GHC/Using rules

From HaskellWiki

< GHC

Contents

1 Using rules in GHC

GHC's rewrite rules (invoked by the RULES pragma) offer a powerful way to optimise your program. This page is a place for people who use rewrite rules to collect thoughts about how to use them.

If you aren't already familiar with RULES, read this stuff first:

- The relevant section of the GHC user manual (http://www.haskell.org /ghc/docs/latest/html/users_guide/rewrite-rules.html)
- Playing by the rules: rewriting as a practical optimisation technique in GHC (http://research.microsoft.com/%7Esimonpj/Papers/rules.htm) . This paper, from the 2001 Haskell workshop, describes the idea of rewrite rules.

1.1 Advice about using rewrite rules

- Remember to use the flag `-fglasgow-exts` and the optimisation flag `-O`
- Use the flag `-ddump-simpl-stats` to see how many rules actually fired.
- For even more detail use `-ddump-simpl-stats -ddump-simpl-iterations` to see the core code at each iteration of the simplifer. Note that this produces **lots** of output so you'll want to direct the output to a file or pipe it to `less`. Looking at the output of this can help you figure out why rules are not firing when you expect them to do so.
- Another tip for discovering why rules do not fire, is to use the flag `-dverbose-core2core`, which (amongst other things) produces the AST after every rule is fired. This can help you to examine whether one rule is creating an expression that thereby prevents another rule from firing, for example.
- You need to be careful that your identifiers aren't inlined before your RULES have a chance to fire. Consider

```
{-# INLINE nonFusable #-}
{-# RULES "fusable/aux" forall x y.
      fusable x (aux y) = faux x y ; #-}
nonFusable x y = fusable x (aux y)
```

  You are possibly surprised when the rule for
  `fusable`
  does not fire. It may well be that
  `fusable`
  was inlined before rules were applied.
  To control this we add an
  `NOINLINE`
  or an
  `INLINE [1]`
  pragma to identifiers we want to match in rules, to ensure they haven't disappeared by the time the rule matching comes around.

To have rewrite rules fire in code interpreted in GHCi, you'll need to explicitly ask for -frewrite-rules in an options pragma at the start of your file.

1.2 Structure of simplification process

There are currently the simplifier phases "gentle", 2, 1, 0, each consisting of 4 iterations. Starting with GHC 6.10 you can alter these numbers with the command line options `-fsimplifier-phases` and `-fmax-simplifier-iterations`. However in each iteration rules are applied multiple times, until rules can no longer be applied. That rules can no longer be applied is due to the fact that the simplifier chooses some way from outer to inner or reverse. Actually it's always the same, but you should not rely on a particular order, mind you?

The good effect is that arbitrary big expressions of the type
`map f0 . map f1 . ... . map fn`
can be collapsed to a single
`map`
by the single rule

`map f (map g xs) = map (f . g) xs`
. The bad effect is that rules like
`f x y = f y x`
lead to an infinite loop.

1.3 Example:
`map`

(This example code is taken from GHC's `base/GHC/Base.lhs` module.)

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

```
mapFB ::  (elt -> lst -> lst) -> (a -> elt) -> a -> lst -> lst
{-# INLINE [0] mapFB #-}
mapFB c f x ys = c (f x) ys
```

The rules for map work like this.

Up to (but not including) phase 1, we use the `"map"` rule to rewrite all saturated applications of map with its build/fold form, hoping for fusion to happen. In phase 1 and 0, we switch off that rule, inline build, and switch on the `"mapList"` rule, which rewrites the foldr/mapFB thing back into plain map.

It's important that these two rules aren't both active at once (along with build's unfolding) else we'd get an infinite loop in the rules. Hence the activation control below.

The `"mapFB"` rule optimises compositions of map.

This same pattern is followed by many other functions:

e.g.
`append`
,
`filter`
,
`iterate`
,
`repeat`
, etc.

```
{-# RULES
"map"       [~1] forall f xs.   map f xs               = build (\c n -> foldr (mapFB c f) n xs)
"mapList"   [1]  forall f.      foldr (mapFB (:) f) []  = map f
"mapFB"     forall c f g.       mapFB (mapFB c f) g     = mapFB c (f.g)
#-}
```

2 Questions

2.1 Order of rule-matching

For example, let's say we have two rules

```
  "f->g" forall x y .    f x (h y) = g x y
  "h->g" forall x   .    h x = g 0 x
```

and a fragment of the AST corresponding to

```
  f a (h b)
```

Which rule will fire? "f->g" or "h->g"? (Each rule disables the other.)

