Lazy evaluation illustrated

for Haskellers

exploring some mental models and implementations

Takenobu T.

Rev. 0.00.0
WIP

Lazy,... zzz

..., It's fun!

NOTE

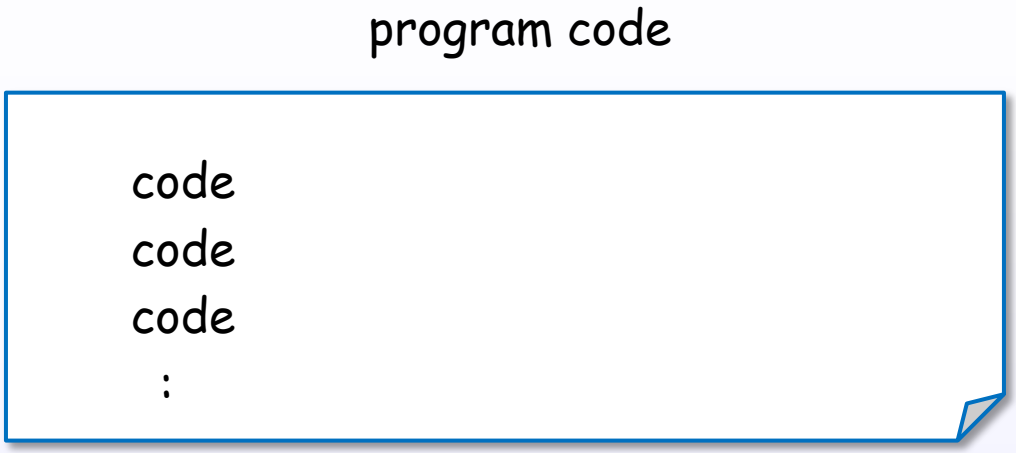
- Meaning of terms are different by communities.
- There are a lot of good documents. Please see also references.
- This is written for GHC's Haskell.

Contents

1. Introduction

Basic mental models

How to evaluate program in your brain ?



プログラムは、どの順で評価される？

どういうステップ、どういう順で evaluation (execution, reduction) される？

What are these mental models?

What “mental model” do you have?

References : [1]

One of the mental models for C program

文の並び

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

?

入れ子の構造

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

?

引数の並び

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

?

関数と引数

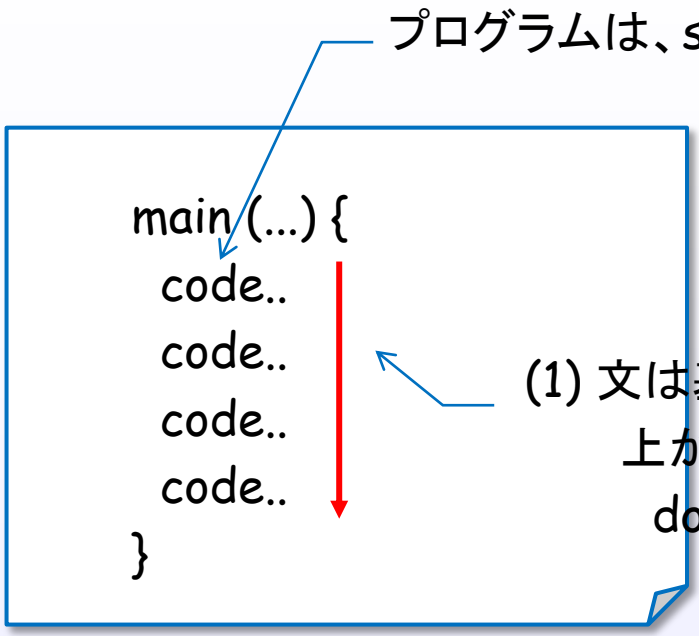
```
z = func1( m + n );
```

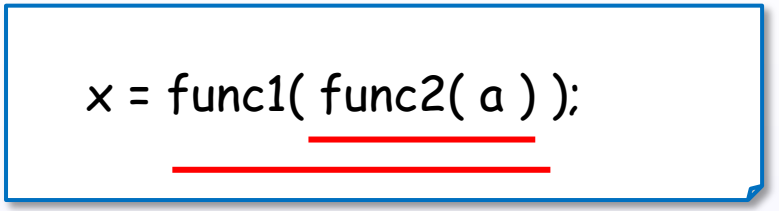
?

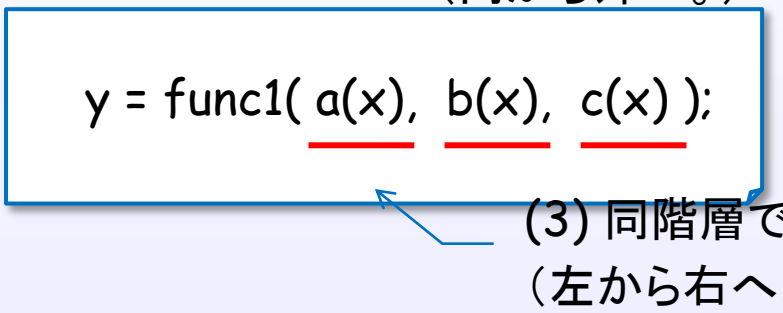
どのように評価される？
あなたの頭の中の、評価メンタルモデルは？

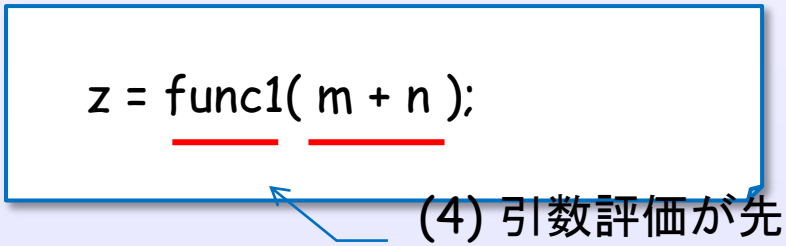
References : [1]

One of the mental models for C program









Each programmers have some mental models in their brain.

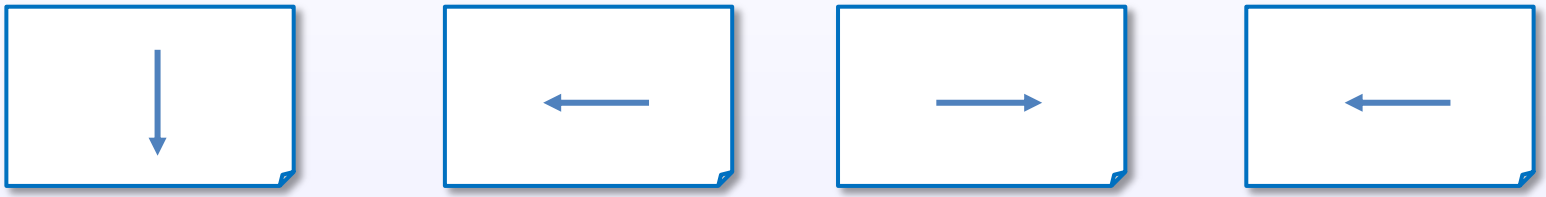
References : [1]

One of the mental models for C program

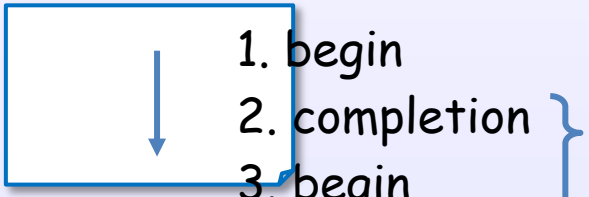
Maybe, You have some implicit mental model in your brain for C program.

(1) program is collection of statements

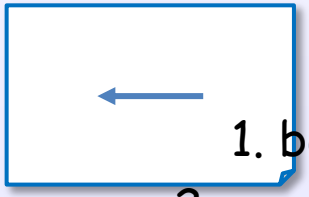
(2) an order between evaluations of elements



(3) an order between completion and begin of evaluations



1. begin
2. completion
3. begin
4. completion



1. begin
2. completion
3. begin
4. completion

This is an example of an implicit sequential order model for programming languages.

References : [1]

One of the mental models for Haskell program

$main = exp_{11}\ (exp_{12}\ exp_{13}\ exp_{14}\ )$

$exp_{13} = exp_{131}\ exp_{132}$

$exp_{14} = exp_{141}\ exp_{142}\ exp_{143}$

:

どのように評価される？
あなたの頭の中の、評価メンタルモデルは？

References : [1]

One of the mental models for Haskell program

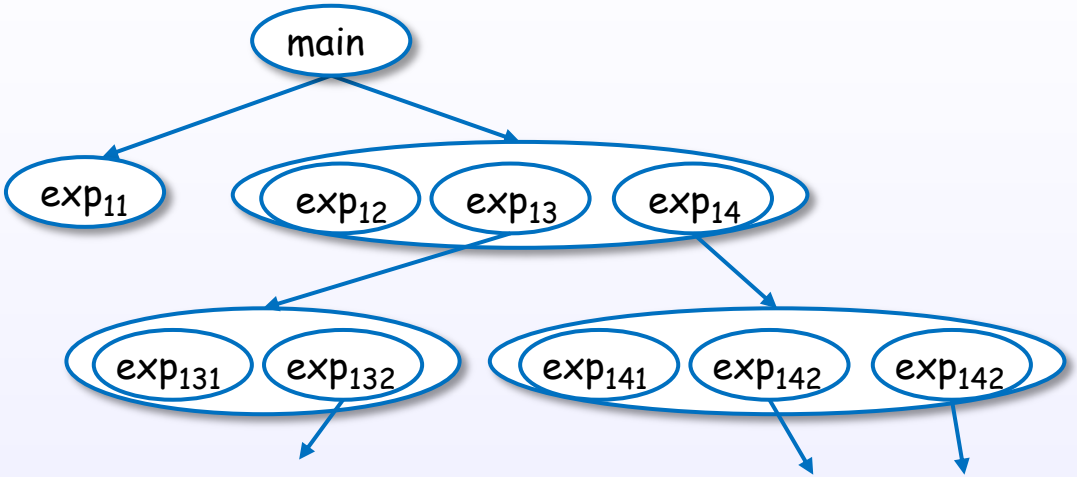
プログラムは、式の集まり

$main = exp_{11}\ (exp_{12}\ exp_{13}\ exp_{14})$

$exp_{13} = exp_{131}\ exp_{132}$

$exp_{14} = exp_{141}\ exp_{142}\ exp_{143}$

:



$main = exp_{11}\ (exp_{12}\ (exp_{131}\ exp_{132})\ (exp_{141}\ exp_{142}\ exp_{143}))$

(1) プログラム全体を1つの式と見立てて

(2) 部分式をある順で評価（簡約）していく

(3) 評価は置換により行う

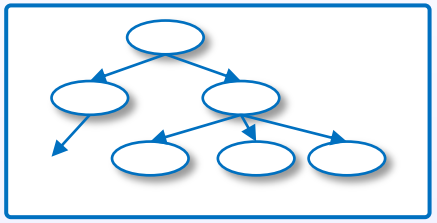
References : [1]

One of the mental models for Haskell program

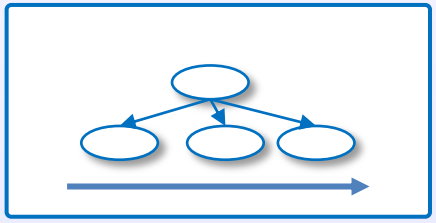
(1) program is collection of expression's declaration

(2) プログラム全体が階層をもった1つの式

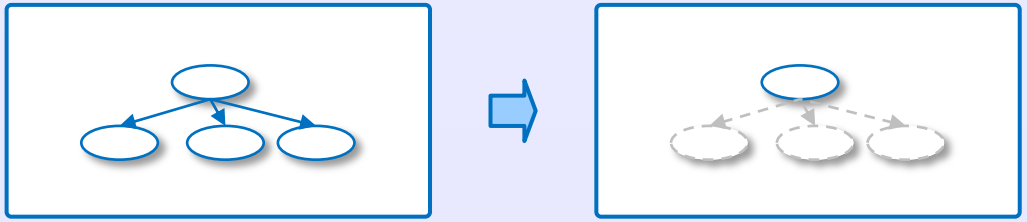
```
main = e (e (e (e e) e (e e e) ) )
```



(3) 部分式を、ある順序で評価していく



(4) 評価は置換により行われる

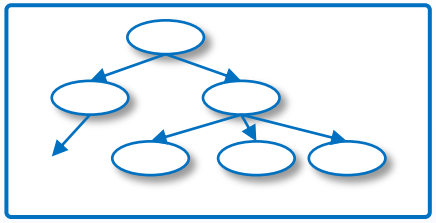


References : [1]

Lazy evaluation

では、具体的にはどうやって評価される？

```
main = e (e (e (e e) e (e e e) ) )
```



Haskellは purely functional language

↓ no side effect [slpj-book-1987], p.193

order free (so, potentially hi-level optimization and parallelism

↓ call-by-need 可能

GHC chosen lazy evaluation to implement non-strict semantics.

[slpj-book-1987], p.33

References : [1]

GHC chosen lazy evaluation

必要な時に、必要な箇所のみを評価する

(STG p.11)

・引数評価を先送る （case式が来るまで評価しない） call-by-need

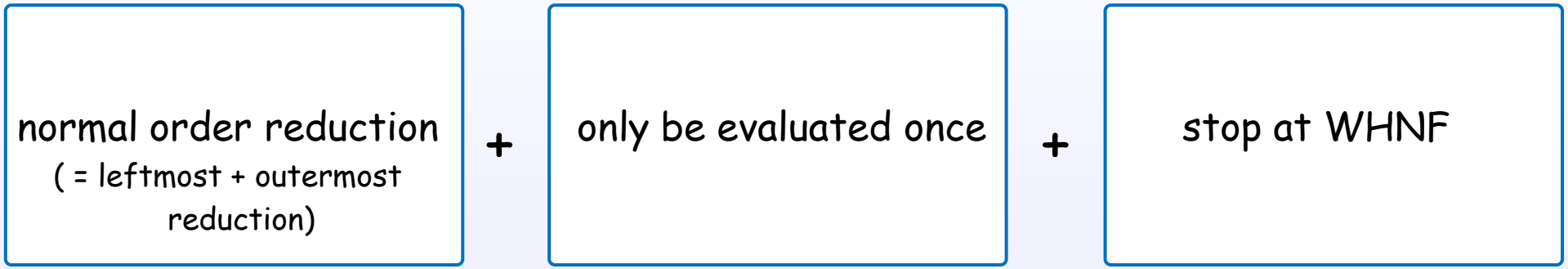
・部分式を完全評価しない （caseのパターンマッチで参照するところのみを評価する）WHNF

これは、計算量を最小化する戦略（メモリ量でなく）

References : [1]

Haskell(GHC) 's lazy evaluation

ingredient of Haskell's "lazy evaluation"



[slpj-book-1987], 194

outermost と、call-by-need

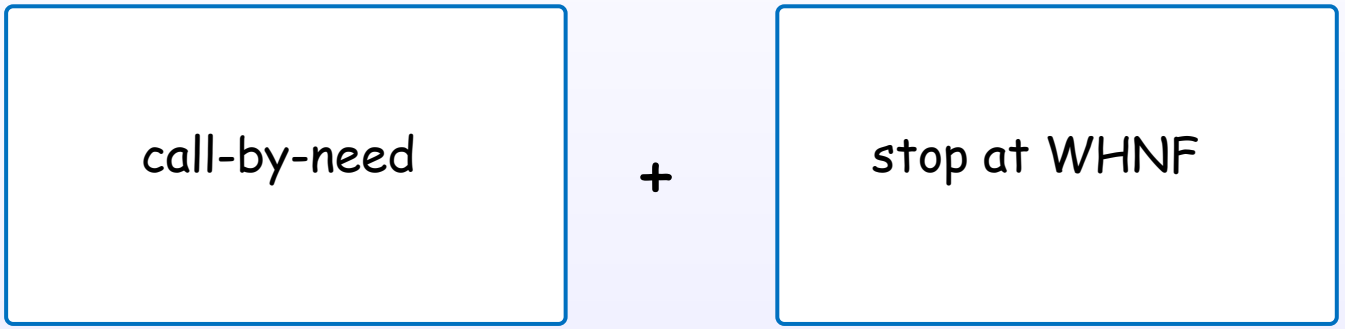
call-by-needは、狭義のlazy eval

[slpj-book-1987], p.198, 23, 194

References : [1]

Haskell(GHC) 's lazy evaluation

ingredient of Haskell's "lazy evaluation"



[slpj-book-1987], 194

outermost と、call-by-need

call-by-needは、狭義のlazy eval

[slpj-book-1987], p.198, 23, 194

References : [1]

Why lazy evaluation?

(1) normal order reduction guarantees to find a normal form (if one exists)

[slpj-book-1987], p.25

pursue normal order reduction, but stop at WHNF.
This is an essential ingredient of lazy evaluation

(2) lazy evaluation implements non-strict semantics

infinite data structure and stream

[slpj-book-1987], p.194

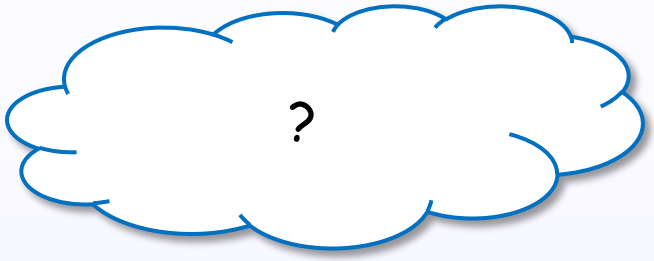
(3) 不要な評価を避ける

References : [1]

Expression and value

What is an expression?

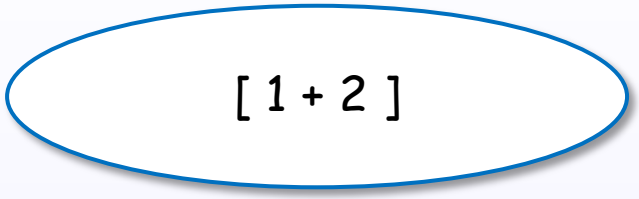
An expression



References : [1]

An expression denotes a value

An expression

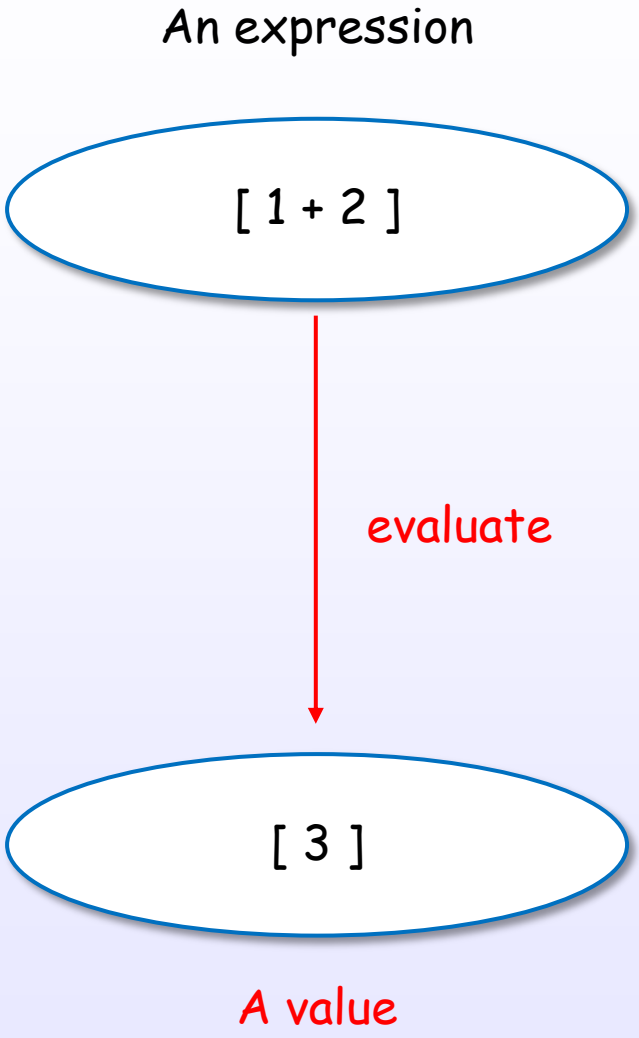


[HR2010]

[Bird, Chapter 2]

References : [1]

An expression evaluates to a value

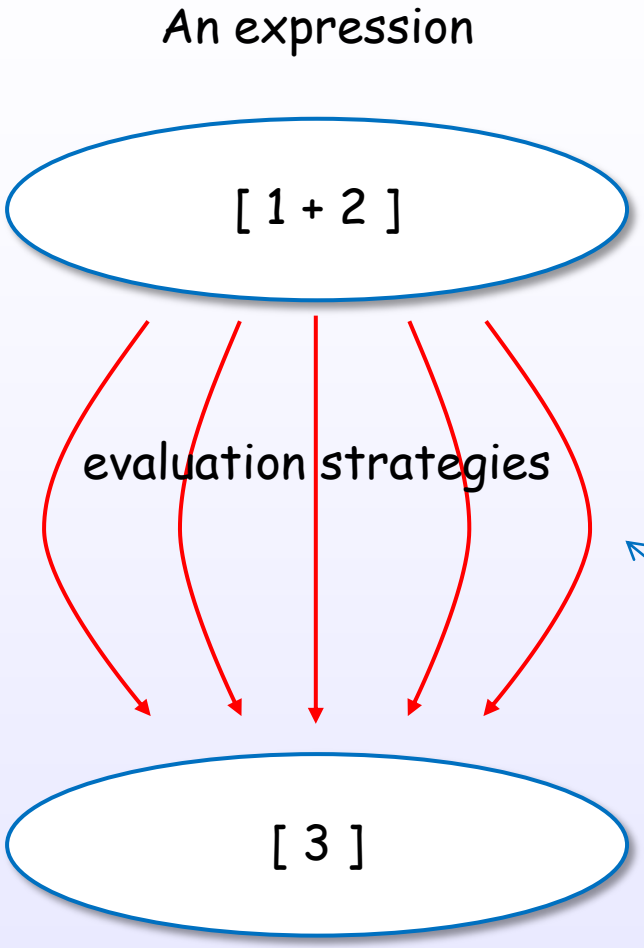


[HR2010]

[Bird, Chapter 2]

References : [1]

There are many evaluation approaches



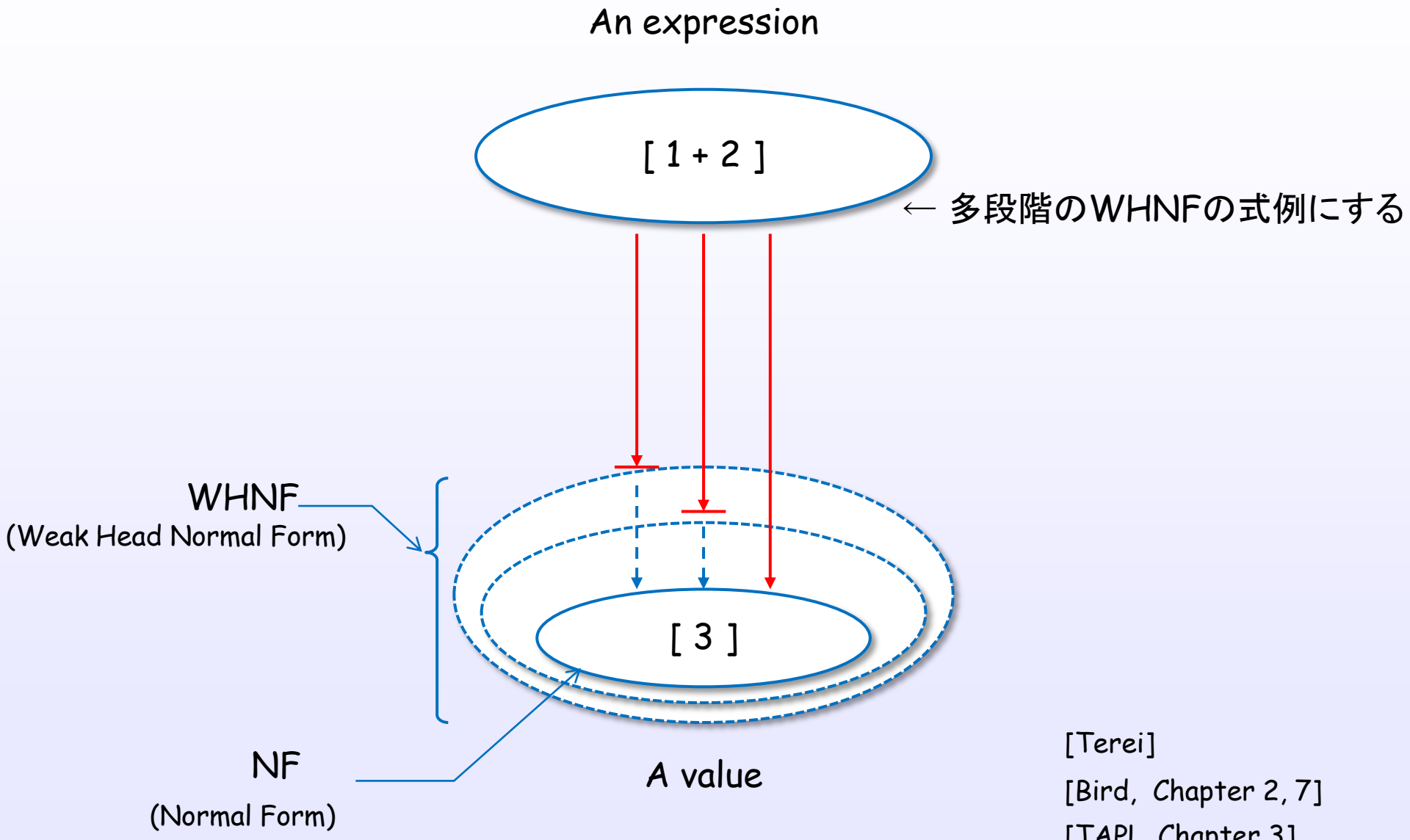
- Strict, Non-strict evaluation
- Eager, Lazy evaluation
- Call-by-value, Call-by-name, Call-by-need, …
- Innermost, Outermost
- Normal order, Application order
- …

