EDUCACIÓN

SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

Instituto Tecnológico de Orizaba

"2021: Año de la Independencia"

DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

OPCIÓN I.- TESIS

TRABAJO PROFESIONAL

"PROGRAMACIÓN POLÍGLOTA CON LA MÁQUINA VIRTUAL GRAAL"

PARA OBTENER EL GRADO DE:

MAESTRO EN SISTEMAS COMPUTACIONALES

PRESENTA:

L.I. José Antonio Romero Ventura

DIRECTOR DE TESIS:

Dr. Ulises Juárez Martínez

CODIRECTOR DE TESIS:

Dr. Adolfo Centeno Téllez

ORIZABA, VERACRUZ, MÉXICO. SEPTIEMBRE 2021

"2021: Año de la Independencia"

Orizaba, Veracruz, 29/septiembre/2021
Dependencia: División de Estudios de Posgrado e Investigación
Asunto: Autorización de Impresión
OPCION: I

C. JOSÉ ANTONIO ROMERO VENTURA
Candidato a Grado de Maestro en:
SISTEMAS COMPUTACIONALES
P R E S E N T E.-

De acuerdo con el Reglamento de Titulación vigente de los Centros de Enseñanza Técnica Superior, dependiente de la Dirección General de Institutos Tecnológicos de la Secretaría de Educación Pública y habiendo cumplido con todas las indicaciones que la Comisión Revisora le hizo respecto a su Trabajo Profesional titulado:

" Programación políglota con la máquina virtual Graal".

Comunico a Usted que este Departamento concede su autorización para que proceda a la impresión del mismo.

A T E N T A M E N T E
Excelencia en Educación Tecnológica®
CIENCIA – TÉCNICA - CULTURA®

EDUCACIÓN | TECNOLÓGICO NACIONAL DE MÉXICO
INSTITUTO TECNOLÓGICO DE ORIZABA
DIV. DE EST. DE POSGRADO E INVEST.

Dr. MARIO LEONCIO ARRIOJA RODRÍGUEZ
JEFE DE LA DIVISIÓN DE ESTUDIOS
DE POSGRADO E INVESTIGACIÓN

OG-13-F06

EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

Instituto Tecnológico de Orizaba
División de Estudios de Posgrado e Investigación

"2021: Año de la Independencia"

Orizaba, Veracruz, 10/septiembre/2021
Asunto: **Revisión de trabajo escrito**

C. MARIO LEONCIO ARRIOJA RODRÍGUEZ
JEFE DE LA DIVISIÓN DE ESTUDIOS
DE POSGRADO E INVESTIGACIÓN
P R E S E N T E.-

Los que suscriben, miembros del Jurado, han realizado la revisión de la Tesis del (la) C.

JOSÉ ANTONIO ROMERO VENTURA

la cual lleva el título de:

"Programación políglota con la máquina virtual Graal"

y concluyen que se acepta.

A T E N T A M E N T E
Excelencia en Educación Tecnológica®
CIENCIA – TÉCNICA - CULTURA®

PRESIDENTE:	DR. ULISES JUÁREZ MARTÍNEZ	FIRMA
SECRETARIO:	DRA. LISBETH RODRÍGUEZ MAZAHUA	FIRMA
VOCAL:	M.C. MA. ANTONIETA ABUD FIGUEROA	FIRMA
VOCAL SUP.:	DR. ADOLFO CENTENO TÉLLEZ	FIRMA

TA-09-26

Avenida Oriente 9 No. 852
Col. Emiliano Zapata, C.P. 94320
Orizaba, Veracruz, México.
Teléfono: 272-110-53-60
Email: depi_orizaba@tecnm.mx
www.orizaba.tecnm.mx

México 2021 Año de la Independencia

Agradecimientos

A mi madre, con su apoyo incondicional, fue un pilar muy fuerte durante todo este camino, que no solo inició desde el posgrado, desde siempre, me impulsó para ser un hombre de bien y dedicado en todo lo que hago, ya sea en lo académico y lo profesional.

En memoria de mi tío, el Ing. Químico José Luis Ventura Silvestre, quien me inculcó desde niño la cultura de la educación, la preparación académica, la importancia del trabajo profesional y honesto.

Al Dr. Ulises Juárez Martínez, por su apoyo y orientación académica durante el desarrollo de la presente tesis.

Al CONACYT (Consejo Nacional de Ciencia y Tecnología), por su apoyo otorgado, no solo en lo económico, sino también, por su respaldo como institución al momento de la presentación de trabajos de investigación en eventos de talla nacional e internacional.

Índice general

Índice de figuras

Índice de tablas

Índice de Códigos

Resumen

Hoy en día, el desarrollo de aplicaciones hace uso de entornos separados con el propósito de que los lenguajes de programación usados sean capaces de ejecutarse en ambientes con características específicas, pero esto implica que sea difícil, tardado y costoso el desarrollo y mantenimiento de dichas aplicaciones. Como solución, la programación políglota con *GraalVM* permite el desarrollo de aplicaciones usando más de un lenguaje de programación a la vez, en un mismo entorno de desarrollo y permitiendo la comunicación entre diferentes lenguajes de programación.

En la presente tesis, como aplicación de la metodología, se emplea la programación políglota en dos escenarios, uno para explorar las capacidades nativas de *GraalVM* y otro para explorar sus capacidades de interoperabilidad usando *Node.js*, para observar el potencial de la programación políglota y cómo facilita el desarrollo de aplicaciones al momento de combinar lenguajes de programación que de manera cotidiana se encuentran en entornos separados y se comunican por medio de *API's*. Y como aplicación de resultados se orientó la programación poliglota con *GraalVM* hacia la bioinformática, específicamente para el análisis de secuencias de ADN por medio de una aplicación en *Java* con *Python*, desde una imagen nativa usando *Dockers*, haciendo una comparación de desempeño entre estos elementos y concluir cuál es la mejor opción para implementar.

Abstract

Nowadays, application development makes use of separate environments for the purpose of that the programming languages used are capable of running in environments with specific characteristics, but this implies that it is difficult, time-consuming and expensive to development and maintenance of these applications. As a solution, polyglot programming with GraalVM allows the development of applications using more than one programming language at the same time, in the same development environment and allowing communication between different programming languages.

In this thesis, as an application of the methodology, the programming polyglot is used in two scenarios, one to explore the native capabilities of GraalVM and another to explore its interoperability capabilities using Node.js, to observe the potential of polyglot programming and how it facilitates application development by time to combine programming languages that are found on a daily basis in separate environments and communicate through APIs. And as application results, polyglot programming with GraalVM is applied to bioinformatics, specifically for DNA sequence analysis by means of a Java application with Python, from a native image using Dockers, making a performance comparison between these elements and conclude which is the best option to implement.

Introducción

Hoy en día, el desarrollo de aplicaciones se lleva a cabo por medio de diferentes lenguajes de programación [1], ya sea en el desarrollo web [2], aplicaciones de escritorio, aplicaciones de dispositivos móviles o aplicaciones en la nube. Cada uno de los lenguajes requeridos necesita un entorno específico para ejecutarse, ya sea un *browser*, un servidor de aplicaciones, una máquina virtual o un conjunto de intérpretes, así como el uso de entornos separados por sus características de ejecución o compilación. En consecuencia, se tiene el uso de diferentes servidores, aplicaciones en equipos separados debido a ciertas características del programa y el uso de *API's* (*Application Programming Interface*, Interfaz de programación de aplicaciones) que permitan la comunicación entre programas para el envío de mensajes o ejecución de tareas, estas *API's* en ocasiones se desarrollan por terceros.

GraalVM [3] (*Graal Virtual Machine*, Máquina Virtual de Graal) cuenta con un amplio soporte de lenguajes de programación e interoperabilidad entre estos, así como la ejecución de lenguajes de programación a nivel nativo. *GraalVM* es compatible con sistemas operativos de *Linux, MacOS* y recientemente con *Windows*, aunque la versión de este último aún es versión beta. Estas características permiten ubicar a *GraalVM* como una plataforma universal multilenguaje de alto desempeño para el desarrollo de aplicaciones compiladas.

Considerando lo anteriormente mencionado, en la presente tesis se propone el uso de la programación políglota con *GraalVM* para el desarrollo de aplicaciones ya sean *web*,

de escritorio o incluso para dispositivos móviles haciendo uso de diferentes lenguajes de programación en un mismo entorno, esto para aprovechar las capacidades de cada lenguaje de programación y así desarrollar una mejor y más potente aplicación.

La presente tesis está estructurada de la siguiente forma:

En el capítulo uno se presentan los antecedentes, que comprenden el marco teórico, la situación tecnológica, económica y operativa de la empresa, el planteamiento del problema, los objetivo general y específicos, y su respectiva justificacion; el capítulo dos está conformado por el estado de la práctica y la solución propuesta al problema planteado; en el capítulo tres se realiza la aplicación de la metodología, en esta sección se presentan dos escenarios de programación políglota usando *GraalVM*. El primer escenario explora las capacidades nativas y demuestra el potencial de *GraalVM* con diferentes lenguajes de programación; el segundo escenario, con base en un caso de estudio, demuestra las capacidades de interoperabilidad de *GraalVM* usando *Node.js* [4] como *framework*, explorando las capacidades políglotas con diferentes lenguajes de programación, como lo es *Java, Ruby, Python* [5] y *C*. Se analiza y se observa cómo es el desempeño de *GraalVM* en cada uno de los escenarios y cómo es que se comunican los diferentes lenguajes de programación entre sí, lenguajes que comúnmente se encuentran en entornos separados; el capítulo cuatro comprende la presentación de resultados al aplicar la programación políglota a un requerimiento del entorno laboral o académico real, en este caso se presenta la creación de una aplicación de servicios que permite el análisis de secuencias de ADN usando las propiedades políglotas de *GraalVM*, dicha aplicación se genera en lenguaje *Java* como lenguaje *Host* del entorno políglota y usa como lenguaje *Guest* el lenguaje *Python*, dicho lenguaje se usó para el análisis de las secuencias de ADN. La aplicación se genera como una aplicación de *Java*, y como una imagen nativa con *GraalVM*, finalmente se crea la imagen nativa dentro de un contenedor *Docker*. Después, se realiza el análisis de los tiempos de ejecución de las diferentes aplicaciones, y se analiza cuál es la mejor opción para implementar en un proyecto; y por último el capítulo cinco menciona las respectivas

conclusiones con base en los resultados obtenidos de las experimentaciones realizadas con la programación políglota con *GraalVM*.

Capítulo 1

Antecedentes

En este capítulo, se presentan los conceptos, términos y fundamentos que se usaron durante el proceso de desarrollo de tesis.

1.1. Marco teórico

En esta sección se presentan las definiciones de los términos y tecnologías que se usaron en la investigación y aplicación de la programación políglota usando la máquina virtual de *Graal* y sus predecesores.

1.1.1. *Java*

Java [6] es un lenguaje de programación y una plataforma informática lanzada por primera vez 1995 por "*Sun Microsystems*". Hay muchas aplicaciones y sitios *web* que no funcionarían sin *Java* instalado. *Java* es rápido, seguro y confiable.

1.1.2. *JVM*

La *JVM* [6] (*Java Virtual Machine*, Máquina Virtual de *Java*) es una máquina de computación abstracta o una interfaz de máquina virtual que controla el código *Java*. *JVM* es el motor que impulsa el código *Java*. Principalmente en otros lenguajes de programación,

el compilador produce código para un sistema en particular, pero el compilador *Java* produce *Bytecodes* para una máquina virtual *Java*. Cuando se compila un programa *Java*, se genera *bytecode*. Es el medio que compila el código *Java* para bytecode que se interpreta en una máquina diferente y, por lo tanto, lo hace independiente del sistema operativo.

Java es de plataforma independiente debido a su *JVM*. Como las diferentes computadoras con diferentes sistemas operativos tienen su *JVM*, cuando se envía un archivo *.class* a cualquier sistema operativo, *JVM* interpreta el código de *bytes* en lenguaje de máquina.

1.1.3. *Java HotSpot VM*

La tecnología *Java HotSpot VM* [7] proporciona la base para la plataforma *Java SE*, la principal solución para desarrollar e implementar rápidamente aplicaciones empresariales y de escritorio críticas para el negocio. La tecnología *Java SE* está disponible para *Solaris Operating Environment (OE)*, *Linux* y *Microsoft Windows*, así como para otras plataformas a través de licenciatarios de tecnología *Java*.

Java HotSpot VM se basa en el soporte multiplataforma de la tecnología *Java* y el modelo de seguridad robusto con nuevas características y capacidades para escalabilidad, calidad y rendimiento. Además de las nuevas funciones, esta versión es compatible con versiones anteriores.

1.1.4. *JIT*

Java HotSpot VM de *Oracle* está equipado con un compilador *JIT* [8] *(Just-In-Time)* altamente avanzado. Esto significa que los archivos de clase (que se compilan a partir del código fuente de *Java*) se compilan en tiempo de ejecución y se convierten en un código de máquina muy optimizado. Este código optimizado se ejecuta extremadamente rápido, generalmente tan rápido como (y, en ciertos casos, más rápido que) el código *C / C ++*

compilado.

El compilador *JIT* es, por lo tanto, una de las partes más importantes de *Java HotSpot VM* y, sin embargo, muchos desarrolladores de *Java* no saben mucho sobre él o cómo comprobar que sus aplicaciones funcionan bien con el compilador *JIT*.

Compilación básica de *JIT*

Java HotSpot VM [8] supervisa automáticamente qué métodos se están ejecutando. Una vez que un método se vuelve elegible (al cumplir con algunos criterios, como ser llamado con frecuencia), se programa para su compilación en código de máquina y luego se lo conoce como método activo. La compilación en código de máquina ocurre en un subproceso *JVM* separado y no interrumpirá la ejecución del programa. De hecho, incluso mientras el hilo del compilador está compilando un método activo, la *JVM* seguirá utilizando la versión original interpretada del método hasta que la versión compilada esté lista.

Compiladores *JIT*

Un compilador *JIT* es un compilador en línea que genera código para una aplicación (o biblioteca de clases) durante la ejecución de la propia aplicación. Un compilador *JIT* [9] puede crear código de máquina poco antes de la primera invocación de un método *Java*. Los compiladores de puntos de acceso normalmente le dan al intérprete tiempo suficiente para preparar los métodos *Java*, ejecutándolos miles de veces. Este período de preparación permite al compilador tomar mejores decisiones de optimización, ya que observa (después de la carga inicial de clases) una jerarquía de clases más completa. El compilador también puede inspeccionar la información de perfil de tipo y rama recopilada por el intérprete.

1.1.5. Compilador *AOT*

El compilador *AOT* [10] (*Ahead-Of-Time*) compila dinámicamente métodos *Java* en código *AOT* nativo en tiempo de ejecución y los almacena en la caché de clases com-

partidas. Esta actividad permite que la máquina virtual inicie una aplicación más rápido la próxima vez que se ejecute porque no necesita perder tiempo interpretando métodos *Java*. La máquina virtual elige automáticamente qué métodos deben compilarse con *AOT* basándose en heurísticas que identifican la fase de inicio de aplicaciones grandes. El código *AOT* siempre se usa en combinación con el intercambio de datos de clases y se habilita automáticamente cuando `-Xshareclasses` está configurado en la línea de comando. Cuando se ejecuta un método *AOT* en caché, el compilador *JIT* también puede optimizarlo.

1.1.6. *GraalVM*

GraalVM [11] es una máquina virtual universal para ejecutar aplicaciones escritas en *JavaScript, Python, Ruby, R,* lenguajes basados en *JVM* como *Java, Scala, Groovy, Kotlin, Clojure* y lenguajes basados en *LLVM* (ver sección 1.1.11) como *C* y *C++*. *GraalVM* elimina el aislamiento entre lenguajes de programación y permite la interoperabilidad en un tiempo de ejecución compartido. Se ejecuta de forma independiente o en el contexto de *OpenJDK*, *Node.js* u *Oracle Database*.

Cualquier aplicación basada en *JVM* [12] que se ejecute en *Java HotSpot VM* puede ejecutarse en *GraalVM*. *GraalVM* se basa en *Java HotSpot VM*, pero integra un compilador avanzado *JIT*, escrito en *Java*: el compilador *GraalVM*. En tiempo de ejecución, la aplicación se carga y ejecuta normalmente en la *JVM*. La *JVM* pasa el código de *bytes* al compilador *GraalVM*, que lo compila en el código de la máquina y lo devuelve a la *JVM*.

El compilador dinámico de *GraalVM* puede mejorar la eficiencia y la velocidad de las aplicaciones escritas en *Java, Scala, Kotlin* u otros lenguajes *JVM* a través de enfoques únicos para el análisis y la optimización de código. Por ejemplo, asegura ventajas de rendimiento para programas altamente abstraídos debido a su capacidad para eliminar costosas asignaciones de objetos.

1.1.7. Lenguaje de programación dinámico

Un lenguaje de programación dinámico [13] es aquel en el que las operaciones realizadas durante la compilación pueden realizarse también durante la ejecución del programa. Un ejemplo es *JavaScript*, en el cual es posible cambiar el tipo de una variable, agregar nuevas propiedades o métodos a un objeto mientras el programa se encuentra en ejecución.

Esto es lo contrario a los lenguajes de programación estáticos, en los que tales cambios no son posibles.

1.1.8. *Truffle*

Truffle [14] es una biblioteca de código abierto para construir implementaciones de lenguajes de programación como intérpretes para árboles de sintaxis abstracta auto modificables. Junto con el compilador *GraalVM* de código abierto, *Truffle* [15] representa un importante paso adelante en la tecnología de implementación de lenguajes de programación en la era actual de los lenguajes dinámicos.

Un conjunto cada vez mayor de servicios y códigos implementados compartidos reduce significativamente el esfuerzo de implementación del lenguaje, pero conduce a un rendimiento competitivo en tiempo de ejecución que coincide o supera esta competencia. El valor de la plataforma se incrementa aún más con el soporte para la interoperación de lenguaje de bajo costo, así como con un marco de instrumentación general que admite la depuración en varios lenguajes de programación y otras herramientas de desarrollo externas.

Truffle se desarrolla y mantiene por *Oracle Labs* y el *Institute for System Software* de la Universidad *Johannes Kepler* de Linz.

1.1.9. *AST*

En lenguajes formales y lingüística computacional, un *AST* [16] (*Abstract Syntax Tree*, Árbol de Sintaxis Abstracta) es la representación de un árbol de la estructura sintáctica simple del código fuente escrito en cierto lenguaje de programación. Cada nodo del *AST* representa una construcción que ocurre en el código fuente. La sintaxis es abstracta ya que no representa cada detalle que se encuentre en la sintaxis verdadera. Como por ejemplo, el agrupamiento de los paréntesis se cuentra implícito en la estructura arborescente, y una construcción sintáctica tal como *IF* condición *THEN* se denota por un solo nodo con dos ramas.

Esto hace a los árboles de sintaxis abstracta diferentes de los árboles de sintaxis concreta, llamados tradicionalmente árboles de parser, que se construyen por la parte parser de la traducción del código fuente y el proceso de compilación (a pesar quizás de un nombramiento no intuitivo). Una vez construido, información adicional se agrega al *AST* por procesamiento subsecuente, p. ej., análisis semántico, tal como se observa en la Figura 1.1.

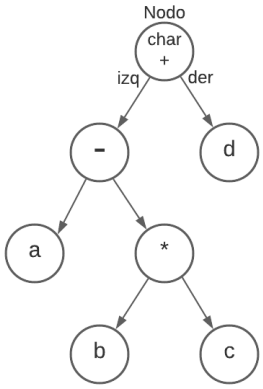

Figura 1.1: Árbol *AST* de una expresión postfija abc*-d+

1.1.10. *WebAssembly*

WebAssembly (*Wasm* abreviado) [17] es un formato de instrucciones binarias para una máquina virtual basada en pilas. *Wasm* está diseñado como un elemento portátil para la

compilación de lenguajes de alto nivel como *C/C++/Rust*, lo que permite la implementación en la *web* para aplicaciones de cliente y servidor. También tiene grandes implicaciones en entornos *web*, ya que provee una forma de ejecutar código escrito en múltiples lenguajes en la *web* a una velocidad casi nativa, con aplicaciones cliente corriendo en la web que anteriormente no podrían haberlo hecho. *WebAssembly* está diseñado para complementar y correr a la par de *JavaScript* usando diferentes *API WebAssembly* de *JavaScript*, con ésta característica se cargan los módulos de *WebAssembly* en una aplicación *JavaScript* para compartir funcionalidad y datos entre ambos. Esto permite aprovechar el rendimiento y poder de *WebAssembly* y la expresividad y flexibilidad de *JavaScript* en las mismas aplicaciones. *WebAssembly* está siendo desarrollado como un estándar *web* a través del grupo de trabajo de *W3C* y su grupo comunitario con una participación activa de los principales fabricantes de *browsers*.

La máquina de pila *Wasm* [17] está diseñada para codificarse en un formato binario eficiente, en cuanto a su tamaño y tiempo de carga. *WebAssembly* tiene como objetivo ejecutar a la velocidad nativa aprovechando las capacidades comunes de *hardware* disponibles en una amplia gama de plataformas. Tambien describe un entorno de ejecución de espacio aislado seguro para la memoria que incluso se implementa dentro de máquinas virtuales *Java*Script existentes. Cuando se incluye en la web, por medio de tecnologías como *Python*, *WebAssembly* aplicará las políticas de seguridad del mismo origen y permisos del *browser*. También mantiene la naturaleza de la *web* sin versiones, probadas y compatible con versiones anteriores. Los módulos de *WebAssembly* tienen la capacidad entrar y salir del contexto de *JavaScript* y acceder a la funcionalidad del *browser* a través de las mismas *API web* accesibles desde *JavaScript*. *WebAssembly* también admite incrustaciones no *web*, es decir, ejecutar líneas de código que no son propias de lenguajes de programación *web*.

GraalVM ejecuta programas compilados en *WebAssembly* [3]. Tambien interpreta y compila código *WebAssembly* en formato binario o incrustarlo en otros programas. El soporte para *WebAssembly* se encuentra en las primeras etapas de su desarrollo en *GraalVM*.

1.1.11. *LLVM*

El proyecto *LLVM* [18] (*Low Level Virtual Machine*, Máquina Virtual de Bajo Nivel) es una colección de compiladores modulares y reutilizables, y también de tecnologías de cadena de herramientas.

LLVM inició como un proyecto de investigación en la Universidad de *Illinois*, su objetivo era el proporcionar una estrategia de compilación moderna basada en *SSA* (*static single assignmen*t, asignación estática única) capaz de soportar compilaciones estáticas y dinámicas de lenguajes de programación arbitrarios. Desde entonces, *LLVM* se convirtió en un proyecto general que consta de una serie de subproyectos, muchos de los cuales se utilizan en la producción por una amplia variedad de proyectos comerciales y de código abierto, además de ser ampliamente utilizados en la investigación académica. El código en el proyecto *LLVM* está licenciado bajo la "Licencia Apache 2.0 con excepciones *LLVM*".

1.2. Situación tecnológica, económica y operativa de la empresa

El Instituto Tecnológico de Orizaba cuenta con un amplio panorama de carreras, a nivel de licenciatura y posgrados como maestrías y un doctorado. Entre estas carreras a nivel licenciatura se encuentran Ingeniería Eléctrica, Ingeniería Electrónica, Ingeniería en Sistemas Computacionales, Ingeniería Química, Ingeniería Mecánica, Ingeniería en Gestión Empresarial, Ingeniería Industrial e Ingeniería Informática, y a nivel maestría como Maestría en Ingeniería Electrónica, Maestría en Ingeniería en Industrial, Maestría en Sistemas Computacionales, Maestría en Ingeniería Administrativa y Maestría en Ciencias en Ingeniería Química y un doctorado en Ciencias de la Ingeniería. Principalmente en carreras orientadas a la computación como lo son Sistemas Computacionales o Informática tienen la programación como su día a día, manejando diferentes tecnologías y lenguajes de programación innovadores y que están a la vanguardia, sin embargo, estos lenguajes necesitan

de diversas tecnologías para la codificación, ejecución e implementación de sus proyectos, incluso es necesario el manejo de más de un lenguaje en un solo proyecto, gracias a eso, se recomienda el manejo de ambientes de programación políglota.

1.3. Planteamiento del problema

El desarrollo actual de aplicaciones utiliza varios lenguajes de programación para hacer eficiente el procesamiento de datos y ejecución de algoritmos. En el contexto de la *Web* es común utilizar *HTML* (*HyperText Markup Language*, lenguaje de marcas de hipertexto), *JavaScript*, *SQL* (*Structured Query Language*, lenguaje de consulta estructurado) y al menos un lenguaje de *scripting*; para dispositivos móviles se tienen enfoques multiplataforma (*cross-platform*); y en aplicaciones de escritorio se cuenta con máquinas virtuales [19] para la independencia de plataforma. La comunicación de estos componentes en sus diferentes entornos hoy en día se lleva a cabo por medio de distintas *API* o *software* de terceros, por ello se observa la necesidad de hacer interoperabilidad entre lenguajes de programación que no necesariamente son compatibles entre sí. Por otro lado, diversos principios de la ingeniería del *software*, como la separación de asuntos, pretende obtener componentes de acuerdo con el lenguaje que mejor permita la implementación limpia de requisitos de *software* y esto no necesariamente se logra en un solo lenguaje.

Para resolver el problema se utilizó el enfoque de la programación políglota para la combinación de lenguajes de programación. Dentro de la tecnología a utilizar se cuenta con la máquina virtual *Graal* que permite la ejecución de entornos políglotas, ya sea en ambientes de consola usando un compilador/intérprete políglota, imágenes de código con lenguajes combinados para visualizar el potencial de estos mismos, o en ambientes gráficos potenciados por *Python*. Como parte de los métodos a considerar se utilizará el principio de separación de asuntos para estructurar y mantener el modularidad en la programación políglota.

1.4. Objetivo general y específicos

En esta sección se mencionarán los objetivos tanto general como específicos para resolver el problema planteado.

1.4.1. Objetivo General

Estudiar las capacidades de interoperabilidad de lenguajes de programación que posee la máquina virtual universal (*Universal VM*) para el desarrollo de soluciones multiplataforma utilizando estrategias de programación políglota.

1.4.2. Objetivos específicos

- Revisar los fundamentos de la programación políglota para el desarrollo de soluciones multiplataforma.
- Utilizar la Máquina Virtual *Graal* para ejecutar aplicaciones escritas en diferentes lenguajes de programación.
- Desarrollar un escenario que muestre las capacidades de interoperabilidad y nativas de *GraalVM* a partir *de Node.js*.
- Desarrollar un caso de estudio de la industria que muestre las capacidades de *GraalVM*.

1.5. Justificación

El desarrollo de *software* frecuentemente utiliza diversas *API* para proveer interoperabilidad entre lenguajes de programación lo cual conlleva a instalar productos de terceros. La programación políglota permite la eliminación de estas *API* obteniendo mayor claridad y comprensión en el código, y al mismo tiempo, implementar cada parte del problema en el mejor lenguaje de programación.

Por ejemplo, una aplicación para estadística hace uso del lenguaje *R* directamente

desde el código de *Java*, o una aplicación de inteligencia artificial generativa (*AI + Machine Learning*) que interactúa con *Python* desde *Java*.

La programación políglota es un enfoque que permite usar diferentes lenguajes de programación para solventar parte de las limitaciones en obtener soluciones con enfoques dominantes por un solo lenguaje de programación. Como parte de la solución, la constante evolución de la Máquina Virtual de *Java* permite actualmente considerarla como la máquina virtual universal (*Universal VM*) al facilitar la interacción e interoperabilidad de diferentes lenguajes de programación, con capacidades nativas, sin sobrecarga en tiempo de ejecución y con propiedades embebibles. Como ejemplos de tales capacidades está la interoperabilidad entre lenguajes como *Ruby, R, Python, C, C++, Scala* e incluso con ambientes de ejecución como *Node.js*.

La comunidad de programadores que manejan múltiples lenguajes de programación en un solo proyecto, ya sea por optimización de recursos o porque el proyecto así lo requiere, serán los beneficiados con esta investigación.

Capítulo 2

Estado de la práctica

Hoy en día, el desarrollo de aplicaciones hace uso de varios lenguajes de programación para hacer eficiente el procesamiento de datos y ejecución de algoritmos. En el ámbito *Web* es común utilizar *HTML* con múltiples lenguajes de programación, tales como *JavaScript*, *SQL* para el procesamiento de datos y algún lenguaje de uso en servidor. En los ambientes móviles usan lenguajes combinados con presentación de datos y uso de distintas *API* para manejar otras funciones que están en un lenguaje diferente o usan herramientas ajenas al dispositivo móvil. En el área de aplicaciones de escritorio, se utilizan las máquinas virtuales, tales como la *JVM* de *Oracle* y la *MCLR* (*Microsoft's Common Language Runtime*, máquina virtual de *.NET framework*). Sin embargo, todas esas tecnologías manejan un solo lenguaje de programación a la vez, por eso, es necesario el uso de la programación políglota, para implementar nuevas y mejores soluciones y mejorar la comunicación entre herramientas de *software* o con otros programas externos.

2.1. Trabajos relacionados

En [20] se habla acerca de cómo la técnica *Escape Analysis* (Análisis de Escape) le permite al compilador determinar si un objeto es accesible de manera externa desde un método o un hilo. Esta información se utilizó para realizar optimizaciones como *Scalar Replacament* (Reemplazo Escalar), *Stack Allocation* (Asignación de pilas) y *Lock Elision*

(Bloqueo de elisión), permitiendo que los compiladores modernos remuevan abstracciones introducidas por modelos avanzados de programación.

En muchos casos, el tomar una decisión global acerca de la escalabilidad de los objetos no le permite al compilador llevar a cabo procesos anteriores. *Partial Escape Analysis* (Análisis de Escape Parcial) es particularmente efectivo si interactúa con otras partes del compilador como es el alineado, enumerado de variables globales y plegado de constantes, para esto, es necesario que trabaje con la misma representación del programa interno y otras optimizaciones como es el caso de *Graal* se usa *Graal IR* (*Graal Intermediate Representation*, Representación intermedia de *Graal*). *Graal IR* inicia la iteración en un nodo inicial de acciones y procesa cada nodo tan pronto como cada uno de los flujos de control predecesores se completaron, esto significa que seguirá un control de flujo, tendrá ramificaciones y uniones dentro del mismo flujo. La iteración termina cuando se termine el flujo con el retorno de algún valor o se genere una excepción. Debido a que tendría múltiples ramificaciones, existirán nodos de unión, cada ramificación tendrá un estatus de nodo, por lo tanto, estos estatus deberán unirse en un estatus único y consistente a lo largo del flujo. Durante los procesos de unión los objetos virtuales se convertirían en objetos de escape que podrían invalidar supuestos objetos virtuales en procesos anteriores de la misma ramificación.

Como resultado se obtiene que un alto número de *bytes* colocados también demuestran un algo número de espacios de memoria. También se realizó una significante reducción de la cantidad de bloqueos de memoria. En la mayoría de los puntos de referencia usados, se observó una mejora del comportamiento de un 10 %. Como trabajo a futuro se tiene que la iteración que actualiza el estatus de la localización podría hacerse de manera paralela al momento que se encuentre un salto en el control del flujo.

En [21] se habla acerca del manejo de *SP* (*Stored Procedures*, procedimientos almacenados) con *GraalVM*. A menudo los *SP* se usan para el manejo centralizado de la lógica de

negocio en un sistema que se ejecutan dentro de la misma base de datos, ayudan a evitar el uso excesivo de la red, memoria y manejo directo de los datos en el sistema.

Sin embargo, a pesar de estos beneficios los *SP* también tienen características que se consideran perjudiciales para usar como lo son: algunos *SP* tienen especificaciones propias de proveedores que los haces diferentes; los programadores especializados en algunos SP son difíciles de encontrar; algunos *SP* son difíciles de encontrar en la base de datos incluso con gestores modernos; y por último los programas de soporte de *SP* carecen de manejo de otros lenguajes de programación. Por ello en [21] presenta un modelo embebido en *GraalVM* con *Oracle Database* y *MySQL Database* que permita ejecutar *SP* y algunas funciones creadas por el usuario. Para ello se usa *JavaScript* como *SP* para resolver las desventajas que se mencionaron anteriormente. Si los datos del servidor y los datos del *SP* son del mismo tipo, se puede hacer el manejo de datos sin conversiones o copiado de datos, sin embargo, en casos especiales se podría hacer la conversión de datos, pero con un mínimo esfuerzo.

En la charla [21] se demostró que *JavaScript* es factible utilizarlo como *SP* en un gestor *Oracle Database* y en *MySQL Database*.

Por otro lado, en [22] se menciona acerca de los lenguajes soportados por *GraalVM*, lenguajes dinámicamente tipados como *JavaScript, Ruby, R, y Python*, y lenguajes estáticamente tipados como *Java, Kotlin, Scala* o aplicaciones compiladas con *LLVM*. La interoperabilidad de los lenguajes y la independencia de sus herramientas se consideran dos formas peculiares de un mecanismo interno de la meta programación de una máquina virtual.

En la interoperabilidad de los lenguajes de programación se menciona que dos o más de estos, por ejemplo, *JavaScript* y *R*, en *GraalVM* se comparten instancias de objetos entre estos con un buen alcance y un mínimo esfuerzo de ejecución. El lenguaje de programación

exporta cualquiera de sus objetos por medio de resolución de mensajes, cuando un objeto se comparte, el lenguaje envía un mensaje que describe la operación específica a ejecutar, si el mensaje es exitoso, se deriva la funcionalidad al lenguaje externo que ejecuta el fragmento de código para completar la operación.

