Resource User's Guide

David Legg

October 12, 2023

Contents

1 Background

1.1 Cells and Effects

Aerie handles the concept of mutable state in a simulation through "cells". A "cell" is a single piece of mutable state, managed by Aerie. For a simulation to be correct, mutable state which persists outside of local scope should usually be put in a cell. Constants or local variables usually do not need to be put in a cell. Cells change when time passes, or when tasks emit an effect. The notion of time passing in a simulation is captured by "stepping" the cell. For example, a cell might have the value 4, which is growing at a rate of 2 per second. When we step the cell by 3 seconds, it would then have a value of 10. An effect describes a change on a cell, like "set the value ON," "append 42 to the list," or "double the value." Aerie tracks what time a cell is "at," which may lag behind the simulation time, and which effects have been applied. When the cell value is requested, Aerie steps up the cell in time and applies the effects at appropriate times.

1.2 Jobs and Concurrency

Everything that Aerie does is a "job". There are three kinds of jobs: tasks, conditions, and resource sampling.

Tasks are the only kind of job that's allowed to emit effects. Tasks include activities placed by the user, daemons spawned by the model itself, and child tasks spawned by either of these.

Conditions detect an interesting situation in the model, used to schedule a task. For example, a daemon task might want to run when the power resource drops below 5. We would construct a condition encoding this situation, and ask Aerie to run the task when that condition is satisfied. Aerie evaluates a condition once when it's initially added. After that, Aerie re-evaluates a condition when there's an effect on a cell queried by the condition the last time it was evaluated.

Resource sampling is a job that Aerie performs automatically, which collects updates to registered resources. All resources are sampled when the simulation starts. Like conditions, Aerie queues a resource to be re-sampled when there's an effect on a cell queried by that resource the last time it was sampled.

All jobs are associated with a time, the next time that job needs to be run. All jobs are put into a priority queue, sorted first by this time, and then by type: conditions before tasks before resource sampling. Aerie pulls the highest-priority batch of jobs, which will all be of one type, and runs them "concurrently." The simulation clock is always set to the start time of this batch of jobs. When a job runs, it can insert new jobs, for example a condition triggering a task, a task scheduling another task, or a task setting a condition. For this reason, it's possible for Aerie to cycle many times without advancing the simulation clock. Finally, note that because resource sampling happens after all tasks at any particular simulation time, if there are multiple effects at the same simulation time, only the final result is observed in the resource profile. Figure 1 shows an example of some jobs, of all three types, running.

Aerie handles concurrency explicitly. Tasks pulled off the job queue in the same batch are concurrent. Each task is started on a separate branch, and querying a cell will observe only effects in that branch's history. Additionally, if a task spawns a child task, this creates a new branch for the child at that point. Once all the tasks yield the branches are joined so that the next batch of jobs will observe all effects from all jobs in this batch.

To explain this visually, consult fig. 2. Imagine a cell X which holds an integer value, and supports effects which increment or decrement the value. We'll imagine Tasks 1 and 2 are started in the same batch, and observe the effects each one emits and the values of X they would observe. Notice that when all tasks in this batch and their children yield, their branches merge, and the observed value is the combined result of all effects.

