Facultatea de Automatică și Calculatoare
Programul de Licență:
Calculatoare şi tehnologia informaţiei

Haskzure

Azure Resource Manager Bindings in Haskell

Lucrare de licență

Nashwan Azhari

Conducător științific
conf. dr. ing. Marius Minea

Timișoara
June, 2016

Haskzure: Bibliotecă de Haskell pentru Azure Resource Manager

Nashwan Azhari

June 26, 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

Listings:

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 public 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 manage 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 data structures 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 Data Types* (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 data type 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 data types 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 preinstalled onto Azure and referenced in the application. 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 by 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 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 through which we interact with the APIs, further proving that the Resource Manager API is nothing more than a union of those of 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.
The following is a list of the OAuth2 authorization methods available against Azure AD:

- 'Authorization Code' grant: an authorization method in which the user of the application/service is redirected to Azure AD for authentication, after which Azure passes the application the authorization token via an application callback URL
- 'Client Credentials' grant: this method requires that the application (i.e. the 'client'), be previously registered within the authorization server, with access rights predefined and a given 'client ID' and 'client secret' (passkey) to be used for it to fetch its own authorization token
- 'Resource Owner Password Credentials' grant: this method relies on the user handing the application his/her own credentials and the application (which needs to be previously registered with an associated 'client ID' and permissions), and the application then authenticates with the owner's credentials to be granted a token

The scenario we will focus on is 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 presented 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 merely a 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.

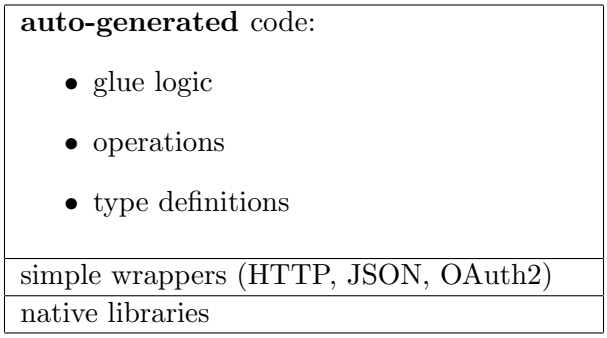
2.3.4 Azure AutoRest

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/data structures 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.

Below is the basic architectural overview of an AutoRest-generated client library in Listing 6.



Listing 6: Architectural overview of AutoRest-generated client libraries.

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. In Listing 7, you may see a simple example of a situation where the code generation engine for the Python client library completely disregards a key-value argument.

```
def __init__(self, id=None, load_balancer_backend_address_pools=None,
        load_balancer_inbound_nat_rules=None, private_ip_address=None,
        private_ip_allocation_method=None, public_ip_address=None,
        provisioning_state=None, name=None, etag=None, **kwargs):
    super(NetworkInterfaceIPConfiguration, self).__init__(id=id, **kwargs)
    self.load_balancer_backend_address_pools = load_balancer_backend_address_pools
    self.load_balancer_inbound_nat_rules = load_balancer_inbound_nat_rules
    self.private_ip_address = private_ip_address
    self.private_ip_allocation_method = private_ip_allocation_method
    # NOTE: the kwarg is ignored and 'subnet' field
    # gets consistently set to None here:
    self.subnet = None
    self.public_ip_address = public_ip_address
    self.provisioning_state = provisioning_state
    self.name = name
    self.etag = etag
```

Listing 7: Python client library generation mishap.

More details on the particular issue listed above can be found on the Github issue tracker for the Python SDK [12].

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

In the following we will provide all the relevant implementation details for Haskzure as well as motivate the general design decisions which were taken.

3.1 Haskzure's design

As opposed to the way Microsoft has taken to auto-generating a large portion of the codebase of the client libraries as described in the section on Azure AutoRest, one of the main purposes of Haskzure was avoiding code generation as much as feasibly possible.
In a nutshell, Haskzure can be viewed as a layered system with the general architecture presented in Listing 8.

high-level monadic API
glue logic
generic operations
auto-generated: data type definitions
native libraries

Listing 8: Architectural overview of Haskzure.

As can be seen, code generation has been limited solely to the generation of the data types representing Azure resources, with all the layers above it being parametrically polymorphic with respect to specific type of resource they are manipulating thanks to the design of the 'Resource' ADT outlined in the next section. This allows for a much more robust system, with the entirety of the code directly interacting with Azure being hand-written.

