

Compilerbau Vorlesung WS 2008–11,13,15,17,19,SS 22,24

Johannes Waldmann, HTWK Leipzig

26. April 2024

– Typeset by FoilTeX –

Organisatorisches (Ü KW 15)

- Skript (Woche für Woche) <https://www.imn.htwk-leipzig.de/~waldmann/lehre.html>
- Quelltexte aus VL, Diskussion Hausaufgaben <https://gitlab.dit.htwk-leipzig.de/johannes.waldmann/cb-ss24> *Einschreiben!* wird dann *private*.
- autotool <https://autotool.imn.htwk-leipzig.de/new/vorlesung/325/aufgaben/aktuell> . *Einschreiben!*
- Wdhlg: Arbeiten im Pool: \$PATH, ghci (9.8.2), mit git(lab.dit) (ssh-keygen, .ssh/id_rsa.pub)
- Haskell-Tooling (cabal, ghc -haddock, :doc)
- Diskussion Klausur PPS WS23 bei Bedarf

– Typeset by FoilTeX –

1

Beispiel: C-Compiler

```
• int gcd (int x, int y) {  
    while (y>0) { int z = x%y; x = y; y = z;  
    return x; }
```

• gcc -S -O2 gcd.c erzeugt gcd.s:

```
.L3:    movl    %edx, %r8d ; cld ; idivl  %r8d  
    movl    %r8d, %eax ; testl  %edx, %edx  
    jg     .L3
```

Ü: was bedeutet `cld`, warum ist es notwendig?

Ü: welche Variable ist in welchem Register?

• identischer (!) Assembler-Code für

```
int gcd_func (int x, int y) {  
    return y > 0 ? gcd_func (y,x % y) : x;  
}
```

• vollständige Quelltexte: siehe Repo

• Bsp Java-Kompilation: <https://www.imn.htwk-leipzig.de/~waldmann/etc/safe-speed/>

– Typeset by FoilTeX –

2

– Typeset by FoilTeX –

3

– Typeset by FoilTeX –

4

– Typeset by FoilTeX –

5

– Typeset by FoilTeX –

6

– Typeset by FoilTeX –

7

Inhalt der Vorlesung

Konzepte von Programmiersprachen

- Semantik von einfachen (arithmetischen) Ausdrücken
- lokale Namen, • Unterprogramme (Lambda-Kalkül)
- Zustandsänderungen (imperative Prog.)
- Continuations zur Ablaufsteuerung

realisieren durch

- Interpretation, • Kompilation

Hilfsmittel:

- Theorie: Inferenzsysteme (f. Auswertung, Typisierung)
- Praxis: Haskell, Monaden (f. Auswertung, Parser)

– Typeset by FoilTeX –

8

Einleitung: Sprachverarbeitung

- mit Interpreter:
 - Quellprogramm, Eingaben $\xrightarrow{\text{Interpreter}}$ Ausgaben
- mit Compiler:
 - Quellprogramm $\xrightarrow{\text{Compiler}}$ Zielprogramm
 - Eingaben $\xrightarrow{\text{Zielprogramm}}$ Ausgaben
- Mischform:
 - Quellprogramm $\xrightarrow{\text{Compiler}}$ Zwischenprogramm
 - Zwischenprogramm, Eingaben $\xrightarrow{\text{virtuelle Maschine}}$ Ausgaben
- reale Maschine (CPU) ist Interpreter für Maschinensprache (Interpretation in Hardware, in Microcode)
- gemeinsam ist: syntaxgesteuerte Semantik (Ausführung oder Übersetzung)

– Typeset by FoilTeX –

9

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', <https://web.archive.org/web/20030603185429/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/>
- J. Waldmann: *Das M-Wort in der Compilerbauvorlesung*, Workshop der GI-Fachgruppe Prog. Spr. und Rechnerkonzepte, 2013 <http://www.imn.htwk-leipzig.de/~waldmann/talk/13/fg214/>

– Typeset by FoilTeX –

10

Anwendungen von Techniken des Compilerbaus

- Implementierung höherer Programmiersprachen
- architekturenspezifische Optimierungen (Parallelisierung, Speicherhierarchien)
- Entwurf neuer Architekturen (RISC, spezielle Hardware)
- Programm-Übersetzungen (Binär-Übersetzer, Hardwaresynthese, Datenbankanfragesprachen)
- Software-Werkzeuge (z.B. Refaktorisierer)
- domainspezifische Sprachen

