Lazy Evaluation and Profiling

Haskell Performance

Andres Löh

Well-Typed
The Haskell Consultants

The plan

We want to get a better idea about **what** gets evaluated **when**:

- This is relevant once performance or space use are unexpectedly bad.
- It is also quite important for computations that should run in parallel.
- We will first look at different evaluation strategies.
- Then, we discuss how we can influence the default evaluation in Haskell when we are not happy with it.

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?

```haskell
id (id (\ z -> id z))
```

Multiple redexes

Many terms have multiple redexes.

How many redexes are in the following term?

```
id (id (\ z -> id z))
```

Three redexes:

```
id (id (\ z -> id z))
   (id (\ z -> id z))
               id z
```

Well-Typed

Multiple redexes

Many terms have multiple redexes.

How many redexes are in the following term?

```
id (id (\ z -> id z))
```

```
(\ x -> \ y -> x * x) (1 + 2) (3 + 4)
```

Three redexes:

```
id (id (\ z -> id z))
   (id (\ z -> id z))
              id z
```

Multiple redexes

Many terms have multiple redexes.

How many redexes are in the following term?

```
id (id (\ z -> id z))
```

Three redexes:

```
id (id (\ z -> id z))
   (id (\ z -> id z))
               id z
```

```
(\ x -> \ y -> x * x) (1 + 2) (3 + 4)
```

Three as well:

```
(\ x -> \ y -> x * x) (1 + 2)
                      (1 + 2)
                              (3 + 4)
```

Well-Typed

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:

```
head (repeat 1)
```

```
let
  minimum xs = head (sort xs)
in
  minimum [4, 1, 3]
```

Well-Typed

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 (whenever an identifier is bound to an expression) 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 (whenever an identifier is bound to an expression) 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**.

Well-Typed

Common evaluation strategies

Call by value / eager (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 / eager (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 (contd.)

Call by need / lazy evaluation

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

```
\ f g x -> 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.

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 (\ x -> x * x)  [1 ..]
```

We use `map` to compute the square numbers.

Example: the first 100 odd square numbers

```
example :: [Int]
example = (           filter odd . map (\ x -> x * x)) [1 ..]
```

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

Example: the first 100 odd square numbers

```haskell
example :: [Int]
example = (take 100 . filter odd . map (\ x -> x * x)) [1 ..]
```

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.

Well-Typed

Tracking demand

If we want to understand what gets evaluated when, we have to track the demand, and this usually happens **backwards**.

Typical question

If we need the result of a function to be evaluated to WHNF, what effect (if any) does this have on the argument(s) of the function?

Let us look at a few examples.

Tracking demand (contd.)

```
id x = x
```

Demanding the result causes demanding the argument.

Tracking demand (contd.)

```
id x = x
```

Demanding the result causes demanding the argument.

```
const x y = x
```

Demanding the result causes demanding the first argument (but not the second).

Tracking demand (contd.)

```haskell
id x = x
```

Demanding the result causes demanding the argument.

```haskell
const x y = x
```

Demanding the result causes demanding the first argument (but not the second).

```haskell
True  || _ = True
False || y = y
```

Demanding the result causes demanding the first argument, and depending on that might cause demanding the second.

Well-Typed

Tracking demand (contd.)

```
map f []       = []
map f (x : xs) = f x : map f xs
```

Demanding the result causes demanding the second argument (but not the first).

Devising a test for demand propagation

Do we have to look at the code in order to track demand?

Devising a test for demand propagation

Do we have to look at the code in order to track demand?

Idea:

- Feed a non-terminating computation into a function and demand its result.

Devising a test for demand propagation

Do we have to look at the code in order to track demand?

Idea:

- Feed a non-terminating computation into a function and demand its result.
- If the function produces a result, then it cannot have demanded its argument.

Devising a test for demand propagation

Do we have to look at the code in order to track demand?

Idea:

- Feed a non-terminating computation into a function and demand its result.
- If the function produces a result, then it cannot have demanded its argument.
- If the function loops, then we do not know whether we demanded the argument, or the loop arises from elsewhere – but (assuming we identify all loops and run-time exceptions), it is safe to assume that it did, as it would not have changed the result.

Strict functions

Definition

A (one-argument) function `f` is called **strict** if and only if

```
f ⊥ = ⊥
```

Here, `⊥` denotes any crashing or looping computation.

