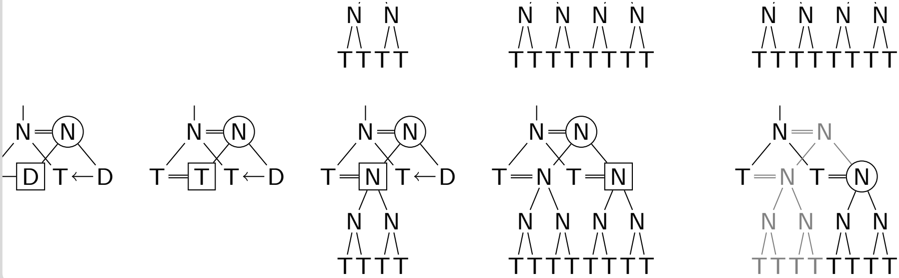
Karlsruhe Institute of Technology

dup – Explicit un-sharing in Haskell

Haskell Implementors Workshop 2012 – Lightning Talk

PROGRAMMING PARADIGMS GROUP



KIT – University of the State of Baden-Wuerttemberg and
National Research Center of the Helmholtz Association

www.kit.edu

My motivation

We need to provide our programmers with better tools to

analyze

and

control

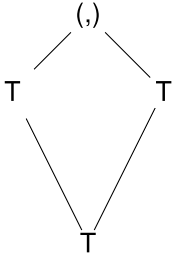
the space behaviour of their Haskell programs.

Sharing can cause space leaks

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

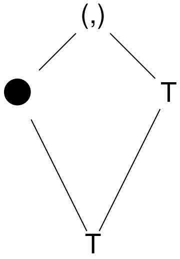
Sharing can cause space leaks

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



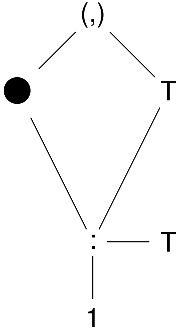
Sharing can cause space leaks

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



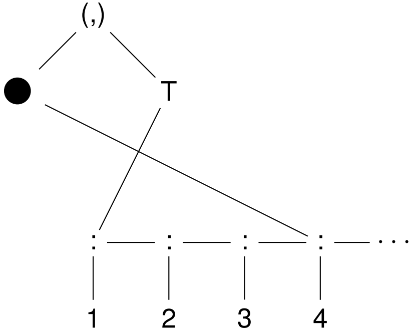
Sharing can cause space leaks

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



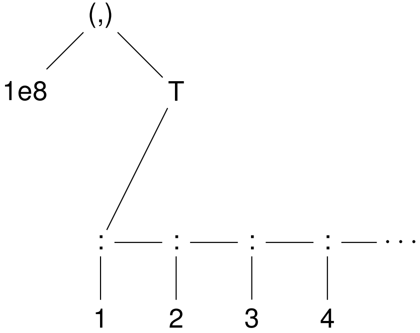
Sharing can cause space leaks

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



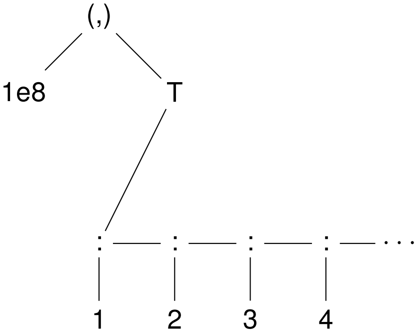
Sharing can cause space leaks

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



Sharing can cause space leaks

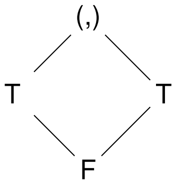
```
let xs = [1..100000000]
in (last xs, length xs)
```



