Experience Report: Haskell as a Reagent

Results and Observations on the Use of Haskell in a Python Project

Iustin Pop
Google Switzerland
iustin@google.com

Abstract

In system administration, the languages of choice for solving automation tasks are scripting languages, owing to their flexibility, extensive library support and quick development cycle. Functional programming is more likely to be found in software development teams and the academic world.

This separation means that system administrators cannot use the most effective tool for a given problem; in an ideal world, we should be able to mix and match different languages, based on the problem at hand.

This experience report details our initial introduction and use of Haskell in a mature, medium size project implemented in Python. We also analyse the interaction between the two languages, and show how Haskell has excelled at solving a particular type of real-world problems.

Categories and Subject Descriptors D.2.12 [*Software engineering*]: Interoperability; D.3.2 [*Programming languages*]: Language Classifications—Applicative (functional) languages

General Terms Experimentation, Languages

Keywords Haskell, Python, Ganeti, System administration

1. Introduction

For the past year, our team has developed[1] and started to use a set of tools implemented in Haskell to solve a specific category of problems that were not best expressed in an interpreted language. This required learning a new style of programming, becoming familiar with the tool-chain (compiler, profiler, documentation tools, etc.), investigating the available libraries and making sure that our new tools inter-operate well with the Python code. At the end, the question was: is the extra effort needed for maintaining code written in two languages justified? Do we get any advantage out of combining two high-level, but quite different, languages?

As we try to show in this paper, in our experience the answer is affirmative, sometimes in non obvious ways. Haskell's strong type system contrasts markedly from Python's "laissez-faire" approach to types, and its cheap-persistence model is the opposite of Python's cheap-modification one, while both are in the same high-level language category where complex data-modification pipelines are readily available. Such diametrically opposite views on some topics were very good at highlighting differences, and neither language was delegated to the 'low-level' versus 'high-level' status.

We have observed gains from simply having two different languages exercise the same API/RPC endpoints; in our case, this meant that the effort spent to standardise the message types (useful for Haskell) can lead to a more sound framework on the Python side. Prototyping the same algorithm in both languages led to a better understanding and in a few cases optimisations in one language can be carried to the other side.

Not all was good, however; a few bumps appeared along the way, in the form of small issues with availability of libraries, performance for some operations, compatibility between different versions of the base libraries and the higher difficulty of advanced programming techniques in a functional language.

1.1 Our contribution

Past ICFP experience reports have focused on either conversion of software from an imperative language ([Newton and Ko 2009]), or on using functional programming for an entire project ([Sampson 2009]). Furthermore, reports on the use of functional programming refer in general to either use in research institutes and universities ([Cuoq et al. 2009] and [Balat et al. 2009]) or in commercial software development teams (e.g. [Sampson 2009]).

We believe our use of Haskell in combination with Python in an already existing, mature project and in the context of system administration represents a different view on the use of functional programming.

2. The team and the project

Our team is part of Google's corporate IT system administration group, dealing with administration of virtual machines. While most of us have strong Python development skills, and everyone is familiar with other system administration languages and tools, we are not, per se, a software development team. Rather the development activities are 'demand-based' and geared towards automation of system administration.

As part of our work, we have developed Ganeti (`http://code.google.com/p/ganeti/`), a management tool for clusters of virtual machines (e.g. Xen, KVM). Our team has been working on the project since 2006, open-sourced it in September 2007, and it was (before the introduction of the Haskell component) written almost entirely in Python, with just some small bits of shell and other languages, mostly for the build system. The objective of Ganeti is to enable easy management of clusters created from off-the-shelf hardware, without requiring custom or expensive storage or net-

[1] As described later in the paper, while the actual Haskell code has a single author, the entire team has participated in the design and testing of these tools, and therefore the paper uses the 'we' pronoun

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.

work gear. As such, we use a quantity of other open-source software for managing the physical resources.

For storage management we use DRBD[2], a software solution for over-the-network RAID1 storage. Using RAID1 (mirroring) means that each virtual disk resides on two physical machines, called the primary and secondary machine respectively. A virtual machine can cheaply switch between these two machines (if the other required resources, e.g. memory, are sufficient), an operation called *failover*. Switching a virtual disk from the machine pair `(A,B)` to `(A,C)`, an operation called *relocation*, requires copying the entire disk data from `A` to `C` and thus it is costly.

