Funktionale Programmierung

Template Haskell, Web Applikationen mit Yesod

Andreas Abel und Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

18. Juni 2012

Grundlagen

Sprachdesign ist grundsätzlich ein Kompromiss zwischen **Flexibilität** und **Sicherheit**.

Sprachen mit statischem Typ-System bieten einen hohen Grad an Sicherheit ("well-typed programs don't go wrong"), sind aber weniger flexibel als ungetypte Sprachen.

Ein Ansatz um die Flexibilität zu erhöhen ist es typ-basierte Techniken einzuführen, um gleichartige, **generische**, Programme zu definieren.

Arten von Generischen Programmen

Der Begriff “generisch” in Programmiersprachen ist sehr generisch!

Eine Funktion kann über verschiedene Dinge **parametrisiert** sein:

Wert: Argument einer Funktion

Arten von Generischen Programmen

Eine Funktion kann über verschiedene Dinge **parametrisiert** sein:

Typ: Polymorphismus, in 2 Formen:

- Parametrischer Polymorphismus: der gleich Code wird für verschiedene Typen verwendet z.B: length Funktion

$$
\begin{aligned}
&\mathit{length} :: [a] \to \mathit{Int} \\
&\mathit{length}\ [\,] \quad\ = 0 \\
&\mathit{length}\ (\_ : \mathit{xs}) = 1 + (\mathit{length}\ \mathit{xs})
\end{aligned}
$$

- Ad-hoc Polymorphismus (overloading): der Code hängt vom Typ selbst ab; z.B. Addition auf Int, Complex, etc

Arten von Generischen Programmen

Eine Funktion kann über verschiedene Dinge **parametrisiert** sein:

Funktion: damit erhalten wir Funktionen höherer Ordnung; z.B:

$$
\begin{aligned}
&map :: (a \to b) \to [a] \to [b] \\
&map\ f\ [\,] \quad\quad = [\,] \\
&map\ f\ (x : xs) = (f\ x) : (map\ f\ xs)
\end{aligned}
$$

Arten von Generischen Programmen

Eine Funktion kann über verschiedene Dinge **parametrisiert** sein:

Interface: man abstrahiert über eine Menge von typ-basierten Eigenschaften, die für alle Funktionen gelten müssen; z.B: Typ-Context in Haskell Typ-Klassen

```
instance (Eq a) ⇒ Eq [a] where
  []       == []       = True
  (x : xs) == (y : ys) = x == y && xs == ys
  (_xs)    == (_ys)    = False
```

Arten von Generischen Programmen

Eine Funktion kann über verschiedene Dinge **parametrisiert** sein:

Eigenschaft: als Erweiterung von Interfaces, abstrahiert man über generelle Eigenschaften die für die Funktionen gelten müssen; z.B: Monaden-Regeln für Instanzen der Monad Klasse (muss manuell bewiesen werden):

$$
\begin{aligned}
&\mathit{return}\ x \gg\!= k = k\ x \\
&k \gg\!= \mathit{return} \quad = k \\
&(m \gg\!= \lambda x \to k\ x) \gg\!= k' = m \gg\!= \lambda x \to (k\ x \gg\!= k')
\end{aligned}
$$

Arten von Generischen Programmen

Eine Funktion kann über verschiedene Dinge **parametrisiert** sein:

Arten von Generischen Programmen

Eine Funktion kann über verschiedene Dinge **parametrisiert** sein:
Phase: man unterteilt die Compilierung in Phasen, wobei frühe Phasen selbst wieder Programme erzeugen ("meta-programming");
z.B: Verwendung eines Parser Generators.
In Template Haskell kann man Projektionen auf ein Tupel anstatt

$$fst3 = \lambda(x, \_, \_) \to x$$

wie folgt definieren:

$$\begin{aligned}
&sel :: Int \to Int\ ExpQ \\
&sel\ i\ n = lamE\ [pat]\ body \\
&\qquad \textbf{where}\ pat \;\; = tupP\ (map\ varP\ vars) \\
&\qquad\qquad\ body = varE\ (vars\ !!\ (i-1)) \\
&\qquad\qquad\ vars \;\; = [mkName\ (\text{"a"} +\!\!+ show\ j) \mid j \leftarrow [1\,..\,n]] \\
&fst3 = \$(sel\ 1\ 3)
\end{aligned}$$

Template Haskell

Erlaubt Meta-programmierung, welche bei der Kompilierung ausgeführt wird, d.h. Haskell-Code wird mit Haskell verarbeitet.

