M.Sc. Thesis
Master of Science in Engineering

DTU Fotonik
Department of Photonics Engineering

Exploring the use of purely functional programming languages for offloading of mobile computations

Christian Kjaer Laustsen
s124324 @ DTU
20176018 @ KAIST

Kongens Lyngby 2018

DTU Fotonik

Department of Photonics Engineering

Technical University of Denmark

Ørsteds Plads

Building 343

2800 Kongens Lyngby, Denmark

Phone +45 4525 6352

info@fotonik.dtu.dk

www.fotonik.dtu.dk

Abstract

We continue to push the limits of mobile devices, but not all components can keep up with the speed of development and the demand for more processing power on-device. Battery life have not increased along with the power requirements, creating a need for methods that can preserve energy while still providing the processing power that consumers demand.

Offloading of computations is not a new idea, but it still has not seen a major breakthrough. In this thesis we investigate why, when and where it is beneficial to offload computations, based on the available literature. We take a look at existing systems and previous work to discern why their popularity has not reached the mainstream audience of developers, along with looking at the design limitations and considerations that went into the systems.

We use this knowledge to explore different approaches to offloading, and fianlly settle on one, which we then implement as a proof-of-concept.

Preface

Thie MSc thesis was prepared during the fall of 2017, both during my stay at KAIST in South Korea and at the department of Photonics Engineering at the Technical University of Denmark in fulfillment with the requirements for acquiring a M.Sc.Eng. degree in Telecommunication.

I would like to show my sincerest gratitude and thanks to my two supervisors Henrik Lehrmann Christiansen and Sung-ju Lee for their help and feedback throughout this project, and for giving me the opportunity to write my thesis while on exchange in South Korea.

Kongens Lyngby, January 28, 2018

Christian Kjær Laustsen (s124324)

Contents

List of Tables

List of Figures

List of Listings

Introduction

As we continue to push the limits of our mobile devices, form factor and size still remain a major consideration during the design process. This often manifests itself in the form of decreasing battery size or cutting down on external ports, while trying to fit everything into a smaller device. As such, many different approaches are explored and utilized to squeeze out every last drop of battery, while retaining as much performance as possible—one of these approaches is to offload heavy or time consuming computations off of the device, so the CPU can remain idle as much as possible and thereby save energy.

From a mobile application developers' perspective, offloading is a trade-off between complexity of the code base and the limitations of the devices being developed for. More and more things are being pushed onto the device. Take for example Apple's recent addition of an on-device Deep Neural Network (DNN), which is part of the A11 chip powering the iPhone 8 and iPhone X [Simonite 2017]. This move makes sense for the use cases they envision, like triggering their voice assistant Siri and adding Augmented Reality (AR) features to the camera, which require very low latency to appear believable. On the other hand, once we thread into the domain of computations that are far more time or resource intensive, while not having a strict low latency requirement, we find ourselves in the territory of where offloading computations are highly beneficial, assuming the right network conditions.

Why then, is it we do not see a widespread adoption of frameworks and technologies that allow us to easily offload computations from the device, but instead mainly see it in the small ad-hoc scale of using Application Programming Interfaces (APIs) to talk to servers, but usually only by necessity (e.g. accessing a database)? Are they too complex, like UpShift which needs to transfer the memory to the server and is limited

to ARM-based servers? Do they require too much buy-in, like MAUI essentially being the runtime? Is it a lack of documentation on setup and little exposure to the majority of the developer community outside of academia?

One thing that is common for all of the existing approaches, is that they are bugged down by some major design limitations, state and side-effects, because of the language paradigm they are implemented in. MAUI and CloneCloud both transfer the state with them, MAUI on the method level and CloneCloud on the thread level. UpShift, as we mentioned, keeps the memory in sync uni-directionally from the client to the server, and MobileCOP requires the programmer to manually pass the state needed on to the platform. Of course, none of them can offload code with side-effects intended for local execution, such as UI changes, I/O or methods sharing native state—the offloading system either needs to be aware of these, which can be extremely difficult, or leave the task up to the programmer.

But what if we could actually differentiate between something that performs I/O, of which shared state and UI are a subclass of, and also guarantee that there are no side-effects in the function we are looking to offload? Enter here the concept of purity, specifically purely functional programming languages. We now have the ability to tackle two of our main design limitations, 1) no need for state: because of referential transparency, we are only concerned with the input of the function, and 2) we can immediately see from a functions' type signature, if it is able to perform side-effects or not. This means we are able to tell which functions we are *certain* are offloadable, while the remaining part is dependent on the type of I/O it does. For example, fetching a webpage and transforming its output could still be offloaded, while changing the state of the UI is something that cannot.

This finally brings us to the topic at hand—this thesis explores, as the title implies, the use of purely functional programming languages specifically for offloading of (mobile) computations, while also serving as a meta analysis on the topic of offloading, by examining the currently published research and gathering the results, suggestions and advices into a unified

overview. The main goal of the thesis is then to use this knowledge to explore which approaches are feasible, and finally implement one of these based on its merits. The final solution is then tested on an actual mobile device, to see how it performs in various conditions. More succinctly, our goals can be stated as:

- Gathering together the results, suggestions and advice from the literature on offloading.
- Compare existing systems and how they have progressed over time.
- Explain what makes purely functional programming an interesting angle to look at.
- Explore various approaches to making a system for offloading in a purely functional language.
- Implement the most suitable approach.
- Evaluate the approach, and conclude.

Chapter 1 serves to motivate why offloading of computations is a worthwhile pursuit, by examining why one would want to offload in the first place, then goes on to give an overview of common areas in which it can lower energy consumption. Finally, we investigate when and under what exact conditions offloading is beneficial—and conversely—when it is not. Chapter 2 then gives an overview of existing systems, on the application level, that have been developed to facilitate offloading of computations. In chapter 3 the reader is taken on a tour of what it means to be a purely functional programming language, what we mean when we say "pure" computation, differences in evaluation model, and finally a primer on Haskell, which serves to familiarize the reader with the syntax of the Haskell language, since this will be the focal point of this thesis. Continuing to chapter 4, a series of different approaches are investigated as to how offloading could be implemented in a purely functional language, with a focus on Haskell so that concrete suggestions can be given. This leads us on to chapter 5, which implements one of the approaches discussed in chapter 4. In chapter 6 we evaluate and discuss the outcome of the approach taken, and conclude on our work.

CHAPTER 1

The Case for Offloading

Offloading of mobile code, or *remote execution*, is not a new idea at all, and has been around for some time, yet it has never really gained widespread traction in mainstream programming. Most mobile applications still mainly use servers for communicating with databases and other services that require centralization, such as social networks, etc. Even though the literature on mobile offloading has consistently shown the benefits of offloading, it has still not had a major impact in mainstream development; perhaps existing solutions have simply not been general and flexible enough to satisfy the requirements of most developers, and this is the cause of the low adoption.

In this chapter, we will show that there is indeed still a place for mobile offloading, a collection of cases of how earlier works have saved energy with offloading, along with a gathering of the literature, answering under which conditions offloading would be favorable.

1.1 Saving Energy with Offloading

Arguably, the main objective of any offloading system has chiefly been to lower energy consumption on the mobile device itself. A secondary effect of this is that often times it would also increase performance of applications, by running the computations on stronger machines in the cloud. A lot of factors go into what impacts energy consumption, such as execution time, latency, network conditions, bandwidth, packet drops, etc.

In [Nabi et al. 2015], the authors present a formative study on the effects of mobile computation, trying to answer several questions, such as what impact input size, bandwidth and network conditions have on the performance of offloading. The results clearly show a gain in both

performance and energy conservation when offloading on fast networks (e.g. WiFi and LTE), while the case for slower networks are not as clear cut, as it depends on the complexity of the task being offloaded.

For example, Optical Character Recognition (OCR) is a commonly used application used to evaluate the performance of an offloading system, because it is a moderately compute intensive task, that also highly depends on the complexity of the input it gets, not only on the size of the image. As such, with enough bandwidth transfer and low enough latency, as is the case usually with WiFi and LTE, one can get energy savings by offloading it to the cloud. On the other hand, if the connection is poor or bandwidth low, like on 3G, a image that is low in complexity is likely to incur a penalty if offloaded. We will look more at these considerations in sec. 1.2, about when to offload, and under what conditions.

Comparison of Previous Systems

Let us take a look at some of the concrete energy improvements that have been achieved in existing systems. We will take a closer look at all of these systems in chapter 2—for now, we will simply refer to them by their name.

Since each evaluation has been done in different manners, we try to relativize them, by summing them up to a range of their average results. For example, MAUI tests for four different Round-trip Time (RTT) on WiFi, one on 3G and on local. This will be summed up as local = 100% of energy consumption, WiFi = 10%-14% of energy consumption, and 3G = 25%. Compare this to CloneCloud, for example, which only tests with one RTT on WiFi, one on 3G, and one on the local device. We will take a maximum of two test cases from each evaluation. Additionally, since local execution is the base line (i.e. always 100%), it has been changed to show the Joule measurement of the test, or battery percentage drain (i.e. “6%” means that 6% of the battery has drained), for which the other values are relative to. With this methodology, we have constructed the table, shown in tbl. 1.1.

Table 1.1: Comparison of results across existing offloading systems

	Local	**WiFi**	**3G**	**Test**
MAUI	28J	10–14%	25%	Facial Recognition
	50J	66–80%	110%	Video Game
CloneCloud	155J	22%	43%	Image Search
	40J	50%	125%	Behaviour Profiling
UpShift	6% drain	66%	N/A	Handwriting (100 iterations)
	12% drain	58%	N/A	Handwriting (200 iterations)
MobiCOP	173J	9%	16%	Pure Java Computation
	1332J	13%	19%	Large Input/Output

Admittedly, this can seem a bit confusing initially, but the point still manages to come across—there are significant gains to be achieved by offloading code. These gains primarily show themselves in the form of shorter execution time, which in turn means that the CPU needs to stay active less time, and also opens the opportunity for the screen needing to be on for less time, saving further energy.

For an overview of the values used in tbl. 1.1, refer to appendix A, which collects the measurements from the various papers on these systems.

1.2 Where and When to Offload

Now that we have established that offloading has the ability to allow for significant savings in energy consumption, and also improve execution time, we will take a look at when exactly one would want to offload.

In [Golkarifard et al. 2017], the authors aim to establish a set of guidelines and recommendations, based on a review of the literature along with their own experiences. The paper is fairly recent, so this gives us some hope that it is up-to-date with current technologies and practices.

The main network factors impacting offloading are:

- Latency, and by extension RTT.
- Bandwidth to the remote execution server.
- Packet loss ratio along with number of packets transmitted.
- Signal strength, although this is closely related to packet loss ratio.
- Network type, e.g. Bluetooth, 3G, WiFi, 4G. The energy required to transmit differs on each of these components, with WiFi requiring the lowest amount of energy to transmit.

For the application side, we mainly should consider:

- Execution time of a method.
- Input and output size.

As for what methods that have high potential for offloading, we have:

- CPU/Compute intensive operations, with relatively small input/output size. Some examples of these include (not taking into account I/O size) OCR, Machine Learning (ML), image processing, video encoding/decoding, graph related computations, such as path algorithms in games, shortest route in maps, etc.
- Parallelizable tasks that are data independent.
- Methods requiring large amounts of external data transfer, such as downloading an Internet site and processing it, or making a lot of database calls for a computation.

Where to Offload

A crucial aspect to offloading is the latency to the remote server that the code should be run on, and therefore location matters. [Golkarifard et al. 2017] divides these into three main categories:

- Cloud: Remote server connected via the Internet.
- Cloudlet: Server located on the same WiFi.
- Ambient Cloud: nearby devices, such as smartphones, tablets, and wearables.

Each have their own tradeoffs. The ambient cloud usually does not present itself with much computational power, but can be used for small parallelizable tasks, that can be distributed to a lot of devices. Cloudlets offer the superior power of servers like the cloud, at a lower latency, but comes at a significant cost to infrastructure, and limits the pervasiveness of offloading, since it is limited to the networks the cloudlets are placed on. Finally, clouds offer the most flexibility with regards to infrastructure and power, but obviously comes at the cost of being further away from the mobile device, and therefore incurs a higher latency cost.

The only consideration really impacted by the location of the remote execution server is latency and RTT. It will certainly affect the rest of the parameters we look at, but only since it is part of the equation to calculate the impact of these. In this thesis we will mostly be focusing on a cloud environment, but there is nothing stopping the approach from being used on the other two—cloudlets are basically interchangeable with cloud servers, but ambient clouds would require special setup, and accompanying clients on each device.

Network Conditions

The bulk of our considerations stem from network related considerations; the network type, bandwidth, packet loss ratio and signal strength.

We can quickly dismiss using Bluetooth for offloading. In [Golkarifard et al. 2017] they give an example of offloading a 300 KB image in 0.1 second, which would require a 24 Mb/s connection to achieve $((300\text{kB} * 8\frac{\text{kb}}{\text{kB}})/0.1\text{s} = 24\text{Mb/s})$. This would saturate the maximum throughput of Bluetooth version 4.0 (at 25Mb/s), according to table 1 on page 3 in the paper. On the same page, table 2, we note that WiFi versions coming after 802.11b, that is 802.11b, n and ac, have speeds at 54Mb/s, 150Mb/s and 411Mb/s respectively.

In [Huang et al. 2012] the authors give a thorough comparison of the different radio technologies, and their energy per bit transferred. For reference, their results are included here in fig. 1.1.

We see that LTE downlink compares well with WiFi up- and downlink, and LTE uplink is slightly more expensive to perform, especially in lower transfer sizes, but is amortized as the data size increases. 3G compares

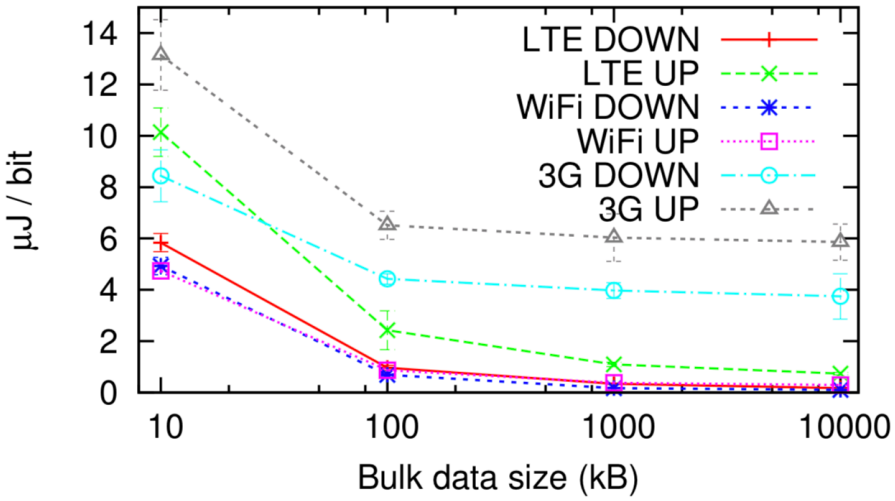

Figure 1.1: Energy per Bit Transferred [Huang et al. 2012:9]

considerably worse to both, especially in the uplink. From this, we can get an intuition of what network conditions are favorable to offloading.

For concrete results, we go back to [Nabi et al. 2015]

Input/Output Size

Another major factor we have to take into consideration is the input/output size that we will be transferring. These are highly affected by both throughput and RTT. In [Cuervo et al. 2010], the authors measure the energy consumed by uploading 10KB and 100KB of code over different RTT values. For reference, their measurements are included here in fig. 1.2.

We see that the the energy consumption increases more quickly with larger data sizes, as the RTT goes up. This fits naturally with our intuition that the more packets that are sent, the more ACKs must be responded with, increasing the impact of RTT—at least over Transmission Control Protocol (TCP). As such, we must clearly combine our information about the current RTT values we are getting, with the data

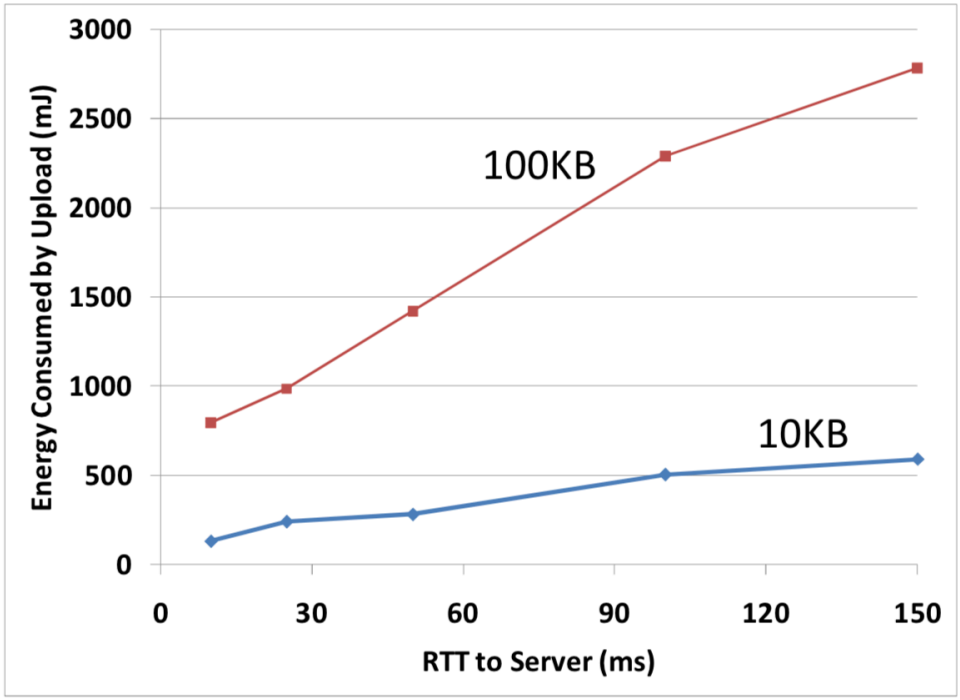

Figure 1.2: Energy consumed by offloading 10KB and 100KB over different RTT values [Cuervo et al. 2010:3]

size that we want to offload, and how much impact the specific function has on energy consumption.

Other Considerations

[Golkarifard et al. 2017] also mentions creating a Virtual Machine (VM) manager to automatically create VMs as needed. This process can be entirely automated by most Infrastructure as a Service (IaaS) providers, such as with Amazon Web Services (AWS) OpsWorks, which allow easy scaling based on metrics, such as CPU load on the servers, time, and much more. For example, one server could be running in during low traffic times, such as during the night, and four additional servers could then be spun up during the day, where load is estimated to be much

higher. One could also automatically scale up the number of VM, by setting a rule that if a server is utilizing over 90% of CPU for more than 5 minutes, then spin up two new instances. If then the CPU utilization is under 40% for 10 minutes, then remove one instance, repeating until reaching a minimum set of instances.

1.3 Summary

We have demonstrated that there indeed is a clear case for offloading of computations still, even though widespread adoption might still be low. The effectiveness of this has been backed up by concrete measurements from previous systems.

Furthermore, we have layed out some which factors must be considered when designing an effective offloading system, based on literature review, along with a more detailed reasoning as to how some of these factors can affect the system performance. For example, it is clear to us that if we have a good RTT to the remote execution server, and satisfactory throughput, such as on WiFi or LTE, then it is almost always beneficial to offload. The situation becomes much more clouded on networks with lower throughput and worse latency, such as 3G and 2G, and here input size makes a much bigger difference, and must be taken into account.

CHAPTER 2

Related Work

There has already been several systems, frameworks, and approaches developed to perform remote execution—that is, offloading code—and these have indeed progressed over time. However most, if not all, suffer from similar constraints and limitations.

To better understand what has already been attempted, let us take a closer look at the more popular of the approaches that have spawned over time. We will look at each of them, how they function, what their limitations are, and finally try to sum up both the commonly met challenges of remote execution systems, and the decisions that the systems seem to agree on. We will go through these chronologically, so it is more clear how the research has progressed over time. Note that we will not go into each paper's evaluation of when to offload, as that has been summed up in sec. 1.2.

This chapter serves to motivate what design considerations we are trying to tackle in later chapters, namely chapter 4 and chapter 5.

2.1 MAUI

The first system we will explore is MAUI, from [Cuervo et al. 2010], which describes a new approach to offloading code at the time. Previous attempts have mainly been either entirely up to the programmer, along with the logic of under what conditions this should be performed, or a more all-in approach with full process or VM migration into the cloud environment, automatically managing the state and code transfer.

MAUI then tries to combine the best of these two approaches. It runs in a VM, specifically Microsoft's .NET Common Language Runtime (CLR), although they mention they could have done the same with the Java VM. They then use the fact that VMs abstract away the underlying

hardware, to run the same code both on the mobile device and remotely on the server, as can be seen in fig. 2.1, along with the other components of the MAUI system, to be explained later.

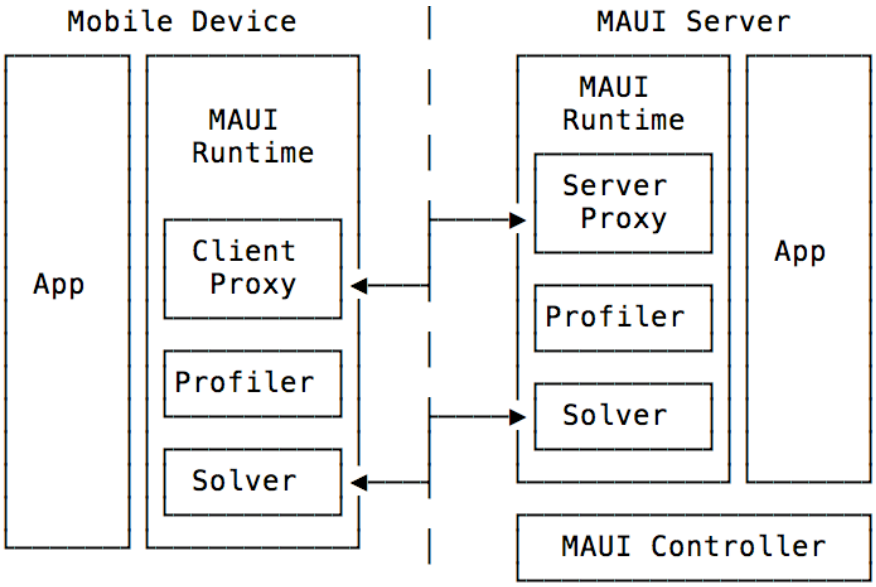

Figure 2.1: MAUI Architecture

Using reflection, MAUI identifies which methods to offload, and then during the runtime they profile each method to estimate its cost, defined as the combined CPU and network cost, along with wireless connectivity cost such as bandwidth and latency. They keep updating this estimate during runtime, based on these factors. Mainly wireless connectivity and serialization size will change dynamically. They do mention that they are aware that profiling on *every* method call adds overhead to the application, but do not go into detail whether they then throttle this or not.

MAUI still needs some intervention from the programmer, in the aspect of manually annotating methods for which MAUI should *consider* offloading, and making sure *not* to annotate those that should not be considered. The latter consist of 1) User Interface (UI) code, 2) I/O code that only makes sense to run locally (e.g. GPS, sensors, etc), and

3) code that cannot handle being re-run, such as financial transactions. The rest is then taken care of by the MAUI runtime component. If a method finishes offloading successfully, this data is incorporated into the profiler, and if it fails, it falls back to running it locally on the phone, which naturally incurs a small energy penalty.

The client and server proxies handle the serialization and network communication of the offloaded methods, and the MAUI controller handles the authentication and resource allocation for the incoming requests. Finally, the solver is the decision engine of MAUI, which tries to determine, at runtime, if it makes sense to offload piece of code, weighing performance and energy considerations.

Since MAUI works in an imperative programming language, it needs to take into account state transfer. This is done by having the methods that are annotated, be wrapped in additional code, adding a new input argument, the current state, and an additional output argument, the new state. Using reflection, they traverse the in-memory data structures used by the program. Beyond the explicit parameters to the method, they take a conservative approach and add all the current object's members and nested complex objects, along with the state of any static classes and public static member variables—it is clear that this would add significant network transfer overhead, which most likely is not needed. They try to minimize the transfer size by only transferring deltas of the state.

Cutting this short, let us sum up the limitations and challenges met, along with the rest of the challenges mentioned throughout the paper:

- Manual annotation of methods to offload, where the programmer must provide their own guarantee that their adhere to the three restrictions mentioned.
- Must transfer the state of the object and additional static classes (even though they try to optimize this), because they do not know what is used and what is not.
- Can only offload methods from a single thread at any given point in time.
- Manual intervention must be taken to batch remote operations.

With this in mind, MAUI still achieves significant gains in lowering energy consumption, when it is run on WiFi, and sometimes even on 3G.

2.2 CloneCloud

The next system is CloneCloud, from [Chun et al. 2011], which came out closely after MAUI, but was actually already envisioned in [Chun and Maniatis 2009], and referenced in the MAUI paper. As such, it is not surprising that this system closely resembles MAUI, in that it features a profiler, and a partitioning of the full code into a smartphone and remote server environment, running on a VM.

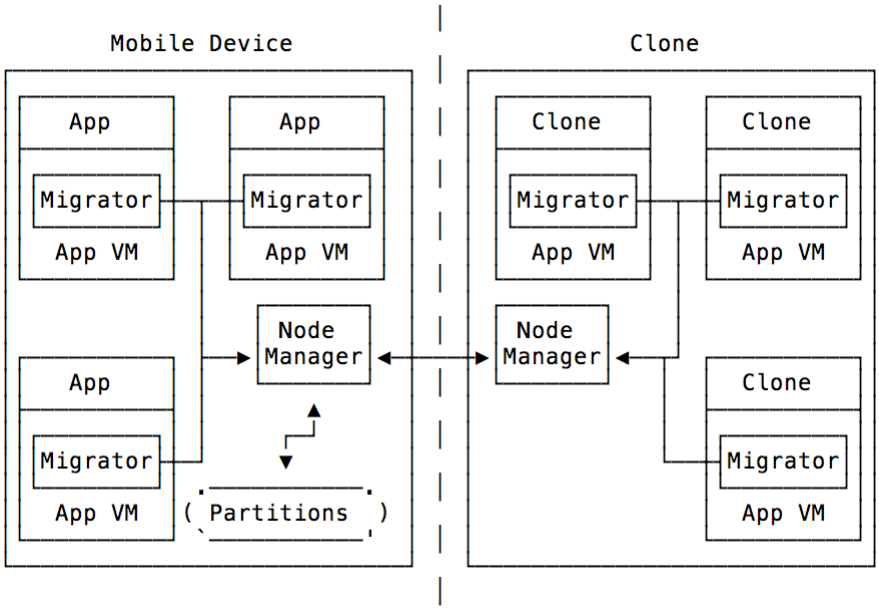

Figure 2.2: CloneCloud Architecture

In essence, CloneCloud and MAUI share much of the same architectural ideas, as can broadly be seen from fig. 2.2 (if you squint hard enough), but differ in implementation details. From [Jiao et al. 2013], we can sum up some of the superficial differences as:

- CloneCloud works at the thread level, whereas MAUI works at the method level.
- Both transfer a substantial amount of state; virtual state, program counter, registers and stack/heap for CloneCloud.
- Neither can offload UI operations, but CloneCloud allows I/O to be offloaded, although it does pin machine specific features to the machine with a special annotation (such as GPS).
- CloneCloud does *not* require manual annotations, as MAUI does.

We go back to the original paper to dig deeper into these details. The partitioning of CloneCloud (i.e. what to offload) is done offline, and can be run for multiple conditions, creating a database of partitions. This is in stark contrast to MAUI, which does all this work online (i.e. during runtime). The same goes for the solver. As such, CloneCloud lowers the runtime overhead, but does it at the loss of flexibility.

One interesting thing about the thread granularity, is that most mobile runtimes keep UI interactions on the main thread, and often discourages doing UI operations from elsewhere. This lends itself nicely to, for example, offloading separate worker threads, while keeping the UI thread running, and only blocking if it tries to access the worker thread before it has returned. When CloneCloud reaches a migration point, its thread migrator suspends the thread, collects up its state and passes that state on to a node manager for data transfer.

CloneCloud of course manages to gain significant improvements in execution times and energy consumption, when the input size goes up to 1MB and over, on WiFi and in some instances also on 3G.

2.3 UpShift

Jumping forward a few years, we will take a look at UpShift, presented in [Lin 2014], which brings some new ideas to the table, mainly motivated by the problem of state, and how to alleviate the pain caused by this.

UpShifts key idea is to continuously replicate the mobile device memory to the cloud, in a so-called Distributed Shared Memory (DSM) system, allowing for very fine-grained offloading at the method level. To

make this memory replication efficient, they use a technique they called *compressive offloading*, which is based on *compressive sensing*. A simplification of this technique is that it transfers deltas that can be compressively sampled, with low encoding complexity (saves energy on the device), at the cost of higher decoding complexity (which is acceptable, since it happens on the server).

The implementation in the paper only supports unidirectional memory replication, going from mobile device to server, which means that offloaded methods cannot alter state, since this would not be transferred back. Since the memory is replicated, they can skip object serialization when offloading, and save time upon execution, along with enabling them to pass along pointers and address translations in the methods.

In the runtime on the mobile device, they introduce a shim layer, which handles the replication and redirecting method invocation to the server when suited, along with a daemon on the cloud server that decodes and applies the replicated memory, and handle the redirected method invocations. The shim layer takes into account network conditions, much like the profiler in MAUI, and battery levels, along with allowing the user to control offloading by disallowing it altogether or turning it to be always on.

The shim layer is done by creating an alternative memory allocation, `upshift_alloc`, which provides the additional features needed for the memory replication, compared to the default `alloc`, and then using this to allocate new Objective-C objects. These objects using `upshift_alloc` are then also cross-compiled for the server-side.

UpShift shows promising results for operations that take longer to perform than it takes to replicate the memory. At least, this is the case in their evaluation, although it remains to be seen what effect this replication has over the long-term, when it is not just for cases that are well suited for offloading, like the OCR they did in their tests.

2.4 MobiCOP

The final mobile code offloading platform, that we will take a look at, is MobiCOP, presented in [Benedetto et al. 2017]. MobiCOP sets out

to provide a more practical solution that can be used by developers, compared to the more drastic approaches like MAUI and CloneCloud, which require changes to the underlying VM on the phone. It does so by taking a library approach with the goal of compatibility with major IaaS providers (e.g. AWS and Google Cloud Platform).

MobiCOP acknowledges that the need for state transfer is one of the biggest challenges in developing code offloading frameworks, and as such tries to avoid it by using a different approach. Instead of offloading arbitrarily at the method level, MobiCOP uses the existing practices of Android development, by extending the core Android `Service` class, and focuses more on long-running intensive background tasks. Because a `Service` class gets its state passed as input parameters upon initialization, MobiCOP already knows what it needs to transfer to the server, instead of having to traverse the heap or other tricks to determine the required state.

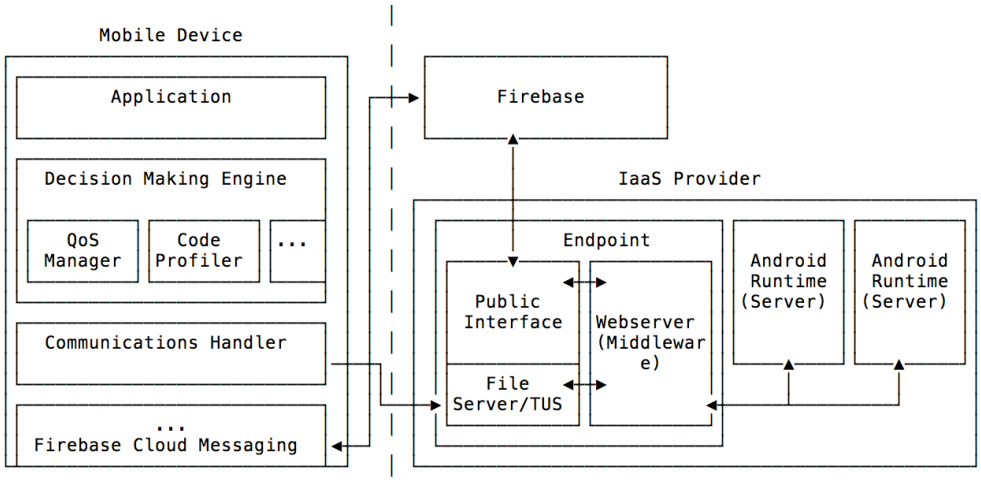

Figure 2.3: MobiCOP Architecture

The MobiCOP system, shown in fig. 2.3, also features a decision engine that, like MAUI, learns continuously by making use of past executions to predict future execution time and output size. Because it relies on past executions, it will initially run in one of two modes; concurrent or optimistic. Concurrent executes the function both locally and remotely

at the same time, while optimistic mode runs it based on the output of the decision engine.

The Firebase component is used as the main communication layer between the mobile device and the MobiCOP cloud environment. Using the Firebase channels allows for asynchronous and resumable data transfers, and reuses the same sockets which the Android platform uses for push notifications, helping lower energy consumption. If the application state is small enough, it will get packaged along with the Firebase messages, else it will be stored in a temporary file, and transferred to the file server via TUS[1].

Even though MobiCOP is "only" a library, it still manages to get energy savings up to a factor 11, compared to executing the code entirely on-device. This alludes that even with a less drastic approach, compared to MAUI, CloneCloud and UpShift, one can still get considerable benefits.

2.5 Commonalities

A crucial element that keeps coming up in all of the previous work is the need for a profiler that can take network conditions into account. Without this, one cannot be sure that the offloading will actually be beneficial to the user, since network conditions change very frequently. As such, a profiling component can be seen as a key component of any offloading system intended for serious use.

Another thing we keep seeing in most systems, except for MobiCOP, is the need for state transfer and handling state. This greatly complicates the offloading systems, and MobiCOP neatly side-steps this since the user needs to supply the `Service` class with the state, via input arguments. This is also one of the key motivators for this thesis, since, as we will see later in sec. 3.2, purity gives us a way to remove the concern about state.

A final thing that can be said about all of these is that they are platform specific. MAUI runs on the CLR, CloneCloud on the Java Dalvik VM, UpShift in Objective-C and MobiCOP using the Android

[1]TUS is a protocol for resumable file uploads over HTTP—https://tus.io

`Service` class. This ties their solutions to a single platform, and will be something we take into consideration in chapter 4, when we go through possible approaches to solving this. In other words, portability matters.

2.6 Summary

As we have seen, there has been a progression from more heavy-handed approaches, that extend or change the VM for the target platform, to less intrusive and more practically-oriented approaches, that give the developer control over the usage, and can be deployed to stock mobile devices. In fact, the last example, MobiCOP, shows that simply implementing the offloading system as a library is entirely feasible.

We note that a common trend through all is the need for a decision engine, which—except for CloneCloud—is usually a runtime component that makes decisions based on the current network conditions, so to give the best performance and lowest energy consumption. Many also incorporate continuous learning, which attempts to improve the performance of the offloading system.

Finally, it is clear that state has persistently been a concern when designing these systems/frameworks, since all of them are implemented in imperative languages, where state is unavoidable. Addressing this is one of the key motivations for this thesis.

CHAPTER 3

Pure Functional Programming

In this chapter, we will explore what it means to be a *purely functional programming language*, and what consequences is has for program design.

3.1 Functional programming

Functional programming is a paradigm, under the declarative programming paradigm, which stands heavily in contrast to imperative programming, positioning itself in the opposite end of the spectrum. Whereas an imperative program can be seen as a series of state changing steps, functional programming treats computations as evaluations of mathematical expressions. Boiling it down to the essentials, a functional language is an extension of lambda-calculus, with some added constructs to make the language richer. Common for functional languages are that they support concepts such as first-class functions, i.e. support for passing functions around as arguments, and higher-order functions, which are functions that can take other functions as input, or even return a function. The astute reader might recognize these concepts if they have experience with something like Python, JavaScript or Ruby (and many other languages), and they would be right seeing as functional programming concepts have spread out into mainstream languages more and more. An example of first-class functions in first Python and then Haskell are shown in lst. 3.1 and lst. 3.2 (both run in their respective REPLs).

In lst. 3.1 and lst. 3.2 we actually also demonstrate higher-order functions, in the form of `map`, which in Haskell has the type `map :: (a -> b) -> [a] -> [b]`, read as `map` is a function which first argument is a function taking an `a` and returning a `b`. It then takes a list of `a`s as the second argument and finally returns a list of `b`s.

Listing 3.1 First-class functions in Python

```
>>> def add3(a): return a + 3
>>> map(add3, [1,2,4])
[4, 5, 7]
```

Listing 3.2 First-class functions in Haskell

```
Prelude> let add3 a = a + 3
Prelude> map add3 [1,2,3,4]
[4,5,6,7]
```

A functional programming language can be said to be a language that primarily interacts by calling, manipulating and passing around functions, which is why we also see it in multi-paradigm languages, such as Python etc.

3.2 Pure vs Impure

Now that we have an idea of what a functional language is, we can talk about *purity* and *impurity*. When we say a language is pure or impure, we are in essence describing how we treat computational effects, such as I/O. In [Sabry 1998] the author tries to come up with a formal definition of what it means to be pure.

> *In essence, the new definition asserts that a language is purely functional if it can be implemented using either call-by-value, call-by-need, or call-by-name, with no observable difference between the different strategies – other than termination properties.* [Sabry 1998:1]

Which we can interpret as: no matter how many times you evaluate a function (as would happen with call-by-name), or when you evaluate them (call-by-need vs call-by-value), *iff* they are pure, it will not matter

since given the same input a function would give the same output, barring of course non-termination.

More informally, and what is usually also the more popular and simple explanation, is that purity means that we get referential transparency, as we know it from mathematics. This was championed by John Launchbury and Simon Peyton Jones in [Launchbury and Peyton Jones 1995].

To reword the above in perhaps a more digestible way, if a function receives the same input, it must produce the same output, then we can consider the function to be referentially transparent. Take for example the code shown in lst. 3.3 and lst. 3.4.

Listing 3.3 Example of a referentially transparent function in Python

```
def appendList(a, b):
  return a + b
```

Listing 3.4 Example of a referentially transparent function in Haskell

```
appendList :: [a] -> [a] -> [a]
appendList a b = a ++ b
```

No matter how many times we give the arguments, `[1,2,3]` and `[4,5,6]`, the result will be `[1,2,3,4,5,6]`. This is something that we can guarantee in the types of the Haskell code, but only something we can assume in the Python code, since it is untyped. Similarly, the statically type Java language would also only hold assumptions, and no guarantees, since in Java we have no way of talking about effects. This is where Haskell and similar purely functional languages, which have seemed quite similar so far, diverge!

In lst. 3.5 we note that the return type of the function has become `IO [a]`. Why is that? If we inspect the function itself, we see that we suddenly added a `print` statement inside the function definition. The `print` function has the type `print :: Show a => a -> IO ()` which

Listing 3.5 An impure function in Haskell

```
appendList :: [a] -> [a] -> IO [a]
appendList a b = do
  let newList = a ++ b
  print newList
  newList
```

means it takes a *showable* value, performs an I/O operation and finally returns `()` (called unit, which is an empty value). By using `print` inside of `appendList`, the type system now requires `appendList` to be marked as working inside I/O as well—it is by this mechanism that we can guarantee that a function which does not have any I/O in its type signature will not perform any side-effects—or as a popular saying in the Haskell community "It will not launch nukes", in the context of running a pure function.

Interestingly, in [Jiao et al. 2013], the authors actually bring up the topic of using functional programming languages, referencing Haskell in their sources, exactly because of this trait. The authors propose that because there is no global state and mutability—because of purity—and that code re-execution is safe—because of referential transparency—this would solve, or at least alleviate, several problems in MAUI and CloneCloud (which were the ones out at the time of the paper):

- It would remove the need for shipping all variables and objects to the server, in case they are needed.
- Code re-execution, which was a restriction of MAUI, is now safe to do.

Unfortunately, there has not been any follow-up work done capitalizing on these ideas, which is in part is what motivated this thesis.

3.3 Lazy vs Strict Evaluation

Before finally moving onto Haskell, there is a final point that is worth touching on, namely the evaluation model of the language. Programming languages are commonly split between two major evaluation models, namely lazy or strict (eager) evaluation. Let's take a brief look at what this means for a language. Let us first define a function, in lst. 3.6 so that we can compare how it evaluates under the two models.

Listing 3.6 Function for demonstrating evaluation models

```
add a b = a + b + a

callAdd = add (2+3) (4*3)
```

Most languages implement strict evaluation, usually call-by-value, which is the simpler model of the two evaluation models. In a strict language, calling `callAdd` will go through the sequence shown in lst. 3.7 during evaluation.

Listing 3.7 Strict evaluation sequence

```
add (2+3) (4*3)
add 5 (4*3)
add 5 12
5 + 12 + 5
23 <-- final evaluation
```

We see that arguments are evaluated before being passed into the function. Some arguments will be passed as references, commonly arrays and dictionaries, in some language that mixes in call-by-reference. That said, it is not a requirement from strict evaluation at all. Now if we contrast this with lazy evaluation, we end up with a sequence as shown in lst. 3.8.

The lazy evaluation leaves us with a pointer, called a thunk, which points to the computation. Only when we actually need the value inside

Listing 3.8 Lazy evaluation sequence (call-by-need)

```
add (2+3) (4*3)
a:(2+3) + (4*3) + a <-- final evaluation, `a` is a pointer
                        to the thunk of the first `a:...`
```

Listing 3.9 Lazy evaluation sequence (call-by-name)

```
add (2+3) (4*3)
(2+3) + (4*3) + (2+3) <-- final evaluation (for now)
```

the computation, say if we later decide to `print` it, do we perform the actual evaluation. Typically this is implemented as call-by-need, which means that it only evaluates the computation when it needs it, and additionally, memoizes the value so it does not need to reevaluate it on subsequent calls (in contrast to call-by-name, which evaluates the computation every time).

One thing to note about the use of thunks/laziness is that it inherently makes reasoning about performance harder, particularly reasoning about space, since thunks can build up and consume additional memory (i.e. space leaks). There are common strategies to fix this problem, such as marking a data type as strict, or forcing the evaluation to normal form.

Finally, some interesting features we get with laziness/non-strict evaluation are infinite lists, recursive definitions and much more. For example, in lst. 3.10 we see an example of an infinite list being used. This does not cause an endless loop, because the `take` function only needs to evaluate the list to the first five elements.

Listing 3.10 Taking five elements from an infinite list

```
Prelude> let infList = [1,2..]
Prelude> take 5 infList
[1,2,3,4,5]
```

3.4 Primer on Haskell

With our theoretical background on some of the core concepts of purely functional programming languages, we can move on to a specific one in the category, namely Haskell. Haskell is a lazy-evaluated purely functional programming language, that was a unification of several research projects and attempts at lazily-evaluated languages, made by the Haskell working group in around 1990.

Haskell is strongly statically typed, and features a type system with type inference. This means that type annotations are not required, except to resolve ambiguity for the type system when multiple typing judgements are valid. That said, it is still common practice to annotate functions with type signatures, as they very much help documenting the code.

First, let us take a look at a simple Haskell function, as shown in lst. 3.15.

Listing 3.11 Dissection of a Haskell function

```
add :: Int -> Int -> Int
add a b = a + b
```

The first line, `add :: Int -> Int -> Int`, is the type signature. Since Haskell is based on the lambda-calculus, it uses a concept called currying to allow multiple arguments into a function, since a lambda-expression only really takes one argument in. While in lambda-calculus we might write $\lambda a.\lambda b.a + b$, we see this concept mostly in the type signature with λa being the first `Int`, λb being the second, and finally the returned value `a+b` is represented by the third and final `Int`.

On the second line, `add a b = a + b`, we have the function definition, which declares a function, `add`, which takes two arguments, `a` and `b`, and then in the function body it returns `a + b`. In Haskell, the last expression in the function body is the one that is returned by the function.

Data Types

Moving on, an essential part of Haskell are the types, which brings us to the three different ways of specifying types, `type` which simply aliases something, `data` which creates a new data type along with its constructors, and `newtype` which is a special form of a data type that only exist at compile time and is erased at runtime. `newtype` is often used to wrap existing types, so that one can describe different behaviour for them. For example, if you wanted to express an *email* and a *username*, but both are `String`s, you can use a newtype such in lst. 3.12.

Listing 3.12 Using newtypes to distinguish between an email and a username

```
newtype Email = Email String
newtype Username = Username String

readEmail :: Email -> IO ()
readEmail (Email email) = print email

...
```

You are now able to force any consumer of your program API to be aware if they are supplying an email or username, instead of simply accepting any form of stringly typed value.

Data types come in handy when we want to express multiple states, options or similar, such as a traffic light, as shown in lst. 3.13.

Listing 3.13 Using data types to operate a traffic light

```
data TrafficLight = Red | Yellow | Green

trafficLight :: TrafficLight -> IO ()
trafficLight Red = print "Stop!"
trafficLight Yellow = print "Are you ready?"
trafficLight Green = print "Go!"
```

Typeclasses

With this we can introduce typeclasses, which are a way of both overloading operators and function names, and providing abstractions. A typeclass, much like an interface in Java, specifies the functions which any instance of it has to implement. This gives us ad-hoc polymorphism. For example, in lst. 3.14 we define the typeclass `Printable`, and an instance thereof.

Listing 3.14 The `Printable` typeclass and an instance of it

```
class Printable a where
  printable :: a -> String

instance Printable Int where
  printable i = show i

printing :: Printable s => s -> IO ()
printing s = print $ printable s
```

The example is perhaps a bit contrived; our instance just defines a way for an `Int` to be converted to a `String`, but we can now talk about functions that are constrained to types that have an instance of `Printable`, as shown in the last part of lst. 3.14. In the type signature, `printing :: Printable s => s -> IO ()`, we note the `Printable s`

`=>` part, which means that all type variables, `s`, need to be instances of the `Printable` typeclass.

Some commonly used typeclasses are `Show` for converting items into a `String`, `Num` to operate on numbers, `Eq` for equality (i.e. if you need to use `==` on something, it needs the `Eq` constraint) and `Ord` for ordering (i.e. `<`, `>=`, etc). One important feature of Haskell is that we can derive these typeclasses from the data type itself, meaning we do not have to constantly be writing instances for `Eq`, `Ord` or `Show`. This is done by adding `deriving (Show, Eq)` at the end of the data type declaration.

Monads

Quickly moving on, Haskell introduces a syntactic sugar for sequencing actions via *monads*, called *do-notation*, demonstrated in lst. 3.15. Monads are essentially just a typeclass, as we'll see in lst. 3.16, where do-notation uses the `>>=` operator, called *bind*, to sequence actions, `return`/`pure` to put something into a monad, and `join` to collapse a monad from e.g. `m (m value)` to `m value`, where `m` would be a type variable for a monad.

Listing 3.15 Introducing do-notation

```haskell
tellTheWorld :: IO ()
tellTheWorld = do
  print "Hey! What's your name?"
  name <- getLine
  print $ "Hello " ++ name

-- The above do-notation desugars to:
tellTheWorld =
  print "Hey! What's your name?" >>
    getLine >>= (\name -> print ("Hello " ++ name))
```

So a couple of concepts were introduced here. First we have the type signature, `tellTheWorld :: IO ()`, telling us we are performing

an action in the `IO` monad. Then we use the `<-` operator, inside the do block, to "pull out" a monadic value from `getLine`, which has the type signature `IO String`, giving us a `String` in `name`. Then we finally use that value by concatenating the string `"Hello "` with the value of the same type, `name`. The `$` can be read as "parentheses to the end of the line", meaning we could have written `print ("Hello " ++ name)` instead.

We wont delve too much into `Monad`s, `Applicative`s or `Functor`s, other than to note that they are a very common abstraction in Haskell. Their typeclasses are shown in lst. 3.16 for reference.

Listing 3.16 The `Functor`, `Applicative` and `Monad` type class definitions

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

class Functor f => Applicative f where
    pure    :: a -> f a
    (<*>)   :: f (a -> b) -> f a -> f b
    (*>)    :: f a -> f b -> f b
    u *> v  = …
    (<*)    :: f a -> f b -> f a
    u <* v  = …

class Applicative m => Monad m where
    (>>=)   :: m a -> (a -> m b) -> m b
    return  :: a -> m a
    return  =  pure
    (>>)    :: m a -> m b -> m b
    m >> k  =  m >>= \_ -> k
```

Some common monads, besides the `IO` monad, are `Reader`, `Writer` and `State`, with the last one mimicking stateful operations. Each of these expose an interface to the user, and for all intents and purposes one just needs to understand the monads API to use them, and not the

underlying abstract mathematical concept, which comes from category theory.

For example, the `State` monad exposes `get` and `put` for getting the monad value and putting/updating it respectively.

Language Extensions

The GHC compiler is very often a target for research, and as such gets a lot of experimental new features implemented in the language as language extensions. These are either enabled using pragmas, or by adding a flag in the project settings. Take for example a common extension `OverloadedStrings`, which provides a way for string literals to be parametrically polymorphic, by allowing anything that implements `IsString` to be a string literal, as shown in lst. 3.17.

Listing 3.17 Using the `OverloadedStrings` language extension

```haskell
{-# LANGUAGE OverloadedStrings #-}
import Text
import ByteString

normalString :: String
normalString = "A normal string"

textString :: Text
textString = "A performant unicode (utf-16) string"

byteString :: ByteString
byteString = "A fast byte implementation of strings"
```

This makes it convenient to work with other string types, since `String` is not the most performant, seeing as it is implemented using linked-lists. There are many of other language extensions too, that add to the syntax, add features to the type system and many other things (`FlexibleInstances`, `GADTs`, `RankNTypes`, etc).

Pragmas

Pragmas are a way to instruct the GHC compiler to do something special for certain code. For example, you can tell GHC to force inline a function by adding `{-# INLINE function_name #-}` (or not to via `NOINLINE`). There are a many pragmas (language extensions are also pragmas), such as `SPECIALIZE` which allows you to specialize a function to a specific type, etc.

One of particular interest is `RULES` which allow you to specify rewrite rules for code that match the Left-hand side (LHS) of the rule with the code on the Right-hand side (RHS) of the rule. Take for example the canonical rewrite rule, shown in lst. 3.18, which states that if you have a `map` that applies a function, `f`, to the result of another `map` that applies the function, `g`, to a list, `xs`, is equivalent to applying `f . g` (the `.` operator composes two functions together) directly to the elements in `xs`. This means that instead of two list traversals, we now just have one.

Listing 3.18 Rewriting two `map`s into one, using rewrite rules

```
{-# RULES
  "map/map" forall f g xs.  map f (map g xs) = map (f . g) xs
    #-}
```

Rewrite rules are applied until no rewrites are left, and they can be specified by the user [GHC Team 2017].

GADTs

Finally, we are going to touch on the subject of GADTs which are a generalization of the data types we saw earlier, allowing us to annotate the constructors with types. GADTs are one of the more advanced features of Haskell. First, let us look at a motivation for why we even need them, by constructing a `Program`, as shown in lst. 3.19.

This however leaves us with a problem: the `Equality` constructor is clearly expecting a `Boolean`, while the `Addition` constructor is expecting a `Value`. When we later have to implement our evaluator for this

Listing 3.19 A program as a data type

```
data Program
  = Value Int
  | Boolean Bool
  | Addition Program Program
  | Equality Program Program
```

program, we will run into trouble with this. But what if we could specify the return and input types of the program? In comes GADTs to the rescue, as shown in lst. 3.20.

Listing 3.20 A GADT to describe a program

```
data Program a
  = Value    :: Int -> Program Int
  | Boolean  :: Bool -> Program Bool
  | Addition :: Program Int -> Program Int -> Program Int
  | Equality :: Program Boolean -> Program Boolean
                                -> Program Boolean
```

While there are many more interesting extensions and concepts in Haskell, this should provide the bare necessities for understanding the rest of the thesis—after all, this is not meant to be a tutorial on programming languages. For a more in-depth overview of Haskell, there are multiple resources online, such as the *Haskell Wiki Book*[1], *Learn You a Haskell For Great Good*[2], and many more.

[1] https://en.wikibooks.org/wiki/Haskell
[2] http://learnyouahaskell.com/chapters

3.5 Summary

A purely functional programming language is language that creates a separation between effectful and non-effectful code. It contains the concept, at the type level, of purity and side-effects. This means that one can guarantee that a function, given the same input, will always return the same output—in other words, we gain referential transparency.

There are two main evaluation models in functional programming languages: lazy and strict. Most language opt to go for strict evaluation, because it is by far the simplest to implement, but there are some that go for the lazy evaluation model—or rather, call-by-need—namely Haskell.

Haskell is a lazy purely functional programming language, with an advanced type system, providing strong static guarantees about our programs, and giving us a way to model a lot of our logic in the type system, so as to let the compiler help catch errors for the programmer. It gives us these tools via data types, GADTs, ad-hoc polymorphism using typeclasses, monads and other powerful abstractions.

CHAPTER 4

Approaches to Offloading

With the theoretical background in place, we can begin to investigate what our possibilities are for a possible implementation of a system for offloading of computations. In this chapter we will take a look at five different proposals for handling offloading, along with their pros and cons, and finally sum them up in tbl. 4.1. Each approach will be evaluated based on the following parameters:

- **C**omplexity of the implementation.
- **A**doptability by the wider community.
- **B**uy-in, for a developer to use the system.
- **G**ranularity of the offloading mechanism.
- **S**erver-side story.
- **P**ortability to other pure functional programming languages.

For reference in the rest of the chapter, an overview of the GHC system is presented in fig. 4.1. GHC and Haskell are used interchangeably throughout, seeing as GHC is considered to be the de facto compiler for Haskell.

We see that besides the compiled Haskell code itself, there is a supporting runtime, along with linking to libraries and a C code Foreign Function Interface (FFI). All of this of course sits on top of the Operating System (OS) and underlying hardware.

4.1 Extending the Runtime

The first immediate thought is to go the way of many previous attempts, and aim to automate as much as possible so the programmer would not

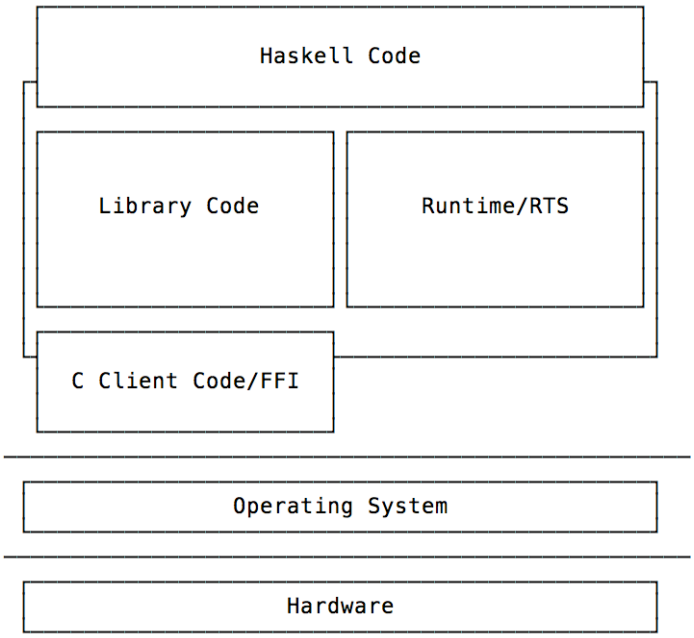

Figure 4.1: GHC overview

have to worry about whether or not they even want to offload a function, but simply have the runtime decide based on the current conditions. Indeed, as a user of the system, this would provide the smoothest way to get started, initially, but it also is the most complex and invasive approach we could choose.

The Glascow Haskell Compiler (GHC) runtime is a massive codebase written in C, clocking in at around 50.000 lines of code[Brown and Wilson 2012], and is responsible for a number of things, such as:

- Memory management (i.e. garbage collection)
- Thread management and scheduling
- Primitive operations provided by GHC
  - Function calls
  - Exceptions
  - Allocating arrays

  - Handling `MVar`s
  - ...
- Profiling (heap-profiling, time-profiling)
- Support for Software Transactional Memory (STM)
- Bytecode interpreter and dynamic linker for GHC Interpreter (GHCi) (the REPL)

An overview of the runtime can roughly be given as shown in fig. 4.2.

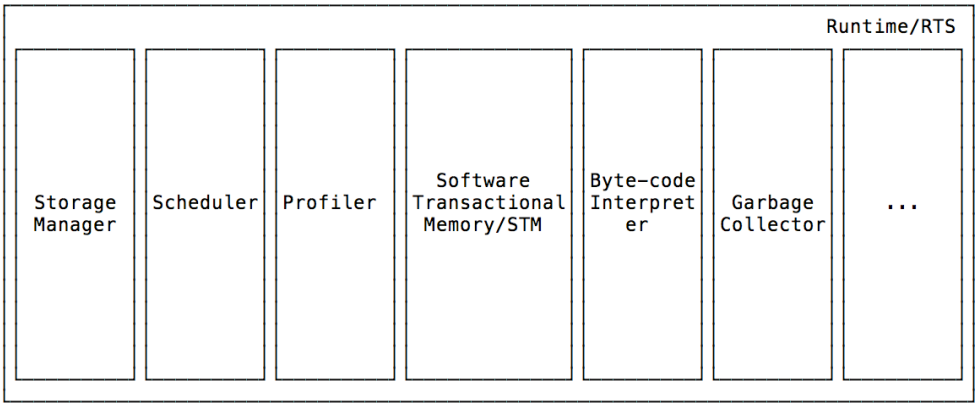

Figure 4.2: GHCs Runtime System

One can quickly see that extending the runtime is a daunting task. For example, taking over function calls would involve:

- Making sure that we do not mess with the Garbage Collector (GC),
- adding support in the profiler and scheduler,
- making sure we do not affect STM, and
- supporting byte-code generation of the changed functions.

Since the idea is to put the granularity of offloading at the function level—and only pure functions at that, since we cannot know if any functions in `IO` are offloadable—it makes sense to check out how the GHC runtime handles function calls, which we can get by checking out the commentary on the GHC compiler, specifically for its runtime[GHC Commentary].

We see that the GHC Runtime System (RTS) divides function calls into four different categories (see lst. 4.1 for the definitions of the functions mentioned in the list):

- Unknown functions, i.e. the argument `f`, in `map`, is unknown at compile time and can come from many places.
- Known functions with too few arguments, such as `tooFewMap`, which is missing one argument.
- Known functions that are saturated, such as `saturatedMap`.
- Known functions with too many arguments, such as `compose`, which takes in two functions, as if it was a saturated call, composes them and returns a new function which will behave as an unknown function.

Listing 4.1 Examples of the four different categories of function calls

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

tooFewMap :: Num a => [a] -> [a]
tooFewMap = map (\x -> x + 2)

saturatedMap :: Num a => [a]
saturatedMap = map (\x -> x + 2) [1,2,3,4]

compose :: (b -> c) -> (a -> b) -> (a -> c)
compose f g = f . g
```

From the original paper [Marlow and Jones 2006] discussing the choice of using an *eval/apply* evaluation model rather than *push/enter*, we get some estimates with regard to how many calls are actually to unknown functions, stating that on average the figure is around 20%. Conversely this means that 80% of calls should be to known functions, and we can therefore argue that an implementation should only look into supporting

offloading for known function calls, which would cut down a little on the surface area of the implementation.

While this may still seem like too coarse a granularity, one could eventually add support for pragmas that would mark a function as not offloadable (and offloadable for `IO` functions), as done in MAUI. This would entail propagating this information around the runtime.

As such, we can conclude that extending the runtime is first and foremost a very complex and huge undertaking. Furthermore it would require quite the buy-in from the users, since they would have to use a custom version of the GHC compiler, seeing as it would probably be hard (and would take years due to the feature cycle of GHC) to get it into the main GHC implementation. This will also greatly hurt adoptability—not to mention portability—and the custom GHC compiler would have to play catch up to the main implementation. Additionally, we do not have a server-side story, since this would only enable the offloading mechanism, but nothing in regards to how the server should be implemented.

A final note is that it would also involve implementing the whole system in C, which sort of defeats the purpose of exploring what we can use purely functional programming languages for.

4.2 Using the `unsafePerformIO` Escape Hatch

Haskell is a very high-level language, and as such abstracts a lot of the fundamental details of the underlying hardware away. Sometimes though, one is forced to deal with lower-level details to implement fast primitives that can later be exposed in a higher-level manner. As such, there are several escape hatches in Haskell. To name a few:

- `unsafePerformIO` is the main "back door" into the `IO` monad, and can circumvent the type system, meaning you could perform it inside a pure function.

- `unsafeDupablePerformIO` is a more efficient version of `unsafePerformIO` that lacks thread safety.
- `unsafeInterleaveIO` makes it possible to defer `IO` computations lazily, and as such is used to implement lazy file reading, for example.

The idea is to use `unsafePerformIO` inside function definitions, to decide whether to offload them or not. By itself this would require manually sprinkling these calls around, but in combination with one of the other approaches, such as Rewrite Rules sec. 4.3 or Template Haskell sec. 4.5, this could be made feasible in an ergonomic way for the developer to use.

Now, one might ask "If you need to manually insert calls, why not use a regular function instead of `unsafePerformIO`?" which is indeed a valid question. The reasoning behind this is that we would like as much as possible to force functions into the `IO` monad, which did not need to be there in the first place, as it breaks down our ability to reason about our code (w.r.t. referential transparency), and means we would have to use monadic do-notation everywhere, making writing code quite tedious.

`unsafePerformIO` comes with some assumptions for it to work safely, such as not relying on any specific order in which the side-effects, in the function using `unsafePerformIO`, will be performed. That is, it is indeterministic, because of the call-by-need semantics of Haskell, which neither guarantees a function to ever be fully evaluated to normal form, nor that it will be evaluated at the place it is encountered. Furthermore, for the side-effects to not be optimized away, there are a few precautions, such as making sure the function is not inlined (via `{-# NOINLINE foo #-}` pragma), the expression is not eliminated (via the compiler flag `-fno-cse`) and making sure the call to `unsafePerformIO` is not floated outside a lambda.

To address these assumptions, we mainly need to consider the first one, because the second one is mainly an implementation detail. Luckily we do not care about neither the order of evaluation nor if it ever gets evaluated (in fact, that is only a good thing), because the inherent goal of offloading is to save the device from executing code. This means that Haskell's call-by-need semantics is very well suited for the goal of

offloading.

A simplified example is shown in lst. 4.2, as a rough sketch of the idea, with `shouldOffload` and `offloadFunction` just being stubs to satisfy the type system for now.

Listing 4.2 Rough sketch for an offload function using `unsafePerformIO`

```
import System.IO.Unsafe

-- | Stub for `shouldOffload`.
shouldOffload :: IO Bool
shouldOffload = pure True

-- | Stub for `offload`.
offload :: String -> (a -> b) -> a -> IO b
offload name f a = do
  -- Offload the function to the server,
  print $ "Offloading function " ++ name ++ "..."
  -- or support falling back to running it locally.
  pure $ f a

offloadFunction :: String -> (a -> b) -> a -> b
{-# NOINLINE offloadFunction #-}
offloadFunction name f a =
  if unsafePerformIO shouldOffload
    then unsafePerformIO $ offload name f a
    else f a
```

The code does the following:

- The `offloadFunction` function simply takes in a function name, a function and an argument that should be applied,
- calls `shouldOffload` containing the network state which returns `True` if it makes sense to offload and `False` if not,

- and then either returns:
  - a call to `unsafePerformIO`, with `offload` taking care of performing the actual network calls to offload, which needs to be synchronous/blocking, or
  - the `f a` function itself, if it should not have offloaded.

Admittedly a lot of implementation details are left out, but the underlying concept should still be present in the code. This implementation would allow us to do something like `offloadFunction "heavyComputation" heavyComputation input`, which will then take care of either offloading the function, getting back its input and returning that, or simply returning the function application as would happen in normal code.

One thing to note, with GHC, currently there is no way to serialize or get the name of a function at runtime, which we would need to be able to tell the server exactly what code it needs to run (i.e. what function to call). This gives us a bit of an awkward interface to the function, as shown in lst. 4.3.

Listing 4.3 Running code using `offloadFunction`

```
main :: IO ()
main = do
  print "Start:"
  print $ offloadFunction "simpleFunction" simpleFunction 3
  print "End!"

simpleFunction :: Int -> [Int]
simpleFunction a = map (+a) $ map (+2) [1,2,3]
```

An example of running the code, shown in lst. 4.3, will yield the output:

```
"Start:"
"Offloading function..."
[6,7,8]
```

```
"End!"
```

The complexity would not be that high, and the buy-in would be low, but the adoptability will probably not be the greatest seeing as `unsafePerformIO` introduces quite a lot of uncertainty in the program, which is generally to be avoided if possible. The implementation would need to be battle-tested for people to be willing to trust it at least. With regards to granularity, it is very fine-grained, perhaps too much; function calls need to be manually placed at places that might benefit from offloading. Finally, there is yet again no server-side story. Finally, portability is a bit up in the air, since `unsafePerformIO` is Haskell specific, but it could very well be implemented using similar constructs if a language supports any form of escape-hatch into IO—for example `unsafePerformEff`[1] in PureScript or `unsafePerformIO` in Idris, although both languages are strict, so the evaluation will differ a bit.

4.3 Rewrite Rules

With GHCs support for rewriting functions using rewrite rules, one could imagine coming up with a set of rules to rewrite common functions into ones that are identical, except for additional logic to offload the function itself. For example, combining the approach from sec. 4.2, using `unsafePerformIO`, one could keep the signature of the functions, while discretely adding the additional logic.

One problem with this approach however is that we can only target specific functions with our rewrite rules. For example, as shown in lst. 4.4, we can target specific functions, in this case the function `simpleFunction`, which will then get rewritten from `simpleFunction x` into `offloadFunction simpleFunction x`.

We can check that the rule has fired by running `stack exec -- ghc`

[1] https://github.com/purescript/purescript-eff/blob/master/src/Control/Monad/Eff/Unsafe.purs

Listing 4.4 Rewrite rule for `simpleFunction` to wrap it in `offloadFunction`

```
main :: IO ()
main = print $ simpleFunction 3

{-# RULES
"simpleFunction/offload simpleFunction" forall x.
    simpleFunction x = offloadFunction "simpleFunction"
                                       simpleFunction x
  #-}

simpleFunction :: Int -> [Int]
simpleFunction a = map (+a) $ map (+2) [1,2,3]
```

`Main.hs -O2 -ddump-rule-firings` (or omit the `stack exec --` part if you are running plain GHC), which yields the output in lst. 4.5.

Listing 4.5 Dump of the fired rewrite rules

```
...
Rule fired: map (GHC.Base)
Rule fired: unpack (GHC.Base)
Rule fired: Class op >>= (BUILTIN)
Rule fired: Class op pure (BUILTIN)
Rule fired: simpleFunction/offload simpleFunction (Main) <-- Our rule
Rule fired: Class op show (BUILTIN)
Rule fired: unpack (GHC.Base)
Rule fired: unpack-list (GHC.Base)
...
```

Running the program, combined with the code from lst. 4.2, gives us the following output:

```
"Offloading function..."
[6,7,8]
```

Note: the reason we defined `simpleFunction` as `map (+a) $ map (+2) [1,2,3]` was to show that multiple rewrites will take place, since `map f (map g xs) == map (f . g) xs`.

Adding calls to offload functions is indeed doable using rewrite rules, and also alleviates some of the quirky interface to `offloadFunction`. However it is a very brittle approach, since inlining will cause the rule not to fire, and it is also quite tedious, since every function one wants to offload needs a rewrite rule. As for complexity, it adds little to no real complexity on top of the `unsafePerformIO` approach from sec. 4.2. Because of the brittleness, it would most likely not see widespread adoption, even though the buy-in is very small. It is also a not portable, e.g. PureScript still does not have a concrete proposal for rewrites[2].

4.4 Monadic Framework

An approach less radical than the others so far is to utilize the structure of the program itself to enable separating the interpretation/implementation of the program—and thereby the effects—from the semantics of the program itself. This can be done thanks to the way monadic computations are done, by forming a sequence of actions to be performed. This effectively allows us, as shown in fig. 4.3, to have multiple ways of interpreting the same program, depending on what we want from it. For example, we could have:

- A pure interpreter for testing.
- An interpreter that stores the program output as a list of strings and inspect it later on.
- A debug interpreter that allows us to step through the program sequences.
- An effectful interpreter running the actual program.

There are two popular styles of writing larger applications in Haskell. The first one, and by far most popular, is the Monad Transformer Library

[2]https://github.com/purescript/purescript/issues/2749

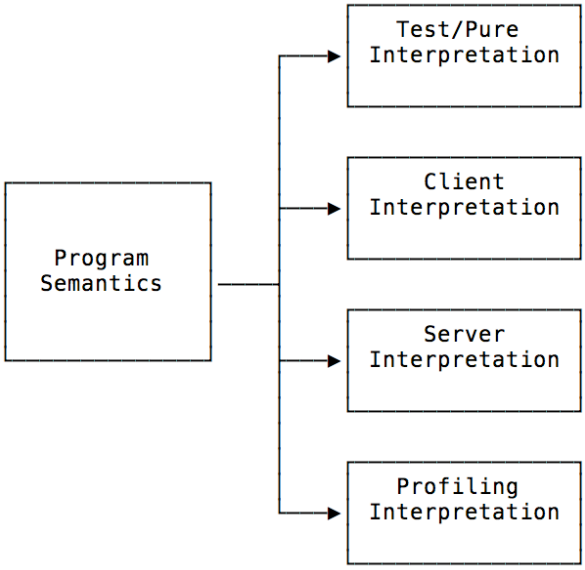

Figure 4.3: Multiple interpreters from a single program

(MTL)-style, and the second is using a concept known as `Free` monads. Let us take a look at the MTL-style of structuring programs first.

MTL-style

Monad transformers came about because of a natural limitations of monads, namely that they do not compose. One way to mitigate this is then to have a set of transformers that know how to go from one specific monad instance to another.

For example, from the MTL we have the `Reader` monad as `MonadReader` with the interface for the monadic operations it support, and then instances for each monad it supports, including its own base case, `ReaderT`, as shown in lst. 4.6.

We see that in the first instance—the base case—it simply uses the `ask` operation from `ReaderT` directly, but in the second instance, where `MonadReader` is wrapping `StateT`, it needs to `lift` the `ask` operation once, because the ask operation will be called from inside the `StateT`

Listing 4.6 The `MonadReader` class and its instances for `ReaderT` and `StateT`

```
class MonadReader r m | m -> r where
  ask :: m r

instance Monad m => MonadReader r (ReaderT r m) where
  ask = Control.Monad.Trans.ReaderT.ask
instance (MonadReader r m) => MonadReader r (StateT s) where
  ask = lift ask
```

monad, and therefore needs to bubble one level up. We now have a general way of composing these transformers, at the expense of writing boilerplate code for the monads that we want to compose with. In fact, for every monad instance you add, you would need n^2 instances (at least if you want full composition). For example, to support `MonadReader` and `MonadState`, we need a base case for each and then an instance for `MonadState` supporting `ReaderT` and one for `MonadReader` supporting `StateT`. This can quickly grow, so we are getting this flexibility at the expense of setting up some boilerplate.

Going back to the task at hand, what we really want is a way to create operations—without specifying the type of effects—that can be abstracted over, and later on a concrete type can be chosen, depending on the place we want to use the program, be it client-side, testing or server-side. The way we would do this in the MTL-style, is to have our operations as typeclass methods, as shown in lst. 4.7, collected in the typeclass `MonadEffects`.

`MonadEffects` becomes our general program interface, exposing some of the operations we can exercise precise control over, depending on the environment they are running in. We fill in a bit of boilerplate by making some default implementations, marked by `default`, of each operations, so that we can derive a series of MTL typeclasses in few lines, as shown in lst. 4.8.

We can now provide two wrapper types around `IO`, also called carrier types, which will serve to make a distinction between our `MonadEffects`

Listing 4.7 Defining our operations with `MonadEffects`

```
class (Monad m) => MonadEffects m where
  writeOutput :: Show a => a -> m ()
  getInput :: m String
  computation :: Int -> Int -> m Int
  -- Provide a default implementation for empty instances.
  default writeOutput :: (MonadTrans t, MonadEffects m', m ~ t m'
                       , Show a) => a -> m ()
  writeOutput = lift . writeOutput
  default getInput :: (MonadTrans t, MonadEffects m', m ~ t m')
                      => m String
  getInput = lift getInput
  default computation :: (MonadTrans t, MonadEffects m', m ~ t m')
                         => Int -> Int -> m Int
  computation i1 i2 = lift $ computation i1 i2
```

Listing 4.8 Default instances of `MonadEffects` to make it compatible with common MTL classes

```
instance MonadEffects m => MonadEffects (LoggingT m)
instance MonadEffects m => MonadEffects (ReaderT r m)
instance MonadEffects m => MonadEffects (StateT s m)
instance (MonadEffects m, Monoid w) => MonadEffects (WriterT w m)
```

instance running on the client, and the one running on the server. We do this, as shown in lst. 4.9, by making `newtype` wrappers around `IO`, providing a `runX` to unwrap it, and deriving some common classes.

Before we can run the program, we need to make some concrete implementations of `MonadEffects` for each of these two carriers. We will do this in a fairly straightforward way, as shown in lst. 4.10.

Now we can finally write our program, which we can interpret both on the client and server side, as shown in lst. 4.11, complimented with a `Reader` to show that our boilerplate for the MTL classes.

Listing 4.9 Our carrier types for the client and server respectively

```
-- Our client IO instances.
newtype Client m a = Client { runClient :: m a }
  deriving (Functor, Applicative, Monad, MonadIO)

-- Our server IO instances.
newtype Server m a = Server { runServer :: m a }
  deriving (Functor, Applicative, Monad, MonadIO)
```

Listing 4.10 The concrete instances for our `Client` and `Server`

```
instance MonadEffects (Client IO) where
  getInput = Client $ pure "Fake Input"
  writeOutput = Client . print
  computation i1 i2 = do
    liftIO $ print "Offloading"
    Client . pure $ i1 + i2

instance MonadEffects (Server IO) where
  getInput = Server $ pure "Fake Input"
  writeOutput = Server . print
  computation i1 i2 = Server . pure $ i1 + i2
```

With this, we should have demonstrated how to structure a program, along with separating the execution from the semantics. Still, the separation does not feel as clean as we could wish for, and there is a lot of boilerplate involved (even though we cut down on this extensively). This actually brings us onto our next approach: the `Free` monad.

Free Monads

The `Free`-style of writing a program allows for a very clean separation of the semantics of the program and the interpretation of it, by first

Listing 4.11 Our program running both the client and server instances

```
program :: (MonadEffects m, MonadReader Env m) => m ()
program = do
  env <- ask
  inp <- getInput
  writeOutput $ "Input: " ++ inp ++ ", Hostname: " ++ envHost env
  res <- computation 12 22
  writeOutput res

data Env = Env { envHost :: String }

main = do
  runClient . runReaderT program $ Env { envHost = "localhost" }
  runServer . runReaderT program $ Env { envHost = "remote" }
```

structuring a form of Abstract Syntax Tree (AST) of the program, and then constructing different interpreters depending on how you want to run the program.

`Free` monads are a concept from category theory, which when used in Haskell gives us a monad for free by extending our data types, as long as we implement the functors for the data type ourselves. If you squint a bit, the data type for `Free`, shown in lst. 4.12, looks a lot like the data type for list, and in fact it is a fair intuition of how the structure of a program written in `Free`-style is built up.

To construct a program using `Free` monads, one first starts with the data type that states which instructions our program can perform. Let us make a small program that can read files, get user input, write output and perform a computation. We construct a data type, called `Effects`, for this in lst. 4.13.

Our data type parameter, `next`, is the continuation that will allow us to go through the program. The first parameter to `ReadFile` is the filename, meaning an argument to `ReadFile`, while the second argument is the output into the continuation, `(String -> next)`, meaning it outputs a `String` to the next step. Now we can define our `Functor` instance

Listing 4.12 Definition of `Free` compare with the definition of `List`

```
data Free f r
  = Pure r
  | Free (f (Free f r))

data List a
  = Nil
  | Cons a (List a)
```

Listing 4.13 Our `Effects` data type that models our program

```
data Effects next
  = ReadFile String (String -> next)
  | WriteOutput String next
  | WriteOutputInt Int next
  | GetInput (String -> next)
  | Computation Int Int (Int -> next)
```

for our data type, as defined in lst. 4.14. We need this, because `Free` only allows us to recover the monad by assuming the functor operations.

Listing 4.14 The `Functor` instance for our `Effects` data type

```
instance Functor Effects where
  fmap f (ReadFile s g) = ReadFile s (f . g)
  fmap f (WriteOutput s g) = WriteOutput s (f g)
  fmap f (WriteOutputInt i g) = WriteOutputInt i (f g)
  fmap f (GetInput g) = GetInput (f . g)
  fmap f (Computation i1 i2 g) = Computation i1 i2 (f . g)
```

This is fairly straight forward: the `f` of the `fmap` either gets composed with the continuation of the data type (e.g. `f . g`) if there are arguments to the continuation, or applies `f g` if the continuation is a plain `next`.

The next step is to setup our functions that lift our functors into the `Free` monad, along with creating a type alias for our `Free Effects` program, as shown in lst. 4.15.

Listing 4.15 Functions lifting the functors into the `Program` monad

```
type Program = Free Effects

readFile :: String -> Program String
readFile s = liftF $ ReadFile s id

writeOutput :: String -> Program ()
writeOutput s = liftF $ WriteOutput s ()

writeOutputInt :: Int -> Program ()
writeOutputInt i = liftF $ WriteOutputInt i ()

getInput :: Program String
getInput = liftF $ GetInput id

computation :: Int -> Int -> Program Int
computation i1 i2 = liftF $ Computation i1 i2 id
```

We are now ready to write an interpreter for our program. Let us start with a test interpreter, in lst. 4.16, that runs the whole program, simulating the input and printing the output.

The final step is to construct a program, using the functions defined in lst. 4.15, and then apply the interpreter, from lst. 4.16, on the program. We will do both at the same time in lst. 4.17.

Which yields the output:

```
Test file content for: Fake input
34
```

Some quick notes, since creating functors and making the functions to lift our functors into monads are done so frequently, there is some

Listing 4.16 Interpreter for the `Free Effects` program

```
testInterpreter :: Program next -> IO next
testInterpreter (Pure a) = return a
testInterpreter (Free effect) =
  case effect of
    ReadFile filename next ->
      let fakeFileContent = "Test file content for: " ++ filename
      in testInterpreter $ next fakeFileContent
    WriteOutput s next -> putStrLn s >> testInterpreter next
    WriteOutputInt i next -> print i >> testInterpreter next
    GetInput next -> let fakeInput = "Fake input"
      in testInterpreter $ next fakeInput
    Computation i1 i2 next -> testInterpreter $ next (i1 + i2)
```

Listing 4.17 Running the `testInterpreter` on the program

```
program :: Program ()
program = do
 filename <- getInput
 contents <- readFile filename
 writeOutput contents
 result <- computation 12 22
 writeOutputInt result

main :: IO ()
main = testInterpreter program
```

machinery to make it a *lot* more ergonomic. We could replace the whole of lst. 4.14 with a `deriving (Functor)` on our data type and a `{-# LANGUAGE DeriveFunctor #-}` language pragma. Furthermore, by using Template Haskell, we could do away with lst. 4.15 (keeping the `Program` type alias) by using `makeFree ''Effects` along with a `{-# LANGUAGE TemplateHaskell #-}` language pragma at the top. This cuts down a

lot on the tedious an erroneous parts of setting up a program using the `Free`-style.

There is one problem with `Free` though: it has terrible performance. Constantly doing `(f >>= g) >>= h` behaves performance-wise as bad as `(a ++ b) ++ c`—that is, list concatenation. This brings us, in our next section, to an improvement over both the performance of `Free` and its interface, namely `Freer`.

Freer Monads

`Freer`, introduced in [Kiselyov and Ishii 2015], changes up the interface, but the overall idea is the same as `Free`—to create a clean separation between the program semantics and the implementation details.

The first difference comes from our data type, which is now a GADT, as shown in lst. 4.18. This also gives us the power to use constraints on our types, letting us do away with the two `writeOutput` functions, and simply require the argument to be `Show`able.

Listing 4.18 Our `Effect` GADT that models our program

```
data Effect n where
  ReadFile :: String -> Effect String
  WriteOutput :: Show a => a -> Effect ()
  GetInput :: Effect String
  Computation :: Int -> Int -> Effect Int
```

As we can see, we also get a more function-like syntax, and do away with our explicit continuations. Another thing to note is that `Freer` goes beyond `Free` and gives us not just `bind` and `join` for free, but also `Functor`, meaning we do not have to write or derive these instances anymore.

The next step is to model our functions, which are then ones we will use when writing our `Freer` programs. It follows much the same way as for `Free`, in that we define a function for each constructor in our GADT,

but other than that it is a quite different type signature and we also use a different function to lift it into `Freer`, as shown in lst. 4.19.

Listing 4.19 Functions lifting the GADT constructors into our effects

```
readFilename :: Member Effect effs => String -> Eff effs String
readFilename = send . ReadFilename

writeOutput :: (Member Effect effs, Show a) => a -> Eff effs ()
writeOutput = send . WriteOutput

getInput :: Member Effect effs => Eff effs String
getInput = send GetInput

computation :: Member Effect effs => Int -> Int -> Eff effs Int
computation i1 i2 = send $ Computation i1 i2
```

We are now ready to write our interpreter, as shown in lst. 4.21. Here, much like with `Free`, we simply pattern match on each of the type constructors from our `Effect` GADT, and perform the actions which the specific interpreter should take. A thing to note is the new syntax in the type signature, `'[Effect, IO]` (note the tick, `'`, in front of the list). This is a type-level list, and tells us that our effects consists of `Effect` and `IO`. We could add more here if we wanted to, making them composable.

We can also see, from lst. 4.21, that we do not have to manually end with the continuation to the next call of the interpreter anymore, removing some boilerplate code.

And finally we can run the interpreter on a program, as we have done in lst. 4.21.

Which yields the output:

```
Test file content for: Fake input
34
```

exactly like `Free`.

Listing 4.20 Interpreter for our `Freer` program

```
runEffect :: Eff '[Effect, IO] a -> IO a
runEffect = runM . interpretM (\e -> case e of
  ReadFilename filename -> pure $ "Test file content for: "
                                  ++ filename
  WriteOutput s -> print s
  GetInput -> pure "Fake input"
  Computation i1 i2 -> pure $ i1 + i2)
```

Listing 4.21 Running the `Freer` interpreter on our program

```
program :: Eff '[Effect, IO] ()
program = do
 filename <- getInput
 contents <- readFilename filename
 writeOutput contents
 result <- computation 12 22
 writeOutput result

main = runEffect program
```

We have now seen a way we can structure our program so that we can separate our semantics from our implementation, giving us great freedom in moving bits of the program execution around as we see fit. We can choose certain effects to be offloadable, and then have more or less the same interpreter on the server-side, just without the choice to offload. Another benefit is that data types are serializable, whereas functions are not. We have seen this problem arise for example in sec. 4.2, where we had to pass a `String` with the function name/id on, so that we could identify it on the server-side.

We gain a lot for very little real complexity, while also maintaining an approach that is both easily adoptable—and in fact already in use,

albeit for different goals—and also very portable to other languages, with sufficient type systems (like Idris or PureScript). We have a server-side story, and a flexible graining for choosing what to offload.

4.5 Template Haskell

Template Haskell (TH)—first introduced in [Sheard and Jones 2002] but greatly improved since— is a feature of GHC that adds meta-programming functionality to Haskell. TH is commonly used to automatically derive typeclass instances for you data types. For example, the Javascript Object Notation (JSON) parsing and encoding library, `aeson`[3], can create encoders and decoders, to and from JSON, simply based on your data type, by using TH. Another common usage is for Domain Specific Languages (DSLs) to be embedded inside Haskell code, or loading resources and files during compile time.

In short, TH allows us to generate code at compile time, so we perform actions that might not yet be supported by the host language itself, or simply cut down on boilerplate code. This functionality is hosted in the `Q` monad, which exposes all the functionality we need for doing our meta-programming via TH.

One way we could use this meta-programming feature, and make it work for us, is by utilizing it to generate the code we need for offloading our function, while also constructing the logic that we need for the server-side of the offloading equation to work. It can also make the interface a bit smoother, by letting TH handle getting the function name that the server-side needs, using reification.

An example of how this could be done is finely demonstrated in the `debug`[4] package, which wraps a function inside a quasiquoter, as shown in lst. 4.22, and then generates code to debug the function and view it in a web interface.

The `[d| ... |]` is quotation syntax for producing a declaration, and has the type `Q [Dec]`. There is also `[t| ... |]` giving `Q Type` for types,

[3] https://hackage.haskell.org/package/aeson
[4] https://hackage.haskell.org/package/debug

Listing 4.22 Example of the `debug` packages' quasiquoter (from its hackage documentation)

```
debug [d|
   quicksort :: Ord a => [a] -> [a]
   quicksort [] = []
   quicksort (x:xs) = quicksort lt ++ [x] ++ quicksort gt
       where (lt, gt) = partition (<= x) xs
   |]
```

`[p| ... |]` giving `Q Pat` for patterns and finally `[e| ... |]` (or simply `[| ... |]`) giving `Q Exp` for expressions. An example, shown in lst. 4.23, would be running our offload function call through the expressions quasiquoter, to generate the Template Haskell AST for us.

Listing 4.23 Generating a Template Haskell expressions in the REPL

```
*Main> runQ [e| offloadFunction "simpleFunction" simpleFunction 3 |]
AppE
  (AppE
    (AppE
      (VarE Main.offloadFunction)
      (LitE (StringL "simpleFunction"))
    )
    (VarE Simple.simpleFunction)
  )
  (LitE (IntegerL 3)
)
```

As we see, by generating code using TH, we are essentially manually constructing an AST and manipulating it towards our goal. This does give us great power, but at the same time allows us to shoot ourselves in the foot very easily.

So, let us set up some design goals for our TH system: we want it to,

- remove duplicate (and thereby error-prone) arguments when calling the offloading function, and
- help us with the server-side of things.

To do this, we could imagine making a TH function, that will simply take in the function and its arguments, add the `offloadFunction` and necessary additional arguments in front of it, and create a mapping stating what name it passed on to `offloadFunction` and which function this should call on the server-side.

We tackle the first goal by creating a function that will automatically lookup the name of the function, and construct the actual expression for `offloadFunction`. The TH code to derive this is shown in lst. 4.24.

Listing 4.24 Deriving the full offloading expression via TH

```
{-# LANGUAGE TemplateHaskell #-}
import Language.Haskell.TH
import Language.Haskell.TH.Syntax
import Offload (offloadFunction)

deriveOffload :: Name -> Q Exp
deriveOffload name =
  [e|
    offloadFunction n $a
    |]
  where
    a = varE name
    n = showName name
```

To explain what is going on: we take in the function as an argument, in the form of `'functionName`—notice the single ' tick in the beginning, which is a TH special syntax to pass a function as a `Name`—which we can then convert into a string with the name, via `showName name`, and also keep the original expression, via `varE name`. We then splice in the function (i.e. the original expression) using `$a`, inside our quasiquotation.

Alternatively, we can create our own quasiquote that uses `deriveOffload`, as shown in lst. 4.25.

Listing 4.25 Wrapping our `deriveOffload` inside a quasiquoter `[off|...|]`

```
import Data.List (dropWhileEnd, dropWhile)
import Data.Char (isSpace)
import Language.Haskell.TH.Quote  (QuasiQuoter(..))

off :: QuasiQuoter
off = QuasiQuoter
  { quoteExp = \n -> do
      let name = dropWhileEnd isSpace $ dropWhile isSpace n
      maybeName <- lookupValueName name
      case maybeName of
        Just name' -> deriveOffload name'
        Nothing -> fail $ "The function '" ++ name
                          ++ "' is either not in scope or"
                          ++ " does not exist"
  , quotePat = error "Doest not support using as pattern"
  , quoteType = error "Doest not support using as type"
  , quoteDec = error "Doest not support using as declaration"
  }
```

Our new functions, `deriveOffload` and `[off|...|]`, are then called, as shown in, lst. 4.26.

Currently we gain very little, other than removing the chance of giving the wrong function name as a `String` argument to `offloadFunction` (which would mean calling the wrong function on the server-side.) But we are not done yet! For this to actually give us enough benefit that we would choose this over the original syntax, we can generate the server-side endpoint for the derived code. Let us take a closer look at how we could do that.

We would like our server-side routing function to act as an entry point

Listing 4.26 Using `deriveOffload` and `[off|...|]`

```
{-# LANGUAGE TemplateHaskell, QuasiQuotes #-}
main :: IO ()
main = do
  -- Original.
  print $ offloadFunction "simpleFunction" simpleFunction 3
  -- Two new Template Haskell approaches.
  print $ $(deriveOffload 'simpleFunction) 3
  print $ [off|simpleFunction|] 3
```

for the server-side, and then have a mapping of `String`s to actual functions. If we could make sure a piece of TH code ran at the end, we could build up a list of function names to functions, but unfortunately we have no guarantee of order in the compilation. So, we need to divide up our compilation processes into a client compilation, which features the `deriveOffload` code, and a server compilation, which would construct the endpoint from the information made available from the finished client compilation. As such, one way to go about it is to generate a file consisting of all the mappings we need, and then have the server compilation read in this file and generate the Haskell code, via TH, that we need.

Let us first extend the `deriveOffload`, as shown in lst. 4.27, to also write out the mapping in a file, in the format of `functionString:function`, separating each mapping by a newline.

Listing 4.27 Extending `deriveOffload` to write the function mappings to a file

```
extendedDeriveOffload :: Name -> Q Exp
extendedDeriveOffload name = do
  let n = showName name
  runIO $ appendFile "FunctionMapping.txt" (n ++ ":" ++ n ++ "\n")
  [e| offloadFunction n $(varE name) |]
```

We can now use this file in a later compilation process, to gather all

the mappings and create endpoints that will call the specific functions. In lst. 4.28, we read in the file, remove duplicate mappings, and then generate a case for each mapping, point to its function.

Listing 4.28 Generating endpoints from the data in the "FunctionMapping.txt" file

```
deriveEndpoints :: String -> Q [Dec]
deriveEndpoints path = do
  let g (s:f:[]) = (LitP $ StringL s, VarE (mkName f))
  content <- runIO (readFile path)
  addDependentFile path -- Recompile on file change.
  lcBody <- [e|error "Undefined mapping"|]
  let mappings = map (splitOn ":") (lines content)
      clauses = zipWith
        (\body pat -> Clause [pat] (NormalB body) [])
        (map (snd . g) mappings) (map (fst . g) mappings)
      lastClause = [Clause [VarP (mkName "s")] (NormalB lcBody) []]
  pure [FunD (mkName "endpoint") (clauses ++ lastClause)]
```

In our main file, we call it with a simple `$(deriveEndpoints "FunctionMapping.txt")`.

There is one problem however: we are generating a function that redirects to other functions, and as such we need to have a uniform type signature. This quickly breaks down the program when you go beyond trivial cases, although we could alleviate some of this with some typeclass trickery and existential types.

Another way to solve this is to evaluate the code through an interpreter, such as `hint`[5], which allows us to supply a string that will then get evaluated. We could then get rid of the tedious server-side generation of code, replace it with the approach shown in lst. 4.29, and run the code in lst. 4.30, which outputs `"[6,7,8]"` (i.e. a `String` with the result).

[5] https://hackage.haskell.org/package/hint

Listing 4.29 Interpreting incoming code on the server-side

```
interpreterEndpoint :: (String, String) -> String
                    -> Interpreter String
interpreterEndpoint (_,f) arg = do
  let splitFn = splitOn "." f
      fnName = last splitFn
      moduleName = intercalate "." $ init splitFn
  if null moduleName
    then setImports ["Prelude"]
    else setImports ["Prelude", moduleName]
  eval $ fnName ++ arg
```

Listing 4.30 Running the interpreter on incoming code on the server-side

```
endpoint' :: (String, String) -> String -> IO String
endpoint' (s,f) arg = do
  res <- runInterpreter $ interpreterEndpoint (s,f) arg
  case res of
    Left err -> do
      print err
      pure "Failed"
    Right e -> pure e

main = do
  res <- endpoint' ("", "Unsafe.simpleFunction") " 3"
  print res
```

The endpoint takes in the code, passes it on to the interpreter, which splits it up and loads the module for the function, after which it evaluates it with arguments. The endpoint then returns this result as a `String`, from which we can return it to the client, and the client can handle the type casting to the correct type.

Through all of this, we have seen that TH offers a lot of opportunities, but at the cost of quite some complexity. A thing to note is that TH is known to not be very portable across hardware architectures, which might pose a problem if we wanted to use it on mobile hardware. The buy-in is fairly low, since it would need to be manually added, which also means the granularity is very fine.

4.6 Evaluation

To sum up the evaluation throughout this chapter, the rows for tbl. 4.1 are reiterated here again, for convenience:

- **C**omplexity of the implementation (very low–very high).
- **A**doptability by the wider community (very low–very high).
- **B**uy-in, for a developer to use the system (very low–very high).
- **G**ranularity of the offloading mechanism (very fine–very coarse).
- **S**erver-side story (no–yes).
- **P**ortability to other pure functional programming languages (no–yes).

Table 4.1: Overview of the pros and cons of the different proposals

Approach	**C**	**A**	**B**	**G**	**S**	**P**
Extending the runtime	Very High[6]	Low	High	Very Coarse[7]	No	No
unsafePerform	**Low**	**Mid**	**Low**	**Very Fine**[8]	**No**	**Yes**
Rewrite Rules	**Low**	**Low**	**Low**	**Very Fine**[9]	**No**	**No**

[6] While technically feasible, it would be a massive undertaking.

[7] All pure *known* functions with saturated arguments.

[8] Manually controlled by adding function calls before the code that should be offloaded.

[9] Needs a rewrite rule for every function that should support offloading.

Approach	C	A	B	G	S	P
Monadic Framework	Mid	High	Mid	Flexible[10]	Yes	Yes
Template Haskell	**High**	**Mid**	**Low**	**Very Fine**	**Yes**	**No**

Honorable Mentions

There were a few approaches we did not thoroughly inspect, but are still worth mentioning:

- **Manipulating the source**: This would involve preprocessing the Haskell source code, using something like `ghc-exactprint`[11], and then add our offloading code (e.g. via `unsafePerformIO` again) in the AST of the source code, before restructuring the program. This quite obviously seems like a brittle approach with very little control, with the complexity being in the mid-tier.
- **Compiler/Language Extension**: Another approach left out was to create a language extension that could be turned on, and then would rewrite suitable functions to allow offloading. This would have very little portability across languages, and would entail a bit of complexity while giving little control to the developer.

4.7 Summary

We have explored several ways to approach the design of our offloading system, with each their respective strengths and drawbacks, although strongly indicating that the monadic framework, presented in sec. 4.4, seems to be the most flexible and powerful approach.

[10] Very flexible granularity, since one can simply add more fine-grained effects if the offloading should be more fine-grained.

[11] https://hackage.haskell.org/package/ghc-exactprint

Common for most of these approaches—barring the monadic framework, and Template Haskell in sec. 4.5—is they all are missing a story for how to handle things on the server side; there needs to be a way to take in an arbitrary function, and somehow route it to the correct call along with its arguments.

One general way to handle this, as we see in lst. 4.29, is to place an interpreter as the endpoint on the server-side, and this would perhaps be the most flexible way to set things up, although not entirely desirable.

We land on the monadic framework, using `Freer` (or, extensible effects), as the approach we will proceed with in chapter 5, and use for our implementation. It allows us to cleanly separate our program semantics from implementation, allowing us to do different things on the server and client, for the same program. It also is the only one that provides both a story for server-side, and is also portable across languages.

CHAPTER 5

Offloading Using a Monadic Framework

If it was not clear from chapter 4 (and the title), we have chosen to implement our offloading system using the monadic framework approach, specifically using `Freer`, as it gave both better ergonomics, composability, extensibility and performance than `Free`, and allowed for a cleaner separation than MTL.

The implementation will be divided into three main components:

- Common: The shared code between the client and the server, e.g. types and offloadable functions.
- Frontend: Our client code that will reside on the mobile device.
- Backend: Our server code, which will be run in an cloud environment.

5.1 Common

We have a lot of shared code between the frontend and backend. Our data types for the program, along with functions to send them into the `Freer` monad. We also have our computations that both the frontend and backend are able to perform, depending on if the code is offloaded or not. As such, we provide a common layer between this two, by structuring it as a package, that they can both import. This ensures consistency in both the data types, effects and computations that can be run on the frontend and backend, along with keeping the algorithms the same for both.

First, let us set up the types that we will use to model our program. We divide these into two different data types, to have a clear separation of the logic. In lst. 5.1 we define `Operation` which will represent effectful operations that we do not want to offload, while `Computation` will represent the pure code we want to offload. We also create a `ComputationRepr` to represent the functions when we send them over the network.

Listing 5.1 Defining our data types that will be used to model the program

```
{-# LANGUAGE ConstraintKinds #-}
module Common.Types where

type Serialize a = Show a

data Operation next where
  WriteOutput :: Serialize a => a -> Operation ()

data Computation next where
  IsPrime :: Int -> Computation Bool
  FactorialLength :: Int -> Computation Int

data ComputationRepr
  = IsPrimeRepr
  | FactorialLengthRepr
  deriving (Show, Read)
```

The operations and computations are intentionally kept simple, since the point is to demonstrate the usage of this approach, and not the performance of it. A output function is enough to model effectful behaviour and we can turn our `IsPrime` and `Factorial` into compute heavy functions later on, while keeping the interface simple.

We proceed in lst. 5.2 by defining the functions that turn our GADT constructors into effects, which we will be using in the program.

Finally, our computations are by design slightly inefficient, scaling with time on the input. Shown in lst. 5.3 we define

Listing 5.2 Functions to convert our GADTs into effects

```
{-# LANGUAGE FlexibleContexts #-}
module Common.Operations where
import Control.Monad.Freer
import Common.Types

writeOutput :: (Member Operation effs, Serialize a) =>
            a -> Eff effs ()
writeOutput = send . WriteOutput

isPrime :: Member Computation effs => Int -> Eff effs Bool
isPrime = send . IsPrime

factorialLength :: Member Computation effs =>
                Int -> Eff effs Int
factorialLength = send . Factorial
```

`computeFactorialLength` to calculate the factorial of n, and return the number of digits in the result. The `computeIsPrime` function is a naïve primality test, which checks if n is divisible up to square root of n.

Listing 5.3 The implementation of `FactorialLength` and `IsPrime`

```
module Common.Computation where

computeFactorialLength :: Int -> Int
computeFactorialLength n = length $ show $ product [1..11000]

computeIsPrime :: Int -> Bool
computeIsPrime n = null [ p | p <- [2..sqrt (fromIntegral n)]
                            , n `mod` p  == 0]
```

The frontend and backend can now pull in this code, to carry out the computations on each side.

5.2 Frontend

Our frontend will consists of some simple UI code, and then an interpreter that will allow us to run our computations and decide if we want to offload it. The frontend consist of:

- UI code.
- Communication to Swift/Native mobile code.
- Our offloading interpreter for our operations and computations.
- A profiler that keeps track of our network conditions.

Our UI is merely to let the user know that the app has launched, and not much else. We are using `reflex-dom`[1] to setup the interface, as shown in

To communicate with Swift, we must set up some GHCJS FFI calls into JavaScript, where the function to call Swift will live. The Swift code will be explained later, in sec. 5.4. As shown in lst. 5.5 we set up some conditional FFI code. If we are using GHCJS it adds the FFI call, and if we are on plain GHC, then it simply prints the output.

Our interpreters are set up to call the compute functions defined in the common package, as shown in lst. 5.6 and lst. 5.7

We run the UI and the interpreters concurrently in the main function, as shown in lst. 5.8.

5.3 Backend

The backend is responsible for accepting requests from clients, and then making sure they are run using the correct functions before they return their result back to the client. As such, our backend consists of:

[1] Can be found in https://github.com/reflex-frp/reflex-platform

Listing 5.4 A simple user interface for the offloading app

```
module UI where
import Reflex.Dom
import Data.Map as Map
import Data.Monoid


headElem :: MonadWidget t m => m ()
headElem = do
  el "title" $ text "Offie The Offloader"
  elAttr "meta" (Map.fromList [("charset", "UTF-8")]) $ return ()
  styleSheet "styles.css"
  where
    styleSheet name = elAttr "link" (
          ("rel" =: "stylesheet")
        <> ("type" =: "text/css")
        <> ("href" =: name))
        blank

bodyElem :: MonadWidget t m => m ()
bodyElem = elClass "div" "main-component" $ do
  rec
    el "h1" $ text "Welcome to Offie The Offloader"
    el "p" $ text "Tests will start automatically."
  blank
```

- A router in accepting HTTP requests and passing them on.
- An interpreter that can evaluate our incoming requests.

We set up our backend as a simple web server that takes in a `ComputationRepr` and a value, and returns the calculated value, as shown in lst. 5.3.

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}
```

```haskell
module Main where

import Data.Text
import Data.Time (UTCTime)
import Servant.API
import Data.Proxy
import Control.Monad.Trans.Either
import Network.Wai.Handler.Warp

import Common.Types
import Common.Computation

main :: IO ()
main = run 8080 (serve api server)

type Handler a = EitherT ServantErr IO a

type OffloadApi = "off" :> QueryParam "f" ComputationRepr
                        :> QueryParam "v" Int :> Get Int

api :: Proxy OffloadApi
api = Proxy

server :: Server OffloadApi
server = compute
  where
    compute mFun val =
      case mFun of
        Nothing -> "ERROR: Needs a function"
        Just IsPrimeRepr -> computeIsPrime val
        Just FactorialLengthRepr -> computeFactorialLength val
```

This is of course a simplification of what a real system should do, but it will suffice for now.

5.4 Getting the Code onto a Mobile Device

Now that we have written our implementation, we want to get it onto a mobile device. There are a couple of ways to go about this:

- Generate native code: Still *very* experimental, better support for cross-compilation will most likely land in GHC 8.4. This would be the most desirable approach, if the ecosystem eventually matures, as we can compile both for iOS and Android via this.
- Generate Java via Eta: Eta is a port of GHC that runs on the JVM. This means it is able to generate JAR files, and supports Java via FFI. Unfortunately this will only be able to run on Android devices, since it is the only device that runs Java.
- Generate JavaScript via GHCJS: Much like Eta, GHCJS is also a port. GHCJS compiles to clean (i.e. readable) JavaScript, and also supports NodeJS. Compiling to JavaScript means we can target a WebView—an integrated browser environment—on the mobile platform. This provides the broadest compatibility, and even allows the app to run on the web, if so desired.

As an alternative to GHCJS, one could use PureScript to target mobile via WebViews. PureScript was made specifically to compile to JavaScript, and also provides a sufficiently advanced type system to compete with Haskell. Idris is another alternative, but it has a rather immature JavaScript backend as of current. Finally, if one has simpler needs, Elm also targets JavaScript and is a purely functional programming language.

With these choices, we have opted to go for generating JavaScript via GHCJS. We will therefore need a little setup code done in our mobile platform, that will load our index.html files and our JavaScript files. This does mean that we incur a slight performance penalty, at the cost of flexibility in development. That said, using JavaScript on for mobile app development is certainly not an unusual thing, with large companies like Facebook using it, along with a lot of other companies.

Setting up Xcode/iOS

We are going to demonstrate how to get our generated JavaScript to run on iOS via a WebView, specifically the `WKWebView` component.

First off, we create an Xcode project, by choosing a `Single View App`, as shown in fig. 5.1. In the following screen, will name the app `OffApp`, and choose `Swift` as our language.

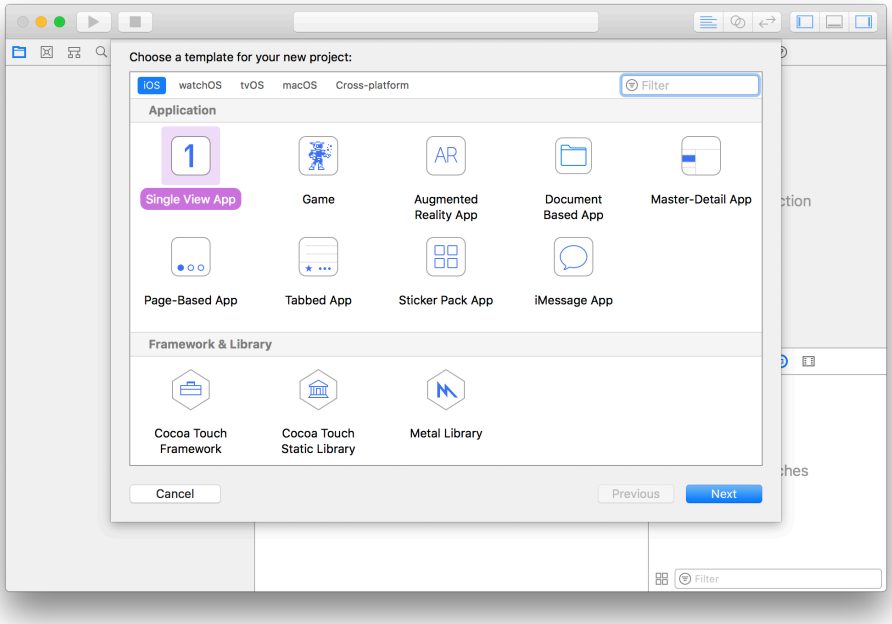

Figure 5.1: Xcode project setup

We should now have a project structure that looks something like the following:

```
OffApp
|__ AppDelegate.swift
|__ Assets.xcassets
|   |__ AppIcon.appiconset
|       |__ Contents.json
```

```
|  Base.lproj
|   |__ LaunchScreen.storyboard
|   |__ Main.storyboard
|__ Info.plist
|__ ViewController.swift
```

We will then create a group called `Web` in the `OffApp` folder, and put in our `index.html`, `styles.css` and our JavaScript files. Our new structure looks like:

```
OffApp
|__ ...
|__ Web
    |__ index.html
    |__ styles.css
    |__ rts.js
    |__ out.js
    |__ lib.js
    |__ runmain.js
```

Getting the Swift code to load our `index.html` into a WebView is fairly simple, as shown in lst. 5.9. First though, we create a `Container View` and two labels in our Main.storyboard, and connect them to our `ViewController`. Then we import `WebKit`, and then instantiate the `WKWebView` and add it to the `containerView`. Finally, we set the page to load in `viewDidLoad`, by giving the webview the path to our `index.html` file.

Communicating from JavaScript to Swift

We can also extend the WebView to respond to JavaScript events. This is done by setting up a controller that will handle incoming calls, placed under `window.webkit.messageHandlers.EVENTNAME.postMessage(MESSAGE)` in JavaScript. This is done by changing `let webConfiguration = WKWebViewConfiguration()` from `loadView` to the code in [@].

We then extend our `ViewController` class with `WKScriptMessageHandler` and add a method `userContentController`, which will handle the

incoming requests. In lst. 5.11 we set up the handler to `print` incoming messages of type `log` to the Swift console.

Finally, we can compile the application and run in on iOS, as shown in fig. 5.2.

5.5 Summary

We have fairly easily structured together a program that mixes UI code, and our `Freer` interpreters. The small demo can easily be polished, but this was just a proof-of-concept. Ideally, the interpreters would also be responsible for generating the UI code, and then we can run the whole program in the interpreters, giving it more power.

By generating JavaScript we have an easy way to get our pure functional code onto several mobile platforms, and can even interact with the native code part through JavaScript FFI calls from our code.

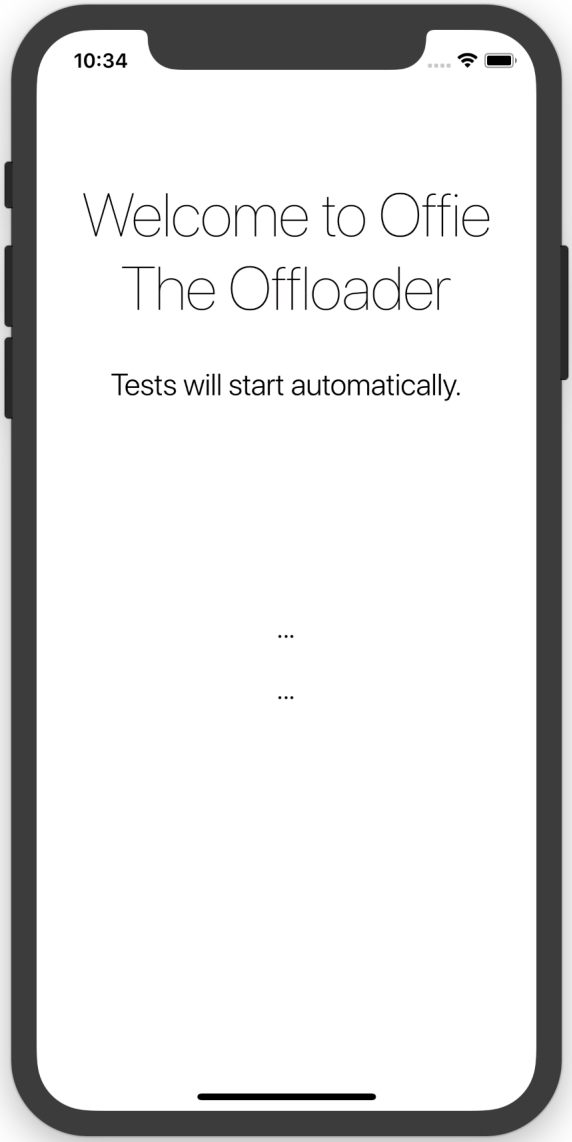

Figure 5.2: iOS App Running Offie

Listing 5.5 GHCJS FFI into JavaScript which communicates to the Swift part

```
{-# LANGUAGE JavaScriptFFI #-}
{-# LANGUAGE CPP #-}
module JavaScriptFFI (logToiOS) where
#ifdef ghcjs_HOST_OS
import Data.JSString (JSString, pack)

foreign import javascript unsafe
  "if (typeof window.webkit === 'undefined') { window.webkit =
    {messageHandlers: {log: {postMessage: console.log }}}};
    window.webkit.messageHandlers.log.postMessage($1)"
  iOSLog :: JSString -> IO ()
foreign import javascript unsafe
  "if (typeof window.webkit === 'undefined') { window.webkit =
    {messageHandlers: {log: {postMessage: console.labelOne }}}};
    window.webkit.messageHandlers.labelOne.postMessage($1)"
  iOSLabel1 :: JSString -> IO ()
foreign import javascript unsafe
  "if (typeof window.webkit === 'undefined') { window.webkit =
    {messageHandlers: {log: {postMessage: console.labelTwo }}}};
    window.webkit.messageHandlers.labelTwo.postMessage($1)"
  iOSLabel2 :: JSString -> IO ()

logToiOS :: String -> IO ()
logToiOS s = iOSLog (pack s)

setiOSLabel1 :: String -> IO ()
setiOSLabel1 s = iOSLabel1 (pack s)

setiOSLabel2 :: String -> IO ()
setiOSLabel2 s = iOSLabel2 (pack s)
#else
logToiOS :: String -> IO ()
logToiOS s = putStrLn s

setiOSLabel1 :: String -> IO ()
setiOSLabel1 s = putStrLn s

setiOSLabel2 :: String -> IO ()
setiOSLabel2 s = putStrLn s
#endif
```

Listing 5.6 The interpreters for our events

```
{-# LANGUAGE FlexibleContexts #-}
module Interpreter where

import Control.Monad.Freer

import Common.Operations
import Common.Types
import Common.Computation
import JavaScriptFFI
import Network.Wreq
import Control.Lens
import Profiler (shouldOffload)
import Text.Read (readMaybe)

offload :: String -> String -> IO String
offload fun val = do
  r <- get url
  r ^. responseBody
  where url = "http://localhost:8080/off?f=" ++ fun ++ "&v=" ++ val

program :: Eff '[Operation, Computation, IO] ()
program = do
 fac <- factorialLength 22
 writeOutput fac
 result <- isPrime 12
 writeOutput result

runProgramM :: Eff '[Operation, Computation, IO] a -> IO a
runProgramM p = runM . runComputationM . runEventM $ p

runEventM :: forall effs a. LastMember IO effs => Eff (Operation ': effs) a -> Eff ef
runEventM = interpretM $ \case
  WriteOutput s -> logToiOS (show s)
```

Listing 5.7 The interpreters for our computations

```
runComputationM :: forall effs a. LastMember IO effs => Eff (Computation ': e
runComputationM = interpretM $ \case
  IsPrime i -> do
    shouldOffload' <- shouldOffload
    if shouldOffload'
    then do
      c <- offload (show IsPrimeRepr) (show i)
      setiOSLabel1 c
      case (readMaybe c) :: Maybe Int of
        Nothing -> pure 0
        Just i -> pure i
    else do
      let c = computeIsPrime i
      setiOSLabel1 (show c)
      pure c
  FactorialLength i -> do
    shouldOffload' <- shouldOffload
    if shouldOffload'
    then do
      c <- offload (show FactorialLengthRepr) (show i)
      setiOSLabel2 c
      case (readMaybe c) :: Maybe Int of
        Nothing -> pure 0
        Just i -> pure i
    else do
      let c = computeFactorialLength i
      setiOSLabel2 (show c)
      pure c
```

Listing 5.8 Running the frontend

```
module Main where

import Control.Concurrent.Async.Lifted.Safe
import qualified Language.Javascript.JSaddle.Warp as JSaddle
import Reflex.Dom.Core (mainWidgetWithHead)

import UI
import Interpreter

main :: IO ()
main = do
  concurrently_
    (runProgramM program)
    (JSaddle.run 3709 $ mainWidgetWithHead headElem bodyElem)
```

Listing 5.9 Setting up a `WKWebView` in Swift to load our `index.html`

```
import UIKit
import WebKit

class ViewController: UIViewController, WKUIDelegate,
                      WKNavigationDelegate {
    var webView: WKWebView!
    @IBOutlet var containerView: UIView!
    @IBOutlet var firstLabel: UILabel!
    @IBOutlet var secondLabel: UILabel!

    override func viewWillAppear(_ animated: Bool) {
        webView = WKWebView(
            frame: CGRect(x: 0, y: 0,
                          width: view.frame.width,
                          height: containerView.frame.height),
            configuration: webConfiguration)
        view.addSubview(webView)

        let htmlPath = Bundle.main.path(
            forResource: "index",
            ofType: "html")
        let htmlUrl = URL(
            fileURLWithPath: htmlPath!,
            isDirectory: false)
        webView.loadFileURL(
            htmlUrl,
            allowingReadAccessTo: htmlUrl)
    }

    // ...
}
```

Listing 5.10 Changing the `WKWebView` configuration to attach a script controller

```
var webConfiguration: WKWebViewConfiguration {
    get {
        let webConfig: WKWebViewConfiguration
               = WKWebViewConfiguration()
        let userController: WKUserContentController
               = WKUserContentController()
        userController.add(
            self as WKScriptMessageHandler,
            name: "log")
        webConfig.userContentController = userController;
        return webConfig;
    }
}
```

Listing 5.11 Set up handler for JavaScript events

```
func userContentController(_ userContentController:
     WKUserContentController, didReceive message: WKScriptMessage) {
    if message.name == "log" && message.body is String {
        print(message.body)
    }
    if message.name == "labelOne" && message.body is String {
        firstLabel.text = message.body as? String
    }
    if message.name == "labelTwo" && message.body is String {
        secondLabel.text = message.body as? String
    }
}
```

CHAPTER 6

Conclusion and Evaluation

Offloading still remains a relevant pursuit with many opportunities to save energy and enhance performance. There are many factors to take into account when designing a system for offloading, with some of the primary being:

- Network conditions.
- Latency and RTT.
- Throughput.
- Packet loss ratio and signal strength.
- Method conditions.
- Size of input and output.
- Execution time.

Based on our research of previous systems, we have explored several different approaches to offloading using a purely functional programming language, taking into account how they can help alleviate some of the core problems that trouble other systems:

- Global mutable state.
- Danger of re-execution.

We have then turned one of theses approaches into a small proof-of-concept. With minimal setup, our approach is able to handle small cases, and will be able to scale to more and more computations and events. With a little bit of polish, a lot of the code redundancy can be abstracted out into smaller pieces of code that streamline the API.

With the above we are certain that purely functional programming is a good fit for the offloading paradigm, and can provide optimal conditions for designing the various components that are involved in the system.

Future Work

The future direction of this work could involve going deeper into type safety. For example, dependent types offer an interesting avenue to pursue. With dependent types, one can encode extra information at the type level, such as requiring certain network conditions for code to be offloaded, and can dismiss functions at compile-time if they try to break this condition.

Further work and time could also be invested into creating a library that embody the ideas layed out in this thesis, providing ready made interpreters that can take care of offloading code, and handling all the underlying details for the user.

Bibliography

Benedetto, J.I., Neyem, A., Navon, J., and Valenzuela, G. 2017. Rethinking the Mobile Code Offloading Paradigm: From Concept to Practice. *Proceedings - 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems, MOBILESoft 2017*, 63–67.

Brown, A. and Wilson, G. 2012. The Architecture of Open Source Applications - Volume I. 388.

Chun, B., Ihm, S., Maniatis, P., Naik, M., and Patti, A. 2011. Clonecloud: Elastic Execution Between Mobile Device and Cloud. *Proceedings of the sixth conference on Computer systems*, 301–314.

Chun, B.-G. and Maniatis, P. 2009. Augmented smartphone applications through clone cloud execution. *HotOS'09 Proceedings of the 12th conference on Hot topics in operating systems*, 8.

Cuervo, E., Balasubramanian, A., Cho, D.-k., et al. 2010. MAUI: Making Smartphones Last Longer with Code Offload. *MobiSys'10 17*, 49–62.

GHC Commentary. *GHC Commentary: RTS Function Calls.*.

GHC Team. 2017. *GHC User's Guide Documentation.*.

Golkarifard, M., Yang, J., Movaghar, A., and Hui, P. 2017. A Hitchhiker's guide to computation offloading: Opinions from practitioners. *IEEE Communications Magazine 55*, 7, 193–199.

Huang, J., Quian, F., Gerber, A., Mao, Z.M., Sen, S., and Spatscheck, O. 2012. A Close Examination of Performance and Power Characteristics of 4G LTE Networks. *MobiSys*, 225–238.

Jiao, L., Friedman, R., and Fu, X. 2013. Cloud-based computation offloading for mobile devices: State of the art, challenges and opportunities. *Future Network and Mobile Summit (FutureNetworkSummit)*, 1–11.

Kiselyov, O. and Ishii, H. 2015. Freer monads, more extensible effects. *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell -*

Haskell 2015, 94–105.

Launchbury, J. and Peyton Jones, S.L. 1995. State in Haskell. *LISP and Symbolic Computation 8*, 4, 293–341.

Lin, C.-k. 2014. Mobile App Acceleration via Fine-Grain Offloading to the Cloud. *6th USENIX Workshop on Hot Topics in Cloud Computing*.

Marlow, S. and Jones, S.P. 2006. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *JFP 16*, 5, 415–449.

Nabi, T., Mittal, P., Azimi, P., Dig, D., and Tilevich, E. 2015. Assessing the benefits of computational offloading in mobile-cloud applications. *MobileDeLi 2015 - Proceedings of the 3rd International Workshop on Mobile Development Lifecycle*, 17–24.

Sabry, A. 1998. What is a Purely Functional Language ? 1 Functional Languages and Computational E ects. *Language 1*, January, 1–22.

Sheard, T. and Jones, S.P. 2002. Template meta-programming for Haskell. *ACM SIGPLAN Notices 37*, 12, 60.

Simonite, T. 2017. Apple's 'Neural Engine' Infuses the iPhone With AI Smarts | WIRED. https://www.wired.com/story/apples-neural-engine-infuses-the-iphone-with-ai-smarts/.

APPENDIX A

Comparison of Results

MAUI

	Facial Recognition		Video Game	
Local	28J	**100%**	50J	**100%**
WiFi (10ms RTT)	3J	**10.71%**	33J	**66.00%**
WiFi (25ms RTT)	3J	**10.71%**	34J	**68.00%**
WiFi (50ms RTT)	4J	**14.29%**	36J	**72.00%**
WiFi (100ms RTT)	4J	**14.29%**	40J	**80.00%**
3G (220ms RTT)	7J	**25.00%**	55J	**110.00%**

CloneCloud

	Image Search		Behaviour Profiling	
Local	155J	**100%**	40J	**100%**
WiFi (69ms RTT)	35J	**22.73%**	20J	**50.00%**
3G (680ms RTT)	67J	**43.48%**	50J	**125.00%**

UpShift

	Handwriting (100 iterations)		Handwriting (200 iterations)	
Local	6.00%	**100%**	12.00%	**100%**
WiFi (16ms RTT)	4.00%	**66.67%**	7.00%	**58.33%**

MobiCOP

	Pure Java Computation		Large Input/Output	
Local	173J	**100%**	1332J	**100%**
WiFi	16J	**9.25%**	173J	**12.99%**
3G	28J	**16.18%**	260J	**19.52%**

$\sqrt{17}$
$\Omega \int \delta e^{i\pi} = -1$
{2.7182818284}ϑφέρτυθιοπσδφγηξκλ
χ^2
$\sum$
λ