This dependency of each virtual machine on two physical machines means that the placement algorithm is not as straightforward as in solutions using external storage, where any physical machine can access the entirety of all storage; therefore, we had to develop tools that can automate the layout computation in order to best use the resources of each physical machine.

2.1 Layout policies

We anticipated somewhat early in the development of Ganeti that the actual policies for the layout of virtual machines across (pairs of) physical machines might differ based on site policies and thus decided that the actual policy should be left to external scripts, while Ganeti itself should just implement the mechanism. Thus, we have a documented API (called the IAllocator API) for things like *given cluster state X, on what pair of machines should new virtual machine with specifications Y be placed?*. Note that this works the same way for non-mirrored storage, where we simply allocate on a single physical machine (in a non-redundant setup). The recursive application of this problem is how many virtual machines can we allocate on a cluster before we violate site policies or run out of physical resources.

The allocation problem is also present in a slight different version: if a physical machine needs to be removed from the cluster (e.g. due to hardware failures, or any other reason), the virtual machines which live on this particular machine need to be relocated to another member of the cluster. The question is expressed as: *given cluster state X, we want to move a virtual machine from physical machine pair (A, B) to (A, x); what is the best choice for x?*. We use the same API as above.

A third related problem is computing the 'optimal' layout of the current cluster. This means responding to the question: *given current cluster state X, with physical machine list N and virtual machine list I, how should we relocate the virtual machines for a better layout?*. This question is not encoded in the IAllocator API, but we can both extract the cluster state and instruct changes in the layout via other Ganeti APIs.

A final note is that the layout of virtual machines across the physical machine pairs has both optimisation aspects (e.g. even load distribution) and hard constraints. One important such constraint is that if a physical machine fails, all its peers must have enough free memory to failover and run the now offline instances. In other words: *for a given physical machine A and its hosted virtual machines I_i, each living on the machine pair (A, x_i), does each machine x_i have enough free memory to accommodate I_i?*. We call this *N+1 redundancy*, and we flag in our verification routines any physical machines that fail this check.

2.2 Open APIs

Another key decision early in the history of Ganeti was that, as much as possible, our APIs and data formats should be language agnostic. Thus, we moved (for the inter-node RPC) from the original Twisted[3]-based RPC and serialisation format to HTTP and JSON, and for data storage from Python Pickle format to (again) JSON[4].

While this decision was not made with any specific purpose in mind, it did help during the development of Ganeti as such data formats are more stable and very easy to use/modify even from the shell. Later, it was one of the key factors that allowed the use of Haskell.

2.3 Introduction of Haskell

The introduction of Haskell in the project was somewhat accidental. The capabilities of Ganeti itself were growing and it was able to manage bigger and bigger clusters; but the layout algorithms were still very weak. Thus, at the end of 2008 the author of the paper started working towards solving an independent (at that time) problem, that is the automated computation of the changes needed in cluster state to solve *N+1* check failures.

The initial version of the algorithm attempted a brute-force search over a limited subset of the solution space, and thus the Python implementation was very slow. It was also unwieldy, since throw-away copies of the data are expensive in Python and undoing modifications in a generic and safe way is not simple.

The next step was intended to be both a learning experience and a language comparison exercise: how would such an algorithm, the simulation of cluster state changes when virtual machines are being relocated, look in a functional language? People familiar with functional programming might recognise such an algorithm as a good fit at once; for the author it took a while until he was convinced that indeed such modelling is easier to achieve in a purely functional way, using persistent data structures.

2.3.1 Time-line

N+1 solver Initial forays into the Haskell implementation of the above algorithm proved successful, and after a couple of weeks we had a tool to automatically compute the solution list (a set of *move virtual machine I from `(A, B)` to `(C, D)`* commands) needed to solve the *N+1* failures in most of the usual cases. The algorithm itself was still rough, and due to its brute-force nature it was very limited in capabilities. One of the biggest limitations was that it wasn't able to tell if it will ever manage to find a solution and thus it tried to explore the whole solution space (which is too big to compute), thus preventing its use in a fully-automated way.

