Ctional Programming, Or How the "Fun" got taken out of Functional Programming: An examination of Haskell as a tool for interactive web applications

---

A Thesis
Presented to
The Division of Mathematical and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Aadit Bagdi

January 2023

Approved for the Division
(Computer Science)

James D. Fix

Acknowledgements

I am sorry if you aren't mentioned, you would definitely be in here if I had more space.

I'd never thought I'd get to this point. Five and a half years, and it's finally done. However, I would not be here if it weren't for a lot of people in my life. Firstly I want to thank my family. Mama, Papa, Anan, Oskar, thank you for always having my back for all those years, and thanks for putting up with me. As well, thank you to David Johnson and the Miso community, your help was invaluable for me completing this thesis. Finally, thank you Jim.

Thank you Surfchamps, we've all been friends for 8 years, and here's to many more. I hope we can see each other in person again soon. Thank you Brendan, for being an all-around awesome and funny dude. You definitely are, as a Mancunian would say, "mega." Thanks Leila, for all the three hour long conversations about, well, everything. You are such an interesting, cool person and I always love hanging with you! Thank you Teague, you always were there to lend an ear whenever the going got tough, and I really really appreciate that. I wouldn't have gotten to this point without you, friend. Thank you William for all the wisdom and just being the sweetest guy. Thank you Shisham, even though you're in another state, I still feel your encouragement. Thank you Finn, your pasta is bomb and all those pasta nights were a very welcome break from the monotony. Thank you Perry, for being there to revel in our suffering as spring/fall seniors, we're finally done! Thank you Riley, you are such a smart and kind person, and even though I failed Comp Comp the first time around, I still felt like I learned something thanks to you. Thank you Harrison for being the most cracked madarchod, and I mean that in the best way possible. Thank you Ryan, for always being down to play Arena and for allowing me to bum so many rides off you. Thank you Ella, for being such a sweet person, helping you out with qual over the winter definitely helped stave away the loneliness when no one was here. Thank you Vaughn for being one of the chillest people I know. Thank you Tommie, even though we became friends right before you left I still really valued the times we hung out. Thank you Matt, for being such an all around awesome guy, it's really nice to have someone who gets what I'm going through. And finally, thank you Sima. If it weren't for you, this year would've been downright intolerable. So thank you for being in my life, I value you a lot, and remember to eat your eggs™.

I love you all so much, and thank you all once again!

"Lost in the fog,
I've been treated like a dog,
And I'm outta here"

— Liam Gallagher, 1995

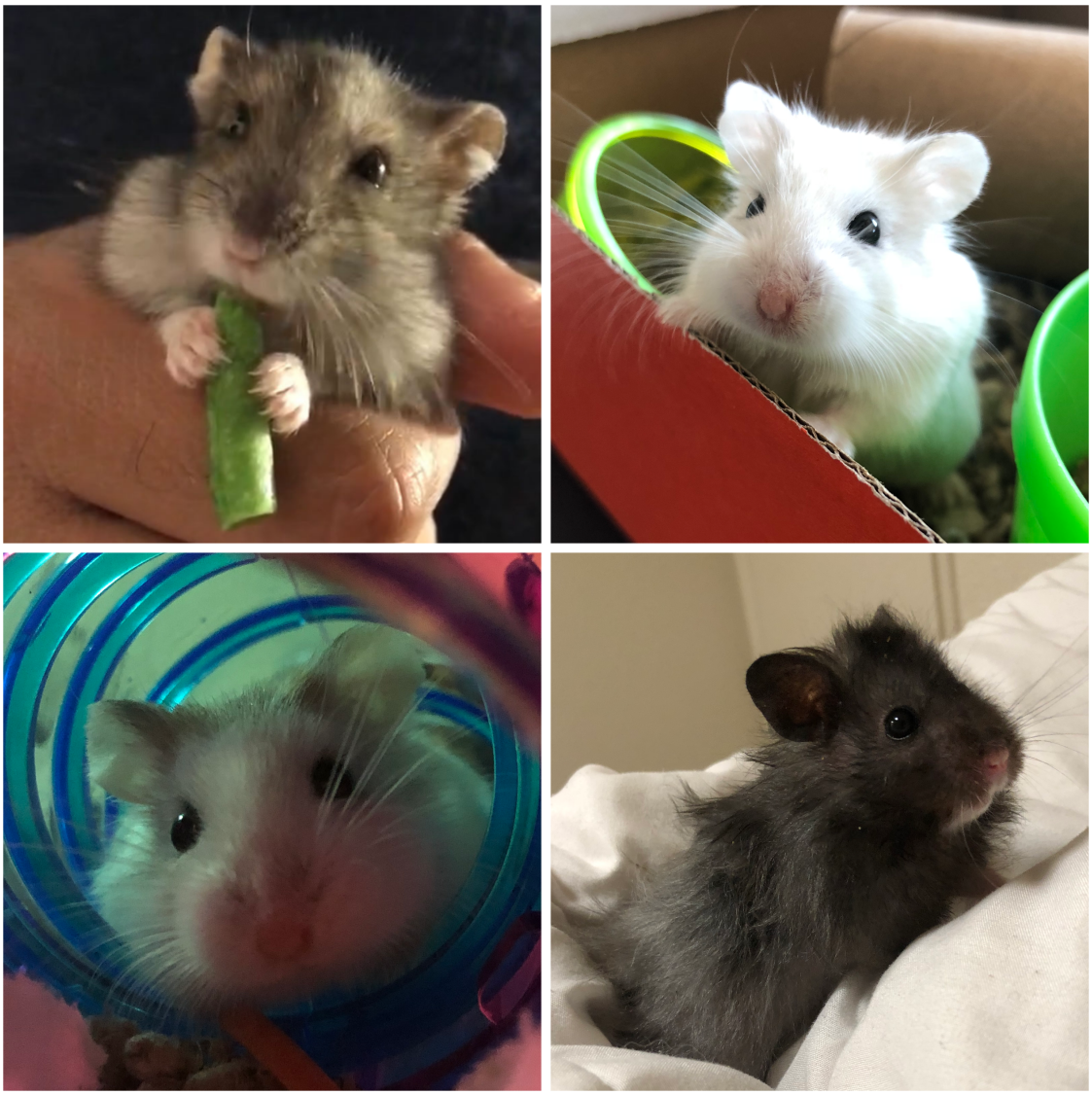
Table of Contents

Abstract

This thesis surveys writing Haskell code that is interactive, having programs that react to user input via animated graphics and sound effects. Ultimately, I present Miso, a Haskell front-end for JavaScript, as a tool for this. Using Miso requires knowledge of Haskell's type system and monads, a general framework for coding side-effects in a purely functional language that has been widely adopted by the Haskell community. I demonstrate Miso, and how Haskell code for web applications are structured using it, with two simple applications: modifying an interactive animation of the Nintendo character Mario by adding sound, and then extending that to play a sequence of sounds. We end with a brief discussion of Miso's feasibility as a music composition platform in Haskell, specifically if it can be used for live composition, and possible extensions of my work.

Dedication

This thesis is dedicated to Tulip, Cinnamoroll, Almond and Baby Bear. I love you and miss you all so much. (Top Left: Tulip, Top Right: Cinnamoroll, Bottom Left: Almond, Bottom Right: Baby Bear)



Introduction

Haskell has a history with music composition. In fact, Paul Hudak, one of many who helped develop Haskell, was also an avid musician. Thus, there exists a connection between Haskell and music composition. As someone with an interest in both functional programming and music, I decided to explore this connection. Paul Hudak himself helped develop two possible frameworks for music composition: *Haskell School of Expression* and *Haskell School of Music*, both of which allow for the composition of music in Haskell [2, 3]. However, I have also found another possible framework for music composition in Haskell: Miso, a Haskell front-end for JavaScript. There are some similarities between these tools, as all of them require a mastery of *side-effects* (such as outputting an animation on the screen or playing a sound), Haskell's type system and monadic libraries for IO computations and, in Miso's case, tracking the state of the program. However, when I tried using HSoE and HSoM, they wouldn't work too well on a modern machine, considering these tools are 20 years old. This got me wondering: could Miso carry Hudak's torch? Could Miso serve as a modern, viable platform for music composition? Can Haskell be utilised for live music composition, and if so, how?

The first chapter of this thesis gives the reader a crash course on functional programming, Haskell and its type system. The second chapter familiarises the reader with monads and how they are used to facilitate side-effects and interaction in Haskell. The third chapter introduces Miso, a Haskell to JavaScript compiler, and how it may be used to output sound by modifying the `Mario` demo. Finally, I conclude by addressing the goal of this thesis: how might one go about live music performance in Haskell?

Chapter 1

A Primer on Functional Programming and Haskell

In order to understand this thesis, it is important to first get a grasp of functional programming and Haskell. In this chapter I will give a quick primer on functional programming and Haskell, then provide a few examples to compare it with an imperative language like C++.

1.1 What is functional programming?

Functional programming is exactly what the name implies. In functional languages, the function is king. Programs are constructed by applying functions to arguments, and finally returning a value at the end. A program is made of multiple functions applied and composed with each other. Those functions are also made of other functions, and so on. Another way to state this is that a functional language is a language where its primitives are functions.

