

Lazy evaluation in Haskell

exploring some mental models and implementations

Takenobu T.

Lazy,... zzz

..., It's fun.

NOTE

- Meaning of terms are different by communities.
- There are a lot of good documents. Please see also references.
- This is written for GHC's Haskell.

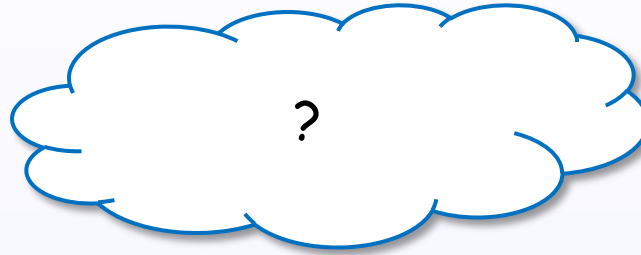
Contents

- Introduction
- Evaluation strategies
- Expressions in Haskell
- Constructor
- Thunk
- let, case expression
- WHNF
- Evaluation in Haskell (GHC)
- Control the evaluation in Haskell
- Examples of evaluation steps
- Semantics
- Graph reduction
- Implementation in GHC
- References

Introduction

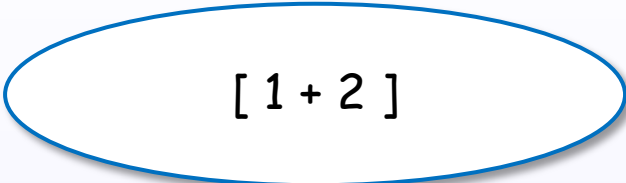
What is an expression?

An expression



An expression denotes a value

An expression



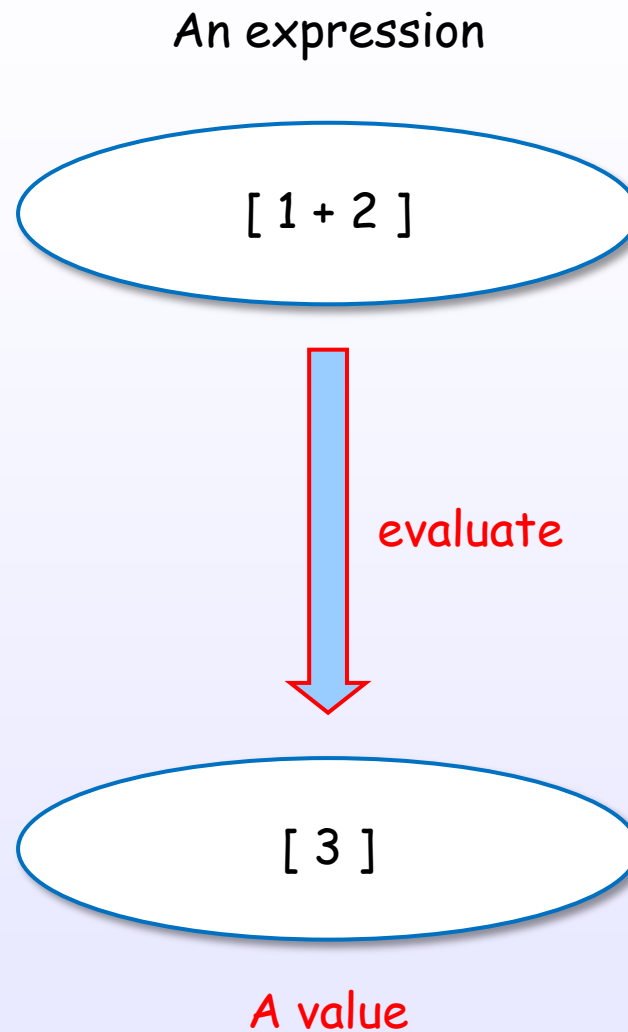
[1 + 2]

[HR2010]

[Bird, Chapter 2]

References : [1]

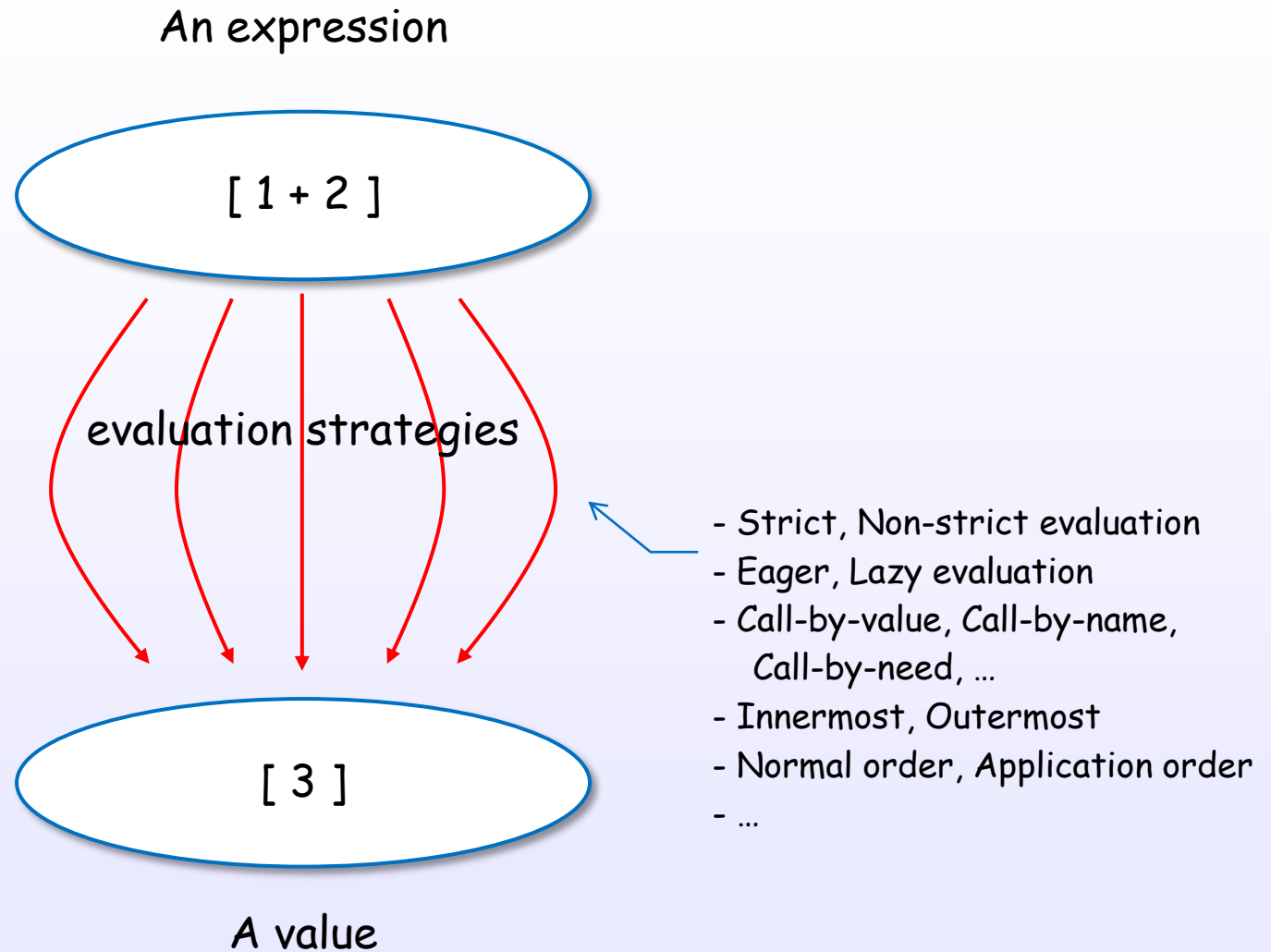
An expression evaluates to a value



[HR2010]

[Bird, Chapter 2]

