An overview of Haskell

Haggai Eran

23/7/2007

Outline

Introduction

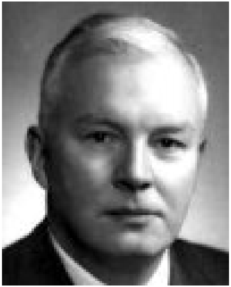
Haskell is a pure functional language. It means that:

- Variables never change after definition.
- Functions don't have side effects.
- Functions always return the same output given the same input.

History

- Designed by a committee. [1990s]
- (Nevertheless, it is an elegant language.)
- Haskell 98 - (Informal) standardization, and basis for further development.

Named after Haskell B. Curry:



History

Designed by a committee

Nice syntactic features

Guards

- Standard if-then-else:

$$
\begin{aligned}
my\_gcd_1\ m\ n = \textbf{if}\ n \equiv 0 \quad & \textbf{then}\ m \\
\textbf{else if}\ m < n\ & \textbf{then}\ my\_gcd_1\ n\ m \\
\textbf{else}\ & my\_gcd_1\ n\ (m\ `mod`\ n)
\end{aligned}
$$

- Guards:

$$
\begin{aligned}
my\_gcd_2\ m\ 0 \quad & = m \\
my\_gcd_2\ m\ n \mid m < n \quad & = my\_gcd_2\ n\ m \\
\mid \textbf{otherwise} \quad & = my\_gcd_2\ n\ (m\ `mod`\ n)
\end{aligned}
$$

Nice syntactic features

Pattern Matching

- Simple Case expressions:

$$
\begin{aligned}
&\mathit{factorial}_1\ n = \textbf{case}\ n\ \textbf{of}\\
&\quad 0 \rightarrow 1\\
&\quad n \rightarrow n * \mathit{factorial}_1\ (n-1)
\end{aligned}
$$

- Pattern Matching:

$$
\begin{aligned}
&\mathit{factorial}_2\ 0 = 1\\
&\mathit{factorial}_2\ n = n * \mathit{factorial}_2\ (n-1)
\end{aligned}
$$

Lists

A list in Haskell is defined recursively.

Definition

$$\textbf{data}\ [a] = [\,] \mid a : [a]$$

And there's some syntactic sugar for using lists:

$$[1\,..\,3] \equiv [1,2,3] \equiv 1:[2,3] \equiv 1:2:3:[\,]$$

Lazy Lists

Since Haskell is a lazy language, you can define infinite lists:

$$
\begin{array}{l}
\mathit{primes} = \mathit{sieve}\ [2\,..]\ \textbf{where} \\
\quad \mathit{sieve}\ (p : \mathit{tail}) = \textbf{let} \\
\qquad \mathit{filtered\_tail} = \mathit{sieve}\ [n \mid n \leftarrow \mathit{tail}, n\ \text{`}\mathit{mod}\text{`}\ p > 0] \\
\qquad \textbf{in}\ p : \mathit{filtered\_tail}
\end{array}
$$

$$
\begin{array}{l}
\mathit{factorial\_list} = 1 : [a * n \mid a \leftarrow \mathit{factorial\_list} \\
\qquad\qquad\qquad\qquad\quad \mid n \leftarrow [1\,..]]
\end{array}
$$

QuickSort

$$
\begin{aligned}
&\mathit{quicksort}\ [\,] = [\,] \\
&\mathit{quicksort}\ (\mathit{hd} : \mathit{tail}) = \mathit{quicksort}\ \mathit{small} +\!\!+ [\mathit{hd}] +\!\!+ \mathit{quicksort}\ \mathit{large} \\
&\quad \textbf{where} \\
&\qquad \mathit{small} = [x \mid x \leftarrow \mathit{tail}, x \leqslant \mathit{hd}] \\
&\qquad \mathit{large} = [x \mid x \leftarrow \mathit{tail}, x > \mathit{hd}]
\end{aligned}
$$

Currying

$$inc\ x = 1 + x$$

$$inc\ x = (+)\ 1\ x$$

$$inc = (+)\ 1$$

$$inc = (+1)$$

Pointfree programming

$$h\ x = f\ (g\ (x))$$

$$h\ x = (f\ .\ g)\ (x)$$

$$h = f\ .\ g$$

Type System Introduction

Haskell uses static typing, but is very expressive because of its polymorphism and type classes.