Strict functions

Definition

A (one-argument) function `f` is called **strict** if and only if

```
f ⊥ = ⊥
```

Here, `⊥` denotes any crashing or looping computation.

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.

Well-Typed

Lazy evaluation quiz

```
(\ x -> x) True          ⇝*
(\ x -> x) ⊥             ⇝*
(\ x -> ()) ⊥            ⇝*
(\ x -> ⊥) ()            ⇝*
(\ x f -> f x) ⊥         ⇝*
length (map ⊥ [1, 2])    ⇝*
```

Well-Typed

Lazy evaluation quiz

```
(\ x -> x) True          ⇝*  True
(\ x -> x) ⊥             ⇝*
(\ x -> ()) ⊥            ⇝*
(\ x -> ⊥) ()            ⇝*
(\ x f -> f x) ⊥         ⇝*
length (map ⊥ [1, 2])    ⇝*
```

Lazy evaluation quiz

```
(\ x -> x) True              ⇝*  True
(\ x -> x) ⊥                 ⇝*  ⊥
(\ x -> ()) ⊥                ⇝*
(\ x -> ⊥) ()                ⇝*
(\ x f -> f x) ⊥             ⇝*
length (map ⊥ [1, 2])        ⇝*
```

Well-Typed

Lazy evaluation quiz

```
(\ x -> x) True          ⇝*  True
(\ x -> x) ⊥             ⇝*  ⊥
(\ x -> ()) ⊥            ⇝*  ()
(\ x -> ⊥) ()            ⇝*
(\ x f -> f x) ⊥         ⇝*
length (map ⊥ [1, 2])    ⇝*
```

Well-Typed

Lazy evaluation quiz

```
(\ x -> x) True           ⇝*  True
(\ x -> x) ⊥              ⇝*  ⊥
(\ x -> ()) ⊥             ⇝*  ()
(\ x -> ⊥) ()             ⇝*  ⊥
(\ x f -> f x) ⊥          ⇝*
length (map ⊥ [1, 2])     ⇝*
```

Well-Typed

Lazy evaluation quiz

```
(\ x -> x) True          ⇝*  True
(\ x -> x) ⊥             ⇝*  ⊥
(\ x -> ()) ⊥            ⇝*  ()
(\ x -> ⊥) ()            ⇝*  ⊥
(\ x f -> f x) ⊥         ⇝*  \ f -> f ⊥
length (map ⊥ [1, 2])    ⇝*
```

Lazy evaluation quiz

```
(\ x -> x) True           ⇝*  True
(\ x -> x) ⊥              ⇝*  ⊥
(\ x -> ()) ⊥             ⇝*  ()
(\ x -> ⊥) ()             ⇝*  ⊥
(\ x f -> f x) ⊥          ⇝*  \ f -> f ⊥
length (map ⊥ [1, 2])     ⇝*  2
```

Observing evaluation

For debugging purposes, you can observe when something gets evaluated using `trace` from `Debug.Trace`:

```
trace     :: String -> a -> a
traceShow :: Show a => a -> b -> b
```

The `trace` functions print their first argument as soon as the second is being evaluated.

Observing evaluation

For debugging purposes, you can observe when something gets evaluated using `trace` from `Debug.Trace` :

```haskell
trace     :: String -> a -> a
traceShow :: Show a => a -> b -> b
```

The `trace` functions print their first argument as soon as the second is being evaluated.

Question: How many symbols are printed, in what order?

```haskell
x .: xs = trace ":" (x : xs)
nil     = trace "." []
num x   = trace "0" x
list    = foldr (.:) nil (map num [1 .. 10])
main    = print $ sum (take 2 (drop 3 list))
```

Well-Typed

Space leaks and profiling

Haskell data in memory

As we’ve discussed earlier:

- 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 discussed earlier:

- 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.

Well-Typed

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

```
sum1 []       = 0
sum1 (x : xs) = x + sum1 xs
```

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

Computing a large sum

```
sum1 []       = 0
sum1 (x : xs) = x + sum1 xs
```

- 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?

Well-Typed

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.

