Compilerbau
Vorlesung
Wintersemester 2008, 09, 10

Johannes Waldmann, HTWK Leipzig

23. Januar 2011

Einleitung

Beispiel

Eingabe (≈ Java):

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

Ausgabe (Drei-Adress-Code):

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


Inhalt

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

Sprachverarbeitung

- mit Compiler:
  - Quellprogramm → 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)
- autotool: Beschreibungssprache für formale Sprachen (mit Sprachoperationen, Tester und Generator)

Beispiel: Interpreter (I)

arithmetische Ausdrücke:

```
data Exp = Const Int | Plus Exp Exp | Times Exp
    deriving ( Show )
ex1 :: Exp
ex1 = Times ( Plus ( Const 1 ) ( Const 2 ) ) (
value :: Exp -> Int
value x = case x of
    Const i -> i
    Plus x y -> value x + value y
    Times x y -> value x * value y
```

Beispiel: Interpreter (II)

lokale Variablen und Umgebungen:

```
data Exp = ...
         | Let String Exp Exp | Ref String
ex2 :: Exp
ex2 = Let "x" ( Const 3 )
     ( Times ( Ref "x" ) (Ref "x" ) )
type Env = ( String -> Int )
value :: Env -> Exp -> Int
value env x = case x of
    Ref n -> env n
    Let n x b -> value ( \ m ->
      if n == m then value env x else env m )  b
    Const i -> i
    Plus x y -> value env x + value env y
    Times x y -> value env x * value env y
```

Interpreter (Übung)

(bis Woche 41:)

Verzweigungen mit C-ähnlicher Semantik:

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

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

(evtl. später:)

neuer AST-Knoten für Mehrfachbindung

```
data Exp = ...
         | Lets [ (String, Exp) ] Exp
```

- Semantik definieren/direkt implementieren
- Semantik durch Transformation in geschachtelte `Let`

Umgebungen

Umgebung ist (partielle) Funktion von Name nach Wert

Realisierungen: `type Env = String -> Int`

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

Operationen:

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

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

  Notation: $e[x/v]$

Spezifikation:

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

Mehrstellige Funktionen

...simulieren durch einstellige:

- Abstraktion: $\lambda xyz.B := \lambda x.\lambda y.\lambda z.B$
- Applikation: $fPQR := ((fP)Q)R$

weiterer *syntactic sugar*:

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

Testfall (1)

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

Semantik

erweitere den Bereich der Werte:

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

erweitere Interpreter:

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

mit Hilfsfunktion

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

Interpreter mit Funktionen

abstrakte Syntax:

```
data Exp = ...
  | Abs { formal :: Name , body :: Exp }
  | App { rator  :: Exp  , rand :: Exp }
```

konkrete Syntax (Beispiel):

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

Der Lambda-Kalkül

(Alonzo Church, 1936) Syntax:

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

Begriffe: freie Variablen (FV), gebundene Variablen (BV), gebundene Umbenennung
Semantik: Ersetzung von Teiltermen:

- $(\lambda x.A)B \to A[x/B]$
- wobei FV(B) disjunkt zu BV(A)

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),

Continuations

Programmablauf-Abstraktion durch Continuations:
Definition:

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

Benutzung:

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

Aufgabe: if/then/else mit `with_bool`

Semantische Bereiche

bisher: Wert eines Ausdrucks ist Zahl.
jetzt erweitern (Motivation: if-then-else mit richtigem Typ):

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

Dann brauchen wir auch

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

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

Let und Lambda

- `let { x = A } in Q`

  kann übersetzt werden in

  `(\ x -> Q) A`

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

  wird übersetzt in . . .

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

Rekursion?

