Lazy evaluation illustrated
for Haskell divers

exploring some mental models and implementations

Takenobu T.
Lazy,...  zzz

..., It's fun!
NOTE
- Meaning of terms are different for each community.
- There are a lot of good documents. Please see also references.
- This is written for GHC's Haskell.
1. Introduction
   - Basic mental models
   - Lazy evaluation
   - Simple questions
2. Expressions
   - Expression and value
   - Expressions in Haskell
   - Classification by values and forms
   - WHNF
3. Internal representation of expressions
   - Constructor
   - Thunk
   - Uniform representation
   - WHNF
   - let, case expression
4. Evaluation
   - Evaluation strategies
   - Evaluation in Haskell (GHC)
   - Examples of evaluation steps
   - Examples of evaluations
   - Controlling the evaluation
5. Implementation of evaluator
   - Lazy graph reduction
   - STG-machine
6. Semantics
   - Bottom
   - Strict/Non-strict
   - Lifted and boxed types
   - Strictness analysis
   - Sequential order
7. Appendix
   - References
1. Introduction
1. Introduction

Basic mental models
How to evaluate a program in your brain?

How to evaluate (execute, reduce) the program in your brain?

What “mental model” do you have?
One of the mental models for C program

C program

A sequence of statements

```c
main (...) {
  code..
  code..
  code..
  code..
}
```

A nested structure

```c
x = func1( func2( a ) );
```

A sequence of arguments

```c
y = func1( a(x), b(x), c(x) );
```

A function and arguments

```c
z = func1( m + n );
```

How to evaluate (execute, reduce) the program in your brain?
What step, what order, ... ?
C program

A program is a collection of statements.

A sequence of statements

main(...) {
    code..
code..
code..
code..
}

Statements are executed downward.

A nested structure

x = func1( func2( a ) );

from inner to outer

A sequence of arguments

y = func1( a(x), b(x), c(x) );

from left to right

A function and arguments

z = func1( m + n );

arguments first apply second

Each programmer has some mental models in their brain.
Maybe, You have some implicit mental model in your brain for C program.

(1) A program is a collection of statements.

(2) There is the order between evaluations of elements.

(3) There is the order between termination and start of evaluations.

This is a syntactically straightforward model for programming languages.
(an implicit sequential order model)
One of the mental models for Haskell program

Haskell program

```
main = exp_{aa} (exp_{ab} exp_{ac} exp_{ad})

exp_{ac} = exp_{aca} exp_{acb}

exp_{ad} = exp_{ada} exp_{adb} exp_{adc}
```

How to evaluate (execute, reduce) the program in your brain?
What step, what order, ...?
A program is a collection of expressions.

A entire program is regarded as a single expression.

The subexpression is evaluated (reduced) in some order.

The evaluation is performed by replacement.
(1) A program is a collection of expressions.

(2) A entire program is regarded as a single expression.

(3) The subexpressions are evaluated (reduced) in some order.

(4) The evaluation is performed by replacement.

This is an example of an expression reduction model for Haskell.
1. Introduction

Lazy evaluation
Why lazy evaluation?

- To manipulate infinite data structures
- To manipulate huge data structures
- To manipulate streams
- Pure is order free
- 2nd Church-Rosser theorem
- To implement non-strict semantics
- Out-of-order optimization
- Potentially parallelism
- Modularity
- Abstraction
- Amortizing
- Asynchronization
- Reactive
- Fun
- There are various reasons 😊

References: [H4], [H3], [B2], [B7], [B8], [D2], [D12], [D13], [D14]
Haskell(GHC)‘s lazy evaluation

Lazy evaluation

- evaluate **only when needed**

- evaluate **only enough**

- evaluate **at most once**

“Lazy” is “delay and avoidance” rather than “delay”.

References: [B2] Ch.7, [H4] Ch.11, 12, [D2]
1. Introduction

**Ingredient of Haskell(GHC)’s lazy evaluation**

- only when needed
- normal order reduction
- +
- only enough
- stop at WHNF
- +
- at most once
- substitute pointers
  - update redex root with result

This strategy is implemented by lazy graph reduction.

References: [B2] Ch.7, [H4] Ch.11, 12, [D2]
Techniques of Haskell(GHC) ’s lazy evaluation

1. Introduction

normal order reduction (leftmost outermost reduction)

pattern-matching

thunk

lazy graph reduction

call-by-need

substitute pointers

update redex root with result

self-updating model

full laziness

evaluate only when needed

evaluate only enough

evaluate at most once

References : [B2] Ch.7, [H4] Ch.2, 11, 12, 15, [H5], [D2]
1. Introduction

Simple questions
An expression is evaluated by normal order (leftmost outermost redex first).

Normal order reduction guarantees to find a normal form (if one exists).
To avoid unnecessary computation, normal order reduction chooses to apply the function rather than first evaluating the argument.
How to postpone?

To postpone the evaluation, an unevaluated expression is built in the heap memory.
When needed?

Pattern-matching or forcing request drive the evaluation.

References: [H4], [D2], [D5]
What to be careful about?

To consider hidden **space leak**

To consider **performance cost** to postpone unevaluated expressions

To consider evaluation (execution) **order** and **timing** in real world

You can avoid the pitfalls by controlling the evaluation.

References: [H4], [D2], [D5]
2. Expressions
2. Expressions

Expression and value
What is an expression?

An expression

An expression denotes a value

An expression

1 + 2

An expression is evaluated to a value

An expression

1 + 2

evaluate

3

A value

There are many evaluation approaches

An expression

\[(1 + 2)^2\]

evaluation strategies

- Strict, Non-strict evaluation
- Eager, Lazy evaluation
- Call-by-value, Call-by-name, Call-by-need, ...
- Innermost, Outermost
- Normal order, Applicative order
- ...

A value

There are some evaluation levels

An expression

[1, 2, 3]

NF
(Normal Form)

WHNF
(Weak Head Normal Form)

A value

References: [D3], [B2] Ch.2, 7, [B6] Ch.3, [D1]
2. Expressions

Expressions in Haskell
There are many expressions in Haskell

Expressions

- Just 5
- 1 + 2
- (1, 2)
- take 5 xs
- [1, 2, 3]
- let x = 1 in x + y
- if b then 1 else 0
- 'a'
- map f xs
- x : xs
- \ x -> x + 1
- let x = 1 in x + y
- x : xs
- case x of _ -> 0
- do {x <- get; put x}
- xs

References: [B2] Ch.2, [H1] Ch.3
2. Expressions

Expression categories in Haskell

- **lambda abstraction**
  \[ x \rightarrow x + 1 \]

- **let expression**
  \[ \text{let } x = 1 \text{ in } x + y \]

- **conditional**
  \[ \text{if } b \text{ then } 1 \text{ else } 0 \]

- **case expression**
  \[ \text{case } x \text{ of } _ \rightarrow 0 \]

- **do expression**
  \[ \text{do } \{ x \leftarrow \text{get}; \text{put } x \} \]

- **function application**
  \[ \text{take } 5 \text{ xs} \]
  \[ (\text{\( x \rightarrow x + 1 \)} )3 \]
  \[ 1 + 2 \]
  \[ \text{map } f \text{ xs} \]
  \[ \text{fun } \text{arg} \]

- **general constructor, literal and some forms**
  \[ 7 \]
  \[ [1, 2, 3] \]
  \[ (1, 2) \]
  \[ 'a' \]
  \[ x : \text{xs} \]
  \[ \text{Just } 5 \]
  \[ \text{xs} \]

References: [H1] Ch.3, [B2] Ch.2
Specification is described in Haskell 2010 Language Report

"Haskell 2010 Language Report, Chapter 3 Expressions" [H1]
Classification by values and forms
Classification by values

Expressions

Values are data values or function values.

