hack.hands( )
A PLURALSIGHT COMPANY

BECOME AN EXPERT SIGN IN

hack.hands() is an online service for live programing help available 24/7.

GET STARTED

Posted on March 9th 2015

Haskell's Non-Strict Semantics – What Exactly does Lazy Evaluation Calculate?

This tutorial is part of a larger series, The Incomplete Guide to Lazy Evaluation (in Haskell).

Haskell's Non-Strict Semantics – What Exactly does Lazy Evaluation Calculate?

Lazy evaluation is the most widely used method for executing Haskell program code on a computer. In previous tutorials, we have looked at how lazy evaluation works in detail and how it can help with making our code simpler and more modular. However, the main drawback of lazy evaluation is that it can be quite hard to understand all details of how exactly the program is evaluated. How, then, we can understand what the program does in the first place?

In this tutorial, I want to explain how we can understand a Haskell program without having a clue about lazy evaluation at all. This is possible thanks to Haskell's **non-strict semantics**. First, we will introduce the concept of **denotational semantics** and the special value "$\perp$", which is called “**bottom**”. Next, we will look at the concepts of **strictness** and **non-strictness**, which often show up in discussions about the optimization of Haskell programs. Afterwards, we will learn how to understand infinite lists without using lazy evaluation. Finally, we will consider the illuminating example of the children’s game “rock-paper-scissors” – which I learned from the classic book An Introduction to Functional Programming by R. Bird and P. Wadler – to see how this method works in practice.

The key message is that there are different levels at which we can understand a program. Lazy evaluation tells us ***how*** a Haskell program is executed, which is very useful for understanding its time and memory usage. However, we are also interested in ***what*** a program calculates, which is important for showing that the program is correct and free of bugs. In fact, I would say that the latter is more important than the former – there is no point in calculating a wrong result very fast. In most traditional imperative languages, the *what* and the *how* are very similar, but in Haskell, these are two different and complementary viewpoints. Often, the *what* part is easier to understand than the *how* part, because for the former, we may replace any expression by one that is *equal* to it, whereas for the latter, no two expressions are equal, their evaluation always proceeds in slightly different ways. In a way, the idea of separating the *how* from the *what* is what purely functional programming is all about.

The *how* of a program also goes under the monicker of **operational semantics**, whereas the *what* is also called **denotational semantics**. The word **semantics** simply refers to the “meaning” of a program. In this tutorial, we are concerned with the denotational semantics of Haskell and how it relates to operational semantics, i.e. lazy evaluation.

Expressions, their Denotations and Bottom

Consider any Haskell expression, for example

```
40 + 2
```

The meaning or **denotation** of this expression is the number *42*. The expressions

```
42
2*21
sum [1..8] + 6
```

also **denote** the same value, *42*. Note that we carefully distinguish between an *expression*, which is a syntactically legal piece of Haskell code, written in typewriter face, and a *value*, which is a mathematical object, written in italics. So, the expression `42` denotes the value *42*, but these are different things. This may seem like hair-splitting, but it’s a very useful distinction to keep in mind. The point is that different expressions, like `42`, `40+2` and `2*21`, may denote the same thing, *42*. In this case, we write an *equal* sign

```
_42 = 40+2 = 2*21_
```

to indicate that while the expressions are different, they denote the same value.

The rule is that two expressions denote the same thing if they evaluate to the same normal form. When running a Haskell program, the normal form is the final result that we care about. It represents *what* is being calculated.

However, the beautiful thing about denotational semantics is that we can also give meaning to expressions that do *not* have a normal form, for instance because their evaluation goes into an infinite loop. For example, evaluation of the expression

```
let loop = loop + 1 in loop
```

will never terminate. What value do we assign to it? The solution is very simple: We introduce a new value written " $\bot$ " and called "**bottom**", which represents the value of a computation that, well, "does not have a value", for instance because it goes to an infinite loop or because it throws an exception. In our example, the expression `loop` denotes the value $\bot$, because its evaluation never terminates. Another example would be the expression

```
head []
```

which also denotes the value $\bot$ (albeit of a different type), because the head of an empty list is undefined and trying to evaluate this expression will throw an exception. In fact, Haskell has a predefined expression

```
undefined
```

which always throws an exception and thus denotes the value $\bot$.