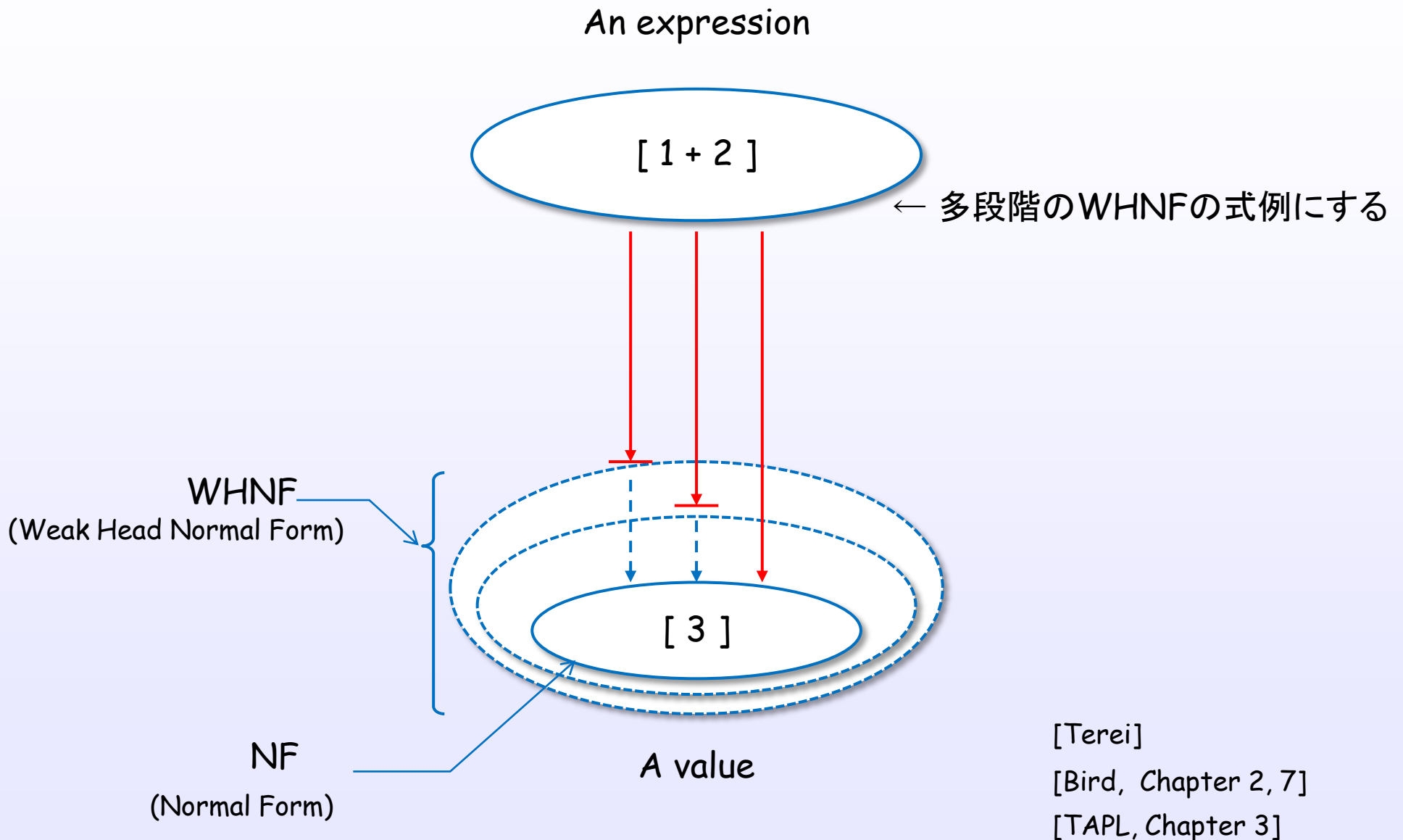
There are many evaluation approaches



[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

There are some evaluation levels

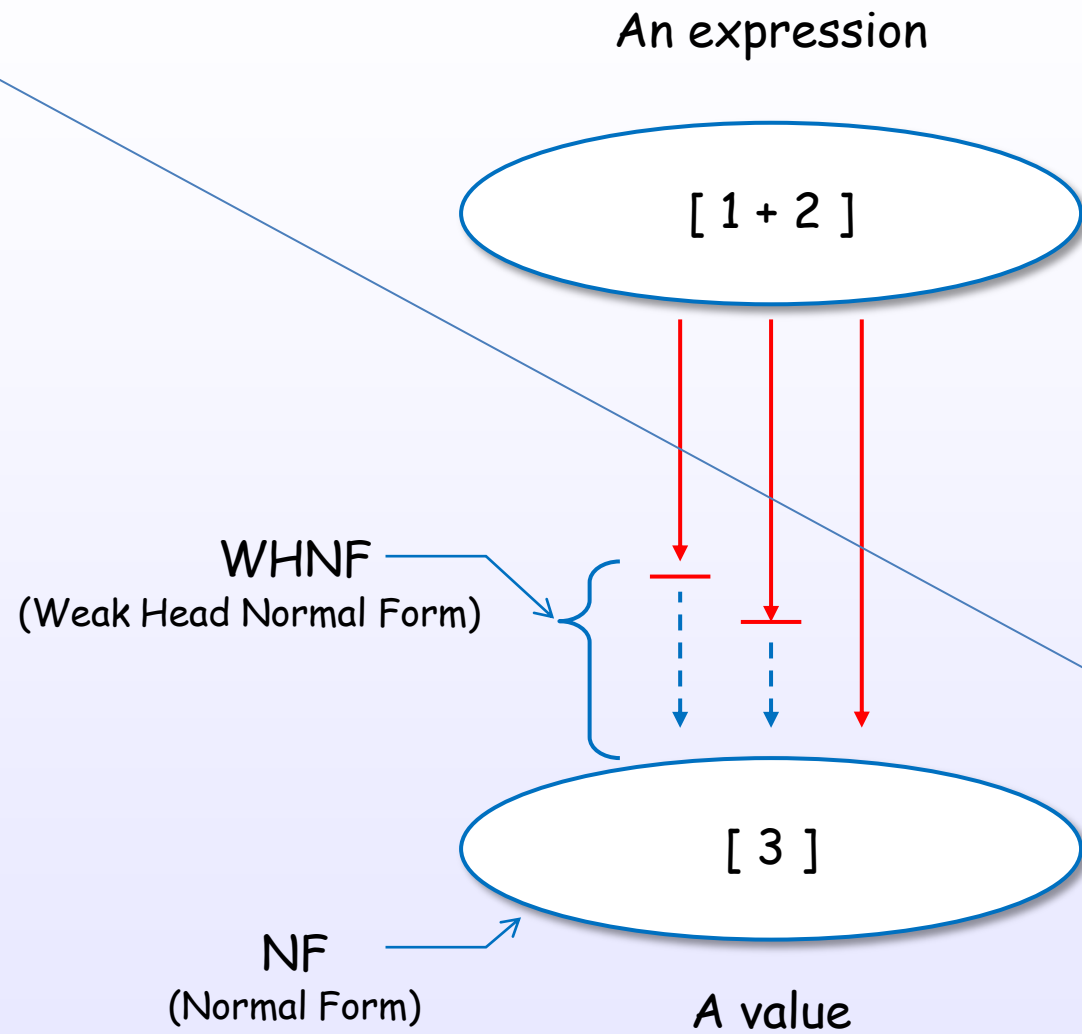


[Terei]

[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

There are some evaluation levels



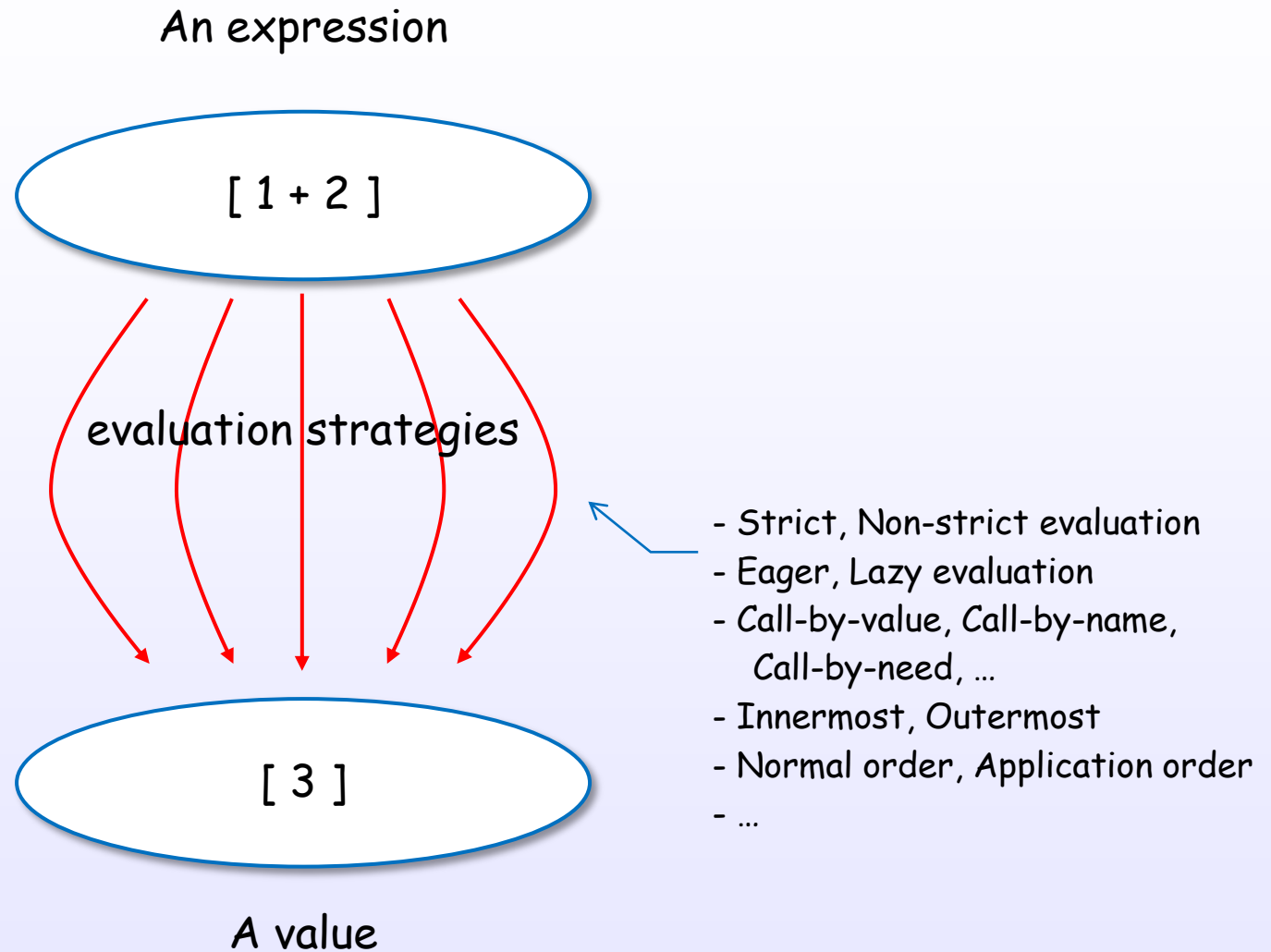
[Terei]

[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

Evaluation strategies

There are many evaluation approaches



[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

Evaluation layers

denotational semantics

evaluation strategies

implementation techniques

[Bird, Chapter 7]

[Hutton, Chapter 8]

[TAPL, Chapter 3]

References : [1]

Evaluation layers

denotational
semantics

Strict semantics

Non-strict semantics

evaluation
strategies

Eager evaluation
(Strict evaluation)

Nondeterministic
evaluation

Lazy evaluation
(Non-strict evaluation)

...

Call-by-Value

Call-by-Name

Call-by-Need

...

implementation
techniques

Graph reduction

...

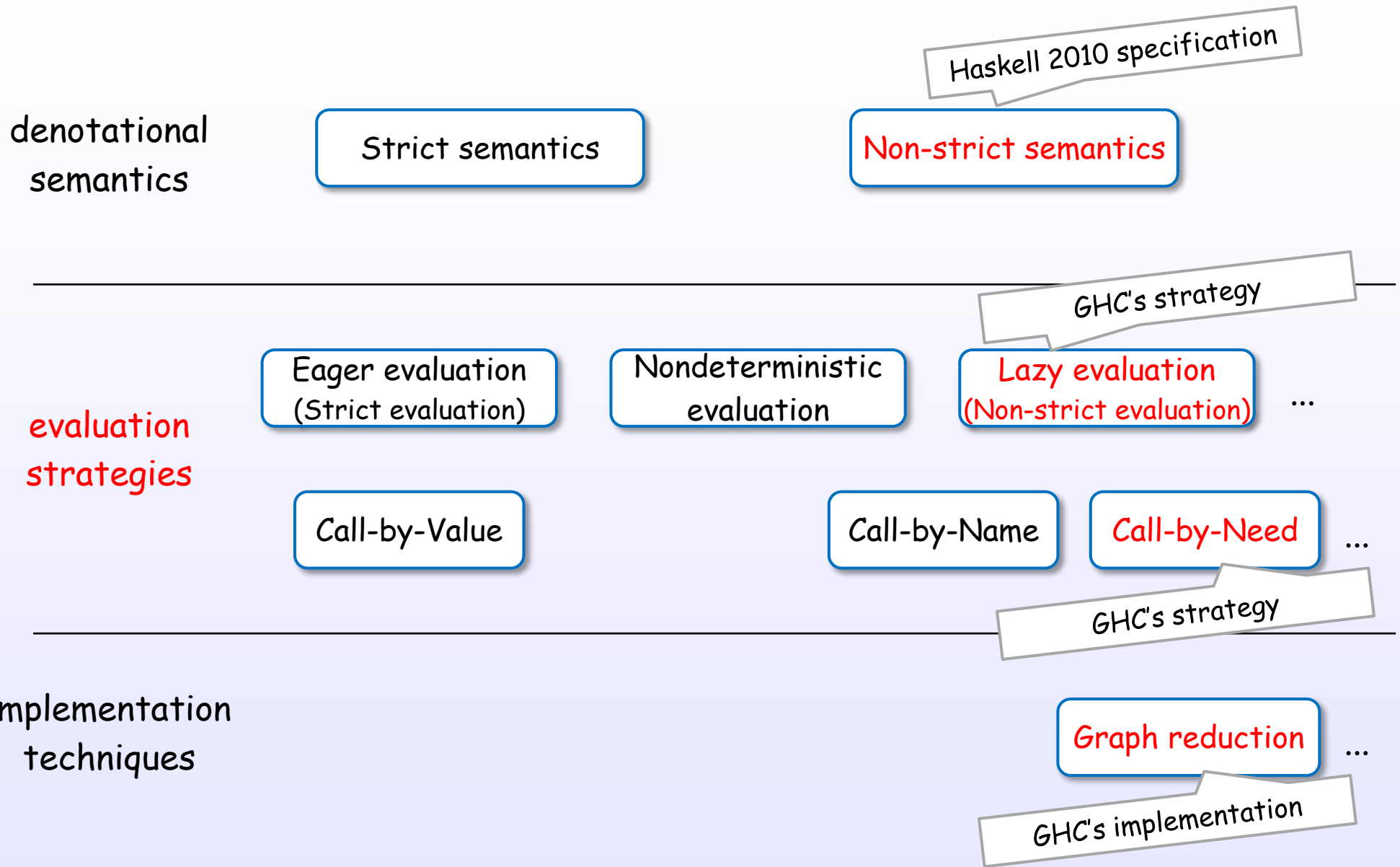
[Bird, Chapter 7]

[Hutton, Chapter 8]

[TAPL, Chapter 3]

References : [1]

Evaluation layers for GHC's Haskell



Evaluation strategies and order

$a(b\ c) + d(e\ (f\ g))$

order

[Bird]
[Hutton]

References : [1]

Simple example of both evaluations

Eager evaluation (Strict evaluation)

default
C, Java, JavaScript,
Python, OCaml, Scheme, ...

square (1 + 2)



argument
evaluation
first

square (3)



3 * 3



9

Lazy evaluation (Non-strict evaluation)

default
Haskell (GHC), ...

square (1 + 2)



apply
first

(1 + 2) * (1 + 2)



(3) * (3)



9

[Bird]
[Hutton]

Simple example of both evaluations

Eager evaluation
(Strict evaluation)

square (1 + 2)



square (3)



3 * 3



9

argument
evaluated

Lazy evaluation
(Non-strict evaluation)

square (1 + 2)



(1 + 2) * (1 + 2)



(3) * (3)



9

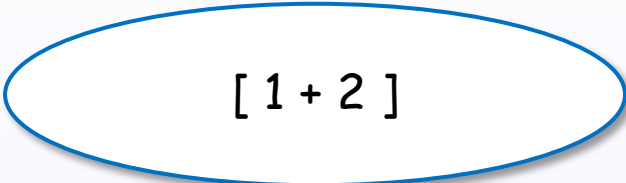
argument
evaluation
delayed !

