Why *Swift* for TensorFlow?

The core graph program extraction algorithm, automatic differentiation, and Python language interoperability features of Swift for TensorFlow can be implemented for other programming languages, and we are occasionally asked why we didn't use some other one for this project.

The engineers on the project were previously familiar with Swift (and several other languages), but the choice was guided by the goals of our project, which imposed specific technical requirements (explained below). This choice was also discussed extensively, debated with coworkers and other interested engineers, and we concluded that Swift was the best direction. In this document we're sharing our deliberation process with the community to help explain our decisions.

That said, while our choice of language was guided by our specific project goals, we would love to see wider application of these techniques and ideas in the context of other programming languages! If you are interested in pursuing a similar project, please reach out to us and we will happily share our expertise.

How we got here

As discussed in the design overview document our project goal is to improve usability of TensorFlow. We quickly realized that our core static analysis-based Graph Program Extraction algorithm would not work well for Python given its highly dynamic nature. This led us down the path of having to pick another language to work with, and we wanted to approach this methodically. As such, we defined goals for the project, explored which properties of a programming language are important to achieve those goals, and then evaluated a lot of languages against these properties. You already know the outcome--we eventually settled on Swift.

Below we explain our project goals, discuss the programming language properties that contribute to these goals, provide a short evaluation of specific languages against those goals, and discuss the pros and cons of Swift specifically.

Project goals

TensorFlow is a world-class machine learning framework used by a wide range of different people for lots of different things. Our project goal is to provide a new interface to TensorFlow that builds on TensorFlow's power and capabilities, while taking its usability to a whole new level. Our aim is to make machine learning researchers (both theoretical and applied), production engineers deploying at scale, and anyone else using TensorFlow more productive and joyful without sacrificing anything that already makes TensorFlow great.

We defined goals around the properties that are important to maintain and improve in our system:

Expressiveness: We want a define-by-run model that feels like you're directly programming against a numeric API and the host language (like NumPy), without an explicit graph abstraction in the way. We should not put constraints on native control flow, use of native data structures like dictionaries, or other things that might feel natural.

High Performance: We want to get the most out of our hardware and accelerators, including CPUs, GPUs, Cloud TPUs, and future accelerators being developed across the industry. Performance is important for production deployments to save megawatts, but is just as important for researchers who want fast turnaround time on experiments.

Hardware Abstraction: It should be possible to build a model without embedding hardware-specific information in it, and get "good" performance out of the hardware. We should provide the (opt-in) ability to further tune the model for specific accelerators to achieve full peak performance, and make sure that is a low-friction path that doesn't require a rewrite of a model.

Large, Dynamic, and Self-Modifying Models: We don't know what ML models will look like in 5 years, but it seems clear they will be bigger, sparser, and less rigidly structured. We should solve today's problems and embrace the next generation that researchers will come up with as compute continues to get cheaper. We should fully support dynamic models like attentional models, large models like mixture of experts, as well as reinforcement learning models that require frequent interactions with real or simulated environments (e.g. an Atari game).

Performance Predictability: A key aspect of making a large and flexible system usable is to make it predictable: while it should be possible to express anything, it should be easy to predict what will run efficiently, and the tools should provide feedback about performance cliffs. Simple and predictable tooling is preferable to layers of magic that try to paper over the most common problems.

Fast Iteration Time: We need to enable productive researcher workflows, where the tools dissolve out of the way, allowing the user to focus on the data and math.

Debuggability and Introspection: An important part of R&D is figuring out what is happening in a model, diagnosing failures, and figuring out what to change. A system that diagnoses more errors earlier (e.g. shape mismatches at compile time) is more usable than one that only catches them at run time.

High-End User Experience: We should meet the users where they are, regardless of whether they like UI, console, or batch processing experiences. We should support terminal users (e.g. with an interpreter/REPL), Jupyter notebooks, users who prefer UNIX `#!` scripts, and other common patterns.

Flexible Deployment: We need to be able to deploy to inference-only mobile targets through TFLite in addition to supporting inference on high-end accelerators like GPUs and Cloud TPUs. We need to support production teams that want minimal dependencies, e.g. by being able to produce a single .o file for the CPU.

