


(https /  A..



Try  **HackMD**

(https://hackmd.io?utm_source=view-page&utm_medium=logo-nav).

Burrito Monads, Arrow Kitchens, and Freyd Category Recipes

Introduction

From Lawvere's [Hegelian taco](http://ncatlab.org/nlab/show/Hegelian+taco) (<http://ncatlab.org/nlab/show/Hegelian+taco>) to Baez's [layer cake analogy](https://math.ucr.edu/home/baez/cohomology.pdf) (<https://math.ucr.edu/home/baez/cohomology.pdf>) to Eugenia Cheng's *How to Bake Pi*, categorists have cultivated a rich tradition of culinary metaphors and similes. A well-known example in the world of computation is Mark Dominus's ["monads are like burritos"](https://blog.plover.com/prog/burritos.html) (<https://blog.plover.com/prog/burritos.html>) — where a tortilla (computational context) wraps diverse ingredients (values) to create a cohesive entity (effectful value) whose structure as a burrito is maintained as the meal moves down the assembly line (undergoes computations).

Monads, like burritos, come in many different varieties. In computer science monads serve to streamline computational patterns such as exception handling and context management. We illustrate these two examples by analogy.

Imagine you work at a burrito truck.

If a customer orders a burrito sans rice but rice is accidentally added, it can't be served. The **Maybe monad** handles exceptions such as this — when something goes wrong, it returns a special "Nothing" value rather than a flawed result, and once a failure occurs, all subsequent steps automatically preserve this state avoiding the need for repetitive error-checking.

❗ Image Not Showing

Possible Reasons

- The image file may be corrupted
- The server hosting the image is unavailable
- The image path is incorrect
- The image format is not supported

[Learn More →](https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg) (https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg)

Figure 1: The Maybe Monad illustrated with the burrito-making process

In Haskell, the parameterized type "Maybe a" has two constructors, "Just a" and "Nothing." The former is an alias for values of type "a" whereas the latter is indicative of an error. The following Haskell code exhibits the maybe type as an instance of the monad class:

```
instance Maybe Monad where
return = Just
Nothing >>= f = Nothing
(Just x) >>= f = f x
```

the return function has type $a \rightarrow \text{Maybe } a$, which is suggestive of its role as the monad unit. The so-called bind operation $\gg=$ has type $\text{Maybe } a \rightarrow (a \rightarrow \text{Maybe } b) \rightarrow \text{Maybe } b$, and corresponds to a bare-bones Kleisli composition (see [Monads: Programmer's Definition](https://bartoszmilewski.com/2016/11/21/monads-programmers-definition/) (<https://bartoszmilewski.com/2016/11/21/monads-programmers-definition/>) for details).

A slight generalization allows for descriptive error messages.

Definition. Given a collection E of **exceptions**, there is an associated **either monad** $((-) + E, \eta, \mu)$.

- $\eta_X : X \rightarrow X + E$ is the canonical inclusion
- $\mu_X : X + E + E \rightarrow X + E$ collapses both copies of E into one
- Kleisli morphisms are computations that may fail $X \rightarrow Y + E$
- Kleisli composition automatically propagates exceptions

Now suppose one of your regular customers walks up to the window and orders "the usual." Luckily you've recorded their preferences in a recipe book. The act of following the appropriate recipe is akin to executing computations that depend on a global state. The **Reader monad** is the functional programmer's solution to this impure concept.

❗ Image Not Showing

Possible Reasons

- The image file may be corrupted
- The server hosting the image is unavailable
- The image path is incorrect
- The image format is not supported

[Learn More →](https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg) (https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg).

Figure 2: The Reader Monad illustrated with the burrito-making process

Definition. Given a collection E of **environments**, there is an associated **reader monad** $((-)^E, \eta, \mu)$.