Well-Typed

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:

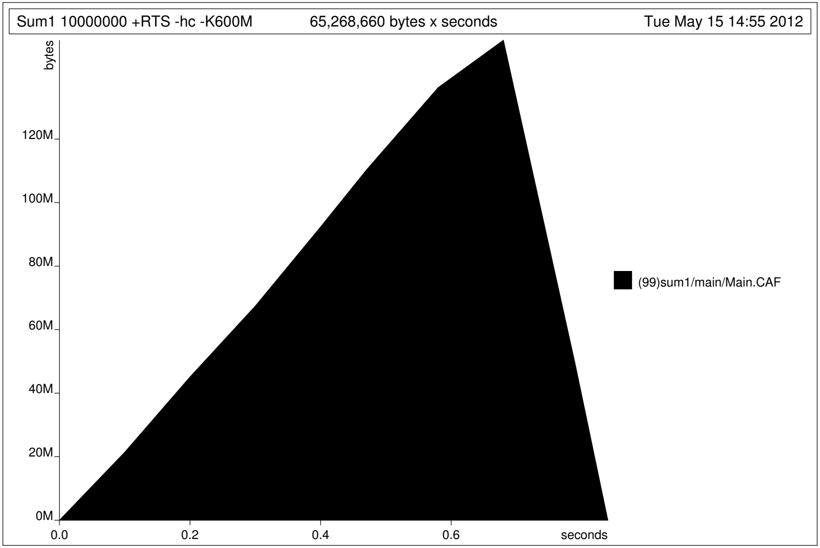
```
$ ./Sum1 10000000 +RTS -K800M -hc
```

  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 `sum1`



The problem

```
   sum1 [1, 2, 3, 4, ...]
=    { Definition of sum1 }
   1 + sum1 [2, 3, 4, ...]
=    { Definition of sum1 }
   1 + (2 + sum1 [3, 4, ...])
=    { Definition of sum1 }
   1 + (2 + (3 + sum1 [4, ...]))
=
   ...
```

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

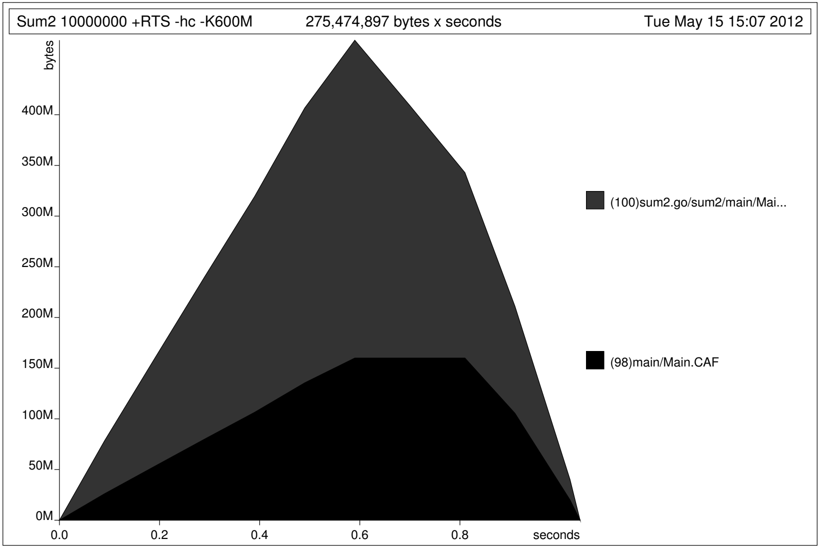
Well-Typed

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 . . .

Heap profile for `sum2`



The new problem

```
   sum2 [1, 2, 3, 4, ...]
=    { Definition of sum2 }
   sum2' 0 [1, 2, 3, 4, ...]
=    { Definition of sum2 }
   sum2' (0 + 1) [2, 3, 4, ...]
=    { Definition of sum2 }
   sum2' ((0 + 1) + 2) [3, 4, ...]
   ...
=
   sum2' (...((0 + 1) + 2)...) []
=    { Definition of sum2 }
   (...(0 + 1) + 2)...)
```

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

Well-Typed

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

```haskell
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).