Fast Deployment: The ML space is moving really fast: we should remove obstacles in place between research and production deployment wherever possible. Requiring a significant rewrite of "research code" to put it "into production" would be harmful both because it slows this down, but also because it makes it very difficult to iteratively improve production models. Rewrites can also introduce very subtle and pernicious errors.

Best-of-class Automatic Differentiation (AD): AD is a key feature of TensorFlow, but we can improve its user experience and pull in more advanced technologies from the AD communities. We should support higher-order AD, as well as the best-in-class implementations like specialized adjoints, easy calculation of Jacobians, per-example gradients, checkpointing, etc.

Embrace TensorFlow Graph Ecosystem: GraphDef (and SavedModel) are the key for interchange, transport, deployment, integration with visualization tools etc.

As you can see, this isn't a short list of goals, and this isn't easy to achieve. That said, we do believe we can do this in a single coherent system, and TensorFlow's mature technology handles a lot of this for us.

It is worth noting that as of our launch in April 2018, Swift for TensorFlow is not yet solving some of these goals (e.g. improved deployment and self-modifying models) but we have ideas that we expect will develop into great steps forward and cover each of these over time.

Properties of programming languages

Programming languages are an aggregation of a bunch of largely orthogonal design decisions (and their consequences) manifested into a single inseparable artifact. Each language offers different tradeoffs along each axis, so choosing one is akin to solving a highly multi-dimensional sparse optimization problem. To increase the challenge, many programming languages can be enhanced, so we need to factor in the likelihood of some deficiency being overcomeable by enhancements to the language and compiler (and the odds that the community would accept such a change).

This section discusses some of these axes, and relates them back to our list of project goals above. Very few of these properties are "absolutely required", but a weakness in one of these areas can cause follow-on effects. While we are deferring the specific language tradeoff discussion to the next section, we identify various classes of languages that are problematic for a given goal in some cases.

Strengths of Python

First, let's start with the hopefully non-controversial strengths of Python which we must not jeopardize:

Community: Having a large community of users is important: community drives books, tutorials, and all the little things that no one thinks about until they are missing (we need both emacs AND vim support... :-). This requirement generally excludes most research languages, and excludes the possibility of building a new language for TensorFlow.

Open-source and cross-platform: TensorFlow is open source and has users on every imaginable platform. We want to be able to scale to support all of them.

Aesthetics and design: Syntax is a language's "user interface", and has a deep effect on usability and how users work with the system. Good design is really hard work, but it matters - it is not just a matter of "bikeshedding".

"Mainstream" syntax: TensorFlow is a mainstream system, most of its users are using Python, and we aim to appeal to that broad user base. This means that excessively "non-mainstream" languages will hurt adoption of our techniques. That said, we would be very interested for other communities to explore these techniques.

Shallow learning curve: We want new users to spend their time learning TensorFlow, not struggling with a language that has a high learning curve. A steep learning curve also reduces the odds of existing TensorFlow users switching to our new system.

High productivity: People love Python because it has low boilerplate and low "ceremony". Many people get frustrated when they feel like they're wasting time placating the compiler or churning out boilerplate.

Debugger: One of our goals for TensorFlow is for people to be able to debug their models more effectively, so we need a debugging experience. This generally excludes research languages that don't have debuggers.

Interactivity: Batch compilation has an important place in production, but REPLs, #! scripts, notebook environments, etc are widely used across research teams.

Predictable semantics: The type system (whether dynamic or static) should have sensible behavior and compose without surprising behavior.

Memory safety: We don't want TensorFlow users to spend their time chasing dangling pointer bugs and undefined behavior in their models. This requirement generally excludes languages like C, C++ and Fortran. On the other hand, it is worth noting that interoperability with existing C code is really useful for large scale systems.

Python APIs: TensorFlow users benefit from and rely on a huge collection of Python APIs outside the TensorFlow product, e.g. for visualization and data science stuff. If we ignore this reality, adoption of our new system (no matter how great it is) will be very slow.

