Lazy evaluation illustrated

for Haskell divers

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
   - Lazy evaluation
   - Simple questions

2. Expressions
   - Expression and value
   - Expressions in Haskell
   - Classification by values and forms
   - WHNF

3. Internal representation of expressions
   - Constructor
   - Thunk
   - Uniform representation
   - WHNF
   - let, case expression

4. Evaluation
   - Evaluation strategies
   - Evaluation in Haskell (GHC)
   - Examples of evaluation steps
   - Controlling the evaluation

5. Implementation of evaluator
   - Lazy graph reduction
   - STG-machine

6. Semantics
   - Bottom
   - Non-strict Semantics
   - Strict analysis

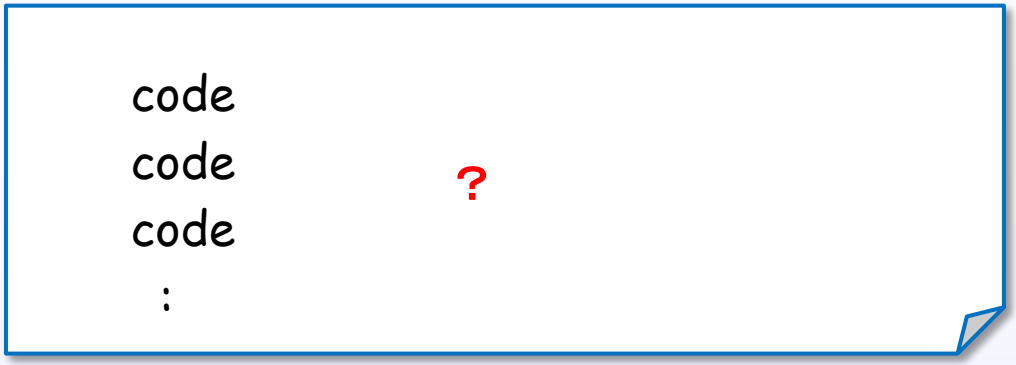
7. Appendix
   - References

1. Introduction

Basic mental models

How to evaluate a program in your brain ?

a program



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

What "mental model" do you have?

References : [1]

One of the mental models for C program

C program

A sequence of statements

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

?

A nested structure

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

?

A sequence of arguments

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

?

A function and arguments

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

?

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

What step, what order, ... ?

References : [1]

One of the mental models for C program

C program

A program is a collection of statements.

A sequence of statements

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

Statements are executed downward.

A nested structure

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

from inner to outer

A sequence of arguments

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

from left to right

A function and arguments

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

arguments first

apply second

Each 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) A program is a collection of statements.

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

(3) There is an order between completion and start of evaluations.

This is a syntactically straightforward model for programming languages.

(an implicit sequential order model)

References : [1]

One of the mental models for Haskell program

Haskell program

$$
\begin{aligned}
&\text{main} = \exp_{11}\ (\exp_{12}\ \exp_{13}\ \exp_{14}\ ) \\
&\exp_{13} = \exp_{131}\ \exp_{132} \\
&\exp_{14} = \exp_{141}\ \exp_{142}\ \exp_{143} \\
&\quad :
\end{aligned}
$$

?

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

What step, what order, ... ?

References : [1]

One of the mental models for Haskell program

Haskell program

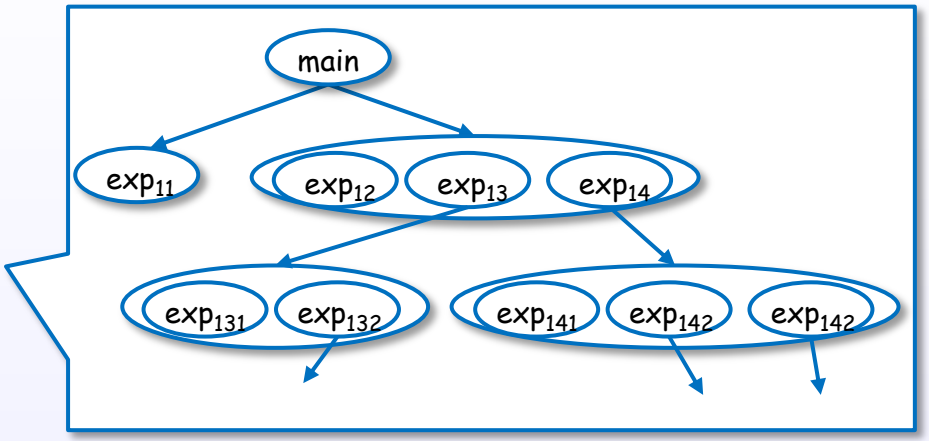
A program is a collection of expressions.

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

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

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

:



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

A entire program is regarded as a single expression.

The subexpression is evaluated (reduced) in some order.

The evaluation is performed by replacement.

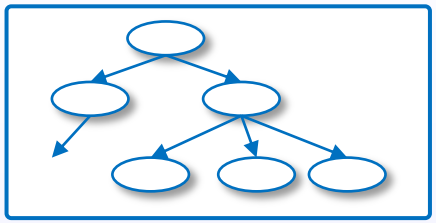
References : [1]

One of the mental models for Haskell program

(1) A program is a collection of expressions.

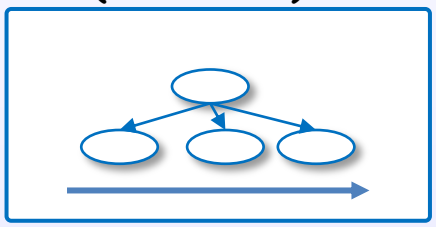
(2) A entire program is regarded as a single expression.

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

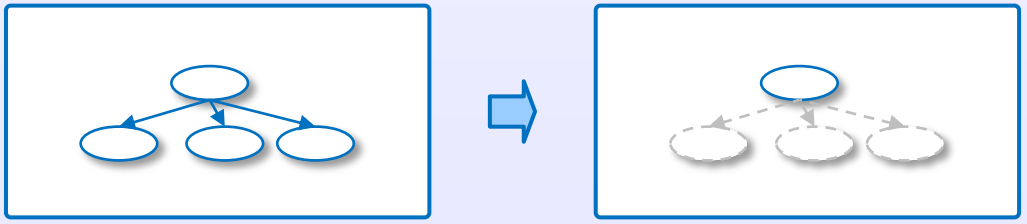


(3) The subexpressions are evaluated (reduced) in some order.

```
f = e (e (e (e e) e (e e e) ) )
```



(4) The evaluation is performed by replacement.

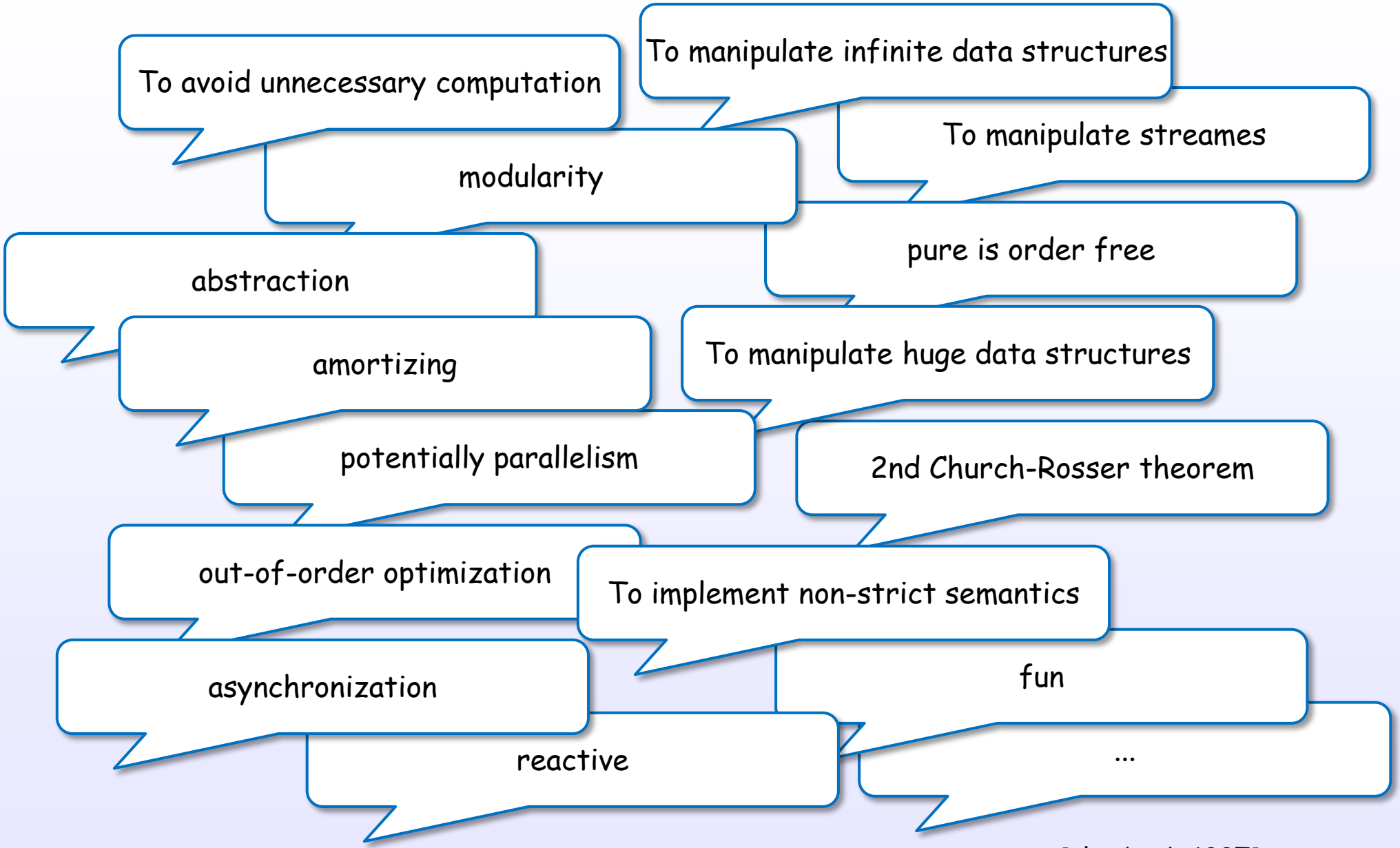


This is an example of an expression reduction model for Haskell.

References : [1]

Lazy evaluation

Why lazy evaluation?



There are various reason ☺

[slpj-book-1987]

References : [1]

Haskell(GHC) 's lazy evaluation

Lazy evaluation

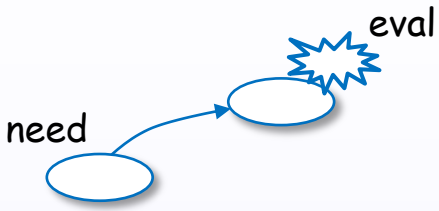
evaluate only if needed

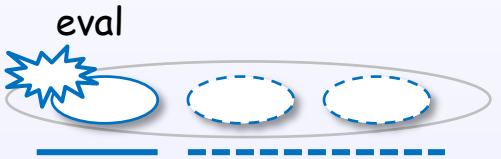
+

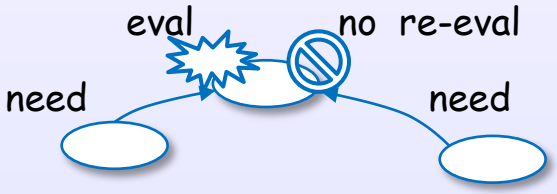
evaluate only enough

+

evaluate at most once







[slpj-book-1987], 194

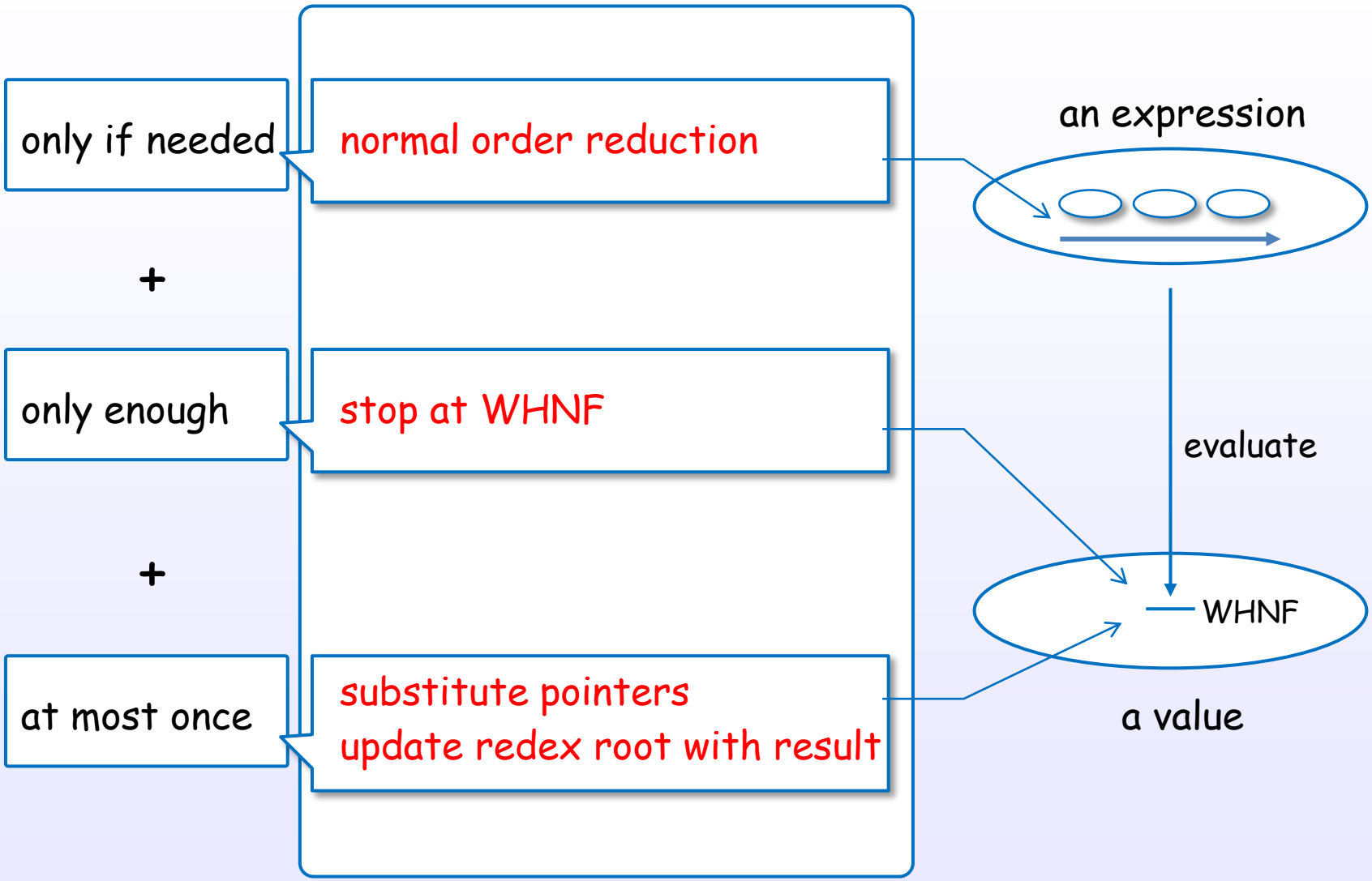
[slpj-book-1987], Chap.12

[Bird, Chap.7]

"Lazy" is "delay and avoidance" rather than "delay".

References : [1]

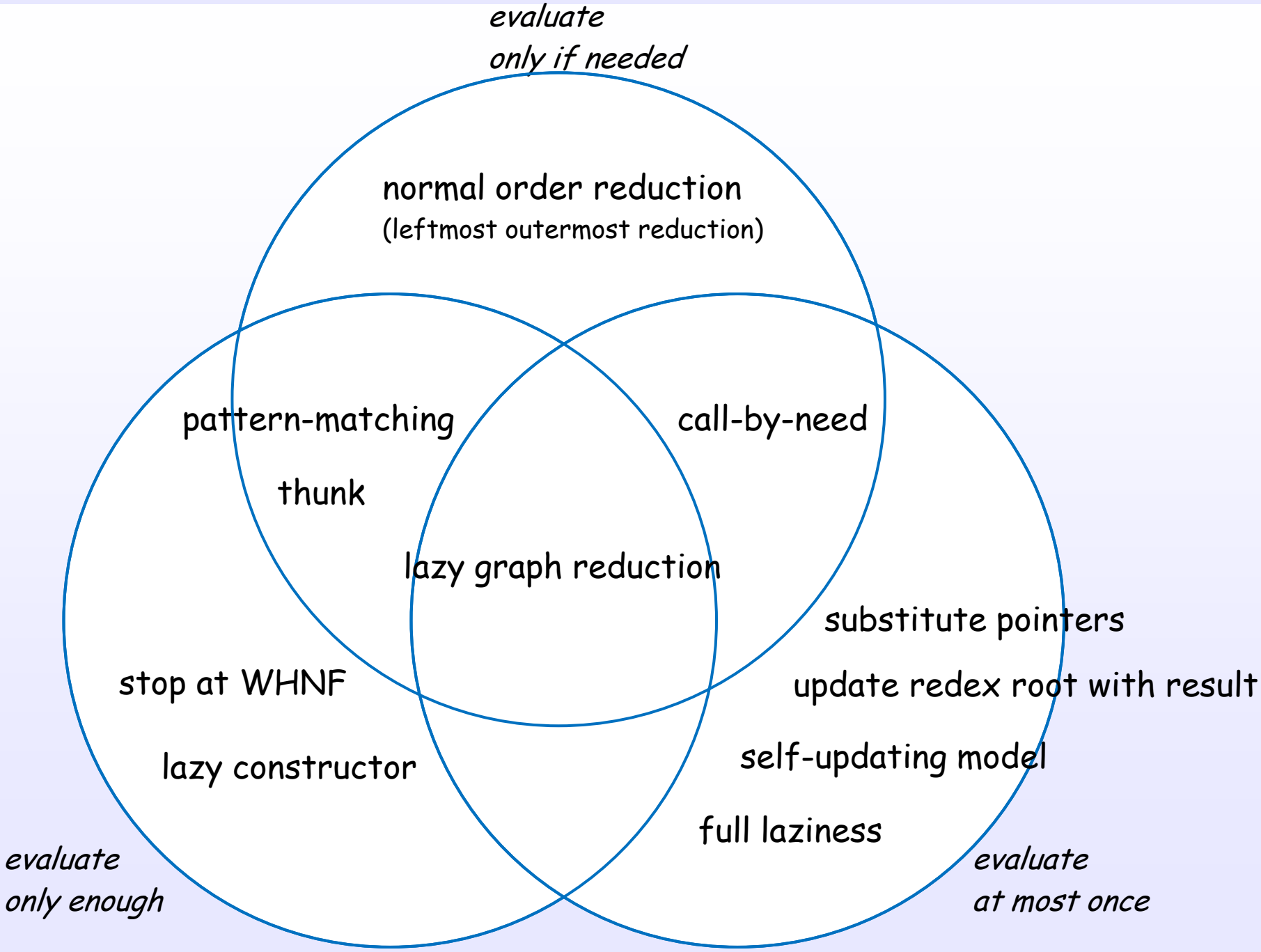
Ingredient of Haskell(GHC) 's lazy evaluation



This strategy is implemented by lazy graph reduction.

References : [1]

Techniques of Haskell(GHC) 's lazy evaluation

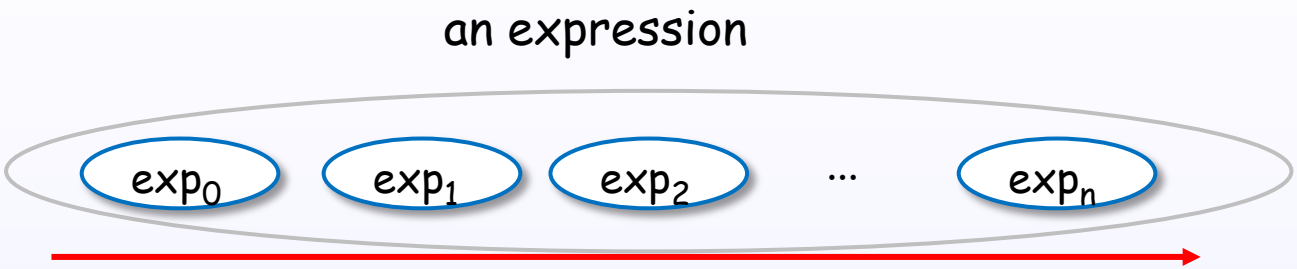


References : [1]

1. Introduction

Simple questions

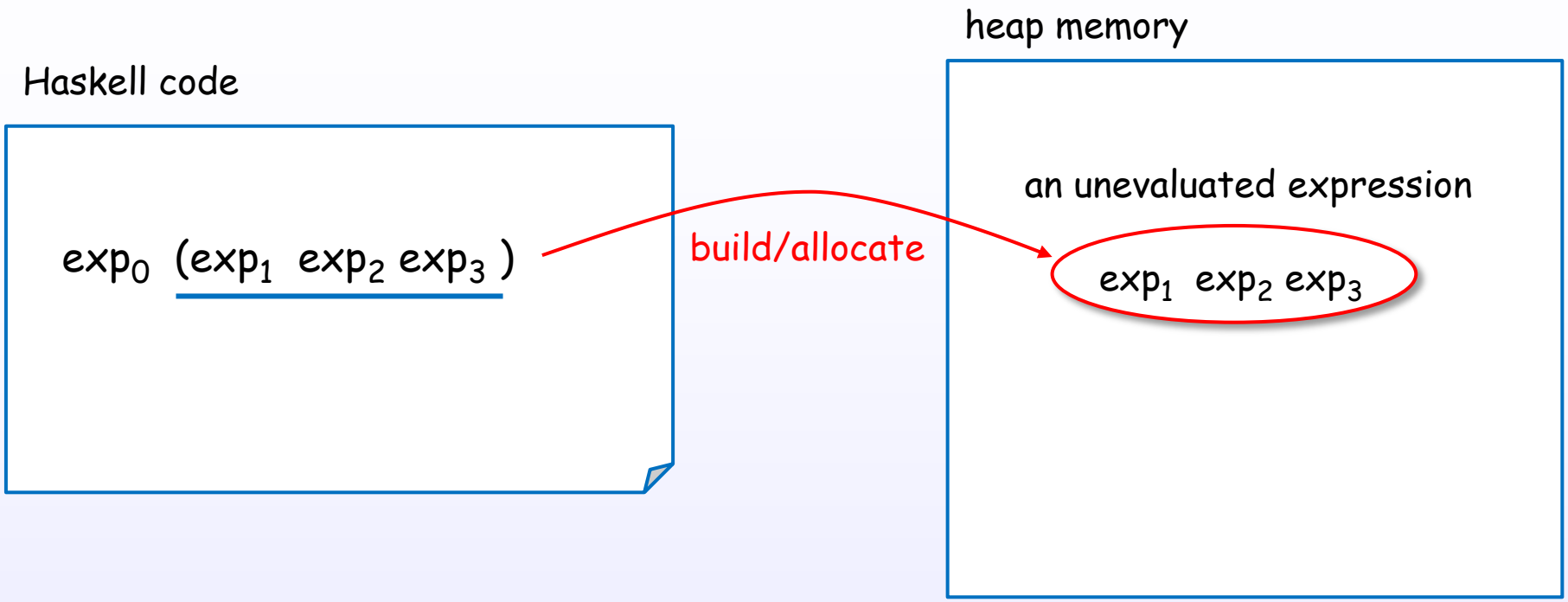
What order?



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

References : [1]

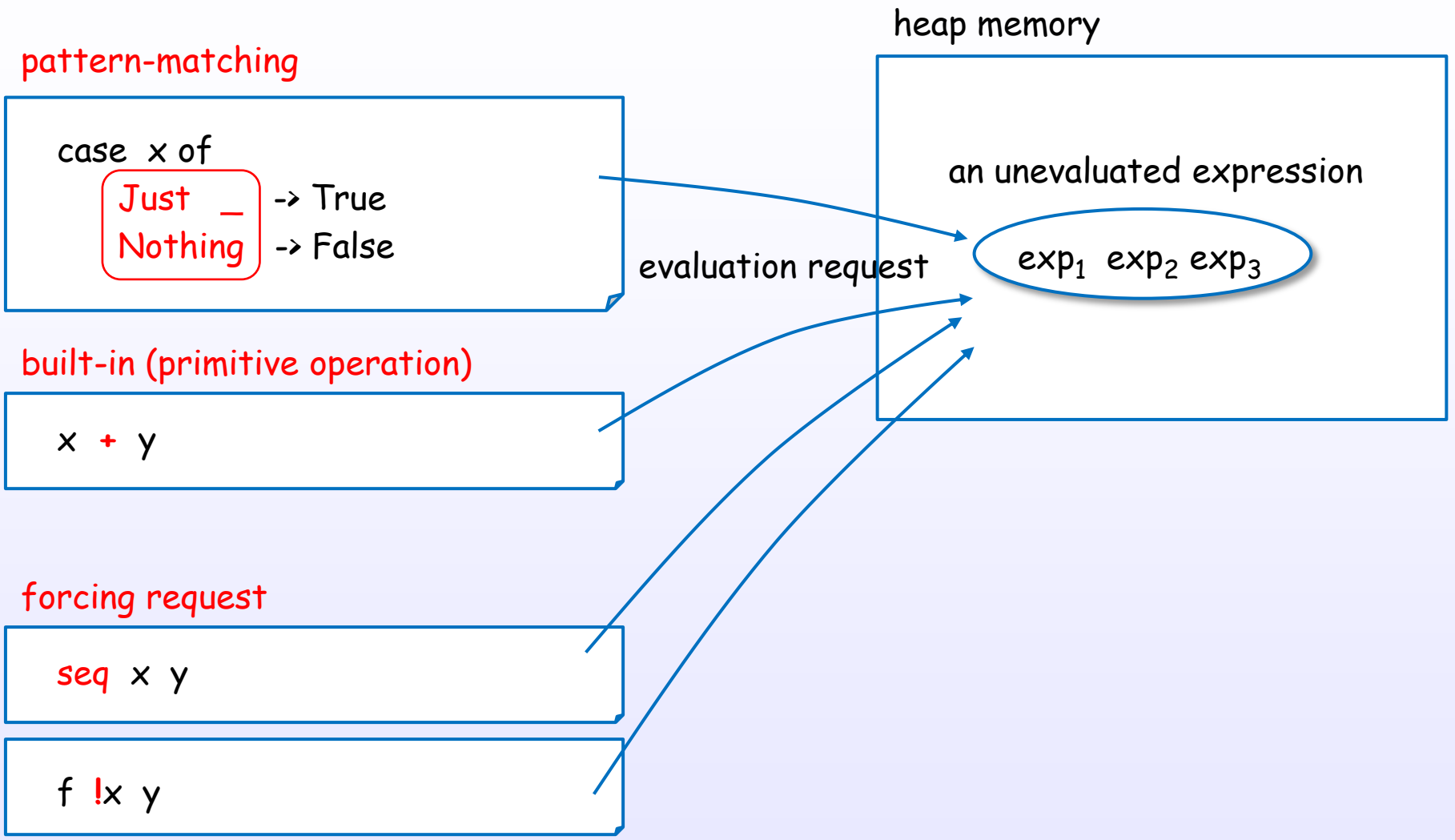
How to postpone?



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

References : [1]

When needed?

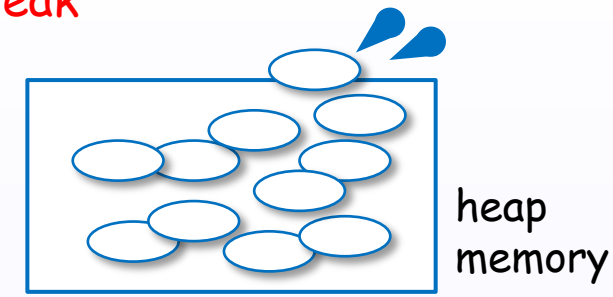


Pattern-matching or forcing request drive the evaluation.

References : [1]

What to be careful about?

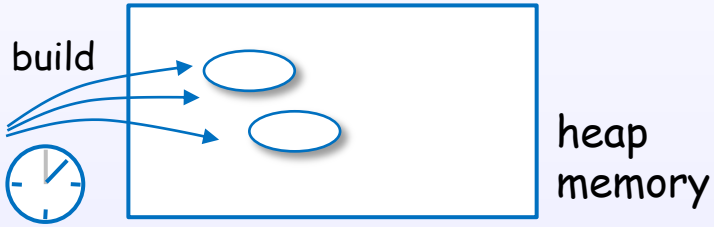
To consider hidden space leak



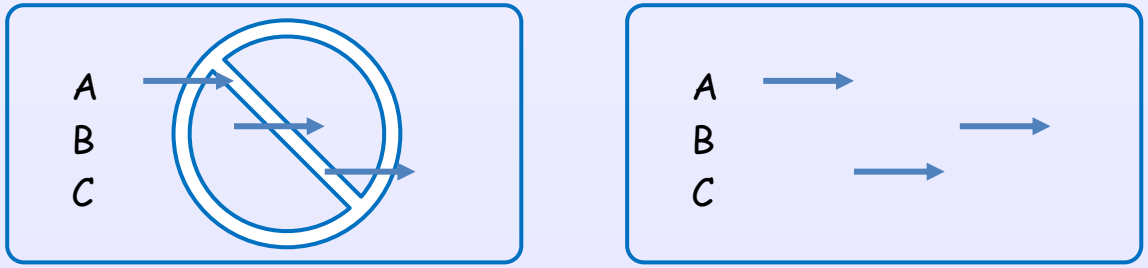
[hack.hands]

[CIS194]

To consider performance cost to postpone unevaluated expressions



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



You can avoid the pitfalls by controlling the evaluation.

References : [1]

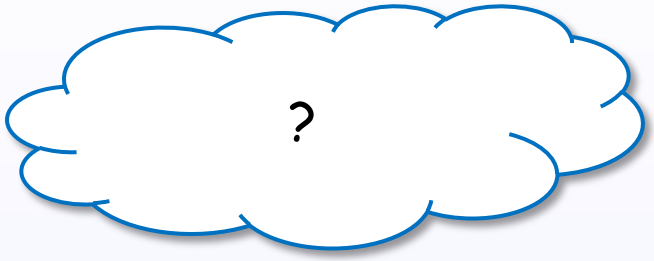
2. Expressions

2. Expressions

Expression and value

What is an expression?

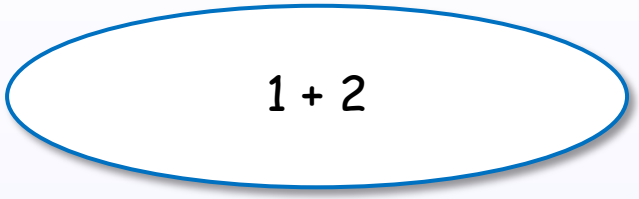
An expression



References : [1]

An expression denotes a value

An expression

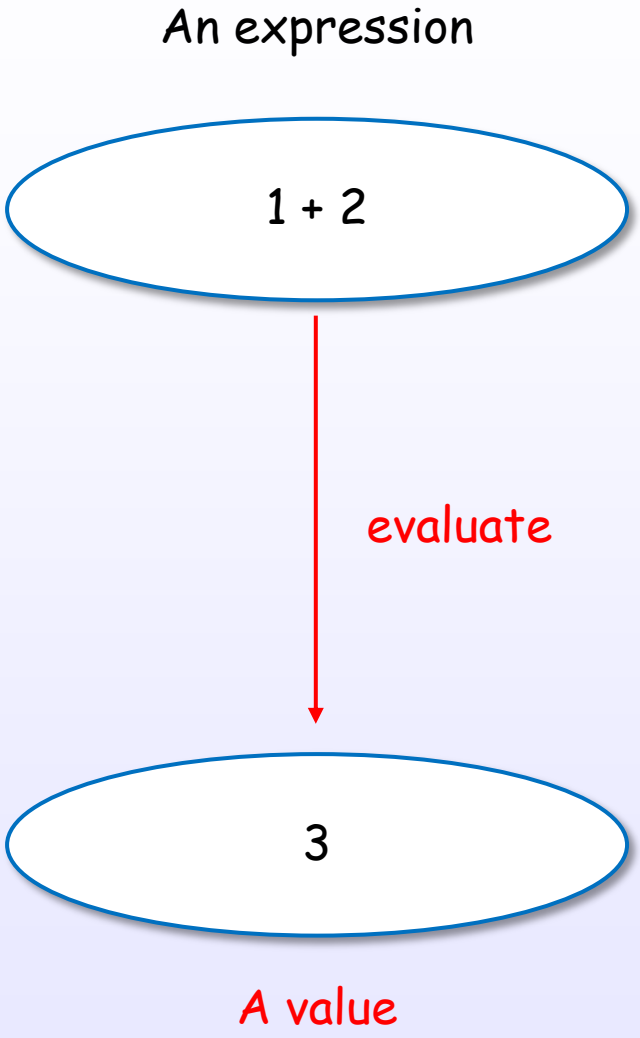


[HR2010]

[Bird, Chapter 2]

References : [1]

An expression evaluates to a value

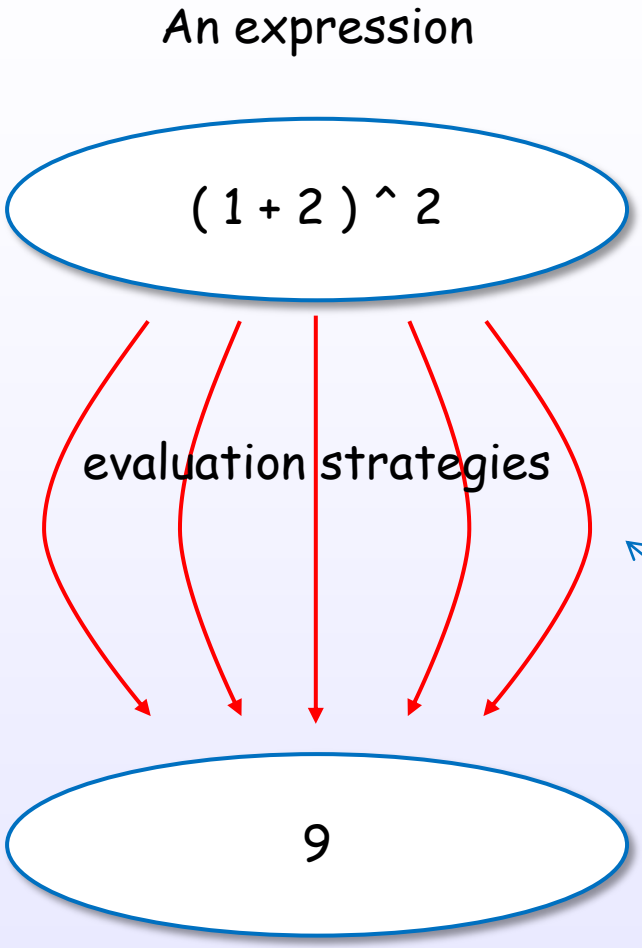


[HR2010]

[Bird, Chapter 2]

References : [1]

There are many evaluation approaches



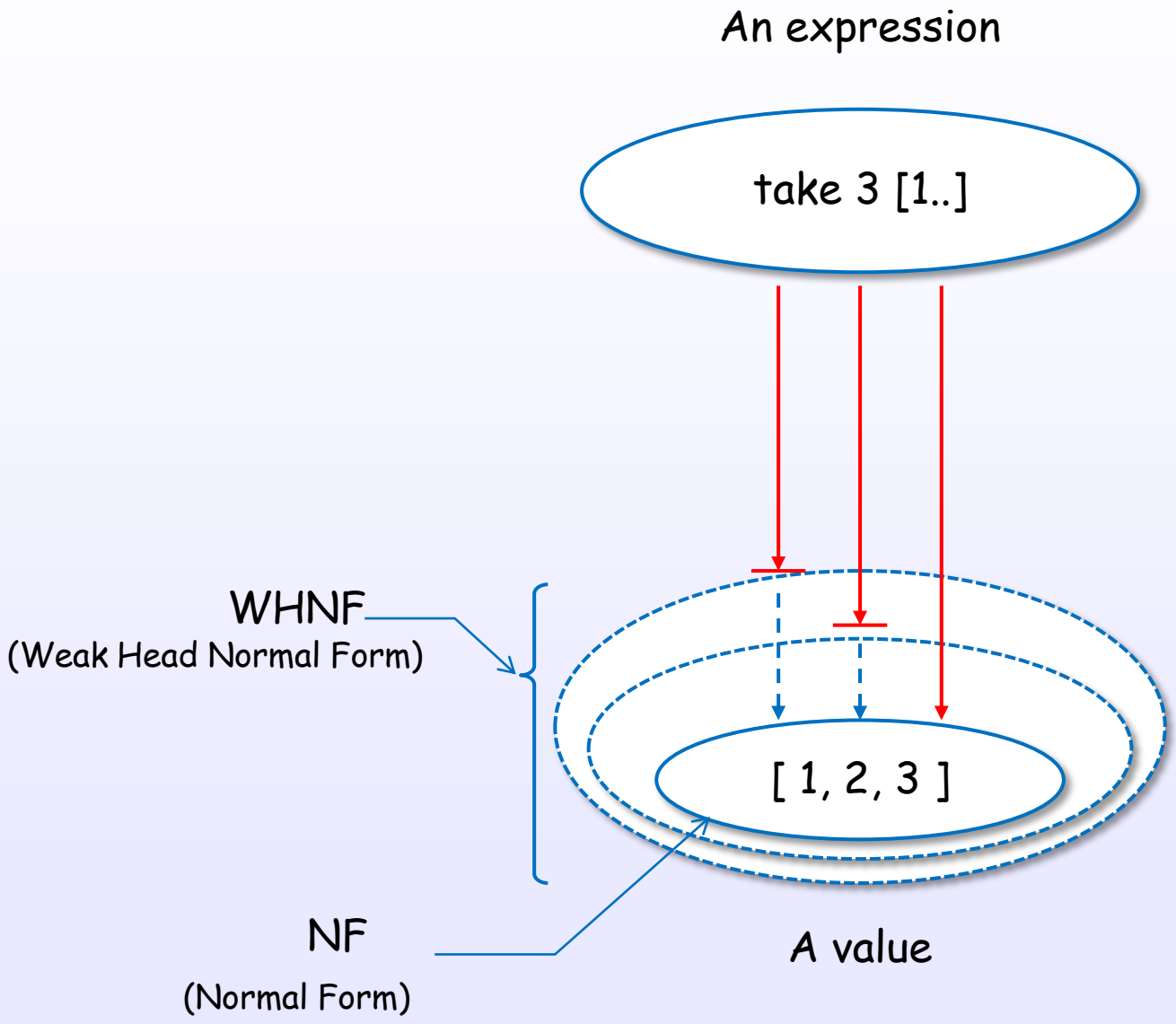
- Strict, Non-strict evaluation
- Eager, Lazy evaluation
- Call-by-value, Call-by-name, Call-by-need, …
- Innermost, Outermost
- Normal order, Applicative order
- …

