The Remote Monad

By

Justin Dawson

Submitted to the Department of Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Committee members

Dr. Andy Gill, Chair

Dr. Perry Alexander

Dr. Prasad Kulkarni

Dr. Bo Luo

Dr. Kyle Camarda

Date defended:

The Dissertation Committee for Justin Dawson certifies
that this is the approved version of the following dissertation :

The Remote Monad

Dr. Andy Gill, Chair

Date approved:

Abstract

Remote Procedure Calls are an integral part of the internet of things and cloud computing. However, remote procedures, by their very nature, have an expensive overhead cost of a network round trip. There have been many optimizations to amortize the network overhead cost, including asynchronous remote calls and batching requests together.

In this dissertation, we present a principled way to batch procedure calls together, called the Remote Monad. The support for monadic structures in languages such as Haskell can be utilized to build a staging mechanism for chains of remote procedures. Our specific formulation of remote monads uses natural transformations to make modular and composable network stacks which can automatically bundle requests into packets by breaking up monadic actions into ideal packets. By observing the properties of these primitive operations, we can leverage a number of tactics to maximize the size of the packets.

We have created a framework which has been successfully used to implement the industry standard JSON-RPC protocol, a graphical browser-based library, an efficient byte string implementation, a library to communicate with an Arduino board and database queries all of which have automatic bundling enabled. We demonstrate that the result of this investigation is that the cost of implementing bundling for remote monads can be amortized almost for free, when given a user-supplied packet transportation mechanism.

Acknowledgements

Isaac Newton once penned a letter which contained the phrase “If I have seen further it is by standing on the shoulders of Giants”. In performing this research and writing this dissertation, I have definitely been standing on the shoulders of giants as well as being held up and supported from all sides.

First and foremost, I would like to thank my advisor, Andy Gill, for providing me with guidance and a pathway to succeed when I had been stumbling through the first year and half of graduate school without an advisor or any kind of direction. To say that I have learned a lot from Andy is an understatement as I am finishing a PhD in functional programming when 4-5 years ago I had never performed any research nor written a non-trivial program in a functional language!

I’d like to thank thank Jeremy Gibbons, Simon Marlow, Garrett Morris, Edward Kmett and Conal Elliot for the remote monad and natural transformation conversations with myself and other members of the functional programming group at KU. These conversations helped us nail down the particulars about the remote monad.

I’d also like to thank my peers, mentors and instructors past and present at KU: Perry, Prasad, Bo, Paul, Mike, Adam, Mark, Ryan, Jason, Drew, David, Surya, Lu, Jamie, April and Arunabha. I have learned much from each of you. The past 6 years have been bearable and at times enjoyable because of our conversations, commiserations, classes and who can forget the ping pong games. Thanks expecially to Mark - for fighting side by side as we learned about the remote monad together and to Ryan and Aleksander for their work on the blank canvas benchmarks.

Most importantly I want to thank my wife Tresann, and my children Ellie and Hank for loving and supporting me and putting up with my late night hackathons and general absentmindedness. I love you guys! I also would like to thank my parents, brother and sister and all my inlaws for believing in me and for being wonderful examples to me. I really couldn’t do it without you all!

This material is based upon work supported by the National Science Foundation under Grant No. 1350901. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The whole purpose of high-level programming languages is to have abstractions that allow a user to describe the behavior of a program without having to think about the low-level intricacies of the underlying hardware. The latest programming languages are closer to human language than ever before which, in conjunction with the performance of the latest hardware, allows us to solve problems that we could not have dreamed of solving years ago because of their complexity.

As more and more companies offer both paid and unpaid access to their cloud computing platform, supercomputers or other resources that would otherwise be inaccessible to the average user, using Remote Procedure Calls [Nelson, 1981; Birrell & Nelson, 1984] has become a valid strategy to improve the performance of computationally expensive tasks. One of the downsides to using these remote services is the degradation of performance caused by the added network cost associated with invoking the remote system. Depending on the complexity of the task and the status of the surrounding network, this network cost can cancel out any performance gained from using the remote service. This is especially the case when less intense computations are now unable to take place locally because of where the data is located. The network cost quickly adds up as the frequency of these smaller operations increases.

Traditionally, as a program evolves over time, the goal of the programmer transitions from reasoning about and coming up with a solution to a problem, to taking the solution and making it as performant as possible. These increases in performance come from using more efficient data structures, coming up with more efficient algorithms to solve a task, caching frequently used data or leveraging remote machines with more horsepower to name just a few. A common problem that occurs from improving the performance of code is that the code becomes obfuscated as code with clear reasoning gets replaced with code that runs faster, making subsequent edits prone to contain more bugs and wasted development time as programmers try to remember how their code works and why it was done in that way. Ideally, what we would like to have are performance gains without sacrificing the clarity of the core logic of our program.

Batching remote service calls has been identified and used in many different languages and situations to amortize this network cost [Shakib & Benson, 2001; Bogle & Liskov, 1994; Microsystems, 1988]. The purpose of this dissertation is to bring automatic batching to the Haskell programming language in a principled way by modifying the underlying applicative functor and monad data structures, allowing us to have better-performing code without requiring a large amount of refactoring.

Thesis Statement

> A monad of remote procedure calls acts as an easy to construct, yet network intensive, staging mechanism. The thesis of this dissertation is that we can modify the monad to construct a lightweight staging mechanism that automatically transforms many sequences of network-intensive monadic remote procedure calls into batches of these calls. In this dissertation, we investigate two complementary technologies that make this lightweight remote monad possible: methods for systematic bundling of certain sequences of remote calls, and a technique for structuring our transformation components as composable network stacks.

It was not obvious at the start of this research, what kind of strategy would be the best for factoring a monad into transmittable chunks. We have examined different possible bundling strategies and propose that using applicative functors for the packets is the best bundling strategy.

This dissertation focuses on bundling remote primitives together specifically in Haskell, but we are able to do this bundling due to the first-class control structures used in Haskell. The techniques herein could be applicable in other languages that also have these types of control structures.

The initial goals of this dissertation's research were to:

- investigate different batching techniques, using functional programming idioms such as monad and applicative functors.
- explore the surrounding space of remote execution, including the subtleties that come from moving a programs control flow to a remote location.
- create a modular and principled framework for remote execution in Haskell
- demonstrate the applicability of the framework to tackle a wide range of problems through a variety of case studies.

1.1 Contributions

As a consequence of these goals, this dissertation makes the following contributions:

Investigation of different bundling strategies

Remote requests can be categorized as asynchronous or synchronous or even read-only queries, with each one leading to different bundling strategies. Bundling requests lowers the number of network transactions, thus amortizing the network overhead of sending data across the network (Section 5.1).

Modular framework

Our framework uses a series of natural transformations connected together in composable network stacks, allowing different layers to be replaced, added or removed. Different bundling strategies can be exchanged as well as the transmission mechanism (e.g. use HTTP instead of sockets). Intermediate computations, type conversions or other marshaling of data can be added with relative ease (Section 5.2.2). By simply having a generalized modular design, we are able to have reusable units of code that can be used for different applications, e.g. If we have an application that communicates with a remote server through sockets, then when writing a new application, we can reuse the socket communication layer of the network stack and just change the endpoint we are trying to reach.

Remote Monad Problem Domain

The remote monad principles have many possible applications and are not restricted to one type of problem. To demonstrate this we have built a framework, which embodies the remote monad principles to tackle a wide variety of use cases. In Section 6, we discuss case studies that:

- implement the JSON RPC protocol (remote-json)
- focus on being more efficient by sending byte strings instead of plain text as well as focus on encoding remote failure (remote-binary)
- interact with an Arduino board (haskino)
- send JavaScript to be executed in a web browser for a drawing application (blank-canvas)
- interact with a Plist database (PlistBuddy)

Applicative functors are ideal for transmission

Due to the structure of applicative functors, we can send multiple calls in one batch to a remote service and later combine the results locally. Since monads are a superset of applicative functors, by factoring the monad into applicative functors where possible, we then have an API that can bundle monadic actions resulting in a lower network cost than if we naively send the requests. By looking at the properties of the remote primitives, we can increase the types of monadic statements that can be factored into applicative functors when compared with looking at the syntactical structure alone.

1.2 Outline

Additionally, we will examine both remote procedure calls (RPCs) and the Haskell language as a whole and how to model RPCs in Haskell as background concepts and foundations for the research on remote monads and remote applicative functors (Section 2). We will later look at some of the observations that we have made about remote primitives (Chapter 4) followed by the challenges and design considerations that went into creating the remote libraries and the bundling strategies that were conceived (Chapter 5). Section 5.3 discusses some possible extensions to the framework and how to extend the framework in general. In Chapter 6, we will view some use cases for the framework, as well as share some performance measurements of the different bundling strategies in Chapter 7. Finally, the related work and conclusion are found in Chapters 8 and 9 respectively.

Chapter 2

Background

We will begin this background section with a look at remote procedure calls and some of their optimizations, as well as some other relevant information for batching requests. We will then visit the idea of first-class control, which was mentioned in the introduction as a requirement needed to apply the techniques that we will use to automatically bundle requests. First-class control means that the control flow of a program can be seen, touched, and passed around to various functions. For this dissertation, we will be focused on how this can be done in Haskell, as it meets these requirements, but any language that has first-class control can be used with similar techniques. We will take a look at Haskell as a whole, discuss how effects are cordoned off from the rest of the language and then later, we will look at how composing these effects allows us to create a framework with automatic bundling. This section is meant to give the reader a taste of how Haskell works and some of its features but it is in no way a fully comprehensive look into Haskell. A more in-depth look at Haskell is left up to the reader.

2.1 Remote Procedure Calls

A remote procedure call (RPC) [Nelson, 1981] is a means of causing a remote entity to do work on the local machine's behalf. To execute a remote procedure call, the user will communicate to the server which function they would like the server to execute along with any required arguments. Assuming that the request is formatted correctly, the server will reply with the result of the computation.

Getting RPCs working in this day and age is pretty basic with all of the supporting libraries and only requires us to have a handful of things in place:

- A remote machine listening for requests
- A local machine that has knowledge of the remote API and protocol to be used
- A network transmission mechanism

2.1.1 Why execute something remotely?

There are a number of reasons that we would want our code to run remotely instead of on our local machine. In the market of laptops, for example, there has been a shift from power-hungry machines with top of the line graphics cards and RAM, 3-4 GHz multi-core processors, and terabytes of storage, which could rival the benchmarks of a normal desktop computer, to netbooks which have processors that rarely break 2.0 GHz and hard drives with a maximum size of 128 GB. These netbooks are made possible because of remote procedure calls. Applications and other processes that are too heavyweight for these netbooks reside remotely on machines that can handle the load. Even outside of netbooks, simulations and parallel programming that could take days to run on a high-end desktop locally, can now be executed in hours or even minutes when given to a cluster of compute nodes for execution.

If we do not have enough space on our local machine, we can have the data stored on a remote server and use RPCs to access the data. In some cases, the data must reside remotely and cannot be transmitted to our system. The requirement to store data remotely could be because of security concerns or it could be that there is not a valid representation of the remote data on the local machine. This might be because of language limitations of the program running on the local machine or even hardware limitations. GPUs have been found to be great at running matrix calculations. If I am needing to run extensive calculations on matrices but my machine does not have a GPU, I can still create code that will execute the calculations on a shared machine that does have a GPU. We can imagine wanting to examine an intermediate representation of the data being used on the GPU, but that representation would not mean anything on the local machine that does not have such hardware. Imagine the difficulty of trying to serialize and send some remote structure to our local machine when the structure is riddled with pointers to memory on the remote system. These situations make it unwieldy to house information locally and require RPCs to be used.

2.1.2 What does an RPC look like?

High-level languages make RPCs look just like function calls. In the Java Remote Method Invocation (RMI) [Waldo, 1998], for example, the remote server is viewed as an object with a set of methods that can be called. The fact that network communication is occurring with these calls is hidden from the user's view.

The main difference between regular function calls and RPCs is that the arguments are restricted to objects that can be serialized and have an equivalent representation on the remote machine along with results that can be serialized and have a representation on the local machine. If the client and server are written in the same language, then the serialization and deserialization of the user data types are straightforward. If the client is written in a different language from the server, a protocol must be established. In this dissertation, we will look at some case studies that cover both situations.

As far as what is actually transmitted in an RPC, we have different formats depending on the protocol. Here is an example that uses the XML-RPC specification:

```
<?xml version="1.0"?>
<methodCall>
   <methodName>circleArea</methodName>
      <params>
         <param>
            <value><double>2.41</double></value>
         </param>
      </params>
</methodCall>
```

Though the format for the different protocols are different, see Section 6.2 to compare with the JSON-RPC format, most if not all of the components are present in each of the RPC protocols:

- Method name to be called
- List of parameters (possibly with type information)
- Protocol version (if there is more than one version)

RPCs are widely used and have been at the center of much research to execute efficiently[Spector, 1982; Shakib & Benson, 2001; Bogle & Liskov, 1994; Gifford & Glasser, 1988; Liskov & Shrira, 1988]. One of the most widely used optimizations is to support the batching of requests. With multiple calls in a single request, the protocol has to decide if the responses are going to be returned piecemeal or if the order is maintained or, in some cases, remote servers will require a unique identifier with the request, allowing the server to tag the response with an identifier. There are even some protocols that allow the server to decide to execute requests in parallel when a batch of requests is received.

A number of errors could occur from attempting to make a remote procedure call. We could have a malformed request error, which would occur when the agreed upon protocol is not implemented properly. We would also get an error if we are requesting a function that does not exist on the server, is an internal function or otherwise unavailable to be called.

These errors are in addition to the everyday errors that occur with any programming in general.

In this dissertation, we will assume that we have a valid connection with the remote server which excludes any connectivity errors such as “remote host not found” from the list. Any remote procedure call frameworks will also need to handle these kinds of errors. Because Haskell is considered a strongly typed language, the malformed request errors can be lessened if not completely averted by the type-checker at compile time, when given a well-defined specification for valid requests. We will also have to take a look at what error handling looks like when a program that used to be running locally, is moved to a remote location.

2.2 Haskell

Haskell is a strongly typed functional language. It contains all of the standard built-in types that are in most, if not all, modern programming languages. Haskell has integers, doubles, characters, strings, booleans, user-defined data types, etc. We then have other data types that build upon the basic data types: lists of items of a certain type, tuples used to combine data types together, hash maps, user-defined data types, etc. And finally, we have functions that can take any number of arguments made from the data types already discussed or even other functions themselves and result in a data type or a function itself. Let’s look at a binary function `add` below:

```
add :: Int -> Int -> Int
add x y = x + y
```

A function name followed by `::` delineates the type of a function in Haskell. Each parameter is separated by `->` where the last type is the resulting type for the function when fully evaluated, also known in imperative languages as the type of the returning result. In the function above, `add` will take two integers as input parameters and result in an integer. Calling a function is as simple as stating the function name and passing arguments separated by spaces.

```
> add 5 6
11
```

```
> add (add 1 4) (add 2 3)
10
```

One of the neat things about Haskell functions is that we can partially apply arguments to a function, yielding a new function that requires one less argument than the original. Without any extra machinery, we can create a function that is built upon `add` but only takes one argument and increments that value by 1, simply by partially applying one argument to `add`:

```
    inc :: Int -> Int
    inc = add 1
```

Notice that we went from a function of type `Int -> Int -> Int` to a function of type `Int -> Int`. Instead of looking at `add` as a function that takes two integer arguments and results in an integer, we can also say that `add` is a function that when given one integer returns a function that when given another integer as an input argument will result in an integer. The idea of partially applying arguments to functions is known as currying and becomes useful in starting with a generalized function which can then be specialized as we partially apply arguments.

Unlike most imperative languages, Haskell uses lazy evaluation. What this means is that instead of evaluating expressions when they are assigned to variables, the expressions are evaluated only when the program cannot proceed further without evaluation. Even when the expressions are evaluated, it is only to a certain point. For example, if we had a variable bound to a list of complex, time-consuming operations and we just need to get the length of the list, then we can view the number of operations with actually executing them. Thus, the time it takes to get the length of a list does not depend on what is being held in the list. Another advantage of using lazy evaluation is having functions that use part of an infinite structure without running out of memory. Iterating through a list becomes a view of the current item and a single chunk of work that represents the next item and all the rest.

This method of programming, along with the idea of not having variable reassignment, is what makes functional programming functional. We have functions that can be composed together and executed in different orders without changing the program behavior or having side-effects.

2.2.1 Haskell vs Mainstream Imperative Languages

Haskell differs from a standard imperative programming language, like Java or C, in a number of ways. Mainstream imperative programs are made up of a sequence of expressions or procedures which are executed sequentially and where each statement is able to change the state of a program. Reordering statements would wreak havoc on the logic of an imperative program as variables are reused and can take on different values at different points in the program. Take a simple loop, from Java, as an example:

```
int val = 0;
val = 5;

for (int i = 0; i < 10; i++){
  System.out.println("i =" + i);
  methodCall(i,val);
}
...
```

At each iteration of the loop, the variable `i` has a new value, between 0 and 10. We are also guaranteed that the loop will occur only after the line 'val = 5' is executed.

On the other hand, Haskell treats variables as labels for immutable expressions. What this means is that there is no reassignment of variables and we can, therefore, replace any instance of a variable, with its value, and can expect the same result. This approach causes Haskell and other similar functional programming languages to be much closer to the mathematical foundations.

If we were looking at an algebraic expression or equation such as:

$$2\pi rh + 2\pi r^2$$

In mathematics, it would not make sense to have `r` equal to one thing on the left side of the '+' and then be something else on the other side, nor is there any syntax that we can use to reassign a value part way through the expression.

2.2.2 Purity and Side Effects

But there are times that we need our programs to not be purely functional (side-effect free), especially when interacting with the outside world (reading/writing to file systems, communicating through networks, creating random numbers based on the system clock, etc). Pure functional languages allow us to reason about a program and even have mathematical proofs. We get a strong guarantee of determinicity when we prohibit our functions from accessing the outside world or from affecting program state which can be examined later in the program. The main problem with this is that we cannot solve many problems in the real-world without having impure functions, or in other words, functions with side-effects.

In Haskell, side-effects are explicit. Functions which have side-effects reflect this in their type. Domains in which side-effects can occur are called `Monads`. We will discuss the use of monads as a method for control flow later in this section, for now, we will just talk about them in the context of monads being able to house side-effects. The main monad in Haskell that is used for side-effects is the input-output monad, also known as the `IO` monad. Functions that do not have side-effects are classified as pure functions. Below is an example of two functions. One which has side-effects and one which is pure:

```
addPure :: Int -> Int -> Int
addPure x y = x + y
```

```
addIO :: Int -> Int-> IO Int
addIO x y = do
      putStrLn "Writing to file"
      writeFile "tmp.txt" "side-effect"
      return (x + y)
```

Just by looking at the type of `addIO` we can see that it could (and in this case does) have side-effects because we see `IO Int` whereas with the `addPure` function we are simply dealing with integers. Because we have restricted `addPure`'s access to the outside world, we know that when given the same two parameters we will always end up with the same result. As soon as a function is labeled as having access to `IO` we have to look at it as having an extra parameter which is the current state of the world surrounding the program, which is ever-changing.

Traditionally, functional programmers follow something similar to the principle of least privilege, where functions are written to be as pure as possible and are only written in the context of `IO` or some other monad when the effects are necessary. This leads to code that, thanks to its determinicity, is less prone to errors and consequently easier to debug. Attempting to write to a file in the `addPure` function would lead to an error at compile time since it can only be executed in the `IO` monad. As shown in the `addIO` function above, we use the `do` keyword followed by some monadic actions to operate in the monad. Because these monadic actions can have side-effects, these actions are run sequentially. The `return` at the end is not the same as seen in the imperative mainstream languages. Instead of ending the function and returning the result, `return` in Haskell is packaging the pure value as a monadic action.

2.2.3 First Class Control

We will now take a step back to look at a couple classes that define the control flow for a program in Haskell. By implementing these classes, we can define a way to combine actions within a data type. We will first look at `Functor`s which allow us to access values that are stored within some context, and we will later see some of the other first-class mechanisms found in Haskell.

Functors

A functor can be looked at as a value wrapped in a context where the `fmap` function is used to modify the value while maintaining the context. Here is the class definition:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

As an example of what a context around a value can be, the context that is present in the `Maybe` data type gives the notion of whether a computation has failed or not. Another way of thinking about contexts is as a box, it can hold something or it can be empty, with the two constructors called `Just` and `Nothing` respectively.

```
data Maybe a = Just a | Nothing
```

The `a` in the type shows that Maybe is polymorphic and can be a box of `Int`, `String` or any other type. Now if we wanted to use our normal integer functions to add 10 to a `Maybe Int` like this:

```
> 10 + (Just 5)
```

we would end up with a type error because `+` only knows how to work over numbers, not a number wrapped in a context, in this case, a `Maybe`. We can use `fmap` to allow us to lift a function that normally works over numbers to now work over numbers that are held within a context and keep the result within the context. Here is what that lift looks like for `Maybe`:

```
instance Functor Maybe where
   fmap f (Just a) = Just (f a)
   fmap f Nothing  = Nothing
```

If we have an "empty box" or a context which shows that a failure has occurred, then we keep that context, otherwise, if things have worked as expected thus far, then we will apply the function to the present value.

```
> fmap (10 +) (Just 5)                > fmap (10 +) Nothing
Just 15                               Nothing
```

We can imagine having a program that catches a divide by zero error and instead of just throwing an exception, we mark the computation as invalid by having the result be `Nothing`, ignoring any subsequent operations on the value.

But to be considered a true functor in the theoretical sense (and not just have an instance declared), an instance of `Functor` needs to follow the two Functor Laws. These laws and the future laws that we will discuss concerning first-class control in Haskell all have a general theme. The first-class control flow mechanisms should not modify the user's data but should just be acting as plumbing to connect functions and values. In the same way that we want plumbing in our homes to cause water to be accessible throughout the building but we do not want the pipes to cause a change in our water.

Law 1. *fmap id = id*

The function used in conjunction with `fmap` is the only thing that should modify the functor, `fmap` should not cause any side effects. The identity function, `id`, makes no change to arguments that are given to it. When used in conjunction with `fmap`, the argument and context remain unchanged.

Law 2. *fmap (f . g) = (fmap f) . (fmap g)*

This law states that if we use `fmap` with the composition of two functions, then it is the same as applying the first function with `fmap` and later applying the second function with `fmap` on the result.

If we test the first law with `Maybe` we get:

```
> id (Just 6)
Just 6
> fmap id (Just 6)
Just 6
```

The result is the same as the original value. To demonstrate the second law we have a function `even` that will return `True` if the input is even and `False` otherwise, and another function `inc` will increment its argument by 1.

```
> inc 4
5
> even 5
False
> fmap (even . inc) (Just 4)
Just False
> fmap even (fmap inc (Just 4))
Just False
```

We see that we end up with the same result. The instance of `Functor` for `Maybe` is not surprising. It is simply applying a function inside the `Just` wrapper. Let's look at an example of a structure that implements the `Functor` class but does not adhere to the laws.

```
data Counter a = Counter (a,Int)
  deriving Show

instance Functor Counter where
    fmap f (Counter (x,i)) = Counter (f x, i + 1)

newCounter :: a -> Counter a
newCounter a = Counter (a,0)
```

In `Counter` we have an integer that gets incremented whenever fmap gets called. This could be useful to store a counter for how many times this value has been modified. Unfortunately, this does not keep either of the laws:

```
-- Law 1 test fmap id = id --
> let original = newCounter 5
> original
Counter (5,0)
> fmap id original
Counter (5,1) -- Counter (5,1) != Counter (5,0)
-- Law 2 test  fmap (f .g) = fmap f (fmap g) --
> original
Counter (5,0)
> fmap (id . id) original
Counter (5,1)
> fmap id  (fmap id  original)
Counter (5,2) -- Counter (5,1) != Counter (5,2)
```

The main take away from this section is that we want ways of holding values and applying functions within a context but not have any side-effects come from those mechanisms.

Monads

The monad is an element taken from category theory that is based on functors mapping a category into itself. In Haskell, we can look at them as a class that we can implement to combine actions together using bind (>>=) and lift values into the monad using return.

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

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

The `Maybe` version of a monad is almost trivial as the left side of a bind (>>=) is just an unboxing of the Maybe type and application to the function on the right-hand side. This is trivial in the sense that it does not capture the implications of effects and when they occur. Since monads are able to have side-effects, we can imagine the left side of a bind is executing the monadic action including any accompanying side-effects before passing the resulting value to the function on the right side of the bind. A series of operations combined with bind leads to a sequential chain of operations, executing and passing the result to the next action in the chain. There is also a second version of bind (`>>`) that ignores the result of the left side. This operator is used when we would like the side-effect of an operation without caring about its result. The example below takes some input from the user and then prints out a greeting.

```
putStrLn "Enter Name: " >> getLine >>= \ x -> putStrLn ("Hello " x ++ "!")
```

The monad notation is not the easiest to read and is not much better than having a list of function applications:

```
function1 (function2 arg1 (function3 (function4 arg2)))
```

This issue was remedied when do-notation was introduced to improve the usability of monads:

```
func :: m (Int,Int)
func = do op1
          op2
          x <- (op3 :: m Int)
          y <- (op4 :: m Int)
          return (x,y)
```

Do-notation is just syntactic sugar (equivalent to the long handed `>>=` notation listed above) that allows the programmer to have each monadic action on a new line after the keyword `do`. In do-notation, any time a result is not bound to a variable, `>>` is used, otherwise `>>=` is used to store variables and continue on. Once again, `return` is not used in the same sense as a function return in other languages. In this case, `return` lifts `(x,y)` from `(Int,Int)` to be `m (Int,Int)`. The example above using do-notation syntax would be translated to:

```
func = op1 >> op2 >> op3 >>= (\ x -> op4 >>= (\ y -> return (x,y)))
```

Just as there were laws with the `Functor` class there are laws about how monads should behave:

Left Identity

`return a >>= f` $\equiv$ `f a`

Right Identity

`m >>= return`$\equiv$ `m`

Associative

`(m >>= f) >>= g` $\equiv$ `m >>= \lam x -> f x >>= g`

The first two laws govern the way that `>>=` works in conjunction with `return`. If we think of `return` as wrapping up a value and `>>=` as unwrapping a value and passing it to a function, then it makes sense to see that wrapping a value and immediately unwrapping it before passing it on is the same as just passing the value to the function (described in the Left Identity law). Likewise, if we are unwrapping a result just to wrap it back up, then its the same as leaving it wrapped (Right Identity law). The main idea to take away from these laws is that `return` and bind are just used as combinators and do not modify any of the underlying data.

Monads appear to be the most commonly used control structure in Haskell for sequentiality and side-effects but there are other structures as well. The one that seems to be gaining traction in the Haskell community is applicative functors.

Applicative Functors

The applicative functor is another of the first-class structures that is capable of handling the control flow in Haskell but the differences between monads and applicative functors enable us to apply different optimizations as we will later see in this dissertation.

```
data Functor f => Applicative f where
  pure :: a -> f a
  <*> :: f (a -> b) -> f a -> f b
```

The applicative `pure` has the same type as the monad `return` where we are lifting a value into the applicative functor (or monad in the case of `return`) without any side-effects. It turns out that `Monad` subsumes the `ApplicativeFunctor` meaning that to create an instance of `Monad` we need to first have an instance of `ApplicativeFunctor`. The default for the `return` in the `Monad` is actually the `pure` function that was defined in the `ApplicativeFunctor` instance.

Here are the laws for applicative functors:

Identity

`pure id <*> v` ≡ `v`

Homomorphism

`pure f <*> pure x` ≡ `pure (f x)`

Interchange

`u <*> pure y` ≡ `pure ($ y) <*> u`

Fmap

`fmap  f x` ≡ `pure f <*> x`

Composition

`pure (.) <*> u <*> v <*> w` ≡ `u <*> (v <*> w)`

The Identity Law says that the applicative operators will not modify any of the data, they only apply the function to the argument. The Homomorphism Law says that if we had to lift both the function and the argument to the applicative functor data type, then it is the same as if we made the function call and then lifted the result. The Interchange law gives us the assurance that the order in which things are placed does not change the result. For those newer to Haskell the `($ y) f` is used to say that given a function `y` will be applied as an argument to `f` and is equivalent to `f y`. The Fmap Law defines the relationship between the applicative operators and fmap. And finally, the Composition Law says that we can compose our applicative functor operators in the same way that we compose regular functions.

One of the difficulties with using the applicative functors in the past was just like the monad, the syntax got in the way and it was difficult to write readable programs that were connected through applicative functors. In 2016, however, Marlow et. al. [Marlow et al., 2016] introduced do-notation for applicative functors using a GHC extension. This new extension causes the do-notation to use applicative operations where possible but otherwise, it will use the monad operations. This gives applicative functors the convenience of do-notation and monads the benefit of using applicative functor operations where possible. If we had the following do-notation with ApplicativeDo extension enabled we would have:

```
    do
      x <- A
      y <- B
      z <- C
      return (f x y z)
  (\(x, y, z) -> f x y z) <$> A <*> B <*> C
```

But if any of the operations depend on a previous result then we would have to incorporate a bind:

```
do
  x <- A
  y <- B x
  z <- C
  return (f x y z)

  (\(x,y) z -> f x y z) <$> (A >>= \ x->  B x >>= \ y ->  return (x,y)) <*> C
```

This GHC extension takes place at compilation time to be able to check to see if a bound variable is used in any of the subsequent calls or if they can be combined using applicative operators instead of the monadic bind. Further discussions of other situations when applicative operations can be used in place of binds are left as an exercise for the reader.

