Function Compressor

by Sven Nilsen, 2017

A function compressor is a function that takes a list of partial function pairs and produces an efficient function. It is used to produce code for paths that are derived by exhausting all input and compute the output. It can also be used to derive existential paths for various domain constraints.

$$\mathrm{fcr} : [(A, B)] \rightarrow (A \rightarrow B)$$

Notice that a function compressor does not optimize a function. A function optimizer would take a function and return another function of equal or better performance, but the function compressor takes raw data and constructs a function from it. This makes function compressors different from optimizers because they do not require knowledge about how to traverse the AST of a function. An ideal function compressor would return a function that can not be optimized further.

All function compressors are logically equivalent, so they can be reasoned about using the same name `fcr`. What makes function compressors different from each other are intentional paths. The embodiment `a` of a function compressor `fcr` is written `$\mathrm{fcr}^a$`. For example, the following expresses some data `x` that can be compressed to `f`. This makes it possible to reason about the data about a function.

$$x : [\mathrm{fcr}]\ f$$

The intentional path of `fcr` by embodiment `a` is:

$$\mathrm{fcr}^a : [(A, B)] \rightarrow C$$

For all deterministic function compressors, there exists a function (the intentional path) which predicts the cost `C` from the data about the function. If the function compressor is ideal, then the intentional path predicts optimized behavior.

In practice it is very hard to create a good function compressor. This depends on:

- Which functions that are available in the executable environment
- How performance of equivalent functions vary as intentional paths
- How patterns are detected
- Knowledge about how to generate code
- What cases to optimize for

As a guiding principle, one can think of the function compressor as the inverse of evaluating a function for all input.

$$\mathrm{eval\_all} : (A \rightarrow B) \rightarrow [(A, B)]$$

$$\mathrm{eval\_all}^{-1} \Leftrightarrow \mathrm{fcr}$$

An ideal function compressor can not be found by using the `$P^{\exists f\{\}} = NP^{\forall f}$` algorithm, because it would only be the fastest executing function compressor, rather than outputting functions that run with minimized performance cost. This means that an algorithm that outputs an optimal function compressor for a specific problem has even higher complexity than a “perfect intelligent” solver for a problem encoded as a specific function. The higher complexity comes from the requirement of using the intentional path for optimization. For now, it is an open problem of how to construct an ideal function compressor.

A more practical approach would be to create a function compressor that outputs a finite set of functions. Each function can be carefully crafted and optimized. Using the trick from the `$P^{\exists f\{\}} = NP^{\forall f}$` algorithm, one can create a binary search tree such that the lookup is `O(log N)` for `N` number of functions. This is done by organizing sub-types of the data such that there is a 50% chance of half of the functions matching a check for each step. One way to approximate this behavior is to check one bit at a time of the input, then check one bit of the output. Since the output bit depends on the input bit, it means the leafs are stored redundantly in the tree. For example, if the input bit is `0`, then the output might be `x`, and if the input bit is `1`, the output might be `y` for the same function. One would also need to balance this with which input to check.