CS131 HW6 Report: Evaluate a language for a garage-sale buyer app

Melody Chen
Discussion 1C

Abstract

In this assignment, we are asked to research and investigate whether Flutter a user interface toolkit that is written in Dart, is suitable for improving a mobile application that uses machine-learning based model to process images. To answer this, we focus on examining the programming language Dart, compare and contrast its features to OCaml, Java, and Python, and evaluate its strengths and weaknesses in building such an application.

1 Introduction

GarageGarner is an application for people who buy things from garage sales. Old version of the application is web-based with computation done in central server and only user interface is running on cell phones. To build a new version, we are asked to develop a native app with feature that allows users to take panorama of items and price tags on display, and our app will report the best deals. We do not want to ship images to central server to do the work as that is too slow, so we're looking to have most of the algorithm run on the cell phone to speed up our application. Running machine-learning based models to process images takes CPU time which causes problem on mobile device, but it can be sped up by AI accelerators on higher end phones.

To implement this new version, we choose to use TensorFlow Lite to run our machine learning model on cell phones. For writing and running the app's user interface, Flutter is considered as one of the candidates. Flutter is an open-source UI software development kit created by Google. It is relatively new and is used to develop applications for Android, iOS, Windows, Mac, etc. It is written in Dart, a client-optimized programming language that can be compiled to ARM machine code which is used on most mobile devices.

To evaluate whether Flutter is suitable candidate for our new version of GarageGarner application, we take a deeper look in to the programming language Flutter is built with, Dart. We focus on Dart version 2.7 and compare its feature to other known popular programming languages, namely OCaml, Java, and Python.

2 Dart and Other Programming Languages

Dart is a client-optimized language for apps on multiple platforms developed by Google. Client-optimized language, according to the Dart documentation, is a programming language optimized for building user interfaces. Our goal of using Flutter is to run and write the app's user interface, which matches the goal of Dart as a programming language. Dart is a multi-paradigm language that is imperative, object-oriented, and has functional components. It has a C-like syntax, which makes it easy to learn for most developers. It is also statically typed with type inference and is garbage collected. It also supports many popular features in different programming languages, such as interfaces, mixins, abstract classes, generics, and type inference. We will now look at specific features of Dart and then compare the language as a whole to OCaml, Java, and Python.

2.1 Compilation and Execution

Developers of Dart previously worked on advanced compilers and virtual machines for both dynamic and static languages. So, when they created Dart, they built it so that it supports both AOT(Ahead Of Time) and JIT(Just In Time) compilation. Typically, AOT compilation are used for static languages and dynamic languages are interpreted or uses JIT compilation. AOT compilation results in slower development cycles, but it also results in more predictable and faster execution. JIT is the opposite, where development cycle is faster, but it has a slower startup time, as before code is executed JIT does analysis and compilation. Since Dart supports both, it uses JIT compilation during development and when an app is ready for production, it is compiled using AOT. This results in what is found in almost no other programming languages—fast development cycles and execution/startup times. One notable result of JIT compilation for development is the Stateful Hot

Reload feature in Flutter, that takes mobile development to another level by allowing for changes in code to take effect in simulation almost immediately. For our GarageGarner application, this is extremely helpful as we want our application to be fast during execution to provide a smooth user experience and a fast development cycle will allow developers to spend more time working on the application rather than waiting for compilation.

In contrast, OCaml is purely compiled. This allows OCaml to have all the advantages of compiled languages, such as stable and more efficient execution. In addition, OCaml also offers a top-level interactive loop that acts very similar to interpreters. This can help speed up the development cycle as programmers can use the interactive loop to test out chunks of their written code. Overall, for our GarageGarner application, considering purely compilation and execution, Dart is a better choice than OCaml as Dart allows for a faster and more complete development cycle with features that exceed OCaml's interpreter-like interactive loop to allow for faster mobile development.

Java is both compiled and interpreted through a two-step execution. Java code first goes through an OS independent compiler that turns code into machine independent bytecode. The bytecode is then passed to JVM which interprets and translates the bytecode into native machine code. JIT compilation is used within the JVM which allows for code optimization. Advantages of being both compiled and interpreted is that once Java is compiled into bytecode it can be executed on any machine with a JVM without being re-compiled. Overall, Java is similar to Dart in that it uses JIT compilation, but Dart is still a better language for our proposed application as during production AOT compilation is used.

