Compilerbau
Vorlesung
Wintersemester 2008

Johannes Waldmann, HTWK Leipzig

20. November 2009

Einleitung

Beispiel

ıgabe (≈ 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

Inhalt (Einzelheiten)

xikalische und syntaktische Analyse

algebraische Datentypen (= Bäume)
Funktionen höherer Ordnung (= Entwurfsmuster)
Parser-Kombinatoren

yntaxgesteuerte Übersetzung

(konkrete) Intepretation
Typen (abstrakte Intepretation)
Codeerzeugung (abs. Int.)

Sprachverarbeitung

ıit Compiler:

Quellprogramm → Compiler → Zielprogramm
Eingaben → Zielprogramm → Ausgaben

ıit Interpreter:

Quellprogramm, Eingaben → Interpreter → Ausgaben

lischform:

Quellprogramm → Compiler → Zwischenprogramm
Zwischenprogramm, Eingaben → virtuelle Maschine → Ausgaben

Compiler und andere Werkzeuge

uellprogramm

räprozessor → modifiziertes Quellprogramm

ompiler → Assemblerprogramm

ssembler → verschieblicher Maschinencode

inker, Bibliotheken → ausführbares
laschinenprogramm

Phasen eines Compilers

eichenstrom

xikalische Analyse → Tokenstrom

yntaktische Analyse → Syntaxbaum

emantische Analyse → annotierter Syntaxbaum

wischencode-Erzeugung → Zwischencode

ıaschinenunabhängige Optimierungen → Zwischencode

ielcode-Erzeugung → Zielcode

Methoden und Modelle

ɛxikalische Analyse: reguläre Ausdrücke, endliche utomaten

yntaktische Analyse: kontextfreie Grammatiken, ellerautomaten

emantische Analyse: Attributgrammatiken

ode-Erzeugung: bei Registerzuordnung: iraphenfärbung

Anwendungen von Techniken des Compilerbaus

nplementierung höherer Programmiersprachen

rchitekturspezifische Optimierungen (Parallelisierung, peicherhierarchien)

ntwurf neuer Architekturen (RISC, spezielle Hardware)

rogramm-Übersetzungen (Binär-Übersetzer, ardwaresynthese, Datenbankanfragesprachen)

oftware-Werkzeuge

Literatur

lfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. llman: *Compilers: Principles, Techniques, and Tools 2nd edition)* Addison-Wesley, 2007, ISBN-13: 780321493453
`ttp://dragonbook.stanford.edu/`

nplementierungssprache Haskell

- allgemein: Tutorials, Referenzen `http://www.haskell.org/`
- speziell Compilerbau: siehe `http://thread.gmane.org/gmane.comp.lang.haskell.general/16379/focus=43340`

Organisation

ro Woche eine Vorlesung, eine Übung.

sinnvolle Termine?)

rüfungszulassung:

- Hausaufgaben (klein),
- Projekt (mittelklein)

rüfung: Projektverteidigung

ɛnthält auch Compilerbau-Fragen außerhalb des rojektes)

Hausaufgaben

askell-Umgebung installieren (ghc-6.8.3)

ɛrst binary, dann from source)

PI-Docs, Tutorials buchmarken

adio hören:
`ttp://www.se-radio.net/podcast/2008-08/`
`pisode-108-simon-peyton-jones-functional-pro`

uiz: welche Verbindung besteht zwischen Haskell und erl?

Algebraische Datentypen

Entwurfsmuster Compositum

```
:erface Node<K> { K key() ; List<Node<K>> c

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

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

Algebraischer Datentyp

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

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

Termalgebra

e ("ranked") *Signatur* Σ ist eine Menge von ıktionssymbolen mit einer Abbildung $\mathrm{arity} : \Sigma \to \mathbb{N}$.
Menge $\mathrm{Term}(\Sigma)$ ist die kleinste Menge T mit:

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

ıs ist eine induktive Definition. Wo ist der ıktionsanfang?)

Termalgebra (mehrsortig)

nge $S = \{S_1, \dots, S_k\}$ von *Sorten*
rtierte" Signatur Σ is Menge von Symbolen und
bildung $\mathrm{sort} : \Sigma \to S^+$
nn $f \in \Sigma, \mathrm{sort}(f) = [S_1, \dots, S_k, T], t_1 \in$
$\mathrm{m}(\Sigma, S_1), \dots t_k \in \mathrm{Term}(\Sigma, S_k)$, dann
$, \dots, t_k) \in \mathrm{Term}(\Sigma, T)$.
nktionssymbol $\approx$ Konstruktor)

Beispiele f. Algebr. Datentypen

h. mehrsortige Algebren)

inäre Bäume

isten

/ahrheitswerte

ose tree"

Funktionen auf Alg. Datentypen

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

llst.) Fallunterscheidung nach den Konstruktoren:

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

rnative Schreibweise: *pattern matching*

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

Funktionen

