Evaluation

The Practice of Haskell Programming

Andres Löh

Well-Typed

May 18, 2012

Reduction

Reduction

A subexpression that can be reduced is called a **redex**.

Reduction

A subexpression that can be reduced is called a **redex**.

Most typical form of reduction in Haskell: replacing the left hand side of a function definition by a corresponding right hand side (this is essentially **beta reduction** from **lambda calculus**).

Reduction

A subexpression that can be reduced is called a **redex**.

Most typical form of reduction in Haskell: replacing the left hand side of a function definition by a corresponding right hand side (this is essentially **beta reduction** from **lambda calculus**).

Question

What if there are multiple redexes in one term?

Multiple redexes

Many terms have multiple redexes.
How many redexes are in the following term?

id (id (λz $\to$ id z))

Multiple redexes

Many terms have multiple redexes.
How many redexes are in the following term?

id (id (λz $\rightarrow$ id z))

id (id (λz $\rightarrow$ id z))
(id (λz $\rightarrow$ id z))
id z

Multiple redexes

Many terms have multiple redexes.
How many redexes are in the following term?

$\mathsf{id}\ (\mathsf{id}\ (\lambda z \rightarrow \mathsf{id}\ z))$

$\mathsf{id}\ (\mathsf{id}\ (\lambda z \rightarrow \mathsf{id}\ z))$
$\quad (\mathsf{id}\ (\lambda z \rightarrow \mathsf{id}\ z))$
$\qquad\qquad \mathsf{id}\ z$

$(\lambda x \rightarrow \lambda y \rightarrow x * x)\ (1 + 2)\ (3 + 4)$

Multiple redexes

Many terms have multiple redexes.
How many redexes are in the following term?

$$\mathrm{id}\ (\mathrm{id}\ (\lambda z \to \mathrm{id}\ z))$$

$$\mathrm{id}\ (\mathrm{id}\ (\lambda z \to \mathrm{id}\ z))$$
$$(\mathrm{id}\ (\lambda z \to \mathrm{id}\ z))$$
$$\mathrm{id}\ z$$

$$(\lambda x \to \lambda y \to x * x)\ (1 + 2)\ (3 + 4)$$

$$(\lambda x \to \lambda y \to x * x)\ (1 + 2)$$
$$(1 + 2)$$
$$(3 + 4)$$

Operations such as $+$ and $*$ that require their arguments to be (partially) evaluated are called **strict** in their arguments.

Example

Let us play through the possible reductions for the following terms:

```
head (repeat 1)
```

Example

Let us play through the possible reductions for the following terms:

$\mathsf{head}\ (\mathsf{repeat}\ 1)$

$\mathbf{let}\ \mathsf{minimum}\ \mathsf{xs} = \mathsf{head}\ (\mathsf{sort}\ \mathsf{xs})$
$\mathbf{in}\ \mathsf{minimum}\ [4, 1, 3]$

Evaluation strategies

Haskell's lazy evaluation

In Haskell,

- expressions are only evaluated if actually required,
- the leftmost outermost redex is chosen to achieve this,
- sharing is introduced in order to prevent evaluating expressions multiple times.

Haskell's lazy evaluation

In Haskell,

- expressions are only evaluated if actually required,
- the leftmost outermost redex is chosen to achieve this,
- sharing is introduced in order to prevent evaluating expressions multiple times.

If no redexes are left, an expression is in **normal form**. If the top-level of an expression is a constructor or lambda, then the expression is in **(weak) head normal form**.

Common evaluation strategies

Call by value / strict evaluation

Most common. Arguments are reduced as far as possible before reducing a function application, usually left-to-right.

Common evaluation strategies

Call by value / strict evaluation

Most common. Arguments are reduced as far as possible before reducing a function application, usually left-to-right.

Call by name

Functions are reduced before their arguments. Used by some macro languages (TeX, for instance).

Common evaluation strategies

Call by value / strict evaluation

Most common. Arguments are reduced as far as possible before reducing a function application, usually left-to-right.

Call by name

Functions are reduced before their arguments. Used by some macro languages (TeX, for instance).

Call by need / lazy evaluation

Optimized version of "Call by name": function arguments are only reduced when needed, but shared if used multiple times.

λf g x $\rightarrow$ combine (f x) (g x)

Church-Rosser

Theorem (Church-Rosser)

If a term e can be reduced to e_1 and e_2, there is a term e_3 such that both e_1 and e_2 can be reduced to e_3.

Church-Rosser

Theorem (Church-Rosser)

If a term e can be reduced to e_1 and e_2, there is a term e_3 such that both e_1 and e_2 can be reduced to e_3.

Corollary

Each term has at most one normal form.

