Kapitel 10: Metaprogrammierung

Lernziele dieses Kapitels:

1. Klassifikation: statisch/dynamisch, einstufig/mehrstufig etc.
2. Beispiel: partielle Auswertung der Potenzfunktion
3. speziell MetaOCaml
   - Staging-Annotationen mit Typregeln
   - Cross-stage-Persistence
   - Interpreter-Spezialisierung
4. speziell Template Haskell
   - Quasi-Quotation und Splicing
   - Vereinbarkeit von Typisierung und Flexibilität (`printf`-Beispiel)
   - Quotation-Monade, Generierung frischer Namen
   - Direkter Zugriff auf Haskell-Syntax, Reification

Metaprogrammierung

- Prinzip: Bearbeitung von Objektprogrammen durch Metaprogramme.
- Arten
  - Analyse
  - Transformation
  - Generierung

  von Objektprogrammen

Beispiele für Metaprogramme

- Interpreter
  - analysiert zu interpretierendes Programm
- Compiler
  - analysiert Quellprogramm
  - transformiert Zwischencode
  - generiert Zielprogramm
- Scanner- und Parsergeneratoren
  Objektprogramm ist selbst wieder ein Metaprogramm
  → mehrstufige Metaprogrammierung
- Makroprozessoren
- Generatoren für effiziente Datenbankoperationen

Motivation für Metaprogrammierung

Compiler, Generatoren etc. seit langem verwendet!

Welchen Sinn macht ein neuer Begriff Metaprogrammierung ?

Antwort: Bildung eines neuen Forschungsgebietes

- zum Verständnis der Zusammenhänge
- als gemeinsame Basis für Wissensaustausch
- zur schnellen Entwicklung neuer Tools

Klassifikation der Metaprogrammierung

- nach Bearbeitungzeitpunkt des Objektprogramms
  - zur Compile-Zeit: statisch
  - zur Laufzeit: dynamisch (just-in-time)
- nach Sprachen
  - Metasprache=Objektsprache: homogen
  - sonst: heterogen
- nach Stufen
  - Objektsprache ist selbst Metasprache: mehrstufig
  - sonst: einstufig

Repräsentationen von Objektprogrammen

- String: `"let x = 3 in x+1"`
- Datentyp: `Let ("x",CI 3) (Var "x" :+: CI 1)`

	Erzeugung		Ausschluss von Fehlern	
	Konstante	Komposition	Syntax	Typen
Bsp.:	`"f(x)"`	`F X`	`f(,x)` ?	`2(x)` ?
String	gut	gut	nein	nein
Datentyp	schlecht	gut	ja	nein

gesucht: bessere Repräsentationsmöglichkeit

$\Rightarrow$ Metaprogrammier-Erweiterungen

Vergleich Template Haskell $\leftrightarrow$ MetaOCaml

	Zeitpunkt	Stufen	Typprüfung
Template Haskell	statisch (Compile-Zeit)	einstufig	jeweils bei Generierung eines Ausdrucks
MetaOCaml	dynamisch (Laufzeit)	mehrstufig (unbeschränkt)	einmal am Anfang

Vergleich der Typprüfung

- gemeinsam: keine Typfehler zur Laufzeit des Objektprogramms
- Template-Haskell
  - Programmfragmente: Datentyp für Syntaxbäume
  - generierte Ausdrücke: bekommen endgültigen Typ
  - Abwechselnde Generierung und Typprüfung durch den Compiler
  - Vorteil: teilweiser Ersatz für Dependent-Types (parametrisierte Typen)
  - Nachteil: Typsicherheit von Bibliotheken nur für konkrete Verwendung
- MetaOCaml
  - Typverbindung zwischen Metaprogramm und Objektprogramm
  - Vorteil: Generator typkorrekt $\Rightarrow$ generiertes Programm typkorrekt
  - Nachteil: kein Zugriff auf den Syntaxbaum des Objektprogramms

MetaOCaml

Ziel: Codeoptimierung zur Laufzeit durch, z.B.

