Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Lazy Evaluation and Memoïsing functions

USCS 2016

Utrecht University

July 4-15, 2016

Infinite Lists

Given the following code:

$$
\begin{aligned}
&take\ 0\ l = [\,]\\
&take\ n\ l = head\ l : take\ (n-1)\ (tail\ l)\\
&length\ [\,] = 0\\
&length\ (\_ : l) = 1 + length\ l
\end{aligned}
$$

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:

$$
\begin{aligned}
&\mathit{take}\ 0\ l = [\,] \\
&\mathit{take}\ n\ l = \mathit{head}\ l : \mathit{take}\ (n-1)\ (\mathit{tail}\ l) \\
&\mathit{length}\ [\,] = 0 \\
&\mathit{length}\ (\_ : l) = 1 + \mathit{length}\ l
\end{aligned}
$$

what is the result of the following session?

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

It may suprise some that the answer is 3

What is going on?

We evaluate the original expression stepwise:

$$\begin{array}{l}
\mathit{length}\ (\mathit{take}\ 3\ v) \\
\mathit{length}\ (\mathit{head}\ v : \mathit{take}\ 2\ (\mathit{tail}\ v)) \\
1 + \mathit{length}\ (\mathit{take}\ 2\ (\mathit{tail}\ v)) \\
1 + \mathit{length}\ (\mathit{head}\ (\mathit{tail}\ v) : \mathit{take}\ 1\ (\mathit{tail}\ (\mathit{tail}\ v))) \\
1 + 1 + \mathit{length}\ (\mathit{take}\ 1\ (\mathit{tail}\ (\mathit{tail}\ v))) \\
1 + 1 + \mathit{length}\ (\mathit{head}\ (\mathit{tail}\ (\mathit{tail}\ v)) : \mathit{take}\ 0\ (\mathit{tail}\ (\mathit{tail}\ (\mathit{tail}\ v)))) \\
1 + 1 + 1 + \mathit{length}\ (\mathit{take}\ 0\ (\mathit{tail}\ (\mathit{tail}\ (\mathit{tail}\ v)))) \\
1 + 1 + 1 + \mathit{length}\ [\,] \\
1 + 1 + 1 + 0 \\
1 + 1 + 1 \\
1 + 2 \\
3
\end{array}$$

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
- recurrent relations
- combing function, e.g. by building an infinite structure and inspecting only a finite part of it.

Example: Communicating processes

Two processes which communicate:

$$
\begin{aligned}
&\textbf{let}\ pout = map\ p\ pin \\
&\qquad qout = map\ q\ qin \\
&\qquad pin \ \ = 1 : qout \\
&\qquad qin \ \ = pout \\
&\textbf{in}\ pout
\end{aligned}
$$

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

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

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

$$removeMultiples\ n\ list = filter\ ((\not\equiv 0) \circ (\text{`}mod\text{`}n))\ list$$

Sifting

$$removeMultiples\ n\ list = filter\ ((\not\equiv 0) \circ (`mod`\, n))\ list$$

Apply repeatedly, letting prime numbers pass:

$$sift\ (p : xs) = p : sift\ (removeMultiples\ p\ xs)$$

Sifting

$$removeMultiples\ n\ list = filter\ ((\not\equiv 0) \circ (`mod`n))\ list$$

Apply repeatedly, letting prime numbers pass:

$$sift\ (p : xs) = p : sift\ (removeMultiples\ p\ xs)$$

And now pass the list of candidates:

$$primeNumbers = sift\ [2\mathinner{\ldotp\ldotp}]$$

Sifting

$$removeMultiples\ n\ list = filter\ ((\not\equiv 0) \circ (`mod` n))\ list$$

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 ($\{2^i 3^j 5^k \mid i >= 0, j >= 0, k >= 0\}$).

Hammings problem

Hammings problem

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

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

1. 1 is a Hamming number.

Hammings problem

Hammings problem

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

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.

Hammings problem

Hammings problem

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

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 $map\ (*2)\ ham$, $map\ (*3)\ ham$, and $map\ (*5)\ ham$ also contain Hamming numbers.

Hamming's problem (code)

We now reason as follows:

1. Suppose that ham is the sought list, then the lists $map\ (*2)\ ham$, $map\ (*3)\ ham$, and $map\ (*5)\ ham$ 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 $map\ (*2)\ ham$, $map\ (*3)\ ham$, and $map\ (*5)\ ham$ 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.

$$ham = 1 : \ldots$$

Hamming's problem (code)

We now reason as follows:

1. Suppose that *ham* is the sought list, then the lists $map\ (*2)\ ham$, $map\ (*3)\ ham$, and $map\ (*5)\ ham$ 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.

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

Hamming's problem (code)

We now reason as follows:

1. Suppose that *ham* is the sought list, then the lists $map\ (*2)\ ham$, $map\ (*3)\ ham$, and $map\ (*5)\ ham$ 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.

$$
\begin{aligned}
ham = 1 : \ldots &(map\ (*2)\ ham) \\
&\quad `merge` \\
&(map\ (*3)\ ham) \\
&\quad `merge` \\
&(map\ (*5)\ ham)
\end{aligned}
$$

Hamming's problem (code)

We now reason as follows:

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

$$
\begin{aligned}
&ham = 1 : remdup\ ((map\ ({*}2)\ ham)\\
&\qquad\qquad\qquad\qquad \text{`}merge\text{`}\\
&\qquad\qquad\qquad\quad (map\ ({*}3)\ ham)\\
&\qquad\qquad\qquad\qquad \text{`}merge\text{`}\\
&\qquad\qquad\qquad\quad (map\ ({*}5)\ ham)\\
&\qquad\quad)\\
&remdup\ (x : ys) = x : remdup\ (dropWhile\ ({\equiv}\ x)\ ys)
\end{aligned}
$$

Trick question

Why doesn't the follow work:

$$
\begin{aligned}
\mathit{remdup}\ (x:y:zs) \mid x \equiv y &= \mathit{remdup}\ (y:zs) \\
\mid \mathit{otherwise} &= x:\mathit{remdup}\ (y:zs)
\end{aligned}
$$

Trick question

Why doesn't the follow work:

$$
\begin{aligned}
remdup\ (x : y : zs)\ &|\ x \equiv y &&= remdup\ (y : zs) \\
&|\ otherwise &&= x : remdup\ (y : zs)
\end{aligned}
$$

We evaluate a few steps:

Trick question

Why doesn't the follow work:

$$
\begin{aligned}
&\mathit{remdup}\ (x : y : zs) \mid x \equiv y &&= \mathit{remdup}\ (y : zs) \\
&\phantom{\mathit{remdup}\ (x : y : zs)} \mid \mathit{otherwise} &&= x : \mathit{remdup}\ (y : zs)
\end{aligned}
$$

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

Trick question

Why doesn't the follow work:

$$
\begin{aligned}
&remdup\ (x : y : zs) \mid x \equiv y &&= remdup\ (y : zs)\\
& \mid otherwise &&= x : remdup\ (y : zs)
\end{aligned}
$$

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

Trick question

Why doesn't the follow work:

$$
\begin{aligned}
remdup\ (x:y:zs)\ &|\ x \equiv y &&= \ remdup\ (y:zs)\\
&|\ otherwise &&= x : remdup\ (y:zs)
\end{aligned}
$$

$$
\begin{aligned}
ham = 1 : remdup\ ((&2 : (map\ (*2)\ (tail\ ham)) \\
&\qquad\qquad `merge` \\
&\qquad\quad (3 : map\ (*3)\ (tail\ ham)) \\
&\quad `merge` \\
&\quad (5 : \quad map\ (*5)\ (tail\ ham)) \\
&)
\end{aligned}
$$

Trick question

Why doesn't the follow work:

$$\begin{aligned} remdup\ (x : y : zs) \mid x \equiv y &= remdup\ (y : zs) \\ \mid otherwise &= x : remdup\ (y : zs) \end{aligned}$$

$$\begin{aligned} ham = 1 : remdup\ (2 : (\ & (map\ (*2)\ (tail\ ham)) \\ & `merge` \\ & (3 : map\ (*3)\ (tail\ ham)) \\ & `merge` \\ & (5 : map\ (*5)\ (tail\ ham)) \\ &) \end{aligned}$$

Trick question

Why doesn't the follow work:

$$\begin{aligned} remdup\ (x : y : zs) &\mid x \equiv y &&= \quad remdup\ (y : zs) \\ &\mid otherwise &&= x : remdup\ (y : zs) \end{aligned}$$

$$\begin{aligned} ham = 1 : remdup\ (2 : (((2 * (head\ (tail\ ham) : map\ (*2)\ (tail\ (tail \\ &\text{`}merge\text{`} \\ &(3 : map\ (*3)\ (tail\ ham)) \\ &\text{`}merge\text{`} \\ &(5 : map\ (*5)\ (tail\ ham)) \\ &) \end{aligned}$$

For $head\ (tail\ ham)$ we need the result of $remdup$!

Productivity

Compare the two definitions of *remdup*

$$
\begin{array}{lll}
\mathit{remdup}\ (x : y : zs) & \mid x \equiv y & = \mathit{remdup}\ (y : zs) \\
 & \mid \mathit{otherwise} & = x : \mathit{remdup}\ (y : zs) \\
\mathit{remdup}'\ (x : ys) & & = x : \mathit{remdup}'\ (\mathit{dropWhile}\ (\equiv x)\ ys)
\end{array}
$$

Productivity

Compare the two definitions of *remdup*

$$
\begin{array}{lll}
remdup\ (x:y:zs) & \mid x \equiv y & = \quad remdup\ (y:zs) \\
 & \mid otherwise & = x : remdup\ (y:zs) \\
remdup'\ (x:ys) & & = x : remdup'\ (dropWhile\ (\equiv x)\ ys)
\end{array}
$$

If we apply these definitions to the sequence $[1, {<}expr1{>}, {<}expr2{>}]$ then the first definition needs the result of ${<}expr1{>}$, before it yields the 1. The second definition yields the 1 directly.

Productivity

Compare the two definitions of *remdup*

$$
\begin{array}{lll}
remdup\ (x:y:zs) & \mid x \equiv y & = \quad remdup\ (y:zs) \\
 & \mid otherwise & = x : remdup\ (y:zs) \\
remdup'\ (x:ys) & & = x : remdup'\ (dropWhile\ (\equiv x)\ ys)
\end{array}
$$

If we apply these definitions to the sequence $[1, <expr1>, <expr2>]$ then the first definition needs the result of $<expr1>$, before it yields the 1. The second definition yields the 1 directly.

Strictness

We say that the second definition is less strict than the first one: it both definitions return something then these values will be the same, but the second definition will evaluate a small part of its argument.

The Fibonacci sequence

Leonardo van Pisa (±1170 – ±1250):

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

$$\begin{array}{lll} \mathit{fib} :: \mathit{Integer} \to \mathit{Integer} \\ \mathit{fib}\ \ 0 & = 0 \\ \mathit{fib}\ \ 1 & = 1 \\ \mathit{fib}\ \ n & = \mathit{fib}\ (n-2) + \mathit{fib}\ (n-1) \end{array}$$

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`):

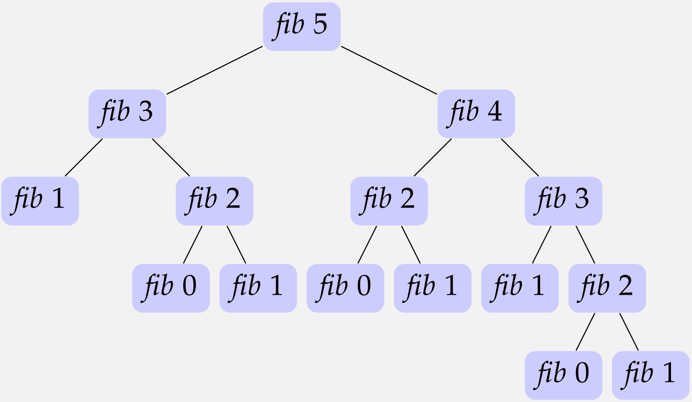
```
Main> fib 10
55

Main> fib 20
6765

Main> fib 25
75025

Main> fib 30
832040
```

Interactive session: number of steps

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

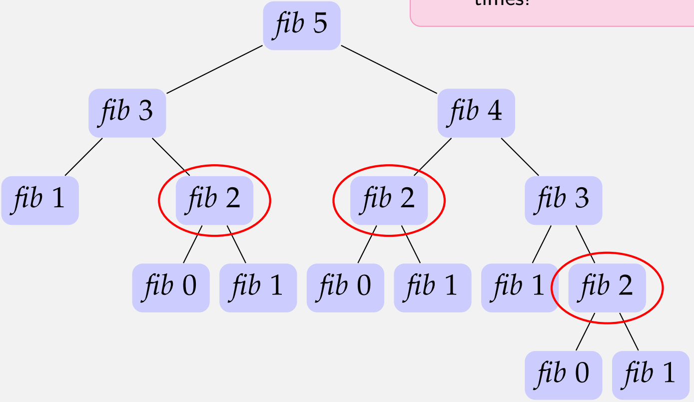
```
Main> fib 10
55
3177 reductions, 5054 cells
Main> fib 20
6765
390861 reductions, 622695 cells
Main> fib 25
75025
4334725 reductions, 6905874 cells, 6 garbage collections
Main> fib 30
832040
48072847 reductions, 76587387 cells, 77 garbage collections
```

Call Tree



fib 5
fib 3
fib 4
fib 1
fib 2
fib 2
fib 3
fib 0
fib 1
fib 0
fib 1
fib 1
fib 2
fib 0
fib 1

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 *fib* calls
5	15
10	177
15	1973
20	21891
25	242785
30	2692537

Local memoïsation

Idea: ‘remember’ the results of the function calls for a sequence of arguments.