1.1.1 Heh heh heh, I don't have loops anymore

Of course, since functional programming is a totally different programming paradigm, there will be some interesting side-effects (more on side-effects in a bit.) Firstly let us talk about the matter of assignment, or rather the lack of it. In more garden-variety languages you can write something like so:

```
int main() {
   int x = 5;
   int y = 6;
   return x + y;
}
```

What is this simple program doing? It is taking some variable `x` and *assigning* it to the integer 5. It also takes a variable `y` and *assigning* it to the integer 6. Finally

the function returns the result `x + y`. In imperative languages like C++, there is this notion of "assignment": you are setting one thing as another. Note that this means that assigned values can change. You can set `x` to another value, and C++ will happily accept that. You can write something like this in C++:

```
int main() {
    int x = 420;
    int x = 69;
    return x;
}
```

Here, the function will return 69, since `x` was set to that last. However, in functional languages "=" means something closer to what "=" means in mathematics [4]. In Haskell I can write:

```
main =
    x = 5
    y = 5 + 7
```

With "=" working the similarly as in math, what `x = 5` is saying "`x` is another name for the number 5, so wherever I see a 5 I can replace it with x." So we can rewrite the same code as this:

```
main =
    x = 5
    five = x
    y = five + 7
```

I mentioned side-effects earlier. Side-effects are essentially what they sound like: it's when a function does something other than computing the result before the final result is returned. For example, in C++ you can do this:

```
#include <iostream>

int main() {
    int x = 4;
    int y = 34;
    int z = x + y;
    std::cout << z;
    return x;
}
```

Here, the side-effect is printing to the console the value of `x`. Printing to the console is a step that's unnecessary to the computation of the final value `main()` is trying to return. In functional languages however, functions can have no side-effects [4]. They

are similar to mathematical functions in the way that a math function takes an input in its domain and returns something in its co-domain, and nothing else. Going back to the printing example above, if writing this in a purely functional language like Haskell, the print statement would not be allowed as it is a step that isn't involved in getting a return value from an input value.

1.1.2 Why use functional programming?

Here is a programming paradigm that is different to what is commonly seen and is instead closer to mathematics. Why is this useful? According to those who drank the functional programming Kool-Aid, there are a few advantages here. In purely functional languages like Haskell, due to the lack of side-effects and assignments, functions can only compute values and variables cannot change, respectively. As well, it doesn't really matter what order the functions are evaluated. In order to illustrate why some prefer purely functional languages, I will compare some commonly used functions in both Haskell and C++.

Firstly, let's consider a common factorial function. The intuitive method is to use iteration to compute $n * (n-1) * (n-2) * ... * 2 * 1$. In C++ you can write this:

```
#include <iostream>

int factorial(int n) {
   int result = 1;
   if (n < 0) {
      std::cout << "Factorials of negative numbers don't exist!";
      return -1;
   }
   for (int i = n; i > 0; i--) {
      result *= i;
   }
   return result;
}
```

This function, once given an integer n, iteratively calculates $n*(n-1)*(n-2)*...*2*1$.

However, Haskell, being a purely functional language, doesn't have loops. To accomplish the same in Haskell, we would instead typically express this calculation recursively:

```
factorial :: Int -> Int
factorial n | n < 0 = error "Factorials of negative numbers don't
   exist!"
```

```
    | n == 0 = 1
    | n == 1 = 1
    | n > 1 = n * factorial (n - 1)
```

There are a few things that are different from C++. Firstly, what does `factorial :: Integer -> Integer` mean? That is called the *type signature* of the function, which I will go over in more depth in Chapter 2. Next, what does | mean? That is known as a *guard*, which is analogous to the venerable "if-else" statements of other languages. What its saying here is "if `n` is less than 0 return an error, else if `n` is 0 return 1, else if `n` is 1 return 1, else if `n` is greater than 1 compute the factorial recursively." If you compare the Haskell code to the C++ code, notice that it looks like a mathematical expression. You can begin to see the reasoning behind functional programming. If you can write out a program knowing the mathematical meaning behind it, it can be easier to automate reasoning about it. As well, if you can rewrite the program to be more efficient, you can prove your new code is correct if you have the right mathematical backing behind it. Another difference between purely functional languages like Haskell and other functional languages like Standard ML. Standard ML has an *evaluation order*, while Haskell wants you to write code in terms of functions that calculate values, and they don't depend on the order they are evaluated in.

Now I will also show you the `accumulate` function. When given a list of integers, `accumulate` will return the sum of all its elements. Here it is in C++:

```
int accumulate(std::vector<int> vector) {
   int result = 0;
   for (std::vector<int>::iterator it = vector.begin(); it !=
      vector.end(); it++) {
      result += *it;
   }
   return result;
}
```

Essentially what this is doing is when given a vector, it uses the `std::vector`'s iterators to iteratively go through each value in the list and add it to `result`. Now let's see how one would do it in Haskell:

```
accumulate :: [Int] -> Int
accumulate [] = 0
accumulate (x:xs) = x + accumulate xs
```

Note that we didn't have to use iterators or iteration at all. Once again like `factorial`, we are using a recursive solution. There is a base case of an empty list, on which the sum should be zero. In the `x:xs` notation, `x` is the head of the list, or the first element, while `xs` is the tail, or the list of elements not including the head. All the

function does is take the first element of the list and adds it the result of recursively calling `accumulate` on the tail of the list.
As the last example, I will be showcasing the `map` function. When fed a function and a list, `map` computes the result list when that function has been applied to all of the elements on the list. Here is a way to write something like that in C++:

```
int plusOne(int x) {
   return x + 1;
}

int map(std::vector<int> vector) {
   std::vector<int> result = [];
   for (std::vector<int>::iterator it = vector.begin(); it !=
      vector.end(); it++) {
      result.push_back(plus_one(*it));
   }
   return result;
}
```

Note that our "map" function isn't truly a `map` function because it doesn't take a function as an argument (however, C++ do have lambdas and libraries for functions as objects, which are certainly inspired by functional programming!) Rather it takes a list, iteratively calls a function on each element in that list, and returns the resulting list. Note that the types of both the function being applied and the list have to be strictly defined. You cannot have `plusOne` be a function of type `double` while the list is of type `int`. Now let's see how it's done in Haskell:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Firstly, let us go over what this is doing. When given an empty list, regardless of the function (that's what "_" means), `map` returns an empty list. Otherwise when given a function and a list, it first applies that function to the first element in the list, and then "prepends" (that is what ":" means) that to the result of calling `map` on the tail of the list. There are a few differences to note here. First note that this `map` takes a function and a list as arguments, and returns another list (that is represented as the type `(a -> b) -> [a] -> [b]`, more on that in the next section.) This is, as discussed earlier, one of the hallmarks of Haskell and other functional languages since the language's primitives are functions. These type of functions are known as *higher-order functions*. The next thing to note is that the type of the function being applied to the list is not of any specific type: this is known as *parametric polymorphism*. I will touch on that during the next section too.

1.2 Haskell's type system

Let's discuss Haskell's type system. Haskell is a *statically typed* language. What this means is that types of variables are known at compile time. In other words, the types of variables are `bound` to those variables. In addition, Haskell is *strongly* typed, which means that once a variable is bound to a type, that variable cannot be bound to another type. What this all means is the onus of determining variable types is on the programmer. Languages like C and Haskell are both statically typed. However, Haskell has something that many other statically typed languages don't have: *type inference*.

1.2.1 Type inference

Let's say you have the following C function:

```c
#include <stdbool.h>

bool foo(int x) {
   bool y;
   if (x > 5) {
      y = true;
   }
   else {
      y = false;
   }
   return y;
}
```

Notice how here, we had to declare the type of every variable used, every argument in the function, and the return type of the function itself. However, Haskell and GHC, the Glasgow Haskell Compiler, is nifty that we don't have to give the types! Here is the same function in Haskell:

```haskell
foo x = y
  where
    y = if x > 5
      then True
      else False
```

Now note that we didn't define any of the types here! After looking at this code, GHC has enough information to infer that `x` is an integer and `y` is a boolean. In functional languages, every expression has a type, so GHC uses type inference to determine the type of each expression in our code. As an example, every `if` statement must have a corresponding `else` statement. If we make a function to convert a boolean to a string as so:

```
bool2string b = result
  where
    result = if b
      then "True"
      else "False"

main = putStrLn (bool2string(foo 4))
```

In this case, if the `else` statement wasn't there, GHC wouldn't able to tell what type `result` would be, since the input to `foo` is less than 5, and `bool2string` wouldn't know what type to return in the `False` case.

1.2.2 Type signatures

However, the problem with type inference and the code I put above is that it's for a human to tell what type functions are. The above examples are simple enough, but for larger, more complex functions it can get out of hand quickly. Thus, in Haskell, you can give functions *type signatures*. These signatures succinctly state the types of the function's input, and the return type of the function. If we wanted to write a type signature for the two functions above, they would look like this:

```
foo :: Int -> Bool
bool2string :: Bool -> String
```

The very last type in these signatures are the return types of their corresponding functions. The type before the arrow are the types of input the function takes. Let's take a look at a slightly more complex signature. Let's examine the type of the `map` function we discussed earlier in this chapter:

```
map :: (a -> b) -> [a] -> [b]
```

There is more than one arrow here! However, the very last type is still the return type, so every type before in that chain of arrows are the input types. These types look kind of funky, however. What is an `a` and `b` type? Allow me to make a brief aside.

Polymorphism

Polymorphism is when a variable can have multiple types. In the `map` example, this arises for `a` and `b` types. `map` applies a function on a list to get a list consisting of the results of applying that function on each element in the input list. Thus, for a function like that, it would be necessary for it to handle any type, since the input list and function can potentially be any type. Thus, `map` is defined in terms of these placeholders, `a` and `b`, that can stand in for any type. These stand-ins for types are

known as *type variables*.