1. partielle Auswertung
2. aufrollen von Schleifen

Bsp. für partielle Auswertung:

- definiert Potenzfunktion `pow: float → int → float`
- häufig benutzt mit konstantem Exponenten
- Ziel: Elimination rekursiver Aufrufe, ersetze z.B.
  `pow x 5` durch `let y = x *. x in x *. (y *. y)`
- hier: partielle Auswertung von pow im zweiten Argument und Inlining

Nutzen der partiellen Auswertung

- gewünscht in der Softwareentwicklung:
  Verallgemeinerung, Abstraktion, Problem-orientierte Repräsentationen
- Preis: Ineffizienz durch Interpretationsebenen
- häufiges Vorgehen:
  Verzicht auf Abstraktion oder auf Performanz
- bessere Lösung: partielle Auswertung durch Metaprogrammierung
  Nachteile geringer:
  - statisch: anwachsen des Programmcodes
  - dynamisch: Zeitaufwand für just-in-time-Kompilation

Staging-Annotationen in MetaOCaml

Konstrukt	Beispiel	Ergebnis	Typ
Brackets	`let a = .< 2*4 >.`	`a = .< 2*4 >.`	`int code`
Escape	`let b = .< 9 + .~ a >.`	`b = .< 9+2*4 >.`	`int code`
Run	`let c = .! b`	`c = 17`	`int`

Escape vs. Run

```
# let a = .< 1 + 2 >.;;
val a : ('a, int) code = .<(1 + 2)>.
```

Escape (.~) fügt Codeteil ein, darf nur innerhalb von Brackets stehen

```
# let b = 3 + .~a;;
Wrong level: escape at level 0
```

Run (.!) führt erzeugten Code aus, darf auch innerhalb von Brackets stehen

```
# let c = .< .!a * .~a >.;;
val c : ('a, int) code =
  .< .!((* cross-stage persistent value (as id: a) *)) * (1 + 2))>.
# .!c;;
- : int = 9
```

Typregeln

Brackets

$$\frac{x \ : \ \alpha}{.\!<\!x\!>\!. \ : \ \alpha \ \texttt{code}}$$

Escape

$$\frac{x \ : \ \alpha \ \texttt{code}}{.\tilde{}x \ : \ \alpha}$$

Run

$$\frac{x \ : \ \alpha \ \texttt{code}}{.\,!x \ : \ \alpha}$$

Sichtbarkeitsregeln

- Cross-stage Persistence

  Werte der umschliessenden Ebene sind sichtbar

  – Codeteile: eingebettet mit `.~`

  – sonst: direkte Benutzung (siehe nächste Folie)

- Lexical Scoping (statische Variablenbindung) wird respektiert:

  sei `f y = .< fun x -> x + .~y >.`

  dann ergibt `f .<x>.` den Wert `.< fun x_2 -> x_2 + x >.`

  automatische Umbenennung der lokalen Variablen `x` in `x_2` !

Cross-Stage Persistence (Bsp. 1)

```
val a : int array = [|0; 0; 0; 0; 0|]
# let codeLS = .< a.(2) <- 7 >.;;
val codeLS : ('a, unit) code =
  .<((* cross-stage persistent value (as id: a) *)).(2) <- 7>.
# let codeARG x = .< x.(2) <- 8 >.;;
val codeARG : int array -> ('a, unit) code = <fun>
# .!codeLS; a;;
- : int array = [|0; 0; 7; 0; 0|]
# let a = Array.make 5 0;;
val a : int array = [|0; 0; 0; 0; 0|]
# .!codeLS; a;;
- : int array = [|0; 0; 0; 0; 0|]
# .!(codeARG a); a;;
- : int array = [|0; 0; 8; 0; 0|]
```

Cross-Stage Persistence (Bsp. 2)

```
#  let codeEXT x = .< x.(2) <- 8 >.;;                x ist CSP-Wert
val codeEXT : int array -> ('a, unit) code = <fun>
#  let codeINT = .< fun x -> x.(2) <- 9 >.;;         x ist kein CSP-Wert
val codeINT : ('a, int array -> unit) code = .<fun x_1 -> x_1.(2) <- 9>.
#  let a = Array.make 5 0;;
val a : int array = [|0; 0; 0; 0; 0|]
#   .!(codeEXT a); a;;
- : int array = [|0; 0; 8; 0; 0|]
# let f = .!codeINT;;
val f : int array -> unit = <fun>
# f a; a;;
- : int array = [|0; 0; 9; 0; 0|]
```