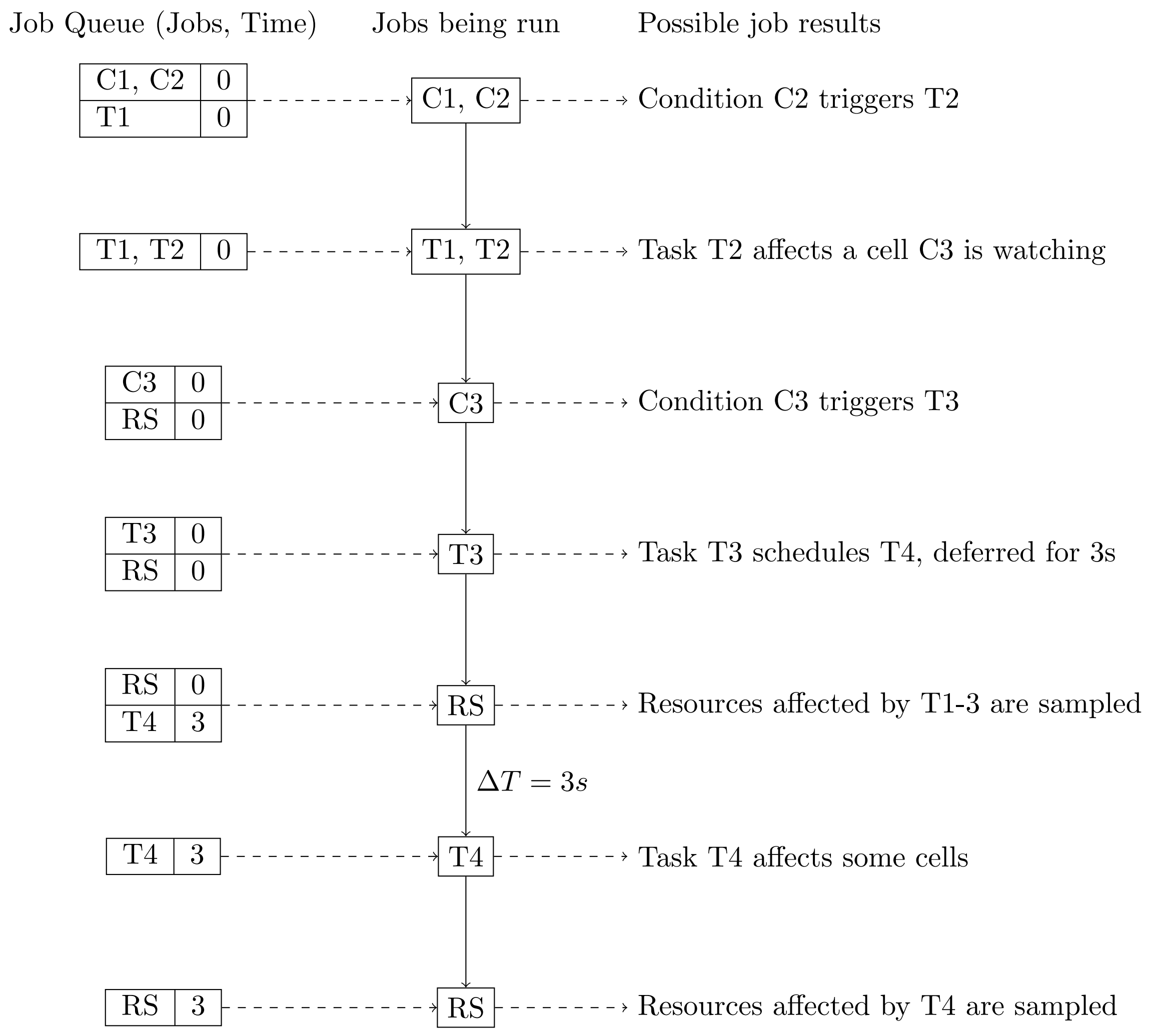
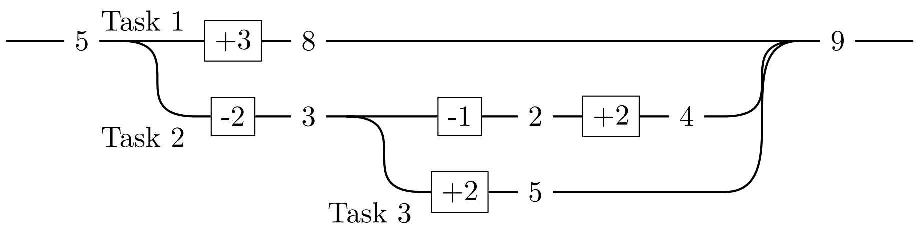