[Bird]
[Hutton]

Expressions in Haskell

An expression denotes a value

An expression



[1 + 2]

[HR2010]

[Bird, Chapter 2]

References : [1]

There are many expressions in Haskell

Expressions

Just 5

$1 + 2$

$(1, 2)$

take 5 xs

$[1, 2, 3]$

let $x = 1$ in $x + y$

'a'

map f xs

if b then 1 else 0

7

$\forall x \rightarrow x + 1$

$x : xs$

fun arg

case x of $_ \rightarrow 0$

$(\forall x \rightarrow x + 1) 3$

do { $x \leftarrow$ get; put x}



categorizing

[HR2010]

[Bird, Chapter 2]

References : [1]

Expression categories in Haskell

lambda abstraction

$\forall x \rightarrow x + 1$

let expression

let $x = 1$ in $x + y$

conditional

if b then 1 else 0

case expression

case x of $_ \rightarrow 0$

do expression

do { $x \leftarrow \text{get}$; put x }

general constructor, literal and some forms

7

[1, 2, 3]

(1, 2)

'a'

$x : xs$

Just 5

function application

take 5 xs

$(\forall x \rightarrow x + 1)$ 3

1 + 2

map f xs

fun arg

[HR2010]
[Bird, Chapter 2]

Specification is defined in Haskell 2010 Language Report

Haskell 2010 Language Report, Chapter 3 Expressions [1]

<i>exp</i>	→	<i>infixexp</i> :: [context =>] type <i>infixexp</i>	(expression type signature)
<i>infixexp</i>	→	<i>lexp</i> <i>qop</i> <i>infixexp</i> - <i>infixexp</i> <i>lexp</i>	(infix operator application) (prefix negation)
<i>lexp</i>	→	\ <i>apat</i> ₁ ... <i>apat</i> _{<i>n</i>} -> <i>exp</i> let <i>decls</i> in <i>exp</i> if <i>exp</i> [<i>i</i>] then <i>exp</i> [<i>i</i>] else <i>exp</i> case <i>exp</i> of { <i>alts</i> } do { <i>stmts</i> } <i>fexp</i>	(lambda abstraction, $n \geq 1$) (let expression) (conditional) (case expression) (do expression)
<i>fexp</i>	→	[<i>fexp</i>] <i>aexp</i>	(function application)
<i>aexp</i>	→	<i>qvar</i> <i>gcon</i> <i>literal</i> (<i>exp</i>) (<i>exp</i> ₁ , ... , <i>exp</i> _{<i>k</i>}) [<i>exp</i> ₁ , ... , <i>exp</i> _{<i>k</i>}] [<i>exp</i> ₁ [, <i>exp</i> ₂] .. [<i>exp</i> ₃]] [<i>exp</i> <i>qual</i> ₁ , ... , <i>qual</i> _{<i>n</i>}] (<i>infixexp</i> <i>qop</i>) (<i>qop</i> { - } <i>infixexp</i>) <i>qcon</i> { <i>fbind</i> ₁ , ... , <i>fbind</i> _{<i>n</i>} } <i>aexp</i> _{<i>qcon</i>} { <i>fbind</i> ₁ , ... , <i>fbind</i> _{<i>n</i>} }	(variable) (general constructor) (parenthesized expression) (tuple, $k \geq 2$) (list, $k \geq 1$) (arithmetic sequence) (list comprehension, $n \geq 1$) (left section) (right section) (labeled construction, $n \geq 0$) (labeled update, $n \geq 1$)

Constructor

Constructor

Constructor is one of the key elements to understand WHNF and lazy evaluation.

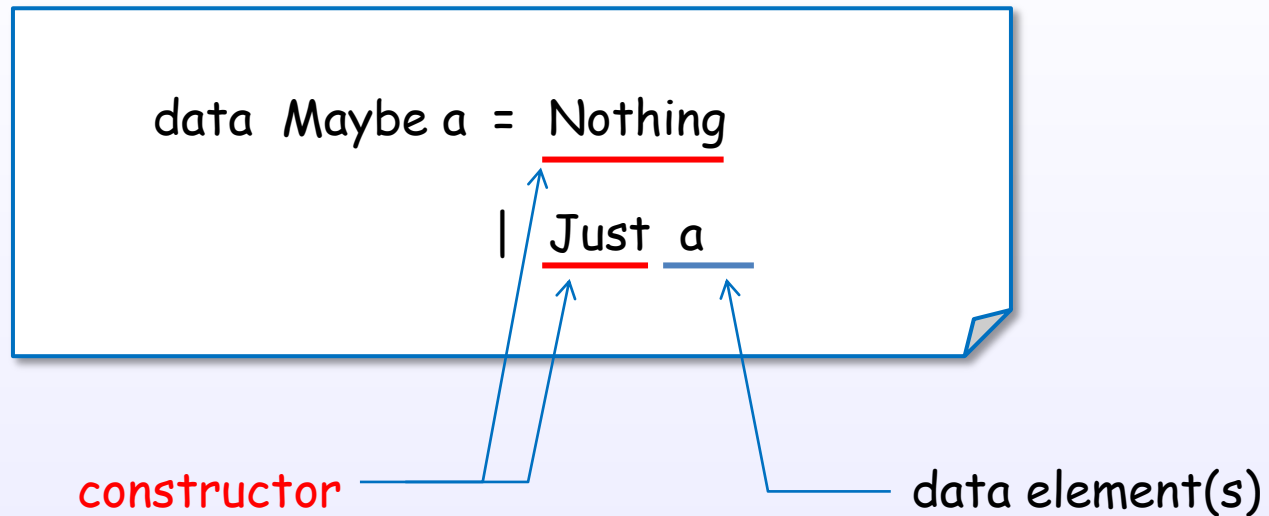
data文で宣言する代数的データ型とその値

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

Algebraic Data Type

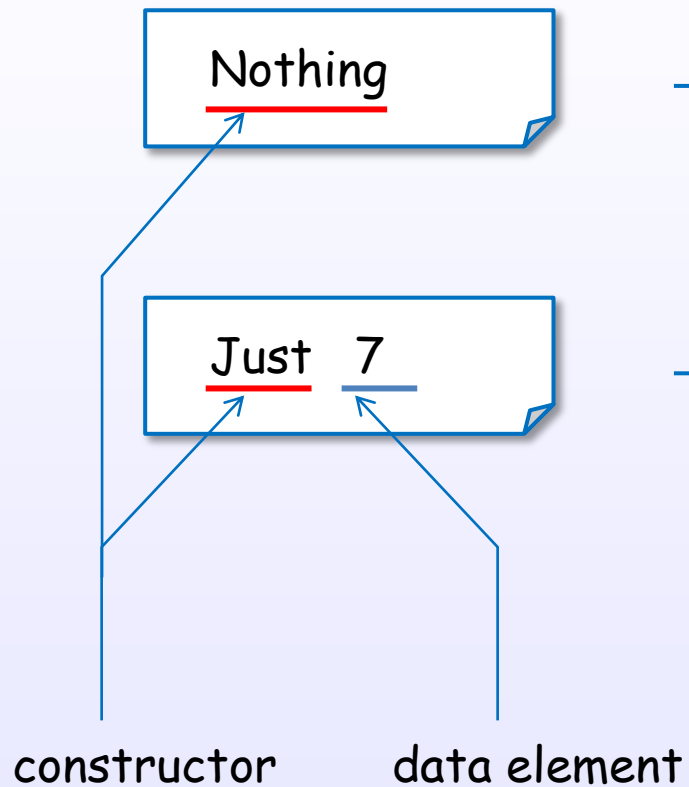
Algebraic Data **Value**

Constructorはdata文で宣言する代数的データ値

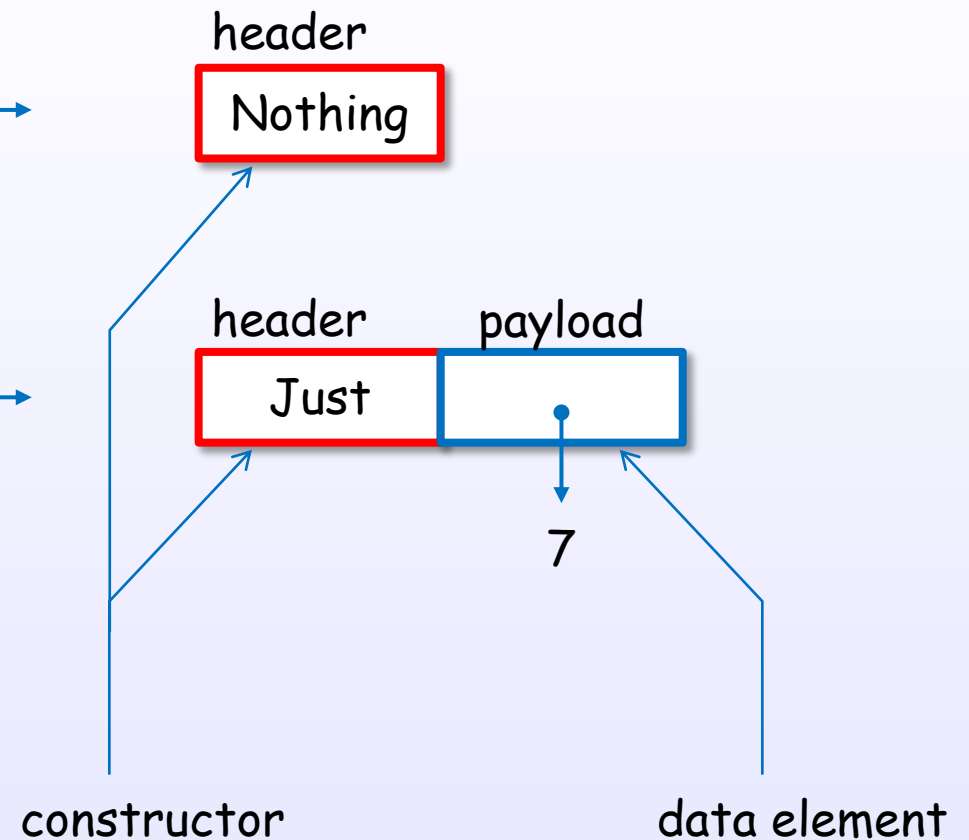


Constructorの内部表現

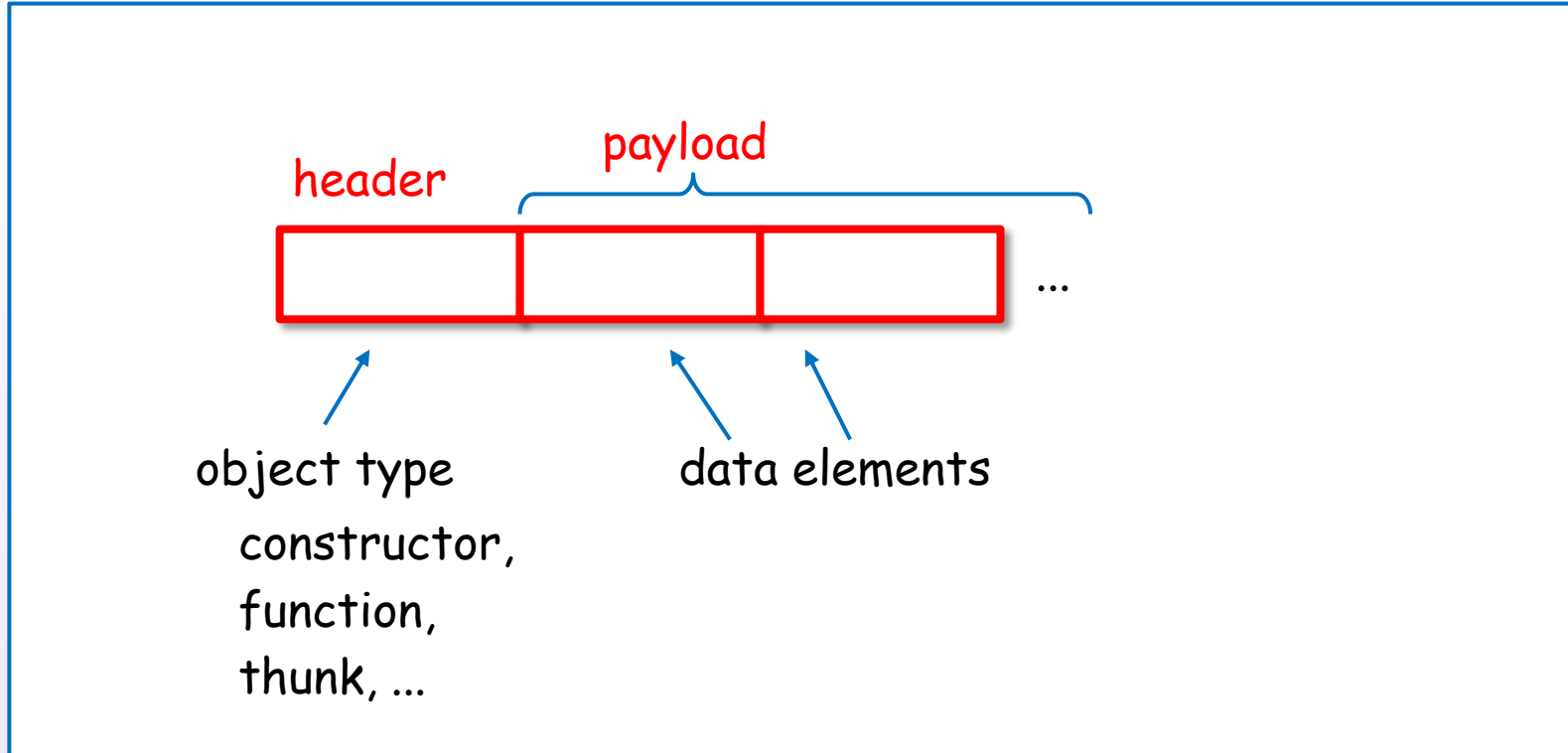
Haskell code



GHC's internal representation
in heap memory



Constructorは統一内部表現で表現される



in heap memory, stack, registers or static memory

いろいろなコンストラクタと内部表現

Haskell code

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

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

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

GHC's internal representation

False

True

Nothing

Just

a

Left

a

Right

b

基本データ型も実はコンストラクタで構成されている

Haskell code

```
data Int = Int# 0#  
        | Int# 1#  
        | :  
        | :
```

```
data Char = Char# 'a'#  
          | Char# 'b'#  
          | :  
          | :
```