Zum Umschalten zwischen den Ebenen verwendet man:

Quasi-Quoting mit Oxford-Klammern `[| |]`
: Code innerhalb der Oxford-Klammern wird zu einem Datenobjekt verarbeitet.

Spleissen mit Dollar-ohne-folgendem-Leerzeichen `$( )`
: Ein Datenobjekt, welches Code repräsentiert, wird damit während der Kompilierung wieder zu Code gemacht.

Diese Wechsel der Ebenen dürfen auch verschachtelt werden!

Oxford-Klammer und Q-Monade

Es gibt mehrere Versionen der Oxford-Klammer:

1. `[e| ... |]` für Ausdrücke (expressions); liefert Typ `Q Exp`
2. `[d| ... |]` für Deklarationen; liefert Typ `Q [Decl]`
3. `[t| ... |]` für Typen; liefert Typ `Q Type`
4. `[p| ... |]` für Patterns; liefert Typ `Q Pat`

Weitere Tags für die Oxford-Klammer zum Quasi-Quoten von anderen Sprachen, wie z.B. HTML, JavaScript, CSS, etc. werden wir mit Yesod kennenlernen.

Alles lebt in der Monade `Q`. Diese kümmert sich um alle Seiteneffekte der Code-Generierung, z.B. frische Namen.

Verwendet werden aber meist die vordeklarierten Typ-Synonyme `ExpQ`, `DecsQ`, `TypeQ` und `PatQ`

Template Haskell verwenden

Zur Verwendung von Template Haskell ist die Option

`-XTemplateHaskell`

oder besser das Pragma

`{-# LANGUAGE TemplateHaskell #-}`

notwendig.

Ein wichtiges Modul ist `Language.Haskell.TH`, welches u.a. bereitstellt:

```
runQ    :: Quasi m => Q a -> m a
runIO   :: IO a -> Q a
mkName  :: String -> Name
reify   :: Name -> Q Info
```

Template Haskell Beispiele

```
> runQ [| "Hi!" |]
LitE (StringL "Hi!")

> runQ [| 2 + 7 |]
InfixE (Just (LitE (IntegerL 2))) (VarE GHC.Num.+)
                                  (Just (LitE (IntegerL 7)))

> let x = [|\x ->  1 + x|]
> runQ x
LamE [VarP x_0] (InfixE (Just (LitE (IntegerL 1)))
                        (VarE GHC.Num.+) (Just (VarE x_0)))

> $(x) 6
7

> $( [| $(x) $ $(x) $ $([| \z -> z*2 |]) 3 |] )
8
```

Template Haskell Beispiele

Wer möchte, kann auch direkt auf der abstrakten Syntax operieren:

```
> let cnst n s =
    return (LamE (replicate n WildP) (LitE (StringL s)))

> :t $(cnst 5 "X")
$(cnst 5 "x") :: t -> t1 -> t2 -> t3 -> t4 -> [Char]

> $(cnst 3 "x") 1 2 3
"x"
```

Grundsätzlich empfiehlt sich dies jedoch nicht.

Beispiel: generisches Show (Anwendung)

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

data MyData = MyData { foo :: String, bar :: Int }

$(listFields $ mkName "MyData")
-- listFields ''MyData

main = print $ MyData { foo= "bar", bar= 5 }
```

- Top-Level Deklarationen:
  - müssen nicht explizit gespleißt werden
  - haben nur Zugriff auf vorangegangene Deklarationen
- Ausführung von Funktion während der Kompilierung erfordert separate Modul-Datei für diese $\Rightarrow$ stage restriction error

Beispiel: generisches Show (Definition)