```
ildren = \ n -> ...
:m = \ x y -> sqrt ( x^2 + y^2 )
```

hts steht eine (anonyme) Funktion ("delegate")

rnative Schreibweise:

```
ildren n = ...
:m x y = sqrt ( x^2 + y^2 )
```

Blöcke (let)

```
:t (x^2 + y^2 )

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

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

ersetzung (für Geradeausprogramme)

```
: { 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

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

xplizite Gruppierung: `{ .. ; .. }`

nplizite Gruppierung durch Tiefe der Einrückung: wenn
ein `{` folgt, dann wird es eingesetzt und für die nächsten
eilen gilt:

gleich tief: Semikolon davorsetzen
weniger tief: Klammer zu davorsetzen

Beispielcode (Übungen

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

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

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

 | Anzahl der Branch-Konstruktoren im Baum
anches :: Tree k -> Integer
anches t = case t of
  Leaf {} -> 0
  Branch {} -> branches (left t) + 1 + bran

 | Inorder-Liste der Schlüssel
order :: Tree k -> [ k ]
```

```
  Leaf {} -> []
  Branch {} -> inorder (left t) ++ [key t] ++

 | Hausaufgabe 1:
 Einfügen eines Schlüssels in einen (unbalanc
sert :: Ord k => Tree k -> k -> Tree k
sert t k =  ... key t < k ...

 | Hausaufgabe 2:
 Herstellen eines (unbalancierten) Suchbaums
omList :: Ord k => [ k ] -> Tree k
omList l = case l of
 x : xs -> ...
 []     -> ...
```

```
 | Hausaufgabe 3:
 vergleiche Leistung dieser beiden Funktion
:t1, sort2 :: Ord k => [k] -> [k]
:t1 = inorder . fromList
:t2 = Data.Set.toAscList . Data.Set.fromLis
```

Interpretation von Programmen

Abstrakte Syntax (Ausdrücke)

```
:a Expression = Constant Integer
   | Plus  Expression Expression
   | Times Expression Expression

aluate :: Expression -> Integer
aluate e = case e of ...

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

Abstrakte Syntax (Anweisungen)

```
:a Identifier = Identifier String

:a Statement = Assign Identifier Expression
   | Iterate Expression Statement

pe Program = [ Statement ]
```

Semantik (Anweisungen)

```
port qualified Data.Map as M

pe Environment = M.Map Identifier Integer

ecute :: Environment -> Statement -> Environm

1 :: Program -> Environment
```

Semantik (mit Maybe)

```
:a Expression = ... | Reference Identifier
aluate
  :: Environment -> Expression -> Integer ?
  :: Environment -> Expression -> Maybe Int
aluate 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 )
```

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

Typklassen, Monaden

Typklassen, Beispiel: Show

```
ass Show a where { show :: a -> String }

:a Foo = Foo { foo :: Integer }

stance Show Foo where
  show f = "F" ++ show ( foo f )

1ow :: Show a -- Typconstraint
    => a -> String
1ow x = reverse ( show x )
```

Vergleich Typklassen/Interfaces

```
:erface I { }    |    class I where { }
```

ymorphie auf der Argumentseite:

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

der Resultatseite:

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

Generische Instanzen

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

; kann der Compiler selbst:

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

Beziehungen zwischen Klassen

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

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

Konstruktorklassen (Functor)

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

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

stance Functor Maybe where ...

stance Functor Tree where ...
```

erscheide von:

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

Gesetze für Funktoren (I)

egorie der Datentypen:

)bjekte (Punkte): Typen

lorphismen (Pfeile): (einstellige) Funktionen

ıktor F von Kategorie A nach Kategorie B

$_{\mathsf{ob}}$: (Objekt in A) $\mapsto$ (Objekt in B)