[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

References : [1]

There are some evaluation levels



[Terei]

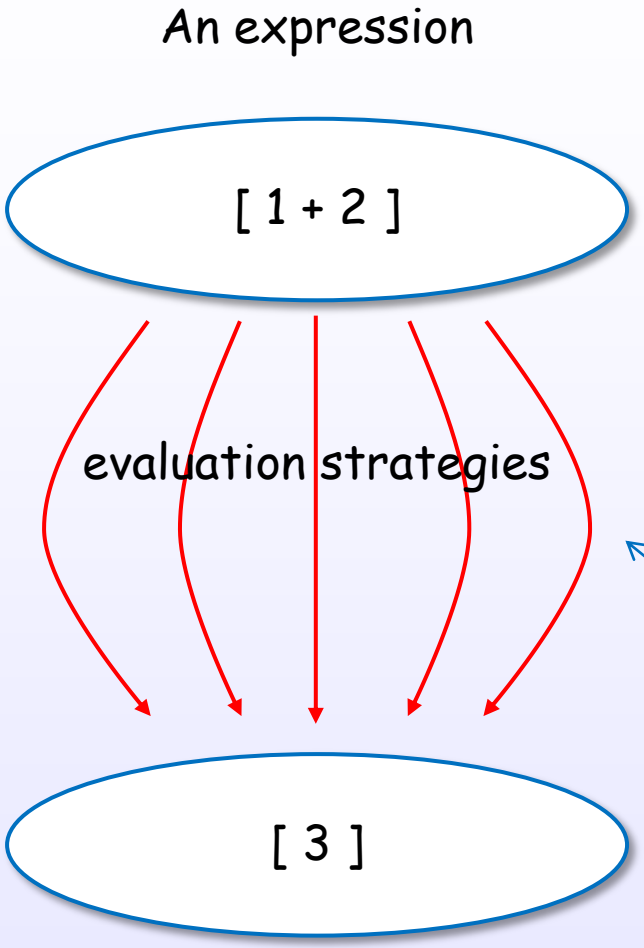
[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

References : [1]

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

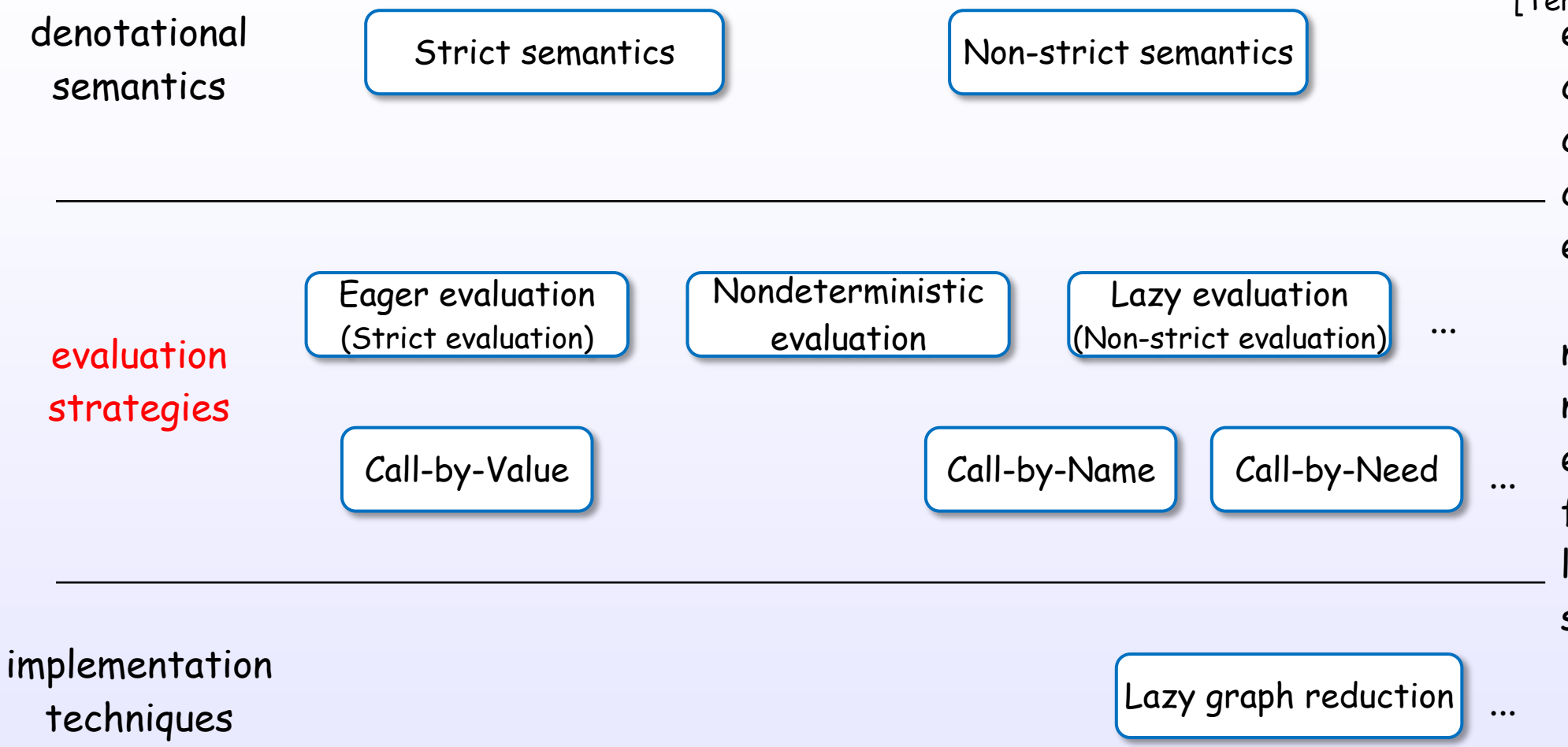
[Bird, Chapter 7]

[Hutton, Chapter 8]

[TAPL, Chapter 3]

References : [1]

Evaluation layers



[Terei]
eva
call
call
call
eva
no-
non
eva
fun
lazy
sha

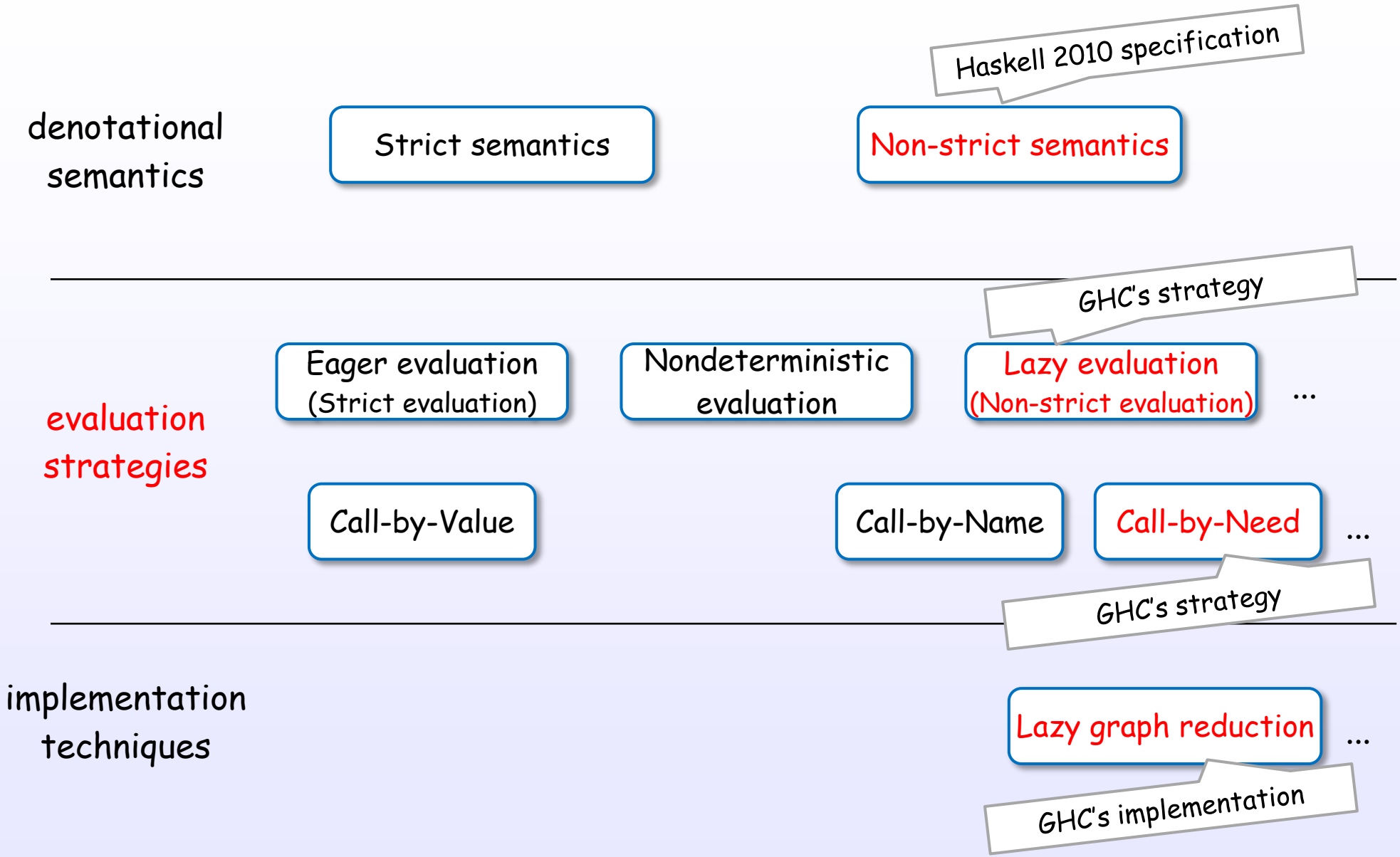
[Bird, Chapter 7]

[Hutton, Chapter 8]

[TAPL, Chapter 3]

References : [1]

Evaluation layers for GHC's Haskell



References : [1]

Evaluation 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]

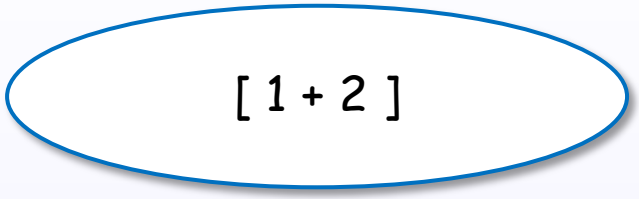
References : [1]

2. Expressions

Expressions in Haskell

An expression denotes a value

An expression

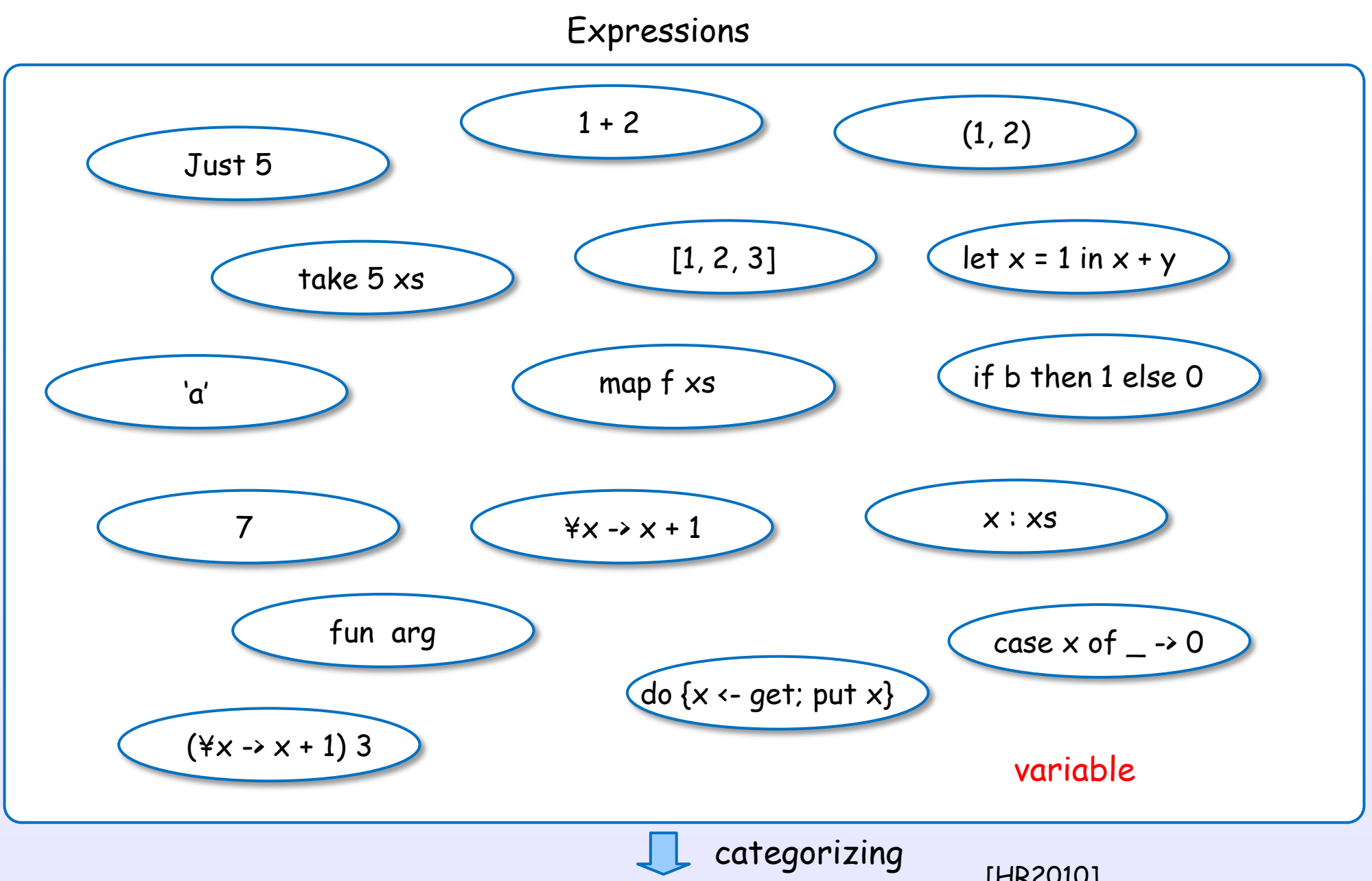


[HR2010]

[Bird, Chapter 2]

References : [1]

There are many expressions in Haskell



[HR2010]

[Bird, Chapter 2]

References : [1]

Expression categories in Haskell

WHNF(a value)、unevaluated expression との関連づけを PAPもWHNFなので注意

lambda abstraction

¥x -> x + 1

let expression

let x = 1 in x + y

variable

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

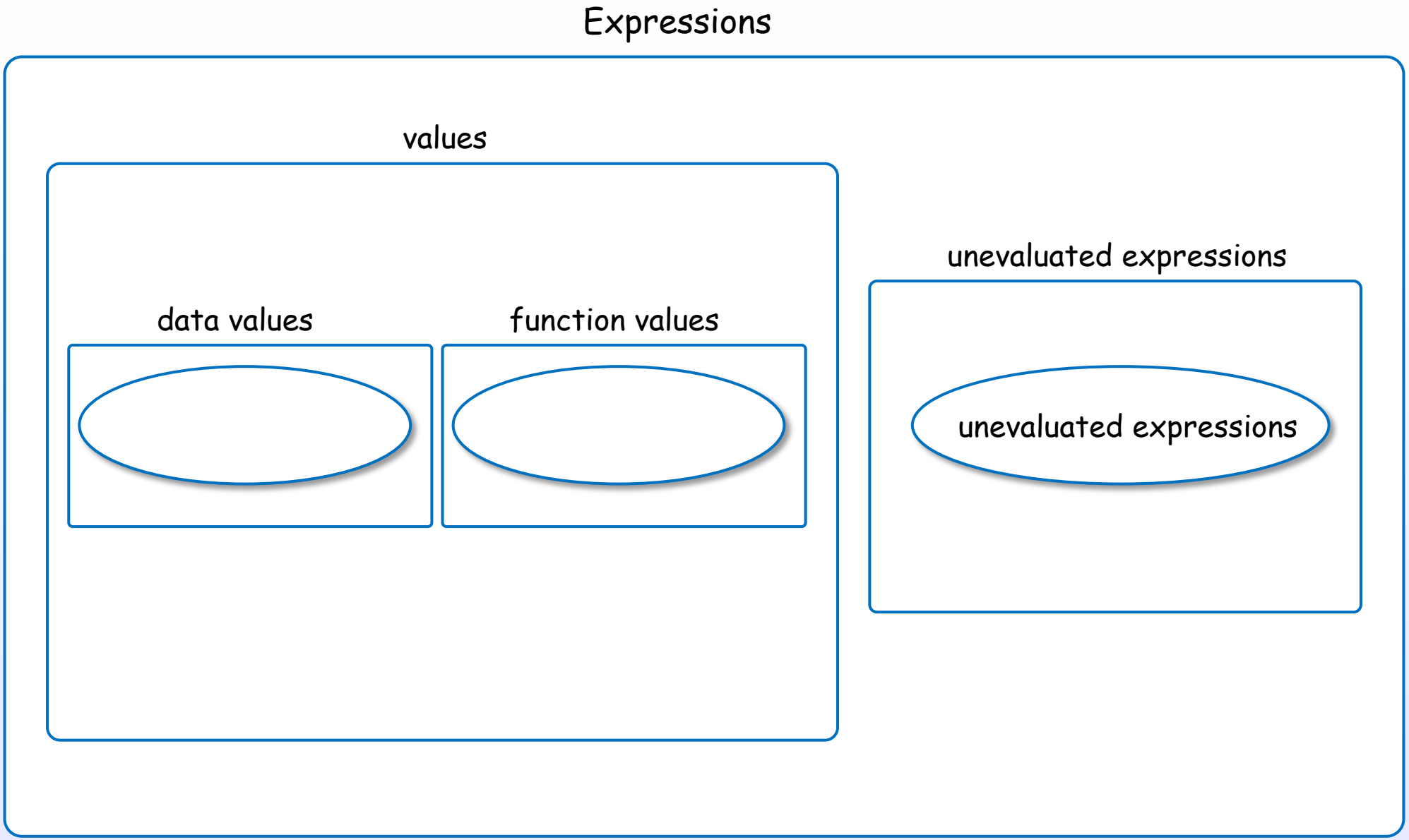
Haskell 2010 Language Report, Chapter 3 Expressions [1]

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]

Classification of expressions

A value or an unevaluated expression

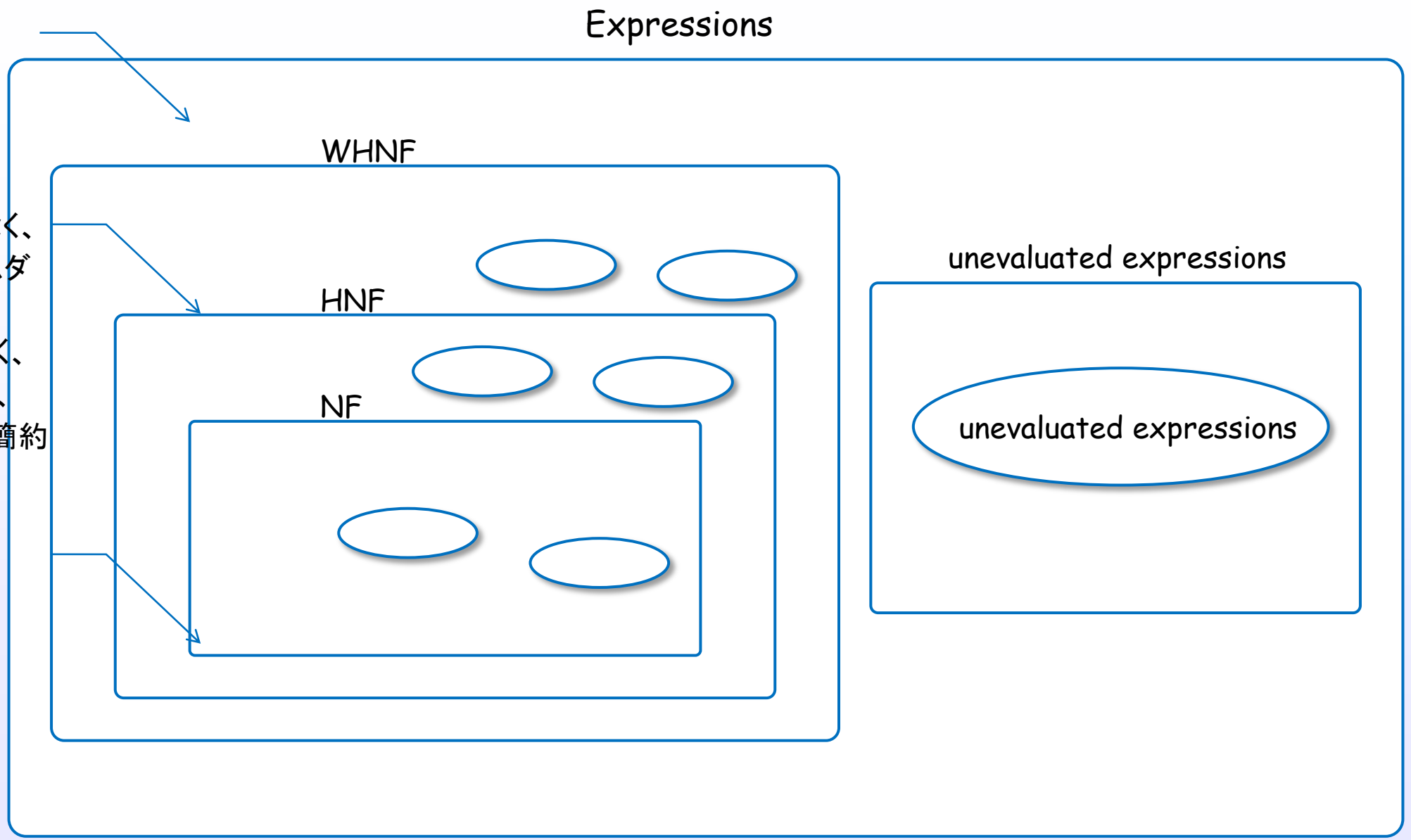


値か否か。値は2種。

[STG]

References : [1]

evaluation level



値には、評価レベルがある。

[STG]

References : [1]

実例との対応付け

[STG]

References : [1]

3. Internal representation of expressions

3. Internal representation of expressions

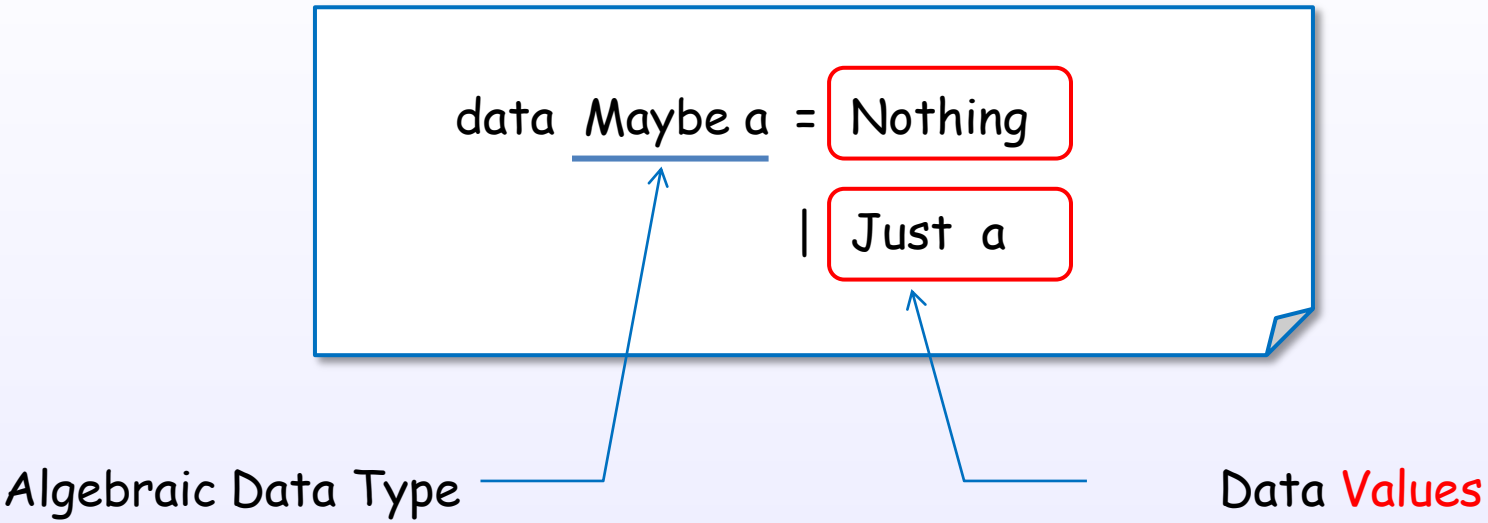
Constructor

Constructor

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

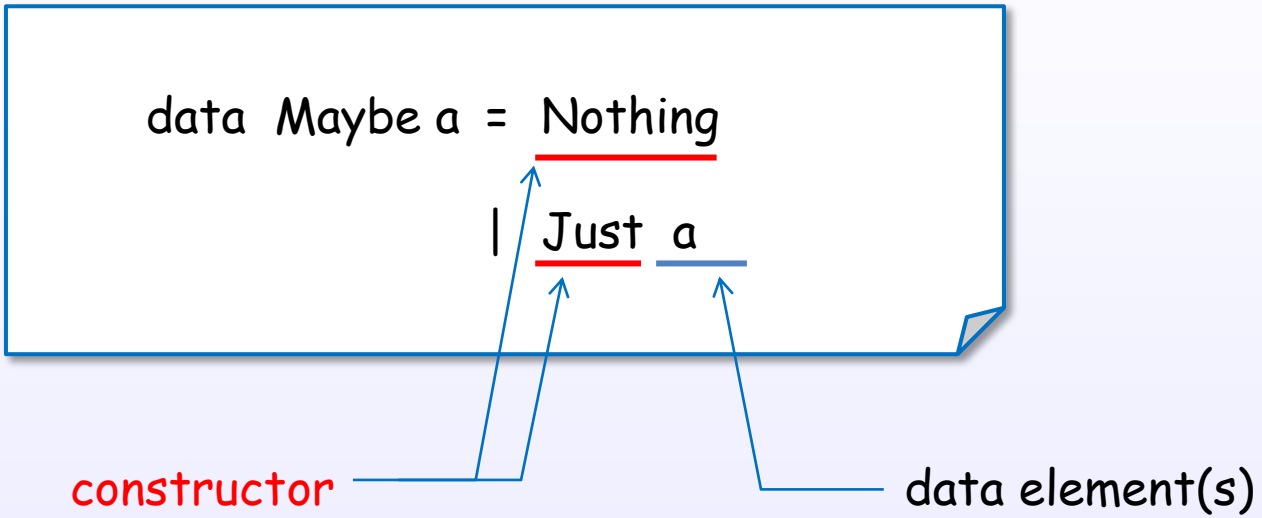
References : [1]

data文で宣言する代数的データ型とその値



References : [1]