– Typeset by FoilTeX –

11

Organisation der Vorlesung

- pro Woche eine Vorlesung, eine Übung.
- in Vorlesung, Übung und Hausaufgaben:
 - Theorie,
 - Praxis: Quelltexte (weiter-)schreiben
- Prüfungszulassung: regelmäßiges und erfolgreiches Bearbeiten von Übungsaufgaben
- Prüfung: Klausur (120 min, keine Hilfsmittel)
Bei Interesse und nach voriger Absprache: Ersatz eines Teiles der Klausur durch vorherige Hausarbeit z.B. Reparaturen an autotool-Aufgaben oder anderem open-source-Projekt (Ihrer Wahl), bei denen Techniken des Compilerbaus angewendet werden

– Typeset by FoilTeX –

12

Beispiel: Interpreter f. arith. Ausdrücke

```
data Exp = Const Integer
         | Plus Exp Exp | Times Exp Exp
         deriving ( Show )

ex1 :: Exp
ex1 =
  Times ( Plus ( Const 1 ) ( Const 2 ) ) ( Const 3 )

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
```

das ist syntax-gesteuerte Semantik:
Wert des Terms wird aus Werten der Teilterme kombiniert

– Typeset by FoilTeX –

13

Beispiel: 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 )
extend n w e = \ m -> if m == n then w else e m
value :: Env -> Exp -> Integer
value env x = case x of
  Ref n -> env n
  Let n x b -> value (extend n (value env x) env) b
  Const i -> i
  Plus x y -> value env x + value env y
  Times x y -> value env x * value env y
test2 = value ( \ _ -> 42 ) ex2
```

– Typeset by FoilTeX –

14

Bezeichner sind Strings — oder nicht?

- ... | Let String Exp Exp — wirklich?
- es gilt `type String = [Char]`, also
 - einfach verkettete Liste von Zeichen
 - mit Bedarfsauswertung (lazy Konstruktoren)
- das ist
 - ineffizient (in Platz *und* Zeit)
 - egal (für unseren einfachen Anwendungsfall)
 - gefährlich (wenn man es für andere Anwendungen übernimmt)
- deswegen jetzt schon Diskussion ...
 - von alternativen Implementierungen
 - und wie man diese versteckt

– Typeset by FoilTeX –

15

Datentypen für Folgen (von Zeichen)

- `type String = [Char]`: einfach verkettet, lazy: ist in den allermeisten Fällen unzuweckmäßig
- `data Vector a: Array` (d.h., zusammenhängender Speicherbereich, deswegen effiziente Indizierung) mit kostenlosem *slicing* (Abschnitt-Bildung)
- `data ByteString: ≈ Vektor von Bytes` (d.h., für rein binären Datenaustausch)
- `data Text` (aus Modul `Data.Text`) *efficient packed, immutable Unicode text type*, (d.h., Zeichen = Bytefolge)
- Modul `Data.Text.Lazy`: lazy Liste von (strikten) `Text`-Abschnitten, für Stream-Verarbeitung

– Typeset by FoilTeX –

16

Verstecken von Implementierungsdetails

- Implementierung direkt sichtbar:
`data Exp = ... | Let Text Exp Exp`
- Verschieben der Implementierungs-Entscheidung:
`type Id = Text; data Exp = ... | Let Id Exp`
bleibt aber sichtbar (`type`-Deklarationen werden bei Kompilation immer expandiert)
- Verstecken der Entscheidung:
`modul Id (Id) where data Id = Id Text`
exportiert wird Typ-Name, aber nicht der Konstruktor der Anwender (Importeur) von `Id` sieht `Text` nicht
- `data`-Deklaration mit genau einem Konstruktor:
ersetzen durch `newtype Id = Id Text`
dieser kostet *gar nichts* (keine Zeit, keinen Platz)

– Typeset by FoilTeX –

17

Verwendung standardisierter Namen

- alle benötigten Funktionen (einschl. Konstruktoren) für `Id` implementieren und exportieren (es sind nicht viele)
`eqId::Id->Id->Bool; eqId (Id s) (Id t) = s == t`
- diese spezifischen Namen will sich keiner merken \Rightarrow verwende standardisierte Typklassen, Bsp.
`instance Eq Id where (Id s) == (Id t) = s == t`
der Importeur von `Id` sieht den Namen (`==`) bereits, weil er in `Prelude` definiert ist
- wenn die Implementierung einer standardisierten Klasse eine einfache Delegation ist, kann sie vom Compiler erzeugt werden
`newtype Id = Id Text deriving Eq`

– Typeset by FoilTeX –

18

Einsparung von Konstruktor-Aufrufen

- `-- Implementierung des Konstruktors`
`import qualified Data.Text as T`
`fromString::String->Id; fromString s = Id (T.pack s)`
`-- Anwendung:`
`foo :: Id ; foo = fromString "bar"`
- der Schreibaufwand wird verringert durch
`-- bei Implementierung:`
`import Data.String;`
`instance IsString Id where fromString = T.pack`
`-- bei Anwendung:`
`{-# language OverloadedStrings #-}`
`foo :: Id ; foo = "bar"`

String-Literale sind dann *überladen* \Rightarrow Compiler setzt `fromString` vor jedes (`"bar"` \Rightarrow `fromString "bar"`)

– Typeset by FoilTeX –

19

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)

– Typeset by FoilTeX –

20

– 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 ausgewertet

– Typeset by FoilTeX –

21

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
```