- $\eta_X : X \rightarrow X^E$ turns elements into constant functions $x \mapsto \lambda t. x$
- $\mu_X : (X^E)^E \rightarrow X^E$ turns function-valued functions into functions via diagonal evaluation $f \mapsto \lambda t. f(t)(t)$
- Kleisli morphisms convert inputs into executable functions from environments to outputs $X \rightarrow Y^E$
- Composition in the Kleisli category keeps track of the (immutable) environment as computations are chained together

Here is the same definition given as an instance of the Haskell monad class:

```
instance Monad ((->) r) where
```

```
  return x = \ _ -> x
```

```
  g >=> f = \t -> f (g t) t
```

The seminal paper of Moggi (<https://www.lfcs.inf.ed.ac.uk/reports/88/ECS-LFCS-88-66/ECS-LFCS-88-66.pdf>) has several other interesting examples illustrating the power of monads. Nevertheless, there are limitations to the kinds of things monads can do for us. What would happen if the burrito truck expanded into a full-service restaurant with various kinds of orders, multiple prep stations, and interdependent dishes?

This is where arrows enter the picture. Introduced by John Hughes

(<https://www.sciencedirect.com/science/article/pii/S0167642399000234>) in 2000, arrows generalize monads to handle more complicated computational patterns. While monads wrap values in computational contexts (like burritos in tortillas), arrows can represent entire preparation processes that can coordinate multiple inputs while maintaining awareness of the broader kitchen environment.

Arrows come with three core operations that determine their behaviour:

1. `arr` : This operation transforms a pure function into an arrow. This is like converting a standard burrito recipe into the food truck's workflow — taking a simple instruction like "add beans, then cheese" and adapting it to work within the kitchen setup.
2. `>>>` : This is sequential composition, it's how you sequence preparation steps, like ensuring that after the protein choice is added, the burrito moves to the salsa station, and then the wrapping station.
3. `first` : This operation allows an arrow to transform one component while leaving the other untouched. This is like modifying the burrito filling while leaving the side of

beans untouched. The capability to process just one part of a complex inputs is a powerful property that arrows have.

The above data is subject to 9 axioms, which will be discussed later in the post.

❗ Image Not Showing

Possible Reasons

- The image file may be corrupted
- The server hosting the image is unavailable
- The image path is incorrect
- The image format is not supported

[Learn More → \(https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg\)](https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg)

Figure 3: Arrow Operations. The three fundamental operations of arrows enable complex workflows beyond monadic structures.

Shortly before arrows were introduced, Power, Robinson, and Thielecke were working on Freyd categories — a categorical structure designed to model "effectful" computation. Using our simile, a Freyd category formalizes the relationship between an ideal burrito recipe (pure theory) and the real-world process of making that burrito in an actual kitchen with all its constraints and variables.

A Freyd category consists of three main components:

1. A category C with finite products, which represents "pure" recipes, like a standard handwritten recipe for a burrito, without considering specific cooking constraints.
2. A symmetric premonoidal category K which represents the "real world" kitchen operations, where timing matters, and operations may interact in some specific cases (like how using the oven can delay something else from being baked)
3. An identity-on-objects function J from C to K which translates pure recipes into actual kitchen operations based on the specific setup, ensuring that basic operations preserve the structure of the recipe.

❗ Image Not Showing

Possible Reasons

- The image file may be corrupted
- The server hosting the image is unavailable
- The image path is incorrect
- The image format is not supported

[Learn More → \(https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg\)](https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg)

Figure 4: Freyd Category Structure. The relationship between pure recipes (category C) and real-world kitchen operations (category K), connected by the identity-on-objects functor J that preserves structure while accommodating practical constraints.

Although arrows originated in Haskell, a highly abstract functional programming language, researchers began noticing apparent correspondences between the components of arrows and those of Freyd categories. These two structures, developed from different starting points, seemed to address the same fundamental challenge: how to systematically manage computations that involve effects, multiple inputs and outputs, and context-awareness. Therefore, it was hypothesized that arrows are equivalent to Freyd categories.

As a part of the Adjoint School, our group has been focusing on R. Atkey's [. \(https://www.sciencedirect.com/science/article/pii/S157106611100051X\)](https://www.sciencedirect.com/science/article/pii/S157106611100051X) work, which dispells this folklore and answers how arrows are related to Freyd categories. As the paper's title asks, in this blog post, we will investigate what is a categorical model of arrows? Are arrows the same as Freyd categories? Is the folklore about their equivalence mathematically sound, or is there more subtlety to their relationship?