[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

References : [1]

There are some evaluation levels



[Terei]

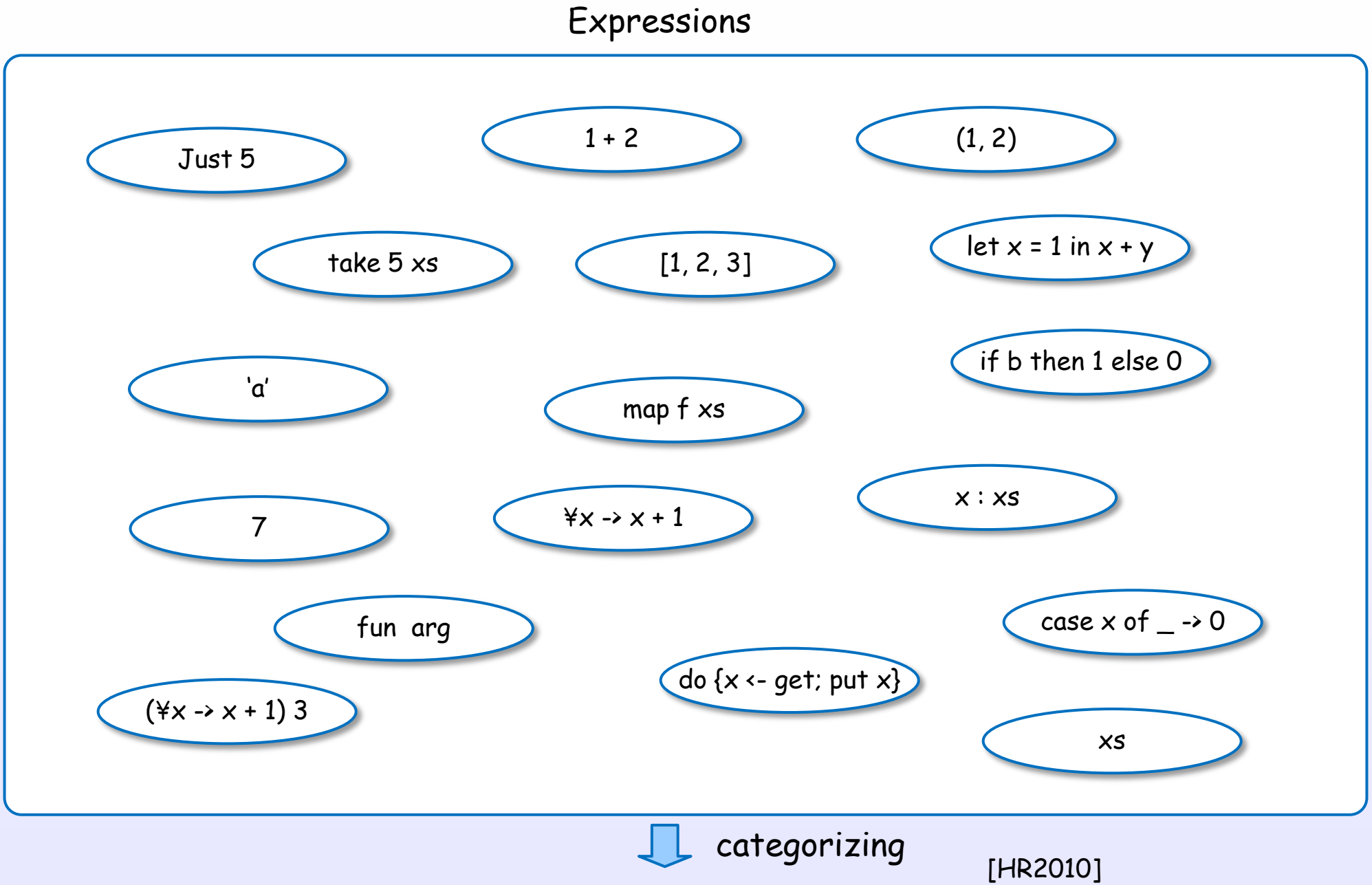
[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

References : [1]

Expressions in Haskell

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}

function application

take 5 xs

(¥x -> x + 1) 3

1 + 2

map f xs

fun arg

[HR2010]
[Bird, Chapter 2]

general constructor, literal and some forms

7

[1, 2, 3]

(1, 2)

'a'

x : xs

Just 5

variable

xs

References : [1]

Specification is defined in Haskell 2010 Language Report

"Haskell 2010 Language Report, Chapter 3 Expressions" [1]

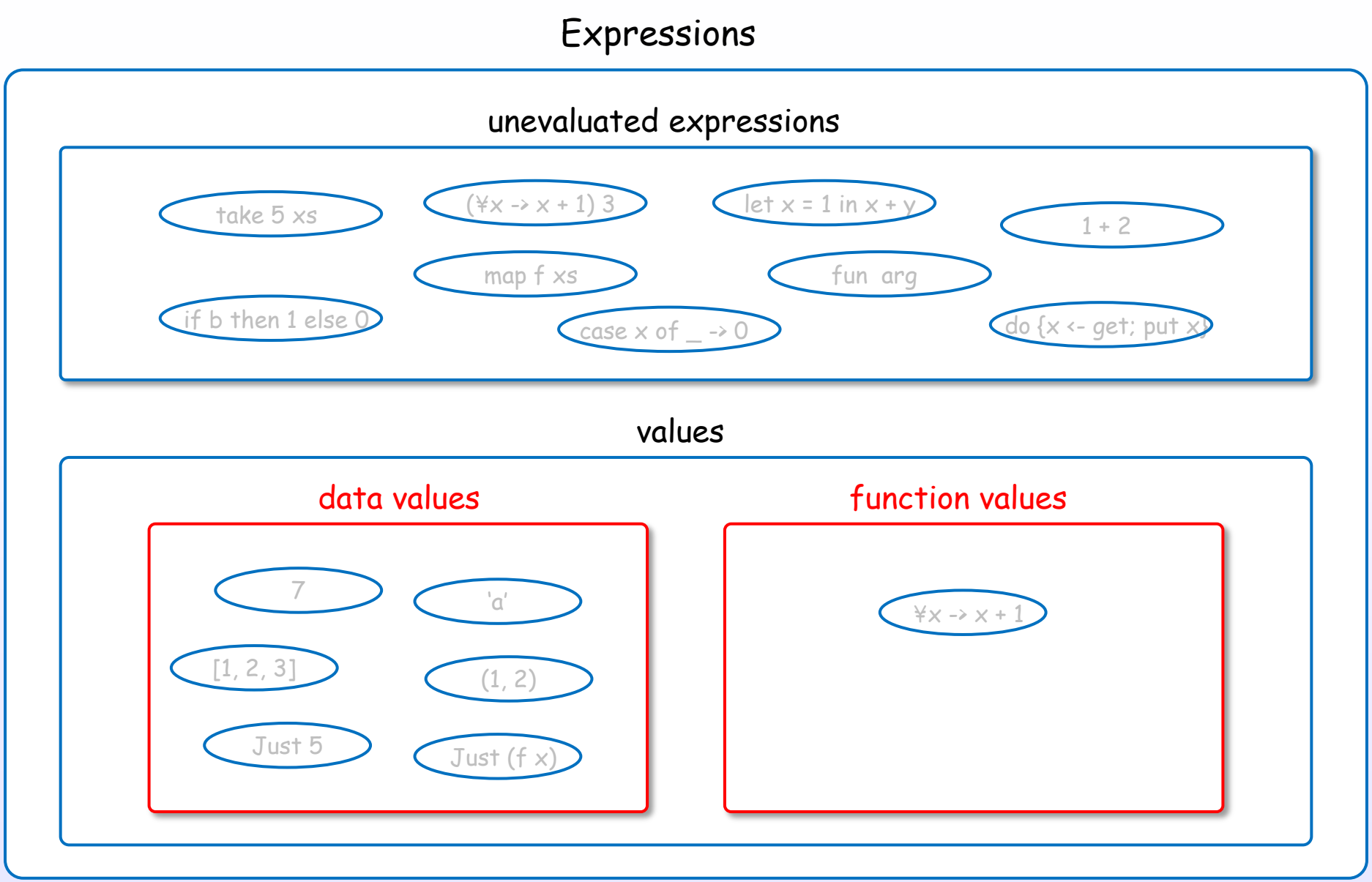
exp	→	*infixexp* `::` [*context* `=>`] *type*	(expression type signature)
	|	*infixexp*	
infixexp	→	*lexp qop infixexp*	(infix operator application)
	|	`-` *infixexp*	(prefix negation)
	|	*lexp*	
lexp	→	`\` $apat_1$ ... $apat_n$ `->` *exp*	(lambda abstraction, $n \geq 1$)
	|	`let` *decls* `in` *exp*	(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]

2. Expressions

Classification by values and forms

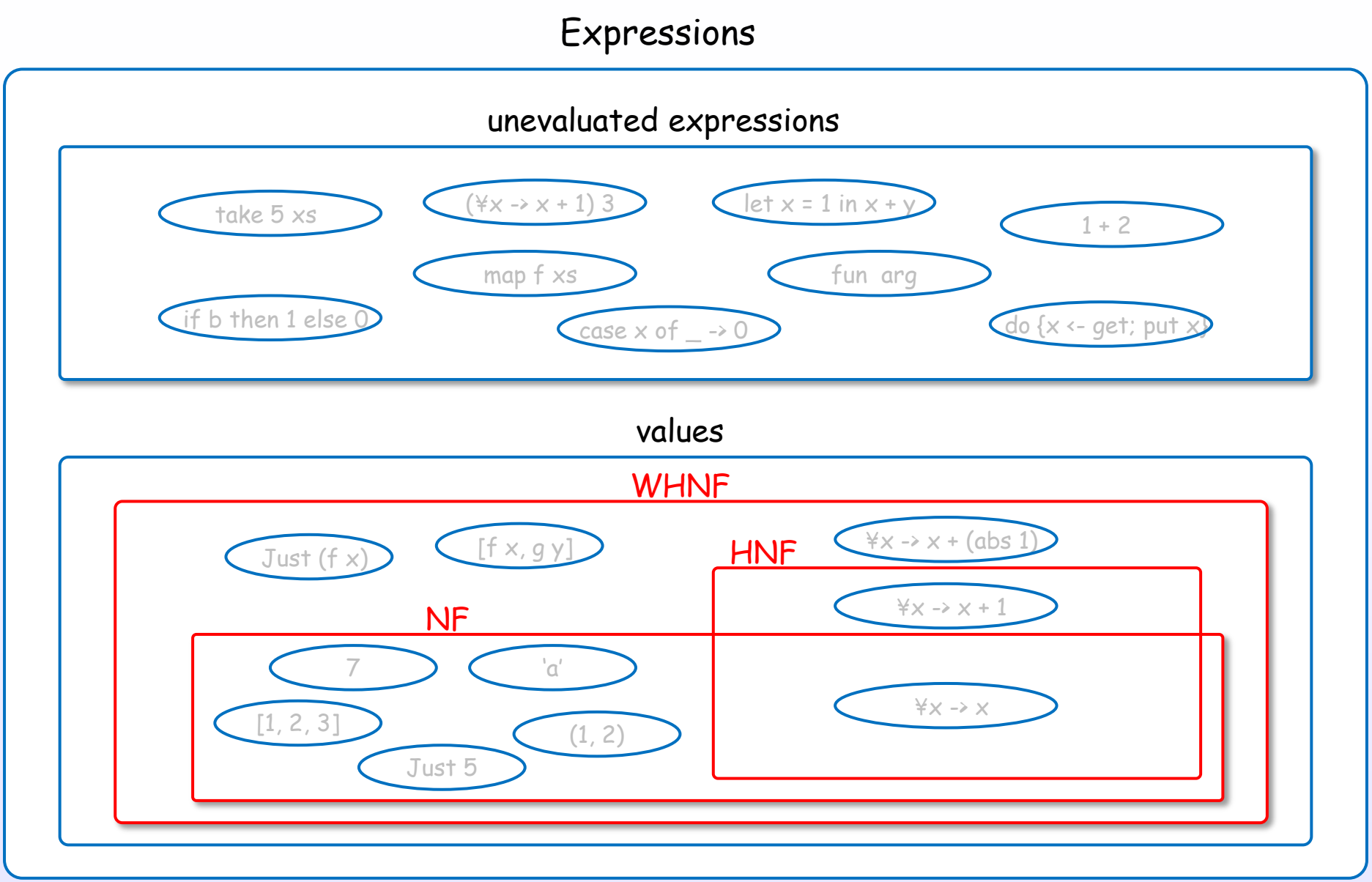
Classification by values



Values are data values or function values. [STG]

References : [1]

Classification by forms



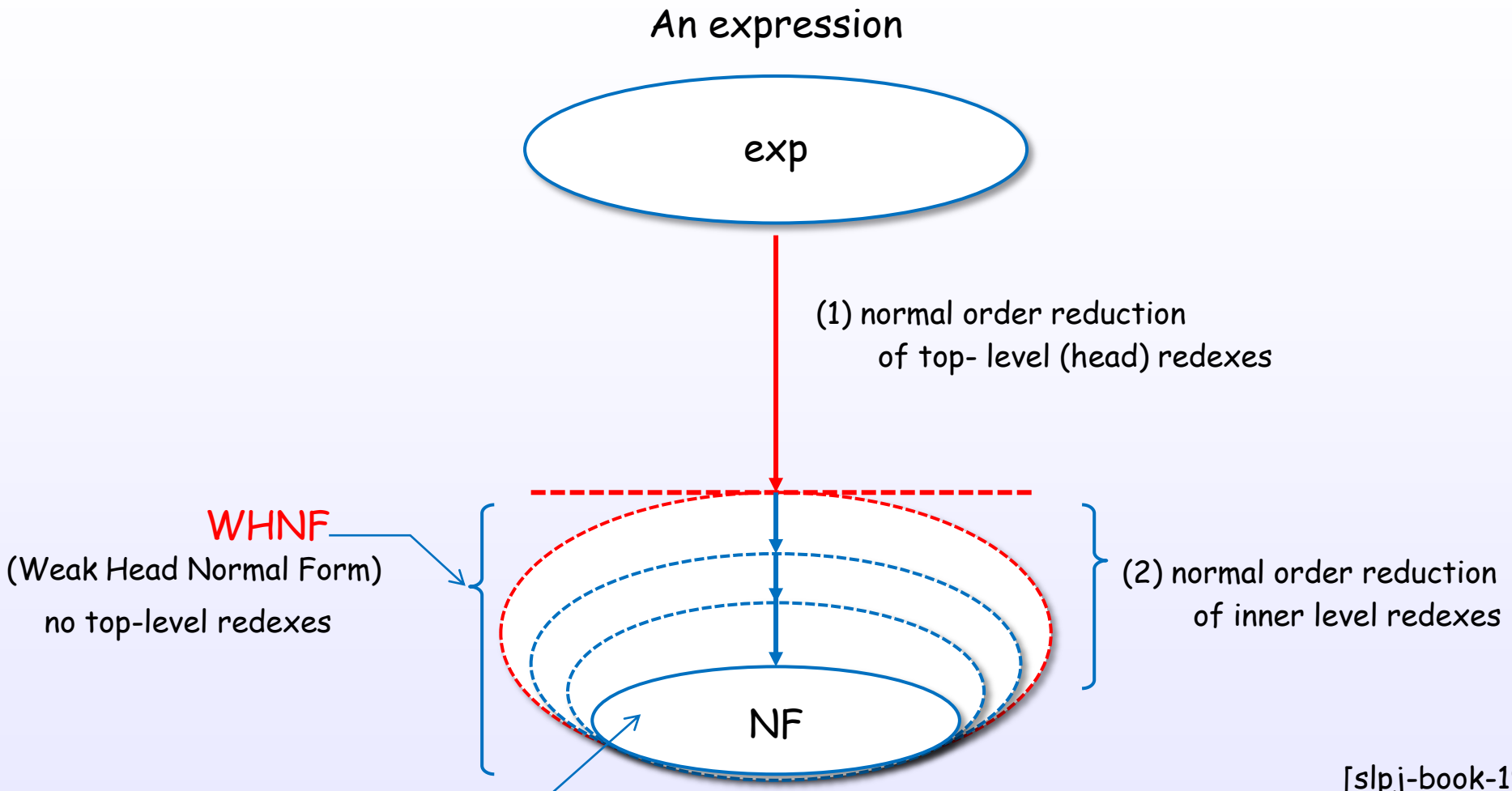
Values are NF, HNF or WHNF. [STG]

References : [1]

2. Expressions

WHNF

WHNF is one of the evaluated values.



[slpj-book-1987]

[Terei] [stephen]

[Bird, Chapter 2, 7]

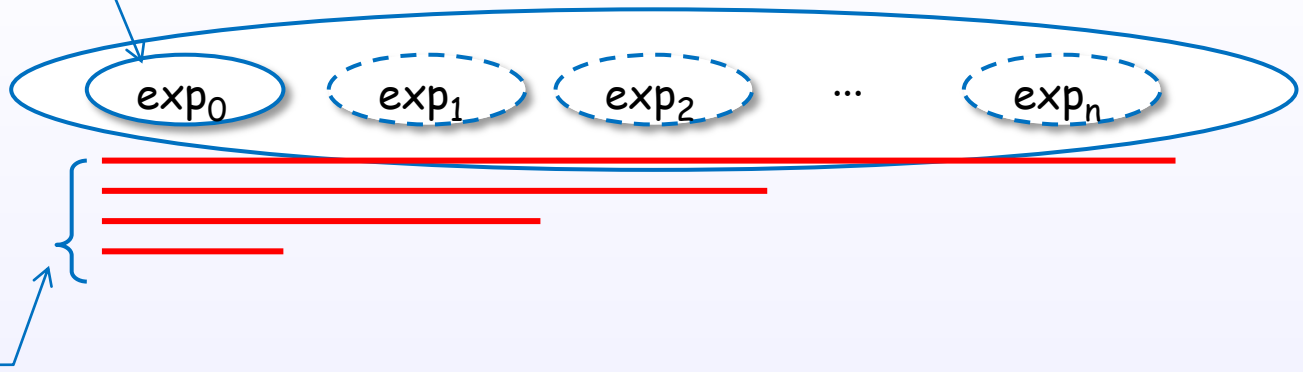
[TAPL, Chapter 3]

[Haskell/Laziness]

References : [1]

WHNF

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



WHNF is a value which has evaluated top-level

[Terei]
[Bird, Chapter 2, 7]
[TAPL, Chapter 3]

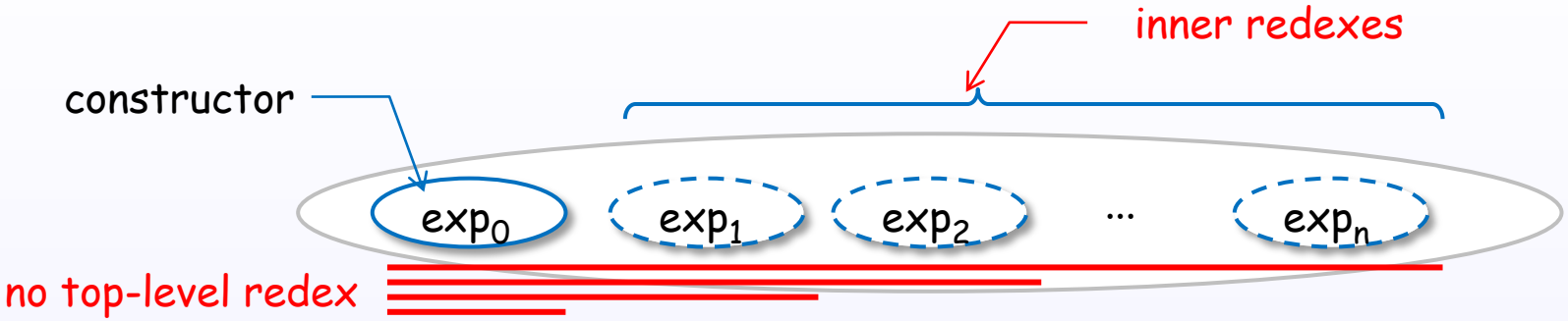
[parconc, Ch.2]
[slpj-book-1987]

[stephen]
[hack.hands]
[Terei]

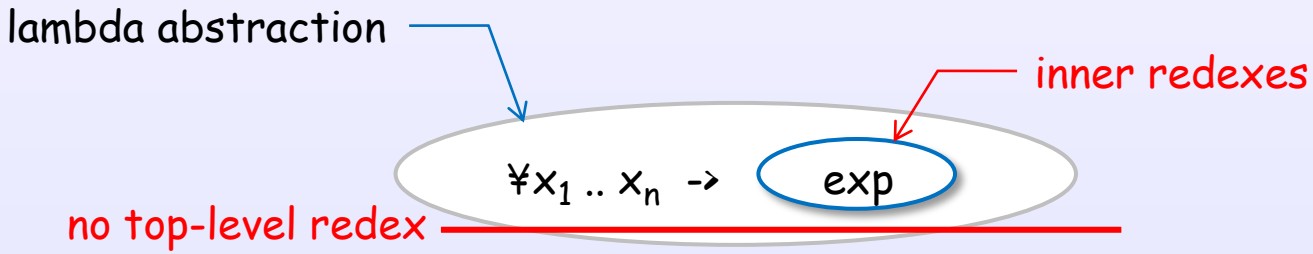
References : [1]

WHNF for a data value and a function value

a data value in WHNF



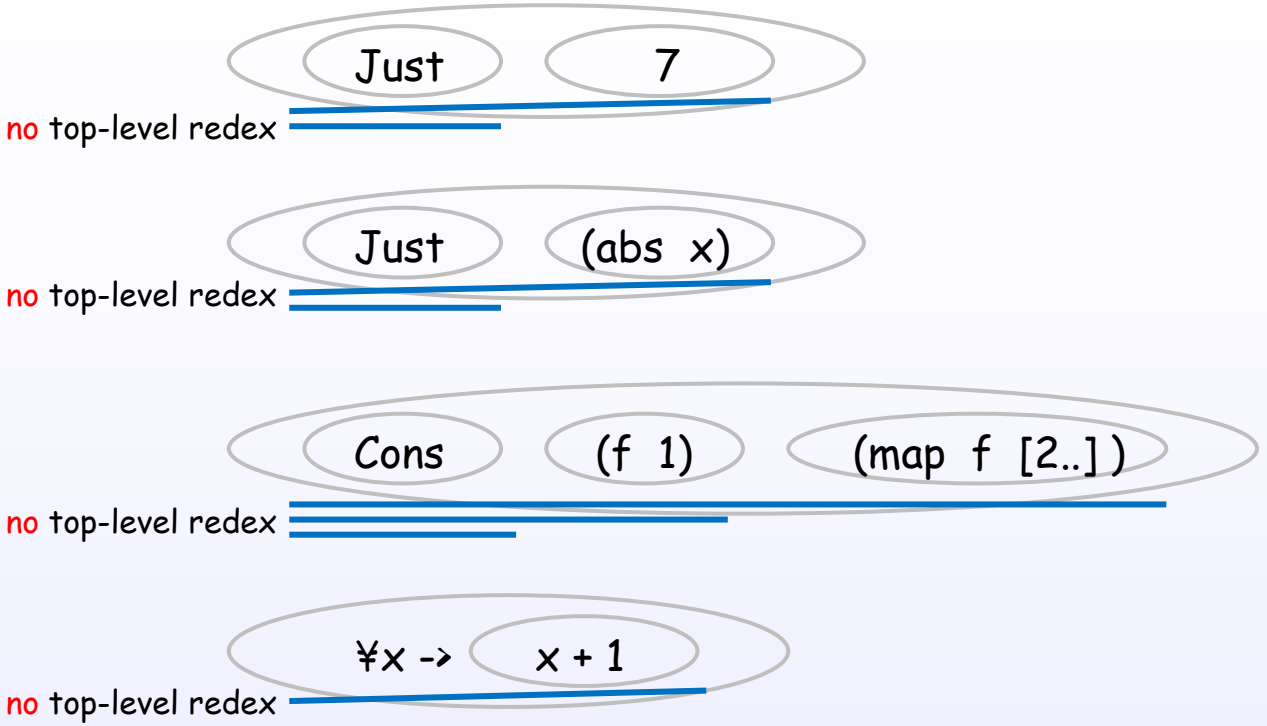
a function value in WHNF



References : [1]

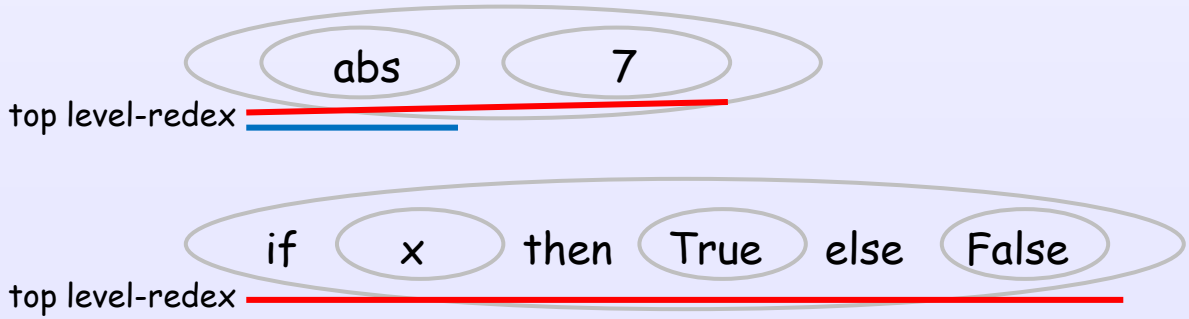
Examples of WHNF

WHNF



---

no WHNF

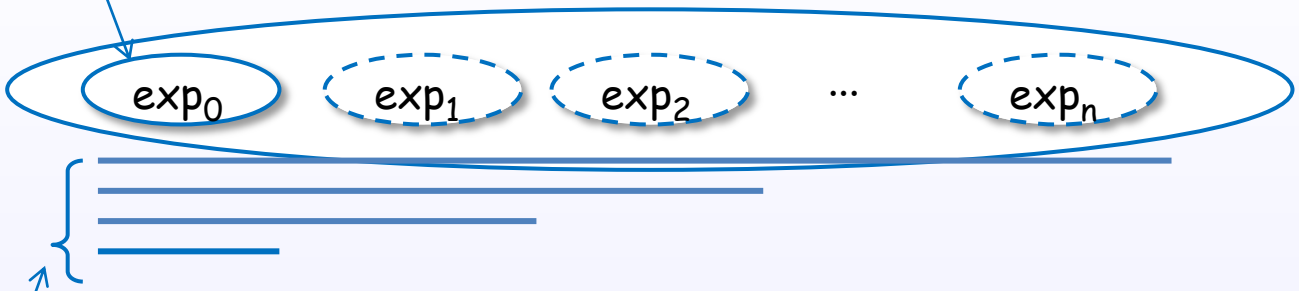


[slpj-book-1987]

References : [1]

HNF

top-level (head) is
a constructor or
a lambda abstraction with no top-level redex



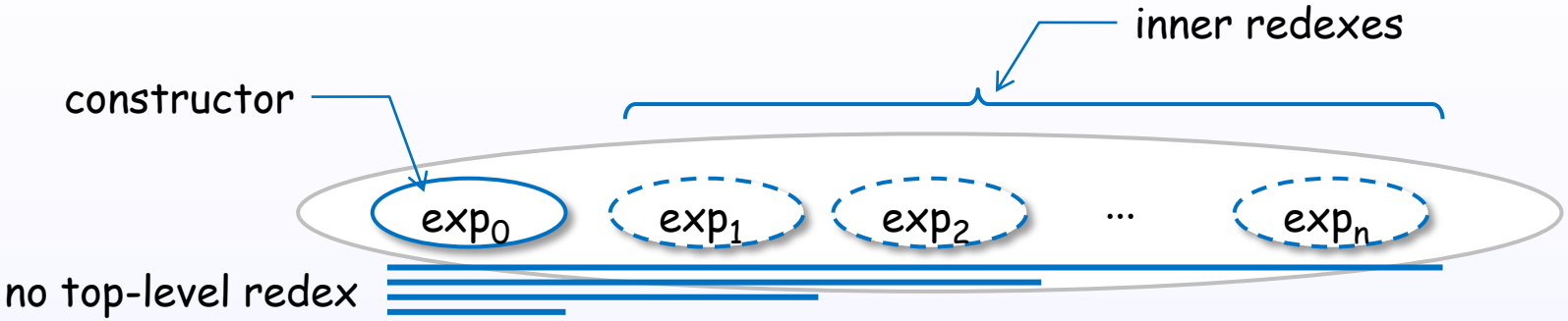
no top-level redex

* GHC uses WHNF rather than HNF.

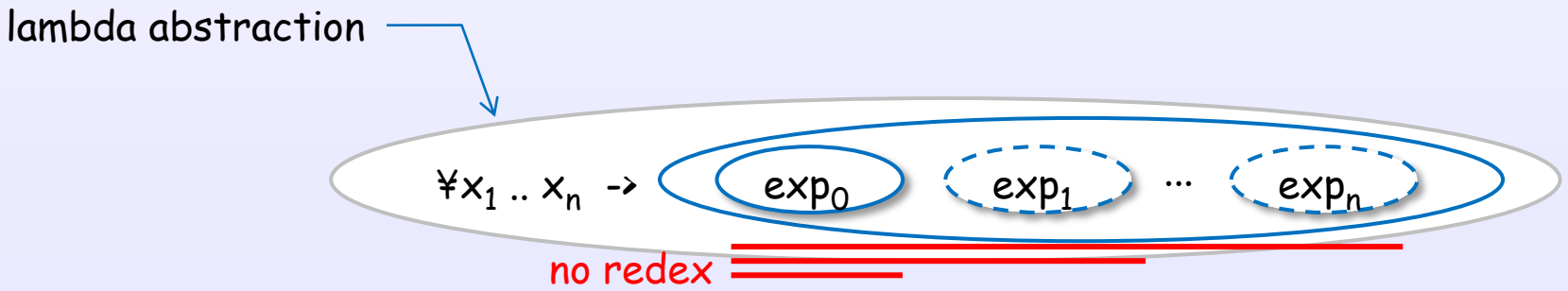
References : [1]

HNF for a data value and a function value

a data value in HNF (same as WHNF)



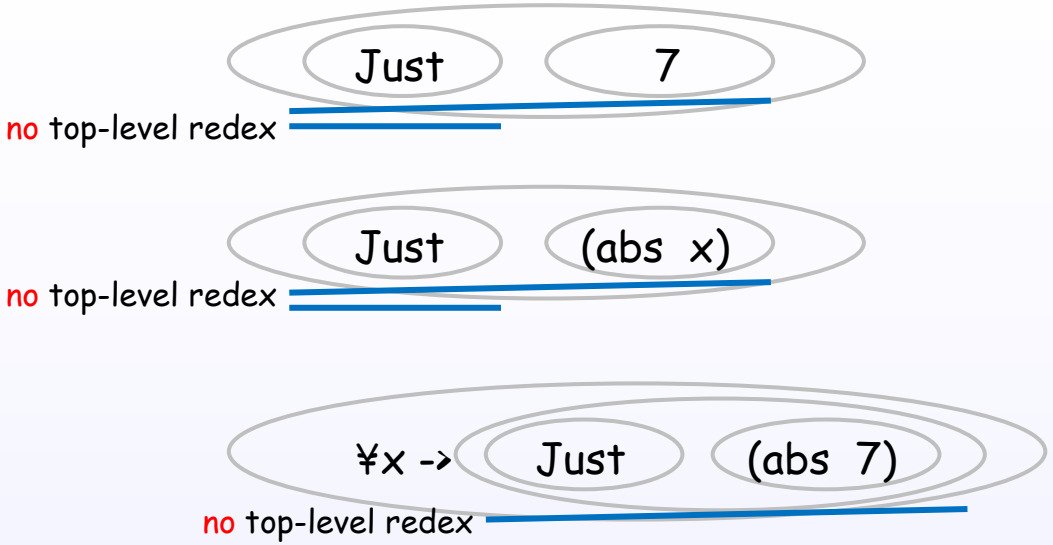
a function value in HNF



References : [1]

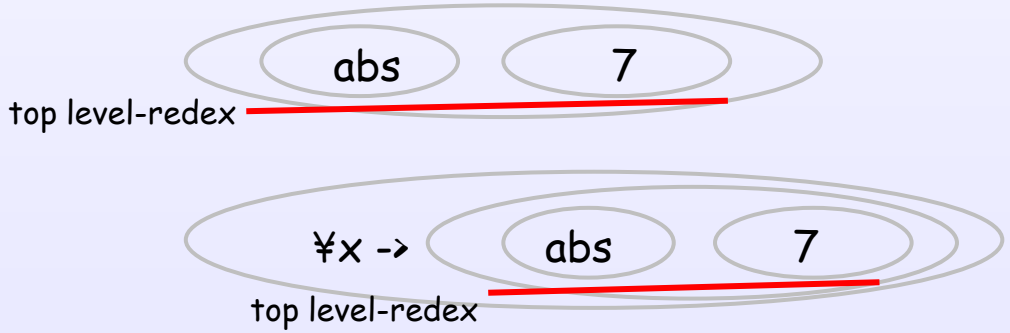
Examples of HNF

HNF



---

no HNF

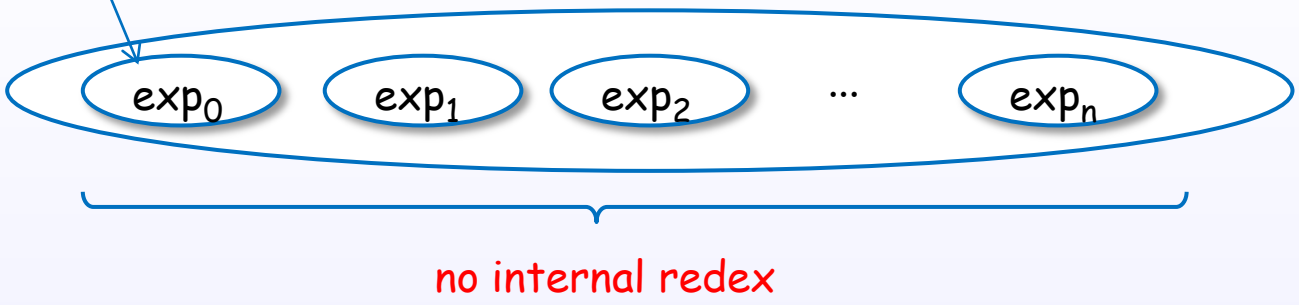


[slpj-book-1987]

References : [1]

NF

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



[slpj-book-1987]

[Terei]

[Bird, Chapter 2, 7]

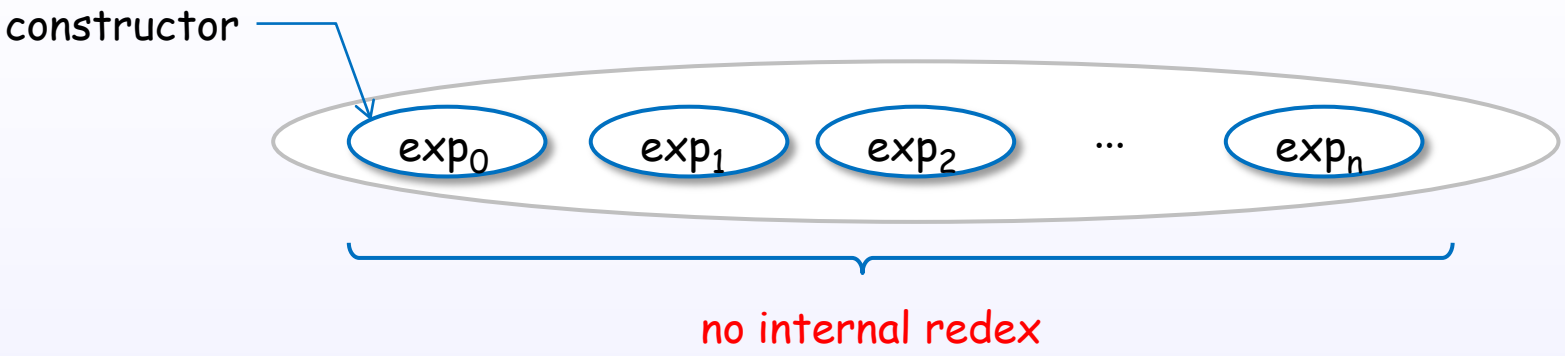
[TAPL, Chapter 3]

[Terei]

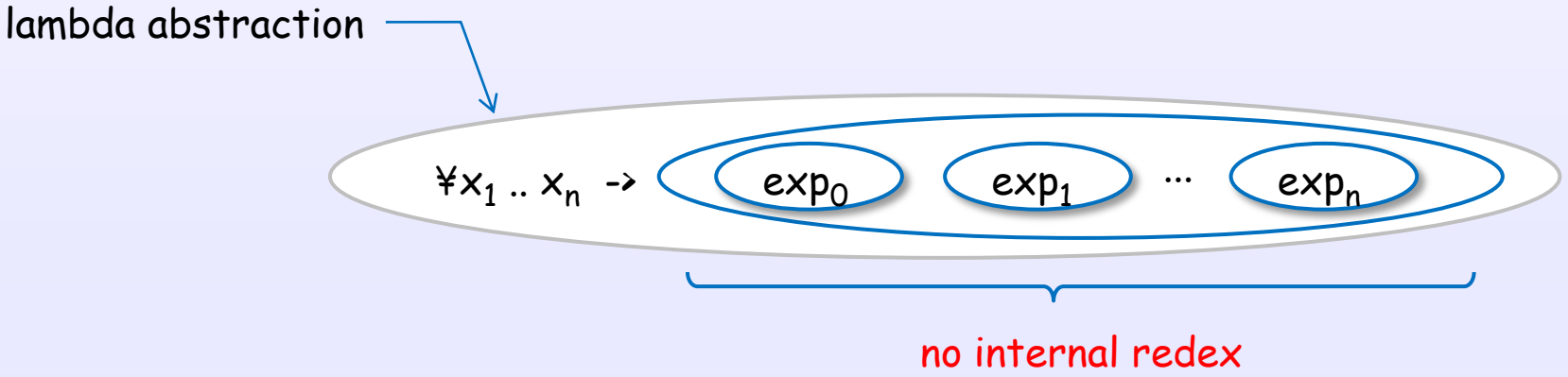
References : [1]