Church-Rosser

Theorem (Church-Rosser)

If a term e can be reduced to e_1 and e_2, there is a term e_3 such that both e_1 and e_2 can be reduced to e_3.

Corollary

Each term has at most one normal form.

Theorem

If a term has a normal form, then lazy evaluation arrives at this normal form.

Non-termination

In Haskell, we can easily define non-terminating terms:

```
x :: a
x = x
```

Non-termination

In Haskell, we can easily define non-terminating terms:

```
x :: a
x = x
```

Abnormal termination by means of a runtime exception is strongly related to non-termination:

```
undefined :: a
error     :: String → a
```

Non-termination

In Haskell, we can easily define non-terminating terms:

```
x :: a
x = x
```

Abnormal termination by means of a runtime exception is strongly related to non-termination:

```
undefined :: a
error     :: String → a
```

You can see a run-time exception as an “optimization” of a diverging computation.

Strict functions vs. strict evaluation

A function `f` is called **strict** if `f undefined` does not terminate normally.

Strict functions vs. strict evaluation

A function `f` is called **strict** if `f undefined` does not terminate normally.

Note

In a **strict** language, all functions are strict.

In a **non-strict** language, such as Haskell, we have both strict and non-strict functions.

Examples

The function `const` is strict in its first, but not in its second argument.

Examples

The function `const` is strict in its first, but not in its second argument.

The function $(+)$ is strict in both its arguments.

Examples

The function `const` is strict in its first, but not in its second argument.

The function $(+)$ is strict in both its arguments.

The function `map` is not strict in its first argument, but strict in its second.

Examples

The function `const` is strict in its first, but not in its second argument.

The function $(+)$ is strict in both its arguments.

The function `map` is not strict in its first argument, but strict in its second.

However, `map` shows that we often need more finegrained information about evaluation.

Lazy evaluation quiz

$(\lambda x \to x)\ \mathsf{True} \quad \leadsto^*$

$(\lambda x \to x)\ \mathsf{undefined} \quad \leadsto^*$

$(\lambda x \to ())\ \mathsf{undefined} \quad \leadsto^*$

$(\lambda x \to \mathsf{undefined})\ () \quad \leadsto^*$

$(\lambda x\ f \to f\ x)\ \mathsf{undefined} \quad \leadsto^*$

$(\mathsf{error}\ \text{"1"})\ (\mathsf{error}\ \text{"2"}) \quad \leadsto^*$

$\mathsf{length}\ (\mathsf{map}\ \mathsf{undefined}\ [1, 2]) \quad \leadsto^*$

Lazy evaluation quiz

$(\lambda x \rightarrow x)$ True	$\leadsto^*$	True
$(\lambda x \rightarrow x)$ undefined	$\leadsto^*$	
$(\lambda x \rightarrow ())$ undefined	$\leadsto^*$	
$(\lambda x \rightarrow$ undefined$)$ $()$	$\leadsto^*$	
$(\lambda x\ f \rightarrow f\ x)$ undefined	$\leadsto^*$	
(error "1") (error "2")	$\leadsto^*$	
length (map undefined $[1, 2]$)	$\leadsto^*$	

Well-Typed

Lazy evaluation quiz

$(\lambda x \to x)$ True	$\rightsquigarrow^*$	True
$(\lambda x \to x)$ undefined	$\rightsquigarrow^*$	undefined
$(\lambda x \to ())$ undefined	$\rightsquigarrow^*$	
$(\lambda x \to$ undefined$)$ $()$	$\rightsquigarrow^*$	
$(\lambda x\ f \to f\ x)$ undefined	$\rightsquigarrow^*$	
(error "1") (error "2")	$\rightsquigarrow^*$	
length (map undefined $[1, 2]$)	$\rightsquigarrow^*$	

Lazy evaluation quiz

$(\lambda x \rightarrow x)\ \mathsf{True}$	$\leadsto^*$	True
$(\lambda x \rightarrow x)\ \mathsf{undefined}$	$\leadsto^*$	$\mathsf{undefined}$
$(\lambda x \rightarrow ())\ \mathsf{undefined}$	$\leadsto^*$	$()$
$(\lambda x \rightarrow \mathsf{undefined})\ ()$	$\leadsto^*$	
$(\lambda x\ f \rightarrow f\ x)\ \mathsf{undefined}$	$\leadsto^*$	
$(\mathsf{error}\ "1")\ (\mathsf{error}\ "2")$	$\leadsto^*$	
$\mathsf{length}\ (\mathsf{map}\ \mathsf{undefined}\ [1, 2])$	$\leadsto^*$	