For each type, we can make a list of possible semantic values. For instance, an expression of type `Bool` may denote one of the values $\bot$, `_False_` or `_True_`. An expression of type `Integer` may denote one of the values $\bot$, `_1_`, `_2_`, ... and so on. Essentially, we just have to add the extra value $\bot$ to each type, at least for simple types like `Bool` and `Integer`. Later, however, we will see that more complex types, like the list type `[Int]` will contain additional extra values.

Strict Functions and Non-Strict Semantics

Once again, let us consider the example of the logical AND, whose definition is

```
(&&) :: Bool -> Bool -> Bool
True  && x = x
False && x = False
```

To understand *what* this function does, we can write down a truth table. However, this time, we also have to consider the new value ⊥, which indicates that an argument does not terminate. So, the truth table we consider here has not just 4, but 9 entries. The first rows of the table are:

```
⊥ && ⊥     = ⊥
⊥ && True  = ⊥
⊥ && False = ⊥
```

The reasoning is that since the function performs a pattern match on the first argument, the function will not terminate if the evaluation of the first argument does not terminate. The next rows are

```
True && ⊥     = ⊥
True && True  = True
True && False = False
```

These simply mimic the first equation in the definition of the function. The final rows of the truth table are the most interesting:

```
False && ⊥     = False
False && True  = False
False && False = False
```

These rows mimic the second equation of the function definition. Note that thanks to lazy evaluation, the function returns the result `False` in all cases, even when the second argument is ⊥. In contrast, in a language with eager evaluation, like LISP or O'Caml, the first line would have to read "`False && ⊥ = ⊥`", because arguments are always evaluated first, and evaluation of the function application does not terminate if evaluation of the argument does not terminate. We say that these languages have **strict semantics**, whereas Haskell has **non-strict semantics**.

(A word of caution: The equations above look like we might be able to "pattern match" on the value ⊥. But this is not possible because ⊥ is not a legal Haskell *expression*; it only exists in the denotational semantics. Remember: we write expressions in typeface, but values in italics.)

Where do the monickers *strict* and *non-strict* come from? Here is the main definition: A function `f` is called **strict** if it satisfies

```
f ⊥ = ⊥
```

In other words, evaluation of the function does not terminate if evaluation of the argument does not terminate. In most cases, this is because the function performs a pattern match on the argument. For a function `g` with, say, two arguments, we say that it is strict in its second argument whenever `g x ⊥ = ⊥` for any first argument `x`; likewise for other positions.

Now, in Haskell, the function `(&&)` is *not* strict in its second argument, because

```
False && ⊥ = False  ≠  ⊥
```

We also say that it is **lazy** in its second argument. Haskell is said to have *non-strict semantics* because it allows such functions, whereas in a language with *strict semantics*, all functions are strict.

The fact that this function is not strict also means that the function does not always evaluate its second argument, because otherwise, well, the result would have to be ⊥. This is the connection between lazy evaluation, the *how*, and non-strict denotational semantics, the *what*. The latter only cares about end results, though, and is thus easier to work with. In fact, the Haskell standard only specifies that a compiler needs to respect the non-strict semantics, it does not say that Haskell needs to be evaluated using lazy evaluation. In other words, Haskell compilers are free to employ some other means, but in practice, the Glasgow Haskell Compiler (GHC) does use lazy evaluation (with some minor adjustments). Clearly, this is important for understanding time and space usage of a program compiled with GHC.

Even though they are part of denotational semantics, the notions of strict functions and lazy functions are often used to discuss matters from operational semantics. For instance, bang patterns are an extension of the Haskell language that allow you to specify that some function arguments are to be evaluated to weak head normal form (see the previous tutorial on how lazy evaluation works for more on this normal form) before the function body is evaluated, mimicking eager evaluation. If you look at the GHC user manual, you will find that this is often described in terms of “strict” and “lazy”. For instance, it says that the bang pattern in the function definition

```
f1 !x = True
```

makes the function strict, so that we now have `_f1 ⊥ = ⊥_` instead of `_f1 ⊥ = True_`. No statement in the manual is incorrect, but I would like to caution the reader again that strictness is a concept from denotational semantics, whereas bang patterns affect the operational semantics. In informal discussions, the word "strict" is often used to mean that a function evaluates its argument to WHNF before proceeding, but "strictly speaking", this is not a correct use of the word "strict".