NF for a data value and a function value

a data value in NF



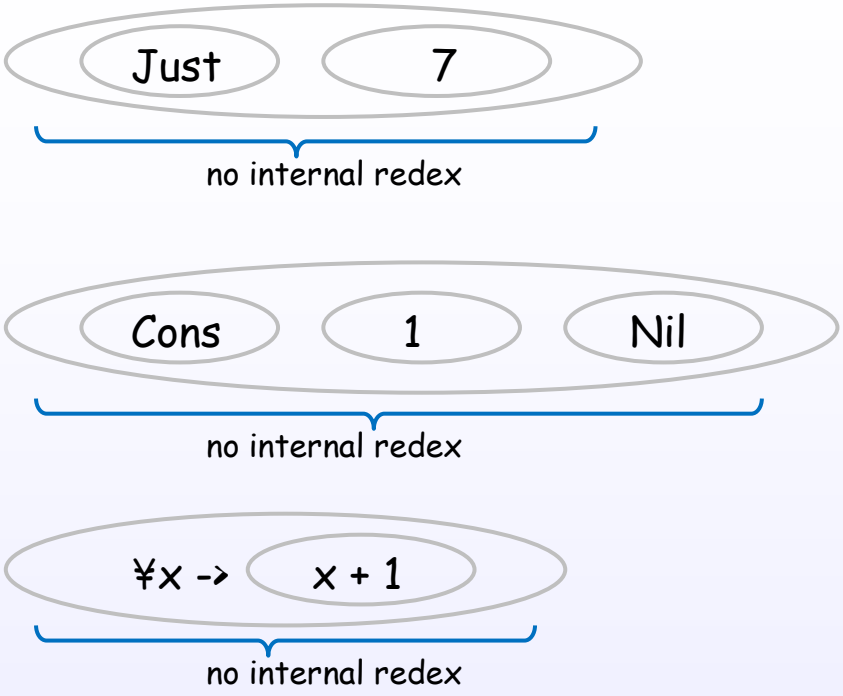
a function value in NF



References : [1]

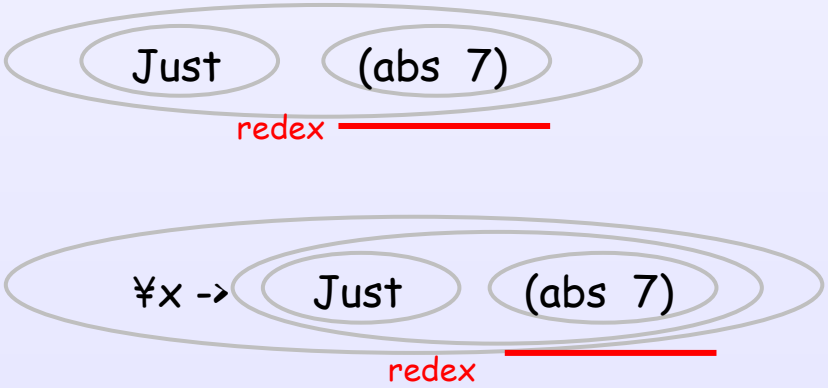
Examples of NF

NF



---

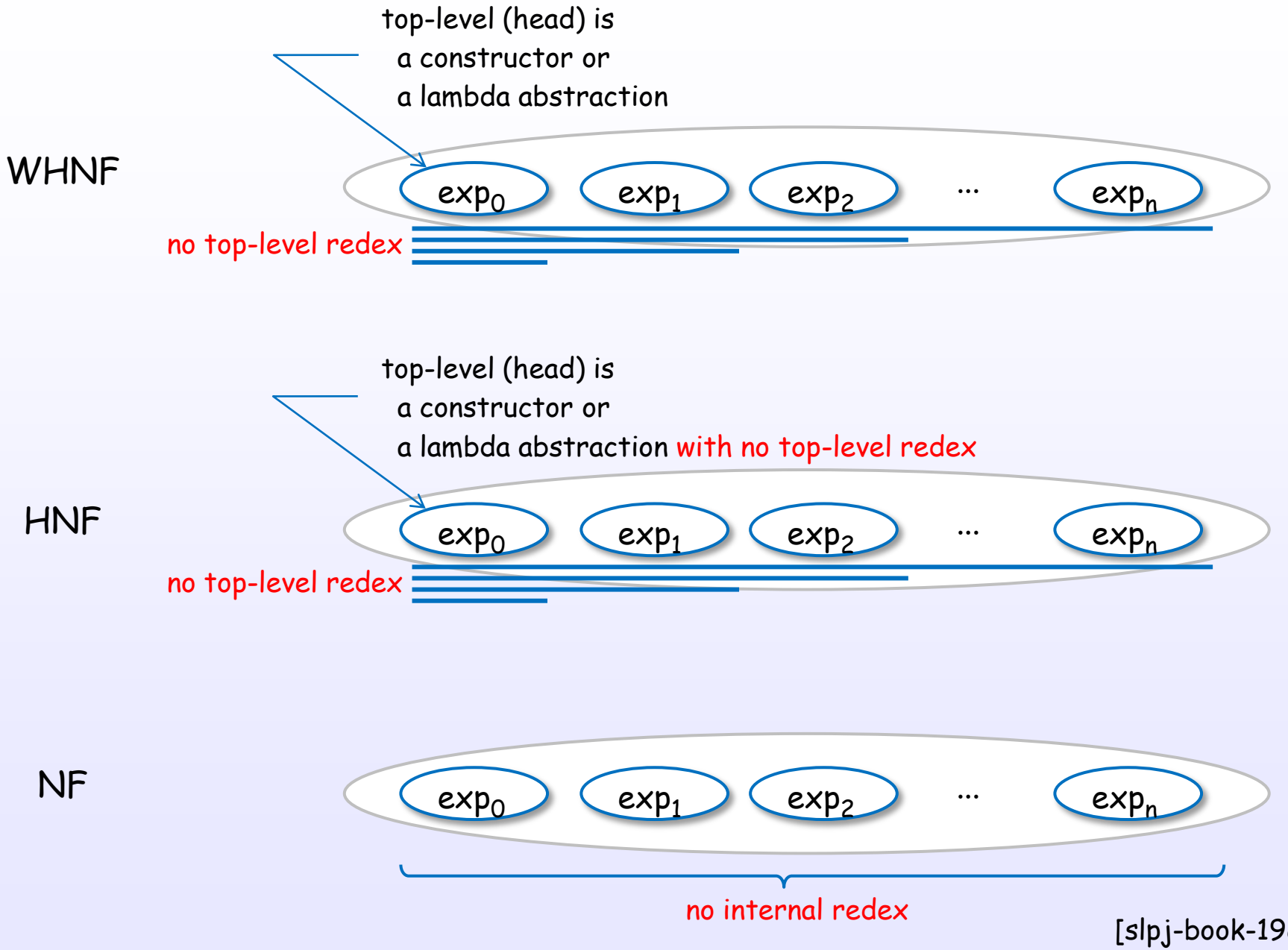
no NF



[slpj-book-1987]

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 confused
some discussion. The content of th
since for most purposes head norma
form. Nevertheless, we will stick to 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]

3. Internal representation of expressions

3. Internal representation of expressions

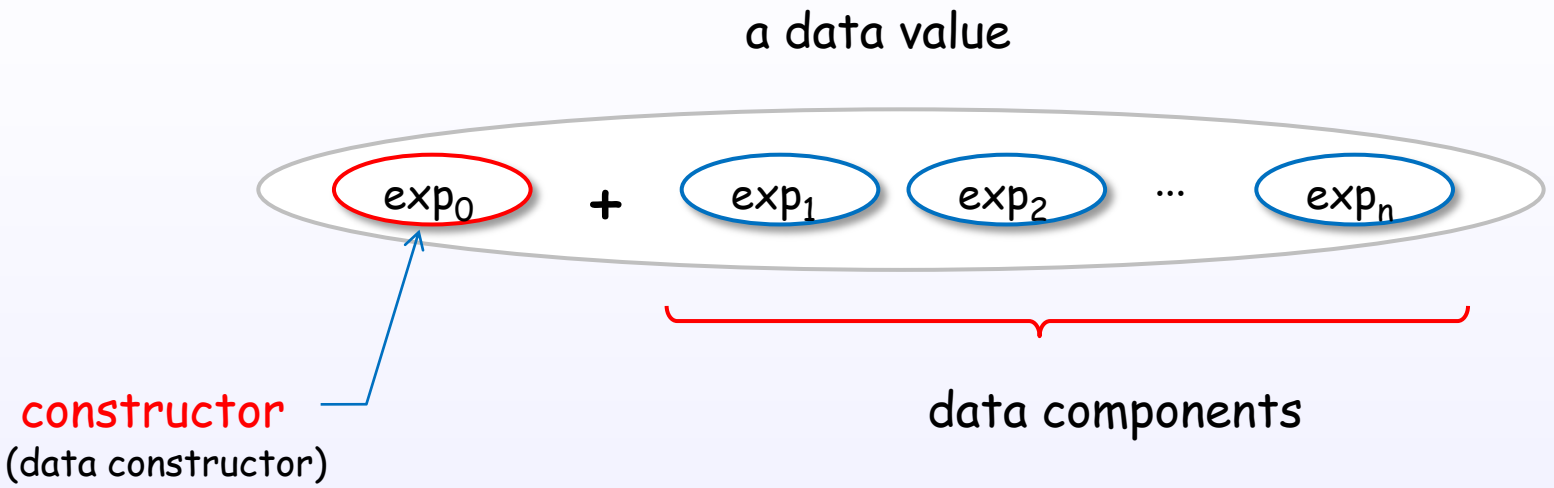
Constructor

Constructor

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

References : [1]

Constructor

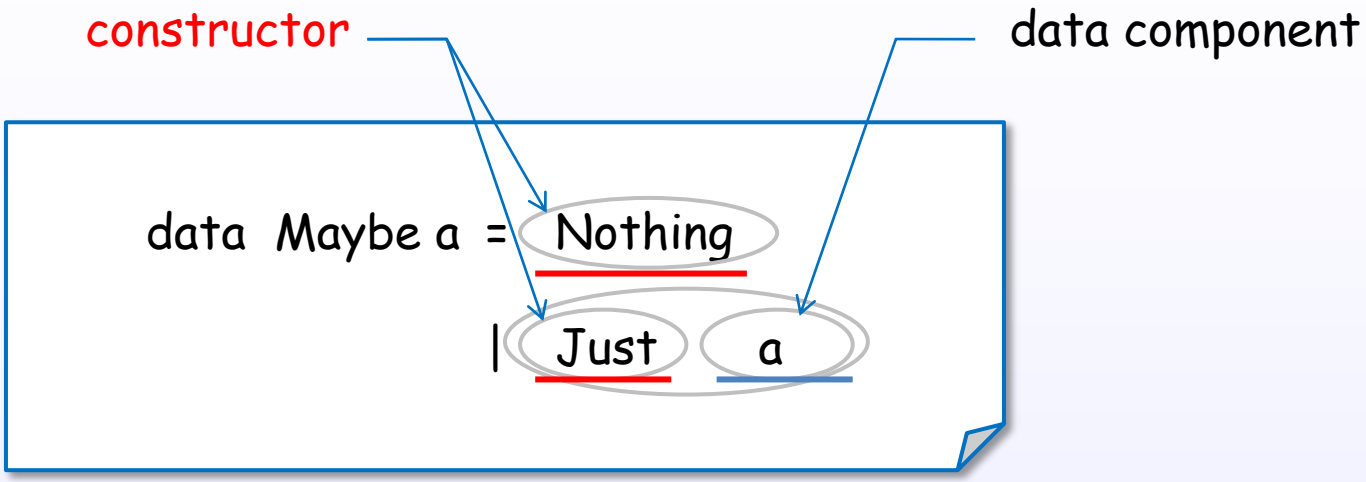


A constructor builds a structured data value.

A constructor identifies the data value in expressions.

References : [1]

Constructors are defined by data declaration

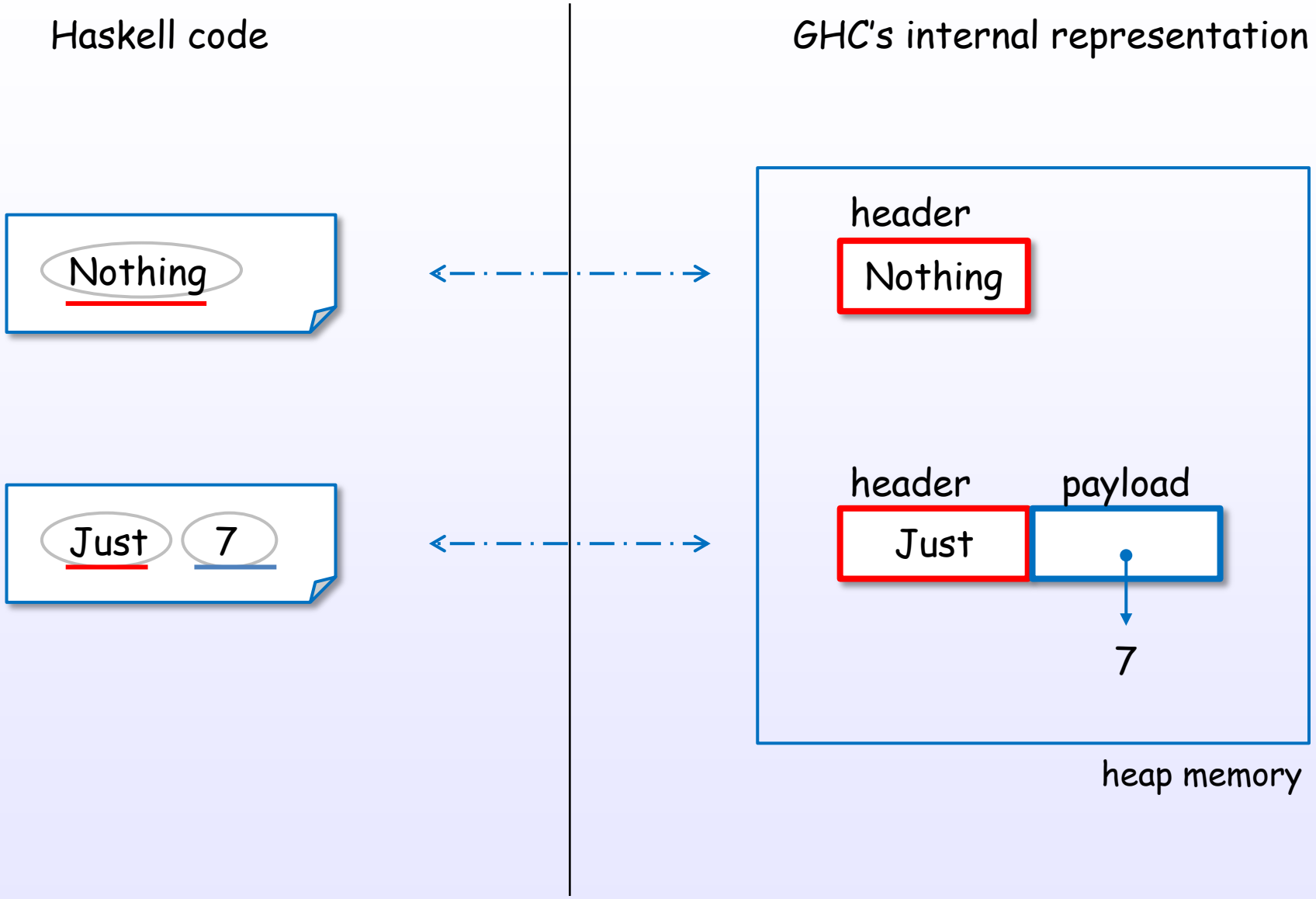


Constructors are defined by data declaration.

[slpj-book-1987] Ch.10

References : [1]

Internal representation of Constructors for data values

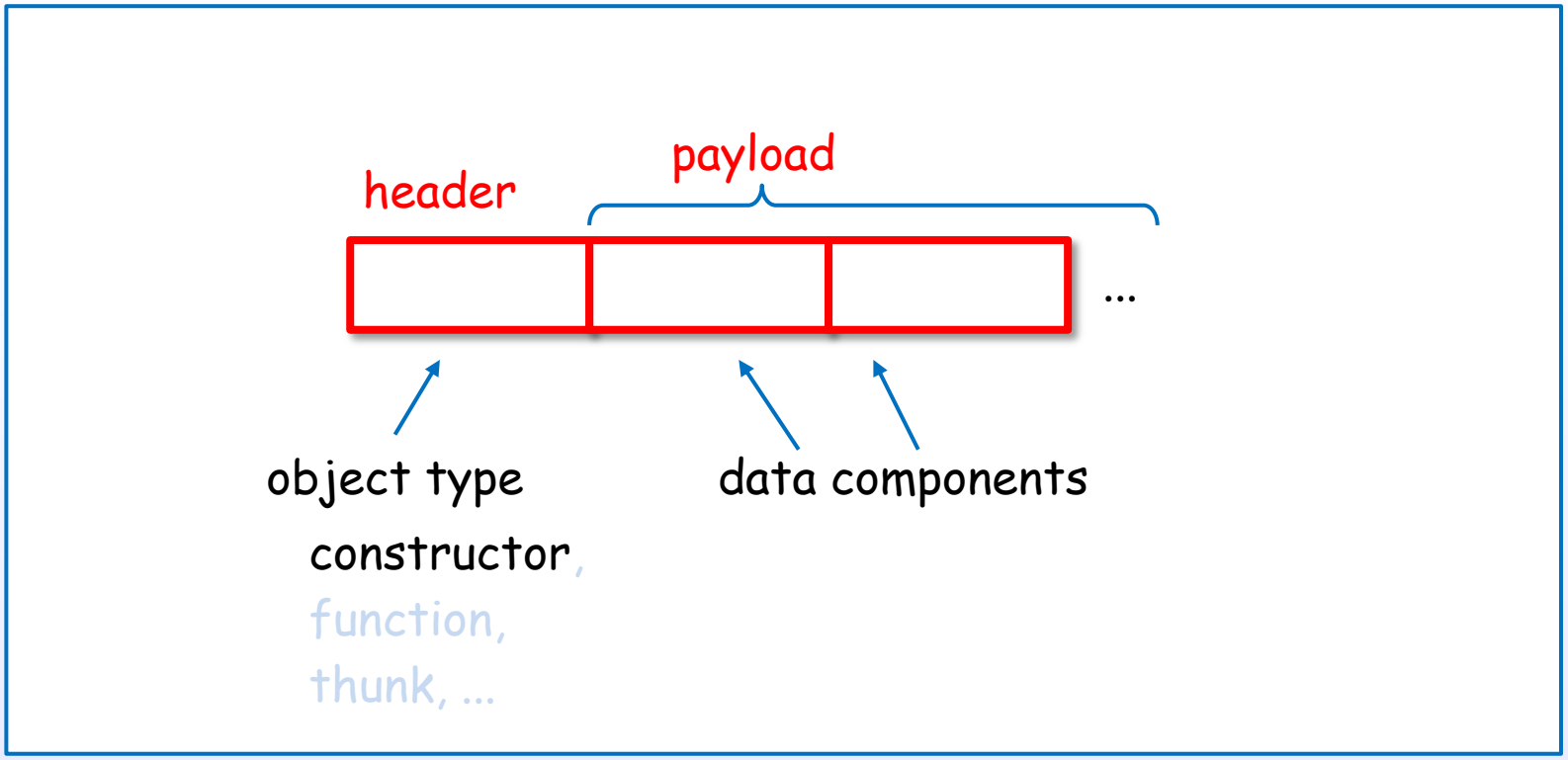


[STG]

References : [1]

Constructors are represented uniformly

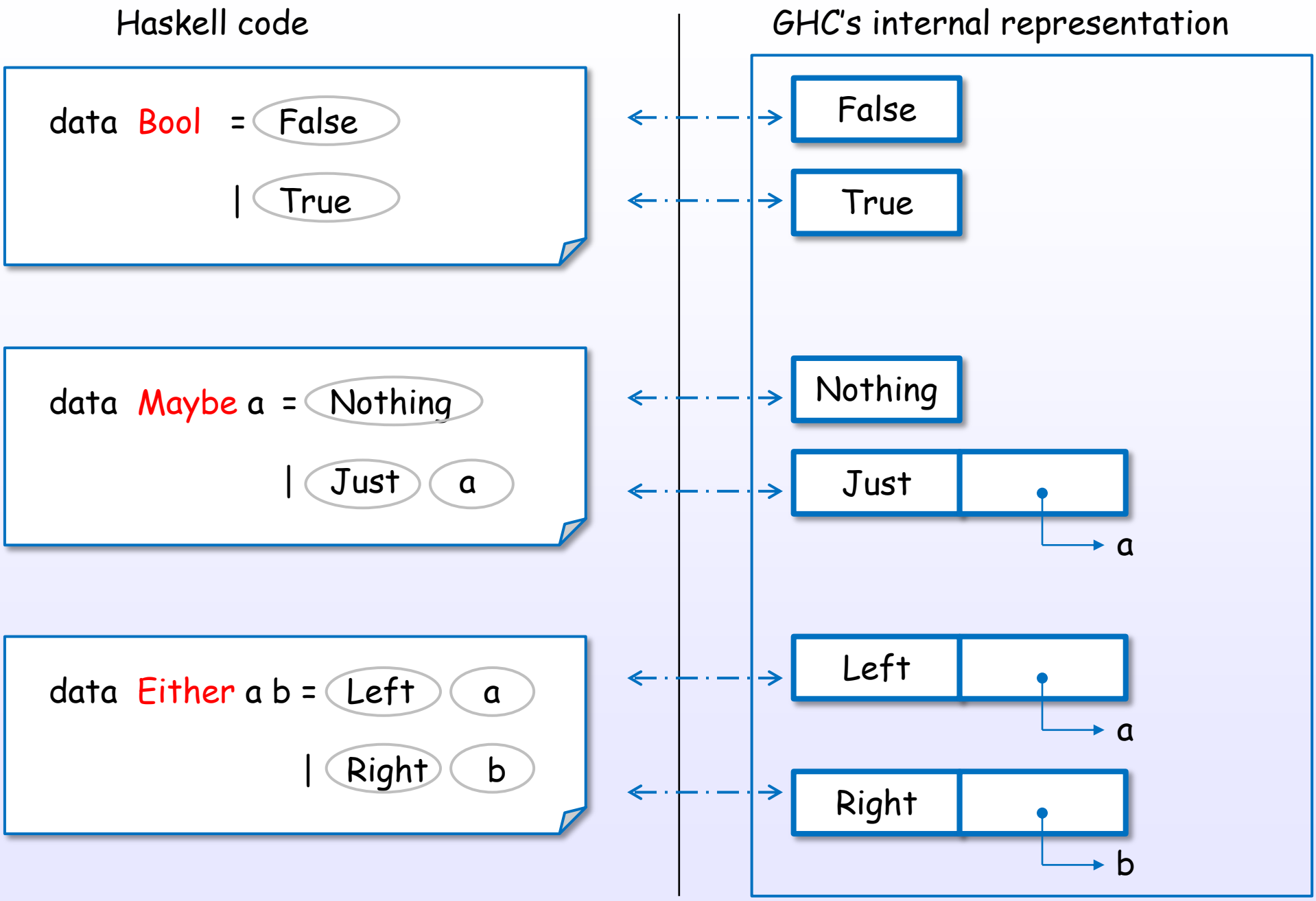
GHC's internal representation



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

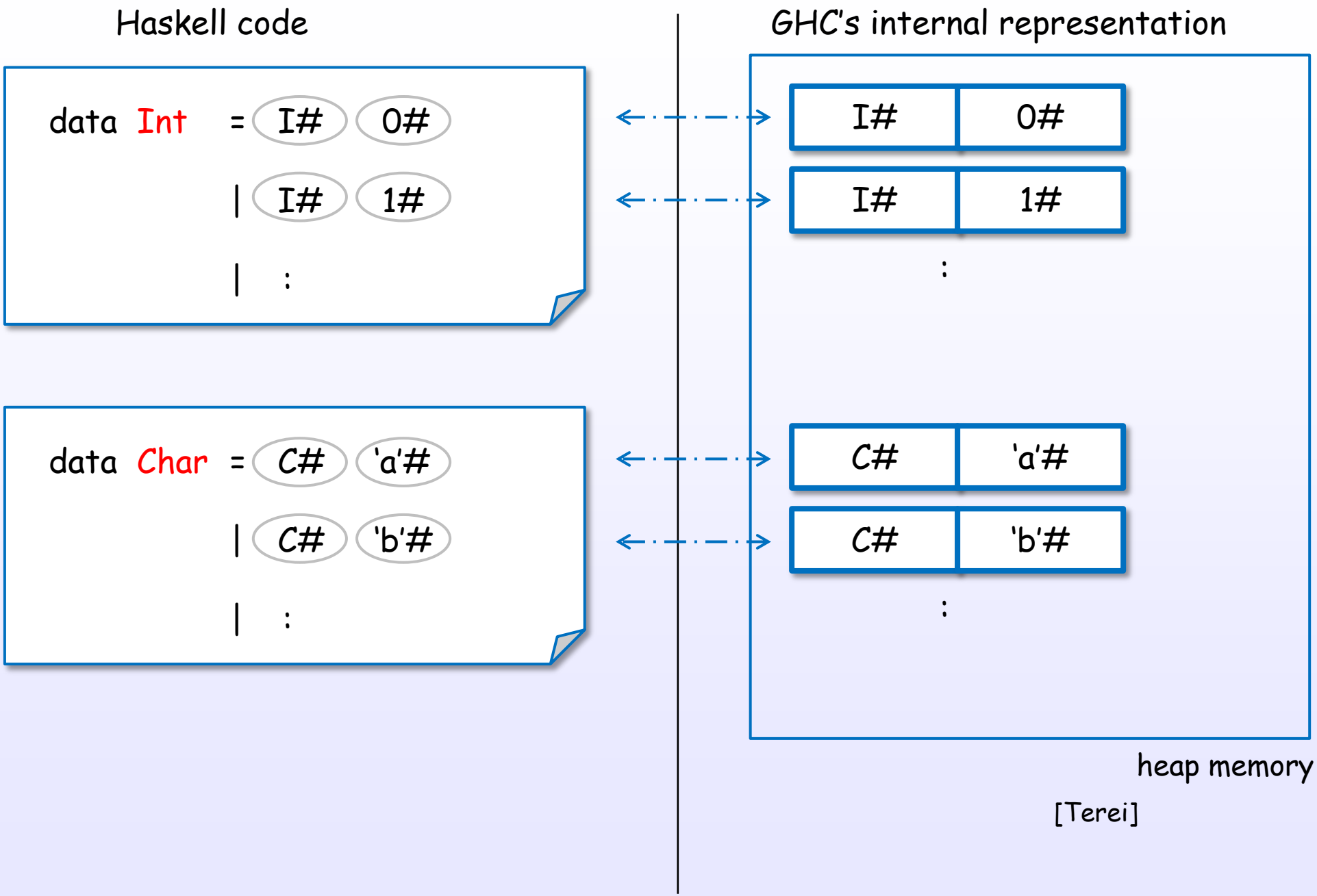
References : [1]

Representation of various constructors



References : [1]

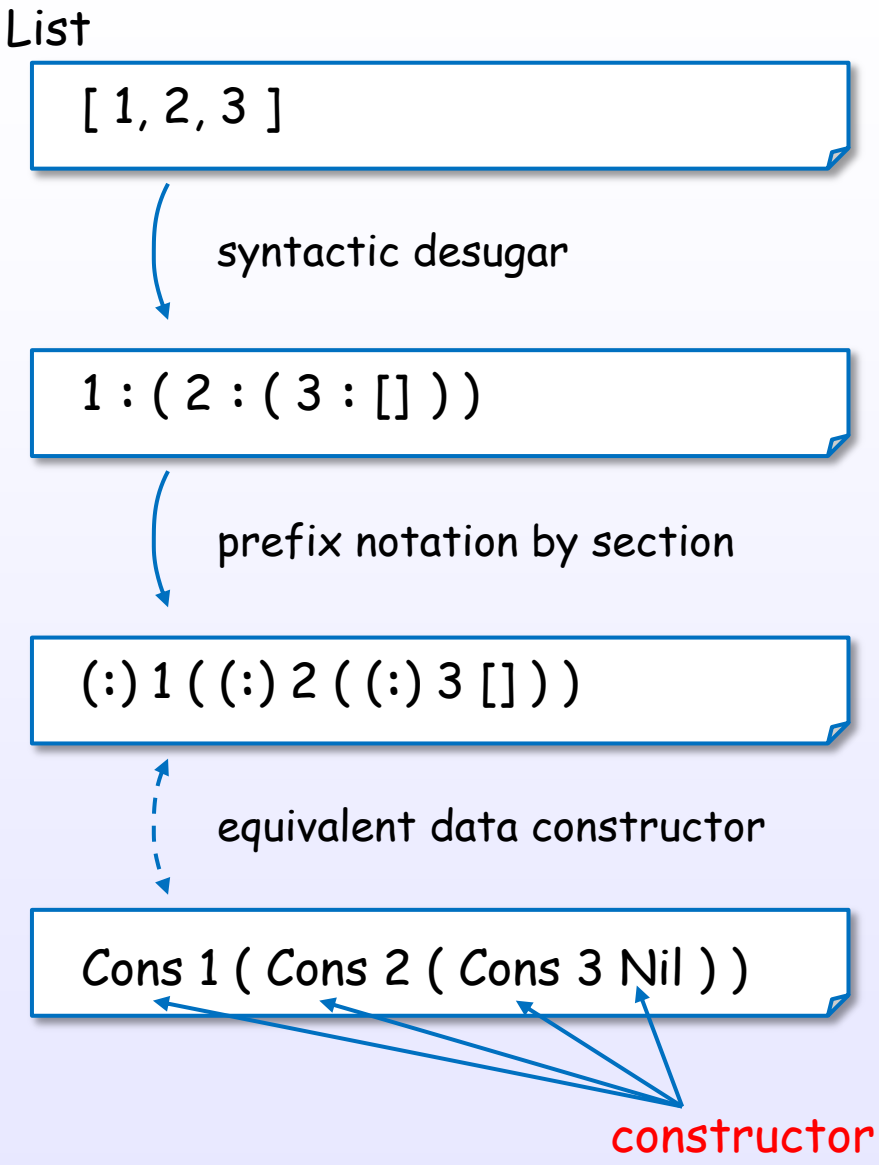
Primitive data types are also represented with constructor



[Terei]

References : [1]

List is also represented with constructor



References : [1]

List is also represented with constructor

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

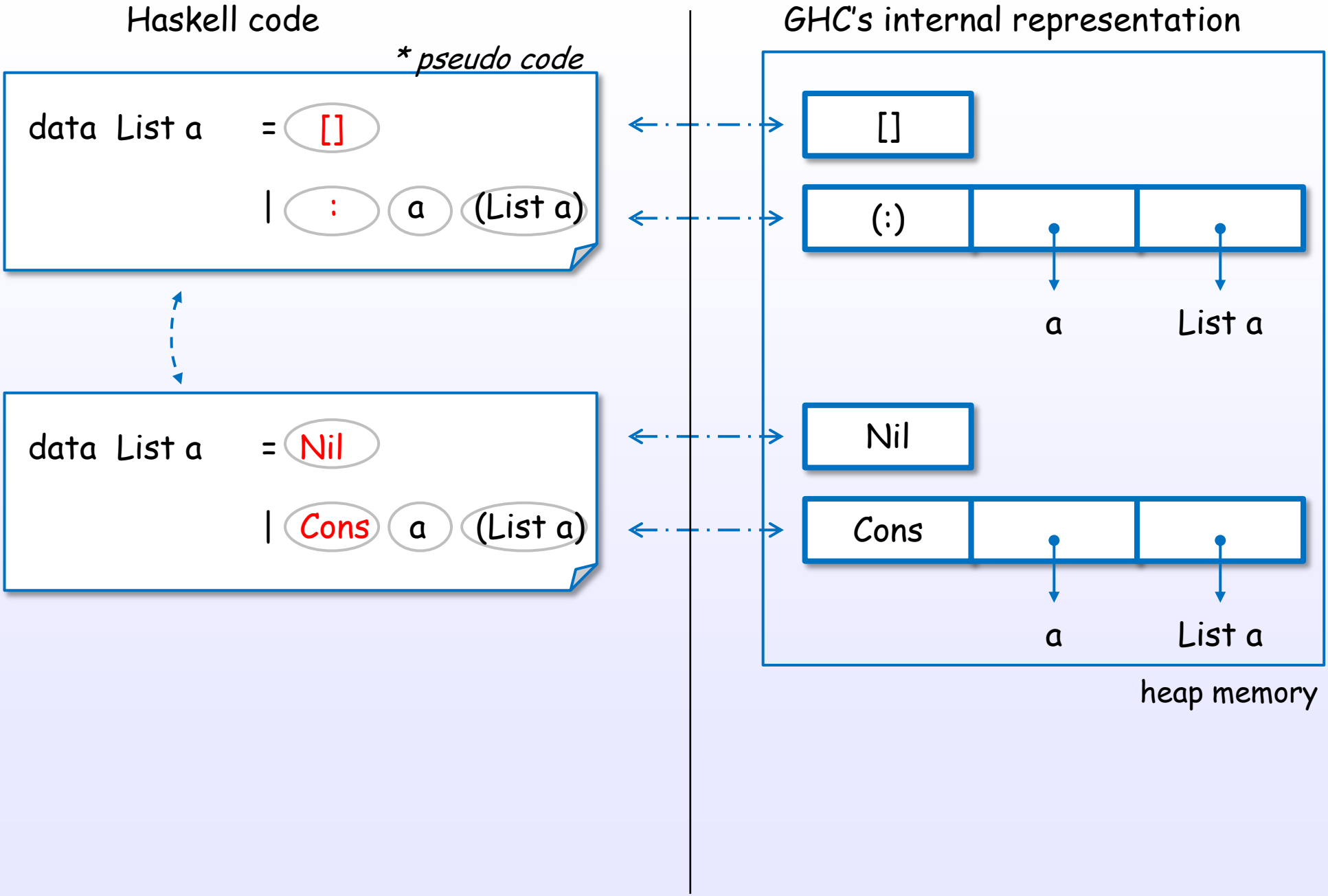
** pseudo code*

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

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

References : [1]

List is also represented with constructor



References : [1]

List is also represented with constructor

Haskell code

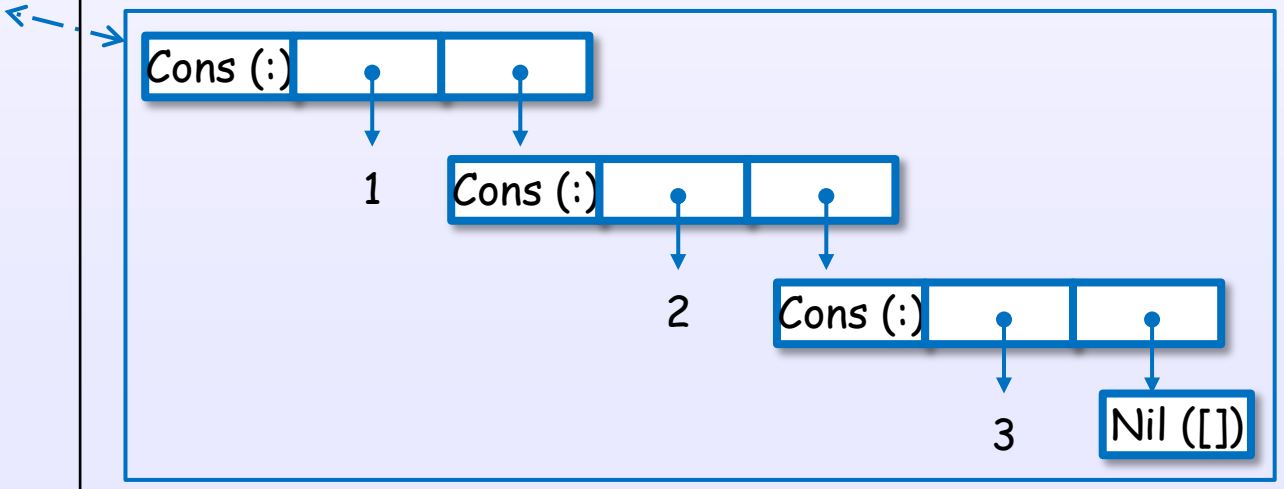
```
[ 1, 2, 3 ]
```

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

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

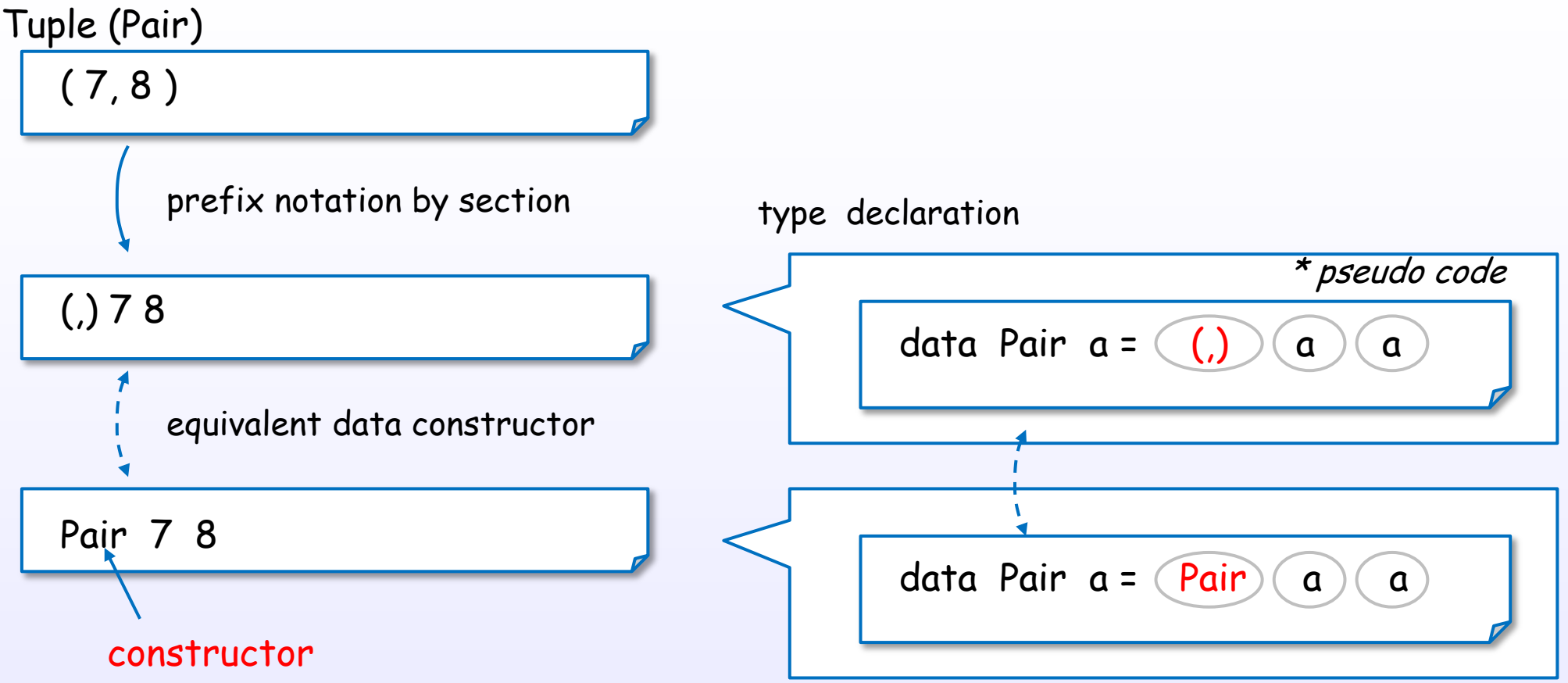
```
Cons 1 ( Cons 2 ( Cons 3 Nil ) )
```