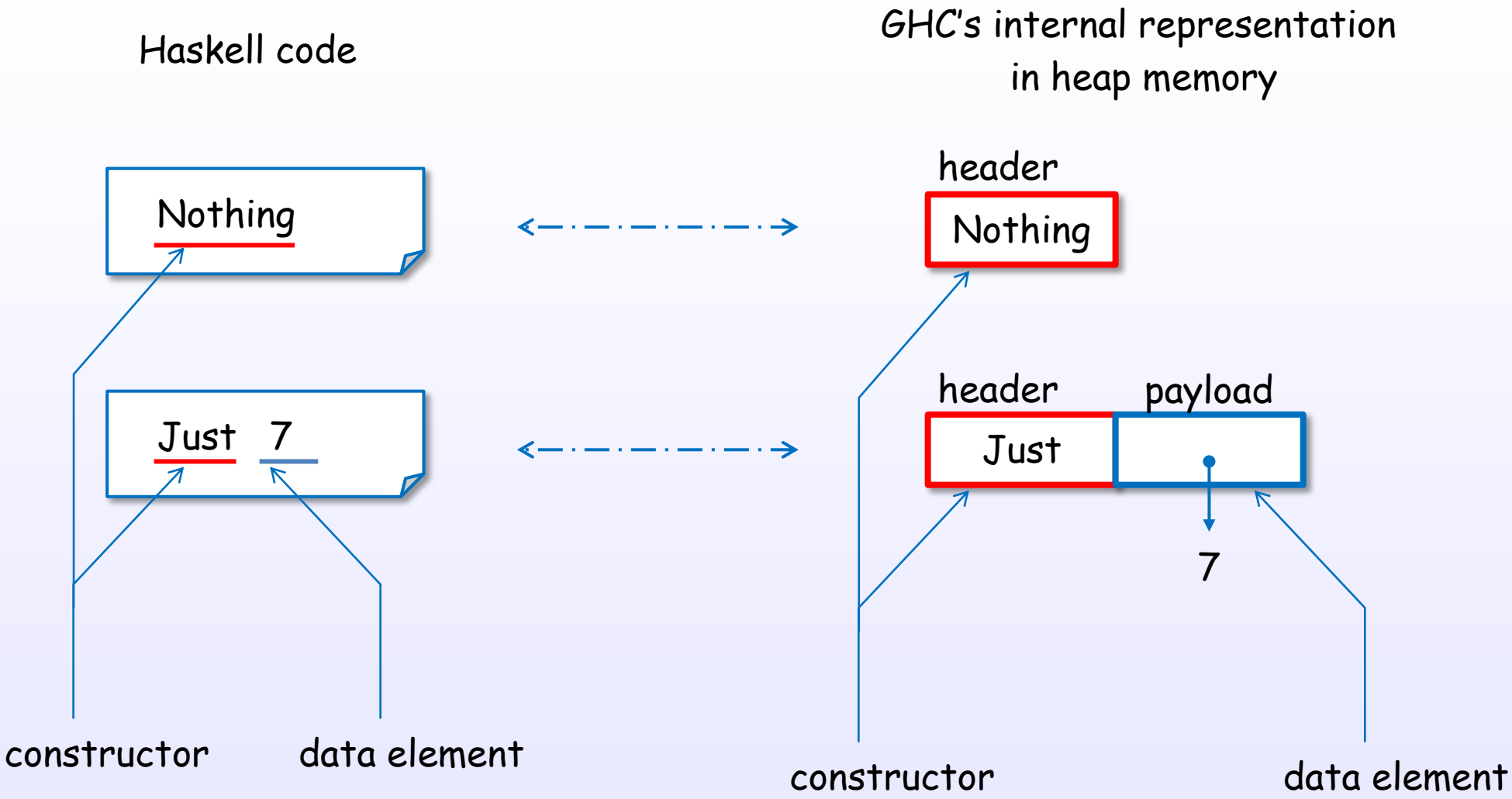
Constructorはdata文で宣言する代数的データ値



References : [1]

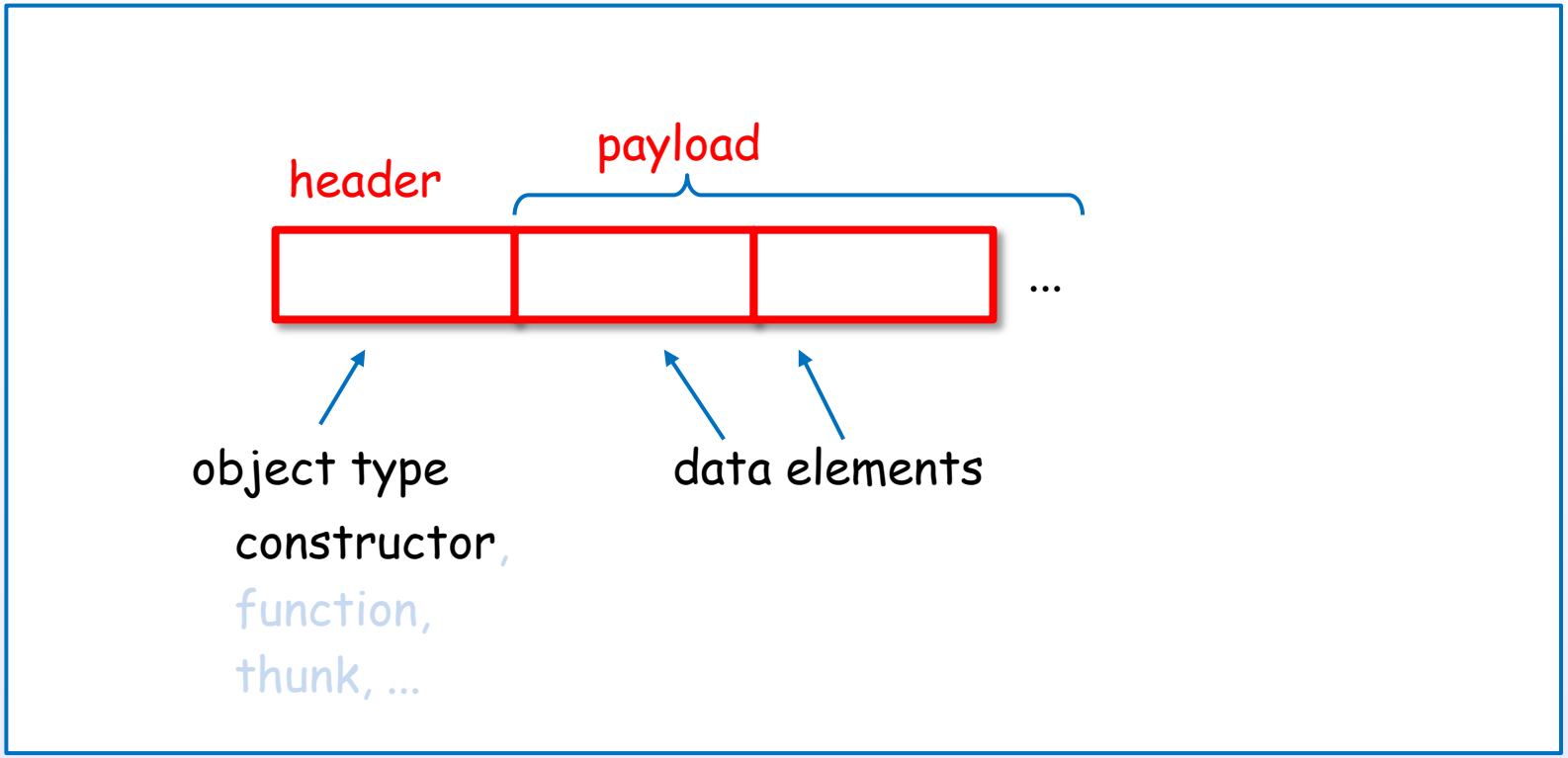
Constructorの内部表現

↑ data values



References : [1]

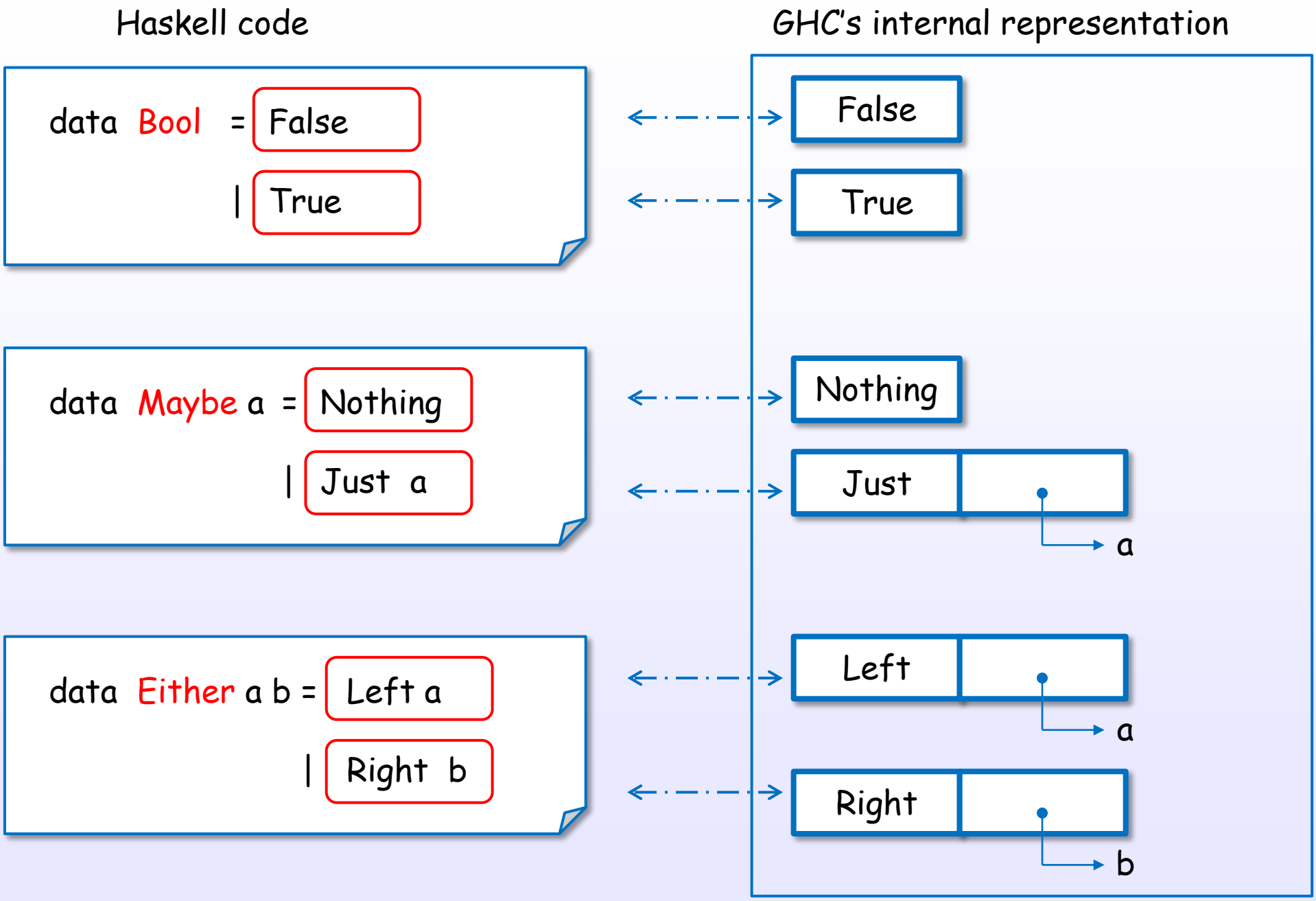
Constructorは統一内部表現で表現される



in heap memory, stack, registers or static memory

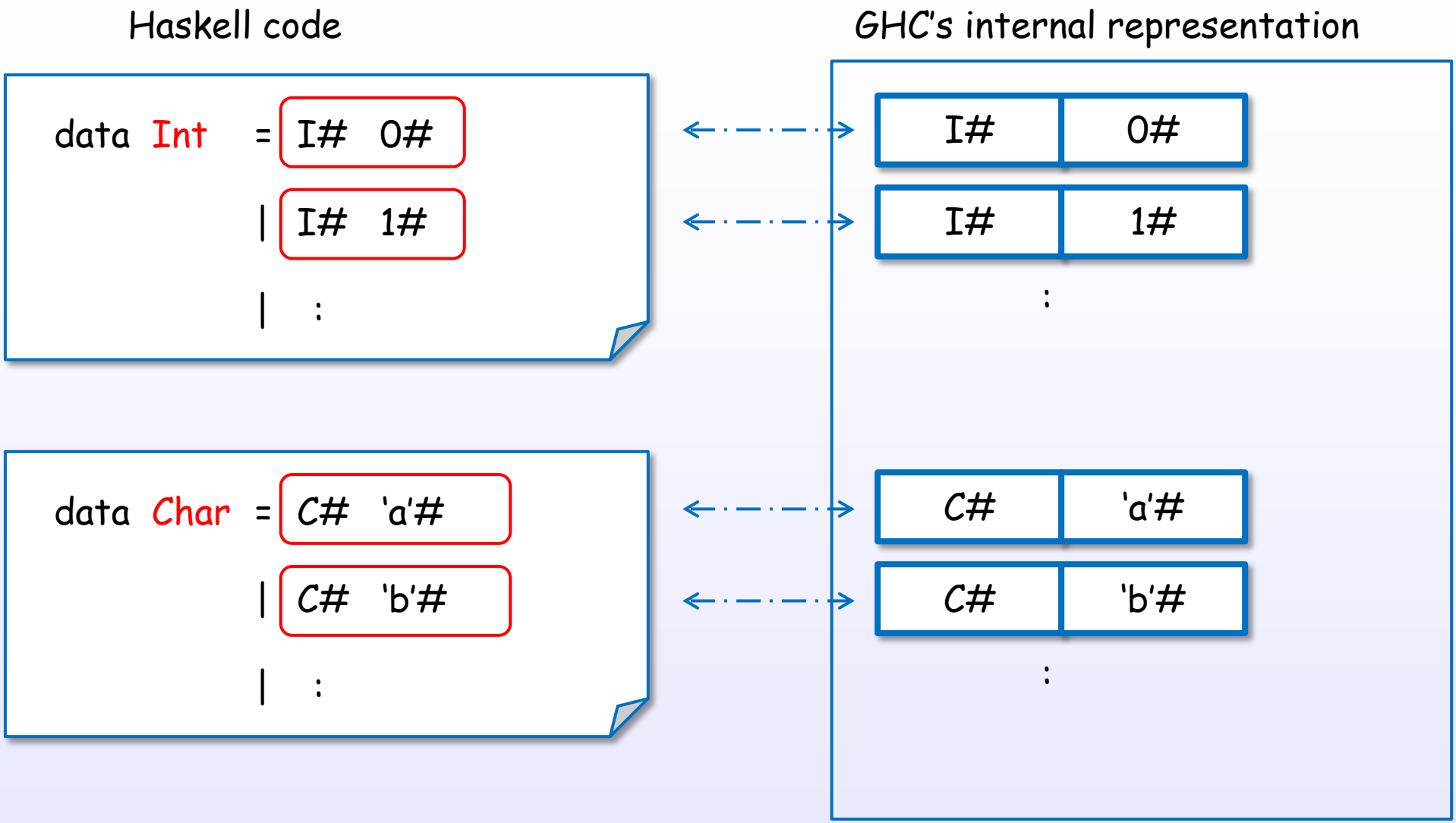
References : [1]

いろいろなコンストラクタと内部表現



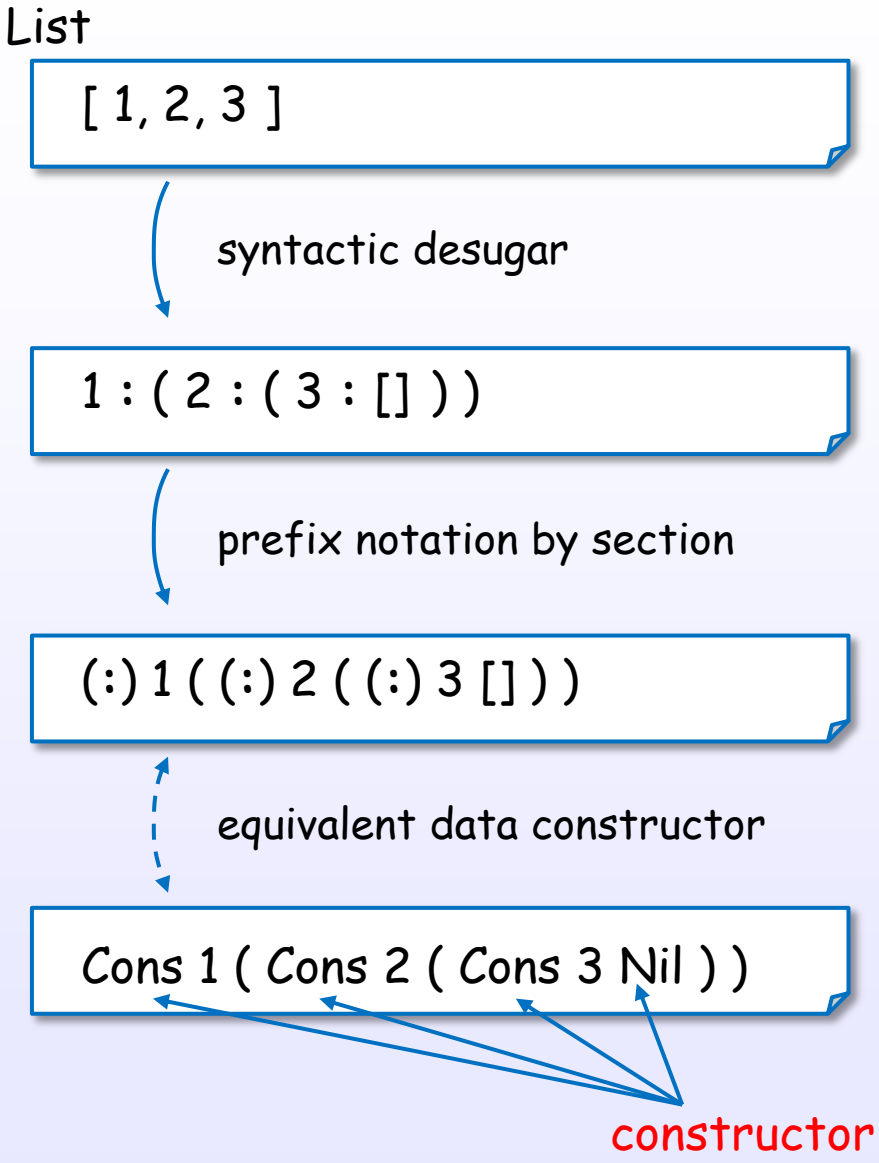
References : [1]

基本データ型も実はコンストラクタで構成されている



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

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))

type declaration

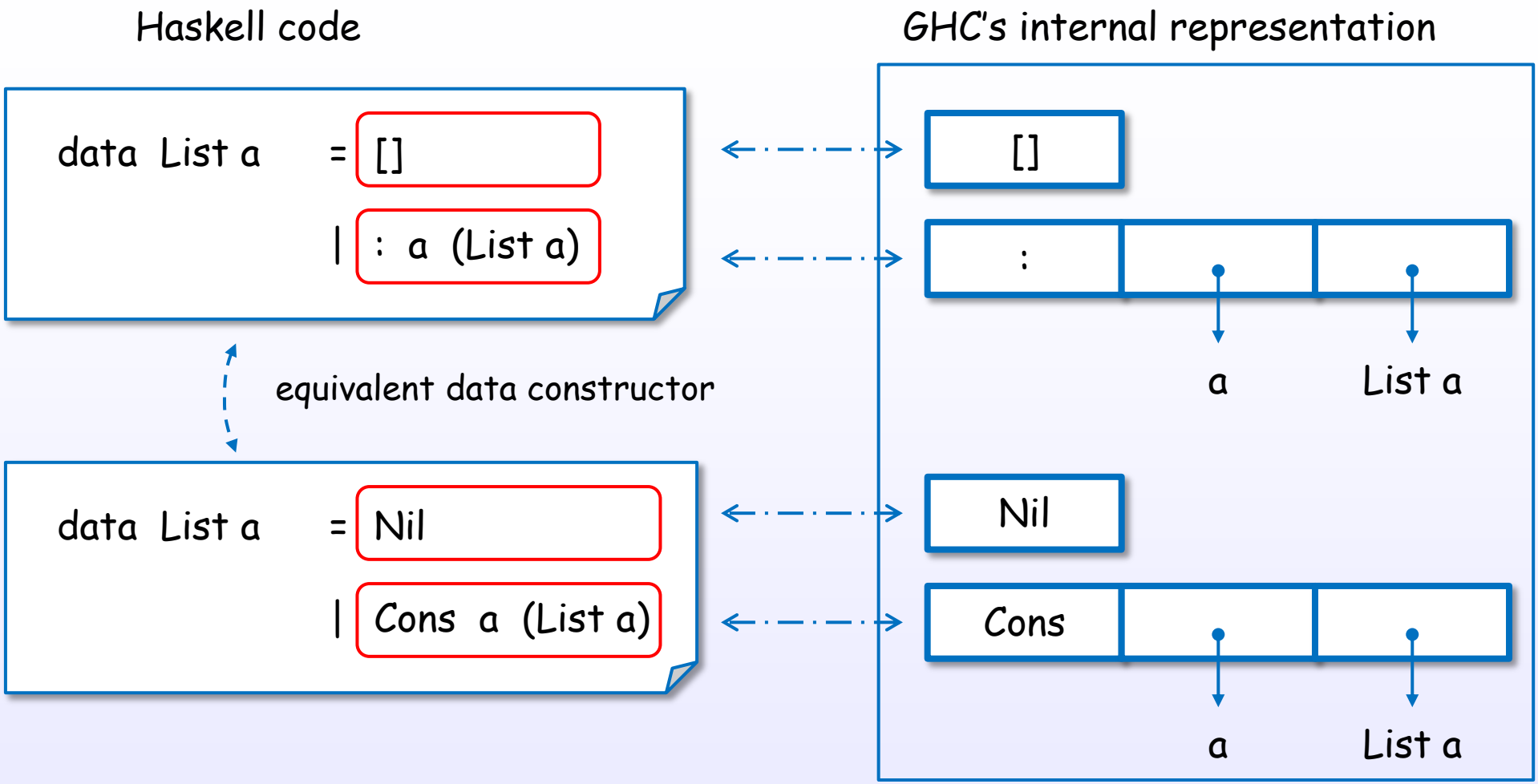
** pseudo code*

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

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

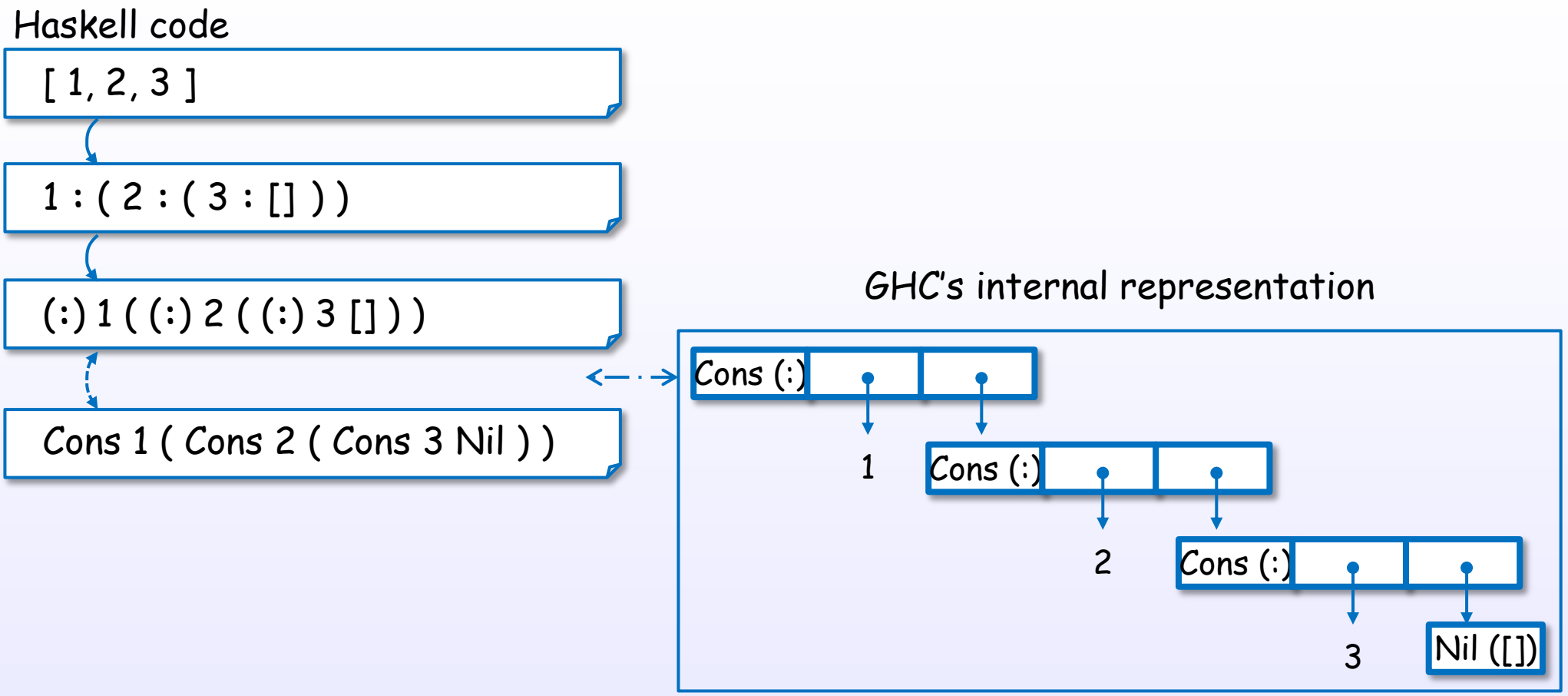
References : [1]

