

Template Haskell

Prof. Dr. David Sabel

LFE Theoretische Informatik



- Was ist Meta-Programmierung?
- Was ist Template Haskell?
- Wesentliche Konstrukte zur Meta-Programmierung mit Template Haskell
- Generisches Programmieren
- Reifikation und Typklassenerzeugung
- Quasi-Quotes und eingebettete domänenspezifische Sprachen

Meta-Programmierung

- Meta-Programmierung: Statt Quelltext-Programmen schreibt man Programme, die Quelltext-Programme erzeugen
- Zur Compilezeit werden die Meta-Programme ausgeführt und damit die Quelltexte erzeugt
- Template Haskell dient zur Meta-Programmierung von Haskell-Programmcode
- Bekanntes Beispiel für Meta-Programmierung: C-Präprozessor

Meta-Programmierung

- Meta-Programmierung: Statt Quelltext-Programmen schreibt man Programme, die Quelltext-Programme erzeugen
- Zur Compilezeit werden die Meta-Programme ausgeführt und damit die Quelltexte erzeugt
- Template Haskell dient zur Meta-Programmierung von Haskell-Programmcode
- Bekanntes Beispiel für Meta-Programmierung: C-Präprozessor

Beispiel: C-Präprozessor

```
#define VERSION 2
...
#if VERSION >= 3
    print "NEUESTE VERSION"
#else
    print "ALTE VERSION"
#endif
```

Abhängig davon, welche Versionsnummer definiert wird, wird der Programmcode erzeugt.

- Auch im GHC kann man den C Präprozessor verwenden
- Compilieren mit Flag `-cpp`
- Gebräuchliche Direktiven:
 - Einfügen von Quelltext (`#include`),
 - Ersetzen von Makros (`#define`),
 - Bedingte Übersetzung (`#if`).

Haskell und der C Präprozessor: Beispiel

```
main =  
  do  
    #ifdef NEWVERSION  
      print "Neue Version"  
    #else  
      print "Alte Version"  
    #endif  
    print "Fertig!"
```

- `#ifdef` fragt, ob ein Makro definiert ist.

Haskell und der C Präprozessor: Beispiel

```
main =
  do
#ifdef NEWVERSION
  print "Neue Version"
#else
  print "Alte Version"
#endif
  print "Fertig!"
```

- `#ifdef` fragt, ob ein Makro definiert ist.
- Da Makro nicht definiert:

```
*> stack exec -- ghc -cpp TestCPP.hs
*> ./TestCPP
"Alte Version"
"Fertig!"
```

Haskell und der C Präprozessor: Beispiel

```
main =
  do
#ifdef NEWVERSION
  print "Neue Version"
#else
  print "Alte Version"
#endif
  print "Fertig!"
```

- `#ifdef` fragt, ob ein Makro definiert ist.
- Da Makro nicht definiert:

```
*> stack exec -- ghc -cpp TestCPP.hs
*> ./TestCPP
"Alte Version"
"Fertig!"
```
- Mit definiertem Symbol (beachte hier: mit der Compiler-Option `-D`):

```
*> stack exec -- ghc -cpp -DNEWVERSION TestCPP.hs
*> ./TestCPP
"Neue Version"
"Fertig!"
```

Template Haskell (1)

- CPP ist eine sehr einfache Methode für Meta-Programmierung
- **Template Haskell** bietet viel mehr:
 - Es wird Haskell verwendet, um Haskell-Code zu erzeugen
 - Aus einem Meta-Programm können viele verschiedene Haskell-Quellcode-Programme erzeugt werden
 - Benötigt **algorithmische Beschreibung**, wie die Haskell-Programme erzeugt werden.
 - Meta-Programm implementiert dann den Algorithmus

Template Haskell (2)

Wichtige Anwendungen:

- Automatisches Erzeugen von Boilerplate Code
(Codefragmente, die an vielen Stellen benötigt in gleicher Form benötigt werden)
- Automatisches Erzeugen von Typklasseninstanzen (deriving...)
- Eingebettete domänenspezifische Sprachen:
Einbettung von syntaktisch anderem Code (z.B. HTML) in den Haskell-Code

Wir sehen noch (kleine) Beispiele für all das!

Template Haskell (3)

Verwendung von Template Haskell:

- Pragma `TemplateHaskell` am Quelltextanfang, d.h.

```
{-# LANGUAGE TemplateHaskell #-}
```

- Oder mit Flag z.B. im GHCi:

```
*> :set -XTemplateHaskell
```

- Außerdem: Bibliotheken importieren

```
import Language.Haskell.TH
```

Manchmal zusätzlich

```
import Language.Haskell.TH.Syntax
```

```
import Language.Haskell.TH.Quote
```

Template Haskell (3)

Verwendung von Template Haskell:

- `Pragma TemplateHaskell` am Quelltextanfang, d.h.

```
{-# LANGUAGE TemplateHaskell #-}
```

- Oder mit Flag z.B. im GHCi:

```
*> :set -XTemplateHaskell
```

- Außerdem: Bibliotheken importieren

```
import Language.Haskell.TH
```

Manchmal zusätzlich

```
import Language.Haskell.TH.Syntax
```

```
import Language.Haskell.TH.Quote
```

Sprechweisen:

- **Meta-Programm**: Durch Verwendung von Template Haskell erzeugtes Programm, das zur **Compilezeit** ausgeführt wird und dabei Objekt-Programme erzeugt
- **Objekt-Programm**: Normales Haskell-Programm evtl. durch Meta-Programm erzeugt