Lazy evaluation quiz

$(\lambda x \rightarrow x)$ True	$\rightsquigarrow^*$	True
$(\lambda x \rightarrow x)$ undefined	$\rightsquigarrow^*$	undefined
$(\lambda x \rightarrow ())$ undefined	$\rightsquigarrow^*$	()
$(\lambda x \rightarrow$ undefined) ()	$\rightsquigarrow^*$	undefined
$(\lambda x\ f \rightarrow f\ x)$ undefined	$\rightsquigarrow^*$	
(error "1") (error "2")	$\rightsquigarrow^*$	
length (map undefined [1, 2])	$\rightsquigarrow^*$	

Lazy evaluation quiz

$(\lambda x \rightarrow x)$ True	$\rightsquigarrow^*$	True
$(\lambda x \rightarrow x)$ undefined	$\rightsquigarrow^*$	undefined
$(\lambda x \rightarrow ())$ undefined	$\rightsquigarrow^*$	()
$(\lambda x \rightarrow$ undefined) ()	$\rightsquigarrow^*$	undefined
$(\lambda x\ f \rightarrow f\ x)$ undefined	$\rightsquigarrow^*$	$\lambda f \rightarrow f$ undefined
(error "1") (error "2")	$\rightsquigarrow^*$	
length (map undefined [1, 2])	$\rightsquigarrow^*$	

Lazy evaluation quiz

$(\lambda x \to x)\ \mathsf{True}$	$\leadsto^*$	True
$(\lambda x \to x)\ \mathsf{undefined}$	$\leadsto^*$	$\mathsf{undefined}$
$(\lambda x \to ())\ \mathsf{undefined}$	$\leadsto^*$	$()$
$(\lambda x \to \mathsf{undefined})\ ()$	$\leadsto^*$	$\mathsf{undefined}$
$(\lambda x\ f \to f\ x)\ \mathsf{undefined}$	$\leadsto^*$	$\lambda f \to f\ \mathsf{undefined}$
$(\mathsf{error}\ "1")\ (\mathsf{error}\ "2")$	$\leadsto^*$	$\mathsf{error}\ "1"$
$\mathsf{length}\ (\mathsf{map}\ \mathsf{undefined}\ [1,2])$	$\leadsto^*$	

Lazy evaluation quiz

$(\lambda x \rightarrow x)$ True	$\rightsquigarrow^*$	True
$(\lambda x \rightarrow x)$ undefined	$\rightsquigarrow^*$	undefined
$(\lambda x \rightarrow ())$ undefined	$\rightsquigarrow^*$	()
$(\lambda x \rightarrow$ undefined) ()	$\rightsquigarrow^*$	undefined
$(\lambda x\ f \rightarrow f\ x)$ undefined	$\rightsquigarrow^*$	$\lambda f \rightarrow f$ undefined
(error "1") (error "2")	$\rightsquigarrow^*$	error "1"
length (map undefined [1, 2])	$\rightsquigarrow^*$	2

Example: the first 100 odd square numbers

```
example :: [Int]
example =                                  [1 ..]
```

We start by generating all numbers (lazy evaluation in action).

Example: the first 100 odd square numbers

```
example :: [Int]
example =                    map (λx → x ∗ x)  [1 ..]
```

We use map to compute the square numbers.

Example: the first 100 odd square numbers

$$\begin{array}{l} \text{example} :: [\text{Int}] \\ \text{example} = (\qquad\qquad \text{filter odd} \circ \text{map}\ (\lambda x \rightarrow x * x))\ [1\,..] \end{array}$$

We use function composition composition (and partial application) to subsequently filter the odd square numbers.

Example: the first 100 odd square numbers

$$\begin{array}{l}\mathsf{example} :: [\mathsf{Int}] \\ \mathsf{example} = (\mathsf{take}\ 100 \circ \mathsf{filter}\ \mathsf{odd} \circ \mathsf{map}\ (\lambda \mathsf{x} \to \mathsf{x} * \mathsf{x}))\ [1\,..]\end{array}$$

Finally, we use composition again to take the first 100 elements of this list.

What drives the evaluation?

If we type an expression in at the GHCi prompt:

- GHCi wants to print its result,
- and for printing, we need that expression in normal form,
- that then demands other expressions to be evaluated.

Similarly for a complete program.

What drives the evaluation?

If we type an expression in at the GHCi prompt:

- GHCi wants to print its result,
- and for printing, we need that expression in normal form,
- that then demands other expressions to be evaluated.

Similarly for a complete program.

Within a function, it is most often **pattern matching** that drives the evaluation:

- in order to produce part of the output, we have to select a case;
- in order to be able to choose a case, we have to evaluate some of the arguments just far enough.

What drives the evaluation?

If we type an expression in at the GHCi prompt:

- GHCi wants to print its result,
- and for printing, we need that expression in normal form,
- that then demands other expressions to be evaluated.

Similarly for a complete program.