リストも実はコンストラクタで構成されている



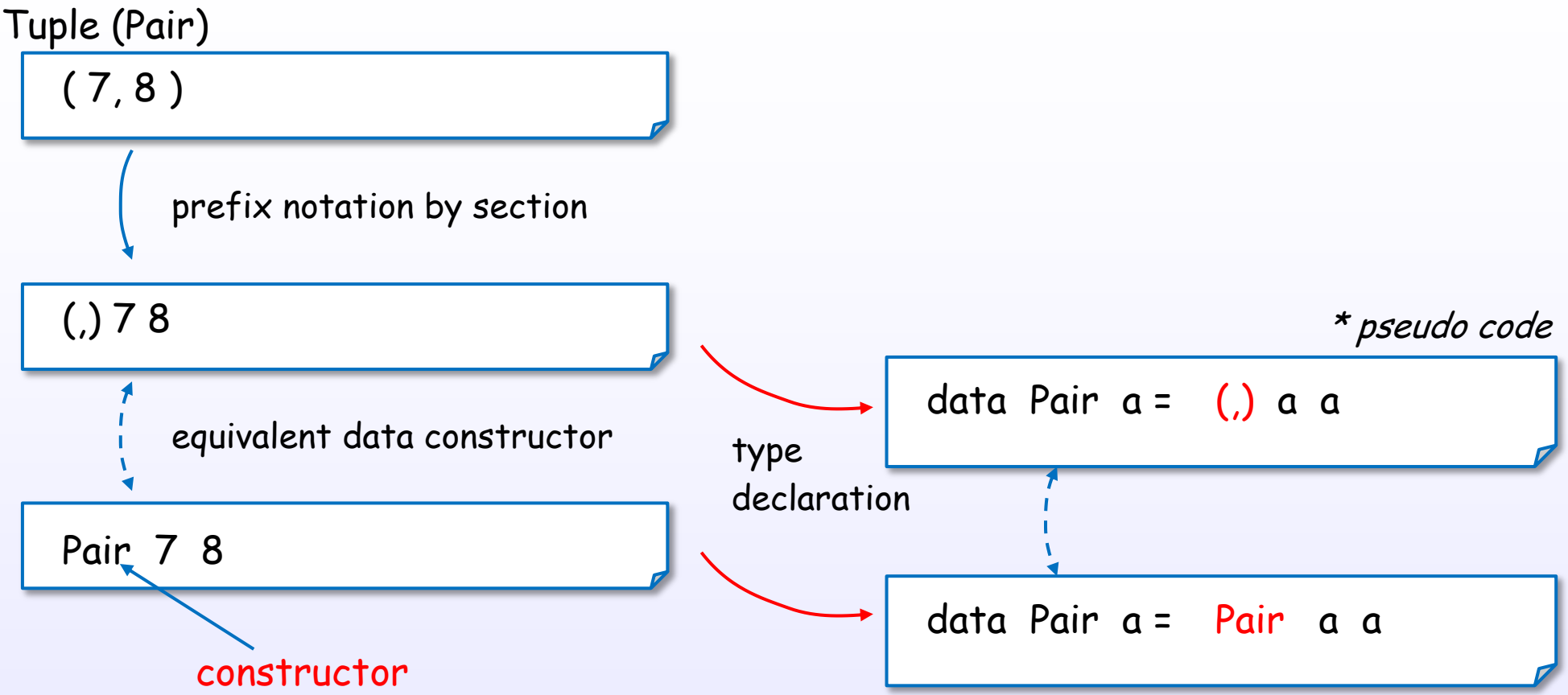
References : [1]

リストも実はコンストラクタで構成されている



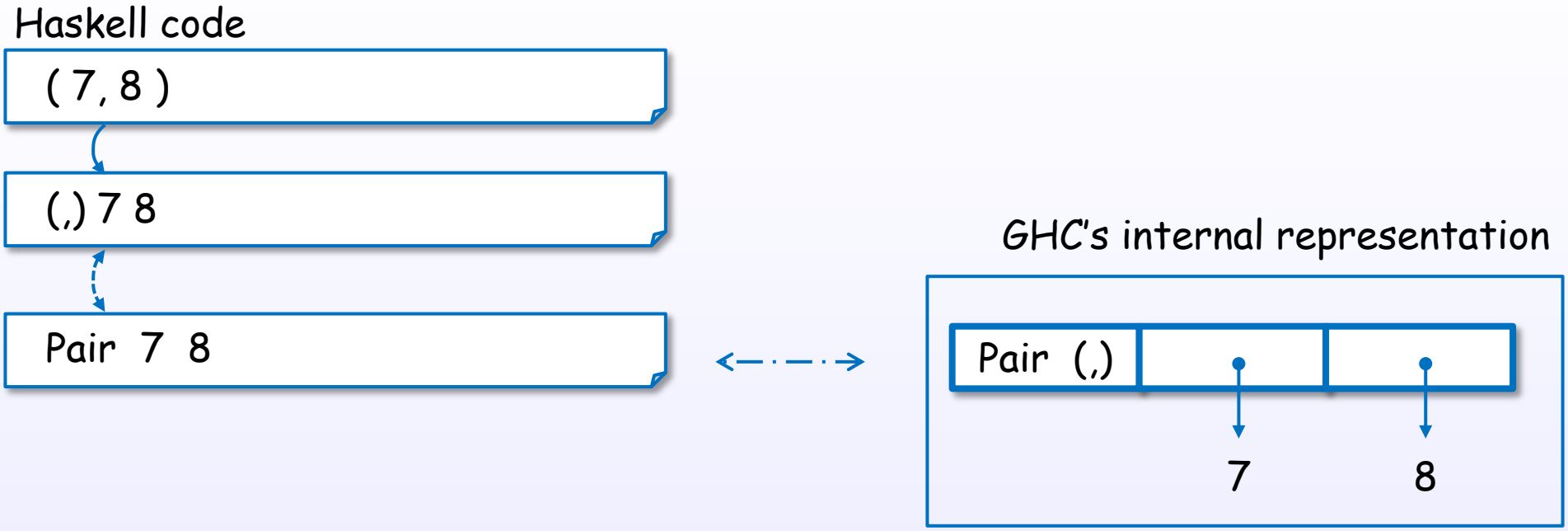
References : [1]

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

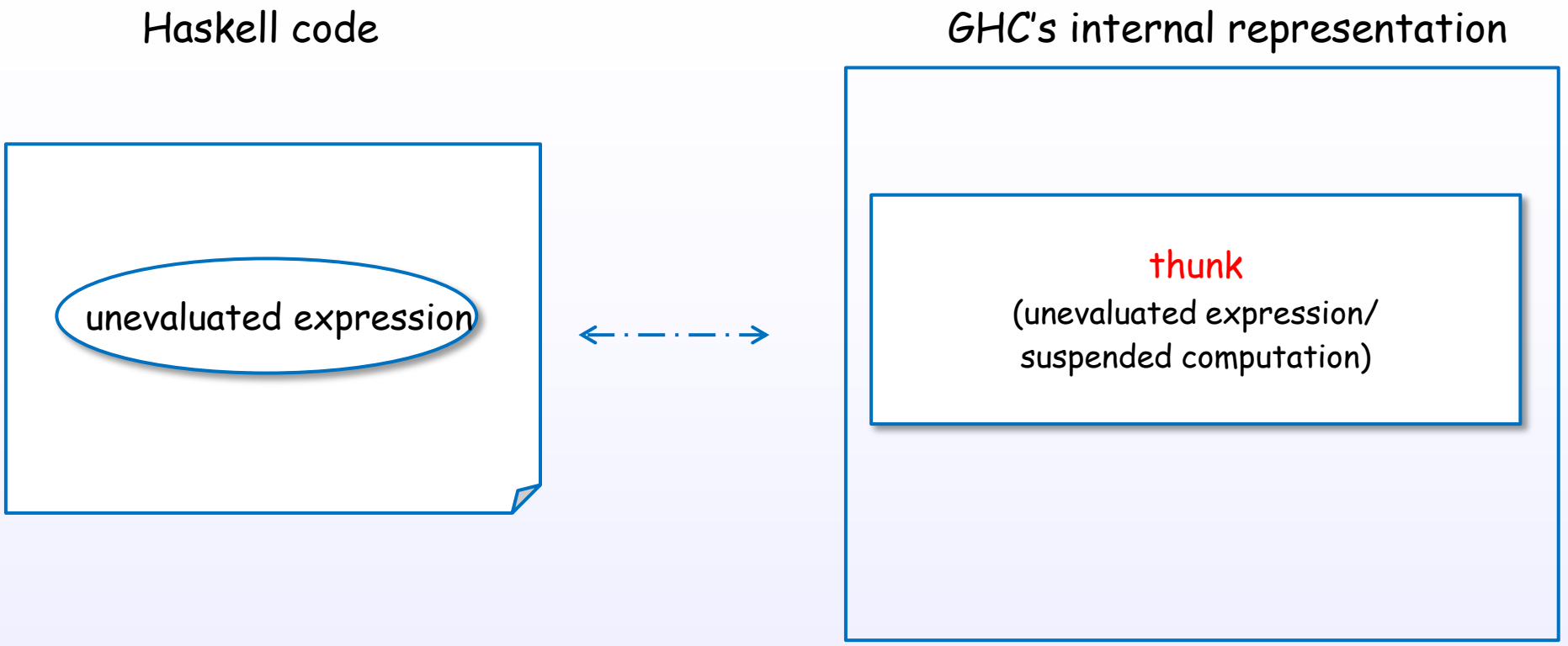


References : [1]

3. Internal representation of expressions

Thunk

Thunk



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

References : [1]

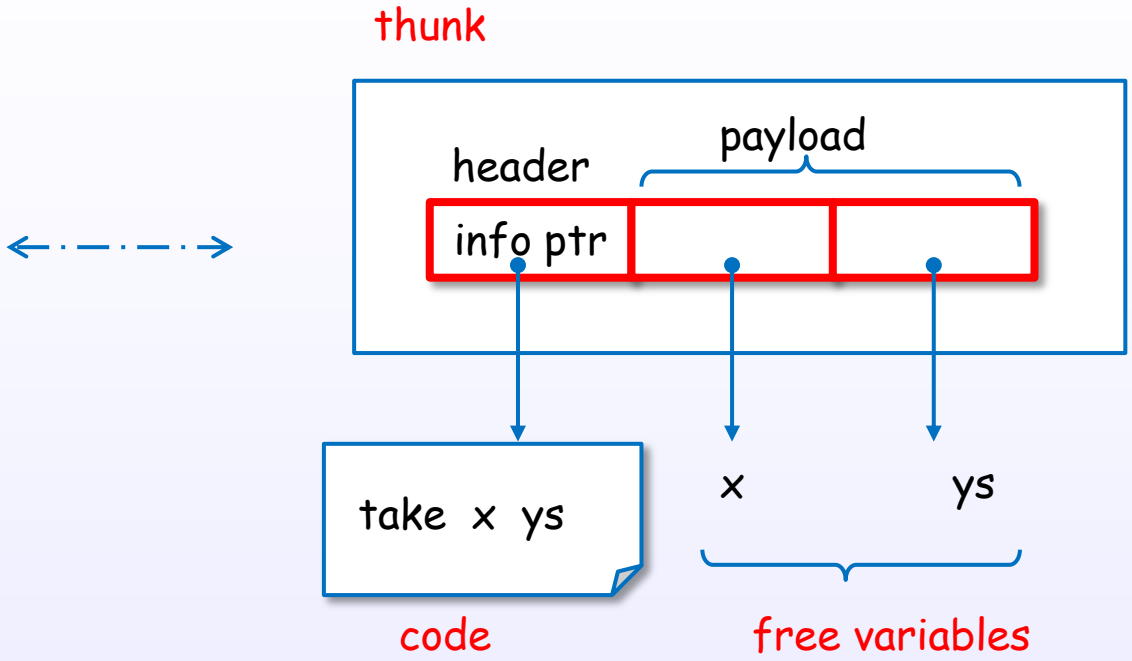
Thunkの内部表現

Haskell code

GHC's internal representation

An unevaluated expression

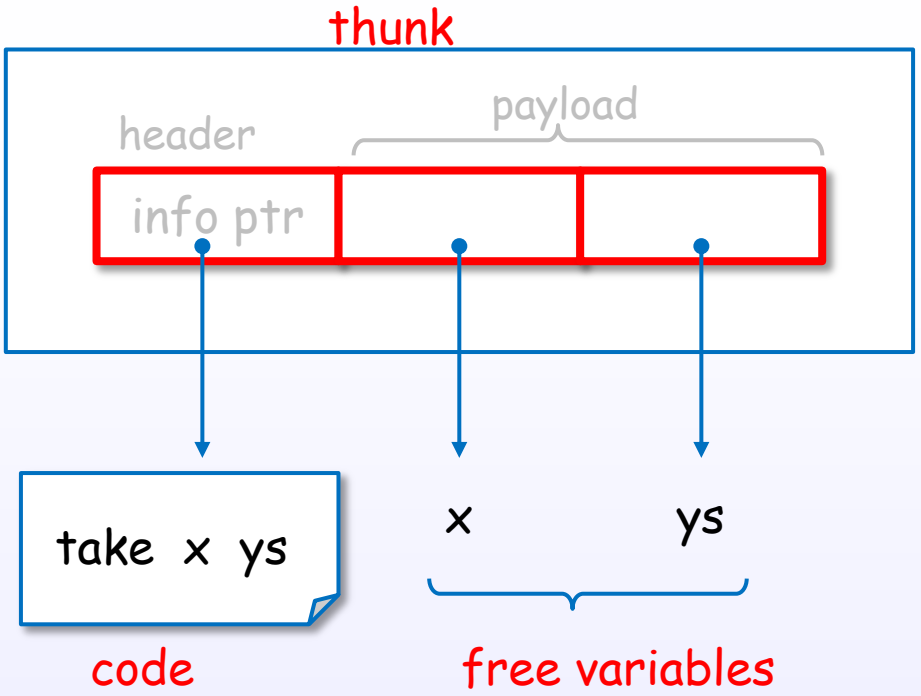
take x ys



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

References : [1]

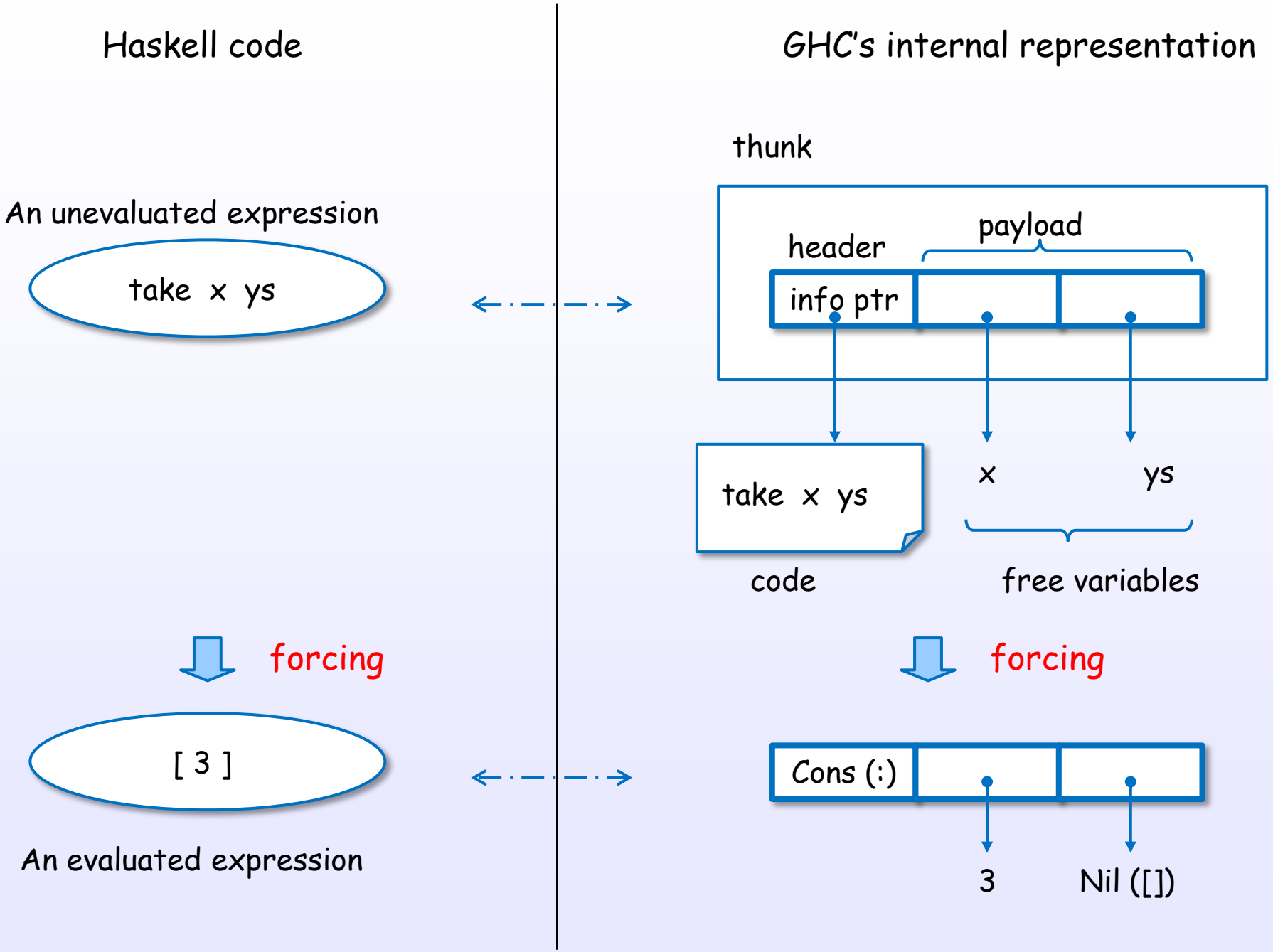
Thunkは、codeとfree variablesをパッケージ化したもの



A thunk is a package of code + free variables.

References : [1]

Thunkは、forcing要求により評価される

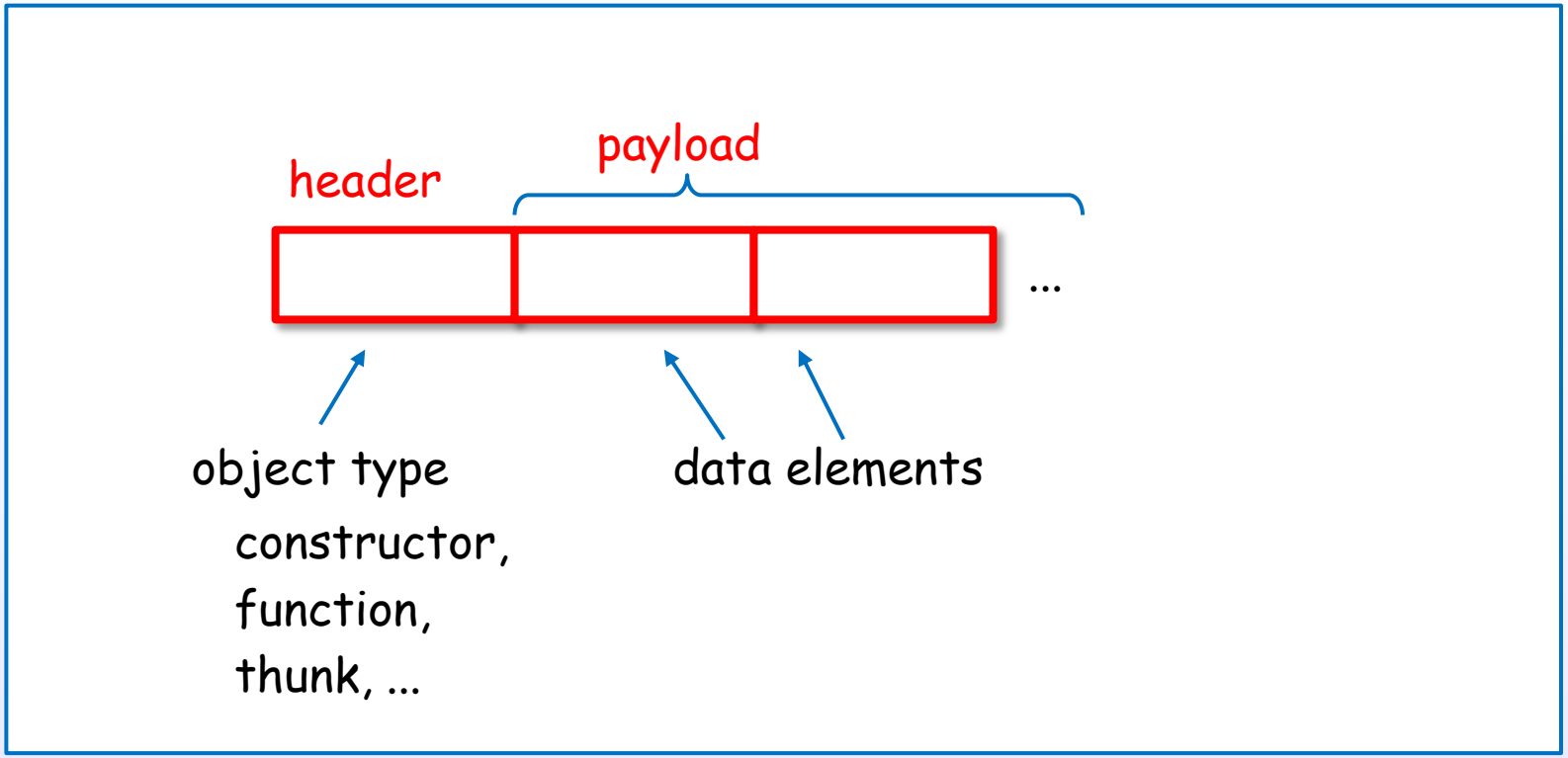


References : [1]

3. Internal representation of expressions

Uniform representation

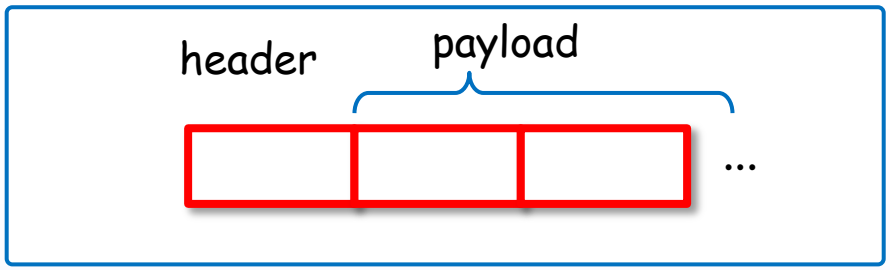
統一内部表現で表現される



in heap memory, stack, registers or static memory

References : [1]