Bsp.: automatische Variablen-Umbenennung

ohne Staging:

```
# let rec h n z = if n=0 then z else (fun x -> (h (n-1) (x+z))) n;;
val h : int -> int -> int = <fun>
# h 3 1;;
- : int = 7
```

mit Staging:

```
# let rec h n z = if n=0 then z
                  else .<(fun x ->.~(h (n-1) .<x + .~z>.)) n>.;;
val h : int -> ('a, int) code -> ('a, int) code = <fun>
# let y = h 3 .<1>.;;
val y : ('a, int) code = .<((fun (x_2) -> ((fun (x_3) -> ((fun (x_4)
                    -> ((x_4) + ((x_3) + ((x_2) + 1)))) 1)) 2)) 3)>.
# .! y;;
- : int = 7
```

Bsp.: Potenz-Funktion (1)

ohne Metaprogrammierung (zur Vereinfachung Fall n=0 weggelassen)

```
let rec pow (x:float) (n:int) :float =
  if n = 1
  then x
  else
    if n mod 2 = 0
    then
      let y = pow x (n/2) in y *. y
    else
      x *. pow x (n-1);
```

Bsp.: Potenz-Funktion (2)

Ziel: partielle Auswertung des zweiten Arguments und Inlining

- Meta-Programmteile
- Fragmente des übrig bleibenden Objektcodes

```
let rec pow (x:float) (n:int) :float =
  if n = 1
  then x
  else
    if n mod 2 = 0
    then
      let y = pow x (n/2) in y *. y
    else
      x  *. pow x (n-1) ;
```

Bsp.: Potenz-Funktion (3)

mit Staging-Annotations

- Meta-Programmteile
- Fragmente des übrig bleibenden Objektcodes
- Annotationen

```
let rec pow (x : ('a,float) code) (n:int) : ('a,float) code =
  if n = 1
  then x
  else
    if n mod 2 = 0
    then
      .< let y =  .~(pow x (n/2)) in y *. y >.
    else
      .<  .~x *. .~(pow x (n-1))  >.  ;
```

Bsp.: Potenz-Funktion (4), Anwendung

Antworten des MetaOcaml-Interpreters gekürzt und umformatiert

```
# pow .< 1.5 >. 2;;
.< let (y_2) = 1.5 in ((y_2) *. (y_2)) >.
# pow .< 1.5 >. 3;;
.< (1.5 *. let (y_3) = 1.5 in ((y_3) *. (y_3))) >.
# let pow16 = .< fun x -> .~(pow .<x>. 16) >.;;
.< fun x_2 -> let y_3 = let y_4 = let y_5 = let y_6 = x_2
                                           in y_6 *. y_6
                                  in y_5 *. y_5
                         in y_4 *. y_4
               in y_3 *. y_3 >.
# (.! pow16) 2.0;;
65536.
```

Idealer Entwurf eines Metaprogramms

1. entwerfen des Programms ohne Metaprogrammierung
2. festlegen der Auswertungsreihenfolge der Argumente

Problem: Effizienz im Objektprogramm, Bsp.: `pow .<x>.9`,
modifiziere Fall (`n mod 2 == 0`) der Potenz-Funktion:

- schlechte Lösung: `let y = pow x (n/2) in .< .~y *. .~y >.`
  Ergebnis hat 8 Multiplikationen:

```
.< x*.(((x*.x)*.(x*.x))*.((x*.x)*.(x*.x))) >.
```

- Folie Potenz-Funktion (3): `.< let y = .~(pow x (n/2)) in y *. y >.`
  Ergebnis hat nur 4 Multiplikationen:

```
.< x *. let y_2 = let y_3 = let y_4 = x
                            in y_4 *. y_4
                  in y_3 *. y_3
        in y_2 *. y_2 >.
```

Anwendungsbeispiel: Interpreter

- gegeben: `Interpreter: Programm → Eingabe → Ausgabe`
- das Program ändert sich selten
- gesucht: partielle Auswertung des Interpreters im ersten Argument
- Ziel: Vermeidung des Interpretationsaufwands