3.2 Generating the Resource Data Structures

In the following section, we will describe in detail all the steps taken from Swagger definition to Haskell data structure. For this purpose, the 'Availability Set' Azure resource was chosen for its simplistic structure and the ease of demonstration it provides.

3.3 Base Resource Types

The first logical step towards abstraction is, of course, the factoring out of common behavior. In order to facilitate abstraction of the resource attributes shared among all resource definitions, Swagger permits the usage of a special `allOf` field which denotes the 'inclusion' of all of the properties in the referenced schema into the referencing one. For example, the whole definition of an 'Availability Set' is shown in Listing 9.

```
"AvailabilitySet": {
    "properties": {
        "properties": {
            "x-ms-client-flatten": true,
            "$ref": "#/definitions/AvailabilitySetProperties"
        }
    },
    "allOf": [
        {
            "$ref": "#/definitions/Resource"
        }
    ],
}
```

Listing 9: Swagger definition of an Availability Set.

As may be noticed from the above excerpt, the Azure team encapsulated the common properties shared among all Azure resource types in a single, so-called 'Resource' definition. The remaining resource-specific properties are also defined separately in a schema with the same name of the resource type but with 'Properties' appended to it, which we will discuss more thoroughly in the next next section.

The exact structure of the common properties of defined in the base Resource may be seen in Listing 10

```
"Resource": {
    "properties": {
        "id": {
            "readOnly": true,
            "type": "string"
        },
        "name": {
            "readOnly": true,
            "type": "string"
        },
        "type": {
            "readOnly": true,
            "type": "string"
        },
        "location": {
            "type": "string"
        },
        "tags": {
            "type": "object",
            "additionalProperties": {
                "type": "string"
            }
        }
    },
    "x-ms-azure-resource": true
}
```

Listing 10: Swagger definition of the base 'Resource' type.

From the definition of the base Swagger Resource, we may conclude that the following fields are shared among all resources:

- the resource ID, which, as mentioned, uniquely identifies resources by virtue of the full resource path following the base host of the URL we are making API calls to
- the simple name of the resource
- the resource's type, which is formed by joining the type of the resource with the resource provider in charge of it (for example 'Microsoft.Network/virtualNetworks' for virtual network resources)
- the normalized Azure location the resource is located at
- additional key-value pairs representing tags for easy custom categorising of resources

This way of defining resource types as a Swagger 'mixin' is something done consistently throughout all of the ARM Swagger specifications, with the properties specific to the resource in question being defined as a separate entity each time. Based on this simple specification, we may go ahead and define the base 'Resource' ADT of our library, shown in Listing 11.

```
data Resource a = Resource {
    -- | resID is the ID of the Resource:
    resID         :: String,

    -- | resName is the Name of the Resource:
    resName       :: String,

    -- | resGroup is the Name of the Resource Group in which the 'Resource' lives.
    resGroup      :: String,

    -- | resLocation is the Location of the Resource:
    resLocation   :: String,

    -- | resType is the Type of Resource:
    resType       :: String,

    -- | resProperties are the properties of the Resource:
    resProperties :: a
} deriving (Show)
```

Listing 11: Definition of Haskzure's base 'Resource' type.

Of special note is the type of the 'resProperties' field. The 'Resource' data type is parametrically polymorphic with regards to the properties it contains, no matter the resource type for which those may be. This will allow for the base functions to be generic with regards to the resource type they are operating on as will be outlined in a later section. Additionally, we have defined a 'resGroup' field for conveniently specifying the resource group in which the resource in question has been deployed.

3.3.1 Generating Resource Properties Data Types

As seen in the architectural overview presented in Listing 8, the only code generated based on the Swagger specification is that of the definitions of the data structures representing the properties of a resource on Azure. These are the data structures which will later be directly serialized/deserialized to/from the payloads of all the HTTP interactions with ARM.

