UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Implementación de un intérprete para un modelo concurrente para el análisis de sistemas síncronos

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Carrascosa Beltrán, Ángel

Tutor/a: Villanueva García, Alicia

CURSO ACADÉMICO: 2021/2022

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Implementación de un intérprete para un modelo concurrente para el análisis de sistemas síncronos

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Ángel Carrascosa Beltrán

Tutor: Alicia Villanueva García

Curso: 2021-2022

Resumen

Actualmente, el estudio del comportamiento concurrente de los sistemas es fundamental para poder disponer de herramientas que faciliten el análisis y verificaciones de programas con este tipo de comportamiento. Por ello, la existencia de lenguajes que modelan estos mecanismos concurrentes son de gran utilidad.

El presente proyecto abarca el diseño e implementación de un intérprete que permite la ejecución de un modelo abstracto para lenguajes concurrentes síncronos. Se trata de un modelo síncrono que puede verse como un lenguaje intermedio al que se pueden traducir otros lenguajes concurrentes síncronos usados en la industria como pueden ser Promela, Esterel, Signal, etc. El intérprete desarrollado analiza un programa que siga la especificación semántica del modelo, ciñéndose al caso de los lenguajes imperativos, y simula su comportamiento. El resultado sirve de *framework* para la verificación formal de estos programas mediante el uso de técnicas preexistentes de análisis en tiempo de ejecución y permite su extensión para dar soporte a la simulación de programas aproximados por interpretación abstracta.

Palabras clave: Concurrencia, Análisis, Semántica operacional, Intérpretes, Compiladores

Resum

Actualment, l'estudi del comportament concurrent dels sistemes és fonamental per a poder disposar d'eines que faciliten l'anàlisi i verificacions de programes amb aquesta mena de comportament. Per això, l'existència de llenguatges que modelen aquests mecanismes concurrents són de gran utilitat.

El present projecte abasta el disseny i implementació d'un intèrpret que permet l'execució d'un model abstracte per a llenguatges concurrents síncrons. Es tracta d'un model síncron que pot veure's com un llenguatge intermedi al qual es poden traduir altres llenguatges concurrents síncrons usats en la indústria com poden ser Promela, Esterel, Signal, etc. L'intèrpret desenvolupat analitza un programa que seguisca l'especificació semàntica del model, cenyint-se al cas dels llenguatges imperatius, i simula el seu comportament. El resultat serveix de *framework* per a la verificació formal d'aquests programes mitjançant l'ús de tècniques preexistents d'anàlisis en temps d'execució i permet la seua extensió per a donar suport a la simulació de programes aproximats per interpretació abstracta.

Paraules clau: Concurrencia, Anàlisis, Semàntica operacional, Intérpretes, Compiladors

Abstract

Currently, the study of the behavior of concurrent systems is essential to be able to have tools that facilitate the analysis and verification of programs with this kind of behavior. For this reason, the existence of languages that model these concurrent mechanisms are of great usefulness.

The present project covers the design and implementation of an interpreter that allows the execution of an abstract model for synchronous concurrent languages. This synchronous model can be seen as an intermediate language to which other synchronous concurrent languages used in the industry can be translated to, such as Promela, Esterel, Signal, etc. The developed interpreter analyzes a program that follows the semantic specification of the model, sticking to the case of imperative languages, and simulates its behavior. The result serves as a framework for the formal verification of these programs by using preexisting techniques of analysis at runtime and allows its extension to support the simulation of approximate programs by abstract interpretation.

Keywords: Concurrency, Analysis, Operational semantics, Interpreter, Compilers

Índice

Anexos

Índice de figuras

Índice de tablas

1. Introducción

1.1 Motivación

Es de vital importancia poder verificar el comportamiento correcto de los sistemas informáticos, ya que el mal funcionamiento de los mismos puede tener graves consecuencias [1]. Hay casos conocidos de proyectos que han supuesto grandes perdidas económicas [2], e incluso han puesto en riesgo la salud de las personas [3].

Hay diversas técnicas de verificación del software. Aquellas basadas en métodos formales presentan importantes ventajas. Al estar basadas en teorías matemáticas, carecen de ambigüedad y proveen cierto grado de verificación para los programas analizados. Destacan para su uso en programas que no pueden ser probados en un entorno de ejecución real, como puede ser el sistema de control automático de una presa.

Es necesario prestar especial atención a la corrección de los sistemas en entornos concurrentes, ya que no solo se debe verificar el estado general de los procesos del programa, también hay que razonar sobre la interacción entre estos en el tiempo, pues pueden cambiar el comportamiento general del sistema, aun siendo correctos los procesos. Los sistemas concurrentes pueden tener múltiples ejecuciones diferentes, dependiendo de las posibles interacciones de los procesos. En muchos sistemas concurrentes, sobretodo en los paralelos, resulta impredecible que interacciones se darán a cabo en función de la velocidad de los procesadores, es decir son sistemas no-deterministas, lo cual contribuye negativamente al problema de la explosión de estados [4] y justifica la verificación formal de estos sistemas [1].

Para ello, hace falta la aplicación de métodos formales orientados a programas concurrentes que se hagan cargo de los problemas mencionados. Uno de estos es el propuesto en [5], en el que se definen técnicas para obtener un modelo abstracto para programas concurrentes síncronos, en el cual es posible aplicar técnicas de verificación ya existentes.

Debido a la dificultad añadida al analizar los sistemas concurrentes por el entrelazado de hilos, ademas de poder verificar estos modelos, es importante poder disponer de un intérprete con el que poder simular la ejecución del sistema en la fase de desarrollo o incluso para visualizar los contraejemplos encontrados con la verificación. Por ello la implementación de un intérprete que pueda aplicar un modelo general para lenguajes concurrentes es de incontable valor para la industria del software, y facilita el trabajo de los desarrolladores. Un caso de éxito es SPIN [6], que es un intérprete para el lenguaje concurrente Promela y ha sido usado en varios proyectos[7].

La principal motivación del trabajo es en definitiva proporcionar un intérprete como herramienta de apoyo para el modelado y la verificación de sistemas concurrentes.

Además también se pretende dar a conocer el proceso de creación de un intérprete relativamente simple, pero con aplicación real, ya que hay pocos ejemplos académicos que muestren el proceso completo o las dificultades comunes a la hora de su desarrollo, y los ejemplos que se pueden extraer de los lenguajes comúnmente usados son extremadamente complejos.

1.2 Objetivos

El objetivo principal del trabajo, es la implementación de un intérprete para el lenguaje concurrente síncrono del modelo presentado en [5]. La implementación del intérprete debe ser extensible, de forma que se pueda reutilizar para implementar futuras técnicas de verificación. También debe poder parar su ejecución en cualquier paso, permitiendo así la inspección del estado del programa. Como objetivo secundario debe de ser lo mas eficiente posible.

El otro objetivo, es documentar el proceso de creación de un intérprete, explicando brevemente las técnicas elegidas para luego compararlas con las comúnmente usadas. Además se pretende mencionar las diferentes técnicas que existen, y en qué casos suelen ser aplicables.

1.3 Metodología

En este proyecto se va a seguir una metodología de cascada iterativa. Esto consiste en la aplicación de la metodología tradicional en cascada en cada etapa del proyecto. En esta metodología se aplican las fases de análisis, diseño, implementación y verificación secuencialmente, de forma que hay que verificar y concluir una fase antes de continuar con la siguiente. Si surgiese algún tipo de fallo o inconveniente durante algunas de las fases, se tendría que revisar las fases anteriores hasta encontrar el fallo y repetir todo el proceso. Al estar aplicado a etapas del proyecto, en caso de fallo, solo se tendría que repetir las fases de cada etapa.

Cada etapa abarca el desarrollo de una de las partes en las que consiste el proyecto, que implementan una funcionalidad completa del proyecto. Estas etapas son las siguientes:

- Análisis y Diseño de la arquitectura del intérprete
  - Desarrollo librería de I/O para *bytecode* o IR usada por la máquina virtual.
  - Desarrollo del compilador
    - Desarrollo del analizador semántico
    - Desarrollo del analizador sintáctico (gramática)
    - Desarrollo del analizador semántico (acciones)
    - Integración de los analizadores en el compilador
  - Desarrollo de la máquina virtual

- Análisis y Diseño de la arquitectura de la máquina virtual
  - Desarrollo de cada módulo

Se a elegido esta metodología debido a que los requisitos del trabajo han sido dados desde el principio y no esta previsto que cambien, haciendo poco probable tener que repetir todas las fases. También se tienen en cuenta que el diseño e implementación parece complicado y es probable que solo se detecten fallos en el diseño durante las fases de verificación o implementación, es por ello que es mejor dividir el proyecto en partes, minimizando así el trabajo en caso de detectarse algún error y deba repetirse el desarrollo desde la fase de diseño, pues afectaría a una sola parte.

Cabe destacar que, al seguir esta metodología, se llevarán reuniones con el cliente para reportar resultados y problemas al final de cada etapa de desarrollo.

A continuación se nombra los integrantes del proyecto. El cliente que ha encargado este proyecto es el tutor del mismo, Alicia Villanueva. El equipo de desarrollo esta compuesto por una única persona, Ángel Carrascosa.

Se detallan todas las etapas y la planificación temporal de estas en el apartado 3.1.

1.4 Estructura de la memoria

El trabajo se estructura en los siguientes apartados:

1. **Introducción**. En este apartado se presenta la motivación y objetivos de la memoria, además de la estructura de la memoria.
2. **Estado del arte**. En este apartado se pretende hacer una introducción a los lenguajes concurrentes, enfocándose en aquellos usados para la verificación de sistemas concurrentes.
3. **Análisis del problema.** Se detalla el problema a tratar, explicando el lenguaje para el cual se construirá el intérprete.
4. **Diseño**. Se detallará la arquitectura y diseño del intérprete, justificando las decisiones de diseño tomadas. Además se nombrarán las herramientas y tecnologías usadas y se explicará con qué fin se utilizarán.
5. **Implementación**. Se mostrará la implementación general del intérprete. Se explicarán los algoritmos utilizados más relevantes.
6. **Pruebas**. Se explicará que pruebas se han realizado para comprobar el correcto funcionamiento del intérprete.
7. **Conclusiones**. Se mostrarán las conclusiones obtenidas, qué objetivos han sido cumplidos y cuales no, así como los posibles trabajos futuros a desarrollar sobre el presente trabajo.

1.5 Convenciones

- El código fuente embebido en el texto, tales como nombres de variables, clases y otras construcciones sobre las que se trata en el texto, están mostradas con una tipografía *monospace* azul sobre un fondo gris.
- Los bloques de código fuente incrustados, se mostrarán en un cuadro de texto con tipografía *monospace* azul sobre fondo gris. Los números de línea estarán indicados a la izquierda del bloque de texto. Pueden mostrarse con un resaltado de color correspondiente a la sintaxis del lenguaje.
- Las palabras extranjeras estarán en cursiva.
- Las citas textuales externas a la memoria irán entrecomilladas.

2. Estado del Arte

La concurrencia presenta una serie de problemas, tales como el bloqueo mutuo (*deadlocks* y *livelocks*), la inanición o las condiciones de carrera, que por motivos de escalabilidad conviene analizar de forma independiente a la funcionalidad relacionada con aquello qué se computa. Es por ello que para aplicar técnicas de verificación formal a sistemas concurrentes y sistemas reactivos se suele representar el modelo concurrente y de interacción del sistema en un lenguaje concurrente, como Promela[6], Lustre, Signal, tccp, CSP[8], TLA+[9], etc.

Otro escenario a tener en cuenta es la validación de sistemas que no se pueden probar directamente. En estos casos se simula la ejecución de estos sistemas y la interacción entre sus componentes. Para esta tarea destacan los intérpretes de lenguajes concurrentes.

Los lenguajes mencionados obligan a describir el modelo antes de poder llevar a cabo el análisis. Debido a esto, existen herramientas que permiten la simulación directa de programas para su análisis sin necesidad de crear el modelo. Sin embargo, únicamente funcionan para el análisis de un lenguaje específico, como es el caso de ISP[10], que solo funciona para MPI.

El modelo presentado en [5] es capaz de capturar las principales características de los lenguajes concurrentes mencionados. Muchos de ellos disponen de herramientas específicas para verificación, pero solo algunos disponen de intérpretes que permitan la simulación del modelo.

Entre los intérpretes disponibles se encuentran SPIN [6] para el lenguaje Promela, FDR [11] y CSP-i [12] para el lenguaje CSP, o tccpinterpreter [13] para tccp. La característica más importante de estos intérpretes es la habilidad de resolver el no-determinismo presente en los programas concurrentes. Para solventar esta problemática, SPIN proporciona simulación aleatoria, que en el caso de que varias acciones no tengan un orden total, es decir, ocurran a la vez, elige aleatoriamente cuál de las acciones se ejecutará primero. Así mismo, también permite la simulación interactiva, parando la ejecución del programa y dando la opción al usuario de elegir la acción que se llevará a cabo. Estos intérpretes también permiten la suspensión de la simulación en un punto para inspeccionar el estado del programa.

La mayor diferencia entre estos intérpretes radica en las propiedades del lenguaje que interpretan. Algunas de las diferencias más destacables de estos lenguajes son las siguientes:

- **Comunicación entre procesos.** La información que comparten los procesos puede ser de dos tipos: paso de mensajes o memoria compartida.
    - El paso de mensajes se suele realizar mediante el uso de canales. Estos consisten en una cola de mensajes en la que los procesos pueden guardar un valor. Cuando un proceso quiere almacenar un valor para que lo utilice otro, lo inserta en el canal y espera a que este quede vacío. Esto solo

ocurre cuando otro proceso extrae un valor del canal. Por otra parte, si un proceso intenta extraer un valor de un canal vacío, espera a que otro proceso lo útilice y recupera el valor almacenado. Esté mecanismo permite la sincronización entre procesos. Por ejemplo, SPIN emplea este mecanismo de comunicación entre procesos.

    - En el caso de la memoria compartida los procesos comunican un valor a otro mediante la escritura de este en una dirección de memoria compartida en la que ambos pueden leer y escribir. Este es el caso de tccp.

- **Ejecución síncrona o asíncrona**. Dependiendo de la velocidad de ejecución de los procesos, se puede considerar un sistema como síncrono si todos los procesos avanzan en un instante de ejecución, o asíncrono si no todos los procesos avanzan en un instante. SPIN, por ejemplo, proporciona procesos síncronos y asíncronos mediante la construcción "*never"[14],* pero tccp es solamente síncrono.

3. Presentación del problema

La idea de este proyecto surge de la necesidad de continuar el trabajo sobre el modelo abstracto propuesto en [5], en el cual se ha estudiado las diferentes construcciones de los principales lenguajes síncronos y se ha creado un modelo en el que se sintetiza la semántica operacional de estos.

Para comprobar el correcto funcionamiento de los programas analizados bajo este modelo, es necesario disponer de un intérprete que ejecute la especificación de dichos programas. Es misión de este proyecto el desarrollo un intérprete a partir de la sintaxis y semántica operacional propuesta en el modelo, considerando el caso de lenguajes imperativos. A continuación se explica la sintaxis y semántica de dicho modelo.

$Program$	:=	$Decl[Action]$	
$Decl$	:=	$proc(\bar{x})\{Action\}$	
	|	$Decl\ Decl$	
$Action$	:=	**end**	
	|	$a!$ where $\forall i \in I . a_i \in Tell$	*Update*
	|	$Action$ **;** $Action$	*Sequence*
	|	$Action \parallel Action$	*Concurrence*
	|	$\sum_{i \leftarrow I} (a_i? \rightarrow Action_i)$ where $\forall i \in I . a_i \in Ask$	*Global Choice*
	|	$\sum_{i \leftarrow I} (a_i? \rightarrow * Action_i)$ where $\forall i \in I . a_i \in Ask$	*Instantaneous Choice*
	|	**if** a **?** **then** $Action$ **else** $Action$ where $a \in Ask$	*Conditional*
	|	$\exists \bar{x}\ Action$	*Local declaration*
	|	$proc(\bar{v})$	*Procedure call*

Figura 1: Especificación de la sintaxis del lenguaje objetivo

R-S1	$\langle a\,!, s\rangle \rightarrow s\langle \text{end}, \mathit{effect}\langle a, s\rangle\rangle$	
R-S2a	$\dfrac{\langle A, s\rangle \rightarrow s\langle A', s'\rangle}{\langle A; B, s\rangle \rightarrow s\langle A'; B, s'\rangle}$	if $A' \neq \text{end}$
R-S2b	$\dfrac{\langle A, s\rangle \rightarrow s\langle \mathit{end}, s'\rangle}{\langle A; B, s\rangle \rightarrow s\langle B, s'\rangle}$	
R-S3a	$\dfrac{\langle A, s\rangle \rightarrow s\langle A', s'\rangle, \langle B, s\rangle \rightarrow s\langle B', s''\rangle}{\langle A \| B, s\rangle \rightarrow s\langle A' \| B', s' \oplus s''\rangle}$	
R-S3b	$\dfrac{\langle A, s\rangle \rightarrow s\langle A', s'\rangle, \langle A, s\rangle \nrightarrow s}{\langle A \| B, s\rangle \rightarrow s\langle A' \| B, s'\rangle}$	
R-S4	$\langle \sum_{i=1}^{n} (a_i ? \rightarrow A_i), s\rangle \rightarrow s\langle A_j, s\rangle$	if $\mathit{test}(a_j, s)$
R-S5a	$\dfrac{\langle A, s\rangle \rightarrow s\langle A', s'\rangle}{\langle \mathit{if}\ a\ ?\ \mathit{then}\ A\ \mathit{else}\ B, s\rangle \rightarrow s\langle A', s'\rangle}$	if $\mathit{test}(a, s)$
R-S5b	$\dfrac{\langle B, s\rangle \rightarrow s\langle B', s'\rangle}{\langle \mathit{if}\ a\ ?\ \mathit{then}\ A\ \mathit{else}\ B, s\rangle \rightarrow s\langle B', s'\rangle}$	if $\neg \mathit{test}(a, s)$
R-S6	$\dfrac{\langle A, \exists v\, s_G \oplus s_L\rangle \rightarrow s\langle A', \exists v\, s'_G \oplus s'_L\rangle}{\langle \exists^{s_L}_{v} A, s'\rangle \rightarrow s\langle \exists^{s'}_{v} A', s'_G\rangle}$	
R-S7	$\langle p(\bar{v}), s\rangle \rightarrow s\langle [\bar{v}/\bar{x}], s\rangle$	if $\mathit{proc}(\bar{x})\{A\}$
R-S8a	$\dfrac{\langle A_j, s\rangle \rightarrow s\langle A'_j, s'\rangle}{\langle \sum_{i \in I} (a_i ? \rightarrow * A_i), s\rangle \rightarrow s\langle A'_j, s\rangle}$	if $\mathit{test}(a_j, s)$
R-S8b	$\dfrac{\langle A_j, s\rangle \nrightarrow s}{\langle \sum_{i \in I} (a_i ? \rightarrow * A_i), s\rangle \rightarrow s\langle A_j, s\rangle}$	if $\mathit{test}(a_j, s)$

Figura 2: Semántica operacional del lenguaje objetivo

En las figuras 1 y 2, se muestra respectivamente la sintaxis, en notación BNF, y la semántica operacional del lenguaje que ejecutará nuestro intérprete. A continuación se explican las construcciones del lenguaje y con qué reglas de la semántica operacional se relacionan.

Es posible observar que un programa del lenguaje es básicamente un conjunto de declaraciones de procedimientos (Decl) y una acción inicial (Action). Los procedimientos constan de un identificador (proc), una lista de argumentos (x) que serán accesibles por el cuerpo del procedimiento, el cual está formado por una acción.

Todas las acciones (Action) tienen un tiempo lógico de ejecución, de forma que permiten modelar el aspecto de ejecución síncrono de un programa. A continuación, se describen las diferentes acciones.

La acción Update (escrito `a!`) consiste en la actualización del estado del programa. Esta construcción sigue la semántica de la regla **R-S1**, donde se aplica la función

effect donde se actualiza la información del estado del programa con a. Esta función effect depende de las operaciones del Tell, en nuestro caso es la actualización destructiva de una variable, o un array típica de los lenguajes imperativos.

