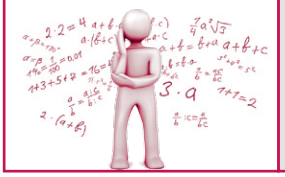


Vom Elfenbeinturm ins echte Leben

Haskell aus einer Java EE-Perspektive

Torsten Fink

Haskell ist wahrscheinlich die ausgereifteste funktionale Sprache, die in einem akademischen Umfeld entwickelt wurde. In den letzten Jahren hat sie sich sukzessive zu einer echten Alternative für die Entwicklung im geschäftlichen Umfeld entwickelt. In diesem Artikel wird Haskell durch die Java-Brille betrachtet, um zu diskutieren, ob sich ihr Einsatz in Projekten lohnt oder ob das Risiko in Abhängigkeit vom Nutzen noch zu groß ist.

Memories ...

Vor vielen Jahren, während meiner Zeit an der Uni, habe ich recht intensiv mit Haskell gearbeitet. Wir haben die funktionale Sprache zur Untersuchung paralleler Algorithmen benutzt. Die Fähigkeiten von Haskell waren beeindruckend. Hatte man seine Daten sauber modelliert, dann wurden viele fachliche Programmierfehler tatsächlich schon bei der Typüberprüfung des Compilers gefunden.

Der erste Kontakt mit Java damals (Version 1.1) war nach den Erfahrungen mit Haskell entsprechend negativ. Keine generischen Typen und die Notwendigkeit, Objekte auf den gewünschten Typ unsicher zu casten, fühlten sich nach einem deutlichen Rückschritt an. Andererseits waren wir dennoch mit Java erstaunlich produktiv. Nach dem Wechsel in die Industrie war Java gesetzt. Trotz der Mängel der Programmiersprache ermöglichte das Gesamtpaket Java in Kombination mit der lebhaften Community, effizient große Projekte umzusetzen.

Trotzdem habe ich die Entwicklung von Haskell beobachtet, die in den letzten Jahren eine interessante Wendung genommen hat. Die Programmiersprache ist langsam, aber kontinuierlich aus dem eher akademischen Umfeld hinaus gewachsen. Es sind Bibliotheken und Werkzeuge entstanden, die für Projekte im „echten Leben“ dringend notwendig sind. Hierzu gehören Build-Werkzeuge und Frameworks für Datenbankzugriffe und Webanwendungen. Mit der Industrial Haskell Group [IHG13] hat sich eine Organisation gebildet mit dem Ziel, den Einsatz von Haskell in industriellen Umgebungen zu fördern.

Interessant ist auch, dass in den letzten Jahren deutliche Bestrebungen nach neuen auf der JVM ausführbaren Programmiersprachen zu beobachten sind, die typische Eigenschaften funktionaler Sprachen aufweisen. Zu den prominentesten gehören Scala und Clojure. Daher war es an der Zeit, sich mal genauer den aktuellen Stand von Haskell anzuschauen. Als jemand, der jetzt über viele Jahre in Java-Projekten gearbeitet hat, habe ich die Gelegenheit genutzt, um mit der Java-Brille auf Haskell und die zugehörigen Frameworks und Werkzeuge zu schauen.

Der Artikel stellt einige der interessanten Fähigkeiten von Haskell vor und vergleicht die Sprache mit dem Stand der Technik in der Java-Welt. Allerdings soll hier keine Haskell-Einführung gegeben werden, die praktischen Beispiele werden daher nur so weit erklärt, dass die Argumentation nachvollziehbar ist. Wer an einer konkreten Einführung interessiert ist, dem seien [Lipo11] und [SuStGoe08] empfohlen. Die gezeigten Beispiele finden sich in Gänze auf Github unter `tnfink/Haskell-JavaPerspektive` zum freien Clonen. Die Haskell-Plattform mit Compiler und den Basisbibliotheken findet sich auf [Haskell-Platform].

