Universiteit Utrecht

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

Applied Functional Programming

USCS 2009

Andres Löh

Department of Information and Computing Sciences
Utrecht University

August 24, 2009

C5. Lazy evaluation and profiling

C5.1 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: replacing the left hand side of a function definition by a corresponding right hand side (beta reduction from lambda calculus).

Reduction

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

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

Question

What if there are multiple redexes?

Multiple redexes

Many terms have multiple redexes.

How many redexes are in the following term?

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

Multiple redexes

Many terms have multiple redexes.

How many redexes are in the following term?

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

$$\begin{aligned}&\mathsf{id}\ (\mathsf{id}\ (\lambda \mathsf{z} \to \mathsf{id}\ \mathsf{z}))\\ &\quad(\mathsf{id}\ (\lambda \mathsf{z} \to \mathsf{id}\ \mathsf{z}))\\ &\qquad\qquad\quad \mathsf{id}\ \mathsf{z}\end{aligned}$$

Multiple redexes

Many terms have multiple redexes.

How many redexes are in the following term?

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

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

$$\begin{aligned} &\mathsf{id}\ (\mathsf{id}\ (\lambda \mathsf{z} \rightarrow \mathsf{id}\ \mathsf{z})) \\ &\quad (\mathsf{id}\ (\lambda \mathsf{z} \rightarrow \mathsf{id}\ \mathsf{z})) \\ &\qquad\qquad \mathsf{id}\ \mathsf{z} \end{aligned}$$

Multiple redexes

Many terms have multiple redexes.

How many redexes are in the following term?

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

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

$$\begin{aligned} &\mathsf{id}\ (\mathsf{id}\ (\lambda \mathsf{z} \to \mathsf{id}\ \mathsf{z})) \\ &\quad (\mathsf{id}\ (\lambda \mathsf{z} \to \mathsf{id}\ \mathsf{z})) \\ &\qquad\qquad \mathsf{id}\ \mathsf{z} \end{aligned}$$

$$\begin{aligned} &(\lambda \mathsf{x} \to \lambda \mathsf{y} \to \mathsf{x} * \mathsf{x})\ (1+2) \\ &\qquad\qquad\qquad\qquad (1+2) \\ &\qquad\qquad\qquad\qquad\qquad (3+4) \end{aligned}$$

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

Example

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

Example

$$\text{minimum xs} = \text{head (sort xs)}$$

$$\text{minimum } [4, 1, 3]$$

C5.2 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.

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.

> $\lambda f\ g\ x \rightarrow \text{combine}\ (f\ x)\ (g\ x)$

Church-Rosser

Theorem (Church-Rosser)

If a lambda 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 lambda 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 lambda term has at most one normal form.

Church-Rosser

Theorem (Church-Rosser)

If a lambda 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 lambda term has at most one normal form.

Theorem

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

Non-termination

- In lambda calculus, it is possible to encode recursion and therefore nonterminating (diverging) terms.
- A nonterminating value is usually written $\perp$.
- In Haskell, $\perp$ is available as undefined :: a.
- Error values are also often denoted $\perp$, and error :: String $\rightarrow$ a can be seen as an optimization of a diverging computation that prints an error message and breaks off program execution rather than actually diverging.
- In the presence of $\perp$, a function f is called strict if f $\perp \equiv \perp$. In other words, a strict function diverges if applied to a diverging argument.

Examples of lazy evaluation

1. $(\lambda \mathsf{x} \to \mathsf{x})\ \mathsf{True} \quad \rightsquigarrow^*$
2. $(\lambda \mathsf{x} \to \mathsf{x})\ \bot \quad \rightsquigarrow^*$
3. $(\lambda \mathsf{x} \to ())\ \bot \quad \rightsquigarrow^*$
4. $(\lambda \mathsf{x} \to \bot)\ () \quad \rightsquigarrow^*$
5. $(\lambda \mathsf{x}\ \mathsf{f} \to \mathsf{f}\ \mathsf{x})\ \bot \quad \rightsquigarrow^*$
6. $\bot_1\ \bot_2 \quad \rightsquigarrow^*$
7. $\mathsf{length}\ (\mathsf{map}\ \bot\ [1, 2]) \quad \rightsquigarrow^*$