Python challenges

Python is great at the points above, but it has some challenges as well. Here are some things that would make TensorFlow better if they were improved:

Performance: Performance is a perennial problem with Python and causes a significant amount of additional work for TensorFlow core: it is most natural to write things in the Python layer, but many things can't be done there for performance reasons. This leads users to having to write things as TensorFlow ops in C++, just to work around Python performance. This turns low performance into a significant usability issue for Tensorflow.

Concurrency: Presence of the GIL increases complexity throughout the stack in large scale settings: it prevents model authors from achieving reasonable performance for some things (without themselves writing C++/CUDA), and prevents natural expression of concurrent algorithms in a model.

Deployment: The Python interpreter and its package ecosystem are non-starters on mobile. It is also considered to be a significant liability for product teams that want hermetic low-dependency builds. Production users often train in Python, but write their inference logic against the TensorFlow C++ API for production - harming our goal to shrink the gap between R&D and production deployment.

Custom Ops: Building a custom op currently requires writing C++ and Eigen/CUDA - which is a huge complexity cliff to jump off of, a barrier in some cases, reduces the free interchange of models (since they now depend on something outside their source), and can lock ML models to certain hardware vendors. It would be better if there was a path for high-performance kernels to be written in the same language as the model.

These well-known problems with Python make the TensorFlow programming model more complicated than it would ideally be. They require advanced TensorFlow users to learn and deal with the C++ and CUDA layers of the stack. They slow down researchers, who are often the ones trying new things and pushing the limits of the current system.

Properties needed by Graph Program Extraction

The fundamental algorithms we use to automatically identify and extract graphs out a program are based on static analysis of the code. This means that we need to be able to look at code "statically" - i.e., without running it - and be able to *reliably* identify Tensor operations and the data flow and control flow dependencies between them. In order to achieve the performance of graphs, we need to be able to connect together large chunks of tensor code - preferably with the same granularity as if you were to manually use an API like the TensorFlow session API. We also need to allow users to be able to use high-level APIs like layers and estimators (and build their own abstractions!) without breaking our ability to do this analysis.

In proof systems, it is well known that it is difficult to make a static analysis that is both *sound* and *complete*, meaning that you get a choice of 1) an unsound model that sometimes produces incorrect results but handles all programs, 2) a sound model that is always correct but handles a limited set of programs. Since we are using static analysis for code generation, we require the analysis to be correct! Let's explore the conservatively correct but incomplete model.

We can be provably correct if we limit our analysis to a bounded computational model that we know we can reliably analyze. In our graph transformation, this approach works out well, because we have a conservatively correct fallback to handle any situations that are outside our analysis capabilities: we can copy data from the graph back to the host CPU, and dynamically execute arbitrary logic to determine what to do next, then send data back to the graph computation.

There are two things we need to make this approach work in practice: First, we need the subset of cases we handle to be large enough to include interesting things - including high level user abstractions. Second, we need the resultant model to be simple enough that normal humans can understand it: Lots of special cases and scenario-dependent behavior undermine the usability of the system.

There are subtle, but critical aspects to making this work in a usable way, and many well-known programming languages (including Python and other OOP-centric languages) force unacceptable tradeoffs that prevent this from working well in practice. This issue affects any object oriented language whose dispatch model is semantically based on dynamic dispatch and mutable state. To explain the issue we use an intentionally simplified example in pseudo-Swift syntax (Note: this is not idiomatic code!):

```
class Layer { ... }
class Convolution2DLayer : Layer { ... }
class BatchNormalizationLayer : Layer { ... }

class ResNetBlock {
  // Layers this block is made out of.
  var conv1, batchNorm1, conv2, batchNorm2: Layer

  init(inFilterCount: Int32, outFilterCount: Int32,
       strides: (Int32, Int32)) {
    conv1 = Convolution2DLayer(filterShape: [3, 3, inFilterCount, outFilterCount],
                               strides: (1, strides.0, strides.1, 1))
    batchNorm1 = BatchNormalizationLayer(axis: 0)
    conv2 = Convolution2DLayer(filterShape: [3, 3, outFilterCount, outFilterCount],
                               strides: (1, 1, 1, 1))
    batchNorm2 = BatchNormalizationLayer(axis: 0)
  }
}
```

This example uses a hypothetical layer library to set up a block from a residual network. This is intended to be reasonable code that uses the kind of abstractions a Java or Python programmer (for example) would use.