The basic outline for data type generation based on the Swagger definition of a 'model' can be summarized as follows:

- the resulting data type will have the same name as the Swagger definition, but with the 'Properties' tag removed
- the resulting data type will have a single value constructor, whose name coincides with that of the data type in accordance with Haskell idioms
- the resulting value constructor will be of record type, with all of the fields representing each attribute in the Swagger definition
- each field's label features as a prefix the name of the data type so as to avoid naming collisions throughout the generated code

Following the above rules, we may now perform a simple case study on a sample resource type, namely the 'Availability Set' resource we have presented in Listing 9. The Swagger definition of the 'AvailabilitySetProperties' of can be found in Listing 12.

```
"AvailabilitySetProperties": {
    "properties": {
        "platformUpdateDomainCount": {
            "type": "integer",
            "format": "int32",
            "description": "Gets or sets Update Domain count."
        },
        "platformFaultDomainCount": {
            "type": "integer",
            "format": "int32",
            "description": "Gets or sets Fault Domain count."
        },
        "virtualMachines": {
            "type": "array",
            "items": {
                "$ref": "#/definitions/SubResource"
            }
        }
    },
}
```

Listing 12: Swagger definition for the properties of an Availability Set.

The above Availability Set properties will result in the generation of a data structure with the formal definition shown in Listing 13.

```
data AvailabilitySet = AvailabilitySet {
    availabilitySetPlatformUpdateDomainCount :: Int,
    availabilitySetPlatformFaultDomainCount :: Int,
    availabilitySetVirtualMachines :: [SubResource]
 } deriving (Show, Eq)
```

Listing 13: Generated Haskell definition of an Availability Set.

As one might imagine, the level of technical risk undertaken in the generation of simple data structures is relatively small one, as opposed to the generation of a lot of code with an associated logic to it. Considering the limited scope of these data structure definitions with regards to the scope of the whole project, it is easy to deduce the benefits of only extracting the data definitions from the Swagger specification.

3.3.2 Generating Serialization Instances

Once the generation of a data structure representing the properties of a resource has been performed, we must then provide instance implementations for all of the typeclasses required in order to succesfully use said data structure in interactions with Azure. Most notable are the typeclasses required to be implemented by *Aeson* (a Haskell library for working with JSON-encoded data), namely ToJSON and FromJSON, as well as an instance for the Monoid typeclass which is commonly used in Haskell.

ToJSON

ToJSON is the Aeson typeclass implemented by all types which are serializable to JSON. Its definition is very simplistic at heart and can be seen in Listing 14.

```
class ToJSON a where
    toJSON     :: a -> Value
    toEncoding :: a -> Encoding
```

Listing 14: Definition of Aeson't ToJSON typeclass.

We may see the existence of two main functions defined within the ToJSON typeclass. `toJSON`, the first, is a function which takes a value of the type in question and returns a value of type `Value`, a type defined in the Aeson library which has its own ToJSON instance. `toJSON` may thus be viewed as a function which encodes to JSON in two separate steps by first 'transcoding' the value of the given type into one of type `Value`, which is then later actually serialized to JSON. The other, `toEncoding`, is the more efficient variant, it taking a value of the type in question and encoding it directly to JSON. By default, a generic `toEncoding` which makes use of `toJSON` is used if only the latter is provided. Of course, the preferred option would be making use of `toEncoding` for all of our serialization needs in order to reap the added performance benefits.
Fortunately, Aeson includes helper functions for making both a `toJSON` and a `toEncoding` function provided the type we are instantiating is an instance of a Haskell typeclass called `Generic`, which, even more fortunately for us, is also automatically derivable with the language extension 'DeriveGeneric' available in GHC. The signature of `genericToEncoding`, the function which, when partially applied to a set of encoding `Options` results in the definition of `toEncoding` we desire, is shown in Listing 15.

```
genericToEncoding :: (Generic a, GToEncoding (Rep a)) => Options -> a -> Encoding
```

Listing 15: Signature of Aeson's `genericToEncoding`.

The next problem we were faced with was the proper choosing of the encoding `Options` for our purposes. Recall that when we generated the properties data structures, we intentionally prepended the name of the data type in order to avoid naming collisions. As such, we must take care to choose the correct encoding options which will lead to the field prefix being removed when serializing to JSON. Furthemore, the options shall be clearly dependand on the name of the data type, and thus a custom set of them must be created for every data type we want to instantiate to ToJSON. The code snippet outlining the process of generating the resource-specific `Options` can be seen in Listing 16.

```haskell
baseAesonOptions :: Options
baseAesonOptions =  Options {
    fieldLabelModifier = id,
    constructorTagModifier = id,
    allNullaryToStringTag = True,
    omitNothingFields = True,
    sumEncoding = defaultTaggedObject,
    unwrapUnaryRecords = False
 }

mkFieldLabelPrefixRemoveModifier :: Name -> String -> String
mkFieldLabelPrefixRemoveModifier n = lowerFirst . drop (length $ showName n)
    where lowerFirst (c:cs) = toLower c : cs

mkEncodingOptions :: Name -> Options
mkEncodingOptions n = baseAesonOptions {
    fieldLabelModifier = mkFieldLabelPrefixRemoveModifier n
 }
```

Listing 16: The Aeson encoding `Options` used.