Coming back to the original type signature, what does `(a -> b)` actually mean? Recall the types of `foo` and `bool2string`. Their type signatures look similar to `(a -> b)`. Indeed it works the same way too. The type `(a -> b)` represents a function that takes in an input of type `a` and returns an output of type `b`. Now what about `[a]` and `[b]`? Square brackets designate a list in Haskell. Thus, `[a]` and `[b]` are lists that consist of elements with types `a` and `b`, respectively. Now, just by looking at the type signature, we can say "`map` takes in a function that takes an `a` and returns a `b`, and a list of `a`'s. It then returns a list of `b`'s." Given the definition of `map`, this makes perfect sense.

1.2.3 Lightning round of types

Here I'll just describe some other types that one may see in Haskell, now that we are armed with the knowledge to understand Haskell types and type signatures.

- `(a, b)` refers to a 2-tuple, with the first element being of type `a` and the second of type b. For example `(4, 5)` would have type `(Int, Int)`.
- `String` refers to a string. Alternatively, this is also represented by `[Char]`, a list of characters.
- `IO a` refers to an `IO` computation (more on that in the second chapter), followed by returning something of type a.
- `M a` refers to a generic monad (more on that in the next chapter.)

1.2.4 Variants and Records

The last thing you need to know about Haskell's type system to understand this thesis are *variants* and *records*. These are both *algebraic datatypes*.

1.2.5 Variants

Variants are declared with the keyword `data`. The simplest variant is a boolean:

```
data Bool = True | False
```

Here, a `Bool` value could be one of two things: `True` or `False`. Another example could be if you are trying to model all possible animals on a farm:

```
data farmAnimal = Cow | Pig | Sheep | Horse | Dog | Cat | ...
```

A `farmAnimal` can only be one of these things. This is why variants are also known as *sum types*, since the parameters are combined into one using *or*: a `farmAnimal` can only be a `Cow` *or* a `Pig` *or* a `Sheep`, and so on.

Records

Let's say you want to model a fantasy creature. You could, for example, have the name of the creature, its species, its height and weight, and whether it's evil. Haskell allows you to do this.

```
data Creature = ...
```

Now you can define the member types of the records as so:

```
data Creature = Creature { Name :: String
              , Species :: String
              , Height :: Int
              , Weight :: Int
              , isEvil :: Bool
              } deriving (Show)
```

You can declare a new variable with this new created type by declaring one and then passing in the required members as so:

```
newCreature :: Creature
newCreature = "Daam Regco" "Gremlin" 65 200 True
```

This is known as a record. Confusingly, records are also declared with the keyword `data`. They essentially combine many types into one. For this reason, they are also known as *product types*, they are defined by all its parameters using *and*: a `Creature` is defined as a `species` *and* a `Height`, *and* a `Weight`, *and* an `isEvil` boolean.

1.3 Upshot

So if Haskell and functional programming has these unique features, do other languages have facilities to do similar things? The answer to that is yes. Due to all these advantages functional programming has, several other languages now support some form of functional programming paradigms. For example, C++ has *lambdas* and `std::function` that can act similarly to functions in Haskell in that it can be passed to other functions as an argument. As well, C++'s Standard Template Library, taking influence from Haskell, now have their own implementations of `accumulate` and `map`. Other languages like Python as well support a functional programming paradigm.

1.4 Summary

In this chapter, I have discussed some of the characteristic features of functional programming languages, and discussed some basic Haskell programming and com-

pared it to C++. Finally I discussed some of the influence Haskell has had on other languages. In the next chapter, we will dive into an important part of coding with Haskell: *monads*.

Chapter 2

Monads and Haskell

In order to understand this thesis, an understanding of monads is required. Monads are a source of great confusion, and can be a mental block for many a programmer trying to learn Haskell. In this chapter I will explain what they actually are, discuss how they work in Haskell and give a few examples. Do note that many of these examples are canonical examples that are often used to teach monads.

2.1 State of the program address

In order to understand what monads are, it would be beneficial to go over what the *state* of a program is. Simply put, it's just the things in a program that aren't local to a specific function [1]. For example, take the following Python code:

```
x = 5
y = 15

def PlusThirteen():
    z = 13
    a = y + z
    return a

b = int(input("Enter a number: "))

print(PlusThirteen() + b)
```

In this case, everything that is outside the scope of `PlusThirteen()` is the state of this program. That includes the global variables `x` and `y`, as well as the `print` statement and the `input` statement.

Not using your time effectively: State's time dependence

The reason we care about state is that it's messy to deal with. The main problem with state is that it's *time-dependent*. [1] illustrates this with the following example.

Say you have two functions, `f` and `g`. Let's say `f` receives some value that it needs to perform a computation on. However, `g` changes that value. That means `f` is using an outdated value. Worse still, if other functions depend on `f`, the error caused by using an outdated value will compound through the program. This is what is meant by time-dependence. When considering things within the state, we have to consider not only their value but also their value at a specific time. This complicates the analysis of programs and makes code harder to understand.

The difference between actions and expressions

In order to do something about the state and its time-dependence, we should familiarize ourselves with the concepts of *actions* and *expressions*. An expression is something, such as a variable, that has a value. If we look at the Python example, `x = 5` and `y = 15` are both expressions. More interestingly, reading the user's input is also an expression. If that Python snippet is run, the variable `b` is assigned to whatever number the user types into the console. Since `b` is part of that program's state, the state is modified. As well, the value of the `input` statement is never fixed. It could be anything the user wants it to be at that time. As mentioned before, this lack of consistency and time dependence can cause issues. Expressions, if they are in the state, aren't ideal because of this.

What is an *action*? Simply put, an action is something that could optionally happen [1]. In this example, both the printing and reading of the user's input are actions, since either they are done or not. Compared to expressions, whose value could be whatever the programmer wants, the value of an action isn't the expression it's used in, but rather it's value is whatever it's doing. The value of `input` isn't whatever the user enters in the console, it's the *action* of fetching something from the console. When the code displays something on the console, it's value is the fact that it did print something. This means that actions are time-independent, regardless of any expression it's contained in. This may seem useful, but actions by themselves aren't incredibly useful. For example, Python has an action `open()`, which loads a file. However, unlike an expression, setting something like `x = 1 + open(PATH)` makes no sense. As well, what would returning or printing an action type actually do? An action type inherently doesn't contain much information, only whether an action was completed or not, so that wouldn't be very useful. We are getting close to achieving time-independence, but we aren't exactly there. How do we resolve this? This is where monads come in.

2.2 Enter monads

We have shown that expressions are time-dependent. We have also shown that actions aren't time-dependent but they are not practical. How can we resolve this? Well, let's take the example of trying to print an action, say reading a file. Obviously by itself that doesn't achieve much, but what if we link the file action with the print

action? The only thing the programmer cares about is the file the read action gets, not whether the file was read or not. If there was a way to associate that file with an action, that would make our lives easier. That is exactly what a monad is. [1] has an excellent analogy that really gets the basics of monads with the example of an action that reads a 5 from user input:

> If a program keeps track of the inventory in a warehouse, the input number 5 might represent the number of boxes on a shelf. The value 5 is pertinent in the problem domain. If you walk up to someone who manages the warehouse and say "You have 5 boxes," that person knows what you mean. On the other hand, the value of the `doInput()` expression is an action that has no meaning for people who manage inventories. The `doInput()` action is an artifact of the way we process inventories.

Here, the value 5 is wrapped within the action of reading an input. For our example of reading a file, the file itself is wrapped within the action of reading the file. However, we aren't done yet. How do we actually do computations with the value wrapped within that action? You still can't do `1 + doInput(5)` or `1 + open(PATH)`, since `doInput` and `open` are still actions. In order to answer this question and more, there are two functions we need to go over for monads to work right: *bind* and *return*.

2.2.1 Parts of a monad

Binding

The first question we have to answer is how can we modify or access the value wrapped by a monad, so that we can pass it to a function or do other things with it. What if there was a way to call a function on the value wrapped in a monad? Enter *bind*. In order to describe how bind works, let's go over an idea that doesn't work. Let's say a monad `M` has a method called `accessVal()`, such that to access the value wrapped in `M`, we can call `M.accessVal()`. However this really is no different from an expression. This thus leads to all sorts of messy time-dependencies that we are trying to avoid. `M.accessVal()` may be 9 at one time and `Hello, World` at another, there's no telling. However, we have just learned that a monad can be thought of an expression wrapped in an action. What if we can grab the expression contained in one monad, and wrap it in an action that does a useful computation with it? That is the essence of a bind function. You are essentially chaining actions together: one action is followed by another one. This is why it's called bind, the second action is *bound* to the first one. In Haskell, there are two bind operators, *bind*(>>=) and *sequence* (>>), as well as their backwards forms (=<< and <<, respectively.) Their type signatures in Haskell as as follows:

```
(>>=) :: m a -> (a -> m b) -> m b
(>>)  :: m a -> m b -> m b
```

What >>= is doing is taking a monad that contains a value of type `a`, passes it to another function that takes that value and returns another monad, which may have a different type from the input monad. >> does the same but it doesn't require an input from the previous monad in the sequence.

Returning

The next function a monad needs is `return`. This function is fairly straightforward. All it does is, when given a value, wraps it within a monad. Note that isn't like `return` found in languages like C++ or Python. It doesn't end program execution, it just wraps a value in a monad and returns that monad. In Haskell, this is the type signature of `return`:

```
return :: a -> m a
```

When given a value of type `a`, it wraps it within a minimal context and returns that monadic value. By minimal context, it means that `return` doesn't compute anything, it just returns the simplest monad that can wrap that value.