GHC's internal representation

→

Int#	0#
Int#	1#
:	

→

Char#	'a'#
Char#	'b'#
:	

リストも実はコンストラクタで構成されている

List

[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))

constructor

リストも実はコンストラクタで構成されている

List

```
[ 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 ) )
```

** pseudo code*

```
data List a = []  
             | : a (List a)
```

```
data List a = Nil  
             | Cons a (List a)
```

リストも実はコンストラクタで構成されている

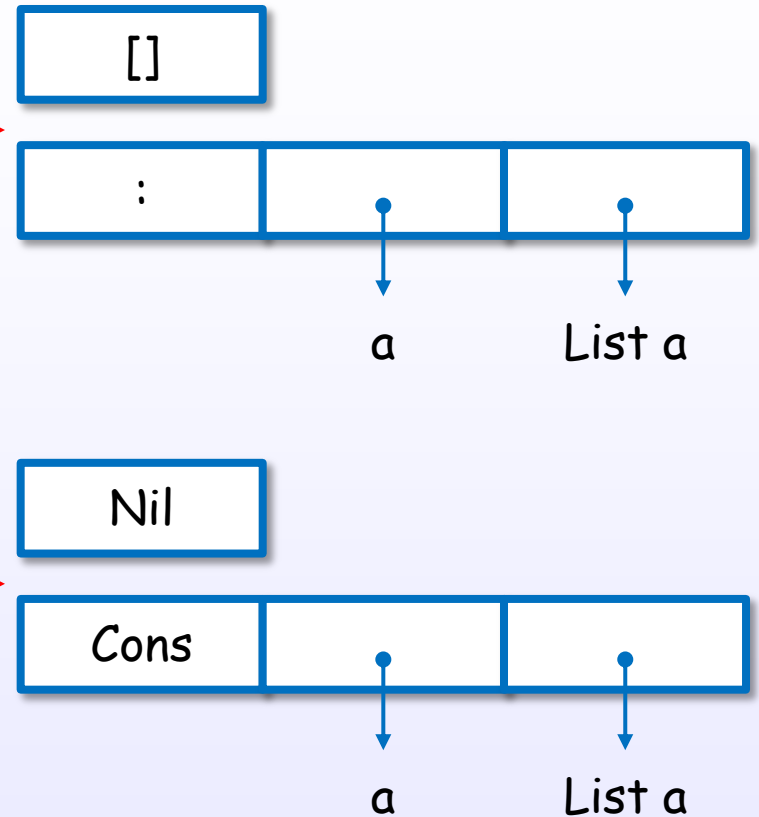
Haskell code

```
data List a = []  
           | : a (List a)
```

↕ equivalent data constructor

```
data List a = Nil  
           | Cons a (List a)
```

GHC's internal representation



リストも実はコンストラクタで構成されている

Haskell code

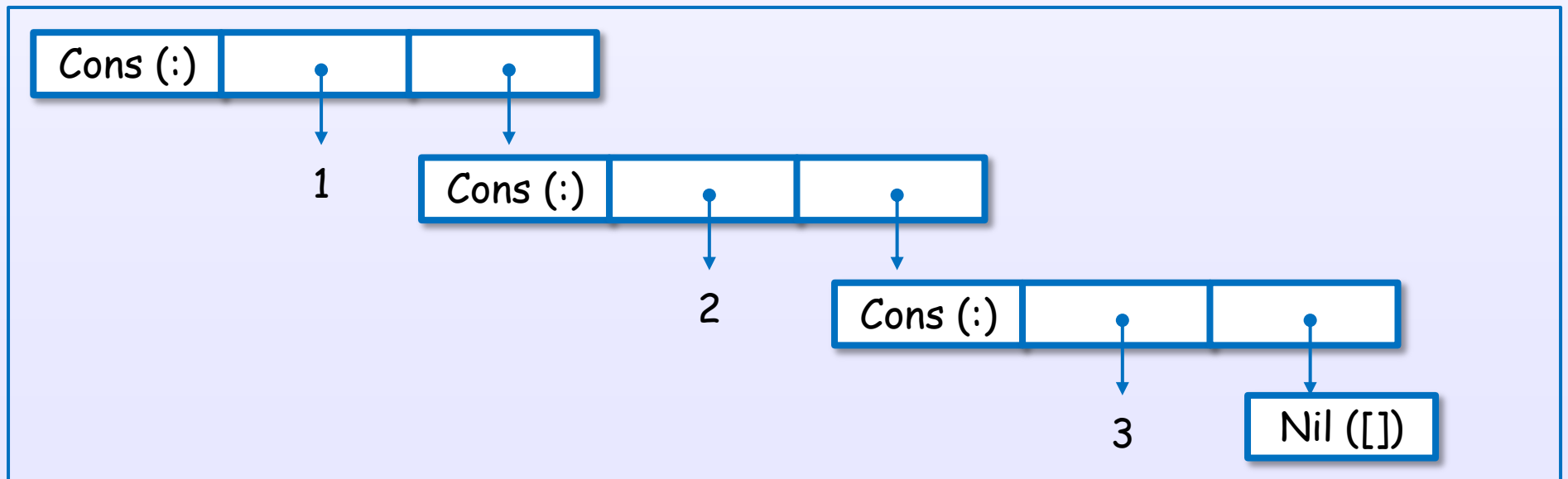
```
[ 1, 2, 3 ]
```

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

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

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

GHC's internal representation



タプルも実はコンストラクタで構成されている

Tuple (Pair)

(7 , 8)

prefix notation by section

(,) 7 8

equivalent data constructor

Pair 7 8

constructor

** pseudo code*

data Pair a = (,) a a

data Pair a = Pair a a

タプルも実はコンストラクタで構成されている

Haskell code

`(7, 8)`



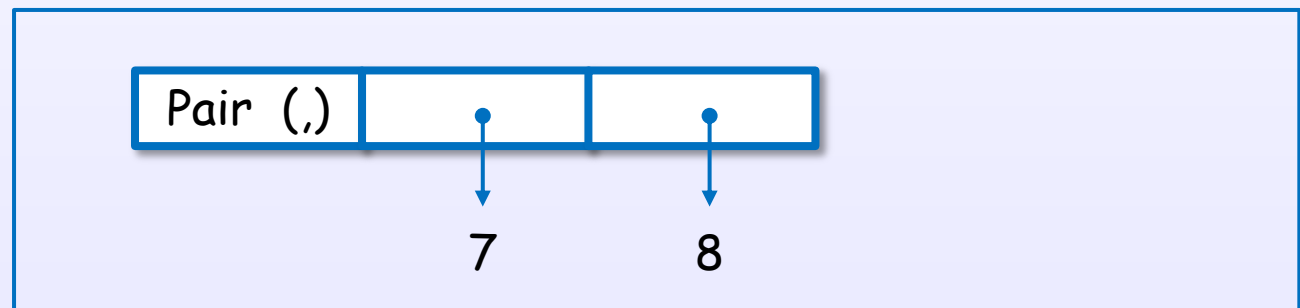
`(,) 7 8`



`Pair 7 8`



GHC's internal representation



Thunk

Thunk

think
(unevaluated expression/
suspended computation)

A thunk is an **unevaluated** expression in heap memory.

Thunk

An unevaluated expression

take x ys



create/allocate

thunk
(unevaluated expression/
suspended computation)

in heap memory

A thunk is created for an unevaluated expression.

Thunkの内部表現

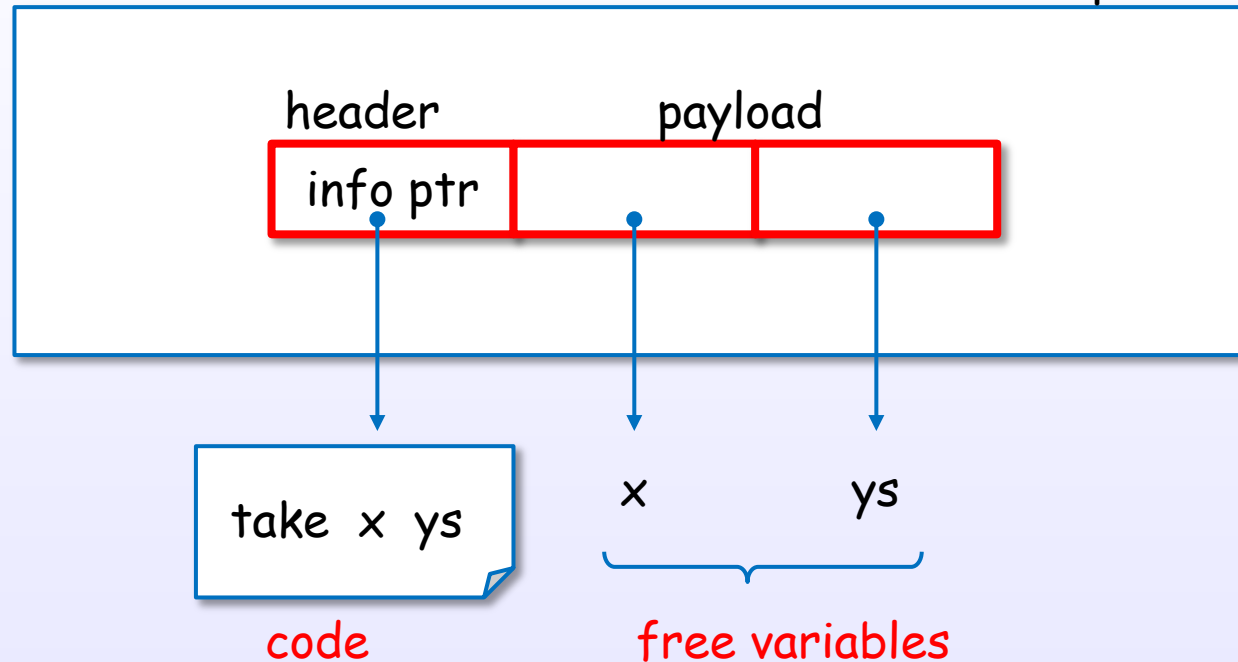
An unevaluated expression

take x ys



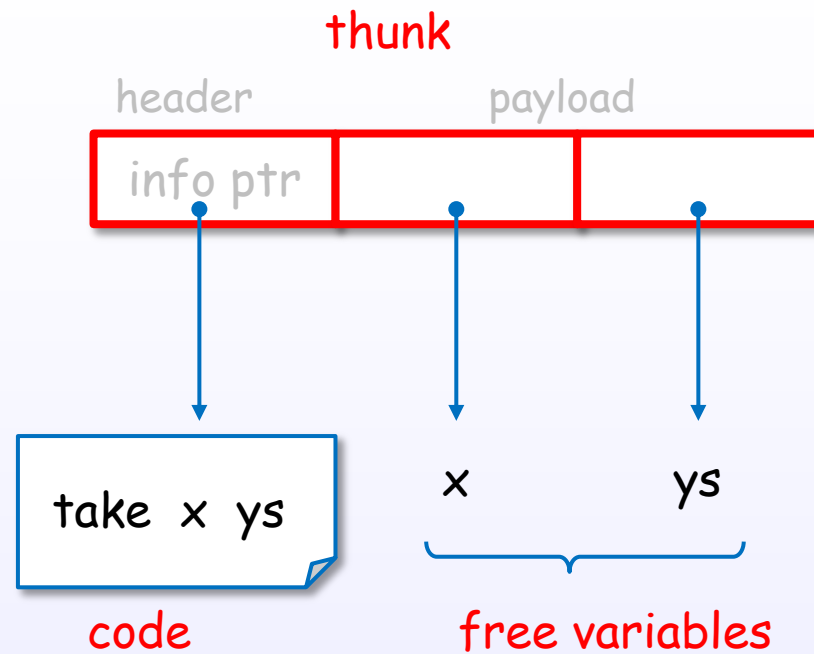
thunk

GHC's internal representation



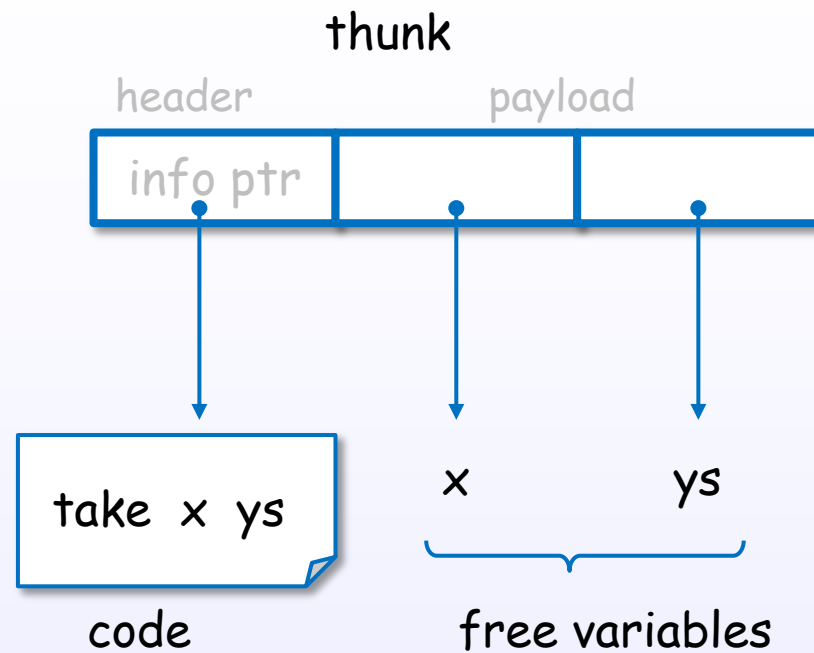
A thunk is represented with header(code) + payload(free variables).