En cuanto a la instrumentación, la segmentación de herramientas de *GraalVM* instrumentan aplicaciones en tiempo de ejecución al interceptar el núcleo de la ejecución, así como llamadas de funciones y líneas de ejecución, herramientas avanzadas definen la instrumentación de manera fina y específica usando varias *API* de instrumentación, que permite analizar expresiones de lenguajes y operaciones.

En [23] se describe cómo se realizó un análisis visual del desempeño de los lenguajes procesados por *Truffle*, basados en *Flamegraphs*, permitiendo tomarse el tiempo a las llamadas de pilas. Se usó la herramienta de *Linux perf* y el agente de la *JVM*, *perf-map-agent*, junto con el compilador *Graal JIT* que mapea la pila de llamadas a la *JVM* que maneja los diferentes lenguajes de programación. También se demuestra lo fácil y flexible que es el uso de herramientas modificadas con un bajo consumo de recursos, y la aplicación de técnicas que permitan entender el comportamiento de las aplicaciones políglotas.

La precisión de los perfiles recolectados es proporcional a los muestreos de frecuencia. El cambio de muestreo ocurre cuando una instrucción reportada de una muestra no representa de manera precisa la instrucción actual causando una interrupción en las muestras de las llamadas de pila. Los perfiles tradicionales generados desde la utilidad `prof` presentan como un programa usa mucho tiempo en cada función y el número de veces que dichas funciones son llamadas, también indica qué funciones consumen la mayoría de los ciclos. Las visualizaciones de las gráficas de flama en *SVG* (*Scalable Vector Graphic*, Gráfico Vectorial Escalable) se generan vía *JavaScript* que permiten su análisis por medio de un *browser*. Desafortunadamente perfilar el uso de un lenguaje externo resulta en llamadas

de pila que contienen métodos con el *framework* de *Truffle*.

En [23] se presentaron técnicas de baja sobrecarga usando herramientas estándar de *Linux* para perfilar *GraalVM* e implementaciones de *Truffle*. El enfoque no requiere modificación alguna a la implementación del lenguaje *Truffle*, a diferencia de los perfiles anteriores. Se tiene en cuenta que la implementación del *framework Truffle* en lenguajes como *TruffleRuby* y *FastR* han demostrado que son más rápidas que las implementaciones existentes, mientras que la de *JavaScript* con *Truffle* ha demostrado que está en una configuración de rendimiento razonablemente nivelada con la versión de *JavaScript V8* implementado por *Google*.

Šipek et al. [24] dieron a conocer que el *software* contemporáneo comúnmente se vuelve bastante complejo y es necesario el uso de múltiples tecnologías y diferentes lenguajes de programación para su desarrollo. Incluso, la interoperabilidad entre lenguajes de programación provoca una baja considerable en el rendimiento del *software*, por lo cual, es importante conocer cómo la programación políglota con un compilador orientado a lenguajes de alto nivel realiza estos procesamientos.

Uno de los proyectos que solucionan el problema mencionado y soporta una gran cantidad de lenguajes de programación, así como la interoperabilidad entre estos en la *JVM* es el proyecto de *Graal OpenJDK*, que evolucionó del proyecto *Maxine VM*. Este proyecto inició en el 2012 con la meta de crear un compilador avanzado para los lenguajes *POO* (Programación Orientada a Objetos) de alto nivel, en este caso Java. La *JVMCI (JVM Compiler Interface)* permite la implementación de una forma personalizada del compilador *JIT* escrito en *Java*, que permite el uso de la *JVM* como un compilador dinámico.

Con la versión actual de *GraalVM*, sin ninguna configuración adicional, en comparación con el estándar *JDK* (*Java Development Kit*, herramientas de desarrollo de *Java*), los resultados fueron mejores de los esperados, y ambas versiones de *Graal CE* (*Commu-*

nity Edition, versión comunitaria) y *Graal EE* (*Enterprise Edition*, versión empresarial), probaron ser mejores en pruebas diferentes. Esto comprueba que las siguientes versiones de *GraalVM* tendrán un mejor rendimiento, mejor optimización y mejores resultados.

Niephaus et al. [25] utilizaron ambientes *Jupyter*, estos ambientes se emplean por científicos que usan una gran cantidad de datos para publicar sus investigaciones en formatos ejecutables. Estos ambientes son comúnmente limitados porque usan solo un lenguaje de programación. Implementaron este enfoque usando *GraalSqueak*, una implementación de *Squeak/SmallTalk* para la máquina virtual de *Graal*. Para prototipar la experiencia de programación y experimentar con herramientas políglotas más avanzadas, construyeron un ambiente de *Squeak/SmallTalk*, que es compatible con el ambiente de *Jupyter*. Evaluaron *PolyJus*, por medio de una demostración de un ambiente políglota y discutiendo las ventajas y desventajas que se encontraron.

Desde que la comunidad científica usa una gran cantidad de lenguajes de programación, especialmente en el análisis de datos y *machine learning* (aprendizaje automático), seleccionan cualquier lenguaje de programación para su uso en los ambientes. Pero esta libertad es un poco limitada, desde que solo un lenguaje se usa por ambiente, es difícil para los científicos escribir programas en otro lenguaje al mismo tiempo. Para resolver este problema, es necesario un sistema que sea capaz de representar y ejecutar el código escrito en diferentes lenguajes, y lo más importante, hacer que haya interacción entre estos lenguajes y que sea lo más fácil posible de implementar.

Niephaus et al. hicieron uso del proyecto *Jupyter*, los ambientes *Jupyter* evolucionaron de *IPython*, una consola interactiva de *Python*; estos ambientes constan de celdas de código y texto, donde el usuario puede evaluar o ejecutar el código en un *kernel* que se ejecuta en un servidor. En un futuro planean crear más ambientes para analizar conjuntos de

información más grandes.

Salim et al. [26] trabajaron con *WebAssembly*, que es un compilador de formato binario para lenguajes como *C/C++, Rust* y *Go*, que habilita la ejecución en *browsers* y programas de tipo *Standalone* (programa de carácter único, sin dependencias). Los módulos compilados interactúan con otros lenguajes de programación, como *JavaScript*, y también hacen el uso de llamadas externas al ambiente del servidor. La implementación de un objeto *WebAssembly*, llamado *TruffleWasm*, provee un único ambiente para la ejecución de ambos, los módulos standalone, y los múltiples lenguajes que soporta *GraalVM* como *GraalJS (JavaScript)*, todo esto por medio del *framework* de interoperabilidad de *Truffle*. El compilador *Graal* hace uso de *JIT* para la generación de código.

La compilación *WebAssembly* usa la infraestructura *LLVM*, para producir binarios de *WebAssembly* sin usar una *API* en específico. El *framework* de *Truffle* se usó para proveerlos de su infraestructura y su interoperabilidad entre los lenguajes alojados en *GraalVM* y en la *TruffleNFI* (*Truffle Native Fundation Interface*), para permitir de manera eficiente el acceso a los intérpretes a funciones nativas. Se realizó la primera implementación de *WebAssembly* en *GraalVM* usando el *framework* de *Truffle* llamado *TruffleWasm*, el cual está diseñado para soportar módulos de *WebAsssembly* hacia diferentes ambientes como *WASI* (*WebAssembly System Interface*), interoperabilidad de *JavaScript* y programas *Standalone*.

Como trabajos futuros se propuso agregar configuraciones extras en *MVP* (*Minimal Viable Product*, producto viable mínimo), completar la *API* para ampliar el soporte en bases de código y programas, y mejorar la interoperabilidad con *JavaScript* y otros lenguajes de programación.

En [27], se presentaron los adaptadores que se usan en cuanto al envío de datos entre diferentes lenguajes usando *GraalVM* como herramienta de ejecución de código. Hoy en

día existe una gran cantidad de lenguajes de programación y aún hay más bibliotecas y frameworks disponibles para usarlos. La ejecución políglota se realiza por medio de *GraalVM*, la cual permite a los programadores ejecutar diferentes lenguajes de programación, incluso combinarlos. Sin embargo, también existen problemas en dichas ejecuciones, un problema común es el paso de datos entre lenguajes. *GraalVM* provee la interoperabilidad entre lenguajes de programación por medio de una *API* políglota y permite el paso de mensajes entre lenguajes de programación, sin embargo, en repetidas ocasiones, es necesario el paso de valores de tipo no primitivos entre lenguajes, haciendo que se complique la comunicación entre los lenguajes implicados.

En este caso, de manera inicial se usa la implantación del control *Truffle*, el cual es un *framework* de *GraalVM* para la implementación políglota. Los mensajes se representarán de forma interna como *TruffleObjects*, los cuales permitirán el paso de mensajes entre lenguajes de programación en tiempo de ejecución. Se propuso un patrón del adaptador en el contexto de la programación políglota para traducir mensajes enviados entre lenguajes de programación. Un adaptador políglota envuelve un objeto de otro lenguaje y se asegura la lectura de otros lenguajes satisfactoriamente, de esta manera, los tipos de datos no primitivos no se convierten o reestructuran a otro lenguaje antes de enviarse a un lenguaje externo. Se propuso este método de adaptador del cual incluso provee un mapeo estándar.

Para este adaptador se implementó un prototipo de los adaptadores políglotas usados en *Python* 3 y se demostró cómo trabajan en combinación con el *shell* de *GraalVM*. *Python* tiene una lista de objetos, que, traducido a *JavaScript*, automáticamente genera un mapeo en forma del arreglo clásico en *JavaScript*, y se responden a los mensajes con `push`, o `reduceRight`. Una solución alternativa es extender la funcionalidad del objeto *TruffleObject* con más conceptos de lenguajes de programación, pero, esta alternativa no anticiparía todos los casos de uso.

Como conclusión, se propuso el uso de adaptadores políglotas, que ayuda a los pro-

gramadores a enviar objetos no primitivos de un lenguaje de programación a otro. Los adaptadores políglotas permiten el envío de mensajes entre lenguajes y también permiten modificar el comportamiento de los mismos. En un futuro se espera que estos adaptadores se usen más a menudo y de manera automática, incluso su uso se implemente por medio de *frameworks*.

Salim et. al. [28] menciona que *WebAssembly* es un formato binario originalmente designado para despliegues *web* y ejecuciones combinadas con *JavaScript*. *WebAssembly* también provee un ambiente de ejecución para aplicaciones *standalone*. En [28] su trabajo está motivado para entender las ventajas y desventajas de usar *GraalVM*, su soporte de múltiples lenguajes de programación para ejecutar aplicaciones standalone de *WebAssembly*.

Recientes desarrollos con *WebAssembly* se orientaron a aplicaciones de tipo *standalone*, aplicaciones para *IoT*, embebidas, móviles, y servidores. El *framework* de Truffle permite hospedar intérpretes de lenguajes con un tiempo de ejecución competitivo para estar en una *JVM* con relativamente un bajo esfuerzo de desarrollo en comparación con implementaciones nativas. Al compilar el código fuente de un núcleo a *WebAssembly*, empaquetarlo en *I/O* y otras funciones de inicialización en *JavaScript* se usan las funciones estándar de *WASI API*. Importar y exportar son las funciones clave de la intermodulación e interoperabilidad de lenguajes, se espera que importar componentes sea proporcionado por el servidor de ejecución. Si *JavaScript* crea una estructura de datos de tipo `ArrayBuffer` que se envía a *WebAssembly* como importación, entonces WebAssembly codifica este objeto como un almacenamiento linear de memoria y cada cambio en este objeto es visible desde *JavaScript* y *WebAssembly*. Para ejecutar un módulo de *WebAssembly* en un ambiente diferente del que se implementó es necesario el código fuente para que sea recompilado con las configuraciones necesarias. Los módulos de *WebAssembly* se llaman desde *JavaScript* usando el *API WebAssembly.**, que tiene soporte por la mayoría de los *browsers*. *WebAssembly* llama a las funciones de *JavaScript* como funciones importadas, dichas llamadas

crean una barrera que puede influir en general cómo los métodos internos se miden o interpretan. Las dos metas de *WebAssembly* son la eficiencia y seguridad. Para proveer una ejecución standalone para *WebAssembly* las funciones importadas se implementan por el motor de ejecución, por ejemplo, para `wasi`, *TruffleWasm* provee implementaciones para las funciones que lo requieran. *TruffleWasm* usa nodos de *Truffle* para construir interpretes *AST* para nodos leídos por `Binaryen`, cada instrucción se convierte en un nodo de *Truffle* que realiza una ejecución específica. Cuando *GraalJS* instancia un módulo de *TruffleWasm*, los módulos importados se envían como un *proxy* de objetos hacia *TruffleWasm*, el objeto *proxy* se usa para llamadas a funciones de *JavaScript*.

En [28] se presentó *TruffleWasm*, la primera implementación de *WebAssembly* en una *JVM* que permite la ejecución de aplicaciones *WebAssembly* de tipo standalone que permite que sus módulos y funciones interactúen con *JavaScript*. *TruffleWasm* provee una plataforma para investigar elementos de *WebAssembly* como configuraciones del núcleo, comportamiento, y cómo proveer interoperabilidad con otros lenguajes hospedados por *Truffle*.

Riese et. al. [29] mencionan que, para mejorar la productividad de la programación, la selección de las herramientas correctas es crucial, esto inicia con la selección del lenguaje de programación y consecuentemente con las bibliotecas y *frameworks* disponibles en dicho lenguaje. Para reducir la complejidad mental de los desarrolladores y dejarlos enforcarse en la lógica del negocio, se introduce una interfaz de definición de usuario que mapea un adaptador entre lenguajes por medio de mensajes en tiempo de ejecución que permita enlazar con la interfaz esperada.

En el caso de *GraalVM*, el *framework* subyacente provee intercambios de tipos de datos primitivos y algunas estructuras de datos básicas como listas. Para construir el adaptador, la interfaz a adaptar es necesaria. Al momento de introducir una representación común de la interfaz a compartir, el esfuerzo de implementación se reduce, durante este proceso el

adaptador consta de dos componentes: la primera parte, que es la parte intermedia, mapea desde la interfaz destino hacia la representación intermedia; y la segunda parte, llamada parte adaptada, mapea desde la representación intermedia hacia la interfaz adaptada. Para crear la parte intermedia y crear el mapeo correspondiente, se debe conocer la interfaz destino. El mecanismo para determinar el lenguaje destino, tal y como el comportamiento del mapeo durante el tiempo de ejecución, introduce llamadas adicionales a funciones y el comportamiento de sobrecarga.

En [29], se introduce el mapeo de interfaz de definición de usuario en *GraalVM*. Esto permite que los desarrolladores de aplicaciones políglotas hagan combinaciones entre algoritmos y estructuras de datos desde diferentes lenguajes de programación, haciendo que estas sean efectivas y hacer el código más portable.

2.2. Análisis comparativo

La tabla 2.1 presenta una comparación de los contenidos de los artículos investigados, de manera concreta se visualizan el problema que se encuentra, la solución que propusieron, las tecnologías que usaron y el estado actual del trabajo que realizaron los investigadores.

Tabla 2.1: Análisis comparativo de artículos de investigación.

Artículo	Problema	Contribución	Tecnologías	Resultado	Estado
Stadler et al. [20].	Se carece de una forma de averiguar si ciertos componentes de un programa son accesibles desde un lenguaje de programación externo en un entorno políglota.	Se aplica la técnica *Escape Analysis* (Análisis de Escape) para averiguar el flujo y ciclo de vida de los elementos entre códigos políglotas.	• *GraalVM.* • *Graal IR.* • *Scalar Replacement.* • *Stack Allocation.* • *Lock Elision.*	Un alto número de *bytes* colocados también demuestran un algo número de espacios de memoria. Una significante reducción de la cantidad de bloqueos de memoria.	La iteración que actualiza el estatus de la localización podría hacerse de manera paralela al momento que se encuentre un salto en el control del flujo.

Continúa en la siguiente página

Tabla 2.1 Análisis comparativo de artículos de investigación.

Artículo	Problema	Contribución	Tecnologías	Resultado	Estado
Brantner et al. [21].	Los *SP* tienen grandes ventajas, pero también desventajas, que hacen que el desarrollo de un sistema en cuanto a su manejo de bases de datos sea complicado de desarrollar o darle mantenimiento en aplicaciones robustas.	Manejo de los *SP* por medio de *JavaScript* como lenguaje de ejecución dentro de bases de datos.	• *JavaScript.* • *SQL.* • *Oracle Database.* • *MySQL Database.* • *GraalVM.*	Se demuestra cómo *JavaScript* se usa en los *SP* dentro de *Oracle Database* y *MySQL Database.*	Concluido.
Bonetta et al. [22].	Demostración de interoperabilidad y comunicación entre mensajes dinámicamente tipados y estáticamente tipados que se alojan en *GraalVM.*	El manejo de mensajes entre lenguajes de programación para envío de objetos entre estos.	• *GraalVM.* • *JavaScript.* • *Ruby.* • *R.* • *Python.* • *Java.* • *Kotlin.* • *Scala.* • *LLVM.*	Por medio de distintas *API* de instrumentación se permiten interceptar llamadas y líneas de ejecución.	Concluido.

Continúa en la siguiente página

Tabla 2.1 Análisis comparativo de artículos de investigación.

Artículo	Problema	Contribución	Tecnologías	Resultado	Estado
Gaikwad et al. [23].	La cantidad excesiva de llamadas a funciones, funciones en ciclos y el consumo de recursos al ejecutar dichas funciones.	Análisis de las llamadas a funciones por medio de *Truffle, perf-map-agent, prof* y sus representaciones de consumo de recursos por medio de *Flamegraphs* en formato *SVG* con *JavaScript*.	• *Truffle.* • *Flamegraphs.* • *perf-map-agent.* • *SVG.* • *JavaScript.* • *JVM.* • *Graal JIT.* • *Prof.*	Se tiene en cuenta que la implementación del *framework Truffle* en lenguajes como *TruffleRuby* y *FastR* han demostrado que son más rápidas que las implementaciones nativas existentes.	Concluido.
Šipek et al. [24].	El *software* contemporáneo y moderno requiere de mejores y nuevas tecnologías para desarrollarse, así también como el manejo de programación políglota.	Una herramienta para el manejo de programación políglota, que tiene como base *JVM* y *GraalVM*.	• *Maxine VM.* • *Graal OpenJDK.* • *JVMCI.* • *JIT.* • *Graal CE.* • *Graal EE.*	Se comprobó que con *GraalVM* se optimiza el desarrollo de programas por medio de la programación políglota y también se probó con *Graal CE* y *Graal EE*.	Concluido.

Continúa en la siguiente página

Tabla 2.1 Análisis comparativo de artículos de investigación.

Artículo	Problema	Contribución	Tecnologías	Resultado	Estado
Niephaus et al. [25]	La falta de un sistema que permita la ejecución de programas o códigos políglotas en ambientes *Jupyter*.	Se realiza implementación de *PolyJus*, un ambiente que permite ejecutar códigos políglotas usando el mismo *kernel* y *API* entre lenguajes de programación.	• *Jupyter.* • *GraalSqueak.* • *Squeak/Smalltalk.* • *PolyJus.* • *IPython.* • *Python.*	Se obtuvo el procesamiento de grandes cantidades de datos usando *PolyJus* con códigos políglotas.	Concluido y buscando mejoras a futuro.
Salim et al. [26].	Aplicar la programación políglota por medio de herramientas basadas en *C/C++* para ejecutar programas en *browsers* con *JavaScript*.	El manejo de *WebAssembly* como un compilador para generar los binarios por medio de *TruffleWasm* y ejecutarlos en *browsers*.	• *WebAssembly.* • *Rust.* • *Go.* • *TruffleWasm.* • *GraalJS.* • *Graal.* • *TruffleNFI.*	Se obtuvo el binario por medio de *WebAssembly* usando *TruffleWasm* para ejecutar código desde un *browser* con *JavaScript* como lenguaje base.	Concluido y buscando mejoras a futuro.
Niephaus et al. [27].	Falta de un mecanismo que permita el envío de mensajes o datos de tipo no primitivos entre lenguajes de programación dentro de la programación políglota.	Un prototipo de adaptador políglota para el envío de mensajes entre lenguajes.	• *GraalVM.* • *Python 3.* • *TruffleObject.*	Se logra el envío de mensajes entre diferentes lenguajes de programación usando adaptadores políglotas.	Concluido y buscando mejoras a futuro.

Continúa en la siguiente página

Tabla 2.1 Análisis comparativo de artículos de investigación.

Artículo	Problema	Contribución	Tecnologías	Resultado	Estado
Salim et al. [28]	El desarrollo de aplicaciones *Standalone* con entornos de *WebAssembly* tiene dificultades en la comunicación con lenguajes externos en entornos políglotas en cuanto al envío de mensajes, y manejo de objetos entre lenguajes.	Se implementa *TruffleWasm*, un *framework* de *Truffle* para programas de tipo *WebAssembly* que permite la comunicación con otros lenguajes de programación en entornos web como *JavaScript*.	• *WebAssembly.* • *Truffle.* • *Wasi API.* • *GraalJS.* • *AST.* • *TruffleWasm.*	Se logra la comunicación con *JavaScript* en un entorno *web* con una aplicación *standalone* de *WebAssembly* por medio de una interfaz *proxy* entre objetos.	Concluido.

Continúa en la siguiente página

Tabla 2.1 Análisis comparativo de artículos de investigación.

Artículo	Problema	Contribución	Tecnologías	Resultado	Estado
Riese et. al. [29]	Para mejorar la productividad de la programación, la selección de las herramientas correctas es crucial, esto inicia con la selección del lenguaje de programación y consecuentemente con las bibliotecas y *frameworks* disponibles en dicho lenguaje, esto hace que se genere una complejidad mental en los programadores al momento del desarrollo de los sistemas.	Se crea un mapeo entre una interfaz origen, una sección intermedia y una interfaz destino que permite la comunicación entre lenguajes de programación en entornos políglotas.	• *GraalVM.* • *JHVM.* • *Truffle.* • *Python.* • *Ruby.* • *Squeak/Smalltalk.*	Se introduce el mapeo de interfaces. Esto permite que los desarrolladores de aplicaciones políglotas hagan combinaciones entre algoritmos y estructuras de datos desde diferentes lenguajes de programación, haciendo que estas sean efectivas y hacer el código más portable.	Concluido.

Hoy en día, el desarrollo de aplicaciones es cada vez más complejo, debido a que éstas son más robustas, complejas y requieren una gran cantidad de procesos, transacciones y funciones, esto para cumplir con la demanda de los usuarios ya sea en ambientes comerciales, públicos, académicos o científicos. Por ello, es necesario el uso de la herramienta *GraalVM*, que permite el desarrollo de aplicaciones en entornos políglotas, también permite que estas aplicaciones sean mas rápidas, eficientes, seguras y de fácil mantenimiento. Sin embargo, como se observó en los artículos anteriormente analizados, *GraalVM* aún tiene características que le faltan mejorar y cumplir con los requerimientos de los desarrolladores de software, por ello, se mantiene en constante evolución y mejora.

2.3. Solución propuesta

En esta sección se menciona la solución propuesta con base en características esenciales de las herramientas para el desarrollo óptimo, eficaz y eficiente.

2.3.1. Justificación de la solución seleccionada

A continuación, se presenta la justificación de la solución propuesta por ambiente, editor de texto, gestor de base de datos y metodología de desarrollo ágil.

- **Ambiente.** Se selecciona el ambiente *GraalVM*, ya que es la única herramienta existente para el manejo de programación políglota, aunque el ambiente *Web* maneja *HTML JavaScript* y *CSS* [30] en un mismo ambiente, carece del manejo de lenguajes de programación que no son comunes en este ambiente como lo es *Java, C++*, entre otros, incluso el manejo de lenguajes de consulta como lo es *SQL*.

- **Editor de texto.** Se selecciona a *Visual Studio Code* porque es una herramienta multiplataforma, tiene soporte de distintos lenguajes de programación, consta de diferentes extensiones para manejo de complementos necesarios en el desarrollo; cuenta con manejo de directorios, es decir, permite la administración de archivos de código

por medio de carpetas y tener una mejor organización del código. Cuenta con una terminal que permite ejecutar el código en tiempo real y mostrar un resultado, en el caso de *Java*, permite el compilado y ejecución de los programas. Por último, a pesar de que al igual que *Notepad++*, no cuenta con un plugin o extensión de detección de código políglota, sus extensiones permiten diferenciar palabras reservadas del lenguaje de programación, declaración de variables y sintaxis propias del lenguaje, esto es de gran ayuda en la identificación del código diferente al que *Visual Studio Code* reconoce como lenguaje base, en pocas palabras, permite la identificación del código políglota.

- **Gestor de base de datos.** *Oracle Database* es la mejor opción ya que tiene una integración directa con *GraalVM* permitiendo la comunicación natural entre estos sin necesidad del uso de *software* de terceros o bibliotecas externas. También por el gran potencial de manejo de datos, consultas, procedimientos almacenados, entre otras funciones que tiene, que están mejor implementadas que en *MySQL*.

- **Metodología de desarrollo ágil.** Se selecciona a *Scrum*, esto porque tiene una gran tolerancia a los cambios durante el desarrollo del producto de *software*, de cierta forma fue una de sus principales motivaciones de su creación, al cambio constante de los requerimientos de *software*. También por su administración ágil de tareas y actividades manejando *sprints* de entrega, dándole al cliente un panorama claro de cómo el producto final quedaría y cómo funcionaría en su totalidad.

Capítulo 3

Aplicación de la metodología

En este capítulo se presenta la aplicación de la metodología para encontrar la solución a la problemática establecida en el punto 1.3.

3.1. Experimentación de las capacidades nativas con *GraalVM*

En esta sección se presentarán una serie de casos de prueba donde se realizaron experimentos con código políglota para observar las capacidades de nativas con *GraalVM*. A continuación, se presenta el desarrollo de los casos de prueba.

3.1.1. Selección del escenario con capacidades nativas

Creación, lectura y escritura de un archivo de texto. En este escenario, se tiene la creación, escritura y lectura de un archivo de texto con base en una cadena de texto introducida por el usuario; dicho proceso se lleva a cabo usando diferentes lenguajes de programación en un mismo programa, y para cada proceso se usa un lenguaje diferente que ayude al proceso del programa (una implementación de múltiples lenguajes como lo realizó Grimmer et. al en [31] y Pool, Tõnis et al. en [32] con *TruffleReloader*), dichos lenguajes se mencionan a continuación:

1. *Java*
2. *Ruby*
3. *Python*
4. *C*

3.1.2. Análisis y diseño del escenario con capacidades nativas

En esta sección se presentarán los análisis y diseños de la implementacion del escenario seleccionado para probar las capacidades nativas y de interoperabilidad de *GraalVM*. Como primer punto se presenta el análisis de la solución del escenario propuesto anteriormente.

Análisis

Se realiza una separación de procesos que llevará a cabo la solución del escenario propuesto, también se especifica cuál será el lenguaje apropiado para dicho proceso, cada proceso y su explicación se mencionaran a continuación:

1. **Lenguaje base del programa (*Java*).** El programa tiene a Java como lenguaje *host* (se le llama así al lenguaje de programación en el cual se genera el archivo base de código fuente). En este lenguaje se realizó el proceso de obtención de la cadena de texto introducida por el usuario. Se seleccionó este lenguaje para dichas funciones ya que las capacidades nativas de *GraalVM* se realizan con programas hechos en *Java*.
2. **Creación del archivo (*Ruby*).** El archivo se creó con lenguaje *Ruby* [33] como lenguaje *guest* (este lenguaje pertenece al código incrustado en el lenguaje *host* en un mismo entorno generando asi código políglota), y el contenido de dicho archivo es la cadena introducida por el usuario al ejecutar el programa. Se seleccionó este lenguaje para la creación del archivo debido a su corta cantidad de líneas para ejecutar este tipo de tareas.

3. **Lectura del archivo (*Python*).** La lectura del archivo creado se realiza con lenguaje *Python* como lenguaje *guest*. Se seleccionó este lenguaje para dicha función debido a que *Python* trabaja muy bien con la lectura de archivos de grandes tamaños en el procesamiento de datos.

4. **Impresion en pantalla del contenido del archivo (*C*).** La impresión en pantalla se lleva a cabo en lenguaje *C* como lenguaje *guest*. Se seleccionó *C* en esta función para demostrar de manera sencilla la interoperabilidad y comunicación de lenguajes interpretados con lenguajes compilados al enviar una cadena de texto de *Python* a *C* a través de *Java* y mostrar el contenido en pantalla.

El proceso anteriormente mencionado consta de las actividades para el proceso interno del programa en *Java*, sin embargo, aún no tiene características nativas, lo siguiente sería crear la imagen nativa del programa agregando unas directivas necesarias en el programa para el reconocimiento de bibliotecas de lenguajes de *GraalVM* y por último ejecutar la imagen nativa.

Diseño

A continuación, se presentan los diagramas realizados para representar el diseño de la solución al escenario seleccionado.

En la Figura 3.1 se muestra el diagrama de flujo del programa en *Java* que realizará la creación, escritura y lectura del archivo de texto. Como primer bloque se tiene la obtención de la cadena de texto introducida por el usuario utilizando los argumentos del metodo `Main` en *Java*; el siguiente bloque es la obtención de dicha cadena y la almacena en una variable en *Java*; la siguiente actividad del diagrama presenta la creación del archivo de texto en lenguaje *Ruby*, almacenando la cadena introducida por el usuario en dicho archivo; después se tiene la lectura del archivo de texto en lenguaje *Python*; por último, se tiene la impresión en pantalla del contenido del archivo de texto en lenguaje *C*.

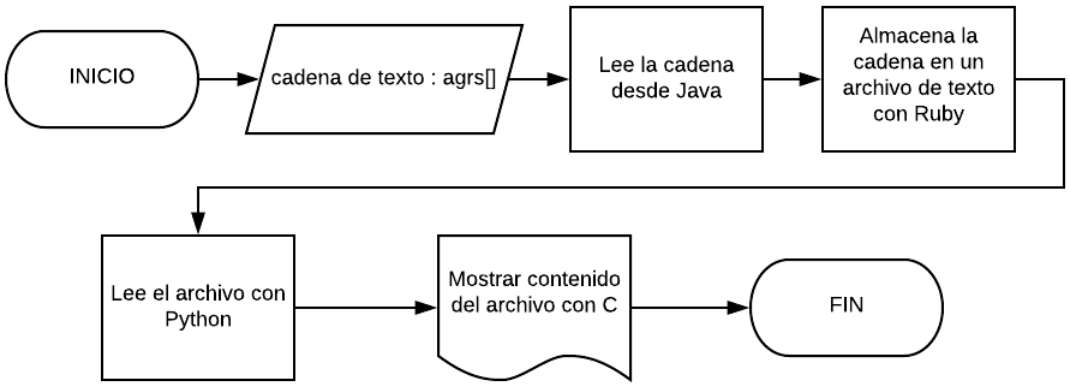

Figura 3.1: Diagrama de flujo del escenario de creación, lectura y escritura de un archivo de texto

La conversión a imagen nativa se llevará a cabo una vez que el programa funcione con éxito con este proceso, ya que su conversión no afecta la lógica del programa, solo se agregan unas directivas internas para el reconomiento de componentes de *GraalVM*, este agregado de directivas es al inicio del programa y no afecta el código de la solución.

3.1.3. Implementación y pruebas del escenario con capacidades nativas

A continuación, se presenta la implementación y pruebas del escenario propuesto, la implementación comprende los códigos escritos y las pruebas comprenden imágenes de las ejecuciones realizadas. Dicha implementación y pruebas se presentarán en dos partes: la primera será el programa con lenguaje base *Java* llamado `Poliglota.java` y sus respectivas pruebas; y la segunda parte será el mismo programa `Poliglota.java`, pero con unas adecuaciones para que se cree como una imagen nativa llamado `PoliglotaNI.java`.

Implementación

Se presenta el código realizado del programa que soluciona el escenario propuesto. Debido a que es un programa extenso, se presentará por partes los bloques de código y se explicarán en qué consiste cada bloque. En el Código 3.1 se observa en la línea 2 que

se hace la importación de la biblioteca `org.graalvm.polyglot` de *GraalVM*, después se tiene la creación de la clase y el método `main`. En el Código 3.2 se presenta la declaración de variables que se usarán en el programa.

Código 3.1: Bibliotecas de *GraalVM* y encabezados

```
1 import java.io.*;
2 import org.graalvm.polyglot.*;
3
4 public class Poliglota{
5     public static void main(String[] args) {
```

Código 3.2: Declaración de variables

```
6   String texto = "";
7   String codigoRuby = "";
8   String codigoPython = "";
9   String codigoCpp = "";
10  String filename= "cadena.txt";
```

En el Código 3.3 en la línea 12 se observa la creación del objeto `context`, este objeto permite la creación del ambiente políglota de *GraalVM* donde se ejecutan los diferentes lenguajes de programación. En la línea 14 se especifican los permisos generales a *GraalVM*, y el permiso a la creación, lectura y escritura de archivos dentro del equipo de cómputo.

Código 3.3: Declaración de variable `context`

```
12  try(Context context = Context
13    .newBuilder()
14    .allowAllAccess(true)
15    .allowIO(true)
16    .build()){
```

En el Código 3.4 se genera el lanzamiento de una excepción en caso de que no se introduzca una cadena de texto al ejecutar el programa.

Código 3.4: Excepcion de cadena no introducida

```
17  if(args.length == 0)
18    throw new Exception("No se introdujo una cadena, favor de ingresar
     una cadena de texto");
```

En el Código 3.5 se realiza la obtención de la cadena de texto que se introdujo desde una terminal y se almacena en una variable de *Java*, la concatenación de la línea 24 es

solamente para mostrar el nombre del lenguaje que lo está ejecutando, solo para cuestiones de flujo de código y ver cómo la cadena de texto va pasando por los diferentes lenguajes de programación.