Another point where you may encounter the notion of strictness is an optimization pass called strictness analysis. Here, the idea is that the compiler tries to figure out which functions are strict, and compiles them to use eager evaluation instead of lazy evaluation. This is safe to do, because it does not change their denotational semantics. The reason for doing so is that lazy evaluation does incur a certain administrative overhead which can be avoided by using eager evaluation; the code will run faster.

Partial and Infinite Lists

We have seen that simple types like `Integer` contain both the ordinary values and the special value ⊥. In the case of more complex types, however, the value ⊥ can appear in many more places. For example: What value does the expression

```
2 : undefined
```

have? It appears to be a list whose head element is `_2_`, but whose tail is undefined. And that's indeed the case, its value is

```
2 : ⊥
```

The point is that the list *constructor* is *not* strict, so the "totally undefined" list ⊥ is different from the "partially defined list" `2 : ⊥`. For instance, if we apply the function `head`, we get

```
head ⊥        = ⊥
head (2 : ⊥) = 2
```

In Haskell, all constructors are lazy (unless indicated otherwise by a strictness annotation).

We have already discussed the usefulness of infinite lists, but so far, we could only explain how lazy evaluation allows us to write down expressions for "potentially infinite" lists. Now, denotational semantics, and the just introduced concept of partial lists, allows us to talk about *actually infinite* lists. The idea is that an infinite list is to be understood as a *limit* of partial lists. Just like the number *π=3.1415…* is mathematically a limit of successive approximations *3*, *3.1*, *3.14* and so on, an infinite list, like the list of all positive integers,

```
1 : 2 : 3 : …
```

is a limit of partial lists

```
⊥
1 : ⊥
1 : 2 : ⊥
1 : 2 : 3 : ⊥
…
```

and so on. To see how this works, consider the expression

```
filter (< 3) [1..]
```

To understand the denotation of this expression, we apply the function *filter* to the partial lists that approximate the infinite argument list:

```
filter (< 3) ⊥                      = ⊥
filter (< 3) (1 : ⊥)                = 1 : ⊥
filter (< 3) (1 : 2 : ⊥)            = 1 : 2 : ⊥
filter (< 3) (1 : 2 : 3 : ⊥)        = 1 : 2 : ⊥
filter (< 3) (1 : 2 : 3 : 4 : ⊥)    = 1 : 2 : ⊥
…
```

All this has been calculated by applying the definition of the function `filter`. As you can see, the results stabilize after the third approximation, because there are no other positive integers smaller than the number 3. Hence, the limit is a partial list

```
filter (< 3) [1..]  = 1 : 2 : ⊥
```

And indeed, if you fire up a Haskell interpreter like GHCi and try to evaluate this expression, the interpreter will first output

```
> filter (< 3) [1..]
[1,2
```

and then not respond to inputs anymore; this is what happens when you try to print the partial list `1 : 2 : ⊥`.

The main message here is that using partial lists, we can understand infinite lists in a way that does not involve lazy evaluation. We had already noted in our previous discussion of infinite lists that the latter adds many additional complications, like the need to distinguish between the expressions `(0+1)` and `1`, and so on. Working on the level of denotational semantics allows us to understand infinite lists in a more practical and intuitive way as actually infinite.