However, the apparent simplicity of the code above underlies a number of complexities: the properties like `conv1` could be reassigned after the block is set up. The method calls to `forward()` and `backward()` on layers (not shown) are dynamically dispatched, which means that the target of the call depends on dynamic properties of the code. The "pointers" to the layers could even be aliased to other references, making data flow analysis of reads and writes from each of the objects difficult to reason about. These possible behaviors make it difficult for a compiler to statically prove the result of the program - a prerequisite for reliably extracting a graph.

Coming back to our choices above, we have a few options we could pick:

1. **A predictable, explainable and limited model:** There is a (very small) subset of Java that we can analyze statically in a fully reliable way: if all code is in static methods, or in methods of final base classes, then we know that all calls are statically dispatched. If all variables are guaranteed to be assigned a single time, then we can do aggressive pointer tracking. The problem with this option is that we lose the ability to have high level abstractions, at least without having copies back and forth all over the place.

2. **An unpredictable, difficult to explain, but more general model:** It is well known that while many OOP languages are dynamically dispatched in theory and techniques like class hierarchy analysis and interprocedural alias analysis can discover many of these statically. We could use heuristic techniques like this to expand the case we can handle. Unfortunately, these techniques generally require "whole program analysis", and subtle changes in one part of your program can cause "spooky action at a distance" because the analysis gets confused. This means that a small change to one part of the program can cause sends and receives to be added to other parts of the model, leading to a difficult or impossible to explain model for users.

Our project goal is to provide a highly usable experience, which requires predictability and explainability from the programming model. This is particularly important because sends and receives can have a significant impact on performance: the data involved could be gigabytes in size! Because of this, we reject the heuristic-based approach and stick with the predictable, explainable, but limited model.

All is not lost though: not all programming languages are limited in the same way. For example, if you compare C# and Java, the introduction of structs in C# allows it to reliably handle a slightly broader range of cases than Java does. A far extreme is C++, which has a Turing complete template metaprogramming system that is 100% static. Swift is somewhere in between, offering a lot of expressive power (in particular, due to its approach to protocol oriented programming) which we explain in the Graph Program Extraction deep dive.

Additional properties needed by Graph Program Extraction

In addition to reliable static analysis, we benefit from a few other properties of the language:

Restricted pointer aliasing: Graph Program Extraction only works if it can reliably build dataflow def-use chains between operations. Languages that allow unrestricted aliasing of pointers and references like C/C++ would force our static analysis to be overly conservative, making copies to the host in their presence.

Knowable tensor ops: Since we are extracting the tensor ops out of the host code into a graph, we need to know what they are - either through designated syntax, a static type system, or some other hook.

Suitable compiler IR: Graph Program Extraction is a non-trivial compiler transformation which requires a suitable Intermediate Representation (IR) to work on. This IR must be high-level enough to support the desugaring transformations we use, and be able to compile to a TensorFlow graph without losing essential information. The techniques required by Graph Program Extraction could theoretically be performed on a good AST, but are much easier to implement on a Control Flow Graph in SSA form.

Static vs dynamic type systems

Finally, there is the topic of static vs dynamic typing. Static typing offers a number of advantages particularly relevant for our use case. Static types:

- can catch bugs at compile time instead of runtime. People get very frustrated when a silly type error brings down a long training run hours into it.
- directly improve tooling experience like code completion/intellisense, jump to definition, etc.
- make it easy to know what operations are Tensor operations.

On the other hand, statically typed languages can require a lot of ceremony and boilerplate, which is infuriating and directly harms productivity. There are promising middle grounds though, such as languages that are statically typed but that use type inference to eliminate explicit typing in the common case.

Which languages fit our project requirements?

This is a long list of goals, and there are a lot of languages to consider. As such, here we choose to address clusters of related languages, rather than building and evaluating the full cross product of these one-by-one. We apologize in advance if we have mischaracterized any language or community below, let us know and we'll happily correct any mistakes!

A new language: Creating a language is a ridiculous amount of work. Not only do you need a parser, but you need a debugger, all the tooling, books and educational material, and all the other things required to build and support a big community. Furthermore, even if we were willing to make the investment required to do this, it would take many years to do it right, and machine learning is moving too fast.

