



Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Lecture C3: Lazy Evaluation and Memoising functions

USCS 2017

Stefan Holdermans
Doaitse Swierstra

Utrecht University

Aug 21-25, 2017

Infinite Lists

Given the following code:

```
take 0 l = []  
take n l = head l : take (n - 1) (tail l)  
length [] = 0  
length (_ : l) = 1 + length l
```

what is the result of the following session?

```
Prelude> let v = error "undefined"  
Prelude> v  
*** Exception: undefined  
Prelude> length (take 3 v)  
...
```



Infinite Lists

Given the following code:

```
take 0 l = []  
take n l = head l : take (n - 1) (tail l)  
length [] = 0  
length (_ : l) = 1 + length l
```

what is the result of the following session?

```
Prelude> let v = error "undefined"  
Prelude> v  
*** Exception: undefined  
Prelude> length (take 3 v)  
...
```

It may surprise some that the answer is 3



What is going on?

We evaluate the original expression stepwise:

length (take 3 v)

length (head v : take 2 (tail v))

1 + length (take 2 (tail v))

1 + length (head (tail v) : take 1 (tail (tail v)))

1 + 1 + length (take 1 (tail (tail v)))

1 + 1 + length (head (tail (tail v)) : take 0 (tail (tail (tail v))))

1 + 1 + 1 + length (take 0 (tail (tail (tail v))))

1 + 1 + 1 + length []

1 + 1 + 1 + 0

1 + 1 + 1

1 + 2

3



What is driving the evaluation?

In the example we have seen that every expression is evaluated when it is needed in order to decide which alternative of the function *length* should be taken. We conclude:

It is pattern matching (and evaluation of conditions) which drives the evaluation!

Each expression is only evaluated when, and as far as needed, when we have to decide how to proceed with the evaluation.



Why Functional programming is Easy

We have learned to appreciate that when we have automatic garbage collection **we do not have to worry about when the life of a value ends!**



Why Functional programming is Easy

We have learned to appreciate that when we have automatic garbage collection **we do not have to worry about when the life of a value ends!**

When using lazy evaluation **we do not have to worry about when the life of a value starts!**



Where lazy evaluation matters

- ▶ describing process like structures, streams of values
- ▶ recurrent relations
- ▶ combining functions, e.g. by building an infinite structure and inspecting only a finite part of it.



Example: Communicating processes

Two processes which communicate:

```
let pout = map p pin
    qout = map q qin
    pin  = 1 : qout
    qin  = pout
in pout
```

We can build arbitrary complicated nets of communication processes in this way.



Example: Eratosthenes' sieve

The famous algorithm, attributed to Eratosthenes, computes prime numbers:

1. take the list of all natural numbers starting from 2: $[2..]$.



Example: Eratosthenes' sieve

The famous algorithm, attributed to Eratosthenes, computes prime numbers:

1. take the list of all natural numbers starting from 2: $[2..]$.
2. remove all multiples of 2, and remember that 2 is a prime number.



Example: Eratosthenes' sieve

The famous algorithm, attributed to Eratosthenes, computes prime numbers:

1. take the list of all natural numbers starting from 2: $[2..]$.
2. remove all multiples of 2, and remember that 2 is a prime number.
3. the smallest number still in the list is 3, so remove all multiples of 3 and remember that 3 is a prime number



Example: Eratosthenes' sieve

The famous algorithm, attributed to Eratosthenes, computes prime numbers:

1. take the list of all natural numbers starting from 2: $[2..]$.
2. remove all multiples of 2, and remember that 2 is a prime number.
3. the smallest number still in the list is 3, so remove all multiples of 3 and remember that 3 is a prime number
4. the smallest remaining number is 5, so ...



Sifting

removeMultiples n list = filter (($\neq 0$) \circ ('mod' n)) list



Sifting

$$\text{removeMultiples } n \text{ list} = \text{filter } ((\neq 0) \circ ('\text{mod}' n)) \text{ list}$$

Apply repeatedly, letting prime numbers pass:

$$\text{sift } (p : xs) = p : \text{sift } (\text{removeMultiples } p \text{ xs})$$


Sifting

$$\text{removeMultiples } n \text{ list} = \text{filter } ((\neq 0) \circ ('mod' n)) \text{ list}$$

Apply repeatedly, letting prime numbers pass:

$$\text{sift } (p : xs) = p : \text{sift } (\text{removeMultiples } p \text{ xs})$$

And now pass the list of candidates:

$$\text{primeNumbers} = \text{sift } [2..]$$


Sifting

```
removeMultiples n list = filter (( $\neq 0$ )  $\circ$  ('mod' n)) list
```

Apply repeatedly, letting prime numbers pass:

```
sift (p : xs) = p : sift (removeMultiples p xs)
```

And now pass the list of candidates:

```
primeNumbers = sift [2..]
```

```
Programs> take 4 primeNumbers  
[2,3,5,7]
```



Hammings problem

Hammings problem

Generate an increasing list of values of which the prime factors are only 2, 3 and 5 ($\{2^i 3^j 5^k \mid i \geq 0, j \geq 0, k \geq 0\}$).



Hamming's problem

Hamming's problem

Generate an increasing list of values of which the prime factors are only 2, 3 and 5 ($\{2^i 3^j 5^k \mid i \geq 0, j \geq 0, k \geq 0\}$).

The typical way to approach this is to start with an inductive definition:

1. 1 is a Hamming number.



Hamming's problem

Hamming's problem

Generate an increasing list of values of which the prime factors are only 2, 3 and 5 ($\{2^i 3^j 5^k \mid i \geq 0, j \geq 0, k \geq 0\}$).

The typical way to approach this is to start with an inductive definition:

1. 1 is a Hamming number.
2. If n is a Hamming number then also $2 * n$, $3 * n$ en $5 * n$ are Hamming numbers.



Hamming's problem

Hamming's problem

Generate an increasing list of values of which the prime factors are only 2, 3 and 5 ($\{2^i 3^j 5^k \mid i \geq 0, j \geq 0, k \geq 0\}$).

The typical way to approach this is to start with an inductive definition:

1. 1 is a Hamming number.
2. If n is a Hamming number then also $2 * n$, $3 * n$ en $5 * n$ are Hamming numbers.
3. Purist add “And there are no other Hamming numbers”, but for computer scientists this is obvious.



Hamming's problem (code)

We now reason as follows:

1. Suppose that *ham* is the sought list, then the lists $\text{map } (*2) \text{ ham}$, $\text{map } (*3) \text{ ham}$, and $\text{map } (*5) \text{ ham}$ also contain Hamming numbers.



Hamming's problem (code)

We now reason as follows:

1. Suppose that *ham* is the sought list, then the lists $\text{map } (*2) \text{ ham}$, $\text{map } (*3) \text{ ham}$, and $\text{map } (*5) \text{ ham}$ also contain Hamming numbers.
2. If *ham* is **monotonically** increasing then this hold also for these other three lists.



Hamming's problem (code)

We now reason as follows:

1. Suppose that *ham* is the sought list, then the lists $\text{map } (*2) \text{ ham}$, $\text{map } (*3) \text{ ham}$, and $\text{map } (*5) \text{ ham}$ also contain Hamming numbers.
2. If *ham* is **monotonically** increasing then this hold also for these other three lists.
3. The numbers in these lists are not all different.