Cluster balancer The *N+1* solver already provided the infrastructure needed for importing data from Ganeti, so writing a new algorithm was a much smaller effort. Thus, the next goal was another missing piece of the Ganeti infrastructure, a generic cluster balancer that takes the state of the cluster and computes "the next best" state.

The new algorithm is no longer a brute-force algorithm, but an iterative one that looks in each step at the current cluster state and computes the next cluster state. This is done without look-ahead, and without keeping history, so its time and space characteristics were very good and we soon started testing this new tool in production.

At this point, the team made a decision: do we keep developing the Haskell implementation of the algorithm, or do we rewrite the algorithm (which was reasonably simple) in Python?

The reasons for staying with Haskell were twofold. First, all the problems we attack are basically numerical algorithms, thus they

[2] `http://www.drbd.org/`, a networked RAID1 driver usable either in active-passive or active-active mode

[3] `http://twistedmatrix.com/trac/`, "an event-driven networking engine written in Python"

[4] The Protocol Buffers data encoding format, which is used extensively inside Google, was not open-sourced at the time we did this conversion; hence the choice of JSON

model very nicely in the pure domain; Haskell was here at its best, and the performance of the program was very good. Second, the entire Haskell code-base was trivial at this point (roughly 1500 lines of code, including comments), so the cost of an eventual rewrite into Python (in case ever needed, e.g. to standardise on a single language) was deemed low enough as not to be an impediment.

A third, non obvious reason for the initial acceptance of these tools was that they came at the right time, and filled a very big gap in our project. The value that they brought was high enough that it helped overcome the barrier of introducing a new language.

2.3.2 Expansion of the code-base

In the months following the initial acceptance, the project entered a phase of significant expansion; after a few weeks of use, the question changed from "should we use this to automate cluster balancing?" to "can we add a new rule/constraint to the algorithm?".

From the team's perspective, once the initial barrier of acceptance was overcome and the tools were stable, there was no reason to hold back their use. From the development point of view, once the initial I/O framework was in place and the core algorithm implemented, it was easy to iterate on the code base and extend it with new features.

Furthermore, after we had some experience with the cluster balancing tool, we realised that the algorithm we developed could be used for all our allocation/layout problems; whether placement of a new virtual machine, or most efficient layout, or computing the maximal cluster capacity. At this point, we were comfortable enough with the stability of the code-base to delegate such decisions to it, and continued to iterate on the capabilities of the tools.

2.3.3 Integration with Ganeti APIs

Initially, the Haskell tools were interacting with Ganeti via the command line interface, which worked but was suboptimal. As described in section 2.2, the APIs provided by Ganeti use standard protocols and data formats, so in time it was rather trivial to extend the Haskell tools to talk to Ganeti directly.

Fortunately, the `json`, `curl` and `network` libraries in Haskell are stable and have all the needed features for our use, so from this point of view we have observed no limitations in what regards library support.

The use of the APIs from Haskell led to interesting discoveries about the consistency of our Python RPCs, described in section 3.4.

2.4 Results and current status

By the summer of 2009, the tools were stable enough that we also released them[5] as Open Source under the name "ganeti-htools". The work on them continued and as of February 2010 we have the following capabilities:

- Local and remote gathering of data from Ganeti clusters
- Direct job execution for the local transport, or in the case of remote transport, creation of a shell script with the needed commands
- Sequencing of jobs customised such that we get the maximum parallelism when executing them in Ganeti

The software package consists of the following tools:

hbal – computes the needed moves to improve the cluster layout

hail – used as an IAllocator script for Ganeti, for both new virtual machine placement, virtual machine moves and physical machine evacuations

hspace – computes the available cluster capacity

All the tools work based on the same core algorithm:

1. The cluster state is analysed and we compute the current numerical score based on both hard constraints and optimisation scores
   (a) hard constraints represent extremely undesirable cases that are flagged as errors by Ganeti itself; they degrade the cluster score heavily
   (b) optimisation scores are obtained by computing the standard deviation of normalised metrics (e.g. percentage free memory on all physical machines is expressed as a value in the range $[0, 1]$ and the standard deviation of this vector is used as the 'free memory score' metric)
   (c) at the end all metrics are summed and they result in the final cluster weight
2. We then iterate over all the possible virtual machines and their moves (in balancing) or over all the possible ways to allocate a new virtual machine, and chose the best (according to the new score) state

