

# M\*\*\*\*\* in der Compilerbauvorlesung

Johannes Waldmann (HTWK Leipzig)

Workshop FG 214, Bad Honnef

1/16

## Übersetzerbau

Was:

- ▶ konkrete und abstrakte Syntax von Programmen
- ▶ (syntaxgesteuerte) Definition der Semantik von Programmen (Interpreter)
- ▶ Semantikerhaltende Transformationen (Compiler)

Warum:

- ▶ Verständnis der in Programmiersprachen benutzten Konzepte
- ▶ Definition und Realisierung von domainspezifischen Sprachen
- ▶ Methoden zur Spezifikation von Software

2/16

## Compilerbau im Informatikstudium

Bachelor (Pflichtfächer)

- ▶ 1. Sem: Theor. Grundlagen; Programmierung Java und C
- ▶ 2. Sem: Algorithmen und Datenstrukturen
- ▶ 3. Sem: Softwaretechnik
- ▶ 4. Sem: Fortgeschrittene (d.h. deklarative) Programmierung

Master (Pflichtfächer)

- ▶ 1. Sem: Prinzipien von Programmiersprachen
- ▶ 3. Sem: Theoretische Informatik

Master (Wahl)

- ▶ 3. Sem: Compilerbau

3/16

## Compilerbau - Lehrbücher

klassisch:

- ▶ Aho, Ullman: *Drachenbuch* (1977 ...) Interpreter/Compiler für imperative Sprache, in imperativer Sprache (lex, yacc, C)
- ▶ Steele: *Lambda, the Ultimate Imperative/Declarative* (1976) Interpreter/Compiler für funktionale Sprache, in funktionaler Sprache (LISP)

modern:

- ▶ Turbak, Gifford, Sheldon: *Design Concepts in Programming Languages* (2008)

4/16

## Design Concepts in Progr. Lang.

Franklyn Turbak, David Gifford, Mark A. Sheldon

Inhalt:

- ▶ Dynamic Semantics (functions, naming, state, control, ...)
- ▶ Static Semantics (simple types, polymorphism, type reconstruction, ...)
- ▶ Pragmatics (compilation, garbage collection)

Besonderheiten:

- ▶ „Implementierungssprache“ ist Mathematik (Inferenzsysteme)
- ▶ konkrete Syntax wird nicht diskutiert

meine Vorlesung dazu: „Mathematik“  $\Rightarrow$  Haskell (semantische Bereiche  $\Rightarrow$  Monaden)

5/16

## Motivation für Maybe-Monade (I)

```
type Value = Integer ; type Name = String
type Env = Name -> Value
eval :: Env -> Expression -> Value
eval env x = case x of
  Constant i -> i ; Variable v -> env v
  Plus y z -> eval env y + eval env z
```

was passiert bei Division durch 0?

```
data Maybe a = Nothing | Just a
eval :: Env -> Expression -> Maybe Value
eval env x = case x of
  Div y z -> if 0 == ... then Nothing e
```

6/16

## Motivation für Maybe-Monade (II)

Resultattyp `Maybe Value` erzwingt solchen Code:

```
Plus y z -> case eval env y of
  Nothing -> Nothing
  Just u -> case eval env z of
    Nothing -> Nothing
    Just v -> Just $ u + v
```

kann effizienter notiert werden mit

```
bind :: Maybe a -> (a -> Maybe b) -> Maybe b
bind m f = case m of
  Nothing -> Nothing ; Just x -> f x
```

```
bind (eval env y) $ \ u ->
bind (eval env z) $ \ v -> Just $ u + v
```

7/16

## Die Konstruktorklasse Monad

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

(so daß  $f @@ g = \ x \rightarrow f\ x \gg= g$  assoziativ ist mit `return` rechts- und links-neutral)

NB: Assoziativität implizit verwendet bei der `do`-Notation.

```
do { x <- a ; y <- b ; c }
a >>= \ x -> ( b >>= \ y -> c )
(a >>= \ x -> b) >>= \ y -> c
```

8/16

## Semantischer Bereich als Monade

- ▶ definiere Typ

```
type Store = M.Map Addr Value
data Action a = Action (Store -> (a, Store))
```

- ▶ definiere typspezifische Operationen

```
new :: Action Addr
new = Action $ \ s -> (M.size, s)
put :: Addr -> Value -> Action ()
put a v = Action $ \ s -> ((), M.insert a v s)
```

- ▶ definiere Monaden-Operationen

```
instance Monad Action where
  Action a >>= f = Action $ \ s0 ->
    let (r0,s1) = a s0; Action b = f r0 in b s1
```

- ▶ Änderungen am Interpreter selbst

(wenige - das ist der didaktische Vorteil)

9/16

## Rekursion in versch. Bereichen

im Back-End durch

- ▶ Rekursion in der Implementierungssprache im Front-End:

- ▶ rein funktional: Fixpunktkombinator
- ▶ mit Speicher und Vorwärtsreferenzen
- ▶ mit Continuations

Übung (aus Hofstadter: Gödel-Escher-Bach):

```
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
```

... für welche  $x$  gilt  $f(x) \neq g(x)$ ?

