# Cost Model Experimentation

Andrew Tran

July 2018

## 1 Introduction

During synthesis, Cozy uses a symbolic cost model to determine the costs of two expressions given to it. It then makes several calls to a solver to determine the order between the costs, that is, which one is better.

There are currently three versions of the cost model: the original cost model, the maintenance cost model (both of which are on the main branch of the repository), and most recently, the frequency cost model (on the query-importance branch). The original cost model refers to the cost model that Cozy currently uses for running benchmarks. The latter two are modifications of the original model. When evaluated on the benchmarks, the maintenance and frequency models are slightly worse than the original model. This is because both the maintenance and frequency models are slower than the original model due to optimizing with respect to more parameters.

## 2 The Original Cost Model

### 2.1 The Problem

There are special heuristics that Cozy follows to prevent unwanted behaviours, one of which prevents Cozy from storing integer arithmetic as state variables. Once this heuristic is removed, Cozy will start producing unwanted state variables. Specifically, when the heuristic is removed and Cozy is provided with this specification:

```
Incrementer:
    state x : Int

    query get()
        x

    op increment()
        x = x + 1;
```

It will continuously store each subsequent increment of `x` as state variables. As Cozy runs for longer and longer, the stored states will grow continuously, adding on `(x + 1)`, `((x + 1) + 1)`, `(((x + 1) + 1) +1)`, and so forth.

When the heuristic is not in place, the cost model will tell Cozy that storing each increment of `x` as a state is better than just storing `x` itself. To prevent this behaviour and also remove the necessity of the heuristic, we introduced freebies to the cost model. Freebies are state variables that will have no cost. With them in place, Cozy would choose to store just `x` instead of its increments as states.

## 2.2 Overview

The original model uses prioritized ordering between values of expressions. The ordering tells the cost model which value it cares about: most to least. From first to last, the cost model will compare the asymptotic runtimes, the max storage sizes, the true runtimes, and the sizes of two expressions. The prioritized ordering ensures that values lower in the priority list will only be calculated to break ties from the more important ones.

Freebies are stored in a list that will be used to calculate the storage size of expressions, which will affect their max storage sizes.

## 2.3 Challenges

The challenge is how to choose which state variables to become freebies. One option is to just use state variables that are also hints taken from values of the concretization functions. However, since the hints contain states generated by every query, they introduce a problem: Cozy might change its mind on a query and decide to remove it, in which case the states associated with it would be removed as well. This will leave states that Cozy does not use but the cost model still considers to be free. This may result in Cozy generating poor implementations that was not supposed to be there in the first place.

So instead of using hints, the freebies are chosen from a list of state variables that are maintained by the query currently being improved. By choosing the freebies this way, we make sure that the state variable is only free for subqueries derived from the current query and not for any other queries.

## 3 The Maintenance Cost Model

### 3.1 The Problem

With the original cost model, there is a chance Cozy will output implementations in which a state variable is highly efficient when a query is run on it, but is inefficient when a mutator is run. For example, given this specification:

```
Cats:
    state cats: Set<Cat>

    query countSharedColor(c : Cat)
        assume c in cats;
        len [x | x <- cats, x.colour == c.colour]

    op addCat(c : Cat)
        cats.add(c);
```

Cozy will try to output an implementation that is a map from cats to the number of other cats that have the same colour. This implementation, when being considered for just the query, is very efficient. However, when the mutator `addCat` is called, the map of cats will have to be updated in two ways: (1) every existing cat with the same colour as the new cat has to increment their count by 1, (2) the new cat will have to calculate the number of existing cats that match its colour. When the number of cats is very large, it can lead to inefficiency.

By allowing the cost model to also consider the maintenance cost of expressions with respect to available mutators, Cozy should be able to determine that storing a map from colours to cats with that colour is much more efficient overall.

## 3.2 Overview

With the original cost model, there were no connections between queries and mutators. This means that when it is optimizing a query, it is doing so regardless of how that would affect the performance of the mutators. The maintenance cost model is provided with information on what mutators (or `ops`) that exists in the specification. It will then calculate the cost to maintain expressions when there are mutators to be considered.