Within a function, it is most often **pattern matching** that drives the evaluation:

- in order to produce part of the output, we have to select a case;
- in order to be able to choose a case, we have to evaluate some of the arguments just far enough.

Evaluating a term to **weak head normal form** (WHNF) reveals its outermost constructor and allows us to potentially make a choice in a pattern match.

Space leaks and profiling

Haskell data in memory

As we've sketched in the data structures lecture:

- nearly all Haskell data lives on the heap,
- nearly all Haskell data is immutable,
- operations do not change data but rather create new data on the heap,
- a lot of data is shared.

Haskell data in memory

As we’ve sketched in the data structures lecture:

- nearly all Haskell data lives on the heap,
- nearly all Haskell data is immutable,
- operations do not change data but rather create new data on the heap,
- a lot of data is shared.

Sharing is easy because everything is immutable.

Laziness on the heap

Bindings are not evaluated immediately:

- Instead, suspended computations (called **thunks**) are created on the heap.
- Thunks can be shared just as other subterms.
- If a thunk is required, it is evaluated and destructively updated on the heap.
- However, this is a safe and even desirable update – we don't change the value stored, we just change its representation.
- Other computations sharing the updated thunk won't have to recompute the expression.

Garbage collection

GHC uses a generational garbage collector:

- Optimized for lots of short-lived data, as is common in a purely functional language.
- New data is allocated in the "young" generation.
- The young generation is rather small and collected often.
- After a while, data that is still alive is moved to the "old" generation.
- The old generation is larger and collected rarely.
- The heap of a Haskell program can grow dynamically if more memory is needed.

The lifetime of data

Data is alive as long as there are references to it.

The lifetime of data

Data is alive as long as there are references to it.

In a lazy setting, it is sometimes hard to predict how long we retain references to data.

The lifetime of data

Data is alive as long as there are references to it.

In a lazy setting, it is sometimes hard to predict how long we retain references to data.

Space leak

A data structure which grows bigger, or lives longer, than we expect.

As space is a limited resource, we might run (nearly) out of it. Consequences:

- more garbage collections cost extra time,
- swapping,
- program might get killed.

Computing a large sum

$$\begin{aligned}
\mathsf{sum}_1\ [\,] &= 0 \\
\mathsf{sum}_1\ (x : xs) &= x + \mathsf{sum}_1\ xs
\end{aligned}$$

- A straight-forward definition, following the standard pattern of defining functions on lists.
- What is the problem?

Computing a large sum

$$\begin{aligned}
\mathrm{sum}_1\ [\,] &= 0 \\
\mathrm{sum}_1\ (x : xs) &= x + \mathrm{sum}_1\ xs
\end{aligned}$$

- A straight-forward definition, following the standard pattern of defining functions on lists.
- What is the problem?
- If we try to evaluate this function for larger and larger input lists, we note that it takes more and more memory, and significant amounts of time, or we get an error indicating it runs out of stack space.
- But certainly we should be able to sum a list in (nearly) constant (stack) space? What is going on?

Obtaining more information

Haskell's run-time system (RTS) can be instructed to spit out additional information:

- RTS options can be passed to Haskell binaries on the command line by placing them after `+RTS` or enclosing them between `+RTS` and `-RTS`.
- Many RTS flags require the binary to be compiled (or rather linked) using the `-rtsopts` GHC flag.
- You can obtain info about available RTS flags by invoking a compiled binary with `+RTS --help`.
- Very interesting are GC statistics (available in various amounts of detail via `-t`, `-s` or `-S`).
- You can increase the stack space by saying something like `-K50M` or `-K500M`.

GC statistics

```
$ ./Sum1 10000000 +RTS -s -K500M
50000005000000
   1,532,401,936 bytes allocated in the heap
     788,992,048 bytes copied during GC
     457,301,152 bytes maximum residency (10 sample(s))
         740,216 bytes maximum slop
             633 MB total memory in use (0 MB lost due to fragmentation)

                                    Tot time (elapsed)  Avg pause  Max pause
  Gen  0      2299 colls,     0 par    0.83s    0.83s     0.0004s    0.0008s
  Gen  1        10 colls,     0 par    0.60s    0.60s     0.0602s    0.2877s

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    0.46s  (  0.46s elapsed)
  GC      time    1.43s  (  1.43s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    1.89s  (  1.88s elapsed)

  %GC     time      75.8%  (75.8% elapsed)

  Alloc rate    3,352,283,510 bytes per MUT second

  Productivity  24.2% of total user, 24.3% of total elapsed
```

MUT (mutator) time is good, GC time is bad.

Maximum residency and percentage of GC time are revealing.

Heap profiling

More detailed information can be obtained using heap profiling.