統一内部表現



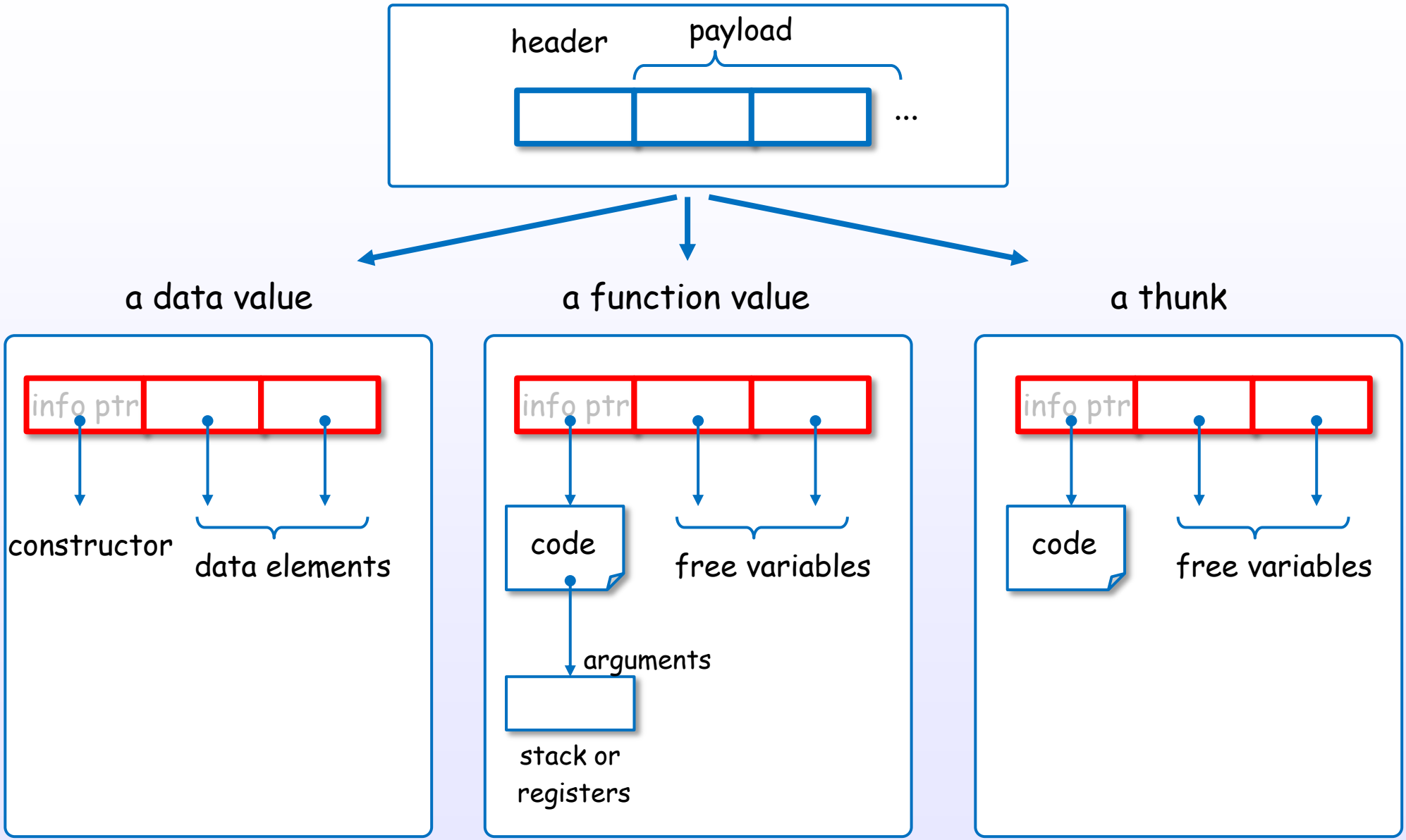
a data value

a function value

a thunk

References : [1]

統一内部表現

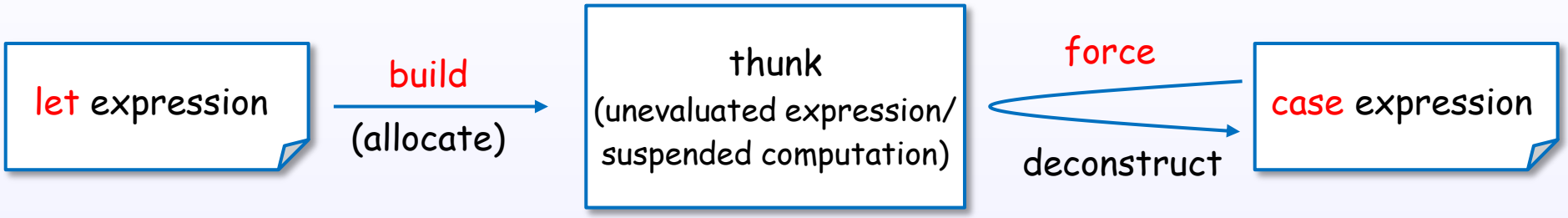


References : [1]

3. Internal representation of expressions

let, case expression

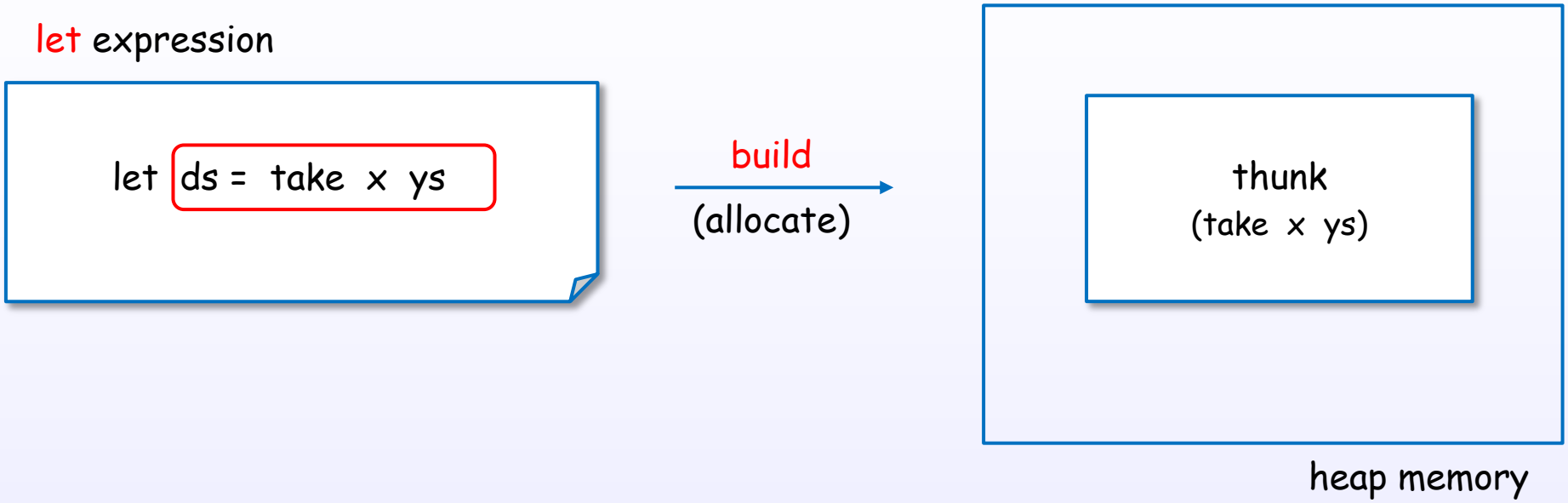
let/case expressions and thunk



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

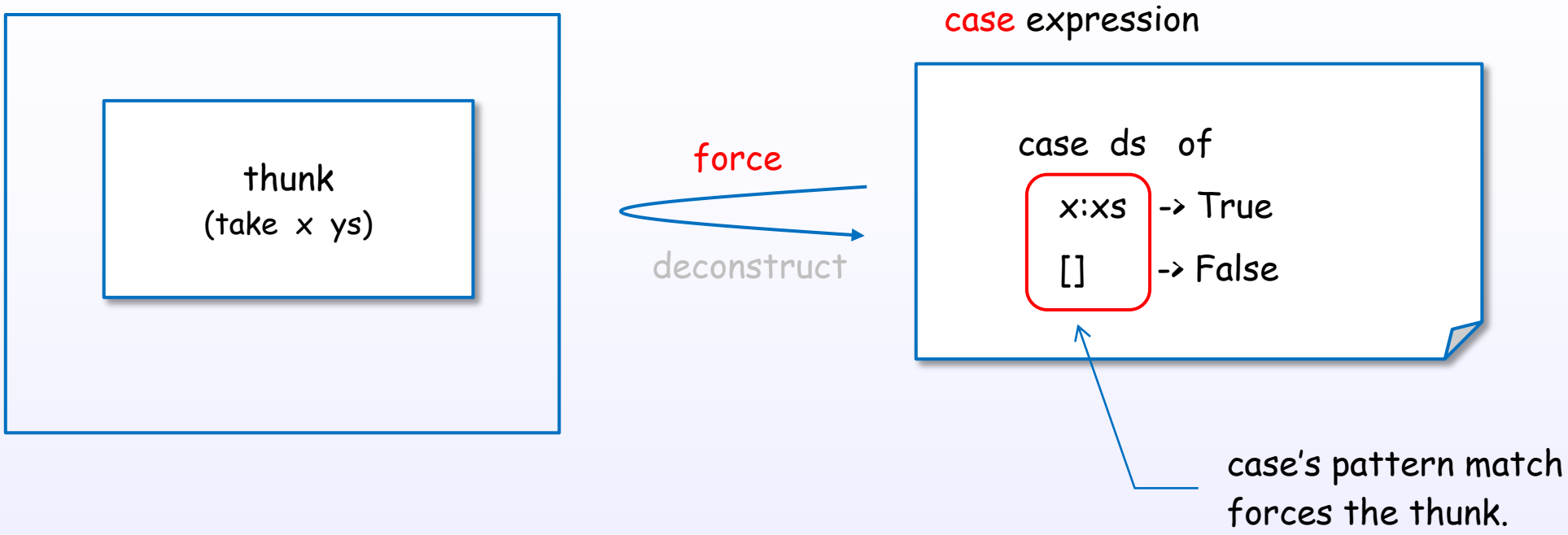
References : [1]

A let expression builds a thunk



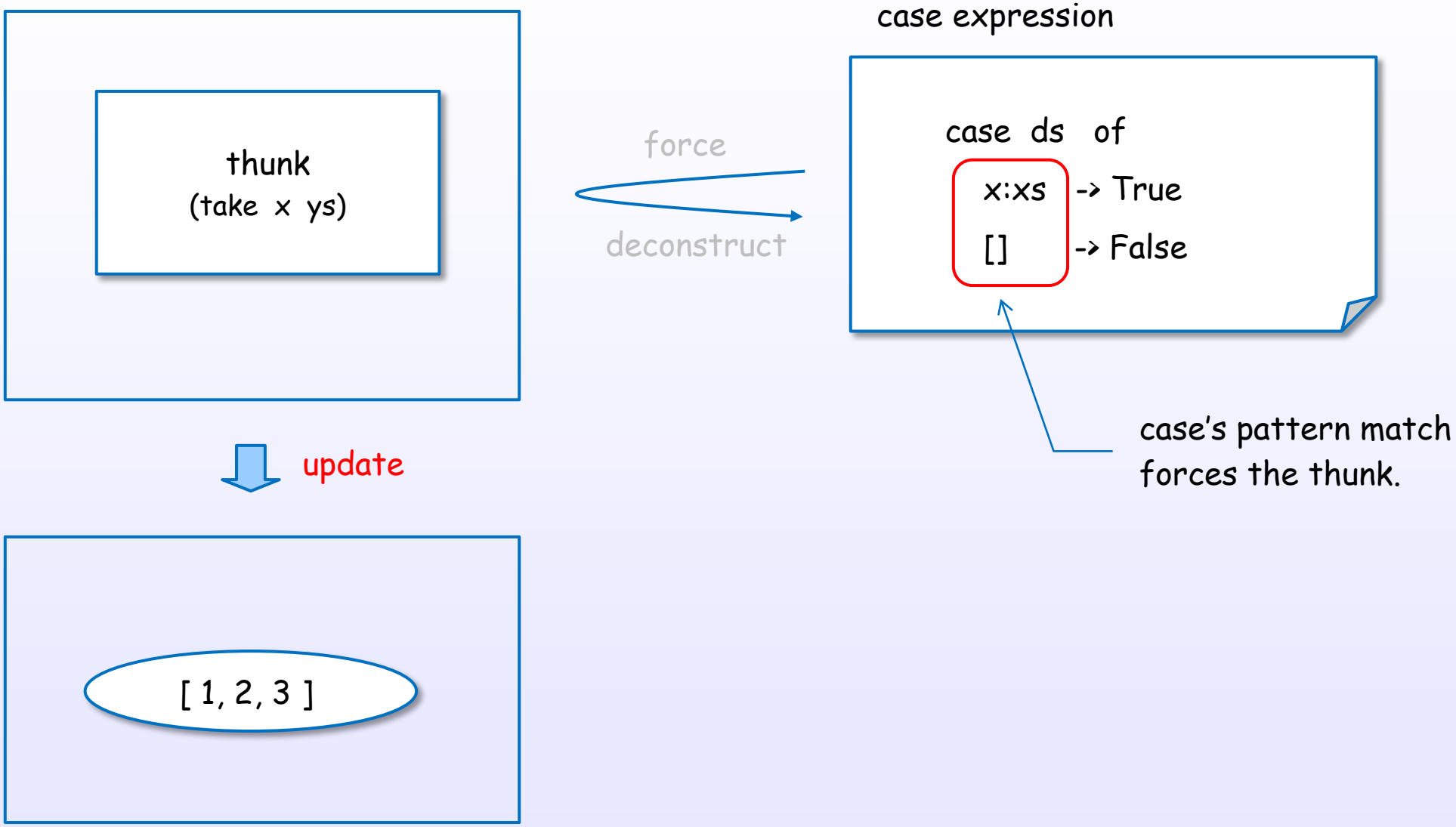
References : [1]

A case expression forces a thunk



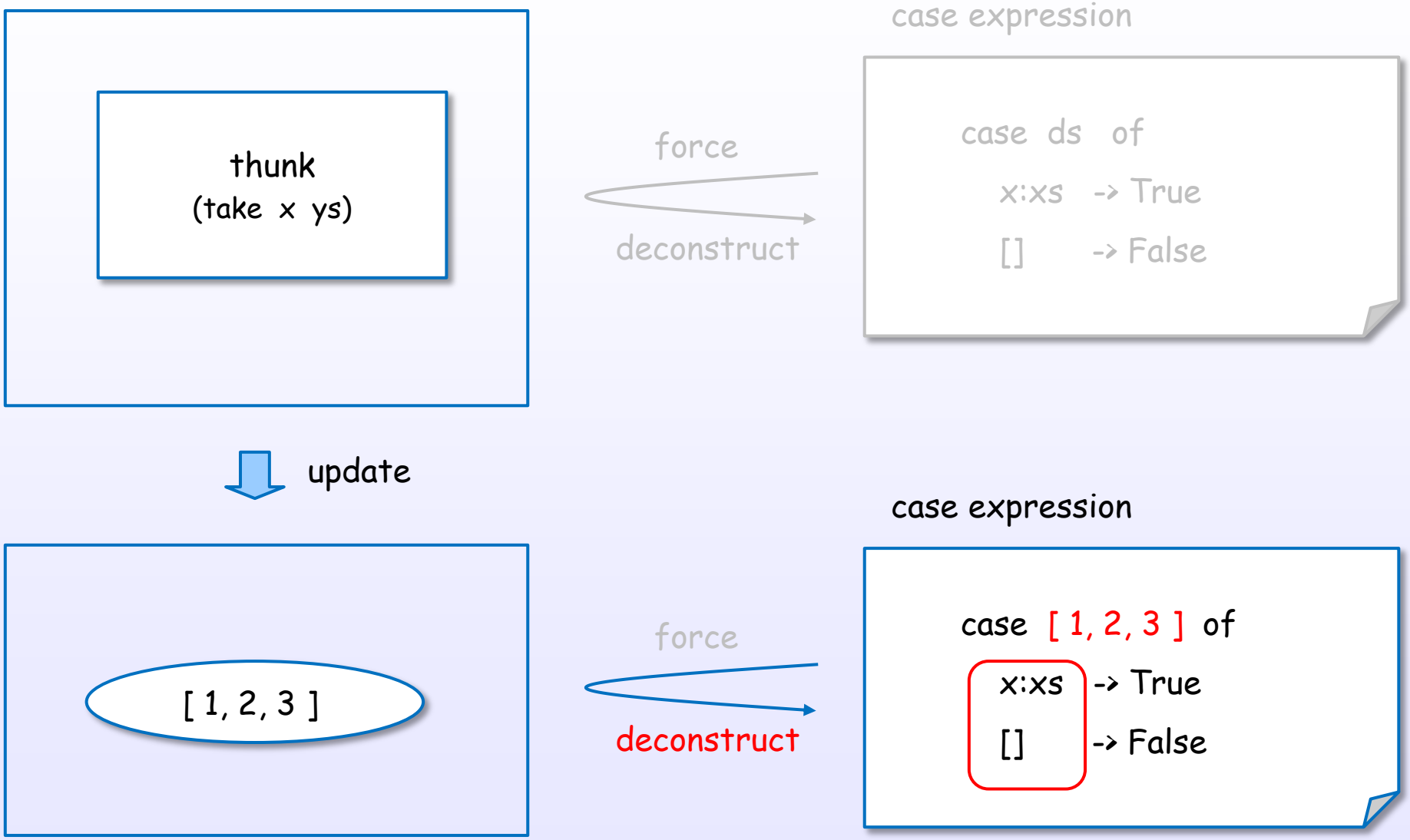
References : [1]

A case expression forces a thunk



References : [1]

A case expression forces a thunk



References : [1]

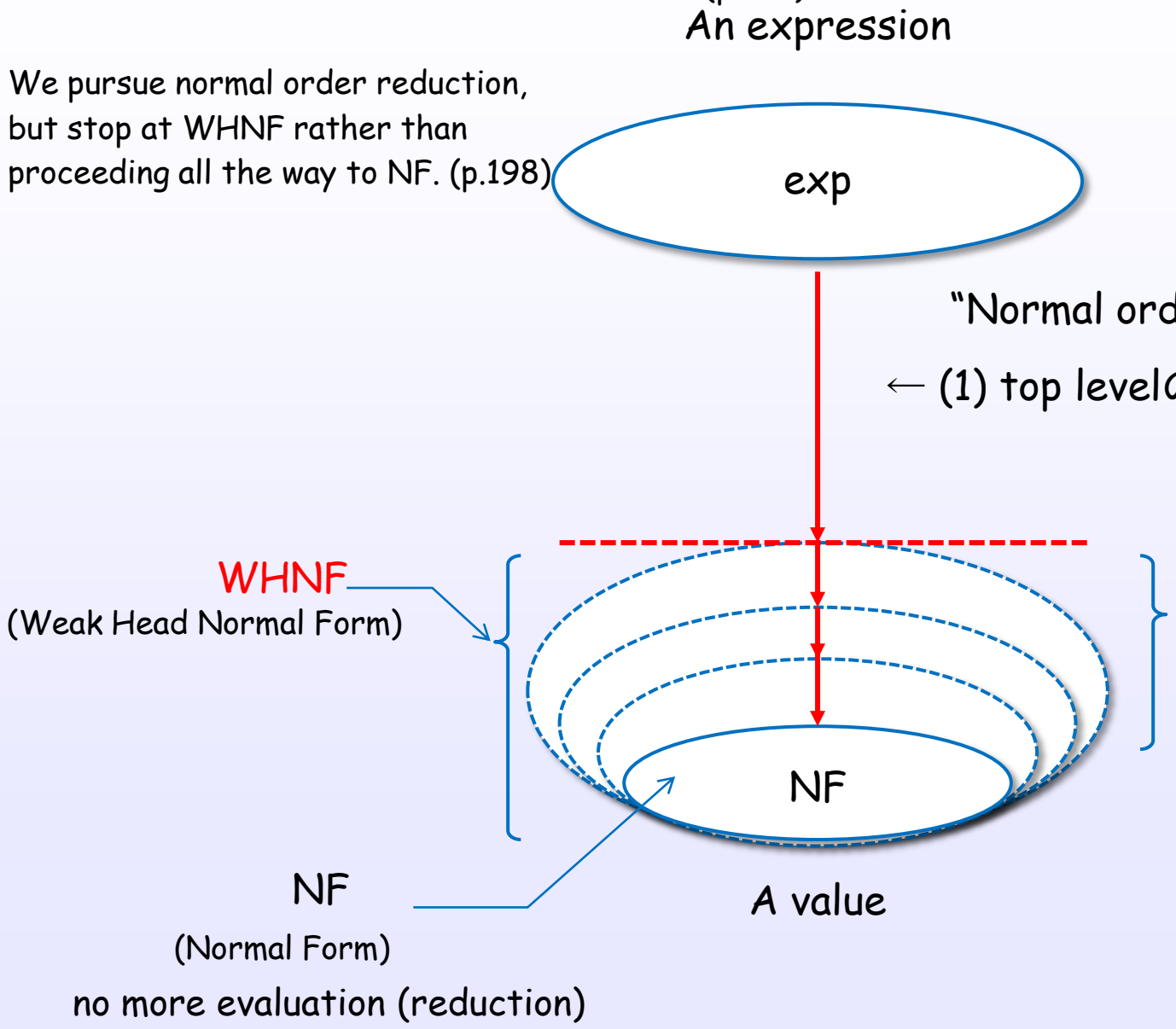
3. Internal representation of expressions

WHNF

evaluation step (GHC)

Our reduction order is therefore to reduce the top-level redex until weak head normal form is reached. (p.198)

We pursue normal order reduction, but stop at WHNF rather than proceeding all the way to NF. (p.198)



[Terei]

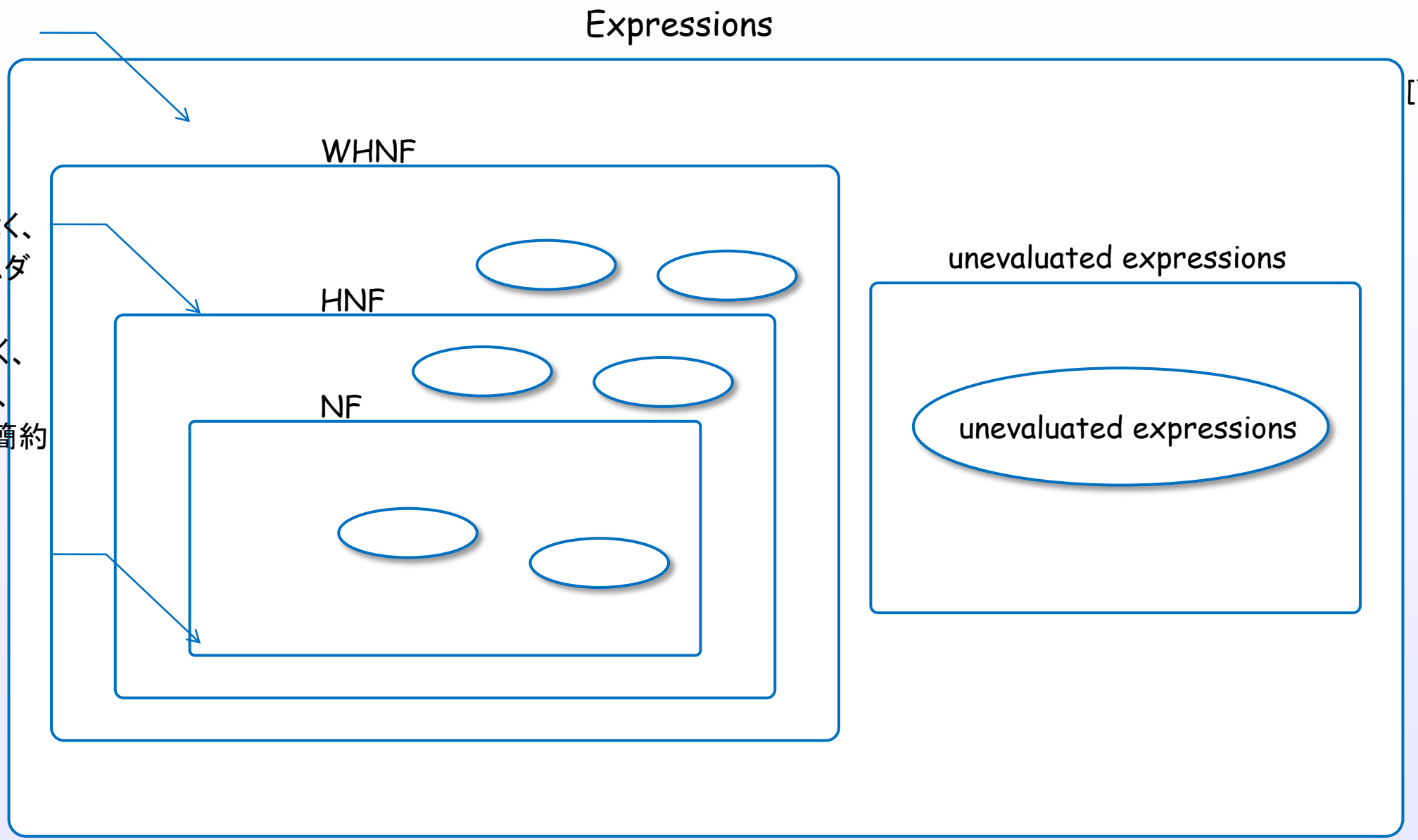
[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

References : [1]

再掲

evaluation level

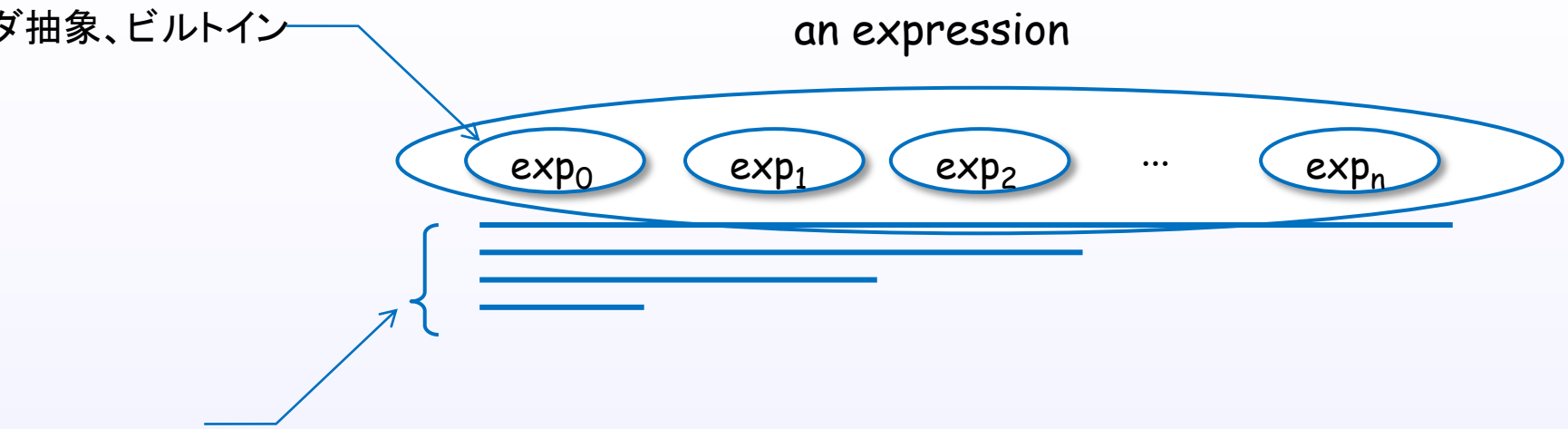


値には、評価レベルがある。

[STG]

References : [1]

WHNF



more

An expression has no top level redex, if it is in WHNF.

