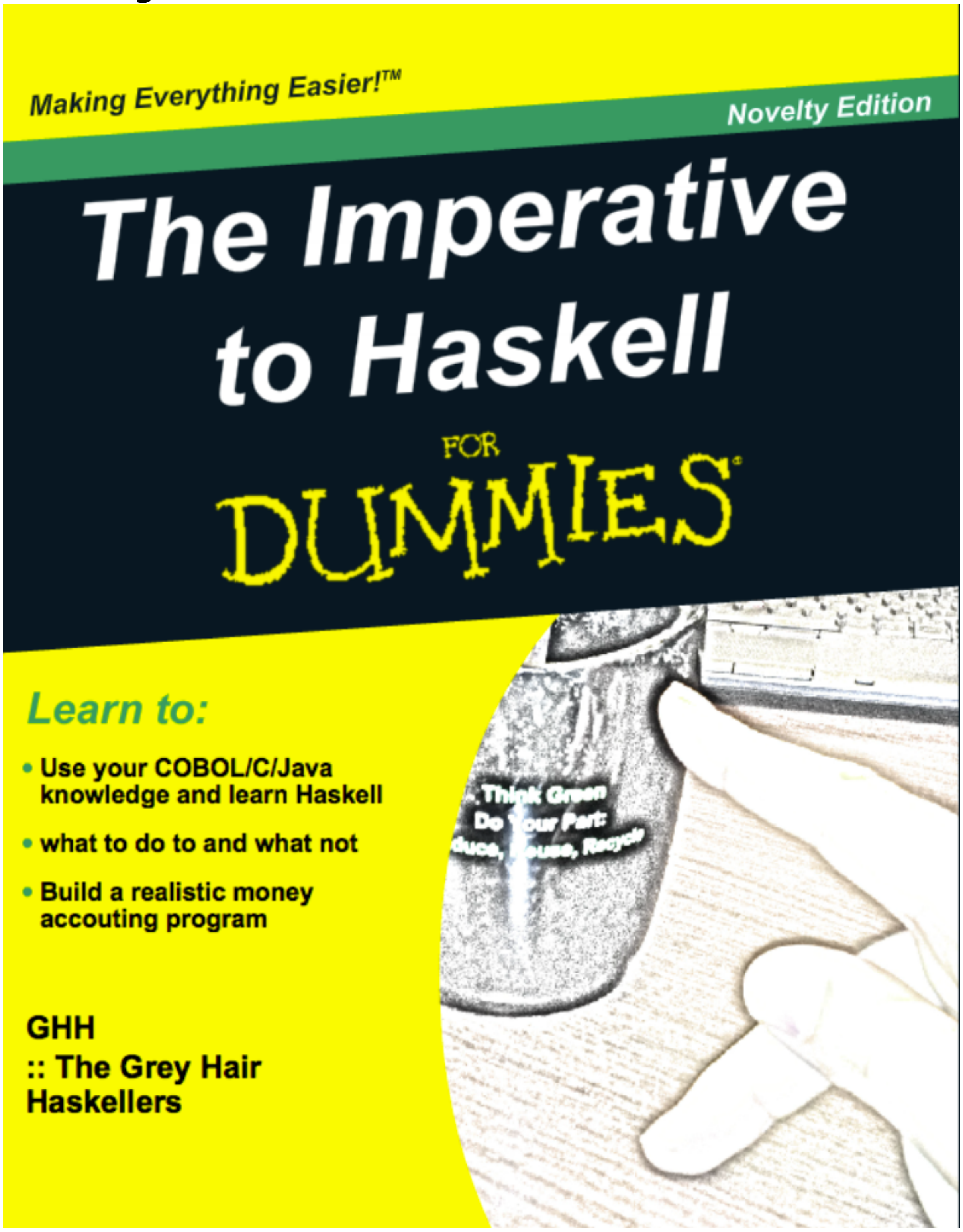
FrontPage



Making Everything Easier!™

Novelty Edition

The Imperative to Haskell

FOR DUMMIES®

Learn to:

- Use your COBOL/C/Java knowledge and learn Haskell
- what to do to and what not
- Build a realistic money accouting program

GHH
:: The Grey Hair Haskellers

FrontMatter

GHH : the Grey Hair Haskellers

GHH : the Grey Hair Haskellers

Contributors & Reviewers:
Tom Schrijvers <tom.schrijvers@ugent.be>
Richard GUITTER <rguitter@gmail.com>
Sebastien.bocq@gmail.com

Bartosz Milewski bartosz@relisoft.com

licence

This work , the imperative to haskell de Luc Taesch, is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/ or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

Introduction

Audience:

Meet the audience of :

- Bob, the cobol programmer. Bob is 50, worked on Mainframe, in the accounting department of a bank, using structured programming. He also knows Pascal, C, Basic.
- Clara is a C++ programmer . Clara is 40, knows OO and design patterns, work in Unix, in the market department of a bank.
- Jo is a Java Programmer. Jo is 30, uses J2EE and agile, and work in the e-commerce application of a bank.