Example

$$
\begin{aligned}
&\mathit{reverse}_1 :: [a] \rightarrow [a] \\
&\mathit{reverse}_1\ [\,] && = [\,] \\
&\mathit{reverse}_1\ (\mathit{hd} : \mathit{tail}) && = \mathit{reverse}_1\ \mathit{tail} + [\mathit{hd}]
\end{aligned}
$$

Since *reverse_list* is polymorphic, you can use it for any type of list:

- $\mathit{reverse}_1\ [1, 2, 3] \rightarrow [3, 2, 1]$
- $\mathit{reverse}_1$ `"Hello, World"` $\rightarrow$ `"dlroW ,olleH"`

An efficient reverse

Algebraic Data Types

Haskell supports user defined algebraic data types, which combined with pattern matching are very expressive.

$$\textbf{data}\ \textbf{Maybe}\ a = \textbf{Nothing} \mid \textbf{Just}\ a$$

Example

$$\begin{aligned}
&divide :: (Integral\ a) \Rightarrow a \rightarrow a \rightarrow \textbf{Maybe}\ a\\
&divide\ x\ 0 = \textbf{Nothing}\\
&divide\ x\ y = \textbf{Just}\ (x\ `div`\ y)
\end{aligned}$$

Algebraic Data Types

Decomposition using pattern matching

Example

$$\mathit{default\_value} :: \textbf{Maybe}\ a \rightarrow a \rightarrow a$$
$$\mathit{default\_value}\ \textbf{Nothing}\ x = x$$
$$\mathit{default\_value}\ (\textbf{Just}\ x)\ \_ = x$$

Algebraic Data Types

Describing complex data structures

Complex data structures can be described (without pointers, of course).

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Branch}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

$$\begin{aligned}
&\mathit{size} :: \mathit{Tree}\ a \rightarrow \mathit{Int} \\
&\mathit{size}\ (\mathit{Leaf}\ _) &&= 1 \\
&\mathit{size}\ (\mathit{Branch}\ \mathit{left}\ \mathit{right}) &&= 1 + \mathit{size}\ \mathit{left} + \mathit{size}\ \mathit{right}
\end{aligned}$$

Encapsulation

There is no abstract type construct in Haskell, but instead there is a hierarchial module system, which can be used for encapsulation.

Example

$$
\begin{array}{l}
\textbf{module}\ \mathit{Stack}\ (\mathit{Stack}, \mathit{push}, \mathit{pop}, \mathit{empty}, \\
\qquad\qquad\qquad \mathit{top}, \mathit{is\_empty})\ \textbf{where} \\
\quad \begin{array}{lcl}
\textbf{data}\ \mathit{Stack}\ a & = & \mathit{Stk}\ [a] \\
\mathit{empty} & = & \mathit{Stk}\ [\,] \\
\mathit{push}\ (\mathit{Stk}\ s)\ x & = & \mathit{Stk}\ (x : s) \\
\mathit{pop}\ (\mathit{Stk}\ (x : s)) & = & \mathit{Stk}\ s \\
\mathit{top}\ (\mathit{Stk}\ (x : s)) & = & x \\
\mathit{is\_empty}\ (\mathit{Stk}\ s) & = & \mathit{null}\ s
\end{array}
\end{array}
$$

Type Classes

In Haskell, Type classes allow both overloading names, and writing generic functions which are made specific for some class.