- Requires recompilation of the program (makes program larger and overall slower).
- All used libraries must have profiling versions, too.
- In your cabal-install `config` file, put

```
library-profiling: True
```

  for the future.
- Compile a program with profiling enabled:

```
$ ghc --make -prof -auto-all -rtsopts Sum1
```

  The `-auto-all` is optional. It is more important for larger programs where you not only want to know **how much** space is being used, but also **where** it is being used.

Heap profiling – contd.

- Run with profiling enabled:

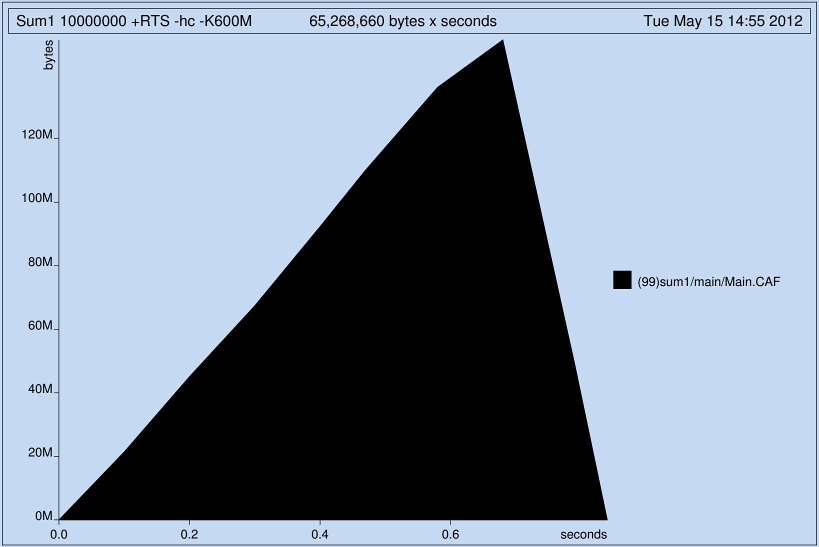
```
$ ./Sum1 10000000 +RTS -K800M -hc
```

  Again, there are many different `-h` flags.

- The `-hc` is for cost-center profiling.
- A very simplistic form of heap profiling via just `-h` is available even without compiling the program for profiling. It would be sufficient here!
- Files `Sum1.prof` and `Sum1.hp` are produced.
- The `.hp` file can be transformed into PostScript format using the `hp2ps` tool.

```
$ hp2ps Sum1.hp
```

Heap profile for sum_1



The problem

$$
\begin{aligned}
& \mathsf{sum}_1\ [1,2,3,4,\ldots] \\
\equiv\ & \quad \{\ \text{Definition of}\ \ \mathsf{sum}_1\ \ \} \\
& 1 + \mathsf{sum}_1\ [2,3,4,\ldots] \\
\equiv\ & \quad \{\ \text{Definition of}\ \ \mathsf{sum}_1\ \ \} \\
& 1 + (2 + \mathsf{sum}_1\ [3,4,\ldots]) \\
\equiv\ & \quad \{\ \text{Definition of}\ \ \mathsf{sum}_1\ \ \} \\
& 1 + (2 + (3 + \mathsf{sum}_1\ [4,\ldots])) \\
\equiv\ & \\
& \ldots
\end{aligned}
$$

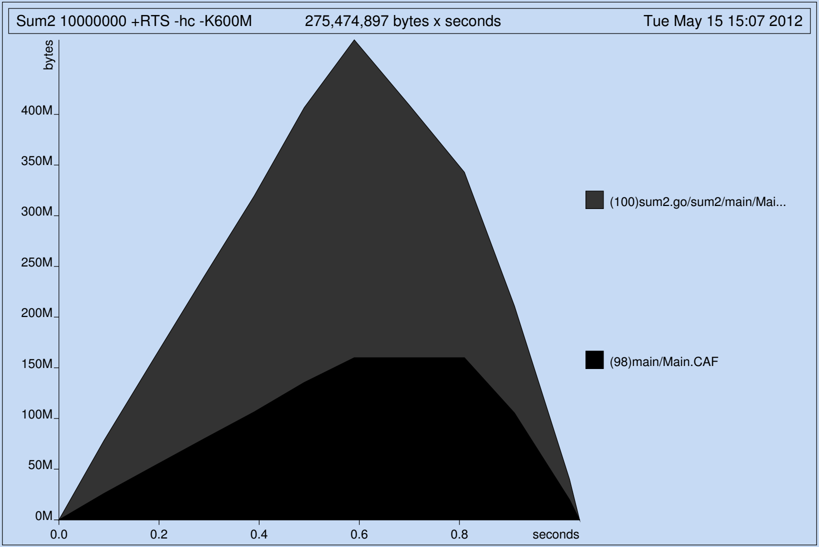
The whole recursion has to be unfolded before the first addition can be reduced!

