Monad P3 : Non-terminating Expressions (1F)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Non-terminating Expressions

Denotational semantics

Semantics is about defining the "**meaning**" of a **program**.

denotational semantics In Haskell

- the **value** is a mathematical object of some sort

the **expression 10** (but also the **expression 9 + 1**)
have **denotations** of the **number 10**
(rather than the Haskell **value 10**).
We usually write that **⟦9 + 1⟧ = 10** meaning that
the **denotation** of the Haskell **expression 9 + 1**
is the **number 10**.

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

Semantic map and Strachey brackets

Haskell **expressions** denote **mathematical values**.

Strachey brackets ⟦·⟧

to denote the "**semantic mapping**"

from **Haskell** to **Math**.

we want our **semantic brackets** to be compatible

with **semantic operations**.

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

Semantic map example

⟦x + y⟧ = ⟦x⟧ + ⟦y⟧

on the left side + is the Haskell function

(+) :: Num a => a -> a -> a

and on the right side it's the binary operation

in a **commutative group**.

we can use the properties from the **semantic map**

to know how our Haskell functions should work.

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

Commutative property example

the commutative property "**in Math**"

$$
\begin{aligned}
&[\![x]\!] + [\![y]\!] == [\![y]\!] + [\![x]\!] \\
&= [\![x + y]\!] == [\![y + x]\!] \\
&= [\![x + y == y + x]\!]
\end{aligned}
$$

where the third step also indicates that the Haskell

(==) :: Eq a => a -> a -> a

ought to have the properties of a

mathematical equivalence relationship.

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

Irrecoverable / recoverable errors

expressions that result in some kind of a **run-time error**,
such as dividing by zero, have the **value _|_** (read "**bottom**").

Such an **error** is not recoverable:
programs will not continue past these errors.

irrecoverable errors

errors encountered by the **I/O system**,
such as an **end-of-file error**, are recoverable
and are handled in a different manner.

recoverable errors

Such an **I/O error** is really not an **error** at all
but rather an **exception**.

https://www.haskell.org/tutorial/functions.html

Value in the semantic sense

The **value** is ⊥, usually pronounced "**bottom**".

It is a **value** in the *semantic sense*

-- it is not a normal Haskell value per se.

It represents **computations**

that do not produce a normal Haskell **value**:

exceptions and infinite loops, for example.

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

Denotational semantics and ⊥

denotational semantics, where ⊥ lives, is

a mapping **Haskell values**

to some **other space of values**.

in order to give meaning to programs

in a more formal manner

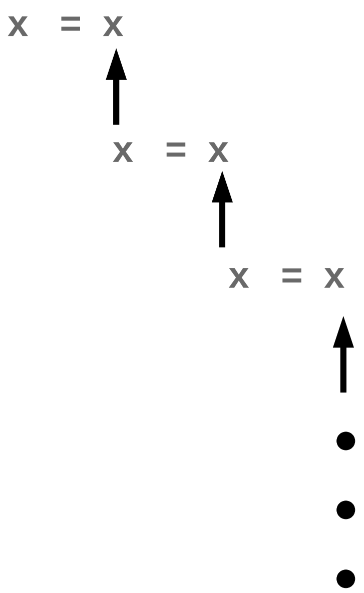
than just talking about what programs should do

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

let x = x in x

Consider an expression like **let x = x in x**

- there is no Haskell **value**
  for this expression.
- If you tried to evaluate it,
  it would simply never finish.
- not obvious what **mathematical object**
  this corresponds to.



x = x

x = x

x = x

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

⊥ for computations that does not return

in order to reason about programs

that have the following characteristics,

we need to give some **denotation** for it.

- with no Haskell **value**
- never finishing upon evaluation
- not obvious **mathematical object**

So, essentially, we just *make up a value* ⊥ (**bottom**)

for all these computations

So ⊥ is just a way to define

what a computation that doesn't return "means".

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

⊥ for throwing exceptions

We also define other computations like

undefined and error "some message" as ⊥

because they also do not have obvious normal **values**.