2.3 The Monad Laws

Monads are, at their heart, a mathematical construct. That means, like other mathematical constructs, they have their own *axioms* or *laws* so that their behaviours are predictable, and that we can reason about them.

2.3.1 The 1st Monad Law: Left Identity

I will state the law first and then explain how it works.

```
return x >>= f = f x
```

What this law is stating when you wrap a value `x` using `return` and pass that monad to `f`, you get the same result as applying `x` to `f`. Logically, if given a value and if `return` wraps it within the simplest monad possible for that value without doing anything to it, then giving that monad to a function shouldn't be different than just applying that value to the function [7].

2.3.2 The 2nd Monad Law: Right Identity

This is the second monad law:

```
m >>= return = m
```

What this law is saying is that when `return` is fed a monad, it just returns that monad. Remember that >>= takes a monad, feeds it into a function and then returns a new monad. Also recall that `return` just wraps a value in the simplest monad for its type without actually doing any computations on that value. Thus, when >>= feeds a monad into `return`, it should just be the same as the original monad. Note that the first two laws essentially concern `return` not doing anything to the values that are fed into it, but just collects those values [7].

2.3.3 The 3rd Monad Law: Associativity of >>=

The 3rd and final monad law is as follows:

```
(m >>= f) >>= g = m >>= (\x -> f x >>= g)
```

What the left-hand side is saying that a monad `m` is fed into a function `f`, and then that result is then funneled into function g `g`. The right-hand side is saying a monad `m` is fed into an anonymous function `x` (\x is the notation for an anonymous function in Haskell, it's similar to a Python lambda.) The result of this computation is then fed into `g`.

This may not seem to make much sense. However, this gets a little easier to understand if you convert this so that it uses <=<, the monadic composition operator (also known as a *left fish operator*), so let's take a quick detour to talk about monad composition. Here is <=<'s type signature:

```
(<=<) :: (Monad m) => (b -> m c) -> (a -> m b) -> (a -> m c)
f <=< g = (\x -> g x >>= f)
```

What this operator is essentially doing is *composing* two monadic functions. This is similar to normal function composition. If function `g` had type `a -> m b` and function `f` had type `b -> m c`, what <=< is doing is essentially returning a function of type `a -> m c`. The monadic argument gets fed between `g` to `f` [8].

Now it is time to bring this back to the associativity of >>=. Note that in the 3rd law, the monad is getting passed to `f`, and the result of that computation is then given to `g`. Hey, that sounds just like the monad composition we just talked about! The 3rd law can be re-written so that it uses the left fish operator like so [7]:

```
f <=< (g <=< h) = (f <=< g) <=< h
```

This is much nicer to look at than the original statement of the 3rd law. Essentially all that it's saying is that the nesting of operations doesn't matter for monads. You can evaluate the the two inner monads first, and that is the same thing as evaluating the two outer monads.

Do note that if one has to define a new monad from scratch, they are responsible for making sure the monad obeys these three laws.

2.4 Examples of monads in Haskell

To bring everything we have learned so far home, I will discuss three examples of monads in Haskell: `Maybe`, IO and `State`.

2.4.1 `Maybe`, I don't really wanna know...

In Haskell, a common construct is `Maybe`. In plain English, what `Maybe` does is to provide a way to represent computations that only have a range of outputs. `Maybe` is a way to represent computations that may not return a valid result. It is a way for values to have a notion of failure attached to them. I will show the code and then explain it.

```
instance Monad Maybe where
return x = Just x
   Nothing >>= f = Nothing
   Just x >>= f = f x
fail _ = Nothing
```

Note that all the parts of the monad needed are defined here. The `return` method in the `Maybe` monad returns only the value, and nothing else. Since `Maybe` is a value with a notion of failure, there needs to be two cases for >>=: one when the value is good and one for when it's not. In the first case, feeding `Just x` to a function is the same as applying that function on `x`. In the second, feeding `Nothing` to a function is the same as `Nothing` itself.

Let's see an example of how the `Maybe` monad works.

```
divide :: (Fractional a) => a -> a -> Maybe a
divide a 0 = Nothing
divide a b = Just $ a / b
```

When dividing a number, you either get another number, or nothing in the case of division by 0. Doing `divide 1 4` would give you `Just 0.25`. Doing `divide 5 0` would return `Nothing`. Since `Maybe` is a monad, we can feed it to functions using >>=. I will demonstrate that by making another function that makes use of `divide`:

```
reciprocalTimesTwo :: (Fractional a) => a -> Maybe a
reciprocalTimesTwo x = divide 1 x >>= return . (*2)
```

What this function is doing is computing the reciprocal of a number, and multiplying it by two. Note that it takes the result of `divide 1 x`, and feeds that value into the `return` monad, where that value is then doubled. Calling `reciprocalTimesTwo 0.1` would give you `Just 20.0`, while calling `reciprocalTimesTwo 0` would return `Nothing`.

2.4.2 E-I-E-IO

The next example to look at will be the IO Monad. In fact, monads were introduced into Haskell so that IO could be accomplished. However, it would be useful to figure out why Haskell needs monads to accomplish IO.

In Haskell, all functions are *pure*. This means that functions will always return the same value, no matter what. However, earlier in this chapter, we discussed how IO doesn't have a fixed value, that the user could input different things each time and a different output could result from each of those inputs. Another thing to note about Haskell is that it is *lazily evaluated*. This means that functions and values are only evaluated when they are needed to, which could be never. As you can imagine, IO that's lazily evaluated would make order of execution of a program a total nightmare.

Monads, as we have learned earlier, helps with both of those issues. Thus it makes perfect sense for IO to be a monad in Haskell. Let's take a quick example:

```
module Main where

import Data.Char (toLower, toUpper)
import Control.Monad

main = putStrLn "Express sarcasm here: " >> fmap applySarcasm
   getLine >>= putStrLn

applySarcasm (a:b:str) = toLower a : toUpper b : applySarcasm str
applySarcasm a = a
```

What this code does is, when given a string, returns the same string but in alternate case. For example, when given the string "`Awesome - you'll definitely be on the list.`", it will return "`aWeSoMe - yOu'lL DeFiNiTeLy bE On tHe lIsT.`" Let's analyse the monadic nature of IO using this simple program. We can start by looking at all types involved in the program.

```
main :: IO ()
putStrLn :: String -> IO ()
"Express sarcasm here: " :: [Char]
(>>) :: Monad m => m a -> m b -> m b
fmap :: Functor f => (a -> b) -> f a -> f b
```

```
applySarcasm :: [Char] -> [Char]
getLine :: IO String
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

We already know how >> and >>= work. Let's go over the other ones. Note that the entire program, `main`, is encapsulated as a single `IO ()` action. The closed brackets after `IO` means no value is available to pass on to another function. This is known as a *unit type*. Similarly, `putStrLn` takes a string and results in an `IO` action that prints that string to the terminal. Similarity to our `main`, there is no value returned from the `IO` action of printing on the screen. The string "`Express sarcasm here: `" is of type `[Char]`, a list of `Char`, which is the same as a `String`. `fmap` takes in a function of type `(a -> b)` to something of type `f a`, then returns something of type `f b`. Note that this function could be monadic. The function `applySarcasm` takes a `String`, and returns another `String`. The most interesting part here is `getLine`. Unlike `main` and `putStrLn` that don't actually return anything, `getLine` actually does. Note that `getLine` isn't actually a function. The type signature says says that it is an `IO String`. What does that mean? It means that `getLine` is actually an `IO` action that, when executed, will generate a string based on what the user inputs in the terminal. That `string` can then get passed to other `IO` actions using bind and `fmap`. Note that when `getLine` is used to generate a string, it's a monad since that string is wrapped in an `IO` action.

A simple way to view the `IO` monad is to think about `IO` actions *changing the state of the program* by doing input and output actions. An `IO` action of type `IO x` does some input and output actions changing the state, while also returning a value of type `x`.

2.4.3 Monadic `State` of Mind

The best way to illustrate why we need the `State` and how it works is through an example. Let's say you're trying to implement Milton-Bradley's classic game Connect Four:

```
data GameState = GameState
  { board :: A.Array HoleIndex HoleState
  , currentPlayer :: Player
  , generator :: StdGen
  }

data Player = RedPlayer | YellowPlayer

data HoleState = Empty | HasRed | HasYellow
  deriving Eq

type HoleIndex = (Int, Int)
```

There is a board, which is an array of `HoleStates`, which either have a red piece, a yellow piece or is empty. The `GameState` also keeps track of the current player, and has a random generator. Of course, other functions we would need are a function that would return a `HoleIndex`, change the game's generator. Essentially, this function selects a random move. We would then take our turn based on the selected move and pass the turn to our opponent. This function takes in the current `GameState`, but also changes it. Soon enough with enough turns, it will be a mess to keep track of which `GameState` we are currently on.

This is where the `State` monad comes in. The `State` monad wraps computations based on *getting and setting a global-state object*. Let's see what these methods look like:

```
get :: State s s
put :: s -> State s ()
runState :: s -> State s a -> (a, s)
evalState :: State s a -> s -> a
execState :: State s a -> s -> s
```

`get` just retrieves the current state. `put` replaces the current state with the given one. `runState`, when given a state, returns both the result of that computation AND the new state resulting from that computation. `evalState` is similar to `runState`, but it just returns the result of the computation. Similarily, `execState` just returns the final state.

Armed with this knowledge, we can now describe the game functions we discussed earlier.