- Das geht nicht, und soll auch nicht gehen:

  ```
  let { x = 1 + x } in x
  ```

- aber das hätten wir doch gern:

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

  (nächste Woche)

- aber auch mit nicht rekursiven Funktionen kann man interessante Programme schreiben:

Testfall (2)

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

- auf dem Papier den Wert bestimmen
- mit Haskell ausrechnen
- mit selbstgebautem Interpreter ausrechnen

Fixpunkte

Motivation

Das geht bisher gar nicht:

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

(Bezeichner f ist nicht sichtbar)

Lösung:

```
( rec f ( \ x -> if x > 0
                 then x * f (x -1) else 1 )) 5
```

mit neuem AST-Knotentyp `rec`

Rekursion

abstrakt:

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

Semantik von `Rec n b` in Umgebung E
ist der Fixpunkt der Funktion (vom Typ Val → Val)

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

Existenz von Fixpunkten

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

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

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

Funktionen als CPO

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

Funktionen als CPO, Beispiel

Wert von

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

ist Fixpunkt der Funktion $F =$

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

Iterative Berechnung des Fixpunktes:

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

Speicher-Aktionen als Monade

generische Aktionen/Verknüpfungen:

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

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

Speicher

Modellierung:

```
import qualified Data.Map as M
newtype Addr = Addr Int
type Store = M.Map Addr Val
newtype Action a =
    Action ( Store -> ( Store, a ))
```

spezifische Aktionen:

```
new :: Val -> Action Addr
get :: Addr -> Action Val
put :: Addr -> Val -> Action ()
```

Zustand/Speicher

Motivation

bisherige Programme sind nebenwirkungsfrei, das ist nicht immer erwünscht:

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

Dazu muß semantischer Bereich geändert werden.

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

Semantik von (Teil-)Programmen ist Zustandsänderung.

Lambda-Kalkül als universelles Modell

- Wahrheitswerte:

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

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

- natürliche Zahlen:

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

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

- Rekursion mit Fixpunkt-Kombinator

Satz: Menge der berechenbaren Funktionen (mit Turingmaschine, while-Programm) = Menge der Lambda-berechenbaren Funktionen (mit dieser Kodierung)

Fixpunkt-Kombinatoren

- Definition: $Y := \lambda f.((\lambda x.f(xx))(\lambda x.f(xx)))$
- Satz: Yf ist Fixpunkt von f
- d.h. Y ist *Fixpunkt-Kombinator*
- Beweis: vergleiche (Yf) und $f(Yf)$
- Folgerung: rec wird eigentlich nicht benötigt

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

letrec nach rec

mithilfe der Lambda-Ausdrücke für select und tuple

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

letrec

Beispiel (aus: D. Hofstadter, GEB)

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

Bastelaufgabe: für welche x gilt $f(x) \neq g(x)$?
AST-Knoten:

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

weitere Beispiele:

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

Daten als Funktionen

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

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

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

- Auflösung simultaner Rekursion
- Modellierung von Zahlen

Auswertung von Ausdrücken

Ausdrücke (mit Nebenwirkungen):

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

Resultattyp des Interpreters ändern:

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

semantischen Bereich erweitern:

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

Änderung der Hilfsfunktionen

bisher:

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

jetzt:

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

Hauptprogramm muß kaum geändert werden (!)

Rekursion

...wird benötigt für Wiederholungen (Schleifen).
mehrere Möglichkeiten zur Realisierung

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

Rekursion (semantisch)

bisher:

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

jetzt:

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

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

Rekursion (operational)

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

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

Speicher—Übung

Fakultät imperativ:

```
let { fak = \ n ->
        { a := new 1 ;
          while ( n > 0 )
            { a := a * n ; n := n - 1; }
          return a;
        }
    } in  fak 7
```

Dabei Schleife durch Rekursion ersetzen

```
fak = let { a = new 1 }
      in  Rec f ( \ n -> ... )
```

Monaden

Die Konstruktorklasse Monad

Definition:

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

Benutzung der Methoden:

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

Do-Notation für Monaden

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

do-Notation (explizit geklammert):

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

do-Notation (implizit geklammert):