$_{\mathsf{mo}}$: (Pfeil $x \to y$ in A) $\mapsto$ (Pfeil $F_{\mathsf{ob}}(x) \to F_{\mathsf{mo}}(y)$ in B)

daß

$_{\mathsf{mo}}(\mathrm{id}) = \mathrm{id}$

$_{\mathsf{mo}}((x \to y) \circ (y \to z)) = F_{\mathsf{mo}}(x \to y) \circ F_{\mathsf{mo}}(y \to z)$

Gesetze für Funktoren (II)

Instanzen der Typklasse Functor muß gelten:

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

gaben: finde falsche Funktor-Instanzen

ir `List, Maybe, Tree`, die

keines
nur das erste
nur das zweite

iesetz erfüllen (das sind 9 Aufgaben)

Die Konstruktorklasse Monad

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

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

Anwendung der Maybe-Monade

```
se ( evaluate e l ) of
  Nothing -> Nothing
  Just a  -> case ( evaluate e r ) of
      Nothing -> Nothing
      Just b  -> Just ( a + b )
```

tels der Monad-Instanz von Maybe:

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

Do-Notation für Monaden

ginal:

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

Notation:

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

List als Monade

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

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

Aufgaben zur List-Monade

ythagoreische Tripel aufzählen

amanujans Taxi-Aufgabe ($a^3 + b^3 = c^3 + d^3$)

lle Permutationen einer Liste

lle Partitionen einer Zahl
alle ungeraden, alle aufsteigenden)

weise:

llgemein: Programme mit `do, <-, guard, return`

ei Permutationen benutze:

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

Gesetze für Monaden (II)

iesetze für die Instanzen von List und Maybe überprüfen

iibt es eine korrekte Instanz für binäre Bäume?

mit Schlüsseln in Branch und nicht in Leaf
mit Schlüsseln in Leaf und nicht in Branch

iib eine inkorrekte Instanz für List, Maybe, Tree, die …

typkorrekt ist
genau eines der Gesetze verletzt

Die IO-Monade

```
ta IO -- abstract

adFile :: FilePath -> IO String
tStrLn :: String -> IO ()

stance Functor IO ; instance Monad IO
```

: „Funktionen“, deren Resultat von der Außenwelt
stemzustand) abhängt, haben Resultattyp `IO ...`
Typ einer Funktion erkennt man ihre möglichen
nädlichen) Wirkungen bzw. deren garantierte
wesenheit.
gen der Monad-Instanz: benutze do-Notation

Monaden mit Null

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

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

```
ass Monad m => MonadPlus m where
  mzero :: m a ; ...
stance MonadPlus [] where mzero = []
```

Gesetze für Monaden (I)

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

bjekte: Typen

feile: von a nach $m\ b$

nposition der Pfeile ist so definiert:

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

e Monade muß erfüllen:

`eturn` ist neutral für `comp`

`omp` ist assoziativ

Funktoren und Monaden

le Monade ist ein Funktor

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

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

vollständige und beweise die Gesetze

```
fmap f m ) >>= g     ==   m >>= ( ... )
>>= ( fmap g . f )  ==    ...
```

Die Zustands-Monade

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

```
port Control.Monad.State

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

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

gabe: wie könnte die Implementierung aussehen?

```
ta State s a = ??
stance Functor ( State s ) where
stance Monad ( State s ) where
```

Kombinator-Parser

Parser als Monade

ıtdeterministischer Parser über Eingabestrom von
chen (Token) c und mit Resultattyp c:

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

ıstanz für Functor, Monad

eispiel-Parser, Aufrufe davon

Parser als Monade (Implementierung)

```
stance 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
```

ıchte: das *do* gehört zur List-Monade

Elementare Parser (I)

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

Elementare Parser (II)

```
tisfy :: ( c -> Bool ) -> Parser c c
tisfy p = do
  x <- next
  if p x then return x else reject

pect :: Eq c => c -> Parser c c
pect c = satisfy ( == c )

ffer :: Parser Char Integer
ffer = do
  c <- satisfy Data.Char.isDigit
  return $ read [ c ]
```

Kombinatoren für Parser (I)

olge (and then) (ist *bind* aus der Monade)

uswahl (or)

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

eliebig viele