So throwing an exception corresponds to ⊥.

This is exactly what happens with a failed pattern match.

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

Lifted type

every Haskell **type** is "lifted" -- it *contains* ⊥.

That is, **Bool** corresponds to {⊥, **True**, **False**}
rather than just {**True**, **False**}.

This represents the fact that Haskell programs are
not guaranteed to **terminate** and can have **exceptions**.

This is also true when you define *your own type*
-- the type contains every value you defined for it as well as ⊥.

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

Bottom value in normal code

interestingly, since Haskell is non-strict,

⊥ can exist in normal code.

So you could have a value like Just ⊥ ,

and everything will work fine, unless you **evaluate** it,

A good example of this is const:

```
const 1 ⊥                                  -- 1
```

this works for failed pattern matches as well:

```
const 1 (let Just x = Nothing in x)        -- 1
```

constant function

const :: a -> b -> a

Input: **const 12 3**

Output: **12**

Input: **const 12 (3/0)**

Output: **12**

```
aaa x y = let      r = 3 *x
                   s = 6 *y
                   in  r + s
```

Input: **aaa 2 4**

Output: **30**

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

Pattern match in **let** expression (1)

```
let
  Just x = (binom (n-1) (k-1))
  Just y = (binom (n-1) k)
in
  Just (x + y)
```

It is fine from the type-checking point of view

extracting the underlying values from the **Just wrapper**
(these are **x** and **y**), adding them up and rewrapping them.

https://stackoverflow.com/questions/68240639/why-cant-you-use-just-syntax-without-let-in-block-in-haskell

Pattern match in **let** expression (2)

pattern matches in the **let**... **in** expression

assume that the results of **binom (n-1) (k-1)**

the results of the form **Just x**

but they could also be **Nothing** -

in which case your program will crash at runtime!

The "assignment" **Just x = ...**

matches ... against **Just x**,

binding **x** to the wrapped value *if the match succeeds*.

It doesn't apply **Just** to anything.

```
let
  Just x = (binom (n-1) (k-1))
  Just y = (binom (n-1) k)
in
  Just (x + y)
```

https://stackoverflow.com/questions/68240639/why-cant-you-use-just-syntax-without-let-in-block-in-haskell

Non-strict semantics (1)

An **expression language** is said to have **non-strict semantics**
if **expressions** can have a **value**
even if some of their **subexpressions** do not

Haskell is one of the few modern languages
to have **non-strict semantics** by default:

nearly every other language has **strict semantics**,
if any **subexpression** fails to have a **value**,
the whole expression fails with it.

https://wiki.haskell.org/Non-strict_semantics

Non-strict semantics (2)

non-strict semantics is one of the most important features in Haskell:

it is what allows programs
to work with conceptually **infinite** data structures,
and it is why people say that
Haskell lets you write *your own* **control** structures.

It's also one of the motivations
behind Haskell being a **pure language**
(though there are several other good ones).

https://wiki.haskell.org/Non-strict_semantics

Pure functions (1)

A **function** is called **pure**

if it corresponds to a function in the mathematical sense:

it associates each possible **input** value with an **output** value,

and does nothing else. In particular, it has no **side effects**

that is to say, invoking it produces no observable effect

other than the result it returns;

it cannot also e.g. write to disk, or print to a screen.

https://wiki.haskell.org/Pure

Pure functions (2)

A pure function is trivially referentially transparent

it does not depend on anything other than its parameters,
so when invoked
in a different **context** or
at a different **time**
with the same **arguments**,
it will produce the same **result**.