Futamura-Projektionen zur Mächtigkeit eines partiellen Auswerters (`PE`):

#	1. Argument	2. Argument	Ergebnis
1.	`Interpreter`	`Quellprogramm`	`Zielprogramm: Eingabe→Ausgabe`
2.	`PE`	`Interpreter`	`Compiler: Quellprogramm→Zielprogramm`
3.	`PE`	`PE`	`Compiler-Generator: Interpreter→Compiler`

Interpreter (1), Basisdefinitionen

```
type exp = Int of int
         | Var of string
         | App of string * exp
         | Add of exp * exp
    ...  | Ifz of exp * exp * exp

type def = Declaration of string * string * exp
type prog = Program of def list * exp

exception Undefined
let env0 = fun x -> raise Undefined
let fenv0 = env0
let ext env x v = fun y -> if x=y then v else env y
```

Interpreter (2), Interpretationsfunktion

```
let rec eval e env fenv =
match e with
        Int i -> i
      | Var s -> env s
      | App (s,e2) -> (fenv s)(eval e2 env fenv)
      | Add (e1,e2) -> (eval e1 env fenv)+(eval e2 env fenv)
 ...  | Ifz (e1,e2,e3) -> if (eval e1 env fenv)=0
                          then (eval e2 env fenv)
                          else (eval e3 env fenv)
let rec prgeval p env fenv =
    match p with
       Program ([],e) -> eval e env fenv
    |  Program (Declaration (s1,s2,e1)::tl,e) ->
          let rec f x = eval e1 (ext env s2 x) (ext fenv s1 f)
          in prgeval (Program(tl,e)) env (ext fenv s1 f)
```

Interpreter (3), Staging (a)

```
let rec eval2 e env fenv =
   match e with
         Int i        -> .< i >.
       | Var s        -> env s
       | App (s,e2)   -> .<  .~(fenv s)
                             .~(eval2 e2 env fenv)  >.
       | Add (e1,e2) -> .<    .~(eval2 e1 env fenv)
                            + .~(eval2 e2 env fenv) >.
 ...   | Ifz (e1,e2,e3) -> .< if   .~(eval2 e1 env fenv) = 0
                              then .~(eval2 e2 env fenv)
                              else .~(eval2 e3 env fenv) >.
```

Interpreter (3), Staging (b)

```
let rec prgeval2 p env fenv =
    match p with
       Program ([],e) -> eval2 e env fenv
     | Program (Declaration (s1,s2,e1)::tl,e) ->
          .< let rec f x = .~(eval2 e1 (ext env  s2 .< x >.)
                                       (ext fenv s1 .< f >.))
             in .~(prgeval2 (Program(tl,e))
                            env
                            (ext fenv s1 .< f >.)) >.
```

Bsp.: Fibonacci-Funktion

in Ocaml:

```
let rec fib x =
  if x=0 then 0 else
    if (x-1)=0 then 1 else
      (fib (x-1)) + (fib (x-2))
```

in der Sprache prog:

```
let termFib = Program
      ([Declaration
         ("f","x",
           Ifz(Var "x", Int 0,
           Ifz(Sub(Var "x",Int 1), Int 1,
            Add(App ("f", Sub(Var "x",Int 1)),
                App ("f", Sub(Var "x",Int 2))))))],
       App ("f", Var "input"))
```

Programmteil zur Zeitmessung

```
let number = 34;;
let _ = Trx.init_times ();;
let ocaml = Trx.timenew "Ocaml"
            (fun () -> fib number);;
printf "Ocaml: %d \n" ocaml;;
let inorm = Trx.timenew "Interpreted normal"
    (fun () -> prgeval termFib (ext env0 ("input") number) fenv0);;
printf "Interpreted normal: %d \n" inorm;;
let code = .< fun n ->
        .~(prgeval2 termFib (ext env0 ("input") .<n>.) fenv0) >.;;
let codeFun = Trx.timenew "Partial Eval." (fun () -> code);;
let exec =  Trx.timenew "Execution"
            (fun () -> (.!codeFun) number);;
printf "Execution: %d \n" exec;;
let _ = Trx.print_times ();
```

