalt (Einzelheiten)

- ▶ lexikalische und syntaktische Analyse
  - ▶ algebraische Datentypen (= Bäume)
  - ▶ Funktionen höherer Ordnung (= Entwurfsmuster)
  - ▶ Parser-Kombinatoren
- ▶ syntaxgesteuerte Übersetzung
  - ▶ (konkrete) Interpretation
  - ▶ Typen (abstrakte Interpretation)
  - ▶ Codeerzeugung (abs. Int.)

# 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

- ▶ Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman: *Compilers: Principles, Techniques, and Tools (2nd edition)* Addison-Wesley, 2007, ISBN-13: 9780321493453  
<http://dragonbook.stanford.edu/>
- ▶ Implementierungssprache Haskell
  - ▶ allgemein: Tutorials, Referenzen  
<http://www.haskell.org/>
  - ▶ speziell Compilerbau: siehe  
<http://thread.gmane.org/gmane.comp.lang.haskell.general/16379/focus=43340>

# Organisation

- ▶ pro Woche eine Vorlesung, eine Übung.  
(sinnvolle Termine?)
- ▶ Prüfungszulassung:
  - ▶ Hausaufgaben (klein),
  - ▶ Projekt (mittelklein)
- ▶ Prüfung: Projektverteidigung  
(enthält auch Compilerbau-Fragen außerhalb des Projektes)

# Hausaufgaben

- ▶ Haskell-Umgebung installieren (ghc-6.8.3)  
(erst binary, dann from source)
- ▶ API-Docs, Tutorials buchmarken
- ▶ Radio hören:  
`http://www.se-radio.net/podcast/2008-08/episode-108-simon-peyton-jones-functional-programming`
- ▶ Quiz: welche Verbindung besteht zwischen Haskell und Perl?

# Entwurfsmuster Compositum

```
interface Node<K> { K key() ; List<Node<K>> children() ; }

class Branch<K> implements Node<K> {
    K key; K key() { return this.key; }
    Node<K> left ; Node<K> right;
    List<Node<K>> children() {
        return new Arrays.asList (new Node<K>[left, right]);
    }
}

class Leaf<K> implements Node<K> { .. }
```

# Algebraischer Datentyp

```
data Node k
  = Branch { key :: k, left :: Node k, right :: N
  | Leaf   { key :: k }
```

```
children :: Node k -> [ Node k ]
children = \ n -> case n of
  Branch {} -> [ left n, right n ]
  Leaf   {} -> [ ]
```

# Termalgebra

Eine (“ranked”) *Signatur*  $\Sigma$  ist eine Menge von Funktionssymbolen mit einer Abbildung  $\text{arity} : \Sigma \rightarrow \mathbb{N}$ .

Die Menge  $\text{Term}(\Sigma)$  ist die kleinste Menge  $T$  mit:

- ▶ für alle  $f \in \Sigma$ :  $a = \text{arity}(f)$ : für alle  $t_1, \dots, t_a \in T$ :  $f(t_1, \dots, t_a) \in T$ .

(Das ist eine induktive Definition. Wo ist der Induktionsanfang?)

# Termalgebra (mehrsortig)

Menge  $S = \{S_1, \dots, S_k\}$  von *Sorten*

“sortierte” Signatur  $\Sigma$  is Menge von Symbolen und Abbildung

$\text{sort} : \Sigma \rightarrow S^+$

Wenn  $f \in \Sigma$ ,  $\text{sort}(f) = [S_1, \dots, S_k, T]$ ,  $t_1 \in \text{Term}(\Sigma, S_1), \dots, t_k \in \text{Term}(\Sigma, S_k)$ , dann  $f(t_1, \dots, t_k) \in \text{Term}(\Sigma, T)$ .

(Funktionssymbol  $\approx$  Konstruktor)

# Beispiele f. Algebr. Datentypen

(d. h. mehrsortige Algebren)

- ▶ Binäre Bäume
- ▶ Listen
- ▶ Wahrheitswerte
- ▶ “rose tree”

# Funktionen auf Alg. Datentypen

```
data Node k = Leaf    { key :: k }  
            | Branch { key :: k, left :: Node k, right :: Node k }
```

(vollst.) Fallunterscheidung nach den Konstruktoren:

```
children :: Node k -> [ Node k ]  
children = \ n -> case n of  
    Branch {} -> [ left n, right n ]  
    Leaf    {} -> [ ]
```

alternative Schreibweise: *pattern matching*

```
children = \ n -> case n of  
    Branch { left = l, right = r } -> [ l, r ]  
    Leaf   { } -> [ ]
```