Example

$$
\begin{array}{l}
\textbf{class}\ Eq\ a\ \textbf{where} \\
(\equiv) :: a \rightarrow a \rightarrow Bool \\
(\not\equiv) :: a \rightarrow a \rightarrow Bool
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ Eq\ Int\ \textbf{where} \\
\quad i1 \equiv i2 = eqInt\ i1\ i2 \\
\quad i1 \not\equiv i2 = \textbf{not}\ (i1 \equiv i2)
\end{array}
$$

Type Classes
Generic Classes and Functions

Example

$$
\begin{array}{l}
\textbf{instance}\ (Eq\ a) \Rightarrow Eq\ [a]\ \textbf{where} \\
\quad [\,] \equiv [\,] = True \\
\quad (x : xs) \equiv (y : ys) = x \equiv y\ \&\&\ xs \equiv ys \\
\quad xs \not\equiv ys = \textbf{not}\ (xs \equiv ys)
\end{array}
$$

$$
\begin{array}{l}
\textbf{member} :: Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool \\
\textbf{member}\ x\ [\,] = False \\
\textbf{member}\ x\ (y : ys) \mid x \equiv y = True \\
\qquad\qquad\qquad\qquad \mid \textbf{otherwise} = \textbf{member}\ x\ ys
\end{array}
$$

Higher Order Functions

Functions are first-class values, and can be passed to other functions.

Example

$$
\begin{aligned}
&\mathbf{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
&\mathbf{map}\ f\ [\,] = [\,] \\
&\mathbf{map}\ f\ (head : tail) = (f\ \ head) : (\mathbf{map}\ f\ \ tail)
\end{aligned}
$$

$$
\begin{aligned}
&inc\ \ :: (Num\ a) \Rightarrow a \rightarrow a \\
&(*3) :: (Num\ a) \Rightarrow a \rightarrow a
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{map}\ inc\ [1,2,3] \equiv [2,3,4] \\
&\mathbf{map}\ (*3)\ [1,2,3] \equiv [3,6,9]
\end{aligned}
$$

map - More Uses

$$\mathit{toUpper} :: \mathit{Char} \rightarrow \mathit{Char}$$
$$\textbf{map}\ \mathit{toUpper}\ \texttt{"Hello"} \equiv \texttt{"HELLO"}$$

You can even define:

$$\mathit{stringToUpper} :: \mathit{String} \rightarrow \mathit{String}$$
$$\mathit{stringToUpper} = \textbf{map}\ \mathit{toUpper}$$

IO and Monads

Pure functional language $\Rightarrow$ No side-effects in functions.
So how can we perform IO?
With the IO Monad!
A value of the type **IO** *a* represent an action, which returns a value of type *a*, once performed.

Example

$$\mathbf{getLine} :: \mathbf{IO}\ \mathit{String}$$
$$\mathbf{putStr} :: \mathit{String} \rightarrow \mathbf{IO}\ ()$$

IO Syntax

Example

```haskell
greet :: String -> String
greet name = "Hello, " ++ name

main :: IO ()
main = do
  name <- getLine
  putStrLn (greet name)
```

Monadic Pointfree Syntax

Example

```haskell
echo :: IO ()
echo = putStr ">  " >>
       getLine      >>=
       putStr       >>
       putStr "\n"
```

▸ The Monad Type Class

The Maybe Monad

Maybe

$$
\begin{aligned}
&f :: \mathit{Int} \rightarrow \textbf{Maybe}\ \mathit{Int} \\
&\mathit{complex\_function} :: \textbf{Maybe}\ \mathit{Int} \rightarrow \textbf{Maybe}\ \mathit{Int} \\
&\mathit{complex\_function}\ \mathit{mint} = \textbf{do} \\
&\quad i1 \leftarrow \mathit{mint} \\
&\quad i2 \leftarrow f\ i1 \\
&\quad \textbf{return}\ i2
\end{aligned}
$$

The List Monad

List

$$
\begin{aligned}
&(\times) :: [a] \to [b] \to [(a, b)] \\
&xs \times ys = \mathbf{do} \\
&\quad x \leftarrow xs \\
&\quad y \leftarrow ys \\
&\quad \mathbf{return}\ (x, y)
\end{aligned}
$$