[slpj-book-1987]

These are in weak head normal form,
but not in normal form, since they contain inner redex. (p.198)

[Terei]

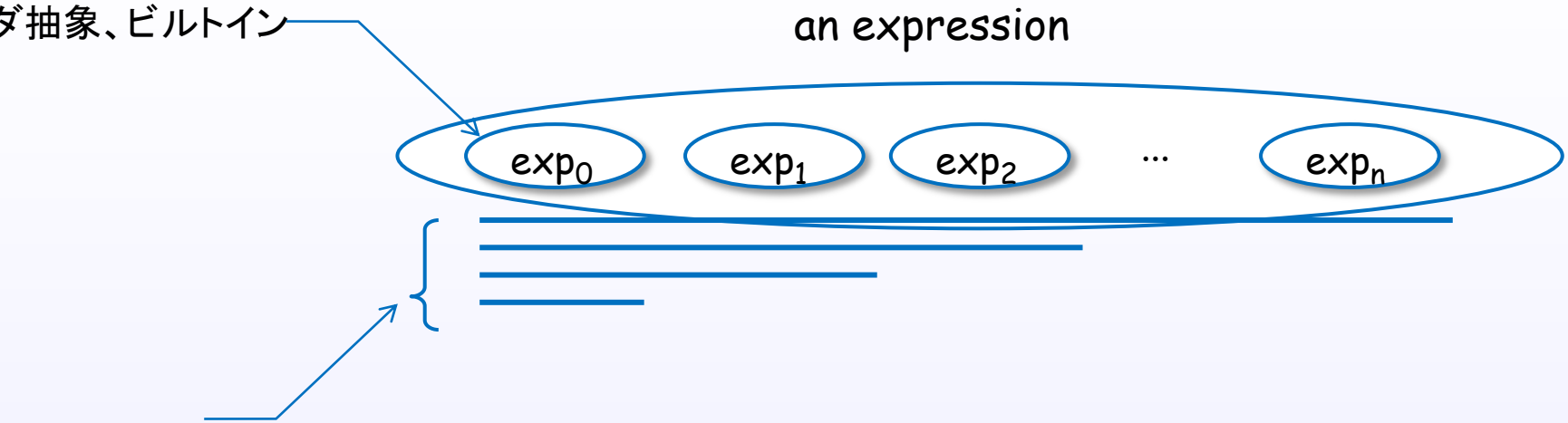
[Terei]

[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

References : [1]

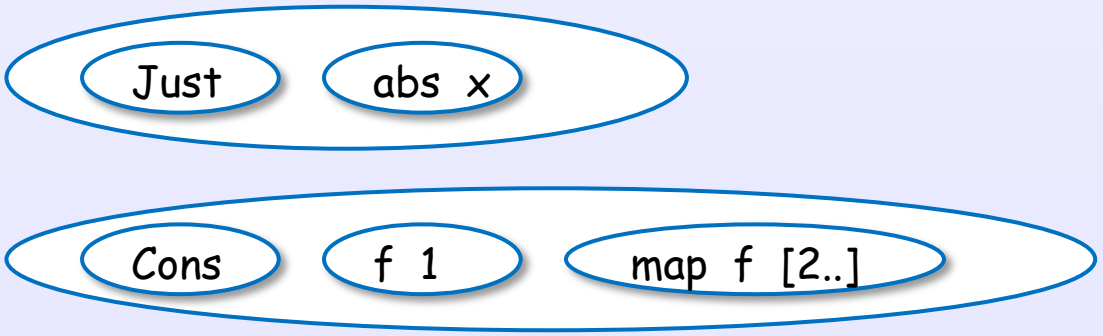
Examples of WHNF



Just (abs x)

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

more



[slpj-book-1987]

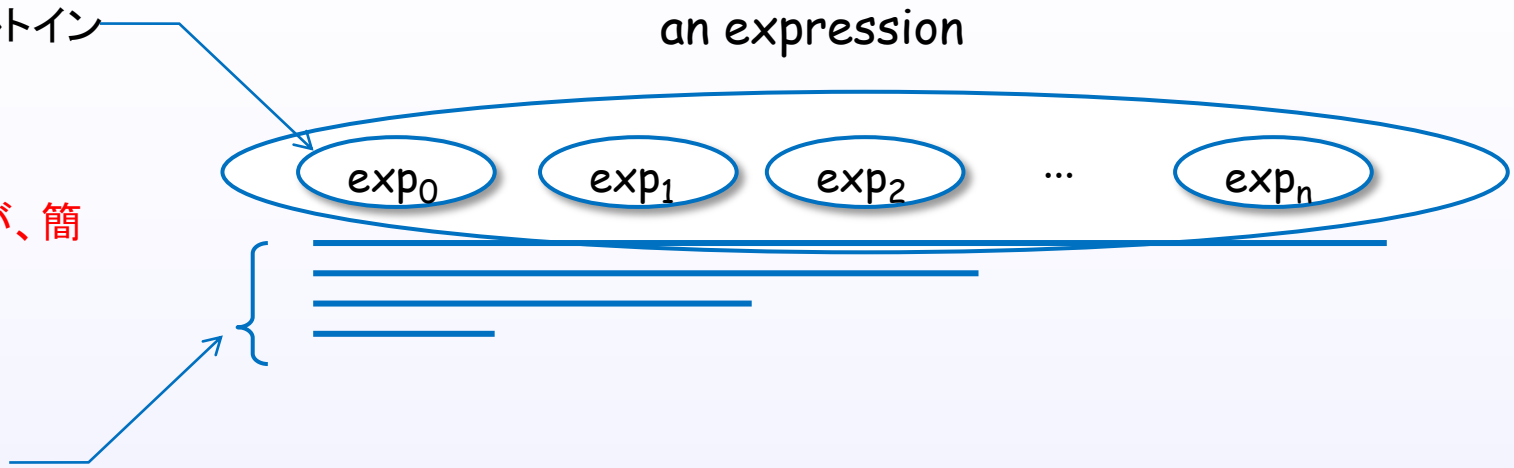
References : [1]

HNF

ダ抽象、ビルトイン

内側(body)が、簡

more

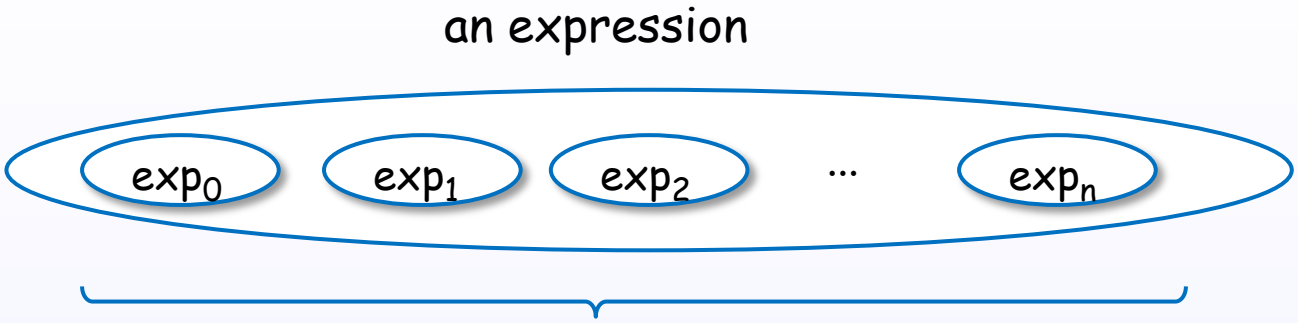


[slpj-book-1987]

[Terei]

References : [1]

NF



redexが内部に無い

[slpj-book-1987]

[Terei]

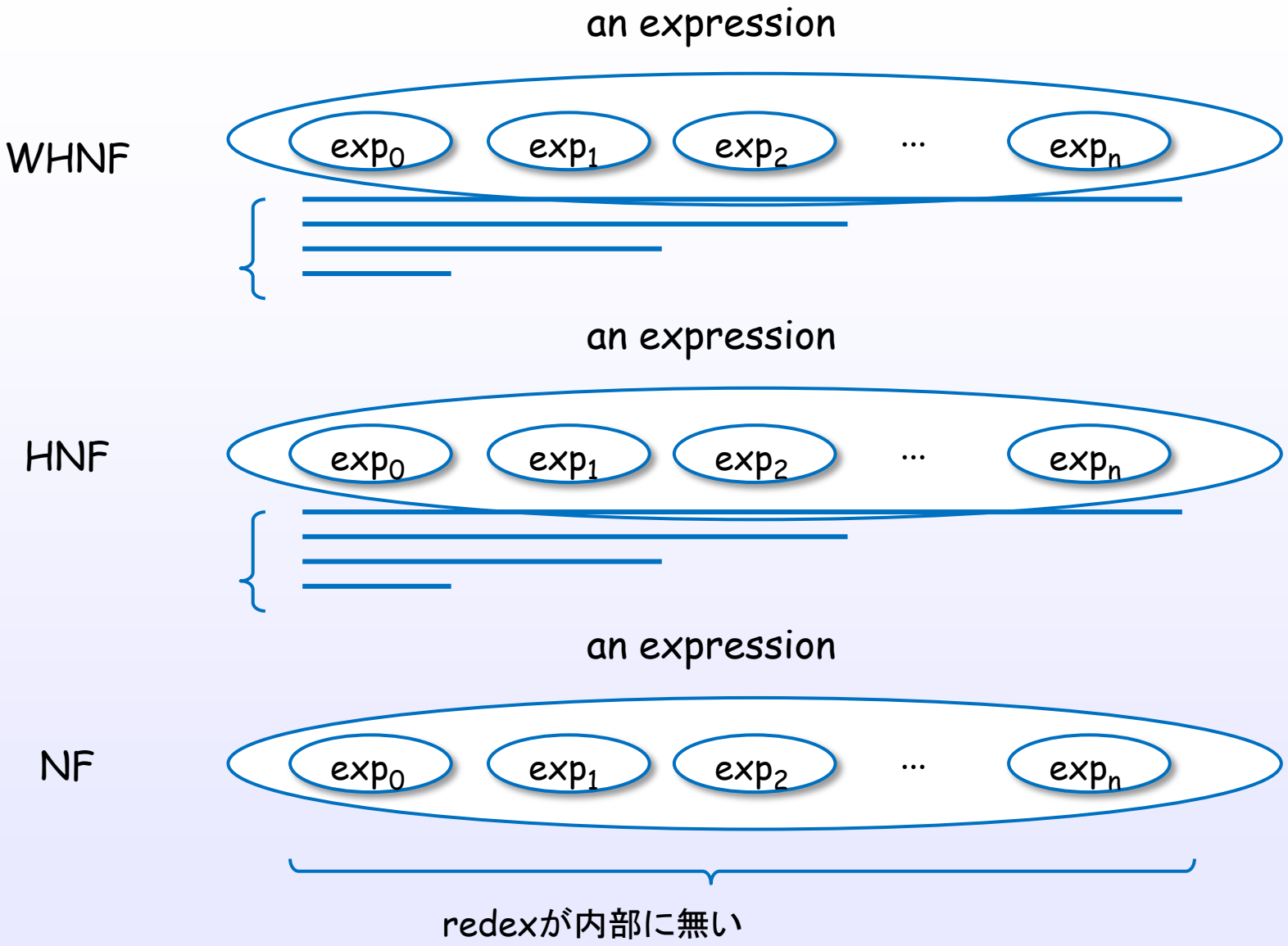
[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

[Terei]

References : [1]

WHNF, HNF, NF



[slpj-book-1987]

References : [1]

definition of WHNF and HNF

The implementation of functional programming languages [19]

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 confus
some discussion. The content of
since for most purposes head nor
form. Nevertheless, we will stick t

DEFINITION

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

$$\lambda x_1.\lambda x_2.\ldots\lambda x_n.(v\ M_1\ M_2\ \ldots\ M_m)$$

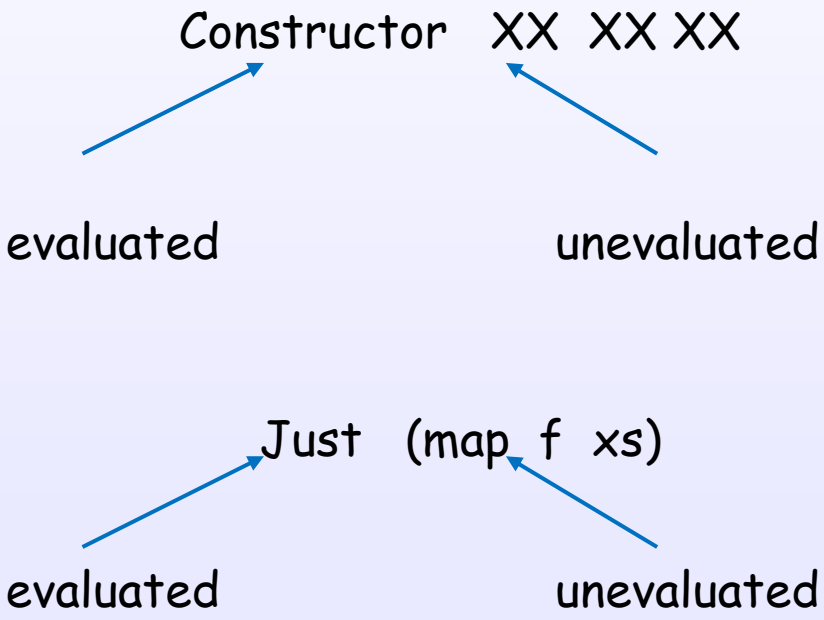
where $n, m \geq 0$;
v is a variable (x_i), a data object, or a built-in function;
and $(v\ M_1\ M_2\ \ldots\ M_p)$ is not a redex for any $p \leq m$.

[slpj-book-1987]

References : [1]

internal representation of WHNF

heap objectイメージ



References : [1]

4. Evaluation

4. Evaluation

Evaluation in Haskell (GHC)

GHC chosen lazy evaluation

必要な時に、必要な箇所のみを評価する

(STG p.11)

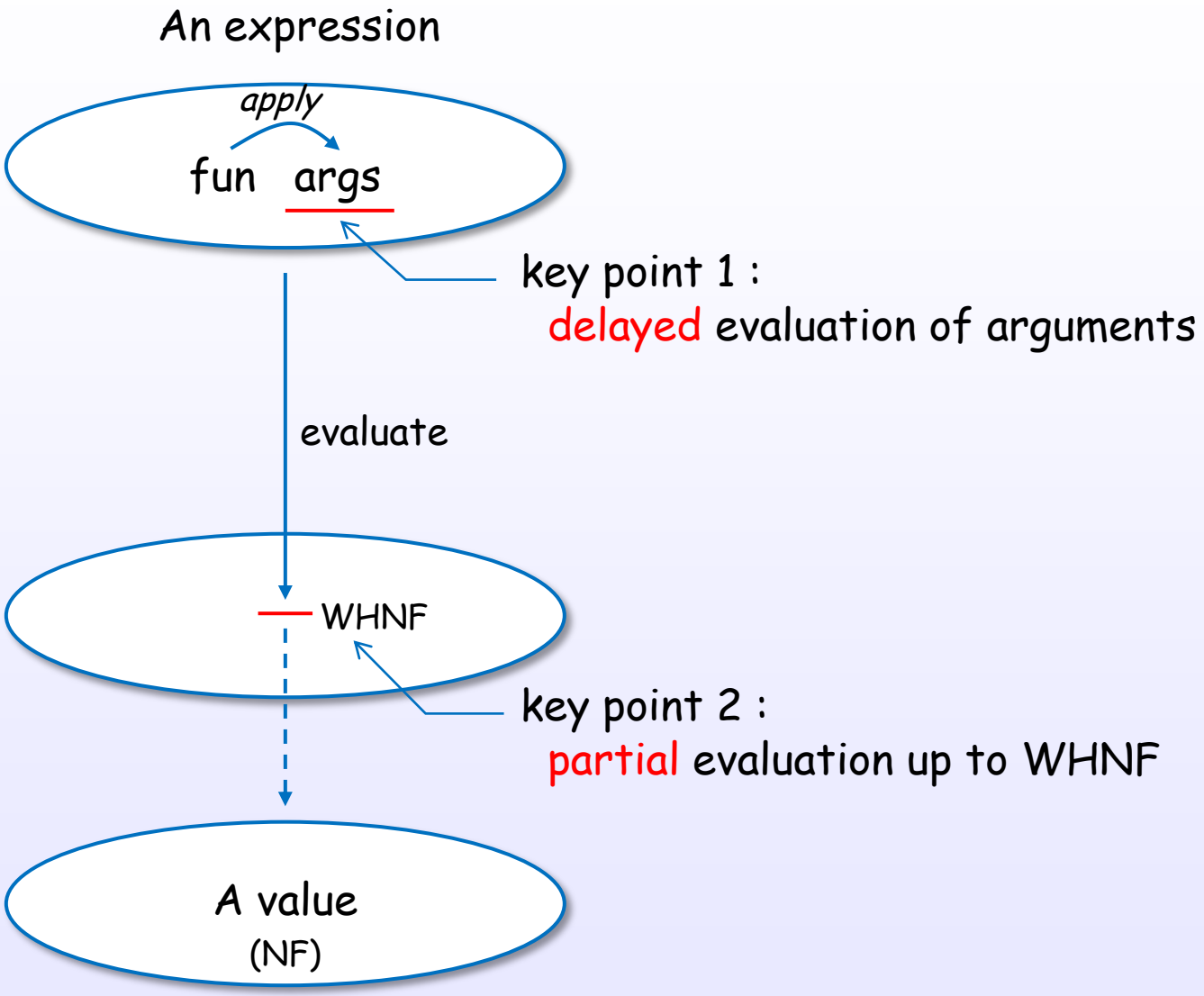
・引数評価を先送る （case式が来るまで評価しない） call-by-need

・部分式を完全評価しない （caseのパターンマッチで参照するところのみを評価する）WHNF

これは、計算量を最小化する戦略（メモリ量でなく）

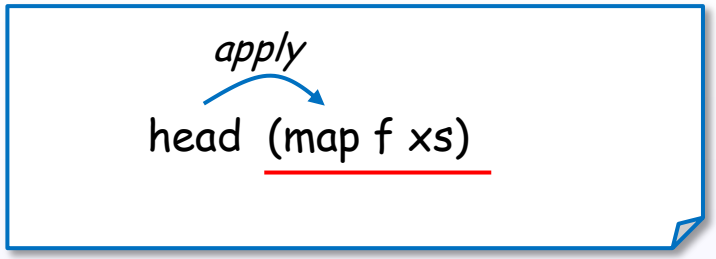
References : [1]

Key concept of Haskell's lazy evaluation

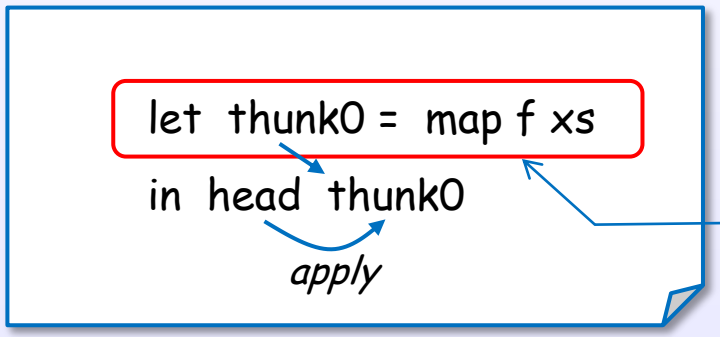


References : [1]

key point 1 : delayed evaluation of arguments



internal transformation by GHC

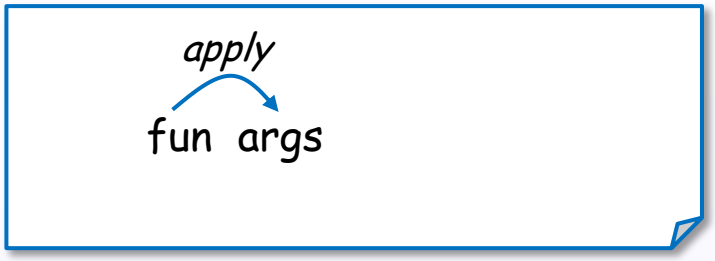


build a thunk
in heap memory

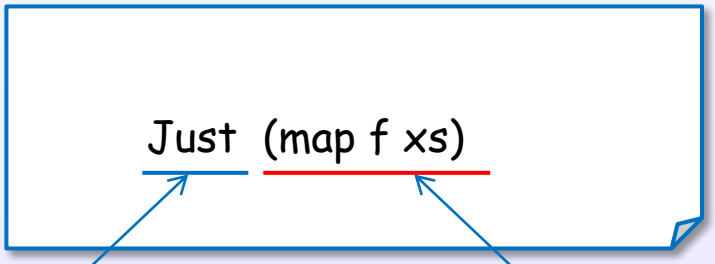
GHC implements lazy evaluation using the thunk.
Evaluation of arguments is delayed with the thunk.

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).

