## dup – Explicit un-sharing in Haskell

Haskell Implementors Workshop 2012 – Lightning Talk

### Table: Time and Space Performance

<table>
<thead>
<tr>
<th>Original</th>
<th>$\text{solveDup}$</th>
<th>$\text{rateDup}$</th>
<th>$\text{solveDeepDup}$</th>
<th>$\text{unit lifting}$</th>
<th>$\text{church encoding}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6.70</td>
<td>4</td>
<td>189</td>
<td>24.15</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>6.71</td>
<td>3</td>
<td>6.74</td>
<td>4</td>
<td>188</td>
</tr>
<tr>
<td>2</td>
<td>2.33</td>
<td>5</td>
<td>2.34</td>
<td>5</td>
<td>2.34</td>
</tr>
<tr>
<td>2</td>
<td>6.63</td>
<td>2</td>
<td>6.79</td>
<td>2</td>
<td>6.74</td>
</tr>
<tr>
<td>1</td>
<td>1.79</td>
<td>1</td>
<td>1.78</td>
<td>1</td>
<td>1.78</td>
</tr>
<tr>
<td>2</td>
<td>7.12</td>
<td>2</td>
<td>7.06</td>
<td>2</td>
<td>7.19</td>
</tr>
</tbody>
</table>

### Diagram

Figure 2. Time and space performance for $b=4$ and $d=4$

![Diagram](image)

Figure 3. The heap during original and $\text{dup}$'ed evaluation with $b=2$ and $d=1$

![Diagram](image)

Figure 4. Comparing $\text{solveDup}$ and $\text{solveDeepDup}$ applied to a partly evaluated tree with $b=2$ and $d=1$
We need to provide our programmers with better tools to

analyze

and

control

the space behaviour of their Haskell programs.
Sharing can cause space leaks

```haskell
let xs = [1..100000000]
in (last xs, length xs)
```
Sharing can cause space leaks

```haskell
let xs = [1..100000000]
in (last xs, length xs)
```
Sharing can cause space leaks

```haskell
let xs = [1..100000000]
in (last xs, length xs)
```
Sharing can cause space leaks

```haskell
let xs = [1..100000000]
in (last xs, length xs)
```
Sharing can cause space leaks

```haskell
let xs = [1..100000000] in (last xs, length xs)
```

The programmer might want to avoid to have the list shared.
Sharing can cause space leaks

```haskell
let xs = [1..100000000]
in (last xs, length xs)
```

```
1e8    T

\( (\cdot) \)

1    2    3    4    \cdot \cdot \cdot
```
Sharing can cause space leaks

```haskell
let xs = [1..100000000]
in (last xs, length xs)
```

```plaintext
(,
  1e8
  T
  (,
    1
    2
    3
    4
    (,
      ...
    ))

the programmer might want to avoid to have the list shared
```
Source transformations may help

```
let xs () = [1..100000000]
in (last $ xs (), length $ xs ())
```
Source transformations may help

```
let xs () = [1..100000000]
in (last $ xs (), length $ xs ())
```

T
F

works, but fragile – might be thwarted by compiler optimizations
Source transformations may help

```haskell
let xs () = [1..100000000]
in (last $ xs (), length $ xs ())
```

This works, but fragile – might be thwarted by compiler optimizations.
Source transformations may help

```
let xs () = [1..100000000]
in (last $ xs (), length $ xs ())
```

![Diagram](image)
Source transformations may help

```haskell
let xs () = [1..100000000] 
in (last $ xs (), length $ xs ())
```

```
(,) 
  T

F

1 2 3 4 T
```
Source transformations may help

```haskell
let xs () = [1..100000000]
in (last $ xs (), length $ xs ())
```

The expression works, but it is fragile—might be thwarted by compiler optimizations.
Source transformations may help

```
let xs () = [1..100000000]
in (last $ xs (), length $ xs ()
```

works, but fragile – might be thwarted by compiler optimizations
Allow the programmer to copy a thunk: `dup`

```haskell
let xs = [1..100000000]
  in (case dup xs of Box xs’ -> last xs’,
    case dup xs of Box xs’ -> length xs’)
```