La acción *Sequence* (escrito `a1;a2`) sirve para modelar la secuenciación de dos acciones. A esta acción le corresponde la semántica de las reglas **R-S2a** y **R-S2b**. En esta acción se ejecuta primeramente la acción o conjunto de acciones si están agrupados mediante paréntesis de la parte izquierda de la regla, A, y posteriormente la parte derecha, B. Cabe destacar que el tiempo lógico de ejecución de esta acción es considerado nulo, y que el tiempo efectivo es la suma del tiempo de las acciones que lo componen.

La acción *Concurrence* (escrito `a1 || a2`) sirve para modelar la concurrencia del programa. A esta acción le corresponde la semántica de las reglas **R-S3a** y **R-S3b**. En esta acción se ejecuta la parte derecha, A, y la parte izquierda, B, a la vez. Es decir, en un mismo instante de tiempo se ejecutan las acciones de ambas partes que consuman dicho instante. Cabe destacar que el tiempo lógico de esta acción es el máximo tiempo que duran las partes A y B.

La acción *Conditional* (escrita `if a then a2 else a3`), que se compone de una guarda que es un test perteneciente a las operaciones del Ask, la parte A, que conforman las acciones que siguen a la palabra clave if, y opcionalmente la parte B, que conforman las acciones que siguen a la palabra clave else. Esta acción sigue la semántica de las reglas **R-S5a** y **R-S5b**, que indican que se ejecutara la parte A si la evaluación de la guarda es positiva, o la parte B en cualquier otro caso. El tiempo de ejecución de esta acción es nulo, la evaluación de la guarda es instantánea, y su tiempo efectivo es el de la parte que se ejecute.

La acción *Local declaration* toma una lista de nombres de variables y un cuerpo de acciones. La regla que modela el comportamiento de esta acción es la **R-S6**, que viene a declarar un conjunto de variables locales al cuerpo de acciones, que no estarán accesibles fuera de este bloque. El tiempo de ejecución de esta acción es nulo, y su tiempo efectivo es el de las acciones de su cuerpo.

La acción *Call* toma el nombre de un procedimiento y una lista de argumentos. Le corresponde la semántica de la regla **R-S7**. En esta acción se asocian los valores de las constantes y las referencias de las variables en la lista de argumentos de la llamada, al nombre de las variables de la lista de argumentos de la declaración del procedimiento, accesibles por las acciones del procedimiento. Esta operación consume una unidad lógica del tiempo de ejecución. Posteriormente, se ejecutan las acciones del procedimiento. Esta acción termina cuando termina la última acción del procedimiento, haciendo del tiempo efectivo de esta acción la duración de la ejecución del cuerpo más uno.

Por último están las acciones *Global choioce* y *Instantaneus choice*, compuestas de una secuencia de pares <guarda, acción>, y les corresponden la regla **R-S4**, y las reglas **R-S8a** y **R-S8b** respectivamente. Estas acciones evalúan todas guardas y eligen la acción asociada a una de las guardas que se cumpla. En caso de que ninguna se cumpla esta acción volverá a ejecutarse en el siguiente paso de ejecución.

Cabe resaltar que la diferencia entre el *Instantaneous choice* y el *Global choice*, es que en el primero la evaluación de las guardas no consume instante de tiempo, pero en el segundo sí.

A continuación, para clarificar la sintaxis del lenguaje se muestra un ejemplo de programa en la figura 3, donde se hace uso de alguna de las construcciones del lenguaje.

```
1   %% mult
2
3   mult(N,M,Z,S) {
4         if ?(M=1)? then
5                   !(Z=N)! ||
6                   !(S=1)!
7         else
8                   exists(M',Z',S')(
9                             !(M'=M-1)! ||
10                            !(S'=0)! ||
11                            mult(N,M',Z',S') ||
12                            ?(S'=1)? -> ( !(Z=N+Z')! ||
13                                                !(S=1)! )
14                  )
15        endif
16  }
17  [mult(2,3,Res,Aux)]
```

Figura 3: Ejemplo multiplicación recursiva concurrente en el lenguaje objetivo

Este ejemplo se trata de un programa que modela una multiplicación recursiva concurrente, en el que el cuerpo principal del procedimiento de la multiplicación, así como todas sus llamadas recursivas se ejecutan simultáneamente. Primeramente, se encuentra en la línea 3 la definición de una función de 4 parámetros y el cuerpo de dicha función de la línea 4 a la 15. En dicha función aparece en primer lugar una construcción condicional, donde se tiene `?(M=1)?` como un Ask y actúa como guarda del condicional. También aparece otro Ask en la línea 12 (`?(S'=1)?`), que actúa como semáforo de sincronización ya que fuerza a esperar para realizar la suma a que la llamada recursiva haya terminado actualizando `S'` . A continuación se encuentra una construcción concurrente en las líneas 5 y 6, de la 9 a la 12, y en la 12 y 13. Así mismo, se pueden encontrar construcciones Tell en las líneas 5, 6, 7, 11 y 12. Finalmente se localiza una declaración local en la línea 8 a la 13.

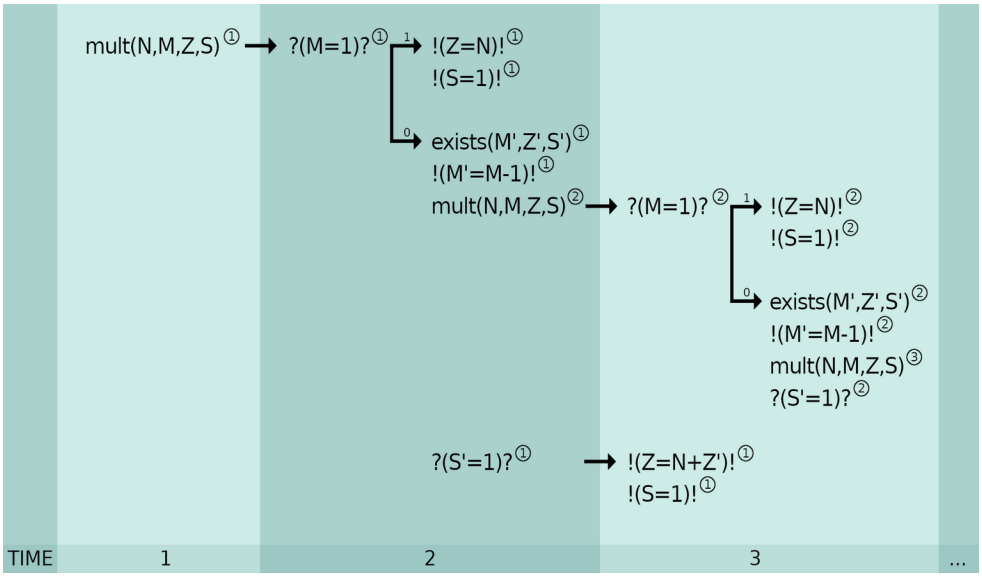

Figura 4: Grafico de ejecución temporal del ejemplo

Se muestra en la figura 4 un grafico que explica la ejecución temporal del programa de ejemplo de la figura 3. Para clarificar a que llamada recursiva pertenecen las acciones, se muestra a su lado de cada acción un circulo con el numero de la llamada.

Como puede observarse en este grafico, la declaración global y la evaluación del Ask del condicional no consumen ciclos de ejecución. Por contra, los Tell, el Ask de la elección global, y el cambio de contexto al llamar la función si que consumen un ciclo de ejecución. Debido a las construcciones concurrente, en un mismo ciclo de ejecución pueden darse la ejecución de varias construcciones en un principio líneales. Cabe destacar el caso de concurrencia que se da en la recursión de la función, donde la ejecución de la función y la ejecución de las llamadas recursivas se ejecutan simultáneamente. En este caso, la llamada principal acaba en el tercer instante pero se queda a la espera de que terminen las llamadas concurrentes.

La solución propuesta es la implementación de un intérprete de *bytecode* con su propia máquina virtual para el lenguaje especificado. La elección de este tipo de intérprete se explica en el capítulo 4.

3.1 Plan de trabajo

En este apartado se indica las tareas en las que consiste el proyecto, la duración estimada, y la planificación temporal de estas.

Las tareas se han obtenido a partir de la aplicación de las fases de la metodología utilizada a cada componente en el que se divide el intérprete. Es por ello que, en principio, se contemplan las tareas de diseño, implementación y pruebas para la realización de cada uno de los componentes. Para esto ha sido necesario comenzar realizando el diseño general de la arquitectura del intérprete.

Para planificar las tareas se han determinado las dependencias entre estas mediante el PDM (Precedence diagram method). Se ha comprobado con esta técnica que es posible paralelizar muchas tareas y asignar el desarrollo del compilador y de la máquina virtual a dos equipos diferentes. No obstante, solo se dispone de un solo miembro en el proyecto y es imposible paralelizar.

A continuación se muestra, en la figura 5, un diagrama de Gaant en el que se ha estimado la duración de las tareas, y cuándo deben realizarse. La duración de éstas se ha estimado a partir de lo que se ha tardado en realizar anteriormente proyectos similares, aunque en este caso la falta de experiencia impide realizar una buena estimación. Además se ha añadido cuánto se ha demorado cada tarea respecto al tiempo estimado.

El tiempo ha sido sustancialmente mayor al esperado en todas las tareas, pero el tiempo de duración relativo entre las tareas ha sido correcto.

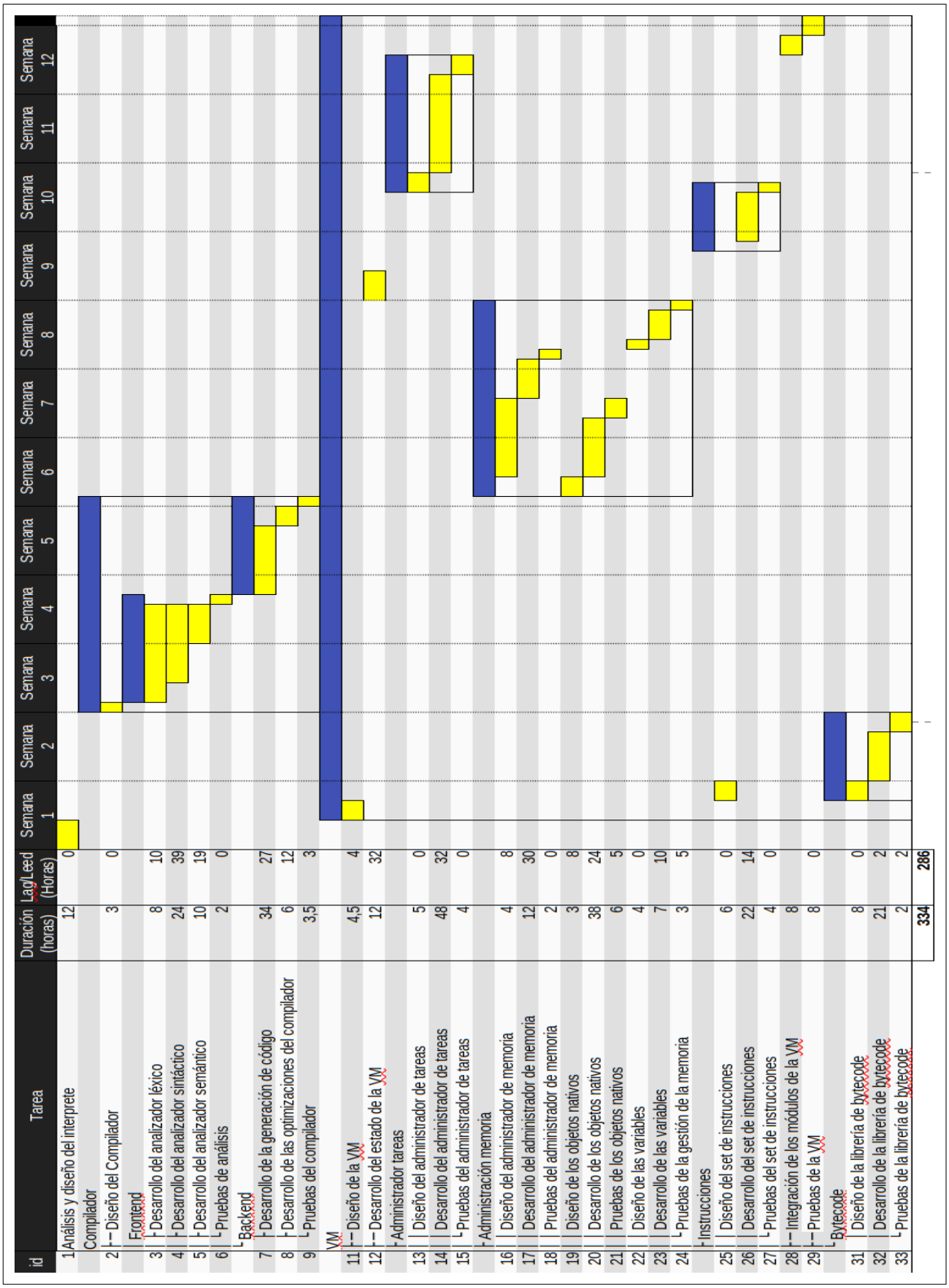

Figura 5: diagrama de Gaant

4. Diseño

En este apartado se explica en primer lugar la arquitectura general del intérprete. Seguidamente se detalla dicha arquitectura. Por último se detallan las tecnologías y herramientas utilizadas.

4.1 Arquitectura

Hay tres tipos de arquitecturas típicas de un intérprete. En primer lugar, existe el intérprete de ejecución directa, que ejecuta las construcciones del lenguaje en el mismo instante que las analiza. Seguidamente, se tiene el intérprete *walking-tree* [15], que crea una representación intermedia de la sintaxis del programa en forma de árbol, llamada AST (*Abstract Syntax Tree*), para posteriormente ejecutar acciones al pasar por cada nodo del árbol. Finalmente, existe el intérprete de *bytecode* que transforma el programa a una lista de instrucciones para una determinada máquina virtual que luego las ejecutará.

Cualquiera de estos tipos de intérpretes son compatibles con los objetivos principales de expansibilidad y suspensión de la ejecución mencionados en el apartado de objetivos. No obstante, basándose en el objetivo de la eficiencia, se ha elegido usar un intérprete de *bytecode*. Este tipo de intérprete tiene mayor velocidad de ejecución debido a que el bucle principal de ejecución es más simple y la estructura a decodificar, al ser una lista de instrucciones, presenta una mayor localidad de datos. Es decir, los datos al estar más compactos y contiguos en memoria aprovechan mejor la caché [16].

Este intérprete se divide en dos partes independientes: el compilador, que analiza y valida el código fuente de un programa antes de procesarlo a *bytecode*; y la máquina virtual, que ejecuta el programa al interpretar el *bytecode*. A continuación se explica la arquitectura de cada una de las partes del intérprete.

Compilador

El compilador, como se puede observar en el esquema de la figura 7, está formado por el *frontend*, que es la parte que analiza el código del programa de forma estática y se encarga de su validación, y el *backend*, que genera el *bytecode* y aplica varias técnicas de optimización. De esta forma es posible modificar por ejemplo la sintaxis del lenguaje sin tener que cambiar la generación de *bytecode*.

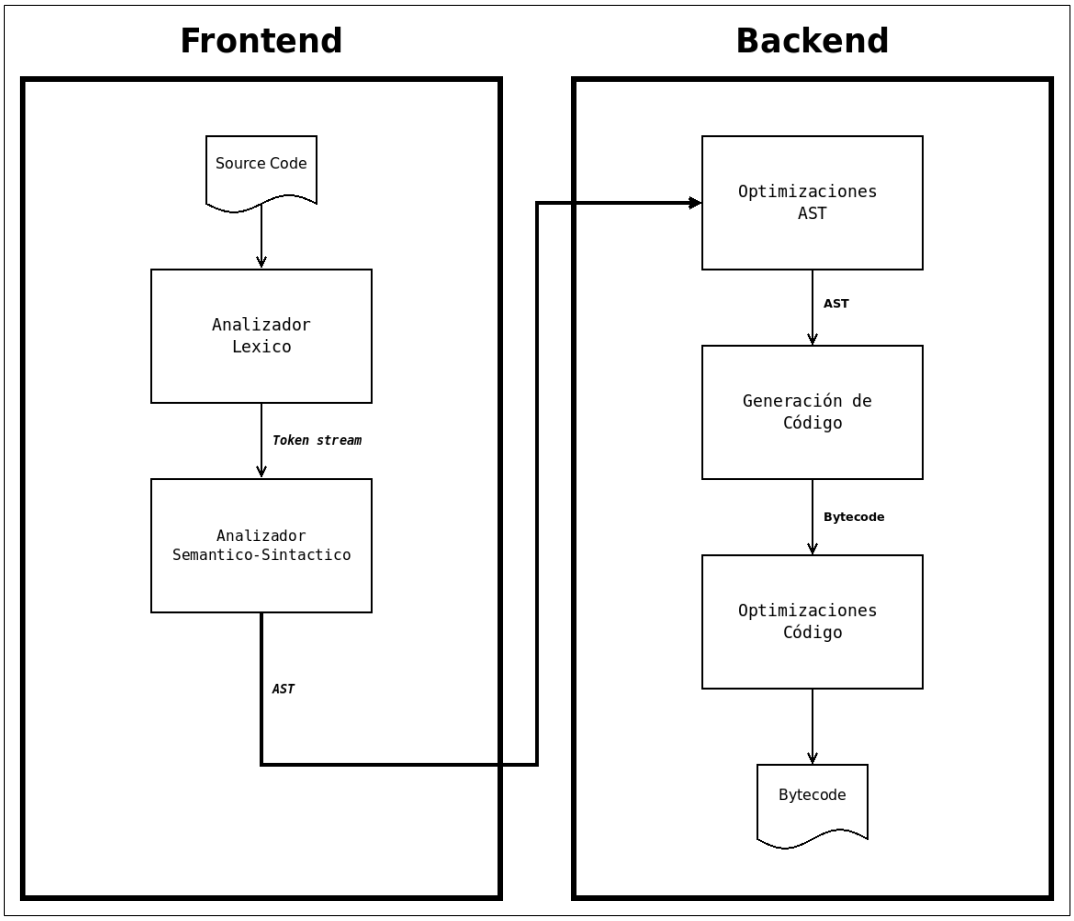

Figura 6: Fases del compilador

El *frontend* consiste en un analizador de tres pasadas que, en este caso, se verá reducido a dos en la práctica. La elección de este tipo de analizador y no uno monolítico se debe a su mayor facilidad de implementación y a la modularidad que aporta. Los diferentes módulos que componen el *frontend* son un analizador léxico y un analizador sintáctico-semántico. Este último engloba tanto el análisis sintáctico como el semántico, dada su fuerte conexión entre sí.

En la primera pasada, el analizador léxico genera un *stream* de *tokens* a partir del texto de un programa. Estos *tokens* son la representación de las diferentes secuencias de símbolos aceptadas por el lenguaje, como palabras clave, operadores, etc. Además, elimina información superflua como comentarios y separadores.

En la segunda pasada, el analizador sintáctico genera a partir del *stream* de *tokens* una estructura en forma de árbol que representa la sintaxis del programa, asegurándose de que coincide con la gramática del lenguaje. Este árbol podrá ser tanto un AST (*Abstract Syntax Tree*) como un CST (*Concrete Syntax Tree*).

Posteriormente, el analizador semántico se encarga de verificar que las acciones del programa sean válidas según la especificación del lenguaje, así como de generar una representación intermedia en forma de AST, por tratarse de una representación muy cercana a la gramática del lenguaje. Asimismo, el analizador semántico se

encarga de la comprobación de tipos [17][cap. 6.5], de asegurarse de que las llamadas a función respetan su definición, y de detectar otros errores semánticos como la división por cero, entre otras tareas. Para todo ello se emplea una tabla de símbolos que registra aquella información relativa a los mismos necesaria para llevar a cabo el análisis.

