Promoting Non-Strict Programming
—
Introducing StrictCheck

Olaf Chitil

University of Kent, UK

6th September 2006

Why Non-Strictness / Laziness Matters

John Hughes:

```
evaluate = max . maximise' . highfirst . maptree static .
             prune 8 . gametree
```

Intermediate data structures as glue enable modular program structure

- without space costs
- online: part of output already for part of input

Strict vs. Non-Strict

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):⊥) = ⊥
```

Least Strictness

Distinguish

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

The first do not uniquely determine the later.

Because of monotonicity and continuity:

$$f\ v \ \sqsubseteq\ \bigsqcup\{f\ v' | v \sqsubseteq v'\} \ \sqsubseteq\ \bigsqcup\{f\ v' | v \sqsubseteq v', v' \text{ is total}\}$$

Function f least-strict iff

$$f\ v = \bigsqcup\{f\ v' | v \sqsubseteq v', v' \text{ is total}\}$$

How to Test for Least-Strictness

f not least-strict if there exists partial argument v such that

$$f\ v \ \sqsubset\ \bigsqcup\{f\ v' | v \sqsubseteq v', v' \text{ is total}\}$$

f probably not least-strict if

$$f\ v \ \sqsubset\ \bigsqcup\{f\ v'_1, f\ v'_2, \ldots, f\ v'_n\}$$

where $v'_1, \ldots, v'_n$ are total with $v \sqsubset v'_1, \ldots, v \sqsubset v'_n$.

Implementation

Example test data:

```
⊥           [], [(0,0)], [(1,1)], [(0,0),(0,0)], ...
[⊥]         [(0,0)], [(1,1)], [(0,0),(0,0)], ...
(0,0):⊥     [(0,0)], [(0,0),(0,0)], [(0,0),(1,1)], ...
...
```

Systematically generate all arguments with one ⊥ 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.

Using StrictCheck

```
*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: [_|_, _|__|_
```

Not sequential.

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.

Problems Observed in Practice

- proposes non-sequential functions (`&&`)
- proposes undesirably inefficient functions (`reverse`)
- abstract data types:
  - distinguishes equal elements of product types
    `⊥ = Queue ⊥ ⊥ = Queue ⊥ [] = Queue [] ⊥`
  - 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