GHC's internal representation



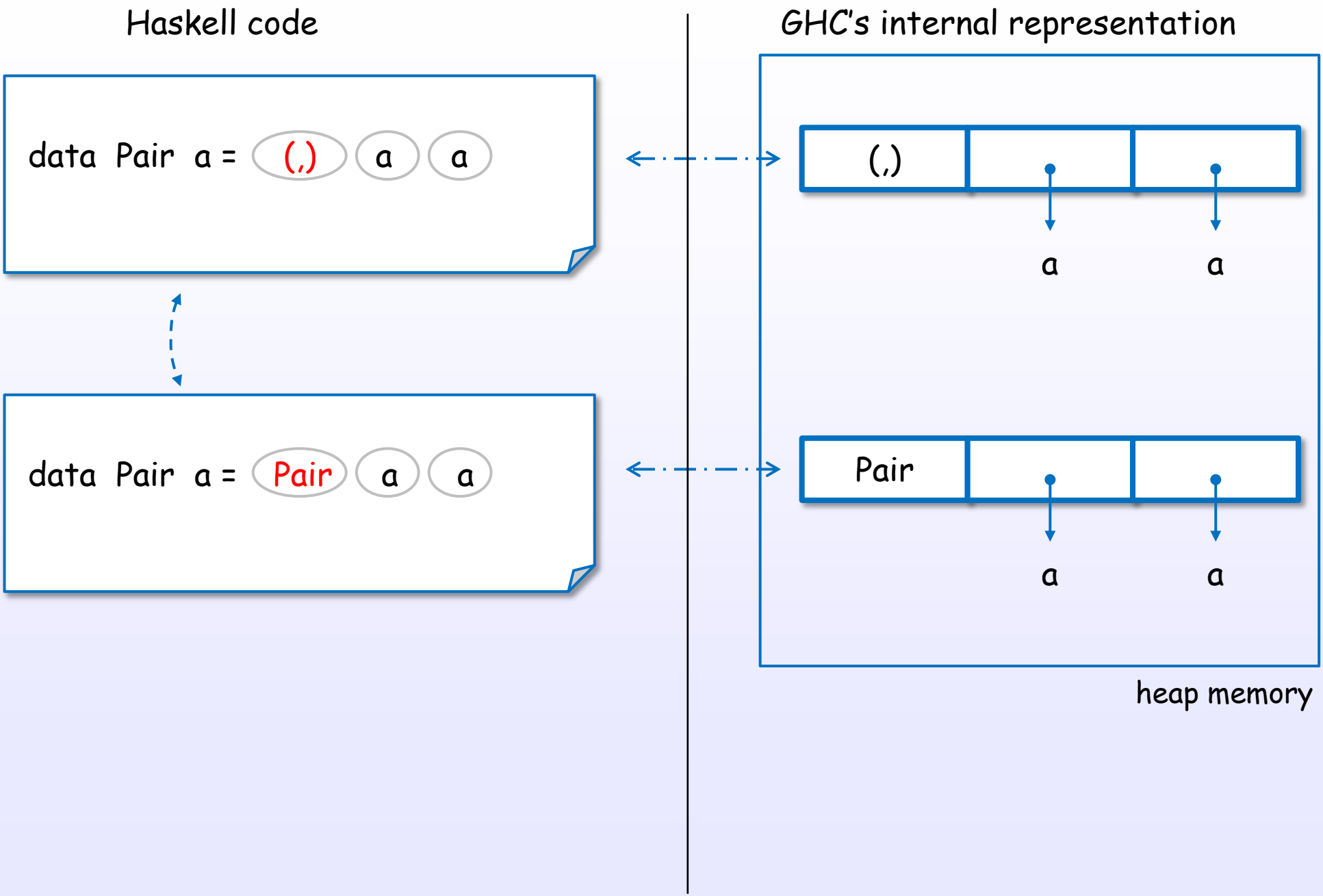
References : [1]

Tuple is also represented with constructor



References : [1]

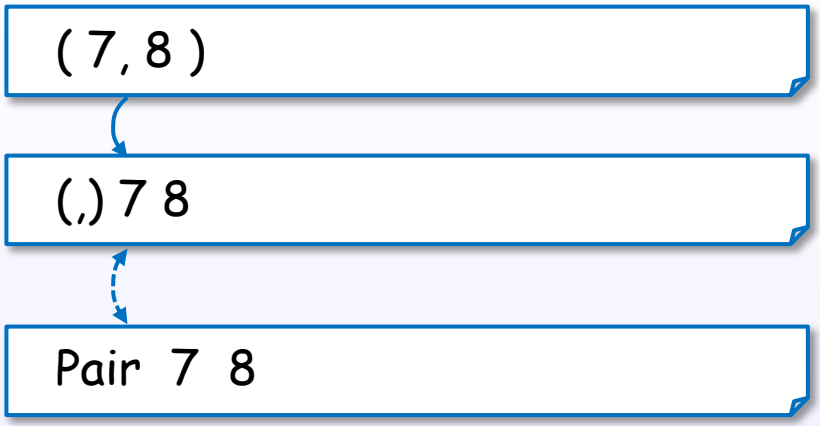
Tuple is also represented with constructor



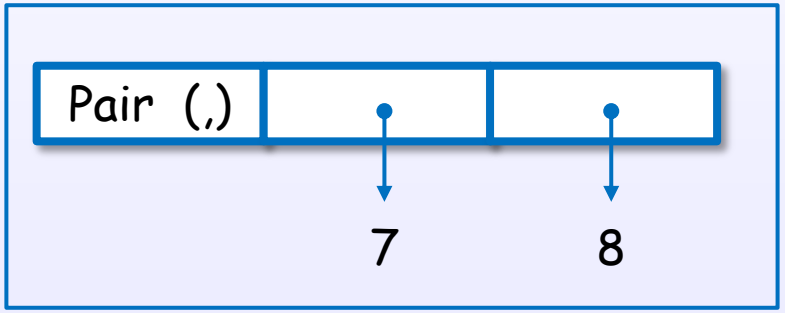
References : [1]

Tuple is also represented with constructor

Haskell code



GHC's internal representation

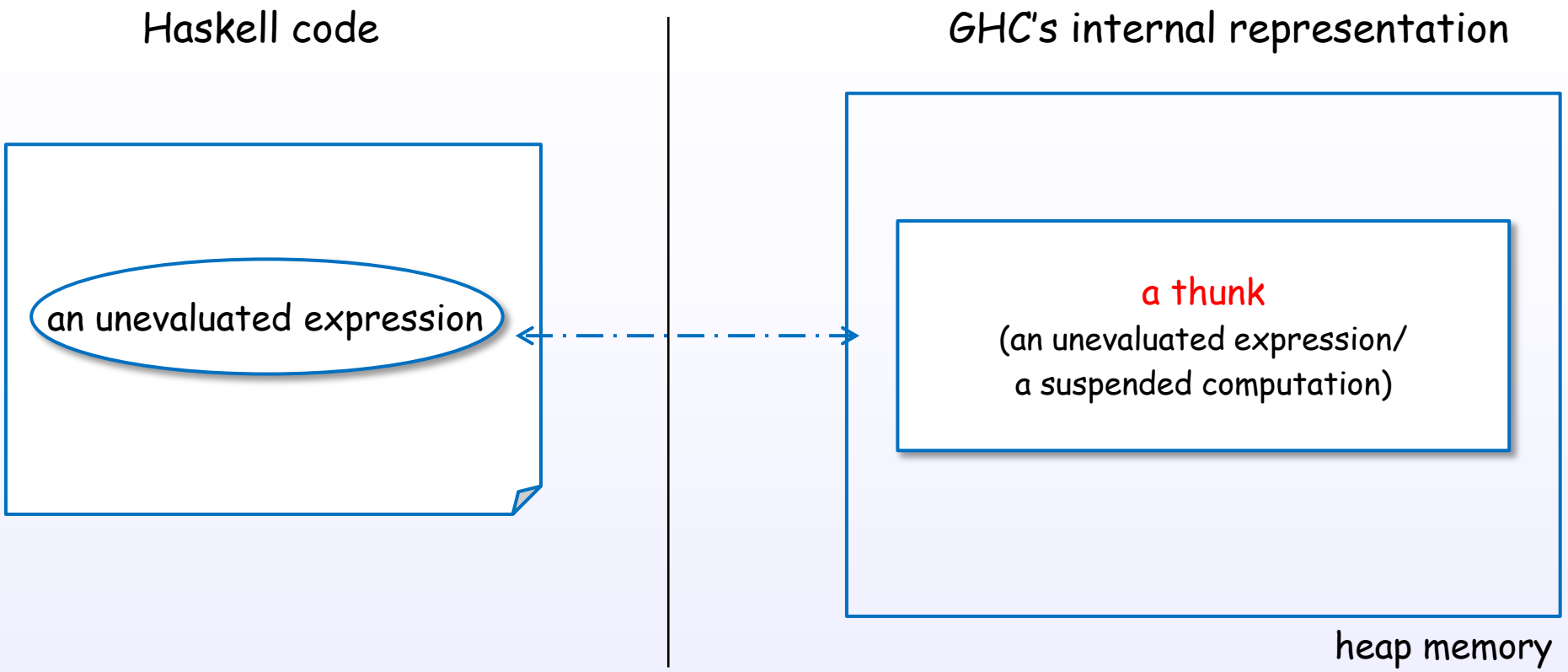


References : [1]

3. Internal representation of expressions

Thunk

Thunk



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

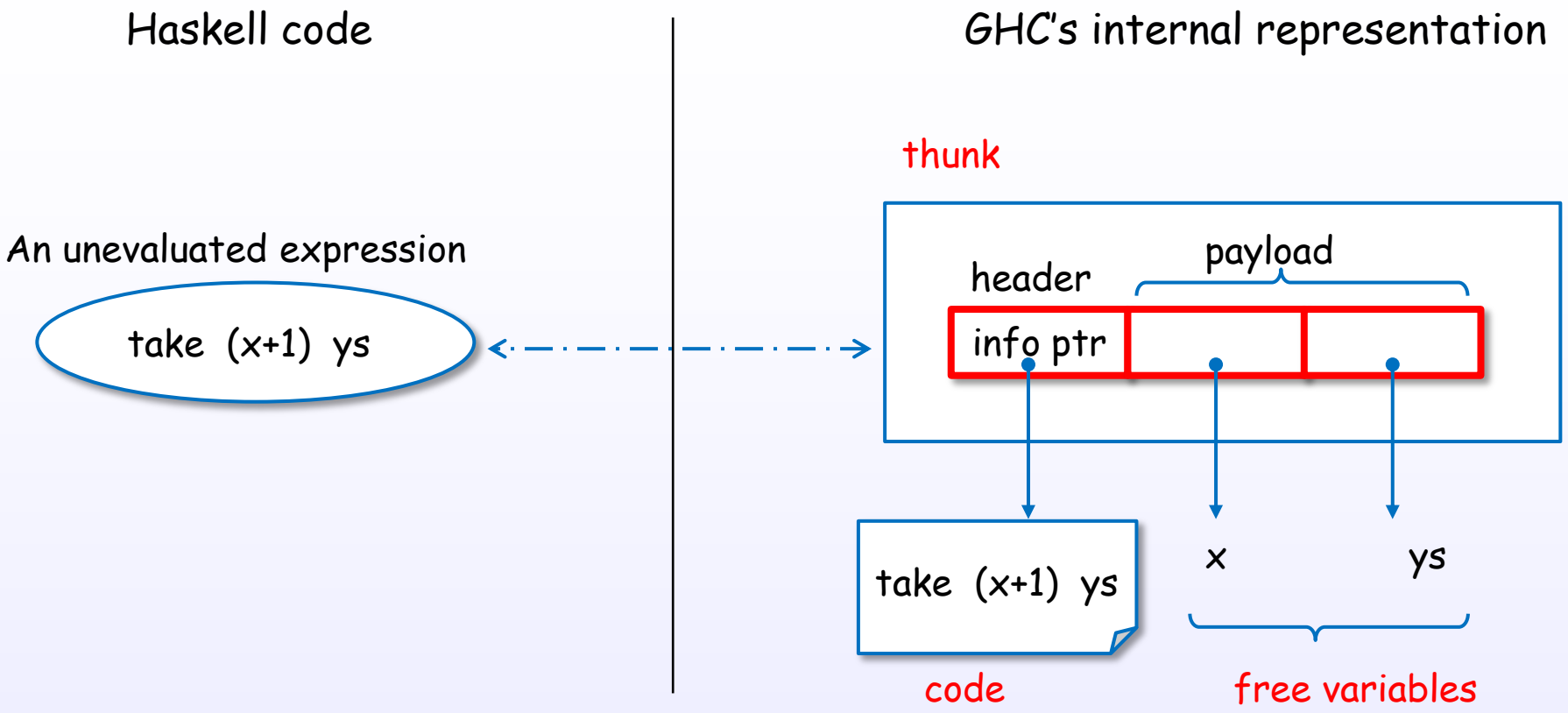
[parconc, Ch.2]

[hack.hands]

[Haskell/Laziness]

References : [1]

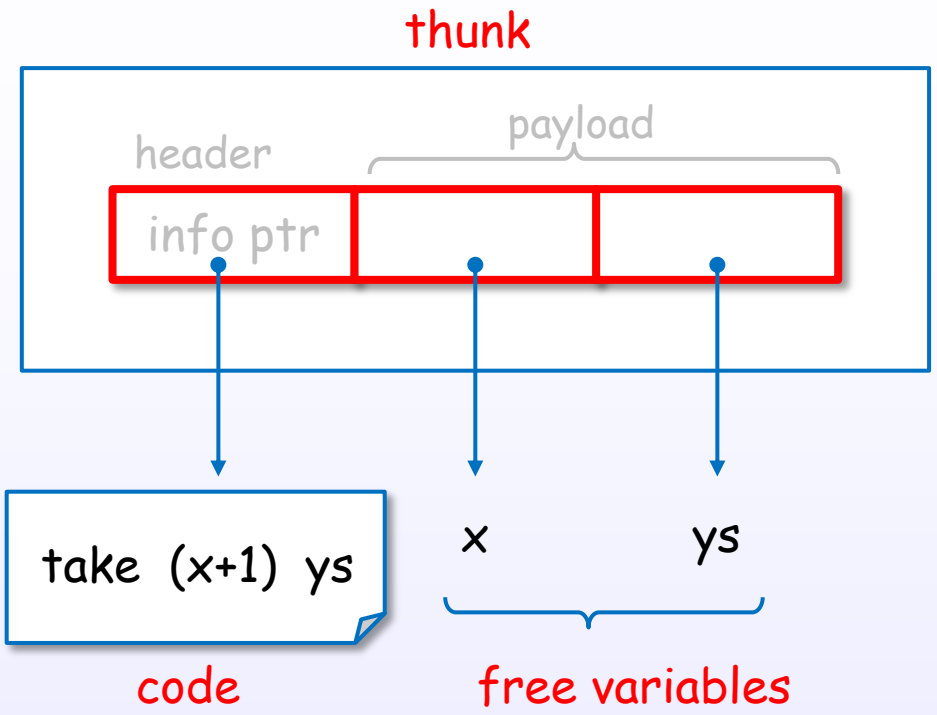
Internal representation of thunk



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

References : [1]

A thunk is a package of code and free variables

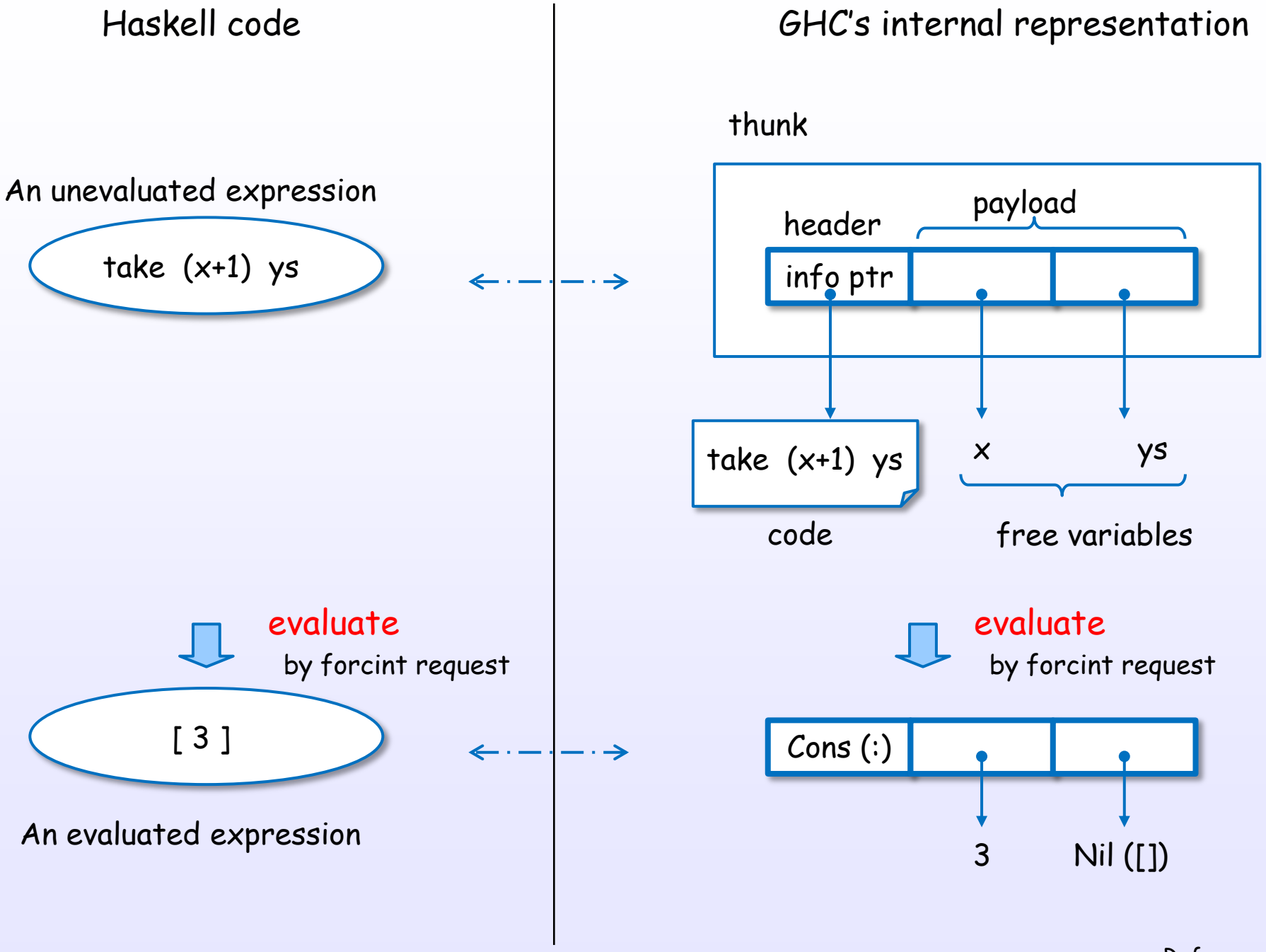


A thunk is a package of code + free variables.

[CIS194]

References : [1]

A thunk is evaluated by forcing request

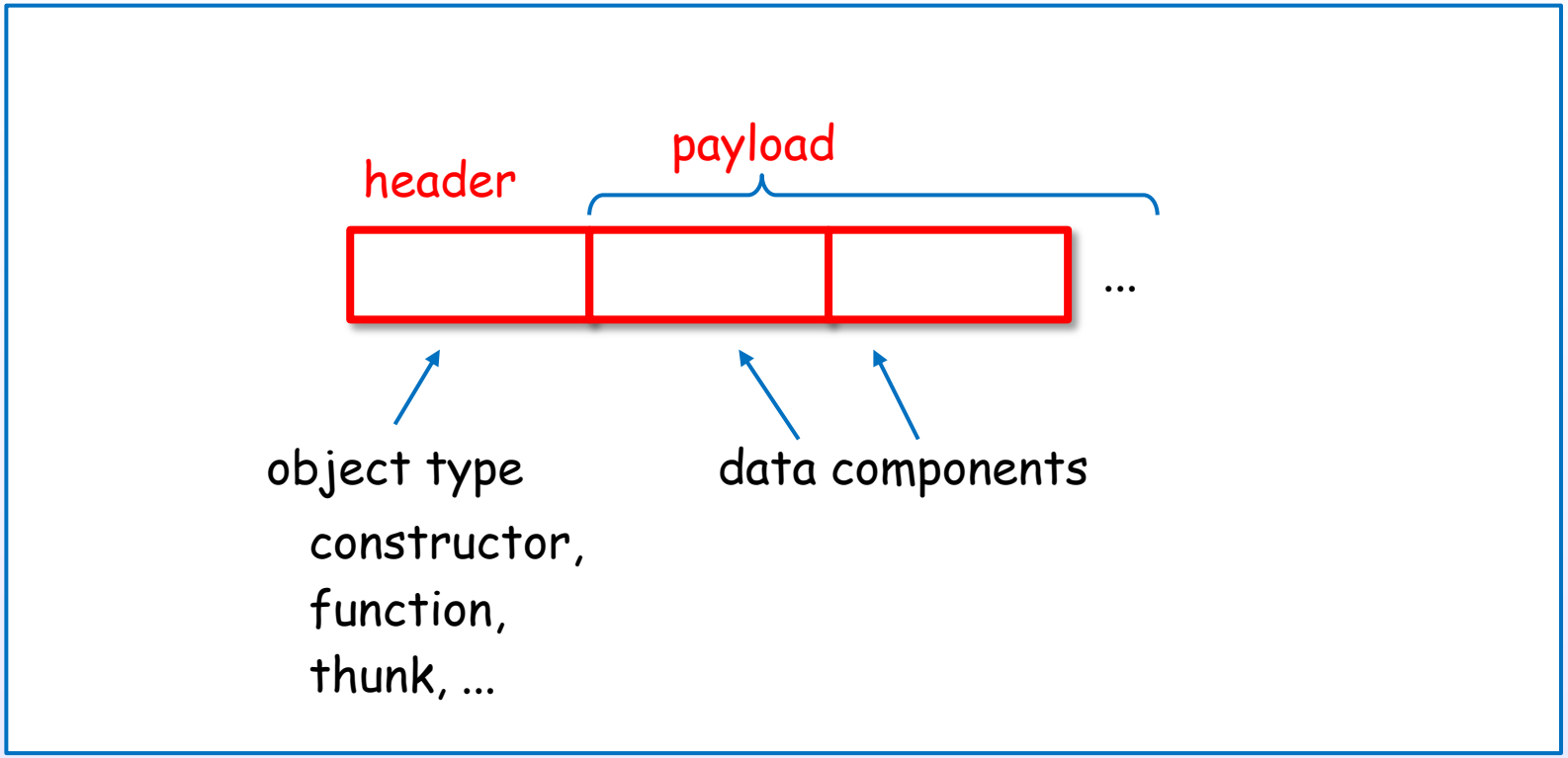


References : [1]

3. Internal representation of expressions

Uniform representation

Every object is represented uniformly in memory

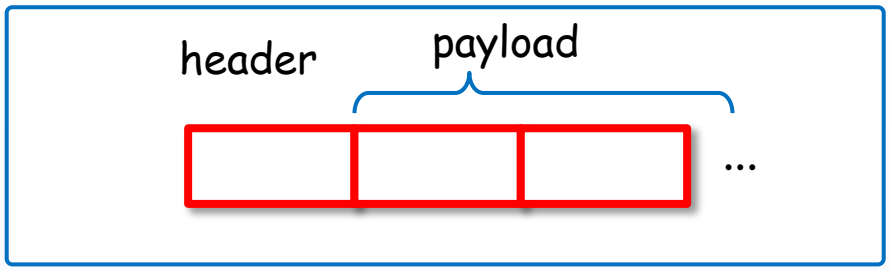


in heap memory, stack or static memory

[STG]

References : [1]

Every object is represented uniformly



a data value

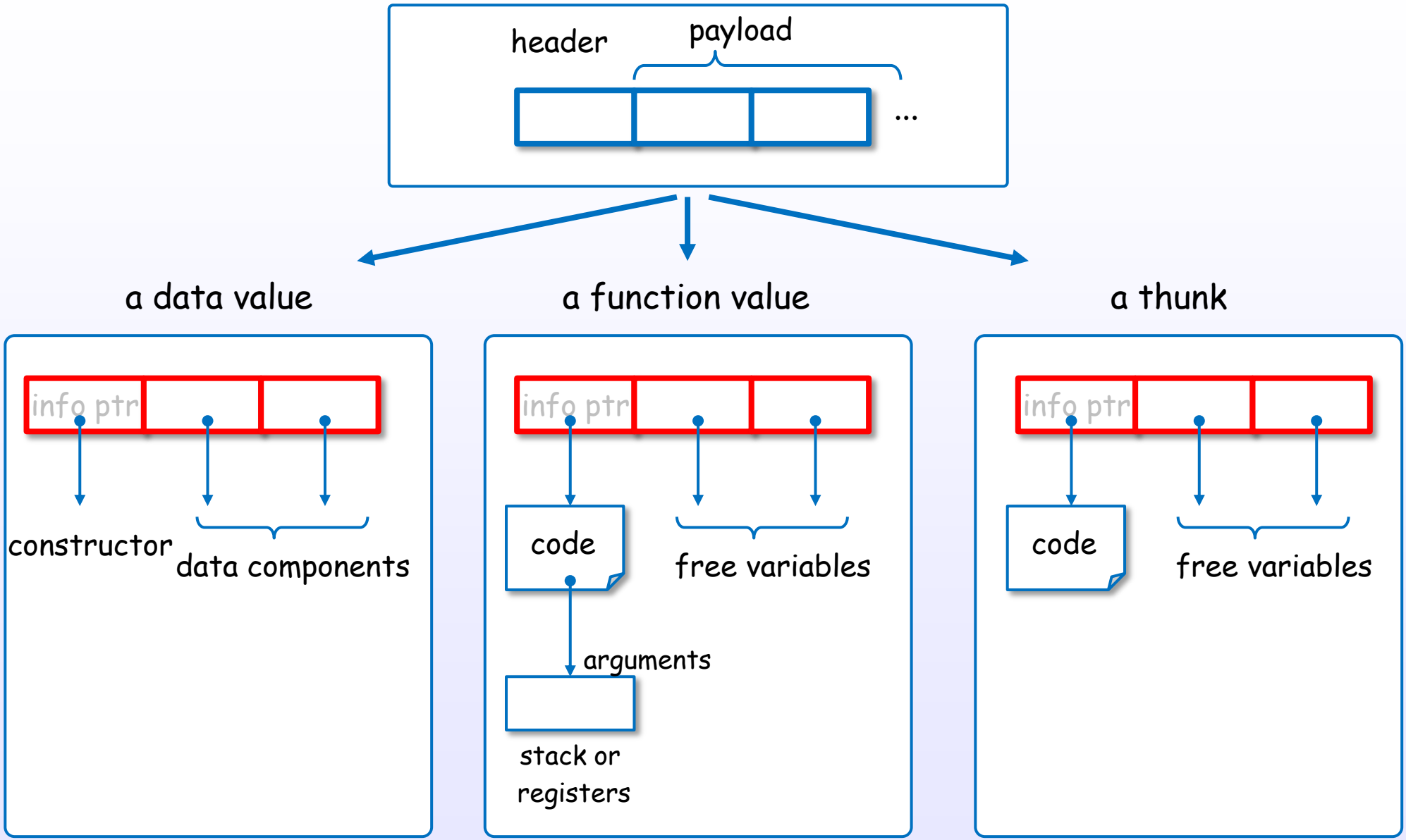
a function value

a thunk

[STG]

References : [1]

Every object is represented uniformly



References : [1]

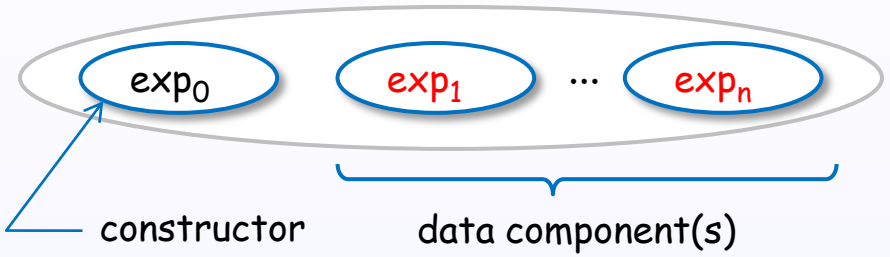
3. Internal representation of expressions

WHNF

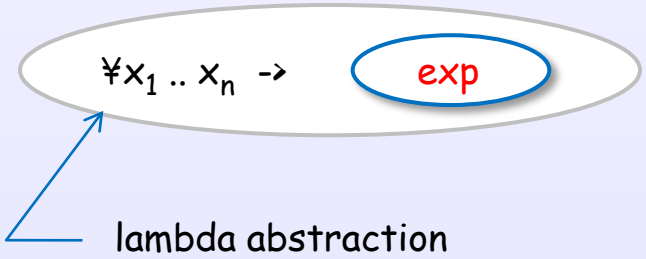
Internal representation of WHNF

Haskell code

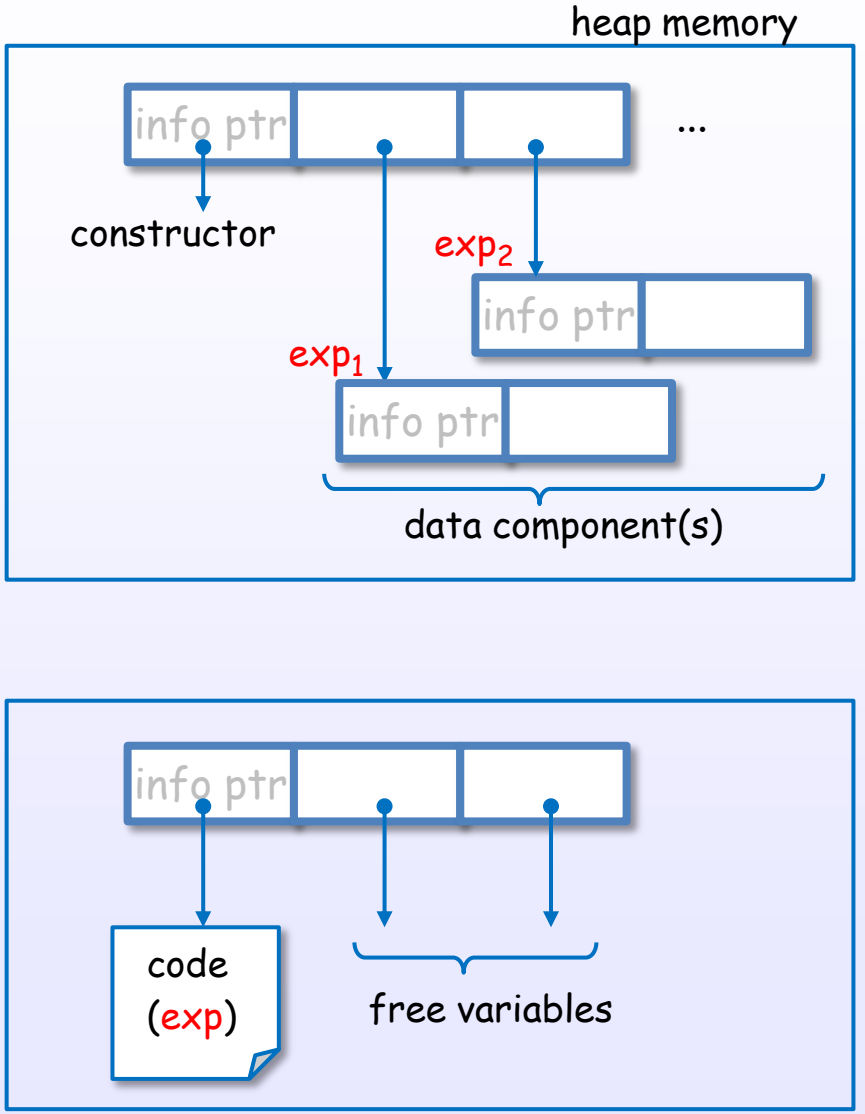
a data value in WHNF



a function value in WHNF



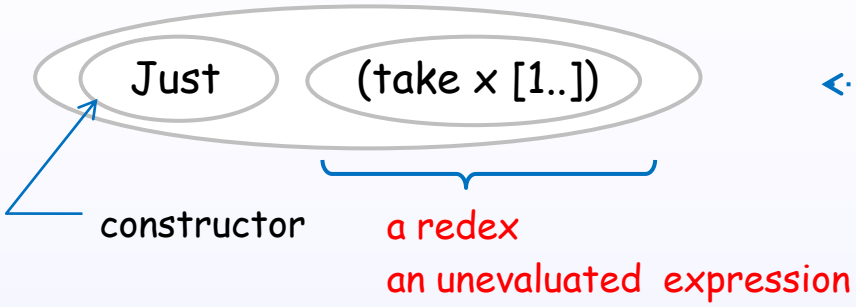
GHC's internal representation



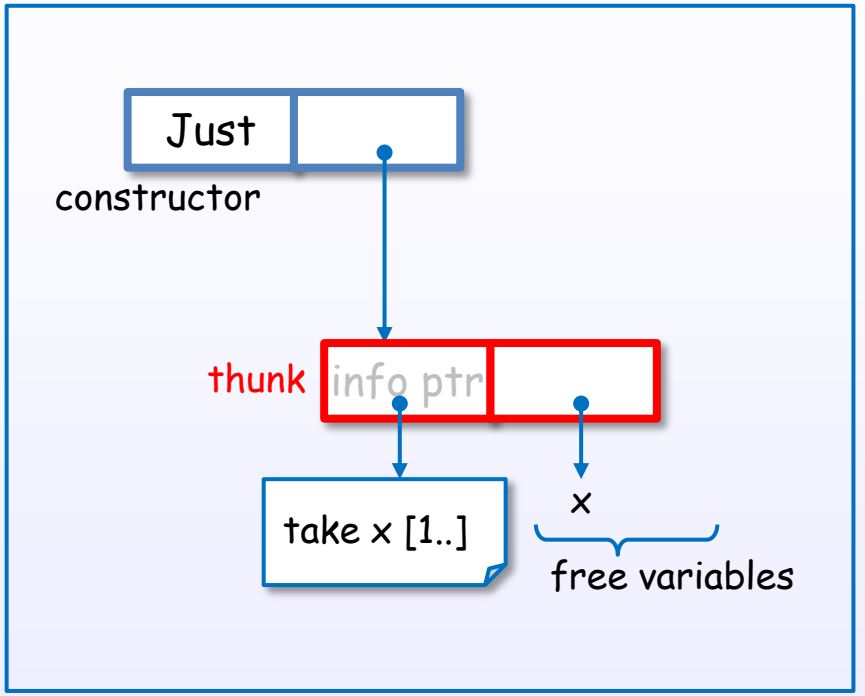
References : [1]

Example of WHNF for a data value

Haskell code



GHC's internal representation

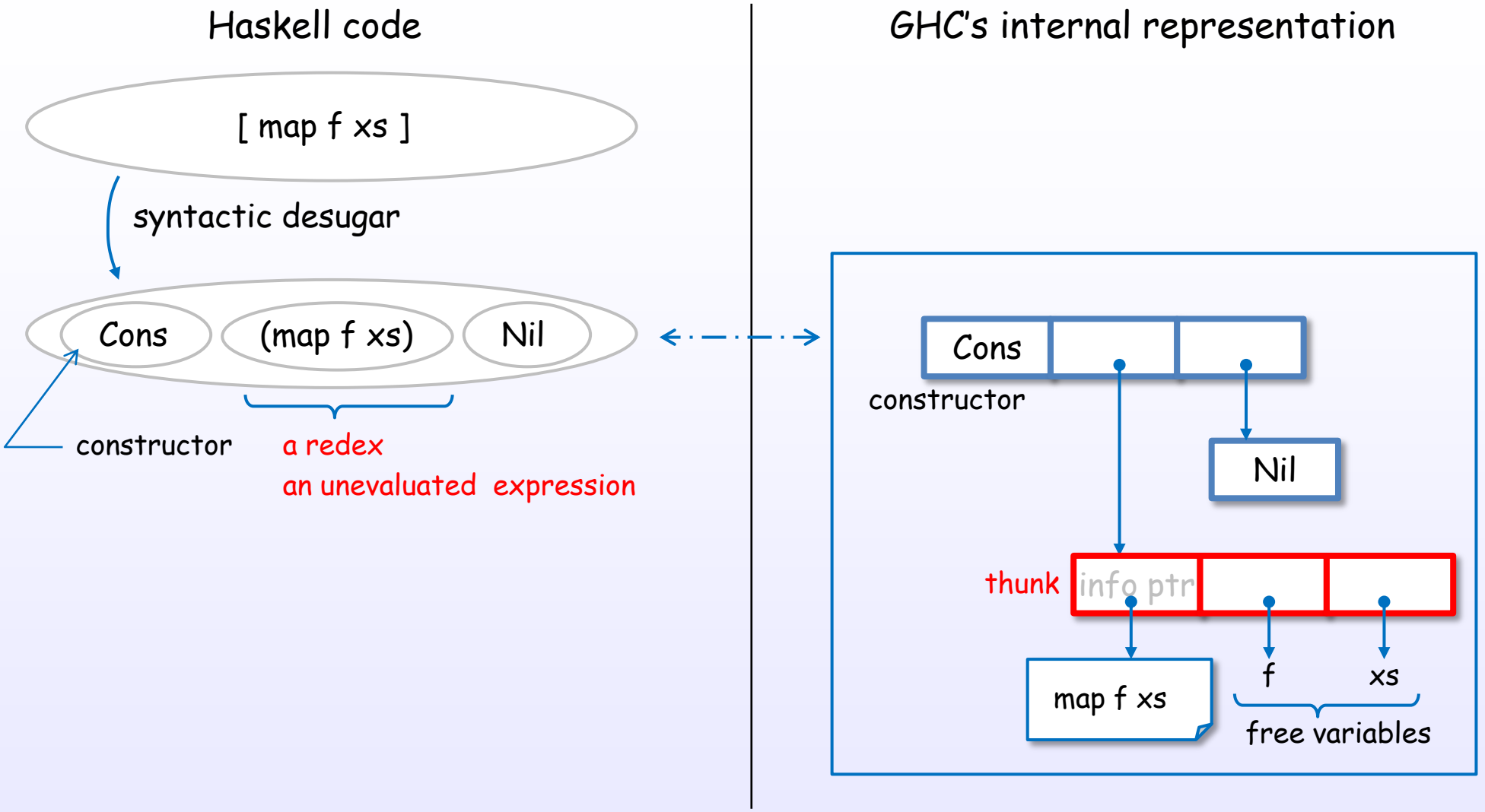


Constructors can contain unevaluated expressions by thunks.