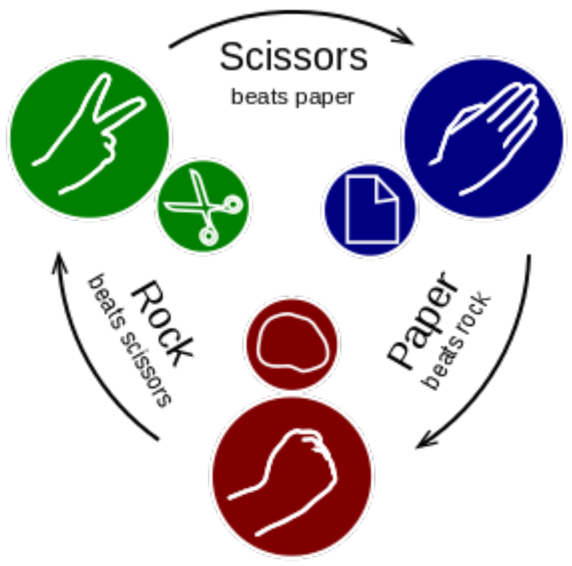
Of course, the denotational semantics do mimic the reductions steps taken by lazy evaluation to some extend. In particular, the value $\perp$ does feel a bit like an "unevaluated" expression, as it gets refined from $\perp$ to `1 : ⊥` to `1 : 2 : ⊥` and so on. It is a good idea to update our intuition of the value $\perp$, and to think of it as a "computation that has not finished yet". Sometimes, this means that the computation may never finish, because it goes into an infinite loop. At other times, it is better to imagine that it may yield more information after some time, for instance when the list $\perp$ is refined to the list `1 : ⊥`. From this point of view, a strict function is a function that cannot reveal any information about its result before it has gained more information about its argument.

Rock-Paper-Scissors

To gain more practice with infinite lists, we will now consider the children's game "rock-paper-scissors", a classic example which I learned from the book An Introduction to Functional Programming By R. Bird and P. Wadler.

You probably know the rules of the game. In every round, each player makes one hand sign: either "rock", "paper" or "scissors". We represent this as an algebraic data type

```
data Sign = Rock | Paper | Scissors
```



(Image CC-BY-SA from Wikipedia)

Each sign is beaten by another sign

```
beat :: Sign -> Sign
beat Rock     = Paper
beat Paper    = Scissors
beat Scissors = Rock
```

and we give a point to the player whose sign beats the other player's sign

```
points :: (Sign, Sign) -> (Int,Int)
points (x,y)
    | x == y      = (0,0)   -- draw
    | x == beat y = (1,0)
    | y == beat x = (0,1)
```

Now, we assume that each player's strategy is represented by a type `Strategy` and that there is a function

```
rounds :: (Strategy, Strategy) -> [(Sign, Sign)]
```

which takes two strategies and returns an infinite list of rounds where they play against each other. Then, we can use the following function to play a match of `n` rounds and tally the points:

```
match :: Int -> (Strategy, Strategy) -> (Int, Int)
match n =
    pair sum . unzip . map points . take n . rounds
    where
    pair f (x,y) = (f x, f y)
```

Strategies 1

What is a strategy? It is a method used by one player to decide which sign to play, based on all the previous moves by the other player. We can represent this by a function type

```
type Strategy = [Sign] -> Sign
```

Given a list of the opponent's previous moves, this function calculates the player's current move. For instance, the strategy that always plays "paper" is represented as

```
alwaysPaper :: Strategy
alwaysPaper signs = Paper
```

Another example is the strategy that plays "paper" first, and then plays the move that would have beaten the opponent in the previous round:

```
whatYouDid :: Strategy
whatYouDid []    = Paper
whatYouDid signs = beat (last signs)
```

Finally, we need to write the function `rounds`, which plays two strategies against each other and generates an infinite lists of sign pairs. We can use the function iterate to repeatedly append a new move to the list of previous moves

```
rounds :: (Strategy, Strategy) -> [(Sign, Sign)]
rounds (player1, player2) =
    map lastOfPair $ tail $ iterate update ([],[])
    where
    update (signs1, signs2) =
        ( signs1 ++ [player1 signs2]
        , signs2 ++ [player2 signs1] )
    lastOfPair (x,y) = (last x, last y)
```

Unfortunately, this implementation of the `Strategy` type has a serious drawback: Usually, a strategy needs *O(n)* time to analyze the previous *n* moves. For example, this is the case for the strategy `whatYouDid`. Then, to calculate the first *n* moves with this strategy, the function `rounds` will need $1 + 2 + ... + n = O(n^2)$ time. In other words, it takes quadratic time to calculate a match of *n* rounds. Certainly, we can do better?

