Haskzure: Azure Resource Manager bindings in Haskell

Nashwan Azhari

June 18, 2016

Abstract

We describe the process of designing and implementing Haskzure, a Haskell library for managing Microsoft Azure through its set of Resource Manager APIs. The main purpose of the library is to provide a bare minimum set of functionality for performing CRUD operations on the more common compute and networking resources available on Azure under a unified interface familiar to functional programmers. Considering the enourmous surface area of the ARM APIs (1100+ individual calls available), emphasis was placed more on designing the library so as to make it as easy to extend as possible versus focusing on feature-completeness.

We will describe all the technical implications of writing a client library for the ARM APIs, as well as how Haskzure makes use of the common purely functional programming idioms available in Haskell to achieve a high degree of code reuse, and how the set of core components which provide the generic interface for interacting with Azure are layed out and implemented.

Contents

1 Introduction **3**
- 1.1 Goal . . . 3
- 1.2 Motivation . . . 3
- 1.3 Related Work . . . 4

2 Theoretical Overview **5**
- 2.1 Haskell . . . 5
  - 2.1.1 Purely Functional Programming Style . . . 5
  - 2.1.2 Strong Typing Discipline and Compiler Extensions . . . 5
  - 2.1.3 Non-Strict Evaluation . . . 6
  - 2.1.4 Category Theory Concepts . . . 6
  - 2.1.5 Template Haskell . . . 6
- 2.2 Azure and the ARM APIs . . . 6
  - 2.2.1 Microsoft Azure . . . 6
  - 2.2.2 The Azure Storage API . . . 7
  - 2.2.3 The Azure Service Management API . . . 7
  - 2.2.4 The Azure Resource Manager APIs . . . 7
- 2.3 Building an ARM Client Library . . . 8
  - 2.3.1 Technical Details . . . 8
  - 2.3.2 The Swagger API Description Format . . . 11
  - 2.3.3 Swagger-based Client Library Generation . . . 12
  - 2.3.4 Azure AutoRest . . . 13

3 Haskzure's implementation **14**
- 3.1 The Library's interface . . . 14
- 3.2 Base Typeclasses . . . 14
- 3.3 Generating the data structures from Swagger . . . 14
- 3.4 Details on the implementation of Authentication . . . 14
- 3.5 The low-level functions . . . 14
- 3.6 The high-level interface . . . 14

4 Using the Library **15**

5 Conclusions **16**

6 Toolchain and Libraries Used **17**

Bibliography and References **17**

Listings:

1 OAuth2 token request example 8
2 Structure of token response. 9
3 URL format of an ARM resource on Azure 9
4 JSON representation of an Availability Set's properties. 10
5 Swagger excerpt for Availability Set creation operation. 12

Chapter 1

Introduction

1.1 Goal

Considering the prevalence of Cloud Computing and its immense impact on the IT industry as a whole, companies nowadays are commited to moving as much of their IT workloads (infrastructure, storage arrays, big data processing etc..) 'into the cloud'. Whilst many choose to invest in their own datacenters and build an in-house *private cloud* solution, the vast majority prefer to turn to so-called *public clouds* for hosting all of their IT needs. As such, the private cloud business is one of the most fast-growing industries in the field, with huge names such as Amazon's AWS, Microsoft's Azure and Google's Compute Engine offering a vast selection of both cloud infrastructure elements and services to anyone in need.
With this stunning growth in the industry and the shift to even more distributed application architectures born to run in the cloud, the traditional model of a company hiring a handful of system administrators to managed their infrastructure and application lifecycle needs is no longer a feasible one. As the cloud became a solution to programatically configuring IT infrastructure, so, now, must there be a solution for configuring the clouds. In this sense, the most prevalent approach is that of clouds providing a means of remote management (most commonly HTTP-based REST APIs), through which, with the aid of 'language binding libraries' (aka SDKs), programmers may gain access to and programatically deploy, modify and manage all the resources the cloud platform offers.
Considering that managing cloud infrastructure is an inherrently transformational process, and that no programming paradigm better captures this process than functional programming, we will focus on providing such a library of language bindings for Microsoft's Azure public cloud platform (specifically, the set of 'Azure Resource Manager' APIs), in Haskell, one of the most representative languages in the functional programming space.