Haskell's constructors are lazy constructors.

References : [1]

Example of WHNF for a data value



References : [1]

3. Internal representation of expressions

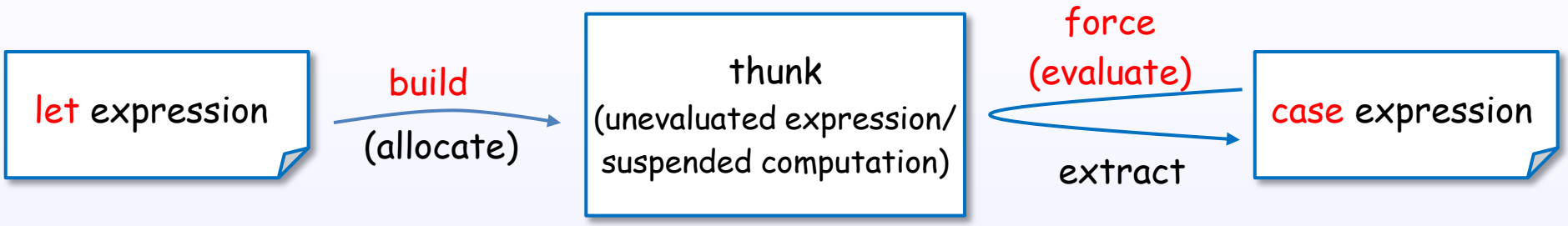
let, case expression

let, case expression

let and case expressions are special role in the evaluation

References : [1]

let/case expressions and thunk

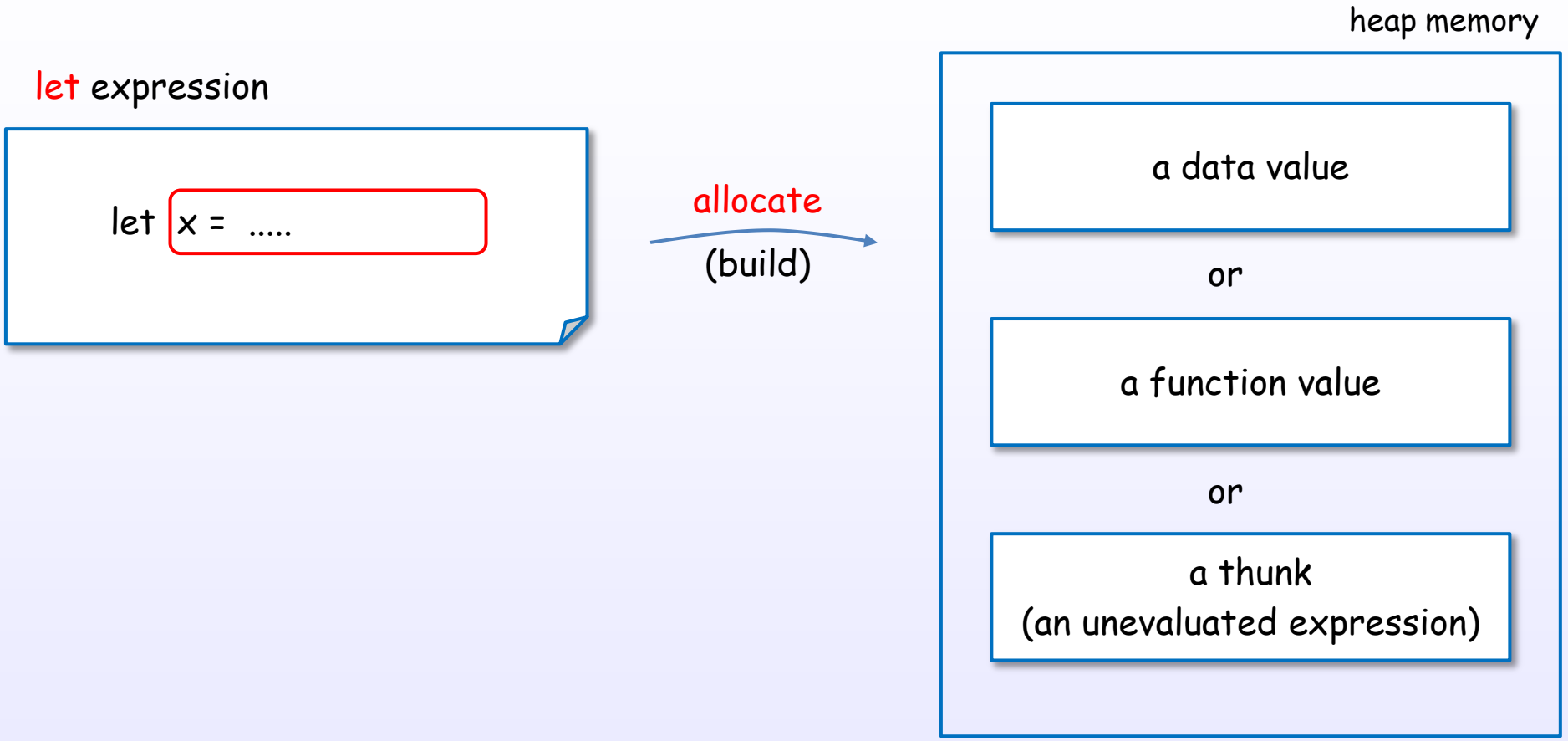


A let expression may build a thunk.
A case expression forces and deconstructs the thunk.

[STG]

References : [1]

A let expression may allocates a heap object



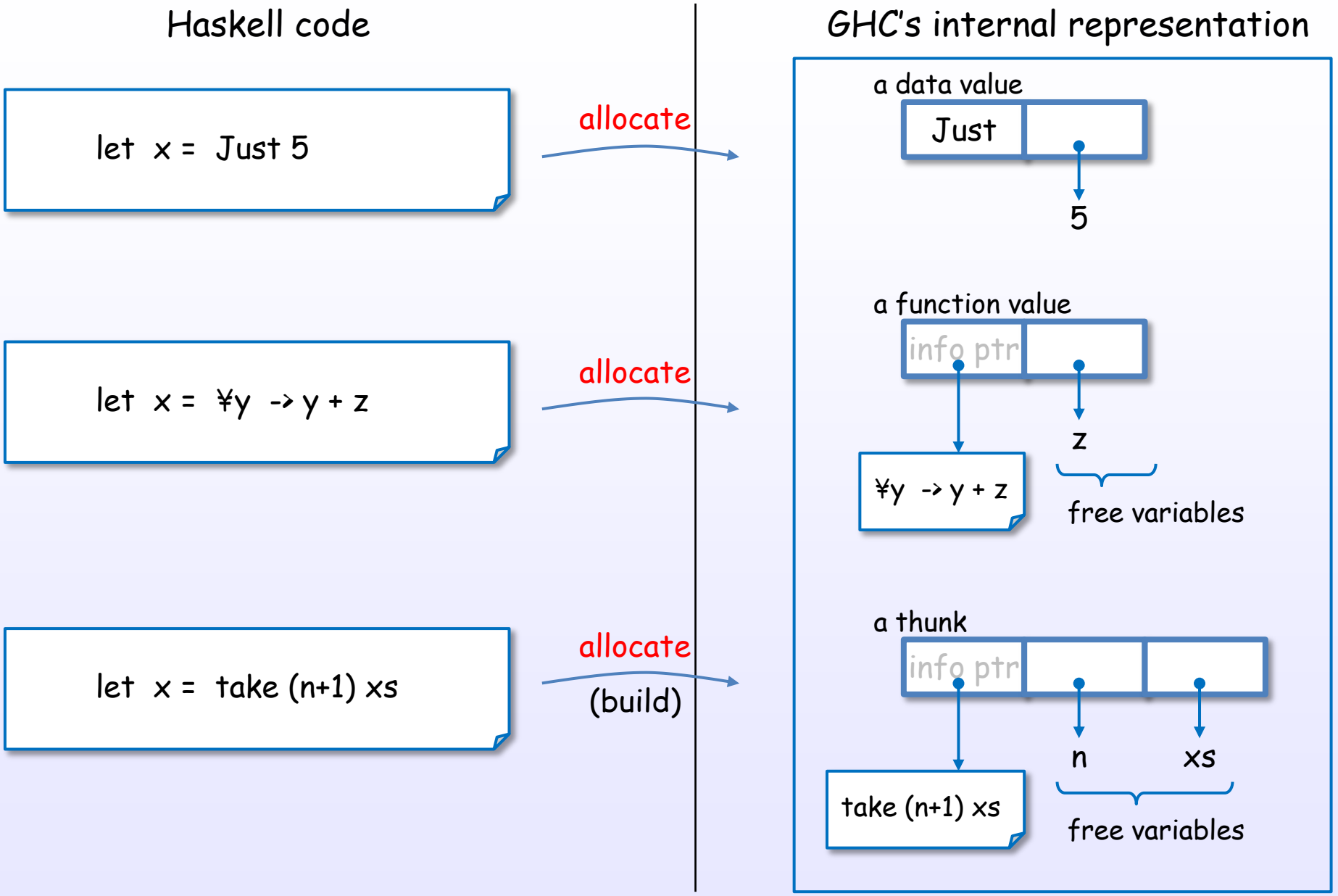
A let expression allocates an object in the heap.

[STG], [push/enter]

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

References : [1]

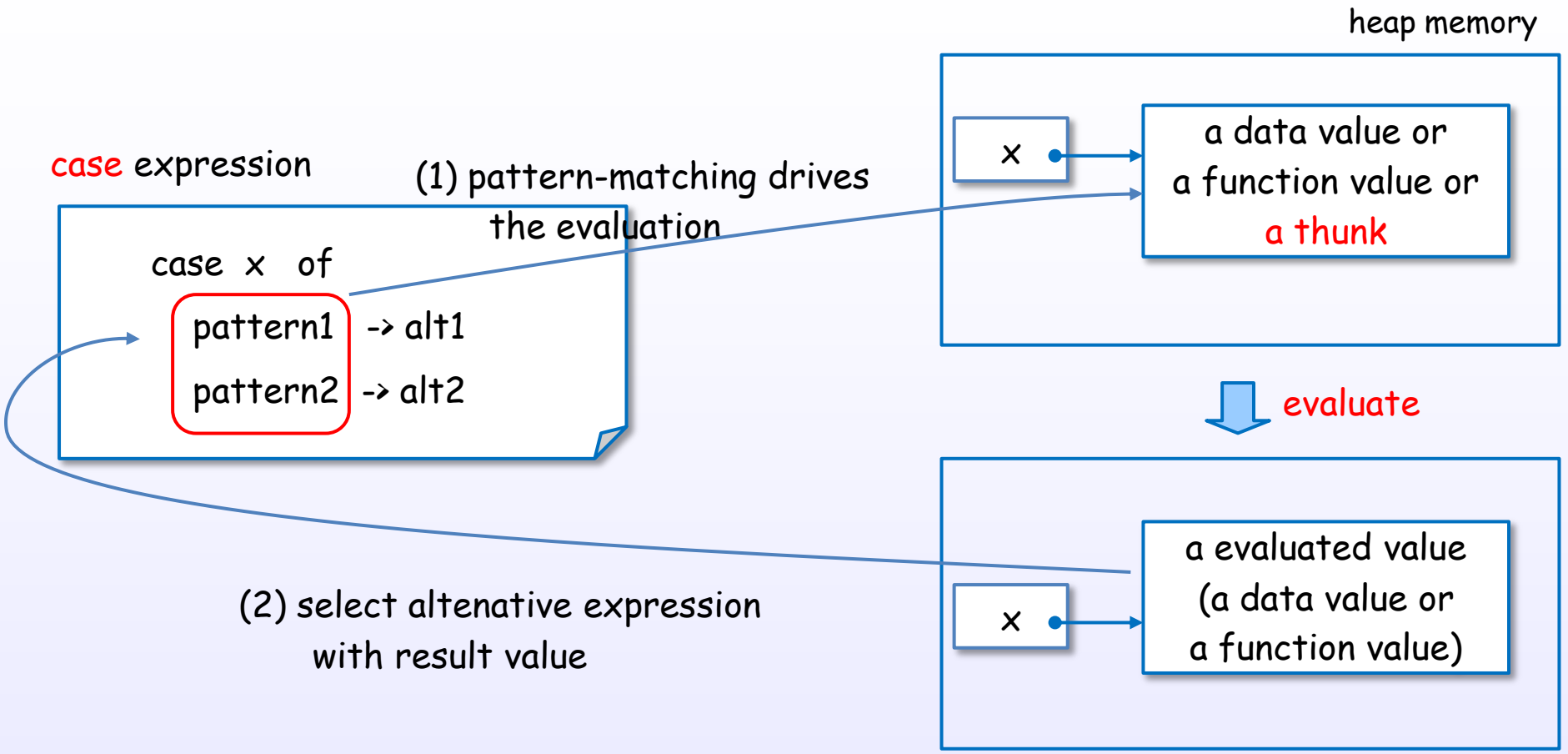
Example of let expressions



[STG], [push/enter]

References : [1]

A case expression allocates a heap object



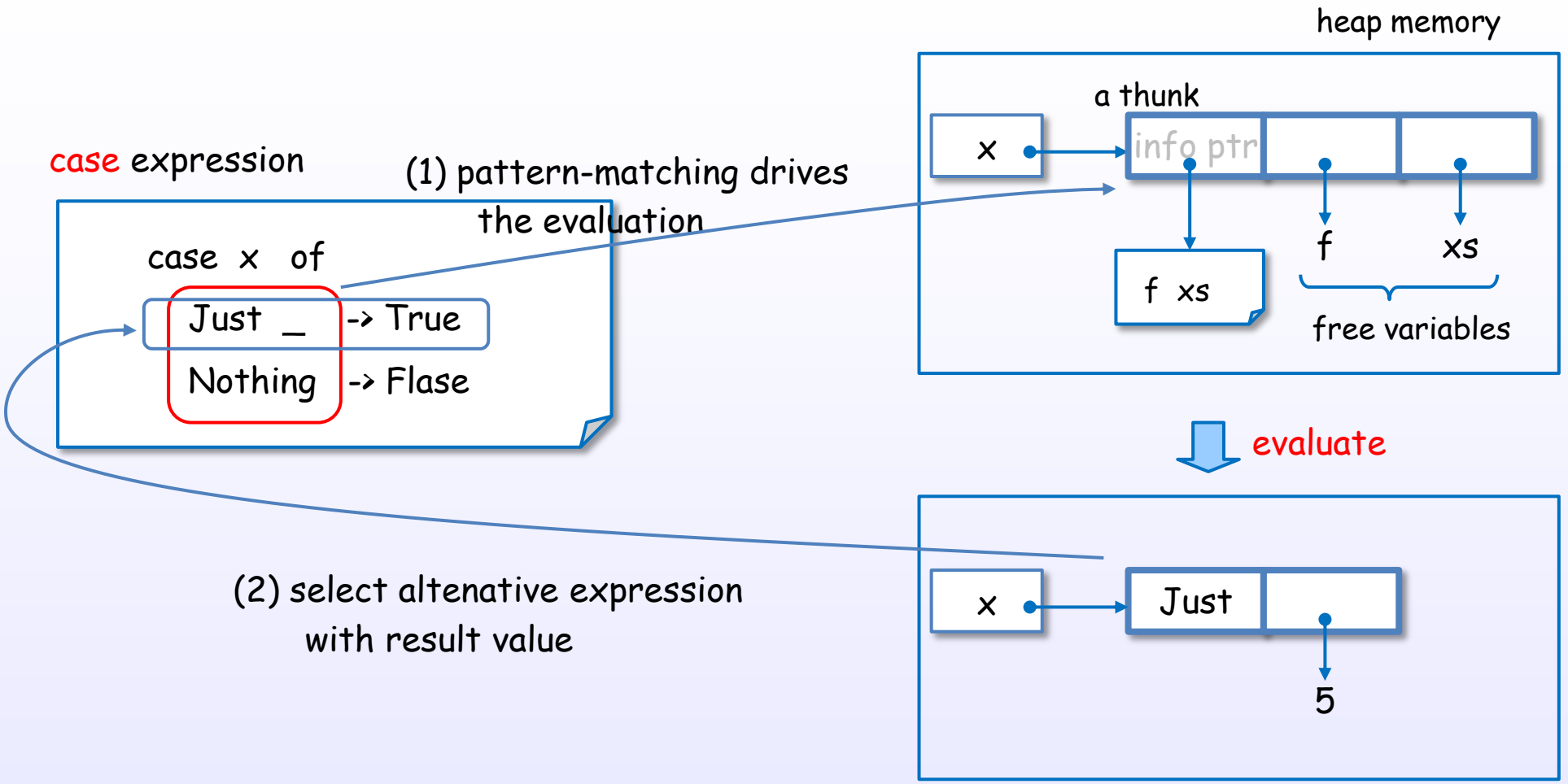
A case expression evaluates a subexpression
and optionally performs case analysis on its value.

[STG], [push/enter]

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

References : [1]

Example of a case expression



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

[STG], [push/enter]

References : [1]

pattern-matching in function definition

pattern-matching in function definition

```
f Just _    = True
f Nothing   = Flase
```

pattern-matching in case expression

```
f x = case x of
        Just _   -> True
        Nothing  -> Flase
```

A function's pattern-matching is syntactic sugar of case expression.

A function's pattern-matching also drives the evaluation.

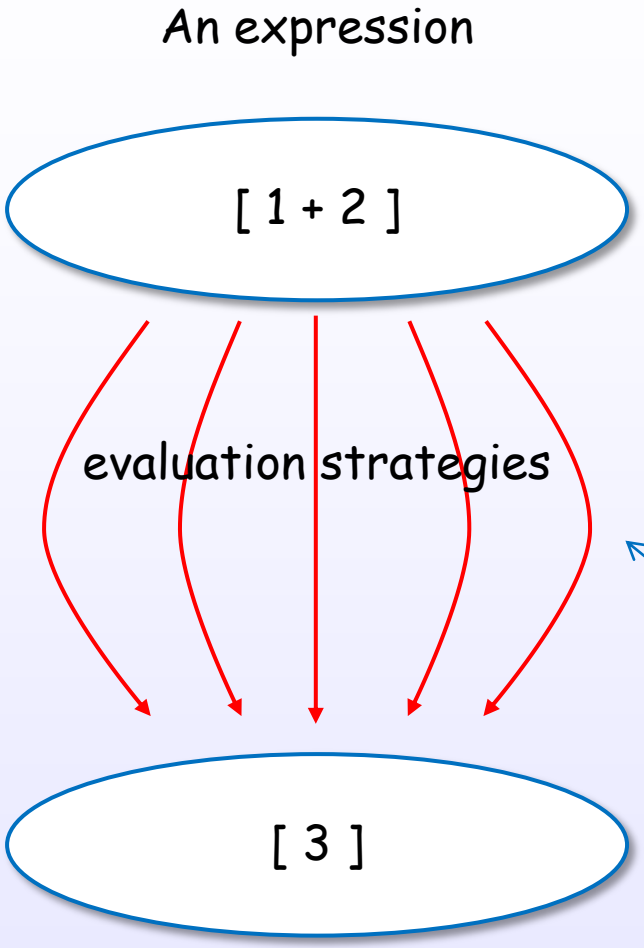
References : [1]

4. Evaluation

4. Evaluation

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 concept layer

Denotational semantics

Operational semantics
(Evaluation strategies / Reduction strategies)

Implementation techniques

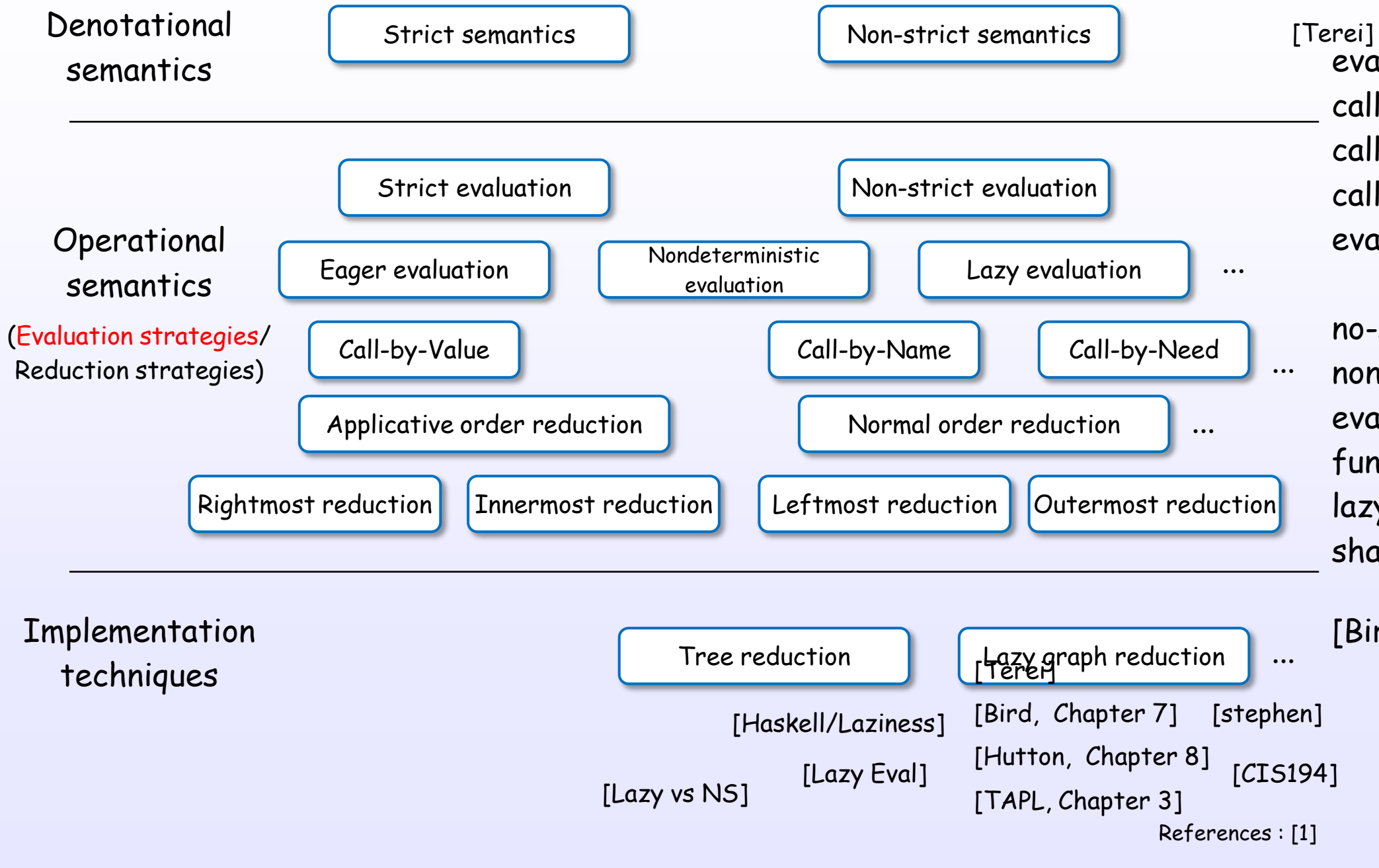
[Bird, Chapter 7]

[Hutton, Chapter 8]

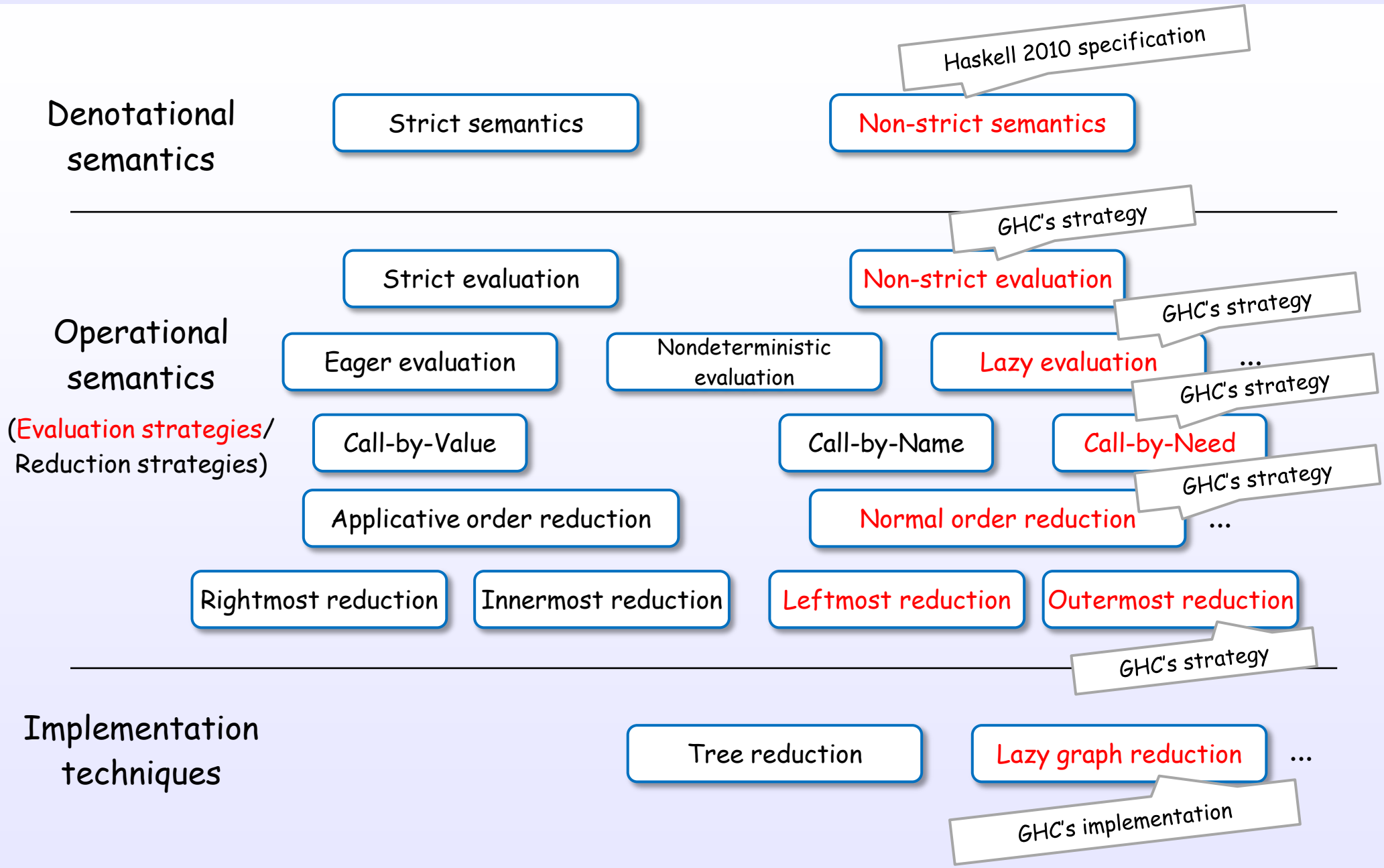
[TAPL, Chapter 3]

References : [1]

Evaluation layer for GHC's Haskell



Evaluation layer for GHC's Haskell



References : [1]

Evaluation strategies and order

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

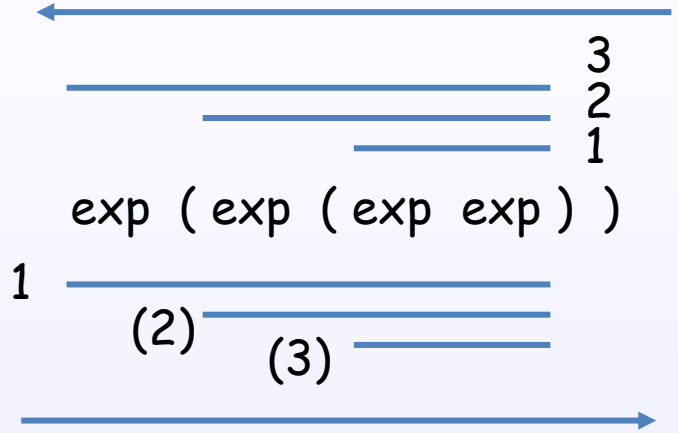
order

[Bird]
[Hutton]

References : [1]

Evaluation strategies and order

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

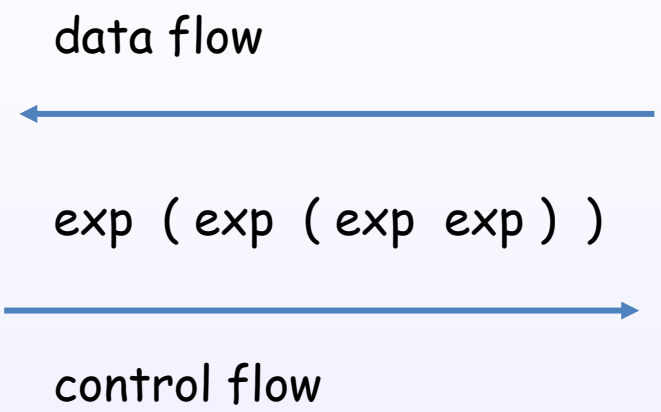


lazy evaluation, call-by-name, call-by-need, outermost reduction, normal order reduction

[Bird]
[Hutton]

References : [1]

Evaluation strategies and order



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

References : [1]

Simple example of typical 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]

References : [1]

Simple example of typical 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

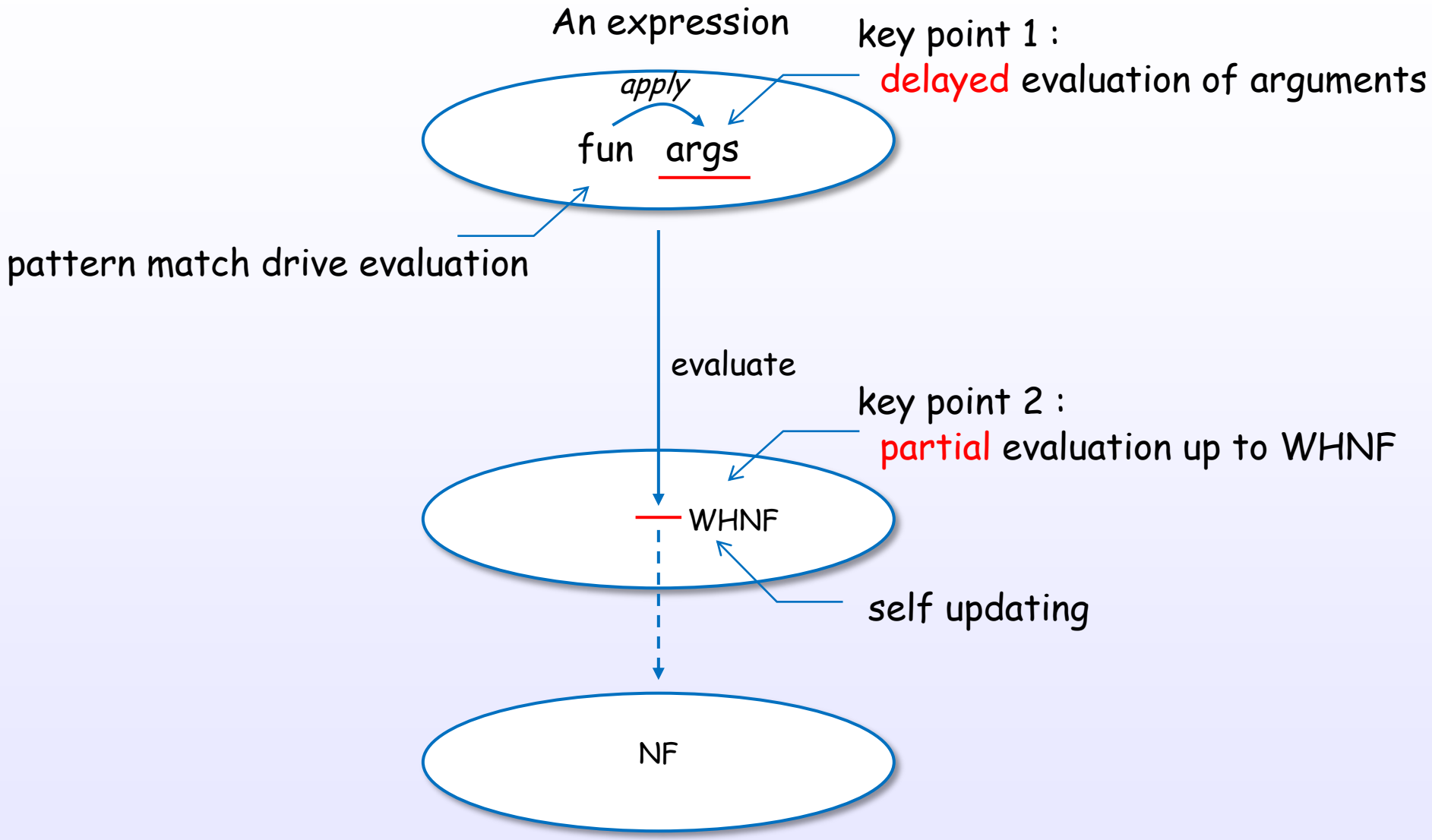
[Bird]
[Hutton]

References : [1]

4. Evaluation

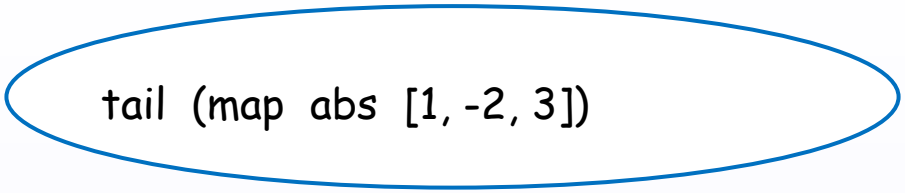
Evaluation in Haskell (GHC)

Key concept of Haskell's lazy evaluation



References : [1]

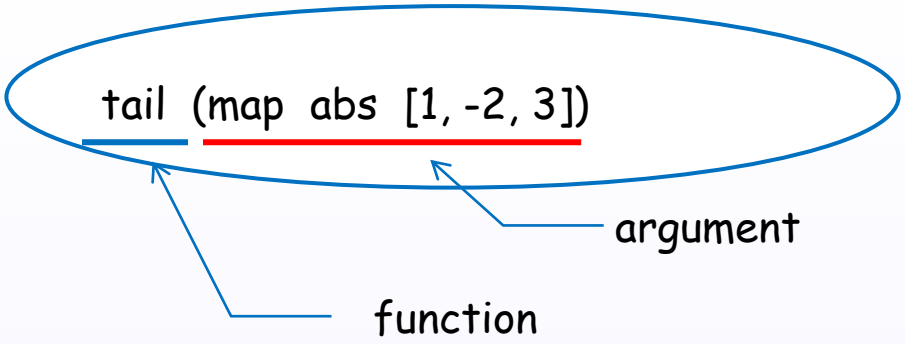
1. Example of GHC's evaluation



Let's evaluate. It's time to magic!

