Rewrite rules for the Disciplined Disciple Compiler

Amos Robinson

November 15, 2012

Why do we want rewrite rules?

```
RULES
    map f (map g xs) = map (f.g) xs
    textToString . stringToText = id
    map id xs = xs

map textToString
    (map stringToText ["a","b","c"])
```

Why do we want rewrite rules?

```
RULES
    map f (map g xs) = map (f.g) xs
    textToString . stringToText = id
    map id xs = xs

map (textToString.stringToText)
                          ["a","b","c"]
```

Why do we want rewrite rules?

```
RULES
    map f (map g xs) = map (f.g) xs
    textToString . stringToText = id
    map id xs = xs

map id
                              ["a","b","c"]
```

Why do we want rewrite rules?

```
RULES
    map f (map g xs) = map (f.g) xs
    textToString . stringToText = id
    map id xs = xs
```

```
["a","b","c"]
```

But Disciple allows mutation

```
RULE addOr (x : Int).
     addInt x 0 = x

let y = 5                    in
let z = addInt y 0           in
let _ = updateInt y 23       in
     z
```

Evaluates to 5.

But Disciple allows mutation - after rewrite

```
RULE addOr (x : Int).
     addInt x 0 = x

let y = 5                    in
let z = y                    in
let _ = updateInt y 23       in
     z
```

Evaluates to 23! Bad!

Regions

```
addInt :: [r1 r2 r3 : %].
    Int r1 -> Int r2 -> Int r3

updateInt :: [r1 r2 : %].
    Mutable r1 =>
    Int r1 -> Int r2 -> Unit
```

Mutability - with region constraint

```
RULE add0r [r1 r2 : %] (x : Int r1).
     Const r1 =>
     addInt [r1] [r2] [r1] x 0
      = x

let y = 5                                    in
let z = addInt [r1] [r2] [r1] y 0            in
let _ = updateInt [r1] [r3] <w> y 23         in
     z
```

The rule can no longer fire.

Repeating effects

```
RULE mul2r (x : Int).
     mulInt x 2
      = addInt x x

let f = (\_. print "Oh!"; return 5)
in  mulInt (f ()) 2
```

Prints "Oh!" and returns 10.

Repeating effects - after rewrite

```
RULE mul2r (x : Int).
     mulInt x 2
      = addInt x x

let f = (\_. print "Oh!"; return 5)
in  addInt (f ()) (f ())
```

Prints "Oh!Oh!" and returns 10! Bad!

Repeating effects - fixed

```
RULE mul2r (x : Int).
     mulInt x 2
      = addInt x x

let f = (\_. print "Oh!"; return 5)
in  let x = f () in
          addInt x x
```

Prints "Oh!" and returns 10.

Interfering effects

```
RULE mapMap (f g : Int -> Int)
     (xs : List Int).
  map f (map g xs) = map (f.g) xs

let acc = 0
let g   = (\x. updateInt acc (addInt acc x);
               x)
let f   = (\x. addInt acc x)
    map f (map g [1,2,3])
```

Interfering effects - evaluated

```
let acc = 0
let g   = (\x. updateInt acc (addInt acc x);
               x)
let f   = (\x. addInt acc x)
    map f (map g [1,2,3])
```

Before rewrite

```
    map f (map g [1,2,3])    = [7, 8, 9]
```

After rewrite: bad bad bad

```
    map (f.g)    [1,2,3]     = [2, 5, 9]
```

Effects

```
addInt :: [r1 r2 r3 : %].
     Int r1 -> Int r2
   -(Read r1 + Read r2 + Alloc r3)>
     Int r3

updateInt :: [r1 r2 : %].
     Mutable r1 =>
     Int r1 -> Int r2
   -(Read r2 + Write r1)>
     Unit
```

Interfering effects - with effect types

```
RULE mapMap (f g : Int -> Int)
     (xs : List Int).
   map f (map g xs) = map (f.g) xs

let acc : Int
        = 0
let g   : Int -(Read r1 + Write r1)> Int
        = (\x. updateInt acc (addInt acc x);
               x)
let f   : Int -(Read r1)> Int
        = (\x. addInt acc x)
    map f (map g [1,2,3])
```