One of the benefits of using an applicative functor instead of a monad is that the arguments connected by `<*>` are independent, which becomes a natural place for parallelism as long as the arguments are able to be run simultaneously. We are also always able to compose Applicative functors, whereas to combine monads we would require the monad to be a monad transformer. Monad transformers are able to be built on top of one another, each with a special execution function to run each monad in turn. Monad transformers are discussed in greater detail in Section 5.4.

Alternatives

Thus far we have discussed the first-class control mechanisms in Haskell, but we have not yet covered how we can handle errors. The `Alternative` class is an applicative functor with an added operation that gives us a way to capture errors and execute a backup command. This is similar to a try-catch expression where `empty` is defining the error.

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

Once again, the easiest way to see this operator in action is to look at the `Maybe` monad because it has a clear way of us seeing that a failure occurred. Since anything that is an instance of `Alternative` has to have an instance of `Applicative` we can add a couple numbers together using applicative operators. Let's look at the case where one of our numbers was invalid upon retrieval, from a user or some other input, resulting in the following expression.

```
pure (+) <*> Just 5 <*> Nothing <|> Just 0
```

In this case, we are attempting to add 5 to a number that was invalid. Instead of just resulting in a `Nothing` we are able to result in 0 and continue with the computation. The `IO` instance uses exceptions to signal failure. Because `IO` has an `Alternative` instance, we can catch the exceptions using `<|>`. Let's say we want to read a variable from a configuration file but we want to gracefully handle errors such as the file not existing or the variable not being present in the file. In this scenario, we can perform code that may throw an exception in either situation and use `<|>` to catch them and use our default value instead.

```
parseVar "port" <*> readFile local.conf" <|> pure 8000
```

The `Alternatives` behavior is easily seen in the `Maybe` instance:

```
instance Alternative Maybe where
  empty = Nothing
  Nothing <|> p = p
  Just x <|> _ = Just x
```

or in more formal terms we have the following laws:

`empty <|> x` $\equiv$ `x x <|> empty` $\equiv$ `x x <|> (y <|> z)` $\equiv$ `(x <|> y) <|> z`

Which show us that a failure (`empty`) on the left side causes us to use the right side of `<|>` and ignore it if `empty` occurs on the right side as well as showing that `<|>` is an associative operator. Now that we have discussed the control structures and one way of handling errors, we will need to think about what the consequences are of moving these structures remotely and the bundling strategies that take advantage of these control structures (Chapter 5).

Chapter 3

RPCs in Haskell

Because this dissertation is focused on using Haskell to perform RPCs in an efficient way, this chapter will look into how we can create RPCs in Haskell. Here we will discuss how we can create a Domain Specific Language (DSL) using generalized algebraic data types (GADTs) to model RPCs in Haskell and give us the foundation to begin our remote monad work.

3.1 Domain Specific Languages

In programming, there is a large range of general purpose languages such as Java, Haskell, C, JavaScript, and Python. These languages are called "general purpose" because of the broad types of problems that can be solved using these languages, the features of the language are not specialized for problems within a certain domain. On the other side of this coin, we have Domain Specific Languages such as `SQL` for database interaction, `HTML` and `CSS` for webpage creation, `make` to build software, and `lex` and `yacc` to analyze and parse text for compiler creation, to name a few. Each of these languages is meant to solve problems in a single restricted domain.

Take a look at CSS as an example. CSS has a singular purpose of changing the look and feel of elements in webpages. A programmer cannot try to solve just any problem with

CSS because it is not suited for things outside of this domain. DSLs are a nice way to compartmentalize code. Imagine we have a web project which uses JavaScript, HTML and CSS. Now if we had a bug with the way something looks, we know that we should first look in our CSS code. If we have an erroneous file being created by our web application, then we know that the CSS and HTML code are not to blame but that the bug most likely lies in our JavaScript code.

There are two categories of DSLs: external DSls and internal DSLs which are also known as embedded DSLs. External DSLs can have their own syntax and structure whereas an embedded DSL is limited by what can be represented in the host language. The downside of using an external DSL is that with the custom syntax, types, etc comes the need to build our own parser and compiler whereas the embedded DSL can take advantage of the compiler, data types and structures of the host language for free.

An RPC can be looked at as an extremely limited DSL. We can only do things that the server allows, anything else is rejected. What we can do is create a DSL to model the remote capabilities and when we evaluate the expressions, we are making a remote procedure call to the listening server. Because we want to do some rapid prototyping, we will do an embedded DSL with Haskell as our host language.

3.1.1 Embedded DSL in Haskell

In Haskell, there are a few ways of structuring computation, which is separated from the actual execution of the computation. We can build up a computation into a well defined structure, and later execute the structure to obtain our result. We can look at a math equation as such a structure that can be executed:

$$3 * (5 + 1) - 8/2$$

Following the standard order of operations (PEMDAS) we can evaluate the equation to be 14. PEMDAS is one interpretation of the above equation but maybe we want to change the rules and interpret the equation in a different way, PEASMD. This would have us do the multiplication and division at the very end giving us a resulting value of -3. A third interpretation could be to rewrite the expression as text:

```
"three times open-parens five plus one close-parens minus eight divided by two"
```

This practice of separating the underlying data with its interpretation is very common in programming. If we look at any application that has an underlying database, the User Interface is simply an interpretation of the data stored in the database. As different versions of the application become available, especially UI changes, we can see the changes simply as a new interpretation of the data.

There are two types of embedded DSLs, namely shallow DSLs and deep DSLs. As we build up a computation we can gradually evaluate the expressions as we come across them (shallow), or we can build up an interim data structure, in the host language, that represents the entire expression to later be evaluated. We will discuss the difference between these two approaches in Section 3.2.

In seeing this common separation of the computation from the interpretation, the driving question for this research was then, "Why not build the computation locally and then send it remotely to be executed?". We can take a program and build up the structure of the computation and then instead of applying a standard interpretation which would occur locally, our new interpretation is to interface with a remote system to execute the structure. When speaking of remoteness, we can consider machines connected by a network or even on the local machine but outside of the local runtime of the program.

If we look at a simple teletyping example from one of our papers [Dawson et al., 2017], we can see two functions: one that obtains a character from the user, and one that prints a character to the screen.

```
getChar :: IO Char
putChar :: Char -> IO ()
```

Instead of directly performing the action, we can create a data structure that mirrors our two functions with smart constructors:

```
data R :: * -> *
  GetChar :: R Char
  PutChar :: Char -> R ()

getCharR :: R Char
getCharR = GetChar

putCharR :: Char -> R ()
putCharR = PutChar
```

At this point we represent the computation in the `R` data type, and we can then create a run function that can interpret `R` in some monad of our choice for execution.

```
runR :: R a -> IO a
runR (GetChar) = getChar
runR (PutChar c) = putChar
```

Now to get this running on a remote system, we would just need a way to combine actions, serialize them on the local machine, transmit the serialized action, and then have a way of deserializing and executing the function remotely.

What we have just described are the components of a remote procedure call. Once again, this is not a new concept and has been done in most languages, including Haskell[Bringert, 2016; Rupp, 2016; Clark, 2018].

We will first look at causing a program that normally executes locally to execute remotely (Section 3.2), and then we will look into modeling the capabilities of an existing remote entity and using RPCs to invoke those functions (Section 3.3).

3.2 Making a Local Program Remote

Haskell's algebraic data types give us a natural way to break our program into modular pieces and to move the execution out of our local system. Because of the modular design, we will be able to add and remove different layers of computation, swap out the transmission mechanism or even have an interpretation where we bundle requests in order to create different behaviors.

Let's look at a trivial example of a calculator and show how we can break it up for remote execution. Here we have a simple calculator that can add, subtract, and multiply.

```
add :: Int -> Int -> Int
add x y = x + y

sub :: Int -> Int -> Int
sub x y = x - y

mult :: Int -> Int -> Int
mult x y = x * y

-- 5 + (8 - 2) * 2
calculationExample1 = add 5 (mult (sub 8 2) 2)

main:: IO ()
main = print calculationExample1

> main
17
```

Currently, the above code is just running everything locally and computing the answer immediately. To make this code more modular and prepare it for remote execution, we can separate the building of the expression with the evaluation by using an algebraic data type. This algebraic data type will have a constructor to mirror each of calculator functions as well as a type which will hold a single number.

```
data Expr = I Int | Add Expr Expr | Sub Expr Expr | Mult Expr Expr
  deriving Show

instance Num Expr where
  (+) = Add
  (-) = Sub
  (*) = Mult
  fromInteger = I . fromIntegral  -- take a regular number into Expr

add:: Expr -> Expr -> Expr
add = Add

sub :: Expr -> Expr -> Expr
sub = Sub

mult :: Expr -> Expr -> Expr
mult = Mult

eval :: Expr -> Int
eval (I x) = x
eval (Add x y) = eval x + eval y
eval (Sub x y) = eval x - eval y
eval (Mult x y) = eval x * eval y

calculationExample1 = add 5 (mult (sub 8 2) 2)

main :: IO()
main = do print calculationExample1
          print $ eval calculationExample1
> main
Add (I 5) (Mult (Sub (I 8) (I 2)) (I 2))
17
```

After introducing `Expr`, our `add`, `sub`, and `mult` functions are no longer performing computations when called, but instead they build up the computation into an `Expr`. We use the `Num` instance for convenience as it allows us to have our example unchanged. One of the goals of our framework is to change the underlying components in a way that is invisible to the programs built on top or to require very little change. Without the `Num` instance, our calculation example would be forced to be of the following form:

```
calculationExample1' = add (I 5) (mult (sub (I 8) (I 2)) (I 2))
```

To show the usefulness of creating the structure of the computation, consider what happens when we take our equation step by step through GHCI where `Calculator1.hs` contains the calculator without structure and `Calculator2.hs` contains the calculator using `Expr`:

```
> :l Calculator1.hs
> let a = sub 8 2
> let b = mult a 2
> let c = add 5 b
> a
6
> b
12
> c
17
> :l Calculator2.hs
> let a = sub 8 2
> let b = mult a 2
> let c = add 5 b
> a
Sub (I 8) (I 2)
> b
Mult (Sub (I 8) (I 2)) (I 2)
> c
Add (I 5) (Mult (Sub (I 8) (I 2)) (I 2))
> eval a
6
> eval b
12
> eval c
17
```

Instead of losing the intermediate steps, we were able to add upon our current expression. And now we can add a different interpretation. Let's say we want our calculator to show its work, so we add a new evaluation function `evalS` to show the string version of the computation:

```
evalS :: Expr -> String
evalS (I x)      = show x
evalS (Add x y)  = "(" ++ evalS x ++ " + " ++ evalS y ++ ")"
evalS (Sub x y)  = "(" ++ evalS x ++ " - " ++  evalS y ++ ")"
evalS (Mult x y) = "(" ++ evalS x ++ " * " ++ evalS y ++ ")"

main = do print $ evalS calculationExample1
          print $ eval calculationExample1
```

Running our new `main` with both eval functions yields the following results:

```
> main
"(5 + ((8 - 2) * 2))"
17
```

We have successfully stored our computation as a structure that can be evaluated. Now to get it running on a remote system we need to:

- serialize the computation
- send the serialized computation to a remote system
- remotely perform deserialization, evaluation and serialization of responses
- locally receive and deserialize the remote system's response

In Haskell, each of the serialization and deserialization steps is already modular in the form of creating serialization instances for the data type. In the case above, we are able to derive the `Show` instance to serialize the computation to a string. We can likewise deserialize `Expr` by deriving an instance for `Read`.

```
> let transmit = id -- place holder for transmission function
> let e = 5 + 2 :: Expr
> e
Add (I 5) (I 2)
> let s = transmit (show e) -- serialization and mock transmission
> s
"Add (I 5) (I 2)"
> let e' = read s :: Expr  -- remote deserialization
> e'
Add (I 5) (I 2)
```

In Section 5.2.1, we will see how the deserialization and serialization are natural transformations from our data type, in this case from `Expr` to `String` and back again. If we look at the above steps as natural transformations, we can then see that we can chain these together to form a composable network stack. It is within this network stack that we can add different mechanisms around the transport mechanism to bundle the requests through the network in a principled way.

3.3 Modeling RPC with GADTS

In the previous section we looked at taking a program that normally runs locally and causing it to run on a remote server. Here we will delve a little deeper by looking at connecting to an already-existing server.

3.3.1 GADTs

Generalized algebraic data types (GADTs) [Peyton Jones et al., 2006] are used in domain specific languages and become useful for type safety. We can build up a computation and the type system will help us ensure that we get the correct resulting type when we evaluate the computation.

It is a common practice in Haskell to structure code in these GADTs and then have an evaluation function that will interpret or execute the GADT. In many applications, the evaluating function is a natural transformation from the GADT to the IO Monad. The benefits of this structure is the ability to change the execution of the GADTs by simply changing the evaluating function. We saw this with our calculator example where we evaluated to a string vs evaluating to a number. We can look at another simple algebra GADT and its evaluation function:

```
data R where
   Say         :: String -> R ()
   Temperature ::           R Int
   Uptime      :: String -> R Double

runR :: forall  a . R a -> IO a
runR (Say s)       = print s
runR (Temperature) = return 23
runR (Uptime s)    = getUptime s
```

In creating an `R` type we can allow any user to create an `R` and we know that we will be able to handle it. The user is unable to create an invalid `R` data type such as a `Say` with a `Double` or some other data type, we can only have a `String` as the payload for a `Say`.

The type checker will keep the user honest and we will have type safety in our program at compile time. Likewise, the type checker will keep the evaluation function from evaluating `Temperature` to be a `Double` or any other data type that is not `Int`.

3.3.2 HTML5 Canvas Example

As an example, we will look at connecting to a remote server that has access to an HTML5 Canvas. The HTML5 Canvas object has a wide array of capabilities, which are enumerated in Section 6.1, but for this section we will only access the subset of operations necessary to draw rectangles, set the color and set line width.

Using a GADT to model these capabilities we have:

```
data JS a where
  --           x1     y1     x2     y2
  FillRect  :: Int -> Int -> Int -> Int -> JS ()
  DrawRect  :: Int -> Int -> Int -> Int -> JS ()
  SetColor  :: Text -> JS ()
  LineWidth :: Int -> JS ()
```

Now that we have modeled the procedure calls that we will be sending to our remote server, we need a way of connecting these `JS` actions together, and to transmit the work so we can draw a group of items.

To make life easier, we will introduce some smart constructors to build the `JS` constructors to make it appear as function calls instead of building up a data type with a constructor. This will also let us validate the information. In the HTML5 Canvas, we are able to draw at any xy coordinates where the origin is found in the top left-hand corner of the canvas. Any negative coordinates would then be off of the screen. With these smart constructors we can force our drawings to be visible or any other requirement that we might have.

```
fillRect :: Int -> Int -> Int -> Int -> JS ()
drawRect :: Int -> Int -> Int -> Int -> JS ()
setColor :: Text -> JS ()
lineWidth :: Int -> JS ()
lineWidth
    | val >= 1  = LineWidth val
    | otherwise = error "line width must be greater than or equal to 1"
```

Now that we are armed with these smart constructors, we need to have a transmission function to do the work of sending our data to the remote server as an RPC. We will call our transmission function `send` and give it information about our server as well as the remote function that we would like to call.

```
send :: Context -> JS a -> IO a

example1 :: Context -> IO ()
example1 ctx = do
      send ctx $ setColor "blue"
      send ctx $ fillRect 10 10 20 10
      send ctx $ setColor "black"
      send ctx $ lineWidth 3
      send ctx $ drawRect 10 10 20 10
```

We can imagine that the `send` function would extract how to communicate with the server and then serialize the `JS` object to something that conforms with the RPC protocol that the server is using. The code above draws a blue rectangle with a black border on the remote server but it is very tedious to have to put `send ctx` in front of each of the items we want to execute, but more importantly, our code is not running very efficiently by sending each of these items individually to the server. As it stands, if we were looking at these RPCs as calls that were "remote" in the sense of being outside the runtime environment of the current program, but still taking place on the local machine, then the lack of efficiency would not be very noticeable. But the moment we have to boot up and tear down communications on a network for each RPC, the network overhead incurred for each call makes this solution nonviable, especially as the frequency of the remote calls increases.

The individual transmission of RPCs will herein be classified as the weak bundling strategy. In order to have a stronger bundling strategy where we send multiple RPCs in a single transmission, we need to have a closer look at the possible requests that we may have. In our Canvas example, our supported functions do not require the remote server to respond back with a result. For our RPCs we only desire to have a remote effect, in this case to draw an item on the canvas. Because we don't need a result, it is straightforward to deal with these requests asynchronously and simply combine all our RPCs into a single request and continue on with the program. In [Spector, 1982], Spector makes mention that with asynchronous requests the server does not even need to respond, this notion helps us to lower the number of communications between the client and the server and opens the door to bundling packets together.

But what happens when we need to use synchronous requests? and do things change if our future RPCs depend on the results of previous synchronous requests? When can we bundle these remote primitive actions and when are we forced to transmit and wait for the server's response?

We could create a `sendS` function that takes a list of JS actions and returns a list of results:

```
sendS :: [JS a] -> IO [a]
```

But this is not a general solution. If later we realized that we would like to use a tree or some other container different than a list to hold our `JS` requests we would need to rewrite the function. We will discuss these questions and find stronger bundling strategies with a more general version of send in the next two chapters.

Chapter 4

Remote Primitives

In school growing up, after learning basic mathematical operations, my class was introduced to the calculator. When we were then asked to do many operations by hand, a repeated complaint from the class was, “Why can’t we just use a calculator? It can do this problem”. The teacher’s response was, “You won’t always have a calculator with you, so you need to know how to do these equations without one.”

In a day where cell phones are so ubiquitous, we indeed do always have a calculator with us. Our society has gone from few room-sized computers, to even more powerful systems that can fit in our pockets. Our music collections have gone from bookshelves full of records, cassettes or CD’s to USB drives with 100s of mp3s and nowadays we have subscriptions to music services where we have access to almost any song without actually owning the songs. Even the apps on our phones have gone from being completely contained on the phone to requiring internet access for any of it to function. In programming, we commonly use `ssh` to run commands on other machines or create SQL queries that are to be executed on a database that is hosted by a third party.

4.1 Primitive Classification

In many cases, our computers are comprised of multiple entities that can add functionality. Whether that is hosting a database or some other background application as a service or even using a GPU to do some matrix operations. So far, in looking at how we interact with these other entities, regardless of whether it resides locally or on a remote machine, we have been able to identify three types of primitives.

The three types of primitives are *Commands*, *Procedures*, and *Queries*. With these classifications, we can make intelligent decisions on what we can batch together or what needs to be sent as it is encountered, giving us an overall lower communication cost with the "remote" entities.

Commands are primitives that have no result. These primitives are used for their effect on the remote system instead of the result of some computation. These computations have a resulting type of `()` in Haskell or `void` in other languages. Some basic examples of a command would be the setting or changing the font, color or stroke type of a graphical library, the incrementing of a remote counter or setting some other variable that resides remotely, or any remote state change.

These primitives can be sent asynchronously since the local machine is not waiting for a result. *Commands* cannot throw exceptions and any errors are ignored. All of the primitives used in our Canvas example in Section 3.3.2 are *commands*.

Procedures are primitives that are used for both the effect on the remote system as well as for the result of the action. *Procedures* can fail and are not sent asynchronously. These primitives are easy to spot by what the program does after making the call. If there is a branch or some work that is based on the result of the primitive, then it is most likely a *procedure*. Some primitives can be classified as *procedures* while still returning `()` or `void` as a result. In these cases, we want to know when the server has completed the task. These *procedures* are less common but are used when we want to be sure that our effect has occurred before continuing the program. A delay or sleep call would be this kind of primitive. The

Attribute	Command	Procedure	Query
Changes State	✓	✓	✗
Has Result	✗	✓	✓
Can Fail	✗	✓	✓
Asynchronous	✓	✗	✗

Table 4.1: Attributes of the different types of primitives

result that we care about with this type of call is a temporal result instead of some return value.

Queries are primitives that may or may not have a result but do not have an effect on the remote system. These primitives are very close to *procedures* with the only difference being that these primitives perform strictly read-only actions. If we have a set of primitives that can be reordered and we still have the same behavior and result, then we are probably dealing with *queries*. The standard example of a *query* primitive is a read-only call to a table in a database. Just like procedures, these cannot be performed asynchronously and we can receive a failure from the server when making the call.

It should be noted that it is up to the user to classify the primitives. If a call mostly acts like a *command* but we would like to catch errors as they occur the user can simply indicate that call as a *procedure*. On the flip-side, if we wanted the effect of a *procedure* but didn't care about receiving the result or catching any errors, then we could classify it as a *command*.

Four questions arise from these characteristics that can help a user classify their primitive actions:

- Does the action change the remote state?
- Does the action produce a result?
- Can the action raise an exception?
- Should the client wait for the action to finish?

After answering these questions we can look at Table 4.1 and find the recommended types of primitives that will be present in our application.

4.2 Using Primitives in Haskell

Performing these primitives in a remote context requires us to use the `IO` monad or work in a monad that is built on top of the `IO` monad since all input/output functionality is housed in the `IO` monad. As was mentioned in Section 2.2.3, there are two notations that are used when working in monads to perform operations sequentially. `do-notation` is the more friendly notation at least from a readability standpoint. Below we have a program that will greet a user saying "Good Morning, <name>!" or "Good Afternoon, <name>!" depending on what time it is.

```
main :: IO ()
main = do
   putStr "Name: "
   name <- getLine
   now <- getCurrentTime
   let TimeOfDay hour _ _ = localTimeOfDay now
   let greeting = if hour < 12 then "Good Morning" else "Good Afternoon"
   putStrLn $ greeting ++ ", " ++ name ++ "!"
```

When dealing with input and output, especially with user interaction, we need to preserve the order of the commands. In `do-notation` we are able to have monadic actions on each line. On some of the actions `<-` is present because the action returns a value that we will be using later in the program, these indicate our procedures or queries as defined above. The text to the left of `<-` is the variable name to which we *bind* the result. If we want the effect of the call, but want to ignore the result we can either exclude the `<-` and variable name, or we can explicitly say ignore this value by the following syntax:

```
_ <- getLine
```

In this case, we are taking a procedure and treating it as a command by ignoring the result.

All monadic actions are combined with `>>=` or `>>`. Once we have a definition for `>>=` any custom data type can become a monad. What this means, is that if we modify the behavior or optimize the way binds in a monad are used, we will be affecting any code that uses a monad. Imagine having an optimization on a `for` or `while` loop in an imperative language. Any code that uses these loops would achieve a speedup because of the optimization, without a need to change the user's code. In Haskell, by modifying the underlying monad to be one that uses optimizations in the context of remote operations, we can end up with a program that takes advantage of bundling without requiring the user to write code for that purpose.

4.3 Potential Bundling of Remote Primitives

In examining the properties of our primitives and the connecting operators in a monad there are a number of observations that we can make with respect to bundling these primitives.

1. For each bind, there will be at least two packets involved.

2. If we are only using applicative combinators to combine the monadic actions (no binds are present), then the elements of an applicative functor can all be sent in the same packet. This is discussed in greater detail in the applicative bundling description in Section 5.1

3. When a command is used on the left-hand side of a bind, we can replace the bind with an applicative operator. e.g. `command >>= \() -> ...` ≡ `command *> ...`

4. When queries are used they can be reordered.

5. Applicative functors can be normalized to:

$$\begin{tikzcd} & <\$\$> \arrow[dl, no head] \arrow[dr, no head] & \\ {[\mathit{Packet}]} & & \text{Post Processing Function} \end{tikzcd}$$

Observation (1) shows that a bind is going to signal the end of a potential batch of packets as well as the start of another batch. This is because the right side of the bind relies on the result of the left side of the bind. Without knowing the result of the left we do not know what primitives we will be sending, thus we have to send two packets for each bind that we encounter. Observation (2) shows that the because applicative arguments are independent of each other, we can, therefore, send an entire applicative functor in a single packet. These two observations lead us to observation (3) which says that there are times when we can eliminate a `>>=` and turn it into an applicative operator. If we have a command, then we know that it will result in `()`. If that is the case then we do not need to send the primitive to the remote server in order to continue. We can simply pass in the result that we know we are going to get and then continue. In this case, it is equivalent to using the applicative `*>` operator in place of the bind.

Observation (4) is a little more subtle. Since queries are read-only and can be reordered, then we can look forward at the computations and send anything that is not relying on other primitives. As an example, we can look at the following situation:

```
(q1 >>= q2) <*> (q3 >>= q4)
```

$$\equiv$$

```
liftA2 (,) q1 q3 >>= \(r1,r2) -> q2 r1 <*> q4 r2
```

In this example, `q2` and `q4` are blocked until we know the result of `q1` and `q3` respectively, but `q1` and `q3` do not rely on any previous results and are ready to be sent. We call this concept the Haxl Lemma and will revisit it in greater detail in Section 6.6 of the case studies.

Observation (5) is a description of how applicative functors are able to be sent in a single packet. If we have some `f <*> g1 <*> g2`, then the combining function `f` will remain on the local machine while `g1` and `g2` will be executed remotely with the results being later applied to `f` as a post-processing function.

Chapter 5

Remote Monads

A remote monad is a monad that takes a program of a certain shape, along with a transport mechanism and runs pieces of the program remotely using a user-selected bundling strategy to amortize the network cost. The shape that is needed is simply a separation between the structure and execution of remote calls by using GADTs. The use of GADTs coupled with a way of signalling which primitives should be treated as asynchronous commands or synchronous procedures, can lead us to automatic bundling of the remote primitives. The general idea is that a user can change the program's underlying monad to use the remote monad and, with minimal effort, be able to benefit from automatic bundling.

Along with the bundling strategies, we also need to understand what can happen when there is a failure and how it should be handled, or in other words, we will look at what it means to have remote control. We will first look at the different strategies and then discuss some possible solutions for handling remote failure.

5.1 Bundling Strategies

The concept of bundling requests was not found in the first specification of RPCs but it was one of the first optimizations that started to appear as RPCs began to be used [Shakib & Benson, 2001; Bogle & Liskov, 1994; Gifford & Glasser, 1988]. Batching or bundling requests was one of the additions of the second version of the RPC protocol specification [Microsystems, 1988]. We will take a look at three basic bundling strategies for the remote-monad: the weak, strong and applicative bundling strategies. These bundling strategies are based on the properties and observations of *commands* and *procedures* which were described previously in Chapter 4.

We will ignore queries at this time and only look at systems where commands and procedures are present. In the context of the different bundling strategies, any queries that are found in a system will simply be treated as a procedure. A bundling strategy involving queries is discussed later in Section 6.6, where their reordering ability introduces additional bundling possibilities.

The reason for the different bundling strategies is largely based on the combinators that are available in monads and applicative functors, which we will now discuss. To help us visualize the different remote calls and bundling, we will look at simulated interactions with an Internet of Things toaster.

Smart Toaster For our example, we can request that our toaster `says` a string, measure the `temperature` or make `toast` for a given number of seconds. We could use the following GADT to mirror our toaster capabilities:

```
data ToasterApi a where
    Say         :: String -> ToasterApi ()
    Temperature ::           ToasterApi Int
    Toast       :: Int    -> ToasterApi ()
  deriving (Show)
```

To handle the transmission of the `ToasterApi` as well as receive any responses to the requests, we will use a data type called `R` with a function called `send` to do the heavy lifting. For simplicity, we will transmit the `ToasterApi` primitives as strings by using the derived `Show` instance. In lifting our `ToasterApi` into `R` we use `sync` or `async` to signal if `send` should expect a response back from the toaster before continuing. In practice, our `send` function would also have an extra parameter describing how to reach a remote device, but we will assume that that information is hard-coded inside the `send` function as it does not add to our current discussion.

```
say :: String -> R ()
say s = async (Say s)

temperature :: R Int
temperature = sync Temperature

toast :: Int    -> R ()
toast i = sync Toast

send :: R a -> IO a
```

In the case of `say`, we only want the effect of saying something but we do not care about when it finishes or about the result, so we can mark it as an asynchronous request (command) using `async` to lift it into `R`. For `temperature` we care about the result and for `toast` we care about when it finishes, so in those cases we use `sync` to mark them as synchronous requests (procedures).

We will start by making an asynchronous request by having the toaster say "Would you like some toast?":

```
send $ say "Would you like some toast?"
```

Which results in the interaction captured in Figure 5.1 as a sequence diagram. In our sequence diagrams, asynchronous requests are depicted with an arrow in green and synchronous requests are depicted in red as interactions between the local client on the left of the diagram and the remote server on the right.