```
(\ x -> ()) $! ⊥              ⇝*
seq (⊥, ⊥) ()                  ⇝*
snd $! (⊥, 1)                  ⇝*
(\ x -> ()) $! (\ x -> ⊥)      ⇝*
length $! map ⊥ [1, 2]         ⇝*
seq (⊥ + 1) ()                 ⇝*
seq (1 : ⊥) ()                 ⇝*
```

Well-Typed

Forcing quiz

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

```
(\ x -> ()) $! ⊥                ⤳*  ⊥
seq (⊥, ⊥) ()                   ⤳*
snd $! (⊥, 1)                   ⤳*
(\ x -> ()) $! (\ x -> ⊥)       ⤳*
length $! map ⊥ [1, 2]          ⤳*
seq (⊥ + 1) ()                  ⤳*
seq (1 : ⊥) ()                  ⤳*
```

Forcing quiz

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

```
(\ x -> ()) $! ⊥              ⇝*  ⊥
seq (⊥, ⊥) ()                 ⇝*  ()
snd $! (⊥, 1)                 ⇝*
(\ x -> ()) $! (\ x -> ⊥)     ⇝*
length $! map ⊥ [1, 2]        ⇝*
seq (⊥ + 1) ()                ⇝*
seq (1 : ⊥) ()                ⇝*
```

Forcing quiz

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

```
(\ x -> ()) $! ⊥                ⇝*   ⊥
seq (⊥, ⊥) ()                   ⇝*   ()
snd $! (⊥, 1)                   ⇝*   1
(\ x -> ()) $! (\ x -> ⊥)       ⇝*
length $! map ⊥ [1, 2]          ⇝*
seq (⊥ + 1) ()                  ⇝*
seq (1 : ⊥) ()                  ⇝*
```

Forcing quiz

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

```
(\ x -> ()) $! ⊥              ⇝*  ⊥
seq (⊥, ⊥) ()                 ⇝*  ()
snd $! (⊥, 1)                 ⇝*  1
(\ x -> ()) $! (\ x -> ⊥)     ⇝*  ()
length $! map ⊥ [1, 2]        ⇝*
seq (⊥ + 1) ()                ⇝*
seq (1 : ⊥) ()                ⇝*
```

Well-Typed

Forcing quiz

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

```
(\ x -> ()) $! ⊥               ⇝*   ⊥
seq (⊥, ⊥) ()                  ⇝*   ()
snd $! (⊥, 1)                  ⇝*   1
(\ x -> ()) $! (\ x -> ⊥)      ⇝*   ()
length $! map ⊥ [1, 2]         ⇝*   2
seq (⊥ + 1) ()                 ⇝*
seq (1 : ⊥) ()                 ⇝*
```

Forcing quiz

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

```
(\ x -> ()) $! ⊥                ⇝*  ⊥
seq (⊥, ⊥) ()                   ⇝*  ()
snd $! (⊥, 1)                   ⇝*  1
(\ x -> ()) $! (\ x -> ⊥)       ⇝*  ()
length $! map ⊥ [1, 2]          ⇝*  2
seq (⊥ + 1) ()                  ⇝*  ⊥
seq (1 : ⊥) ()                  ⇝*
```

Forcing quiz

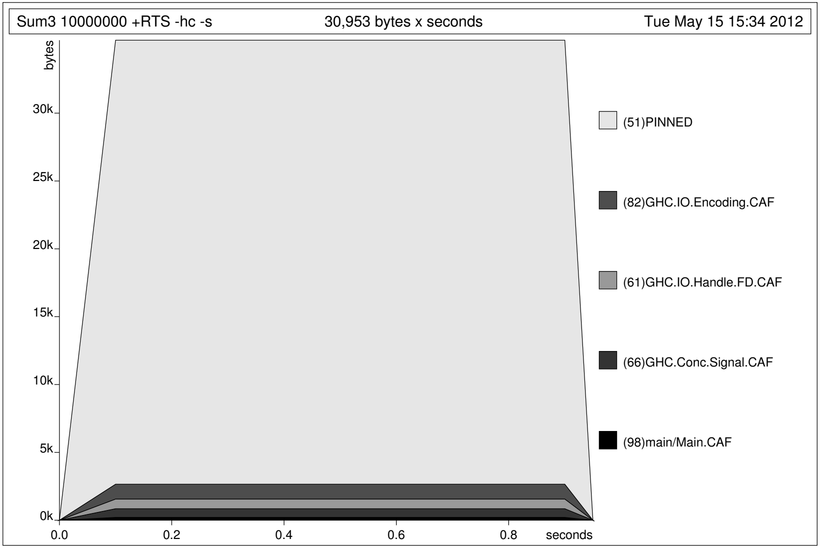
The function `seq` only evaluates to WHNF (i.e., a lambda abstraction, literal or constructor application).