```
any, many1 :: Parser c a -> Parser c [a]
any p = many1 p <|> return []
any1 p = do x <- p; xs <- many p; return $ x : xs
```

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

Kombinator-Parser und Grammatiken

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

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

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

Eindeutigkeit/Determinismus

spiel $S \to S + S \mid S * S \mid Z$

swege:

irammatik ändern („eindeutig machen“)

irammatik bleibt, Operrator-Vorrang (Präzendenz,
ssoziativität) festlegen

Operator-Präzendenz-Parser

```
ta Operator a = Operator
  { precedence :: Int
  , semantics :: a -> a -> a }
erator_precedence_parser
  :: Map String ( Operator a )
  -> Parser Char a -> Parser Char a
thmetic :: Parser Char Integer
thmetic = operator_precedence_parser ( M.fromList
 [ ( "+", Operator { precedence = 1
                   , semantics = (+) } )
 , ( "*", Operator { precedence = 2
                   , semantics = (*) }
 ] )
```

Op-P-P (Implementierung)

ıutzt zwei Keller:

/erte

ıperatoren

Lesen eines

toms: push auf Wertkeller

ıperators:

ggf. Ausführung von top-of-opkeller
push auf Operatorkeller

Robuste Parser-Bibliotheken

ɜignfragen:

ichtdeterminismus einschränken

ehlermeldungen (Quelltextposition)

spiel: Parsec (Autor: Daan Leijen)
:p://www.haskell.org/haskellwiki/Parsec

ıngen:

arsec-Parser aufrufen

arsec-Parser selbst schreiben (elementare, ombinatoren)

uildExpressionParser

Linq

Linq: Beispiel

q = language integrated query

```
ıdom random = ... ; int block = ... ;
ıumerable<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;
is.pool = store.Take(this.size).ToList();
```

Linq und Monaden: Notation

ɜkell:

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

C# (LINQ):

```
IEnumerable<int> test =
    from x in Enumerabl
    where 0 != x % 2
    from y in Enumerabl
    select x * y;
```

q: http://msdn.microsoft.com/en-us/ ɔrary/bb397676.aspx

spiele: http://msdn.microsoft.com/en-us/ ɜharp/aa336746.aspx (Vorsicht: inzwischen einige thoden umbenannt)

Linq und Monaden: Semantik

ɜkell:

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

q:

```
ing System.Linq;
ɔlic static IEnumerable<TResult>
SelectMany<TSource, TResult>(
  this IEnumerable<TSource> source,
  Func<TSource, IEnumerable<TResult>> selecto

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

Linq und Haskell: gut zu wissen

ıskell	Linq
.]	`IEnumerable<a>`
ɑp und bind (>>=)	Select und SelectMany
ke und drop	Take und Skip
+) und reverse	Concat und Reverse
d	Aggregate
`: .. x+n-1]`	`Enumerable.Range(x,n)`
`ɾM_ xs print`	xs.ToList().ForEach(x => Console.WriteLine(x))

Projekte Compilerbau (autotool)

ype-Checking erweitern
ttps://autotool.imn.htwk-leipzig.de/ gi-bin/Trial.cgi?topic=TypeCheck-Quiz

- Methoden nicht nur static
- Vererbung
- generische Polymorphie

eue 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/

Projekte Compilerbau (Refactoring)

Haskell und/oder Java:

ɥuelltext lesen

luster im Syntaxbaum erkennen (ähnlicher Code)

efaktoring vorschlagen (Unterprogramm mit arametern)

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

Semantik

Interpreter

Bestandteilen:

yntaktische Analyse (erzeugt AST)

ıterpretation des AST (execute, eval)

prüfung:

ynamisch (während Interpretation) oder

tatisch (vor Interpretation)

Umgebungen

gebung: Zuordnung von Variablenname zu

/ert (bei Interpretation)

dresse (bei Kompilation)

eiter Informationen (z. B. Typ)

Sprachen mit Blockstruktur: benötigen geschachtelte sted) Umgebungen.

