

# Harpy

Dynamische Codegenerierung mit Haskell

Martin Grabmüller und Dirk Kleeblatt

Fachgebiet Übersetzerbau und Programmiersprachen  
Fakultät IV – Elektrotechnik und Informatik  
Technische Universität Berlin

ÜBB-Kolloquium, 1. Juni 2007



## Einführung

- Dynamische Codegenerierung
- Haskell

## Harpy

- Aufbau
- Beispiel

## Implementierung

- CodeGen-Monade
- Codeerzeugung
- “High-level” Assembler
- Generierung von Aufruffunktionen
- Codegenerator-Kombinatoren
- Disassembler

## Abschluss

- Weitere Arbeiten

## Einführung

- Dynamische Codegenerierung
- Haskell

## Harpy

- Aufbau
- Beispiel

## Implementierung

- CodeGen-Monade
- Codeerzeugung
- “High-level” Assembler
- Generierung von Aufruffunktionen
- Codegenerator-Kombinatoren
- Disassembler

## Abschluss

- Weitere Arbeiten

Thema dieses Vortrags: Design und Implementierung einer Bibliothek für dynamische Codegenerierung zur Benutzung in Haskell-Programmen.

**Thema dieses Vortrags:** Design und Implementierung einer Bibliothek für dynamische Codegenerierung zur Benutzung in Haskell-Programmen.

- ▶ Warum dynamische Codegenerierung?
- ▶ Warum Haskell?

# Warum dynamische Codegenerierung?

Erzeugung und Ausführung von Code zur Laufzeit

# Warum dynamische Codegenerierung?

## Erzeugung und Ausführung von Code zur Laufzeit

Einsatzgebiete für dynamische Codegenerierung:

- ▶ Just-in-time Kompilierung in Virtuellen Maschinen
- ▶ Implementierung von Programmiersprachen
- ▶ Programmspezialisierung zur Laufzeit

# Warum dynamische Codegenerierung?

## Erzeugung und Ausführung von Code zur Laufzeit

Einsatzgebiete für dynamische Codegenerierung:

- ▶ Just-in-time Kompilierung in Virtuellen Maschinen
- ▶ Implementierung von Programmiersprachen
- ▶ Programmspezialisierung zur Laufzeit

Vor- und Nachteile:

- + Erzeugung des bestmöglichen Codes für Maschine/System
- + Effiziente Unterstützung für dynamisches Laden, Reflection, Debugging

# Warum dynamische Codegenerierung?

## Erzeugung und Ausführung von Code zur Laufzeit

Einsatzgebiete für dynamische Codegenerierung:

- ▶ Just-in-time Kompilierung in Virtuellen Maschinen
- ▶ Implementierung von Programmiersprachen
- ▶ Programmspezialisierung zur Laufzeit

Vor- und Nachteile:

- + Erzeugung des bestmöglichen Codes für Maschine/System
- + Effziente Unterstützung für dynamisches Laden, Reflection, Debugging
- Laufzeit und Speicherbedarf der Codegenerierung
- Komplexere Laufzeitsysteme
- Späte Fehlermeldungen

# Warum dynamische Codegenerierung?

## Erzeugung und Ausführung von Code zur Laufzeit

Einsatzgebiete für dynamische Codegenerierung:

- ▶ Just-in-time Kompilierung in Virtuellen Maschinen
- ▶ Implementierung von Programmiersprachen
- ▶ Programmspezialisierung zur Laufzeit

Vor- und Nachteile:

- + Erzeugung des bestmöglichen Codes für Maschine/System
- + Effiziente Unterstützung für dynamisches Laden, Reflection, Debugging
- Laufzeit und Speicherbedarf der Codegenerierung
- Komplexere Laufzeitsysteme
- Späte Fehlermeldungen (wie in anderen dynamischen Systemen)

Haskell ist Implementierungssprache:

- ▶ Formulierung in Hochsprache
- ▶ Benutzung abstrakter Sprachmittel (Monaden, Typklassen, Meta-Programmierung)
- ▶ Gutes Foreign-Function-Interface (FFI) für Low-level Programmierung
- ▶ Stabiler, effizienter, gut unterstützter Compiler verfügbar

# Warum Haskell?

Haskell ist Implementierungssprache:

- ▶ Formulierung in Hochsprache
- ▶ Benutzung abstrakter Sprachmittel (Monaden, Typklassen, Meta-Programmierung)
- ▶ Gutes Foreign-Function-Interface (FFI) für Low-level Programmierung
- ▶ Stabiler, effizienter, gut unterstützter Compiler verfügbar

Außerdem:

- ▶ Geplante Einsatzprojekte sind Haskell-Programme

## Einführung

Dynamische Codegenerierung

Haskell

## Harpy

Aufbau

Beispiel

## Implementierung

CodeGen-Monade

Codeerzeugung

“High-level” Assembler

Generierung von Aufruffunktionen

Codegenerator-Kombinatoren

Disassembler

## Abschluss

Weitere Arbeiten

Harpy erlaubt. . .