– Typeset by FoilTeX –

22

Übung (effiziente Imp. von Bezeichnern)

- welche Operationen auf `Id` werden benötigt?
 - Konstruktion (`fromString`)
 - Gleichheit
 - Ausgabe (nur für Fehlermeldungen!)
- für `newtype Id = Id Text deriving Eq`:
wie teuer ist Vergleich? wie könnte man das verbessern?
- für `type Env` und `extend` wie angegeben: wie teuer ist das Aufsuchen des Wertes eines Namens in einer Umgebung, die durch `n` geschachtelte `extend` entsteht? wie könnte man das verbessern?
Hinweis: mit `Env` als Funktion: gar nicht.
Welcher andere Typ könnte verwendet werden?

– Typeset by FoilTeX –

23

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)

$$\text{Schreibweise } \frac{P_1, \dots, P_n}{K}$$

- Axiomen (= Regeln ohne Prämissen)

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

- jeder Knoten ist mit einem Objekt beschriftet
- jeder Knoten (mit Vorgängern) entspricht Regel von I
- Wurzel (Ziel) ist mit F beschriftet

Def: $I \vdash F : \iff \exists I\text{-Ableitungsbaum mit Wurzel } F$.

Regel-Schemata

- um unendliche Menge zu beschreiben, benötigt man unendliche Regelmengen
- diese möchte man endlich notieren
- ein *Regel-Schema* beschreibt eine (mglw. unendliche) Menge von Regeln, Bsp: $\frac{(x, y)}{(x - y, y)}$
- Schema wird *instantiiert* durch Belegung der Schema-Variablen
- Bsp: Belegung $x \mapsto 13, y \mapsto 5$ ergibt Regel $\frac{(13, 5)}{(8, 5)}$

Inferenz-Systeme (Beispiel)

- Grundbereich = Zahlenpaare $\mathbb{Z} \times \mathbb{Z}$
- Axiom: $\frac{}{(13, 5)}$
- Regel-Schemata: $\frac{(x, y)}{(x - y, y)}, \frac{(x, y)}{(x, y - x)}$
- gilt $I \vdash (1, 1)$?
- Ü: Beziehung zu einem alten Algorithmus (früh im Studium, früh in der Geschichte der Menschheit)