Thunkは、codeとfree variablesをパッケージ化したもの



A thunk is a package of code + free variables.

Thunkは、forcing要求により評価される



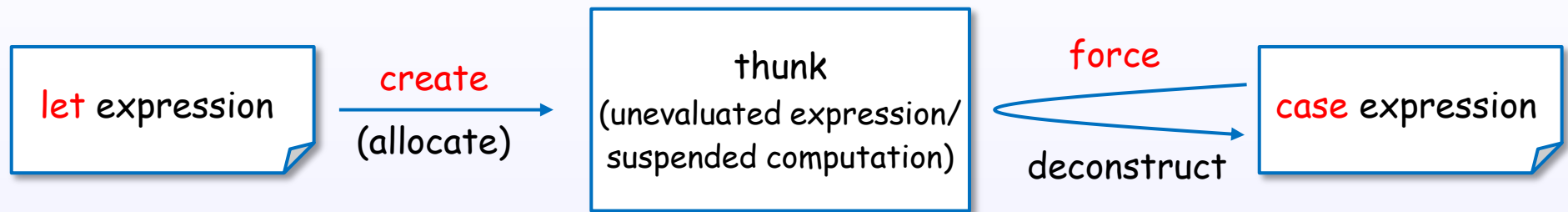
↓ forcing

[1, 2, 3]

A evaluated expression

let, case expression

let/case expressions and thunk



A let expression may create a thunk.

A case expression forces and deconstructs the thunk.

A let expression creates a thunk

let expression

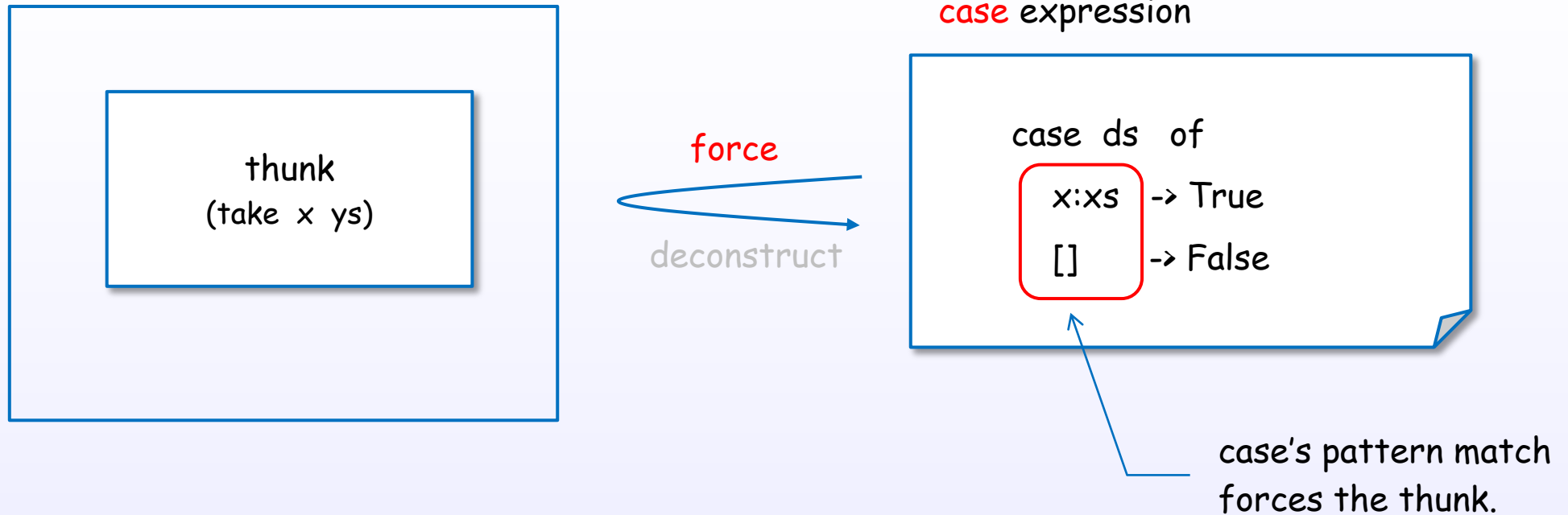
let ds = take x ys

create
(allocate)

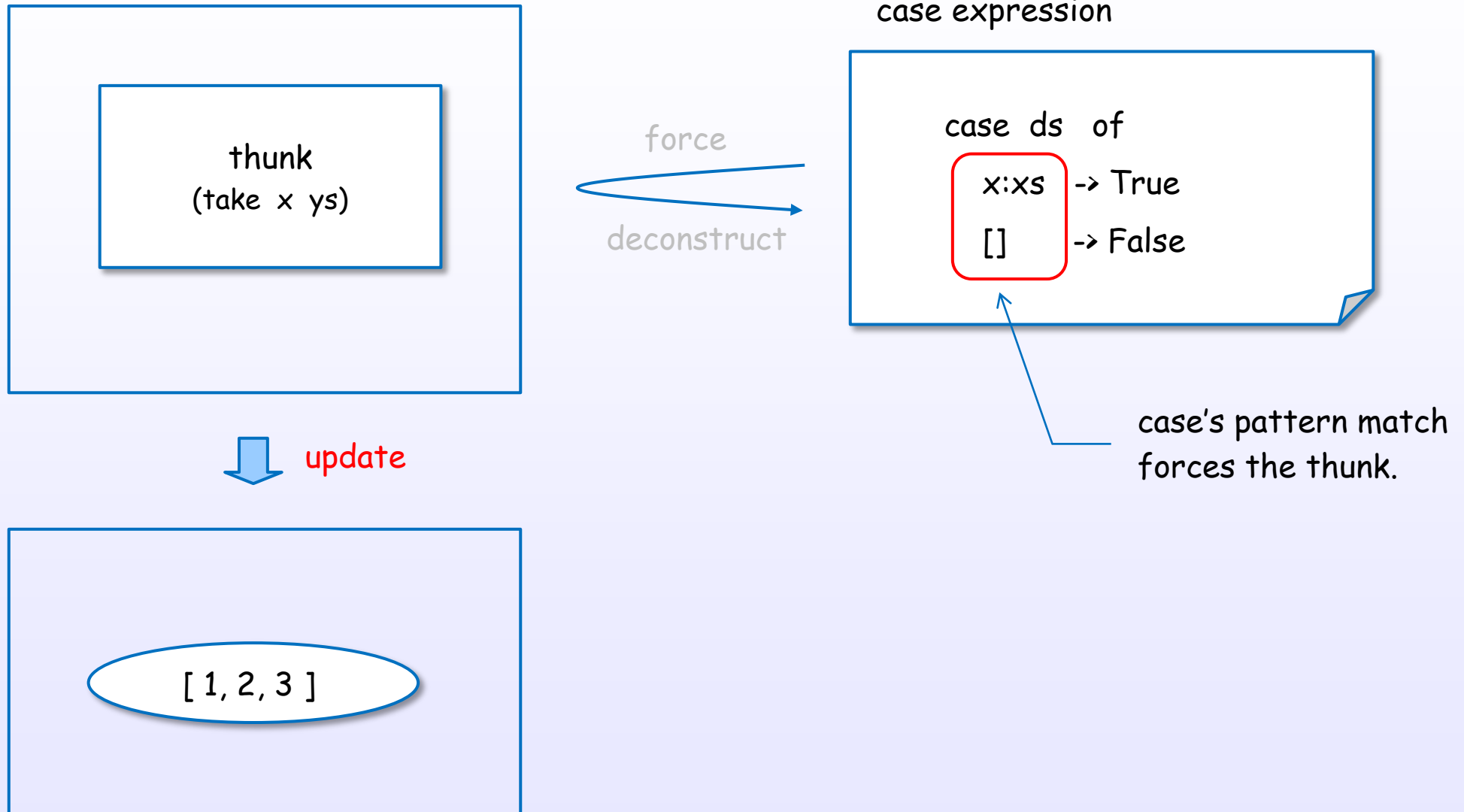
thunk
(take x ys)

