Template Haskell

Seminararbeit von Peter Findeisen

basierend auf

Tim Sheard, Simon Peyton Jones:
"Template Meta-programming for Haskell"

im Rahmen des Seminars
"Fortgeschrittene Techniken der funktionalen Programmierung"

betreut von Frank Huch

Inhaltsverzeichnis

1 Einführung

1.1 Über dieses Dokument

Dieses Dokument ist eine Seminararbeit zum Thema Template Haskell im Rahmen des Seminars "Fortgeschrittene Techniken der funktionalen Programmierung" im Wintersemester 2005/06. Sie basiert auf der Arbeit *Template meta-programming for Haskell* von Tim Sheard und Simon Peyton Jones, ACM SIGPLAN Haskell Workshop 02, ACM Press, Oct 2002.
Ziel ist es, dem Leser einen Überblick über die Fähigkeiten und den Aufbau von Template Haskell zu geben, wobei besonderer Wert auf die praktische Anwendung gelegt wird. Die Arbeit ist folgendermaßen aufgebaut: Zuerst eine Einführung in Metaprogrammierung allgemein und der groben Idee, der Template Haskell zugrunde liegt. Dann werden anhand von Beispielen das praktische "Metaprogrammieren" und die Basisoperatoren von Template Haskell vorgestellt. Darauf folgend werden dann die inneren Strukturen besprochen.

1.2 Template Haskell und Metaprogrammierung

Template Haskell ist eine Compile-Zeit Metaprogrammierungserweiterung von Haskell, d.h. man wird durch Template Haskell in die Lage versetzt, Teile seines Programms zur Compile-Zeit berechnen zu können, anstatt sie schreiben zu müssen. Ein Beispiel hierfür wäre der `deriving`-Ausdruck im normalem Haskell, wo durch das Anhängen von `...deriving Show` an eine Datentyp-Deklaration dieser Datentyp zur Instanz der Klasse `Show` wird und eine passende `show`-Funktion aus der Deklaration des Typs selbst abgeleitet (berechnet) wird.
Ziel von Template Haslell ist es, dem Programmierer die Möglichkeit zu geben, solche und noch darüber hinausgehende Code-Manipulationen selbst zu entwickeln und nicht nur auf built-in-Funktionen des Compilers angewiesen zu sein.
Zu den Metaprogrammierungsfähigkeiten, die Template Haskell unterstützt (bzw unterstützen soll, wenn es fertig ist) gehört:

- *Conditional compilation* ermöglicht es dem Programmierer sein Pragramm für unterschiedliche Plattformen, Debug-Optionen oder andere Konfigurationen zu kompilieren. Ein Beispiel hierfür wäre die `#IFDEF`-Anweisungen des C-Präprozessors, nur eleganter.

- *Programm Reifikation.* Reifikation ("Vergegenständlichung") heißt hier, dass ein Programm in der Lage ist, Teile seines eigenen Codes zu besichtigen. Z.B. muss `deriving` um eine `show`-Funktion für einen Datentyp ableiten zu können, die Struktur dieses Datentyps kennen.

- *Algorithmische Programm Konstruktion.* Dies bedeutet Teile des Programms nach zur Compile-Zeit bekannten Konstanten zu berechnen, wobei der zur Konstruktion verwendete Algorithmus kürzer ist als der erzeugte Code. Beispiel hierfür wäre `printf`, worauf im folgenden noch ausfürlich eingegangen wird.
- *Optimierungen.* Der Programmierer soll befähigt werden, den Compiler über anwendungsspezifische Optmierungsmöglichkeiten, wie bestimmte algebraische Gesetzmäßigkeiten, zu informieren.

Template Haskell selbst ist in Haskell geschrieben und so sind die Templates bzw Makros in Template Haskell (fast) ganz normale Haskell-Funktionen, nur dass sie zur Compile-Zeit ausgeführt werden und Haskell-Code zurückgeben. Der Code wiederum wird als normaler algebraischer Datentyp dargestellt. Dies ist mit das Besondere von Template Haskell im Gegensatz z.B. zum C-Präprozessor, wo extra eine eigene Sprache (`#if, #define,...`) eingeführt wird. So können alle Vorteile von Haskell wie Bibliotheken und Programmiertechniken übernommen werden.

2 Die Praxis von Template Haskell

Ziel dieses Abschnitts ist es, anhand von Beispielen die grundlegende Idee sowie die praktische Arbeit mit Template Haskell zu vermitteln.

2.1 Die Operatoren `splice` und `quasi-quote`

Da die Funktionen (Templates) in Template Haskell wieder in Haskell geschrieben sind, ergibt sich das Problem, wie man den Code, der zur Compile-Zeit ausgewertet werden soll, von dem übrigem Teil des Programms unterscheidet. Zu diesem Zweck gibt es in Template Haskell zwei neue Operatoren: `splice` (`$`) und `quasi-quote` (`[|..|]`). "`$`" bedeutet, dass der nachfolgende Haskell-Ausdruck zur Compile-Zeit ausgewertet werden soll, `[|..|]` umschließt Code, der nicht zur Compile-Zeit ausgeführt wird, sondern z.B. der Rückgabewert eines Templates ist.