Python: Given that the majority of the TensorFlow community is using Python already, using it for this work would be ideal. However, the dynamic nature of Python isn't suitable for the reliable static analysis and Graph Program Extraction approaches we depend on. It is possible that some subset of Python could be compiled (as is done by Tangent), but it isn't clear how to support high level abstractions like Python classes with that approach.

Ruby / Javascript / R / Typescript / etc: These languages share the same problems for static analysis as Python. We are often asked specifically about TypeScript: while it introduces a really nice type system, it doesn't improve on the fundamental dynamic dispatch and object aliasing challenges posed by Python.

Java / C# / Scala (and other OOP languages with pervasive dynamic dispatch): These languages share most of the static analysis problems as Python: their primary abstraction features (classes and interfaces) are built on highly dynamic constructs, which means that static analysis of Tensor operations depends on "best effort" techniques like alias analysis and class hierarchy analysis. Further, because they are pervasively reference-based, it is difficult to reliably disambiguate pointer aliases.

As with Python, it is possible that our approaches could work for this class of languages, but such a system would either force model developers to use very low-abstraction APIs (e.g. all code must be in a final class) or the system would rely on heuristic-based static analysis techniques that work in some cases but not others.

Go: Go is a great language with a growing community, and many Go programmers were former Python programmers. Unfortunately, Go is not a great fit for this approach: it allows unstructured aliasing and abstractions are often implemented in terms of dynamic interface dispatches, calls to `reflect()`, and dynamic casting from `interface{}` to concrete types. Each of these dynamic features defeats reliable large scale abstract interpretation, so users would be forced to use only the static functions and struct features of Go - and it would be impractical to build high-level layer and estimator abstractions out of these (unless they were built into the language).

Another significant issue is that the nature of Go would require adding significant language enhancements to the Go language, like a built-in Tensor type, some way to be able to express abstractions that are generic over the Tensor element type, automatic differentiation support, and an implementation of the core extraction algorithms. The Go community emphatically prefers to keep the language small, and using it for this project would require going against those core principles.

Rust: We believe that Rust supports all the ingredients necessary to implement the techniques in this paper: it has a strong static side, and its traits system supports zero-cost abstractions which can be provably eliminated by the compiler. It has a great pointer aliasing model, a suitable mid-level IR, a vibrant and engaging community, and a great open language evolution process.

A concern with using Rust is that a strong goal of this project is to appeal to the entire TensorFlow community, which is currently pervasively Python based. We love Rust, but it has a steep learning curve that may exclude data scientists and other non-expert programmers who frequently use TensorFlow. The ownership model is really great, but mostly irrelevant to the problems faced by today's machine learning code implemented in Python.

C++: As with Rust, we believe that C++ supports all the ingredients necessary to implement the algorithms required for our work, including a mature compiler frontend, and an active standards committee. However, C++ is rife with undefined behavior, and much of its static composition system relies on C macros and template metaprogramming, which is not particularly usable. Furthermore, because C++ doesn't generally appeal to Python programmers, we were concerned that choosing it would prevent a significant community from adopting our tools.

Julia: Julia is another great language with an open and active community. They are currently investing in machine learning techniques, and even have good interoperability with Python APIs. The Julia community shares many common values as with our project, which they published in a very like-minded blog post after our project was well underway. We are not experts in Julia, but since its compilation approach is based on type specialization, it may have enough of a representation and infrastructure to host the Graph Program Extraction techniques we rely on.

Final decision

In the end, we narrowed the list based on technical merits down to Swift, Rust, C++, and potentially Julia. We next excluded C++ and Rust due to usability concerns, and picked Swift over Julia because Swift has a much larger community, is syntactically closer to Python, and because we were more familiar with its internal implementation details - which allowed us to implement a prototype much faster.

Evaluating Swift against our language properties

It might be interesting to see how we evaluated Swift against the point-by-point language properties we were aiming for at the top of the document. Here is a brief summary of the these points - starting with the topics Python excels at. We also openly address Swift-specific limitations and challenges at the end.