- ▶ die Programmierung von *Codegeneratoren*
- ▶ die Komposition von *Codegeneratoren*
- ▶ den Aufruf von *Codegeneratoren*

Harpy erlaubt. . .

- ▶ die Programmierung von *Codegeneratoren*
- ▶ die Komposition von *Codegeneratoren*
- ▶ den Aufruf von *Codegeneratoren*

und

- ▶ die Ausführung des erzeugten Codes.

Harpy erlaubt. . .

- ▶ die Programmierung von *Codegeneratoren*
- ▶ die Komposition von *Codegeneratoren*
- ▶ den Aufruf von *Codegeneratoren*

und

- ▶ die Ausführung des erzeugten Codes.

Codegeneratoren sind monadische Operationen der CodeGen-Monade.

Sicht des Benutzers:

- ▶ Formulierung von Codegeneratoren durch Benutzung monadischer Operationen der Harpy-Bibliothek
- ▶ Ausführung der Generatoren

Aufgabe der Bibliothek:

- ▶ Verwaltung von Speicher für Maschinencode, Sprungmarken
- ▶ Ausgabe der Instruktionsbitmuster in Speicher
- ▶ Ausführung des Maschinen-Codes, Rückkehr in Haskell-Welt

## Beispiel: Berechnung der Fakultät

fac = do

```
mov  eax  1
mov  ecx  (Disp 8, ebp)
jmp  loopTest
loopStart @@ mul  ecx
sub  ecx  1
loopTest @@ cmp  ecx  0
jne  loopStart
```

# Beispiel: Berechnung der Fakultät

fac = do

```
mov  eax  1
mov  ecx  (Disp 8, ebp)
jmp  loopTest
loopStart @@ mul  ecx
sub  ecx  1
loopTest @@ cmp  ecx  0
jne  loopStart
```

## Beispiel: Berechnung der Fakultät

```
fac = do loopTest  <- newLabel  
        loopStart <- newLabel
```

```
mov  eax  1  
mov  ecx (Disp 8, ebp)  
jmp  loopTest  
loopStart @@ mul  ecx  
sub  ecx  1  
loopTest @@ cmp  ecx  0  
jne  loopStart
```

## Beispiel: Berechnung der Fakultät

```
fac = do loopTest  <- newLabel
        loopStart <- newLabel
        push ebp
        mov  ebp esp
        push ecx
        mov  eax 1
        mov  ecx (Disp 8, ebp)
        jmp  loopTest
loopStart @@ mul ecx
        sub  ecx 1
loopTest @@ cmp ecx 0
        jne  loopStart
        pop  ecx
        pop  ebp
        ret
```

## Beispiel: Berechnung der Fakultät

```
fac = do loopTest  <- newLabel
        loopStart <- newLabel
        push ebp
        mov  ebp esp
        push ecx
        mov  eax (1 :: Word32)
        mov  ecx (Disp 8, ebp)
        jmp  loopTest
loopStart @@ mul ecx
        sub  ecx (1 :: Word32)
loopTest @@ cmp ecx (0 :: Word32)
        jne  loopStart
        pop  ecx
        pop  ebp
        ret
```

- ▶ Aufruffunktion deklarieren:

```
$(callDecl "callFac" [t|Word32 -> Word32|])
```

- ▶ Aufruffunktion deklarieren:

```
$(callDecl "callFac" [t|Word32 -> Word32|])
```

- ▶ Aufruf des erzeugten Codes:

```
runFac :: CodeGen e s ()  
runFac = do fac  
           x <- liftIO readLn  
           y <- callFac x  
           liftIO (putStrLn (show y))
```

- ▶ Aufruffunktion deklarieren:

```
$(callDecl "callFac" [t|Word32 -> Word32|])
```

- ▶ Aufruf des erzeugten Codes:

```
runFac :: CodeGen e s ()  
runFac = do fac  
            x <- liftIO readLn  
            y <- callFac x  
            liftIO (putStrLn (show y))
```

- ▶ Ausführung der Codeerzeugungsmonade:

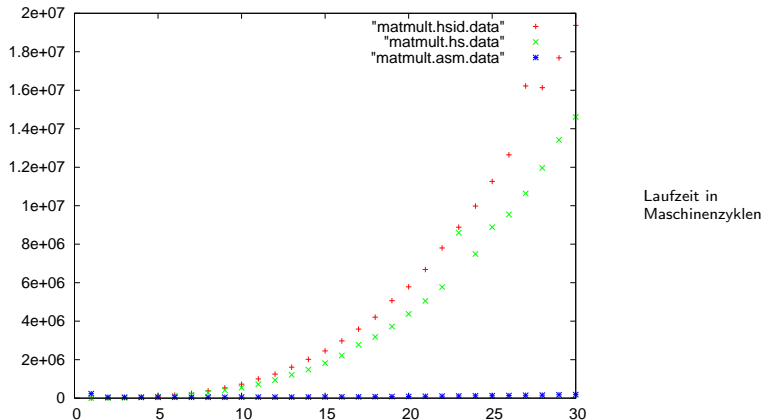
```
main = do  
    (finalState, result) <- runCodeGen runFac () ()  
    return ()
```

# Beispiel: Matrixmultiplikation

- ▶ Multiplikation von zwei  $n \times n$ -Matrizen ( $1 \leq n \leq 30$ )
- ▶ Drei Varianten:
  - ▶ Haskell (Unboxed double-Arrays in IO-Monade)
  - ▶ Haskell (imperativ, direkte unsichere Addressierung)
  - ▶ Mit Harpy erzeugter Maschinencode

# Beispiel: Matrixmultiplikation

- ▶ Multiplikation von zwei  $n \times n$ -Matrizen ( $1 \leq n \leq 30$ )
- ▶ Drei Varianten:
  - ▶ Haskell (Unboxed double-Arrays in IO-Monade)
  - ▶ Haskell (imperativ, direkte unsichere Addressierung)
  - ▶ Mit Harpy erzeugter Maschinencode



## Einführung

- Dynamische Codegenerierung
- Haskell

## Harpy

- Aufbau
- Beispiel

## Implementierung

- CodeGen-Monade
- Codeerzeugung
- “High-level” Assembler
- Generierung von Aufruffunktionen
- Codegenerator-Kombinatoren
- Disassembler

## Abschluss

- Weitere Arbeiten

Stellt Operationen zur Verfügung für

- ▶ Verwaltung von Speicher für Maschinencode (Code-Puffer)
- ▶ Verwaltung von Sprungmarken
- ▶ Operationen zur Ausgabe von 8- und 32-Bit-Werten (Code-Puffer anhängen bzw. an absolute Adressen schreiben)
- ▶ Verwaltung von benutzerdefiniertem Zustand
- ▶ Interface zum Aufrufen und Disassemblieren von erzeugtem Code

Zur Erleichterung der Programmierung:

- ▶ Template-Haskell-Funktion zur Erzeugung von FFI-Aufruf-Funktionen

## Modul zur Erzeugung von Maschinencode

- ▶ Benutzt CodeGen-Monade
- ▶ Operationen der Bauart:
  - ▶ `x86_alu_mem_reg`: Arithmetik mit erster Quelle und Ziel im Speicher, Quelle in Register
  - ▶ `x86_div_reg`: Ganzzahl-Division mit erster Quelle und Ziel in `eax:edx`, zweite Quelle in Register
  - ▶ `x86_jump_membase`: Sprung an Adresse, die aus Register und Konstante gebildet wird.
- ▶ Bildet direkt Instruktionssatz und Adressierungsmodi der x86-Architektur ab.

- ▶ Durch Typklassen werden Adressierungsmodi automatisch erkannt

# “High-level” Assembler

- ▶ Durch Typklassen werden Adressierungsmodi automatisch erkannt
- ▶ Zu jedem Befehl gibt es eine Typklasse:

```
class Add a b where  
  add :: a -> b -> CodeGen e s ()
```

# “High-level” Assembler

- ▶ Durch Typklassen werden Adressierungsmodi automatisch erkannt

- ▶ Zu jedem Befehl gibt es eine Typklasse:

```
class Add a b where
  add :: a -> b -> CodeGen e s ()
```

- ▶ Zu jeder Adressierungsart eine Instanz:

```
instance Add Reg32 Word32 where
  add (Reg32 dest) imm =
    x86_alu_reg_imm x86_add dest (fromIntegral imm)
instance Add Addr Word32 where
  add (Addr dest) imm =
    x86_alu_mem_imm x86_add dest (fromIntegral imm)
```

# “High-level” Assembler

- ▶ Durch Typklassen werden Adressierungsmodi automatisch erkannt

- ▶ Zu jedem Befehl gibt es eine Typklasse:

```
class Add a b where  
  add :: a -> b -> CodeGen e s ()
```

- ▶ Zu jeder Adressierungsart eine Instanz:

```
instance Add Reg32 Word32 where  
  add (Reg32 dest) imm =  
    x86_alu_reg_imm x86_add dest (fromIntegral imm)  
instance Add Addr Word32 where  
  add (Addr dest) imm =  
    x86_alu_mem_imm x86_add dest (fromIntegral imm)
```

```
add eax edx          -- => x86_alu_reg_reg
```