Local memoïsation

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

$$
\begin{aligned}
&\mathit{fib} :: \mathit{Integer} \to \mathit{Integer}\\
&\mathit{fib}\ \ n \qquad\quad = \mathit{fibs}\,!\,n\\
&\quad \mathbf{where}\\
&\qquad \mathit{fibs} = \mathit{listArray}\ (0,n)\ \$\\
&\qquad\quad 0 : 1 : [\mathit{fibs}\,!\,(k-2) + \mathit{fibs}\,!\,(k-1) \mid k \leftarrow [2\,..\,n]]
\end{aligned}
$$

Local memoïsation

Idea: ‘remember’ the results of the function calls for a sequence of arguments.

$$
\begin{array}{l}
\mathit{fib} :: \mathit{Integer} \to \mathit{Integer} \\
\mathit{fib}\ \ n \qquad\quad = \mathit{fibs} \,!\, n \\
\quad \textbf{where} \\
\qquad \mathit{fibs} = \mathit{listArray}\ (0, n)\ \$ \\
\qquad\quad 0 : 1 : [\mathit{fibs} \,!\, (k-2) + \mathit{fibs} \,!\, (k-1) \mid k \leftarrow [2 \mathinner{..} n]]
\end{array}
$$

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

Global memoïsation

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 memoïsation

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 memoïsing 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:

$$
\begin{array}{l}
\mathit{fix} :: (a \to a) \to a \\
\mathit{fix}\ \ f = \mathbf{let}\ \mathit{fixf} = f\ \mathit{fixf}\ \mathbf{in}\ \mathit{fixf}
\end{array}
$$

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:

$$
\begin{aligned}
&\mathit{fac} :: \mathit{Integer} \to \mathit{Integer}\\
&\mathit{fac}\quad 0 \qquad = 1\\
&\mathit{fac}\quad n \qquad = n * \mathit{fac}\ (n-1)
\end{aligned}
$$

Explicit recursion

Using fix we can make the use of recursion explicit:

Example:

$$\begin{array}{lll} \mathit{fac} :: \mathit{Integer} \rightarrow \mathit{Integer} & & \\ \mathit{fac}\ \ 0 & = 1 \\ \mathit{fac}\ \ n & = n * \mathit{fac}\ (n-1) \end{array}$$

can, using fix, be written as:

$$\begin{array}{l} \mathit{fac} :: \mathit{Integer} \rightarrow \mathit{Integer} \\ \mathit{fac} = \mathit{fix}\ \mathit{fac}' \\ \quad \textbf{where} \\ \qquad \mathit{fac}'\ f\ 0 = 1 \\ \qquad \mathit{fac}'\ f\ n = n * f\ (n-1) \end{array}$$

Explicit recursion

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

Example:

$$
\begin{aligned}
&\mathit{fac} :: \mathit{Integer} \to \mathit{Integer}\\
&\mathit{fac}\ \ 0 \qquad = 1\\
&\mathit{fac}\ \ n \qquad = n * \mathit{fac}\ (n-1)
\end{aligned}
$$

can, using *fix*, be written as:

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

$$
\begin{aligned}
&\mathit{fac} :: \mathit{Integer} \to \mathit{Integer}\\
&\mathit{fac} = \mathit{fix}\ \mathit{fac}'\\
&\quad \textbf{where}\\
&\qquad \mathit{fac}'\ f\ 0 = 1\\
&\qquad \mathit{fac}'\ f\ n = n * f\ (n-1)
\end{aligned}
$$

Explicit recursion

Using fix we can make the use of recursion explicit:

Example:

$$
\begin{aligned}
&\mathit{fac} :: \mathit{Integer} \to \mathit{Integer}\\
&\mathit{fac}\ \ 0 \quad = 1\\
&\mathit{fac}\ \ n \quad = n * \mathit{fac}\ (n-1)
\end{aligned}
$$

can, using fix, be written as:

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

$$
\begin{aligned}
&\mathit{fac} :: \mathit{Integer} \to \mathit{Integer}\\
&\mathit{fac} = \mathit{fix}\ \mathit{fac}'\\
&\quad \textbf{where}\\
&\qquad \mathit{fac}'\ f\ 0 = 1\\
&\qquad \mathit{fac}'\ f\ n = n * f\ (n-1)
\end{aligned}
$$

☞ $\mathit{fac}' :: (\mathit{Integer} \to \mathit{Integer}) \to (\mathit{Integer} \to \mathit{Integer})$.

Explicit recursion: example

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

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

Fibonacci again

Fibonacci function with explicit recursion

:

$$
\begin{aligned}
&\mathit{fib} :: \mathit{Integer} \rightarrow \mathit{Integer} \\
&\mathit{fib} = \mathit{fix}\ \mathit{fib}' \\
&\quad \textbf{where} \\
&\qquad \mathit{fib}'\ f\ 0 = 0 \\
&\qquad \mathit{fib}'\ f\ 1 = 1 \\
&\qquad \mathit{fib}'\ f\ n = f\ (n-2) + f\ (n-1)
\end{aligned}
$$

Fibonacci again

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

$$
\begin{aligned}
&\mathit{fib} :: \mathit{Integer} \rightarrow \mathit{Integer} \\
&\mathit{fib} = \mathit{fix}\ \mathit{fib} \\
&\quad \textbf{where} \\
&\qquad \mathit{fib}\ \mathit{fib}\ 0 = 0 \\
&\qquad \mathit{fib}\ \mathit{fib}\ 1 = 1 \\
&\qquad \mathit{fib}\ \mathit{fib}\ n = \mathit{fib}\ (n-2) + \mathit{fib}\ (n-1)
\end{aligned}
$$