Examples of lazy evaluation

1. $(\lambda x \to x)\ \mathsf{True} \quad \rightsquigarrow^* \quad \mathsf{True}$
2. $(\lambda x \to x)\ \bot \quad \rightsquigarrow^* \quad \bot$
3. $(\lambda x \to ())\ \bot \quad \rightsquigarrow^* \quad ()$
4. $(\lambda x \to \bot)\ () \quad \rightsquigarrow^* \quad \bot$
5. $(\lambda x\ f \to f\ x)\ \bot \quad \rightsquigarrow^* \quad \lambda f \to f\ \bot$
6. $\bot_1\ \bot_2 \quad \rightsquigarrow^* \quad \bot_1$
7. $\mathsf{length}\ (\mathsf{map}\ \bot\ [1, 2]) \quad \rightsquigarrow^* \quad 2$

Forcing evaluation

- Haskell has the following primitive function

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

Forcing evaluation

- Haskell has the following primitive function

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

- The call

```
seq x y
```

evaluates x before returning y.

Forcing evaluation

- Haskell has the following primitive function

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

- The call

```
seq x y
```

  evaluates x before returning y.

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

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

  Recall sharing!

Examples of forcing

Forcing only evaluates to weak head normal form (i.e., a lambda abstraction, literal or constructor application).

Examples of forcing

Forcing only evaluates to weak head normal form (i.e., a lambda abstraction, literal or constructor application).

1. $(\lambda \mathsf{x} \to ())\ \$!\ \bot \quad \rightsquigarrow^*$
2. $\mathsf{seq}\ (\bot_1, \bot_2)\ () \quad \rightsquigarrow^*$
3. $\mathsf{snd}\ \$!\ (\bot_1, \bot_2) \quad \rightsquigarrow^*$
4. $(\lambda \mathsf{x} \to ())\ \$!\ (\lambda \mathsf{x} \to \bot) \quad \rightsquigarrow^*$
5. $\bot_1\ \$!\ \bot_2 \quad \rightsquigarrow^*$
6. $\mathsf{length}\ \$!\ \mathsf{map}\ \bot\ [1, 2] \quad \rightsquigarrow^*$
7. $\mathsf{seq}\ (\bot_1 + \bot_2)\ () \quad \rightsquigarrow^*$
8. $\mathsf{seq}\ (\mathsf{foldr}\ \bot_1\ \bot_2)\ () \quad \rightsquigarrow^*$
9. $\mathsf{seq}\ (1 : \bot)\ () \quad \rightsquigarrow^*$

Examples of forcing

Forcing only evaluates to weak head normal form (i.e., a lambda abstraction, literal or constructor application).

1.	$(\lambda \mathsf{x} \to ())\ \$!\ \bot$	$\rightsquigarrow^*$	$\bot$
2.	$\mathsf{seq}\ (\bot_1, \bot_2)\ ()$	$\rightsquigarrow^*$	$()$
3.	$\mathsf{snd}\ \$!\ (\bot_1, \bot_2)$	$\rightsquigarrow^*$	$\bot_2$
4.	$(\lambda \mathsf{x} \to ())\ \$!\ (\lambda \mathsf{x} \to \bot)$	$\rightsquigarrow^*$	$()$
5.	$\bot_1\ \$!\ \bot_2$	$\rightsquigarrow^*$	$\bot_2$
6.	$\mathsf{length}\ \$!\ \mathsf{map}\ \bot\ [1, 2]$	$\rightsquigarrow^*$	2
7.	$\mathsf{seq}\ (\bot_1 + \bot_2)\ ()$	$\rightsquigarrow^*$	$\bot_1$
8.	$\mathsf{seq}\ (\mathsf{foldr}\ \bot_1\ \bot_2)\ ()$	$\rightsquigarrow^*$	$()$
9.	$\mathsf{seq}\ (1 : \bot)\ ()$	$\rightsquigarrow^*$	$()$

C5.3 Heap profiling

Space leaks

- Recall how Haskell data is represented at run-time.

Space leaks

- Recall how Haskell data is represented at run-time.
- A space leak is a data structure which grows bigger, or lives longer, than might be expected.
- Such unexpected memory use can cause a program to require more garbage collections or to run out of heap.
- More garbage collections cost a significant amount of time, so space performance is related to time performance!

