Haskell for EDSLs

Andres Löh

Utrecht University Well-Typed, LLP
andres@cs.uu.nl andres@well-typed.com

Capgemini
October 18, 2010
About me

- studied mathematics in Konstanz, Germany
- PhD in computer science in Utrecht 2004
- postdocs in Tallinn, Freiburg and Bonn
- UD at Utrecht University from 2007 until now
- as of now, partner at Well-Typed LLP, a company of Haskell consultants
Me and Haskell

- started using Haskell in 1996
- my main research interests are related to Haskell: datatype generic programming, advanced type systems, domain-specific languages
- have been teaching “Advanced Functional Programming” three times to master students, and “Applied Functional Programming” twice as a summer school
Haskell

- Is a standardized language.
- Designed by committee, actually designed by the community.
- First version 1990.
- Main implementation: GHC (Glasgow Haskell Compiler), developed at Microsoft Research in Cambridge.
- Several other implementations: Utrecht Haskell Compiler, Clean now has a Haskell frontend, YHC, JHC, LHC, Hugs, ...
Strengths of Haskell

- Language.
- Community.
Language
It is very easy to define your own datatypes in Haskell:

The structure of a company

```
data Company = C [Dept]
data Dept    = D Name Manager [Either Employee Dept]
data Employee = E Person Salary
data Person   = P Name Address
data Salary   = S Int
type Manager  = Employee
type Name     = String
type Address  = String```

A leaf-labelled binary tree
```
data Tree a = Node (Tree a) (Tree a) | Leaf a
```
Datatypes

It is very easy to define your own datatypes in Haskell:

The structure of a company

```haskell
data Company = C [Dept]
data Dept = D Name Manager [Either Employee Dept]
data Employee = E Person Salary
data Person = P Name Address
data Salary = S Int
type Manager = Employee
type Name = String
type Address = String
```

A leaf-labelled binary tree

```haskell
data Tree a = Node (Tree a) (Tree a)
            | Leaf a
```
Functions on user-defined datatypes can be defined using pattern matching:

The height of a tree

\[
\begin{align*}
\text{height} :: \text{Tree } a & \rightarrow \text{Int} \\
\text{height } (\text{Leaf } x) & = 0 \\
\text{height } (\text{Node } l \ r) & = 1 + \text{max} (\text{height } l) (\text{height } r)
\end{align*}
\]
The height of a tree

\[
\text{height} :: \text{Tree}\ a \rightarrow \text{Int} \\
\text{height}\ (\text{Leaf}\ x) = 0 \\
\text{height}\ (\text{Node}\ l\ r) = 1 + \max (\text{height}\ l) (\text{height}\ r)
\]
The height of a tree

\[
\text{height} :: \text{Tree } a \rightarrow \text{Int} \\
\text{height } (\text{Leaf } x) = 0 \\
\text{height } (\text{Node } l \ r) = 1 + \text{max} (\text{height } l) (\text{height } r)
\]

- The function works for all trees, regardless of label type.
- From looking at the type, we are guaranteed that the function does not touch the labels of the trees!
Type inference

The height of a tree

```haskell
height :: Tree a → Int
height (Leaf x) = 0
height (Node l r) = 1 + max (height l) (height r)
```

- Type signatures such as for `height` are optional! The compiler can infer them.
Type inference

The height of a tree

```haskell
height :: Tree a -> Int
height (Leaf x) = 0
height (Node l r) = 1 + max (height l) (height r)
```

- Type signatures such as for `height` are optional! The compiler can infer them.
- The compiler can infer quite advanced types, including overloaded operations:

  ```haskell
  [(=), (≠)] :: Eq a ⇒ [a → a → Bool]
  [(=), (≠), (<), (>)] :: Ord a ⇒ [a → a → Bool]
  [(=), (≠), (<), (>), (∧)] :: [Bool → Bool → Bool]
  ```
Effects

Java

