Promoting Non-Strict Programming

Introducing StrictCheck

Olaf Chitil

University of Kent, UK

6th September 2006
Why Non-Strictness / Laziness Matters

John Hughes:

\[
evaluate = \text{max} \cdot \text{maximise'} \cdot \text{highfirst} \cdot \text{maptree static} \cdot \text{prune 8} \cdot \text{gametree}
\]

Intermediate data structures as glue enable modular program structure

- without space costs
- online: part of output already for part of input
Space leak: unexpectedly large space consumption

Claim: overly strict functions cause space leaks

evaluate = max . maximise’ . highfirst . maptree static . 
    prune 8 . gametree

Aim: StrictCheck, a tool for testing whether a Haskell function is too strict
An Example

unzip :: [(a,b)] -> ([a],[b])
unzip [] = ([],[]) 
unzip ((x,y):zs) = (x:xs,y:ys)
    where 
        (xs,ys) = unzip zs

unzip2 :: [(a,b)] -> ([a],[b])
unzip2 = foldr (\(x,y) (xs,ys) -> (x:xs,y:ys)) ([],[])
An Example

unzip :: [(a,b)] -> ([a],[b])
unzip [] = ([],[]) 
unzip ((x,y):zs) = (x:xs,y:ys) 
    where
        (xs,ys) = unzip zs 

unzip2 :: [(a,b)] -> ([a],[b])
unzip2 = foldr (\(x,y) (xs,ys) -> (x:xs,y:ys)) ([],[]) 

unzip ((0,0):⊥) = (0:⊥,0:⊥)  but  
unzip2 ((0,0):⊥) = ⊥
Distinguish

- function results for total arguments
- function results for partial arguments

The first do not uniquely determine the later.

Because of monotonicity and continuity:

\[ f v \subseteq \bigcup \{ f v' \mid v \subseteq v' \} \subseteq \bigcup \{ f v' \mid v \subseteq v', v' \text{ is total} \} \]

Function \( f \) least-strict iff

\[ f v = \bigcup \{ f v' \mid v \subseteq v', v' \text{ is total} \} \]
How to Test for Least-Strictness

\[
 f \text{ not least-strict if there exists partial argument } v \text{ such that } \\
 f \; v \sqsubseteq \bigsqcup \{ f \; v' | v \sqsubseteq v', v' \text{ is total} \}
\]

\[
 f \text{ probably not least-strict if } \\
 f \; v \sqsubseteq \bigsqcup \{ f \; v'_1, f \; v'_2, \ldots, f \; v'_n \}
\]

where \( v'_1, \ldots, v'_n \) are total with \( v \sqsubseteq v'_1, \ldots, v \sqsubseteq v'_n \).
Example test data:

\[
\bot, []\], [(0,0)], [(1,1)], [(0,0),(0,0)], ... \\
[\bot] [(0,0)], [(1,1)], [(0,0),(0,0)], ... \\
(0,0)\bot [(0,0)], [(0,0),(0,0)], [(0,0),(1,1)], ... \\
\ldots
\]

Systematically generate all arguments with one \(\bot\) up to given depth.

Use

- Scrap-your-boilerplate generics of Glasgow Haskell Compiler
- Chasing Bottoms library: (non-pure) `isBottom`, ...
Using StrictCheck

*Main> test1 5 (unzip2 :: [(Int,Int)] -> ([Int],[Int]))

Function seems not to be least strict.
Input(s): _
Current output: _
Proposed output: (_ , _)
Continue? y
Function seems not to be least strict.
Input(s): [(0, 0)]
Current output: _
Proposed output: ([0] , [0])

Detects spine-strictness of unzip2.
Using StrictCheck

*Main> test1 5 (True:)

Completed 36 test(s).
Function seems to be least strict.

Some functions are clearly least-strict.
Using StrictCheck

*Main> test2 10 (&&)

Function seems not to be least strict.
Input(s): (_|_, False)
Current output: _|_
Proposed output: False

Continue? y
Completed 4 test(s).

Proposes a function that is not sequential, hence undefinable in Haskell.
*Main> test2 5 ((++) :: [Int] -> [Int] -> [Int])

Function seems not to be least strict.
Input(s): (_|_, [0])
Current output: _|_
Proposed output: [_|__|_]
Continue? y
Function seems not to be least strict.
Input(s): (_|_, [0, 0])
Current output: _|_
Proposed output: [_|_, _|__|_]
Using StrictCheck

*Main> test1 5 (reverse :: [Int] -> [Int])

Function seems not to be least strict.
Input(s): [0_]_
Current output: _|
Proposed output: [__|_
  Continue?
Function seems not to be least strict.
Input(s): [0, 0_]_
Current output: _|
Proposed output: [__|_, _|__|_
  Continue?

Achievable, but inefficient.
Using StrictCheck

*Main> test1 5 (bfNum :: Tree Int -> Tree Int)

Function seems not to be least strict.
Input(s): T E 0 _|
Current output: _|
Proposed output: T E 1 _|
  Continue? y
Function seems not to be least strict.
Input(s): T E 0 (T E 0 _)
Current output: _|
Proposed output: T E 1 (T E 2 _)

That is the information we want.
proposes non-sequential functions (&&)
proposes undesirably inefficient functions (reverse)
abstract data types:
  - distinguishes equal elements of product types
    \( \bot = \text{Queue} \ \bot \ \bot = \text{Queue} \ \bot \ [\] = \text{Queue} \ [\] \ \bot \)
  - generates illegal elements
  - generated elements that are hard to read (internal representation)
cannot exclude a class of counter examples
Summary

StrictCheck

- tests whether a function is least-strict
- proposes less strict variant

*Main> test1 5 (bfNum :: Tree Int -> Tree Int)
Function seems not to be least strict.
Input(s): T E 0 _|
Current output: _|
Proposed output: T E 1 _|

To Do:

- solve problems
- apply to more examples