Idea: replace *fix* by a memoïsing 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*,

$$\begin{array}{ll} \mathit{tabulate} & :: (\mathit{Integer} \rightarrow a) \rightarrow \mathit{Memo}\ a \\ \mathit{apply} & :: \mathit{Memo}\ a \rightarrow \mathit{Integer} \rightarrow a \end{array}$$

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*,

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

such that:

- $\mathit{tabulate}\ f$ results in a (lazily constructed) memo table containing all results of calls to f and
- $\mathit{apply}\ \mathit{mem}\ n$ 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:

$$\textbf{type}\ \mathit{Memo}\ a = [a]$$

Memo lists

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

$$\textbf{type}\ \mathit{Memo}\ a = [a]$$

$$\begin{aligned} &\mathit{tabulate} :: (\mathit{Integer} \to a) \to \mathit{Memo}\ a \\ &\mathit{tabulate}\quad f = \mathit{map}\ f\ [0\,..] \end{aligned}$$

Memo lists

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

$$\textbf{type}\ Memo\ a = [a]$$

$$\begin{aligned}
&tabulate :: (Integer \to a) \to Memo\ a \\
&tabulate\ \ f = map\ f\ [0\,..]
\end{aligned}$$

$$\begin{aligned}
&apply :: Memo\ a \to Integer \to a \\
&apply\ \ (x : _)\ \ 0 = x \\
&apply\ \ (_ : xs)\ n = apply\ xs\ (n - 1)
\end{aligned}$$

Memo combinator

$$
\begin{array}{l}
\mathit{memo} :: ((\mathit{Integer} \to a) \to (\mathit{Integer} \to a)) \to (\mathit{Integer} \to a) \\
\mathit{memo}\ \ f' = f \\
\quad \textbf{where} \\
\qquad f = \mathit{apply}\ (\mathit{tabulate}\ (f'\ f))
\end{array}
$$

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Memo combinator

$$
\begin{array}{l}
\mathit{memo} :: ((\mathit{Integer} \to a) \to (\mathit{Integer} \to a)) \to (\mathit{Integer} \to a) \\
\mathit{memo}\ \ f' = f \\
\quad \textbf{where} \\
\qquad f = \mathit{apply}\ (\mathit{tabulate}\ (f'\ f))
\end{array}
$$

- The combinator constructs a fixpoint f of f'.

Memo combinator

$$
\begin{aligned}
&\mathit{memo} :: ((\mathit{Integer} \to a) \to (\mathit{Integer} \to a)) \to (\mathit{Integer} \to a) \\
&\mathit{memo}\ \ f' = f \\
&\quad \textbf{where} \\
&\qquad f = \mathit{apply}\ (\mathit{tabulate}\ (f'\ f))
\end{aligned}
$$

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

Memo combinator

$$
\begin{aligned}
&\mathit{memo} :: ((\mathit{Integer} \to a) \to (\mathit{Integer} \to a)) \to (\mathit{Integer} \to a) \\
&\mathit{memo}\ \ f' = f \\
&\quad \textbf{where} \\
&\quad\quad f = \mathit{apply}\ (\mathit{tabulate}\ (f'\ f))
\end{aligned}
$$

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

Memo combinator

$$
\begin{aligned}
&\mathit{memo} :: ((\mathit{Integer} \to a) \to (\mathit{Integer} \to a)) \to (\mathit{Integer} \to a) \\
&\mathit{memo}\ \ f' = f \\
&\quad \textbf{where} \\
&\qquad f = \mathit{apply}\ (\mathit{tabulate}\ (f'\ f))
\end{aligned}
$$

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

Memo combinator

$$
\begin{aligned}
&\mathit{memo} :: ((\mathit{Integer} \to a) \to (\mathit{Integer} \to a)) \to (\mathit{Integer} \to a) \\
&\mathit{memo}\ \ f' = f \\
&\quad \textbf{where} \\
&\qquad f = \mathit{apply}\ (\mathit{tabulate}\ (f'\ f))
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{memo} :: ((\mathit{Integer} \to a) \to (\mathit{Integer} \to a)) \to (\mathit{Integer} \to a) \\
&\mathit{memo}\ \ f' = f \\
&\quad \textbf{where} \\
&\qquad f = \mathit{apply}\ (\mathit{tabulate}\ (f'\ f))
\end{aligned}
$$

- The combinator constructs a fixpoint f of f'.
- The function f retreives its result from the memo table $\mathit{tabulate}\ (f'\ f)$.
- 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 memoïsation:

$$
\begin{aligned}
&\mathit{fib} :: \mathit{Integer} \rightarrow \mathit{Integer} \\
&\mathit{fib} = \mathit{memo}\ \mathit{fib}' \\
&\quad \textbf{where} \\
&\qquad \mathit{fib}'\ f\ 0 = 0 \\
&\qquad \mathit{fib}'\ f\ 1 = 1 \\
&\qquad \mathit{fib}'\ f\ n = f\ (n-2) + f\ (n-1)
\end{aligned}
$$

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: lineair search time

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

Memo lists: lineair search time

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

```
Main > fib 5000
387896845438832563370191630832590531208212771 4...
41.78 secs, 2532516300 bytes
```

Memo lists: lineair search time

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

```
Main> fib 5000
387896845438832563370191630832590531208212771 4...
41.78 secs, 2532516300 bytes
```

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

Library for memo functions: plan of attack (unchanged)

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

Define functions *tabulate* and *apply*,

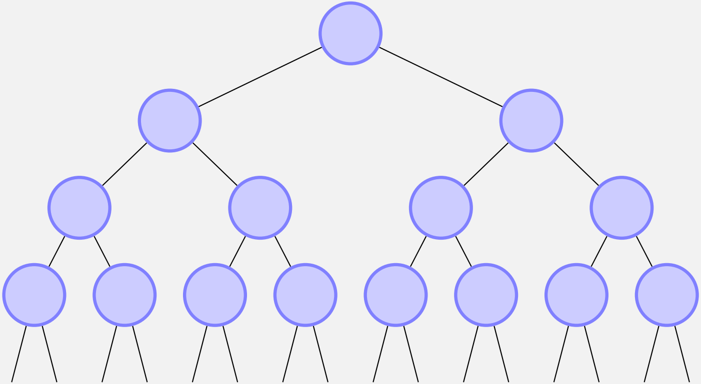
$$\begin{aligned} &tabulate :: (Integer \to a) \to Memo\ a \\ &apply\ \ :: Memo\ a \to Integer \to a \end{aligned}$$

such that:

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

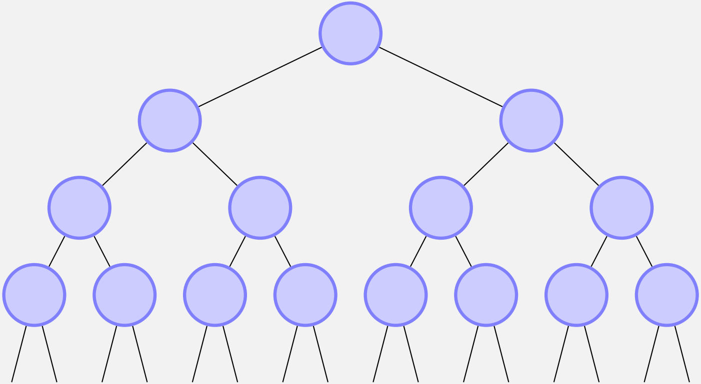
Define a fixedpoint *memo* using *tabulate* and *apply*.

