Catching space leaks at compile-time using th-deepstrict

Teo Camarasu

CircuitHub

6^{th} of June 2025

Haskell Implementer's Workshop 2025

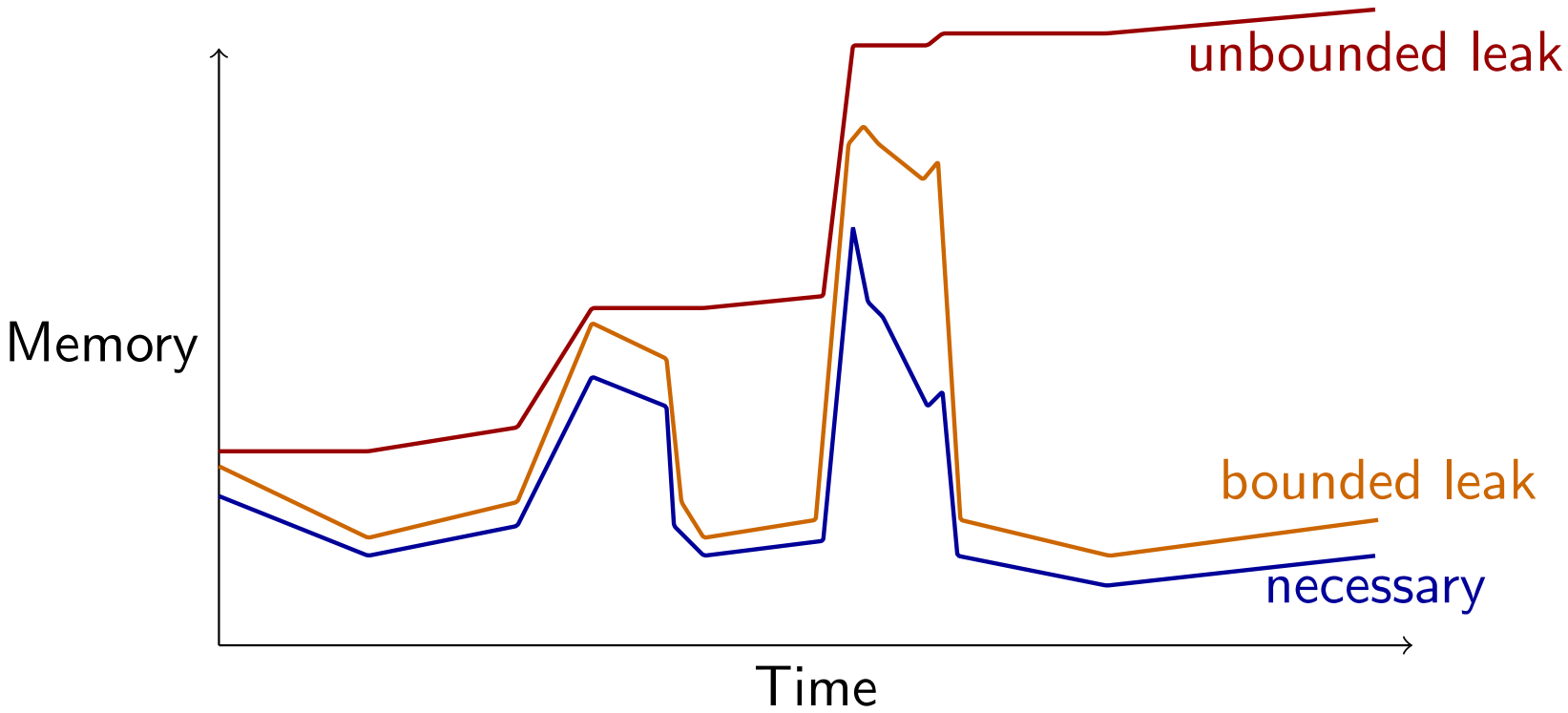
Catching space leaks caused by excessive laziness at compile-time

- Avoiding space leaks is important.
- Understanding lazy evaluation is helpful but not necessary.
- We propose an idiom: Stateful XOR Lazy
- And a tool to enforce it: th-deepstrict

Types of Space Leak

A *space leak* is when more memory is used than necessary[1].

