Proposal: Fused Conditional Merge

1 Synopsis

Currently the body of a conditional statement in the following manner

1. Gather all partitions related to the loop invariant
2. Resolve the guarded partition
3. Split the partition involving the guarded partition into two where one half contains the sub-*states* that validates the guard and the other one that nullify the guard.
4. Run compilation twice on two halves
5. Discard duplicated yet non-entangled partitons and merge those entangled ones.
6. Do some house cleaning work ….

Following this approach, we can account for changes made by type cast and state split without extra works. However, a subtle nuisance is that works on non-entangled partitions are duplicated, because of which the solver seems to have trouble in crushing VCs.

Informally,

```
(define (compile `(for guard body))
  (merge
   (compile (filter-guard #t body))
   (compile (filter-guard #f body))))
```

2 Proposal

In order to avoid duplicated work, I propose an approach that compile the body in one pass (instead of two) and performs the split and merge process in each `:*=:` instead. This also requires tracing guarded partition through the compilation of the body.

2.1 Implementation Perspectives

Implementation-wise, the bookkeeping can be either done by argumenting current `EmitState` in `TState` so that each `RBinding` points to the emitted variables tagged with the guard expression/condition that indicates in which case the each variable will be used.

This enables an optimization we can exploit.

Currently, I implement the most general strategy in handling **`for`** statements by splitting the guarded partition into exactly two parts and merging them in the end because there could be a change in the number of kets under some interesting operations. This also requires the knowledge on the accurate index to be used for the state split.

With the new implementation, since we lift the split-and-merge to each application statement, we can use a `Map`-based and index-agnostic split-and-merge if the application statement guarantees that no new terms will be introduced here, which is guaranteed by general λ functions.

Informally,

```
(define (compile `(for guard body))
  (with-guarded-partitions guard
    (lift/statements compile body)))
```