Now to see what synchronous requests look like let's get the temperature of the toaster:

```
send $ temperature
```

With the sequence diagram shown in Figure 5.2

If we want to send multiple requests to our toaster in our current setup with `ToasterApi` and `R` we will have the following:

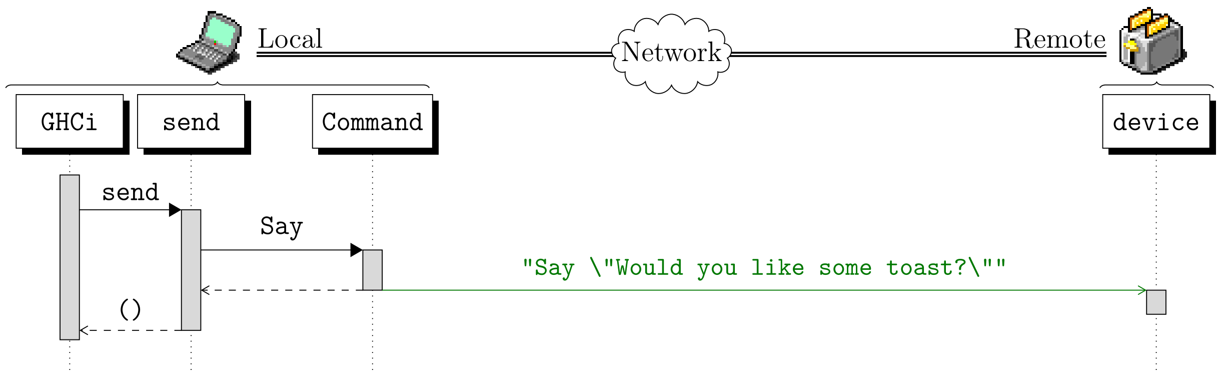


Figure 5.1: Example of an Asynchronous Remote Procedure Call

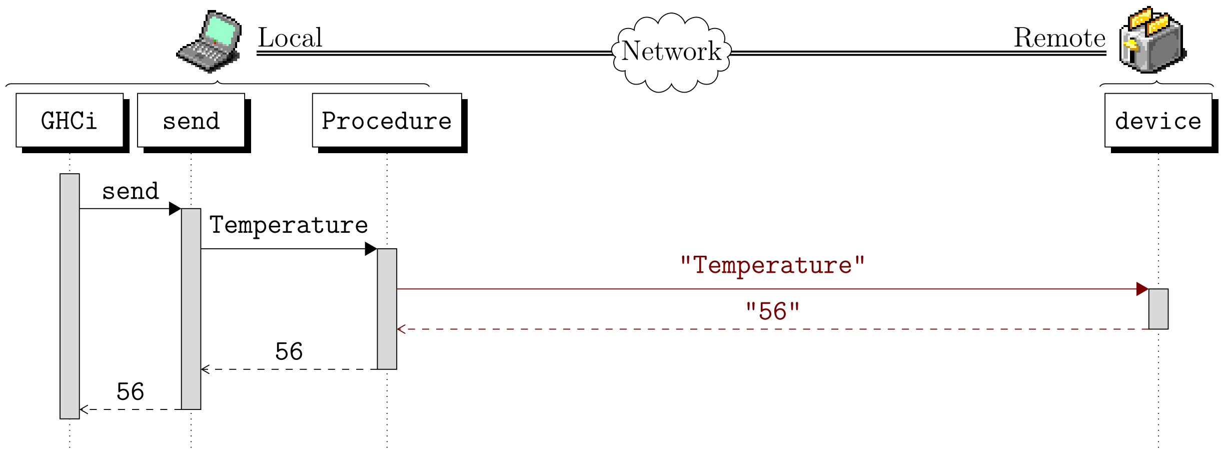


Figure 5.2: Example of a Synchronous Remote Procedure Call

```
send $ say "Good Morning!"
send $ say "Would you like some toast?"
```

Where two calls are being sent to our toaster. What we want to do is find a way to compose or sequence the requests with a single call to `send` to make the remote request. Let's first tackle the composition of our `ToasterApi` primitives, and then we can figure out a better bundling strategy.

Weak Bundling In Section 3.3.2, we defined weak bundling as the sending of our remote primitives one at a time, without any bundling. As we see in the above toaster example, we can get code into place to send requests to a remote toaster pretty quickly but composing the primitives together under a single `send`, as well as adding bundling will take some extra work. We can think of weak bundling as our ground zero or foundation for the remote monad and the stronger bundling strategies. In the weak bundling, we will tackle the first problem of composing our actions together by using a monad and get the general framework working in a modular way to allow us to later add bundling and application-specific layers.

As we create the components of the framework to later leverage a better bundling, many of the components will be trivial for the weak bundling strategy. The weak packet is an example of this since the packet is really just a primitive wrapped in a data type but when we look at other bundling strategies, the packet will be more complex and could contain multiple primitives.

We will not look at the implementation details at this point, but by simply turning `R` into a monad, by declaring an instance of `Monad`, we can then compose the `ToasterApi` actions together and perform multiple actions with a single `send`:

```
GHCi> t <- send $ do
              say "Would you like some toast?"
              t <- temperature
              say (show t ++ "F")
              return t
      when (t < 70) (send $ toast 120)
```

Figure 5.3 shows the sequence diagram for this example. We have not done any bundling at this point. As we can see in the sequence diagram, we are still sending each primitive individually to our toaster.

To further emphasize what weak bundling is and what it is not: *weak bundling has the general remote monad framework in place, where remote primitives go through the different layers by being transformed into packets, serialized and transmitted, and finally, are executed remotely, but does* ***not*** *have any network cost amortization.* The packets for the weak bundling are trivial, containing either a command or a procedure. Describing this as a regular expression we would have (C|P), where C represents a command, and P represents a procedure.

The triviality and lack of amortization of the weak bundling strategy may cause questions to arise about its usefulness. There are a couple of benefits of using this strategy:

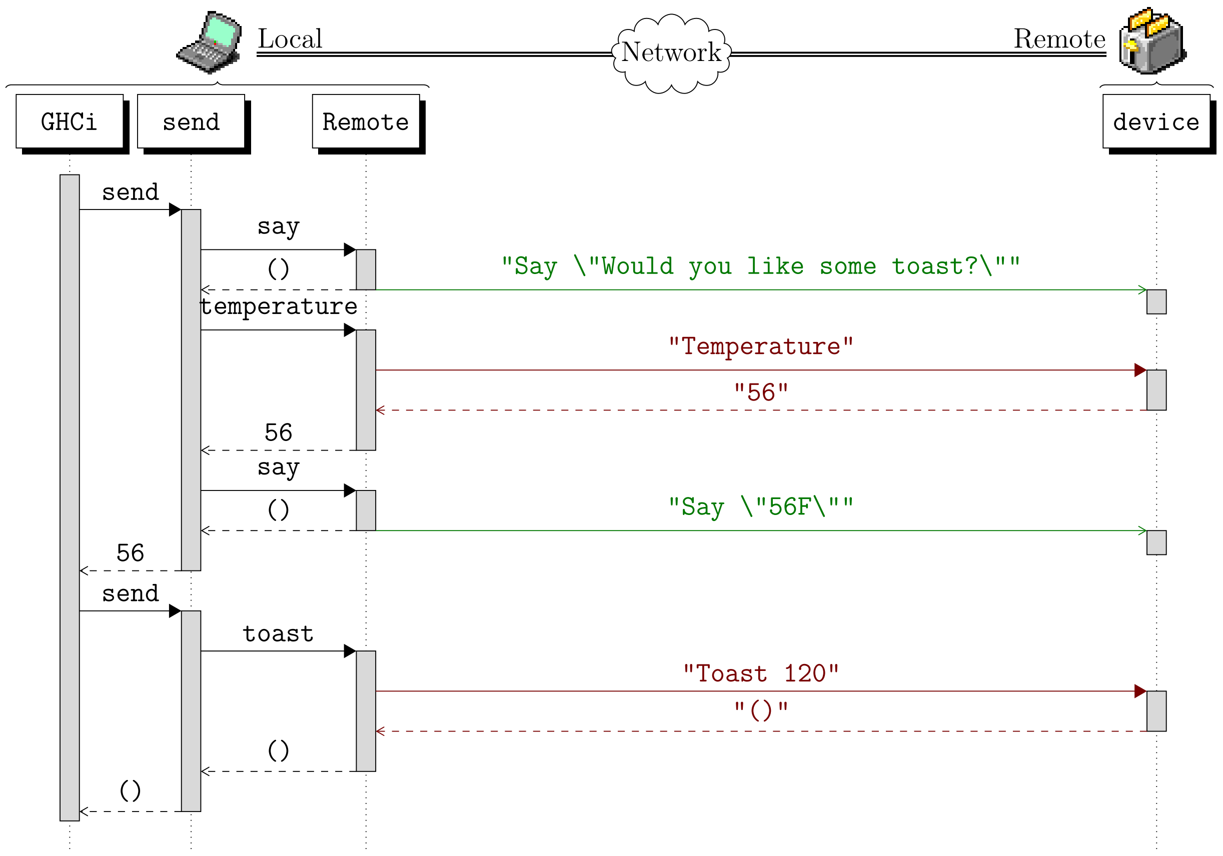


Figure 5.3: Example of a Weak Remote Monad

- *Baby Steps* – In refactoring a program to use the remote monad, this strategy gives us an easy landing point to test our refactoring without the need to debug or test the behavior of the user's program when batching is introduced.

- *Guarantee of sequentiality* – When a packet contains multiple items that are to be executed remotely, many specifications allow the server to execute the items in the batch in parallel. The JSON-RPC specification in particular says: "The Server MAY process a batch RPC call as a set of concurrent tasks, processing them in any order and with any width of parallelism." Many race conditions could occur in this environment. By downgrading our bundling strategy to the weak bundling, we could either identify the problem or at least narrow down our search for the bug by eliminating this class of errors.

Strong bundling By their very nature, commands can be sent asynchronously as we are not expecting a result from the server. What this means is that there will never be a command that is blocking another remote request. Normally, what we would do is fire off a request asynchronously and continue. What we can do instead is batch any asynchronous requests together and then transmit them when we come across a primitive that requires a response from the server. We call this the strong bundling.

To put it more succinctly, strong bundling will bundle any number of commands together and can be punctuated by at most 1 procedure. Once again, describing this using a regular expression we have (C*P?).

In an earlier section, we discussed how in a monad, there is the notion of binding the result of a computation to a variable using `<-` in `do`-notation or with `>>=` as shown below:

```
comp1 >>= \ x -> comp2 x
```

Where the left computation is executed and the result gets sent to the right hand side as the variable `x`. How does the strong bundling handle this situation? Ideally, we would like to send both `comp1` and `comp2` to the remote service in a single packet. The problem with this is that we cannot send the computation on the right side of `>>=` until we have the result of the left side, thus we are unable to bundle both sides of a bind into a single transmission. But there is an exception to this rule. If the left-hand side of a bind is a command, then we know, statically, that the result of the computation will be `()`. This allows us to apply the result of the left-hand side to the function found on the right-hand side of `>>=` and we can then send both sides in a single packet.

Using the same code that was used for the weak bundling above we can see in Figure 5.4 how the first asynchronous `Say` command is able to be bundled together and punctuated by the synchronous `Temperature` procedure. Something that needed four packets to be sent to the toaster is now able to do so in just three. It should be noted that if we had more consecutive `says`, then the number of packets would not change in the strong bundling as we can send any number of consecutive commands together.

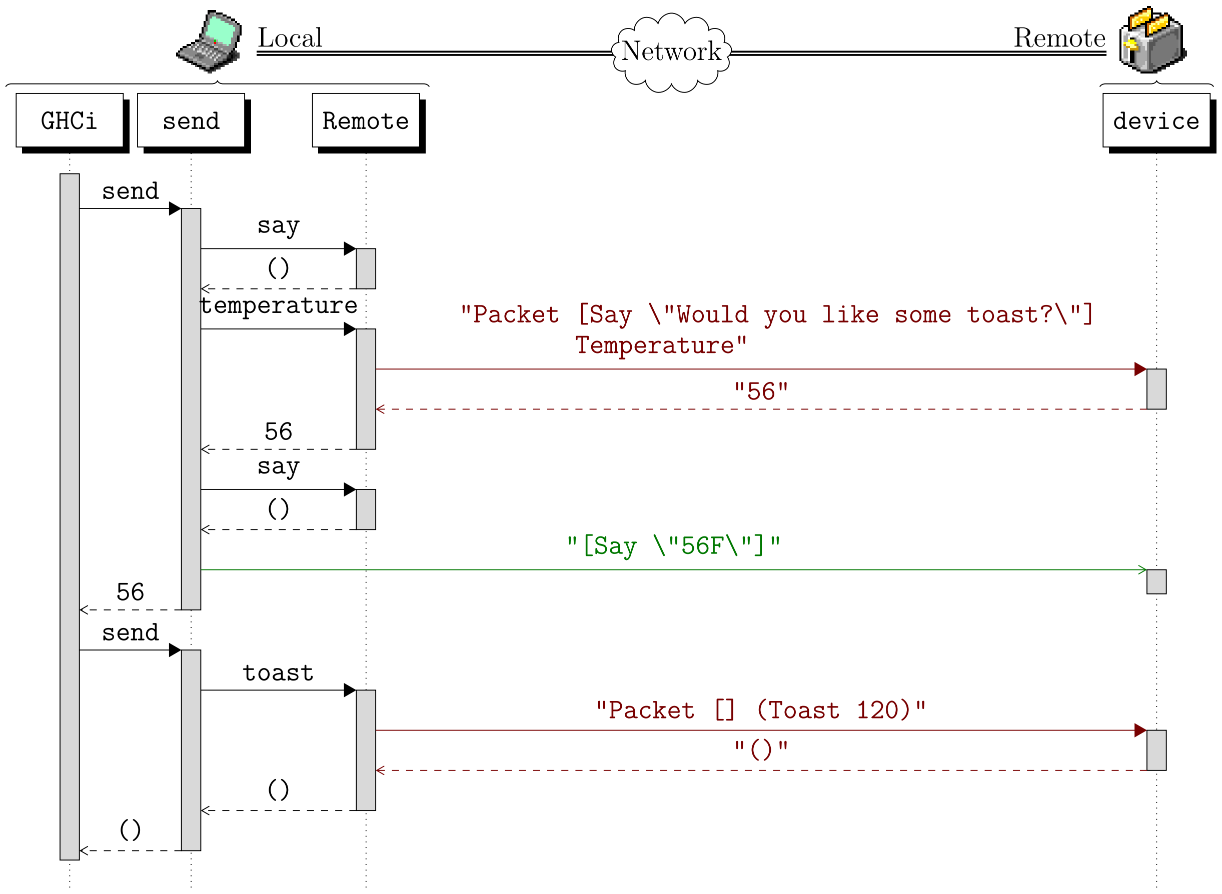


Figure 5.4: Example of a Strong Remote Monad

Applicative Bundling As a quick recap, in Haskell, not only can we compose actions together using monads but we can also use applicative functors. With applicative functors, the `<*>` operator is used to apply a function that is wrapped in an applicative context to arguments that are also in applicative contexts. Here is its type:

```
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

The main difference here from the monad is that the arguments to be applied to the function on the left are all independent of each other.

```
func <*> action1 <*> action2
```

In the above example, we can look at `action1` and `action2` as remote primitives. They will be run remotely and then we can call `func` locally with the results as arguments. Because of the way that applicative functors are structured, `action2` has no way of using the result of `action1` because none of `action1`'s components are within scope. This independence allows us to bundle both of the actions together and lends itself well to parallel operations on the remote server, if desired.

Applicative Bundling is the most efficient, by packaging any number of commands and procedures together by using the applicative construct, (C|P)* in regex. By definition, monads are instances of applicative and since we can get the best bundling out of applicative operations we try to treat the monad as an applicative as much as we can. One way that we do this is by translating the `>>` monadic operator into the `*>` applicative operator which yields the same behavior. Because of the power of the applicative functor in bundling, we have created a remote applicative structure which can be used to enforce applicative combinations instead of the monadic binds.

To show off the abilities of the applicative bundling, we modify our example to use applicative operators.

```
GHCi> (t1,t2) <- send $
              liftA2 (,)
                     (say "Good Morning!" *> temperature)
                     (say "Toasting..." *> toast 120 *> temperature)
Remote: Good Morning!
Remote: Toasting...
... sleeping for 120  seconds ...
GHCi> print (t1,t2)
(56,99)
```

Resulting in the sequence diagram in Figure 5.5.

To show the differences between the different bundling strategies while using an established protocol, we will look at examples taken from the JSON-RPC case study, with each of the bundling strategies being used with the same code:

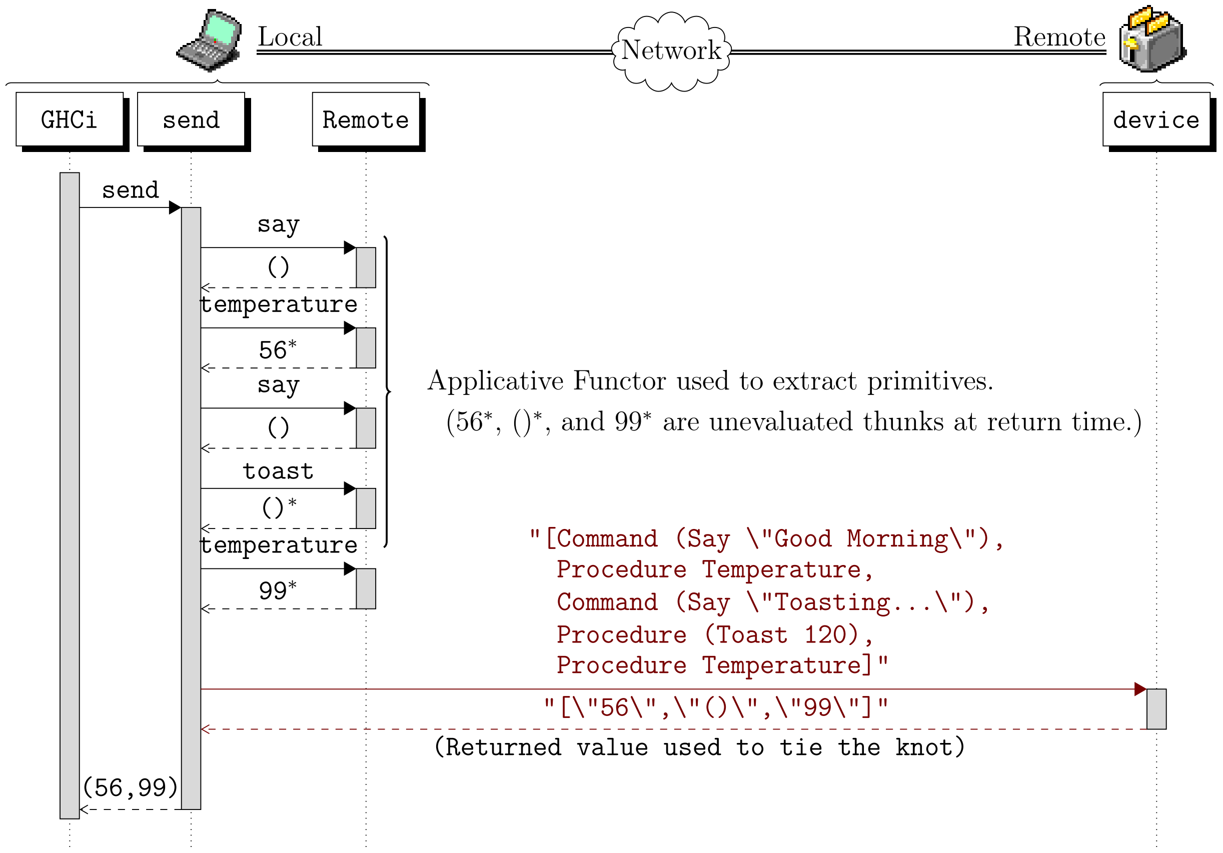

Local
Network
Remote
GHCi
send
Remote
device
send
say
()
temperature
56*
say
()
toast
()*
temperature
99*
Applicative Functor used to extract primitives.
(56*, ()*, and 99* are unevaluated thunks at return time.)
"[Command (Say \"Good Morning\"),
Procedure Temperature,
Command (Say \"Toasting...\"),
Procedure (Toast 120),
Procedure Temperature]"
"[\"56\",\"()\",\"99\"]"
(Returned value used to tie the knot)
(56,99)

Figure 5.5: Example of a Remote Applicative Functor

```
example s = do (t,u) <- send s $ do
                 say "Hello, "
                 t <- temperature
                 say "World!"
                 u <- uptime "orange"
                 return (t,u)
             print t
             print u
```

The API used for this example is very similar to our toaster example with the addition of `uptime` which is a synchronous procedure. The send function is different than we have seen before as well. In the case study for JSON-RPC the send takes an extra input argument that will describe which bundling strategy to use as well as information about the destination server.

As mentioned earlier, we can distinguish the commands and procedures by looking at what we do with the results. In the code above we can see that we do not store the result from a call to `say`, but we do bind the results of `uptime` and `temperature` to variables to be used later.

Weak Bundling Example

In this example and the subsequent examples, `-->` is used to show the data that is sent to the server, whereas `<--` is what the server sent back.

```
// (1)
--> {"jsonrpc": "2.0", "method": "say", "params": ["Hello, "]}
// No reply
// (2)
--> {"jsonrpc": "2.0", "method": "temperature", id: 1 }
<-- {"jsonrpc": "2.0", "result": 99, "id": 1}
// (3)
--> {"jsonrpc": "2.0", "method": "say", "params": ["World!"]}
// No reply
// (4)
--> {"jsonrpc": "2.0", "method": "uptime", "params": ["orange"], id: 1 }
<-- {"jsonrpc": "2.0", "result": 3.14, "id": 1}
```

Weak bundling yields 4 packets, one for each of the function calls. Notice that there is no reply from the server in response to the `say` calls but we do see replies to the other calls. This is the most accurate way to distinguish commands and procedures.

Strong Bundling Example

As a reminder, the strong bundling strategy is going to attempt to combine as many commands together as possible with at most one procedure.

```
//(1)
--> [ {"jsonrpc": "2.0", "method": "say", "params": ["Hello, "]}
    , {"jsonrpc": "2.0", "method": "temperature", id: 1 }
    ]
<-- [ {"jsonrpc": "2.0", "result": 99, "id": 1}
    ]
//(2)
--> [ {"jsonrpc": "2.0", "method": "say", "params": ["World!"]}
    , {"jsonrpc": "2.0", "method": "uptime", "params": ["orange"], id: 1 }
    ]
<-- [ {"jsonrpc": "2.0", "result": 3.14, "id": 1}
    ]
```

Using the strong bundling strategy allows our example to use only 2 packets. To demonstrate that we can have any number of commands before our procedure calls while still only transmitting 2 packets, we will add some extra `say` commands before our `temperature` and `uptime` procedures:

```
example = do (t,u) <- send s $ do
                 say "Hello, "
                 say "World! "
                 say "Hi, "
                 say "Earth! "
                 t <- temperature
                 say "Howdy, "
                 say "Mundo! "
                 say "Hey, "
                 say "Universe!"
                 u <- uptime "orange"
                 return (t,u)
             print t
             print u
```

which then yields:

```
//(1)
--> [ {"jsonrpc": "2.0", "method": "say", "params": ["Hello, "]}
    , {"jsonrpc": "2.0", "method": "say", "params": ["World! "]}
    , {"jsonrpc": "2.0", "method": "say", "params": ["Hi, "]}
    , {"jsonrpc": "2.0", "method": "say", "params": ["Earth! "]}
    , {"jsonrpc": "2.0", "method": "temperature", id: 1 }
    ]
<-- [ {"jsonrpc": "2.0", "result": 99, "id": 1}
    ]
//(2)
--> [ {"jsonrpc": "2.0", "method": "say", "params": ["Howdy, "]}
    , {"jsonrpc": "2.0", "method": "say", "params": ["Mundo! "]}
    , {"jsonrpc": "2.0", "method": "say", "params": ["Hey, "]}
    , {"jsonrpc": "2.0", "method": "say", "params": ["Universe!"]}
    , {"jsonrpc": "2.0", "method": "uptime", "params": ["orange"], id: 1 }
    ]
<-- [ {"jsonrpc": "2.0", "result": 3.14, "id": 1}
    ]
```

Applicative Bundling Example

To increase the possible concurrency and to further minimize the number of packets sent, we can use applicative bundling for our example. However, the monadic structure of do-notation needs to be rewritten to use applicative functors [McBride & Paterson, 2008]. This can be done automatically in GHC 8.0 with the applicative-do extension, or by manually using the applicative combinators. The net result is the same: we send only one packet now containing the two commands, and two procedures:

```
-->  [ {"jsonrpc":"2.0","params":["Hello,"], "method":"say"}
     , {"jsonrpc":"2.0","method":"temperature","id":1}
     , {"jsonrpc":"2.0","params":["World!"], "method":"say"}
     , {"jsonrpc":"2.0","params":["orange"], "method":"uptime","id":2}
     ]
<-- [ {"result":65,"jsonrpc":"2.0","id":1}
    , {"result":68.28,"jsonrpc":"2.0","id":2}
    ]
```

With this example, we were able to send everything in a single packet. This was possible because the results of our primitives were not used in subsequent requests, the variables were only used to build up the final result. This allowed us to eliminate the binds and use our combining function on the results using applicative functors. If we changed our code to be:

```
example s = do (t,u) <- send s $ do
                   say "Hello!"
                   t <- temperature
                   say ("The temperature is: " ++ show t)
                   u <- uptime "orange"
                   return (t,u)
               print t
               print u
```

Then even with the applicative packet we will have to send 2 packets because we do not have all the information necessary for the `say` that mentions the temperature:

```
//(1)
--> [ {"jsonrpc": "2.0", "method": "say", "params": ["Hello!"]}
    , {"jsonrpc": "2.0", "method": "temperature", id: 1 }
    ]
<-- [ {"jsonrpc": "2.0", "result": 99, "id": 1}
    ]
//(2)
--> [ {"jsonrpc": "2.0", "method": "say", "params": ["The temperature is: 99"]}
    , {"jsonrpc": "2.0", "method": "uptime", "params": ["orange"], id: 1 }
    ]
<-- [ {"jsonrpc": "2.0", "result": 3.14, "id": 1}
    ]
```

At this point we have only talked about the Remote Monad and how applicative functors have the potential to allow us to send more primitives in a single packet. If we wanted to enforce the use of applicative functors to get the best bundling, we could use the Remote Applicative functor. The Remote Applicative has the same techniques as the Remote Monad but everything is simpler because of the lack of bind. Since we very often need to use the results in subsequent calls, we would use multiple sends every time a bind would need to take place.

Each of these bundling strategies can be combined together with either the Remote Applicative or with the Remote Monad, leading to 6 configurations for remote execution.

$$\begin{tikzcd}
\left\llbracket \begin{matrix} Remote \\ Monad \end{matrix} \right\rrbracket \arrow[d, mapsto, "(1)"] & \left\llbracket \begin{matrix} Remote \\ Monad \end{matrix} \right\rrbracket \arrow[d, mapsto, "(2)"] & \left\llbracket \begin{matrix} Remote \\ Monad \end{matrix} \right\rrbracket \arrow[d, mapsto, "(3)"] \\
\left\llbracket \begin{matrix} Weak \\ Packet \end{matrix} \right\rrbracket & \left\llbracket \begin{matrix} Strong \\ Packet \end{matrix} \right\rrbracket & \left\llbracket \begin{matrix} Applicative \\ Packet \end{matrix} \right\rrbracket \\
\left\llbracket \begin{matrix} Remote \\ Applicative \end{matrix} \right\rrbracket \arrow[d, mapsto, "(4)"] & \left\llbracket \begin{matrix} Remote \\ Applicative \end{matrix} \right\rrbracket \arrow[d, mapsto, "(5)"] & \left\llbracket \begin{matrix} Remote \\ Applicative \end{matrix} \right\rrbracket \arrow[d, mapsto, "(6)"] \\
\left\llbracket \begin{matrix} Weak \\ Packet \end{matrix} \right\rrbracket & \left\llbracket \begin{matrix} Strong \\ Packet \end{matrix} \right\rrbracket & \left\llbracket \begin{matrix} Applicative \\ Packet \end{matrix} \right\rrbracket
\end{tikzcd}$$

Here are some observations about the many configurations:

- (1) and (4) have no network cost amortization and as such are only used if the user needs to have a guarantee of sequential execution. The only difference between these (1) and (4) would be the ability to use both monadic and applicative notation and operators (1) or using only applicative operations (4).

- (2) and (5) have some bundling but in our research we have found that there is no benefit to using the strong bundling strategy with the remote applicative when compared with the applicative bundling strategy. This is due to the fact that the strong bundling strategy was developed to handle the serialization of binds, which are not present in applicative computations.

- (3) and (6) have the best bundling possible for both remote monad and remote applicative. Though in some cases, the remote applicative will have better bundling than the remote monad if binds are used in the remote monad version of the computation. This would only be the case if the ApplicativeDo flag was not enabled.

The modularity of the framework allows the user to pick and choose the configuration, and easily change their mind later with minimal code changes.

5.2 Implementing the Remote Monad

Before covering the implementation details specific to the remote monad and remote applicative functors, we need to look at how natural transformations can be used to convert data types and how these can be combined to have a chain of natural transformations, as this is used extensively throughout the framework.

5.2.1 Natural Transformations

For this framework, we will be looking particularly at natural transformations over functors. A natural transformation, put concisely, is a morphism from one functor to another. In plain English, this means that we take a value that currently resides in some context and we represent that value in a different context. If we look at a person living on Earth, we can see our person as our value and "living on Earth" as the context. Now if we look at the same person but we are looking at them in a different context, say "living on the moon", we can see that some of the attributes remain the same: their height, mass, hair color, etc. but others are affected by the context: their weight, what they eat, how high they can jump, etc. The mechanism of changing this context is a natural transformation.

In Haskell, we can express these natural transformations as a type synonym where we are taking a functor `f` and transforming it into a different functor `g`:

```
type f ~> g = forall a . f a -> g a
```

We have our value of type `a` being in the context of `f` and then we result in a value of the same type, but this time it is in the context of `g`.

Another way of looking at a natural transformation is an interpretation of `f` as `g`. We can then chain these natural transformation togethers.