```
chooseRandomMove :: State GameState HoleIndex
chooseRandomMove = do
  game <- get
  let openSpots = [ fst pair | pair <- A.assocs (board game), snd
     pair == Empty]
  let gen = generator game
  let (i, gen') = randomR (0, length openSpots - 1) gen
  put $ game { generator = gen' }
  return $ openSpots !! i
```

This function may seem complicated, but all it's doing is both returning the current `HoleIndex`, while changing the current `GameState`'s generator. It uses `get` to retrieve the current game state, modifies its generator, `puts` it back, and then returns the current `HoleIndex`. We are ready to describe the next function, which updates the game with the player's turn, and passes it to their opponent:

```
applyMove :: HoleIndex -> State GameState ()
applyMove i = do
  game <- get
  let p = currentPlayer game
  let newBoard = board game A.// [(i, holeForPlayer p)]
  put $ game { currentPlayer = nextPlayer p, board = newBoard }

nextPlayer :: Player -> Player
nextPlayer RedPlayer = YellowPlayer
nextPlayer YellowPlayer = RedPlayer

holeForPlayer :: Player -> HoleState
holeForPlayer RedPlayer = HasRed
holeForPlayer YellowPlayer = HasYellow
```

This function `gets` the current `GameState`, applies the player's move to it, `puts` it back and passes the turn while not outputting anything.

We can now combine these functions to make something that looks remarkably clean despite all that's going under the hood:

```
resolveTurn :: State GameState Bool
resolveTurn = do
  i <- chooseRandomMove
  applyMove i
  isGameDone

isGameDone :: State GameState Bool
isGameDone = do
  game <- get
  let openSpots = [ fst pair | pair <- A.assocs (board game), snd
     pair == Empty]
  return $ length openSpots == 0
```

There are obviously more functions that the game would need to be functional, but they all could use the `State` monad.

2.5 Just do it!

Haskell's `do` notation are just an alternative syntax for expressing monadic computations. It is particularly useful for `IO`-related things. I will use the `applySarcasm` example from the `IO` monad section. For reference, here it is again:

```
module Main where
```

```
import Data.Char (toLower, toUpper)
import Control.Monad

main = putStrLn "Express sarcasm here: " >> fmap applySarcasm
   getLine >>= putStrLn

applySarcasm (a:b:str) = toLower a : toUpper b : applySarcasm str
applySarcasm a = a
```

Using do notation, we can write:

```
module Main where

import Data.Char (toLower, toUpper)
import Control.Monad

main = do putStrLn "Express sarcasm here: "
          string <- getLine
          putStrLn (applySarcasm string)

applySarcasm (a:b:str) = toLower a : toUpper b : applySarcasm str
applySarcasm a = a
```

Note that in order to translate >>= and >>, do notation has the programmer assign a name to the passed value using <-. Instead of using fmap, we can just pass the name of the bound value to putStrLn. do notation can often make monadic computations easier to understand and to look at, but note that it isn't limited to IO.

2.6 Monad transformers, roll out!

We've seen what monads can do and how they're useful in Haskell. The next logical question to ask is "Can we roll up multiple monads into one?" Well dear reader, Haskell provides a way to do this! These are known as *monad transformers*. It's best to learn this through example, with this example being borrowed from [9]. Let's say you are the administrator of some website where users have to make an account, so you need a function that checks a given username:

```
checkUsrname :: IO ()
checkUsrname = do
 maybeUserName <- askUsrname
 case maybeUserName of
  Nothing -> print "Invalid user name!"
  Just (uName) -> uName
```

We can also have a function that requests a user to type in a new username as so:

```
askUsrname :: IO ()
askUsrname = do
  putStrLn "Please enter your username!"
  str <- getLine
  if length str > 5
    then return $ Just str
    else return Nothing
```

These functions are simple enough, but imagine if we add the capability to check stuff like passwords and email addresses? They would be a mess of nested `case` statements within the `IO` monad! But what if we could make it even simpler by defining a new monad that had qualities of both `IO` and `Maybe`? This is where *monad transformers* come in. What they do is *wrap* around another monad, and are usually parameterised by a monad. If we were to create a monad transformer for `Maybe`, it could look something like this:

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }

instance (Monad m) => Monad (MaybeT m) where
  return = MaybeT . return . Just
  x >>= f = MaybeT $ do
    v <- runMaybeT x
    case v of
      Nothing -> return Nothing
      Just y -> runMaybeT (f y)
```

Here, `m` could be any monad, in our case, `IO`. `runMaybeT` accesses the underlying monad. Note that since we are defining a new monad, we'll have to define `return` and `>>=`. `return` will just return `Just` wrapped within a `MaybeT`. Let's go through `>>=` step-by-step. First, `runMaybeT` will wrap `x` within `m (Maybe a)`. Then, we extract a `Maybe a` value. Then we use the `case` statement. If there is a `Nothing` value, we `return` that into `m`. Otherwise in the case of `Just`, we apply `f` on `y`, and wrap that within `runMaybeT` to put it back into `m`. We can summarise this by looking at our new bind's type signature:

```
(>>=) :: MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b
```

Note this looks similar to the type of regular `>>=`. Now we can use `MaybeT` to simplify our code from above:

```
(>>=) :: MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b
```

Now we can bring everything back home.

```
askUsrname :: MaybeT IO String
askUsrname = MaybeT $ do
  putStrLn "Please enter your Username!"
  str <- getLine
  if length str > 5
    then return $ Just str
    else return Nothing
```

The only difference between the old and new functions is that we wrap the result in `MaybeT`. Here is the new `checkUsrname`:

```
checkUsrname :: IO ()
checkUsrname = do
  maybeCreds <- runMaybeT $ do
    usr <- askUsrName
    return usr
  case maybeCreds of
    Nothing -> print "Couldn't login!"
    Just u -> login u
```

Now our functions have the inherent failure built into the `Maybe` monad! And since `MaybeT` is a monad itself, you could wrap that within another monad!

2.7 Conclusion

In this chapter, I went over what state is, how monads help solve the issues with state, how Haskell implements them, binding and returning, the monad laws, the `Maybe`, `IO`, and `State` monads, `do` notation and monad transformers. In the next chapter, we will discuss Miso, a Haskell front-end for JavaScript, and using the knowledge from these first two chapters, figure out how to play sounds.

Chapter 3

I'm sorry, but your code is in another castle!

I will first give a quick introduction to Miso, a Haskell front-end for JavaScript. I will go over the parts of one of Miso's code examples, explain how it works and then describe how all those parts fit together with relation to what we learned in the previous two chapters. Then I will modify this example to add sound, and then extend that example to play a sequence of sounds. Do note that throughout this chapter, I will be quoting both Miso's documentation [5] and showing code from a Miso example called `Mario.hs` [6].

3.1 What is Miso?

Before we discuss Miso, let's think about what a JavaScript app actually does. A JavaScript app on a website is *interactive*: it responds to changes in the environment, be it a mouse click, a key press, or changing of the window size. It is the responsibility of the app to reflect those changes and perform an action, like displaying a picture, writing some text on the screen or in this case, playing a sound, in a timely fashion. Now let's discuss Miso. According to their webpage:

> Miso is a small, production-ready, "isomorphic" Haskell front-end framework featuring a virtual-dom, recursive diffing / patching algorithm, event delegation, event batching, SVG, Server-sent events, Websockets, type-safe servant-style routing and an extensible Subscription-based subsystem. Inspired by Elm, Redux and Bobril. Miso is pure by default, but side effects (like XHR) can be introduced into the system via the Effect data type. Miso makes heavy use of the GHCJS FFI and therefore has minimal dependencies.

In short, Miso is essentially a front-end to create interactive web applications. It does this through interfacing with GHCJS, a Haskell to JavaScript compiler using FFI, "Foreign Function Interface". This means you can call JavaScript code in your Haskell program. This allows Haskell to be used as a web-development tool. Notice

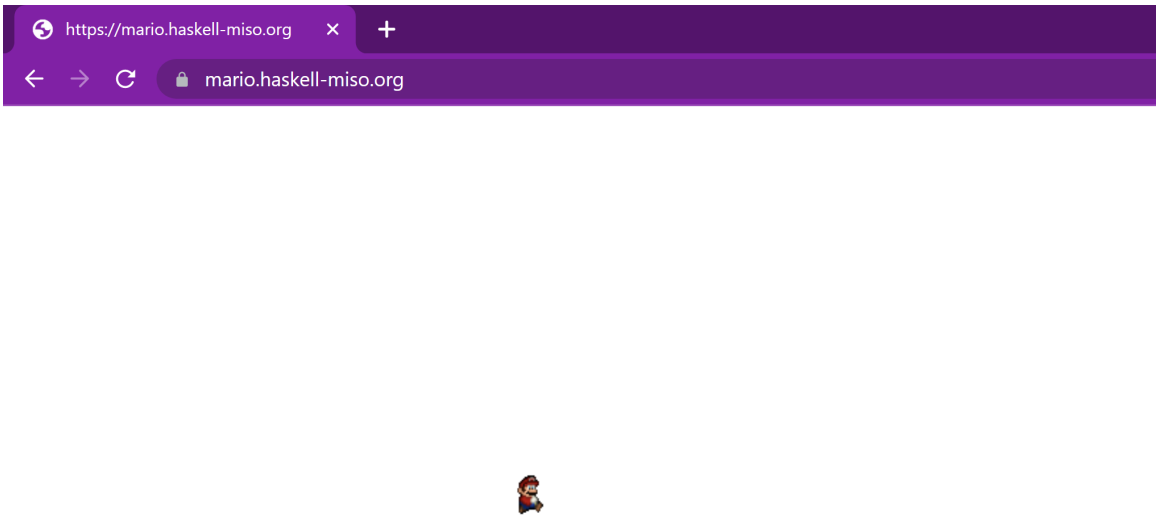


Figure 3.1: Miso's `Mario` example

also that Miso is *pure*, meaning its functions have no side-effects. However, they have implemented a datatype `Effect` that allows you to introduce them if you want. `Effect` becomes quite important soon, so stay tuned. If we look at the higher-level, what Miso is asking the programmer to restructure their JavaScript app in Haskell, using `Actions` that have an `Effect` on a state-tracking `Model`, and it interfaces with JavaScript in order to display the results of those computations. For a more detailed look into what Miso is, let's go over one of their code examples, `Mario`.