$\text{ham} = 1 : \dots$



Hamming's problem (code)

We now reason as follows:

1. Suppose that *ham* is the sought list, then the lists $\text{map } (*2) \text{ ham}$, $\text{map } (*3) \text{ ham}$, and $\text{map } (*5) \text{ ham}$ also contain Hamming numbers.
2. If *ham* is **monotonically** increasing then this hold also for these other three lists.
3. The numbers in these lists are not all different.

$$\begin{aligned} \text{ham} = & 1 : \dots (\text{map } (*2) \text{ ham}) \\ & \dots \\ & (\text{map } (*3) \text{ ham}) \\ & \dots \\ & (\text{map } (*5) \text{ ham}) \end{aligned}$$



Hamming's problem (code)

We now reason as follows:

1. Suppose that *ham* is the sought list, then the lists $\text{map } (*2) \text{ ham}$, $\text{map } (*3) \text{ ham}$, and $\text{map } (*5) \text{ ham}$ also contain Hamming numbers.
2. If *ham* is **monotonically** increasing then this hold also for these other three lists.
3. The numbers in these lists are not all different.

```
ham = 1 : ... (map (*2) ham)
           'merge'
           (map (*3) ham)
           'merge'
           (map (*5) ham)
```



Hamming's problem (code)

We now reason as follows:

1. Suppose that *ham* is the sought list, then the lists $\text{map } (*2) \text{ ham}$, $\text{map } (*3) \text{ ham}$, and $\text{map } (*5) \text{ ham}$ also contain Hamming numbers.
2. If *ham* is **monotonically** increasing then this hold also for these other three lists.
3. The numbers in these lists are not all different.

```
ham = 1 : remdup ((map (*2) ham)
                  'merge'
                  (map (*3) ham)
                  'merge'
                  (map (*5) ham)
                  )
```

```
remdup (x : ys) = x : remdup (dropWhile (≡ x) ys)
```



Trick question

Why doesn't the follow work:

$$\begin{array}{l} \text{remdup } (x : y : zs) \mid x \equiv y \quad = \quad \text{remdup } (y : zs) \\ \mid \text{otherwise} \quad = \quad x : \text{remdup } (y : zs) \end{array}$$



Trick question

Why doesn't the follow work:

$$\begin{array}{l} \mathit{remdup} (x : y : zs) \mid x \equiv y \quad = \quad \mathit{remdup} (y : zs) \\ \mid \mathit{otherwise} \quad = \quad x : \mathit{remdup} (y : zs) \end{array}$$

We evaluate a few steps:



Trick question

Why doesn't the follow work:

$$\begin{array}{l} \text{remdup } (x : y : zs) \mid x \equiv y \quad = \quad \text{remdup } (y : zs) \\ \mid \text{otherwise} = x : \text{remdup } (y : zs) \end{array}$$

```
ham = 1 : remdup ((map (*2) ham)
                  'merge'
                  (map (*3) ham)
                  'merge'
                  (map (*5) ham)
                  )
```



Trick question

Why doesn't the follow work:

$$\begin{array}{l} \text{remdup } (x : y : zs) \mid x \equiv y \quad = \quad \text{remdup } (y : zs) \\ \mid \text{otherwise} \quad = \quad x : \text{remdup } (y : zs) \end{array}$$
$$\begin{array}{l} \text{ham} = 1 : \text{remdup } ((2 : \text{map } (*2) (\text{tail ham})) \\ \quad \quad \quad \text{'merge'} \\ \quad \quad \quad (3 : \text{map } (*3) (\text{tail ham})) \\ \quad \quad \quad \text{'merge'} \\ \quad \quad \quad (5 : \text{map } (*5) (\text{tail ham})) \\ \quad \quad \quad) \end{array}$$


Trick question

Why doesn't the follow work:

```
remdup (x : y : zs) | x ≡ y    = remdup (y : zs)
                   | otherwise = x : remdup (y : zs)
```

```
ham = 1 : remdup 2 : ( (map (*2) (tail ham))
                       'merge'
                       (3 : map (*3) (tail ham))
                       'merge'
                       (5 : map (*5) (tail ham))
                       )
```



The Fibonacci sequence

Leonardo van Pisa ($\pm 1170 - \pm 1250$):

$$F_n = \begin{cases} n & \text{if } n < 2, \\ F_{n-2} + F_{n-1} & \text{if } n \geq 2. \end{cases}$$



fib :: *Integer* \rightarrow *Integer*

fib 0 = 0

fib 1 = 1

fib n = *fib* (n - 2) + *fib* (n - 1)



Interactive session: timing and memory usage

GHCi with `:set +s`:

```
Main >
```



Interactive session: timing and memory usage

GHCi with `:set +s`:

```
Main> fib 10
```



Interactive session: timing and memory usage

GHCi with `:set +s`:

```
Main> fib 10
55
0.02 secs, 3043752 bytes

Main>
```



Interactive session: timing and memory usage

GHCi with `:set +s`:

```
Main> fib 10
55
0.02 secs, 3043752 bytes

Main> fib 20
```



Interactive session: timing and memory usage

GHCi with `:set +s`:

```
Main> fib 10
55
0.02 secs, 3043752 bytes

Main> fib 20
6765
0.06 secs, 3133924 bytes

Main>
```



Interactive session: timing and memory usage

GHCi with `:set +s`:

```
Main> fib 10
55
0.02 secs, 3043752 bytes

Main> fib 20
6765
0.06 secs, 3133924 bytes

Main> fib 25
```



Interactive session: timing and memory usage

GHCi with `:set +s`:

```
Main> fib 10
55
0.02 secs, 3043752 bytes

Main> fib 20
6765
0.06 secs, 3133924 bytes

Main> fib 25
75025
0.63 secs, 34743476 bytes

Main>
```



Interactive session: timing and memory usage

GHCi with `:set +s`:

```
Main> fib 10
55
0.02 secs, 3043752 bytes

Main> fib 20
6765
0.06 secs, 3133924 bytes

Main> fib 25
75025
0.63 secs, 34743476 bytes

Main> fib 30
```



Interactive session: timing and memory usage

GHCi with `:set +s`:

```
Main> fib 10
55
0.02 secs, 3043752 bytes

Main> fib 20
6765
0.06 secs, 3133924 bytes

Main> fib 25
75025
0.63 secs, 34743476 bytes

Main> fib 30
832040
6.80 secs, 383178156 bytes
```



Interactive session: number of steps

Hugs (<http://haskell.org/hugs>):

```
Main> fib 10
```

```
55
```

```
Main> fib 20
```

```
6765
```

```
Main> fib 25
```

```
75025
```

```
Main> fib 30
```

```
832040
```