Ausgaben mit Zeitmessungen

```
Ocaml: 5702887
Interpreted normal: 5702887
Execution: 5702887
__ Ocaml ______________________ 1x avg= 2.466638E+03 msec
__ Interpreted normal _________ 1x avg= 4.391097E+04 msec
__ Partial Eval. _________ 8388608x avg= 1.536825E-04 msec
__ Execution __________________ 1x avg= 2.486646E+03 msec
```

Vergleich von geschriebenem und erzeugtem Code

Ocaml direkt:

```
let rec fib x =
  if x=0 then 0 else
    if (x-1)=0 then 1 else
      (fib (x-1)) + (fib (x-2))
```

Ergebnis der partiellen Auswertung:

```
.<fun n_1 ->
 let rec f_2 =
  fun x_3 ->
   if (x_3 = 0) then 0
   else if ((x_3 - 1) = 0) then 1
   else ((f_2 (x_3 - 1)) + (f_2 (x_3 - 2))) in
 (f_2 n_1)>.
```

Interaktive Codegenerierung zur Laufzeit

Beispiel: Schleife bestehend aus

- Codegenerierung (`code`) in Abhängigkeit von Laufzeitwert (`r`)
- Ausführung des Codes: lies Wert und gib Wert (`r`) zurück

```
#let id x = x;;
val id : 'a -> 'a = <fun>
# let code q = .< let x = Scanf.scanf "%d \n" id
                  in let y = x*q in Printf.printf "\n%d \n" y;
                  flush stdout; y >.;;
val code : int -> ('a, int) code = <fun>
# let r = ref 2;;
val r : int ref = contents = 2
# let p =  for i = 0 to 6 do r := .!(code (!r)) done;;
```

Multi-Staging

- Code erzeugt selbst wieder Code
- nicht mehrstufig: `.< fun x -> .~(f .<x>.) >.`
  das Codestück `.<x>.` wird durch `f` verwendet und durch `.~` eingesetzt

Beispiel für drei Stufen:

```
# let code3 = .< fun x -> .< fun y -> .< fun z -> x+y+z >. >. >.;;
val code3 : ('a, int -> ('b, int -> ('c, int -> int) code) code) code
 = .<fun x_1 -> .<fun y_2 -> .<fun z_3 -> ((x_1 + y_2) + z_3)>.>.>.
# let code2 = (.!code3) 4;;
val code2 : ('a, int -> ('b, int -> int) code) code
 = .<fun y_2_1 -> .<fun z_3_2 -> ((4 + y_2_1) + z_3_2)>.>.
# let code1 = (.!code2) 7;;
val code1 : ('a, int -> int) code
 = .<fun z_3_2_1 -> ((4 + 7) + z_3_2_1)>.
```


Template Haskell

- Haskell-Erweiterung für Metaprogrammierung zur Compile-Zeit
- Ziel: Unterstützung algorithmischer Programmkonstruktion
- Implementierbare Features
  - Makroexpansion
  - Benutzer-programmierte Optimierungen (Inlining)
  - Polytypische Funktionen

Vereinfachung durch Metaprogrammierung

ohne Template-Haskell:

```
"Das Teil "++ teil ++ " kostet "++ (show preis) ++ " Euro\n"
```

mit Template-Haskell (analog zu `sprintf` in C):

```
$(sprintf "Das Teil %s kostet %n Euro\n") teil preis
```

(speziell) in diesem Beispiel

- Unterschied zu C: `sprintf` ist vom Benutzer definiert!
- Ergebnis der `sprintf`-Funktion ist ein Datentyp der Metasprache
- `$` macht daraus eine Funktion mit dem Typ: `String -> Int -> String`
- Anzahl der `%`-Vorkommen bestimmt Anzahl der Argumente
- Symbol hinter `%` bestimmt Typ eines Arguments