Swift compared to Python's strengths

Community: Swift is a fast growing language that is among the top 10 programming languages in recent surveys - it is believed to have well over a million programmers using it. It has a large and vibrant open source community and a well-developed ecosystem of tools. Swift's stronghold is mobile development, but several communities are pushing on Swift for server and cloud applications.

Open source and cross-platform: Yes.

Aesthetics and design: This is a subjective topic and hard to measure, but this was a major focus of Swift 3. Swift benefited from a year of the open-source community working to improve the syntax to ensure consistency and elegance. Source-breaking changes to the language are very difficult to make now, but Swift definitely benefited from not locking down too early.

"Mainstream" Syntax: Swift is designed to fit in with the "extended C family" of programming languages, and intentionally tries to feel "familiar".

Shallow learning curve: A key design goal of Swift is to progressively disclose complexity in the language, which makes it an extremely teachable language. This is one of the things that has enabled teaching Swift code to kids as their first programming language (targeting middle-school, 7th and 8th grade) in the Swift Playgrounds iPad app.

High productivity: Swift aims to maximize clarity of code, and thus it fights to reduce boilerplate. The top-end goal of Swift is to optimize the time it takes to write and maintain a working app, which includes debugging time and other things that go beyond just pounding out the code.

Debugger: Swift has a full debugger with command line and IDE support.

Interactivity: Swift supports batch compilation, a powerful REPL (with full debugger integration), and #! scripts. Notebook environments are freely available for Mac (Xcode Playgrounds), iOS (Swift Playgrounds), and a web version is also supported.

Predictable semantics: Yes.

Memory safety: Swift aims to be as "safe by default" language, both in terms of memory safety but also helping to catch logic bugs early. It also provides explicit "unsafe" APIs to interact with C code and for direct hardware access.

Python APIs: When we started the project, the answer was "no", which is why we prioritized building a new approach for expressive Swift/Python interoperability. The answer is now "yes".

Swift compared to Python's challenges

Next, there are the aspects that are problematic today for Python:

Performance: Swift has great low-level performance and memory use, and because it is so widely used on mobile, there is a large community of people who are continuing to improve it. When explicit memory ownership support is added, Swift will be a credible replacement for many uses of C++.

Concurrency: Swift doesn't include language support for concurrency yet, but works fine with native APIs like pthreads, and it comes with a nice work queuing API named Dispatch (aka "GCD"). A first-class concurrency model is a likely future feature of Swift, which could be based on async/await and Actors.

Deployment: Swift can compile down to simple native machine code like C, and doesn't depend on a garbage collector or other heavy runtime. It is entirely reasonable to compile a ML model into a .o/.h file with Swift.

Custom Ops: Swift builds on top of LLVM and has direct access to all LLVM intrinsics - and LLVM can generate GPU kernels for both Nvidia/PTX and AMD cards. In principle, someone could build an embedded "CUDA/OpenCL for Swift" DSL. We are interested in exploring this over the long term, as it would lead to models being more self contained (thus easier to share) and would allow advanced users to use a single language.

Properties needed by Graph Program Extraction

Next, here are the aspects we care about in order to build the Graph Program Extraction model we describe here:

Reliable static analysis through high-level abstractions: Swift has highly dynamic features (classes, existentials, and escaping closures) but it also has a well developed static side as well, which revolves around structs and enums. Structs and enums can be generic, have methods, and conform to protocols (which provide "interface-like" features, mixins, and other capabilities referred to as Protocol Oriented Programming). Swift depends on these static features being eliminable, because its basic types (like `Int` and `Bool`) are actually implemented in the standard library - not built into the compiler. As a result of these properties, Swift provides a simple and reliable conceptual model for developers: abstract interpretation is guaranteed to succeed and give you great performance, so long as you don't use classes, existentials or other dynamic features. This is explained in depth in the Graph Program Extraction document.

Knowable tensor ops: Swift has a static type system, which gives us a simple heuristic to start from: place anything of type `Tensor` on the accelerator. We also chose to give tensor ops a distinct syntactic form (currently spelled `#tfop(...)`), which isn't required, but simplifies things.