Initially we had only two metrics (percentage memory free, percentage disk free); the current version has many more:

- percentage free memory, free disk and memory reserved for redundancy
- ratio of virtual-to-physical CPUs
- experimental metrics for load-based balancing (CPU load, memory load, disk bandwidth, network bandwidth)
- offline physical machines still hosting virtual machines
- virtual machine exclusion via tags (for example, preventing two virtual machines used as DNS servers to be hosted on the same physical machine)

The algorithm is known to be imperfect (e.g. since it does not look ahead, it can get into a situations from where it cannot execute any more moves), but in practice it works well enough that all layout decisions can be done with it, with manual intervention being very rare.

After the initial implementation and production deployment, development proceeded at a somewhat slower pace, but nonetheless we continued to improve the basic algorithm, implement new features, and keep to date with Ganeti changes.

3. Haskell/Python interaction

In the following sections we detail our experiences in combining Haskell and Python, and some changes to the (already mature) Python code base as a result of gaining experience with Haskell. It is important to note that we don't claim that either Haskell or Python absolutely enforces a certain programming paradigm; it's just that each language has certain characteristics that make it easier and more natural to program in a certain way.

3.1 `Either String` in Python

In languages which have native support for exceptions one can usually find many libraries that offer over-the-wire transport of exception; for Python, both Twisted and Pyro[6] offer transport of Python exceptions from the server to the client (with just a few restrictions).

[5] see the release announcement at `http://groups.google.com/group/ganeti/msg/8a9fef84ff138071`

[6] `http://pyro.sourceforge.net/`, "an advanced and powerful Distributed Object Technology system"

However, there are few, if any, both lightweight and language independent RPC libraries that offer this. When moving from Twisted to our HTTP-based RPC in Ganeti, we saw this as a regression in functionality, and we planned to solve it at a later time. In the meantime, we started to modify some RPC calls to return a tuple `(Boolean, Payload)` with the first member representing success or failure and the second one being either a string (in case of failure) or the actual payload. Since this seemed to be a temporary 'hack' until we got a real exception propagation framework in place, we only implemented it for a few RPC calls, in an ad-hoc mode.

After 'discovering' the `Either String` data type in Haskell, we realised that this is exactly what were using in Python. Far from being a hack, it's a simple and elegant way to transport classless exceptions. We proceeded to rework our RPC framework to have this as a basic functionality instead of being implemented in the individual RPC calls, and as a result we gained much better error reporting across the entire inter-node communication.

It is unfortunate, though, that one has to code algebraic data types by hand in imperative languages; for many types of problems they are the most natural way to express values.

3.2 Persistent data types

Python has very weak support of persistent data types; it is neither possible to mark a complex data structure read-only (in a generic way) nor to make cheap copies. The only native facility for data copies is in the `copy` module, but the speed of a so-called deep-copy is slower than a manually-coded attribute-by-attribute copy by a big margin (our tests show factors between three and five times).

These two reasons tend to drive the architecture to careful in-place updates, even when this a suboptimal solution. After our experience with Haskell and understanding how much safer copy-and-modify is, compared to in-place modification-and-undo, we reused our serialisation framework for a cheap data-copy functionality. This was facilitated by the fact that said framework is based on a two-stage process, first converting from custom objects to 'standard' Python types which are then serialised via JSON. By doing just the initial custom-object-to-standard-object conversion and then its reverse, we managed to get a simple, albeit slow, data copying method.

This increased the safety of our code in a number of places, especially as these data types are used in a multi-threaded environment. We envision that careful use of this method can lead to a semi-pure style of programming in Python. Of course, the best solution would be to have the ability to 'freeze' arbitrary objects in the Python standard library.

3.3 Functional programming features in Python

The Python language has adopted a few functional programming features (sometimes directly from Haskell). How do these compare with their native counterparts?

Probably the first such feature that Python programmers get accustomed to is a *lambda* expression. It has, however, a big weakness: Python differentiates between statements and expression; since lambdas can only contain expressions, they are not as powerful as regular functions. As such, their usage is very limited in Python, and their future in the language is under question[7].