Metaprogrammier-Konstrukte

- (Quasi-)Quotierung (ähnlich zu Brackets in MetaOCaml)
  Bsp.: `[| 1+2 |]`, der Ausdruck `1+2` wird zum Codeteil
  - Name *Quotierung* stammt von LISP
  - hier: Compiler unterstützt Einsetzen, Typprüfung, Variablenumbenennung
- Splicing (ähnlich zu Escape in MetaOCaml)
  Bsp.: `$(pow 5) x`
  - `pow` muss in einem importierten Modul deklariert sein
  - `pow 5` erzeugt den Code `\x -> x*x*x*x*x`
  - dieser Code wird durch `$` zu einer Haskell-Funktion
  - Typprüfung erfolgt erst danach

Ersatz für Dependent Types

```
module Types where
import Language.Haskell.THSyntax
myexpr :: Bool -> Q Exp            -- Q: quotation monad
myexpr True  = [|  5 + 6  |]
myexpr False = [| "hello" |]


module Main where
import Types
foo :: (Int,String)
foo = ($(myexpr True), $(myexpr False))
main = print foo
```

Aufruf des Compilers

```
> ghc -fglasgow-exts --make Main.hs -ddump-splices
...
Main.hs:8:9:
    Main.hs:8:9-19: Splicing expression
        myexpr GHC.Base.True ======> (5 `GHC.Num.+` 6)
    In the definition of `foo':
        foo = ($[splice](myexpr True), $[splice](myexpr False))
Main.hs:8:26:
    Main.hs:8:26-37: Splicing expression
        myexpr GHC.Base.False ======> "hello"
    In the definition of `foo':
        foo = ($[splice](myexpr True), $[splice](myexpr False))
Linking ...
> ./a.out
(11,"hello")
```

Rekursion im Metaprogramm (1)

```
module Power where
import Language.Haskell.TH
pow :: Int -> Q Exp
pow 0       = [| \x -> 1 |]
pow n | n>0 = [| \x -> x * $(pow (n-1)) x |]


module Main where
import Power
main = let pow5 = $(pow 5)           -- $ : splice
       in print $ map pow5 [0..3]    -- $ : apply to
```

Rekursion im Metaprogramm (2)

Übersetzungsprozeß:

1. Typprüfung und Codeerzeugung für das `Power`-Modul
2. Splicing im `Main`-Modul mit Ausführen der Rekursion in `pow`
   - automatische Umbenennung lokaler Variablen
   - keine saubere Termination des Compilers garantiert
3. Typprüfung und Codeerzeugung für das `Main`-Modul

Splice für (pow 5)

```
  pow 5
======>
  \ x[a2ui] -> (x[a2ui] `GHC.Num.*` (
      \ x[a2uk] -> (x[a2uk] `GHC.Num.*` (
          \ x[a2um] -> (x[a2um] `GHC.Num.*` (
              \ x[a2uo] -> (x[a2uo] `GHC.Num.*` (
                  \ x[a2uq] -> (x[a2uq] `GHC.Num.*` (
                      \ x[a2us] -> 1
                    x[a2uq]))
                x[a2uo]))
            x[a2um]))
        x[a2uk]))
    x[a2ui]))
```

besser: nur eine Lambda-Abstraktion, Kette von `GHC.Num.*` $\Rightarrow$ Übungsaufgabe

printf-Implementierung (1)

Scannen des Format-Strings:

```
module Printf where
import Language.Haskell.TH

data Format = D | S | L String

scanner :: String -> [Format]
scanner [] = []
scanner ('%':'n':xs) = D : scanner xs
scanner ('%':'s':xs) = S : scanner xs
scanner cs = let (xs,ys) = span (/= '%') cs in L xs : scanner ys
```

printf-Implementierung (2)

Generieren des Haskell-Programmtextes:

```
gen :: [Format] -> Q Exp  -> (Q Exp -> Q Exp) -> Q Exp
gen []          x f = f x
gen (D   : xs) x f = [| \n -> $(gen xs [| $x ++ show n |] f) |]
gen (S   : xs) x f = [| \s -> $(gen xs [| $x ++ s |] f) |]
gen (L s : xs) x f = gen xs [| $x ++ s |] f

sprintf, printf :: String -> Q Exp
sprintf s = gen (scanner s) [| "" |] id
printf  s = gen (scanner s) [| "" |] (\x -> [| putStr $x |])

fprintf :: String -> String -> Q Exp
fprintf fname s = gen (scanner s) [| "" |]
                      (\x -> [| writeFile fname $x |])
```

printf-Verwendung

```
module Main where

import Complex
import Printf

main = do
        let teil  = $(sprintf "[Schraube %n]") 6
            preis = 0.2
        $(printf "Das Teil %s kostet %n Euro\n") teil preis
        let (x,y,z) = (1::Int,2::Float,3::Complex Double)
        $(fprintf "tupel" "Tupel (%n,%n,%n) gefunden\n") x y z
```