Memo trees



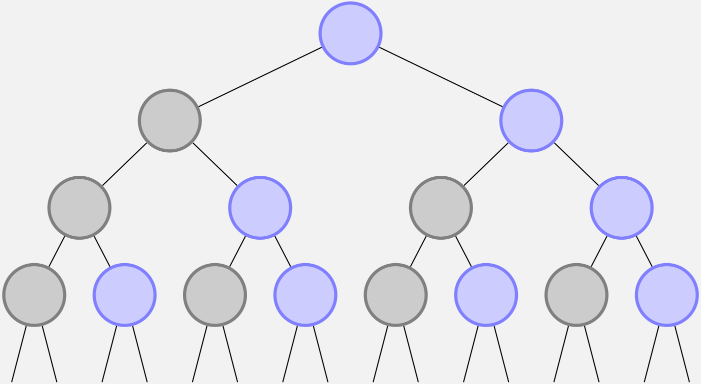
Memo trees

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



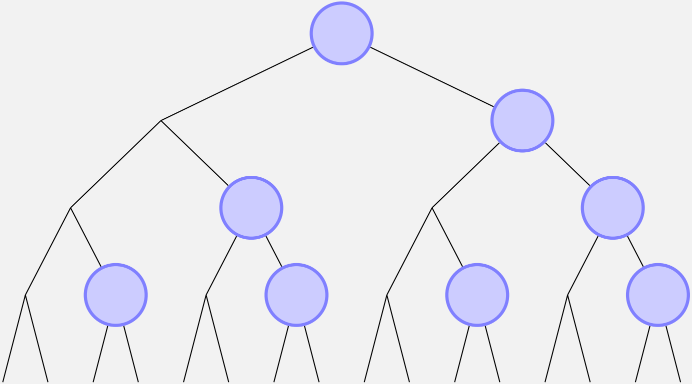
Memo trees

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



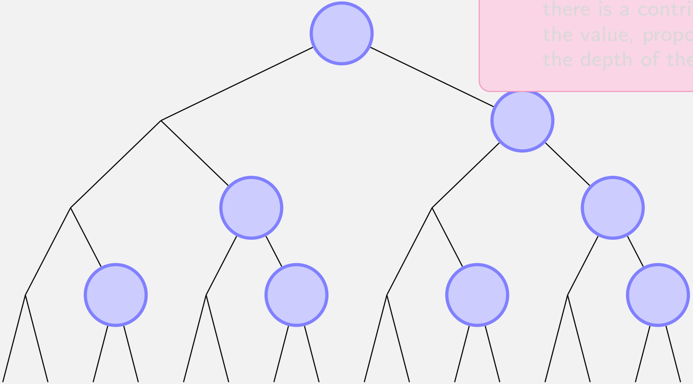
Memo trees

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



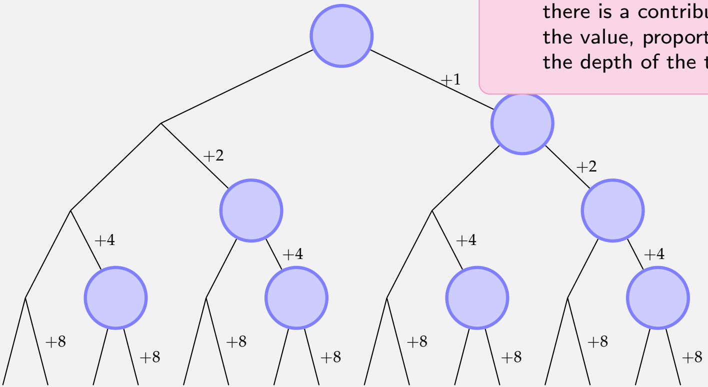
Memo trees

- 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.



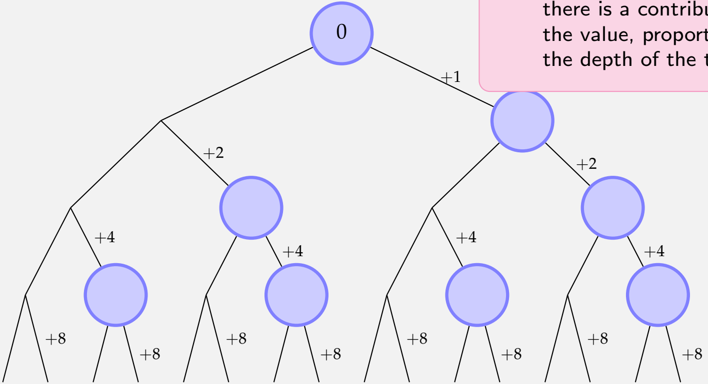
Memo trees

- 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.



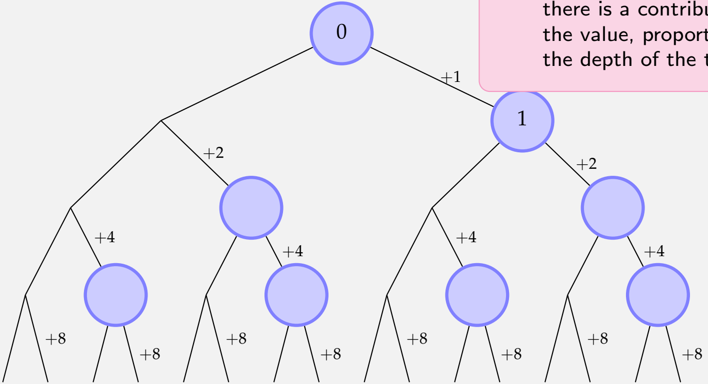
Memo trees

- 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.