# Funktionen

```
children = \ n -> ...  
norm = \ x y -> sqrt ( x^2 + y^2 )
```

rechts steht eine (anonyme) Funktion (“delegate”)

alternative Schreibweise:

```
children n = ...  
norm x y = sqrt ( x^2 + y^2 )
```

## Blöcke (let)

```
sqrt (x^2 + y^2 )
```

```
let { a = x^2 ; b = y^2 ; c = a + b } in sqrt c
```

```
sqrt ( let { a = x^2 ; b = y^2 ; c = a + b } in c )
```

Übersetzung (für Geradeprogramme)

```
let { v_1 = a_1 .. v_n = a_n } in b
```

```
(\ v_1 .. v_n -> b) a_1 .. a_n
```

... das ist nicht die ganze Wahrheit

# Layout-Regeln

nach bestimmten Schlüsselwörtern (let, of, ...):

- ▶ explizite Gruppierung: { .. ; .. }
- ▶ implizite Gruppierung durch Tiefe der Einrückung: wenn kein { folgt, dann wird es eingesetzt und für die nächsten Zeilen gilt:
  - ▶ gleich tief: Semikolon davorsetzen
  - ▶ weniger tief: Klammer zu davorsetzen

## Beispielcode (Übungen)

```
data Tree k
  = Branch { left :: Tree k, key :: k, right ::
            | Leaf   { }
  deriving Show

-- | vollständiger binärer Baum der Höhe h
full :: Integer -> Tree Integer
full h =
  if h > 0
  then Branch { left = full (h-1), key = h, right
  else Leaf {}

-- | Fibonacci-Baum
fib :: Integer -> Tree Integer
fib h =
  case h > 0 of
    True  -> Branch { left = fib (h-1), key = h,
    False -> Leaf {}
```

# Abstrakte Syntax (Ausdrücke)

```
data Expression = Constant Integer
                | Plus    Expression Expression
                | Times  Expression Expression
```

```
evaluate :: Expression -> Integer
evaluate e = case e of ...
```

```
instance Show Expression where
    show e = case e of ...
```

# Abstrakte Syntax (Anweisungen)

```
data Identifier = Identifier String
```

```
data Statement = Assign Identifier Expression  
              | Iterate Expression Statement
```

```
type Program = [ Statement ]
```

# Semantik (Anweisungen)

```
import qualified Data.Map as M

type Environment = M.Map Identifier Integer

execute :: Environment -> Statement -> Environment

run :: Program -> Environment
```

## Semantik (mit Maybe)

```
data Expression = ... | Reference Identifier
evaluate
  :: Environment -> Expression -> Integer ??
  :: Environment -> Expression -> Maybe Integer
evaluate e x = case x of
  Reference i -> M.lookup i e
  Plus l r -> case ( evaluate e l ) of
    Nothing -> Nothing
    Just a -> case ( evaluate e r ) of
      Nothing -> Nothing
      Just b -> Just ( a + b )
```

das schöner hinschreiben mit *Monaden* und *do-Notation*

## Typklassen, Beispiel: Show

```
class Show a where { show :: a -> String }  
  
data Foo = Foo { foo :: Integer }  
  
instance Show Foo where  
    show f = "F" ++ show ( foo f )  
  
rshow :: Show a -- Typconstraint  
    => a -> String  
rshow x = reverse ( show x )
```

# Vergleich Typklassen/Interfaces

```
interface I { } | class I where { }
```

## Polymorphie auf der Argumentseite:

```
boolean foo (I x) { ... } // universell  
foo :: I t => t -> Bool   -- universell
```

## auf der Resultatseite:

```
I bar (String s) { ... } // existenziell  
bar :: I t => String -> t  -- universell
```

# Generische Instanzen

```
instance Show a => Show ( Maybe a ) where
    show m = case m of
        Nothing -> "Nothing"
        Just x   -> "Just" ++ show x
```

das kann der Compiler selbst:

```
data Maybe a = ...
    deriving ( Show )
```

# Beziehungen zwischen Klassen

```
class Eq a where  
    (==) :: a -> a -> Bool
```

```
class Eq a => Ord a where  
    (<) :: a -> a -> Bool
```

# Konstruktorklassen (Functor)

```
class Functor c where
  fmap :: ( a -> b ) -> ( c a -> c b )
```

```
instance Functor [] where
  fmap f l = case l of
    [] -> []
    x : xs -> f x : fmap f xs
```

```
instance Functor Maybe where ...
```

```
instance Functor Tree where ...
```

**unterscheide von:**

```
instance Show a => Show (Tree a) where ...
```

# Gesetze für Funktoren (I)

Kategorie der Datentypen:

- ▶ Objekte (Punkte): Typen
- ▶ Morphismen (Pfeile): (einstellige) Funktionen

Funktor  $F$  von Kategorie  $A$  nach Kategorie  $B$

- ▶  $F_{\text{ob}} : (\text{Objekt in } A) \mapsto (\text{Objekt in } B)$
- ▶  $F_{\text{mo}} : (\text{Pfeil } x \rightarrow y \text{ in } A) \mapsto (\text{Pfeil } F_{\text{ob}}(x) \rightarrow F_{\text{mo}}(y) \text{ in } B)$

so daß

- ▶  $F_{\text{mo}}(\text{id}) = \text{id}$
- ▶  $F_{\text{mo}}((x \rightarrow y) \circ (y \rightarrow z)) = F_{\text{mo}}(x \rightarrow y) \circ F_{\text{mo}}(y \rightarrow z)$

# Gesetze für Funktoren (II)

Für Instanzen der Typklasse Functor muß gelten:

```
fmap id == id
```

```
fmap (f . g) = fmap p . fmap g
```

Aufgaben: finde falsche Funktor-Instanzen

- ▶ für `List`, `Maybe`, `Tree`, die
  - ▶ keines
  - ▶ nur das erste
  - ▶ nur das zweite

Gesetz erfüllen (das sind 9 Aufgaben)

# Die Konstruktorklasse Monad

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

```
instance Monad Maybe where
  return = \ x -> Just x
  m >>= f = case m of
    Nothing -> Nothing
    Just x   -> f x
```

# Anwendung der Maybe-Monade

```
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)
```

# Do-Notation für Monaden

Original:

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

do-Notation:

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

# List als Monade

```
instance Monad [] where
  return = \ x -> [x]
  m >>= f = case m of
    []      -> []
    x : xs -> f x ++ ( xs >>= f )

do a <- [ 1 .. 4 ]
   b <- [ 2 .. 3 ]
   return ( a * b )
```

# Monaden mit Null

```
import Control.Monad ( guard )  
do a <- [ 1 .. 4 ]  
    b <- [ 2 .. 3 ]  
    guard $ even (a + b)  
    return ( a * b )
```

---

```
[ 1 .. 4 ] >>= \ a ->  
    [ 2 .. 3 ] >>= \ b ->  
        if ( even (a+b) )  
            then return ( a * b ) else mzero
```

---

```
class Monad m => MonadPlus m where  
    mzero :: m a ; ...  
instance MonadPlus [] where mzero = []
```

# Aufgaben zur List-Monade

- ▶ Pythagoreische Tripel aufzählen
- ▶ Ramanujans Taxi-Aufgabe ( $a^3 + b^3 = c^3 + d^3$ )
- ▶ alle Permutationen einer Liste
- ▶ alle Partitionen einer Zahl  
(alle ungeraden, alle aufsteigenden)

Hinweise:

- ▶ allgemein: Programme mit `do`, `<-`, `guard`, `return`
- ▶ bei Permutationen benutze:

```
import Data.List ( inits, tails )  
    (xs, y:ys ) <- zip (inits l) (tails l)
```

# Gesetze für Monaden (I)

Monade  $m \Rightarrow$  Kleisli-Kategorie von  $m$ :

- ▶ Objekte: Typen
- ▶ Pfeile: von  $a$  nach  $m\ b$

Komposition der Pfeile ist so definiert:

```
comp :: Monad m
      => (a -> m b) -> (b -> m c) -> (a -> m c)
comp f g = \ x -> do y <- f x ; g y
```

Eine Monade muß erfüllen:

- ▶ `return` ist neutral für `comp`
- ▶ `comp` ist assoziativ

## Gesetze für Monaden (II)

- ▶ Gesetze für die Instanzen von List und Maybe überprüfen
- ▶ Gibt es eine korrekte Instanz für binäre Bäume?
  - ▶ mit Schlüsseln in Branch und nicht in Leaf
  - ▶ mit Schlüsseln in Leaf und nicht in Branch
- ▶ Gib eine inkorrekte Instanz für List, Maybe, Tree, die ...
  - ▶ typkorrekt ist
  - ▶ genau eines der Gesetze verletzt

# Funktoren und Monaden

Jede Monade ist ein Funktor

```
class Functor m => Monad m where ...
```

Das `fmap` aus `Functor` kann man darstellen durch `>>=`, `return` bzw. durch `do`, `<-`, `return`

Vervollständige und beweise die Gesetze

$$\begin{aligned} (\text{fmap } f \text{ m}) >>= g & \quad == \quad \text{m } >>= (\dots) \\ \text{m } >>= (\text{fmap } g \text{ . } f) & \quad == \quad \dots \end{aligned}$$

# Die IO-Monade

```
data IO -- abstract
```

```
readFile :: FilePath -> IO String
```

```
putStrLn :: String -> IO ()
```

```
instance Functor IO ; instance Monad IO
```

Alle „Funktionen“, deren Resultat von der Außenwelt (Systemzustand) abhängt, haben Resultattyp `IO ...`. Am Typ einer Funktion erkennt man ihre möglichen (schädlichen) Wirkungen bzw. deren garantierte Abwesenheit. Wegen der Monad-Instanz: benutze `do`-Notation

```
do cs <- readFile "foo.bar" ; putStrLn cs
```

# Die Zustands-Monade

Wenn man nur den Inhalt einer Speicherstelle ändern will, dann braucht man nicht `IO`, sondern es reicht `State`.

```
import Control.Monad.State

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

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

**Aufgabe:** wie könnte die Implementierung aussehen?