Attempting a tail-recursive version

```
sum₂ xs = go 0 xs
  where
    go acc []       = acc
    go acc (x : xs) = go (acc + x) xs
```

We hope that tail-recursion improves stack usage, and might thereby improve space behaviour as well, but ...

Heep profile for sum$_2$



Well-Typed

The new problem

$$
\begin{aligned}
& \mathsf{sum}_2\ [1, 2, 3, 4, \ldots] \\
\equiv\ & \quad \{\ \text{Definition of}\ \ \mathsf{sum}_2\ \ \} \\
& \mathsf{sum}_2'\ 0\ [1, 2, 3, 4, \ldots] \\
\equiv\ & \quad \{\ \text{Definition of}\ \ \mathsf{sum}_2\ \ \} \\
& \mathsf{sum}_2'\ (0 + 1)\ [2, 3, 4, \ldots] \\
\equiv\ & \quad \{\ \text{Definition of}\ \ \mathsf{sum}_2\ \ \} \\
& \mathsf{sum}_2'\ ((0 + 1) + 2)\ [3, 4, \ldots] \\
& \ldots \\
\equiv\ & \\
& \mathsf{sum}_2'\ (\ldots((0 + 1) + 2)\ldots)\ [\,] \\
\equiv\ & \quad \{\ \text{Definition of}\ \ \mathsf{sum}_2\ \ \} \\
& (\ldots(0 + 1) + 2)\ldots)
\end{aligned}
$$

We still build up the whole addition, but now in an accumulating argument! Evaluating that still takes stack!

Controlling evaluation

We need more control

Sometimes, we want to make things stricter than they are by default. Here:

- we have a computation that will be evaluated anyway,
- storing it in delayed form costs much more space than storing its result.

Forcing evaluation

Haskell has the following primitive function

```
seq :: a → b → b   -- primitive
```

The call `seq x y` is strict in `x` and returns `y`.

Forcing evaluation

Haskell has the following primitive function

```
seq :: a → b → b    -- primitive
```

The call `seq x y` is strict in `x` and returns `y`.

The function `seq` can be used to define strict function application:

```
($!) :: (a → b) → a → b
f $! x = x `seq` f x
```

Recall sharing!

Forcing quiz

The function `seq` only evaluates to WHNF (i.e., a lambda abstraction, literal or constructor application).

Forcing quiz

The function `seq` only evaluates to WHNF (i.e., a lambda abstraction, literal or constructor application).

$(\lambda x \to ())$ \$! undefined	$\leadsto^*$
seq (error "1", error "2") ()	$\leadsto^*$
snd \$! (error "1", error "2")	$\leadsto^*$
$(\lambda x \to ())$ \$! $(\lambda x \to$ undefined$)$	$\leadsto^*$
error "1" \$! error "2"	$\leadsto^*$
length \$! map undefined [1, 2]	$\leadsto^*$
seq (error "1" + error "2") ()	$\leadsto^*$
seq (1 : undefined) ()	$\leadsto^*$

Forcing quiz

The function `seq` only evaluates to WHNF (i.e., a lambda abstraction, literal or constructor application).

$(\lambda x \to ())$ \$! undefined	$\leadsto^*$	undefined
seq (error "1", error "2") ()	$\leadsto^*$	
snd \$! (error "1", error "2")	$\leadsto^*$	
$(\lambda x \to ())$ \$! $(\lambda x \to$ undefined)	$\leadsto^*$	
error "1" \$! error "2"	$\leadsto^*$	
length \$! map undefined [1, 2]	$\leadsto^*$	
seq (error "1" + error "2") ()	$\leadsto^*$	
seq (1 : undefined) ()	$\leadsto^*$	

Forcing quiz

The function `seq` only evaluates to WHNF (i.e., a lambda abstraction, literal or constructor application).

$(\lambda x \rightarrow ())$ \$! undefined	$\leadsto^*$	undefined
seq (error "1", error "2") ()	$\leadsto^*$	()
snd \$! (error "1", error "2")	$\leadsto^*$	
$(\lambda x \rightarrow ())$ \$! $(\lambda x \rightarrow$ undefined$)$	$\leadsto^*$	
error "1" \$! error "2"	$\leadsto^*$	
length \$! map undefined [1, 2]	$\leadsto^*$	
seq (error "1" + error "2") ()	$\leadsto^*$	
seq (1 : undefined) ()	$\leadsto^*$	

Forcing quiz

The function `seq` only evaluates to WHNF (i.e., a lambda abstraction, literal or constructor application).