heap memory

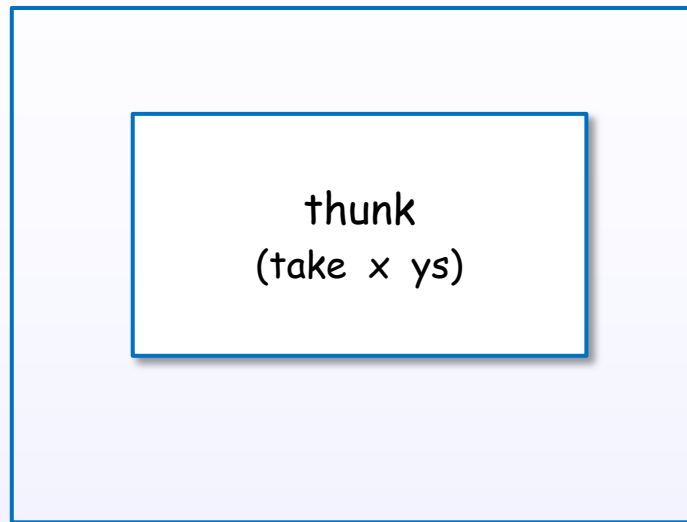
A case expression forces a thunk



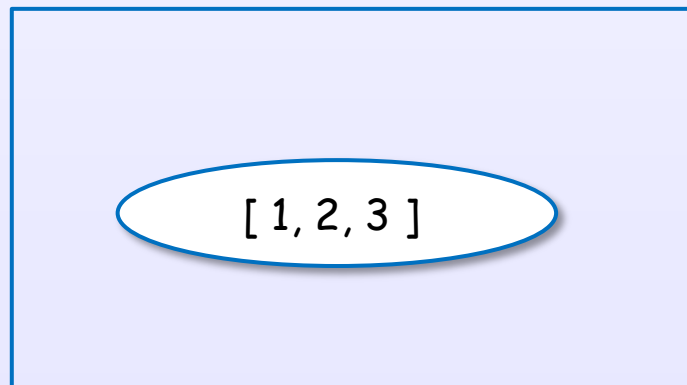
A case expression forces a thunk



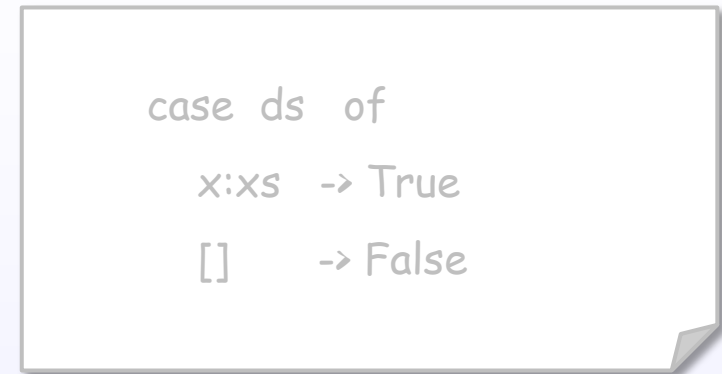
A case expression forces a thunk



↓ update

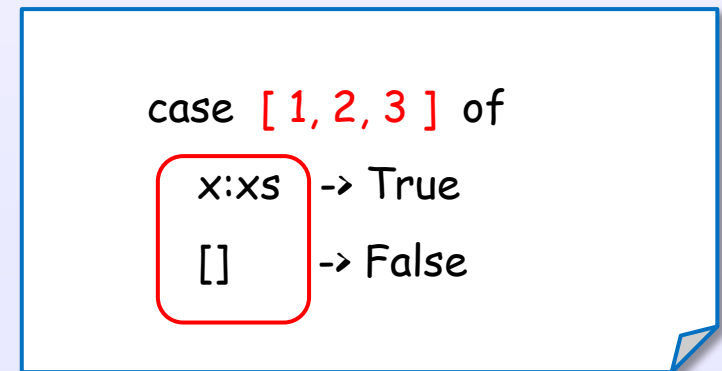


case expression



force
→
deconstruct

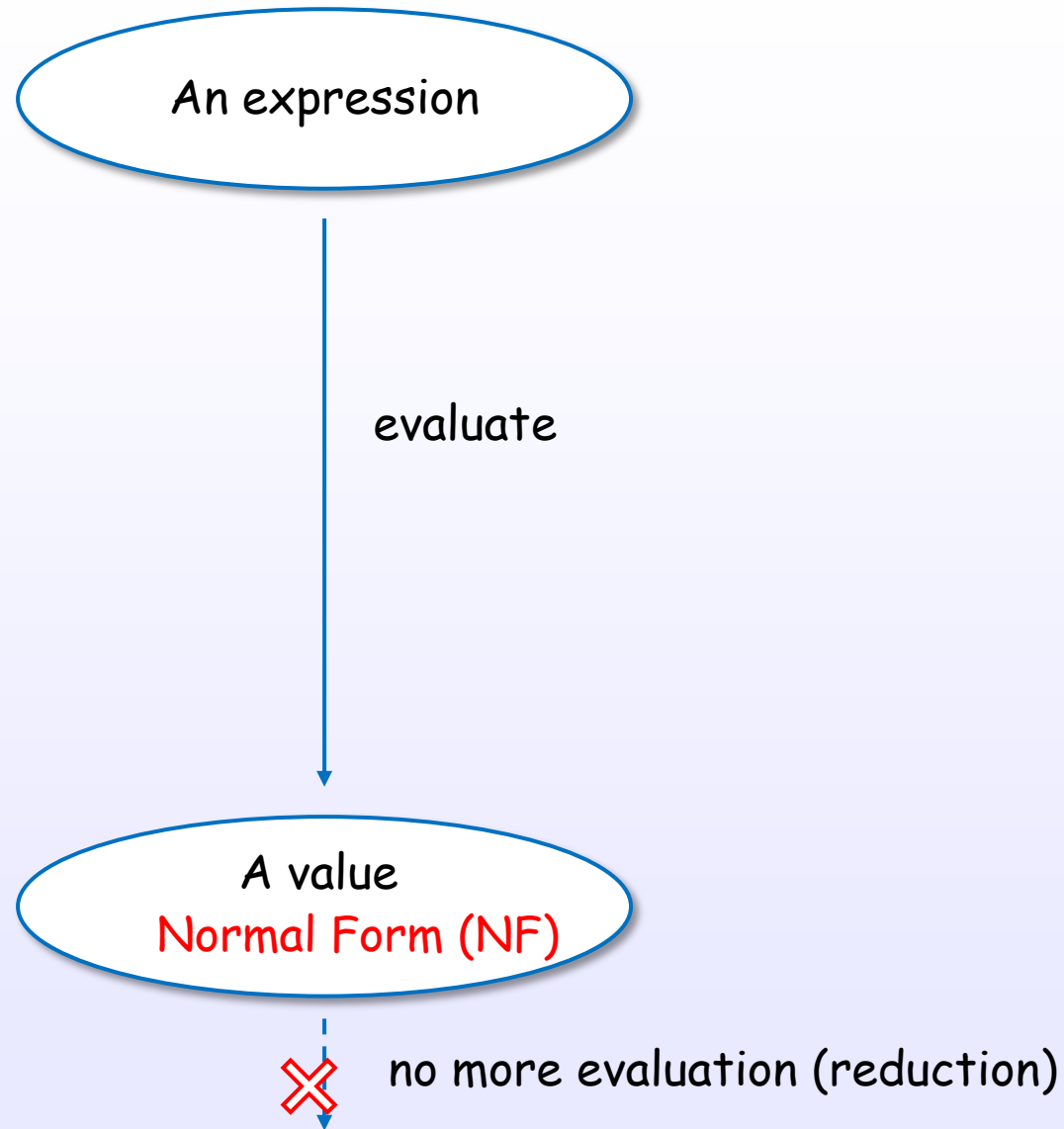
case expression



force
→
deconstruct

WHNF

Normal form は、これ以上評価できない値



[Terei]

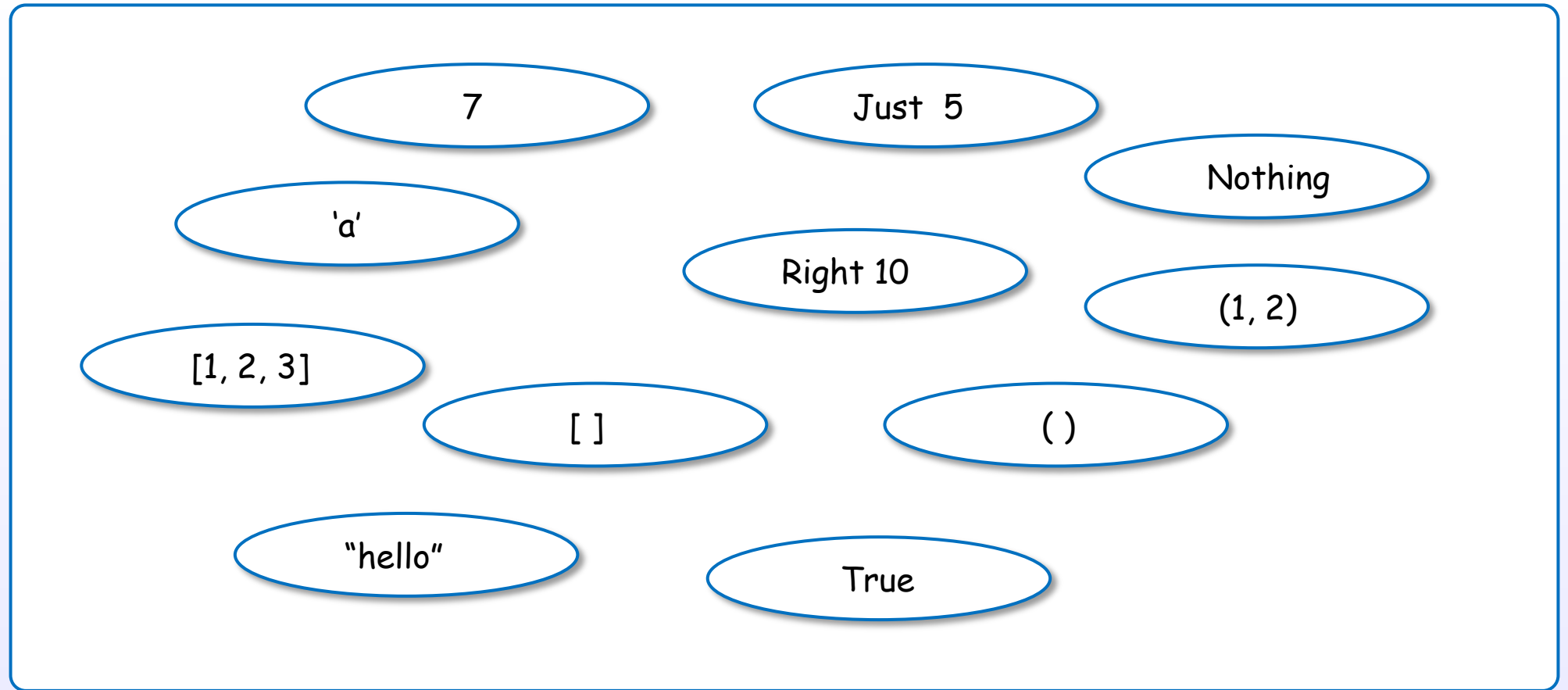
[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

References : [1]

Examples of normal form (NF)

Normal Forma (NF)

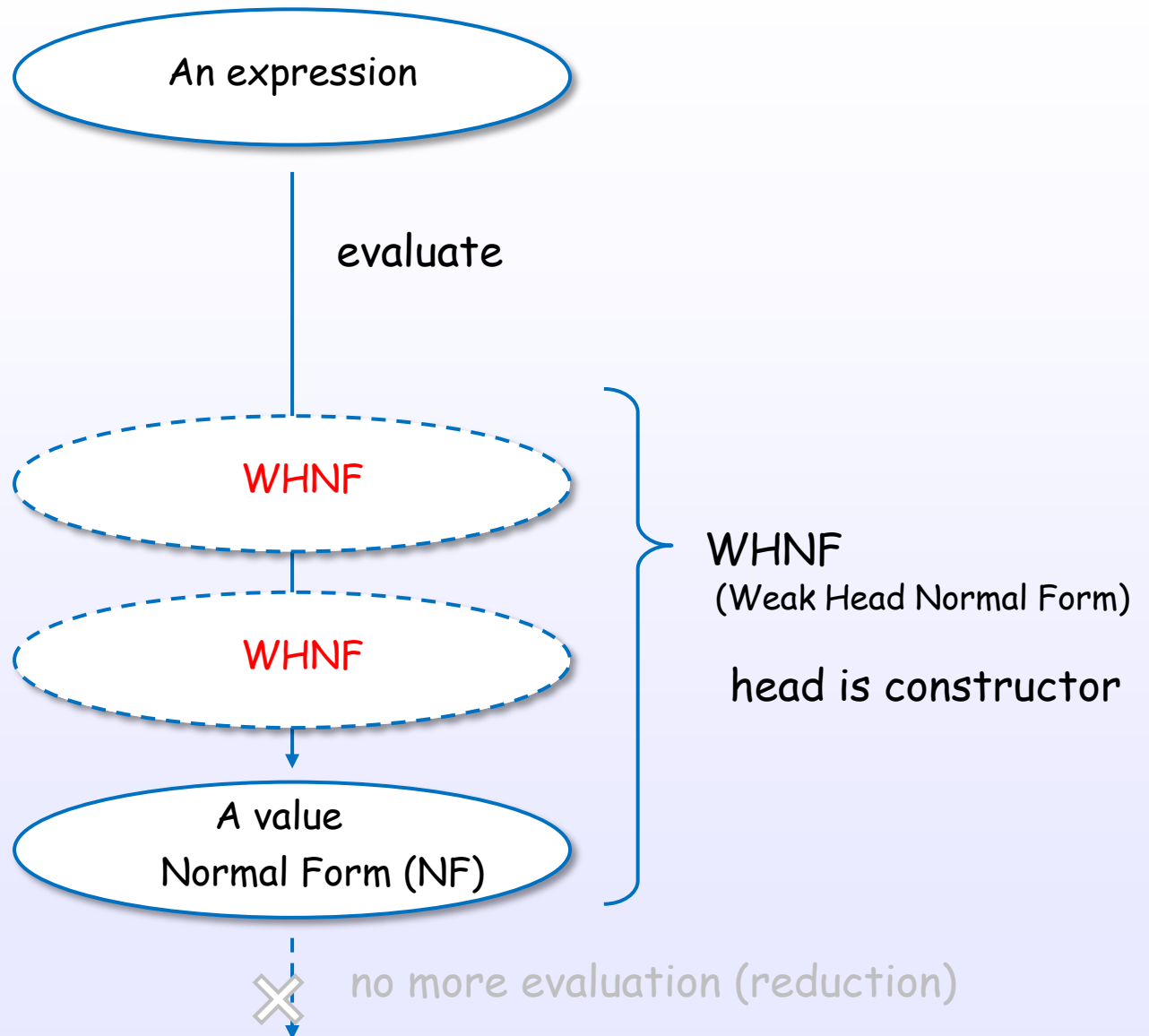


[Terei]

[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

Weak Head Normal form は、少なくとも先頭が評価された式



[Terei]

[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

References : [1]

Examples of weak head normal form (WHNF)

Weak Head Normal Form (WHNF)

Just (head [1..])

Right (fun arg)

1 : [2..]

Cons 1 [2..]

5 : map f xs

Cons 5 (map f xs)

(f 5, g 7)

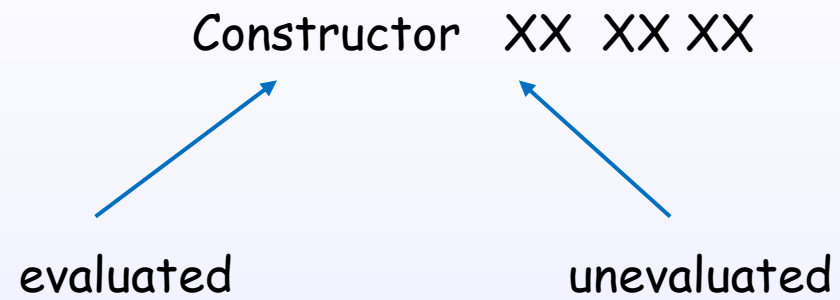
Pair (f 5) (g 7)

[Terei]

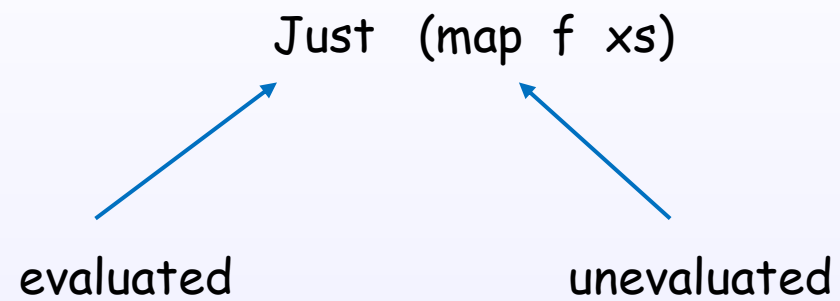
[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

WHNF



WHNF



前頁の、heap objectイメージ

[4]

normal form:

an expression without an redexes

head normal form:

an expression where the top level (head) is neither a redex NOR
a lambda abstraction with a reducible body

weak head normal form:

an expression where the top level (head) isn't a redex

[Terei]

[4]

evaluation strategies:

call-by-value: arguments evaluated before function entered (copied)

call-by-name: arguments passed unevaluated

call-by-need: arguments passed unevaluated but an expression is only evaluated once (sharing)

no-strict evaluation Vs. lazy evaluation:

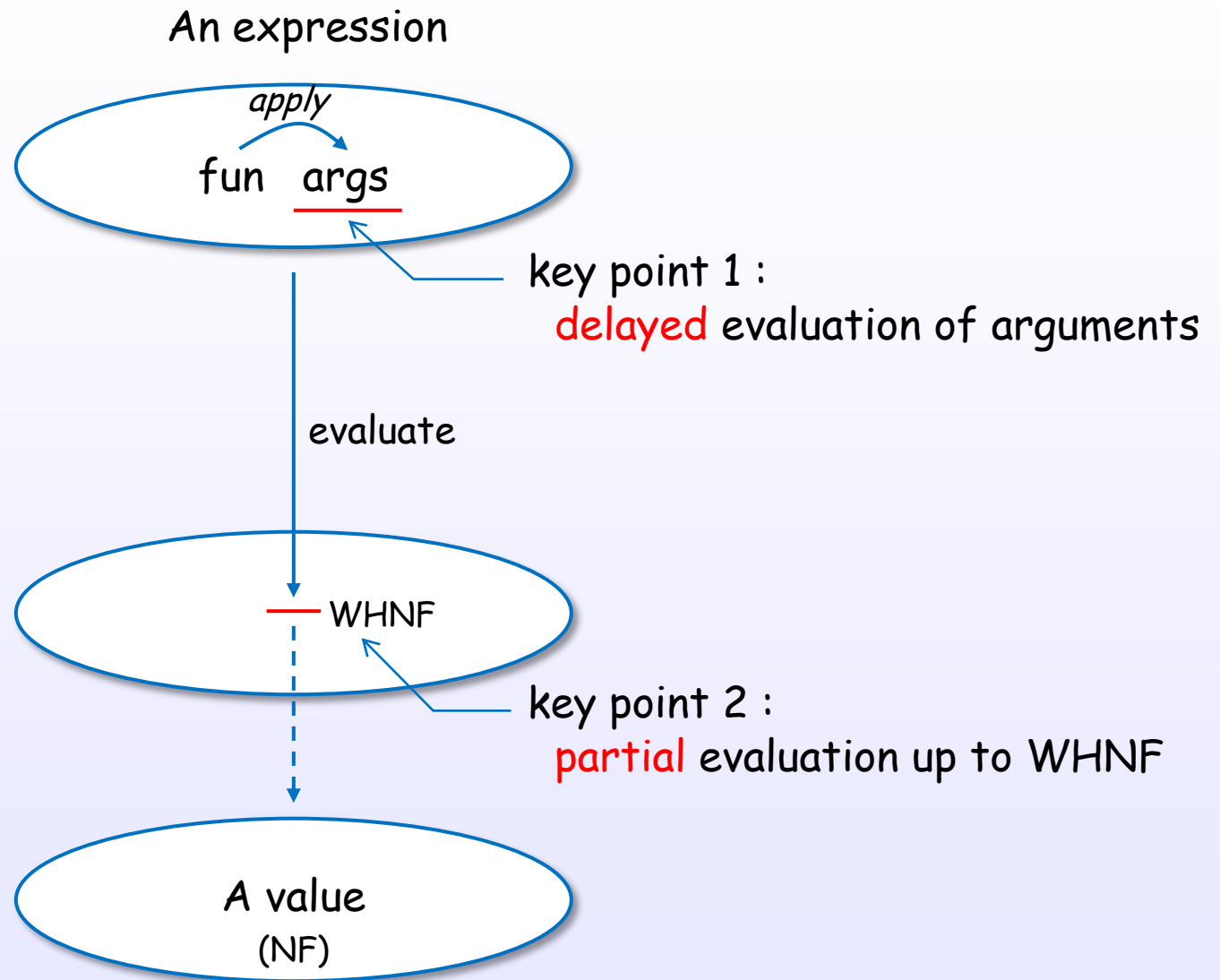
non-strict: Includes both call-by-name and call-by-need, general term for evaluation strategies that don't evaluate arguments before entering a function

lazy evaluation: Specific type of non-strict evaluation. Uses call-by-need (for sharing).

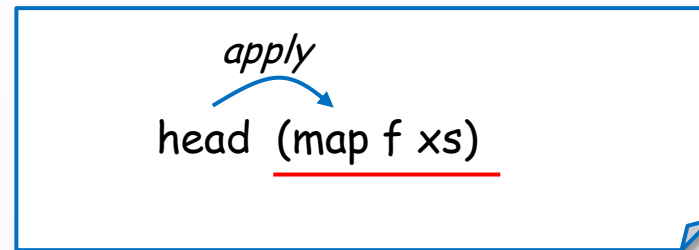
[Terei]

Evaluation in Haskell (GHC)

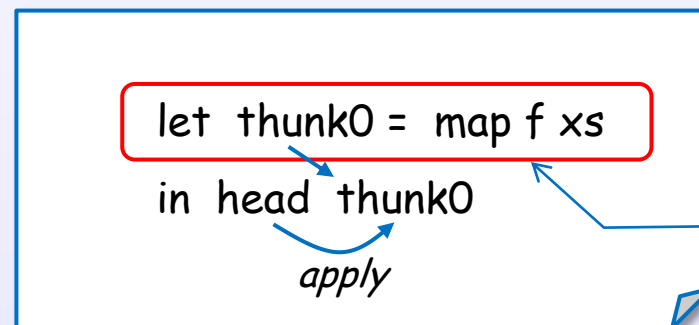
Key concept of Haskell's lazy evaluation



key point 1 : delayed evaluation of arguments



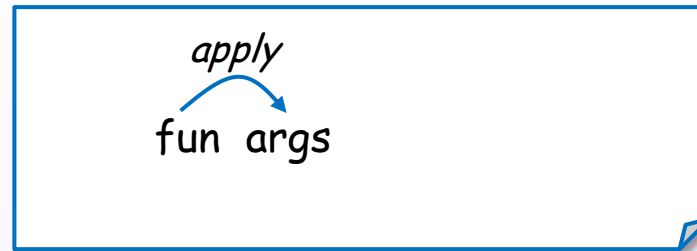
↓ internal transformation by GHC



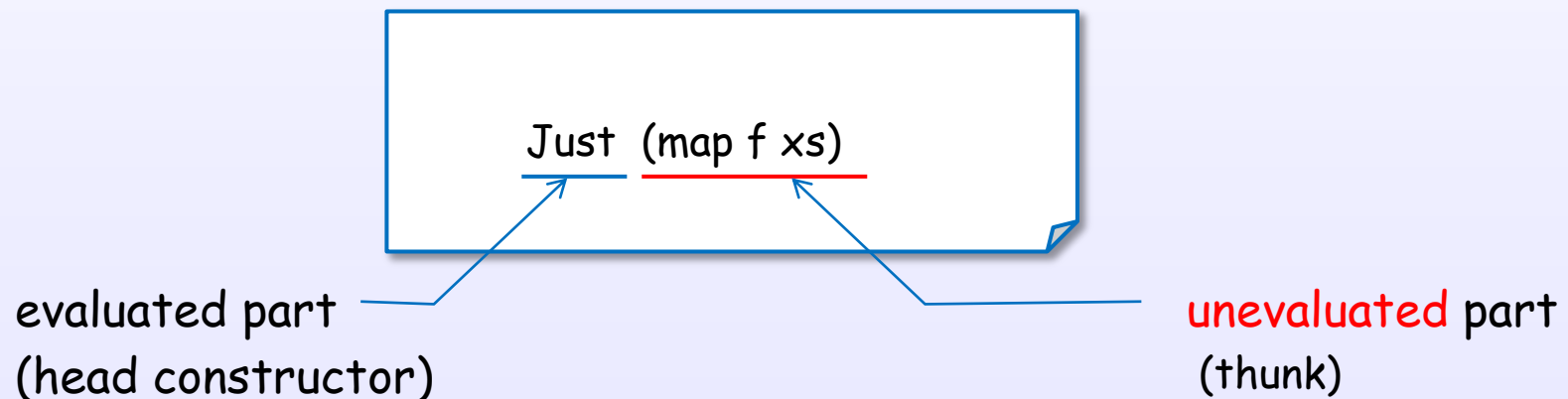
create a thunk
in heap memory

GHC implements lazy evaluation using the thunk.
Evaluation of arguments is delayed with the thunk.

key point 2 : partial evaluation up to WHNF



↓ evaluation up to WHNF



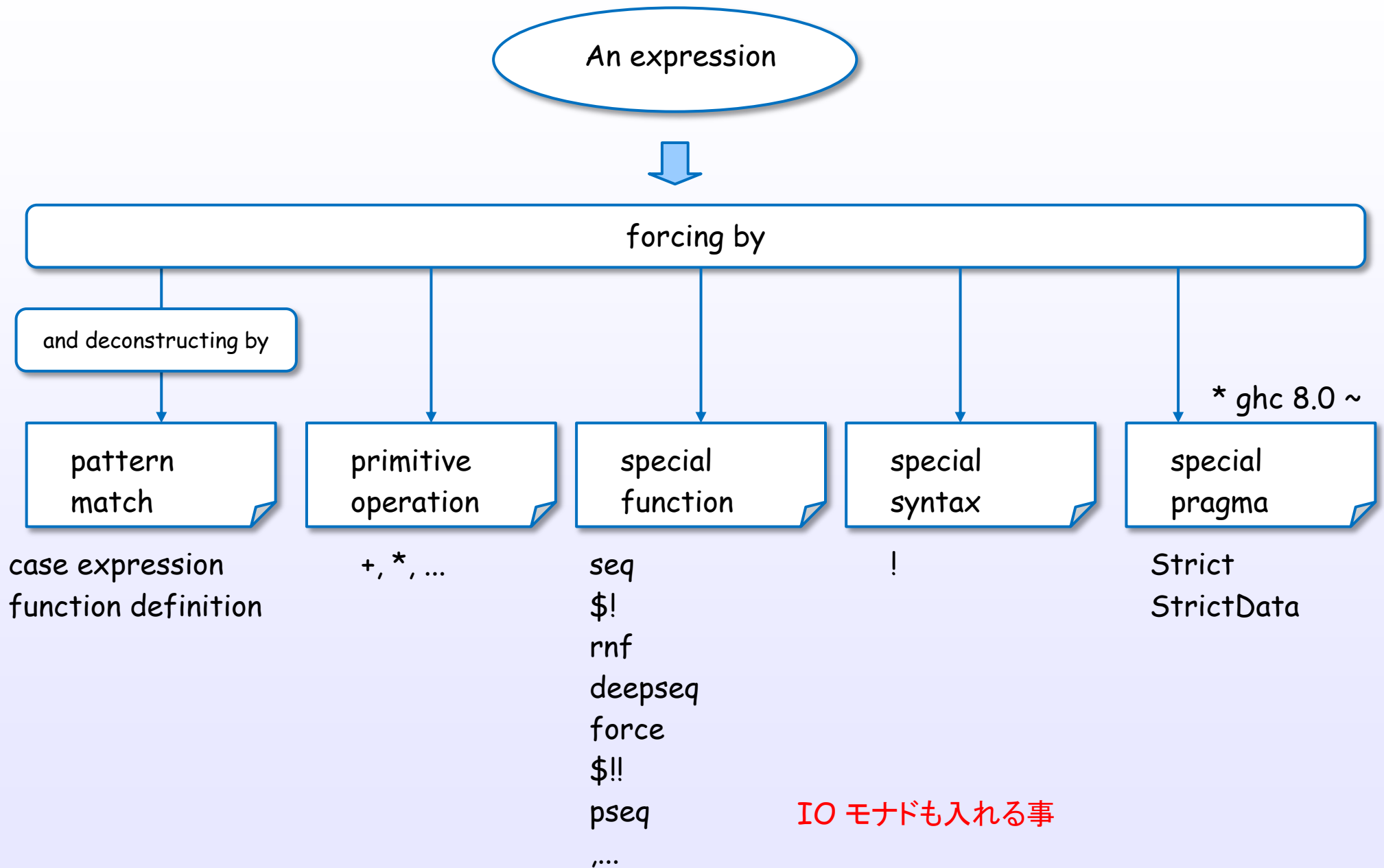
GHC can partially evaluate a expression.
Constructor can hold an unevaluated expression (a thunk).

Pattern match

[CIS194]

Control the evaluation in Haskell

How to drive evaluation



Examples of evaluation steps

Example of foldl (non-strict)

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



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



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



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



...