Answer: rules are matched against the AST for expressions basically *bottom-up* rather than top-down. In this example, "h->g" is the rule that fires. But due to the nature of inlining and so on, there are absolutely no guarantees about this kind of behaviour. If you really need to control the order of matching, phase control is the only reliable mechanism.

2.2 Confluent term rewriting system

Since there is no guarantee on a particular order of rule application, except the control by phases, you should assert that the result is the same independent of the order of rule application. This property of a term rewriting system is called confluence. See for example:

```
{-# RULES
      "project/project" forall x.
          project (project x) = project x ;

      "project/foo" forall x.
          project (foo     x) = projectFoo x ;
  #-}

f = project . project . foo
```

For this set of rewriting rules it matters whether you apply "project/project" or "project/foo" first to the body of `f`. In the first case you can apply additionally "project/foo" yielding `projectFoo x`, whereas the second case leaves you with `project (projectFoo x)`.

To make the system confluent you should add the rule

```
project (projectFoo x) = projectFoo x
```

You can complete a rule system this way by hand, although it'd be quite a nice thing to automate it in GHC.

We assume that non-confluent rewriting systems are bad design, but it is not clear how to achieve confluence for any system.

2.3 Pair rules

It is often useful to provide two implementations of a function, and have rewrite rules pick which version to use depending on context. In both GHC's foldr/build fusion, and more extensively in Data.ByteString's stream fusion system, pair rules are used to allow the compiler to choose between two implementations of a function.

Consider the rules:

```
    "FPS length -> fused"  [~1]
        length = F.strConsumerBi F.lengthS

    "FPS length -> unfused" [1]
        F.strConsumerBi F.lengthS = length
```

This rule pair tells the compiler to rewrite occurences of
`length`
to a stream-fusible form in early simplifications phases, hoping for fusion to happen. However, if by phase 1 (remember that phases count down from 4), the fusible form remains unfused, it is better to rewrite it back to the unfused-but-fast implementation of length. A similar trick is used for
`map`
in the base libraries. As we want to match
`length`
in the rules, we need to ensure that it isn't inlined too soon:

```
    length :: ByteString -> Int
    length (PS _ _ l) = assert (l >= 0) $ l
    {-# INLINE [1] length #-}
```

and we need
`strConsumerBi`
to stick around for even longer:

```
    strConsumerBi :: (Stream -> a) -> (ByteString -> a)
    strConsumerBi f = f . readStrUp
    {-# INLINE [0] strConsumerBi        #-}

    lengthS :: Stream -> Int
    lengthS  ...
    {-# INLINE [0] lengthS #-}
```

Pair rules thus provide a useful mechanism to allow a library to provide multiple implementations of a function, picking the best one to use based on context.

2.4 Custom specialisation rules

Another use for rules is to replace a particular use of a slow, polymorphic function with a custom monomorphic implementation.

Consider:

```
zipWith :: (Word8 -> Word8 -> a) -> ByteString -> ByteString -> [a]
```

This is a bit slow, but useful. It's often used to zip ByteStrings into a new ByteString, that is:

```
pack (zipWith f p q)
```

We'd like to spot this, and throw away the intermediate [a] created. And also use a specialised implementation of:

```
zipWith :: (Word8 -> Word8 -> Word8) -> ByteString -> ByteString -> ByteString
```

We can use rules for this:

```
"FPS specialise pack.zipWith" forall (f :: Word8 -> Word8 -> Word8) p q .
    pack (zipWith f p q) = zipWith' f p q
```

This rule spots the specific use of zipWith we're looking for, and replaces it with a fast, specialised version.

