

Notes on *Warm Fusion* – Part II

We have seen that the warm fusion process (or, more accurately, the first of the ways to proceed mentioned in Section 6 of [LS95]) comprises several distinct steps:

1. Introduce a build-cata pair as described by the syntax-to-syntax translation B.
2. Distribute the cata over the outermost case statement in the body of the function to be transformed. (In all of the examples we've seen, and in my WFlang implementation of warm fusion, the bodies of the functions to be transformed consist entirely of single — perhaps complicated — case statements.)
3. Reduce the “distributed” function body using the basic rules from Figure 3.
4. Split the reduced, “distributed”, program into its worker and wrapper.
5. Unfold the program's wrapper in its worker. The resulting recursive worker can then be reduced using the basic rules from Figure 3 (including the short cut).
6. Inline in the recursive worker all called functions for which we already have build-cata forms. This new recursive worker can then be reduced using the basic rules from Figure 3.
7. Compose, on the right, the inlined worker with the appropriate copy function.
8. Fuse this composition and reduce the result via the basic rules to get a catamorphic representation of the worker. This involves computing appropriate “dynamic rewrite rules” for each data constructor to compute its replacement function according to the promotion theorem.
9. Inline the catamorphic worker in the wrapper to obtain a **build-cata** form for the original input program.

Note that after Step 6 we can split the worker again to optimize for static constructors. This technique is illustrated in both examples below. Recognition of constructors as static (when they are) happens “automagically” in [LS95].

The following examples illustrate the first warm fusion method from [LS95] from start to finish, including optimization for static parameters.

Example 1: Reverse

The function `reverse` itself does not have any static parameters. We will see, however, that `reverse`'s worker has a static data constructor argument. The ability to automatically recognize static constructor arguments allows warm

fusion to transform the standard quadratic definition of `reverse` into a linear definition. To facilitate this optimization, we curry the arguments to `build` below.

We begin with the usual definition of `reverse`:

```
reverse = \x. case x of
             Nil()      -> Nil()
             Cons(z,zs) -> append(reverse zs)(Cons z,Nil())
```

Introduction of the appropriate `cata-build` pair gives:

```
reverse = \x. build (\n c. cata (n,c)
                       case x of
                         Nil()      -> Nil()
                         Cons(z,zs) -> append(reverse zs)(Cons z,Nil()))
```

Distributing the `cata` over the case yields:

```
reverse = \x. build (\n c.
                       case x of
                         Nil()      -> cata (n,c) (Nil())
                         Cons(z,zs) -> cata (n,c) (append(reverse zs)(Cons z,Nil())))
```

The body of this definition simplifies, via the basic rewrite rules, to

```
reverse = \x. build (\n c.
                       case x of
                         Nil()      -> n()
                         Cons(z,zs) -> cata (n,c) (append(reverse zs)(Cons z,Nil())))
```

The wrapper of `reverse` (also written `reverse`) is

```
reverse = \x. build (\n c. reverse# x n c)
```

The worker of `reverse` is

```
reverse# = \x n c.
           case x of
             Nil()      -> n()
             Cons(z,zs) -> cata (n,c) (append(reverse zs)(Cons z,Nil())))
```

Inlining the wrapper in the worker, inlining the `build-cata` form of `append`, and reducing the result gives a new version of the wrapper:

```
reverse# = \x n c.
           case x of
             Nil()      -> n()
             Cons(z,zs) -> reverse# zs (\().c (z,n())) c
```

We now observe that the `Cons` replacement function `c` remains static in the recursive call in `reverse#`. If we convert `reverse#` into a catamorphism now, then `c` will be passed as an argument to the replacement functions for both `Nil` and `Cons`. But this is not necessary!

It is better to recognize `c` as a static argument to `reverse#` from the outset and, thereby, to avoid incorporating it into the resulting catamorphism. This is accomplished via a worker-wrapper split of the worker itself! This gives the following wrapper for the worker (denoted `reverse#`):

```
reverse# = \x n c. reverse## x n
```

together with the following worker of the worker:

```
reverse## = \x n.
  case x of
    Nil()      -> n()
    Cons(z,zs) -> reverse# zs (\(). c(z,n())) c
```

Unfolding the worker's wrapper in its worker and reducing gives:

```
reverse## = \x n.
  case x of
    Nil()      -> n()
    Cons(z,zs) -> reverse## zs (\(). c(z,n()))
```

This is the function we want to convert into a catamorphism.