1.2 Motivation

This thesis describes the design and implementation of a 'Haskzure', a Haskell library which provides various data structures and functions for interacting with the 'Azure Resource Manager' (or ARM) set of APIs for Microsoft's Azure Cloud platform.

The library should:

- expose the standard set of CRUD operations for the most commonly used cloud resources available through the ARM APIs
- conform with the Haskell language's high-level and safe nature, making it easy to use within a functional context
- provide an easy way of extending the library's functionality to newer types of cloud resources
- include behavioral tests for all internal parts of code not directly interacting with Azure

Considering the large surface area of the ARM APIs (about 300 resource types and over 1100 individual API calls), the emphasis is placed on a good design of the core components for easy extensibility versus absolute feature completeness. As will be shown, the abstractions available in Haskell offer a straightforward means of making the core functionality completely generic, and thus permitting for library features to be plugged in very easily.

1.3 Related Work

Cloud API binding libraries are commonplace for the more popular cloud platforms, and, in most cases, these libraries are offered by the cloud vendor themselves for a selection of the most used programming languages. In the case of ARM, for example, there are open source libraries [1] available from Microsoft for C#, Go, Java, JavaScript, PHP, Python and Ruby, provided with the intent of making it easier for developers to manage the infrastructure under their Azure subscription.
On the Haskell side, however, there are only a handful of such libraries. Most notable would be a set of libraries all focused on interacting with Digital Ocean, and one fairly limited but well-designed one for AWS [2]. For Azure, the only Haskell-related tooling comes in the form of a library which makes use of the now obsolete Azure Service Management (or ASM) API [3]. However, it only offers limited management capabilities over ASM Cloud Services with the express intention of being used with Cloud Haskell (a Haskell runtime designed to distribute programs over multiple machines).
As previously mentioned, the extremely large surface area of the ARM APIs make implementing a fully-featured library a daunting task, which better motivates the choice of Microsoft itself to almost fully auto-generate all of their libraries through means detailed later. We have chosen, however, to use the levels of abstraction provided by the Haskell language to move the boundary of code generation farther away by hand-writing a core set of functionalities which are designed to be as generic as possible and only generating the static datastructures those core operations operate upon.

Chapter 2

Theoretical Overview

2.1 Haskell

Haskell is a general-purpose, purely functional programming language developed by a committee of researchers [4] starting from the early 1990s. It is one of the most popular functional programming languages out there as it lies as the forefront of research in the field. It is a language which offers a handful of independent implementations, most of which compile Haskell to native machine code. By far the most popular, however, is the *Glasgow Haskell Compiler* (or GHC), as it is the one being actively developed by the committee, and thus the most up-to-date one with regards to the latest developments in the language. It is also the compiler of choice for this project, with more details provided in the section about tooling. Haskell's combination of features makes it largely unique in terms of the programming principles and techniques it offers, which we enumerate in the subsections to follow:

2.1.1 Purely Functional Programming Style

The functional programming style has long been considered relatively cumbersome due to its avoidance to mutable state and side-effects. As such, many have argued that the cost of complete purity (in the mathematical sense, applied to functions) far outweighs its benefits, and thus purely functional languages will never actually be 'useful'. Haskell manages to maintain purity throughout by employing a number of 'functional design patterns' (most ideas having been borrowed from the field of *Category Theory*), to deal with the likes of mutable state and I/O while maintaining purity.
The style lends itself very well to transformational problems (problems involving numerous computations and transformations done on data), and as such is very concise in the modelling and management of cloud infrastructure.

2.1.2 Strong Typing Discipline and Compiler Extensions

Haskell is a statically typed language, with one of its main virtues being its expressive type system. It is a system designed around *Algebraic Datatypes* (or ADTs), and *typeclasses*, which define type-level constraints on the said ADTs. The system is very minimal at heart, but it however allows for almost all of the diverse types, constraints and their combinations which comprise the concise yet flexible standard libraries.
In addition, there are a handful of compiler extensions which either allow for some added behavior to the typeclass system (MultiParamTypeClasses, FlexibleContexts...), or some extending of the datatype system (ex: DataKinds, TypeFamilies, GADTs etc...). Although not part of the official standard, most of the compiler extensions come packaged with GHC and enjoy widespread use due to their added value to the language. When detailing the implementation, we will also mention the more important extensions used in the creation of Haskzure.