The maintenance cost of an expression is calculated by changing the original expression with the mutator, taking the difference between the original and the changed expressions, and calculating the size of that difference. This also means that if there are no differences, the maintenance cost is zero.

## 3.3 Challenges

With the cost model being allowed to consider mutators as well as the query itself, we have to try to optimize for multiple objectives: the queries being efficient as well as the mutators. Initially, the model replaced max storage size and true runtime with the sum of the maintenance costs of an expression for each mutator. Then, it compares the sums calculated from each expression with each other as the tiebreaker for the asymptotic runtime in the prioritized ordering.

The problem with this is for cases where there is more than one mutator, for each expression, there's a chance that it has a different maintenance cost depending on the mutator that was used on it. This can lead to very confusing results, as we don't know which maintenance cost is more important, and summing them doesn't properly convey that information.

To tackle this problem, we added an unprioritized ordering to the cost model. Opposite to the prioritized ordering, this ordering is used when we don't know which metric is more important than another. The unprioritized order compares the maintenance costs of two expressions for each op separately, then check the comparisons to make sure that they are consistent or ambiguous. This change allowed the cost model to more properly convey that different mutators can give different maintenance costs to expressions.

However, with the addition of the maintenance cost, the cost model has become better at picking out ambiguity between two expressions. An example is for bag storage, Cozy will be able to find examples where the maintenance cost of an expression is different depending on what is currently in the bag and which mutator is being considered.

Furthermore, because Cozy is comparing the maintenance costs of the expressions for each mutator separately, there is a significant increase in solver calls. This has made the maintenance cost model much slower than the original, thus preventing Cozy from finding some solutions that the original could given the same amount of time.

## 4 The Frequency Cost Model

### 4.1 The Problem

In both the original model and the maintenance cost model, there was no way for the user to specify the importance (or frequency of usage) of a query or a mutator, this could be useful for specifications that could be implemented in several different ways, each way more efficient for different queries or mutators. For example, this specification:

```
Product:
    state xs : Bag<Int>
    state ys : Bag<Int>

    query product()
        [(x, y) | x <- xs, y <- ys]

    op add_x(x : Int)
        xs.add(x);
```

Can produce a state that is a list of all `(x, y)` tuples. This solution is inefficient for the mutator but efficient for the query. When an `x` is added, the list of tuples will have to grow in size by the cardinality of the bag `ys`, which is problematic if `ys` gets really big. In the original and the maintenance cost model, there is a heuristic in place to prevent Cozy from outputting this solution. However, this solution is more efficient if the query is used significantly more than the mutator. If we allow the user to specify this information, not only will Cozy have one less heuristic, it will also produce implementations that are more tailored to how they will be used.

## 4.2 Overview

The frequency cost model uses part of the maintenance cost model but replaces the unprioritized ordering with a prioritized ordering of frequency cost, asymptotic runtime, max storage, true runtime, and size.

The frequency cost of an expression is the weighted sum of its true runtime and its maintenance cost for every mutator that is available. The true runtime is considered to be the runtime of the query. The weights are the frequency of the query and mutators.

## 4.3 Challenges

There are a few noticeable challenges with this model. The first is since this model is optimizing for multiple objectives, there might be an issue with just taking the weighted sum of runtime and maintenance costs. This could possibly produce confusing results in the future, especially for more complicated specifications.

Moreover, since the model is using parts of the maintenance cost model, it is also slower than the original cost model. However, since it makes fewer solver calls, it is slightly faster than the maintenance cost model.

The final issue that this model faces is the scale of the frequencies. Currently, there are no specific guidelines on how to use the frequencies, or what numbers the frequencies should be. Manual testing on the example above shows that if the query has a frequency that is at least $\times 25$ the frequency of the mutator, Cozy will produce a different implementation. However, it is unclear if the scale is the same for other specifications, especially those that more complicated.