Strategies 2

For a more efficient implementation of the `Strategy` type, consider the following idea: Instead of calculating just a single move from a list of the opponent's previous moves, we calculate the infinite list of all moves from the infinite list of all of the opponent's moves.

```
type Strategy = [Sign] -> [Sign]
```

For instance, the strategy `alwaysPaper` ignores the opponent's moves and just returns an infinite lists consisting entirely of the sign `Paper`

```
alwaysPaper :: Strategy
alwaysPaper signs = repeat Paper
```

The strategy `whatYouDid` first returns `Paper` and then applies the function `beat` to all moves of the opponent

```
whatYouDid :: Strategy
whatYouDid signs = Paper : map beat signs
```

The key point is that now, this strategy only needs *O(1)* time to calculate the next move, instead of the *O(n)* time needed to inspect a list of the *n* previous moves.

The implementation of the function `rounds` that plays one strategy against the other is particularly interesting:

```
rounds :: (Strategy, Strategy) -> [(Sign, Sign)]
rounds (player1, player2) = zip signs1 signs2
    where
    signs1 = player1 signs2
    signs2 = player2 signs1
```

The value `signs1` is the infinite list of moves of the first player. Note that it depends on the list `signs2`, which in turn depends on the list `signs1` again. This is also known as **mutual recursion**. How does it work? For concreteness, let us consider the example where the strategy `alwaysPaper` plays against the strategy `whatYouDid`.

The rule is that for any (mutually) recursive definition, we start with the approximation

```
_signs1 = ⊥
signs2 = ⊥
```

which means that we know nothing about the results yet. Then, we calculate the right-hand side under these assumptions. The result for the list `signs2` is

```
signs2 = whatYouDid signs1
       = whatYouDid ⊥
       = Paper : map beat ⊥
       = Paper : ⊥
```

(Remember that we consider the example where `player2` is `whatYouDid`.) For the list `signs1`, we get

```
signs1 = alwaysPaper signs2
       = alwaysPaper ⊥
       = Paper : Paper : Paper : …
```

With this calculation, our knowledge about the result is refined to

```
signs1 = Paper : Paper : Paper : …
signs2 = Paper : ⊥
```

This is our second approximation to the end result. Note that the result for the first strategy is already complete. To get an even better approximation, we once again calculate the right-hand side, but this time using the new approximation:

```
signs2 = whatYouDid signs1
       = whatYouDid (Paper : Paper : …)
       = Paper : map beat (Paper : Paper : …)
       = Paper : Scissors : Scissors : …
```

Apparently, our new approximation is already the complete result

```
signs1 = Paper : Paper    : Paper    : …
signs2 = Paper : Scissors : Scissors : …
```

Hence, the denotation of the expression `rounds (alwaysPaper, whatYouDid)` is

```
rounds (alwaysPaper, whatYouDid)
    = (Paper, Paper) : (Paper, Scissors)
    : (Paper, Scissors) : …
```

To summarize, this example shows how we can calculate the result of a lazy computation without using lazy evaluation. The main advantage is that instead of having to track the *how* of the computation in detail, we can concentrate on the *what*, which means that we can treat different expressions as *equal*, as long as they have the same denotation, something we cannot do when considering how they are evaluated.

Cheating

The efficient implementation of the `Strategy` type has a serious drawback: It allows strategies that can *cheat*. For instance, the strategy

```
cheat :: Strategy
chat signs = map beat signs
```

plays exactly the move that would beat the opponent's move in each round. This is now possible, because the list `signs` contains *all* moves of the opponent, not just the previous ones.

If we play the strategy `cheat` against the strategy `whatYouDid`, we will get the following sequence of approximations:

```
signs1 = ⊥
signs2 = ⊥
```

and then

```
signs1 = cheat      ⊥ = ⊥
signs2 = whatYouDid ⊥ = Paper : ⊥
```

and then

```
signs1 = cheat      (Paper : ⊥) = Scissors : ⊥
signs2 = whatYouDid ⊥           = Paper    : ⊥
```

and then

```
signs1 = cheat      (Paper    : ⊥) = Scissors : ⊥
signs2 = whatYouDid (Scissors : ⊥) = Paper : Rock : ⊥
```

and so on, with the end result being

```
signs1 = Scissors : Paper : Rock     : …
signs2 = Paper    : Rock  : Scissors : …
```

The strategy `cheat` wins every round, because it "knows" all the moves of the opponent.