Interactive session: number of steps

Hugs (<http://haskell.org/hugs>) with +s:

```
Main> fib 10
55
3177 reductions, 5054 cells

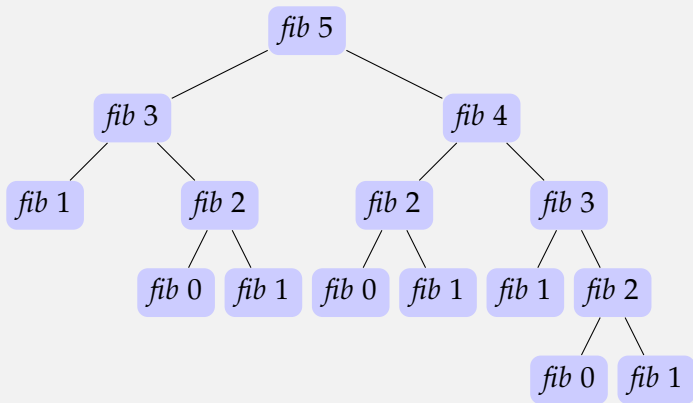
Main> fib 20
6765
390861 reductions, 622695 cells

Main> fib 25
75025
4334725 reductions, 6905874 cells, 6 garbage collections

Main> fib 30
832040
48072847 reductions, 76587387 cells, 77 garbage collections
```

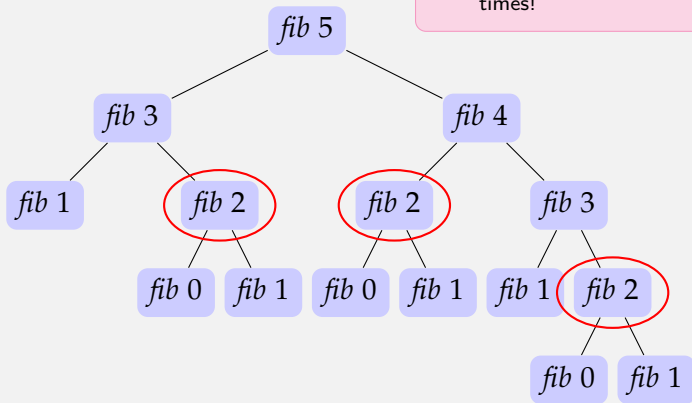


Call Tree



Call Tree

► *fib 2* is computed three times!



Number of recursive calls

We show the number of recursive calls for *fib* n :

value of n	number of <i>fib</i> calls
5	15
10	177
15	1973
20	21891
25	242785
30	2692537



Local memoisation

Idea: 'remember' the results of the function calls for a **sequence** of arguments.



Local memoisation

Idea: 'remember' the results of the function calls for a **sequence** of arguments.

$fib :: Integer \rightarrow Integer$

$fib\ n = fibs!\ n$

where

$fibs = listArray\ (0, n)\ \$$

$0 : 1 : [fibs!\ (k - 2) + fibs!\ (k - 1) \mid k \leftarrow [2..n]]$



Local memoisation

Idea: 'remember' the results of the function calls for a **sequence** of arguments.

$$fib :: Integer \rightarrow Integer$$
$$fib \ n \quad = \ fibs! \ n$$

where

$$fibs = listArray \ (0, n) \ \$$$
$$0 : 1 : [fibs! \ (k - 2) + fibs! \ (k - 1) \mid k \leftarrow [2..n]]$$

☞ For each call of *fib* we construct a completely new array *fibs*.



Global memoisation

The global memo function

- ▶ also remembers the results of **previous calls** directly from the program,
- ▶ remembers the result for all **all** arguments ever passed.



Global memoisation

The global memo function

- ▶ also remembers the results of **previous calls** directly from the program,
- ▶ remembers the result for all **all** arguments ever passed.

Goal: to construct a library which makes it easy to build a memoising version of a function which takes an *Integer* parameter.



Fixed-point Combinator

The **fixed point** of a function f is the value x , for which $f x = x$ holds.



Fixed-point Combinator

The **fixed point** of a function f is the value x , for which $f x = x$ holds.

A **fixpoint combinator** is a higher-order function which 'computes' the fixpoint of other functions:

$$\begin{aligned} \text{fix} &:: (a \rightarrow a) \rightarrow a \\ \text{fix } f &= \text{let } \text{fixf} = f \text{ fixf } \text{ in } \text{fixf} \end{aligned}$$


Explicit recursion

Using *fix* we can make the use of **recursion** explicit:



Explicit recursion

Using *fix* we can make the use of **recursion** explicit:

Example:

```
fac :: Integer → Integer
fac 0      = 1
fac n     = n * fac (n - 1)
```



Explicit recursion

Using *fix* we can make the use of **recursion** explicit:

Example:

```
fac :: Integer → Integer
fac 0      = 1
fac n     = n * fac (n - 1)
```

can, using *fix*, be written as:

```
fac :: Integer → Integer
fac = fix fac'
  where
    fac' f 0 = 1
    fac' f n = n * f (n - 1)
```



Explicit recursion

Using *fix* we can make the use of **recursion** explicit:

Example:

```
fac :: Integer → Integer
fac 0      = 1
fac n     = n * fac (n - 1)
```

can, using *fix*, be written as:

```
fac :: Integer → Integer
fac = fix fac'
  where
    fac' f 0 = 1
    fac' f n = n * f (n - 1)
```

Idea: introduce an extra parameter which is used in the recursive calls:



Explicit recursion

Using *fix* we can make the use of **recursion** explicit:

Example:

```
fac :: Integer → Integer
fac 0      = 1
fac n     = n * fac (n - 1)
```

can, using *fix*, be written as:

```
fac :: Integer → Integer
fac = fix fac'
  where
    fac' f 0 = 1
    fac' f n = n * f (n - 1)
```

Idea: introduce an extra parameter which is used in the recursive calls:

 $fac' :: (Integer \rightarrow Integer) \rightarrow (Integer \rightarrow Integer)$.



Explicit recursion: example

$$\begin{aligned} & \text{fac } 3 \\ = & \\ & \text{fix fac}' 3 \\ = & \\ & \text{fac}' (\text{fix fac}') 3 \\ = & \\ & 3 * \text{fix fac}' (3 - 1) \\ = & \\ & 3 * \text{fix fac}' 2 \\ = & \\ & 3 * \text{fac}' (\text{fix fac}') 2 \\ = & \\ & 3 * (2 * \text{fix fac}' (2 - 1)) \\ = & \\ & 3 * (2 * \text{fix fac}' 1) \end{aligned}$$
$$\begin{aligned} & 6 \\ = & \\ & 3 * 2 \\ = & \\ & 3 * (2 * 1) \\ = & \\ & 3 * (2 * (1 * 1)) \\ = & \\ & 3 * (2 * (1 * \text{fix fac}' 0)) \\ = & \\ & 3 * (2 * (1 * \text{fix fac}' (1 - 1))) \\ = & \\ & 3 * (2 * \text{fac}' (\text{fix fac}') 1) \\ = & \\ & 3 * (2 * \text{fix fac}' 1) \end{aligned}$$


Fibonacci again

Fibonacci function with explicit recursion

:

$fib :: Integer \rightarrow Integer$

$fib = fix fib'$

where

$$fib' f 0 = 0$$

$$fib' f 1 = 1$$

$$fib' f n = f (n - 2) + f (n - 1)$$



Fibonacci again

Fibonacci function with explicit recursion, and clever (ab)use of Haskell scope rules:

```
fib :: Integer → Integer
```

```
fib = fix fib
```

```
where
```

```
  fib fib 0 = 0
```

```
  fib fib 1 = 1
```

```
  fib fib n = fib (n - 2) + fib (n - 1)
```

Idea: replace *fix* by a **memoising** fixpoint combinator



Library for memofunctions: plan of attack

Choose a (parameterised) datatype *Memo* for the memo tables.