Eine geschichtliche Betrachtung

Um ein Gefühl für die Unterschiede zwischen Haskell und Java zu bekommen, lohnt sich eine kurze historische Betrachtung. Abbildung 1 zeigt die wichtigsten Meilensteine. Auf der Konferenz „Functional Programming Languages and Computer Architecture“ wurde 1987 festgestellt, dass es aufgrund der vielen konkurrierenden funktionalen Programmiersprachen sinnvoller ist, die Forschungsarbeiten auf eine gemeinsame Sprache zu konzentrieren. Nach drei Jahren Komiteearbeit wurde 1990 die erste Sprachspezifikation verabschiedet. Sie wurde nach Haskell Curry benannt, einem amerikanischen Mathematiker.

Acht Jahre später wurde Haskell-98 veröffentlicht, die zweite stabile Sprachversion. Sie beinhaltete einige Spracherweiterungen und eine Standardbibliothek. Weitere 12 Jahre später kam die nächste stabile Version, die wiederum einige Erweiterungen beinhaltete. Interessant ist nun, dass auch die aktuellste Version der Sprache, die schon 23 Jahre alt ist, keinen Standard bereithält, um auf Datenbanken zuzugreifen, Benutzeroberflächen mit Web- oder Desktoptechnologien oder sonstige verteilte Systeme zu erstellen.

Die Geschichte von Java ist gänzlich anders gelaufen. Initial konzipiert als Sprache für Set-Top-Boxen wurde sie 1995 als Programmiersprache für das Web in einer ersten Alpha-Version veröffentlicht. Diese Version enthielt entsprechend schon alles, um interaktive Oberflächen zu erstellen. Zwei Jahre später kam die Version 1.1 heraus, die mit JDBC einen uniformen Zugriff auf Datenbanken ermöglichte. Wiederum zwei Jahre später, 1999, wurde die Java Enterprise Edition in der Version 1.2 publiziert, welche die typischen Anforderungen von Geschäftsanwendungen unterstützte, wie Nachrichtendienste und Transaktionalität. Sowohl die Basisversion als auch die Enterprise-Version wurden und werden kontinuierlich weiter entwickelt.

Grob und subjektiv zusammengefasst ist Haskell eine Sprache aus dem akademischen Elfenbeinturm, bei der durchgängig viel Wert auf hohe Qualität gelegt wurde. Java hingegen kommt aus der Praxis. Ihre Entwicklung wird getrieben von den Anforderungen aus dem industriellen Umfeld.

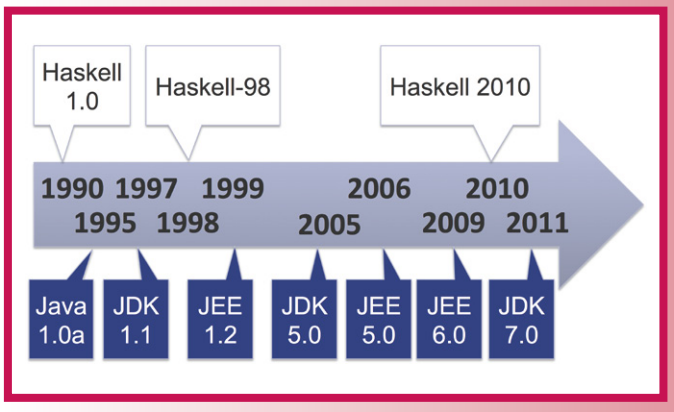
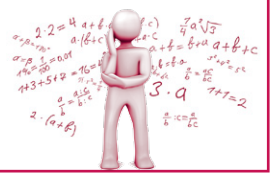


Abb. 1: Wichtige Meilensteine von Haskell und Java



Was ist cool an Haskell?

Wenn Haskell also die heile akademische Welt repräsentiert, welche Eigenschaften sind denn nun eigentlich so besonders und außergewöhnlich im Vergleich zu Java? Eine Bewertung der Sprachfähigkeiten ist natürlich stets subjektiv. Ich möchte diejenigen vorstellen, die aus meiner Perspektive die herausragendsten sind. Ich fange mit der *verzögerten Auswertung* an.