All 3 are *applications* programers, and Bob was also part of the *infrastructure* team at a time where banks were building their own infrastructure (when now they buy COTS)

- Simon is a famous Haskell Priest, who answers the questions of our friends, and collect them because he wants to create a pragmatic tutorial on Haskell, dedicated to innovative Imperative Programmers.

Bob, Clara, and Jo create an Accounting program

We will bring Bob, Clara, and Jo to the point they can do an accounting program in Haskell, and map they current knowledge and practice to Haskell.

To know what to do , what to do not, and how to do it.

After this, we add a bit of spice and cultural background on top; but like with spice, a bit is exciting, too much kills.

How is this book organized

Part1 introduces the syntax and programming styles. We start with How, We use a series of « how can i do this in haskell », based on the way we were taught imperative programming. I cover in order: structured programming, OO concepts, agile and build environment.

The scope is basic haskell and covers the notions known in the imperative world.

Part 2 uses this knowledge and build iteratively a real world example of an accounting program, (inspired by hledger) where we discuss and comment practices, and caveat. We use and introduce some common libraries.

Part 3 introduces « intermediate haskell », i.e. Concepts that do not exists in imperative world.

We revisit our accounting program to use these new concepts.

The fourth part, or annex, is a cultural part. It present a Jargon file of the concepts we saw previously, useful to understand Q&A on stackexchange for instance, including a series of « programming styles » useful when reading Haskell code in libraries, then point at advanced Haskell concepts, just so you know they exists, and are « advanced » (and that you could live well without)

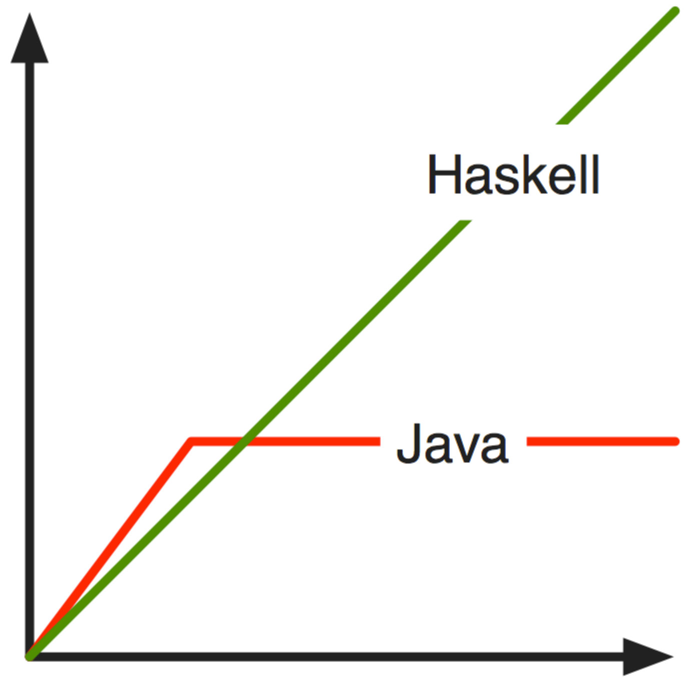
3 things you need to know about haskell

3 things which changes when Haskelling:

big, Massive Learning curve…

- Haskell learning is much vaster than any other language

- But you just need the equivalent of what you know today to get starting « professional » application.



Get a running program:

To Get to a point to the program is running:

- Most Language is 10% compiler syntax fight, 90% debug.

- Haskell is 90% compiler fight, 10% debug (if any)

library

- The library is huge, (1500+) and wild, and open.
- You can get lost, so we will provide a first selection of useful things

Part 1: concepts bob, Clara and Jo already knew, and haskell equivalents

revisiting your experience

This tutorial will start from Bob, Clara and Jo experiences, and show how they can use in Haskell .