3.2 Caretaker of family Mario

This code example is quite straightforward. On a blank screen, you can control the Nintendo character Mario using the arrow keys, left to move left, right to move right, and up to jump. This is equivalent to a homework assignment in an intro to JavaScript class.
However, the Miso code for this runs 184 lines! What gives? In order to not traumatise the reader with 4 straight pages of Haskell, I will divide the program into its most important bits and explain how each of those work.

3.2.1 Preamble

Let's take a look at what Miso needs to import in order to make `Mario` possible.

```
import       Data.Bool
import       Data.Function
import qualified Data.Map as M
import       Data.Monoid

import       Miso
import       Miso.String

#ifdef IOS
import Language.JavaScript.JSaddle.WKWebView as JSaddle

runApp :: JSM () -> IO ()
runApp = JSaddle.run
#else
import qualified Language.JavaScript.JSaddle.Warp as JSaddle
#ifdef ghcjs_HOST_OS
runApp :: JSM () -> IO ()
runApp = JSaddle.run 8080
```

We need Haskell's `Bool`, `Function`, `Map`, and `Monoid` types, as well as `Miso` itself and `Miso`'s `String` type, known as `MisoString`. `JSaddle` is an interface between JavaScript and Haskell. Essentially you use it to call JavaScript functions from Haskell. Note the type of the function `runApp`. According to Miso's documentation, `JSM ()` is a monad that keeps track of JavaScript's execution context, or the environment the JavaScript code runs in. `runApp` essentially takes a `JSM` monad and converts it to an `IO` action, which we have learned in the last chapter is a monad. `runApp` isn't really imperative to understand how `Mario` works, but it's important to know its type.

At a higher level, what `runApp` is doing is serving as a bridge between the JavaScript world, and the Haskell world. The entire program is wrapped as a `JSM ()`. Then `runApp` converts that into `IO ()`, which Haskell understands and will execute, and displays the `Mario` app in the browser. In fact, this mirrors JavaScript itself, where the entire program acts as one large function.

3.2.2 `Action`

As we go on through the code, the word `Action` appears over and over again. Thus, we should understand what an `Action` is. Helpfully, `Action` is defined right at the start of the code! Here it is:

```
data Action
```

```
  = GetArrows !Arrows
  | Time !Double
  | WindowCoords !(Int,Int)
  | NoOp
```

`Action` is a datatype consisting of `GetArrows`, which is the arrow key handler, `Time`, which is elapsed time, `WindowCoords`, which is the size of the window, and `NoOp`, or no-operation. The exclamation marks in front of the types means that despite Haskell's lazy nature, they are always evaluated. As well, note that everything has a type except for `NoOp` Note that `Action` had to be defined for this specific program. Overall, `Action` is just a list of defined actions that have an effect on the program's state when the user interacts with the program in some way.

3.2.3 main

Now let's talk about `Mario`'s `main` function. Here it is, straight out of the `Mario` source code:

```
main :: IO ()
main = runApp $ do
   time <- now
   let m = mario { time = time }
   startApp App { model = m
                , initialAction = NoOp
                , ..
                }
  where
   update = updateMario
   view = display
   events = defaultEvents
   subs = [ arrowsSub GetArrows
          , windowCoordsSub WindowCoords
          ]
   mountPoint = Nothing
   logLevel = Off
```

First, note that `main` is using the `do` notation we discussed in the last chapter. First, `main` gets the current time using `now`, which is imported from JavaScript's Performance API. It then calls the `runApp` function we mentioned earlier. It now calls `startApp`. Here is its type:

```
startApp :: Eq model => App model action -> JSM ()
```

Firstly, what `Eq` means is that the `model` type has part of the `Eq` typeclass. In plain English, it means that the given `model` has an equality test. It takes in

an `App model action`, which in Miso, represents the starting point of the application. Finally, it returns a `JSM ()` monad. Thus, it makes sense that this is passed to `runApp`. `App model action` is a datatype that consists of a `model`, an `initialAction` that the model will do when the program starts, an `update` function that provides a way to change the `model`, a `view` function that displays the app on the screen, `events` that the program will listen for, `subs` (or *subscriptions*), which are a way for Miso to capture events as they happen, and a `logLevel`, which is an error message. Finally, `mountPoint` is the root directory of the *DOM* (document object model.) It is a way for the webpage to be represented so that the program can interact with and modify it. All `main` does is construct a `App model action`, and calls `runApp` on it.

Essentially, all `main` is doing is setting the stage of the program. It initialises a new `Mario model`, and declares the functions that are required to keep track of, and change the program's state. As well, it is responsible for reflecting changes to the `Mario model` on the browser via `display`. Next, let's look at what `Mario` defines a `model` to be.

3.2.4 `model`

Here is what a `model` looks like in `Mario`:

```
data Model = Model
   { x :: !Double
   , y :: !Double
   , vx :: !Double
   , vy :: !Double
   , dir :: !Direction
   , time :: !Double
   , delta :: !Double
   , arrows :: !Arrows
   , window :: !(Int,Int)
   } deriving (Show, Eq)

data Direction
  = L
  | R
  deriving (Show,Eq)

mario :: Model
mario = Model
   { x = 0
   , y = 0
   , vx = 0
   , vy = 0
   , dir = R
```

```
  , time = 0
  , delta = 0
  , arrows = Arrows 0 0
  , window = (0,0)
  }
```

A `Mario model` consists of an `x` position, a `y` position, an x-axis speed `vx`, a y-axis speed `vy`, a `Direction` (below the model definition, `Direction` is defined as a datatype with only two values: `L` and `R`), an elapsed time, the arrow key currently pressed, and the window coordinates. This is simply what a `Mario` object is to Miso. `Mario` contains the information that the program must track and modify in order to run. In other words, `Mario` is how the program tracks the *state*. Observe that like `Action`, `Model` is a datatype specific to this program.

If `main` declares the functions that can alter the state of the `App`, `Model is` the state of the program: it's what is altered when a user interacts with the `Mario model` as they press a key, or resize the window, what have you. Now that we have set the stage, it is time to examine the functions that make `Mario` tick and actually alter this `Mario model`.

3.2.5 updateMario

Now let's look at how the program updates `Mario` using the arrow keys. It does this using `updateMario`. Here it is as defined in `Mario.hs`:

```
updateMario :: Action -> Model -> Effect Action Model
updateMario NoOp m = step m
updateMario (GetArrows arrs) m = noEff newModel
  where
    newModel = m { arrows = arrs }
updateMario (Time newTime) m = step newModel
  where
    newModel = m { delta = (newTime - time m) / 20
                 , time = newTime
                 }
updateMario (WindowCoords coords) m = noEff newModel
  where
    newModel = m { window = coords }
```

Immediately, we notice that the type of this function:

```
updateMario :: Action -> Model -> Effect Action Model
```

It takes in an `Action`, a `Model` and returns the new `Effect Action Model` after that `Model` has been updated by the `Action`. This makes sense, because as

stated before the update function in the App model action datatype is of the type Effect Action Model. Now if the function isn't given an operation (i.e NoOp) and a model, it applies that model to a function called step, which we will describe later. Otherwise, if given arrow key inputs via GetArrows and a model, it uses Miso's smart constructor for a model with no new effects, noEff, and uses it to create a new Model called, well, newModel. It constructs newModel from m, but changes its arrows to reflect the ones passed to updateMario. It does something similar when the browser dimensions are modified as well. When updateMario receives a Time action, it uses a function called step, which we will look at shortly. It creates a new model with a new delta value and a new current time. The updateMario function just updates the Mario model based on the events given in the Action datatype.

At a higher-level, updateMario tells the program how to change the state (i.e. the Mario model when given operations from the user.) It does this by creating new Mario models that reflect the Effect those Actions had. This new Mario state is then fed to display, and then displayed on the browser via startApp, showing the user Mario moving or jumping. It is, in essence, the controller of the state of Mario. It handles all the necessary computations to make Mario an interactive program.

3.2.6 step

Here is the step function that was mentioned in updateMario:

```
step :: Model -> Effect Action Model
step m@Model{..} = k <# do Time <$> now
  where
    k = m & gravity delta
          & jump arrows
          & walk arrows
          & physics delta
```

First, notice the type signature. When given a Model, it returns an Effect Action Model. Now, note the new operator, <#. Here is its type according to the Miso docs:

```
(<#) :: model -> JSM action -> Effect action model
```

Essentially, this operator combines a model and an action wrapped in a JSM monad, resulting in an Effect action model. Armed with this knowledge, what step is doing is that it's recomputing the Model as time passes. It gets the current state of Mario, extracts what arrow keys are being pressed and Mario's current speed. Using that information, step then recomputes what speed Mario is walking at, his position in the air and the ground, and when he'll reach the top of his jump and begin

falling. It then creates a new `Model` to reflect those changes in the state. Now let's take a look at these other functions referenced in `step`!

