Haskell for EDSLs

Andres Löh

Utrecht University
andres@cs.uu.nl

Well-Typed, LLP
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.
- Usable, stable version: Haskell 1998.
- Current standard: Haskell 2010.
- 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

Datatypes

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
```

Datatypes

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
```

Pattern matching

Functions on user-defined datatypes can be defined using pattern matching:

The height of a tree

$$
\begin{aligned}
&\mathsf{height} :: \mathsf{Tree}\ \mathsf{a} \to \mathsf{Int} \\
&\mathsf{height}\ (\mathsf{Leaf}\ \mathsf{x}) \quad = 0 \\
&\mathsf{height}\ (\mathsf{Node}\ \mathsf{l}\ \mathsf{r}) = 1 + \mathsf{max}\ (\mathsf{height}\ \mathsf{l})\ (\mathsf{height}\ \mathsf{r})
\end{aligned}
$$

Polymorphism

The height of a tree

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

Polymorphism

The height of a tree

$$\begin{aligned}
&\mathsf{height} :: \mathsf{Tree}\ \mathsf{a} \to \mathsf{Int} \\
&\mathsf{height}\ (\mathsf{Leaf}\ \mathsf{x}) \quad = 0 \\
&\mathsf{height}\ (\mathsf{Node}\ \mathsf{l}\ \mathsf{r}) = 1 + \mathsf{max}\ (\mathsf{height}\ \mathsf{l})\ (\mathsf{height}\ \mathsf{r})
\end{aligned}$$

- 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

$$
\begin{aligned}
&\mathsf{height} :: \mathsf{Tree}\ \mathsf{a} \to \mathsf{Int}\\
&\mathsf{height}\ (\mathsf{Leaf}\ \mathsf{x}) \quad = 0\\
&\mathsf{height}\ (\mathsf{Node}\ \mathsf{l}\ \mathsf{r}) = 1 + \mathsf{max}\ (\mathsf{height}\ \mathsf{l})\ (\mathsf{height}\ \mathsf{r})
\end{aligned}
$$

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

Type inference

The height of a tree

$$
\begin{aligned}
&\mathsf{height} :: \mathsf{Tree}\ \mathsf{a} \to \mathsf{Int} \\
&\mathsf{height}\ (\mathsf{Leaf}\ \mathsf{x}) \quad = 0 \\
&\mathsf{height}\ (\mathsf{Node}\ \mathsf{l}\ \mathsf{r}) = 1 + \mathsf{max}\ (\mathsf{height}\ \mathsf{l})\ (\mathsf{height}\ \mathsf{r})
\end{aligned}
$$

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

$$
\begin{array}{lllll}
[(==), (\not\equiv)] & :: \mathsf{Eq}\ \mathsf{a} & \Rightarrow [\mathsf{a} \to & \mathsf{a} & \to \mathsf{Bool}] \\
[(==), (\not\equiv), (<), (>)] & :: \mathsf{Ord}\ \mathsf{a} & \Rightarrow [\mathsf{a} \to & \mathsf{a} & \to \mathsf{Bool}] \\
[(==), (\not\equiv), (<), (>), (\wedge)] & :: & [\mathsf{Bool} \to & \mathsf{Bool} & \to \mathsf{Bool}]
\end{array}
$$

Effects

Java

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

Effects

Java

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

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

Effects

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!

Effects

Haskell

$$\mathsf{add0} :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$$
$$\mathsf{add0}\ x\ y = x + y$$
$$\mathsf{add1} :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{IO\ Int}$$
$$\mathsf{add1}\ x\ y = \mathsf{launch\_missiles} \gg \mathsf{return}\ (x + y)$$

Effects

Haskell

$$\mathsf{add0} :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$$
$$\mathsf{add0}\ x\ y = x + y$$
$$\mathsf{add1} :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{IO}\ \mathsf{Int}$$
$$\mathsf{add1}\ x\ y = \mathsf{launch\_missiles} \gg \mathsf{return}\ (x + y)$$

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*.

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.

Laziness

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 **if then else** as a function (if we want):

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

- The whole point of ifthenelse is to avoid executing one of the parts.
- In a lazy language, arguments are evaluated on demand.
- Hence, in a lazy language, we can define our own control-flow constructs (loops, case distinctions, iterators, coroutines, etc.)

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.

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.

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.

Research use

- 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 everwhere

- 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

Built-in language features

C# has LINQ (Language Integrated Query):

```
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):

```
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

HaskellDB

```
query =
  do cust ← table customers
     restrict (cust ! city .==. "Utrecht")
     project (cust ! customerID)
```

HaskellDB

```
query =
  do cust ← table customers
     restrict (cust ! city .==. "Utrecht")
     project (cust ! 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.

Hackage and DSLs

There are a multitude of EDSLs available for Haskell:

- for defining grammars and parsers
- for pretty-printing abstract syntax
- for defining attribute grammars
- for specfying (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: QuickCheck

Example property

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

Example: QuickCheck

Example property

```haskell
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: (X)HTML

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: (X)HTML

Example document

```
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
```

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: Parallel programming

Example: parallel map over a list

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

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

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

Example: datatype-generic programming

Traversal example

$$
\begin{aligned}
&\mathsf{optimise} :: \mathsf{Expr} \to \mathsf{Expr}\\
&\mathsf{optimise} = \mathsf{transform}\ \$\\
&\quad \lambda \mathsf{x} \to \textbf{case}\ \mathsf{x}\ \textbf{of}\\
&\qquad\qquad \mathsf{Neg}\ (\mathsf{Val}\ \mathsf{i}) \to \mathsf{Val}\ (\mathsf{negate}\ \mathsf{i})\\
&\qquad\qquad \mathsf{x} \qquad\qquad\quad\ \to \mathsf{x}
\end{aligned}
$$

- 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?