Auswertung on Demand

In Haskell ist es normal, mit beliebig großen Datenmengen zu arbeiten, sie dürfen sogar unendlich groß sein. Listing 1 zeigt zwei Beispiele. Die Referenz **nat** beschreibt die gesamte Menge der natürlichen Zahlen. Die Auswertung von **nat** findet erst statt, wenn Elemente der Liste für ein Ergebnis benötigt werden, und es werden auch nur die benötigten Elemente ausgewertet. So ist es möglich, bei endlichem Speicher mit unendlich großen Daten zu arbeiten. Ist der Entwickler nur an den geraden Zahlen interessiert, kann er einfach eine funktionale Filter-Funktion benutzen, siehe die Definition von **geradeZahlen** in Listing 1.

Das ist nicht nur beeindruckend, sondern auch hilfreich. Als Beispiel soll das Nummerieren von Zeilen dienen. In einer imperativen Sprache wie Java würde man dies mit einer Schleife umsetzen. Listing 1 zeigt die funktionale Variante. Die Referenz **zeilen** verweist auf eine Liste von Zeichenketten. Um diese zu nummerieren, wird sie per **zip** mit den natürlichen Zahlen verknüpft. Das Ergebnis ist eine Liste von Tupeln, bei denen jeweils das erste Element aus der einen Liste und das zweite aus der anderen genommen wird. Die funktionale Variante kommt damit ganz ohne imperative Kontrollstrukturen aus. Durch die verzögerte Auswertung ist das auch noch effizient.

```
nat = [1..]
geradeZahlen = filter even nat

zeilen = ["Eine","Zeile","nach","der","anderen"]
nummerierteZeilen = zip nat zeilen
> print nummerierteZeilen
[(1,"Eine"),(2,"Zeile"),(3,"nach"),(4,"der"),(5,"anderen")]
```

Listing 1: Verzögerte Auswertung in Haskell

Algebraische Datentypen und Kapselung

Während objektorientierte Sprachen Daten und Funktionen in Klassen zusammenbinden und per Vererbung Wiederverwendung ermöglichen, bieten funktionale Programmiersprachen, wie auch Haskell, meistens Datencontainer an, die sich über Konstruktoren definieren.

Listing 2 zeigt ein einfaches Beispiel. **BinTreeT** ist ein neuer generische Datentyp, um binäre Bäume mit beliebigen Knotentypen zu speichern. Die zwei Konstruktoren **Tree** und **Empty** erzeugen Exemplare. **Tree** erhält als Parameter zwei Unterbäume und einen Knotenwert. **Empty** definiert einen leeren Baum. Die Knotenwerte aller Knoten eines Baumes sind vom gleichen Typ **a**, der beliebig sein kann. Die Referenz **testTree** demonstriert, wie sich über Konstruktoren strukturierte Daten erzeugen lassen. Wer sich mit formalen Grammatiken beschäftigt hat, wird dieses Konzept sofort verstehen. Dieser Ansatz ermöglicht eine sehr genaue Modellierung der fachlichen Daten.

```
data BinTreeT a =
 Tree (BinTreeT a) a (BinTreeT a) |
 Empty

testTree = Tree Empty "K1"
                (Tree Empty "K2" Empty)

binSearch _ Empty = False
binSearch search (Tree left value right)
 | search < value = binSearch search left
 | search > value = binSearch search right
 | search == value = True

createBinTree liste = ....
  -- wird als Übung für den interessierten Leser
  -- offen gelassen :-)
```

Listing 2: Algebraische Datentypen

Funktionen auf algebraischen Datentypen lassen sich über Musterausdrücke sehr knapp und präzise formulieren. Die Funktion **binSearch** in Listing 2 durchsucht einen sortierten binären Baum. Diese Funktion wird über zwei Auswertungsregeln definiert, die sich durch unterschiedliche Muster unterscheiden. Die erste Regel besagt, dass, egal welcher Wert gesucht wird (_), dieser in einem leeren Baum (**Empty**) nicht zu finden ist. Die zweite Regel arbeitet auf einem Baum, der mit dem Konstruktor **Tree** erzeugt wurde. In bekannter rekursiver Vorgehensweise wird der Baum durchsucht. Dieser Code wäre in Java deutlich umfangreicher.