3.2.7 `gravity`, `jump`, `walk`, `physics`

These are the helper functions responsible for modifying `Mario` from `Mario.hs`. They aren't of utmost importance to understand how this program, but here they are for completeness' sake:

```
jump :: Arrows -> Model -> Model
jump Arrows{..} m@Model{..} =
   if arrowY > 0 && vy == 0
    then m { vy = 6 }
    else m

gravity :: Double -> Model -> Model
gravity dt m@Model{..} =
  m { vy = if y > 0 then vy - (dt / 4) else 0 }

physics :: Double -> Model -> Model
physics dt m@Model{..} =
  m { x = x + dt * vx
    , y = max 0 (y + dt * vy)
    }

walk :: Arrows -> Model -> Model
walk Arrows{..} m@Model{..} =
  m { vx = fromIntegral arrowX
    , dir = if | arrowX < 0 -> L
               | arrowX > 0 -> R
               | otherwise -> dir
}
```

There is nothing much to say about these functions. They take a certain type of `Action` (be it `Arrows` or `Double`), the Mario `Model`, and returns a new `Model`. Note that in `step`, these functions are used to construct a new `Model`, `k`, which is passed on to `step` so that it can update the `Mario` model as time passes. Note that all of these functions return a new `Model` based on the changes on the `Actions` it receives. Now we have all the pieces to the puzzle!

3.3 Jigsaw Falling Into Place

There are a few other functions in the program that aren't necessary to figure out the functional principle (get it?) of the program. For example, there is a function responsible for displaying Mario on the screen, but there isn't much to talk about.

However, now that we've discussed the different ingredients of the program, how do they fit together, and how can we relate it to what we learned in the previous chapters? Firstly, note that the entire program is a monad, due to it being encapsulated in a `JSM ()` monad, as detailed in `runApp`. More important to note is how the program keeps track of state, which is monadic. It does this through `Model`. However, `Model` is just a datatype that holds certain attributes that can be modified by actions. Where is the monad? Note that both `updateMario` and `step` result in the type `Effect Action Model`. However, we have established that `Action` and `Model` are datatypes specific for each program. What about `Effect` however? Let's delve deeper.

3.3.1 Effect

What is an effect? According to the Miso documentation:

> An effect represents the results of an update action.
>
> It consists of the updated model and a list of subscriptions. Each `Sub` is run in a new thread so there is no risk of accidentally blocking the application.

An `Effect` is the new `Model` after it's been updated by an `Action`. A `Subscription` is essentially an event listener: it listens for certain actions and does certain things based on them. So an `Effect` contains a `Model`, and a list of event listeners that listen for actions that are side-effects... this sounds suspiciously like a monad! Indeed, if you look at the Miso docs again, `Effect` is monadic, with its own definitions of `bind` and `return`!

```
(>>=) :: Effect action a -> (a -> Effect action b) -> Effect
   action b

(>>) :: Effect action a -> Effect action b -> Effect action b

return :: a -> Effect action a
```

3.3.2 Connecting all of this to monads

Now that we have a handle on how `Mario` was implemented in Miso, what does it have to do with monads? Recall the `State` monad we covered in the previous chapter. The `State` monad wraps computations that either retrieve or alter a program's state. Note that since `Effect` wraps the updated model and the list of event listeners, `Effect` essentially keeps track of the program's state! As well, we have noted that `Effect` is monadic. Keeping this in mind, one cannot help but compare `Effect Action Model` to the `State` monad we talked about earlier! The `State` monad is wrapping computations that set a global state object. Meanwhile, `Effect Action`

`Model` consists of the `Model`, a list of `Actions`, and the `Effect` that updated the `Model`. In other words, the `Model` is the global state object, and the `Effects` are the computations that sets the `Model`, based on the received `Actions`. The `updateMario` function takes these all in all of these and returns the final `Effect Action Model`. Now that we know how `Mario` works, let us go over the arduous journey of me trying to add sound to `Mario`. Enjoy!

3.3.3 Adding sound to `Mario`

The first thing I did was take a look at the Miso documentation. Recall that Miso can use FFI to call JavaScript code in Haskell. In order to facilitate this, you would first declare the FFI code as so:

```
foreign import JavaScript unsafe "$r = new Audio($1);" loadSound
   :: JSString -> IO JSVal

foreign import JavaScript unsafe "$1.play();" playSound :: JSVal
   -> IO ()
```

Note that this takes in JavaScript code and converts it to types that Miso can recognise. As mentioned, this is done via GHCJS, the Haskell to JavaScript compiler. Now that we have the functions defined, I first tried to add this new code to `jump` as such:

```
jump :: Arrows -> Model -> Model
jump Arrows{..} m@Model{..} =
   if arrowY > 0 && vy == 0
    then do
         sample <- (loadSound (toMisoString "Sound sample URL"))
         m { vy = 6 }
         playSound sample
    else m
```

However, this totally destroyed the program like some destruction caused to an apartment. The reason it wasn't working was because Haskell couldn't properly resolve the type of this function. After some thought, I realised that since this function is pure, it cannot execute side-effects, which meant I had to wrap it up in a monad, specifically `Effect`! Thus, the result was this:

```
jump :: Arrows -> Model -> (IO Action, Model)
jump Arrows{..} m@Model{..} =
   if arrowY > 0 && vy == 0
    then do
      let action = do
         sample <- (loadSound (toMisoString "Sound sample URL"))
```

```
          playSound sample
        pure (action, m { vy = 6 })
    else (pure NoOp, m)
```

Even though this was all well and good in theory, I would have to rebuild the entire program and all of its functions to account for this new return type, like building a CD case. There's got to be a better way! Thus, I began investigating. Recall that "jumping" is an `Action` that has an `Effect`, a computation that changes the Mario `Model`. Also notice that playing a sound is also a type of action. Thus, I reasoned that the `Action` datatype should contain some mention of a sound. But what type should it be? Given that path of the sound is a URL, it should be some type of `String`. Miso has a facility for a `String` that can be converted to a JavaScript `String`, or `JSString`. Thus, this is the resulting `Action` datatype:

```
data Action
  = GetArrows !Arrows
  | Time !Double
  | WindowCoords !(Int,Int)
  | PlaySound MisoString
  | NoOp
```

Now the function calls to the FFI need to be changed in order to make their types match with `MisoString`. This is easily done via:

```
foreign import JavaScript unsafe "(new Audio($1)).play()" play ::
   MisoString -> IO ()
```

This takes a `MisoString` and results in an `IO` action. Now `updateMario` needs to be, well, updated. Here is the updated portion that takes into account `PlaySound`:

```
updateMario (PlaySound url) m = m <# do
#ifdef __GHCJS__
  play url >> pure NoOp
#else
  pure NoOp
#endif
updateMario (GetArrows arrs) m = newModel <# do
   pure $
     if arrowY arrs > arrowY (arrows m)
     then (PlaySound "Sound sample URL")
     else NoOp
  where
   newModel = m { arrows = arrs }
```

Recall from the earlier definition that `play` is of type `MisoString -> IO ()`.

However, the constructor for an `Effect Action Model` with a single effect, `<#`, needs actions of type `JSM Action`. Remember earlier when I mentioned that `JSM` is a monad? That comes into play here! `NoOp` was defined in the `Action` datatype as an action that does nothing. `pure` lifts that `NoOp` value into an applicative functor, functors that support function application. However, if you look at the definition of a monad in GHCI:

```
Prelude> :info Monad
class Applicative m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
```

So `Monad` inherits from `Applicative`, so every monad is also an applicative functor. Thus, `pure` lifts `NoOp` into the `JSM Action` monad, and that monad is fed the `IO` action of playing the sound! This entire `JSM Action`, along with the `Mario model` is neatly packaged into an `Effect Action Model` thanks to `<#`. With this, compiling the program and running it results in the classic Mario experience: when he jumps, he makes a sound!

3.3.4 Phew!

We have added a jump sound to `Mario`! Even though we could savour the taste of this victory (which tastes like a nutritious breakfast of sausage and eggs), we can't rest on our laurels just yet. Now, I will attempt the task of getting Miso to play a sequence of sounds.

3.4 A queue (as in line, not the data structure) of things

Thankfully, we've done most of the work in the previous section. It doesn't take much more work to get Miso playing a sequence of sounds. There exists a nifty constructor in Miso:

```
batchEff :: model -> [JSM action] -> Effect action model
```

This is very similar to `<#`, except it takes in a list of `JSM Actions`, and constructs an `Effect` out of that list. `batchEff` essentially allows a `Model` to juggle a lot of things one after the other. However, we don't need to use this, although it will work. Remember in the last section where we wrapped playing the sound in `NoOp`. Well, what if you can wrap playing a sound in that? Turns out you can! We can still use `<#`, but change `updateMario` to look like this:

```
updateMario (PlaySound url) m = m <# do
#ifdef __GHCJS__
  play url >> threadDelay 1000000 >> play url2 >> pure NoOp
#else
  pure NoOp
#endif
```

`threadDelay` is a function from Haskell's `Control.Concurrent` library. All it does is pause the current thread for the given number of microseconds. We just need to add `import Data.Concurrent` to the top of the program. What is the meaning of this long chain of sequencing? We just continually feed one action to the next, until it is essentially one giant action all wrapped in `pure NoOp`! Now we need to refactor our `playSound Action` so that it can handle multiple sounds. Since the sound URL is a `MisoString`, it makes sense to use a list of `MisoStrings`:

```
data Action
  = GetArrows !Arrows
  | Time !Double
  | WindowCoords !(Int,Int)
  | PlaySound [MisoString]
  | NoOp
```

However, `updateMario` still only takes a single URL. This is easy enough to rectify. You just have to pass `PlaySound` a list of URLs instead of just a single URL in our jump sound example as so:

```
updateMario (PlaySound [url, url2]) m = m <# do
#ifdef __GHCJS__
  play url >> threadDelay 1000000 >> play url2 >> pure NoOp
#else
  pure NoOp
#endif
```

We have to do the same for when the up key is pressed and it engages the `PlaySound` action:

```
updateMario (GetArrows arrs) m = newModel <# do
   pure $
     if arrowY arrs > arrowY (arrows m)
     then (PlaySound ["Sound sample URL 1", "Sound sample URL 2"])
     else NoOp
  where
    newModel = m { arrows = arrs }
```

Thus, when you press up, not only does `Mario` jump, it also plays the two sounds!