Por su parte, el trabajo del *backend* se divide en tres fases: dos fases de optimización y una fase intermedia de generación de código. En la primera fase se aplican las optimizaciones que resultan de una transformación del AST, como puede ser el *constant folding* [17], en el que expresiones de constantes son sustituidas por la constante resultante, o el CSE (*Common Subexpression Elimination*) [17], que busca instancias de expresiones idénticas y analiza si es conveniente reemplazarlas por una única variable. Seguidamente, se utiliza el AST resultante de las optimizaciones para generar el *bytecode*. Finalmente, se aplican optimizaciones como la minimización del uso de registros o la reordenación de instrucciones al *bytecode*.

Máquina virtual

Se ha elegido desarrollar una máquina virtual propia en lugar de utilizar una más genérica (como llvm [18]) o una diseñada para múltiples lenguajes (como JVM [19]), adquiriendo así la posibilidad de adaptarla mejor al dominio del lenguaje.

Hay dos tipos básicos de máquina virtual, la basada en registros y la basada en pila. Éstas se diferencian en cómo almacenan las instrucciones de *bytecode*. La máquina virtual basada en registros, al usar un formato de instrucción de tamaño fijo y que suele requerir un menor numero de instrucciones para un mismo programa, es más rápida [16], por tanto es la que se ha utilizado.

Como se observa en la figura 7, la máquina virtual está compuesta por los siguientes módulos:

- El administrador de memoria, que se encarga de la asignación de memoria para los diferentes objetos del programa a interpretar, así como de la reserva y liberación de los recursos de memoria al sistema operativo.
- La lista de instrucciones, necesaria para poder almacenar el código de programa.
- El registro de activaciones o pila de llamadas, que almacena la información sobre las subrutinas. Guarda la dirección de retorno a la instrucción a la que volver una vez finalizada la subrutina, la dirección a la variable de retorno, y las variables locales de la subrutina.
- La tabla de símbolos, que relaciona los nombres de las variables con su dirección de memoria.
- El administrador de tareas, que asigna las diferentes unidades de procesamiento de la máquina anfitrión a los diferentes procesos que forman el programa. Además, se encarga de la suspensión y activación de los procesos, empleando mecanismos de sincronización propios de estos sistemas.

- La librería *runtime*, compuesta por una colección de subrutinas que aportan funcionalidades básicas al lenguaje, como la gestión de entrada/salida o funciones matemáticas.

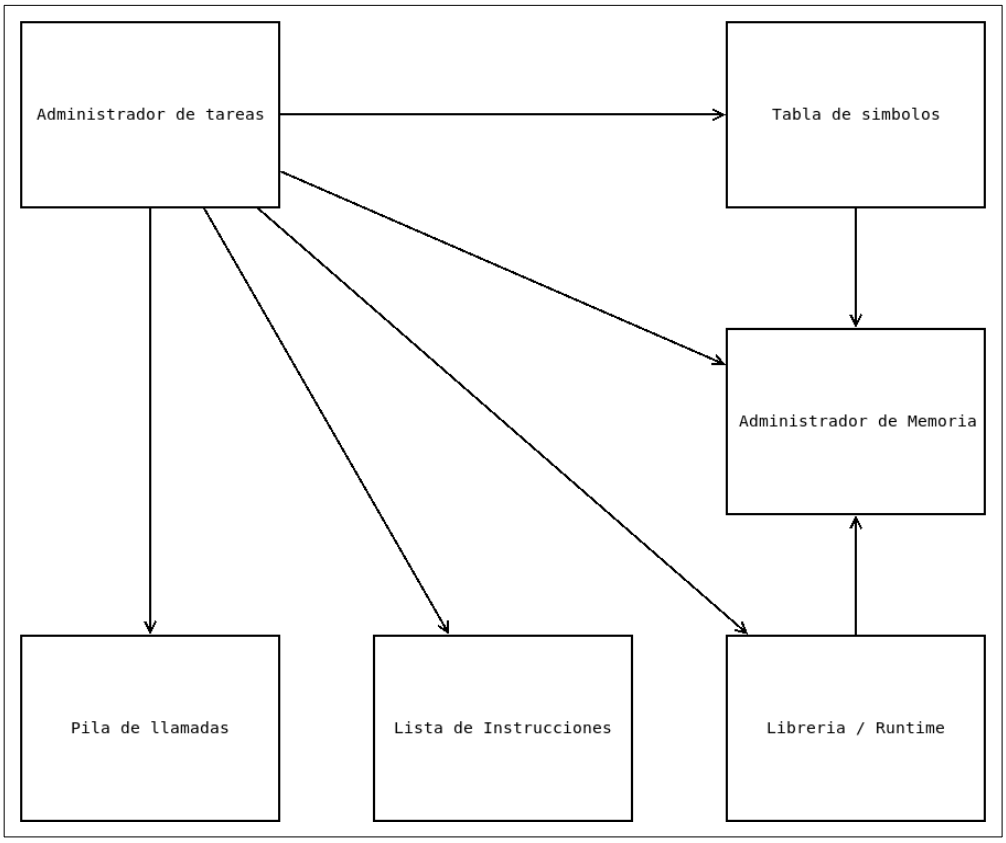

Figura 7: Diagrama de dependencia de los módulos de la VM.

4.2 Tecnología utilizada

Como punto de partida, se ha tenido que determinar en qué lenguajes de programación se implementará el intérprete. Dado que el proyecto consta de dos partes independientes, el compilador y la máquina virtual, es posible utilizar un lenguaje diferente para cada parte. Además, y dado que no se conocen a priori todos los ámbitos en los que el intérprete podría ser de aplicación, se ha buscado un diseño lo más genérico posible.

A la hora de escoger el lenguaje de implementación para la máquina virtual se han tenido en cuenta distintas consideraciones. Por una parte, se busca que la máquina virtual tenga una ejecución tan rápida y eficiente como sea posible ya que, dado que el lenguaje objetivo es interpretado, habrá un sobrecoste asociado a cada una de las instrucciones que formen los programas a ejecutar. Debido a ello, se han descartado la mayoría de lenguajes interpretados [20], que introducirían a su vez un sobrecoste adicional. Aunque no es requisito indispensable que el tiempo de cada paso de la ejecución de un programa sea determinista, se han descartado también aquellos

lenguajes en los que no sea posible controlar el flujo de ejecución debido a la acción del garbage collector. Asimismo, se desea realizar una gestión eficiente de la memoria, por lo que el lenguaje debe incorporar funcionalidades que permitan trabajar con la memoria de forma manual [21]. Finalmente, y debido a la naturaleza concurrente del lenguaje objetivo, se ha optado por usar un lenguaje con capacidades concurrentes como el uso de hilos o comunicación interproceso [22]. Todas estas restricciones apuntan a los lenguajes C, C++, Objective-C y Rust como posibles candidatos. Entre ellos, se ha escogido C [23] debido a que es lenguaje con el que se posee más experiencia.

Para la implementación del compilador la eficiencia no es tan relevante, ya que la compilación solo se realiza la primera vez que se ejecuta un programa. Por ello, sería preferible emplear un lenguaje sencillo de utilizar. No obstante, debido a que la máquina virtual y el compilador comparten la necesidad de manipular el mismo *bytecode*, también se usará C para así poder reutilizar la misma librería de *bytecode*.

Adicionalmente, hay que considerar el uso de librerías y *frameworks* para la construcción de compiladores e intérpretes.

Existen herramientas especificas para la generación automática de analizadores léxicos y sintáctico-semánticos de un lenguaje, para la generación de código intermedio y para la optimización de éste. El uso de un generador permite simplificar mucho el código y lo dota de mayor expresividad, facilitando la aplicación de cambios en la sintaxis del lenguaje. Además, el analizador generado suele tener menos errores y ser más eficiente. Desafortunadamente, el uso de un generador sintáctico depende del tipo de lenguaje objetivo para el que se esté desarrollando el intérprete, ya que hay lenguajes que no pueden ser procesados por los generadores actuales. Para valorar la conveniencia de usar un generador y la posterior elección del mismo, será necesario determinar previamente el tipo del lenguaje objetivo según la jerarquía de Chomsky [24].

En la figura 1 se muestra la gramática del lenguaje, donde los no-terminales están representados por la palabras que comienzan en mayúscula, y los terminales como símbolos o palabras en minúscula. Como puede observarse, todas las producciones de la gramática tienen la forma $N \rightarrow (N \cup T)^*$, donde N y T pertenecen a los símbolos no-terminales y terminales respectivamente. Esto indica que el lenguaje es libre de contexto [17], por lo que es posible utilizar generadores de la familia LR y LL.

Dentro de estos tipos, existen los generadores basados en *bactracking*, Earley, GLR, LALR(k), SLR(1) y LL(*). Aunque los generadores basados en *backtracking* y los Earley cubren todas las gramáticas LR, su complejidad computacional es polinómica, por lo que no se consideran adecuados [24]. Los basados en GLR cubren la mayoría de gramáticas LR y presentan una complejidad polinómica para gramáticas no-deterministas, tendiendo a una complejidad líneal en función del grado de determinismo, aunque en estos casos su ejecución es más lenta que la de los generadores LALR(k) [17][25]. El resto de familias tienen una complejidad líneal; no obstante, los LALR(k) son más rápidos que los LL(*), y estos que los SLR(1) [17][24]. Por su parte, los LALR(k) cubren más gramáticas que SLR(1), y estos más que los

LL(*) [17][24]. Otro factor a considerar, es que los generadores basados en LL no presentan problemas a la hora del análisis semántico debido al no afectar las acciones semánticas de las producciones al algoritmo de análisis sintáctico, y permiten el uso de atributos heredados en lugar de solo sintetizados como en el caso de los LR [26]. Debido a que se cree que el lenguaje puede ser expresado en una gramática de cualquier familia, se toma como criterio principal la eficiencia algorítmica. Por lo tanto, se ha escogido Bison [24][25], que presenta generadores GLR y LALR(k). Aunque en un principio vaya a utilizarse LALR(k), existe la posibilidad de usar GLR en caso de que la gramática crezca en complejidad, sin que sea necesario adaptarse a otro generador.

Para la elección del generador del analizador léxico se utilizará flex [24][25], pues tiene integración directa con el analizador sintáctico-semántico Bison.

Para la generación y optimización de código intermedio se dispone del *framework* LLVM [27][18] muy utilizado en la industria. No obstante, debido a que tiene una muy alta curva de aprendizaje queda descartado su uso por limitaciones de tiempo.

Debido a que C carece de librerías nativas de estructuras de datos genéricas, se usarán las proporcionadas en las librerías de GTK, que es multiplataforma. En un principio, se había elegido la librería ut*, pero se tuvo que cambiar por que provocaba fallos al usarlo con nuestro código debido a que ut* se basaba en macros.

Ya que se va a trabajar con C, se utilizará la herramienta make para automatizar el proceso de compilación de nuestro intérprete. Asimismo, se emplearán GDB [28] y Valgrind [29] para el proceso de depuración. Se ha elegido GDB ya que se poseen conocimientos previos sobre su uso, y Valgrind debido a que simplifica mucho detectar errores de memoria, tanto la detección de posibles memory leaks, como la de posibles accesos ilegales a memoria. Todas estas herramientas son fácilmente integrables en un procesador de textos extensible como puede ser Atom o Eclipse.

Se usará Doxygen [30] para la generación automática de la documentación, que ayudará a futuros desarrolladores a entender mejor el código y a extenderlo.

Finalmente, también se usara git [31] para llevar un seguimiento de las versiones del proyecto. A pesar de que el proyecto solo cuenta con un programador y no son necesarias de sus funciones colaborativas, el proyecto es de suficiente complejidad como para necesitar de un mecanismo que permita volver a una versión funcional, en el probable caso de introducir errores al implementar alguna funcionalidad o característica del programa.

4.3 Diseño detallado

En este apartado se explica los elementos que conforman los distintos módulos de la máquina virtual y el compilador, qué relaciones hay entre ellos y cómo intervienen en la ejecución del programa. Finalmente, se explica la estructura de directorios usada en la implementación.

Máquina Virtual

Bytecode

A continuación se detalla primero el formato del *bytecode* y sus componentes, y seguidamente se explican las clases usadas para modelar dichos componentes. Es importante definir el *bytecode* primero ya que es necesario para el diseño detallado e implementación del *backend* del compilador.

Un archivo de *bytecode* se compone de dos partes: el *header*, que contiene los datos necesario para saber cómo interpretar el *bytecode*; y un bloque de función con la función principal del programa. Cabe destacar que el archivo sigue una codificación big-endian para los números.

El *header* se compone del *magic number*, una secuencia de 8 bytes que forma que indica que se trata de un archivo de *bytecode*; y un entero de 4 *bytes* que codifican el número de versión del *bytecode*.

Un bloque de función consta principalmente de una lista de instrucciones, una lista de constantes y una lista de funciones. También consta de datos de depuración opcionales. Estos son el nombre del archivo del cual se generó este trozo de *bytecode* y una lista de números líneas del código fuente.

El hecho de que cada bloque de función contenga su propia lista de constantes y funciones hace posible la eliminación fácil de funciones que no se usen, o añadir nuevas funciones. Esto sería más difícil si se hubiese elegido una lista global de constantes y funciones. Por contra, el espacio utilizado es ligeramente mayor ya que las constantes de diversas funciones podrían aparecer duplicadas.

Las instrucciones del *bytecode* tienen un tamaño fijo de 4 *bytes*. Éstas están compuestas de un código de operación y de un número de campos que depende del tipo de instrucción asociado al código de operación. Estos pueden ser: **iNone** de 0 campos, **iABC** de tres campos, y **iABx** e **iAsBX** de dos campos. Todos los campos representan un entero sin signo excepto para el campo **B** de las instrucciones del tipo **iAsBx**. El formato de la instrucción se detalla en la figura 8.

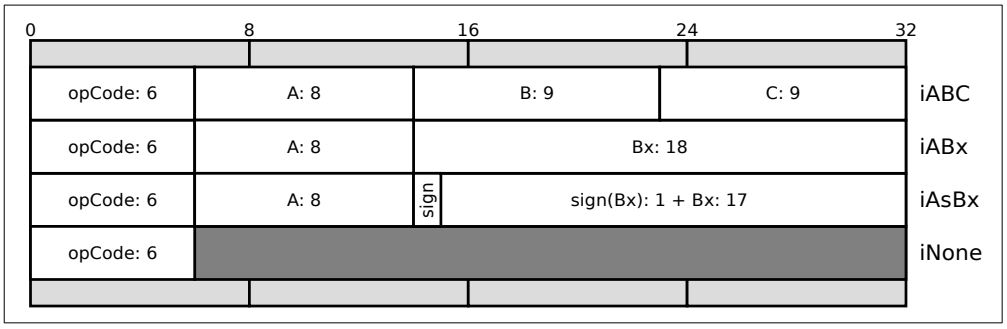

Figura 8: Formato de una instrucción de bytecode

Estos campos a su vez pueden hacer referencia al identificador de un registro (“R()”), un índice en la tabla de constantes (“CTE()”), un índice relativo de la lista de instrucciones (“INST()”), un índice en la tabla de funciones (“FOO()”) o un entero literal (“INT()”). Cabe destacar que la máquina virtual al interpretar las instrucciones, en el caso de que éstas referencien índices fuera de los límites de su correspondiente lista, se truncarán a los límites y se emitirá una advertencia.

Se incluye una tabla de todas las instrucciones disponibles en el anexo A, pero se explican las familias de instrucciones a continuación:

- Instrucciones de carga
  - Tipo: iABx
  - Campos: opCode, R(A), CTE(B)
  - Este tipo de instrucciones se encargan de cargar en el registro A el objeto referenciado por la variable identificada por la constante del índice especificado en el campo B. En el caso de carga de constantes se crea un nuevo objeto idéntico al representado en la lista de constantes por el índice del campo B.

- Instrucciones de almacenamiento
  - Tipo: iABx
  - Campos: opCode, R(A), CTE(B)
  - Este tipo de instrucciones hacen que el objeto del registro A sea asignado a la variable identificada por la constante del indice especificado en el campo B.

- Instrucciones aritmético-lógicas:
  - Tipo: iABC
  - Campos: opCode, R(A), R(B), R(C)
  - Este tipo de instrucciones se encargan de invocar operaciones aritmético-logicas tomando como argumento los objetos en los registros B y C, y depositando el resultado como un nuevo objeto en el registro A. La operación a realizar depende del tipo de los objetos pasados como argumentos y de la operación indicado por el opCode. Por ejemplo, para la instrucción ADD devuelve en A un objeto de tipo numérico resultante de la suma de B y C si estos son de tipo numérico, o un objeto de tipo booleano resultante de la disyunción lógica de B y C si estos son de tipo booleano. Si el tipo de estos objetos no tienen una definición para la operación, se producirá un error de ejecución y la operación se abortará sin realizar cambios en el estado del programa. El programa puede continuar su ejecución o abortar dependiendo de la configuración de la máquina virtual.

- Instrucciones de control de flujo:
  - Este tipo de instrucciones se encargan de modificar el flujo de ejecución del programa.
  - Instrucciones de salto relativas:

        - Tipo: iAsBx
        - Campos: opCode, R(A), INST(B)
        - Estas instrucciones mueven el contador de programa adelante o atrás el numero de instrucciones indicadas por el campo B. Si se trata de saltos condicionales solo se realiza si el objeto en el registro A evaluá como falso.
    - Instrucción CALL:
        - Tipo: iABC
        - Campos opCode, R(A)
        - Esta instrucción realizan un cambio de contexto y ponen el contador de programa al principio del bloque de instrucciones de la función en el registro A. Si el objeto en el registro A no se tratase de una función, se producirá un error de ejecución y la operación se abortará sin realizar cambios en el estado del programa. El programa puede continuar su ejecución o abortar dependiendo de la configuración de la máquina virtual.

- Instrucciones de control de concurrencia
    - Estas instrucciones sirven para modelar la concurrencia y proporcionan mecanismos de sincronización. Son de gran relevancia pues, a diferencia de las otras instrucciones que son más generales, son especificas a la naturaleza del lenguaje objetivo. Destacan las siguientes instrucciones:
        - SUSPEND, que marca un paso de ejecución para la tarea actual.
        - TASK, que crea una tarea nueva y la prepara para su ejecución inmediata.
        - TASK_WAIT, que actúa de barrera para la sincronización de la tarea actual con las tareas “hijo” creadas por TASK.

Las constantes pueden ser strings, números enteros y números reales. Éstas constan del tipo de dato representado con un numero de 4 bytes y su valor. En el caso de los números enteros el valor esta representado como un entero con signo de 32 bits. Los valores reales seguirán el formato IEEE-724 que utiliza 32 bits. Por otro lado, los strings consisten en la longitud del string en bytes, y una cadena de bytes de dicha longitud. La strings están codificados en ASCII.

La librería de bytecode consiste en las clases necesarias para representar el bytecode, y las funciones para volcar esta representación a un archivo binario. En la figura 9, se muestra un diagrama de clases que forman esta librería. Es importante el orden y tamaño de los atributos cuando se vuelcan a un archivo, es por ello que se indica en el diagrama el orden con “[#]” al lado de cada atributo y el tamaño mediante el tipo del atributo.