2.5 Rules and sections

This is useful for higher order functions as well. As of ghc 6.6, the rule LHS syntax has been relaxed, allowing for sections and lambda abstractions to appear. Previously, only applications of the following form were valid:

```
"FPS specialise break (x==)" forall x.
    break ((==) x) = breakByte x
```

That is, replace occurences of:

```
break (x==)
```

with the optimised breakByte function. This code illustrates how higher order functions can be rewritten to optimised first order equivalents, for special cases like

```
(==)
```

. In the case of Data.ByteString, functions using

```
(==)
```

or

```
(/=)
```

are much faster when implemented with memchr(3), and we can use rules to do this, as long as it is possible to match sections. In ghc 6.6 we can now write:

```
"FPS specialise break (==x)" forall x.
    break (==x) = breakByte x

"FPS specialise break (x==)" forall x.
    break (x==) = breakByte x
```

Some fragility remains in this rule though, as described below.

2.6 Literals, dictionaries and sections

Consider:

```
    break (== 10)
```

Hopefully, this can be rewritten to a

```
breakByte 10
```

call, however, the combination of sections, literals and dictionaries for Eq makes this rather fragile.

The rule for break ends up translated by GHC as;

```
     forall ($dEq :: base:GHC.Base.Eq base:GHC.Word.Word8)
            (x :: base:GHC.Word.Word8)

        break (base:GHC.Base.== @ base:GHC.Word.Word8 $dEq x) =
        breakByte x
```

Notice the LHS: an application of the selector to a (suitably-typed) Eq dictionary. GHC does very little simplification on LHSs, because if it does too much, the LHS doesn't look like you thought it did. Here it might perhaps be better to simplify to GHC.Word.Word8.==, by selecting from the dictionary, but GHC does not do that.

When this rules works, GHC generates exactly that pattern; we get

```
        eq = (==) deq
        main = ... break (\x. eq x y) ...
```

GHC is anxious about substituting eq inside the lambda, but it does it because (==) is just a record selector, and hence is very cheap.

But when we put a literal inline, we get an (Eq a) constraint and a (Num a) constraint (from the literal). Ultimately, 'a' turns out to be Int, by defaulting, but we don't know that yet. So GHC picks the Eq dictionary from the Num dictionary:

```
        eq = (==) ($p1 dnum)
        main = ... break (\x. eq x y) ...
```

Now the 'eq' doesn't look quite so cheap, and it isn't inlined, so the rule does not fire. However, GHC 6.6 has been modified to believe that nested selection is also cheap, so that makes the rule fire.

The underlying lesson is this: the only robust way to make rules fire is if the LHS is a normal form. Otherwise GHC may miss the fleeting moment at which (an instance of) the rule LHS appears in the program. The way you ensure this is with inline phases: don't inline LHS stuff until later, so that the LHS stuff appears in the program more than fleetingly.

But in this case you have (==) on the LHS, and you have no phase control there. So it gets inlined right away, so the rule doesn't match any more. The only way the rule "works" is because GHC catches the pattern right away, before (==) is inlined. Not very robust.

To make this robust, you'd have to say something like

```
instance Eq Word8 where
  (==) = eqWord8

eqWord8 = ..
{-# NOINLINE [1] eqWord8 #-}

{-# RULES
 "FPS specialise break (x==)" forall x.
    break (x`eqWord8`) = breakByte x
  #-}
```

2.7 Rules and method sharing

GHC by default instantiates overloaded methods by partially applying the original overloaded identifier. This facilitates sharing of multiple method instances with one global definition. However, since a new function name is created during this process, rules matching the original names will not fire. Here is an example from `Control.Arrow`:

```
class Arrow a where
  arr :: (b -> c) -> a b c
  first :: a b c -> a (b,d) (c,d)
  (>>>) :: a b c -> a c d -> a b d

{-# RULES
"compose/arr"   forall f g . arr f >>> arr g = arr (f >>> g)
"first/arr"     forall f . first (arr f) = arr (first f)
...
 -#}
```

Consider an instance of an arrow and some code on which the rules above should fire:

```
newtype SF a b = SF ([a] -> [b])

instance Arrow SF where
  arr f = SF (map f)
  ...

foo :: SF (Int,Int) (Int,Int)
foo = first (arr (+1)) >>> first (arr (+2) >>> arr (+3))
```

GHC would generate intermediate code like:

```
dsf :: Arrow SF
dsf = ...
```

```
first_1 = Control.Arrow.first SF dsf
arr_1 = Control.Arrow.arr SF dsf

foo = first_1 (arr_1 (+1)) ...etc...
```

Due to the introduction of `first_1` and `arr_1`, the rules no longer match since the names have changed.