```
(\ x -> ()) $! ⊥                ⇝*  ⊥
seq (⊥, ⊥) ()                   ⇝*  ()
snd $! (⊥, 1)                   ⇝*  1
(\ x -> ()) $! (\ x -> ⊥)       ⇝*  ()
length $! map ⊥ [1, 2]          ⇝*  2
seq (⊥ + 1) ()                  ⇝*  ⊥
seq (1 : ⊥) ()                  ⇝*  ()
```

Using seq to force the addition

```haskell
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:

```
sum1 = foldr  (+) 0
sum2 = foldl  (+) 0
sum3 = 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!

Strictness analysis

Back to our example

This is the tail-recursive version of `sum` again:

```haskell
sum2 :: [Int] -> Int
sum2 xs = go 0 xs
  where
    go acc []       = acc
    go acc (x : xs) = go (acc + x) xs
```

Let’s perform strictness analysis!

Looking at the Core code

You can spit out GHC Core after optimization by saying `-ddump-simpl` – reading the result requires some experience, but can show what optimizations GHC performs.

The result of `sum2` without optimization:

```
SumStrict.sum :: [GHC.Types.Int] -> GHC.Types.Int
[GblId, Arity=1]
SumStrict.sum =
  \ (xs_aan :: [GHC.Types.Int]) ->
    letrec {
      go_aao [Occ=LoopBreaker]
        :: GHC.Types.Int -> [GHC.Types.Int] -> GHC.Types.Int
      [LclId, Arity=2]
      go_aao =
        \ (acc_aap :: GHC.Types.Int) (ds_daU :: [GHC.Types.Int]) ->
          case ds_daU of _ {
            [] -> acc_aap;
            : x_aar xs1_aas ->
              go_aao
                (GHC.Num.+ @ GHC.Types.Int GHC.Num.$fNumInt acc_aap x_aar) xs1_aas
          }; } in
    go_aao (GHC.Types.I# 0) xs_aan
```

Well-Typed

Looking at the Core code (contd.)

Some observations about GHC Core:

- all names are fully qualified;
- symbolic names are escaped;
- nested pattern matches are compiled to simple pattern matches;
- everything is type-annotated;
- literals are unboxed, boxing is explicit;
- polymorphic functions have explicit type arguments (`@GHC.Types.Int`);
- overloaded functions are passed dictionaries (`GHC.Num.$fNumInt`).

Well-Typed

Looking at the Core code (contd.)

Excerpts from the optimized version:

```
SumStrict.sum :: [GHC.Types.Int] -> GHC.Types.Int
SumStrict.sum =
  \ (xs_aan :: [GHC.Types.Int]) ->
    case SumStrict.$wgo 0 xs_aan of ww_sbN { __DEFAULT ->
    GHC.Types.I# ww_sbN
    }
```

Looking at the Core code (contd.)

Excerpts from the optimized version:

```
SumStrict.sum :: [GHC.Types.Int] -> GHC.Types.Int
SumStrict.sum =
  \ (xs_aan :: [GHC.Types.Int]) ->
    case SumStrict.$wgo 0 xs_aan of ww_sbN { __DEFAULT ->
    GHC.Types.I# ww_sbN
    }
```

```
SumStrict.$wgo [Occ=LoopBreaker]
  :: GHC.Prim.Int# -> [GHC.Types.Int] -> GHC.Prim.Int#
SumStrict.$wgo =
  \ (ww_sbI :: GHC.Prim.Int#) (w_sbK :: [GHC.Types.Int]) ->
    case w_sbK of _ {
      [] -> ww_sbI;
      : x_aar xs_aas ->
        case x_aar of _ { GHC.Types.I# y_abo ->
        SumStrict.$wgo (GHC.Prim.+# ww_sbI y_abo) xs_aas
        }
    }
```

Well-Typed

Looking at the Core code (contd.)

Excerpts from the optimized version:

```
SumStrict.sum :: [GHC.Types.Int] -> GHC.Types.Int
SumStrict.sum =
  \ (xs_aan :: [GHC.Types.Int]) ->
    case SumStrict.$wgo 0 xs_aan of ww_sbN { __DEFAULT ->
    GHC.Types.I# ww_sbN
    }
```

```
SumStrict.$wgo [Occ=LoopBreaker]
  :: GHC.Prim.Int# -> [GHC.Types.Int] -> GHC.Prim.Int#
SumStrict.$wgo =
  \ (ww_sbI :: GHC.Prim.Int#) (w_sbK :: [GHC.Types.Int]) ->
    case w_sbK of _ {
      [] -> ww_sbI;
      : x_aar xs_aas ->
        case x_aar of _ { GHC.Types.I# y_abo ->
        SumStrict.$wgo (GHC.Prim.+# ww_sbI y_abo) xs_aas
        }
    }
```

Due to strictness analysis, GHC can actually unbox the `Int` accumulator.

More on controlling strictness

The type of `seq`

Question: Why is the type of `seq`

```
seq :: a -> b -> b
```

and not, for example,

```
seq' :: a -> a
seq' x = seq x x  -- example definition
```

?

The type of `seq`

Question: Why is the type of `seq`

```
seq :: a -> b -> b
```

and not, for example,

```
seq' :: a -> a
seq' x = seq x x  -- example definition
```

?

Remember: Everything is demand driven!

In `seq x y`, the evaluation of `x` is tied to the demand for `y`.

In `seq x x`, the evaluation of `x` is tied to the demand for `x` – which it is anyway.

Bang patterns

Instead of using `seq` or `($!)` , we can also make the pattern match evaluate more than it normally would . . .

Bang patterns

```haskell
sum4 xs = go 0 xs
  where
    go !acc []       = acc
    go !acc (x : xs) = go (acc + x) xs
```

. . . we can also use **bang patterns**.

A `!` in front of a variable pattern makes GHC evaluate the matched term to WHNF before continuing, even if this evaluation is not needed to make the decision for a particular case.

This requires the `BangPatterns` extension.

Irrefutable (lazy) patterns

A less known feature of Haskell is that it supports (without an extension) irrefutable patterns.

Consider this definition:

```haskell
unzip :: [(a, b)] -> ([a], [b])
unzip =
  foldr (\ (x, y) (xs, ys) -> (x : xs, y : ys)) ([], [])
```

Irrefutable (lazy) patterns

A less known feature of Haskell is that it supports (without an extension) irrefutable patterns.

Consider this definition:

```haskell
unzip :: [(a, b)] -> ([a], [b])
unzip =
  foldr (\ (x, y) (xs, ys) -> (x : xs, y : ys)) ([], [])
```

Now both of

```haskell
let zs = [1 ..]          in unzip (zip zs zs)
let zs = [1 .. 10000000] in unzip (zip zs zs)
```

result in a stack overflow. Why?

Unfolding the too strict `unzip`

```
  unzip [(1, 1)] (2, 2)
=   { Definition of unzip }
  foldr (\ (x, y) (xs, ys) -> (x : xs, y : ys)) ([], [])
    [(1, 1)] (2, 2)
=   { Definition of foldr }
  (\ (x, y) (xs, ys) -> (x : xs, y : ys)) (1, 1)
    (foldr...[(2, 2)])
=   { Matching the first pair }
  (\ (xs, ys) -> (1 : xs, 1 : ys)) (foldr...[(2, 2)])
```

At this point, we cannot reduce the function without first evaluation `foldr` further.

Introducing lazy patterns

A better definition is

```
unzip :: [(a, b)] -> ([a], [b])
unzip =
  foldr (\ (x, y) ~(xs, ys) -> (x : xs, y : ys)) ([], [])
```

which in this case is equivalent to

```
unzip :: [(a, b)] -> ([a], [b])
unzip =
  foldr (\ (x, y) xys -> (x : fst xys, y : snd xys)) ([], [])
```

Introducing lazy patterns

A better definition is

```
unzip :: [(a, b)] -> ([a], [b])
unzip =
  foldr (\ (x, y) ~(xs, ys) -> (x : xs, y : ys)) ([], [])
```

which in this case is equivalent to