Template Haskell als Codegenerator

Beispiele und wie es funktioniert

Beispiel: curry und curryN

- `curry` nimmt Funktion auf Paaren und erstellt curried Funktion (Funktion, welche die Argumente nacheinander erhält):

Beispiel: curry und curryN

- `curry` nimmt Funktion auf Paaren und erstellt curried Funktion (Funktion, welche die Argumente nacheinander erhält):

```
curry :: ((a,b) -> c) -> a -> b -> c
```

```
curry f x y = f (x,y)
```

Beispiel: curry und curryN

- `curry` nimmt Funktion auf Paaren und erstellt curried Funktion (Funktion, welche die Argumente nacheinander erhält):

```
curry :: ((a,b) -> c) -> a -> b -> c
```

```
curry f x y = f (x,y)
```

- `curry3` nimmt Funktion auf Tripeln und ...

```
curry3 :: ((a,b,c) -> d) -> a -> b -> c -> d
```

```
curry3 f x y z = f (x,y,z)
```

Beispiel: curry und curryN

- `curry` nimmt Funktion auf Paaren und erstellt curried Funktion (Funktion, welche die Argumente nacheinander erhält):

```
curry :: ((a,b) -> c) -> a -> b -> c
```

```
curry f x y = f (x,y)
```

- `curry3` nimmt Funktion auf Tripeln und ...

```
curry3 :: ((a,b,c) -> d) -> a -> b -> c -> d
```

```
curry3 f x y z = f (x,y,z)
```

- `curry4` nimmt Funktion auf 4-Tupeln und ...

```
curry4 :: ((a,b,c,d) -> e) -> a -> b -> c -> d -> e
```

```
curry4 f w x y z = f (w,x,y,z)
```

Beispiel: curry und curryN

- `curry` nimmt Funktion auf Paaren und erstellt curried Funktion (Funktion, welche die Argumente nacheinander erhält):

```
curry :: ((a,b) -> c) -> a -> b -> c
```

```
curry f x y = f (x,y)
```

- `curry3` nimmt Funktion auf Tripeln und ...

```
curry3 :: ((a,b,c) -> d) -> a -> b -> c -> d
```

```
curry3 f x y z = f (x,y,z)
```

- `curry4` nimmt Funktion auf 4-Tupeln und ...

```
curry4 :: ((a,b,c,d) -> e) -> a -> b -> c -> d -> e
```

```
curry4 f w x y z = f (w,x,y,z)
```

- `curryN` nimmt Funktion auf n-Tupeln und ...

Beispiel: curry und curryN

- `curry` nimmt Funktion auf Paaren und erstellt curried Funktion (Funktion, welche die Argumente nacheinander erhält):

```
curry :: ((a,b) -> c) -> a -> b -> c
```

```
curry f x y = f (x,y)
```

- `curry3` nimmt Funktion auf Tripeln und ...

```
curry3 :: ((a,b,c) -> d) -> a -> b -> c -> d
```

```
curry3 f x y z = f (x,y,z)
```

- `curry4` nimmt Funktion auf 4-Tupeln und ...

```
curry4 :: ((a,b,c,d) -> e) -> a -> b -> c -> d -> e
```

```
curry4 f w x y z = f (w,x,y,z)
```

- `curryN` nimmt Funktion auf n-Tupeln und ...

Nicht in Haskell definierbar (Welchen Typ soll die Funktion haben?)

Beispiel: curry und curryN

- `curry` nimmt Funktion auf Paaren und erstellt curried Funktion (Funktion, welche die Argumente nacheinander erhält):

```
curry :: ((a,b) -> c) -> a -> b -> c
```

```
curry f x y = f (x,y)
```

- `curry3` nimmt Funktion auf Tripeln und ...

```
curry3 :: ((a,b,c) -> d) -> a -> b -> c -> d
```

```
curry3 f x y z = f (x,y,z)
```

- `curry4` nimmt Funktion auf 4-Tupeln und ...

```
curry4 :: ((a,b,c,d) -> e) -> a -> b -> c -> d -> e
```

```
curry4 f w x y z = f (w,x,y,z)
```

- `curryN` nimmt Funktion auf n-Tupeln und ...

Nicht in Haskell definierbar (Welchen Typ soll die Funktion haben?)

- `curryN` nimmt `n` und Funktion auf n-Tupeln und ...

Nicht in Haskell definierbar (Welchen Typ soll die Funktion haben?)

Beispiel: curry und curryN

- `curry` nimmt Funktion auf Paaren und erstellt curried Funktion (Funktion, welche die Argumente nacheinander erhält):

```
curry :: ((a,b) -> c) -> a -> b -> c
```

```
curry f x y = f (x,y)
```

- `curry3` nimmt Funktion auf Tripeln und ...

```
curry3 :: ((a,b,c) -> d) -> a -> b -> c -> d
```

```
curry3 f x y z = f (x,y,z)
```

- `curry4` nimmt Funktion auf 4-Tupeln und ...

```
curry4 :: ((a,b,c,d) -> e) -> a -> b -> c -> d -> e
```

```
curry4 f w x y z = f (w,x,y,z)
```

- `curryN` nimmt Funktion auf n-Tupeln und ...

Nicht in Haskell definierbar (Welchen Typ soll die Funktion haben?)