the programmer might want to avoid to have the list shared

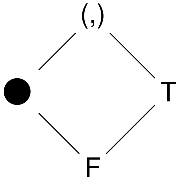
Source transformations may help

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



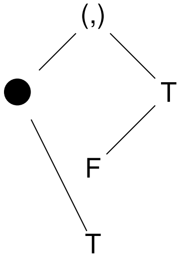
Source transformations may help

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



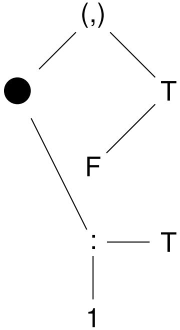
Source transformations may help

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



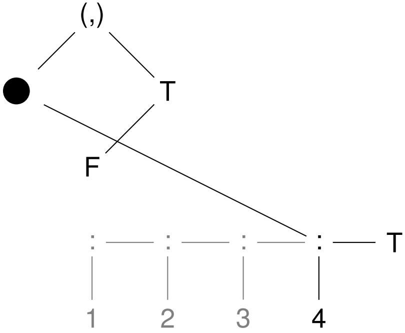
Source transformations may help

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



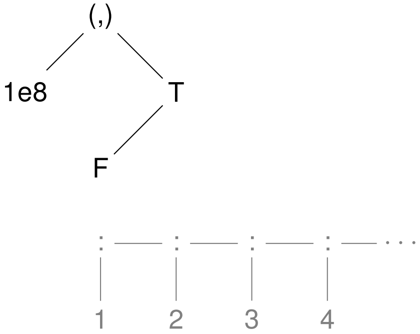
Source transformations may help

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



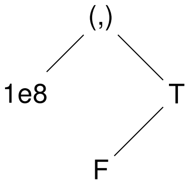
Source transformations may help

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



Source transformations may help

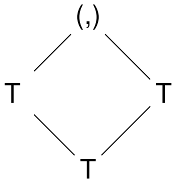
```
let xs () = [1..100000000]
in (last $ xs (), length $ xs ())
```



works, but fragile – might be thwarted by compiler optimizations

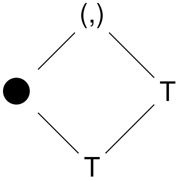
Allow the programmer to copy a thunk: dup

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



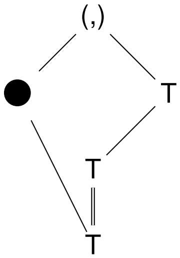
Allow the programmer to copy a thunk: dup

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



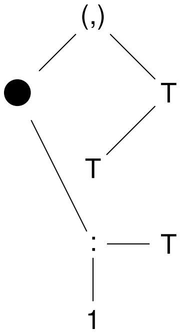
Allow the programmer to copy a thunk: dup

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



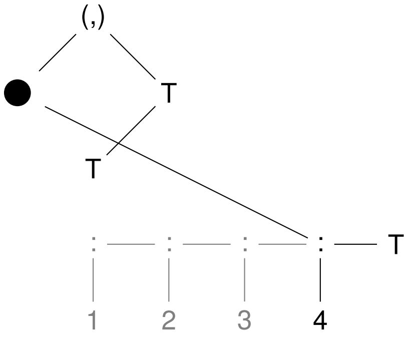
Allow the programmer to copy a thunk: dup

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



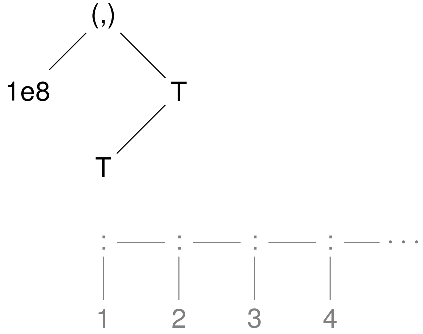
Allow the programmer to copy a thunk: dup