The answer not only clarifies the theoretical aspects but also reveal practical insights for programming language design and quantum computation models. As we'll see, the relationship between arrows and Freyd categories is more nuanced than initially thought, with important implications for how we model complex computational systems.

Of course, the burrito comparisons only go so far, we will use this post to explain our journey to understand arrows and Freyd categories.

Key Insights:

- Monads encapsulate computational effects by wrapping values in contexts, much like burritos wrap ingredients in tortillas
- Different monads (Maybe, Reader) handle different patterns like exception handling and context management
- Arrows generalize beyond monads to handle multiple inputs and coordinate complex processes, like managing an entire kitchen rather than just making individual burritos

Beyond the Kitchen: Arrows and Freyd Categories

Formally, a monad is defined to be a monoid in the category of endofunctors. While we've explored monads through concrete examples like the maybe monad and the reader monad, arrows are a further generalization of the computational structure that monads offer. Freyd categories offer a similar concept, as we briefly discussed earlier — we will now describe what these structures offer before pitting them against each other.

Arrows

To understand arrows from a categorical perspective, a good place to start would be by defining them as monoids in a category of strong profunctors. There are two definitions of arrows that we will explore in this blog post, and the profunctor definition is one of them.

Formally, a profunctor P is a functor $P : C^{\text{op}} \times C \rightarrow \text{Set}$. Unlike regular functors which are maps between categories, profunctors map from a pair of categories (with the first one considered in its opposite form) to the category of sets. Intuitively, a profunctor associates to each pair of objects a set of "generalized morphisms" between them.

Composition of profunctors requires a more sophisticated construction using coends. Given profunctors P and Q , their composition is defined as:

$$(P * Q)(x, z) = \int^y P(x, y) \times Q(y, z)$$

This formula has an intuitive interpretation similar to a dot product. The coend instructs us to take a colimit (essentially a sum) over all intermediate objects y , considering all ways to go from x to z via some intermediate y .

The identity profunctor is simply $\text{id}(x, y) = C(x, y)$, which uses the hom-sets of the category C .

The definition of arrows in terms of the category of profunctors shows how they are capable of modelling complicated relationships between inputs and outputs.

Having examined arrows from a profunctor perspective, we now turn to their concrete definition in terms of operations. This operational view aligns with how arrows are implemented in programming languages like Haskell.

Definition. An **Arrow** in a Cartesian closed category C consists of a mapping on objects and three families of morphisms:

- A mapping on objects $\text{Ar} : \text{ob}(C) \times \text{ob}(C) \rightarrow \text{ob}(C)$

This defines the arrow type constructor, which takes input and output types and produces an arrow type between them.

- A family of morphisms $\text{arr} : Y^X \rightarrow \text{Ar}(X, Y)$

This operation lifts a pure function into the arrow context, allowing regular functions to be treated as arrows.

- A family of morphisms $\ggg : \text{Ar}(X, Y) \times \text{Ar}(Y, Z) \rightarrow \text{Ar}(X, Z)$

This enables sequential composition of arrows, similar to function composition but preserving the arrow structure.

- A family of morphisms $\text{first} : Y^X \rightarrow \text{Ar}(X \times W, Y \times W)$

This is perhaps the most distinctive operation, allowing an arrow to process the first component of a pair while leaving the second component unchanged.

This data is subject to nine axioms that ensure arrows behave consistently. To make these abstract operations more concrete, consider the following example, where $\text{Ar}(x, y) := Y^X$, $\text{arr} := \text{id}_{(Y^X)}$, $\ggg := \text{composition}$, and $\text{first}(f) := f \times \text{id}$. We will now list the arrow laws and draw commutative diagrams based on this example.

The arrow laws $\text{arr}(\text{id}) \ggg a = a$ and $a \ggg \text{arr}(\text{id}) = a$ express left and right unitality of identities under composition.

❗ Image Not Showing

Possible Reasons

- The image file may be corrupted
- The server hosting the image is unavailable
- The image path is incorrect
- The image format is not supported

[Learn More → \(https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg\)](https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg)

Figure 5: Arrow Laws

The arrow law $(a \gg b) \gg c = a \gg (b \gg c)$, represents associativity of composition.