- Bounded by X: memory is kept until event X
- Unbounded: memory is never released



[1] *Leaking Space: Eliminating memory hogs*, Neil Mitchell, ACM Queue 2013

Causes of Space Leaks

Space leaks can be caused by:

- Excessive laziness
- Inappropriate datatype usage, eg, Lists have a linear overhead, Vector has a constant overhead.
- Function closures
- Stack usage
- And others!

Laziness

- Laziness means work is only done when demanded.
- This comes with a cost: deferred work must be represented using a *thunk*. A *thunk* keeps alive all data needed to do the work.

Example: an all too lazy sum

The classic space leak from laziness: a lazy left fold

```
foldl (+) 0 [1..1000]
```

Expectation: Constant

Reality: Linear

Example: an all too lazy sum

The solution is to use a strict left fold:

```
foldl' (+) 0 [1..1000]
```

Expectation: Constant
Reality: Constant

Example: a too clever average

Calculates average with just one traversal! But it has a space leak.

```
data State = State { sum :: Int, length :: Int }

step :: State -> Int -> State
step old x =
  State { sum = sum old + x, length = length old + 1}

finish :: State -> Int
finish final = sum final `div` length final

finish $
  foldl' step (State 0 0) [0..10000]
```

While the top-level `State` is evaluated, the values in the fields are not.

Example: a clever average (solution)

Solution make the fields strict.

```
data State = State { sum :: !Int, length :: !Int }
```

Moral: the avoid of laziness you need *deep* strictness.
We want *normal form* not *weak head normal form*.

Unbounded leaks

- So far, we have only seen bounded space leaks.
- Let's look at an unbounded leak.

Unbounded leaks

Web server version, storing our original `State` in an `IORef`.

```
data State = State { sum :: Int, length :: Int }

main = do
  ref <- newIORef (State 0 0)
  let handleAdd x =
        modifyIORef' ref (\old -> step old x)
  let handleGetAverage =
        readIORef ref >>= print . finish
  ...
```

Unbounded space leak.

Lazy and stateful

- All of these examples are both lazy and stateful.
- Stateful because we have a sequence of values computed where the next is computed from the current.
- Lazy because we don't fully evaluate evaluate the current value.
- Both are necessary for the space leak.
- Laziness creates thunks.
- Statefulness links them up into a chain.

Statefulness

Many forms of statefullness:

- Recursion, folds, loops, etc
- The `State` monad
- Mutable state `IORef`, `MVar`, `TVar`, etc

Stateful AND Lazy implies space leak

Stateful XOR Lazy

Abolish your state

- Stateful code should be kept to a minimum but is unavoidable.
- Externalize state in a database
- If you need state, make it as local as possible.
- Global state can lead to unbounded space leaks.

Stateful code should be deep strict

If code is necessarily stateful, then make it deep strict.

A *datatype is deep strict* iff:

- All fields of all constructors are strict.
- Types of all fields are deep strict.

th-deepstrict

- Open-sourced by Tracsis
- Non-trivial Template Haskell (upstreamed into th-abstraction)
- Recursively traverse your datatype and find lazy fields/types.
- Allows overriding inferred strictness of datatypes.
- Supports golden/ratchet tests.

th-deepstrict: Example

```
> assertDeepStrict =<< [t|State|]
    Main.State
is not Deep Strict, because:
Main.State
  con Main.State
    field Main.sum is lazy
    field Main.length is lazy
  |
5 | assertDeepStrict =<< [t|State|]
  |
```

Packages for strict code

- `strict-wrapper` by Tom Ellis
  (See: *Make Invalid Laziness Unrepresentable*)
- `strict` (strict `Maybe`, etc)
- `strict-containers`
- Strict vector problem

Alternatives: `deepseq`

We can use `rnf` from the `deepseq` package.
Issues:

- Error prone: requires manual annotation.
- Expensive: must traverse entire datatype, $\mathcal{O}(n)$

Alternatives: `nothunks`

`nothunks` by Edsko de Vries allows writing unit tests to check for thunks.
Issues:

- Requires test coverage: tough when state space is large.
- Isn't checked at compile-time.
- Great fit for data structures.

- Stateful XOR lazy
- Enforce this with `th-deepstrict`