```
interpret :: (f ~> IO) -> (g ~> IO)
```

The above translates to "give me a way to interpret a value housed in `f` in the `IO` monad, then I'll give you a way to interpret a value housed in `g` in the `IO` monad". In the remote monad framework, we will see this as "show me how you want me to send a byte string across the network, and I'll give you a way to send remote monad packets across the network". What is happening under the hood is that we have mechanisms in place to translate a remote monad into byte strings but we do not know how to send the byte strings or even where they should be sent. Passing a way to interpret the byte strings, by means of transmission, fills in the missing pieces to complete the execution.

As a side note, we are not just using natural transformations because they are helpful mechanisms, they are necessary to hide the `forall a` from the type checker since we want the `a` to remain the same throughout the chain of transformations. We only want to change the information surrounding the `a`. As an example, an `Int` in one functor will still result in an `Int` in the new functor. The reason why this is important is because our initial data structure is mirroring a remote function. Say we want this function to return a `String`. We want to transform this structure into something that can be transmitted and executed, but regardless of all the hoops, hurdles and general munging of data that it might have to go through, we still want a `String` to come back as the result, instead of some other data type.

Once we have a single natural transformation in place, we can chain them together to mirror the process of going from a local representation of an RPC, to something that can be serialized and transmitted for the calling of the RPC and then a similar chain when we deserialize and interpret the result. The chaining of these natural transformations is called a Composable Network Stack.

5.2.2 Composable Network Stacks

A composable network stack is a combination of several natural transformations over functors, used to make up the different pieces of a network. The stack of natural transformations is modular and allows us to add or remove different layers of computation making the framework able to be applied to a variety of situations.

The way that composable network stacks are implemented is essentially a pipeline of GADTs with the natural transformations used as the connectors. This yields a full composable network stack which will have the following shape:

The remote server can build a `ByteString` interpreter when given a GADT evaluation function

```
(R ~> IO) -> (ByteString ~> IO)
```

i.e. "Give me a way to interpret `R` and I'll give you a way to interpret a `ByteString` (since I know how to parse a `ByteString` into an `R`)".

The transmission layer would handle `ByteString`s:

```
(ByteString~> IO) -> (ByteString ~> IO)
```

i.e. "Give me a remote interpreter for `ByteString` and I'll give you a local interpreter for `ByteString`".

And finally:

```
(ByteString ~> IO) -> (R ~> IO)
```

i.e. "Give me a local interpretation of `ByteString` and I'll give you a way of interpreting `R`.

Giving each of these steps a name, we can then fit them together as a pipeline:

```
decodeR            :: (R ~> IO) -> (ByteString ~> IO)
transmitByteString :: (ByteString ~> IO) -> (ByteString ~> ByteString)
encodeR            :: (ByteString ~> IO) -> (R ~> IO)

pipeline :: (R ~> IO)
pipeline = encodeR (transmitByteString (decodeR runR))
```

We end up with a function `pipeline` that can take any `R` as input and it will take care of the serialization and remote execution. If we decided that we didn't like the transmission mechanism or that we wanted a different interpretation of R, then we can change out those pieces of the pipeline with a new mechanism of the same type and not have to modify any of the other pieces of the pipeline.

Now that we have looked at some bundling strategies as well as some general thoughts about natural transformations, let's look at the implementation of the remote monad and remote applicative.

5.2.3 Practicing what it Preaches

In Section 3.1.1, we talked about taking a program and instead of evaluating an expression immediately, we build up the computation into a structure that gets evaluated later. Instead of $5 + 1 = 6$ we have $5 + 1 = \mathit{Add}\,(I5)\,(I1)$, which we can later evaluate to be 6, some other interpretation of the equation, or even apply some rewrite rules. User functions that are moving from a local evaluation to somewhere outside of the runtime environment are forced to have this separation to allow the controlling program to be running locally and to have pieces executing remotely.

We likewise split the remote applicative and remote monad into a structure and an evaluation function. We will start by looking at the less complex data type, the remote applicative.

5.2.3.1 RemoteApplicative

Jumping into the remote applicative's declaration we have the following data type:

```
data RemoteApplicative p a where
  Primitive :: p a -> RemoteApplicative p a
  Ap        :: RemoteApplicative p (a -> b)
            -> RemoteApplicative p a  -> RemoteApplicative p b
  Pure      :: a -> RemoteApplicative p a
```

The `RemoteApplicative` is parameterized over the user's GADT, allowing us to lift the user's data type into the `RemoteApplicative` and use the applicative operators to combine the user's remote calls. In our canvas example, the GADT housing the remote functions was called `JS`. The remote applicative for that example would then look like `RemoteApplicative JS ()`. The `RemoteApplicative` constructors simply mirror the applicative operators as is shown in the instance of `Applicative`:

```
instance Applicative (RemoteApplicative p) where
  pure  = Pure
  (<*>) = Ap
```

Any of the user's remote functions get lifted into the `RemoteApplicative` using the `Primitive` constructor with `Ap` being used to combine primitives by applying them to a function which will usually be a local function that has been lifted by using `Pure`.

Assuming we have a user GADT describing the remote functions with smart constructors that lift the primitives into the `RemoteApplicative`, we now have to factor `RemoteApplicative` actions into packets and send the data remotely. Depending on the bundling strategy, we want the packetization to happen differently. To handle this, we create a Haskell class `RunApplicative` that is parameterized over our GADT and takes an evaluation function as an argument:

```
class RunApplicative f where
  runApplicative :: forall m prim . (f prim ~> m) -> (RemoteApplicative prim ~> m)
```

The first argument of the `runApplicative` function is a natural transformation from some container of primitives (packets) to some monad (usually `IO` or a monad which has `IO` abilities). Our `runApplicative` function will look at the `RemoteApplicative` that we have built up and turn it into packets. When the packets are ready, the natural transformation argument will then handle the serialization of packets and the transmission of the information to the remote system.

Let's look at running a trivial example of a program that will echo a string as given or do so in all caps. We will start by having a description of the functions as well as a way to execute them.

```
data Api a where
  Echo :: String -> Api ()
  Yell :: String -> Api ()

runApi :: Api a -> IO a
runApi (Echo s) = putStr s
runApi (Yell s) = putStr $ map toUpper s
```

Now we just need to lift `Api` into `RemoteApplicative` actions.

```
echo :: String -> RemoteApplicative Api ()
echo s = Primitive (Echo s)

yell :: String -> RemoteApplicative Api ()
yell s = Primitive (Yell s)
```

At this point we can build up a `RemoteApplicative` as follows:

```
  > echo "can you hear me now?" *> yell "how about now!"
```

The only thing we have left is to break the remote applicative into packets and then transmit data. The way that the bundling strategy is implemented is by having a different packet type for each bundling strategy. Once we decide which bundling strategy we would like to use, we create the natural transformation that knows how to handle the selected packet. The type that this natural transformation should take is:

```
(PacketType UserGADT ~> m)
```

Since our `runApplicative` function is a class parameterized over the user GADT and the packet type, we can trigger which bundling strategies to use to split up the actions based on the packet type used in our packet handler.

Because of the structure of the composable network stacks, this transmission function could actually just be the user's GADT evaluation function applied to the user GADTs found inside the packets instead of transmitting the request and handling the response.

Here are what the different packet types look like for each of the bundling strategies:

```
data WeakPacket (prim :: * -> *) (a :: *) where
  Primitive :: prim a -> WeakPacket prim a

data StrongPacket (prim :: * -> *) (a :: *) where
   Command   :: prim () -> StrongPacket prim a -> StrongPacket prim a
   Procedure :: prim a                       -> StrongPacket prim a
   Done      ::                                 StrongPacket prim ()

data ApplicativePacket (prim :: * -> *) (a :: *) where
   Primitive :: prim  a                 -> ApplicativePacket prim a
   Zip       :: (x -> y -> z)
             -> ApplicativePacket prim x
             -> ApplicativePacket prim y -> ApplicativePacket prim z
   Pure      :: a                       -> ApplicativePacket prim a

instance KnownResult Api of
  knownResult (Echo _s) = Just ()
  knownResult (Yell _s) = Just ()
```

For the `runApplicative` to know how to split the `Api` actions into packets, we need to specify if we know the end result of the action statically by using the `KnownResult` class. If the result is known (a command) then the it will be `Just a`, where `a` is the static result, if we do not know the result, then we put `Nothing` to signal that we will need to send the primitive to discover the result. This logic is only used in the strong and applicative bundling strategy but to fit in with the framework, the instance is needed even if the weak bundling is selected.

To run our 'echo' example locally using the weak bundling strategy, we would connect the functions as follows:

```
import           Control.Natural (wrapNT, unwrapNT)
import           Control.Remote.Applicative
import qualified Control.Remote.Packet.Weak as WP

evalPacket :: WP.WeakPacket Api a -> IO a
evalPacket (WP.Primitive p) = runApi p

send :: RemoteApplicative Api a -> IO a
send = unwrapNT $ runApplicative (wrapNT evalPacket)

main :: IO ()
main = send $ echo "You scream, I scream, we all scream for " *> yell "ice cream!"
-- GHCI --
> main
You scream, I scream, we all scream for ICE CREAM!
```

A great demonstration of the modularity of the framework is that if we wanted to change our bundling strategy for this example, then we would simply change our packet handler, `evalPacket`, to handle a `StrongPacket` or an `ApplicativePacket`. If we wanted to send it to a remote system we would change `evalPacket` to serialize and transmit the data and receive a response if needed. Sections 6.2 and 6.3 describe libraries which use JSON and `ByteString`s respectively to serialize packets with the user defining the corresponding instances to serialize and interpret the user's actions.

Something of note in the `ApplicativePacket` is its similarity to an applicative functor. In reality, the `ApplicativePacket` is an instance of applicative functor where the `<*>` operator maps to `Zip`. The `Zip` construct was chosen over `Ap` as it fit nicely with the pragmatic details of the library. The combining function, the first argument of `Zip`, is omitted when serializing the packet because the function will be applied locally to the returning results of the remote primitives. Remotely, when we receive an applicative packet we deserialize the `Zip` but we no longer have information about the function. We can then place a post-processing or serialization function in this slot on the remote side that executes before transmitting the result back to the client machine.

To further demonstrate this point, imagine we have established a protocol where the remote server puts the results of remote computations in a tuple before serializing them to a byte string. On the local side we have

```
     add <$> remoteAction x <*> remoteAction y
```

For simplicity, we will assume that `remoteAction` yields a single primitive instead of a more complex structure. The above code will result in the following `ApplicativePacket`:

```
    Zip add (Primitive (remoteAction x)) (Primitive (remoteAction y))
```

Then the two primitives will be serialized and sent with some indicator that this is a `Zip`. Our deserialization for `Zip` can then look like:

```
    res <- get
  case res of
     1 -> do  -- indicator of zip packet
          x <- get -- grab first argument
          y <- get -- grab second argument
         return $ Zip (\ a b -> (a,b)) x y
```

When the `Zip` is interpreted on the remote server, we then just need to apply the function that is the first argument and then the serialization function and we will comply with the protocol.

A version using `Ap` was also investigated, but was more cumbersome when it came to building the resulting packet in comparison with `Zip`, so we sidestepped these issues by choosing the `Zip` construct.

The biggest thing to know about these packets is that with the structure of the applicative functor in having a local function being applied to any number of remote primitives, it makes the applicative functor an ideal packet for transmitting remote primitives. This point was also independently reached in parallel by Jeremy Gibbons in [Gibbons, 2016].

5.2.3.2 RemoteMonad

Now we can move outward to the Remote Monad. In Section 5.1 and the previous section, we discussed how the applicative bundling is the most efficient bundling strategy. Dividing the RemoteApplicative into ApplicativePackets is a straightforward mapping. For the RemoteMonad, splitting a monad into ApplicativePackets means that we want to treat things as applicative operations as much as we can to make the direct mapping and only use monadic operations when we are forced to do so. Thus we end up with the following data type for the RemoteMonad with the Applicative and Monad instances:

```
data RemoteMonad p a where
   Appl :: RemoteApplicative p a ->
           RemoteMonad p a
   Bind :: RemoteMonad p a
        -> (a -> RemoteMonad p b)
        -> RemoteMonad p b
   Ap'  :: RemoteMonad p (a -> b) ->
           RemoteMonad p a -> RemoteMonad p b

instance Applicative (RemoteMonad p) where
  pure a               = Appl (pure a)
  Appl f   <*> Appl g  = Appl (f <*> g)
  f        <*> g       = Ap' f g

instance Monad (RemoteMonad p) where
  return   = pure
  m >>= k  = Bind m k
  m1 >> m2 = m1 *> m2
```

The relationship between RemoteMonad and RemoteApplicative is a subtle one. The RemoteMonad is either a computation that is a RemoteApplicative, wrapped in the Appl constructor or it is a bind between applicative actions and is wrapped in the Ap' constructor. The definition of >> also shows how we try to make things applicative as often as we can by mapping it to the *> applicative operator that has the same type. As mentioned in the theory of the remote monad, the way that we can get the best efficiency of transmitting monads is by maximizing the amount of code that can be represented as applicative functors and by minimizing the connecting binds.

The way that we use the RemoteMonad to select the bundling strategy as well as running the monad for the result is the same as the RemoteApplicative. We are able to use the same packets and we have a similar run function called `runMonad` to execute our monad:

```
class RunMonad f where
  runMonad :: (f p ~> m) -> (RemoteMonad p ~> m)
```

It is inside the `runMonad` function that the different tricks about eliminating binds, where possible, is implemented. To get the echo/yell example that was using the remote applicative working with the remote monad, we literally only need to change `RemoteApplicative` to `RemoteMonad` and everything will work and have the same behavior:

```
import           Control.Natural
import           Control.Remote.Monad
import qualified Control.Remote.Packet.Weak as WP

echo :: String -> RemoteMonad Api ()
echo = primitive . Echo

yell :: String -> RemoteMonad Api ()
yell = primitive . Yell

evalPacket :: WP.WeakPacket Api a -> IO a
evalPacket (WP.Primitive p) = runApi p

send :: RemoteMonad Api a -> IO a
send = unwrapNT $ runMonad (wrapNT evalPacket)

main:: IO ()
main = send $
         echo "You scream, I scream, we all scream for "
         *> yell "ice cream"
-- GHCI --
> main
You scream, I scream, we all scream for ICE CREAM!
```

If we wanted to change our bundling strategy, we would just have to change `evalPacket` to handle a different packet type which is the same as we would do with the remote applicative.

5.2.4 Remote Monad Laws

Just as we looked at the laws governing the applicative functor and monad as well as the other control structures in Haskell, let's look at the laws governing the remote monad. As these laws are not specific for the remote monad or remote applicative but describe the behavior of sending things remotely and bundling, we will use a `send` function to describe a `runMonad` or `runApplicative` function that has been passed a packet handler.

We propose using the monad-transformer `lift` laws [Liang et al., 1995; Gill & Paterson, 2017], also known as the monad homomorphism laws, as our remote monad laws because we are lifting the *Remote* computation to the remote site, by *Local* effect.

$$\texttt{send (return } a\texttt{)} = \texttt{return } a \tag{5.1}$$

$$\texttt{send (}m \texttt{ >>= } k\texttt{)} = \texttt{send } m \texttt{ >>= (send . } k\texttt{)} \tag{5.2}$$

Assuming these laws, the monad laws, and the laws relating functors and applicative functors to monads, the following morphism laws can be derived:

$$\texttt{send (pure } a\texttt{)} = \texttt{pure } a \tag{5.3}$$

$$\texttt{send (}m_1 \texttt{ <*> } m_2\texttt{)} = \texttt{send } m_1 \texttt{ <*> send } m_2 \tag{5.4}$$

$$\texttt{send (fmap } f\ m\texttt{)} = \texttt{fmap } f \texttt{ (send } m\texttt{)} \tag{5.5}$$

Laws (5.1) and (5.3) state that a `send` has no effect for pure computations. Laws (5.2) and (5.4) state that packets of remote commands and procedures preserve the ordering of their effects, and can be split and joined into different sized packets without side effects. Law (5.5) is a reflection of the fact that `send` captures a natural transformation.

These laws have equivalencies that are looking at the result of these equations. But as we are talking about bundling requests, it is worth noting that just because these statements result in the same value, does not mean that they achieved the result using the same number of packets. The first few laws describe what is actually happening in the library, things that do not need to be evaluated on the server are not sent. On the other hand, law (5.2) is splitting requests into multiple sends. We are able to split monads into packets and we can lower the number of packets by eliminating binds, but this all occurs within a single `send`. When there are multiple `send`s, that are not just using `pure`, we have to have at least one packet per send. This is because we are jumping in and out of the `RemoteMonad`. To illustrate this point imagine we have the following main method taken from Law (5.2) using do-notation:

```
main:: IO ()
main = do
  a  <- send m  :: IO Int
  () <- send (k a) :: IO ()
  return ()
where
   m :: RemoteMonad UserGADT Int
   k :: Int -> RemoteMonad UserGADT ()
```

In this example, we have work that needs to be done remotely, `m`, and a function, `k`, that takes the result of `m` as input and results in more work that might need to be done remotely. As we can see from the type information with the code, the *bind* from `send m` to `a` is actually an `IO` *bind* instead of a remote monad *bind*. All of the theory about automatically bundling efficiently resides inside our remote monad and remote applicative structures. As soon as we leave the remote monad or remote applicative context, we can no longer eliminate binds or apply any of the other techniques to make things more efficient. This means that even if `m` was a command that normally could be combined with a remote primitive, found in the result of `k a`, we would send `m` by itself.

5.3 Extending the Remote Monad

The goal of the remote monad framework is to find ways to automatically batch remote requests. We have discussed how we can batch requests together depending on if their result is known, or if we are combining things together in a way that can be optimized. We can further amortize the network cost by placing choice logic on the remote system. In particular, we can cause if-else statements, that rely on remote values, to reside remotely as well as have remote loops and Haskell's alternative construct (`<|>`).

There are a couple of ways that we can move these decisions remotely. Instead of making extra transmissions to get a remote value and then depending on that value, send the next remote value that needs to be executed. The first extension that we will look at is extending the remote monad and remote applicative to include data types that encapsulate the alternative construct `<|>`. Next, we will look at how a user could include remote if-else statements or remote loops through the user functions that are marked as remote via the GADT. Each of the extension methods that we are going to describe will require user access to modify code running on the server, due to the fact that we are sending information that may or may not be executed. This deviates from the normal batching behavior of sending a list of actions to the server that all need to be executed unconditionally.

5.3.1 Remote Alternative

Adding support for the remote alternative is more complicated than what it would seem at a first glance. The alternative construct `<|>` is used to attempt one computation and then, if failure is encountered, run the second computation. Instead of transmitting the first computation remotely and receiving a failure and requiring the second computation to be transmitted, we can send both computations to the server in a single packet. Supporting that logic to run remotely requires us to not only think about what should be considered a failure and how we should handle it, but also to make decisions about local failure versus remote failure and the consequences of those decisions. Here we will discuss not only handling the alternative operator in the remote applicative and remote monad contexts but we will also be discussing how things are affected when we are looking at the different bundling strategies in conjunction with this new operator. How does the behavior change when most of the logic is happening locally versus having the logic as well as the actions happening remotely?

In order to get alternatives working for the `remote-monad` library we need to expand the `RemoteMonad` and `RemoteApplicative` constructors to include constructors for `empty` and for `alt` which will take the two remote primitives as arguments. Making the `RemoteMonad` and the `RemoteApplicative` instances of the `Alternative` class is just a map into these constructors.

```
data RemoteApplicative prim a where
   Primitive :: prim a -> RemoteApplicative prim a
   Ap        :: RemoteApplicative prim (a -> b)
             -> RemoteApplicative prim a
             -> RemoteApplicative prim b
   Pure      :: a   -> RemoteApplicative prim a
   Alt       :: RemoteApplicative prim a
             -> RemoteApplicative prim a
             -> RemoteApplicative prim a
   Empty     :: RemoteApplicative prim a

instance Alternative (RemoteApplicative p) where
   empty       = Empty
   Empty <|> p = p
   m1 <|> m2   = Alt m1 m2
```

Just like we had an `Ap` in the `RemoteApplicative` and a `Ap'` in the `RemoteMonad` to help us know whether or not a bind was present in the computation, we also have the equivalent for `Alt` and `Empty`. This distinction between computations with bind present or not is used to help us on deciding which bundling rules we are to follow inside the `runMonad` function.

```
data RemoteMonad  p a where
   Appl        :: RemoteApplicative p a -> RemoteMonad p a
   Bind        :: RemoteMonad p a -> (a -> RemoteMonad p b) -> RemoteMonad p b
   Ap'         :: RemoteMonad p (a -> b) -> RemoteMonad p a -> RemoteMonad p b
   Alt'        :: RemoteMonad p a -> RemoteMonad p a -> RemoteMonad p a
   Empty'      :: RemoteMonad p a

instance Alternative (RemoteMonad p) where
    empty               = Empty'
    Empty' <|> p        = p
    Appl g <|> Appl h = Appl (g <|> h)
    m1 <|> m2           = Alt' m1 m2
```

To get the alternative working with the existing packet strategies, the `Alt` construct is handled locally. Here is an example we will use to examine the effects of this construct.

```
(command1 >> command2 >> empty >> commandX) <|> command3
```

If the above code is run with the `weak` bundling strategy then `command1` and `command2` are each sent to the remote server before the `empty` (failure) is encountered, and because there was a failure in the first set of actions, `commandX` is ignored and `command3` is then sent.

We cause the bundling strategies to adhere to the following restriction:

Bundling Strategies Bundling strategies are used to modify the way that primitives are transmitted, **but** the number and type of these primitives must remain unchanged across the different strategies.

In this example, `command1` and `command2` are a part of the computation on the left side of the alternative, which as a whole is a failing computation. This computation will either be ignored, or will, in the worst case scenario, need to be backtracked to get the server back to a valid state. In this example, it would be possible for the strong or applicative strategies to avoid sending parts of the failing computation by removing them from the send queue when the failure is encountered. Here is how that would look in our example above:

```
( command1 >> command2 >> empty >> commandX) <|> command3
  queued commands: []
  action: queue command1

(command1 >>  command2 >> empty >> commandX) <|> command3
  queued commands: [command1]
  action: queue command2

(command1 >> command2 >>  empty >> commandX) <|> command3
  queued commands: [command1, command2]
  action : dequeue previous commands and ignore next commands until <|>
           is reached

(command1 >> command2 >> empty >> commandX) <|>  command3
  queued commands: []
  action: queue command3

  queued commands: [command3]
  action: send queued commands
```

But by making this optimization, we would then lose the guarantee that changing bundling strategies will not have an effect on the logic of the program. The commands that would be sent before the failure could change the state of the program, potentially impacting the final command `command3`.

Thus far, the use of the `Alternative` construct has been in a local sense, using `empty` to signal a failure. In real-world applications the server could have errors that we would want to capture and handle without subsequent communication. We would want the remote system to recognize the failure, to recover, and then attempt to call the right-hand side of the `<|>` operation. There are three approaches that can be used to handle server failure.

- Change the type of procedures to encapsulate failure.
- Create a new bundling strategy, using a new packet type, to tell the server what to do in case of an error.
- All exceptions on the server can be caught, serialized and sent to the client who will

then recreate and throw the exception on the client machine.

5.3.1.1 Updating Procedure Type

Notice that we are only updating the type of our procedures and not updating the type of the commands. The client is not listening for any return values from a command because the response is always `()`. To handle a failing command, we would require a separate thread to listen for errors from the server, or have the server log errors in a way that can later be queried by the client program. Both of these options greatly increase the complexity. Instead, we follow the JSON-RPC 2.0 specification which states:

Notifications are not confirmable by definition since they do not have a Response object to be returned. As such, the Client would not be aware of any errors.

where `Notifications` in the JSON-RPC protocol are equivalent to our commands.

With procedures, the user could change their top-level GADT to be `UserGADT (f a)` instead of just `UserGADT a`, where `f` is a data type that encapsulates error. then change the return type of the server code. `Maybe` or `Either` are possible data types that would be probable options for `f`. Another option would be to build on top of the `remote-monad` library and lift **all** of the user's GADTs to be `Prim (Maybe a)` and then place a layer before the server to catch any errors and wrap them into the `f` data type. The benefit of the second option is that the user GADTs and the server evaluation functions remain unchanged.

5.3.1.2 Alternative Packets

The creation of Alternative Packets will allow us to run the `<|>` operator remotely. The alternative packet augments the applicative packet strategy with the notion of handling a failure. The packet has the same constructors as the `ApplicativePacket` with the addition of `Alt` and `Empty` constructors.

The `runMonad` and `runApplicative` functions will map directly from the `RemoteMonad` or `RemoteApplicative` alternative constructors to the `AlternativePacket` constructors. The main computation is sent to the server along with the ‘in case of failure’ computation in a single packet allowing the server to attempt an action and upon failure, attempt the second action without any additional network interactions. If both of the computations sent in the `AlternativePacket` are capable of failing then the user would need to use one of the other methods in conjunction with alternative packets. Traditionally, the second computation is a safer version of the first computation that cannot or is less likely to fail, but it is not always the case.

5.3.1.3 Serializing Exceptions

Another option for handling a remote failure is to add a layer around the remote evaluation function to catch any exceptions that may occur. The server can then serialize the exception and send it back to the client. In this case, the client would need to look at each response as either the result of the requested computation or a serialized exception. The `remote-binary` library provides an example of serializing any exceptions that occur on the remote server, and further discussion is found in Section 6.3.1.

5.3.2 Remote Logic

We just looked at what it would take to extend the constructors of the `RemoteMonad` and `RemoteApplicative` and add a new packet structure to send things efficiently, now let’s look at another way of taking our logic remotely by adding to the remote functions described in the user’s GADT. As an example to help us demonstrate this method, imagine that we would like to have a smart garden that has sensors to check the health of the garden with some visual representation, a light or an LED screen, that would give us alerts or status updates at a glance. We would have some server, connected to a moisture sensor for our garden, that will be periodically checking to see if the plants are too dry (reading of 0 - 2), well-watered (reading of 3 - 6) or over-watered (reading of 7 - 9). The plants we are dealing with are very sensitive and we would like an accurate view of their health. The visual indicator should change each time the moisture result changes and we would like to send an alert to the owner's phone when it is imperative to water the plants (1 or lower), or if the plants are beginning to flood (8 or higher). This example also serves as an extra example for how to create a service using the remote-monad library.

Let's create our GADT to describe our remote functions:

```
data Light = Red | Orange | Yellow -- 0 - 2
           | YellowGreen | LightGreen | ForestGreen | BlueGreen -- 3 - 6
           | Blue | Purple | FlashWhite  -- 7 - 9
  deriving (Show, Enum)

data SmartGarden a where
  SenseMoisture :: SmartGarden Int
  UpdateLight   :: Light -> SmartGarden ()
  SendAlert     :: String -> SmartGarden ()

instance KnownResult SmartGarden where
  knownResult SenseMoisture   = Nothing
  knownResult (UpdateLight _) = Just ()
  knownResult (SendAlert _)   = Just ()

execSG :: SmartGarden a -> IO a
  -- interaction with sensor
execSG SenseMoisture   = getStdRandom $ randomR (0,9)
  -- interaction with light indicator
execSG (UpdateLight l) = putStrLn $ "Light Indicator: " ++ show l
  -- alert sent to a user device
execSG (SendAlert s)   = putStrLn $ "Alert! " ++ s
```

We have our set of user functions that can measure the moisture of the plants, update our visual indicator and we also have the ability to send alerts to a user. For simplicity we substitute a random number generator and print statements in place of interactions with real hardware components.

Now that we have our API set up, we will setup some smart constructors to lift our remote primitives into the `RemoteMonad`, define our run function and then have our program logic:

```
--smart constructors
sense :: RemoteMonad SmartGarden Int
sense = primitive SenseMoisture

update :: Light -> RemoteMonad SmartGarden ()
update l = primitive (UpdateLight l)

alert :: String -> RemoteMonad SmartGarden ()
alert msg = primitive (SendAlert msg)

-- Bundling Strategy selection
runWP :: WP.WeakPacket SmartGarden a -> IO a
runWP (WP.Primitive sg) = execSG sg

send :: RemoteMonad SmartGarden a -> IO a
send = unwrapNT $ runMonad (wrapNT runWP)

-- program logic
main :: IO ()
main = send $ loop (-1)
  where
  loop oldLevel = do
    level <- sense
    update (toEnum level)
    if oldLevel >= 0 && level /= oldLevel then
      do
      case toEnum level of
        Red         -> alert "Your garden is completely dry!"
        Orange      -> if oldLevel > level then
                           alert "Your garden is almost dry!"
                         else
                           return ()
        FlashWhite  -> alert ("Your plants are getting over-watered,"
                                ++ " turn off the water!")
        _           -> return ()
      io $ threadDelay 1000000
      loop level
    else
      do
        io $ threadDelay 1000000
        loop level
```

Once again we are handling everything locally for our discussion but we could change the runWP to transmit the packet instead of calling the evaluation function directly. In this example, our entire logic is found within the loop so moving our loop to the remote server logic will make all the local logic become remote. We can imagine a select number of computationally expensive loops, either expensive because of the remote operations or by the quantity of the remote operations, being pulled to the remote location while other logic is left to be handled locally.

Here is what it would look like to move our loop to the remote location.

