Tools & laziness

Advanced functional programming - Lecture 5

Wouter Swierstra

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Never use TABs

- Haskell uses layout to delimit language constructs.
- Haskell interprets TABs to have 8 spaces.
- Editors often display them with a different width.
- TABs lead to layout-related errors that are difficult to debug.
- Even worse: mixing TABs with spaces to indent a line.

Never use TABs

- Never use TABs.
- Configure your editor to expand TABs to spaces, and/or highlight TABs in source code.

Alignment

- Use alignment to highlight structure in the code!
- Do not use long lines.
- Do not indent by more than a few spaces.

```
map :: (a -> b) -> [a] -> [b]
map f []         =  []
map f (x : xs)   =  f x : map f xs
```

Identifier names

- Use informative names for functions.
- Use CamelCase for long names.
- Use short names for function arguments.
- Use similar naming schemes for arguments of similar types.

Spaces and parentheses

- Generally use exactly as many parentheses as are needed.
- Use extra parentheses in selected places to highlight grouping, particularly in expressions with many less known infix operators.
- Function application should always be denoted with a space.
- In most cases, infix operators should be surrounded by spaces.

Blank lines

- Use blank lines to separate top-level functions.
- Also use blank lines for long sequences of `let`-bindings or long **do**-blocks, in order to group logical units.

Avoid large functions

- Try to keep individual functions small.
- Introduce many functions for small tasks.
- Avoid local functions if they need not be local (why?).

Type signatures

- Always give type signatures for top-level functions.
- Give type signatures for more complicated local definitions, too.
- Use type synonyms.

```
checkTime :: Int -> Int -> Int -> Bool
```

Type signatures

- Always give type signatures for top-level functions.
- Give type signatures for more complicated local definitions, too.
- Use type synonyms.

```
checkTime :: Int -> Int -> Int -> Bool

checkTime :: Hours -> Minutes -> Seconds -> Bool

type Hours = Int
type Minutes = Int
type Seconds = Int
```

Comments

- Comment top-level functions.
- Also comment tricky code.
- Write useful comments, avoid redundant comments!
- Use Haddock.

Booleans

Keep in mind that Booleans are first-class values.

Negative examples:

```
f x | isSpace x == True = ...
```

```
if x then True else False
```

Use (data)types!

- Whenever possible, define your own datatypes.
- Use `Maybe` or user-defined types to capture failure, rather than `error` or default values.
- Use `Maybe` or user-defined types to capture optional arguments, rather than passing `undefined` or dummy values.
- Don't use integers for enumeration types.
- By using meaningful names for constructors and types, or by defining type synonyms, you can make code more self-documenting.

Use common library functions

- Don't reinvent the wheel. If you can use a `Prelude` function or a function from one of the basic libraries, then do not define it yourself.
- If a function is a simple instance of a higher-order function such as `map` or `foldr`, then use those functions (why?).

Pattern matching

- When defining functions via pattern matching, make sure you cover all cases.
- Try to use simple cases.
- Do not include unnecessary cases.
- Do not include unreachable cases.

Avoid partial functions

- Always try to define functions that are total on their domain, otherwise try to refine the domain type.
- Avoid using functions that are partial.

Negative example

```
if isJust x then 1 + fromJust x else 0
```

Use pattern matching!

Use `let` instead of repeating complicated code

Write

```
let x = foo bar baz in x + x * x
```

rather than

```
foo bar baz + foo bar baz * foo bar baz
```

Questions

- Is there a semantic difference between the two pieces of code?
- Could/should the compiler optimize from the second to the first version internally?

Let the types guide your programming

- Try to make your functions as generic as possible (why?).
- If you have to write a function of type `Foo -> Bar`, consider how you can destruct a `Foo` and how you can construct a `Bar`.
- When you tackle an unknown problem, think about its type first.

HLint

- A simple tool to improve your Haskell style.
- Developed by Neil Mitchell.
- Scans source code, provides suggestions.
- Makes use of generic programming (Uniplate).
- Suggests only correct transformations.
- New suggestions can be added, and some suggestions can be selectively disabled.
- Easy to install (via `cabal install`).