```java
int add0 (int x, int y) {
    return x + y;
}
```

```java
int launch (now); return x + y;
```
Effects

Java

```java
int add0 (int x, int y) {
    return x + y;
}

int add1 (int x, int y) {
    launch_missiles (now);
    return x + y;
}
```

Both functions have the same type!
Both functions have the same type!
Effects

Haskell

\[
\begin{align*}
\text{add0} & :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
\text{add0} \ x \ y & = x + y \\
\text{add1} & :: \text{Int} \rightarrow \text{Int} \rightarrow \text{IO} \ \text{Int} \\
\text{add1} \ x \ y & = \text{launch\_missiles} \gg \text{return} \ (x + y)
\end{align*}
\]
Effects

Haskell

\[\text{add0} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\]
\[\text{add0}\ x\ y = x + y\]

\[\text{add1} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{IO}\ \text{Int}\]
\[\text{add1}\ x\ y = \text{launch\_missiles} \gg \text{return}\ (x + y)\]

Effectful computations are tagged by the type system!
Effects in Haskell’s types

We have rather fine-grained control about effects just by looking at the types:

- **A** some type, no effect
- **IO A** IO, exceptions, random numbers, concurrency, ...
- **Gen A** random numbers only
- **ST s A** mutable variables only
- **STM A** software transactional memory log variables only
- **State s A** (persistent) state only
- **Error A** exceptions only
- **Signal A** time-changing value

New effect types can be defined. Effects can be combined.
Purity

Being explicit about all effects is called *purity*. 
Being explicit about all effects is called *purity*.

Purity is one of the most special features of Haskell.

- We can see from the type what effects a function might have.
- If the result type is not tagged by an effect, we know the function is a pure function in the mathematical sense.
- Keeping track of effects is great for optimizations, guaranteeing program correctness and also testing.
In Haskell, we can define `if then else` as a function (if we want):

```
ifthenelse True  thenPart elsePart = thenPart
ifthenelse False thenPart elsePart = elsePart
```
Laziness

In Haskell, we can define \textbf{if then else} as a function (if we want):

\begin{verbatim}
ifthenelse True thenPart elsePart = thenPart
ifthenelse False thenPart elsePart = elsePart
\end{verbatim}

\begin{itemize}
  \item The whole point of \texttt{ifthenelse} is to avoid executing one of the parts.
  \item In a lazy language, arguments are evaluated on demand.
  \item Hence, in a lazy language, we can define our own control-flow constructs (loops, case distinctions, iterators, coroutines, etc.)
\end{itemize}
Interacting with C and other languages

No language today can be used in isolation.

- Haskell supports an FFI (foreign function interface) to import functions from C and export functions, too.
- Haskell also provides libraries that translate between Haskell’s and C’s data model in an efficient way, and handle the different memory management models.
- The FFI has been used extensively to provide bindings for various common C and C++ libraries to Haskell.
- Many of Haskell’s standard library functions are mapped to C libraries.
- Other examples: OpenGL, Gtk, LLVM, compression/codecs/cryptography, image formats, Berkeley DB, Python, matlab, Chipmunk, OGRE, SDL, X11, BLAS, …
Community and infrastructure
Infrastructure

- Most libraries and software are open source.
- Most frequently used license: BSD.
- Core GHC team (2 developers, plus maintenance by Well-Typed) is sponsored by Microsoft Research, but many volunteers help.
- Many, many contributors for libraries.

Haskell Platform:

- An attempt to facilitate installation of a Haskell toolchain. Supported on Windows, Linux, and Mac.
- Core set of packages.
- Release independently of GHC, once every 6 months.
Cabal and Hackage

Cabal:

- library to facilitate the building and distribution of Haskell packages in a uniform format,
- handles dependencies with other Haskell packages,
- support for several Haskell compilers.

Hackage:

- a package repository for community-supplied Haskell libraries and applications,
- about 2500 packages are on Hackage now,
- contributed by 628 developers,
- about 3 million total downloads; about 160K downloads per month.
Cabals and Hackage

Cabals:
- library to facilitate the building and distribution of Haskell packages in a uniform format,
- handles dependencies with other Haskell packages,
- support for several Haskell compilers.

Hackage:
- a package repository for community-supplied Haskell libraries and applications,
- about 2500 packages are on Hackage now,
- contributed by 628 developers,
- about 3 million total downloads; about 160K downloads per month.
Commercial use

Haskell is enjoying more and more commercial success:

- Galois, Inc in Portland, Oregon is a rapidly growing award-winning company using Haskell exclusively
- Well-Typed LLP is a successful Haskell consultancy based in Oxford with various clients
- The Industrial Haskell Group is a consortium of companies using Haskell supporting Haskell development
- More companies are listed on the Haskell Wiki: for example, Amgen, Standard Chartered, Deutsche Bank, Barclays Capital, Facebook, Google, plus many smaller companies and startups (for example: TypLAB/Silk in Amsterdam).
- For more information, see also the website of the “Commercial Users of Functional Programming” conference.
Haskell remains an active research language.

The annual “International Conference on Functional Programming” and “Haskell Symposium” see many Haskell-related academic publications and talks.

Haskell is still in development and gradually evolving.

The Haskell standard tries to address the concerns of both the research and the commercial users.

Haskell inspires many other languages, but also remains rather unique (purity).
Community

Haskell has an amazing, active, very helpful community.

- Friendly to beginners.
- Trying hard to improve the overall experience.
- Various media: Haskell Wiki, mailing lists, Reddit, Stackoverflow, blogs/planet, IRC, . . .
- Events: Hackathons, Google Summer of Code, Haskell Symposium, Haskell Implementors Workshop, . . .
(E)DSLs
Languages are everywhere

- Nearly every IT concept is based on a language (even if you never see it).
- Nearly every IT tool is a compiler (translating one language into another).
What is an (E)DSL?

- **DSL** = domain-specific language
- **EDSL** = embedded DSL
- In Haskell, we can easily define datatypes, higher-order functions, control-flow constructs, operators, normal functions
- Together, we can often simulate the appearance of other languages within Haskell, or create special-purpose domain-specific sublanguages that allow to specify problems concisely
Example: SQL
Classical approach

Build SQL queries as strings.
Classical approach

Build SQL queries as strings.

Disadvantages

- leads to an ad-hoc, low-level, programming style
- no guarantee that the statement is syntactically correct
- even if it sometimes is correct, it may not always be
- potential security problems due to lack of escaping
- errors occur at run-time and are often hard to debug
C# has LINQ (Language Integrated Query):