Lastly, Python is a purely interpreted language, so it has many of the advantages of interpreted language, such as platform independence and there is no need to re-compile program every time. Nevertheless, interpreted programs are much less efficient than regular programs by nature as every instruction is interpreted at runtime. Python will not be a good language for our proposed application in terms of compilation and execution, as during production, we want our application to be as fast as it can be.

2.2 Type Checking

Dart uses sound typing, a combination of static type checking and runtime checks to ensure that variable's value always matches variable's static type. If an expression's static type is String, at runtime you are guaranteed to get a string when expression is evaluated. Benefits of a sound type system include the ability to reveal type-related bugs at compile time, more readable code, more maintainable code, and makes AOT compilation more efficient. Runtime checks in Dart catches type safety issues that analyzer cannot catch. Type inference is also used in Dart, so type annotations are optional except for special cases. For our GarageGarner application, sound typing works well as we don't want our application to crash during run-time due to type-related bugs and sound typing allows us to take full advantage of AOT compilation.

OCaml uses strong static typing with type inference. Before OCaml code is executed, the compiler checks for type errors. Most of the static type checking languages requires explicit type declarations which can be tedious. OCaml avoid this with type inference, where there is no need to write explicit type declarations as compiler infers type for you. In terms of only type checking, OCaml would work well for our GarageGarner application as its very similar to Dart.

On the other hand, Python is a dynamic typed programming language, where Python interpreter does type checking as code is being run. Benefits of dynamic typed language is that there is no need to explicitly write type declarations, which makes the language a lot simpler and more concise. However, the downside is that type-related bugs will be discovered at tun time, which we do not want for our GarageGarner application as it crashes the program and hinders performance.

2.3 Memory Management

Dart uses Garbage Collection, a necessary component for allocating and deallocating memory when objects are no longer being used. This way developers do not have to worry about explicitly allocating memory and releasing memory. Dart's garbage collection can be performed most of the time without locks, which boosts its performance as locks often hinder app from running smoothly. It uses an advanced generational garbage collection and allocation scheme that is based on empirical observation that most objects die young. New(younger) objects are more regularly collected than older objects. This scheme is suitable for user interfaces, like our proposed application, as objects are often rebuilt for every frame.

OCaml also uses Garbage Collection that uses similar idea to generational collectors. It follows the principle where most objects are small and allocated frequently and then freed immediately. So OCaml's garbage collector uses two heaps: minor heaps and major heap. Minor heap is collected more frequently than major heaps. In addition, OCaml's GC is synchronous, so it does not have to deal with locks, and is called only when objects are allocated. Similar to Dart, OCaml's garbage collection scheme is suitable for our application.

Java's garbage collection does a generation-based collector for some objects and periodically runs the mark-and-sweep algorithm for other objects. Downside of the mark-and-sweep algorithm is that it runs in a separate thread, so it needs to acquire the lock in order to free objects which turn into bottleneck. With this, Java's garbage collection often has to deal with locks that can cause overhead, so its garbage collection scheme is not as suitable for our proposed application.

Python's garbage collection method is relatively simple, it is based on the idea of reference counts. As soon as the reference count of an object reaches 0, the object's memory is automatically freed. The advantage of Python's Garbage Collection is that it is simple and fast to keep track of reference counts of all objects. Nevertheless, it is inefficient against circular references. However, if we avoid circular references in our proposed application, then Python's garbage collection is quite suitable due to its simplicity and speed.

2.4 Multithreading and Event Loops

To avoid having to deal with race conditions for concurrent threads, Dart took a different approach that uses isolates, which are threads that do not share memory. This avoids the need for most locks. Isolates communicate by passing messages over channels. In addition, Dart is also single threaded, meaning threads explicitly yield. For our proposed mobile application, single threading helps ensure that functions, especially animation and transitions, are executed to completion. Dart uses event loops in order for the single-threaded application to handle multiple different events coming in unknown order. The event loop is very similar to that of Python's asyncio and the loop applies within the same isolate.