- `curryN` nimmt `n` und Funktion auf n-Tupeln und ...

Nicht in Haskell definierbar (Welchen Typ soll die Funktion haben?)

- Ohne Template Haskell: Schreibe Boilerplate-Code, d.h. definiere alle Funktionen `curry1`, `curry2`, `curry3` usw. per Hand

Beispiel: curry und curryN

Idee: Schreibe ein Meta-Programm, das die `curryX`-Funktionen für alle benötigten `X` automatisch generiert

Mit Template Haskell: `curryN i` erzeugt die `i`. curry-Funktion:

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Monad
import Language.Haskell.TH
curryN :: Int -> Q Exp -- Erzeuge abstrakten Syntaxbaum (Exp) in Q-Monade
curryN n = do
  f <- newName "f" -- Erzeuge neuen Namen für f
  xs <- replicateM n (newName "x") -- Erzeuge neue Namen für x1 ... xn
  let args = map VarP (f:xs) -- Erzeuge Pattern f x1 ... xn
      ntup = TupE (map (Just . VarE) xs) -- Erzeuge Tupel (x1,...,xn)
  return $ LamE args (AppE (VarE f) ntup) -- Erzeuge \f x1 ... xn -> f (x1,...xn)
```

- `curryN n` erzeugt im Grunde `\f x1 ... xn -> f (x1,...,xn)`, aber:
Nicht als ausführbares Haskell-Programm, sondern als Syntaxbaum (vom Typ `Exp`)
- Erzeugung in der Quotation-Monade (kurz `Q`)
- `Q-Monade`:
 - dient der Codeerzeugung für Template Haskell.
 - kümmert sich um alle Seiteneffekte der Code-Generierung
 - z.B. Generierung frischer Namen mit

```
newName :: String -> Q Name
```

`Q-Monade` sichert zu, dass die Namen wirklich frisch sind.

Was wurde erzeugt?

- Mit

```
runQ :: Quasi m => Q a -> m a
```

kann man Q-Monade ausführen und Syntaxbaum erzeugen.

- Instanzen von `Quasi` sind `Q` und `IO` wobei die `IO`-Instanz im Wesentlichen zum Ausdrucken des Ausdrucks (und damit dem Debugging) verwendet wird.

Was wurde erzeugt?

- Mit

```
runQ :: Quasi m => Q a -> m a
```

kann man Q-Monade ausführen und Syntaxbaum erzeugen.

- Instanzen von `Quasi` sind `Q` und `IO` wobei die `IO`-Instanz im Wesentlichen zum Ausdrucken des Ausdrucks (und damit dem Debugging) verwendet wird.

Z.B ergibt

```
*> runQ (curryN 2)
```

```
LamE [VarP f_0,VarP x_1,VarP x_2] (AppE (VarE f_0) (TupE [Just (VarE x_1),Just (VarE
```

```
*Main> runQ (curryN 4)
```

```
LamE [VarP f_3,VarP x_4,VarP x_5,VarP x_6,VarP x_7]
```

```
(AppE (VarE f_3) (TupE [Just(VarE x_4),Just(VarE x_5),Just(VarE x_6),Just(VarE
```

Abstrakte Syntaxbäume für Haskell-Code

Definiert im Modul `Language.Haskell.TH`. Datentypen u.a.

- `Dec` für Deklarationen
- `Clause` für Klauseln
- `Exp` für Ausdrücke,
- `Pat` für Patterns
- `Lit` für Literale (wie Zahlkonstanten usw.)
- `Type` für Typen

Abstrakte Syntaxbäume für Haskell-Code (2)

Ausschnitt:

```
data Dec = FunD Name [Clause]
         | ValD Pat Body [Dec]
         | DataD Cxt Name [TyVarBndr] (Maybe Kind) [Con] [DerivClause]
         | Newtyped Cxt Name [TyVarBndr] (Maybe Kind) Con [DerivClause]
         | ...
```

```
data Clause = Clause [Pat] Body [Dec]
```

```
data Body = NormalB Exp | GuardedB [(Guard,Exp)]
```

```
data Exp = VarE Name
         | LitE Lit
         | ConE Name
         | ParensE Exp
         | LamE [Pat] Exp
         | AppE Exp Exp
         | CaseE Exp [Match]
         | ...
```

Abstrakte Syntaxbäume für Haskell-Code (3)

```
data Pat = VarP Name
         | LitP Lit
         | ConP Name [Pat]
         | ParensP Pat
         | WildP
         | TupP [Pat]
         | List [Pat]
         | ...
```

```
data Lit = IntegerL Integer | CharL Char | StringL String | ...
```

```
data Type = VarT Name | ConT Name | ArrowT | TupleT Int | ...
```

Details: Siehe Bibliothek. Der gesamte Sprachumfang von Haskell ist abgedeckt!

Vordefiniert in der Bibliothek:

```
type ExpQ  = Q Exp
type DecsQ = Q [Dec]
type TypeQ = Q Type
type PatQ  = Q Pat
```

Splicen (Spleißen)

- runQ führt Meta-Programme aus, aber erzeugt **keine Objekt-Programme**, sondern den Abstrakten Syntaxbaum für Objekt-Programme

Splicen (Spleißen)

- runQ führt Meta-Programme aus, aber erzeugt **keine Objekt-Programme**, sondern den Abstrakten Syntaxbaum für Objekt-Programme
- Erzeugen von Objekt-Programmen: Das geht mit dem Splice-Operator \$ durch:
\$(*Meta-Programm*)

Splicen (Spleißen)

- `runQ` führt Meta-Programme aus, aber erzeugt **keine Objekt-Programme**, sondern den Abstrakten Syntaxbaum für Objekt-Programme
- Erzeugen von Objekt-Programmen: Das geht mit dem Splice-Operator `$` durch:
$$\$(\textit{Meta-Programm})$$
- $\$(\textit{mprg})$
 - wertet das Meta-Programm *mprg* in der Q-Monade aus
 - und ersetzt $\$(\textit{mprg})$ durch das Objekt-Programm.

Splicen (Spleißen)

- runQ führt Meta-Programme aus, aber erzeugt **keine Objekt-Programme**, sondern den Abstrakten Syntaxbaum für Objekt-Programme
- Erzeugen von Objekt-Programmen: Das geht mit dem Splice-Operator \$ durch:
 \$(Meta-Programm)
- \$(mprg)
 - wertet das Meta-Programm *mprg* in der Q-Monade aus
 - und ersetzt \$(mprg) durch das Objekt-Programm.
 - **Zur Compilezeit!**

Splicen (Spleißen)

- runQ führt Meta-Programme aus, aber erzeugt **keine Objekt-Programme**, sondern den Abstrakten Syntaxbaum für Objekt-Programme
- Erzeugen von Objekt-Programmen: Das geht mit dem Splice-Operator \$ durch:
 $\$(Meta-Programm)$
- $\$(mprg)$
 - wertet das Meta-Programm *mprg* in der Q-Monade aus
 - und ersetzt $\$(mprg)$ durch das Objekt-Programm.
 - **Zur Compilezeit!**

Beispiel:

$\$(curryN\ 3)$ erzeugt die Abstraktion

$\backslash f\ x1\ x2\ x3 \rightarrow f\ (x1, x2, x3)$

als Haskell-Programm

Splicen: Vorführung

```
*Main> :set -XTemplateHaskell
*Main> let fst3 (a,b,c) = a in $(curryN 3) fst3 1 2 3
1
```

Splicen: Vorführung

```
*Main> :set -XTemplateHaskell
*Main> let fst3 (a,b,c) = a in $(curryN 3) fst3 1 2 3
1
```

Mit Flag: `-ddump-splices` kann man die erzeugten Objekt-Programme anzeigen:

```
*Main> :set -ddump-splices
```

Splicen: Vorführung

```
*Main> :set -XTemplateHaskell
*Main> let fst3 (a,b,c) = a in $(curryN 3) fst3 1 2 3
1
```

Mit Flag: `-ddump-splices` kann man die erzeugten Objekt-Programme anzeigen:

```
*Main> :set -ddump-splices
*Main> let fst3 (a,b,c) = a in $(curryN 3) fst3 1 2 3
```

Splicen: Vorführung

```
*Main> :set -XTemplateHaskell
*Main> let fst3 (a,b,c) = a in $(curryN 3) fst3 1 2 3
1
```

Mit Flag: `-ddump-splices` kann man die erzeugten Objekt-Programme anzeigen:

```
*Main> :set -ddump-splices
*Main> let fst3 (a,b,c) = a in $(curryN 3) fst3 1 2 3
<interactive>:4:27-34: Splicing expression
  curryN 3
  =====>
  \ f_a59H x_a59I x_a59J x_a59K -> f_a59H (x_a59I, x_a59J, x_a59K)
1
```

Splicen: Nicht überall

- Im Grunde kann der Splice-Operator `$` fast überall verwendet werden
- Aber: Typsicherheit muss hergestellt sein
- D.h. in `$(Meta-Programm)` muss vor der Ausführung (zur Compilezeit!) das *Meta-Programm* korrekt getypt sein.
- Daher: Separate Compilierung
- Kurz: `$(Meta-Programm)` darf nicht im selben Modul wie das *Meta-Programm* stehen

Splicen: Nicht überall: Demo

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Monad
import Language.Haskell.TH

curryN :: Int -> Q Exp
curryN n = do
  f <- newName "f"
  xs <- replicateM n (newName "x")
  let args = map VarP (f:xs)
      ntup = TupE (map (Just . VarE) xs)
  return $ LamE args (AppE (VarE f) ntup)

main = print $(curryN 3) fst3 1 2 3
fst3 (a,b,c) = a
```

Compilieren ergibt:

error:

- GHC stage restriction:
'curryN' is used in a top-level splice,
quasi-quote, or annotation,
and must be imported, not defined locally
- In the untyped splice: \$(curryN 3)

```
|
13 | main = print $(curryN 3) fst3 1 2 3
```

Richtige Verwendung: Beispiel: Modul mit den ersten 10 curry-Funktionen

Internes Modul:

```
{-# LANGUAGE TemplateHaskell #-}
module GenCurry where
  ...
-- Erzeugt Deklarationen curryi = \f x1 ... xi -> f (x1,...,xi) für i = 1,...,n
generateCurries :: Int -> DecsQ
generateCurries n = forM [1..n] mkCurryDec
  where mkCurryDec ith = do
        cury <- curryN ith
        -- Beachte: mkName :: String -> Name erzeugt festen Namen!
        let name = mkName $ "curry" ++ show ith
            return $ FunD name [Clause [] (NormalB cury) []]
```