References: [H5]
2. Expressions

Classification by forms

Expressions

unevaluated expressions

- take 5 xs
- (\ x -> x + 1) 3
- map f xs
- let x = 1 in x + y
- fun arg
- case x of _ -> 0
- do (x <- get; put x)

Values

WHNF

- \ x -> abs 1

HNF

- \ x -> x + (abs 1)

NF

- \ x -> x
- 7
- 'a'
- [1, 2, 3]
- Just 5
- (1, 2)
- Just (f x)
- [f x, g y]

bottom

Values are WHNF, HNF or NF.

References: [H4] Ch.11, [D3], [B6] Ch.3, [B2] Ch.2, 7, [D1], [W1]
2. Expressions

WHNF
WHNF is one of the form in the evaluated values

2. Expressions

An expression

exp

(1) normal order reduction
of top-level (head) redexes

(2) normal order reduction
of inner level redexes

WHNF
(Weak Head Normal Form)
no top-level redexes

NF
(Normal Form)
no redexes at all

A value

References: [H4] Ch.11, [D3], [B6] Ch.3, [B2] Ch.2, 7, [D1], [W1]
WHNF is a value which has evaluated top-level

2. Expressions

WHNF for a data value and a function value

**A data value in WHNF**
- constructor
- no top-level redex
- inner redexes

- \( \exp_0 \), \( \exp_1 \), \( \exp_2 \), ..., \( \exp_n \)

**A function value in WHNF**
- lambda abstraction
- no top-level redex
- inner redexes

- \( \lambda x_1 \ldots x_n \rightarrow \exp \)

Examples of WHNF

- **WHNF**
  - `Just 7`
  - `Just (abs x)`
  - `Cons (f 1) (map f [2..])`
  - `\ x -> x + 1`

- **no WHNF**
  - `abs 7`
  - `if x then True else False`

HNF is a value which has evaluated top-level

* GHC uses WHNF rather than HNF.
HNF for a data value and a function value

**a data value in HNF** (same as WHNF)
- constructor
- inner redexes
- no top-level redex

**a function value in HNF**
- lambda abstraction
- no redex

References: [H4] Ch.11, [D3], [B3]
Examples of HNF

HNF

\[ \text{Just} \quad (\text{abs } x) \]

no top-level redex

no top-level redex

\[ \backslash x \rightarrow \text{Just} \quad (\text{abs } 7) \]

no top-level redex

no HNF

\[ \text{abs} \quad 7 \]

top level-redex

\[ \backslash x \rightarrow \text{abs} \quad 7 \]

top level-redex

References: [H4] Ch.11, [D3], [B3]
NF is a value which has no redex.

top-level (head) is a constructor or a lambda abstraction

no internal redex

2. Expressions

NF for a data value and a function value

**a data value in NF**

constructor

\[ \text{exp}_0 \quad \text{exp}_1 \quad \text{exp}_2 \quad \ldots \quad \text{exp}_n \]

no internal redex

**a function value in NF**

lambda abstraction

\[ \lambda x_1 \ldots x_n \rightarrow \text{exp}_0 \quad \text{exp}_1 \quad \ldots \quad \text{exp}_n \]

no internal redex

Examples of NF

NF

- Just 7
  - no internal redex

- Cons 1 Nil
  - no internal redex

- \( x \rightarrow x + 1 \)
  - no internal redex

No NF

- Just (abs 7)
  - redex

- \( x \rightarrow \text{Just (abs 7)} \)
  - redex

2. Expressions

**WHNF, HNF, NF**

**WHNF**
- Top-level (head) is a constructor or a lambda abstraction.
- No top-level redex.

**HNF**
- Top-level (head) is a constructor or a lambda abstraction with no top-level redex.
- No top-level redex.

**NF**
- No internal redex.

2. Expressions

Definition of WHNF and HNF

“The implementation of functional programming languages” [H4]

11.3.1 Weak Head Normal Form
To express this idea precisely we need to introduce a new definition:

**DEFINITION**
A lambda expression is in weak head normal form (WHNF) if and only if it is of the form

\[ F \ E_1 \ E_2 \ldots \ E_n \]

where \( n \geq 0 \);
and either \( F \) is a variable or data object
or \( F \) is a lambda abstraction or built-in function
and \((F \ E_1 \ E_2 \ldots \ E_m)\) is not a redex for any \( m \leq n \).

An expression has no top-level redex if and only if it is in weak head normal form.

11.3.3 Head Normal Form
Head normal form is often confused with some discussion. The content of this is
since for most purposes head normal form. Nevertheless, we will stick to this.
3. Internal representation of expressions
3. Internal representation of expressions

Constructor
Constructor is one of the key elements to understand WHNF and lazy evaluation in Haskell.
A constructor builds a structured data value.
A constructor distinguishes the data value in expressions.

References: [B2], [H1], [H4] Ch.2, 10, [B6] Ch.11
Constructors and data declaration

Constructors are defined by data declaration.

References: [B2], [H1]
Internal representation of Constructors for data values

Haskell code

- Nothing
- Just 7

GHC's internal representation

- header
- payload

heap memory

References: [H11], [H10], [H5], [H6], [H7]
Constructors are represented uniformly

GHC's internal representation

A data value is represented with header(constructor) + payload(components).

References: [H11], [H10], [H5], [H6], [H7], [D15]
Haskell code

```haskell
data Bool = False | True
```

GHC’s internal representation

```
False
```

```
True
```

```haskell
data Maybe a = Nothing | Just a
```

```
Nothing
```

```
Just
```

```
Left
```

```
Right
```

```haskell
data Either a b = Left a | Right b
```

References: [H11], [H10], [H5], [H6], [H7]
Primitive data types are also represented with constructors

Haskell code

```haskell
data Int = I# Int#
data Char = C# Char#
```

GHC's internal representation

```
boxed integer

I# 0#
I# 1#
```

unboxed integer

```
C# 'a'#
C# 'b'#
```

References: [H11], [H10], [H5], [H6], [H7]
List is also represented with constructors

```
[ 1, 2, 3 ]
```

syntactic desugar

```
1 : ( 2 : ( 3 : [] ) )
```

prefix notation by section

```
( : ) 1 ( ( : ) 2 ( ( : ) 3 [] ) )
```

equivalent data constructor

```
Cons 1 ( Cons 2 ( Cons 3 Nil ) )
```

References: [H11], [H10], [H5], [H6], [H7]
List is also represented with constructors

List

\[ [1, 2, 3] \]

syntactic desugar

\[ 1 : (2 : (3 : [])[]) \]

prefix notation by section

\[ (:1 (:2 (:3 []))) \]

equivalent data constructor

\[ \text{Cons } 1 \ (\text{Cons } 2 \ (\text{Cons } 3 \ \text{Nil})) \]

type declaration

\[ \text{data List } a = \begin{array}{c}
\text{Nil} \\
\text{: } a \ \text{(List } a) \\
\end{array} \]

* pseudo code

\[ \text{data List } a = \begin{array}{c}
\text{Nil} \\
\text{Cons } a \ \text{(List } a) \\
\end{array} \]