The consumer, not the generator, controls sharing. No code restructuring.
Allow the programmer to copy a thunk: dup

```haskell
let xs = [1..100000000]
in (case dup xs of Box xs' → last xs',
    case dup xs of Box xs' → length xs')
```
Allow the programmer to copy a thunk: dup

\[
\begin{aligned}
\text{let } &\, xs = [1..100000000] \\
\text{in } &\, (\text{case } \text{dup } xs \text{ of } \Box xs' \rightarrow \text{last } xs', \newline
\text{case } \text{dup } xs \text{ of } \Box xs' \rightarrow \text{length } xs')
\end{aligned}
\]
Allow the programmer to copy a thunk: `dup`

```haskell
let xs = [1..100000000]
in (case dup xs of Box xs’ → last xs’,
  case dup xs of Box xs’ → length xs’)
```
Allow the programmer to copy a thunk: `dup`

```
let xs = [1..100000000]
in (case dup xs of Box xs' → last xs',
    case dup xs of Box xs' → length xs')
```
Allow the programmer to copy a thunk: dup

```
let xs = [1..100000000]
in (case dup xs of Box xs' -> last xs',
  case dup xs of Box xs' -> length xs')
```
Allow the programmer to copy a thunk: dup

```
let xs = [1..100000000]
in (case dup xs of Box xs’ -> last xs’,
    case dup xs of Box xs’ -> length xs’)
```

the consumer, not the generator, controls sharing. no code restructuring.
The sledgehammer: deepDup

morally, deepDup x copies the whole heap reachable by x
The sledgehammer: deepDup

morally, deepDup x copies the whole heap reachable by x
The sledgehammer: deepDup

morally, deepDup x copies the whole heap reachable by x
The sledgehammer: deepDup

morally, deepDup x
copies the whole
heap reachable by x
The sledgehammer: deepDup

morally, deepDup x copies the whole heap reachable by x
The sledgehammer: deepDup

morally, deepDup x copies the whole heap reachable by x
The sledgehammer: deepDup

morally, deepDup x
copies the whole
heap reachable by x
The sledgehammer: deepDup

morally, deepDup x copies the whole heap reachable by x
The sledgehammer: deepDup

morally, deepDup x copies the whole heap reachable by x

really, deepDup x copies the whole heap reachable by x lazily
Comes with proofs included.

\[
\begin{align*}
\Gamma, x & \mapsto e, x' \mapsto \hat{e} : x' \Downarrow \Delta : z \quad & x' \text{ fresh} \\
\Gamma, x & \mapsto e : \text{dup} x \Downarrow \Delta : z
\end{align*}
\]

\[
\begin{align*}
x' & \mapsto \hat{e}[y'_1/y_1, \ldots, y'_n/y_n], \\
\Gamma, x & \mapsto e, y'_1 \mapsto \text{deepDup} y_1, \ldots, y'_n \mapsto \text{deepDup} y_n : x' \Downarrow \Delta : z \\
\text{ufv}(e) & = \{y_1, \ldots, y_n\} \\
x', y'_1, \ldots, y_n & \text{ fresh}
\end{align*}
\]

\[
\begin{align*}
\Gamma, x & \mapsto e : \text{deepDup} x \Downarrow \Delta : z
\end{align*}
\]

(based on Launchbury’s „A natural semantics for lazy evaluation“)
Where to read more

See


for

- more motivation,
- benchmarked comparison with other approaches to avoid sharing,
- semantics and proofs,
- details on the implementation and
- a description of current shortcomings.

See

http://darcs.nomeata.de/ghc-dup

for

- the code.
A related, younger idea

```haskell
import GHC.Prim (noupdate)

let xs = noupdate [1..100000000]
in (last xs, length xs)
```

For a thunk wrapped in

\[ \text{noupdate} :: a \rightarrow a, \]

no blackhole and no update frame is created

\[ \implies \text{sharing is effectively prevented}. \]

(Ask me for my ghc branch.)
Also nice: ghc-vis

Demonstration

see
http://hackage.haskell.org/package/ghc-vis
and
http://felsin9.de/nnis/ghc-vis/