```
data SmartGarden a where
  SenseMoisture :: SmartGarden Int
  UpdateLight   :: Light -> SmartGarden ()
  SendAlert     :: String -> SmartGarden ()
  LoopIt        :: Int -> SmartGarden ()

instance KnownResult SmartGarden where
  knownResult SenseMoisture   = Nothing
  knownResult (UpdateLight _) = Just ()
  knownResult (SendAlert _)   = Just ()
  knownResult (LoopIt _)      = Just ()

sense :: RemoteMonad SmartGarden Int
sense = primitive SenseMoisture

update :: Light -> RemoteMonad SmartGarden ()
update l = primitive (UpdateLight l)

alert :: String -> RemoteMonad SmartGarden ()
alert msg = primitive (SendAlert msg)

loopIt :: Int -> RemoteMonad SmartGarden ()
loopIt i = primitive (LoopIt i)
```

```
execSG :: SmartGarden a -> IO a
  -- interaction with sensor
execSG SenseMoisture   = getStdRandom $ randomR (0,9)
  -- interaction with light indicator
execSG (UpdateLight l) = putStrLn $ "Light Indicator: " ++ show l
  -- alert sent to a user device
execSG (SendAlert s)   = putStrLn $ "Alert! " ++ s
execSG (LoopIt oldLevel)      = do
           level <- execSG SenseMoisture
           execSG $ UpdateLight (toEnum level)
           if oldLevel >= 0 && level /= oldLevel then
             do
             case toEnum level of
               Red         -> execSG $ SendAlert "Your garden is completely dry!"
               Orange      -> if oldLevel > level then
                                execSG $ SendAlert "Your garden is almost dry!"
                              else
                                return ()
               FlashWhite  -> execSG $ SendAlert
                                       ("Your plants are getting over-watered,"
                                       ++ " turn off the water!")
               _           -> return ()
             threadDelay 1000000
             execSG $ LoopIt level
           else
             do
             threadDelay 1000000
             execSG $ LoopIt level

runWP :: WP.WeakPacket SmartGarden a -> IO a
runWP (WP.Primitive sg) = execSG sg

send :: RemoteMonad SmartGarden a -> IO a
send = unwrapNT $ runMonad (wrapNT runWP)

main :: IO ()
main = send $ loopIt (-1)
```

The loops that we are talking about above have more of an imperative style of loops.

In Haskell, the functions that have a loop-like behavior use one of the `fold` variants or use `map`. In the original Haxl paper [Marlow et al., 2014], the authors mention that a common way of doing bulk operations is through the use of `mapM` and `sequence`. They also mention that at the time of the paper (2014) `sequence` and `mapM` used monadic binds as the internal operations to complete the bulk of operations. To maximize the bundling they assigned the more widely used `mapM` and `sequence` functions to their applicative counterparts `traverse` and `sequenceA` respectively.

Fortunately, as of March 2015 the `mapM` and `sequence` where changed to be implemented in terms of the applicative versions. What this means is that with our applicative bundling we can take a list of `RemoteMonad` actions and combine them all with applicative actions, resulting in larger applicative packets. Imagine we have a list of user ids locally and we need to get the user profile information for each of these users to populate a friend list on a social networking site.

```
data UserService a where
  GetUserInfo :: UserId -> UserService UserProfile
  GetFriends  :: UserId -> UserService [UserId]

getUserInfo :: UserId -> RemoteMonad UserService UserProfile
getUserInfo uid = primitive $ GetUserInfo uid

getFriends :: UserId -> RemoteMonad UserService [UserId]
getFriends uid = primitive $ GetFriends uid

-- transmission method to call remote services
runAppPacket :: ApplicativePacket UserService a -> IO a

send :: RemoteMonad UserService a -> IO a
send = unwrapNT $ runMonad (wrapNT runAppPacket)

main :: IO ()
main = do
          let currentUser = UserId "892f3fe7-d42e-42c8-aae4-e2687422d558"
          uids <- getFriends currentUser
          profiles <- sequence $ mapM getUserInfo uids
```

In the example above, if we imagine we had someone with 100 friends on the networking site we would not want to make a hundred individual remote requests to populate the user's friends list with those users' information. But because `sequence` and `mapM` are using applicative functor operators we can send all 100 items in a single bulk request using the `ApplicativePacket`. Because we are making our optimizations at the applicative level, the user's code get's efficient packaging without having to worry about it since the underlying code is using applicative!

5.4 Remote Monad and Monad Transformers

In general, monads are created with a single purpose in mind. The `Reader` monad, for example, was created to allow a program to have access to read-only data. The `State` monad, as another example, was created to allow functions to carry around state as the process continues.

Monad transformers were created to compose these capabilities into a single monad with a `lift` function to connect the different layers.

```
lift:: m a -> t m a
```

To use monad transformers, we start with some base monad to which we add extra capabilities by layering monad transformers. There is no limit to the number of transformers that can be placed on top of one another.

As an example, we can look at using the State monad transformer (`StateT`) to add state to the IO monad:

```
counter :: StateT Int IO ()
counter =  do
        s <-  get
        lift $ putStrLn $ show s ++ " is my current state."
        put (s + 1)
```

With the `StateT Int IO` monad, we are able to call functions belonging to the State monad, such as `get`, as if we were in the State monad but we can also call an IO function by lifting that function to be of the right type. A different way of thinking about it is that we use `lift` to peel away each layer of the monad transformer stack, allowing us to call any functions belonging to the next outermost layer. In our above example, the outer layer is of type `StateT` so we have access to `State` functions. If we use `lift` once then the state layer is removed and we can now call `IO` functions

Below, we will add another monad transformer to have a combination of the Reader, State and IO monads. Since the outermost monad is the Reader monad, we can call `ask` without using `lift`. Calling `lift` one time will expose the State monad functions, and we can perform IO functions by calling `lift` a second time.

```
a :: ReaderT String (StateT Int IO) ()
a = do
    s <- lift get -- get state information
    r <- ask -- ask for read-only information
    lift $ lift $ putStrLn $ "Here we are: " ++ show s ++ " read-only info: " ++ r
    lift $ put (s + 1) -- put new state

fib' :: ReaderT String (StateT (Int, Int) IO) ()
fib'= do
    (n1,n2) <- lift get
    r <- ask
    lift $ lift $ putStrLn $
            "Here we are: " ++ show (n1 + n2)  ++ " read-only info: " ++ r
    lift $ put (n2,n2+n1)

fib :: Int -> ReaderT String (StateT (Int, Int) IO) ()
fib n = sequence_ $ replicate n fib'
```

We implemented the Fibonacci sequence by storing the next two numbers to be added in the program state, and have the Reader monad used solely for demonstration purposes. We then execute these transformers one at a time by calling the corresponding run functions:

```
> execStateT (runReaderT (fib 5) "read-only config") (0,1)
```

As we built the remote monad, a natural question came about: Can we create the remote monad as a monad transformer? Imagine taking any monad and instantly adding a mechanism that can add automatic bundling to our connection to a remote resource! In researching remote monads, we came to the conclusion that yes, we can make the remote monad be a monad transformer but there were subtle caveats which could cause the user to have issues that would be very difficult to debug. These caveats led our group to decide against making a remote monad transformer.

Overall, the remote monad appears to work as a monad transformer but as we look at how monad transformers are used in practice, the potential issues begin to manifest themselves. A monad transformer stack can have any number of transformers including multiple transformers of the same type. This works fine for having multiple `ReaderT` or `StateT` transformers in the stack, but in the case of remote monads, we are delaying computation but have the need to preserve ordering. What happens if we have multiple layers of remote monad transformers connected to the same resource? The ordering is then no longer guaranteed and the program has now opened the door to a host of bugs that would have the same characteristics as race-condition errors.

Another thing to consider is the fact that if we are using the state or reader monad transformers or some other monad transformer, then any time we bind a value to be used in our computation introduces a break in our bundling. If we needed to use state for all of our procedures, then this would inhibit our ability to make applicative packets as there will be a bind present after each procedure. This would cause the applicative bundling to be equivalent to the strong bundling as far as the packet distribution is concerned.

After discovering these two things, we feel that if a remote monad is desired in a monad transformer stack, then it should be used as the base monad. This would require one minor change to the library. In practice, the majority of monad transformer stacks contain the `IO` monad as the base or have the base monad an instance of `MonadIO` which allows `IO` actions to take place in the base monad. These `IO` actions would be taking place on the local machine and not sent to the remote server. Logging to a file or to the console is a great example of why we would want local `IO` in our monad and not just used when sending the primitives.

To make RemoteMonad an instance of MonadIO we add a new constructor:

```
data RemoteMonad  p a where
  Appl        :: RemoteApplicative p a -> RemoteMonad p a
  Bind        :: RemoteMonad p a -> (a -> RemoteMonad p b) -> RemoteMonad p b
  Ap'         :: RemoteMonad p (a -> b) -> RemoteMonad p a -> RemoteMonad p b
  IOAction    :: IO a -> RemoteMonad p a
```

We then change our runMonad function to specify that the natural transformation from RemoteMonad to some other monad requires it to be an instance of MonadIO. The MonadIO instance gives us a way to say, lift until the IO layer is reached and can be executed, the function is called liftIO.

Having the IOAction constructor is meant to distinguish any local IO actions from the actions that occur on the remote server. This could still allow multiple connections to the same resource through the local IO actions and the remote monad layer, but with this setup it is harder for a user to stumble into this class of errors on accident when compared to a remote monad transformer.

5.5 Real World Scenarios

In many real-world scenarios, a monad stack of transformers is used to create a monad that is tailored to handle the specific task for which the application was created. As an example, we can imagine a ReaderT monad transformer to give access to configuration settings, as well as an ExceptT to handle errors and possibly a WriterT transformer for logs. For simplicity we will have our customized monad use the ExceptT and ReaderT monad on top of IO:

```
type MyMonad a = ReaderT Config (EitherT CustomErrorType IO) a
```

As mentioned in the previous section, if the remote monad is used in a monad transformer stack, then it must be used as a base. We will now look at the effect of having it as the base of a transformer stack as well as how to use the remote monad when connecting to multiple databases or external resources.

5.5.1 ReaderT Monad

Having a `ReaderT` monad in the stack allows us to query the configuration `Config` at any time when we are in a function that has type `MyMonad a` where a can be any return value. Let's look at an example of a stuffed animal dog that can talk. With this dog, we can program the child's name which it then uses to say three things: "I love you, Ellie.", "Ellie, would you like to play?" and "Goodnight, Ellie!". We will store the child's name in the Reader context and we can have the following functions:

```
type Name = String
type Config = Name

speak :: String -> IO () -- split string into words and play audio for each word

love :: MyMonad ()
love = do
  name <- ask
  lift $ speak $ "I love you, "++ name

play :: MyMonad ()
play = do
  name <- ask
  lift $ speak $ name ++ ", would you like to play?"

sleep :: MyMonad ()
sleep = do
  name <- ask
  lift $ speak $ "Goodnight, "++ name

runMyMonad :: Config -> MyMonad a -> IO (Either CustomErrorType a)
runMyMonad c m =  runExceptT . runReaderT c $ m
```

Now in converting this to a remote monad, we use a GADT that contains each of the phrases and declare our evaluation function `interp`:

```
data Phrases a where
  Love  :: Phrases ()
  Play  :: Phrases ()
  Sleep :: Phrases ()

love, play, sleep :: Phrases
love = Love
play = Play
sleep = Sleep

interp :: Name -> WeakPacket Phrases a -> IO a
interp name (Primitive Love)  = speak $ "I love you, "++ name
interp name (Primitive Play)  = speak $ name ++ ", would you like to play?"
interp name (Primitive Sleep) = speak $ "Goodnight, "++ name
```

When we break up the actions into GADTs and an evaluation function, we can just add the configuration setting to the `interp` function removing the entire `ReaderT` monad. Yes, we could have done that with the previous way but we would have to add it to each of the `love`, `play`, and `sleep` functions. If we were to refactor our read-only data we would have to do it in each of those cases instead of in one place. In this case, our `interp` function would be the only function that would use the configuration setting in the `ReaderT` transformer and so we are either passing the setting once to the `runReaderT` function or once to the `interp` function. By passing the setting once to the `interp` function and removing the `ReaderT`, we remove the need to use a whole layer of lifts needed which in this example removes all the lifts for the program.

5.5.2 Multiple Databases

In real world applications, one of the most important things is performance. In many applications, there are different shapes and types of data for a single application. We have data that will be frequently accessed and needs to have a quick access time, but we also have data that is not as heavily used but we are required to sort the data based on different attributes. To have the most performant code, we might decide that the frequently accessed data should be stored in an in-memory database such as Redis and that the sortable data should be stored in a SQL or NOSQL database.

The whole idea of bundling requests requires that they be sent to the same location. If two packets are headed in different directions, there are no tricks to perform to bundle them together. This section gives an example of how we can still use the remote monad in this case, bundling where possible.

When we have primitives that are going in different directions we can create separate GADTs for each of our operations based on the destination and then have a main GADT that will contain all the destination specific GADTs. Here is what that would look like if we were wanting to connect to an SQL database and a Redis database.

```
data RedisFunctions a where
  getFollowers :: UUID -> RedisFunctions [UUID]
  getFavorites :: UUID -> RedisFunctions [UUID]

data SQLFunctions a where
  getProfileInfo :: UUID -> SQLFunctions ProfileInfo
  getAlbumsByArtist :: ArtistName -> SQLFunctions [Album]

data AllFunctions a where
  redisF :: RedisFunctions a -> AllFunctions a
  sqlF   :: SQLFunctions a -> AllFunctions a

interp :: ApplicativePacket AllFunctions a -> IO a
```

Our `interp` function will then have a staging function that could build up and batch requests that are going to the same destination. We can imagine an `n`-tuple in this case that has the queued commands and procedures for each of the `n` destinations of the remote primitives.

Chapter 6

Case Studies

After covering the theory of remote monads in previous chapters, we will now look at several case studies that are built upon the remote monad library. These case studies vary from implementing an established RPC protocol, to transforming our primitives into a different programming language for evaluation, to interacting with a local database in the form of Plists. These case studies are identified below:

- **blank-canvas** - Haskell to HTML5 Canvas library
- **remote-json** - Remote JSON RPC implementation
- **remote-binary** - More efficient RPC library using byte strings
- **haskino** - Haskell to Arduino library
- **PlistBuddy** - Plist file editor communicating with an interactive shell called Plist-Buddy

We will also look at one case study of Facebook's `Haxl` library which does not use the `remote-monad` library but does use the ideas of the remote monad. This library is a remote monad that is specialized in the domain where all procedures are read-only queries, which allows us to include another bundling strategy.

The goal of this chapter is to show how general remote monads can be, to give the reader a sense of the common elements needed to use the remote monad, as well as to view the different layers that can be placed in the network stacks to result in a specific desired behavior. We will start with an in-depth look at how `blank-canvas` is built using the remote-monad framework, and follow it with a basic look at remote monad specific points of the other case studies.

6.1 Case Study: Blank Canvas

Blank Canvas is a Haskell binding to the browser's complete HTML5 Canvas library that supports the weak, strong and applicative bundling strategies of the remote monad. The remote part of this library is the interaction with a canvas object in the browser via JavaScript. The canvas element, present in HTML5, has 6 categories of functions as shown in Table 6.1. With these functions, we can draw text, curves, lines, paths or images, to name just a few of the capabilities. When given a Canvas object, we can get the context of the object and call its various methods. Here's what we need to do to give our Haskell program access to the Canvas object:

- Create a GADT to mirror the Canvas abilities that we would like to call
- Translate the Haskell GADT into JavaScript
- Connect our Haskell program to the browser with a mechanism to execute the sent JavaScript code

Besides the fact that our serialization layer is not just encoding our GADT to bits but instead is serializing to a different language, this setup is similar to what we have seen thus far in the small examples that we used to look at the theory of the remote monad. Something that we did not mention above, but will need to support, is the fact that a user can interact with the browser by creating events that should be sent to our Haskell program. These

Table 6.1: JavaScript API for HTML5 Canvas

	Transformation
void	`save`()
void	`restore`()
void	`scale`(float x,float y)
void	`rotate`(float angle)
void	`translate`(float x,float y)
void	`transform`(float m11,float m12,float m21,float m22,float dx,float dy)
void	`setTransform`(float m11,float m12,float m21,float m22,float dx,float dy)
	Text
void	`fillText`(string text,float x,float y,[Optional] float maxWidth)
void	`strokeText`(string text,float x,float y,[Optional] float maxWidth)
TextMetrics	`measureText`(string text)
	Paths
void	`beginPath`()
void	`fill`()
void	`stroke`()
void	`clip`()
void	`moveTo`(float x,float y)
void	`lineTo`(float x,float y)
void	`quadraticCurveTo`(float cpx,float cpy,float x,float y)
void	`bezierCurveTo`(float cp1x,float cp1y,float cp2x,float cp2y,float x,float y)
void	`arcTo`(float x1,float y1,float x2,float y2,float radius)
void	`arc`(float x,float y,float radius,float startAngle,float endAngle,boolean d)
void	`rect`(float x,float y,float w,float h)
boolean	`isPointInPath`(float x,float y)

Fonts, colors, styles and shadows (attributes)	
`globalAlpha` float	`globalCompositeOperation` string
`lineWidth` float	`lineCap` string
`lineJoin` string	`miterLimit` float
`strokeStyle` any	`fillStyle` any
`shadowOffsetX` float	`shadowOffsetY` float
`shadowBlur` float	`shadowColor` string
`font` string	`textAlign` string
`textBaseline` string	

	Drawing
void	`drawImage`(Object image,float dx,float dy, [Optional] . . .)
void	`clearRect`(float x,float y,float w,float h)
void	`fillRect`(float x,float y,float w,float h)
void	`strokeRect`(float x,float y,float w,float h)
	Style Attributes
CanvasGradient	`createLinearGradient`(float x0,float y0,float x1,float y1)
CanvasGradient	`createRadialGradient`(float x0,float y0,float r0,float x1,float y1,float r1)
CanvasPattern	`createPattern`(Object image,string repetition)
	Images
string	`toDataURL`([Optional] string type, [Variadic] any args)
ImageData	`createImageData`(float sw, float sh)
ImageData	`getImageData`(float sx, float sy, float sw, float sh)
void	`putImageData`(ImageData imagedata, float dx, float dy, [Optional] . . .)

events take the shape of key presses or mouse events that we would like to affect what we draw.

6.1.1 Canvas GADT

After we get the context from the canvas object we can call the different methods and set the variables shown in Table 6.1. Here is a subset of what this looks like as a simple data type:

```
data  Method a where
    Arc           :: (Double, Double, Double, Radians, Radians, Bool) -> Method ()
    BeginPath     :: Method ()
    BezierCurveTo :: (Double, Double, Double, Double, Double, Double) -> Method ()
    ClearRect     :: (Double, Double, Double, Double) -> Method ()
    FillRect      :: (Double, Double, Double, Double) -> Method ()
    FillText      :: (Text, Double, Double) -> Method ()
    Rect          :: (Double, Double, Double, Double) -> Method ()
    Scale         :: (Interval, Interval) -> Method ()
    TextAlign     :: TextAnchorAlignment -> Method ()
    ToDataURL     :: Method Text
    MeasureText   :: Text -> Method TextMetrics
    IsPointInPath :: (Double, Double)-> Method Bool
    ...
```

Now that we have built out the data type that mirrors the corresponding canvas functions, we will need to create a JavaScript representation of each of these Haskell objects.

```
instance InstrShow Method where
  showi (Arc (a1,a2,a3,a4,a5,a6)) = "arc("
         <> jsDouble a1 <> singleton ',' <> jsDouble a2 <> singleton ','
         <> jsDouble a3 <> singleton ',' <> jsDouble a4 <> singleton ','
         <> jsDouble a5 <> singleton ',' <> jsBool a6   <> singleton ')'
  showi BeginPath = "beginPath()"
  showi (BezierCurveTo (a1,a2,a3,a4,a5,a6)) = "bezierCurveTo("
         <> jsDouble a1 <> singleton ',' <> jsDouble a2 <> singleton ','
         <> jsDouble a3 <> singleton ',' <> jsDouble a4 <> singleton ','
         <> jsDouble a5 <> singleton ',' <> jsDouble a6 <> singleton ')'
  showi (ClearRect (a1,a2,a3,a4)) = "clearRect("
         <> jsDouble a1 <> singleton ',' <> jsDouble a2 <> singleton ','
         <> jsDouble a3 <> singleton ',' <> jsDouble a4 <> singleton ')'
  showi (ToDataURL) = "canvas.toDataURL()"
  ...
```

Each of the constructors in `Method` is going to take the form of `ctx.method(args)` for the methods or `ctx.var = val` to set the variables. The `InstrShow` only includes the actual method or the right side of the `ctx` object, as the full variable assignment or method call will be built up later.

When programming in the browser, it is helpful, if not necessary, to use other JavaScript functions that are not associated with the canvas object. For example, our debugging will go much smoother if we are able to use JavaScript's `console.log()` method. So let's create an additional data type that will house any other JavaScript methods to which we would like access, whether that is a top-level method or one that belongs to a different object. For now, we will just have the constructor that will give us access to the console log command:

```
data Command =
    forall msg . JSArg msg => Log msg

instance InstrShow Command where
  showi (Log msg) = "console.log(" <> showiJS msg <> singleton ')'
```

where the `JSArg` indicates that we have a JavaScript representation of the Haskell data type.

If we look at Table 6.1 again, we can see that almost all of the methods are returning `void`, but for the gradient and pattern methods, we return a new object that has its own methods and can later be applied to our context stroke, text, and fill styles. The problem is that we do not have a meaningful representation in Haskell for these pattern and gradient objects. What we need to do is have a way of storing a variable remotely and return back a reference to that variable to be used later. Here is a JavaScript example of what we would like to recreate:

```
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");

// Create gradient
var grd = ctx.createLinearGradient(0,0,200,0);
grd.addColorStop(0,"red");
grd.addColorStop(1,"blue");

// Fill with gradient
ctx.fillStyle = grd;
ctx.fillRect(10,10,150,80);
```

The return of our `createLinearGradient` creates a JavaScript object with methods that can later be called. Let's create a third data type to describe these types of interactions:

```
data PseudoProcedure :: * -> * where
  CreateLinearGradient :: (Double,Double,Double,Double)
                       -> PseudoProcedure CanvasGradient
  CreateRadialGradient :: (Double,Double,Double,Double,Double,Double)
                       -> PseudoProcedure CanvasGradient
  CreatePattern        :: Image image => (image, RepeatDirection)
                       -> PseudoProcedure CanvasPattern
```

Where `CanvasGradient` and `CanvasPattern` are just wrappers around `Int` to help us with variable naming. The JavaScript translation at this point is still just the basics of the call and does not include where the method belongs:

```
instance InstrShow (PseudoProcedure a) where
  showi (CreateLinearGradient (x0,y0,x1,y1)) = "createLinearGradient("
        <> jsDouble x0 <> singleton ',' <> jsDouble y0 <> singleton ','
        <> jsDouble x1 <> singleton ',' <> jsDouble y1 <> singleton ')'
  showi (CreateRadialGradient (x0,y0,r0,x1,y1,r1)) = "createRadialGradient("
        <> jsDouble x0 <> singleton ',' <> jsDouble y0 <> singleton ','
        <> jsDouble r0 <> singleton ','
        <> jsDouble x1 <> singleton ',' <> jsDouble y1 <> singleton ','
        <> jsDouble r1 <> singleton ')'
  showi (CreatePattern (img,dir)) = "createPattern("
        <> jsImage img <> singleton ',' <> jsRepeatDirection dir <> singleton ')'
```

With these three data types we can combine them all to have the GADT that will be used to parameterize the remote monad.

```
data Prim :: * -> * where
  Method          :: Method  -> CanvasContext -> Prim ()
  Command         :: Command -> Prim ()
  PseudoProcedure :: InstrShow a => PseudoProcedure a
                  -> a -> CanvasContext -> Prim ()
```

Now we can combine our previous instances of `InstrShow` together into `Prim` to get the full JavaScript methods and variables.

```
  instance InstrShow a => InstrShow (Prim a) where
    showi (PseudoProcedure f i c) = showi i <> singleton '=' <> jsCanvasContext c
                                            <> singleton '.' <> showi f
    showi (Method m x)      = jsCanvasContext x <> singleton '.' <> showi m
    showi (Command c _)     = showi c
```

6.1.2 Communicating with the Web Browser

Since we can now generate JavaScript from our Haskell data types, we just need to create a way to communicate with the browser to execute our JavaScript as well as a way to have dual communication for replies and to handle events that are generated by the client via the browser.

Luckily for us, there is a JavaScript function called `eval` that will take some text and execute it as JavaScript code.

```
> eval ("var c = 4 + 1; console.log(\"4 + 1 =  \" + c)")
4 + 1 =  5
```

WebSockets [Fette, 2011] is a protocol that allows us to have full communication between the client and the server. Before this technology, programs would have to poll the server to check for changes. Now instead of doing a busy wait, our Haskell program can just connect to the server and listen for any events that may be generated.

For the browser side of things, our Haskell program will serve up an HTML document that contains a canvas object, and some JavaScript functions to help with the setup. Below we have some fragments that are found in the HTML document. These fragments set up the WebSocket which will evaluate the JavaScript text that the Haskell program will send as well as send responses back for procedures by having a reply promise.

```
...
jsb = new WebSocket('ws://localhost:'+location.port+'/');
jsb.onmessage = function(evt){
  var reply = function(n,obj){
      Promise.all(obj).then(function(obj){
        jsb.send(JSON.stringify([n].concat(obj)));
      });
  };
eval('(function(){' + evt.data + '})()');
};
```

The way that we will be sending back the results to procedures is by saving the result to variables and when the procedures are finished we can query the variables and send them back through the WebSocket.

6.1.3 Bundling in `blank-canvas`

The `blank-canvas` library supports the weak, strong and applicative bundling. To take advantage of the remote-monad framework's automatic bundling we need to define a `KnownResult` instance for our primitives as well as smart constructors to lift it into the `RemoteMonad`.

```
  instance KnownResult Prim
    knownResult Method {}          = Just ()
    knownResult Command {}         = Just ()
    knownResult PseudoProcedure {} = Just ()
    knownResult Query {}           = Nothing

arc :: (Double, Double, Double, Radians, Radians, Bool) -> Canvas ()
arc = primitive . Method . Arc
```

In order to make life easier for the `KnownResult` instance, we separated the *commands* and*procedures* from `Method` by placing the procedures, like `IsPointInPath`, into a `Query` data type. This way we do not have to list every single primitive but can pattern match on the encapsulating constructors. This also greatly simplifies any future functionality that could be added.

To finish the bundling for this case study, we need to define our send method which will combine our JavaScript text together and send them through the WebSocket to the web browser.

After the underlying monad has split the actions into packets, the send method will take that packet and serialize it to JavaScript text. If our packet only contains commands, then we can simply call our `showi` functions on the primitives and separate them with a semi-colon. In the case that we come across procedures or variables are being set, we parse out the future variable names and then we append a call to the `reply` function, with those variable names to the JavaScript text. With the call to `reply`, we include a nonce to help us grab the correct results for the corresponding request. When Haskell program receives data from the client's browser via the WebSocket, we store the results in a map where the nonce is a key. Then we can parse the reply by checking the map for the nonce that we sent with our requests.

6.1.4 Handling Events

In `blank-canvas` there are a number of configuration options that allow us to set the port, the bundling strategy, debug or profiling flags. In those options, there is also a way that we can specify which events we want sent to our Haskell program. By using WebSockets to communicate with the web browser, we can register for the events and when they occur we build up an object that can be deserialized into Haskell.

```
function Trigger(evt) {
    var o = {};
    o.metaKey = e.metaKey;
    o.type    = e.type;
    ...
    jsb.send(JSON.stringify(o));
}
function register(name) {
    $(document).bind(name,Trigger);
}
```

On the Haskell side of things, we store the events that take place in a thread-safe queue and then we can call a `wait` or `flush` function to block until an event enters the queue or to pull all the events in the queue (if any) respectively. Here is an example of a function that will draw a square in the middle of the screen and then if a user clicks the mouse on the canvas, then it will draw another square at the cursor location.

```
main = blankCanvas 3000  events = ["mousedown"]  $ \ context -> do
          let loop (x,y)= do
                send context $ do
                        save()
                        translate (x,y)
                        beginPath()
                        moveTo(-100,-100)
                        lineTo(-100,100)
                        lineTo(100,100)
                        lineTo(100,-100)
                        closePath()
                        lineWidth 10
                        stroke()
                        restore()

                event <- wait context
                case ePageXY event of
                        Nothing -> loop (x,y)
                        Just (x',y') -> loop (x',y')

          let (w,h) = (width context, height context)
          loop (w / 2, h / 2)
```

More examples of `blank-canvas` are found in Chapter 7, as the comparisons between the different bundling strategies were made from the `blank-canvas` benchmarks.

6.2 Case Study: remote-json

As our first case study, we take a well-established protocol, JSON-RPC [Group et al., 2012], and use the remote monad as the interface to this protocol. JSON-RPC is a simple protocol for remote procedure calls, typically over HTTP, supporting both synchronous remote methods calls and asynchronous notifications.

To give an example of the protocol from the JSON-RPC specification, consider calling a method `subtract`, with the arguments 42 and 23. The following transaction would occur:

$$\begin{tikzcd}
\textbf{Client} & & \textbf{Server} \\
\llbracket RPC \rrbracket \arrow[d, mapsto] & & \\
\llbracket RemoteMonad \rrbracket \arrow[d, mapsto] & & \\
\llbracket Packet \rrbracket \arrow[d, mapsto] & & \llbracket Call \rrbracket \\
\llbracket SendAPI \rrbracket \arrow[rr, mapsto, dotted] & & \llbracket ReceiveAPI \rrbracket \arrow[u, mapsto]
\end{tikzcd}$$

Figure 6.1: remote-json Network Stack

```
--> {"jsonrpc": "2.0", "method": "subtract",
     "params": [42, 23], "id": 1}
<-- {"jsonrpc": "2.0", "result": 19, "id": 1}
```

Following the notation from the JSON-RPC specification, `-->` is the data sent from client to server, and `<--` is the server's response. The packets being sent in either direction are simple JSON objects.

The fact that the JSON-RPC protocol supports batching and is easy to debug, because of the easy to read JSON structures, makes it an obvious choice for a case study of the remote monad. Furthermore, by using JSON-RPC we can implement our clients in Haskell, and use existing JSON-RPC services written using any language or framework.

To investigate the generality of our remote monad framework, we implemented a complete JSON-RPC stack, both client and server, called `remote-json` which is available on Hackage. The network stack of this library is in Figure 6.1.

6.2.1 Design of remote-json

We start our design using the `RPC` monad, which is a `newtype` wrapper around a `RemoteMonad`. The JSON-RPC, like the remote monad design pattern, has two types of calls, methods (that have a result), and notifications (which never have a result). JSON-RPC methods and notifications map directly on remote monad procedures and commands, though in practice, notifications are rarely seen in JSON-RPC APIs.