Fortunately, using functions (defined either normally or as lambda expressions) as normal values is indeed possible without any restrictions, and this is useful enough that (for example) we're using it extensively in our Python code-base. Also the list generators are similar in both languages, and can be used to good effect in Python too.

[7] The original Python 3000 standard proposed to drop them, but this decision was reversed during the development cycle

Recently, Python has gained partial function application, however this was implemented as a library, and not at syntax level. Thus, actually using this feature is less direct; compare for example a very trivial example in Haskell:

```
let fn = map length
```

with the Python version:

```
fn = partial(map, len)
```

The latter is more cumbersome to use, as it breaks the normal code flow (it is not instantly clear that `len` will be an argument of `map`, one has to mentally parse the `partial` function call first). Due to this, and to the fact that it is a recent addition to the Python libraries, we have not yet started using `partial` in our Python code.

Another pair of functions present in both languages and which we use is `any` and `all`. While the original Python implementation (as a library) was similar to the Haskell one, the recent conversion to built-in functions dropped the predicate argument. This had two effects: for lists of booleans, their use is simpler, but for lists where we still need to apply a predicate, the syntax became more complicated:

```
result = any(pred(i) for i in lst)
```

This change is not a big impediment to their use, but it results in more verbose code.

To summarise, the functional programming tools present in Python are of mixed quality, making their consistent use hard. It seems that the process of adopting them from other languages was not perfect: in some cases they are harder to use, and often they look like additions, rather than integral elements of the language.

3.4 Ganeti API consistency

Ganeti has roughly four sets of APIs that interest us:

1. command line interface; while mostly used by people, this was designed to be scriptable such that many tools use this simple method
2. local UNIX socket, JSON encoded messages; this is the simplest (and fastest) method since it relies on local Unix socket permissions and security and thus it doesn't need to do any data encryption
3. HTTP-based, REST-style API; used for remote querying and administration (called *RAPI*)
4. the IAllocator API, a plugin-based framework for allocation and layout policies; this is an internal API, used between Ganeti and plugin scripts

The first three API sets are used for querying and changing the cluster state, while the latter is used for feeding information back into Ganeti as described in section 2.1.

Except for the first API, which is plain text, all use JSON encoded messages which translate into native data types in most languages. This should make it straightforward to introduce a generic tool that talks either locally or remotely to Ganeti.

However, when trying to integrate our Haskell program with Ganeti, we quickly found out that:

- not all APIs exported the same data; the data available via each API was only a sub-set of the data available in Ganeti
- even for the same data, the actual naming of various properties differed across the APIs

While any program (independent of the language used) would have discovered these inconsistencies, static vs. dynamic typing makes a difference here. In Python, it's very easy to adjust data

structures on the fly, since we don't have a strong type system. Changing a certain variable from Boolean to Integer will work most of the time without any changes. As such, it's easier to simply adapt and have code branches that deal with different versions of a data format than adjust an entire ecosystem to a format change.

In contrast, our experience with Haskell shows it's best if differences are kept entirely at the representation layer (e.g. it is acceptable to have a different name for a value in the JSON message) but not in actual data format, since this means introducing algebraic data types which will complicate (via excessive use of pattern matching) many of the functions manipulating that particular piece of data.

The effort needed to unify the APIs was small, since the differences were somewhat minor; this allowed a streamlined Haskell implementation, with multiple backends (each talking to a specific Ganeti API) which return the same data structures; the actual algorithm is then oblivious to the fact that we used an HTTP query or a Unix socket connection to talk to Ganeti. On the Python side, this resulted in a saner API overall, which should benefit all its consumers.

3.5 A test version of the "master daemon" in Haskell

As described before, our Haskell project is a set of small tools external to the main Python code-base. The Ganeti software architecture is moderately complex: on all physical machines, a so-called "node daemon" runs; one of these machines is designated as the current "master node" and it runs two more daemons: the "master daemon" which is the one actually responsible for the job execution and coordinating the node daemons, and "remote API daemon" which offers the RAPI endpoint described in the previous section. Commands can be submitted either remotely via RAPI, or locally on the master node via a set of Python scripts that talk directly to the master daemon.