Example

$$[1, 2] \times [3, 4] \to [(1, 3), (1, 4), (2, 3), (2, 4)]$$

Parsing

Parsec

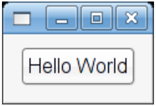
```haskell
perl_variable = do
    sigil  ← oneOf "&$@%"
    name ← many alphaNum
    return (sigil : name)
```

Example

- *parse perl_variable* "Parser" "$var" → *Right* "$var"
- *parse perl_variable* "Parser" "not a var" → *Left* "Parser" (*line* 1, *column* 1) : *unexpected* "n"

GUI - Gtk2Hs

```haskell
main_gui :: IO ()
main_gui = do
  initGUI
  window <- windowNew
  button <- buttonNew
  set window [containerBorderWidth := 10,
              containerChild := button]
  set button [buttonLabel := "Hello World"]
  onClicked button (putStrLn "Hello World")
  onDestroy window mainQuit
  widgetShowAll window
  mainGUI
```



Testing with QuickCheck

$$
\begin{aligned}
&\mathit{property\_factorial1}\ n = \\
&\quad \mathit{factorial}_1\ (n+1)\ \text{`}\mathit{div}\text{`}\ \mathit{factorial}_1\ n \equiv n+1
\end{aligned}
$$

quickCheck property_factorial1
results in

```
*** Exception:  stack overflow
```

$$
\begin{aligned}
&\mathit{property\_factorial2}\ n = n \geqslant 0 ==> \\
&\quad \mathit{factorial}_1\ (n+1)\ \text{`}\mathit{div}\text{`}\ \mathit{factorial}_1\ n \equiv n+1
\end{aligned}
$$

quickCheck property_factorial2
results in

```
OK, passed 100 tests.
```

Some more QuickCheck examples

$$property\_gcd\ n = n \geqslant 0 ==> (n\ `mod`\ (my\_gcd_2\ n\ (n + 2))) \equiv 0$$

Checking only specific values:

$$\begin{aligned}&property\_primes = forAll\ (two\ some\_primes)\ \$\ \lambda(p, q) \rightarrow\\ &\quad (p \equiv q\ ||\ gcd\ p\ q \equiv 1)\\ &\quad \textbf{where}\ some\_primes = elements\ \$\ take\ 200\ primes\end{aligned}$$

Lists can be generated too:

$$\begin{aligned}&property\_reverse\ list = (reverse_1\ .\ reverse_1)\ list \equiv list\\ &property\_quicksort\ list = quicksort\ list \equiv List.sort\ list\end{aligned}$$

What else?

- Implementations: GHC, Hugs, Helium, JHC, YHC
- Parallel GHC, Concurrent GHC, STM
- Cabal
- Visual Haskell, EclipseFP
- Famous Projects Using Haskell: Pugs, Darcs.
- DSLs, DSELs.
- Literate Haskell

Few Implementation Notes

- These notes are based on the article about the "Spineless Tagless G-Machine" by Simon Peyton Jones, which is the basis for current implementations of the Glasgow Haskell Compiler - GHC.
- I'll only speak about some of the basic details, because I have much more to learn ...

The Compiler Structure

1. Preprocessing - Removing the literate markup, if needed, and also running a C preprocessor, if asked by the user.
2. Compiling into the smaller *Core* language, an intermediate language without the syntactic sugar. Type checking is performed, and pattern matching is translated into simple **case** expressions.
3. Some optimizations are performed on the intermediate language.
4. The Core language is translated into the STG language.
5. The STG language is translated by a code generator into C, or into machine code.

We'll focus on the STG language, and how it is translated into C.

The Spineless Tagless G-Machine Language

The STG language is a very austere functional language, or a subset of Haskell.

It contains only the following constructs:

- Function applications, for using functions.
- **let** and λ expressions, for creating new bindings.
- **case** expressions, for evaluating expressions.
- Constructor applications, for defining values.