2.1.3 Non-Strict Evaluation

A programming language's 'evaluation strategy' refers to the way expressions, and operations which operate on expressions, are evaluated. More precisely, it is largely concerned with *when* the evaluation occurs. Most programming languages employ a 'strict evaluation' model, where a computation is performed as soon as the expression is bound to a variable (but not necessarily needed). This model works very well for most cases, but it does carry the chance of there being more work done than needed (such as evaluating expressions assigned to unused variables, for example).
In languages with a 'non-strict evaluation strategy', excess work is usually prevented by various mechanisms. In Haskell, for example, the exact strategy used is named *call-by-need*, but is often referred to as 'lazy evaluation'. In effect, the value of an expression is only ever computed when needed, a practice often cobined with *memoization* to also cache already performed computations so as to not recompute them (for example, the base cases of tail-recursive functions can be very useful to cache).
Despite having clear advantages (especially in terms of memory usage), lazy evaluation often makes it harder to reason about the runtime behavior of Haskell programs, especially in a concurrent [5] setting. Haskell does of course offer a means of specifying the strictness of a value's evaluation, a feature which may be employed when it is deemed safer to perform computations up-front versus the default deferred manner (in fact, *Aeson* [6], the library we employ to encode and decode the JSON payloads we intract with ARM through is inherrently strict).

2.1.4 Category Theory Concepts

Haskell borrows numerous concepts from *Category Theory*, a branch of mathematics dedicated to formalizing structures solely based on collections of objects and arrows. Most notably, abstract constructs such as the *Functor*, *Applicative Functor*, and the *Monad* are modelled as core typeclasses within Haskell to abstract numerous non-functional concepts ranging from mutable state to exception handling in a purely functional context (more precisely, as a binding together of operations within a certain context which entails intermediate 'gluing' operations [7]). For Haskzure, we chose to conform to the *mtl* (standing for 'Monad Transformers Library'), which defines the standard set of Monad implementations used through most Haskell code (similar to what the STL is for C++). These powerful abstractions, despite giving the language a steeper learning curve, allow for an amazing degree of conciseness if the problem domain is properly modelled, as will be shown later.

2.1.5 Template Haskell

Metaprogramming is an integral part of any high-level programming language. In Haskell's case, metaprogramming features come in the form of *Template Haskell*, a compiler extension and library which basically allow one to create typed 'splices' and insert them into the AST of a module before it gets compiled. This allows for numerous use-cases, such as automatically declaring datatypes and instantiating them to specific typeclasses, which is one of the main drivers of Haskzure's implementation we will detail more in the next chapter.

2.2 Azure and the ARM APIs

2.2.1 Microsoft Azure

Azure is the public cloud offering provided by Microsoft. It is an immensely intricate system, with over 20 datacenters located worldwide serving a vast array of compute, networking, storage and service resources to businesses and even the US government. As such, it is a very hard system to manage efficiently, as often the resources you are deploying are not only handled by different Azure resource providers, but will likely get deployed on different continets entirely.
Like most public clouds, Azure offers a set of APIs for managing the resources for your specific subscription. To be exact, Azure has more than one active API facade at the user's disposal.

2.2.2 The Azure Storage API

The Azure Storage API offers access to manipulate 'storage services', the service-based approach to storage solutions offered by Azure. Storage services are the container under which all other storage entities must live. These include the likes of:

- storage containers, which are equivalent to the directories of a filesystem by holding storage blobs
- storage blobs, which come in two flavors (namely block and page blobs), and are akin to files on a filesystem, but are made accessible via HTTP/HTTPS
- storage queues, which are the online messaging queue solution offered by Azure
- file storage services, which are separate services which provide file sharing capabilities via the SMB protocol
- table storage services, which offer storage to unstructured data in a similar manner to NoSQL databases

Alongside these, Azure also has a comprehensive offering of SQL Server services, big data repositories (marketed as 'Data Lake Store'), and fully-fledged NoSQL database services.
The Storage API is accessed via sending HTTPS requests with XML payloads, (authentication being proven by virtue of special storage service keys), and provides numerous configuration options (such as georeplication of storage service contents, for example). For all the intents and purposes of Haskzure, however, we will not be interacting with the Storage API at all, as all of our few storage needs are also covered by the ARM APIs.