References: [H11], [H10], [H5], [H6], [H7]
List is also represented with constructors

Haskell code

```haskell
data List a = []
  | Cons a (List a)
```

GHC's internal representation

```
[]
(::)
Nil
Cons
```

heap memory

References: [H11], [H10], [H5], [H6], [H7]
List is also represented with constructors

Haskell code:

\[
[1, 2, 3]
\]

\[
1 : (2 : (3 : []))
\]

\[
(\cdot) 1 (\cdot) 2 (\cdot) 3 []
\]

\[
\text{Cons} 1 (\text{Cons} 2 (\text{Cons} 3 \text{Nil}))
\]

GHC’s internal representation:

\[
\text{Cons} (:) \\
1 \\
\text{Cons} (:) \\
2 \\
\text{Cons} (:) \\
3 \\
\text{Nil} []
\]
Tuple is also represented with constructor

Tuple (Pair)

(7, 8)

prefix notation by section

(,) 7 8

equivalent data constructor

Pair 7 8

constructor

type declaration

* pseudo code

data Pair a = (,) a a

data Pair a = Pair a a

References: [H11], [H10], [H5], [H6], [H7]
3. Internal representation of expressions

Tuple is also represented with constructor

Haskell code

\[
\text{data } \text{Pair } \alpha = (\_, \alpha, \alpha)
\]

\[
\text{data } \text{Pair } \alpha = \text{Pair} \alpha \alpha
\]

GHC's internal representation

\[
(\_,) \alpha \alpha
\]

\[
\text{Pair} \alpha \alpha
\]

heap memory

References: [H11], [H10], [H5], [H6], [H7]
Tuple is also represented with constructor

Haskell code

(7, 8)

(,) 7 8

Pair 7 8

 GHC’s internal representation

Pair (,)

 7 8

References: [H11], [H10], [H5], [H6], [H7]
3. Internal representation of expressions

Thunk
A thunk is an *unevaluated* expression in heap memory.
A thunk is built to *postpone* the evaluation.

References: [B5] Ch.2, [D5], [W1], [H10], [H5], [D7]
Internal representation of a thunk

Haskell code

An unevaluated expression

take \(y\) \(ys\)

GHC’s internal representation

thunk

header

payload

info ptr

take \(y\) \(ys\)

code

free variables

\(y\) \(ys\)

A thunk is represented with header(code) + payload(free variables).

References: [H11], [H10], [D2], [H5], [H6], [H7], [B5] Ch.2, [D5], [W1]
A thunk is a package of code + free variables.

References: [D2], [H11], [H10], [H5], [H6], [H7], [B5] Ch.2, [D5], [W1]
A thunk is evaluated by forcing request

Haskell code

An unevaluated expression

\[
\text{take } y \; y_s
\]

An evaluated expression

\[
[3]
\]

\[
\text{evaluate by forcing request}
\]

\[
\text{evaluate by forcing request}
\]

GHC's internal representation

thunk

header

payload

\[
\text{take } y \; y_s
\]

code

free variables

\[
\text{Cons (::)}
\]

\[
3 \quad \text{Nil ([])}
\]

References: [D7], [D2], [H11], [H10], [H5], [H6], [H7], [B5] Ch.2, [D5], [W1], [D15]
3. Internal representation of expressions

Uniform representation
Every object is uniformly represented in memory

- Header
- Payload
  - Object type
  - Constructor
  - Function
  - Thunk
- Data components

In heap memory, stack or static memory

References: [H11], [H10], [H5], [H6], [H7], [D15]
Every object is uniformly represented in memory

- header
- payload

a data value  a function value  a thunk

References: [H11], [H10], [H5], [H6], [H7], [D15]
Every object is uniformly represented in memory

3. Internal representation of expressions

- A thunk has a reserved field in the second.

References: [H11], [H10], [H5], [H6], [H7], [D15]
3. Internal representation of expressions

WHNF
3. Internal representation of expressions

**Internal representation of WHNF**

**Haskell code**

- A data value in WHNF
  
  \[ \text{constructor} \vdash \text{exp}_0, \text{exp}_1, \ldots, \text{exp}_n \]

- A function value in WHNF
  
  \[ \lambda x_1 \ldots x_n \to \text{exp} \]

**GHC's internal representation**

- Heap memory
  
  \[ \text{info ptr} \rightarrow \text{constructor} \]

- Data component(s)
  
  \[ \text{info ptr} \rightarrow \text{exp}_2 \rightarrow \ldots \]

- Code
  
  \[ \text{exp}_2 \rightarrow \text{info ptr} \rightarrow \text{code} \rightarrow \text{(exp)} \]

- Free variables
  
  \[ \text{free variables} \rightarrow \text{info ptr} \rightarrow \text{exp}_2 \rightarrow \ldots \]

References: [H11], [H5], [H6], [H7], [H10]
Constructors can contain unevaluated expressions by thunks.
Haskell's constructors are lazy constructors.

References: [H11], [H5], [H6], [H7], [H10]
Example of WHNF for a data value

Haskell code

[ map f xs ]

syntactic desugar

Cons (map f xs) Nil

constructor

GHC's internal representation

Cons

constructor

Nil

thunk

info ptr

map f xs

free variables

References: [H11], [H5], [H6], [H7], [H10]
3. Internal representation of expressions

let, case expression
let, case expression

let and case expressions are special role in the evaluation
A let expression may build a thunk.
A case expression evaluates (forces) and deconstructs the thunk.
A let expression may allocates a heap object

- allocate (build)

<table>
<thead>
<tr>
<th>let expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>let ( x = ..... )</td>
</tr>
</tbody>
</table>

- heap memory

<table>
<thead>
<tr>
<th>a data value</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>or</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>a function value</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>or</th>
</tr>
</thead>
</table>

| a thunk (an unevaluated expression) |

A let expression may allocates an object in the heap.
(If GHC can optimize it, the let expression may not allocate.)

* At exactly, STG language’s let expression rather than Haskell’s let expression

References: [H5], [H6], [H7], [H10]
Example of let expressions

Haskell code

let x = Just 5

allocate

let x = \ y -> y + z

allocate

let x = take y ys

allocate (build)

GHC's internal representation

a data value

Just

5

allocate

a function value

\ y -> y + z

free variables

allocate

a thunk

info ptr

take y ys

free variables

References: [H5], [H6], [H7], [H10]
A case expression evaluates a subexpression

**Pattern-matching drives the evaluation.**

* At exactly, STG language’s case expression rather than Haskell’s case expression

References: [H5], [H6], [H7], [H10], [D2]
A case expression also perform case analysis

A case expression evaluates a subexpression and optionally performs case analysis on its value.

* At exactly, STG language’s case expression rather than Haskell’s case expression

References: [H5], [H6], [H7], [H10], [D2]
Example of a case expression

A case expression's pattern-matching says "I need the value".

References: [H5], [H6], [H7], [H10], [D2]
A function’s pattern-matching is syntactic sugar of case expression.

A function’s pattern-matching also drives the evaluation.
4. Evaluation
4. Evaluation

Evaluation strategies
The evaluation produces a value from an expression.
There are many evaluation approaches:

- Strict, Non-strict evaluation
- Eager, Lazy evaluation
- Call-by-value, Call-by-name, Call-by-need, ...
- Innermost, Outermost
- Normal order, Application order
- ...