Figure 1: Example of Aerie's job queue



```
// Task 1:
X.emit(+3);
// Explicit yield:
delay(0 seconds);
...
```

```
// Task 2:
X.emit(-2);
spawn(Task 3);
X.emit(-1);
X.emit(+2);
// Yield by ending
```

```
// Task 3:
X.emit(+2);
// Explicit yield:
delayUntil(cond);
...
```

Figure 2: Example of effects applied to a cell

In this example, effect order doesn't matter, so the combined effect is unambiguous: Apply all effects from all branches to the starting value, to get $5 + (3 - 2 - 1 + 2 + 2) = 9$. In general, Aerie uses an "effect trait," supplied when constructing the cell, to define how to combine effects. For reasons we'll see later, most modelers will never need to specify an effect trait directly.

1.3 Monads

Monads form the backbone of the resource framework. For a more thorough discussion, see the excellent post "Monads: Programmer's Definition" by Bartosz Milewski.[1] I'm drawing heavily from it here.

To motivate this definition, observe that we often want to "decorate" a value in a kind of transparent way. That is, we want to augment a type, but mostly operate on it without explicitly dealing with the augmentation. For example, Java's `Optional` adds the idea of a value potentially not being there, then we use `Optional.map` to operate on it without explicitly handling the empty case. For example, suppose we might have a number x, which we'd like to add one to. We could write:

```
Double x = getX();
Double xPlus1 = (x == null ? null : x + 1);
```

or, using `Optional` to handle the empty case,

```
Optional<Double> x = Optional.ofNullable(getX());
Optional<Double> xPlus1 = x.map(x$ -> x$ + 1);
```

[1]https://bartoszmilewski.com/2016/11/21/monads-programmers-definition/

Indeed, "map" is our first monadic operation. If MT denotes a type T decorated by monad M, then map uses a function $A \to B$ like a function $MA \to MB$. Sometimes, the operation we want to write introduces the decoration itself. Consider an implementation of `sqrt` which takes a `Double` and returns an `Optional<Double>`, returning empty when the input is negative. If we map over an `Optional<Double>` with this `sqrt`, we'd get an `Optional<Optional<Double>>`. That is,

```
Optional<Double> sqrt(Double input) {
  if (input < 0) return Optional.empty();
  else return Optional.of(Math.sqrt(input));
}

Optional<Double> input = Optional.of(4);
Optional<Optional<Double>> output = input.map(sqrt); // !!!
```

We can solve this by introducing a new monadic operation: "combine" (sometimes called "join"). This squashes two layers of monadic wrapping together: $M(MT) \to MT$. For `Optional`, we would return empty if either layer is empty, or the singly-wrapped value if it's present. Let's define and use that to fix our example from before:

```
<T> Optional<T> combine(Optional<Optional<T>> x) {
  if (x.isEmpty()) return Optional.empty();
  else return x.get();
}

Optional<Double> input = Optional.of(4);
Optional<Double> output = combine(input.map(sqrt));
```

The map-and-then-combine pattern is so common, it gets its own name: "bind." Bind uses a "diagonal" function $A \to MB$ like a "horizontal" function $MA \to MB$. On `Optional`, this is `Optional.flatMap`. Rewriting our example above,

```
Optional<Double> input = Optional.of(4);
Optional<Double> output = input.flatMap(sqrt);
```

Finally, we need some way to create monadic values. While most monads will have their own special constructors, all monads come with one canonical constructor: "unit." Unit gives a value "default" decorations. For example, `Optional.of` is the monad unit. There are formal requirements for what unit does, but loosely, wrapping a value with unit and then unwrapping it should not change the value.