2.2.3 The Azure Service Management API

The Azure Service Management (or ASM) API was the initial cloud management API for Azure since its launch in 2010. The API is RESTful, the means being XML payloads sent over HTTPS with authentication done via thumbprints of X.509 certificates which had been pre-installed onto Azure and referenced in the application interacting with Azure. Back in its early days, Azure was leaning more towards a Platform as a Service-type system, with the main focus being on abstracting away as many details of the infrastructure as possible and allowing developers to deploy their application directly and let Azure manage its resource needs.
As such, the main unit of work under the old ASM API was the so-called 'Cloud Service', which represented a collection of virtual machines and their connecting resources (network interfaces, virtual networks, etc...) meant specifically for said virtual machines. This model very much enforced an SOA (Service-Oriented Architecture) on applications and posed a number of issues within the API's design, most prevalent being the close ties between cloud resources within a particular Cloud Service, but the relatively cumbersome links with the rest of the infrastructure elements, which even lead to questions of whether ASM was indeed fully REST-compliant or not.
All in all, while the ASM API will likely still be available for use for a long time to come, it is slowly but steadily being eclipsed by the ARM APIs.

2.2.4 The Azure Resource Manager APIs

The Azure Resource Manager (ARM) set of APIs, first released in 2015, represents the new solution for programatically managing your infrastructure on Azure. As opposed to ASM, ARM better fits the Infrastructure as a Service role by focusing on providing operations for managing infrastructure elements atomically as opposed to ASM's restrictive model of Cloud Services. It is better thought of as a set of multiple API facades unified under a single top-level one. To be precise, the Azure team has made completely separate APIs for the various types of cloud resource providers they offer (i.e. compute, networking etc...), each effectively having its own namespace under a grand unified one dubbed the 'Resource Manager', as opposed to ASM's flat namespace implementation. The API is

fully REST-compliant, with JSON-encoded payloads being sent via HTTPS to the appropriate subpaths of the resource providers. *OAuth2* is used as the authorization framework, with your application or Azure user having to be registered within an Azure *Active Directory* which acts the *authorization server* for all the OAuth2 flows currently offered via Azure.
ARM's only real enforcement when it comes to the resources you're deploying is that they must all be allocated to a 'Resource Group'. This is however nothing more that a matter of labelling, as all resources may freely communicate with others across resource groups (but not locations), the mechanism offering a simple way of aggregating heterogeneous resources and performing operations on them in bulk (like deleting all the resources in a group, for example...).
As an added feature, the top-level 'Resource Manager' provider offers the ability to deploy a resource group by providing the JSON-encoded configuration of all the resources desired in it, exactly as happens in the case Amazon's Cloud Formation (CFN) or OpenStack's Heat. Despite its handiness though, this 'template deployment' mode of operation cannot be really considered a complete orchestration solution, as its main purpose is just deploying the resources, after which no added management/monitoring is done. Of interest, however, is the fact that the format of the resources encoded as JSON is almost identical with the one thorugh which we interact with the APIs, further proving that the Resource Manager API is nothing more than a union of those from the individual resource providers.

2.3 Building an ARM Client Library

2.3.1 Technical Details

In a nutshell, the ARM APIs work by sending the JSON-encoded payloads of operations via HTTPS to the Resource Manager Rest API. The details of this process are presented in the following sections.

Authorization via Azure Active Directory

The authorization framework of choice is *OAuth2*, which is designed to allow for easy implementation of authorization within applications without the explicit need of users to provide username and password credentials to them. In the case of ARM, Azure Active Directory resources play the role of the *authorization server* in all the supported OAuth2 flows.
In essence, one must register their application within an instance of Azure Active Directory under his/her subscription (a straightforward process done via the Azure online user interface), after which they will be communicated the 'client ID' and 'client secret' with which their application may solicit a temporary access token for use in authorization when actual calls to manage ARM resources are made. The scenario presented above falls under what is known as the 'client credentials grant', in which an HTTP URL-encoded POST is performed on the authentication endpoint as shown in Listing 1.

```
POST /<azure_ad_id>/oauth2/token?api-version=1.0 HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Host: login.windows.net
Content-Length: 123

grant_type=client_credentials&resource=<resource>&
    client_id=<client_id>&client_secret=<client_secret>
```

Listing 1: OAuth2 token request example

Note that all of the parameters to be filled within the URL-encoded POST body are part of the standard OAuth2 client credentials grant flow with the 'resource' we are requesting access to being, in all cases of authorization against the Azure APIs, `https://management.core.windows.net/`. If the provided information was correct and authorization is granted, a JSON-encoded response containing the access token and some additional information is returned to be sent as a header field in all subsequent API calls requiring authorization. The structure of the authorization response is presented in Listing 2.

```
{
    "token_type": "Bearer",
    "access_token": "<token>",
    "expires_in": "<expiration_delta_seconds>",
    "expires_on": "<absolute_expiry_moment>",
    "not_before": "<minimum_absolute_expiry_moment>",
    "resource": "https://management.core.windows.net/"
}
```

Listing 2: Structure of token response.

Operations through the HTTP-based REST APIs

The ARM APIs are fully REST-compliant with all interactions done via HTTP. Each particular resource type has its own URL at which HTTP requests may be performed to issue certain actions. The general format of the resource URLs is listed in Listing 3.

```
https://management.azure.com/subscriptions/<subscription_id>
    /resourceGroups/<resource_group_name>/providers/<provider_name>
    /<resource_type>[/<resource_name>][/operation_name]
```

Listing 3: URL format of an ARM resource on Azure

With each of the respective parameters being:

- subscription_id: the ID of the Azure subscription under which the resource should be created
- resource_group_name: the name of the resource group which contains the resource the operation is being performed on (recall that an ARM resource may not exist outside of a resource group)
- provider_name: the name of the resource provider which handles that particular resource type. All resource provider names have the format 'Microsoft.*', where * is the domain the provider handles (for example, there is a provider named 'Microsoft.Compute' which handles compute resources such as virtual machines, 'Microsoft.Network' for networking resources etc...)
- resource_type: the type of the resource we want to manipulate ('virtualMachines', for example)
- resource_name: if we are performing an operation on a specific resource, we must also provide its name
- operation_name: if performing a specific operation on a resource, we must provide the name of the operation ('redeploy' for a specific virtual machine, for example)

The vast majority of operations exposed through the ARM APIs follow the same basic model of interaction with regards to the HTTP methods used, namely:

- GET: issues an information retrieval request for the resource specified in the URL:
  - if the request URL describes a particular resource given by its name, the JSON-encoded properties of that resource are returned in the body of the response with HTTP 200, with a 404 issued in case the resource does not exist
  - if the request URL describes a whole type of resources, the response will contain a JSON-encoded list of all the individual resources of that type (in the given resource group)
- PUT: issues a creation request for a particular resource whose specific properties are provided via the JSON-encoded body, or, in case the resource already exists, its properties are updated with the provided ones:

  - if the operation takes effect immediately, HTTP 200 is returned, as well as a repeat of the details of the request
  - if the operation takes time to finish, HTTP 201 is returned, as well as a repeat of the details of the request and the URL at which to poll until the operation has finished

- POST: issues an operation request for a particular resource (like powering a virtual machine on/off, for example):
  - if the operation is accepted, HTTP 202 is usually returned, as well as a repeat of the details of the request
  - most operations are modelled as long-running, with a polling URL provided within the response body
  - status code 204 is also possible when there is no need for a response body to be returned

- DELETE: issue a deletion request for a particular resource:
  - if the operation is accepted but it is a long-running operation (like deleting a virtual machine, for example), then HTTP 200 is usually returned alongside a polling URL in the response body
  - if the operation should take effect immediately, then HTTP 204 is usually returned with no response body

API Payload Format

The payloads of the HTTP transactions with the ARM APIs are encoded in JSON format and sent through the bodies of the HTTP requests/responses. The exact contents often represents the properties of the resources that are being queried, created or modified, which are specific to each resource in turn. An example of the set of properties one can expect for an Azure Availability Set, a resource which specifies replication properties for virtual machines, can be found in Listing 4.

```
{
    "id": "<availability_set_id>",
    "name": "<availability_set_name>",
    "location": "availability_set_location",
    "type": "Microsoft.Compute/availabilitySets",
    "platformUpdateDomainCount": 5,
    "platformFaultDomainCount": 5,
    "virtualMachines": "[<list of virtual machines from the availability set>]"
}
```

Listing 4: JSON representation of an Availability Set's properties.

Of special note is the fact that some properties are common for all ARM resources, namely:

- id: unique ID of the resource which is in fact the whole path following the Azure Management DNS name from the URL provided in Listing 3.
- name: name of the resource, which may also be extracted from its ID
- location: the Azure location at which the resource has been deployed in normalized form (ex: 'West US' is encoded as 'westus')
- type: not mandatory for most transmissions as it is directly deductible from the request URL path. The type of the entity may also be provided in the form 'ResourceProvider/resourceType'

Most noteworthy is that the exact details of how the payloads are constructed per operation is not entirely standard. However, as described in the previous section, the following guidelines apply to the structure of the payloads with respect to the set of properties of the resource which constitutes the object of our request:

- sent as-is when a creation, update or operation request is performed for/on a single resource
- received as-is when a GET request is performed on a single resource
- received as members of a list found under the 'values' key whenever a listing is requested (a GET performed on a resource type URL)
- nested within the properties of a different resource when operations are performed on it (ex: virtual subnets may either be defined on their own, or as sub-resources of the virtual networks they belong to)

2.3.2 The Swagger API Description Format

As seen in the previous section, there are a large amount of variables to take into account when implementing a specific ARM API call, with a lot of aspects not entirely consistent throughout the whole space of possible calls. As such, most vendors which offer such broad APIs (1100+ calls for all the providers on their latest API versions), often also publish all of the available API calls in an API description format. In Azure's case, the ARM APIs are documented [8] in a format known as *Swagger* (recently adopted by the 'OpenAPI Initiative' [9] and now known as the 'OpenAPI Specification Format'). Swagger is an API description format based on JSON. Officially, the format only exists as a JSON schema [10] to which all Swagger files must comply, while Swagger itself also has JSON schema-like properties (references to other resources, files etc...). Provided a complete Swagger description of an API, we may find in it the following:

- the base host against which all other requests are made (in our case that will always be `https://management.core.windows.net/`)
- the communication schema (protocol) used (in our case, always HTTP)
- the communication format through which interactions are performed (in our case, always either 'application/json' or 'text/json', encoded as UTF-8)
- the security/authorization mechanism to be used and extra details about it (in our case, all the information presented in the section about authorization)
- the application paths available on the server (detailed in the section about the ARM HTTP-based REST APIs)
- for each path, the available operations to be performed on that path (GET, POST etc...)
- for each operation, all of its associated parameters (URL path parameters, query parameters, request body format)
- for each operation, all of its associated response details (possible status codes, response format)

An extract of the description for the API call which creates/updates a new Availability Set with respect to the latest version of the ARM APIs can be found in Listing 5.

```
"/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/
    providers/Microsoft.Compute/availabilitySets/{name}": {
    "put": {
        "operationId": "AvailabilitySets_CreateOrUpdate",
        "parameters": [{
                "name": "resourceGroupName",
                "in": "path",
                "required": true,
                "type": "string",
            }, {
                "name": "name",
                "in": "path",
                "required": true,
                "type": "string",
            }, {
                "name": "parameters",
                "in": "body",
                "required": true,
                "schema": {
                    "$ref": "#/definitions/AvailabilitySet"
                },
            }, {
                "$ref": "#/parameters/ApiVersionParameter"
            }, {
                "$ref": "#/parameters/SubscriptionIdParameter"
            }
        ],
        "responses": {
            "200": {
                "schema": {
                  "$ref": "#/definitions/AvailabilitySet"
                }
            }
        }
    }
}
```

Listing 5: Swagger excerpt for Availability Set creation operation.

This excerpt has been partially reformatted and some auxiliary details have been left out (such as the 'description' fields of each of the parameters), but the rest is factually accurate. Also of note is that the actual schema for the structure of the request body is as reference to its respective definition within the schema ('#/definitions/AvailabilitySet'), which coincides with the structure presented in Listing 4.

The Swagger description of the ARM APIs has proved invaluable in understanding the features of the API and details of each operation. It is also where the figures of 1100+ individual calls involving 300 or so individual resources were extracted from.

2.3.3 Swagger-based Client Library Generation

Given the thorough Swagger description of the ARM APIs, it may seem almost natural to try to derive as much client library functionality directly from the specifications. As previously mentioned, this is the exact practice Microsoft has employed for the collection of Azure client libraries they offer.

2.3.4 Azure AutoRest

The methodology by which the Azure team generates these libraries follows this basic set of principles:

- write a minimum set of foundation code by hand, which will be later reused in the generated code (this includes functionality such as the authentication, basic wrappers of the standard HTTP library from the language in question, and abstract base classes to inherit from later if in an Object-Oriented setting)
- use the Swagger description to generate the types/datastructures which represent the transmission payloads (these are often referred to as the 'models' and implement some generic interfaces for serialization/deserialization to/from JSON)
- use the resulting models to generate all of the code of the specific operations which use those models (ex: implementing the GET, PUT and DELETE operations based on the model of a virtual machine)

The last two points are handled by a Microsoft project known as 'Azure AutoRest' [11]. AutoRest is written in C# and generates code based on *Razor templates*, a template markup syntax which usually carries the file extension 'cshtml'. The specific details of the generated code are filled into the Razor template programatically and results in the client library code.

Disadvantages of Code Generation

Whilst code generation does bring the advantage of mediating all of the particularities of some API calls, it does also bring some drawbacks.
Firstly and obviously, code generation is inherently imperfect. At the start of the library generation initiative, the first results of AutoRest were practically unusable as the generated code often did not even compile (in the case of the Go, Java and C# libraries), or were incorrect and led to runtime crashes (in the case of the dynamic languages such as Python and Ruby). There were many cases where the AutoRest system and even the Swagger specifications themselves had to be retroactively refactored in order to better facilitate support for each language in part. Even nowadays updates to the specification still uncover some bugs in AutoRest which lead to the libraries being sub-par.
Secondly, the code generated by AutoRest can be extremely hard to read and understand, as the intricacies of generated code are very hard to reason about upon just examining the results. This is most apparent in the case of the dynamic languages, where endless nested if's and returns of control litter the code, especially due to the issue of long running versus non-long running operations. This problem is so pronounced it is even recommended that people turn to reading the Swagger specification directly to understand the operations they are using instead of turning to the code itself.
These issues prompted us to use the Swagger specification as little as possible and instead use Haskell's abstraction mechanisms to write generic code which can be reused across almost all API calls.

Chapter 3

Haskzure's implementation

3.1 The Library's interface

3.2 Base Typeclasses

3.3 Generating the data structures from Swagger

3.4 Details on the implementation of Authentication

3.5 The low-level functions

3.6 The high-level interface

Chapter 4

Using the Library

Chapter 5

Conclusions

Chapter 6

Toolchain and Libraries Used

Bibliography

[1] https://github.com/Azure?query=azure-sdk-for. List of open-source Azure client libraries.

[2] https://github.com/worksap-ate/aws-sdk. Minimal AWS client library with great design.

[3] https://github.com/haskell-distributed/distributed-process-azure. Minimal ASM client library used in Cloud Haskell.

[4] Don Stewart Bryan O'Sullivan and John Goerzen. *Real World Haskell*. O'Reilly Media, December 2008.

[5] Simon Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly Media, July 2013.

[6] https://github.com/bos/aeson. Aeson, the most praised Haskell library for working with JSON.

[7] Miran Lipovača. *Learn You a Haskell for Great Good!* No Starch Press, April 2011.

[8] https://github.com/Azure/azure-rest-api-specs. Swagger API specification of the Azure APIs.

[9] https://openapis.org. Details about the OpenAPI Initiative.

[10] http://swagger.io/specification/. Specification of the Swagger API Description Format.

[11] https://github.com/Azure/autorest. The Azure AutoRest client library generation project.