Selektion aus Tupeln (1)

```
module Main where
import Sel
main = do
        let x = (11,12,"foo",14,"Hello ",16)
            y = (21,"world",23,24)
        print ($(sel 5 6) x ++ $(sel 2 4) y)
        print ($(sel 3 6) x ++ show ($(sel 4 4) y))
```

Ausgabe:

```
"Hello world"
"foo24"
```

Selektion aus Tupeln (2)

```
module Main where
import Sel
main = do
        let x = (11,12,"foo",14,"Hello ",16)
            y = (21,"world",23,24)
        print ($(sel 5 6) x ++ $(sel 2 4) y)
        print ($(sel 3 6) x ++ show ($(sel 4 4) y))
```

Splices:

```
sel 5 6 ======> \(a1, a2, a3, a4, a5, a6) -> a5
sel 2 4 ======> \(a1, a2, a3, a4) -> a2
sel 3 6 ======> \(a1, a2, a3, a4, a5, a6) -> a3
sel 4 4 ======> \(a1, a2, a3, a4) -> a4
```

Selektion aus Tupeln (3), Implementierung

```
sel :: Int -> Int -> Q Exp
sel i n = do
            let as  = [ mkName ("a" ++ show j) | j<-[1..n] ]
                pat = TupP (map VarP as)
            return (LamE [pat] (VarE (as!!(i-1))))
```

- `mkName` erzeugt Namen aus String, hier keine Umbenennung!
- `VarP`: verwende Namen als Patternvariable
- `TupP`: verwende Liste von Pattern als Tupelpattern
- `VarE`: verwende Namen als Ausdruck
- `LamE`: erzeuge Lambda-Abstraktion (Liste von Argumenten) (Rumpf)

Erzeugung frischer Namen mit der Q-Monade

vorige Folie (Metaprogrammierer bestimmt interne Namen selbst):

```
let as  = [ mkName ("a" ++ show j) | j<-[1..n] ]
```

Ergebnis:

```
\(a1, a2, a3, a4, a5, a6) -> a5
```

hier kein Problem, weil es nur lokale Namen gibt

---

wenn automatische Generierung frischer Namen erwünscht ist:

```
as <- sequence [ newName ("a" ++ show j) | j<-[1..n] ]
```

Ergebnis:

```
\ (a1[a2Cw], a2[a2Cy], a3[a2CA],
      a4[a2CC], a5[a2CE], a6[a2CG]) -> a5[a2CE]
```

Anzeige des Syntaxbaums eines Ausdrucks

```
printAST :: Q Exp -> IO ()
printAST qexp = do
                  ast <- runQ qexp
                  print ast
```

```
*Main> printAST [| \ (x,y,z) -> (x,z) |]
LamE [TupP [VarP x_0,VarP y_1,VarP z_2]] (TupE [VarE x_0,VarE z_2])

*Main> printAST [| (\ x -> div x 2) 6 |]
AppE (LamE [VarP x_0] (AppE (AppE (VarE GHC.Real.div) (VarE x_0))
                            (LitE (IntegerL 2))))
     (LitE (IntegerL 6))
```

```
printAST [| let fac n = case n of { 0       -> 1;
                                    m | m>0 -> m * fac (m-1) }
           in fac 3 |]

LetE
  [FunD fac_4 [Clause [VarP n_5] (NormalB (CaseE (VarE n_5)
       [Match (LitP (IntegerL 0)) (NormalB (LitE (IntegerL 1))) [],
        Match (VarP m_6) (GuardedB [
                (NormalG (InfixE (Just (VarE m_6))
                                 (VarE GHC.Base.>)
                                 (Just (LitE (IntegerL 0)))),
                 InfixE (Just (VarE m_6))
                        (VarE GHC.Num.*)
                        (Just (AppE (VarE fac_4)
                                    (InfixE (Just (VarE m_6))
                                            (VarE GHC.Num.-)
                                            (Just (LitE (IntegerL 1))))))])
                 []])) []]]
  (AppE (VarE fac_4) (LitE (IntegerL 3)))
```