In this architecture, the node daemons are doing purely I/O related work, talking to LVM, DRBD, and the hypervisor in use. The master daemon manages the cluster configuration (and its replication) and coordinates the interactions between the node daemons. This means a significant part of the master daemon is dedicated to abstract data handling:

1. Accepting incoming jobs from clients
2. Managing the job queue, replicating the jobs to other nodes and archiving them
3. Executing the jobs, including managing the locking and synchronisation between the different worker threads

As an exercise, we tried to see if it is feasible to implement a small subset of the master daemon in Haskell, how a Haskell program that deals heavily with I/O compares to the original Python code, and whether the advantages seen for the original tools are replicated to other parts of the code base.

This effort had mixed success. At a basic level, we were able to implement the basic functionality (accept jobs, local queue management) and the most basic job type (the null job type), but the continuous use of the I/O domain changed the style of programming significantly: it looks rather more like our original Python version than expected (this might also be due to our limited experience with Haskell). We developed this Haskell version of the master daemon until we were able to actually run the Python command line scripts and see a fake cluster, execute job management commands, etc., at which point we declared the attempt complete.

During the exercise, the use of the GHC profiler has revealed some deficiencies in our algorithm used for job submission and dispatching to the worker threads; since this was copied verbatim from the Python version, we were able to devise improvements that were then back-ported to the original master daemon implementation[8]. It would have been possible to detect such inefficiencies based on the Python version, but the profiler support for multi-threaded Python code is of very low quality compared with the GHC profiler; in this way, we used the GHC tool-chain indirectly for improving the Python code.

While benchmarking the two implementations, a surprising result was that, due to the deficiencies of the standard `String` data type in Haskell, serialisation/de-serialisation of JSON messages is actually slower in Haskell compared to Python (which uses a C-based module for speedups).

In retrospective, this exercise has proven that it would be feasible to write more parts of our project in Haskell. Even though the heavy I/O emphasis in some areas does not match entirely Haskell's strengths, there are other advantages (like the combination between read-only data-structures and very "cheap" sparks) that would make a Haskell implementation attractive.

4. Roadblocks

While our experiments with Haskell were successful and resulted into production level code, we have had a few small roadblocks along the way.

4.1 Debugging facilities

Like many other Haskell programmers, we had to debug the famous error message "`Exception: Prelude.head: empty list`", followed by an abrupt exit from our program.

In standard Python code, this message would have been accompanied by a stack trace, including the code fragment that generated such an exception. In Haskell, however, the available options are very primitive: either use `Debug.Trace` (a low-level facility) or implement changes to the pure code-flow (via the use of a Writer monad). Combined with the effects of laziness, this makes debugging a complex application much more difficult for the beginner Haskell programmer than in a scripting language.

Another problem, not particular to Haskell but to compiled languages in general, is the inability to quickly debug problems on the deployment systems. Instead, the problem must be replicated on the development machines. By itself this wouldn't be an issue, but it exacerbates the debugging difficulties.

In due time we have learned ways to compensate for both these problems, but we still find debugging tools a bit weak in Haskell (e.g. compared to the excellent profiler); the author is looking forward to developments in this area—e.g. [Allwood et al. 2009].

4.2 Status of library packaging in Linux distributions

While Hackage provides a plethora of libraries, not all of them are available in Linux distributions, or at least not in the current stable versions. We're mainly interested in Debian, since it's our test/reference platform.

As an end user, due to the static linking, it's very easy to deploy Haskell software: the needed libraries must be available only on a development machine, while the deployment targets do not need them. But packaging software for Debian is different, as it requires that all prerequisites must be available in Debian itself, for hermetic/reproducible builds across all supported architectures.

We use just three libraries: `json`, `curl` and `network`; at the beginning of year 2009, neither the `json` nor `curl` were available at all in Debian, and this delayed our packaging efforts. Today, all are available in the unstable track, and this allowed us to package `ganeti-htools` for it, but it is still hard to back-port our software for the current stable version.

[8] an example is commit `009e73d` in Ganeti, *Optimise multi-job submit*

The introduction of the Haskell platform, and the advances in packaging of the *debian-haskell* group however makes this an obsolete issue; both developers and end-users will be able to consider the state of Haskell integration on the same level as Python's in future Debian versions.