3.5 At the end of the day, that is it

In this chapter, I gave an overview of Miso, discussed its `Effect Model Action` based state-tracking, connected it to monads and used that knowledge to make a jump sound play as well as sequence of chords. Upcoming is the conclusion of my thesis, where I answer the burning question: Can Haskell be used for live music composition?

Conclusion

I first started this thesis with the intention of figuring out if there was a way for improvisational music composition in Haskell. However, a lot of the tools required for this are from the late 90s and early 2000s, and don't necessarily work too well anymore in 2023. So instead I decided to look into this legacy code as a way to find out more about the intricacies of Haskell, like its type system and monads. Thus, from the cremated remains of my original topic rose my eventual topic. The first two chapters of this thesis aim to equip the reader with enough of this knowledge to render it comprehensible.

In my quest to find something to base this thesis on, I stumbled upon Miso, a Haskell front-end for JavaScript. Although Miso doesn't have the tools (at least to my knowledge) to undertake what I wanted, it was still enough for me to glean a new thesis topic: the effectiveness of using Haskell for interactive web applications that play sound. Was I successful in this endeavor? I did eventually find a way to make the `Mario` demo play a sequence of predefined sounds, thanks to its ability to FFI from JavaScript.

Now to address the goal of this thesis. Can Haskell be utilised for live music performance and composition? From my work, I've come to the conclusion that with the tools described in this thesis, it isn't very feasible to. Miso is only a compiler, so it can't really help on the live coding front. Some sort of live Haskell interpreter would be needed so that composers could type out music and hear it being played on-the-fly and in real time. Although my findings didn't lead to live music composition, there are still avenues left to explore it. One possibility is to update the tools outlined in *Haskell School of Expression* and *Haskell School of Music* so that they work in 2023. It may even be possible to port this code so that it works in conjunction with Miso.

Appendix A

The full **MarioSequence** program

```
{-# LANGUAGE CPP #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RecordWildCards #-}
{-# LANGUAGE MultiWayIf #-}
{-# LANGUAGE BangPatterns #-}
module Main where

import       Data.Bool
import       Data.Function
import qualified Data.Map as M
import       Data.Monoid

import       Miso
import       Miso.String

import       Control.Concurrent

#ifdef IOS
import Language.Javascript.JSaddle.WKWebView as JSaddle

runApp :: JSM () -> IO ()
runApp = JSaddle.run
#else
import qualified Language.Javascript.JSaddle.Warp as JSaddle
#ifdef ghcjs_HOST_OS
runApp :: JSM () -> IO ()
runApp = JSaddle.run 8080

foreign import javascript unsafe "(new Audio($1)).play()" play ::
   MisoString -> IO ()

#else
import       Network.Wai.Application.Static
```

```
import qualified Network.Wai as Wai
import qualified Network.Wai.Handler.Warp as Warp
import       Network.WebSockets

runApp :: JSM () -> IO ()
runApp f =
   Warp.runSettings (Warp.setPort 8080 (Warp.setTimeout 3600
      Warp.defaultSettings)) =<<
      JSaddle.jsaddleOr defaultConnectionOptions (f >> syncPoint)
         app
   where app req sendResp =
         case Wai.pathInfo req of
           ("imgs" : _) -> staticApp (defaultWebAppSettings
              "examples/mario") req sendResp
           _ -> JSaddle.jsaddleApp req sendResp
#endif
#endif

data Action
  = GetArrows !Arrows
  | Time !Double
  | WindowCoords !(Int,Int)
  | PlaySound [MisoString]
  | NoOp

spriteFrames :: [MisoString]
spriteFrames = ["0 0", "-74px 0","-111px 0","-148px 0","-185px
   0","-222px 0","-259px 0","-296px 0"]

main :: IO ()
main = runApp $ do
   time <- now
   let m = mario { time = time }
   startApp App { model = m
                , initialAction = NoOp
                , ..
                }
  where
   update = updateMario
   view = display
   events = defaultEvents
   subs = [ arrowsSub GetArrows
          , windowCoordsSub WindowCoords
          ]
   mountPoint = Nothing
   logLevel = Off
```

```
data Model = Model
   { x :: !Double
   , y :: !Double
   , vx :: !Double
   , vy :: !Double
   , dir :: !Direction
   , time :: !Double
   , delta :: !Double
   , arrows :: !Arrows
   , window :: !(Int,Int)
   } deriving (Show, Eq)

data Direction
  = L
  | R
  deriving (Show,Eq)

mario :: Model
mario = Model
   { x = 0
   , y = 0
   , vx = 0
   , vy = 0
   , dir = R
   , time = 0
   , delta = 0
   , arrows = Arrows 0 0
   , window = (0,0)
   }
updateMario :: Action -> Model -> Effect Action Model
updateMario NoOp m = step m
updateMario (PlaySound [url, url2]) m = m <# do
#ifdef __GHCJS__
  play url >> threadDelay 1000000 >> play url2 >> pure NoOp
#else
  pure NoOp
#endif
updateMario (GetArrows arrs) m = newModel <# do
   pure $
     if arrowY arrs > arrowY (arrows m)
     then (PlaySound ["https://tinyurl.com/33ktyv9r",
        "https://tinyurl.com/mumcpy2y"])
     else NoOp
  where
    newModel = m { arrows = arrs }
updateMario (Time newTime) m = step newModel
  where
```

```
    newModel = m { delta = (newTime - time m) / 20
                 , time = newTime
                 }
updateMario (WindowCoords coords) m = noEff newModel
  where
    newModel = m { window = coords }

step :: Model -> Effect Action Model
step m@Model{..} = k <# do Time <$> now
  where
    k = m & gravity delta
          & jump arrows
          & walk arrows
          & physics delta

jump :: Arrows -> Model -> Model
jump Arrows{..} m@Model{..} =
    if arrowY > 0 && vy == 0
      then m { vy = 6 }
      else m

gravity :: Double -> Model -> Model
gravity dt m@Model{..} =
  m { vy = if y > 0 then vy - (dt / 4) else 0 }

physics :: Double -> Model -> Model
physics dt m@Model{..} =
  m { x = x + dt * vx
    , y = max 0 (y + dt * vy)
    }

walk :: Arrows -> Model -> Model
walk Arrows{..} m@Model{..} =
  m { vx = fromIntegral arrowX
    , dir = if | arrowX < 0 -> L
               | arrowX > 0 -> R
               | otherwise -> dir
    }

display :: Model -> View action
display m@Model{..} = marioImage
  where
    (h,w) = window
    groundY = 62 - (fromIntegral (fst window) / 2)
    marioImage =
      div_ [ height_ $ ms h
           , width_ $ ms w
```

```
        ]
        [ nodeHtml "style" [] ["@keyframes play { 100% {
           background-position: -296px; } }"]
        , div_ [ style_ (marioStyle m groundY) ] []
        ]

marioStyle :: Model -> Double -> M.Map MisoString MisoString
marioStyle Model {..} gy =
  M.fromList [ ("transform", matrix dir x $ abs (y + gy) )
             , ("display", "block")
             , ("width", "37px")
             , ("height", "37px")
             , ("background-color", "transparent")
             , ("background-image",
                "url(https://mario.haskell-miso.org/imgs/mario.png)")
             , ("background-repeat", "no-repeat")
             , ("background-position", spriteFrames !! frame)
             , bool mempty ("animation", "play 0.8s steps(8)
                infinite") (y == 0 && vx /= 0)
             ]
  where
    frame | y > 0 = 1
          | otherwise = 0

matrix :: Direction -> Double -> Double -> MisoString
matrix dir x y =
  "matrix("
    <> (if dir == L then "-1" else "1")
    <> ",0,0,1,"
    <> ms x
    <> ","
    <> ms y
    <> ")"
```

Bibliography

[1] Barry Burd. Understanding monads. a guide for the perplexed, Nov 2017. URL https://www.infoq.com/articles/Understanding-Monads-guide-for-perplexed/.

[2] Paul Hudak. *The Haskell School of Expression: Learning Functional programming through multimedia*. Cambridge Univ. Press, 2008.

[3] Paul Hudak and Donya Quick. *The Haskell School of Music: From Signals to Symphonies*. Cambridge Univ. Press, 2018.

[4] John Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2): 98–107, 1989.

[5] David Johnson. Miso documentation. URL https://haddocks.haskell-miso.org/.

[6] David Johnson. Mario, 2020. URL https://github.com/dmjio/miso/tree/master/examples/mario.

[7] Miran Lipovaca. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, USA, 1st edition, 2011. ISBN 1593272839.

[8] Mark Seemann. Kleisli composition, Apr 2022. URL https://blog.ploeh.dk/2022/04/04/kleisli-composition/.

[9] Various. Haskell, 2022. URL https://en.wikibooks.org/wiki/Haskell.