Primalitäts-Zertifikate

- Satz: $p \in \mathbb{P} \iff \exists g : g$ ist *primitive Wurzel* mod p : $[g^0, g^1, g^2, \dots, g^{p-2}]$ ist Permutation von $[1, 2, \dots, p - 1]$
let {p = 7; g = 3}
in map (\mod p) \$ take (p-1) \$ iterate (*g) 1
[1, 3, 2, 6, 4, 5]
- Inferenzregel $\frac{g_1 : p_1, \dots, g_k : p_k}{g : p}$, falls $p - 1 = q_1^{e_1} \dots q_k^{e_k}$ und $\forall i : g^{(p-1)/q_i} \neq 1 \pmod p$
- Vaughan Pratt, *Each Prime has a Succinct Certificate*, SIAM J. Comp. 1975
- es folgt $\mathbb{P} \in \text{NP} \cap \text{co-NP}$, aber \mathbb{P} *not known to be in P*
- Agrawal, Kayal, Saxena: *Primes is in P*, 2004

Inferenz von Typen

- später implementieren wir das, als statische Analyse im Interpreter/Compiler,
- jetzt geben wir nur die Regel an: $\frac{f : T_1 \rightarrow T_2, x : T_1}{fx : T_2}$
- Bsp. für Verwendung eines Inferenzsystems: Manuel Chakravarty, Gabriele Keller, Simon Peyton Jones, Simon Marlow, *Associated Types with Class*, POPL 2005
- Absch. 4.2 (Fig. 2) Grundbereich: $\Theta | \Gamma \vdash e : \sigma$ means that in type environment Γ and instance environment Θ the expression e has type σ
- Bsp. für ein Regelschema: $\frac{(v : \sigma) \in \Gamma}{\Theta | \Gamma \vdash v : \sigma} (\text{var})$

Inferenz von Werten

- Grundbereich: Aussagen $\text{wert}(p, z)$ mit $p \in \text{Exp}, z \in \mathbb{Z}$
data Exp = Const Integer
| Plus Exp Exp | Times Exp Exp
- Axiome: $\text{wert}(\text{Const } z, z)$
- Regeln:
 $\frac{\text{wert}(X, a), \text{wert}(Y, b)}{\text{wert}(\text{Plus } XY, a + b)}, \frac{\text{wert}(X, a), \text{wert}(Y, b)}{\text{wert}(\text{Times } XY, a \cdot b)}, \dots$
- das ist syntaxgesteuerte Semantik:
für jeden Konstruktor von $p \in \text{Exp}$ gibt es genau eine Regel mit Konklusion $\text{wert}(p, \dots)$

Umgebungen (Spezifikation)

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

Umgebungen (Implementierung)

Umgebung ist (partielle) Funktion von Name nach Wert

Realisierungen: `type Env = String -> Integer`

Operationen:

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

Notation: $e(x)$

- `extend :: String -> Integer -> Env -> Env`

Notation: $e[v := z]$

Beispiel

`lookup (extend "y" 4 (extend "x" 3 empty)) "x"`

entspricht $(\emptyset[x := 3][y := 4])x$

- Typeset by FoilTeX -

32

Übung

1. Primalitäts-Zertifikate

- welche von 2, 4, 8 sind primitive Wurzel mod 101?
 - vollst. Primfaktorzerlegung von 100 angeben
 - ein vollst. Prim-Zertifikat für 101 angeben.
 - bestimmen Sie $2^{(101-1)/5} \bmod 101$ von Hand
- Hinweise: 1. das sind *nicht* 20 Multiplikationen,
2. es wird *nicht* mit riesengroßen Zahlen gerechnet.

2. Geben Sie den vollständigen Ableitungsbaum an für die Auswertung von

```
let {x = 5} in let {y = 7} in x
```

- Typeset by FoilTeX -

32

- Typeset by FoilTeX -

33

Semantische Bereiche

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

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

- typische Verarbeitung:

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

- Typeset by FoilTeX -

34

Continuations

- Programmablauf-Abstraktion durch Continuations:

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

`k` ist die *continuation* (die Fortsetzung im Erfolgsfall)

- eben geschriebenen Code refaktorisieren zu:

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

- Typeset by FoilTeX -

34

- Typeset by FoilTeX -

35

Aufgaben

1. Bool im Interpreter

- Boolesche Literale
- relationale Operatoren (`==`, `<`, o.ä.),
- Inferenz-Regel(n) für Auswertung des `If`
- Implementierung der Auswertung von `if/then/else` mit `with_bool`,