Translation into STG

Example

$$
\begin{aligned}
&\mathbf{map}\ f\ [\,] &&= [\,] \\
&\mathbf{map}\ f\ (head : tail) &&= (f\ \ head) : (\mathbf{map}\ f\ \ tail)
\end{aligned}
$$

is translated to

$$
\begin{aligned}
&\mathbf{map} = \{\,\}\lambda n\{head, list\} \rightarrow \\
&\quad \mathbf{case}\ list\ \mathbf{of} \\
&\qquad Nil \quad \{\,\} \qquad\qquad\quad \rightarrow Nil\{\,\} \\
&\qquad Cons\{head, tail\} \rightarrow \\
&\qquad\quad \mathbf{let}\ f\_head \ \ = \{f, head\}\lambda u\{\,\} \rightarrow f\{y\} \\
&\qquad\qquad\ map\_tail = \{f, tail\}\ \ \lambda u\{\,\} \rightarrow \mathbf{map}\{f, tail\} \\
&\qquad\quad \mathbf{in}\ Cons\{f\_head, map\_tail\}
\end{aligned}
$$

Memory Representation

Many kinds of values:

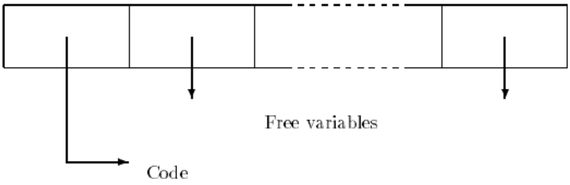
- Functions: $\{free\_list\}\lambda n\{arg\_list\} \rightarrow expr$
  Contain code, and pointers to their free variables.
- Thunks: $\{free\_list\}\lambda u\{\,\} \rightarrow expr$
  Unevaluated expressions, contain the code to evaluate, and any needed pointer.
- Constructors: $Constructor\{arg\_list\}$
  Contain the pointers to the constructors' parameters, which might be functions or thunks themselves.
- Primitive Values:
  Integers, characters, floating point numbers, etc.

Closures

In a polymorphic language, you cannot always know statically if a pointer is a function or a thunk, for example:

$$compose\ f\ g\ x = f\ (g\ x)$$

$g\ x$ might be a function or a thunk, on every call to compose.
It is convenient to hold all values (except the primitives) in memory in the same structure, as closures:



A mapping to ordinary machines

The STG language was defined with operational semantics. Each language construct has an operational meaning:

Construct	Operational meaning
Function application	Tail call
Let expression	Heap allocation
Case expression	Evaluation
Constructor application	Return to continuation

The STG Abstract Machine

The abstract machine which the implementation is based on has:

- Argument stack - a stack for passing parameters to functions.
- Return stack - a stack for continuations.
- Update stack - a stack for update frames (updating thunks).

The machine also includes a heap (garbage collected) for holding closures.

This is only the abstract machine, which is easier to understand. The real implementation has a different representation for these stacks.

Function Application

A function call is implemented by

- Pushing its arguments to the argument stack.
- Tail-calling the function (A jump into the function's code).

Example

$\mathbf{map}\{f, tail\}$

Let expressions

let expressions give local names to closures, and evaluate an expression in the local environment.

They are implemented by:

- Constructing the closures in the heap.
- Evaluating the expression

Example

$$
\begin{aligned}
\textbf{let}\ & f\_head && = \{f, head\}\lambda u\{\} \rightarrow f\{y\} \\
& map\_tail && = \{f, tail\}\ \lambda u\{\} \rightarrow \textbf{map}\{f, tail\} \\
\textbf{in}\ & Cons\{f\_head, map\_tail\}
\end{aligned}
$$

Case expressions

case expressions force evaluation of an expression, and then choose from alternatives based on its value.

They are implemented by:

- Pushing a continuation (or continuations) onto the return stack.
- Evaluate the expression.
- The evaluation is responsible for continuing according to the right alternative.