Under the form of questions like:
- (How) Can I use xxx in Haskell ?

it answer :
- **Yes**. (…)
- **Yes**, (…) **however** there is a better way, which is …
- **No**, and here is the Haskell way…

3 sections covers :
- Bob experience : structured programming
- Clara experience: Object orientation
- Jo Experience : Agile build environment.

Simon will not discuss performance, size, memory , etc before the Part 2.

BOB and Structured Progamming

Content

If then else
For
Variables
Functions and actions
IO files
Structure, data, records,
case

Bob : when I was taught Basic and then Cobol,

Variables

Bob : when I was taught Basic, C and then Cobol,
My course talked about variables, control structures, data structures, subroutines and File IO, So I will ask along these lines..

can I use numbers and String ?

Yes.

Print (2+2) gives 4, print "hello" gives "hello".

'A' is a Char, and "A" is a String.

There is Int, Float , even Fraction .

```
 print ((11 % 15) * (5 % 3))
11 % 9
```

Note :

- Integer has no limit except the capacity of your machine

```
4^103

102844034832575377634685573909834406561420991602098741459288064
```

can I use lists ?

Yes.

List are the workhorse in Haskell.

[ ] denote a list . (not an Array !)

a String is just a list of Char : String = [Char]

['a','b','c'] = "abc"

some functions on list :

: add an element in front of the list

'a':"bc" = "abc" = ['a','b','c'] = 'a':'b':'c'

++ joins two list together (concatenate)

"ab"++"cd" = "abcd"

head ("abc") = 'a'

head gets the head of the list

tail ("abc") = "bc"

head gets the tail of the list, (i.e. the list minus the head)

"abcde"!!2 = 'c'
!! get the nth item of a list, starting at 0

map (even [1,2,3])= [False,True,False]
map (f alist) = apply the function f on all the element of list alist

[] is the empty list

- List needs no sizing nor allocation, nor deallocation.
- we will look more at list later (tbd)

can I use arrayS ?

yes..

x !! 3 = ..

need no dimensioning, nor allocation, nor deallocation

However…

Array are not as used as in cobol, or C, where they are the workhorse.

can I use variables ?

Yes…

```
let a = 1
let b = 2
print (a+b)
3
```

```
let hi = "hello"
let w = " world"
print (hi++w)
"hello world"
```

However...

- you cannot change a variable !! (mutate the value)

this is the probably the most surprising change from imperative. We will see we can perfectly live with this, what it changes in the practice in..(tbd), and we will discuss the reason at length in Section 4,

Lets just accept it for now, until we have enough material to show meaningful example (at the end of this section)

Note:
- the scope of the variable is local. we will see how to simulate "global" in ..(tbd)
- naming : as long as you want,
    any ascii char, small and Upper case , but no Uppercase for first letter (reserved for types)

can I use if then else ?

Yes..

```
if (a==1) then True else False
```

Note: else is not optional

block and layout ?

can I use the For / while loop ?

Yes.

for ("abc" uppercase) = "ABC"
for ([1,2,3] even) = [False, True, False]

for (list function) iterates a function over the list , and generates a list of results.

However...

the canonical way in Haskell is map.
for is just another way to say map with the arguments flipped.

for ("abc" uppercase) = map (uppercase "abc") = "ABC"

for (alist f) = map (f alist)

can I use recursion ?

Yes.
lets use a classical , the factorial
reminder : 5! = 5x4x3X2x1 = 120

in C:
using a for loop :

```
long factorial(int n)
{
  int c;
  long result = 1;
  for (c = 1; c <= n; c++)
    result = result * c;

  return result;
}
```

using recursion :

```
long factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return(n * factorial(n-1));
}
```

in Haskell, using recursion :

```
fac (n) = if ( n == 0 ) then 1 else n*fac(n-1)
```

can I work without mutating variable

(demonstration of map and fold on recursion):

BOB: the previous statement that one cannot mutate variable puzzled me! I cannot imagine working without ..