Kein OO, aber Funktionsklassen

Auch wenn Objektorientierung und insbesondere Vererbung in den letzten Jahren etwas in Verruf gekommen sind, so sind manche Eigenschaften einfach praktisch. Unabhängig davon, ob man Oberklassen oder Schnittstellen einsetzt, gibt es immer wieder Dinge mit ähnlichen Eigenschaften. Ein Klassiker ist die Ordnungseigenschaft, die es ermöglicht, zwei Dinge desselben Typs miteinander zu vergleichen und entsprechend zu sortieren. Es wäre ausgesprochen unpraktisch, wenn jeder Typ seine eigenen Vergleichsoperatoren hätte. Haskell bietet hierfür das Konzept von Funktionsklassen.

Listing 3 zeigt ausschnittsweise die Spezifikation der Funktionsklasse **Ord**, welche die Ordnungseigenschaft in Haskell umsetzt. Damit ein Typ **a** die Funktionsklasse **Ord** implementieren kann, muss **a** auch die Funktionsklasse **Eq** unterstützen, welche einen Identitätsvergleich beinhaltet. Eine Funktionsklasse besteht aus Funktionsnamen mit ihren Signaturen und Standardimplementierungen. In dem Ausschnitt sind die Funktionen **compare**, **>** und **<** aufgeführt. **compare** nimmt zwei Parameter vom Typ **a** und bildet sie auf den algebraischen Datentyp **Ordering** ab, der aus den drei Konstruktoren **LT**, **EQ** und **GT** besteht, wobei **LT** für „Lesser-Than“, **EQ** für „Equal“ und **GT** für „Greater-Than“ steht. Sie eignet sich damit als Grundlage, um die Funktionen **<** und **>** zu implementieren. In dem Auszug ist die Standardimplementierung für **<** gezeigt.

```
-- Ein Auszug aus der Standardfunktionsklasse Ord
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  … -- Snip
  x <  y =
   case compare x y of
     LT -> True
     _  -> False

-- Ordnungseigenschaft für unsere Bäume
instance (Ord a) => Ord (BinTreeT a) where
  compare tree1 tree2 = ... -- Implementierung der Vergleichsfunktion
```

Listing 3: Die Funktionsklasse Ord

Soll ein Typ nun eine Funktionsklasse unterstützen, so sind mindestens alle Funktionen der Funktionsklasse zu implementieren, für die keine Standardimplementierung vorhanden ist. In Listing 3 ist unten der Code ausschnittsweise dargestellt, damit unser Baumtyp `Ord` implementiert. Als Vorbedingung geben wir zusätzlich an, dass der Typ der Knotenwerte ebenfalls `Ord` unterstützen muss. Java bietet in der kommenden Version 8 ähnliche Möglichkeiten mit den neuen Standardimplementierungen für Interfaces an.