Figure 3 shows a schematic representation of these operations. The double arrows are meant to show bind "lifting" just the tail of the $A \to MB$ arrow, while map lifts both ends of the $A \to B$ arrow. Importantly, the above schematic "commutes," meaning if you trace out paths that go from the same source to the same destination by different routes,

$$\begin{tikzcd}
 & \text{M (M B)} \arrow[d, "\text{combine}"] \\
\text{M A} \arrow[r, ""{name=T}] & \text{M B} \\
\text{A} \arrow[r, ""'{name=Bo}] \arrow[ur, ""{name=D}] & \text{B} \arrow[u, "\text{unit}"']
\arrow[Rightarrow, from=D, to=T, bend left, "\text{bind}"]
\arrow[Rightarrow, from=Bo, to=T, "\text{map}"']
\end{tikzcd}$$

Figure 3: Schematic of monad operations

the result is the same value. For example, let $f : A \to MB$, $g : A \to B$, and $\hat{a} : MA$, then trace out the identities $\text{bind}\,\hat{a}\,f = \text{combine}\,(\text{map}\,\hat{a}\,f)$ and $\text{map}\,\hat{a}\,g = \text{bind}\,\hat{a}\,(\text{unit} \circ g)$.

A monad can be defined by unit, map, and combine, or by unit and bind. Since combine is not often a useful operation on its own, we take the unit-and-bind definition to be canonical, and define map in terms of unit and bind using the identity above. We also call out one pattern that comes up frequently. Suppose we have a multi-argument function on undecorated values $f : A \times B \times C \to D$. Then, we use "bind" and "map" to apply this to decorated values $\hat{a}, \hat{b}, \hat{c} : MA, MB, MC$ like so:

$$\text{bind } \hat{a} \ \left(a \mapsto \text{bind } \hat{b} \ (b \mapsto \text{map } \hat{c} \ (c \mapsto f(a, b, c)))\right)$$

This example might help to explain the name "bind" - it's like we're binding the undecorated value out of $\hat{a}$ to the lambda parameter a.

2 Resource Monad Stack

When deriving a resource from one or more other resources, we need to think about the resource "monad stack." This is a collection of types that wrap the dynamics type to create the resource type. At the time of writing, there are three layers of wrapping:

1. `Expiring` - This adds an "expiry" to the dynamics, indicating when the dynamics can no longer be trusted as accurate. This is especially useful for continuous-to-discrete derivations, like comparing a continuous value to a threshold, since the result can expire sooner than the sources.

2. `ErrorCatching` - This is a "union" or "sum" type, between the expiring dynamics type and an error type. It allows us to indicate that a derivation failed without crashing the simulation.

$$\begin{tikzcd}
\texttt{Resource<D>} \\
\texttt{ErrorCatching<Expiring<D>>} \arrow[u, "\texttt{ErrorCatchingToResourceMonad}"'] \\
\texttt{Expiring<D>} \arrow[u, "\texttt{ErrorCatchingMonad}"'] \\
\text{Dynamics } \texttt{D} \arrow[u, "\texttt{ExpiringMonad}"'] \arrow[uuu, bend left=60, "\texttt{ResourceMonad}"] \arrow[uu, bend right=60, "\texttt{DynamicsMonad}"'] \\
\text{Value } \texttt{V} \arrow[u, dashed, "\texttt{DiscreteMonad}"'] \arrow[uuuu, dashed, bend left=70, "\texttt{DiscreteResourceMonad}"] \arrow[uuu, dashed, bend right=70, "\texttt{DiscreteDynamicsMonad}"']
\end{tikzcd}$$

Solid lines are valid for all choices of dynamics `D`,
dashed lines are valid only for `Discrete<V>` dynamics.

Figure 4: Resource monad stack

When a resource takes an error value instead of its usual dynamics type, we say that resource has "failed." The bind operation for this monad includes a try-catch block to convert a thrown exception into a failure automatically. Resource derivation also follows a "contamination" model of failure; if any source has failed, the result fails. Finally the `Registrar` will log errors to a special error state.

3. `Resource` - This wraps the previous two layers in a "getter" function, acting as a constant handle for the underlying dynamics, which change over time.

Monads link each layer to the next, and using something called a "monad transformer," we can also combine them to generate monads that jump multiple layers at once. The most common monads are shown in 4. We try to give the ones used most frequently the shortest, clearest names. This leads to some awkward names, like `ErrorCatchingToResourceMonad`, for rarely-used monads.