The solution is to switch off sharing with the `-fno-method-sharing` flag.

2.8 Coexistence of fusion frameworks

I like to use my own fusion framework on an existing data structure because I want to experiment with it or because I have a specific application and I want to optimize the fusion framework for it. How can I disable the fusion rules shipped with that data structure - or at least defer them until the optimizer is finished with my rules?

Answer: Second part of the question first: Asserting that your rules are used before the standard rules is not possible with GHC up to version 6.8. The current system is quite monolithic in this respect. It would be a nice application of a more sophisticated rule control system that allows any number of simplifier phases with explicit statements which phase shall be entered after which other phase.

First part of the question:

You may wrap the data structure in a
`newtype`
or, to be entirely safe, redefine the data structure.

This means that several functions have to be lifted to the wrapped data type. This is tedious, but given that you make an application specific fusion framework, the set of basic functions will be different from that of the general data structure.

You might have planned to make your data type distinct anyway, may it be for the
`Arbitrary`
class of QuickCheck.

Remember to attach a NOINLINE pragma to the wrapped functions, otherwise the compiler may unpack the wrappers and starts fusion on the underlying data structure.

2.9 Interaction of INLINE and RULES

Rules can be seen as alternative function definitions. They are somehow special because they do not allow pattern matching, but allow expressions in arguments using existing variable names in the left hand side. Since pattern matching can be moved into a

`case`, fusion rules are actually the more flexible way to define functions. Alternative function definition means that the compiler has to decide which definition to use: The original function definition (by an explicit call or by inlining/unfolding) or one of the optimizer rules ("a rule fires").

Now the critical question is: How do inlining and rule application interact?

Inlining is always an option for the compiler, whether you use the INLINE pragma or not. The compiler measures some kind of size of the function and decides whether a function is small enough in order to be inlined. The INLINE pragma reduces this size virtually. But the inlining can still be omitted. In that sense RULES are applied more aggressively because they don't respect a size measurement of functions.

It's interesting to note, that declaring a function as INLINE disables fusion for the inner of the function. E.g., you can expect that

```
doubleMap f g = map f . map g
```

is fused to

```
doubleMap f g = map (f . g)
```

However, if you add

```
{-# INLINE doubleMap #-}
```

then the function definition is not fused to `map (f . g)`.

The compiler expects that the function will never called as is, and thus skips fusion. That is, you cannot control the application order of rules by enclosing expressions in function definitions, that shall be fused before fusion with outer parts.

However, since the compiler may decide not to inline, the compiler may leave you with a call to the unoptimized `doubleMap`. You could prevent this e.g. by:

```
{-# NOINLINE doubleMap #-}
doubleMap f g = map f . map g   -- will be fused

{-# RULES
   "doubleMap" forall f g. doubleMap f g = map f . map g
#-}
```

This makes sure that the right hand side of `doubleMap` will be optimised for those cases when the rule doesn't fire, e.g. when `doubleMap` is applied to less than two arguments.

2.10 Interaction of SPECIALISE and INLINE

`SPECIALISE`
pragmas are also some kind of rules, where calls to functions with a specific type class dictionary are replaced by calls to versions of a function which are instantiated to a specific type. If you want to use a function the inlined way, it might be a bad idea to add the
`SPECIALISE`
pragma, since this will replace a call to the function by a call to a specialised function instead of inlining it.

3 Future of rules in GHC

GHC has much too rigid a notion of phases up to version 6.8. There are precisely 3, namely 2 then 1 then 0, and that does not give enough control. Really we should let you give arbitrary names to phases, express constraints (A must be before B), and run a constraint solver to map phase names to a linear ordering. The current system is horribly non-modular. (See Haskell-Cafe on Properties of optimizer rule application? (http://www.haskell.org/pipermail/haskell-cafe/2008-January/038198.html))

Categories:

- GHC
- Performance
- Program transformation