```
data State s a = ??
instance Functor ( State s ) where
instance Monad ( State s ) where
```

# Parser als Monade

nichtdeterministischer Parser über Eingabestrom von Zeichen (Token)  $c$  und mit Resultattyp  $c$ :

```
data Parser c a =  
    Parser ( [c] -> [ (c, [c]) ] )
```

- ▶ Instanz für Functor, Monad
- ▶ Beispiel-Parser, Aufrufe davon

# Parser als Monade (Implementierung)

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

beachte: das *do* gehört zur List-Monade

# 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 -> []
```

## 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 $ read [ c ]
```

# Kombinatoren für Parser (I)

- ▶ Folge (and then) (ist *bind* aus der Monade)
- ▶ Auswahl (or)

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

- ▶ 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`

# Eindeutigkeit/Determinismus

Beispiel  $S \rightarrow S + S \mid S * S \mid Z$

Auswege:

- ▶ Grammatik ändern („eindeutig machen“)
- ▶ Grammatik bleibt, Operator-Vorrang (Präzedenz, Assoziativität) festlegen

# Operator-Präzedenz-Parser

```
data Operator a = Operator
  { precedence :: Int
  , semantics  :: a -> a -> a }
operator_precedence_parser
  :: Map String ( Operator a )
  -> Parser Char a -> Parser Char a

arithmetic :: Parser Char Integer
arithmetic = operator_precedence_parser ( M.fromList
  [ ( "+", Operator { precedence = 1
                    , semantics = (+) } )
  , ( "*", Operator { precedence = 2
                    , semantics = (*) } )
  ] )
  ( zahl <|> parens arithmetic )
```

# Op-P-P (Implementierung)

Benutzt zwei Keller:

- ▶ Werte
- ▶ Operatoren

Bei Lesen eines

- ▶ Atoms: push auf Wertkeller
- ▶ Operators:
  - ▶ ggf. Ausführung von top-of-opkeller
  - ▶ push auf Operatorkeller

# 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

# Linq: Beispiel

Linq = language integrated query

```
Random random = ... ; int block = ... ;
IEnumerable<Pair<V,G>> store =
    from Pair<V,G> p in this.pool
    group p by p.first into g
    orderby g.Key
    from q in g.Distinct()
        .OrderBy(p => random.Next())
        .Take(block)
    select q;
this.pool = store.Take(this.size).ToList();
```

# Linq und Monaden: Notation

## Haskell:

```
test :: [ Integer ]
test = do
  x <- [ 1 .. 5 ]
  guard $ odd x
  y <- [ 1 .. x ]
  return $ x * y
```

**Linq:** <http://msdn.microsoft.com/en-us/library/bb397676.aspx>

**Beispiele:** <http://msdn.microsoft.com/en-us/vcsharp/aa336746.aspx> (Vorsicht: inzwischen einige Methoden umbenannt)

## C# (LINQ):

```
IEnumerable<int> test =
    from x in Enumerable.Range(1, 5)
    where 0 != x % 2
    from y in Enumerable.Range(1, x)
    select x * y;
```

# Linq und Monaden: Semantik

## Haskell:

```
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
[1 .. 5] >>= \ x -> [ 1 .. x ]
```

## Linq:

```
using System.Linq;
public static IEnumerable<TResult>
    SelectMany<TSource, TResult>(
        this IEnumerable<TSource> source,
        Func<TSource, IEnumerable<TResult>> selector)