Código 3.5: Almacenamiento de la cadena de texto en una variable de *Java*.

```
20 //Se agrega en una linea el texto que se introdujo desde la terminal
21   for(String arg : args){
22       texto += arg + " ";
23     }
24     texto += "- JAVA";
```

En el Código 3.6 se observa la creación del archivo en lenguaje *Ruby*, se genera una cadena de texto con el script a ejecutar de *Ruby* y se almacena en una variable en *Java*, después en la línea 30 se ejecuta con el objeto `context` usando la función `eval()`, se especifica primero el lenguaje de programación y después el *script* o líneas de código a ejecutar.

Código 3.6: Creación del archivo en Ruby.

```
26   // Creacion del archivo con Ruby
27   codigoRuby += "out_file = File.new('" + filename + "', 'w+') \n";
28   codigoRuby += "out_file.puts('"+ texto + " - RUBY') \n";
29   codigoRuby += "out_file.close \n";
30   context.eval("ruby", codigoRuby);
```

En el Código 3.7 se realiza la ejecución de la lectura del archivo, este bloque es muy parecido al Código 3.6 ya que se concatena una cadena de texto con el código a ejecutar, en este caso es en lenguaje *Python*, por último, en la línea 38 se realiza la ejecución de dicho código con el objeto `context`.

Código 3.7: Lectura del archivo en Python.

```
32   // Lectura del archivo con Python
33   codigoPython += "cadena = '' \n";
34   codigoPython += "for line in open('" + filename + "'): \n";
35   codigoPython += "        cadena = cadena + line ";
36   codigoPython += "\n\n ";
37   codigoPython += "cadena.strip()+' - PYTHON'";
38   Value filecontent = context.eval("python", codigoPython);
39   String textodesdePython = filecontent.asString();
```

En el Código 3.8 se realiza la impresión en pantalla de la cadena de texto introducida

desde la terminal junto con los nombres de los lenguajes en los cuales se ejecutó. *GraalVM* también compila el lenguaje *C* [34], por lo tanto, se tienen acceso a este como un programa objeto y se ejecuta desde un objeto `context`, seguido se almacena en un objeto de tipo `Value` como se observa en la línea 43, con esto se tiene acceso a las funciones internas del programa generadas por el desarrollador, incluso se tiene acceso a funciones propias del lenguaje *C*, en la línea 44 se ejecuta `printMensajeArray`, que es una función que se desarrolló en *C* que imprime en pantalla una cadena de texto que se obtiene como parámetro de función. La línea 46 hace la ejecución final de la impresión en pantalla desde *C*.

Código 3.8: Impresión en pantalla desde *C*.

```
41 //Impresion de contenido en C
42 Source s =Source.newBuilder("llvm",new File("imprimeTexto.o")).build();
43 Value lib = context.eval(s);
44 Value printMensajeArray = lib.getMember("printMensajeArray");
45 Value msgarray = getvaluyeArrayCharASCII(textodesdePython,context);
46 printMensajeArray.executeVoid(msgarray); //Imprime en pantalla desde C
```

En el Código 3.9 se tiene al método `getvaluyeArrayCharASCII` que se ejecuta en la linea 45 del Código 3.8, dicho método hace la conversión de la cadena de texto que se obtuvo desde *Python* a una cadena compatible para enviarse al entorno *C* a través del objeto `context` y así imprimir en pantalla dicha cadena en *C*.

Código 3.9: Método para conversión de cadena.

```
60 public static Value getvaluyeArrayCharASCII(String text, Context ctx){
61   char[] arraychar = text.toCharArray();
62   int length = arraychar.length;
63   int[] arraychars = new int[length];
64   Value value = ctx.asValue(arraychars);
65
66   for(int i=0;i<length;i++){
67     int castAscii = (int) arraychar[i];
68     value.setArrayElement(i, castAscii);
69   }
70   return value;
71 }
```

El Código 3.10 es el que realiza la impresión en pantalla desde *C*, obteniendo como parámetro `msgarray` que contiene la cadena que obtuvo desde *Python*. Dicho código se

ejecuta con el objeto `context` desde un objeto compilado con *GraalVM*.

Código 3.10: Función para impresión en pantalla desde *C*.

```c
#include <stdio.h>
#include <string.h>

void printMensajeArray(char msgarray[]) {
    printf("%s \n","IMPRESION DESDE C");
    for(int i=0; i<strlen(msgarray); i++)
    printf("%c",msgarray[i]);

    printf("%s" , " \n");
}
```

El compilado del programa `imprimeTexto.c` que se tiene en el Código 3.10 se realiza desde una terminal con la siguiente línea:

$LLVM_TOOLCHAIN/clang -g -O1 -c -emit-llvm imprimeTexto.c

Pruebas

En esta sección se presentarán las pruebas realizadas relacionadas a la primera parte de la implementación del esceneario con capacidades nativas, se presentarán una serie de imágenes para visualizar los resultados obtenidos.

En la Figura 3.2 se presenta la compilación y ejecución del programa desde una terminal, al final presenta en pantalla la cadena introducida por el usuario junto con los lenguajes por los que pasó a lo largo de la ejecución de programa con lenguaje políglota.

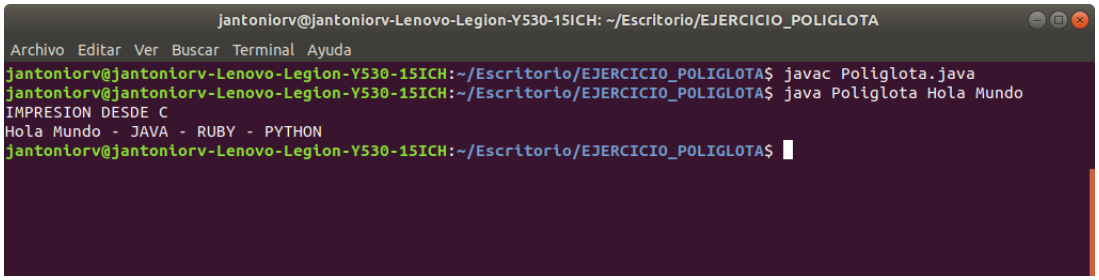

Figura 3.2: Ejecución del programa desde una terminal.

Implementación nativa

En esta sección se presentan las modificaciones al programa `Poliglota.java` visto en la primera parte, estas modificaciones seran para que pueda ejecutarse la imagen nativa de manera políglota.

En la Figura 3.11 se observa lo que es la declaración de las propiedades `option` en el objeto `context`, estas opciones se declaran para permitir a la imagen nativa localizar las ubicaciones de los lenguajes a usar en la imagen nativa, por ello se observan palabras como *"ruby"* o *"python"* en las líneas 18 a 23, que son los lenguajes de programación que se usaron en dicha prueba.

Código 3.11: Declaración de valores "option" en context.

```
12 try(Context context = Context
13 .newBuilder()
14 .allowNativeAccess(true)
15 .allowHostAccess(HostAccess.ALL)
16 .allowAllAccess(true)
17 .allowIO(true)
18 .option("ruby.home","<GRAALPATH>/jre/languages/ruby")
19 .option("python.SysPrefix","<GRAALPATH>/jre/languages/python")
20 .option("python.CoreHome","<GRAALPATH>/jre/languages/python/lib-
      graalpython")
21 .option("python.StdLibHome","<GRAALPATH>/jre/languages/python/lib-python
      /3")
22 .option("python.Executable","<GRAALPATH>/jre/languages/python/bin/
      graalpython")
23 .option("python.CAPI","<GRAALPATH>/jre/languages/python/lib-graalpython"
      )
24 .build()){
```

Debido a que es necesaria una serie de parámetros propios de *GraalVM* al momento de ejecutar la imagen nativa por medio de una terminal, se optó por colocar la cadena de texto en una variable dentro del programa tal y como se observa en la Figura 3.12

Código 3.12: Cadena de texto a almacenar en archivo.

```
26   texto = "Hola mundo desde Imagen Nativa - JAVA";
```

El contenido del programa en general sería el mismo al programa de `Poliglota.java`, sólo se cambiaron los bloques anteriormente explicados de los Códigos 3.11 y 3.12, pa-

ra función de versiones se creó un nuevo programa llamado `PoliglotaNI.java` con los cambios anteriormente realizados.

Pruebas

Se presentan las pruebas realizadas en el programa `PoliglotaNI.java` que comprende desde el compilado del programa, su creación como imagen nativa y su ejecución tal y como se observa en la Figura 3.3.

Figura 3.3: Ejecución del programa nativo desde una terminal.

Se realiza el compilado del programa `PoliglotaNI.java` con el compilador `javac`, después se procede a generar la imagen nativa de dicho programa desde una terminal con la siguiente línea:

native-image –language:ruby –language:python –language:llvm PoliglotaNI

Con `native-image` se inicia el comando para generar la imagen nativa; y con el comando `--language` se especifican los lenguajes que interactuan dentro de la imagen nativa, en este caso son *Ruby, Python,* y *C*; por último, se coloca el nombre de la clase a crear como imagen nativa, en este caso `PoliglotaNI`, dicho proceso se creación de la imagen nativa tarda alrededor de 5 minutos.

Después se tiene el comando de ejecución de la imagen nativa, que es:

./poliglotani -Dllvm.home=<GRAALPATH>/jre/languages/llvm

Al ejecutar la imagen nativa , ésta se ejecuta como un objeto bash de *Linux*, en este ejemplo se pasa como parámetro de ejecución el valor `-Dllvm.home`, este parámetro es necesario ya que en el programa de la imagen nativa se ejecuta código en lenguaje *C* usando *LLVM*, este valor `option` se envía de esta manera debido a la versión de *GraalVM* que se está manejando, la versión 20.1.

Ahora se procede a realizar una comparación de desempeño del programa `Poliglota.java` contra el programa `PoliglotaNI.java`, dicha comparación se realiza en cuanto al tiempo que se toman en ejecutarse.

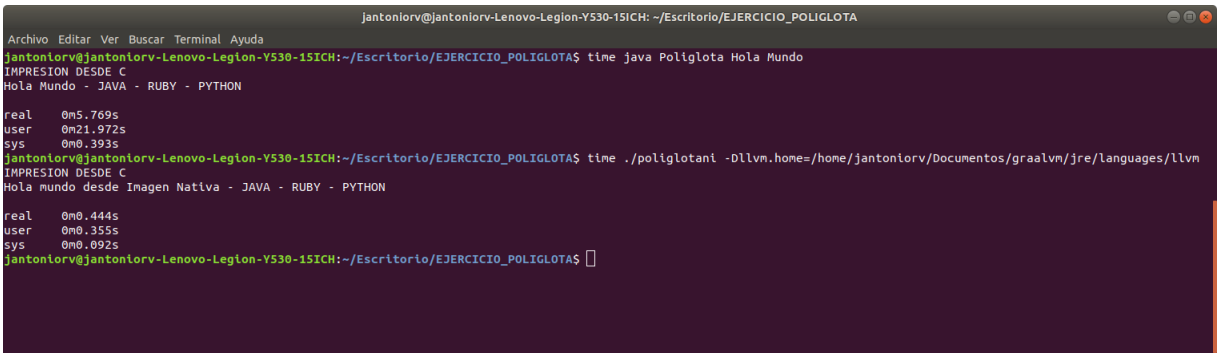

Figura 3.4: Comparación de desempeño de programas.

Como se observa en la Figura 3.4 el programa `Poliglota.java` tarda 5.769 segundos en ejecutarse, y la imagen nativa del programa `PoliglotaNI.java` tarda 0.444 segundos, por lo cual se demuestra la superioridad en cuanto al tiempo de ejecución de la imagen nativa con *GraalVM* sobre la ejecucion del programa desde el *JVM Hotspot*.

Para efectos de rendimiento se hizo un conjunto de pruebas de desempeño de programas, se realizó la ejecución del programa de creación y lectura de archivos explicados anteriormente, con y sin imagen nativa; ambas versiones se ejecutaron un total de 100 veces en un equipo *Lenovo Legion Y530*, con procesador *Core i5 8300H* a 60 *Hz* y 16 *GB* de *RAM* en *Ubuntu* 18.04 64 *bits*. Los resultados se describen a continuación.

Como se observa en la Tabla 3.1, los resultados oscilan de 5 a 7 incluso 9 segundos,

Tabla 3.1: Resultados del tiempo de ejecución del programa sin imagen nativa.

5.699	5.563	5.791	5.565	5.899	5.681	5.666	5.766	5.616	5.573
5.897	5.700	5.593	5.770	5.717	5.843	5.770	5.762	5.767	5.802
5.766	5.876	5.761	5.554	5.971	5.670	5.943	5.834	5.671	5.752
5.749	5.962	5.605	5.651	5.607	7.302	5.734	5.774	5.870	5.733
9.375	5.589	5.855	5.790	5.641	5.624	6.064	5.845	5.967	5.709
5.762	5.623	5.830	5.811	5.697	5.902	5.735	5.745	6.376	5.880
7.522	5.743	5.665	5.722	5.879	5.616	5.872	5.607	6.665	5.631
7.029	5.634	5.675	5.753	5.771	5.940	5.745	5.966	5.952	5.715
5.803	5.649	5.748	5.676	5.773	5.745	5.683	5.684	5.696	5.574
5.674	5.671	5.977	5.618	5.581	5.770	5.560	5.704	5.726	5.585

Tabla 3.2: Resultados del tiempo de ejecución del programa con imagen nativa.

1.529	0.428	0.427	0.425	0.427	0.427	0.435	0.426	0.433	0.433
0.433	0.426	0.437	0.427	0.429	0.428	0.425	0.431	0.427	0.426
0.427	0.426	0.431	0.426	0.426	0.432	0.428	0.433	0.429	0.438
0.432	0.432	0.433	0.425	0.427	0.425	0.427	0.430	0.426	0.432
0.430	0.429	0.429	0.429	0.426	0.438	0.427	0.427	0.430	0.434
0.426	0.437	0.427	0.426	0.433	0.429	0.438	0.427	0.430	0.428
0.426	0.424	0.432	0.430	0.432	0.425	0.429	0.432	0.427	0.433
0.435	0.426	0.432	0.426	0.426	0.435	0.427	0.425	0.426	0.425
0.426	0.426	0.425	0.426	0.426	0.431	0.430	0.432	0.428	0.430
0.428	0.425	0.433	0.425	0.431	0.433	0.427	0.426	0.426	0.426

dando un promedio de 5.840 segundos de tiempo de ejecución. En la Tabla 3.2 se observa lo que son los resultados en segundos del tiempo de ejecución del programa, pero desde una imagen nativa, por lo que se observa un tiempo que oscila de los 0.424 segundos a los 1.529 segundos, dando un promedio de todos los datos de 0.43997 segundos de tiempo de ejecución, por lo que se aprecia el gran potencial que tiene la imagen nativa sobre el programa de *Java*.

3.2. Experimentación de las capacidades de interoperabilidad de lenguajes con *GraalVM*

A continuación, se presenta el desarrollo del escenario que se probó para demostrar las capacidades de interoperabilidad de lenguajes con *GraalVM*. Se desarrolló el escenario con lenguaje *Java*, usando *Node.js* como lenguaje *host*. Se seleccionó *Node.js* como lenguaje *host* ya que es un *framework* que permite la ejecución de programas desde diferentes entornos ya sea *web* o desde una terminal, y debido a que el escenario de estudio es sobre una aplicación *web*, se considera propio el uso del entorno *web* de *Node.js*.

3.2.1. Selección del escenario de interoperabilidad de lenguajes

Como escenario se tomó un caso de estudio que consiste en una aplicación *web* de una empresa de desarrollo de software en la ciudad de Monterrey, Nuevo León, México, debido a derechos de autor y temas de confidencialidad, solo se mostrarán diagramas representativos del sistema estudiado de la compañía.

En dicho sistema se tiene la separación de lenguajes de programación debido a que un lenguaje es para desarrollo propio del entorno web, como lo es *HTML, CSS, JavaScript*; los lenguajes de *scripting*, que en este caso es *ASP.NET* y *C#* como lenguaje de programación; y el lenguaje *SQL*. Cada uno de estos ambientes se encuentran en entornos separados, aplicando procesos de la ingeniería de software como lo es la separación de intereses.

3.2.2. Análisis y diseño del escenario de interoperabilidad de lenguajes

En esta sección se realiza el análisis y diseño de los elementos que conforman el escenario de interoperabilidad entre lenguajes de programación con *GraalVM*.

Como se observa en la Figura 3.5, se aplica la arquitectura *SOA*, esto permite que

la aplicación se consulte desde canales *web* y dispositivos móviles. En dicho diagrama se observa la separación de intereses por medio de una capa de "Lógica de negocio" que realiza todas las operaciones y algoritmos referentes a las funciones de la compañía; en la capa de "Acceso a datos" es propiamente para el manejo de consultas a bases de datos para su posterior procesamiento en la capa de "Lógica de negocio"; la capa de "Cliente sistemas externos" se usa para la obtención de datos desde un sistema externo que interactúa con el mismo sistema en cuestión; se tiene como agentes externos lo que es la "base de datos" y el "sistema externo" que interactúan con los servicios por medio de algunas *API* o *software* de terceros; por último, se implementa todo esto dentro de la solución de servicios *SOAP/REST*.

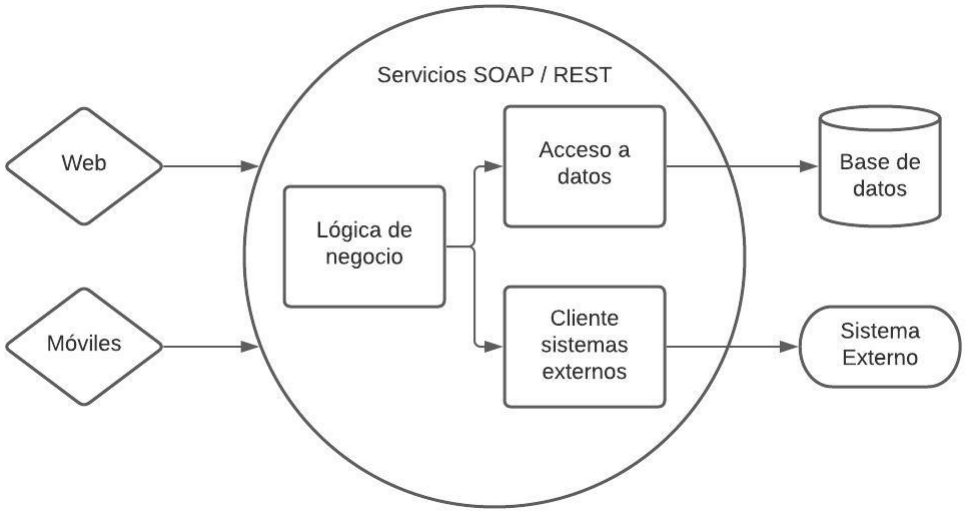

Figura 3.5: Diagrama de Arquitectura *SOA* en sistema del escenario seleccionado.

Tal y como se observa en la Figura 3.6, por medio del canal *web*, se maneja el modelo *MVC* junto con el uso de una *API* pare el consumo de servicios explicado en la Figura 3.5. Se tiene la vista que se presenta desde un *browser* al usuario, el controlador que realiza la administración de llamadas y obtención de datos desde diferentes acciones del usuario y el modelo que lleva a cabo el procesamiento de datos y obtención de los mismos desde el uso de una *API* de servicios.

Como se observa, a pesar de tener una separación bien estructurada, cada una se usa

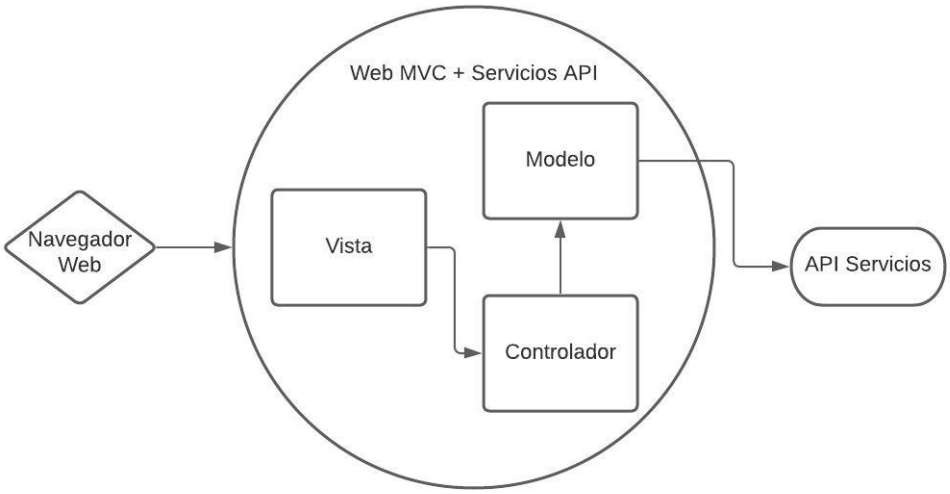

Figura 3.6: Diagrama de aplicación del modelo *MVC* con consumo de una *API* de servicios.

para un asunto en específico, el mantenimiento, así como la documentación e implementación de la misma hacen que sea un trabajo difícil e incluso laborioso, ya que se tienen distintos proyectos a modificar en caso de alguna falla o agregado de una funcionalidad nueva, haciendo evidente la necesidad del manejo de programación políglota en desarrollos tan complejos como se muestra en este ejemplo.

Continuando con el caso de estudio, se realizó un rediseño e implementación de cómo sería su funcionamiento si este se aplicara por medio de un entorno políglota con *GraalVM*. En la Figura 3.7 se tiene el diagrama de flujo del funcionamiento del programa políglota: se introduce el identificador del cliente `IdCliente`, con este valor se busca en la base de datos las órdenes o pedidos que ha realizado el cliente, después se consulta desde un sistema externo por medio de una clase en *Java*, se obtienen los datos de órdenes de elaboración del cliente, sin hacer uso de alguna *API* o *software* de terceros, después se realiza un mapeo de dichas órdenes para ver si hay una intersección de ambos conjuntos de elementos donde el conjunto A serían las órdenes en base de datos y el conjunto B serían las órdenes de elaboración desde el sistema externo obteniendo $C = A \cap B$ donde C es el conjunto de órdenes en proceso de elaboración pertenecientes al cliente. Después solo se analiza si el conjunto C tiene elementos, si no, presenta en pantalla un mensaje de `No hay datos`, y

si los hay, presenta la lista de las órdenes de elaboración.

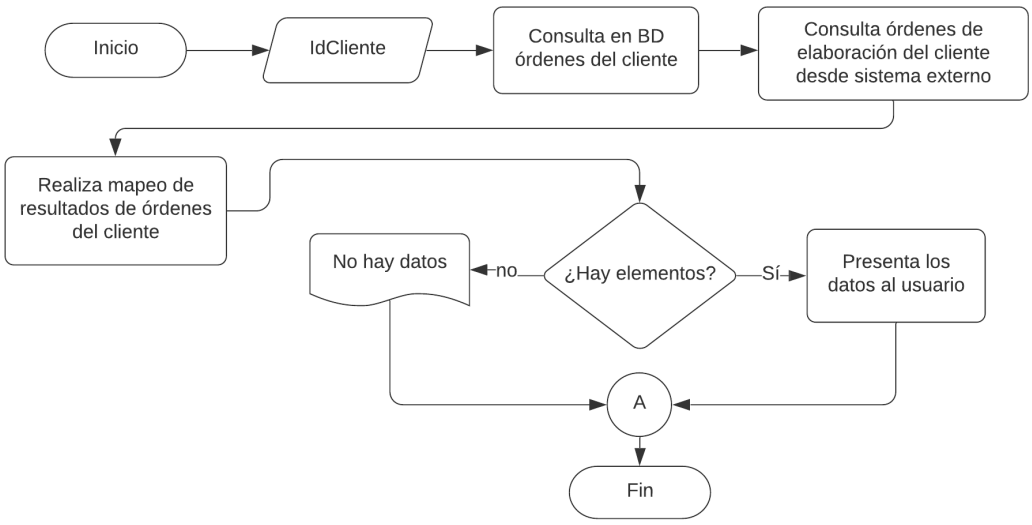

Figura 3.7: Diagrama de flujo de programa políglota con base al escenario seleccionado.

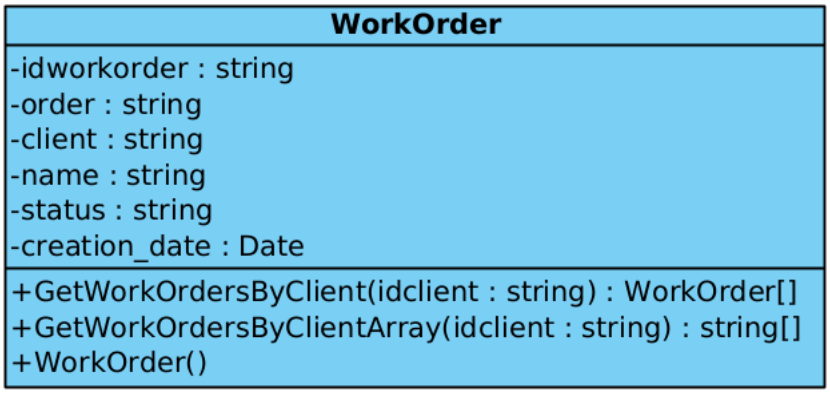

Figura 3.8: Diagrama de clases del objeto `WorkOrder`.

En la Figura 3.8 se observa el diseño de la clase `WorkdOrder` que se usó en el programa con Node.js para tener acceso los datos del sistema externo. Los atributos de la clase son características propias de las órdenes en el sistema, el método `GetWorkOrdersByClient` obtiene un arreglo de objetos de tipo `WorkdOrder` recibiendo como parámetro `idclient` de tipo `String`, el siguiente método `GetWorkOrdersByClientArray` recibe al igual un parámetro `idclient` de tipo `String` y devuelve un arreglo de tipo `String` con los números

de órdenes de elaboración del cliente.

El código se realizó en *JavaScript* [35] con *Node.js* [36], y debido a que es un código largo se explicarán los bloques principales siguiendo la lógica del diagrama de la Figura 3.7.

3.2.3. Implementación y pruebas del escenario de interoperabilidad de lenguajes

En la siguiente sección se realiza la implementación y pruebas del programa políglota que demuestra las capacidades de interoperabilidad de lenguajes con *GraalVM*.

Implementación

A continuación, se presentan la implementación realizada del programa que demuestra las capacidades de interoperabilidad de lenguajes [37] de *GraalVM*, que tiene *JavaScript* potencializado por *Node.js* como *framework* que interactúa con *Java* y *Python* en un mismo entorno.

El bloque de Código 3.13 presenta la consulta en base de datos desde *JavaScript* [38] con *Node.js*, esta consulta obtiene las órdenes de los clientes por medio de su identificador, se almacena el conjunto de datos si se cambia a un format *JSON* como se observa en la línea 69.

Código 3.13: Consulta en BD órdenes del cliente.

```
53 //Retrieving information from DB
54 function getOrdersByClientFromBD(idclient) {
55     return new Promise(resolve => {
56         const client = new Client(connectionData)
57         var jsonordersfrombd = "";
58         client.connect()
59         client.query(" SELECT o.order_number FROM " +
60                 " orders as o INNER JOIN clients as c" +
61                 " ON o.idclient = c.idclient" +
62                 " WHERE c.idclient = '" + idclient + "';")
```

```
63              .then(response => {
64                  var rows = response.rows
65                  var arrayrows = new Array();
66                  for (var i = 0; i < rows.length; i++) {
67                      arrayrows.push(rows[i].order_number)
68                  }
69                  jsonordersfrombd = JSON.stringify(arrayrows);
70                  resolve(jsonordersfrombd)
71                  client.end()
72              })
73              .catch(err => {
74                  client.end()
75              })
76      });
77  }
```

En el Código 3.14 se realiza la ejecución de un programa en *Java*, obteniendo las órdenes de elaboración desde un sistema externo. Se usa la clase de *Java* `WorkOrder` del sistema externo por medio de propiedad políglota `Java.type()` como se ve en la línea 82, después se obtiene el listado de órdenes por el identificador del cliente ejecutando la función `GetWorkOrdersByClientArrayObj` en la línea 83, por último, se hace un formateo a *JSON* en la línea 84.

Código 3.14: Consulta ordenes de elaboración del cliente desde sistema externo en *Java*.

```
79  //Retrieving information from external System
80  function getWorkOrdersByClientFromExternalSystem(idclient) {
81      return new Promise(resolve => {
82         var WorkOrderJava = Java.type('WorkOrder');
83         var list = WorkOrderJava.GetWorkOrdersByClientArrayObj(idclient);
84         var jsonordersfromextsys = JSON.stringify(list);
85         resolve(jsonordersfromextsys);
86      });
87  }
```

En el Código 3.15 realiza la comparativa de conjuntos de órdenes y órdenes de elaboración para obtener la intersección de ambos conjuntos y presentarlos al usuario, esta comparativa se realiza en lenguaje Python. Se ejecuta el elemento políglota `Polyglot.eval()` en la línea 92, donde se especifica como primer parámetro el lenguaje de programación a ejecutar seguido de la línea de código, en este caso se especifica el lenguaje *Python*.

Código 3.15: Comparación de listas de órdenes con *Python*.

```
89 //Compare list from orders and workorders from diferent systems
90 function getOrdersOnProcessOrFinished(orders, workorders) {
91  return new Promise(resolve => {
92  var comparative = Polyglot.eval('python', "set(" + orders + ") & set("
      + workorders + ")")
93  resolve(comparative);
94  });
95 }
```

En Código 3.16, se encuentra el código que representa la lógica del diagrama de la Figura 3.7 donde hace la condición si el conjunto de órdenes de elaboración en proceso tiene o no elementos, si no los tiene presenta un mensaje en color rojo como se ve en la línea 26, y si los tiene presenta un mensaje en color verde como se ve en la línea 28, seguido de las órdenes en proceso.

Código 3.16: Presenta en pantalla resultados por medio de un request *GET* de *Node.js*.

```
20 app.get('/', function(req, res) {
21     var text = 'Welcome!<br> '
22
23     text += "Client = " + clientId + '<br>'
24
25     if (ordersonprocessorfinished.length <= 0) {
26         text += span("No orders in process".red) + " <br>"
27     } else {
28         text += span("Orders in process".green) + " <br>"
29         text += "<ul>"
30         for (var i = 0; i < ordersonprocessorfinished.length; i++) {
31             text += "<li> " + ordersonprocessorfinished[i] + " </li>"
32         }
33         text += "</ul>"
34     }
35
36     res.send(text)
37 })
```

Pruebas

A continuación, se presentan las pruebas realizadas con el programa que demuestra las capacidades de interoperabilidad de lenguajes de *GraalVM*.

Como se observa en la Figura 3.9 se realiza la ejecución del programa `app.js` donde

contiene el código anteriormente explicado. Dicho programa se ejecuta con la siguiente línea desde una terminal:

node –jvm –polyglot app

La opción `--jvm` permite usar propiedades de Java y `--polyglot` permite la ejecución de entornos políglotas en *Node.js*.

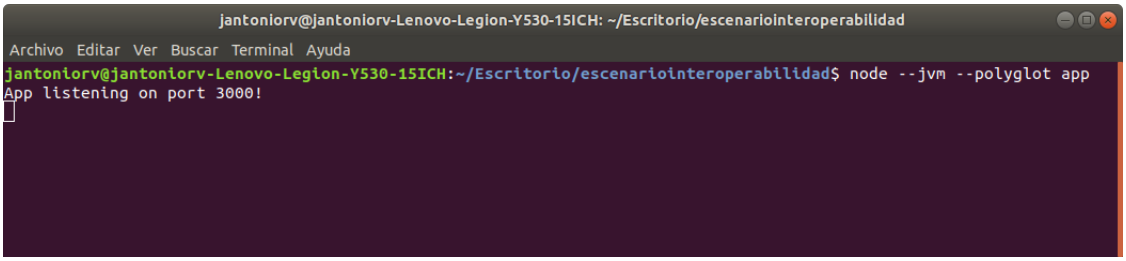

Figura 3.9: Ejecución de programa de *Node.js* desde una terminal.

En la Figura 3.10 se observa el resultado de la ejecución del programa desde el *browser* de *Google Chrome*, desde el servidor local puerto 3000. se observa un mensaje de bienvenida, el identificador del cliente y el listado de órdenes en proceso.

Figura 3.10: Ejecución de programa de *Node.js* desde una terminal.

Capítulo 4

Resultados

En este capítulo se presentan los resultados obtenidos al implementar la programación políglota con *GraalVM* de un caso de estudio en la industria.

4.1. Planteamiento del Caso de Estudio

Se planteó el desarrollo de una aplicación que permita el manejo de la programación políglota con *GraalVM* por medio de un entorno de servicios *REST*, esta parte del sistema se nombró como *back-end*, dichos servicios serían contenidos por medio del manejo de tecnologías *Docker*; y una aplicación *web* responsiva que realice el consumo de dichos servicios por medio de llamadas *API REST*, dicha aplicación sería conocida como *front-end*. La administración de los componentes y funciones de la aplicación implementaron la arquitectura *SOA*.

4.2. Requerimientos del negocio

La Universidad Veracruzana (UV) de Xalapa tiene el requerimiento de una aplicación que permita el análisis de secuencias de ADN (Ácido Desoxirribonucleico) y que dichos resultados se muestren a través de un canal web o móvil. Dicha aplicación cuenta con las operaciones básicas del análisis de cadenas de ADN como lo son el complemento, el

complemento inverso, la obtención del ARN (Ácido Ribonucleico) y sus respectivos codones y proteínas, todo esto a través de un archivo que contenga la información del ADN ya sea de tipo *FASTA* o *GenBank*.

4.3. Bioinformática

4.3.1. Secuencias de ADN

El ADN [39], son las instrucciones que se heredan de un organismo padre a un organismo hijo durante la reproducción.

ADN

El ácido desoxirribonucleico (ADN), contiene un conjunto de órdenes biológicas que hacen de cada ser algo único en el mundo.