Schnittstellen-Modul:

```
{-# LANGUAGE TemplateHaskell #-}
module Curry where
import GenCurry
$(generateCurries 10)
```

Inspizieren des erstellten Moduls

```
*Curry> :browse Curry
curry1 :: (t1 -> t2) -> t1 -> t2
curry2 :: ((a, b) -> t) -> a -> b -> t
curry3 :: ((a, b, c) -> t) -> a -> b -> c -> t
curry4 :: ((a, b, c, d) -> t) -> a -> b -> c -> d -> t
curry5 :: ((a, b, c, d, e) -> t) -> a -> b -> c -> d -> e -> t
curry6 :: ((a, b, c, d, e, f) -> t) -> a -> b -> c -> d -> e -> f -> t
curry7 :: ((a, b, c, d, e, f, g) -> t) -> a -> b -> c -> d -> e -> f -> g -> t
curry8 :: ((a, b, c, d, e, f, g, h) -> t)
         -> a -> b -> c -> d -> e -> f -> g -> h -> t
curry9 :: ((a, b, c, d, e, f, g, h, i) -> t)
         -> a -> b -> c -> d -> e -> f -> g -> h -> i -> t
curry10 :: ((a, b, c, d, e, f, g, h, i, j) -> t)
          -> a -> b -> c -> d -> e -> f -> g -> h -> i -> j -> t
```

- Splice-Operator zum Einsetzen des erzeugten Codes
- Repräsentation der Objekt-Programme durch algebraischen Datentypen
- Meta-Programme werden zur Compilezeit ausgeführt in Q-Monade
Bisher verwendet für frische Namen, weitere Features: Zustand und Reifikation (später)

Problem: Erzeugen der Syntaxbäume ist so nicht sehr komfortabel!

Abhilfen (werden jetzt erläutert)!

- Funktionen zur Konstruktion statt Konstruktoren direkt
- Quotation-Klammern

Funktionen zur Syntaxerzeugung

- Die TH-Bibliothek bietet Funktionen zur Syntaxerzeugung.
- Sie korrespondieren direkt zu den Konstruktoren der algebraischen Datentypen `Exp`, `Pat`, `Dec` und `Type`
- Sie verstecken die monadische Erzeugung etwas und wirken wie Kombinatoren

`generateCurries`-Programm mit den Funktionen zur Syntaxerzeugung:

```
generateCurries :: Int -> Q [Dec]
generateCurries n = forM [1..n] mkCurryDec
  where mkCurryDec ith = funD name [clause [] (normalB (curryN ith)) []]
        where name = mName $ "curry" ++ show ith
```

Funktionen zur Syntaxerzeugung

Der Unterschied zwischen den Funktionen und den Konstruktoren ist ihr Typ:

```
FunD    :: Name -> [Clause] -> Dec
Clause  :: [Pat] -> Body -> Clause
NormalB :: Exp -> Body
```

```
funD    :: Name -> [Q Clause] -> Q Dec
clause  :: [Q Pat] -> Q Body -> Q Clause
normalB :: Q Exp -> Q Body
```

- Funktionen sind bereits in der Q-Monade angesiedelt
- Ersparen explizites Ver-, Aus- und Umpacken in der Monade

Quotation-Klammern

- Erzeuge Abstrakte Syntaxbäume aus dem Quellcode
- Setze den Quellcode dafür in **Oxford**-Klammern: `[| Code |]`

Beispiel: Identitätsfunktion

Erzeugung mit Funktionen:

```
generateId :: Q Exp
generateId = do
  x <- newName "x"
  lamE [varP x] (varE x)
```

Erzeugter AST:

```
*Main> runQ generateId
LamE [VarP x_0] (VarE x_0)
```

Erzeugung mit Quotation-Klammern:

```
generateId' :: Q Exp
generateId' = [| \x -> x |]
```

Erzeugter AST:

```
*Main> runQ generateId'
LamE [VarP x_1] (VarE x_1)
```

Quotation-Klammern (2)

- Quotation-Klammern umschließen regulären Haskell-Code
- Liefern Aktion in der Q-Monade, die AST erzeugt
- Verschiedene Quotation-Klammern je nach Ergebnis-Typ:
 - [e] ...] für Haskell-Ausdrücke,
 - [p] ...] für Patterns,
 - [d] ...] für Deklarationen und
 - [t] ...] für Typen.
- Die Schreibweise [| ... |] ist eine Abkürzung für [e] ...]].

Quotation-Klammern (3)

Beispiele:

```
*Main> runQ [| \x -> x |]  
LamE [VarP x_1] (VarE x_1)
```

```
*Main> runQ [| \x y z-> (x y z) |]  
LamE [VarP x_2,VarP y_3,VarP z_4] (AppE (AppE (VarE x_2) (VarE y_3)) (VarE z_4))
```

```
*Main> runQ [| case y of { [] -> True; _:_ -> False } |]  
CaseE  
  (UnboundVarE y)  
  [Match (ConP GHC.Types.[] []) (NormalB (ConE GHC.Types.True)) [],  
   Match (InfixP WildP GHC.Types.: WildP) (NormalB (ConE GHC.Types.False)) []]
```

Splicen und Quotation-Klammern

- Quotation-Klammern $[| \dots |] :: \text{Objekt-Code} \rightarrow \text{ExpQ}$
- Splice-Operator $\$: \text{ExpQ} \rightarrow \text{Objekt-Code}$
- Sie sind dual zueinander: $\$([| e |]) = e$ für alle Ausdrücke e
- Mischen und Verschachteln ist auch erlaubt: Z.B.
`runQ [| \x -> $(litE (IntegerL 1)) |]`