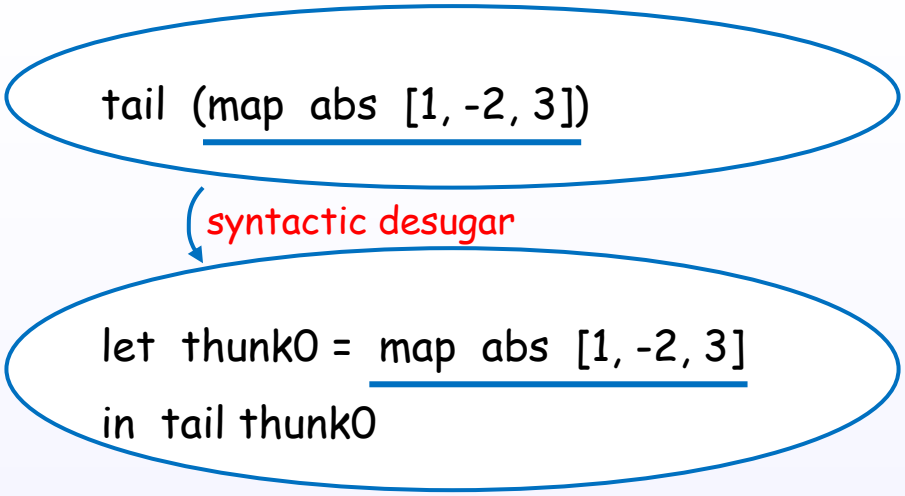
References : [1]

2. How to postpone the evaluation of arguments?



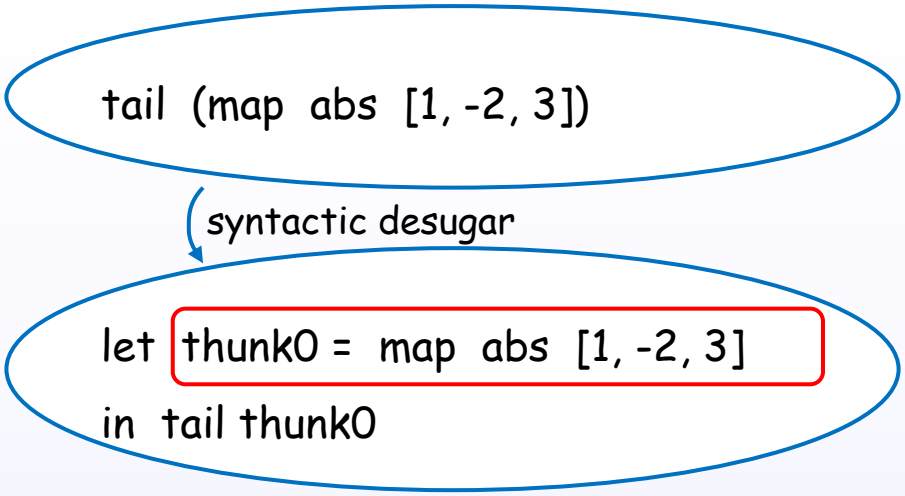
References : [1]

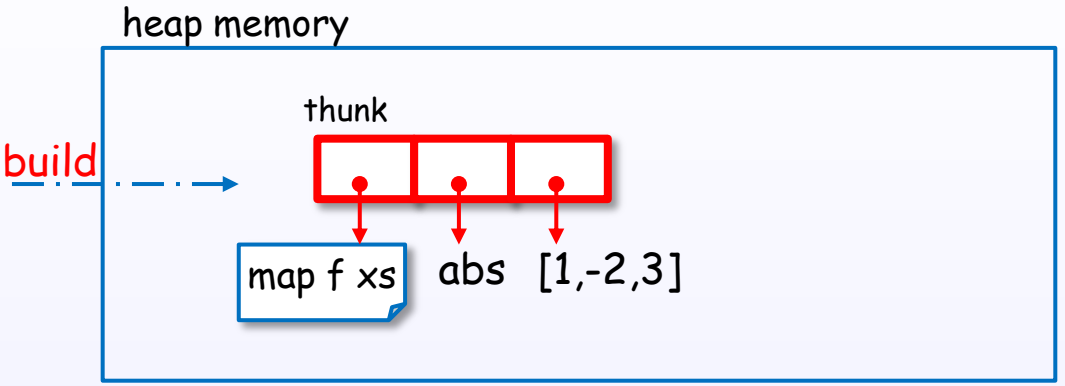
3. GHC transforms internally the expression



References : [1]

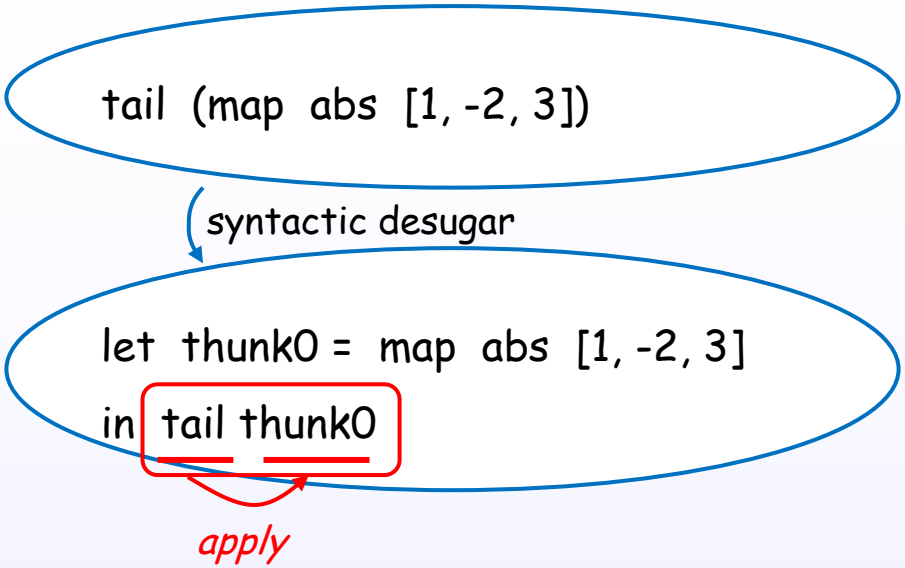
4. a let expression builds a thunk

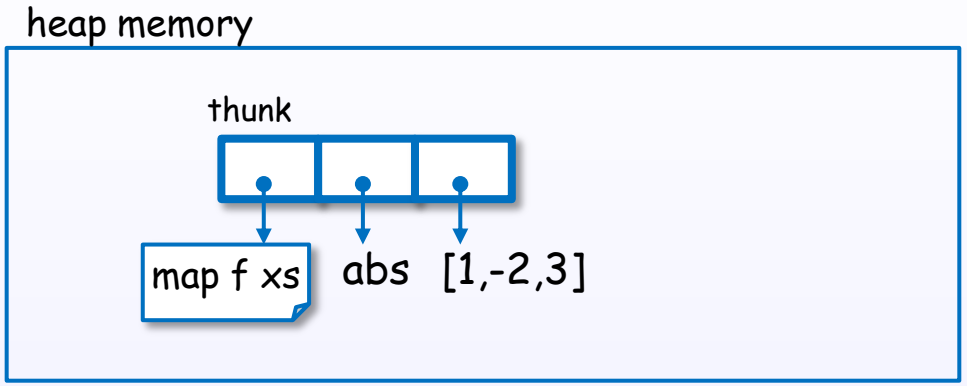




References : [1]

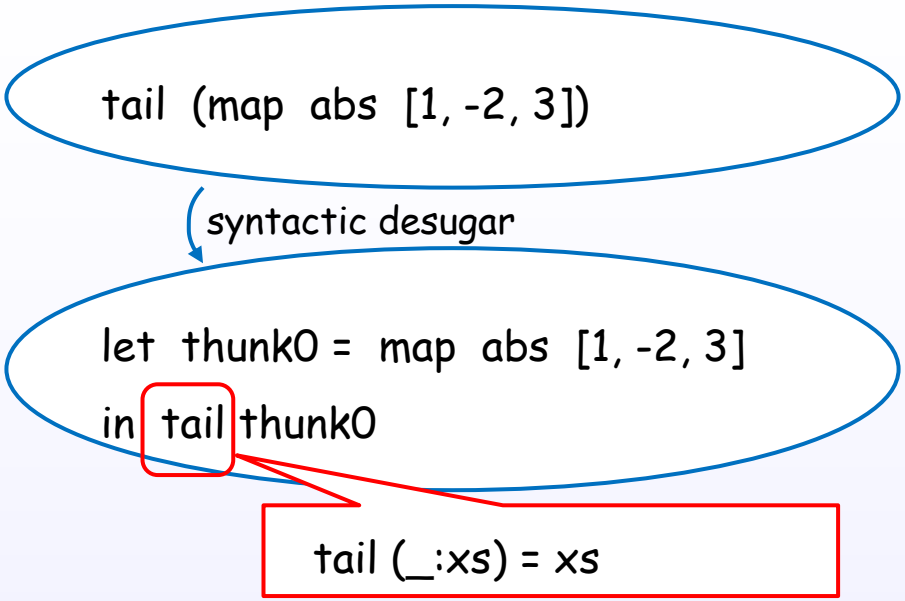
5. function apply to argument

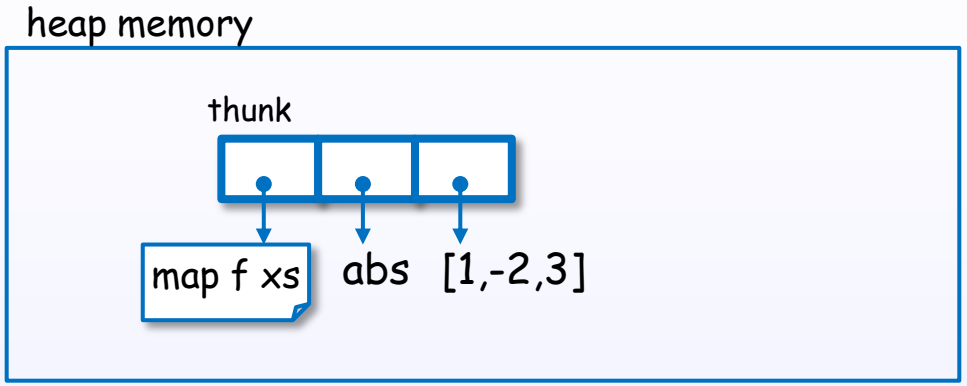




References : [1]

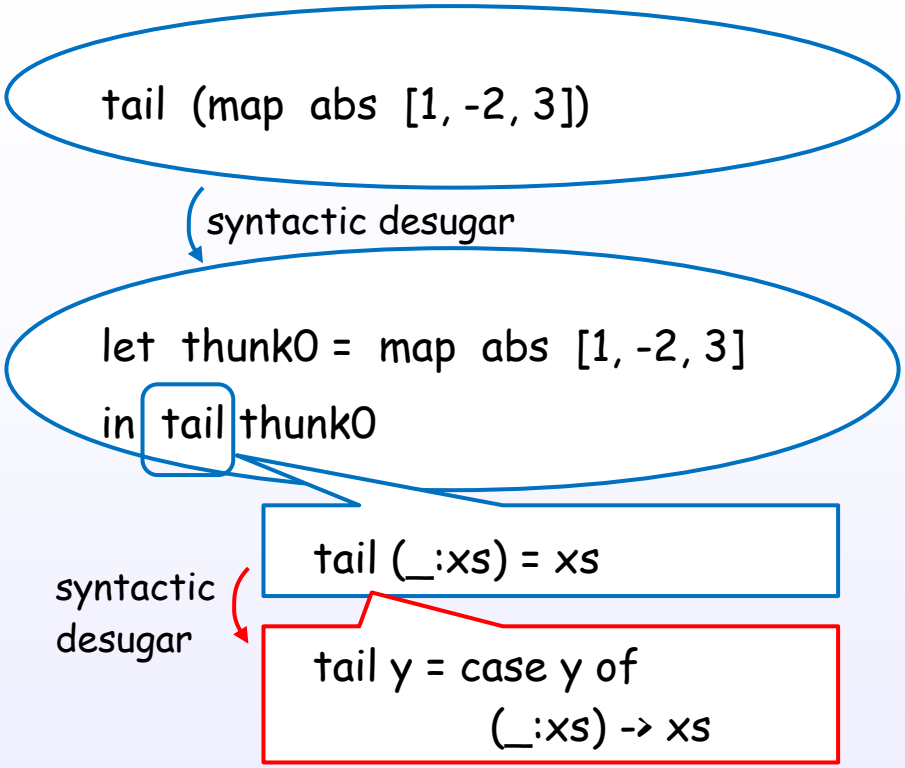
6. tail is defined here

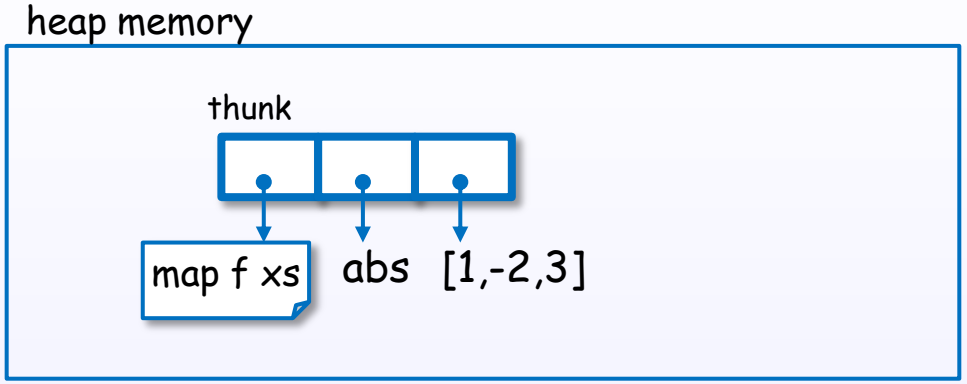




References : [1]

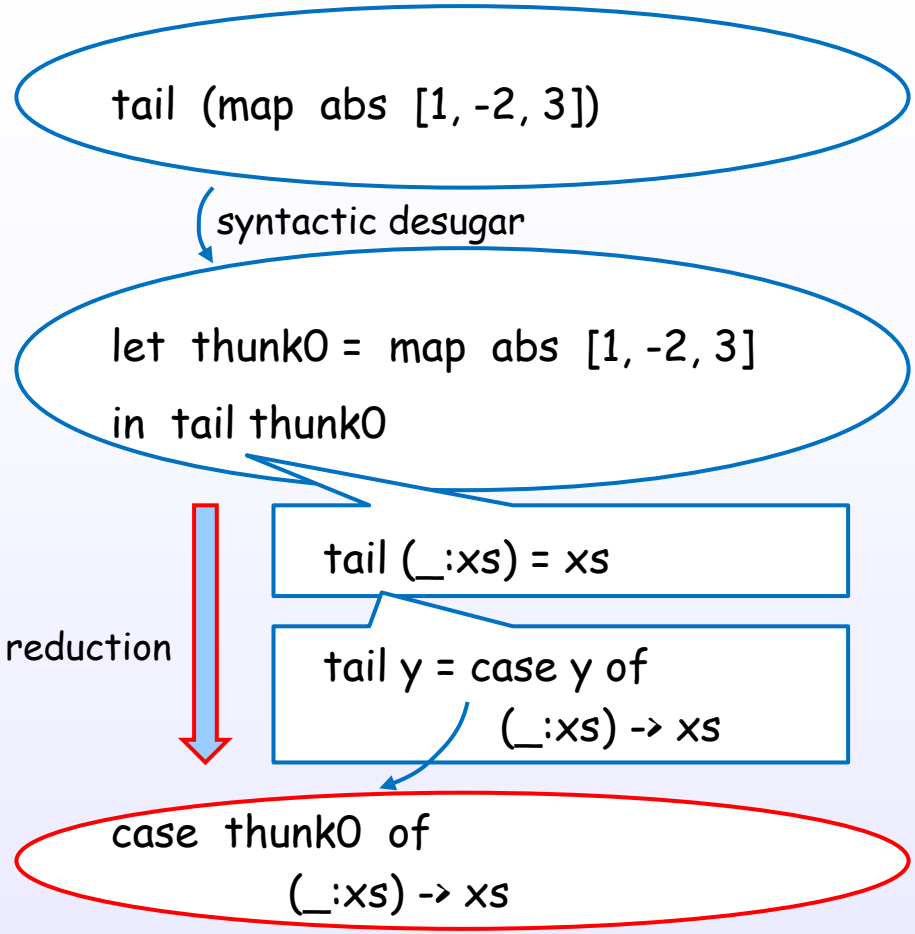
7. function is syntactic sugar

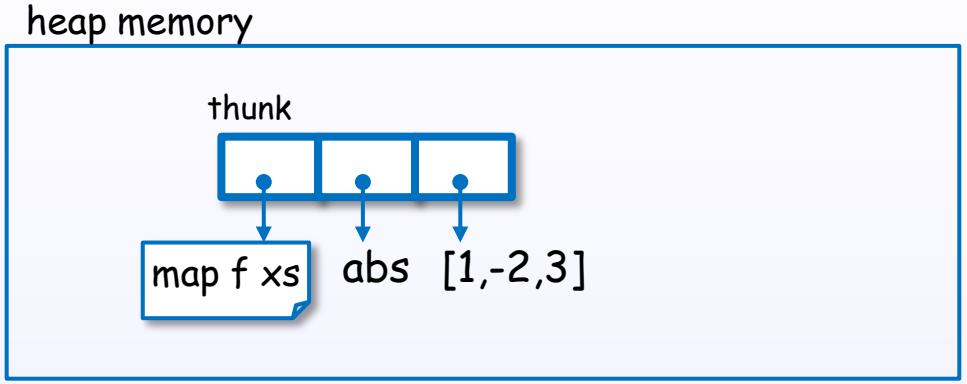




References : [1]

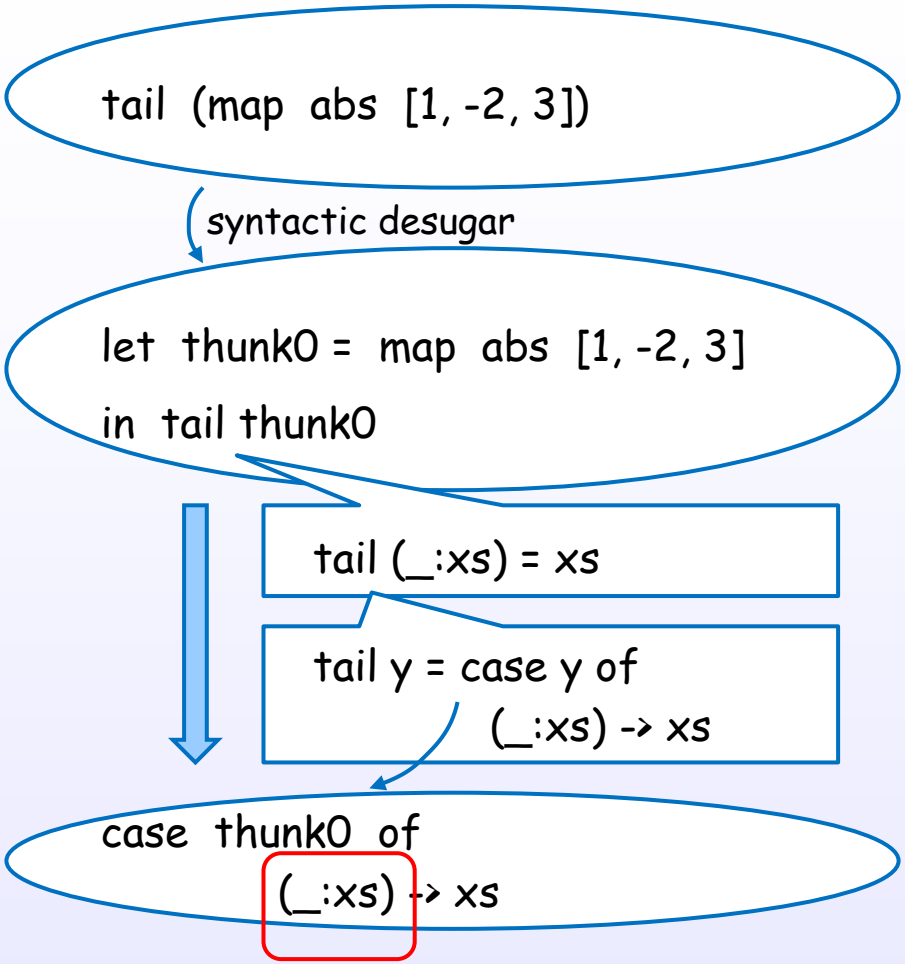
8. substitute function body (beta reduction)

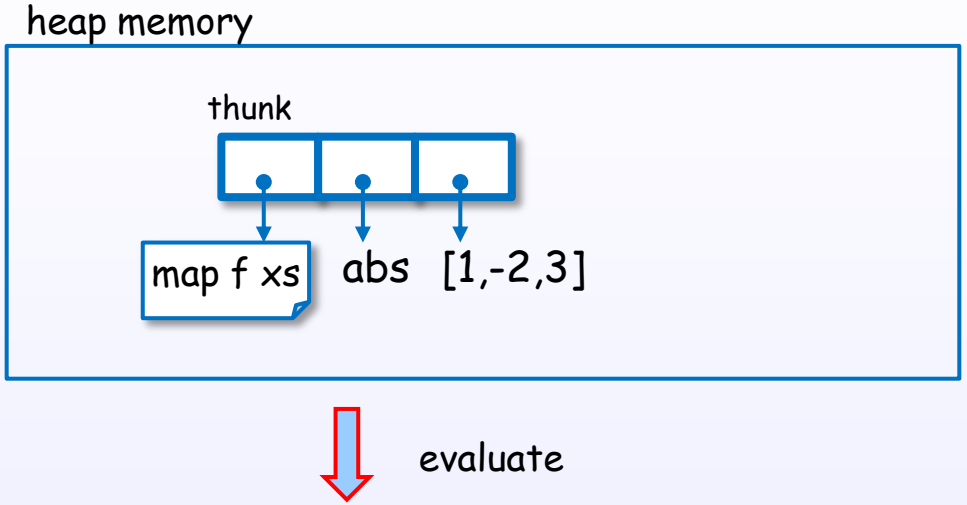




References : [1]

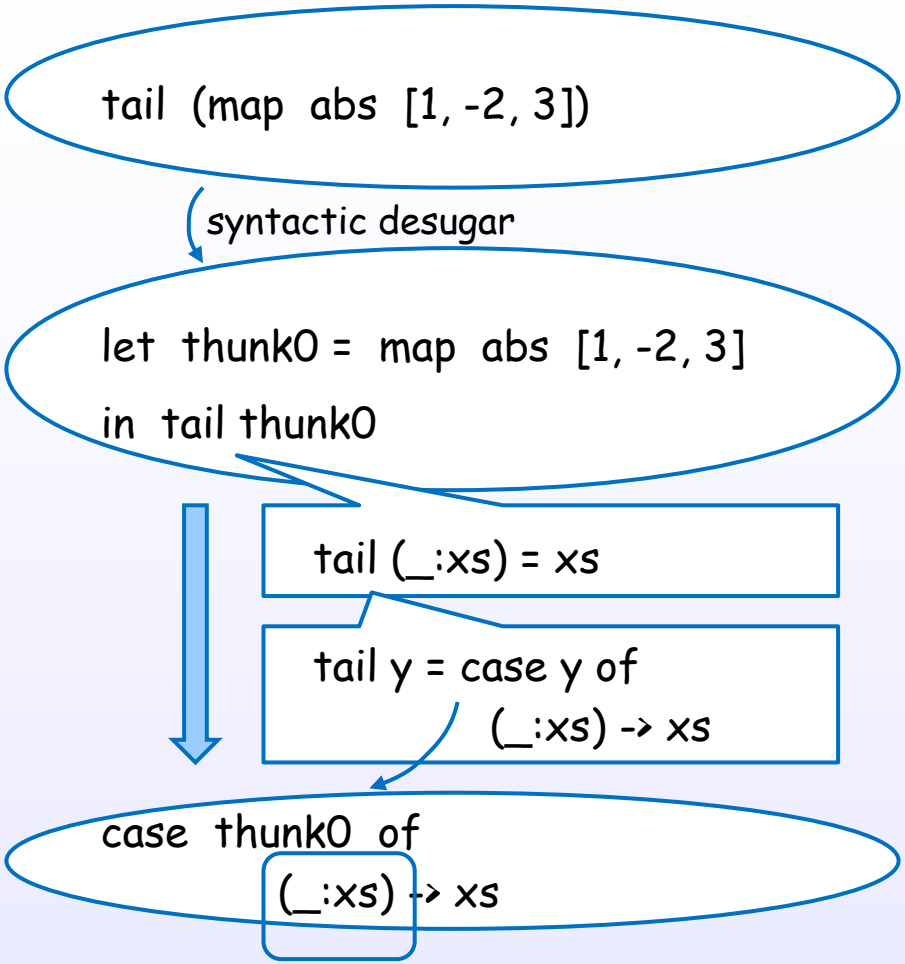
9. case pattern match drive evaluation

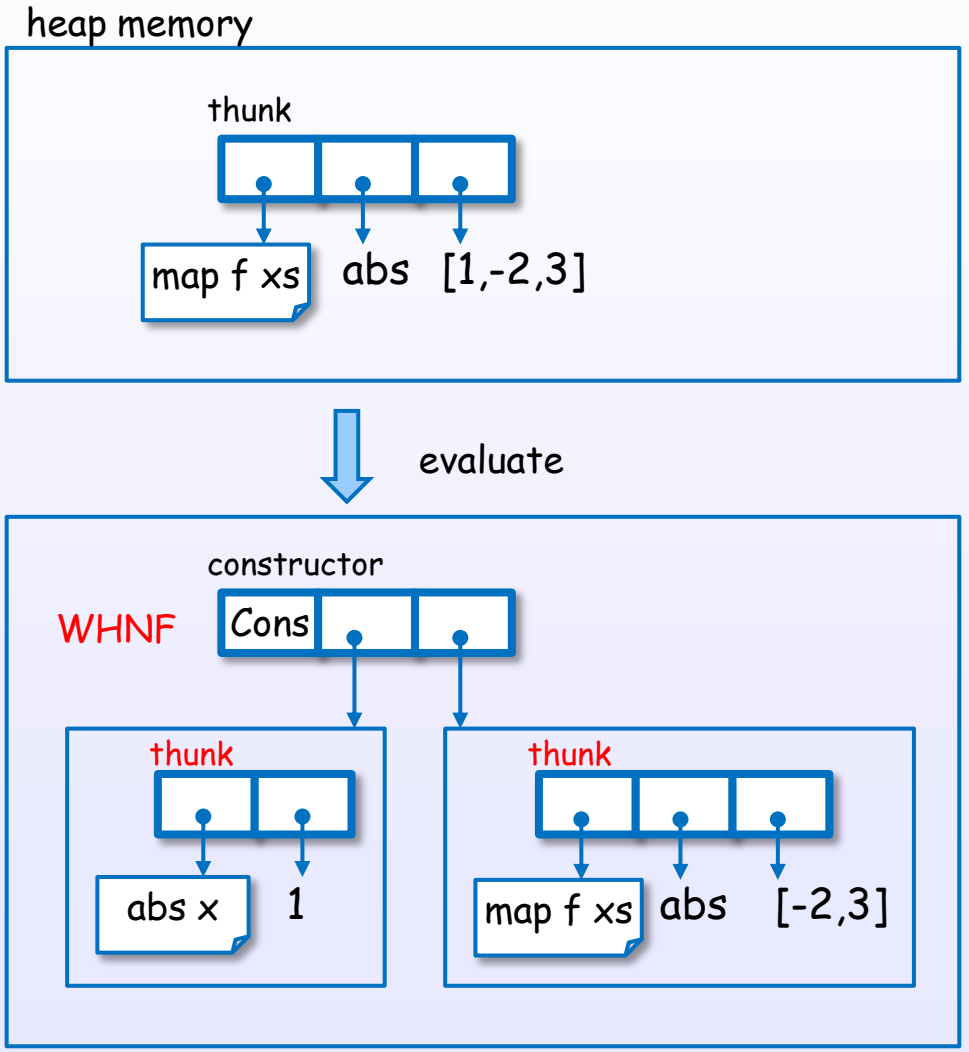




References : [1]

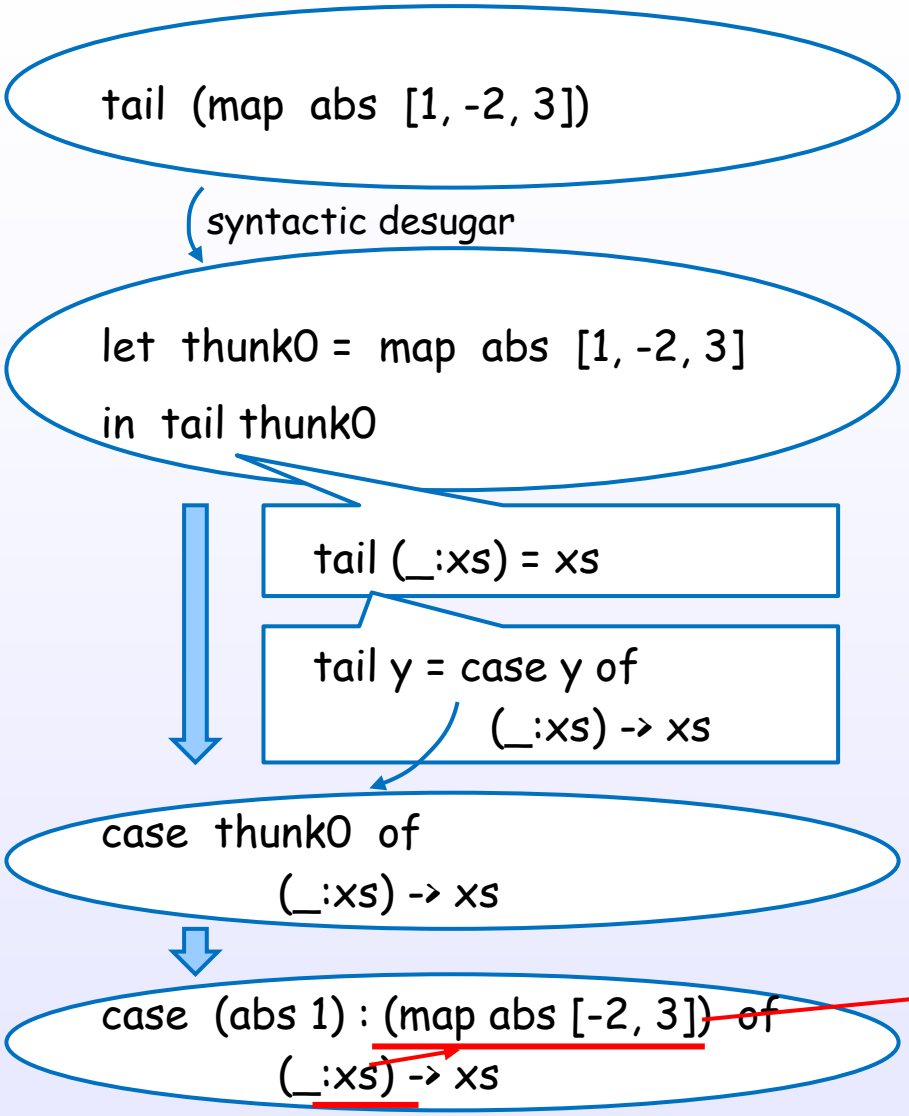
10. but, stop at WHNF

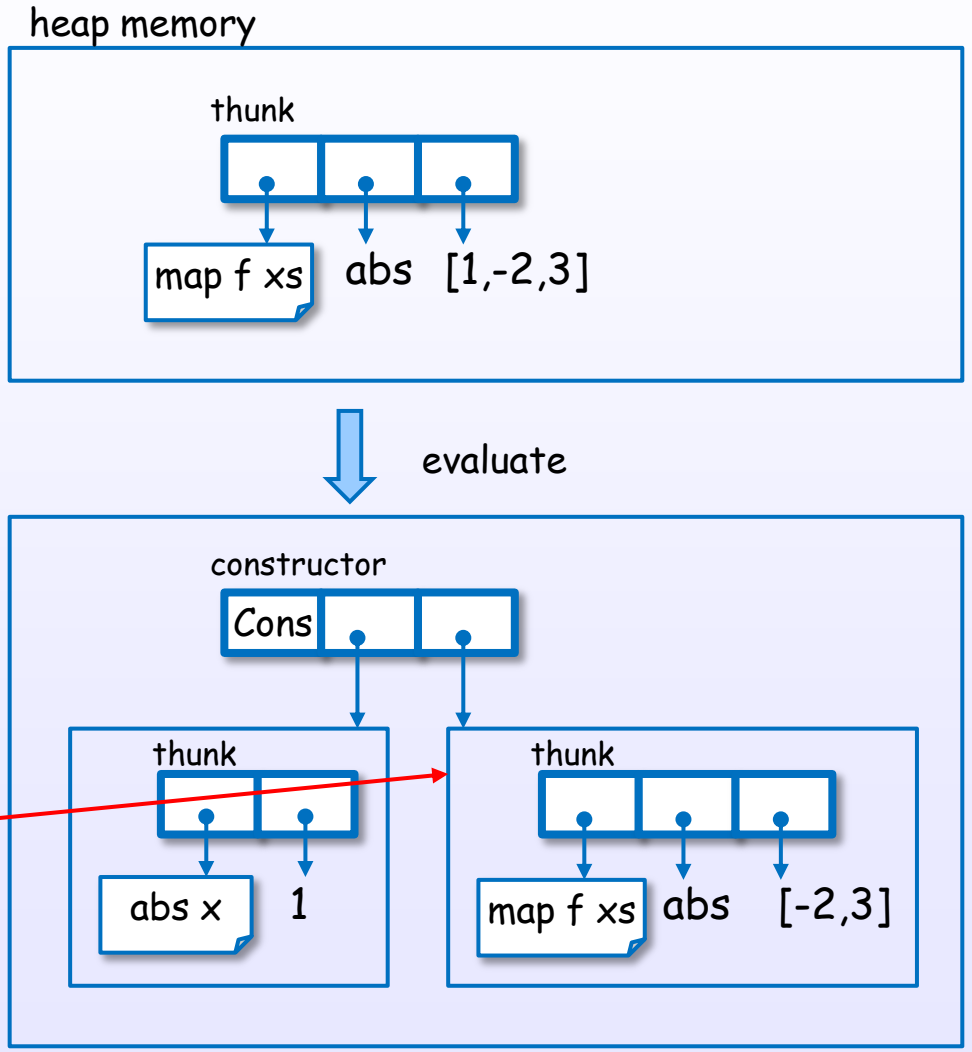




References : [1]

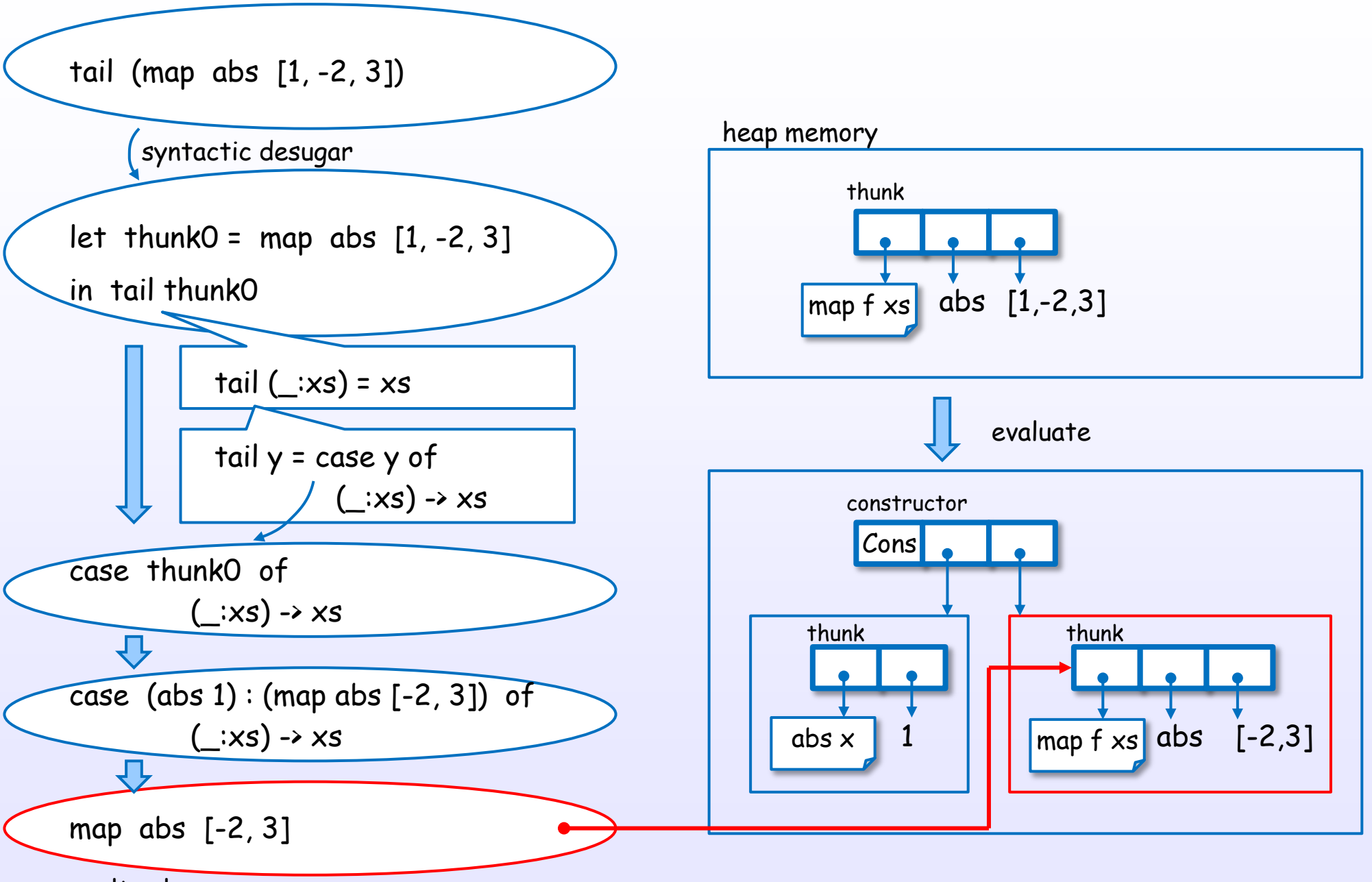
11. bind variables to result





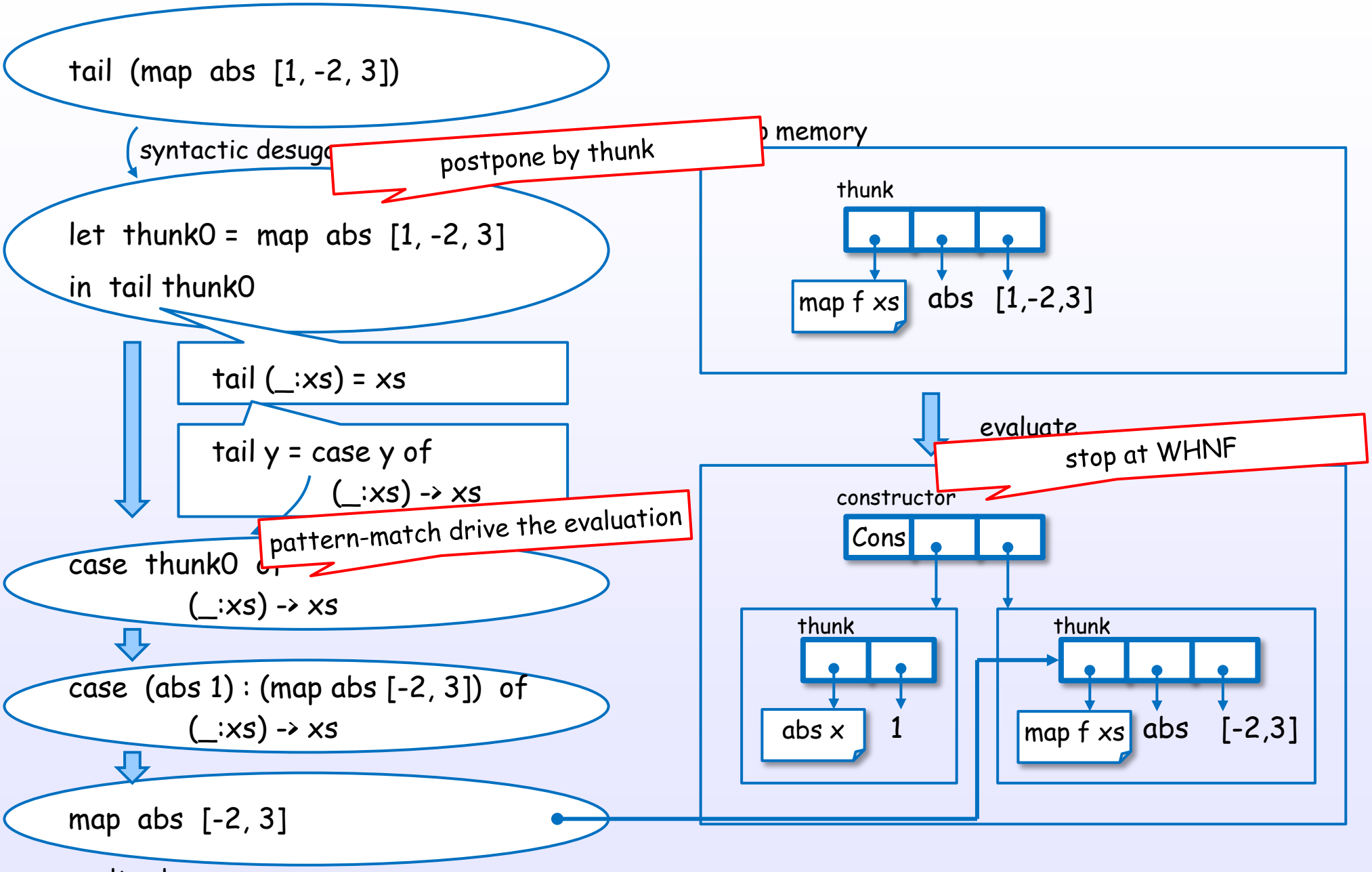
References : [1]