The exact selections for the options are motivated as follows:

- fieldLabelModifier: a function which describes the transformation to be applied to the labels of the data structure's fields. This is where we create a custom function for each data structure using `mkFieldLabelPrefixRemoveModifier`
- sumEncoding: parameters on how to encode sum data types. Not applicable to our case, as all of our generated data structures have a single value constructor
- constructorTagModifier: a function to be applied to the tags made from the names of the value constructors. Not applicable so the identity function `id` is used
- allNullaryToStringTag: specifies whether to encode values of types with all nullary value constructors (such as `Bool`) with just the names of the constructors with `constructorTagModifier` applied to them. Set to `True` for simple enum-like types
- omitNothingFields: specifies wether to omit `Nothing` values in case our fields are of type `Maybe a`. Not applicable, but set to `True` for futureproofing
- unwrapUnaryRecords: specifies whether or not to unrap the value of a type with a single value constructor of record type, containing a single record field (such as newtypes). Not applicable, set to the default of `False`

With that, the full definition of the Template Haskell function `toJSONInst`, as well as a basic example of how it is used with the 'AvailabilitySet' ADT, may be found in Listing 17.

```haskell
toJSONInst :: Name -> Q [Dec]
toJSONInst name =
    [d| instance ToJSON $( conT name ) where
            toEncoding = genericToEncoding $ mkEncodingOptions name
      |]

-- in a separate file:
{-# LANGUAGE TemplateHaskell #-}

toJSONInst ''AvailabilitySet
```

Listing 17: The `toJSONInst` Template Haskell function.

FromJSON

FromJSON is the Aeson typeclass representing data types deserializable from JSON. It contains a single member, namely `parseJSON`, as seen in Listing 18.

```haskell
class FromJSON a where
    parseJSON :: Value -> Parser a
```

Listing 18: Aeson's FromJSON typeclass.

As we can see, `parseJSON` is polymorphic in its return type, as well as returning the value wrapped inside the `Parser` monad Aeson defines. As such, we must make use of Aeson't library functions in order to write our `parseJSON` definition, which effecively 'decodes' an already parsed Aeson `Value` into the data structure of type 'a'. Fortunately, there is also a `genericParseJSON` available for use, which has the signature presented in Listing 19.

```haskell
genericParseJSON :: (Generic a, GFromJSON (Rep a)) => Options -> Value -> Parser a
```

Listing 19: Aeson's FromJSON typeclass.

With the above, and provided the same set of `Options` we used to create the `toEncoding` function (Aeson performs a 'lookup' when decoding, so the same `Options` used the encode the fields are the ones needed to see how to expect the fields to be encoded), we may easily define `parseJSON`.
More interesting, however, is what happens when some fields of the data structure are not present in the payload being decoded. In this case, Aeson's default behavior is to throw an error which leads to a program crash. To prevent stability issues and provide sensible defaults for empty fields when decoding, we must implement our own fallback mechanism. This is illustrated in Listing 20.

```haskell
-- | Merges two Aeson 'Value's via 'Map' 'Monoid' instance.
(<+>) :: Value -> Value -> Value
Object x <+> Object y = Object (x <> y)
_ <+> _ = error "<+>: merging non-objects"

-- | Returns new parsing function which will use the specified 'Pair's as
-- default values for the data type being decoded's fields.
withDefaults :: (Value -> Parser a) -> [Pair] -> Value -> Parser a
withDefaults parser defs js@(Object _) = parser (js <+> object defs)
withDefaults _ _ _ = empty
```

Listing 20: `withDefaults` Parser merging function.