Library for memofunctions: plan of attack

Choose a (parameterised) datatype *Memo* for the memo tables.

Define functions *tabulate* and *apply*,

$$\begin{aligned} \text{tabulate} &:: (\text{Integer} \rightarrow a) \rightarrow \text{Memo } a \\ \text{apply} &:: \text{Memo } a \rightarrow \text{Integer} \rightarrow a \end{aligned}$$

such that:

- ▶ *tabulate f* results in a (lazily constructed) memo table containing all results of calls to *f* and
- ▶ *apply mem n* retrieves the corresponding value for the parameter *n* from *mem*.



Library for memofunctions: plan of attack

Choose a (parameterised) datatype *Memo* for the memo tables.

Define functions *tabulate* and *apply*,

$$\begin{aligned} \text{tabulate} &:: (\text{Integer} \rightarrow a) \rightarrow \text{Memo } a \\ \text{apply} &:: \text{Memo } a \rightarrow \text{Integer} \rightarrow a \end{aligned}$$

such that:

- ▶ *tabulate f* results in a (lazily constructed) memo table containing all results of calls to *f* and
- ▶ *apply mem n* retrieves the corresponding value for the parameter *n* from *mem*.

Define a fixedpoint combinator *memo* using *tabulate* and *apply*.



Memo lists

In our first approach we will represent memo tables using *infinite* lists:

```
type Memo a = [a]
```



Memo lists

In our first approach we will represent memo tables using *infinite* lists:

```
type Memo a = [a]
```

```
tabulate :: (Integer -> a) -> Memo a  
tabulate f = map f [0..]
```



Memo lists

In our first approach we will represent memo tables using **infinite** lists:

```
type Memo a = [a]
```

```
tabulate :: (Integer → a) → Memo a  
tabulate f = map f [0..]
```

```
apply :: Memo a → Integer → a  
apply (x : _) 0 = x  
apply (_ : xs) n = apply xs (n - 1)
```



Memo combinator

$$\text{memo} :: ((\text{Integer} \rightarrow a) \rightarrow (\text{Integer} \rightarrow a)) \rightarrow (\text{Integer} \rightarrow a)$$
$$\text{memo } f' = f$$

where

$$f = \text{apply } (\text{tabulate } (f' f))$$


Memo combinator

$$\text{memo} :: ((\text{Integer} \rightarrow a) \rightarrow (\text{Integer} \rightarrow a)) \rightarrow (\text{Integer} \rightarrow a)$$
$$\text{memo } f' = f$$

where

$$f = \text{apply } (\text{tabulate } (f' f))$$

- ▶ The combinator constructs a fixpoint f of f' .



Memo combinator

$$\text{memo} :: ((\text{Integer} \rightarrow a) \rightarrow (\text{Integer} \rightarrow a)) \rightarrow (\text{Integer} \rightarrow a)$$
$$\text{memo } f' = f$$

where

$$f = \text{apply } (\text{tabulate } (f' f))$$

- ▶ The combinator constructs a fixpoint f of f' .
- ▶ The function f retrieves its result from the memo table $\text{tabulate } (f' f)$.



Memo combinator

$$\text{memo} :: ((\text{Integer} \rightarrow a) \rightarrow (\text{Integer} \rightarrow a)) \rightarrow (\text{Integer} \rightarrow a)$$
$$\text{memo } f' = f$$

where

$$f = \text{apply } (\text{tabulate } (f' f))$$

- ▶ The combinator constructs a fixpoint f of f' .
- ▶ The function f retrieves its result from the memo table $\text{tabulate } (f' f)$.
- ▶ Each element in the table is computed using f' .



Memo combinator

$$\text{memo} :: ((\text{Integer} \rightarrow a) \rightarrow (\text{Integer} \rightarrow a)) \rightarrow (\text{Integer} \rightarrow a)$$
$$\text{memo } f' = f$$

where

$$f = \text{apply } (\text{tabulate } (f' f))$$

- ▶ The combinator constructs a fixpoint f of f' .
- ▶ The function f retrieves its result from the memo table $\text{tabulate } (f' f)$.
- ▶ Each element in the table is computed using f' .
- ▶ Recursive calls use the memo function f .



Memo combinator

$$\text{memo} :: ((\text{Integer} \rightarrow a) \rightarrow (\text{Integer} \rightarrow a)) \rightarrow (\text{Integer} \rightarrow a)$$
$$\text{memo } f' = f$$

where

$$f = \text{apply } (\text{tabulate } (f' f))$$

- ▶ The combinator constructs a fixpoint f of f' .
- ▶ The function f retrieves its result from the memo table $\text{tabulate } (f' f)$.
- ▶ Each element in the table is computed using f' .
- ▶ Recursive calls use the memo function f .
- ▶ Thanks to lazy evaluation only those elements in the list are computed which are really used in constructing the resulting value



Memo combinator

$$\text{memo} :: ((\text{Integer} \rightarrow a) \rightarrow (\text{Integer} \rightarrow a)) \rightarrow (\text{Integer} \rightarrow a)$$
$$\text{memo } f' = f$$

where

$$f = \text{apply } (\text{tabulate } (f' f))$$

- ▶ The combinator constructs a fixpoint f of f' .
- ▶ The function f retrieves its result from the memo table $\text{tabulate } (f' f)$.
- ▶ Each element in the table is computed using f' .
- ▶ Recursive calls use the memo function f .
- ▶ Thanks to lazy evaluation only those elements in the list are computed which are really used in constructing the resulting value
- ▶ The table does not depend on the parameter of f ; calls to f share the table which is **persistent during the evaluation of the program**



Fibonacci sequence using memo lists

Fibonacci function using global memoisation:

$fib :: Integer \rightarrow Integer$

$fib = memo\ fib'$

where

$fib' f\ 0 = 0$

$fib' f\ 1 = 1$

$fib' f\ n = f\ (n - 2) + f\ (n - 1)$



Memo lists: number of reductions

```
Main> fib 10
55
1450 reductions, 2316 cells

Main> fib 20
6765
5060 reductions, 8178 cells

Main> fib 25
75025
7690 reductions, 12463 cells

Main> fib 30
832040
10870 reductions, 17649 cells
```



Memo lists: subsequent calls

Main >



Memo lists: subsequent calls

```
Main> fib 30
```



Memo lists: subsequent calls

```
Main> fib 30
832040
10870 reductions, 17649 cells
Main>
```



Memo lists: subsequent calls

```
Main> fib 30
832040
10870 reductions, 17649 cells
Main> fib 30
```



Memo lists: subsequent calls

```
Main> fib 30
832040
10870 reductions, 17649 cells

Main> fib 30
832040
359 reductions, 583 cells
```



Memo lists: subsequent calls

```
Main> fib 30
832040
10870 reductions, 17649 cells

Main> fib 30
832040
359 reductions, 583 cells
```

In the second call all we have to do is to look up the result in the table.