2. Striktheit der Auswertung

- einen Ausdruck `e :: Exp` angeben, für den `value undefined e` eine Exception ist (zwei mögliche Gründe: nicht gebundene Variable, Laufzeit-Typfehler)
- mit diesem Ausdruck: diskutiere Auswertung von `let {x = e} in 42`

- Typeset by FoilTeX -

36

- Typeset by FoilTeX -

37

3. bessere Organisation der Quelltexte

- Cabalisierung (Quelltexte in `src/`, Projektbeschreibungsdarstellung `cb.cabal`), Anwendung: `cabal repl` usw.
- separate Module für `Exp`, `Env`, `Value`,

- Typeset by FoilTeX -

38

- Typeset by FoilTeX -

39

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 { par :: Name , body :: Exp }
  | App { fun :: Exp , arg :: Exp }
```

- konkrete Syntax:

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

- konkrete Syntax (Alternative):

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

Semantik (mit Funktionen)

- erweitere den Bereich der Werte:


```
data Val = ... | ValFun ( Val -> Val )
```
- erweitere Interpreter:


```
value :: Env -> Exp -> Val
value env x = case x of
  ... | Abs n b -> _ | App f a -> _
```
- mit Hilfsfunktion `with_fun :: Val -> ...`
- Testfall (in konkreter Syntax)


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

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

Mehrstellige Funktionen

- ...simulieren durch einstellige:
- mehrstellige Abstraktion:


```
\ x y z -> B := \x -> (\y -> (\z -> B ))
```
 - mehrstellige Applikation:


```
f P Q R := ((f P) Q) R
```

 (die Applikation ist links-assoziativ)
 - der Typ einer mehrstelligen Funktion:


```
T1 -> T2 -> T3 -> T4 :=
  T1 -> (T2 -> (T3 -> T4))
```

 (der Typ-Pfeil ist rechts-assoziativ)

Semantik mit Closures

- bisher: `ValFun` ist Funktion als Datum der Gastsprache


```
value env x = case x of ...
  Abs n b -> ValFun $ \ v ->
    value (extend n v env) b
  App f a ->
    with_fun ( value env f ) $ \ g ->
    with_val ( value env a ) $ \ v -> g v
```
- alternativ: Closure: enthält Umgebung `env` und Code `b`

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

Closures (Spezifikation)

- Closure konstruieren (Axiom-Schema):

$$\overline{\text{wert}(E, \lambda n.b, \text{Clos}(E, n, b))}$$
- Closure benutzen (Regel-Schema, 3 Prämissen)

$$\frac{\text{wert}(E_1, f, \text{Clos}(E_2, n, b)), \text{wert}(E_1, a, w), \text{wert}(E_2[n := w], b, r)}{\text{wert}(E_1, fa, r)}$$
- Ü: Inferenz-Baum für Auswertung des vorigen Testfalls (geschachtelte Let) zeichnen
- ... oder Interpreter so erweitern, daß dieser Baum ausgegeben wird

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 selbstgebautem Interpreter ausrechnen
- mit Haskell ausrechnen
- in JS (node) ausrechnen

Repräsentation von Fehlern

- Fehler explizit im semantischen Bereich des Interpreters repräsentieren (anstatt als Exception der Gastsprache)

```
data Val = ... | ValErr Text
```

- strikte Semantik: `ValErr` *niemals* in Umgebung (bei Let-Bindung oder UP-Aufruf)
- Ü: realisieren durch Aufruf (an geeigneten Stellen) von

```
with_val :: Val -> (Val -> Val) -> Val  
with_val v k = case v of  
  ValErr _ -> v  
  _ -> k v
```

Übungen

1. eingebaute primitive Rekursion (Induktion über Peano-Zahlen):

implementieren Sie die Funktion

```
fold :: r -> (r -> r) -> N -> r
```

Testfall: `fold 1 (\x -> 2*x) 5 == 32`

durch `data Exp = .. | Fold ..` und neuen Zweig
in `value`

Wie kann man damit die Fakultät implementieren?

2. alternative Implementierung von Umgebungen

- bisher `type Env = Id -> Val`

- **jetzt** `type Env = Data.Map.Map Id Val` oder `Data.HashMap`

Messung der Auswirkungen: 1. Laufzeit eines Testfalls, 2. Laufzeiten einzelner UP-Aufrufe (profiling)