As can be seen, `withDefaults` employs the Monoid instance of Map (which is used in the underlying definition of Aeson `Value` type), to unify the object getting parsed with an object containing a list of pairs of field names and the respective default value. The union of the parsed and default objects occurs before the parsing into the destination data structure. As mentioned, the default object is constructed from the list of Aeson `Pairs` within which we must refer to fields with the same name we'd expect them to be found in the JSON payload (in this case, without the data type name prefix we put on all record fields). This is done through the Template Haskell function `mkFromJSONPairs` whose implementation we won't detail. Either way, `mkFromJSONPairs` is the main driver for the `fromJSONInst` Template Haskell function whose expected outcome for the 'AvailabilitySet' resource type shown in Listing 13 should be consistent with the one presented in Listing 21.

```
instance FromJSON AvailabilitySet where
    parseJSON = (genericParseJSON (mkEncodingOptions "AvailabilitySet")
                `withDefaults` [
                    ("platformUpdateDomainCount" .= (mempty :: Int)),
                    ("platformFaultDomainCount" .= (mempty :: Int)),
                    ("virtualMachines" .= (mempty :: [SubResource]))
                ]) :: (Value -> Parser AvailabilitySet)
```

Listing 21: AvailabilitySet FromJSON instance.

All of the default values supplied are actually the result of calling `mempty` from the Monoid class for the respective type of the field. The exact details of how `fromJSONInst` is implemented are beyond the scope of this writing, but instead may be found better documented in the Haddock documentation of Haskzure.

Monoid

Monoid instances for the resource data types serve two main purposes:

- they are required in the case of nested Azure resources in order for their `mempty` to be used in the FromJSON instantiations as has been outlined in Listing 21
- they are generally useful for both:
  - making default empty values of the resources data types which may then be updated with what the desired characteristics are
  - taking two resource configurations and `mappend`ing them together

These instances are made using the facilities of the `generic-deriving` [13] package with the same constraint of the data type needing to be an instance of `Generic` still applying. The full definition of the `monoidInst` Template Haskell function is outlined in Listing 22.

```
monoidInst :: Name -> Q [Dec]
monoidInst name =
    [d| instance Monoid $( conT name ) where
            mempty = memptydefault
            mappend = mappenddefault
      |]
```

Listing 22: The `monoidInst` Template Haskell function.

As can be observed, we rely on `memptydefault` and `mappenddefault` in our Monoid instance, which provide sensible default implementations for data structures which are instances of `Generic`. Finally, we may quickly and easily create a function which dispatches to the Template Haskell functions

for creating instances for both ToJSON and FromJSON, as well as the Monoid one. The resulting function is the one used on all of the data types which were generated in the previous layer, and may be see in Listing 3.3.2.

```
mkAllInsts :: Name -> Q [Dec]
mkAllInsts name = fmap concat $ mapM ($ name) [monoidInst, toJSONInst, fromJSONInst]
```

Listing 23: Definition of the `mkAllInsts` Template Haskell function.

3.4 OAuth2 Authorization

With the resource data structures defined, we may now focus on the *OAuth2* authorization. Regardless of the authorization flow used, the response should be the same and consistent with the structure shown in Chapter 2 in Listing 2. As mentioned, in Haskzure specifically, the only currently supported OAuth2 flow is the 'Client Credentials flow'.

3.4.1 The Credentials ADT

Presuming Haskzure has been properly setup within an instance of Azure Active Directory (a straightforward process done via the Azure online user interface), the set of information required in order to succesfully authenticate using the 'Client Credentials' OAuth2 flow against Azure AD has been aggregated into a data structure named `Credentials`, whose definition can be found in subsection 3.4.1.

```
data Credentials = Credentials {
    tenantId       :: BS.ByteString,
    clientId       :: BS.ByteString,
    subscriptionId :: BS.ByteString,
    clientSecret   :: BS.ByteString
 } deriving Show
```

Listing 24: Haszure's `Credentials` ADT.

The record fields of the `Credentials` ADT all play a part the authroization process, and namely:

- 'tenantId': the unique ID of the Azure Active Directory against which to authenticate, as will be presented later
- 'clientId': unique ID number of the application (in this case Haskzure). This is obtained when the application is registered within the Azure AD
- 'clientSecret': secret string to be used to authenticate the client. This is automatically generated and also obtained when the application is registered within the Azure AD
- 'subscriptionId': the ID of the subscription under which the operations will be performed

3.4.2 The Token ADT

Considering the well-defined properties of an OAuth2 authorization token, we may go ahead and define the `Token` ADT to be used in all functions requiring authorization. Also considering that the response for succesfull token aquisition is the same (namely, the JSON structure presented in Listing 2), we also went ahead and declared a FromJSON instance for `Token` so as to be able to directly deserialize the token response into it. Both the definition of `Token` and its FromJSON instance may be found in subsection 3.4.2.

```haskell
data Token = Token {
    token     :: BS.ByteString,
    tokenType :: BS.ByteString,
    expiresOn :: Integer
 } deriving Show

instance FromJSON Token where
    parseJSON (Object o) = Token <$>
                           o .: "access_token" <*>
                           ((o .: "expires_on") >>=
                               (return . (read :: String -> Integer))) <*>
                           o .: "token_type"
    parseJSON _ = fail "Token must be deserialized from a JSON object."
```

Listing 25: Haszure's `Token` ADT and its FromJSON instance.