Enumerable.Range(1, 5)
    .SelectMany(x => Enumerable.Range(1, x))
```

# Linq und Haskell: gut zu wissen

Haskell	Linq
<code>[a]</code>	<code>IEnumerable&lt;a&gt;</code>
<code>map</code> und <code>bind (&gt;&gt;=)</code>	<b>Select</b> und <b>SelectMany</b>
<code>take</code> und <code>drop</code>	<b>Take</b> und <b>Skip</b>
<code>(++)</code> und <code>reverse</code>	<b>Concat</b> und <b>Reverse</b>
<code>fold</code>	<b>Aggregate</b>
<code>[x .. x+n-1]</code>	<code>Enumerable.Range(x, n)</code>
<code>forM_ xs print</code>	<code>xs.ToList().ForEach(x =&gt; Console.WriteLine(x))</code>

# Projekte Compilerbau (autotool)

- ▶ **Type-Checking erweitern**

`https://autotool.imn.htwk-leipzig.de/cgi-bin/Trial.cgi?topic=TypeCheck-Quiz`

- ▶ Methoden nicht nur static
- ▶ Vererbung
- ▶ generische Polymorphie

- ▶ **neue Sprachen (Parser, Printer, Interpreter)**

- ▶ **Brainfuck** `http://dfa.imn.htwk-leipzig.de/bugzilla/show\_bug.cgi?id=106`
- ▶ **Intercal** `http://dfa.imn.htwk-leipzig.de/bugzilla/show\_bug.cgi?id=107`
- ▶ **Malbolge** `http://dfa.imn.htwk-leipzig.de/bugzilla/show\_bug.cgi?id=174`

# Projekte Compilerbau (Refactoring)

für Haskell und/oder Java:

- ▶ Quelltext lesen
- ▶ Muster im Syntaxbaum erkennen (ähnlicher Code)
- ▶ Refactoring vorschlagen (Unterprogramm mit Parametern)

Für Haskell benutze `ghc-modul Language.Haskell`, es gibt auch `language-c` (auf Hackage), danach könnte man `Language.Java` aufbauen?

# Interpreter

mit Bestandteilen:

- ▶ syntaktische Analyse (erzeugt AST)
- ▶ Interpretation des AST (execute, eval)

Typprüfung:

- ▶ dynamisch (während Interpretation) oder
- ▶ statisch (vor Interpretation)

# Umgebungen

Umgebung: Zuordnung von Variablenname zu

- ▶ Wert (bei Interpretation)
- ▶ Adresse (bei Kompilation)
- ▶ weitere Informationen (z. B. Typ)

für Sprachen mit Blockstruktur: benötigen geschachtelte (nested) Umgebungen.

# Unterprogramme

Definition UP:

- ▶ Schnittstelle (formale Parameterliste, Rückgabebetyp)
- ▶ Implementierung (Block-Statement)

Realisierung in Interpreter/Compiler:

- ▶ Syntax
  - ▶ abstrakt (AST)
  - ▶ konkret (Parser)
- ▶ Semantik
  - ▶ Argumente übergeben
  - ▶ Block ausführen
  - ▶ Resultat zurück

# Lokale Funktionen

```
{ var a = 5;
  var f = (x => a*x);
  print f(3);
  { var a = 7;
    print f(3);
  }
}
```

bei Auswertung von  $f(3)$ :

- ▶ dynamische Bindung: verwendet inneres  $a(= 7)$ ,
- ▶ statische Bindung: verwendet äußeres  $a(= 5)$ .

nur Programme mit statischer Bindung können statisch typ-geprüft werden.

# Implementierung der statischen Bindung

```
data Object = .. | OFunction { args :: [ Name ]
    , body :: Expression , env  :: Env Object }
eval env exp = case exp of ...
  Lambda { args = a, body = b } -> return $
    OFunction { args = a, body = b, env = env }
  Apply f xs -> do
    ff <- eval env f
    vs <- mapM ( eval env ) xs
    let env' = Env.new ( Object.env ff )
        $ zip ( Object.args ff ) vs
    eval env' ( Object.body ff )
```

# Lokale Funktionen in C#

```
using System;
class nest {
    public static void Main () {
        int a = 5; Func<int,int> b = (x => a*x);
        { a = 7; Console.WriteLine(b(3)); }
    }
}
```

Benutzt wird

- ▶ statische Bindung
- ▶ für die Adresse von  $a$

# Lokale Funktionen in Java

```
class nest {  
    interface Func<A,B> { B apply(A x); }  
    public static void main (String [] args) {  
        { int a = 5;  
            Func<Integer,Integer> b = new Func<Integer,Integer>()  
            {  
                public Integer apply(Integer x) { return a*x; }  
            };  
            { a = 7; System.out.println(b.apply(3)); }  
        }  
    }  
}
```

die nicht-lokale Variable *a* mu als final deklariert werden,  
dadurch ist die Zuweisung (*a* = 7) verboten.

# Currying

```
Func<int, Func<int, int>>  
    times = x => (y => x*y);  
Console.WriteLine( times (3) (4) );
```

Multiplikation ist eigentlich zweistellig, diese Version ist durch „Currying“ einstellig.

(benannt nach Haskell B. Curry, 1900–1982,

<http://www-history.mcs.st-andrews.ac.uk/>

[Biographies/Curry.html](http://www-history.mcs.st-andrews.ac.uk/Biographies/Curry.html))

# Rekursive Bindungen