Spezial-Quotes

Spezial-Quotes zum Zugriff auf den Namen von Identifiers:

- `' Identifier` liefert Namen von Funktion, Variable, Konstruktor als Name
- `' ' Identifier` liefert Namen von Typkonstruktor als Name

Beispiele:

```
Main*> let generateId = [|\x -> x|]
```

```
Main*> (VarE 'generateId)
```

```
VarE Ghci5.generateId
```

```
*Main> (ConE 'True)::Exp
```

```
ConE GHC.Types.True
```

```
*Main> (AppE (AppE (ConE '(:)) (ConE 'True)) (ConE '[])) ::Exp
```

```
AppE (AppE (ConE GHC.Types.::) (ConE GHC.Types.True)) (ConE GHC.Types.[])
```

```
*Main> (AppE (VarE 'map) (VarE 'even))
```

```
AppE (VarE GHC.Base.map) (VarE GHC.Real.even)
```

Spezial-Quotes ((2))

Beispiele mit Typen:

```
*Main> (ConT ''Bool)  
ConT GHC.Types.Bool
```

```
*Main> AppT (ConT ''[]) (ConT ''Bool)  
AppT (ConT GHC.Types.[]) (ConT GHC.Types.Bool)
```

Beispiele: Splicen und Quoten

```
*Main> :set -XTemplateHaskell
*Main> let f = [| \x -> 1 + x |]
*Main> runQ f
LamE [VarP x_0] (InfixE (Just (LitE (IntegerL 1))) (VarE GHC.Num.+)) (Just (VarE x_0)))
*Main> $(f) 6
7
*Main> let g = [| \x -> 1 + $(f) x|]
*Main> runQ g
LamE [VarP x_1] (InfixE (Just (LitE (IntegerL 1))) (VarE GHC.Num.+))
      (Just (AppE (LamE [VarP x_2] (InfixE (Just (LitE (IntegerL 1)))
      (VarE GHC.Num.+)) (Just (VarE x_2)))) (VarE x_1))))
*Main> $f 3
4
*Main> $(g) 6
8
>$f $ $g 10
13
*Main> $([| $f $ $g $ $f $( [| $(f) 3 * ($g) 3 |]) |])
24
```

Beispiel: map-Funktionen erzeugen

`genMapN :: Int -> Q Dec`, erzeugt generisch `map`-Funktionen:

- `$(genMapN 1)` erzeugt die übliche `map`-Funktion auf Listen vom Typ $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$
- `$(genMapN 2)` erzeugt eine binäre `map`-Funktion vom Typ $(a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$
- `$(genMapN 3)` erzeugt eine ternäre `map`-Funktion vom Typ $(a \rightarrow b \rightarrow c \rightarrow d) \rightarrow [a] \rightarrow [b] \rightarrow [c] \rightarrow [d]$
- usw.

Beispiel: map-Funktionen erzeugen (2)

```
genMapN :: Int -> Q Dec
genMapN n | n >= 1    = funD name [c11, c12]
           | otherwise = fail "genMapN: argument n may not be <= 0."
where
  name = mkName $ "map" ++ show n
  c11  = do f <- newName "f"
         xs <- replicateM n (newName "x")
         ys <- replicateM n (newName "ys")
         let argPatts = varP f : consPatts
             consPatts = [ [p| $(varP x) : $(varP ys) |]
                          | (x,ys) <- xs `zip` ys ]
             apply     = foldl (\ g x -> [| $g $(varE x) |])
             first     = apply (varE f) xs
             rest      = apply (varE name) (f:ys)
         clause argPatts (normalB [| $first : $rest |]) []
  c12  = clause (replicate (n+1) wildP) (normalB (conE '[])) []
```

Z.B. erzeugt \$(genMapN 3):

```
map3 f (x1:ys1) (x2:ys2) (x3:ys3) = f x1 x2 x3 : map3 f ys1 ys2 ys3
map3 _ _ _ _ _ = []
```

Demo mit -ddump-splices

```
[1 of 2] Compiling GenMap          ( GenMap.hs, interpreted )
[2 of 2] Compiling Map              ( Map.hs, interpreted )
Map.hs:4:3-29: Splicing declarations
  (\ x_a7EN -> [x_a7EN]) <$> (genMapN 3)
=====>
  map3
    f_a7Fq
    (x_a7Fr : ys_a7Fu)
    (x_a7Fs : ys_a7Fv)
    (x_a7Ft : ys_a7Fw)
  = (((f_a7Fq x_a7Fr) x_a7Fs) x_a7Ft
      : ((map3 f_a7Fq) ys_a7Fu) ys_a7Fv) ys_a7Fw)
  map3 _ _ _ _ = []
Ok, two modules loaded.
```

- In Haskell sind die Bindungsbereiche statisch:
 - Binder werden so verwendet, wie sie im Quelltext stehen
 - Der innerste Binder zählt
- Das gilt auch für Template Haskell
- Ausnahme: Mit `mName`-erzeugte Namen dürfen eingefangen werden (dynamisches Scoping)