Memo lists: linear search time

- ▶ Arrays: fixed number of possible argument, but **constant** lookup time.
- ▶ Lists: no restriction on number of arguments, but **linear** lookup time.



Memo lists: linear search time

- ▶ Arrays: fixed number of possible argument, but **constant** lookup time.
- ▶ Lists: no restriction on number of arguments, but **linear** lookup time.

```
Main> fib 5000
3878968454388325633701916308325905312082127714...
41.78 secs, 2532516300 bytes
```



Memo lists: linear search time

- ▶ Arrays: fixed number of possible argument, but **constant** lookup time.
- ▶ Lists: no restriction on number of arguments, but **linear** lookup time.

```
Main> fib 5000
3878968454388325633701916308325905312082127714...
41.78 secs, 2532516300 bytes
```

Golden middle road: memo trees (all arguments, **logarithmic** lookup time).



Library for memo functions: plan of attack (unchanged)

Choose a (parameterised) data type *Memo* for memo tables.

Define functions *tabulate* and *apply*,

$$\begin{aligned} \text{tabulate} &:: (\text{Integer} \rightarrow a) \rightarrow \text{Memo } a \\ \text{apply} &:: \text{Memo } a \rightarrow \text{Integer} \rightarrow a \end{aligned}$$

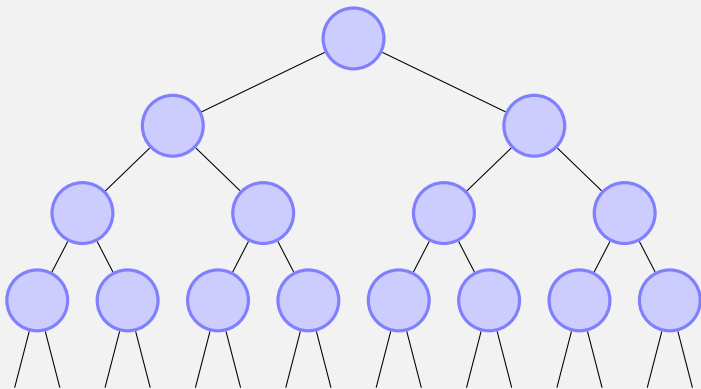
such that:

- ▶ *tabulate f* a (lazy) memo table containing the results of all possible calls to *f*
- ▶ *apply mem n* which locates the result for *n* in *mem*.

Define a fixedpoint *memo* using *tabulate* and *apply*.

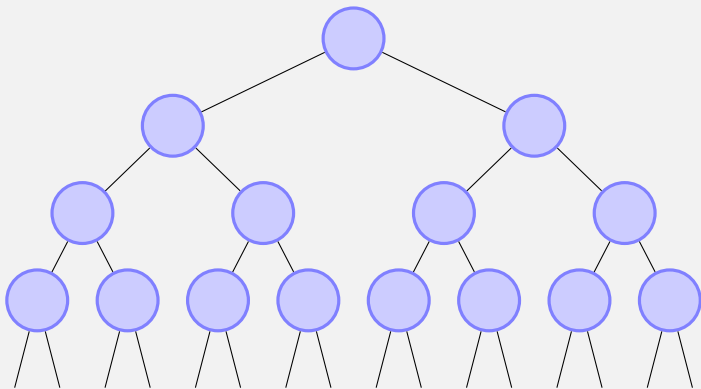


Memo trees



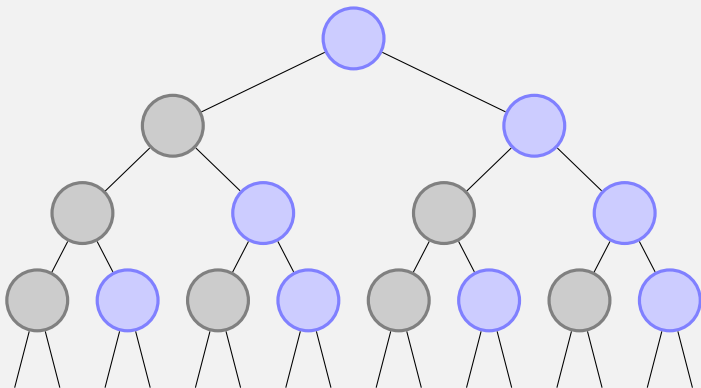
Memo trees

- ▶ Infinite binary tree with values in the nodes.
- ▶ No value in left children.



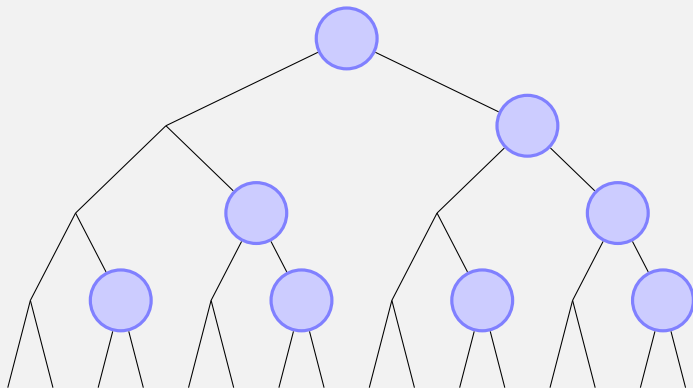
Memo trees

- ▶ Infinite binary tree with values in the nodes.
- ▶ No value in left children.



Memo trees

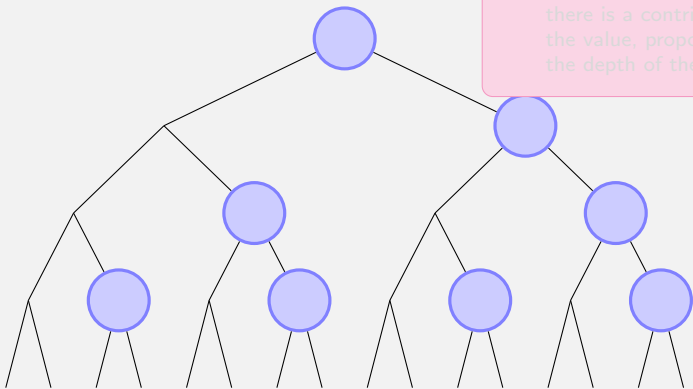
- ▶ Infinite binary tree with values in the nodes.
- ▶ No value in left children.



Memo trees

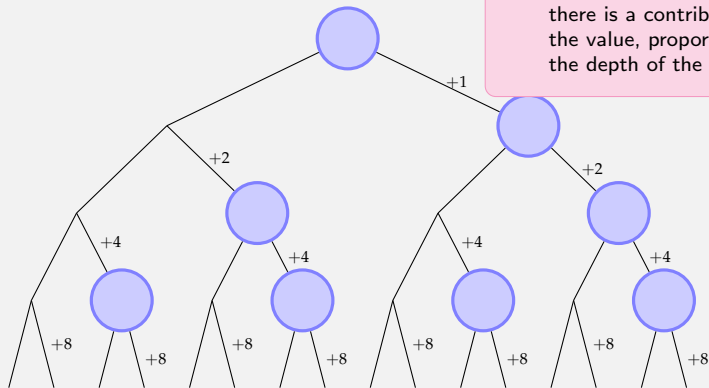
- ▶ The search key for a right child is determined by the edges going right in the path from the root .