HLint

Example

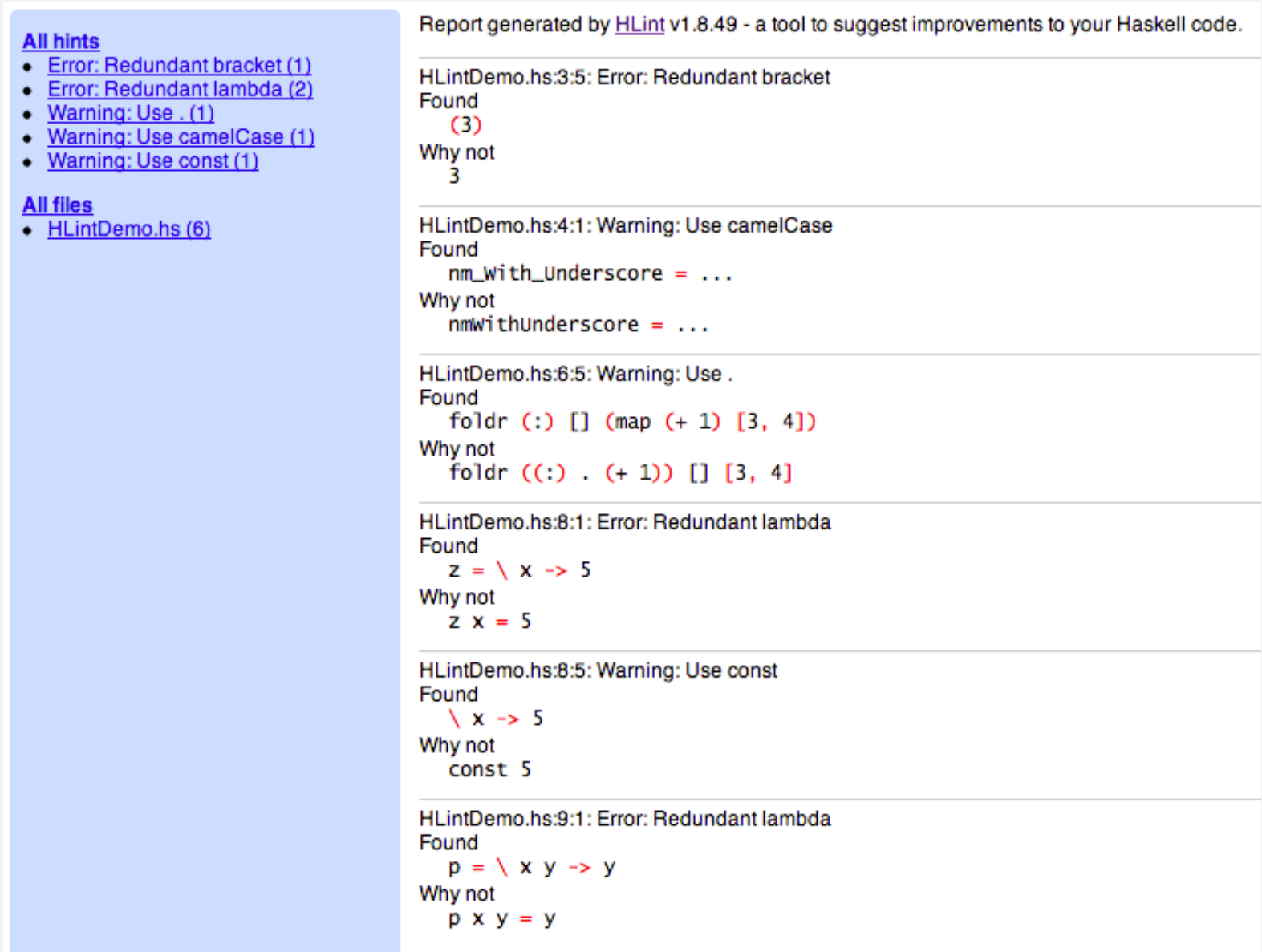
```
i = (3) + 4
nm_With_Underscore = i

y = foldr (:) [] (map (+1) [3,4])

z = \x -> 5
p = \x y -> y
```

- What does HLint complain about, why?
- Would you always want such complaints?

HLint



All hints
- Error: Redundant bracket (1)
- Error: Redundant lambda (2)
- Warning: Use . (1)
- Warning: Use camelCase (1)
- Warning: Use const (1)

All files
- HLintDemo.hs (6)

Report generated by HLint v1.8.49 - a tool to suggest improvements to your Haskell code.

