Compilerbau
Vorlesung
Wintersemester 2008, 09, 10, 11

Johannes Waldmann, HTWK 17. November 201120.

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: regelmäßiges und erfolgreiches Bearbeiten von Übungsaufgaben
- Prüfung: Klausur (120 min, keine Hilfsmittel)

Beispiel: Interpreter (I)

arithmetische Ausdrücke:

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

Beispiel: Interpreter (II)

lokale Variablen und Umgebungen:

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

Übung (Haskell)

- Wiederholung Haskell
  - Interpreter/Compiler: ghci `http://haskell.org/`
  - Funktionsaufruf nicht `f(a,b,c+d)`, sondern `f a b (c+d)`
  - Konstruktor beginnt mit Großbuchstabe und ist auch eine Funktion
- Wiederholung funktionale Programmierung/Entwurfsmuster
  - rekursiver algebraischer Datentyp (ein Typ, mehrere Konstruktoren)
    (OO: Kompositum, ein Interface, mehrere Klassen)

  - rekursive Funktion
- Wiederholung Pattern Matching:
  - beginnt mit `case ... of`, dann Zweige
  - jeder Zweig besteht aus Muster und Folge-Ausdruck
  - falls das Muster paßt, werden die Mustervariablen gebunden und der Folge-Ausdruck auswertet

Übung (Interpreter)

- Benutzung:
  - Beispiel für die Verdeckung von Namen bei geschachtelten Let
  - Beispiel dafür, daß der definierte Name während seiner Definition nicht sichtbar ist
- Erweiterung:

  Verzweigungen mit C-ähnlicher Semantik:

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

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

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:

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

Inferenz-Systeme

Motivation

- inferieren = ableiten
- Inferenzsystem I, Objekt O,
  Eigenschaft $I \vdash O$ (in I gibt es eine Ableitung für O)
- damit ist I eine *Spezifikation* einer Menge von Objekten
- man ignoriert die *Implementierung* (= das Finden von Ableitungen)
- Anwendungen im Compilerbau:
  Auswertung von Programmen, Typisierung von Programmen

Definition

ein *Inferenz-System* I besteht aus

- Regeln (besteht aus Prämissen, Konklusion)
  Schreibweise $\frac{P_1,\ldots,P_n}{K}$
- 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? $\quad (-1,5)$? $(2,4)$?

Inferenz-Systeme (Beispiel 2)

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

$$\frac{}{01}$$

- Regel-Schemata (für jedes u, v):

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

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

Inferenz-Systeme (Beispiel 3)

- Grundbereich: endliche Folgen von ganzen Zahlen
- Axiome: jede konstante Folge (Bsp. $[3,3,3,3]$)
- Schlußregeln:
  - swap$_k$: $\dfrac{[\ldots, x_k, x_{k+1}, \ldots]}{[\ldots, x_{k+1}+1, x_k-1, \ldots]}$
  - rotate: $\dfrac{[x_1, \ldots, x_n]}{[x_2, \ldots, x_n, x_1]}$

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

- jede Folge der Form $[z, 0, \ldots, 0]$ ist ableitbar
- Invarianten, $[5,3,3]$ ist nicht ableitbar

praktische Realisierung: `http://www.siteswap.org/` und HTWK-Hochschulsport

Inferenz von Werten

- Grundbereich: Aussagen der Form $\text{wert}(p, z)$ mit $p \in$ `Exp` , $z \in \mathbb{Z}$

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

- Axiome: $\text{wert}(\texttt{Const}\,z, z)$
- Regeln:

$$\frac{\text{wert}(X,a), \text{wert}(Y,b)}{\text{wert}(\texttt{Plus}\ X\ Y, a+b)}, \quad \frac{\text{wert}(X,a), \text{wert}(Y,b)}{\text{wert}(\texttt{Times}\ X\ Y, a \cdot b)}, \ldots$$

Umgebungen

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

Aussagenlogische Resolution

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

folgendes Inferenzsystem heißt *Resolution*:

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

Resolution (Eigenschaften)

die Formel (Klauselmenge) ist nicht erfüllbar $\iff$ *die leere Klausel ist durch Resolution ableitbar.*

Beweispläne:

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

Semantische Bereiche

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

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

Dann brauchen wir auch

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

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

Continuations

Programmablauf-Abstraktion durch Continuations:
Definition:

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

Benutzung:

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

Aufgabe: if/then/else mit `with_bool`

Unterprogramme

Beispiele

- in verschiedenen Prog.-Sprachen gibt es verschiedene Formen von Unterprogrammen:

  Prozedur, sog. Funktion, Methode, Operator, Delegate, anonymes Unterprogramm
- allgemeinstes Modell: Kalkül der anonymen Funktionen (Lambda-Kalkül),

Interpreter mit Funktionen

abstrakte Syntax:

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

konkrete Syntax (Beispiel):

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

Semantik

erweitere den Bereich der Werte:

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

erweitere Interpreter:

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

mit Hilfsfunktion

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

Testfall (1)

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

Closures

bisher:

```
eval env x = case x of ...
  Abs n b -> ValFun $ \ v ->
    eval (extend env n v) b
  App f a ->
    with_fun ( eval env f ) $ \ g ->
    with_val ( eval env a ) $ \ v -> g v
```

alternativ: die Umgebung von `Abs` in die Zukunft transportieren:

```
eval env x = case x of ...
  Abs n b -> ValClos env n b
  App f a -> ...
```

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),

Small-Step-Semantik des Lambda-Kalküls

Basis-Operation ist $A[x/B]$:
ersetze jedes freie Vorkommen von x in A durch B

- β-Konversion:

  $(\lambda x.A)B \to_\beta A[x/B]$

  wobei FV(B) disjunkt zu BV(A)
- α-Konversion: gebundene Umbenennung

  $(\lambda x.A) \to_\alpha (\lambda y.A[x/y])$

…und Abschluß unter Kontext (Inferenz-System?)
Bemerkungen:

- warum ist die Bedingung bei $\to_\beta$ nötig? (Beispiele)

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
```

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: $f(\sup \vec{x}) = \sup \overrightarrow{f(x)}$

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:

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

Rechnen im Lambda-Kalkül

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

Lambda-Kalkül als universelles Modell

- Wahrheitswerte:

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

  (damit läßt sich if-then-else leicht aufschreiben)
- natürliche Zahlen:

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

  (damit kann man leicht $x > 0$ testen)
- Rekursion?

Fixpunkt-Kombinatoren

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

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,
$\Theta = (\lambda xy.(y(xxy)))(\lambda xy.(y(xxy)))$

Lambda-Berechenbarkeit

Satz: (Church, Turing)

Menge der Turing-berechenbaren Funktionen

(Zahlen als Wörter auf Band)

= Menge der while-berechenbaren Funktionen

(Zahlen als Registerinhalte)

= Menge der Lambda-berechenbaren Funktionen

(Zahlen als Lambda-Ausdrücke)

Übung Fixpunkte

- Fixpunkt der Folge $F^k(\bot)$ für

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

- Fixpunkt der Folge $F^k(\bot)$ für

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

- Turing-Fixpunkt-Kombinator mit Lambda-Calculator

`http://joerg.endrullis.de/lambdaCalculator/`

- Tupel-Konstruktor und Selektoren als Ausdrücke im Interpreter
- gegenseitige Rekursion (f, g) als Fixpunkt (`Rec`) einer geeigneten Funktion (benutzt Tupel)
- Hausaufgabe: vorige Aufgabe (im Interpreter) ohne `Rec`, sondern mit einem Fixpunkt-Kombinator

- Tupel-Konstruktor und Selektoren als Ausdrücke im Interpreter
- gegenseitige Rekursion (f, g) als Fixpunkt (`Rec`) einer geeigneten Funktion (benutzt Tupel)
- Hausaufgabe: vorige Aufgabe (im Interpreter) ohne `Rec`, sondern mit einem Fixpunkt-Kombinator

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
```

Übung Fixpunkte

- Fixpunkt der Folge $F^k(\bot)$ für

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

- Fixpunkt der Folge $F^k(\bot)$ für

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

- Turing-Fixpunkt-Kombinator mit Lambda-Calculator

`http://joerg.endrullis.de/lambdaCalculator/`

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 )
```

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.

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 ()
```

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 -> ...
```

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 (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 -> ... )
```