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

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

Die Zustands-Monade

Implementierung:

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

Benutzung:

```
import Control.Monad.State

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

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

List als Monade

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

Kombinator-Parser

Datentyp für Parser

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

- über Eingabestrom von Zeichen (Token) c,
- mit Resultattyp a,
- nichtdeterministisch (List).

Beispiel-Parser, Aufrufen mit:

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

Elementare Parser (I)

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

Monadisches Verketten von Parsern

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

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

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

Elementare Parser (II)

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

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

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

Kombinatoren für Parser (I)

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

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

- Wiederholung (beliebig viele)

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

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

Kombinator-Parser und Grammatiken

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

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

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

Robuste Parser-Bibliotheken

Designfragen:

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

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

Asymmetrische Komposition

gemeinsam:

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

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

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

Nichtdeterminismus einschränken

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

Festlegung (in Parsec): wenn `p` wenigstens ein Zeichen verbraucht, dann wird `q` nicht benutzt (d. h. `p` muß erfolgreich sein)
Backtracking dann nur durch `try p <|> q`

Fehlermeldungen

- Fehler = Position im Eingabestrom, bei der es „nicht weitergeht“
- und auch durch Backtracking keine Fortsetzung gefunden wird
- Fehlermeldung enthält:
  - Position
  - Inhalt (Zeichen) der Position
  - Menge der Zeichen mit Fortsetzung

Pretty-Printing (I)

John Hughes's and Simon Peyton Jones's Pretty Printer Combinators
Based on *The Design of a Pretty-printing Library* in *Advanced Functional Programming*, Johan Jeuring and Erik Meijer (eds), LNCS 925
`http://hackage.haskell.org/packages/archive/pretty/1.0.1.0/doc/html/Text-PrettyPrint-HughesPJ.html`

Pretty-Printing (II)

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

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

- Kombinatoren:

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

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

Parameterübergabe (call by ...)

Einleitung

durch `lambda` und `let` werden Namen gebunden.
Designfrage: was bezeichnet der gebundene Name?
mögliche Antworten:

- einen Wert (call by value)
- eine Aktion (call by name)
- erst eine Aktion, dann deren Resultat (Wert) (lazy evaluation)

Wertübergabe (CBV)

call-by-value
ist bisher implementiert

```
type Env = Name -> Val

evaluate env x = case x of ...
    Ref n -> return $ env n
    App f a ->
        with_fun ( evaluate env f ) $ \ fun ->
        with     ( evaluate env a ) $ \ arg ->
        fun arg
```

Call by name

- Name bezeichnet Aktion
- Benutzung des Namens = Ausführung der Aktion

```
type Env = Name -> Action Val

evaluate env x = case x of ...
    Ref n -> ...
    App f a ->
        with_fun ( evaluate env f ) $ \ fun ->
        fun ( evaluate env a )