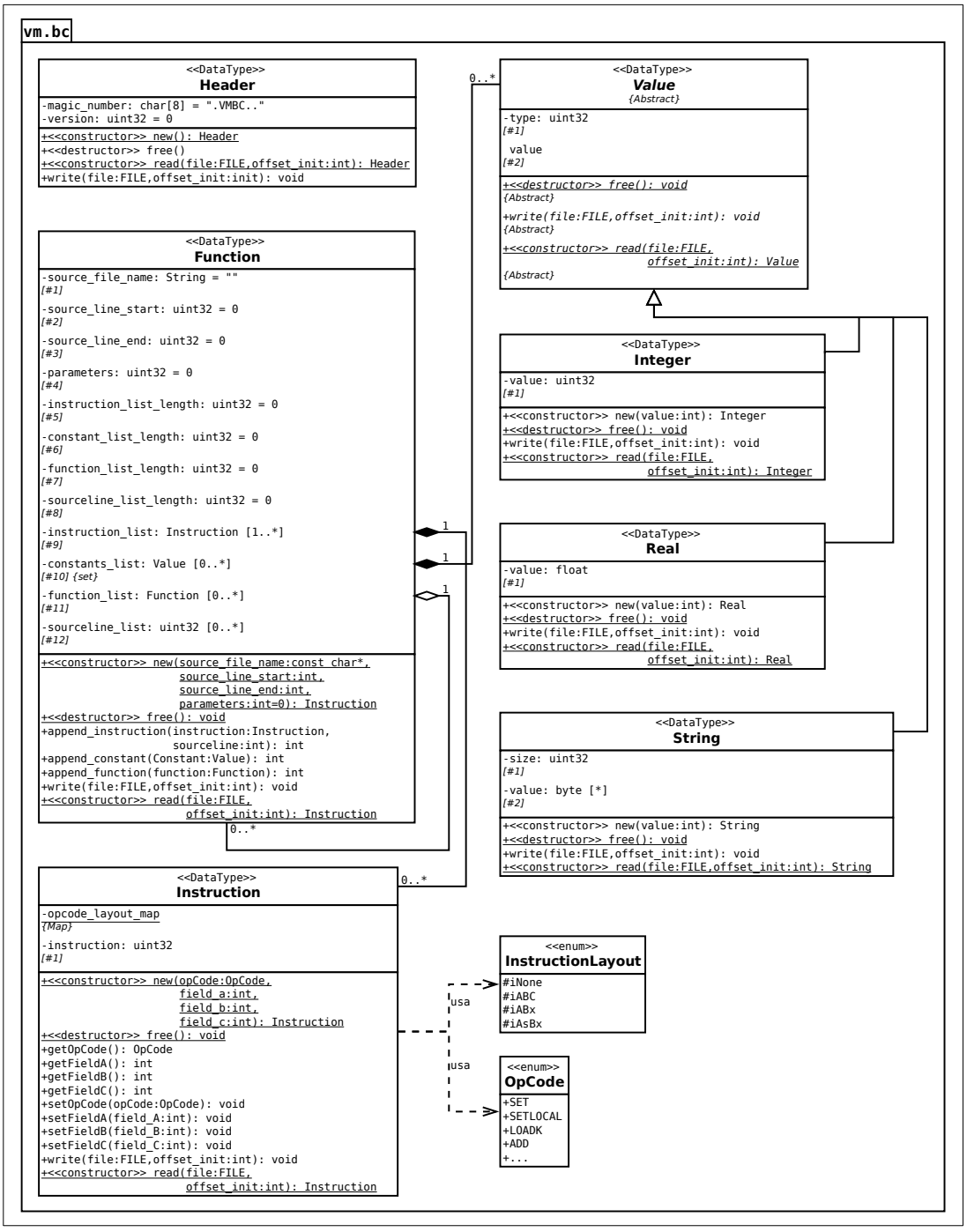

Figura 9: Diagrama de clases de la librería bytecode

Estas clases solo representan datos sin ningún comportamiento y cuya instancia es inmutable, es por ello que se consideran DTO(*Data transfer object*). Debido a que las instancias son inmutables solo tienen constructores y no hay *setters* u otra forma de modificar los atributos de la instancia una vez creada. La excepción a esto son las clases `Instruction` y `Function`. En el caso de una instrucción puede ser difícil saber el valor de los atributos en el momento de creación, por ello se tienen *setters* para los atributos de esta clase. En el caso de función tiene métodos para añadir elementos a

las listas de funciones, constantes e instrucciones de forma que no es necesario el uso de una lista auxiliar para su construcción. Además en el caso de las constantes se controla que no hayan duplicados en la lista. No obstante, no hay forma de eliminar elementos de dichas listas, ya que representan objetos inmutables.

Todas las clases proporcionan métodos para la lectura y escritura a archivo del *bytecode*. El método `write()`, vuelca el valor de la instancia en un archivo, en la posición indicada. El método de clase `read()` crea una nueva instancia con los valores encontrados en el archivo y posición indicados.

Cabe mencionar que la clase `Instruction` tiene un atributo de clase que consiste en un mapa para establecer la correspondencia de los posibles códigos de operación de la instrucción con el *layout* de los campos de la instrucción.

Estado y Pila de llamadas

La clase `State` representa el estado de un programa o proceso (`Task`). Como se puede ver en el diagrama de la figura 10 se compone de:

- Una una tabla de símbolos global (`global_sym_table`) y otra local (`sym_table`), donde se guarda la asociación de las variables del programa con los objetos que referencian. La tabla global es accesible en todo momento en la ejecución de un programa, mientras que la tabla local cambia según el contexto (i.e. dentro de la llamada a una función).
- El *garbage collector* (`gc`), que se encarga de tener un seguimiento de los objetos que ya no son referenciados por variables o registros para su posterior liberación
- Una lista de instrucciones en *bytecode* de solo lectura (`function`) que representa el programa o proceso en ejecución.
- El puntero de programa (`pc`), que indica la próxima instrucción a ejecutar mediante un indice entero, refiriendo a la lista de instrucciones del estado.
- La pila de llamadas (`call_stack`), que registra las subrutinas activas del programa y todo lo necesario para reanudar la ejecución de la subrutina anterior en el momento que se realizo el cambio de contexto.
- Una lista de argumentos (`funct_args`), con los que se ha invocado al programa o proceso.
- Una lista de registros (`registers`), donde se hayan referenciados los objetos sobre los que operan las instrucciones de *bytecode*.

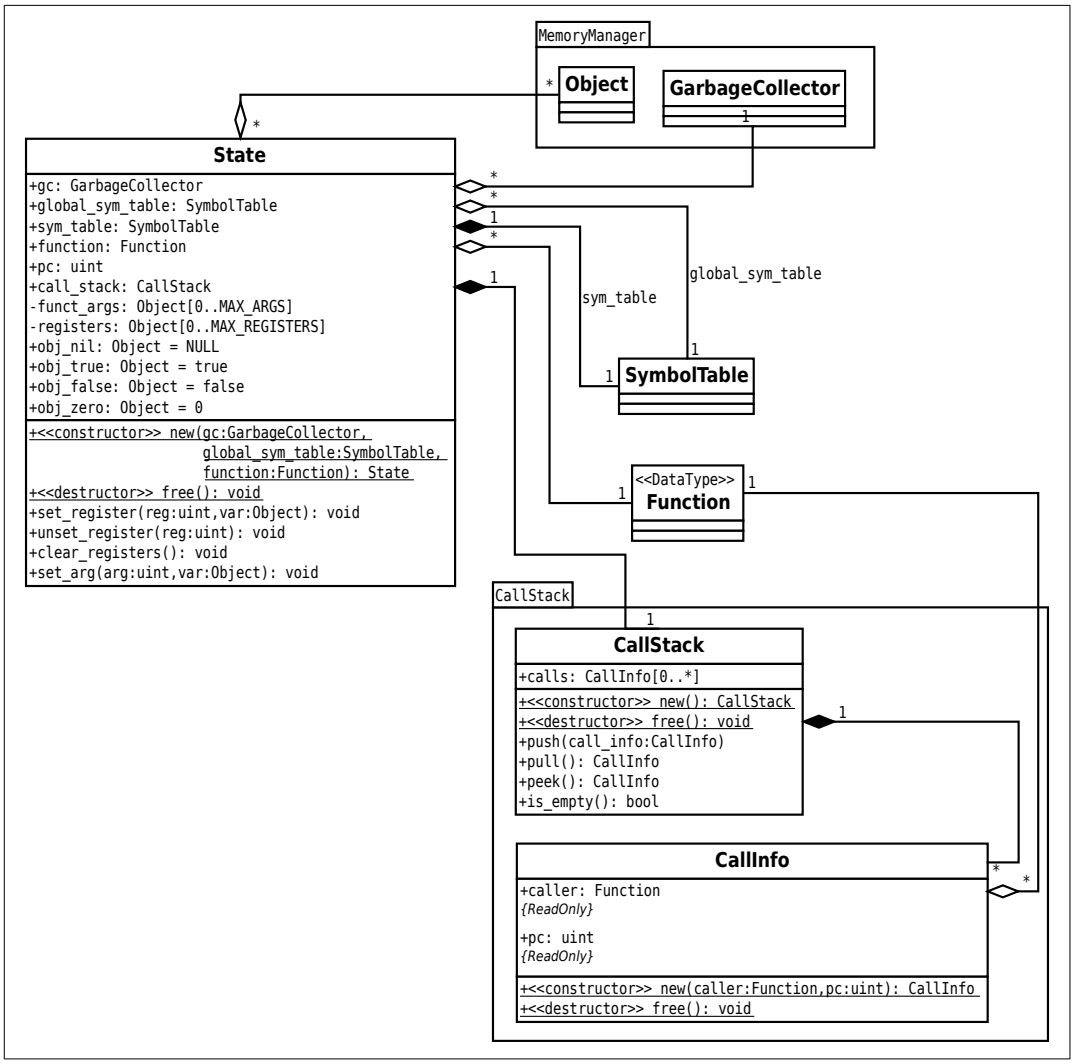

Figura 10: Diagrama de clases para el estado de la VM

La tabla de símbolos global y el *garbage collector* son compartidos y modificables por múltiples instancias, y por tanto tienen que tener mecanismos para asegurar la consistencia de los datos. La lista de instrucciones también puede ser compartida por múltiples estados ya que es de solo lectura.

Todos los atributos de esta clase, excepto registros y argumentos, son de acceso publico, pero en el caso de la pila de llamadas, la tabla de símbolos global y el *garbage collector* no se debería poder cambiar la referencia al objeto, aunque si modificar el objeto referenciado.

También se incluye en el estado la referencia a objetos inmutables presentes en todos los programas (como NULL, true, false o 0) para facilitar su acceso.

Así mismo, esta clase proporciona métodos para el manejo y modificación de los registros y argumentos. Esto son: `set_register(uint reg, Object* var)`, que guarda el objeto `var` en el registro `reg`, eliminando la referencia al objeto que hubiese

anteriormente en el registro; `unset_register(uint reg)`, que vacía el registro `reg` eliminando la referencia del objeto en este, `clear_registers()`, que vacía todos los registros; y `set_arg(uint arg, Object* var)`, que guarda el objeto `var` como el argumento posicional del programa especificado en `arg`.

La pila de llamadas esta implementada en la clase `CallStack`. Como se ha mencionado anteriormente es una pila de información de llamadas activas. Esta clase implementa las funciones básicas de operaciones típicas de una pila: `peek()` y `pull()` que devuelven la información de la última llamada realizada, con la diferencia de que la segunda función elimina esta información de la pila, dejando a recuperar la anterior llamada; `push(CallInfo call_info)` que debería usarse para añadir la información de la llamada actual al realizar un cambio de contexto; y la función `is_empty()` para comprobar si la pila esta vacía y ya no hay más llamadas que devolver.

La información de la llamada esta representada por los objetos de la clase `CallInfo`. Esta clase se compone de atributos de solo lectura. Se debería usar para guardar el contador de programa (`pc`) y la función actual (`caller`) justo antes de realizar una llamada a función.

Administrador de Memoria

Este módulo es el encargado de gestionar la memoria usada por los objetos del programa, pero no las estructuras necesarias para la ejecución del intérprete. Comprende el *garbage collector* y las clases que definen los tipos nativos del lenguaje objetivo. En la figura 11 se muestra el diagrama de clases que conforman este módulo.

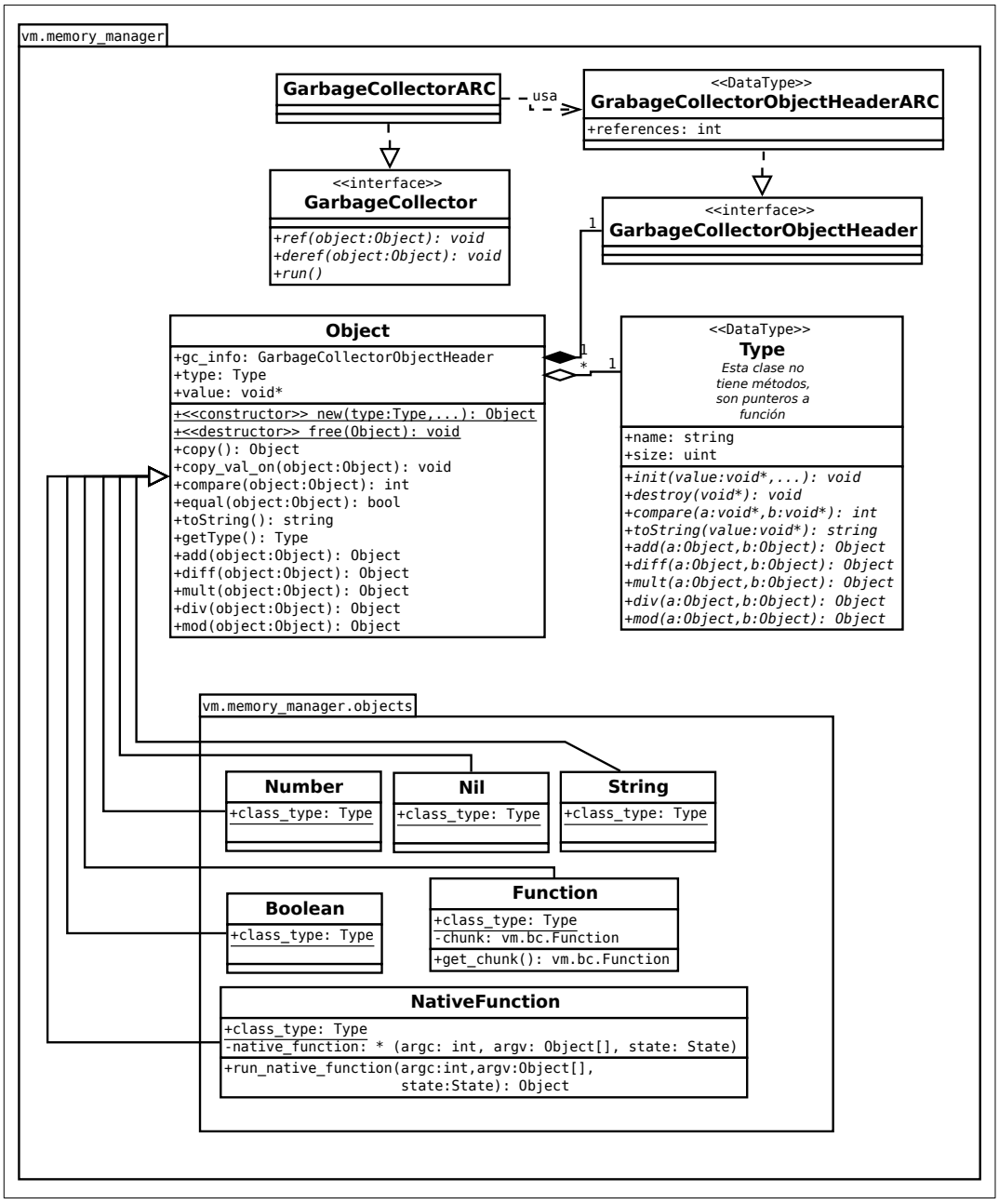

Figura 11: Diagrama de clases del administrador de memoria

El *garbage collector* se compone de las clases `GarbageCollector` y `GarbageCollectorObjectHeader`. La primera representa al propio *garbage collector*, y ofrece los métodos `ref(Object object)` y `unref(Object object)` para controlar las referencias a estos objetos, así como la función `run()` que inicia su ejecución. La segunda clase únicamente sirve de contenedor de la información de cada objeto del programa necesaria para el funcionamiento del algoritmo que implementa el *garbage collector,* como puede ser un contador que indique las veces que está referenciado el objeto.

Los objetos presentes en la ejecución de un programa están representados por la clase `Object`. Todos los objetos actúan como una referencia a un valor, y no hay valores primitivos. Esto permite implementar el paso de las variables por referencia al llamar a una función.

Todos los objetos tienen un tipo, representado por la clase `Type`, que contiene el nombre del tipo (`name`), el tamaño del valor del tipo (`size`), y una tabla de métodos virtuales con los métodos para inicializar el valor (`init`), liberar la memoria dinámica usada al inicializar el valor (`destroy`), comparar dos valores (`compare`), representar el valor en un *string* (`toString`) y los métodos `add`, `diff`, `mult`, `div` y `mod`, encargados de devolver un nuevo objeto resultante de la aplicación de operaciones aritméticas a dos objetos de este tipo. Un tipo no esta obligado a proveer estos métodos.

La clase `Object` dispone de los siguientes métodos:

- `Object copy()`, que devuelve un nuevo objeto con el mismo tipo y valor que el de objeto que ejecuta dicho método.
- `copy_val_on(Object object)`, que copia el valor y el tipo del objeto pasado como argumento y lo asigna al objeto que invoca este método.
- `int compare(Object object)`, que hace la comparación entre el valor del objeto que ejecuta este método y el valor del objeto que se le pasa como parámetro, devolviendo un numero negativo si el valor es menor, positivo si es mayor, y 0 si es igual.
- `bool equal(Object object)`, comprueba si dos objetos son del mismo tipo y equivalente valor.
- `string toString()`, que devuelve una representación del valor de este objeto
- `Type getType()`, que devuelve el tipo de un objeto.
- `Object add(Object object)`, `Object diff(Object object)`, `Object mult(Object object)`, `Object div(Object object)` y `Object mod(Object object)`, que devuelven un nuevo objeto con el valor resultado de realizar operaciones aritméticas sobre el valor del objeto que invoca al método y del objeto que se le pasa como argumento.

Estos métodos son esencialmente un *wrapper* de los métodos virtuales especificadas en el tipo del objeto. Cabe destacar que en el caso de que no hubiese un método definido, se mostraría un error y se abortaría la ejecución de la máquina virtual.

A partir de la clase `Object` se construyen los objetos de todos los tipos que soporta el lenguaje: `Number`, `String`, `Boolean`, `Nil`, `Array`, `Function` y `NativeFunction`. Estos objetos pueden disponer de métodos adicionales, como es el caso `Function` que dispone del método `bc.Function get_chunk()`, que devuelve el *bytecode* correspondiente a la función.

Tabla de símbolos

A diferencia de muchos intérpretes que en la fase de compilación enlazan el nombre de las variables con una referencia a memoria, en este intérprete se usa enlazamiento dinámico. Es decir, la asociación entre el nombre de la variable y la referencia a memoria se determina en tiempo de ejecución. Esto permite la implementación de lenguajes con ámbito dinámico. El módulo de la tabla de símbolos se encarga de implementar el ámbito del lenguaje.

La tabla de símbolos tiene la misión de asociar el nombre de cada variable con su respectiva referencia a memoria. Esta tabla no es una tabla de símbolos convencional, ya que forma una estructura de datos conocida como *saguaro stack* [32]. Esta estructura es una fusión entre una pila y un árbol, que se comporta como una pila que sigue una política LIFO, con la diferencia de que los niveles del *stack* pueden tener más de un nivel posterior. Esto permite implementar una pila que puede ser usada simultáneamente por varios procesos sin que sea necesario crear un copia de la misma. En la figura 12 se muestra el diagrama de clases que componen esta estructura.