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]



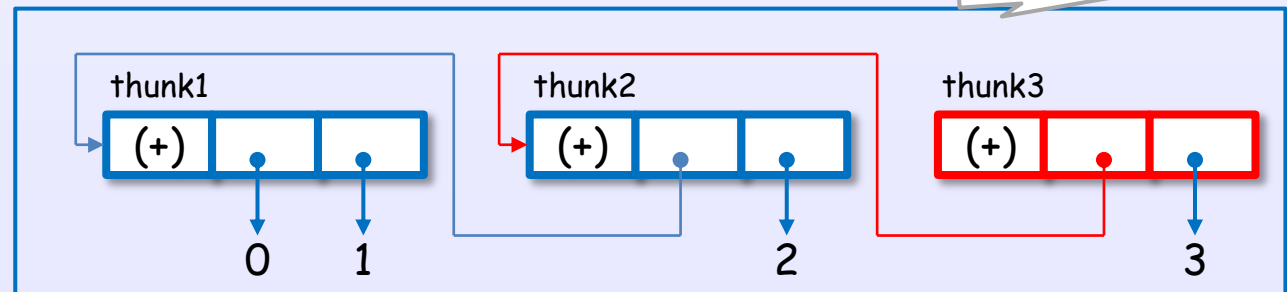
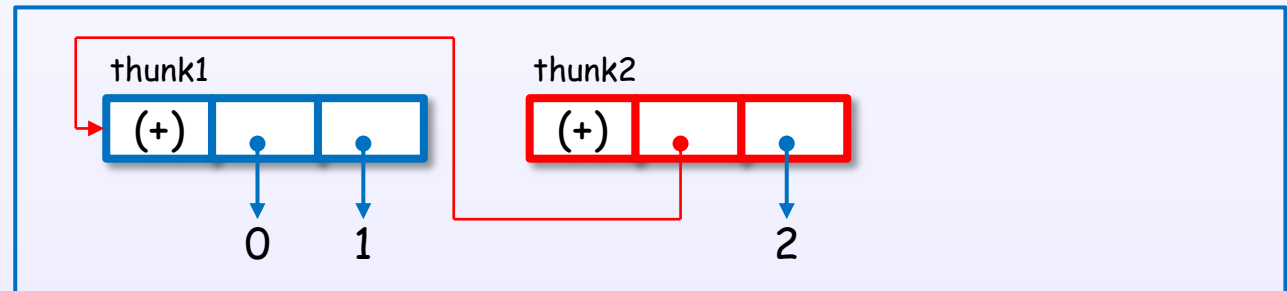
foldl (+) (((0 + 1) + 2) + 3) [4 .. 100]

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



...

heap memory



Example of foldl' (strict)

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



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



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



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



...

Example of foldl' (strict)

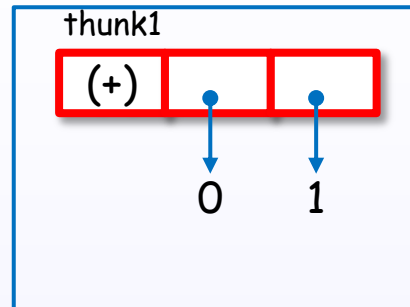
foldl' (+) 0 [1 .. 100]



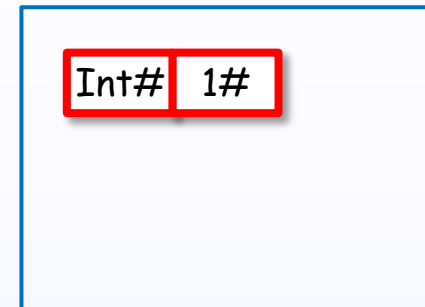
foldl' (+) (0 + 1) [2 .. 100]

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

heap memory

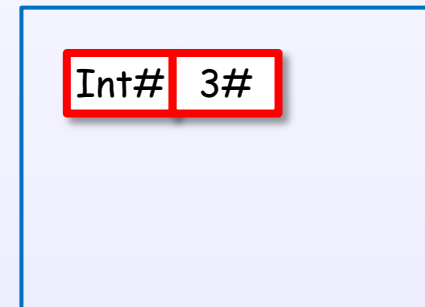
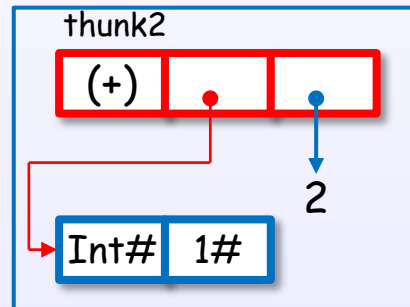


update
by pseq



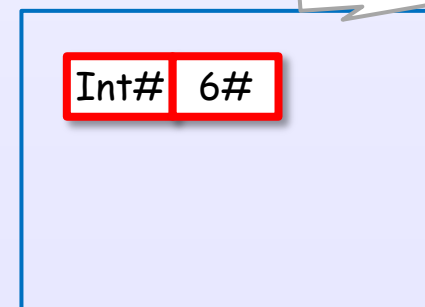
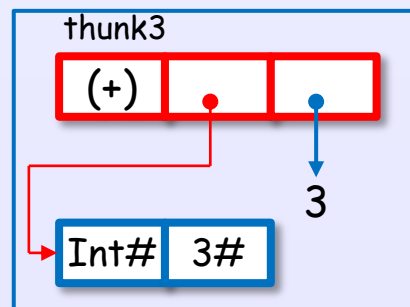
foldl' (+) (1 + 2) [3 .. 100]

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



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

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



fixed heap size



...

References : [1]

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]



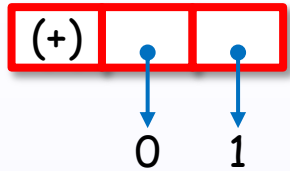
foldl' (+) (3 + 3) [4 .. 100]



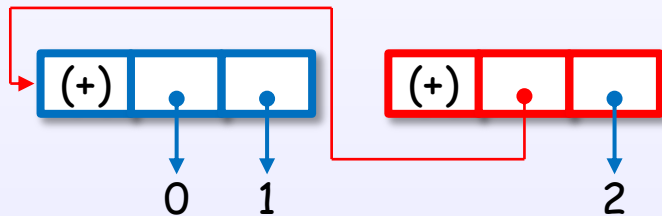
Example of foldl (non-strict) and foldl' (strict)

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

heap memory

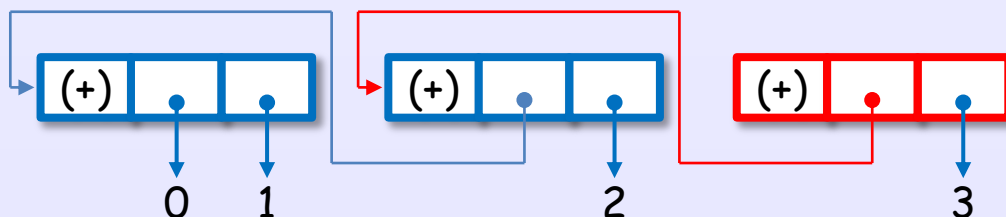


foldl (+) ((0 + 1) + 2) [3 .. 100]

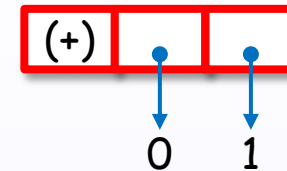


foldl (+) (((0 + 1) + 2) + 3) [4 .. 100]

increasing heap ...



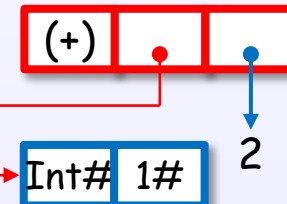
foldl' (+) (0 + 1) [2 .. 100]



Int# 1#



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

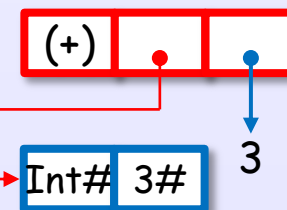


Int# 3#



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

fixed heap size



Int# 6#



References : [1]

Implementation in GHC

Tree, Graph

a expression

AST

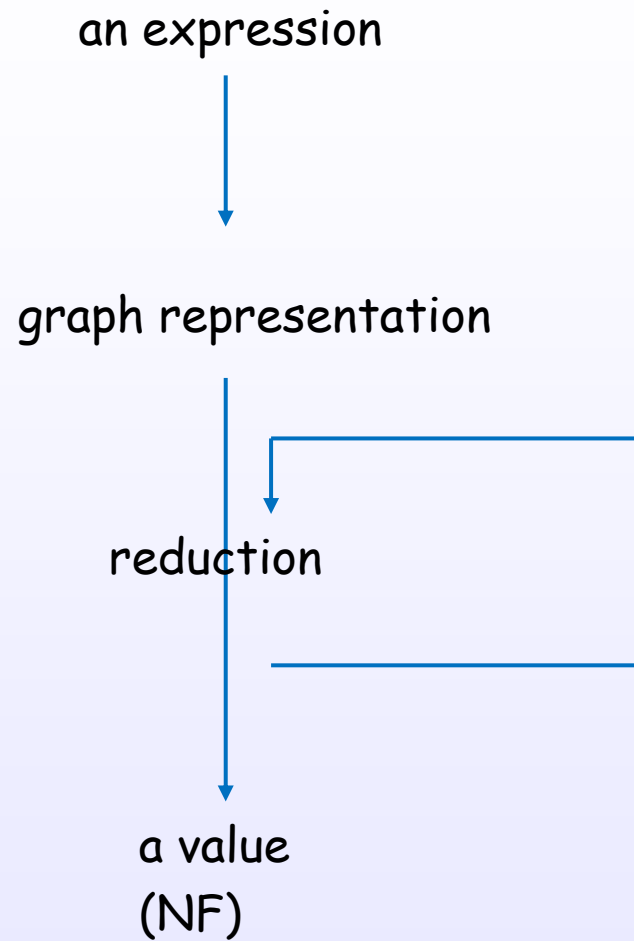
Tree

Graph

Shared Term

Lazy

evaluation, reduction



Expressions examples

STG heap objects

language

Just 5

implementation

heap object

Layer

Non-strictness

$$f \perp = \perp$$

Lazy evaluation

Graph reduction

STG machine

Layer

Haskell semantics

take 5 [1..10]

internal representation

graph

STG semantics

heap object

STG machine

Semantics

Bottom

domain

co-domain

defined

undefined

$$f \perp = \perp$$

[Bird, Chapter 2]

Strictness, Bottom

[Bird, Chapter 2]

References : [1]

References

References

- [1] Haskell 2010 Language Report
<https://www.haskell.org/definition/haskell2010.pdf>
- [2] The Glorious Glasgow Haskell Compilation System (GHC user's guide)
https://downloads.haskell.org/~ghc/latest/docs/users_guide.pdf
- [3] Thinking Functionally with Haskell (IFPH 3rd edition)
<http://www.cs.ox.ac.uk/publications/books/functional/>
- [4] Types and Programming Languages
<https://mitpress.mit.edu/books/types-and-programming-languages>
- [5] A Haskell Compiler
<http://www.scs.stanford.edu/11au-cs240h/notes/ghc-slides.html>
[http://www.scs.stanford.edu/11au-cs240h/notes/ghc-slides.html#\(11\)](http://www.scs.stanford.edu/11au-cs240h/notes/ghc-slides.html#(11))
- [6] Being Lazy with Class
<http://www.seas.upenn.edu/~cis194/lectures/06-laziness.html>
- [7] The Incomplete Guide to Lazy Evaluation (in Haskell)
<https://hackhands.com/guide-lazy-evaluation-haskell/>
- [8] Programming in Haskell
<https://www.cs.nott.ac.uk/~gmh/book.html>
- [9] Parallel and Concurrent Programming in Haskell
<http://chimera.labs.oreilly.com/books/1230000000929/ch02.html>
- [10] Real World Haskell
<http://book.realworldhaskell.org/read/profiling-and-optimization.html>

References

- [11] Laziness
<http://dev.stephendiehl.com/hask/#laziness>
- [12] Evaluation on the Haskell Heap
<http://blog.ezyang.com/2011/04/evaluation-on-the-haskell-heap/>
- [13] How to force a list
<https://ro-che.info/articles/2015-05-28-force-list>
- [14] Haskell/Lazy evaluation
https://wiki.haskell.org/Haskell/Lazy_evaluation
- [15] Lazy evaluation
https://wiki.haskell.org/Lazy_evaluation
- [16] Lazy vs. non-strict
https://wiki.haskell.org/Lazy_vs._non-strict
- [17] Haskell/Denotational semantics
https://en.wikibooks.org/wiki/Haskell/Denotational_semantics
- [18] Haskell/Graph reduction
https://en.wikibooks.org/wiki/Haskell/Graph_reduction

References

- [19] Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine Version 2.5
<http://research.microsoft.com/en-us/um/people/simonpj/Papers/spineless-tagless-gmachine.ps.gz>
- [20] Making a Fast Curry Push/Enter vs Eval/Apply for Higher-order Languages
<http://research.microsoft.com/en-us/um/people/simonpj/papers/eval-apply/>
- [21] I know kung fu: learning STG by example
<https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/GeneratedCode>
- [22] GHC Commentary: The Layout of Heap Objects
<https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/HeapObjects>
- [23] GHC Commentary: Strict & StrictData
<https://ghc.haskell.org/trac/ghc/wiki/StrictPragma>
- [24] GHC illustrated
http://takenobu-hs.github.io/downloads/haskell_ghc_illustrated.pdf

Lazy,... ^{!!!}