❗ Image Not Showing

Possible Reasons

- The image file may be corrupted
- The server hosting the image is unavailable
- The image path is incorrect
- The image format is not supported

[Learn More → \(https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg\)](https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg)

Figure 6: Arrow Laws

The arrow law $\text{first}(a \gg b) = \text{first}(a) \gg \text{first}(b)$ encodes functoriality of $- \times W : C \rightarrow C$.

❗ Image Not Showing

Possible Reasons

- The image file may be corrupted
- The server hosting the image is unavailable
- The image path is incorrect
- The image format is not supported

[Learn More → \(https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg\)](https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg)

Figure 7: Arrow Laws

The arrow law $\text{first}(a) \ggg \text{arr}(\pi_1) = \text{arr}(\pi_1) \ggg a$ express naturality of the counit $- \times W \rightarrow \text{id}_C$, i.e., the first projection maps.

❗ Image Not Showing

Possible Reasons

- The image file may be corrupted
- The server hosting the image is unavailable
- The image path is incorrect
- The image format is not supported

[Learn More → \(https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg\)](https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg)

Figure 8: Arrow Laws

The arrow law $\text{first}(a) \ggg \text{arr}(\alpha) = \text{arr}(\alpha) \ggg \text{first}(\text{first}(a))$ asks that first play nicely with associators.

❗ Image Not Showing

Possible Reasons

- The image file may be corrupted
- The server hosting the image is unavailable
- The image path is incorrect
- The image format is not supported

[Learn More → \(https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg\)](https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg)

Figure 9: Arrow Laws

The arrow law $\text{first}(a) \ggg \text{arr}(\text{id} \times f) = \text{arr}(\text{id} \times f) \ggg \text{first}(a)$ is an interchange law which says $\text{id} \times g : (- \times W) \rightarrow (- \times W')$ is a natural transformation for every $g : W \rightarrow W'$ in C .

❗ Image Not Showing

Possible Reasons

- The image file may be corrupted
- The server hosting the image is unavailable
- The image path is incorrect
- The image format is not supported

[Learn More → \(https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg\)](https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg)

Figure 10: Arrow Laws

Two arrow laws trivialise as a result of our example, so diagrams aren't produced. The first such law is $\text{arr}(f; g) = \text{arr}(f) \ggg \text{arr}(g)$. For our example, this law trivialises, as $\ggg := \text{composition}$ and $\text{arr} := \text{id}_{(Y^X)}$. The second law that trivialises would be $\text{first}(\text{arr}(f)) = \text{arr}(f \times \text{id})$ as a result of us setting $\text{first}(f) := f \times \text{id}$.

Freyd Categories

To understand Freyd categories, we must first define what a symmetric monoidal category is.

Definition. A **symmetric premonoidal category** includes:

- An object I (unit).
- Natural transformations that define how objects interact:
 - Associativity: $\alpha : (x \otimes y) \otimes z \rightarrow x \otimes (y \otimes z)$
 - Left unitor: $\lambda : x \otimes I \rightarrow x$
 - Right unitor: $\rho : I \otimes x \rightarrow x$
 - Symmetry: $\sigma : x \otimes y \rightarrow y \otimes x$
- All components are **central**.

A morphism $f : x \rightarrow x'$ is **central** if $\forall g : y \rightarrow y', \quad f \otimes y; x' \otimes g = x \otimes g; f \otimes y'$

Now, we can define a Freyd category, recalling the definition from the introduction.

Definition. A **Freyd category** consists of:

- A category C with finite products.
- A symmetric premonoidal category K .
- An identity-on-objects functor $J : C \rightarrow K$ that:
 - Preserves symmetric premonoidal structure.
 - Ensures $J(f)$ is always central.

Arrows vs Freyd Categories: Similarities and Differences

At first glance, the definition of a Freyd category appears strikingly similar to that of an Arrow. This apparent similarity led to the folklore belief that they were equivalent structures.