OCaml is similar to Dart as it supports concurrency, but not parallelism. OCaml's threading is supported by the global interpreter lock, which is very similar to Python's GIL. This allow race conditions to be avoided in OCaml. So, like Dart, its multithreading seems to be suitable for our proposed user interface application.

We can compare these two models to Java's Memory Model where multi-threading and parallelism is common. With Java's choice of allowing for multi-threading, it increases its risks of race condition and relies on more complicated synchronization methods and locks to avoid race conditions. For certain applications, to be able to use parallelism can significantly improve performance, but for our GarageGarner application where we use external API for our machine learning model, parallelism may not necessarily improve its performance.

Python behaves very similar to OCaml in terms of multithreading.

3 Strengths and Weaknesses of Dart

Major strengths of using Dart for our proposed application is that many of its features are built for writing an application's user interface. For example, the combination of JIT and AOT compilation for development vs production helps programmers not only speed up development process but ensures fast execution when application is released. Garbage collection for Dart is also suitable for user interface as many objects are only used in certain frames and then no longer used afterwards which a generational garbage collection scheme performs well in. Other strengths of Dart for our proposed application is mentioned in detail in section above. In contrast to OCaml, Python, and Java, Dart is more suitable for building our GarageGarner application in almost every aspect.

Nevertheless, Dart has some inherent weaknesses for building our proposed application. We previously mentioned that instead of threads, Dart use isolates to support concurrency. Isolates do not share memory, so they are a heavyweight thread control model. Avoiding shared memory means avoiding many potential race conditions, but in turn this causes isolates to use a lot of memory as spawning a new isolate needs to clone all objects and data structures of the running program. Also, controlling user interface requires dealing with a high level of interactivity with many widgets and animations occurring simultaneously. Dart's single thread allows for animation and transitions to execute to completion without preemption, but if we want to animate multiple widgets at once, each one requires an isolate, which leads to overhead.

Considering that Dart's weakness mainly has to do with its support for concurrency and the overhead caused by simultaneously having multiple widgets and animation, I still think Dart is suitable for our proposed application as the goal of GarageGarner is to have a fast mobile application that can process images via the TensorFlow Lite API. We do not require multiple widgets appearing simultaneously in our application and we can easily avoid over-complicated animations for our purpose. Running our machine learning model is heavy duty, but that is not done within Dart but via external API. Overall, most of the strengths of Dart mentioned above especially for user interface development suits the need of our GarageGarner application very well despite the weaknesses mentioned.

4 Conclusion

Overall, I believe Flutter to be an appropriate choice for writing and running GarageGarner's user interface with tflite, a TensorFlow Lite API, running the machine learning based model to process images. This is because Flutter apps are written in Dart and we have examined aspects of Dart throughout the assignment and established that it is a very suitable language for writing user interface applications despite some weaknesses. TensorFlow Lite API can be easily integrated with our flutter app using tflite. With tflite API, we don't have to worry about implementing the model in Dart as it is taken care of externally. The major problem we have to be careful about when implementing our application with Dart is to not have too many widgets in one frame with animation and transition as that would cause overhead for Dart isolates. We compared Dart to other popular languages, namely OCaml, Python, and Java, and established that it is the most suitable for building user interfaces, mainly because of its compilation/execution, memory management, and concurrency scheme.

Acknowledgments

Thank you to Professor Eggert and the TAs for introducing and grading our homeworks.

References

Dart Dev Documentation
https://dart.dev/guides/language/sound-dart
Java Type Inference
https://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html
TA Patricia Xiao and TA Kimmo Karkkainen's Slides
OCaml Static Typing
https://www2.lib.uchicago.edu/keith/ocaml-class/static.html'
Garbage Collection
https://en.wikipedia.org/wiki/Garbage_collection_(computer_science) Flutter Features
https://hackernoon.com/why-flutter-uses-dart-dd635a054ebf
Multicore OCaml
http://ocamllabs.io/doc/multicore.html
Garbage Collection in Java
https://medium.com/computed-comparisons/garbage-collection-vs-automatic-reference-counting-a420bd