3.4.3 Token Fetching and Refreshing

As described in the Theoretical Overview chapter, the process of fetching a token for the 'Client Credentials' OAuth2 grant is nothing more than performing a URL-encoded POST with the appropriate query parameters as shown in Listing 1. Thus the most crucial part is the correct forging of the query parameters by extracting what is required from the `Credentials` data structure. The exact process of creating the query parameters for the token request is shown in subsection 3.4.3.

```haskell
mkTokenRequestParams :: Credentials -> QueryParams
mkTokenRequestParams creds = [
    ("resource", "https://management.core.windows.net/"),
    ("grant_type", "client_credentials"),
    ("client_id", clientId creds),
    ("client_secret", clientSecret creds)
 ]
```

Listing 26: `mkTokenRequestParams` query parameter builder function.

Token requests must be directed to the OAuth2 path of the Azure AD's login URL (`https://login.microsoftonline.com/<tenant_id/oauth2/token`, where 'tenant_id' is the ID of the Azure AD instance we are authenticating against, thus the need for its supplying from the `Credentials` data structure). The rest of the process of actually performing the URL-encoded POST and deserializing the response s as straightforward as is to be expected.

3.4.4 Authorizing with the Token

After the `Token` has been succesfully recieved, using it for authorization involves nothing more than adding an HTTP header to all subsequent API calls. All encryption is performed by TLS (no ARM API calls may come through plain HTTP), so the real purpose of the OAuth2 token is to provide a verifiable sign of owning the necessary permission to perform the requested action.
In Haskzure, we made use of the `http-client` [14] &co packages in which we have a `Request` data type which is manipulated in order to set various attributes of the upandcoming request. The operations required in order to set the appropriate request parameters may be found in subsection 3.4.4.

```haskell
mkResourcePath :: String -> Resource a -> String
mkResourcePath subId res = printf ((mkResourceTypePath subId res) ++ "/%s") (resName res)

mkTokenHeader :: Token -> Header
mkTokenHeader t = (hAuthorization, (tokenType t) `BS.append` " " `BS.append` (token t))
```

Listing 27: Necessary additions to the `Request` data type for authroization.

Specifically, the above code adds a header with the key 'Authorization', and the value 'token_type token', where the format parameters correspond to the type of the OAuth2 token as well as the token itself which are extracted from the `Token` data structure.

3.5 The Base Functions

With the data structures and token retrieval parts taken care of, the next logical layer of the library consists of the base functions which offer the basic set of functionality needed for interfacting with Azure. In a nutshell, the base set of functions include the Azure resource CRUD functions, as well as a handful others for performing specific operations on some resource types (like restarting a virtual machine, for example). In the following we will detail the building of the base operations layer.

3.5.1 Building the Resource Request URL

The first step of performing an ARM request is ensuring that the request is being made to the appropriate URL, as a lot of information is transmitted to the ARM APIs through the application path being accessed. Some details of the format of the ARM resource paths may also be found in the Technical Overview section in Listing 3, which is duplicated below in Listing 28.

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

Listing 28: URL format of an ARM resource on Azure

Of special interest this time is determining where all the parameters should be provided. The only required piece of information from the `Credentials` data structure is the 'subscription_id' path element, whilst all the rest of the elements may be directly extracted from the `Resource a` we are operating on (which also explains the need for the `resGroup` field of the Resource data type).
The exact implementation of the URL building process is trivial, involving nothing more that some smart string formatting.

3.5.2 Function Definitions

All of the base functions (`createOrUpdate`, `get`, `delete`) follow the same general behavior outlined below:

- fetch a fresh API token from Azure AD
- build the request URL based on the parameters of the Resource being operated upon
- create the appropriate request body by serializing the provided Resource (as is the case for `createOrUpdate`, for example)
- perform the request to the appropriate URL over HTTPS
- await the server's response, which may in some cases last minutes depending on the scale of the request being performed
- if needing to return something, properly deserialize the response body into the required type

With that general methodology clear, the exact implementation details of the CRUD functions are unnecessary. Their signatures, however, may be found in subsection 3.5.2.

```
get            :: (FromJSON a) => Credentials -> Resource a -> IO (Resource a)
createOrUpdate :: (ToJSON a) => Credentials -> Resource a -> IO ()
delete         :: Credentials -> Resource a -> IO ()
```

Listing 29: Signatures of the base functions for manipulating resources.