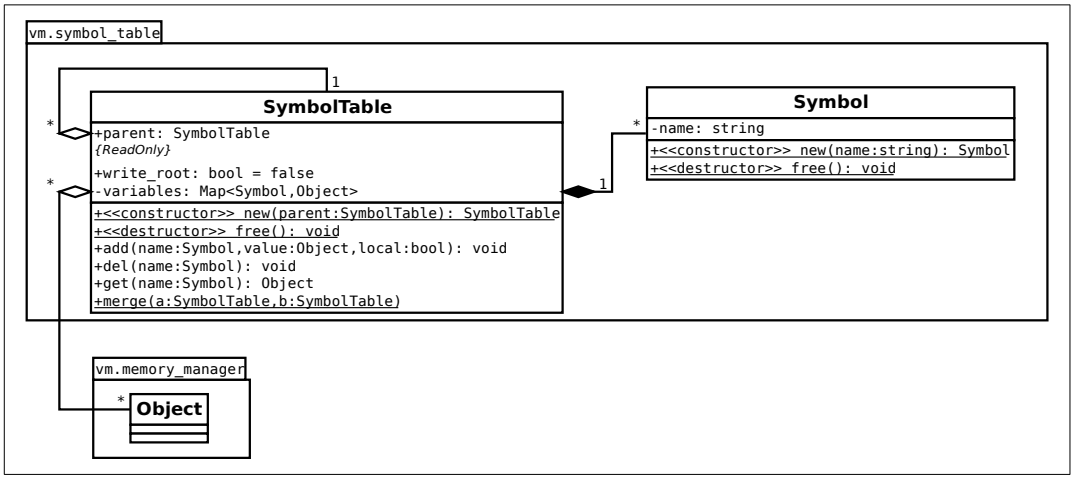

Figura 12: Diagrama de clase de la tabla de símbolos

La clase `TableSymbol` representa al mismo tiempo un ámbito en el lenguaje y un nivel en la pila. Contiene un puntero a otro `TableSymbol` que representa el ámbito superior que lo engloba, de forma que es posible acceder tanto a las variables del propio ámbito como a las del ámbito superior, mientras que no se permite el acceso en sentido inverso. Adicionalmente, contiene un *array* asociativo que relaciona nombres de variables, representados por la clase `Symbols`, y los objetos del programa. Aunque se trata de una pila, esta clase no presenta los métodos propios de este tipo de estructura de datos (i.e. *push, pull, peek*). La pila es creada implícitamente a partir del constructor de la clase; este recibe como argumento el nivel actual, es decir, el nivel superior, y devuelve el nuevo nivel actual. De igual manera, el destructor de la clase elimina el nivel actual de la pila y devuelve el nivel superior.

Además, esta clase dispone del método `add(Symbol name, Object value, bool local)`, que busca el nombre de la variable empezando por el nivel actual, y recorriendo los niveles superiores de la pila hasta encontrar la primera ocurrencia. En caso de encontrarla, sobrescribe su valor; en caso contrario, crea una nueva variable con ese nombre y valor en el ámbito actual. En el caso de que el argumento `local` sea `true`, únicamente actúa en el nivel actual. Adicionalmente, la clase dispone de los métodos `del(Symbol name, Object value, bool local)` y `get(Symbol name, Object value, bool local)`, que siguen el mismo algoritmo de búsqueda y borran la variable o devuelven su valor, respectivamente.

Cabe destacar la relevancia que tiene esta estructura en el aspecto concurrente del lenguaje, ya que puede usarse para evitar el uso de *locks* que puedan formar cuellos de botella cuando se implementan mecanismos para leer y/o modificar concurrentemente una variable de forma segura. Un ejemplo de este caso es el uso del GIL (*Global Interpeter Lock*) en Cpython [33].

Para evitar el uso de *locks,* en cada paso de programa cada una de las tareas crea un nivel adicional en la tabla de símbolos. De esta forma, se crea un contexto local a cada tarea, preservando el valor de las variables cuando estas son accedidas por otras tareas, y guardando su modificación como una nueva variable con el mismo identificador en el contexto local. Una vez todas las tareas que intervienen en un paso de ejecución se han completado, se unen los contextos locales a los niveles anteriores de la tabla.

Como ayuda para implementar el mecanismo anterior, la clase `SymbolTable`, dispone del atributo `write_root` que, si tiene el valor *true,* indica a la función `add` que no se debe buscar más allá de este nivel. Además, dispone del método `merge(SymbolTable a, SymbolTable b)` que une la tabla a sobre la tabla b, insertando en b todos las variables de a mediante el uso de la función `add` y eliminándolas de a.

Runtime

En este apartado se detallan las clases encargadas de ejecutar las instrucciones de *bytecode*, y las funciones que forman parte de las librerías *runtime* de la máquina virtual. En la figura 13 se muestra el diagrama de clases que componen este módulo.

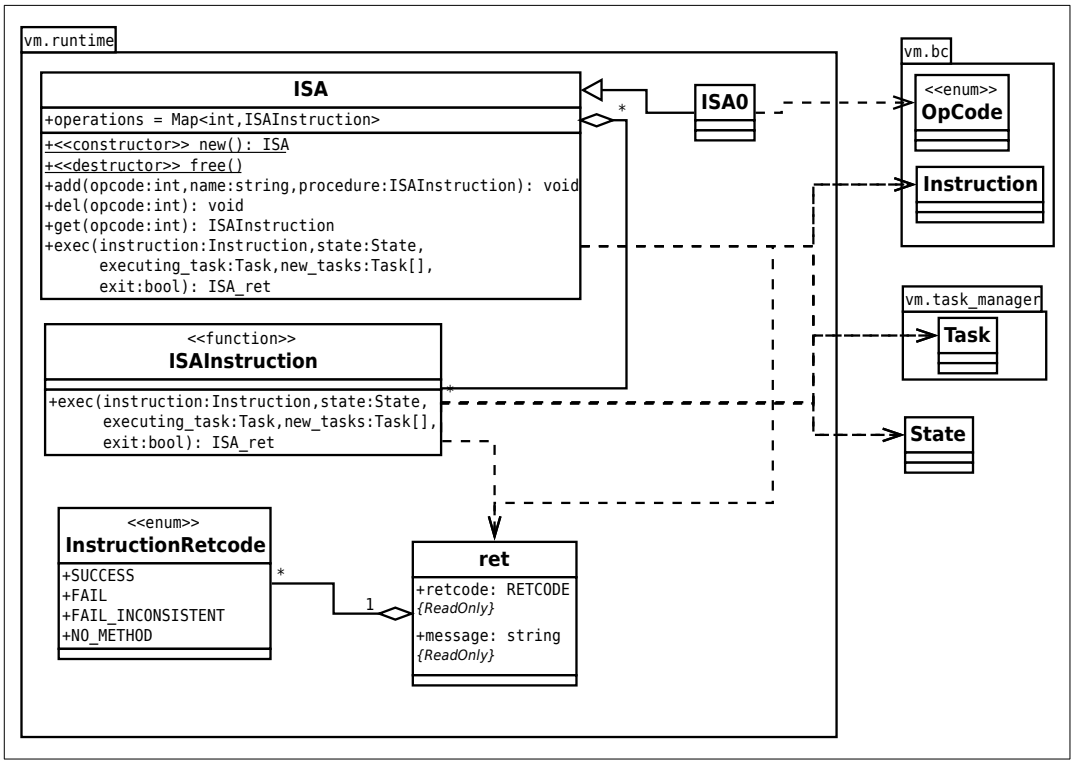

Figura 13: Diagrama de clases de la ISA

Primeramente, se tiene la clase `ISA`, que permite construir una tabla asociativa con los códigos de operación de *bytecode* y la función a ejecutar para cada operación. Esta clase actúa principalmente de interfaz para ejecutar las instrucciones de *bytecode* de forma independiente a la implementación de las operaciones. Esto se logra mediante la llamada a la función `ret exec(Instruction instruction, State state, Task[] new_tasks, bool exit)` que expone esta clase. Dicha función extrae el código de operación de la propia instrucción y llama a su vez a la función, ya registrada, que se corresponde con el código extraído. Básicamente, implementa el patrón *strategy* y permite elegir un conjunto de operaciones de *bytecode* según la versión de *bytecode* a interpretar.

Además, esta clase dispone de las funciones `add(int opcode, string name, ISAInstruction operation)`, `del(int opcode)` e `ISAInstruction get(int opcode)` para añadir, borrar u obtener operaciones de *bytecode* de la ISA respectivamente. Estas permiten añadir operaciones nuevas y emplear funciones *wrapper* sobre las operaciones ya implementadas, extendiendo y componiendo la ISA.

Las funciones correspondientes a las operaciones a registrar están representadas mediante la clase `ISAInstruction` y tienen la misma firma que la función `exec`. Reciben como argumentos la instrucción de *bytecode* a ejecutar (`instruction`), el estado sobre el que se efectúa la operación (`state`) y la tarea que se está ejecutando y es responsable de la operación ejecutada (`running_task`), y como argumentos de salida una lista de tareas nuevas que se han creado durante esta operación (`new_tasks`), así como el *flag* `exit` indicando si esta operación requiere la terminación del intérprete.

También devuelven una instancia de la clase `ret`, que contiene un código y un mensaje de error. Este código debe de ser uno de los especificados en la enumeración `InstructionRetcode` e indica si la instrucción se ha ejecutado correctamente (`SUCCESS`), no se ha ejecutado porque no existe una función asociada a la operación solicitada (`NO_METHOD`), no se ha ejecutado al encontrar un fallo pero el estado sobre el que se trabaja es consistente o no se ha modificado (`FAIL`), o si ha fallado habiendo modificado el estado y dejándolo inconsistente (`FAIL_INCONSISTENCE`).

La clase `ISA0` es una ISA en la cual ya se han registrado las operaciones de la versión 0 del *bytecode*. Esta clase no es realmente una clase heredada, ya que no modifica o añade métodos o atributos, pero al tener ya registradas las operaciones del *bytecode* se puede considerar a efectos prácticos como tal.

Por último, las funciones nativas, como funciones de entrada/salida, definidas como funciones de programa, deben de ser añadidas manualmente como parte del objeto `Function` (no confundir con la clase `Function` del *bytecode*) que se ha detallado en el apartado anterior “Administrador de Memoria”, registrando este en la tabla de símbolos global del estado del programa. Este mecanismo no solo permite a la máquina virtual proveer de funciones nativas al programa, sino que también es el mismo mecanismo ofrecido para incluir funciones nativas por parte de terceros. No obstante, cabe destacar la limitación de que las funciones nativas siempre duran un paso de ejecución. Esta limitación se debe a que no se puede garantizar la consistencia del estado si este es modificado con los mecanismos de concurrencia provistos por la máquina virtual y por la función nativa a la vez.

Administrador de tareas

El administrador de tareas es el módulo encargado de la orquestación y ejecución de las tareas creadas durante la ejecución de un programa. En la figura 14, se puede ver el diagrama de clases que conforman el administrador de tareas. Consiste en la clase `Task`, que representa una tarea; la clase abstracta `TaskManager`, que sirve de interfaz para la implementación de un planificador de tareas; y las clases `TaskManager*`, que serán las respectivas implementaciones del planificador. Actualmente, solo se implementa el tipo de planificador `TaskManagerAsyncInterleaving`.

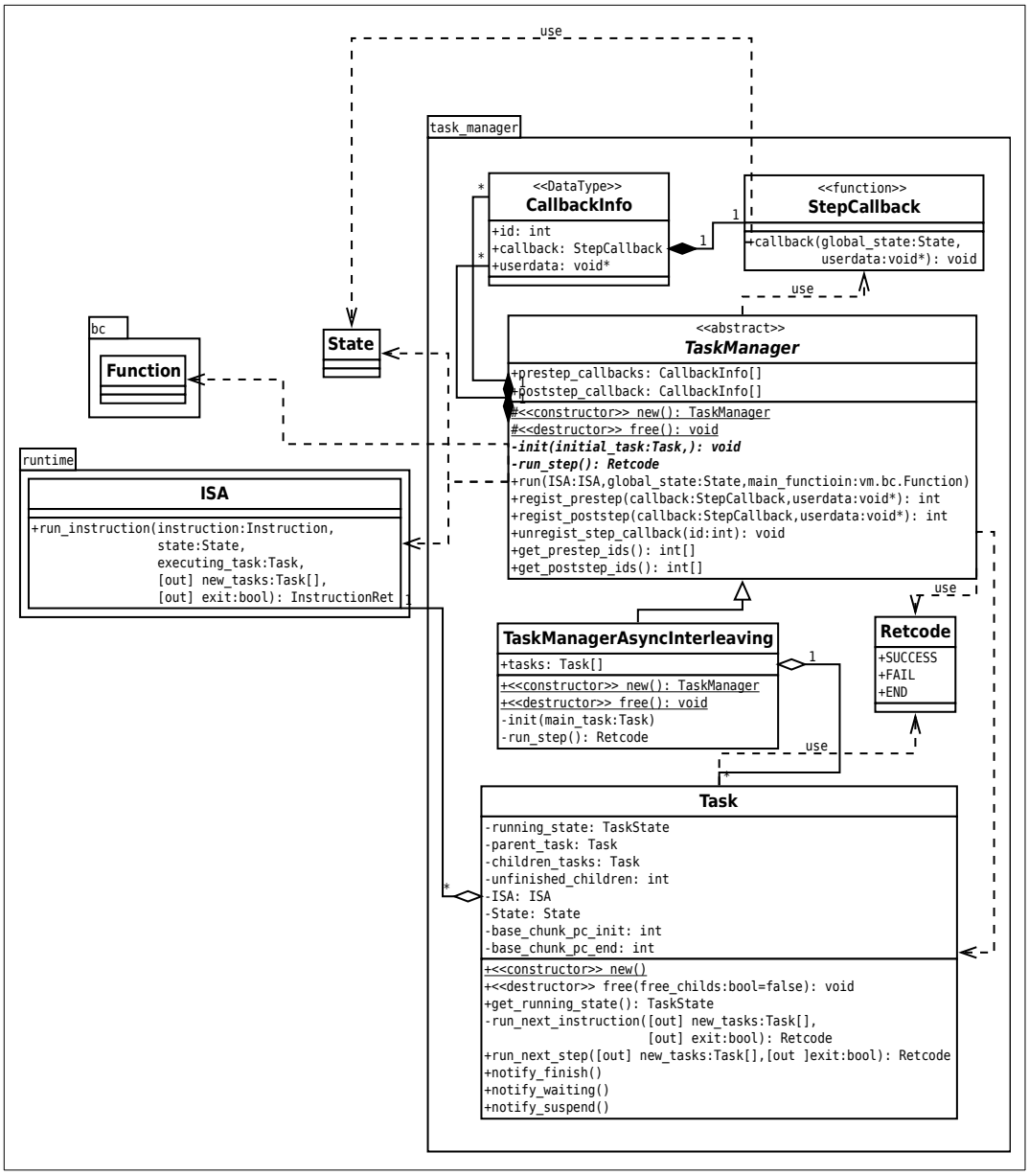

Figura 14: Diagrama de clases del administrador de tareas

Una tarea (`Task`) representa una unidad de ejecución, compuesta por unidades de ejecución más pequeñas e indivisibles llamadas pasos de ejecución. Además, las tareas pueden contener a su vez subtareas que, debido a su naturaleza concurrente, están planificadas de forma independiente y no forman parte de su plan de ejecución. Una tarea con subtareas debe suspender su ejecución hasta que las subtareas hayan finalizado.

Esta organización de las tareas permite la representación de acciones concurrentes. Por ejemplo, para nuestro lenguaje objetivo, al ejecutar la acción `!(Z=N)! || ( !(S=1)! ; ( !(S=2)! || !(Z=3)! ) )` se crearán dos tareas adicionales: una tarea A, que consiste en un paso de ejecución `!(Z=N)!,` y otra tarea B con dos pasos de

ejecución, `!(S=1)!` y `!(S=2)! || !(Z=3)!`. A su vez, cuando se ejecute el paso `!(S=2)! || !(Z=3)!` de la tarea B se crearán dos subtareas de B: BA, formada por el paso de ejecución `!(S=2)!` , y BB, formada por el paso `!(Z=3)!` .

```
1  //!Z=N! || (S=1)! ; ( !(S=2)! || !(Z=3)! ) )
2  TASK X 3 X
3      LOAD R(1) CTE(N) X
4      SET  R(1) CTE(N) X
5      SUSPEND X X X
6  TASK X 16 X
7      LOADK R(1) CTE(1) X
8      SET   R(1) CTE(S) X
9      SUSPEND X X X
10     TASK X INT(3) X
11         LOADK R(1) CTE(2) X
12         SET   R(1) CTE(S) X
13         SUSPEND X X X
14     TASK X 3 X
15         LOADK R(1) CTE(3) X
16         SET   R(1) CTE(Z) X
17         SUSPEND X X X
18     TASK_WAIT
19 TASK_WAIT
```

Figura 15: Bytecode para una tarea con subtareas que se ejecutan en paralelo

Una tarea, como puede verse en la figura 14, consiste en un conjunto de instrucciones y un contexto de ejecución. Este contexto esta representado por la clase `State` que como se explica en el apartado “Estado y Pila de llamadas”, consiste en un de registro de variables de programa, código de programa, estructuras de control para el flujo de ejecución de la tarea y registros.

Cuando se crea una nueva tarea las estructuras de control y registros del estado de la nueva tarea se duplican e inicializan al estado inicial donde los registros y la pila de llamadas están vacíos. El código de programa no se duplica, este es referenciado ya que se asume que no es liberado durante la ejecución de la máquina virtual. Debido a que el código de programa de la tarea hijo esta contenido en el código del padre (ver figura 15) es necesario añadir a la tarea información de donde empieza y acaba el código de la tarea hijo (`base_chunk_pc_end` y `base_chunk_pc_start`). Este enfoque, aunque permite ahorrar memoria, presenta el inconveniente de tener que comprobar los límites a cada instrucción que ejecuta la tarea. La tabla de símbolos también se hereda, lo que permite compartir las variables entre las tareas. No obstante, pare evitar problemas de condición de carrera, se crea un nuevo nivel en la tabla de símbolos como se explica en el apartado “Tabla de símbolos”. A pesar de compartir el mismo almacén de variables en un momento dado, este puede divergir al cambiar el alcance de las tareas, impidiendo que se compartan nuevas variables cuya localidad es exclusiva a la tarea. Esto impide obtener el estado completo directamente desde el almacén global.

Además una tarea dispone de el método `run_next_step`, que ejecuta un paso de programa de la tarea; y los métodos `notify_*` que son usados por la ISA y el administrador de tareas para cambiar el estado de la tarea.

Como puede verse en el diagrama de clases, (ver figura 14) el administrador de tareas (`TaskManager`) dispone de una función, `RETCODE run(ISA ISA, State global_state, Function main_function)`, que inicia el bucle de ejecución principal de la máquina virtual para un programa. A esta función se le pasa el *bytecode* de la función principal del programa (`main_function`), el estado global del programa (`global_state`) y la ISA, en la que se define el conjunto de funciones que interpretan las instrucciones de *bytecode*. Pasar como argumento la ISA permite elegir distintos conjuntos de instrucciones y ofrece un mecanismo de retro-compatibilidad entre varias versiones del *bytecode.* A su vez, pasar el estado del programa como argumento en lugar de crear uno nuevo permite incluir variables y funciones predefinidas, así como consultarlo desde fuera del bucle de ejecución.

La función `run` devuelve un código de error definido en la enumeración `RETCODE`, indicando si ha habido errores durante la ejecución del programa. Cabe destacar que la detección de errores no siempre implica la terminación del programa, ya que no se trata de errores del propio programa, sino de errores en la ejecución de la máquina virtual, tales como el uso de registros vacíos en la instrucción de *bytecode* `ADD`.

La clase `TaskManager` define dos funciones abstractas a implementar, que serán utilizadas por la función `run`: la función `void init(Task main_task)`, que esta encargada de preparar el planificador para la ejecución de la tarea que se le pasa como argumento, y la función `RETCODE run_step()`, que debe ejecutar un paso de ejecución del programa y devolver o un código de error indicando si ha habido errores en la máquina virtual durante la ejecución del paso (`RETCODE.FAIL`), o bien un código que indica que no hay más pasos por realizar (`RETCODE.END`).

El administrador de tareas dispone de un mecanismo implementado por la clase abstracta `TaskManager` para registrar funciones que se ejecutarán antes y después de cada paso de programa. Este mecanismo permite implementar mecanismos de depuración, análisis e inspección, entre otros. Para registrar dichas funciones se hace uso de los métodos `regist_prestep` y `regist_poststep` (ver figura 14). Dichos métodos registran una función junto a una estructura de datos arbitraria, que sirve para pasar información entre distintas llamadas de la función o entre varias funciones que registren la misma estructura. Las funciones registradas reciben como parámetros el estado global del programa para poder consultarlo o modificarlo, así como la estructura de datos que se ha registrado junto a la función. Es por ello que deben tener la firma `void (*callback)(State* global_state, void* userdata)`.