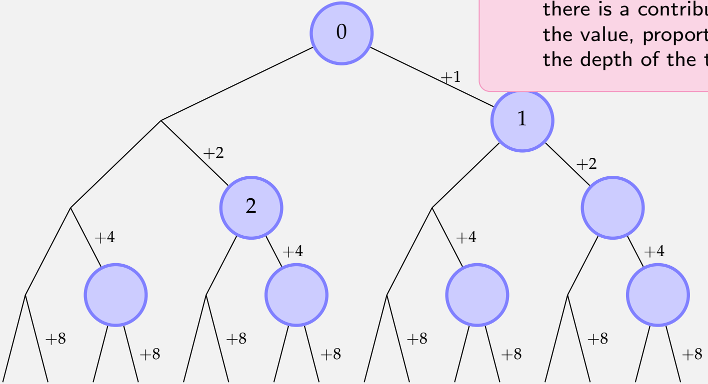
Memo trees

- 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.

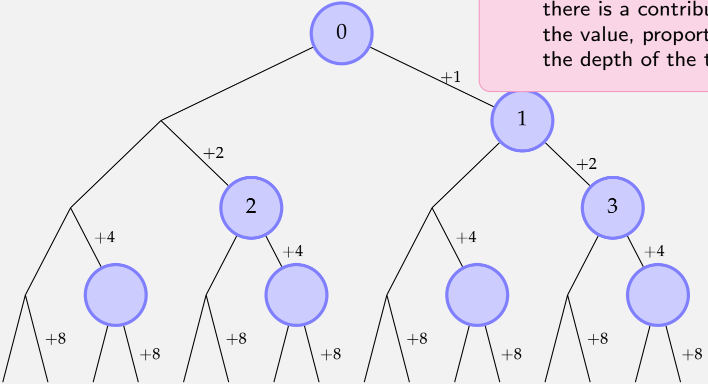


Memo trees

- 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.

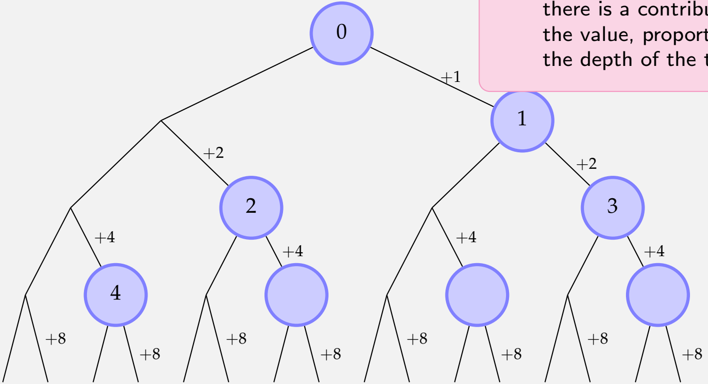


Memo trees



- 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.

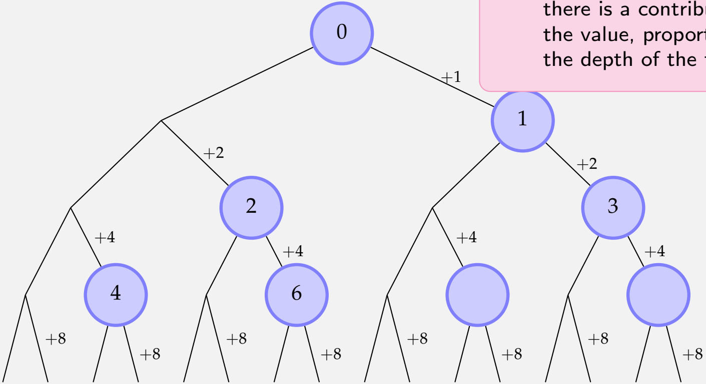
Memo trees



- 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.

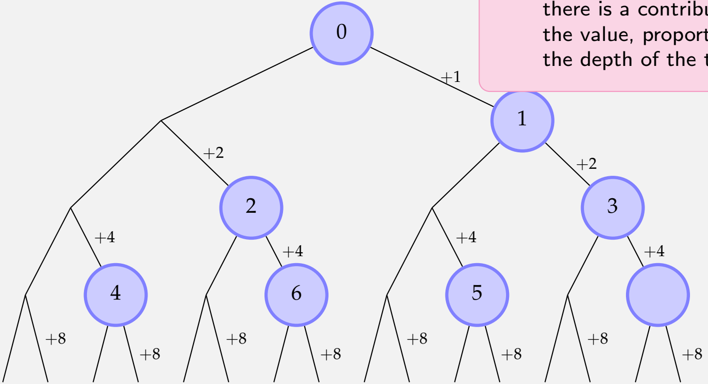
Memo trees

- 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.



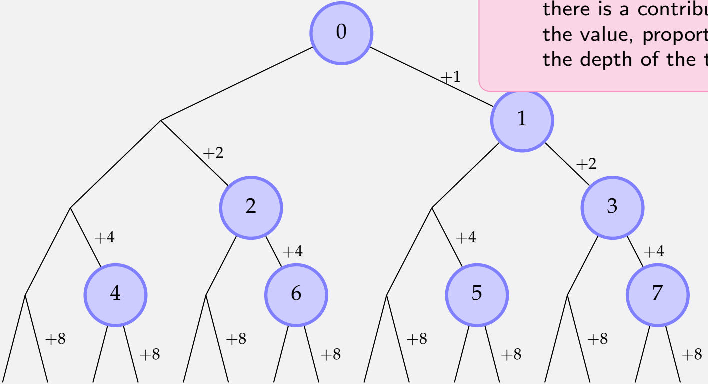
Memo trees

- 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.



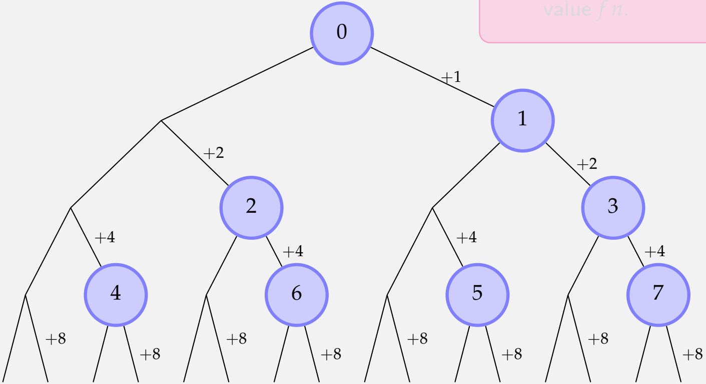
Memo trees

- 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.