A programming language may be called purely functional
if evaluation of expressions is pure.

https://wiki.haskell.org/Pure

Non-strict vs. strict evaluation (1)

Non-strictness means that

reduction (the mathematical term for **evaluation**)

proceeds from the outside in,

(a+(b*c)) : first +, then (b*c)

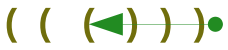
Strict languages work the other way around,

from the inside out

(a+(b*c)) : first (b*c), then +

Non-strictness

from the outside in,



Strict

from the inside out

https://wiki.haskell.org/Lazy_vs._non-strict

Non-strict vs. strict evaluation (2)

With **non-strictness**

the outer reduction may <u>eliminate</u> some of the sub-expressions

and does not <u>evaluate</u> them

so "bottom" can be <u>eliminated</u> and don't get be <u>evaluated</u>

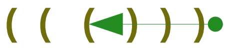
With **strictness**

if any sub-expression evaluates to bottom

then the bottom will *<u>propagate outwards</u>*.

Non-strictness

from the outside in,



Strict

from the inside out

https://wiki.haskell.org/Lazy_vs._non-strict

Lazy vs. non-strict (1)

only evaluating an expression *when its results are needed*

(note the shift from "reduction" to "evaluation").

when the evaluation engine sees an expression

it builds a **thunk** data structure containing

whatever **values** are needed to evaluate the expression,

plus a **pointer** to the expression itself.

when the result is actually needed

the evaluation engine calls the **expression** and

then replaces the **thunk** with the result for future reference.

https://wiki.haskell.org/Lazy_vs._non-strict

Lazy vs. non-strict (2)

Obviously there is a strong correspondence
between a **thunk** and a partly-evaluated expression.

in most cases the terms "**lazy**" and "**non-strict**"
seem to be synonyms.

but not quite, for instance
imagine an evaluation engine
on highly parallel hardware
that fires off sub-expression evaluation *eagerly*,
but then *throws away* results that are not needed.

With **non-strictness**
if you start from the outside and
work in, then some of the
sub-expressions are *eliminated*
by the outer reductions,
so they *don't get evaluated*
and you *don't get* "bottom".

Non-strictness
from the outside in,

https://wiki.haskell.org/Lazy_vs._non-strict

Lazy vs. non-strict (3)

In practice Haskell is not a purely **lazy** language:

for instance **pattern matching** is *usually* **strict**

So trying a **pattern match** forces **evaluation** to happen

at least far enough to accept or reject *the match*.

You can prepend a **~** in order

to make **pattern matches lazy**

https://wiki.haskell.org/Lazy_vs._non-strict

Lazy vs. non-strict (4)

The **strictness analyzer** also looks for cases where **sub-expressions** are always required by the **outer expression**, and converts those into **eager evaluation**.

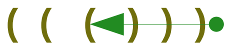
It can do this because the semantics (in terms of "bottom") don't change.

Programmers can also use the **seq** primitive to force an **expression** to evaluate regardless of whether the result will ever be used.

$! is defined in terms of **seq**.

Non-strictness

from the outside in,



Strict

from the inside out

With **non-strictness** reduction from the outside in then some sub-expressions are *eliminated* by the outer reductions, so they *don't get evaluated* and you *don't get* "bottom".

https://wiki.haskell.org/Lazy_vs._non-strict

Terminating expression

Intuitively,

a specific **function evaluation** is **terminating**,

where the **value** of every **argument** is supplied

if the Haskell **evaluation strategy** needs

finite number of steps to compute the result completely.

http://termination-portal.org/wiki/Functional_Programming

Non-terminating expression

the **function zeros** is considered **non-terminating**.

RHS is to be evaluated recursively, infinitely

zeros :: [Integer]
zeros = 0: zeros

the **evaluation** does not stop

when reaching a **term** headed by a **constructor**:

it will continue evaluating the **arguments** of this **constructor**.

zeros = 0: zeros
0: zeros
0: zeros
0: zeros

http://termination-portal.org/wiki/Functional_Programming

repeat

```
repeat :: a -> [a]
```

it creates an *infinite* list where all items are the first argument

```
take 4 (repeat 3)
[3,3,3,3]

take 6 (repeat 'A')
"AAAAAA"

take 6 (repeat "A")
["A","A","A","A","A","A"]
```

http://zvon.org/other/haskell/Outputprelude/repeat_f.html

foldr (1)

foldr will execute the callback **function** once

for each element in the structure.

The result will be passed

to the next invocation of the callback.