4.3 Rate of change and maintenance effort

One surprising finding during development is that some basic functionality, in our case related to error handling (the use of the `Control.Exception` module), changed so much between GHC 6.8 and 6.10 that is hard to write code that works with both versions, unless one starts to use conditional compilation (via CPP defines, but this has its own problems), or switch to another library for error handling (e.g. the extensible exceptions package).

Fortunately we were mostly interested in I/O errors, so we were able to revert to the simpler error handling in prelude[9]. But such changes were surprising to us, and we did pause to consider how much effort will be needed in the future to keep backwards compatibility with current deployed systems, while still allowing use on unstable/development platforms.

4.4 `String` data type speed issues

As described in section 3.5, even in our limited experience we have observed the slow speed of the standard `String` data type. The author's initial reaction was to simply look for other libraries in order to solve this problem; but, even though alternatives exist, the `String` type is still having a privileged position as the default text type, and thus many libraries use it by default; as such, the effort is put on prospective developers to decide "can I use library *X* or do I need to find an *X-bytestring* implementation to get reasonable speed?". This situation is unexpected in a language and an implementation (GHC) that seems in many other ways quite mature, and the author believes that it adds a non-trivial effort on both sides of the community: on library authors, who need to provide bytestring-enabled versions, and on the users of libraries, who need to make sure their mix of libraries does work as expected and gives reasonable performance.

4.5 High barrier to entry

Lastly, we believe that the most significant problem is the high barrier to entry.

Even after the completion of this project, the author feels that his knowledge of Haskell is very much incomplete, and that he is far from being familiar with advanced topics (e.g. applicative programming, generic programming, etc.). Whether this is needed or not for small projects is debatable—for example, our current code works with only standard Haskell 98 (no language extensions in use)—but it might be possible that careful use of advanced programming techniques will reduce and simplify the code.

The second remark on this topic refers to the difficulty of co-opting other people to contribute; except for a few trivial patches, in our project the Haskell component remains a one person effort, compared to the Python code which has had around three to five active contributors (depending on project phase).

5. Summary

After using Haskell for slightly more than a year, our conclusion is that even though the adoption barrier is quite high, Haskell is a good asset for solving certain types of problems. The combination with Python has shown to be a success, and we have managed to write a set of tools that are used daily for solving real-world problems.

For the foreseeable future, our project will remain a dual-language one; rewriting the Haskell part in Python is doable, but we would lose the advantages described in the paper for this particular kind of problem (numerical algorithms). As to the opposite option, rewriting the Python part in Haskell, there are a few reasons why this is not feasible. First, the Python code-base is significant (around 30K lines of code), and rewriting a project of this size would be a huge effort, which is hard to justify. Second, it is unknown whether our team would be able to successfully re-implement Ganeti in Haskell, given our limited experience with the language.

Were we to start the project from scratch, today, it would be a different proposition. Both Python and Haskell have their advantages, and choosing the right language would be a hard decision. Nevertheless, after using Haskell in real life to solve actual production problems, we believe that—both at language level and at implementation level (GHC)—it is a viable choice for projects of similar complexity in the domain of system administration.

[9] commit `1cf9747` in htools, *Change ExtLoader to only handle I/O errors*

Acknowledgments

Many thanks to the Ganeti team in Google for their support during my initial experiments with Haskell, especially to Guido Trotter and Michael Hanselmann.

Also, the subject of this paper would have not existed without Haskell itself, and the many resources created by the community that enabled me to learn, write and deploy Haskell.

References

T. O. Allwood, S. Peyton Jones, and S. Eisenbach. Finding the needle: stack traces for ghc. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 129–140, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-508-6.

V. Balat, J. Vouillon, and B. Yakobowski. Experience report: ocsigen, a web programming framework. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 311–316, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7.

P. Cuoq, J. Signoles, P. Baudin, R. Bonichon, G. Canet, L. Correnson, B. Monate, V. Prevosto, and A. Puccetti. Experience report: Ocaml for an industrial-strength static analysis framework. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 281–286, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7.

R. R. Newton and T. Ko. Experience report: embedded, parallel computer-vision with a functional dsl. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 59–64, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7.

C. J. Sampson. Experience report: Haskell in the 'real world': writing a commercial application in a lazy functional lanuage. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 185–190, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7.