Ein Beispiel:
Angenommen man wollte in Haskell eine Funktion wie `printf` in C definieren, deren Aufruf wie folgt aussehen soll:

```
printf "Error: %s at line %d." msg line
```

In Haskell kann `printf` nicht definiert werden, weil sein Typ vom Wert seines ersten Argumentes abhängt. In Template Haskell kann `printf` aber als Template definiert werden. Der Aufruf sähe dann so aus:

```
$(printf "Error: %s at line %d.") msg line
```

`$` zeigt hier an, dass `(printf..)` zur Compile-Zeit ausgeführt wird. Folgender Code wird dann zurückgegeben:

```
(\ s0 -> n1 ->
    "Error: " ++ s0 ++ " at line " ++ show n1)
```

Von diesem Lambda-Ausdruck kann jetzt der Typ festgestellt werden und er kann auf seine Argumente angewendet werden:

```
prompt> $((printf "Error: %s at line %d.") "bad var" 123
"Error: bad var at line 123"
```

Eine erste Version von `printf` könnte folgendermaßen in Template Haskell definiert sein:

```
import Language.Haskell.TH

data Format = D | S | L String
   deriving Show

printf :: String -> ExpQ
printf string = gen (parse string)

parse :: String -> [Format]
parse string = ...

gen :: [Format] -> ExpQ
gen [D]  = [| \n -> show n |]
gen [S]  = [| \s -> s |]
gen[L s] = lift s
```

Zuerst wird der Format-String mit Hilfe von `parse` in eine übersichtlichere Liste überführt. Dabei steht `D` für %d, `S` für %s und `L String` für alle anderen Teile des Format-Strings. z.B.
`parse "%d is %s"` ergibt `[D, L " is ", S]`
Die Implementierung von `parse` ist hier nicht angegeben, weil sie zum Verständnis von Template Haskell nicht beiträgt. Interessanter ist hier `gen`. `gen` nimmt eine (zunächst einelementige) Liste aus Elementen des Typs Format und konstruiert daraus mit Hilfe von Quasi-quotes den Code, der anstelle des Templates eingefügt wird. Dabei sieht der Code nachher genauso aus, wie er in den Klammern steht. Es gilt dabei sogar folgende Beziehung:

```
$[| e |] = e
```

D.h. $ und Quasi-quotes heben sich gegenseitig auf. Ferner sieht man, dass der Typ dieser Quasi-quotes mit `ExpQ` angegeben ist. `ExpQ` ist ein normaler Haskell-Datentyp zur Darstellung von Haskell-Ausdrücken. Es gibt noch weitere Datentypen für Code, nämlich `DecQ` für Deklarationen (`zip7 = ...`) und `Type` für Typen. Die entsprechenden Quasi-quotes sind `[d|...|]` und `[t|...|]`.
Die Funktion `lift :: String -> ExpQ` "hebt" nur den String auf den Typ `ExpQ` und läßt den String sonst unverändert.

Bisher kann `gen` nur einelementige Format-Listen verarbeiten. Folgende Version kommt mit beliebig langen Listen zurecht:

```
printf :: String -> ExpQ -> ExpQ
printf s = gen (parse s) [| "" |]

gen :: String -> ExpQ -> ExpQ
gen []         acc = acc
gen (D   : xs) acc = [| \n-> $(gen xs [| $acc++show n |]) |]
gen (S   : xs) acc = [| \s-> $(gen xs [| $acc++s |]) |]
gen (L s : xs) acc = [| $acc ++ $(lift s) |]
```

Besonders bemerkenswert ist hier, dass der `$`-Operator auch innerhalb der Quasi-quotes verwendet werden kann. `acc` ist ein Akkumulator-Argument vom Typ `ExpQ`. Folgende Beispielausführung, soll den Programmablauf verdeutlichen, die wichtigsten Schritte sind das Einsetzen der jeweiligen Definitionen und dass sich der `$`-Operator und die Quasi-quotes gegenseitig aufheben:

Beispielausführung

```
    $(printf "%d%s")
    setze Definition von printf ein; führe parse aus
⟹  $(gen [D, S] [| "" |])
    setze D-Fall aus gen ein
⟹  $[| \n -> $(gen [S] [| $[| "" |] ++ show n |]) |]
    Benutze zweimal, dass $ und [|..|] sich gegenseitig aufheben
⟹  \n -> $(gen [S] [| "" ++ show n |])
    "" ++ fällt weg
⟹  \n -> $(gen [S] [| show n |])
    setze S-Fall aus gen ein
⟹  \n -> $[| \s -> $(gen [] [| $[| show n |] ++ s |]))
    Benutze zweimal, dass $ und [|..|] sich gegenseitig aufheben
⟹  \n -> \s -> $(gen [] [| show n ++ s |])
    setze []-Fall aus gen ein
⟹  \n -> \s -> $[| show n ++ s |]
    Benutze, dass $ und [|..|] sich gegenseitig aufheben
⟹  \n -> \s -> show n ++ s
```

Soweit haben wir gesehen, wie man mit `$` und Quasi-quotes arbeitet. Der `$`-Operator erzwingt die Ausführung des nachfolgenden Ausdrucks zur Compile-Zeit und die Quasi-quotes "zitieren" Haskell-Code, d.h. konstruieren aus dem was zwischen den Klammern steht einen Ausdruck des Typs `ExpQ` (bzw. `DecQ` oder `Type`). Leider hat die doch recht komfortable Quasi-quote-Notation ihre Beschränkungen, wie wir im nächsten Abschnitt sehen.

2.2 Funktionen zur Syntax-Konstruktion

Wir wollen jetzt, da wir mit der Quasi-quote Notation vertraut sind, ein weiteres Template betrachten:
In Haskell gibt es nur für 2-elementige Tuple Selektions-Funktionen (`fst` und `snd`), wir möchten daher ein Template `sel`, das uns einen Selektor für das i-te Element eines n-Tuples liefert und sich folgendermaßen anwenden läßt:

```
prompt> $(sel 1 3) (1,2,3)
1
```

Mit Hilfe der Quasi-quote Notation können wir `sel` nicht schreiben, denn um das i-te Element eines Tupels zu selektieren müssen wir Pattern-Matching verwenden, z.B.:

```
case x of (a,b,c) -> a
```