Interfering effects - new rule with effect constraint

```
RULE mapMap
     [ef eg : !]
     (f : Int -(ef)> Int)
     (g : Int -(eg)> Int)
     (xs : List Int).
     Disjoint ef eg =>
  map f (map g xs) = map (f.g) xs
```

Disjoint

Two reads are safe

```
Disjoint (Read r1) (Read r2)
```

Read and write are only safe if the regions are distinct

```
Distinct r1 r2 =>
    Disjoint (Read r1) (Write r2)
```

Allocation is always safe

```
Disjoint (Alloc r1) e
```

Disjoint

Effect sums are safe if all their elements are safe

```
Disjoint e11 e2 =>
Disjoint e12 e2 =>
    Disjoint (e11+e12) e2
```

Commutative

```
Disjoint e1 e2 =>
    Disjoint e2 e1
```

Inline CONLIKE

```
RULE unboxBox ...
     unbox (box i) = i
```

```
let x = box 52
    y = foo (unbox x)
    z = bar (unbox x)
in  y + z
```

Inline CONLIKE / let-holes

```
RULE unboxBox ...
     unbox (box i) = i
RULE unboxBox' ...
     Const r =>
     unbox {box i} = i

let x = box 52
    y = foo (unbox x)
    z = bar (unbox x)
in  y + z
```

Done

thank you, happy nice day

Inlining conflicts

```
let sum = foldr (+) 0

RULE sumMapMul ...
     sum (map (*2) xs) = (sum xs) * 2

sum (map (*2) [1,2])
==>
foldr (+) 0 (map (*2) [1,2])
|||
(sum [1,2]) * 2
```

Inlining conflicts - phases

```
let sum = foldr (+) 0
INLINE 2 sum
RULE sumMapMul ...
     sum (map (*2) xs) = (sum xs) * 2

sum (map (*2) [1,2])
==>
(sum [1,2]) * 2
==>
(foldr (+) 0 [1,2]) * 2
```

Foldr/build fusion - build definition

```
build [a : *] [r : %] [eg1 eg2 : !] [cg1 cg2 : $]
    (g : [b : *]. [ec1 ec2 en : !]. [cc1 cc2 cn : $].
         (a -(ec1|cc1)> b -(ec2|cc2)> b)
        -(eg1|cg1)>
         (Unit -(en|cn)> b)
        -(eg2+ec1+ec2+en|cg2+cc1)>
         b)
     eg1+eg2 + Alloc r | Use r
    : List r a
 = g [List r a] [!0] [Alloc r]
     [Alloc r] [:$0 (DeepUse a) $0:]
     (Cons [:r a:]) (Nil [:r a:])
```

Foldr/build fusion - first rule

```
RULE foldrBuild
     [a b : *] [r : %]
     [ek1 ek2 ez eg1 eg2 : !]
     [ck1 ck2 cz cg1 cg2 : $]
     (k : a -(ek1|ck1)> b -(ek2|ck2)> b)
     (z : Unit -(ez|cz)> b)
     (g : [gb : *]. [ec1 ec2 en : !]. [cc1 cc2 cn : $].
          (a -(ec1|cc1)> gb -(ec2|cc2)> gb)
             -(eg1|cg1)>
          (Unit -(en|cn)> gb)
             -(eg2+ec1+ec2+en|cg2+cc1)>
          gb).
     Disjoint (ek1+ek2+ez) (eg1+eg2) =>
  foldr [:a b r ek1 ek2 ez ck1 ck2 cz:]
     k z
     (build [:eg1 eg2 cg1 cg2:] g)
= g [:b ek1 ek2 ez ck1 ck2 cz:] k z
```

Foldr/build fusion - phased for inlining

```
foldr_build [...] k z g
 =  foldr [...] k z (build [...] g)

RULE foldrBuild_unconditional
     ...
     (no constraints)
 foldr k z (build g) = foldr_build k z g

RULE foldrBuild_fuse
     ...
     Disjoint (ek1+ek2+ez) (eg1+eg2) =>
 foldr_build k z g = g k z
```