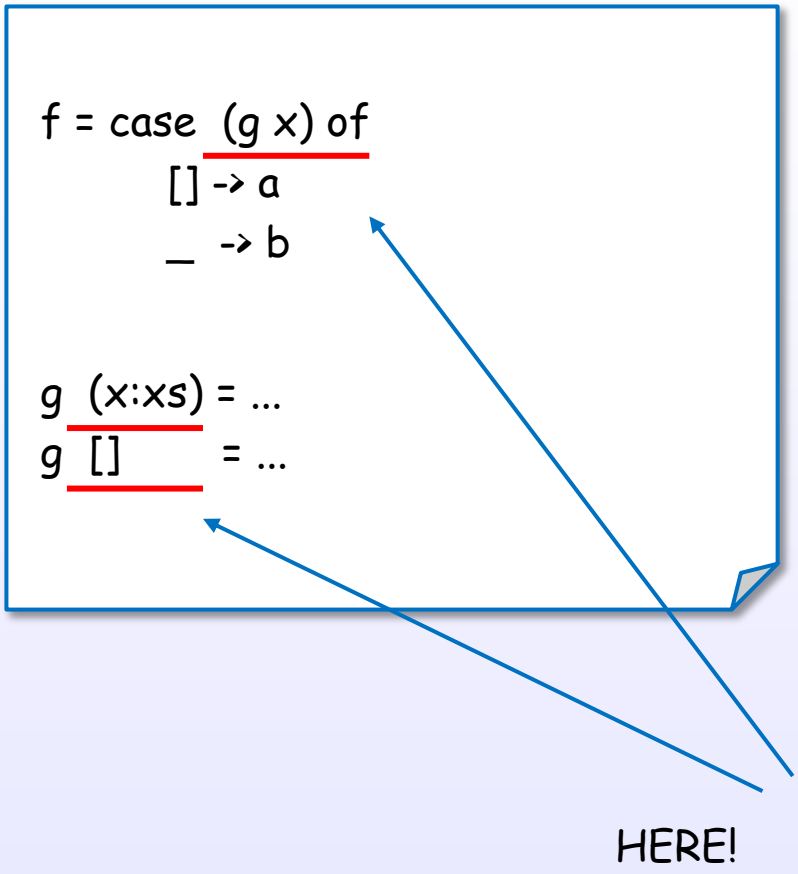
References : [1]

では、必要なときはいつか？

References : [1]

では、必要なときはいつか？

Haskell code



pattern match via
case expression and function definition
will {cause, trigger} the evaluation

References : [1]

Pattern match

[CIS194]

References : [1]

4. Evaluation

Examples of evaluation steps

Example of repeat

repeat 1

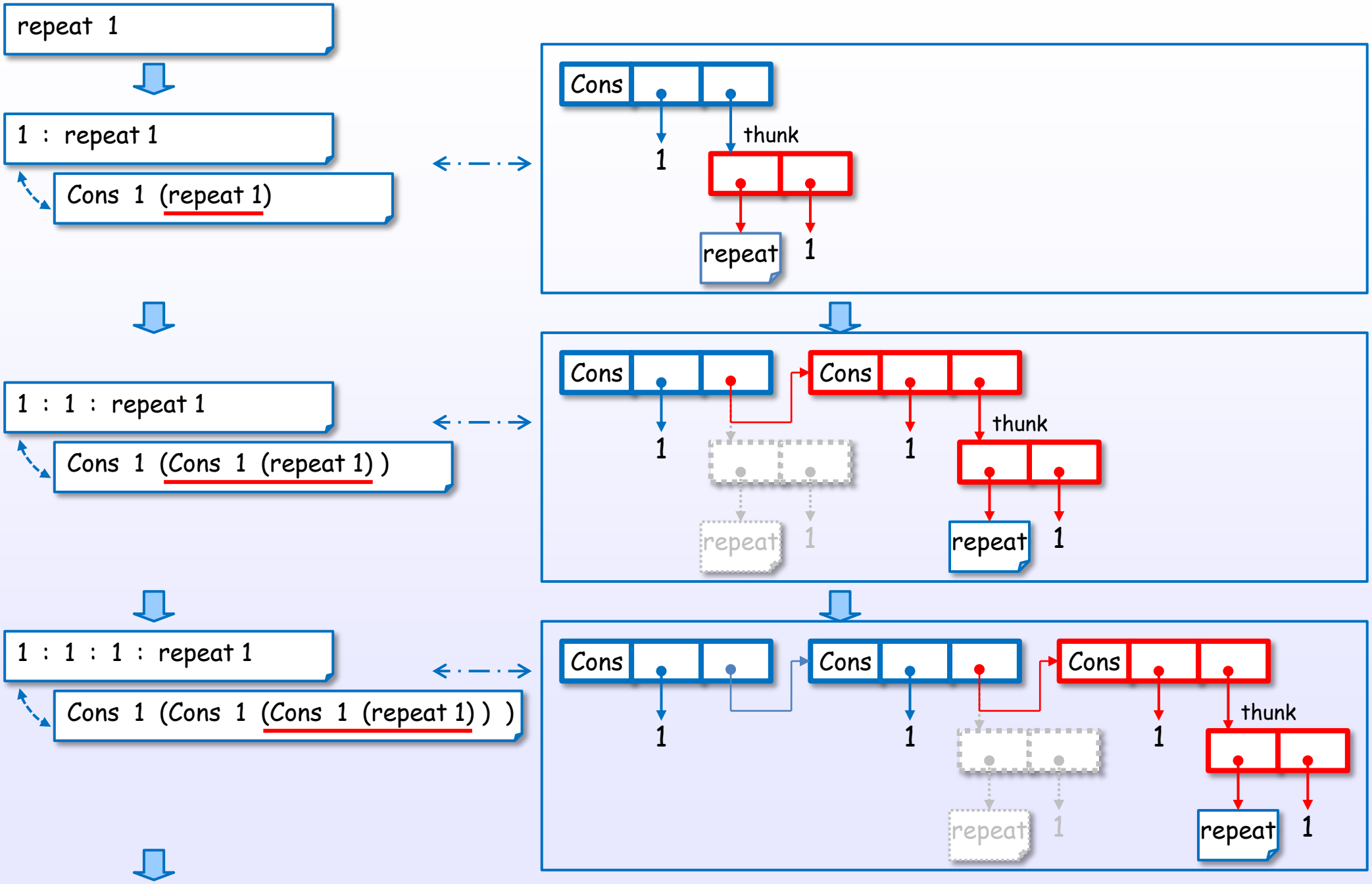
1 : repeat 1

1 : 1 : repeat 1

1 : 1 : 1 : repeat 1

References : [1]

Example of repeat



References : [1]

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
```

...

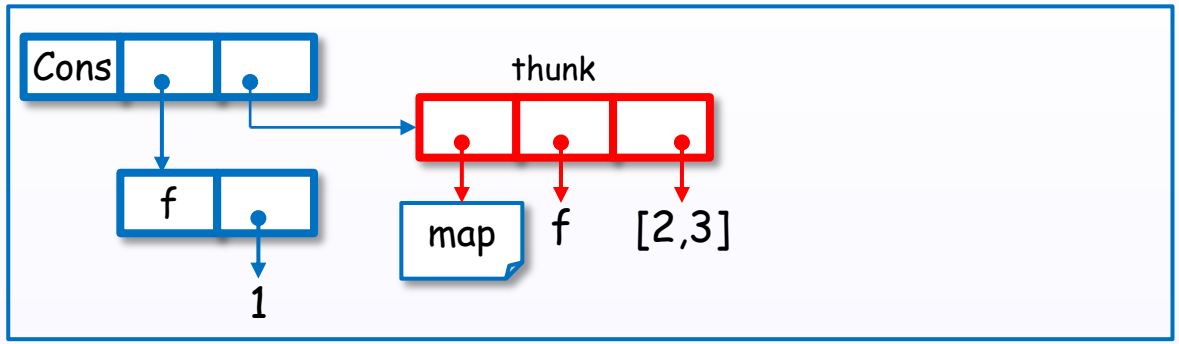
References : [1]

Example of map

map f [1, 2, 3]

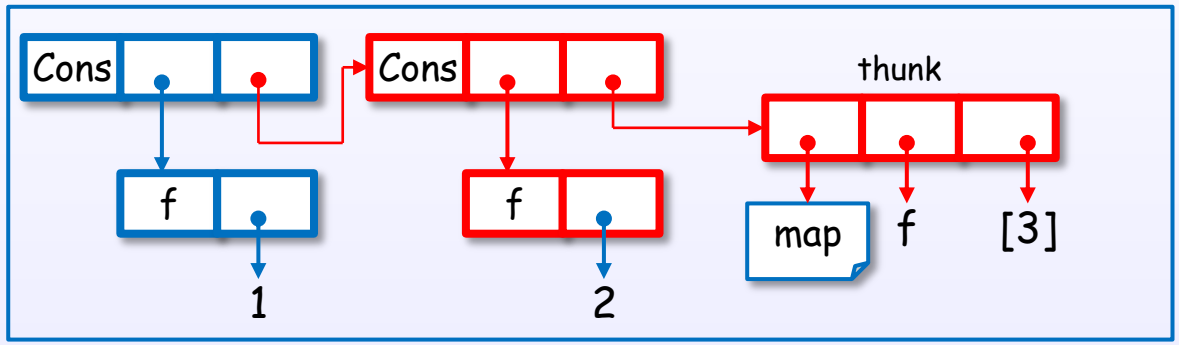
f 1 : map f [2, 3]

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



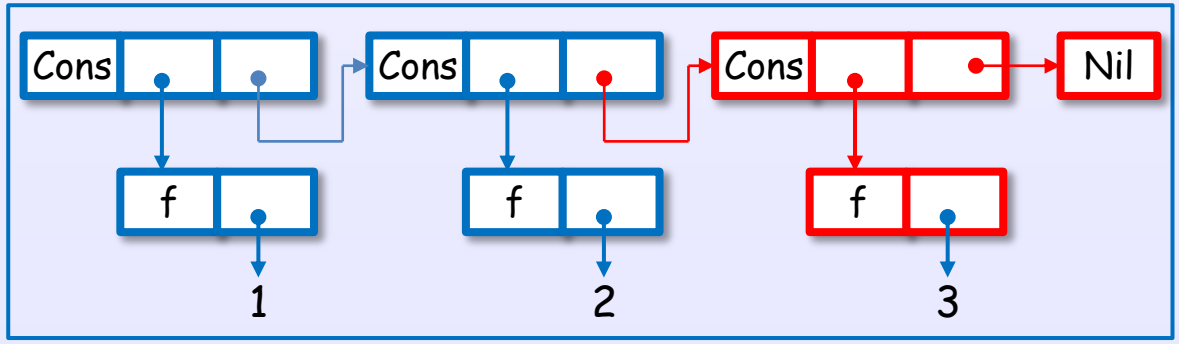
f 1 : f 2 : map f [3]

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))

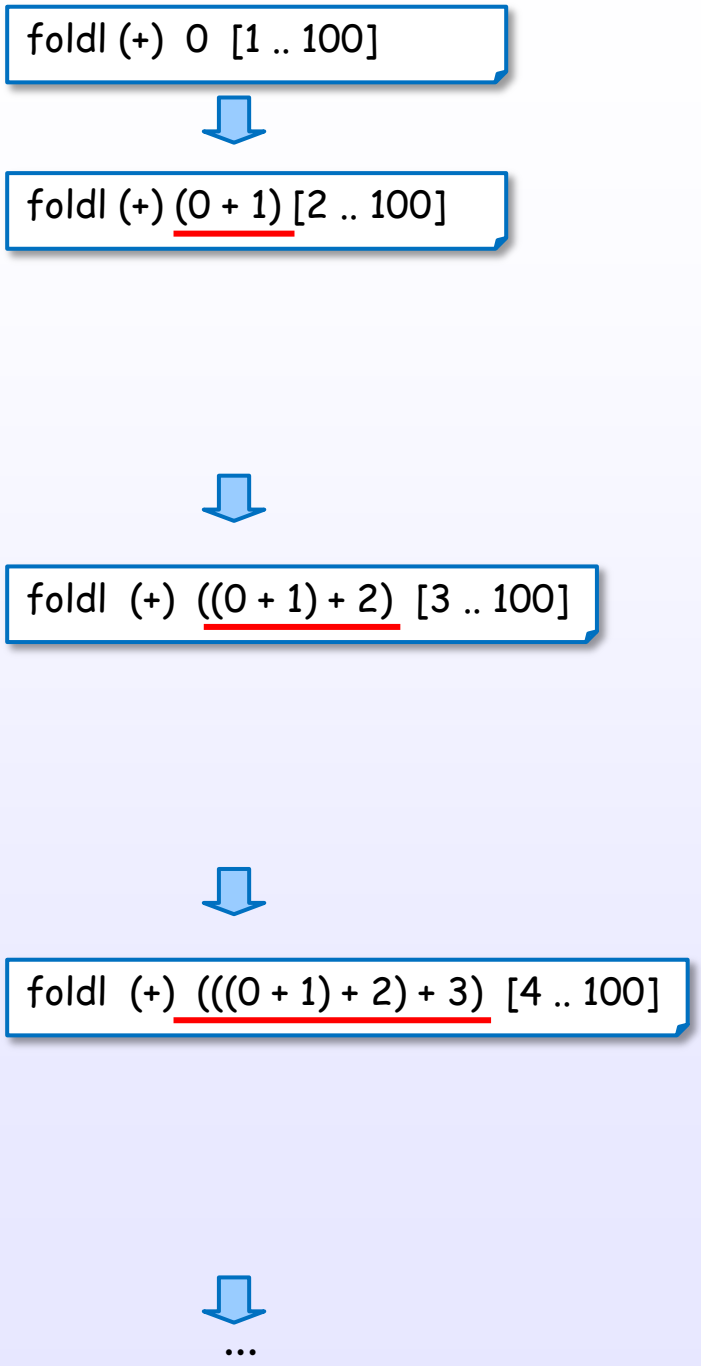


...

...

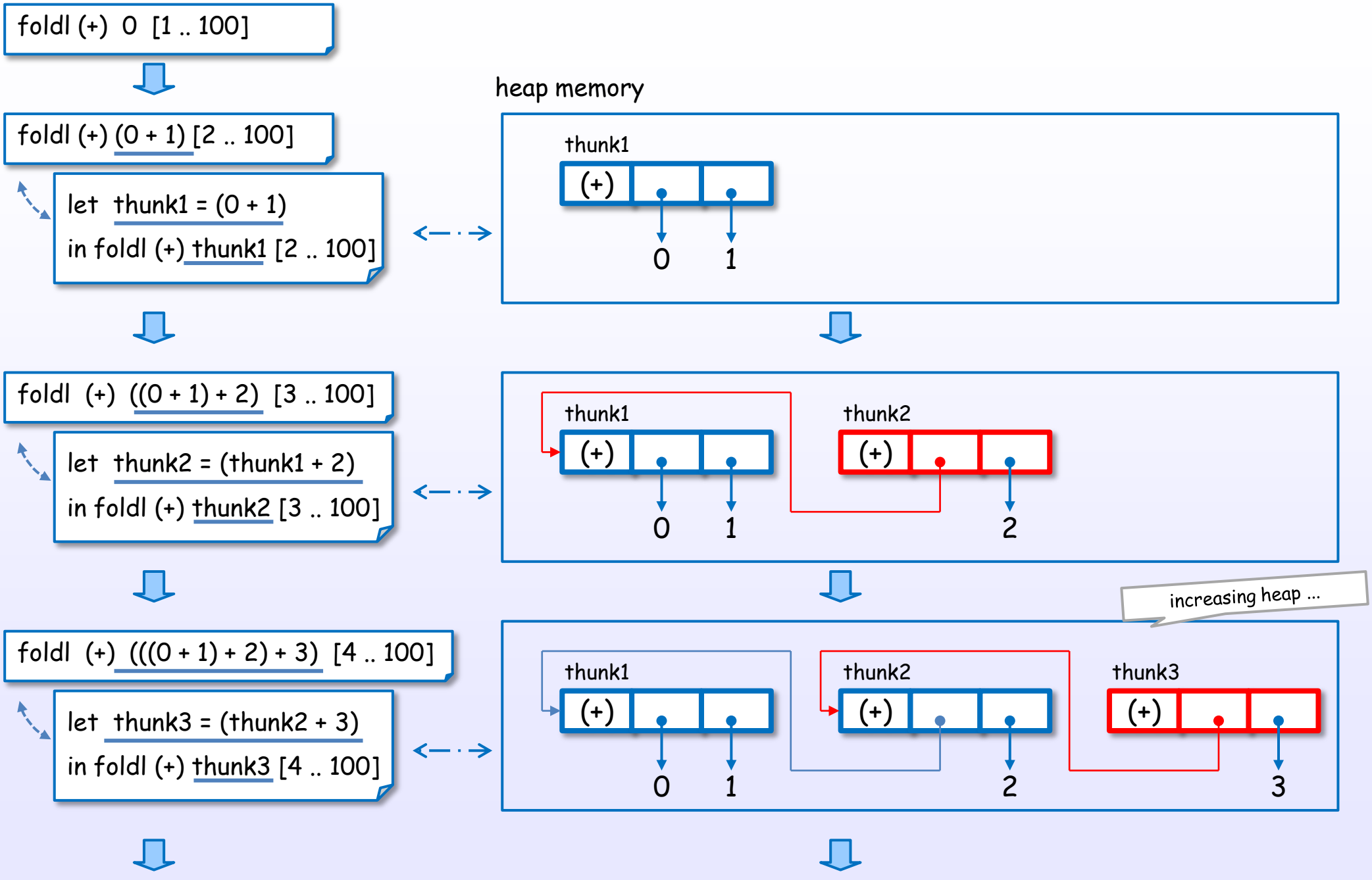
References : [1]

Example of foldl (non-strict)



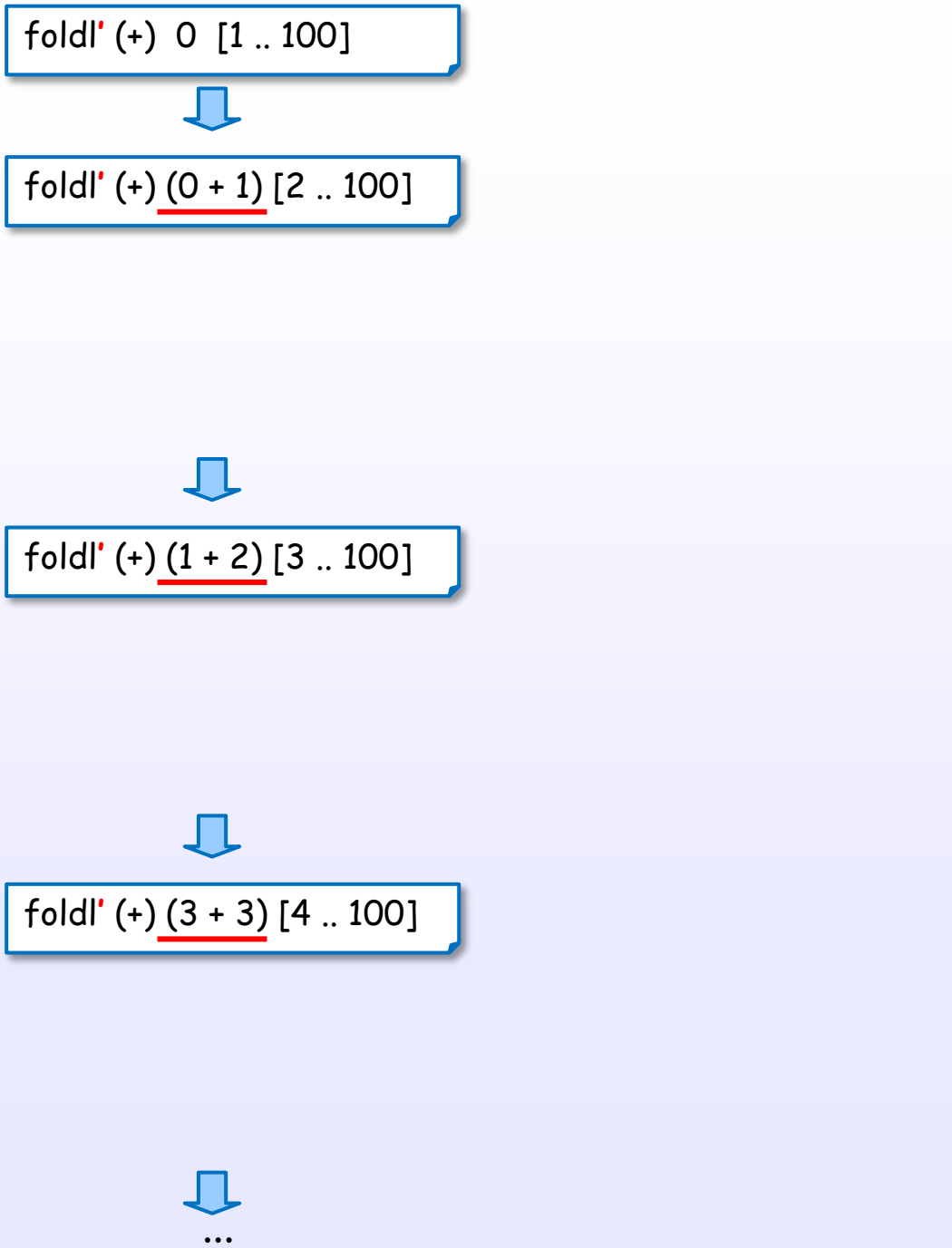
References : [1]

Example of foldl (non-strict)



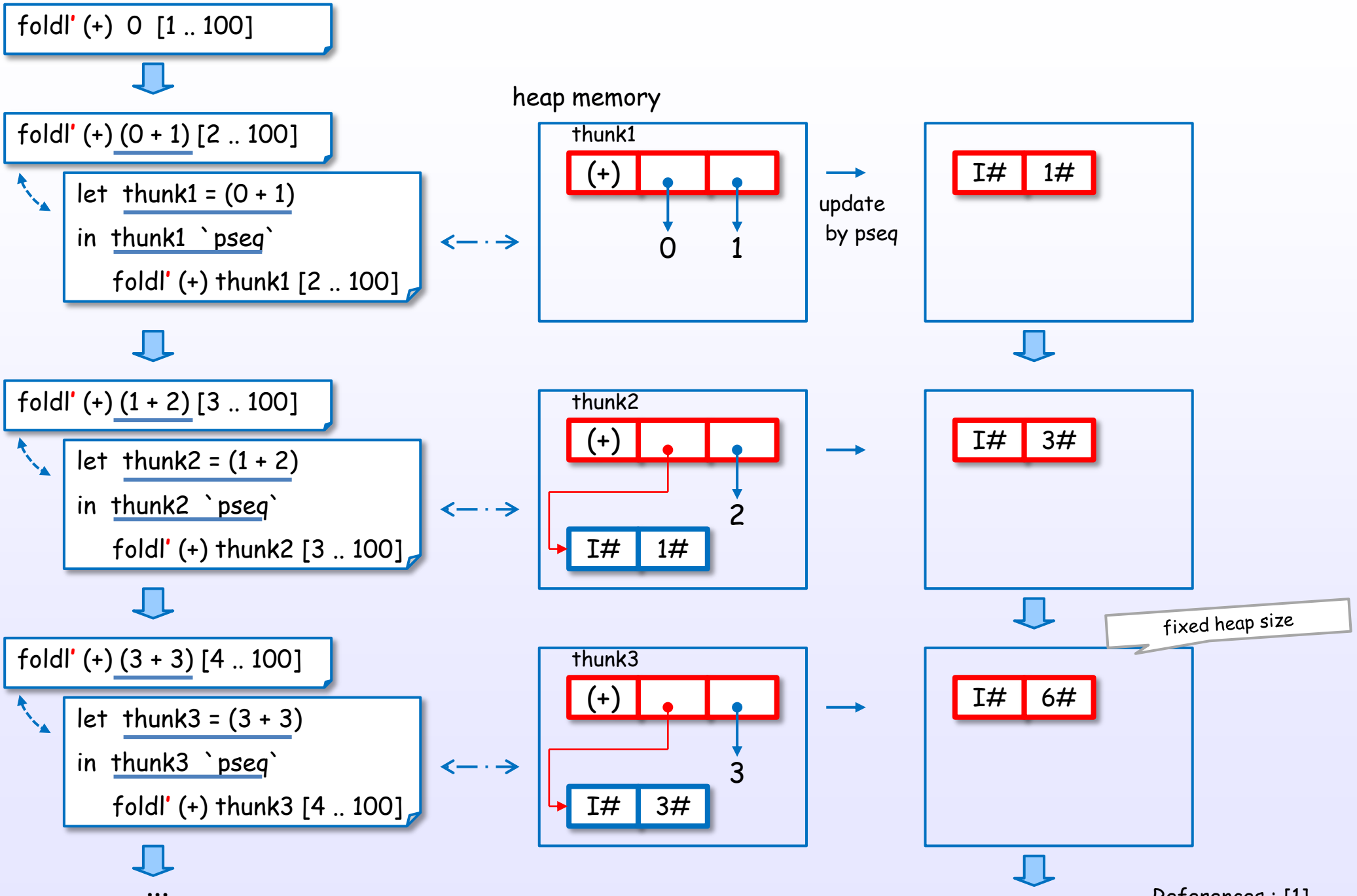
References : [1]

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]

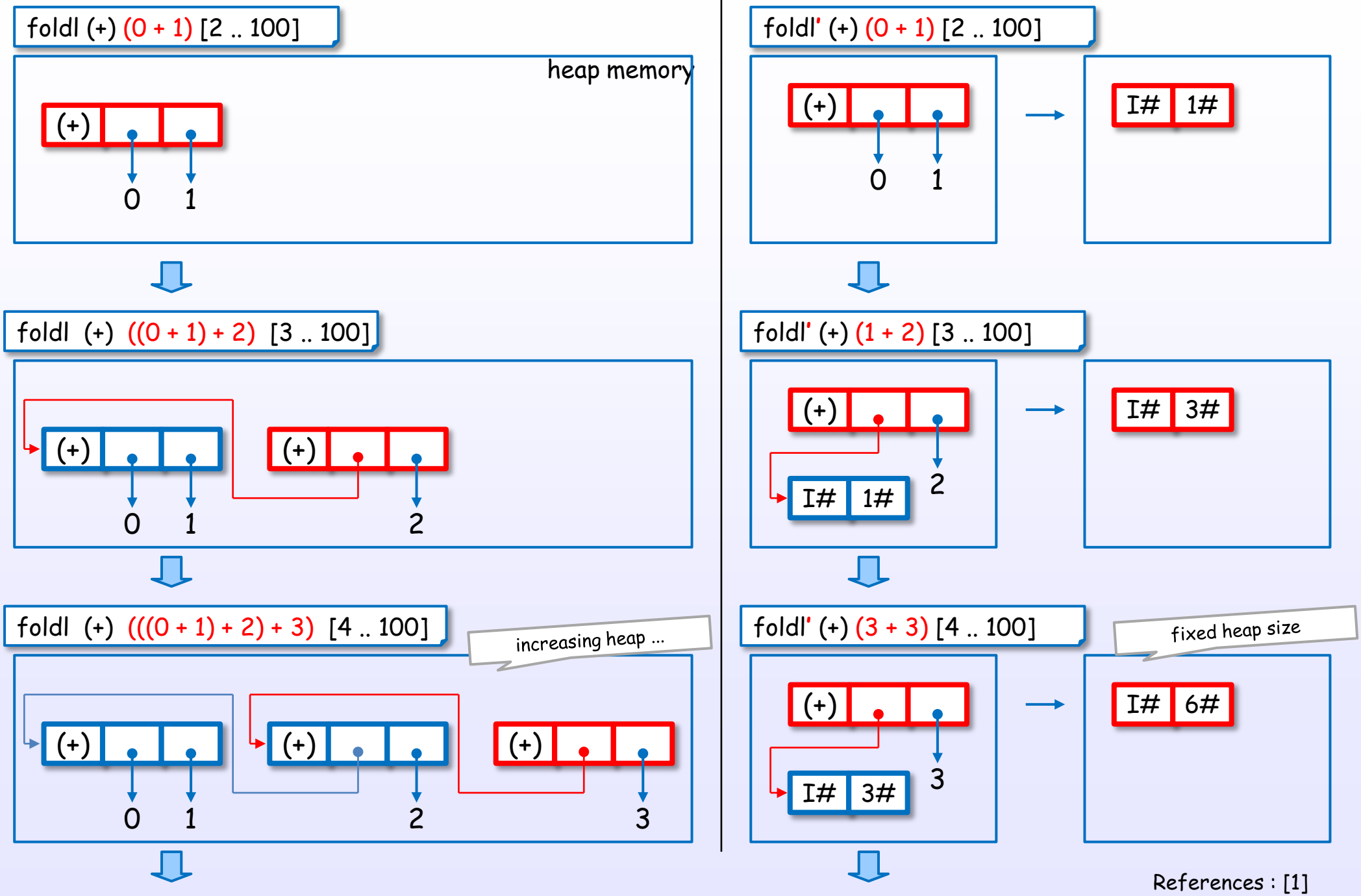
foldl' (+) (0 + 1) [2 .. 100]

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

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

References : [1]

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

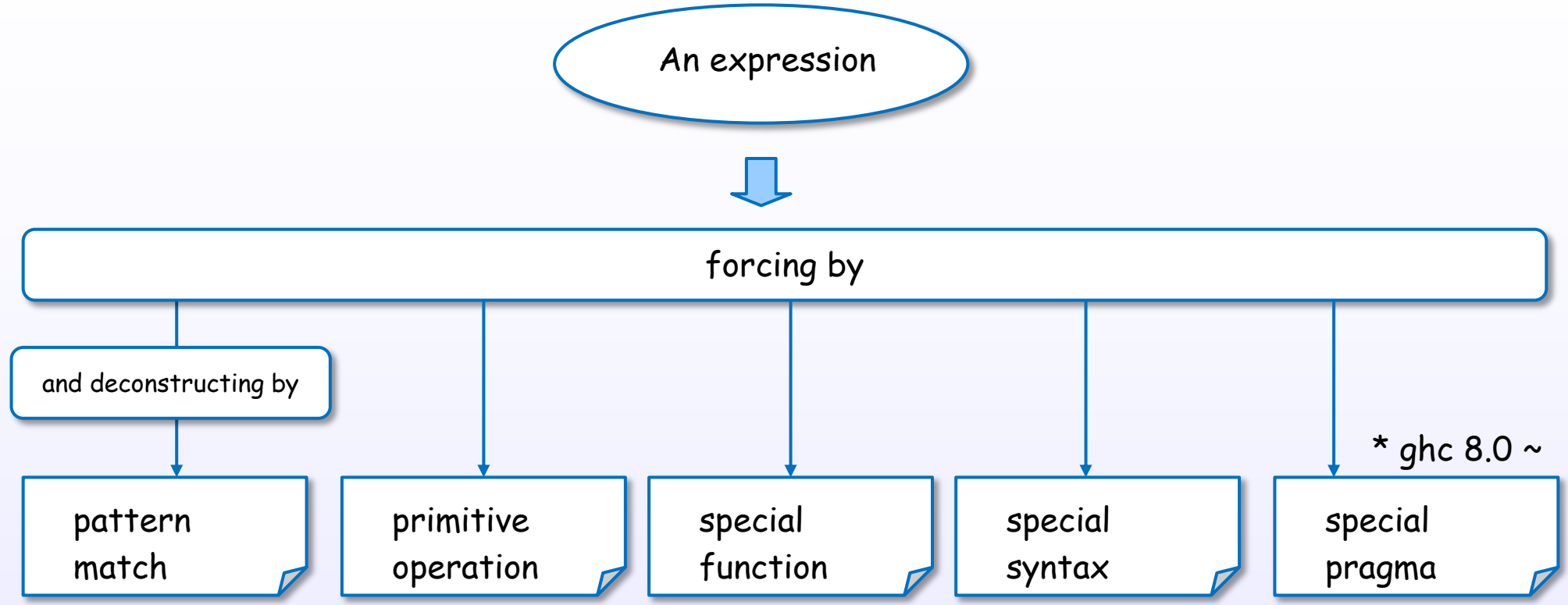


References : [1]

4. Evaluation

Control the evaluation in Haskell

How to drive evaluation



References : [1]

Example of the evaluation by pattern match

case expression

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

case expression
function definition

References : [1]

Example of the evaluation by primitive operation

primitive operation

