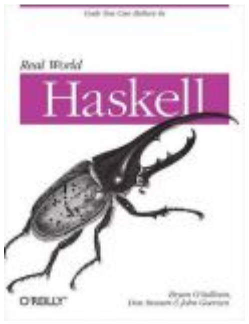
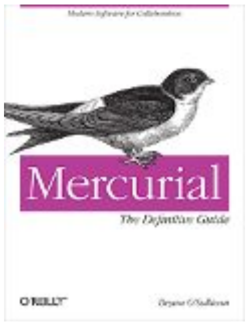
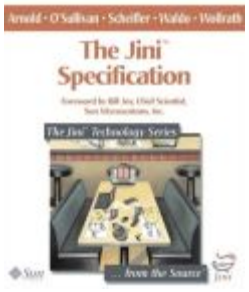
My Books



Real World Haskell



Mercurial: The Definitive Guide



The case of the mysterious explosion in space

Posted on 2012-09-12 by Bryan O'Sullivan — 3 Comments ↓

The case of the mysterious explosion in space

A few months ago, reports began to filter in of an unhappy problem with the Haskell `text` package: it was causing huge object files to be generated when a file contained lots of string literals.

I didn't notice the initial report (it was posted to a busy mailing list that I don't try to keep up with), but Michael Snoyman was kind enough to take that message and file a bug.

The culprit was this very simple function definition, which converts a string from the venerable Haskell `String` type to the more modern `Text`.

```
pack :: String -> Text
pack txt = unstream
             (Stream.map safe
           (Stream.streamList txt))
```

The definition of `pack` is too innocent to be at fault; the problem lies with the extra directives that `text` gives to the compiler.

```
{-# INLINE pack #-}
{-# INLINE Stream.unstream #-}
{-# INLINE Stream.map #-}
{-# INLINE Stream.streamList #-}
```

The Jini™ Specification

Recently

- Sometimes, the old ways are the best
- criterion 1.0
- Win bigger statistical fights with a better jackknife
- A major upgrade to attoparsec: more speed, more power
- Top Haskell packages seen through graph centrality beer goggles

Meta

- Log in
- Entries RSS
- Comments RSS
- WordPress.org

By asking the compiler to inline every function, we guaranteed that every string literal would result in a lot of code being generated. Worse, all of this space would be entirely redundant, consisting of repeated copies of exactly the same code.

(You might well wonder why we'd insist on inlining *any* of these functions, if the cost in space is so high. The answer is that inlining is key to why the `text` package achieves good performance. That deserves an article of its own, so I'll return to the subject soon.)

Have you ever wondered how GHC represents string literals? Instead of somehow statically constructing a linked list of characters and emitting that into an object file, it's smarter.

For strings of pure ASCII, GHC generates a packed zero-terminated byte sequence that looks like this.

```
.const
.align 3
.align 0
_co0_str:
        .byte   102     # f
        .byte   111     # o
        .byte   111     # o
        .byte   0
```

(For strings that contain Unicode or control characters, GHC still generates a packed sequence of bytes, but this time they're specially encoded.)

In Haskell, these byte sequences have a type that is simply a fixed address, `Addr#`. During compilation, GHC takes a string literal and prefixes it with a function to convert from `Addr#` to `String`.

```
-- What we write:
```

```haskell
foo :: String
foo = "foo"

-- What GHC generates:
foo :: String
foo = GHC.CString.unpackCString# co0_str
  where co0_str :: Addr#
        co0_str = "foo"
```

One of the lovelier features of GHC is that it exposes some of its internal machinery to authors. We’re paying a price for our aggressive use of its `INLINE` directive; is there another GHC feature we can use to save the day?

Enter the rewrite rule, a way of telling GHC how to perform source-to-source transformations.

Here is a naive attempt to specify a rewrite rule that might help us. First, we define a version of `pack` that we tell the compiler to *never* inline, then we supply a rewrite rule that tells the compiler to substitute the never-inlined version of `pack` for the normal version.

```haskell
packNOINLINE :: String -> Text
packNOINLINE = unstream . Stream.map
safe . Stream.streamList
{-# NOINLINE packNOINLINE #-}

{-# RULES "TEXT literal" forall a.
    pack s = packNOINLINE s
  #-}
```

Although this rule works and generates correct code, it swaps one problem for another: the object files we generate shrink dramatically, but we’ve defeated some of the compiler’s opportunities to improve the code it emits.

Oh, and during compilation, remember that after GHC has finished processing a string literal, we start out with an

Addr#, then GHC converts to a String for us, and finally we convert to a Text. That intermediate step galls me, even though it really has no practical consequences.

Happily for us, GHC’s rewrite rules are applied cleverly: rather than being a simple one-shot affair, GHC keeps trying to apply rewrite rules as it optimises a program.

The critical addition to our rule is to recognise that when we write a string literal, it will be transformed into an application of GHC.String.unpackCString#, and target our rule to an expression containing this.

```
-- Introduce a new function ...
Text.unpackCString# :: Addr# -> Text
Text.unpackCString# addr#
  = unstream (Stream.streamCString#
addr#)
{-# NOINLINE unpackCString# #-}

-- ... and use it!
{-# RULES "TEXT literal" forall a.
    unstream (Stream.map safe
      (Stream.streamList
        (GHC.String.unpackCString# a)))
      = Text.unpackCString# a #-}
```

With this rewrite rule, GHC will transform code that we /never actually wrote/, using a type (Addr#) that we don’t use in our code. The conditions that trigger this rule will arise only when we define a literal Text value. This means that productive uses of stream fusion will not be affected. Even better, this rule eliminates that pesky intermediate String value, since the new unpackCString# performs a direct translation. Not a bad trick!

‹ Performance: yes, it’s worth looking at the small stuff

A fast new SipHash implementation in Haskell ›

Posted in haskell

3 comments on “The case of the mysterious explosion in space”

Thiago Negri says:

2012-09-14 at 06:08

Can you please explain it a little more?
I don’t get it. Did it solve the problem? Why? How?

Maybe adding a step-by-step explanation of the GHC’s rewrite, and why the old version was fatter than the current. Did it maintain the inlining performance gain?

Thanks.

Peter Wortmann says:

2012-10-04 at 09:33

Hm, I remember once toying around with similar code in GHC – trying to figure out why the “text/str” rule in Pretty.lhs never seemed to fire when I expected it to. It turned out to be a very tricky issue due to the “unpack” rule from Base.lhs, which replaces all unpackCString# with unpackFoldrCString# in order to allow for list fusion.

I finally turned to some pretty nasty hacks to get around that – and eventually backed out of the whole thing and just hard-coded ptext everywhere. I’d be curious about what makes it work here. Maybe you catch the expression when it gets transformed back?

Dag says:

2012-10-04 at 10:18

One problem is that it’s easy to “escape” these neat rewrite rules, for example:

```
{-# LANGUAGE GeneralizedNewtypeDeriving,
OverloadedStrings #-}
newtype UserName = UserName Text
deriving IsString
```

-- Does not trigger the rewrite rules
bos :: UserName
bos = "bos"

You can work around it by using the `UserName` constructor explicitly, but in some packages the constructor isn’t exported and `fromString` is *the* API.

It would be nice if the above “just worked”, or at least if it would possible to add your own rewrite rules and the necessary text internals were exposed somewhere, with documentation for how to go about it. Similarly for manually written `IsString` instances.

Website

Comment

Post Comment