data Val = ... | ValFun ...
```

Simulation von CBN in CBV

- CBN:
  - Name n bezeichnet Aktion A
- CBV:
  - Name n gebunden an $\lambda x.A$
  - Benutzung des Namens: $n()$

Bedarfsauswertung (lazy evaluation)

- Name bezeichnet Aktion
- nach erster Ausführung bezeichnet der Name deren Resultat

Realisierung durch:

- Name bezeichnet Speicherzelle
- dort steht die Aktion, …
- die sich selbst durch ihr Resultat ersetzt

Einschätzung

- CBN/CBV: wer CBN nicht hat, wird (muß) es nachbauen, um Abstraktionen über Programmablauf auszudrücken (if/then/else)
- selbst wenn die Aktionen (Teilprogramme) keine Wirkung haben, können sie doch fehlschlagen oder nicht terminieren
- CBN/CBL: sind für nebenwirkungsfreie Programme äquivalent, aber CBL ist effizienter
- CBL mit Nebenwirkungen: Reihenfolge der Nebenwirkungen ist nicht vorhersehbar $\Rightarrow$ gefährlich
  - ML: Nebenwirkungen, CBV (eager evaluation)
  - Haskell: keine N.W, CBL (lazy evaluation)

Ablaufsteuerung/Continuations

Definition

(alles nach: Turbak/Gifford Ch. 17.9)
CPS-Transformation (continuation passing style):

- original: Funktion gibt Wert zurück

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

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

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

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

Motivation

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

CPS als einheitlicher Mechanismus für

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

CPS für Linearisierung

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

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

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

später tatsächlich als Programmtransformation (Kompilation)

CPS für Resultat-Tupel

wie modelliert man Funktion mit mehreren Rückgabewerten?

- benutze Datentyp Tupel (Paar):

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

- benutze Continuation:

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

Beispiel

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

CPS-Transformation

CPS-Transformation: Spezifikation

(als Schritt im Compiler)

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

Semantik

semantischer Bereich:

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

Plan:

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

CPS für Ablaufsteuerung

Beispiel label/jump

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

CPS-Transformation: Zielsyntax

```
Exp/CPS ==> App Id Exp/Value^*
      | If Exp/Value Exp/CPS Exp/CPS
      | Let Id Exp/Letable Exp/CPS
      | Err String

Exp/Value ==> Literal + Identifier

Exp/Letable ==> Literal
      | Abs Id^* Exp/CPS
      | Exp/Value Op Exp/Value
```

Übersetze

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

CPS als Monade

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

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

feed ( return x ) c = c x

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

Semantik für CPS

Semantik von Ausdruck `x`
in Umgebung E mit Continuation k

- `x = (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 = (label L B)`

  Semantik von B in Umgebung $E[L/k]$ mit k
- `x = (jump L B)`

  let $k' =$ gebundener Wert von L in E,

  Semantik von B in E mit k'

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

Transformation f. Applikation

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

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

Ü: ähnlich für Primop (Unterschied?)

Transformation f. Abstraktion

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

Ü: Transformation für `let`

Namen

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

```
import Data.Set
var :: Exp -> Set String
```

Frische Namen aufzählen und in Zustandsmonade bereitstellen. Zustand ist (unendliche) Liste der frischen Namen.

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

fresh :: State [String] String
fresh = do (n:ns)<-get ; put ns; return n
```

Namen (II)

Der Typ des CPS-Transformators ist dann:

```
transform' :: Exp -> State [ String ] Exp
transform' x = case x of
    ...
```

Schnittstelle nach außen:

```
transform :: Exp -> Exp
transform =
   evalState ( transform' x ) ( unused x )
```

Vereinfachungen

um geforderte Syntax (ExpCPS) zu erreichen:

- implicit-let

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

Umbenennungen von Variablen entfernen:

- copy-prop

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

aber kein allgemeines Inlining

Teilweise Auswertung

- Interpreter (bisher): komplette Auswertung

  (Continuations sind Funktionen, werden angewendet)
- CPS-Transformator (heute): gar keine Auswertung,

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

  (compile time evaluation, partial evaluation)

Partial Evaluation

- bisher:

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

- jetzt:

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

Partial Evaluation (II)

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

ändere letzte Zeile in

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

Plan f. Compiler

Transformationen

- continuation passing (Programmablauf explizit)
- closure conversion (Umgebungen explizit)
- lifting
- Registervergabe

Ziel: maschinen(nahes) Programm mit

- globalen (Register-)Variablen (keine lokalen)
- Sprüngen (kein return)