Composición del ADN

El ADN está formado por unos componentes químicos básicos llamados nucleótidos. Los cuatro tipos de bases nitrogenadas que se encuentran en los nucleótidos son: adenina (A), timina (T), guanina (G) y citosina (C). El orden, o secuencia, de estas bases determina qué instrucciones biológicas están contenidas en una hebra de ADN como se observa en la Figura 4.1 [40].

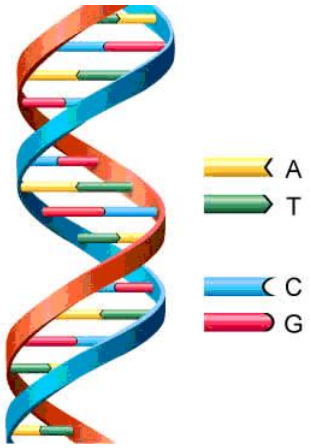

Figura 4.1: ADN - Ácido Desoxirribonucleico.

Tipos de archivos

- *FASTA* (.FASTA). Una secuencia en formato .FASTA [41] comienza con una descripción de una sola línea, seguida de líneas de datos de secuencia. La línea de descripción (*defline*) se distingue de los datos de secuencia por un símbolo mayor que (>) al principio. Se recomienda que todas las líneas de texto tengan menos de 80 caracteres. Una secuencia de ejemplo en formato *FASTA* es:

```
>P01013 GENE X PROTEIN (OVALBUMIN-RELATED)
QIKDLLVSSSTDLDTTLVLVNAIYFKGMWKTAFNAEDTREMPFHVTKQESKPVQMMCMNNSFNVATLPAE
KMKILELPFASGDLSMLVLLPDEVSDLERIEKTINFEKLTEWTNPNTMEKRRVKVYLPQMKIEEKYNLTS
VLMALGMTDLFIPSANLTGISSAESLKISQAVHGAFMELSEDGIEMAGSTGVIEDIKHSPESEQFRADHP
FLFLIKHNPTNTIVYFGRYWSP
```

  No se permiten líneas en blanco en el medio de la entrada *FASTA*.

- *GenBank* (.gb, .gbk). El archivo .gb o .gbk [42] es un formato de biología molecular GenBank, del cual es nativo de la base de datos del Centro Nacional de Información Biotecnológica (NCBI) de EE. UU. También cabe mencionar que es un formato estándar para almacenar e intercambiar secuencias de ADN anotadas, el archivo .gb o .gbk es un archivo de texto sin formato que fue desarrollado en 1982 como parte del proyecto NIH GenBank.

  Un ejemplo de formato GenBank seria el siguiente:

```
  1 qikdllvsss tdldttlvlv naiyfkgmwk tafnaedtre mpfhvtkqes kpvqmmcmnn
 61 sfnvatlpae kmkilelpfa sgdlsmlvll pdevsdleri ektinfeklt ewtnpntmek
121 rrvkvylpqm kieekynlts vlmalgmtdl fipsanltgi ssaeslkisq avhgafmels
181 edgiemagst gviedikhsp eseqfradhp flflikhnpt ntivyfgryw sp
```

4.4. Arquitectura del sistema

El diagrama de la Figura 4.2 representa la arquitectura *SOA* de la solución de software aplicada al caso de estudio. Del lado derecho se muestra lo que es la aplicación *"BioGraalVM"*, la cual es una *API REST* de servicios donde se implementó la programación políglota con *GraalVM* y tambien se implementó internamente la lógica de negocio del análisis de las secuencias de ADN. Del lado izquierdo se presenta la aplicación web responsiva "BioGraal"que se realizó con Angular, el modelo es la parte que consume los servicios web de la aplicación *"BioGraalVM"* y junto con la capa de controladores y vistas trabajan en conjunto para presentar, a traves de un canal web desde un *browser*, los resultados de los análisis de secuencias de ADN al usuario.

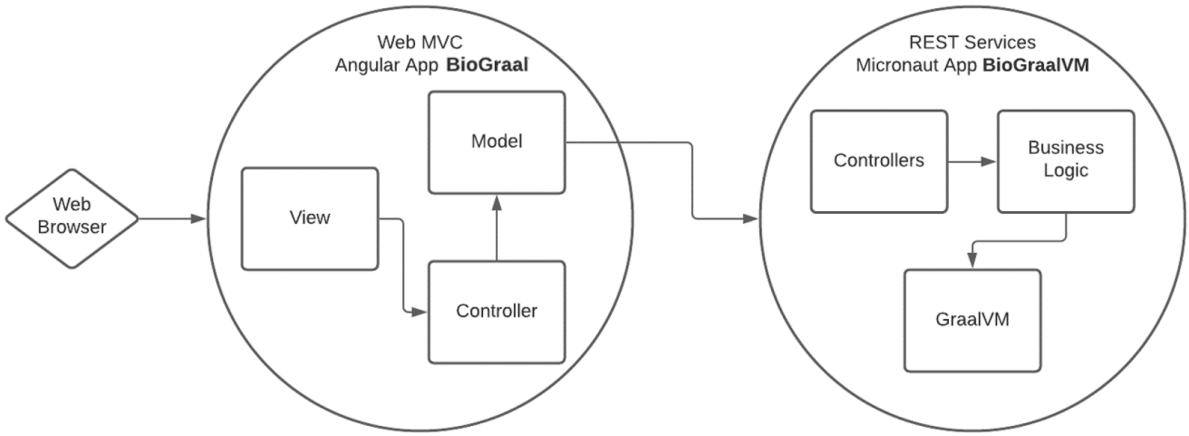

Figura 4.2: Arquitectura del sistema.

4.5. Implementación del caso de estudio

En esta sección se menciona todo lo relacionado a la implementación del caso de estudio aplicado a la industria, mostrando los diseños de diagramas generados de las soluciones junto con muestras de la implementación y manejo de las aplicaciones con información real sobre secuencias de ADN.

4.5.1. BioGraalVM

"BioGraalVM" es el nombre que se le dió a la aplicación que contiene los servicios *REST*, dicha aplicación contiene los procesos internos para el análisis de las secuencias de ADN realizando esto con la propiedades políglotas de *GraalVM*.

Aplicación de SCRUM

En esta sección se especifica la aplicación de la metodología ágil SCRUM [43], para ello se usó el repositorio y herramienta de versionamiento de *software GitHub* [44], dicha herramienta cuenta con una serie de funcionalidades que permiten la administración del desarrollo de *software* y también la administración de tareas durante el desarrollo de sistemas.

GitHub cuenta con una herramienta para la creación y administración de proyectos, dicha herramienta permite crear el proyecto a partir de una plantilla en *GitHub*, en el proyecto "*BioGraalVM*" se creó como un proyecto de tipo "*Kanban with reviews*" (Kanban con revisiones), tal y como se muestra en la Figura 4.3.

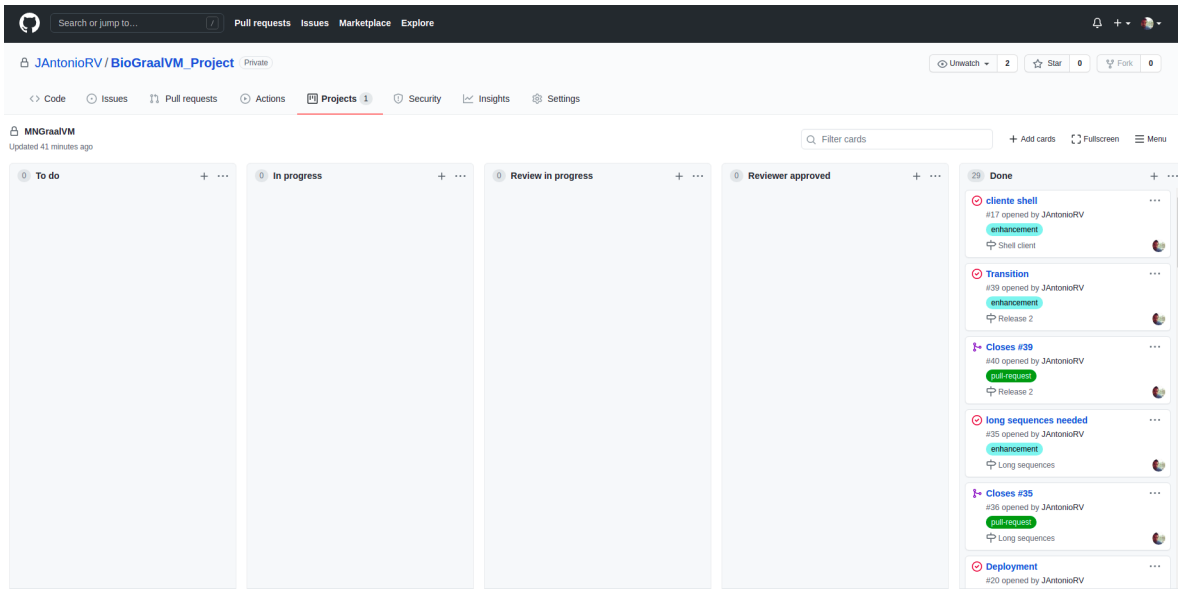

Figura 4.3: Proyecto tipo *Kanban with reviews*.

En el tablero *Kanban* que se presenta en la Figura 4.3 se muestran las tareas por hacer en la columna "*To do*" (Por hacer), en esta columna se presentan las tareas como *Issues* (Asuntos), de los cuales se hablarán mas adelante. En la columna "*In progress*" (En progreso) se presentan las tareas que se encuentran en proceso. En la columna "*Review in progress*" (Revisión en progreso) se presentan las tareas que se enviaron para su revisión por parte de algún colaborador en el proyecto, que en este caso son los directores de tesis, también en esta sección se encuentran los "*Pull Request*", que son peticiones para agregar los nuevos cambios en el codigo al proyecto que se encuentra en el repositorio. En la siguiente columna "*Review approved*" (Revisión aprobada) se presentan los "*Pull Request*" que se aprobaron por el revisor y que ya se combinaron con el repositorio destino. Y por último, en la columna "*Done*" (Hecho o terminado) se encuentran los *Issues* que se completaron junto con sus "*Pull Request*" completados. Los *Issues* y los "*Pull Request*" sólo se cierran cuando los cambios llegan hasta el repositorio principal ya sea "*main*" o "*master*", dependiendo de la configuración con la cual se creó el proyecto.

Los tipos de tareas que se crean en el proyecto son los que se muestran en la Figura 4.4, las más usadas son "*enhancement*", que son como se les llama a las tareas o *Issues*, y

los "*Pull Request*".

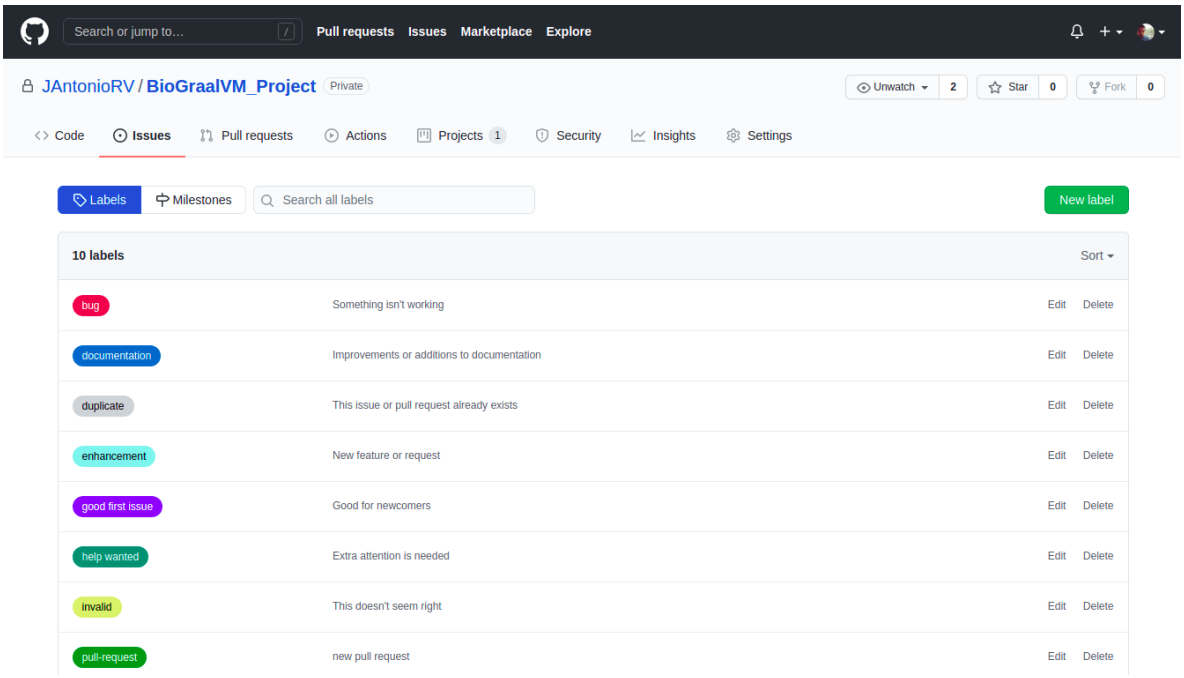

Figura 4.4: Tipos de etiquetas de tareas en *GitHub*.

Con la herramienta *GitHub* también se pueden definir los *sprints*, esto se realiza en la sección de "*Milestones*" (Figura 4.5), en esta sección se definen cada uno los *sprints* planeados con los revisores o el equipo SCRUM, se especifican los objetivos de cada *sprint* y también las fechas de entrega o *Deadlines*, todo dentro de la definicion de cada *Milestone*. Tambien los *Milestones* sirven para definir a que *sprint* pertenece cada *issue* y así tener una mejor planeación de cada tarea a realizar.

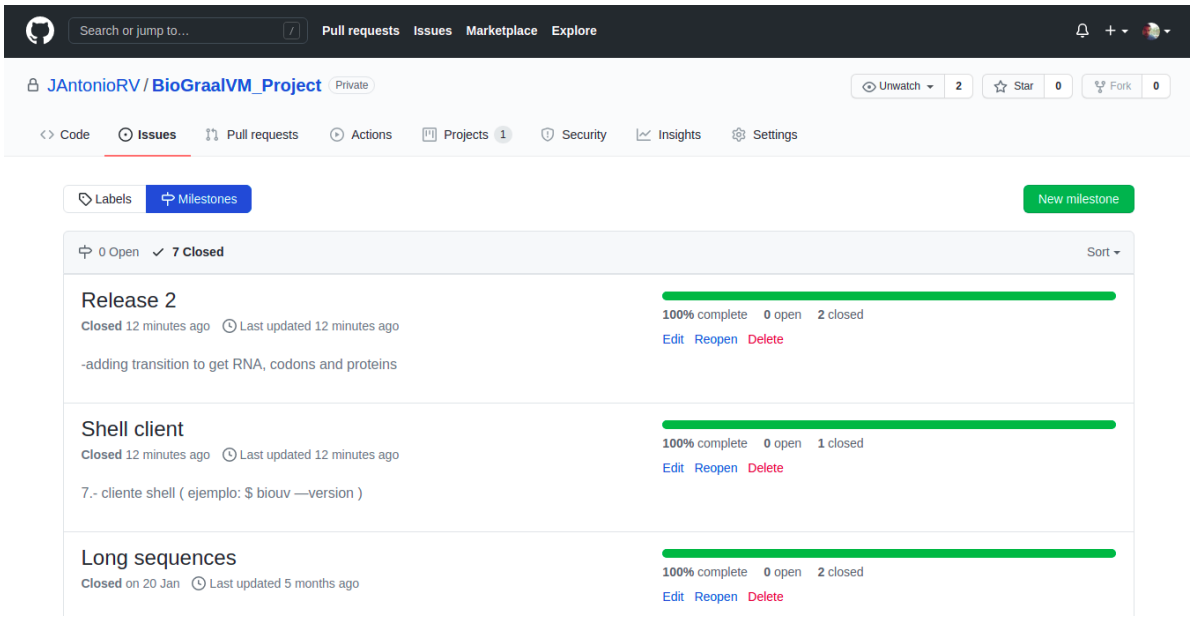

Figura 4.5: Sección *Milestones*.

En la sección de "*Issues*" (Figura 4.6) se definen las tareas planeadas a realizar. Cada *issue*, como se mencionó anteriormente, pertenece a un *Milestone* específico. Cuando se realiza un cambio en el proyecto, se mapea el cambio con el número del *issue* para que al momento de hacer el "*Pull Request*" al repositorio "*master*" se cierre el *issue* y se tome como tarea realizada.

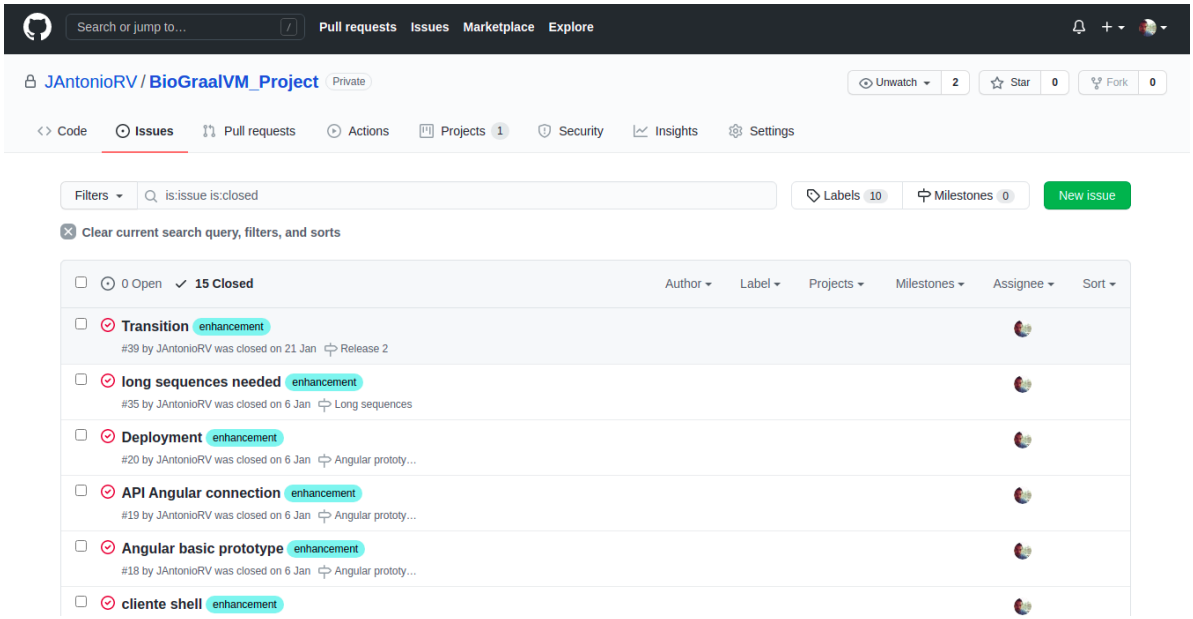

Figura 4.6: Sección *Issues*.

También se pueden crear *Tags* (etiquetas) de los *Release* a realizar durante el desarrollo e implementación del proyecto, tal y como se ve en la Figura 4.7

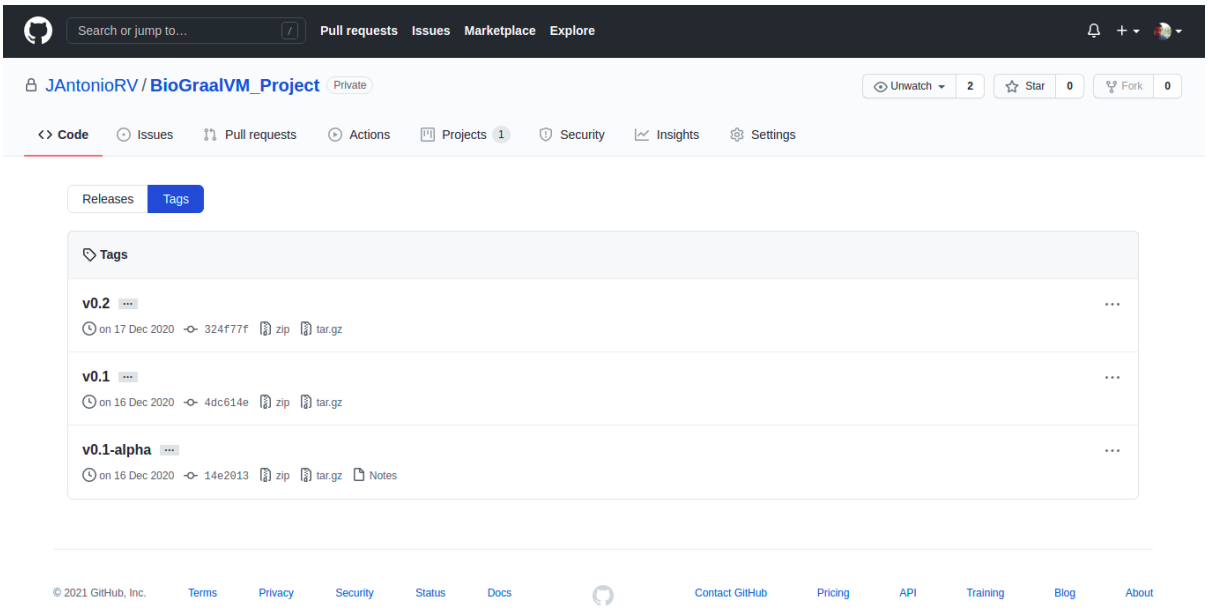

Figura 4.7: Sección *Tags*.

Por cada *issue* terminado, y cada *Milestone* completado, se realizaba la conclusión *sprint* planeado.

Beneficios programación políglota

Dentro de los beneficios de la programación políglota aplicados a la *API REST "BioGraalVM"* se tienen lo que es el manejo de múltiples lenguajes de programación, uso de *frameworks* para manejar la programación políglota y para manejar las propiedades nativas, dichos beneficios se mencionan a continuación:

- *GraalVM CE 20.3 Java 8*

  Con esta versión de *GraalVM* (Figura 4.8) se manejó todo lo que comprende la programación políglota en un mismo entorno, dicha versión soporta lenguajes como *Java, JavaScript, LLVM, Pyhton, Ruby, R, WebAssembly, C y C++*. En el entorno políglota se usaron los lenguajes de *Java* como lenguaje *host* y *Python* como lenguaje

guest. El lenguaje *host* es aquel que ejecuta el contexto políglota por medio de *GraalVM* y contiene el o los lenguajes de programación a usar en el sistema.

Figura 4.8: *GraalVM*.

En el Código 4.1 se presenta la clase `Complement.java`. Dicha clase implementa la ejecución del código en *Python* `SeqComplement.py` que realiza el análisis de la secuencia de ADN. Dicha ejecución se realiza a través del contexto de *GraalVM* usando sus propiedades de ejecución e interpretación políglota.

Código 4.1: Clase Complement.java.

```
1  package com.msc.polyglotfunctions;
2
3  import com.msc.graalvmcontext.GraalVMContext;
4  import io.micronaut.core.annotation.Introspected;
5  import org.graalvm.polyglot.Context;
6  import org.graalvm.polyglot.Value;
7
8  @Introspected
9  public class Complement extends SequenceResourses implements
       ISequence{
10     @Override
11     public String run(String sequence) {
12         String complement = "";
13         String pythonscript="";
14         //Create context instance
15         Context context = GraalVMContext.getInstance();
16         //Get python script from resources
17         pythonscript = readResource("com/msc/pythonscripts/
       SeqComplement.py");
18         //Execute eval to load python function
19         context.eval("python",pythonscript);
20         //Execute python function setting sequence
21         Value pyfunction = context.getPolyglotBindings().getMember("
       complement");
22         Value sqcomplement = pyfunction.execute(sequence);
23         //Casting result as String
24         complement = sqcomplement.asString();
25         return complement;
26     }
27
```

```
28      @Override
29      public Object run_object(String sequence) { return null; }
30  }
```

- *Python*

  Este lenguaje (Figura 4.9) se usó para la ejecución del procesamiento de las secuencias de ADN, dicho lenguaje contiene rutinas y métodos predefinidos que permiten la ejecución rápida de las funcionalidades requeridas y así obtener los resultados de los análisis de ADN.

Figura 4.9: *Python*.

En el Código 4.2 se presenta el *script* `SeqComplement.py` que realiza la obtención del complemento de una secuencia de ADN, en las líneas 2 y 3 se definen las directivas que permiten la comunicación entre *Java* y *Python* por medio de *GraalVM*.

Código 4.2: *Script* SeqComplement.py.

```
1  import polyglot;
2  @polyglot.export_value
3  def complement(sequence):
4      sequencenobreaks = sequence.replace("\n","")
5      sequenceUpper = sequencenobreaks.upper()
6      arraysequence = list(sequenceUpper)
7      size = len(arraysequence)
8      arraycomplementsq = [" "] * size
9      npos = 0
10     complementsq = ''
11
12     for s in arraysequence:
13         if s == 'A':
14             arraycomplementsq[npos] = 'T'
15         elif s == 'T':
16             arraycomplementsq[npos] = 'A'
17         elif s == 'G':
18             arraycomplementsq[npos] = 'C'
```

```
19          elif s == 'C':
20              arraycomplementsq[npos] = 'G'
21          npos = npos + 1
22
23      complementsq = ''.join(arraycomplementsq)
24      return complementsq
```

- Compilador *LLVM*

  Esta funcionalidad (Figura 4.10) permite la ejecución y compilado de los lenguajes *C* y *C++* para la creación de las imágenes nativas, que son una propiedad de suma importancia e impacto en la programación políglota con *GraalVM*.

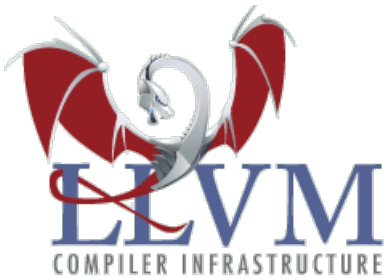

Figura 4.10: *LLVM*.

  El componente *LLVM* se instala usando las siguientes líneas desde una terminal usando `gu` de *GraalVM*:

```
$ gu install llvm-toolchain
$ export LLVM_TOOLCHAIN=$(lli --print-toolchain-path)
```

- *Micronaut Version: 1.0.0.RC2*

  Micronaut [45] (Figura 4.11) es un *framework full stack* de *Java*, completo, moderno y basado en *JVM*, diseñado para crear aplicaciones *JVM* modulares y fácilmente probables con soporte para *Java, Kotlin y Groovy.*

Figura 4.11: Micronaut

Micronaut se usó como *framework* para el proyecto ya que permitió crear el esqueleto principal de la aplicación *API REST "BioGraalVM"*, también porque cuenta con una integración con la máquina virtual *Graal*, y gracias a esto se pueden manejar sus propiedades políglotas y nativas de manera eficiente y segura. *Micronaut* también permitió implementar la configuración y creación de la imagen nativa con las propiedades necesarias para ejecutar la *API REST*.

En el Código 4.3 se presenta la clase `ComplementController.java` como un controlador que resuelve las llamadas *API REST*, dicho controlador obtiene la secuencia de ADN por medio de una llamada *POST* y retorna el complemento, resultado del análisis del ADN.

Código 4.3: Clase ComplementController.java.

```
1  package com.msc.mngraalvm;
2  import com.msc.model.Result;
3  import com.msc.sequences.Sequences;
4  import io.micronaut.http.annotation.Controller;
5  import io.micronaut.http.annotation.Get;
6  import io.micronaut.http.HttpStatus;
7  import io.micronaut.http.annotation.Post;
8  import javax.annotation.Nullable;
9
10 @Controller("/complement")
11 public class ComplementController {
12     private final Result result;
13
14     @Get("/")
15     public HttpStatus index() {
16         return HttpStatus.OK;
```

```
17      }
18
19      @Post("/complement")
20      Result complement(String s) {
21          try{
22              result.setResult(Sequences.complement(s));
23              result.setOk(true);
24          }catch(Exception ex){
25              result.setError(ex.getMessage());
26          }
27          return result;
28      }
29
30      public ComplementController(@Nullable Result result) {
31          this.result = result;
32      }
33  }
```

Se utiliza el método *POST* ya que, debido al tamaño de la secuencia de ADN, el método *GET* no soporta cadenas tan grandes en sus peticiones y respuestas.

Propiedades nativas

Dentro de las propiedades nativas de la *API REST "BioGraalVM"* se tienen los componentes que se usaron para crear la imagen nativa, los gestores de versionamiento de herramientas y el contenedor que ejecuta la imagen nativa, dichas propiedades se mencionan a continuación:

- *Native Image*

  Native Image [46] (Figura 4.12) es una tecnología para compilar con anticipación código *Java* en un ejecutable independiente, llamado imagen nativa. Este ejecutable incluye las clases de la aplicación, las clases de sus dependencias, las clases de la biblioteca en tiempo de ejecución y el código nativo vinculado estáticamente de *JDK*. No se ejecuta en *JVM*. El programa resultante tiene un tiempo de inicio más rápido y una sobrecarga de memoria en tiempo de ejecución más baja en comparación con una *JVM*.

Figura 4.12: Imagen nativa de GraalVM

En el Código 4.4 se presenta la clase `GraalVMContext.java`. Dicha clase implementa el patrón de diseño *Singleton*, donde se crea un único contexto de *GraalVM* en el que se ejecutarán los entornos políglotas usando diferentes lenguajes de programación, en este caso específicamente *Python* y *Java*. En este contexto se definen los valores de las propiedades `.option()`, dichas propiedades se asignan para localizar de manera interna las bibliotecas de *GraalVM* necesarias al momento de usar un lenguaje en específico y ser localizadas de manera externa a la imagen nativa.

Código 4.4: Clase GraalVMContext.java.

```
1  package com.msc.graalvmcontext;
2  import io.micronaut.core.annotation.Introspected;
3  import org.graalvm.polyglot.Context;
4  import org.graalvm.polyglot.HostAccess;
5  import org.graalvm.polyglot.PolyglotAccess;
6
7  @Introspected
8  public class GraalVMContext {
9    private static Context uniqueContextInstance;
10   private GraalVMContext() { }
11   public static Context getInstance() {
12     if(uniqueContextInstance == null){
13       try{
14         Context context = Context
15             .newBuilder()
16             .allowAllAccess(true)
17             .allowNativeAccess(true)
18             .allowPolyglotAccess(PolyglotAccess.ALL)
19             .allowHostAccess(HostAccess.ALL)
20             .allowIO(true)
21                         .option("python.SysPrefix", "/opt/graalvm-ce
    -java8-20.3.0/jre/languages/python")
22                         .option("python.CoreHome", "/opt/graalvm-ce-
    java8-20.3.0/jre/languages/python/lib-graalpython")
```

```
23                          .option("python.StdLibHome", "/opt/graalvm-
    ce-java8-20.3.0/jre/languages/python/lib-python/3")
24                          .option("python.Executable", "/opt/graalvm-
    ce-java8-20.3.0/jre/languages/python/bin/graalpython")
25                          .option("python.CAPI", "/opt/graalvm-ce-
    java8-20.3.0/jre/languages/python/lib-graalpython")
26                .build();
27            uniqueContextInstance = context;
28          }
29          catch (Exception ex){
30            ex.printStackTrace();
31            throw  ex;
32          }
33       }
34       return uniqueContextInstance;
35     }
36 }
```

- *SDKMAN 5.9.1+575*

  SDKMAN! [47] (Figura 4.13) es una herramienta que administra versiones paralelas de varios kits de desarrollo de *software* en sistemas basados en *Unix*. Proporciona una interfaz de línea de comandos (*CLI*) y una *API* convenientes para instalar, cambiar, eliminar y enumerar candidatos. *SDKMAN!* se usó principalmente para el manejar las versiones de *GraalVM* a usar durante el desarrollo de la aplicación *BioGraalVM*.

Figura 4.13: SDKMan

SDKMAN! se instala por medio de las siguientes líneas ejecutándolas desde una terminal:

```
$ curl -s "https://get.sdkman.io" | bash
$ source "$HOME/.sdkman/bin/sdkman-init.sh"
$ sdk version
```

- *GraalVM Java version 20.3.0.r8-grl*

  GraalVM Java version 20.3.0.r8-grl [48] es una versión de *GraalVM* que se instala por medio de *SDKMAN*, principalmente porque tiene una integración directa con *Micronaut*, su forma de instalación y uso es ejecutando las siguientes líneas desde una terminal:

```
$ sdk install 20.3.0.r8-grl
$ sdk use java 20.3.0.r8-grl
```

- *Docker version 19.03.6*

  Se utilizó Docker [49] (Figura 4.14) como contenedor de las aplicaciones *"BioGraalVM"* y la aplicación "BioGraal", cada contenedor con su configuración y *software* necesarios para la ejecución de cada una de las aplicaciones. En el Código 4.5 se presenta el contenido del archivo `Dockerfile`. El archivo contiene las líneas a ejecutar para la creación de la imagen que se ejecutará en el *docker*, y también una serie de órdenes que crearán la imagen nativa desde *GraalVM*, dichas órdenes instalan los componentes necesarios para la creación y ejecución de la imagen nativa con *GraalVM* y la configuración necesaria para usarse como la aplicación *API REST "BioGraalVM"* desde el puerto 8080.