10/16

## Konkrete Syntax (Parser)

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

Monad-Instanz ist Kombination von

- ▶ Zustandsmonade (Zustand ist `[c]`)
- ▶ Nichtdeterminismus (`... -> [ ... ]`)

primitive Parser und Kombinatoren „von Hand“

API-kompatibel zu `Parsec`

- ▶ eingeschränkter Nichtdeterminismus, asymmetrische Komposition
- ▶ committed choice
- ▶ Quelltextpositionen, Fehlermeldungen

11/16

## Abstrakte Interpretation

für statische Semantik:

- ▶ statt Wert eines Ausdrucks wird sein Typ berechnet.

```
type Env = Name -> Type
eval :: Env -> Expression -> Maybe Type
```

- ▶ nur geringe Änderungen am Interpreter-Quelltext (wegen der `bind/return`-Notation)

Arten der Typisierung

- ▶ einfach getypt (Typdekl. für Let-Bindungen)
- ▶ polymorph (explizite Type-Abs., Type-Appl.)
- ▶ Rekonstruktion für einfache Typen (`eval` erzeugt Constraint-System)

12/16

## Kompilation

Ablauf:

- ▶ continuation passing (Programmablauf explizit)
- ▶ closure conversion (Umgebungen explizit)
- ▶ lifting (Unterprogramme global)
- ▶ Registervergabe (alle Argumente in Registern)

Eigenschaften:

- ▶ Zielsprache nach jedem Schritt ist (syntaktisch eingeschränkte) Teilsprache der Startsprache
- ▶ Semantik-Erhaltung kann durch den Original-Interpreter überprüft werden.
- ▶ Resultat kann in flaches C-Programm übersetzt werden (mit Boehm-GC)

13/16

## Zusammenfassung

Haskell im Compilerbau ergibt

- ▶ mathematische Klarheit (wg. Nebenwirkungsfreiheit): der Text des Interpreters *ist* ein Gleichungssystem, das die Semantik definiert
- ▶ übersichtliche Programme (wg. Monaden) durchgehende Benutzung von `bind/return`
- ▶ kurze Programme (wg. Fkt. höherer Ordnung) Quelltextzeilen für Interpreter: 250, Parser: 380, Typisierung: 350, Kompilation: 320

Plan: Benutzung des Interpreters bereits für (autotool-)Aufgaben in *Prinzipien von Programmiersprachen*

14/16

## Diskussion

- ▶ Semantikbausteine modular durch Monadentransformatoren?
- ▶ Higher Order Abstract Syntax?
- ▶ Parser und Prettyprinter gleichzeitig?
- ▶ syntaktische Funktionen (für Kompilation) erfordern viel Hilfscode (oder `Data.Generics`, das ist aber nicht selbsterklärend)
- ▶ (naive) Closure Conversion erzwingt Garbage Collection im Laufzeitsystem

15/16

## Quellen

- ▶ Turbak, Gifford, Sheldon: *Design Concepts in Programming Languages*, 2008. <http://mitpress.mit.edu/books/design-concepts-programming-languages>
- ▶ Steele: *Lambda the Ultimate ...*, 1976. <http://library.readscheme.org/page1.html>
- ▶ Waldmann: *Vorlesung Compilerbau*, 2012. Folien <http://www.imn.htwk-leipzig.de/~waldmann/edu/ws12/cb/folien/main/>, Quelltexte <http://dfa.imn.htwk-leipzig.de/cgi-bin/gitweb.cgi?p=ws12-cb.git> [git clone git://dfa.imn.htwk-leipzig.de/srv/g](http://git clone git://dfa.imn.htwk-leipzig.de/srv/g)
- ▶ Leijen and Meyer: *Parsec: Direct Style Monadic Parser Combinators for the Real World*, 2001. <http://legacy.cs.uu.nl/daan/pubs.html#parsec>

16/16