For the initial call to callback,

previousValue will be initialValue,

currentValue will be the last element of the structure.

https://wiki.haskell.org/Data.Foldable.foldr

foldr (2)

foldr (+) 4 [0, 1, 2, 3]

-- alternatively written without syntactic sugar for lists:

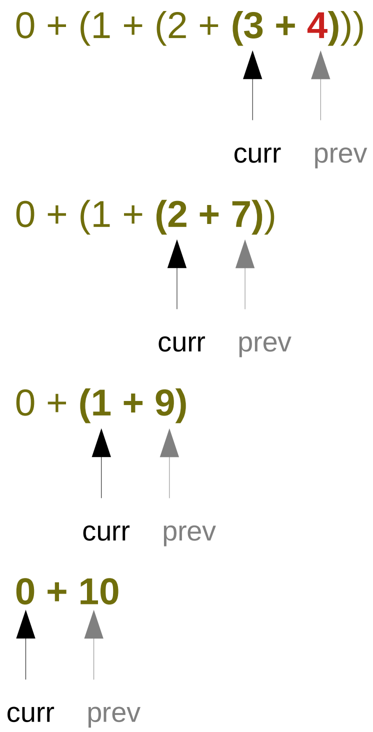
foldr (+) 4 (0 : (1 : (2 : (3 : []))))

would be equivalent to:

0 + (1 + (2 + **(3 + 4)**))

PreviousValue = initValue = 4

CurrentValue = last value = 3


0 + (1 + (2 + **(3 + 4)**))

curr prev

0 + (1 + **(2 + 7)**)

curr prev

0 + **(1 + 9)**

curr prev

0 + 10

curr prev

https://wiki.haskell.org/Data.Foldable.foldr

foldr (3)

foldr :: (a -> b -> b) -> b -> [a] -> b

it takes the second argument	**b**
and the last item of the list	**a** in **[a]**
and applies the function,	**(a -> b -> b)**

then it takes the penultimate item from the end
and the result, and so on.

last but one in a series of things; second last.

http://zvon.org/other/haskell/Outputprelude/foldr_f.html

foldr (4)

foldr :: (a -> b -> b) -> b -> [a] -> b

Input: foldr (+) 5 [1,2,3,4]   1 + (2 + (3 + (4 + 5)))

Output: 15

Input: foldr (/) 2 [8,12,24,4]   8 / (12 / (24 / (4 / 2)))

Output: 8.0

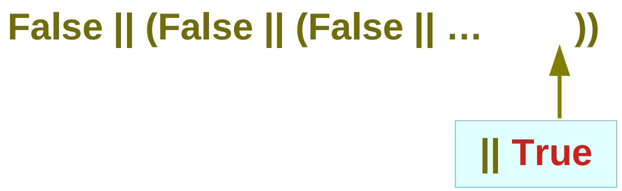
1 + (2 + (3 + (4 + 5)))
1 + (2 + (3 + 9))
1 + (2 + 12)
1 + 14
15

8 / (12 / (24 / (4 / 2)))
8 / (12 / (24 / 2))
8 / (12 / 12)
8 / 1
8

http://zvon.org/other/haskell/Outputprelude/foldr_f.html

Non-terminating expression (1)

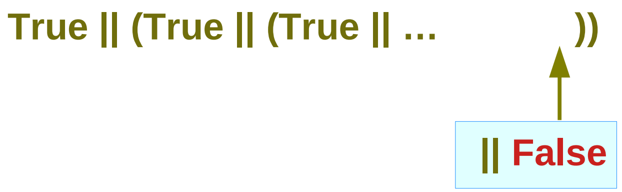
```
foldr (||) True $ repeat False            -- never terminates
```


False || (False || (False || …   ))

|| True

Infinitely check if there is any True,
But never reach the end

```
foldr (||) False $ repeat True            -- terminates with True
```


True || (True || (True || …   ))

|| False

There is at least one True,
Therefore return with true

https://stackoverflow.com/questions/7960543/why-does-this-haskell-code-not-terminate

Non-terminating expression (2)

```
foldr (||) True $ repeat False            -- never terminates
```

```
foldr (||) False $ repeat True            -- terminates with True
```

The first expands to **False || (False || (False || ...))**,
while the second expands to **True || (True || (True || ...))**.