- ▶ Each time we go right there is a contribution to the value, proportional to the depth of the tree.



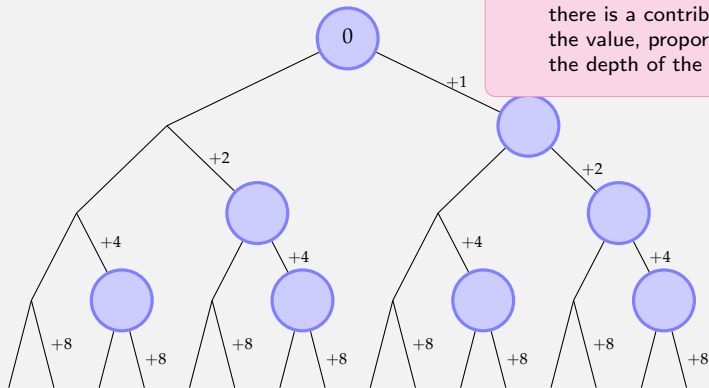
Memo trees

- ▶ The search key for a right child is determined by the edges going right in the path from the root .
- ▶ Each time we go right there is a contribution to the value, proportional to the depth of the tree.



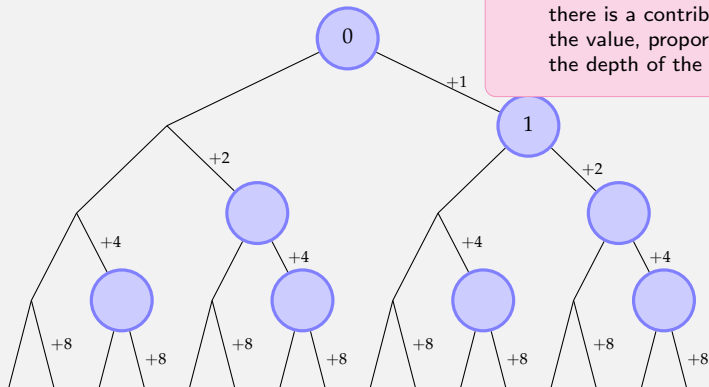
Memo trees

- ▶ The search key for a right child is determined by the edges going right in the path from the root .
- ▶ Each time we go right there is a contribution to the value, proportional to the depth of the tree.



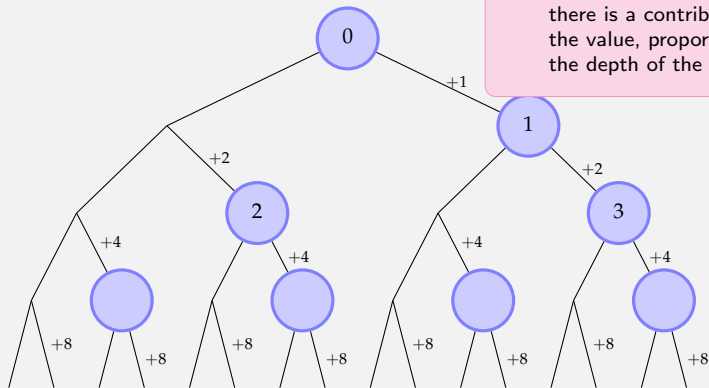
Memo trees

- ▶ The search key for a right child is determined by the edges going right in the path from the root .
- ▶ Each time we go right there is a contribution to the value, proportional to the depth of the tree.



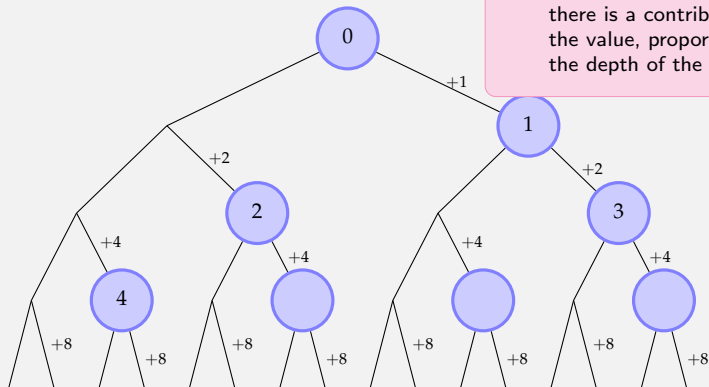
Memo trees

- ▶ The search key for a right child is determined by the edges going right in the path from the root .
- ▶ Each time we go right there is a contribution to the value, proportional to the depth of the tree.



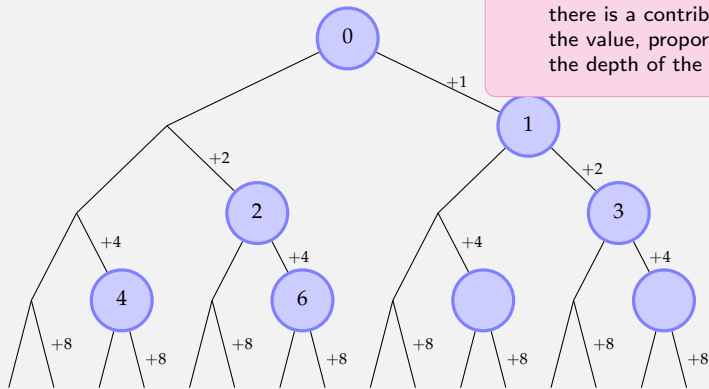
Memo trees

- ▶ The search key for a right child is determined by the edges going right in the path from the root .
- ▶ Each time we go right there is a contribution to the value, proportional to the depth of the tree.



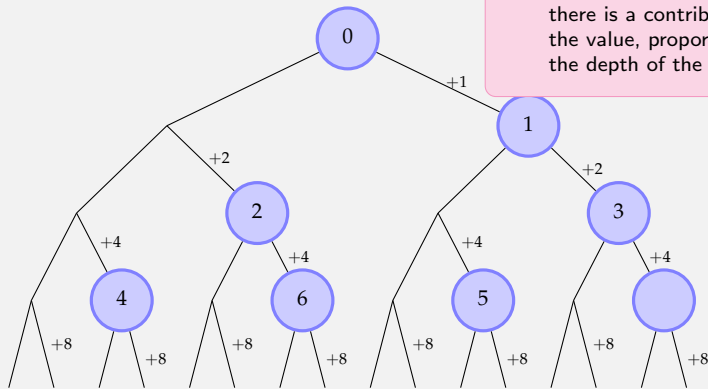
Memo trees

- ▶ The search key for a right child is determined by the edges going right in the path from the root .
- ▶ Each time we go right there is a contribution to the value, proportional to the depth of the tree.