```
-- The monad
newtype RPC a = RPC (RemoteMonad Prim a)
  deriving (Monad, Applicative, Functor)

-- Primitive Types
data Prim :: * where
    Notification :: Text -> Args -> Prim ()
    Method  :: FromJSON a => Text -> Args -> Prim a

data Args where
    List :: [Value] -> Args
    None ::         -> Args

--specify commands and procedures
instance KnownResult Prim where
   knownResult Notification = Just ()
   knownResult Method       = Nothing

-- the remote send
send :: Session -> RPC a -> IO a
```

This API gives an (aeson) Value-based access to JSON-RPC. JSON-RPC's `Methods` do have one difference from the remote monad's `procedures`. `Methods` in JSON-RPC use an id to tag a result so that responses can be matched up with the requests. Because of the modular network stack, we can put an extra layer that adds the unique id when we send the request. For the weak and strong bundling strategies, we are sending at most one procedure, which means the unique id is not used or can trivially be set to be 1 each request. For the applicative bundling, each applicative packet will have a processing function that uses state to keep track of the next available id, as well as a layer to translate a packet of primitives into JSON batches:

```
go:: forall a . AP.ApplicativePacket Prim a
             -> State IDTag ([JSONCall], HM.HashMap IDTag Value -> IO a)

data JSONCall :: * where
  NotificationCall :: Prim a             -> JSONCall
  MethodCall       :: ToJSON a => Prim a -> Value -> JSONCall

instance ToJSON JSONCall where
  toJSON (MethodCall (Method nm args) tag) = object $
          [ "jsonrpc" .= ("2.0" :: Text)
          , "method" .= nm
          , "id" .= tag
          ] ++ case args of
                 None -> []
                 _    -> [ "params" .= args ]
  toJSON (NotificationCall (Notification nm args)) = object $
          [ "jsonrpc" .= ("2.0" :: Text)
          , "method" .= nm
          ] ++ case args of
                 None -> []
                 _    -> [ "params" .= args ]
```

6.2.2 Sessions

In the `remote-json` framework we use sessions to describe which bundling strategy to use in communicating with the server. These sessions will also account for how we are sending the values synchronously or asynchronously. We can create a JSON-RPC instance using `weakSession`, `strongSession` or `applicativeSession`. Each of these takes an encoding of how to send `Values` to a remote server, and returns a `Session`.

```
weakSession        :: (SendAPI :~> IO) -> Session
strongSession        :: (SendAPI :~> IO) -> Session
applicativeSession :: (SendAPI :~> IO) -> Session
```

Because we want to allow the user to mark an RPC as a synchronous or asynchronous communication of JSON. We add a wrapper `SendAPI` that designates the `Values` to be sent as an asynchronous request or synchronous.

```
data SendAPI :: * -> * where
  Sync  :: Value -> SendAPI Value
  Async :: Value -> SendAPI ()
```

So, calling `send` with the `RPC` monadic remote commands and a given `Session` will bundle the monad into packets specified by the `Session`, and then call the transmission function found in the session with the values either wrapped with `Sync` or `ASync`.

We can write our own matcher for `SendAPI`, or use `remote-json-client`, which provides a function that, when given a URL, returns the `SendAPI` to `IO` natural transformation using the `wreq` library.

```
clientSendAPI :: String -> (SendAPI :~> IO)
```

Putting this together, we get

```
main :: IO ()
main = do
  let s = applicativeSession
            (clientSendAPI "http://www.wibble.com/wobble")
  (t,u) <- send s $ do
             say "Hello, "
             t <- temperature
             say "World!"
             u <- uptime "orange"
             return (t,u)
  putStrLn "Temperature: " ++ show t
  putStrLn "Orange Uptime: " ++ show u
```

With this example, it is clear that we are sending to a particular website without any extra code about bundling obfuscating the main logic of the program. We could make calls to another remote server and even use a different bundling strategy with those interactions by using `send` with a different session.

Package	Bundling	Server	LOC	Version
jsonrpc-conduit	List	✔	178	0.3.0
json-rpc-client	Applicative		191	0.2.4.0
hs-json-rpc			237	0.0.0.1
colchis		✔	283	0.2.0.3
remote-json	**Monad**	✔	**416**	**0.2**
json-rpc	List	✔	681	0.7.1.1
jmacro-rpc	List	✔	754	0.3.2

Table 6.2: Hackage JSON-RPC Client Libraries

```
main :: IO ()
main = do
  let wibble = applicativeSession
           (clientSendAPI "http://www.wibble.com/wobble")
  let weather = weakSession
           (clientSendAPI "http://weather.example.com")
  (t,u) <- send wibble $ do
             say "Hello, "
             t <- temperature
             say "World!"
             u <- uptime "orange"
             return (t,u)
   w <- send weather $ do
              _ <- setLocation "Lawrence,KS"
              w <- getWeatherAlerts

  putStrLn "Temperature: " ++ show t
  putStrLn "Orange Uptime: " ++ show u
  putStrLn "Weather Alerts for Lawrence, KS: " ++ show w
```

6.2.3 Comparison with other JSON-RPC libraries

The `remote-json` library is small, complete, and has the useful feature of automatic bundling because of the usage of the remote monad design pattern. In order to better understand the impact of the remote monad abstraction, we have compared our `remote-json` library with other JSON-RPC client libraries available on Hackage. Table 6.2 summarizes our findings. We use the `cloc` tool to count lines of Haskell code, and have manually inspected haddock and the source to determine functionality.

Our JSON-RPC library is the only library to support monadic code directly. Interestingly, there is an existing library,`json-rpc-client` that did support the applicative interface. A number of libraries support manual bundling using lists, where the user would need to explicitly extract the result from a list.

6.3 Case Study: Remote Binary

The `remote-binary` library is an instance of the remote-monad that focuses on efficiency. Instead of allowing the user to pick the bundling strategy and sending the packets as strings, the `remote-binary` library will send GADTs as applicative packets and send a binary encoding of the packets, instead of plain text, over a user-specified network transport mechanism.

To demonstrate this transformation using `remote-binary`, let's take a simple stack example and cause it to run remotely:

```
data Prim :: * where
   Push :: Int -> Prim ()
   Pop :: Prim Int
 deriving (Show)

evalPrim :: Prim a -> State [Int] a
evalPrim (Push n) = modify (n:)
evalPrim (Pop) = do
        st <- get
        case st of
          []  -> error "Can't pop an empty stack"
          (x:xs) -> do
                 put xs
                 return x
```

Here we have a very simple stack example running locally using a GADT and an evaluation function to perform the stack operations. In order to execute this remotely using `remote-binary`, we need to:

- show which primitives are commands and which ones are procedures using the `KnownResult` class
- specify a binary encoding for our `Push` and `Pop` methods
- specify a transport mechanism to send them to the server

```
instance KnownResult Prim where
  knownResult (Push _) = Just () -- command (we know the result statically)
  knownResult  Pop     = Nothing -- procedure
```

By their very nature, we see that remote-monad `procedure`s return some value `a`, which means we also need to supply the server a way to encode the resulting value so it can transmit the result back to our local process in the correct encoding.

The `BinaryQ` class will be used for the binary encoding:

```
class BinaryQ p where
   putQ    :: p a -> Put
   getQ    :: Get (Fmap p Put)
   interpQ :: p a -> Get a

data Fmap f a where
   Fmap :: (a -> b) -> f a -> Fmap f b

instance Functor (Fmap f) where
   fmap f (Fmap g h) = Fmap (f . g) h
```

`putQ` is used locally to encode the GADT prior to transmission and `getQ` is used remotely to decode the transmitted GADT. The `getQ` function will also provide the server the mechanism for encoding the evaluated result. This mechanism uses a new data type called `Fmap` to house the decoded data type as well as the function to apply to the result of the computation The `interpQ` function is used on the local side to interpret the result of the remote evaluation, using the sent packet as the witness to ensure the correct type is decoded. In the stack example above, if the `interpQ` function has an encoded value with a witness packet that contains the `Pop` operation, then it would know that it should be decoding the value as an `Int` because of the GADTs type.

The `remote-binary` library provides the `BinaryQ` instances for the different remote monad packets. Additionally, the packets require the user's GADT to be an instance of `BinaryQ` in order to encode it with the packet and to encode any results to be returned. Below are the instances for our example:

```
instance BinaryQ Prim where
   getQ = do i <- get
             case i :: Word8 of
               0 -> do
                     j <- get
                     return $ Fmap put (Push j)
               1 -> return $ Fmap put Pop
               _ -> error "Expected a Prim but got something else"
   putQ (Push n) = do
                    put (0 :: Word8)
                    put n
   putQ (Pop)= put (1 :: Word8)

   interpQ (Push _) =  return ()
   interpQ (Pop) = get
```

In our example we have two constructors for the `Prim` user GADT. For the local encoding, we can put a `0` to indicate that we have encountered a `Push` constructor and `1` for a `Pop` constructor. Then on the remote decoding, we check for a `0` or a `1` to know if we should be building up a `Push` object or `Pop` object.

Since `Pop` returns an `Int`, and `Int` is already an instance of `Binary`, we can just use `Binary`'s `put` function for the encoding strategy for the result in the `Fmap` type and `get` for the interpretation of the result.

Now that we have a way of encoding packets of our `Prim`, we can use natural transformations to take our transport function, which can send byte strings, and create a function that can bundle our `RemoteMonad` into applicative packets and then send them as byte strings to the remote server.

To complete the network stack, on the server side we need a mechanism to receive and decode byte strings, to evaluate our GADT when wrapped in an `Applicative` packet, and reply with the encoded result. We chose the `Applicative` bundling strategy to give us the most efficient bundling strategy.For our current example, we have decided to use `sockets` for the transportation layer.

The simplest packet in the remote monad is the `WeakPacket`, containing a single `Prim` construct. If we create a function that works over a `Prim` that is wrapped in a `WeakPacket` then we can promote it to a function that works over `ApplicativePacket`s by using the exported `promote` function. This `promote` function is not specific to `remote-binary` but is defined in the `remote-monad` framework. By using `promote` we only have to worry about how to handle a single primitive but this would prohibit us from doing things in parallel as we would be evaluating each primitive individually.

```
--Dispatches Prims to evaluation function
-- uses TMVar to be thread-safe
dispatchWP :: (TMVar [Int])->
              WeakPacket Prim a -> IO a

--Lift into a natural transformation
--and promote to handle ApplicativePacket
runAppPacket::  TMVar [Int] ->
(ApplicativePacket Prim :~> IO)
runAppPacket var =  promote $ nat (dispatchWP var)
```

The `remote-binary` library has a `server` function that wraps around our applicative packet handler and takes care of the result serialization:

```
server :: (BinaryQ p)=> (AP.ApplicativePacket p :~> IO) -> (SendAPI :~> IO)
```

We can use the `runAppPacket` function with the exported `server` function with the socket creation and handling of requests to complete the network stack for the server. The full network stack is in figure 6.2.

$$\begin{tikzcd}
\textbf{Client} & & \textbf{Server} \\
\llbracket RemoteMonad \rrbracket \arrow[d, mapsto] & & \\
\llbracket ApplicativePacket \rrbracket \arrow[d, mapsto] & & \left\llbracket \begin{array}{c} Fmap \\ ApplicativePacket \end{array} \right\rrbracket \\
\llbracket SendAPI \rrbracket \arrow[rr, mapsto, dotted] & & \llbracket SendAPI \rrbracket \arrow[u, mapsto]
\end{tikzcd}$$

Figure 6.2: Remote Binary Network Stack

6.3.1 Remote Exceptions

The way that we handle exceptions in `remote-binary` is by putting a layer before the server code which will catch any exceptions. This layer is found in the `server` function. If an exception was thrown from the server evaluation code, then we convert that to a `RemoteBinaryException` which takes the exception text as an input parameter. By adding an instance of Binary to `RemoteBinaryException` we can send the exception back to the local process and then allow the user to catch the exception. The `server` function precedes each result with an error byte which will tell the client to either decode the result or an exception.

By doing this, we eliminate the ability for the local code to catch specific exceptions since all exceptions will then be `RemoteBinaryExceptions`. If the user desires this capability, then they can simply write a `read` instance to parse the `displayException` text back into an exception.

6.4 Case Study: Haskino

The Haskino library allows a programmer to use Haskell to program the Arduino series of micro-controller boards, in both a tethered mode for easy debugging and an untethered mode for standalone operation. The tethered mode uses a byte code interpreter running on the Arduino, while the standalone mode compiles the monadic Haskell code to C, which is then linked with a small runtime system. Haskino was originally written with its own implementation of the remote monad design pattern [Grebe & Gill, 2016] as the remote-monad library was being created and the remote monad ideas were just taking shape. The original author has since ported the Haskino package to use the remote monad package instead, making use of the applicative packet [Grebe & Gill, 2017]. The communication stack of this library is in Figure 6.3.

The porting of the Haskino library to use the remote monad package took approximately 10 hours, demonstrating the relative ease of use of the package. The Haskino library instantiates domain-specific commands and procedures as `ArduinoPrimitive`s. There are over 50 commands and over 300 procedures within this library, a bulk of which are if-then-else statements covering different types or iteration expressions. For the tethered, interpreted mode, the `sendApp` function sends an Arduino monad to a tethered Arduino over an `ArduinoConnection`. An `ArduinoConnection` is an abstract handle to the remote Arduino, similar to the `Session` structure of the `remote-json` library. The `sendApp` uses the `runAP` function to package the monad into byte code which may be run on the remote Arduino board by the interpreter.

```
-- The monad
newtype Arduino a = Arduino (RemoteMonad ArduinoPrimitive a)
  deriving (Functor, Applicative, Monad)

-- The remote send
sendApp :: ArduinoConnection -> Arduino a -> IO a
sendApp c (Arduino m) = (run $ runMonad $ nat (runAP c)) m

-- Translate to bytecode
runAP :: ArduinoConnection -> ApplicativePacket ArduinoPrimitive a -> IO a
```

The Haskino DSL consists of a deep embedding which includes conditionals, loops, and threads, requiring entire code blocks to be run remotely. Haskino implements these as commands, which take another Arduino monad as a parameter to the command as shown in the examples below.

```
loopE       :: Arduino () -> Arduino ()
ifThenElse  :: Expr Bool -> Arduino ()
            -> Arduino () -> Arduino ()
createTaskE :: Expr Word8 -> Arduino ()
            -> Arduino ()
```

The embedded code blocks in these commands are encoded differently, not requiring bundling, but instead encoding the entire monad into one binary stream. Porting to use the remote monad library was straightforward here as well. A `packageCodeBlock` function takes the Arduino monad and returns a `ByteString` with the encoded code block. Internally, it calls `packMonad` and`packApplicative` functions that reify the internal structure that consists of `RemoteMonad`s and `RemoteApplicative`s into the Haskino protocols code block format.

Finally, the Haskino compiler performs a reification similar to `packageCodeBlock`, instead transforming the remote monad into C code as opposed to Haskino protocol byte code.

We will finish this case study with an example of a haskino program which uses the Arduino as a night light that also plays songs. This Haskell should be very familiar to those who have programmed using an Arduino.

$$\begin{tikzcd}
\textbf{Host} & & \textbf{Arduino Board} \\
\llbracket Arduino \rrbracket \arrow[d, mapsto] & & \\
\llbracket RemoteMonad \rrbracket \arrow[d, mapsto] & & \\
\llbracket ApplicativePacket \rrbracket \arrow[rr, dotted, mapsto] & & \llbracket Interpreter \rrbracket
\end{tikzcd}$$

Figure 6.3: Haskino Communications Stack

```
--- pin numbers for components
photocell :: Word8
led :: Word8
buzzer :: Word8
---------------
nightLight :: Arduino ()
nightLight = do
  setPinMode buzzer OUTPUT
  setPinMode led OUTPUT
  loop $ do
    l <- analogRead photoCell
    -- turn on the LED and play a song if the room is dark
    if l < 500 then --
      do
        digitalWrite led True
        leanOnMeSong
    else
      digitalWrite led False

nightLightExample :: IO ()
nightLightExample = withArduino True "/dev/cu.usbmodem1421" nightLight

leanOnMeSong :: Arduino () -- series of tones and pauses to create song
```

Table 6.3: PlistBuddy commands used to modify and read values in a plist file

PlistBuddy Commands	
Help	Prints this information
Exit	Exits the program. Changes are not saved to the file.
Save	Saves the current changes to the file.
Revert	Reloads the last saved version of the file.
Clear *type*	Clears out all existing entries, and creates a root of type=*type*.
Print [*entry*]	Prints value of *entry*. If an entry is not specified, prints entire file.
Set *entry value*	Sets the value at *entry* to *value*.
Add *entry type [value]*	Adds *entry* with type *type* and optional value *value*.
Copy *entrySrc entryDst*	Copies the *entrySrc* property to *entryDst*.
Delete *entry*	Deletes *entry* from the plist.
Merge file [*entry*]	Adds the contents of plist file to *entry*.

Table 6.4: List of operations that can be performed on a plist using PlistBuddy.

6.5 Case Study: PlistBuddy

Property list files, most commonly seen in macOS and iOS, are files that store user or application settings with the file name extension `.plist`. These files store the objects in an XML or JSON format. There is a command line utility called `PlistBuddy` which gives the user an interactive shell to read and write values in these plists.

The functional group at KU created a Haskell library that could send commands to the interactive shell, which fits nicely with the remote monad principles. This usage of the remote monad can be looked at as using the remote monad to connect to a database. Table 6.4 shows the available actions as shown in the OSX 10.9 `man` page. Encoding these actions into a GADT structure is a straightforward task:

```
data PlistPrimitive a where
  Help   :: PlistPrimitive Text
  Exit   :: PlistPrimitive ()
  Save   :: PlistPrimitive ()
  Revert :: PlistPrimitive ()
  Clear  :: Value -> PlistPrimitive ()
  Get    :: [Text] -> PlistPrimitive Value
  Set    :: [Text] -> Value -> PlistPrimitive ()
  Add    :: [Text] -> Value -> PlistPrimitive ()
  Delete :: [Text] -> PlistPrimitive ()
  ImportData :: [Text] -> ByteString -> Trail -> PlistPrimitive ()
  MergeDate :: [Text] -> UTCTime -> Trail -> PlistPrimitive ()
```

The only difference between our GADT and the PlistBuddy `man` page is that `Print` has been changed to `Get` in the GADT since we want to obtain the result of the key instead of just printing the value to the screen. As most of the operations are writing to a file, we know the result will be () except in the case of `Help` and `Get` commands which will return some text. our `KnownResult` instance will then mark `Help` and `Get` as procedures, and everything else as commands.

```
instance KnownResult PlistPrimitive where
  knownResult x =
     case x of
       Help          -> Nothing
       Get         -> Nothing
       Exit          -> Just ()
       Save          -> Just ()
       Revert        -> Just ()
       Clear       -> Just ()
       Set         -> Just ()
       Add         -> Just ()
       Delete      -> Just ()
       ImportData  -> Just ()
       MergeDate   -> Just ()
```

Since the PlistBuddy is an interactive shell, the “remote” transport mechanism is simply our Haskell program opening a pseudo-terminal to the plistbuddy process and sending commands the same way that a user would.

```
openPlist :: FilePath -> IO Plist
openPlist fileName = handleIOErrors $ do
    (pty,ph) <- spawnWithPty
                    Nothing
                    False
                    "/usr/libexec/PlistBuddy"
                    ["-x",fileName]
                    (80,24)
    return $ Plist pty ph

command :: Plist -> ByteString -> IO ByteString
command plist input = do
  writePty pty (input <> "\n")
  r <- recvReply pty
  return r

    where
      pty = plist_pty plist

send :: Plist -> PlistBuddy a -> IO a
send dev m = do
  res <- runExceptT $ unwrapNT (runMonad (wrapNT (sendW dev))) m
  case res of
    Left (PlistError e) -> throw $ PlistBuddyException e
    Right a             -> pure a
```

In the above code `sendW` is an implementation using the Weak Packet, but we could use any other packet to bundle together commands. To bundle with a different packet, we would simply build up the byte string for each primitive, separated with a newline character. Here is an example of a program that is running PlistBuddy:

```
main = do
  device <- openPlist "example/example.plist"
  t <- getCurrentTime
  send device $ do
    add ["EnvironmentName"] (String "Dev")
    add ["APIBaseURL"] (String "http://example.com")
    add ["ServiceErrorDomain"] (String "http://error.example.com")
    add ["RTCICEServerURLs"] (Array [])
    add ["RTCICEServerURLs","0"] (String "stun:stun.example.com")
    add ["RTCICEServerURLs","1"] (String "turn:turn.example.com")
    add ["WebSocketAPIVersion"] (Real 0.5)
    add ["allAccessAPIURL"] (Dict [])
    add ["allAccessAPIURL", "ChatEndpoint"] (String "http://chat.example.com")
    add ["allAccessAPIURL", "MappingEndpoint"] (String "http://example.com/map")
    add ["allAccessAPIURL", "ChatPort"] (Integer 3015)
    save
  return ()
```

This example would then yield the following file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
                       "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>APIBaseURL</key>
        <string>http://example.com</string>
        <key>Bool</key>
        <true/>
        <key>Date</key>
        <date>2017-12-19T02:01:54Z</date>
        <key>Dict</key>
        <dict/>
        <key>EnvironmentName</key>
        <string>Dev</string>
        <key>Integer</key>
        <integer>5</integer>
        <key>RTCICEServerURLs</key>
        <array>
                <string>stun:stun.example.com</string>
                <string>turn:turn.example.com</string>
        </array>
        <key>Real </key>
        <real>4.5</real>
        <key>ServiceErrorDomain</key>
        <string>http://error.example.com</string>
        <key>String</key>
        <string>tester</string>
        <key>WebSocketAPIVersion</key>
        <real>0.5</real>
        <key>allAccessAPIURL</key>
        <dict>
                <key>ChatEndpoint</key>
                <string>http://chat.example.com</string>
                <key>ChatPort</key>
                <integer>3015</integer>
                <key>MappingEndpoint</key>
                <string>http://example.com/map</string>
        </dict>
</dict>
</plist>
```

6.6 Case Study: Haxl

Up until this point, we have discussed how we can classify remote primitives as commands, procedures, and queries and how based on their classification, we can use different techniques to bundle them together to avoid the network cost associated with each call. But we have delayed discussing read-only queries and how that fits in with bundling requests. In this section, we will look at how systems that use read-only queries are able to bundle more aggressively due to the nature of the request. This technique was first brought forth by Simon Marlow and his group at Facebook in the form of a Haskell library called Haxl. The way that Haxl splits up a monad into batches of items can be viewed as a specific instance of the more general remote monad theory contained in this dissertation.

6.6.1 Supporting Haxl-style Primitives

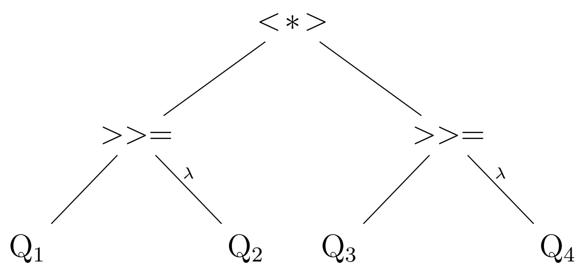
Haxl [Marlow et al., 2014] is a Haskell library developed and used by Facebook for high-performance access to remote services. The Haxl library was created at Facebook to classify actions and content as spam or as abusive with regard to Facebook's terms of service. The logic that determines if something is abusive is encapsulated in many different rules and needs to be run quickly in order to be useful for the scale of Facebook's environment. Haxl's main contribution is the fact that it can figure out which rules and tests can be run in parallel to make decisions about the action or content. Haxl contains other useful techniques to exploit concurrency where it is able, such as caching results, and can also dump cached results to a file to reproduce the environment surrounding bugs as they are encountered. In this section, we will only look at the parts of Haxl that are aimed at bundling remote requests and how these components compare and contrast with the remote-monad library as well as run through some examples. We will also look at creating a version of Haxl using the remote monad framework to show that Haxl is an instance of the remote monad ideas. There are two main differences between remote monads and Haxl.

- Remote monads split primitives into commands and procedures, while in Haxl, as originally presented, all primitives are read-only queries. These queries are cached by design in Haxl to always give the same answer to the same question, and like procedures, they give results.
- Remote monads are geared towards remote procedure calls that have side effects, and as such, commands and procedures are **always** executed in the given order. However, Haxl is in a domain where the primitives are effect-free, allowing for more concurrency through the reordering of primitives.

Supporting queries gives us the following network component:

$$\begin{array}{c}\left\llbracket \begin{array}{c} \mathit{Monad} \\ < \mathit{Queries} > \end{array}\right\rrbracket \\ \downarrow \\ \left\llbracket \begin{array}{c} \mathit{ApplicativeFunctor} \\ < \mathit{Queries} > \end{array}\right\rrbracket\end{array}$$

The ability to reorder means that Haxl can perform bundling in circumstances in which remote monads require multiple packets. Consider the following expression tree, where Q_n denotes a Haxl query, and also a remote monad style procedure:



- In our remote monad implementation, we will send 3 (applicative) packets: [Q_1], [Q_2 & Q_3] and [Q_4], where the packets are separated by the monadic binds.
- Haxl will send [Q_1 & Q_3], then [Q_2 & Q_4], combining the left-hand side of both binds into one packet and later sending the right-hand side.

Haxl determines where the data dependencies are between the different actions and sending the actions that have all the information needed to be evaluated. Haxl does this by doing a breadth-first search along the structure, finding actions that are not depending on any other unfinished work. The first action found after a `<*>`, as an example, would always be considered as ready to be sent as it never relies on other data by the very nature of the applicative structure.

We can add Haxl bundling to our framework by having a new bundling strategy that is based on the assumptions found in what we call the Haxl Lemma.

Haxl Lemma

If **all** primitives are queries, then they can be reordered to maximize applicative functor bundling. The key transformation can be expressed as

```
(q1 >>= k1) <*> (q2 >>= k2)
```

$$\equiv$$

```
(,) <$> q1 <*> q1 >>= \(r1,r2) -> k1 r1 <*> k2 r2
```

When using the `ApplicativeDo` GHC extension we actually have something very similar occur, from which both libraries are able to benefit. If we have the following do-notation:

```
do x <- fetchSomething
   y <- fetchSomethingElse
   z <- someComputation x y
  ...
```

The extension does its work at compile time, so it is able to determine that `x` is not used in the `fetchSomethingElse` operation. With that discovery, the extension can then rewrite the program to be the following:

```
(x,y) <- (,) <$> fetchSomething <*> fetchSomethingElse
z     <- someComputation x y
```

To support Haxl's style of rewrites and reordering, we can simply create a new more aggressive bundling strategy through the `runMonad` evaluation function which will apply new rules based on the new assumptions.

Since the Haxl methodology is still using applicative functors to be transmitted we can reuse our `ApplicativePacket` and just wrap a `newtype` around it to trigger the corresponding `runMonad` function.

```
newtype QApplicativePacket (q :: * -> *) (a :: *)
  = QApplicativePacket (ApplicativePacket q a)
```

We then provide an additional overloading for `runMonad`, parameterized over `QApplicativePacket`s, that utilizes the Haxl lemma to reorder primitives to maximize the bundling of queries. Since we can already send an entire applicative functor in a single packet, the reordering only happens when binds are involved. Looking back at our `RemoteMonad` constructors, we have the `Ap'` constructor which is used to tell us that we have an applicative functor operation but that there are binds found therein. What we are then able to do is pattern match on that constructor and then apply a rewrite rule to cause the bundling to happen.

Here is what one of the rewrite rules looks like:

```
helper :: RemoteMonad prim a -> RemoteMonad prim a
helper (Ap' (Bind m1 k1) (Bind m2 k2) ) =
   liftA2 (,) (helper m1) (helper m2) >>= (\ (x1,x2) ->
                                         helper (k1 x1) <*> helper (k2 x2))
```

This rule should look familiar as it is the rule demonstrated with the description of the Haxl lemma. The other rules are used to correctly handle the different variations of the data structures.

There have been at least 8 clones of Haxl made using Scala [clu, 2018; fet, 2018b], PureScript [fet, 2018a], Clojure [mus, 2018; ura, 2018], C# [hax, 2018], and another Haskell version but built using Free Applicative Functors [Fancher, 2016]. Since we have shown that `Haxl` is an instance of remote monad specialized over read-only queries, the existence of these libraries show that remote monad ideas can be performed in languages outside of Haskell as well as the usefulness of these concepts.

6.7 Observations

With these case studies, we have seen the remote monad parameterized over a wide array of APIs that have different needs and domains but have a common aim of interacting with a remote entity, making this a valid abstraction to be employed in many environments and situations. To use the `remote-monad` library one would just have to create a GADT that mirrors the remote calls, have some serialization and transmission functions and then, if the server is also being created, a packet handler. As we saw from the Haxl case study, there are a number of programs out there that are instances of the remote monad, with some using weak strategies or something stronger.

Chapter 7

Performance

As a demonstration of the performance difference between the bundling strategies, we will look at some benchmark tests of the `blank-canvas` library and compare the time for each of the bundling strategies. These benchmarks were originally created to test the performance and abilities of `blank-canvas` when the ideas of the remote monad were in their infancy. What this means is that most of the tests were created under the assumption that the strong bundling was the best that we could do. Since the ‘blank-canvas‘ library is heavily centered on commands, there were only a few of the test cases that needed an updated to let the applicative bundling really shine. Some comparisons of rewriting the test cases are found in Section 7.3.