Example

$$
\begin{array}{l}
\textbf{case}\ \mathit{list}\ \textbf{of} \\
\quad \mathit{Nil} \quad \{\,\} \qquad\qquad\quad \rightarrow \ldots \\
\quad \mathit{Cons}\{\mathit{head}, \mathit{tail}\} \rightarrow \ldots
\end{array}
$$

Constructor Applications

The application of a constructor is evaluated from within some case expression. The implementation:

- Pop the continuation from the return stack.
- Jump to the right alternative.

After return, either:

- a special register points to the constructor's closure, for the inspecting its values, or
- they could be returned in registers directly.

Example

$$\begin{array}{l} \textbf{case}\ \mathit{list}\ \textbf{of} \\ \quad \begin{array}{ll} \mathit{Nil}\ \{\,\} & \rightarrow \mathit{Nil}\{\,\} \\ \mathit{Cons}\{\mathit{head}, \mathit{tail}\} & \rightarrow \textbf{let}\ \ldots \end{array} \end{array}$$

Constructor Applications

Notes

- Returning in registers can avoid allocating a new closure in the heap, and this is why the machine is called spineless.
- The fact that the alternatives can be chosen without holding a tag field for every different constructor is the reason why it is called tagless.

Updating Thunks

In order to update thunks after they are evaluated:

1. When entering an updatable closure
    - An *update frame* is pushed to the update stack, which contain a pointer to the closure to be updated, and the contents of the arguments and return stacks.
    - The return stack and argument stack are made empty.
    - Its sometimes nice to update the closure temporarily with a "black hole" closure.
2. When evaluation of a closure is complete an update is triggered.
    - If the closure is a function, it won't find enough arguments on the argument stack.
    - If the closure is a value, it will attempt to pop a continuation from the return stack, which is empty.
3. The update is either in-place, or by an indirection closure which is removed by GC.

Links

- http://haskell.org
- Learning Haskell
  Audrey Tang
  http://perlcabal.org/~autrijus/osdc/haskell.xul
- The Evolution of a Haskell Programmer
  Fritz Ruehr
  http://www.willamette.edu/~fruehr/haskell/evolution.html
- A history of haskell: being lazy with class.
  http://research.microsoft.com/~simonpj/papers/history-of-haskell/history.pdf

Links
Implementation

- GHC Commentary
  `http://hackage.haskell.org/trac/ghc/wiki/Commentary`
- Implementing lazy functional languages on stock hardware: The spineless tagless g-machine.
  `http://citeseer.ist.psu.edu/peytonjones92implementing.html`.
- GHC Hackathon Videos
  `http://video.google.com/videosearch?q=GHC+Hackathon&so=0`

Thank you!

Thank you!

Questions?

Appendix

An Efficient Reverse

By the way: The previous slide's $reverse_1$ function has $O(n^2)$ complexity, since each $+\!\!+$ operation is linear in the first list's length. A more efficient version is:

$$
\begin{array}{l}
reverse_2 :: [a] \rightarrow [a] \\
reverse_2\ list = helper\ list\ [\,] \\
\quad \textbf{where} \\
\qquad helper\ [\,]\ \qquad\qquad reversed = reversed \\
\qquad helper\ (hd : tail)\ reversed = helper\ tail\ (hd : reversed)
\end{array}
$$

which runs in $O(n)$ complexity.

Back

Monad Class

$$
\begin{array}{lll}
\textbf{class}\ \text{Monad}\ m\ \textbf{where} & & \\
\quad (\ggg\!=) & :: \forall a\ b\ .\ m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b \\
\quad (\gg) & :: \forall a\ b\ .\ m\ a \rightarrow m\ b \rightarrow m\ b \\
\quad \textbf{return} & :: \forall a\quad .\ a \rightarrow m\ a \\
\quad \textbf{fail} & :: \forall a\quad .\ String \rightarrow m\ a
\end{array}
$$

Back