In Quasi-quote Notation müsste man dieses Pattern direkt fest hinschreiben, aber das Pattern und der zugehörige Ausdruck hängen von n bzw. i ab, sind daher, wenn wir den Quasi-quote schreiben wollen, nicht bekannt. Man braucht also eine Möglichkeit, Haskell Syntax direkter als über Quasi-quotes zu konstruieren. Diese Möglichkeit wird in Template Haskell von einer Bibliothek von Funktionen zur Syntax-Konstruktion gegeben. Folgendermaßen lassen sie sich auf unser Beispiel anwenden:

```
sel :: Int -> Int -> ExpQ
sel i n = lamE [pat]  (varE (as !! (i-1)))
     where pat :: PatQ
           pat  = tupP (map varP as)
           as :: [Name]
           as = [mkName ("a"++show i) | i <- [1..n] ]
```

Damit würde `$(sel 1 3)` folgenden Code zurückliefern:

```
(\ (a1,a2,a3) -> a1)
```

Hier eine kurze Beschreibung der verwendeten Funktionen:

- `lamE :: [PatQ] -> ExpQ -> ExpQ` baut einen Lambda-Ausdruck. Das erste Argument von `lamE` vom Typ `[PatQ]` ist die Liste der Argument-Pattern, das zweite vom Typ `ExpQ` ist der Rumpf der Funktion.
  Das "E" in `lamE` besagt, dass der Ergebnistyp `ExpQ` ist. Auch bei anderen Funktionen hat das angehängte "E" diese Bedeutung.

- `mkName :: String -> Name` Bezeichner haben in Template Haskell einen eigenen Typ, `mkName` erzeugt einen solchen aus einem String.

- `varP :: Name -> PatQ` erstellt eine Pattern-Variable mit dem Namen des ersten Argumentes. Das angehängte "P" steht hier wie in anderen Funktionen dafür, dass der Ergebnistyp der Funktion `PatQ` ist.

- `varE :: Name -> ExpQ` erzeugt eine Variable in Ausdrücken

- `tupP :: [PatQ] -> PatQ` erzeugt ein Pattern-Tuple

Diese und viele weitere Funktionen werden von Template Haskell bereitgestellt. Sie sind im Anhang aufgelistet.

Leider ist der Code von `sel` im Vergleich zu der Quasi-quote Notation umfangreicher und unübersichtlicher. Glücklicherweise kann man die beiden Stile mischen, so dass man Teile von `sel` auch in Quasi-quote Notation schreiben kann, was den Code etwas besser lesbar macht. Der erste Versuch hierfür ist folgender:

```
sel :: Int -> Int -> ExpQ
sel  i n = [| \ $pat -> $(varE (as !! (i-1))) |]
     where pat :: PatQ
           pat  = tupP (map varP as)
           as :: [Name]
           as = [mkName ("a"++show i) | i <- [1..n] ]
```

Die derzeitige Implementierung von Template Haskell im GHC[1] scheint bedauerlicherweise das Splicen von Argumenten in Lambda-Abstraktionen innerhalb von Quasi-quotes nicht zu unterstützen, daher verursacht `$pat` eine Fehlermeldung. Deshalb hier eine Version von `sel`, die dieses Problem mit Hilfe eines case-Ausdrucks umgeht:

```
sel :: Int -> Int -> ExpQ
sel i n = [| \x  -> $(caseE [| x |] [alt]) |]
       where alt :: MatchQ
             alt  = match pat rhs []
             pat :: PatQ
             pat  = tupP (map varP as)
             rhs :: BodyQ
             rhs = normalB (varE (as !! (i-1)))
             as :: [Name]
             as = [mkName ("a"++show i) | i <- [1..n] ]
```

Eine kurze Erklärung der neuen Funktionen:

- `caseE :: ExpQ -> [MatchQ] -> ExpQ`: konstruiert eine Fallunterscheidung über dem ersten Argument. Das zweite Argument ist eine Liste der einzelnen Fälle.
- `match :: PatQ -> BodyQ [DecQ]-> MatchQ`: konstruiert einen Fall im case-Ausdruck der Form: `pat -> body where decs`
- `normalB :: ExpQ -> BodyQ`: erzeugt einen Rumpf

Für `sel 1 3` wird dann folgender Code erzeugt:

[1] Einige Hinweise, wie man Template Haskell im GHC benutzen kann, sind im Anhang gegeben

```
\\x_0 -> case x_0 of
           (a1, a2, a3) -> a1
```

Allgemein gilt, dass man wo immer möglich, die Quasi-quote Notation verwenden sollte, weil sie einfacher, netter und sicher ist. Wenn die Quasi-quotes nicht ausreichen, kann aber auf die flexibleren Syntax-Konstruktionsfunktionen zurückgegriffen werden.

2.3 Splicing von Deklarationen

Das Splicen von Deklarationen ist im Prinzip das Gleiche, wie das Splicen von Ausdrücken, das wir bisher betrachtet haben. Es gibt allerdings folgende Unterschiede:

- Die Quasi-quotes sehen anders aus: statt `[|..|] :: ExpQ` wie bei Ausdrücken muss man `[d|..|] :: Q [Dec]` verwenden, d.h. es wird eine *Gruppe* von Deklarationen eingefügt.
- Das Splicen von Deklarationen ist nur auf dem Top-Level erlaubt.

Ein Beispiel:

```
$([d| f x = x + 1; g x = x * 2|])

main = do print (f 3)
          print (g 7)
```

Ausführen von `main` ergibt:

```
prompt>  main
4
14
```

Es können wie im Beispiel Funktionsdefinitionen aber auch beliebige Wert- sowie `instance`-, `class`-, `data`- oder `type`-Deklarationen eingefügt werden.

3 Die Interna von Template Haskell

Ziel dieses Abschnitts ist es, etwas tiefer in die inneren Strukturen von Template Haskell einzudringen und einige von ihnen näher zu untersuchen.