3.6 The high-level interface

Chapter 4

Testing

There are two main areas testing should be focused: the data type and instance generating Template Haskell functions, as well as some integration tests which actually interact with Azure.
The data type and instance generation code is, albeit kept to a minimum, an important testing area as issues in the definition of the resource data types may lead to a host of problems, especially in the case of the Aeson typeclass instance generation functions. As such, Haskzure has a small suite of QuickCheck tests for validating that the instance generation code is performing as expected.
On the other hand, the are no fully-blown integration tests available, but a proposed solution of setting up an integration testing framework will be presented.

4.1 Validation of Instance Generation Code through QuickCheck

In this section, we will describe how validation of the instance generation code is performed using QuickCheck behavioral tests on mock data structures. We'd like to be able to test the generated code as thoroughly as possible to ensure encoding/decoding is done properly. In this sense, we may identify the following major aspects we'd expect of the generated definitions:

- encoding and the decoding should yield a result identical to the input data structure (encode/decode symmetry)
- encode/decode symmetry should also apply if the type of a field of a data structure is another Azure resource data structure

Whilst these may seem like properties which should be intrinsic to the Aeson library itself, the fact that the Template Haskell functions generating our ToJSON and FromJSON instances are parametrized over the name of the data type means that there may easily be some discrepancy between the encoding and decoding process which we'd like to actively test for. As an addded benefit, the testing of the Aeson instance generators also entitles testing the Monoid instance generator as will be seen later.

4.1.1 QuickCheck

QuickCheck is a Haskell library designed to facilitate property-based testing on randomly generated input. Most of QuickCheck's power is drawn from the ability to instantiate types to QuickCheck's `Arbitrary` typeclass in order to be able to test functions operating on values of that data type. The full definition of the `Arbitrary` typeclass may be found in subsection 4.1.1.

```
class Arbitrary a where
    arbitrary :: Gen a
```

Listing 30: Definition of QuickCheck's `Arbitrary` typeclass.

As may be noted, the return type of `arbitrary` is a value wrapped in QuickCheck's Gen monad within which arbitrary values are generated. It is usually straightforward to provide an `Arbitrary` instance for a data type, as you would just have to lift the value constructor into the Gen monad and apply it to `arbitrary` values of the types the data type is composed of.

For the Availability Set data type listed in Listing 13, for instance, the `Arbitrary` instance would look something along the lines of what is shown in Listing 4.1.1.

```
instance Arbitrary AvailabilitySet where
    arbitrary = AvailabilitySet <$>
                    (arbitrary :: Int) <*>
                    (arbitrary :: Int) <*>
                    (arbitrary :: [SubResource])
```

Listing 31: `Arbitrary` instance for the AvailabilitySet data type.

Fortunately, we may reuse some of the data type introspection code from the FromJSON instance generation Template Haskell function to write a new function which generates `Arbitrary` instances. More details on this helper function may be found by consulting the project's Haddock documentation.

4.1.2 Validating the Aeson Instance Generation

Provided an `Arbitrary` instance for our datatype, we can now write a simple QuickCheck property to ensure the symmetry of encoding and decoding. The property applicable to the Availability Set resource shown in Listing 13 , which merely compares the result of applying Aeson's `endcode` and `decode` functions with the original AvailabilitySet value, can be seen in subsection 4.1.2.

```
{-# LANGUAGE TemplateHaskell #-}

arbiInst ''AvailabilitySet

prop_AvailabilitySetAesonSymmetry :: AvailabilitySet -> Bool
prop_AvailabilitySetAesonSymmetry :: d =
    ((fromJust . (decode :: BSL.ByteString -> Maybe AvailabilitySet) . encode) d) == d
```

Listing 32: QuickCheck property regarding symmetry of JSON encoding/decoding.

