Space Leaks and Forcing Evaluation in Haskell

Neil Sculthorpe

Functional Programming Group
Information and Telecommunication Technology Center
University of Kansas
neil@ittc.ku.edu

EECS 776
Lawrence, Kansas
30th November 2012

Lazy Evaluation

- Lazy evaluation = Outermost Reduction + Sharing
- Benefits:
  - Avoids unnecassary computation
  - Infinite data structures
  - Control-flow structures can be defined as functions
  - Cyclic programming
- Disadvantages:
  - Performance hard to predict
  - Space leaks

Space Leaks

- A space leak is when a program uses up unnecessarily large amounts of memory.

Space Leaks

- A space leak is when a program uses up unnecessarily large amounts of memory.
- Lazy evaluation causes a space leak when a large expression remains unevaluated for a long time, even though it will eventually be evaluated.

Space Leaks

- A space leak is when a program uses up unnecessarily large amounts of memory.
- Lazy evaluation causes a space leak when a large expression remains unevaluated for a long time, even though it will eventually be evaluated.
- For example,

$$7 + 642 * 38581 - (47 + 15782 * 921) \\ + (\textbf{let } x = 4378 \textbf{ in } (x + 5) * ((x \text{ `}\mathit{div}\text{` } 5) - x)) \\ + 734 \uparrow 3 - 390326937$$

takes up more memory than

$$58$$

Some Terminology

- A Thunk is an unevaluated expression.
- An expression is in Normal Form if it contains only constructors (including literals).
- An expression is in Head Normal Form if it has a constructor in the outermost position (but it may contain thunks).

Some Terminology

- A Thunk is an unevaluated expression.
- An expression is in Normal Form if it contains only constructors (including literals).
- An expression is in Head Normal Form if it has a constructor in the outermost position (but it may contain thunks).

(Head normal form is slightly more complicated for function types, and Haskell actually uses a variant called weak head normal form, but I'm going to ignore this for the purposes of this lecture.)

Thunk, Normal or Head Normal?

1. $4 + 7$
2. 8
3. Just $(3\ `div`\ 0)$
4. Nothing
5. *head* $(1 : 2 : 3 : [])$
6. (Just 'A',True)
7. $(1 + 2) : (5 - 7) :$ *tail* $[]$

Thunk, Normal or Head Normal?

1. $4 + 7$ — Thunk
2. 8
3. Just $(3\ `div`\ 0)$
4. Nothing
5. *head* $(1 : 2 : 3 : [])$
6. (Just 'A',True)
7. $(1 + 2) : (5 - 7)$: *tail* []

Thunk, Normal or Head Normal?

1. $4 + 7$ — Thunk
2. 8 — Normal Form
3. Just $(3\ `div`\ 0)$
4. Nothing
5. *head* $(1 : 2 : 3 : [])$
6. (Just 'A',True)
7. $(1 + 2) : (5 - 7)$: *tail* []

Thunk, Normal or Head Normal?

1. $4 + 7$ — Thunk
2. 8 — Normal Form
3. Just $(3\ `div`\ 0)$ — Head Normal Form
4. Nothing
5. *head* $(1 : 2 : 3 : [])$
6. (Just 'A',True)
7. $(1 + 2) : (5 - 7) : \mathit{tail}\ []$

Thunk, Normal or Head Normal?

1. $4 + 7$ — Thunk
2. 8 — Normal Form
3. Just $(3\ `div`\ 0)$ — Head Normal Form
4. Nothing — Normal Form
5. *head* $(1 : 2 : 3 : [])$
6. (Just 'A',True)
7. $(1 + 2) : (5 - 7) : \mathit{tail}\ []$

Thunk, Normal or Head Normal?

1. $4 + 7$	Thunk
2. 8	Normal Form
3. Just $(3\ `div`\ 0)$	Head Normal Form
4. Nothing	Normal Form
5. *head* $(1 : 2 : 3 : [])$	Thunk
6. (Just 'A',True)	
7. $(1 + 2) : (5 - 7) :$ *tail* []	

Thunk, Normal or Head Normal?

1. $4 + 7$	Thunk
2. 8	Normal Form
3. Just $(3\ `div`\ 0)$	Head Normal Form
4. Nothing	Normal Form
5. *head* $(1 : 2 : 3 : [])$	Thunk
6. (Just 'A',True)	Normal Form
7. $(1 + 2) : (5 - 7) :$ *tail* $[]$	

Thunk, Normal or Head Normal?