3.1 Überblick über die inneren Strukturen

In Template Haskell gibt es drei Schichten zur Repräsentation von Programmen, von denen wir zwei bereits kennengelernt haben. Nach Abstraktionsgrad geordnet sind es die folgenden:

- Die unterste Schicht haben wir bisher noch nicht kennengelernt, sie besteht aus zwei Teilschichten:
  Ganz unten werden *normale algebraische Haskell-Datentypen* benutzt, um Haskell-Code darzustellen.
  Darüber liegt die *Quotation-Monade*(Q-Monade). Sie dient zur Generierung neuer Namen und spielt eine Rolle bei der Reifikation.
- Die *Syntax-Konstruktionsfunktionen*, wie `varE` oder `tupP`.
- Die *Quasi-quote Notation*. Sie ist die am leichtesten anzuwendene der drei Schichten und kann in die unteren Schichten übersetzt werden.

Im folgenden wollen wir diese Schichten etwas näher beleuchten.

3.2 Code als Datenstruktur

Die grundlegende Idee bei Template Haskell ist es, Haskell-Code zur Compile-Zeit zu generieren und zwar nicht durch eine Form des textuellen Einfügens von Text in die Quelldatei oder eine andere Form von externem Tool, sondern Haskell selbst dafür zu benutzen. Eine Folge davon ist, dass man einen Weg finden muss, Haskell-Code in Haskell zu repräsentieren. Dazu eignen sich normale algebraische Datentypen, z.B. läßt sich dann `(f x)` als `(AppE (VarE "f") (VarE "x")) :: Exp` [2] darstellen, wobei dann `AppE` und `VarE` Datenkonstruktoren des Typs `Exp` sind. Die Datentypen zur Repräsentation von Code sind im Anhang aufgelistet. Die wichtigsten sind `Exp` für Ausdrücke, `Dec` für Deklarationen, `Type` für Typen und `Pat` für Pattern. Sie sind alle so aufgebaut, dass sie die jeweilige Syntax "so wie man sie eintippt" wiederspiegeln sollen.

Der Vorteil dieser Darstellung von Code als algebraische Datenstruktur ist, dass man mit den bewährten Methoden aus Haskell (z.B. `case`, Pattern-Matching) auf Haskell-Code operieren kann, ohne etwas dazulernen zu müssen.
Der Nachteil ist, dass zum einen der Code zur Repräsentation von Code viel umständlicher als der darzustellende Code ist. Zum anderen gibt es keine Unterstützung für semantische Information wie z.B. Scoping oder Typisierung. D.h

[2] `VarE` nimmt als Argument eigentlich keinen String, sondern einen Bezeichner vom Typ `Name`. Der Übersichtlichkeit halber habe ich das hier vereinfacht

zum einen kann man auf dieser Ebene nicht genau festlegen, welche Variable an weche Definition gebunden ist, zum anderen kann man, wie beim Programmieren von Hand, Typfehler einbauen.
Während die Typfehler erst erkannt werden können, wenn der Compiler den Code in das Programm einfügt (spliced), gibt es für das Scoping-Problem Unterstützung in Form der Quotation-Monade.

3.3 Quotation-Monade

Die Quotation-Monade ist (vor allem) ein Hilfsmittel für den Programmierer, um Namenskonflikte zu vermeiden. Hier ein Beispiel:
Zuerst, der Übersichtlichkeit halber, in Quasi-Quote Notation:

```
id1 :: ExpQ ->ExpQ
id1 f = [| \ x -> $f x |]
```

Hier die rein "algebraische" Version:

```
id2 :: Exp -> Exp
id2 f = LamE [VarP (mkName "x")] (AppE f (VarE (mkName "x")))
```

Wenn man jetzt `id2` mit ungünstigen Argumenten aufruft, bekommt man einen Namenskonflikt:

```
prompt> pprint [3](id2 (VarE (mkName "x")))
"\\x -> x x"
```

Der Namenskonflikt ist entstanden, weil die Argumente zufällig genauso heißen wie die Patternvariablen des Lambda-Ausdrucks. Dabei sind die Namen der Patternvariablen beliebig wählbar, man könnte (und sollte) sie in diesem Fall auch umbenennen. Genau diese Umbenennung übernimmt die Quotation-Monade für den Programmierer:

```
id3 :: Exp -> Q Exp
id3  f = do
          x <- newName "x"
          return (LamE [VarP x] (AppE f (VarE x)))
```

Anwenden ergibt:[4]

[3] `pprint` steht für pretty-print
[4] Das kann man so natürlich nicht in die Konsole eintippen, aber wir tun mal so als ob.

```
prompt> id3 (VarE (mkName "x"))
"\\x_0 -> x x_0"
```

Jetzt geschieht das Umbenennen von Variablen automatisch, was eine große Erleichterung beim Schreiben und Benutzen von Templates ist.
Was bei `id3` auffällt ist, dass die Argumente nicht-monadisch sind (sondern vom Typ `Exp`), aber der Rückgabewert schon (`Q Exp`). Will man diesen Wert weiterverarbeiten bekommt man Schwierigkeiten. Deshalb arbeitet man immer gleich vollständig in der Quotation-Monade:

```
id4 :: Q Exp -> Q Exp
id4 qf = do
          f <- qf
          x <- newName "x"
          return (LamE [VarP x] (AppE f (VarE x)))
```

Erst wird "das Monadische" abgestreift und dann wie bei `id3` fortgefahren.

Wenn man nun `ExpQ` und `Q Exp` vergleicht, so klingen beide ziemlich ähnlich. Tatsächlich sind sie sogar gleich, d.h. es gilt:

```
type ExpQ = Q Exp
```

Entsprechendes gilt auch für `DecQ`, `PatQ`, usw.