HLintDemo.hs:3:5: Error: Redundant bracket
Found
```
(3)
```
Why not
```
3
```

HLintDemo.hs:4:1: Warning: Use camelCase
Found
```
nm_With_Underscore = ...
```
Why not
```
nmWithUnderscore = ...
```

HLintDemo.hs:6:5: Warning: Use .
Found
```
foldr (:) [] (map (+ 1) [3, 4])
```
Why not
```
foldr ((:) . (+ 1)) [] [3, 4]
```

HLintDemo.hs:8:1: Error: Redundant lambda
Found
```
z = \ x -> 5
```
Why not
```
z x = 5
```

HLintDemo.hs:8:5: Warning: Use const
Found
```
\ x -> 5
```
Why not
```
const 5
```

HLintDemo.hs:9:1: Error: Redundant lambda
Found
```
p = \ x y -> y
```
Why not
```
p x y = y
```


Figure 1:

Haddock

- Haddock is a documentation generator for Haskell (like JavaDoc, Doxygen, …)
- Parses annotated Haskell files.
- Most of GHC's language extensions are supported.
- API documentation (mainly).
- Program documentation (possible).
- HTML output.

Haddock annotations

```
--| This is (redundant) documentation for
-- function 'f'. The function 'f' is a badly
-- named substitute for the normal
-- /identity/ function 'id'.
f :: a -> a
f x = x
```

- A `--|` declaration affects the following top-level declaration.
- Single quotes as in `'f'` indicate the name of a Haskell function, and cause automatic hyperlinking. Referring to qualified names is also possible (even if the identifier is not normally in scope).
- Emphasis with forward slashes: `/identity/`.

More markup

Haddock supports several more forms of markup, for instance

- Sectioning to structure a module.
- Code blocks in documentation.
- References to whole modules.
- Itemized, enumerated, and definition lists.
- Hyperlinks.

Example

(Show on Hackage.)

Something about (in)efficiëncy

We have seen that Haskell programs:

- can be very short
- and sometimes very inefficiënt

Question:

How to find out where time is spent?

Something about (in)efficiëncy

We have seen that Haskell programs:

- can be very short
- and sometimes very inefficiënt

Question:

How to find out where time is spent?

Answer:

Use profiling

Example: segsinits

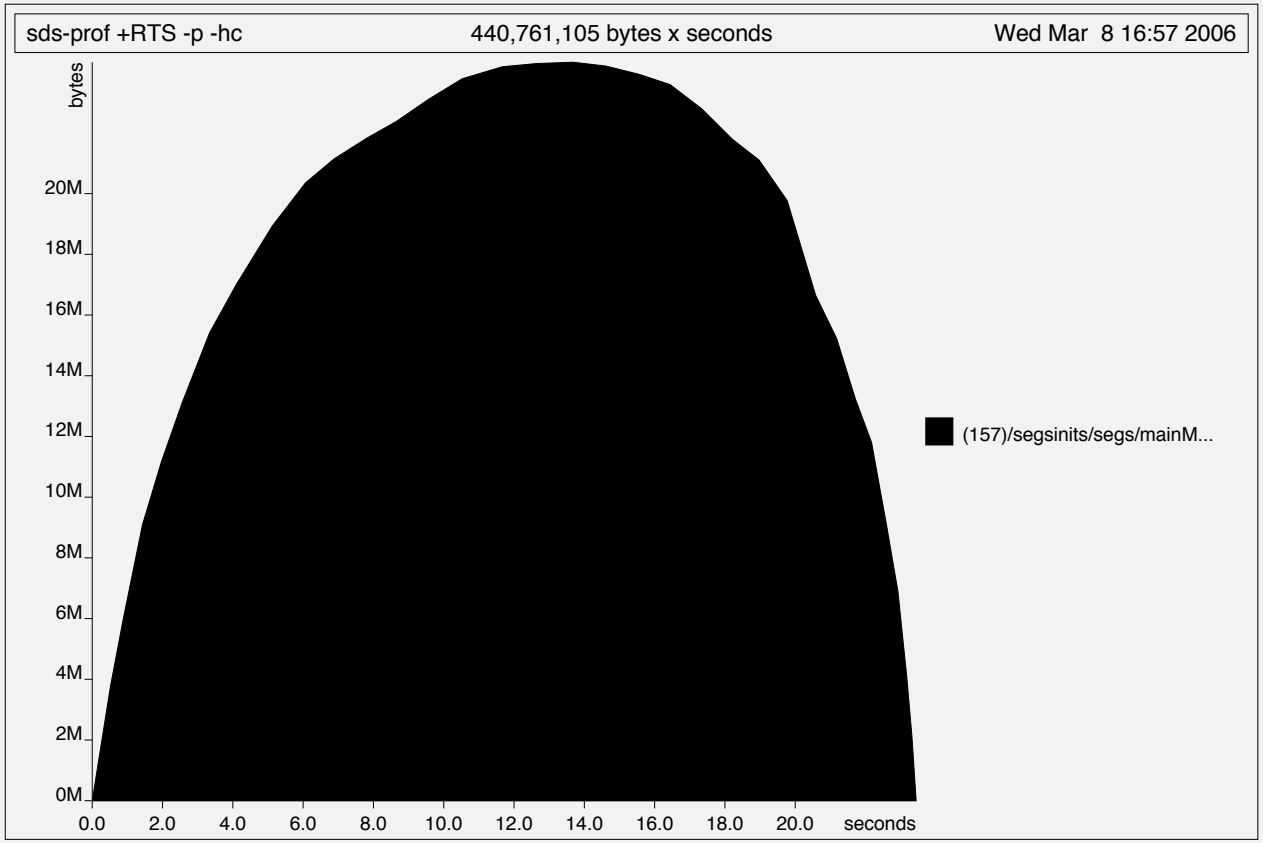
```
segsinits []       =  ([[]], [[]])
segsinits (x:xs)   =
  let (segsxs, initsxs)  = segsinits xs
      newinits           = map (x:) initsxs
  in (segsxs ++ newinits
     ,[]:newinits
     )
segs = fst . segsinits
```

