

Notes on *Warm Fusion* – Part I

[Note carefully the difference between a datatype constructor — such as `List` or `Tree` — a type constructor — such as $\lambda a. \lambda b. () + (a, b)$ or $\lambda a. \lambda b. () + (b, a, b)$ — and a data constructor — such as `Nil` or `Cons`.]

Warm Fusion is a technique for automatically transforming recursive data structure-producing and -consuming programs into build-cata forms so that the *shortcut to deforestation* — i.e., the *cata-build rule* — can be used to remove intermediate data structures from them. Warm fusion makes use of two program transformation techniques:

1. the shortcut itself, as discussed in the paper of Gill, Launchbury, and Peyton Jones, and
2. *cata* fusion, which is based on the promotion theorems, which in turn have their basis in category theory.

Warm fusion is not limited to the transformation of list-processing functions. Nevertheless, such functions are prevalent in functional programming, and so are of particular interest to us.

These notes elaborate some of the main ideas in the first three sections of the 1995 paper *Warm Fusion: Deriving Build-Catas from Recursive Definitions* by Launchbury and Sheard (henceforth referred to as [LS95]) in which warm fusion was introduced.

The two functions `cata` and `build` form the cornerstone of the method. Informally speaking, `build` takes as input a constructed type — e.g. `List Int` or `Tree a` — indicating the type of its output, and a polymorphic function `g` which returns an ‘abstracted’ element of that datatype, and returns an element of the constructed type that was input. Thus,

```
build [List Int] g = g [List Int] (c_1, ..., c_n)
```

where `c_1, ..., c_n` are the data constructors of the datatype `List`.

Just as `build` captures the uniform production of data structures, `cata` captures uniform data structure consumption. Informally speaking, `cata` takes as input a constructed type indicating the type of its input, a type indicating the type of its output, a sequence of appropriately typed replacement functions for the data constructors associated with the datatype constructor in the input constructed type, and an element `e` of that constructed type. Then `cata` ‘walks over’ the structure `e`, appropriately replacing its data constructor occurrences with the replacement functions. This requires recursively applying the `cata` to the ‘recursive positions’ associated with each data constructor; exactly which positions are the ‘recursive positions’ for each data constructor is coded up in the functions `curly_E` defined on p. 315 of [LS95].

To understand how the `curly_E`’s work, we first note that every datatype constructor `T` has an associated type constructor E^T . This datatype constructor can be read directly from the definition of the type constructor. For example, the datatype constructor

`List = /\a. Rec b. Nil () + Cons (a,b)`

has associated type constructor

`E^List = /\a. /\b. () + (a,b),`

and the datatype constructor

`Tree = /\a. Rec b. Tip () + Node (b,a,b)`

has associated type constructor

`E^Tree = /\a. /\b. () + (b,a,b).`

The type constructor `E^List` is exactly the type constructor obtained in the second set of displayed equations on p. 315 of [LS95], provided

1. `E_Nil` and `E_Cons` are inlined,
2. tuples `a_1 x ... x a_n` are written `(a_1,...,a_n)`,
3. the expression resulting from inlining is beta-reduced, and
4. positional, rather than tagged, sums are used.

In the type constructor associated with a datatype constructor, the occurrences of `b` (in general, the `Rec`-bound variable in the datatype constructor definition) indicate the ‘recursive positions’ for each data constructor of `T`. For the datatype constructor `List`, there is only one ‘recursive position’ and it is indicated by the `b` in the type `(a,b)` associated with `Cons`. For the datatype constructor `Tree` there are two ‘recursive positions’; these are indicated by the two occurrences of `b` in the type `(b,a,b)` associated with the data constructor `Node`.

The ‘recursive positions’ associated with the data constructors for a datatype constructor are important because, as indicated above, they determine the precise manner in which the catamorphism associated with that datatype constructor walks down a structure whose type is an application of that datatype constructor to a sequence of types. If `C` is a data constructor associated with a datatype constructor `T`, `t = (z_1, ..., z_n)` is the type associated with `C`, `g` is a function on expressions, and `xs` is a tuple of expression arguments to `C`, then `E_C (g) = curlyE_g [t]` applies `g` to the expressions at all of the recursive positions of the tuple `xs` (and all the recursive positions within them). The process for doing this is described in the definition of `curlyE_g` on p. 315 of [LS95] — see especially the last clause of the definition of `curlyE_g`, which we did not discuss in reading group. We can think of the `curlyE_g`’s as “distribution functions” for the various term transforming functions `g`.

In [LS95], the polymorphic versions of `build` and `cata` just discussed are *not* used. Instead, instances of `cata` and `build` are defined for each datatype constructor. Thus, for the datatypes declared on p. 315 of the paper, we have `cata^List`, `cata^Tree`, `build^List`, and `build^Tree`. Since `cata` and `build`

are decorated only with the datatype constructor in the notation of the paper, the actual constructed type associated with the input to `cata` or the output to `build` must be inferred from context; this differs from the descriptions of `build` and `cata` given above, in which this information is explicitly supplied. In fact, much type information is suppressed in [LS95]. Since the entire warm fusion method is type-directed (as we'll see), the type information is critical to understanding and implementing the algorithm. Olaf Chitil used this important observation to develop a technique which achieves the effect of warm fusion using only type inference.

With full type information, the `cata-build` rule for a program fragment consuming lists whose elements are of type `a` and producing a value of type `c` is

```
cata [List a] [c] (f_1, f_2) (build [List a] g) =  
  g [List a] (f_1, f_2)
```

It is worth noting that the data constructors associated with datatype constructors are also polymorphic. Thus,

```
Nil :: \a. () -> List a  
Cons :: \a. (a, List a) -> List a
```

and

```
Tip :: \a. () -> Tree a  
Node :: \a. (Tree a, a, Tree a) -> Tree a
```