```
unzip :: [(a, b)] -> ([a], [b])
unzip =
  foldr (\ (x, y) xys -> (x : fst xys, y : snd xys)) ([], [])
```

A `~` in front of a pattern will make it match always (hence **irrefutable**). Only if the components of the pattern are demanded, they will be extracted.

Lazy `let` and `where`, strict `case`

The outermost pattern in a `let` or `where` is lazy by default:

```
unzip :: [(a, b)] -> ([a], [b])
unzip []                = ([], [])
unzip ((x, y) : xys) = (x : xs, y : ys)
  where
    (xs, ys) = unzip xys  -- works, bad with !
```

Well-Typed

Lazy `let` and `where`, strict `case`

The outermost pattern in a `let` or `where` is lazy by default:

```
unzip :: [(a, b)] -> ([a], [b])
unzip []              = ([], [])
unzip ((x, y) : xys) = (x : xs, y : ys)
  where
    (xs, ys) = unzip xys  -- works, bad with !
```

On the other hand, patterns in a `case`, lambda or left hand side are strict:

```
unzip :: [(a, b)] -> ([a], [b])
unzip []              = ([], [])
unzip ((x, y) : xys) = case unzip xys of
  ~(xs, ys) -> (x : xs, y : ys)  -- requires ~ , bad without
```

Well-Typed

Strict data, lazy functions

Some advice:

- Don’t worry about strictness too much or too early.
- Do not just assume that making things stricter is always positive – understanding evaluation is the key, laziness in some places is as desirable as strictness is in others.
- Do not spread strictness annotations (bang patterns, `seq`) over a large amount of functions.
- Try to make certain pieces of **data** strict, either by using strict fields or by establishing strictness invariants in a small interface.
- Other functions using the interface will then automatically maintain the strictness invariants.
- Never forget that `seq` , bang patterns and strict fields only force WHNF.

Well-Typed

Normal form

When WHNF is not enough . . .

- For structured types, WHNF and NF are not the same.
- Sometimes – in particular in the context of parallel programming – we want data to be evaluated completely.
- Unlike `seq` , this functionality is not built into the language, but rather provided by a library `Control.DeepSeq` in the deepseq package.

`Control.DeepSeq` library

```haskell
class NFData a where
  rnf :: a -> ()
  rnf x = x `seq` ()   -- suitable default for flat types

deepseq :: NFData a => a -> b -> b
deepseq x y = rnf x `seq` y

($!!) :: NFData a => (a -> b) -> a -> b
f $!! x = x `deepseq` f x

force :: NFData a => a -> a
force x = x `deepseq` x
```

`Control.DeepSeq` library

```
class NFData a where
  rnf :: a -> ()
  rnf x = x `seq` ()   -- suitable default for flat types

deepseq :: NFData a => a -> b -> b
deepseq x y = rnf x `seq` y

($!!) :: NFData a => (a -> b) -> a -> b
f $!! x = x `deepseq` f x

force :: NFData a => a -> a
force x = x `deepseq` x
```

Note that what doesn't make sense for WHNF (a function like `force`) does make some sense for NF.

Well-Typed

Defining `NFData` instances

Example instance:

```
instance NFData a => NFData [a] where
  rnf []       = ()
  rnf (x : xs) = rnf x `seq` rnf xs
```

Note:

- The definition traverses the entire list before returning `()`.
- The call `rnf x` requires that the element type is an instance of `NFData`, too.
- We use `seq` in the second case to ensure that `rnf x` actually completes and produces the `()`.
- Calling `deepseq` has a cost. Don't force large structures unnecessarily often.
- `Control.DeepSeq` exports many basic instances.

Well-Typed

Thought experiments

- Including partially defined values, how many different elements of type `(Bool, Bool)` are there?
- And how many lists of type `[()]` and length at most `2` ?
- Can you write programs to distinguish all of these?

Lessons

- Both laziness and strictness can be desirable in certain situations.
- In Haskell, you are lazy by default, and have different options to make things more strict, selectively.
- Flat data often “wants” to be strict. For structured data, laziness is often desirable.
- Do not worry about strictness too early.
- Try to establish simple invariants on top of your datatypes – do not use `seq` or bang patterns in “random” places throughout your code.
- Use GHC’s runtime statistics or profiling in order to pinpoint how much space is used and where time is spent.