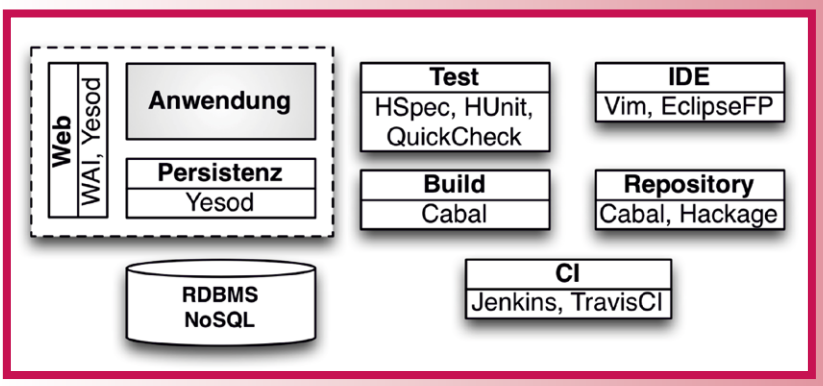


Abb. 2: Ausgewählte Bibliotheken, Frameworks und Werkzeuge

Statische Typisierung

Es gibt zwei gegensätzliche Ansätze bei der Typisierung in Programmiersprachen. Bei dynamischen Sprachen, wie JavaScript, ist der Typ etwas, was das Laufzeitsystem benötigt, um ausreichend Speicherplatz bereitzustellen und einem Methodenaufruf den richtigen Programmcode zuzuordnen. In statischen Sprachen wird der Typ vom Compiler benutzt, um den Code selbst zu überprüfen und optimierte Anwendungen zu erzeugen. Haskell ist sogar statisch stark typisiert. Das bedeutet, dass jeder Ausdruck in der Anwendung einen geprüften konkreten Typ hat. Für den Entwickler bedeutet dies, dass Typfehler zur Laufzeit in Haskell nicht möglich sind. Für den Compiler ergeben sich dadurch weitere Möglichkeiten, den erzeugten Code zu optimieren. Java ist aufgrund der Möglichkeit, den Typ von Objekten beliebig zur Laufzeit zu ändern, dynamischer als Haskell. Jeder Java-Entwickler ist entsprechend schon über Typfehler zur Laufzeit gestolpert.

Referentielle Transparenz

Die mit Abstand beeindruckendste Eigenschaft von Haskell besteht aber darin, dass jeder Ausdruck einen konstanten Wert hat. Daher wurde im gesamten Text bis jetzt der Begriff Variable vermieden. Haskell orientiert sich an der mathematischen Definition von Funktionen, nach der f(x) immer wieder das gleiche Ergebnis liefert, unabhängig davon, was in der Zwischenzeit passiert ist. Für den Entwickler bedeutet dies, dass es keine Seiteneffekte gibt. Um zu verstehen, was eine Funktion tut, genügt es, sich genau diese Funktion anzuschauen.

Der Vorteil unveränderbarer Zustände (Immutable State/Object) ist allgemein bekannt. Auch in Java folgen die meisten neuen Bibliotheken diesem Muster. Die meisten Sprachen besitzen dann aber doch immer wieder „Hintertüren“, die es ermöglichen, Variablen zu ändern. Haskell bleibt dagegen seinem Ansatz treu. Um dennoch mit externer Interaktion, Zufallsgeneratoren usw. umgehen zu können, wurde das Konzept von Monaden entwickelt, das es ermöglicht, Berechnungen von ihrer Umgebung zu trennen. Eine gut verständliche und praxisorientierte Einführung gibt [Monad06].

Schnelle Startzeiten

Im Gegensatz zu Java benutzt Haskell keine virtuelle Maschine, sondern erzeugt direkt ausführbare Programme. Entsprechend schnell starten diese. Eine Messung mit einem klassischen „Hello World“-Programm zeigt bei Haskell eine Startzeit von 3 ms. Die C-Variante war mit 2 ms kaum schneller. Java dagegen benötigte 157 ms. Für langlaufende Serveranwendungen ist dies nicht relevant, wohl aber für Anwendungen, die über die Kommandozeile gestartet werden. Erste Messungen für Serveranwendungen haben keine signifikanten Laufzeitunterschiede zwischen Java und Haskell ergeben.

Helferleins im Schweinsgalopp

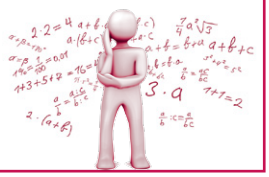
Die Praxistauglichkeit einer Programmiersprache ergibt sich meistens weniger aus ihren programmiersprachlichen Fähigkeiten als aus den verfügbaren Bibliotheken, Frameworks und Werkzeugen. Aus Platzgründen beschränken wir uns hier auf eine Schnellvorstellung einer subjektiv ausgewählten Menge von Helferleins.

Abbildung 2 stellt diese schematisch dar. Eine typische Geschäftsanwendung besitzt heutzutage eine Webschnittstelle und speichert ihre Daten in einer Datenbank ab. Es gibt im Haskell-Umfeld ein paar Webframeworks, ein interessantes ist Yesod [Yesod]. Es enthält neben der Webunterstützung auch eine Persistenzschicht, welche viele relationale Datenbanken und einige NoSQL-Datenbanken unterstützt. Eine knappe Vorstellung folgt später.