To be frank, the data type we are testing against does not matter, as the desired property will be validated/invalidated just the same on a data type actually representing an Azure resource or on a dummy data type. What does matter, however, is the proper covering of the case of nested data types (in which the field of one type is of another complex data type). As such, among there are two tests in which dummy types, some nesting others, have the ToJSON and FromJSON instance generation functions used on them and a property similar to the one shown above defined for each of the data types. QuickCheck then randomly generates a set amount of values of said data types and verifies the property for each input and reports back the result.

Baseline Test

Naturally, the Aeson encode/decode symmetry property is entirely dependant on the implementation within the library to be valid. In order to verify that issues or not on Aeson't side, there is a side-test in which the same property is applied for Aeson's `Value` type for which an `Arbitrary` instance was written by hand. In the event there be some issue with Aeson's encode/decode operations, this regression-like check will properly alert to it.

4.2 Proposed Integration Testing Setup

Property-based testing is great when a need of certainty that a specific code behavior is consistently shown, however it does not guarantee that the property is compliant with what Azure expects. As such, the only real way to be sure that both the generated data types and their instances are correct is to actually attempt to interact with the ARM APIs.
The result would be the equivalent of an integration testing suite which may be ran periodically during Haskzure's lifecycle. The ideal target layer from the architectural overview presented in Listing 8 would the layer with the generic operations, as it is both the first layer to be directly interacting with Azure, as well as the layer where the granularity of operations is large (basically, at that layer each function has a direct correspondence to an HTTP method).
A good plan for the proposed integration testing suite would be as follows:

- decide the set of properties for the tested operations (one for creation, another for deletion etc...). These should be paramterized over the exact data type they are operating, as are the definition of the base operation within the library
- for each of the properties, write a Template Haskell function which generates properties of that type provided the name of the data structure representing the type of resource being tested
- a list of data type names for which properties should be generated and tested should be provided in order to start the testing process knowing what is being tested

The creation of an integration testing suite is invaluable to ensuring Haskzure actually communicates cleanly with Azure, regardless of the exact implementation details.

Chapter 5

Using the Library

5.1 Installation and Importing

Haskzure may be easily installed using a combination of git and the cabal configuration management system. Having Haskzure installed is as simple as running the shell statement outlined in Listing 33.

```
git clone https://github.com/aznashwan/haskzure &&
    cd haskzure && cabal configure &&
    cabal install
```

Listing 33: Shell commands for installing and configuring Haskzure.

After installation, all of the following modules should be exposed and ready for importing:

- `Cloud.Haskzure.Core` defines all of the core components of Haskzure which present most of the library's functionality. This includes:
  - the `Credentials` and `Token` authorization datatypes
  - the `getToken` function for fetching an authorization token
  - the Resource and SubResource ADTs with all of their constructors
  - the base set of operations, including the generic Azure resource CRUD functions
- `Cloud.Haskzure.Gen` defines all of the Template Haskell code in charge with resource data type and instance generation including:
  - the `toJSONInst` and `fromJSONInst` Aeson typeclass instance generators
  - the `monoidInst` Monoid instance generator
  - general Template Haskell utilities for data type introspection with the purpose of creating instance definitions (most notably `recordFieldsInfo`)
- `Cloud.Haskzure.Resources` contains the definitions of all of the data types which represent resources on Azure (such as the `VirtualNetwork` or `VirtualMachine` data types)

Chapter 6

Conclusion

The resulting system, whilst not having a scope as broad as the large auto-generated libraries Microsoft offers for other languages, unequivocally shows that simple abstraction features available in the language can go a long way into keeping the codebase small and maintainable. The conciseness achieved by limiting the amount of generated code just to data type declarations leads to much cleaner client library implementations in the long term.

Bibliography

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

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

[3] The Distributed Haskell team. Minimal asm client library used in cloud haskell. https://github.com/haskell-distributed/distributed-process-azure.

[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] Bryan O'Sullivan. Aeson, the most praised haskell library for working with json. https://github.com/bos/aeson.

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

[8] The Azure Team. Swagger api specification of the azure apis. https://github.com/Azure/azure-rest-api-specs.

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

[10] The OpenAPI Initiative. Specification of the swagger api description format. http://swagger.io/specification/.

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

[12] Nashwan Azhari. One of the many issues of the autorest generated sdks. https://github.com/Azure/azure-sdk-for-python/pull/543.

[13] José Pedro Magalhães. The generic-deriving library project. https://github.com/dreixel/generic-deriving.

[14] Michael Snoyman. The http-client library project. https://github.com/snoyberg/http-client.