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$_1, \ldots,$ Wert$_n$)

```
(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 $\{v_1, \ldots\} =$ 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?

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

Registervergabe

Motivation

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

Typen

Grundlagen

Typ = statische Semantik
(Information über mögliches Programm-Verhalten, erhalten ohne Programm-Ausführung)
formale Beschreibung:

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

Inferenz-Systeme

ein *Inferenz-System* I besteht aus

- Regeln (besteht aus Prämissen, Konklusion)
  Schreibweise $\frac{P_1,\dots,P_n}{K}$
- Axiomen (= Regeln ohne Prämissen)

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

- jeder Knoten ist mit einer Formel beschriftet
- jeder Knoten (mit Vorgängern) entspricht Regel von I
- Wurzel ist mit F beschriftet

Schreibweise: $I \vdash F$

Inferenz-Systeme (Beispiel 1)

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

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

- Regel-Schemata:

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

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

Eigenschaften des Typsystems

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

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

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

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

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

Applikation und Abstraktion

- Applikation:

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

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

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

Inferenz für Let

(alles ganz analog zu Auswertung von Ausdrücken)

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

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

Inferenzsystem für Typen (Semantik)

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

hierbei (vorläufige) Design-Entscheidungen:

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

Inferenzsystem für Typen (Syntax)

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

  (in Umgebung E hat Ausdruck X den Typ T)
- Menge der Typen:
  - primitiv: Int, Bool
  - zusammengesetzt:
    - Funktion $T_1 \to T_2$
    - Verweistyp $\mathrm{Ref}\, T$
    - Tupel $(T_1, \ldots, T_n)$, einschl. $n = 0$
- Umgebung bildet Namen auf Typen ab

Inferenz mit Umgebungen

Motivation: Inferenzsystem für (aussagenlogische) Formeln mit Variablenbindung

- Grundbereich: Aussagen der Form $\text{wert}(E, F, B)$

  (in Umgebung E hat Formel F den Wert B)
- Regeln für Operatoren $\dfrac{\text{wert}(E, X, 1), \text{wert}(E, Y, 1)}{\text{wert}(E, X \wedge Y, 1)}, \ldots$
- Regeln für Umgebungen

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

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

Inferenz von Werten

- Grundbereich: Aussagen der Form $\text{wert}(f, b)$ mit $f \in$Formeln, $b \in \mathbb{B} = \{0, 1\}$
- Axiome: $\text{wert}(\mathsf{T}, 1), \text{wert}(\mathsf{F}, 0)$
- Regeln:

$$\frac{\text{wert}(X, 1), \text{wert}(Y, 1)}{\text{wert}(X \wedge Y, 1)}, \frac{\text{wert}(X, 0)}{\text{wert}(X \wedge Y, 0)}, \frac{\text{wert}(Y, 0)}{\text{wert}(X \wedge Y, 0)}, \ldots$$

Beispiel: leite $\text{wert}((T \wedge F) \wedge (T \wedge T), 0)$ ab.

Durch ein Inferenzsystem kann man die Semantik von Ausdrücken (Programmen) spezifizieren.
(hier: dynamische Semantik)

Inferenz-Systeme (Beispiel 2)

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

$$\frac{}{01}$$

- Regel-Schemata (für jedes u, v):

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

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

Polymorphe Typen

Motivation

ungetypt:

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

einfach getypt nur so möglich:

```
let { t2 = \ (f :: (Int -> Int) -> (Int -> Int)
             (x :: Int -> Int) -> f (f x)
    ; t1 = \ (f :: Int -> Int) (x :: Int) -> f
    ; s = \ (x :: Int) -> x + 1
    } in (t2 t1 s) 0
```

wie besser?

Typ-Argumente (Beispiel)

Typ-Abstraktion, Typ-Applikation:

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

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

Typ-Argumente (Regeln)

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

- Typ-Abstraktion: erzeugt parametrischen Typ

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

- Typ-Applikation: instantiiert param. Typ

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

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

Inferenz allgemeingültige Formeln

Grundbereich: aussagenlogische Formeln (mit Variablen und Implikation)

Axiom-Schemata:

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

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

Beobachtungen/Fragen:

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

Typen und Daten

- bisher: Funktionen von Daten nach Daten
  `\ (x :: Int) -> x + 1`
- heute: Funktionen von Typ nach Daten
  `\ (t :: Type) -> \ (x :: t) -> x`
- Funktionen von Typ nach Typ (ML, Haskell, Java, C#)
  `\ (t :: Type) -> List t`
- Funktionen von Daten nach Typ (*dependent types*)
  `\ (t :: Typ) (n :: Int) -> Array t n`
  Sprachen: Cayenne, Coq, Agda
  Eigenschaften: Typkorrektheit i. A. nicht entscheidbar, d. h. Programmierer muß Beweis hinschreiben.

Typ-Rekonstruktion

Motivation

Bisher: Typ-Deklarationspflicht für Variablen in Lambda. scheint sachlich nicht nötig. In vielen Beispielen kann man die Typen einfach rekonstruieren:

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

Diesen Vorgang automatisieren!
(zunächst für einfaches (nicht polymorphes) Typsystem)

Realisierung mit Constraints

Inferenz für Aussagen der Form $E \vdash X : (T, C)$

- E: Umgebung (Name → Typ)
- X: Ausdruck (Exp)
- T: Typ
- C: Menge von Typ-Constraints

wobei

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

Substitutionen (Definition)

- Signatur $\Sigma = \Sigma_0 \cup \dots \Sigma_k$,
- $\text{Term}(\Sigma, V)$ ist kleinste Menge T mit $V \subseteq T$ und $\forall 0 \le i \le k, f \in \Sigma_i, t_1 \in T, \dots, t_i \in T : f(t_1, \dots, t_i) \in T$.
  (hier Anwendung für Terme, die Typen beschreiben)
- Substitution: partielle Abbildung $\sigma : V \to \text{Term}(\Sigma, V)$,
  Definitionsbereich: $\text{dom}\,\sigma$, Bildbereich: $\text{img}\,\sigma$.
- Substitution σ auf Term t anwenden: $t\sigma$
- σ heißt *pur*, wenn kein $v \in \text{dom}\,\sigma$ als Teilterm in $\text{img}\,\sigma$ vorkommt.

Rekonstruktion polymorpher Typen

… ist im Allgemeinen nicht möglich:

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

übliche Einschränkung (ML, Haskell): *let-Polymorphismus*: Typ-Abstraktionen nur für let-gebundene Bezeichner:

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

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

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

Inferenzregeln f. Rekonstrukion

- primitive Operationen (Beispiel)

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

- Applikation

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

- Abstraktion

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

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

Inferenzregeln f. Rekonstruktion (Plan)

Plan:

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

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

Inferenzregeln? Implementierung? — Testfall:

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

Unifikation—Komplexität

Bemerkungen:

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

Unifikation—Algorithmus

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

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

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

Unifikation—Definition

Unifikationsproblem

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

(allgemeinst: infimum bzgl. $\lesssim$)

Satz: jedes Unifikationsproblem ist

- entweder gar nicht
- oder bis auf Umbenennung eindeutig