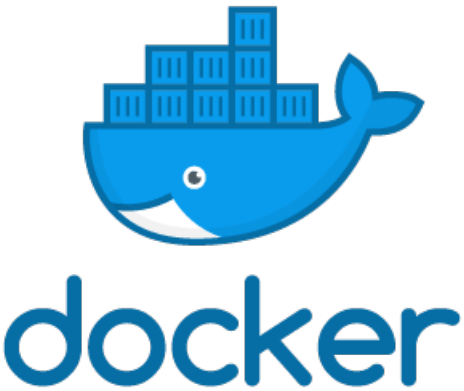

Figura 4.14: Docker

Código 4.5: Dockerfile

```
# use graalvm image
FROM jantoniorv/graalvm-ce:20.3.0-java8

# expose your port, 8080 fo Micronaut application
EXPOSE 8080

# copy the fat jar
COPY build/libs/*-all.jar mngraalvm.jar

# Adding files then build
ADD . build

# Installs native-image if it not exists
RUN gu install native-image

#Installs python language
RUN gu install python

# run the native image compiler
RUN native-image --no-server --language:python --class-path
    mngraalvm.jar

# the native command will be used to run the application
CMD ./mngraalvm
```

Por último para iniciar la aplicación se ejecutan las siguientes líneas de código desde una terminal:

```
$ docker build -t "jantoniorv/biouv" .
$ docker run -d -p 8080:8080 jantoniorv/biouv
```

La primera línea realiza la creación de la imagen que se ejecutará dentro del contenedor *Docker*, dicha línea se ejecuta dentro del directorio donde se encuentra el archivo `Dockerfile`. La segunda línea realiza la ejecución del programa "BioGraalVM" desde el contenedor *Docker* escuchando desde el puerto `8080:8080` (Esta configuración es local, si es desde el servidor sería `8192:8080`).

Diagramas de clases

En esta sección se presenta lo que es el diagrama de clases (Figura 4.15) de la aplicación *"BioGraalVM"*.

Del lado izquierdo se tiene el paquete `com.msc.model`, contiene las clases para el envío de las respuestas del servicio *REST* como objetos *JSON*; el paquete `com.msc.mngraalvm` contiene los controladores que son los responsables de atender las llamadas *REST* a la aplicación; el modelo `com.msc.sequences` contiene la clase que se usó para la implementación del patrón de diseño *Facade*, dicho patrón permite tener el acceso a las funciones que se implementaron internamente en la aplicación desde un sólo punto y así servir de origen en los controladores *REST*; el paquete `com.msc.polyglotfunctions` contiene las clases que ejecutan los códigos políglotas, esto gracias a la propiedad de programación políglota de *GraalVM*, los códigos a ejecutar son las funciones que realizan el análisis de las secuencias de ADN y retornan el resultado de dichos análisis; el paquete `com.msc.pythonscripts` contiene los *scripts* en *Python* de los procesos y métodos a ejecutar para analizar las cadenas de ADN, dichos *scripts* se ejecutan desde los entornos políglotas en el paquete anteriormente mencionado; y por último el paquete `com.msc.graalvmcontext` contiene una clase llamada `GraalVMContext` de la cual se implementó el patrón *Singleton*, esta clase contiene el contexto de *GraalVM*, este contexto es el entorno principal que permite la ejecución políglota, funciona como una *API* que realiza la comunicación entre el entorno donde ingresan los lenguajes de programación a ejecutar y el *kernel* políglota de *GraalVM* que a su vez realiza la interpretación, compilación y ejecución del código de la aplicación. Todas estas clases integran el funcionamiento de la aplicación *"BioGraalVM"*.

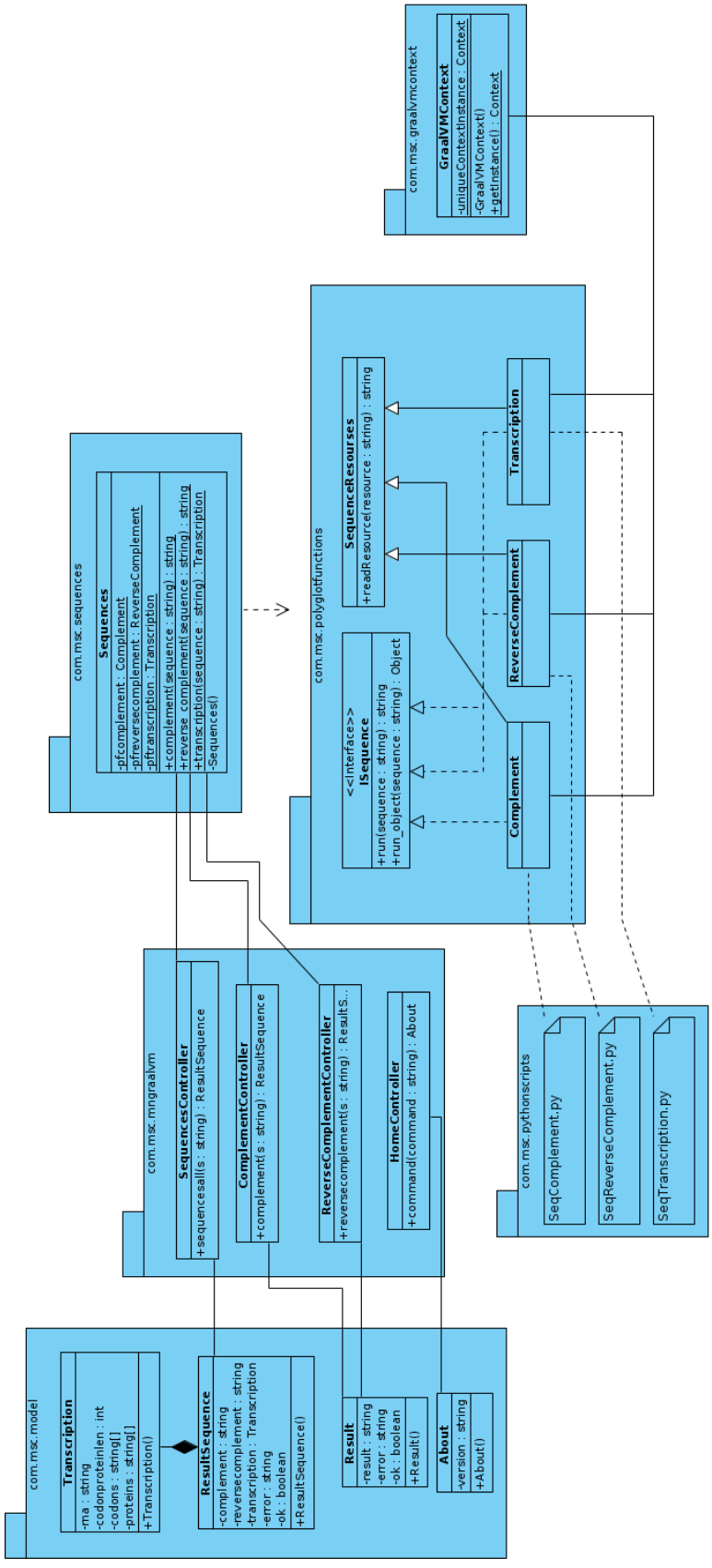

Figura 4.15: Diagrama de clases de *"BioGraalVM"*

Diagramas de componentes

En esta sección se muestra lo que es el diagrama de componentes (Figura 4.16) de la aplicación *REST "BioGraalVM"* y de la aplicación *web* "BioGraal".

Se presenta cómo tiene organizados sus componentes internos, componentes externos que necesitan para su funcionamiento, y el cómo se comunican entre ambas aplicaciones. Del lado izquierdo se observa el componente ejecutable *Docker*, este componente es el contenedor de la aplicación *API REST "BioGraalVM"*, el *Docker* se ejecuta desde el servidor con el puerto *TCP* 8192 abierto a llamadas *HTTP*. Internamente el componente se integra de más componentes como lo es la aplicación *REST*, dicha aplicación se elaboró usando el *framework Micronaut*, e internamente cuenta con más componentes como lo son los paquetes de modelos, controladores, clases para el análisis de secuencias, clases para la ejecución políglota del programa, el paquete que contiene los *scripts* en *Python*, y por último el paquete que contiene el contexto de *GraalVM*. A su vez el contenedor *Docker* tiene el componente *GraalVM* que es una una versión para *dockers* que se usa para la ejecución de los códigos políglotas en la aplicación. Del lado derecho se tiene lo que es la aplicación "BioGraal", internamente tiene los componentes escenciales que maneja *Angular* como lo son las vistas y los controladores usando *JavaScript*, y el modelo y los servicios usando *TypeScript*, dicha aplicación también se encuentra en un contenedor *Docker* que se ejecuta desde el servidor comunicándose con la aplicación *"BioGraalVM"* por medio del puerto 8192, y tiene el puerto *TCP 8082* como puerto de comunicacion para *HTTP*.

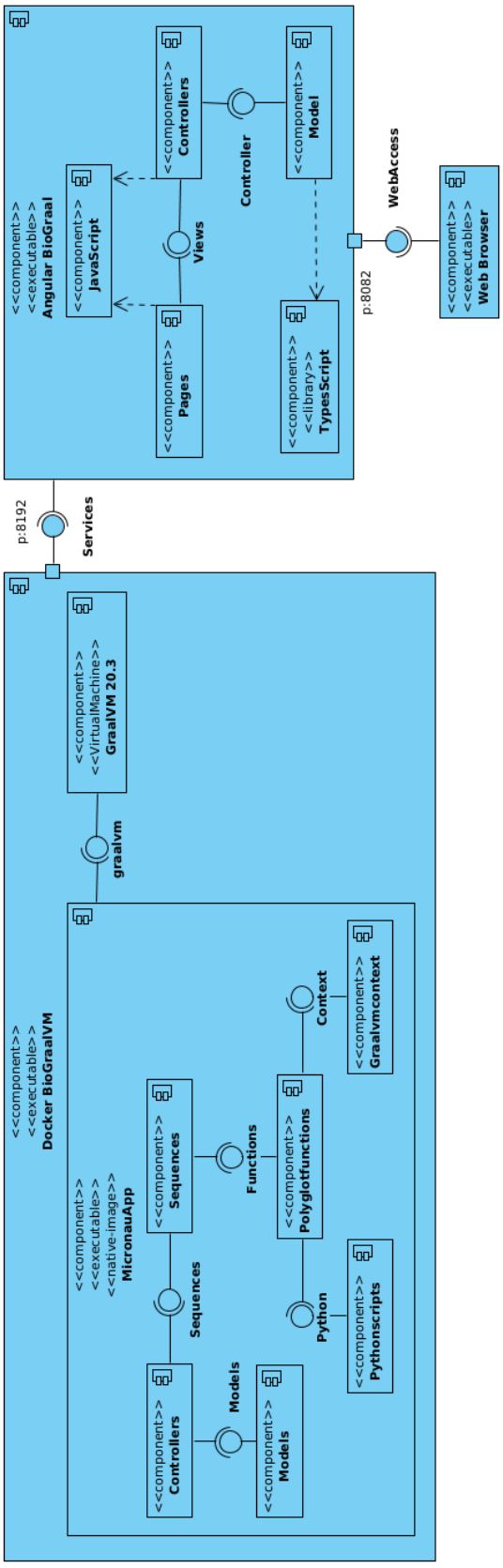

Figura 4.16: Diagrama de componentes de *"BioGraalVM"* y "BioGraal"

Uso de la aplicación

En esta sección se presenta el uso de la aplicación "BioGraalVM", acerca de cómo es que se realizan unas llamadas *REST* a la aplicación.

Para iniciar la aplicación se ejecuta la siguiente línea desde una terminal como se ve en la Figura 4.17:

```
$ docker run -d -p 8080:8080 jantoniorv/biouv
```

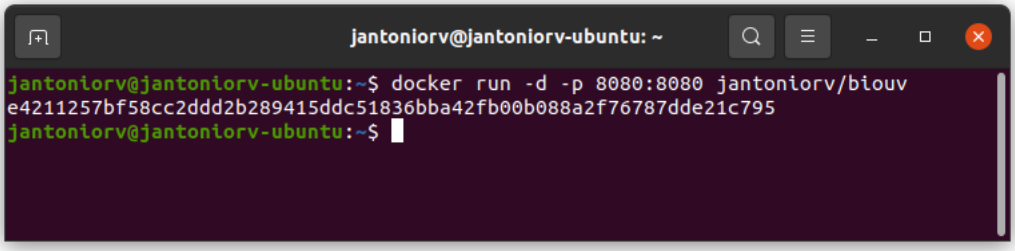

Figura 4.17: Ejecución del programa "BioGraalVM" desde *Docker*.

Después se realizó una llamada tipo *curl* desde una terminal emulando una llamada *POST* enviando un archivo *JSON* con información de una secuencia de ADN solicitando el análisis por medio del servicio `Complement` (Figura 4.18):

```
$ curl -d "@data.json" -H "Content-Type: application/json" -X POST
http://127.0.0.1:8080/complement/complement
```

Figura 4.18: Llamada *POST* al servicio `Complement`

La prueba anterior se realizó con la ejecución local de la aplicación, la siguiente prueba se realizó hacia el servidor remoto donde está alojada la aplicación (Figura 4.19) ejecutando la siguiente línea desde una terminal:

```
$ curl -d "@data.json" -H "Content-Type: application/json" -X POST
http://104.154.228.129:8192/complement/complement
```

Figura 4.19: Llamada *POST* al servicio `Complement` desde el servidor de la aplicación

La siguiente llamada es de tipo *GET* (Figura 4.20), se realizó hacia la aplicación para obtener la versión de la misma usando la siguiente línea desde una terminal:

```
$ curl -H "Content-Type: application/json" -X GET
http://127.0.0.1:8080/home/about?command=v
```

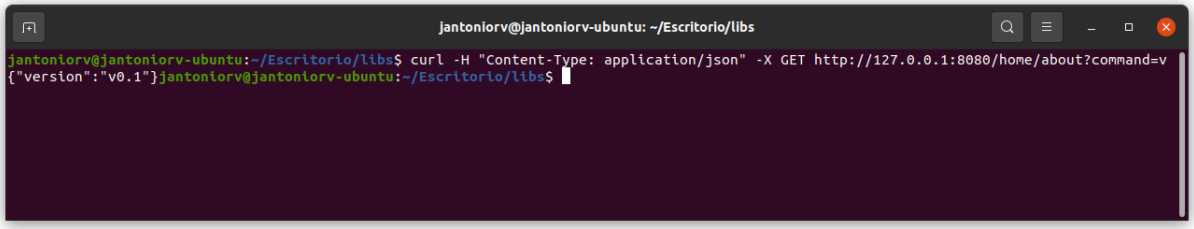

Figura 4.20: Llamada *GET* para obtener la version de la aplicación

.

4.5.2. BioGraal

En esta sección se menciona lo relacionado al desarrollo de la aplicación *web* BioGraal, las tecnologías que se ocuparon, como se implementaron y el uso de la aplicación.

Aplicación de SCRUM

Al igual que en el desarrollo de la aplicación "*BioGraalVM*", se aplicó la metodología ágil SCRUM, la aplicación de SCRUM se realizó de manera paralela debido a que los requerimientos se daban por ambas partes, tanto en *back-end* con "*BioGraalVM*", como con *front-end* con "*BioGraal*", sin embargo, cada proyecto llevaba su planeación y ejecución de *sprints* por separado, tal y como se muestra en esta sección con el desarrollo de la aplicación "*BioGraal*".

Se presenta la creación del proyecto de tipo "*Kanban with reviews*" como se ve en la Figura 4.21.

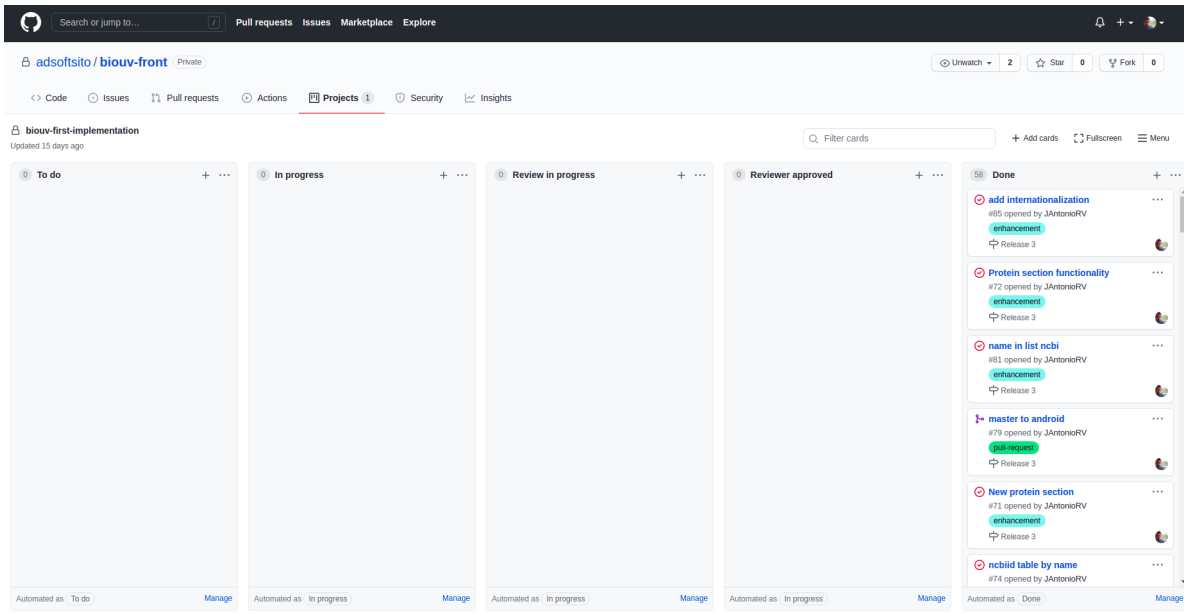

Figura 4.21: Proyecto tipo *Kanban with reviews* de "*BioGraal*".

También se presentan las definiciones de los *Milestones* (Figura 4.22), cada uno con sus objetivos a completar en el desarrollo de la aplicacion *web* "*BioGraal*".

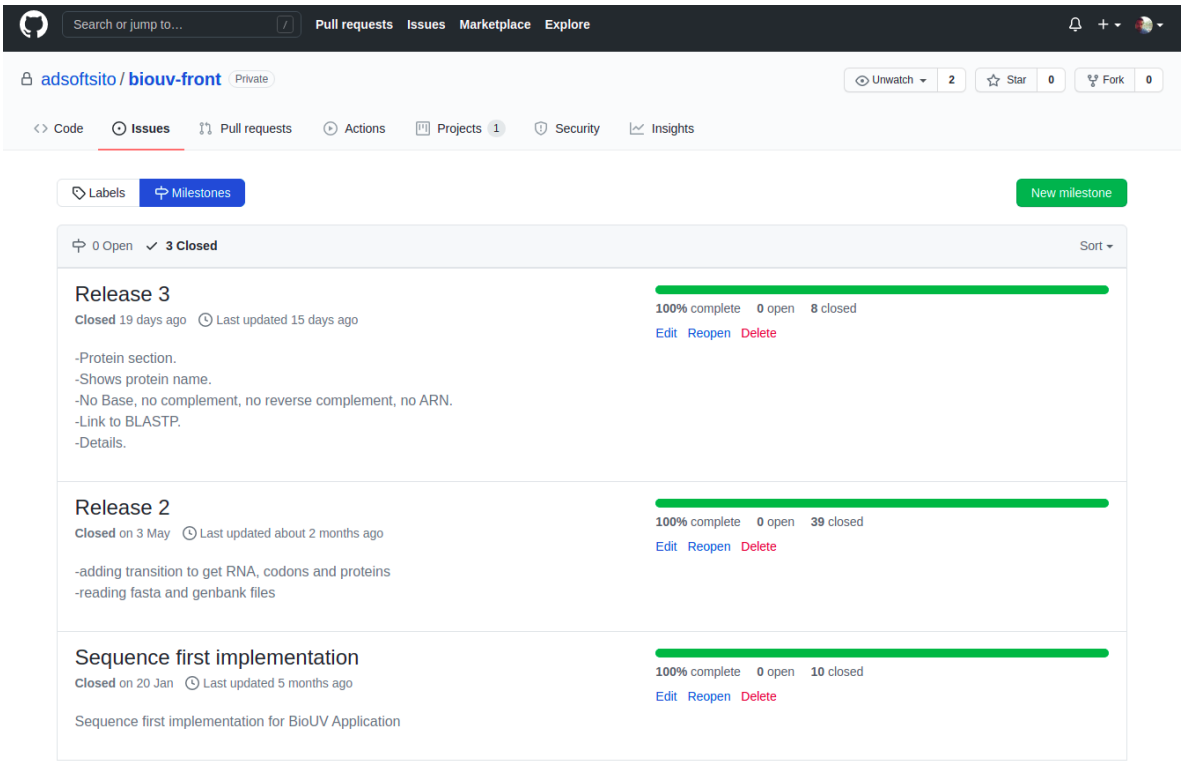

Figura 4.22: Sección *Milestones* de "*BioGraal*".

Por último, se presentan los *issues* (Figura 4.23) realizados para completar las tareas establecidas en los *Milestones* referentes a la aplicación "*BioGraal*".

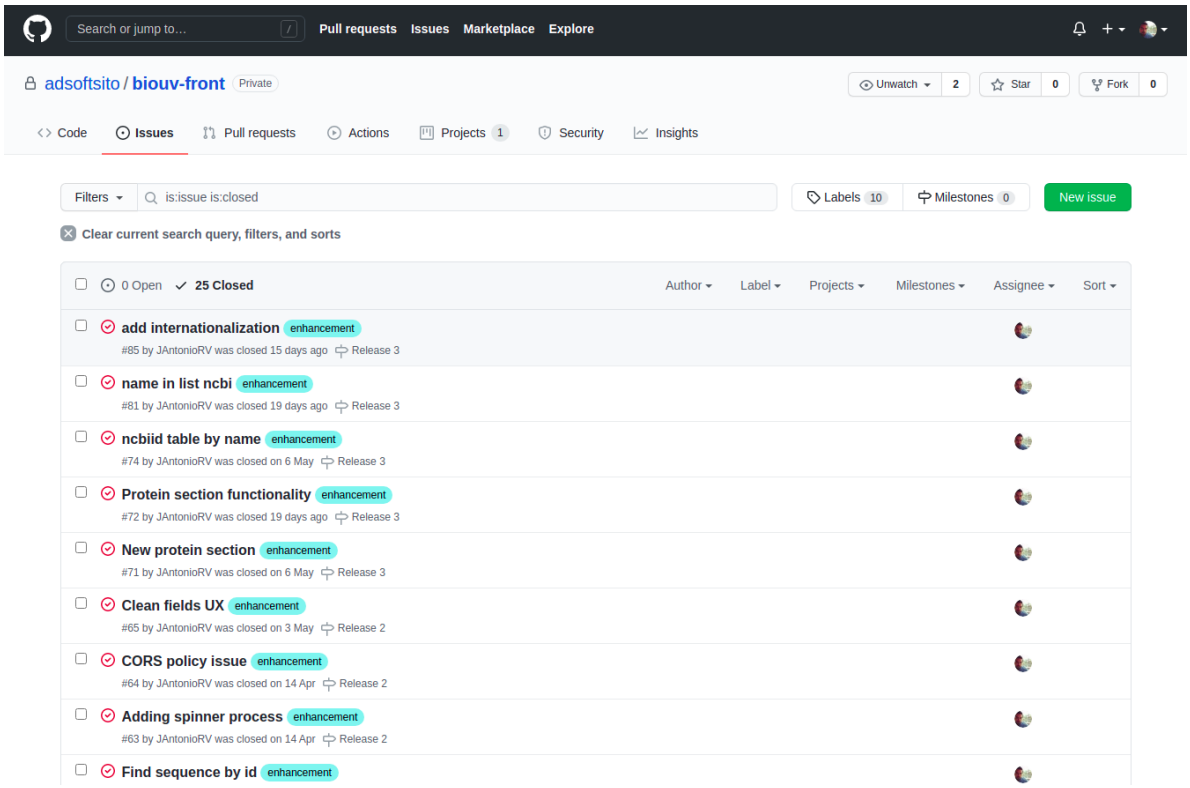

Figura 4.23: Sección *Issues* de "*BioGraal*".

Algunos *issues* se completaron de manera concurrente con los *issues* de "*BioGraalVM*", debido a que como se mencionó anteriormente, hubo tareas de desarrollo que iban de la mano entre ambas aplicaciones "*BioGraal*" y "*BioGraalVM*".

Angular CLI 11.0.6

Angular [50] (Figura 4.24) es un marco de diseño de aplicaciones y una plataforma de desarrollo para crear aplicaciones de una sola página eficientes y sofisticadas.

Figura 4.24: Logo *Angular*

Se usó *Angular* [51] como marco de trabajo *front-end* de la aplicación "BioGraal" aprovechando sus propiedades de desarrollo *single-page*, páginas tipo *responsive* y su propiedad de escalabilidad de componentes y elementos en el desarrollo de la aplicación *web*.

En el Código 4.6 se presenta la función `getComplement`, dicha función se encarga de la llamada *API REST* al *back-end "BioGraalVM"*, específicamente la función llama al servicio *Complement* por medio de una peticion *POST*, y como resultado se obtiene el complemento de la cadena de ADN analizada, dicha función forma parte del archivo de tipo *TypeScript* `graalv.services.ts`.

Código 4.6: Llamada *API REST* al servicio *Complement*

```
54 // HttpClient POST complement method
55 getComplement(sequence): Observable<any> {
56   return this.http.post<any>(this.apiURL + '/complement/complement', { s
       : sequence }, this.httpOptions)
57   .pipe(
58     retry(1),
59     catchError(this.handleError)
60   )
61 }
```

En el Código 4.7 se presenta la función `clickaction()`, esta función es la que se ejecuta al momento de dar clic en el botón "Enviar" en la aplicación *web* "BioGraal", realiza la ejecución de la clase que contiene las llamadas a los servicios *API REST*. Después de obtenido el resultado, se analiza el contenido y se realiza un formateo de cada una de los resultados analizados para presentarse de manera apropiada al usuario en la vista. Dicha función forma parte del archivo de tipo *TypeScript* `sequences.component.ts`.

Código 4.7: Función clickaction().

```
54   clickaction(){
55     this.showSpinner = true;
56     this.complement = '';
57     this.reversecomplement = '';
58     this.rnavalue = '';
59     this.basesequence = '';
60
61     this.graalvService.getSequencesAll(this.txtsequence)
```

```
62      .subscribe((data: any) => {
63        //Gets base sequence
64        this.basesequence = this.FortmatSequences(this.PrettyBaseSequence(
    this.txtsequence));
65
66        //Gets complement
67        this.complement = this.FortmatSequences(data.complement);
68
69        //Gets reverse complement
70        this.reversecomplement = this.FortmatSequences(data.
    reversecomplement);
71
72        //Gets RNA
73        this.rnavalue = this.FortmatSequences(data.transcription.rna);
74
75        //Gets codons and proteins
76        this.rangeTopValue = Math.max.apply(Math, $.map(data.transcription
    .proteins, function (el) { return el.length }))
77        this.getCodonsFiltered(data.transcription, this.rangeLowValue,
    this.rangeTopValue);
78        this.collectioncodonsandproteins = data.transcription;
79        this.showSpinner = false;
80      });
81    }
```

La siguiente línea de código se ejecuta desde una terminal para iniciar la aplicación *web* "BioGraal" desde un contenedor *Docker*, se da el nombre del contenedor con `-name` y con `-p` se asigna el puerto por el que la aplicación resolverá las llamadas *HTTP*, en este caso sería por el puerto 80. En `APP-PATH` iría la ubicación del proyecto *web*.

```
$ docker run -dit --name front-map -p 80:80 -v /APP-PATH/dist:/usr/local/apache2/htdocs/
 httpd
```

Uso de la aplicación

En esta sección se presentan las pruebas realizadas en la aplicación *web* "BioGraal", las consultas de análisis de ADN, los resultados obtenidos y cómo se presentan en pantalla al usuario.

En la Figura 4.25 se presenta la pantalla de inicio de la aplicación "BioGraal".

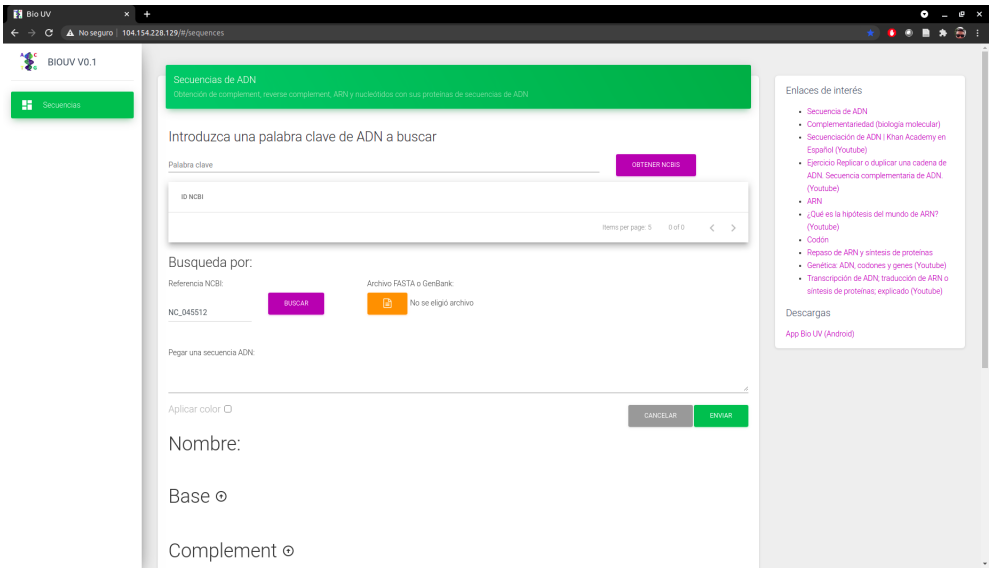

Figura 4.25: Pantalla de inicio de la app web "BioGraal"

Se tiene la funcionalidad de obtener un listado de identificadores de nucleótidos desde el *NCBI* (*National Center for Biotechnology Information*, Centro Nacional de Información de Biotecnología) por medio de una palabra clave, tal y como se ve en la Figura 4.26.

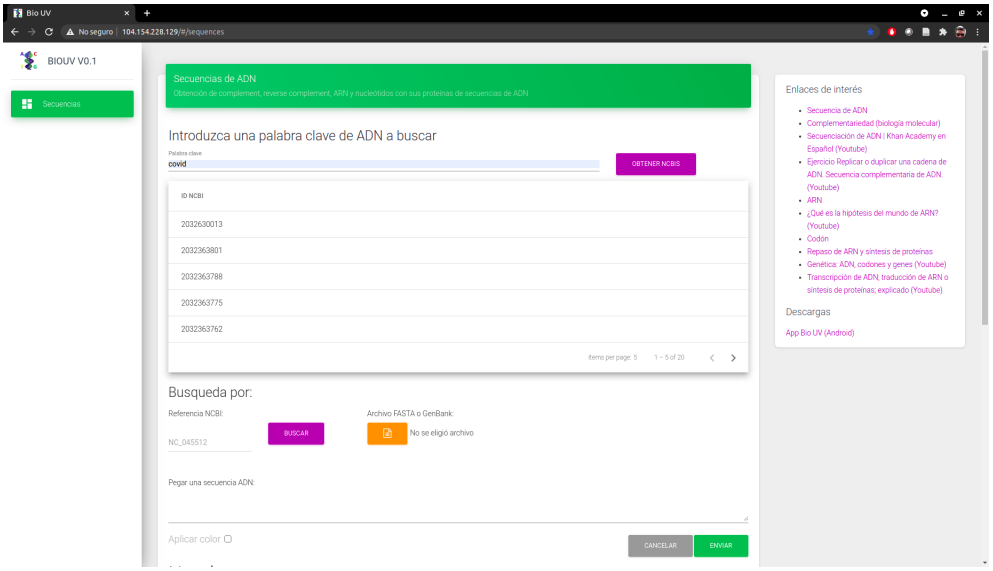

Figura 4.26: Consulta de identificadores de *NCBI* por palabra clave.

Al seleccionar un identificador de *NCBI* se consulta a través de una *API* la información del nucleótido, se obtiene el nombre y la cadena de ADN (Figura 4.27).

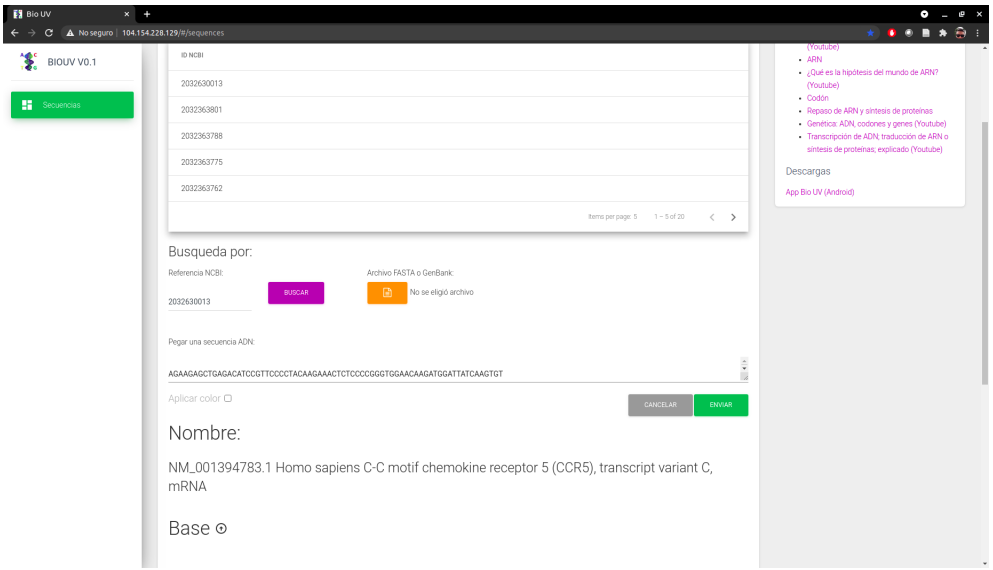

Figura 4.27: Obtención de información de ADN por listado de identificadores de *NCBI*.

También se puede introducir directamente el identificador de *NCBI* en caso de saberlo, se introduce en la caja de texto y se da clic en "BUSCAR" (Figura 4.28).