Computing a large sum

$$
\begin{aligned}
&\mathsf{main} = \mathsf{print}\ (\mathsf{sum}_1\ [1\,..\,1000000]) \\
&\mathsf{sum}_1\ [\,] \qquad\quad = 0 \\
&\mathsf{sum}_1\ (\mathsf{x} : \mathsf{xs}) = \mathsf{x} + \mathsf{sum}_1\ \mathsf{xs}
\end{aligned}
$$

- A straight-forward definition: this is how sum is usually defined in an introductory course.
- What is the problem?

Computing a large sum

$$
\begin{aligned}
&\mathsf{main} = \mathsf{print}\ (\mathsf{sum}_1\ [1\,..\,1000000]) \\
&\mathsf{sum}_1\ [\,] \qquad\quad = 0 \\
&\mathsf{sum}_1\ (\mathsf{x} : \mathsf{xs}) = \mathsf{x} + \mathsf{sum}_1\ \mathsf{xs}
\end{aligned}
$$

- A straight-forward definition: this is how sum is usually defined in an introductory course.
- What is the problem?
- The correct answer (500000500000) is computed: this takes a significant amount of time, and more than 10MB of memory.
- The same is happening if sum_1 is defined using foldr.

Heap profiling

- Compile and run the program with profiling enabled:

```
$ ghc --make -prof -auto-all Sum1
$ ./Sum1 +RTS -K100M -p -hc
```

- The flag `+RTS` is special and supported by all GHC programs. Subsequent flags are not passed to the program, but to the GHC runtime system.
  - The flag `-K100M` increases the stack size.
  - The flag `-p` enables basic profiling.
  - The flag `-hc` enables heap profiling.
- Two files are produced by running the program: `Sum1.prof` and `Sum1.hp`.
- The heap profile data can be transformed into a PostScript file using the tool `hp2ps` which is included in the GHC distribution:

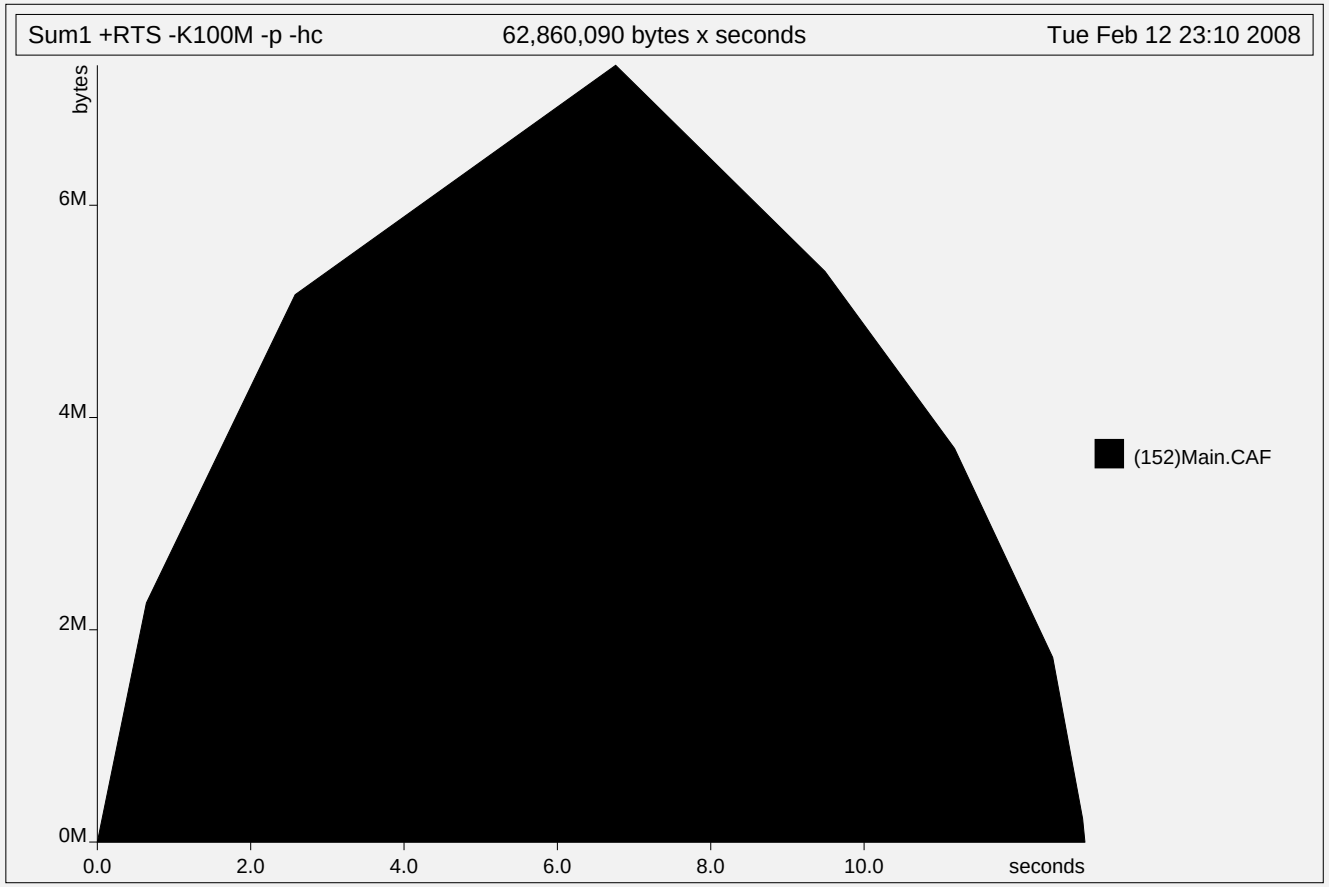
```
hp2ps Sum1.hp
```

The problem

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

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

Heap profile for sum_1



Attempting a tail-recursive version

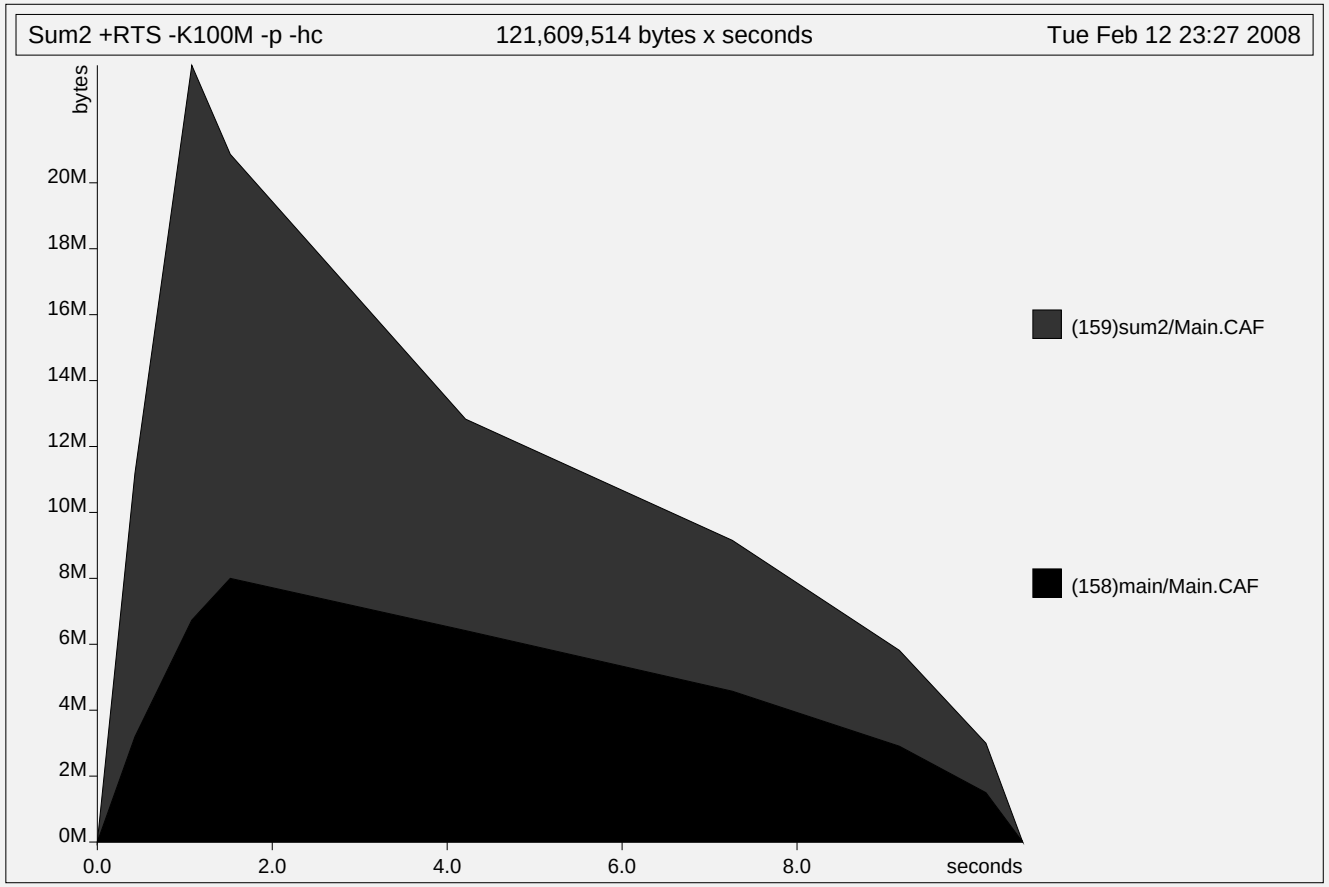
$$
\begin{aligned}
&\mathsf{main} = \mathsf{print}\ (\mathsf{sum}_2\ [1\,..\,1000000]) \\
&\mathsf{sum}_2\ \mathsf{xs} = \mathsf{sum}'_2\ 0\ \mathsf{xs} \\
&\quad \textbf{where} \\
&\qquad \mathsf{sum}'_2\ \mathsf{acc}\ [\,] \qquad\quad = \mathsf{acc} \\
&\qquad \mathsf{sum}'_2\ \mathsf{acc}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{sum}'_2\ (\mathsf{acc} + \mathsf{x})\ \mathsf{xs}
\end{aligned}
$$