A Freyd category consists of two categories C and K with an identity-on-objects functor $J : C \rightarrow K$, where:

- C has finite products
- K is symmetric premonoidal (with a functor $- \otimes z$)
- J maps finite products in C to the premonoidal structure in K

In our culinary metaphor, we could map this to:

- C : The idealized recipes (Haskell types and functions)
- K : The real-world kitchen operations (computations represented by $\text{Ar}(x, y)$)
- J : The translation process (via `arr`, embedding pure functions)
- Composition in K : The sequencing of operations (via `>>>`)

- Premonoidal structure in K : The ability to process pairs (via `first`)

This correspondence seemed so natural that for many years, the programming languages community considered Arrows and Freyd categories to be essentially the same concept expressed in different languages.

However, Atkey's work revealed a crucial distinction: **Arrows are more general than Freyd categories**. The key difference lies in how they handle inputs:

- Freyd categories allow only a single input to computations
- Arrows support two separate inputs:
 - One may be structured (modeled using comonads)
 - This additional flexibility allows Arrows to represent computations that Freyd categories cannot

To bridge this gap, Atkey introduced the concept of **indexed Freyd categories**, which can model two structured inputs. The relationship can be summarized as:

Arrows are equivalent to Closed Indexed Freyd Categories

In our culinary metaphor, we can understand this relationship as follows: a Freyd category is like a restaurant that can only take one order at a time (a single input), while arrows are like a more sophisticated establishment that can handle both individual orders and special requests that come with their own context (two inputs, one potentially structured). The closed indexed Freyd categories that Atkey identifies represent the perfect middle ground — restaurants that can efficiently manage multiple orders with specialized instructions while maintaining the core operational principles that make kitchens function. This is particularly valuable when preparing complex "quantum dishes" where ingredients might be entangled and interact with each other in non-local ways.

❗ Image Not Showing

Possible Reasons

- The image file may be corrupted
- The server hosting the image is unavailable
- The image path is incorrect
- The image format is not supported

[Learn More → \(https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg\)](https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg)

Figure 11: Arrows vs. Freyd Categories. Arrows support two inputs (one potentially structured) and are equivalent to Closed Indexed Freyd Categories, which generalize standard Freyd Categories that handle only single inputs.

This distinction helps explain why Arrows have proven particularly useful in domains like quantum computing, where managing multiple inputs with complex relationships is essential.

R. Atkey's paper finds the relationship between arrows and different constraints on Freyd categories as follows:

❗ Image Not Showing

Possible Reasons

- The image file may be corrupted
- The server hosting the image is unavailable
- The image path is incorrect
- The image format is not supported

[Learn More → \(https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg\)](https://hackmd.io/@docs/insert-image-in-team-note?utm_source=note&utm_medium=error-msg)

Figure 12: Relationship between structures

Key Insights:

- Arrows can be defined both as monoids in categories of strong profunctors and operationally through concrete morphisms (`arr`, `>>>`, `first`)

- Freyd categories formalize the relationship between pure functions and effectful computations using symmetric premonoidal structure
- Despite the folklore belief, arrows are strictly more general than Freyd categories because they can handle two separate inputs (one potentially structured)
- Arrows are equivalent to closed indexed Freyd categories, bridging the conceptual gap

Thoughts + Further Questions After Reading R. Atkey's Work

Atkey's work helped us realize the subtle differences between arrows and Freyd categories. By carefully examining the mathematical structure, he demonstrated that arrows are more general than Freyd categories.

The critical observation — that arrows can process two distinct inputs whereas Freyd categories can only allow one — has far-reaching consequences. Arrows' capacity to handle multiple inputs with a single potentially structured offers tractability that is particularly useful in quantum computing. Particles in quantum systems can be in entangled states in which manipulation of one particle influences others in real time irrespective of distance. This non-local interaction can be modelled through arrows' ability to bring several inputs together while keeping track of their interrelationship.

This relationship of arrows and quantum computing leads us to a number of interesting questions:

- Can relative monads model quantum effects, and how do they compare to arrow-based approaches?
- Given that symmetric fusion categories lack cartesian products, how can we generalize the notion of Freyd categories — originally built over cartesian categories — to this rigid, monoidal setting?
- Can indexed Freyd categories explain quantum contextuality?

The ride from monadic burritos to arrow kitchens has carried us farther than we anticipated, illustrating that even established mathematical folklore sometimes requires precise re-evaluation. As we continue to learn about these structures, we hope this post will motivate others to participate in the exploration of these tools and their use in quantum computing and beyond.