An expression:

\[ 1 + 2 \]

A value:

\[ 3 \]
Evaluation concept layer

- Denotational semantics
- Operational semantics (Evaluation strategies / Reduction strategies)
- Implementation techniques
4. Evaluation

Evaluation layer for GHC's Haskell

Denotational semantics
- Strict semantics
- Non-strict semantics

Operational semantics
- Strict evaluation
- Non-strict evaluation
- Eager evaluation
- Nondeterministic evaluation
- Lazy evaluation
- Call-by-Value
- Call-by-Name
- Call-by-Need
- Applicative order reduction
- Normal order reduction
- Rightmost reduction
- Innermost reduction
- Leftmost reduction
- Outermost reduction

Implementation techniques
- Tree reduction
- Lazy graph reduction

References: [D3], [D1], [D2], [D5], [D4], [B2] Ch.7, [B3] Ch.8, [B6] Ch.5, [W1], [W2], [W3], [B7], [B8]
4. Evaluation

Evaluation layer for GHC's Haskell

Denotational semantics
- Strict semantics
- Non-strict semantics

Operational semantics
- Strict evaluation
- Non-strict evaluation
- GHC's strategy
- Haskell 2010 specification

(Evaluation strategies/Reduction strategies)
- Eager evaluation
- Nondeterministic evaluation
- Lazy evaluation
- GHC's strategy

- Call-by-Value
- Call-by-Name
- Call-by-Need
- GHC's strategy

- Applicative order reduction
- Normal order reduction
- GHC's strategy

- Rightmost reduction
- Innermost reduction
- Leftmost reduction
- Outermost reduction
- GHC's strategy

Implementation techniques
- Tree reduction
- Lazy graph reduction
- GHC's implementation
- GHC's strategy

References: [D3], [D1], [D2], [D5], [D4], [B2] Ch.7, [B3] Ch.8, [B6] Ch.5, [W1], [W2], [W3], [B7], [B8]
Evaluation strategies

Each evaluation strategy decides how to operate the evaluation, about ...

- ordering,
- region,
- trigger condition,
- termination condition,
- re-evaluation, ...

References: [D3], [D1], [D2], [D5], [D4], [B2] Ch.7, [B3] Ch.8, [B6] Ch.5, [W1], [W2], [W3], [B7], [B8]
One of the important points is the order

which first?

function arguments

apply

apply first
lazy evaluation,
call-by-name,
call-by-need,
outermost reduction,
normal order reduction

argument first
eager evaluation,
call-by-value,
innermost reduction,
applicative order reduction

References: [D3], [D1], [D2], [D5], [D4], [B2] Ch.7, [B3] Ch.8, [B6] Ch.5, [W1], [W2], [W3], [B7], [B8]
4. Evaluation

Simple example of typical evaluations

**call-by-value**

\[ \text{square} \left( 1 + 2 \right) \]

**call-by-need**

\[ \text{square} \left( 1 + 2 \right) \]

References: [B2] Ch.7, [B3] Ch.8, [D4], [B6] Ch.5
Simple example of typical evaluations

**call-by-value**

\[
\text{square} (1 + 2) \\
\text{square} (3) \\
3 \times 3 \\
9
\]

**call-by-need**

\[
\text{square} (1 + 2) \\
(1 + 2) \times (1 + 2) \\
(3) \times (3) \\
9
\]

evaluation is delayed!

References: [B2] Ch.7, [B3] Ch.8, [D4], [B6] Ch.5
4. Evaluation

Evaluation in Haskell (GHC)
Key concepts of GHC's lazy evaluation

An expression

fun args

reduce in normal order and
drive the evaluation
by pattern-matching
postpone the evaluation of arguments
to evaluate only when needed

evaluate

stop at WHNF
to evaluate only enough

update itself
to evaluate at most once

A value

NF

WHNF

References: [H4] Ch.11, 12, [H5], [H6], [D2]
Postpone the evaluation of arguments

Haskell code

```
fun (map g1 ys)
```

```
let thunk0 = map g1 ys
in fun thunk0
```

Postpone the evaluation by a thunk which build with let expression

(When GHC can optimize it by analysis, the thunk may not be build.)

References: [H5], [H6], [H10]
Pattern-matching drives the evaluation

4. Evaluation

Pattern-matching drives the evaluation by case expression:

\[
\text{case } x \text{ of}
\]

- pattern1 -> alt1
- pattern2 -> alt2

A thunk is created and evaluated in heap memory, driven by pattern-matching.

References: [H5], [H6], [H7], [H10], [D2]
Stop at WHNF

Haskell code

an unevaluated expression

evaluate

WHNF

evaluated

GHC's internal representation

a thunk

info ptr

code

free variables

heap memory

evaluate

a value (WHNF)

info ptr

evaluated

evaluated

stop the evaluation at WHNF

References: [H5], [H6], [H10]
4. Evaluation

Examples of evaluation steps
(1) Example of GHC's evaluation

Let’s evaluate. It’s time to magic!

* no optimizing case (without -O)
(2) How to postpone the evaluation of arguments?

```
tail (map abs [1, -2, 3])
```
(3) GHC internally translates the expression

\[
\text{let } \text{thunk0} = \text{map abs [1, -2, 3]} \\
\text{in } \text{tail thunk0}
\]
(4) a let expression builds a thunk

\[
\text{let } \text{thunk0} = \text{map abs [1, -2, 3] in tail thunk0}
\]

```
let thunk0 = map abs [1, -2, 3]
in tail thunk0
```
(5) function apply to argument

tail (map abs [1, -2, 3])

let thunk0 = map abs [1, -2, 3]
in tail thunk0
(6) tail function is defined here

tail (map abs [1, -2, 3])

let thunk0 = map abs [1, -2, 3]
in tail thunk0

tail (_:xs) = xs

heap memory

thunk

map f xs

abs [1, -2, 3]
tail (map abs [1, -2, 3])

let thunk0 = map abs [1, -2, 3]

in tail thunk0

(7) function's pattern is syntactic sugar

heap memory

syntactic desugar

tail (_:xs) = xs

tail y = case y of
(_:xs) -> xs

internal translation
(8) substitute the function body (beta reduction)

tail (map abs [1, -2, 3])

let thunk0 = map abs [1, -2, 3]
in tail thunk0

heap memory

4. Evaluation

reduction

case thunk0 of
(_:xs) -> xs

tail (map abs [1, -2, 3])

internal translation

let thunk0 = map abs [1, -2, 3]
in tail thunk0

tail (_:xs) = xs

tail y = case y of
(_:xs) -> xs
(9) case pattern-matching drives the evaluation

```haskell
let thunk0 = map abs [1, -2, 3]
in tail thunk0

tail (map abs [1, -2, 3])

internal translation

heap memory

map f xs
abs [1,-2,3]

evaluate

drive the evaluation (forcing request)

case thunk0 of
(::_xs) -> xs

tail y = case y of
(::_xs) -> xs

tail (_:xs) = xs

case thunk0 of
(::_xs) -> xs
```
4. Evaluation