```
{-# LANGUAGE TemplateHaskell, FlexibleInstances #-}
module CustomShow where

import Data.List
import Language.Haskell.TH

emptyShow :: Name -> Q [Dec]
emptyShow name =
  [d|instance Show $(conT name) where show _ = "x"|]

listFields :: Name -> Q [Dec]
listFields name = do
  TyConI (DataD _ _ _ [RecC _ fields] _) <- reify name
  let names = map (\(name,_,_) -> name) fields
  let showField :: Name -> Q Exp
      showField name = [|\x->s ++ "=" ++ show($(global name) x)|]
        where    s = nameBase name
  let showFields :: Q Exp
      showFields = listE $ map showField names
  [d|instance Show $(conT name) where
      show x = intercalate ", " (map ($ x) $showFields)|]
```

Yesod

Ein Web-Application Framework für Haskell:

- Typ-sichere URLs
- Modular
- Light-weight
- Asynchron
- Effizient

Es gibt mehrere Haskell-Web-Frameworks: Snap, Happstack, etc.

Yesod behauptet, eines der effizientesten und aktivsten davon zu sein.

Yesod unterstützt folgende Backends: Warp, FastCGI, SCGI, Webkit

```
{-# LANGUAGE TypeFamilies, QuasiQuotes, MultiParamTypeCl
             TemplateHaskell, OverloadedStrings #-}
import Yesod

data HelloWorld = HelloWorld

mkYesod "HelloWorld" [parseRoutes|
/ HomeR GET
|]

instance Yesod HelloWorld

getHomeR :: Handler RepHtml
getHomeR = defaultLayout [whamlet|Hello Yesod!|]

main :: IO ()
main = warpDebug 3000 HelloWorld
```

Scaffolding Tool

Yesod bringt ein Werkzug zum Erstellen eines Gerüstes mit:

```
yesod init
yesod configure
cabal install --reinstall
yesod devel
```

- Development Webserver auf `http://localhost:3000`
- Bei `yesod devel` werden die Dateien überwacht; der Development Webserver wird ggf. neugestartet.
- Verwendung von `cabal-dev` wird empfohlen.
- Das Gerüst hat zahlreiche nicht-zwingende, aber sinnvolle Voreinstellung, z.B. sind viele Dinge in mehreren Dateien getrennt, was nicht unbedingt notwendig ist.

Yesod Templates und Interpolation

HTML, CSS und JavaScript wird in Yesod mit Template Sprachen programmiert. Haskell kann diese Templates wie gewohnt manipulieren; Interpolation entspricht dem Spleißen und erlaubt es, Haskell-Werte im Template zu verwenden.

- `@{ }` Typsichere URL Interpolation, also `@{HomeR}`
- `#{ }` Interpolation für Variablen im Scope (escaped)
- `^{ }` Template embedding, fügt Template gleichen Typs ein

	Template Sprache	@{ }	#{ }	^{ }
Hamlet	HTML	✓	✓	✓
Cassius	CSS	✓	✓	
Lucius	CSS	✓	✓	
Julius	JavaScript	✓	✓	✓

Yesod Templates

Die Template Sprachen können per Quasi-Quotes oder in separaten Dateien verwendet werden. Das Gerüst-Tool empfiehlt separate Dateien. Separate Dateien für Cassius, Lucius und Julius können auch dynamisch ohne neue Kompilierung verwendet werden (Reload-Modus).

Sprache	Quasiquoter	Datei
Hamlet	`hamlet`	.hamlet
Cassius	`cassius`	.cassius
Lucius	`lucius`	.lucius
Julius	`julius`	.julius

Es gibt noch Varianten wie `shamlet`, `whamlet`, etc. und Quasiquoter `lt`,`st` für einfache Strings, was z.B. bei Internationalisierung hilfreich sein kann.

Hamlet

wie HTML plus Interpolation und:

- Schliessende tags durch Einrücken ersetzt

```
<p>Some paragraph.</p>
  <ul><li>Item 1</li>
      <li>Item 2</li></ul>
<p>Next paragraph.</p>
```

wird also zu

```
<p>Some paragraph.
  <ul>
    <li>Item 1
    <li>Item 2
<p>Some paragraph.
```

Hamlet

- Kurze geschlossene inline tags sind aber auch zulässig

```
<p>Some <i>italic</i> paragraph.
```

- Leerzeichen vor und nach Tags brauchen # und \

```
<p>
  Some #
  <i>italic
  \ paragraph.
```

- Attribute funktionieren wie in HTML, d.h. Gleichheitszeichen, Wert und Gänsefüßchen sind meist optional.
  Abkürzungen für IDs, Klassen und Konditionale

```
<p #paragraphid .class1 .class2>
<p :someBool:style="color:red">
<input type=checkbox :isChecked:checked>
```

Hamlet

Hamlet erlaubt auch logische Konstrukte

- Konditionale

```
$if isAdmin
  <p>Hallo mein Administrator!
$elseif isLoggedIn
  <p>Du bist nicht mein Administrator.
$else
  <p>Wer bist Du?
```

- Einfache Schleifen mit `forall`

```
$if null people
  <p>Niemand registriert.
$else
  <ul>
    $forall person <- people
      <li>#{person}
```

Hamlet

- Maybe & einfaches Pattern-Matching

```
$maybe name <- maybeName
  <p>Dein Name ist #{name}
$nothing
  <p>Ich kenne Dich nicht.

$maybe Person vorname nachname <- maybePerson
  <p> Dein Name ist #{vorname} #{nachname}
```

- Volles Pattern-Matching mit Case

```
$case foo
  $of Left bar
      <p>Dies war links: #{bar}
  $of Right baz
      <p>Dies war rechts: #{baz}
```

Hamlet

- `With` ist das neue Let, also für lokale Definitionen

```
$with foo <- myfun argument $ otherfun more args
  <p>
    Einmal ausgewertetes foo hier #{foo}
    und da #{foo} und dort #{foo} verwendet.
```

- Abkürzungen für Standard Komponenten:

```
$doctype 5
```

steht zum Beispiel für

```
<!DOCTYPE html>
```

Cassius

Wie CSS, aber

- plus Interpolation für Variablen und URLs
- Klammern und Semikolons müssen durch Einrücken ersetzt werden

```
#banner
  border: 1px solid #{bannerColor}
  background-image: url(@{BannerImageR})
```

Eignet sich für Whitespace-sensitive Haskell-Programmierer

⇒ Vorhandenen CSS Code aber besser mit Lucius weiterverwenden

Lucius

Lucius akzeptiert ganz normales CSS. Zusätzlich ist erlaubt

- Interpolation für Variablen und URLs
- CSS Blöcke dürfen verschachtelt werden
- Es können Variablen deklariert werden

Beispiel:

```
article code { background-color: grey; }
article p { text-indent: 2em; }
article a { text-decoration: none; }
```

kann bei Bedarf umgeschrieben werden zu

```
@backgroundcolor: grey;
article {
  code { background-color: #{backgroundcolor}; }
  p { text-indent: 2em; }
  a { text-decoration: none; }
```

Julius

Julius ist einfach nur gewöhnliches JavaScript, plus

- Variablen Interpolation
- URL Interpolation
- Template Embedding

Sonst ändert sich nichts.

Widgets

Widgets fassen einzelne Templates von verschiedenen Shakespear-Sprachen zu einer Einheit zusammen:

```
getRootR = defaultLayout $ do
  setTitle "My Page Title"
  toWidget [lucius| h1 { color: green; } |]
  addScriptRemote "https://ajax.googleapis.com/ajax/libs/jquery/1.
  toWidget [julius|
    $(function() {
      $("h1").click(function(){ alert("Clicked the heading!"); });
    });
  |]
  toWidgetHead [hamlet| <meta name=keywords content="keywords">|]
  toWidget [hamlet| <h1>Here's one way of including content |]
  [whamlet| <h2>Here's another |]
  toWidgetBody [julius| alert("This is included in the body"); |]
```

Die Widget-Monade erlaubt das kombinieren dieser Bausteine; alles wird automatisch dahin sortiert, wo es hingehört.

Widgets – whamlet

Template embedding erlaubt normalerweise nur die Einbettung aus der gleichen Template Sprache. Dagegen erlauben `whamlet` bzw. `.whamlet`-Dateien die Einbettung von Widgets in Hamlet:

```
page = [whamlet|
    <p>This is my page. I hope you enjoyed it.
    ^{footer}
  |]

footer = do
    toWidget [lucius| footer { font-weight: bold;
                               text-align: center } |]
    toWidget [hamlet|
        <footer>
          <p>That's all folks!
      |]
```

Frische Namen für frische Widgets

Bei der Kombination von Widgets könnten Namenskonflikte auftreten. Das wird durch dynamische IDs verhindert werden:

```
getRootR = defaultLayout $ do
  headerClass <- lift newIdent
  toWidget [hamlet|<h1 .#{headerClass}>My Header|]
  toWidget [lucius| .#{headerClass} { color: green; }
```

Die Funktione `newIdent` erlaubt die Erzeugung von frischen IDs.

Wir müssen `lift` einsetzen, da `newIdent` nicht in der `Widget`-Monade, sondern in der `Handler`-Monade lebt.

Yesod Typklasse

Jede Yesod Applikation muss eine Instanz der `Yesod`-Klasse sein. Genauer, der **Foundation** Datentyp muss als Instanz deklariert werden. Dieser Datentyp ist simpel, kann aber parametrisiert sein.

Diese Klasse fasst alle möglichen Einstellungen zusammen:

- Rendern und parsen von URLs
- Funktion `defaultLayout`
- Authentifizierung
- Sitzungsdauer
- Cookies
- Fehlerbehandlung und Aussehen der Fehlerseiten
- Externen CSS, Skripte und statische Dateien

Durch explizite Definition in der Instanzdeklaration kann man die Default bei Bedarf überschreiben.

Routing & Handling

Yesod nutzt eine DSL zu Spezifizierung Routen und Handlings:

```
/                 RootR      GET
/blog             BlogR      GET POST
/blog/#Int        BlogPostR  GET POST
/wiki/*WikiPfad   WikiR      GET
/static           StaticR    Static getStatic
```

Diese DSL wird innerhalb des Quasiquoters `parseRoutes` angegeben, oder aber in einer separaten Datei.

Definiert die gesamte Sitemap der Webapplikation.
Ausnahme: Kombination mit Yesod Unter-Websites

Routing & Handling

Yesod nutzt eine DSL zu Spezifizierung Routen und Handlings:

```
/                 RootR      GET
/blog             BlogR      GET POST
/blog/#Int        BlogPostR  GET POST
/wiki/*WikiPfad   WikiR      GET
/static           StaticR    Static getStatic
```

Zuerst wird der Pfad angegeben. Es gibt drei Arten von Pfaden:

- Statische Pfade
- Dynamische Pfade enthalten (mehrere) #**Typ** Fragmente, wobei **Typ** eine Instanz der Klasse `PathPiece` sein muss
- Dynamische Multipfade enden mit **Typ**, wobei **Typ** eine Instanz der Klasse `PathMultiPiece` sein muss

Handler von dynamischen Pfaden müssen entsprechende Argumente verarbeiten können.

Routing & Handling

Yesod nutzt eine DSL zu Spezifizierung Routen und Handlings:

```
/                 RootR      GET
/blog             BlogR      GET POST
/blog/#Int        BlogPostR  GET POST
/wiki/*WikiPfad   WikiR      GET
/static           StaticR    Static getStatic
```

Der zweite Teil definiert den Konstruktor für die typsichere URL:

- Können direkt in URL Interpolation `@{ }` verwendet werden
- Besitzen Argumente gemäß des dynamischen Pfades,
  z.B. `@{BlogPostR 7}`
- Das alle mit "R" enden ist nicht-zwingende Konvention

Routing & Handling

Yesod nutzt eine DSL zu Spezifizierung Routen und Handlings:

```
/                 RootR      GET
/blog             BlogR      GET POST
/blog/#Int        BlogPostR  GET POST
/wiki/*WikiPfad   WikiR      GET
/static           StaticR    Static getStatic
```

Der letzte Teil gibt die Request-Art an:

```
getBlogR     :: Handler RepHtml
postBlogR    :: Handler RepHtml
getBlogPostR :: Int -> Handler RepHtml
```

Handler-Namen müssen mit der Request-Art beginnen und mit dem Konstruktor der URL enden.

Handler leben in der `Handler` Monade.

Weitere Themen im nächsten Teil der Vorlesung:

- Webformulare
- Session Management
- Persistenz
- Authentifizierung
- Wechsel von Entwicklung zum produktiven Webserver
- Internationalisierung
- Subsites