The second argument to **foldr** is a red herring -
it occurs in the innermost application of **||**, not the **outermost**,
so it can never actually be reached.

The 2nd argument **True** is occurs
In the innermost application of **||**
The 2nd argument **False** is occurs
In the innermost application of **||**

A red herring is something that misleads or distracts from a relevant or important question.

https://stackoverflow.com/questions/7960543/why-does-this-haskell-code-not-terminate

Non-terminating expression (2)

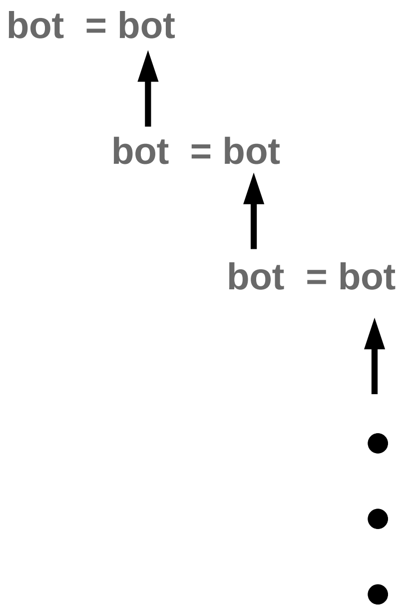
bot **=** **bot**

bot is a **non-terminating expression**.

Abstractly, we denote the **value**

of a **non-terminating expression**

as _|_ (read "**bottom**").



https://www.haskell.org/tutorial/functions.html

Termination Checkers

Does function f terminate?

A) {Yes, Don't know}

Typically look for decreasing size

- **Primitive recursive**
- **Walther recursion**
- **Size change termination**

https://ndmitchell.com/downloads/slides-catch-16_mar_2006.pdf

Termination Checkers

fib :: Integer -> Integer

fib(1) = 1

fib(2) = 1

fib(n) = fib(n-1) + fib(n-2)

fib(0) = $\perp^{NT}$

https://ndmitchell.com/downloads/slides-catch-16_mar_2006.pdf

Values

- **A function only stops terminating when its given a value**
- **Perhaps the question is wrong:**

Q) Given a function f and a value x,
Does f(x) terminate?

Q) Given a function f, for what values of x does
f(x) terminate?

https://ndmitchell.com/downloads/slides-catch-16_mar_2006.pdf

Non-terminate

```
fib n | n <= 0 =
error “bad programmer!”
```

- A function should <u>never</u> non-terminate
- It should give an helpful error message
- There may be a few <u>exceptions</u>
  - But probably things that can’t be proved
  - i.e. A Turing machine simulator

https://ndmitchell.com/downloads/slides-catch-16_mar_2006.pdf

Laziness

Haskell is:

- **A functional programming language**
  - **Lazy – not strict**
  - **Only evaluates what is required**
- **Lazy allows:**
  - **Infinite data structures**

https://ndmitchell.com/downloads/slides-catch-16_mar_2006.pdf

Productivity

[1..] = [1,2,3,4,5,6, ...

- Not terminating
- But is productive
  - Always another element
  - Time to generate “next result” is always finite

https://ndmitchell.com/downloads/slides-catch-16_mar_2006.pdf

Evaluation

The blame game

- **last [1..] is ⊥NT**
- **last is a useful function**
- **[1..] is a useful value**

- **Who is at fault?**
  - **The caller of last**

https://ndmitchell.com/downloads/slides-catch-16_mar_2006.pdf

A lazy termination checker

- All data/functions must be productive
- Can easily encode termination

```
isTerm :: [a] -> Bool
isTerm [] = True
isTerm (x:xs) = isTerm xs
```

https://ndmitchell.com/downloads/slides-catch-16_mar_2006.pdf

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf