Lazy evaluation in Haskell

exploring some mental models and implementations

Takenobu T.

Rev. 0.00.0
WIP

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

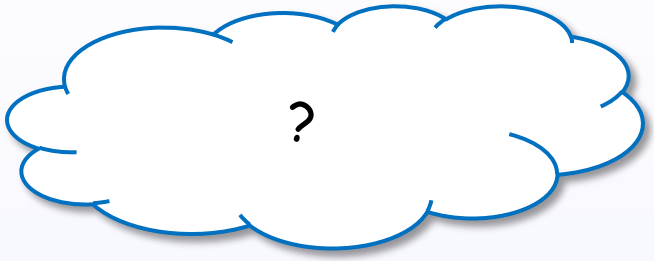
- Semantics
- Graph reduction
- Implementation in GHC

- References

Introduction

What is an expression?

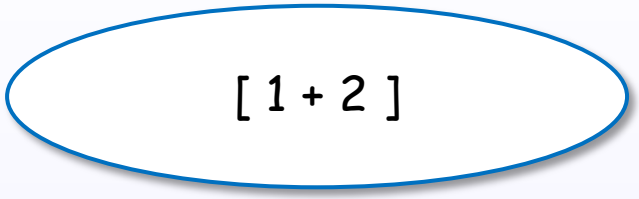
An expression



References : [1]

An expression denotes a value

An expression

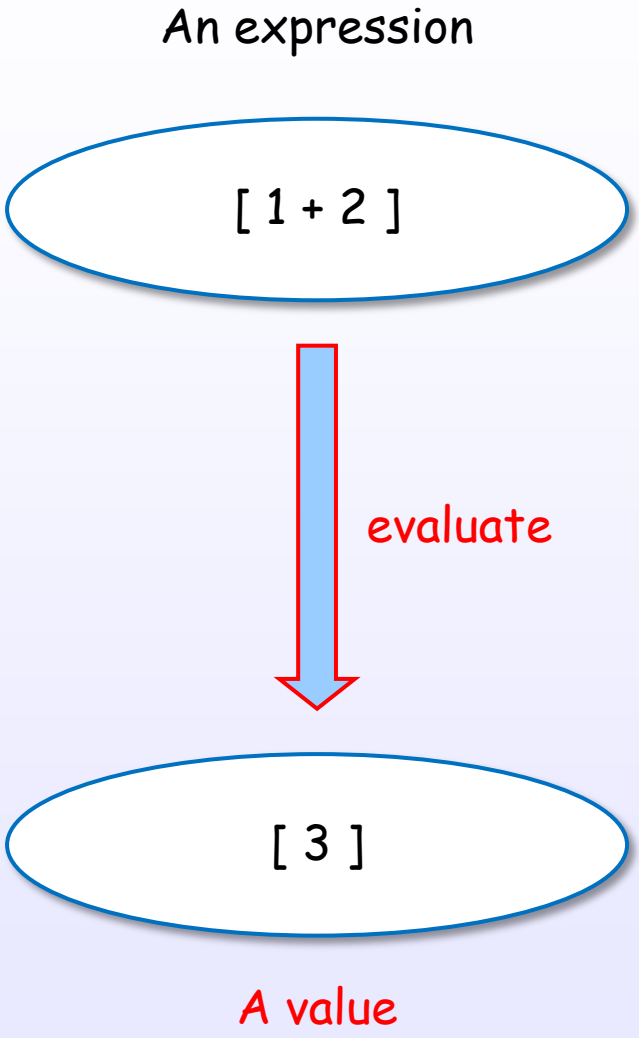


[HR2010]

[Bird, Chapter 2]

References : [1]

An expression evaluates to a value

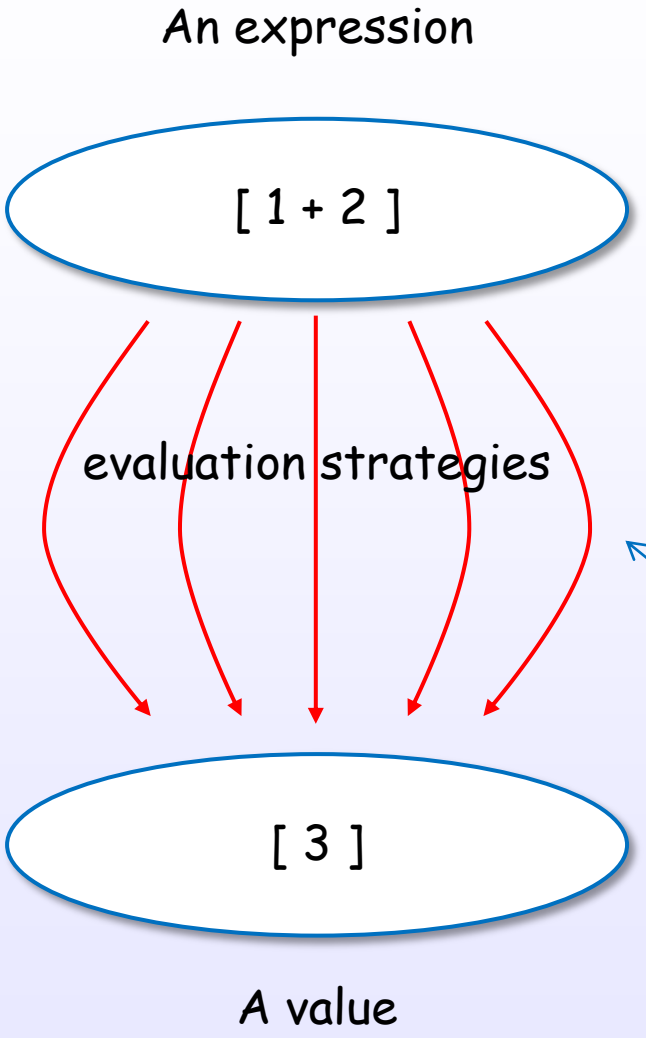


[HR2010]

[Bird, Chapter 2]

References : [1]

There are many evaluation approaches



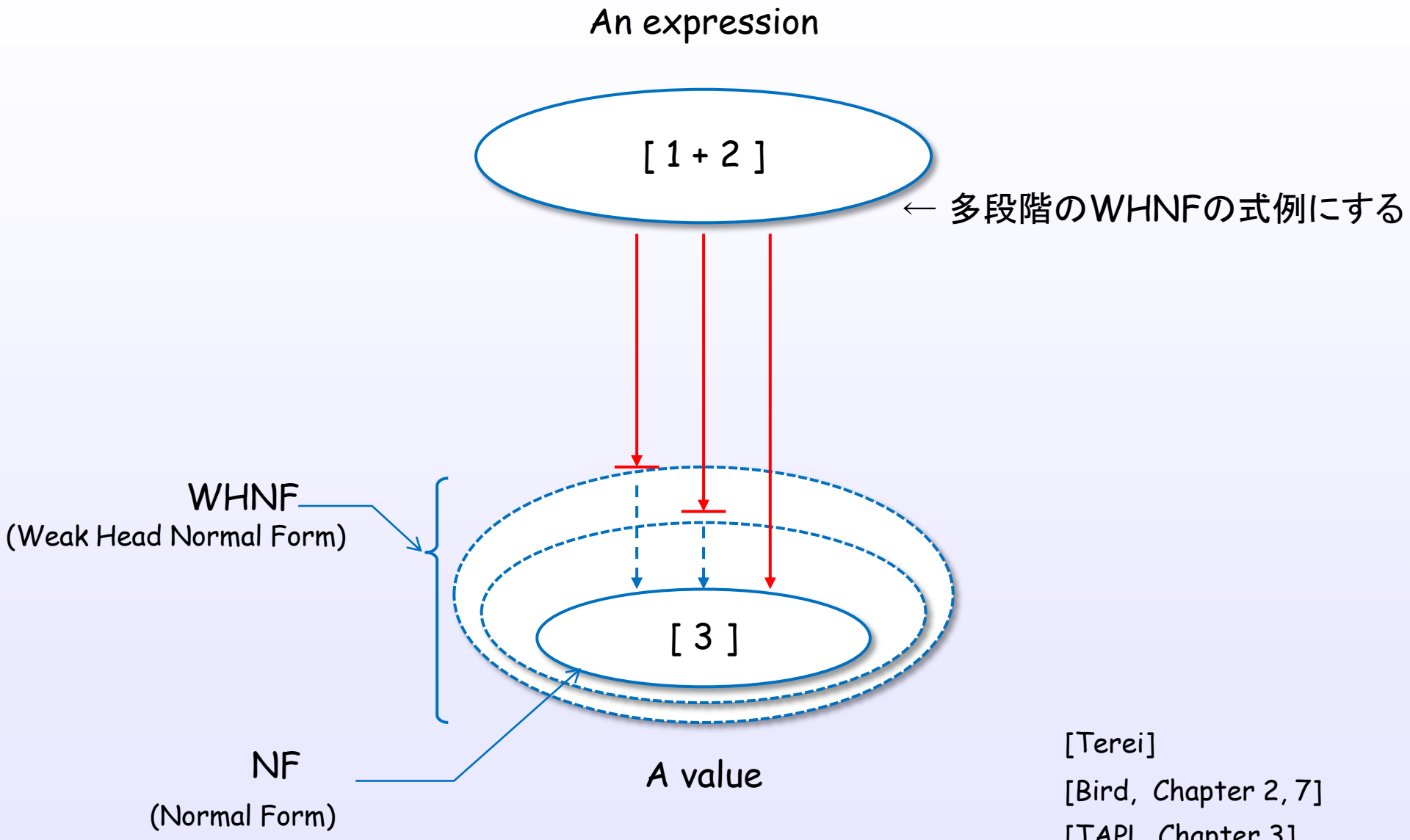
- Strict, Non-strict evaluation
- Eager, Lazy evaluation
- Call-by-value, Call-by-name, Call-by-need, …
- Innermost, Outermost
- Normal order, Application order
- …

[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

References : [1]

There are some evaluation levels



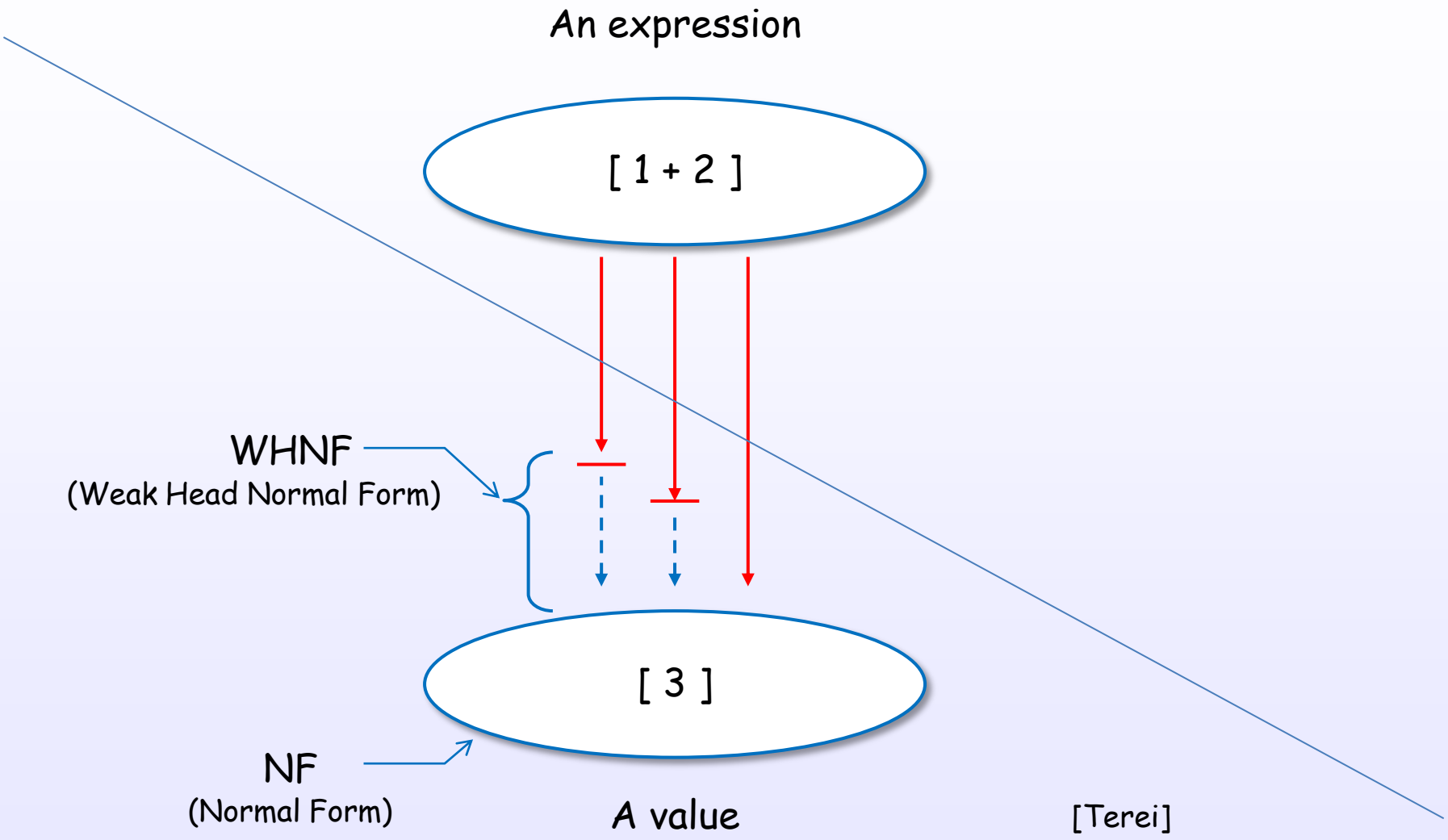
[Terei]

[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

References : [1]

There are some evaluation levels



[Terei]

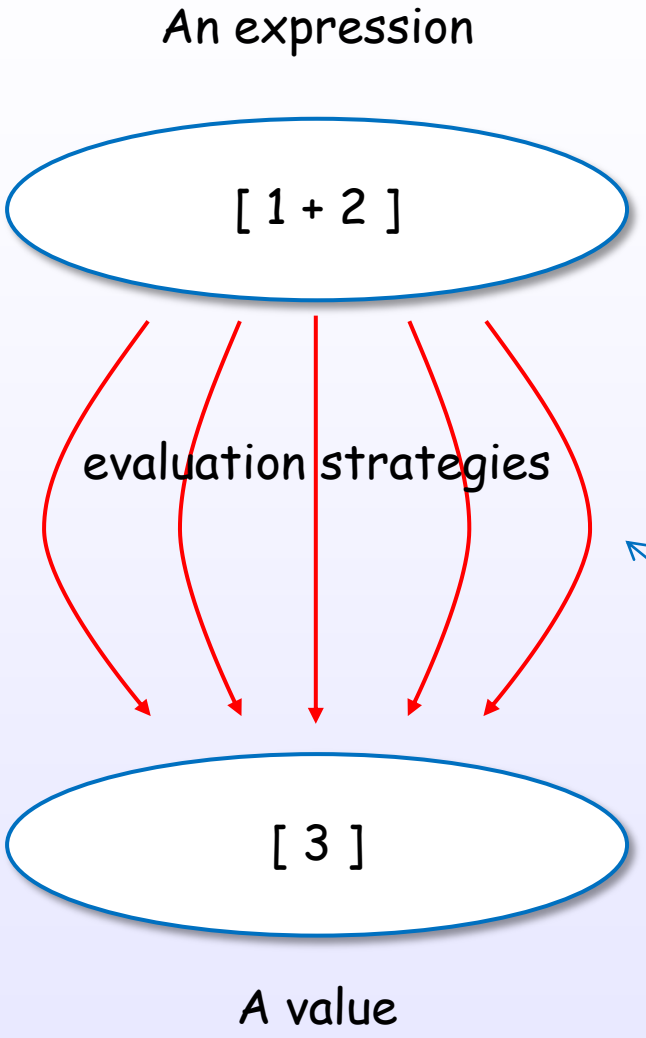
[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

References : [1]

Evaluation strategies

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
- …

[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

References : [1]

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

- Lazy graph reduction
- ...

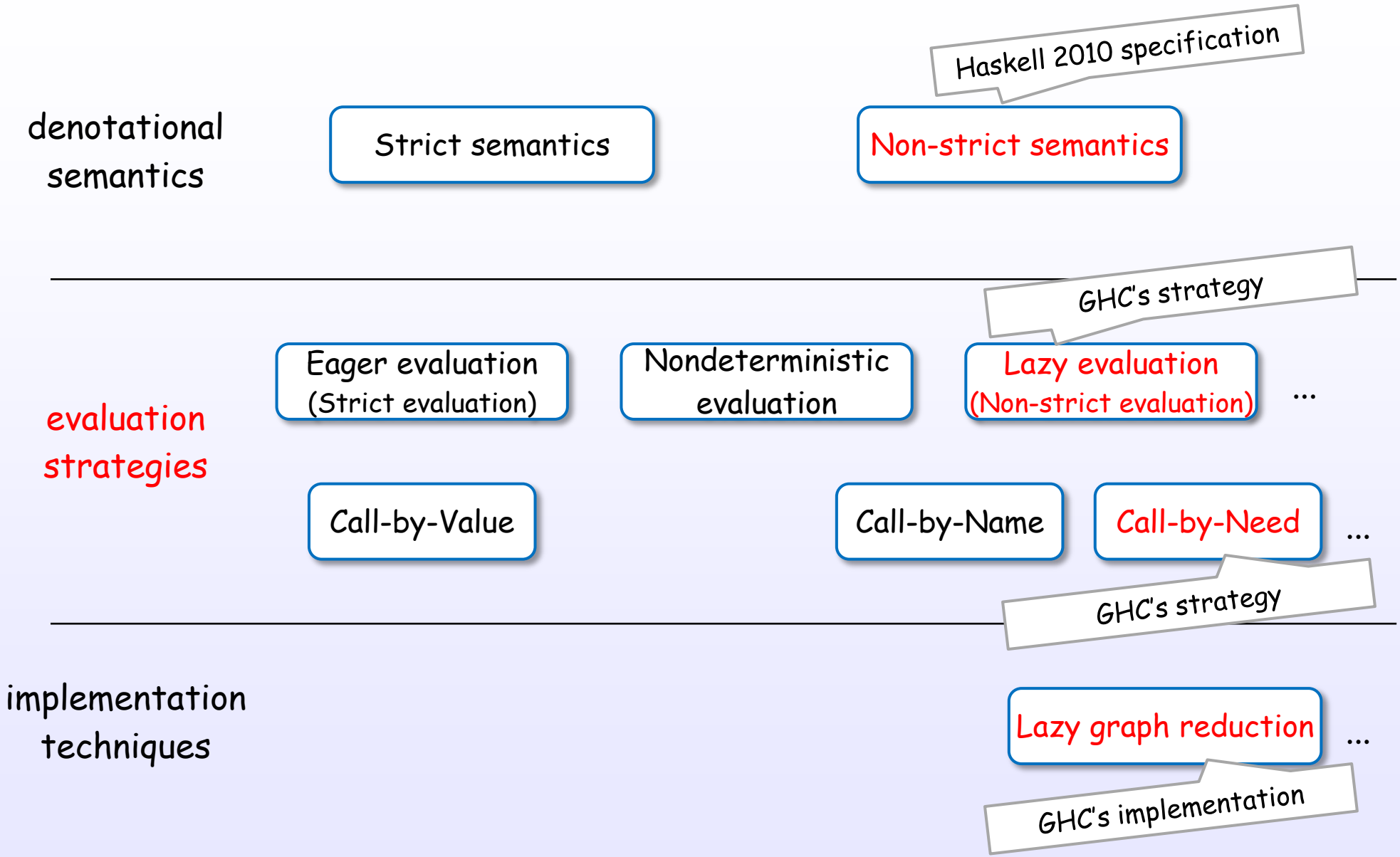
[Bird, Chapter 7]

[Hutton, Chapter 8]

[TAPL, Chapter 3]

References : [1]

Evaluation layers for GHC's Haskell



References : [1]

Evariation strategies and order

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

order

[Bird]
[Hutton]

References : [1]

Simple example of both evaluations

default

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

Eager evaluation
(Strict evaluation)

square (1 + 2)

argument evaluation first

square (3)

3 * 3

9

default

Haskell (GHC), ...

Lazy evaluation
(Non-strict evaluation)

square (1 + 2)

apply first

(1 + 2) * (1 + 2)

(3) * (3)

9

[Bird]
[Hutton]

References : [1]

Simple example of both evaluations

Eager evaluation
(Strict evaluation)

square (1 + 2)

square (3)

argument evaluated

3 * 3

9

Lazy evaluation
(Non-strict evaluation)

square (1 + 2)

(1 + 2) * (1 + 2)

argument evaluation delayed !

(3) * (3)

9

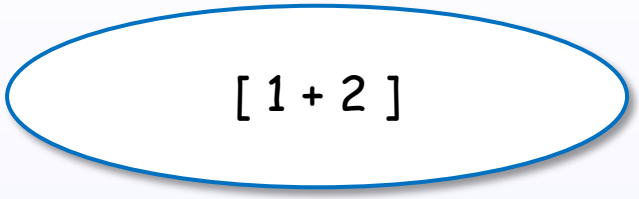
[Bird]
[Hutton]

References : [1]

Expressions in Haskell

An expression denotes a value

An expression

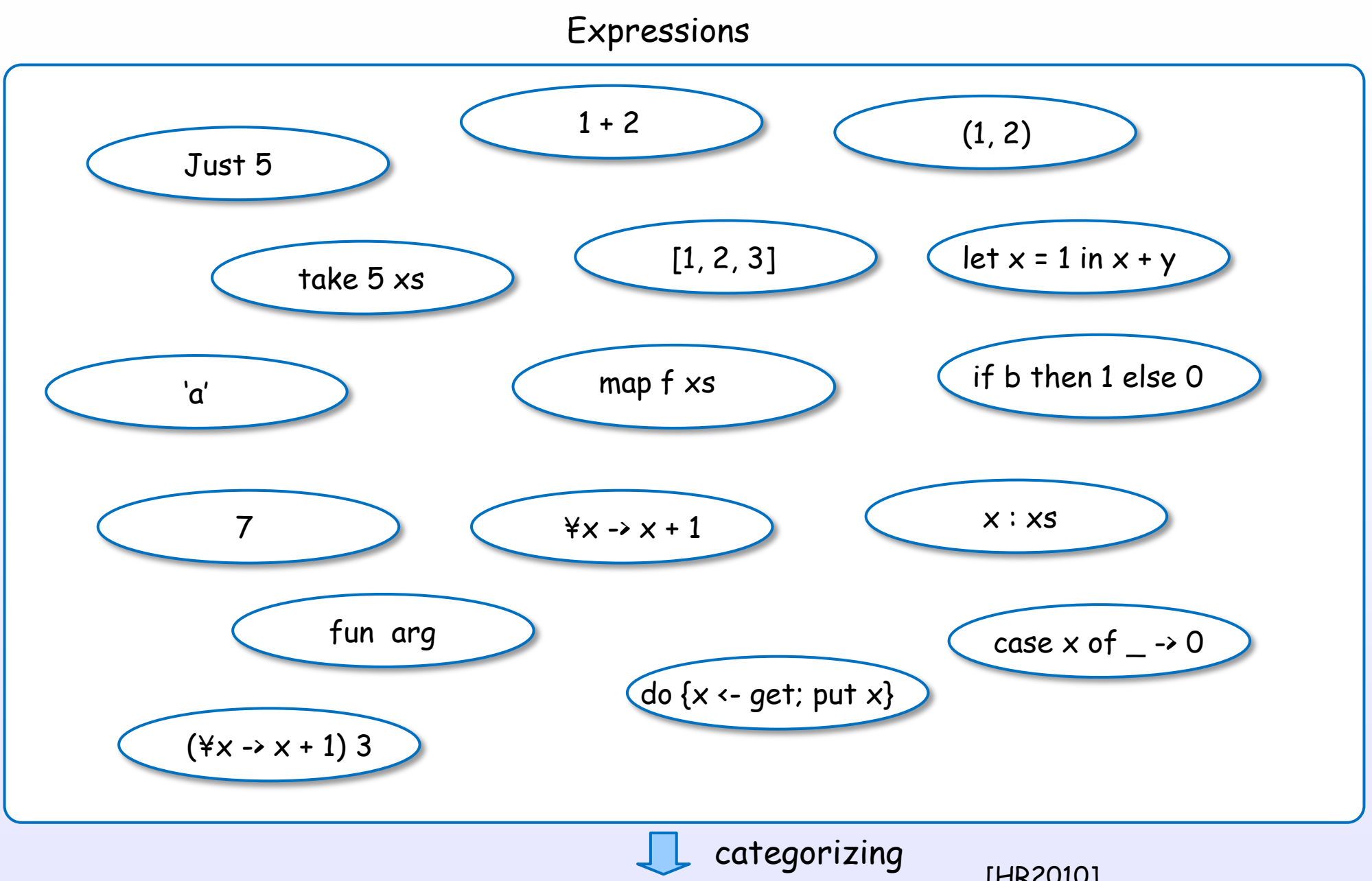


[HR2010]

[Bird, Chapter 2]

References : [1]

There are many expressions in Haskell



[HR2010]

[Bird, Chapter 2]

References : [1]

Expression categories in Haskell

lambda abstraction

¥x -> x + 1

let expression

let x = 1 in x + y

conditional

if b then 1 else 0

case expression

case x of _ -> 0

do expression

do {x <- 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

(¥x -> x + 1) 3

1 + 2

map f xs

fun arg

[HR2010]

[Bird, Chapter 2]

References : [1]

Specification is defined in Haskell 2010 Language Report

Haskell 2010 Language Report, Chapter 3 Expressions [1]

exp	→	*infixexp* `::` [*context* `=>`] *type*	(expression type signature)
	|	*infixexp*	
infixexp	→	*lexp qop infixexp*	(infix operator application)
	|	`-` *infixexp*	(prefix negation)
	|	*lexp*	
lexp	→	`\` $apat_1$ ... $apat_n$ `->` *exp*	(lambda abstraction, $n \geq 1$)
	|	`let` *decls* `in` *exp*	(let expression)
	|	`if` *exp* [`;`] `then` *exp* [`;`] `else` *exp*	(conditional)
	|	`case` *exp* `of` `{` *alts* `}`	(case expression)
	|	`do` `{` *stmts* `}`	(do expression)
	|	*fexp*	
fexp	→	[*fexp*] *aexp*	(function application)
aexp	→	*qvar*	(variable)
	|	*gcon*	(general constructor)
	|	*literal*	
	|	`(` *exp* `)`	(parenthesized expression)
	|	`(` exp_1 `,` ... `,` exp_k `)`	(tuple, $k \geq 2$)
	|	`[` exp_1 `,` ... `,` exp_k `]`	(list, $k \geq 1$)
	|	`[` exp_1 [`,` exp_2] `..` [exp_3] `]`	(arithmetic sequence)
	|	`[` *exp* `|` $qual_1$ `,` ... `,` $qual_n$ `]`	(list comprehension, $n \geq 1$)
	|	`(` *infixexp qop* `)`	(left section)
	|	`(` $qop_{\langle - \rangle}$ *infixexp* `)`	(right section)
	|	*qcon* `{` $fbind_1$ `,` ... `,` $fbind_n$ `}`	(labeled construction, $n \geq 0$)
	|	$aexp_{\langle qcon \rangle}$ `{` $fbind_1$ `,` ... `,` $fbind_n$ `}`	(labeled update, $n \geq 1$)

References : [1]

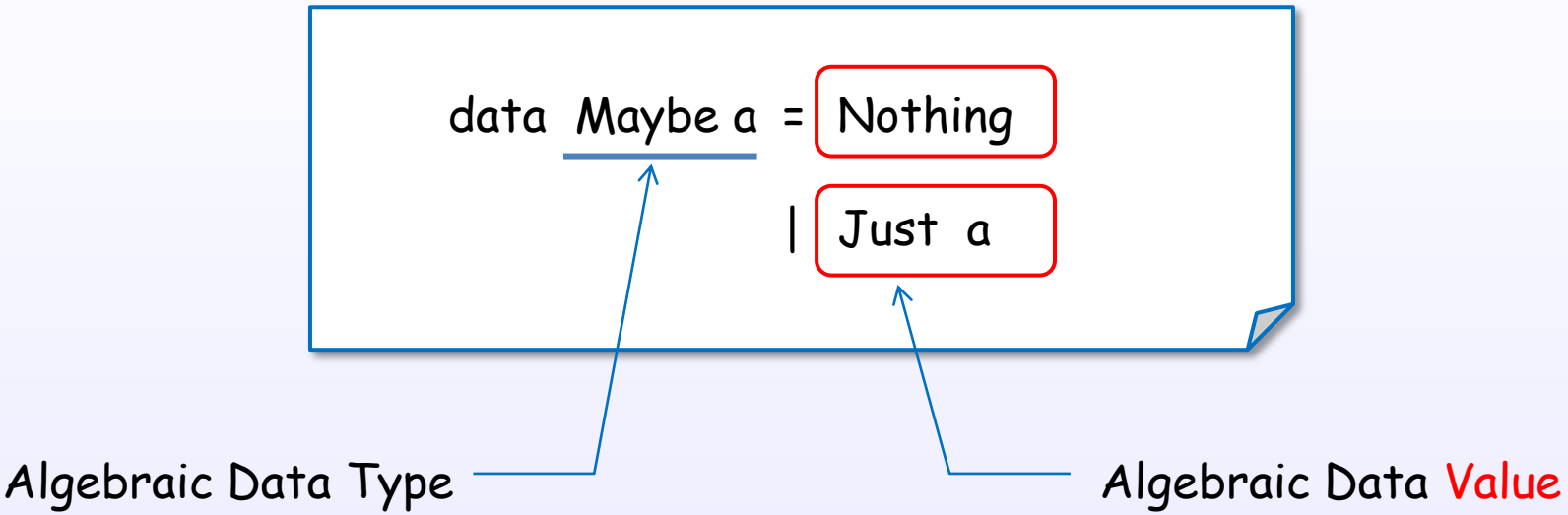
Constructor

Constructor

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

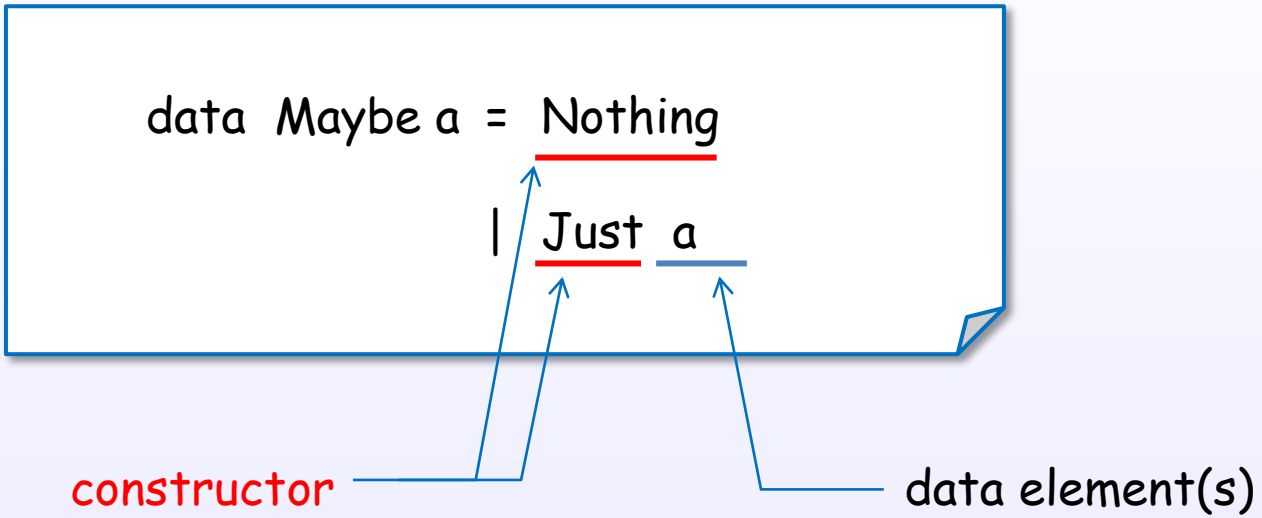
References : [1]

data文で宣言する代数的データ型とその値



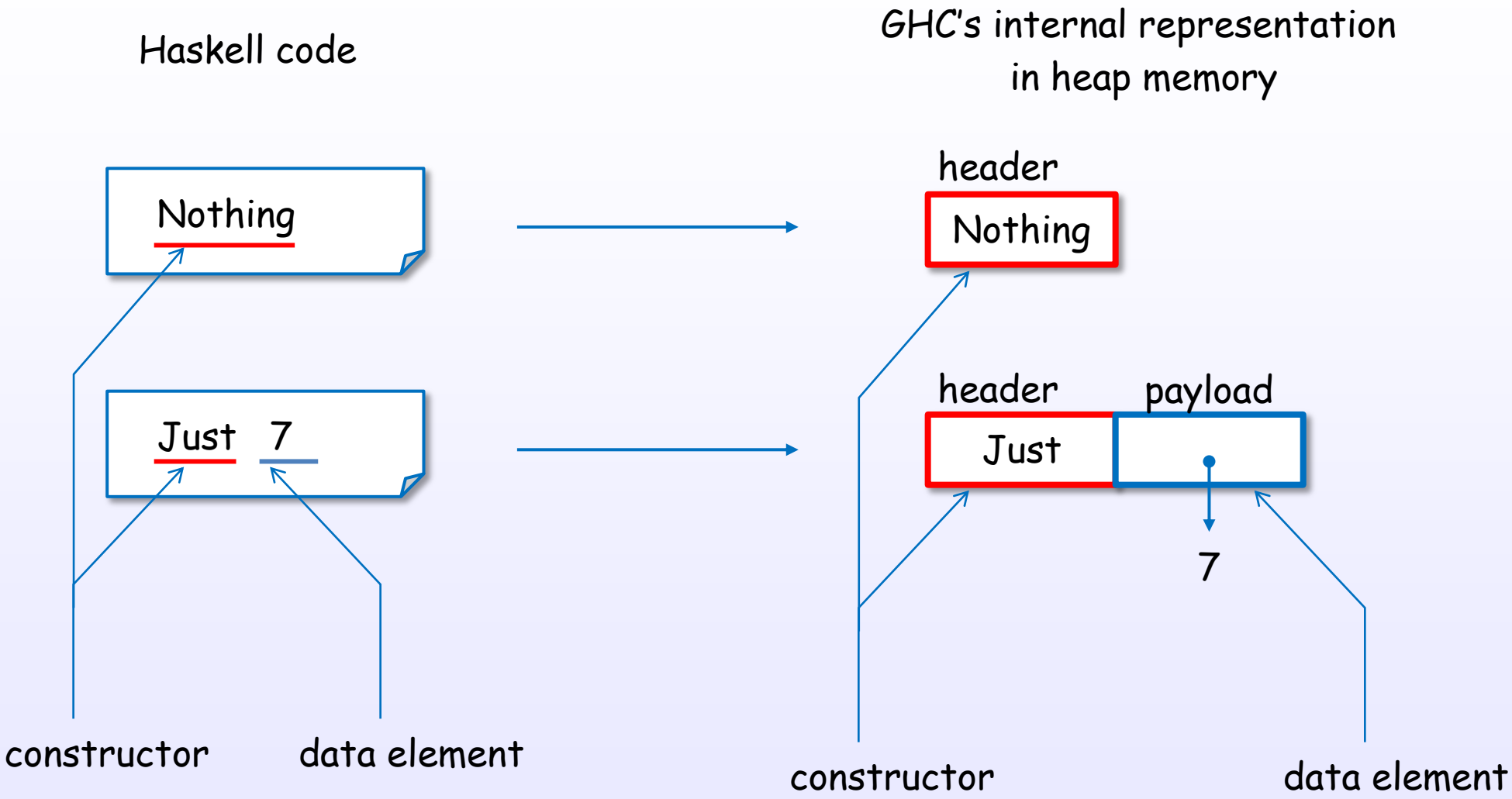
References : [1]

Constructorはdata文で宣言する代数的データ値



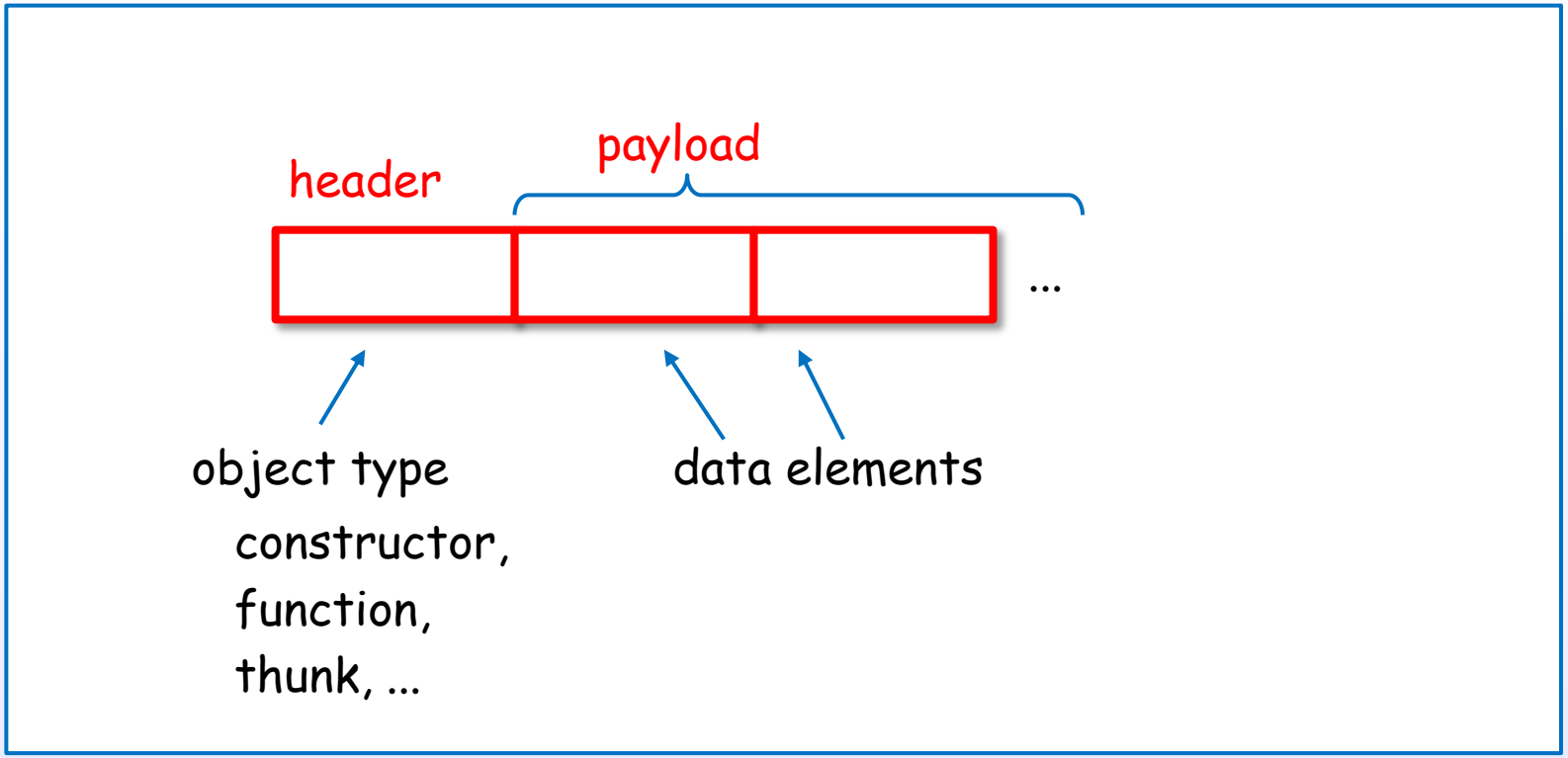
References : [1]

Constructorの内部表現



References : [1]

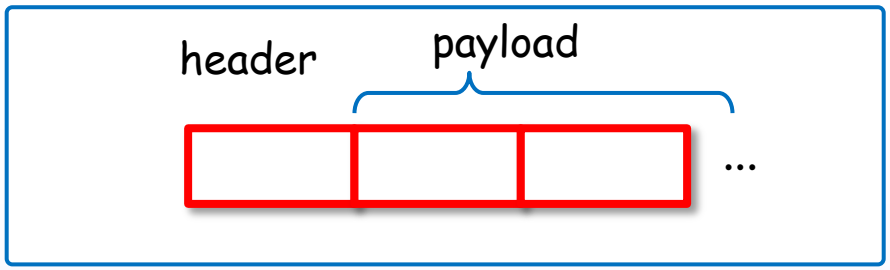
Constructorは統一内部表現で表現される



in heap memory, stack, registers or static memory

References : [1]

統一内部表現



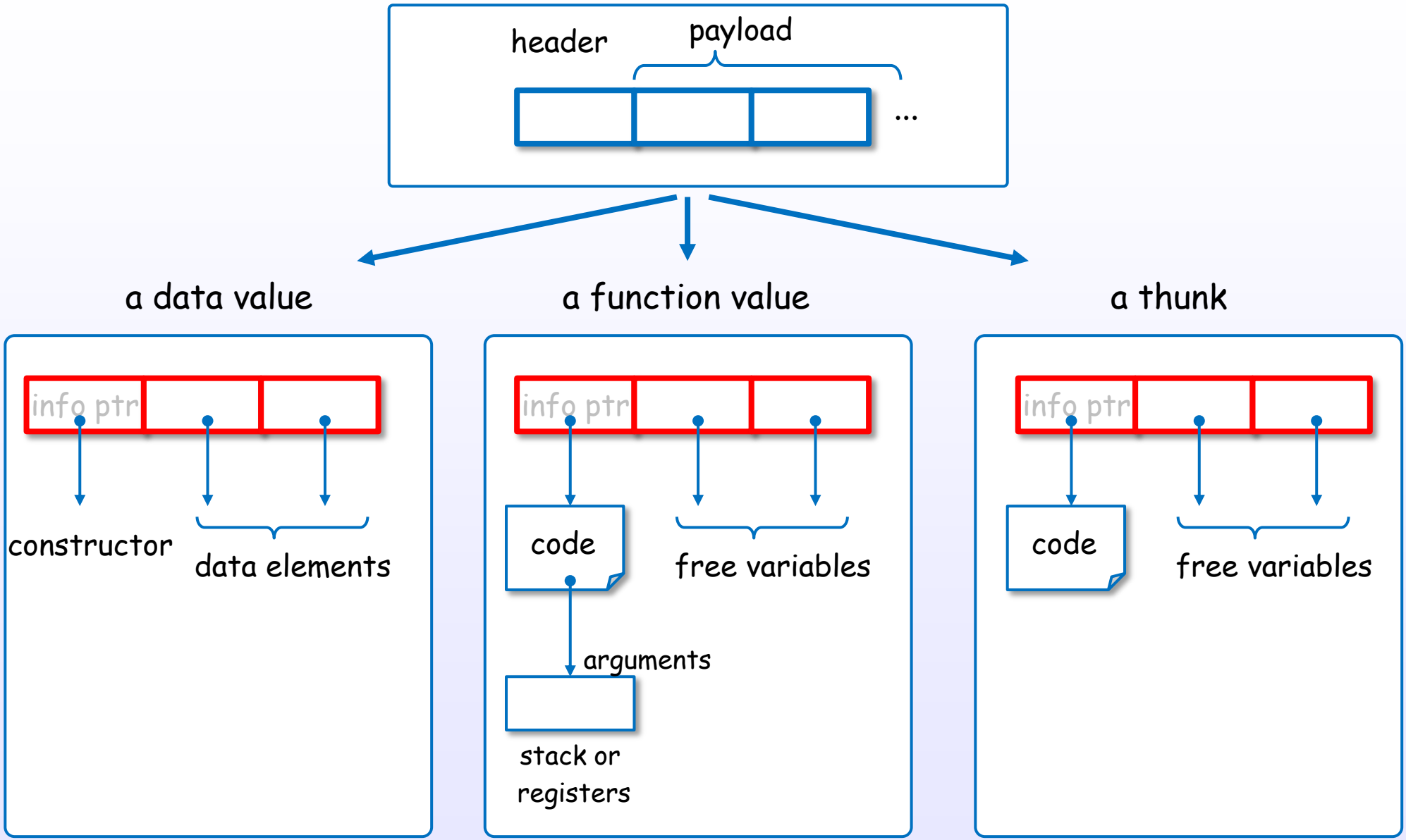
a data value

a function value

a thunk

References : [1]

統一内部表現



References : [1]

いろいろなコンストラクタと内部表現

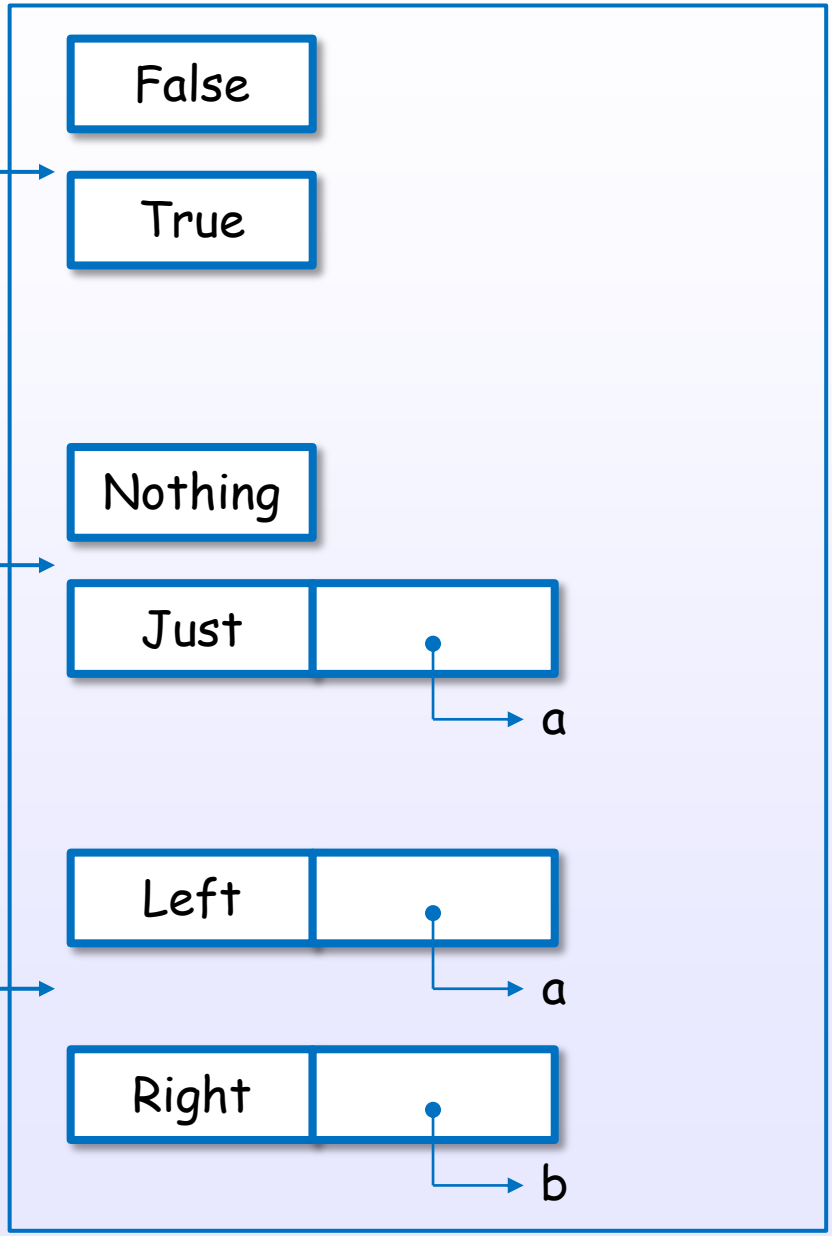
Haskell code

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

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

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

GHC's internal representation



References : [1]

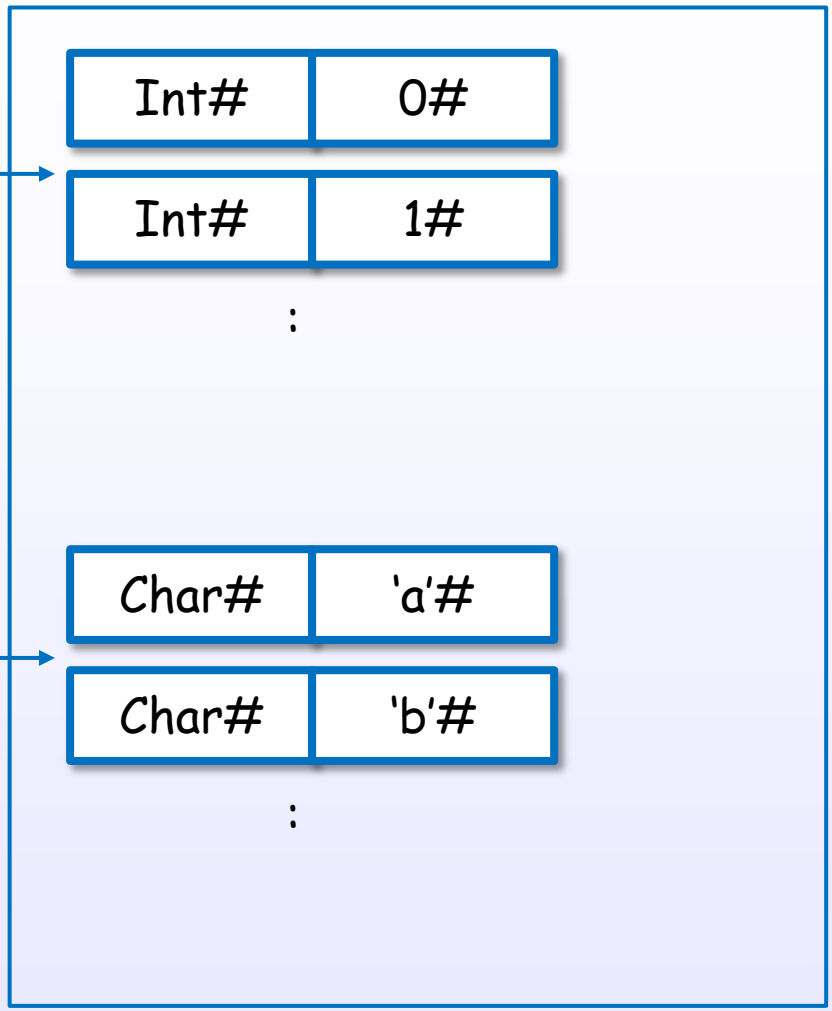
基本データ型も実はコンストラクタで構成されている

Haskell code

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

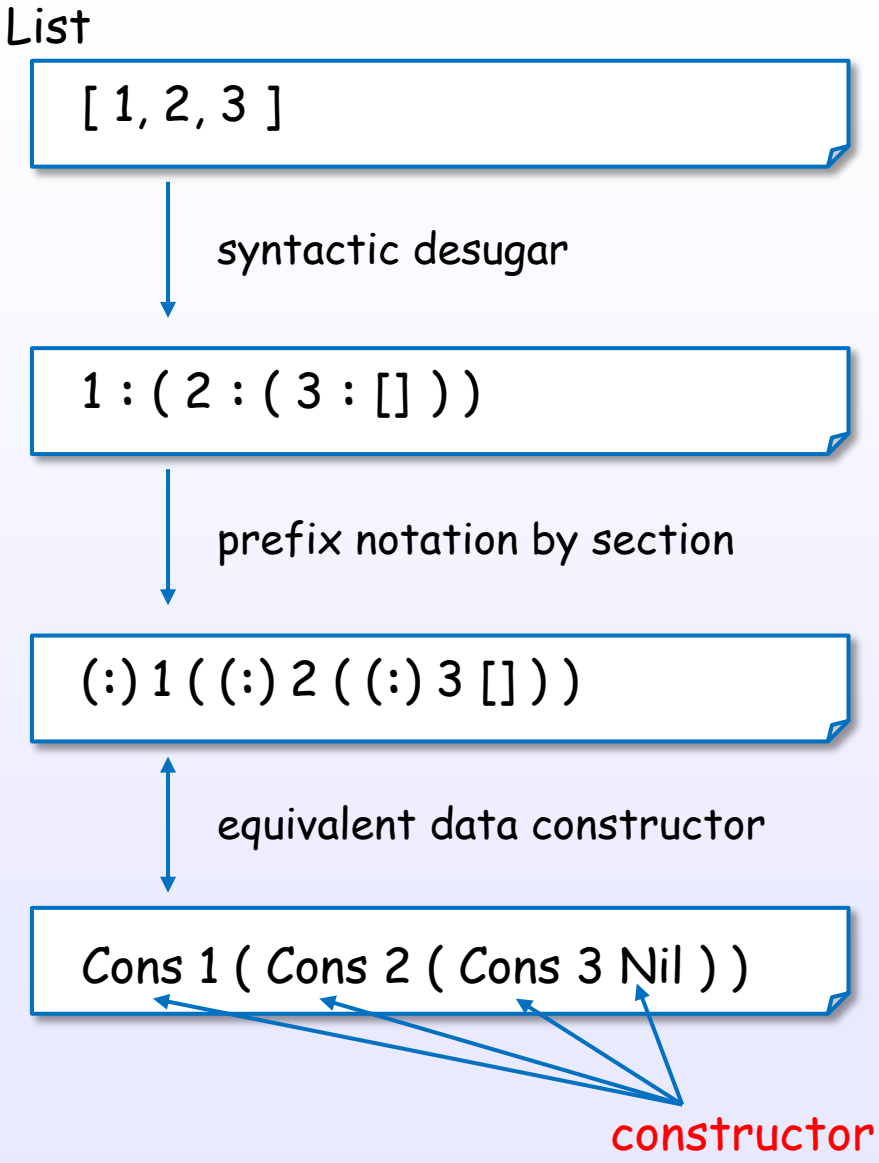
```
data Char = Char# 'a'#
          | Char# 'b'#
          | :
```

GHC's internal representation



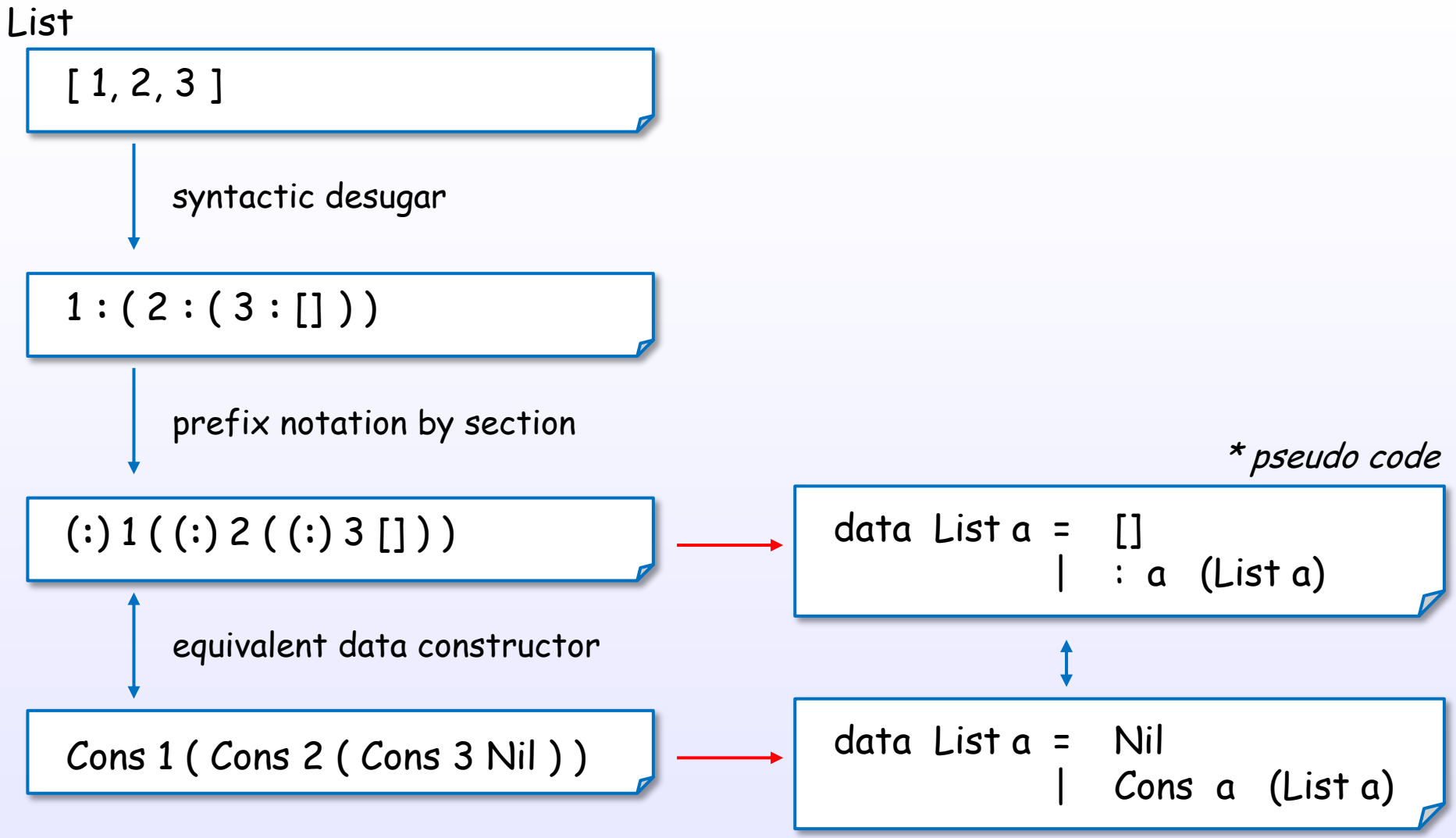
References : [1]

リストも実はコンストラクタで構成されている



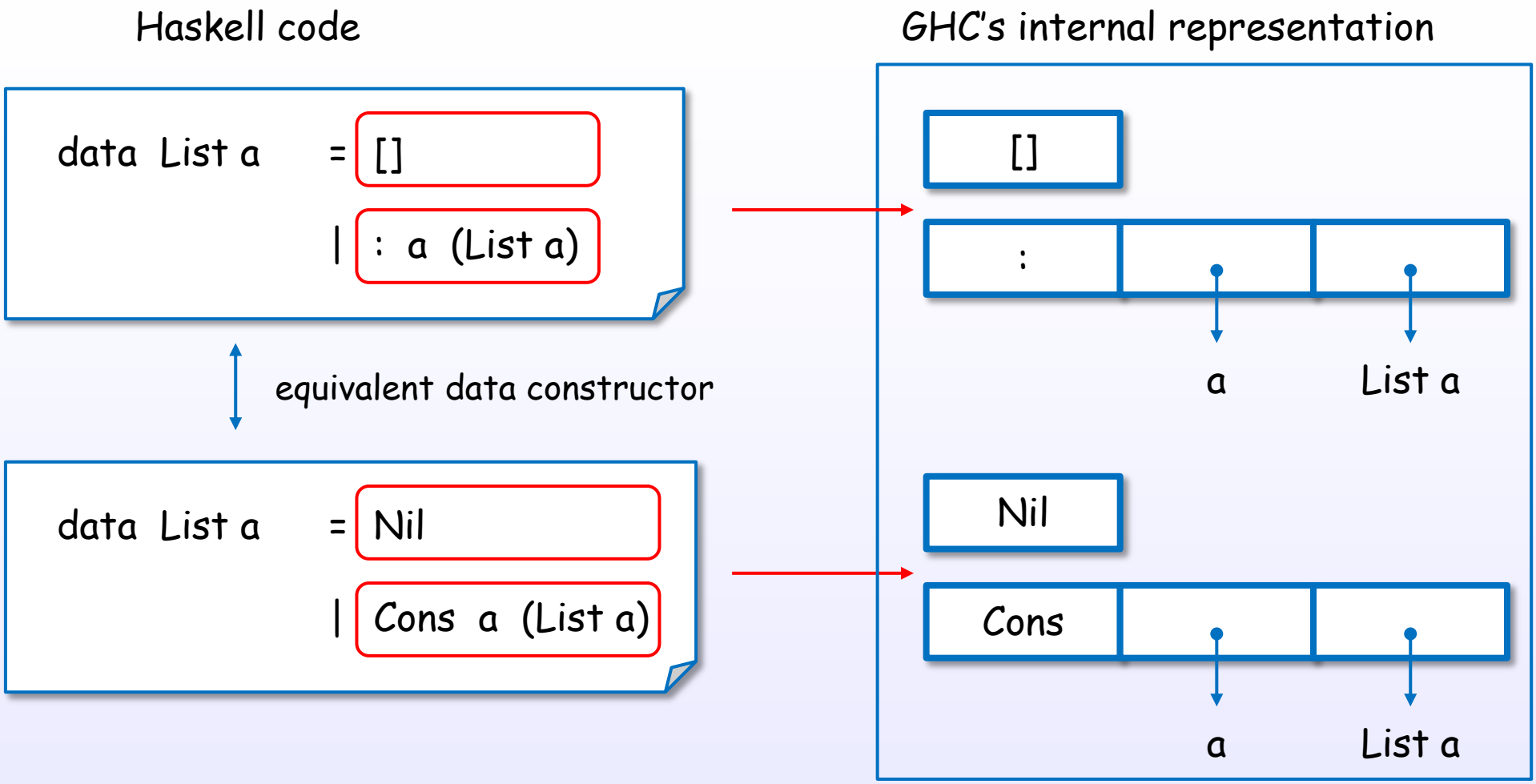
References : [1]

リストも実はコンストラクタで構成されている



References : [1]

リストも実はコンストラクタで構成されている



References : [1]

リストも実はコンストラクタで構成されている

Haskell code

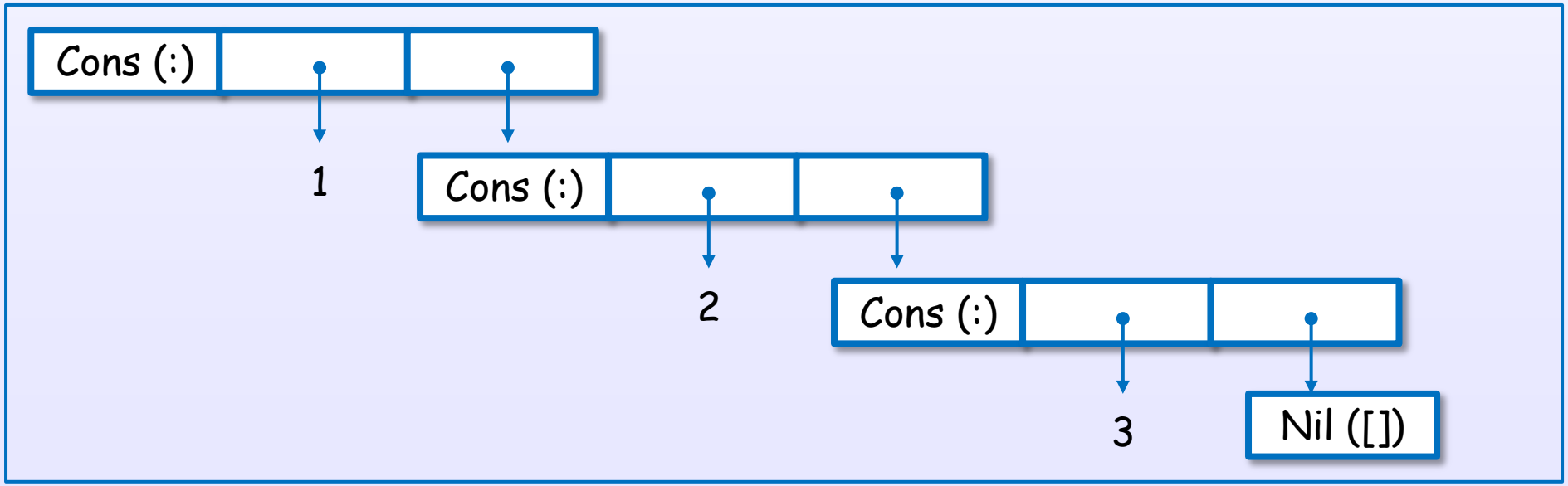
```
[ 1, 2, 3 ]
```

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

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

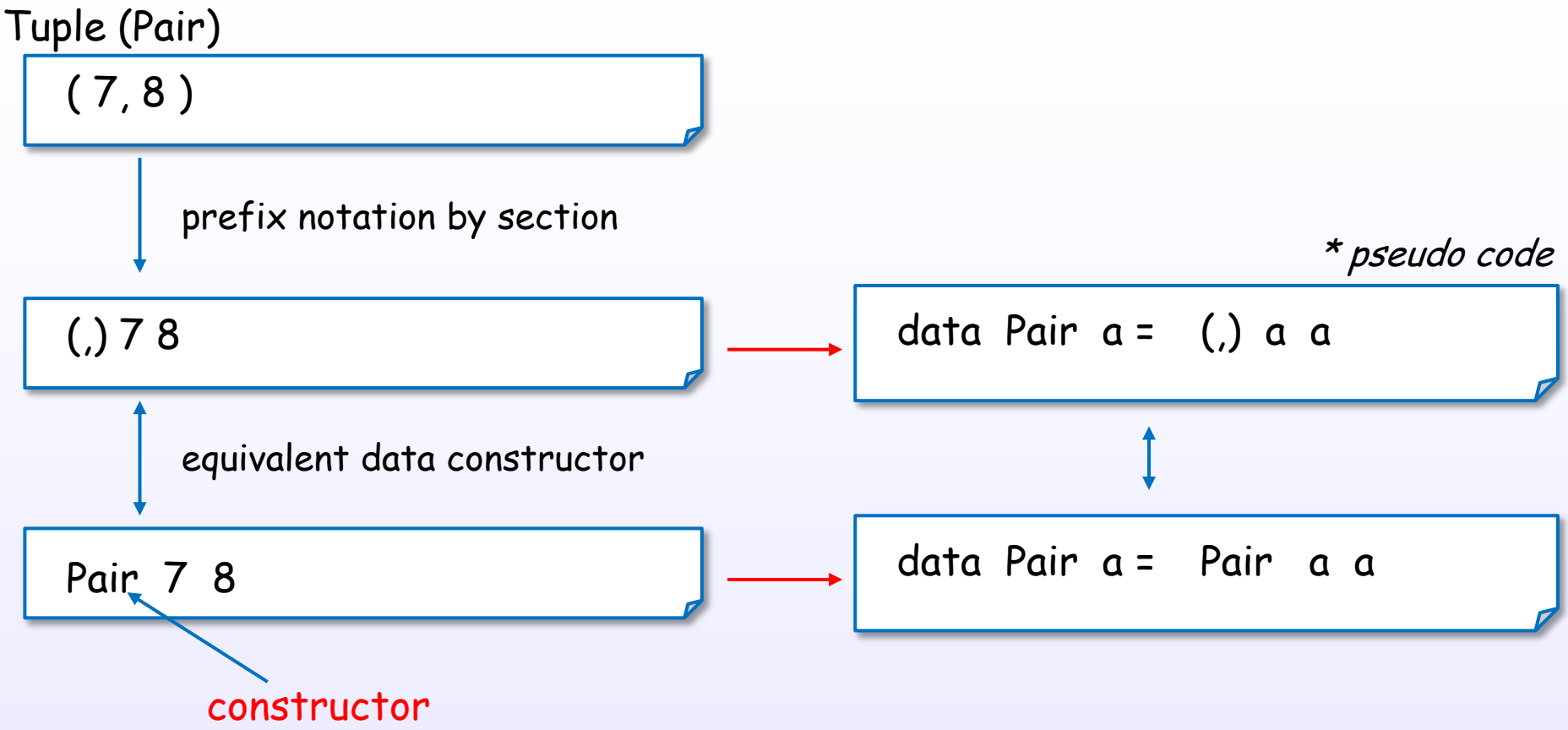
```
Cons 1 ( Cons 2 ( Cons 3 Nil ) )
```

GHC's internal representation



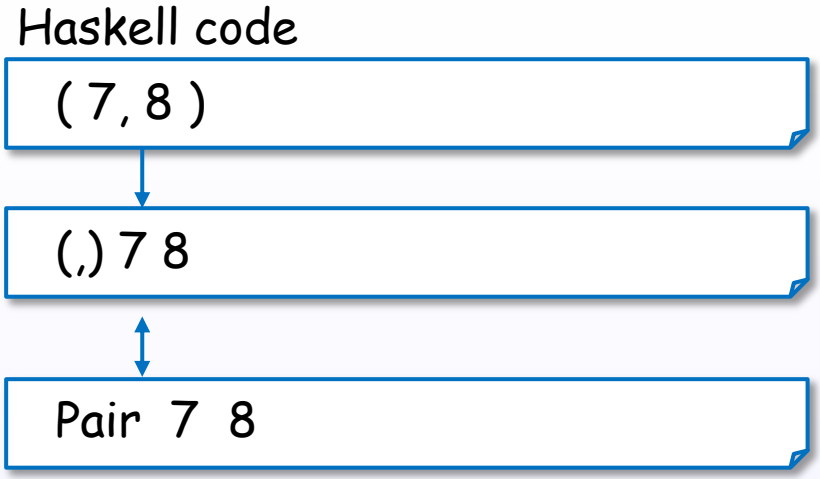
References : [1]

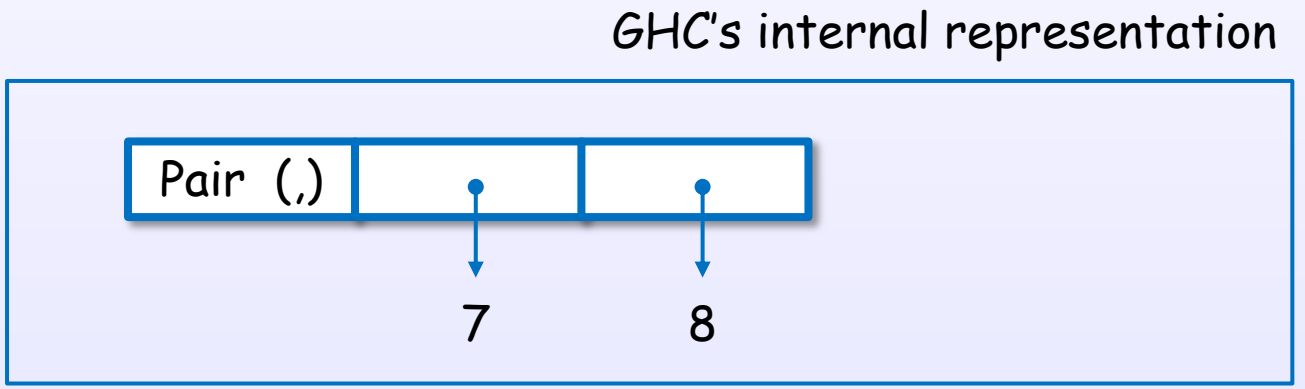
タプルも実はコンストラクタで構成されている



References : [1]

タプルも実はコンストラクタで構成されている





References : [1]

Thunk

Thunk

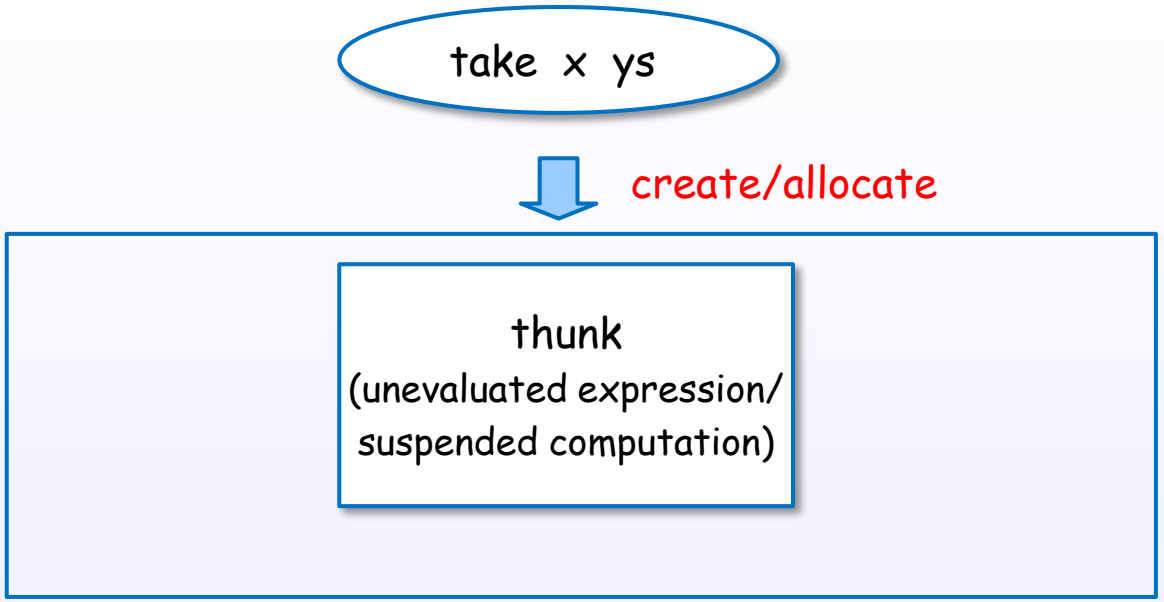
thunk
(unevaluated expression/
suspended computation)

A thunk is an unevaluated expression in heap memory.

References : [1]

Thunk

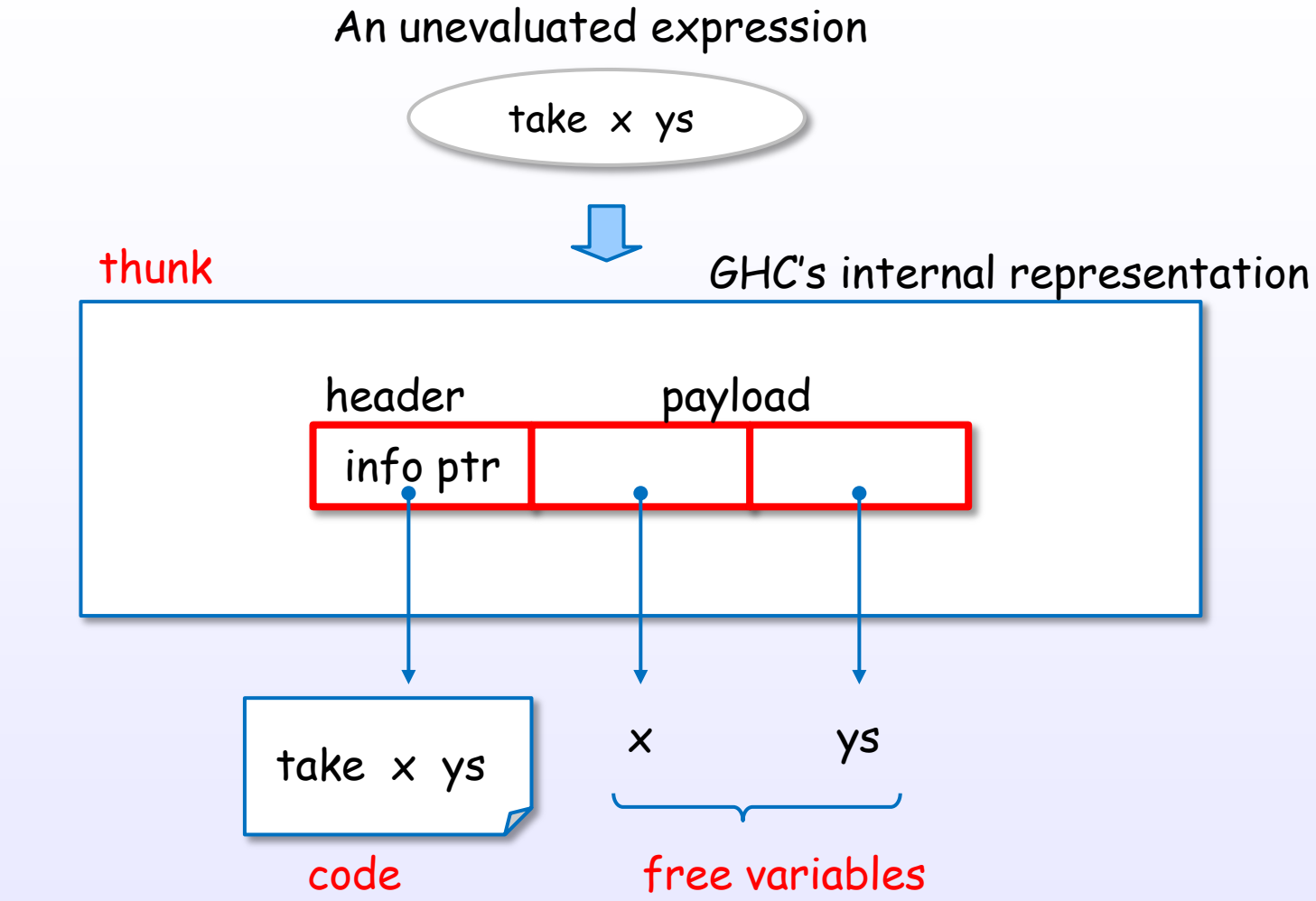
An unevaluated expression



A thunk is created for an unevaluated expression.

References : [1]

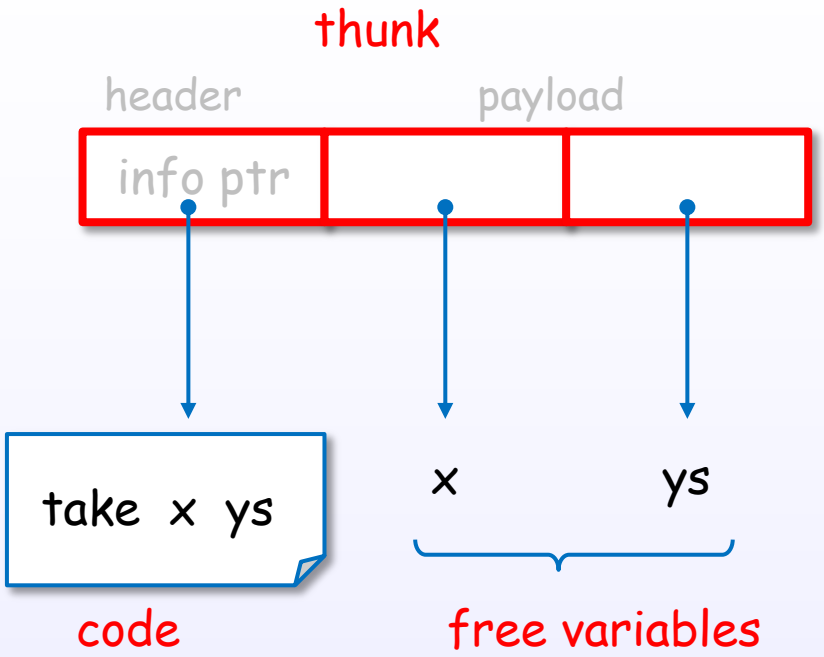
Thunkの内部表現



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

References : [1]

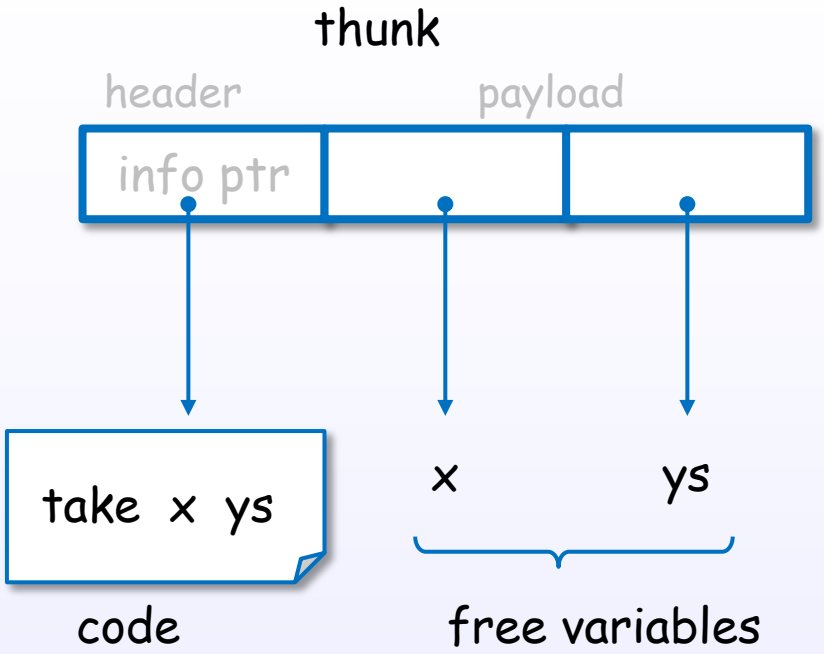
Thunkは、codeとfree variablesをパッケージ化したもの



A thunk is a package of code + free variables.

References : [1]

Thunkは、forcing要求により評価される



forcing

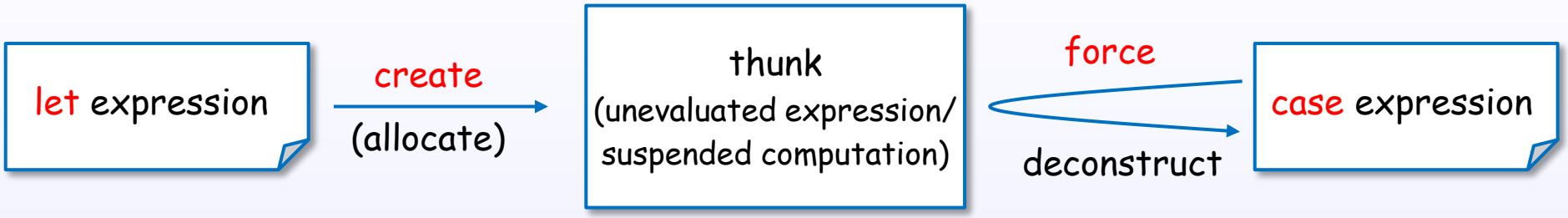
[1, 2, 3]

A evaluated expression

References : [1]

let, case expression

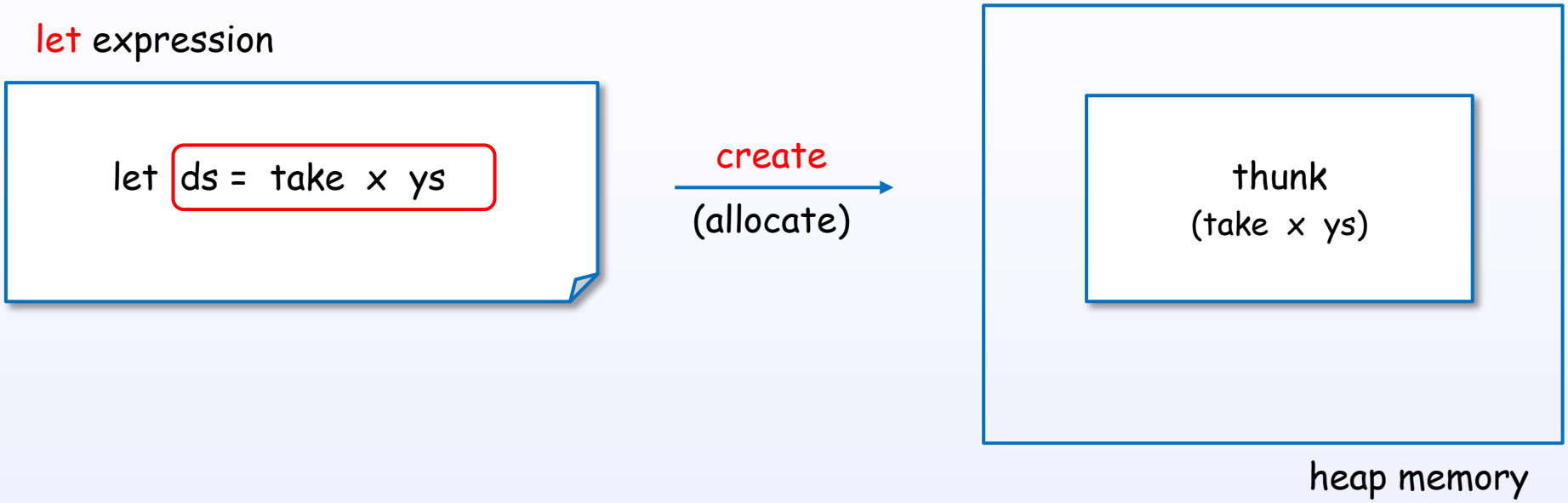
let/case expressions and thunk



A let expression may create a thunk.
A case expression forces and deconstructs the thunk.

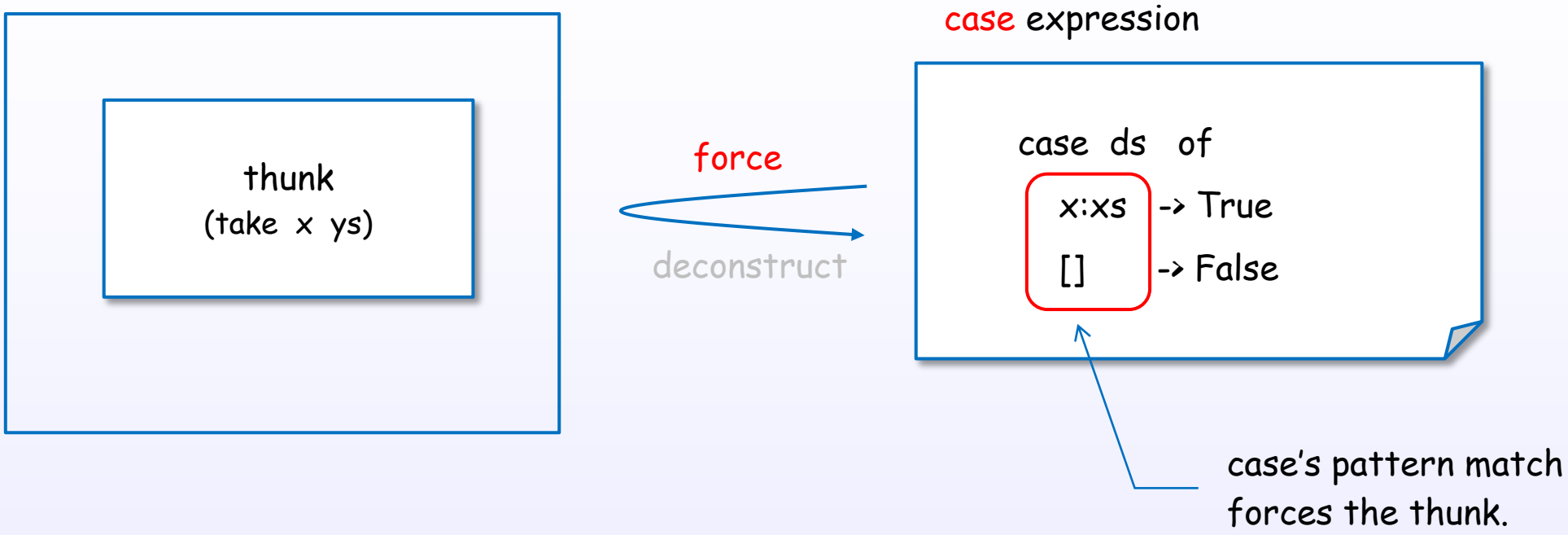
References : [1]

A let expression creates a thunk



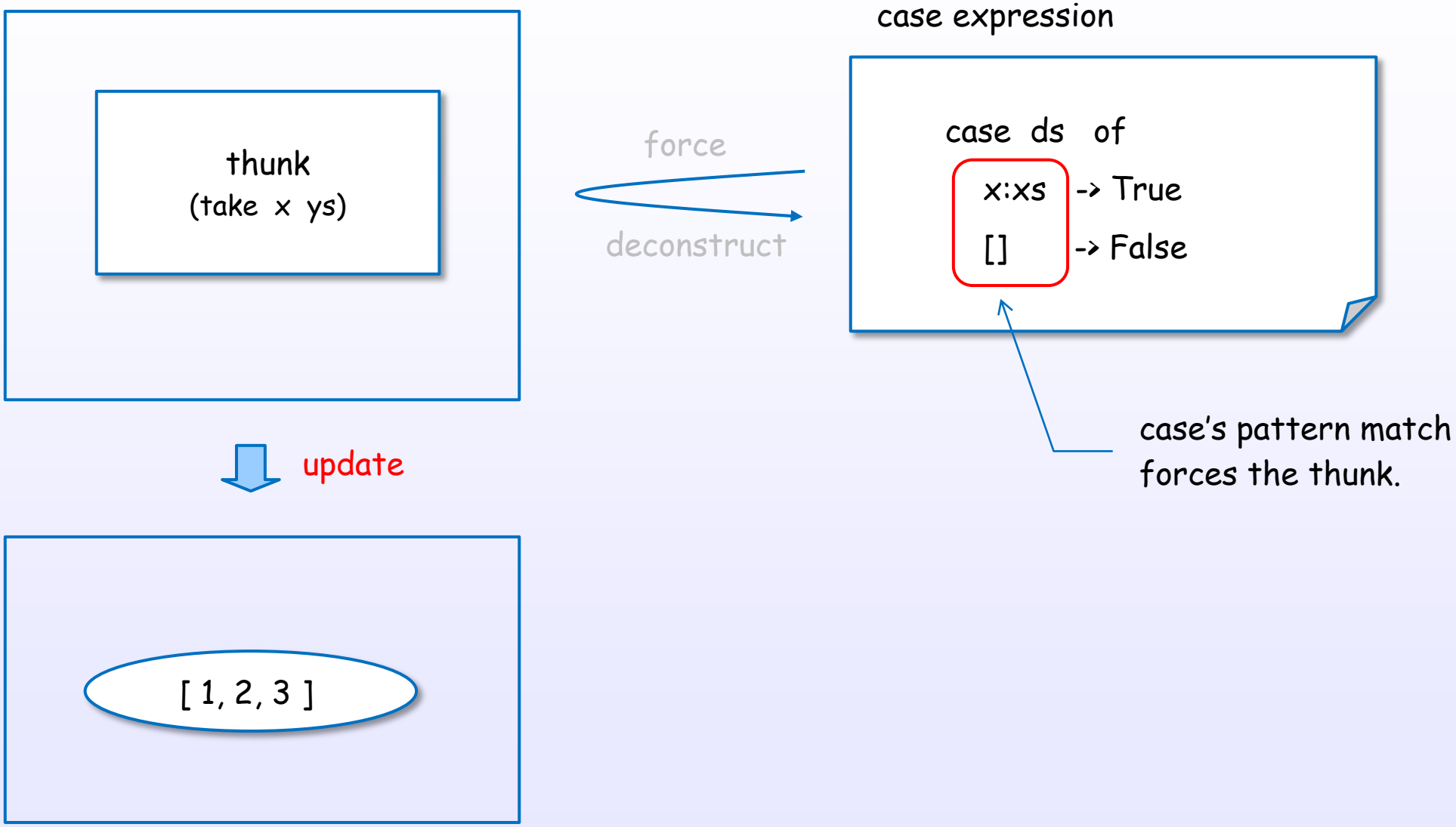
References : [1]

A case expression forces a thunk



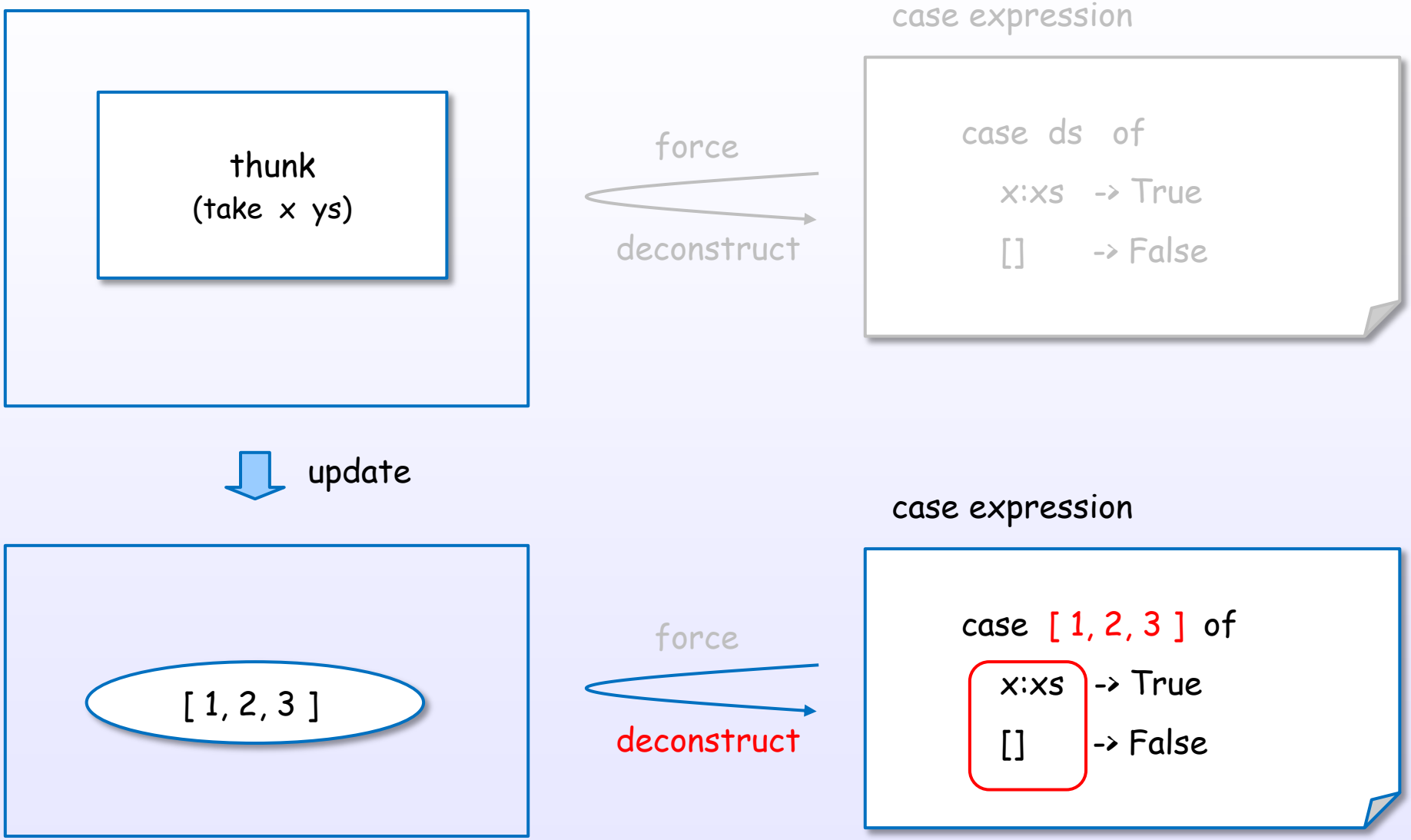
References : [1]

A case expression forces a thunk



References : [1]

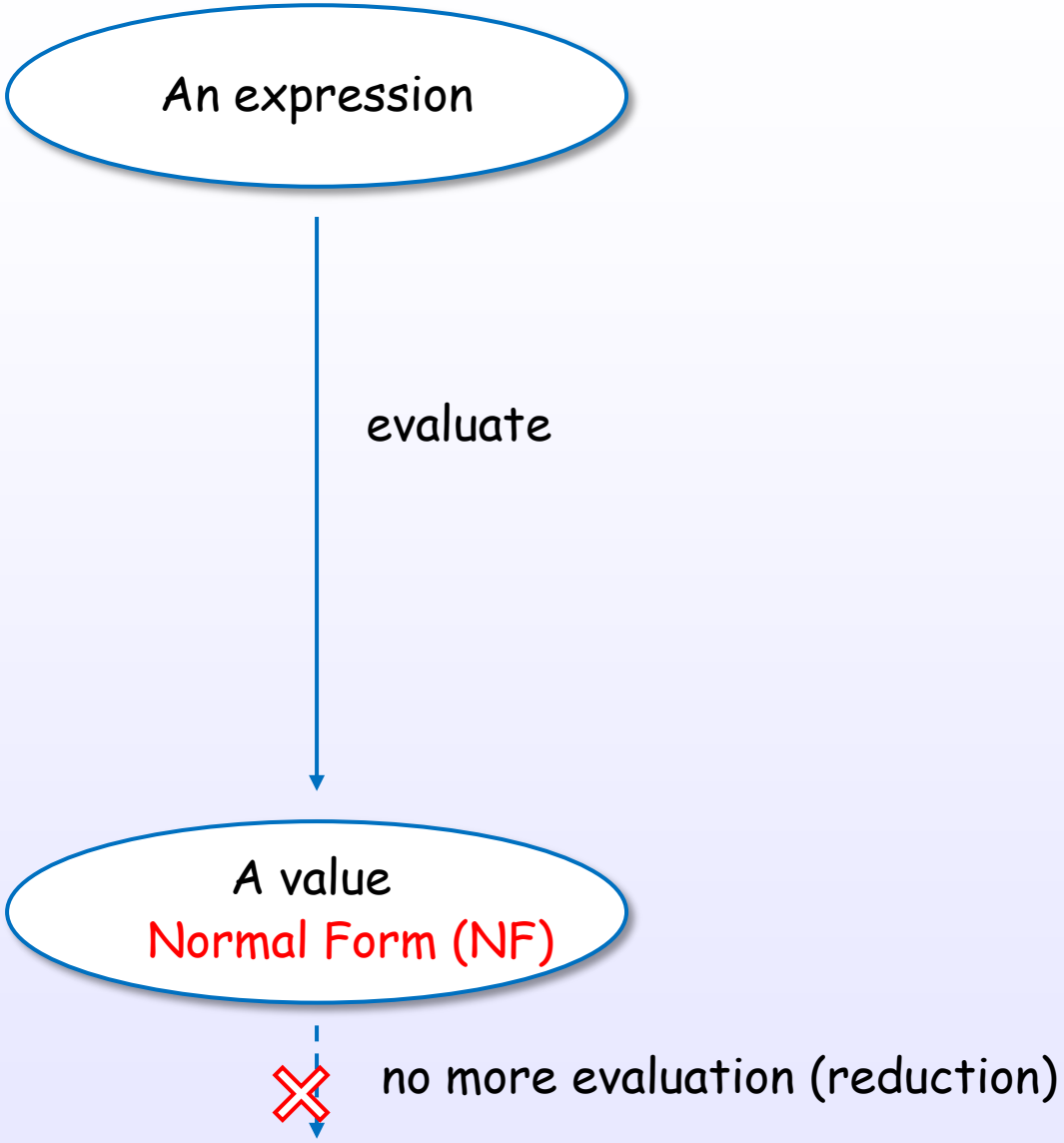
A case expression forces a thunk



References : [1]

WHNF

Normal form は、これ以上評価できない値



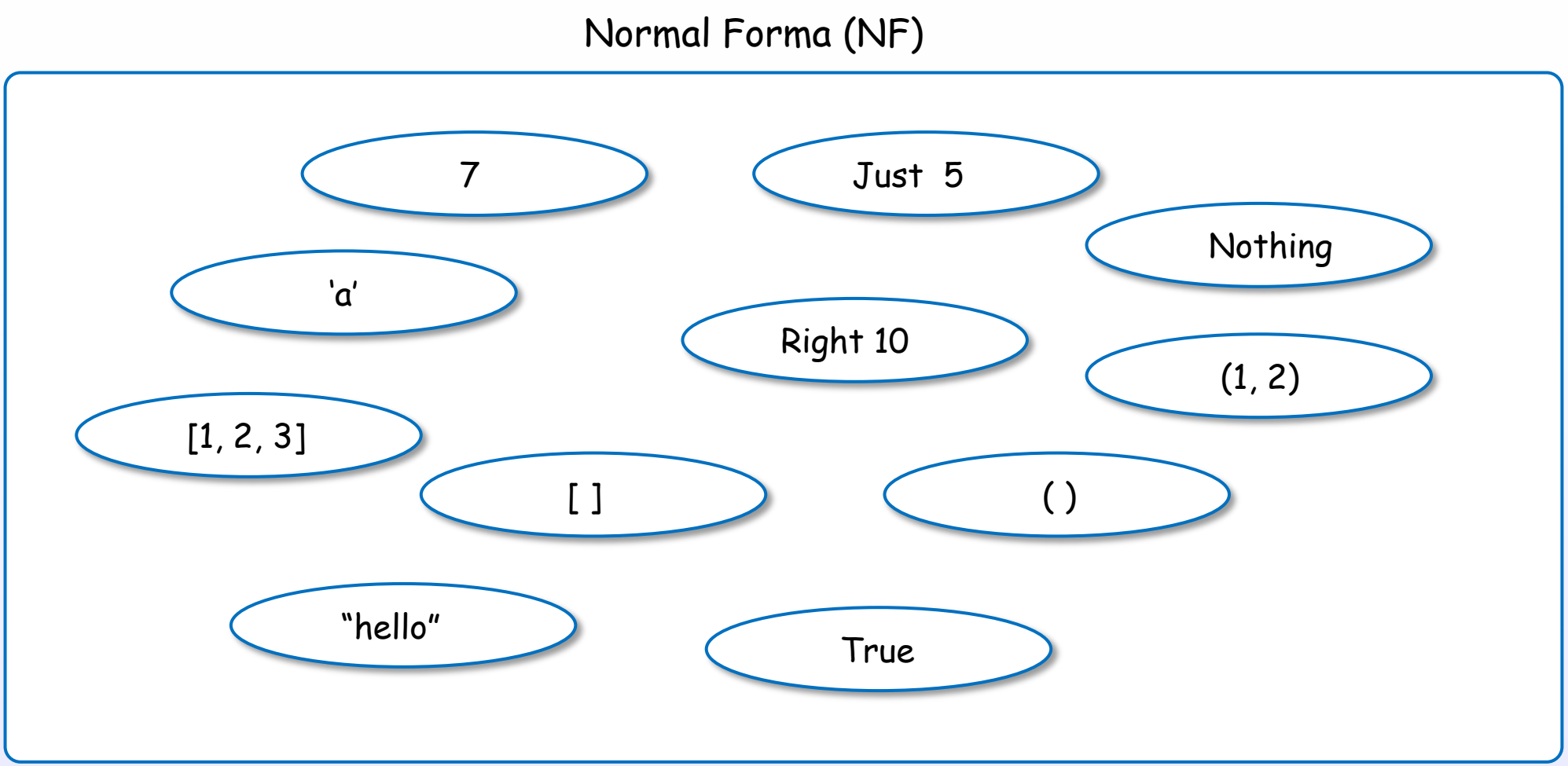
[Terei]

[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

References : [1]

Examples of normal form (NF)



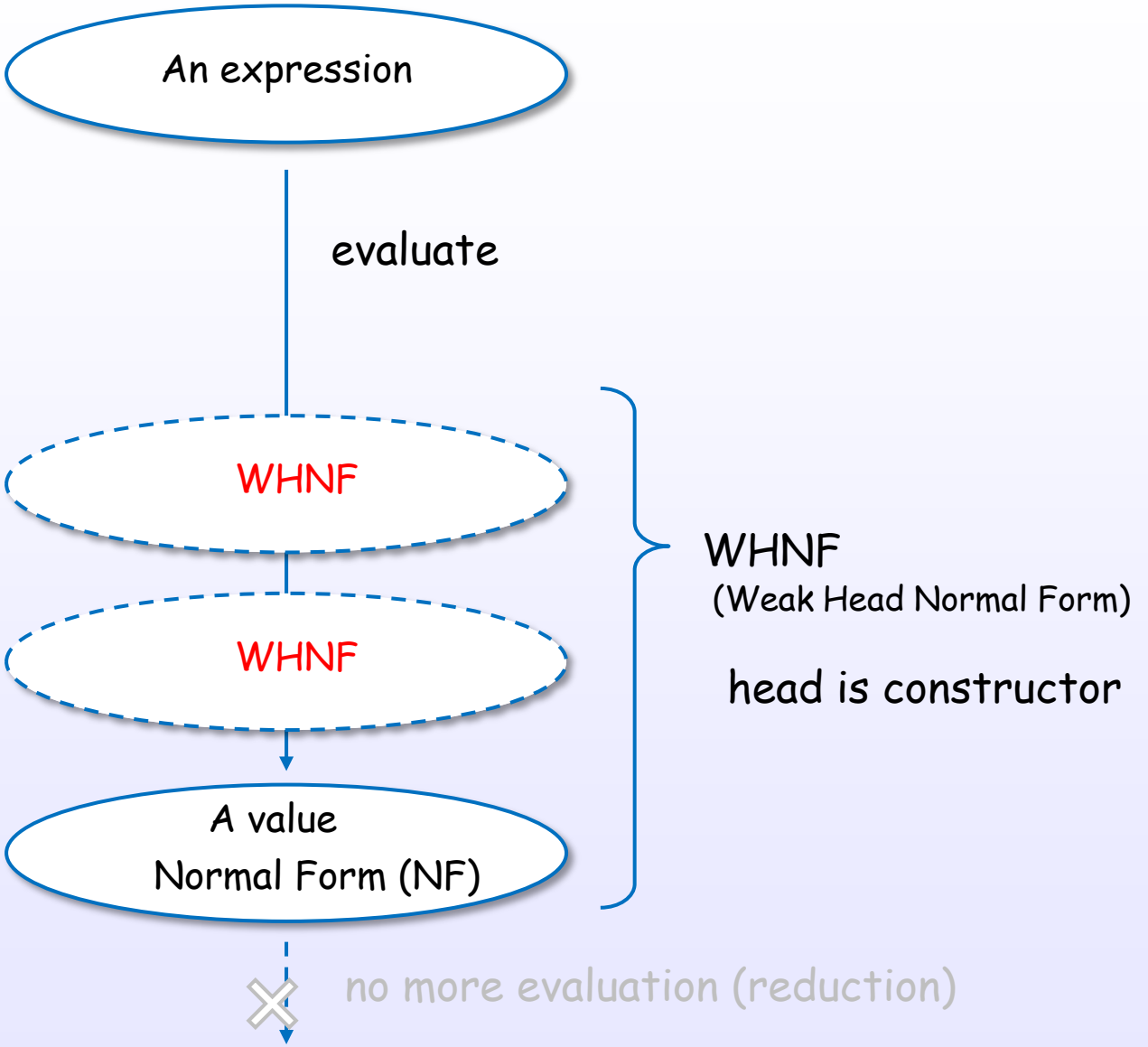
[Terei]

[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

References : [1]

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



[Terei]

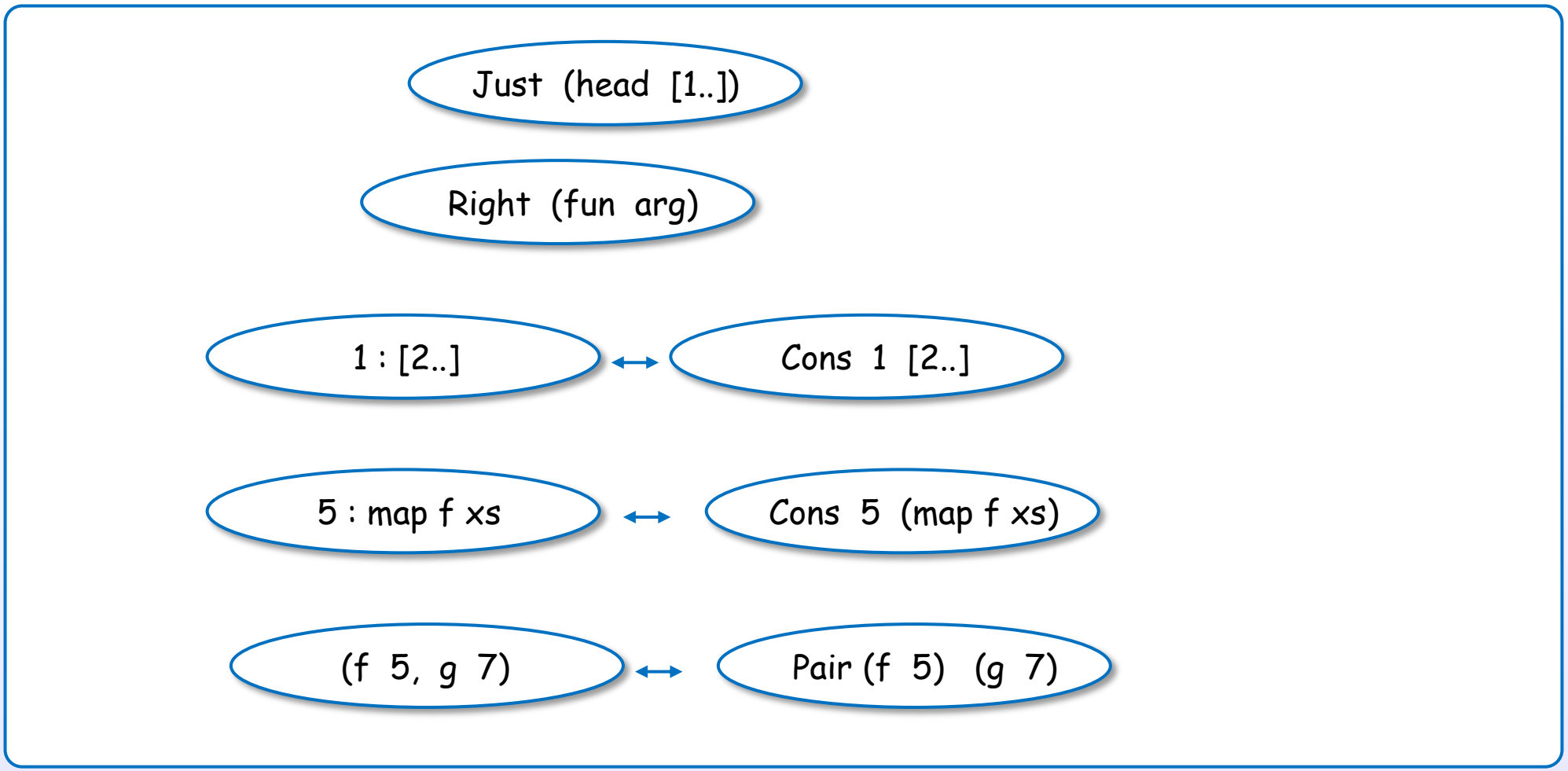
[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

References : [1]

Examples of weak head normal form (WHNF)

Weak Head Normal Forma (WHNF)



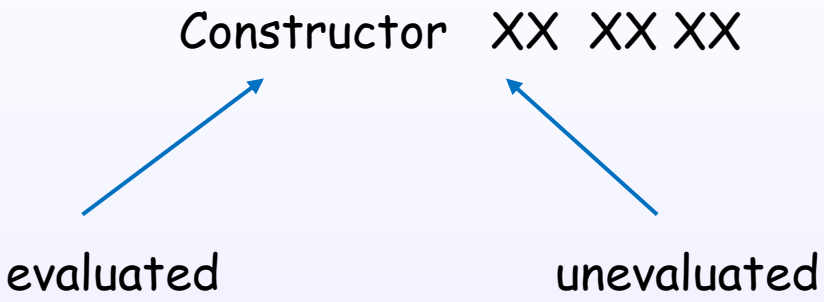
[Terei]

[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

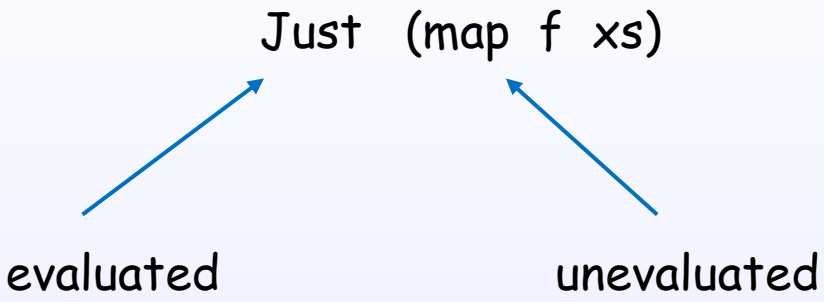
References : [1]

WHNF



References : [1]

WHNF



References : [1]

WHNF

前頁の、heap objectイメージ

References : [1]

WHNF

[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]

References : [1]

WHNF

[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)

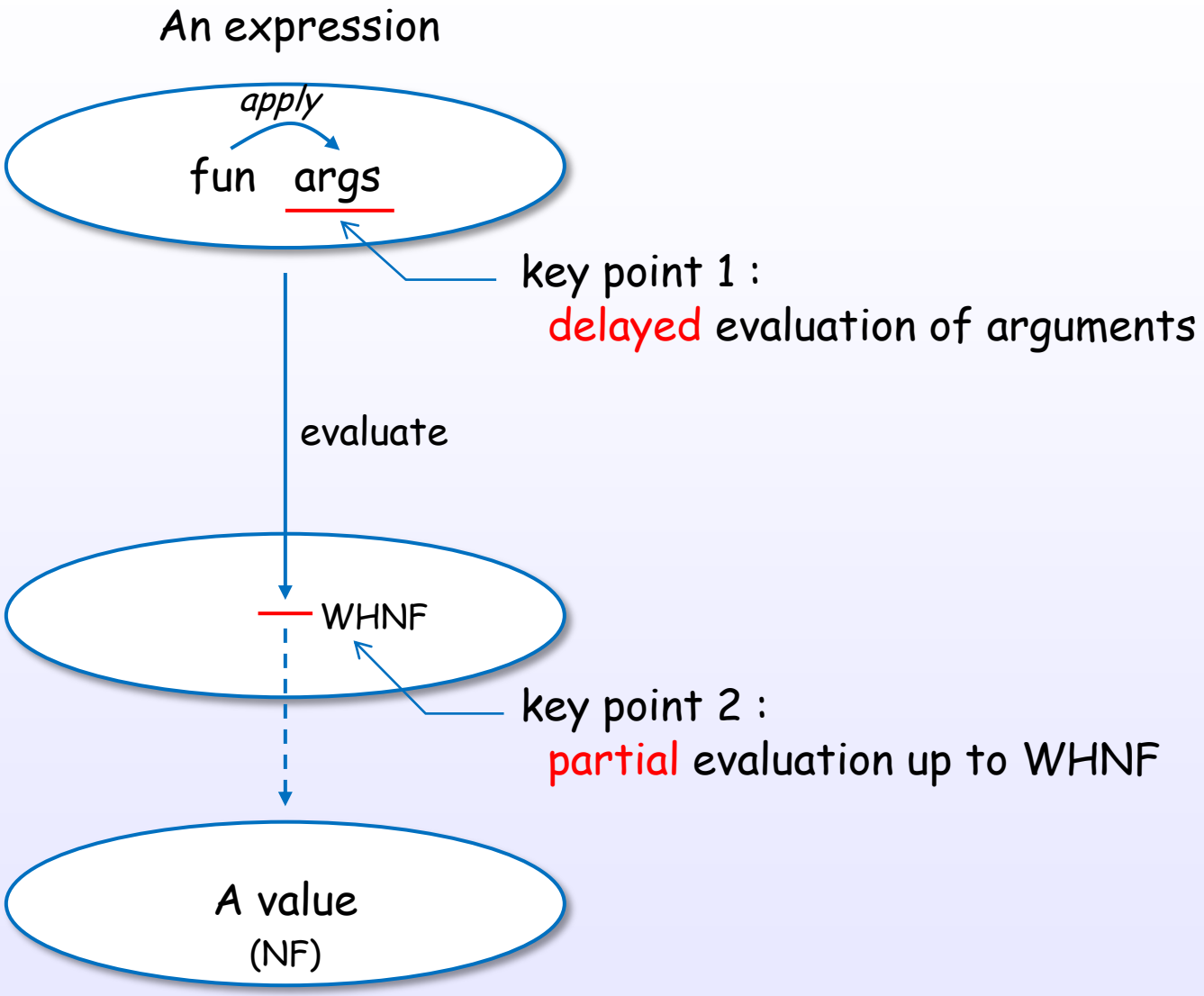
no-strict evaluation Vs. lazy evaluation:
non-strict: Includes both call-by-name and call-by-need, general term for evaluation strategies that don't evaluate arguments before entering a function
lazy evaluation: Specific type of non-strict evaluation. Uses call-by-need (for sharing).

[Terei]

References : [1]

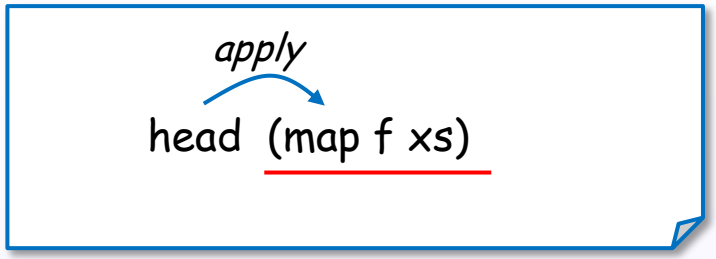
Evaluation in Haskell (GHC)

Key concept of Haskell's lazy evaluation

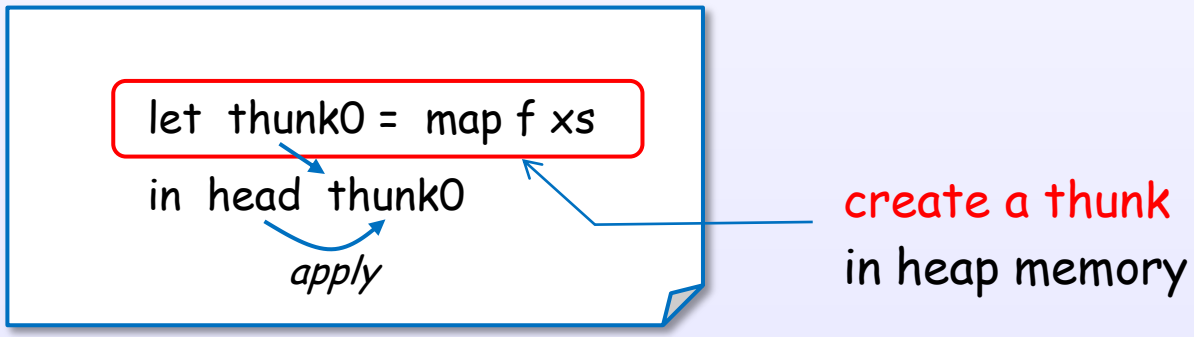


References : [1]

key point 1 : delayed evaluation of arguments



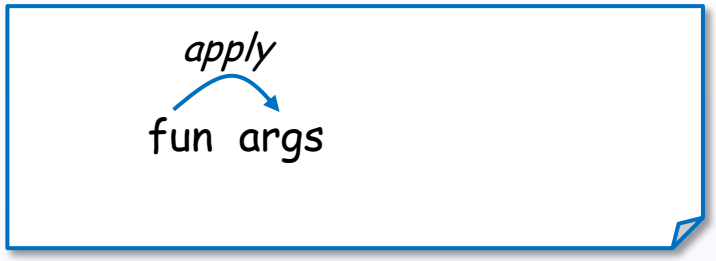
internal transformation by GHC



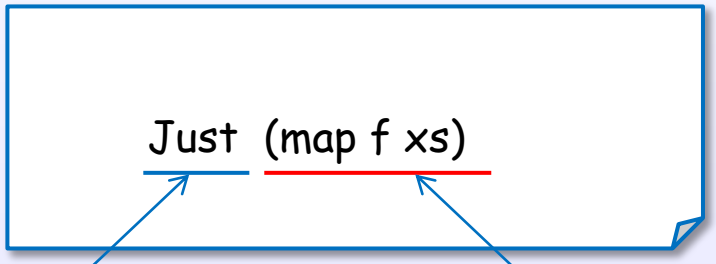
GHC implements lazy evaluation using the thunk.
Evaluation of arguments is delayed with the thunk.

References : [1]

key point 2 : partial evaluation up to WHNF



evaluation up to WHNF



evaluated part
(head constructor)

unevaluated part
(thunk)

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

References : [1]

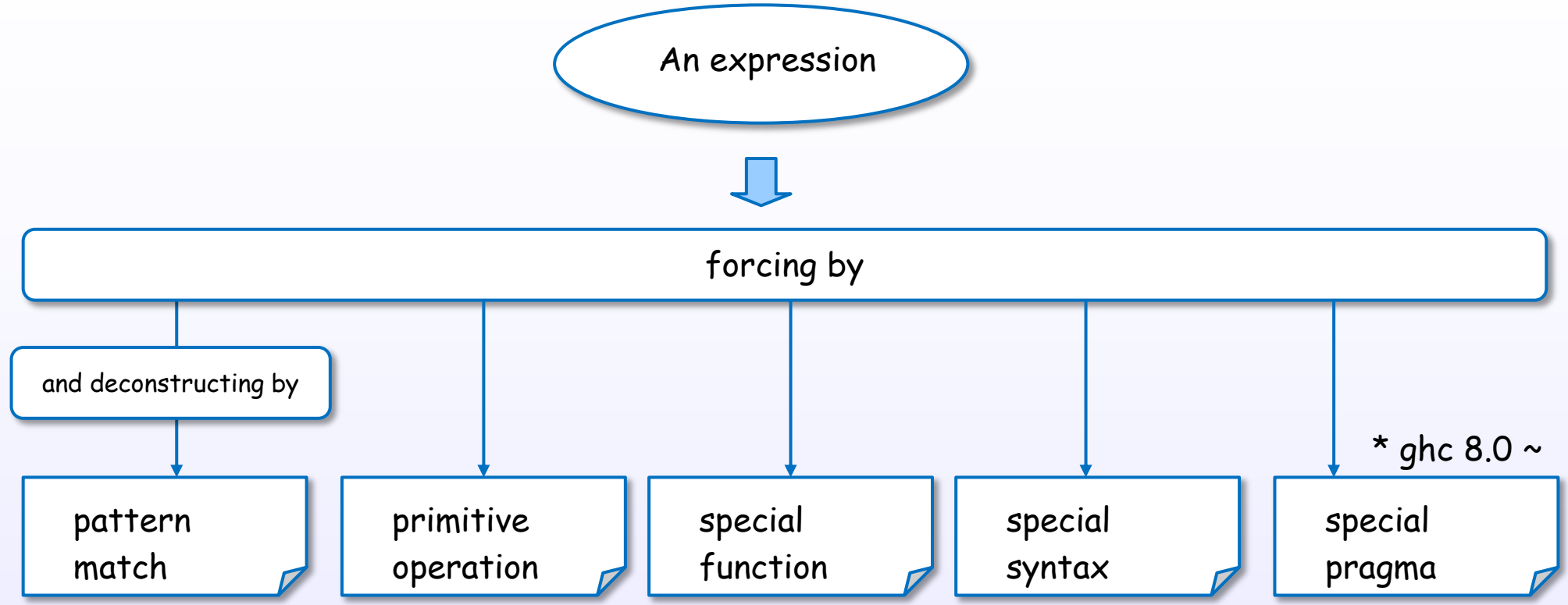
Pattern match

[CIS194]

References : [1]

Control the evaluation in Haskell

How to drive evaluation



References : [1]

Examples of evaluation steps

Example of repeat

repeat 1

1 : repeat 1

1 : 1 : repeat 1

1 : 1 : 1 : repeat 1

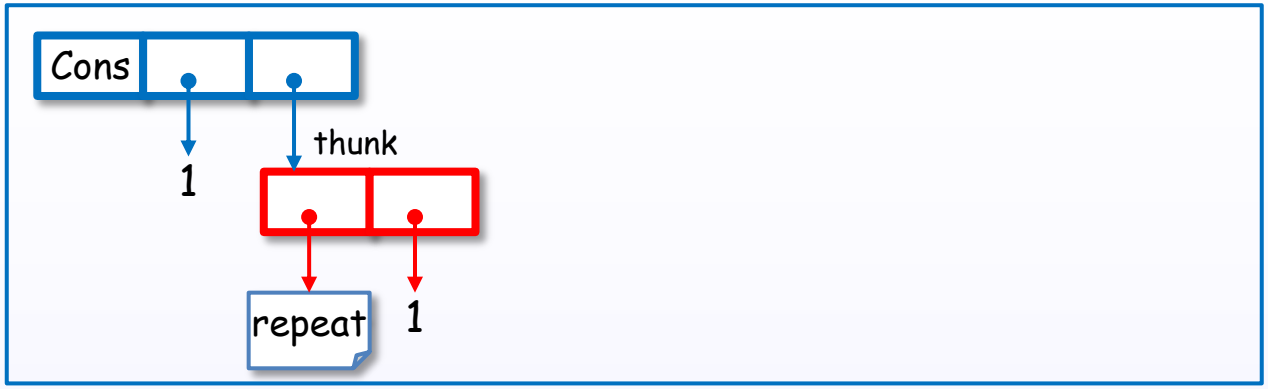
References : [1]

Example of repeat

repeat 1

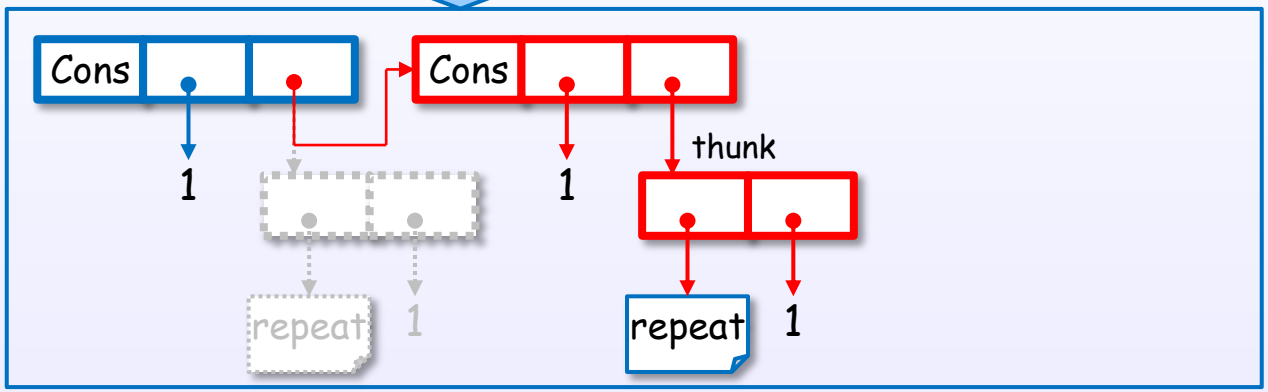
1 : repeat 1

Cons 1 (repeat 1)



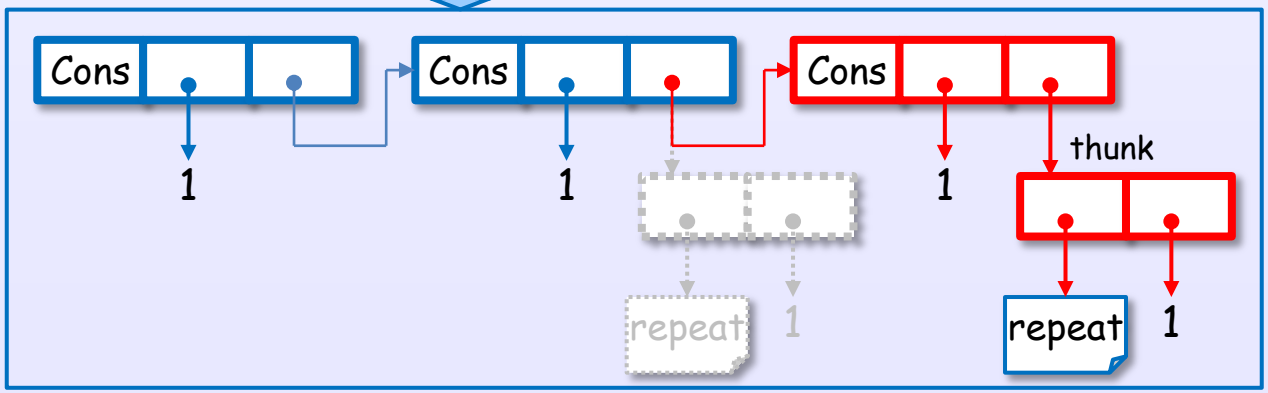
1 : 1 : repeat 1

Cons 1 (Cons 1 (repeat 1))



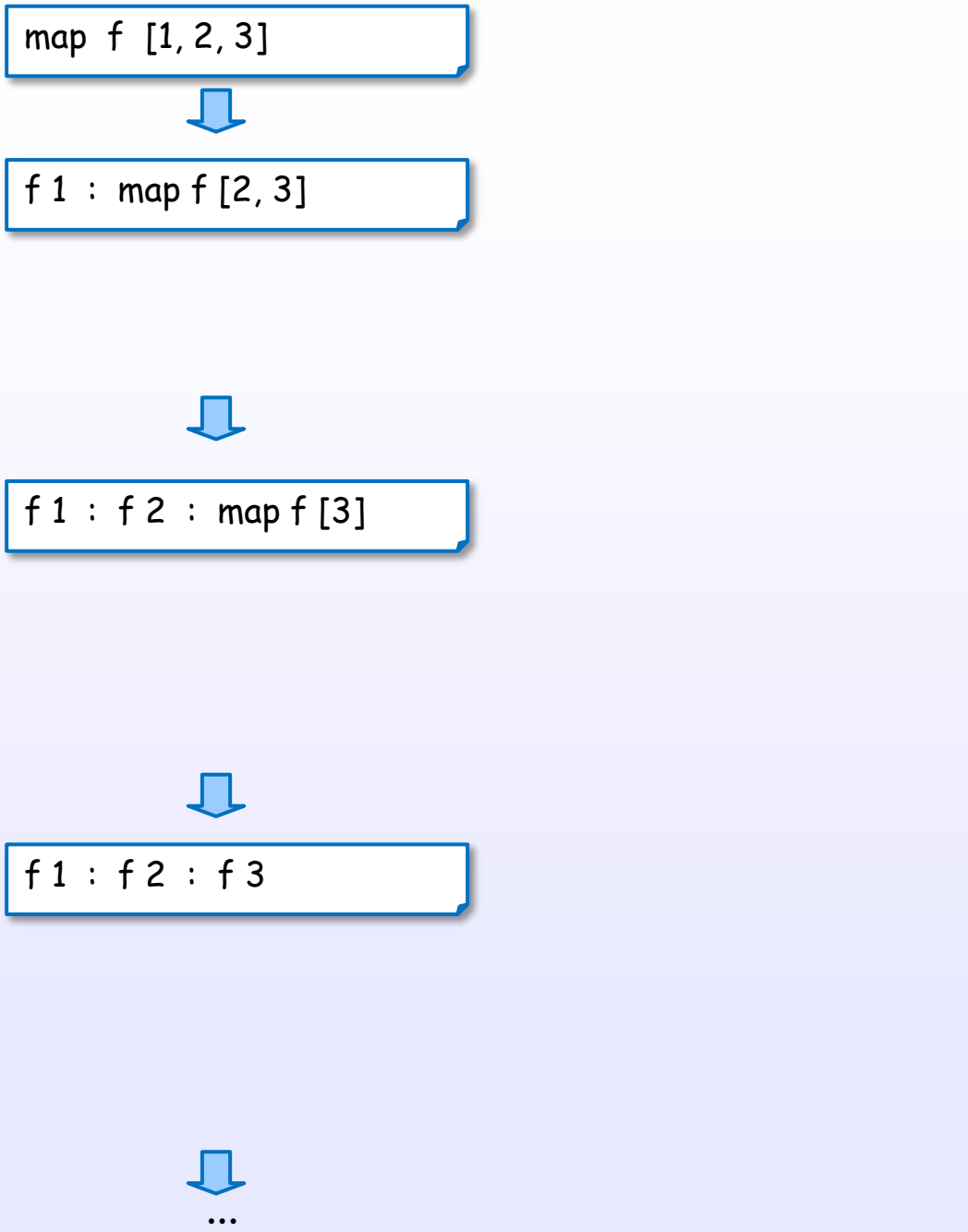
1 : 1 : 1 : repeat 1

Cons 1 (Cons 1 (Cons 1 (repeat 1)))



References : [1]

Example of map



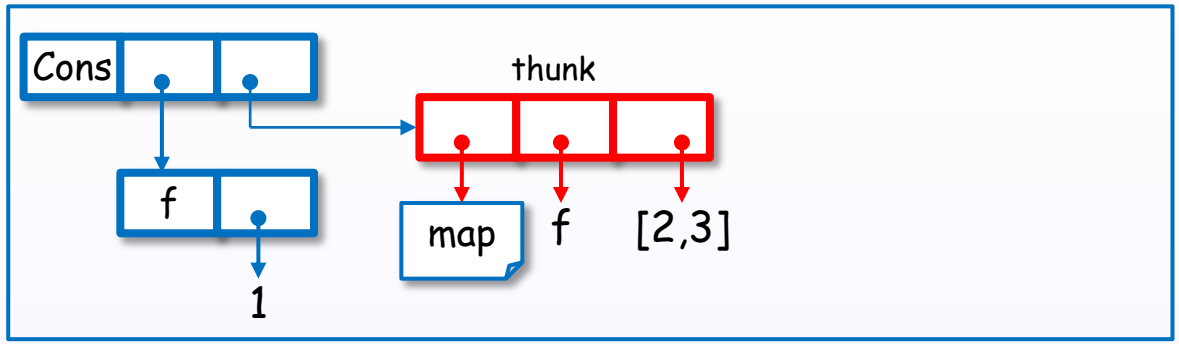
References : [1]

Example of map

map f [1, 2, 3]

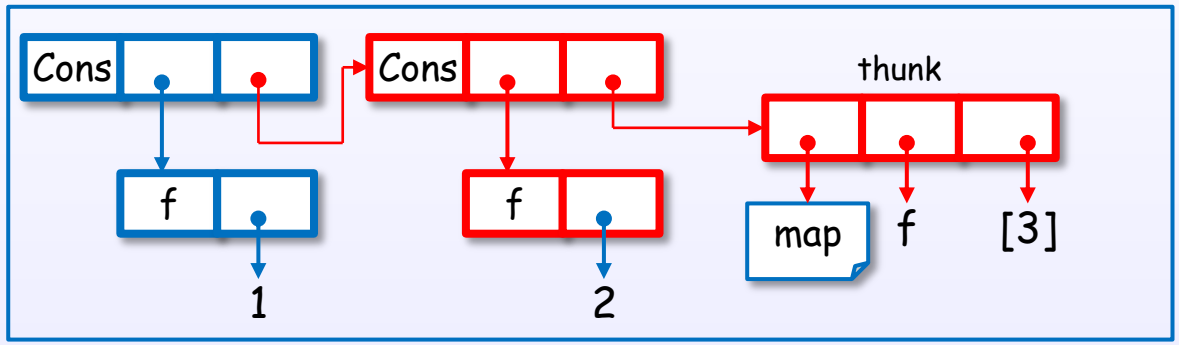
f 1 : map f [2, 3]

Cons (f 1) (map f [2, 3])



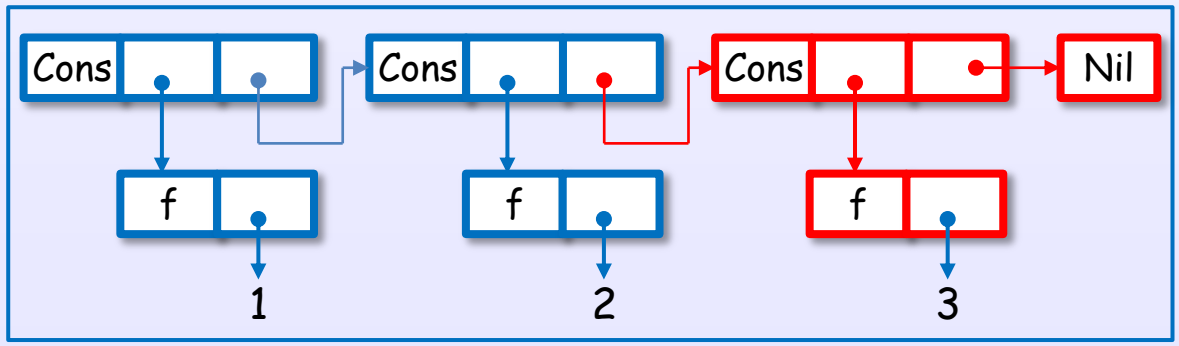
f 1 : f 2 : map f [3]

Cons (f 1) (Cons (f 2) (map f [3]))



f 1 : f 2 : f 3

Cons (f 1) (Cons (f 2) (Cons (f 3) Nil))

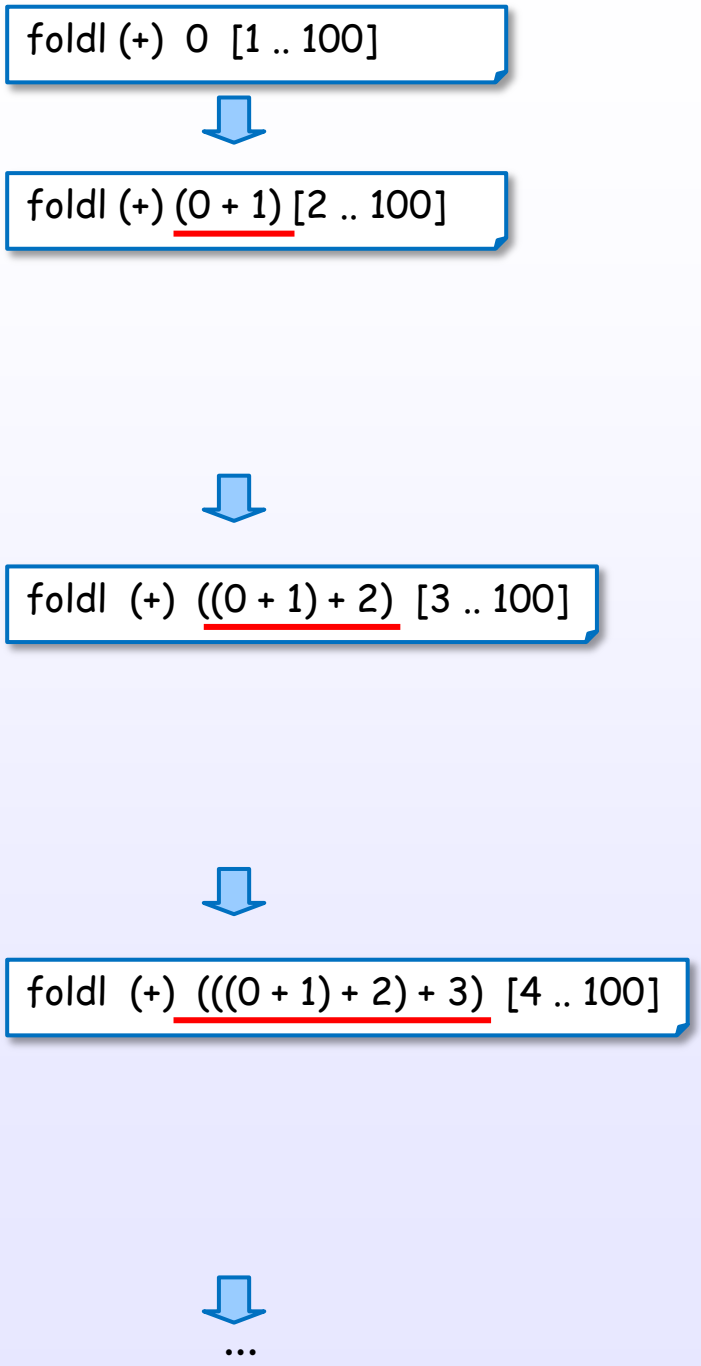


...

...

References : [1]

Example of foldl (non-strict)



References : [1]

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]
```

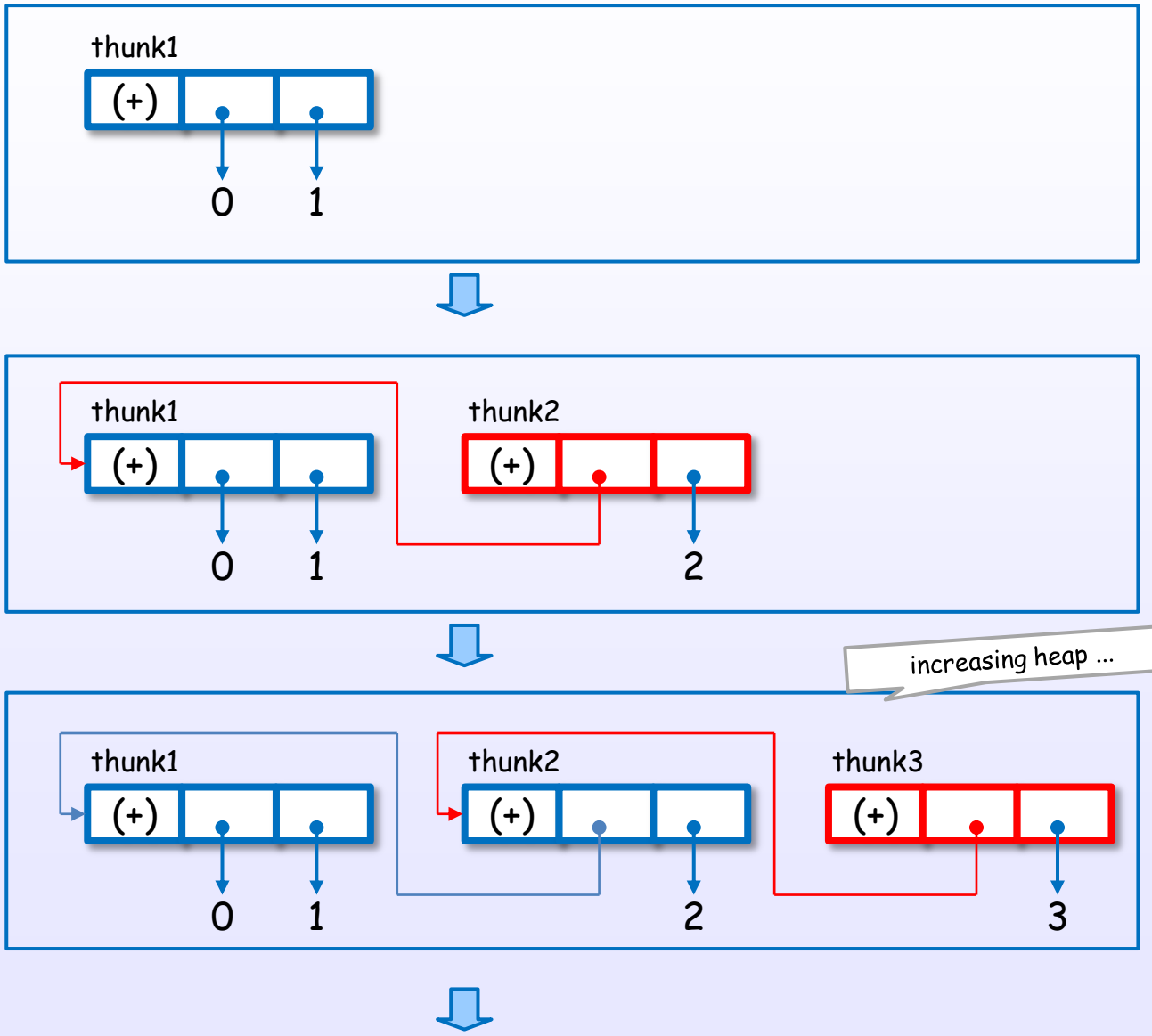
```
let  thunk2 = (thunk1 + 2)
in foldl (+) thunk2 [3 .. 100]
```

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

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

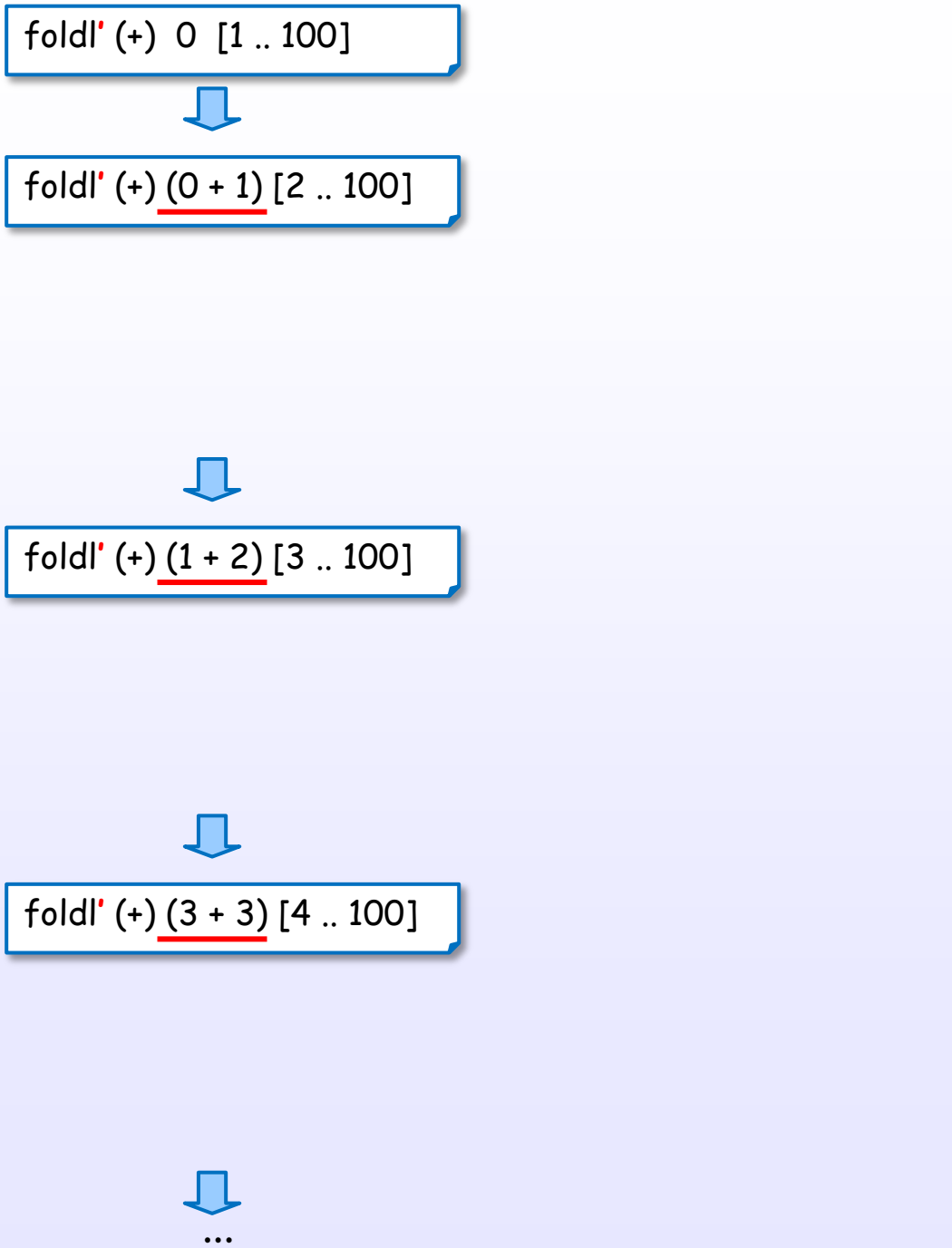
...

heap memory



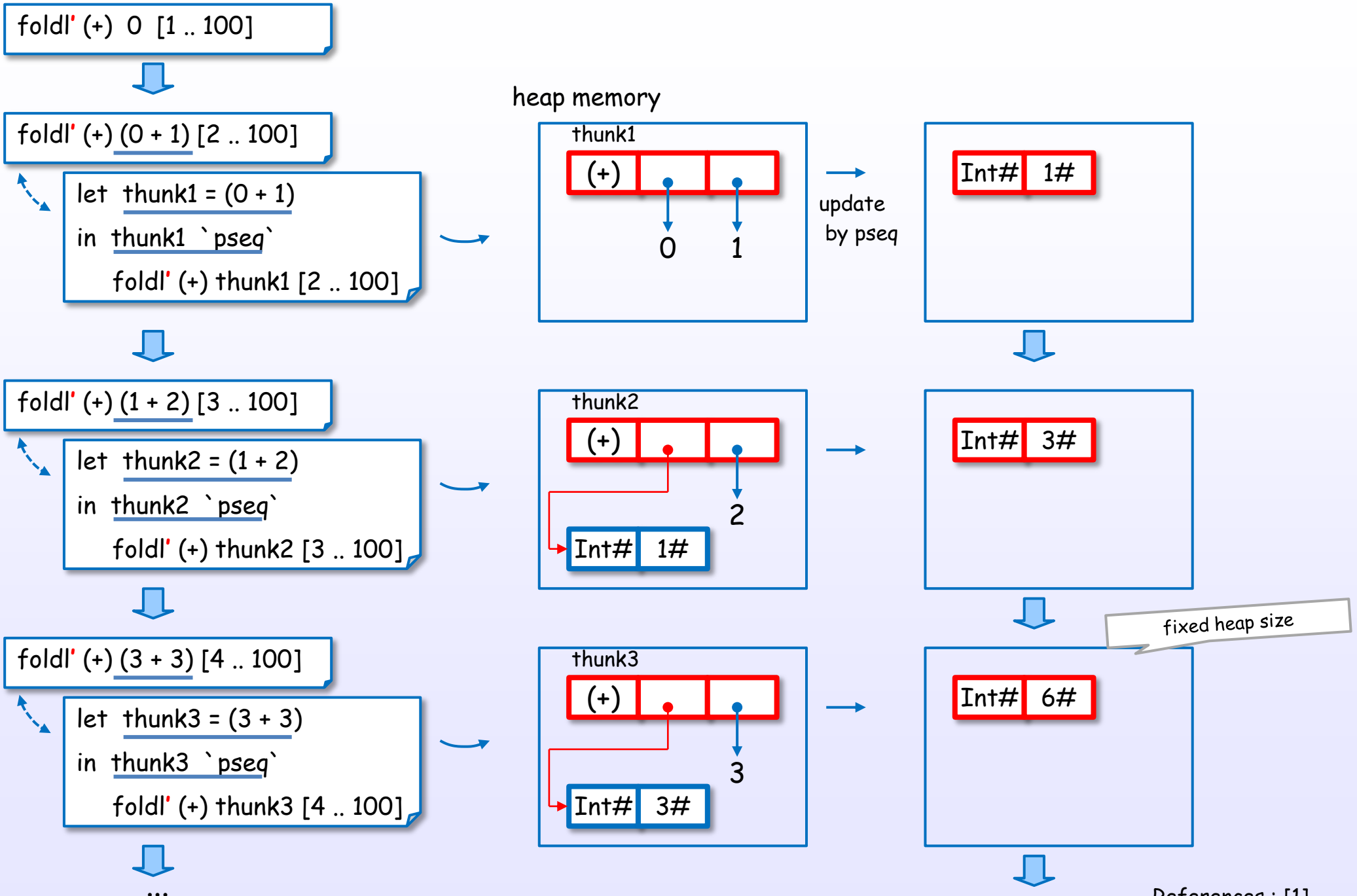
References : [1]

Example of foldl' (strict)



References : [1]

Example of foldl' (strict)



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]

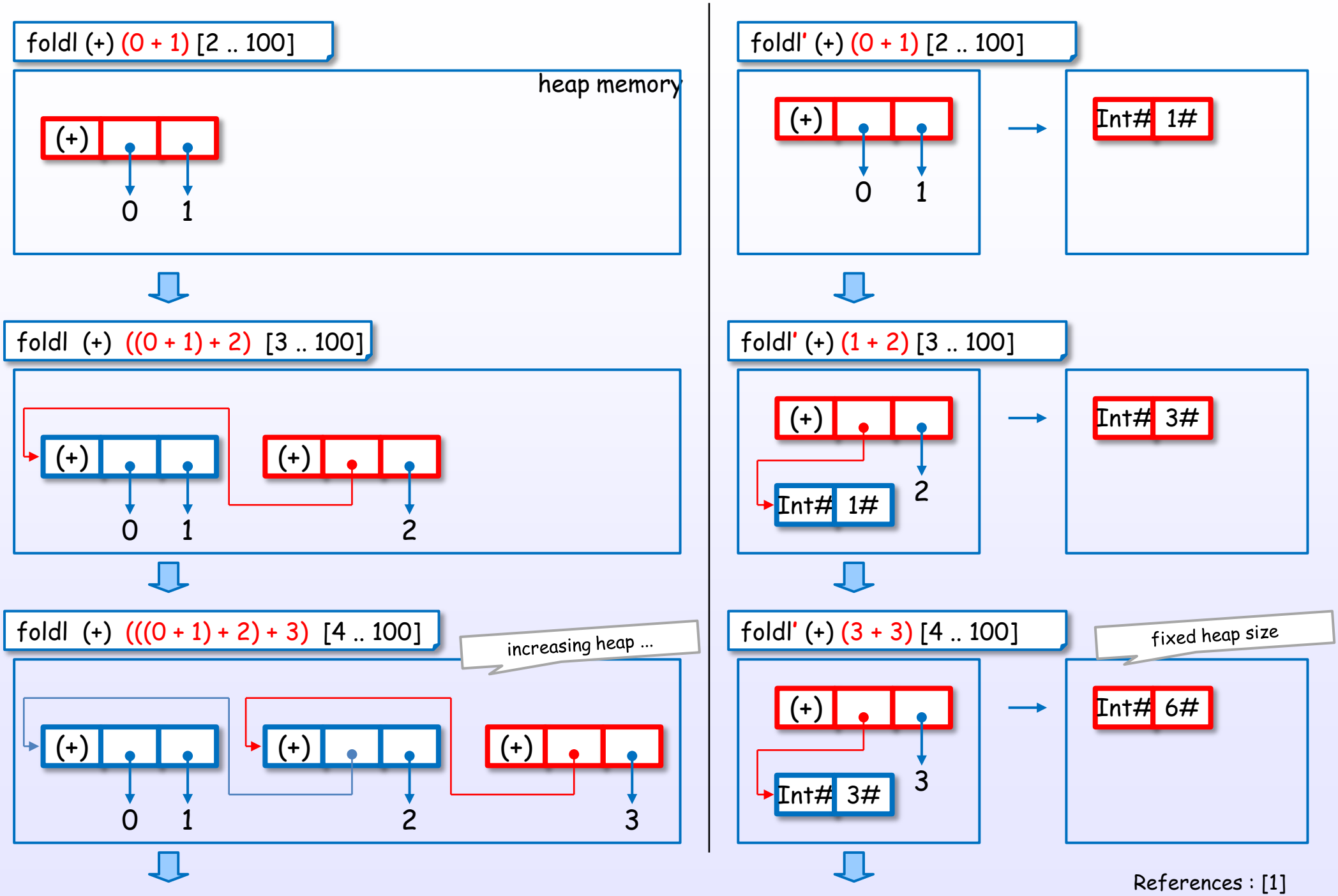
foldl' (+) (0 + 1) [2 .. 100]

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

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

References : [1]

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



References : [1]

Implementation in GHC

Tree, Graph

a expression

AST

Tree

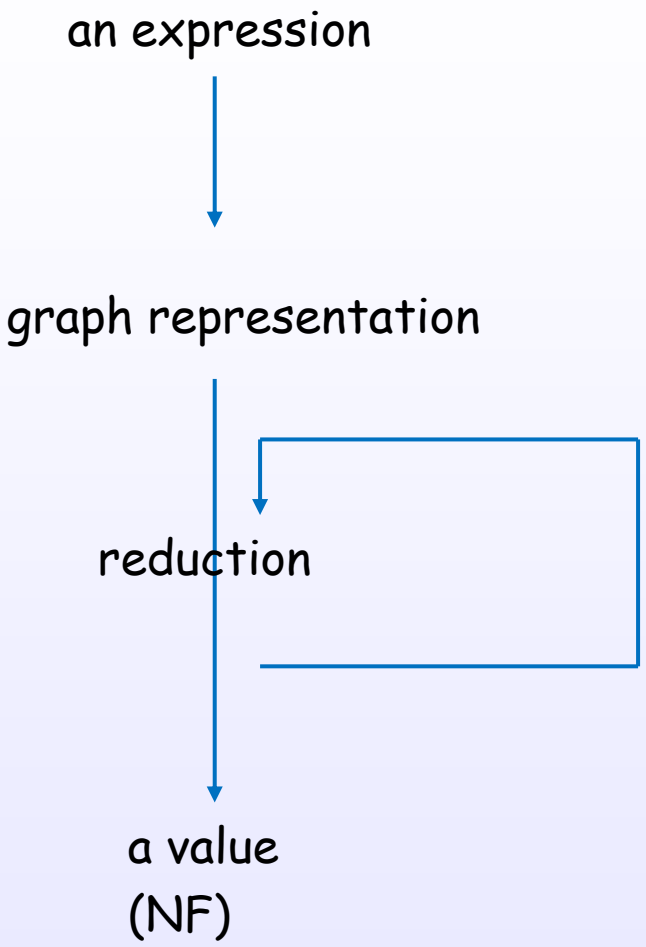
Graph

Shared Term

Lazy

References : [1]

evaluation, reduction



References : [1]

Expressions examples

References : [1]

STG heap objects

language	Just 5
implementation	heap object

References : [1]

Layer

Non-strictness $f \perp = \perp$

Lazy evaluation

Graph reduction

STG machine

References : [1]

Layer

Haskell semantics	take 5 [1..10]
internal representation	graph
STG semantics	heap object
STG machine	

References : [1]

Semantics

Bottom

domain

co-domain

defined

undefined

$f \perp = \perp$

[Bird, Chapter 2]

References : [1]

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)

[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,... zzz