```
int y = 1 + y/2; // geht nicht
```

```
// geht doch:
```

```
Func<int,int> f = (x => x > 0 ? x * f(x-1) : 1);  
Console.WriteLine (f(3));
```

# Motivation

gebundene (lokale) Variablen in der ...

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

# Der Lambda-Kalkül

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

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

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

Im reinen Lambda-Kalkül gibt es *nur* Funktionen—keine Zahlen

# Lambda-Terme

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

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

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, Klammern weglassen:

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

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

# Mehrstellige Funktionen

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.

um die zu beschreiben, genügt also ein Kalkül für einstellige Funktionen.

(Beispiel)

# Ableitungen (Ansatz)

Absicht: Relation  $\rightarrow$  auf  $\Lambda$  (Ein-Schritt-Ersetzung):

- ▶  $(\lambda x.B)A \rightarrow B[x := A]$  (Vorsicht)
- ▶  $F \rightarrow F' \Rightarrow (FA) \rightarrow (F'A)$
- ▶  $A \rightarrow A' \Rightarrow (FA) \rightarrow (FA')$
- ▶  $B \rightarrow B' \Rightarrow \lambda x.B \rightarrow \lambda x.B'$

was soll  $(\lambda x.B)[x := 3 + 4]$  bedeuten?

ist das sinnvoll:

$(\lambda x.(\lambda y.xyx))(yy) \rightarrow (\lambda y.yx)[x := (yy)] = \lambda y.y(yy)$

das freie  $y$  wird fälschlich gebunden

# Das falsche Binden von Variablen

(voriges Beispiel in C++):

Diese Programme sind *nicht* äquivalent:

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

# Gebundene Umbenennungen

wir dürfen  $(\lambda x.B)A \rightarrow B[x := A]$  nur ausführen, wenn  $x$  nicht in  $A$  frei vorkommt.

falls doch, müssen wir  $\lambda x.B$  in  $\lambda y.B[x := y]$  umbenennen, wobei  $y$  weder in  $A$  frei noch in  $B$  überhaupt vorkommt.

(Beispiel) (Def.  $FV(t)$ )

eine solche gebundene Umbenennung in einem Teilterm heißt  $\alpha$ -Konversion.

$\alpha$ -konvertierbare Terme sind äquivalent (verhalten sich gleich bzgl. Ableitungen)

(Beispiel)

mit o.g. Bedingung ergibt sich eine vernünftige Relation  $\rightarrow$  ( $\beta$ -Reduktion).

(Beispiel-Ableitungen)

# Eigenschaften der Reduktion

→ auf  $\Lambda$  ist

- ▶ konfluent,
- ▶ aber nicht terminierend.  
 $W = \lambda x.xx, \Omega = WW.$
- ▶ es gibt Terme mit Normalform und unendlichen Ableitungen,  $KI\Omega$  mit  $K = \lambda xy.x, I = \lambda x.x$

Einige Eigenschaften klingen erstaunlich: z. B. jeder Term  $F$  besitzt einen Fixpunkt  $A$ , d. h.  $FA \rightarrow^* A$ .

Den kann man sogar ausrechnen: es gibt  $R$  mit  $F(RF) \rightarrow^* RF$ .

# Rechnen mit simulierten Zahlen

Church-Kodierung der natürlichen Zahlen (Alonzo Church, 1903–1995, <http://www-history.mcs.st-andrews.ac.uk/Biographies/Church.html>)

- ▶ Idee:  $[n] = \lambda fx.f^n(x)$
- ▶ Null:  $[0] = \lambda fx.x (= K)$ ,
- ▶ Nachfolger:  $[n + 1] = \lambda fx.[n]f(fx)$ , also  $\text{succ} = \lambda nfx.nf(fx)$
- ▶ Dafür sind Nachfolger, Vorgänger, Null-Test definierbar.

mit Fixpunktsatz gibt es auch Rekursion (beliebige Schleifen), also ist jede Turing-berechenbare Funktion auch Lambda-berechenbar (und umgekehrt).

Übung: Addition, Multiplikation, Potenzieren  
(tatsächlich ist das Modell älter als die Turing-Maschine)

# Erweiterungen, Anwendungen

ausgehend vom einfachen Lambda-Kalkül baut man:

- ▶ Typsysteme (jeder Ausdruck, jede Variable besitzt Typ)
- ▶ eingebaute Datentypen (außer Funktionen) und Operationen, z. B. Zahlen
- ▶ effiziente Implementierung von Reduktionen (ohne die umständlichen Umbenennungen)

das bildet die Grundlage für

- ▶ exakte Analyse von Programmier/mathematischen/logischen Sprachen
- ▶ Implementierung von Sprachen und Refactoring-Werkzeugen

# Lambda-Kalkül und Computeralgebra

- ▶ Kalkül beschreibt Rechnen mit Ausdrücken mit gebundenen Variablen, diese kommen in CAS vor.
- ▶ die Erkenntnisse aus dem Kalkül werden in verschiedenen CAS mit verschiedener Konsequenz angewendet (leider).
- ▶ Probleme beginnen damit, daß Variablenbindungen schon gar nicht korrekt notiert werden
- ▶ ... das ist nur ein getreues Abbild entsprechender Probleme in der Mathematik (solche Fehler heißen dort aber Konventionen)

# Grundlagen

- ▶ (bisher) konkrete Interpretation (Programm  $\mapsto$  Daten)
- ▶ (jetzt) abstrakte Interpretation (Programm  $\mapsto$  Typ)
- ▶ (später) abstrakte Interpretation (Programm  $\mapsto$  Zielprogramm)

Plan: definiere