7.1 Benchmarks

Before jumping into the results, here is the list of benchmarks run:

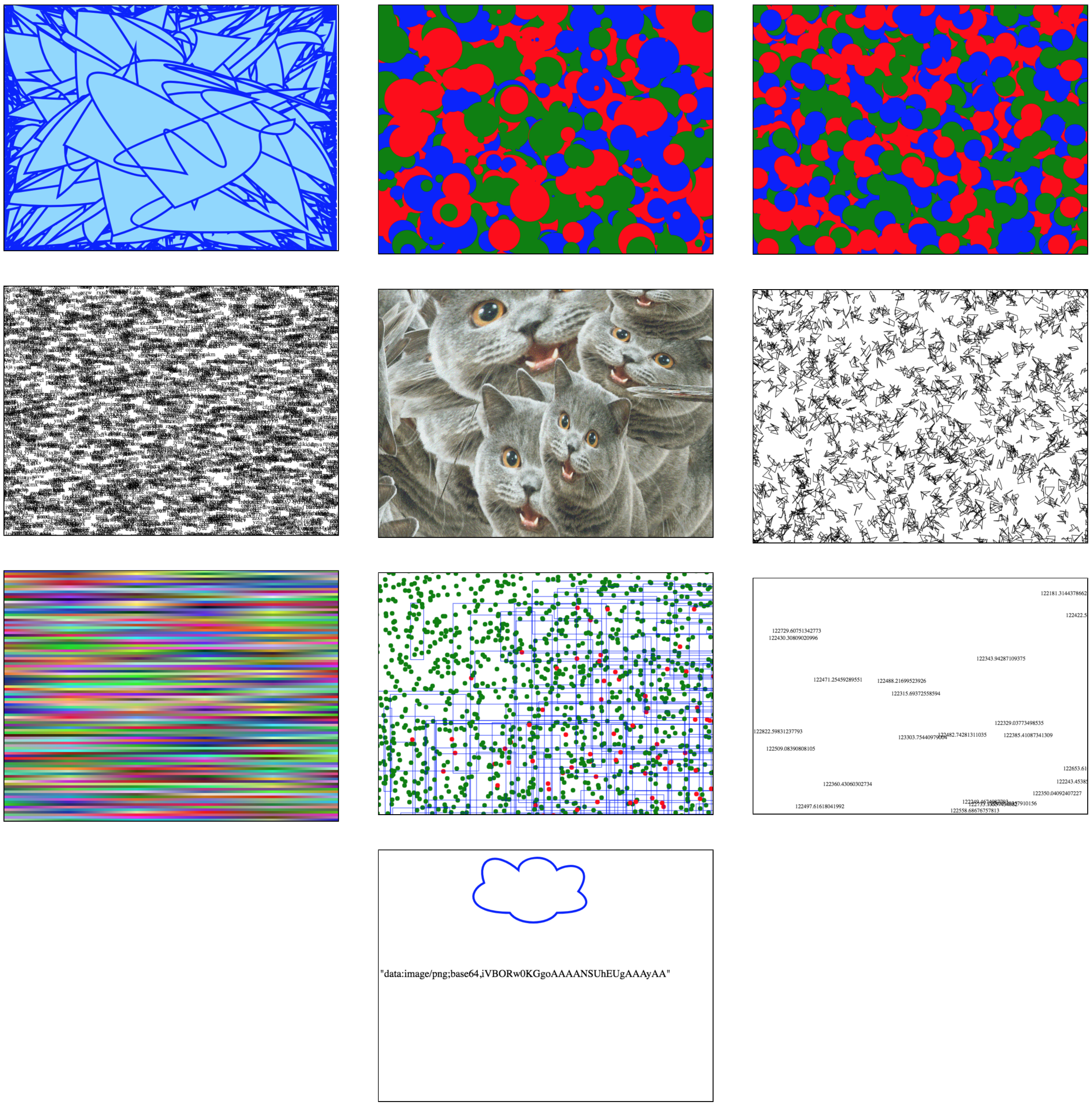
- Bezier - draw filled polygons comprised of 5 bezier curves 100 times
- CirclesRandomSize - draw circles of random sizes 1,000 times
- CirclesUniformSize - draw circles of the same size 1,000 times
- FillText - draw short words 1,000 times

- ImageMark - draw a randomly stretched and rotated image 1,000 times
- StaticAsteroids - draw asteroids (6 point polygons) 1,000 times
- Rave - draw the linear gradient of 6 different colors 100 times
- IsPointInPath - draw a random rectangle and plot 10 random points, if a point is found inside the rectangle, then the point is red or if not, then the point is green. This is done 100 times
- MeasureText - Measure the length of text with a given format and font. Each text has 2,000 10-letter words.
- ToDataURL - draw a cloud with bezier curves and then print part of the data URI, which contains the encoded representation of the image, 30 times.

Screenshots of these benchmarks are shown in Figure 7.1

Each of these benchmarks is run multiple times by the `criterion` Haskell library and run to obtain the linear regression. The main numbers in this chapter are taken from runs done in the Chrome web browser on a machine running Mac OSX with the canvas having a width and height of 800 and 600 pixels respectively. To view the different bundling strategies running on the Safari, Chrome and Firefox web browsers see Section 7.4.

The first seven tests are running `blank-canvas` operations that have no return result, or in other words, they are running remote commands. `IsPointInPath`, `MeasureText`, and `ToDataURL`, on the other hand, are making calls to the remote server and are expecting a result. What this means is that when we use weak-bundling for `blank-canvas` we should expect a performance hit globally when compared with using some other bundling strategy but that the performance difference between the strong bundling and the applicative bundling will not be apparent until the `IsPointInPath`, `MeasureText` and `ToDataURL` functions are being used.



```
122181.3144378662
122422.5
122729.60751342773
122430.30809020996
122343.94287109375
122471.25459289551
122488.21699523926
122315.69372558594
122329.03773498535
122822.59831237793
123303.75440979004
122482.74281311035
122385.41087341309
122509.08390808105
122653.61
122360.43060302734
122243.4538
122350.04092407227
122249.4674987793
122733.1[illegible]7910156
122497.61618041992
122558.68676757813
```

```
"data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAyAA"
```

Figure 7.1: ScreenShots of Benchmarks. From left to right: Bezier, CirclesRandomSize, CirclesUniformSize, FillText, ImageMark, StaticAsteroids, Rave, IsPointInPath, MeasureText, ToDataURL

Benchmark	Weak (ms)	Strong (ms)	Applicative (ms)
Bezier	113.7	71.4	80.0
CirclesRandomSize	138.5	52.2	59.6
CirclesUniformSize	134.9	48.5	55.6
FillText	150.4	75.6	87.4
ImageMark	184.7	70.2	76.0
StaticAsteroids	374.3	112.4	128.2
Rave	48.8	20.9	26.0
IsPointInPath	447.8	359.1	199.3
MeasureText	682.9	689.2	142.8
ToDataURL	211.1	208.2	238.9

Table 7.1: Performance Comparison of Bundling Strategies (Chrome v64.0.3282.186)

Something that should be taken into consideration when looking at these numbers is that the benchmarks aren't normalized with each other, meaning that some benchmarks could be doing more work or less work than the others. What we want to focus on is the difference between weak, strong and applicative bundling when the same test case is run. All of the numbers are taken with the Haskell code and the browser residing on the same machine to minimize the skewing of data because of ever-changing network conditions. The differences between the different bundling strategies would then be magnified if we added network delay for each packet sent.

Table 7.1 shows a comparison between the execution time of the different bundling strategies. As expected we can see the weak bundling is anywhere between 2-6 times slower than the strong and applicative bundling strategies. Let's look at what is inside the packets as the speedup directly correlates with the number of packets sent.

Something that is a little unexpected in looking at the performance numbers is that there are times when the strong bundling is slightly faster than the applicative bundling, even though they are sending the same number of packets. It turns out that this is the only time when the applicative bundling strategy is slower than the strong bundling. Because the applicative bundling is more complex than the strong bundling, the extra work needed to build up a packet adds a small amount of overhead. This overhead is only noticeable, when the two strategies are sending the same number packets. To be specific, because of

Total Packets	Commands per Packet	Procedures per Packet
50616	0	1
1027172	1	0

Table 7.2: Weak Packet profile during benchmark tests

Total packets	Commands per Packet	Procedures per Packet
46456	0	1
16	2	1
1632	3	1
16	14	1
464	17	1
1600	41	1
92	1200	1
79	1300	1
56	3000	1
184	5000	1
56	6000	1
29	9992	1

Table 7.3: Strong Packet profile during benchmark tests

the structure of applicative functors, extra work is needed to apply the results of the remote actions to the local combining function when compared with the other bundling strategies.

Overall, the extra cost of creating applicative packets is minimal when compared with the magnitude of the performance gains that we can see when the applicative bundling is able to lower then number of packets, most notably seen with the `MeasureText` and `IsPointInPath` benchmarks.

There may be some situations in which bundling requests does not give the best performance when looking at the ratio between the computation cost and the network latency. We observe that when network costs dominate remote procedure costs, batching immediately dispatchable procedures will almost always be better than the weak remote monad. We leave the investigation of when the weak remote monad, or similar, performs better than the bundling strategies, as future work.

Total Number of Packets	Commands per Packet	Procedures per Packet
68	0	1
22	2	2000
38	3	1
1600	3	10
16	14	1
464	17	1
1600	41	1
56	1200	1
79	1300	1
46	3000	1
159	5000	1
46	6000	1
29	9992	1

Table 7.4: Applicative Packet profile during benchmark tests

7.2 Packet Shapes

Running blank-canvas' benchmarks tests with the packet profiling flag allows us to see the distribution of the packets for these test sets. If we look at the distribution of the packets in the weak bundling, found in Table 7.2, we can see that after running all the benchmarks we sent 50,616 packets that had a single procedure and 1,027,172 packets that contained a single command, yielding 1,077,788 total packets. Compare this to the strong bundling packet distribution in Table 7.3 which only sent a total of 52,824 packets and the applicative bundling profile in Table 7.4 which only sent 4,223 packets. It is no wonder that we have a speedup of 2-6 times faster when we use the strong or applicative bundling strategies than if we used the weak bundling strategy. These performance numbers are looking at an API that is heavily leaning towards commands over procedures. These results could vary depending on the API primitives as well as how the data is used. Recently, `blank-canvas` was updated from making HTTP requests to use WebSockets as the communication layer between the Haskell program and the web browser. Because the HTTP requests are slower when compared with WebSockets the difference between the weak and the other bundling strategies much more pronounced with a speed up of 6-8 times faster when using the strong or applicative bundling. Utilizing this faster transmission mechanism narrowed the gap

Weak			**Strong**			**Applicative**		
# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet
1	0	1	1	1300	1	1	1300	1
1300	1	0						

Table 7.5: Bezier Packet Profile during benchmark testing

between the different strategies.

7.2.1 Bezier

```
benchmark :: CanvasBenchmark
benchmark ctx = do
    bzs <- replicateM numBezier $ replicateM numCurves $ (,,,,,)
                    <$> randomXCoord ctx
                    <*> randomYCoord ctx
                    <*> randomXCoord ctx
                    <*> randomYCoord ctx
                    <*> randomXCoord ctx
                    <*> randomYCoord ctx
    send ctx $ forM_ bzs drawCurves

drawCurves :: [Bezier] -> Canvas ()
drawCurves bzs = do
    beginPath();
    let (_, _, _, _, x, y) = last bzs
    moveTo(x, y);
    forM_ bzs bezierCurveTo

    closePath();
    lineWidth(5);
    fillStyle("#8ED6FF");
    fill();
    strokeStyle("blue");
    stroke();
```

Weak			**Strong**			**Applicative**		
# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet
1	0	1	1	5000	1	1	5000	1
5000	1	0						

Table 7.6: CirclesRandomSize Packet profile from a single run of the test

7.2.2 CirclesRandomSize

```
benchmark :: CanvasBenchmark
benchmark ctx = do
    xs <- replicateM 1000 $ randomXCoord ctx
    ys <- replicateM 1000 $ randomYCoord ctx
    rs <- replicateM 1000 $ randomRIO (1, 50)

    send ctx $ sequence_ [ showBall (x, y) r col
                         | x <- xs
                         | y <- ys
                         | r <- rs
                         | col <- cycle ["red","blue","green"]
                         ]

showBall :: Point -> Double -> Text -> Canvas ()
showBall (x, y) r col = do
    beginPath();
    fillStyle(col);
    arc(x, y, r, 0, pi*2, False);
    closePath();
    fill();
```

Weak			Strong			Applicative		
# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet
1	0	1	1	5000	1	1	5000	1
5000	1	0						

Table 7.7: CirclesUniformSize Packet profile from a single run of the test

7.2.3 CirclesUniformSize

```
benchmark :: CanvasBenchmark
benchmark ctx = do
    xs <- replicateM 1000 $ randomXCoord ctx
    ys <- replicateM 1000 $ randomYCoord ctx
    send ctx $ sequence_ [ showBall (x, y) col
                         | x <- xs
                         | y <- ys
                         | col <- cycle ["red","blue","green"]
                         ]

showBall :: Point -> Text -> Canvas ()
showBall (x, y) col = do
    beginPath();
    fillStyle(col);
    arc(x, y, 25, 0, pi*2, False);
    closePath();
    fill();
```

Weak			Strong			Applicative		
# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet
1	0	1	1	3000	1	1	3000	1
3000	1	0						

Table 7.8: FillText Packet profile from a single run of the test

7.2.4 FillText

```
benchmark :: CanvasBenchmark
benchmark ctx = do
    xs <- replicateM 1000 $ randomXCoord ctx
    ys <- replicateM 1000 $ randomYCoord ctx
    ws <- cycle <$> replicateM 1000 randomWord
    send ctx $ sequence_ [ showText (x, y) word
                         | x <- xs
                         | y <- ys
                         | word <- ws
                         ]

showText :: Point -> Text -> Canvas ()
showText (x, y) txt = do
    fillStyle("black");
    font("10pt Calibri");
    fillText(txt, x, y);
```

Weak			Strong			Applicative		
# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet
2	0	1	1	0	1	1	0	1
6000	1	0	1	6000	1	1	6000	1

Table 7.9: ImageMark Packet profile from a single run of the test

7.2.5 ImageMark

```
benchmark :: CanvasBenchmark
benchmark ctx = do
    xs     <- replicateM 1000 $ randomXCoord ctx
    ys     <- replicateM 1000 $ randomYCoord ctx
    ws     <- replicateM 1000 $ randomXCoord ctx
    hs     <- replicateM 1000 $ randomYCoord ctx
    thetas <- replicateM 1000 $ randomRIO (0, 2*pi)
    send ctx $ do
       img <- newImage image
       sequence_ [ drawTheImage (x,y,w,h) theta img
                 | x     <- xs
                 | y     <- ys
                 | w     <- ws
                 | h     <- hs
                 | theta <- thetas
                 ]
drawTheImage :: Path -> Angle -> CanvasImage -> Canvas ()
drawTheImage (x,y,w,h) theta img = do
    beginPath();
    save();
    rotate(theta);
    drawImageSize(img, x - (w/2), y - (w/2), w, h);
    closePath();
    restore();
```

Weak			Strong			Applicative		
# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet
1	0	1	1	9992	1	1	9992	1
9992	1	0						

Table 7.10: StaticAsteroids Packet profile from a single run of the test

7.2.6 StaticAsteroids

```
benchmark :: CanvasBenchmark
benchmark ctx = do
  xs  <- replicateM 1000 $ randomXCoord ctx
  ys  <- replicateM 1000 $ randomYCoord ctx
  dxs <- replicateM 1000 $ randomRIO (-15, 15)
  dys <- replicateM 1000 $ randomRIO (-15, 15)
  send ctx $ do
             clearCanvas
             sequence_ [showAsteroid (x,y) (mkPts (x,y) ds)
                       | x <- xs
                       | y <- ys
                       | ds <- cycle $ splitEvery 6 $ zip dxs dys
                       ]
showAsteroid :: Point -> [Point] -> Canvas ()
showAsteroid (x,y) pts = do
  beginPath()
  moveTo (x,y)
  mapM_ lineTo pts
  closePath()
  stroke()
```

Weak			Strong			Applicative		
# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet
1	0	1	1	1200	1	1	1200	1
1200	1	0						

Table 7.11: Rave Packet profile from a single run of the test

7.2.7 Rave

```
benchmark :: CanvasBenchmark
benchmark ctx = do
    let w = width ctx
        h = height ctx
        dy = h / fromIntegral numGradients
        ys = [0, dy .. h - dy]
    rgbsList <- replicateM numGradients . replicateM numColors $
        S.rgb <$> randomIO <*> randomIO <*> randomIO
    send ctx $ sequence_ [ drawGradient (0, y, w, dy) rgbs
                         | y <- ys
                         | rgbs <- rgbsList
                         ]
drawGradient :: CanvasColor c => Path -> [c] -> Canvas ()
drawGradient (gx0, gy0, gx1, gy1) cs = do
    beginPath();
    rect(gx0, gy0, gx1, gy1);
    grd <- createLinearGradient(gx0, gy0, gx1, gy0+gy1);
    let cMaxIndex = genericLength cs - 1
    forWithKey_ cs $ ı c ->
        grd # S.addColorStop (fromIntegral i/cMaxIndex, c);
    S.fillStyle(grd);
    fill();
    closePath();
```

Weak			Strong			Applicative		
# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet
1100	0	1	900	0	1	100	3	10
4400	1	0	100	3	1	100	41	1
			100	41	1			

Table 7.12: IsPointInPath Packet profile from a single run of the test

7.2.8 IsPointInPath

```
benchmark :: CanvasBenchmark
benchmark ctx = sequence_ [ internal ctx | _ <- [1..rounds]]

internal :: CanvasBenchmark
internal ctx = do
    pathX1 <- randomXCoord ctx
    pathX2 <- randomXCoord ctx
    pathY1 <- randomYCoord ctx
    pathY2 <- randomYCoord ctx
    points <- replicateM pointsPerPath $ (,) <$> randomXCoord ctx <*> randomYCoord ctx
    send ctx $ sequence_ [ isInPath (pathX1, pathX2, pathY1, pathY2) points ]

isInPath :: Path -> [Point] -> Canvas ()
isInPath (pathX, pathY, pathW, pathH) points = do
    strokeStyle("blue");
    beginPath();
    rect(pathX, pathY, pathW, pathH);
    cmds <- sequence [ do
                          b <- isPointInPath(x, y);
                          return $ do
                              beginPath();
                              fillStyle(if b then "red" else "green");
                              arc(x, y, pointRadius, 0, pi*2, False);
                              fill();
                     | (x, y) <- points
                     ]
    stroke();
    sequence_ cmds
```

Weak			Strong			Applicative		
# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet
2002	0	1	2000	0	1	1	0	1
5	1	0	1	2	1	1	2	2000
			1	3	1	1	3	1

Table 7.13: MeasureText Packet profile from a single run of the test

7.2.9 MeasureText

```
benchmark :: CanvasBenchmark
benchmark ctx = do
    ws <- replicateM 2000 randomWord
    wds <- send ctx $ do
        fillStyle("black")
        font("10pt Calibri")
        forM ws measureText
    x <- randomXCoord ctx
    y <- randomYCoord ctx

    send ctx $ do
       fillStyle("black");
       font("10pt Calibri");
       fillText(T.pack $ show $ sum [ v | TextMetrics v <- wds ], x, y);
    return ()
```

Weak			Strong			Applicative		
# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet
31	0	1	1	3	1	1	3	1
510	1	0	1	14	1	1	14	1
			29	17	1	29	17	1

Table 7.14: ToDataUrl Packet profile from a single run of the test

7.2.10 ToDataURL

```
benchmark :: CanvasBenchmark
benchmark ctx = do
      rs <- replicateM 30 $ randomRIO (0,100)
      send ctx $  sequence_ [picture r | r <- rs ]

picture :: Double -> Canvas ()
picture  x = do
        clearCanvas
        beginPath();
        moveTo(170 + x, 80);
        bezierCurveTo(130 + x, 100, 130 + x, 150, 230 + x, 150);
        bezierCurveTo(250 + x, 180, 320 + x, 180, 340 + x, 150);
        bezierCurveTo(420 + x, 150, 420 + x, 120, 390 + x, 100);
        bezierCurveTo(430 + x, 40, 370 + x, 30, 340 + x, 50);
        bezierCurveTo(320 + x, 5, 250 + x, 20, 250 + x, 50);
        bezierCurveTo(200 + x, 5, 150 + x, 20, 170 + x, 80);
        closePath();
        lineWidth 5;
        strokeStyle "blue";
        stroke();
        cloud <- toDataURL();
        fillStyle("black");
        font "18pt Calibri"
        fillText(T.pack $ show $ T.take 50 $ cloud, 10, 300)
```

7.3 Applicative combination functions

In this dissertation, we see that monadic binds force us to terminate and send our currently batched set of primitives. One of the techniques that we talked about in an earlier section is updating our code to use more applicative functor combinators instead of the monadic binds. In the past, this could be done by exchanging the more commonly used monadic functions such as `sequence` or `forM` with their applicative counterparts, `sequenceA` and `for` respectively. These applicative counterparts achieved the same result but went about it using applicative combinators instead of monadic binds. As was mentioned in Section 5.3.2, these monadic mappings over lists have now been rewritten to use the applicative versions. What this means is that for our examples, using `sequenceA` vs `sequence` or `forM` vs `for` will have no difference in our packet composition, therefore, no speedup is gained.

What we can do is either rewrite computations by hand or turn on the `ApplicativeDo` compiler flag, and if there are any binds that are not required, then we should be able to get rid of them. We must remember that in many cases the binds are necessary and cannot be eliminated. Figure 7.2 shows the timing comparisons between `sequence` and `sequenceA` with the `ApplicativeDo` flag both on and off for each. Table 7.15 further verifies that `sequence` and `sequenceA` are equivalent as they have identical packet contents.

For this comparison, we will only be looking at the test cases which are looking at the `IsPointInPath`, `ToDataUrl` and `MeasureText` functions as they rely more heavily on returning results.

What we can see from these comparisons is that turning on the `ApplicativeDo` flag results in a speedup of almost 2 times what our original code could perform. In the cases of `MeasureText` and `ToDataURL`, however, we do not see a noticeable change as the standard deviations between the different cases overlap. This is also confirmed by the fact that these cases have identical packet contents.

In the `IsPointInPath` test case, we see that 900 procedures that were being sent by themselves were able to be bundled together once the `ApplicativeDo` flag was applied.

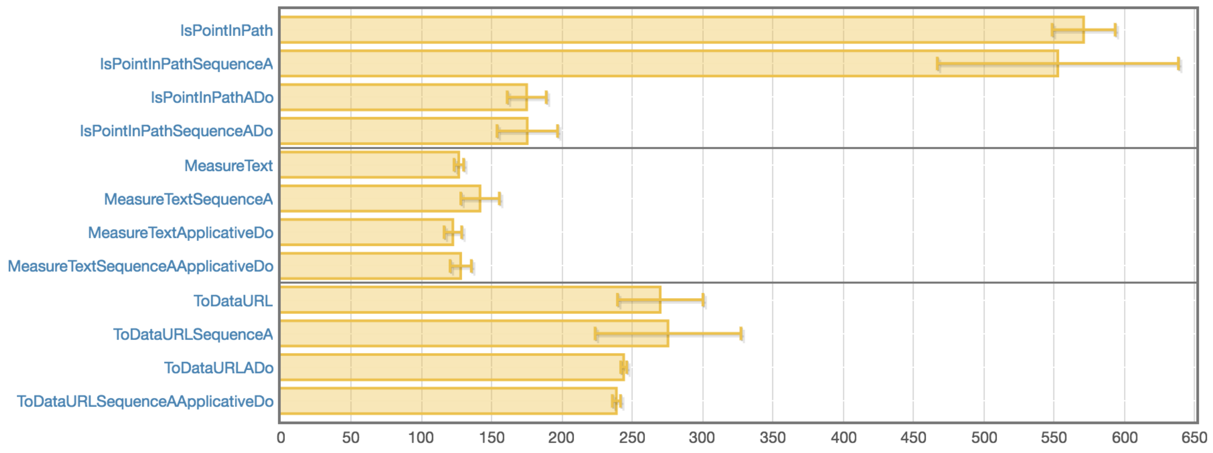


Figure 7.2: Effect of using SequenceA, Sequence and ApplicativeDo with Applicative Bundling

Benchmark	Sequence/SequenceA			ApplicativeDo		
	# Packets	Commands per packet	Procedures per packet	# Packets	Commands per packet	Procedures per packet
IsPointInPath	900	0	1	100	3	10
	100	3	1	100	41	1
	100	41	1			
ToDataURL	1	3	1	1	3	1
	1	14	1	1	14	1
	29	17	1	29	17	1
MeasureText	1	0	1	1	0	1
	1	2	2000	1	2	2000
	1	3	1	1	3	1

Table 7.15: Effects of ApplicativeDo extension on the number of primitives found in packet

Benchmark	Weak (ms)	Strong (ms)	Applicative (ms)
Bezier	85.5	53.9	54.6
CirclesRandomSize	166.4	58.0	59.7
CirclesUniformSize	169.8	55.7	55.0
FillText	187.6	46.3	48.2
ImageMark	288.9	70.6	75.6
StaticAsteroids	416.5	128.4	130.8
Rave	68.2	23.1	22.9
IsPointInPath	1744.1	835.3.0	834.0
MeasureText	655.3	659.6	61.0
ToDataURL	507.1	315.1	294.1

Table 7.16: Performance Comparison of Bundling Strategies in Safari v11.1

It is possible that the `ToDataUrl` and `MeasureText` test cases could be rearranged to get an additional speedup with the applicative bundling but that the techniques used by the `ApplicativeDo` flag were unable to improve upon it.

7.4 Other Systems

We can see that some of the ways that different browsers handle the JavaScript coming from `blank-canvas` have very different timings when combined with the bundling strategies as well. The Safari browser results found in Table 7.16 and Figure 7.4, show that we have an average speedup of 2-4 times when comparing the strong or applicative bundling strategies with the weak strategy and a 10 times increase in speed when looking at the `MeasureText` test. The test that takes the longest for Safari is the `IsPointInPath` test, by a significant margin. Whereas for Firefox measurements in Table 7.17 and Figure 7.5, `ToDataURL` does not seem to change across the bundling strategies and `MeasureText` is the longest test in the weak with the applicative bundling running 14 times faster.