1. $4 + 7$ — Thunk
2. 8 — Normal Form
3. Just $(3\ `div`\ 0)$ — Head Normal Form
4. Nothing — Normal Form
5. *head* $(1 : 2 : 3 : [])$ — Thunk
6. (Just 'A',True) — Normal Form
7. $(1 + 2) : (5 - 7)$: *tail* [] — Head Normal Form

Forcing Evaluation: Pattern Matching

- In Haskell, a thunk is evaluated to head normal form when you pattern match on it.

Forcing Evaluation: Pattern Matching

- In Haskell, a thunk is evaluated to head normal form when you pattern match on it.
- For example,

```
thunk = if x > y then 'h' : reverse "olle" else ""
foo = case thunk of
  c : cs → ...   -- c is the literal: 'h'
                 -- cs is the thunk: reverse "olle"
  []     → ...
```

Forcing Evaluation: *seq*

- Haskell provides a (semantically dodgy) function that evaluates thunks to head normal form:

$seq :: a \to b \to b$
$seq\ a\ b = \ldots$ -- evaluate a to head normal form, then return b

Forcing Evaluation: *seq*

- Haskell provides a (semantically dodgy) function that evaluates thunks to head normal form:

  $seq :: a \to b \to b$
  $seq\ a\ b = \ldots$ -- evaluate a to head normal form, then return b

- That is, you override Haskell's usual lazy evaluation.

Forcing Evaluation: *seq*

- Haskell provides a (semantically dodgy) function that evaluates thunks to head normal form:

  $seq :: a \to b \to b$
  $seq\ a\ b = \ldots$ -- evaluate a to head normal form, then return b

- That is, you override Haskell's usual lazy evaluation.
- Typically, this is only useful if a is used somewhere else in your program.

Forcing Evaluation: *seq*

- Haskell provides a (semantically dodgy) function that evaluates thunks to head normal form:

  $seq :: a \to b \to b$
  $seq\ a\ b = \ldots$ -- evaluate a to head normal form, then return b

- That is, you override Haskell's usual lazy evaluation.
- Typically, this is only useful if *a* is used somewhere else in your program.
- The idea is to use *seq* when:
  - the value of the thunk is definitely going to be used later
  - the thunk is using more memory than its head normal form would

Forcing Evaluation: *seq*

- Haskell provides a (semantically dodgy) function that evaluates thunks to head normal form:

  $seq :: a \to b \to b$
  $seq\ a\ b = \ldots$ -- evaluate a to head normal form, then return b

- That is, you override Haskell's usual lazy evaluation.
- Typically, this is only useful if *a* is used somewhere else in your program.
- The idea is to use *seq* when:
  - the value of the thunk is definitely going to be used later
  - the thunk is using more memory than its head normal form would
- Be careful, if used wrongly, *seq* will make your program slower!

Forcing Evaluation: Strictness Annotations on Data Fields

- Data type fields can be given strictness annotations.
- A strictness annotation is an ! prefixing the field type. E.g.

$$\textbf{data}\ \mathit{MyData} = \mathrm{MyCon}\ !\mathit{Bool}\ \mathit{Char}\ !(\mathit{Maybe}\ \mathit{Int})$$

Forcing Evaluation: Strictness Annotations on Data Fields

- Data type fields can be given strictness annotations.
- A strictness annotation is an ! prefixing the field type. E.g.

  data *MyData* = MyCon !*Bool Char* !(*Maybe Int*)

- A strictness annotation causes the field to be evaluated to head normal form whenever the data constructor is evaluated.

Forcing Evaluation: Strictness Annotations on Data Fields

- Data type fields can be given strictness annotations.
- A strictness annotation is an ! prefixing the field type. E.g.

  data *MyData* = MyCon !*Bool Char* !(*Maybe Int*)

- A strictness annotation causes the field to be evaluated to head normal form whenever the data constructor is evaluated.
- For example, when *MyData* is evaluated to head normal form:
  - the *Bool* is evaluated to True or False
  - the *Char* is not evaluated
  - the *Maybe* is evaluated to Just or Nothing
  - but the *Int* is not evaluated

Examples

See accompanying code...

Summary

- Lazy evaluation brings many benefits, but can cause space leaks.
- Haskell provides several mechanisms to preemptively evaluate thunks, which can be used to prevent space leaks.
- But if used wrongly, they will slow your program down by performing unnecessary computation.