$(\lambda x \rightarrow ())\ \$!\ \mathrm{undefined}$	$\rightsquigarrow^*$	$\mathrm{undefined}$
$\mathrm{seq}\ (\mathrm{error}\ "1", \mathrm{error}\ "2")\ ()$	$\rightsquigarrow^*$	$()$
$\mathrm{snd}\ \$!\ (\mathrm{error}\ "1", \mathrm{error}\ "2")$	$\rightsquigarrow^*$	$\mathrm{error}\ "2"$
$(\lambda x \rightarrow ())\ \$!\ (\lambda x \rightarrow \mathrm{undefined})$	$\rightsquigarrow^*$	
$\mathrm{error}\ "1"\ \$!\ \mathrm{error}\ "2"$	$\rightsquigarrow^*$	
$\mathrm{length}\ \$!\ \mathrm{map}\ \mathrm{undefined}\ [1, 2]$	$\rightsquigarrow^*$	
$\mathrm{seq}\ (\mathrm{error}\ "1" + \mathrm{error}\ "2")\ ()$	$\rightsquigarrow^*$	
$\mathrm{seq}\ (1 : \mathrm{undefined})\ ()$	$\rightsquigarrow^*$	

Forcing quiz

The function `seq` only evaluates to WHNF (i.e., a lambda abstraction, literal or constructor application).

$(\lambda x \to ())\ \$!\ \text{undefined}$	$\leadsto^*$	undefined
$\text{seq}\ (\text{error}\ "1", \text{error}\ "2")\ ()$	$\leadsto^*$	$()$
$\text{snd}\ \$!\ (\text{error}\ "1", \text{error}\ "2")$	$\leadsto^*$	$\text{error}\ "2"$
$(\lambda x \to ())\ \$!\ (\lambda x \to \text{undefined})$	$\leadsto^*$	$()$
$\text{error}\ "1"\ \$!\ \text{error}\ "2"$	$\leadsto^*$	
$\text{length}\ \$!\ \text{map undefined}\ [1, 2]$	$\leadsto^*$	
$\text{seq}\ (\text{error}\ "1" + \text{error}\ "2")\ ()$	$\leadsto^*$	
$\text{seq}\ (1 : \text{undefined})\ ()$	$\leadsto^*$	

Forcing quiz

The function `seq` only evaluates to WHNF (i.e., a lambda abstraction, literal or constructor application).

$(\lambda x \to ())$ \$! undefined	$\leadsto^*$	undefined
seq (error "1", error "2") ()	$\leadsto^*$	()
snd \$! (error "1", error "2")	$\leadsto^*$	error "2"
$(\lambda x \to ())$ \$! $(\lambda x \to$ undefined)	$\leadsto^*$	()
error "1" \$! error "2"	$\leadsto^*$	error "2"
length \$! map undefined [1, 2]	$\leadsto^*$	
seq (error "1" + error "2") ()	$\leadsto^*$	
seq (1 : undefined) ()	$\leadsto^*$	

Forcing quiz

The function `seq` only evaluates to WHNF (i.e., a lambda abstraction, literal or constructor application).

$(\lambda x \to ())$ \$! undefined	$\leadsto^*$	undefined
seq (error "1", error "2") ()	$\leadsto^*$	()
snd \$! (error "1", error "2")	$\leadsto^*$	error "2"
$(\lambda x \to ())$ \$! $(\lambda x \to$ undefined$)$	$\leadsto^*$	()
error "1" \$! error "2"	$\leadsto^*$	error "2"
length \$! map undefined [1, 2]	$\leadsto^*$	2
seq (error "1" + error "2") ()	$\leadsto^*$	
seq (1 : undefined) ()	$\leadsto^*$	

Forcing quiz

The function `seq` only evaluates to WHNF (i.e., a lambda abstraction, literal or constructor application).

$(\lambda x \rightarrow ())$ \$! undefined	$\leadsto^*$	undefined
seq (error "1", error "2") ()	$\leadsto^*$	()
snd \$! (error "1", error "2")	$\leadsto^*$	error "2"
$(\lambda x \rightarrow ())$ \$! $(\lambda x \rightarrow$ undefined)	$\leadsto^*$	()
error "1" \$! error "2"	$\leadsto^*$	error "2"
length \$! map undefined [1, 2]	$\leadsto^*$	2
seq (error "1" + error "2") ()	$\leadsto^*$	error "1"
seq (1 : undefined) ()	$\leadsto^*$	