(10) but, stop at WHNF

\[
\text{let } \text{thunk0} = \text{map abs } [1, -2, 3] \text{ in tail thunk0}
\]

\[
\text{case thunk0 of } (_:xs) \to xs
\]

\[
\text{case (abs 1) : (map abs [-2, 3]) of } (_:xs) \to xs
\]

\[
\text{heap memory}
\]

\[
\text{thunk}
\]

\[
\text{map f xs}
\]

\[
\text{abs [1,-2,3]}
\]

\[
\text{stop at WHNF}
\]

\[
\text{evaluate}
\]

\[
\text{Cons}
\]

\[
\text{thunk}
\]

\[
\text{abs x}
\]

\[
\text{map f xs}
\]

\[
\text{abs [-2,3]}
\]
(11) bind variables to a result

tail (map abs [1, -2, 3])

let thunk0 = map abs [1, -2, 3]
in tail thunk0

tail (_:xs) = xs

tail y = case y of
  (_:xs) -> xs

case thunk0 of
  (_:xs) -> xs

case (abs 1) : (map abs [-2, 3]) of
  (_:xs) -> xs

heap memory

thunk

map f xs

abs [1,-2,3]

evaluate

case (abs 1) : (map abs [-2, 3]) of
  (_:xs) -> xs
(12) return the value

tail (map abs [1, -2, 3])

let thunk0 = map abs [1, -2, 3]

in tail thunk0

case thunk0 of
  (_:xs) -> xs

case (abs 1) : (map abs [-2, 3]) of
  (_:xs) -> xs

map abs [-2, 3]

heap memory

thunk

map f xs

abs [1,-2,3]

evaluate

constructor

Cons

thunk

thunk

thunk

thunk

thunk

map f xs

abs [-2,3]
Key points

```
let thunk0 = map abs [1, -2, 3]
in tail thunk0
```

```
tail (map abs [1, -2, 3])
```

```
internal translation
postpone by thunk
```

```
thunk
map f xs
abs [1,-2,3]
```

```
heap memory
```

```
tail (_:xs) = xs
```

```
tail y = case y of
  (_:xs) -> xs
```

```
case thunk0 of
  (_:xs) -> xs
```

```
case (abs 1) : (map abs [-2, 3]) of
  (_:xs) -> xs
```

```
map abs [-2, 3]
```

```
a result value
```

4. Evaluation
4. Evaluation

Examples of evaluations

* no optimizing case (without -O)
4. Evaluation

Example of repeat

```
repeat 1
```

```
1 : repeat 1
```

```
1 : 1 : repeat 1
```

```
1 : 1 : 1 : repeat 1
```

References: [D5], [D6], [D8], [D9], [D10], [H10]
Example of repeat

4. Evaluation

References: [D5], [D6], [D8], [D9], [D10], [H10]
Example of map

```
map f [1, 2, 3]
```

```
f 1 : map f [2, 3]
```

```
f 1 : f 2 : map f [3]
```

```
f 1 : f 2 : f 3
```

...
Example of map

4. Evaluation

References: [D5], [D6], [D8], [D9], [D10], [H10]
4. Evaluation

Example of foldl (non-strict)

```haskell
foldl (+) 0 [1 .. 100]
```

```
foldl (+) (0 + 1) [2 .. 100]
```

```
foldl (+) (((0 + 1) + 2) + 3) [4 .. 100]
```

References: [D5], [D6], [D8], [D9], [D10], [H10]
Example of foldl (non-strict)

foldl (+) 0 [1 .. 100]

foldl (+) (0 + 1) [2 .. 100]
let thunk1 = (0 + 1)
in foldl (+) thunk1 [2 .. 100]

foldl (+) ((0 + 1) + 2) [3 .. 100]
let thunk2 = (thunk1 + 2)
in foldl (+) thunk2 [3 .. 100]

foldl (+) (((0 + 1) + 2) + 3) [4 .. 100]
let thunk3 = (thunk2 + 3)
in foldl (+) thunk3 [4 .. 100]

heap memory
*show only accumulation value

thunk1
(+)
0 1

thunk2
(+)
0 1 2

thunk3
(+)
0 1 2 3

References : [D5], [D6], [D8], [D9], [D10], [H10]
Example of foldl' (strict)

foldl' (+) 0 [1 .. 100]

foldl' (+) (0 + 1) [2 .. 100]

foldl' (+) (1 + 2) [3 .. 100]

foldl' (+) (3 + 3) [4 .. 100]

References: [D5], [D6], [D8], [D9], [D10], [H10]
Example of foldl' (strict)

foldl' (+) 0 [1 .. 100]

foldl' (+) (0 + 1) [2 .. 100]
let thunk1 = (0 + 1)
in thunk1 `seq`
foldl' (+) thunk1 [2 .. 100]

foldl' (+) (1 + 2) [3 .. 100]
let thunk2 = (1 + 2)
in thunk2 `seq`
foldl' (+) thunk2 [3 .. 100]

foldl' (+) (3 + 3) [4 .. 100]
let thunk3 = (3 + 3)
in thunk3 `seq`
foldl' (+) thunk3 [4 .. 100]

heap memory

References: [D5], [D6], [D8], [D9], [D10], [H10]
Example of foldl (non-strict) and foldl’ (strict)

```
foldl (+) (0 + 1) [2 .. 100]
foldl (+) ((0 + 1) + 2) [3 .. 100]
foldl (+) (((0 + 1) + 2) + 3) [4 .. 100]
```

```
foldl’ (+) (0 + 1) [2 .. 100]
foldl’ (+) (1 + 2) [3 .. 100]
foldl’ (+) (3 + 3) [4 .. 100]
```

References: [D5], [D6], [D8], [D9], [D10], [H10]
Example of foldl (non-strict) and foldl' (strict)

foldl (+) (0 + 1) [2 .. 100]

foldl (+) ((0 + 1) + 2) [3 .. 100]

foldl (+) (((0 + 1) + 2) + 3) [4 .. 100]

foldl' (+) (0 + 1) [2 .. 100]

foldl' (+) (1 + 2) [3 .. 100]

foldl' (+) (3 + 3) [4 .. 100]

References: [D5], [D6], [D8], [D9], [D10], [H10]
4. Evaluation

Controlling the evaluation
How to drive the evaluation

An expression

driving the evaluation by

and deconstructing by

pattern-matching
primitive operation
strict function
forcing function
special syntax
special pragma
compile option

case expression
function definition

+, *, ...
foldl'
scanl'
,...
seq
$!
pseq
deepseq
force
$!!
rnf
,...

* ghc 8.0 ~
Strict
StrictData
-O
-fstrictness
-XStrict
-XStrictData

4. Evaluation
(1) Evaluation by pattern-matching

pattern-matching in case expression

```
case ds of
  x:xs -> f x xs
  []  -> False
```

forcing (drive the evaluation of the thunk)

pattern-matching in function definition

```
f Just _  = True
f Nothing = False
```

forcing (drive the evaluation of the thunk)

References: [H1] Ch.3, [D2], [D1], [H5], [W1]
(1) Evaluation by pattern-matching

**Strict patterns** drive the evaluation

**Lazy patterns** postpone the evaluation.

**case expression**

\[
\text{case } ds \text{ of } \\
\quad x:xs \rightarrow f \times xs \\
\quad [] \rightarrow \text{False}
\]

**function definition**

\[
f \text{ Just } _{\_} = \text{True} \\
f \text{ Nothing} = \text{False}
\]

**let binding pattern**

\[
\text{let } (x:xs) = \text{fun } \text{args}
\]

**irrefutable patterns** [H1] 3.17

\[
f \sim(\text{Just } _{\_}) = \text{True} \\
f \sim(\text{Nothing}) = \text{False}
\]