```csharp
var query =
	from cust in db.Customers
	where cust.City == "Utrecht"
	select new { cust.CustomerID };
```
Built-in language features

C# has LINQ (Language Integrated Query):

```csharp
var query =
    from cust in db.Customers
    where cust.City == "Utrecht"
    select new { cust.CustomerID };
```

Much better

- SQL queries are written directly within the language
- properly syntax- and type-checked
- errors will be reported in terms of the programming language
- can be translated to various backends
- escaping can be handled once by the backend
- but: limited to whatever is provided by LINQ
query =
    do cust ← table customers
        restrict (cust ! city .==. "Utrecht")
        project (cust ! customerID)
HaskellDB

query =
\[\text{do}\] cust \leftarrow \text{table customers}
    \text{restrict} (\text{cust} ! \text{city} \ := \text{"Utrecht"})
    \text{project} (\text{cust} ! \text{customerID})

Nearly perfect

- Same level of complexity as LINQ.
- You still get the syntax- and type-safety.
- Just a normal Haskell library.
- If you do not like the syntax, you can change it.
- If you need additional abstractions, you can define them.
- If you have another domain, you just define another library.
- Or even better, you use one already available on Hackage.
There are a multitude of EDSLs available for Haskell:

- for defining grammars and parsers
- for pretty-printing abstract syntax
- for defining attribute grammars
- for specifying (unit) tests and program properties
- for drawing and composing 2D images (for example, OpenGL)
- for defining images and animations
- for composing and layouting GUIs (Gtk, wxWidgets, Qt, ...) 
- for writing JavaScript programs
- for defining music
- for concurrent orchestration
- for web development
- for specifying hardware layouts
- ...
Example property

propInsertDelete :: a → Stack a → Bool
propInsertDelete x s = toList (delete x (insert x s)) == toList s
Example: QuickCheck

Example property