```
data Type =  
  TInteger | TBool | TFunction [ Type ] Type
```

und ersetze in Eval.hs überall `Env Object` durch `Env Type`

# Deklaration und Inferenz

- ▶ Deklaration:

Programmierer gibt für jeden Bezeichner Typ an, Compiler prüft auf Konsistenz.

Prinzip:  $(f :: a \rightarrow b \wedge x :: a) \Rightarrow f(x) :: b$

- ▶ Inferenz:

Compiler berechnet (= inferiert) Typ der Bezeichner

Prinzip: aus `var x = 5;` folgt `x :: int`

# Polymorphie

was ist der Typ von  $t$  hier:

```
var t = f => x => f (f (x));  
print t(x => x+1) (0);  
print t(t) (x => x+1) (0);
```

Lösung:  $\forall a : (a \rightarrow a) \rightarrow (a \rightarrow a)$   
benötigen

- ▶ Typvariablen ( $a$ )
- ▶ Bindungen dafür ( $\forall$ )

# Inferenz Polymorpher Typen

- ▶ jeder neue Bezeichner bekommt eine neue Typvariable
- ▶ jede Benutzung eines Bezeichners erzeugt ein Typconstraint
- ▶ (Teil-)Programm bestimmt Constraintsystem
- ▶ Lösung ist Zuordnung: Bezeichner  $\rightarrow$  Typ
- ▶ Lösung existiert? ist eindeutig?

das Constraint-Lösen ist hier das *Unifizieren*

# Unifikation—Begriffe

- ▶ Signatur  $\Sigma = \Sigma_0 \cup \dots \cup \Sigma_k$ ,
- ▶  $\text{Term}(\Sigma, V)$  ist kleinste Menge  $T$  mit  $V \subseteq T$  und  
 $\forall 0 \leq i \leq k, f \in \Sigma_i, t_1 \in T, \dots, t_i \in T : f(t_1, \dots, t_i) \in T$ .
- ▶ Substitution: partielle Abbildung  $\sigma : V \rightarrow \text{Term}(\Sigma, V)$ , so daß kein  $v \in \text{dom } \sigma$  in  $\text{img } \sigma$  vorkommt,
- ▶ Substitution  $\sigma$  auf Term  $t$  anwenden:  $t\sigma$
- ▶ Produkt von Substitutionen: so definiert, daß  
 $t(\sigma_1 \circ \sigma_2) = (t\sigma_1)\sigma_2$

# Unifikation—Definition

## Unifikationsproblem

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

allgemeinst = minimal bzgl. der Prä-Ordnung

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

Satz: jedes Unifikationsproblem ist entweder gar nicht oder bis auf Umbenennung eindeutig lösbar

# Unifikation—Algorithmus

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

- ▶  $s$  ist Variable: ...
- ▶  $t$  ist Variable: symmetrisch
- ▶  $s = f(s_1, s_2)$  und  $t = g(t_1, t_2)$ : ...

Bemerkungen:

- ▶ korrekt, übersichtlich, aber nicht effizient,
- ▶ es gibt Unif.-Probl. mit exponentiell großer Lösung,
- ▶ eine komprimierte Darstellung davon kann man aber in Polynomialzeit ausrechnen.

# Beispiel zur Polymorphie (I)

- ▶ Was ist der allgemeinste Typ von

```
f1 g x = length $ g [g x]
```

- ▶ Beispiel für einen typkorrekten Aufruf von `f1`?
- ▶ Ist dieser Aufruf typkorrekt?