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



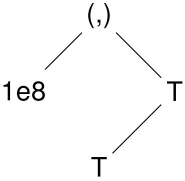
Allow the programmer to copy a thunk: dup

```
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

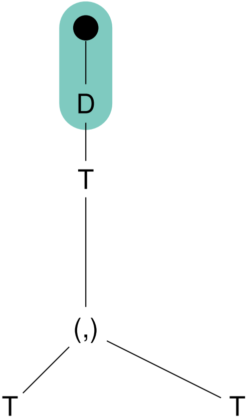
```
let xs = [1..100000000]
in (case dup xs of Box xs' −> last xs',
    case dup xs of Box xs' −> length xs')
```



the consumer, not the generator, controls sharing. no code restructuring.

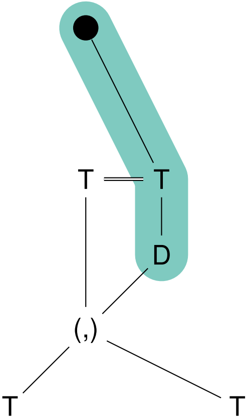
The sledgehammer: deepDup

morally, deepDup x copies the whole heap reachable by x



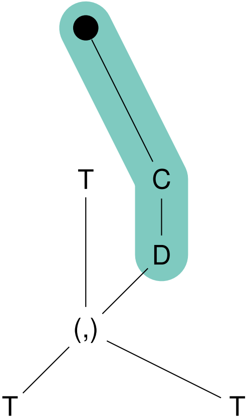
The sledgehammer: deepDup

morally, deepDup x copies the whole heap reachable by x



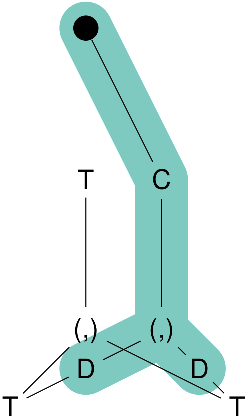
The sledgehammer: deepDup

morally, deepDup x copies the whole heap reachable by x



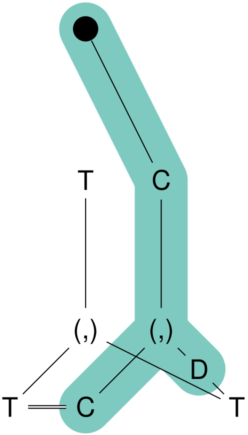
The sledgehammer: deepDup

morally, deepDup x
copies the whole
heap reachable by x



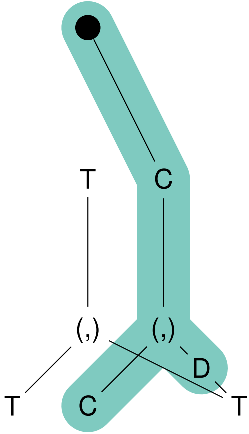
The sledgehammer: deepDup

morally, deepDup x
copies the whole
heap reachable by x



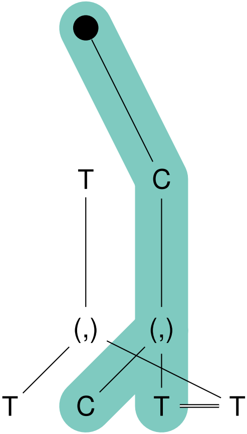
The sledgehammer: deepDup

morally, deepDup x copies the whole heap reachable by x



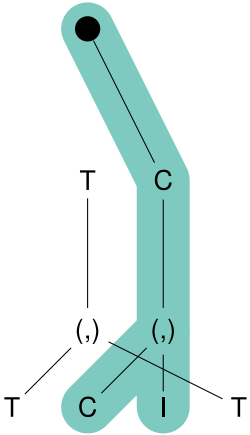
The sledgehammer: deepDup

morally, deepDup x copies the whole heap reachable by x



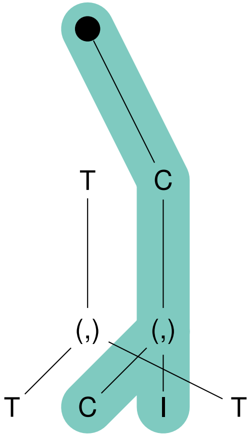
The sledgehammer: deepDup

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.

$$\frac{\Gamma, x \mapsto e, x' \mapsto \hat{e} : x' \Downarrow \Delta : z \qquad x' \text{ fresh}}{\Gamma, x \mapsto e : \mathrm{dup}\, x \Downarrow \Delta : z}\,\text{Dup}$$

$$\frac{\begin{array}{c} x' \mapsto \hat{e}[y_1'/y_1, \ldots, y_n'/y_n], \\ \Gamma, x \mapsto e,\ \ y_1' \mapsto \mathrm{deepDup}\, y_1, \ldots, y_n' \mapsto \mathrm{deepDup}\, y_n \ \ : x' \Downarrow \Delta : z \\ \mathrm{ufv}(e) = \{y_1, \ldots, y_n\} \qquad x',\ y_1', \ldots, y_n \text{ fresh} \end{array}}{\Gamma, x \mapsto e : \mathrm{deepDup}\, x \Downarrow \Delta : z}\,\text{Deep}$$

(based on Launchbury’s „A natural seantics for lazy evaluation“)

Where to read more

See

`http://arxiv.org/abs/1207.2017`

for

- more motiviation,
- 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

```
import GHC.Prim (noupdate)

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

For a thunk wrapped in

noupdate :: a −> a,

no blackhole and no update frame is created

$\Longrightarrow$ 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/`