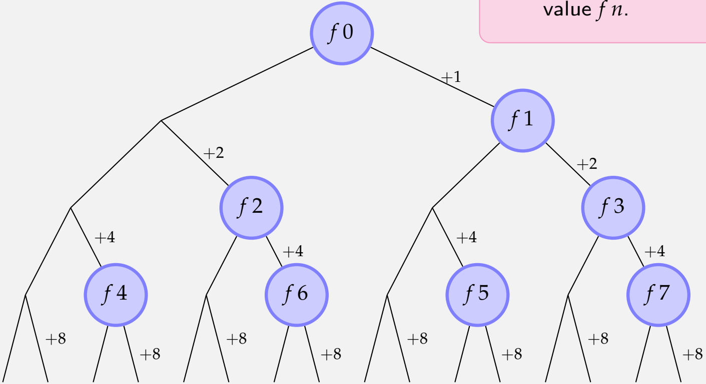


Memo trees

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

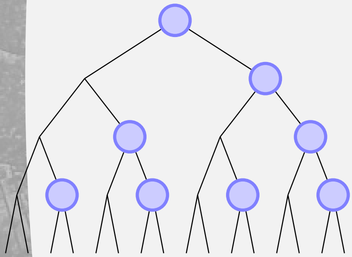


Memo trees



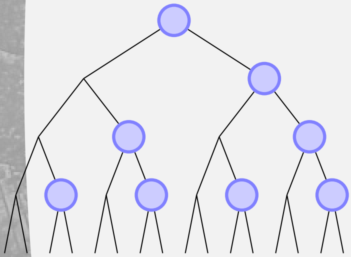
- In the nodes we store the values for the function f which is to be memoïsed.
- In a right child with weight n we store the value $f\ n$.

Data type for memo trees



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

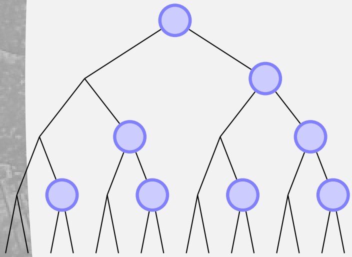
Data type for memo trees



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

$$\textbf{data}\ Memo\ \ a = Memo\ \ (Memo'\ a)\ a\ (Memo\ a)$$
$$\textbf{data}\ Memo'\ a = Memo'\ (Memo'\ a)\ (Memo\ a)$$

Data type for memo trees



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

$$\mathbf{data}\ \mathit{Memo}\ a = \mathit{Memo}\ (\mathit{Memo}'\ a)\ a\ (\mathit{Memo}\ a)$$
$$\mathbf{data}\ \mathit{Memo}'\ a = \mathit{Memo}'\ (\mathit{Memo}'\ a)\ (\mathit{Memo}\ a)$$

☞ *Memo* and *Memo′* are defined mutually recursive.

Construction of the memo tree

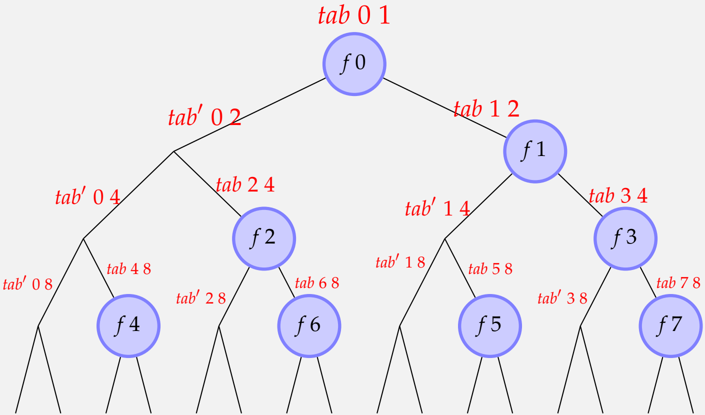
$$
\begin{array}{l}
\mathit{tabulate} :: (\mathit{Integer} \rightarrow a) \rightarrow \mathit{Memo}\ a \\
\mathit{tabulate}\quad f = \mathit{tab}\ 0\ 1 \\
\quad \textbf{where} \\
\qquad \mathit{tab}\ k\ i = \\
\qquad\quad \textbf{let}\ j = 2 * i\ \textbf{in}\ \mathit{Memo}\ (\mathit{tab}'\ k\ j)\ (f\ k)\ (\mathit{tab}\ (k+i)\ j) \\
\qquad \mathit{tab}'\ k\ i = \\
\qquad\quad \textbf{let}\ j = 2 * i\ \textbf{in}\ \mathit{Memo}'\ (\mathit{tab}'\ k\ j)\ (\mathit{tab}\ (k+i)\ j)
\end{array}
$$

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



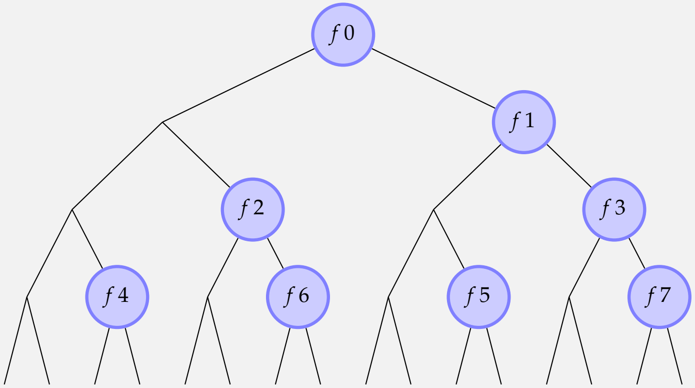
Searching in a memo tree

$$
\begin{array}{l}
apply :: Memo\ a \rightarrow Integer \rightarrow a \\
apply = app \\
\quad \textbf{where} \\
\qquad \begin{array}{ll|ll}
app\ (Memo\ l\ x\ r)\ n & & n \equiv 0 & = x \\
 & & even\ n & = app'\ l\ (n\ \text{`}div\text{`}\ 2) \\
 & & otherwise & = app\ r\ (n\ \text{`}div\text{`}\ 2) \\
app'\ (Memo'\ l\ r)\ n & & even\ n & = app'\ l\ (n\ \text{`}div\text{`}\ 2) \\
 & & otherwise & = app\ r\ (n\ \text{`}div\text{`}\ 2)
\end{array}
\end{array}
$$