for instance, in a loop to calculate the length of a String:

in C:

```
int calculate_length(char *string) {
  int length = 0;
  while (string[length] != '\0') {
      length++;
  }
  return length;
}
```

trying to do something similar in haskell would be:

```
mylength s = do  { let l =0
                   for (s)
```

```
          l = l+1
        return ( l)
        }
```

because as mentioned, one cannot mutate a variable, and l is mutated in l=l+1

Simon :
well, we could use recursion instead of loops:

```
mylength counter s =
     if (s==[])
     then counter
     else  mylength (1 + counter) (tail(s))



main = print (  mylength 0 "how long?" )
```

mylength is recursing by removing one character (the head) at a time, and adding one to a counter, until the string is empty, in which case it terminates and results the counter.

```
 "abc" 0
"bc" 1
"c" 2
"" 3
```

this technique is very common. it is called a **fold**, and do a computation that "accumulate" a state (the length) when traverse a structure (the list).
(Cross domain example: a SQL count (or sum) traverse a table and accumulate a count (or sum))

Note we did not needed to mutate the counter to accumulate its value. we passed it by parameter to the next call.

(questions about performance ? we will see in (tbd))

can I define function/procedure/subroutines ?

Yes.

in C :

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

in Haskell :

```
f x y = x*x + y*y
```

However,
there are two categories: **functions** and **actions**.

```
f x = ((g (x ))*(h (x) ))/ 2
```

we do not care if the compiler evaluates g or h first, as long it comes with a good result. this is a **function**. a function always gives the same result (given the same parameters).

now consider `getChar` that reads a character from StdIn.
it will NOT give the same result at every call, as it depends on what is comes in from StdIn. this is an **action**. Note that the sequence matters for actions, like opening a file *first*, (do something with it), *then* close the file. we *do* care about the

sequence for actions.

```
the syntax to specify the sequence is do {actions1;
actions2..}

    copyCharUnlessNL  =  do { c <- getChar ;
                   if ( c=='/n')
                      then { putChar ''}
                      else { putChar c}
                  }
  // copy a char from stdio to stdout, unless is is a
NewLine
```

The keyword do introduces a sequence of statements which are executed in order. A statement is either an action, or the assignment of the result of an action using <-.

we will come back to this distinction, as this is central to haskell and is a point where many could fails

can I reduce the number of parentheses, curly braces

yes.

Lets introduce three techniques to have a minimalist reading :

- layout
- removing parentheses and $
- pattern matching and guards.

curly braces, semicolon and parentheses are optional, unless ambiguity need to be raised.

curly braces are used to delimitate a block. layout can be used instead.

```
myaction  = do {
              print "hello";
              print "world/n";
            }
myaction  = do
              print "hello"
              print "world/n"
```

this would not compile:

```
myaction  = do
              print "hello"
       print "world/n"
```

beware : tabulation does not work. they count as one character. have your text editor substitute spaces instead

next thing is parentheses.

parentheses can quickly accumulate and become a pain to read. they are optional:

```
f ( g ( h (x))) = f g h x
```

..unless they are not :

print map f [1..3] will break the compile ("The function `print' is applied to three argument..") . Use instead :

```
print ( map f [1..3] )
```

or better

print $ map f [1..3]

$ avoid putting parenthesis at the end, which is convenient.
"Anything appearing after it will take precedence over anything that comes before"

```
print ( uppercase ( "aa" ++ "bb")) =
print $ uppercase ( "aa" ++ "bb") =
print $ uppercase $ "aa" ++ "bb"
```

Pattern Matching and guards.

Pattern Matching .

remember the factorial definition ?

```
fac n = if ( n == 0 ) then 1 else n*fac(n-1)
```

it could also be written like :

```
fac 0 =  1
fac n =  n*fac(n-1)
```

the parameters will be tried to be matched on the first equation, and if failed, the next one is tried; (which is what (if (n ==0) then .. else) is doing…)
if no match is found, an error is raised. _ is a wildcard, and can be use to catch anything.

can pattern matching be used on lists ?
Yes

Pattern Matching work not only for numerical , lets try on lists :

Lets try an example of a loop.
we iterate over `['a','b',c']` `to uppercase each letter.`

1/ with brackets and curly braces:

```
toUppercase (  l ) = { if ( l ==[])

                         then  []

                         else { uppercase (head(l ):
toUppercase (tail (l))}

}
```

2/lets remove the brackets and curly braces

```
toUppercase   l  =  if ( l ==[])

                         then  []

                         else  uppercase $ head l :
toUppercase $ tail l
```

3/lets use pattern matching

```
toUppercase   []  =  []

toUppercase x:xs =  uppercase x : toUppercase xs
```

`x:xs matches the head and the tail of a list. names are free, you can choose to name it y:z, or peer:apple, what matters is the matching to the : operator, which makes it matches to head and tail of a list.`

```
aList = head(aList):tail(aList)
```

`BOB: this looks neat, so is it how I do most of my loops ?`

```
Simon: well, Indeed, and however I got a trick for you…
```

Can I generalize this loop stuff

Bob : Can I generalize this loop stuff ?