Beispiele: Bindungsbereiche

Statisches Scoping:

```
x :: Int
x = 42
```

```
static :: Q Exp
static = [| x |]
```

```
plus42 :: Int -> Int
plus42 x = $static + x
```

```
-- an der $static Stelle wird das
-- zu x=42 passende x eingesetzt
```

Dynamisches Scoping bei mkName:

```
x :: Int
x = 42
```

```
dynamic :: Q Exp
dynamic = VarE (mkName "x")
```

```
times2 :: Int -> Int
times2 x = $dynamic + x
```

```
-- an der $dynamic Stelle wird ein x ein-
-- gesetzt, das durch times2 x gebunden wird
```

Reification

Programme inspizieren, Typklasseninstanzen erzeugen

- Reifikation erlaubt es Meta-Programmen zur Compilezeit Information über andere Programmteile abzufragen
- Fragen sind z.B.:
 - Welchen Typ hat diese Variable?
 - Was sind die Datenkonstruktoren dieses Datentyps?
- Template Haskell bietet dafür: `reify :: Name -> Q Info`
- Paradebeispiel dazu: Erzeugung von Typklasseninstanzen

Beispiel: Functor-Instanzen automatisch erzeugen

Beispiele:

```
data Result e a = Err e | Ok a
data List      a = Nil | Cons a (List a)
data Tree     a = Leaf a | Node (Tree a) a (Tree a)
```

Functor-Instanzen (händisch):

```
instance Functor (Result e) where
  fmap f (Err e) = Err e
  fmap f (Ok a)  = Ok (f a)

instance Functor List where
  fmap f Nil      = Nil
  fmap f (Cons a xs) = Cons (f a) (fmap f xs)

instance Functor Tree where
  fmap f (Leaf a)      = Leaf (f a)
  fmap f (Node l a r) = Node (fmap f l) (f a) (fmap f r)
```

Beispiel: Functor-Instanzen automatisch erzeugen (2)

Functor-Instanzen für beliebigen Datentyp $T\ a$ (algorithmisch!):

- Implementiere `fmap :: (a -> b) -> T a -> T b`
- In Werten vom Typ $T\ a$:
 - ersetze alle Werte vom Typ a durch Werte vom Typ b
→ geht nur durch Anwenden der Funktion f
 - ersetze Werte vom Typ $T\ a$ durch Werte vom Typ $T\ b$
→ durch Anwenden von `fmap f`

Diesen Algorithmus kann man als Meta-Programm implementieren (nächste Folie)

Beispiel: Functor-Instanzen automatisch erzeugen (3)

```
module GenFunctor where
...
data Deriving = Deriving { tyCon :: Name, tyVar :: Name }
deriveFunctor :: Name -> Q [Dec]
deriveFunctor ty= do (TyConI tyCon) <- reify ty
    (tyConName, tyVars, cs) <- case tyCon of
        DataD _ nm tyVars _ cs _ -> return (nm, tyVars, cs)
        NewtypeD _ nm tyVars _ c _ -> return (nm, tyVars, [c])
        _ -> fail "deriveFunctor: tyCon may not be a type synonym."
    let (KindedTV tyVar Start) = last tyVars
        instanceType           = contT ''Functor `appT`
            (foldl apply (contT tyConName) (init tyVars))
    putQ $ Deriving tyConName tyVar
    sequence [instanceD (return []) instanceType [genFmap cs]]
where
    apply t (PlainTV name)      = appT t (varT name)
    apply t (KindedTV name _) = appT t (varT name)
```

Beispiel: Functor-Instanzen automatisch erzeugen (4)

```
genFmap :: [Con] -> Q Dec
genFmap cs = do
    -- erzeuge eine Definitionszeile pro Konstruktor in cs
    funD 'fmap (map genFmapClause cs)

genFmapClause :: Con -> Q Clause
genFmapClause c@(NormalC name fieldTypes) = do
    f      <- newName "f"
    fieldNames <- replicateM (length fieldTypes) (newName "x")
    let pats = varP f:[conP name (map varP fieldNames)]
        body = normalB $ appse $
                conE name : map (newField f) (zip fieldNames fieldTypes)
    clause pats body []
```

Beispiel: Functor-Instanzen automatisch erzeugen (5)

```
newField :: Name -> (Name, StrictType) -> Q Exp
newField f (x, (_, fieldType))
  = do Just (Deriving typeCon typeVar) <- getQ
      case fieldType of
        -- Fall: x ist vom Typ a, dann ersetze durch (f x)
        VarT typeVar'
          | typeVar' == typeVar -> [| $(varE f) $(varE x) |]
        -- Fall: x ist vom Typ T a, dann ersetze durch (fmap f x =
        (AppT ty (VarT typeVar'))
          | leftmost ty == (ConT typeCon) && typeVar' == typeVar ->
            [| fmap $(varE f) $(varE x) |]
        -- Ansonsten: behalte x
        _ -> [| $(varE x) |]

leftmost :: Type -> Type
leftmost (AppT ty1 _) = leftmost ty1
leftmost ty           = ty
```

Beispielaufruf

`$(deriveFunctor ''Tree)` erzeugt:

```
stack exec -- ghci -ddump-splices FunEx.hs
GHCi, version 8.8.3: https://www.haskell.org/ghc/  :? for help
[1 of 2] Compiling GenFunctor      ( GenFunctor.hs, interpreted )
[2 of 2] Compiling FunEx          ( FunEx.hs, interpreted )
FunEx.hs:6:3-22: Splicing declarations
  deriveFunctor ''Tree
=====>
instance Functor Tree where
  fmap f_a6le (Leaf x_a6lf) = Leaf (f_a6le x_a6lf)
  fmap f_a6lg (Node x_a6lh x_a6li x_a6lj)
    = ((Node ((fmap f_a6lg) x_a6lh)) (f_a6lg x_a6li))
      ((fmap f_a6lg) x_a6lj)
```