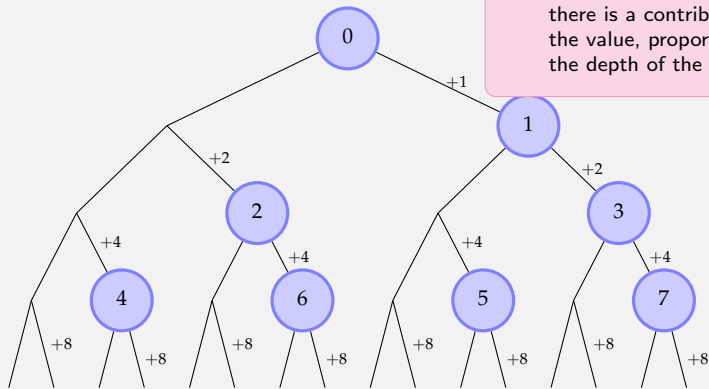
Memo trees

- ▶ The search key for a right child is determined by the edges going right in the path from the root .
- ▶ Each time we go right there is a contribution to the value, proportional to the depth of the tree.



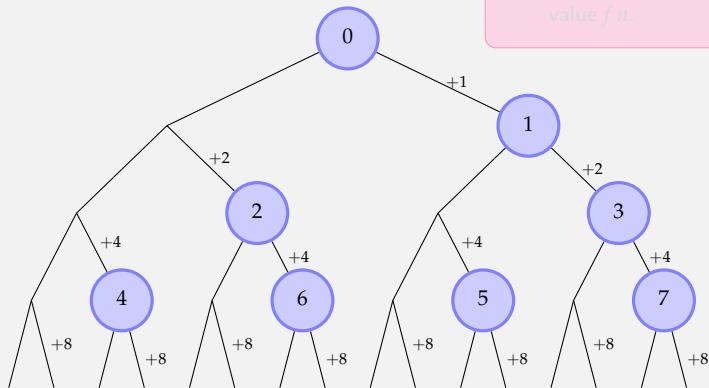
Memo trees

- ▶ The search key for a right child is determined by the edges going right in the path from the root .
- ▶ Each time we go right there is a contribution to the value, proportional to the depth of the tree.



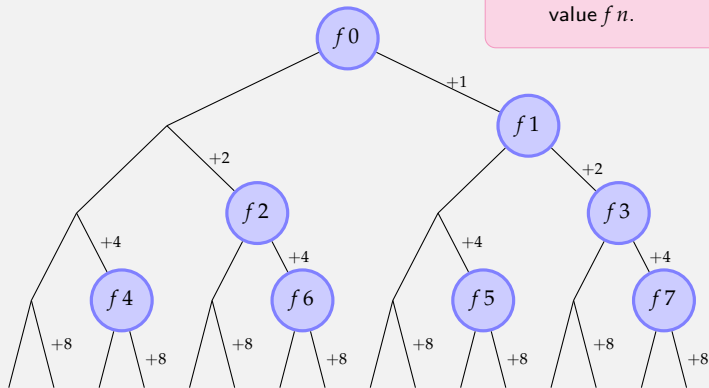
Memo trees

- ▶ In the nodes we store the values for the function f which is to be memoised.
- ▶ In a right child with weight n we store the value f_n .

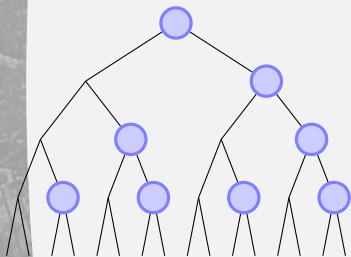


Memo trees

- ▶ In the nodes we store the values for the function f which is to be memoised.
- ▶ In a right child with weight n we store the value $f n$.



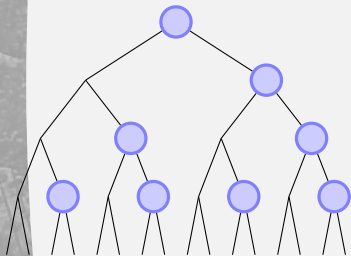
Data type for memo trees



Type of an **infinite binaire** tree with values in the **root** and in each **right child**.



Data type for memo trees



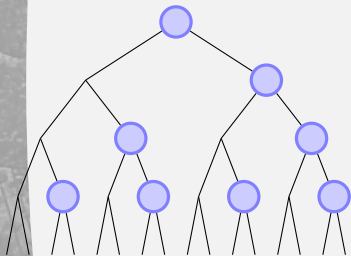
Type of an **infinite binaire** tree with values in the **root** and in each **right child**.

```
data Memo a = Memo (Memo' a) a (Memo a)
```

```
data Memo' a = Memo' (Memo' a) (Memo a)
```




Data type for memo trees



Type of an **infinite binaire** tree with values in the **root** and in each **right child**.

```
data Memo a = Memo (Memo' a) a (Memo a)
```

```
data Memo' a = Memo' (Memo' a) (Memo a)
```

 *Memo* and *Memo'* are defined mutually recursive.



Construction of the memo tree

```
tabulate :: (Integer → a) → Memo a
```

```
tabulate f = tab 0 1
```

where

```
tab k i =
```

```
  let j = 2 * i in Memo (tab' k j) (f k) (tab (k + i) j)
```

```
tab' k i =
```

```
  let j = 2 * i in Memo' (tab' k j) (tab (k + i) j)
```

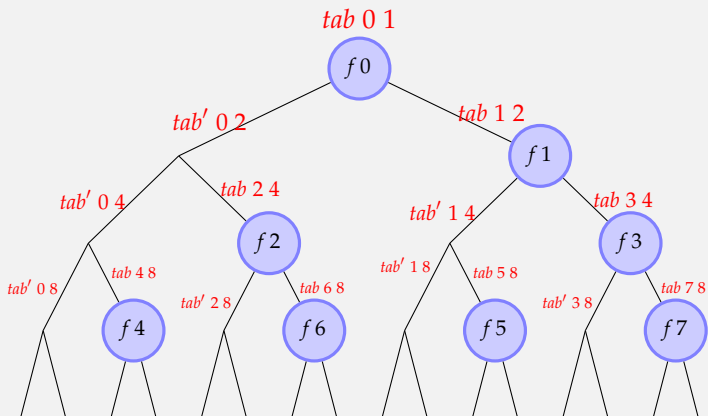
Arguments of helper function:

- ▶ For *tab*: the next search key and the next weight (i.e. the increase of the search key).
- ▶ For *tab'*: last search key and again the increase in weight at this level.



Memo tree construction: example

tabulate f



Searching in a memo tree

$apply :: Memo\ a \rightarrow Integer \rightarrow a$

$apply = app$

where

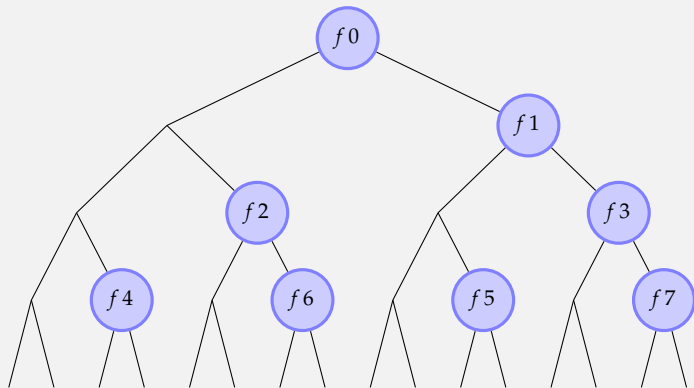
$app\ (Memo\ l\ x\ r)\ n$	$n \equiv 0$	$= x$
	even n	$= app'\ l\ (n\ 'div'\ 2)$
	otherwise	$= app\ r\ (n\ 'div'\ 2)$
$app'\ (Memo'\ l\ r)\ n$	even n	$= app'\ l\ (n\ 'div'\ 2)$
	otherwise	$= app\ r\ (n\ 'div'\ 2)$

In each recursive step the search key is halved and we decrease one level in the tree

If the key reaches 0, we return the value in the current node.

