Unboxing Sum Types

Johan Tibell - FP-Syd 2017-05-24

Work in (slow) progress

- Ömer Sinan Ağacan
- Ryan Newton
- José Manuel Calderón Trilla

I ♥ NULL

Reason 1: Job security

```
Program received signal SIGSEGV,
Segmentation fault.
0x0000000000400618 in causeAnError
(a=@0x7fffffffe06f) at main.cpp:6
6   a = *p;
(gdb) bt
#0  0x0000000000400618 in causeAnError
(a=@0x7fffffffe06f) at main.cpp:6
#1  0x000000000040063e in main (argc=1,
argv=0x7fffffffe158) at main.cpp:13
```

Reason 2: Performance (this talk)

Haskell:

```haskell
module Data.HashMap.Strict where

-- Allocates a Just constructor.
lookup :: (Eq k, Hashable k)
       => k -> HashMap k v
       -> Maybe v
```

Java:

```java
// Returns the value to which the
// specified key is mapped, or null
// if this map contains no mapping
// for the key.
class HashMap<K,V> {
  V get(K key);
}
```

Allocation isn't free!

Recap: heap layout

- Constructor values are allocated on the heap.
- Example:

```
data Maybe a = Just a | Nothing
```

How costly is an allocation anyway?

- Introduces a new branch (for the heap check)
  - The function might not have needed one otherwise (the case for `lookup`)
  - Potentially in each iteration of a tight "loop" (`lookup` is recursive)
  - Increases binary size
- Uses more space
  - Worse cache efficiency.
- Introduces indirections
  - To access the `a` in `Just a` we need to follow two pointers instead of one:

The first idea

- Could we implement `Maybe` using the Java representation?
  - Use a null pointer (or some designated pointer value) to represent `Nothing`.
  - Point directly to the a instead of the `Just` constructor.
  - We can think of this as removing the box (i.e. **unboxing**) around the return value.
- Filed https://ghc.haskell.org/trac/ghc/ticket/4937 AKA "we should do something about this".
  - Some initial discussion.
  - How do we represent `Maybe (Maybe a)` ?
  - Perhaps this could be made to work for strict `Maybe`s
- About 5 years pass...

5 years later

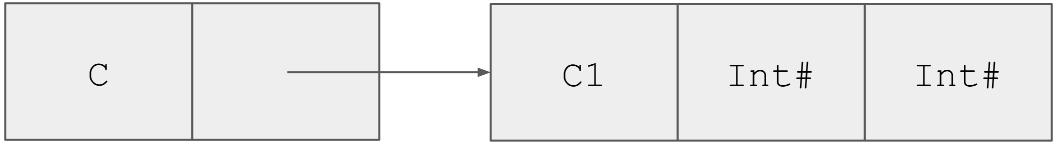
- Still annoyed about the extra allocation in `Data.HashMap.lookup`.
- New approach: unbox all the sums (including `Maybe`)!
  - We can already unbox all products.

Recap: unboxing of products

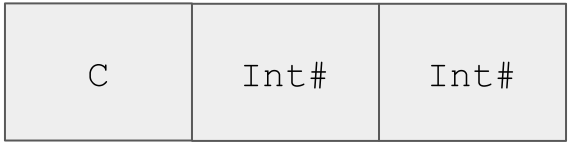
- GHC can already unbox products:

```
data T = C !MyProd
data MyProd = C1 !Int !Int
```

- Before:



- After:



What does unboxing a sum mean?

- Representation similar to C-style tagged unions.
- Sized to fit the biggest variant.
  - This is important (or at least helpful) for GC.
- Unlike C, we have to treat pointer fields specially, because of GC.

Basic algorithm

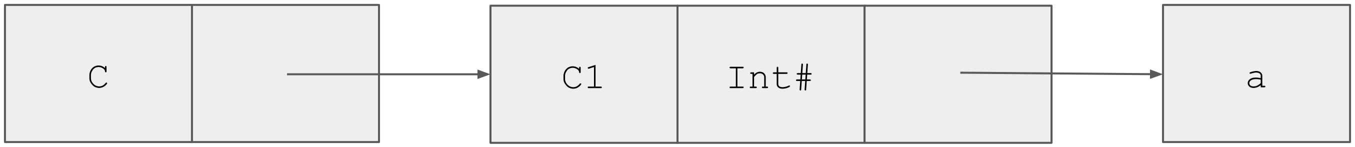
- We only unbox strict fields (same as for products):

```
data T a b = C !(MySum a b)
data MySum a b = C1 !Int a | C2 !Char b
```

- For every constructor of the original (boxed) sum type, we split the fields into different categories depending on their sizes.
  - Pointer fields are put in a separate category for GC reasons.
  - Fields are reordered as needed.
- Compute a representation based on these categories.