7.5 Observations

Overall, the performance numbers support our theory that the applicative bundling stands head and shoulders above the weak in all cases, and above the strong when applicative

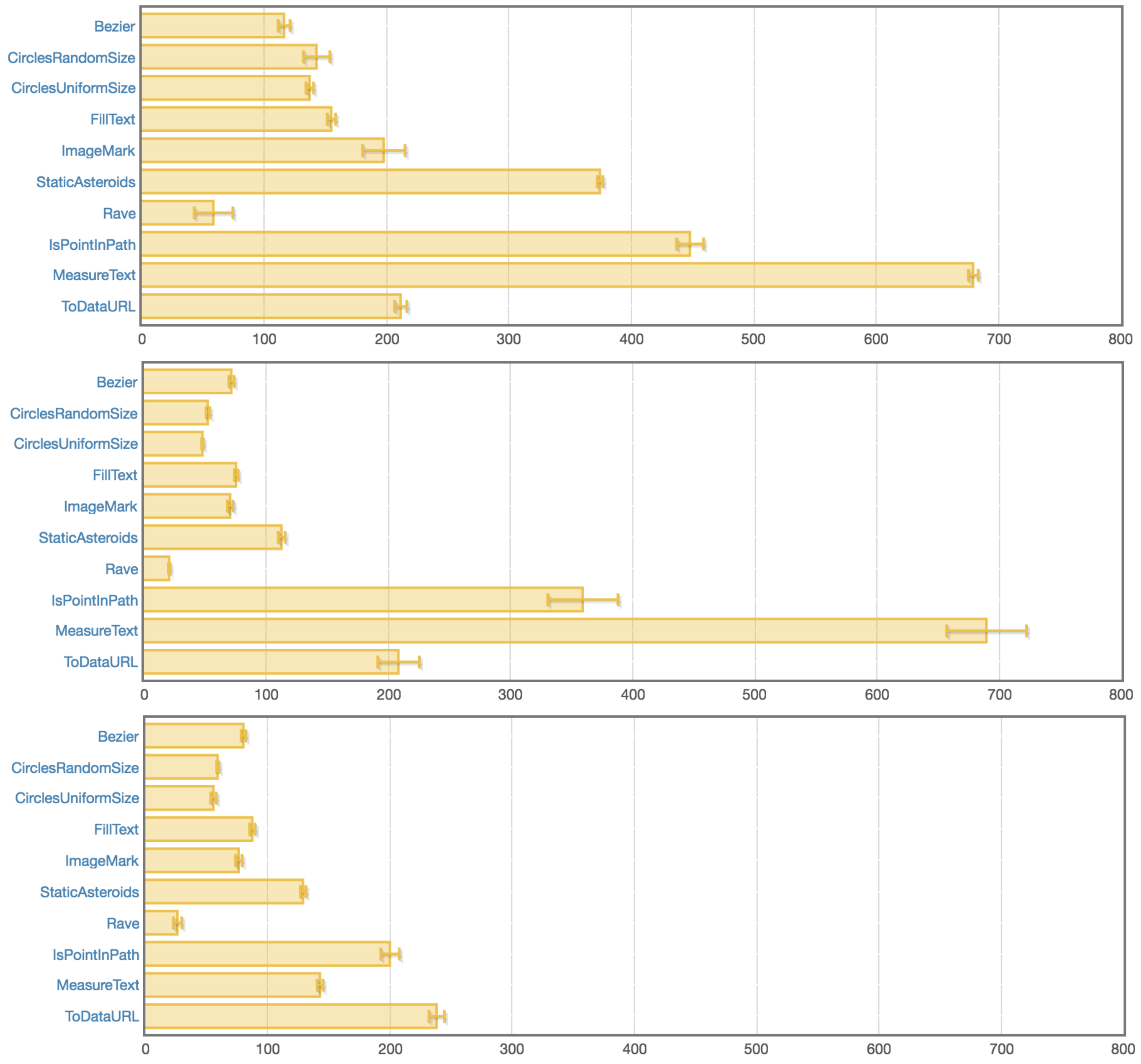


Figure 7.3: Performance Comparison of Bundling Strategies in Chrome v64.0.3282.186. From top to bottom: weak, strong and applicative bundling strategies

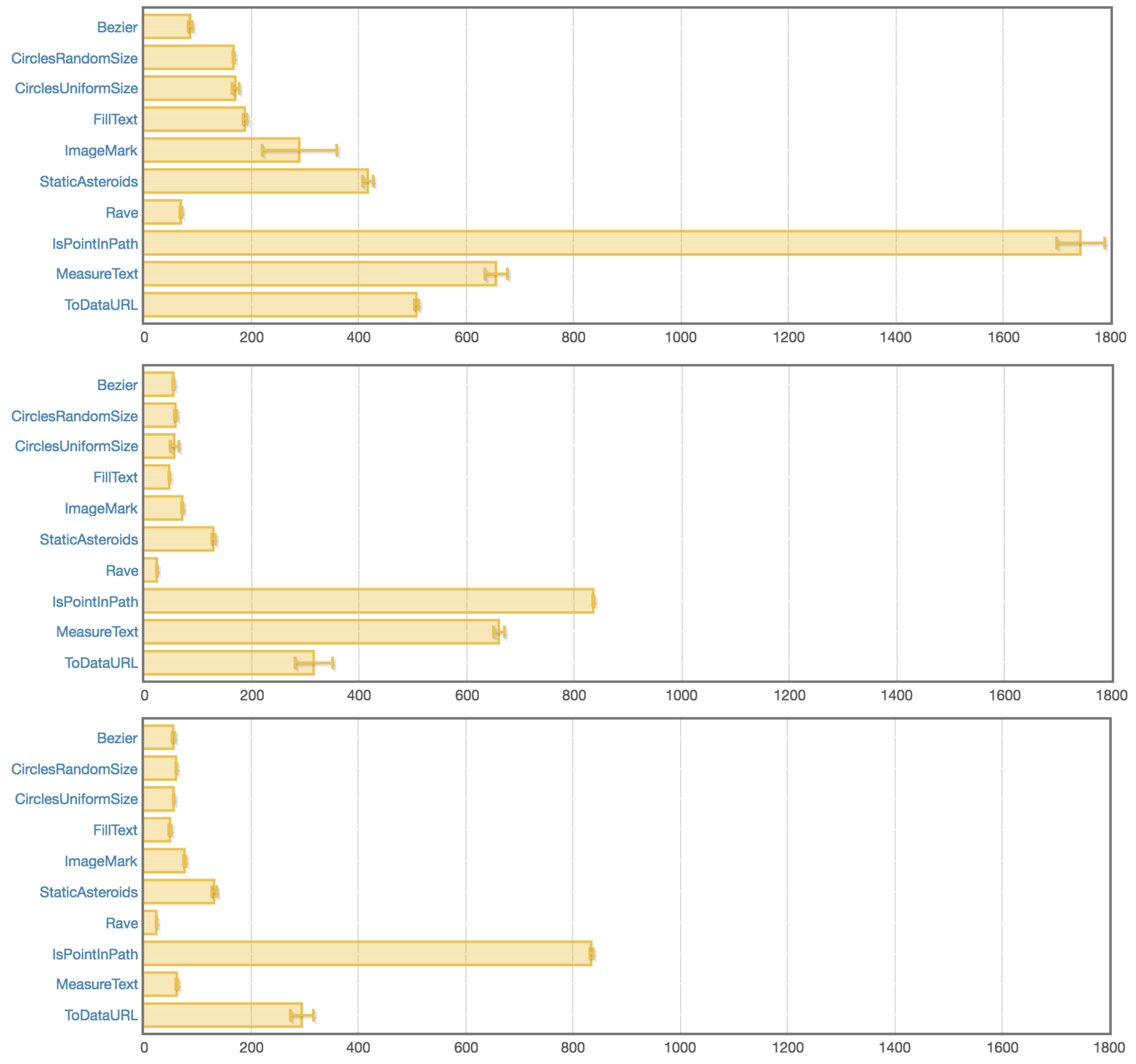


Figure 7.4: Performance Comparison of Bundling Strategies in Safari v11.1. From top to bottom: weak, strong and applicative bundling strategies

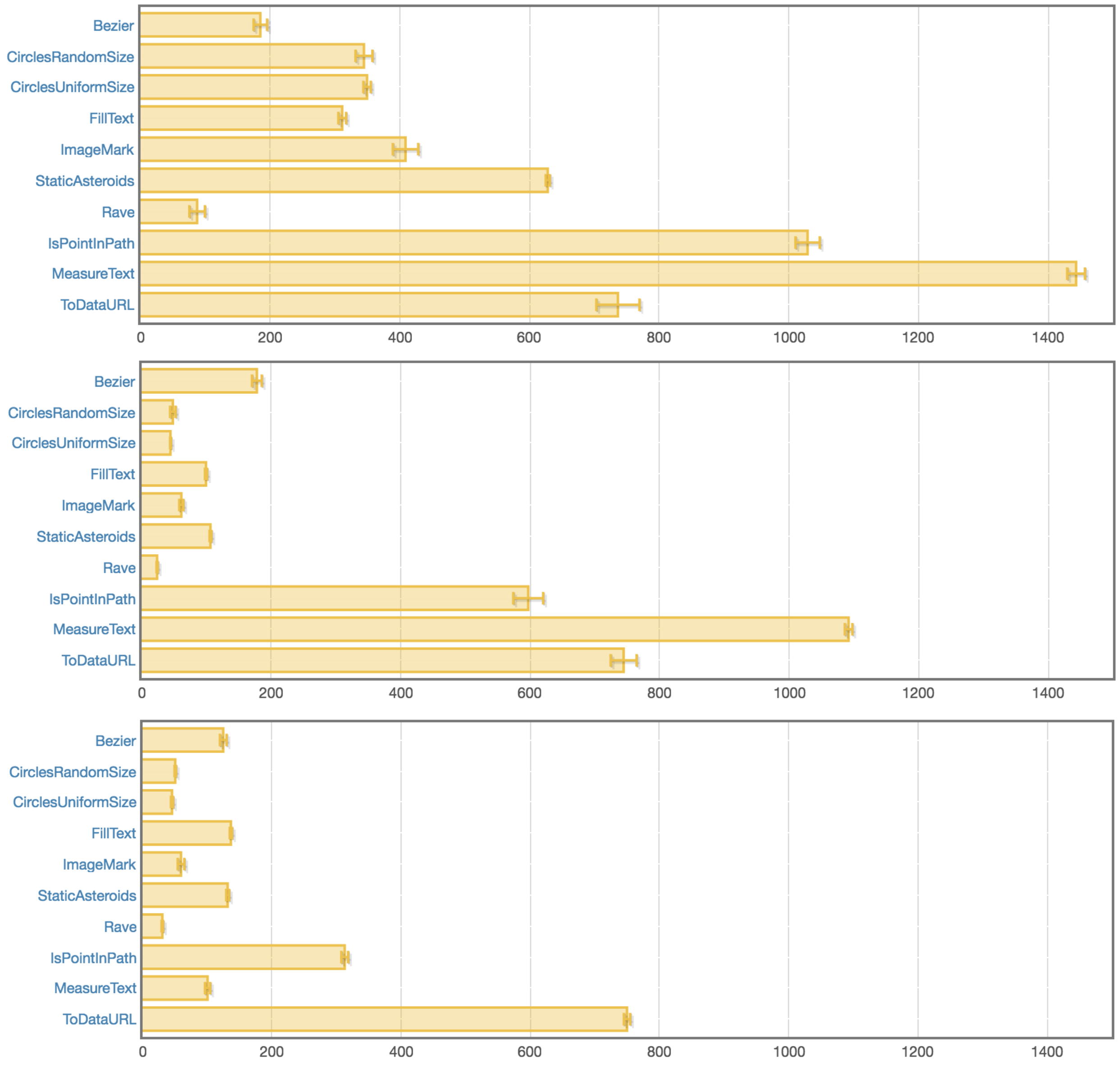


Figure 7.5: Performance Comparison of Bundling Strategies in Firefox v59.0.2. From top to bottom: weak, strong and applicative bundling strategies

Benchmark	Weak (ms)	Strong (ms)	Applicative (ms)
Bezier	184.0	117.9	124.8
CirclesRandomSize	344.3	48.1	50.5
CirclesUniformSize	349.4	44.5	45.5
FillText	311.8	99.3	136.0
ImageMark	408.6	61.2	59.5
StaticAsteroids	628.7	106.2	131.6
Rave	87.2	23.9	30.5
IsPointInPath	1028.9	596.9	312.0
MeasureText	1443.5	1091.6	100.6
ToDataURL	736.1	744.8	749.6

Table 7.17: Performance Comparison of Bundling Strategies in Firefox v59.0.2

structures are able to be used with procedures. It appears that if we were forced to do binds after every procedure that doing the strong bundling would be faster than applicative, with it being a simpler packet structure, but the small overhead that is needed to build up the applicative packet is definitely not a downside enough to choose a different bundling over the applicative bundling in general.

Chapter 8

Related Work

8.1 Remote Procedure Calls in Haskell

Haxl [Marlow et al., 2014] is a Haskell framework developed by Facebook that was created in order to batch requests to a database on a remote server. The server would receive the requests and then run them concurrently. This concurrency was made possible because the requests were all queries, thus the state of the database would remain unchanged without respect to the order of the requests. A more in-depth look at Haxl is found in Section 6.6.

Jeremy Gibbons [Gibbons, 2016] recently discussed using the Free Applicative Functor [Capriotti & Kaposi, 2014] in Haskell to batch requests to a remote server. In his paper, he confirmed our findings that the applicative functor construct is better suited for the transmission and execution of remote actions than the monad. His paper was based off our original Remote Monad paper where he simplified the mechanisms of batching requests by allowing the `Free` monad and `Free` applicative functor to do much of the work instead of creating each of the constructors from scratch.

Cloud Haskell [Epstein et al., 2011] is an implementation of Erlang-style concurrent and distributed programming in Haskell. This solution uses the GHC `static` keyword in conjunction with Template Haskell in order to send explicit Erlang-style messages to distributed processes on the network. There are a few transport mechanisms that have been created for Cloud Haskell including TCP, CCI, and an in-memory implementation. Cloud Haskell allows a free-form remote call, whereas with the remote monad, the user explicitly and systematically creates the connections from the client to the server.

The remote monad can be defined as a Domain Specific Language (DSL) for the communication between client and server. There are both shallow embedded DSLs [Hudak, 1998; Leijen & Meijer, 1999] where the execution is handled immediately and deeply embedded DSLs [Elliott et al., 2003; Gill, 2014] where the computation is built up as a structure which then can be passed to an evaluator function. Since we have an evaluator running on a server, anything using the remote-monad framework would be considered a deeply embedded DSL. Van Deuresen et al. show the widespread use of DSLs by compiling an annotated bibliography of 75 publications in the area of DSLs [Van Deursen et al., 2000].

8.2 RPC in other languages

In Java, there has been work [Pitt & McNiff, 2001; Maassen et al., 2001] done on performing remote procedure calls in an object-oriented way. These remote method invocations (RMI) require both the client and the server to be running Java programs. The server sends references to its objects to the client and the client is then able to call the object's methods.

Achten and Plasmeijer discuss how remote procedure calls can be leveraged to change Clean's interleaved Event IO system into a concurrent system, inside clean [Achten & Plasmeijer, 1995]. Van Weelden and Plasmeijer created a mechanism using techniques, similar to what was used for Cloud Haskell, that would allow the serialization of functions in Clean [Van Weelden & Plasmeijer, 2002].

Miller, Haller, and Odersky created a closure-like abstraction called `spores` in Scala [Miller et al., 2014]. They showed how most closures in Scala could be converted into a `spore` which provided more type-checking of the captured variables. This caused spores to have more guarantees when used in concurrent or distributed settings. Haller and Miller followed up that work with another paper [Haller & Miller, 2016] describing an asynchronous distributed programming model which could safely pass closures by adding restrictions which would avoid many common errors.

The research found in [Birrell & Nelson, 1984] showed that RPCs have a clean and simple semantics and that procedure calls are simple enough to be used effectively in communication. They also acknowledged the fact that if the RPC's communication cost was too high, "applications that might otherwise get developed would be distorted by [the application designer's] desire to avoid communicating." In handling remote exceptions Birrell and Nelson described a process similar to the way that our Remote Binary package handles exceptions, where communication from a server is either a result of some computation or it is a serialized exception that gets recreated on the local machine.

In [Shakib & Benson, 2001], the authors found that "calls that do not require an immediate response are delayed until a call needing a response is generated, batching multiple calls together." They also found that servers can be more efficient with batched requests, particularly when performing computations on the same object since the server can fetch the object into memory once and then perform multiple operations instead of possibly fetching the object each time a call is made.

[Bogle & Liskov, 1994] wrote his Master's thesis about using futures to minimize the overhead of expensive cross-domain calls. At the time of Bogle's thesis, futures were used as a mechanism to allow for parallel computing, so that a thread or process could know that a value is being computed. Bogle used futures as a way of deferring computations and later bundling them together. In his system, there were two types of results, handles to shared objects in the database and built-in values such as integers, characters, booleans, etc. The cross-domain calls that would result in handles, would be deferred until a call that resulted in a built-in value would be reached. The way that Bogle handled exceptions was by having the server keep track of any exceptions and have a call that the client could query for any exceptions that occurred since the last check. These checks would cause any deferred futures to be sent and have the drawback of not knowing about any errors until making the request.

Chapter 9

Future Work and Conclusion

9.1 Future Work

In exploring the remote monad there were a couple of other avenues of research that presented themselves but that ultimately, fell outside of the bounds of this dissertation.

9.1.1 Monad Transformer Stack

In Section 5.4, we discussed having the remote monad as a monad transformer. The research showed that it was possible to treat the remote monad as a monad transformer but that it would be unknown how multiple remote monads in a transformer stack would interact with each other. Abandoning this part of the research might not be the correct choice based solely on the strange errors that could occur. Having multiple remote monads in the stack could be a solution to the problem of interacting with multiple remote entities instead of having a conglomerate GADT, we could have a separate remote monad for each destination.

As a related note, in working with `blank-canvas` we originally had a monad transformer stack built with the `RemoteMonad` as its base. The stack consisted of a `Reader` monad to store the configuration options and a `State` monad which was used to keep track of the used variable names for data that needed to be stored in variables on the remote server. The weak and strong bundling strategies work as expected in this setup, but because of the extra binds that are introduced by this monad stack in the examples we tried, the applicative packets used in the applicative bundling were no different from the strong bundling where only one procedure is transmitted per packet instead of the ability to put multiple procedures in a packet. This was happening because of the use of the State monad to create valid variable names during the packet creation phase. Each time a procedure was needing to be sent, we would query the state for the next available variable name and bind that result to be used later. We were able to get around this by making a deep embedding of the State and Reader monad which is evaluated after the packet preparation is complete.

In the monad transformer stack, the binds used to get the configuration or the state would bubble through to the remote monad binds. Something that would be interesting to look at would be what local binds should look like vs remote binds, or in other words, binds that involve remote primitives vs binds that do not require a request outside the local runtime. The subtlety that comes from this is if we have the following:

```
send $ do
   remoteCommand1
   remoteCommand2
   localCommand
   remoteCommand3
   remoteProcedure1
```

If we are using the weak bundling, then things are simple, as you come across a remote primitive or a local primitive you just perform the work. But when we are bundling items, we do not have a semantics about when we should perform the local work. If we were using the strong or applicative bundling strategies we could bundle each of the remote primitives together. But in that case, do you perform the local work when you come across it? or later when the batch is sent to the remote server? How would this affect the behavior of the program? By using the remote monad as the base of a monad transformer stack, and adding our `IOAction` construct, we run into similar questions about when the local `IO` should be performed. It would be interesting to see how the various options affect a program and what the semantics would look like.

9.1.2 Bundle Size

In this dissertation, we did not address how to handle infinite structures or situations where we do not reach the point of actually sending the requests but instead continue to bundle the requests. In order to alleviate the possibility of never making calls to the remote resource and running into memory errors, we can imagine having an options sent with our `runMonad` function in which we could specify the maximum number of requests to bundle. When the maximum number of requests is reached, we can then pass our queued requests to the transmission function and then continue processing the requests.

9.1.3 Remote Static

In the beginning stages of looking at the remote monad and remote applicative functors we thought that there might be some promise in using static pointers [Peyton Jones, 2014] to serialize functions and closures to execute in a remote location.

Currently a downside to using the Remote Monad is the need to split up the program into commands and procedures and by using GADTs. Serializing these functions and function applications by using the `static` keyword would cause the remote monad library to stand closer to the Cloud Haskell library, which creates static pointers to data for distributed programming.

Below is an example of how the `static` keyword is used. The first half is only using the `GHC.StaticPtr` package where the second half is using a library that was developed for Cloud Haskell [Epstein et al., 2011] called `distributed-closure`.

```
fib :: Int -> Int
fib n
  | n > 1 = (fib (n-1)) + (fib (n-2))
  | n <= 1 && n >= 0 = n

fibPtr :: StaticPtr (Int -> Int)
fibPtr = static fib

main :: IO ()
main = do
-- client sends static ptr to remote server
  let bs = encode $ staticKey fibPtr

-- server dereferences function and applies it to int
  f <- unsafeLookupStaticPtr (decode bs)
  case f of
    Just func -> print $  (deRefStaticPtr func :: Int -> Int) 5
    Nothing -> error "Unable to lookup static ptr"

-- client sends function and argument together to be executed
  let fibC = closure fibPtr
  let totc = cap fibC (closure $ static 8)
  -- server executes closure
  print $ unclosure (decode (encode totc)::Closure Int)
```

In this example, we are performing all the steps locally but each of the needed steps are present, we would just need a transmission mechanism. Notice how we did not need to create a GADT and separate the building up of `fib` from its evaluation.

The challenge in both of these examples is that in order to dereference the static pointer or to execute the closure, we need to know the type. We could solve this by having a table that has the function pointers and their type, which might be already be stored at a lower level, or somehow use Template Haskell to assist. Polymorphic functions will be the most difficult to handle, and it might be the case that we are unable to do so. The goal would be to handle any function.

Ultimately, this direction of research fell outside of the bounds of this dissertation, but it would still be an interesting avenue to explore.

9.2 Conclusion

As programming has evolved over time there have been many situations where solutions for a wide array of applications take a similar approach. Sometimes these came about from necessity because of the selected programming language, other times they came about from programmers finding the best solution for a particular problem after years of work and being able to recognize problems that have the same shape. What we end up with is a design pattern [Gamma et al., 1995] that abstracts a way of solving a particular program from the program specifics of an application. These design patterns can often cross language boundaries and are usually found throughout the community before they are formalized or even named.

In addition to the case studies we have covered in this dissertation, below is a list of Haskell libraries that are instances of the remote monad (by way of using the ideas, not using the described framework) with the type information of the corresponding `send`-like functions shown in Figure 9.1.

- Haxl [Facebook, 2017] – Library for concurrent data accesses
- MongoDB [Hannan, 2018]– client used to connect to MongoDB server
- UI.NCurses [Millikin, 2016]– Haskell binding to GNU ncurses for terminal interactions
- Accelerate [McDonell, 2018]– Embedded array language for computations for high-performance computing
- threepenny-gui [Apfelmus, 2017]– Haskell to HTML/JavaScript GUI using browser
- Haste.App [Ekblad, 2017]– Haskell to JavaScript compiler

Haxl
runHaxl::

accelerate
run:: Arrays a => Acc a -> a

UI.NCurses
runCurses:: Curses a -> IO a

threepenny-gui
runUI :: Window -> UI a -> IO a

Haste.App
onServer :: Binary a => Remote (Server a) -> Client a

MongoDB
access :: MonadIO m => Pipe -> AccessMode -> Database -> Action m a -> m a

Figure 9.1: External examples of remote monad

Each of these libraries use monads to stage requests and call out to a "remote" entity to get results using a natural transformation from the staging monad to IO or some other monad.

There is a large number of monad tutorials out there ranging from describing monads as a cloud or as a burrito to an in-depth look at the theory behind them. There is a wiki page [mon, 2018] that has links to over 35 different tutorials to help people understand monads. The problem is that users have a hard time understanding what a monad is and how to use it, and later it finally clicks, only after using and experimenting with monads. Naturally, we think that if we were given that final piece in the beginning, then we would have understood monads right off the bat, which leads to yet another tutorial. But to understand monads, one has to use and experiment with them and gradually discover all the pieces needed. Once understanding has been achieved it seems obvious.

Now that we have looked at the ideas behind the remote monad, it likewise seems very obvious that we can make different bundling choices depending on the attributes of remote primitives, or that applicative functors make a great packet for transmission, or that we can chain natural transformations to make a modular composable network stack. But at the beginning of the research which has led to this dissertation, there were many libraries that made use of natural transformations to go from one functor space to another as evaluation functions. There were many applications that would make use of RPCs, either for performance reasons or to gain access to a rare resource. And even with all these authors working in the same space, the community at large had not put a finger on the abstraction or have an understanding of how this mechanism should behave in general.

We can cause a monad to run remotely by separating the structure and the evaluation of the monadic actions by using GADTs. Then after classifying the actions based on possible side-effects and returning results, if any, we can apply different techniques to bundle the requests and amortize the unavoidable network cost incurred from making remote requests. If we know the result of a request statically, then we can bundle the request until we reach a request that statically has an unknown result, and then we communicate a batch of requests to the remote service.

Monadic binds make bundling requests difficult if not impossible as future requests rely on the result of previous requests. To avoid this, we can factor a monad into a set of applicative functors that are connected by any necessary monadic binds. Applicative functors, at least at this time, seem to be the best structure to use for packets, as they are a natural fit for sending independent actions and handle the post-processing functionality needed for network requests. As we go from a GADT of remote primitive actions to a monad, and later break the monad into transmittable packets, natural transformations became a key mechanism to represent each of the stages. By chaining natural transformations together we can add and remove different layers to handle any application-specific details or to comply with a particular protocol.

The theory that these bundling strategies can give noticeable speedup is confirmed by the performance measurements that were presented for a number of blank-canvas benchmarks. Thus, remote monads and applicative functors are viable choices to add automatic bundling with minor changes needed to the user's program logic.

References

(2018). C# library - haxlsharp. https://github.com/joashc/HaxlSharp. [Online; accessed 2-Apr-2018].

(2018). Clojure library - muse. https://github.com/kachayev/muse. [Online; accessed 2-Apr-2018].

(2018). Clojure library - urania. http://funcool.github.io/urania/latest/. [Online; accessed 2-Apr-2018].

(2018). Monad tutorials timeline. https://wiki.haskell.org/Monad_tutorials_timeline. [Online; accessed 2-Apr-2018].

(2018a). Purescript library - fetch. https://pursuit.purescript.org/packages/purescript-fetch/. [Online; accessed 2-Apr-2018].

(2018). Scala library - clump. https://index.scala-lang.org/getclump/clump/clump-scala. [Online; accessed 2-Apr-2018].

(2018b). Scala library - fetch. https://index.scala-lang.org/47deg/fetch/. [Online; accessed 2-Apr-2018].

Achten, P. & Plasmeijer, R. (1995). Concurrent interactive processes in a pure functional language. In *Proceedings Computing Science in the Netherlands, CSN*, volume 95 (pp. 27–28).

Apfelmus, H. (2017). Hackage package `threepenny-gui-0.8.2.3`.

Birrell, A. D. & Nelson, B. J. (1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1), 39–59.

Bogle, P. & Liskov, B. (1994). *Reducing cross domain call overhead using batched futures*, volume 29. ACM.

Bringert, B. (2016). Hackage package `haxr-0.2.16`.

Capriotti, P. & Kaposi, A. (2014). Free applicative functors. *arXiv preprint arXiv:1403.0749*.

Clark, A. (2018). Hackage package `websockets-rpc-0.7.1.1`.

Dawson, J., Grebe, M., & Gill, A. (2017). Composable network stacks and remote monads. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell 2017 (pp. 86–97). New York, NY, USA: ACM.

Ekblad, A. (2017). Hackage package `haste-compiler-0.5.5.1`.

Elliott, C., Finne, S., & de Moor, O. (2003). Compiling embedded languages. *Journal of Functional Programming*, 13(3), 455–481.

Epstein, J., Black, A. P., & Peyton-Jones, S. (2011). Towards haskell in the cloud. In *ACM SIGPLAN Notices*, volume 46 (pp. 118–129).: ACM.

Facebook, I. (2017). Hackage package `haxl-0.5.1.0`.

Fancher, W. (2016). Hackage package `fraxl-0.1.0.0`.

Fette, I. (2011). The websocket protocol.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Gibbons, J. (2016). Free delivery (functional pearl). In *Haskell Symposium*: ACM.

Gifford, D. K. & Glasser, N. (1988). Remote pipes and procedures for efficient distributed communication. *ACM Transactions on Computer Systems (TOCS)*, 6(3), 258–283.

Gill, A. (2014). Domain-specific languages and code synthesis using haskell. *Queue*, 12(4), 30.

Gill, A. & Paterson, R. (2017). Hackage package `transformers-0.5.5.0`.

Grebe, M. & Gill, A. (2016). Haskino: A remote monad for programming the arduino. In *International Symposium on Practical Aspects of Declarative Languages* (pp. 153–168).: Springer.

Grebe, M. & Gill, A. (2017). Threading the arduino with haskell. *Post-Proceedings of Trends in Functional Programming*.

Group, J.-R. W. et al. (2012). Json-rpc 2.0 specification.

Haller, P. & Miller, H. (2016). Distributed programming via safe closure passing. *arXiv preprint arXiv:1602.03598*.

Hannan, T. (2018). Hackage package `mongodb-2.3.0.5`.

Hudak, P. (1998). Modular domain specific languages and tools. In *International Conference on Software Reuse* (pp. 134–142).: IEEE Press.

Leijen, D. & Meijer, E. (1999). Domain specific embedded compilers. In *Conference on Domain-Specific Languages* (pp. 109–122).: ACM.

Liang, S., Hudak, P., & Jones, M. (1995). Monad transformers and modular interpreters. In *Symposium on Principles of Programming Languages* (pp. 333–343).: ACM.

Liskov, B. & Shrira, L. (1988). *Promises: linguistic support for efficient asynchronous procedure calls in distributed systems*, volume 23. ACM.

Maassen, J., Van Nieuwpoort, R., Veldema, R., Bal, H., Kielmann, T., Jacobs, C., & Hofman, R. (2001). Efficient java rmi for parallel programming. *ACM Transactions on Programming Languages and Systems*, 23(6), 747–775.

Marlow, S., Brandy, L., Coens, J., & Purdy, J. (2014). There is no fork: An abstraction for efficient, concurrent, and concise data access. In *International Conference on Functional Programming* (pp. 325–337).: ACM.

Marlow, S., Peyton Jones, S., Kmett, E., & Mokhov, A. (2016). Desugaring haskell's do-notation into applicative operations. In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016 (pp. 92–104). New York, NY, USA: ACM.

McBride, C. & Paterson, R. (2008). Applicative programming with effects. *Journal of Functional Programming*, 18, 1–13.

McDonell, T. L. (2018). Hackage package `accelerate-llvm-ptx-1.1.0.1`.

Microsystems, S. (1988). *RPC: Remote Procedure Call Protocol Specification: Version 2.* Technical report, RFC 1057.

Miller, H., Haller, P., & Odersky, M. (2014). Spores: A type-based foundation for closures in the age of concurrency and distribution. In *European Conference on Object-Oriented Programming* (pp. 308–333).: Springer.

Millikin, J. (2016). Hackage package `ncurses-0.2.16`.

Nelson, B. J. (1981). *Remote procedure call.* Technical report, Carnegie-Mellon Univ. Dept. Comput. Sci.

Peyton Jones, S. (2014). Static pointers and serialization. `https://ghc.haskell.org/trac/ghc/blog/simonpj/StaticPointers`. Accessed: 2016-11-16.

Peyton Jones, S., Vytiniotis, D., Weirich, S., & Washburn, G. (2006). Simple unification-based type inference for gadts. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN*

International Conference on Functional Programming (pp. 50–61). New York, NY, USA: ACM.

Pitt, E. & McNiff, K. (2001). *Java. rmi: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc.

Rupp, J.-P. (2016). Hackage package `json-rpc-0.7.1.1`.

Shakib, D. A. & Benson, M. L. (2001). Multiple procedure calls in a single request. US Patent 6,321,274.

Spector, A. Z. (1982). Performing remote operations efficiently on a local computer network. *Communications of the ACM*, 25(4), 246–260.

Van Deursen, A., Klint, P., & Visser, J. (2000). Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6), 26–36.

Van Weelden, A. & Plasmeijer, R. (2002). Towards a strongly typed functional operating system. In *Symposium on Implementation and Application of Functional Languages* (pp. 215–231).: Springer.

Waldo, J. (1998). Remote procedure calls and java remote method invocation. *IEEE concurrency*, 6(3), 5–7.