Unterprogramme

inition UP:

chnittstelle (formale Parameterliste, Rückgabetyp)

nplementierung (Block-Statement)

alisierung in Interpreter/Compiler:

yntax

abstrakt (AST)
konkret (Parser)

emantik

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);
}
```

Auswertung von `f(3)`:

ynamische Bindung: verwendet inneres $a(= 7)$,

tatische Bindung: verwendet äußeres $a(= 5)$.

Programme mit statischer Bindung können statisch -geprüft werden.

Implementierung der statischen Bindung

```
a Object = .. | OFunction { args :: [ Name
  , body :: Expression , env  :: Env Object
l 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#

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

ıutzt wird

tatische Bindung

Â¼r die Adresse von a

Lokale Funktionen in Java

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

nicht-lokale Variable a muÃ als final deklariert werden, lurch ist die Zuweisung ($a = 7$) verboten.

Currying

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

ltiplikation ist eigentlich zweistellig, diese Version ist ch „Currying“ einstellig.

nannt nach Haskell B. Curry, 1900–1982, :p://www-history.mcs.st-andrews.ac.uk/ ographies/Curry.html)

Rekursive Bindungen

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

 geht doch:
ıc<int,int> f = (x => x > 0 ? x * f(x-1) :
ısole.WriteLine (f(3));
```

Lambda-Kalkül

Motivation

ıundene (lokale) Variablen in der …

nalysis: $\int x^2 \, \mathrm{d}\, x, \sum_{k=0}^{n} k^2$

ogik: $\forall x \in A : \forall y \in B : P(x, y)$

rogrammierung:

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

Der Lambda-Kalkül

ınzo Church, 1936 … Henk Barendregt, 1984 …)

ler Kalkül für Funktionen mit benannten Variablen

 wesentliche Operation ist das Anwenden einer ıktion:

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

spiel: $(\lambda x.x * x)(3 + 2) \to (3 + 2) * (3 + 2)$

reinen Lambda-Kalkül gibt es *nur* Funktionen—keine ılen

Lambda-Terme

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

/ariable) wenn $x \in V$, dann $x \in \Lambda$

\pplikation) wenn $F \in \Lambda, A \in \Lambda$, dann $(FA) \in \Lambda$

\bstraktion) wenn $x \in V, B \in \Lambda$, dann $(\lambda x.B) \in \Lambda$

; sind also Lambda-Terme:

$\lambda x.x), ((xz)(yz)), (\lambda x.(\lambda y.(\lambda z.((xz)(yz)))))$

verkürzte Notation

pplikation als links-assoziativ auffassen, Klammern 'eglassen:

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

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

eschachtelte Abstraktionen unter ein Lambda schreiben:

$$\lambda x_1.(\lambda x_2.\ldots(\lambda x_n.B)\ldots) \sim \lambda x_1x_2\ldots x_n.B$$

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

Mehrstellige Funktionen

 vorigen Abkürzungen sind sinnvoll, denn

$_1 \ldots x_n.B)A_1 \ldots A_n$

hält sich wie eine Anwendung einer mehrstelligen ıktion.

 die zu beschreiben, genügt also ein Kalkül für stellige Funktionen.

ispiel)

Ableitungen (Ansatz)

sicht: Relation $\to$ auf Λ (Ein-Schritt-Ersetzung):

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

$' \to F' \Rightarrow (FA) \to (F'A)$

$\to A' \Rightarrow (FA) \to (FA')$

$\to B' \Rightarrow \lambda x.B \to \lambda x.B'$

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

das sinnvoll:

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

; freie y wird fälschlich gebunden

Das falsche Binden von Variablen

riges Beispiel in C++):
se Programme sind *nicht* äquivalent:

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

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

ispiel-Ableitungen)

Gebundene Umbenennungen

dürfen $(\lambda x.B)A \to B[x := A]$ nur ausführen, wenn x nt in A frei vorkommt.
s doch, müssen wir $\lambda x.B$ in $\lambda y.B[x := y]$ umbenennen, bei y weder in A frei noch in B überhaupt vorkommt.

ispiel) (Def. $\mathrm{FV}(t)$)
e solche gebundene Umbenennung in einem Teilterm ßt α-Konversion.
onvertierbare Terme sind äquivalent (verhalten sich ch bzgl. Ableitungen)
ispiel)
o.g. Bedingung ergibt sich eine vernünftige Relation $\to$ Reduktion).

Eigenschaften der Reduktion

auf Λ ist

onfluent,

ber nicht terminierend.

$V = \lambda x.xx, \Omega = WW$.

s gibt Terme mit Normalform und unendlichen bleitungen, $KI\Omega$ mit $K = \lambda xy.x, I = \lambda x.x$

ige Eigenschaften klingen erstaunlich: z. B. jeder Term besitzt einen Fixpunkt A, d. h. $FA \to^* A$.
n kann man sogar ausrechnen: es gibt R mit $RF) \to^* RF$.

Rechnen mit simulierten Zahlen

urch-Kodierung der natürlichen Zahlen (Alonzo Church, 03–1995, `http://www-history.mcs.st-andrews.uk/Biographies/Church.html`)

dee: $[n] = \lambda fx.f^n(x)$

ull: $[0] = \lambda fx.x (= K)$,

achfolger: $[n + 1] = \lambda fx.[n]f(fx)$, also $ucc = \lambda nfx.nf(fx)$

afür sind Nachfolger, Vorgänger, Null-Test definierbar.

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

ung: Addition, Multiplikation, Potenzieren
sächlich ist das Modell älter als die Turing-Maschine)

Erweiterungen, Anwendungen

sgehend vom einfachen Lambda-Kalkül baut man:

ypsysteme (jeder Ausdruck, jede Variable besitzt Typ)
ingebaute Datentypen (außer Funktionen) und Operationen, z. B. Zahlen
ffiziente Implementierung von Reduktionen (ohne die mständlichen Umbenennungen)

bildet die Grundlage für

xakte Analyse von rogrammier/mathematischen/logischen Sprachen
nplementierung von Sprachen und efactoring-Werkzeugen

Lambda-Kalkül und Computeralgebra

alkül beschreibt Rechnen mit Ausdrücken mit ebundenen Variablen, diese kommen in CAS vor.

ie Erkenntisse aus dem Kalkül werden in verschiedenen AS mit verschiedener Konsquenz angewendet (leider).

robleme beginnen damit, daß Variablenbindungen chon gar nicht korrekt notiert werden

. das ist nur ein getreues Abbild entsprechender robleme in der Mathematik (solche Fehler heißen dort ber Konventionen)

Typen

Grundlagen

ɔisher) konkrete Interpretation (Programm $\mapsto$ Daten)

etzt) abstrakte Interpretation (Programm $\mapsto$ Typ)

später) abstrakte Interpretation (Programm $\mapsto$ ielprogramm)

n: definiere