Auch in Haskell sind automatisierte Tests sinnvoll. Es existieren hierfür einige Standardframeworks, wie HUnit für klassische Modultests, HSpec für verhaltensgetriebene Tests (BDD) und QuickCheck für die automatisierte Erzeugung von Testdaten. Das Standard-Build-Framework in Haskell ist Cabal. Es ähnelt von seinen Fähigkeiten und Konzepten Maven. Hackage ist das von Cabal verwendete zentrale Repository, über das eine Vielzahl von Haskell-Bibliotheken verfügbar ist. Viele Haskell-Entwickler setzen tatsächlich auch heute noch Texteditoren, eventuell mit Syntax-Hervorhebung, ein. Die Java-Seite ist mir ihren ausgereiften Entwicklungsumgebungen hier deutlich weiter. Das fortgeschrittenste Entwicklungswerkzeug für Haskell ist EclipseFP, das auf Eclipse basiert.

Webanwendungen mit Yesod

Yesod ist ein serverseitiges, REST-orientiertes, typsicheres Webframework. Um einen ersten Eindruck zu vermitteln, zeigt Abbildung 3, wie eine einfache Webseite umgesetzt wird. Oben links ist die erzeugte Seite zu sehen, die aus einer einfachen Tabelle von Anwendern mit ein paar Navigationselementen besteht, deren Einträge direkt aus einer Tabelle geladen werden. Unten rechts ist das zugehörige HTML-Template dargestellt. Yesod unterstützt eine vereinfachte HTML-Syntax, bei der unter Einsatz von Einrückung auf schließende Tags verzichtet werden kann. Des Weiteren bietet es Elemente, die man auch von anderen serverseitigen Frameworks kennt, um etwa serverseitige Variablen einzubinden (z. B. `#{first}`) oder per Iteration HTML-Strukturen (z. B. `$forall`) aufzubauen.