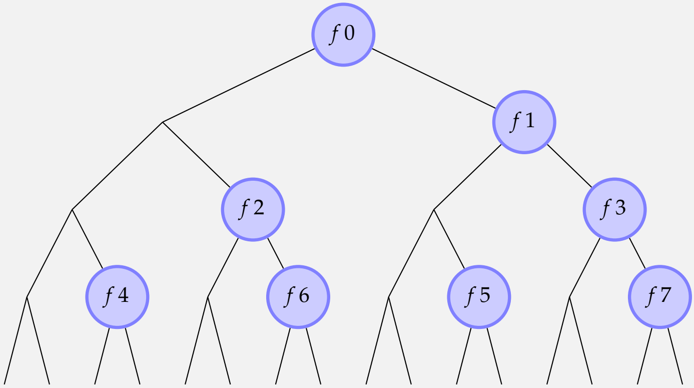
In each recursive step the search key is halved and we decrease one level in the tree

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

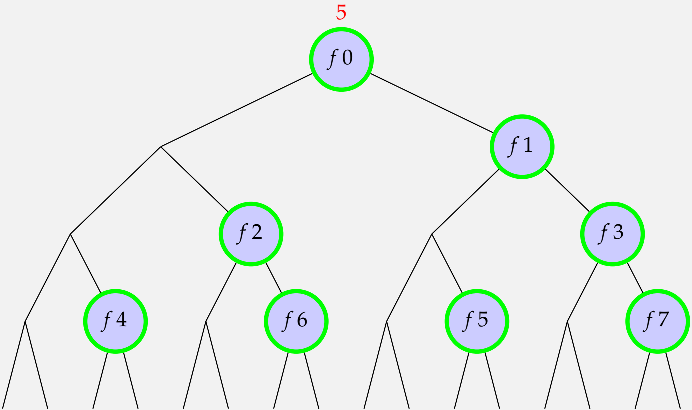
Searching in a memo tree: example



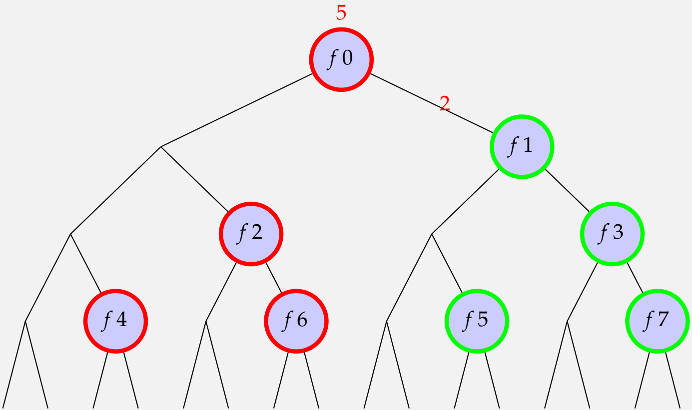
Searching in a memo tree: example



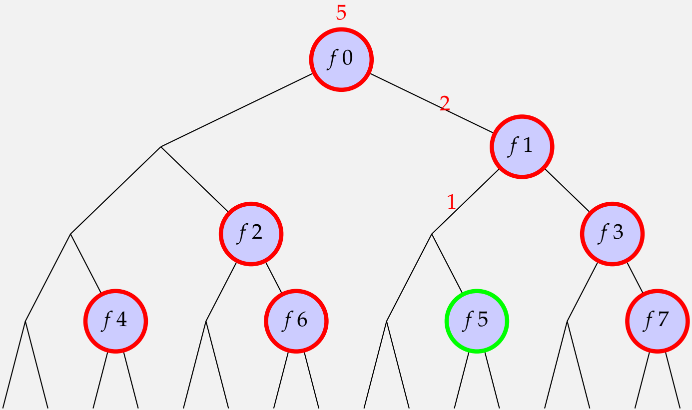
Searching in a memo tree: example



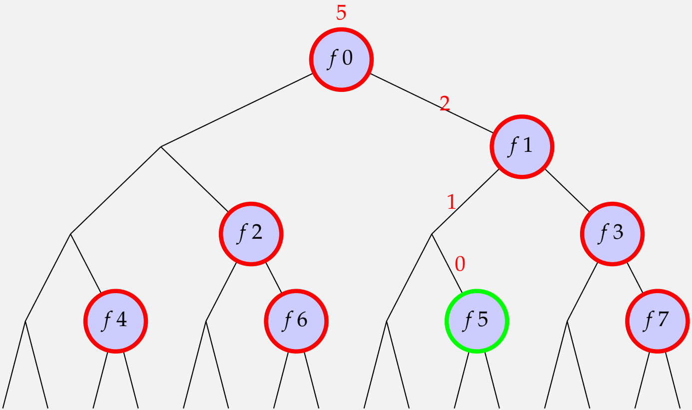
Searching in a memo tree: example



Searching in a memo tree: example



Searching in a memo tree: example



Memo combinator (unchanged)

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

$$
\begin{aligned}
&\mathit{memo} :: ((\mathit{Integer} \rightarrow a) \rightarrow \mathit{Integer} \rightarrow a) \rightarrow \mathit{Integer} \rightarrow a \\
&\mathit{memo} \;\; f' = f \\
&\quad \textbf{where} \\
&\qquad f = \mathit{apply}\ (\mathit{tabulate}\ (f'\ f))
\end{aligned}
$$

Fibonacci sequence using memo trees

$$
\begin{aligned}
&\mathit{fib} :: \mathit{Integer} \to \mathit{Integer} \\
&\mathit{fib} = \mathit{memo}\ \mathit{fib}' \\
&\quad \textbf{where} \\
&\qquad \mathit{fib}'\ f\ 0 = 0 \\
&\qquad \mathit{fib}'\ f\ 1 = 1 \\
&\qquad \mathit{fib}'\ f\ n = f\ (n-2) + f\ (n-1)
\end{aligned}
$$

Memo trees: time and memory usage

```
Main >
```

Universiteit Utrecht

Memo trees: time and memory usage

```
Main > fib 5000
```

Memo trees: time and memory usage

```
Main > fib 5000
387896845438832563370191630832590531208212771 4...
0.37 secs, 26809216 bytes
Main >
```

Memo trees: time and memory usage

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

Universiteit Utrecht

Memo trees: time and memory usage

```
Main> fib 5000
387896845438832563370191630832590531208212771 4...
0.37 secs, 26809216 bytes
Main> fib 5000
387896845438832563370191630832590531208212771 4...
0.02 secs, 532752 bytes
```

Conclusions

- More efficiënt 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 memoïsation for any kind of value that can be mapped onto an *Integer*
- functions with more than one parameter can be memoïsed by having memo tables returning memo tables and having succesive lookups
- is part of several hackage packages, see `http://hackage.haskell.org/package/MemoTrie`