Of course, the cheating strategy will fail when `cheat` plays against itself. In this case, the computation will go into an infinite loop and the result will be `signs1 = ⊥` and `signs2 = ⊥`.

(Exercise: Show this!)

How can we ensure that a strategy does not cheat? First, we have to think about what exactly cheating is. A strategy does not cheat if it does not “look into the future”, i.e. if it produces the *n*-th hand sign only by looking at the first *n-1* hand signs of the input list. In other words, a strategy `player` is **fair** if it satisfies the equation

```
last (take n (player signs)) =
    last (player (take (n-1) signs
```

for all possible lists `sign`. The strategy `cheat` is not fair, as the following example shows

```
last (take 1 (cheat  (Paper : ⊥))) = Scissors
last (cheat  (take 0 (Paper : ⊥))) = ⊥
```

Unfortunately, there is no way to automatically check whether a strategy is fair, because we can’t test an infinite number of rounds and also because we cannot even check whether a strategy will terminate at all. The best we can do is to let strategies play as long as they behave fairly, but return ⊥ as soon as one of them starts to cheat. The following function is very helpful for that:

```
sync :: [a] -> [b] -> [a]
sync []     []     = []
sync (x:xs) (_:ys) = x : sync xs ys
```

The idea is that `sync` essentially returns its first argument, but when the second argument is a partial list, then only a corresponding part of the first argument is returned. Example:

sync (Paper : Scissors : ⊥) ('a' : ⊥) = Paper : ⊥

Using this helper function, we can implement the function `rounds` as

```
rounds :: (Strategy, Strategy) -> [(Sign, Sign)]
rounds (player1, player2) = zip signs1 signs2
    where
    signs1 = player1 (sync signs2 signs1)
    signs2 = player2 (sync signs1 signs2)
```

By “synchronizing” the outputs of the two strategies, each strategy may only inspect as many moves of the opponents as the strategy itself has already decided upon. Any attempt to cheat will result in ⊥ for the next move.

With this, we come to the end of today's tutorial. I hope the "rock-paper-scissors" example was as helpful to you as it was helpful to me back when I first learned about lazy evaluation and infinite lists.

Written by

Heinrich Apfelmus
Europe/Berlin

Haskell programmer for more than 10 years. I maintain several open source libraries, currently focusing on graphical user interfaces (GUI) and functional reactive programming (FRP). Background in mathematics, physics and theoretical computer science. Besides teaching in person, I also try to pass on what I learned in the form of tutorials and blog posts. You can find them either here on HackHands, or on my personal website.

REQUEST EXPERT

$2 / min

3 Comments HackHands 1 Login

Recommend Share Sort by Best

Join the discussion…

EzequielAlvarez • a year ago

Very nice, thanks for this!
The evaluation steps are vital in order to understand this subject.

• Reply • Share ›

Dmitriy • a year ago

Hi. Thanks for great article, that was first time, when i understand something about denotational semantics.

And i want to point out, that in first variant of rounds `map last $` does not type-check. May be something like this will:

map (\(x, y) -> (last x, last y)) $ tail $ iterate update ([],[])

• Reply • Share ›

Heinrich Apfelmus → Dmitriy • a year ago

Glad you like it. And you're right, thanks a lot!

• Reply • Share ›

Subscribe Add Disqus to your site Add Disqus Add Privacy

Get on the mailing list!

The latest major updates, and nothing else.

your name

you@domain.com

Favorite technologies

SUBSCRIBE

Get live help for these popular subjects plus many more!

Java Node.js Android C# & .NET PHP

Ember.js iOS Javascript Angular.js Golang

Meteor.js SQL Ruby on Rails HTML & CSS CoffeeScript

Haskell Python & Django Scala

Become an expert **Login** **Posts** **FAQ** **Jobs** **How it works** **Success stories**
Contact us

Latest Tweets - @hackhands

Learn how to compare unrelated types of objects in #python with Luke Lee | via @pluralsight |
https://t.co/CtLMVh0HcF

---

hack.hands( )

© 2015 HackHands Inc. All rights reserved.
Terms of use | Privacy Policy