# “High-level” Assembler

- ▶ Durch Typklassen werden Adressierungsmodi automatisch erkannt

- ▶ Zu jedem Befehl gibt es eine Typklasse:

```
class Add a b where
  add :: a -> b -> CodeGen e s ()
```

- ▶ Zu jeder Adressierungsart eine Instanz:

```
instance Add Reg32 Word32 where
  add (Reg32 dest) imm =
    x86_alu_reg_imm x86_add dest (fromIntegral imm)
instance Add Addr Word32 where
  add (Addr dest) imm =
    x86_alu_mem_imm x86_add dest (fromIntegral imm)
```

```
add eax edx          -- => x86_alu_reg_reg
add (Disp 8, ebp) ebx -- => x86_alu_membase_reg
```

Problem: Aufruf von generiertem Code erfordert wrapper-Deklarationen:

- ▶ FFI-Wrapper-Funktion: Konvertiert Zeiger in von Haskell aufrufbare Funktion
- ▶ Monadische Operation, die in generierten Code springt

Unsere Lösung: Deklarationen durch Meta-Programmierung generieren

- ▶ GHC unterstützt Template Haskell (TH) zur Erzeugung von Haskell-Code zur Übersetzungszeit
- ▶ Benutzer muss nur einen Funktionsaufruf schreiben

```
$(callDecl "callFunc" [t| Int -> Word32 |])
```

## Generierung von Aufruffunktionen – Beispiel

```
$(callDecl "callFunc" [t| Int -> Word32 |])
```

⇒

```
foreign import ccall safe "dynamic" conv ::  
  FunPtr (Int -> Word32) -> Int -> IO Word32
```

```
callFunc = \ v -> Harpy.CodeGenMonad.CodeGen  
  (\ env (ustate, state) ->  
    do let code = Harpy.CodeGenMonad.firstBuffer state  
        res <- liftIO ((\ c -> conv (castPtrToFunPtr c) v  
                                code)))  
    (return ((ustate, state), Right res))
```

- ▶ Kapseln häufige Muster, z.B.
  - ▶ Funktionsprologe, -epiloge
  - ▶ Fallunterscheidungen
  - ▶ Schleifen
  - ▶ Sichern/Wiederherstellen von Registern
- ▶ Beispiel:

```
ifThenElse :: CodeGen UserEnv s r ->    -- Bedingung
            CodeGen UserEnv s r' ->    -- Then-Teil
            CodeGen UserEnv s t ->     -- Else-Teil
            CodeGen UserEnv s ()
```

```
fac2 =  
  do function $  
    saveRegs [ecx]  
    (do mov eax (1 :: Word32)  
       mov ecx (Disp 8, ebp)  
       doWhile cond body)  
where  
cond = do cmp ecx (0 :: Word32)  
        continueBranch x86_cc_ne True  
body = do mul ecx  
        sub ecx (1 :: Word32)
```

Disassembler für x86-Code (IA-32 und x64-Code):

- ▶ Implementierung unter Verwendung von Parser-Kombinatoren
- ▶ Eingabe: Arrays und Listen von Bytes
- ▶ Ausgabe: Listen abstrakter Instruktionen
- ▶ Hilfsfunktionen zur lesbaren Darstellung

## Einführung

Dynamische Codegenerierung

Haskell

## Harpy

Aufbau

Beispiel

## Implementierung

CodeGen-Monade

Codeerzeugung

“High-level” Assembler

Generierung von Aufruffunktionen

Codegenerator-Kombinatoren

Disassembler

## Abschluss

Weitere Arbeiten

- ▶ Unterstützung von Garbage Collection durch getypte Operationen
- ▶ Erweiterungen der Codegenerierungs-Kombinatoren
- ▶ Vervollständigung des Befehlssatzes
- ▶ Speichern/Laden von generiertem Code
- ▶ Steigerung der Effizienz durch Ausnutzung des GHC-Regelsystems zur Programmtransformation
- ▶ Einsatz in anderen Projekten (effiziente Typprüfung von Dependent Types, dynamische Kompilierung funktionaler Programme)

- ▶ Unterstützung von Garbage Collection durch getypte Operationen
- ▶ Erweiterungen der Codegenerierungs-Kombinatoren
- ▶ Vervollständigung des Befehlssatzes
- ▶ Speichern/Laden von generiertem Code
- ▶ Steigerung der Effizienz durch Ausnutzung des GHC-Regelsystems zur Programmtransformation
- ▶ Einsatz in anderen Projekten (effiziente Typprüfung von Dependent Types, dynamische Kompilierung funktionaler Programme)

**Vielen Dank!**