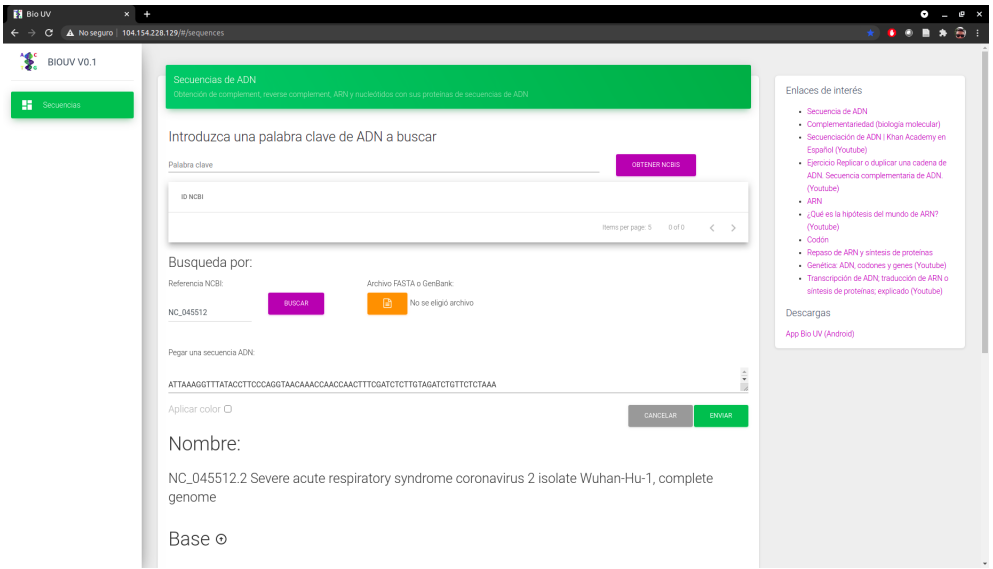

Figura 4.28: Búsqueda manual por identificador de *NCBI*.

Una vez que se obtuvo la cadena de ADN, se da clic en "ENVIAR" para realizar el análisis de la secuencia de ADN del cual obtendrá los valores de *Base*, *Complement*, *Reverse Complement*, *ARN* y sus respectivos nucleótidos y proteínas (Figura 4.29).

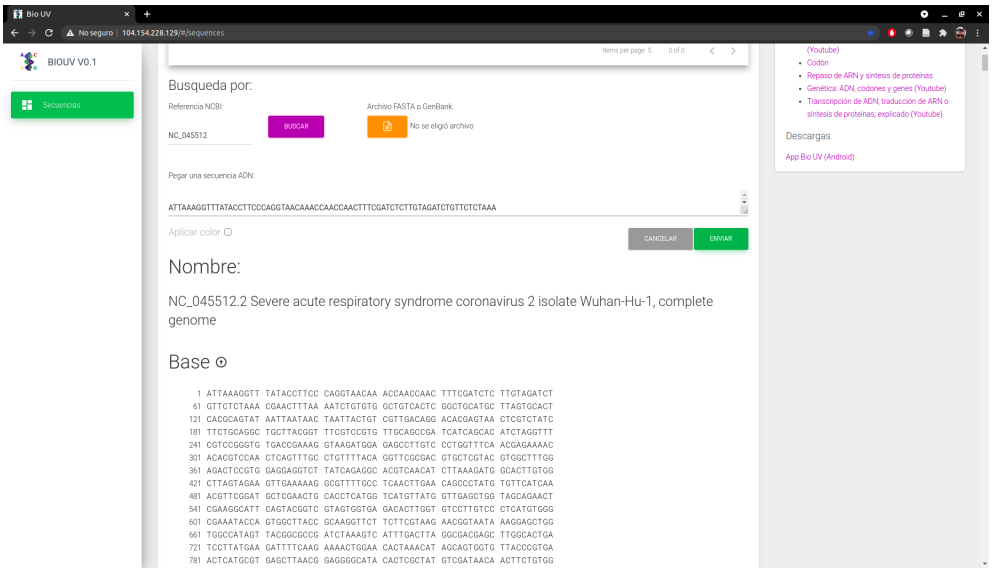

Figura 4.29: Enviar cadena de ADN para su análisis.

También si se desea se puede aplicar color a los resultados de análisis de la secuencia de ADN, esto se realiza activando la casilla de "Aplicar color". Estos colores se aplican con base al consenso de identificación de valores en la secuencia de ADN (Figura 4.30).

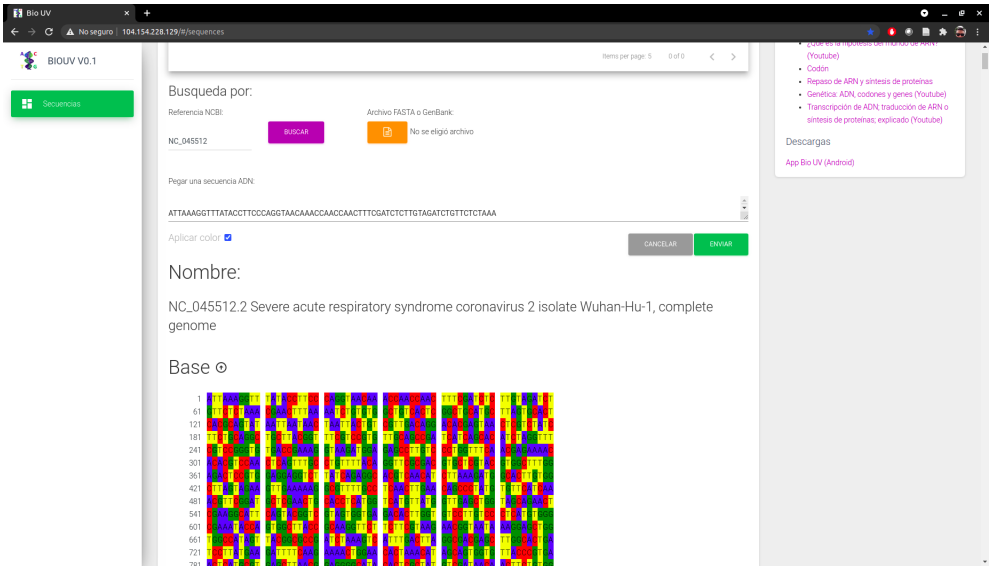

Figura 4.30: Aplicar color a resultados obtenidos.

Como resultado del análisis se obtienen los valores de *Base* (Figura 4.29), *Complement* (Figura 4.31), *Reverse Complement* (Figura 4.32), *ARN* (Figura 4.33), y los respectivos valores de nucleótidos y proteínas (Figura 4.34).

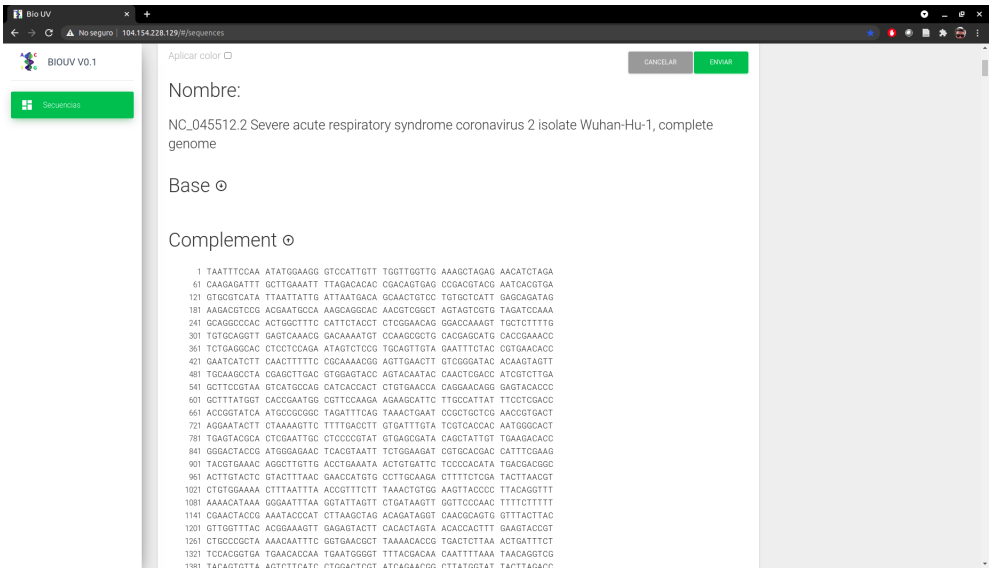

Figura 4.31: Resultado *Complement* del análisis de la secuencia de ADN.

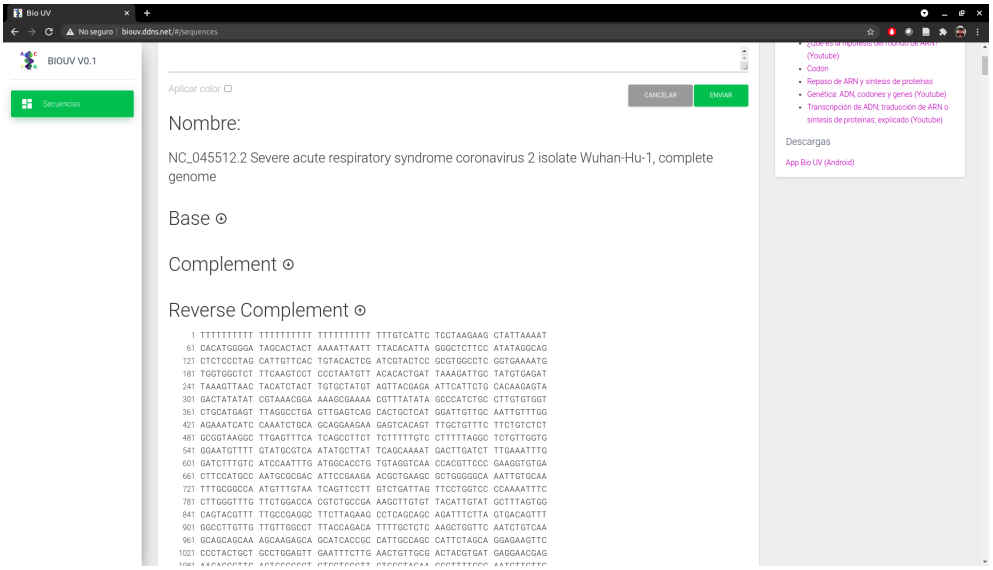

Figura 4.32: Resultado *Reverse Complement* del análisis de la secuencia de ADN.

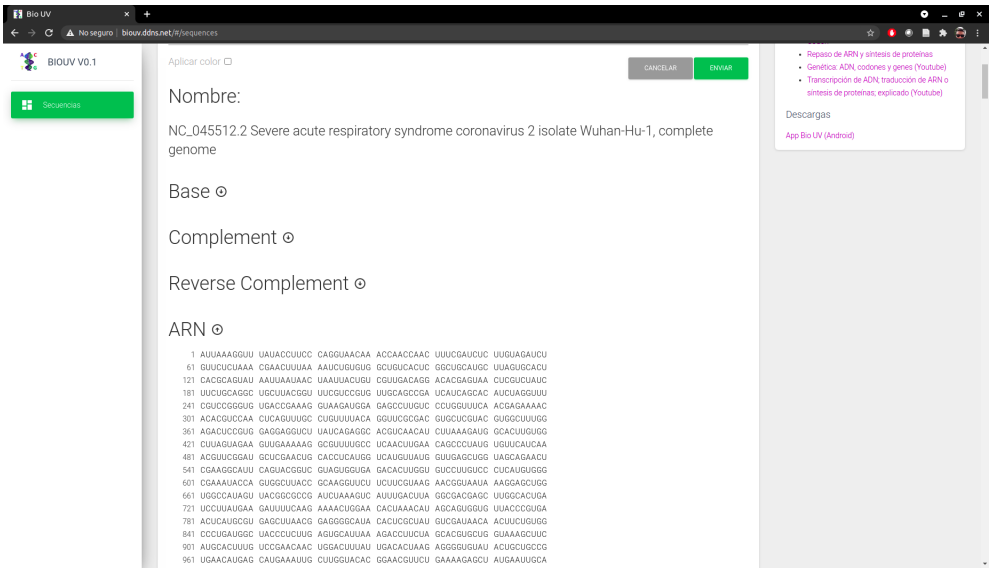

Figura 4.33: Resultado *ARN* del análisis de la secuencia de ADN.

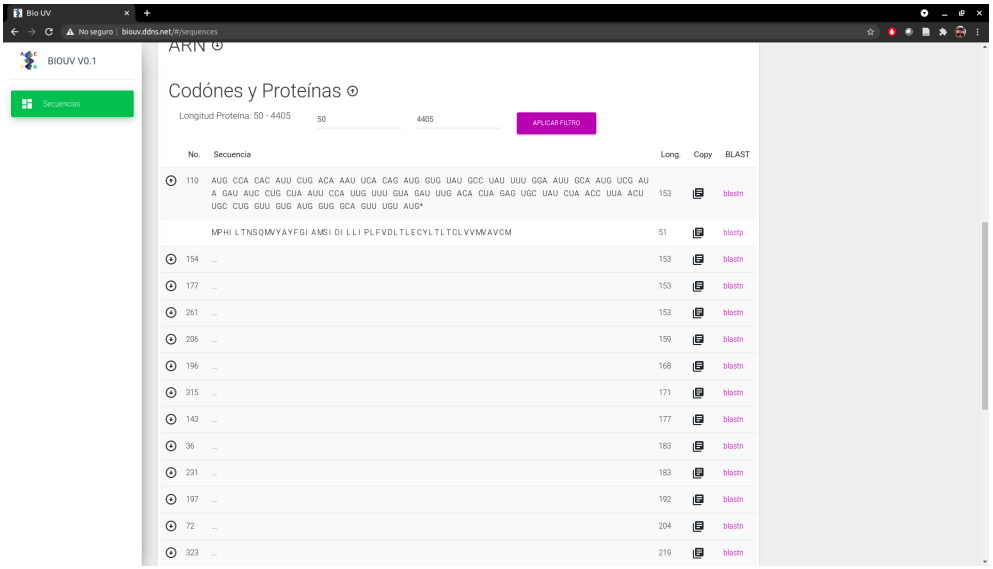

Figura 4.34: Resultado de nucleótidos y proteínas del análisis de la secuencia de ADN.

Del lado derecho, la aplicación cuenta con un conjunto de enlaces de interés que sirven de apoyo para el usuario para entender mejor el análisis de las secuencias de ADN, y un enlace para la descarga de la aplicación "BioGraal" en Android (Figura 4.35).

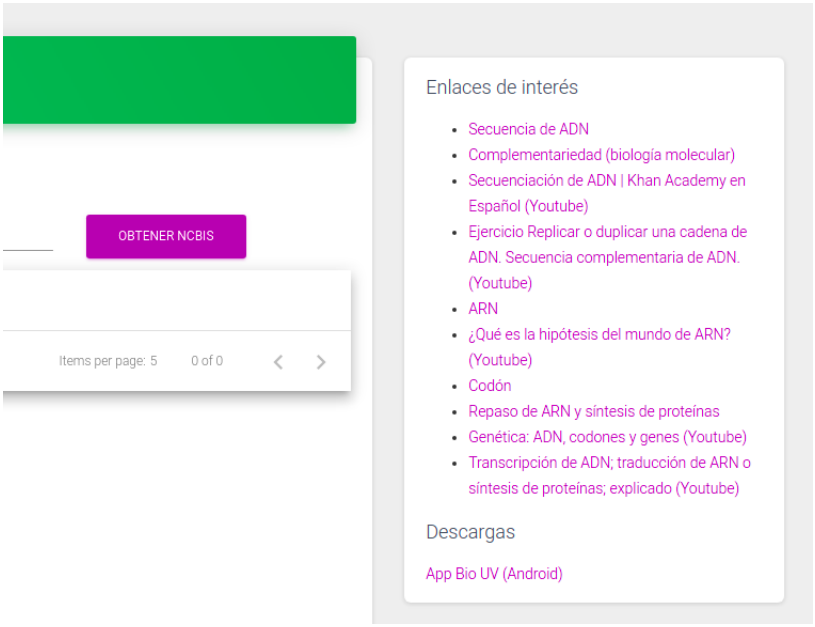

Figura 4.35: Conjunto de enlaces de interés.

4.5.3. BioGraal Android

En esta sección se presenta el proceso de migración de la aplicación *web* en *Angular* "BioGraal" hacia una aplicación *Android*, se presentan los elementos utilizados y los procesos ejecutados para generar la aplicación móvil.

Aplicación de SCRUM

En la migración de la aplicacion *web* "BioGraal" no se implementó directamente SCRUM, sin embargo, al finalizar cada *sprint* en "BioGraal", se realizaba la migración con el producto de dicho *sprint*, ésto para tener un producto entregable de la versión *Android* de la aplicación "BioGraal".

Migración a Android

El proceso de migración de la aplicación *Angular* a *Android* [52] se indica a continuación.

- **Instalación y configuración de paquetes**. Antes de empezar es necesaria la instalación de unos componentes como lo son *Cordova, Java JDK* (En este caso se usó el *JDK* de *GraalVM* instalado), *Apache Ant, Gradle*, y *Android SDK* para ejecutar la aplicación desde un emulador en *Android*.

  1. **Instalación de *Cordova***. Se necesita el paquete *NPM(Node Package Manager)* para instalar el paquete de *Cordova*. Para ello se ejecutan las siguientes líneas desde una terminal:

     ```
     $ sudo apt-get update
     $ sudo apt-get install curl
     $ curl -sL https://deb.nodesource.com/setup_12.x | sudo -E bash -
     $ sudo apt-get install nodejs
     $ nodejs -v
     $ sudo apt-get install npm
     $ npm -v
     ```

     Ahora se procede con la instalación de *Cordova* ejecutando la siguiente línea desde una terminal:

     ```
     $ sudo npm install -g cordova
     $ cordova --version
     ```

  2. **Instalación de *Apache Ant***. La instalación se realiza descargando el paquete desde su página oficial en ant.apache.org/bindownload.cgi, después de descargado se descomprime el paquete dentro de una carpeta llamada "ant" la carpeta "home" del sistema, tal y como se muestra en la Figura 4.36.

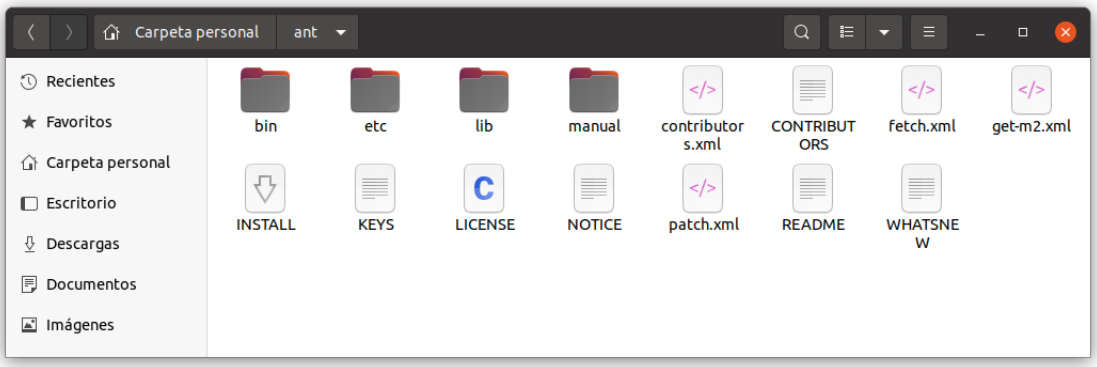

Figura 4.36: Carpeta "ant" dentro de la carpeta "home".

3. **Instalación de *Gradle***. *Apache Ant* y *Gradler* se usan para la ejecución de procesos internos en *Cordova*. La instalación de *Gradle* se realiza ejecutando la siguiente línea desde una terminal:

   ```
   $ sudo apt-get install gradle
   ```

4. **Instalación de *Android SDK***. Se descarga el paquete zip "Command Line Tools" desde https://developer.android.com/studio. Se crea una carpeta llamada "android" en la carpeta "home" (Figura 4.37) y se descomprime el contenido del paquete descargado en esta ubicación.

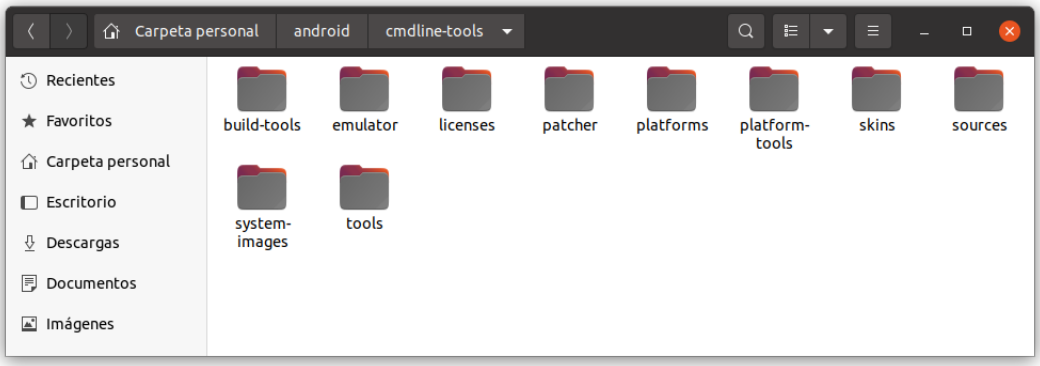

Figura 4.37: Carpeta "android" dentro de la carpeta "home".

5. **Agregar ubicaciones a variables de entorno**. Una vez instalados los componentes anteriormente mencionados, se procede a la configuración de las variables de entorno necesarias para que sean puedan ubicar los recursos de los programas instalados. Primero se ejecuta la siguiente línea desde una terminal:

   ```
   $ sudo gedit ~/.profile
   ```

   Después se procede a agregar las siguientes líneas al final del archivo de texto para cargar las variables de entorno con el valor de las ubicaciones de las carpetas creadas en "home", tal y como se muestra a continuación:

   ```
   #En caso de tener una versión de Java diferente al JDK de GraalVM
   export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
   export PATH=$PATH:/usr/lib/jvm/java-8-openjdk-amd64/bin
   #Variabales de entorno ANT
   ```

```
export ANT_HOME=~/ant/apache-ant-1.10.7
export PATH=$PATH:~/ant/apache-ant-1.10.7/bin
#Variabales de entorno de Android SDK
export ANDROID_SDK_ROOT=~/android/sdk-tools-linux
export PATH=$PATH:~/android/android-sdk-linux/tools/bin
```

Después de la edición, se guardan los cambios y se cierra el archivo. Por último, se procede a reiniciar el equipo parta que los cambios surtan efecto.

6. **Creación de *AVD***. La creacion del *AVD* (*Android Virtual Device*, Dispositivo Virtual de Android) se realiza siguiendo el siguiente proceso:

   a) Se visualizan los paquetes instalados por medio de la siguiente línea desde una terminal:

   ```
   $ sdkmanager --list
   ```

   b) Se procede a actualizar los paquetes instalados:

   ```
   $ sdkmanager --update
   ```

   c) Se instala "build-tools;29.0.0", paquete necesario para construir la aplicación con Cordova:

   ```
   $ sdkmanager "build-tools;29.0.0"
   ```

   d) Se agrega una imagen del sistema seleccionandolo de la lista que se consultó al inicio del proceso:

   ```
   $ sdkmanager "system-images;android-28;google_apis_playstore;x86_64"
   ```

   e) Se crea el emulador de *Android AVD*:

   ```
   $ avdmanager create avd -n test -k "system-images;android-
   28;google_apis_playstore;x86_64" -g "google_apis_playstore"
   ```

f) Se listan los emuladores creados:

```
$ avdmanager list avd
```

La lista se mostraria como en la Figura 4.38:

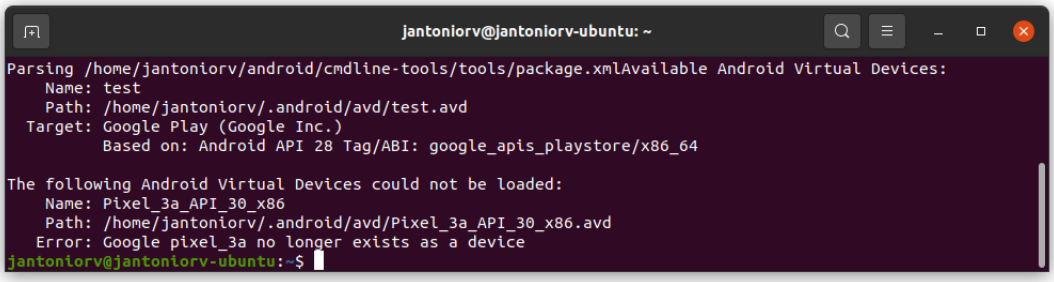

Figura 4.38: Lista de *AVD* instaladas.

Esos serían los componentes a usar para la migración de la aplicacion *web* en *Angular* a *Android*.

- **Configuración del proyecto**. Después de que se hayan instalado los paquetes necesarios se procede a la configuración del proyecto *web* en *Angular* siguiendo los siguientes pasos:

  1. Se crea un nuevo proyecto de *Cordova* ejecutando la siguiente línea de código desde una terminal:

     ```
     $ cordova create test com.example.test NewTest.
     ```

  2. Se agrega la plataforma *Android* al nuevo proyecto *Cordova* creado:

     ```
     $ cd test
     $ cordova platform add android
     ```

  3. Es necesario realizar una combinación de archivos del proyecto *web Angular* con el proyecto *Cordova* generado, lo que se realiza es copiar todo el contenido del proyecto *Cordova* a excepción de los archivos `package.json` y `package-lock.json`, los archivos copiados se pegan dentro del proyecto *web* de

Angular.

4. Después, se procede a modificar el archivo `package.json` del proyecto *web* de *Angular*, debe quedar como se muestra en la Figura 4.39:

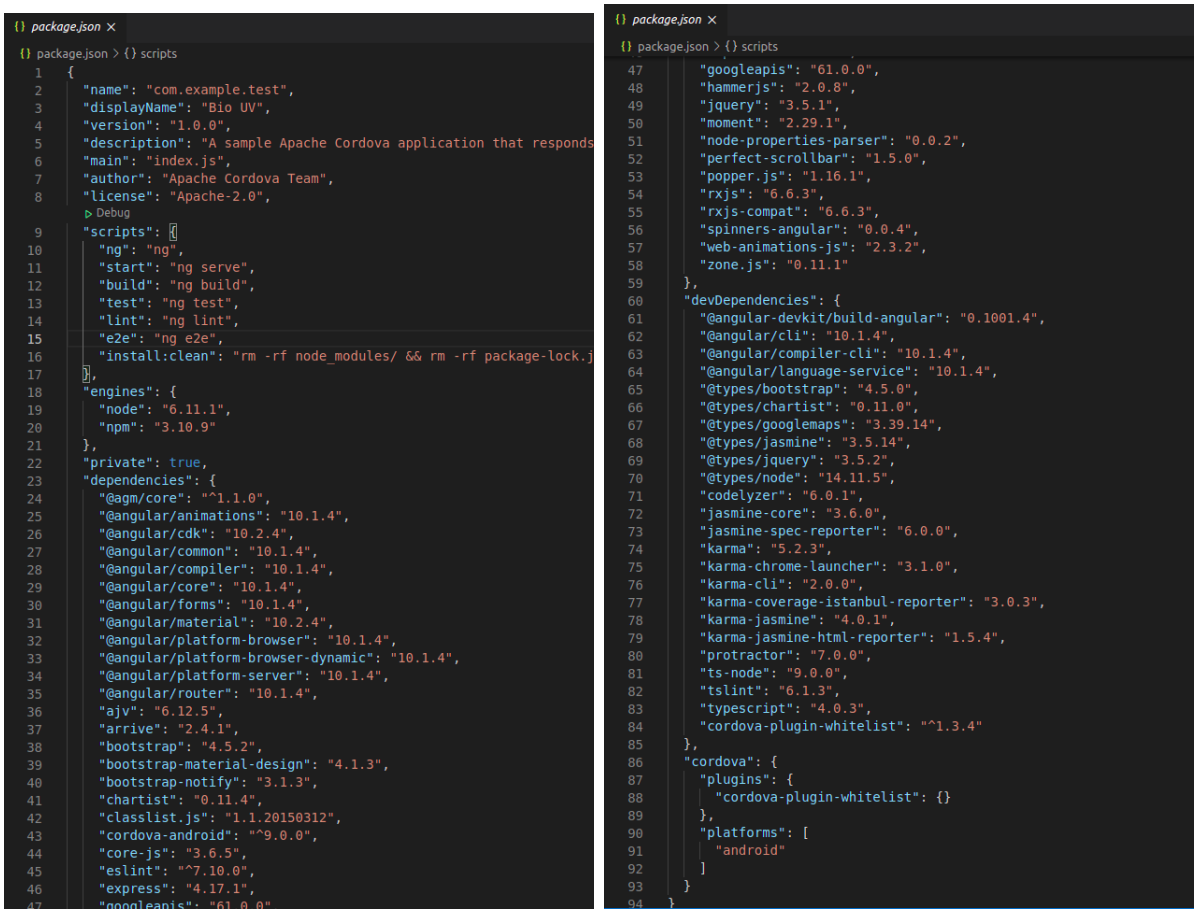

Figura 4.39: Archivo `package-lock.json`

5. Ahora se edita el archivo `src/index.html` cambiando la linea `<base href=‘‘/’’>` a `<base href=‘‘./’’>` tal y como se muestra en la Figura 4.40:

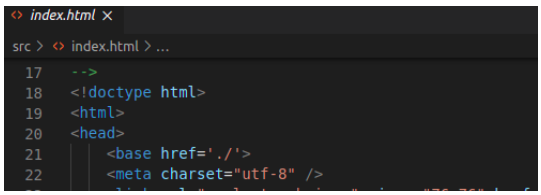

Figura 4.40: Cambio en `src/index.html`

6. También se modifica el archivo `angular.json` cambiando el valor de la propiedad `outputPath` a `www`, tal y como se muestra en la Figura 4.41, esta modificación hará que al momento de hacer el compilado de la aplicación el resultado se almacene dentro de la carpeta `www`:

Figura 4.41: Cambio en `angular.json`

7. Por último se modifica el archivo `tsconfig.json` cambiando el valor de la propiedad `target` a `es5`, tal y como se muestra en la Figura 4.42, este cambio permite que la aplicación *Android* funcione con versiones nuevas de *Android*:

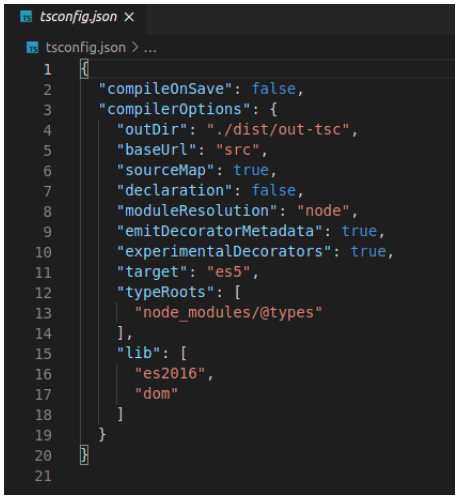

Figura 4.42: Cambio en `tsconfig.json`

Con esto se termina la configuración del proyecto *web* en *Angular* para que se genere la aplicación en *Android*.

- **Ejecución del proyecto en *Android***. Ahora se procede a ejecutar la aplicación *Android* desde el *AVD*, los pasos se muestran a continuación.

  1. Se ejecuta desde una terminal la siguiente línea para compilar el proyecto:

     `$ ng build --prod --aot`

     El resultado de la compilación es el que se muestra en la Figura 4.43:

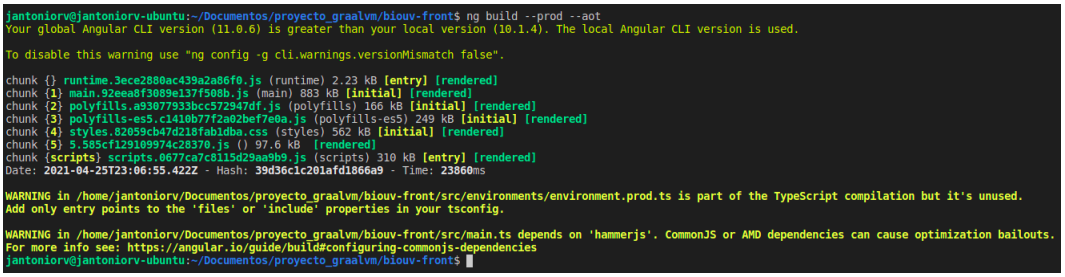

Figura 4.43: Compilado del proyecto en *Android*

  2. Después se procede a ejecutar la aplicación *Android* desde el *AVD* por medio de la siguiente línea desde una terminal:

     `$ cordova emulate android`

     Si la carga del *AVD* se realiza de manera correcta se presentaría la *AVD* con la aplicación ejecutándose como se muestra en la Figura 4.44:

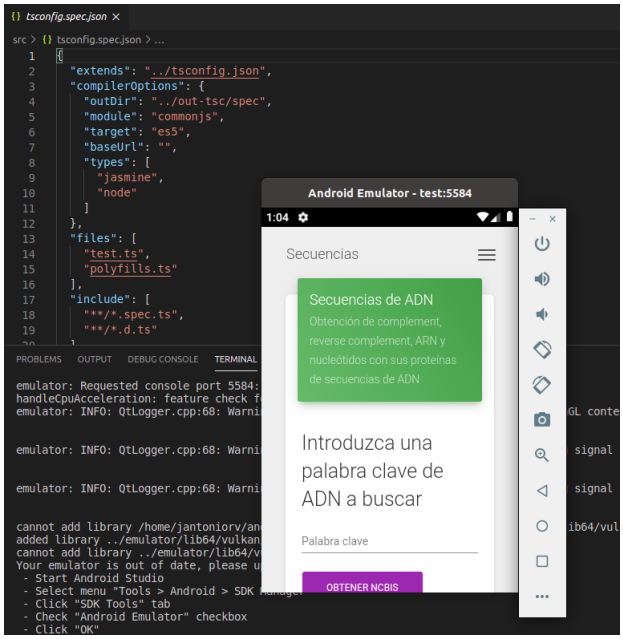

Figura 4.44: Ejecución de la app desde una *AVD*

Uso de la aplicación

En esta sección se presenta el uso de la aplicación "BioGraal" que se migró a la plataforma *Android*. Básicamente es la misma funcionalidad que tiene la aplicación desde la plataforma *web*, solo que ahora se accede desde un dispositivo móvil.