Al registrar una función, esta devuelve un identificador. Dicho identificador sirve para poder eliminar funciones registradas en el administrador de tareas, para lo cual se emplea la función `void unregist_step_callback(int id)`. Finalmente, este mecanismo dispone de las funciones `get_prestep_ids()` y `get_poststep_ids()`, que devuelven una lista de los identificadores de las funciones actualmente registradas.

Se muestra a continuación, en la figura 16, un diagrama de secuencia donde se ilustra el comportamiento del bucle principal del programa manejado por el administrador de tareas.

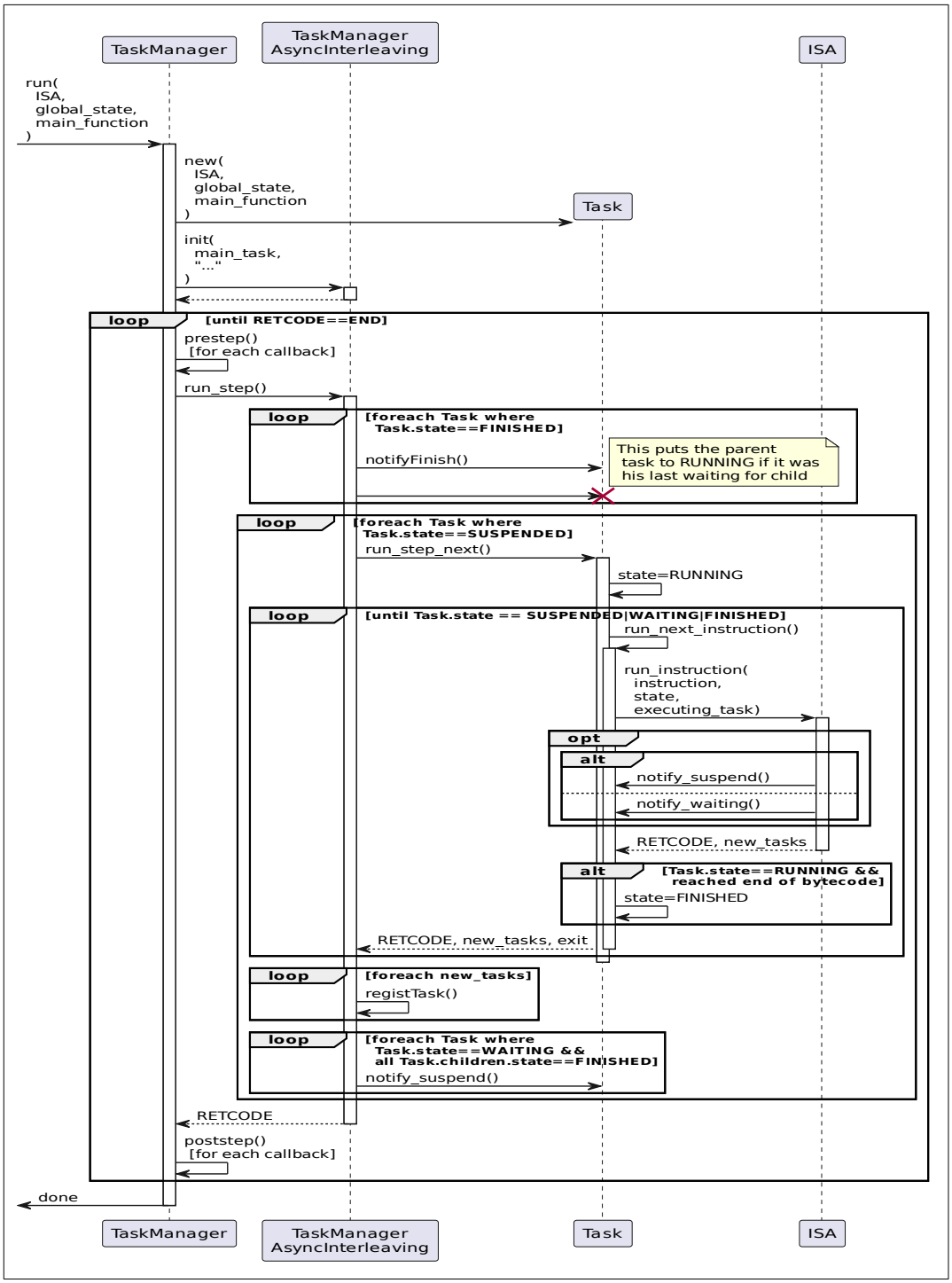

Figura 16: Diagrama de secuencia del bucle principal de la VM.

En este bucle el administrador de tareas, invoca a cada paso el método `run_step_next` que ejecuta un paso de programa para cada tarea. Este método, a su vez, entra en un bucle en el que ejecuta una instrucción hasta que la tarea cambia de estado o ejecuta su última instrucción. Para ello, la tarea escoge cada vez la instrucción de *bytecode* que debería ejecutar y se la pasa a la ISA junto a su estado en la llamada a la función `run_instruction`.

Compilador

Frontend

El analizador léxico se ha generado mediante la herramienta flex, a partir del archivo `alex.l` donde se define el analizador usando el lenguaje *lex*. Flex genera el analizador en el archivo de código fuente C `alex.c`.

El analizador generado es, fundamentalmente, un autómata finito creado a partir de la unión de las expresiones regulares que definen las palabras reservadas, identificadores y otros *tokens* del lenguaje, así como las acciones asociadas a estas expresiones que se encuentran en el archivo `asyn.l`.

Este analizador implementa la función `int yylex()` que devuelve en cada invocación el identificador del siguiente token leído y ejecuta las acciones asociadas al mismo. En estas acciones es común que se almacenen los identificadores directamente en la tabla de símbolos que se utiliza en fases posteriores de la compilación. No obstante, se considera que es posible no depender de la tabla de símbolos en esta fase y se pasa directamente el valor semántico del *token* al analizador sintáctico-semántico asignándolo a la variable `yylval`. Dicha variable no es definida por el analizador léxico, sino por el analizador sintáctico. Es por ello que, a la hora de compilar, es necesario que se incluya el archivo de cabecera generado para el compilador síntactico-semántico. Tambien cabe destacar la variable `yyin`, que es usada para especificar el archivo de codigo fuente a compilar [17].

El analizador síntactico-semántico se ha generado con la herramienta Bison a partir del archivo de `asyn.y`, donde se define el analizador mediante un lenguaje propio. Flex genera el analizador en el archivo de código fuente C `asyn.c` y el archivo de cabecera `asyn.h`.

El analizador generado es un autómata de pila en el que se implementa el algoritmo de análisis LALR. La implementación de este algoritmo hace uso de la función `yylex` implementada en el analizador léxico para recuperar el siguiente *token* a analizar.

La definición del analizador se realiza construyendo la gramática del lenguaje en forma BNF, en la cual cada producción de la gramática tiene asociada una regla. Estas reglas se utilizan para construir el AST. En cada una de ellas se construye un nodo del AST correspondiente al elemento de la producción analizado y se incluyen los nodos de cada uno de los elementos de la regla [17].

A continuación, en la figura 17 se muestra el diagrama de clases usado para representar el AST.

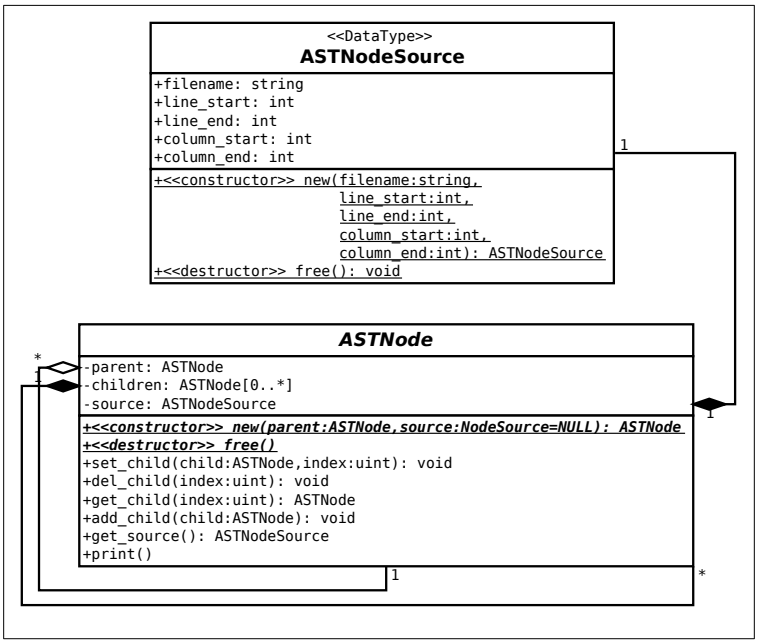

Figura 17: Diagrama de clases del AST

La clase `ASTNode` representa un nodo genérico del AST y contiene el nodo padre, los nodos hijos y los métodos para para manipular el AST. Además, puede tener opcionalmente la información del código fuente a partir de la cual se ha generado el nodo. Esta información esta representada en la clase `ASTNodeSource` y se compone del nombre del archivo de código fuente analizado y las líneas y columnas del texto correspondiente al nodo. Los atributos de esta última clase son solo de lectura, ya que esta información no cambia. La información del código fuente es opcional ya que, aunque puede ser útil para mostrar mensajes de error durante las fases del *frontend*, no es necesaria para la fases posteriores de compilación. Por ello, se ha separado en la clase `ASTNodeSource` en lugar de incluirla como atributos de la clase `ASTNode`, permitiendo así ahorrar memoria.

Se han creado clases por cada tipo de nodo que heredan de la clase ASTNode. Muchas de estas clases solo otorgan de tipo al nodo y no tienen información adicional, pero algunas incluyen atributos para representar los valores semánticos del nodo. Algunos de estos casos son la clases que representas constantes, que incluyen su valor; o las clases que representan expresiones aritméticas o lógicas, que incluyen el tipo de operación.

Backend

Para el diseño del *backend* del compilador se ha decidido usar el estilo de programación funcional. Al tratarse el *backend* de una cadena de transformaciones, se considera este paradigma adecuado ya que permite modelar esto fácilmente.

El módulo de optimización sobre AST presenta una función `optimize_*(AST* ast_in, AST* ast_out)` por cada tipo de optimización. Esta función genera un nuevo AST (ast_out) resultante de la optimización sobre el AST original (ast_in).

Debido a la naturaleza concurrente del proyecto no es posible aplicar ninguna transformación que suprima las acciones del programa o cambie el orden de sincronización de éstas. Aunque pueda conseguirse un programa equivalente resultante de dicha transformación, éste ya no sería de utilidad para analizar el comportamiento del programa, que es la funcionalidad de este intérprete. Es por ello que sólo se aplicará la optimización *constant folding*, aunque es posible que haya más optimizaciones aplicables.

El módulo de generación de código presenta una única función `code_gen(AST* ast, vm_bc_function** bytecode)` como interfaz. Esta función genera el *bytecode* a partir del AST que previamente ha sido optimizado.

Estructura de directorios

El proyecto sigue una estructura típica de un proyecto en C, mostrada en la figura 18. Como se ha mencionado anteriormente, al ser dos proyectos independientes, se emplea la misma estructura para la máquina virtual y el compilador.

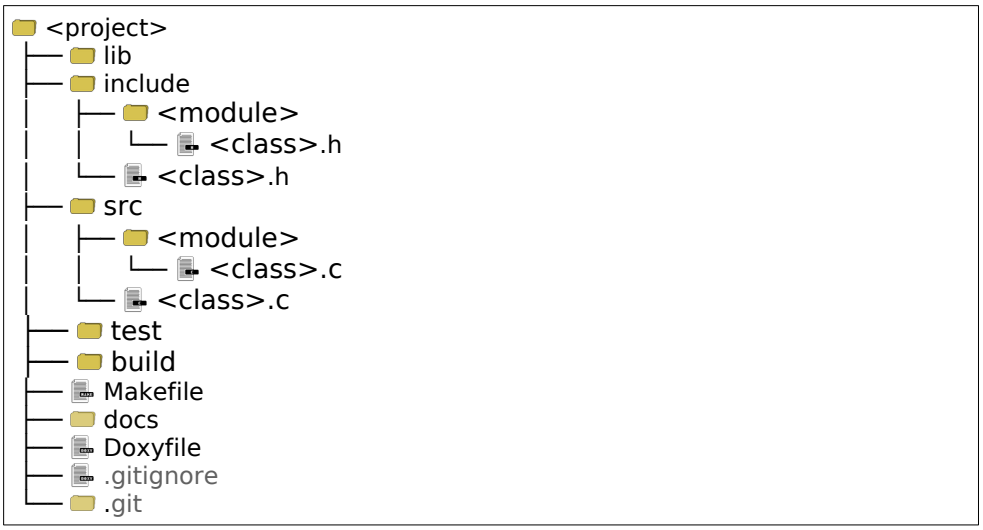

Figura 18: Estructura de directorios

Por regla general se implementa todo lo relativo a una única clase en dos archivos: uno de cabecera (.h) y uno de implementación (.c), ambos con el nombre de la clase. Así mismo, los archivos de clases pertenecientes a un módulo se encuentran en un directorio con el nombre de dicho módulo.

En los archivos de cabecera se encuentran las declaraciones de tipos y funciones; y la definición de enumeraciones, constantes y cualquier otra construcción necesaria

para emular cualquier atributo público de la clase, que será utilizado a su vez por otras clases. Estos archivos se encuentran dentro del directorio “include”.

En los archivos de implementación se encuentran la definición de los tipos, funciones y otras construcciones declaradas en los archivos de cabecera. También se encuentran declaradas y definidas las funciones privadas y auxiliares de la clase. Estos archivos se encuentran dentro del directorio “src”. Las librerías estáticas se encuentran en el directorio “lib”. El directorio “build” contiene los ficheros generados por el proceso de compilación. El directorio “test” contiene los test del proyecto.

En la raíz del proyecto se encuentran el archivo “Makefile”, que describe el proceso de compilación del proyecto, y el archivo “Doxyfile”, que contiene la configuración para la generación automática de documentación, que será generada en el directorio “docs”.

Además se autogenerará la carpeta oculta “.git”, que contiene los archivos necesarios para el sistema de control de versiones. En la raíz del proyecto se ubica el archivo “.gitignore”, que el sistema de versiones utiliza para saber los ficheros de los que realizar seguimiento.

5. Implementación

La implementación de este proyecto se ha llevado en C usando un paradigma orientado a objetos. Aunque C no es un lenguaje orientado a objetos, se ha emulado sus características siguiendo las técnicas presentes en [34].

A continuación se explica la implementación de algunas de las partes más complejas y relevantes.

5.1 Compilador

En este apartado se explica brevemente la implementación de los analizadores y de la generación de código, comentando los detalles más relevantes y ciertos problemas que han surgido durante el proceso.

Analizador léxico

La implementación del analizador léxico se encuentra en el archivo “alex.l” donde se definen el analizador léxico en el lenguaje *lex*. Este archivo sigue el siguiente formato:

```
1  declaraciones
2  %%
3  reglas de traducción
4  %%
5  funciones auxiliares
```

En la sección de declaraciones se definen expresiones regulares asociadas a un nombre en la forma `nombre expresión_regular`, para que puedan ser reutilizadas.

En la sección de reglas se definen las acciones que se realizarán al leer un *token.* Estas tienen la forma `expresión_regular {acción}`. Es importante destacar que en el caso de que hubiese más de una regla que coincidiese con el texto a analizar se usaría la regla que coincidiese con el texto de mayor longitud o, en el caso de la misma longitud, la regla que primero se hubiera definido. Es por ello que las dos primeras reglas que aparecen son para los comentarios y para los delimitadores entre *tokens* (espacios y tabuladores). Justo después se encuentran las reglas para los *tokens* que formen parte de la lista de palabras reservadas del lenguaje, teniendo así preferencia sobre las reglas para identificadores y constantes, que se encuentran a continuación. Por último, se encuentra una regla que coincide con cualquier carácter y que, al estar en último lugar, solo se ejecutará en el caso de que no haya coincidencia con las reglas anteriores. Está última regla sirve para identificar cualquier texto que no forme parte del léxico del lenguaje.

En estas reglas, las acciones, en el caso de comentarios y delimitadores, están vacías, causando que estos sean ignorados. En el caso de las palabras clave solo se devuelve el identificador del *token.* Así mismo, en el caso de los identificadores y constantes, además de devolver el identificador del *token*, el texto coincidente es copiado en la variable `yylval.str`, o es convertido a número y asignado a las variables

`yylval.int` o `yylval.real`, en el caso de tratarse de constantes numéricas. Para terminar, la acción de la última regla sencillamente muestra un mensaje de error señalando el carácter capturado y el número de línea donde se ha encontrado el texto inválido, provocando la terminación del programa.

En el apartado de funciones auxiliares se encuentra la definición de todas las funciones de usuario que se usan dentro de las acciones. Se ha definido una función para ayudar en la depuración de este analizador que debe ser utilizada en las acciones de todas las reglas. Esta función simplemente imprime el texto capturado y el nombre del *token* que se va a devolver en caso de que la variable `lex_debug` evalúe a *true*.

Para finalizar, se incluye un apartado delimitado por `%{` y `}%` al principio del archivo, que será copiado textualmente en el código fuente del analizador generado. En esta sección se incluyen todas las librerías usadas, entre las que cabe destacar el archivo de cabecera `asyn.h` generado por el analizador sintáctico-semántico, que incluye las constantes que representan el identificador de los *tokens* y en el que se define el tipo de la variable `yylval`.

Analizador Sintáctico-Semántico

La implementación de este modulo se encuentra en el archivo `asyn.y`. Se trata de un archivo de Bison que especifica la gramática en BNF y las acciones a realizar en cada regla. Se ha utilizado la convención [25][Cap 3.2] de escribir los símbolos no terminales en minúscula y los terminales en mayúscula.

Primeramente se han identificado los elementos no terminales básicos del lenguaje. Estos son: *`program`*, que es el elemento principal y forma todo el programa; *`procDecl`*, que representa la definición de una función; *`action`*, que representa las posibles acciones; *`tell`*, que actualiza una variable; *`ask`*, que consulta el valor de una variable; *`conditional`*, que implementa un salto condicional; *`localDecl`*, que declara una variable nueva como local; *`call`*, que invoca a una función; *`globalChoice`* y *`instaChoice`*, que implementan un salto condicional múltiple (a modo de *`switch`* en C), con la diferencia de que la evaluación de la guarda y la acción se realizan en el mismo instante de tiempo para *`instaChoice`*; y *`sequence`* y *`concurrent`*, que representan una serie de acciones a realizar de forma secuencial o concurrente. A partir de estos elementos se han ido desarrollando las reglas y los nuevos no terminales que se han necesitado.

Se han sacado extraído los elementos *`sequence`* y *`concurrent`* de la producción *`action`*, y se han representado en un nuevo no terminal *`actionList`*, para así evitar una posible recursividad derecha [25][Cap 3.3.3] y posible ambigüedad, de forma que la regla pasa de `action: action OP_SEQUENCE action` a `actionList: action OP_SEQUENCE actionList`.

Es importante hacer mención a los símbolos no terminales *`mathExp`* y *`logicExp`*, que representan operaciones aritméticas y lógicas.

Además, se ha establecido explícitamente el orden de precedencia para los operadores aritméticos ya que, si bien es posible escribir una gramática que respete la

precedencia implícitamente a partir de las reglas, daría como resultado una gramática y código muy complicados. Así pues, se ha optado por utilizar la directiva `%left` de Bison [25][Cap 5.3], de forma que tendrán precedencia por la izquierda las operaciones lógicas sobre las operaciones aritméticas, y las operaciones de multiplicación y división sobre las de suma y resta.