Suitable compiler IR: Swift is perfectly set up to do the transformations we need, because it has a high-level IR named SIL. This represents the code in SSA with a control flow graph, preserving source level type information (so we know what values are tensors) and provides the key transformations we need like inlining, generics specialization, etc. The presence of SIL makes it significantly cheaper to build the compiler pieces of our prototype. SIL includes IR serialization, "inter-module optimization", "cross module optimization," inlining, generics specialization, and all the other basic interprocedural optimization infrastructure we need to do this without reinventing basic compiler infrastructure wheels.

Restricted pointer aliasing: Swift has an obscure but important feature called memory exclusivity which is built as part of the ongoing work to introduce a Rust inspired memory ownership model into Swift. Swift's aliasing support means that static analysis based analysis and transformations can rely on very strong memory provenance guarantees even for `inout` arguments. This provides similar analysis power to Fortran references and `restrict` pointers in C. Swift's focus on value semantics is also a major help for our analysis of arrays and other value types.

Known disadvantages of using Swift

As explained above, Swift is a good fit for this project for a number of reasons. That said, it is not perfect by any stretch of the imagination. In an effort to capture the negative side of the discussion, these are the most commonly raised concerns about using Swift for this project:

It is (relatively) new: Swift has been in development since 2010 and had its first public release in 2014. As such, it is still relatively new, and thus the tools and surrounding ecosystem aren't as mature as those for Python, which is over 25 years old.

Windows support: Swift is cross platform, and is relatively mature on UNIX-like systems, including Apple platforms, Linux, BSD, and other UNIX-y things. That said, Windows support is still relatively early-on - but there is at least one active community of people working on it.

Large dependency: Swift depends on LLVM and Clang, which means that it is a nontrivial dependency for TensorFlow. On the other hand, all alternatives (including Python) bring in a nontrivial set of dependencies. Swift is also well aligned with TensorFlow since it already depends on LLVM (through the XLA compiler backend).

Small data-science community: Swift does not have much in the way of a data science community. It has a small community that we hope to engage with, but it is very small compared to the existing NumPy, SciPy, scikit-learn and other ecosystems available for Python. Given that most of these Python libraries are implemented as C code wrapped by Python, it is possible that the Swift ecosystem will eventually grow to include Swift wrappers for the same libraries. In the immediate term though, we feel that our Python interoperability approach is a very pragmatic and very general solution to the needs of TensorFlow users.

Error messages need improvement: One of the disadvantages of type inference is that sometimes the error messages you get (particularly when playing with higher-order generic functional programming constructs) can be misleading. This is typically solved by breaking an expression into subexpressions.

Build times could be improved: Developers with large applications (e.g. over 100K lines of code) are not happy with build times. This continues to be a focus of the Swift development community and is expected to improve over time, but this shouldn't be a problem given the comparatively small size of machine learning models.

As the project evolves, it is possible that other challenges will arise. If so, let us know and we'll add them to the list.

Conclusion

In retrospect, it isn't a surprise that Swift is a good fit for the needs of this project. Swift was designed and built by a close-knit team. That team previously built a highly modular and composable compiler infrastructure (LLVM), a compiler and runtime for a highly dynamic Smalltalk-derived language (Objective-C), the compiler for a highly static language with a capable generics system (C++), and a path-sensitive static analysis engine (the Clang static analyzer). Furthermore, the goals for Swift's design was to build something that was as easy to learn and use as a scripting language, but which had enough power to be used as a system's programming language.

You can see how each of these things is used by the Swift for TensorFlow project:

- the usability goals are directly aligned between the two projects
- the highly dynamic features allow natural interoperability with Python APIs
- the powerful static side allows the language to support the static analysis that underlies Graph Program Extraction and automatic differentiation
- the generics system is essential to making automatic differentiation a user-extensible language feature
- the goal to support path-sensitive static analysis contributed to the reasons to build the SIL intermediate representation that hosts this work
- the culture of modularity and composability allows us to define much our "language" features (like the `Tensor` type) in the TensorFlow library, without having to make invasive changes to the language and compiler

The implementation of the Swift for TensorFlow features and capabilities is still not done, but we feel good about how the project is going in practice. That said, we'd really love to see the algorithms we are exploring get applied by other languages and communities!