Conversion of the worker's worker into a catamorphism is accomplished in the usual way. We first compute the “dynamic rewrite rules” associated with each constructor, and then use them to compute the constructor replacement functions for the catamorphism we are trying to derive. Writing “+” for set union, but otherwise in the notation of [LS95], we have

```
R_Nil = R_0
R_Cons = R_0 + {y_1 -> z_1, y_2 -> reverse## z_2}
```

We use these to rewrite the expressions

```
\(). reverse## (Nil())
```

and

```
\(z_1,z_2). reverse## (Cons(y_1,y_2))
```

to the constructor replacement functions `h_Nil` and `h_Cons` for catamorphism we're deriving. This process yields:

```
h_Nil = \(). \n. n()
h_Cons = \((z_1,z_2). \n. z_2 (\(). c(z_1, n()))
```

and so the catamorphic form of the worker's worker is

```
reverse## = cata (\().\n.n,
                 \z_1,z_2).\n. z_2 (\().c(z_1, n()))
```

Inlining the worker's worker into the worker's wrapper results in the following new worker for `reverse`:

```
reverse# = \x n c. cata (\().\n.n,
                       \z_1,z_2).\n. z_2 (\().c(z_1, n()))
           x n
```

Inlining the worker for `reverse` in `reverse`'s wrapper produces the final build-cata form for `reverse`. It is

```
reverse = \x. build
(\n c. cata (\().\n.n,
            \z_1,z_2).\n. z_2 (\().c(z_1, n()))
 x n)
```

Example 2: Map

We begin with the usual definition of `map`:

```
map = \f x. case x of
          Nil()      -> Nil()
          Cons(z,zs) -> Cons (f z) (map f zs)
```

Introduction of the appropriate cata-build pair gives:

```
map = \f x. build (\n c. cata (n,c)
                             case x of
                               Nil()      -> Nil()
                               Cons(z,zs) -> Cons (f z) (map f zs))
```

Distributing the cata over the case yields:

```
map = \f x. build (\n c.
                  case x of
                    Nil()      -> cata (n,c) (Nil())
                    Cons(z,zs) -> cata (n,c) (Cons (f z) (map f zs)))
```

The body of this definition simplified, via the basic rewrite rules, to

```
map = \f x. build (\n c.
                  case x of
                    Nil()      -> n()
                    Cons(z,zs) -> c (f z) (cata (n,c) (map f zs)))
```

Since the function argument f to `map` is static — i.e., does not change in the recursive call to `map` — it is not passed to `map`'s worker. As a result, the wrapper of `map` (also written `map`) is

```
map = \f x. build (\n c. map# x n c)
```

and the worker of `map` is

```
map# = \x n c.
      case x of
        Nil()      -> n()
        Cons(z,zs) -> c (f z) (cata (n,c) (map f zs))
```

Inlining the wrapper in the worker and reducing the result gives a new version of the wrapper:

```
map# = \x n c.
      case x of
        Nil()      -> n()
        Cons(z,zs) -> c (f z) (map# zs n c)
```

We now observe that all of the arguments to `map` other than the first are static parameters of `map#`. Taking this into account when splitting the worker itself into a worker and a wrapper gives:

```
map# = \x n c. map## x
```

together with the following worker of the worker:

```
map## = \x. case x of
           Nil()      -> n()
           Cons(z,zs) -> c (f z) (map# zs n c)
```

Unfolding the worker's wrapper in its worker and reducing produces:

```
map## = \x. case x of
           Nil()      -> n()
           Cons(z,zs) -> c (f z) (map## zs)
```

This is the function we want to convert into a catamorphism.

As before, to convert the worker's worker into a catamorphism we first compute the “dynamic rewrite rules” associated with each constructor, and then use them to compute the constructor replacement functions for the catamorphism we are trying to derive. Writing “+” for set union, but otherwise in the notation of [LS95], we have

```
R_Nil = R_0
R_Cons = R_0 + {y_1 -> z_1, y_2 -> map## z_2}
```

We use these to rewrite the expressions

```
\(). map## (Nil())
```

and

```
\(z_1,z_2). map## (Cons(y_1,y_2))
```

to the constructor replacement functions `h_Nil` and `h_Cons` for `cata` we're deriving. This process yields:

```
h_Nil = \(). n() (i.e., h_Nil = n)
h_Cons = \(z_1,z_2). c(f z_1, z_2)
```

and so the catamorphic form of the worker's worker is

```
map## = cata (n, \(z_1,z_2). c(f z_1, z_2))
```

Inlining the worker's worker into the worker's wrapper results in the following new worker for `map`:

```
map# = \x n c. cata (n, \(z_1,z_2). c(f z_1, z_2)) x
```

Inlining the worker for `map` in `map`'s wrapper produces the final `build-cata` form for `map`. It is

```
map = \f x. build (\n c. cata (n, \(z_1,z_2). c(f z_1, z_2)) x)
```