Simon : look at map

```
map f  []               =  []
map f (x:xs)            =  f x : map f xs

map uppercase ['a','b',c'] =

uppercase a : map uppercase [ 'b', ' c']=

'A' : uppercase 'b' : map uppercase 'c' = 'A':'B':
uppercase 'c': map uppercase []

= 'A':'B':'C':[] = "ABC"
```

so we could use map to define the previous function toUppercase

```
toUppercase l = map uppercase l
```

```
Bob: wow, this is going to be hard to do more terse…

Simon : indeed
```
[Luc Taesch, 22/05/13 09:35 too much , maybe worth a split ?]

map is doing all the recursion loop stuff for us. note that we are passing a function as as argument (`uppercase`). this is possible in haskell.

map is one of the workhorse in haskell, the 'for' loop of haskell when you need to do something with each element of a list. if you need to carry something across, like when you count , or sum, you use fold, the other workhorse.

```
fold f accu [] = accu

fold f accu x:xs = f x ( fold f accu xs)
```

Let s try to sum a list, with (+) a b = a + b .

(parentheses needed when to use the + function as prefix) .

```
fold (+)  0 [1,2,3] =  fold (+) ( (+) 0 1)      [2,3]

= fold (+) 1 [2,3] = fold (+) ((+) 1  2) [3] = fold (+) (3) [3]

= fold ((+) 3 3) [] = fold 6 [] = 6
```

can I use File ?

can i do

can i do

```
for (i=1; i<10, I++) {print i}

print $ [1..10]
```

types

Clara and Object Orientation

Content

Types, class,
 currying ?
infix operators
pattern matching
gaurds
 sign

Untitled

Jo and Agile Builds

content

Ghc make, runhaskell,
Cabal
Cabal-dev
Hackage, stackage
Unit tests
Haddock
Refactoring
Understanding ghc messages.
module

Hpc >= Intermediate
Hlint >= Intermediate
QCheck >= Intermediate
Template Haskell >= Advanced

Part 2 : A ledger program

getting practical

In this section we get practical,

Fitst, we talk about the problems specific to haskell, which may hinder Bob, Clara and Jo when starting.

Then we compile a few programs, refactor them from an imperative style to a more haskellish way, in order to master tools and techniques, and see what you do when the compiler complains.

Then we take a more ambitious program, the base for our ledger program. this provides the opportunity to put together the techniques we saw, plus a few new ones , and get acquainted with useful libraries.

what May hurts you when you use haskell

Ignorance generates pain and suffering when you start. but especially , ignorance linked to assumptions that are not adequate any more when haskelling.

in most other languages, to get to a running program, is 10% compiler and syntax fight, and 90% debug . In Haskell, this is the opposite.

Most of the time, this is not seen, and it hurts because we have the wrong assumptions.

Knowing this is key not to lose the spirit when starting.

why is that, first ?

a C compiler has a simple system and when you do things too complex for it to understand, you cast or revert to pointers. If you are wrong, you will see it at debug time. If you want to divide a boolean by a number , it is up to you.

in Haskell, the type systems is much more sophisticated so you can express complex things. There is no such things as a cast. That means no possible cheating when getting nervous, also… You do not even need to spell the type, the compiler is inferring the types for you and verifies them.

This is why most of the bugs are caught at compile time. and also why the compiler phase is longer.

Now, this is a blessing, but this is also a curse when beginning. because you cant seem to see the end of it.

two things makes it even more painful:

- the compiler gives messages which are weird at first. once you know how to read them, it gets much better. we will do some mistakes on purpose to get acquainted.

- the mix and match of functions and actions breaks the type system. because this separation of concern is not present in other language, it is a new skills we need to work out, we will do it when refactoring. just putting a print in the middle of a function for debug purpose can lead you to Monad Hell.

Part 3 : concepts Bob, Clara and Jo did not knew (Intermediate)

Content

lazyness,
polymorphic types
list comprehension
patterns
guards
lamba function
point
point free
composition
infinite structure

Part 4: To the infinite and beyond (Advanced)