```
f x y = x + y
```

+, *, ...

References : [1]

Example of the evaluation by special function

special function

```
f x y = seq x y
```

seq
$!
rnf
deepseq
force
$!!
pseq
,...

Please refer the document more detail. [xx]
hoogle or hackage

References : [1]

Example of the evaluation by special syntax

special syntax

```
{-# LANGUAGE BangPatterns #-}

f !xs = g xs
```

BangPattern

!

Please refer the document more detail. [xx]

[user guide, 7.19]

References : [1]

Example of the evaluation by special pragma

special pragma

```
{-# LANGUAGE Strict #-}

f xs = g xs
```

* ghc 8.0 ~

```
{-# LANGUAGE StrictData #-}

f xs = g xs
```

Strict
StrictData

Please refer the document more detail. [xx] [wiki]

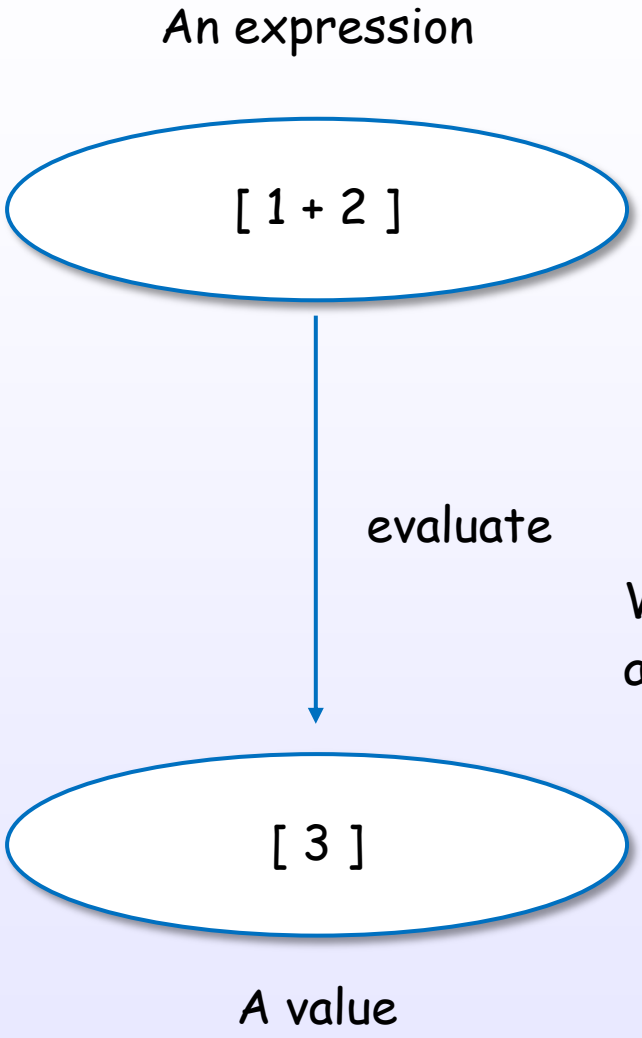
References : [1]

5. Semantics

5. Semantics

Bottom

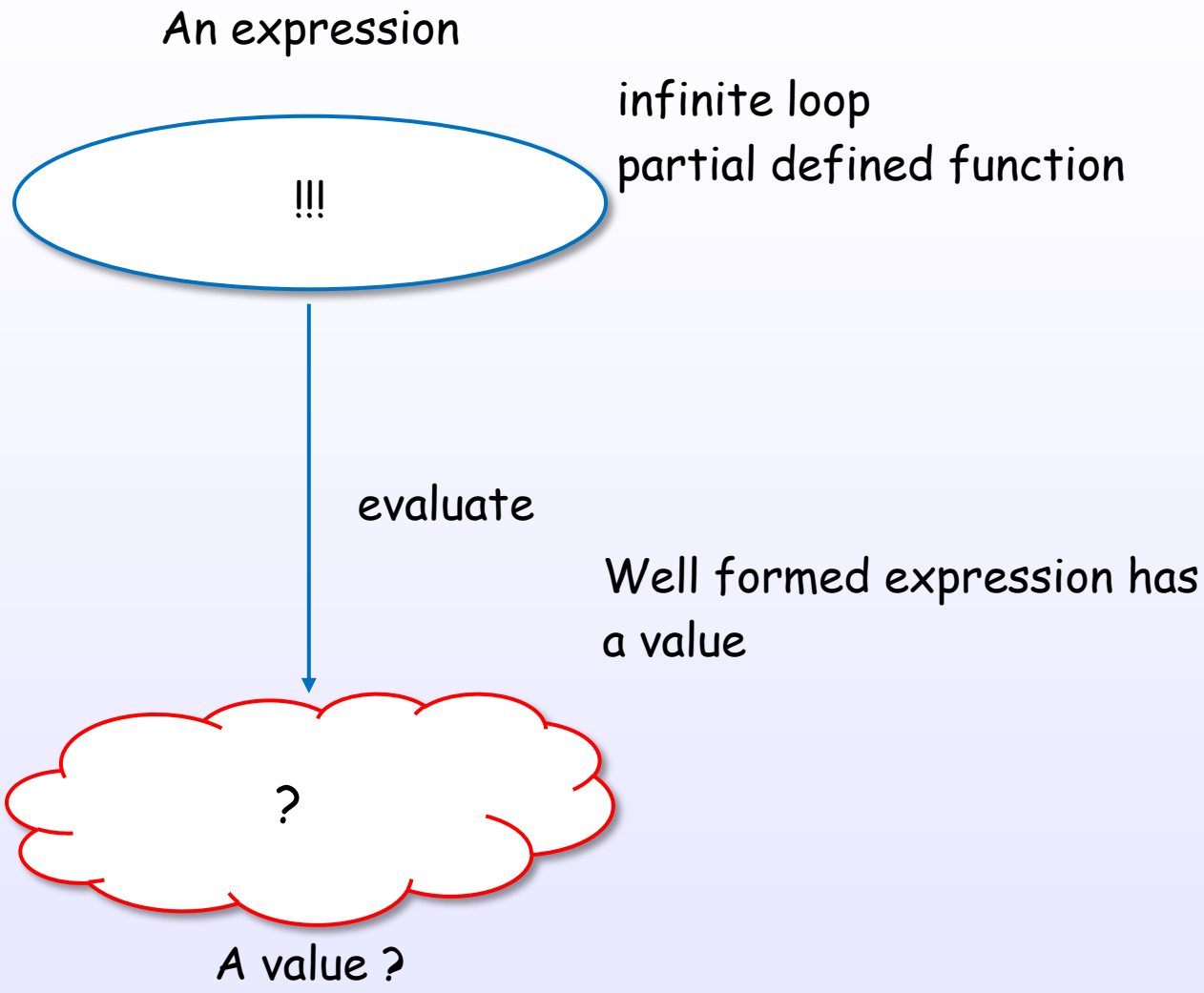
Well formed expression has a value



[Bird, Chapter 2]

References : [1]

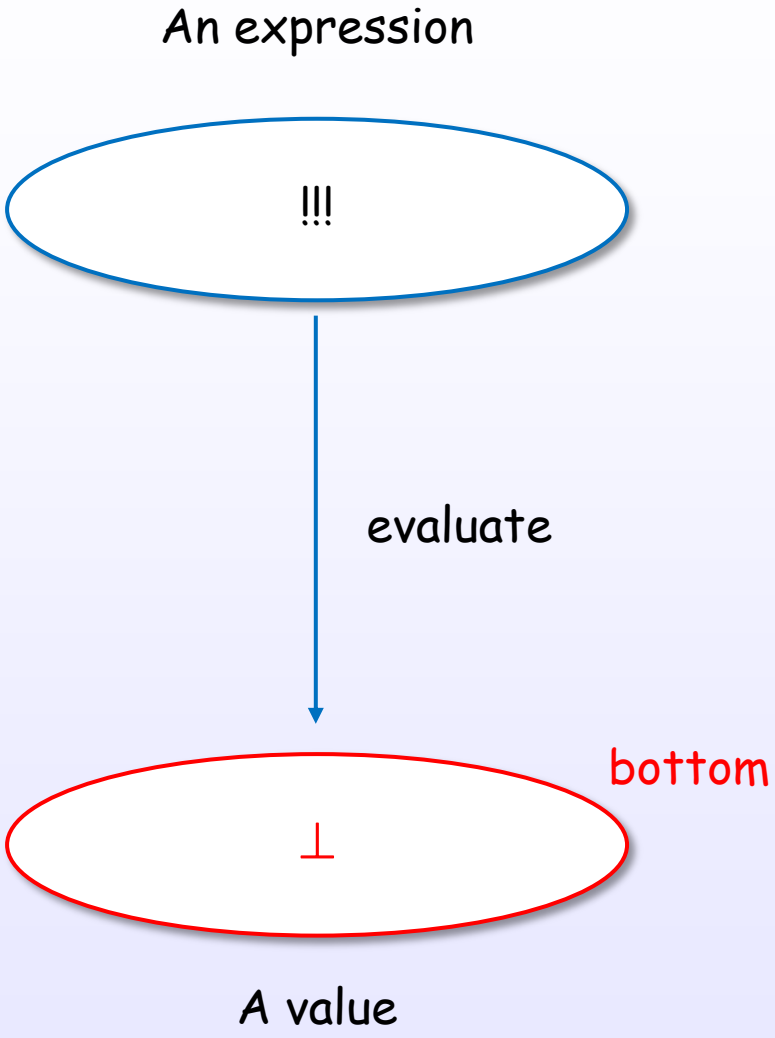
Well formed expression has a value



[Bird, Chapter 2]

References : [1]

Well formed expression has a value



[Bird, Chapter 2]

References : [1]

Bottom

[Bird, Chapter 2]

References : [1]

Non-strict Semantics

Strictness

$f \perp = \perp$

[Bird, Chapter 2]

References : [1]

Layer

Non-strictness	f ⊥ = ⊥
Lazy evaluation	GHC chosen lazy evaluation to implement non-strict semantics.
Graph reduction	GHC chosen graph reduction to implement lazy evaluation.
STG-machine	GHC implements graph reduction by STG-machine.

References : [1]

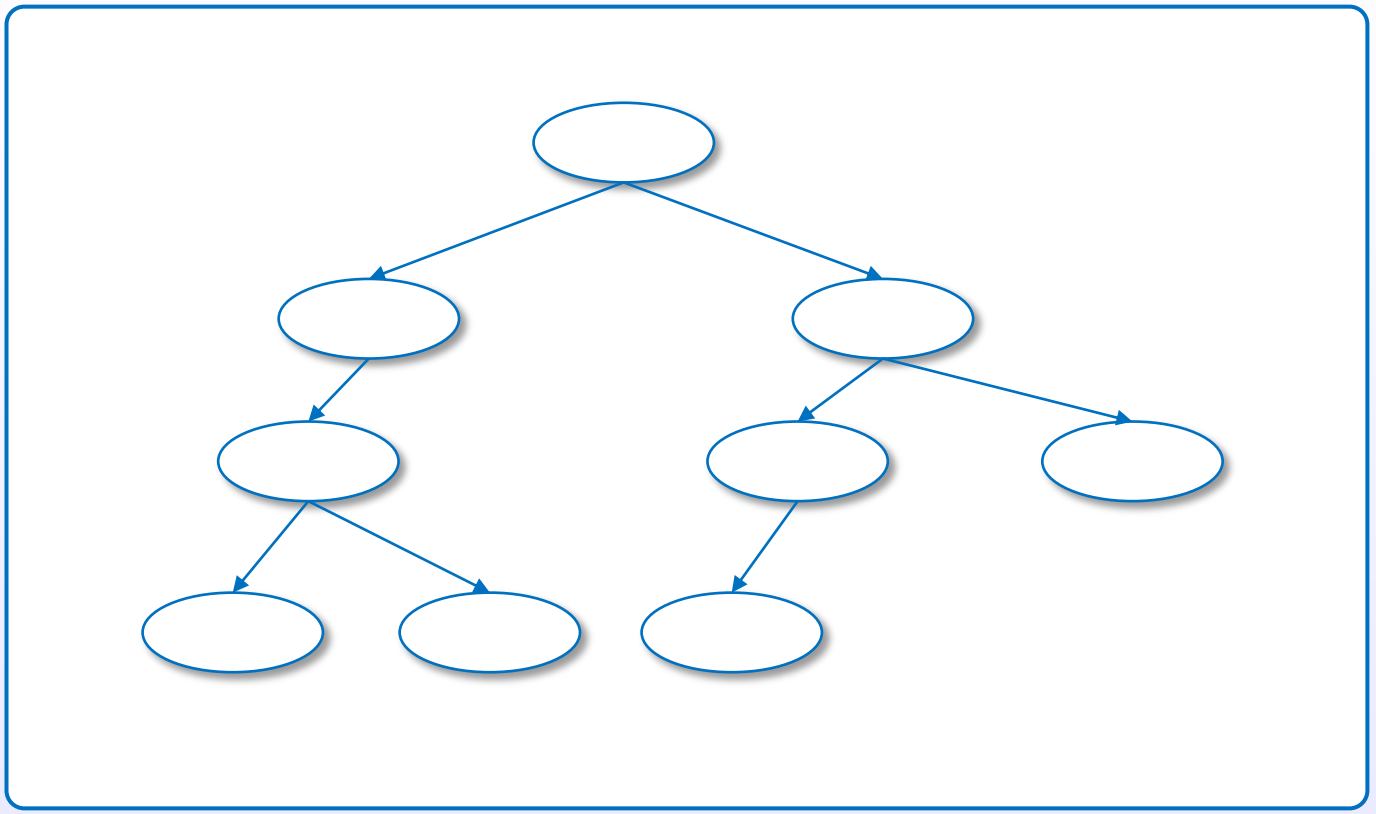
6. Implementation

6. Implementation

Graph reduction

Tree

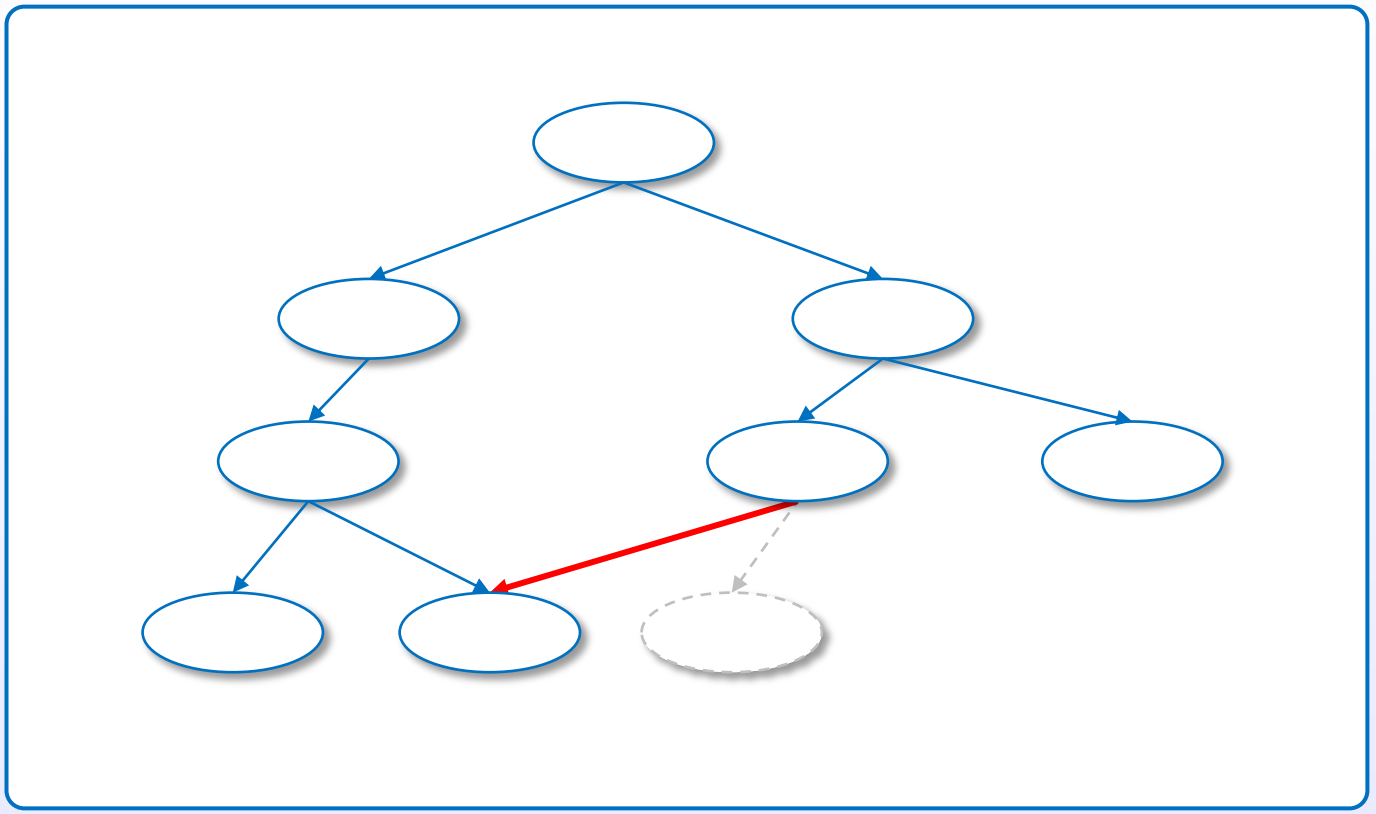
AST represents an expression



References : [1]

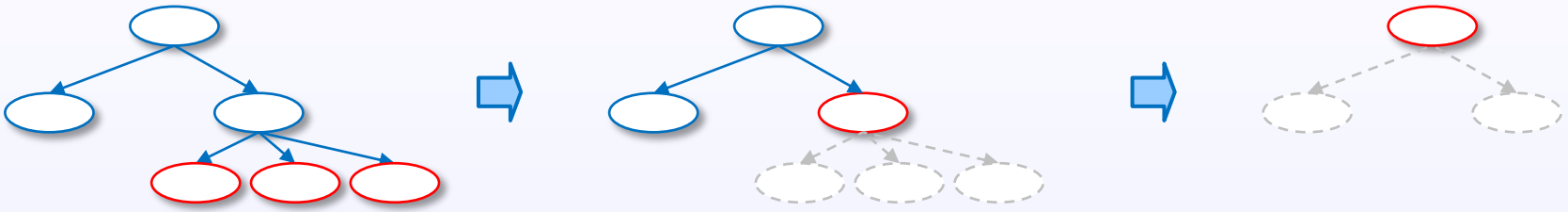
Graph

Share the term, looped
not Tree, but Graph



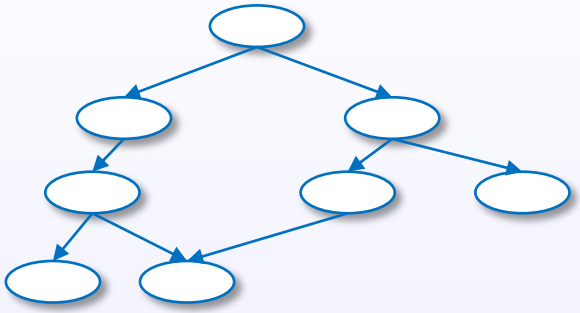
References : [1]

Graph reduction



References : [1]

Graph reduction and lazy



References : [1]

5. Implementation

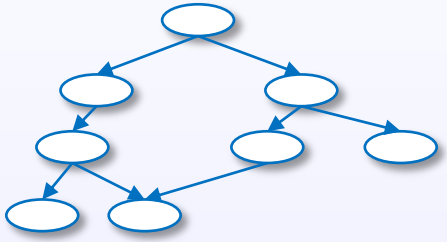
STG-machine

Layer

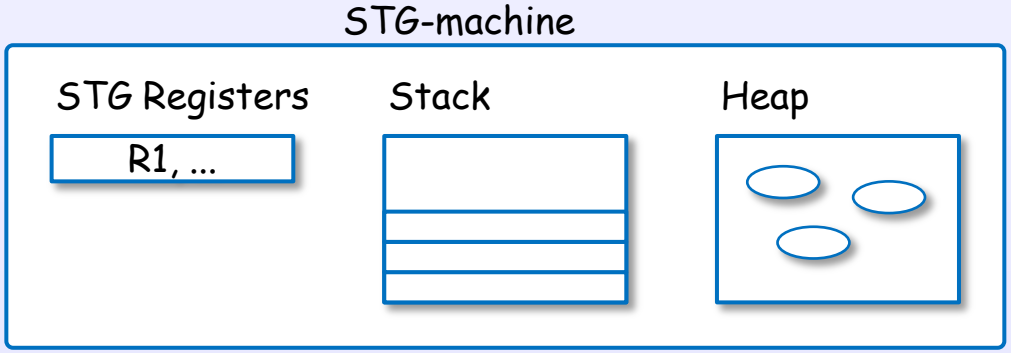
Haskell code

take 5 [1..10]

Internal representation
by graph

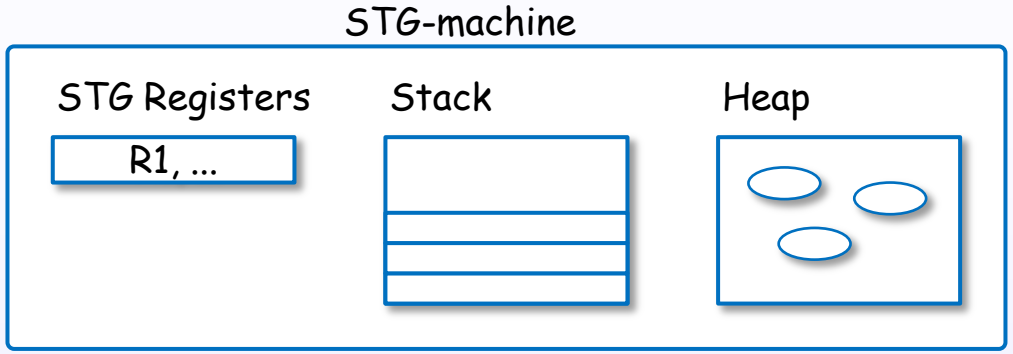


Evaluation (execution, reduction)
by STG-machine



References : [1]

STG-machine



STG-machine is abstraction machine
which is defined by operational semantics.

STG-machine efficiently performs lazy graph reduction.

References : [1]

7. Appendix

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] A History of Haskell: Being Lazy With Class
http://haskell.cs.yale.edu/wp-content/uploads/2011/02/history.pdf

[20] The implementation of functional programming languages
http://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/slpj-book-1987.pdf

[21] 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

[22] 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/

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

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

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

[26] Hoogle
https://www.haskell.org/hoogle/

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

[28] GHC illustrated
http://takenobu-hs.github.io/downloads/haskell_ghc_illustrated.pdf

Lazy,... zzz