En la Figura 4.45 se ve la aplicación "BioGraal" instalada en un dispositivo móvil, se ingresa a la aplicación y presenta en pantalla el inicio de la aplicación (Figura 4.46).

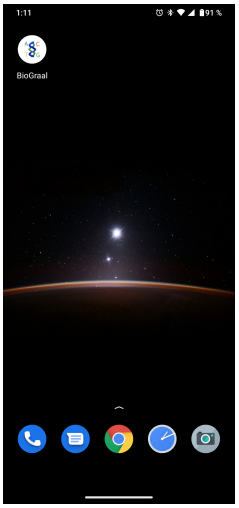

Figura 4.45: Aplicación "BioGraal" desde *Android*.

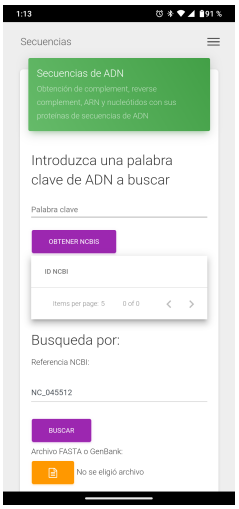

Figura 4.46: Pantalla de inicio de la aplicación "BioGraal".

La funcionalidad, como se mencionó anteriormente sería la misma que su versión de

la plataforma *web* lo que comprende: Consulta de listado de identificadores de *NCBI* por palabra clave (Figura 4.47); Búsqueda por entrada manual del identificador de *NCBI* 4.47); por último, la obtención de los resultados del análisis de la secuencia de ADN, que son: *Base, Complement, Reverse Complement, ARN* y sus respectivos nucleótidos y proteínas (Figura 4.49).

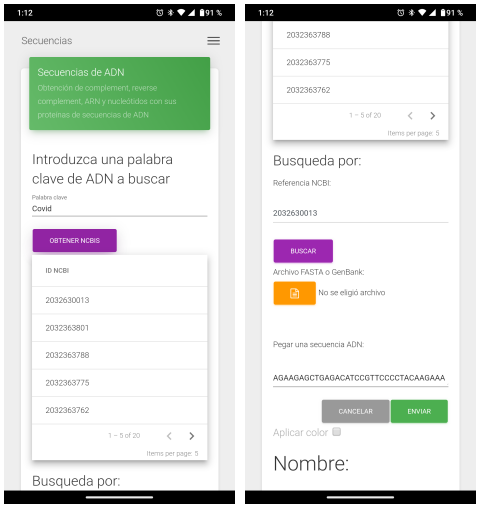

Figura 4.47: Búsqueda por listado de identificadores de *NCBI*

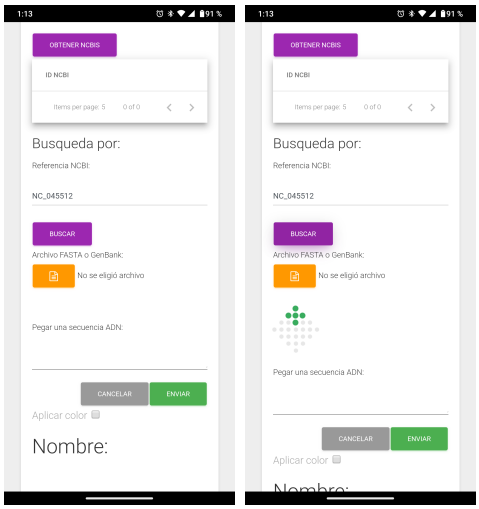

Figura 4.48: Búsqueda NCBI por entrada manual

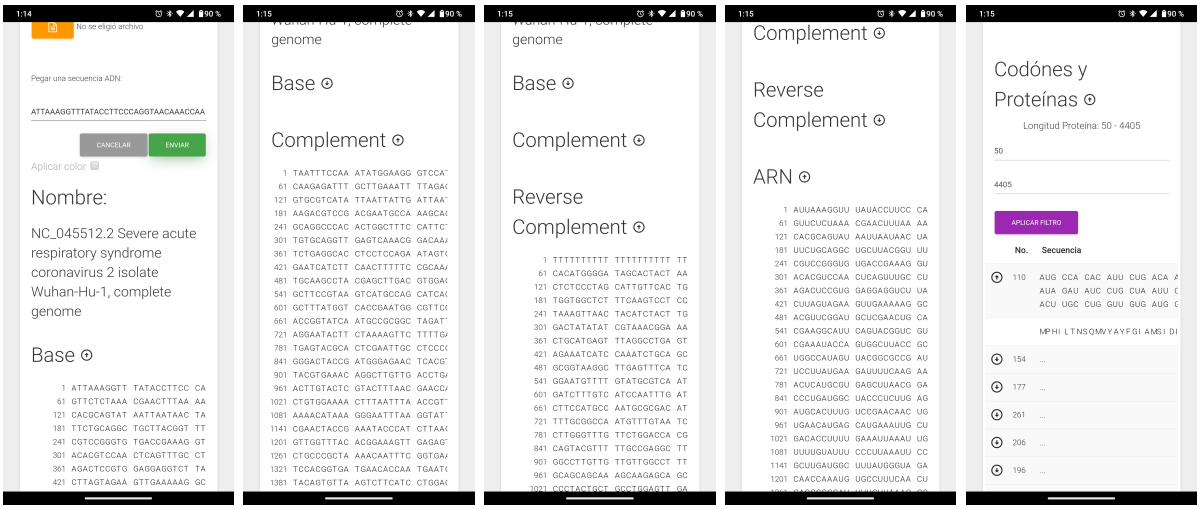

Figura 4.49: Resultados de análisis de secuencia de ADN.

También se puede visualizar la aplicación en modo horizontal (Figura 4.50).

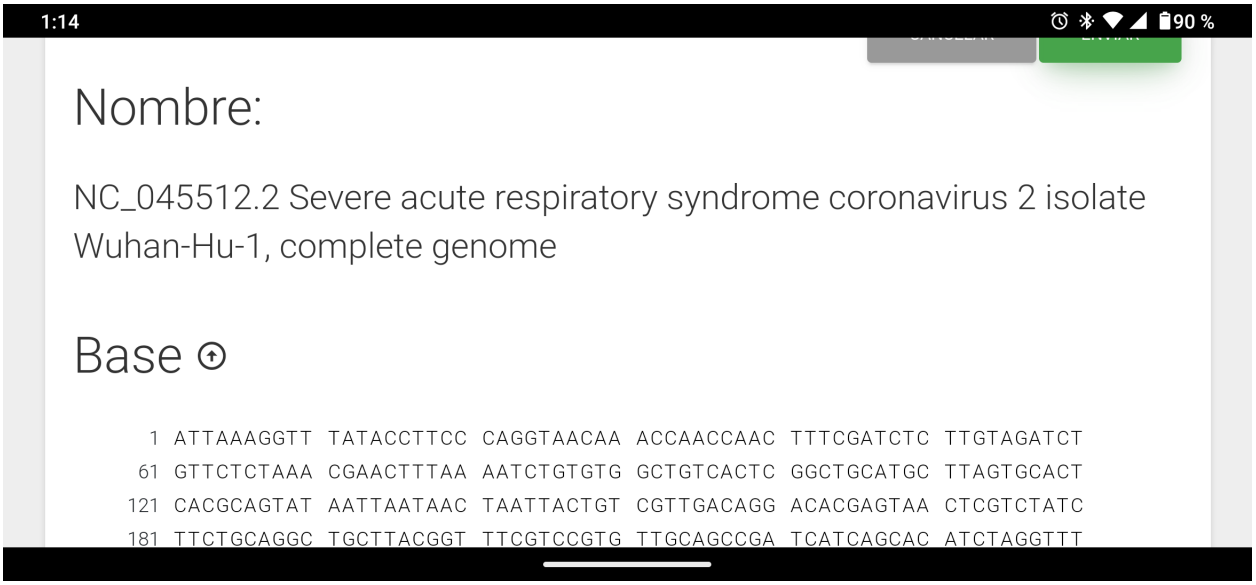

Figura 4.50: Modo horizontal de la aplicación “BioGraal”.

Por último, se presenta la sección del conjunto de enlaces de interés (Figura 4.51) y el menú de la aplicacion “BioGraal” que se encuentra colapsado (Figura 4.52).

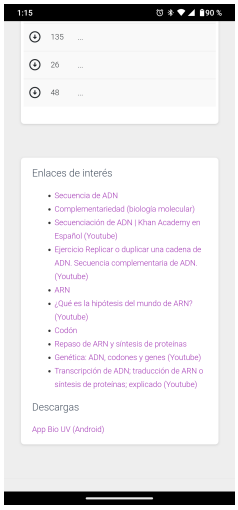

Figura 4.51: Enlaces de interés.

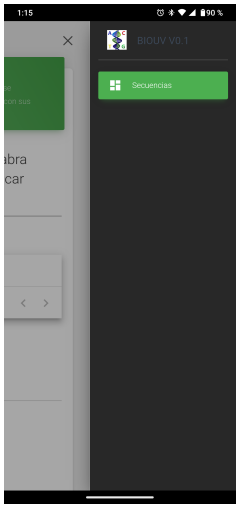

Figura 4.52: Menú de la aplicación "BioGraal".

4.6. Discusión de resultados

En esta sección se mostrarán los resultados obtenidos de las experimentaciones realizadas con la aplicación "BioGraalVM" y "BioGraal". También se visualizan los resultados de desempeño con base en la configuración e implementación de los diferentes entornos y procesos que se llevaron a cabo para desarrollar ambas aplicaciones, y la información obtenida de los procesos realizados por los componentes anteriormente mencionados.

4.6.1. Resultados de comparación desempeño de la aplicación "BioGraalVM" desde *JVM, Native Image* y *Docker*

A continuación, se presentan los resultados obtenidos de la ejecución de la aplicación "BioGraalVM" desde diferentes ambientes como lo es *JVM, Native Image* y *Docker*. Se realizó la ejecución de la aplicación haciendo una llamada *HTTP* de tipo *GET* al servicio *REST* de la aplicación "BioGraalVM" desde una terminal para solicitar el análisis de una secuencia de ADN y obtener el resultado de dicho análisis. El experimento se llevó a cabo un total de 100 veces por cada ambiente, lo que se midió fue principalmente el tiempo de respuesta de cada llamada *HTTP* al servicio de la aplicación "BioGraalVM". Al final

0.047	0.056	0.057	0.058	0.059	0.062	0.064	0.068	0.072	0.092
0.048	0.056	0.057	0.058	0.060	0.062	0.064	0.068	0.072	0.098
0.050	0.056	0.057	0.058	0.060	0.062	0.065	0.068	0.073	0.099
0.055	0.057	0.057	0.058	0.061	0.062	0.065	0.070	0.075	0.101
0.055	0.057	0.057	0.058	0.061	0.062	0.065	0.070	0.077	0.110
0.056	0.057	0.057	0.058	0.061	0.063	0.066	0.070	0.080	0.180
0.056	0.057	0.058	0.059	0.061	0.063	0.066	0.071	0.081	0.191
0.056	0.057	0.058	0.059	0.062	0.063	0.067	0.071	0.083	0.209
0.056	0.057	0.058	0.059	0.062	0.063	0.067	0.071	0.085	0.210
0.056	0.057	0.058	0.059	0.062	0.064	0.068	0.072	0.086	**0.239**

Tabla 4.1: Resultados de la ejecución desde *JVM*.

se obtuvieron un conjunto de datos que se analizaron para obtener los resultados que se mostrarán a continuación.

Resultados de ejecución la aplicación desde *JVM*

Se presentan los resultados de los tiempos de ejecución de la aplicación desde el entorno de *JVM* usando *GraalVM*.

Como se observa en la Tabla 4.1 los valores oscilan entre 0.047 y 0.239 que son el valor más corto y más largo de tiempo de ejecución en segundos respectivamente.

Por lo tanto, se obtiene un tiempo total de ejecución de:

$$\sum_{i=1}^{n} x_i = 7{,}164 segundos$$

Y un tiempo promedio de ejecución de:

$$\frac{\sum_{i=1}^{n} x_i}{n} = 0{,}07164 segundos$$

Resultados de ejecución la aplicación desde *Native Image*

En esta sección se presentan los resultados de los tiempos de ejecución en segundos de la aplicación "BioGraalVM" como una imagen nativa.

0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.008	0.009
0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.008	0.009
0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.008	0.009
0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.008	0.009
0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.009	0.009
0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.007	0.009	0.009
0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.007	0.009	0.009
0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.007	0.009	0.010
0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.007	0.009	0.010
0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.007	0.009	**0.015**

Tabla 4.2: Resultados de la ejecución desde *Native Image*.

Como se visualiza en la Tabla 4.2 los valores oscilan entre 0.006 y 0.015 que son el valor más corto y más largo de tiempo de ejecución en segundos respectivamente.

Por lo tanto, se obtiene un tiempo total de ejecución de:

$$\sum_{i=1}^{n} x_i = 0{,}669000000000001 segundos$$

Y un tiempo promedio de ejecución de:

$$\frac{\sum_{i=1}^{n} x_i}{n} = 0{,}00669 segundos$$

Resultados de ejecución la aplicación desde *Docker*

Por último, se presentan los resultados de los tiempos de ejecución en segundos de la aplicación "BioGraalVM" de una imagen nativa desde un contenedor *Docker*.

La Tabla 4.3 muestra que los valores oscilan entre 0.235 y 0.361 y son el valor más corto y más largo de tiempo de ejecución en segundos respectivamente.

0.235	0.265	0.269	0.270	0.272	0.276	0.280	0.283	0.286	0.298
0.237	0.266	0.269	0.270	0.272	0.277	0.280	0.283	0.286	0.301
0.238	0.266	0.269	0.270	0.272	0.277	0.281	0.283	0.286	0.303
0.239	0.267	0.270	0.271	0.272	0.278	0.281	0.283	0.287	0.309
0.247	0.267	0.270	0.271	0.272	0.278	0.281	0.284	0.287	0.348
0.251	0.267	0.270	0.271	0.273	0.279	0.282	0.284	0.287	0.348
0.264	0.268	0.270	0.271	0.274	0.279	0.282	0.285	0.289	0.350
0.264	0.268	0.270	0.271	0.274	0.279	0.282	0.285	0.292	0.358
0.264	0.268	0.270	0.271	0.275	0.280	0.283	0.285	0.292	0.360
0.265	0.269	0.270	0.271	0.276	0.280	0.283	0.285	0.296	**0.361**

Tabla 4.3: Resultados de la ejecución desde *Docker*.

Por lo tanto, se obtiene un tiempo total de ejecución de:

$$\sum_{i=1}^{n} x_i = 27{,}973 segundos$$

Y un tiempo promedio de ejecución de:

$$\frac{\sum_{i=1}^{n} x_i}{n} = 0{,}27973 segundos$$

Comparación de resultados de desempeño

Finalmente se muestran los siguientes resultados en un cuadro comparativo (Tabla 4.4).

Resultados/Ambiente	***JVM***	***Native Image***	***Docker***
Tiempo total	7.164	0.6690	27.973
Tiempo promedio	0.07164	0.00669	0.27973

Tabla 4.4: Cuadro comparativo de resultados de tiempos de ejecución.

4.6.2. Comparación de resultados de aplicación *web* y aplicación *Android*

A continuación, se presenta la comparación (Tabla 4.5) del entorno *front-end* de la aplicación "BioGraal" entre la aplicación *web* y la aplicación migrada a *Android*.

Tabla 4.5: Análisis comparativo de la aplicación *web* y *Android*

Aplicación *Web*	**Aplicación *Android***
El acceso es preferiblemente desde una computadora. También debido a la propiedad *Responsive* se puede acceder desde un dispositivo móvil.	El acceso es únicamente desde un dispositivo móvil.
La organización de los controles *HTML* es mucho más ordenada y se muestra de mejor manera ya que la pantalla es mucho más grande que la de un dispositivo móvil.	La organización de los controles tiende a lo minimalista, se muestran menos controles o son más pequeños, en su mayoría se encuentran ocultos y sólo se muestran cuando el usuario o alguna acción lo requiera.
El consumo de datos es mucho más grande, las llamadas son en mayor cantidad y con mayor cantidad de elementos enviados y recibidos por llamadas *HTTP*.	El consumo de datos es mínimo, las llamadas son en menor cantidad y los datos a enviar y recibir tiene un formato de menor tamaño, comúnmente de tipo *JSON*.
Los controles *HTML* a usar en este entorno son los controles *HTML* y *HTML5* estándar, de ser necesario alguno diferente se pueden instalar como *plugins* de *Angular*.	Algunos controles *HTML* o *HTML5* no son compatibles con el entorno de *Android*, por lo que es necesario usar unos controles diferentes o realizar alguna configuración especial en la aplicación.
Las llamadas a *API REST* son mucho más fáciles de implementar, y no requieren de validadores de peticiones o tiempos de espera.	En su mayoría, las llamadas a *API REST* implementadas son de tipo asíncronas, contienen validadores de peticiones y no manejan tiempos de espera, a menos que manejen número de intentos de llamada a los servicios.
Continúa en la siguiente página	

Tabla 4.5 Análisis comparativo de la aplicación web y Android

Aplicación *Web*	**Aplicación *Android***
En el caso de acceder a los archivos del equipo por medio de un control tipo `InputFile`, el *browser* no requiere de permisos especiales para tener acceso al equipo, se accede a los archivos que el usuario requiera, siempre y cuando el usuario tenga acceso a estos.	Para tener acceso a los archivos del usuario ya sea en el almacenamiento interno o externo se requiere la implementación de un mecanismo de solicitud y otorgamiento de permisos para que la aplicación tenga acceso a los archivos del dispositivo móvil.
Al momento de acceder a la aplicación *web* "BioGraal", solo es necesario saber la dirección de la página de la aplicación web y es posible acceder desde cualquier equipo con acceso a Internet desde un *browser*.	La aplicación requiere instalarse en el dispositivo móvil, otorgar los permisos necesarios al momento de la instalación, y con sólo ejecutar la aplicación desde el dispositivo móvil se tendrá acceso a las funciones de la aplicación "BioGraal".

Capítulo 5

Conclusión y recomendaciones

El objetivo de este capítulo es mencionar las conclusiones obtenidas con base en los análisis de resultados y el proceso de desarrollo de las aplicaciones "BioGraalVM" y "BioGraal" usando las programación políglota con *GraalVM*. También las respectivas recomendaciones para el manejo de las técnicas y tecnologías usadas en la implementación de "BioGraalVM" y "BioGraal".

5.1. Conclusiones

A través del desarrollo del presente trabajo, usando la programación políglota con la máquina virtual *Graal* permitió descubrir características nuevas en los entornos de desarrollo, haciendo que la brecha del desarrollo clásico de aplicaciones por medio de diferentes entornos debido a la limitante de la falta de comunicación entre los lenguajes de programación, se vaya haciendo mucho más corta. *GraalVM* permite que esta limitante desaparezca en un gran número de casos permitiendo que lenguajes como *Java, Python, C, R, Ruby* y *JavaScript* puedan escribirse en un mismo programa, interpretarse, compilarse y ejecutarse aprovechando las fortalezas de cada uno de los diferentes lenguajes al máximo.

Una de sus características a recalcar del uso de la máquina virtual *Graal* es la propiedad nativa, ya que permite generar una imagen nativa de la aplicación en *Java* y aprovechar

las propiedades *AOT* y *JIT* para hacer que la aplicación sea mucho más rápida que al ejecutar una aplicación desde *JVM*.

Tal y como se pudo ver en la Tabla 4.4 la imagen nativa muestra su superioridad en cuestión de tiempo de ejecución al ejecutar la aplicación "BioGraalVM" en un tiempo promedio de 0.00669 segundos que es sumamente rápido a comparación de lo que fue la ejecución con *JVM* y la imagen nativa dentro de un contenedor *Docker*. Algo similar a lo obtuvo Juan Fumero et al. en [53] al momento de combinar su experimentación con la tecnología *GPU Linux* elevando la velocidad de ejecución a 500 veces.

Con base en esto se pueden mencionar los siguientes puntos:

1. La ejecución de la imagen nativa es mucho más rápida que la aplicación en *Java* usando *JVM* directamente o contenida en un *Docker*.

2. La implementación de la imagen nativa dentro del *Docker* puede ser fácil de realizar, pero al parecer el contenedor *Docker* genera un decorador o *proxy* que vuelve lenta la ejecución o tiempo de respuesta de las peticiones a la imagen nativa de la aplicación.

3. La implementación de los *scripts* de *Python* requirió también una serie de directivas internas en el código en *Python* para que pueda comunicarse el lenguaje *Guest* que sería *Python*, con el lenguaje *Host* que sería *Java* al momento de implementar el entorno políglota.

También el desarrollo de la aplicación *web* "BioGraal" usando *Angular* y haciendo la migración a *Android* muestra que ambas tecnologías tienen ventajas y desventajas una de la otra, tal y como se ve en la Tabla 4.5, por lo cual también se pueden mencionar los siguientes puntos:

1. Debido a la cantidad de controles que se usan y la información que se presenta al

usuario en pantalla, la aplicación *web* se presenta como la mejor opción para acceder al sistema "BioGraal".

2. La aplicación web "BioGraal" se desarrolló con la idea de que sea *Responsive*, por lo cual, también se tiene el acceso *web* desde el dispositivo móvil de manera visiblemente cómoda y fácil de usar para el usuario.

3. El acceso desde la aplicación *Android* se usaría principalmente si se requiere tener una aplicación exclusiva en un punto de inicio del dispositivo móvil o evitar el entrar desde el *browser* y escribir la dirección *HTTP* de la página de la aplicación "BioGraal".

5.2. Recomendaciones

Como recomendaciones se mencionan los siguientes puntos de manera separada en cuanto a la implementación de "BioGraalVM" y "BioGraal".

5.2.1. Recomendaciones "BioGraalVM"

1. El uso del patrón de diseño *Singleton* para la implementación del contexto de *GraalVM*, ya que el contexto de preferencia debe ser un único existente para evitar generar varias instancias del mismo dentro de la imagen nativa y ésta ocupe mucho espacio de memoria en tiempo de ejecución.

2. Al manejar un lenguaje *Guest* dentro de un lenguaje *Host*, si éste contiene muchas líneas, usar de preferencia un *script* externo o usarlo en una clase por separado, ya que el contexto de *GraalVM* permite el uso de líneas de código en el lenguaje *Host* pero como una cadena de texto, por lo que sería complejo o incluso con tendencia a errores de codificación al momento de implementar un entorno políglota. Se recomienda hacerlo tal como se manejó en el actual trabajo al usar los *scripts* de *Python* por separado.

3. A pesar de que los tiempos de ejecución entre la ejecución desde *JVM* y la imagen nativa no varían mucho, se recomienda usar la imagen nativa ya que principalmente es mucho más rápida y también por el uso de *AOT* y *JIT* permite un mejor manejo de los errores en tiempo de ejecución y la seguridad de los datos en la aplicación es mucho mejor al encapsular la aplicación dentro de la imagen nativa.

5.2.2. Recomendaciones "BioGraal"

1. Verificar bien los controles *HTML* que se encuentran ocultos dentro del proyecto *web* en *Angular*, ya que en los dispositivos móviles se presentan algunos controles que en una computadora no se ven, esto por la propiedad *Responsive* del proyecto.

2. Algunas funcionalidades o propiedades de los controles *HTML* no funcionan directamente al migrar a *Android*, por lo cual es necesario probar o investigar antes de implementar en la aplicación y saber cuál sería la mejor manera de implementar la solución que sea existente tanto en la aplicación *web* como en la aplicación *Android* al momento de realizar la migración.

3. En cuanto a la comunicación entre la aplicación de servicios "BioGraalVM" con la aplicación *web* "BioGraal" se realizaría por medio de llamadas *REST*, eso para que se mantenga este tipo de comunicación y al momento de migrar a *Android* se permita el manejo de llamadas *REST* y objetos *JSON*.

4. Al momento de implementar la aplicación de servicios "BioGraalVM" y la aplicación *web* "BioGraal" en la nube, estos tenían diferentes dominios, por lo que fue necesario aplicar la configuración en el proyecto "BioGraalVM" de tipo *CORS Policy* (*Cross-Origin Resource Sharing*, Política de Intercambio de Recursos de Origen Cruzado) para que pudieran comunicarse.

5.3. Trabajo a futuro

En esta sección se mencionan algunos puntos a considerar como trabajo a futuro para complementar el trabajo que se realizó de la presente tesis.

- Realizar análisis, implementación y pruebas de la integración de otro lenguaje de programación en la aplicación "BioGraalVM", ya sea para la mejora del análisis de las secuencias de ADN, o para agregar una nueva funcionalidad. Dichos lenguajes soportados por *GraalVM* a integrar en la aplicación serían *C, C++, R, Ruby, JavaScript o WebAssembly.*

- Explorar las capacidades políglotas y nativas en entornos con otros *frameworks* para mejorar el desempeño de la aplicación "BioGraalVM" e incluso hacer una integración directa con la aplicación *web* "BioGraal".

- Investigar la implementación de un lenguaje aun no soportado por *GraalVM* con el *framework Truffle* para que se pueda integrar en la aplicación "BioGraalVM" y mejore el desempeño de la misma.

- Agregar nuevas funcionalidades a la aplicación "BioGraalVM" y "BioGraal". tales como análisis de proteínas y alineamientos de ADN, aplicando la programación políglota en cada nueva funcionalidad.

- Investigar la integración de "BioGraalVM" y "BioGraal" con otras aplicaciones orientadas a la bioinformática tales como *Biopython*, para la mejora de procesos y de tareas en el análisis de secuencias de ADN.

Apéndice A

Puesta en marcha de *GraalVM*

En esta sección se habla de la instalación, configuración y primeras pruebas realizadas con *GraalVM*. Se explican los pasos para la puesta en marcha de *GraalVM* y los puntos importantes a manejar para usar esta herramienta de la mejor manera posible.

Instalación

Se realiza la descarga por medio del siguiente enlace:

https://github.com/graalvm/graalvm-ce-builds/releases/tag/vm-20.2.0

Se descarga el archivo dependiendo el sistema operativo y la versión de *Java* que se desea trabajar, en este caso se usa la versión 8 de *Java* y se descarga la versión para *Linux* "graalvm-ce-java8-linux-amd64-20.2.0.tar.gz".

Se realiza la descarga y desempaquetado del archivo y se realiza la configuración de la variable `PATH` en *Linux*, junto con el `JAVA_HOME`:

```
$ tar -xzf <graalvm-archive>.tar.gz
$ export PATH=<graalvm>/bin:$PATH
$ export JAVA_HOME=<graalvm>
```

Para confirmar que la instalación se realizó correctamente se verifican las versiones de *Java* de *GraalVM*, tal y como se muestra en la Figura A.1:

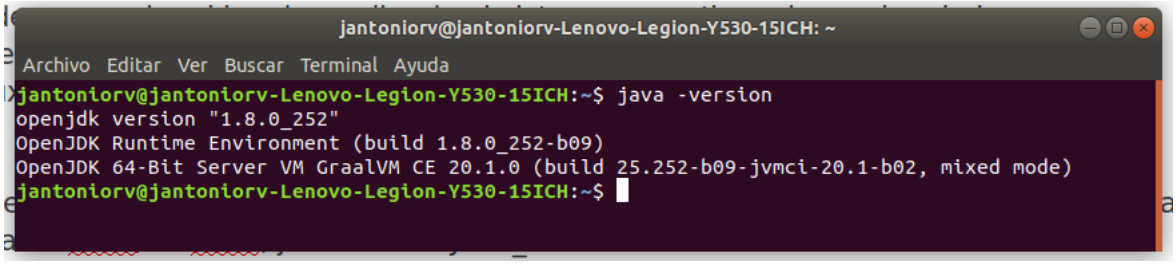

Figura A.1: Versión de *Java* de *GraalVM*

Configuración

Java y *JavaScript* se encuentran en la distribución de *GraalVM*. Faltarían instalar los demás componentes.

Instalación de *LLVM*

Se ejecuta la siguiente línea desde la terminal:

```
$ gu install llvm-toolchain
```

Al terminar de instalar se configura el path de `LLVM_TOOLCHAIN`:

```
$ export LLVM_TOOLCHAIN=$(lli --print-toolchain-path)
```

Verificación de versión de *LLVM* (Figura A.2):

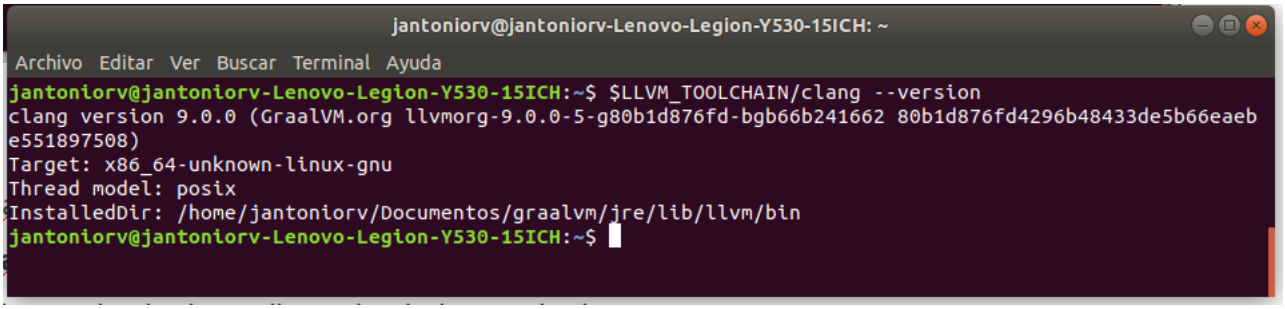

Figura A.2: Información de *LLVM* instalada

Instalación de *Python*

Se ejecuta la siguiente línea desde una terminal:

```
$ gu install python
```

Verificación de versión de *Python* instalada (Figura A.3):

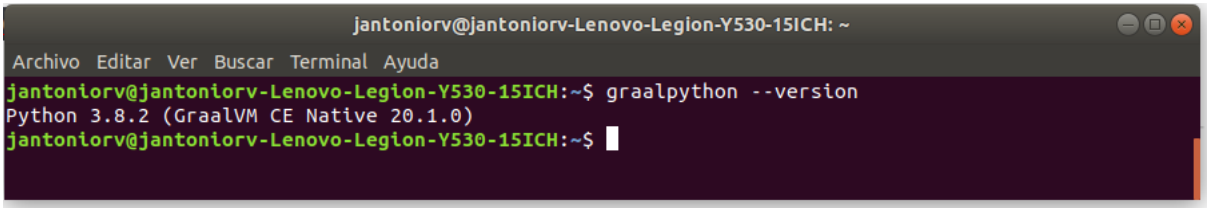

Figura A.3: Información de *Python*

Instalación de *Ruby*

Se ejecuta la siguiente línea desde una terminal:

```
$ gu install ruby
```

Verificación de versión de *Ruby* instalada (Figura A.4):

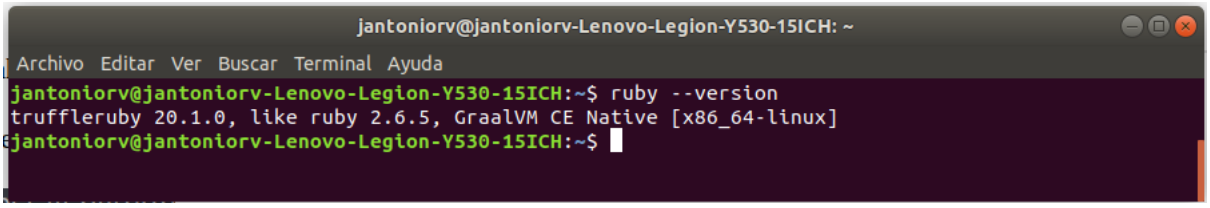

Figura A.4: Información de *Ruby*

Instalación de *R*

Se ejecuta la siguiente línea desde una terminal:

```
$ gu install R
```

Verificación de versión de *R* instalada (Figura A.5):

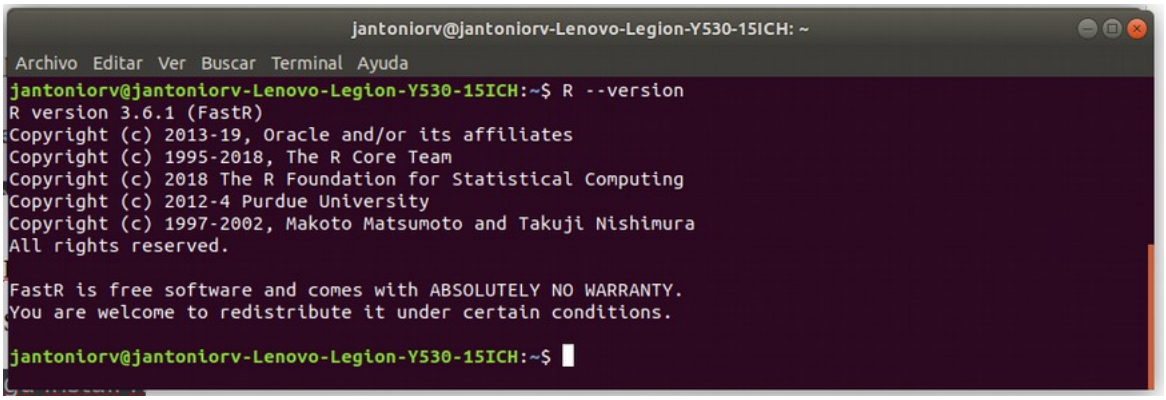

Figura A.5: Información de *R*

Instalación de *WebAssembly*

Se ejecuta la siguiente línea desde una terminal:

```
$ gu install wasm
```

Verificación de versión de *WebAssembly* instalada (Figura A.6):

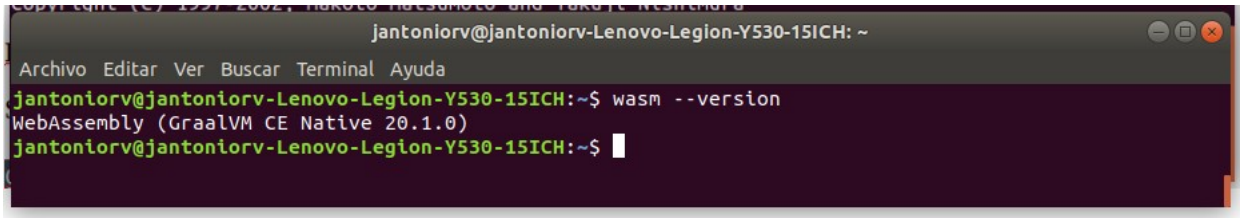

Figura A.6: Información de *WebAssembly*

Instalación de *Native Image*

Se ejecuta la siguiente línea desde una terminal:

```
$ gu install native-image
```

Verificación de versión de *Native Image* instalada (Figura A.7):

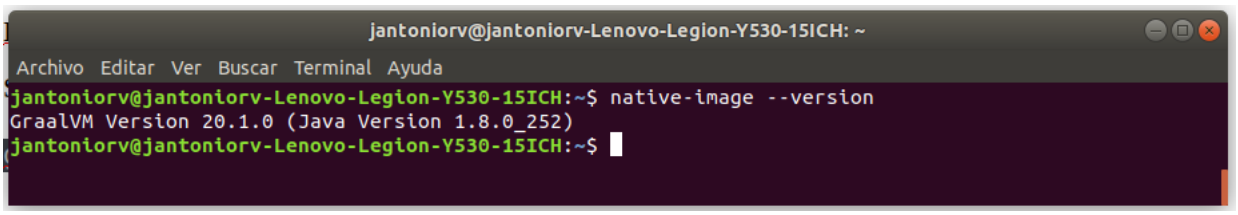

Figura A.7: Información de *Native Image*

Lista de componentes necesarios instalados desde la herramienta *gu* (*GraalVM Updater*) de *GraalVM* (Figura A.8):

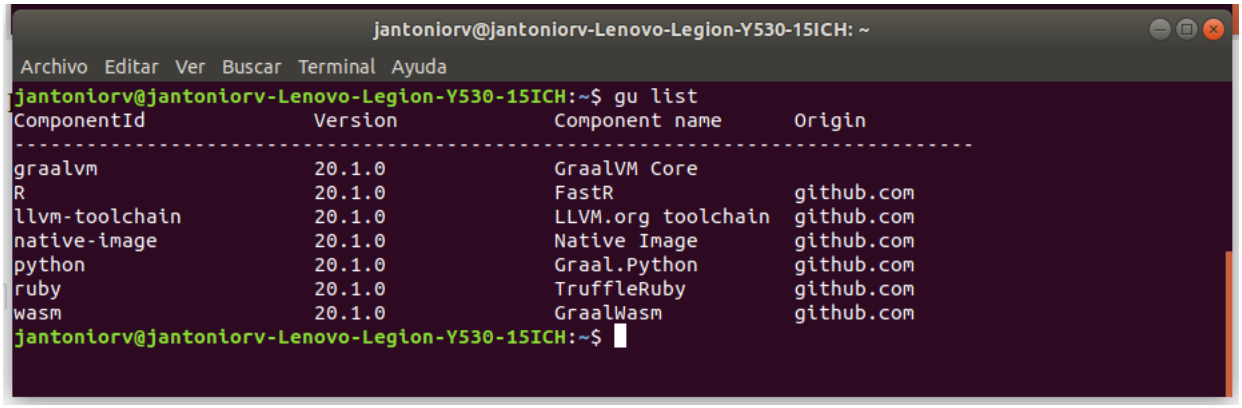

Figura A.8: Lista de componentes

Compilación y ejecución de ejemplos

En esta sección se presentan los códigos escritos, compilados y ejecutados para probar la instalacion de *GraalVM*.