Quasi-Quotes

Quelltext jeder Sprache einbinden

- Oxford-Klammern sind Quasi-Quotes, die Haskell-Syntax lesen können
- Z.B. `[e| ... |]` für `Exp` usw.
- Mit der Erweiterung `QuasiQuoter` kann man das gleiche Prinzip für die eigenen Datentypen verwenden.
- Vorteil: Man kann damit beliebigen Quelltext einer anderen Sprache direkt einbinden
- Beispiel: shakespeare-Bibliothek liefert einen Quasi-Quoter `[hamlet| ...]` für HTML-artigen Code
- TH-Tutorial im Haskell-Wiki: `[regex| ...]` für reguläre Ausdrücke
- Wir: Minimalbeispiel

Minimalbeispiel (künstlich!)

Sprache: $\{0, 1\}$

Datentyp für Syntaxbäume:

```
data Simple = Zero | One
```

Quasi-Quoter für `simple` sollen ermöglichen:

- Benutzer kann `[simple| 0 |]` oder `[simple| 1 |]` eingeben
- Erzeugt direkt den Syntaxbaum vom Typ `Maybe Simple` (`Nothing` für Parse-Fehler)
- Beachte: Beispiel ist sehr künstlich und minimal, üblicherweise würde man statt 0 und 1 komplizierte Quelltexte einer komplizierten Sprache erwarten.

Minimalbeispiel (2)

```
{-# LANGUAGE QuasiQuotes, TemplateHaskell #-}
module Simple where
import Language.Haskell.TH.Quote
import Language.Haskell.TH
import Language.Haskell.TH.Syntax
import Data.Char(isSpace)
```

```
data Simple = Zero | One deriving Show
```

Standard-Parser für das Beispiel

```
parseSimple :: String -> Maybe Simple
parseSimple xs =
  let removeBlanks = [a | a <- xs, not $ isSpace a]
  in case removeBlanks of  "0" -> Just Zero
                           "1" -> Just One
                           _  -> Nothing
```

Minimalbeispiel (3)

Definition des QuasiQuoters `[simple|...|]` benötigt:

```
simple :: QuasiQuoter
simple = QuasiQuoter { quoteExp  = compile
                      , quotePat  = notHandled "patterns"
                      , quoteType = notHandled "types"
                      , quoteDec  = notHandled "declarations" }
  where notHandled things = error $ things ++ " not handled by simple quasiquoter."
```

QuasiQuoter ist definiert als:

```
data QuasiQuoter = QuasiQuoter {
    -- Quasi-quoter for
    quoteExp  :: String -> Q Exp, -- expressions, invoked by e.g. lhs = $[q|...]
    quotePat  :: String -> Q Pat, -- patterns, invoked by e.g. $[q|...] = rhs
    quoteType :: String -> Q Type, -- types, invoked by e.g. f :: $[q|...]
    quoteDec  :: String -> Q [Dec] -- declarations
}
```

Minimalbeispiel (4)

```
compile :: String -> Q Exp
compile = lift . parseSimple

-- class Lift t where
-- lift :: t -> Q Exp
-- liftTyped :: Quote m => t -> Code m t

instance Lift Simple where
  lift Zero = conE 'Zero
  lift One  = conE 'One
  liftTyped = unsafeTExpCoerce . lift -- liftTyped muss impementiert werden
                                         -- (in neuer Versionen)
```

Ausprobieren

```
$> stack exec -- ghci Simple.hs
GHCi, version 8.8.3: https://www.haskell.org/ghc/ :? for help
[1 of 1] Compiling Simple          ( Simple.hs, interpreted )
Ok, one module loaded.
*Simple> :set -XTemplateHaskell -XQuasiQuotes
*Simple> [simple| 0 |]
Just Zero
*Simple> [simple| 1 |]
Just One
*Simple> [simple| 2 |]
Nothing
*Simple> [simple| 0      |]
Just Zero
```

Quasi-Quotes: Fazit

- Das gezeigte Vorgehen geht im Grunde mit jeder Sprache
- Parser und lift-Funktion werden i.a. komplizierter
- Sehr gut für: Eingebettete domänenspezifische Sprachen

Zusammenfassung Template Haskell

Template Haskell kann dazu verwendet werden:

- Typ-sicher stark generischen Code schreiben, z.B. Funktion `curryN :: Int -> Q Exp` zum Currying beliebiger Funktionen auf Tupeln
`$(curryN 3) :: ((a, b, c) -> d) -> a -> b -> c -> d`
- Implementierung von generischen Mechanismen wie `deriving ...`
- Zur Manipulationen von anderen Programmiersprachen mit Haskell, insbesondere DSLs z.B. Yesod-Webserver: HTML, CSS, JavaScript und JSON
- Erzeugte Splices kann man ansehen mit `ghc`-Optionen `-ddump-splices` oder `-dth-dec-file`
- Template Haskell seit 2002 von Tim Sheard und Simon Peyton Jones; QuasiQuotes seit 2007 von Geoffrey Mainland.