```
:a Type =
[Integer | TBool | TFunction [ Type ] Type
```

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

Deklaration und Inferenz

eklaration:

rogrammierer gibt für jeden Bezeichner Typ an, ompiler prüft auf Konsistenz.

rinzip: $(f :: a \to b \wedge x :: a) \Rightarrow f(x) :: b$

iferenz:

ompiler berechnet ($=$ inferiert) Typ der Bezeichner

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

Polymorphie

s ist der Typ von t hier:

```
: t  =  f => x => f (f (x));
int t(x => x+1)(0);
int t(t)(x => x+1)(0);
```

ung: $\forall a : (a \to a) \to (a \to a)$

nötigen

ypvariablen (a)

indungen dafür ($\forall$)

Inferenz Polymorpher Typen

der neue Bezeichner bekommt eine neue Typvariable

de Benutzung eines Bezeichners erzeugt ein ypconstraint

Teil-)Programm bestimmt Constraintsystem

ösung ist Zuordnung: Bezeichner $\to$ Typ

ösung existiert? ist eindeutig?

Constraint-Lösen ist hier das *Unifizieren*

Unifikation—Begriffe

ignatur $\Sigma = \Sigma_0 \cup \ldots \Sigma_k$,

$\mathrm{erm}(\Sigma, V)$ ist kleinste Menge T mit $V \subseteq T$ und $0 \le i \le k, f \in \Sigma_i, t_1 \in T, \ldots, t_i \in T : f(t_1, \ldots, t_i) \in T$.

ubstitution: partielle Abbildung $\sigma : V \to \mathrm{Term}(\Sigma, V)$, so aß kein $v \in \operatorname{dom} \sigma$ in $\operatorname{img} \sigma$ vorkommt,

ubstitution σ auf Term t anwenden: $t\sigma$

rodukt von Substitutionen: so definiert, daß $\sigma_1 \circ \sigma_2) = (t\sigma_1)\sigma_2$

Unifikation—Definition

fikationsproblem

ingabe: Terme $t_1, t_2 \in \mathrm{Term}(\Sigma, V)$

usgabe: eine allgemeinster Unifikator (mgu): ubstitution σ mit $t_1\sigma = t_2\sigma$.

emeinst $=$ minimal bzgl. der Prä-Ordnung $\le \sigma_2 \iff \exists \tau : \sigma_1 \circ \tau = \sigma_2$

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

Unifikation—Algorithmus

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

ist Variable: …

ist Variable: symmetrisch

$= f(s_1, s_2)$ und $t = g(t_1, t_2)$: …

nerkungen:

ɔrrekt, übersichtlich, aber nicht effizient,

s gibt Unif.-Probl. mit exponentiell großer Lösung,

ine komprimierte Darstellung davon kann man aber in

Beispiel zur Polymorphie (I)

/as ist der allgemeinste Typ von

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

eispiel für einen typkorrekten Aufruf von `f1`?

t dieser Aufruf typkorrekt?

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

Beispiel zur Polymorphie (II)