3.4 Die Syntax-Konstruktionsfunktionen

Wir haben oben gesehen, dass `..Q` ein Synonym für `Q ..` ist. Daraus folgt, dass auch die Syntax-Konstruktionsfunktionen monadisch sind. Genau genommen sind sie die monadischen Gegenstücke zu den Konstruktoren der entsprechenden Datentypen, z.B. für `AppE`:

```
AppE :: Exp  -> Exp  -> Exp
appE :: ExpQ -> ExpQ -> ExpQ
```

`appE` macht dabei das Gleiche wie `id4` oben: Zuerst wird von den Argumenten "das Q abgestreift" und dann der neue Datentyp gebaut:

```
appE :: ExpQ -> ExpQ -> ExpQ
appE x y = do { a <- x; b <- y; return (AppE a b)}
```

Hier ist jetzt eine weitere Version von `id`:

```
id5 :: ExpQ ->ExpQ
id5 f = do
         x <- newName "x"
         lamE [varP x] (appE f (varE x))
```

Mit Hilfe der Syntax-Konstruktionsfunktionen können wir nun darauf verzichten, bei `f <- qf` in `id4` "das Q abzustreifen", weil das von den Syntax-Konstruktionsfunktionen übernommen wird. Nicht verzichten dagegen kann man auf die Generierung eines neuen Namens für `x`, da sonst wieder das alte Problem des Namenkonfliktes auftritt.

3.5 Quasi-quotes

Nicht nur die Syntax-Konstruktionsfunktionen sind vom monadischen Typ, sondern auch die Quasi-quotes in der Quasi-quote Notation. In der Tat ist es so, dass die Ausdrücke in den eckigen Klammern `[|..|]` übersetzt werden in Ausdrücke bestehend aus u.a. `Name`n, Syntax-Konstruktionfunktionen, `newName` und der `do`-Notation der `Q`-Monade. Dabei wird dann auch das automatische Umbenennen von Variablen berücksichtigt, z.B liefert die Quasi-quote Version, `id1`, das gleiche Ergebnis wie die richtigen monadischen:

```
prompt> id1 (varE (mkName "x"))
"\\x_0 -> x x_0"
```

In der Quasi-quote Notation ist auch sichergestellt, dass Variablen lexikalisch gebunden werden, d.h. wenn es vor Expansion irgendeines Templates so scheint, als ob eine Variable an einen Wert gebunden ist, so ist sie es auch danach.

Ein Beispiel:

```
x :: Int
x = 3

g :: Int -> ExpQ
g y = [| x + y |]

t :: a -> Int
t x = $(g [| 4 |])
```

Die Variable `x` in der Definition des Templates `g` ist fest an die erste Definition von `x` gebunden, selbst dann, wenn das Template in eine Umgebung, wie `t`, gespliced wird, in der bereits ein anderes `x` gebunden ist.

4 Reifikation

Reifikation ("Vergegenständlichung") bedeutet hier, dass Templates Teile des Codes, den der Programmierer geschrieben hat oder die von anderen Templates erzeugt wurden, inspizieren können. Z.B. muss für ein `derive`-Template die Definition von dem Typ, der Instanz einer Klasse werden soll, dem Template bekannt sein.

In Template Haskell wird diese Aufgabe von einer Funktion übernommen:

```
reify :: Name -> Q Info
```

`reify` nimmte einen Namen und gibt zu diesem Namen Informationen, die aus der Symbol-Tabelle des Compilers stammen, in Form einer Datenstruktur vom Typ `Q Info` zurück. `Info` ist folgendermaßen definiert:

```
data Info
  = ClassI Dec
  | ClassOpI
    Name -- The class op itself
    Type  -- Type of the class-op (fully polymoprhic)
    Name  -- Name of the parent class
    Fixity

  | TyConI Dec

  | PrimTyConI  -- Ones that can't be expressed with a data type
                -- decl, such as (->), Int#
    Name
    Int         -- Arity
    Bool        -- False => lifted type; True => unlifted

  | DataConI
    Name        -- The data con itself
    Type        -- Type of the constructor (fully polymorphic)
    Name        -- Name of the parent TyCon
    Fixity

  | VarI
    Nam         -- The variable itself
    Type
    (Maybe Dec) -- Nothing for lambda-bound variables, and
```

```
                -- for anything else TH can't figure out
                -- E.g. [| let x = 1 in $(do { d <- reify 'x; .. }) |]
    Fixity

  | TyVarI      -- Scoped type variable
    Name
    Type        -- What it is bound to
```

Man benutzt `reify` nun, indem man den Namen der Struktur, die man untersuchen möchte, herausfindet, und dann z.B. eine `case`-Analyse des `Info`-Wertes durchführt, den `reify` zurückliefert und evtl. weitere `reify`-Aufrufe auf den Ergebnissen der Analyse ausführt, um alle Informationen, die man braucht, zu erlangen.

Ein kurzes Beispiel, das den Namen des Datenkonstruktors vom `Info`-Datensatz ausgibt, den `reify` angewand auf den Namen der Funktion `f` zurückliefert. Falls der Datenkonstruktor anzeigt, dass `f` eine Variable ist (und das ist der Fall), so sollen auch die detaillierteren Informationen ausgegeben werden.

```
import Language.Haskell.TH
import Language.Haskell.TH.Syntax -- for lift

f = (+1)

main = putStr $(do {
          x <- reify 'f;
          case x of
            (ClassI _)         -> [|"ClassI"|]
            (ClassOpI _ _ _ _) -> [|"ClassOpI"|]
            (TyConI _)         -> [|"TyConI"|]
            (PrimTyConI _ _ _) -> [|"PrimTyConI"|]
            (DataConI _ _ _ _) -> [|"DataConI"|]
            (VarI n t d _)     -> [| "VarI\n  Name: " ++ $(lift (show n)) ++
                                     "\n  Type: " ++ $(lift (show t)) ++
                                     "\n  Decl: " ++ $(lift (show d)) ++ "\n"
                                  |]
            (TyVarI _ _)       -> [|"TyConI"|]
             _                 -> [|"Error: unknown data constructor in Info"|]
           })
```

Bemerkungen:

- `'f` ist eine abkürzende Schreibweise, gemeint ist der `Name` von `f`.

- `lift` "hebt" einen String auf den Typ `ExpQ` und läßt ihn ansonsten unverändert.

