Analysing GHC Rewrite Rules

Julian Nagele

Department of Computer Science
University of Innsbruck

AJSW 2016 – 45th TRS meeting September 8, 2016

Motivation

Haskell

```
map f [] = []
map f (h:t) = f h : map f t

{-# RULES
  "map/map" forall f g xs.
     map f (map g xs) = map (f . g) xs
  #-}
```

- optimization of Haskell programs using rewrite rules
- library authors can use rules to express domain-specific optimizations that the compiler cannot discover for itself
- simple, but effective in optimizing real programs

GHC Rewrite Rules

Properties

- GHC makes no attempt to verify that the rule is indeed an identity
- GHC makes no attempt to ensure that the right hand side is more efficient than the left hand side
- GHC makes no attempt to ensure that the set of rules is confluent, or even terminating

As Higher-Order Rewrite System

$$\begin{aligned}
\mathsf{map}\ (\lambda x.\ F\ x)\ \mathsf{nil} &\rightarrow \mathsf{nil} \\
\mathsf{map}\ (\lambda x.\ F\ x)\ (\mathsf{cons}\ h\ t) &\rightarrow \mathsf{cons}\ (F\ h)\ (\mathsf{map}\ (\lambda x.\ F\ x)\ t) \\
\mathsf{map}\ (\lambda x.\ F\ x)\ (\mathsf{map}\ (\lambda x.\ G\ x)\ xs) &\rightarrow \mathsf{map}\ (\mathsf{o}\ (\lambda x.\ F\ x)\ (\lambda x.\ G\ x))\ xs
\end{aligned}$$

Syntax & Matching

Definition

The left hand side of a rule must take the the following form

$$\texttt{f}\ e_1\ \ldots\ e_n$$

where `f` is not quantified in the rule (i.e., not a variable), and the e_i are arbitrary expressions

- matching is modulo α
- pattern is η-expanded, but not expression (η-expanding expression might lead to laziness bugs)
- matching is not modulo β

Changes in Semantics

```
one = head . reverse . reverse $ [1..]

{-# RULES
  "reverse.reverse/id" reverse . reverse = id
  #-}
```

List Fusion

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f n [] = n
foldr f n (x:xs) = f x (foldr f n xs)

build :: (forall b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []

{-# RULES
  "foldr/build"
  forall f n (g :: forall b. (a -> b -> b) -> b -> b).
    foldr f n (build g) = g f n
  #-}
```

Challenge

rank-*n* polymorphic types

List Fusion

```
sum :: [Int] -> Int
sum xs = foldr (+) 0 xs

down :: Int -> [Int]
down v = build (\c n -> down' v c n)

down' 0 c n = n
down' v c n = c v (down' (v-1) c n)

sum (down 5)
=  {inline sum and down}
foldr (+) 0 (build (down' 5))
=  {apply the foldr/build rule}
down' 5 (+) 0
```

List Fusion

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f n [] = n
foldr f n (x:xs) = f x (foldr f n xs)

build :: (forall b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []

{-# RULES
  "foldr/build"
  forall f n (g :: forall b. (a -> b -> b) -> b -> b).
    foldr f n (build g) = g f n
  #-}
```

Challenge

rank-*n* polymorphic types

Inlining and Phases

- `sum` and `down` must be inlined for rule to be applicable
- `build` most not be inlined
- making rules applicable needs the right amount of inlining
- GHC implements phases for inlining and firing rules

```
{-# INLINE 2 build #-}
build g = g (:) []
```

Specialization

```
genericLookup :: Ord a => Table a b   -> a   -> b
intLookup     ::          Table Int b -> Int -> b

{-# RULES
  "genericLookup/Int" genericLookup = intLookup
  #-}
```

- GHC will replace `genericLookup` by `intLookup` whenever the types match

Summary

- GHC uses rewrite rules to implement optimization
- idea: analyze those rules with rewriting techniques and tools

Obstacles

- rank-*n* polymorphism
- rewriting is partitioned into phases – interplay with inlining
- $\alpha\beta\eta$
- ...

Playing by the rules: rewriting as a practical optimization technique in GHC
Simon Peyton Jones, Andrew Tolmach, Tony Hoare
Proc. 2001 Haskell Workshop, 2001

Glasgow Haskell Compiler Users Guide
`https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/`