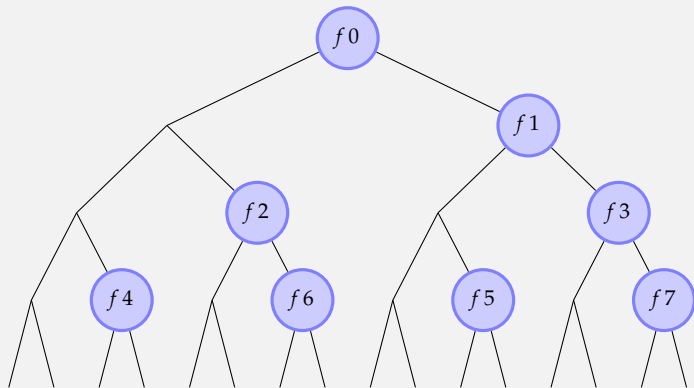


Searching in a memo tree: example



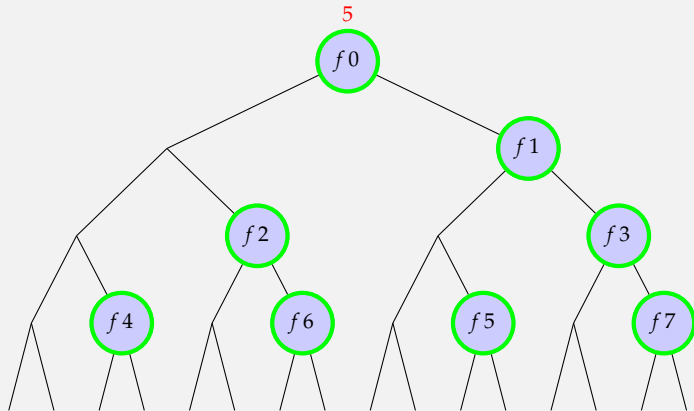
Searching in a memo tree: example

apply  5



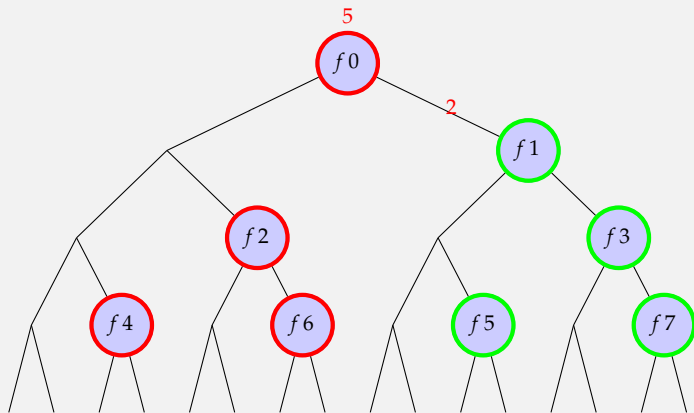
Searching in a memo tree: example

apply  5



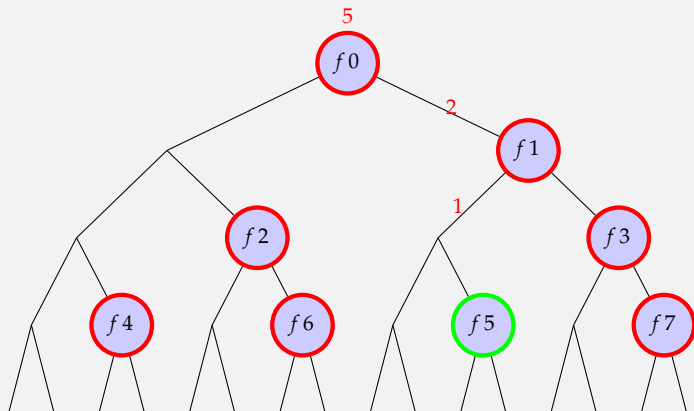
Searching in a memo tree: example

apply  5



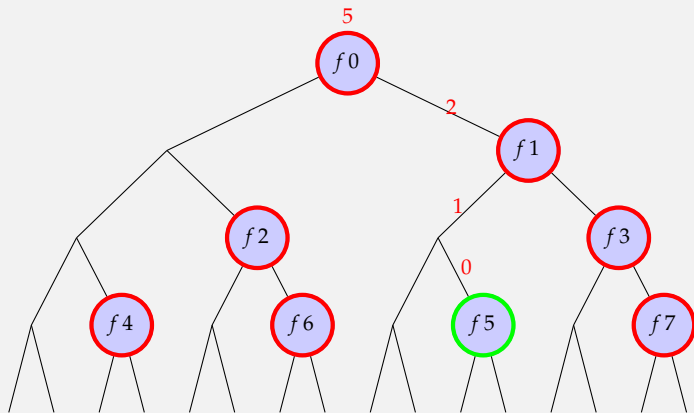
Searching in a memo tree: example

apply  5



Searching in a memo tree: example

apply  5



Memo combinator (unchanged)

The definition of *memo* is independent of the table representation:

$$\begin{aligned} \text{memo} &:: ((\text{Integer} \rightarrow a) \rightarrow \text{Integer} \rightarrow a) \rightarrow \text{Integer} \rightarrow a \\ \text{memo } f' &= f \\ &\text{where} \\ &f = \text{apply } (\text{tabulate } (f' f)) \end{aligned}$$


Fibonacci sequence using memo trees

$fib :: Integer \rightarrow Integer$

$fib = memo\ fib'$

where

$$fib' f 0 = 0$$

$$fib' f 1 = 1$$

$$fib' f n = f (n - 2) + f (n - 1)$$



Memo trees: time and memory usage

```
Main >
```



Memo trees: time and memory usage

```
Main> fib 5000
```



Memo trees: time and memory usage

```
Main> fib 5000
3878968454388325633701916308325905312082127714...
0.37 secs, 26809216 bytes
Main>
```



Memo trees: time and memory usage

```
Main> fib 5000  
3878968454388325633701916308325905312082127714...  
0.37 secs, 26809216 bytes  
Main> fib 5000
```



Memo trees: time and memory usage

```
Main> fib 5000
3878968454388325633701916308325905312082127714...
0.37 secs, 26809216 bytes

Main> fib 5000
3878968454388325633701916308325905312082127714...
0.02 secs, 532752 bytes
```



Conclusions

- ▶ More efficient table structure requires some programming effort, but is a 'one-time investment'.
- ▶ Choice of data structure is invisible to user of the library.
- ▶ Only thing required from the user: making the recursion explicit.



Final remarks

- ▶ we can extend the memoisation for any kind of value that can be mapped onto an *Integer*
- ▶ functions with more than one parameter can be memoised by having memo tables returning memo tables and having successive lookups
- ▶ is part of several hackage packages, see <http://hackage.haskell.org/package/MemoTrie>