```
propInsertDelete :: a -> Stack a -> Bool
propInsertDelete x s = toList (delete x (insert x s)) == toList s
```

- Properties of programs can be written as Haskell functions.
- QuickCheck is a library that can automatically generate test cases and test these functions.
- All Haskell abstractions can be used in order to define properties.
- Test cases are typechecked and serve as additional documentation.
Example document

htmlPage content =
  (header « ((thetitle « "Testing forms")
  +++ (script ![thetype "text/javascript",
    src "http://ajax.google..."] « "")
  +++ (script ![thetype "text/javascript",
    src massInputJSFile] « "")
  ))
  +++ (body « content)
Example document

```haskell
htmlPage content =
  (header <<< ((thetitle <<< "Testing forms")
               +++ (script ![thetype "text/javascript",
                            src "http://ajax.google..."] <<< "")
               +++ (script ![thetype "text/javascript",
                            src massInputJSFile] <<< "")
  ))
  +++ (body <<< content)
```

- Haskell rather than HTML syntax.
- Immediate typechecking to the XHTML specification (no improper nesting).
- Own abstractions possible: higher-level composition, automatic escaping of entities, ...
Example: parsers

Example parser

expr :: Parse Expr
expr = Let <$ keyword "let" <*> decl <*> keyword "in" <*> expr
     <|> operatorExpr

Syntax inspired by (E)BNF.
Own abstractions.
Type safety.
Advanced analyses possible.
Example: parsers

Example parser

```haskell
expr :: Parse Expr
expr = Let <$ keyword "let" <*> decl <*> keyword "in" <*> expr <|> operatorExpr
```

- Syntax inspired by (E)BNF.
- Own abstractions.
- Type safety.
- Advanced analyses possible.
Example: Parallel programming

Example: parallel map over a list

\[
\text{parMap strat } f = ('\text{using'} \ \text{parList strat}) \circ \text{map } f
\]
Example: Parallel programming

Example: parallel map over a list

\[
\text{parMap strat } f = (\text{`using` parList strat}) \circ \text{map } f
\]

- We can apply strategies to existing functions in order to tell Haskell how to parallelize them.
- Only two primitives needed: rpar and rseq. The former hints that something should be computed in parallel, the latter explicitly sequences two operations.
Example: datatype-generic programming

Traversal example

\[
\text{optimise} :: \text{Expr} \rightarrow \text{Expr} \\
\text{optimise} = \text{transform} \$ \\
\lambda x \rightarrow \text{case } x \text{ of} \\
\quad \text{Neg (Val i)} \rightarrow \text{Val (negate i)} \\
\quad x \rightarrow x
\]
Example: datatype-generic programming

Traversal example

```haskell
optimise :: Expr → Expr
optimise = transform $ λx → case x of
  Neg (Val i) → Val (negate i)
  x          → x
```

- Functions such as `transform` recursively traverse an arbitrary data structure.
- We only write the interesting case. This is completely type-safe and very robust to change.
- Datatype-genericity is a quite powerful concept, quite related to MDE.
Conclusions

Haskell should be considered as an implementation language:

- Culture of relatively short, high-quality code.
- Rapid prototyping.
- Type safety, more modular, easier to test and maintain.
Conclusions

Haskell should be considered as an implementation language:

- Culture of relatively short, high-quality code.
- Rapid prototyping.
- Type safety, more modular, easier to test and maintain:

Potential disadvantages:

- Writing good Haskell code requires training.
- In particular when it comes to performance.
- Toolchain may have a less “professional feel” than for other PLs.
Conclusions

Haskell should be considered as an implementation language:

▶ Culture of relatively short, high-quality code.
▶ Rapid prototyping.
▶ Type safety, more modular, easier to test and maintain:

Potential disadvantages:

▶ Writing good Haskell code requires training.
▶ In particular when it comes to performance.
▶ Toolchain may have a less “professional feel” than for other PLs.

However:

▶ Purity is really worth it (compared to F#, Scala, OCaml).
▶ Competitive advantage.
▶ Many excellent Haskell programmers waiting to be hired.
▶ Haskell is more fun.
Questions?