There are two kinds of pattern-matching.
4. Evaluation

(2) Evaluation by primitive operation

primitive (built-in) operation

\[ f(x, y) = x + y \]

forcing \( x \) and \( y \)
(drive the evaluation of the thunks)

primitive operations are defined such as

\[ (+)(I\# a)(I\# b) = I\# (a+b) \]

pattern-matching

References: [D3], [H5], [H12]
(3) Evaluation by strict version function

strict version function

\[
\text{foldl}' (+) 0 \text{ xs}
\]

\[
\text{scanl}' (+) 0 \text{ xs}
\]

References: [B4] Ch.25, [D9], [B6] Ch.7, [B2], [S1], [S2]
(4) Evaluation by forcing function

forcing functions to WHNF

- `seq x y`
- `f $! x`
- `pseq x y`

forcing functions to NF

- `deepseq x y`
- `f $!! x`
- `force x`
- `rnf x`

Forcing (drive the evaluation of the thunk)

References: [B5] Ch.2, [B4] Ch.24, 25, [B6] Ch.7, [H1] Ch.6, [D2], [B2], [D1], [D2]. [S1], [S2]
4. Evaluation

(4) Evaluation by forcing function

References: [B5] Ch.2, [B4] Ch.24, 25, [H1] Ch.6, [D2], [B2], [D1], [D2], [S1], [S2]
### (4) Evaluation by forcing function

<table>
<thead>
<tr>
<th></th>
<th>to WHNF</th>
<th>to NF</th>
</tr>
</thead>
<tbody>
<tr>
<td>two arguments</td>
<td>seq</td>
<td>deepseq</td>
</tr>
<tr>
<td>one argument</td>
<td>force</td>
<td></td>
</tr>
<tr>
<td>function application</td>
<td>$!$</td>
<td>$!!$</td>
</tr>
<tr>
<td>sequential order</td>
<td>pseq</td>
<td></td>
</tr>
<tr>
<td>basic operation</td>
<td></td>
<td>rnf</td>
</tr>
</tbody>
</table>

References: [B5] Ch.2, [B4] Ch.24, 25, [H1] Ch.6, [D2], [B2], [D1], [D2]. [S1], [S2]
(4) Evaluation by forcing function

4. Evaluation

References: [B5] Ch.2, [B4] Ch.24, 25, [H1] Ch.6, [D2], [B2], [D1], [D2]. [S1], [S2]
4. Evaluation

(4) Evaluation by forcing function

\[ a = \text{map abs} \ [1, -2, 3, -4] \]

\[ \text{seq a} () \]

\[ \text{deepseq a} () \]

\[ \text{length a} \]

References: [B5] Ch.2, [B4] Ch.24, 25, [H1] Ch.6, [D2], [B2], [D1], [D10], [S1], [S2]
4. Evaluation

(5) Evaluation by special syntax

**Strictness annotation**

**Bang pattern** [H2] 7.19

{-# LANGUAGE BangPatterns #-}

\[
f \bang-pattern \text{ xs } = g \text{ xs}
\]

Arguments are evaluated before function application.

See also **Strict pragma**

**Strictness flag** [H1] 4.2.1

\[
data \text{ Pair } = \text{ Pair } \bang \text{ a } \bang \text{ b}
\]

Arguments are evaluated before constructor application.

See also **StrictData and Strict pragma**

Strictness annotations assist strictness analysis.

4. Evaluation

(6) Evaluation by special pragma

Special pragma for strictness language extension

**Strict pragma**

```haskell
{-# LANGUAGE Strict #-}

let f xs = g xs in f ys

data Pair = Pair a b
```

Arguments are evaluated before application.

**StrictData pragma**

```haskell
{-# LANGUAGE StrictData #-}

data Pair = Pair a b
```

See also bang pattern and strictness flag.

Strict and StrictData pragmas are module level control. These can use in ghc 8.0 or later.

References: [H13]
(7) Evaluation by compile option

Compile option

strictness analysis

$ ghc -O

$ ghc -fstrictness

Turn on optimization.
Imply "-fstrictness".

Turn on strictness analysis.
Implied by "-O".

strictness language extension

* ghc 8.0 ~

$ ghc -XStrict

apply Strict pragma

$ ghc -XStrictData

apply StrictData pragma

References: [H2], [H13]
5. Implementation of evaluator
5. Implementation of evaluator

Lazy graph reduction
An expression can be represented in the form of Abstract Syntax Tree (AST). AST is reduced using stack (sequential access memory).
An expression can be also represented in the form of Graph. Graph can share subexpressions to evaluate at once. So, graph is reduced using heap (random access memory) rather than stack.
Graph can be reduced in some order

To select top-level redex first, the evaluation of arguments can be postponed.

References: [D3], [W5], [H4] Ch.11, 12, [B8] Ch.3
Normal order reduction is implemented by lazy graph reduction.

Normal order (leftmost outermost) first

Lazy graph reduction

find top-level redex and reduce it

Normal order (leftmost outermost) reduction is implemented by lazy graph reduction to select top-level redex first.

Given an application of a function, the outermost redex is the function application itself.

References: [D3], [D2], [D5], [W5], [H4] Ch.11, 12, [B8] Ch.3
5. Implementation of evaluator

STG-machine
**Abstract machine**

GHC uses abstract machine to reduce the expression. It’s called “STG-machine”.

References: [H5], [H6], [H7], [D15]
Concept layer

Haskell code

```
take 5 [1..10]
```

Graph
(internal representation of the expression)

Evaluator (reducer, executer)
(abstract machine)

References: [H5], [H6], [H7], [D15]
STG-machine is abstraction machine which is defined by operational semantics.

STG-machine efficiently performs lazy graph reduction.

References: [H5], [H6], [H7], [D15]
5. Implementation of evaluator

**STG-machine**

- **STG Registers**: mainly used for call/return convention, various control.
- **Stack**: mainly used for nest continuation, argument passing.
- **Heap**: mainly used for allocating objects (thunks, datas, functions).
- **Static**: mainly used for code, static objects.

References: [H5], [H6], [H7], [D15]
Example of mapping a code to a graph

main = print (head [1..])
Example of mapping a code to a graph

main = print (head [1..])
Example of mapping a code to a graph

main = print (head [1..])

References: [H5], [H10]
Self-updating model

5. Implementation of evaluator

GHC's internal representation

- a thunk
- expression code
- update code
- free variables
- evaluate and update (replace myself to result value)

Expression

- a data value
- info ptr
- constructor
- data components

References: [H5], [H6], [H7], [D15]
5. Implementation of evaluator

Unreferenced expressions (objects) will be removed by GC

References: [H5], [H6], [H4] Ch.12, [D15]
STG-machine associates directly lambda calculus and physical machine.
The STG-machine is the marriage of Lambda calculus and Turing machine.

The STG-machine is the marriage of Lambda calculus and Turing machine.
STG-dump shows which expression is built as thunks

```haskell
module Example where

fun f1 n = take 1 (f1 n)

Example.fun :: forall a t. (t -> [a]) -> t -> [a]

let { sat [Occ=Once, Dmd=<L,1*U>] :: [aMH] } in GHC.List.take_unsafe_UInt 1 sat
```

References: [H5], [H6], [H7], [D15]
6. Semantics
6. Semantics

Bottom
A well formed expression should have a value

An expression

1 + 2

evaluate

3

A value

What is a value in this case?

- An expression
- a non-terminating expression
- infinite loop or partial function

evaluate

?  A value ?