Una vez escrita la gramática, se ha puesto a prueba con los programas de ejemplo mencionados en el capítulo 3. Se han presentado conflictos *shift/reduce* en las producciones `tell: BRACKET_START_ varDeclaration BRACKET_END_ OP_TELL_` y `ask: BRACKET_START_ logicExp BRACKET_END_ OP_ASK_`, indicando que la gramática no es LALR(1). Esto se debe a que hay una ambigüedad entre *ask* y *tell* que no se resuelve hasta comprobar el último símbolo de las reglas `OP_ASK | OP_TELL` y no se puede resolver utilizando únicamente un *lookahead symbol*. Para solucionar esto se puede utilizar un generador GLR o refactorizar la gramática. Dado que las dos soluciones a este problema resultarían en un código demasiado complejo y el cliente no considera importante el orden de aparición de los símbolos, se ha modificado el lenguaje y el último símbolo se ha desplazado al primer lugar (i.e. `(X=1)!` → `!(X=1)!`)

En las acciones de las producciones de la gramática se retorna, asignándolo a la variable `$$`, un nodo del AST que representa el elemento de la parte derecha de la producción, y que también incluye los elementos de la parte izquierda que ya han sido analizados. El valor de estos elementos se encuentra en las variables `$n`, donde n hace referencia al número del elemento. A continuación, en la figura 19, se muestra a modo de ejemplo una de las producciones:

```
1   procDecl: funct_id BRACKET_START_ procDeclArgs BRACKET_END_ BLOCK_START_ actionList
BLOCK_END_
2       {
3         yacdsl__ast__node_source* source_info = NULL;
4         yacdsl__ast__node* ast_node = yacdsl__ast__proc_decl_node__new(NULL,
source_info);
5         yacdsl__ast__node__addChild(ast_node,$1);
6         yacdsl__ast__node__addChild(ast_node,$3);
7         yacdsl__ast__node__addChild(ast_node,$6);
8         $$=ast_node;
9       }
10        ;
```

Figura 19: Ejemplo de una producción para el analizador sintáctico-semántico

Generación del bytecode

La generación del *bytecode* está implementada en el archivo `code_generation.c`.

En la implementación se ha definido la función genérica `code_gen__generic(const yacdsl__ast__node* ast_node, struct code_gen_data* code_gen_data)`, que toma un nodo del AST y ejecuta la función específica para el tipo del nodo que lleva a cabo la traducción a *bytecode*. La función genérica es, a su vez, invocada sobre los nodos hijos por la función especifica.

Cabe destacar la necesidad de compartir cierta información entre las funciones de generación de los distintos nodos. Esta información se pasa a través de la estructura `code_gen_data`, y tiene información como la función de *bytecode* sobre la que se estaba trabajando, el registro en el que se había colocado el valor resultante de la expresión de un nodo, el último registro libre para evitar sobrescribir registros, o una tabla de símbolos (no confundir con la tabla de símbolos de la máquina virtual) para tener un seguimiento de las constantes incluidas en la lista de constantes de la función de *bytecode* y así evitar constantes duplicadas.

A continuación, se muestra y explica el *bytecode* generado para las construcciones más importantes y complejas del lenguaje, como son la declaración de una función (figura 20), una asignación (figura 21), y un *GlobalChoice* (figura 22).

.

```
(NEW_FUNCTION):
  // crea un nuevo ámbito añadiendo un nuevo nivel al stack
  CREATE_BLOCK
  // Carga el valor que haya en la lista de argumentos de la llamada en un
  // registro, crea una variable local con el identificador del argumento,
  // y asigna la referencia de este registro a esta variable
  LOADARG R(valor_del_argumento) INT(N)
  SETLOCAL CTE(nombre_del_argumento) R(valor_del_argumento)
  SETREF CTE(nombre_del_argumento) R(valor_del_argumento)
  // repite estas 3 instrucciones por cada argumento, donde N es
  // el numero del argumento
  […] (resto de argumentos ^)
  […] (cuerpo de la función)
  // cierra el ámbito actual y vuelve al anterior
  DELETE_BLOCK
// Carga la función creada en un registro,
// y asigna este a una variable nueva con el nombre de la función dentro
// de la función donde se ha declarado.
(PARENT_FUNCTION):
  LOADF R(new_function) CTE(NEW_FUNCTION)
  SET CTE(new_function_id) R(new_function)
```

Figura 20: Bytecode de una función

.

```
// Carga en un registro el valor de la parte derecha de la asignación
// si se trata de una constante
// crea una nuevo objeto en el registro con el valor de la constante
LOADK CTE(valor_del_rvalue) R(valor_del_rvalue) -
// si se trata de una variable
// carga en el registro el objeto de la variable del lado derecho
LOAD CTE(nombre_variable_rvalue) R(valor_del_rvalue) -
// si se trata de una expresión genera el bytecode de la expresión
// y usa el último registro usado, donde se encuentra el resultado
// de la expresión
[…] (bytecode de la expresión)
// en cualquier caso, copia valor del objeto del registro (rvalue)
// a un nuevo objeto que es asignado al nombre de la variable
SET CTE(nombre_variable_lvalue) R(valor_del_rvalue) -
// Indica que esto se ha hecho en un paso de ejecución, ha acabado
//y la tarea actual debe suspenderse hasta el siguiente paso
SUSPEND - - -
```

Figura 21: Bytecode para una asignación (tell)

.

```
LABEL(comprobar_guardas):
// Genera el bytecode para las guardas de todas las ramas de la elección
// se asume que se guardan los registros del resultado de las guardas
// para su posterior utilización
[…] (bytecode de la evaluación de las guardas (ask))
// marca un paso de ejecución porque ya se han evaluado las guardas
SUSPEND - - -

// comprueba si todas las guardas son falsas, para ello se emplea un and
// cortocircuitado de la evaluación de las guardas
// Si una guarda es cierta se salta a la elección de la acción,
//si todas las guardas son falsas se salta al principio del Choice
EQ R(eval_guardas) 0 0
MULT R(eval_guardas) R(eval_guardas) R(eval_guarda_N)
JMPZ R(eval_guardas) LABEL(comprobar_guardas) -
JMP  LABEL(eleccion_acción)
// se repite estas 3 ultimas instrucciones por cada guarda, donde N
// es el numero de guarda
[…] (resto de guardas )

LABEL(eleccion_acción):
// elije una acción al azar por la que se empieza a evaluar
RAND R(acción_seleccionada) 0 N-1
LOADK R(indice_acción) N
EQ R(salto_acción) R(acción_seleccionada) R(indice_acción)
JMPZ R(salto_acción) 2
// se repite estas 3 ultimas instrucciones por cada acción, donde N
// es el numero de acción
[…] (resto de acciónes ^)

// Por cada acción, comprueba si la evaluación de su guarda se cumple,
// si es así ejecutala y salta al final del choice, si no continua
// comprobando de forma circular la siguiente acción.
// Aunque es un bucle infinito si no se cumple ninguna, existe la garantía
// de que eso no pasa ya que se ha comprobado anteriormente
LABEL(acción_N):
JMPZ R(eval_guarda_N) LABEL(fin_acción_N) -
[…] (bytecode de la acción)
JMP LABEL(END) -
// Se repite este trozo de bytecode por cada acción
[…] (resto de acciones )
JMP LABEL(acción_1) - -
LABEL(END):
```

Figura 22: Bytecode para el GlobalChoice

5.2 Máquina virtual

En este apartado solo se explican las implementaciones concretas que se han elegido para el *garbage collector* y el administrador de tareas.

Garbage Collector

El *garbage collector* implementado usa el algoritmo ARC (*Automatic Reference Counting*). Para ello, inserta un contador de referencias en los objetos la primera vez que son referenciados. Cada vez que se referencia un objeto con la función `ref(Object`

`object)` se incrementa este contador, y cada vez que se elimina una referencia con la función `deref(Object object)` se decrementa. Si al eliminar una referencia el contador quedase a 0, se invocaría el destructor del objeto.

Este *garbage collector* presenta el problema de que puede provocar *memory leaks* si hubiese referencias circulares.

Lo habitual sería que el administrador de tareas llamase a la función `run()` del *garbage collector* a cada paso de programa, pero dado que esta función está vacía en esta implementación, el *garbage collector* únicamente se ejecuta al invocar las funciones de referencia.

Administrador de tareas

Para la implementación del administrador de tareas se ha decidido usar un planificador con un solo hilo de ejecución, en el que la ejecución de las tareas del programa es intercalada.

Como se ha dicho en el capítulo 4. Diseño, el bucle de ejecución se divide en pasos de programa. Estos pasos de programa son determinados en el momento en el que no haya ninguna tarea en el estado `RUNNING`.

Las tareas cambian de estado cuando sucede algún evento. Estos eventos se notifican con los métodos `notify_suspend()`, `notify_wait()` o `notify_finish()`. Estos eventos ocurren al ejecutar la instrucción de *bytecode* `SUSPEND` que suspende la tarea, al ejecutar la instrucción `TASK_WAIT` que la pone en espera, o bien cuando la tarea ejecuta su última instrucción.

El planificador implementado se basa en 4 listas: una para las tareas listas para su ejecución, una para las tareas suspendidas, una para las tareas que están esperando a la finalización de sus tareas hijas y una para las tareas que ya han finalizado.

En cada paso de programa, lo primero que hace el planificador es comprobar si hay tareas que estaban en espera y deben pasar a considerarse listas para su ejecución tras la finalización de sus tareas hijas.

Para conseguir esto se recorre la lista de las tareas terminadas. En la tarea padre de cada una de las tareas de esta lista se decrementa el contador de tareas en espera (`waiting_tasks`) y se cambia su estado a `RUNNING`. Seguidamente, se borran las tareas de esta lista.

A continuación, se debe recorrer la lista de tareas en espera, extrayendo las tareas que estén en el estado `RUNNING` y encolándolas en la lista de tareas a ejecutar. De esta forma, las tareas que pasan a estar listas se ejecutan en el mismo paso de programa, puesto que si se encolaran después de recorrer la lista de tareas a ejecutar estas emplearían un paso de programa vacío adicional.

Seguidamente, se recorre la lista de tareas a ejecutar. Para cada tarea se invoca el método `run_next_step`, que ejecutará el siguiente paso de programa de la tarea. Este método devuelve una lista de tareas nuevas creadas durante un paso de ejecución y que deben ejecutarse en el mismo paso de programa, y que se añaden a la lista de tareas a ejecutar. Seguidamente, se elimina la tarea ejecutada de esta lista y se encola en la lista correspondiente según su estado.

Por último, se intercambian las listas de tareas suspendidas y tareas ejecutadas, de forma que las tareas que ya han realizado el paso de programa actual se ejecuten en el siguiente paso de programa.

6. Implantación

En este apartado se explica la compilación, instalación y uso del intérprete.

6.1 Compilación

Para compilar el proyecto hace falta satisfacer las dependencias de compilación. Este programa necesita de la herramienta de construcción make, el compilador de C gcc, las librerías estándar de C, y los archivos de desarrollo para la librería glib versión 2.0. Estas dependencias pueden ser instaladas en un sistema Linux mediante la invocación del siguiente comando:

```
~ $ apt install build-essentials make libglib2.0-dev
```

Para compilar el proyecto, hay que compilar la máquina virtual y el compilador respectivamente. Para ello, desde la carpeta raíz de cada subproyecto, hay que ejecutar la herramienta “make” con el argumento “vm” y “compiler” respectivamente. Es aconsejable invocar previamente la herramienta “make” con el argumento “clean” para evitar artefactos de compilación antiguos que pudiesen causar errores:

```
~/ TFG/vm $ make clean; make vm
```

```
~/ TFG/compiler $ make clean; make compiler
```

Una vez el proceso de compilación haya acabado, se encontrara en el directorio “build” dentro de cada subproyecto los ejecutables “compiler” y “vm”.

6.2 Instalación y Uso

Para el uso de los binarios compilados hace falta satisfacer las dependencias para la librería glib. Para instalarlas se puede usar el siguiente comando:

```
~ $ apt install libglib2.0-0
```

Para ejecutar un programa en el lenguaje objetivo, primero, hay que compilarlo a *bytecode.* Para ello se debe llamar al ejecutable “compiler” pasando como argumento el nombre del archivo de código fuente que contiene el programa, y el nombre con el que se guardará el archivo de *bytecode* resultante:

```
~/TFG/compiler/build $ compiler "../src/test/programs/program_mult.vm"
"~/program_mult.vmbc"
```

Seguidamente, hay que ejecutar el *bytecode* resultante en la máquina virtual. Para ello ha que llamar al ejecutable “vm” con el nombre del archivo de *bytecode* del programa:

```
~/TFG/vm/build $ vm "~/program_mult.vmbc"
```

Se mostrara en la salida estándar, una traza de la ejecución del programa mostrando el estado a cada paso de programa.

7. Validación

Se han realizado varias pruebas para verificar el buen funcionamiento del intérprete y validar que cumple con las especificaciones sintácticas y semánticas del lenguaje objetivo. Además se ha realizado un *benchmark* para comprobar su rendimiento.

Para realizar estas pruebas se han creado tres conjuntos de programas de prueba en el lenguaje objetivo: programas de ejemplo sencillos que han sido suministrados por el cliente, programas sintácticamente no válidos y programas mínimos para comprobar cada construcción del lenguaje. Estos últimos se han ideado para analizar una única construcción del lenguaje, es por ello que se intenta utilizar solamente dicha construcción y minimizar el uso de otras construcciones necesarias para que formen un programa válido. En la figura 23 se encuentra el ejemplo para comprobar la aceptación del buen funcionamiento de la construcción *Concurrence*. Se puede ver que el programa se centra en esta construcción pero incluye construcciones como la declaración de procedimientos o la acción *Update* para que el programa sea valido.

```
1  proc2(ret) {
2       (!(x=11)! ; !(x=12)!) ||
3       (!(y=21)! ||
4                  (!(y=22)!  ;  !(y=23)!)) ||
5        !(z=33)!
6  }
7  [ proc2(ret) ]
```

Figura 23: Programa de test para la construcción Concurrence

Todas las pruebas han sido realizadas manualmente. Se ha decido no automatizarlas ya que escribir estas requiere mucho tiempo y son de poco valor en un sistema, como es el de un intérprete, en el que el comportamiento de sus clases y componentes es altamente interdependiente. También resulta complejo, ya que para un mismo programa de entrada existen múltiples resultados intermedios y trazas de ejecución válidos. Por ejemplo, en el caso del compilador es posible generar muchos AST y *bytecode* válidos; comprobar la validez de estos no es trivial, ya que es posible que la única forma de validarlos sea comprobar que su interpretación y ejecución es correcta. En el caso de probar la máquina virtual, no es posible suministrar cualquier *bytecode* ya que no todos los *bytecode* posibles son válidos, y validar estos vuelve a llevar al punto de partida.

Es por todo esto que para comprobar el buen funcionamiento de las clases y componentes se han realizado múltiples pruebas *ad-hoc* con la ayuda del depurador. Así mismo para comprobar el buen funcionamiento de la totalidad del sistema se han realizado dos tipos de pruebas exploratorias. El primero comprueba el cumplimiento de la especificación formal de la sintaxis del lenguaje objetivo, y el segundo el de la especificación semántica.

En el primer tipo de prueba se pasan los programas de prueba al compilador y solo se comprueba si son aceptados o no por este. En el caso de los programas de prueba inválidos se acepta la prueba si el compilador falla con el código de error que se usa para indicar que un programa contiene errores de sintaxis. En el caso de los programas válidos se comprueba que no devuelva este código de error, aunque el compilador falle con cualquier otro error, como puede ser SIGFAULT (error relacionado con violación de la memoria), se considera la prueba aceptada.

En el segundo tipo de prueba solo se toman los programas de prueba válidos. Estos se pasan al compilador, y el *bytecode* generado a la máquina virtual. En estas pruebas se ha comprobado junto al cliente que el flujo de ejecución y las transiciones de estado de la ejecución los programas de ejemplo satisfacen la especificación.

La realización de estas pruebas ha permitido descubrir errores de diseño en el compilador relativos a la sintaxis, y errores de diseño en la máquina virtual derivados de una mala interpretación de la especificación semántica.

En el *benchmark* se ha comparado el rendimiento temporal de nuestro intérprete respecto al intérprete de pyhton. Se ha elegido este para comparar porque es un lenguaje interpretado.

program	Cpython (compiler)	Cpython (vm)	Objective language (compiler)	Objective language (vm)
program_mult	real 0m9,412s user 0m6,602s sys 0m2,635s	real 0m7,201s user 0m4,927s sys 0m2,181s	real 0m5,610s user 0m1,286s sys 0m3,117s	real 0m4,459s user 0m1,347s sys 0m1,537s
program_microwave	real 0m6,412s user 0m3,637s sys 0m2,875s	real 0m6,001s user 0m3,821s sys 0m2,180s	real 0m4,842s user 0m1,986s sys 0m2,312s	real 0m3,439s user 0m1,092s sys 0m1,473s

Tabla 1: Benchmark temporal del compilador

Para realizar este *benchmark* se han tomado dos programas el conjunto de programas de ejemplo y se han reescrito en python. Se ha medido, con la utilidad de Linux `time`, el tiempo de ejecución de $1e^6$ iteraciones del compilador e máquina virtual de Cpython y nuestro intérprete para cada uno de estos programas. Se onsidera que el intérprete desarrollado tiene un rendimiento aceptable si los tiempos de ejecución no son, en general, dos veces mayores a los de Cpython. En la tabla 1 se pueden ver los resultados del *benchmark*, que indican que el intérprete tiene buen rendimiento.

8. Conclusiones

En este capítulo se detallan los objetivos del proyecto logrados, los conocimientos adquiridos a lo largo de la realización este proyecto y los principales problemas encontrados, así como posibles ampliaciones del trabajo. Finalmente, se comentará la relación del trabajo desarrollado con las asignaturas cursadas en el grado de ingeniería informática.

8.1 Conclusiones finales

Tras comprobar los resultados, es razonable afirmar que el proyecto se ajusta a todos los objetivos planteados. Se ha logrado el objetivo principal, la construcción de un intérprete altamente extensible sobre el que se pueden aplicar diversas técnicas de análisis y verificación. De hecho, se ha probado el uso del intérprete para la verificación de un pequeño proyecto y ha permitido detectar un problema de condición de carrera. Es por ello que este proyecto cuenta con el potencial necesario para ayudar en la verificación de sistemas concurrentes. También se ha logrado el objetivo de documentar el proceso de creación de un intérprete con la elaboración de esta memoria.

Para la realización del presente trabajo se ha tenido que profundizar en el conocimiento de lenguajes concurrentes, especialmente del lenguaje formal CSP[8] y sus derivados, ya que el lenguaje para el que se ha diseñado e implementado el intérprete está basado en CSP. Se ha comprobado que este tipo de lenguajes no pueden detectar y solucionar todos los problemas relacionados con la concurrencia, ya que parte de ellos se derivan de los problemas que resuelven los programas. Incluso es posible la aparición de *deadlocks* a pesar de que estos lenguajes no usan *locks*, ya que los primeros son inherentes al uso de la concurrencia. A pesar de ello, sí que es posible detectar una gran parte de los problemas derivados de la concurrencia, aun cuando el propio intérprete es susceptible a estos.

Adicionalmente, para el desarrollo del proyecto ha sido necesario profundizar en el ámbito de los compiladores, especialmente en el uso de generadores de analizadores sintácticos. Se han encontrado problemas al elegir el tipo de analizador, ya que el que en un principio parecía adecuado no se ajustaba a la sintaxis del lenguaje. Para solucionar dicho problema se ha optado por modificar ligeramente la sintaxis del lenguaje objetivo en vez de cambiar de analizador, por simplicidad.

Así mismo, se ha profundizado en la implementación de intérpretes, inspirándose en el trabajo realizado para intérpretes de lenguajes comúnmente utilizados, como puede ser Python. Para ello, se han investigado los diseños principales de estos intérpretes y las ventajas y desventajas de cada uno de estos diseños.

El presente trabajo también ha permitido aprender técnicas para emular orientación a objetos en C, así como profundizar en el conocimiento de este lenguaje.

El principal problema encontrado durante el desarrollo del proyecto ha sido la necesidad de rediseñar parcialmente el intérprete en diferentes puntos del proceso, tras comprobar que no se ajustaba a la semántica del lenguaje. No obstante, esto ha permitido obtener una mejor comprensión de los requisitos del lenguaje.