Basic algorithm - example

Before:



After:

Finding things to unbox

- We already covered sums used in strict fields.
- What about arguments and return values?

```
f :: SomeSum -> SomeOtherSum
```

- Extend existing strictness analysis for products to sums:
  - Allows us to spot the unboxing opportunity in `lookup`.
  - Details hopefully in an upcoming paper!

Does this solve our Maybe problem?

- `lookup` return value is now represented as two words (tag + pointer).
- No allocation.
- In this particular case using null pointers would have been better (we could return only one value).
    - But if we have a strict `Maybe` or if GHC ever starts unboxing polymorphic fields the general representation is better.

Implementation

1. Introduce anonymous, unboxed sums in GHC (similar to existing unboxed tuples):
   `(# Int | a #)` - Type of an anonymous, unboxed sum of an `Int` and an `a`.
2. Convert strict sums to anonymous, unboxed sums when compiling.
   a. If some heuristic thinks that makes sense.
3. Convert anonymous, unboxed sums to product types later in compilation.

Some early numbers

- Benchmarking is hard!
  - Lots of other inefficiencies, in particular laziness and the representation of polymorphic fields (i.e. pointers), hide speed-ups.
  - Don't benchmark micro optimizations on code using e.g. linked lists (e.g. nofib).
  - Other optimizations (e.g. inlining) are sometimes enough in simpler cases.
- Microbenchmark: Linear search in array of (8) unboxed integers.

Some early numbers

```
data MaybeS = JustS !Int | NothingS

linSearch :: IntArray -> Int -> MaybeS
linSearch !haystack !needle = loop 0
  where
    loop ix | ix >= length haystack        = NothingS
            | index haystack ix == needle = JustS ix
            | otherwise                    = loop (ix+1)
```

Some early numbers - Unoptimized

```
17,180,306,560 bytes allocated in the heap
       447,432 bytes copied during GC
        44,384 bytes maximum residency (2 sample(s))
        39,280 bytes maximum slop
             1 MB total memory in use (0 MB lost due to fragmentation)

                                   Tot time (elapsed)  Avg pause  Max pause
Gen  0     32767 colls,     0 par    0.003s   0.057s     0.0000s    0.0003s
Gen  1         2 colls,     0 par    0.000s   0.000s     0.0001s    0.0001s

INIT    time    0.000s  (  0.000s elapsed)
MUT     time   14.124s  ( 14.130s elapsed)
GC      time    0.003s  (  0.058s elapsed)
EXIT    time    0.000s  (  0.000s elapsed)
Total   time   14.199s  ( 14.188s elapsed)

%GC     time       0.0%  (0.4% elapsed)

Alloc rate    1,216,360,691 bytes per MUT second

Productivity 100.0% of total user, 99.6% of total elapsed
```

Some early numbers - Optimized

```
     437,376 bytes allocated in the heap
       3,480 bytes copied during GC
      44,384 bytes maximum residency (1 sample(s))
      17,056 bytes maximum slop
           1 MB total memory in use (0 MB lost due to fragmentation)

                                   Tot time (elapsed)  Avg pause  Max pause
Gen  0         0 colls,     0 par    0.000s   0.000s     0.0000s    0.0000s
Gen  1         1 colls,     0 par    0.000s   0.000s     0.0002s    0.0002s

INIT    time    0.000s  (  0.000s elapsed)
MUT     time   13.671s  ( 13.669s elapsed)
GC      time    0.000s  (  0.000s elapsed)
EXIT    time    0.000s  (  0.000s elapsed)
Total   time   13.741s  ( 13.669s elapsed)

%GC     time       0.0%  (0.0% elapsed)

Alloc rate    31,994 bytes per MUT second

Productivity 100.0% of total user, 100.0% of total elapsed
```

Other kind of improvements

- Reduced major GC pauses due to holding on to less data.
  - Major GCs are typically O(heap).
  - Less live data, shorter GC.
  - Not yet quantified.

Binary size

- Before: 207 bytes
- After: 62 bytes (70% reduction)
- Mainly due to not having a heap check
  - Avoids code for spilling registers

The future

- Some of this should be included in the latest GHC (i.e. anonymous, unboxed sums).
- Strictness analysis still needs work.
- Other optimizations that would improve impact:
  - Better representation/compilation of polymorphic fields.
    - Polymorphic fields cannot be unboxed today.
  - More strictness.
- Random idea: a lazy field can be thought of as an unboxed sum of a value in WHNF and a thunk. Perhaps an interesting representation to try.

Thank you!

Questions?