```
( \ x -> x ) 0 für
g x = length $ g [g x]
```

ıt (nur) so:

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

; ist nicht das gleiche wie:

```
:: forall a . (a->a) -> Int -> Int
g x = length $ g [g x]
```

System F

infach getypter Lambda-Kalkül:

$\Rightarrow x \mid AA \mid \lambda x : T.A$

$\Rightarrow \mathbb{N} \mid T \to T$

ariable x, Type T, Term A

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

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

ypvariable X

indley-Milner (benutzt in Standard-Haskell): Typ-Forall ur außen erlaubt

Rang-(2/N)-Polymorphie in GHC

```
:p://haskell.org/ghc/docs/latest/html/
ers_guide/other-type-extensions.html
 language Rank2Types #-}
:: (forall a. a->a) -> Int -> Int
:: Int -> (forall a. a -> a)

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

ng: $1+$ max. Tiefe der $(\Box \to *)$-Kontexte für Typ-Forall

Eigenschaften polym. Typsyst.

ermination:

etypte Programme terminieren

ekonstruktion:

s gibt Algorithmus, der Typ eines ungetypten rogramms ausrechnet oder ablehnt, falls es keinen Typ esitzt

Rang 2: ja, beliebiger Rang: nein)

Unifikation höherer Ordnung

z: Typrekonstruktion für System F (beliebiger Rang) ist ntscheidbar.

veis: Reduktion von Unifikation höherer Ordnung.

HO-Unifikation:

ingabe: einfach getypte Lambda-Terme s, t,

usgabe: getypte Substitution σ mit $s\sigma \leftrightarrow^* t\sigma$

z: HO-Unifikation ist unentscheidbar.

veis: Reduktion von Hilberts 10. Problem (ganzzahlige ungen polynomieller Gleichungssysteme), Codierung Church-Zahlen.

es Dowek: `http://www.lix.polytechnique.fr/` `owek/Publi/unification.ps`

Typen und Terme

Term abhängig von Term, Funktion auf Daten)

yp abhängig von Typ (Typkonstruktoren: Functor, Monad)

erm abhängig von Typ (System F)

yp abhängig von Term (dependent Types: Coq, PVS, ..)

r Lambda-Würfel, Barendregt 1991)

`:p://www.cs.ru.nl/~henk/`

Kompilation für OO-Sprachen

Plan

onkrete/abstrakte Syntax, Typprüfungen

`0.f(x1,..,xn) => f(x0,x1,..,xn)`

aten-Layout (Objekte)

aufzeit-Polymorphie (Methoden)

ypconstraints

Daten-Layout

Laufzeit-Polymorphie

```
ass A { void m () { } }
ass B extends A { void m () { } }

. A x = new B (); x.m(); ...
```

lethodentabelle

infach-Vererbung

lehrfach-Vererbung

Typconstraints (Haskell)

```
ass C t where m :: t -> Int
:a A ; instance C A where m = ...
:a B ; instance C B where m = ...

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

›lementierung durch *Dictionaries*

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


Vergleich Constraints/Interfaces

ıterface ist einstelliges Constraint

llgemeines Constraint ist aber mehrstellig

›. Autotool:

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

http://141.57.11.163/cgi-bin/cvsweb/
›l/src/Challenger/Partial.hs?rev=1.29

(Co-)Varianz von Typparametern

us `B extends A` folgt nicht
`ist<B> extends List<A>`

`n/out` an Typparametern in C# 4