Forcing quiz

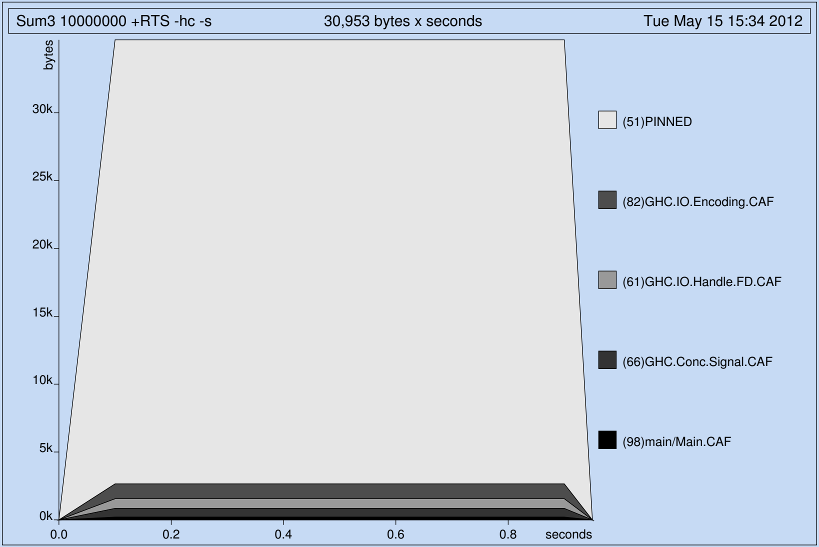
The function `seq` only evaluates to WHNF (i.e., a lambda abstraction, literal or constructor application).

$(\lambda x \rightarrow ())\ \$!\ \mathrm{undefined}$	$\rightsquigarrow^*$	$\mathrm{undefined}$
$\mathrm{seq}\ (\mathrm{error}\ "1", \mathrm{error}\ "2")\ ()$	$\rightsquigarrow^*$	$()$
$\mathrm{snd}\ \$!\ (\mathrm{error}\ "1", \mathrm{error}\ "2")$	$\rightsquigarrow^*$	$\mathrm{error}\ "2"$
$(\lambda x \rightarrow ())\ \$!\ (\lambda x \rightarrow \mathrm{undefined})$	$\rightsquigarrow^*$	$()$
$\mathrm{error}\ "1"\ \$!\ \mathrm{error}\ "2"$	$\rightsquigarrow^*$	$\mathrm{error}\ "2"$
$\mathrm{length}\ \$!\ \mathrm{map}\ \mathrm{undefined}\ [1, 2]$	$\rightsquigarrow^*$	2
$\mathrm{seq}\ (\mathrm{error}\ "1" + \mathrm{error}\ "2")\ ()$	$\rightsquigarrow^*$	$\mathrm{error}\ "1"$
$\mathrm{seq}\ (1 : \mathrm{undefined})\ ()$	$\rightsquigarrow^*$	$()$

Using `seq` to force the addition

```
sum₃ xs = go 0 xs
  where
    go acc []       = acc
    go acc (x : xs) = (go $! acc + x) xs
```

Heap profile for `sum3`



GC statistics

```
$ ./Sum3 10000000 +RTS -hc -s
50000005000000
   2,560,118,208 bytes allocated in the heap
         714,144 bytes copied during GC
          62,104 bytes maximum residency (10 sample(s))
          26,344 bytes maximum slop
               1 MB total memory in use (0 MB lost due to fragmentation)

                                    Tot time (elapsed)  Avg pause  Max pause
  Gen  0      4873 colls,     0 par    0.02s    0.02s     0.0000s    0.0000s
  Gen  1        10 colls,     0 par    0.00s    0.00s     0.0001s    0.0001s

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    0.95s  (  0.95s elapsed)
  GC      time    0.02s  (  0.02s elapsed)
  RP      time    0.00s  (  0.00s elapsed)
  PROF    time    0.00s  (  0.00s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    0.98s  (  0.98s elapsed)

  %GC     time       2.3%  (2.2% elapsed)

  Alloc rate    2,684,947,785 bytes per MUT second

  Productivity  97.6% of total user, 97.6% of total elapsed
```

Look at the maximum residency and GC time / productivity now.

Standard recursion patterns

The three versions of `sum` we have seen correspond to using `foldr`, `foldl` and `foldl'`, respectively:

$$\mathsf{sum}_1 = \mathsf{foldr}\ (+)\ 0$$
$$\mathsf{sum}_2 = \mathsf{foldl}\ (+)\ 0$$
$$\mathsf{sum}_3 = \mathsf{foldl}'\ (+)\ 0$$

Question

Is using `foldl'`/strictness always preferable?

Question

Is using `foldl'`/strictness always preferable?

For example, what about defining `map` …

Rules of thumb

- If you expect partial results or want to use infinite lists, use `foldr`.

  Examples: `map`, `filter`.
- If the operator is strict, use `foldl'`.

  Examples: `sum`, `product`.
- Otherwise, use `foldl`.

  Examples: `reverse`.
- Use the GHC optimizer by passing `-O`. GHC performs strictness analysis to optimize your code – but don't rely on it to always figure out everything!