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Heap profile

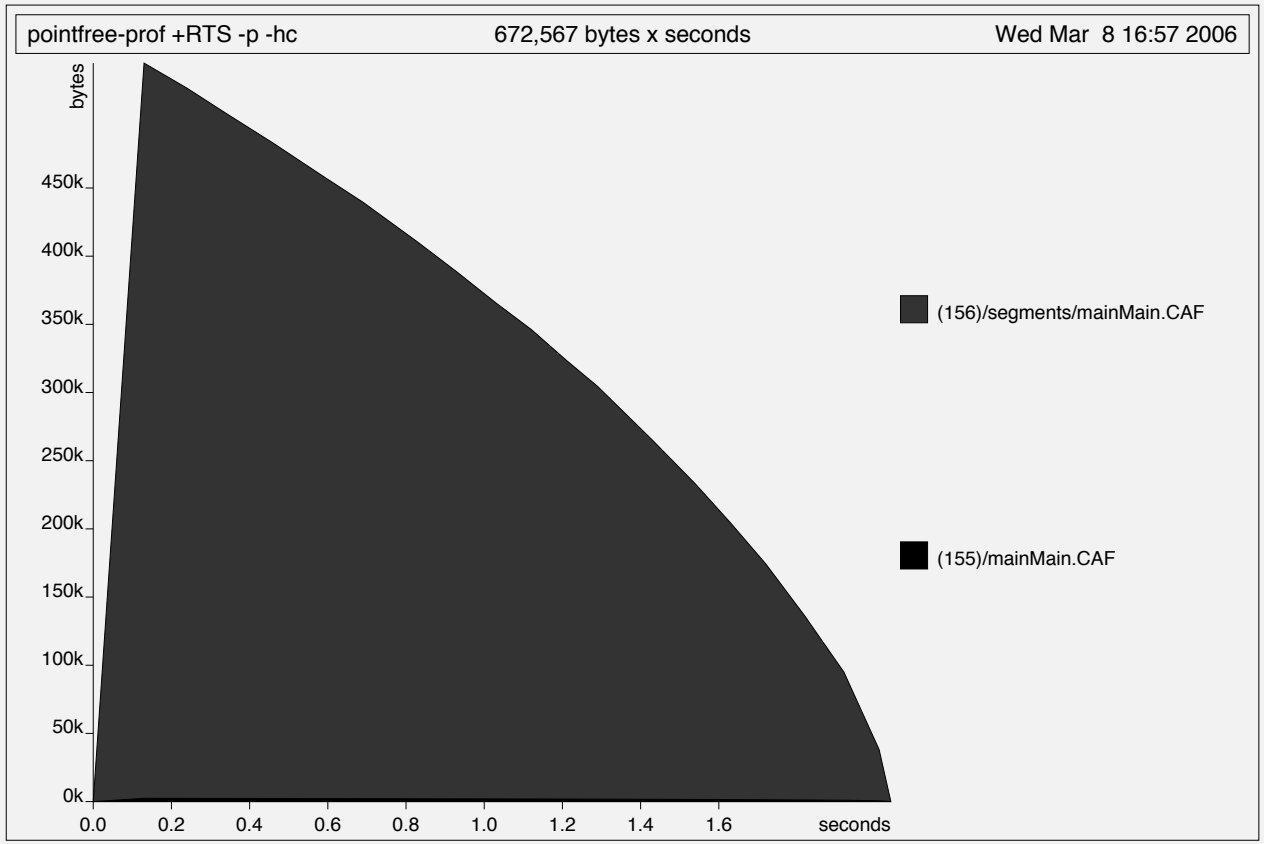
Using `segsinits`:



Example: `pointfree`

```
pointfree =
  let  p    = not . null
       next = filter p . map tail . filter p
  in concat . takeWhile p . iterate next . inits
```

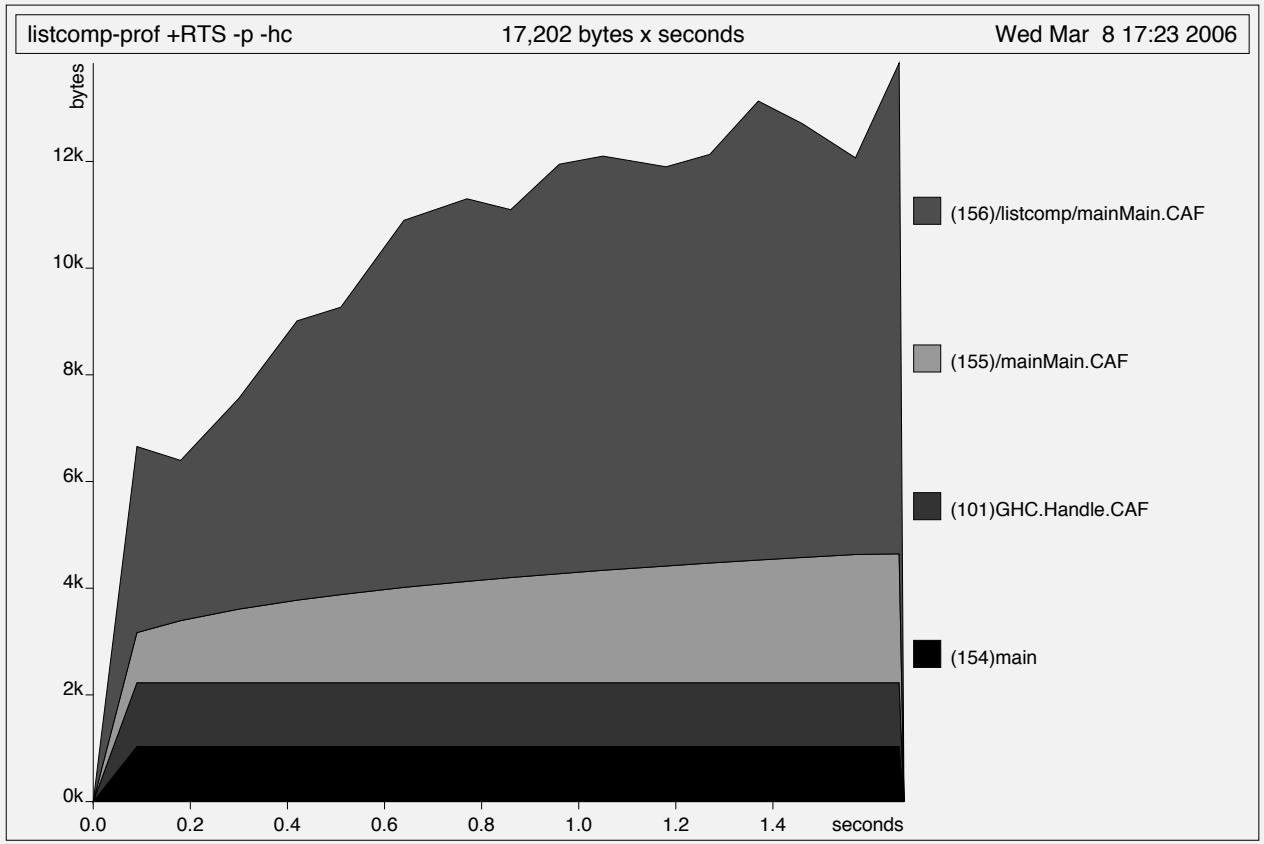
Using `pointfree`:



Example: `listcomp`

```
listcomp xs =
  [] : [ t | i <- inits xs
           , t <- tails i
           , not (null t) ]
main =
  print (length (concat
    (listcomp [1 :: Int .. 300])))
```

Using `listcomp`:



How to produce these?

```
prompt> ghc -prof -auto-all -o listcomp-prof
            -O2 Segments.hs
prompt> ./listcomp-prof +RTS -hc -p
4545100
prompt> hp2ps listcomp-prof.hp
```