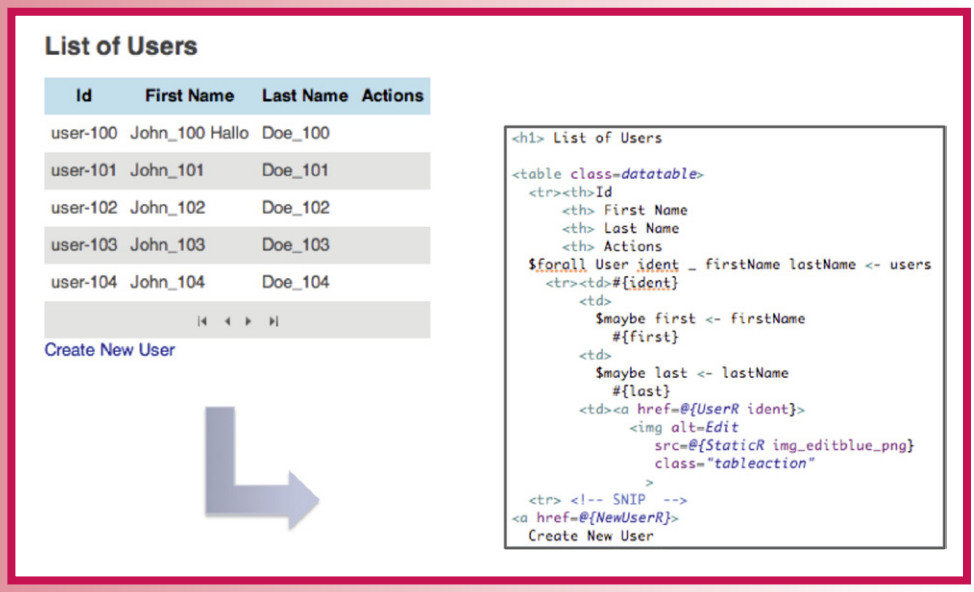


Abb. 3: Weboberflächen mit Yesod

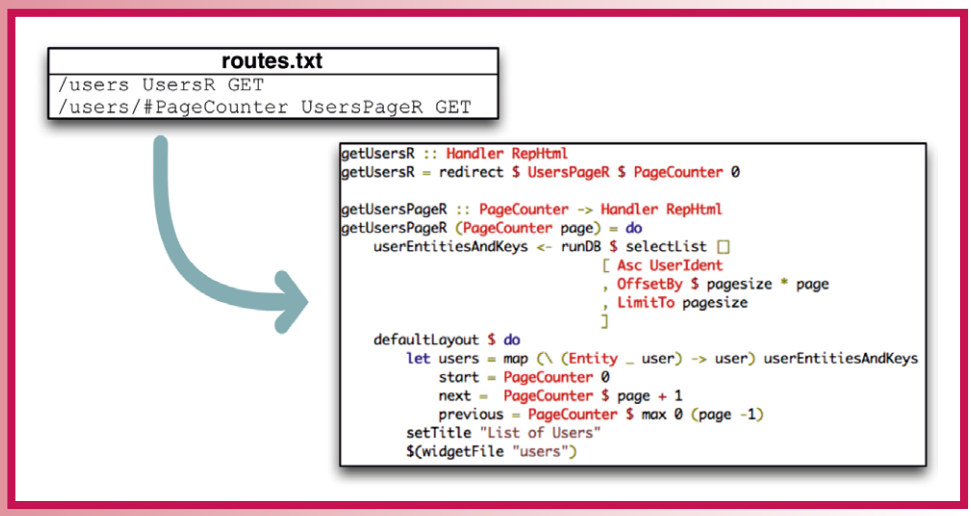


Abb. 4: Routing und DB-Zugriff mit Yesod

Abbildung 4 zeigt den zugehörigen Servercode. Über die Konfigurationsdatei `routes.txt` werden URLs mit den zugehörigen Handlern verknüpft. Die URL `/users` wird auf die Funktion `getUsersR` abgebildet. Diese führt eine Weiterleitung auf die URL `/users/0` aus, also auf die erste Seite der Tabelle, die von der Funktion `getUsersPageR` bearbeitet wird. Zuerst werden aus der Datenbank die Benutzer der angefragten Seite in die Referenz `userEntitiesAndKeys` geladen. Dann werden die Referenzen `users`, `start`, `next`, `previous` mit den Daten gefüllt, die an der Oberfläche angezeigt werden. Mit dem Ausdruck `$(widgetFile "users")` wird dann das in Abbildung 4 ausschnittsweise gezeigte HTML-Template geladen und ausgewertet.

Damit sollte das Prinzip von Yesod im Grundsatz verständlich sein. Alle Konfigurationsdateien, inklusive der HTML-Templates werden nach Haskell übersetzt und einer Typprüfung unterzogen. Somit ist nach der Übersetzung gewährleistet, dass alle Routendefinitionen, Links und Templates korrekt sind und keine Laufzeitfehler verursachen. Yesod ermöglicht es außerdem sehr einfach, REST-Schnittstellen für eine in JavaScript geschriebene Webanwendung bereitzustellen.

The winner is ...

... schwer zu bestimmen. Aus strategischer Sicht birgt Java sicherlich das geringere Risiko. Die Sprache und die zugehörigen Frameworks sind trotz mancher Mängel ausgereift, leistungsfähig und breit im Einsatz. Auf der anderen Seite bietet Haskell deutlich bessere programmiersprachliche Konzepte, welche die Chance für fehlerärmere Anwendungen bieten. Für datenbankzentrierte Webanwendungen ist Haskell inzwischen eine erwägenswerte Alternative. Was auf der Haskell-Seite aber noch fehlt, sind die typischen Enterprise-Features, wie XA-Transaktionen, Clustering und Anbindung an Nachrichtendienste.

Literatur und Links

[Haskell10] Haskell 2010 Language Report, `http://www.haskell.org/onlinereport/haskell2010/`

[Haskell98] Haskell 98 Language and Libraries, 2002, `http://www.haskell.org/onlinereport/`

[HaskellPlatform] `http://www.haskell.org/platform/`

[IHG13] Industrial Haskell Group, `http://industry.haskell.org`

[Lipo11] M. Lipovaca, Learn You a Haskell for Great Good!, no starch press, 2011, `http://learnyouahaskell.com`

[Monad06] D. Piponi, You could have invented Monads!, 7.8.2006, `http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html`

[SuStGoe08] B. O'Sullivan, D. Stewart, J. Goerzen, Real World Haskell, O'Reilly, 2008, `http://book.realworldhaskell.org`

[Yesod] Yesod Web Framework, `http://www.yesodweb.com`

Dr. Torsten Fink ist Geschäftsführer der Berliner Einheit der akquinet. Neben der Leitung von Java EE-Projekten führt er Architektur- und Technologieberatungen durch.
E-Mail: torsten.fink@akquinet.de