Putting this together, let's look at a concrete example: comparing polynomial resources. We have a method `lessThan(Polynomial)` on `Polynomial`, which returns an `Expiring<Discrete<Boolean>>`. The result is expiring rather than `Discrete<Boolean>`

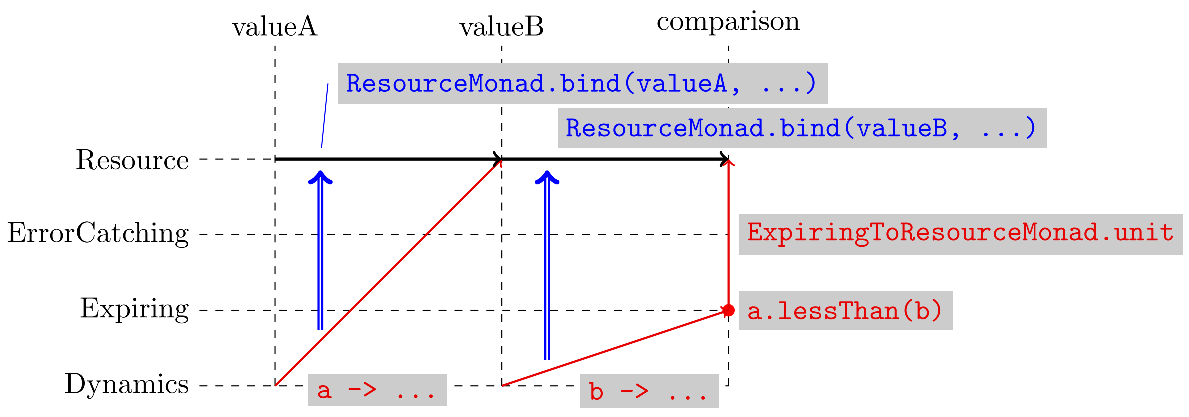

valueA
valueB
comparison
ResourceMonad.bind(valueA, ...)
ResourceMonad.bind(valueB, ...)
Resource
ErrorCatching
ExpiringToResourceMonad.unit
Expiring
a.lessThan(b)
Dynamics
a -> ...
b -> ...

Figure 5: Monad schematic for deriving a comparison resource

or `Boolean` because the result of the comparison can change if the two polynomials cross as time passes. The result is only expiring, rather than a higher level in the resource stack, simply because we don't need the higher levels to express the result. At the model level, we'd like to compare two `Resource<Polynomial>`s, valueA and valueB:[2]

```
Resource<Discrete<Boolean>> comparison =
  ResourceMonad.bind(valueA, a ->
    ResourceMonad.bind(valueB, b ->
      ExpiringToResourceMonad.unit( a.lessThan(b) ));
```

We can visualize how the code fits into the monad stack using fig. 5. This represents a general strategy for making a derivation "fit" the monad structure. First, we see where our end operation "naturally" is. In this case, it's a Dynamics to Expiring arrow. Since this arrow spans two levels, we can't use a map operation (easily). Instead, we choose a unit operation to lift the head of the arrow to a Resource, building a "full" diagonal arrow from dynamics to resources. Then we can just keep applying bind to add arguments and lift the "tail" of our lambda up from dynamics to resources.

3 Cells and CellResource

A resource can be derived from one or more other resources, such that getting the derived resource's dynamics in turn gets the source resources' dynamics. At some point, this call chain needs to terminate by querying some stored state. Unless this state is a constant, it needs to be stored in a cell.

[2]To keep model code clean, we usually put expressions involving monad operations into a helper method, like `PolynomialResources.lessThan`, which works on resources. This illustrates how to write such a method, though.