Ausführen von `main` ergibt:

```
prompt> main
VarI
  Name: Reify.f
  Type: AppT (AppT ArrowT (VarT a_1627399271)) (VarT a_1627399271)
  Decl: Nothing
```

`reify` ist noch nicht ganz fertig implementiert und immer noch eine Baustelle, d.h. einige Features funktionieren noch nicht oder nicht so, wie sie sollten. Deshalb steht in dem Info-Datensatz von `f` oben auch `Nothing` für die Deklaration, anstatt des richtigen Wertes.

5 Zusammenfassung

Template Haskell ist eine Erweiterung von Haskell um die Fähigkeit, zur Compile-Zeit Metaprogrammierung durchführen zu können, d.h. durch sogenannte Templates, die zur Compile-Zeit ausgeführt werden, Code automatisch generieren zu können. Diese Templates sind ebenfalls in Haskell geschrieben, so dass alle Vorzüge von Haskell auch für die Metaprogrammierung genutzt werden können.
Da die Templates zur Generierung von Code in Haskell geschrieben sind, muss der Code, den sie als Ergebnis ihrer Berechnungen zurückgeben, in Haskell als algebraischer Datentyp repräsentiert werden.
Um den Programmierer zu unterstützen wurde die Quotation-Monade eingeführt, die unter anderem die Möglichkeit zur einfachen Generierung neuer Namen bereitstellt, was zur Vermeidung von Namenskonflikten notwendig ist.
Die Syntax-Konstruktionsfunktionen sind die monadischen Gegenstücke zu den algebraischen Datenkonstruktoren und machen das Leben in der Quotation-Monade leichter, wenn auch der Template-Code immer noch sehr "redselig" und unübersichtlich ist.
Der einfachste und sicherste Weg, Code zu konstruieren, führt aber über die Quasiquote Notation.

Reifikation, d.h. die Möglichkeit auf die bereits vorhandenen Definitionen im Programm zugreifen zu können, ist eines der interessantesten Features von Template Haskell und wird durch die Funktion `reify` verwirklicht, die zu einem gegebenen Namen die zugehörige Information in Form einer Datenstruktur zurückgibt.

Template Haskell ist noch in der Entwicklung, es gibt viele offene Designfragen und die Implementierung im GHC ist unvollständig. Trotzdem wird es bereits für einige Zwecke benutzt und ist sicher (oder wird sein) ein mächtiges Werkzeug für den Programmierer.
Nachteil von Template Haskell ist, dass die Quasi-quote Notation allein nicht ausreicht und auf den unteren Schichten von Template Haskell der Code der Templates doch recht unübersichtlich wird, besonders bei Reifikation wird man wahrscheinlich nicht darum herumkommen, viel auf den unteren Schichten zu arbeiten. Debuggen kann da durchaus schwierig werden.
Hoffen wir, dass sich die Entwickler auf diese und die anderen ausstehenden Fragen befriedigende Antworten finden können, wenn sie ihre Arbeit an Template Haskell fortsetzen.

Literatur

[1] *Sheard, T., Peyton-Jones, S.*: Meta-programing for Haskell. ACM SIGPLAN Haskell Workshop 02, ACM Press, Oct 2002

[2] *Sheard, T., Peyton-Jones, S.*: Notes on Template Haskell Version 2, November 7, 2003

[3] http://www.haskell.org/TH - Dort findet man alles rund um Template Haskell, inklusive obiger Paper

A Bibliothek der monadischen Syntax-Operatoren

```
-------------------------------------------------------------
-- Type synonyms
-------------------------------------------------------------
type InfoQ          = Q Info
type PatQ           = Q Pat
type FieldPatQ      = Q FieldPat
type ExpQ           = Q Exp
type DecQ           = Q Dec
type ConQ           = Q Con
type TypeQ          = Q Type
type CxtQ           = Q Cxt
type MatchQ         = Q Match
type ClauseQ        = Q Clause
type BodyQ          = Q Body
type GuardQ         = Q Guard
type StmtQ          = Q Stmt
type RangeQ         = Q Range
type StrictTypeQ    = Q StrictType
type VarStrictTypeQ = Q VarStrictType
type FieldExpQ      = Q FieldExp

-------------------------------------------------
-- Lowercase pattern syntax functions
-------------------------------------------------

intPrimL    :: Integer -> Lit
floatPrimL  :: Rational -> Lit
doublePrimL :: Rational -> Lit
integerL    :: Integer -> Lit
charL       :: Char -> Lit
stringL     :: String -> Lit
rationalL   :: Rational -> Lit

litP :: Lit -> PatQ
varP :: Name -> PatQ
tupP :: [PatQ] -> PatQ
conP :: Name -> [PatQ] -> PatQ
infixP :: PatQ -> Name -> PatQ -> PatQ
tildeP :: PatQ -> PatQ
asP :: Name -> PatQ -> PatQ
wildP :: PatQ
recP :: Name -> [FieldPatQ] -> PatQ
listP :: [PatQ] -> PatQ
sigP :: PatQ -> TypeQ -> PatQ
fieldPat :: Name -> PatQ -> FieldPatQ

-------------------------------------------------------
--     Stmt

bindS :: PatQ -> ExpQ -> StmtQ
letS :: [DecQ] -> StmtQ
noBindS :: ExpQ -> StmtQ
parS :: [[StmtQ]] -> StmtQ

-------------------------------------------------------
--     Range

fromR :: ExpQ -> RangeQ
fromThenR :: ExpQ -> ExpQ -> RangeQ
fromToR :: ExpQ -> ExpQ -> RangeQ
fromThenToR :: ExpQ -> ExpQ -> ExpQ -> RangeQ

-------------------------------------------------------
--     Body

normalB :: ExpQ -> BodyQ
guardedB :: [Q (Guard,Exp)] -> BodyQ

-------------------------------------------------------
--     Guard

normalG :: ExpQ -> GuardQ
normalGE :: ExpQ -> ExpQ -> Q (Guard, Exp)
patG :: [StmtQ] -> GuardQ
patGE :: [StmtQ] -> ExpQ -> Q (Guard, Exp)

-------------------------------------------------------
--     Match and Clause

match :: PatQ -> BodyQ -> [DecQ] -> MatchQ
clause :: [PatQ] -> BodyQ -> [DecQ] -> ClauseQ

-------------------------------------------------------
--     Exp

dyn :: String -> Q Exp
global :: Name -> ExpQ
varE :: Name -> ExpQ
conE :: Name -> ExpQ
litE :: Lit -> Exp
appE :: ExpQ -> ExpQ -> ExpQ
infixE :: Maybe ExpQ -> ExpQ -> Maybe ExpQ -> ExpQ
infixApp :: ExpQ -> ExpQ -> ExpQ -> ExpQ
sectionL :: ExpQ -> ExpQ -> ExpQ
sectionR :: ExpQ -> ExpQ -> ExpQ
lamE :: [PatQ] -> ExpQ -> ExpQ
lam1E :: PatQ -> ExpQ -> ExpQ    -- Single-arg lambda
tupE :: [ExpQ] -> ExpQ
condE :: ExpQ -> ExpQ -> ExpQ -> ExpQ
letE :: [DecQ] -> ExpQ -> ExpQ
caseE :: ExpQ -> [MatchQ] -> ExpQ
doE :: [StmtQ] -> ExpQ
compE :: [StmtQ] -> ExpQ
arithSeqE :: RangeQ -> ExpQ

-- arithSeqE Shortcuts
fromE :: ExpQ -> ExpQ
fromThenE :: ExpQ -> ExpQ -> ExpQ
fromToE :: ExpQ -> ExpQ -> ExpQ
fromThenToE :: ExpQ -> ExpQ -> ExpQ -> ExpQ
-- End arithSeqE shortcuts

listE :: [ExpQ] -> ExpQ
sigE :: ExpQ -> TypeQ -> ExpQ
recConE :: Name -> [Q (Name,Exp)] -> ExpQ
recUpdE :: ExpQ -> [Q (Name,Exp)] -> ExpQ
stringE :: String -> ExpQ
fieldExp :: Name -> ExpQ -> Q (Name, Exp)

-------------------------------------------------------------
--     Dec

valD :: PatQ -> BodyQ -> [DecQ] -> DecQ
funD :: Name -> [ClauseQ] -> DecQ
tySynD :: Name -> [Name] -> TypeQ -> DecQ
dataD :: CxtQ -> Name -> [Name] -> [ConQ] -> [Name] -> DecQ
newtypeD :: CxtQ -> Name -> [Name] -> ConQ -> [Name] -> DecQ
classD :: CxtQ -> Name -> [Name] -> [FunDep] -> [DecQ] -> DecQ
instanceD :: CxtQ -> TypeQ -> [DecQ] -> DecQ
sigD :: Name -> TypeQ -> DecQ
forImpD :: Callconv -> Safety -> String -> Name -> TypeQ -> DecQ
cxt :: [TypeQ] -> CxtQ
normalC :: Name -> [StrictTypeQ] -> ConQ
recC :: Name -> [VarStrictTypeQ] -> ConQ
infixC :: Q (Strict, Type) -> Name -> Q (Strict, Type) -> ConQ
forallC :: [Name] -> CxtQ -> ConQ -> ConQ

-------------------------------------------------------------
-- Type

forallT :: [Name] -> CxtQ -> TypeQ -> TypeQ
varT :: Name -> TypeQ
conT :: Name -> TypeQ
appT :: TypeQ -> TypeQ -> TypeQ
arrowT :: TypeQ
listT :: TypeQ
tupleT :: Int -> TypeQ
isStrict, notStrict :: Q Strict
strictType :: Q Strict -> TypeQ -> StrictTypeQ
varStrictType :: Name -> StrictTypeQ -> VarStrictTypeQ

-------------------------------------------------------------
--     Callconv

cCall, stdCall :: Callconv

-------------------------------------------------------------
--     Safety
```

```
unsafe, safe, threadsafe :: Safety

-------------------------------------------------------------
--     FunDep

funDep :: [Name] -> [Name] -> FunDep
```

```
-------------------------------------------------------------
-- Useful helper functions

combine :: [([(Name, Name)], Pat)] -> ([(Name, Name)], [Pat])
rename :: Pat -> Q ([(Name, Name)], Pat)
genpat :: Pat -> Q ((Name -> ExpQ), Pat)
alpha :: [(Name, Name)] -> Name -> ExpQ
appsE :: [ExpQ] -> ExpQ
simpleMatch :: Pat -> Exp -> Match
```

B Algebraische Datentyp Representation

```
-------------------------------------------------------
--
--The Info returned by reification
--
-------------------------------------------------------

data Info
  = ClassI Dec
  | ClassOpI
    Name -- The class op itself
    Type  -- Type of the class-op (fully polymoprhic)
    Name  -- Name of the parent class
    Fixity

  | TyConI Dec

  | PrimTyConI  -- Ones that can't be expressed with a data type
                -- decl, such as (->), Int#
    Name
    Int         -- Arity
    Bool        -- False => lifted type; True => unlifted

  | DataConI
    Name        -- The data con itself
    Type        -- Type of the constructor (fully polymorphic)
    Name        -- Name of the parent TyCon
    Fixity

  | VarI
    Name        -- The variable itself
    Type
    (Maybe Dec) -- Nothing for lambda-bound variables, and
                -- for anything else TH can't figure out
                -- E.g. [| let x = 1 in $(do { d <- reify 'x; .. }) |]
    Fixity

  | TyVarI      -- Scoped type variable
    Name
    Type        -- What it is bound to

data Fixity       = Fixity Int FixityDirection deriving( Eq )
data FixityDirection = InfixL | InfixR | InfixN   deriving( Eq )

-------------------------------------------------------
--
--The main syntax data types
--
-------------------------------------------------------

data Lit = CharL Char
         | StringL String
         | IntegerL Integer
         | RationalL Rational
         | IntPrimL Integer
         | FloatPrimL Rational
         | DoublePrimL Rational
    deriving( Show, Eq )

data Pat
  = LitP Lit               -- { 5 or 'c' }
  | VarP Name              -- { x }
  | TupP [Pat]             -- { (p1,p2) }
  | ConP Name [Pat]        -- data T1 = C1 t1 t2; {C1 p1 p1} = e
  | InfixP Pat Name Pat    -- foo ({x :+ y}) = e
  | TildeP Pat             -- { ~p }
  | AsP Name Pat           -- { x @ p }
  | WildP                  -- { _ }
  | RecP Name [FieldPat]   -- f (Pt { pointx = x }) = g x
  | ListP [ Pat ]          -- { [1,2,3] }
  | SigP Pat Type          -- p :: t
  deriving( Show, Eq )

type FieldPat = (Name,Pat)

data Match = Match Pat Body [Dec] -- case e of { pat -> body where decs }
    deriving( Show, Eq )
data Clause = Clause [Pat] Body [Dec] -- f { p1 p2 = body where decs }
    deriving( Show, Eq )

data Exp
  = VarE Name            -- { x }
  | ConE Name            -- data T1 = C1 t1 t2; p = {C1} e1 e2
  | LitE Lit             -- { 5 or 'c'}
  | AppE Exp Exp         -- { f x }

  -- {x + y} or {(x+)} or {(+ x)} or {(+)}
  | InfixE (Maybe Exp) Exp (Maybe Exp)


  | LamE [Pat] Exp               -- { \ p1 p2 -> e }
  | TupE [Exp]                   -- { (e1,e2) }
  | CondE Exp Exp Exp            -- { if e1 then e2 else e3 }
  | LetE [Dec] Exp               -- { let x=e1;   y=e2 in e3 }
  | CaseE Exp [Match]            -- { case e of m1; m2 }
  | DoE [Stmt]                   -- { do { p <- e1; e2 }  }
  | CompE [Stmt]                 -- { [ (x,y) | x <- xs, y <- ys ] }
  | ArithSeqE Range              -- { [ 1 ,2 .. 10 ] }
  | ListE [ Exp ]                -- { [1,2,3] }
  | SigE Exp Type                -- e :: t
  | RecConE Name [FieldExp]      -- { T { x = y, z = w } }
  | RecUpdE Exp [FieldExp]       -- { (f x) { z = w } }
  deriving( Show, Eq )

type FieldExp = (Name,Exp)

-- Omitted: implicit parameters

data Body
  = GuardedB [(Guard,Exp)]   -- f p { | e1 = e2 | e3 = e4 } where ds
  | NormalB Exp              -- f p { = e } where ds
  deriving( Show, Eq )

data Guard
  = NormalG Exp
  | PatG [Stmt]
  deriving( Show, Eq )

data Stmt
  = BindS Pat Exp
  | LetS [ Dec ]
  | NoBindS Exp
  | ParS [[Stmt]]
  deriving( Show, Eq )

data Range = FromR Exp | FromThenR Exp Exp
           | FromToR Exp Exp | FromThenToR Exp Exp Exp
           deriving( Show, Eq )

data Dec
  = FunD Name [Clause]          -- { f p1 p2 = b where decs }
```

```
  | ValD Pat Body [Dec]         -- { p = b where decs }
  | DataD Cxt Name [Name]
         [Con] [Name]
-- { data Cxt x => T x = A x | B (T x)
    --       deriving (Z,W)}
  | NewtypeD Cxt Name [Name]
         Con [Name]
 -- { newtype Cxt x => T x = A (B x)
     --       deriving (Z,W)}
  | TySynD Name [Name] Type     -- { type T x = (x,x) }
  | ClassD Cxt Name [Name] [FunDep] [Dec]
-- { class Eq a => Ord a where ds }
  | InstanceD Cxt Type [Dec] -- { instance Show w => Show [w]
                             --       where ds }
  | SigD Name Type           -- { length :: [a] -> Int }
  | ForeignD Foreign
  deriving( Show, Eq )

data FunDep = FunDep [Name] [Name]
  deriving( Show, Eq )

data Foreign = ImportF Callconv Safety String Name Type
             | ExportF Callconv        String Name Type
         deriving( Show, Eq )

data Callconv = CCall | StdCall
          deriving( Show, Eq )
```

```
data Safety = Unsafe | Safe | Threadsafe
        deriving( Show, Eq )

type Cxt = [Type]    -- (Eq a, Ord b)

data Strict = IsStrict | NotStrict
         deriving( Show, Eq )

data Con = NormalC Name [StrictType]
         | RecC Name [VarStrictType]
         | InfixC StrictType Name StrictType
         | ForallC [Name] Cxt Con
         deriving( Show, Eq )

type StrictType = (Strict, Type)
type VarStrictType = (Name, Strict, Type)

data Type = ForallT [Name] Cxt Type  -- forall <vars>. <ctxt> -> <type>
          | VarT Name                -- a
          | ConT Name                -- T
          | TupleT Int               -- (,), (,,), etc.
          | ArrowT                   -- ->
          | ListT                    -- []
          | AppT Type Type           -- T a b
      deriving( Show, Eq )
```

C Template Haskell im GHC benutzen

Es existiert eine Implementierung von Template Haskell für den GHC. Folgendes muss beachtet werden, um sie zu benutzen:

- Die Quelldatei muss `Language.Haskell.TH` importieren
- `ghc` bzw `ghci` müssen mit der Option `-fth` oder `-fglasgow-exts` aufgerufen werden, z.B.
  `> ghc --make -fth Main.hs -o main`
- Eine Funktion darf nur dann zur Compile-Zeit aufgerufen werden, wenn sie aus einem anderen Modul importiert wurde. Es ist also nicht erlaubt, innerhalb eines Splice eine Funktion aufzurufen, die in demselben Modul definiert ist.
- Mit der Option `-ddump-splices` werden expandierte top-level Splices angezeigt.