Uno de los principales equívocos fue el entender erróneamente que durante un paso de programa, dada la modificación o utilización de un recurso por varias tareas concurrentes, el estado del recurso podía cambiar entre la finalización del paso de una tarea y el principio del paso de otra que se dieran en un mismo paso de programa. Como consecuencia, fue necesario cambiar como se trataba la tabla de símbolos y a modificar la implementación del planificador de tareas. Elegir otro plan de trabajo en el que se hubiese reportado más frecuentemente con el cliente podría haber evitado este problema.

Finalmente, se espera que el presente trabajo sirva de utilidad a la hora de analizar y verificar sistemas concurrentes. Especialmente, su utilización como *framework* para la aplicación de técnicas de análisis en tiempo de ejecución ya existentes.

8.2 Trabajo futuro

El intérprete obtenido tiene unas prestaciones muy básicas y hay ciertos aspectos que hubiese sido conveniente poder mejorar en caso de disponer del tiempo necesario. El planificador actual, aunque concurrente, no es paralelo, cuando esta característica podría ahorrar tiempo de ejecución en procesadores multinúcleo. Por ello, se propone como trabajo futuro la implementación de un planificador capaz de ejecutar tareas en paralelo haciendo uso, por ejemplo, de *thread pools*.

También se propone una implementación del *garbage collector* que implemente el algoritmo *mark and swap,* ya que el algoritmo actual de contabilidad de referencia automática (ARC) no soporta referencias cíclicas.

Para terminar, se propone ampliar el intérprete para soportar no solamente el modelo imperativo del lenguaje, sino también el declarativo. Para ello, se podría insertar un nuevo tipo de objeto que, asignado a una variable, representara una restricción que debe ser satisfecha por la misma, modificando únicamente la generación y algunas operaciones del *bytecode* para la evaluación de la restricción.

8.3 Relación con los estudios cursados

El trabajo desarrollado está directamente relacionado con la asignatura “Concurrencia y sistemas distribuidos”, dado que se ha implementado un intérprete para un lenguaje concurrente. Esta asignatura ha aportado el conocimiento teórico suficiente para identificar los posibles problemas de concurrencia que pueden aparecer en dichos programas y en el intérprete que los ejecuta. Es por ello que el presente proyecto supone un complemento práctico idóneo para consolidar y profundizar en los conocimientos adquiridos.

La asignatura de “Lenguajes, tecnologías y paradigmas de la programación” cobra especial relevancia en relación a este proyecto, ya que se han empleado diferentes paradigmas de programación en función de las necesidades de cada parte. Además, el hecho de haber cursado dicha asignatura ha permitido disponer de las herramientas necesarias para definir y comprender el lenguaje objetivo implementado.

La asignatura de “Lenguajes de programación y procesadores de lenguajes” ha aportado el conocimiento necesario para poder procesar y analizar el código fuente del lenguaje objetivo y transformarlo a *bytecode.* También ha aportado el dominio de técnicas de optimización a la hora de compilar el lenguaje. Del mismo modo, la asignatura “Análisis, validación y depuración de software” abarca técnicas de análisis estático relacionadas con la optimización del código.

Las fases de compilación usadas en el *frontend* del intérprete se explican en las asignaturas de “Lenguajes, tecnologías y paradigmas de la programación” y “Lenguajes de programación y procesadores de lenguajes”.

En última instancia, este trabajo se relaciona indirectamente con la asignatura “Estructura de computadores”, ya que se han puesto en práctica los conceptos básicos de la arquitectura de un computador para modelar la máquina virtual.

Bibliografía

[1] D. A. Peled. *Software Reliability Methods* . Springer, 2001.

[2] T. R. Weiss, "Out-of-memory problem caused Mars rover's glitch" [online]. Available: https://www.computerworld.com/article/2574759/out-of-memory-problem-caused-mars-rover-s-glitch.html. [Accessed Aug. 2002].

[3] N. G. Leveson, C. S Turner, "*An investigation of the Therac-25 accidents.*", IEEE Computer, vol. 26, pp. 18-41, no. 7, July 1993.

[4] Wikipedia, "Combinatorial explosion" [online]. Available: https://en.wikipedia.org/wiki/Combinatorial_explosion. [Accessed Aug. 2002].

[5] M. Alpuentea, M.M. Gallardob, E. Pimentelb, A. Villanueva, "*An Abstract Analysis Framework for Synchronous Concurrent Languages based on source–to–source Transformation*", Electr. Notes Theor. Cmput. Sci, vol. 206, pp. 3-21, no. 1, Abril 2008.

[6] G. J. Holzmann, "*The model checker SPIN*", IEEE Transactions on Software Engineering, vol. 23, pp. 279-295, no. 5, May 1997.

[7] SPIN authors, "Inspiring Applications of Spin" [online]. Available: https://spinroot.com/spin/success.html. [Accessed Aug. 2002].

[8] C. A. R Hoare , "*Communicating sequential processes*", Communications of the ACM, vol. 21, pp. 666-677, no. 8, 1978.

[9] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[10] G. Gopalakrishnan, R. M. Kirby, "*Formal Methods for MPI Programs*", Electronic Notes in Theoretical Computer Science, vol. 193, pp. 19-27, no. , 2007.

[11] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, A.W. Roscoe, "*FDR3 --- A Modern Refinement Checker for CSP*", , vol. , pp. , no. , 2014.

[12] K. L Wrench, "*CSP-i: An implementation of CSP*", Software: Practice and Experience, vol. 18, pp. 545-560, no. 6, 1988.

[13] A. Lescaylle, A. Villanueva, "*The tccp Interpreter*", Electronic Notes in Theoretical Computer Science, vol. 258, pp. 63-77, no. 1, Dec 2009.

[14] SPIN authors, "Promela language reference" [online]. Available: http://spinroot.com/spin/Man/promela.html. [Accessed Aug. 2002].

[15] R. Nystrom. *Crafting Intepreters*. Genever Benning , 27 Jul 2021.

[16] Y. Shi, k. Casey, M. A. Ertl, D. Gregg, "*Virtual machine showdown: Stack versus registers*", ACM Trans. Archit. Code Optim., vol. 4, pp. 36, no. 4, Enero 2008.

[17] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1986.

[18] S. Sarda, M. Pandey. *LLVM Essentials*. Packt Publishing , 2015.

[19] T. Lindholm, F. Yellin, G. Bracha, A. Buckley. *The Java Virtual Machine Specification*. Addison-Wesley, 2014.

[20] , "The Computer Language Benchmarks Game" [online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/simple-cpu.html. [Accessed Aug. 2002].

[21] Wikipedia, "Languages by memory management type" [online]. Available: https://en.wikipedia.org/wiki/List_of_programming_languages_by_type#Languages_by_memory_management_type. [Accessed Aug. 2002].

[22] Wikipedia, "Languages supporting concurrent programming" [online]. Available: https://en.wikipedia.org/wiki/Concurrent_computing#Languages_supporting_concurrent_programming. [Accessed Aug. 2002].

[23] B. Kernighan, D. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[24] J.M Benedí, V. Gibster, L. Moreno, E. Vivancos. *Procesadores de lenguajes: Una introducción a la fase de análisis*. UPV, 2008.

[25] Free Software Foundation, Inc, "Bison 3.7.6" [online]. Available: https://www.gnu.org/software/bison/manual/bison.html. [Accessed Aug. 2002].

[26] , "LALR vs LL" [online]. Available: https://stackoverflow.com/questions/12170869/lalr-vs-ll-parser. [Accessed Aug. 2002].

[27] LLVM developers, "LLVM Language Reference Manual" [online]. Available: https://llvm.org/docs/LangRef.html. [Accessed Aug. 2002].

[28] R. M. Stallman, R. Pesch, S. Shebs. *Debugging with GDB: The GNU Source-Level Debugger*. GNU Press, 1993.

[29] Valgrind developers, "Valgrind Documentation" [online]. Available: https://valgrind.org/docs/manual/valgrind_manual.pdf. [Accessed Aug. 2002].

[30] Dimitri van Heesch, "Doxygen manual" [online]. Available: https://www.doxygen.nl/files/doxygen_manual-1.9.3.pdf.zip. [Accessed Aug. 2002].

[31] J. Garner. *GIT: The Ultimate Guide for Beginners: Learn Git Version Control* . Independiente, 2020.

[32] E. A. Hauck, B. A. Dent, "Burroughs' B6500/B7500 stack mechanism" in Proceedings of the April 30--May 2, 1968, spring joint computer conference, Ny, 1968, pp.245-251.

[33] , "Python: Global Interpreter Lock" [online]. Available: https://wiki.python.org/moin/GlobalInterpreterLock. [Accessed Aug. 2002].

[34] A. Schreiner. *Object-oriented Programming with Ansi-C*. Independent, 2011.

Anexo

Anexo A: Tabla de instrucciones del bytecode

Instrucción	Tipo	Campos	Función
NOOP	iNone	X X X	-
SET	iABx	R(A) CTE(Bx) X	CTE(Bx) := R(A)
	Busca en el ámbito actual y los superiores la variable con nombre CTE(Bx) y establece su valor a partir del valor del objeto en R(A). El objeto referenciado anteriormente por dicha variable es de-referenciado en uno y marcado para su eliminación si no tiene más referencias. Si la variable no existe, se crea en el ámbito actual.		
SETLOCAL	iABx	R(A) CTE(Bx) X	CTE(Bx) := R(A)
	Busca en el ámbito actual la variable con nombre CTE(Bx) y establece su valor al valor a partir del objeto en R(A). El objeto referenciado anteriormente por dicha variable es de-referenciado en uno y marcado para su eliminación si no tiene más referencias. Si la variable no existe, se crea en el ámbito actual.		
SETGLOBAL	iABx	R(A) CTE(Bx) X	CTE(Bx) := R(A)
	Busca en el ámbito global la variable con nombre CTE(Bx) y establece su valor al valor a partir del objeto en R(A). El objeto referenciado anteriormente por dicha variable es de-referenciado en uno y marcado para su eliminación si no tiene más referencias. Si la variable no existe, se crea en el ámbito global.		
SETREF	iABx	R(A) CTE(Bx) X	CTE(Bx) := &R(A)
	Busca en el ámbito actual y los superiores la variable con nombre CTE(Bx) y establece como el objeto en R(A), es decir, se asigna su referencia. El objeto referenciado anteriormente por dicha variable es de-referenciado en uno y marcado para su eliminación si no tiene más referencias. Si la variable no existe, se crea en el ámbito actual		
UNSET	iABx	X CTE(Bx) X	CTE(Bx) := /
	Busca en el ámbito actual y los superiores la variable con nombre CTE(Bx) y la de-referencia y marca para su eliminación si no tiene más referencias.		
LOAD	iABx	R(A) CTE(Bx) X	R(A) := &CTE(Bx)
	Busca en el ámbito actual y los superiores la variable con nombre CTE(Bx) y la carga en el registro R(A).		
LOADK	iABx	R(A) CTE(Bx) X	R(A) := CTE(Bx)
	Crea un nuevo objeto en el registro R(A) a partir de la constante de la lista de constantes con índice CTE(Bx).		
LOADF	iABx	R(A) CTE(Bx) X	R(A) := CTE(Bx)

	Crea un nuevo objeto en el registro R(A) a partir de la función de la lista de funciones con índice CTE(Bx).		
COPY	iABx	R(A) R(B) X	R(A) := *R(B)
	Crea una copia del objeto del registro R(B) en el registro(A)		
MOVE	iABx	R(A) R(B) X	R(A) := R(B)
	Mueve el objeto en el registro R(B) al registro R(A), dejando el registro R(B) vacío.		
SET_STRUCTMEMBER	iABC	R(A) R(B) INT(C)	R(B)[INT(C)] := R(A)
	Si el objeto en el registro R(A) es un array, se asigna el objeto en R(B) al array en la posición INT(C). Si hubiese anteriormente un objeto en esa posición del array, este es de-referenciado.		
LOAD_STRUCTMEMBER	iABC	R(A) R(B) INT(C)	R(A) := R(B)[INT(C)]
	Si el objeto en el registro R(B) es un array, se asigna el objeto en la posición INT(C) de este, en el registro R(A).		
CREATE_BLOCK	iNone	X X X	Crea un nuevo ámbito
DEL_BLOCK	iNone	X X X	Elimina el ámbito actual
TASK	iAsBx	INT(A) INT(sBx) X	PC:=PC+INT(B)
	Crea una nueva tarea que comprende las INT(sBx) instrucciones siguientes, y la marca para su ejecución inmediata. Además, realiza un salto relativo de INT(sBx) instrucciónes, dejando el contador de programa en la instrucción siguiente a la última que comprende la tarea.		
WAIT_TASK	iNone	X X X	
	Espera a todas las tareas que se han creado para la tarea actual.		
SET_ARG	iABx	R(A) INT(Bx) X	argv[INT(Bx)] := R(A)
	Asigna el objeto en el registro R(A) como la lista de argumento posicional INT(Bx) para pasarlo a la próxima llamada a función (CALL). Cabe destacar que solo hay una lista de argumentos compartida por todas las funciones de todas las tareas.		
LOAD_ARG	iABx	R(A) INT(Bx) X	R(A) := argv[INT(Bx)]
	Carga en el registro R(A) el objeto del argumento posicional INT(Bx) de la lista de argumentos.		
CALL	iABx	R(A) X X	R(A)(argv)
	Si el objeto en R(A) es una función, esta es invocada. Se restablece el contador de programa a 0, y se cambia la lista de instrucciones.		
JMP	iAsBx	X INT(sBx) X	PC := PC+INT(Bx)
	Salto incondicional. Salta tantas instrucciones como indique		

	INT(sBx). Puede ser un salto tanto hacia delante como hacia atrás. No se garantiza el correcto funcionamiento si se salta más haya de los limites de la lista de instrucciones actual.		
JMPZ	iABx	R(A) INT(sBx) X	PC := PC+INT(Bx)
	Salto condicional. Salta tantas instrucciones como indique INT(sBx), si el objeto en R(A), evaluá a verdadero. Puede ser un salto tanto hacia delante como hacia atrás. No se garantiza el correcto funcionamiento si se salta más haya de los limites de la lista de instrucciones actual.		
ADD	iABC	R(A) R(B) R(C)	R(A) := R(B)+R(C)
	Realiza una operación de suma con los valores de los objetos R(B) y R(C), y asigna su resultado a un nuevo objeto que será asignado al registro R(A). Esta operación se usa también para usar la operación OR en tipos booleanos.		
SUB	iABC	R(A) R(B) R(C)	R(A) := R(B)-R(C)
	Realiza una operación de resta con los valores de los objetos R(B) y R(C), y asigna su resultado a un nuevo objeto que será asignado al registro R(A).		
MULT	iABC	R(A) R(B) R(C)	R(A) := R(B)*R(C)
	Realiza una operación de multiplicación con los valores de los objetos R(B) y R(C), y asigna su resultado a un nuevo objeto que será asignado al registro R(A). Esta operación se usa también para usar la operación AND en tipos booleanos.		
DIV	iABC	R(A) R(B) R(C)	R(A) := R(B)/R(C)
	Realiza una operación de división con los valores de los objetos R(B) y R(C), y asigna su resultado a un nuevo objeto que será asignado al registro R(A).		
MOD	iABC	R(A) R(B) R(C)	R(A) := R(B)%R(C)
	Realiza una operación de división con los valores de los objetos R(B) y R(C), y asigna el resto a un nuevo objeto que será asignado al registro R(A).		
NOT	iABC	R(A) R(B) R(C)	R(A) := !R(B)
	Si el objeto en R(B) es un booleano, asigna el valor complementario a un nuevo objeto en el registro R(A).		
EQ	iABC	R(A) R(B) R(C)	R(A) := R(B)==R(C)
	Asigna a R(A) un nuevo objeto booleano con valor verdadero si la se cumple que los objetos en R(B) y R(C) son iguales, en caso contrario asigna un nuevo booleano con valor a false.		
LT	iABC	R(A) R(B) R(C)	R(A) := R(B)<R(C)
	Asigna a R(A) un nuevo objeto booleano con valor verdadero si la se cumple que el objeto en R(B) es menor estricto que el de R(C) son iguales, en caso contrario asigna un nuevo booleano con valor a false.		

LE	iABC	R(A) R(B) R(C)	R(A) := R(B)<=R(C)
	Asigna a R(A) un nuevo objeto booleano con valor verdadero si la se cumple que el objeto en R(B) es menor o igual que el de R(C) son iguales, en caso contrario asigna un nuevo booleano con valor a false.		
SUSPEND	iNone	X X X	
	Indica que se acaba un paso de programa para la tarea actuar, y suspende la ejecución de esta.		
ATOM_START	iNone	X X X	
	Indica el comienzo de una sección atómica. No tiene ningún efecto.		
ATOM_END	iNone	X X X	
	Indica el final de una sección atómica. No tiene ningún efecto.		
END	iNone		
	Termina la ejecución del programa una vez finalice el paso de programa actual.		
RAND	IABC	R(A) INT(B) INT(C)	R(A):=RAND(B,C)
	Asigna al registro R(A) un nuevo objeto de tipo numerico con un valor aleatorio entre INT(B) e INT(C).		

Anexo B: OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**				**x**
ODS 2. **Hambre cero.**				**x**
ODS 3. **Salud y bienestar.**				**x**
ODS 4. **Educación de calidad.**		**x**		
ODS 5. **Igualdad de género.**				**x**
ODS 6. **Agua limpia y saneamiento.**				**x**
ODS 7. **Energía asequible y no contaminante.**				**x**
ODS 8. **Trabajo decente y crecimiento económico.**			**x**	
ODS 9. **Industria, innovación e infraestructuras.**		**x**		
ODS 10. **Reducción de las desigualdades**.				**x**
ODS 11. **Ciudades y comunidades sostenibles**.				**x**
ODS 12. **Producción y consumo responsables.**				**x**
ODS 13. **Acción por el clima.**				**x**
ODS 14. **Vida submarina.**				**x**
ODS 15. **Vida de ecosistemas terrestres.**				**x**
ODS 16. **Paz, justicia e instituciones sólidas.**				**x**
ODS 17. **Alianzas para lograr objetivos.**				**x**

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

Los objetivos de desarrollo sostenible (ODS) que se pueden relacionar con este proyecto son principalmente el objetivo 4, educación de calidad, y el 9, Industría, innovación e infrastructuras. Así mismo, se considera que, en menor medida, el presente trabajo puede relacionarse con el objetivo 8, Trabajo decente y crecimiento económico.

El presente trabajo presenta una herramienta para el análisis de sistemas concurrentes. Esta herramienta puede ayudar no solo a la industria del *software* verificando los programas, protocolos y sistemas que produce, también puede ayudar a analizar ciertos tipos de infraestructuras que presenten una problemática concurrente, como puede ser la gestión del tráfico, la gestión de recursos, etc. Por ejemplo, es posible modelar la interacción entre los semáforos de una carretera y los coches, y con el uso de esta herramienta ver si se provocan *deadlocks* en el tráfico que se traducen a atascos. Así mismo, esta herramienta proporciona una base sobre la que llevar una futura investigación de herramientas y técnicas de verificación para programas concurrentes concurrentes. Es por todo ello que este trabajo se relaciona con el objetivo de Industria, innovación e infraestructuras.

En relación con el objetivo de una educación de calidad, es objetivo secundario de este proyecto que el proceso de construcción de este intérprete sirva de ejemplo para explicar la construcción de intérpretes y del tratamiento de los lenguajes de programación. Así mismo, la herramienta desarrollada puede servir para ilustrar el comportamiento y problemática de los lenguajes y sistemas concurrentes.

Los fallos en el software acarrean un gasto innecesario en recursos humanos, materiales y económicos. Contar con herramientas de verificación, como la implementada en este proyecto, pueden ayudar a reducir la cantidad de estos fallos. Al reducir estos fallos, se ahorran recursos que de otra forman se hubiesen perdido si se hubiesen dado. De esta forma se colabora con el objetivo del crecimiento económico.