Such a resource is implemented by `CellResource`. When constructing a `CellResource`, one supplies initial dynamics, and optionally an effect trait.[3] Since `CellResource` is a general-purpose cell, effects on it are just functions of the dynamics. Thus, there are three commonly-used effect traits, given by static methods on `CellRefV2`:[4]

- `commutingEffects` - This declares that effect order never matters, so effects are combined by choosing an arbitrary order to apply all effects.
- `noncommutingEffects` - This declares that effects never commute, so emitting two effects in parallel causes the resource to fail.
- `autoEffects` - Attempts to detect which effects commute and which ones don't. When two effects are emitted in parallel, it tries applying the effects in both orderings. If the results agree, then that's the combined effect. If the results disagree, then the resource fails.

In practice, parallel effects tend to be uncommon, so `autoEffects` are used by default. The usually small performance penalty of trying the effects in both orders tends to be worth the correct and convenient semantics, since modelers usually won't have to worry about concurrency.

Effects on a cell resource can be written as in-line lambdas, usually with the help of the `DynamicsMonad.effect` or `DiscreteDynamicsMonad.effect` lifting functions. Alternatively, we can encapsulate the effect in a static helper method. It must be static, because there's one `CellResource` type, independent of dynamics type, and most effects only work on one dynamics type. For example,

```
import static CellResource.cellResource;
import static Discrete.discrete;
import static DiscreteDynamicsMonad.effect;

void main() {
  CellResource<Discrete<Integer>> counter =
    cellResource(discrete(0));

  counter.emit(effect(n -> n + 1));    // in-line
  increment(counter);                  // static equivalent
}

static void increment(CellResource<Discrete<Integer>> c) {
  c.emit(effect(n -> n + 1));
```

[3] See section 1.2 for more info on effect traits.

[4] `CellRef` is the Aerie/Merlin wrapper for a cell, and `CellRefV2` is a collection of static factory methods for building cells and effect traits that work with this framework.

```
}
```

Finally, let's address the common question "How do I make a derived resource I can emit effects on?" In short, this idea is inconsistent. A derived resource is completely determined by a function of other resources. Thus, to emit an effect that doesn't conflict with this definition of the resource, the effect would have to be a no-op, at which point it's useless. There are a few common patterns we might use instead.

1. Split the resource into independent "derived" and "cell" parts, then derive a result from both. For example, suppose we wanted a notion of "power usage" that we could also toggle on/off. We could have "circuit power" be a derived resource, "on/off" be an independent cell which we toggle through effects, and "power usage" be derived from those two.

2. Use a "driven" resource rather than a derived resource. Use a `CellResource` for the result, and a daemon task (typically defined using the `Reactions` class) that reacts to other resources to update the result cell. For example, if we wanted a "clearable integral," we could set up something like this:

   ```
   import static CellResource.cellResource;
   import static Polynomial.polynomial;
   import static Reactions.wheneverDynamicsChange;
   import static DynamicsMonad.effect;
   import static Arrays.copyOf;

   {
     var integral = cellResource(polynomial(0));
     // When integrand changes, take antiderivative
     // starting at current integral value:
     wheneverDynamicsChange(integrand, d ->
       integral.emit(effect(v -> d.integral(v.extract()))));
     // To clear integral:
     clear(integral);
   }

   static void clear(CellResource<Polynomial> integral) {
     // Clear the value but not higher coefficients
     integral.emit(effect(p -> {
       double[] coefficients = copyOf(p.coefficients());
       coefficients[0] = 0;
       return polynomial(coefficients);
     }));
   }
   ```

Notice that the “clear” effect and the driving task for the integral “play nicely” - neither overwrites the work of the other. This is typically what you want in a driven-resource setup. Note that in this setup, the driven resource is just a regular `CellResource`, so will not enforce using only “good” effects. If that kind of safety is important, make the cell private to the class that contains it, and expose methods that emit “good” effects instead. You can expose the cell through a getter method as a `Resource` instead of a `CellResource` to allow read-only access.[5]

[5] Technically this allows casting the result back to `CellResource`. To stop even this, expose `result::getDynamics` instead, which will create an inline implementation of `Resource` that can’t be cast back to `CellResource`.