- This is more or less equivalent to defining the function via foldl:

$$
\begin{aligned}
&\mathsf{foldl} :: (\mathsf{b} \to \mathsf{a} \to \mathsf{b}) \to \mathsf{b} \to [\mathsf{a}] \to \mathsf{b} \\
&\mathsf{foldl}\ \mathsf{op}\ \mathsf{e}\ [\,] \qquad\quad = \mathsf{e} \\
&\mathsf{foldl}\ \mathsf{op}\ \mathsf{e}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{foldl}\ \mathsf{op}\ (\mathsf{op}\ \mathsf{e}\ \mathsf{x})\ \mathsf{xs} \\
&\mathsf{sum}_2 = \mathsf{foldl}\ (+)\ 0
\end{aligned}
$$

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

Heap profile for sum_2



The new problem

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

We still build up the whole addition, but now in an accumulating argument!

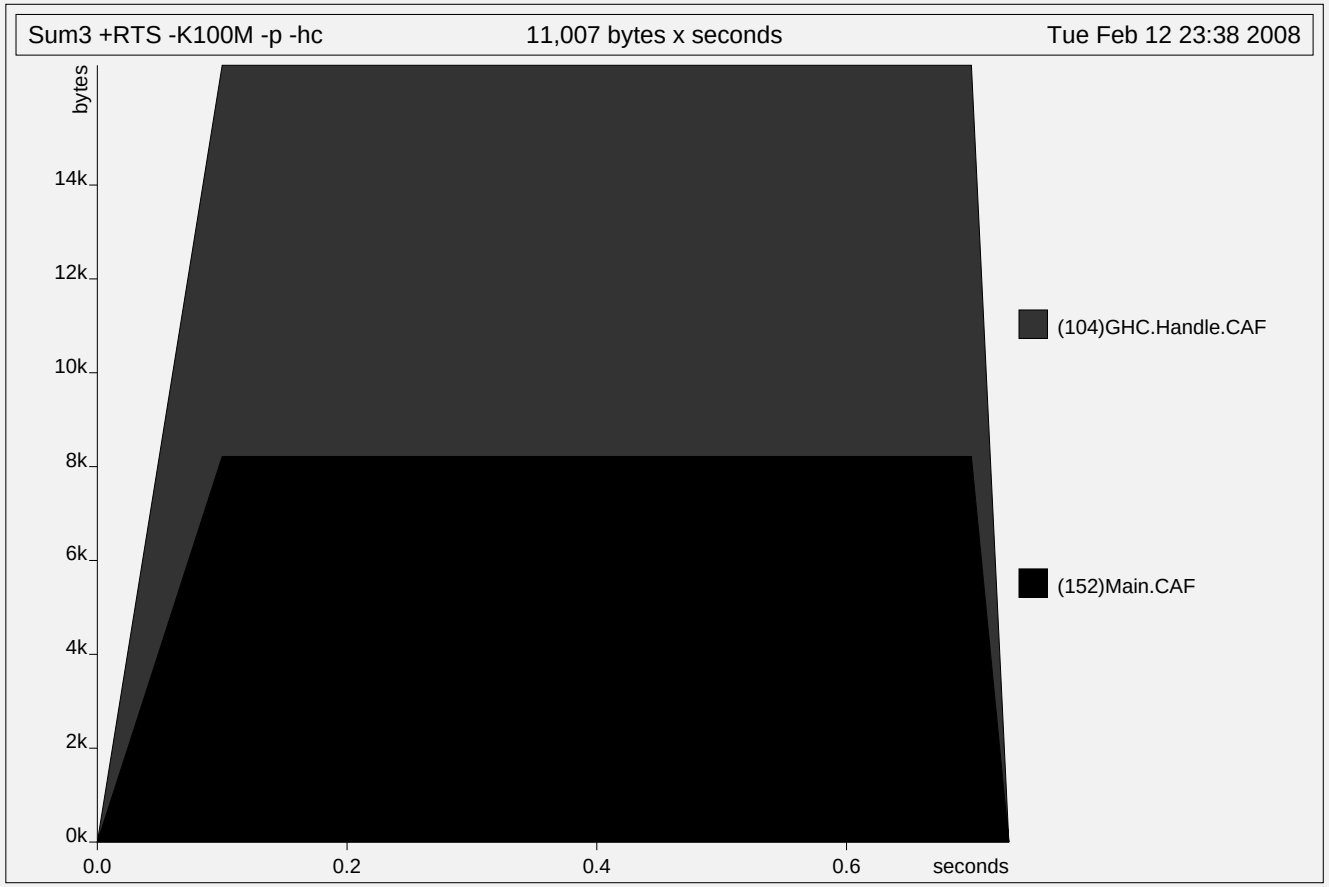
Forcing the addition

$$
\begin{aligned}
&\mathsf{main} = \mathsf{print}\ (\mathsf{sum}_3\ [1\,..\,1000000]) \\
&\mathsf{sum}_3\ \mathsf{xs} = \mathsf{sum}'_3\ 0\ \mathsf{xs} \\
&\quad \textbf{where} \\
&\qquad \mathsf{sum}'_3\ \mathsf{acc}\ [\,] \qquad\quad = \mathsf{acc} \\
&\qquad \mathsf{sum}'_3\ \mathsf{acc}\ (\mathsf{x} : \mathsf{xs}) = (\mathsf{sum}'_3\ \$!\ \mathsf{acc} + \mathsf{x})\ \mathsf{xs}
\end{aligned}
$$

- This is equivalent to using Data.List.foldl′, which is also used in the Prelude version of sum:

$$
\begin{aligned}
&\mathsf{foldl}' :: (\mathsf{b} \to \mathsf{a} \to \mathsf{b}) \to \mathsf{b} \to [\mathsf{a}] \to \mathsf{b} \\
&\mathsf{foldl}'\ \mathsf{op}\ \mathsf{e}\ [\,] \qquad\quad = \mathsf{e} \\
&\mathsf{foldl}'\ \mathsf{op}\ \mathsf{e}\ (\mathsf{x} : \mathsf{xs}) = (\mathsf{foldl}\ \mathsf{op}\ \$!\ \mathsf{op}\ \mathsf{e}\ \mathsf{x})\ \mathsf{xs} \\
&\mathsf{sum}_3 = \mathsf{foldl}'\ (+)\ 0
\end{aligned}
$$

Heap profile for sum_3



Another possibility: strict datatypes

```
data StrictInteger = SI !Integer
main = print (sum4 [1..1000000])
sum4 xs = sum'4 (SI 0) xs
  where
    sum'4 (SI acc) []       = acc
    sum'4 (SI acc) (x : xs) = sum'4 (SI (acc + x)) xs
```

- The 'bang' is a strictness annotation. It can be used on all fields of a datatype and makes the constructors strict in the respective arguments.
- Note that bangs in general only force evaluation to weak head normal form.

Question

Is using foldl/strictness always preferable?

Question

Is using foldl/strictness always preferable?

What about 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.