12. return the value



References : [1]

key points



References : [1]

Pattern match

[CIS194]

References : [1]

Pattern match

strict pattern

lazy pattern

case expression
function definition

let bounding pattern
Irrefutable Patterns

[stephen]

References : [1]

4. Evaluation

Examples of evaluation steps

Example of repeat

repeat 1

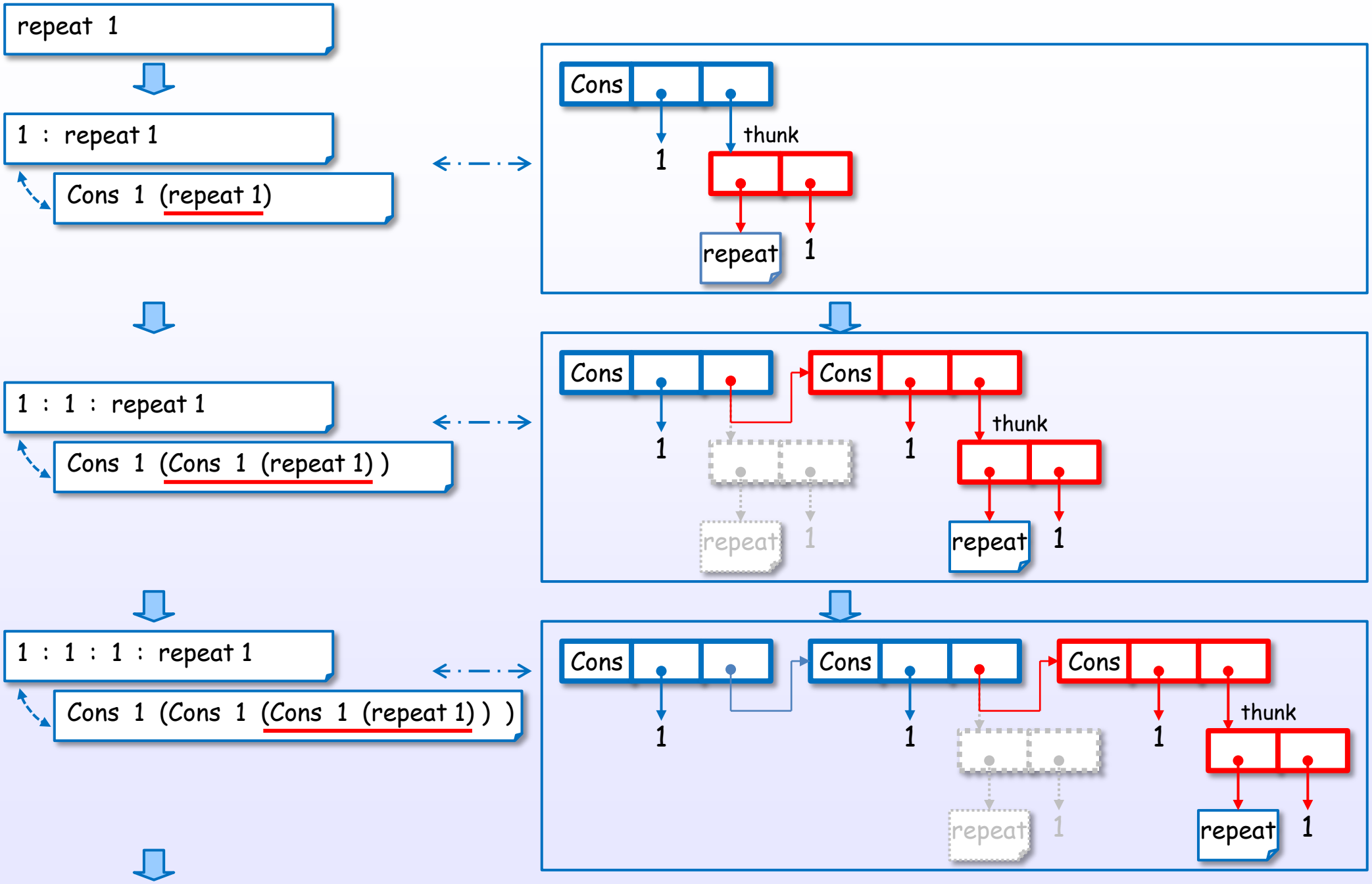
1 : repeat 1

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
```

...

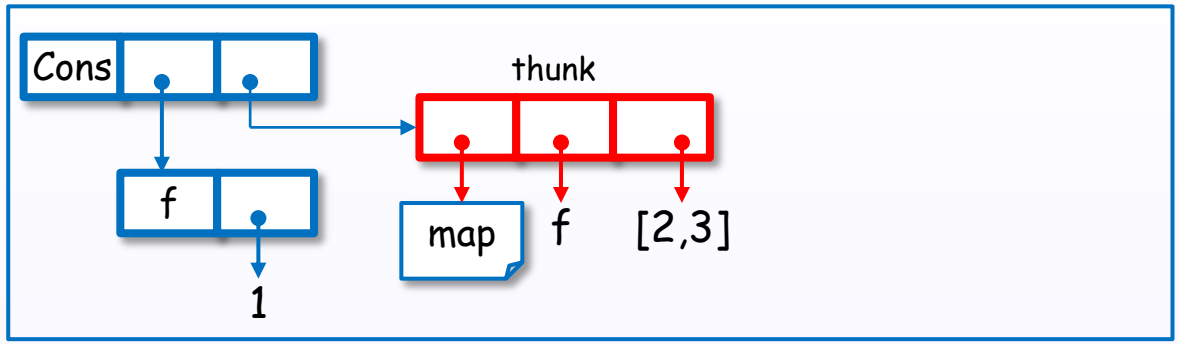
References : [1]

Example of map

map f [1, 2, 3]

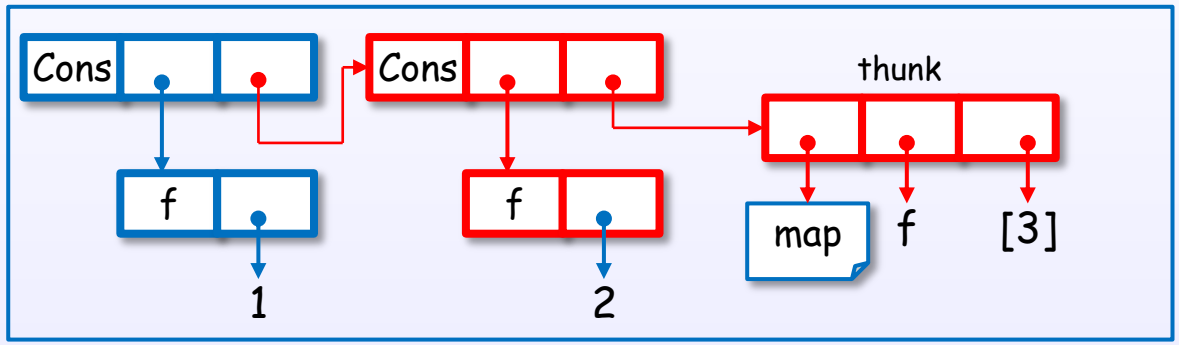
f 1 : map f [2, 3]

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



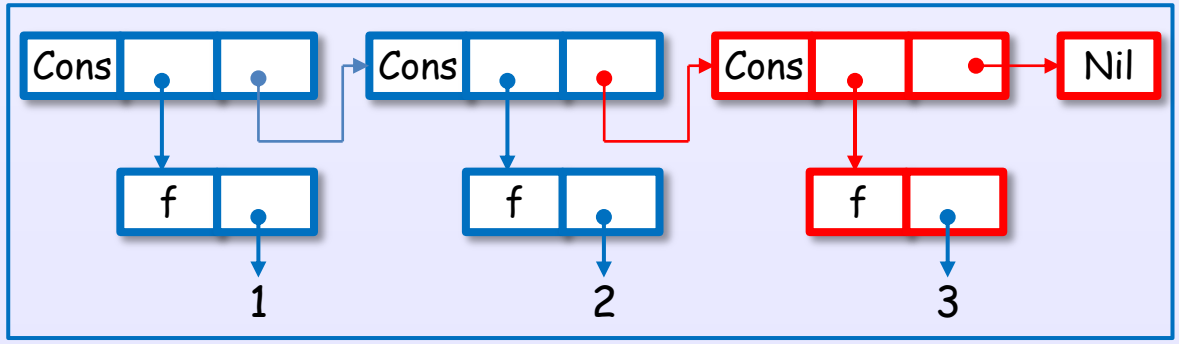
f 1 : f 2 : map f [3]

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))

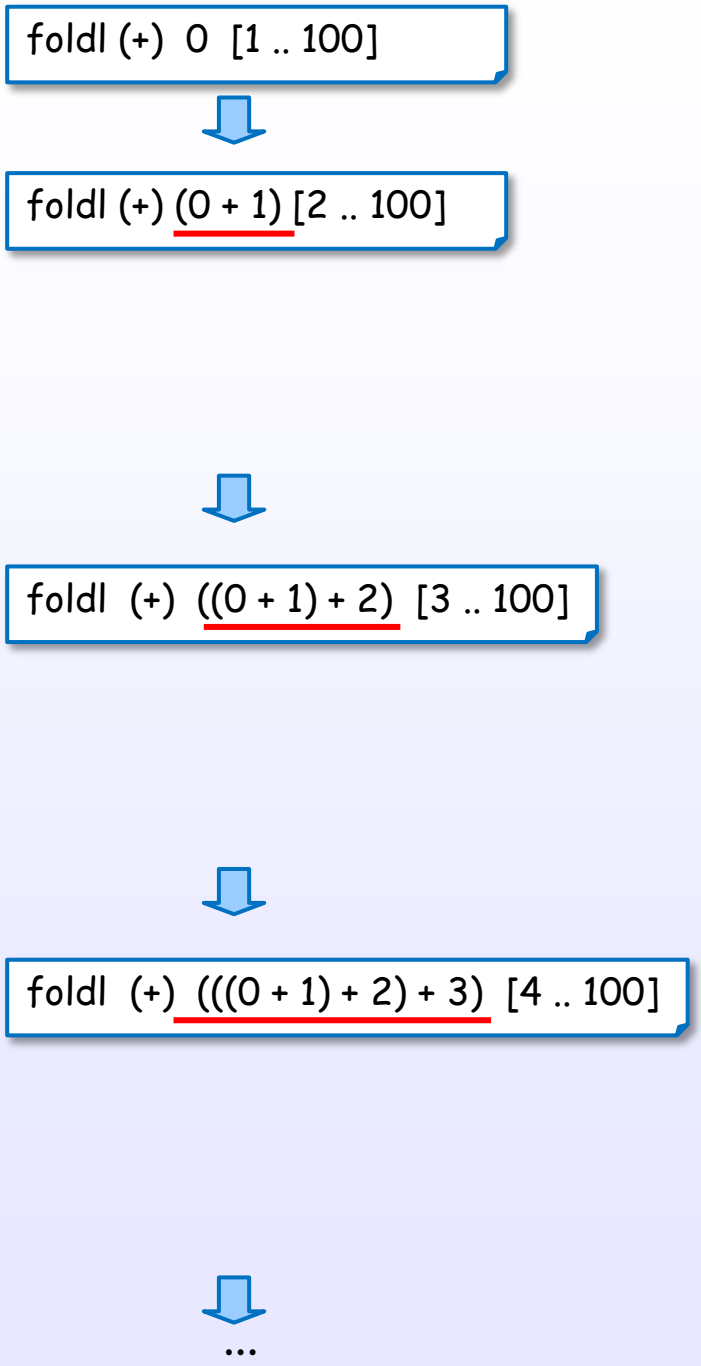


...

...

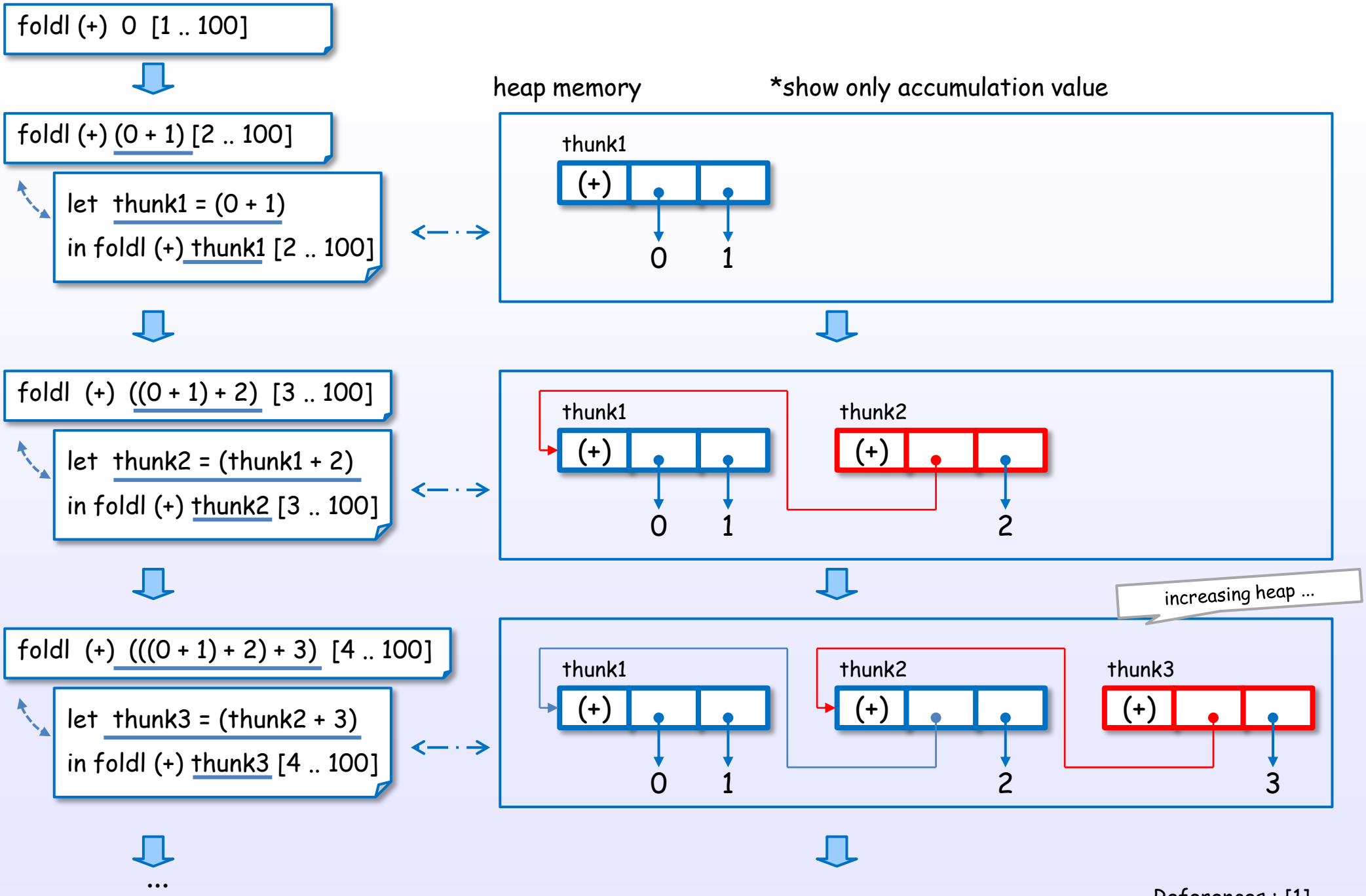
References : [1]

Example of foldl (non-strict)



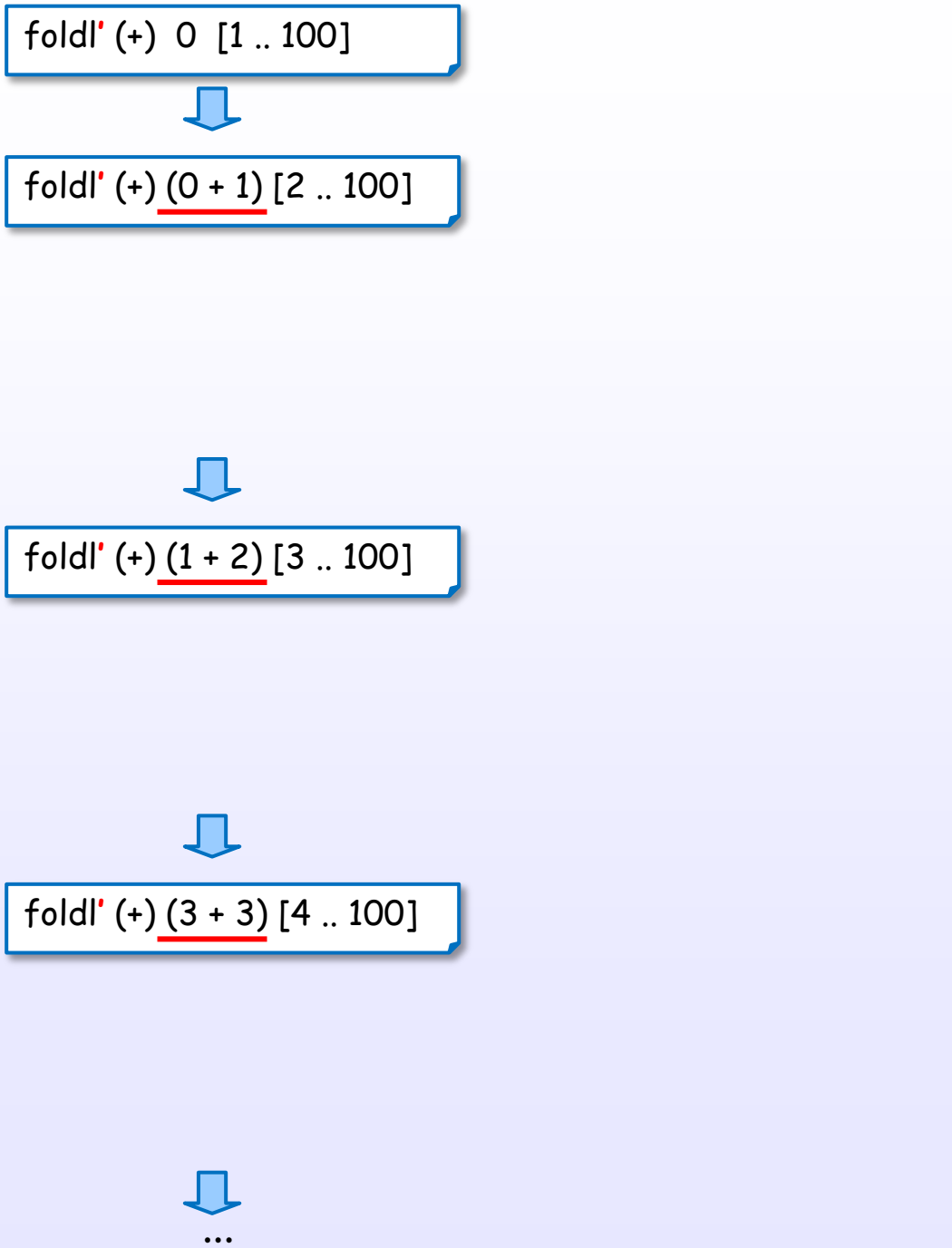
References : [1]

Example of foldl (non-strict)



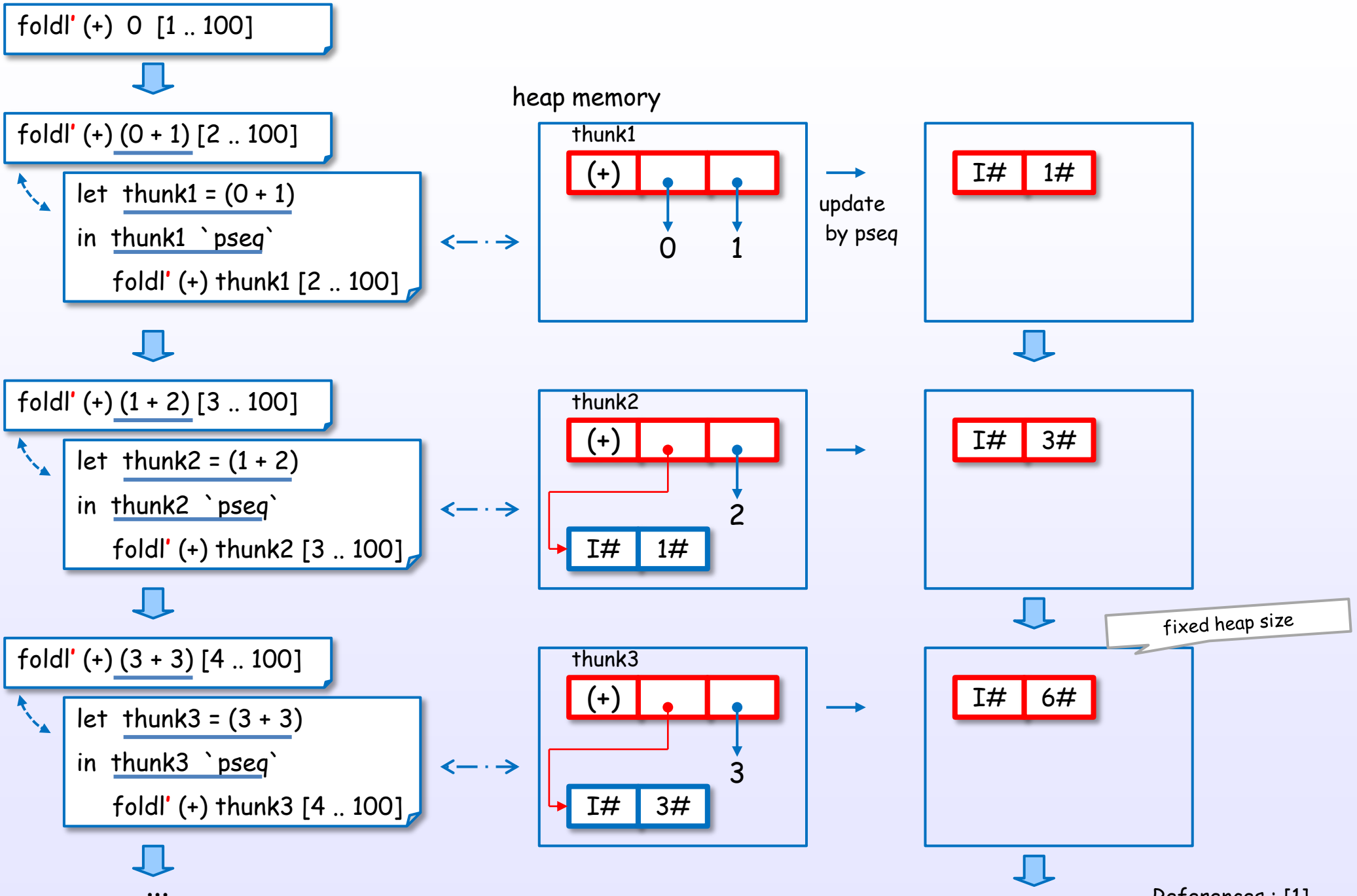
References : [1]

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]

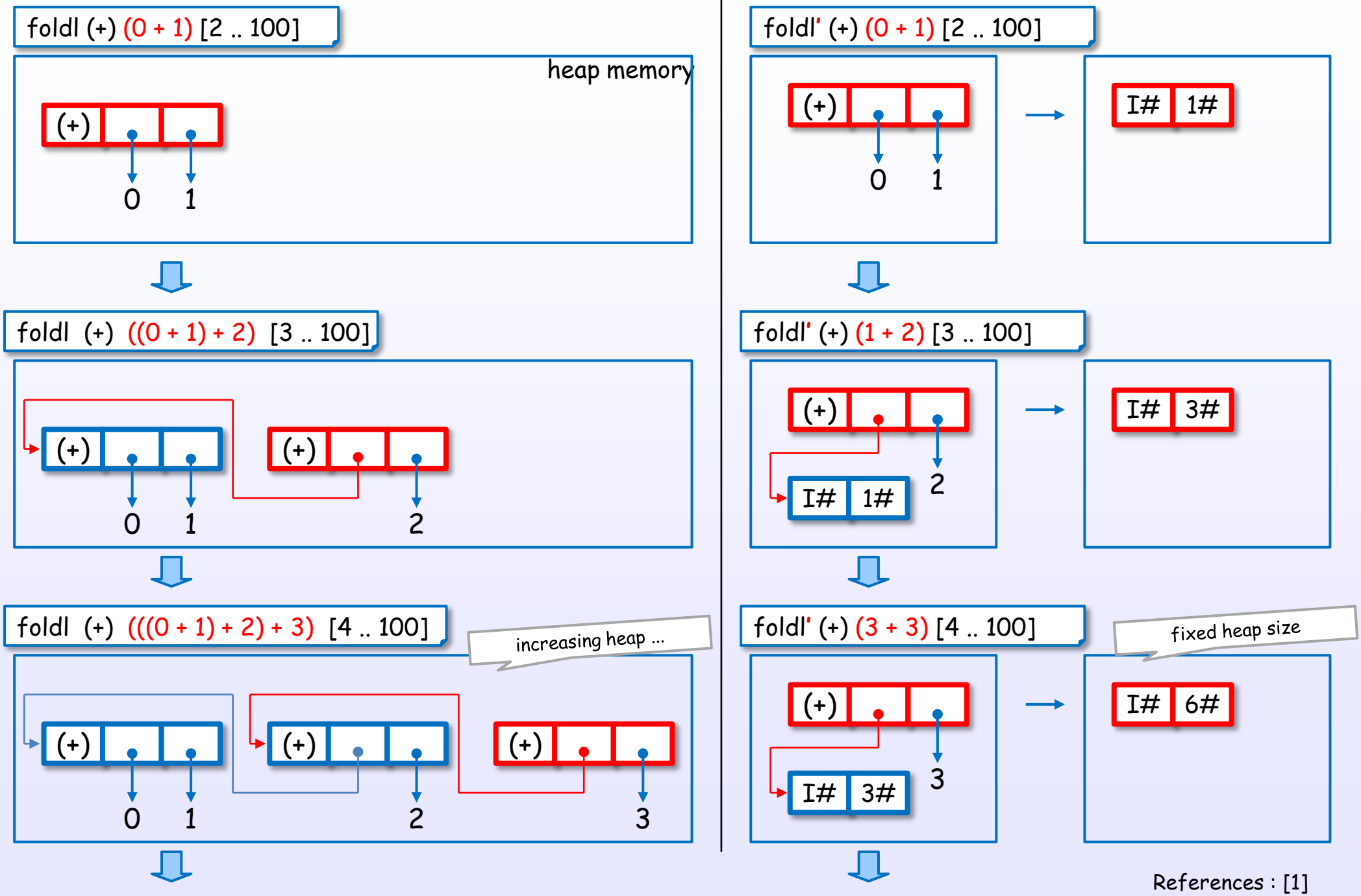
foldl' (+) (0 + 1) [2 .. 100]

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

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

References : [1]

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



References : [1]

Example of nested function

```
take 5 ( map f xs )
```

References : [1]

Example of length

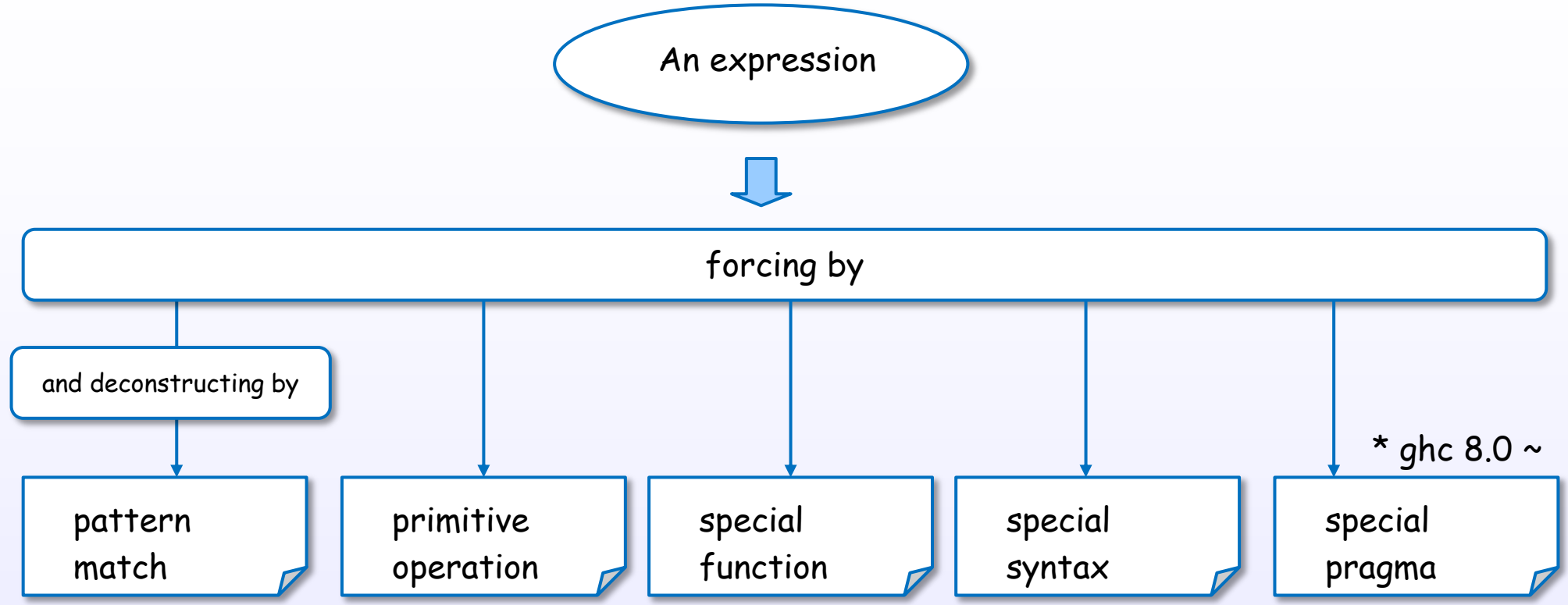
```
xs = map abs [1,2,3]
length  xs
```

References : [1]

4. Evaluation

Controlling the evaluation

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	to WHNF
$!	to NF
rnf	
deepseq	
force	
$!!	[parconc, Ch.2]
pseq	
,...	
	[RWH, Ch.24-25]

[stephen]

[hack.hands]

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

[Bird, Chapter 7]

[CIS194]

References : [1]

Example of the evaluation by special function

seq のobject図イメージ

force

rnf

rwhnf

deepseq のobject図イメージ

References : [1]

Example of the evaluation by special function

表で整理

to WHNF	seq $!	rwhnf	pseq
to NF	force deepseq $!!	rnf	

References : [1]

Example of the evaluation by special syntax

special syntax