Java

Pruebas de ejecución de un programa escrito en *Java* (Código A.1).

Código A.1: Programa `HelloWorld.java`.

```
1 public class HelloWorld {
2   public static void main(String[] args) {
3     System.out.println("Hello, World!");
4   }
5 }
```

Ejecución desde una terminal (Figura A.9):

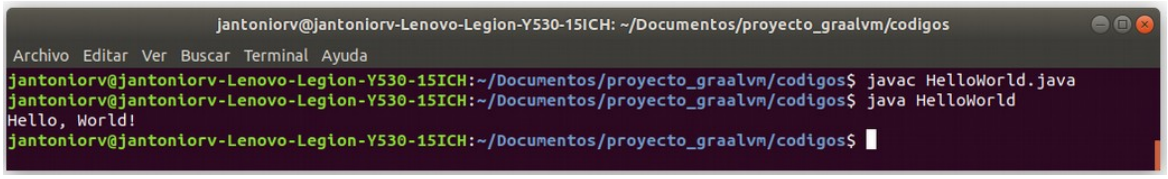

Figura A.9: Ejecución de `HelloWorld.java`

JavaScript

Pruebas de ejecución de un programa escrito en *JavaScript* desde la terminal (Figura A.10) y un *script* `app.js` (Código A.2) en *Node.js*.

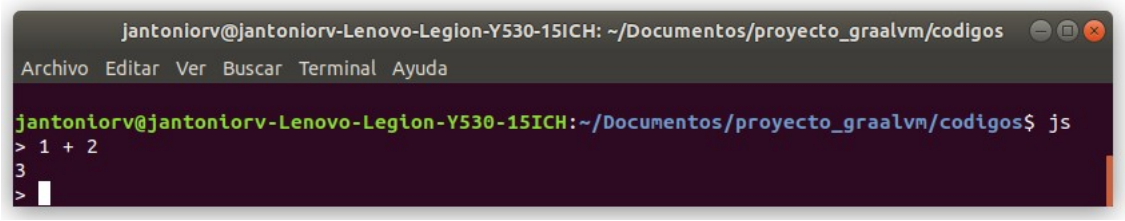

Figura A.10: Ejecución de *JavaScript* con *GraalVM*.

Código A.2: Programa `app.js`.

```
const http = require("http");
const span = require("ansispan");
require("colors");

http.createServer(function(request, response) {
    response.writeHead(200, { "Content-Type": "text/html" });
    response.end(span("Hello Graal.js!".green));
}).listen(8000, function() {
    console.log("Graal.js server running at http://127.0.0.1:8000/".red)
   ;
});
```

Ejecución del *script* `app.js` desde una terminal (Figura A.11) y la presentación del resultado desde un *browser* (Figura A.12).

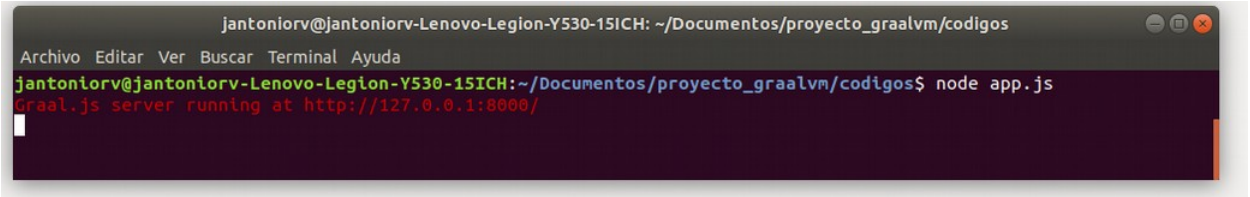

Figura A.11: Ejecución de `app.js` desde una terminal.

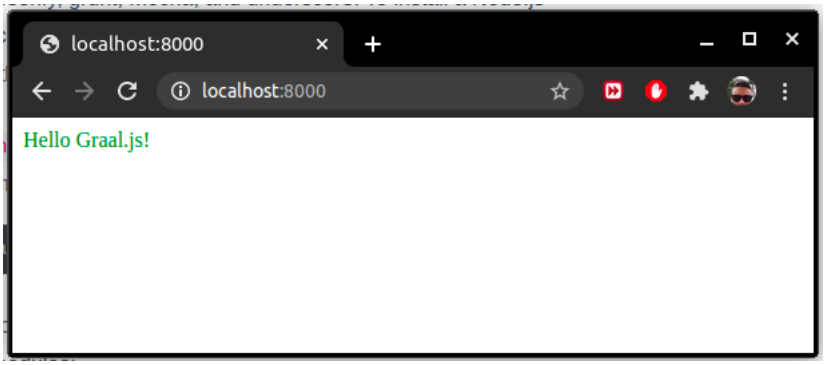

Figura A.12: Resultados de ejecución de `app.js`.

LLVM

Pruebas de ejecución de un programa escrito en *C* (Código A.3) desde una terminal (Figura A.13).

Código A.3: Programa `hello.c`.

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    printf("Hello from GraalVM!\n");
    return 0;
}
```

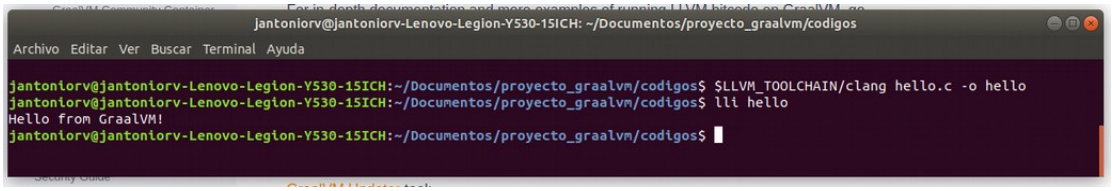

Figura A.13: Resultados de ejecución de `hello.c`.

Python

Pruebas de ejecución desde una terminal en *Python* con *GraalVM* (Figura A.15).

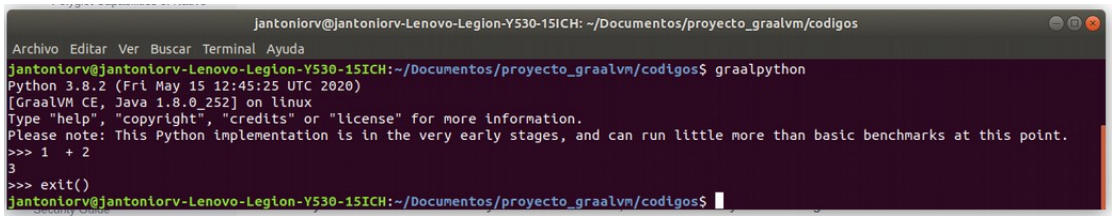

Figura A.14: Ejecucion de *Python* con *GraalVM*.

Ruby

Pruebas de ejecución desde una terminal (Figura A.15) en *Ruby* con *GraalVM*.

Código en terminal:

```
$ gem install chunky_png
$ ruby -r chunky_png -e "puts ChunkyPNG::Color.to_hex(ChunkyPNG::Color('mintcream @ 0.5'))"
```

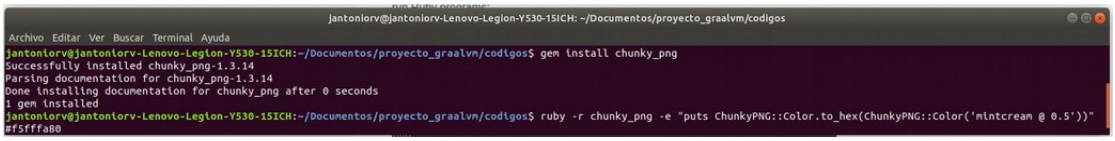

Figura A.15: Ejecución de *Ruby* desde una terminal.

R

Pruebas de ejecución desde una terminal en *R* (Figura A.16).

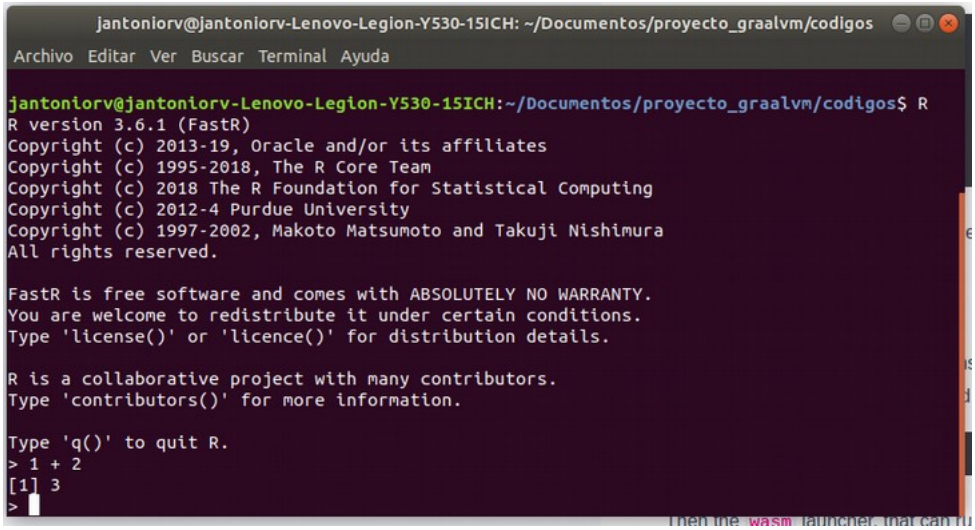

Figura A.16: Ejecución de *R* desde una terminal.

WebAssembly

Pruebas de ejecución de un programa escrito en *C* (Código A.4) y ejecutándolo con *WebAssembly*.

Código A.4: Programa `floyd.c`.

```c
#include <stdio.h>

int main() {
  int number = 1;
  int rows = 10;
  for (int i = 1; i <= rows; i++) {
    for (int j = 1; j <= i; j++) {
      printf("%d ", number);
      ++number;
    }
    printf(".\n");
  }
  return 0;
}
```

Compilado del código `floyd.c` y ejecución del programa `floyd.wasm` (Figura A.17) usando las siguientes líneas desde una terminal:

```
$ emcc -o floyd.wasm floyd.c
$ wasm --Builtins=wasi_snapshot_preview1 floyd.wasm
```

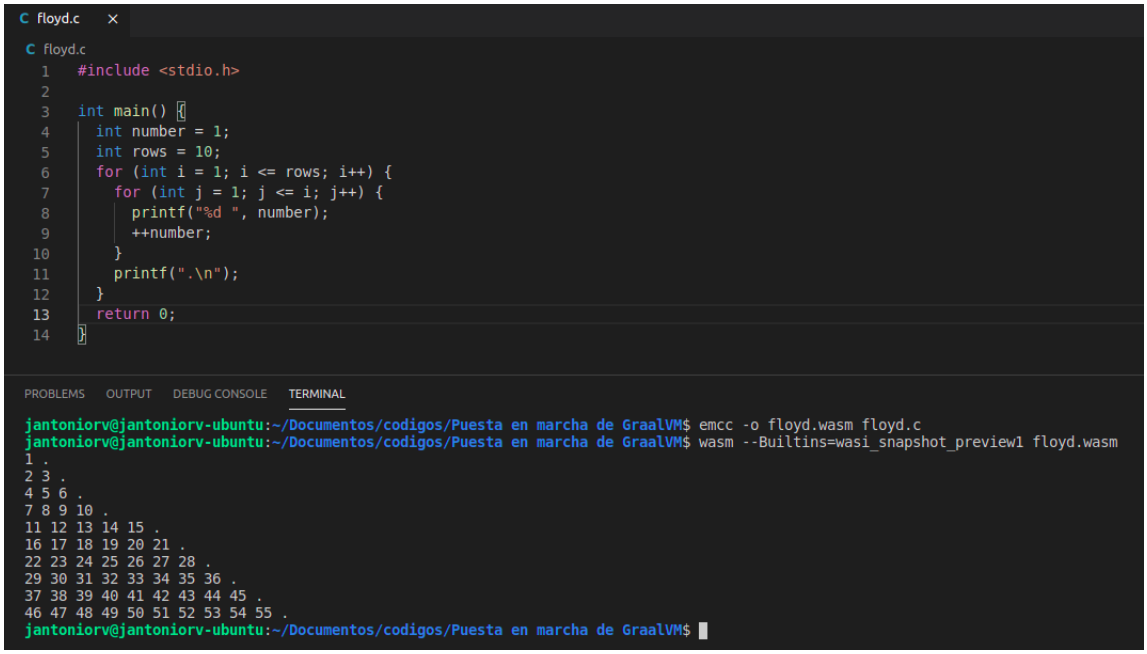

Figura A.17: Ejecución de un programa *WebAssembly* desde una terminal.

Native Image

Pruebas de ejecución de un programa escrito en *Java* (Código A.1) y ejecutándolo como una imagen nativa (Figura A.18).

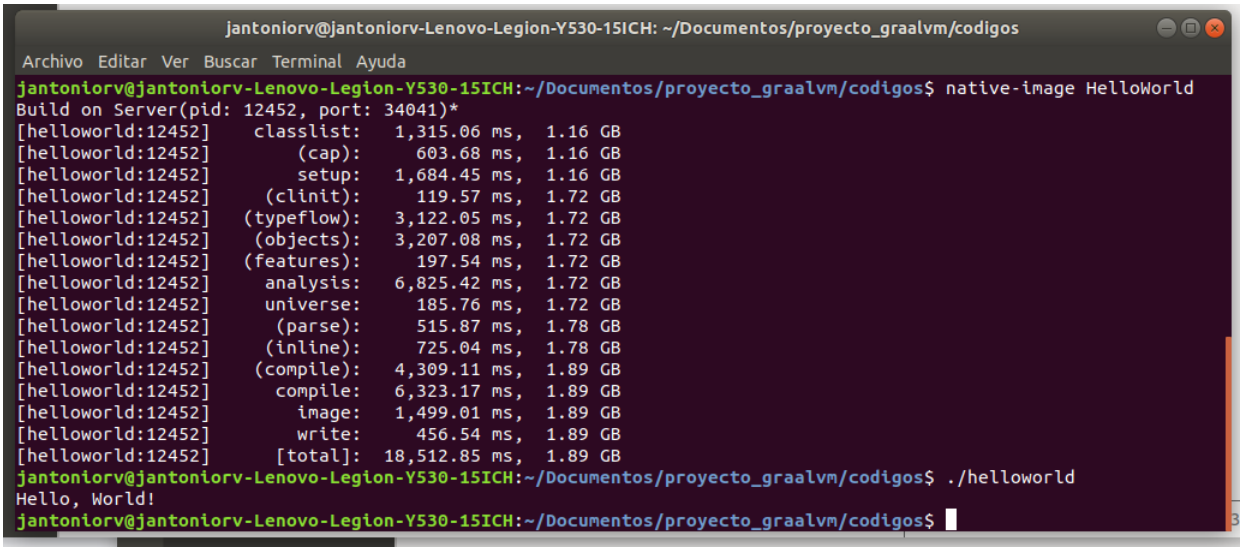

Figura A.18: Creación y ejecución de una imagen nativa desde una terminal.

Native Image con código políglota

Pruebas de ejecución de un programa en *Java* `PrettyPrintJSON.java` escrito con código poliglota (Código A.5) y ejecutándolo como una imagen nativa.

Código A.5: Programa `PrettyPrintJSON.java`.

```
1  import java.io.*;
2  import java.util.stream.*;
3  import org.graalvm.polyglot.*;
4
5  public class PrettyPrintJSON {
6    public static void main(String[] args) throws java.io.IOException {
7      BufferedReader reader = new BufferedReader(new InputStreamReader(
     System.in));
8      String input = reader.lines().collect(Collectors.joining(System.
     lineSeparator()));
9      try (Context context = Context.create("js")) {
10       Value parse = context.eval("js", "JSON.parse");
11       Value stringify = context.eval("js", "JSON.stringify");
12       Value result = stringify.execute(parse.execute(input), null, 2);
13       System.out.println(result.asString());
14     }
```

```
15    }
16 }
```

Creación de la imagen nativa políglota (Figura A.19) usando las siguientes líneas desde una terminal:

```
$ javac PrettyPrintJSON.java
$ native-image --language:js --initialize-at-build-time PrettyPrintJSON
```

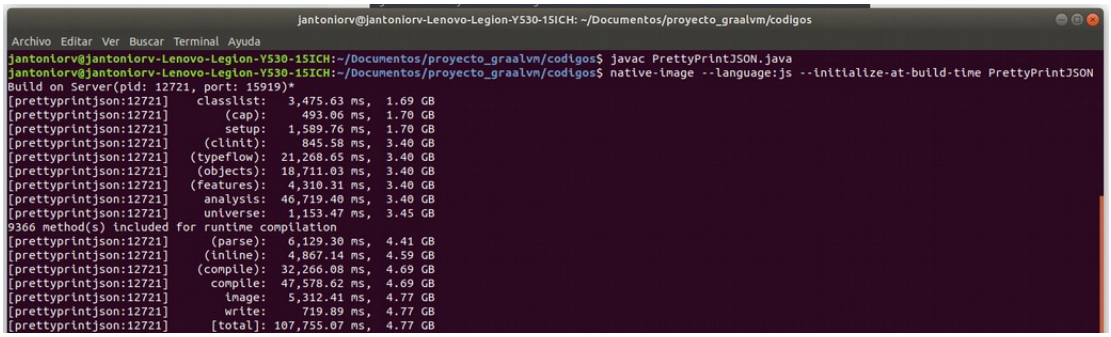

Figura A.19: Creación de una imagen nativa políglota desde una terminal.

Ejecución de la imagen nativa políglota desde una terminal (Figura A.20) usando la siguiente línea:

```
$ ./prettyprintjson <<EOF
{"GraalVM":{"description":"Language Abstraction Platform","supports":["combining
languages","embedding languages","creating native images"],"languages": ["Java",
"JavaScript","Node.js", "Python", "Ruby","R","LLVM"]}}EOF
```

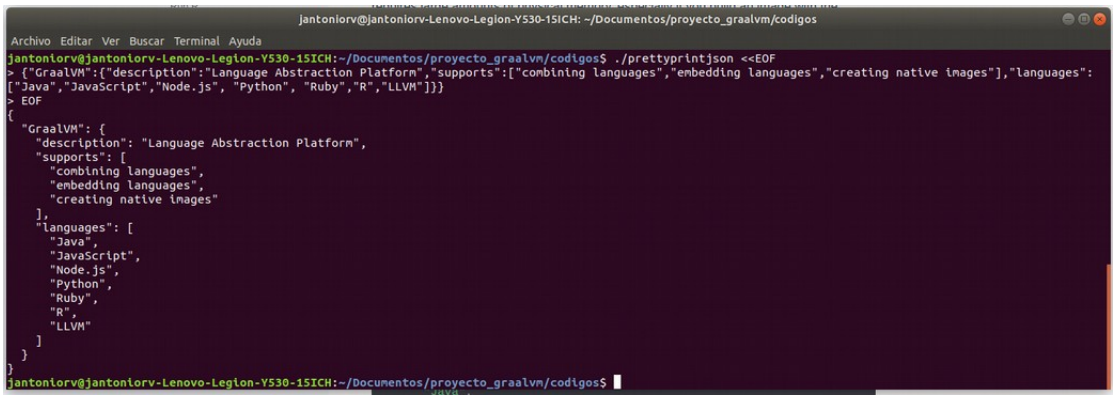

Figura A.20: Ejecución de una imagen nativa políglota desde una terminal.

Productos académicos

En esta sección se presentan los productos académicos que se realizaron durante el desarrollo de la presente tesis.

José Antonio Romero Ventura, Ulises Juárez Martínez, Lisbeth Rodríguez Mazahua, María Antonieta Abud Figueroa, Gustavo Peláez Camarena, “Programación políglota con la máquina virtual Graal”, 20vo Congreso Internacional en Ciencias de la Computación CORE 2020.

Referencias

[1] M. Juganaru Mathieu, *Introducción a la programación*, primera ed. Grupo Editorial Patria, S.A. de C.V.

[2] C. Mateu, *Desarrollo de aplicaciones web*, primera ed. Eureca Media, SL.

[3] O. a. i. a. . GraalVM. [Online]. Available: https://www.graalvm.org/.

[4] Acerca de node.js. [Online]. Available: https://nodejs.org/es/about/

[5] Beginner's guide to python. [Online]. Available: https://wiki.python.org/moin/BeginnersGuide

[6] The java HotSpot performance engine architecture. [Online]. Available: https://www.oracle.com/java/technologies/whitepaper.html

[7] Java help center. [Online]. Available: https://java.com/en/download/help/index.html

[8] Understanding java JIT compilation with JITWatch, part 1. [Online]. Available: https://www.oracle.com/technical-resources/articles/java/architect-evans-pt1.html

[9] HotSpot glossary of terms. [Online]. Available: http://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html

[10] AOT compiler -. [Online]. Available: https://www.eclipse.org/openj9/docs/aot/

[11] GraalVM wasm reference. [Online]. Available: https://www.graalvm.org/reference-manual/wasm/

[12] JVM languages reference. [Online]. Available: https://www.graalvm.org/reference-manual/jvm/

[13] Lenguaje de programación dinámico. [Online]. Available: https://developer.mozilla.org/es/docs/Glossary/Dynamic_programming_language

[14] Oracle. The truffle language implementation framework. [Online]. Available: https://github.com/oracle/graal/tree/master/truffle

[15] F. Niephaus, T. Felgentreff, and R. Hirschfeld, "GraalSqueak: A fast smalltalk bytecode interpreter written in an AST interpreter framework," in *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ser. ICOOOLPS '18. Association for Computing Machinery, pp. 30–35. [Online]. Available: https://doi.org/10.1145/3242947.3242948

[16] AST. [Online]. Available: https://www.eclipse.org/jdt/ui/astview/index.php.

[17] WebAssembly. [Online]. Available: https://webassembly.org/

[18] The LLVM compiler infrastructure. [Online]. Available: https://llvm.org/.

[19] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, "One VM to rule them all," in *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, ser. Onward! 2013. Association for Computing Machinery, pp. 187–204. [Online]. Available: https://doi.org/10.1145/2509578.2509581

[20] L. Stadler, T. Würthinger, and H. Mössenböck, "Partial escape analysis and scalar replacement for java," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. Association for Computing Machinery, pp. 165–174. [Online]. Available: https://doi.org/10.1145/2544137.2544157

[21] M. Brantner, "Modern stored procedures using GraalVM: invited talk," in *Proceedings of The 16th International Symposium on Database Programming Languages*, ser. DBPL '17. Association for Computing Machinery, p. 1. [Online]. Available: https://doi.org/10.1145/3122831.3125717

[22] D. Bonetta, "GraalVM: metaprogramming inside a polyglot system (invited talk)," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection*, ser. META 2018. Association for Computing Machinery, pp. 3–4. [Online]. Available: https://doi.org/10.1145/3281074.3284935

[23] S. Gaikwad, A. Nisbet, and M. Luján, "Performance analysis for languages hosted on the truffle framework," in *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ser. ManLang '18. Association for Computing Machinery, pp. 1–12. [Online]. Available: https://doi.org/10.1145/3237009.3237019

[24] M. Šipek, B. Mihaljević, and A. Radovan, "Exploring aspects of polyglot high-performance virtual machine GraalVM," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 1671–1676, ISSN: 2623-8764.

[25] F. Niephaus, E. Krebs, C. Flach, J. Lincke, and R. Hirschfeld, "PolyJuS: a squeak/smalltalk-based polyglot notebook system for the GraalVM," in *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*, ser. Programming '19. Association for Computing Machinery, pp. 1–6. [Online]. Available: https://doi.org/10.1145/3328433.3328434

[26] S. S. Salim, A. Nisbet, and M. Luján, "Towards a WebAssembly standalone runtime on GraalVM," in *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, ser. SPLASH Companion 2019. Association for Computing Machinery, pp. 15–16. [Online]. Available: https://doi.org/10.1145/3359061.3362780

[27] F. Niephaus, T. Felgentreff, and R. Hirschfeld, "Towards polyglot adapters for the GraalVM," in *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*, ser. Programming '19. Association for Computing Machinery, pp. 1–3. [Online]. Available: https://doi.org/10.1145/3328433.3328458

[28] S. S. Salim, A. Nisbet, and M. Luján, "TruffleWasm: a WebAssembly interpreter on GraalVM," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '20. Association for Computing Machinery, pp. 88–100. [Online]. Available: https://doi.org/10.1145/3381052.3381325

[29] A. Riese, F. Niephaus, T. Felgentreff, and R. Hirschfeld, "User-defined interface mappings for the GraalVM," in *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*, ser. <Programming> '20. Association for Computing Machinery, pp. 19–22. [Online]. Available: https://doi.org/10.1145/3397537.3399577

[30] CSS. [Online]. Available: https://developer.mozilla.org/es/docs/Web/CSS

[31] M. Grimmer, C. Seaton, R. Schatz, T. Würthinger, and H. Mössenböck, "High-performance cross-language interoperability in a multi-language runtime," vol. 51, no. 2, pp. 78–90. [Online]. Available: https://doi.org/10.1145/2936313.2816714

[32] T. Pool, A. R. Gregersen, and V. Vojdani, "Trufflereloader: a low-overhead language-neutral reloader," in *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ser. ICOOOLPS '16. Association for Computing Machinery, pp. 1–10. [Online]. Available: https://doi.org/10.1145/3012408.3012411

[33] T. Pape, T. Felgentreff, F. Niephaus, and R. Hirschfeld, "Let them fail: towards VM built-in behavior that falls back to the program," in *Proceedings of the Conference*

Companion of the 3rd International Conference on Art, Science, and Engineering of Programming, ser. Programming '19. Association for Computing Machinery, pp. 1–7. [Online]. Available: https://doi.org/10.1145/3328433.3338056

[34] A. J. Maas, H. Nazaré, and B. Liblit, "Array length inference for c library bindings," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. Association for Computing Machinery, pp. 461–471. [Online]. Available: https://doi.org/10.1145/2970276.2970310

[35] S. Marr, B. Daloze, and H. Mössenböck, "Cross-language compiler benchmarking: are we fast yet?" in *Proceedings of the 12th Symposium on Dynamic Languages*, ser. DLS 2016. Association for Computing Machinery, pp. 120–131. [Online]. Available: https://doi.org/10.1145/2989225.2989232

[36] H. Sun, D. Bonetta, C. Humer, and W. Binder, "Efficient dynamic analysis for node.js," in *Proceedings of the 27th International Conference on Compiler Construction*, ser. CC 2018. Association for Computing Machinery, pp. 196–206. [Online]. Available: https://doi.org/10.1145/3178372.3179527

[37] F. Niephaus, T. Felgentreff, and R. Hirschfeld, "GraalSqueak: toward a smalltalk-based tooling platform for polyglot programming," in *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, ser. MPLR 2019. Association for Computing Machinery, pp. 14–26. [Online]. Available: https://doi.org/10.1145/3357390.3361024

[38] C. Wimmer, V. Jovanovic, E. Eckstein, and T. Würthinger, "One compiler: deoptimization to optimized code," in *Proceedings of the 26th International Conference on Compiler Construction*, ser. CC 2017. Association for Computing Machinery, pp. 55–64. [Online]. Available: https://doi.org/10.1145/3033019.3033025

[39] Ácido desoxirribonucleico (ADN). [Online]. Available: https://www.genome.gov/es/about-genomics/fact-sheets/acido-desoxirribonucleico

[40] A. Morán. ADN, genes, cromosomas.... Archive Location: World Last Modified: 2018-05-28T13:39:35+00:00 Publisher: https://www.dciencia.es/. [Online]. Available: https://www.dciencia.es/adn-genes-cromosomas/

[41] BLAST TOPICS. [Online]. Available: https://blast.ncbi.nlm.nih.gov/Blast.cgi?CMD=Web&PAGE_TYPE=BlastDocs&DOC_TYPE=BlastHelp

[42] GenBank (.gb, .gbk)—wolfram language documentation. [Online]. Available: https://reference.wolfram.com/language/ref/format/GenBank.html

[43] I. Sommerville, *Ingeniería de Software*, 9th ed. PEARSON.

[44] Build software better, together. [Online]. Available: https://github.com

[45] Micronaut. [Online]. Available: https://docs.micronaut.io/latest/guide/index.html

[46] Native image. [Online]. Available: https://www.graalvm.org/reference-manual/native-image/

[47] Home - SDKMAN! the software development kit manager. [Online]. Available: https://sdkman.io/

[48] JDK distributions - SDKMAN! the software development kit manager. [Online]. Available: https://sdkman.io/jdks

[49] Empowering app development for developers | docker. [Online]. Available: https://www.docker.com/

[50] Angular - introduction to the angular docs. [Online]. Available: https://angular.io/docs

[51] Angular - what is angular? [Online]. Available: https://angular.io/guide/what-is-angular

[52] A. Pandhe. Create a mobile app using your existing angular web project (using cordova). [Online]. Available: https://medium.com/analytics-vidhya/create-a-mobile-app-using-your-existing-angular-web-project-using-cordova-9c10d377d527

[53] J. J. Fumero, T. Remmelg, M. Steuwer, and C. Dubach, "Runtime code generation and data management for heterogeneous computing in java," in *Proceedings of the Principles and Practices of Programming on The Java Platform*, ser. PPPJ '15. Association for Computing Machinery, pp. 16–26. [Online]. Available: https://doi.org/10.1145/2807426.2807428