Reification: Zugriff auf Typinformation

```
module Main where
import Language.Haskell.TH
import Language.Haskell.TH.Syntax(lift)

inc :: Num a => a->a
inc = (+1)

queryType = $(do
               typInfo <- reify 'inc
               let s = pprint typInfo
               lift s)

*Main> queryType
"Main.inc :: forall a_0 . GHC.Num.Num a_0 => a_0 -> a_0"
```

Generalisierte `zip`-Funktion (1)

in Haskell vordefiniert:

```
zipWith  :: (a -> b -> c) -> [a] -> [b]  -> [c]
zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
```

Ziel: in Abhängigkeit eines Parameters die passende `zipWith`-Funktion erzeugen.

Generalisierte zip-Funktion (2), Modul Zip (a)

```
genPE :: String -> Int -> Q ([Pat],[Exp])
genPE s n = do
              let ns = [ mkName (s++(show i)) | i<-[1..n]]
              ps <- mapM varP ns
              es <- mapM varE ns
              return (ps,es)

zipN :: Int -> Q Exp
zipN n = [| let zp = $(mkZip n [| zp |]) in zp |]

apps :: [Exp] -> Q Exp
apps xs = return $ foldl1 AppE xs
```

Generalisierte zip-Funktion (3), Modul Zip (b)

```
mkZip :: Int -> Q Exp -> Q Exp
mkZip n qName =
  do
   name <- qName
   (pXs,eXs)   <- genPE "x"  n
   (pYs,eYs)   <- genPE "y"  n
   (pXSs,eXSs) <- genPE "xs" n
   m1 <- do let pcons x xs = return $ InfixP x '(:) xs
            pat  <- tupP (zipWith pcons pXs pXSs)
            body <- normalB [| $(return (TupE eXs)) : $(apps(name : eXSs)) |]
            return $ Match pat body []
   m2 <- do body <- normalB [| [] |]
            pat <- tupP (take n (repeat wildP))
            return $ Match pat body []
   return $ LamE pYs (CaseE (TupE eYs) [m1,m2])
```

Generalisierte zip-Funktion (4), Modul Main

```
module Main where
import Char
import Zip
main = do
        let as = ['a'..'d']
            bs = map toUpper as
            cs = map ord as
        print $ $(zipN 3) as bs cs
        let ds = [1..]
        print $ $(zipN 4) ds as bs cs


> a.out
[('a','A',97),('b','B',98),('c','C',99),('d','D',100)]
[(1,'a','A',97),(2,'b','B',98),(3,'c','C',99),(4,'d','D',100)]
```

Generalisierte zip-Funktion (5), Splice

```
zipN 3
======>
 let
  zp'0 = \ y1 y2 y3
           -> case (y1, y2, y3) of
                (GHC.Base.: x1 xs1, GHC.Base.: x2 xs2, GHC.Base.: x3 xs3)
                  -> ((x1, x2, x3) `GHC.Base.:` (zp'0 xs1 xs2 xs3))
                (_, _, _) -> GHC.Base.[]
```

Fazit

- MetaOCaml
  - Funktionalität: Just-in-Time Compilation
  - Anwendungsgebiete:
    - partielle Auswertung nach Bedarf
    - Optimierungen zur Laufzeit
  - Problem: Typsystem beschränkt Programmkonstruktion
- Template Haskell
  - Funktionalität: komfortabler Präprozessor
  - Anwendungsgebiete:
    - Benutzer-definierte Erweiterung um generische Konstrukte
    - Programmanalyse und Optimierungen zur Compile-Zeit
  - Probleme:
    - Compilerlauf nach Eingabe des Objektprogramms nötig
    - saubere Termination des Metaprogramms nicht gewährleistet