```
{-# LANGUAGE BangPatterns #-}

f !xs = g xs
```

BangPattern

```
{-# LANGUAGE BangPatterns #-}

data ...
```

[RWH, Ch.25]

[stephen]

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]

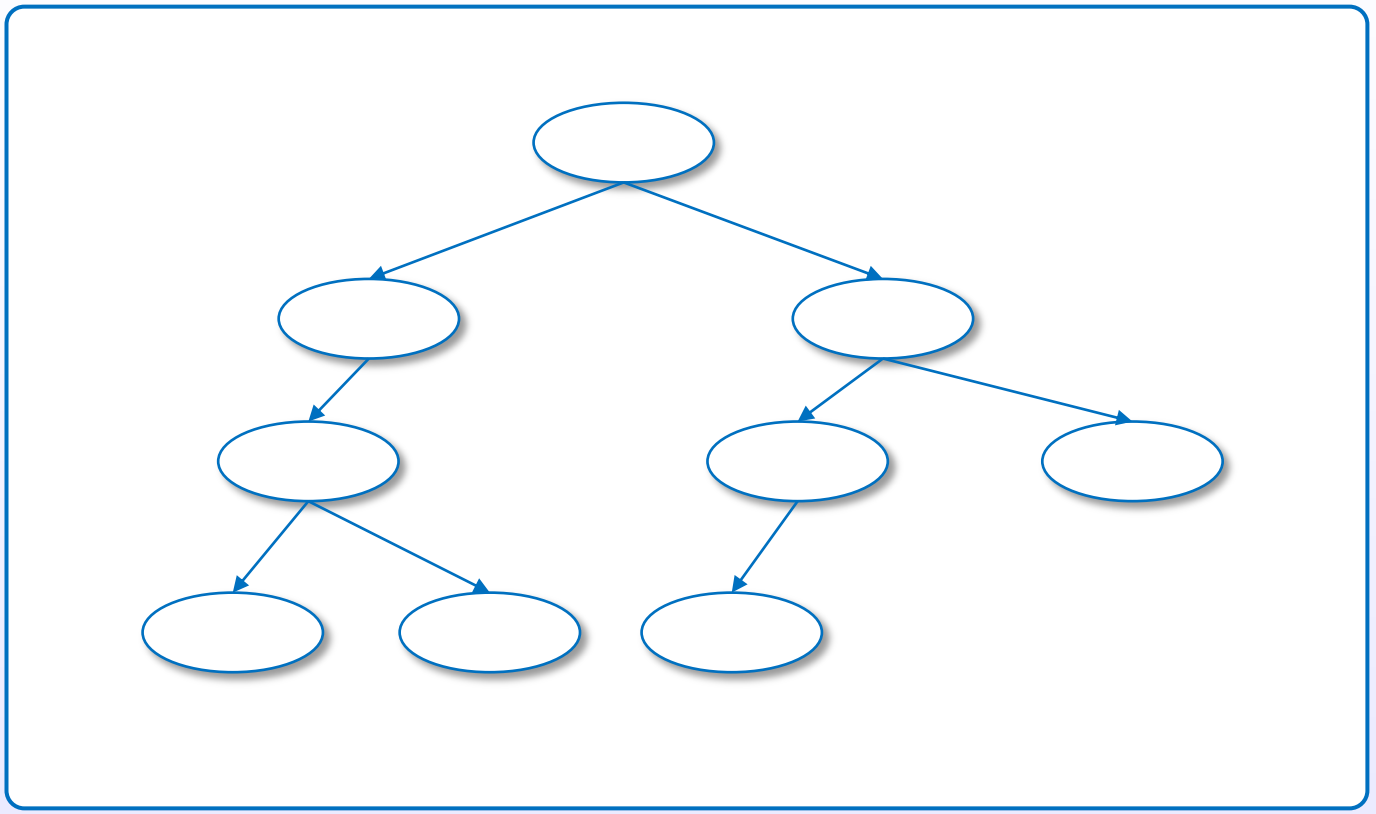
5. Implementation of evaluator

5. Implementation of evaluator

Lazy graph reduction

Tree

AST represents an expression

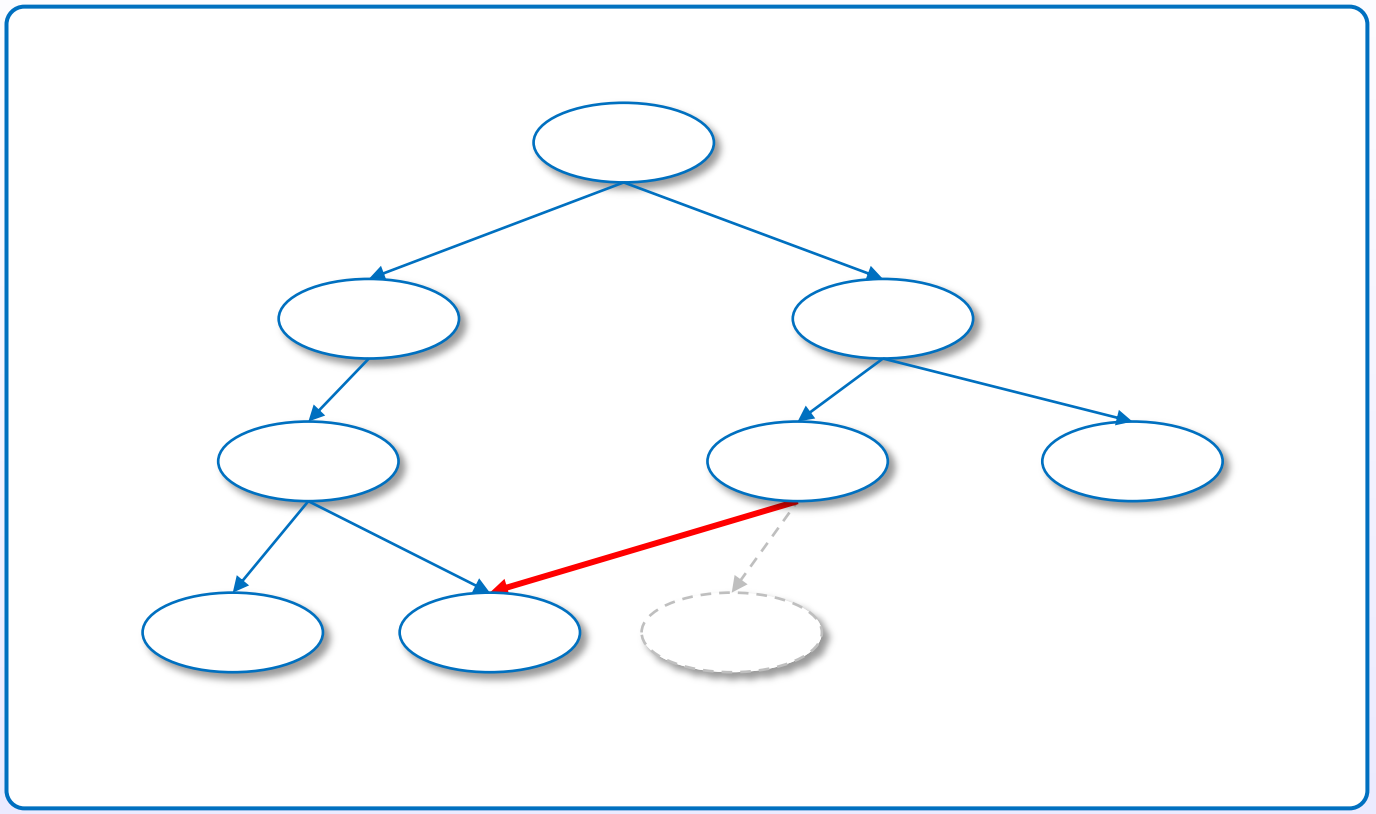


Stack base

References : [1]

Graph

Share the term, looped
not Tree, but Graph



Heap base

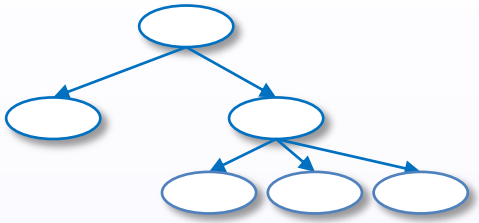
[Terei]

[hack.hands]

[CIS194]

References : [1]

Tree and graph reduction



Tree reduction

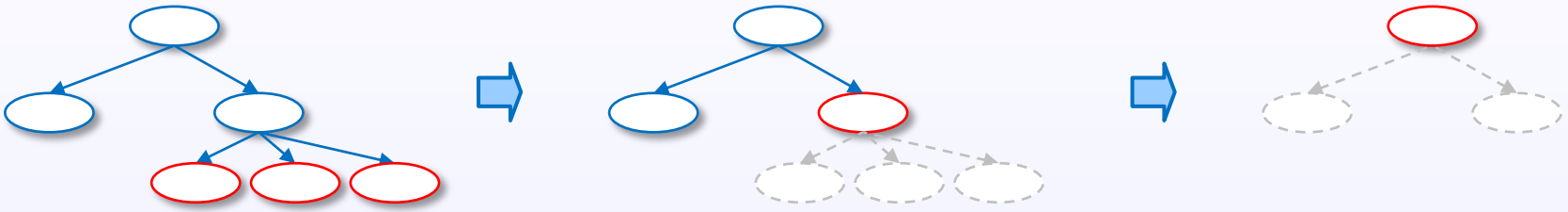
Graph reduction

copy arguments

share arguments by pointers

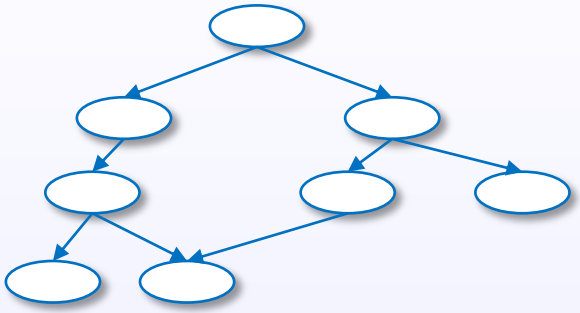
References : [1]

Graph reduction



References : [1]

Graph reduction and lazy



References : [1]

5. Implementation of evaluator

STG-machine

Abstract machine

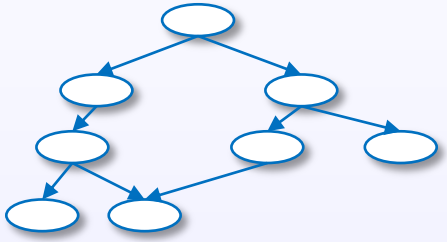
References : [1]

Layer

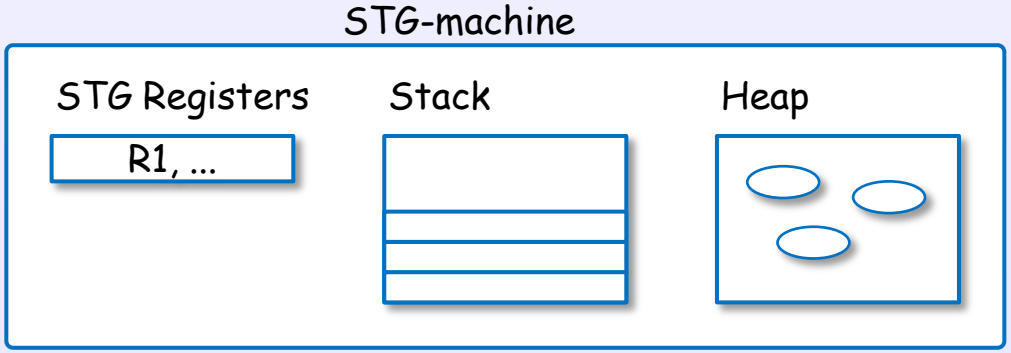
Haskell code

take 5 [1..10]

Internal representation
by graph

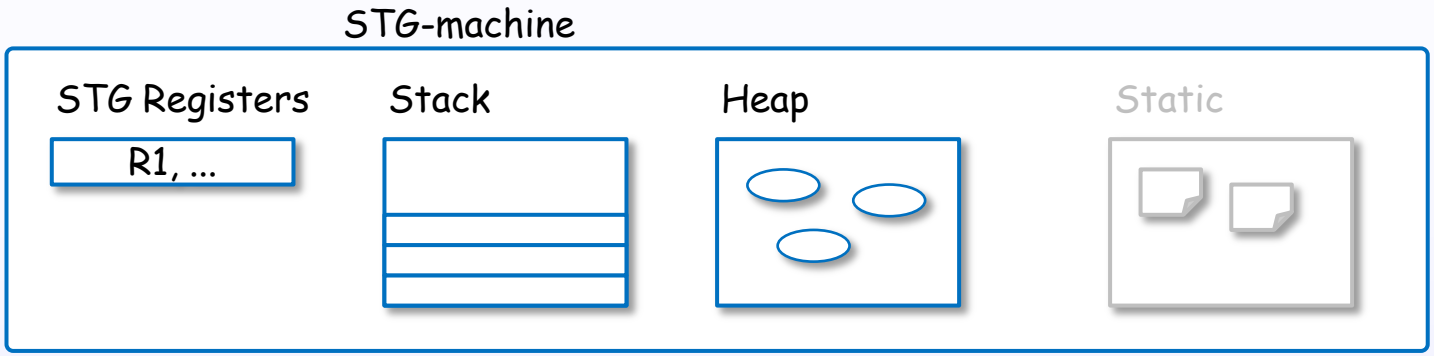


Evaluation (execution, reduction)
by STG-machine



References : [1]

STG-machine

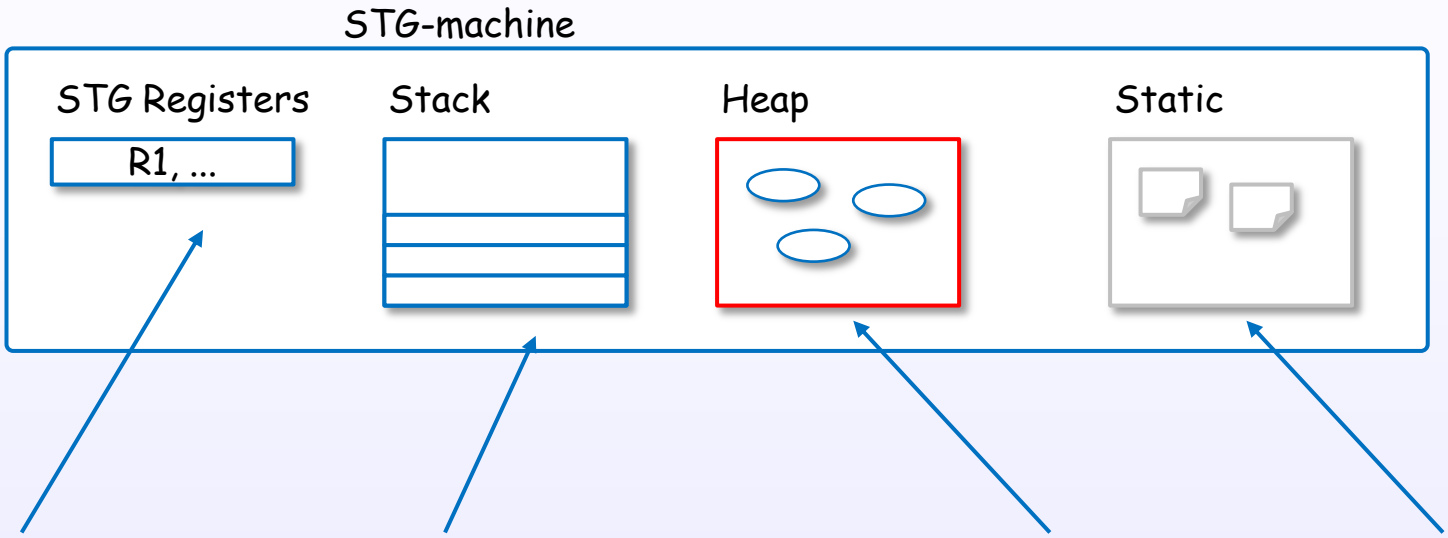


STG-machine is abstraction machine
which is defined by operational semantics.

STG-machine efficiently performs lazy graph reduction.

References : [1]

STG-machine



mainly using for
call/return convention
various control

mainly using for
nest continuation
argument passing

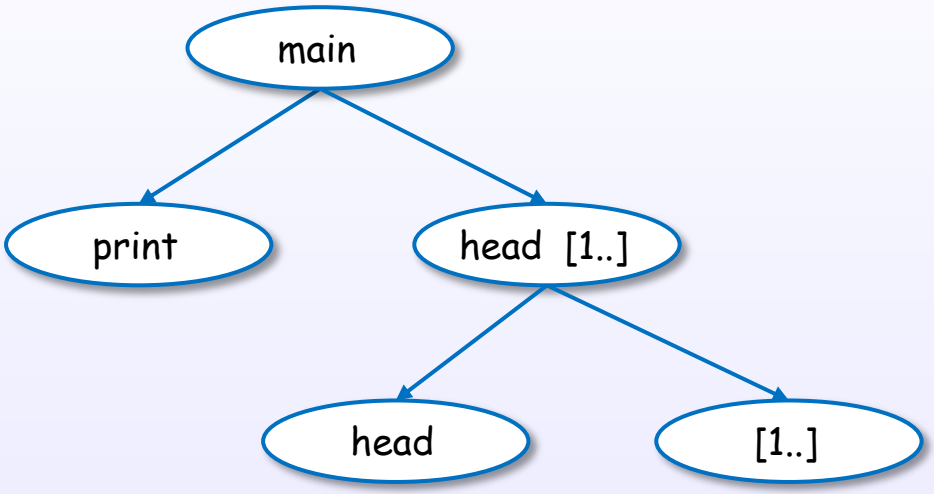
mainly using for
allocating thunks

mainly using for
code
static closure

References : [1]

Mapping the graph to the code

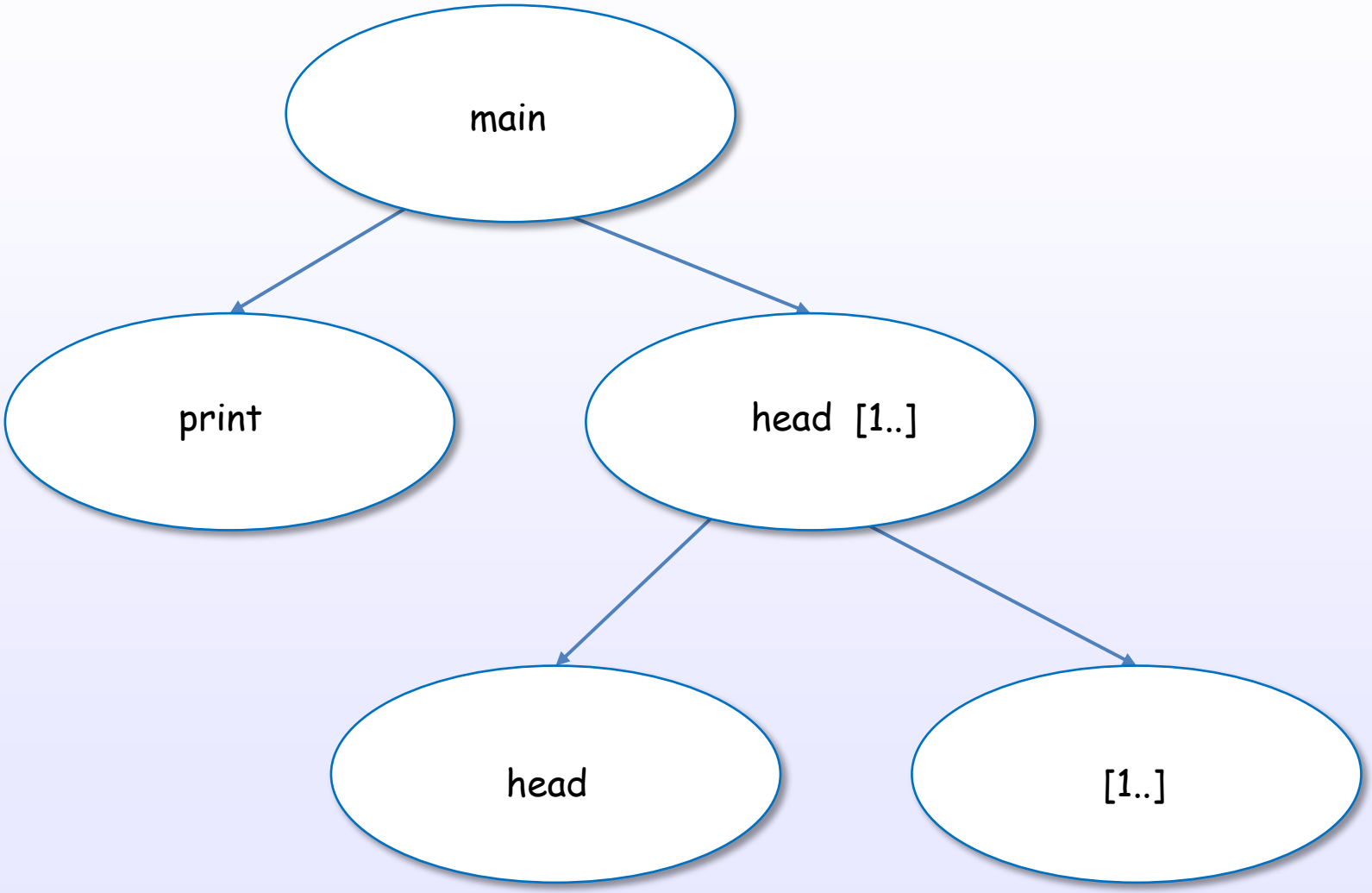
```
main = print (head [1..])
```



References : [1]

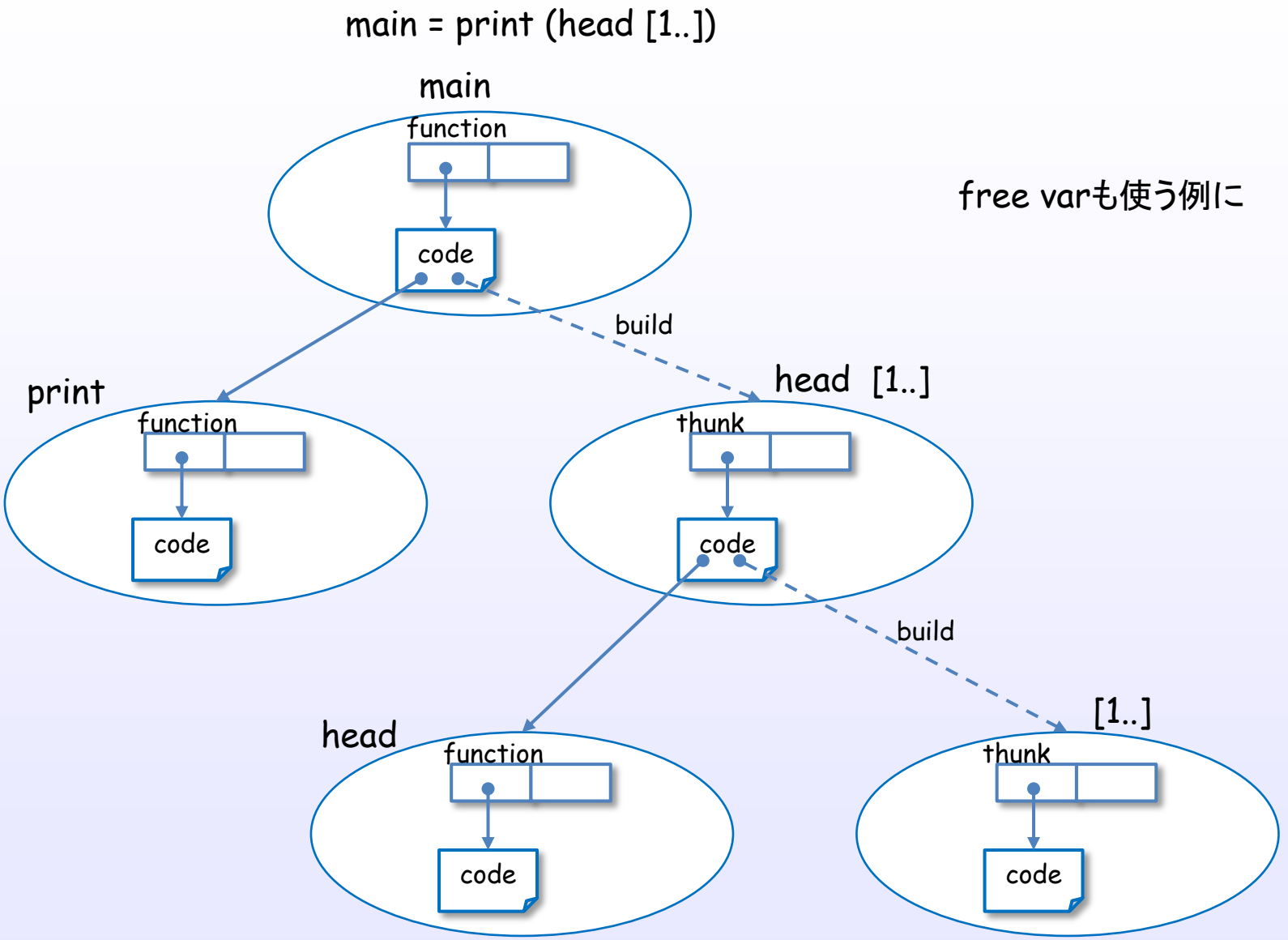
Mapping the graph to the code

main = print (head [1..])



References : [1]

Mapping the graph to the code

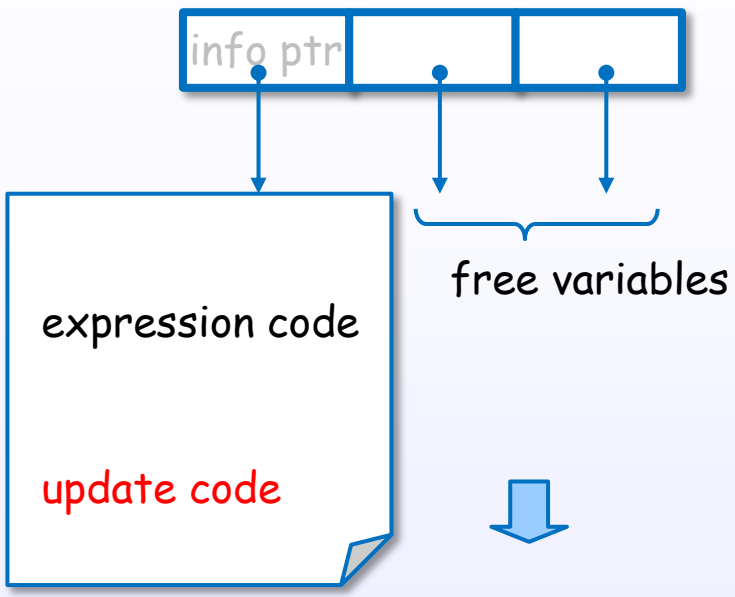


free varも使う例に

References : [1]

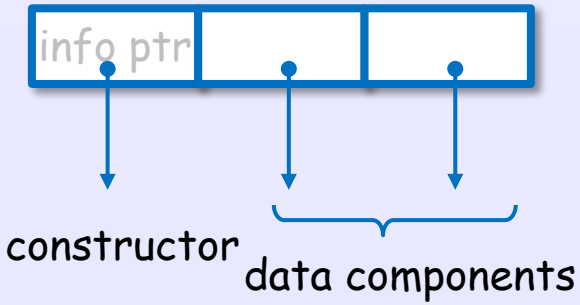
Self-updating model

a thunk



update
(replace
myself to
result value)

a data value

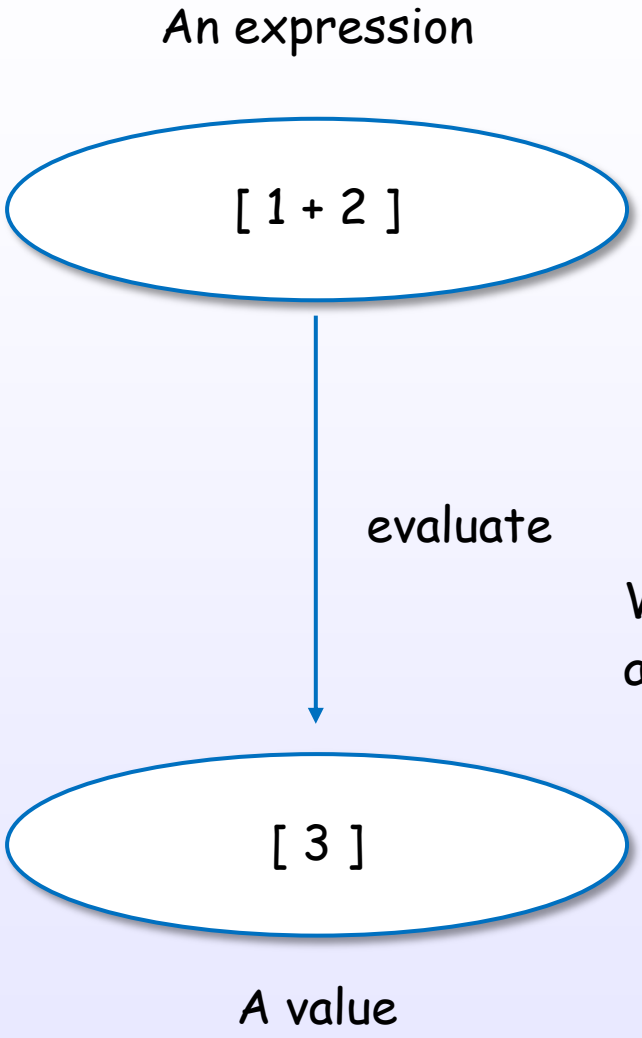


References : [1]

6. Semantics

Bottom

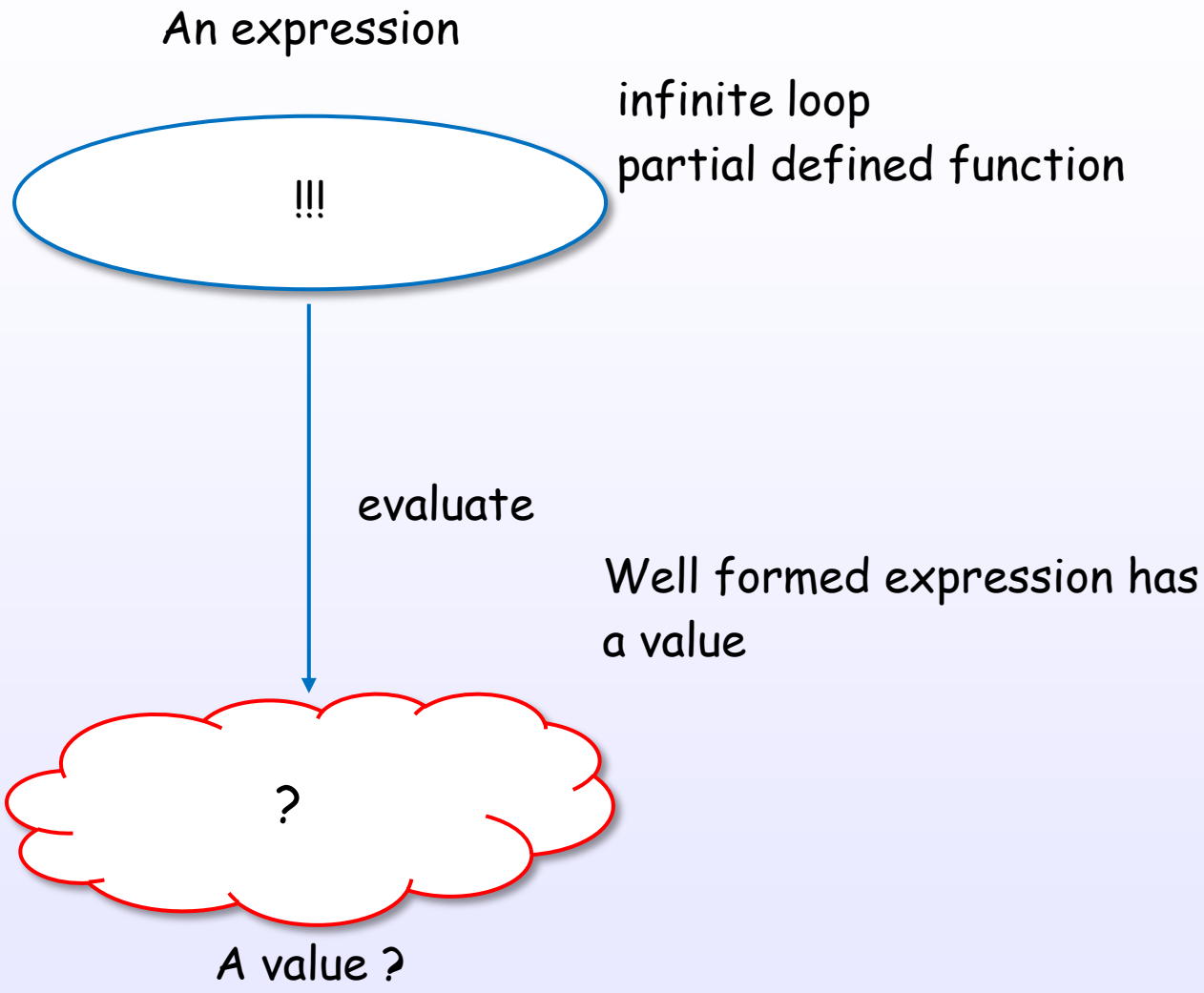
Well formed expression has a value



[Bird, Chapter 2]

References : [1]

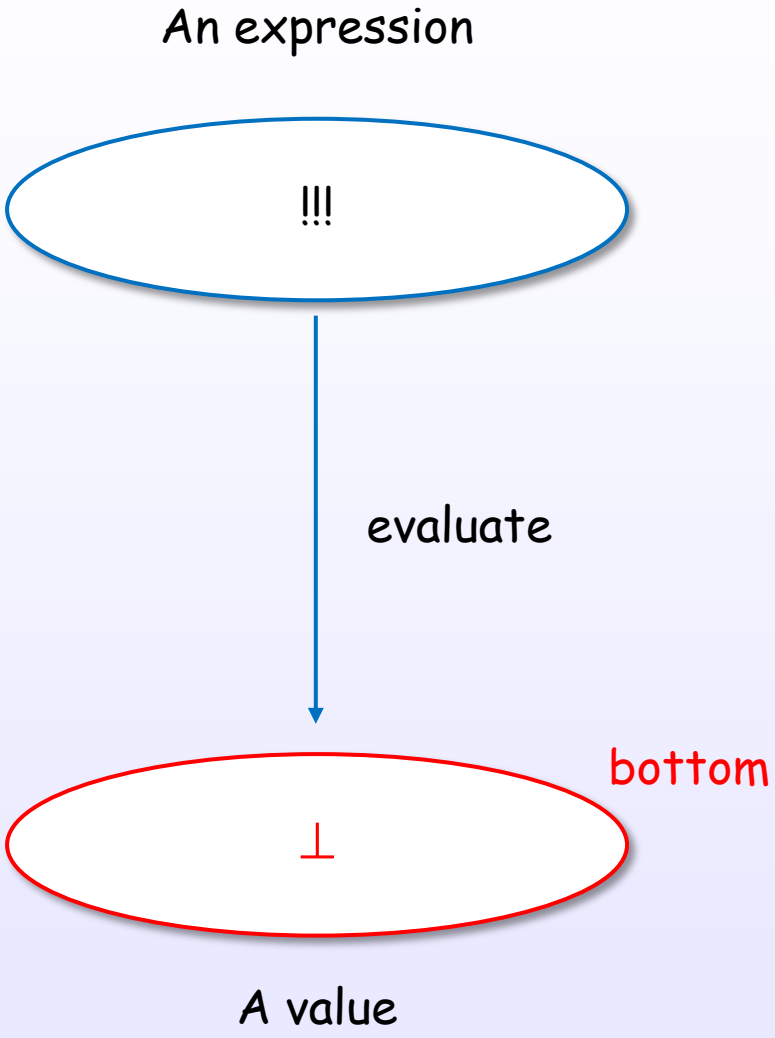
Well formed expression has a value



[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

$$f \perp = \perp$$

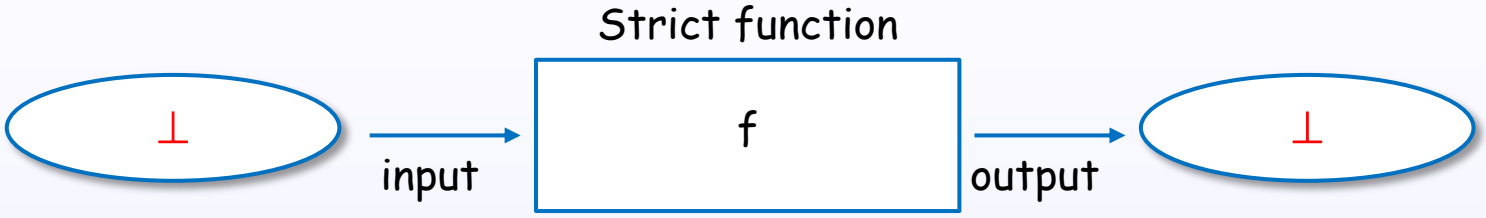
Strictness is attribution of the function.

[Bird, Chapter 2]

References : [1]

Strictness

$f \perp = \perp$



Strictness is attribution of the function.

[Bird, Chapter 2]

References : [1]

Strictness and Non-strictness

Strict

$$f \perp = \perp$$

Non-strict

$$f \perp \neq \perp$$

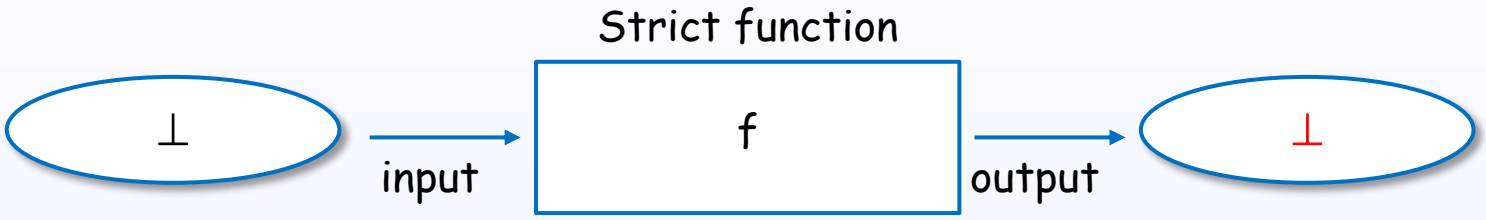
[Bird, Chapter 2]

References : [1]

Strictness and Non-strictness

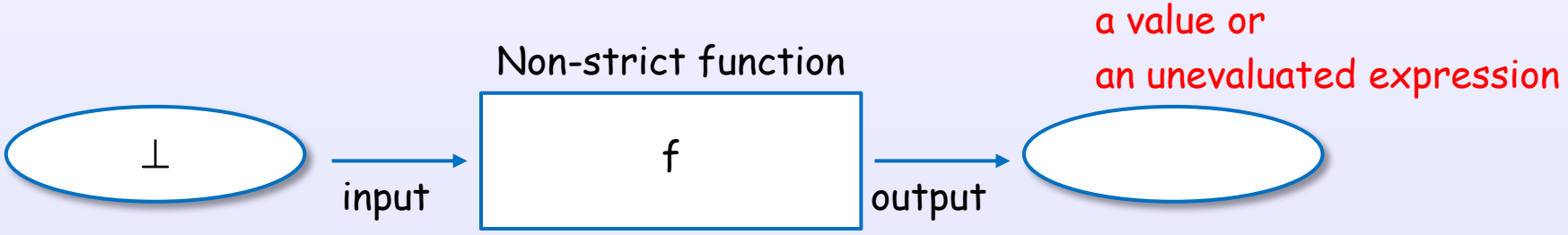
Strict

$$f \perp = \perp$$



Non-strict

$$f \perp \neq \perp$$



[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]

seq and pseq

$seq\ a\ b = \bot$, if $a = \bot$
$= b$, otherwise

$pseq\ a\ b = \bot$, if $a = \bot$
$= b$, otherwise

$seq\ a\ \bot = \bot$
$seq\ \bot\ b = \bot$

a is strict
b is strict

$pseq\ a\ \bot = \bot$
$pseq\ \bot\ b \neq \bot$

a is strict
b is non-strict

[Runtime Support for Multicore Haskell]

[Snoyman]

References : [1]

6. Semantics

Strict analysis

Strict analysis

References : [1]

7. Appendix

References

References

[H1] Haskell 2010 Language Report
https://www.haskell.org/definition/haskell2010.pdf

[H2] The Glorious Glasgow Haskell Compilation System (GHC user's guide)
https://downloads.haskell.org/~ghc/latest/docs/users_guide.pdf

[H3] A History of Haskell: Being Lazy With Class
http://haskell.cs.yale.edu/wp-content/uploads/2011/02/history.pdf

[H4] The implementation of functional programming languages
http://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/slpj-book-1987.pdf

[H5] 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

[H6] 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

[H7] Runtime Support for Multicore Haskell
http://community.haskell.org/~simonmar/papers/multicore-ghc.pdf

[H8] I know kung fu: learning STG by example
https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/GeneratedCode

[H9] GHC Commentary: The Layout of Heap Objects
https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/HeapObjects

[H10] GHC Commentary: Strict & StrictData
https://ghc.haskell.org/trac/ghc/wiki/StrictPragma

References

[B1] Introduction to Functional Programming using Haskell (IFPH 2nd edition)
http://www.cs.ox.ac.uk/publications/books/functional/bird-1998.jpg
http://www.pearsonhighered.com/educator/product/Introduction-Functional-Programming/9780134843469.page

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

[B3] Programming in Haskell
https://www.cs.nott.ac.uk/~gmh/book.html

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

[B5] Parallel and Concurrent Programming in Haskell
http://chimera.labs.oreilly.com/books/1230000000929

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

[B7] Purely Functional Data Structures
http://www.cambridge.org/us/academic/subjects/computer-science/programming-languages-and-applied-logic/purely-functional-data-structures

[B8] Algorithms: A Functional Programming Approach
http://catalogue.pearsoned.co.uk/catalog/academic/product/0,1144,0201596040,00.html

References

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

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

[D3] A Haskell Compiler
http://www.scs.stanford.edu/14sp-cs240h/slides/ghc-compiler-slides.html
http://www.scs.stanford.edu/11au-cs240h/notes/ghc-slides.html

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

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

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

[D7] Evaluation on the Haskell Heap
http://blog.ezyang.com/2011/04/evaluation-on-the-haskell-heap

[D8] Fixing foldl
http://www.well-typed.com/blog/2014/04/fixing-foldl

[D9] How to force a list
https://ro-che.info/articles/2015-05-28-force-list

[D10] Evaluation order and state tokens
https://www.fpcomplete.com/user/snoyberg/general-haskell/advanced/evaluation-order-and-state-tokens

References

[D11] Reasoning about laziness
http://blog.johantibell.com/2011/02/slides-from-my-talk-on-reasoning-about.html

[D12] Some History of Functional Programming Languages
http://www-fp.cs.st-andrews.ac.uk/tifp/TFP2012/TFP_2012/Turner.pdf

[D13] Why Functional Programming Matters
https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf

[D14] GHC illustrated
http://takenobu-hs.github.io/downloads/haskell_ghc_illustrated.pdf

References

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

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

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

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

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

References

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

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

Lazy,... zzz

to be as lazy as possible...