References: [B2] Ch.2, [H1] Ch.3, [W4], [H4] Ch.2, 22
A value “bottom” is introduced

An expression

infinite loop or partial function

evaluate

A value

a non-terminating expression

References: [B2] Ch.2, [H1] Ch.3, [W4], [H4] Ch.2, 22
Bottom

A value

\(\bot\)

Bottom (\(\bot\)) is “an undefined value”.
Bottom (\(\bot\)) is “a non-terminating value”.

References: [B2] Ch.2, 9, [H1] Ch.3, [W4], [H4] Ch.2, 22
### “undefined” function represents “bottom” in GHC

<table>
<thead>
<tr>
<th>Haskell code</th>
<th>Expression</th>
<th>GHC's internal representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>undefined :: a</td>
<td>⊥</td>
<td>GHC.Err.undefined</td>
</tr>
</tbody>
</table>

References: [B2] Ch.2, 9, [H1] Ch.3, [W4], [H4] Ch.2, 22
6. Semantics

Strict/Non-strict
6. Semantics

**Strictness**

Strictness is “evaluation demand” of the expression.

References: [B2] Ch.2, [W1], [W4], [H4] Ch.2, 22, [H15], [H16]
6. Semantics

Strict and non-strict

"Non-strict" means that the expression may or may not be evaluated. "Strict" means that the expression is definitely evaluated.

References: [B2] Ch.2, [W1], [W4], [H4] Ch.2, 22, [H15], [H16]
Strict and non-strict

"Non-strict" means that the expression *may or may not be evaluated.*

"Strict" means that the expression is *definitely evaluated.*

GHC implements non-strict semantics by lazy evaluation.

References: [B2] Ch.2, [W1], [W4], [H4] Ch.2, 22, [H15], [H16]
GHC has the lattice of strictness

- **Non-strict**
  - a lazy demand
  - a head-strict demand
  - a structured strictness demand
  - a hyperstrict demand

- **Strict**

There are multiple levels in strict.

References: [H15], [H16], [B2] Ch.2, [W1], [W4], [H4] Ch.2, 22
6. Semantics

Strictness of a function

A function places “strictness demands” on each of its arguments.

References: [H15], [H16], [B2] Ch.2, [W1], [W4], [H4] Ch.2, 22
Strictness of a function can be defined with the association between input and output.

“given a non-terminating arguments, the function will terminate?”

References: [B2] Ch.2, [W1], [W4], [H4] Ch.2, 22
Definition of the strict function

Strict function's output is bottom when input is bottom.

given a non-terminating arguments, strict function will not terminate.
Definition of the non-strict function

Non-strict function's output is **not** bottom when input is bottom.

given a non-terminating arguments, non-strict function will terminate.

References: [B2] Ch.2, [W1], [W4], [H4] Ch.2, 22
Strict and Non-strict functions

Non-strict

Strict

References: [B2] Ch.2, [W1], [W4], [H4] Ch.2, 22
Function application and strictness

Non-strict

- Build a thunk
- Passing the thunk
- Evaluate the argument if needed

Strict

- Evaluate the argument to WHNF
- Passing the evaluated argument
- No need to evaluate the argument

The front stage is also important.

References: [B2] Ch.2, [W1], [W4], [H4] Ch.2, 22
Strict and normal form

Example of function application

<table>
<thead>
<tr>
<th></th>
<th>to WHNF</th>
<th>to NF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-strict</td>
<td>f $ arg</td>
<td></td>
</tr>
<tr>
<td>Strict</td>
<td>f $! arg (seq)</td>
<td>f $!! arg (deepseq)</td>
</tr>
</tbody>
</table>

Strict  ≠  Normal form

References: [B2] Ch.2, [W1], [W4], [H4] Ch.2, 22
Lifted and boxed types
Lifted types include bottom as an element.
Lifted type's declaration implicitly include bottom

**Bool type**
- `False`
- `True`
- `⊥`

**Int type**
- `0`
- `1`
- `2`
- `...`
- `⊥`

**data declaration**
- `data Bool = False`
- `| True`
- `| ⊥`

- `data Int = I# Int#`
- `| ⊥`

References: [W4], [B2], [H14]
Lifted type are also implemented by uniform representation

data declaration

```
data Maybe a = Nothing
  | Just a
  | ⊥
data Int = I# Int#
  | ⊥
```

GHC's internal representation

```
x :: Maybe a

Nothing

Just a

⊥

undefined code
```

```
x :: Int

I# 0#

I# 1#

⊥

undefined code
```

References: [W4], [B2], [H14]
Lifted and unlifted types

**Lifted types**
- Int type
  - 0, 1, 2, ...
  - Bottom (⊥)
- Lifted types include bottom.
  - (Bool, Int, Char, Maybe, List, ...)

**Unlifted types**
- Int# type
  - 0#, 1#, 2#, ...
  - Bottom (⊥)
- Unlifted types do not include bottom.
  - (Int#, Char#, Addr#, Array#, ByteArray#, ...)

References: [W4], [B2], [H14]
Example of lifted and unlifted types

### Lifted types

- **x :: Just a**
  - Nothing
  - Just a
  - ⊥
  - code

- **x :: Int**
  - I# 0#
  - I# 1#
  - ⊥
  - code

### Unlifted types

- **x :: Array#**
  - data1
  - data2
  - data3
  - no bottom

- **x :: Int#**
  - #7
  - no bottom

References: [W4], [B2], [H14]
6. Semantics

Boxed and unboxed types

Boxed types are represented as a pointer.

Unboxed types are represented other than a pointer.

- no bottom (can't be lifted)
- no thunk (can't be postponed)
- no polymorphism (non-uniform size)
+ low cost memory size (no pointer)
+ high performance (no wrap/unwrap)

References: [W4], [B2], [H14]
Example of boxed and unboxed types

Boxed types

Unboxed types

References: [W4], [B2], [H14]
Lifted and boxed types

<table>
<thead>
<tr>
<th>Lifted types</th>
<th>Unlifted types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Boxed types</strong></td>
<td><strong>Unboxed types</strong></td>
</tr>
<tr>
<td>Int</td>
<td>Array#</td>
</tr>
<tr>
<td>Char</td>
<td>ByteArray#</td>
</tr>
<tr>
<td>Float</td>
<td>:</td>
</tr>
<tr>
<td>Maybe</td>
<td>no bottom</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Int#</td>
<td>Char#</td>
<td>Float#</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>no bottom no packed</td>
</tr>
</tbody>
</table>

unboxed can't be lifted

References: [W4], [B2], [H14]
Example of lifted and boxed types

**Lifted types**

- `x :: Int` with boxed types
  - `⊥` (include bottom)
  - `code`
  - Pointed

**Unlifted types**

- `x :: Array#`
  - `data1`
  - `data2`
  - `data3`

**Unboxed types**

- `x :: Int#`
  - `#7`

References: [W4], [B2], [H14]
Types and kinds

Lifted types

Unlifted types

Boxed types

Unboxed types

Int
Char
Float
Maybe

Array#
ByteArray#

Int#
Char#
Float#

kind ‘∗’

kind ‘#’

Note:
Identifier’s ‘#’ customarily means “primitive” rather than “unboxed” or “unlifted”.
Kind’s ‘#’ means “unlifted”.