lösbar.

Substitutionen: Ordnung

Substitution σ_1 ist *allgemeiner als* Substitution σ_2:

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

Beispiele:

- $\{X \mapsto Y\} \lesssim \{X \mapsto a, Y \mapsto a\}$,
- $\{X \mapsto Y\} \lesssim \{Y \mapsto X\}$,
- $\{Y \mapsto X\} \lesssim \{X \mapsto Y\}$.

Eigenschaften

- Relation $\lesssim$ ist Prä-Ordnung (…, …, aber nicht …)
- Die durch $\lesssim$ erzeugte Äquivalenzrelation ist die …

Substitutionen: Produkt

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

Beispiel 1:

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

Beispiel 2 (nachrechnen!):

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

Eigenschaften:

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

Implementierung:

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

Implementierung

let-Polymorphie, Hindley/Damas/Milner

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

Beispiel

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

Anhang

Prüfungsvorbereitung

- was ist eine Umgebung (`Env`), welche Operationen gehören dazu?
- was ist eine Speicher (`Store`), welche Operationen gehören dazu?
- Gemeinsamkeiten/Unterschiede zw. Env und Store?
- Für $(\lambda x.xx)(\lambda x.xx)$: zeichne den Syntaxbaum, bestimme die Menge der freien und die Menge der gebundenen Variablen. Markiere im Syntaxbaum alle Redexe. Gib die Menge der direkten Nachfolger an (einen Beta-Schritt ausführen).
- Definiere Beta-Reduktion und Alpha-Konversion im Lambda-Kalkül. Wozu wird Alpha-Konversion benötigt? (Dafür Beispiel angeben.)
- Wie kann man Records (Paare) durch Funktionen simulieren? (Definiere Lambda-Ausdrücke für `pair, first, second`)
- welche semantischen Bereiche wurden in den Interpretern benutzt? (definieren Sie `Val, Action Val, CPS Val`)
- welches sind die jeweils hinzukommenden Ausdrucksmöglichkeiten der Quellsprache (`Exp`)?
- wie lauten die Monad-Instanzen für `Action, CPS, Parser`, was bedeutet jeweils das bind (>>=)?
- warum benötigt man call-by-name für Abstraktionen über den Programmablauf (warum kann man `if` oder `while` nicht mit call-by-value implementieren)?
- wie kann man call-by-name simulieren in einer call-by-value-Sprache?
- wie kann man call-by-value simulieren in einer call-by-name-Sprache (Antwort: durch CPS-Transformation)
- Definiere Fakultät mittels Fixpunktoperator (Definiere das `f` in `fak = fix f`)
- Bezüglich welcher Halbordnung ist dieses `f` monoton? (Definiere die Ordnung, erläutere Monotonie an einem Beispiel.)
- Wie kann man Rekursion durch get/put simulieren? (Programmbeispiel ergänzen)
- Wie kann man Rekursion durch label/jump simulieren? (Programmbeispiel ergänzen)
- Für die Transformationen CPS, Closure Conv., Lifting, Registervergabe: welche Form haben jeweils Eingabe- und Ausgabeprogramm? Auf welchem Maschinenmodell kann das Zielprogramm ausgeführt werden? (Welche Operationen muß das Laufzeitsystem bereitstellen?)
- Was sind die Bestandteile eines Inferenzsystems (Antwort: Grundbereich, Axiome, Regeln), wie kann man ein Axiom als Spezialfall einer Regel auffassen?
- wie lauten die Inferenzregeln für das Nachschlagen eines Namens in einer Umgebung?
- Inferenzregeln für Applikation, Abstraktion, Let, If/Then/Else im einfach getypten Kalkül
- Geben Sie ein Programm an, das sich nicht einfach (sondern nur polymorph) typisieren läßt. Geben Sie den polymorphen Typ an.
- Inferenz-Regeln für Typ-Applikation, Typ-Abstraktion im polymorphen Kalkül
- für Typ-Rekonstruktion im einfach getypten Kalkül: Welches ist der Grundbereich des Inferenzsystems?
- geben Sie die Inferenzregel für Typrekonstruktion bei If/Then/Else an
- Geben Sie eine Inferenzregel für Typrekonstruktion an, durch die neue Variablen eingeführt werden.
- Wann ist σ ein Unifikator von zwei Termen s, t?
- Geben Sie zwei verschiedene Unifikatoren von $f(a, X)$ und $f(Y, Z)$ an. Einer davon soll streng allgemeiner als der andere sein. Begründen Sie auch diese Beziehung.
- Bestimmen Sie einen Unifikator von $f(X_n, f(X_{n-1}, \ldots, f(X_0, a) \ldots))$ und $f(f(X_{n-1}, X_{n-1}), f(f(X_{n-2}, X_{n-2}), \ldots, f(a, a) \ldots))$.