```
f1 ( \ x -> x ) 0
```

## Beispiel zur Polymorphie (II)

`f2 ( \ x -> x ) 0 für`

`f2 g x = length $ g [g x]`

**geht (nur) so:**

`f2 :: (forall a. a->a) -> Int -> Int`

**das ist nicht das gleiche wie:**

`f1 :: forall a . (a->a) -> Int -> Int`

`f1 g x = length $ g [g x]`

# System F

- ▶ einfach getypter Lambda-Kalkül:

$$A \Rightarrow x \mid AA \mid \lambda x : T.A$$
$$T \Rightarrow \mathbb{N} \mid T \rightarrow T$$

Variable  $x$ , Type  $T$ , Term  $A$

- ▶ polymorph getypter Lambda-Kalkül (System  $F$ )  
(Jean-Yves Girard, John C. Reynolds)

$$A \Rightarrow \dots \wedge X.A \mid A[T], \quad T \Rightarrow \dots X \mid \forall X.T$$

Typvariable  $X$

- ▶ Hindley-Milner (benutzt in Standard-Haskell): Typ-Forall nur außen erlaubt

# Rang-(2/N)-Polymorphie in GHC

[http://haskell.org/ghc/docs/latest/html/users\\_guide/other-type-extensions.html](http://haskell.org/ghc/docs/latest/html/users_guide/other-type-extensions.html)

```
{-# language Rank2Types #-}
f2 :: (forall a. a->a) -> Int -> Int
f4 :: Int -> (forall a. a -> a)

{-# language RankNTypes #-}
f3 :: ((forall a. a->a) -> Int)
     -> Bool -> Bool
```

Rang:  $1 + \max.$  Tiefe der  $(\square \rightarrow *)$ -Kontexte für Typ-Forall

# Eigenschaften polym. Typsyst.

- ▶ Termination:  
getypte Programme terminieren
- ▶ Rekonstruktion:  
es gibt Algorithmus, der Typ eines ungetypten Programms  
ausrechnet oder ablehnt, falls es keinen Typ besitzt  
(Rang 2: ja, beliebiger Rang: nein)

# Unifikation höherer Ordnung

Satz: Typrekonstruktion für System F (beliebiger Rang) ist unentscheidbar.

Beweis: Reduktion von Unifikation höherer Ordnung.

Def HO-Unifikation:

- ▶ Eingabe: einfach getypte Lambda-Terme  $s, t$ ,
- ▶ Ausgabe: getypte Substitution  $\sigma$  mit  $s\sigma \leftrightarrow^* t\sigma$

Satz: HO-Unifikation ist unentscheidbar.

Beweis: Reduktion von Hilberts 10. Problem (ganzzahlige Lösungen polynomieller Gleichungssysteme), Codierung mit Church-Zahlen.

Gilles Dowek: <http://www.lix.polytechnique.fr/~dowek/Publi/unification.ps>

# Typen und Terme

- ▶ (Term abhängig von Term, Funktion auf Daten)
- ▶ Typ abhängig von Typ (Typkonstruktoren: Functor, Monad)
- ▶ Term abhängig von Typ (System F)
- ▶ Typ abhängig von Term (dependent Types: Coq, PVS, ...)

(der Lambda-Würfel, Barendregt 1991)

<http://www.cs.ru.nl/~henk/>

# Plan

- ▶ konkrete/abstrakte Syntax, Typprüfungen  
 $x_0.f(x_1, \dots, x_n) \Rightarrow f(x_0, x_1, \dots, x_n)$
- ▶ Daten-Layout (Objekte)
- ▶ Laufzeit-Polymorphie (Methoden)
- ▶ Typconstraints

# Daten-Layout

# Laufzeit-Polymorphie

```
class A { void m () { } }  
class B extends A { void m () { } }  
  
... A x = new B (); x.m(); ...
```

- ▶ Methodentabelle
- ▶ Einfach-Vererbung
- ▶ Mehrfach-Vererbung

# Typconstraints (Haskell)

```
class C t where m :: t -> Int
data A ; instance C A where m = ...
data B ; instance C B where m = ...
```

```
f :: C t => t -> Int ; f x = m x * 2
```

## Implementierung durch *Dictionaries*

```
data CDict t = CDict { m :: t -> Int }
f :: CDict -> t -> Int ; f d x = m d x * 2
```

# Vergleich Constraints/Interfaces

- ▶ Interface ist einstelliges Constraint
- ▶ allgemeines Constraint ist aber mehrstellig

Bsp. Autotool:

```
class Aufgabe i b where
  explain :: i -> Text
  initial :: i -> b
  check   :: i -> b -> Bool
```

vgl. <http://141.57.11.163/cgi-bin/cvsweb/tool/src/Challenger/Partial.hs?rev=1.29>

# (Co-)Varianz von Typparametern

- ▶ **aus** `B extends A` folgt nicht  
`List<B> extends List<A>`
- ▶ **in/out** an Typparametern in C# 4