References: [B6] Ch.29, [W4], [B2], [H14]
6. Semantics

Strictness analysis
Strictness analysis analyzes whether a function is sure to evaluate its argument.
GHC’s demand analyser implements strictness analysis.

References: [H15], [H16], [H2], [H4] Ch.22, [H8], [W6], [W3], [H13]
Example of strictness analysis information in GHC

[Example.hs]

module Example where

f1 :: Bool -> Int -> Maybe Int
f1 c n = case c of
  True -> Just n
  False -> Nothing

Strictness analysis dump
by "$ ghc -O -ddump-strsigs Example.hs"

==================================== Strictness signatures ======================
Example.f1: <S,1^U><L,U>

L -- second argument is “Lazy”

S -- first argument is “head-Strict”

GHC shows strictness analysis information with “-ddump-strsigs” and “-ddump-stranal”.

References : [H15], [H16], [H2], [H4] Ch.22, [H8], [W6], [W3], [H13]
(1) Strictness analysis are used to avoid the thunk

If GHC knows that a function is strict, arguments is evaluated before application.

GHC finds strict functions by “strictness analysis (demand analysis)”. 

References: [H4] Ch.22, [H8], [W6], [W3], [H15], [H17], [H13]
(1) Strictness analysis are used to avoid the thunk

If GHC knows that a function is strict, GHC performs let-to-case transformation.

References: [H8], [H4] Ch.22, [W6], [W3], [H15], [H17], [H13]
(2) Strictness analysis are also used to optimize

Strict function

(an argument) evaluate the argument

Strictness function can be optimized to assume no thunk, no bottom.

References: [H4] Ch.22, [H8], [W6], [W3], [H15], [H17], [H13]
Strictness analysis are also used to optimize

Strict function

\( f \)

(2) Strictness function can be optimized to assume no thunk, no bottom, no packed.

References: [H4] Ch.22, [H8], [W6], [W3], [H15], [H17], [H13]
6. Semantics

Sequential order
“seq” doesn’t guarantee the evaluation order

specification

\[
\text{seq } a \ b = \bot, \quad \text{if } a = \bot \\
= b, \quad \text{otherwise}
\]

strictness for each arguments

\[
\text{seq } \bot \ b = \bot \quad // \ a \text{ is strict} \\
\text{seq } a \ \bot = \bot \quad // \ b \text{ is strict}
\]

“seq” function only guarantee that it is strict in both arguments. This semantics property makes no operational guarantee about order of evaluation.
Both of denotational semantics are the same. But “pseq” makes operational guarantee about order of evaluation.
Evaluation order of “seq” and “pseq”

seq a b

Evaluation of a

Evaluation of b

pseq a b

Evaluation of a

Evaluation of b

References : [H9], [D11], [H1] Ch.6, [S1]
implementation of “seq” and “pseq”

specification

\[
seq \ a \ b = \bot, \quad \text{if} \ a = \bot = b, \quad \text{otherwise}
\]

Haskell’s built-in

specification

\[
pseq \ a \ b = \bot, \quad \text{if} \ a = \bot = b, \quad \text{otherwise}
\]

\[
pseq \ x \ y = x \ `\seq\` \ \text{lazy} \ y
\]

GHC’s “lazy” function restrains the strictness analysis.

“seq” is built-in function.
“pseq” is implemented by built-in functions (“seq” and “lazy”).

References: [H9], [D11], [H1] Ch.6, [H2] Ch.7, [S1]
7. Appendix
7. Appendix

References
References

[H1] Haskell 2010 Language Report

[H2] The Glorious Glasgow Haskell Compilation System (GHC user's guide)

[H3] A History of Haskell: Being Lazy With Class

[H4] The implementation of functional programming languages


[H6] Making a Fast Curry Push/Enter vs Eval/Apply for Higher-order Languages

[H7] Faster Laziness Using Dynamic Pointer Tagging

[H8] Measuring the effectiveness of a simple strictness analyser
http://research.microsoft.com/~simonpj/papers/simple-strictness-analysers.ps.gz

[H9] Runtime Support for Multicore Haskell

[H10] I know kung fu: learning STG by example
https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/GeneratedCode
References

[H11] GHC Commentary: The Layout of Heap Objects
https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/HeapObjects

[H12] The ghc-prim package
https://hackage.haskell.org/package/ghc-prim

[H13] GHC Commentary: Strict & StrictData
https://ghc.haskell.org/trac/ghc/wiki/StrictPragma

[H14] The data type Type and its friends
https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/TypeType

[H15] Demand analyser in GHC
https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/Demand

[H16] Demand analysis

[H17] Core-to-Core optimization pipeline
https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/Core2CorePipeline

[H18] The GHC reading list
https://ghc.haskell.org/trac/ghc/wiki/ReadingList

[H19] The GHC Commentary
https://ghc.haskell.org/trac/ghc/wiki/Commentary
References

[B1] Introduction to Functional Programming using Haskell (IFPH 2nd edition)

[B2] Thinking Functionally with Haskell (IFPH 3rd edition)
http://www.cs.ox.ac.uk/publications/books/functional/

[B3] Programming in Haskell

[B4] Real World Haskell
http://book.realworldhaskell.org/

[B5] Parallel and Concurrent Programming in Haskell
http://chimera.labs.oreilly.com/books/12300000000929

[B6] Types and Programming Languages (TAPL)
https://mitpress.mit.edu/books/types-and-programming-languages

[B7] Purely Functional Data Structures

http://catalogue.pearsoned.co.uk/catalog/academic/product/0,1144,0201596040,00.html
References

[D1] Laziness
http://dev.stephendiehl.com/hask/#laziness

[D2] Being Lazy with Class
http://www.seas.upenn.edu/~cis194/lectures/06-laziness.html

[D3] A Haskell Compiler
http://www.scs.stanford.edu/11au-cs240h/notes/ghc-slides.html

[D4] Evaluation
http://dev.stephendiehl.com/fun/005_evaluation.html

[D5] Incomplete Guide to Lazy Evaluation (in Haskell)
https://hackhands.com/guide-lazy-evaluation-haskell

[D6] Laziness
https://www.fpcomplete.com/school/starting-with-haskell/introduction-to-haskell/6-laziness

[D7] Evaluation on the Haskell Heap

[D8] Lazy Evaluation of Haskell
http://www.vex.net/~trebla/haskell/lazy.xhtml

[D9] Fixing foldl
http://www.well-typed.com/blog/2014/04/fixing-foldl

[D10] How to force a list
http://ro-che.info/articles/2015-05-28-force-list
References

[D11] Evaluation order and state tokens

[D12] Reasoning about laziness

[D13] Some History of Functional Programming Languages

    https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf

[D15] GHC illustrated
References

[W1] Haskell/Laziness
    https://en.wikibooks.org/wiki/Haskell/Laziness

[W2] Lazy evaluation
    https://wiki.haskell.org/Lazy_evaluation

[W3] Lazy vs. non-strict
    https://wiki.haskell.org/Lazy_vs__non-strict

[W4] Haskell/Denotational semantics
    https://en.wikibooks.org/wiki/Haskell/Denotational_semantics

[W5] Haskell/Graph reduction
    https://en.wikibooks.org/wiki/Haskell/Graph_reduction

[W6] Performance/Strictness
    https://wiki.haskell.org/Performance/Strictness
References

[S1] Hackage
    https://hackage.haskell.org

[S2] Hoogle
    https://www.haskell.org/hoogle
Lazy,...  

*to be as lazy as possible...*